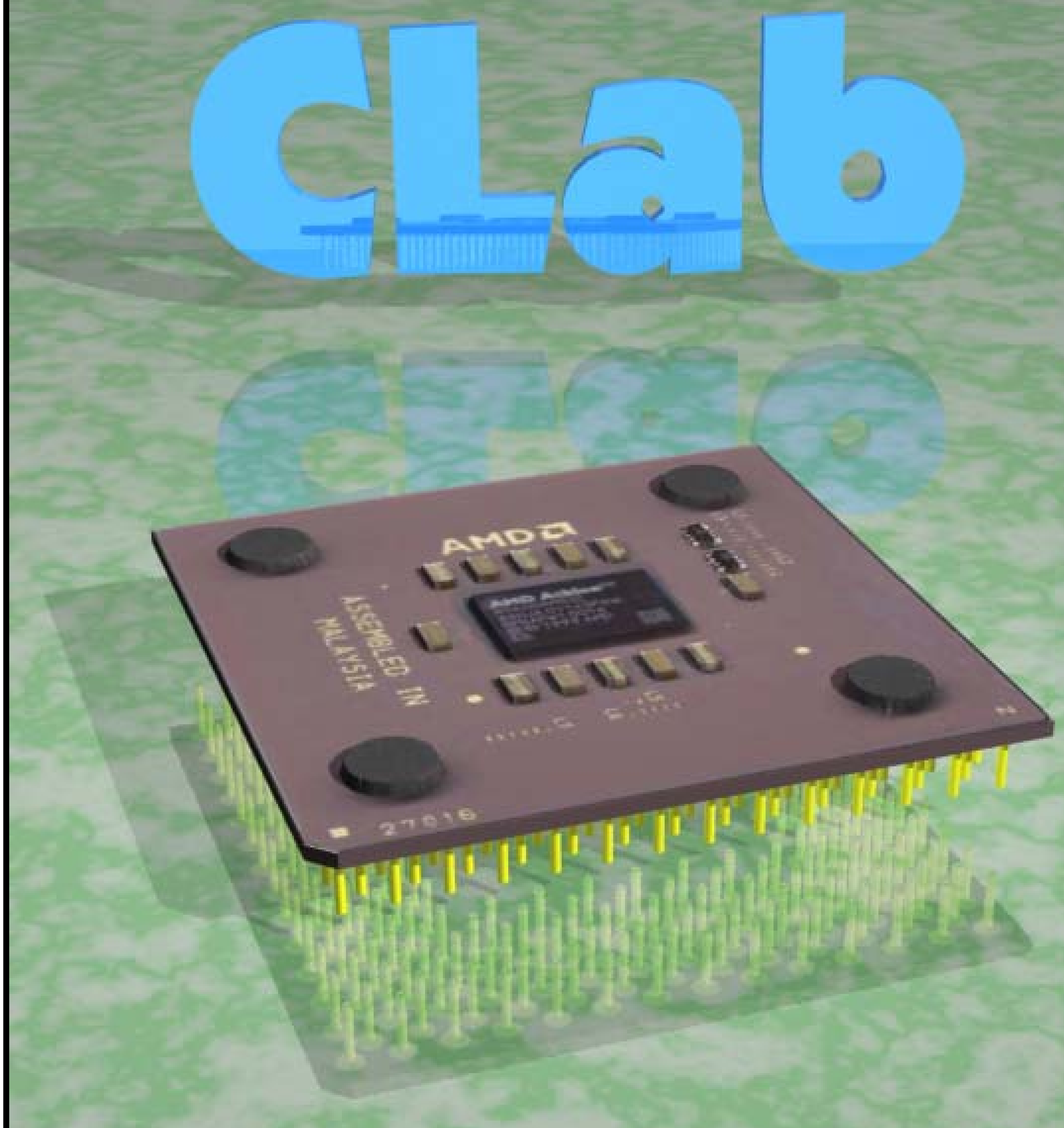


Roman Starkov

Davies Laing & Dick College
London 2002

Analysis

1. Introduction

I am a Computer Science student, and I know from my own experience that it may be hard to grasp at some concepts and ideas. I have noticed that people find some topics in Computing very difficult to understand. When it came to choosing my A-level project I remembered that. Computing is quite a different subject from others. It describes a lot of dynamic processes unlike A-level Maths, and involves a lot of completely new concepts unlike, for example, English or A-level Physics. Practical demonstrations are a well-known teaching technique, and have proven to be very useful. But it is a lot easier to demonstrate, say, attraction between charged objects than a memory read operation. The reason seems to be that Physics has been taught for a lot longer than Computing, and so while most schools have enough equipment in their physics labs, hardly any schools have appropriate software equipment in their computing labs. They have word-processing and program development software, but apart from that only an occasional program to demonstrate a concept or a technique. So I though that I could contribute to that area and develop a system to assist students in learning Computing, and hopefully provide some useful hands-on experience in the field.

The most obious users for my system is one of my Computing teachers and his students. I assume that as an experienced teacher he will know exactly what he wants such a system to do. Although students will also use the system, they are hardly likely to have any idea as to what they want the system to do, apart from it being simple and easy to understand. So I will concentrate on my teacher as the main user.

2. Investigation

As the main user of my system will be my Computer Science teacher, I have had an interview with him. The goal of the interview was to find out the requirements for the new system. Here is an approximate log of the interview:

– *I am interested in creating a computer system which would aid students in learning the principles of computers. I am sure there are a lot of things that could be done. Can you think of any programs you might be interested in?*
– There are quite a few simulations that could be useful. Starting from showing what arrays are and how data can be sorted to simulating a whole virtual computer in work.

– *Which ones do you think would be most useful to you?*
– The more complicated the topic is, the more we need a good illustration of how things happen. Probably internal structure of a computer is one of the more complicated topics, at the same time being a vital part of the syllabus.

– *Do you mean you need a program that would show the role of different parts in a computer and how they communicate with each other to do useful work?*
– That's right.

– *OK. Let me think about it. How interactive do you think the simulation should be? It could be just a set of video clips, or it could be based on some sort of initial data.*
– In theory the best way I can think of doing this is to create an assembly language interpreter which would show every step required to get a program executed. In that way a student would be able, for instance, to investigate what happens to the buses, or 'feel' the von Neumann machine principle, etc.

– *The amount of work required to create such a system is quite high. Do you think it's worth it?*
– An assembly language interpreter by itself might not be worth it because although it is a profound topic in computer science, in the exam you usually get no more than 5% of the marks on assembly language. But if there were a system which would help students really understand what computers are made of and why – that would be very useful in my opinion.

– *Another issue is how detailed you want the simulation to be. A very detailed system might resemble a real PC well but be too complicated to be of any use in teaching. On the other hand, if you have a very simple system you might not be able to show as much as you want.*
– I agree with you. I would like to be able to teach both GCSE and A-level students, so it would be great if there was a way of switching complexity levels. These levels should contain as much information as the syllabuses do.

– *How do you teach assembly language at the moment?*
– I have to show everything on paper or on whiteboard. It is quite hard to show how a program works if you can't run it.

– *What areas of assembly language cause most problems to the students?*

– Different addressing modes definitely cause most problems, especially indirect addressing. It would be nice if some sort of animations were available. Other aspects, such as shifts, take some time for the students to grasp at.

– *Let's think about the diagrams. Which ones do you want for GCSE and A-level?*
– Well, probably a general diagram showing all computer components, and a diagram for the internal structure of the CPU for A-level. I understand we would be able to see everything animated on the diagram, e.g. how an instruction is fetched, or how an interrupt is processed?
– *Yes, that is the idea. By the way, you would probably need more than two detail levels because, for example, you don't want to see anything but CPU, RAM and buses when you tell people about buses.*
– That's true.

– *Now let's concentrate on the assembly language. Examiners use different names for instructions and registers than those that are more common in Windows-targeted programming languages. I believe you would prefer examiners' names?*
– Yes. And I think that anybody who understands assembly language would learn new names quickly if they need to.

– *The instruction set would probably include arithmetic, logic and branching instructions. Anything else?*
– Yes. I would like to have a kind of screen which would be able to output either text or graphics. Also primitive interrupt simulation, e.g. when a key is pressed, would be a good idea.

– *How many registers would you like to have?*
– Accumulator, plus general purpose registers. Four or five general purpose registers would do. I would prefer arithmetic commands to always have one of the operands being the accumulator. Also I would like to see special purpose registers such as Program Counter, Stack Pointer, Flags Register.

– *Do you want to have different register sizes, e.g. ah, al being 8 bits each and eax being 32 bits, while still being the same register?*
– I think that would confuse students without teaching them anything. It is not such an important part of assembly language to go into trouble of allowing that, and it is very CPU-specific.

– *So what register size, or bus width, do you think you need?*
– In the majority of things I will use your program for I will not need more than 8 bits address and 8 bits data registers. But if you find it easy you might allow for 16 bits address and data buses. What I definitely want is each memory location to be 8 bits in size.

– *Do you want to have any segment registers? What addressing modes do you want to see in the program?*
– Immediate, direct and indirect addressing. You might allow for indexed addressing, but it is of no major importance. In addition I would like to have relative jumps and calls. Segment registers… Well, as long as they would not make everything very confusing they would be OK. Probably I would prefer those as modifiers, not true

registers, e.g. when you access vs:[010h] then you always access location 10h in video memory segment. But I want to have a simple mode where everything would be in one big ‘segment’, including code, stack, data and video memory, to show the von Neumann principle.

– *How are we going to store a program? We can either store and interpret it in text form, which would be the easiest way of doing it, where you have code separate from everything else. Alternatively we could write a program in a special window and then compile it into machine code. The latter is definitely more complicated to implement but otherwise I see no big difference.*
– I think there is a big difference. The latter would give students an even better idea of how a byte can be interpreted in a lot of different ways, even as an instruction. Also that way we will have an assembler – showing how it works would be wonderful.

– *OK. Is there anything else you have on your mind?*
– Well, you must understand that good user interface is crucial. It should not look complicated because that would repel students. It is also very important that things are intuitive – a student who can’t do something would probably not persist and just go away.
– *I will have that in mind.*

3. Requirements

3.1. General requirements

This section describes general functionality which the system is expected to have.

- File operations: Load/Save project
- Writing a program: Program editor (with syntax highlighting if possible)
- Running a program: Run/Stop, Step
- Debugging: Breakpoints, watch variable/register values

Teaching assembly language should be one of requirements for the project, but not the only one. It should be possible to use this system to teach internal workings of a computer, and the CPU in particular, by showing how different components communicate with each other and letting the user see exactly what is going on.

3.2. Simulation requirements

Code

The user will develop the program in assembly language and compile it in machine codes. The virtual CPU will then be able to execute the program.

Memory

The most appropriate memory model will be flat – everything, including code, stack, data and video memory, in one long array at different offsets. This model is the best one to show von Neumann machine, and it is the simplest one to write code for. Memory should be at least 4 kilobytes long.

Storing 16-bit data

All 16-bit data will be stored in RAM in big-endian format, that is the most significant byte will be stored at the lower memory address.

Stack

Stack should start somewhere around the middle of RAM. It should grow towards the end of RAM in order to prevent it overwriting user code.

3.3. Detail Levels

As the system will be aimed at different user levels, it should be possible to change program accordingly. The system should allow the user to choose between Basic mode and A-level mode. For example, in the Basic mode it should be possible for the user to write and debug a program without ever knowing that it is compiled at all.

3.4. Instruction set

- Data movement – move data between registers and memory, stack operations
- Arithmetic – addition, subtraction, multiplication etc.

- Logic – standard operations such as AND, OR, shifts, etc.
- Flags – operations concerning the FLAGS register.
- Branching – conditional/unconditional jumps, subroutine calls, halt.
- Input/output – IO port operations, interrupt operations.

3.5. Registers

- General purpose – used as temporary storage for data
- Special purpose – registers with special meaning, e.g. stack pointer
- Internal registers – cannot be used in programs, but can be viewed

3.6. Addressing modes

- Immediate – target is a number
- Register – target is a register
- Memory – target is a memory cell

3.7. Extra functionality

- Animation of different addressing modes
- Animation on number representation
- Animation of logical and arithmetic shifts

3.8. Security

This system will not contain any sensitive data, and therefore no security is required.

4. Constraints

4.1. Hardware & Software

The college already has a network of computers set up, so it would be most cost-effective if this system was able to utilise software and hardware that is already in place. Below is a summary of a typical machine set up:

Hardware:

- IBM-compatible machine
- AMD/PIII 400MHz
- 64 Mb RAM
- 500Mb+ free HDD space
- Video card, 1024x768+ resolution

The hardware requirements for this system will be up to 10Mb of disk space and a minimum 1024x768 resolution video card.

Software:

- Windows 2000 / Windows 98
- Visual Basic 6.0

The only software requirement for this system will be a Win32 operating system.

4.2. CS proficiency

Targeted users of this system will be students, and although they will be Computer Science students they will still need an easily understandable program. Therefore, using the program should be as simple as possible, that is, interface should be as intuitive as possible.

4.3. Feasibility

Taking all requirements and constraints into account, it seems that this project is feasible. It is technically feasible (technology exists to implement the solution), economically feasible (enough funds is available to implement it), legally feasible (a licensed copy of VB will be used, and all extra controls that may be used will be either freeware or licensed), operationally feasible (it will be relatively easy to incorporate new teaching methods using this system, and the users should only be happy to transfer to it).

The project should be completed in about 5 months. Because VB is new to me, I cannot estimate very well the amount of time necessary to implement the solution, so I will concentrate on core features at first and then implement optional functionality if any time is available. That way the project will be schedule feasible.

5. Objectives

The main objective of this project is to provide an interactive teaching tool which can be used in Computing lessons to demonstrate computer science concepts and give students some hands-on experience. The system will have to meet three distinct needs. One is to demonstrate operation of internal components of a computer, especially the CPU. Another objective is to provide a fully functional fool-proof and easy-to-use assembly language simulator. The third requirement is to provide a set of interactive simulations (e.g. sorting algorithms). This requirement is optional and depends on how much time will be available.

It is difficult to assess the system from its teaching potential point of view because it is hard to judge how much this program contributed towards a particular student's achievement. Therefore, project's success should be analysed based on its usability. This project should be considered successful if at least 8 out of 10 students will be able to write their first ever assembly language program on their first ever encounter with the system in a 90 minute session with a teacher available to give advice. The system should also withstand at least half an hour of intentional attempts to make it crash in order to make sure some students will not be able to crash it to avoid work.

6. Solution system

This system should be written in one of the high-level programming languages as its main characteristics are:

- Is not very speed-critical
- Intuitive user interface is crucial
- Design time is limited
- Funds are very limited
- Relatively advanced programming techniques are required

Several programming languages are available. Below is a summary of these languages with all their advantages and disadvantages:

Visual Basic 6.0

Advantages:

- Low cost (already available to programmer)
- Rapid application development
- Easy creation of advanced user interface features

Disadvantages:

- Low programming flexibility
- Low efficiency of compiled code
- Programmer has very little experience with it

Delphi 5.0

Advantages:

- Rapid application development
- Easy creation of advanced user interface features
- Programmer has a lot of experience with it
- High programming flexibility
- Good efficiency of compiled code

Disadvantages:

- High cost (license required)
- Executables tend to be big

Microsoft Visual C++

Advantages:

- High programming flexibility
- High efficiency of compiled code
- Comparatively small executables
- Programmer has some experience with it

Disadvantages:

- High cost (license required)
- Takes a lot of time to create good GUI

Borland C++ 5.02

Advantages:

- Very high programming flexibility
- High efficiency of compiled code
- Comparatively small executables

Disadvantages:

- High cost (license required)
- A lot of time required to develop applications
- Difficult to create good GUI
- Programmer has little experience with it

Conclusion

Below is a summary of language correspondence to key features:

Feature (Importance)	Best Worst
– Is not very speed-critical (3)	VC, Borland C, Delphi, VB
– Intuitive user interface is crucial (8)	VB, Delphi, VC, Borland C
– Design time is limited (10)	VB, Delphi, VC, Borland C
– Advanced programming is required (6)	VC, Borland C, Delphi, VB
– Funds are very limited (7)	VB, VC, Delphi, Borland C
– Programmer experience (5)	Delphi, VC, VB, Borland C

If every language was assigned points from 3 to 0 depending on how well they match a requirement, and multiplied by the weight (importance) of the requirement, here are the totals for the languages:

VB:	0x3 + 3x8 + 3x10 + 0x6 + 3x7 + 1x5	= 80
VC:	3x3 + 1x8 + 1x10 + 3x6 + 2x7 + 2x5	= 69
Delphi:	1x3 + 2x8 + 2x10 + 1x6 + 1x7 + 3x5	= 67
Borland C:	2x3 + 0x8 + 0x10 + 2x6 + 0x7 + 0x5	= 18

Therefore, Visual Basic is more suitable for developing this project than any other language considered.

Design

The Imaginary Computer

7. Introduction

This section describes the way the system should work. Every system designed for the end-user is a lot more complicated on the inside than it seems on the outside. The same will be true of CLab. Design section will describe the internal workings of a complex system, so the description will be technical and complicated, with a lot of subtle details.

I do not see how to avoid this complexity and detail, but moreover, I do not see the need to do so. The users will only be aware of a fraction of what will be described in this section.

Note that the system will have several complexity levels, and while at full level the users will be able to interact with a good deal of all this, in the basic mode they will only see a tiny fraction.

8. Central Processing Unit

The CPU in this system is *not* going to process commands. It will only be there to show the users how a real CPU executes a program. Therefore, it doesn't have to be fully functional from electronics point of view, but it should show different structural elements of the CPU and how they interact.

8.1. Architecture

The architecture of this CPU is very loosely based on that of the Z80 processor. The whole CPU has been designed from scratch with its teaching purpose always being the main guideline.

The major difference between this CPU and real modern CPU architectures will be that no steps are taken to optimize and speed up program execution in order to keep the structure as simple as possible.

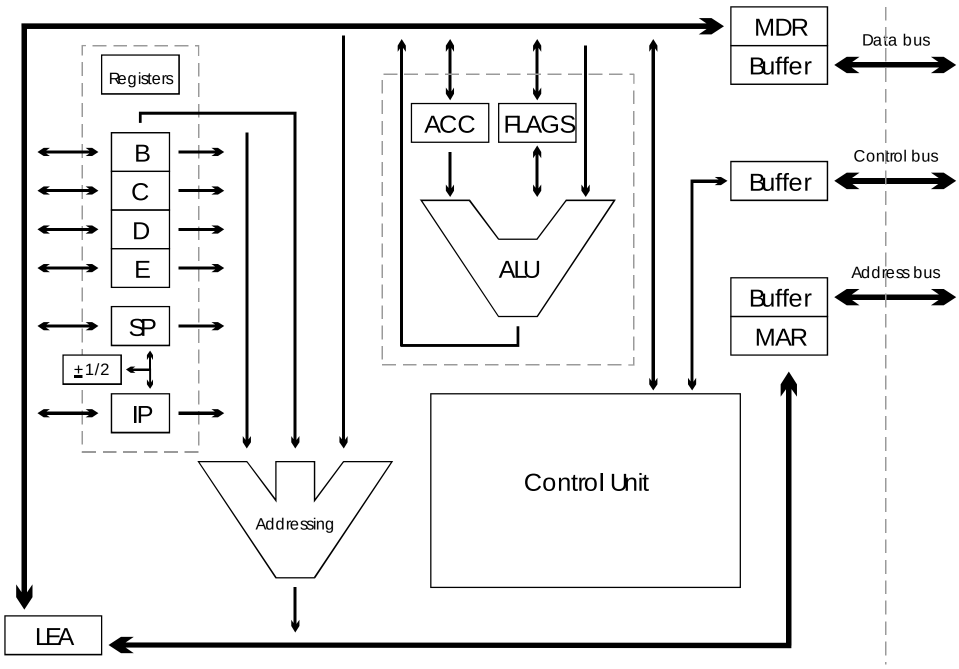


Fig.6.1. CPU diagram

In this diagram, the Control Unit is the main section that does all the "clever" work of the CPU. Its internal structure is hidden because it is too advanced for the level this system is aimed for. Control Unit will run the fetch-decode-execute cycle, coordinating all components around it.

All components of the CPU are connected to the **Internal Data Bus** (IDB). Control Unit uses it to transfer data between itself and components such as registers or the ALU. It also uses it to interact with the world outside the CPU via MDR and Data Bus Buffer, including fetching instructions and port I/O operations.

Internal Address Bus (IAB) is connected to the Control Unit through the **Addressing** component. Addressing component is an adder which, in response to Control Unit's directions, adds together one or more pieces of data that are connected to it in order to obtain a full absolute address for operations with memory. It outputs the resulting address directly to the Internal Address Bus and indirectly (at Control Unit's discretion) to the Address Bus outside the CPU.

Internal Control Bus is not shown in the diagram as it would overload it. There is a network of signals going to each component in the CPU which tell components when and what to do. For example, there are such signals as Select, Read and Write going to the Registers component. They allow Control Unit to choose on which register to operate, and then whether that register should read the data from the IDB or write what it contains to the IDB. Another example would be a signal going to the Addressing component telling it that it should add, say, B register and value on the IDB together.

ALU stands for the Arithmetic Logic Unit, and, as the name suggests, it is responsible for all arithmetic and bitwise calculations. It can do following arithmetic operations: add, subtract, multiply, divide, take remainder from division, increment/decrement, change sign, arithmetic shift. Among its bitwise operations are: AND, OR, XOR, NOT, and logical shifts. If it takes two operands, one always has to be the Accumulator register.

The **±1/2** component is a separate adder which can add or subtract 1 or 2 from `IP` and `SP` registers. This is a very frequent operation, so it would not be wise to do it through the ALU.

The **MDR** and **MAR** registers (Memory Data Register and Memory Address Register) hold the data that has arrived from the respective internal or external bus, and can also output that data to either of the buses. These registers are used in operations with external buses. Some operations also use them as temporary storage.

The **LEA** section (Load Effective Address) is a link between the IAB and IDB. Sometimes, when an effective address has been calculated by the Addressing section, the address needs to be used rather than transferred to the external Address Bus. The LEA section is responsible for transferring the address from the IAB to IDB. As soon as the address is on IDB, it can be saved in any register or used otherwise.

8.1.1. Memory models

To simplify the project, only one memory model will be supported – flat memory model. Segmented memory model, although mentioned in A-level course, is not really studied at all, so it would be better to reduce program functionality with no negative effect on the users, while greatly increasing program simplicity, which is very important for the users. In flat model RAM is seen as a single long block of data. In this project RAM will be 64 Kb long. Anything can be stored anywhere. Though, some memory areas will have a special meaning. Program execution will begin at address 0, so the beginning of RAM will act like a code segment. Stack pointer will be initialised to address 6000h and will grow upwards, so area of RAM from 6000h onwards will act like a stack segment. Video memory will be initialised to addresses E000h-EFFFh. And finally, the last 256 bytes of RAM will be used for interrupt vectors table. All this will be discussed in detail further in this document.

8.1.2. Addressing modes

The CPU will support three basic addressing modes, one of which will be split into four sub-modes. The modes are listed below:

- **Immediate**. The operand is a numerical constant, 8 or 16 bits.
- **Register**. The operand is a register.
- **Memory**. The operand is a memory cell. Is subdivided into following:
    - **Direct (Immediate)**
    - **Indirect (via Register)**
    - **Indirect (via Immediate)**
    - **Indexed**

Direct (Immediate) mode points at a memory cell whose address is specified as a 16-bit immediate constant. The absolute address calculated is the value of the immediate constant.

Indirect (via Register) mode points at a memory cell whose address is held in a register. The absolute address calculated is the value held in the register.

Indirect (via Immediate) mode points at a memory cell whose address is held in another memory cell (intermediary cell). The address of the intermediary cell is specified as a 16-bit immediate constant. The CPU loads contents of the intermediary cell and uses it as the final absolute address.

Indexed mode allows the program to specify an expression to calculate the required address. The Addressing section of the CPU can add together B register (offset), a 16-bit immediate constant (another offset) and a general purpose register multiplied by 1, 2 or 4 in order to calculate the effective address.

8.1.3. Registers

As shown in fig.1, there will be several registers which the CPU will utilise to execute a program. They are divided into sections by type, and their meaning is described below.

General purpose

These registers are used as a temporary storage for data. They are all 16 bits wide.

Name	Description
A	Accumulator. This register is involved in operations with the ALU (it has to be one of the operands in two-operand operations). Some operations are shorter when they use the accumulator.

B	General purpose register. Also used as base register in memory addressing.
C	General purpose register.
D	General purpose register.
E	General purpose register.

Special purpose

These registers have a special meaning, e.g. to show where the next program instruction is. They can't be used as sources or destinations in most operations. Though they will be used by some operations indirectly. They are all 16 bits wide.

Name	Description
PC	Program counter. The next instruction to be executed starts at address stored in PC. Can be modified indirectly by jump instructions.
SP	Stack Pointer. The next empty cell in stack is at address stored in SP. Used indirectly by stack operations.
FLAGS	Flags register. Contains information about current CPU state or the result of some operations. Flags map can be found below.

Flags

In the FLAGS register, the low-order byte will contain the basic flags, and the high-order byte will contain auxiliary flags which will help students learn but will not be used by the system otherwise.

Low-order byte High-order byte

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
z	s	o	c	i				n	p						

z – zero flag. Is set every time the result of a calculation is zero.
s – sign flag. Is always equal to the high-order bit of the result.
o – overflow flag. Set when operation causes a carry into OR out of high-order bit.
c – carry flag. Set when operations cause a carry out of high-order bit.
i – interrupts flag. When this flag is cleared, processor ignores all interrupts

n – negative flag. Is set when result is negative. n = s and not z
p – positive flag. Is set when result is positive. p =not(s or z) =not s and not z

Internal registers

These registers cannot be used in programs, but can be viewed. They will be contained inside the CPU, and an assembly language programmer will not need to know that they exist at all. But they are crucial to understand how the CPU works.

All registers under the thick line are contained within the Control Unit, all others are outside the Control Unit but inside the CPU.

Name	Description
MAR	Memory Address Register. Acts as a link between the internal CPU address bus and the external address bus.
MDR	Memory Data Register. Acts as a link between the internal CPU data bus and the external data bus.
CIR	Current Instruction Buffer. Accumulates instruction to be decoded and executed over several fetch cycles (for multi-byte instructions).

`IS`	Interrupt Status. Contains a bit for every hardware interrupt, indicating whether that interrupt is pending or not.

8.2. Input/output

8.2.1. Interrupts

The CPU will support software and hardware interrupts. Each external device will be connected to an Interrupt Request line (IRQ) and to an Interrupt Acknowledge line (INTA), which are separate for every device. When a device wants to send an interrupt request, it will send a signal down its IRQ. If the CPU decides to process it, it will send an INTA signal back. The project will allow for up to 16 external devices, so there will be 16 IRQ lines, for interrupt request numbers 0 to 15.

Interrupt requests will have different priorities. IRQ0 will have the highest priority, and IRQ15 – the lowest priority.

Interrupt request number (IRQ) and actual interrupt number that the IRQ generates (INT) are two different concepts, and do not always match in real computers. For example, in IBM-compatible computers IRQ0 (timer) generates INT8, and IRQ1 (keyboard data ready) generates INT9. But because at A-level these concepts are not differentiated, they will be hidden from the user in this project, and IRQ0 to IRQ15 will always invoke interrupts 0 to 15 respectively.

There will be a special register in the CPU to store pending interrupts. The register will be called `IS` for Interrupt Status. It will have one bit for every one of the 16 hardware interrupts. If a bit is set, then a respective interrupt has been requested and accepted by the CPU.

When the CPU receives an interrupt request, it will first check whether interrupts are allowed. Programs will be able to allow or disallow hardware interrupts with special instructions. If interrupts are allowed and the corresponding bit in IS is cleared then the CPU will accept the interrupt request by sending INTA signal to the requesting device and set the corresponding bit in IS.

Having completed an instruction, the CPU will check whether any of the IS bits is set to zero. Interrupt 0 will have the highest priority, and interrupt 15 – the lowest. Therefore, the CPU will start checking bits from 0 to 15. If a bit is set, the CPU will initiate the interrupt.

To initiate a hardware interrupt, the CPU will first clear that interrupt's bit in IS. It will then push `PC` and `FLAGS` on stack, clear the Interrupts Flag in `FLAGS` register, and jump to a respective interrupt handler. Interrupt handlers will end with a special instruction, which will pop `FLAGS` and `PC` from stack.

There will be an Interrupt Vector Table of size 256 bytes in RAM which will store effective addresses of interrupt service procedures (ISPs) for respective interrupts. This table will start at address 0FF00h (for interrupt 0) and end at 0FFFEh (for interrupt 127) and contain 16-bit absolute addresses of respective ISPs.

Software interrupts, unlike hardware interrupts, will be called by programs themselves, and cannot be disabled. Programs will invoke them with a special instruction, specifying which interrupt number they want to invoke. It will be possible to invoke any interrupt, from 0 to 127. The procedure for invoking software interrupts will be the same, except for the fact that Interrupts Flag will be left intact by the CPU.

Clearing the “interrupts allowed” flag will *not* cancel pending interrupts. It will only forbid acceptance of new hardware interrupts. As said above, this flag will have no effect on software interrupts.

8.2.2. I/O ports

The CPU will support operations with input/output ports. There will be 256 I/O ports. All devices outside the CPU will communicate with the CPU using input/output ports.

If a program wants to receive data from a device, it will use a special instruction which will cause the CPU to put port number on the Address Bus and send a “port read” signal down the Control Bus. External devices, on receiving this signal, will test the Address Bus to see if their port number was specified. If they decide that they want to respond to this port read operation, they put one word of data on the Data Bus. That is the data that the destination of the instruction will receive.

If a program wants to send data to an external device, it will call another instruction which will cause the CPU to put port number on the Address Bus, the specified data on the Data Bus, and send a “port write” signal down the Control Bus. On receiving this signal, external devices will check if their port number was specified, and take the data from the Data Bus if they decide to.

The computer will be set up in such a way that no two devices use same port numbers. In this case this is not a major concern because the virtual “computer” will be set up by the system developer once, with all ports assigned to devices without clashes, and the user need not be aware of that. See (Peripherals.Architecture) section to read more about external devices and their ports.

8.3. Instruction set

Below is a list of instructions that the CPU will support, with their arguments (operands) and a description of what they do.

Operand types

- `M` memory (any addressing mode),
- `R` register (`A`, `B`, `C`, `D` or `E`),
- `Rn` register (`B`, `C`, `D` or `E`),
- `A` accumulator,
- `I` 16-bit immediate,
- `I8` 8-bit immediate,
- `N` immediate as part of the machine code.

8.3.1. Data movement

These instructions move data between registers and memory. They also include stack operations. None of these modify the `FLAGS` register.

Name	Arguments	Description
`ld`	dest, src	Copies contents of `src` to `dest`. Allowed `dest`/`src` combinations: `R/R`, `R/M`, `R/I`, `M/R`.
`st`	src, dest	Copies contents of `src` to `dest`. Allowed `src`/`dest` combinations: `R/R`, `M/R`, `I/R`, `R/M`.
`push`	src	Copies contents of `src` to `[SP]`, then increments `SP` by 2. `Src` is type `R` or `I`.
`pop`	dest	Decrements `SP` by 2, then copies contents of `[SP]` to `dest`. `Dest` is type `R`.
`pushpc`	-	Pushes `PC` register onto stack, pointing to **after** the `pushpc` instruction.
`pushsp`	-	Pushes `SP` register onto stack. `SP` value before this operation is pushed.
`pushfl`	-	Pushes `FLAGS` register onto stack.
`popsp`	-	Pops `SP` register from stack.
`popfl`	-	Pops `FLAGS` register from stack.
`sp2b`	-	Copies the contents of `SP` register into `B` register. Used to access parameters that are passed on stack quickly.
`lea`	dest, src	Load effective address. Allowed `dest`/`src` combinations: `Rn/M`. Loads the address calculated for `src` into register `dest`.
`xchg`	r1, r2	Swaps values in registers r1 and r2 so that value in r1 goes to r2 and vice versa. r1 and r2 are type `Rgn`.

8.3.2. Arithmetic

These instructions do addition, subtraction, multiplication etc. All of these set the `FLAGS` register (flags `z`, `s`, `o`, `c`; `n`, `p`) according with the result.

Name	Arguments	Description
`add`	addto, addwhat	Adds `addwhat` to `addto`, saves result in `addto`. Allowed `addto`/`addwhat` combinations: `A/I`, `A/M`, `A/R`, `R/A`.
`sub`	subfrom, subwhat	Subtracts `subwhat` from `subfrom`, saves result in `subfrom`. Allowed `subfrom`/`subwhat` combinations: `A/I`, `A/M`, `A/R`, `R/A`.
`adc`	addto, addwhat	Adds `addwhat`, `addto` and carry, saves result in `addto`. Allowed `addto`/`addwhat` combinations: `A/I`, `A/M`, `A/R`, `R/A`.

`sbb`	subfrom, subwhat	Subtracts `subwhat` from `subfrom`, then subtracts carry from the result, saves final result in `subfrom`. Allowed `subfrom`/`subwhat` combinations: `A/I`, `A/M`, `A/R`, `R/A`.
`cmp`	left, right	Compares `left` with `right` and sets flags so that conditional jumps work correctly. E.g. if `left<right` then `JL` will do a jump. The opeartion subtracts `right` from `left` and discards the result. Allowed `left`/`right` combinations: `A/I`, `A/M`, `A/R`, `R/A`.
`mul`	arg1, arg2	Multiplies `arg1` by `arg2`. Saves result in `arg1`. Treats values as unsigned integers. Allowed `arg1`/`arg2` combinations: `A/I`, `A/M`, `A/R`, `R/A`.
`div`	num, denom	Divides `num` by `denom`, saves the integer part of the result in `num`. Interprets `num` and `denom` as unsigned integers. Allowed `num`/`denom` combinations: `A/I`, `A/M`, `A/R`, `R/A`.
`imul`	arg1, arg2	Multiplies `arg1` by `arg2`. Saves result in `arg1`. Treats values as signed integers. Allowed `arg1`/`arg2` combinations: `A/I`, `A/M`, `A/R`, `R/A`.
`idiv`	num, denom	Divides `num` by `denom`, saves the integer part of the result in `num`. Interprets `num` and `denom` as signed integers. Allowed `num`/`denom` combinations: `A/I`, `A/M`, `A/R`, `R/A`.
`mod`	num, denom	Divides `num` by `denom`, saves the remainder part of the result in `num`. Interprets `num` and `denom` as unsigned integers. Allowed `num`/`denom` combinations: `A/I`, `A/M`, `A/R`, `R/A`.
`inc`	arg	Increments `arg`, that is adds 1 to it. `Arg` is type `R`.
`dec`	arg	Decrements `arg`, that is subtracts 1 from it. `Arg` is type `R`.
`neg`	arg	Reverses the sign of `arg`. This is equivalent to `not arg`; `inc arg`; but occupies only one byte. `Arg` is type `R`.

8.3.3. Bitwise

Bitwise operations such as AND, OR, shifts, etc. All of them modify the `FLAGS` register (flags `z`, `s`, `c`; `n`, `p`) according with the result.

Name	Arguments	Description
`not`	arg	Bitwise NOT – inverts all bits in `arg`. `Arg` is type `R`.
`and`	arg1, arg2	Bitwise AND. Allowed `arg1`/`arg2` combinations: `A/I`, `A/M`, `A/R`, `R/A`.
`or`	arg1, arg2	Bitwise OR. Allowed `arg1`/`arg2` combinations: `A/I`, `A/M`, `A/R`, `R/A`.
`xor`	arg1, arg2	Bitwise XOR. Allowed `arg1`/`arg2` combinations: `A/I`, `A/M`, `A/R`, `R/A`.
`test`	left, right	Performs a bitwise AND operation on `left` and `right` and sets the flags according with the result. Result itself is discarded. `A/I`, `A/M`, `A/R`, `R/A`.
`lshr`	arg, num	Shifts[3] bits in `arg` by `num` to the right. Low-order bit goes to carry, high-order bit becomes zero. Allowed `arg`/`num` combinations: `A/Rn`, `Rn/N`.
`lshl`	arg, num	Shifts[3] bits in `arg` by `num` to the left. High-order bit goes to carry, low-order bit becomes zero. Allowed `arg`/`num` combinations: `A/Rn`, `Rn/N`.
`ashr`	arg, num	Shifts[3] bits in `arg` by `num` to the right. Low-order bit goes to carry, high-order bit stays the same. Allowed `arg`/`num` combinations: `A/Rn`, `Rn/N`.
`ashl`	arg, num	Entirely equivalent to `lshl`.
`ror`	arg, num	Rotates[3] bits in `arg` by `num` to the left. Low-order bit goes to high-order bit and carry. Allowed `arg`/`num` combinations: `A/Rn`, `Rn/N`.
`rol`	arg, num	Rotates[3] bits in `arg` by `num` to the left. High-order bit goes to low-order bit and carry. Allowed `arg`/`num` combinations: `A/Rn`, `Rn/N`.
`rcr`	arg, num	Rotates[3] bits in `arg` by `num` to the left through carry. Carry goes to high-order bit and low-order bit goes to carry. Allowed `arg`/`num` combinations: `A/Rn`, `Rn/N`.
`rcl`	arg, num	Rotates[3] bits in `arg` by `num` to the left through carry. Carry goes to low-order bit and high-order bit goes to carry. Allowed `arg`/`num` combinations: `A/Rn`, `Rn/N`.
`bswp`	src	Swaps bytes in `src` so that the high-order byte becomes the low-order byte and vice versa. `Src` is type `R`.

8.3.4. Flags

These operations are used to modify the FLAGS register.

Name	Arguments	Description
stz	-	Sets zero flag.
clz	-	Clears zero flag.
stc	-	Sets carry flag.
clc	-	Clears carry flag.
sto	-	Sets overflow flag.
clo	-	Clears overflow flag.
sts	-	Sets sign flag.
cls	-	Clears sign flag.
sti	-	Sets interrupt flag.
cli	-	Clears interrupt flag.

8.3.5. Branching

These are all operations that change execution order. They change IP register (and CS where applicable), so the next instruction to be executed changes as well.

Name	Arguments	Description
jg, jnle	addr	Jumps to addr if z = 0 and s = o. Addr is an absolute address of type M.
jl, jnge	addr	Jumps to addr if s <> o. Addr is an absolute address of type M.
jge, jnl	addr	Jumps to addr if s = o. Addr is an absolute address of type M.
jle, jng	addr	Jumps to addr if z = 1 and s <> o. Addr is an absolute address of type M.
jz, je	addr	Jumps to addr if z = 1. Addr is an absolute address of type M.
jnz, jne	addr	Jumps to addr if z = 0. Addr is an absolute address of type M.
jc	addr	Jumps to addr if c = 1. Addr is an absolute address of type M.
jnc	addr	Jumps to addr if c = 0. Addr is an absolute address of type M.
jo	addr	Jumps to addr if o = 1. Addr is an absolute address of type M.
jno	addr	Jumps to addr if o = 0. Addr is an absolute address of type M.
js	addr	Jumps to addr if s = 1. Addr is an absolute address of type M.
jns	addr	Jumps to addr if s = 0. Addr is an absolute address of type M.
jmp	addr	Unconditional jump. Addr is an absolute address of type M.
call	addr	Pushes IP registers onto stack; then jumps to addr. Addr is an absolute address of type M.
ret	-	Pops data from stack to IP (i.e. does a jump to address on stack)
int	num	Initiates software interrupt num. Num is type I8.
iret	-	Return from interrupts handlers. Pops FLAGS and IP from stack.
halt	-	Brings processor to a halt. In this project this instruction will stop simulation.

8.3.6. Input/output

This section contains operations that send and receive data via input/output ports.

Name	Arguments	Description

`in`	dest, prt	Reads data from port `prt` and places it to `dest`. `Dest`/`prt` can be following combinations: `R/R`, `R/I8`.
`out`	prt, src	Writes data `src` to port `prt`. Allowed `prt`/`src` combinations: `R/R`, `R/I`, `I8/R`, `I8/I`.

8.3.7. Other

Name	Arguments	Description
`nop`	-	No operation. The CPU goes on to fetch next instruction after fetching this one.

8.3.8. Notes

[3] Shifts/rotations with `num` greater than 1 are equivalent to several shifts/rotations by 1. Only 4 low-order bits matter in `num` operand. Therefore, the maximum number of shifts/rotates in one operation is 15 and the minimum is 0.

8.4. Machine codes

When an assembly language program is assembled, it will be converted into machine codes. The CPU will be able to understand only those codes. This section describes the format of machine codes which this CPU will use, including instruction codes, operand formats etc.

8.4.1. Conventions

All instructions will be listed in a table containing the following fields: Instruction, Binary, Hexadecimal, Length and Operands.

Instruction field
This field will contain assembly language instruction with operands. Part of the *Instruction* field will be in bold dark red font – that is the part coded in the first instruction byte. Bright red font will show the part described in extra bytes.

Binary and Operands field
The operands will be shown in bright red font in the *Operands* field. Each bit of the machine code (including the first byte) may be shown as `1`, `0` or a lower-case letter. Letters will be used to show that several different options are available and their meaning is described in the *Operands* field. Also, a combination of upper-case letters can be used to describe a whole byte in Operands. Underscore `_` will separate bits in the same byte, forward slash `/` will separate bytes.

Operand types
Operands shown in the *Instruction* field consist of a mnemonic followed by a list of operand types. Operand types can consist of one to three letters, optionally followed by a number when there are two operands of the same type. Allowed types are:

- **`M`** memory (any addressing mode),
- **`R`** register (`A`, `B`, `C`, `D` or `E`),
- **`Rn`** register (`B`, `C`, `D` or `E`),
- **`A`** accumulator,
- **`I`** 16-bit immediate,
- **`I8`** 8-bit immediate,
- **`N`** immediate as part of the machine code.

Memory addressing
Wherever an operand is a pointer to a memory cell, one or more bytes will be added *to the end* of the whole instruction. They can describe any addressing mode available, and will be shown as type `M` in the *Instruction* field. Nothing will be said about it in the *Operands* field other than to show their presence with the `MEM` word. The structure of those bytes is described in section (Memory Addressing Bytes) below.

8.4.2. Registers

All general purpose registers except the will have a binary ID code associated with them. This code will be used as part of instructions to show that a specific register should be used. The IDs are as follows:

Name	Binary ID
B	000
C	001
D	010
E	011

The accumulator doesn't have an ID associated with it. This is due to the architecture of the CPU – as can be seen in fig.1, the accumulator stands very separate from the other registers – inside the ALU block. Either a separate instruction or a separate bit is required to specify operations on the accumulator.

8.4.3. Memory addressing bytes

In all instructions operating on memory addresses at least one byte will be added to the end of the machine code describing the way the address should be calculated. The CPU will use this first byte to decide which addressing mode it is dealing with. The format of this byte and other bytes if any is described below. In all format descriptions question mark `?` will represent an unused bit, which does not matter and can be set to anything. Square brackets `[]` will enclose optional bytes. For a description of memory addressing modes, refer to (CPU.Architecture) section.

Direct (via Immediate)
The length will always be 3 bytes. The format is as follows:

```
00_?????_0 /Y1 /Y2
```

Bytes `Y1Y2` are the offset constant.

Indirect (via Immediate)
The length will always be 3 bytes. The format is as follows:

```
00_?????_1 /Y1 /Y2
```

Bytes `Y1Y2` are the offset constant.

Indirect (via Register)
The length will always be 1 byte. The format is as follows:

```
01_????_rr
```

`RR` is the binary ID code for the register used.

Indexed
The length will vary from 1 to 3 bytes. The format is as follows:

```
1_b_p_?_mm_rr [/Y1 /Y2]
```

`B` specifies whether to use the base register (`1` means use). `P` specifies if any offset bytes are present (`1` means they are). `RR` specifies the binary ID for the offset register. `MM` is the scaling factor – `00` if `RR` should not be taken into account, `01` if it is to be multiplied by 1, `10` – by 2, `11` – by 4. `Y1Y2` is the offset constant.

8.4.4. Instructions

Instruction	Binary	Hex	Len	Operands
adc A,I	1000 0111	87	3	Y1/Y2. Y1Y2 is the constant I.
adc A,M	1000 1000	88	2-5	MEM.
adc A,R	1000 0110	86	2	????_0_a_xx. xx is ID for R. If a=1 then R is accumulator.
adc R,A	1000 0110	86	2	????_1_a_xx. xx is ID for R. If a=1 then R is accumulator.
add A,I	1000 0001	81	3	Y1/Y2. Y1Y2 is the constant I.
add A,M	1000 0010	82	2-5	MEM.
add A,R	1000 0000	80	2	????_0_a_xx. xx is ID for R. If a=1 then R is accumulator.
add R,A	1000 0000	80	2	????_1_a_xx. xx is ID for R. If a=1 then R is accumulator.
and A,I	1011 0001	B1	3	Y1/Y2. Y1Y2 is the constant I.
and A,M	1011 0010	B2	2-5	MEM.
and A,R	1011 0000	B0	2	????_0_a_xx. xx is ID for R. If a=1 then R is accumulator.
and R,A	1011 0000	B0	2	????_1_a_xx. xx is ID for R. If a=1 then R is accumulator.
ashl A,Rn	1100 0010	C2	2	0_xx_?_????. xx is ID for Rn.
ashl Rn,N	1100 0010	C2	2	1_xx_a_nnnn. xx is ID for Rn. If a=1 then Rn is accumulator. nnnn is the constant N.
ashr A,Rn	1100 0011	C3	2	0_xx_?_????. xx is ID for Rn.
ashr Rn,N	1100 0011	C3	2	1_xx_a_nnnn. xx is ID for Rn. If a=1 then Rn is accumulator. nnnn is the constant N.
bswp A	1001 1111	9F	1	-
bswp Rn	1101 11xx	DC-DF	1	xx is ID for Rn.
call M	0111 0001	71	2-4	MEM.
clc	0101 0111	57	1	-
cli	0111 0111	77	1	-
clo	0110 0101	65	1	-
cls	0110 0111	67	1	-
clz	0101 0101	55	1	-
cmp A,I	1000 1101	8D	3	Y1/Y2. Y1Y2 is the constant I.
cmp A,M	1000 1110	8E	2-5	MEM.
cmp A,R	1000 1100	8C	2	????_0_a_xx. xx is ID for R. If a=1 then R is accumulator.
cmp R,A	1000 1100	8C	2	????_1_a_xx. xx is ID for R. If a=1 then R is accumulator.
dec A	1010 1101	AD	1	-
dec Rn	1010 01xx	A4-A7	1	xx is ID for Rn
div A,I	1001 0100	94	3	Y1/Y2. Y1Y2 is the constant I.
div A,M	1001 0101	95	2-5	MEM.
div A,R	1001 0011	93	2	????_0_a_xx. xx is ID for R. If a=1 then R is accumulator.
div R,A	1001 0011	93	2	????_1_a_xx. xx is ID for R. If a=1 then R is accumulator.
halt	0111 0101	75	1	-
idiv A,I	1001 1010	9A	3	Y1/Y2. Y1Y2 is the constant I.
idiv A,M	1001 1011	9B	2-5	MEM.
idiv A,R	1001 1001	99	2	????_0_a_xx. xx is ID for R. If a=1 then R is accumulator.
idiv R,A	1001 1001	99	2	????_1_a_xx. xx is ID for R. If a=1 then R is accumulator.
imul A,I	1001 0111	97	3	Y1/Y2. Y1Y2 is the constant I.
imul A,M	1001 1000	98	2-5	MEM.
imul A,R	1001 0110	96	2	????_0_a_xx. xx is ID for R. If a=1 then R is accumulator.
imul R,A	1001 0110	96	2	????_1_a_xx. xx is ID for R. If a=1 then R is accumulator.
in A,I8	1110 0101	E5	2	Y. Y is the constant I8.
in Rg1,Rg2	1110 0100	E4	2	00_a1_a2_xx_yy. xx is ID for Rg1, yy is ID for Rg2. a1=1 makes Rg1 accumulator, a2=1 – Rg2.
in Rgn,I8	1101 10xx	D8-DB	2	Y. Y is the constant I8. xx is ID for Rgn
inc A	1010 1100	AC	1	-
inc Rn	1010 00xx	A0-A3	1	xx is ID for Rn
int I8	0111 0100	74	2	Y. Y is the constant I8.

Instruction	Binary	Hex	Len	Operands
iret	0111 0011	73	1	-
jc M	0101 0010	52	2-4	MEM.
jg M	0100 0000	40	2-4	MEM.
jge M	0100 0011	43	2-4	MEM.
jl M	0100 0010	42	2-4	MEM.
jle M	0100 0001	41	2-4	MEM.
jmp M	0111 0000	70	2-4	MEM.
jnc M	0101 0011	53	2-4	MEM.
jno M	0110 0001	61	2-4	MEM.
jns M	0110 0011	63	2-4	MEM.
jnz M	0101 0001	51	2-4	MEM.
jo M	0110 0000	60	2-4	MEM.
js M	0110 0010	62	2-4	MEM.
jz M	0101 0000	50	2-4	MEM.
ld A,I	0010 0100	24	3	Y1/Y2. Y1Y2 is the constant I.
ld M,R	0010 0111	27	3-5	?????_axx/MEM. xx is ID for R. A makes R accumulator.
ld R,M	0010 0110	26	3-5	?????_axx/MEM. xx is ID for R. A makes R accumulator.
ld R1,R2	0010 0101	25	2	a1_a2_?xx_?yy. xx is binary ID for R1, yy – for R2, a1 makes R1 accumulator, a2 – R2.
ld Rn,I	0010 00xx	20-23	3	Y1/Y2. xx is binary ID for Rn, Y1Y2 is the constant I.
lea A,M	0000 1100	0C	2-4	MEM.
lea Rn,M	0000 10xx	08-0B	2-4	MEM. xx is ID for Rgn.
lshl A,Rn	1100 0000	C0	2	0_xx_?_????. xx is ID for Rn.
lshl Rn,N	1100 0000	C0	2	1_xx_a_nnnn. xx is ID for Rn. If a=1 then Rn is accumulator. nnnn is the constant N.
lshr A,Rn	1100 0001	C1	2	0_xx_?_????. xx is ID for Rn.
lshr Rn,N	1100 0001	C1	2	1_xx_a_nnnn. xx is ID for Rn. If a=1 then Rn is accumulator. nnnn is the constant N.
mod A,I	1001 1101	9D	3	Y1/Y2. Y1Y2 is the constant I.
mod A,M	1001 1110	9E	2-5	MEM.
mod A,R	1001 1100	9C	2	????_0_a_xx. xx is ID for R. If a=1 then R is accumulator.
mod R,A	1001 1100	9C	2	????_1_a_xx. xx is ID for R. If a=1 then R is accumulator.
mul A,I	1001 0001	91	3	Y1/Y2. Y1Y2 is the constant I.
mul A,M	1001 0010	92	2-5	MEM.
mul A,R	1001 0000	90	2	????_0_a_xx. xx is ID for R. If a=1 then R is accumulator.
mul R,A	1001 0000	90	2	????_1_a_xx. xx is ID for R. If a=1 then R is accumulator.
neg A	1010 1110	AE	1	-
neg Rn	1010 10xx	A8-AB	1	xx is ID for Rn
nop	1000 1111	8F	1	-
not A	1010 1111	AF	1	-
not Rn	1011 11xx	BC-BF	1	xx is ID for Rn
or A,I	1011 0100	B4	3	Y1/Y2. Y1Y2 is the constant I.
or A,M	1011 0101	B5	2-5	MEM.
or A,R	1011 0011	B3	2	????_0_a_xx. xx is ID for R. If a=1 then R is accumulator.
or R,A	1011 0011	B3	2	????_1_a_xx. xx is ID for R. If a=1 then R is accumulator.
out A,I	1101 0101	D5	3	Y1/Y2. Y1Y2 is the constant I.
out I8,A	1101 0110	D6	2	Y. Y is the constant I8.
out I8,I	1101 0111	D7	4	Y1/Y2/Y3. Y1 is the constant I8. Y2Y3 is the constant I.
out I8,Rgn	1101 00xx	D0-D3	2	Y. Y is the constant I8. xx is ID for Rgn
out Rg1,Rg2	1101 0100	D4	2	00_a1_a2_xx_yy. xx is ID for Rg1, yy is ID for Rg2. a1=1 makes Rg1 accumulator, a2=1 – Rg2.
out Rgn,I	1110 00xx	E0-E3	3	Y1/Y2. Y1Y2 is the constant I. xx is ID for Rgn
pop A	0001 0001	11	1	-
pop Rn	0000 01xx	04-07	1	xx is ID for Rn.
popfl	0001 0111	17	1	-

Instruction	Binary	Hex	Len	Operands
popsp	0001 0110	16	1	-
push A	0001 0000	10	1	-
push I	0001 0010	12	3	Y. Y is the constant I.
push Rn	0000 00xx	00-03	1	xx is ID for Rn.
pushfl	0001 0101	15	1	-
pushpc	0001 0011	13	1	-
pushsp	0001 0100	14	1	-
rcl A,Rn	1100 0110	C6	2	0_xx_?_????. xx is ID for Rn.
rcl Rn,N	1100 0110	C6	2	1_xx_a_nnnn. xx is ID for Rn. If a=1 then Rn is accumulator. nnnn is the constant N.
rcr A,Rn	1100 0111	C7	2	0_xx_?_????. xx is ID for Rn.
rcr Rn,N	1100 0111	C7	2	1_xx_a_nnnn. xx is ID for Rn. If a=1 then Rn is accumulator. nnnn is the constant N.
ret	0111 0010	72	1	-
rol A,Rn	1100 0100	C4	2	0_xx_?_????. xx is ID for Rn.
rol Rn,N	1100 0100	C4	2	1_xx_a_nnnn. xx is ID for Rn. If a=1 then Rn is accumulator. nnnn is the constant N.
ror A,Rn	1100 0101	C5	2	0_xx_?_????. xx is ID for Rn.
ror Rn,N	1100 0101	C5	2	1_xx_a_nnnn. xx is ID for Rn. If a=1 then Rn is accumulator. nnnn is the constant N.
sbb A,I	1000 1010	8A	3	Y1/Y2. Y1Y2 is the constant I.
sbb A,M	1000 1011	8B	2-5	MEM.
sbb A,R	1000 1001	89	2	????_0_a_xx. xx is ID for R. If a=1 then R is accumulator.
sbb R,A	1000 1001	89	2	????_1_a_xx. xx is ID for R. If a=1 then R is accumulator.
sp2b	0000 1111	0F	1	-
st I,A	0011 0100	34	3	Y1/Y2. Y1Y2 is the constant I.
st I,Rn	0011 00xx	30-33	3	Y1/Y2. xx is binary ID for Rn, Y1Y2 is the constant I.
st M,R	0011 0110	36	3-5	?????_axx/MEM. xx is ID for R. A makes R accumulator.
st R,M	0011 0111	37	3-5	?????_axx/MEM. xx is ID for R. A makes R accumulator.
st R2,R1	0011 0101	35	2	a1_a2_?xx_?yy. xx is binary ID for R1, yy – for R2, a1 makes R1 accumulator, a2 – R2.
stc	0101 0110	56	1	-
sti	0111 0110	76	1	-
sto	0110 0100	64	1	-
sts	0110 0110	66	1	-
stz	0101 0100	54	1	-
sub A,I	1000 0100	84	3	Y1/Y2. Y1Y2 is the constant I.
sub A,M	1000 0101	85	2-5	MEM.
sub A,R	1000 0011	83	2	????_0_a_xx. xx is ID for R. If a=1 then R is accumulator.
sub R,A	1000 0011	83	2	????_1_a_xx. xx is ID for R. If a=1 then R is accumulator.
test A,I	1011 1010	BA	3	Y1/Y2. Y1Y2 is the constant I.
test A,M	1011 1011	BB	2-5	MEM.
test A,R	1011 1001	B9	2	????_0_a_xx. xx is ID for R. If a=1 then R is accumulator.
test R,A	1011 1001	B9	2	????_1_a_xx. xx is ID for R. If a=1 then R is accumulator.
xchg A,Rn	1111 00xx	F0-F3	1	xx is ID for Rgn.
xchg b,c	1111 0101	F5	1	-
xchg b,d	1111 0110	F6	1	-
xchg b,e	1111 0111	F7	1	-
xchg c,d	1110 0110	E6	1	-
xchg c,e	1110 0111	E7	1	-
xchg d,e	1111 0100	F4	1	-
xor A,I	1011 0111	B7	3	Y1/Y2. Y1Y2 is the constant I.
xor A,M	1011 1000	B8	2-5	MEM.
xor A,R	1011 0110	B6	2	????_0_a_xx. xx is ID for R. If a=1 then R is accumulator.
xor R,A	1011 0110	B6	2	????_1_a_xx. xx is ID for R. If a=1 then R is accumulator.

8.4.5. Instructions allocation

Below is a table showing allocation of instruction codes to different instructions.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	push b	push c	push d	push e	pop b	pop c	pop d	pop e	lea b,M	lea c,M	lea d,M	lea e,M	lea a,M			sp2b
10	push a	pop a	push I	pushpc	pushsp	pushfl	popsp	popfl								
20	ld b,I	ld c,I	ld d,I	ld e,I	ld a,I	ld R,R	ld R,M	ld M,R								
30	st I,b	st I,c	st I,d	st I,e	st I,a	st R,R	st M,R	st R,M								
40	jg M	jle M	jl M	jge M												
50	jz M	jnz M	jc M	jnc M	stz	clz	stc	clc								
60	jo M	jno M	js M	jns M	sto	clo	sts	cls								
70	jmp M	call M	ret	iret	int I8	halt	sti	cli								
80	add a,RN add RN,a	add a,I	add a,M	sub a,RN sub RN,a	sub a,I	sub a,M	adc a,RN adc RN,a	adc a,I	adc a,M	sbb a,RN sbb RN,a	sbb a,I	sbb a,M	cmp a,RN cmp RN,a	cmp a,I	cmp a,M	nop
90	mul a,RN mul RN,a	mul a,I	mul a,M	div a,RN div RN,a	div a,I	div a,M	imul a,RN imul RN,a	imul a,I	imul a,M	idiv a,RN idiv RN,a	idiv a,I	idiv a,M	mod a,RN mod RN,a	mod a,I	mod a,M	bswp a
A0	inc b	inc c	inc d	inc e	dec b	dec c	dec d	dec e	neg b	neg c	neg d	neg e	inc a	dec a	neg a	not a
B0	and a,RN and RN,a	and a,I	and a,M	or a,RN or RN,a	or a,I	or a,M	xor a,RN xor RN,a	xor a,I	xor a,M	test a,RN test RN,a	test a,I	test a,M	not b	not c	not d	not e
C0	lshl a,RN lshl RN,N	lshr a,RN lshr RN,N	ashl a,RN ashl RN,N	ashr a,RN ashr RN,N	rol a,RN rol RN,N	ror a,RN ror RN,N	rcl a,RN rcl RN,N	rcr a,RN rcr RN,N								
D0	out I8,b	out I8,c	out I8,d	out I8,e	out RN,RN	out a,I	out I8,a	out I8,I	in b,I8	in c,I8	in d,I8	in e,I8	bswp b	bswp c	bswp d	bswp e
E0	out b,I	out c,I	out d,I	out e,I	in RN,RN	in a,I8	xchg c,d xchg d,c	xchg c,e xchg e,c								
F0	xchg a,b	xchg a,c	xchg a,d	xchg a,e	xchg d,e xchg e,d	xchg b,c xchg c,b	xchg b,d xchg d,b	xchg b,e xchg e,b								

Legend:

Param	Description
R	Any register (B, C, D, E)
RN	Any register (A, B, C, D, E)
I	16-bit immediate constant
I8	8-bit immediate constant
M	Memory addressing

Color	Category	Color	Category	Color	Category
	Data movement		Flags		Other
	Arithmetic		Branching		
	Bitwise		Input/output		

9. Peripherals

A computer consists of the CPU at its centre, main memory and peripheral devices surrounding them. As the purpose of this system is not only teaching people assembly language but also what different components of a computer do, the system should show some devices and how they interact with the CPU.

9.1. Computer structure

The structure of the computer being simulated is *very* simplified. No system devices are shown at all, and the ones that are shown are those that a typical user would be aware of plus a component linking them to the CPU known as *I/O controller*.

The structure of the computer is shown in fig.2.

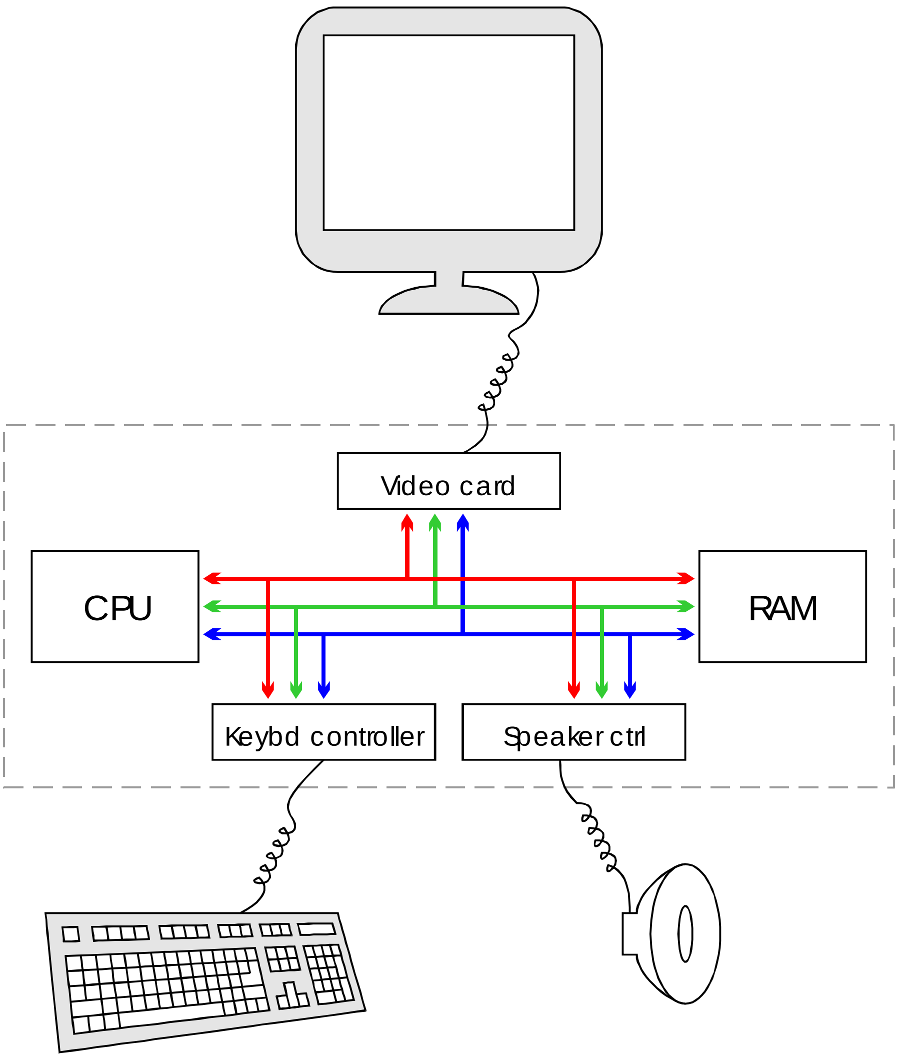


Fig.7.1. Computer structure

Components shown are: CPU, buses, RAM, video card and screen, keyboard controller and keyboard, speaker controller and speaker.

9.2. CPU

The CPU in the diagram is the Central Processing Unit. A whole section (CPU) was devoted to describing how it works, so there is nothing else to discuss here.

9.3. Buses

The imaginary computer will use buses as communication lines between the CPU, RAM and external devices. There will be three buses, just as in a generic computer:

- Data Bus
- Address Bus
- Control Bus

Buses are relatively simple devices, and because this is technical documentation, it is assumed that the way buses work need not be described. So buses won't be discussed any further here.

9.4. RAM

RAM (Random Access Memory) is the place where the program being executed and its data are stored. Memory organisation is discussed in detail in the (CPU.Architecture) section.

Memory is connected to data, address and control buses. The sole purpose of the RAM "device" is to provide the CPU with a place to store the data it needs. Therefore, RAM supports only two functions: memory read and memory write.

Read

When the CPU wants to read data from memory, it will put the address of the first byte to be read on the address bus, and then send a Memory Read signal down the Control Bus. On receiving this signal, the RAM device will read the address from the address bus and put the two bytes starting at the specified address on the Data Bus.

Write

When the CPU wants to write data to memory, it will put the address of the memory location where the data is to be written on the address bus, the actual data to be written on the data bus, and then send a Memory Write signal. On receiving this signal, the RAM device will read both buses and write the data from the data bus to the specified address.

9.5. Video controller

Video card is the component linking the monitor and the CPU. Video card will be assigned several input/output ports to show how the CPU would normally work with external devices.

The image currently displayed on the screen will be stored in RAM in an area 4096 bytes long. Video controller will have an internal register to store the pointer to the first byte of video memory, which will be `E000h` by default. The user will be able to change this register easily, thus allowing for such techniques as page switching.

To give user control as to when a picture is formed and ready to be displayed, video controller will have a mode in which it does not reflect changes to RAM until explicitly told to do so. See below for further detail.

Screen modes

There will be several screen modes supported by the video card. The only limitation imposed on it is that the whole video memory should fit into 4096 bytes. The following screen modes will be available:

`01h`: Monochrome text; 1 byte per char; 40x15 characters
Every byte represents one character's ASCII code.

`02h`: Color text; 2 bytes per char; 40x15 characters
Every two bytes represent one character's ASCII code and color. The first byte in the pair is the character's ASCII code, the second one – its colour. The color byte format is: `LRGB lrgb`, where `R`, `G` and `B` are Red, Green and Blue components respectively, `L` is a brightness bit, uppercase means background color, lowercase – text color.

`03h`: Monochrome graphics; 1 bit per pixel; 208x156 pixels
Every byte describes eight pixels. If a bit is set, color seen will be white; otherwise – black.

`04h`: 16 color graphics; 4 bits per pixel; 104x78 pixels
Every byte describes two pixels. The format is: `LRGB`, where `R`, `G` and `B` are Red, Green and Blue components respectively, `L` is a brightness bit.

`05h`: 256 color graphics; 8 bits per pixel; 74x55 pixels; paletted
Every byte describes one pixel. The color that is seen on screen will be taken from a palette array inside the video controller memory which is 256x3 bytes long. That is the palette memory, which stores three bytes (RGB) for every color in this mode.

`07h`: 16M color graphics; 24 bits per pixel; 42x32 pixels
Every three bytes describe one pixel. The format is, `RGB` where `R`, `G` and `B` are *bytes* describing respective colors.

To switch between different modes the programmer will write a word with mode number to a specific port, described below.

Input/Output ports

Screen mode port `50h`

Writing screen mode number to this port will cause the video controller to switch screen modes. If it receives any other word apart from valid screen mode numbers, it will ignore it. The changes will be reflected immediately, even in manual refresh mode.

Reading from this port will cause the video controller to return its current screen mode.

Palette port `51h`

To change an entry in the palette array, programs will write two words to this port. The first one will contain palette entry number in the low-order byte and the red component in the high-order byte. The second word will contain green and blue components in low- and high-order bytes respectively. Note that once sent to the video controller, palette cannot be read from it. Also, palette only influences images in screen mode `05h`.

Memory port `52h`

Writing to this port will change the offset to video memory buffer in RAM. The changes will be reflected immediately. That is, even in manual refresh mode the screen will be updated to reflect changes to video memory.

If a program reads from this port, it will receive current pointer to video memory.

Refresh port `54h`

Writing 0 to this port will disable auto screen refresh, so changes to video RAM will only be reflected when the programmer wants to. Writing 1 will enable auto screen refresh, so the screen will be updated every once in a while. Writing anything else will force the screen to be refreshed.

Reading from this port will return either 0 or 1 to indicate the state of auto refresh.

9.6. Keyboard controller

Keyboard controller will operate through I/O ports. The user will be able to get pending keys via an input port. Keyboard controller will also send a specific interrupt every time a key is pressed.

Key In port `60h`

The programmer will read from this port to get the pending key code. If no key is pending, keyboard controller will return `0FFFFh`. As soon as a pending key is read once, keyboard controller will forget about that key. Only one key can be pressed at one time, and only Key Down events will be recognised. There will be no way of determining whether a key is still down. This should not be a problem unless someone decides to write a game in this system. Considering that the interpreter will most probably be not fast enough for a game, advanced keyboard features should not be required.

