Technion, Israel Institute of Technology.

Software Laboratory
Electrical Engineering Department
Technion, Haifa, Israel

VAX-11
Assembler & Simulator

Adar Nir, Grosman Rotem
Instructor – Zion Gal

1. Contents

2. Introduction

2.1. Project Objective

The project goal was the implementation of an assembler and simulator for the VAX-11 computer. It is intended that the Technion Israel Institute of Technology will use the software as a teaching tool.

2.1.1. General Project Description

VAX-11 is a 32-bit word length computer dating from the 80's. Currently, it has almost entirely been replaced. However, the Technion University among other institutions uses it as an initial teaching tool to introduce the world of assembler to students.
Therefore, there is a need for an assembler program that is able to compile VAX-11 assembly programs to machine code and a simulator to test these programs.

2.1.2. The Project Goal

The Technion is currently using a VAX-11 simulator written in 1989. It is a DOS program with many bugs and limitations. Our goal is to create a similar system for Windows without those limitations with a better user-interface to make it easy to use.

2.1.3. Times

The project comprises two parts: The first part is writing the working environment and the assembler while the second is writing the simulator.
Total time for the project is 8 months, equally divided between the two parts.

First part's goals:

- Write the Environment
- Write the Assembler
- Generate List File in order to test the assembler.

Second part's goals:

- Write the Simulator
- Debug…

2.1.4. Required Features

- Assembler and Simulator that supports all the old simulator features.
- A Graphic User Interface.
- Windows application.

2.2. Project Result

We have already fully implemented the assembler & simulator and any feature required.

We have also added some additional features:

- Many opcodes the old simulator didn't support.
- Working with multiply files simultaneously.
- Syntax highlight.
- GUI designed especially to be attractive, intuitive and easy to use.
- Help system including VAX-11 Manual, for easier work.
- HEX calculator to help writing code.
- Code optimization option - resulting in shorter code than the old simulator.
- Enhanced console, that supports both text and graphics modes.
- Virtual memory support.
- Physical Memory simulation using LRU policy
- Memory allocation using Best-Fit policy
- Interrupts and Exceptions Support
- Option to analyze the program running time.

The size of the executable program is around 1MB.

2.3. The old simulator

The Technion is currently using a VAX-11 simulator written on 1989. It is a DOS program with many limitations and bugs.

2.3.1. Problems with the Old Simulator - General

- Dos Program
- Unfriendly Interface
- Can handle only one file at time
- Memory and others limitation
- BUGS...

2.3.2. Limitations

- Handles a maximum of 1000 lines of code.
- Handles a maximum of 300 symbols.
- Handles a maximum of 400 forward expressions.
- Allows the use of only 64KB of memory.
- Doesn't support quad data type.
- Doesn't support floating-point numbers.
- Doesn't support all VAX-11 directives commands.
- Doesn't support all the VAX-11 addressing modes (6a, 6b, 6a, 7b).
- Doesn't implement many of the VAX-11 opcodes.
- The internal editor is very simple - it doesn't support functions that look trivial in our days like copy, paste, etc.

2.3.3. Bugs

- Memory leak: Crashes the command prompt when exiting the simulator.
- Handles long labels & constants incorrectly
- Bugs cause variables to be redefined occasionally.
- Some code compiles incorrectly.
- Some of the VAX-11 commands behave incorrectly on the simulator: CMPC5, PRINTF, BGTR and ACBL.
- CALLS command fails to limit the number of function parameters to 255 as VAX-11 manual specific.
- The internal simulator editor ruins the user code from time to time.
- More bugs are documented later in this document.

2.3.4. Advantages of Our Simulator

- Windows Application.
- Familiar Interface (Menus, Toolbars, Dialog Forms, Shortcuts, Environment Looks like Visual Studio .Net).
- Can Handle Multiple Files at a Time.
- Advanced code editor.
- Syntax Highlighting, Advanced debug tools, significant documentation and other tools to create a good development environment.
- No Memory Limitations.
- No limits on the user's code.
- Supports all VAX-11 addressing modes.
- Supports quad data type.
- More directive commands.
- Supports many new opcodes.
- Hopefully fewer bugs.

2.4. Design Overview

The Project was designed in several stages:

- **Stage 1**
  - Learning C# Language and the tools that .Net offers to us.
  - Learning the VAX-11 assembler.
  - Learning to use Regular expressions.
  - Learning XML language.
  - Learning to design Windows GUI.

- **Stage 2**
  - Theory Design of the assembler.
  - Design the environment and finding the most appropriate tools to write it.

- **Stage 3**
  - Implementation of the assembler and the environment.

- **Stage 4**
  - Theory Design of the simulator.

- **Stage 5**
  - Implementation of the simulator.

2.4.1. Stage 1 – Learning

Before we started the implementation of the project, we examined several possible environments for the implementation.

2.4.1.1. Development Language

2.4.1.1.1. C++ With MFC

Advantages:

- Very popular environment. Many large projects already developed on C++.
- Already 5 years in the market – Most of the known problems with the language has already solved.
- We could start work immediately, as we had already learnt C++ before.

Disadvantage:

- As .Net is out, C++ with MFC might become out-of-date in the next several years.
- MFC is a very complicated system to work with.

2.4.1.1.2. .Net Technology using C#

Advantages:

- Fully object-oriented language.
- Top of today's technology.
- .Net offers wide set of libraries, making the creating of the GUI and the implementation of the assembler & simulator much easier and faster.

Disadvantage:

- As a new product, Microsoft C# implementation suffers from many bugs.
- Very few information can be found about C#, so it was harder to master it.

We selected C# as the implementation language, as we decided we want to master new technology and explore its features. The major reason was the long term viability of C#; it would be a strategic mistake to design a program using a language that apparently will become obsolete in the near future.

2.4.1.2. Other Technologies

We combined several technologies in our project.