Every time a key is pressed, the keyboard controller will request intterupt 1 (IRQ1). If accepted by the CPU, this IRQ will cause interrupt 1. If not accepted by the CPU, controller will keep sending requests and ignoring all key inputs until the request is successful.

Keyboard scancodes

A scancode is the code returned by the keyboard controller when a given key is pressed. The table of scancodes for this keyboard controller is given below.

Scancode	Hex	Key
0	00	A
1	01	B
2	02	C
3	03	D
4	04	E
5	05	F
6	06	G
7	07	H
8	08	I
9	09	J
10	0A	K
11	0B	L
12	0C	M
13	0D	N
14	0E	O
15	0F	P
16	10	Q
17	11	R

Scancode	Hex	Key
18	12	S
19	13	T
20	14	U
21	15	V
22	16	W
23	17	X
24	18	Y
25	19	Z
26	1A	.
27	1B	Enter
28	1C	Spacebar
29	1D	=
30	1E	0
31	1F	1
32	20	2
33	21	3
34	22	4
35	23	5

Scancode	Hex	Key
36	24	6
37	25	7
38	26	8
39	27	9
40	28	Numpad .
41	29	/
42	2A	*
43	2B	-
44	2C	+
45	2D	Left
46	2E	Right
47	2F	Up
48	30	Down
49	31	Circle
50	32	Square
51	33	Triangle

9.7. Speaker

Speaker can be used as a means of giving signals to the user easily. It will be much easier and faster to switch speaker state than output something on the screen. Speaker will be operated through a single port – `80h`.

Speaker port `80h`

Writing `0` to this port will set speaker to the low state. Writing `1` will set speaker to the high state. Writing any other number will set speaker frequency to `20/65536*W` where `W` is the word sent to this port.

Reading from this port will return the last word written to this port.

10. Assembly language

Generally, the syntax of assembly language in this project should be as similar to the one used in the exams as possible. But because not one examining board is consistent even with its own past papers, this syntax will be only approximately like that in exams or textbooks.

10.1. Statements

A *statement* is the smallest unit of division of programs which can be taken out of context and still have a meaning. The whole statement has to be written on one line, and there can only be one statement on every line.

The structure of a typical statement is shown below:

```
[label:] [data | command ] [;comment]
```

Square brackets `[]` enclose elements that are optional, vertical bar `|` indicates that there are two possibilities, and only one can be present. Only spaces or tabs can separate the elements, but there can be as many of those as needed. Empty lines are allowed since there is no element which is not optional. No part of assembly language syntax is case sensitive.

10.2. Labels

A label is a pointer to a part of code which enables the programmer to reference that part in instructions, letting the compiler do all the calculations. Labels can be used to reference code to use with jump and call instructions, or they can be used to reference data in data movement instructions. See (Referencing) for information about how to reference labels in operands.

A label has to start with a letter and can contain letters, numbers and underscores. Every label has to end with a colon `:` followed by at least one space or tab character, and there can be no whitespace between the name of the label and the colon.

10.3. Data declarations

When a program is compiled, compiler generates code for every operation. To tell compiler to insert specific bytes in compiled code programmers can use keywords `db`, `dw` or `ds`. The purpose of these keywords is to reserve some space in the machine code for data. If the programmer declares a label pointing to that data, the data effectively becomes a variable. To see how to use variables as operands see sections (Offset macro) and (Referencing). `db` stands for “declare byte” and reserves 1 byte of space, `dw` – “declare word” and reserves 2 bytes, `ds` – “declare string”, the number of bytes depends on the length of the string, between 0 and 255.

Every data declaration keyword should be followed by initialisation sequence to tell the compiler the initial contents of the data. `db` should be followed by a numerical constant that fits into 8 bits, `dw` – by a numerical constant that fits into 16 bits, `ds` – by a string literal, described below. Alternatively, each of these keywords may be followed by a question mark `?` to indicate uninitialised variable. Compiler will then initialise `db` and `dw` with zeroes, and `ds` – an empty string, reserving no space for it in the code.

String literals are constants, but unlike numerical constants, string literals can be used in only one case – to initialise `ds` data declaration. String literals must be enclosed by quotation marks `"`. Everything between the marks is the value of string literal. Double quotation marks can be used to include quotation marks as part of string literal's value. When compiled, every byte of string literal's value will be copied directly to machine code, as is. Note that, although string literals cannot be used where numerical constants can, variables declared as string constants are completely identical to those declared as numerical constants.

Because the data will be stored together with the code, the programmer will have to make sure that data is not executed accidentally.

10.4. Commands

Commands are basic instructions which the CPU can process, written in a form readable by humans. Every command has the following syntax:

```
opcode [operand1 [, operand2 ] ]
```

`Opcode` is a symbolic form of writing a CPU instruction; it should be one of the opcodes listed in section (CPU.Instruction Set). Operands should be separated by a comma, and there has to be at least one space or tab character between `opcode` and `operand1`. More tabs or spaces can be used between the elements if needed. Operands are described in detail in the next section.

10.5. Operands

Operands can be of the following types:

- Register
- Immediate
- Memory

Register

This operand type corresponds to the Register addressing mode. It can be `A`, `B`, `C`, `D`, `E`. Sometimes only specific registers can be used – it depends on operand. If register is source, data passed to the instruction is the contents of the register. If register is destination, result is written to the register.

Registers can appear as separate operands, or can be used as part of memory operands described below.

Immediate

Immediate operands are numerical constants. Assembly language will support decimal, hexadecimal and binary numbers. It will have to discern whether a constant is a byte or word. The syntax is as follows:

```
integer[h|b]
```

Integer is any combination of digits from 0 to 9 and letters from A to F, but it has to start with a digit. If the number is followed by letter `h`, `integer` will be interpreted as a hexadecimal number. If it is followed by `b`, `integer` will be interpreted as a binary number, and `integer` should consist of 0's and 1's only. If it is not followed by either `h` or `b`, `integer` is interpreted as a decimal number and should contain only digits from 0 to 9. If the resulting constant is greater than `0FFh`, it is always interpreted as a 16-bit constant. If it isn't, interpretation depends on the opcode.

Negative immediate constants are allowed. They will be stored in two's complement format, and the way numbers with the highest bit set are interpreted will depend on the opcode.

Immediate constants can stand as separate operands, or can be used as part of memory operands described below.

Memory

Memory type operands are used for indexed, direct and indirect (register/immediate) addressing. The syntax of each one is described below.

Memory Direct

```
offset
```

The whole construction should be enclosed in square brackets `[]`, and no spaces are allowed between any sections of the construction. `Offset` is a 16-bit immediate constant which specifies absolute memory address. Variable name as an operand has Memory Direct operand type.

Memory Indirect Register

```
register
```

The whole construction should be enclosed in square brackets `[]`, and no spaces are allowed between any sections of the construction. `Register` specifies the register holding absolute memory address, and can be `B`, `C`, `D` or `E`.

Memory Indirect Immediate

```
offset
```

The whole construction should be enclosed in double square brackets `[[ ]]`, and no spaces are allowed between any sections of the construction. `Offset` is a 16-bit immediate constant which specifies absolute memory address. The CPU will first read the two bytes at `[offset]` and then use the value read as the final absolute memory address.

Memory Indexed

`[base+]register[*scale][+offset]`

The whole construction should be enclosed in square brackets `[]`, and no spaces are allowed between any sections of the construction. `Register` specifies the register holding offset, and can be `B`, `C`, `D` or `E`. This offset is multiplied by `scale`, and `offset` (16-bit immediate constant) can be added to act as base offset. This parameter is often specified with the (Offset macro). If this operand is a source, its value is the contents of the two bytes of memory at the address obtained by calculating the expression. If it is a destination, the result is saved in the two bytes of memory at the address obtained by calculating the expression.

10.6. Offset macro

A *macro* will be supported by the assembly language. Programmers will not be able to define new macros, like in C. The built-in macro will be called `offset`. It will take one parameter – a variable name – and return its absolute address as if an immediate constant was specified. For example, assuming that my_var is a variable at offset `200h`, the following code:

```
ld a,offset(my_var)
```

is equivalent to

```
ld a,200h
```

The advantage of using macros is that the address at which a variable is stored depends on the length of all the code preceding that variable. If a programmer uses a constant to specify its address, he will have to manually recalculate the address every time it changes, whereas `offset` macro will do all calculations for the programmer. The net effect of the substitution is that `offset(my_var)` is converted to a 16-bit immediate constant type operand.

10.7. References

A *reference* is a special operand type which is used to refer to labels. Only absolute references will be supported in this assembly language for simplicity.

The idea behind references is the same as that behind `offset` macro – let the compiler calculate the address. Wherever a memory operand is required, variable name can be used to specify the address. Compiler will replace that with `[offset(varname)]` and compile as described above. The net effect after the substitution is that the variable name is converted to a direct memory addressing operand.

10.8. Comments

In every line of code, everything that follows a semi-colon `;` is completely ignored by the compiler together with the semi-colon. Every line which is empty or consists of only spaces or tabs is ignored as well.

Design

The Real Program

11. Interface

This section describes the interface of the program – the part that the user will see and interact with.

It is probably a good idea to design the whole system in such a way that the user sees the system as a computer, rather than as a program which simulates a computer. This will be taken into account at all stages of interface design.

11.1. Conventions

Numbers

Whenever a number is shown on screen, the user should be able to change its representation between binary, decimal signed/unsigned and hexadecimal. All binary numbers will be followed by a lower-case “b” letter. All hexadecimal numbers will be followed by a lower-case “h” letter and have an extra zero in front of them if they start with a letter.

Window types

There will be three basic window types in the system. To indicate that, every picture of a window in this document will have one of the following icons in the top-left corner:

– windows that make the imaginary computer look like a usual PC does.
– windows that represent internal computer hardware.
– windows that provide operating system and software facilities, such as assembly language Integrated Development Environment (IDE), or stack viewer.

11.2. Main window

Main window will be positioned at the top of the screen. It will provide a menu to load and save the project, run the simulation and change global settings. It will also provide access to every other window in the system.

Main window will contain a toolbar, giving the user quick access to the most frequently used functions, such as open/save project, run/pause/stop simulation, and quick access to system’s windows. Minimizing the main window will minimise all windows of the system, and closing the main window will shut down the system. The main window will look something like this:

Fig.9.2. Main window

The menu will have approximately the following structure:

- File
  - New project
  - Open project
  - Save project
  - Exit
- View
  - A list of all windows in the project, separated into submenus by category
- Simulation
  - Start
  - Pause
  - Stop
  - Reset
  - Simulation speed
- Options
  - Complexity level
  - Number format

11.3. Computer window

This window will simulate the end-user computer. The user will be able to see the computer as if they were sitting in front of a real PC, with a keyboard and a monitor in front of them. They will be able to start or reset a program, and to interface with a running program.

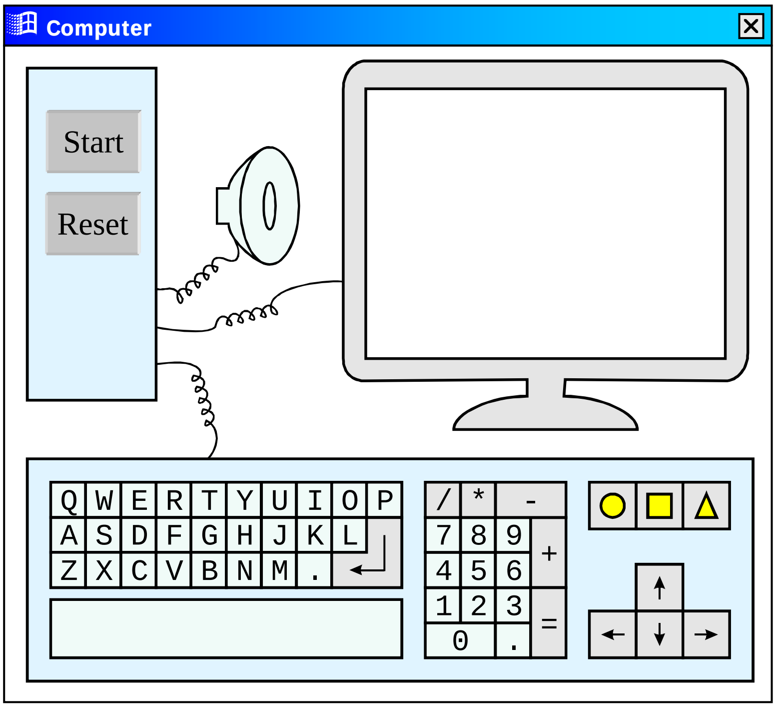


Fig.9.3. Computer window

The user will be able to press any key on the keyboard, which will be sent to the program in an appropriate way. When the user presses a key, that key will become red for a fraction of a second, so that the user knows the key has been pressed.

11.4. Monitor window

This window will duplicate the monitor on the computer window. Apart from the fact that it can be moved around more easily, it will provide scaling facilities and some manual monitor operations such as manual mode switching. The window will look something like this:

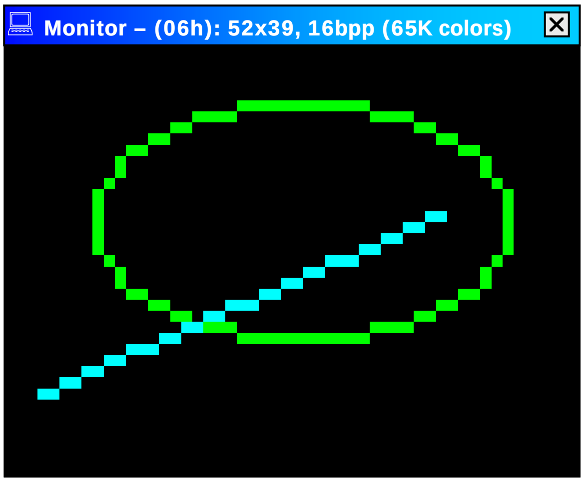


Fig.9.4. The Monitor window

The caption of this window will contain information about the current screen mode. The window will be sizable, so that the user will be able to change the scale if necessary. Right-clicking on the window will bring up a popup menu with the following structure:

- Screen mode
  - 1 – Monochrome text
  - 2 – Color text
  - 3 – Monochrome graphics
  - 4 – 16 color graphics
  - 5 – 256 color graphics (paletted)
  - 6 – 65k color graphics
  - 7 – 16M color graphics
- Scale
  - 200%
  - 400%
  - 800%
- View video RAM (brings up the RAM window)
- View controller (brings up the Video Controller window)

11.5. Keyboard window

This window will duplicate the keyboard in the computer window. The main advantage of having this window separately from the main window is that in this way the keyboard can be placed wherever the user wants to see it.

A possible keyboard window layout is shown below:

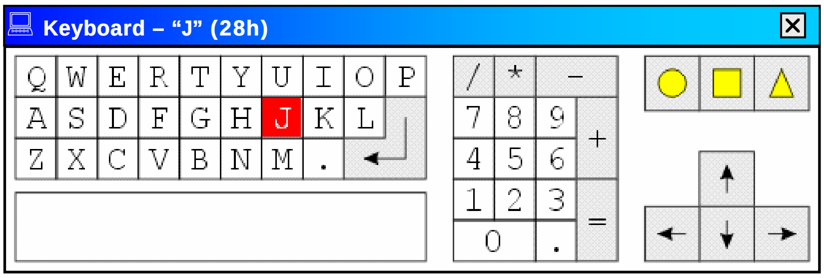


Fig 9.5. Keyboard

The caption of the window will contain information about currently pressed key. That key will also be highlighted with red color. The user will be able to press keys with the left mouse button. Only one key can be pressed at a time.

11.6. Speaker window

This window will just be a tiny window to indicate current state of the speaker. There is a chance that producing real sound through PC speaker will not be possible under VB, so the speaker will flash rapidly to indicate sound.

11.7. RAM window

The RAM window will allow to view and edit the Random Access Memory. It will have to provide such facilities as finding a specific part of memory, jumping to specific addresses (such as current Stack Pointer), show selected byte(s) as different data types (numbers in all representations, disassembled instruction).

The RAM window will consist of two main parts – the memory table and the interpretation panel.

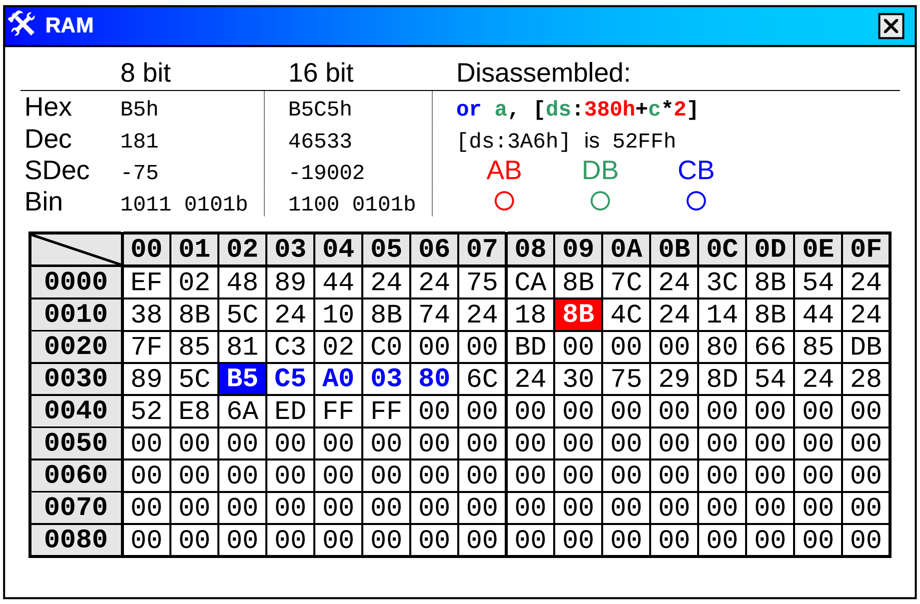


Fig.9.7. RAM window

The memory table will display a part of the RAM contents. It will have a fixed column displaying row address and a fixed row to display byte offset within the row.

The main part will show the byte at the corresponding address. The user will be able to move a cursor around the table. The cell with the cursor will be called the *current* cell and filled with blue. The bytes that would make an instruction starting from the current byte will be shown in blue font. Bytes pointed at by `PC` and `SP` register values will be highlighted with red and green background respectively. The beginning of every new segment will be highlighted with an aqua colored row indicating segment number.

The interpretation panel is the top part of the RAM window. The user will be able to choose to show it or hide it. It will show the interpretation of the current byte (and the one that follows) in different number bases and formats for 8 bit values (16 bit values). It will also disassemble and display the instruction that the CPU would execute if IP was pointing at the current byte.

When the user right-clicks anywhere in the window, a popup menu will appear with the following structure:

- ☑ Show interpretation panel
- Jump to…
    - …Current PC
    - …Current SP
    - Stack
    - Video Memory
- Set…
    - …PC to current byte
    - … SP to current byte

- Fill
- Cut
- Copy
- Paste

- Open stack
- Open variables
- Open disassembler

11.8. Buses

The buses window is intended to show at all times what data is currently going through the buses. It will show the actual data going through the buses and its textual interpretation (where applicable). By right-clicking in the window the user will be able to change number representation (binary, decimal signed/unsigned, hex), and show/hide a special panel which would show a small circle for every wire on every bus, red if a wire is in high logical state and blue for low state. This should help the user to understand the bus concept easily.

When many windows representing actual components are displayed on the screen, they need to be connected with each other through system buses. Rather than showing buses as thick lines going through the screen, which would be very awkward, the buses will not be shown at all. Instead, every window will have a “connector” – a set of three circles labelled "AB", "DB" and "CB" for Address Bus, Data Bus and Control Bus respectively. It will be implied that all AB circles are connected with each other, and so will be DB and CB circles. Every time data is sent across a bus from one device to another, that data will be shown as a tiny window with the data printed in it flying from the sender connector to the connector on the Buses window and then to all destinations it may go to. The data will skip the Buses window if it is closed (it will fly straight to the destination(s)).

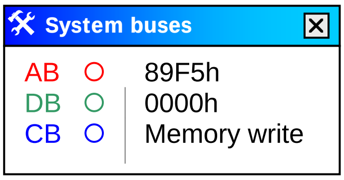


Fig.9.8. Buses window

The left part of the window is the "connector". The right part will display the same text as that in the flying windows. Changing the numerical representation will also affect the representation for the flying windows. The text in this window will not be updated to the new value put by a device until a respective window "lands" on this window's connector.

11.9. CPU window

The CPU window will show internal structure of the CPU and what happens when it executes a program. The window will show the components the CPU consists of and the values of all registers. Fig.9.9 below shows a possible layout of the form.

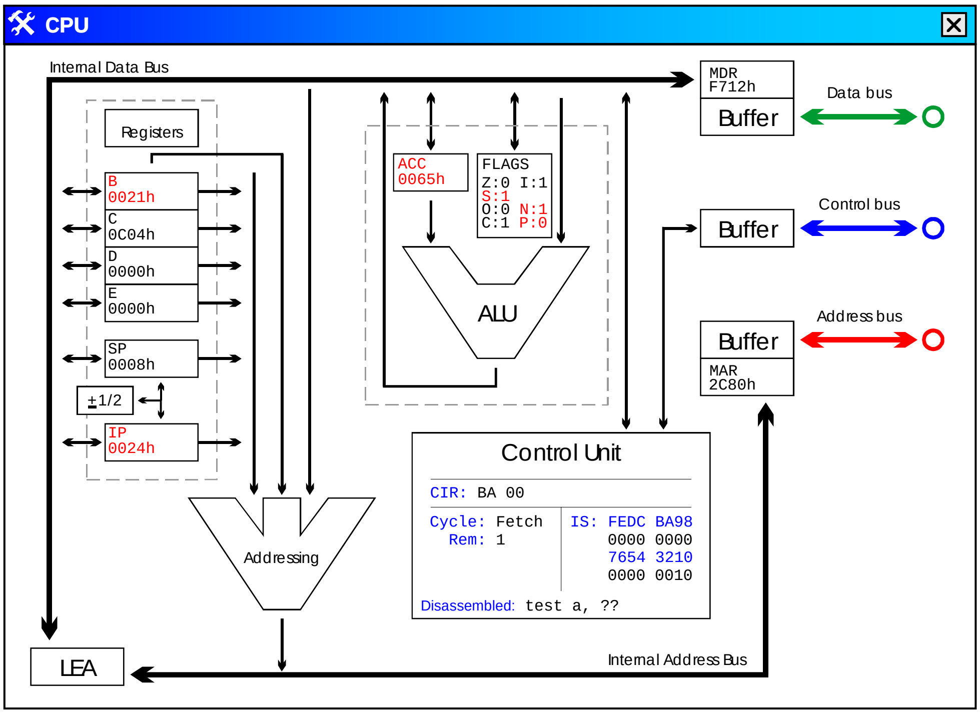


Fig.9.9. CPU window

The purpose of all components that will be shown is described in detail in section CPU.Architecture. It will also be described in user manual.

There are a lot of different buses in the diagram. Some of them are major buses such as the Internal Data and Address Buses, some are minor and have no special name. In any case, the data flowing through them will have to be shown somehow. The most visual way to do that is to show tiny windows with data flowing along the buses, very similar to the way described in section Interface.Buses.

The contents of all registers, including the internal registers and the decoded contents of the `FLAGS` register, will be displayed in the diagram to make it easier to see how the CPU operates. Whenever a register's value changes, it will be displayed in red for one clock cycle. Whenever the Control Unit chooses a register for a read/write operation, that register will be displayed in a thick blue frame.

The top right part of the window will display a "connector", as described in Interface.Buses. In this diagram, more than anywhere else, the user will be able to see how data that goes through connectors is used.

There are a lot of components in the diagram, and if the user sees the diagram of the inside of the CPU for the first time, they will be repelled by its apparent complexity. It is therefore absolutely vital that it is possible to choose which components are to be displayed. By default the simplest view will be selected. The three available views will be as follows:

Basic view:
SP and ±1/2 will be invisible
LEA with arrows leading to it will be invisible
Base register going to addressing will be invisible
Insides of the Control Unit will be invisible
Flags register will be invisible
MDR and MAR registers will be invisible

A-level view:
LEA with arrows leading to it will be invisible
Base register going to addressing will be invisible

Full view:
Everything will be visible

Independent of whether a specific component is turned on or off, the data flow will remain unaffected (with one exception – signals controlling invisible devices will also be invisible). The user will be aware that there is something in the white space where the data comes from, but they will not be able to see that component until they decide they are more familiar with the CPU structure.

If the user right-clicks in the window, a following popup menu will appear:

- Detail level
    - Basic

  - o A-level
  - o Full

- Number representation
  - o Decimal signed
  - o Decimal unsigned
  - o Hexadecimal

- Open registers
- Open flags
- Open control unit
- Open ALU

11.10. Control Unit window

This window will display the status of the control unit to give an interested student a slight idea of how it works. Showing decoded instruction will not be a complicated programming task as the system will have to decode instructions into microinstructions anyway in order to show signals flowing around the CPU.

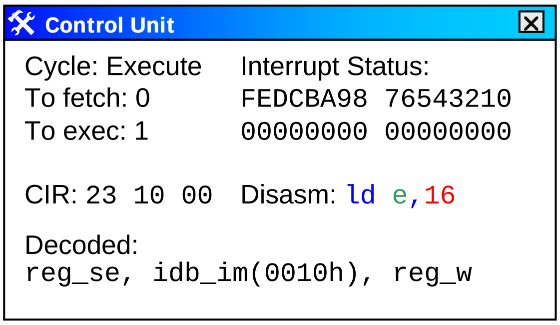


Fig.9.10. Control Unit window

At the top of the window the system will show some internal registers, such as the Current Instruction Register (CIR), and how many fetch or execute cycles are left in the current instruction. It will also show the disassembled instruction to make it easier for the user to understand what the control unit is doing. The Decoded part will show a decoded microprogram for current instruction. It will contain one or more microinstructions, consisting of one or more signals for internal control bus.

There will be no complexity level setting for the control unit window. In the Basic mode the user will not be able to open this window at all, whereas in the A-level mode the user will be warned that CPU windows contains all the required information, and Control Unit window is more advanced than required at A-level.

11.11. ALU window

The Arithmetic Logic Unit (ALU) window will teach students how the ALU works by showing what it does, and in some cases animating the operations. Right-clicking in the window will bring up a menu with an option to change number representation. If an animation for the operation is available, the user will be able to run it from the menu.

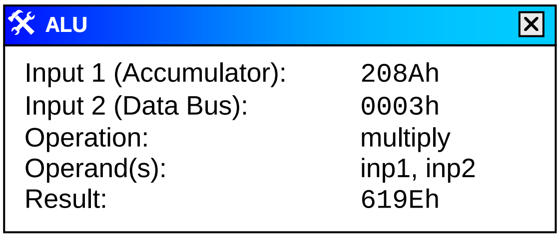


Fig.9.11. ALU window

11.12. Video controller

The video controller window will show details about video controller's state, as well as give user some reminder regarding how to work with the controller. This should help the user understand how devices in general and the video controller in particular work.

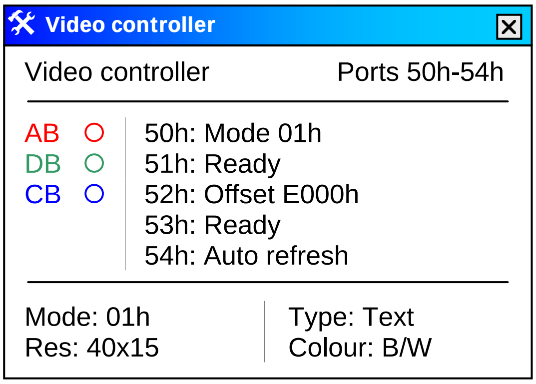


Fig.9.12. Video controller window

As usual for controller windows, this window will have a bus "connector" on it (see section (Interface. Buses)). To the right of the "connector" the window will display the status of its ports. The bottom part will decode current screen mode. To remind the user what each of the ports does, a hint will pop up every time the user hovers over a port number. Port 50h hint will also display a list of all screen modes.

11.13. Keyboard controller window

The keyboard controller window will show the user current keyboard controller status. This should help the user understand how devices in general and the keyboard controller in particular work.

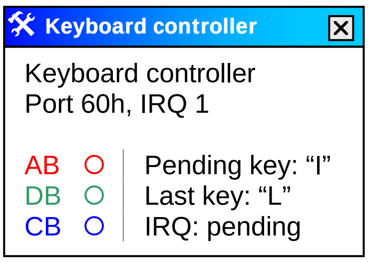


Fig.9.13. Keyboard controller window

The window will contain the system buses "connectors" (see section (Interface. Buses)) as well as status information. The window will tell the user whether interrupt request for last key pressed has been accepted and whether the last key has been sent to the program.

11.14. Speaker controller window

The speaker window will be a small window showing current state of the system speaker and its controller. This window will be the simplest of all device controller windows, and so may be used as a good aid in introducing such concepts as external devices and I/O ports.

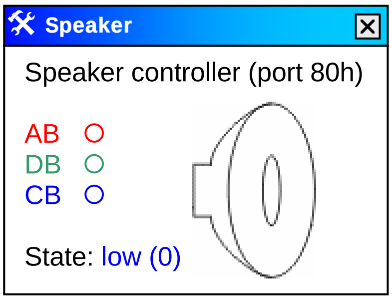


Fig.9.14. Speaker controller window

Speaker state (high or low) will be indicated in a field at the bottom on the window. Also, when speaker is high the picture of the speaker will become red. This way, a buzzing speaker will be blinking rapidly. Right-clicking anywhere in the window will bring up a menu with two options – switch to high state and switch to low state. The AB, DB and CB labels are "connectors" for the respective system buses (see section (Interface.Buses)).

11.15. Code window

The user will usually write and debug programs in the code window. Using this window the user will be able to load and save their programs on disk, compile them into machine codes, load them into memory and run them, and debug them.

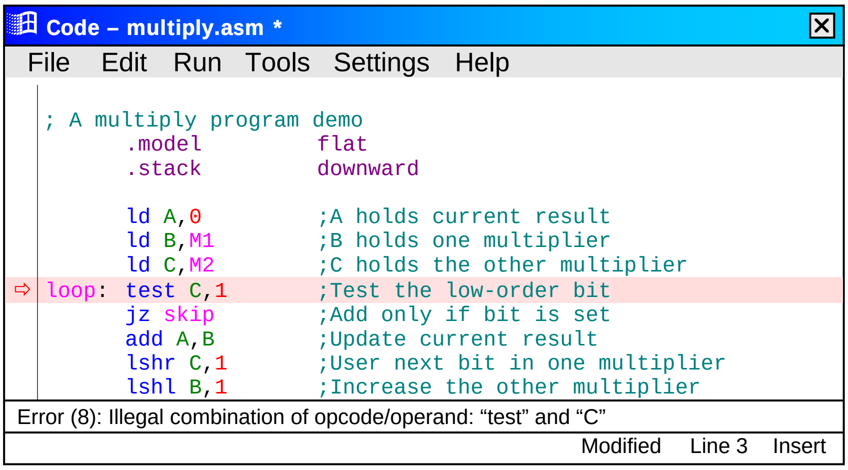


Fig.9.15.1. Code window

The main part of the window will be the code editor. To the left of it will be the so-called “gutter”, where different types of markers will be displayed (e.g. error, breakpoint, current execution point). The bottom part of the window will display messages, such as error and warning messages, generated by syntax check or the compiler. Double-clicking on a message will show the offending line. In the bottom right corner the user will be able to see file status (modified/saved), current line in the editor and the editing mode (insert/overwrite). The code window will support syntax highlighting, which will greatly assist writing programs.

The integrated help system will show the user help on current instruction in the editor if the user presses F1. If the user selects an error/warning message and presses F1, the system will display a relevant help topic. Full help will be available through the window menu.

There will be a menu at the top of the screen, which will have the structure displayed in fig. 9.15.2.

- File
  - New program
  - Open program
  - Save program
  - Save as…
  - Open workspace
  - Save workspace
- Edit
  - Cut
  - Copy
  - Paste
  - Replace
  - Find
  - Find next
- Run
  - Syntax check
  - Compile
  - Run
  - Pause
  - Stop
  - Step
  - Toggle breakpoint
- Tools
  - Trace table
  - Interface
    - Computer
    - Monitor
    - Keyboard
    - Speaker
  - Hardware
    - Speaker
    - Keyboard controller
    - Video controller
    - System buses
    - RAM
    - CPU
    - Control Unit
    - ALU
  - OS/Debug
    - Stack
    - Variables
    - Disassembler
    - Registers
    - Flags
- Help
  - Full help
  - Keyword

Fig.9.15.2. Menu structure for Code window

11.16. Registers subwindow

This window will show all the registers that there are in the CPU. The user will be able to choose whether to show segment and internal registers. The user will also be able to change number representation for the registers.

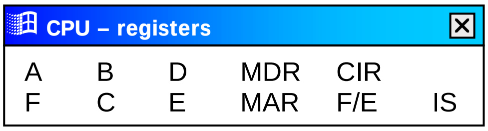


Fig.9.16.1. Registers window – horizontal layout

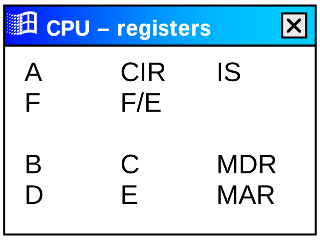


Fig.9.16.2. Registers window
Compact layout

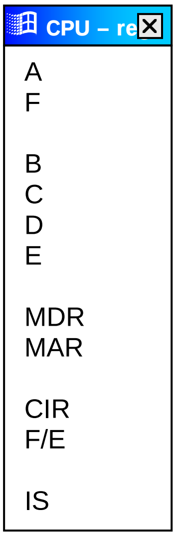


Fig.9.16.3. Registers window
Vertical layout

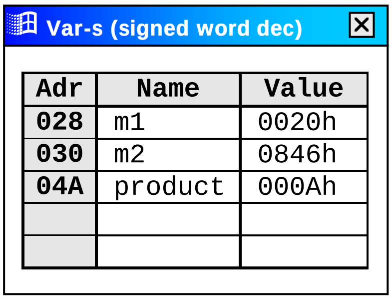
Right-clicking in the registers window will bring up a popup menu which will allow to change layout and number representation. If the user right-clicks on a register, the menu will also allow to change its value. Only registers A-F registers can be edited in this way.

11.17. Variables subwindow

The variables window will display the variables declared in the source code. It will show variable address, variable name and its value. The user will be able to choose number representation and variable size.

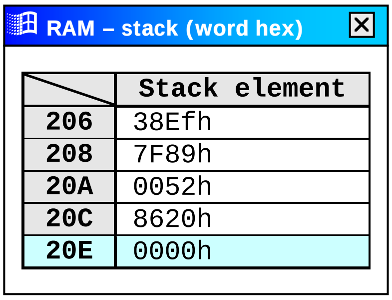
11.18. Stack subwindow

The stack window will display stack contents, decoded into separate stack elements (unlike in the RAM window where everything is just an array of bytes). The user will see element addresses and data stored in those elements. The user will be able to choose different number representation. The element to which SP points at a given moment will be highlighted with aqua background.



Adr	Name	Value
028	m1	0020h
030	m2	0846h
04A	product	000Ah

Fig.9.17. Variables window



	Stack element
206	38Efh
208	7F89h
20A	0052h
20C	8620h
20E	0000h

Fig.9.18. Stack window

12. Modules

This section describes the modules of which the program will consist. A module is a logically and physically separate unit of a program which either contains a set of specific operations (a procedural module) or code related to window interface (a form module).

12.1. Form modules

Interface windows

- fiMain – Main window
- fiComp – Computer window
- fiDisplay – Display window
- fiKeyboard – Keyboard window
- fiSpeaker – Speaker window

Hardware windows

- fhCPU – CPU window
- fhCU – Control Unit window
- fhALU – ALU window
- fhRAM – RAM window
- fhBus – Buses window
- fhVideo – Video controller window
- fhKeyboard – Keyboard controller window
- fhSpeaker – Speaker controller window

OS/Debug windows

- fsCode – Code window
- fsRegs – Registers window
- fsVars – Variables window
- fsStack – Stack window

12.2. Procedural modules

- pWinAPI – all necessary declarations to use WinAPI functions
- pUtils – commonly used procedures not available in standard libraries
- pGlobals – global variable and data type declarations
- pExec – procedures that are responsible for execution of an instruction
- pCompile – assemble a program, syntax check, plus assemble/disassemble a given string procedures
- pIO – input/output procedures to simplify interfacing with device modules, providing such functions as PortRead, PortWrite, RequestInterrupt etc.

13. Data structures and globals

It is very hard to think about data structures in advance because they will be severely affected by exact implementation of algorithms. At this stage many points are not clear enough yet, and will only be decided upon in the process of implementation. What makes laying down data structures even harder is that I am not familiar with Visual Basic, and I have never coded anything more advanced than a simple single-windowed program in it. I could use my experience in Delphi programming to compile a list of all global data structures that I would expect, but most probably they will not be entirely applicable to Visual Basic. I am planning to develop data structures while learning about VB coding principles during implementation stage.

I would expect to declare all global variables and structures in a module called pGlobals. I would have a structured variable to hold all information about current project, another one – to hold execution (simulation) state, and probably one for application-related variables.

14. Assembly process

When an assembly language program is compiled into machine code, a specific algorithm is at work. This section describes how the algorithm will work.

14.1. Conventions and terms

Tokens

A token is the smallest unit of program that is meaningful to the compiler. In this assembler, a token will be a string containing no spaces after pass 1 (see below). Therefore, a token can be:

- Label
- Variable declaration type
- Variable initialisation sequence
- Opcode
- Operand

Note that an operand token can not be divided any further. That is, even a complete indexed memory addressing will be parsed into a single token.

14.2. Passes overview

To assemble a program the algorithm works through it several times; each time is called a *pass*. To fully compile a program, three passes are required.

Pass 1. Tokenize

At this stage the source code is converted into a set of tokens which, unlike such languages as C and Pascal, will still be separated into lines. Internally a tokenized program will be stored as an array of token lines, each being an array of strings containing one token each.

Pass 2. Code generation

Each token line is converted into respective machine codes which are added up and saved in a special byte array. Where a reference is used, the program will remember reference name and backpatch its address in pass 3.

Pass 3. Backpatching

All references are replaced by physical addresses.

14.3. Pass 1. Tokenize

This is a relatively simple pass. The following procedures are carried out on every line of code:

- Remove all comments, unnecessary spaces and empty lines
- Split every line into tokens
- Determine token types

- Analyse token patterns (see Token Patterns)
- Prepare references for pass 2.

14.4. Pass 2. Code generation

At this stage machine codes can be generated. The algorithm will go through all token lines and generate respective machine codes. The following procedures will be carried out:

Generate code for correct instructions
Issue errors for incorrect instructions
Build a list of labels with their physical addresses
Build a list of reference backpatch requests.

14.5. Token patterns

Only specific combinations of tokens will be valid. To simplify referring to token types, the following abbreviations will be used:

- Label — label
- Variable declaration type — vardecl
- Variable initialisation sequence — varinit
- Opcode — opcode
- Operand — operand

Allowed token patterns

label
opcode
opcode operand
opcode operand operand
vardecl varinit

Patterns that can be corrected

If a pattern contains label tokens that follow some non-label tokens then a warning is issued and the label tokens are moved to the beginning of the line.

If a pattern contains a label token (or several label tokens) then the token line is split into two or more token lines, with a token line for every label and the last token line being what remains of the original token line. If it has no other tokens, it is removed completely. This procedure generates no warnings.

If a pattern contains only a vardecl token then a varinit token with no initialization will be added, generating a warning.

All other patterns will generate an error. The error message will say that a specific token combination is invalid.

15. Execution process

This section describes the algorithm that executes a program. The algorithm will effectively simulate a clock tick. Below is a flowchart for the algorithm.

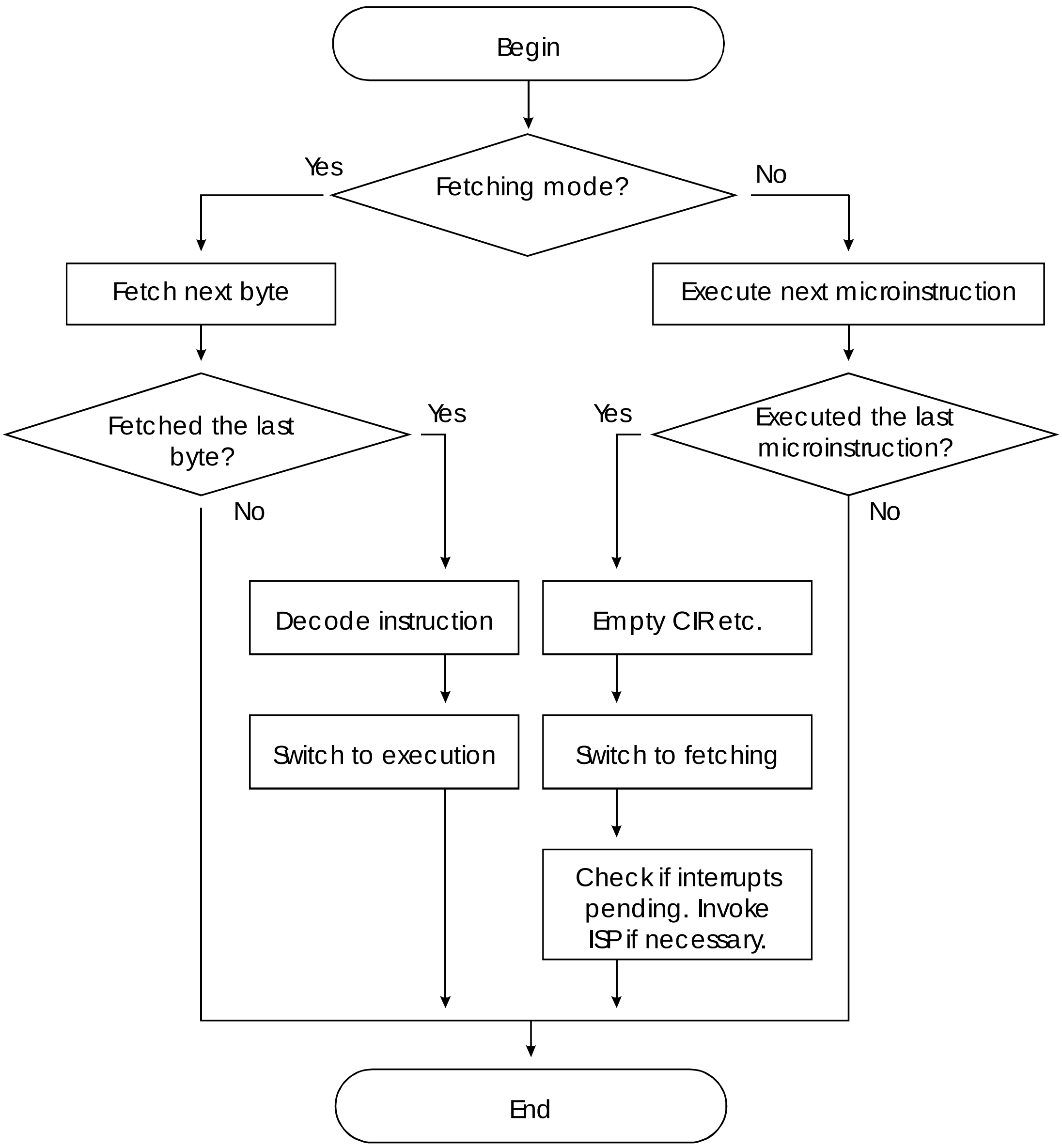


Fig.14.1. Execute instruction algorithm

To invoke an interrupt the system will create a microprogram which will load ISP address from the interrupt vector table, save flags and return address on stack and jump to ISP address. As soon as the microprogram is in the buffer, all that will necessary is just to continue running the clock tick procedure.

16. Sample scenario

The purpose of this section is to analyse whether design is acceptable by going through all stages of development of a simple assembly language program in an A-level computing lesson, and thus clearly showing what this project is going to be like.

Suppose that the goal of the lesson is to write a program which will add the values of two variables, `V1` and `V2` and store the result in a third variable, `R`. Each student is issued with a copy of user manual for A-level students. The teacher has already taught students some theory about assembly language before, so students have a general idea of what they will be doing.

The students start the system. By default they are in GCSE mode. The teacher tells students to switch to A-level mode. At the top of the screen each student sees a window with a menu and many buttons. The teacher tells them to click on the one saying, “Write a program”. A window with a text editor appears. Students can start to code. Let’s concentrate on one student, named James.

James knows that there is a special instruction “`add`” that will instruct the computer to add two numbers. James also knows that normally the result of this operation will be stored in the accumulator. So James types:

```
add V1, V2
st a, R
```

When James clicks “Run”, he gets an error message, with the first line highlighted in red, saying *“Syntax error in operand OR opcode and operand incompatible. Offending operand: V1”*. James has no idea what this means, so he asks the teacher. The teacher notices that many people have the same problem and reminds everybody that one can only add together two registers or a register and a variable, but not two variables. So the values of the variables should be loaded into registers first. James remembers that this can be done with the “`ld`” instruction. So he modifies his code and gets the following:

```
ld b, V1
ld c, V2
add b, c
st a, R
```

James reasons that he should not load anything into the accumulator because the result will be stored there. So he loads the variables into two other registers. When he tries to run this code, he sees that the third line is highlighted, and he gets a similar error message: *“Syntax error in operand OR opcode and operand incompatible. Offending operand: c”*. James seeks for some help from the teacher, but teacher is busy helping someone else and tells James to consult the manual regarding the ADD instruction and the error message James is getting. James reads the description of the ADD instruction, which, among other things, mentions that if one tries to add two operands apart from those that are allowed they will get exactly this error message. So James realises that again, he is trying to add something he is not allowed to add. He notices in an example that two registers will generate an error unless one of them is the accumulator. James decides that he could load V2 into accumulator. He modifies his code accordingly:

```
        ld b, V1
        ld a, V2
        add b, a
        st a, R
```

When James tries to run this program, he gets an error message – for the third time. The message is pointing at the first line and saying, *"Undeclared reference: V1"*. He gets quite annoyed, but soon remembers that he has to declare the variables V1 and V2 in his code. So he adds to the end of the code:

```
V1:     dw 14
V2:     dw 8
R:      dw 0
```

This time the program actually runs. James can see the instruction pointer slowly moving down. But when it reaches the end of the program, it doesn't stop but tries to execute variable declaration, generating an error message saying *"Failed to decode instruction starting with 0Eh"*. James asks the teacher how to make the program stop. The teacher tells him to put a "halt" instruction where he wants the program to end. James updates the program, which by now looks like this:

```
        ld b, V1
        ld a, V2
        add b, a
        st a, R
        halt

V1:     dw 14
V2:     dw 8
R:      dw 0
```

The program runs successfully and ends by displaying a message, *"CPU halted"*. James now wants to see the results of his work. So he clicks on a button in the main window saying, "Variables". He sees that V1 contains 14 and V2 contains 8, just as they should, but R contains 8 instead of 22 as expected. He asks the teacher what to do. The teacher advices James to open the "Registers" window and carefully go through every line of code by pressing F8 and looking at what happens. James does so. He sees that after the first two instructions the registers contain what they should. But after he executes the third instruction he notices that contents of register b changes to red (indicating that value changed), and it is indeed the sum of two numbers. James realises that result goes into b and not the accumulator for some reason, so he changes the fourth instruction to store register b in R. James runs the program again and sees that this time everything works perfectly.

17. File formats

There will be only one file type – assembly language program. The file will have no special format – it will simply store all the code that the user writes in text form, “as is”. Among the advantages of this approach are the facts that it will be very simple to implement, and the users will be able to edit their code without having CLab.

18. Security and integrity

This system will store no sensitive data and therefore will require no security measures to be taken.

All data that may suffer loss or corruption is programs saved on disk or the program under development. It is not crucial to make sure that programs that are loaded from disk are error-free, but if any time is available then an integrity check such as a checksum may be implemented. To maintain integrity of the program under development error traps should be used in implementation so that even if something goes wrong the user will still be able to save their work on disk.

19. Design confirmation

Having completed the design of the system, it should be confirmed with the user(s) the system is developed for. I have discussed this design section in detail with my end-user, and below is a list of all modifications that should be made.

- Disassembler window and disassembled instructions in Control Unit, although useful, will not be worth the implementation time necessary to have them
- The user should be able to edit all register values in the Registers and CPU window, as well as variables in the Variables window. Editing stack will not be crucial but could be useful if it won't be too time-consuming to implement it.
- Microprogram in the Control Unit window should be hidden in A-level mode, and only showed in the Full mode. The warning about complexity of the window should not be shown.
- The user should be able to turn off the "flying data windows" if they are not necessary.
- The Buses window should be extended to show an overview of the system unit, showing the CPU, RAM, buses and the three controllers – video, keyboard and speaker – and show the "flying windows" to display data flow between them.
- All animated examples should be available through a menu on the Main window. It should be possible to run them without having anything to do with the rest of the system. Apart from ALU operation animations already mentioned, the following animations should be added provided there is enough time to develop them:
  - Differences between different addressing modes
  - Arrays uses, especially in loops
  - Sorting algorithms
  - Binary trees explained, binary trees and searching.
- Window captions should not contain any detailed information
- Each window should have a What's This button in the caption, and a short note should popup describing any window element should the user use this button.
- RAM window: It is not necessary to show what the instruction would be if PC was to point at the selected cell; rather, the bytes being executed should be highlighted in some way. Also, all the code generated by the compiler should be highlighted. Interrupt vector table should be highlighted with a special color. Stack values should be highlighted. All non-empty memory cells should be highlighted. All other memory cells should be dimmed. Block operations such as copy/paste are not required.
- Whenever the user is not running a program, all debug windows should clearly indicate that so that the user does not accidentally try to use them.

20. Testing strategy

Fully testing a system as big as this is extremely time-consuming. Therefore having a testing strategy is vital.

In this document, I will only discuss alpha testing. Beta testing takes a lot of time and requires many people to use the program for a while. Unfortunately, no time is available for beta testing. Note that I *will* let my end-user use the system and get some feedback from him – that will be discussed in Appraisal.

Alpha testing will be mostly black box testing. I have two reasons for choosing black box as opposed to white box testing. One is that preparing for white box testing is a lot more time consuming. The other is that I tested the procedures while implementing them, trying out every possibility, so doing white box testing again may not be very efficient.

Alpha testing will be split into the following parts:

- Testing assembly language – this will involve writing different instructions with different opcodes etc. and making sure they are executed correctly.
- Testing windows – this will be going through all windows and make sure they function the way they are supposed to.
- Overall testing – this will involve developing several programs entirely in CLab. The purpose of this is to make sure it is in fact possible to develop a program in CLab (which neither of the previous two tests can prove).

Implementation

21. Plan

Developing a system of this complexity is a serious and time consuming task. To minimise time losses in case something goes wrong the system will be developed incrementally, in steps, so that at the end of each step the system can be run to see the results. An alternative way of developing it would be to write everything from the beginning to the end and only then run it for the first time. The advantage of the latter approach is that no time is spent on making an intermediary state work, but the disadvantage is that in case of a major design flaw a lot of code will have to be changed.

The outline of the implementation process is laid down below.

- Main form with stubs on most events; most frequent procedures in pUtils and pWinAPI; main types and variables in pGlobals.
- RAM window with basic functionality; pSynHigh unit with the required procedures doing everything in a single color
- CPU window with basic functionality; pExec executing simple instructions, do not show any data flow yet.
- pExec executes most instructions (except those that need anything not yet implemented)
- Code window with open/save facilities; program compilation
- Implement hardware windows; hardware interacts with code
- Interface windows
- OS/Debug windows
- Finish syntax highlighting etc.
- Any extra facilities if time left, such as a primitive Basic to assembly compiler

Note that at every stage some extra functionality may be implemented (this is especially true of pWinAPI, pUtils and pGlobals); the further the plans go the harder it becomes to predict precisely what will seem reasonable to develop next. This is why the further into the implementation the more general the points become.

At this point the actual coding begins; the next section will list the code after everything will have been written.

22. Listings

22.1. pGlobals

```
Option Explicit

'--------------------'
'--- SUBSTRUCTURES ---'
'--------------------'

'Video system state structure
Public Type TpVideo
  Mode As Integer  'Mode number character
  autoUpdate As Boolean 'Whether screens are refreshed automatically

  mdResX As Integer     'number of characters/pixels horizontally
  mdResY As Integer     'number of characters/pixels vertically
  mdType As Integer     '0=text, 1=graphics direct, 2=graphics
paletted
  mdColors As Integer   '0=monochrome, 1=16, 2=256, 3=65536,
4=16777216
  mdFntX As Integer     'one char width in screen pixels
  mdFntY As Integer     'one char height in screen pixels

  MemOff As Long             'Offset to video memory in RAM
  PalMem(0 To 255) As Long   'Video controller palette memory
  vDC As VirtualDC           'Virtual DC - bitmap
End Type

'-----------'
'--- APP ---'
'-----------'
Private Type TApp
  'Is set to true while unloading forms when app shuts down
  Terminating As Boolean
  'Previous window procedure pointer for fiMain
  PrevWndProc As Long

  'OS version
  RunningOnWinXP As Boolean
End Type
Public Appp As TApp

'------------'
'--- PROJ ---'
'------------'
Private Type TProj
  Modified As Boolean          'Variable for confirm queries

  '--- Global settings ---'
  Complexity As Integer        '0-basic, 1-alvl, 2-full
  NmbRep As Integer            '0hex 1bin 2decU 3decS

  '--- Program ---'
  P As TpPrg

  '--- Execution ---'
  Running As Boolean
  Paused As Boolean
  Halted As Boolean
  TickCount As Long
  CPU As TpCPU                 'fhCPU, fhCU, fhALU
  RAM() As Byte                'fhRAM
  Video As TpVideo             'fhVideo
End Type
Public Proj As TProj

'-----------------------------------------------'
' Sub Main()                                    '
'                                               '
' DESCRIPTION: Defines application entry point '
'                                               '
' NOTES: all this is rather unusual and         '
'   unnatural in VB but it gives me much more   '
'   control and it seems to work fine.          '
'-----------------------------------------------'
Sub Main()
  'We have started
  Appp.Terminating = False
  'XP controls if running on XP, no harm otherwise
  InitCommonControls

  'Get OS version
  Dim osver As OSVERSIONINFO
  osver.dwOSVersionInfoSize = 148 'according to Delphi's SizeOf.
Jesus, Microsoft, what would I do without Delphi? Buy VC?
  Call GetVersionEx(osver)
  If (osver.dwMajorVersion > 4) And (osver.dwMinorVersion > 0) Then
Appp.RunningOnWinXP = True Else Appp.RunningOnWinXP = False

  'Display startup form
  fiSplash.Show
  fiSplash.Refresh

  'Load all forms but keep them invisible
  fdKeyboard.Hide
  fdSpeaker.Hide
  fdVideo.Hide
  fhCPU.Hide
  fhCU.Hide
  fhRAM.Hide
  fiComp.Hide
  fiDisplay.Hide
  fiKeyboard.Hide
  fiMain.Hide
  fsCode.Hide
  fsRegs.Hide
  fsStack.Hide
  fsVars.Hide

  'Initialise Proj
  Proj.Modified = False
  Proj.Complexity = 0
  Proj.NmbRep = 0
  Proj.Running = False
  Proj.Paused = False
  'Initialize Proj.P
  ReDim Proj.P.Ref(-1 To -1)
  ReDim Proj.P.TknLine(-1 To -1)
  ReDim Proj.P.Backpatch(-1 To -1)
  ReDim Proj.P.ErrL.lError(-1 To -1)
  ReDim Proj.P.ErrL.lWarning(-1 To -1)
  ReDim Proj.P.ErrL.nError(-1 To -1)
  ReDim Proj.P.ErrL.nWarning(-1 To -1)
  ReDim Proj.P.ErrL.sError(-1 To -1)
  ReDim Proj.P.ErrL.sWarning(-1 To -1)
  ReDim Proj.P.Code_O2L(-1 To -1)
  ReDim Proj.P.Code_L2O(-1 To -1)
  ReDim Proj.P.Vars(-1 To -1)
  Proj.P.Code = ""
  Proj.P.CompileNeeded = True
  'Set complexity
  fhCPU.SetComplexity

  'Initialize p* modules
  Call pCompile.cmpInit
  Call pExec.exeInit
  'Initialise hardware modules
  Call fhCPU.Reset
  Call fhCU.Init
  Call fhRAM.Init
  'Initialise devices
  Call devInit
  'Initialize fs* and fi* modules
  Call fsRegs.Init
  Call fiMain.Init
  Call fsCode.Init
  Call fsStack.Init
  Call fsVars.Init

  'Show main window
  fiMain.Show
  'Show computer
  fiComp.Show
  'Destroy splash form
  Unload fiSplash

  ' This is the end of the Main procedure, but     '
  '   the application will keep running until     '
```

```
'    all forms have been unloaded.              '
'  Note that when the user presses Close button '
'    in forms' system menu the form is *hidden*, '
'    not actually closed and unloaded. The only '
'    way to shutdown the application is to close '
'    the fiMain window, which will unload all forms. '
End Sub

'---------------------------------------------------'
' Function WindowProc(ByVal hw As Long,             '
'   ByVal uMsg As Long, ByVal wParam As Long,       '
'   ByVal lParam As Long) As Long                   '
'                                                   '
' DESCRIPTION: Window procedure for fiMain          '
'                                                   '
' NOTES:                                            '
'   I wish someone knew how much I hate VB. This is '
'   one of hundreds of other things which cause it. '
'   VB does not allow to use AddressOf on any procs '
'   which are declared in a Form module, so even a  '
'   function so closely related to the form (much   '
'   closer than ANY other function) cannot be in the '
'   same module. It is very clumsy having it here.  '
'---------------------------------------------------'
Function WindowProc(ByVal hw As Long, ByVal uMsg As Long, ByVal
wParam As Long, ByVal lParam As Long) As Long
  WindowProc = fiMain.WindowProc(hw, uMsg, wParam, lParam)
End Function
```

22.2. pWinAPI

```
Option Explicit

'-----------'
'--- GDI ---'
'-----------'

'--- Text ---'
Public Declare Function SelectObject Lib "gdi32" (ByVal hdc As Long,
ByVal hObject As Long) As Long
Public Declare Function DeleteObject Lib "gdi32" (ByVal hObject As
Long) As Long
Public Declare Function GetStockObject Lib "gdi32" (ByVal nIndex As
Long) As Long
Public Declare Function SetBkMode Lib "gdi32" (ByVal hdc As Long,
ByVal nBkMode As Long) As Long
Public Declare Function SetTextColor Lib "gdi32" (ByVal hdc As Long,
ByVal crColor As Long) As Long
Public Declare Function CreateFont Lib "gdi32" Alias "CreateFontA"
(ByVal h As Long, ByVal W As Long, ByVal E As Long, ByVal o As Long,
ByVal W As Long, ByVal i As Long, ByVal u As Long, ByVal s As Long,
ByVal c As Long, ByVal OP As Long, ByVal CP As Long, ByVal Q As Long,
ByVal PAF As Long, ByVal f As String) As Long
Public Declare Function CreateBrushIndirect Lib "gdi32" (lpLogBrush
As LOGBRUSH) As Long
Public Type LOGBRUSH
        lbStyle As Long
        lbColor As Long
        lbHatch As Long
End Type
Public Declare Function GetTextExtentPoint32 Lib "gdi32" Alias
"GetTextExtentPoint32A" (ByVal hdc As Long, ByVal lpsz As String,
ByVal cbString As Long, lpSize As Size) As Long
Public Declare Function TextOut Lib "gdi32" Alias "TextOutA" (ByVal
hdc As Long, ByVal x As Long, ByVal y As Long, ByVal lpString As
String, ByVal nCount As Long) As Long
Public Declare Function DrawText Lib "user32" Alias "DrawTextA"
(ByVal hdc As Long, ByVal lpStr As String, ByVal nCount As Long,
lpRect As RECT, ByVal wFormat As Long) As Long

'--- Pictures ---'
Public Declare Function BitBlt Lib "gdi32" (ByVal hDestDC As Long,
ByVal x As Long, ByVal y As Long, ByVal nWidth As Long, ByVal nHeight
As Long, ByVal hSrcDC As Long, ByVal XSrc As Long, ByVal YSrc As
Long, ByVal dwRop As Long) As Long
Public Declare Function StretchBlt Lib "gdi32" (ByVal hdc As Long,
ByVal x As Long, ByVal y As Long, ByVal nWidth As Long, ByVal nHeight
As Long, ByVal hSrcDC As Long, ByVal XSrc As Long, ByVal YSrc As
Long, ByVal nSrcWidth As Long, ByVal nSrcHeight As Long, ByVal dwRop
As Long) As Long
Public Declare Function MoveToEx Lib "gdi32" (ByVal hdc As Long,
ByVal x As Long, ByVal y As Long, lpPoint As POINTAPI) As Long
Public Declare Function LineTo Lib "gdi32" (ByVal hdc As Long, ByVal
x As Long, ByVal y As Long) As Long
Public Declare Function Rectangle Lib "gdi32" (ByVal hdc As Long,
ByVal X1 As Long, ByVal Y1 As Long, ByVal X2 As Long, ByVal Y2 As
Long) As Long
Public Declare Function SetPixelV Lib "gdi32" (ByVal hdc As Long,
ByVal x As Long, ByVal y As Long, ByVal crColor As Long) As Long

Public Declare Function InvalidateRect Lib "user32" (ByVal hwnd As
Long, lpRect As RECT, ByVal bErase As Long) As Long
Public Declare Function GetClientRect Lib "user32" (ByVal hwnd As
Long, lpRect As RECT) As Long
Public Const TRANSPARENT = 1
Public Type POINTAPI
        x As Long
        y As Long
End Type
Public Const SRCCOPY = &HCC0020
Public Type Size
        cx As Long
        cy As Long
End Type

'---------------'
' System colors '
'---------------'
Public Declare Function GetSysColor Lib "user32" (ByVal nIndex As
Long) As Long
Public Const COLOR_ACTIVEBORDER = 10
Public Const COLOR_ACTIVECAPTION = 2
Public Const COLOR_APPWORKSPACE = 12
Public Const COLOR_BACKGROUND = 1
Public Const COLOR_BTNFACE = 15
Public Const COLOR_BTNHIGHLIGHT = 20
Public Const COLOR_ADJ_MAX = 100
Public Const COLOR_ADJ_MIN = -100
Public Const COLOR_BTNSHADOW = 16
Public Const COLOR_BTNTEXT = 18
Public Const COLOR_CAPTIONTEXT = 9
Public Const COLOR_GRAYTEXT = 17
Public Const COLOR_HIGHLIGHT = 13
Public Const COLOR_HIGHLIGHTTEXT = 14
Public Const COLOR_INACTIVEBORDER = 11
Public Const COLOR_INACTIVECAPTION = 3
Public Const COLOR_INACTIVECAPTIONTEXT = 19
Public Const COLOR_MENU = 4
Public Const COLOR_MENUTEXT = 7
Public Const COLOR_SCROLLBAR = 0
Public Const COLOR_WINDOW = 5
Public Const COLOR_WINDOWFRAME = 6
Public Const COLOR_WINDOWTEXT = 8

'-------------------------------'
' Window procedures & messaging '
'-------------------------------'
Public Declare Function CallWindowProc Lib "user32" Alias
"CallWindowProcA" (ByVal lpPrevWndFunc As Long, ByVal hwnd As
Long, ByVal Msg As Long, ByVal wParam As Long, ByVal lParam As
Long) As Long
Public Declare Function SetWindowLong Lib "user32" Alias
"SetWindowLongA" (ByVal hwnd As Long, ByVal nIndex As Long, ByVal
dwNewLong As Long) As Long
Public Declare Function DefWindowProc Lib "user32" Alias
"DefWindowProcA" (ByVal hwnd As Long, ByVal wMsg As Long, ByVal
wParam As Long, ByVal lParam As Long) As Long

Public Const GWL_WNDPROC = -4

Public Const WM_SYSCOMMAND = &H112
Public Const SC_MINIMIZE = &HF020&
Public Const SC_RESTORE = &HF120&
Public Const WM_NCLBUTTONDOWN = &HA1
Public Const HTCAPTION = 2
Public Const WM_SIZING = 532
Public Const WM_SIZE = &H5
Public Type RECT
        Left As Long
        Top As Long
        Right As Long
```

```
        Bottom As Long
End Type
Public Declare Function LockWindowUpdate Lib "user32" (ByVal hwndLock
As Long) As Long
Public Declare Function InitCommonControls Lib "comctl32" () As Long

'---------------------'
'--- COMMON DIALOGS ---'
'---------------------'
Public Declare Function GetOpenFileName Lib "comdlg32.dll" Alias
"GetOpenFileNameA" (pOpenfilename As OPENFILENAME) As Long
Public Declare Function GetSaveFileName Lib "comdlg32.dll" Alias
"GetSaveFileNameA" (pOpenfilename As OPENFILENAME) As Long
Public Declare Function ChooseColor Lib "comdlg32.dll" Alias
"ChooseColorA" (pChoosecolor As TCHOOSECOLOR) As Long
Public Declare Function CommDlgExtendedError Lib "comdlg32.dll" () As
Long
Public Type OPENFILENAME
        lStructSize As Long
        hwndOwner As Long
        hInstance As Long
        lpstrFilter As String
        lpstrCustomFilter As String
        nMaxCustFilter As Long
        nFilterIndex As Long
        lpstrFile As String
        nMaxFile As Long
        lpstrFileTitle As String
        nMaxFileTitle As Long
        lpstrInitialDir As String
        lpstrTitle As String
        FLAGS As Long
        nFileOffset As Integer
        nFileExtension As Integer
        lpstrDefExt As String
        lCustData As Long
        lpfnHook As Long
        lpTemplateName As String
End Type
Public Type TCHOOSECOLOR
        lStructSize As Long
        hwndOwner As Long
        hInstance As Long
        rgbResult As Long
        lpCustColors As Long
        FLAGS As Long
        lCustData As Long
        lpfnHook As Long
        lpTemplateName As String
End Type
Public Const OFN_PATHMUSTEXIST = &H800
Public Const OFN_FILEMUSTEXIST = &H1000
Public Const OFN_OVERWRITEPROMPT = &H2
Public Const WM_INITDIALOG = &H110

'-----------------'
'--- OS Version ---'
'-----------------'

Public Declare Function GetVersionEx Lib "kernel32" Alias
"GetVersionExA" (lpVersionInformation As OSVERSIONINFO) As Long
Public Type OSVERSIONINFO
        dwOSVersionInfoSize As Long
        dwMajorVersion As Long
        dwMinorVersion As Long
        dwBuildNumber As Long
        dwPlatformId As Long
        szCSDVersion As String * 128      '  Maintenance string
for PSS usage
End Type

'-----------'
'--- Misc ---'
'-----------'
Public Declare Function GetTickCount Lib "kernel32" () As Long
```

22.3. pUtils

```
Option Explicit

'----------------------------------------'
' Public declarations in this module:     '
'                                         '
' PROCEDURES:                             '
'   Errr                                  '
'   Tally                                 '
'   FieldStr                              '
'   InStrBack                             '
'   Hex2Dec                               '
'   Dec2Hex                               '
'   Bin2Dec                               '
'   Dec2Chr                               '
'   Chr2Dec                               '
'   Str2Chr                               '
'   TestCharset                           '
'   StringIsInt                           '
'   StringIsLong                          '
'----------------------------------------'

'Font for easier custom-drawn text
Public Type TFnt
  'Parameters
  ForeColor As Long
  BackColor As Long
  FaceName As String
  Size As Long
  Weight As Long
  'Associated GDI objects - call CreateFnt to init
  fntFont As Long
  fntBrush As Long
  'Internal params
  fntCreated As Boolean  'to free it safely
End Type

'---------------------------------------------'
' Public Sub Errr(msg As String)               '
'                                              '
' DESCRIPTION:                                 '
' Displays an error messsage box with a red    '
'   error icon and an OK button.               '
'                                              '
' PARAMETERS:                                  '
' msg - the message to be displayed.           '
'---------------------------------------------'
Public Sub Errr(Msg As String)
  Call MsgBox(Msg, vbOKOnly Or vbExclamation, "Error")
End Sub

'------------------------------------------------'
' Public Function Tally(where As String, what As  '
'   String)                                       '
'                                                 '
' DESCRIPTION: Counts the number of occurences of '
'   What in Where. What should be one character   '
'   long only, or the function will return 0      '
'------------------------------------------------'
Public Function Tally(where As String, what As String)
  Dim i As Integer, A As Integer
  A = 0
  For i = 1 To Len(where)
    If Mid(where, i, 1) = what Then A = A + 1
  Next
  Tally = A
End Function

'------------------------------------------------'
' Public Function FieldStr(row As String, index   '
'   As Integer, separator As String)              '
'                                                 '
' DESCRIPTION: Returns element number Index from  '
'   Row in which elements are separated by char   '
'   Separator. Separator should be 1 char long!   '
'------------------------------------------------'
Public Function FieldStr(row As String, Index As Integer,
separator As String)
  Dim i As Integer
  Dim s As String
```

```
  s = row + separator
  For i = 0 To Index
    If i = Index Then
      s = Left(s, InStr(s, separator) - 1)
      GoTo finished
    End If
    s = Mid(s, InStr(s, separator) + 1)
    If s = "" Then
      s = ""
      GoTo finished
    End If
  Next

finished:
  FieldStr = s
End Function

'-------------------------------------'
' Public Function InStrBack(where As  '
'   String, what As String) As Integer '
'                                     '
' DESCRIPTION: searches string where  '
'   for what starting at the end of   '
'   where. Returns offset of first    '
'   occurrence.                       '
'-------------------------------------'
Public Function InStrBack(where As String, what As String) As Integer
  Dim i As Integer
  For i = Len(where) To 1 Step -1
    If Mid(where, i, Len(what)) = what Then
      InStrBack = i
      Exit Function
    End If
  Next
  InStrBack = -1
End Function

'-----------------------------------------------'
' Public Function Hex2Dec(src As String) As Long '
'                                               '
' DESCRIPTION: Converts hexadecimal number to   '
'   decimal.                                    '
'-----------------------------------------------'
Public Function Hex2Dec(src As String) As Long
  Dim i As Long, A As Long, h16 As Long, Result As Long
  Dim s As String
  s = UCase(Mid(src, Len(src), 1))
  Result = IIf(s >= "0" And s <= "9", Asc(s) - 48, Asc(s) - 55)
  A = 1
  For i = Len(src) - 1 To 1 Step -1
    s = UCase(Mid(src, i, 1))
    h16 = IIf(s >= "0" And s <= "9", Asc(s) - 48, Asc(s) - 55)
    Result = Result + h16 * (16 ^ A)
    A = A + 1
  Next
  Hex2Dec = Result
End Function

'-------------------------------------------'
' Public Function Dec2Hex(src As Long, lng  '
'   As Integer) As String                   '
'                                           '
' DESCRIPTION: Converts decimal number src  '
'   to a hexadecimal number of given length '
'-------------------------------------------'
Public Function Dec2Hex(src As Long, lng As Integer) As String
  Dim s As String
  s = Hex(src)
  While Len(s) < lng
    s = "0" + s
  Wend
  Dec2Hex = s
End Function

'-----------------------------------------------'
' Public Function Bin2Dec(src As String) As Long '
'                                               '
' DESCRIPTION: Converts binary number to decimal '
'-----------------------------------------------'
Function Bin2Dec(src As String) As Long
  Dim i As Long, cost As Long, Result As Long
  cost = 1
  Result = 0
  For i = Len(src) To 1 Step -1
    Result = Result + (IIf(Mid(src, i, 1) = "0", 0, 1) * cost)
    cost = cost * 2
  Next
  Bin2Dec = Result
End Function

'-----------------------------------------'
' Public Function Dec2Bin(src As Long, lng '
'   As Integer) As String                 '
'                                         '
' DESCRIPTION: Converts a denary number   '
'   into a binary number.                 '
'-----------------------------------------'
Public Function Dec2Bin(src As Long, lng As Integer) As String
  Dim s As Long, o As Long, rs As String
  s = src
  rs = ""
  Do
    o = s
    s = s \ 2
    rs = IIf(s * 2 = o, "0", "1") + rs
  Loop While s > 0
  If Len(rs) < lng Then rs = String(lng - Len(rs), "0") + rs
  Dec2Bin = rs
End Function

'-----------------------------------------'
' Public Function Dec2Chr(src As Long, lng '
'   As Integer) As String                 '
'                                         '
' DESCRIPTION: Converts decimal number src '
'   to a string of chars (ie to base 256). '
'   lng is the length of the resulting    '
'   string. The result is big-endian.     '
'-----------------------------------------'
Public Function Dec2Chr(src As Long, lng As Integer) As String
  Dim n As Integer, mysrc As Long, cn As Long, pwr As Long
  Dim s As String
  s = String(lng, "0")
  mysrc = src
  For n = lng - 1 To 0 Step -1
    pwr = 256 ^ n
    cn = Int(mysrc / pwr)
    Mid(s, lng - n, 1) = Chr(cn)
    mysrc = mysrc - (cn * pwr)
  Next
  Dec2Chr = s
End Function

'-----------------------------------------------'
' Public Function Chr2Dec(str As String) As Long '
'                                               '
' DESCRIPTION: Converts a string of characters  '
'   to a number, interpreting the string as base '
'   256 big-endian number.                      '
'-----------------------------------------------'
Public Function Chr2Dec(str As String) As Long
  Dim i As Integer, l As Long
  l = 0
  For i = 1 To Len(str)
    l = l + Asc(Mid(str, i, 1)) * (256 ^ (Len(str) - i))
  Next
  Chr2Dec = l
End Function

'-----------------------------------------'
' Function Str2Chr(src As String) As String '
'                                         '
' DESCRIPTION: Formats the string src so  '
'   that each char is given by a hex number '
'-----------------------------------------'
Function Str2Chr(src As String) As String
  Dim i As Integer, s As String, rslt As String
  rslt = ""
  For i = 1 To Len(src)
    s = Hex(Asc(Mid(src, i, 1)))
    If Len(s) < 2 Then s = "0" + s
    rslt = rslt + s + " "
  Next
  Str2Chr = rslt
End Function

'-----------------------------------------------'
' Function TestCharset(testwhat As String, _    '
'   charset As String) As Boolean               '
'                                               '
' DESCRIPTION: Tests whether string testwhat    '
'   contains only allowed characters from charset '
'                                               '
' RETURNS: True if testwhat contains only chars '
'   from charset, False otherwise.              '
'-----------------------------------------------'
```

```
Function TestCharset(testwhat As String, charset As String) As
Boolean
  Dim i As Integer
  For i = 1 To Len(testwhat)
    If InStr(charset, Mid(testwhat, i, 1)) = 0 Then
      TestCharset = False
      GoTo tested
    End If
  Next
  TestCharset = True
tested:
End Function

'---------------------------------------------'
' Function StringIsInt(s As String) As Boolean '
'                                             '
' DESCRIPTION: Returns true if string can be  '
'   converted to type Integer.                '
'---------------------------------------------'
Function StringIsInt(s As String) As Boolean
  On Error GoTo strNotInt
  Dim i As Integer
  i = CInt(s)
  StringIsInt = True
  Exit Function
strNotInt:
  StringIsInt = False
End Function

'----------------------------------------------'
' Function StringIsLong(s As String) As Boolean '
'                                              '
' DESCRIPTION: Returns true if string can be   '
'   converted to type Long.                    '
'----------------------------------------------'
Function StringIsLong(s As String) As Boolean
  On Error GoTo strNotLong
  Dim i As Long
  i = CLng(s)
  StringIsLong = True
  Exit Function
strNotLong:
  StringIsLong = False
End Function

Function GetFilename(Save As Boolean, ByRef FileName As String, _
    InitDir As String, Filter As String, _
    DefExt As String, Title As String) As Boolean

  'Fill open structure
  Dim o As OPENFILENAME
  o.lStructSize = 88 'Delphi's SizeOf(TOpenFilename)
  o.hInstance = 0
  o.lpstrFilter = Replace(Filter, "|", Chr(0)) + Chr(0) + Chr(0)
  o.nFilterIndex = 0
  o.nMaxFile = 260
  o.lpstrFile = FileName + String(262 - Len(FileName), Chr(0))
  o.lpstrInitialDir = InitDir
  o.lpstrTitle = Title
  o.lpstrDefExt = DefExt
  o.lpfnHook = 0&
  o.hwndOwner = 0
  If Save Then
    o.FLAGS = OFN_PATHMUSTEXIST + OFN_OVERWRITEPROMPT
  Else
    o.FLAGS = OFN_FILEMUSTEXIST
  End If

  'Return result
  If Save Then
    GetFilename = (GetSaveFileName(o) <> 0)
  Else
    GetFilename = (GetOpenFileName(o) <> 0)
  End If
  'Return file name
  If InStr(o.lpstrFile, Chr(0)) > 0 Then o.lpstrFile =
Left(o.lpstrFile, InStr(o.lpstrFile, Chr(0)) - 1)
  FileName = o.lpstrFile
End Function

Public Function AppDir() As String
  AppDir = App.Path + IIf(Right(App.Path, 1) = "\", "", "\")
End Function

Public Function Dec2Fmt16(num As Long, fmt As Integer) As String
  If fmt = 0 Then 'hex
    Dec2Fmt16 = Dec2Hex(num, 4) + "h"
  ElseIf fmt = 1 Then 'bin
    Dec2Fmt16 = "binary not supported yet"
  ElseIf fmt = 2 Then 'decU
    Dec2Fmt16 = CStr(num)
  ElseIf fmt = 3 Then 'decS
    Dec2Fmt16 = CStr(IIf(num >= 32768, -65536 + num, num))
  Else
    Dec2Fmt16 = "Invalid format number"
  End If
End Function

'-----------------------------------------------'
' Public Function IsFmt16(num As String) As Boolean '
'                                               '
' RETURNS: True if operand is a 16-bit immediate '
'   constant, and range checks are passed.      '
'-----------------------------------------------'
Public Function IsFmt16(num As String) As Boolean
  On Error GoTo IsNot

  Dim s As String, testval As Long, minus As Boolean
  If Len(num) = 0 Then GoTo IsNot
  s = UCase(num)
  minus = False
  If Left(s, 1) = "-" Then
    If Len(s) = 1 Then GoTo IsNot
    s = Mid(s, 2)
    minus = True
  End If
  If Right(s, 1) = "H" Or Right(s, 1) = "B" Then If Len(s) = 1
Then GoTo IsNot
  'Check charset and try to convert (overflow will be trapped)
  If Right(s, 1) = "H" Then
    s = Left(s, Len(s) - 1)
    If Not TestCharset(s, "0123456789ABCDEF") Then GoTo IsNot
    testval = Hex2Dec(s)
  ElseIf Right(s, 1) = "B" Then
    s = Left(s, Len(s) - 1)
    If Not TestCharset(s, "01") Then GoTo IsNot
    testval = Bin2Dec(s)
  Else
    If Not TestCharset(s, "0123456789") Then GoTo IsNot
    testval = CLng(s)
  End If
  'Check range
  If minus Then testval = -testval
  If testval < -32768 Or testval > 65535 Then GoTo IsNot
  'Everything is fine
  IsFmt16 = True

  Exit Function
IsNot:
  IsFmt16 = False
End Function

'-----------------------------------------------------'
'-----------------------------------------------------'
Public Function Fmt2Dec16(c As String) As Long

  'Set an error trap and hope we checked C before calling this
  On Error GoTo HoustonWeVeGotAProblem

  'Prepare
  Dim s As String, minus As Boolean, n As Long
  s = UCase(c)
  minus = False
  If Left(s, 1) = "-" Then
    minus = True
    s = Mid(s, 2)
  End If
  'Convert
  If Right(s, 1) = "H" Then
    n = Hex2Dec(Left(s, Len(s) - 1))
  ElseIf Right(s, 1) = "B" Then
    n = Bin2Dec(Left(s, Len(s) - 1))
  Else
    n = CLng(s)
  End If
  'Deal with minus sign
  If minus Then n = -n
  'Return result
  Fmt2Dec16 = n

  Exit Function
HoustonWeVeGotAProblem:
  Fmt2Dec16 = -1  'should not happen unless IsFmt16 not called
before
End Function
```

```
'---------------------------------------'
' Public Sub CreateFnt(ByRef Fnt As TFnt) '
'                                       '
' Creates a font with those parameters  '
' specified in Fnt, by creating required '
' GDI objects.                          '
'---------------------------------------'
Public Sub CreateFnt(ByRef Fnt As TFnt)
  'Font
  Fnt.fntFont = CreateFont(Fnt.Size, 0, 0, 0, Fnt.Weight, _
    False, False, False, 1, 0, 0, 0, 0, Fnt.FaceName)
  Dim LB As LOGBRUSH
  'Brush
  LB.lbColor = Fnt.BackColor: LB.lbHatch = 0: LB.lbStyle = 0
  Fnt.fntBrush = CreateBrushIndirect(LB)
  'Finished
  Fnt.fntCreated = True
End Sub

'-----------------------------------------'
```

```
' Public Sub DestroyFnt(ByRef Fnt As TFnt) '
'                                        '
' Destroys the given font by deleting GDI '
' objects that are associated with it.   '
'----------------------------------------'
Public Sub DestroyFnt(ByRef Fnt As TFnt)
  If Not Fnt.fntCreated Then Exit Sub
  DeleteObject (Fnt.fntBrush)
  DeleteObject (Fnt.fntFont)
End Sub

Public Sub FntWrite(Fnt As TFnt, hdc As Long, str As String, rct
As RECT)
  'Draw text
  Call SelectObject(hdc, Fnt.fntFont)
  Call SetBkMode(hdc, TRANSPARENT)
  Call SetTextColor(hdc, Fnt.ForeColor)
  Call DrawText(hdc, str, Len(str), rct, 0)
End Sub
```

22.4. pCompile

```
Option Explicit

'----------------------------------------'
' Public declarations in this module:    '
'                                        '
' TYPES:                                 '
'   TPrg - assembly language program     '
'                                        '
' PROCEDURES:                            '
'   cmpInit - initialises this module    '
'   PrgCompile - compiles a program      '
'   PrgLoad - loads a program into RAM   '
'   CreatePrg - inits TPrg structure     '
'                                        '
' VARIABLES:                             '
'   Proj.P - program being worked on     '
'                                        '
' CONSTANTS:                             '
'   CharsetLabel - charset for labels    '
'----------------------------------------'

'-------------------------'
'--- Local declarations ---'
'-------------------------'

'Token type constants
Private Const tkUnknown = -1        'Used in the process of
tokenization
Private Const tkLabel = 0
Private Const tkVarDecl = 3
Private Const tkVarInit = 4
Private Const tkOpcode = 5
Private Const tkOperand = 6

'Token descriptions
Private tkName(-1 To 6) As String

'Opcode-to-machinecode reference structure
'Arrays of these structures are grouped by similarity in
compilation
Private Type TOpMCode
  Opcode As String
  Code As Byte
  Code2 As Byte
End Type

'List of type 0 opcodes (all operandless ones)
Private cdType0() As TOpMCode
'List op type 1 opcodes (bitwise shifts)
Private cdType1() As TOpMCode
'List op type 2 opcodes (4-way arithmetic & bitwise)
Private cdType2() As TOpMCode
'List op type 3 opcodes (inc,dec,neg,not,bswp)
Private cdType3() As TOpMCode
'List op type 4 opcodes (jmp, jxx, call)
Private cdType4() As TOpMCode

'Charset for label names
Public CharsetLabel As String
```

```
'Token structure - describes one token
Private Type TpToken
  Text As String      'eg MOV
  Type As Integer     'eg Operand (see constants above)
End Type

'Token line - array of tokens in one token line
Private Type TpTokenLine
  Token() As TpToken
  CodeLine As Integer
  CodeOffset As Integer
End Type

'Structure to store backpatch requests
Private Type TpBackpatch
  Name As String          'which variable's address needed
  Addr As Long            'where to write the address
  IsDW As Boolean         'true for absolute address
  RelTo As Long           'to calculate relative address
  CodeLine As Integer     'where requested
End Type

'Stores all references & their addresses
Private Type TpRef
  Name As String
  Addr As Long
  CodeLine As Integer     'where declared
End Type

'Error log
Private Type TpErrLog
  sError() As String      'Message
  lError() As Integer     'Line number
  nError() As String      'Error number
  sWarning() As String    'Message
  lWarning() As Integer   'Line number
  nWarning() As String    'Warning number
End Type

'Data about all declared variables
Private Type TpVars
  Name As String
  Addr As Long
End Type

'Assembly language program
Public Type TpPrg
  AsmLine() As String
  TknLine() As TpTokenLine
  Code As String
  Code_O2L() As Integer  'convert offset to source code line
  Code_L2O() As Integer  'convert source code line to offset

  Ref() As TpRef
  Backpatch() As TpBackpatch

  Vars() As TpVars      'Stores info about db/dw/ds
  ErrL As TpErrLog
```

```
  CompileNeeded As Boolean 'True if Compile called after editing
End Type

'-------------------------'
' Public Sub cmpInit()    '
'                         '
' Initializes this module '
'-------------------------'
Public Sub cmpInit()
  'Initialise charsets
  CharsetLabel =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890_-"
  'Init token names
  tkName(tkUnknown) = "unknown token"
  tkName(tkLabel) = "label"
  tkName(tkVarDecl) = "variable declaration"
  tkName(tkVarInit) = "variable initialisation"
  tkName(tkOpcode) = "opcode"
  tkName(tkOperand) = "operand"

  'Init opcode-to-machinecode array
  ReDim cdType0(0 To 19)
  cdType0(0).Opcode = "pushpc"   'MUST BE LOWERCASE!
  cdType0(0).Code = &H13
  cdType0(1).Opcode = "pushsp"
  cdType0(1).Code = &H14
  cdType0(2).Opcode = "pushfl"
  cdType0(2).Code = &H15
  cdType0(3).Opcode = "popsp"
  cdType0(3).Code = &H16
  cdType0(4).Opcode = "popfl"
  cdType0(4).Code = &H17
  cdType0(5).Opcode = "sp2b"
  cdType0(5).Code = &HF
  cdType0(6).Opcode = "stz"
  cdType0(6).Code = &H54
  cdType0(7).Opcode = "clz"
  cdType0(7).Code = &H55
  cdType0(8).Opcode = "stc"
  cdType0(8).Code = &H56
  cdType0(9).Opcode = "clc"
  cdType0(9).Code = &H57
  cdType0(10).Opcode = "sto"
  cdType0(10).Code = &H64
  cdType0(11).Opcode = "clo"
  cdType0(11).Code = &H65
  cdType0(12).Opcode = "sts"
  cdType0(12).Code = &H66
  cdType0(13).Opcode = "cls"
  cdType0(13).Code = &H67
  cdType0(14).Opcode = "sti"
  cdType0(14).Code = &H76
  cdType0(15).Opcode = "cli"
  cdType0(15).Code = &H77
  cdType0(16).Opcode = "ret"
  cdType0(16).Code = &H72
  cdType0(17).Opcode = "iret"
  cdType0(17).Code = &H73
  cdType0(18).Opcode = "halt"
  cdType0(18).Code = &H75
  cdType0(19).Opcode = "nop"
  cdType0(19).Code = &H8F
  'Init opcode-to-machinecode array
  ReDim cdType1(0 To 7)
  cdType1(0).Opcode = "lshl"
  cdType1(0).Code = &HC0
  cdType1(1).Opcode = "lshr"
  cdType1(1).Code = &HC1
  cdType1(2).Opcode = "ashl"
  cdType1(2).Code = &HC2
  cdType1(3).Opcode = "ashr"
  cdType1(3).Code = &HC3
  cdType1(4).Opcode = "rol"
  cdType1(4).Code = &HC4
  cdType1(5).Opcode = "ror"
  cdType1(5).Code = &HC5
  cdType1(6).Opcode = "rcl"
  cdType1(6).Code = &HC6
  cdType1(7).Opcode = "rcr"
  cdType1(7).Code = &HC7
  'Init opcode-to-machinecode array
  ReDim cdType2(0 To 13)
  cdType2(0).Opcode = "add"
  cdType2(0).Code = &H80
  cdType2(1).Opcode = "sub"
  cdType2(1).Code = &H83
  cdType2(2).Opcode = "adc"
  cdType2(2).Code = &H86
  cdType2(3).Opcode = "sbb"
  cdType2(3).Code = &H89
  cdType2(4).Opcode = "cmp"
  cdType2(4).Code = &H8C
  cdType2(5).Opcode = "mul"
  cdType2(5).Code = &H90
  cdType2(6).Opcode = "div"
  cdType2(6).Code = &H93
  cdType2(7).Opcode = "imul"
  cdType2(7).Code = &H96
  cdType2(8).Opcode = "idiv"
  cdType2(8).Code = &H99
  cdType2(9).Opcode = "mod"
  cdType2(9).Code = &H9C
  cdType2(10).Opcode = "and"
  cdType2(10).Code = &HB0
  cdType2(11).Opcode = "or"
  cdType2(11).Code = &HB3
  cdType2(12).Opcode = "xor"
  cdType2(12).Code = &HB6
  cdType2(13).Opcode = "test"
  cdType2(13).Code = &HB9
  'Init opcode-to-machinecode array
  ReDim cdType3(0 To 4)
  cdType3(0).Opcode = "inc"
  cdType3(0).Code = &HA0
  cdType3(0).Code2 = &HAC
  cdType3(1).Opcode = "dec"
  cdType3(1).Code = &HA4
  cdType3(1).Code2 = &HAD
  cdType3(2).Opcode = "neg"
  cdType3(2).Code = &HA8
  cdType3(2).Code2 = &HAE
  cdType3(3).Opcode = "not"
  cdType3(3).Code = &HBC
  cdType3(3).Code2 = &HAF
  cdType3(4).Opcode = "bswp"
  cdType3(4).Code = &HDC
  cdType3(4).Code2 = &H9F
  'Init opcode-to-machinecode array
  ReDim cdType4(0 To 13)
  cdType4(0).Opcode = "jmp"
  cdType4(0).Code = &H70
  cdType4(1).Opcode = "jg"
  cdType4(1).Code = &H40
  cdType4(2).Opcode = "jl"
  cdType4(2).Code = &H42
  cdType4(3).Opcode = "jge"
  cdType4(3).Code = &H43
  cdType4(4).Opcode = "jle"
  cdType4(4).Code = &H41
  cdType4(5).Opcode = "jz"
  cdType4(5).Code = &H50
  cdType4(6).Opcode = "jnz"
  cdType4(6).Code = &H51
  cdType4(7).Opcode = "jc"
  cdType4(7).Code = &H52
  cdType4(8).Opcode = "jnc"
  cdType4(8).Code = &H53
  cdType4(9).Opcode = "jo"
  cdType4(9).Code = &H60
  cdType4(10).Opcode = "jno"
  cdType4(10).Code = &H61
  cdType4(11).Opcode = "js"
  cdType4(11).Code = &H62
  cdType4(12).Opcode = "jns"
  cdType4(12).Code = &H63
  cdType4(13).Opcode = "call"
  cdType4(13).Code = &H71
End Sub

'-------------------------------------'
' Private Sub ReadCodeIntoProj()      '
'                                     '
' Copies contents of fsCode.RTB into  '
' the Proj structure                  '
'-------------------------------------'
Private Sub ReadCodeIntoProj()
  'Initialise program
  ReDim Proj.P.AsmLine(-1 To -1)
  'Copy source text
  Dim i As Integer
  ReDim Proj.P.AsmLine(-1 To fsCode.RTB.Lines.Count - 1)
  For i = 0 To UBound(Proj.P.AsmLine)
    Proj.P.AsmLine(i) = fsCode.RTB.Lines.Item(i)
```

```
    'If Right(Proj.P.AsmLine(i), 1) = Chr(13) Then
Proj.P.AsmLine(i) = Mid(Proj.P.AsmLine(i), 1,
Len(Proj.P.AsmLine(i)) - 1)
  Next
End Sub

'---------------------------------------'
' Public Sub PrgCompile()               '
'                                       '
' Compiles program in Proj, returning   '
' machine code in p.Code                '
'---------------------------------------'
Public Sub PrgCompile()
  'Load program into Proj
  ReadCodeIntoProj
  'Empty everything but AsmLine
  ReDim Proj.P.Ref(-1 To -1)
  ReDim Proj.P.TknLine(-1 To -1)
  ReDim Proj.P.Backpatch(-1 To -1)
  ReDim Proj.P.ErrL.lError(-1 To -1)
  ReDim Proj.P.ErrL.lWarning(-1 To -1)
  ReDim Proj.P.ErrL.nError(-1 To -1)
  ReDim Proj.P.ErrL.nWarning(-1 To -1)
  ReDim Proj.P.ErrL.sError(-1 To -1)
  ReDim Proj.P.ErrL.sWarning(-1 To -1)
  ReDim Proj.P.Code_O2L(-1 To -1)
  ReDim Proj.P.Code_L2O(-1 To -1)
  ReDim Proj.P.Vars(-1 To -1)
  Proj.P.Code = ""
  Proj.P.CompileNeeded = True
  'Compile
  Proj.P.CompileNeeded = False
  Call CompilePass1
  If UBound(Proj.P.ErrL.sError) = -1 Then Call CompilePass2
  If UBound(Proj.P.ErrL.sError) = -1 Then Call CompilePass3
  'Display errors
  Dim i As Integer
  fsCode.LErr.Clear
  For i = 0 To UBound(Proj.P.ErrL.sError)
    Call fsCode.LErr.AddItem("Error (" +
CStr(Proj.P.ErrL.lError(i) + 1) + "): " + Proj.P.ErrL.sError(i) +
" (" + Proj.P.ErrL.nError(i) + ").")
  Next
  'Display warnings
  For i = 0 To UBound(Proj.P.ErrL.sWarning)
    Call fsCode.LErr.AddItem("Warning (" +
CStr(Proj.P.ErrL.lWarning(i) + 1) + "): " +
Proj.P.ErrL.sWarning(i) + " (" + Proj.P.ErrL.nWarning(i) + ").")
  Next

  fiMain.UpdateAll True
End Sub

'-------------------------------'
' Public Sub PrgLoad            '
'                               '
' Loads the program in Proj into '
' RAM at offset zero.           '
'-------------------------------'
Public Sub PrgLoad()
  Dim i As Integer
  For i = 1 To Len(Proj.P.Code)
    Proj.RAM(i - 1) = Asc(Mid(Proj.P.Code, i, 1))
  Next
  fhRAM.Update
End Sub

'---------------------------------------------------'
' Private Sub CompilePass1()                        '
'                                                   '
' DESCRIPTION: Compilation pass 1: tokenize program '
'   1. Clean the source code                        '
'   2. Split every line into tokens                 '
'   3. Determine token types                        '
'   4. Token pattern analysis                       '
'   5. Preparations for variable compilation        '
'                                                   '
' OUTPUT: Proj.P.Tkn* containing tokenized program. '
'   Proj will be ready for CompilePass2             '
'---------------------------------------------------'
Private Sub CompilePass1()
  Dim i As Integer, A As Integer, nl As Integer
  Dim s As String
  With Proj.P

  '-------------------------'
  '--- Prepare to tokenize ---'
  '-------------------------'

  'Copy program to work on it (and do some processing in the
meantime)
  Dim sc_line() As Integer
  ReDim sc_line(-1 To UBound(.AsmLine))
  'nl = 0
  For i = 0 To UBound(.AsmLine)
    'Copy
    s = .AsmLine(i)
    'Remove comment
    A = InStr(s, ";")
    If A > 0 Then s = Left(s, A - 1)
    'Clean spaces
    s = CleanSpaces(s)
    'Trim
    s = Trim(s)
    'Save string
    .AsmLine(i) = s
    'Remember line number
    sc_line(i) = i
    'Increase nl to point to next line
    'nl = nl + 1
  Next

  '-------------------------'
  '--- Pure tokenization ---'
  '-------------------------'

  'Prepare Tokenized
  ReDim .TknLine(-1 To UBound(.AsmLine))

  'Tokenize
  nl = 0
  Dim sptr As Long, tmps As String, bs As Integer
  For i = 0 To UBound(.AsmLine)
    If .AsmLine(i) <> "" Then
      'Prep token line
      ReDim .TknLine(nl).Token(-1 To -1)
      .TknLine(nl).CodeLine = sc_line(i)
      'Get the tokens
      sptr = 1
      tmps = .AsmLine(i)
      Do While sptr <= Len(tmps)
        'Skip all spaces
        Do While (Mid(tmps, sptr, 1) = " " Or Mid(tmps, sptr, 1) =
Chr(9)) And sptr <= Len(tmps)
          sptr = sptr + 1
        Loop
        'Prep token
        ReDim Preserve .TknLine(nl).Token(-1 To
UBound(.TknLine(nl).Token) + 1)
        .TknLine(nl).Token(UBound(.TknLine(nl).Token)).Text = ""
        .TknLine(nl).Token(UBound(.TknLine(nl).Token)).Type =
tkUnknown
        'Is it a string?
        If Mid(tmps, sptr, 1) = """" Then
          .TknLine(nl).Token(UBound(.TknLine(nl).Token)).Text =
""""
          sptr = sptr + 1
          If sptr > Len(tmps) Then GoTo dneS 'I know it's bad
programming but...
          bs = 0
          Do
            If Mid(tmps, sptr, 1) = """" Then
              bs = bs + 1
              .TknLine(nl).Token(UBound(.TknLine(nl).Token)).Text
= .TknLine(nl).Token(UBound(.TknLine(nl).Token)).Text + """"
            Else
              bs = 0
              .TknLine(nl).Token(UBound(.TknLine(nl).Token)).Text
= .TknLine(nl).Token(UBound(.TknLine(nl).Token)).Text + Mid(tmps,
sptr, 1)
            End If
            sptr = sptr + 1
          Loop Until ((bs \ 2 <> bs / 2) And (Mid(tmps, sptr, 1) =
"""")) Or (sptr > Len(tmps))
dneS:
          .TknLine(nl).Token(UBound(.TknLine(nl).Token)).Text =
.TknLine(nl).Token(UBound(.TknLine(nl).Token)).Text + """"
        Else
          'There is no need for any advanced token splitting
          'in a language as simple as this. In an expression
          'like [b*c+4356h] we don't care that * and + should
          'be separate tokens - we don't need that. The only
          'reason to setup an algorithm like this which has the
          'power to tokenize like that is to recogize strings.
          'Otherwise all we need is to separate tokens by the
          'fact that there is a whitespace inbetween them.
```

```
            Do
              .TknLine(nl).Token(UBound(.TknLine(nl).Token)).Text =
.TknLine(nl).Token(UBound(.TknLine(nl).Token)).Text + Mid(tmps,
sptr, 1)
              sptr = sptr + 1
            Loop Until (Mid(tmps, sptr, 1) = " ") Or (Mid(tmps,
sptr, 1) = Chr(9)) Or (sptr > Len(tmps))

          End If
        Loop

        'Increment nl
        nl = nl + 1
      End If
    Next
    'Truncate the program
    ReDim Preserve .TknLine(-1 To nl - 1)

    '----------------------------'
    '--- Token type detection ---'
    '----------------------------'

    Dim ft As Integer
    'Identify token types
    For i = 0 To UBound(.TknLine)
      'Indentify single tokens
      For A = 0 To UBound(.TknLine(i).Token)
        s = .TknLine(i).Token(A).Text
        If Right(s, 1) = ":" Then 'Label
          .TknLine(i).Token(A).Type = tkLabel
        ElseIf UCase(s) = "DB" Or UCase(s) = "DW" Or UCase(s) = "DS"
Then 'VarDecl
          .TknLine(i).Token(A).Type = tkVarDecl
        Else 'Everything else
          .TknLine(i).Token(A).Type = tkUnknown
        End If
      Next
      'Identify first token after all labels
      ft = -1
      For A = UBound(.TknLine(i).Token) To 0 Step -1
        If .TknLine(i).Token(A).Type <> tkLabel Then ft = A
      Next
      If ft = -1 Then GoTo alllabels
      'Use info about single tokens to further identify them
      If .TknLine(i).Token(ft).Type = tkUnknown Then
        'All tokens which are tkUnknown at this point are either
        ' opcode tokens or follow-up tokens. Therefore, if a
tkUnkown
        ' token is the first token, it is definitely tkOpcode
        .TknLine(i).Token(ft).Type = tkOpcode
      End If
      'Identify unknown follow-up tokens
      ' (all first tokens have been identified by now)
      If .TknLine(i).Token(ft).Type = tkVarDecl Then 'Variable
declaration
        For A = ft + 1 To UBound(.TknLine(i).Token)
          If .TknLine(i).Token(A).Type = tkUnknown Then
.TknLine(i).Token(A).Type = tkVarInit
        Next
      ElseIf .TknLine(i).Token(ft).Type = tkOpcode Then 'Opcode
        For A = ft + 1 To UBound(.TknLine(i).Token)
          If .TknLine(i).Token(A).Type = tkUnknown Then
.TknLine(i).Token(A).Type = tkOperand
        Next
      Else 'Error trap - just in case
        Errr ("pCompile.TokenizeCode: ft token is of an invalid
type. Contact the author.")
        Exit Sub
      End If
alllabels:
    Next

    'At this point there should be no tkUnknown tokens (in theory at
least)
    ' Now analyse token patterns to 1). adjust some patterns, 2).
pick
    ' up all invalid patterns

    '-----------------------------'
    '--- Token pattern analysis ---'
    '-----------------------------'

    'Calculate how many token lines we will have (depends on label
splitting)
    ft = 0 'store resulting number of token lines
    Dim b As Boolean
    For i = 0 To UBound(.TknLine)
      'A token line for every existing token line...
      ft = ft + 1
      '... plus a token line for every label in the line...
      b = False
      For A = 0 To UBound(.TknLine(i).Token)
        If .TknLine(i).Token(A).Type = tkLabel Then ft = ft + 1 Else
b = True
      Next
      '... minus a token line for every Label-only token line
      If Not b Then ft = ft - 1
    Next

    'Declare and dimension intermediary array
    Dim tkl() As TpTokenLine
    ReDim tkl(-1 To ft - 1)

    'Go through current tokens
    nl = 0    'next free token line in tkn
    For i = 0 To UBound(.TknLine)
      '-----------------------------'
      '-- STAGE ONE: deal with labels --'
      '-----------------------------'

      'Issue label warnings and separate the labels
      b = False
      ft = 0 'number of non-label tokens
      For A = 0 To UBound(.TknLine(i).Token)
        If .TknLine(i).Token(A).Type <> tkLabel Then
          b = True
          ft = ft + 1
        Else
          'Warning
          If b Then Call AddWng("Labels must not be preceded by
other tokens. Label moved to beginning of line. Offending label:
""" + .TknLine(i).Token(A).Text + """", .TknLine(i).CodeLine,
"WC1001")
          'Separation
          ReDim tkl(nl).Token(-1 To 0)
          tkl(nl).CodeLine = .TknLine(i).CodeLine
          tkl(nl).Token(0).Text = .TknLine(i).Token(A).Text
          tkl(nl).Token(0).Type = .TknLine(i).Token(A).Type
          'Take next line in tkn
          nl = nl + 1
        End If
      Next
      'Store the rest (if anything)
      If ft > 0 Then
        ReDim tkl(nl).Token(-1 To ft - 1)
        tkl(nl).CodeLine = .TknLine(i).CodeLine
        ft = 0 'next available token in tkl(nl).Token()
        For A = 0 To UBound(.TknLine(i).Token)
          If .TknLine(i).Token(A).Type <> tkLabel Then
            tkl(nl).Token(ft).Text = .TknLine(i).Token(A).Text
            tkl(nl).Token(ft).Type = .TknLine(i).Token(A).Type
            ft = ft + 1
          End If
        Next
        'nl = nl + 1 DO NOT INCREMENT - STILL WORKING ON IT
      Else
        'In order to avoid doing the rest in this IF block go to the
        'end of the loop when there's nothing left to process
        GoTo nnext
      End If

      'CAUTION! Now the rest of the work should be done on tkl(nl)
      'NL SHOULD BE INCREMENTED AT THE END!

      '------------------------------'
      '-- STAGE TWO: deal with vardecl --'
      '------------------------------'

      If UBound(tkl(nl).Token) + 1 = 1 Then
        If tkl(nl).Token(0).Type = tkVarDecl Then
          'Issue a warning
          Call AddWng("Variable not initialised explicitly. Assuming
uninitialised variable.", tkl(nl).CodeLine, "WC1002")
          'Add a proper token
          .TknLine(i).Token(0).Text = tkl(nl).Token(0).Text
          .TknLine(i).Token(0).Type = tkl(nl).Token(0).Type
          ReDim tkl(nl).Token(-1 To 1)
          tkl(nl).CodeLine = .TknLine(i).CodeLine
          tkl(nl).Token(0).Text = .TknLine(i).Token(0).Text
          tkl(nl).Token(0).Type = .TknLine(i).Token(0).Type
          tkl(nl).Token(1).Text = "?"
          tkl(nl).Token(1).Type = tkVarInit
        End If
      End If
```

```
    '-------------------------------'
    '-- STAGE THREE: check patterns --'
    '-------------------------------'

    b = False
    If UBound(tkl(nl).Token) + 1 = 1 Then
      If tkl(nl).Token(0).Type = tkLabel Or tkl(nl).Token(0).Type
= tkOpcode Then
        b = True
      Else
        If tkl(nl).Token(0).Type = tkVarDecl Then
          s = "a variable declaration without initialisation"
'this is pretty much impossible
        ElseIf tkl(nl).Token(0).Type = tkVarInit Then
          s = "variable initialisation" 'same as above
        ElseIf tkl(nl).Token(0).Type = tkOperand Then
          s = "an operand" 'same as above
        Else
          s = "an unknown token" 'same as above
        End If
        'Because the tokenization procedure should not allow for
any
        'of these errors, this is an internal error
        Call Errr("pCompile.TokenizeCode: Internal Error (" +
CStr(tkl(nl).CodeLine) + "): a line cannot start with " + s + ".
Contact the author.")
      End If
    ElseIf UBound(tkl(nl).Token) + 1 = 2 Then
      If tkl(nl).Token(0).Type = tkOpcode And
tkl(nl).Token(1).Type = tkOperand Then
        b = True
      ElseIf tkl(nl).Token(0).Type = tkVarDecl And
tkl(nl).Token(1).Type = tkVarInit Then
        b = True
      Else
        Call AddErr("Invalid token combination: """ +
tkName(tkl(nl).Token(0).Type) + """ and """ +
tkName(tkl(nl).Token(1).Type) + """", tkl(nl).CodeLine, "EC1002")
      End If
    ElseIf UBound(tkl(nl).Token) + 1 = 3 Then
      If tkl(nl).Token(0).Type = tkOpcode And
tkl(nl).Token(1).Type = tkOperand And tkl(nl).Token(2).Type =
tkOperand Then
        b = True
      Else
        Call AddErr("Invalid token combination: """ +
tkName(tkl(nl).Token(0).Type) + """, """ +
tkName(tkl(nl).Token(1).Type) + """ and """ +
tkName(tkl(nl).Token(1).Type) + """", tkl(nl).CodeLine, "EC1003")
      End If
    ElseIf UBound(tkl(nl).Token) + 1 > 3 Then
      Call AddErr("A line cannot contain more than three tokens.
This line contains " + CStr(UBound(tkl(nl).Token) + 1) + "
tokens.", tkl(nl).CodeLine, "EC1004")
    Else
      Call Errr("pCompile.CompilePass1: token line with less than
1 token encountered. Contact the author.")
    End If
    'Skip next stage if token line is not valid
    If Not b Then nl = nl + 1: GoTo nnext

    nl = nl + 1
nnext:
  Next

  '-----------------'
  '--- Return tkn ---'
  '-----------------'

  'Check if nl is what we thought it would be
  If nl <> UBound(tkl) + 1 Then
    Call Errr("pCompile.TokenizeCode: Internal Error (N/A):
predicted token line count is not equal to actual token line
count. Contact the author.")
  End If

  ReDim .TknLine(-1 To nl - 1)
  For i = 0 To UBound(.TknLine)
    ReDim .TknLine(i).Token(-1 To UBound(tkl(i).Token))
    .TknLine(i).CodeLine = tkl(i).CodeLine
    For A = 0 To UBound(.TknLine(i).Token)
      .TknLine(i).Token(A).Text = tkl(i).Token(A).Text
      .TknLine(i).Token(A).Type = tkl(i).Token(A).Type
    Next
  Next

  '-------------------------'
  '--- Deal with variables ---'
  '-------------------------'

  'After this stage every OFFSET(varX) will be replaced with varX
  'and every varX with [varX]
  Dim acc As String, doing As Boolean

  For i = 0 To UBound(.TknLine)
    For A = 0 To UBound(.TknLine(i).Token)
      If .TknLine(i).Token(A).Type = tkOperand Then
        'Get the string to analyse
        s = .TknLine(i).Token(A).Text
        'Go through every symbol, accumulating segments separated
        'by one of []+* into acc
        acc = ""
        doing = Not TestCharset(Right(s, 1), "[]+*")
        For nl = Len(s) To 1 Step -1
          If TestCharset(Mid(s, nl, 1), "[]+*") Then
            'Acc contains a chunk. Work on it in DoItGoSub
            If doing Then GoSub DoItGoSub
            doing = True
            acc = ""
          Else
            'Just keep accumulating acc
            If doing Then acc = Mid(s, nl, 1) + acc
          End If
        Next
        'Run it once at the end in case s doesn't end with any of
[]+*
        GoSub DoItGoSub
        GoTo EndOfGoSub
DoItGoSub:
        'Analyse
        If UCase(Left(acc, 7)) = "OFFSET(" And Right(acc, 1) = ")"
Then
          'Have an offset. Remove OFFSET( ) completely
          s = Left(s, nl) + Mid(acc, 8, Len(acc) - 8) + Mid(s, nl
+ Len(acc) + 1)
        ElseIf Not OperandIsRg(acc) Then
          'Have either a number or a variable name
          If Not TestCharset(Left(acc, 1), "-0123456789") Then
            'Definitely not a number. Enclose in []
            s = Left(s, nl) + "[" + acc + "]" + Mid(s, nl +
Len(acc) + 1)
          End If
        End If
        Return
EndOfGoSub:
        'Save the analysed string back
        .TknLine(i).Token(A).Text = s
      End If
    Next
  Next

  End With
End Sub

'-----------------------------------------------------------------'
' Private Sub CompilePass2()                                      '
'                                                                 '
' DESCRIPTION: Compilation pass 2: code generation                '
'   1. Generate code for all correct instructions                 '
'   2. Generate errors/warnings for incorrect instructions        '
'   3. Generate a label-to-address list used in backpatching      '
'   4. Generate a "variable requested" list for backpatching      '
'                                                                 '
' PARAMETERS:                                                     '
'   p - should contain tokenized program (Tkn*)                   '
'                                                                 '
' OUTPUT: p.Code containing compiled machine code.                '
'-----------------------------------------------------------------'
Private Sub CompilePass2()
  Dim tli As Integer                        'token line loop var
  Dim tl As TpTokenLine                     'current token line
  Dim ctl As String                         'compiled token line

  Dim i As Integer, l1 As Long, l2 As Long, l3 As Long
  Dim t As String, s As String

  With Proj.P

  'Initialise structures
  .Code = ""
  ReDim .Ref(-1 To -1)

  'Loop through all token lines
  For tli = 0 To UBound(.TknLine)
  tl.CodeLine = .TknLine(tli).CodeLine
  tl.CodeOffset = .TknLine(tli).CodeOffset
```

```
  tl.Token = .TknLine(tli).Token
  ctl = ""
  'Store offset of the beginning of the token line
  .TknLine(tli).CodeOffset = Len(.Code)

  '=================================================================='
  '=================================================================='
  '=================================================================='
  '=================================================================='

  'Validity check
  If UBound(tl.Token) < 0 Then
    Call Errr("pCompile.CompilePass2: empty token line at
compilation stage 2. Contact the author.")
    Exit Sub
  End If

  '-------------------'
  '=== LABEL TOKEN ==='
  '-------------------'
  If tl.Token(0).Type = tkLabel Then
    'Check label name
    If Not TestCharset(Left(tl.Token(0).Text,
Len(tl.Token(0).Text) - 1), CharsetLabel) Then
      Call AddErr("Invalid label name - '" +
Left(tl.Token(0).Text, Len(tl.Token(0).Text) - 1) + "'",
tl.CodeLine, "EC2008")
      GoTo NextTokenLine
    End If
    If UCase(tl.Token(0).Text) = "A" Or UCase(tl.Token(0).Text) =
"B" Or UCase(tl.Token(0).Text) = "C" Or UCase(tl.Token(0).Text) =
"D" Or UCase(tl.Token(0).Text) = "E" Then
      Call AddErr("Label name cannot be same as register name - '"
+ Left(tl.Token(0).Text, Len(tl.Token(0).Text) - 1) + "'",
tl.CodeLine, "EC2009")
      GoTo NextTokenLine
    End If
    'Add reference
    ReDim Preserve .Ref(-1 To UBound(.Ref) + 1)
    .Ref(UBound(.Ref)).Addr = Len(.Code)
    .Ref(UBound(.Ref)).Name = Left(tl.Token(0).Text,
Len(tl.Token(0).Text) - 1)
    .Ref(UBound(.Ref)).CodeLine = tl.CodeLine
    GoTo NextTokenLine
  End If

  '---------------------'
  '=== VARDECL TOKEN ==='
  '---------------------'
  If tl.Token(0).Type = tkVarDecl Then
    If UCase(tl.Token(0).Text) = "DB" Then   'Byte variable
      If OperandIsIm8(tl.Token(1).Text) Then
        ctl = Chr(CIm8(tl.Token(1).Text, Len(.Code), Len(.Code),
tl.CodeLine))
      ElseIf tl.Token(1).Text = "?" Then
        ctl = Chr(0)
      Else
        Call AddErr("Variable initialisation sequence is neither
'?' nor a valid constant.", tl.CodeLine, "EC2002")
      End If
    ElseIf UCase(tl.Token(0).Text) = "DW" Then 'Word variable
      'Store reference
      ReDim Preserve .Vars(-1 To UBound(.Vars) + 1)
      .Vars(UBound(.Vars)).Addr = Len(.Code)
      'Compile
      If OperandIsIm16(tl.Token(1).Text) Then
        'Initialisation can be a variable offset.We don't mind
        'if it is - CIm16 would just add a ref for a backpatch
        ctl = Dec2Chr(CIm16(tl.Token(1).Text, Len(.Code),
tl.CodeLine), 2)
      ElseIf tl.Token(1).Text = "?" Then
        ctl = Chr(0) + Chr(0)
      Else
        Call AddErr("Variable initialisation sequence is neither
'?' nor a valid constant.", tl.CodeLine, "EC2002")
      End If
    Else                                     'String literal
      If Left(tl.Token(1).Text, 1) = """" And
Right(tl.Token(1).Text, 1) = """" Then
        ctl = Replace(Mid(tl.Token(1).Text, 2,
Len(tl.Token(1).Text) - 2), """""", """")
      ElseIf tl.Token(1).Text = "?" Then
        ctl = ""
      Else
        Call AddErr("DS variable should be initialised with either
? or a string literal enclosed with """".", tl.CodeLine, "EC2019")
      End If
    End If
    GoTo NextTokenLine
  End If

  '--------------------'
  '=== OPCODE TOKEN ==='
  '--------------------'
  If tl.Token(0).Type <> tkOpcode Then
    Errr ("pCompile.CompilePass2: first token in token line is not
opcode. Contact the author.")
    Exit Sub
  End If

  'Opcode name
  t = LCase(tl.Token(0).Text)

  '---------------------'
  '--- Type 0 opcodes ---'
  '---------------------'
  For i = 0 To UBound(cdType0)
    If t = cdType0(i).Opcode Then
      If UBound(tl.Token) + 1 > 1 Then
        Call AddErr("Opcode takes 0 operands, not " +
CStr(UBound(tl.Token) + 1) + ".", tl.CodeLine, "EC2001")
        ctl = ""
      Else
        ctl = Chr(cdType0(i).Code)
      End If
      GoTo NextTokenLine
    End If
  Next

  '---------------------'
  '--- Type 1 opcodes ---'
  '---------------------'
  For i = 0 To UBound(cdType1)
    If t = cdType1(i).Opcode Then '(lshr,ashl,rol,rcl etc)
      'Check param count
      If UBound(tl.Token) + 1 <> 3 Then
        Call AddErr("Opcode takes 2 operands, not " +
CStr(UBound(tl.Token) + 1) + ".", tl.CodeLine, "EC2001")
        ctl = ""
        GoTo NextTokenLine
      End If
      'Compile depending on type
      If UCase(tl.Token(1).Text) = "A" And
OperandIsRgn(tl.Token(2).Text) Then
        'A/Rg
        l1 = (Asc(UCase(tl.Token(2).Text)) - 66) * 32
        ctl = Chr(cdType1(i).Code) + Chr(l1)
      ElseIf OperandIsRg(tl.Token(1).Text) Then
        'Rg/N
        If Not StringIsInt(tl.Token(2).Text) Then
          Call AddErr("Syntax error in operand OR opcode and
operand incompatible. Offending operand: '" + tl.Token(2).Text +
"'.", tl.CodeLine, "EC2005")
          ctl = ""
          GoTo NextTokenLine
        End If
        l1 = CInt(tl.Token(2).Text)
        If l1 < 0 Or l1 > 15 Then
          Call AddErr("The number of shift cycles must be between
0 and 15.", tl.CodeLine, "EC2006")
          ctl = ""
          GoTo NextTokenLine
        End If
        l2 = 128 + l1
        l3 = (Asc(UCase(tl.Token(1).Text)) - 66) * 32
        If UCase(tl.Token(1).Text) = "A" Then l2 = l2 + 16 Else l2
= l2 + l3
        ctl = Chr(cdType1(i).Code) + Chr(l2)
      Else
        'Error
        If UCase(tl.Token(1).Text) = "A" Then
          Call AddErr("Syntax error in operand OR opcode and
operand incompatible. Offending operand: '" + tl.Token(2).Text +
"'.", tl.CodeLine, "EC2005")
        Else
          Call AddErr("Syntax error in operand OR opcode and
operand incompatible. Offending operand: '" + tl.Token(1).Text +
"'.", tl.CodeLine, "EC2005")
        End If
        ctl = ""
        GoTo NextTokenLine
      End If
      GoTo NextTokenLine
    End If
  Next
```

```
'---------------------'
'--- Type 2 opcodes ---'
'---------------------'
For i = 0 To UBound(cdType2)
  If t = cdType2(i).Opcode Then '(and,or,add,sub etc)
    'Check param count
    If UBound(tl.Token) + 1 <> 3 Then
      Call AddErr("Opcode takes 2 operands, not " +
CStr(UBound(tl.Token) + 1) + ".", tl.CodeLine, "EC2001")
      ctl = ""
      GoTo NextTokenLine
    End If
    'Compile depending on type
    l1 = 0
    If (OperandIsRg(tl.Token(2).Text) And
UCase(tl.Token(1).Text) = "A") Then
      'A/Rg
      If UCase(tl.Token(2).Text) = "A" Then l1 = 4 Else l1 =
(Asc(UCase(tl.Token(2).Text)) - 66)
      ctl = Chr(cdType2(i).Code) + Chr(l1)
    ElseIf (OperandIsRg(tl.Token(1).Text) And
UCase(tl.Token(2).Text) = "A") Then
      'Rg/A
      If UCase(tl.Token(1).Text) = "A" Then l1 = 4 Else l1 =
(Asc(UCase(tl.Token(1).Text)) - 66)
      l1 = l1 + 8
      ctl = Chr(cdType2(i).Code) + Chr(l1)
    ElseIf UCase(tl.Token(1).Text) = "A" And
OperandIsIm16(tl.Token(2).Text) Then
      'A/I
      l1 = CIm16(tl.Token(2).Text, Len(ctl) + 1, tl.CodeLine)
      If l1 >= 0 And l1 <= 65535 Then
        ctl = Chr(cdType2(i).Code + 1) + Dec2Chr(l1, 2)
      Else
        Call AddErr("16 bit immediate constant is out of
range.", tl.CodeLine, "EC2007")
        ctl = ""
      End If
    ElseIf UCase(tl.Token(1).Text) = "A" And
OperandIsMem(tl.Token(2).Text) Then
      'A/M
      ctl = CompileMemoryAddressing(tl.Token(2).Text, Len(.Code)
+ 1, tl.CodeLine)
      If ctl <> "" Then
        ctl = Chr(cdType2(i).Code + 2) + ctl
      End If
    Else
      If UCase(tl.Token(1).Text) = "A" Or
OperandIsRg(tl.Token(1).Text) Then
        Call AddErr("Syntax error in operand OR opcode and
operand incompatible. Offending operand: '" + tl.Token(2).Text +
"'.", tl.CodeLine, "EC2005")
      Else
        Call AddErr("Syntax error in operand OR opcode and
operand incompatible. Offending operand: '" + tl.Token(1).Text +
"'.", tl.CodeLine, "EC2005")
      End If
      ctl = ""
    End If
    GoTo NextTokenLine
  End If
Next

'---------------------'
'--- Type 3 opcodes ---'
'---------------------'
For i = 0 To UBound(cdType3)
  If t = cdType3(i).Opcode Then '(inc,dec,neg,not,bswp)
    If UBound(tl.Token) + 1 <> 2 Then
      Call AddErr("Opcode takes 1 operand, not " +
CStr(UBound(tl.Token) + 1) + ".", tl.CodeLine, "EC2001")
      ctl = ""
      GoTo NextTokenLine
    End If
    If OperandIsRgn(tl.Token(1).Text) Then
      ctl = Chr(cdType3(i).Code + (Asc(UCase(tl.Token(1).Text))
- 66))
    ElseIf UCase(tl.Token(1).Text) = "A" Then
      ctl = Chr(cdType3(i).Code2)
    Else
      Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: '" + tl.Token(1).Text + "'.",
tl.CodeLine, "EC2005")
      ctl = ""
    End If
    GoTo NextTokenLine
  End If
Next

'---------------------'
'--- Type 4 opcodes ---'
'---------------------'
For i = 0 To UBound(cdType4)
  If t = cdType4(i).Opcode Then '(jmp, jXX, call)
    If UBound(tl.Token) + 1 <> 2 Then
      Call AddErr("Opcode takes 1 operand, not " +
CStr(UBound(tl.Token) + 1) + ".", tl.CodeLine, "EC2001")
      ctl = ""
      GoTo NextTokenLine
    End If
    s = tl.Token(1).Text
    If OperandIsMem(s) Then
      ctl = Chr(cdType4(i).Code) +
CompileMemoryAddressing(tl.Token(1).Text, Len(.Code) + 1,
tl.CodeLine)
    Else
      Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: '" + tl.Token(1).Text + "'.",
tl.CodeLine, "EC2005")
      ctl = ""
    End If
    GoTo NextTokenLine
  End If
Next

'``````````'
'``` LD ```'
'``````````'
If t = "ld" Then
  If OperandIsIm16(tl.Token(2).Text) Then
    If OperandIsRgn(tl.Token(1).Text) Then
      'Rn/Im16
      ctl = Chr(&H20 + (Asc(UCase(tl.Token(1).Text)) - 66)) +
Dec2Chr(CIm16(tl.Token(2).Text, Len(.Code) + 1, tl.CodeLine), 2)
    ElseIf UCase(tl.Token(1).Text) = "A" Then
      'A/Im16
      ctl = Chr(&H24) + Dec2Chr(CIm16(tl.Token(2).Text,
Len(.Code) + 1, tl.CodeLine), 2)
    ElseIf OperandIsMem(tl.Token(1).Text) Then
      Call AddErr("Cannot load a constant into a memory cell
directly.", tl.CodeLine, "EC2014")
      ctl = ""
    Else
      Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: '" + tl.Token(1).Text + "'.",
tl.CodeLine, "EC2005")
      ctl = ""
    End If
  ElseIf OperandIsRg(tl.Token(1).Text) And
OperandIsRg(tl.Token(2).Text) Then
    'R/R
    l1 = 0
    If UCase(tl.Token(1).Text) = "A" Then l1 = l1 + 128 Else l1
= l1 + (Asc(UCase(tl.Token(1).Text)) - 66) * 8
    If UCase(tl.Token(2).Text) = "A" Then l1 = l1 + 64 Else l1 =
l1 + (Asc(UCase(tl.Token(2).Text)) - 66)
    ctl = Chr(&H25) + Chr(l1)
  ElseIf OperandIsRg(tl.Token(1).Text) And
OperandIsMem(tl.Token(2).Text) Then
    'R/Mem
    If UCase(tl.Token(1).Text) = "A" Then l1 = 4 Else l1 =
(Asc(UCase(tl.Token(1).Text)) - 66)
    ctl = Chr(&H26) + Chr(l1) +
CompileMemoryAddressing(tl.Token(2).Text, Len(.Code) + 2,
tl.CodeLine)
  ElseIf OperandIsRg(tl.Token(2).Text) And
OperandIsMem(tl.Token(1).Text) Then
    'Mem/R
    If UCase(tl.Token(2).Text) = "A" Then l1 = 4 Else l1 =
(Asc(UCase(tl.Token(2).Text)) - 66)
    ctl = Chr(&H27) + Chr(l1) +
CompileMemoryAddressing(tl.Token(1).Text, Len(.Code) + 2,
tl.CodeLine)
  Else
    If OperandIsIm16(tl.Token(1).Text) Then
      Call AddErr("Cannot load into a constant (first operand
cannot be a constant).", tl.CodeLine, "EC2013")
    Else
      Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: check both.", tl.CodeLine,
"EC2005")
    End If
    ctl = ""
  End If

  GoTo NextTokenLine
```

```
  End If

  '``````````'
  '```  ST  ```'
  '``````````'
  If t = "st" Then
    If OperandIsIm16(tl.Token(1).Text) Then
      If OperandIsRgn(tl.Token(2).Text) Then
        'Im16/Rn
        ctl = Chr(&H30 + (Asc(UCase(tl.Token(2).Text)) - 66)) +
Dec2Chr(CIm16(tl.Token(1).Text, Len(.Code) + 1, tl.CodeLine), 2)
      ElseIf UCase(tl.Token(2).Text) = "A" Then
        'Im16/A
        ctl = Chr(&H34) + Dec2Chr(CIm16(tl.Token(1).Text,
Len(.Code) + 1, tl.CodeLine), 2)
      ElseIf OperandIsMem(tl.Token(2).Text) Then
        Call AddErr("Cannot store a constant in a memory cell
directly.", tl.CodeLine, "EC2016")
        ctl = ""
      Else
        Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: '" + tl.Token(2).Text + "'.",
tl.CodeLine, "EC2005")
        ctl = ""
      End If
    ElseIf OperandIsRg(tl.Token(2).Text) And
OperandIsRg(tl.Token(1).Text) Then
      'R/R
      l1 = 0
      If UCase(tl.Token(2).Text) = "A" Then l1 = l1 + 128 Else l1
= l1 + (Asc(UCase(tl.Token(2).Text)) - 66) * 8
      If UCase(tl.Token(1).Text) = "A" Then l1 = l1 + 64 Else l1 =
l1 + (Asc(UCase(tl.Token(1).Text)) - 66)
      ctl = Chr(&H35) + Chr(l1)
    ElseIf OperandIsRg(tl.Token(2).Text) And
OperandIsMem(tl.Token(1).Text) Then
      'Mem/R
      If UCase(tl.Token(2).Text) = "A" Then l1 = 4 Else l1 =
(Asc(UCase(tl.Token(2).Text)) - 66)
      ctl = Chr(&H36) + Chr(l1) +
CompileMemoryAddressing(tl.Token(1).Text, Len(.Code) + 2,
tl.CodeLine)
    ElseIf OperandIsRg(tl.Token(1).Text) And
OperandIsMem(tl.Token(2).Text) Then
      'R/Mem
      If UCase(tl.Token(1).Text) = "A" Then l1 = 4 Else l1 =
(Asc(UCase(tl.Token(1).Text)) - 66)
      ctl = Chr(&H37) + Chr(l1) +
CompileMemoryAddressing(tl.Token(2).Text, Len(.Code) + 2,
tl.CodeLine)
    Else
      If OperandIsIm16(tl.Token(2).Text) Then
        Call AddErr("Cannot store in a constant (second operand
cannot be a constant).", tl.CodeLine, "EC2015")
      Else
        Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: check both.", tl.CodeLine,
"EC2005")
      End If
      ctl = ""
    End If

    GoTo NextTokenLine
  End If

  '```  PUSH  ```'
  If t = "push" Then
    If UCase(tl.Token(1).Text) = "A" Then
      'push A
      ctl = Chr(&H10)
    ElseIf OperandIsRgn(tl.Token(1).Text) Then
      'push Rn
      ctl = Chr(0 + Asc(UCase(tl.Token(1).Text)) - 66)
    ElseIf OperandIsIm16(tl.Token(1).Text) Then
      'push I
      ctl = Chr(&H12) + Dec2Chr(CIm16(tl.Token(1).Text, Len(.Code)
+ 1, tl.CodeLine), 2)
    Else
      Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: '" + tl.Token(1).Text + "'.",
tl.CodeLine, "EC2005")
      ctl = ""
    End If
    GoTo NextTokenLine
  End If

  '```  POP  ```'
  If t = "pop" Then
    If UCase(tl.Token(1).Text) = "A" Then
      'pop A
      ctl = Chr(&H11)
    ElseIf OperandIsRgn(tl.Token(1).Text) Then
      'pop Rn
      ctl = Chr(4 + Asc(UCase(tl.Token(1).Text)) - 66)
    Else
      Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: '" + tl.Token(1).Text + "'.",
tl.CodeLine, "EC2005")
      ctl = ""
    End If
    GoTo NextTokenLine
  End If

  '```  LEA  ```'
  If t = "lea" Then
    If UCase(tl.Token(1).Text) = "A" Then
      'lea A,M
      ctl = Chr(&H2F) + CompileMemoryAddressing(tl.Token(2).Text,
Len(.Code) + 1, tl.CodeLine)
    ElseIf OperandIsRgn(tl.Token(1).Text) Then
      'lea Rn,M
      ctl = Chr(&H8 + Asc(UCase(tl.Token(1).Text)) - 66) +
CompileMemoryAddressing(tl.Token(2).Text, Len(.Code) + 1,
tl.CodeLine)
    Else
      Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: '" + tl.Token(1).Text + "'.",
tl.CodeLine, "EC2005")
      ctl = ""
    End If
    GoTo NextTokenLine
  End If

  '```  XCHG  ```'
  If t = "xchg" Then
    If UCase(tl.Token(1).Text) = "A" Then
      'xchg A,Rn
      ctl = Chr(&HF0 + Asc(UCase(tl.Token(2).Text)) - 66)
    ElseIf UCase(tl.Token(2).Text) = "A" Then
      'xchg Rn,A
      ctl = Chr(&HF0 + Asc(UCase(tl.Token(1).Text)) - 66)
    ElseIf ((UCase(tl.Token(1).Text) = "C") And
(UCase(tl.Token(2).Text) = "D")) Or ((UCase(tl.Token(2).Text) =
"C") And (UCase(tl.Token(1).Text) = "D")) Then
      'xchg c,d
      ctl = Chr(&HE6)
    ElseIf ((UCase(tl.Token(1).Text) = "C") And
(UCase(tl.Token(2).Text) = "E")) Or ((UCase(tl.Token(2).Text) =
"C") And (UCase(tl.Token(1).Text) = "E")) Then
      'xchg c,e
      ctl = Chr(&HE7)
    ElseIf ((UCase(tl.Token(1).Text) = "D") And
(UCase(tl.Token(2).Text) = "E")) Or ((UCase(tl.Token(2).Text) =
"D") And (UCase(tl.Token(1).Text) = "E")) Then
      'xchg d,e
      ctl = Chr(&HF4)
    ElseIf ((UCase(tl.Token(1).Text) = "B") And
(UCase(tl.Token(2).Text) = "C")) Or ((UCase(tl.Token(2).Text) =
"B") And (UCase(tl.Token(1).Text) = "C")) Then
      'xchg b,c
      ctl = Chr(&HF5)
    ElseIf ((UCase(tl.Token(1).Text) = "B") And
(UCase(tl.Token(2).Text) = "D")) Or ((UCase(tl.Token(2).Text) =
"B") And (UCase(tl.Token(1).Text) = "D")) Then
      'xchg b,d
      ctl = Chr(&HF6)
    ElseIf ((UCase(tl.Token(1).Text) = "B") And
(UCase(tl.Token(2).Text) = "E")) Or ((UCase(tl.Token(2).Text) =
"B") And (UCase(tl.Token(1).Text) = "E")) Then
      'xchg b,e
      ctl = Chr(&HF7)
    Else
      Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: check both.", tl.CodeLine,
"EC2005")
      ctl = ""
    End If
    GoTo NextTokenLine
  End If

  '```  INT  ```'
  If t = "int" Then
    If OperandIsIm8(tl.Token(1).Text) Then
      'int I8
      ctl = Chr(&H74) + CIm8(tl.Token(1).Text, Len(.Code) + 1,
Len(.Code) + 2, tl.CodeLine)
```

```
    Else
      Call AddErr("Operand for INT must be an 8 bit immediate
constant.", tl.CodeLine, "EC2017")
      ctl = ""
    End If
    GoTo NextTokenLine
  End If

  '``` IN ```'
  If t = "in" Then
    If OperandIsIm8(tl.Token(2).Text) Then
      If UCase(tl.Token(1).Text) = "A" Then
        'in A,I8
        ctl = Chr(&HE5) + Chr(CIm8(tl.Token(2).Text, Len(.Code) +
1, -1, tl.CodeLine))
      ElseIf OperandIsRgn(tl.Token(1).Text) Then
        'in Rn,I8
        ctl = Chr(&HD8 + Asc(UCase(tl.Token(1).Text)) - 66) +
Chr(CIm8(tl.Token(2).Text, Len(.Code) + 1, -1, tl.CodeLine))
      Else
        Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: '" + tl.Token(1).Text + "'.",
tl.CodeLine, "EC2005")
        ctl = ""
      End If
    ElseIf OperandIsRg(tl.Token(2).Text) Then
      If OperandIsRg(tl.Token(1).Text) Then
        'in R1,R2
        If UCase(tl.Token(1).Text) = "A" Then l1 = 32 Else l1 =
(Asc(UCase(tl.Token(1).Text)) - 66) * 4
        If UCase(tl.Token(2).Text) = "A" Then l1 = l1 + 16 Else l1
= l1 + (Asc(UCase(tl.Token(2).Text)) - 66)
        ctl = Chr(&HE4) + Chr(l1)
      Else
        Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: '" + tl.Token(1).Text + "'.",
tl.CodeLine, "EC2005")
        ctl = ""
      End If
    ElseIf OperandIsIm16(tl.Token(2).Text) Then
      Call AddErr("Port address must be an 8 bit immediate
constant (0 to 255).", tl.CodeLine, "EC2018")
      ctl = ""
    Else
      Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: '" + tl.Token(2).Text + "'.",
tl.CodeLine, "EC2005")
      ctl = ""
    End If
    GoTo NextTokenLine
  End If

  '``` OUT ```'
  If t = "out" Then
    If OperandIsRg(tl.Token(1).Text) And
OperandIsRg(tl.Token(2).Text) Then
      'out R1,R2
      If UCase(tl.Token(1).Text) = "A" Then l1 = 32 Else l1 =
(Asc(UCase(tl.Token(1).Text)) - 66) * 4
      If UCase(tl.Token(2).Text) = "A" Then l1 = l1 + 16 Else l1 =
l1 + (Asc(UCase(tl.Token(2).Text)) - 66)
      ctl = Chr(&HD4) + Chr(l1)
    ElseIf OperandIsIm8(tl.Token(1).Text) And
OperandIsIm16(tl.Token(2).Text) Then
      'out I8,I
      ctl = Chr(&HD7) + Chr(CIm8(tl.Token(1).Text, Len(.Code) + 1,
-1, tl.CodeLine)) + Dec2Chr(CIm16(tl.Token(2).Text, Len(.Code) +
2, tl.CodeLine), 2)
    ElseIf OperandIsIm16(tl.Token(1).Text) And
OperandIsIm16(tl.Token(2).Text) Then
      Call AddErr("Port address has to be an 8 bit immediate
constant (0 to 255).", tl.CodeLine, "EC2018")
      ctl = ""
    ElseIf OperandIsIm16(tl.Token(2).Text) Then
      If UCase(tl.Token(1).Text) = "A" Then
        'out A,I
        ctl = Chr(&HD5) + Dec2Chr(CIm16(tl.Token(2).Text,
Len(.Code) + 1, tl.CodeLine), 2)
      ElseIf OperandIsRgn(tl.Token(1).Text) Then
        'out R,I
        ctl = Chr(&HE0 + Asc(UCase(tl.Token(1).Text)) - 66) +
Dec2Chr(CIm16(tl.Token(2).Text, Len(.Code) + 1, tl.CodeLine), 2)
      Else
        Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: '" + tl.Token(1).Text + "'.",
tl.CodeLine, "EC2005")
        ctl = ""
      End If
```

```
    ElseIf OperandIsIm8(tl.Token(1).Text) Then
      If UCase(tl.Token(2).Text) = "A" Then
        'out I8,A
        ctl = Chr(&HD6) + Chr(CIm8(tl.Token(1).Text, Len(.Code) +
1, -1, tl.CodeLine))
      ElseIf OperandIsRgn(tl.Token(2).Text) Then
        'out I8,R
        ctl = Chr(&HD0 + Asc(UCase(tl.Token(2).Text)) - 66) +
Chr(CIm8(tl.Token(1).Text, Len(.Code) + 1, -1, tl.CodeLine))
      Else
        Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: '" + tl.Token(2).Text + "'.",
tl.CodeLine, "EC2005")
        ctl = ""
      End If
    ElseIf OperandIsIm16(tl.Token(1).Text) Then
      Call AddErr("Port address has to be an 8 bit immediate
constant (0 to 255).", tl.CodeLine, "EC2018")
      ctl = ""
    Else
      Call AddErr("Syntax error in operand OR opcode and operand
incompatible. Offending operand: check both.", tl.CodeLine,
"EC2005")
      ctl = ""
    End If
    GoTo NextTokenLine
  End If

  'All instruction processing should end with a GoTo
NextTokenLine.
  'So if we arrive here, opcode was not found. Add error message
  Call AddErr("Opcode not recognized: '" + t + "'. Check
spelling.", tl.CodeLine, "EC2010")

NextTokenLine:
  '================================================================'
  '================================================================'
  '================================================================'
  '================================================================'

  .Code = .Code + ctl
  Next

  'Calculate offset to source code line conversion
  ReDim .Code_O2L(-1 To Len(.Code) - 1)
  For i = Len(.Code) - 1 To 0 Step -1
    For tli = UBound(.TknLine) To 0 Step -1
      If .TknLine(tli).CodeOffset <= i Then
        .Code_O2L(i) = .TknLine(tli).CodeLine
        GoTo takeNext
      End If
    Next
    .Code_O2L(i) = 0
takeNext:
  Next
  'Calculate source code line to offset conversion
  ReDim .Code_L2O(-1 To UBound(.AsmLine))
  For i = UBound(.AsmLine) To 0 Step -1
    For tli = 0 To UBound(.TknLine)
    'For tli = UBound(.TknLine) To 0 Step -1
      If .TknLine(tli).CodeLine = i Then
        .Code_L2O(i) = .TknLine(tli).CodeOffset
        GoTo takeNext2
      End If
    Next
    If i = UBound(.AsmLine) Then .Code_L2O(i) = Len(.Code) Else
.Code_L2O(i) = .Code_L2O(i + 1)
takeNext2:
  Next
  ReDim Preserve .Code_L2O(-1 To UBound(.Code_L2O) + 1)
  .Code_L2O(UBound(.Code_L2O)) = Len(.Code) + 1 'point to last
Halt

  End With
End Sub

'----------------------------------------------------------'
' Private Sub CompilePass3()                                '
'                                                           '
' DESCRIPTION: Compilation pass 3: address backpatching     '
'   1. Check for "label already declared"                   '
'   2. Check for "undeclared reference"                     '
'   3. Backpatch all requseted places                       '
'                                                           '
' PARAMETERS:                                               '
'   p - should contain tokenized program (Tkn*)             '
'                                                           '
' OUTPUT: p.Code containing compiled machine code.          '
```

```
'--------------------------------------------------------'
Private Sub CompilePass3()
  Dim i As Integer, A As Integer, t As Integer

  With Proj.P

  'Check for "label already declared"
  For i = 0 To UBound(.Ref)
    For A = i + 1 To UBound(.Ref)
      If UCase(.Ref(i).Name) = UCase(.Ref(A).Name) Then
        Call AddErr("Label already declared: '" + .Ref(A).Name +
"'. Previous declaration on line " + CStr(.Ref(i).CodeLine) + ".",
.Ref(A).CodeLine, "EC3001")
      End If
    Next
  Next

  'Check for "undeclared reference"
  For i = 0 To UBound(.Backpatch)
    For A = 0 To UBound(.Ref)
      If UCase(.Backpatch(i).Name) = UCase(.Ref(A).Name) Then GoTo
fnd
    Next
    Call AddErr("Undeclared reference: '" + .Backpatch(i).Name +
"'.", .Backpatch(i).CodeLine, "EC3002")
fnd:
  Next

  'Backpatch
  For i = 0 To UBound(.Backpatch)
    For A = 0 To UBound(.Ref)
      If .Backpatch(i).Name = .Ref(A).Name Then
        If .Backpatch(i).IsDW Then
          Mid(.Code, .Backpatch(i).Addr + 1, 2) =
Dec2Chr(.Ref(A).Addr, 2)
        Else
          t = .Ref(A).Addr - .Backpatch(i).RelTo
          If t < 0 Then t = 256 + t
          Mid(.Code, .Backpatch(i).Addr + 1, 1) = Chr(t)
        End If
      End If
    Next
  Next

  'Match variable references against reference names
  For i = 0 To UBound(.Vars)
    .Vars(i).Name = "[unnamed]"
  Next
  For i = 0 To UBound(.Ref)
    For A = 0 To UBound(.Vars)
      If .Ref(i).Addr = .Vars(A).Addr Then .Vars(A).Name =
.Ref(i).Name
    Next
  Next

  'Add HALT at the end
  .Code = .Code + Chr(&H75)

  End With
End Sub

'------------------------------------------------------------'
' Private Function CompileMemoryAddressing(c As String,       '
'   adr As Long, cl As Integer) As String                     '
'                                                             '
' DESCRIPTION: compiles addressing from assembly language     '
'   into machine code.                                        '
'                                                             '
' PARAMETERS:                                                 '
'   c - contains addressing operand in assembly language      '
'   adr - machine code offset for where the addressing will   '
'     start (required for backpatching)                       '
'   cl - code line containing the addressing for error log.   '
'   p - program structure with error log etc                  '
'                                                             '
' RETURNS: machine codes ready to be added to p.Code          '
'------------------------------------------------------------'
Private Function CompileMemoryAddressing(c As String, adr As Long,
cl As Integer) As String
  Dim ctl As String, ma As String, l1 As Long, l2 As Long
  ma = c
  ctl = ""

  'Detect type
  If Left(ma, 2) = "[[" And Right(ma, 2) = "]]" Then
    '---------------------------------'
    '--- Memory Indirect Immediate ---'
    '---------------------------------'
    ctl = Chr(1) + Dec2Chr(CIm16(Mid(ma, 3, Len(ma) - 4), adr + 1,
cl), 2)
  Else
    ma = Mid(ma, 2, Len(ma) - 2)
    If OperandIsIm16(ma) Then
      '--------------------------------'
      '--- Memory Direct Immediate ---'
      '--------------------------------'
      ctl = Chr(0) + Dec2Chr(CIm16(ma, adr + 1, cl), 2)
    ElseIf OperandIsRgn(ma) Then
      '---------------------------------'
      '--- Memory Indirect Register ---'
      '---------------------------------'
      ctl = Chr(64 + (Asc(UCase(ma)) - 66))
    Else
      '-----------------------'
      '--- Memory Indexed ---'
      '-----------------------'
      l1 = 128
      If UCase(Left(ma, 2)) = "B+" Then
        l1 = l1 + 64
        ma = Mid(ma, 3)
      End If
      If InStr(ma, "+") > 0 Then
        ctl = Dec2Chr(CIm16(Mid(ma, InStr(ma, "+") + 1), adr + 1,
cl), 2)
        ma = Left(ma, InStr(ma, "+") - 1)
        l1 = l1 + 32
      End If
      If InStr(ma, "*") > 0 Then
        If Not StringIsLong(Mid(ma, InStr(ma, "*") + 1)) Then
          Call AddErr("Memory addressing scaling factor should be
0, 1, 2 or 4.", cl, "EC2011")
          ctl = ""
          Return
        End If
        l2 = CLng(Mid(ma, InStr(ma, "*") + 1))
        If l2 <> 0 And l2 <> 1 And l2 <> 2 And l2 <> 4 Then
          Call AddErr("Memory addressing scaling factor should be
0, 1, 2 or 4.", cl, "EC2011")
          ctl = ""
          Return
        End If
        If l2 = 4 Then l2 = 3 '00=0,01=1,10=2,11=4
        l1 = l1 + l2 * 4
        ma = Left(ma, InStr(ma, "*") - 1)
      End If
      'Checks have been completed before, so now ma must contain
      'just the indexation register B, C, D or E
      l1 = l1 + (Asc(UCase(ma)) - 66)
      ctl = Chr(l1) + ctl
    End If
  End If
  'Check
  If ctl = "" Then Call Errr("pCompile.CompilePass3: mem addr
compilation successful but returns nothing. Contact the author.")
  'Return compiled addressing
  CompileMemoryAddressing = ctl
End Function

'-----------------------------------------------------------'
' Private Function OperandIsRg(s As String) As Boolean '
'                                                           '
' PARAMETERS:                                               '
'   c - assembly language operand to be tested             '
'                                                           '
' RETURNS: True if operand is a register (A,B,C,D,E)      '
'-----------------------------------------------------------'
Private Function OperandIsRg(s As String) As Boolean
  If UCase(s) = "A" Or UCase(s) = "B" Or UCase(s) = "C" Or
UCase(s) = "D" Or UCase(s) = "E" Then
    OperandIsRg = True
  Else
    OperandIsRg = False
  End If
End Function

'-----------------------------------------------------------'
' Private Function OperandIsRgn(s As String) As Boolean '
'                                                           '
' PARAMETERS:                                               '
'   c - assembly language operand to be tested             '
'                                                           '
' RETURNS: True if operand is a GP register (B,C,D,E)     '
'-----------------------------------------------------------'
Private Function OperandIsRgn(s As String) As Boolean
  If UCase(s) = "B" Or UCase(s) = "C" Or UCase(s) = "D" Or
UCase(s) = "E" Then
```

```
    OperandIsRgn = True
  Else
    OperandIsRgn = False
  End If
End Function

'---------------------------------------------------------'
' Private Function OperandIsMem(c As String) As Boolean  '
'                                                         '
' PARAMETERS:                                             '
'   c - assembly language operand to be tested            '
'                                                         '
' RETURNS: True if operand is a memory operand.           '
'---------------------------------------------------------'
Private Function OperandIsMem(c As String) As Boolean
  Dim s As String, st As String
  s = UCase(c)
  'Checks
  If Len(s) < 3 Then
    OperandIsMem = False
    Exit Function
  End If
  If Left(s, 1) <> "[" Or Right(s, 1) <> "]" Then
    OperandIsMem = False
    Exit Function
  End If
  s = Mid(s, 2, Len(s) - 2)

  'Indirect immediate
  If Left(s, 1) = "[" And Right(s, 1) = "]" Then
    OperandIsMem = OperandIsIm16(Mid(s, 2, Len(s) - 2))
    Exit Function

  'Indirect register
  ElseIf OperandIsRgn(s) Then
    OperandIsMem = True
    Exit Function

  'Direct (immediate)
  ElseIf OperandIsIm16(s) Then
    OperandIsMem = True
    Exit Function

  'Either memory indexed or not memory
  Else
    'Base register
    If Left(s, 2) = "B+" Then s = Mid(s, 3)
    'Offset
    If InStr(s, "+") > 0 Then
      If OperandIsIm16(Mid(s, InStr(s, "+") + 1)) Then
        s = Left(s, InStr(s, "+") - 1)
      Else
        OperandIsMem = False
        Exit Function
      End If
    End If
    'Scale
    If InStr(s, "*") > 0 Then
      st = Mid(s, InStr(s, "*") + 1)
      If TestCharset(st, "0123456789") Then 'no range check here -
        s = Left(s, InStr(s, "*") - 1)      'so can generate error
      Else                                  'msg with explanation
        OperandIsMem = False                'later
        Exit Function
      End If
    End If
    'Register
    'What is left by now should be the central register which is
compulsory
    OperandIsMem = OperandIsRgn(s)
  End If
End Function

'--------------------------------------------------------'
' Private Function OperandIsIm8(c As String) As Boolean  '
'                                                        '
' PARAMETERS:                                            '
'   c - assembly language operand to be tested           '
'                                                        '
' RETURNS: True if operand is an 8-bit immediate         '
'   constant, and range checks are passed.               '
'--------------------------------------------------------'
Private Function OperandIsIm8(c As String) As Boolean
  On Error GoTo IsNot

  'Check type depending on first symbol
  If TestCharset(Left(c, 1), "-0123456789") Then
    '--------------'
    '--- NUMBER ---'
    '--------------'

    Dim s As String, testval As Long, minus As Boolean
    If Len(c) = 0 Then GoTo IsNot
    s = UCase(c)
    minus = False
    If Left(s, 1) = "-" Then
      If Len(s) = 1 Then GoTo IsNot
      s = Mid(s, 2)
      minus = True
    End If
    If Right(s, 1) = "H" Or Right(s, 1) = "B" Then If Len(s) = 1
Then GoTo IsNot
    'Check charset and try to convert (overflow will be trapped)
    If Right(s, 1) = "H" Then
      s = Left(s, Len(s) - 1)
      If Not TestCharset(Left(s, 1), "0123456789") Then GoTo IsNot
      If Not TestCharset(s, "0123456789ABCDEF") Then GoTo IsNot
      testval = Hex2Dec(s)
    ElseIf Right(s, 1) = "B" Then
      s = Left(s, Len(s) - 1)
      If Not TestCharset(s, "01") Then GoTo IsNot
      testval = Bin2Dec(s)
    Else
      If Not TestCharset(s, "0123456789") Then GoTo IsNot
      testval = CLng(s)
    End If
    'Check range
    If minus Then testval = -testval
    If testval < -128 Or testval > 255 Then GoTo IsNot
    'Everything is fine
    OperandIsIm8 = True

  ElseIf UCase(c) = "A" Or UCase(c) = "B" Or UCase(c) = "C" Or
UCase(c) = "D" Or UCase(c) = "E" Then
    OperandIsIm8 = False
  Else
    '----------------'
    '--- VARIABLE ---'
    '----------------'

    OperandIsIm8 = TestCharset(c, CharsetLabel)
  End If

  Exit Function
IsNot:
  OperandIsIm8 = False
End Function

'---------------------------------------------------------'
' Private Function OperandIsIm16(c As String) As Boolean '
'                                                         '
' PARAMETERS:                                             '
'   c - assembly language operand to be tested            '
'                                                         '
' RETURNS: True if operand is a 16-bit immediate          '
'   constant, and range checks are passed.                '
'---------------------------------------------------------'
Private Function OperandIsIm16(c As String) As Boolean
  On Error GoTo IsNot

  'Check type depending on first symbol
  If TestCharset(Left(c, 1), "-0123456789") Then
    '--------------'
    '--- NUMBER ---'
    '--------------'

    Dim s As String, testval As Long, minus As Boolean
    If Len(c) = 0 Then GoTo IsNot
    s = UCase(c)
    minus = False
    If Left(s, 1) = "-" Then
      If Len(s) = 1 Then GoTo IsNot
      s = Mid(s, 2)
      minus = True
    End If
    If Right(s, 1) = "H" Or Right(s, 1) = "B" Then If Len(s) = 1
Then GoTo IsNot
    'Check charset and try to convert (overflow will be trapped)
    If Right(s, 1) = "H" Then
      s = Left(s, Len(s) - 1)
      If Not TestCharset(Left(s, 1), "0123456789") Then GoTo IsNot
      If Not TestCharset(s, "0123456789ABCDEF") Then GoTo IsNot
      testval = Hex2Dec(s)
    ElseIf Right(s, 1) = "B" Then
      s = Left(s, Len(s) - 1)
      If Not TestCharset(s, "01") Then GoTo IsNot
```

```
      testval = Bin2Dec(s)
    Else
      If Not TestCharset(s, "0123456789") Then GoTo IsNot
      testval = CLng(s)
    End If
    'Check range
    If minus Then testval = -testval
    If testval < -32768 Or testval > 65535 Then GoTo IsNot
    'Everything is fine
    OperandIsIm16 = True

  ElseIf UCase(c) = "A" Or UCase(c) = "B" Or UCase(c) = "C" Or
UCase(c) = "D" Or UCase(c) = "E" Then
    OperandIsIm16 = False
  Else
    '---------------'
    '--- VARIABLE ---'
    '---------------'

    OperandIsIm16 = TestCharset(c, CharsetLabel)
  End If

  Exit Function
IsNot:
  OperandIsIm16 = False
End Function

'-----------------------------------------------------------'
' Private Function CIm8(c As String, adr As Long, RelTo     '
'   As Long, cl As Integer, ByRef p As TPrg) As Long        '
'                                                            '
' PARAMETERS:                                                '
'   c - operand to be compiled                               '
'   adr - offset of the operand in code (for backpatch)      '
'   RelTo - offset of the byte that this is relative to.     '
'   cl - code line containing the operand (for errlog)       '
'   p - program containing ErrLog etc                        '
'                                                            '
' RETURNS: unsigned value of constant represented by c.      '
'                                                            '
' NOTES: will backpatch the code if operand is               '
'   represented by a variable name.                          '
'-----------------------------------------------------------'
Private Function CIm8(c As String, adr As Long, RelTo As Long, cl
As Integer) As Long

  'Set an error trap and hope we checked C before calling this
  On Error GoTo HoustonWeVeGotAProblem

  'Check type depending on first symbol
  If TestCharset(Left(c, 1), "-0123456789") Then
    '--------------'
    '--- NUMBER ---'
    '--------------'

    'Prepare
    Dim s As String, minus As Boolean, n As Long
    s = UCase(c)
    minus = False
    If Left(s, 1) = "-" Then
      minus = True
      s = Mid(s, 2)
    End If
    'Convert
    If Right(s, 1) = "H" Then
      n = Hex2Dec(Left(s, Len(s) - 1))
    ElseIf Right(s, 1) = "B" Then
      n = Bin2Dec(Left(s, Len(s) - 1))
    Else
      n = CLng(s)
    End If
    'Deal with minus sign
    If minus Then n = 256 - n
    'Return result
    CIm8 = n

  ElseIf adr >= 0 Then
    '---------------'
    '--- VARIABLE ---'
    '---------------'

    'Return 0
    CIm8 = 0
    'Add backpatching instructions
    ReDim Preserve Proj.P.Backpatch(-1 To UBound(Proj.P.Backpatch)
+ 1)
    Proj.P.Backpatch(UBound(Proj.P.Backpatch)).IsDW = False
    Proj.P.Backpatch(UBound(Proj.P.Backpatch)).Name = c
    Proj.P.Backpatch(UBound(Proj.P.Backpatch)).Addr = adr
    Proj.P.Backpatch(UBound(Proj.P.Backpatch)).CodeLine = cl
    Proj.P.Backpatch(UBound(Proj.P.Backpatch)).RelTo = RelTo
  Else
    CIm8 = 0
  End If

  Exit Function
HoustonWeVeGotAProblem:
  CIm8 = -1 'should not happen unless OperandIsIm8 not called
before
  Call Err("pCompile.CIm8: could not convert given text to 8 bit
immediate. Contact the author.")
End Function

'-----------------------------------------------------------'
' Private Function CIm16(c As String, adr As Long, cl       '
'   As Integer, ByRef p As TPrg) As Long                     '
'                                                            '
' PARAMETERS:                                                '
'   c - operand to be compiled                               '
'   adr - offset of the operand in code (for backpatch)      '
'   cl - code line containing the operand (for errlog)       '
'   p - program containing ErrLog etc                        '
'                                                            '
' RETURNS: unsigned value of constant represented by c.      '
'                                                            '
' NOTES: will backpatch the code if operand is               '
'   represented by a variable name.                          '
'-----------------------------------------------------------'
Private Function CIm16(c As String, adr As Long, cl As Integer) As
Long

  'Set an error trap and hope we checked C before calling this
  On Error GoTo HoustonWeVeGotAProblem

  'Check type depending on first symbol
  If TestCharset(Left(c, 1), "-0123456789") Then
    '--------------'
    '--- NUMBER ---'
    '--------------'

    'Prepare
    Dim s As String, minus As Boolean, n As Long
    s = UCase(c)
    minus = False
    If Left(s, 1) = "-" Then
      minus = True
      s = Mid(s, 2)
    End If
    'Convert
    If Right(s, 1) = "H" Then
      n = Hex2Dec(Left(s, Len(s) - 1))
    ElseIf Right(s, 1) = "B" Then
      n = Bin2Dec(Left(s, Len(s) - 1))
    Else
      n = CLng(s)
    End If
    'Deal with minus sign
    If minus Then n = 65536 - n
    'Return result
    CIm16 = n

  ElseIf adr >= 0 Then
    '---------------'
    '--- VARIABLE ---'
    '---------------'

    'Return 0
    CIm16 = 0
    'Add backpatching instructions
    ReDim Preserve Proj.P.Backpatch(-1 To UBound(Proj.P.Backpatch)
+ 1)
    Proj.P.Backpatch(UBound(Proj.P.Backpatch)).IsDW = True
    Proj.P.Backpatch(UBound(Proj.P.Backpatch)).Name = c
    Proj.P.Backpatch(UBound(Proj.P.Backpatch)).Addr = adr
    Proj.P.Backpatch(UBound(Proj.P.Backpatch)).CodeLine = cl
  Else
    CIm16 = 0
  End If

  Exit Function
HoustonWeVeGotAProblem:
  CIm16 = -1  'should not happen unless OperandIsIm16 not called
before
  Call Err("pCompile.CIm16: could not convert given text to 16
bit immediate. Contact the author.")
End Function
```

```
'----------------------------------------------------'
' Private Sub AddErr(ByRef ErrL As PErrLog, Message As '
'   String, LineNum As Integer, ErrCode As String)    '
'                                                      '
' DESCRIPTION: Appends an error to the given error log '
'                                                      '
' PARAMETERS:                                          '
'   ErrL - error log to add to                         '
'   Message - error message to be displayed            '
'   LineNum - offending line number in source code     '
'   ErrCode - error code to manage help                '
'----------------------------------------------------'
Private Sub AddErr(Message As String, LineNum As Integer, ErrCode
As String)
  Dim i As Integer
  'Add element
  ReDim Preserve Proj.P.ErrL.lError(-1 To
UBound(Proj.P.ErrL.lError) + 1)
  ReDim Preserve Proj.P.ErrL.sError(-1 To
UBound(Proj.P.ErrL.sError) + 1)
  ReDim Preserve Proj.P.ErrL.nError(-1 To
UBound(Proj.P.ErrL.nError) + 1)
  'Set new element
  Proj.P.ErrL.lError(UBound(Proj.P.ErrL.lError)) = LineNum
  Proj.P.ErrL.sError(UBound(Proj.P.ErrL.sError)) = Message
  Proj.P.ErrL.nError(UBound(Proj.P.ErrL.nError)) = ErrCode
End Sub

'----------------------------------------------------'
' Private Sub AddWng(ByRef ErrL As PErrLog, Message As '
'   String, LineNum As Integer, WngCode As String)    '
'                                                      '
' DESCRIPTION: Appends a warning to given error log    '
'                                                      '
' PARAMETERS:                                          '
'   ErrL - error log to add to                         '
'   Message - warning message to be displayed          '
'   LineNum - offending line number in source code     '
'   WngCode - warning code to manage help              '
'----------------------------------------------------'
Private Sub AddWng(Message As String, LineNum As Integer, WngCode
As String)
  Dim i As Integer
  'Add element
  ReDim Preserve Proj.P.ErrL.lWarning(-1 To
UBound(Proj.P.ErrL.lWarning) + 1)
  ReDim Preserve Proj.P.ErrL.sWarning(-1 To
UBound(Proj.P.ErrL.sWarning) + 1)
  ReDim Preserve Proj.P.ErrL.nWarning(-1 To
UBound(Proj.P.ErrL.nWarning) + 1)
  'Set new element
  Proj.P.ErrL.lWarning(UBound(Proj.P.ErrL.lWarning)) = LineNum
  Proj.P.ErrL.sWarning(UBound(Proj.P.ErrL.sWarning)) = Message
  Proj.P.ErrL.nWarning(UBound(Proj.P.ErrL.nWarning)) = WngCode
End Sub
```

```
'----------------------------------------------------'
' Private Function CleanSpaces(ByVal src As String)    '
'                                                      '
' DESCRIPTION: Converts all tab/" "/"," sequences with '
'   a single space character.                          '
'                                                      '
' PARAMETERS:                                          '
'   src - string to process                            '
'                                                      '
' RETURNS: processed string with converted characters  '
'----------------------------------------------------'
Private Function CleanSpaces(ByVal src As String)
  Dim i As Integer, b As Boolean
  Dim s As String, c As String

  b = True
  s = ""
  For i = 1 To Len(src)
    c = Mid(src, i, 1)
    If Asc(c) > 127 Then GoTo skp
    If b Then
      If c = Chr(9) Or c = " " Or c = "," Then
        s = s + " "
        b = False
      Else
        s = s + c
      End If
    Else
      If c <> Chr(9) And c <> " " And c <> "," Then
        s = s + c
        b = True
      End If
    End If
skp:
  Next

  CleanSpaces = s
End Function

'Warnings:
'WC1001, WC1002

'Errors:
'EC1001, EC1002, EC1003, EC1004

'EC2001 - operand takes X operands, not Y.
'EC2002
'EC2005 - Syntax error in operand OR opcode and operand
incompatible. Offending operand: 'Y'.
'EC2006
'EC2007 - 16 bit immediate constant is out of range.
'EC2008, EC2009, EC2010, EC2011, EC2013, EC2014,
'EC2015, EC2016, EC2017, EC2018, EC2019

'EC3001, EC3002
```