2.4.1.2.1. Regular Expressions

- The process of compiling assembly program into machine code includes numerous string manipulation works.
- After examining the subject of string manipulation, we decided to learn the subject of Regular Expressions.

- Regular Expressions are a compact way of describing complex patterns in text.
  We can use them to search for patterns and, once found, to modify the pattern in complex way.

2.4.1.2.2. XML Language

- In order to save the environment settings, we checked several technologies:
  Windows Registry, INI Files and XML files.
- Windows Registry looks like the best way to store per-user information. Yet, many systems don't allow the user to write in the registry. Therefore, so we decided not to use it.
- INI files are common way to store information. Windows itself uses INI files.
  Yet, we didn't use INI files, as we found that the new standard of configuration files is using XML language.
- XML is a meta-language. XML is smaller version of SGML.
  XML lets the user define data structures and stores data in it.

2.4.2. Stage 2 – Design the assembler and the environment

2.4.2.1. The Environment

2.4.2.1.1. Magic Library

We decided to create an environment with an interface similar to Microsoft Visual Studio. To our surprise, the .Net environment contains only basic tools for creating GUI. In order to make our project look professional, we searched the web and found open-source project, named “Crownwood Magic Library”, that offers extra GUI components for .Net, such as tabbed groups and docking windows. Thus, we adopted the Magic project into our environment.

Unfortunately, Magic Library contained many bugs. We used Magic around 5 months, waiting all the time for new releases that didn't solve the problems. Therefore we abandon the using of Magic Library, and moved to other library.

2.4.2.1.2. Divil Library

Divil Library is another library that offers extra GUI components for .Net.
It offers all the components Magic Library offers and some more, as enhanced menus and toolbars.
Divil Library is new development tool - its first release was far after we started our project. Yet it is easier to use than Magic Library, and with less bugs, although the version we used in our project was only a beta version.

2.4.2.1.3. Win32 APIs

The environment uses the RichTextBox component to display the users file.
However, .Net implementation of RichTextBox doesn't fit our needs.
In order to create the desire environment, we used Win32 APIs to enhance the .Net RichTextBox.
We also used Win32 APIs to extend the PropertyGrid control, to make it fit our needs.

2.4.2.1.4. Multithreading

Our simulator lets the user run multiply VAX-11 programs concurrently.
In order to support that option, our project uses several threads, one for the environment and another one for each running program.

2.4.2.2. The Assembler

After we gained a deep understanding of the VAX-11 assembler, we came to design our own assembler.
As many other assemblers do, we decided to compile the code in 2 passes.
We wrote detailed algorithms for the assembler, described later in this document, before starting writing it. Our algorithms worked very well. We didn't need to change any of them during the implementation the assembler simply worked as planned.

2.4.3. Stage 3 – Implementation

After designing the assembler, we implemented it using C#.

In order to test the assembler, we used sources files from different sources:

- Our own samples to test the assembler.
- Examples the Technion uses to teach students, which are published on their web site.
- A massive quantity of student homework in order to test the assembler against "Real-Life" programs.

2.4.4. Stage 4 - Designing the simulator

After the assembler was ready, we started to design the simulator, in order to be able to test our compiled code.
First of all we wrote a detailed description of the VAX-11 computer - including all the features we wanted to support. That description used by us as check-list to see we achieved all our goals.
Then we carefully reviewed all the opcodes we wanted to support and wrote test program for each one of them. We decided that the best way to write the simulator without any hidden bugs is to design a test for each one of the commands to make sure it works as planned.
When the background was ready, we designed the simulator.

2.4.5. Stage 5 - Implementation of the simulator

The last and the biggest stage of our project was the implementation of the simulator. We had many interesting problems during the work: making the different threads of the simulator to work together, share the processor time between the running programs and the working environment, implementing the VAX-11 interrupts and more.
This part included lots and thinking, and also lots of work, of implementing all the opcodes and test the whole project.
During and after the writing of the simulator we did massive testing to all the simulator, checking it works exactly as planned.
Days and night were spent to make the simulator the way we imaged it.

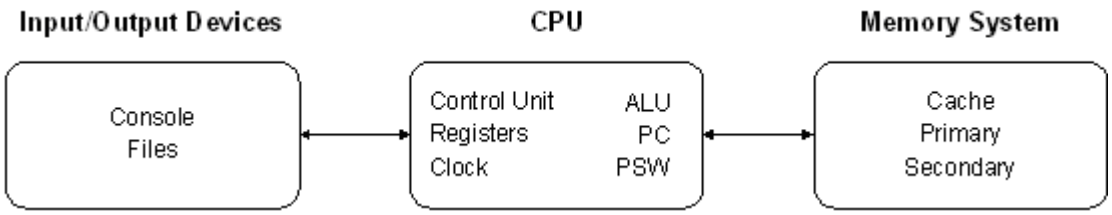
2.5. Acknowledgements

Many people helped us during the working on our project.
First of all, we would like to acknowledge **Dr. Ilana David**, the Software Lab Chief Engineer, for accepting our proposal to make the project and for the hours she spent with us, giving us a deep understanding of object-oriented programming and many other useful hints.
We are also grateful to **Dr. Jair Jehuda** who initiated the idea of creating simulator for the VAX-11, and provided us very useful information and tips about the VAX-11 computer.
Special thanks go to **Victor Kulikov**, the Software lab's system manager. His support and willing to help us took our project steps ahead.
We would like also to thank the other lab workers: **Oleg Romanov** and **Vadim Sherman**.
We wish to thank our instructor **Gal Zion**. Without his support and directives we couldn't complete the project.

Nir Adar,
Rotem Grosman,
August 2003

3. VAX11 Simulator Specifications

3.1. The Basic structure of VAX-11 Computer



This section will describe the different parts of the computer.

3.2. The memory system

The memory contains programs and data. In the original VAX11, the cache memory is used to speed-up the access time to the memory.
From the user's point of view, VAX-11 memory is consecutive.
The addresses of the VAX-11 are 32 bit addresses; therefore each program can use up to 4096MB of memory space.