22.5. pExec

```
Option Explicit

'-----------------------------------------'
' Public declarations in this module:     '
'                                         '
' VARIABLES:                              '
'   CPU - current CPU state               '
'                                         '
' CONSTANTS:                              '
'   flX - flag constants for CPU.FLAGS    '
'                                         '
' PROCEDURES:                             '
'   exeInit - initialises this module     '
'   Tick - advance simulation by 1 tick   '
'   DI2Str - CStr(microinstruction)       '
'-----------------------------------------'

'Decoded microinstruction
Private Type TpDI
  Sig1 As Long   'VB has no built-in support for
  Sig2 As Long   '  64-bit integers, so have to split them
  nToIDB As Long
  nAluOpNum As Integer
  nJmpCond As Integer
  nAdrMul As Integer
  nAluSh As Integer
  nIntIS As Integer
End Type

'CPU state structure
Public Type TpCPU
  '-----------------'
  '--- Registers ---'
  '-----------------'

  'General purpose
  A As Long
  R(0 To 3) As Long
  'Special purpose
  IP As Long
  SP As Long
  FLAGS As Long
  'Internal
  MAR As Long
  MDR As Long
  CIB As String
```

```
  DIB() As TpDI      'decoded buffer
  Fetch As Boolean
  FREM As Integer
  fremMem As Boolean   'invisible register to allow for FREM+
  IS As Long

  '----------------'
  '--- Execution ---'
  '----------------'
  eSelectedReg As Integer  '0..X: b,c,d,e,sp,ip
  eIDB As Long
  eIAB As Long
  eDP As Integer      'which microinstruction doing
  eEDB As Long
  eEAB As Long
  Breakpoint() As Long 'address at which we are when first exec tick
End Type

'Signal numbers
Private Const reg_sb = 0
Private Const reg_sc = 1
Private Const reg_sd = 2
Private Const reg_se = 3
Private Const reg_ssp = 4
Private Const reg_sip = 5
Private Const reg_r = 6
Private Const reg_w = 7
Private Const reg_ipi = 8
Private Const reg_spi = 9
Private Const reg_spd = 10
Private Const adr_br = 11
Private Const adr_sr = 12
Private Const adr_im = 13
Private Const adr_c = 14
Private Const lea_ad = 15
Private Const acc_r = 16
Private Const acc_w = 17
Private Const flg_r = 18
Private Const flg_w = 19
Private Const ctl_mr = 20
Private Const ctl_mw = 21
Private Const ctl_pr = 22
Private Const ctl_pw = 23
Private Const mdr_ri = 24
Private Const mdr_re = 25
Private Const mdr_wi = 26
Private Const mdr_we = 27
Private Const mar_ri = 28
Private Const mar_re = 29
Private Const mar_wi = 30 'skip 31 because there may be errors
Private Const mar_we = 32 '  associated with sign
Private Const alu_swp = 33
Private Const ctl_halt = 34
Private Const lea_da = 35
Private Const flg_stz = 36
Private Const flg_clz = 37
Private Const flg_stc = 38
Private Const flg_clc = 39
Private Const flg_sto = 40
Private Const flg_clo = 41
Private Const flg_sts = 42
Private Const flg_cls = 43
Private Const flg_sti = 44
Private Const flg_cli = 45

Private Const op_alu_sh = 58    'these indicate that the
Private Const op_jmp_cond = 59  'next operand in a call to
Private Const op_idb_im = 60    'SFlg is a respective parameter
Private Const op_adr_mm = 61
Private Const op_alu_c = 62

'Flag constants
Public Const flZ = 1
Public Const flS = 2
Public Const flO = 4
Public Const flC = 8
Public Const flI = 16
Public Const flN = 256
Public Const flP = 512

'Fetch simplification arrays
Public InstructionLen(0 To 255) As Integer 'public because other
Public InstructionMem(0 To 255) As Boolean 'units may need the
lengths
' "Massive decode" simplification
Private AluOpType1Hex(0 To 7) As Integer
Private AluOpType2Hex(0 To 13) As Integer
Private AluOpType3Hex1(0 To 4) As Integer
Private AluOpType3Hex2(0 To 4) As Integer
Private JmpOpType4Hex(0 To 11) As Integer

'-------------------------'
' Public Sub exeInit()    '
'                         '
' Initializes this module '
'-------------------------'
Public Sub exeInit()
  Dim i As Integer
  'Reset arrays
  For i = 0 To 255
    InstructionLen(i) = 1 'let it fetch and try to decode
    InstructionMem(i) = False
  Next
  'Instruction lengths
  'Cannot do it in a loop - there are holes everywhere
  InstructionLen(&H0) = 1
  InstructionLen(&H1) = 1
  InstructionLen(&H2) = 1
  InstructionLen(&H3) = 1
  InstructionLen(&H4) = 1
  InstructionLen(&H5) = 1
  InstructionLen(&H6) = 1
  InstructionLen(&H7) = 1
  InstructionLen(&HF) = 1
  InstructionLen(&H10) = 1
  InstructionLen(&H11) = 1
  InstructionLen(&H13) = 1
  InstructionLen(&H14) = 1
  InstructionLen(&H15) = 1
  InstructionLen(&H16) = 1
  InstructionLen(&H17) = 1
  InstructionLen(&H54) = 1
  InstructionLen(&H55) = 1
  InstructionLen(&H56) = 1
  InstructionLen(&H57) = 1
  InstructionLen(&H64) = 1
  InstructionLen(&H65) = 1
  InstructionLen(&H66) = 1
  InstructionLen(&H67) = 1
  InstructionLen(&H72) = 1
  InstructionLen(&H73) = 1
  InstructionLen(&H75) = 1
  InstructionLen(&H76) = 1
  InstructionLen(&H77) = 1
  InstructionLen(&H8F) = 1
  InstructionLen(&H9F) = 1
  InstructionLen(&HA0) = 1
  InstructionLen(&HA1) = 1
  InstructionLen(&HA2) = 1
  InstructionLen(&HA3) = 1
  InstructionLen(&HA4) = 1
  InstructionLen(&HA5) = 1
  InstructionLen(&HA6) = 1
  InstructionLen(&HA7) = 1
  InstructionLen(&HA8) = 1
  InstructionLen(&HA9) = 1
  InstructionLen(&HAA) = 1
  InstructionLen(&HAB) = 1
  InstructionLen(&HAC) = 1
  InstructionLen(&HAD) = 1
  InstructionLen(&HAE) = 1
  InstructionLen(&HAF) = 1
  InstructionLen(&HBC) = 1
  InstructionLen(&HBD) = 1
  InstructionLen(&HBE) = 1
  InstructionLen(&HBF) = 1
  InstructionLen(&HDC) = 1
  InstructionLen(&HDD) = 1
  InstructionLen(&HDE) = 1
  InstructionLen(&HDF) = 1
  InstructionLen(&HE6) = 1
  InstructionLen(&HE7) = 1
  InstructionLen(&HF0) = 1
  InstructionLen(&HF1) = 1
  InstructionLen(&HF2) = 1
  InstructionLen(&HF3) = 1
  InstructionLen(&HF4) = 1
  InstructionLen(&HF5) = 1
  InstructionLen(&HF6) = 1
  InstructionLen(&HF7) = 1

  InstructionLen(&H25) = 2
  InstructionLen(&H35) = 2
  InstructionLen(&H74) = 2
  InstructionLen(&H80) = 2
  InstructionLen(&H83) = 2
```

```
InstructionLen(&H86) = 2
InstructionLen(&H89) = 2
InstructionLen(&H8C) = 2
InstructionLen(&H90) = 2
InstructionLen(&H93) = 2
InstructionLen(&H96) = 2
InstructionLen(&H99) = 2
InstructionLen(&H9C) = 2
InstructionLen(&HB0) = 2
InstructionLen(&HB3) = 2
InstructionLen(&HB6) = 2
InstructionLen(&HB9) = 2
InstructionLen(&HC0) = 2
InstructionLen(&HC1) = 2
InstructionLen(&HC2) = 2
InstructionLen(&HC3) = 2
InstructionLen(&HC4) = 2
InstructionLen(&HC5) = 2
InstructionLen(&HC6) = 2
InstructionLen(&HC7) = 2
InstructionLen(&HD0) = 2
InstructionLen(&HD1) = 2
InstructionLen(&HD2) = 2
InstructionLen(&HD3) = 2
InstructionLen(&HD4) = 2
InstructionLen(&HD6) = 2
InstructionLen(&HD8) = 2
InstructionLen(&HD9) = 2
InstructionLen(&HDA) = 2
InstructionLen(&HDB) = 2
InstructionLen(&HE4) = 2
InstructionLen(&HE5) = 2

InstructionLen(&H12) = 3
InstructionLen(&H20) = 3
InstructionLen(&H21) = 3
InstructionLen(&H22) = 3
InstructionLen(&H23) = 3
InstructionLen(&H24) = 3
InstructionLen(&H30) = 3
InstructionLen(&H31) = 3
InstructionLen(&H32) = 3
InstructionLen(&H33) = 3
InstructionLen(&H34) = 3
InstructionLen(&H81) = 3
InstructionLen(&H84) = 3
InstructionLen(&H87) = 3
InstructionLen(&H8A) = 3
InstructionLen(&H8D) = 3
InstructionLen(&H91) = 3
InstructionLen(&H94) = 3
InstructionLen(&H97) = 3
InstructionLen(&H9A) = 3
InstructionLen(&H9D) = 3
InstructionLen(&HB1) = 3
InstructionLen(&HB4) = 3
InstructionLen(&HB7) = 3
InstructionLen(&HBA) = 3
InstructionLen(&HD5) = 3
InstructionLen(&HE0) = 3
InstructionLen(&HE1) = 3
InstructionLen(&HE2) = 3
InstructionLen(&HE3) = 3

InstructionLen(&HD7) = 4

InstructionLen(&H8) = -1    'minus saves code lines - see below
InstructionLen(&H9) = -1
InstructionLen(&HA) = -1
InstructionLen(&HB) = -1
InstructionLen(&HC) = -1
InstructionLen(&H40) = -1
InstructionLen(&H41) = -1
InstructionLen(&H42) = -1
InstructionLen(&H43) = -1
InstructionLen(&H50) = -1
InstructionLen(&H51) = -1
InstructionLen(&H52) = -1
InstructionLen(&H53) = -1
InstructionLen(&H60) = -1
InstructionLen(&H61) = -1
InstructionLen(&H62) = -1
InstructionLen(&H63) = -1
InstructionLen(&H70) = -1
InstructionLen(&H71) = -1
InstructionLen(&H82) = -1
InstructionLen(&H85) = -1
InstructionLen(&H88) = -1
InstructionLen(&H8B) = -1
InstructionLen(&H8E) = -1
InstructionLen(&H92) = -1
InstructionLen(&H95) = -1
InstructionLen(&H98) = -1
InstructionLen(&H9B) = -1
InstructionLen(&H9E) = -1
InstructionLen(&HB2) = -1
InstructionLen(&HB5) = -1
InstructionLen(&HB8) = -1
InstructionLen(&HBB) = -1

InstructionLen(&H26) = -2
InstructionLen(&H27) = -2
InstructionLen(&H36) = -2
InstructionLen(&H37) = -2

For i = 0 To 255
  If InstructionLen(i) = -2 Or InstructionLen(i) = -1 Then
    InstructionMem(i) = True
    InstructionLen(i) = -InstructionLen(i)
  End If
Next

'Initialise array AluOpType1Hex
AluOpType1Hex(0) = &HC0
AluOpType1Hex(1) = &HC1
AluOpType1Hex(2) = &HC2
AluOpType1Hex(3) = &HC3
AluOpType1Hex(4) = &HC4
AluOpType1Hex(5) = &HC5
AluOpType1Hex(6) = &HC6
AluOpType1Hex(7) = &HC7
'Initialise array AluOpType2Hex
AluOpType2Hex(0) = &H80
AluOpType2Hex(1) = &H83
AluOpType2Hex(2) = &H86
AluOpType2Hex(3) = &H89
AluOpType2Hex(4) = &H8C
AluOpType2Hex(5) = &H90
AluOpType2Hex(6) = &H93
AluOpType2Hex(7) = &H96
AluOpType2Hex(8) = &H99
AluOpType2Hex(9) = &H9C
AluOpType2Hex(10) = &HB0
AluOpType2Hex(11) = &HB3
AluOpType2Hex(12) = &HB6
AluOpType2Hex(13) = &HB9
'Initialise array AluOpType3Hex
AluOpType3Hex1(0) = &HA0
AluOpType3Hex1(1) = &HA4
AluOpType3Hex1(2) = &HA8
AluOpType3Hex1(3) = &HBC
AluOpType3Hex1(4) = &HDC
AluOpType3Hex2(0) = &HAC
AluOpType3Hex2(1) = &HAD
AluOpType3Hex2(2) = &HAE
AluOpType3Hex2(3) = &HAF
AluOpType3Hex2(4) = &H9F
'Initialise array JmpOpType4Hex
JmpOpType4Hex(0) = &H40
JmpOpType4Hex(1) = &H41
JmpOpType4Hex(2) = &H42
JmpOpType4Hex(3) = &H43
JmpOpType4Hex(4) = &H50
JmpOpType4Hex(5) = &H51
JmpOpType4Hex(6) = &H52
JmpOpType4Hex(7) = &H53
JmpOpType4Hex(8) = &H60
JmpOpType4Hex(9) = &H61
JmpOpType4Hex(10) = &H62
JmpOpType4Hex(11) = &H63
End Sub

'---------------------------------------------'
' Private Function GFlg(di As TpDI, Index As   '
'   Integer) As Boolean
'                                              '
' Returns true if flag Index is set in di     '
'---------------------------------------------'
Private Function GFlg(di As TpDI, Index As Integer) As Boolean
  If Index < 31 Then
    GFlg = ((di.Sig1 And 2 ^ Index) > 0)
  ElseIf (Index >= 32) And (Index <= 62) Then
    GFlg = ((di.Sig2 And 2 ^ (Index - 32)) > 0)
  Else
    Call Errr("pExec.GFlg: index should be between 0 and 62
excluding 31.")
```

```
  End If
End Function

'-----------------------------------------------'
' Private Sub SFlg(di As TpDI, ParamArray arr()) '
'                                               '
' DESCRIPTION: adds the specified signals to the '
'   decoded instruction buffer element.          '
'                                               '
' PARAMETERS:                                   '
'   di - decoded microinstruction to be modigied '
'   arr() - list of signal constants            '
'                                               '
' NOTES: using a signal constant beginning with  '
'   op_ will make SFlg assume that the next     '
'   arr() element is the signal-specific param  '
'   to be stored in di.                         '
'-----------------------------------------------'
Private Sub SFlg(di As TpDI, ParamArray arr())
  Dim i As Integer
  di.nAdrMul = 1
  di.nAluOpNum = -1
  di.nJmpCond = -1
  di.nToIDB = -1
  di.nAluSh = 0
  di.Sig1 = 0
  di.Sig2 = 0
  For i = 0 To UBound(arr)
    'Add signal
    If (arr(i) >= 0) And (arr(i) <= 30) Then
      di.Sig1 = di.Sig1 Or (2 ^ arr(i))
    ElseIf (arr(i) >= 32) And (arr(i) <= 62) Then
      di.Sig2 = di.Sig2 Or (2 ^ (arr(i) - 32))
    End If
    'Check if signal parameter present
    If arr(i) = op_adr_mm Then
      di.nAdrMul = arr(i + 1)
      i = i + 1
    ElseIf arr(i) = op_alu_c Then
      di.nAluOpNum = arr(i + 1)
      i = i + 1
    ElseIf arr(i) = op_idb_im Then
      di.nToIDB = arr(i + 1)
      i = i + 1
    ElseIf arr(i) = op_jmp_cond Then
      di.nJmpCond = arr(i + 1)
      i = i + 1
    ElseIf arr(i) = op_alu_sh Then
      di.nAluSh = arr(i + 1)
      i = i + 1
    End If
  Next
End Sub

'--------------------------------------------'
' Public Sub Tick()                          '
'                                            '
' Advances simulation by one clock tick by   '
' fetching, decoding or executing something. '
'--------------------------------------------'
Public Sub Tick()
  eTick
  'No updates - update manually for more control
End Sub

'--------------------------------------------'
' Public Sub Tick()                          '
'                                            '
' Advances simulation by CPU instruction,    '
' fetching the next one.                     '
'--------------------------------------------'
Public Sub Step()
  Dim wasIS As Integer        'finish step at interrupt
  While Not Proj.CPU.Fetch
    wasIS = Proj.CPU.IS
    eTick
    If wasIS <> Proj.CPU.IS Then Exit Sub
  Wend
  While Proj.CPU.Fetch
    wasIS = Proj.CPU.IS
    eTick
    If wasIS <> Proj.CPU.IS Then Exit Sub
  Wend
  'No updates - update manually for more control
End Sub

'-----------------------------------------------'
' Private Sub eTick()                           '
'                                               '
' Either fetches or executes current instruction, '
' depending on state. Does not update any forms. '
'-----------------------------------------------'
Private Sub eTick()
  If Proj.CPU.Fetch Then
    eFetch
  Else
    eExecute
  End If
  devTick
  Proj.TickCount = Proj.TickCount + 1
End Sub

'-----------------------------------------------'
' Private Sub eFetch()                          '
'                                               '
' Advances simulation by one tick when CPU is   '
' in fetch mode. Does NOT check if it is. After '
' fetching the last byte automatically initiates '
' eDecode, so that decoding takes no further ticks '
'-----------------------------------------------'
Private Sub eFetch()
  Dim b As Byte
  With Proj.CPU

  'Fetch a byte into a temporary location
  b = Proj.RAM(.IP)
  .IP = .IP + 1

  'First loop of fetch cycle
  If .CIB = "" Then
    'How many bytes to fetch
    .FREM = InstructionLen(b)
    .fremMem = InstructionMem(b)

  'Middle loop & recalc of FREM needed
  ElseIf (.FREM = 0) And .fremMem Then
    If (b And 192) = 0 Then .FREM = 3
    If (b And 192) = 64 Then .FREM = 1
    If (b And 192) = 0 Then .FREM = 3
    If (b And 160) = 160 Then .FREM = 3
    If (b And 160) = 128 Then .FREM = 1
    .fremMem = False
  End If

  'Add fetched byte to CIB
  .CIB = .CIB + Chr(b)
  'Update remaining bytes register
  .FREM = .FREM - 1

  'Last loop of fetch cycle
  Dim i As Integer
  If .FREM = 0 And .fremMem = False Then
    .Fetch = False
    eDecode
    'Check breakpoints
    For i = 0 To UBound(.Breakpoint)
      If .IP = .Breakpoint(i) Then
        Proj.Paused = True
        Call fiMain.UpdateAll(True)
      End If
    Next
  End If

  End With
End Sub

'-----------------------------------------------'
' Private Sub eDecode                           '
'                                               '
' DESCRIPTION: Decodes fetched instruction      '
'   from CPU.CIB, placing decoded microprogram  '
'   into CPU.DIB.                               '
'-----------------------------------------------'
Private Sub eDecode()
  With Proj.CPU

  'Get first byte into b
  If .CIB = "" Then
    Call Errr("pExec.execDecode: CIB is empty. Contact the
author.")
  End If
  Dim b As Byte
  b = Asc(Left(.CIB, 1))

  'Empty DIB
  ReDim .DIB(-1 To -1)
```

```
 '---------'
 '- LD/ST -'
 '---------'
 If (b >= &H30) And (b <= &H37) Then b = b - &H10
 If (b >= &H20) And (b <= &H23) Then
   'ld Reg,I
   ReDim .DIB(-1 To 0)
   Call SFlg(.DIB(0), reg_sX(b And 3), reg_w, op_idb_im,
Chr2Dec(Mid(.CIB, 2, 2)))
   Exit Sub
 ElseIf b = &H24 Then
   'ld A,I
   ReDim .DIB(-1 To 0)
   Call SFlg(.DIB(0), acc_w, op_idb_im, Chr2Dec(Mid(.CIB, 2, 2)))
   Exit Sub
 ElseIf b = &H25 Then
   'ld R,R
   b = Asc(Mid(.CIB, 2, 1))
   If (b And 192) = 192 Then
     'ld A,A
     ReDim .DIB(-1 To 0)
     Call SFlg(.DIB(0), acc_r, acc_w)
   ElseIf (b And 192) = 128 Then
     'ld A,Rn
     ReDim .DIB(-1 To 0)
     Call SFlg(.DIB(0), reg_sX(b And 3), reg_r, acc_w)
   ElseIf (b And 192) = 64 Then
     'ld Rn,A
     ReDim .DIB(-1 To 0)
     Call SFlg(.DIB(0), reg_sX((b And 24) / 8), reg_w, acc_r)
   Else
     'ld Rn,Rn
     ReDim .DIB(-1 To 1)
     Call SFlg(.DIB(0), reg_sX(b And 3), reg_r, mdr_wi)
     Call SFlg(.DIB(1), reg_sX((b And 24) / 8), reg_w, mdr_ri)
   End If
   Exit Sub
 ElseIf b = &H26 Then
   'ld R,M
   ReDim .DIB(-1 To -1)
   Call DecodeMem(3)
   b = Asc(Mid(.CIB, 2, 1))
   ReDim Preserve .DIB(-1 To UBound(.DIB) + 2)
   Call SFlg(.DIB(UBound(.DIB) - 1), mar_re, ctl_mr, mdr_we)
   If (b And 4) = 4 Then
     Call SFlg(.DIB(UBound(.DIB)), mdr_ri, acc_w)
   Else
     Call SFlg(.DIB(UBound(.DIB)), mdr_ri, reg_sX(b And 3), reg_w)
   End If
   Exit Sub
 ElseIf b = &H27 Then
   'ld M,R
   ReDim .DIB(-1 To -1)
   Call DecodeMem(3)
   b = Asc(Mid(.CIB, 2, 1))
   ReDim Preserve .DIB(-1 To UBound(.DIB) + 2)
   If (b And 4) = 4 Then
     Call SFlg(.DIB(UBound(.DIB) - 1), acc_r, mdr_wi)
   Else
     Call SFlg(.DIB(UBound(.DIB) - 1), reg_sX(b And 3), reg_r,
mdr_wi)
   End If
   Call SFlg(.DIB(UBound(.DIB)), mar_re, mdr_re, ctl_mw)
   Exit Sub
 End If

 '----------'
 '- TYPE 1 -'
 '----------'
 Dim i As Integer
 For i = 0 To 7
   If (b = AluOpType1Hex(i)) Then
     b = Asc(Mid(.CIB, 2, 1))
     If (b And 128) = 0 Then
       'op A,Rn
       ReDim .DIB(-1 To 0)
       Call SFlg(.DIB(0), reg_sX((b And 96) \ 32), reg_r, op_alu_c,
19 + i, acc_w)
     Else
       ReDim .DIB(-1 To 0)
       If (b And 16) = 0 Then
         'op Rn,N
         Call SFlg(.DIB(0), reg_sX((b And 96) \ 32), reg_r,
op_alu_sh, b And 15, op_alu_c, 27 + i, reg_w)
       Else
         'op A,N
         Call SFlg(.DIB(0), acc_r, op_alu_sh, b And 15, op_alu_c,
27 + i, acc_w)
       End If
     End If
     Exit Sub
   End If
 Next

 '----------'
 '- TYPE 2 -'
 '----------'
 For i = 0 To 13
   If (b = AluOpType2Hex(i)) Then
     b = Asc(Mid(.CIB, 2, 1))
     If (b And 4) = 0 Then
       If (b And 8) = 0 Then
         'op A,Rn
         ReDim .DIB(-1 To 0)
         Call SFlg(.DIB(0), reg_sX(b And 3), reg_r, acc_w,
op_alu_c, i)
       Else
         'op Rn,A
         ReDim .DIB(-1 To 1)
         Call SFlg(.DIB(0), reg_sX(b And 3), reg_r, op_alu_c, i,
alu_swp)
         Call SFlg(.DIB(1), reg_sX(b And 3), reg_w)
       End If
     Else
       'op A,A
       ReDim .DIB(-1 To 0)
       Call SFlg(.DIB(0), acc_r, acc_w, op_alu_c, i)
     End If
     Exit Sub
   ElseIf (b = AluOpType2Hex(i) + 1) Then
     'op A,I
     ReDim .DIB(-1 To 0)
     Call SFlg(.DIB(0), acc_w, op_alu_c, i, op_idb_im,
Chr2Dec(Mid(.CIB, 2, 2)))
     Exit Sub
   ElseIf (b = AluOpType2Hex(i) + 2) Then
     'op A,M
     ReDim .DIB(-1 To -1)
     Call DecodeMem(2)
     ReDim Preserve .DIB(-1 To UBound(.DIB) + 2)
     Call SFlg(.DIB(UBound(.DIB) - 1), mar_re, ctl_mr, mdr_we)
     Call SFlg(.DIB(UBound(.DIB)), mdr_ri, acc_w, op_alu_c, i)
     Exit Sub
   End If
 Next

 '----------'
 '- TYPE 3 -'
 '----------'
 For i = 0 To 4
   If (b = AluOpType3Hex2(i)) Then
     'op A
     ReDim .DIB(-1 To 0)
     Call SFlg(.DIB(0), acc_r, op_alu_c, 14 + i, acc_w)
     Exit Sub
   ElseIf (b >= AluOpType3Hex1(i)) And (b <= AluOpType3Hex1(i) +
3) Then
     'op Rn
     ReDim .DIB(-1 To 0)
     Call SFlg(.DIB(0), reg_sX(b And 3), reg_r, op_alu_c, 14 + i,
reg_w)
     Exit Sub
   End If
 Next

 '----------'
 '- TYPE 4 -'
 '----------'
 For i = 0 To 11
   'jXX M
   If b = JmpOpType4Hex(i) Then
     ReDim .DIB(-1 To 0)
     Call SFlg(.DIB(0), op_jmp_cond, i)
     Call DecodeMem(2)
     ReDim Preserve .DIB(-1 To UBound(.DIB) + 1)
     Call SFlg(.DIB(UBound(.DIB)), mar_ri, lea_ad, reg_sip,
reg_w)
     Exit Sub
   End If
 Next

 '--- JMP ---'
 If b = &H70 Then
   ReDim .DIB(-1 To -1)
```

```
  Call DecodeMem(2)
  ReDim Preserve .DIB(-1 To UBound(.DIB) + 1)
  Call SFlg(.DIB(UBound(.DIB)), mar_ri, lea_ad, reg_sip, reg_w)
  Exit Sub
End If

'--- CALL ---'
If b = &H71 Then
  ReDim .DIB(-1 To 2)
  Call SFlg(.DIB(0), reg_ssp, adr_sr, adr_c, mar_wi)
  Call SFlg(.DIB(1), reg_sip, reg_r, mdr_wi, reg_spi)
  Call SFlg(.DIB(2), mdr_re, mar_re, ctl_mw)
  Call DecodeMem(2)
  ReDim Preserve .DIB(-1 To UBound(.DIB) + 1)
  Call SFlg(.DIB(UBound(.DIB)), mar_ri, lea_ad, reg_sip, reg_w)
  Exit Sub
End If

'--- RET ---'
If b = &H72 Then
  ReDim .DIB(-1 To 2)
  Call SFlg(.DIB(0), reg_spd, reg_ssp, adr_sr, adr_c, mar_wi)
  Call SFlg(.DIB(1), mar_re, ctl_mr, mdr_we)
  Call SFlg(.DIB(2), reg_sip, mdr_ri, reg_w)
  Exit Sub
End If

'--- HALT ---'
If b = &H75 Then
  ReDim .DIB(-1 To 0)
  Call SFlg(.DIB(0), ctl_halt)
  Exit Sub
End If

'--- PUSH ---'
If (b >= &H0) And (b <= &H3) Then
  'push Rn
  ReDim .DIB(-1 To 2)
  Call SFlg(.DIB(0), reg_ssp, adr_sr, adr_c, mar_wi)
  Call SFlg(.DIB(1), reg_sX(b), reg_r, mdr_wi)
  Call SFlg(.DIB(2), mdr_re, mar_re, ctl_mw, reg_spi)
  Exit Sub
ElseIf b = &H10 Then
  'push A
  ReDim .DIB(-1 To 2)
  Call SFlg(.DIB(0), reg_ssp, adr_sr, adr_c, mar_wi)
  Call SFlg(.DIB(1), acc_r, mdr_wi)
  Call SFlg(.DIB(2), mdr_re, mar_re, ctl_mw, reg_spi)
  Exit Sub
ElseIf b = &H12 Then
  'push I
  ReDim .DIB(-1 To 2)
  Call SFlg(.DIB(0), reg_ssp, adr_sr, adr_c, mar_wi)
  Call SFlg(.DIB(1), op_idb_im, Chr2Dec(Mid(.CIB, 2, 2)), mdr_wi)
  Call SFlg(.DIB(2), mdr_re, mar_re, ctl_mw, reg_spi)
  Exit Sub
ElseIf b = &H13 Then
  'pushpc
  ReDim .DIB(-1 To 2)
  Call SFlg(.DIB(0), reg_ssp, adr_sr, adr_c, mar_wi)
  Call SFlg(.DIB(1), reg_sip, reg_r, mdr_wi)
  Call SFlg(.DIB(2), mdr_re, mar_re, ctl_mw, reg_spi)
  Exit Sub
ElseIf b = &H14 Then
  'pushsp
  ReDim .DIB(-1 To 2)
  Call SFlg(.DIB(0), reg_ssp, adr_sr, adr_c, mar_wi)
  Call SFlg(.DIB(1), reg_ssp, reg_r, mdr_wi)
  Call SFlg(.DIB(2), mdr_re, mar_re, ctl_mw, reg_spi)
  Exit Sub
ElseIf b = &H15 Then
  'pushfl
  ReDim .DIB(-1 To 2)
  Call SFlg(.DIB(0), reg_ssp, adr_sr, adr_c, mar_wi)
  Call SFlg(.DIB(1), flg_r, mdr_wi)
  Call SFlg(.DIB(2), mdr_re, mar_re, ctl_mw, reg_spi)
  Exit Sub
End If

'--- POP ---'
If (b >= &H4) And (b <= &H7) Then
  'pop Rn
  ReDim .DIB(-1 To 2)
  Call SFlg(.DIB(0), reg_spd, reg_ssp, adr_sr, adr_c, mar_wi)
  Call SFlg(.DIB(1), mar_re, ctl_mr, mdr_we)
  Call SFlg(.DIB(2), reg_sX(b And 3), mdr_ri, reg_w)
  Exit Sub
ElseIf b = &H11 Then
  'pop A
  ReDim .DIB(-1 To 2)
  Call SFlg(.DIB(0), reg_spd, reg_ssp, adr_sr, adr_c, mar_wi)
  Call SFlg(.DIB(1), mar_re, ctl_mr, mdr_we)
  Call SFlg(.DIB(2), mdr_ri, acc_w)
  Exit Sub
ElseIf b = &H16 Then
  'popsp
  ReDim .DIB(-1 To 2)
  Call SFlg(.DIB(0), reg_spd, reg_ssp, adr_sr, adr_c, mar_wi)
  Call SFlg(.DIB(1), mar_re, ctl_mr, mdr_we)
  Call SFlg(.DIB(2), reg_ssp, mdr_ri, reg_w)
  Exit Sub
ElseIf b = &H17 Then
  'popfl
  ReDim .DIB(-1 To 2)
  Call SFlg(.DIB(0), reg_spd, reg_ssp, adr_sr, adr_c, mar_wi)
  Call SFlg(.DIB(1), mar_re, ctl_mr, mdr_we)
  Call SFlg(.DIB(2), mdr_ri, flg_w)
  Exit Sub
End If

'--- SP2B ---'
If b = &HF Then
  ReDim .DIB(-1 To 1)
  Call SFlg(.DIB(0), reg_ssp, reg_r, mdr_wi)
  Call SFlg(.DIB(1), mdr_ri, reg_sb, reg_w)
  Exit Sub
End If

'--- LEA ---'
If (b >= &H8) And (b <= &HB) Then
  ReDim .DIB(-1 To -1)
  Call DecodeMem(2)
  ReDim Preserve .DIB(-1 To UBound(.DIB) + 1)
  Call SFlg(.DIB(UBound(.DIB)), mar_ri, lea_ad, reg_sX(b And 3),
reg_w)
  Exit Sub
ElseIf b = &H2F Then
  ReDim .DIB(-1 To -1)
  Call DecodeMem(2)
  ReDim Preserve .DIB(-1 To UBound(.DIB) + 1)
  Call SFlg(.DIB(UBound(.DIB)), mar_ri, lea_ad, acc_w)
  Exit Sub
End If

'--- XCHG ---'
If (b >= &HF0) And (b <= &HF3) Then
  'xchg A,Rn
  ReDim .DIB(-1 To 2)
  Call SFlg(.DIB(0), acc_r, mdr_wi)
  Call SFlg(.DIB(1), reg_sX(b And 3), reg_r, acc_w)
  Call SFlg(.DIB(2), mdr_ri, reg_sX(b And 3), reg_w)
  Exit Sub
ElseIf b = &HE6 Then
  'xchg c,d
  ReDim .DIB(-1 To 3)
  Call SFlg(.DIB(0), reg_sc, reg_r, mdr_wi)
  Call SFlg(.DIB(1), reg_sd, reg_r, lea_da, mar_wi)
  Call SFlg(.DIB(2), mdr_ri, reg_sd, reg_w)
  Call SFlg(.DIB(3), mar_ri, lea_ad, reg_sc, reg_w)
  Exit Sub
ElseIf b = &HE7 Then
  'xchg c,e
  ReDim .DIB(-1 To 3)
  Call SFlg(.DIB(0), reg_sc, reg_r, mdr_wi)
  Call SFlg(.DIB(1), reg_se, reg_r, lea_da, mar_wi)
  Call SFlg(.DIB(2), mdr_ri, reg_se, reg_w)
  Call SFlg(.DIB(3), mar_ri, lea_ad, reg_sc, reg_w)
  Exit Sub
ElseIf b = &HF4 Then
  'xchg d,e
  ReDim .DIB(-1 To 3)
  Call SFlg(.DIB(0), reg_sd, reg_r, mdr_wi)
  Call SFlg(.DIB(1), reg_se, reg_r, lea_da, mar_wi)
  Call SFlg(.DIB(2), mdr_ri, reg_se, reg_w)
  Call SFlg(.DIB(3), mar_ri, lea_ad, reg_sd, reg_w)
  Exit Sub
ElseIf b = &HF5 Then
  'xchg b,c
  ReDim .DIB(-1 To 3)
  Call SFlg(.DIB(0), reg_sb, reg_r, mdr_wi)
  Call SFlg(.DIB(1), reg_sc, reg_r, lea_da, mar_wi)
  Call SFlg(.DIB(2), mdr_ri, reg_sc, reg_w)
  Call SFlg(.DIB(3), mar_ri, lea_ad, reg_sb, reg_w)
  Exit Sub
ElseIf b = &HF6 Then
  'xchg b,d
```

```
    ReDim .DIB(-1 To 3)
    Call SFlg(.DIB(0), reg_sb, reg_r, mdr_wi)
    Call SFlg(.DIB(1), reg_sd, reg_r, lea_da, mar_wi)
    Call SFlg(.DIB(2), mdr_ri, reg_sd, reg_w)
    Call SFlg(.DIB(3), mar_ri, lea_ad, reg_sb, reg_w)
    Exit Sub
  ElseIf b = &HF7 Then
    'xchg b,e
    ReDim .DIB(-1 To 3)
    Call SFlg(.DIB(0), reg_sb, reg_r, mdr_wi)
    Call SFlg(.DIB(1), reg_se, reg_r, lea_da, mar_wi)
    Call SFlg(.DIB(2), mdr_ri, reg_se, reg_w)
    Call SFlg(.DIB(3), mar_ri, lea_ad, reg_sb, reg_w)
    Exit Sub
  End If

  '--- NOP ---'
  If b = &H8F Then
    ReDim .DIB(-1 To -1)
    Exit Sub
  End If

  '--- FLAGS ---'
  If b = &H54 Then
    ReDim .DIB(-1 To 0)
    Call SFlg(.DIB(0), flg_stz)
    Exit Sub
  ElseIf b = &H55 Then
    ReDim .DIB(-1 To 0)
    Call SFlg(.DIB(0), flg_clz)
    Exit Sub
  ElseIf b = &H56 Then
    ReDim .DIB(-1 To 0)
    Call SFlg(.DIB(0), flg_stc)
    Exit Sub
  ElseIf b = &H57 Then
    ReDim .DIB(-1 To 0)
    Call SFlg(.DIB(0), flg_clc)
    Exit Sub
  ElseIf b = &H64 Then
    ReDim .DIB(-1 To 0)
    Call SFlg(.DIB(0), flg_sto)
    Exit Sub
  ElseIf b = &H65 Then
    ReDim .DIB(-1 To 0)
    Call SFlg(.DIB(0), flg_clo)
    Exit Sub
  ElseIf b = &H66 Then
    ReDim .DIB(-1 To 0)
    Call SFlg(.DIB(0), flg_sts)
    Exit Sub
  ElseIf b = &H67 Then
    ReDim .DIB(-1 To 0)
    Call SFlg(.DIB(0), flg_cls)
    Exit Sub
  ElseIf b = &H76 Then
    ReDim .DIB(-1 To 0)
    Call SFlg(.DIB(0), flg_sti)
    Exit Sub
  ElseIf b = &H77 Then
    ReDim .DIB(-1 To 0)
    Call SFlg(.DIB(0), flg_cli)
    Exit Sub
  End If

  '--- IN ---'
  If b = &HE4 Then
    b = Asc(Mid(.CIB, 2, 1))
    ReDim .DIB(-1 To 2)
    Call SFlg(.DIB(0), IIf((b And 16) > 0, -1, reg_sX(b And 3)),
IIf((b And 16) > 0, acc_r, reg_r), adr_im, adr_c, mar_wi)
    Call SFlg(.DIB(1), mar_re, ctl_pr, mdr_we)
    Call SFlg(.DIB(2), mdr_ri, IIf((b And 32) > 0, -1, reg_sX((b And
12) \ 43)), IIf((b And 32) > 0, acc_w, reg_w))
    Exit Sub
  ElseIf (b >= &HD8) And (b <= &HDB) Then
    ReDim .DIB(-1 To 2)
    Call SFlg(.DIB(0), op_idb_im, Asc(Mid(.CIB, 2, 1)), adr_im,
adr_c, mar_wi)
    Call SFlg(.DIB(1), mar_re, ctl_pr, mdr_we)
    Call SFlg(.DIB(2), mdr_ri, reg_sX(b And 3), reg_w)
    Exit Sub
  ElseIf b = &HE5 Then
    ReDim .DIB(-1 To 2)
    Call SFlg(.DIB(0), op_idb_im, Asc(Mid(.CIB, 2, 1)), adr_im,
adr_c, mar_wi)
    Call SFlg(.DIB(1), mar_re, ctl_pr, mdr_we)
    Call SFlg(.DIB(2), mdr_ri, acc_w)
    Exit Sub
  End If

  '--- OUT ---'
  If b = &HD4 Then
    b = Asc(Mid(.CIB, 2, 1))
    ReDim .DIB(-1 To 2)
    Call SFlg(.DIB(0), IIf((b And 32) > 0, -1, reg_sX((b And 12) \
4)), IIf((b And 32) > 0, acc_r, reg_r), adr_im, adr_c, mar_wi)
    Call SFlg(.DIB(1), IIf((b And 16) > 0, -1, reg_sX(b And 3)),
IIf((b And 16) > 0, acc_r, reg_r), mdr_wi)
    Call SFlg(.DIB(2), mdr_re, mar_re, ctl_pw)
    Exit Sub
  ElseIf (b >= &HE0) And (b <= &HE3) Then
    ReDim .DIB(-1 To 2)
    Call SFlg(.DIB(0), reg_sX(b And 3), reg_r, adr_im, adr_c,
mar_wi)
    Call SFlg(.DIB(1), op_idb_im, Chr2Dec(Mid(.CIB, 2, 2)),
mdr_wi)
    Call SFlg(.DIB(2), mdr_re, mar_re, ctl_pw)
    Exit Sub
  ElseIf (b >= &HD0) And (b <= &HD3) Then
    ReDim .DIB(-1 To 2)
    Call SFlg(.DIB(0), op_idb_im, Asc(Mid(.CIB, 2, 1)), adr_im,
adr_c, mar_wi)
    Call SFlg(.DIB(1), reg_sX(b And 3), reg_r, mdr_wi)
    Call SFlg(.DIB(2), mdr_re, mar_re, ctl_pw)
    Exit Sub
  ElseIf b = &HD5 Then
    ReDim .DIB(-1 To 2)
    Call SFlg(.DIB(0), acc_r, adr_im, adr_c, mar_wi)
    Call SFlg(.DIB(1), op_idb_im, Chr2Dec(Mid(.CIB, 2, 2)),
mdr_wi)
    Call SFlg(.DIB(2), mdr_re, mar_re, ctl_pw)
    Exit Sub
  ElseIf b = &HD6 Then
    ReDim .DIB(-1 To 2)
    Call SFlg(.DIB(0), op_idb_im, Asc(Mid(.CIB, 2, 1)), adr_im,
adr_c, mar_wi)
    Call SFlg(.DIB(1), acc_r, mdr_wi)
    Call SFlg(.DIB(2), mdr_re, mar_re, ctl_pw)
    Exit Sub
  ElseIf b = &HD7 Then
    ReDim .DIB(-1 To 2)
    Call SFlg(.DIB(0), op_idb_im, Asc(Mid(.CIB, 2, 1)), adr_im,
adr_c, mar_wi)
    Call SFlg(.DIB(1), op_idb_im, Chr2Dec(Mid(.CIB, 3, 2)),
mdr_wi)
    Call SFlg(.DIB(2), mdr_re, mar_re, ctl_pw)
    Exit Sub
  End If

  '--- IRET ---'
  If b = &H73 Then
    ReDim .DIB(-1 To 5)
    'pop fl
    Call SFlg(.DIB(0), reg_spd, reg_ssp, adr_sr, adr_c, mar_wi)
    Call SFlg(.DIB(1), mar_re, ctl_mr, mdr_we)
    Call SFlg(.DIB(2), mdr_ri, flg_w)
    'pop pc
    Call SFlg(.DIB(3), reg_spd, reg_ssp, adr_sr, adr_c, mar_wi)
    Call SFlg(.DIB(4), mar_re, ctl_mr, mdr_we)
    Call SFlg(.DIB(5), reg_sip, mdr_ri, reg_w)
    Exit Sub
  End If

  'Could not decode
  Call Errr("Failed to decode instruction starting with " +
Dec2Hex(CInt(b), 2) + "h.")

  End With
End Sub

'-----------------------------------------------------'
' Private Sub eExecute()                              '
'                                                     '
' Advances simulation by one tick when CPU is in      '
' execute mode. Does NOT check if it is. Executes     '
' current microinstruction, switches CPU to Fetch     '
' mode after executing last microinstruction.         '
'-----------------------------------------------------'
Private Sub eExecute()
  With Proj.CPU
  Dim adr_rslt As Long

  'Check for no action
  If UBound(.DIB) = -1 Then GoTo ItsOver
```

```
  'Halt
  If GFlg(.DIB(.eDP), ctl_halt) Then
    Proj.Paused = True
    Proj.Running = True
    Proj.Halted = True
    fiMain.UpdateAll
    Call MsgBox("CPU halted", vbOKOnly + vbInformation)
    GoTo ItsOver
  End If
  'Condition
  If GFlg(.DIB(.eDP), op_jmp_cond) Then
    'jg - zf=0 and sf=o
    If .DIB(.eDP).nJmpCond = 0 Then If ((.FLAGS And flZ) = 0) And
((.FLAGS And flS) = 0) Then GoTo weContinue
    'jle - zf=1 or sf<>of
    If .DIB(.eDP).nJmpCond = 1 Then If ((.FLAGS And flS) = flS) Or
(((.FLAGS And flS) = flS) <> ((.FLAGS And flO) = flO)) Then GoTo
weContinue
    'jl - sf<>of
    If .DIB(.eDP).nJmpCond = 2 Then If ((.FLAGS And flS) = flS) <>
((.FLAGS And flO) = flO) Then GoTo weContinue
    'jge - sf=of
    If .DIB(.eDP).nJmpCond = 3 Then If ((.FLAGS And flS) = flS) =
((.FLAGS And flO) = flO) Then GoTo weContinue
    'jz - zf=1
    If .DIB(.eDP).nJmpCond = 4 Then If ((.FLAGS And flZ) = flZ) Then
GoTo weContinue
    'jnz - zf=0
    If .DIB(.eDP).nJmpCond = 5 Then If ((.FLAGS And flZ) = 0) Then
GoTo weContinue
    'jc - cf=1
    If .DIB(.eDP).nJmpCond = 6 Then If ((.FLAGS And flC) = flC) Then
GoTo weContinue
    'jnc - cf=0
    If .DIB(.eDP).nJmpCond = 7 Then If ((.FLAGS And flC) = 0) Then
GoTo weContinue
    'jo - of=1
    If .DIB(.eDP).nJmpCond = 8 Then If ((.FLAGS And flO) = flO) Then
GoTo weContinue
    'jno - of=0
    If .DIB(.eDP).nJmpCond = 9 Then If ((.FLAGS And flO) = 0) Then
GoTo weContinue
    'js - sf=1
    If .DIB(.eDP).nJmpCond = 10 Then If ((.FLAGS And flS) = flS) Then
GoTo weContinue
    'jns - sf=0
    If .DIB(.eDP).nJmpCond = 11 Then If ((.FLAGS And flS) = 0) Then
GoTo weContinue
    GoTo ItsOver
weContinue:
  End If

  'Decrememnt SP
  If GFlg(.DIB(.eDP), reg_spd) Then .SP = .SP - 2

  'Flags
  If GFlg(.DIB(.eDP), flg_stz) Then .FLAGS = .FLAGS Or flZ
  If GFlg(.DIB(.eDP), flg_stc) Then .FLAGS = .FLAGS Or flC
  If GFlg(.DIB(.eDP), flg_sto) Then .FLAGS = .FLAGS Or flO
  If GFlg(.DIB(.eDP), flg_sts) Then .FLAGS = .FLAGS Or flS
  If GFlg(.DIB(.eDP), flg_sti) Then .FLAGS = .FLAGS Or flI
  If GFlg(.DIB(.eDP), flg_clz) Then .FLAGS = .FLAGS And Not flZ
  If GFlg(.DIB(.eDP), flg_clc) Then .FLAGS = .FLAGS And Not flC
  If GFlg(.DIB(.eDP), flg_clo) Then .FLAGS = .FLAGS And Not flO
  If GFlg(.DIB(.eDP), flg_cls) Then .FLAGS = .FLAGS And Not flS
  If GFlg(.DIB(.eDP), flg_cli) Then .FLAGS = .FLAGS And Not flI

  'Select register
  If GFlg(.DIB(.eDP), reg_sb) Then .eSelectedReg = 0
  If GFlg(.DIB(.eDP), reg_sc) Then .eSelectedReg = 1
  If GFlg(.DIB(.eDP), reg_sd) Then .eSelectedReg = 2
  If GFlg(.DIB(.eDP), reg_se) Then .eSelectedReg = 3
  If GFlg(.DIB(.eDP), reg_ssp) Then .eSelectedReg = 4
  If GFlg(.DIB(.eDP), reg_sip) Then .eSelectedReg = 5

  'Read operations for IDB
  If GFlg(.DIB(.eDP), reg_r) Then
    If .eSelectedReg = 4 Then
      .eIDB = .SP
    ElseIf .eSelectedReg = 5 Then
      .eIDB = .IP
    ElseIf .eSelectedReg >= 0 And .eSelectedReg <= 3 Then
      .eIDB = .R(.eSelectedReg)
    End If
  End If
  If GFlg(.DIB(.eDP), acc_r) Then .eIDB = .A
  If GFlg(.DIB(.eDP), flg_r) Then .eIDB = .FLAGS
  If GFlg(.DIB(.eDP), mdr_ri) Then .eIDB = .MDR
```

```
  'Immediate
  If GFlg(.DIB(.eDP), op_idb_im) Then .eIDB = .DIB(.eDP).nToIDB
  'Addressing
  If GFlg(.DIB(.eDP), adr_c) Then
    adr_rslt = 0
    If GFlg(.DIB(.eDP), adr_sr) Then
      If .eSelectedReg = 4 Then
        adr_rslt = adr_rslt + .SP * .DIB(.eDP).nAdrMul
      ElseIf .eSelectedReg = 5 Then
        adr_rslt = adr_rslt + .IP * .DIB(.eDP).nAdrMul
      Else
        adr_rslt = adr_rslt + .R(.eSelectedReg) *
.DIB(.eDP).nAdrMul
      End If
    End If
    If GFlg(.DIB(.eDP), adr_br) Then adr_rslt = adr_rslt + .R(0)
    If GFlg(.DIB(.eDP), adr_im) Then adr_rslt = adr_rslt + .eIDB
    .eIAB = adr_rslt
  End If
  'MAR
  If GFlg(.DIB(.eDP), mar_ri) Then .eIAB = .MAR
  'External buses read
  If GFlg(.DIB(.eDP), mar_re) Then .eEAB = .MAR
  If GFlg(.DIB(.eDP), mdr_re) Then .eEDB = .MDR

  'LEA
  If GFlg(.DIB(.eDP), lea_ad) Then .eIDB = .eIAB
  If GFlg(.DIB(.eDP), lea_da) Then .eIAB = .eIDB
  'Memory operations
  If GFlg(.DIB(.eDP), ctl_mr) Then .eEDB = Proj.RAM(.eEAB) *
CLng(256) + Proj.RAM(.eEAB + 1)
  If GFlg(.DIB(.eDP), ctl_mw) Then Proj.RAM(.eEAB) = Int(.eEDB /
256) And &HFF: Proj.RAM(.eEAB + 1) = .eEDB And &HFF
  'Port IO operations
  If GFlg(.DIB(.eDP), ctl_pr) Then .eEDB = devPortRead(.eEAB And
65535)
  If GFlg(.DIB(.eDP), ctl_pw) Then Call devPortWrite(.eEAB And
65535, .eEDB)

  'ALU
  Dim l1 As Long, l2 As Long, l3 As Long
  Dim i As Integer, lf As Long
  Dim b1 As Boolean, b2 As Boolean, b3 As Boolean
  Dim scf As Integer 'set carry flag: -1 leave, 0 set false, 1 set
true
  scf = -1
  If GFlg(.DIB(.eDP), op_alu_c) Then
    i = .DIB(.eDP).nAluOpNum
    If GFlg(.DIB(.eDP), alu_swp) Then
      l1 = .eIDB          'operand 1
      l2 = .A             'operand 2
    Else
      l1 = .A             'operand 1
      l2 = .eIDB          'operand 2
    End If
    l3 = l1                'result to be saved
    lf = l1                'result to be used for flags
    b1 = (l1 And 32768) > 0  'l1 is negative for signed ops
    b2 = (l2 And 32768) > 0  'l2 is negative for signed ops
    If (i >= 27) And (i <= 34) Then
      i = i - 8
      l1 = .DIB(.eDP).nAluSh
    End If
    If i = 0 Then 'add
      l3 = l1 + l2
      lf = l3
    ElseIf i = 1 Then 'sub
      l3 = l1 - l2
      lf = l3
    ElseIf i = 2 Then 'adc
      l3 = l1 + l2 + IIf((.FLAGS And flC) > 0, 1, 0)
      lf = l3
    ElseIf i = 3 Then 'sbb
      l3 = l1 - l2 - IIf((.FLAGS And flC) > 0, 1, 0)
      lf = l3
    ElseIf i = 4 Then 'cmp
      lf = l1 - l2
    ElseIf i = 5 Then 'mul
      l3 = l1 * l2
      lf = l3
    ElseIf i = 6 Then 'div
      l3 = l1 \ l2
      lf = l3
    ElseIf i = 7 Then 'imul
      l3 = (Abs(l1) And &H7FFF) * (Abs(l2) And &H7FFF)
      If (b1 Xor b2) Then l3 = -l3
      lf = l3
```

```
ElseIf i = 8 Then 'idiv
  l3 = Int((Abs(l1) And &H7FFF) / (Abs(l2) And &H7FFF))
  If (b1 Xor b2) Then l3 = -l3
  lf = l3
ElseIf i = 9 Then 'mod
  l3 = l1 Mod l2
  lf = l3
ElseIf i = 10 Then 'and
  l3 = l1 And l2
  lf = l3
ElseIf i = 11 Then 'or
  l3 = l1 Or l2
  lf = l3
ElseIf i = 12 Then 'xor
  l3 = l1 Xor l2
  lf = l3
ElseIf i = 13 Then 'test
  lf = l1 And l2
ElseIf i = 14 Then 'inc
  l3 = l2 + 1
  lf = l3
ElseIf i = 15 Then 'dec
  l3 = l2 - 1
  lf = l3
ElseIf i = 16 Then 'neg
  l3 = (65535 - l2) + 1
  lf = l3
ElseIf i = 17 Then 'not
  l3 = 65535 - l2
  lf = l3
ElseIf i = 18 Then 'bswp
  l3 = (l2 And 65280) \ 256 + (l2 And 255) * 256
  lf = l3
ElseIf i = 19 Then 'lshl A,Rn
  i = l1 And 15
  scf = IIf((l2 And (2 ^ (16 - i))) = 0, 0, 1)
  l3 = l2 * (2 ^ i)
  lf = l3
ElseIf i = 20 Then 'lshr A,Rn
  i = l1 And 15
  If i > 0 Then
    scf = IIf((l2 And (2 ^ (i - 1))) = 0, 0, 1)
    l3 = l2 \ (2 ^ i)
  Else
    l3 = l2
  End If
  lf = l3
ElseIf i = 21 Then 'ashl A,Rn
  i = l1 And 15
  scf = IIf((l2 And (2 ^ (16 - i))) = 0, 0, 1)
  l3 = l2 * (2 ^ i)
  lf = l3
ElseIf i = 22 Then 'ashr A,Rn
  l3 = l2
  For i = 1 To l1 And 15
    scf = IIf((l2 And 1) = 0, 0, 1)
    l3 = l2 \ 2
    If (l2 And 32768) = 1 Then l3 = l3 Or 32768
  Next
  lf = l3
ElseIf i = 23 Then 'rol A,Rn
  l3 = l2
  For i = 1 To (l1 And 15)
    l3 = ((l3 * 2) And 65535) + IIf((l3 And 32768) = 0, 0, 1)
  Next
  scf = IIf((l3 And 1) = 0, 0, 1)
  lf = l3
ElseIf i = 24 Then 'ror A,Rn
  l3 = l2
  For i = 1 To (l1 And 15)
    l3 = (l3 \ 2) + IIf((l3 And 1) = 0, 0, 32768)
  Next
  scf = IIf((l3 And 32768) = 0, 0, 1)
  lf = l3
ElseIf i = 25 Then 'rcl A,Rn
  l3 = l2
  scf = IIf((.FLAGS And flC) > 0, 1, 0)
  For i = 1 To (l1 And 15)
    lf = l3
    l3 = ((l3 * 2) And 65535) + IIf(scf = 1, 1, 0)
    scf = IIf((lf And 32768) = 0, 0, 1)
  Next
  lf = l3
ElseIf i = 26 Then 'rcr A,Rn
  l3 = l2
  scf = IIf((.FLAGS And flC) > 0, 1, 0)
  For i = 1 To (l1 And 15)
    lf = l3
        l3 = (l3 \ 2) + IIf(scf = 1, 32768, 0)
        scf = IIf((lf And 1) = 0, 0, 1)
      Next
      lf = l3
    End If
    'Result to IDB
    .eIDB = (l3 And 65535)
    'I hate VB. I hate VB. I hate VB
    '(l3 and &HFFFF) returned weird results. (65535=&HFFFF)
returned
    'false. (65535=Abs(CLng(&HFFFF))) returned true, but
    'Abs(CLng(&HFFFF)) returned 1. Abs(&HFFFF) returns overflow,
and
    'CLng(&HFFFF) produces same results as &HFFFF. I hate it when
    'VB does this kind of rubbish. Why is 65535 any different from
    '&HFFFF??? Apparently anything bigger than 32767 becomes
negative
    'when written in hexadecimal form. I use hex form quite often,
I
    'wonder how many more errors like that there are in my code.

    'Flags
    b3 = (l3 And 32768) > 0
    .FLAGS = .FLAGS And (Not (flZ + flS + flO + flC + flP + flN))
    .FLAGS = .FLAGS Or IIf(lf = 0, flZ, 0)
    .FLAGS = .FLAGS Or IIf((lf And 32768) > 0, flS, 0)
    If b1 = b2 Then .FLAGS = .FLAGS Or IIf(b1 Xor b3, flO, 0)
    .FLAGS = .FLAGS Or IIf((lf And &HFFFF0000) > 0, flC, 0)
    b1 = (.FLAGS And flS)
    b2 = (.FLAGS And flO)
    .FLAGS = .FLAGS Or IIf(b1 <> b2, flN, 0)
    .FLAGS = .FLAGS Or IIf((b1 = b2) And ((.FLAGS And flZ) = 0),
flP, 0)
    If scf = 1 Then .FLAGS = .FLAGS Or flC
    If scf = 0 Then .FLAGS = .FLAGS And Not flC
  End If

  'External buses write
  If GFlg(.DIB(.eDP), mar_we) Then .MAR = .eEAB
  If GFlg(.DIB(.eDP), mdr_we) Then .MDR = .eEDB
  'Write operations for IDB
  If GFlg(.DIB(.eDP), reg_w) Then
    If .eSelectedReg = 4 Then
      .SP = .eIDB
    ElseIf .eSelectedReg = 5 Then
      .IP = .eIDB
    ElseIf .eSelectedReg >= 0 And .eSelectedReg <= 3 Then
      .R(.eSelectedReg) = .eIDB
    End If
  End If
  If GFlg(.DIB(.eDP), acc_w) Then .A = .eIDB
  If GFlg(.DIB(.eDP), flg_w) Then .FLAGS = .eIDB
  If GFlg(.DIB(.eDP), mdr_wi) Then .MDR = .eIDB

  'MAR
  If GFlg(.DIB(.eDP), mar_wi) Then .MAR = .eIAB

  'Increment IP/SP
  If GFlg(.DIB(.eDP), reg_ipi) Then .IP = .IP + 1
  If GFlg(.DIB(.eDP), reg_spi) Then .SP = .SP + 2

  'Return internal registers to zeroes in order to prevent
  'possibility of data transferred between microinstructions by
  'just being on the buses - impossible in real life.

  '.eEAB = 0  will not do this for now - wait until I will
  '.eEDB = 0  fix a bug with decoding ALU instructions - they
  '.eIAB = 0  relied on values staying in these registers
  '.eIDB = 0

  'DONE! - take next microinstruction
  .eDP = .eDP + 1

  'Last loop of execution cycle
  If .eDP = UBound(.DIB) + 1 Then
ItsOver:
    'Prepare registers
    .CIB = ""
    .Fetch = True
    .eDP = 0
    ReDim .DIB(-1 To -1)
    'Process interrupts
    eInterrupt
  End If

  End With
End Sub
```

```
'---------------------------------------------'
' Private Sub eInterrupt()                    '
'                                             '
' Checks if an interrupt is required, prepares '
' a microprogram to initiate one if necessary '
'---------------------------------------------'
Private Sub eInterrupt()
  With Proj.CPU

  'Interrupt number to be initiated
  Dim INTx As Long, i As Integer
  INTx = -1

  'Get interrupt number
  For i = 0 To 15
    If (.IS And 2 ^ i) > 0 Then
      INTx = i
      .IS = .IS And Not (2 ^ i)   'clear the Pending flag
      Exit For
    End If
  Next

  'Initiate interrupt
  If INTx <> -1 Then
    ReDim .DIB(-1 To 8)
    'push pc
    Call SFlg(.DIB(0), reg_ssp, adr_sr, adr_c, mar_wi)
    Call SFlg(.DIB(1), reg_sip, reg_r, mdr_wi)
    Call SFlg(.DIB(2), mdr_re, mar_re, ctl_mw, reg_spi)
    'push fl
    Call SFlg(.DIB(3), reg_ssp, adr_sr, adr_c, mar_wi)
    Call SFlg(.DIB(4), flg_r, mdr_wi)
    Call SFlg(.DIB(5), mdr_re, mar_re, ctl_mw, reg_spi, flg_cli)
    'get interrupt vector
    Call SFlg(.DIB(6), op_idb_im, 65280 + INTx * 2, adr_im, adr_c,
mar_wi)
    Call SFlg(.DIB(7), mar_re, ctl_mr, mdr_we)
    'jump to received address
    Call SFlg(.DIB(8), mdr_ri, reg_sip, reg_w)

    'Continue execute cycle
    .Fetch = False
    .CIB = ""
  End If

  End With
End Sub

'-----------------------------------------------------'
' Private Sub DecodeMem(MemOffset As Integer)         '
'                                                     '
' DESCRIPTION: after calling this procedure, .DIB     '
'   will have added to it microcommands that put      '
'   required address into MAR.                        '
'                                                     '
' PARAMETERS:                                         '
'   MemOffset - offset of addressing bytes in CIB     '
'-----------------------------------------------------'
Private Sub DecodeMem(MemOffset As Integer)
  With Proj.CPU

  Dim adr As String, z As Integer, b As Integer
  adr = Mid(.CIB, MemOffset) 'separate the addressing
  b = Asc(Left(adr, 1)) 'first byte of the addressing
  z = UBound(.DIB) + 1 'zero offset for DIB - to simplify adding
lines
  If (b And 192) = 0 Then
    If (b And 1) = 0 Then 'direct via immediate
      ReDim Preserve .DIB(-1 To z + 0)
      Call SFlg(.DIB(z + 0), adr_im, adr_c, mar_wi, op_idb_im,
Chr2Dec(Mid(adr, 2, 2)))
    Else 'indirect via immediate
      ReDim Preserve .DIB(-1 To z + 2)
      Call SFlg(.DIB(z + 0), adr_im, adr_c, mar_wi, op_idb_im,
Chr2Dec(Mid(adr, 2, 2)))
      Call SFlg(.DIB(z + 1), mar_re, ctl_mr, mdr_we)
      Call SFlg(.DIB(z + 2), adr_im, adr_c, mdr_ri, mar_wi)
    End If
  ElseIf (b And 192) = 64 Then 'indirect via register
    ReDim Preserve .DIB(-1 To z + 0)
    Call SFlg(.DIB(z + 0), reg_sX(b And 3), adr_sr, adr_c, mar_wi)
  ElseIf (b And 128) = 128 Then 'indexed
    ReDim Preserve .DIB(-1 To z + 0)
    Call SFlg(.DIB(z + 0), IIf((b And 64) = 64, adr_br, -1), reg_sX(b
And 3), _
      IIf((b And 32) = 32, op_idb_im, -1), IIf((b And 32) = 32,
Chr2Dec(Mid(adr, 2, 2)), -1), _
```

```
      op_adr_mm, (b And 12) \ 4, adr_sr, IIf((b And 32) = 32,
adr_im, -1), adr_c, mar_wi)
  End If

  End With
End Sub

'-------------------------------------------------------------'
' Private Function reg_sX(ByVal Index As Integer) As integer '
'                                                             '
' Returns "Select register X" signal for R(Index)            '
'-------------------------------------------------------------'
Private Function reg_sX(ByVal Index As Integer) As Integer
  reg_sX = 0
  If Index = 0 Then reg_sX = reg_sb: Exit Function
  If Index = 1 Then reg_sX = reg_sc: Exit Function
  If Index = 2 Then reg_sX = reg_sd: Exit Function
  If Index = 3 Then reg_sX = reg_se: Exit Function
End Function

'-----------------------------------------------------'
' Public Function DI2Str(ByRef db As TpDI) As String '
'                                                     '
' Converts microinstruction db into text with all    '
' the signals in a logical order.                     '
'-----------------------------------------------------'
Public Function DI2Str(ByRef db As TpDI) As String
  Dim s As String, i As Integer
  s = ""

  'Condition
  If GFlg(db, op_jmp_cond) Then s = s + "jmp_cond(" +
CStr(db.nJmpCond) + "), "

  'Decrement SP
  If GFlg(db, reg_spd) Then s = s + "reg_spd, "

  'Flags
  If GFlg(db, flg_stz) Then s = s + "flg_stz, "
  If GFlg(db, flg_stc) Then s = s + "flg_stc, "
  If GFlg(db, flg_sto) Then s = s + "flg_sto, "
  If GFlg(db, flg_sts) Then s = s + "flg_sts, "
  If GFlg(db, flg_sti) Then s = s + "flg_sti, "
  If GFlg(db, flg_clz) Then s = s + "flg_clz, "
  If GFlg(db, flg_clc) Then s = s + "flg_clc, "
  If GFlg(db, flg_clo) Then s = s + "flg_clo, "
  If GFlg(db, flg_cls) Then s = s + "flg_cls, "
  If GFlg(db, flg_cli) Then s = s + "flg_cli, "

  'Select register
  If GFlg(db, reg_sb) Then s = s + "reg_sb, "
  If GFlg(db, reg_sc) Then s = s + "reg_sc, "
  If GFlg(db, reg_sd) Then s = s + "reg_sd, "
  If GFlg(db, reg_se) Then s = s + "reg_se, "
  If GFlg(db, reg_ssp) Then s = s + "reg_ssp, "
  If GFlg(db, reg_sip) Then s = s + "reg_sip, "

  'Read operations for IDB
  If GFlg(db, reg_r) Then s = s + "reg_r, "
  If GFlg(db, acc_r) Then s = s + "acc_r, "
  If GFlg(db, flg_r) Then s = s + "flg_r, "
  If GFlg(db, mdr_ri) Then s = s + "mdr_ri, "

  'Immediate
  If GFlg(db, op_idb_im) Then s = s + "idb_im(0" +
Dec2Hex(db.nToIDB, 4) + "h), "
  'Addressing
  If GFlg(db, op_adr_mm) Then s = s + "adr_mm_" + CStr(db.nAdrMul)
+ ", "
  If GFlg(db, adr_br) Then s = s + "adr_br, "
  If GFlg(db, adr_im) Then s = s + "adr_im, "
  If GFlg(db, adr_c) Then s = s + "adr_c, "
  'MAR
  If GFlg(db, mar_ri) Then s = s + "mar_ri, "
  'External buses read
  If GFlg(db, mar_re) Then s = s + "mar_re, "
  If GFlg(db, mdr_re) Then s = s + "mdr_re, "

  'LEA
  If GFlg(db, lea_ad) Then s = s + "lea_ad, "
  'Memory operations
  If GFlg(db, ctl_mr) Then s = s + "ctl_mr, "
  If GFlg(db, ctl_mw) Then s = s + "ctl_mw, "
  'Port IO operations
  If GFlg(db, ctl_pr) Then s = s + "ctl_pr, "
  If GFlg(db, ctl_pw) Then s = s + "ctl_pw, "

  'ALU
```