3.3. Central Processing Unit (CPU)

The VAX-11 ALU supports the basic math operations (+, -, *, /) for integers only.
It also supports logic operations: *shift, rotate, and, or* and *not*.

The control unit reads commands from the memory and executing it.
PC, the *Program Counter*, contains the address of the next command to be executed.
The value of the PC is advanced during the decoding of the opcode and the operands.

3.4. Processor Status Word (PSW)

The Processor Status Word (the lower word of the Processor Status Longword) is a special processor register that a program uses to check its status and to control synchronous error conditions. The Processor Status Word contains two sets of bit fields:

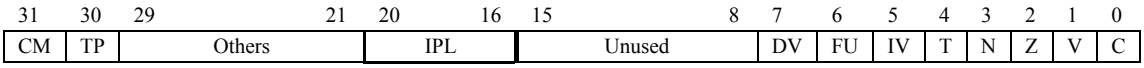
1. The condition codes.
2. The trap enabled flag.

The condition codes indicate the outcome of a particular logical or arithmetic operation.
There are two kinds of traps that concern the user process: trace traps and arithmetic traps. The trace trap is used by debugging programs or performance evaluators.
Arithmetic traps include:

- Integer, floating point, or decimal string overflow, in which the result was too large to be stored in the given format.
- Integer, floating point, or decimal string divide by zero, in which the divisor supplied was zero.

3.5. Processor Status Longword (PSL)

There are a number of processor state variables associated with each process, which VAX-11 groups together into the 32-bit Processor Status Longword. Bits 15-0 of the PSL are referred to separately as Processor Status Word (PSW). The PSW contains unprivileged information, and those bits of the PSW which have defined meaning are freely controllable by any program. Bits 31-15 of the PSL contain privileged status, and while any program can perform the REI instruction (which loads PSL), REI will refuse to load any PSL which increase the privileged of a process, or create an undefined state in the processor.



The PSL

Bits 3:0 of the PSL are termed the condition codes; in general they reflect the result status of the most recent instruction which affects them. The condition codes are tested by the conditional branch instructions.

- *N Bit (Negative)*: Bit 3 is the Negative condition code. In general it is set by instructions in which result stored is negative, and cleared by instructions in which the result is zero or positive.
- *Z Bit (Zero)*: Bit 2 is the Zero condition code. In general it is set by instructions in which result stored is zero, and cleared by instructions in which the result is not zero.
- *V Bit (Overflow)*: Bit 1 is the Overflow condition code; In general it is set after arithmetic operations in which the magnitude of the algebraically correct result is too large to be represented in the available space, and cleared after operations whose result fits.
- *C Bit (Carry)*: Bit 0 is the Carry condition code; In general it is set after arithmetic operations in which a carry out of, or borrow into, the most significant bit occurred. C is cleared after arithmetic operations which had no carry or borrow, and either cleared or unaffected by other instructions.

Bits 7:4 of the PSL are trap-enable flags, which cause traps to occur under special circumstances.

- *T Bit (Trace)*: Bit 4 is the trace bit; when set, it causes a trace trap to occur after execution of the next instruction.
- *IV Bit (Integer Overflow)*: Bit 5 is the Integer overflow trap enable; when set, it causes an integer overflow trap after an instruction which

produced an integer result that could not be correctly represented in the space provided. When bit 5 is clear, no integer overflow trap occurs.

- *FU bit (Floating Underflow)*: Bit 6 is the Floating Underflow bit. Our simulator doesn't support this bit.
- *DV Bit (Decimal Overflow)*: Bit 7 is the Decimal Overflow trap enable. When set, it causes a decimal overflow trap after the execution of any instruction which produces a decimal result whose absolute value is too large to be represented in the destination space provided.

- *IPL Bits*: Bits 16-20 represent the processor's Interrupt Priority level. An interrupt, in order to be acknowledged by the processor, must be at a priority higher than the current IPL.
- *TP Bit*: Bit 30 is the Trace Pending bit, which is used by the processor to ensure that one, and only one, trace trap occurs for each instruction performed with the Trace bit (bit 4) set.

3.6. Registers

A Register is special hardware within the processor that can be used for temporary data storage and addressing. Instruction operands are often stored in the processor's general registers or accessed through them.
The VAX-11 computer has 16 32-bit Registers, named *r0...r15*.
r0…r11 are general purpose registers, while *r12…r15* are special control registers.

Register	Alternative Name	Description
r12	AP	Argument Pointer - contains the address of the base of a software data structure called the argument list, which is maintained for procedure calls.
r13	FP	Frame Pointer - contains the address of the base of a software data structure stored on the stack called the stack frame, which is maintained for procedure calls.
r14	SP	Stack Pointer - contains the address of the base (also called the top) of a stack maintained for subroutine and procedure calls.
r15	PC	Program Counter - contains the address of the next byte to be processed in the instruction stream.

3.7. Input/Output Devices

The simulator supports serial input / output devices.

Input: The keyboard (KBD) or input file.

Output: The monitor (CRT) or output file.

3.8. VAX-11 Information Unit

VAX-11 Information units are: *Byte*(1), *Word*(2), *Long*(4), *Quad*(8).

Data stored in the memory in *Little-Endian* – The first byte is always the LSB.

The CPU always considers the numbers as signed.

3.9. Instruction Format

The first byte denotes:

- The operation type.
- The number of operands and their size.

The following bytes of the instruction contain the operands, each of which may use a different addressing mode.

The VAX-11 instruction length is variable.

3.10. Addressing Modes

	Addressing Mode	Assembly Code	Machine Implementation	
1	Register	r*NUMBER*	50+Number	
2	Register Deferred	(r*NUMBER*)	60+Number	
3	Autoincrement	(r*NUMBER*)+	80+Number	
4	Autodecrement	-(r*NUMBER*)	70+Number	
5	Autoincrement Deferred	*(r*NUMBER*)+	90+Number	
6a 6b 6c	Displacement	OFFSET(r*NUMBER*) or OFFSET[r*NUMBER*]	A0+Number C0+Number E0+Number	Byte Offset Word Offset Long Offset
7a 7b 7c	Displacement Deferred	*OFFSET(r*NUMBER*)	B0+Number D0+Number F0+Number	Byte Offset Word Offset Long Offset
8a 8b 8c 8d 8e 8f	Index	(r*BASE*)[r*INDEX*] (r*BASE*)+[r*INDEX*] -(r*BASE*)[r*INDEX*] *(r*BASE*)+[r*INDEX*] OFFSET(r*BASE*)[r*INDEX*] *OFFSET(r*BASE*)[r*INDEX*]	40+Index	Depending on the BASE addressing mode.
9	Literal	$VALUE (<64)	00+VALUE	
10	Immediate	$VALUE	8F	Long Number
11	Absolute	*$ADDRESS	9F	Long Address
12	Relative	ADDRESS	EF	Long Address
13	Relative Deferred	*ADDRESS	FF	Long Address

We can see that addressing modes 1-9 are the only real addressing modes. Addressing modes 10-13 uses the previous addressing modes to achieve some special effects using PC register.

To show the meaning of the different addressing modes more clearly, we put here the nine basic addressing modes, and their "equivalents" on C language:

	Addressing Mode	Assembly Code	C Code
1	Register	r*NUMBER*	r
2	Register Deferred	(r*NUMBER*)	*r
3	Autoincrement	(r*NUMBER*)+	*(r++)
4	Autodecrement	-(r*NUMBER*)	*(--r)
5	Autoincrement Deferred	*(r*NUMBER*)+	**(r++)
6	Displacement	OFFSET(r*NUMBER*)	*((char*)r+OFFSET)
7	Displacement Deferred	*OFFSET(r*NUMBER*)	**((char*)r+OFFSET)
8	Index	(r*BASE*)[r*INDEX*] (r*BASE*)+[r*INDEX*] -(r*BASE*)[r*INDEX*] *(r*BASE*)+[r*INDEX*] OFFSET(r*BASE*)[r*INDEX*] *OFFSET(r*BASE*)[r*INDEX*]	*(rBASE+rINDEX) *((rBASE++)+rINDEX) *((--rBASE)+rINDEX) **(rBASE+rINDEX) *((OFFSET+(char*)rBASE)+rINDEX) *(*(OFFSET+(char*)rBASE)+rINDEX)
9	Literal	$VALUE (<64)	VALUE

3.11. Function Calls

VAX-11 supplies several mechanisms for calling to functions.

Calling using JSB

Syntax: JSB <function_name>

Actions:

1. Push PC
2. Update PC to new value.

Remarks:

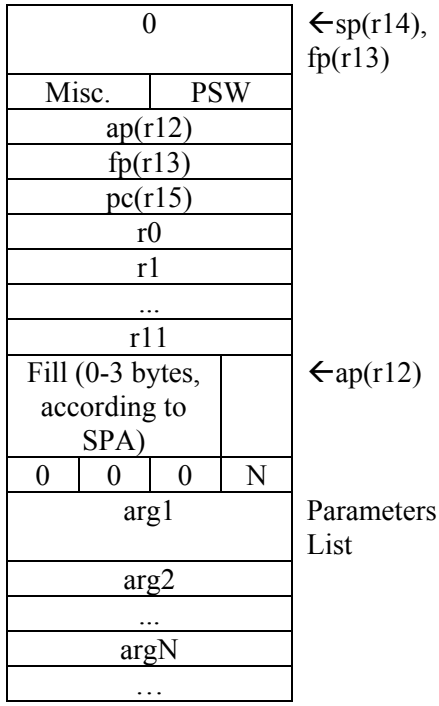
- Returning from the function using RSB.

Calling using CALLS

Syntax: CALLS <parameters_number>, <function_name>

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
PSW	Flags: N, Z, V, C, T, More...															
Misc.	SPA			S	0	Register Mask for r0-r11										

S is 1 for *calls*.



Actions:

0. Caller Pushes parameters.
1. Push parameters number (N). Max: 255
2. The 2 lsb bits of SP goes to SPA. Also temp←sp
3. Using SPA, we decide to fill 0-3 bytes, to align the stack.
4. Pushing r0-r11 according to the mask.
5. Pushing pc, fp, ap (Returning Address).
6. Resetting PSW Flags.
7. Push PSW+Misc.
8. Push 0 to mark the frame's end.
9. FP←SP, AP←temp
10. PC←Function Address + 2

Calling using CALLG

Syntax: CALLG <global_section>, <function name>

- Uses global section to pass parameters.
- The global section should be similar to the one in the CALLS

0	0	0	N
arg1			
arg2			
...			
argN			

Stack structure after CALLG:

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
PSW	Flags: N, Z, V, C, T, More...															
Misc.	SPA			S	0	Register Mask for r0-r11										

0	
Misc.	PSW
ap(r12)	
fp(r13)	
pc(r15)	
r0	
r1	
...	
r11	
Fill (0-3 bytes, according to SPA)	

S is 0 for *callg*.

Actions:

0. Preparing parameters list on the global memory.
1. Including parameters number (N). Max: 255
2. The 2 lsb bits of SP goes to SPA.
3. Using SPA, we decide to fill 0-3 bytes, to align the stack.
4. Pushing r0-r11 according to the mask.
5. Pushing pc, fp, ap (Returning Address).
6. Resetting PSW Flags.
7. Push PSW+Misc.
8. Push 0 to mark the frame's end.
9. FP←SP, AP←global section address.
10. PC←Function Address + 2

- callg doesn't fit for recursive functions.
- Can be used for RPC (Remote Procedure Call) - Function that is called by another process.