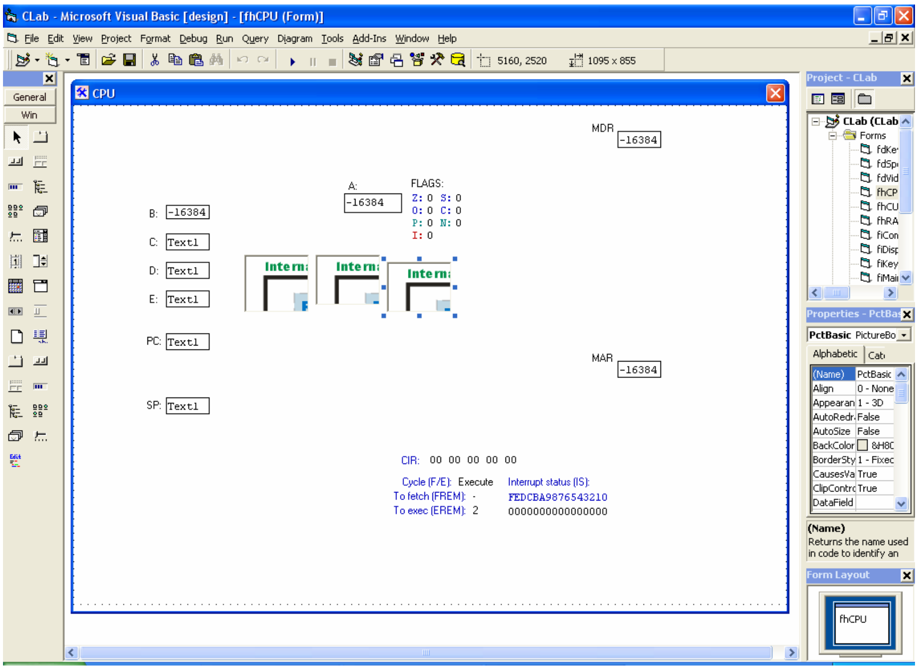
```
  If GFlg(db, alu_swp) Then s = s + "alu_swp, "
  If GFlg(db, op_alu_sh) Then s = s + "alu_sh(" + CStr(db.nAluSh) +
"), "
  If GFlg(db, op_alu_c) Then s = s + "alu_c(" + CStr(db.nAluOpNum) +
"), "

  'External buses write
  If GFlg(db, mar_we) Then s = s + "mar_we, "
  If GFlg(db, mdr_we) Then s = s + "mdr_we, "
  'Write operations for IDB
  If GFlg(db, reg_w) Then s = s + "reg_w, "
  If GFlg(db, acc_w) Then s = s + "acc_w, "
  If GFlg(db, flg_w) Then s = s + "flg_w, "
  If GFlg(db, mdr_wi) Then s = s + "mdr_wi, "

  'MAR
  If GFlg(db, mar_wi) Then s = s + "mar_wi, "

  'Increment IP/SP
  If GFlg(db, reg_ipi) Then s = s + "reg_ipi, "
  If GFlg(db, reg_spi) Then s = s + "reg_spi, "

  'Halt
  If GFlg(db, ctl_halt) Then s = s + "ctl_halt, "

  'Return result
  If Len(s) > 2 Then DI2Str = Left(s, Len(s) - 2) Else DI2Str = s
End Function
```

22.6. pIO

```
Option Explicit

'---------------------------------------------------------------'
' Each device module will start with fd, and will only be       '
' interfaced through this unit (apart from window-related       '
' functions such as .Hide). Device modules should 'export'      '
' the following functions:                                      '
'                                                               '
' Init - whatever intialisation devices want. This is not       '
'   simulation-related - rather, interface-related, as          '
'   this will be called only once at program startup.           '
' Reset - simulation-related initialisations. Should be         '
'   similar to what a real device would do when it              '
'   receives Reset signal. Reset will be sent when user         '
'   presses the computer Reset button, or during program        '
'   development when program is restarted.                      '
' Tick - function to be called every CPU tick. Whatever         '
'   a device would do with every rise of OSC in a real          '
'   CPU is what should be done in Tick.                         '
' PortRead - will simulate arrival of PortRead signal.          '
'   The function should return whatever the device would        '
'   place on the Data Bus in a real computer, or 999999         '
'   if the device decides to ignore the signal.                 '
' PortWrite - simulates arrival of PortWrite signal.            '
'---------------------------------------------------------------'

'--------------------------'
' Public Sub ioInit()      '
'                          '
' Initializes this module  '
'--------------------------'
Public Sub ioInit()
  '
End Sub

'-------------------------'
' Public Sub devInit()    '
'                         '
' Initialises all devices '
'-------------------------'
Public Sub devInit()
  fdVideo.Init
  fdKeyboard.Init
  fdSpeaker.Init
End Sub

'----------------------'
' Public Sub devReset() '
'                      '
' Resets all devices   '
'----------------------'
Public Sub devReset()
  fdVideo.Reset
  fdKeyboard.Reset
  fdSpeaker.Reset
End Sub

'---------------------'
' Public Sub devTick() '
'                     '
' Resets all devices  '
'---------------------'
Public Sub devTick()
  fdVideo.Tick
  fdKeyboard.Tick
  fdSpeaker.Tick
End Sub

'------------------------------------------'
' Public Function devPortRead(PortNum As '
'   Integer) As Long                     '
'                                        '
' Reads a word from the specified port.  '
' If no device accepts the read signal,  '
' 0 is returned.                         '
'------------------------------------------'
Public Function devPortRead(PortNum As Integer) As Long
  Dim i As Long
  i = fdVideo.PortRead(PortNum)
  If i <> 999999 Then devPortRead = i: Exit Function
  i = fdKeyboard.PortRead(PortNum)
  If i <> 999999 Then devPortRead = i: Exit Function
  i = fdSpeaker.PortRead(PortNum)
  If i <> 999999 Then devPortRead = i: Exit Function
  'No one accepted - return 0
  devPortRead = 0
End Function

'------------------------------------------------'
' Public Sub devPortWrite(PortNum As Integer,    '
'   Dt As Long)                                  '
'                                                '
' Writes a word to specified port. If no devices '
' accept the port, nothing happens. If several   '
' devices would accept the port, only the first  '
' one gets the signal.                           '
'------------------------------------------------'
Public Sub devPortWrite(PortNum As Integer, Dt As Long)
  Call fdVideo.PortWrite(PortNum, Dt)
  Call fdKeyboard.PortWrite(PortNum, Dt)
  Call fdSpeaker.PortWrite(PortNum, Dt)
End Sub

'---------------------------------------------------'
' Public Function IRQ(IRQnum As Integer) As Boolean '
'                                                   '
' The function that devices would call to request   '
' an interrupt. Interrupt is acknowledged if this   '
' function returns True.                            '
'---------------------------------------------------'
Public Function IRQ(IRQnum As Integer) As Boolean
  With Proj.CPU

  'Validity - only IRQ numbers 0-15 are valid
  IRQ = False
  If Not (IRQnum >= 0 And IRQnum <= 15) Then Exit Function
  'Are interrupts allowed?
  If (.FLAGS And flI) = 0 Then Exit Function
  'Is this interrupt still pending?
  If (.IS And 2 ^ IRQnum) > 0 Then Exit Function
  'Set Pending bit
  .IS = .IS Or 2 ^ IRQnum
  'INTA signal
  IRQ = True

  fsRegs.Update
  fhCPU.Update
  fhCU.Update

  End With
```

```
End Function
```

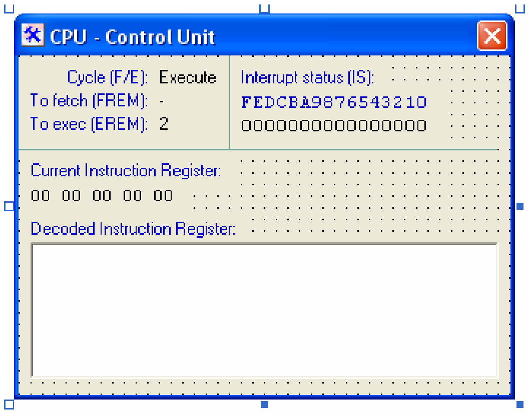
22.7. fhCPU




```
Option Explicit

'-----------------------------------------'
' Public declarations in this module:     '
'                                          '
' PROCEDURES:                              '
'   Reset                                  '
'-----------------------------------------'

'----------------------------'
' Public Sub Reset()         '
'                            '
' Initialises the module and '
' related variables.         '
'----------------------------'
Public Sub Reset()
  'Empty the DIB array
  ReDim Proj.CPU.DIB(-1 To -1)

  'Init registers
  Proj.CPU.A = 0
  Proj.CPU.R(0) = 0
  Proj.CPU.R(1) = 0
  Proj.CPU.R(2) = 0
  Proj.CPU.R(3) = 0
  Proj.CPU.IP = 0
  Proj.CPU.SP = 24576 'meaning 6000h
  Proj.CPU.FLAGS = fII
  Proj.CPU.MAR = 0
  Proj.CPU.MDR = 0
  Proj.CPU.CIB = ""
  Proj.CPU.Fetch = True
  Proj.CPU.FREM = 0
  Proj.CPU.fremMem = False
  Proj.CPU.IS = 0

  Proj.CPU.eDP = 0
  Proj.CPU.eEAB = 0
  Proj.CPU.eEDB = 0
  Proj.CPU.eIAB = 0
  Proj.CPU.eIDB = 0
  Proj.CPU.eSelectedReg = 0

  Proj.P.Code = ""
  Proj.P.CompileNeeded = True

  ReDim Proj.CPU.Breakpoint(-1 To -1)

  'Update once
  Update
End Sub

'----------------------------------------------'
' Private Sub Form_Unload(Cancel As Integer)   '
'                                              '
' DESCRIPTION: Event handler for Form_Unload   '
'   unloads the form if the application is     '
'   really shutting down, and just hides the   '
'   form in case the user requested to close   '
'   it.                                        '
'----------------------------------------------'
Private Sub Form_Unload(Cancel As Integer)
  If Not Appp.Terminating Then
    Cancel = 1
    fhCPU.Hide
  End If
End Sub

'---------------------'
' Public Sub Update() '
'---------------------'
Public Sub Update()
  'General registers
  LA.Text = Dec2Fmt16(Proj.CPU.A, Proj.NmbRep)
  LB.Text = Dec2Fmt16(Proj.CPU.R(0), Proj.NmbRep)
  LC.Text = Dec2Fmt16(Proj.CPU.R(1), Proj.NmbRep)
  LD.Text = Dec2Fmt16(Proj.CPU.R(2), Proj.NmbRep)
  LE.Text = Dec2Fmt16(Proj.CPU.R(3), Proj.NmbRep)
  LIP.Text = Dec2Fmt16(Proj.CPU.IP, Proj.NmbRep)
  LSP.Text = Dec2Fmt16(Proj.CPU.SP, Proj.NmbRep)
  LMAR.Text = Dec2Fmt16(Proj.CPU.MAR, Proj.NmbRep)
  LMDR.Text = Dec2Fmt16(Proj.CPU.MDR, Proj.NmbRep)
  'Flags
  LFZ.Caption = IIf((Proj.CPU.FLAGS And 1) > 0, "1", "0")
  LFS.Caption = IIf((Proj.CPU.FLAGS And 2) > 0, "1", "0")
  LFO.Caption = IIf((Proj.CPU.FLAGS And 4) > 0, "1", "0")
  LFC.Caption = IIf((Proj.CPU.FLAGS And 8) > 0, "1", "0")
  LFI.Caption = IIf((Proj.CPU.FLAGS And 16) > 0, "1", "0")
```

```
  LFN.Caption = IIf((Proj.CPU.FLAGS And 256) > 0, "1", "0")
  LFP.Caption = IIf((Proj.CPU.FLAGS And 512) > 0, "1", "0")
  'Control Unit
  LFE.Caption = IIf(Proj.CPU.Fetch, "Fetch", "Execute")
  If Proj.CPU.Fetch Then
    If Proj.CPU.CIB = "" Then
      LFREM.Caption = "?"
    Else
      LFREM.Caption = IIf(Proj.CPU.fremMem, CStr(Proj.CPU.FREM + 1) +
"+", CStr(Proj.CPU.FREM))
    End If
    LEREM.Caption = "N/A"
  Else
    LEREM.Caption = CStr(UBound(Proj.CPU.DIB) - Proj.CPU.eDP + 1)
    LFREM.Caption = "N/A"
  End If
  LCIR.Caption = Str2Chr(Proj.CPU.CIB)
  LIS.Caption = Dec2Bin(Proj.CPU.IS, 16)
End Sub

'---------------------------'
' Public Sub SetComplexity() '
'---------------------------'
Public Sub SetComplexity()
  If Proj.Complexity = 0 Then
    PctPanel.Picture = PctBasic.Picture
    LMDR.Visible = False
    LMAR.Visible = False
    LCIR.Visible = False
    LFZ.Visible = False
    LFS.Visible = False
    LFO.Visible = False
    LFC.Visible = False
    LFP.Visible = False
    LFN.Visible = False
    LFI.Visible = False
    llMDR.Visible = False
    llMAR.Visible = False
    llCIR.Visible = False
    llFLAGS.Visible = False
    llFZ.Visible = False
    llFS.Visible = False
    llFO.Visible = False
    llFC.Visible = False
    llFP.Visible = False
    llFN.Visible = False
    llFI.Visible = False
    LFE.Visible = False
    llFE.Visible = False
    llFREM.Visible = False
    LFREM.Visible = False
    LEREM.Visible = False
    llEREM.Visible = False
    LIS.Visible = False
    llIS.Visible = False
    lllIS.Visible = False
  ElseIf Proj.Complexity = 1 Then
    PctPanel.Picture = PctAlevel.Picture
    LMDR.Visible = True
    LMAR.Visible = True
    LCIR.Visible = True
    LFZ.Visible = True
    LFS.Visible = True
    LFO.Visible = True
    LFC.Visible = True
    LFP.Visible = True
    LFN.Visible = True
    LFI.Visible = True
    llMDR.Visible = True
    llMAR.Visible = True
    llCIR.Visible = True
    llFLAGS.Visible = True
    llFZ.Visible = True
    llFS.Visible = True
    llFO.Visible = True
    llFC.Visible = True
    llFP.Visible = True
    llFN.Visible = True
    llFI.Visible = True
    LFE.Visible = True
    llFE.Visible = True
    llFREM.Visible = True
    LFREM.Visible = True
    LEREM.Visible = True
    llEREM.Visible = True
    LIS.Visible = True
    llIS.Visible = True
    lllIS.Visible = True
  Else
    PctPanel.Picture = PctFull.Picture
    LMDR.Visible = True
    LMAR.Visible = True
    LCIR.Visible = True
    LFZ.Visible = True
    LFS.Visible = True
    LFO.Visible = True
    LFC.Visible = True
    LFP.Visible = True
    LFN.Visible = True
    LFI.Visible = True
    llMDR.Visible = True
    llMAR.Visible = True
    llCIR.Visible = True
    llFLAGS.Visible = True
    llFZ.Visible = True
    llFS.Visible = True
    llFO.Visible = True
    llFC.Visible = True
    llFP.Visible = True
    llFN.Visible = True
    llFI.Visible = True
    LFE.Visible = True
    llFE.Visible = True
    llFREM.Visible = True
    LFREM.Visible = True
    LEREM.Visible = True
    llEREM.Visible = True
    LIS.Visible = True
    llIS.Visible = True
    lllIS.Visible = True
  End If
End Sub
```

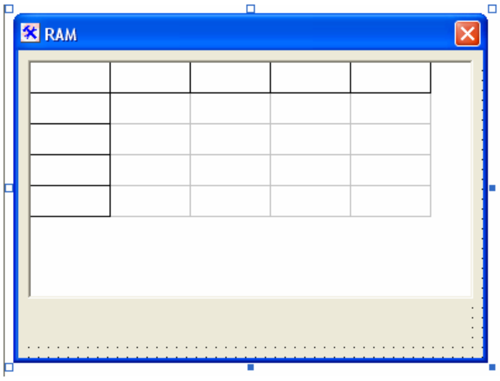
22.8. fhCU




```
Option Explicit

'-----------------------------------------'
' Public declarations in this module:     '
'                                         '
' PROCEDURES:                             '
'   Reset                                 '
'   Update                                '
'-----------------------------------------'

'Stores the string values for all registers displayed
' on this form to track changes and highlight respectively
Private LastStr(0 To 8) As String

'--------------------------'
' Public Sub Init()        '
'                          '
' Initializes this module  '
'--------------------------'
Public Sub Init()
  'Update register values
  Update
  'Save changes
  SaveLast
  'Update again to highlight nothing
  Update
End Sub

'-----------------------------------------------'
' Private Sub Form_Unload(Cancel As Integer)    '
'                                               '
' DESCRIPTION: Event handler for Form_Unload    '
'   unloads the form if the application is      '
'   really shutting down, and just hides the    '
'   form in case the user requested to close    '
'   it.                                         '
'-----------------------------------------------'
Private Sub Form_Unload(Cancel As Integer)
  If Not Appp.Terminating Then
    Cancel = 1
    fhCU.Hide
  End If
End Sub

'-----------------------------------------'
' Public Sub Update()                     '
'                                         '
' Updates the contents of the window to   '
' reflect changes to the state of the     '
' simulation.                             '
'-----------------------------------------'
Public Sub Update()
  'Register values
  LFE.Caption = IIf(Proj.CPU.Fetch, "Fetch", "Execute")
  If Proj.CPU.Fetch Then
    If Proj.CPU.CIB = "" Then
      LFREM.Caption = "?"
    Else
      LFREM.Caption = IIf(Proj.CPU.fremMem, CStr(Proj.CPU.FREM + 1) +
"+", CStr(Proj.CPU.FREM))
    End If
    LEREM.Caption = "N/A"
  Else
    LEREM.Caption = CStr(UBound(Proj.CPU.DIB) - Proj.CPU.eDP + 1)
    LFREM.Caption = "N/A"
  End If
  LCIB.Caption = Str2Chr(Proj.CPU.CIB)
  LIS.Caption = Dec2Bin(Proj.CPU.IS, 16)

  'Register colors
  If LastStr(0) <> LFE.Caption Then LFE.ForeColor = &HFF Else
LFE.ForeColor = 0
  If LastStr(1) <> LFREM.Caption Then LFREM.ForeColor = &HFF Else
LFREM.ForeColor = 0
  If LastStr(2) <> LEREM.Caption Then LEREM.ForeColor = &HFF Else
LEREM.ForeColor = 0
  If LastStr(3) <> LIS.Caption Then LIS.ForeColor = &HFF Else
LIS.ForeColor = 0
  If LastStr(7) <> LCIB.Caption Then LCIB.ForeColor = &HFF Else
LCIB.ForeColor = 0
  SaveLast

  'Decoded instruction buffer
  Dim i As Integer
  ListDIB.Clear
  If Proj.CPU.Fetch Then
    Call ListDIB.AddItem("<N/A>")
  Else
    If UBound(Proj.CPU.DIB) = -1 Then Call
ListDIB.AddItem("<Empty>")
    For i = 0 To UBound(Proj.CPU.DIB)
      Call ListDIB.AddItem(IIf(Proj.CPU.eDP = i, "--> ", "    ") +
DI2Str(Proj.CPU.DIB(i)))
    Next
  End If

  fhCU.Caption = CStr(Proj.TickCount)
End Sub

'-----------------------------------------------'
' Private Sub SaveLast()                        '
'                                               '
' Saves the state of all register in order to   '
' highlight them as they change. Is called by   '
' Update() after getting new values for them    '
'-----------------------------------------------'
Private Sub SaveLast()
  LastStr(0) = LFE.Caption
  LastStr(1) = LFREM.Caption
  LastStr(2) = LEREM.Caption
  LastStr(3) = LIS.Caption
  LastStr(7) = LCIB.Caption
End Sub

'-----------------------------------------'
' Private Sub ListDIB_Click()             '
'                                         '
' Unselects the DIB every time the user   '
' clicks on it to remove the blue line    '
'-----------------------------------------'
Private Sub ListDIB_Click()
  ListDIB.ListIndex = -1
  fhCU.SetFocus
End Sub

'-----------------------------------------'
' Private Sub ListDIB_GotFocus()          '
'                                         '
' Unselects the DIB every time the user   '
' clicks on it to remove the blue line    '
'-----------------------------------------'
Private Sub ListDIB_GotFocus()
```

```
    ListDIB.ListIndex = -1
    fhCU.SetFocus
End Sub
```

22.9. fhRAM

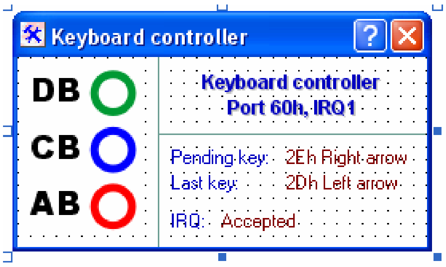



```
Option Explicit

'-----------------------------------------'
' Public declarations in this module:     '
'                                         '
' PROCEDURES:                             '
'   Reset                                 '
'   Update                                '
'-----------------------------------------'

'Prevent specific events
Private BlockSetEditCell As Boolean
'Fonts
Private fFixed As TFnt      'Fixed cells
Private fDef As TFnt        'Usual cells
Private fSel As TFnt        'Cursor
Private fIP As TFnt         'IP
Private fSP As TFnt         'SP
Private fIPi1 As TFnt       'IP cur instruction
Private fIPi2 As TFnt       'IP cur instruction (uncertain)
Private fCode As TFnt       'Memory loaded with compiled code
Private fVal As TFnt        'Other memory with nonzero values

'---------------------------------------------'
' Public Sub Init()                           '
'                                             '
' DESCRIPTION: initialises this module.       '
'---------------------------------------------'
Public Sub Init()
  'SG fonts - initial params
  fSel.FaceName = "Courier New"
  fSel.Size = -11
  fSel.Weight = 400
  fFixed = fSel
  fDef = fSel
  fSP = fSel
  fIP = fSel
  fIPi1 = fSel
  fIPi2 = fSel
  fCode = fSel
  fVal = fSel
  'SG fonts - colors
  fFixed.ForeColor = GetSysColor(COLOR_BTNTEXT)
  fFixed.BackColor = GetSysColor(COLOR_BTNFACE)
  fDef.ForeColor = &HC0C0C0
  fDef.BackColor = &HFFFFFF
  fSel.ForeColor = GetSysColor(COLOR_HIGHLIGHTTEXT)
  fSel.BackColor = GetSysColor(COLOR_HIGHLIGHT)
  fSP.ForeColor = 0
  fSP.BackColor = 65280
  fIP.ForeColor = &H80
  fIP.BackColor = &HFFB366
  fIP.Weight = 800
  fIPi1.ForeColor = &H80
  fIPi1.BackColor = &HFFE0C1
  fIPi1.Weight = 800
  fIPi2.ForeColor = &H80
  fIPi2.BackColor = &HFFF2E6
  fIPi2.Weight = 800
  fCode.ForeColor = 0
  fCode.BackColor = &H1FFFF
  fVal.ForeColor = 0
  fVal.BackColor = &HE0E0FF
  'SG fonts - create
  CreateFnt fFixed
  CreateFnt fDef
  CreateFnt fSel
  CreateFnt fIP
  CreateFnt fSP
  CreateFnt fIPi1
  CreateFnt fIPi2
  CreateFnt fCode
  CreateFnt fVal
  'Set SG options
  SG.Option(goEditing) = True
  SG.Option(goColMoving) = False
  SG.Option(goColSizing) = False
  SG.Option(goRangeSelect) = False
  SG.Option(goRowMoving) = False
  SG.Option(goRowSizing) = False
  SG.Option(goThumbTracking) = True
  'SG visual
  SG.ColCount = 17
  SG.RowCount = 4097
  SG.FixedCols = 1
  SG.FixedRows = 1
  Dim sz As Size
  Call SelectObject(SG.hdc, fDef.fntFont)
  Call GetTextExtentPoint32(SG.hdc, "FF", Len("FF"), sz)
  SG.DefaultColWidth = sz.cx + 4
  SG.DefaultRowHeight = sz.cy + 2
  Call SelectObject(SG.hdc, fDef.fntFont)
  Call GetTextExtentPoint32(SG.hdc, "FFFF", Len("FFFF"), sz)
  SG.ColWidths(0) = sz.cx + 4

  BlockSetEditCell = False

  Reset
  Update
End Sub

'-----------------------------'
' Public Sub Reset()          '
'                             '
' Initialises the module and  '
' related variables.          '
'-----------------------------'
Public Sub Reset()
  'Clear RAM memory
  Dim i As Long
  ReDim Proj.RAM(0 To 65535) 'Redim sets everything to zeroes
End Sub

'---------------------------------------------'
```

```
' Private Sub Form_Unload(Cancel As Integer) '
'                                            '
' DESCRIPTION: Event handler for Form_Unload '
'   unloads the form if the application is   '
'   really shutting down, and just hides the '
'   form in case the user requested to close '
'   it.                                      '
'--------------------------------------------'
Private Sub Form_Unload(Cancel As Integer)
  If Not Appp.Terminating Then
    Cancel = 1
    fhRAM.Hide
  End If
End Sub

'---------------------------'
' Private Sub Form_Resize() '
'---------------------------'
Private Sub Form_Resize()
  SG.Width = ClientW.Width - 240
  SG.Height = ClientH.Height - 240
  NRun.Width = ClientW.Width - 240
  NRun.Height = ClientH.Height - 240
End Sub

'----------------------------------------'
' Public Sub Update()                    '
'                                        '
' Updates the contents of the window to  '
' reflect changes to the state of the    '
' simulation.                            '
'----------------------------------------'
Public Sub Update()
  Dim rct As RECT
  Call GetClientRect(SG.hwnd, rct)
  Call InvalidateRect(SG.hwnd, rct, True)
  'Availability
  SG.Visible = Proj.Running
  NRun.Visible = Not Proj.Running
End Sub

'------------------------------------------'
' Private Sub SG_OnDrawCell(ByVal ACol As  '
'   Long, ByVal ARow As Long, ByVal RectFX '
'   As Long, ByVal RectFY As Long, ByVal   '
'   RectTX As Long, ByVal RectTY As Long)  '
'------------------------------------------'
Private Sub SG_OnDrawCell(ByVal ACol As Long, ByVal ARow As Long,
ByVal RectFX As Long, ByVal RectFY As Long, ByVal RectTX As Long,
ByVal RectTY As Long)
  'Initialise vars
  Dim f As TFnt     'font to be used
  Dim st As String  'value to be printed
  Dim rct As RECT   'clipping rectangle for text
  Dim i As Long
  rct.Left = RectFX + 2
  rct.Top = RectFY + 1
  rct.Right = RectTX - 2
  rct.Bottom = RectTY - 1

  'Determine the value and the colors
  If ARow = 0 And ACol = 0 Then
    f = fFixed
    st = ""
  ElseIf ARow = 0 Then
    f = fFixed
    st = Dec2Hex(ACol - 1, 2)
  ElseIf ACol = 0 Then
    f = fFixed
    st = Dec2Hex((ARow - 1) * 16, 4)
  Else
    i = (ARow - 1) * 16 + ACol - 1

    'Selection
    If SG.GetSelX = ACol And SG.GetSelY = ARow Then
      f = fSel
    'SP
    ElseIf i = Proj.CPU.SP Or i = Proj.CPU.SP + 1 Then
      f = fSP
    'IP
    ElseIf i = Proj.CPU.IP Then
      f = fIP
    'IP instruction
    ElseIf (i >= Proj.CPU.IP - Len(Proj.CPU.CIB)) And (i <
Proj.CPU.IP - Len(Proj.CPU.CIB) + InstructionLen(Proj.RAM(Proj.CPU.IP
- Len(Proj.CPU.CIB)))) Then
      f = fIPi1
    'IP instruction (uncertain)
    ElseIf (i >= Proj.CPU.IP - Len(Proj.CPU.CIB)) And (i < 3 +
Proj.CPU.IP - Len(Proj.CPU.CIB) +
InstructionLen(Proj.RAM(Proj.CPU.IP - Len(Proj.CPU.CIB)))) And
InstructionMem(Proj.RAM(Proj.CPU.IP - Len(Proj.CPU.CIB))) Then
      f = fIPi2
    'Code
    ElseIf i < Len(Proj.P.Code) Then
      f = fCode
    'Other nonzero values
    ElseIf Proj.RAM(i) <> 0 Then
      f = fVal
    'Usual cell
    Else
      f = fDef
    End If
    st = Dec2Hex(CLng(Proj.RAM(i)), 2)
  End If

  'Draw frame (ie background)
  Call SelectObject(SG.hdc, f.fntBrush)
  Call SelectObject(SG.hdc, GetStockObject(7)) 'BLACK_PEN
  Call Rectangle(SG.hdc, RectFX - 1, RectFY - 1, RectTX + 1,
RectTY + 1)
  'Draw text
  Call FntWrite(f, SG.hdc, st, rct)
End Sub

'------------------------------------------'
' Private Sub SG_OnGetEditText(ByVal ACol As  '
'   Long, ByVal ARow As Long, Value As String) '
'------------------------------------------'
Private Sub SG_OnGetEditText(ByVal ACol As Long, ByVal ARow As
Long, Value As String)
  Value = Dec2Hex(CLng(Proj.RAM((ARow - 1) * 16 + ACol - 1)), 2)
End Sub

'-----------------------------------------'
' Private Sub SG_OnSetEditText(ByVal ACol '
'   As Long, ByVal ARow As Long, ByVal    '
'   Value As String)                      '
'-----------------------------------------'
Private Sub SG_OnSetEditText(ByVal ACol As Long, ByVal ARow As
Long, ByVal Value As String)
  'Tests
  If BlockSetEditCell Then Exit Sub
  If SG.EditorMode Then Exit Sub
  BlockSetEditCell = True
  'Valid value?
  If Len(Value) < 1 Or Len(Value) > 2 Then
    Call MsgBox("The length of the number must be between 1 and 2
characters.", vbOKOnly + vbInformation)
    BlockSetEditCell = False
    Exit Sub
  End If
  If Not TestCharset(UCase(Value), "0123456789ABCDEF") Then
    Call MsgBox("The number must contain only characters 0-9 and
A-F to ensure that it is a hexadecimal number.", vbOKOnly +
vbInformation)
    BlockSetEditCell = False
    Exit Sub
  End If
  'Write to memory
  Proj.RAM((ARow - 1) * 16 + ACol - 1) = Hex2Dec(Value)
  'Finished
  BlockSetEditCell = False
End Sub

'------------------------------------------'
' Private Sub SG_OnMouseDown(ByVal MouseButton '
'   As StringGridVBProj.TxMouseButton)     '
'------------------------------------------'
Private Sub SG_OnMouseDown(ByVal MouseButton As
StringGridVBProj.TxMouseButton)
  If MouseButton = mbRight Then PopupMenu MIOptions
End Sub

Private Sub BtnGoto_Click()
'On Error GoTo Errrr
'   'Input address
'   Dim Addr As String, n As Long
'retry:
'   Addr = InputBox("Please enter the address that you would like
to see:", "Go to offset", CStr(rgFrom * 16))
'   If Addr = "" Then Exit Sub
'   If Right(Addr, 1) = "h" Then
'     If Not TestCharset(Left(Addr, Len(Addr) - 1),
"0123456789ABCDEF") Then
```

```
'       Call Errr("The string you entered is not a number. It should
contain 0-9 for decimal numbers, 0-9 & A-F for hexadecimal numbers
ending with 'h', and 0-1 for binary numbers ending with 'b'.")
'       GoTo retry
'     End If
'     n = Hex2Dec(Left(Addr, Len(Addr) - 1))
'   ElseIf Right(Addr, 1) = "b" Then
'     If Not TestCharset(Left(Addr, Len(Addr) - 1), "01") Then
'       Call Errr("The string you entered is not a number. It should
contain 0-9 for decimal numbers, 0-9 & A-F for hexadecimal numbers
ending with 'h', and 0-1 for binary numbers ending with 'b'.")
'       GoTo retry
'     End If
'     n = Bin2Dec(Left(Addr, Len(Addr) - 1))
'   Else
'     If Not TestCharset(Left(Addr, Len(Addr) - 1), "0123456789") Then
'       Call Errr("The string you entered is not a number. It should
contain 0-9 for decimal numbers, 0-9 & A-F for hexadecimal numbers
ending with 'h', and 0-1 for binary numbers ending with 'b'.")
'       GoTo retry
'     End If
'     n = CLng(Addr)
'   End If
'   If n < 0 Or n > 65535 Then
'     Call Errr("The address should be between 0 and 65535 (0000h and
0FFFFh)")
'     GoTo retry
'   End If
'   rgFrom = n \ 16
'   If rgFrom > 4095 - 7 Then rgFrom = 4095 - 7
'   Update
'   Exit Sub
'
'Errrr:
'   Call Errr("The address should be between 0 and 65535 (0000h and
0FFFFh)")
'   Resume retry
End Sub

Private Sub BtnShowIP_Click()
'   rgFrom = Proj.CPU.IP \ 16
'   If rgFrom > 4095 - 7 Then rgFrom = 4095 - 7
'   Update
End Sub

Private Sub BtnShowSP_Click()
'   rgFrom = Proj.CPU.SP \ 16
'   If rgFrom > 4095 - 7 Then rgFrom = 4095 - 7
'   Update
End Sub
```

22.10. fdKeyboard




```
Option Explicit

'Key code to key name map
Private KeyName(0 To 51) As String
'Key code to be sent to programs
Private CurKey As Integer
'Last key pressed - just for user information
Private LastKey As Integer
'Whether interrupt has been sent
Private InterruptSent As Boolean

'-------------------------'
' Public Sub Init()       '
'                         '
' Initializes this module '
'-------------------------'
Public Sub Init()
  'Initialize key names
  Dim i As Integer
  For i = 0 To 25
    KeyName(i) = Chr$(i + 65)
  Next
  KeyName(26) = "."
  KeyName(27) = "Enter"
  KeyName(28) = "Spacebar"
  KeyName(29) = "="
  For i = 0 To 9
    KeyName(i + 30) = Chr$(i + 48)
  Next
  KeyName(40) = "Numpad ."
  KeyName(41) = "/"
  KeyName(42) = "*"
  KeyName(43) = "-"
  KeyName(44) = "+"
  KeyName(45) = "Left arrow"
  KeyName(46) = "Right arrow"
  KeyName(47) = "Up arrow"
  KeyName(48) = "Down arrow"
  KeyName(49) = "Circle"
  KeyName(50) = "Square"
  KeyName(51) = "Triangle"
  'Reset once
  Reset
End Sub

'-------------------'
' Public Sub Reset() '
'                   '
' Callback for pIO.  '
'-------------------'
Public Sub Reset()
  CurKey = -1
  LastKey = -1
  InterruptSent = False
  Update
End Sub

'------------------'
' Public Sub Tick() '
'                  '
' Callback for pIO. '
'------------------'
Public Sub Tick()
  'Send interrupt
  If CurKey <> -1 And Not InterruptSent Then
    InterruptSent = IRQ(1)
    Update
  End If
End Sub

'---------------------------------'
' Public Function PortRead(PortNum '
'   As Integer) As Long           '
'                                 '
' Callback for pIO.               '
'---------------------------------'
```

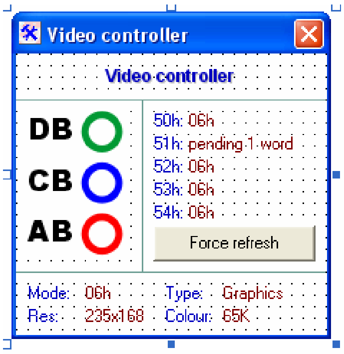
```
Public Function PortRead(PortNum As Integer) As Long
  If PortNum <> &H60 Then PortRead = 999999: Exit Function
  If CurKey = -1 Then
    PortRead = 65535
  Else
    PortRead = CurKey
    LastKey = CurKey
    CurKey = -1
    Update
  End If
End Function

'------------------------------'
' Public Sub PortWrite(PortNum '
'   As Integer, Dt As Long)    '
'                              '
' Callback for pIO.            '
'------------------------------'
Public Sub PortWrite(PortNum As Integer, Dt As Long)
  'Nothing
End Sub

'--------------------------'
' Private Sub Update()     '
'                          '
' Updates screen to reflect '
' current situation.       '
'--------------------------'
Private Sub Update()
  If CurKey = -1 Then LPending.Caption = "None" Else
LPending.Caption = Dec2Hex(CLng(CurKey), 2) + "h " +
KeyName(CurKey)
  If LastKey = -1 Then LLast.Caption = "None" Else LLast.Caption
= Dec2Hex(CLng(LastKey), 2) + "h " + KeyName(LastKey)
  If CurKey = -1 Then
    LIRQ.Caption = "N/A"
  Else
    LIRQ.Caption = IIf(InterruptSent, "Accepted. Waiting port
read.", "Rejected. Retrying.")
  End If
End Sub

'----------------------------------------'
' Public Sub KeyDown(KeyCode As Integer) '
'                                        '
' This sub is called by keyboard windows '
' when a key is pressed. The event is    '
' processed in Tick method.              '
'----------------------------------------'
Public Sub KeyDown(KeyCode As Integer)
  If CurKey = -1 Then
    CurKey = KeyCode
    InterruptSent = IRQ(1)
    Update
  End If
End Sub
```

22.11. fdSpeaker


```
Option Explicit

'Current state - low/high
Private spkState As Boolean
'Current frequency - 20/65535*X Hz
Private spkFreq As Long

'--------------------------'
' Public Sub Init()        '
'                          '
' Initializes this module  '
'--------------------------'
Public Sub Init()
  'Reset once
  Reset
End Sub

'---------------------'
' Public Sub Reset()  '
'                     '
' Callback for pIO.   '
'---------------------'
Public Sub Reset()
  spkState = False
  spkFreq = 0
  TmrFreq.Enabled = False
  Update
End Sub

'-------------------'
' Public Sub Tick() '
'                   '
' Callback for pIO. '
'-------------------'
Public Sub Tick()
  '
End Sub

'----------------------------------'
' Public Function PortRead(PortNum '
'   As Integer) As Long            '
'                                  '
' Callback for pIO.                '
'----------------------------------'
Public Function PortRead(PortNum As Integer) As Long
  If PortNum <> &H80 Then PortRead = 999999: Exit Function
  PortRead = spkFreq
End Function

'------------------------------'
' Public Sub PortWrite(PortNum '
'   As Integer, Dt As Long)    '
'                              '
' Callback for pIO.            '
'------------------------------'
Public Sub PortWrite(PortNum As Integer, Dt As Long)
  If PortNum <> &H80 Then Exit Sub
  spkFreq = Dt
  If Dt = 0 Then spkState = False
  If Dt = 1 Then spkState = True
  If spkFreq >= 2 Then
    TmrFreq.Interval = Int(1 / (20 / 65535 * spkFreq) * 1000)
```

```
    TmrFreq.Enabled = True
  Else
    TmrFreq.Enabled = False
  End If
  Update
End Sub

'---------------------------'
' Private Sub Update()      '
'                           '
' Updates screen to reflect '
' current situation.        '
'---------------------------'
Private Sub Update()
  'Refresh picture
  If LState.Caption <> IIf(spkState, "High", "Low") Then Pct_Paint
  LState.Caption = IIf(spkState, "High", "Low")
  LFreq.Caption = IIf(spkFreq = 0 Or spkFreq = 1, "N/A", CStr(Int(10
/ 65535 * spkFreq * 100) / 100) + " Hz")
End Sub

'-------------------------'
' Private Sub Pct_Paint() '
'-------------------------'
Private Sub Pct_Paint()
  If spkState Then
    Call Pct.PaintPicture(PctH.Picture, 0, 0)
  Else
    Call Pct.PaintPicture(PctL.Picture, 0, 0)
  End If
End Sub

'-----------------------------'
' Private Sub TmrFreq_Timer() '
'-----------------------------'
Private Sub TmrFreq_Timer()
  spkState = Not spkState
  Update
End Sub
```