Returning from function using RET

1. sp←fp+4
2. temp←Misc. + PSW
3. Restoring the pc, fp, ap registers.
4. Restoring r0-r11 according to the mask.
5. Restoring SP using SPA from temp.
6. Restoring PSW from temp.
7. Skipping the fill using SPA.
8. If S = 1, then read N and jump over the parameters. (Assuming each parameter is exact one longword).

Local Variables

We can allocate space for N local variables using: "subl2 $N, sp".
We can use the variables using negative offset from fp:

- -4(fp) for the first word.
- -8(fp) for the second word.
- etc...

3.12. Interrupts

3.12.1. Introduction

At certain times during system operation, internal or external events may require the execution of pieces of software outside of explicit flow of control.
Some of these events are relevant to the currently executing process, and normally invoke software in the context of the current process. The notification of these events is termed an *exception*.
Other events are relevant to other processes, or to the system as a whole, and are serviced in a system-wide context. The notification process for these events is termed an *interrupt*.
Some interrupts are so urgent that they require high priority service. To meet these needs, the VAX-11 has priority logic that grants interrupt service to the highest priority event at any point in time. The priority associated with an interrupt is termed its interrupt priority level (IPL).
The processor arbitrates interrupt requests according to priority. Only when the priority of an interrupt request is higher than the current IPL (bits<20:16> of the Processor Status Longword) does the processor raise the IPL and service the interrupt request. The interrupt service routine is entered at the IPL of the interrupt request and does not usually change the IPL set by the processor.
Interrupt requests can come from devices, controllers, or the processor itself. Software executing in kernel mode can raise and lower the priority of the processor by executing "MTPR src, IPL" where src contains the new priority desired.
Most service routines for software-generated exceptions execute at IPL 0. However, if a serious system failure occurs, the processor raises the IPL to the highest level (1F to prevent interruption until the problem is corrected. Exception service routines are usually coded to avoid exceptions; however, nested exceptions may rarely occur in the case of an access control violation, reserved operand, or reserved addressing mode fault.

3.12.2. Processor Interrupt Priority Levels (IPLs)

The processor has 31 interrupt priority levels (IPLs), divided into 15 software levels (numbered 1 to F) and 16 hardware levels (10_{16} to $1F_{16}$). User applications, system calls, and system services all run at IPL 0, which may be thought of as process level. Higher numbered IPLs have higher priority; that is to say, any requests at an interrupt level higher than the processor's current IPL interrupt immediately, but requests at a lower or equal level are deferred.
Interrupt levels 1 through F exist entirely for use by software. No device can request interrupts on those levels, but software can force an interrupt by executing "MTPR src, SIRR" (Software Interrupt Request Register). Once a software interrupt request is made, it is cleared by hardware when the interrupt is taken.
Interrupt levels 10_{16} to 17_{16} are for use by devices and controllers.
Interrupt levels 18_{16} to $1F_{16}$ are used by urgent conditions, including the interval clock, serious errors, and power fail.

Some known IPLs:

- System Clock: 18_{16}
- Terminal: 14_{16}

3.12.3. Contrast between Exceptions and Interrupts

Generally, exceptions and interrupts are very similar. When either is initiated, both the Processor Status Longword (PSL) and the Program Counter (PC) are pushed onto a stack. However, there are some differences:

1. An exception condition is caused by the execution of the current instruction, while an interrupt is caused by some activity in the computing system that usually is independent of the current in struction.

2. An exception condition usually is serviced in the context of the process that produced the exception condition, while an interrupt is serviced independently from the current process.
3. The IPL of the processor usually is not changed when the processor initiates an exception, while the IPL always is raised when an interrupt is serviced.
4. Enabled exceptions are initiated immediately, independent of the processor IPL. Interrupts, however, are delayed until the processor IPL drops below the IPL of the requesting interrupt.
5. Most exceptions cannot be disabled. However, if an exception-causing event occurs while that exception is disabled, no exception is initiated for that event, even when enabled subsequently. This includes overflow, which is the only exception whose occurrence is indicated by a condition code (V). If an interrupt condition occurs while that interrupt is disabled, or the processor is at the same or higher IPL, the condition eventually initiates an interrupt when the proper enabling conditions are met (if the condition is still present).
6. The previous mode field in the PSL is always set to kernel on an interrupt, but on an exception it indicates the mode in which the exception occurred.

3.12.4. Software Interrupt Summary Register

The Software Interrupt Summary Register (SISR) is a privileged register which records pending software interrupts. The SISR contains 1s in the bit positions corresponding to levels on which software interrupts are pending. All such levels must be lower than the current processor IPL, or the processor would have taken the requested interrupt.

At bootstrap time, the contents of SISR are cleared.

The mechanism for accessing it is:

"MFPR SISR, dst" Reads the Software Interrupt Summary Register.

"MTPR src, SISR" Loads it, but this is not the normal way of making software

interrupt requests. It is useful for clearing the software interrupt system and for reloading it after a power failure, for example.

3.12.5. Software Interrupt Request Register

The Software Interrupt Request Register (SIRR) is a write-only 4-bit privileged register used for making software interrupt requests.
Executing "MTPR src, SIRR" requests an interrupt at the level specified by src<3:0>. Once a software interrupt request is made, the corresponding bit in the SISR is set. The hardware then clears the bit in the SISR when the interrupt is taken. If src<3:0> is greater than the current IPL, the interrupt occurs before execution of the following instruction, If src<3:0> is less than or equal to the current IPL, the interrupt is deferred until the IPL is lowered to less than src<3:0>, with no higher interrupt level pending. The IPL is lowered by either REI or by "MTPR X, IPL". If src<3:0> is 0, no interrupt will occur or be requested.
No indication is given if there is already a request at the selected level, therefore, the service routine must not assume a one-to-one correspondence of interrupts generated and requests made.