22.12. fdVideo



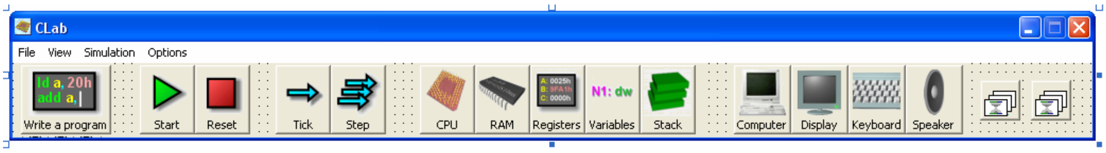

```
Option Explicit

'----------------------------------------'
' Public declarations in this module:    '
'                                        '
' PROCEDURES:                            '
'   Reset                                '
'   SetVideoMode                         '
'   UpdateScr                            '
'----------------------------------------'

'----------------------------------------'
' PORT STATE VARIABLES:                  '
'----------------------------------------'
' port51state values:                    '
'    -1: no operations                   '
' other: first word written to 51h       '
'----------------------------------------'
' port53state values:                    '
'                                        '
' 0000h: no operation                    '
' 0100h: set pixel                       '
' 0500h: set pen colour                  '
' 0600h: set brush colour                '
' 1000h: draw line                       '
' 1010h: set pen position                '
' 1020h: continue line                   '
' 2000h: draw empty circle               '
' 2001h: draw filled circle              '
' 3000h: draw empty rectangle            '
' 3001h: draw filled rectangle           '
'----------------------------------------'
Private port51state As Long
Private port53state As Long
Private port53var1 As Long  'depends on function
Private port53var2 As Long  'depends on function

'--------------------------------------------'
' Private Sub Form_Unload(Cancel As Integer) '
'                                            '
' DESCRIPTION: Event handler for Form_Unload '
'   unloads the form if the application is   '
'   really shutting down, and just hides the '
'   form in case the user requested to close '
'   it.                                      '
'--------------------------------------------'
Private Sub Form_Unload(Cancel As Integer)
  If Not Appp.Terminating Then
    Cancel = 1
    fdVideo.Hide
  End If
End Sub

'----------------------------------------------'
' private Sub SetVideoMode(ModeNum As Long)    '
'                                              '
' DESCRIPTION: sets the specified video mode.  '
'                                              '
' PARAMETERS:                                  '
' ModeNum - a byte specifying video mode as    '
'   described in the manual. This is the same  '
'   code that is used with OUT instruction.    '
'                                              '
' NOTES: It is up to the caller to make sure   '
'   that UpdateScr is called to update image.  '
'----------------------------------------------'
Private Sub SetVideoMode(ModeNum As Long)
  Dim i
  i = ModeNum And 127
  'Check
  If (i < 1) Or ((i > 7) And (i < 129)) Or (i > 135) Then Exit Sub
  'Remember mode
  Proj.Video.Mode = i
  'Modes
  If i = 1 Then
    Proj.Video.mdColors = 0  'do not change these parameters!
    Proj.Video.mdResX = 40   'drawing procedures use constants
instead
    Proj.Video.mdResY = 15
    Proj.Video.mdType = 0
```

```
  ElseIf i = 2 Then
    Proj.Video.mdColors = 1  'do not change these parameters!
    Proj.Video.mdResX = 40   'drawing procedures use constants
instead
    Proj.Video.mdResY = 15
    Proj.Video.mdType = 0
  ElseIf i = 3 Then
    Proj.Video.mdColors = 0  'do not change these parameters!
    Proj.Video.mdResX = 208  'drawing procedures use constants
instead
    Proj.Video.mdResY = 156
    Proj.Video.mdType = 1
  ElseIf i = 4 Then
    Proj.Video.mdColors = 1  'do not change these parameters!
    Proj.Video.mdResX = 104  'drawing procedures use constants
instead
    Proj.Video.mdResY = 78
    Proj.Video.mdType = 1
  ElseIf i = 5 Then
    Proj.Video.mdColors = 2  'do not change these parameters!
    Proj.Video.mdResX = 74   'drawing procedures use constants
instead
    Proj.Video.mdResY = 55
    Proj.Video.mdType = 2
  ElseIf i = 6 Then
    Proj.Video.mdColors = 3  'do not change these parameters!
    Proj.Video.mdResX = 52   'drawing procedures use constants
instead
    Proj.Video.mdResY = 39
    Proj.Video.mdType = 1
  ElseIf i = 7 Then
    Proj.Video.mdColors = 4  'do not change these parameters!
    Proj.Video.mdResX = 42   'drawing procedures use constants
instead
    Proj.Video.mdResY = 32
    Proj.Video.mdType = 1
  End If

  'Create vDC
  If Proj.Video.vDC.IsCreated Then Proj.Video.vDC.Destroy
  If Proj.Video.Mode >= 3 Then
    'Call Proj.Video.vDC.Create(fiComp.PctScr.hdc,
CLng(Proj.Video.mdResX), CLng(Proj.Video.mdResY))
    Call Proj.Video.vDC.Create(fiDisplay.PctScr.hdc,
CLng(Proj.Video.mdResX), CLng(Proj.Video.mdResY))
  Else
    'Call Proj.Video.vDC.Create(fiComp.PctScr.hdc, 320, 240)
    Call Proj.Video.vDC.Create(fiDisplay.PctScr.hdc, 320, 240)
    Dim fFont As New StdFont
    With fFont
      .Bold = False
      .charset = 1  'DEFAULT_CHARSET
      .Italic = False
      .Name = "Courier New"
      .Size = 12
      .Strikethrough = False
      .Underline = False
      .Weight = 500
    End With
    Dim fnt_sz As Size
    Set Proj.Video.vDC.Font = fFont
    Call GetTextExtentPoint32(Proj.Video.vDC.hdc, "W", 1, fnt_sz)
    Proj.Video.mdFntX = fnt_sz.cx
    Proj.Video.mdFntY = fnt_sz.cy
    Proj.Video.vDC.BackStyle = BS_OPAQUE
    Proj.Video.vDC.BackColor = 0
    Proj.Video.vDC.ForeColor = &HFFFFFF
  End If
End Sub

'-----------------------------------------'
' Public Sub Update()                     '
'                                         '
' Updates the contents of the window to   '
' reflect changes to the state of the     '
' simulation.                             '
'-----------------------------------------'
Public Sub Update()
  'Video Mode
  LMode.Caption = Dec2Hex(CLng(Proj.Video.Mode), 2) + "h"
  LRes.Caption = CStr(Proj.Video.mdResX) + "x" +
CStr(Proj.Video.mdResY)
  Select Case Proj.Video.mdColors
  Case 0
    LColor.Caption = "B/W"
  Case 1
    LColor.Caption = "16"
  Case 2
    LColor.Caption = "256 pal"
  Case 3
    LColor.Caption = "64K"
  Case 4
    LColor.Caption = "16M"
  End Select
  Select Case Proj.Video.mdType
  Case 0
    LMType.Caption = "Text"
  Case 1
    LMType.Caption = "Graphics"
  Case 2
    LMType.Caption = "Graphics (pal)"
  End Select
  'Ports state
  LPort50.Caption = "Mode " + Dec2Hex(CLng(Proj.Video.Mode), 2) +
"h"
  LPort51.Caption = IIf(port51state = 1, "Pending 1 word",
"Ready")
  LPort52.Caption = "Offset " + Dec2Hex(Proj.Video.MemOff, 4) +
"h"
  LPort53.Caption = IIf(port53state = 0, "Ready", "Not done yet")
  LPort54.Caption = IIf(Proj.Video.autoUpdate, "Auto refresh",
"Manual refresh")
  'Update screen
  If Not Proj.Running Then fdVideo.UpdateScr
  If Proj.Running And Proj.Paused Then If Proj.Video.autoUpdate
Then fdVideo.UpdateScr
End Sub

'-----------------------------------------'
' Public Sub UpdateScr()                  '
'                                         '
' DESCRIPTION:                            '
'   Draws the video memory on the screen(s) '
'-----------------------------------------'
Public Sub UpdateScr()
  Dim i As Long, A As Long
  Dim x As Integer, y As Integer

  'Output video memory to memory DC
  For x = 0 To Proj.Video.mdResX - 1
    For y = 0 To Proj.Video.mdResY - 1
      If (Proj.Video.Mode And 127) = 1 Then       'text mono
        i = Proj.RAM(Proj.Video.MemOff + y * 40 + x)
        If i = 0 Then i = 32
        Call Proj.Video.vDC.PrintText(Chr(i), x *
Proj.Video.mdFntX, y * Proj.Video.mdFntY, 1000, 1000, 0)
      ElseIf (Proj.Video.Mode And 127) = 2 Then  'text color
        A = Proj.RAM(Proj.Video.MemOff + 2 * y * 40 + 2 * x + 1)
        If A = 0 Then A = 32
        Proj.Video.vDC.BackColor = IIf((A And 16) > 0, 1, 0) *
&H800000 + IIf((A And 32) > 0, 1, 0) * &H8000 + IIf((A And 64) >
0, 1, 0) * &H80 + IIf((A And 128) > 0, 1, 0) * &H7F7F7F
        Proj.Video.vDC.ForeColor = (A And 1) * &H800000 + IIf((A
And 2) > 0, 1, 0) * &H8000 + IIf((A And 4) > 0, 1, 0) * &H80 +
IIf((A And 8) > 0, 1, 0) * &H7F7F7F
        Call
Proj.Video.vDC.PrintText(Chr(Proj.RAM(Proj.Video.MemOff + 2 * y *
40 + 2 * x)), x * Proj.Video.mdFntX, y * Proj.Video.mdFntY, 1000,
1000, 0)
      ElseIf (Proj.Video.Mode And 127) = 3 Then  'graph mono
        i = y * Proj.Video.mdResY + x
        If (Proj.RAM(Proj.Video.MemOff + Int(i / 8)) And (2 ^ (i -
Int(i / 8) * 8))) > 0 Then i = 16777215 Else i = 0
        Call SetPixelV(Proj.Video.vDC.hdc, x, y, i)
      ElseIf (Proj.Video.Mode And 127) = 4 Then  'graph 4bit
        A = Proj.RAM(Proj.Video.MemOff + Int((y * 104 + x) / 2))
        If Int((y * 104 + x) / 2) = Int((y * 104 + x) / 2 + 0.5)
Then 'use most significant
          A = Int(A / 8) And &HF
        Else 'use least significant
          A = A And &HF
        End If
        i = (A And 1) * &H800000 + IIf((A And 2) > 0, 1, 0) *
&H8000 + IIf((A And 4) > 0, 1, 0) * &H80 + IIf((A And 8) > 0, 1,
0) * &H7F7F7F
        Call SetPixelV(Proj.Video.vDC.hdc, x, y, i)
      ElseIf (Proj.Video.Mode And 127) = 5 Then  'graph 8bit
        Call SetPixelV(Proj.Video.vDC.hdc, x, y,
Proj.Video.PalMem(Proj.RAM(Proj.Video.MemOff + y * 74 + x)))
      ElseIf (Proj.Video.Mode And 127) = 7 Then  'graph 24bit
        i = CLng(Proj.RAM(Proj.Video.MemOff + (y * 42 + x) * 3)) +
CLng(Proj.RAM(Proj.Video.MemOff + (y * 42 + x) * 3 + 1)) * 256 +
CLng(Proj.RAM(Proj.Video.MemOff + (y * 42 + x) * 3 + 2)) * 65536
        Call SetPixelV(Proj.Video.vDC.hdc, x, y, i)
      End If
    Next
```

```
  Next

  'Update screen
  Call StretchBlt(fiComp.PctScr.hdc, 0, 0, fiComp.PctScr.Width,
fiComp.PctScr.Height, _
                  Proj.Video.vDC.hdc, 0, 0, Proj.Video.vDC.Width,
Proj.Video.vDC.Height, SRCCOPY)
  Call StretchBlt(fiDisplay.PctScr.hdc, 0, 0, fiDisplay.PctScr.Width,
fiDisplay.PctScr.Height, _
                  Proj.Video.vDC.hdc, 0, 0, Proj.Video.vDC.Width,
Proj.Video.vDC.Height, SRCCOPY)
End Sub

'-------------------------'
' Public Sub Init()       '
'                         '
' Initializes this module '
'-------------------------'
Public Sub Init()
  'Create vDC
  Set Proj.Video.vDC = New VirtualDC
  'Reset once
  Reset
  Update
End Sub

'-------------------'
' Public Sub Reset() '
'                   '
' Callback for pIO.  '
'-------------------'
Public Sub Reset()
  Dim i As Integer, A As Integer
  Dim R As Long, g As Long, b As Long
  'Init palette memory
  For i = 1 To 7
    For A = 0 To 31
      Proj.Video.PalMem((i - 1) * 32 + A) = CLng(IIf((i And 1) > 0,
A, 0)) * 8 + CLng(IIf((i And 2) > 0, A, 0)) * 2048 + CLng(IIf((i And
4) > 0, A, 0)) * 524288
    Next
  Next
  'RAM video memory offset
  Proj.Video.MemOff = 57344 '&HE000 - &HEFFF
  'Set screen mode
  Call SetVideoMode(1)

  'Reset port states
  port51state = -1
  port53state = 0
  'Other variables
  Proj.Video.autoUpdate = True
End Sub

'------------------'
' Public Sub Tick() '
'                  '
' Callback for pIO. '
'------------------'
Public Sub Tick()

End Sub

'---------------------------------'
' Public Function PortRead(PortNum '
'   As Integer) As Long           '
'                                 '
' Callback for pIO.               '
'---------------------------------'
Public Function PortRead(PortNum As Integer) As Long
  Select Case PortNum
    Case &H50
      '--- SCREEN MODE ---'
      PortRead = Proj.Video.Mode

    Case &H51
      '--- PALETTE ---'
      PortRead = 0

    Case &H52
      '--- MEMORY ---'
      PortRead = Proj.Video.MemOff

    Case &H53
      '--- DRAWING ---'
      PortRead = 0

    Case &H54
      '--- UPDATING ---'
      PortRead = IIf(Proj.Video.autoUpdate, 1, 0)

    Case Else
      PortRead = 999999
      Exit Function
  End Select
End Function

'--------------------------------'
' Public Sub PortWrite(PortNum   '
'   As Integer, Dt As Long)      '
'                                '
' Callback for pIO.              '
'--------------------------------'
Public Sub PortWrite(PortNum As Integer, Dt As Long)
  Dim l1 As Long
  Select Case PortNum
    Case &H50
      '--- SCREEN MODE ---'
      If ((Dt >= 1) And (Dt <= 7)) Or ((Dt >= 129) And (Dt <=
135)) Then
        Call SetVideoMode(Dt)
        UpdateScr
      End If

    Case &H51
      '--- PALETTE ---'
      If port51state = -1 Then
        port51state = Dt
      Else
        l1 = port51state And 255
        Proj.Video.PalMem(l1) = CLng((port51state And 65280) \
256) + Dt * CLng(256)
        port51state = -1
      End If

    Case &H52
      '--- MEMORY ---'
      Proj.Video.MemOff = Dt

    Case &H53
      '--- DRAWING ---'
      If port53state = 0 Then
        'Get function
        port53state = Dt
        If (port53state = &H100) Or (port53state = &H500) Or _
           (port53state = &H600) Or (port53state = &H1000) Or _
           (port53state = &H1010) Or (port53state = &H1020) Or _
           (port53state = &H2000) Or (port53state = &H2001) Or _
           (port53state = &H3000) Or (port53state = &H3001) _
           Then port53state = 0
      Else
        Select Case port53state
          Case &H100
            '--- Set pixel ---'
          Case &H500
            '--- Set pen colour ---'
          Case &H600
            '--- Set brush colour ---'
          Case &H1000
            '--- Draw line ---'
          Case &H1010
            '--- Set pen position ---'
          Case &H1020
            '--- Continue line ---'
          Case &H2000
            '--- Draw empty circle ---'
          Case &H2001
            '--- Draw filled circle ---'
          Case &H3000
            '--- Draw empty rectangle ---'
          Case &H3001
            '--- Draw filled rectangle ---'
          Case Else
            port53state = 0
        End Select
      End If

    Case &H54
      '--- UPDATING ---'
      If Dt = 0 Then
        Proj.Video.autoUpdate = False
      ElseIf Dt = 1 Then
        Proj.Video.autoUpdate = True
      Else
        If Not fiMain.MIRefNothing.Checked Then UpdateScr
      End If
```

```
  End Select
End Sub

'------------------------------'

' Private Sub BtnRefresh_Click() '
'--------------------------------'
Private Sub BtnRefresh_Click()
  UpdateScr
End Sub
```

22.13. fiMain




```
Option Explicit

'---------------------------------------'
' Public declarations in this module:   '
'                                       '
' PROCEDURES:                           '
'   Init                                '
'   WindowProc                          '
'---------------------------------------'

'Indicates whether a form was visible prior to minimizing
Private frmWasVisible(0 To 20) As Boolean
'Time (GetTickCount) when the last Step was executed (Run mode)
Private lastStepTime As Long

'---------------------------------------'
' Public Sub Init()                     '
'                                       '
' Initialises this form. This is called '
' before the form is actually shown.    '
'---------------------------------------'
Public Sub Init()
  'Need to set Top to 0 because when visual styles are used
  'it is positions of client areas, and not captions, that
  'stay fixed. So a window's real Top is higher on screen
  'with visual styles.
  Top = 0

  'Change visuals depending on OS
  If Appp.RunningOnWinXP Then
    fiMain.BackColor = &HEDEFEF 'comctl32 v6.0 gives us toolbars with
this particular color
    Height = 1830
  Else
    fiMain.BackColor = GetSysColor(COLOR_BTNFACE)
    Height = 1710
  End If

  'Install own window procedure
  Hook

  'Init toolbars
  SetToolbarButtons
End Sub

'-------------------------------------------'
' Private Sub Form_Unload(Cancel As Integer) '
'                                           '
' DESCRIPTION: Unlike event handlers for    '
'   Unload on all other forms, this handler '
'   should shut down the application in a   '
'   proper way, which involves unloading all '
'   forms. After that the application will  '
'   terminate.                              '
'                                           '
' NOTES: prior to unloading forms, this proc '
'   will query every form whether it is OK  '
'   for it to shut down, giving a chance to '
'   save work for instance. If a form wants '
'   to cancel shutdown, it will return False '
'   as the result of ShutdownQuery.         '
'-------------------------------------------'
Private Sub Form_Unload(Cancel As Integer)
  'Indicate we're really shutting down
  Appp.Terminating = True

  'Unhook this window to be sure to terminate nicely
  Unhook
  'Unload all loaded forms to terminate application.
  'We should NOT use the End statement, otherwise
  ' VB will crash. Took me an hour to figure it out.
  Dim i As Integer
  For i = Forms.Count - 1 To 0 Step -1
    Unload Forms(i)
  Next
End Sub

'-------------------------------------------'
' Private Sub Hook()                        '
'                                           '
' Hook this window to process minimize event. '
'-------------------------------------------'
Private Sub Hook()
  Appp.PrevWndProc = SetWindowLong(fiMain.hwnd, GWL_WNDPROC,
AddressOf pGlobals.WindowProc)
End Sub

'-------------------------------------------'
' Private Sub Unhook()                      '
'                                           '
' Unhook this window prior to unloading it. '
'-------------------------------------------'
Private Sub Unhook()
  'Set previous (VB) window procedure
  Call SetWindowLong(fiMain.hwnd, GWL_WNDPROC, Appp.PrevWndProc)
End Sub

'-------------------------------------------'
' Public Function WindowProc(ByVal hw As Long, '
'   ByVal uMsg As Long, ByVal wParam As Long, '
'   ByVal lParam As Long) As Long           '
'                                           '
' DESCRIPTION: Window procedure for fiMain  '
'-------------------------------------------'
Public Function WindowProc(ByVal hw As Long, ByVal uMsg _
    As Long, ByVal wParam As Long, ByVal lParam As Long) As Long

  If uMsg = WM_SYSCOMMAND Then
    If wParam = SC_MINIMIZE Then
      'Minimize everything, saving state
      frmWasVisible(0) = fhCPU.Visible
      frmWasVisible(1) = fhCU.Visible
      frmWasVisible(2) = fhRAM.Visible
      frmWasVisible(3) = fdVideo.Visible
      frmWasVisible(4) = fiComp.Visible
      frmWasVisible(5) = fsCode.Visible
      frmWasVisible(6) = fsRegs.Visible
      frmWasVisible(7) = fsStack.Visible
      frmWasVisible(8) = fsVars.Visible
      fhCPU.Hide
      fhCU.Hide
      fhRAM.Hide
      fdVideo.Hide
      fiComp.Hide
      fsCode.Hide
      fsRegs.Hide
      fsStack.Hide
      fsVars.Hide
    ElseIf wParam = SC_RESTORE Then
      fhCPU.Visible = frmWasVisible(0)
```

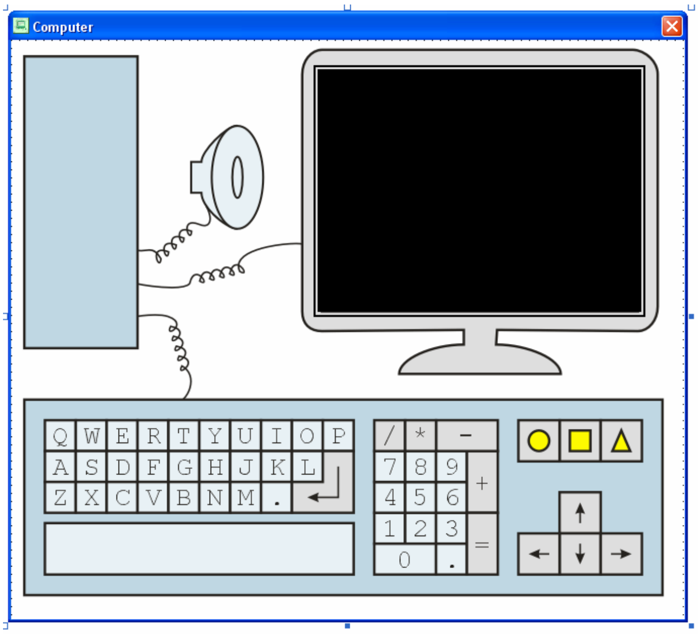
```
      fhCU.Visible = frmWasVisible(1)
      fhRAM.Visible = frmWasVisible(2)
      fdVideo.Visible = frmWasVisible(3)
      fiComp.Visible = frmWasVisible(4)
      fsCode.Visible = frmWasVisible(5)
      fsRegs.Visible = frmWasVisible(6)
      fsStack.Visible = frmWasVisible(7)
      fsVars.Visible = frmWasVisible(8)
    End If
    'Call VB window proc + default window proc
    WindowProc = CallWindowProc(Appp.PrevWndProc, hw, uMsg, wParam,
lParam)
  ElseIf uMsg = WM_NCLBUTTONDOWN Then
    If wParam = HTCAPTION Then fiMain.SetFocus
    'Call VB window proc + default window proc
    WindowProc = CallWindowProc(Appp.PrevWndProc, hw, uMsg, wParam,
lParam)
  Else
    'Call VB window proc + default window proc
    WindowProc = CallWindowProc(Appp.PrevWndProc, hw, uMsg, wParam,
lParam)
  End If
End Function

'---------------------------------'
' Private Sub BtnWritePrg_Click() '
'---------------------------------'
Private Sub BtnWritePrg_Click()
  fsCode.Show
End Sub

Private Sub Command3_Click()
  Step
  fiMain.UpdateAll
End Sub

Private Sub Command4_Click()
  Tick
  fiMain.UpdateAll
End Sub

'---------------------------------'
' Private Sub MICompBasic_Click() '
'---------------------------------'
Private Sub MICompBasic_Click()
  Proj.Complexity = 0
  fhCPU.SetComplexity
  MICompBasic.Checked = True
  MICompMed.Checked = False
  MICompFull.Checked = False
End Sub

'-------------------------------'
' Private Sub MICompMed_Click() '
'-------------------------------'
Private Sub MICompMed_Click()
  Proj.Complexity = 1
  fhCPU.SetComplexity
  MICompBasic.Checked = False
  MICompMed.Checked = True
  MICompFull.Checked = False
End Sub

'--------------------------------'
' Private Sub MICompFull_Click() '
'--------------------------------'
Private Sub MICompFull_Click()
  Proj.Complexity = 2
  fhCPU.SetComplexity
  MICompBasic.Checked = False
  MICompMed.Checked = False
  MICompFull.Checked = True
End Sub

'----------------------------'
' Private Sub MIExit_Click() '
'----------------------------'
Private Sub MIExit_Click()
  Unload fiMain
End Sub

'-----------------------------'
' Private Sub MIIComp_Click() '
'-----------------------------'
Private Sub MIIComp_Click()
  fiComp.Show
End Sub

'--------------------------------'
' Private Sub MIIDisplay_Click() '
'--------------------------------'
Private Sub MIIDisplay_Click()
  fiDisplay.Show
End Sub

'----------------------------'
' Private Sub MIIKbd_Click() '
'----------------------------'
Private Sub MIIKbd_Click()
  fiKeyboard.Show
End Sub

'--------------------------------'
' Private Sub MIISpeaker_Click() '
'--------------------------------'
Private Sub MIISpeaker_Click()
  '
End Sub

'----------------------------'
' Private Sub MIHRAM_Click() '
'----------------------------'
Private Sub MIHRAM_Click()
  fhRAM.Show
End Sub

'------------------------------'
' Private Sub MIHBuses_Click() '
'------------------------------'
Private Sub MIHBuses_Click()
  '
End Sub

'----------------------------'
' Private Sub MIHCPU_Click() '
'----------------------------'
Private Sub MIHCPU_Click()
  fhCPU.Show
End Sub

'---------------------------'
' Private Sub MIHCU_Click() '
'---------------------------'
Private Sub MIHCU_Click()
  fhCU.Show
End Sub

'----------------------------'
' Private Sub MIHALU_Click() '
'----------------------------'
Private Sub MIHALU_Click()
  '
End Sub

'----------------------------'
' Private Sub MIHKbd_Click() '
'----------------------------'
Private Sub MIHKbd_Click()
  fdKeyboard.Show
End Sub

'----------------------------'
' Private Sub MIHVid_Click() '
'----------------------------'
Private Sub MIHVid_Click()
  fdVideo.Show
End Sub

'----------------------------'
' Private Sub MIHSpk_Click() '
'----------------------------'
Private Sub MIHSpk_Click()
  fdSpeaker.Show
End Sub

'-----------------------------'
' Private Sub MIDCode_Click() '
'-----------------------------'
Private Sub MIDCode_Click()
  fsCode.Show
End Sub

'-----------------------------'
' Private Sub MIDRegs_Click() '
'-----------------------------'
Private Sub MIDRegs_Click()
```

```
  fsRegs.Show
End Sub

'-----------------------------'
' Private Sub MIDVars_Click() '
'-----------------------------'
Private Sub MIDVars_Click()
  fsVars.Show
End Sub

'------------------------------'
' Private Sub MIDStack_Click() '
'------------------------------'
Private Sub MIDStack_Click()
  fsStack.Show
End Sub

'----------------------------'
' Private Sub MINBin_Click() '
'----------------------------'
Private Sub MINBin_Click()
  Proj.NmbRep = 1
  Call fiMain.UpdateAll(True)
  MINBin.Checked = True
  MINDecS.Checked = False
  MINDecU.Checked = False
  MINHex.Checked = False
End Sub

'-----------------------------'
' Private Sub MINDecS_Click() '
'-----------------------------'
Private Sub MINDecS_Click()
  Proj.NmbRep = 3
  Call fiMain.UpdateAll(True)
  MINBin.Checked = False
  MINDecS.Checked = True
  MINDecU.Checked = False
  MINHex.Checked = False
End Sub

'-----------------------------'
' Private Sub MINDecU_Click() '
'-----------------------------'
Private Sub MINDecU_Click()
  Proj.NmbRep = 2
  Call fiMain.UpdateAll(True)
  MINBin.Checked = False
  MINDecS.Checked = False
  MINDecU.Checked = True
  MINHex.Checked = False
End Sub

'----------------------------'
' Private Sub MINHex_Click() '
'----------------------------'
Private Sub MINHex_Click()
  Proj.NmbRep = 0
  Call fiMain.UpdateAll(True)
  MINBin.Checked = False
  MINDecS.Checked = False
  MINDecU.Checked = False
  MINHex.Checked = True
End Sub

'------------------------------'
' Private Sub MISpdMax_Click() '
'------------------------------'
Private Sub MISpdMax_Click()
  MISpdMax.Checked = True
  MISpd01.Checked = False
  MISpd05.Checked = False
  MISpd20.Checked = False
  MISpd60.Checked = False
End Sub

'-----------------------------'
' Private Sub MISpd01_Click() '
'-----------------------------'
Private Sub MISpd01_Click()
  MISpdMax.Checked = False
  MISpd01.Checked = True
  MISpd05.Checked = False
  MISpd20.Checked = False
  MISpd60.Checked = False
End Sub

'-----------------------------'
```

```
' Private Sub MISpd05_Click() '
'-----------------------------'
Private Sub MISpd05_Click()
  MISpdMax.Checked = False
  MISpd01.Checked = False
  MISpd05.Checked = True
  MISpd20.Checked = False
  MISpd60.Checked = False
End Sub

'-----------------------------'
' Private Sub MISpd20_Click() '
'-----------------------------'
Private Sub MISpd20_Click()
  MISpdMax.Checked = False
  MISpd01.Checked = False
  MISpd05.Checked = False
  MISpd20.Checked = True
  MISpd60.Checked = False
End Sub

'-----------------------------'
' Private Sub MISpd60_Click() '
'-----------------------------'
Private Sub MISpd60_Click()
  MISpdMax.Checked = False
  MISpd01.Checked = False
  MISpd05.Checked = False
  MISpd20.Checked = False
  MISpd60.Checked = True
End Sub

'----------------------------------'
' Private Sub MIRefNothing_Click() '
'----------------------------------'
Private Sub MIRefNothing_Click()
  MIRefNothing.Checked = True
  MIRefUser.Checked = False
  MIRefAll.Checked = False
End Sub

'-------------------------------'
' Private Sub MIRefUser_Click() '
'-------------------------------'
Private Sub MIRefUser_Click()
  MIRefNothing.Checked = False
  MIRefUser.Checked = True
  MIRefAll.Checked = False
End Sub

'------------------------------'
' Private Sub MIRefAll_Click() '
'------------------------------'
Private Sub MIRefAll_Click()
  MIRefNothing.Checked = False
  MIRefUser.Checked = False
  MIRefAll.Checked = True
End Sub

'-------------------------------------------'
' Private Sub TmrRun_Timer(Index As Integer) '
'                                           '
' Calls Step if project is running. The     '
' procedure is not executed unless a given  '
' time interval has passed (speed control). '
' An array of timers is used to increase    '
' call frequency.                           '
'                                           '
' NOTE: UpdateAll without enforcement will  '
'   be called after each Step               '
'-------------------------------------------'
Private Sub TmrRun_Timer(Index As Integer)
  'Check timing
  Dim i As Integer
  If MISpdMax.Checked Then i = 0
  If MISpd01.Checked Then i = 100
  If MISpd05.Checked Then i = 500
  If MISpd20.Checked Then i = 2000
  If MISpd60.Checked Then i = 6000
  If GetTickCount - lastStepTime < i Then Exit Sub
  'Execute one step
  If Proj.Running And Not Proj.Paused Then
    lastStepTime = GetTickCount
    Step
    UpdateAll
  End If
End Sub
```

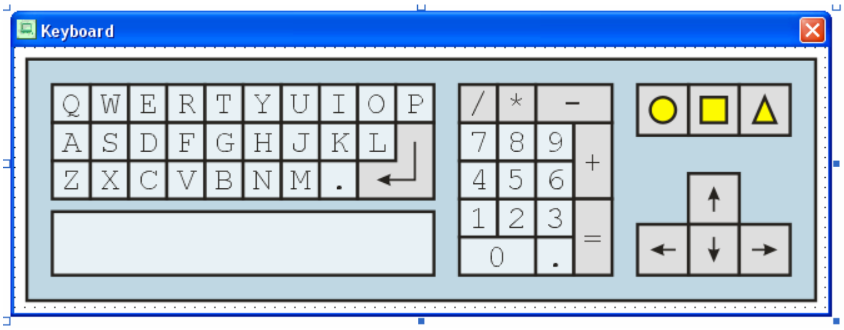
```
'-------------------------------------'
' Public Sub UpdateAll(Optional Force '
'   As Boolean = False)               '
'                                     '
' Updates all windows as necessary. Set '
' Force=True to update all windows in  '
' any situation.                       '
'-------------------------------------'
Public Sub UpdateAll(Optional Force As Boolean = False)
  If Not Force And Proj.Running And Not Proj.Paused Then
    If MIRefNothing.Checked Then Exit Sub
  End If
  If Force Or Not MIRefUser.Checked Or Not Proj.Running Then
    'Update all windows
    fhCPU.Update
    fhCU.Update
    fhRAM.Update
    fsCode.Update
    fsRegs.Update
    fsStack.Update
    fsVars.Update
    fdVideo.Update
    If Not Proj.Paused And fiMain.MISpdMax.Checked Then Call
fsCode.RTB.MarksSetVisible(0, False)
  Else
    'Update user interface windows only
    fdVideo.Update
  End If
End Sub

'-----------------------------------'
' Public Sub ResetAll()             '
'                                   '
' Resets the computer, initialising '
' all hardware                      '
'-----------------------------------'
Public Sub ResetAll()
  Proj.TickCount = 0
  Proj.Running = False
  Proj.Paused = False
  Proj.P.CompileNeeded = True
  fhCPU.Reset
  fhRAM.Reset
  devReset
  fsCode.RTB.ReadOnly = False
  Call fsCode.RTB.MarksSetVisible(0, False)
End Sub

'---------------------------------------------------------------------
------'
' Private Sub ToolbarProgram_ButtonClick(ByVal Button As
ComctlLib.Button) '
'---------------------------------------------------------------------
------'
Private Sub ToolbarProgram_ButtonClick(ByVal Button As
ComctlLib.Button)
  'WRITE A PROGRAM'
  If Button.Index = 1 Then fsCode.Show
End Sub

'---------------------------------------------------------------------
--'
' Private Sub ToolbarRun_ButtonClick(ByVal Button As
ComctlLib.Button) '
'---------------------------------------------------------------------
--'
Private Sub ToolbarRun_ButtonClick(ByVal Button As ComctlLib.Button)
  If Button.Index = 1 Then
    'START'
    fsCode.MIStart_Click
  ElseIf Button.Index = 2 Then
    'RESET'
    fsCode.MIReset_Click
  End If
End Sub

'---------------------------------------------------------------------
---'
' Private Sub ToolbarStep_ButtonClick(ByVal Button As
ComctlLib.Button) '
'---------------------------------------------------------------------
---'
Private Sub ToolbarStep_ButtonClick(ByVal Button As ComctlLib.Button)
  If Button.Index = 1 Then
    'TICK'
    fsCode.MITick_ForFIMAIN
  ElseIf Button.Index = 2 Then
    'STEP'
    fsCode.MIStep_Click
  End If
End Sub

'---------------------------------------------------------------------
-----'
' Private Sub ToolbarWnd_ButtonClick(ByVal Button As
ComctlLib.Button) '
'---------------------------------------------------------------------
-----'
Private Sub ToolbarWnd_ButtonClick(ByVal Button As
ComctlLib.Button)
  If Button.Index = 1 Then
    'CPU'
    fhCPU.Show
  ElseIf Button.Index = 2 Then
    'RAM'
    fhRAM.Show
  ElseIf Button.Index = 3 Then
    'Registers'
    fsRegs.Show
  ElseIf Button.Index = 4 Then
    'Variables'
    fsVars.Show
  ElseIf Button.Index = 5 Then
    'Stack'
    fsStack.Show
  End If
End Sub

'---------------------------------------------------------------------
------'
' Private Sub ToolbarWnd2_ButtonClick(ByVal Button As
ComctlLib.Button) '
'---------------------------------------------------------------------
------'
Private Sub ToolbarWnd2_ButtonClick(ByVal Button As
ComctlLib.Button)
  If Button.Index = 1 Then
    'Computer'
    fiComp.Show
  ElseIf Button.Index = 2 Then
    'Display'
    fiDisplay.Show
  ElseIf Button.Index = 3 Then
    'Keyboard'
    fiKeyboard.Show
  ElseIf Button.Index = 4 Then
    'Speaker'
    fdSpeaker.Show
  End If
End Sub

'------------------------------'
' Public Sub SetToolbarButtons() '
'------------------------------'
Public Sub SetToolbarButtons()
  'Set text and image for Start
  If Proj.Running Then
    If Proj.Paused Then
      ToolbarRun.Buttons(1).Caption = "Continue"
      ToolbarRun.Buttons(1).Image = 1
    Else
      ToolbarRun.Buttons(1).Caption = "Pause"
      ToolbarRun.Buttons(1).Image = 2
    End If
  Else
    ToolbarRun.Buttons(1).Caption = "Start"
    ToolbarRun.Buttons(1).Image = 1
  End If
  'Set enabled for Stop
  If Proj.Running Or Proj.Halted Then
    ToolbarRun.Buttons(2).Enabled = True
  Else
    ToolbarRun.Buttons(2).Enabled = False
  End If
  'Set Enabled for step
  ToolbarStep.Buttons(1).Enabled = Not Proj.Running Or Proj.Paused
  ToolbarStep.Buttons(2).Enabled = Not Proj.Running Or Proj.Paused
End Sub
```

22.14. fiComp




```
Option Explicit

'----------------------------------------'
' Public declarations in this module:   '
'    <NONE>                              '
'----------------------------------------'

'--------------------------------------------'
' Private Sub Form_Unload(Cancel As Integer) '
'                                            '
' DESCRIPTION: Event handler for Form_Unload '
'    unloads the form if the application is  '
'    really shutting down, and just hides the '
'    form in case the user requested to close '
'    it.                                     '
'--------------------------------------------'
Private Sub Form_Unload(Cancel As Integer)
  If Not Appp.Terminating Then
    Cancel = 1
    fiComp.Hide
  End If
End Sub

'-------------------------------------------'
' Private Sub PctScr_Paint()                '
'                                           '
' Repaints the display in case it has been  '
' erased by something.                      '
'-------------------------------------------'
Private Sub PctScr_Paint()
  Call StretchBlt(fiComp.PctScr.hdc, 0, 0, fiComp.PctScr.Width,
fiComp.PctScr.Height, _
                  Proj.Video.vDC.hdc, 0, 0, Proj.Video.vDC.Width,
Proj.Video.vDC.Height, SRCCOPY)
End Sub

'-------------------------------------------'
' Private Sub LKey_Click(Index As Integer)  '
'                                           '
' Initiates key event processing mechanism  '
' in the keyboard "controller".             '
'-------------------------------------------'
Private Sub LKey_Click(Index As Integer)
End Sub

'----------------------------------------------'
' Private Sub LKey_MouseDown(Index As Integer, '
'   Button As Integer, Shift As Integer, x As  '
'   Single, y As Single)                       '
'                                              '
' Highlights the key on the keyboard that the  '
' user clicks with the mouse.                  '
'----------------------------------------------'
Private Sub LKey_MouseDown(Index As Integer, Button As Integer,
Shift As Integer, x As Single, y As Single)
  If Index = 27 Then
    LKey(100).BackStyle = 1
    LKey(101).BackStyle = 1
  Else
    LKey(Index).BackStyle = 1
  End If
End Sub

'----------------------------------------------'
' Private Sub LKey_MouseUp(Index As Integer,   '
'   Button As Integer, Shift As Integer, x As  '
'   Single, y As Single)                       '
'                                              '
' Highlights the key on the keyboard that the  '
' user clicks with the mouse.                  '
'----------------------------------------------'
Private Sub LKey_MouseUp(Index As Integer, Button As Integer,
Shift As Integer, x As Single, y As Single)
  If Index = 27 Then
    LKey(100).BackStyle = 0
    LKey(101).BackStyle = 0
  Else
    LKey(Index).BackStyle = 0
  End If
  Call fdKeyboard.KeyDown(Index)
End Sub
```

22.15. fiKeyboard




```
Option Explicit

'---------------------------------------
' Public declarations in this module:  '
'                                      '
' TYPES:                               '
' VARIABLES:                           '
' CONSTANTS:                           '
' PROCEDURES:                          '
'---------------------------------------

'-------------------------------------------
' Private Sub Form_Unload(Cancel As Integer) '
'                                           '
' DESCRIPTION: Event handler for Form_Unload '
'   unloads the form if the application is  '
'   really shutting down, and just hides the '
'   form in case the user requested to close '
'   it.                                      '
'-------------------------------------------
Private Sub Form_Unload(Cancel As Integer)
  If Not Appp.Terminating Then
    Cancel = 1
    fiKeyboard.Hide
  End If
End Sub

'----------------------------------------------
' Private Sub LKey_MouseDown(Index As Integer, '
'   Button As Integer, Shift As Integer, x As  '
'   Single, y As Single)                       '
'                                              '
' Highlights the key on the keyboard that the  '
' user clicks with the mouse.                  '
'----------------------------------------------
Private Sub LKey_MouseDown(Index As Integer, Button As Integer,
Shift As Integer, x As Single, y As Single)
  If Index = 27 Then
    LKey(100).BackStyle = 1
    LKey(101).BackStyle = 1
  Else
    LKey(Index).BackStyle = 1
  End If
End Sub

'----------------------------------------------
' Private Sub LKey_MouseUp(Index As Integer,   '
'   Button As Integer, Shift As Integer, x As  '
'   Single, y As Single)                       '
'                                              '
' Highlights the key on the keyboard that the  '
' user clicks with the mouse.                  '
'----------------------------------------------
Private Sub LKey_MouseUp(Index As Integer, Button As Integer,
Shift As Integer, x As Single, y As Single)
  If Index = 27 Then
    LKey(100).BackStyle = 0
    LKey(101).BackStyle = 0
  Else
    LKey(Index).BackStyle = 0
  End If
  Call fdKeyboard.KeyDown(Index)
End Sub
```

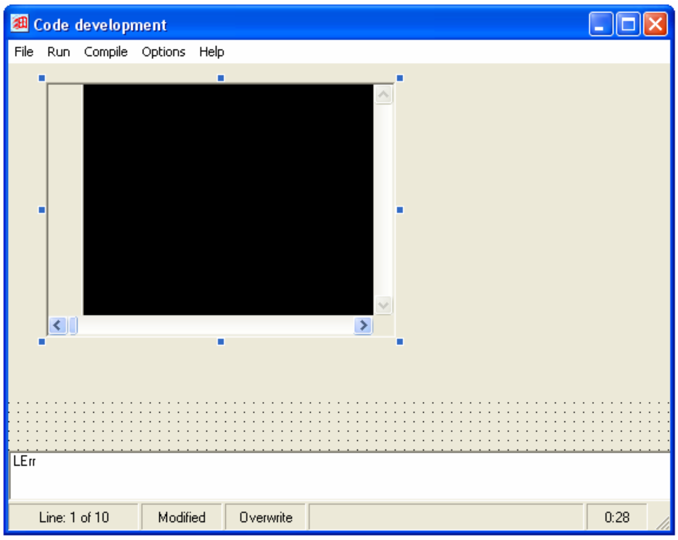
22.16. fiDisplay


```
Option Explicit

'---------------------------------------
' Public declarations in this module:  '
'                                      '
' TYPES:                               '
' VARIABLES:                           '
' CONSTANTS:                           '
' PROCEDURES:                          '
'---------------------------------------

'-------------------------------------------
' Private Sub Form_Unload(Cancel As Integer) '
'                                           '
' DESCRIPTION: Event handler for Form_Unload '
'   unloads the form if the application is  '
'   really shutting down, and just hides the '
'   form in case the user requested to close '
'   it.                                      '
'-------------------------------------------
Private Sub Form_Unload(Cancel As Integer)
  If Not Appp.Terminating Then
    Cancel = 1
    fiDisplay.Hide
  End If
End Sub

'-------------------------------------------
' Private Sub PctScr_Paint()                '
'                                           '
```

```
' Repaints the display in case it has been '
' erased by something.                      '
'-------------------------------------------'
Private Sub PctScr_Paint()
  Call StretchBlt(fiDisplay.PctScr.hdc, 0, 0, fiDisplay.PctScr.Width,
fiDisplay.PctScr.Height, _
                  Proj.Video.vDC.hdc, 0, 0, Proj.Video.vDC.Width,
Proj.Video.vDC.Height, SRCCOPY)
End Sub
```

```
'---------------------------'
' Private Sub Form_Resize() '
'---------------------------'
Private Sub Form_Resize()
  PctScr.Width = ClientW.Width
  PctScr.Height = ClientH.Height
End Sub
```

22.17. fsCode



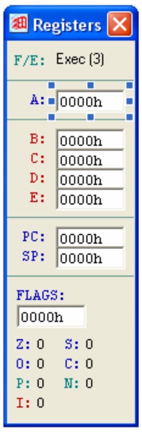

```
Option Explicit

'----------------------------------------'
' Public declarations in this module:    '
'                                        '
' PROCEDURES:                            '
'   Init                                 '
'   Update                               '
'----------------------------------------'

Private errLine As Integer 'highlight an error
Private wngLine As Integer 'highlight a warning
Private Bkpt() As Integer 'breakpoints line numbers

'--------------------------------------------'
' Public Sub Init()                          '
'                                            '
' DESCRIPTION: initialises this module.      '
'--------------------------------------------'
Public Sub Init()
  'RTB highlighting
  errLine = -1
  wngLine = -1
  ReDim Bkpt(-1 To -1)
  'RTB properties
  RTB.MarksXOffset = 0
  RTB.MarksLeftMargin = 2
  Call RTB.SetOptionFlag(eoAutoIndent, True)
  Call RTB.SetOptionFlag(eoSmartTabs, False)
  Call RTB.SetOptionFlag(eoTrimTrailingSpaces, True)
  RTB.TabWidth = 6
  Call RTB.MarksCreateImgList(AppDir + "gutter.bmp", 16, 16)
  'Add one mark for instruction pointer and error pointer
  Call RTB.MarksAdd(0, 0, False)
End Sub

'--------------------------------------------'
' Private Sub Form_Unload(Cancel As Integer) '
'                                            '
' DESCRIPTION: Event handler for Form_Unload '
'   unloads the form if the application is   '
'   really shutting down, and just hides the '
'   form in case the user requested to close '
'   it.                                      '
'--------------------------------------------'
Private Sub Form_Unload(Cancel As Integer)
  If Not Appp.Terminating Then
    Cancel = 1
    fsCode.Hide
  End If
End Sub

'--------------------------------------------'
' Private Sub Form_Resize()                  '
'                                            '
' Resizes all controls on the form in order  '
' to allow for adjustable form size.         '
'--------------------------------------------'
Private Sub Form_Resize()
  LErr.Left = 8
  LErr.Width = PnlErr.Width - 16

  PnlCode.Height = PnlErr.Top
  RTB.Top = 8
  RTB.Height = PnlCode.Height - 16
  RTB.Width = PnlCode.Width - 8 - RTB.Left
End Sub

'-----------------------------'
' Private Sub Form_Activate() '
'-----------------------------'
Private Sub Form_Activate()
  RTB.SetFocus
End Sub

'----------------------------------------'
' Public Sub Update()                    '
'                                        '
' Highlights current execution point.    '
'----------------------------------------'
Public Sub Update()
  InvalidateRTB
End Sub
```

```
'-------------------------------'
' Private Sub InvalidateRTB()   '
'                               '
' Invalidates every line in RTB '
'-------------------------------'
Private Sub InvalidateRTB()
  Dim i As Integer
  RTB.BeginUpdate
  For i = RTB.TopLine To RTB.TopLine + RTB.LinesInWindow
    Call RTB.InvalidateLine(i)
  Next
  RTB.EndUpdate
End Sub

'---------------------------------------'
' Private Sub SetPanelText(s As String) '
'                                       '
' Outputs s in status bar               '
'---------------------------------------'
Private Sub SetPanelText(s As String)
  SB.Panels(4).Text = s
  SB.Refresh
End Sub

'--------------------------------------------'
' Private Sub DisplayError(Index As Integer, '
'   Silent As Boolean)                       '
'                                            '
' Highlights error/warning number Index, and '
' displays a message if Silent is False      '
'--------------------------------------------'
Private Sub DisplayError(Index As Integer, Silent As Boolean)
  'Go to line and highlight it
  RTB.CaretX = 0
  LErr.ListIndex = Index
  If LErr.ListIndex <= UBound(Proj.P.ErrL.sError) Then
    RTB.CaretY = Proj.P.ErrL.lError(Index) + 1
    errLine = RTB.CaretY
    Call RTB.MarksSetImageIndex(0, 1)
    Call RTB.MarksSetLine(0, RTB.CaretY)
    Call RTB.MarksSetVisible(0, True)
    InvalidateRTB
    PnlCode.Refresh
    If Not Silent Then Call MsgBox(Proj.P.ErrL.sError(Index) +
Chr(13) + Chr(10) + Chr(13) + Chr(10) + "Line: " +
CStr(Proj.P.ErrL.lError(Index) + 1) + Chr(13) + Chr(10) + "Code: " +
Proj.P.ErrL.nError(Index), vbOKOnly + vbExclamation, "Compile
program")
  Else
    RTB.CaretY = Proj.P.ErrL.lWarning(Index -
UBound(Proj.P.ErrL.sError) - 1) + 1
    wngLine = RTB.CaretY
    Call RTB.MarksSetImageIndex(0, 1)
    Call RTB.MarksSetLine(0, RTB.CaretY)
    Call RTB.MarksSetVisible(0, True)
    InvalidateRTB
    PnlCode.Refresh
    If Not Silent Then Call MsgBox(Proj.P.ErrL.sWarning(Index -
UBound(Proj.P.ErrL.sError) - 1) + Chr(13) + Chr(10) + Chr(13) +
Chr(10) + "Line: " + CStr(Proj.P.ErrL.lWarning(Index -
UBound(Proj.P.ErrL.sError) - 1) + 1) + Chr(13) + Chr(10) + "Code: " +
Proj.P.ErrL.nWarning(Index - UBound(Proj.P.ErrL.sError) - 1),
vbOKOnly + vbExclamation, "Compile program")
  End If
End Sub

'-------------------------------------'
' Private Sub TransferBreakpoints()   '
'                                     '
' Copies all set breakpoints to the   '
' Proj.CPU structure                  '
'-------------------------------------'
Private Sub TransferBreakpoints()
  'Check if compiled
  If Proj.P.CompileNeeded Then Exit Sub
  'Delete all breakpoints which are out of range now
  Dim i As Integer, A As Integer
  A = 0
  For i = 0 To UBound(Bkpt)
    If Bkpt(i) <= RTB.Lines.Count Then
      Bkpt(A) = Bkpt(i)
      A = A + 1
    End If
  Next
  If UBound(Bkpt) <> (A - 1) Then ReDim Preserve Bkpt(-1 To A - 1)
  'Transfer
  ReDim Proj.CPU.Breakpoint(-1 To UBound(Bkpt))
  For i = 0 To UBound(Bkpt)
    Proj.CPU.Breakpoint(i) = Proj.P.Code_L2O(Bkpt(i)) '+1 for
AFTER the line, -1 for ZERO based
  Next
End Sub

'-------------------------------'
' Private Sub LErr_DblClick()   '
'-------------------------------'
Private Sub LErr_DblClick()
  'Check if anything
  If LErr.ListIndex < 0 Then Exit Sub
  'Show selected warning/error
  Call DisplayError(LErr.ListIndex, True)
End Sub

'------------------------------'
' Private Sub RTB_OnChange()   '
'------------------------------'
Private Sub RTB_OnChange()
  Proj.P.CompileNeeded = True
End Sub

'---------------------------------------------'
' Private Sub RTB_OnKeyPress(Key As Integer)  '
'---------------------------------------------'
Private Sub RTB_OnKeyPress(Key As Integer)
  If RTB.ReadOnly Then Call MsgBox("Please Reset the project
before editing the code.", vbExclamation + vbOKOnly)
End Sub

'-----------------------------------'
' Private Sub RTB_OnStatusChange()  '
'-----------------------------------'
Private Sub RTB_OnStatusChange()
  SB.Panels(1).Text = "Line " + CStr(RTB.CaretY) + " of " +
CStr(RTB.Lines.Count)
  SB.Panels(2).Text = IIf(RTB.Modified, "Modified", "")
  SB.Panels(3).Text = IIf(RTB.InsertMode, "Insert", "Overwrite")
  If errLine <> -1 Or wngLine <> -1 Then
    errLine = -1
    wngLine = -1
    Call RTB.MarksSetVisible(0, False)
    InvalidateRTB
  End If
End Sub

'---------------------------------------------'
' Private Sub RTB_OnSpecialLineColors(ByVal   '
'   Line As Long, Special As Boolean, FG As   '
'   stdole.OLE_COLOR, BG As stdole.OLE_COLOR) '
'---------------------------------------------'
Private Sub RTB_OnSpecialLineColors(ByVal Line As Long, Special As
Boolean, FG As stdole.OLE_COLOR, BG As stdole.OLE_COLOR)
  Dim i As Integer
  'Error
  If errLine = Line Then
    Special = True
    FG = vbWhite
    BG = &HFF
  End If
  'Warning
  If wngLine = Line Then
    Special = True
    FG = vbWhite
    BG = &HFF
  End If
  'Breakpoint
  For i = 0 To UBound(Bkpt)
    If Bkpt(i) = Line Then
      Special = True
      FG = vbWhite
      BG = &H80
    End If
  Next
  'Instruction Pointer
  If Proj.Running And (Proj.Paused Or Not fiMain.MISpdMax.Checked)
Then
    i = Proj.CPU.IP - IIf(Proj.CPU.Fetch And (Proj.CPU.CIB = ""),
0, 1)
    If i <= UBound(Proj.P.Code_O2L) Then
      If Line = Proj.P.Code_O2L(i) + 1 Then
        Special = True
        FG = vbBlack
        BG = &HFFFF00
        Call RTB.MarksSetLine(0, Line)
        Call RTB.MarksSetImageIndex(0, 0)
        Call RTB.MarksSetVisible(0, True)
      End If
```

```
    End If
  End If
End Sub

'---------------------------------------------'
' Private Sub RTB_OnGutterClick(ByVal X As    '
'   Long, ByVal Y As Long, ByVal Line As Long) '
'---------------------------------------------'
Private Sub RTB_OnGutterClick(ByVal x As Long, ByVal y As Long, ByVal
Line As Long)
  'Check if we have a breakpoint for Line
  Dim i As Integer, h As Boolean
  For i = 0 To UBound(Bkpt)
    If Bkpt(i) = Line Then GoTo fnd
  Next
  'Add this line to breakpoints
  ReDim Preserve Bkpt(-1 To UBound(Bkpt) + 1)
  Bkpt(UBound(Bkpt)) = Line
  Call RTB.MarksAdd(Line, 2, True)
  Call RTB.InvalidateLine(Line)
  TransferBreakpoints
  Exit Sub
fnd:
  'Remove this line from breakpoints
  Dim A As Integer
  For A = i + 1 To UBound(Bkpt)
    Bkpt(A - 1) = Bkpt(A)
  Next
  ReDim Preserve Bkpt(-1 To UBound(Bkpt) - 1)
  For i = 0 To RTB.MarksCount - 1
    If RTB.MarksGetLine(i) = Line And RTB.MarksGetImageIndex(i) = 2
Then
      RTB.MarksDelete (i)
      Call RTB.InvalidateLine(Line)
      TransferBreakpoints
      Exit Sub
    End If
  Next
End Sub

'---------------------------------------------'
' Private Function ConfirmCompile(Optional Ask '
'   As Boolean = False) As Boolean            '
'                                             '
' Checks if the program needs to be compiled, '
' and does so if necessary. Warns the user if '
' Ask is set to true                          '
'---------------------------------------------'
Private Function ConfirmCompile(Optional Ask As Boolean = False) As
Boolean
  Dim b As Integer
  'Check if compile needed at all
  If Not Proj.P.CompileNeeded Then
    ConfirmCompile = True
    Exit Function
  End If
  'Confirm
  If Ask Then
    If MsgBox("You have modified the code, and so you have to compile
it again before continuing. Do you want to compile and load your
code?", vbOKCancel + vbQuestion) = vbCancel Then
      ConfirmCompile = False
      Exit Function
    End If
  End If

  'Try to compile
  PrgCompile
  'Check error messages
  If UBound(Proj.P.ErrL.sError) = -1 Then
    'No errors
    PrgLoad
  Else
    'Go to line and highlight it
    Call DisplayError(0, False)
    ConfirmCompile = False
    Exit Function
  End If
  'Continue
  ConfirmCompile = True
End Function

'----------------------------'
' Private Sub MIImport_Click() '
'----------------------------'
Private Sub MIImport_Click()
  'Get file name
  Dim s As String: s = ""
  If Not GetFilename(False, s, "", "Assembly language
files|*.asm|All files|*.*", "asm", "Import assembly language
program") Then Exit Sub
  'Load as RTF if ext is rtf, text otherwise
  SetPanelText "Loading file... Please wait"
  Open s For Binary As 1
  s = String(LOF(1), " ")
  Get 1, , s
  RTB.Text = s
  Close
  SetPanelText ""
  'Need to compile before running
  Proj.P.CompileNeeded = True
End Sub

'----------------------------'
' Private Sub MIExport_Click() '
'----------------------------'
Private Sub MIExport_Click()
  'Get file name
  Dim s As String: s = ""
  If Not GetFilename(True, s, "", "Assembly language
files|*.asm|All files|*.*", "asm", "Export assembly language
program") Then Exit Sub
  'Save file
  Open s For Binary As 1
  Put 1, , RTB.Text
  Close
End Sub

'-------------------------'
' Private Sub MIRun_Click() '
'-------------------------'
Private Sub MIRun_Click()
  'Set submenu according to current state
  If Proj.Running Then
    If Proj.Paused Then
      MIStart.Caption = "Continue"
    Else
      MIStart.Caption = "Pause"
    End If
  Else
    MIStart.Caption = "Start"
  End If
  MIStep.Enabled = Not Proj.Running Or Proj.Paused
End Sub

'---------------------------'
' Private Sub MIStart_Click() '
'---------------------------'
Public Sub MIStart_Click()
  If Proj.Halted Then
    Proj.Running = False
    Proj.Paused = False
    Proj.Halted = False
  End If
  If Proj.Running Then
    If Not ConfirmCompile Then Exit Sub
    Proj.Paused = Not Proj.Paused
    If Not Proj.Paused And fiMain.MISpdMax.Checked Then Call
RTB.MarksSetVisible(0, False)
  Else
    fiMain.ResetAll
    If Not ConfirmCompile Then Exit Sub
    TransferBreakpoints
    Proj.Running = True
    Proj.Paused = False
  End If
  RTB.ReadOnly = True
  fsCode.SetFocus
  fiMain.SetToolbarButtons
End Sub

'--------------------------'
' Private Sub MIStep_Click() '
'--------------------------'
Public Sub MIStep_Click()
  If Proj.Halted Then
    Proj.Running = False
    Proj.Paused = False
    Proj.Halted = False
  End If
  If Proj.Running Then
    If Not ConfirmCompile Then Exit Sub
    Step
  Else
    fiMain.ResetAll
    If Not ConfirmCompile Then Exit Sub
```

```
    TransferBreakpoints
    Proj.Running = True
    Proj.Paused = True
    Step
    RTB.ReadOnly = True
  End If
  fiMain.UpdateAll
  fsCode.SetFocus
  fiMain.SetToolbarButtons
End Sub

'-----------------------------'
' Private Sub MIReset_Click() '
'-----------------------------'
Public Sub MIReset_Click()
  fiMain.ResetAll
  fiMain.UpdateAll
  fsCode.SetFocus
  fiMain.SetToolbarButtons
End Sub

'----------------------------------'
' Private Sub MIBreakpoint_Click() '
'----------------------------------'
Private Sub MIBreakpoint_Click()
  Call RTB_OnGutterClick(0, 0, RTB.CaretY)
End Sub

'------------------------------'
' Private Sub MISyntax_Click() '
'------------------------------'
Private Sub MISyntax_Click()
  'Try to compile
  PrgCompile
  'Check error messages
  If UBound(Proj.P.ErrL.sError) = -1 Then
    'No errors
    Call MsgBox("No errors were found in your code.", vbOKOnly +
vbInformation, "Syntax check")
  Else
    'Go to line and highlight it
    Call DisplayError(0, False)
  End If
End Sub

'---------------------------------'
' Private Sub MIDoCompile_Click() '
'---------------------------------'
Private Sub MIDoCompile_Click()
  'Try to compile
  PrgCompile
  'Check error messages
  If UBound(Proj.P.ErrL.sError) = -1 Then
    'No errors
    PrgLoad
    Call MsgBox("Program compilation successful.", vbOKOnly +
vbInformation, "Compile program")
  Else
    Call DisplayError(0, False)
  End If
End Sub

'--------------------------------'
' Private Sub MINextErr_Click() '
'--------------------------------'
Private Sub MINextErr_Click()
  If LErr.ListCount = 0 Then Exit Sub
  If LErr.ListIndex >= LErr.ListCount - 1 Then
    LErr.ListIndex = 0
  Else
    LErr.ListIndex = LErr.ListIndex + 1
  End If
  LErr_DblClick
End Sub

'--------------------------------'
' Public Sub MITick_ForFIMAIN() '
'--------------------------------'
Public Sub MITick_ForFIMAIN()
  'fiMain form needs to invoke Tick, but it has no access to
  'some important functions which are private fsCode functions.
  If Proj.Halted Then
    Proj.Running = False
    Proj.Paused = False
    Proj.Halted = False
  End If
  If Proj.Running Then
    If Not ConfirmCompile Then Exit Sub
    Tick
  Else
    fiMain.ResetAll
    If Not ConfirmCompile Then Exit Sub
    TransferBreakpoints
    Proj.Running = True
    Proj.Paused = True
    Tick
    RTB.ReadOnly = True
  End If
  fiMain.UpdateAll
  fsCode.SetFocus
  fiMain.SetToolbarButtons
End Sub
```

22.18. fsRegs

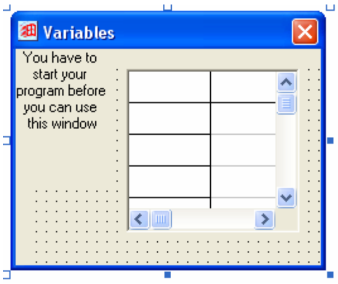



```
Option Explicit

'-----------------------------------------'
' Public declarations in this module:     '
'                                          '
' PROCEDURES:                             '
'   Init                                   '
'   SetNumberFormat                        '
'   Update                                 '
'-----------------------------------------'

'Stores the string values for all registers displayed
```

```
' on this form to track changes and highlight respectively
Private LastStr(0 To 15) As String

'-------------------------------------------'
' Public Sub Init()                         '
'                                           '
' DESCRIPTION: initialises this module.     '
'                                           '
' NOTES: Must be called *after* fhCPU.Init  '
'-------------------------------------------'
Public Sub Init()
  'Set initial values and colors
  Update
  SaveLast
  Update
End Sub

'-------------------------------------------'
' Private Sub Form_Unload(Cancel As Integer) '
'                                           '
' DESCRIPTION: Event handler for Form_Unload '
'   unloads the form if the application is '
'   really shutting down, and just hides the '
'   form in case the user requested to close '
'   it.                                     '
'-------------------------------------------'
Private Sub Form_Unload(Cancel As Integer)
  If Not Appp.Terminating Then
    Cancel = 1
    fsRegs.Hide
  End If
End Sub

'-----------------------------------------'
' Public Sub Update()                     '
'                                         '
' Updates the contents of the window to   '
' reflect changes to the state of the     '
' simulation.                             '
'-----------------------------------------'
Public Sub Update()
  'Mode
  NRun.Visible = Not Proj.Running
  NCtl.Visible = Proj.Running
  Dim s As String
  'Values
  s = IIf(Proj.CPU.Fetch, "Fetch (", "Exec (")
  If Proj.CPU.Fetch Then
    If Proj.CPU.CIB = "" Then
      s = s + "?"
    Else
      s = s + IIf(Proj.CPU.fremMem, CStr(Proj.CPU.FREM + 1) + "+",
CStr(Proj.CPU.FREM))
    End If
  Else
    s = s + CStr(UBound(Proj.CPU.DIB) - Proj.CPU.eDP + 1)
  End If
  LFE.Caption = s + ")"
  LA.Text = Dec2Fmt16(Proj.CPU.A, Proj.NmbRep)
  LB.Text = Dec2Fmt16(Proj.CPU.R(0), Proj.NmbRep)
  LC.Text = Dec2Fmt16(Proj.CPU.R(1), Proj.NmbRep)
  LD.Text = Dec2Fmt16(Proj.CPU.R(2), Proj.NmbRep)
  LE.Text = Dec2Fmt16(Proj.CPU.R(3), Proj.NmbRep)
  LIP.Text = Dec2Fmt16(Proj.CPU.IP, Proj.NmbRep)
  LSP.Text = Dec2Fmt16(Proj.CPU.SP, Proj.NmbRep)
  LFLAGS.Text = Dec2Fmt16(Proj.CPU.FLAGS, Proj.NmbRep)
  LFZ.Caption = IIf((Proj.CPU.FLAGS And 1) > 0, "1", "0")
  LFS.Caption = IIf((Proj.CPU.FLAGS And 2) > 0, "1", "0")
  LFO.Caption = IIf((Proj.CPU.FLAGS And 4) > 0, "1", "0")
  LFC.Caption = IIf((Proj.CPU.FLAGS And 8) > 0, "1", "0")
  LFI.Caption = IIf((Proj.CPU.FLAGS And 16) > 0, "1", "0")
  LFN.Caption = IIf((Proj.CPU.FLAGS And 256) > 0, "1", "0")
  LFP.Caption = IIf((Proj.CPU.FLAGS And 512) > 0, "1", "0")

  'Colors
  LFE.ForeColor = IIf(LastStr(0) = LFE.Caption, 0, &HFF)
  LA.ForeColor = IIf(LastStr(1) = LA.Text, 0, &HFF)
  LB.ForeColor = IIf(LastStr(2) = LB.Text, 0, &HFF)
  LC.ForeColor = IIf(LastStr(3) = LC.Text, 0, &HFF)
  LD.ForeColor = IIf(LastStr(4) = LD.Text, 0, &HFF)
  LE.ForeColor = IIf(LastStr(5) = LE.Text, 0, &HFF)
  LIP.ForeColor = IIf(LastStr(6) = LIP.Text, 0, &HFF)
  LSP.ForeColor = IIf(LastStr(7) = LSP.Text, 0, &HFF)
  LFLAGS.ForeColor = IIf(LastStr(8) = LFLAGS.Text, 0, &HFF)
  LFZ.ForeColor = IIf(LastStr(9) = LFZ.Caption, 0, &HFF)
  LFS.ForeColor = IIf(LastStr(10) = LFS.Caption, 0, &HFF)
  LFO.ForeColor = IIf(LastStr(11) = LFO.Caption, 0, &HFF)
  LFC.ForeColor = IIf(LastStr(12) = LFC.Caption, 0, &HFF)
  LFP.ForeColor = IIf(LastStr(13) = LFP.Caption, 0, &HFF)
  LFN.ForeColor = IIf(LastStr(14) = LFN.Caption, 0, &HFF)
  LFI.ForeColor = IIf(LastStr(15) = LFI.Caption, 0, &HFF)

  'Remember last status
  SaveLast
End Sub

'-------------------------------------------'
' Private Sub SaveLast()                    '
'                                           '
' Saves the state of all register in order to '
' highlight them as they change. Is called by '
' Update() after getting new values for them '
'-------------------------------------------'
Private Sub SaveLast()
  LastStr(0) = LFE.Caption
  LastStr(1) = LA.Text
  LastStr(2) = LB.Text
  LastStr(3) = LC.Text
  LastStr(4) = LD.Text
  LastStr(5) = LE.Text
  LastStr(6) = LIP.Text
  LastStr(7) = LSP.Text
  LastStr(8) = LFLAGS.Text
  LastStr(9) = LFZ.Caption
  LastStr(10) = LFS.Caption
  LastStr(11) = LFO.Caption
  LastStr(12) = LFC.Caption
  LastStr(13) = LFP.Caption
  LastStr(14) = LFN.Caption
  LastStr(15) = LFI.Caption
End Sub

Private Sub LA_KeyPress(KeyAscii As Integer)
  If KeyAscii = 13 Then
    If IsFmt16(LA.Text) Then
      Proj.CPU.A = Fmt2Dec16(LA.Text)
      Call Update
    Else
      Call MsgBox("Please enter a number between -32768 and 65535,
in decimal, hexadecimal or binary. Hexadecimal numbers must be
followed by letter 'h', binary numbers - by 'b'.", vbExclamation +
vbOKOnly)
    End If
  End If
End Sub

Private Sub LB_KeyPress(KeyAscii As Integer)
  If KeyAscii = 13 Then
    If IsFmt16(LB.Text) Then
      Proj.CPU.R(0) = Fmt2Dec16(LB.Text)
      Call Update
    Else
      Call MsgBox("Please enter a number between -32768 and 65535,
in decimal, hexadecimal or binary. Hexadecimal numbers must be
followed by letter 'h', binary numbers - by 'b'.", vbExclamation +
vbOKOnly)
    End If
  End If
End Sub

Private Sub LC_KeyPress(KeyAscii As Integer)
  If KeyAscii = 13 Then
    If IsFmt16(LC.Text) Then
      Proj.CPU.R(1) = Fmt2Dec16(LC.Text)
      Call Update
    Else
      Call MsgBox("Please enter a number between -32768 and 65535,
in decimal, hexadecimal or binary. Hexadecimal numbers must be
followed by letter 'h', binary numbers - by 'b'.", vbExclamation +
vbOKOnly)
    End If
  End If
End Sub

Private Sub LD_KeyPress(KeyAscii As Integer)
  If KeyAscii = 13 Then
    If IsFmt16(LD.Text) Then
      Proj.CPU.R(2) = Fmt2Dec16(LD.Text)
      Call Update
    Else
      Call MsgBox("Please enter a number between -32768 and 65535,
in decimal, hexadecimal or binary. Hexadecimal numbers must be
followed by letter 'h', binary numbers - by 'b'.", vbExclamation +
vbOKOnly)
    End If
  End If
```

```
End Sub

Private Sub LE_KeyPress(KeyAscii As Integer)
  If KeyAscii = 13 Then
    If IsFmt16(LE.Text) Then
      Proj.CPU.R(3) = Fmt2Dec16(LE.Text)
      Call Update
    Else
      Call MsgBox("Please enter a number between -32768 and 65535, in
decimal, hexadecimal or binary. Hexadecimal numbers must be followed
by letter 'h', binary numbers - by 'b'.", vbExclamation + vbOKOnly)
    End If
  End If
End Sub

Private Sub LSP_KeyPress(KeyAscii As Integer)
  If KeyAscii = 13 Then
    If IsFmt16(LSP.Text) Then
      Proj.CPU.SP = Fmt2Dec16(LSP.Text)
      Call Update
    Else
      Call MsgBox("Please enter a number between -32768 and 65535, in
decimal, hexadecimal or binary. Hexadecimal numbers must be followed
by letter 'h', binary numbers - by 'b'.", vbExclamation + vbOKOnly)
    End If
  End If
End Sub

Private Sub LIP_KeyPress(KeyAscii As Integer)
  If KeyAscii = 13 Then
    If IsFmt16(LIP.Text) Then
      Proj.CPU.IP = Fmt2Dec16(LIP.Text)
      Call fiMain.UpdateAll
    Else
      Call MsgBox("Please enter a number between -32768 and 65535,
in decimal, hexadecimal or binary. Hexadecimal numbers must be
followed by letter 'h', binary numbers - by 'b'.", vbExclamation +
vbOKOnly)
    End If
  End If
End Sub

Private Sub LFLAGS_KeyPress(KeyAscii As Integer)
  If KeyAscii = 13 Then
    If IsFmt16(LFLAGS.Text) Then
      Proj.CPU.FLAGS = Fmt2Dec16(LFLAGS.Text)
    Else
      Call MsgBox("Please enter a number between -32768 and 65535,
in decimal, hexadecimal or binary. Hexadecimal numbers must be
followed by letter 'h', binary numbers - by 'b'.", vbExclamation +
vbOKOnly)
    End If
  End If
End Sub
```