3.12.6. Interrupt Priority Level Register

Writing to the IPLR with the MTPR instruction will load the processor priority field in the Processor Status Longword (PSL). That is, bits<20:16> of the PSL are loaded from IPLR<4:0>. Reading from IPLR with the MFPR instruction will read the processor priority field from the PSL. On writing IPLR, bits<31:5> are ignored, and on reading IPLR, bits <31:5> are returned zero.
At boot time, the IPL is initialized to 1F.
Interrupt service routines must follow the discipline of not lowering the IPL below their initial level. If they do, an interrupt at an intermediate level could cause the stack nesting to be improper. This would result in REI faulting. Actually, a service routine could lower the IPL if it ensured that no intermediate levels could interrupt. However, this would result in unreliable code.

3.12.7. SYSTEM CONTROL BLOCK (SCB)

The System Control Block is a page containing the vectors by which exceptions and interrupts are dispatched to the appropriate service routines.
The interrupt vectors that our simulator supports are as follows:

Interrupt vectors:

Vector's Address	Interrupt Type	Priority	Extra Registers
SCBB+0x18	Reserved Operand Fault	31	None
SCBB+0x28	Trace	31	None
SCBB+0x34	Arithmetic	31	None
SCBB+0x84	Software 1	1	SIRR, SISR
SCBB+0x88	Software 2	2	SIRR, SISR
SCBB+0x8C	Software 3	3	SIRR, SISR
SCBB+0x90	Software 4	4	SIRR, SISR
SCBB+0x94	Software 5	5	SIRR, SISR
SCBB+0x98	Software 6	6	SIRR, SISR
SCBB+0x9C	Software 7	7	SIRR, SISR
SCBB+0xA0	Software 8	8	SIRR, SISR
SCBB+0xA4	Software 9	9	SIRR, SISR
SCBB+0xA8	Software 10	10	SIRR, SISR
SCBB+0xAC	Software 11	11	SIRR, SISR
SCBB+0xB0	Software 12	12	SIRR, SISR
SCBB+0xB4	Software 13	13	SIRR, SISR
SCBB+0xB8	Software 14	14	SIRR, SISR
SCBB+0xBC	Software 15	15	SIRR, SISR
SCBB+0xC0	Clock	24	ICCS, NICR, ICR
SCBB+0xF8	Terminal Input	20	RXCS, RXDB
SCBB+0xFC	Terminal Output	20	TXCS, TXDB

3.12.8. System Control Block Base (SCBB)

The SCBB is a privileged register containing the physical address of the System Control Block

At boot time, the contents of SCBB are UNPREDICTABLE. SCBB must specify a valid address in physical memory or the processor operation is UNDEFINED.

3.12.9. Privileged Registers

VAX-11 contains several special registers.

Below is list of these registers, and some information about it.

Number	Register Name	I/O	Description
17	SCBB	RO	System Control Block Base
18	IPL	RW	Interrupt Priority Level (Default = 0)
20	SIRR	WO	Software Interrupt Request
21	SISR	RW	Software Interrupt Summery
24	ICCS	RW	Interval Clock Control/Status bit0=1 Run - Increase ICR every microsecond. bit4=1 Xfr - Load the ICR Clock from NICR bit5=1 Sgl Manual Incresing the clock. For use when bit0 is 0. bit6=1 Ie - Interrupt Enabled bit7=1 Int
25	NICR	WO	Next Interval Count Register
26	ICR	RO	Interval Count Register
32	RXCS	RW	Console Receive Control/Status bit6=1 Ie - Interrupt Enabled bit7=1 Rdy - There is waiting key on the buffer
33	RXDB	RO	Console Receive Data Buffer
34	TXCS	RW	Console Transmit Control/Status bit6=1 Ie - Interrupt Enabled bit7=1 Rdy - Ready for sending new key
35	TXDB	WO	Console Transmit Data Buffer

3.12.9.1. Console Terminal Registers

The console terminal is accessed through four internal registers. Two are associated with receiving from the terminal and two with writing to the terminal. In each direction there is a control/status register and a data buffer register.

3.12.9.2. Interval Clock

The interval clock provides an interrupt at IPL 24 at programmed intervals. The counter is incremented at 1 μs interval. The clock interface consists of three registers in the privileged register space: the read-only interval count register, the write-only next interval count register and the interval clock control/status register.

3.12.9.2.1. Interval Count Register

The interval register is read-only register incremented once every microsecond. It is automatically loaded from NICR upon a carry out from bit 31 which also interrupts at IPL 24 if the interrupt is enabled.

3.12.9.2.2. Next Interval Count Register

The reload register is a write-only register that holds the value to be loaded into ICR when it overflows. The value is retains when ICR is loaded. NICR is capable of begin loaded regardless of the current values of ICS and ICCS.

3.12.9.2.3. Interval Clock Control/Status Register

The ICCS register contains control and status information for the interval clock.

Bit 31 - ERR
Whenever ICR overflows, if INT is already set, then ERR set. Thus. ERR indicates a missed clock tick. Attempts to set this bit via MTPR clears ERR.

Bit 30:8
Must Be Zero

Bit 7 - INT
Set by hardware every time ICR overflows. If IE is set, then an interrupt is also generated. Attempts to set this bit via MTPR clears INT, thereby re-enabling the clock tick interrupt (if IE is set).

Bit 6 - IE
When set, an interrupt request at IPL 24 is generated every time ICR overflows. (INT is set). When clear, no interrupt is requested. Similarly, if INT is already set and the software sets IE, an interrupt is generated.

Bit 5 - SGL

A write-only bit. If RUN is clear, each time this bit is set, ICR is incremented by one.

Bit 4 - XFR

A write-only bit. Each time this bit is set, NICR is transferred to ICR.

Bit 3:1

Must be zero.

Bit 0 - Run

When set, ICR increments each microsecond. When clear ICR doesn't increment automatically. At boot time, RUN is clears.