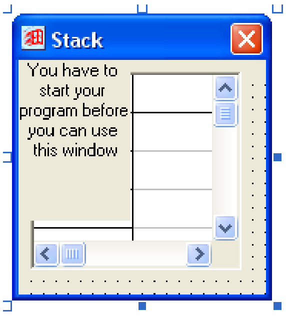
22.19. fsVars




```
Option Explicit

'Font to draw string grid
Private drwFont As Long
'Brush for selected variable
Private brushSel As Long

'Prevent specific events
Private BlockSelectCell As Boolean
Private BlockSetEditCell As Boolean

'-------------------------------------------'
' Public Sub Init()                         '
'                                           '
' DESCRIPTION: initialises this module.     '
'-------------------------------------------'
Public Sub Init()
  'SG font
  drwFont = CreateFont(-11, 0, 0, 0, 400, False, False, False, 1, 0,
0, 0, 0, "Courier New")
  Dim LB As LOGBRUSH
  LB.lbColor = GetSysColor(COLOR_HIGHLIGHT): LB.lbHatch = 0:
LB.lbStyle = 0
  brushSel = CreateBrushIndirect(LB)
  'Set SG options
  SG.Option(goEditing) = True
  SG.Option(goColMoving) = False
  SG.Option(goColSizing) = True
  SG.Option(goRangeSelect) = False
  SG.Option(goRowMoving) = False
  SG.Option(goRowSizing) = False
  SG.Option(goThumbTracking) = True
  'SG visual
  SG.ColCount = 2
  SG.RowCount = 20
  SG.FixedCols = 0
  SG.FixedRows = 1
  SG.Cells(0, 0) = "Variable"
  SG.Cells(1, 0) = "Value"
  SG.ColWidths(0) = 100
  SG.ColWidths(1) = 70

  BlockSelectCell = False
  BlockSetEditCell = False

  Update
End Sub

'-------------------------------------------'
' Private Sub Form_Unload(Cancel As Integer) '
'                                           '
' DESCRIPTION: Event handler for Form_Unload '
'   unloads the form if the application is  '
'   really shutting down, and just hides the '
'   form in case the user requested to close '
'   it.                                     '
'-------------------------------------------'
Private Sub Form_Unload(Cancel As Integer)
  If Not Appp.Terminating Then
    Cancel = 1
    fsVars.Hide
  End If
End Sub

'--------------------------'
' Private Sub Form_Resize() '
'--------------------------'
Private Sub Form_Resize()
  SG.Width = ClientW.Width - 16
  SG.Height = ClientH.Height - 16
  NRun.Width = ClientW.Width
  NRun.Height = ClientH.Height
End Sub
```

```
'---------------------------------------'
' Public Sub Update()                   '
'                                       '
' Updates the contents of the window to '
' reflect changes to the state of the   '
' simulation.                           '
'---------------------------------------'
Public Sub Update()
  Dim rct As RECT
  Call GetClientRect(SG.hwnd, rct)
  Call InvalidateRect(SG.hwnd, rct, True)
  SG.RowCount = UBound(Proj.P.Vars) + 2 + IIf(UBound(Proj.P.Vars) = -1, 1, 0)
  SG.Visible = Proj.Running
  NRun.Visible = Not Proj.Running
End Sub

'-----------------------------------------'
' Private Sub SG_OnDrawCell(ByVal ACol As '
'   Long, ByVal ARow As Long, ByVal RectFX '
'   As Long, ByVal RectFY As Long, ByVal  '
'   RectTX As Long, ByVal RectTY As Long) '
'-----------------------------------------'
Private Sub SG_OnDrawCell(ByVal ACol As Long, ByVal ARow As Long, ByVal RectFX As Long, ByVal RectFY As Long, ByVal RectTX As Long, ByVal RectTY As Long)
  Dim rct As RECT
  rct.Left = RectFX + 2
  rct.Top = RectFY + 1
  rct.Right = RectTX - 2
  rct.Bottom = RectTY - 1
  Dim st As String
  If ARow = 0 Then
    Call SelectObject(SG.hdc, GetStockObject(1)) 'LTGRAY_BRUSH
    st = SG.Cells(ACol, ARow)
  Else
    If SG.GetSelY = ARow Then
      Call SelectObject(SG.hdc, brushSel)
      Call SetTextColor(SG.hdc, GetSysColor(COLOR_HIGHLIGHTTEXT))
    Else
      Call SelectObject(SG.hdc, GetStockObject(0)) 'WHITE_BRUSH
      Call SetTextColor(SG.hdc, 0)
    End If
    st = ""
    If ACol = 0 Then
      If UBound(Proj.P.Vars) <> -1 Then st = Proj.P.Vars(ARow - 1).Name
    Else
      If UBound(Proj.P.Vars) <> -1 Then st = Dec2Fmt16(Proj.RAM(Proj.P.Vars(ARow - 1).Addr) * CLng(256) + Proj.RAM(Proj.P.Vars(ARow - 1).Addr + 1), Proj.NmbRep)
    End If
  End If
  Call SelectObject(SG.hdc, GetStockObject(7)) 'BLACK_PEN
  Call Rectangle(SG.hdc, RectFX - 1, RectFY - 1, RectTX + 1, RectTY + 1)
  Call SelectObject(SG.hdc, drwFont)
  Call SetBkMode(SG.hdc, TRANSPARENT)
  Call DrawText(SG.hdc, st, Len(st), rct, 0)
End Sub

'-------------------------------------------'
' Private Sub SG_OnGetEditText(ByVal ACol As '
'   Long, ByVal ARow As Long, Value As String) '
'-------------------------------------------'
Private Sub SG_OnGetEditText(ByVal ACol As Long, ByVal ARow As Long, Value As String)
  Value = ""
  If ACol <> 1 Then Exit Sub
  If ARow = 0 Then Exit Sub
  If UBound(Proj.P.Vars) = -1 Then Exit Sub
  Value = Dec2Fmt16(Proj.RAM(Proj.P.Vars(ARow - 1).Addr) * CLng(256) + Proj.RAM(Proj.P.Vars(ARow - 1).Addr + 1), Proj.NmbRep)
End Sub

'----------------------------------------'
' Private Sub SG_OnSetEditText(ByVal ACol '
'   As Long, ByVal ARow As Long, ByVal   '
'   Value As String)                     '
'----------------------------------------'
Private Sub SG_OnSetEditText(ByVal ACol As Long, ByVal ARow As Long, ByVal Value As String)
  'Tests
  If BlockSetEditCell Then Exit Sub
  If SG.EditorMode Then Exit Sub
  BlockSetEditCell = True
  If ACol <> 1 Then Call MsgBox("You cannot edit this cell.", vbOKOnly + vbInformation): BlockSetEditCell = False: Exit Sub
  If ARow = 0 Then Call MsgBox("You cannot edit this cell.", vbOKOnly + vbInformation): BlockSetEditCell = False: Exit Sub
  If UBound(Proj.P.Vars) = -1 Then Call MsgBox("There are no variables declared in your source code." + Chr(13) + Chr(10) + "Use structures like 'myvar: DW 10' to declare a variable MYVAR equal to 10.", vbOKOnly + vbInformation): BlockSetEditCell = False: Exit Sub
  'Valid value?
  If Not IsFmt16(Value) Then
    Call MsgBox("The number you entered is not valid. The number has to be between -32768 and 65535 (-8000h and FFFFh), and end with nothing for denary numbers, H for hexadecimal and B for binary.", vbOKOnly + vbInformation)
    BlockSetEditCell = False
    Exit Sub
  End If
  'Convert to string and write to memory
  Dim s As String
  s = Dec2Chr(Fmt2Dec16(Value), 2)
  Proj.RAM(Proj.P.Vars(ARow - 1).Addr) = Asc(Mid(s, 1, 1))
  Proj.RAM(Proj.P.Vars(ARow - 1).Addr + 1) = Asc(Mid(s, 2, 1))
  'Finished
  BlockSetEditCell = False
End Sub

'-----------------------------------------------'
' Private Sub SG_OnSelectCell(ByVal ACol As Long, '
'   ByVal ARow As Long, CanSelect As Boolean)   '
'-----------------------------------------------'
Private Sub SG_OnSelectCell(ByVal ACol As Long, ByVal ARow As Long, CanSelect As Boolean)
  If BlockSelectCell Then Exit Sub
  Dim rct As RECT
  Call GetClientRect(SG.hwnd, rct)
  Call InvalidateRect(SG.hwnd, rct, True)
  If ACol = 0 Then
    BlockSelectCell = True
    SG.SetSelX 1
    SG.SetSelY ARow
    BlockSelectCell = False
    CanSelect = False
  End If
End Sub

'------------------------------------------'
' Private Sub SG_OnMouseDown(ByVal MouseButton '
'   As StringGridVBProj.TxMouseButton)     '
'------------------------------------------'
Private Sub SG_OnMouseDown(ByVal MouseButton As StringGridVBProj.TxMouseButton)
  If MouseButton = mbRight Then PopupMenu MIOptions
End Sub
```

22.20. fsStack




```
Option Explicit

'Stack base address
Private stBase As Long
'Stack length
Private stLen As Long

'Font to draw string grid
Private drwFont As Long
'Brush to highlight SP
Private drwSPBrush As Long

'-----------------------------------------'
' Public Sub Init()                       '
'                                         '
' DESCRIPTION: initialises this module.   '
'-----------------------------------------'
Public Sub Init()
  'SG font
  drwFont = CreateFont(-11, 0, 0, 0, 400, False, False, False, 1, 0,
0, 0, 0, "Courier New")
  Dim LB As LOGBRUSH
  LB.lbColor = &HC0C0FF: LB.lbHatch = 0: LB.lbStyle = 0
  drwSPBrush = CreateBrushIndirect(LB)
  'SG visual
  SG.FixedCols = 0
  SG.FixedRows = 1
  SG.ColCount = 2
  SG.RowCount = 2
  SG.Cells(0, 0) = "Addr"
  SG.Cells(1, 0) = "Value"
  'Column widths
  'I don't know whether this is a VB bug, a Windows bug
  'or just me, but if we don't call SelectObject before
  'every call using SG.hdc then we lose the font.
  Dim sz As Size
  Call SelectObject(SG.hdc, drwFont)
  Call GetTextExtentPoint32(SG.hdc, SG.Cells(0, 0), Len(SG.Cells(0,
0)), sz)
  SG.ColWidths(0) = sz.cx + 4 + 10
  Call SelectObject(SG.hdc, drwFont)
  Call GetTextExtentPoint32(SG.hdc, SG.Cells(1, 0), Len(SG.Cells(1,
0)), sz)
  SG.ColWidths(1) = sz.cx + 4 + 10
  SG.DefaultRowHeight = sz.cy + 2
  'Set SG options
  SG.Option(goEditing) = False
  SG.Option(goColMoving) = False
  SG.Option(goColSizing) = True
  SG.Option(goRangeSelect) = False
  SG.Option(goRowMoving) = False
  SG.Option(goRowSizing) = False
  SG.Option(goThumbTracking) = True
  'Set stack parameters
  stBase = 24576
  stLen = 2048
  SG.RowCount = stLen \ 2 + 1
End Sub

'-----------------------------------------'
' Private Sub Form_Unload(Cancel As Integer) '
'                                         '
' DESCRIPTION: Event handler for Form_Unload '
'   unloads the form if the application is '
'   really shutting down, and just hides the '
'   form in case the user requested to close '
'   it.                                   '
'-----------------------------------------'
Private Sub Form_Unload(Cancel As Integer)
  If Not Appp.Terminating Then
    Cancel = 1
    fsStack.Hide
  End If
End Sub

'---------------------------'
' Private Sub Form_Resize() '
'---------------------------'
Private Sub Form_Resize()
  SG.Width = ClientW.Width - 240
  SG.Height = ClientH.Height - 240
  NRun.Width = ClientW.Width
  NRun.Height = ClientH.Height
End Sub

'-----------------------------------------'
' Public Sub Update()                     '
'                                         '
' Updates the contents of the window to   '
' reflect changes to the state of the     '
' simulation.                             '
'-----------------------------------------'
Public Sub Update()
  Dim rct As RECT
  Call GetClientRect(SG.hwnd, rct)
  Call InvalidateRect(SG.hwnd, rct, True)
  SG.Visible = Proj.Running
  NRun.Visible = Not Proj.Running
End Sub

'-----------------------------------------'
' Private Sub SG_OnDrawCell(ByVal ACol As  '
'   Long, ByVal ARow As Long, ByVal RectFX '
'   As Long, ByVal RectFY As Long, ByVal   '
'   RectTX As Long, ByVal RectTY As Long)  '
'-----------------------------------------'
Private Sub SG_OnDrawCell(ByVal ACol As Long, ByVal ARow As Long,
ByVal RectFX As Long, ByVal RectFY As Long, ByVal RectTX As Long,
ByVal RectTY As Long)
  Dim rct As RECT
  rct.Left = RectFX + 2
  rct.Top = RectFY + 1
  rct.Right = RectTX - 2
  rct.Bottom = RectTY - 1
  Dim st As String
  If ARow = 0 Then
    Call SelectObject(SG.hdc, GetStockObject(1)) 'LTGRAY_BRUSH
    st = SG.Cells(ACol, ARow)
  Else
    If Proj.CPU.SP = stBase + (ARow - 1) * 2 Then
      Call SelectObject(SG.hdc, drwSPBrush)
    Else
      Call SelectObject(SG.hdc, GetStockObject(0)) 'WHITE_BRUSH
    End If
    If ACol = 0 Then
      st = Dec2Hex(stBase + (ARow - 1) * 2, 4)
    Else
      st = Dec2Fmt16(Proj.RAM(stBase + (ARow - 1) * 2) * 256 +
Proj.RAM(stBase + (ARow - 1) * 2 + 1), Proj.NmbRep)
    End If
  End If
  Call SelectObject(SG.hdc, GetStockObject(7)) 'BLACK_PEN
  Call Rectangle(SG.hdc, RectFX - 1, RectFY - 1, RectTX + 1,
RectTY + 1)
  Call SelectObject(SG.hdc, drwFont)
  Call SetBkMode(SG.hdc, TRANSPARENT)
  Call DrawText(SG.hdc, st, Len(st), rct, 0)
End Sub

'-----------------------------------------'
' Private Sub SG_OnMouseDown(ByVal MouseButton '
'   As StringGridVBProj.TxMouseButton)     '
```

```
'-------------------------------------------'
Private Sub SG_OnMouseDown(ByVal MouseButton As
StringGridVBProj.TxMouseButton)
  If MouseButton = mbRight Then PopupMenu MIOptions
End Sub
```

Testing

23. Assembly language testing

In this section I will first test different types of operands using the `ld` instruction. I will then test the most frequently used opcodes.

Note that this section provides no visual proof that tests were passed – mainly because there are too many tests, and supporting each one with a picture would be very excessive. I will support my tests in the other two sections, which will at the same time partly support these tests too.

23.1. Operand testing

No.	Instruction	Expected result	Passed
1	ld a, 10	10 loaded into accumulator	OK
2	ld a, 10b	2 loaded into accumulator	OK
3	ld a, 0FFFFh	65535 loaded into accumulator	OK
4	ld a, -8000h	-32768 loaded into accumulator	OK
5	ld a, 66000	Error message – overflow in 16-bit constant	OK
6	ld a, 0FF0	Error message – incorrect syntax	OK
7	ld b, 0F00h	3840 loaded into B register	OK
8	ld a, b	Contents of B copied into accumulator	OK
9	ld a, myvar	Contents of `myvar` loaded into accumulator	OK
10	ld myvar, a	Accumulator stored in `myvar`	OK
11	ld myvar, 10	Error message – cannot load a constant into a variable	OK
12	ld c, myvar ;myvar is not declared	Error message – `myvar` is not declared	OK

23.2. Opcode testing

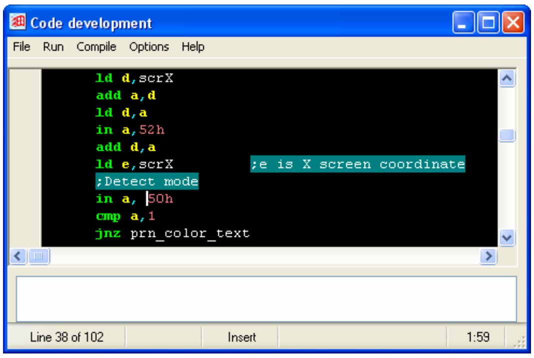
No.	Instruction	Expected result	Passed
13	add a, 10	10 added to accumulator; result stored in accumulator	OK
14	sub b, a	A subtracted from B; result stored in B	OK
15	cmp a, b ; a = b	Zero flag set to true	OK
16	mul a, b	A multiplied by B; result stored in A	OK
17	mul b, c	Error message – incompatible operands	OK
18	div a, myvar	A divided by myvar; result stored in A	OK
19	neg c	Sign of C changed	OK
20	not c	All bits in C changed	OK
21	and a, myvar	Bitwise And performed on A and myvar; result in A	OK
22	or b, a	Bitwise Or on B and A; result stored in B	OK
23	xor b, a	Bitwise Xor performed on B and A; result stored in B	OK
24	xor a,a	Accumulator set to zero	OK
25	lshr b, 5	Bits in B shifted by 5 bits to the right	OK
26	lshl a, 3	Bits in A shifted by 3 bits to the left	OK
27	ashr a, 5	Bits in A shifted by 5 bits to the right; sign preserved	OK
28	ashl b, a	Bits in B shifted by A bits to the right; sign preserved	OK
29	ashl b, 20	Error message – number of shifts exceeds 15	OK
30	jg lbl ; after comparing 2 and 5	Jump not performed	OK
31	jg lbl ; after comparing 5 and 2	Jump performed	OK
32	jl lbl ; after comparing 2 and 5	Jump performed	OK
33	jmp mylabel	Jump to mylabel	OK

34	call myproc	Myproc called; previous address pushed on stack	OK
35	ret ; after a call	Jump to the instruction following the call opcode	OK
36	halt	Program execution stopped	OK

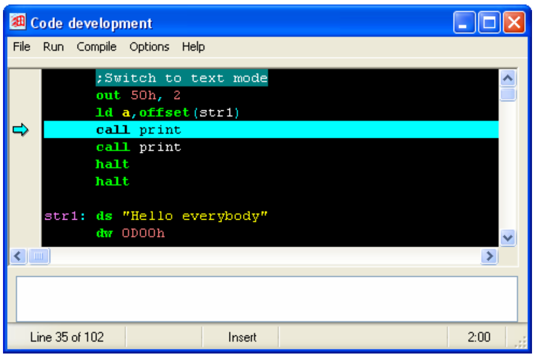
24. Window testing

24.1. Code window

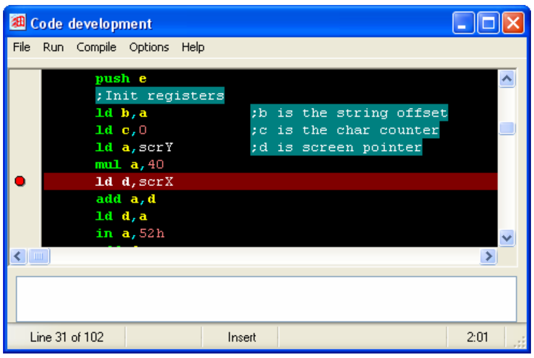
No.	Test	Expected result	Passed	Reference
37	Write a program	Syntax highlighting; editing facilities	OK	24.1.1
38	Step through a program	Current line highlighted	OK	24.1.2
39	Set a breakpoint	Breakpoint line highlighted; execution should pause	OK	24.1.3
40	Run a program with many errors	Error listed at the bottom; The first error displayed in an error message.	OK	24.1.4



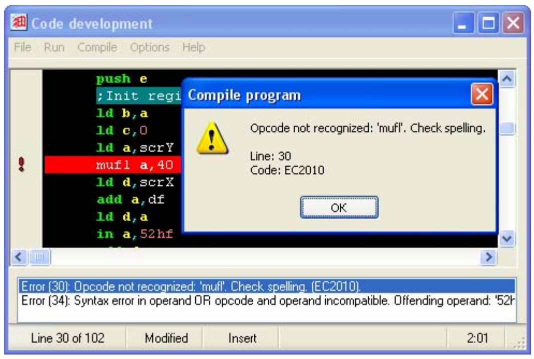
24.1.1



24.1.2



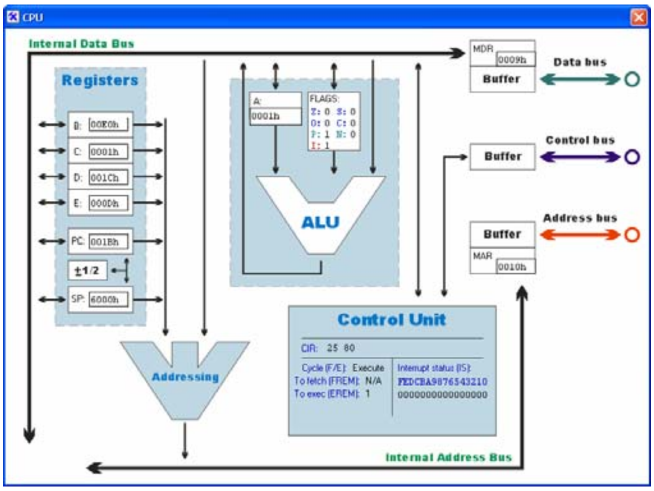
24.1.3



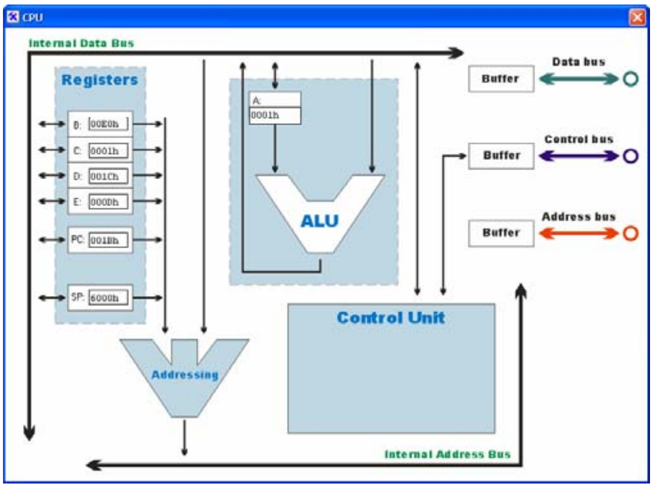
24.1.4

24.2. CPU window

No.	Test	Expected result	Passed	Reference
41	Run a program	Register contents should be displayed, including Current Instruction Register and MAR/MDR	OK	24.2.1
42	Switch complexity mode	Window layout should change – to a simple one in this case	OK	24.2.2



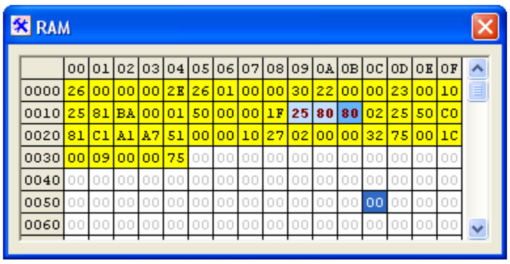
24.2.1



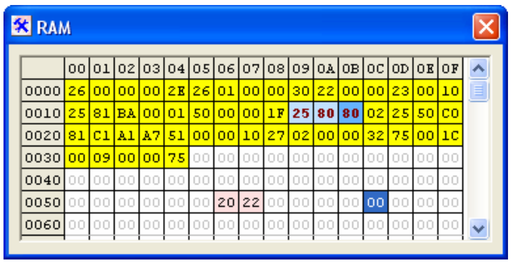
24.2.2

24.3. RAM window

No.	Test	Expected result	Passed	Reference
43	Run a program	Program code should be displayed and highlighted with yellow. Current instruction should be highlighted with blue	OK	24.3.1
44	Edit a cell	Memory byte must be updated; the new byte must be highlighted with red	OK	24.3.2



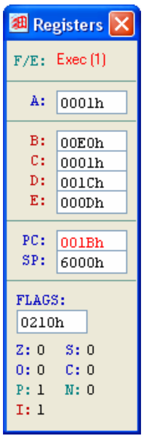
24.3.1



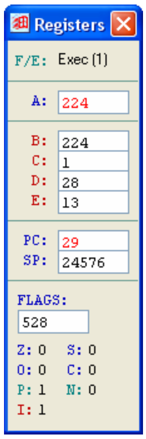
24.3.2

24.4. Registers window

No.	Test	Expected result	Passed	Reference
45	Run a program	Register values should be displayed. Registers that change must be highlighted with red.	OK	24.4.1
46	Edit a register	Register value must be updated	OK	N/A
47	Change number format	Values must be shown in the new number format (decimal unsigned in this case)	OK	24.4.2



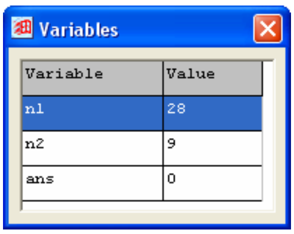
24.4.1



24.4.2

24.5. Variables window

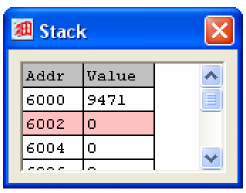
No.	Test	Expected result	Passed	Reference
48	Run a program	All variables declared in the program must be listed in the grid.	OK	24.5.1
49	Edit a variable	Variable value must be updated	OK	N/A



24.5.1

24.6. Stack window

No.	Test	Expected result	Passed	Reference
50	Push a value onto stack	The value should be displayed in the grid; stack pointer (red) must move down	OK	24.6.1
51	Return from a call	Stack pointer must move up	OK	N/A



24.6.1

25. Overall testing

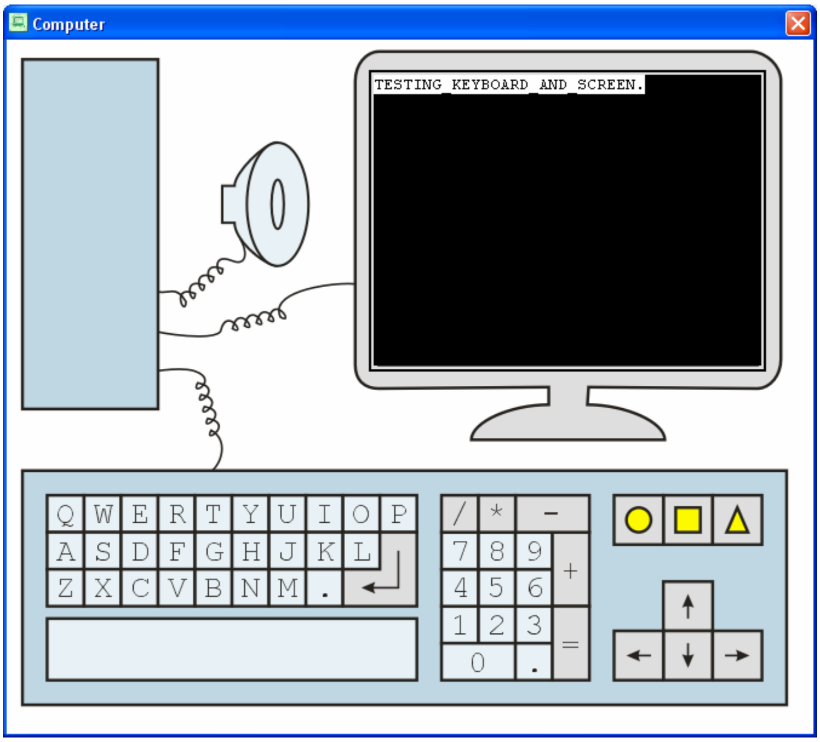
25.1. Print keys program

This program installs a keyboard interrupt service procedure which prints every key that the user presses on the screen. The source code is as follows:

```
      cli

      ;Install interrupt handler
      ld a,offset(isp_kbd)
      ld [0FF02h],a
      ;Initialise screen
      in b,52h          ;get video memory offset
      out 50h,2         ;color text mode
      out 54h,0         ;manual refresh

      ;Variables
      ld d,0            ;number of chars read
      ld e,0            ;exit flag - nonzero to terminate

      sti

      ;Main loop
lp:   xor a,a
      cmp e,a           ;test exit flag
      jz lp             ;continue if zero

      ;Terminate program
      halt

;-----------------------------------;
;--- ISP for keyboard interrupts ---;
;-----------------------------------;
isp_kbd:
      push a
      push c

      ;Get pressed key code
      in c,60h
      ;Check if it is any of special keys
      ld a,c
      cmp a,51
      jz key_exit                  ;these are after
      cmp a,50                     ;this procedure to
      jz key_cls                   ;make this clearer
continue:
      ;Output key to screen
      ld c,[c*1+offset(kbd_xlat)]  ;get key symbol
      lshr c,8                     ;remove second symbol - have read a word
      ld a,c
      ashl a,8                     ;char
      add a,0F0h                   ;color
      ld c,a
      ld [b+d*2],c                 ;print char
      out 54h,2                    ;refresh screen
      inc d                        ;increment printed num

      ;This is the end
exit_isp_kbd:
      pop c
      pop a
      iret

key_exit:
      ld e,1
      jmp exit_isp_kbd
key_cls:
      ;out   FUNCTION NOT SUPPORTED YET BY VIDEOCARD
      jmp continue
```

```
;-----------------------------;
;--- Keyboard key names map ---;
;-----------------------------;
kbd_xlat:
      ds "ABCDEFGHIJKLMNOPQRSTUVWXYZ.e_=0123456789,/*-+lrudcst"
```

The program is started, and the input is a sequence of keys pressed to type, “testing keyboard and screen.”. The program worked fine, and the ouput was:



25.2. Factorial program

This is an assembly language version of the following high-level language code:

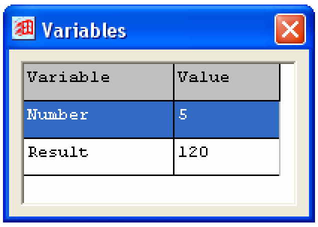
```
Result = Factorial(Number)
END

Sub Factorial(n)
  if n=1 then return 1 and exit sub
  Factorial = Factorial(n-1)*n
Exit Sub
```

This is a rather ineffective recursive implementation of the algorithm, but it helps to show why stack is needed for recursive procedures. It is also a good test of how well stack works. This is the code:

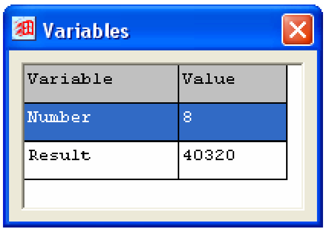
```
      ;Run Factorial with parameter Number
      ld a,Number
      call Factorial
      st a,Result
      halt
```

```
Factorial:
      ;Check if we need to exit
      cmp a,1
      jz Factorial_done
      ;We need to preserve A - otherwise recursion
      ; will modify it incorrectly
      push a
      ;Call Factorial(a-1)
      dec a
      call Factorial
      ;Get old parameter into b
      pop b
      ;Multiply the two numbers and return the result
      mul a,b

Factorial_done:
      ret

Number:      dw 5
Result:      dw ?
```

If the `Number` variable is set to 5, the output is 120, as expected (picture 25.2.1). If `Number` is 8, Result becomes 40320 (picture 25.2.2).



25.2.1



25.2.2

Maintenance

26. Introduction

This section describes the way the program functions in order to allow any programmer to correct minor bugs, or make modifications to the program.

It is important to understand that a lot was written about the way the system functions in the Design part. In this section, I will assume that the reader has the knowledge of the terms and algorithms described in that part.

27. Organisation and conventions

27.1. Modules

The program is composed of several modules, each of which has a specific function. There are two kinds of modules – procedural (*.bas) and form (*.frm). Procedural module names start with a "p", so in general procedural modules are named p*.bas. Form modules are subdivided into four categories – hardware modules (fh*.frm), device modules (fd*.frm), interface modules (fi*.frm) and system modules (fs*.frm). The functions of these modules are as follows:

- *Procedural* – contain procedures, functions, types and variables related to a particular process rather than to a form.
- *Hardware* – contain code related to a particular piece of computer hardware. These modules, unlike *device* modules, provide user with a way to view component's operation and edit its contents if any. Operation of this hardware is simulated in procedural modules.
- *Device* – each such module describes interface *as well as* operation of a device. Device modules exist for all external devices (i.e. all devices which communicate with processes through input/output ports).
- *Interface* – modules which simulate peripherals such as keyboard or screen. These devices are those that a user sees when sitting at a PC.
- *System* – these provide development windows such as code development window, register & variables contents windows etc. These windows could be thought of as an IDE (Integrated Development Environment).

Below is a list of all modules with a short description:

Procedural

- **pGlobals** – defines application entry point, declares two global variables – *Proj* and *Appp* (discussed later), defines window procedure wrapper.
- **pWinAPI** – imports Windows API functions.
- **pUtils** – implements a set of general functions not available in VB, such as conversion between different number representations etc.
- **pCompile** – contains local variables and procedures required to compile a program into machine code.

- **pExec** – contains local variables and code required to execute machine code. Exports functions to perform one clock tick, execute one machine instruction etc.
- **pIO** – provides general functions for interfacing device modules, such as PortWrite, for simple operations with multiple external devices.

Hardware

- **fhCPU** – code associated with CPU window and a little bit with CPU itself.
- **fhCU** – code associated with control unit interface window.
- **fhRAM** – code associated with RAM interface window.

Device

- **fdKeyboard** – code that determines the way keyboard controller works, plus keyboard controller user interface.
- **fdSpeaker** – code that determines the way speaker controller works, plus speaker controller user interface.
- **fdVideo** – code that determines the way video controller works, plus video controller user interface.

Interface

- **fiSplash** – the splash screen window, displayed at startup while the system is loading.
- **fiMain** – main window at the top of the screen.
- **fiComp** – computer window which contains a screen, a keyboard, a speaker and buttons to start/stop program execution.
- **fiKeyboard** – detachable keyboard.
- **fiDisplay** – detachable and scalable display.
- **fiSpeaker** – detachable speaker.

System

- **fsCode** – module providing code design and debug environment, with the primary feature of code editing.
- **fsRegs** – displays and allows modification of internal CPU registers.
- **fsVars** – displays and allows modification of variables declared in the code.
- **fsStack** – displays stack contents with stack pointer.

27.2. Visibility and naming conventions

In VB it is impossible to specify different visibility for the same declaration as viewed from different modules. If something is public then is will be visible in *all* other modules, regardless of whether they want to see it. Form modules are a special case because public procedures and variables need a qualifier (module name), but will be visible in all modules anyway, and neither public types nor arrays are allowed as public members of form modules. Therefore, I tried to minimize the number of public declarations by trying to group code in such a way that some declarations are not required outside the module. Also some naming conventions are adopted to increase code readability.

Procedures and functions

Procedures and functions are always declared with an explicit visibility modifier to make the code easier to understand. All procedures and functions in `pUtils` and `pWinAPI` are public. Otherwise they are only public if they provide some service required globally, such as the public `Tick` function in the `pExec` module. All event handlers are private because that's a VB convention. The only naming convention used is for initialisation procedure for public modules. It is prefixed with `cmp` for `pCompile`, `exe` for `pExec` and `io` for `pIO`.

Variables

All global variables are organised into two structures – `Proj` and `Appp`, discussed below. No other global variables should be declared without a good reason, and if possible such variables should be organised into global structures like `Proj` and `Appp`. No naming convention is required as global variables are already grouped into structures, and all other variables are private and thus easier to manage.

Constants

All constants in this system are local. Some of them are prefixed to group them, but no global naming convention is used.

Types

It is quite annoying that one has to declare all structured types separately in VB. So if I want to create substructures inside a structure I need to declare a separate type and name it. To simplify understanding all such “hidden but actually visible” types will be prefixed with `Tp`. The `Proj` and `Appp` types themselves are also considered “hidden” because the only reason they are delcared is to declare one single variable of the type. Some types which are part of the `Proj` structure will be declared as public because they will be used for local variables in some modules. It seems logical to me to “hide” them as well by prefixing them with `Tp`. All other types will be prefixed with `T` and declared in the `pGlobals` module.

28. Global data structures

There are only two global data structures – *Proj* and *Appp*. `Proj` stores stores all data associated with the current project and simulation state, whereas `Appp` is for data that relates to some housekeeping tasks, e.g. store a flag to indicate that application needs to terminate.

Note that although a lot of structures are declared in different modules, structures will not be discussed at all in the Modules section below.

28.1. Proj structure

`Proj` is a big structure, containing a lot of substructures. This structure is declared in `pGlobals`. Its type is called `TpProj`, it is declared as local structure type in `pGlobals`. Below is a list of all members with their types and a short description.

Project related

- **Modified: Boolean** – true if project has been modified and needs to be saved.
- **Complexity: Integer** – global complexity setting. 0 for GCSE complexity, 1 for A-level, 2 for full complexity. 0 by default.
- **NmbRep: Integer** – global number representation setting. 0 for hexadecimal, 1 for binary, 2 for decimal unsigned, 3 for decimal signed. 0 by default.

Program related

- **P: TpPrg** – stores program written by the user, compiled program and some additional information related to compilation. See below for more details.

Execution related

- **Running: Boolean** – true if simulation is on.
- **Paused: Boolean** – true if simulation has been paused. Should not be true if Running is not true.
- **Halted: Boolean** – true if simulation has been halted. Running will be false, but windows will still display all data. Otherwise windows would say that user has to start the program before using them.
- **TickCount: Long** – number of clock ticks executed since last reset.
- **CPU: TpCPU** – holds CPU simulation state, such as register values etc. See below for more details.
- **RAM: Array Of Byte** – holds the contents of RAM.
- **Video: TpVideo** – holds data about video card, such as current mode, palette memory, etc. See below for more details.

28.1.1. TpPrg substructure

This structure is declared in `pCompile` as a global structure. See algorithms below to understand the meaning of some members. Some members are declared as structured types – see below for description. This structure has the following members:

- **AsmLine: Array Of String** – holds current program in assembly.

- **TknLine: Array Of TpTokenLine** – holds tokenized program.
- **Code: String** – holds assembled program.
- **Code_O2L: Array Of Integer** – holds values to convert offset in Code to a specific line in source code.
- **Code_L2O: Array Of Integer** – holds values to convert a specific line in source code to an offset in Code.

- **Ref: Array of TpRef** – holds a list of all references with their addresses and code line at which they are declared.
- **Backpatch: Array of TpBackpatch** – holds all requests for backpatch produced during second compilation pass.
- **Vars: Array of TpVars** – holds a list of all variables declared in the code with their offsets in the memory and line where they were declared.

- **ErrL: TpErrLog** – a structure to hold all error and warning messages produced during compilation.
- **CompileNeeded: Boolean** – true if user changed the source code and the program needs to be compiled again.

28.1.2. TpCPU substructure

This structure holds CPU simulation state, such as register values etc. It contains the following elements:

General-purpose registers

- **A: Long** – the accumulator
- **R: Array (0..3) of Long** – general-purpose registers stored in such a way that B is R(0), C is R(1) etc.

Special-purpose registers

- **IP: Long** – instruction pointer – points to next instruction to be executed.
- **SP: Long** – stack pointer – points to next free stack element.
- **FLAGS: Long** – flags register.

Internal registers

- **MAR: Long** – Memory Address Register.
- **MDR: Long** – Memory Data Register.
- **CIB: String** – Current Instruction Buffer. Holds fetched instruction, each character in the string representing one fetched byte.
- **DIB: Array of TpDI** – Decoded Instruction Buffer. Holds decoded microprogram. See below for more detail.
- **Fetch: Boolean** – Indicates whether the CPU is fetching or executing an instruction.
- **FREM: Integer** – Indicates how many more bytes there are to fetch.
- **fremMem: Boolean** – This register is invisible to the user, and is only here to indicate that a memory addressing should be fetched too, thus adding another 2-4 bytes to fetch. To read more about this flag, see (Fetch process) below.
- **IS: Long** – Interrupt Status register.

Execution state

- **eSelectedReg: Integer** – indicates the number of general-purpose register selected by the Control Unit for a read/write operation. Can be 0 to 5, respectively, for the following registers: `b`, `c`, `d`, `e`, `sp`, `ip`.
- **eIDB: Long** – value held on the Internal Data Bus. Volatile between different clock cycles.
- **eIAB: Long** – value held on the Internal Address Bus. Volatile between different clock cycles.
- **eDP: Integer** – Decoded Instruction Pointer. Points to the next microinstruction to be executed by the Control Unit in the Decoded Instruction Buffer.
- **eEDB: Long** – value on the External Data Bus (a.k.a. Data Bus). Volatile between different clock cycles.
- **eEAB: Long** – value on the External Address Bus (a.k.a. Address Bus). Volatile between different clock cycles.
- **Breakpoint: Array of Long** – array holding values of `IP` register, encountering which the execution process should stop and hand over control to the Integrated Development Environment. Has nothing to do with CPU simulation.

28.1.3. TpDI

This structure represents a single decoded microinstruction. To read more about how this structure is used, please refer to (Decode process) and (Execute process).

- Sig1: Long – low-order word of control signals
- Sig2: Long – high-order word of control signals
- nToIDB: Long – data to be put on IDB
- nAluOpNum: Integer – ALU operation number
- nJmpCond: Integer – conditional jump number
- nAdrMul: Integer – address multiplier for indexed addressing
- nAluSh: Integer – ALU shift count for shift operations
- nIntIS: Integer - unused

28.1.4. TpVideo substructure

This structure holds data about video card, such as current mode, palette memory, etc. It consists of the following elements:

- **Mode: Integer** – mode number
- **autoUpdate: Boolean** – indicates whether screens are updated automatically by CLab or manually by the user.

- **mdResX: Integer** – horizontal screen resolution for current mode.
- **mdResY: Integer** – vertical screen resolution for current mode.
- **mdType: Integer** – mode type – 0 for text, 1 for direct graphics, 2 for paletted graphics.

- **mdColors: Integer** – color resolution for current mode – 0 for monochrome, 1 for 16 colors, 2 for 256, 3 for 65536 and 4 for 16777216.
- **mdFntX: Integer** – width of one character in screen pixels.
- **mdFntY: Integer** – height of one character in screen pixels.

- **MemOff: Long** – offset to video memory in RAM.
- **PalMem: Array (0..255) of Long** – palette memory
- **vDC: VirtualDC** – a "bitmap" in memory where all drawing occurs and which is then drawn on simulated screens. VirtualDC is a class written by an unknown author, with slight modifications.

28.1.5. TpToken

For more information about how this structure is used please refer to (Compile process) section.

- **Text: String** – string from source code containing the token, e.g. "AND".
- **Type: Integer** – token type, one of the `tk` constants, listed below.

28.1.6. Token Type constants

- **tkUnknown = -1** – used during tokenization
- **tkLabel = 0** – token is a label declaration
- **tkVarDecl = 3** – token is a variable declaration
- **tkVarInit = 4** – token is a variable initialisation
- **tkOpcode = 5** – token is an opcode
- **tkOperand = 6** – token is an operand

28.1.7. TpTokenLine

For more information about how this structure is used please refer to (Compile process) section.

- **Token: Array of TpToken** – all tokens which are located on this line.
- **CodeLine: Integer** – line number in the original code where this line came from.
- **CodeOffset: Integer** – address of whatever code is produced by this token line.

28.1.8. TpRef

For more information about how this structure is used please refer to (Compile process) section.

- **Name: String** – label/variable name.
- **Addr: Long** – address that the reference points to.
- **CodeLine: Integer** – line number in original code where this reference was declared.

28.1.9. TpBackpatch

For more information about how this structure is used please refer to (Compile process) section.

- **Name: String** – name of label/variable referred to.
- **Addr: Long** – where to write the offset of requested label/variable
- **IsDW: Boolean** – unused
- **RelTo: Long** – unused
- **CodeLine: Integer** – line number in original code where this reference was requested.

28.1.10. TpVars

For more information about how this structure is used please refer to (Compile process) section.

- **Name: String** – variable/label name
- **Addr: Long** – variable/label address

28.1.11. TpErrLog

For more information about how this structure is used please refer to (Compile process) section.

- **sError: Array of String** – error message
- **lError: Array of Integer** – number of line producing error
- **nError: Array of String** – error number
- **sWarning: Array of String** – warning message
- **lWarning: Array of Integer** – number of line producing warning
- **nWarning: Array of String** – warning number

29. Processes

This section describes core processes in detail, looking at what happens at each stage and why it happens. This section does *not* look at helper functions used in the processes – those will be discussed later in (Main functions and procedures).

29.1. Startup

Application entry point is defined in the pGlobals module. The function is called `Main`, it returns no value and has no parameters. Below is a list of actions that CLab does when it starts up.

1. Set Running flag, get windows version and initialise XP controls if necessary.
2. Load and display splash screen.
3. Load all forms that CLab contains; this takes most of the startup time.
4. Initialise `Proj` structure.
5. Initialise all modules which require initialisation.
6. Show main form and computer form.
7. Hide and destroy splash screen.

According to VB6 help, the application will keep running after `Main` returns for as long as at least one form is loaded. So we do not need the message loop; VB does it all for us. Please refer to the next section to read about shutting down process.

29.2. Shutdown

To shutdown, CLab unloads all forms. This causes the message loop to stop, and the application terminates. User interface is designed in such a way that the user can only close the main window to shut down the application. Closing any other window simply causes that window to be hidden.

Visual Basic's `End` statement is supposed to terminate the program by unloading all forms. But apparently it does something else, because `End` causes CLab (and VB too) to crash, whereas manually unloading all forms in a loop works fine. Therefore, a loop is used instead of `End`.

When the user closes the main window, `Terminating` flag will be set to `True`, all forms will be asked about shutting down, and if they all agree then they are unloaded. This time forms *will* be unloaded (as opposed to being hidden) because of the `Terminating` flag.

29.3. Assembly

Assembling a program requires five stages, three of which are pure assembling, one – preparation and one – postprocessing.

Preparation

Code is copied into the Proj structure, and the structure is prepared for assembly.

Pass 1

During this pass the program is tokenized (i.e. split into tokens). First of all, the source code is cleaned by removing all comments, ensuring all sequences of tabs and spaces are replaced with single spaces, and then trimming leading and trailing spaces, if any.

Next, the program is split into tokens in the following way. For every non-empty line of source code a token line is created. The program loops through the characters of the line, accumulating them in a special variable. Whenever it encounters a space, it saves whatever it has accumulated as a token in the token line and starts accumulating next token.

Having tokenized the whole program, CLab tries to identify token types. At first it identifies *label* tokens (if there is a semicolon at the end then it is a label) and *vardecl* tokens (if token is `DB`, `DW` or `DS`). Everything else is identified as *unknown* for now. Next CLab analyses token positions to further identify them. This time it ignores all *label* tokens. If the first token is *unknown*, then it is identified as *opcode*, otherwise it remains as it is. Now all *unknown* tokens after an *opcode* token are identified as *operands* and all unknown tokens after a *vardecl* token are identified as *varinit* tokens.

The next stage in this pass is to analyse all token patterns and see if they are valid. Some patterns may be corrected; others may not. First of all, all *label* tokens are placed in their own token lines so that there is only one label per token line and nothing else. Next, CLab checks if there is a *vardecl* token without a *varinit* token after it. If there is, it issues a warning and adds an unitialised *varinit*. Having done this, the algorithm is ready to check token patterns. There are only a few token patterns that are valid at this stage. They are:

label
vardecl varinit
opcode
opcode operand
opcode operand operand

If a given token line does not follow any of these patterns at this stage, an error message is issued, and assembly process is stopped.

The final stage of this pass is to prepare references to variables in such a way as to simplify compilation. Whenever the user wants to use memory addressing, he can either write the address of the variable or variable name, which will be replaced with the address by the compiler. The following notations are equivalent: `[50h]` and `var`, given that variable `var` is stored at address `50h`. There are several addressing levels, which are listed below:

	Pointer	Dereference	Double dereference
Specified by address	`50h`	`[50h]`	`[[50h]]`
Specified by name	`offset(var)`	`var`	`[var]`

There is an obvious mismatch in formats – writing a variable name dereferences its address automatically, whereas writing memory address does not. Therefore, it is impossible to replace variable name by its address directly. The `offset()` syntax makes things even worse. At the end of first assembly pass the program “shifts” all addressings specified by variable name in the following way. If it encounters the structure `offset(X)`, it replaces it with the structure `X`. If it encounters something that must be a reference, it checks whether it is a number or a variable name. If the latter is true then it encloses the variable name with `[]`. It is easy to see that now the table shown above will look like this:

	Pointer	Dereference	Double dereference
Specified by address	`50h`	`[50h]`	`[[50h]]`
Specified by name	`var`	`[var]`	`[[var]]`

Now all variable names can simply be replaced by their addresses. Note that the reason for all this is to simplify assembly language syntax – it would not be nice if users had to write `ld a,[var]` instead of `ld a,var`.

Pass 2

This pass generates machine code for the tokenized program. It loops through all token lines, generating code for every line and adding it to the `Proj.P.Code` string. In this way the current address (the address of the instruction being compiled) will be the length of `Proj.P.Code` string. Below is a description of what the algorithm does for every token line.

First, the algorithm checks if the first token is a label. In that case the algorithm will check if the variable name is valid (and issue an error if it isn’t). Then it will add the label to the reference list (`Proj.P.Ref`), storing reference name, address and source code line.

Next the algorithm starts to actually convert source code into machine codes. To reduce the amount of work, all similar instructions are assembled in loops. There are arrays for every group of similar instruction, containing opcode and the corresponding machine code. The loops go through all opcodes in array, comparing them to the one that is being assembled. If the operand belongs to none of the groups, it is assembled individually.

Having assembled all token lines, this pass generates arrays which help converting between source code lines and offsets in machine code. This is mainly used for breakpoints and to highlight machine code instruction being executed.

Pass 3

This is the last assembly pass. Here the program checks if there are any multiple label definitions or any undeclared references, and then backpatches the program. The algorithm goes through the list generated during the second pass, writing the addresses of all references as requested.

The reason why a separate pass is required to backpatch is that the program does not know addresses of all references until it has finished code generation.

Postprocessing

Having assembled the program, CLab will list all errors and warnings (if any). It will also update all windows to reflect changes.

29.4. Fetch

To fetch an instruction, CLab gets the byte at current `IP`. It then checks if it has already fetched some bytes. If it hasn't, it will calculate and store the number of bytes to fetch, using a special array, `InstructionLen`, generated at startup in `exeInit`. Otherwise the program will simply fetch bytes and add them to `CPU.CIB`. If the program detects that it has finished fetching an instruction which uses a memory addressing, it will use the last fetched byte to determine the length of memory addressing which will be appended to current instruction.

As soon as the last byte is fetched, instruction will be decoded into machine codes, and the CPU will switch to execution mode.

29.5. Decode

Decoding is rather similar to code generation. Some instructions are grouped. There are arrays which hold machine codes for every group, which are compared to current machine code. As soon as instruction is identified, a microprogram which can be directly executed by the CPU is generated.

A list of all microinstructions can be found in section (Microinstructions). A table of microprograms for all machine codes can be found in (reference).

29.6. Execute

All that the execute cycle does is to take every microinstruction from `Proj.CPU.DIB` and do actions if a respective signal is set. There is a list describing what each signal does – see (Microinstructions).

Having executed all microinstructions, the CPU will prepare to fetch the next instruction and then run the interrupt check algorithm. It will be that algorithm that will switch to fetch if no interrupts are pending.

29.7. Interrupt

To check for pending interrupts, CLab goes through the bits of `IS` register, starting with the low-order bit, which corresponds to interrupt request 0, thus giving it highest priority. As soon as it encounters a bit set to 1, it will go and create a microprogram to initiate the interrupt. This microprogram can be found in the (Appendix).

30. Functions and procedures

30.1. pGlobals

30.1.1. Main

This is the application entry point. It initialises the whole application, loading all settings and showing relevant windows.

30.1.2. WindowProc

This is a wrapper for the real window procedure of `fiMain` form. The reason is that VB does not allow to get the address of *any* procedure declared in a form module. But we have to know the address of the window procedure to hook the window with the `SetWindowLong` WinAPI function. Thus this clumsy wrapper in this module.

30.2. pUtils

Errr

Displays an error message containing the string passed to this procedure. It is just a shorthand – this way we don't have to bother about the caption, the icon and buttons.

Tally

Counts the number of occurrences of one string in another string.

FieldStr

Returns a specified element from a list stored in a string separated by a special character.

InStrBack

The same as InStr except for the fact that it works backwards. It starts looking for occurences at the end of the string, and returns the position of the first one.

Hex2Dec

Converts a hexadecimal number to a decimal number.

Dec2Hex

Converts a decimal number to a hexadecimal number.

Bin2Dec

Converts a binary number to a decimal number.

Dec2Bin

Converts a decimal number to a binary number.

Dec2Chr
Converts a decimal number to a base 256 number, returning the result as a big-endian string.

Chr2Dec
Converts a base 256 number to a decimal number. The source number is interpreted as big-endian.

Str2Chr
Formats source string by writing each character in hexadecimal, separated with a space.

TestCharset
Tests if all characters of a given string belong to a given charset.

StringIsInt
Returns true if a given string is a decimal integer.

StringIsLong
Returns true if a given string is a decimal long integer.

GetFilename
Initiates an open or a save dialog using ComDlg functions `GetSaveFileName` or `GetOpenFileName`.

AppDir
Returns application path with a backslash at the end.

Dec2Fmt16
Converts a decimal number to one of the supported number representations.

IsFmt16
Returns true if a given string is a valid 16-bit number in one of the supported number representations.

Fmt2Dec16
Converts a number in any of the supported representations to a decimal number.

FntWrite
Writes text on a given device context using a given font. The reason for using this is that for some reason a font selected into a DC is removed from that DC after the first text output to that DC. Therefore, a font has to be selected every time.

CreateFnt
Creates a font by initialising an application-defined font structure and creating respective GDI objects.

DestroyFnt
Destroys a font created by CreateFnt by destroying respective GDI objects.

30.3. pCompile

cmpInit
Initialises the module by arrays for all opcode compilation groups….

ReadCodeIntoProj
Copies the code written by the user from the text editing control into the `Proj` structure.

PrgCompile
Assembles user program by reading the code into `Proj` structure, initialising some variables, running all three assembly passes, displaying all errors and warnings, and updating all windows.

PrgLoad
Loads an assembled program into RAM and updates RAM window.

CompilePass1
Assembly pass 1. Described in detail in (23.3 assembly).

CompilePass2
Assembly pass 2. Described in detail in (23.3 assembly).

CompilePass3
Assembly pass 3. Described in detail in (23.3 assembly).

CompileMemoryAddressing
This function is used in `CompilePass2`. Compiles a given memory addressing operand into machine code which can be added to the instruction that requires it.

OperandIsRg
Returns true if a given operand is a register (`A`, `B`, `C`, `D` or `E`).

OperandIsRgn
Returns true if a given operand is a general-purpose register (`B`, `C`, `D` or `E`).

OperandIsMem
Returns true if a given operand is a memory addressing.

OperandIsIm8
Returns true if a given operand is an 8-bit immediate constant.

OperandIsIm16

Returns true if a given operand is a 16-bit immediate constant. If the operand is a variable name, the function will still succeed because variable address is a known constant.

CIm8

Converts an operand into an 8-bit number. No error-checking – this function assumes OperandIsIm8 was called on the same operand to check validity.

CIm16

Converts an operand into a 16-bit number. No error-checking – this function assumes OperandIsIm16 was called on the same operand to check validity. If the operand is a variable name, the function will file a backpatch request and return 0.

AddErr

Adds a given error to the error list.

AddWng

Adds a given warning to the error list.

CleanSpaces

Converts all sequences of tabs and spaces into a single space. Used in `CompilePass1`.

30.4. pExec

exeInit

Initialises the module by filling an array of instruction lengths and some group decode arrays.

GFlg

Checks if a given signal in a given microinstruction is set.

SFlg

Sets a given signal in a given microinstruction. Signals are passed as separate parameters. Some signals (their names are prefixed with `op_` for Operation) cause `SFlg` to interpret the following parameter as a number and save it in a special register in `TpDI`, depending on what `op` signal was used.

Tick

Executes one clock tick. This is a public function being a wrapper for the private function `eTick`.

Step

Executes one whole instruction, fetching the next instruction.

eTick
Executes either a fetch or an execute cycle, then calls Tick procedure for all device modules. Also increments the tick counter.

eFetch
Fetches one byte. For a detailed description see (Processes.Fetch).

eDecode
Decodes instruction in `Proj.CPU.CIB`. For a detailed description see (Processes.Decode).

eExecute
Executes one microinstruction. For a detailed description see (Processes.Execute).

eInterrupt
Checks for interrupts. For a detailed description see (Processes.Interrupts).

DecodeMem
Adds such microinstructions to `Proj.CPU.DIB` as to calculate the address specified by the memory addressing in `Proj.CPU.CIB` and store it in `MAR`.

reg_sX
Returns “select register X” signal for the signal specified by an integer (0 for B, 1 for C, 2 for D, 3 for E).

DI2Str
Generates a string with the names of all signals in a given microinstruction.

30.5. pIO

devInit
Initialises all device modules by calling respective initialisation procedures.

devReset
Resets all devices by calling respective reset procedures.

devTick
Lets all devices to do some processing every tick if there is anything they want to do.

devPortRead
Queries all devices if any of them wants to respond to a port read signal with a given address.

devPortWrite
Calls PortWrite for all devices thus simulating a port write operation.

IRQ

Devices call this function to request a given interrupt. Returns false if CPU cannot accept the interrupt. Otherwise sets "pending" flag and returns true.

30.6. Common functions (all form modules)

This section describes functions common to all form modules.

Init – Initialises the module.

Form_Unload – Either hides or unloads the form depending on whether the application is terminating or not.

Update – Changes data on the form to reflect changes to simulation state.

SaveLast – Stores register values in order to highlight them if they change.

30.7. Common functions (device modules)

Reset – initialises the device whenever the user restarts the program.

Tick – does some processing every clock tick.

PortRead – port read operation for the given device.

PortWrite – port write operation for the given device.

30.8. Other functions worth mentioning

fiMain.Hook

Hooks the main window by installing application-defined window procedure.

fiMain.Unhook

Unhooks the main window by returning the VB-defined window procedure. The main window *has* to be unhooked before unloading it, otherwise VB crashes.

fiMain.WindowProc

Window procedure for fiMain initiated via a wrapper defined in pGlobals. Traps minimize event and hides all forms. Also traps restore event and shows all forms hidden during minimization. Traps clicks in the non-client area and activates the form (VB's message procedure does not do that if the form is non-movable).

fdVideo.SetVideoMode

Sets a video mode. Video mode number is the same as the one used with port write operation.

fdVideo.UpdateScr

Repaints the video memory on the internal memory device context, ready to be blitted onto screens.

fsCode.InvalidateRTB

Invalidates every line of the code editor, causing it to repaint fully.

fsCode.DisplayError

Highlights a given error/warning in the error list by showing it in the code editor and highlighting the offending line. Optionally displays an error message with the error/warning text.

fsCode.TransferBreakpoints

Copies breakpoints from the code editor into the `Proj` structure.

31. Sample modifications

31.1. Renaming opcodes

It is very easy to rename opcodes. Opcode names are only used in `pCompile` module. Use text search to find the opcode name – it will be either in `cmpInit` (for grouped instructions) or in `CompilePass2`. Make sure that the new name is lowercase – assembly language is not case-sensitive, so all comparisons are made in lowercase.

31.2. Adding an instruction

Adding an instruction is not exactly straightforward. Many parts of the program will need modifications. The program was not designed with easy instruction set updatability in mind.

Assembling a new instruction

CompilePass2 needs updating. There is a big block of code inside a loop through all token lines. Inside that block there is a variable called `t` which contains current opcode in lowercase. You will need to add an IF block to check if the program is trying to compile the new instruction. The code inside the block will have to do all the assembling and end with a `GoTo NextTokenLine` statement. Whatever your code has assembled must be placed in variable `ctl` – it will be added to machine code automatically.

When assembling an instruction, `tl` will contain current token line. So if you need to check the number of parameters, use `tl.Count`, and `tl.Token()` array will contain all tokens in the token line (e.g. `tl.Token(0)` will return your opcode).

You may find the following functions useful when assembling your instruction. To check what type a given operand is, use `OperandIsRg`, `OperandIsRgn`, `OperandIsIm8`, `OperandIsIm16` or `OperandIsMem`.

Make sure that the first byte of your machine code equivalent is unique to your instruction – otherwise CLab will have problems decoding it.

If the syntax is not correct, use `AddErr` to issue an error. If there is an assumption you make, and you want the user to be aware of it, use `AddWng` to issue a warning.

If one of your operands is a memory addressing, use the `CompileMemoryAddressing` function. Pass to it your operand and some backpatching information (see declaration for details), and it will return compiled memory addressing bytes which can be easily decoded with DecodeMem

Decoding a new machine code

First of all, add the first byte of your machine code to the `InstructionLen` array initialisation. If N is the number corresponding to the first byte of your machine code then the N^{th} element of the array must be the length of your machine code instruction.

If you use memory addressing then specify the length of the instruction without the addressing, with a minus sign.

Now you can decode the instruction. Add your code to the end of `eDecode` procedure. Variable `b` will contain the first byte of the machine code – check if that's the one you want to decode. Now redim `.DIB` with elements from `-1` to how many machine code instructions you will need subtract 1. Call `SFlg` for each element of .DIB, supplying it with all signals that you want.

If you want to place a condition, use `op_jmp_cond` flag followed by one of the numbers from the table below. If a condition is not met, CPU will stop executing current microprogram. Possible conditions are:

0	if greater
1	if not greater (if less or equal)
2	if less
3	if not less (if greater or equal)
4	if equal (if zero)
5	if not equal (if not zero)
6	if carry
7	if not carry
8	if overflow
9	if not overflow
10	if sign
11	if not sign

For example, `SFlg(.DIB(0), op_jmp_cond, 6)` will ensure that your microprogram will only be executed if carry flag is set to true.

If you want to place a constant to IDB, use `op_idb_im`, followed by the constant you need.

To decode a memory operand, place all microinstructions that you need before calculating the address, then call DecodeMem. It will place microinstructions in `Proj.CPU.DIB` which will calculate the address and place it into `MAR`. Redim `.DIB` with a `Preserve` keyword and calculate element number using `UBound(.DIB)` if you need to add more microinstructions.

31.3. Changing the amount of RAM

The amount of RAM is really hard-coded into CLab, so changing it is pretty much impossible. You could in theory reduce the amount of RAM by placing a check whenever RAM is accessed and issue an error if accessing out of bound, but why would you need less RAM? If you wanted to increase the amount of RAM, you would have to increase all register sizes (because 65k RAM uses all bits of 16-bit registers), and you may have to rewrite a lot of code associated with decoding and executing instructions, especially ALU instructions. You would also have to rewrite compilation in order to allow for constants bigger than 16 bits.

32. Appendices

32.1. Error and warning messages

All error and warning messages have codes associated with them. These codes have the following syntax. The first letter is either E for error or W for warning. Second and third characters indicate where the error occured (C1 – assembly pass 1, C2 – assembly pass 2, C3 – assembly pass 3). The last three digits indicate error number.

EC1002 – Invalid token combination: X and Y.
EC1003 – Invalid token combination: X, Y and Z.
EC1004 – A line cannot contain more than three tokens. This line contains X tokens.

EC2001 – Opcode takes 0 operands, not X.
EC2002 – Variable initialisation sequence is neither '?' nor a valid constant.
EC2005 – Syntax error in operand OR opcode and operand incompatible. Offending operand: X.
EC2006 – The number of shift cycles must be between 0 and 15.
EC2007 – 16 bit immediate constant is out of range.
EC2008 – Invalid label name: X.
EC2009 – Label name cannot be same as register name.
EC2010 – Opcode not recognized: X. Check spelling.
EC2011 – Memory addressing scaling factor should be 0, 1, 2 or 4.
EC2013 – Cannot load into a constant (first operand cannot be a constant).
EC2014 – Cannot load a constant into a memory cell directly.
EC2015 – Cannot store in a constant (second operand cannot be a constant).
EC2016 – Cannot store a constant in a memory cell directly.
EC2017 – Operand for INT must be an 8 bit immediate constant.
EC2018 – Port address must be an 8 bit immediate constant (0 to 255).
EC2019 – DS variable should be initialised with either ? or a string literal enclosed with “ ”.

EC3001 – Label already declared: X. Previous declaration on line Y.
EC3002 – Undeclared reference: X.

WC1001 – Labels must not be preceded by other tokens. Label moved to beginning of line. Offending label: X.
WC1002 – Variable not initialised explicitly. Assuming uninitialised variable.

32.2. Microinstructions

The following signals can be parts of a microinstruction.

Name	Num	Description
`reg_sb`	`0`	Select B register
`reg_sc`	`1`	Select C register
`reg_sd`	`2`	Select D register
`reg_se`	`3`	Select E register
`reg_ssp`	`4`	Select SP register

reg_sip	5	Select IP register
reg_r	6	Read from selected register to IDB
reg_w	7	Write from IDB to selected register
reg_ipi	8	Increment IP by 1
reg_spi	9	Increment SP by 2
reg_spd	10	Decrement SP by 2
adr_br	11	Use B register in addressing
adr_sr	12	Use selected register in addressing
adr_im	13	Use constant from IDB in addressing
adr_c	14	Calculate addressing and place result on IAB
lea_ad	15	Load value on IAB to IDB
acc_r	16	Read from accumulator to IDB
acc_w	17	Write from IDB to accumulator
flg_r	18	Read from FLAGS to IDB
flg_w	19	Write from IDB to FLAGS
ctl_mr	20	Send Memory Read signal
ctl_mw	21	Send Memory Write signal
ctl_pr	22	Send Port Read signal
ctl_pw	23	Send Port Write signal
mdr_ri	24	Read from MDR to IDB
mdr_re	25	Read from MDR to EDB
mdr_wi	26	Write from IDB to MDR
mdr_we	27	Write from EDB to MDR
mar_ri	28	Read from MAR to IAB
mar_re	29	Read from MAR to EAB
mar_wi	30	Write from IAB to MAR
mar_we	32	Write from EAB to MAR
alu_swp	33	Swap operands when performing ALU operation
ctl_halt	34	Halt the CPU
lea_da	35	Load value on IDB to IAB
flg_stz	36	Set zero flag
flg_clz	37	Clear zero flag
flg_stc	38	Set carry flag
flg_clc	39	Clear carry flag
flg_sto	40	Set overflow flag
flg_clo	41	Clear overflow flag
flg_sts	42	Set sign flag
flg_cls	43	Clear sign flag
flg_sti	44	Set interrupt flag
flg_cli	45	Clear interrupt flag

The following signals mean that specific data associated with them is present in a respective `TpDI` element.

Name	Num	Description	Data in
op_alu_sh	58	ALU shift operation – number of shifts	.nAluSh
op_jmp_cond	59	Jump condition should be checked	.nJmpCond
op_idb_im	60	A constant should be placed on IDB	.nToIDB

op_adr_mm	61	Addressing multiplier	.nAdrMul
op_alu_c	62	ALU operation must be performed	.nAluOpNum

33. Notes

33.1. Error policy

There are three types of errors in CLab. One is *source code errors* – they occur when there is something wrong with user program because of user error and the program cannot be compiled. Another one is *usage errors* – they occur when the user does something he is not allowed to do. This could be, for instance, entering an incorrect value somewhere in a dialog. The third type of errors is *internal errors*. These are errors that occur when the program does something it is not supposed to do because of a programming error. I tried to foresee what could go wrong had I made a tiny error somewhere and inserted error traps in such sensitive places. If indeed I did make such a tiny error I wouldn't spend a lot of time trying to locate the error – I will see the point where things first started to go wrong. Internal error messages always have a technical explanation of what went wrong, which the user is not supposed to understand. They also ask the user to contact the author.

33.2. Instruction Pointer vs Program Counter

While the program was designed and impleneted, this register was called `IP` for instruction pointer, which seems to be a much more logical and easier to remember name. But after consulting with my end-user, I realised that all syllabuses teach this register as `PC` for Program Counter. So I changed the name everywhere where the user will see it, but I kept it everywhere else. Now the register is called `PC` for the user and it has both names to the programmer.

33.3. Microinstructions and Design

It was not clear at design stage whether real microinstructions would be really necessary. While implementing the solution I realised that this is the easiest way to show the user all that I needed to show about internal workings of a computer. Therefore microinstructions are not mentioned in Design stage. All necessary information about them can be found in this section, esp. in the Appendix.