3.13. Arithmetic Exceptions

This section describes exceptions occurring as the result of an arithmetic or conversion operation. These mutually exclusive exceptions all are assigned to the same vector in the System Control Block. Each of them indicates that an exception occurred during the last instruction and that the instruction has been completed (in the case of a trap) or backed up (fault). A code unique to each exception type is then pushed on the stack as longword.

Trap code	Exception type
1	Integer overflow
2	Integer divide by zero
7	Subscript range

3.13.1. Integer Overflow Trap

An integer overflow trap is an exception indicating that the last instruction executed had an integer overflow which set the V condition code.
The trap only occurs if the integer overflow enable bit (IV) in the PSW is set. The result stored is the low order part of the correct result, and the type code pushed on the stack is a 1. Not that the instructions RET, REI, REMQUE, MOVTUC and BISPSW, do not cause overflow even if they set V.

3.13.2. Integer Divide By Zero Trap

An integer divide by zero trap is an exception indicating that the last instruction executed had an integer zero divisor. The result stored is equal to the dividend, and the condition code V is set. The type code pushed on the stack is 2.

Example:

```
.text

.set ZERO_FAULT,              0x34

main: .word 0

      calls $0, InitZeroHandler
      movb $4, r0
      divl2 $0, r0

      calls $0, ClearZeroHandler
      movb $4, r0
      divl2 $0, r0

      pushl $0
      calls $1, .exit

InitZeroHandler: .word 2
      mfpr SCBB, r1
      moval handlezero, ZERO_FAULT(r1)
      ret

ClearZeroHandler: .word 2
      mfpr SCBB, r1
      movl $0, ZERO_FAULT(r1)
      ret

handlezero:
      pushl 0(sp)
      pushal format
      calls $2, .printf
      rei

.data
format: .asciz "Divide by Zero Handler. Return Adrress: 0x%X\n"
```

3.13.3. Subscript Range Trap

A subscript range trap is an exception indicating that the last instruction was an INDEX instruction with the subscript operand is lower than the low operand or greater than the high operand. The result is stored in the indexout, and the condition codes are set as if the subscript were within range. The type code pushed on the stack is 7.

3.14. Assembler's Directives

The VAX-11 assembler contains many directives. Directives are special commands that not always translated to code, that give different instruction to the assembler.
In the following pages we present the different directives our assembler supports.

3.14.1. .TEXT - Text code segment

This instruction indicates that the following text (source) should be translated into opcode & operands, rather then Data.
This instruction must appear at the first line of source. No expression follows this statement.

3.14.2. .DATA - Data code segment

This instruction indicates that the following text (source) should be translated into numeric & char. Data, rather then Instructions.
No expression follows this statement.

3.14.3. .SET - Symbol declaration

An Expression is assigned to a symbol Name by the instruction:

.set Name , Expression

Normally the Expression is Constant - an address, a value etc.

3.14.4. .SPACE - Byte Array of Nulls

An array of the size Expression bytes is cleared (to zero) in the memory by the instruction:

.space Expression

This instruction usually defines an array of bytes, words or characters.

3.14.5. .ASCII - String declaration

A character array is created by translating the ASCII string in the instruction:

.ascii "String inside brackets"

backslash ('\') followed by a character or combination of characters, translate the sequence to special chars:

\n	new-line	\t	tab
\b	backspace	\r	carriage return
\\	backslash	""	reverse commas
\ddd	byte value in octal notation		
\xdd	byte value in hex notation		

3.14.6. .ASCIZ - String declaration

A character array is created by translating the ASCII string in the instruction:

.asciz "String inside brackets"

This instruction is identical to .ascii except for the char #0 added at the end of the string. (a single byte containing zero)

backslash ('\') followed by a character or combination of characters, translate the sequence to special chars:

\n	new-line	\t	tab
\b	backspace	\r	carriage return
\\	backslash	""	reverse commas
\ddd	byte value in octal notation		
\xdd	byte value in hex notation		

3.14.7. .ASCIC - String declaration

A character array is created by translating the ASCII string in the instruction:

.asciz "String inside brackets"

This instruction is identical to .ascii except for the fact that the first byte of the string contains its size. Therefore .ascic is limited to strings with less than 256 characters

backslash ('\') followed by a character or combination of characters, translate the sequence to special chars:

\n	new-line	\t	tab
\b	backspace	\r	carriage return
\\	backslash	""	reverse commas
\ddd	byte value in octal notation		
\xdd	byte value in hex notation		

3.14.8. .BYTE - Byte assignment

Assign a (single) Byte to a value of an Expression, by the instruction:

.byte Expression [,Expression...]

A Byte is 8 contiguous bits starting on an addressable byte boundary. The bit are numbered from the right 0 through 7.

Range: unsigned byte 0 .. 255
signed byte -128 .. 127

3.14.9. .WORD - Word assignment

Assign a (single) Word to a value of an Expression, by the instruction:

.Word Expression [,Expression...]

A word is 2 contiguous bytes starting on an arbitrary byte boundary.
The bits are numbered from the right 0 through 15.

Range : word 0 .. 65,535

3.14.10. .INT - Integer assignment

Assign a (single) signed-Word to a value of an Expression, by the instruction:

.int Expression [,Expression...]

An Integer is 2 contiguous bytes starting on an arbitrary byte boundary.
The bits are numbered from the right 0 through 15.

Range : int -32,768 .. 32,767

3.14.11. .LONG - Long assignment

Assign a (single) Long Word to a value of an Expression, by the instruction:

.long Expression [,Expression...]

A Long Word is 4 contiguous bytes starting on an arbitrary byte boundary.
The bits are numbered from the right 0 through 31.

Range : signed -2,147,483,648 ..
.. 2,147,483,647
unsigned 0 .. 4,294,967,295

3.14.12. .QUAD - Quadword storage directive

.QUAD generates 64 bits (8 bytes) of binary data.

Format:
.quad Expression [,Expression...]