User manual

A-level

34. Introduction

This system is designed to help you learn some topics which are on your Computing syllabus. There are two main areas this program can help you with. They are *assembly language* and *computer internals*.

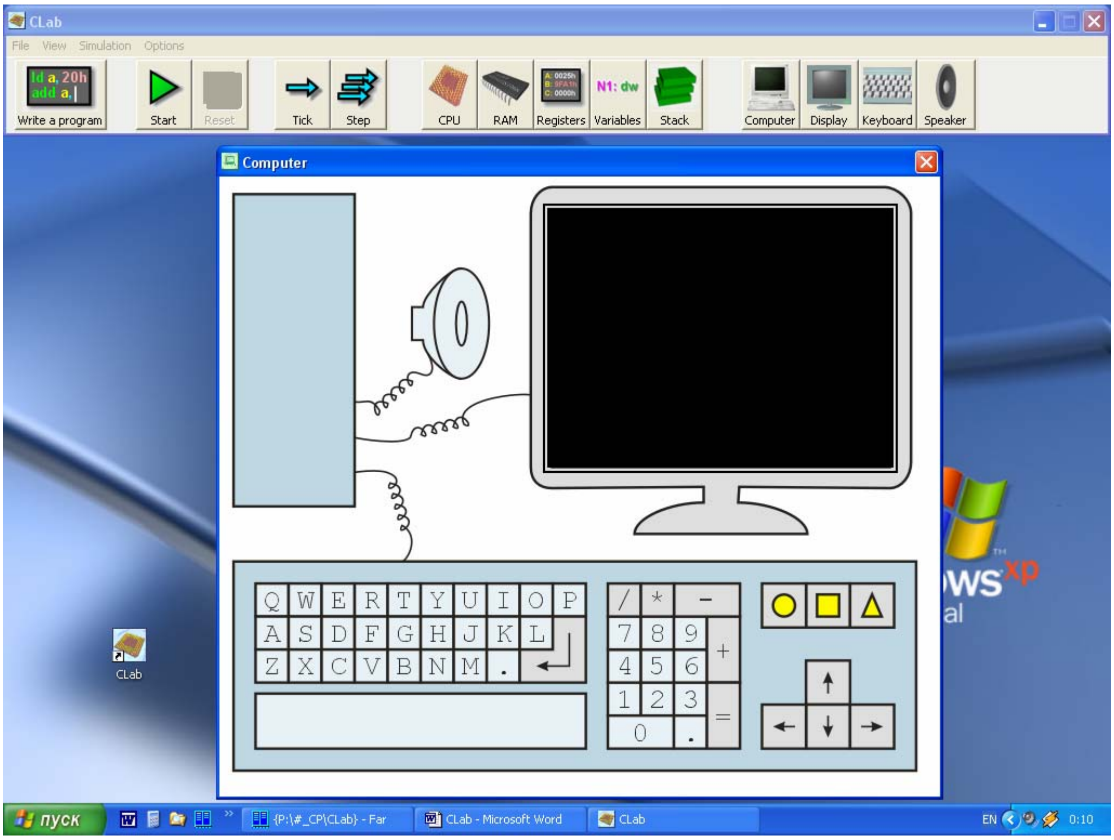
With CLab you can:

- Write programs in assembly language and run them.
- See how each instruction changes values of registers and variables.
- Easily trace loops and stacks – something very complicated when done on a whiteboard.
- See the structure of a computer on several levels, from peripherals down to CPU core.
- Investigate interactions between different components of a computer in real-time.

But most importantly, you will be able to see how your program interacts with hardware – that is, you will be able to see exactly what each instruction does, which can be extermely useful in understanding computers.

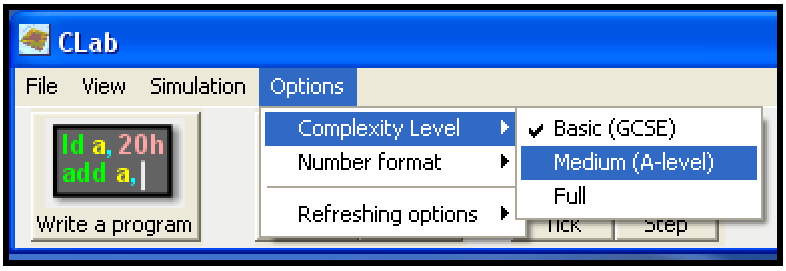
35. Writing and running programs

When you start the system, you see the following two windows on the screen:

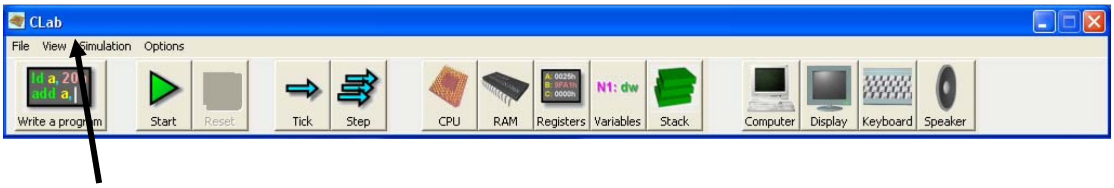


Throughout this manual the window on the top will be called the Main window. Note that *all* program functions can be accessed through the Main window. To close the program, close the Main window.

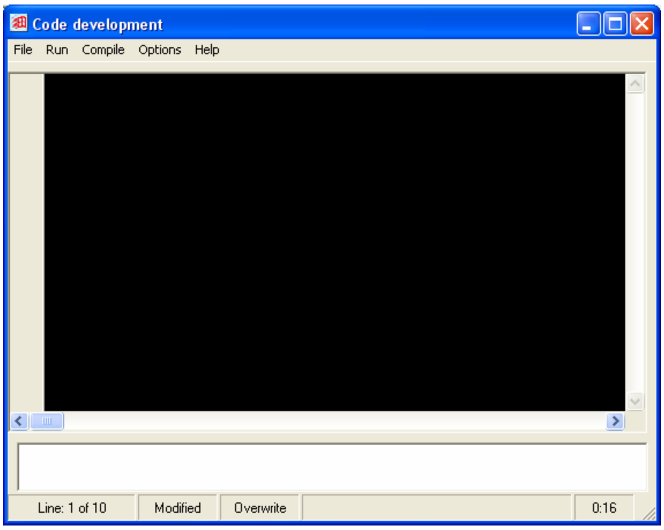
First of all, you will need to select the correct complexity level. Go to Options/Complexity level menu and select the level you need:



To write a program, click the “Write a program” button. The program editor will pop up:

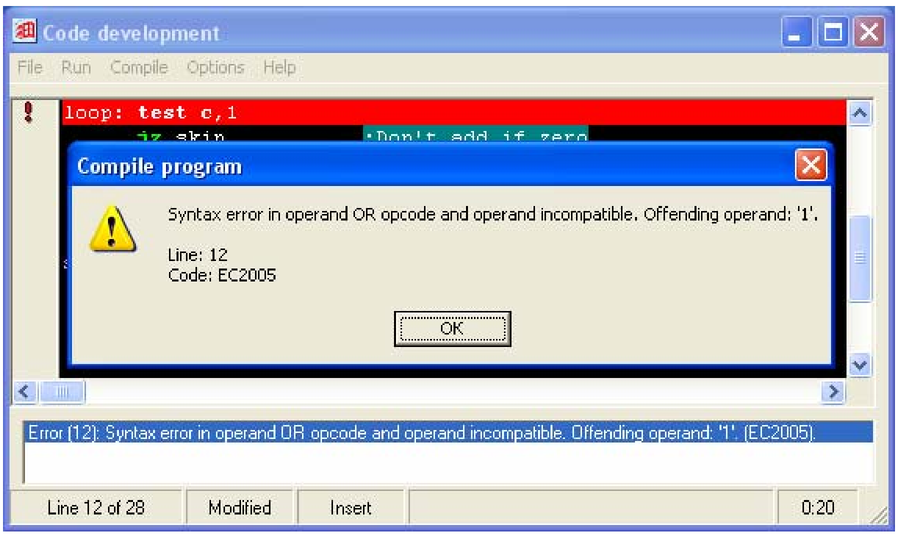


Click – the editor shows:



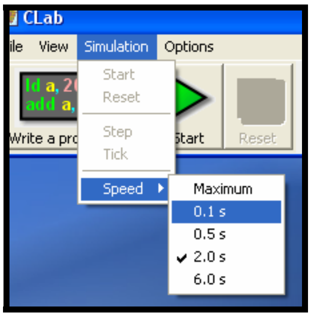
Now you can type your program! Refer to the Assembly language manual to find out how to write programs.

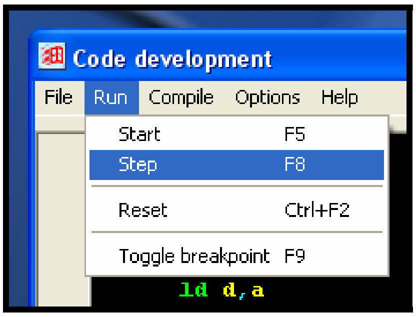
When you have finished writing your code, press `F5` to start the program. If the code is correct, the program will be executed. If there are errors, CLab will list them to you at the bottom of the screen and display an error message for the first error. It will also highlight the first error with red:



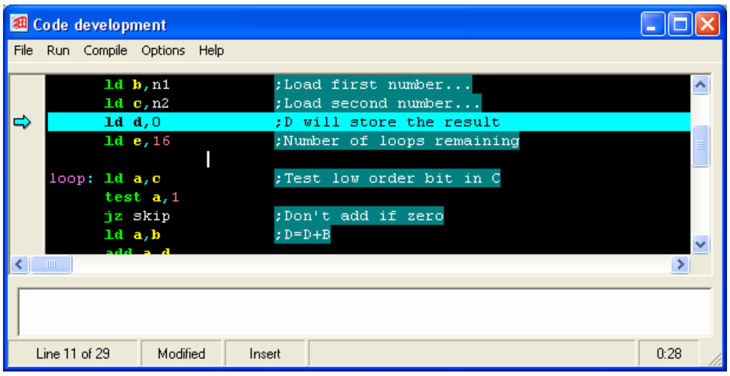
Feel free to play with your code for a while. Use the Registers and Variables button in the Main window to view the contents of registers and variables. These functions will be discussed in more detail later.

When you press F5 (you can use the Run/Start and Run/Continue menu as well), CLab will execute the instructions in your program one by one, at a set speed. You can change this speed in the *Simulation* menu in the Main window. If you don't want the program to execute the next instruction until you tell it to, use F8 key (Run/Step menu).

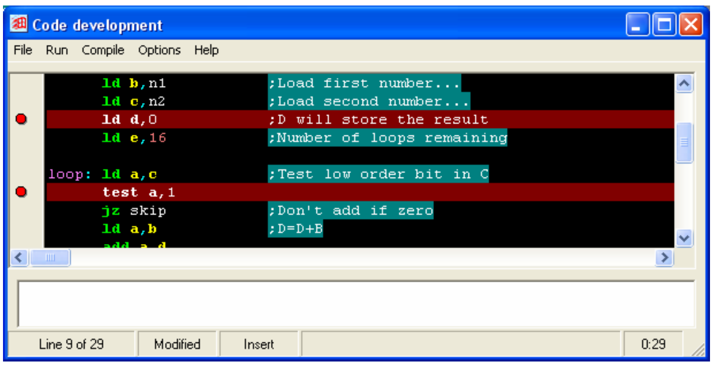




Whenever a program is being executed, the current instruction will be highlighted with aqua colour, with the exception of the maximum speed:



If you want your program to stop at a certain point, use *breakpoints*. CLab will pause program execution whenever it reaches a line with a breakpoint. To place a breakpoint, move the cursor to the line you want and press F9. You can also click with the mouse to the left of the line:



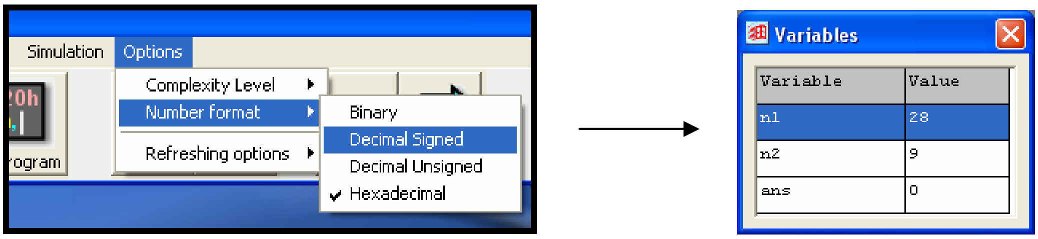
Breakpoints have no effect when you execute program step by step.

36. Tracing program execution

If you want to see how exactly your program works, you should first of all run it in either step by step mode or set exectution speed to slow (see above to find out how to do that). Whenever you are in one of these modes, the current instruction being executed is highlighted.

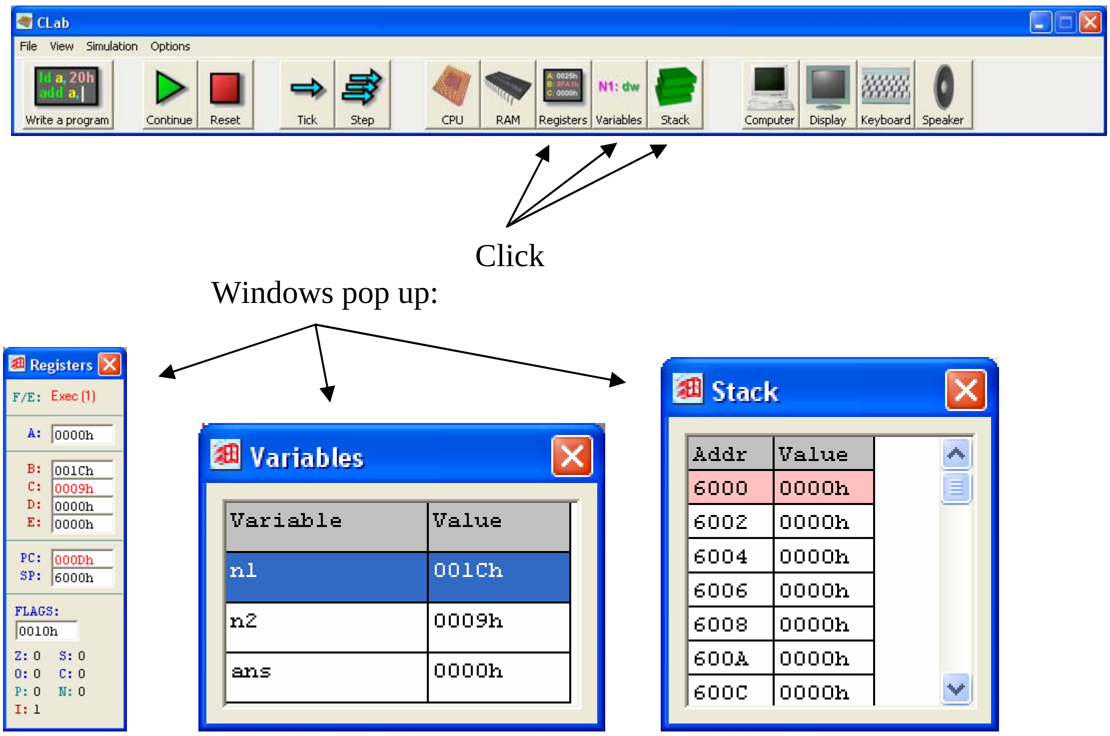
36.1. Number representation

If you want to see all values in decimal or binary rather than hexadecimal, go to Options/Number format menu in the Main window and select the format you need:



36.2. Debugging

The easiest way to investigate how your program works is by watching registers. Click on the Registers button in the main window, and the Registers window will pop up. You can also view variables declared in your project and see the stack:



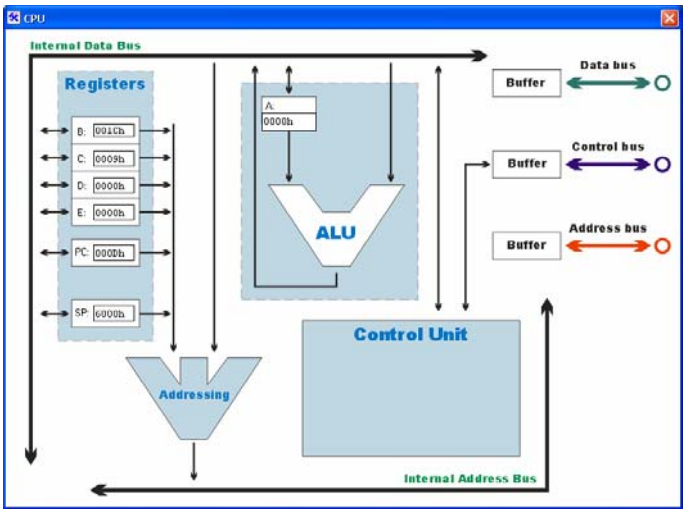
Registers window will show you all the registers accessible to you when you write programs. All registers that have changed since last instruction will be highlighted with red, as in the example above. You can edit registers by simply typing in the new value and pressing `Enter`.

The Variables window will also let you edit variables – just select the one you want and type in the new value. You cannot edit stack. In the stack window, the value to which Stack Pointer points is highlighted with red colour, as in the example above.

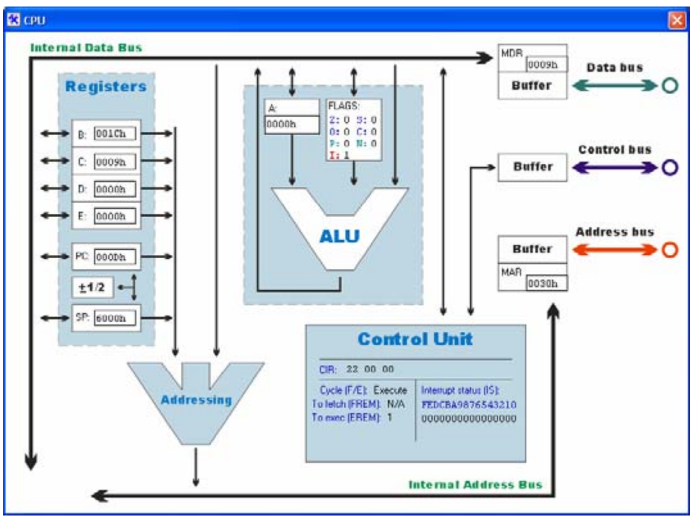
36.3. Viewing execution

If you want to see exactly how your program is executed, you will need to use the CPU window, which will show you the structure of the CPU, its current state (such as Fetching next instruction) and the values of all relevant registers.

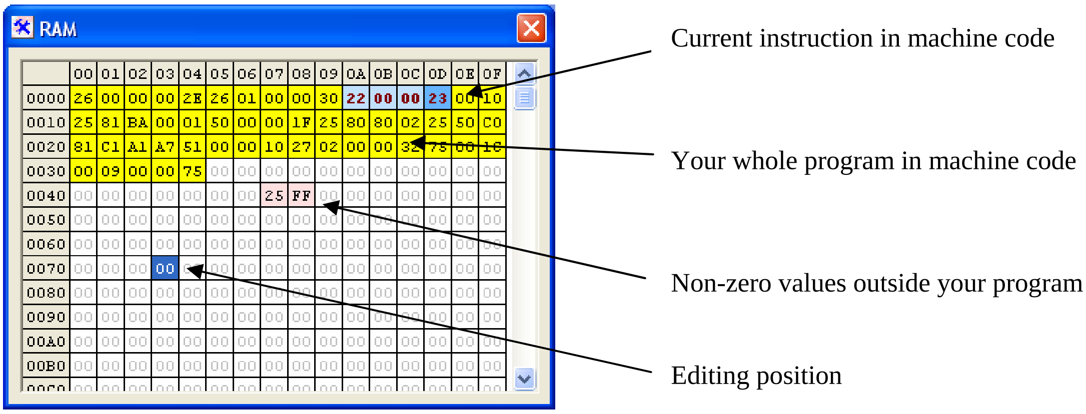
This is the CPU screen in Basic complexity mode:



It has a lot of components missing – they are not displayed for simplicity. When you run your program, you can see the values in all registers. In A-level mode you will be able to see the contents of the Control Unit, and the MAR/MDR registers:

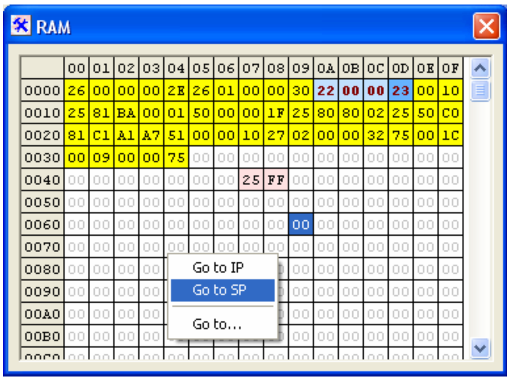


Another tool which can help you with understanding the way programs are executed is the RAM window. This window will show you the machine code associated with your program, as well as program data and stack:



You can easily edit memory – just point the Editing position (shown above) at the required cell and type in a new value. **Be very careful with editing executable code** – your program will most probably generate an error, and there is a possibility of CLab crashing.

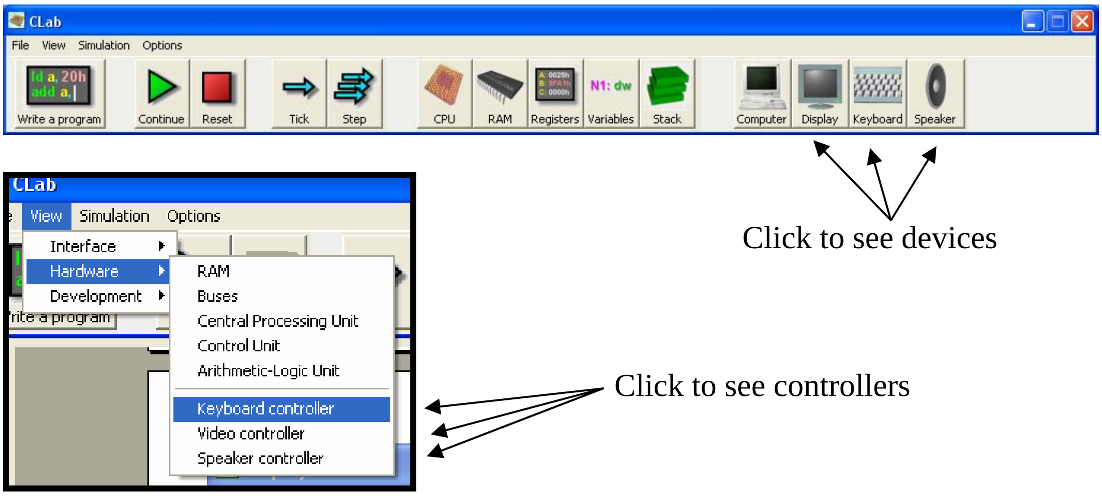
If you right-click in the RAM window, you will see a pop-up menu which will give you quick access to any area of memory:



37. Using devices

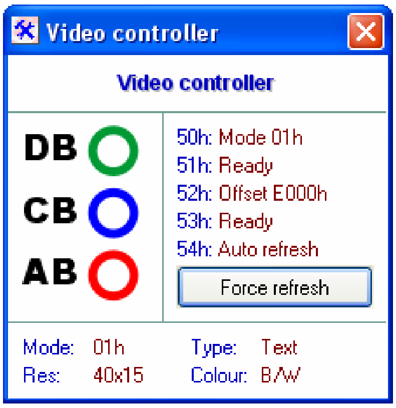
There are three devices in CLab: Display, Keyboard and Speaker. Each of them has a controller. This section will tell you how to write code for each of the devices.

You can access devices by clicking on respective icons on the Main window. You can access their controllers only via the menu – go to View/Hardware:



37.1. Video controller and display

Examples of the Display and Video controller windows are shown below:



You don't need to know everything about the video controller – most of its functions are provided for students who are insterested in how the video system works in computer. You will never need to use video controller window if you simply want to print something on the screen.

Everything that you see on the Display is stored in a special area of RAM, called the *video memory*. It is possible to tell the controller how to interpret information stored in that area – text or graphics, or how many colours there are – by switching *video* *modes*. There are two basic modes you need to be aware of – please consult your teacher if you are insterested in more. They are:

- Monochrome text, 40x15 characters, mode number **1**.
- 24 bit color graphics, 42x32 pixels, mode number **7**.

Prior to doing anything with the screen you will need to switch to the required mode. Type the following line in your program:

```
out 50h, 2    ; To switch to color text mode
```

or

```
out 50h, 7    ; To switch to color graphics mode.
```

Now you can print text or draw!

To print text

Write your characters to address 0E000h and onwards. To calculate the exact address, multiply your Y-coordinate by 40 and add your X coordinate to that. Add the whole thing to 0E000h and print!

Alternatively, you can use the Print routine from the LiBRARY which is supplied with CLab (see Testing for full source code). The Print routine will print out any string you ask it to – just make sure there is a zero byte at the end of your string. Below is all the code needed to print out MYSTRING

```
ld a, offset(MYSTRING)
call Print
...
halt

MYSTRING: ds “Hello, world!”
          dw 0
```

To draw

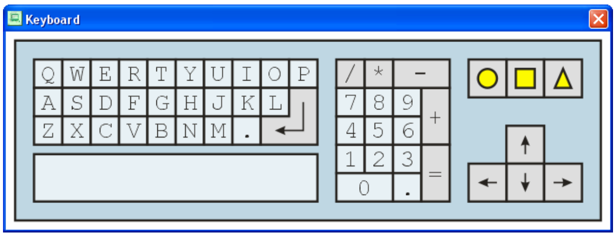
Again, the same idea. Just load your three colour bytes into positions calculated with the following formula:

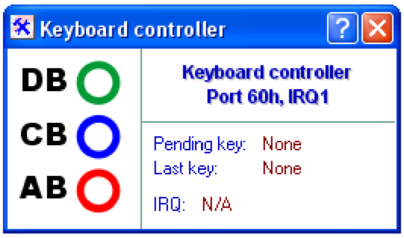
```
address = 0E000h + (y*42 + x)*3
```

The last multiplication by three is needed because every pixel takes up three bytes.

37.2. Keyboard and Keyboard controller

The keyboard and its controller will let you interact with the program you are running. Keyboard windows look like this:





Whenever you click a key on the keyboard, the keyboard controller will request interrupt number 1 from the CPU. If the CPU accepts it, it will invoke the keyboard interrupt service procedure (ISP). So if you want to receive keys pressed on the keyboard, you will need to write an ISP.

Your ISP should be like a normal procedure with the exception that it should end with an `iret` instruction, not `ret`. Also, you *absolutely have to preserve all registers*, except for `PC` and `FLAGS`, because your ISP can be invoked at any point in your code.

Below is an example of an ISP:

```
isp_keyboard:
      push a      ;Preserve A and B – we don’t use other registers
      push b

      in a, 60h   ;Get the key into A
      ld b, a     ;Store the key in B

      pop b       ;Restore registers
      pop a

      iret        ;Finished
```

OK, it doesn’t do anything useful, but it is simple enough for you to get the idea. You may be wondering how to do anything useful with an ISP if it is not allowed to modify *any* registers. Well, there is another means of communication with external world, and this is how it’s done in real PCs. Your ISP could store the keys in a special memory area (which is always fixed in DOS). Then any program which needs to receive keyboard input will simply examine that memory area. A more advanced operating system such as Windows will provide the programmer with a special function which will return if any keys have been pressed.

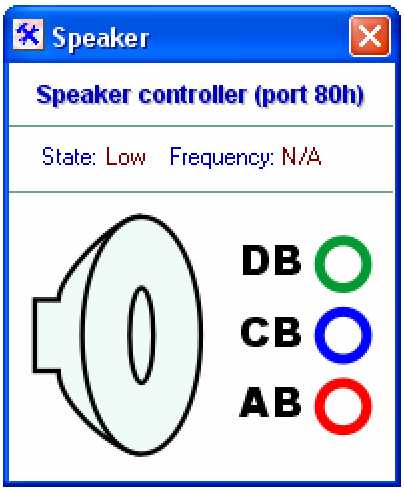
Now you need to install the ISP – that is, tell the CPU which ISP to start when it executes interrupt 1. Just write the following code at the beginning of your program – you need not know how it works:

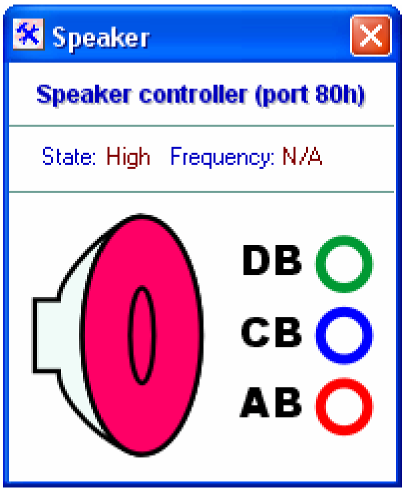
```
      ld a,offset(isp_keyboard)
      ld [0FF02h],a
```

37.3. Speaker and Speaker controller

Speaker is the simplest device in CLab, but at the same time it’s least useful. All you can do is set it to high or low state, or tell it to oscillate at a given frequency. The main purpose of this device is to show students how to use I/O ports on a very simple example. But of course you can use this device as a flag or some sort of an indicator.

The speaker window is combined with speaker controller window:





Whenever the speaker is set to Low, it will look like in the left window above. High setting will be shown as in the right window above.

To change the speaker state, write 0 or 1 to port 80h:

```
out 80h, 1      ;Set speaker high
```

To set a frequency, write any other number to port 80h:

```
out 80h, 0F500h     ;Set speaker frequency to 9.57 Hz
```

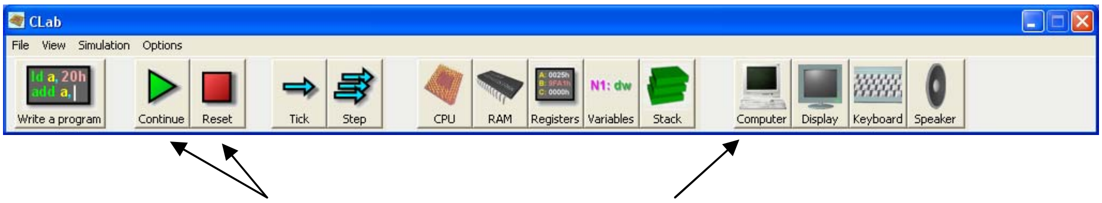
To calculate speaker frequency, use the following relationship:

frequency = 20 / 65536 * byte

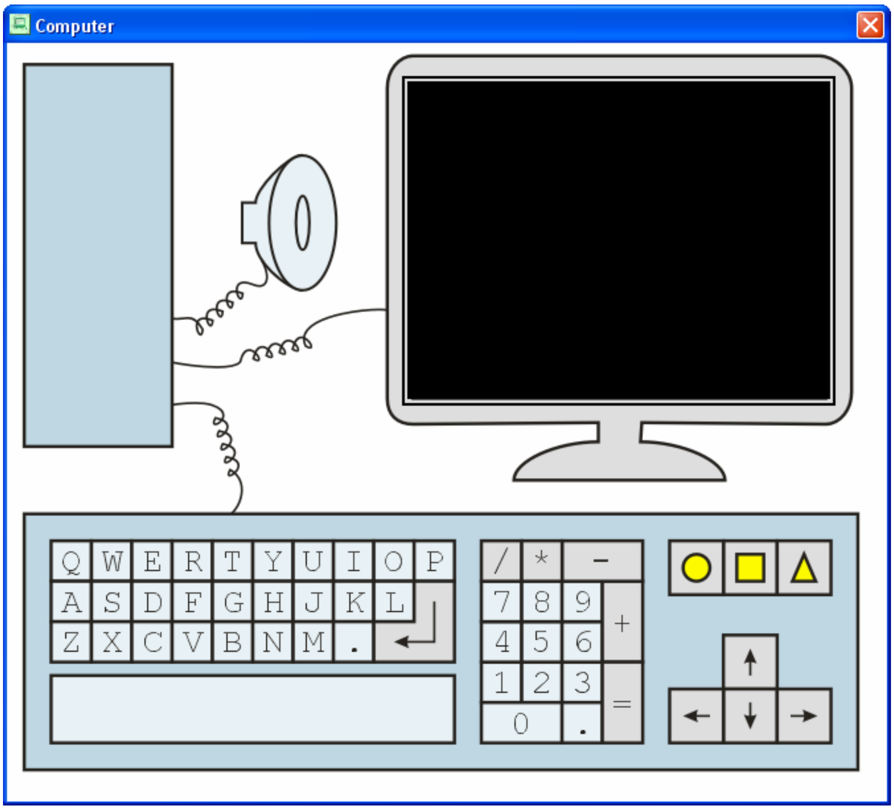
where *byte* is the byte you wrote to port 80h.

38. Testing your program

If you have an interactive end-user ready program, you can test it as if it was running on a real PC. Just use the *Computer window* that pops up when you first run CLab. You can open the window with the Computer button on the Main window:



The computer window looks like this:



To run a program, just make sure you have the code in the Code editor and use the Start/Reset buttons in the Main window to control the execution of the program.

39. Assembly language manual

This section will help you learn how to write programs for this particular version of CLab. Please note that I will only describe the syntax and the most useful instructions. For a full set of instructions, please see the (Design.Assembly) section. That section also contains a more formal definition of assembly language syntax.

Statements

Every line of the code you write is called a *statement*. You have several options as to what you write on a given line. You can write one of the following:

- an empty line
- a line of code
- a variable declaration

Apart from that, every line can start with a *label* declaration and end with a *comment*.

Comments

Comments should start with a semicolon. Everything after the first semicolon to the end of the line will be completely ignored by CLab.

Labels

A *label* helps you define a specific position in your code. If a label precedes an instruction, you will be able to use that label name with a *jump* or a *call* instruction. If it precedes a variable declaration, it will define the name of the variable.

Label names have to start with a letter and contain only letters, numbers and underscores. Every label must end with a colon followed by **at least one space or tab character**, unless the line ends with the label. For example:

```
number: dw 0        ;Label “number” is a variable name; note one space after :
next:   ld a,0      ;Label “next” points to code; note three spaces after :
calculate:          ;Label points to whatever code follows; note no spaces or tabs
```

Variables

To declare a variable, use a label together with either `dw` (declare word) or `ds` (declare string) to declare, respectively, a word or a string variable. `dw` must be followed by a numerical constant between -32768 and 65536. If you want to use hex or binary, add an “h” or a “b” to the end of the number. For example:

```
myvar1:  dw   25        ;Declare myvar1 containing 25 (decimal)
myvar2:  dw   0F00h     ;Declare myvar2 containing 0F00 (hexadecimal)
myvar3:  dw   10000b    ;Declare myvar3 containing 10000 (binary)
```

If you declare a string variable, you must add your string after `ds`, enclosed with double quotes:

```
mystring:  ds   “Hello world”  ;Declare a string containing “Hello world”.
```

You can also initialise variables with addresses of other variables. For example,

```
addr_myvar:  dw  offset(myvar)
```

See Immediate operands below for more information about *offset*.

Code lines

Every line of code must contain operation code (opcode). Most useful opcodes are listed in the (instructions) section below. It can also contain operands, as many as a given opcode requires, separated by a comma.

Operands can be of three different types. These are register, immediate and memory operands.

Register operands

If an instruction requires a register operand, you can specify `A`, `B`, `C`, `D` or `E`. Sometimes opcodes require specific registers – some will ask for `A` only, and some for anything except for `A`. Registers can also appear as part of memory operands.

Immediate operands

Immediate operands are numerical constants. If you want to, say, load zero into a register, then zero will be an immediate operand. Different opcodes let you specify different range of constants. You have the option to specify the number in any of three bases – two, ten or sixteen. Binary numbers have to end with a ‘b’ letter. Hex numbers have to begin with a digit and end with an ‘h’ letter. For example:

```
10100       ;valid, interpreted as 10,100
10100b      ;valid; interpreted as 20
10102b      ;invalid

0F00        ;invalid
0F00h       ;valid; interpreted as 3840
F000h       ;valid; interpreted as variable name
0XG0h       ;invalid
```

All numerical constants can be prefixed with a minus sign to get the negative number. Numerical constants can also appear as part of memory operands.

Whenever you can specify a numerical constant, you can also use the *offset* macro to specify the address of a variable. Just write, `offset(varname)` instead of the constant, and it will be replaced with `varname`’s address. In the following example, the two lines are equivalent, assuming that `myvar` is declared at address 100h.

```
ld a, 100h
ld a, offset(myvar)
```

Memory operands

Whenever you need to address a cell in memory (e.g. when you need to get the value of a variable), you will need to use memory operands. There are four different types of memory operands.

Direct memory operands let you specify the address of the memory cell directly. You should write the memory address as a numerical constant, enclosed with square brackets. You can also simply write variable name. For instance:

```
ld a, [0100h]      ;Loads contents of memory cell 100h into a
ld a, myvar        ;Loads contents of variable myvar into a
```

Indirect register memory operands let you address memory cell at address held in a register. You can use `B`, `C`, `D` or `E` registers. Register name should be enclosed with square brackets. For instance:

```
ld a, [b]    ;Loads contents of memory cell at address held in b into a
```

Indirect immediate memory operands let you address a memory cell whose address is stored in another memory cell. You can either specify a numerical constant enclosed with two pairs of square brackets or a variable name enclosed with square brackets. For instance:

```
ld a, [[0100h]]    ;Load contents of memory cell at address specified in
                   memory cell at address 100h into a.
ld a, [myvar]      ;Load contents of memory cell at address held in myvar
                   into a.
```

Indexed memory operands let you specify an expression to calculate the address. You would usually use this addressing to access arrays of data. You can specify where the array begins (base address), element number (index), element size (multiplier) and an optional offset constant. For example, if you need to get element number three from array of words starting at `myarr` you will use the following code:

```
ld b, offset(myarr)    ;only B register can hold base address
ld c, 3                ;element number
→ ld a, [b+c*2]        ;load element data into a – word is two bytes long
```

It is very easy to loop through arrays with this kind of addressing – just load the offset of your array into b before the loop, and then do a loop on a register. Use that register in your indexed addressing.

Instead of using the base address, you could have added the address of myarr as a numerical constant:

```
ld a, [c*2+offset(myarr)]
```

Please note that the only required parameter is the register that you index on. Everything else is optional. Also be aware that the order in which you specify parameters is crucial, and there must not be a single space in the whole operand. And don’t forget to enclose it with square brackets.

40. Instructions

This section will tell you about the instructions available to you, and how to use them. Instructions are sorted in the order of how often you may need them, and grouped by similarity.

40.1. Data movement

`ld dest, src`

Loads value in `src` into `dest`. You can use `ld` to load:

- register into register `ld a, b`
- variable into register `ld a, myvar`
- constant into register `ld a, 20h`
- register into variable `ld myvar, a`

Please note that you **cannot** load a constant into a variable. Load the constant into a register first, and then load the register into your variable.

`st src, dest`

Stores value in `src` in `dest`. You can use `st` to store:

- register in register `st b, a`
- variable in register `st myvar, a`
- constant in register `st 20h, a`
- register in variable `st a, myvar`

Please note that you **cannot** store a constant in a variable. Store the constant in a register first, and then store the register in your variable.

40.2. Basic arithmetic

`add dest, src`

Performs the following mathematical operation: `dest ← dest + src`
You can use `add` to add:

- constant to accumulator `add a, 20h`
- variable to accumulator `add a, myvar`
- register to accumulator `add a, c`
- accumulator to register `add c, a`

Note that you **cannot** add two numbers if neither of them is stored in the accumulator. Load one of the numbers into the accumulator first.

`sub dest, src`

Performs the following mathematical operation: `dest ← dest - src`
You can use `sub` to subtract:

- constant from accumulator `sub a, 20h`
- variable from accumulator `sub a, myvar`
- register from accumulator `sub a, c`
- accumulator from register `sub c, a`

Note that you **cannot** subtract two numbers if neither of them is stored in the accumulator. Load one of the numbers into the accumulator first.

`mul dest, src`

Performs the following mathematical operation: `dest ← dest * src`
You can use `mul` to multiply:

- constant by accumulator `mul a, 20h`
- variable by accumulator `mul a, myvar`
- register by accumulator `mul a, c`
- accumulator by register `mul c, a`

Note that you **cannot** multiply two numbers if neither of them is stored in the accumulator. Load one of the numbers into the accumulator first. `Mul` does not take sign into account – use `imul` if you want to multiply signed numbers. `Imul` is the same as `mul` in all other respects.

`div dest, src`

Performs the following mathematical operation: `dest ← dest / src`
You can use `div` to divide:

- constant by accumulator `div a, 20h`
- variable by accumulator `div a, myvar`
- register by accumulator `div a, c`
- accumulator by register `div c, a`

Note that you **cannot** divide two numbers if neither of them is stored in the accumulator. Load one of the numbers into the accumulator first. `Div` does not take sign into account – use `idiv` if you want to divide signed numbers. `Idiv` is the same as `div` in all other respects.

40.3. Conditional and unconditional branching

`cmp left, right`

Compares `left` and `right` and sets the flags in such a way that a consequtive call to one of the conditional branching instructions will branch according to its name. For example, if `left` is less than `right` then `jl` (jump if less) will do a jump. You can compare the following numbers:

- accumulator with constant `cmp a, 20h`
- accumulator with variable `cmp a, myvar`
- accumulator with register `cmp a, c`
- register with accumulator `cmp c, a`

Please note that you **cannot** compare a register other than the accumulator with a constant. Load the constant into a register and then compare.

`jg label`

Checks the `FLAGS` register and either performs a jump to `label` or doesn't. If you use the `cmp` instruction before this one, `jg` will perform a jump if the `left` number was greater than the `right` number. `Label` should be a label name which you declared somewhere in your code.

jl label

Checks the `FLAGS` register and either performs a jump to `label` or doesn't. If you use the `cmp` instruction before this one, `jl` will perform a jump if the `left` number was less than the `right` number. `Label` should be a label name which you declared somewhere in your code.

jge label

Checks the `FLAGS` register and either performs a jump to `label` or doesn't. If you use the `cmp` instruction before this one, `jge` will perform a jump if the `left` number was greater than or equal to the `right` number. `Label` should be a label name which you declared somewhere in your code.

jle label

Checks the `FLAGS` register and either performs a jump to `label` or doesn't. If you use the `cmp` instruction before this one, `jle` will perform a jump if the `left` number was less than or equal to the `right` number. `Label` should be a label name which you declared somewhere in your code.

jz label

Checks the `FLAGS` register and either performs a jump to `label` or doesn't. If you use the `cmp` instruction before this one, `jz` will perform a jump if the `left` number was equal to the `right` number. `Label` should be a label name which you declared somewhere in your code.

jnz label

Checks the `FLAGS` register and either performs a jump to `label` or doesn't. If you use the `cmp` instruction before this one, `jnz` will perform a jump if the `left` number was not equal to the `right` number. `Label` should be a label name which you declared somewhere in your code.

jmp label

This will always jump to `label`. `Label` should be a label name which you declared somewhere in your code.

40.4. Procedures and stack

call label

Calls procedure starting at `label`. `Label` must be a label name declared somewhere in your code. When the procedure ends (with a `ret` instruction), your program will continue execution right after the `call` instruction.

ret

Returns from a procedure call initiated by `call` instruction. You should end all your procedures with this instruction. Make sure that stack is the same as when your procedure started before calling `ret` – otherwise you will get unpredictable results.

push src

Pushes value stored in `src` onto stack. You can push registers and constants only.

pop dest

Pops (pulls) a value from stack and stores it in `dest`. You can only pop into registers.

40.5. More arithmetic

neg src & not src

`Neg` changes the sign of value in `src`. `Not` inverts all bits in `src`, so that all 1's become 0's and vice versa. In both cases `src` must be a register.

and dest, src & or dest, src

`And` performs a bitwise *and* operation between `dest` and `src` and stores the result in `dest`. `Or` performs a bitwise *or* operation in the same way. You can `and` / `or` the following numbers:

- Accumulator and constant `and a, 20h`
- Accumulator and variable `and a, myvar`
- Accumulator and register `and a, c`
- Register and accumulator `and c, a`

The truth tables for `and` and `or` are as follows:

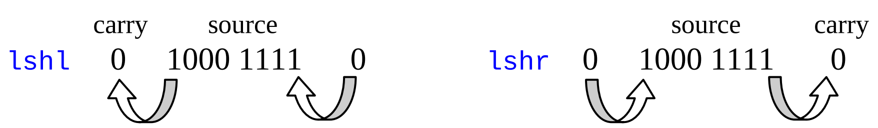
v1	v2	and v1, v2	or v1, v2
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Shifts

There are two types of shifts you need to be aware of – arithmetic and logical. The difference is that arithmetic shifts take sign into account (they preserve it if possible), whereas logical don't. In CLab, you can shift to the left or to the right up to 15 bits at a time, using either shift type.

Logical shifts: `lshl` (logical shift left) and `lshr` (lgical shift right)
Arithmetic shifts: `ashl` (arithmetic shift left) and `ashr` (arithmetic shift right)

Each of these operations takes two operands – `arg` and `num`. The operations will shift the value in `arg` by `num` bits and store it back to `arg`. You can shift the accumulator by a number of bits specified in a non-accumulator register, or a non-accumulator register by a constant number of bits. This is how the shifts are performed:



40.6. Flow control and I/O

halt

This instruction will stop the execution of your program. Use it when you want your program to terminate. You have to use `halt` when you have something written after the point where you want your program to end. Consider the following example:

```
      ld a, myvar
      ld b, a
                        ← This is where the program should stop
myvar: dw 5
```

The program won't stop at that point because, as you may remember from your A-level course there is no way to tell which machine codes mean code and which – data. So CLab will try to execute whatever is represented by the variable 5. You don't want this to happen, so you need to tell CLab explicitly to stop. That's what `halt` is for.

cli

This instruction prevents the CPU from accepting any interrupts. Why you'd want to do that is beyond the scope of A-level computing, but you may definitely write some code to check how it works. `Cli` stands for "clear interrupts".

sti

This instruction enables interrupts after they were disabled with `cli`. `Sti` stands for "set interrupts".

in dest, src

Reads a word from port number `src` and stores it in `dest`. You can `in` into registers, specifying port number as a register or a constant. The following are both valid:

```
      in a, c
      in a, 50h
```

out dest, src

Writes a word `src` to port number `dest`. You can specify port number and data, respectively, as:

- register, register `out a, c`
- register, constant `out a, 20`
- constant, register `out 50h, a`
- constant, constant `out 50h, 5`

41. Error codes

When you try to run your program, you may get an error message. Every message has a short explanation with it and an error code. But the short explanation may be very confusing and not always very helpful. In this section I will try to help you find out what the problem is.

`EC1002` – Invalid token combination: X and Y.
You are trying to put together operands, opcodes or variable declarations in a way that is fundamentally wrong. For instance, you may be trying to put variable declaration as if it was an operand, or specify an opcode after its operand.

`EC1003` – Invalid token combination: X, Y and Z.
You are trying to put together operands, opcodes or variable declarations in a way that is fundamentally wrong. For instance, you may be trying to put variable declaration as if it was an operand, or specify an opcode after its operand.

`EC1004` – A line cannot contain more than three tokens. This line contains X tokens.
There is something wrong with the syntax you are using. It should never happen that a line has more than three distinct part to it. There are no opcodes which require more than two operands, for instance. Make sure you don't have too many spaces or tabs where they shouldn't be, especially in memory operands.

`EC2001` – Opcode takes 0 operands, not X.
The opcode you are using requires no operands. You have specified at least one operand. Make sure there is nothing (except for comment if you need one) after the opcode name.

`EC2002` – Variable initialisation sequence is neither '?' nor a valid constant.
When you declare a variable, there is what is called "variable initialisation sequence" after the `dw` or `ds` keyword. `dw` requires this sequence to be a valid numerical constant. `ds` requires it to be a string enclosed with double quotes. Often students forget they have to initialise a variable when they declare it. If you have indeed initialised it, check if your number is valid, or your string has the closing double quote.

`EC2005` – Syntax error in operand OR opcode and operand incompatible. Offending operand: X.
This error happens in many different contexts. What it basically means is that either one of the operands in completely invalid (e.g. you specify 0Xh as a numerical constant), or that the opcode you are using doesn't like the operand types you have specified. The latter is most often the reason for the error. Consider the following case:

```
test c, 5
```

You will get EC2005 error because `test` requires one of the operands to be the accumulator.

I have tried to clarify the reason for some of the more frequent errors of this type. For instance, if you want to load a numerical constant into a variable, you would get this error because `ld` doesn't allow for operand types memory/immediate. But I trap that as a special case and tell you exactly what the problem is. However, it is very difficult to trap all cases and explain what exactly is wrong. So the best piece of advice I can give you if you get this error is that you should carefully check which operands you can use with your opcode that gives you the error.

EC2006 – The number of shift cycles must be between 0 and 15.
You are using one of the shift instructions and ask it to perform more than fifteen shifts. The maximum number of shifts allowed is 15. You would not need any more because sixteen shifts is the same as no shifts (for cyclic shifts), and seventeen is the same as 1 shift.

EC2007 – 16 bit immediate constant is out of range.
Whenever you specify an immediate constant, CLab checks whether it is in the allowed range (-32768 to 65536, or -8000h to 0FFFFh). If it is not, you will get this error message.

EC2008 – Invalid label name: X.
Label names have to start with a letter and can only contain letters, numbers or underscores. If your label name contains anything else, you will get this error.

EC2009 – Label name cannot be same as register name.
You are trying to declare a label named A, B, C, D or E. That is not allowed.

EC2010 – Opcode not recognized: X. Check spelling.
The compiler sees that you are trying to specify an instruction, but it cannot understand the opcode you use. Usually this means you have misspelled the opcode. For example, `xhcg a,b` will generate this error.

EC2011 – Memory addressing scaling factor should be 0, 1, 2 or 4.
You use indexed memory addressing and try to multiply the index register by a number which is not 0, 1, 2 nor 4. For example, `[b+c*3]` will produce this error.

EC2013 – Cannot load into a constant (first operand cannot be a constant).
You use `ld` instruction with the first operand being a constant. For example, `ld 5,a` is erroneous. You probably meant `ld a,5` or `st 5,a`.

EC2014 – Cannot load a constant into a memory cell directly.
You try to load a constant into a variable or a memory cell in one go. This is not allowed. You must first load the constant into a register, and then load that register into the variable/memory cell.

EC2015 – Cannot store in a constant (second operand cannot be a constant).
You use `st` instruction with the second operand being a constant. For example, `st a,5` is erroneous. You probably meant `ld a,5` or `st 5,a`.

EC2016 – Cannot store a constant in a memory cell directly.
You try to store a constant in a variable or a memory cell in one go. This is not allowed. You must first store the constant in a register, and then store that register in the variable/memory cell.

EC2017 – Operand for INT must be an 8 bit immediate constant.
Interrupt numbers have to be between 0 and 255. Check what interrupt number you have specified. You are not allowed to use registers.

EC2018 – Port address must be an 8 bit immediate constant (0 to 255).
Port addresses have to be between 0 and 255. Check what port address you have specified.

EC2019 – DS variable should be initialised with either ? or a string literal enclosed with " ".
When you declare a variable, there is what is called "variable initialisation sequence" after the `ds` keyword. `ds` requires it to be a string enclosed with double quotes. Often students forget they have to initialise a variable when they declare it. If you have indeed initialised it, check if your string has the closing double quote.

EC3001 – Label already declared: X. Previous declaration on line Y.
A label with a given name can only be declared once. You have declared a label with the same name somewhere else in your code. It sometimes happens that students use the same label name for a procedure and for a variable. That is not allowed – *all* label names have to be unique.

EC3002 – Undeclared reference: X.
You are referring to a label or a variable name which is not declared in your code. Check if you have spelled the name correcly. Also check if you have declared the variables you are referring to. See Variables above for more information on how to do that.

User manual

Teacher's extras

42. Introduction

While writing user manual, I realised that teachers would mostly need the same information as A-level students. So I will not copy and paste the whole A-level manual here – this section will contain all the extra information that a teacher may need.

43. Installation

The system is distributed as a single executable file called `setup.exe`. Run that file. Follow the instructions that you see on the screen. The installation program, created with a freeware installer called Instyler, will prompt you for program path – change it if you need to. Then click Install button.

After the installation, the program may ask you if you would like to restart. If it does, click Yes.

The installation program will place a shortcut to CLab executable in the start menu and on your desktop.

44. Devices

44.1. Video controller

The image displayed on screen is stored in RAM at a given memory address. You can find out that address or set it to something else using port 52h. See below for more detail.

Screen modes

`01h`: Monochrome text; 1 byte per char; 40x15 characters
Every byte represents one character's ASCII code.

`02h`: Color text; 2 bytes per char; 40x15 characters
Every two bytes represent one character's ASCII code and color. The first byte in the pair is the character's ASCII code, the second one – its colour. The color byte format is: `LRGB lrgb`, where `R`, `G` and `B` are Red, Green and Blue components respectively, `L` is a brightness bit, uppercase means background color, lowercase – text color.

`03h`: Monochrome graphics; 1 bit per pixel; 208x156 pixels
Every byte describes eight pixels. If a bit is set, color seen will be white; otherwise – black.

`04h`: 16 color graphics; 4 bits per pixel; 104x78 pixels
Every byte describes two pixels. The format is: `LRGB`, where `R`, `G` and `B` are Red, Green and Blue components respectively, `L` is a brightness bit.

`05h`: 256 color graphics; 8 bits per pixel; 74x55 pixels; paletted

Every byte describes one pixel. The color that is seen on screen will be taken from a palette array inside the video controller memory which is 256x3 bytes long. That is the palette memory, which stores three bytes (RGB) for every color in this mode.

`07h`: 16M color graphics; 24 bits per pixel; 42x32 pixels
Every three bytes describe one pixel. The format is, `RGB` where `R`, `G` and `B` are *bytes* describing respective colors.

Input/Output ports

Screen mode port `50h`

Writing screen mode number to this port will cause the video controller to switch screen modes. If it receives any other word apart from valid screen mode numbers, it will ignore it. The changes will be reflected immediately, even in manual refresh mode.

Reading from this port will cause the video controller to return its current screen mode.

Palette port `51h`

To change an entry in the palette array, programs will write two words to this port. The first one will contain palette entry number in the low-order byte and the red component in the high-order byte. The second word will contain green and blue components in low- and high-order bytes respectively. Note that once sent to the video controller, palette cannot be read from it. Also, palette only influences images in screen mode `05h`.

Memory port `52h`

Writing to this port will change the offset to video memory buffer in RAM. The changes will be reflected immediately. That is, even in manual refresh mode the screen will be updated to reflect changes to video memory.

If a program reads from this port, it will receive current pointer to video memory.

Refresh port `54h`

Writing 0 to this port will disable auto screen refresh, so changes to video RAM will only be reflected when the programmer wants to. Writing 1 will enable auto screen refresh, so the screen will be updated every once in a while. Writing anything else will force the screen to be refreshed.

Reading from this port will return either 0 or 1 to indicate the state of auto refresh.

44.2. Keyboard controller scancodes

Scancode	Hex	Key
0	00	A
1	01	B
2	02	C
3	03	D
4	04	E
5	05	F
6	06	G
7	07	H
8	08	I
9	09	J
10	0A	K
11	0B	L
12	0C	M
13	0D	N
14	0E	O
15	0F	P
16	10	Q
17	11	R

Scancode	Hex	Key
18	12	S
19	13	T
20	14	U
21	15	V
22	16	W
23	17	X
24	18	Y
25	19	Z
26	1A	.
27	1B	Enter
28	1C	Spacebar
29	1D	=
30	1E	0
31	1F	1
32	20	2
33	21	3
34	22	4
35	23	5

Scancode	Hex	Key
36	24	6
37	25	7
38	26	8
39	27	9
40	28	Numpad .
41	29	/
42	2A	*
43	2B	-
44	2C	+
45	2D	Left
46	2E	Right
47	2F	Up
48	30	Down
49	31	Circle
50	32	Square
51	33	Triangle

44.3. Complete instruction manual

This section will list all instructions there are in CLab, with a very short summary on what the instruction does and its syntax.

Operand types

- `M` memory (any addressing mode),
- `R` register (`A`, `B`, `C`, `D` or `E`),
- `Rn` register (`B`, `C`, `D` or `E`),
- `A` accumulator,
- `I` 16-bit immediate,
- `I8` 8-bit immediate,
- `N` immediate as part of the machine code.

44.3.1. Data movement

These instructions move data between registers and memory. They also include stack operations. None of these modify the `FLAGS` register.

Name	Arguments	Description
`ld`	dest, src	Copies contents of `src` to `dest`. Allowed `dest`/`src` combinations: `R/R`, `R/M`, `R/I`, `M/R`.
`st`	src, dest	Copies contents of `src` to `dest`. Allowed `src`/`dest` combinations: `R/R`, `M/R`, `I/R`, `R/M`.
`push`	src	Copies contents of `src` to `[SP]`, then increments `SP` by 2. `Src` is type `R` or `I`.
`pop`	dest	Decrements `SP` by 2, then copies contents of `[SP]` to `dest`. `Dest` is type `R`.
`pushpc`	-	Pushes `PC` register onto stack, pointing to **after** the `pushpc` instruction.
`pushsp`	-	Pushes `SP` register onto stack. `SP` value before this operation is pushed.
`pushfl`	-	Pushes `FLAGS` register onto stack.
`popsp`	-	Pops `SP` register from stack.
`popfl`	-	Pops `FLAGS` register from stack.
`sp2b`	-	Copies the contents of `SP` register into `B` register. Used to access parameters that are passed on stack quickly.

lea	dest, src	Load effective address. Allowed dest/src combinations: Rn/M. Loads the address calculated for src into register dest.
xchg	r1, r2	Swaps values in registers r1 and r2 so that value in r1 goes to r2 and vice versa. r1 and r2 are type Rgn.

44.3.2. Arithmetic

These instructions do addition, subtraction, multiplication etc. All of these set the FLAGS register (flags z, s, o, c; n, p) according with the result.

Name	Arguments	Description
add	addto, addwhat	Adds addwhat to addto, saves result in addto. Allowed addto/addwhat combinations: A/I, A/M, A/R, R/A.
sub	subfrom, subwhat	Subtracts subwhat from subfrom, saves result in subfrom. Allowed subfrom/subwhat combinations: A/I, A/M, A/R, R/A.
adc	addto, addwhat	Adds addwhat, addto and carry, saves result in addto. Allowed addto/addwhat combinations: A/I, A/M, A/R, R/A.
sbb	subfrom, subwhat	Subtracts subwhat from subfrom, then subtracts carry from the result, saves final result in subfrom. Allowed subfrom/subwhat combinations: A/I, A/M, A/R, R/A.
cmp	left, right	Compares left with right and sets flags so that conditional jumps work correctly. E.g. if left<right then JL will do a jump. The opeartion subtracts right from left and discards the result. Allowed left/right combinations: A/I, A/M, A/R, R/A.
mul	arg1, arg2	Multiplies arg1 by arg2. Saves result in arg1. Treats values as unsigned integers. Allowed arg1/arg2 combinations: A/I, A/M, A/R, R/A.
div	num, denom	Divides num by denom, saves the integer part of the result in num. Interprets num and denom as unsigned integers. Allowed num/denom combinations: A/I, A/M, A/R, R/A.
imul	arg1, arg2	Multiplies arg1 by arg2. Saves result in arg1. Treats values as signed integers. Allowed arg1/arg2 combinations: A/I, A/M, A/R, R/A.
idiv	num, denom	Divides num by denom, saves the integer part of the result in num. Interprets num and denom as signed integers. Allowed num/denom combinations: A/I, A/M, A/R, R/A.
mod	num, denom	Divides num by denom, saves the remainder part of the result in num. Interprets num and denom as unsigned integers. Allowed num/denom combinations: A/I, A/M, A/R, R/A.
inc	arg	Increments arg, that is adds 1 to it. Arg is type R.
dec	arg	Decrements arg, that is subtracts 1 from it. Arg is type R.
neg	arg	Reverses the sign of arg. This is equivalent to not arg; inc arg; but occupies only one byte. Arg is type R.

44.3.3. Bitwise

Bitwise operations such as AND, OR, shifts, etc. All of them modify the FLAGS register (flags z, s, c; n, p) according with the result.

Name	Arguments	Description
not	arg	Bitwise NOT – inverts all bits in arg. Arg is type R.
and	arg1, arg2	Bitwise AND. Allowed arg1/arg2 combinations: A/I, A/M, A/R, R/A.
or	arg1, arg2	Bitwise OR. Allowed arg1/arg2 combinations: A/I, A/M, A/R, R/A.
xor	arg1, arg2	Bitwise XOR. Allowed arg1/arg2 combinations: A/I, A/M, A/R, R/A.

`test`	left, right	Performs a bitwise AND operation on `left` and `right` and sets the flags according with the result. Result itself is discarded. `A/I`, `A/M`, `A/R`, `R/A`.
`lshr`	arg, num	Shifts[3] bits in `arg` by `num` to the right. Low-order bit goes to carry, high-order bit becomes zero. Allowed `arg`/`num` combinations: `A/Rn`, `Rn/N`.
`lshl`	arg, num	Shifts[3] bits in `arg` by `num` to the left. High-order bit goes to carry, low-order bit becomes zero. Allowed `arg`/`num` combinations: `A/Rn`, `Rn/N`.
`ashr`	arg, num	Shifts[3] bits in `arg` by `num` to the right. Low-order bit goes to carry, high-order bit stays the same. Allowed `arg`/`num` combinations: `A/Rn`, `Rn/N`.
`ashl`	arg, num	Entirely equivalent to `lshl`.
`ror`	arg, num	Rotates[3] bits in `arg` by `num` to the left. Low-order bit goes to high-order bit and carry. Allowed `arg`/`num` combinations: `A/Rn`, `Rn/N`.
`rol`	arg, num	Rotates[3] bits in `arg` by `num` to the left. High-order bit goes to low-order bit and carry. Allowed `arg`/`num` combinations: `A/Rn`, `Rn/N`.
`rcr`	arg, num	Rotates[3] bits in `arg` by `num` to the left through carry. Carry goes to high-order bit and low-order bit goes to carry. Allowed `arg`/`num` combinations: `A/Rn`, `Rn/N`.
`rcl`	arg, num	Rotates[3] bits in `arg` by `num` to the left through carry. Carry goes to low-order bit and high-order bit goes to carry. Allowed `arg`/`num` combinations: `A/Rn`, `Rn/N`.
`bswp`	src	Swaps bytes in `src` so that the high-order byte becomes the low-order byte and vice versa. `Src` is type `R`.

44.3.4. Flags

These operations are used to modify the `FLAGS` register.

Name	Arguments	Description
`stz`	-	Sets zero flag.
`clz`	-	Clears zero flag.
`stc`	-	Sets carry flag.
`clc`	-	Clears carry flag.
`sto`	-	Sets overflow flag.
`clo`	-	Clears overflow flag.
`sts`	-	Sets sign flag.
`cls`	-	Clears sign flag.
`sti`	-	Sets interrupt flag.
`cli`	-	Clears interrupt flag.

44.3.5. Branching

These are all operations that change execution order. They change `IP` register (and `CS` where applicable), so the next instruction to be executed changes as well.

Name	Arguments	Description
`jg`, `jnle`	addr	Jumps to `addr` if z = 0 and s = o. `Addr` is an absolute address of type `M`.
`jl`, `jnge`	addr	Jumps to `addr` if `s` <> `o`. `Addr` is an absolute address of type `M`.
`jge`, `jnl`	addr	Jumps to `addr` if `s` = `o`. `Addr` is an absolute address of type `M`.
`jle`, `jng`	addr	Jumps to `addr` if `z` = `1` and `s` <> `o`. `Addr` is an absolute address of type `M`.
`jz`, `je`	addr	Jumps to `addr` if `z` = `1`. `Addr` is an absolute address of type `M`.
`jnz`, `jne`	addr	Jumps to `addr` if `z` = `0`. `Addr` is an absolute address of type `M`.

jc	addr	Jumps to addr if c = 1. Addr is an absolute address of type M.
jnc	addr	Jumps to addr if c = 0. Addr is an absolute address of type M.
jo	addr	Jumps to addr if o = 1. Addr is an absolute address of type M.
jno	addr	Jumps to addr if o = 0. Addr is an absolute address of type M.
js	addr	Jumps to addr if s = 1. Addr is an absolute address of type M.
jns	addr	Jumps to addr if s = 0. Addr is an absolute address of type M.
jmp	addr	Unconditional jump. Addr is an absolute address of type M.
call	addr	Pushes IP registers onto stack; then jumps to addr. Addr is an absolute address of type M.
ret	-	Pops data from stack to IP (i.e. does a jump to address on stack)
int	num	Initiates software interrupt num. Num is type I8.
iret	-	Return from interrupts handlers. Pops FLAGS and IP from stack.
halt	-	Brings processor to a halt. In this project this instruction will stop simulation.

44.3.6. Input/output

This section contains operations that send and receive data via input/output ports.

Name	Arguments	Description
in	dest, prt	Reads data from port prt and places it to dest. Dest/prt can be following combinations: R/R, R/I8.
out	prt, src	Writes data src to port prt. Allowed prt/src combinations: R/R, R/I, I8/R, I8/I.

44.3.7. Other

Name	Arguments	Description
nop	-	No operation. The CPU goes on to fetch next instruction after fetching this one.

44.3.8. Notes

[3] Shifts/rotations with num greater than 1 are equivalent to several shifts/rotations by 1. Only 4 low-order bits matter in num operand. Therefore, the maximum number of shifts/rotates in one operation is 15 and the minimum is 0.

Appraisal

44.4. Comparison to original requirements

The General requirements were all achieved without any doubt. The users can load and save their code, write programs with syntax highlighting, run, pause and step through programs, place breakpoints, watch register/variable contents and many other things. But more importantly, as required, CLab can not only be used to teach assembly language. CLab shows in detail the internal structure of a simple computer, and lets the user see exactly what happens inside a computer when it runs a program.

The system supports three different detail levels (i.e. complexity levels), as required. The users can choose between Basic (GCSE) mode, Medium (A-level) mode and Full mode. It is true that the user can write and debug programs without ever knowing it is compiled, and that is true of all complexity modes, and not just Basic mode as stated in the Requirements.

All instructions required were implemented. Also many other instructions are available in CLab. The user has a choice of several addressing modes, all of which are required by the A-level syllabus.

One of the features which was required provided there will be enough time was a set of animations which would help to explain some topics. This requirement was not met due to lack of time.

44.5. Objectives

Three distinct objectives were stated in the Objectives section in Analysis. One of them was to demonstrate the operation of internal computer components, especially the CPU. This objectve was met – CLab shows the workings of the CPU, video controller, keyboard controller and speaker. Another objective was to provide a fully functional assembly language simulator. CLab does indeed provide such a simulator, with syntax highlighting and debugging features. Unfortunately, the third, optional objective of providing a set of interactive simulations was not met because there was not enough time available.

44.6. Success?

My Computing teacher, who is my main end user, agreed to spend a 90-minute session with the class using CLab to write assembly language programs. By the end of the session all students (there were four) had a working program. It is hard to tell whether this can be considered a success according to the quantitative criteria set in Analysis. I was unable to test the program on a bigger class. In such a small class everybody got more help from the teacher. There was another problem – at this time of year, everybody has already had some practice in writing in assembly language. It was not much, but again it doesn't match the criteria.

Still, I think the project is a success. The students did not think CLab was too complicated to understand. They said that user manual was helpful. My teacher also said he thought the program is useful as a teaching means.

44.7. Ways to improve

There are still lots of things I can think of which can be improved about CLab. Three months to half a year of intense work on it could bring it up to a sellable standard. Here is a list of some possible improvements:

- There are still many loose ends in the program. It sometimes crashes because of the hooked minimize/restore events, which *has* to be fixed for a real system.
- Error messages in assembly language compiler can be made a lot more helpful.
- An integrated electronic help would be extremely useful.
- It is not very hard to make a trace table feature – the users can select the variables/registers they need, and CLab will record their value for every instruction executed.
- The tiny data windows which should have shown the signals sent by hardware to communicate were not implemented. It could be done in an improved version of CLab.
- The animations and interactive demonstrations that were not created could be very useful.
- It would be good if the user could save the whole project, including current simulation state, window layout, etc. in a file.
- Syntax highlighting could be made customizable.
- The system could have the “test” feature where the teacher would set up questions and the students would have to answer them.