3.14.13. .ORG - Fill at address

.ORG tells the assembler to locate the next machine code on the position specific by the .org instruction. for example: .ORG 100 tells the assembler to put the next instruction after the .org directive at address 100.

Format:
.org Address

3.14.14. .ENTRYPOINT - Define the starting address of the program.

Every VAX11 program starts at address 0 or on label 'main' address.
.entrypoint allows the user to define other label or address for the starting of the program.

Format:
.entrypoint Label/Address

Example:

```
.text

.entrypoint start

.org 100
start: .word 0
       pushal hello_str
       calls $1, .puts

       pushl $0
       calls $1, .exit

.data
hello_str: .asciz "Hello, World"
```

Please note that if we define address as starting point, for example: ".entrypoint 0x100", the actual running of the program will start on 0x102, after the mask word.

3.15. System Calls

The VAX-11 Simulator contains some high-level functions - system calls, to make the work of the user easier and faster. The following pages contain description of the system calls our simulator supports.

3.15.1. Exit

Exit function is one of the most important system-calls. The function ends the user's program. Every user's program need to be end using it.

Description: The function ends the user's program with specific error code.
Gets: Error Code - On the stack.
0 means the program ended without error.
Any other number indicates about error.
Returns: Nothing.

Example:

```
.text

# User program here
# ...
pushl $0
calls $1, .exit   # Exit with error code 0
```

3.15.2. Getchar

Description: Read one char from the keyboard.

Gets: Nothing.

Returns: ASCII value of the character in R0. -1 if EOF reached.

Example:

```
.text

main: .word 0
      calls $0, .getchar      # Get char from the keyboard
      movb r0, ...
```

3.15.3. Putchar

Description: Put one character on the screen.

Gets: Character on the stack.

Returns: In R0: 0 on success, -1 on error.

Example:

```
.text

main: .word 0
      calls $0, .getchar      # Get char from the keyboard

      pushl r0                # Push the char to the stack
      calls $1, .putchar

      pushl $0
      calls $1, .exit
```

3.15.4. Gets

Description: Read line from the keyboard to a buffer. The function stop getting input when it reaches NUL(0), CR(0x0D) or LF(0x0A). The ending character is replaced with NUL(0).

Gets: In stack: Buffer's address .

Returns: In R0: buffer's address for success, -1 on EOF or 0 if the string contains ctrl+D (0x04) only.
In buffer: user's string, ended with ASCII 0

Example:

```
.text

main: .word 0

      pushal buffer
      calls $1, .gets          # Get a line

      pushl $0
      calls $1, .exit


.data
buffer: .space 80
```

3.15.5. Puts

Description: Put a string on the screen.

Gets: String's address on the stack.

Returns: In R0: 0 on success, -1 on error.

Example:

```
.text

main: .word 0

      pushal szWelcomeMessage
      calls $1, .puts          # Ask for the user's name

      pushal szUserName
      calls $1, .gets          # Get a line

      pushal szWelcome
      calls $1, .puts          # Print hello to the user
      pushal szUserName
      calls $1, .puts

      pushl $0
      calls $1, .exit

.data
szWelcomeMessage: .asciz "Please Enter your name: "
szUserName: .space 20
szWelcome: .asciz "Hello, "
```

3.15.6. Scanf - Read Formatted Data from stdin

scanf reads data, one character at a time from 'stdin' and stores it in the locations given by 'arguments'. 'Format-string' determines how the input fields are to be interpreted. Each argument must be a pointer to a variable with a type that corresponds to a type specifier in 'Format-string'. 'Format-string' is a character string that contains whitespace characters, non-whitespace characters, and format specifications.
Here is a description of the arguments of scanf.

Format-string:

The format string is read from left to right when the first format specification is encountered, the value of the first input field is converted according to the format specification, and the converted value is then stored in the location specified by the first argument. The value of the second input field is converted according to the second format specification and stored in the second location, and so on.
Characters outside the format string- whitespace characters and non-whitespace characters, described below-should match the sequence of characters being read from the input stream.

Whitespace characters: blank (' '), tab ('\t'), or newline ('\n').
The scanf functions will read but not store all whitespace characters up to the next non-whitespace character in the input. One whitespace character in the format-string matches any number and combination of whitespace characters in the input.
Non-whitespace characters: Are all other ASCII characters except the percent character (%). The scanf functions will read but not store a matching non-whitespace character. If the next character scanned does not match, the function will terminate.

Format specifications: Are introduced by a percent sign (%). Format specifications cause the scanf functions to read and convert characters from the input field into specific types of values. These values are assigned to arguments in the argument list. A format specification has the following form:

% [*] [width] type

Type:
The type character, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number.
The simplest format specification contains only the percent sign and a type character (%s, for example).

The various type specifications are:

d decimal integer
D decimal long integer
o octal integer
O octal long integer
x hex integer
X Hex long integer
c character
s string (array of char)

Asterisk:
The asterisk (*) character following the percent sign suppresses assignment of the next input field. The suppressed input data is assumed to be of the type specified by the character type that follows the *. The field is scanned but not stored.

Width:
The width is a positive decimal integer which controls the maximum number of characters to be read from the current input field. No more than 'width' characters are converted and stored at the corresponding argument.

The prefix 'l' indicates the 'long' version is to be used. The corresponding argument should point to a 'long' object. The 'l' modifier can be used with the d, i, o, and x type characters.
The prefix 'h' indicates the 'short' version is to be used. The corresponding argument should point to a 'short' object.
The 'h' modifier can be used with the d, i, o and x type characters.

'l' and 'h' modifiers are ignored if used with any other type.

Returns: R0 =
The number of fields that were successfully converted and assigned.
A return value of EOF (-1) means an attempt was made to read at end-of-file.
(A return value of 0 means no field was assigned).

Notes:
Number of arguments is not limited. Arguments are pointers to data objects which will be stored by Scanf according to "Format string". The Arguments are pushed in reversed order. (So first data read will be saved in last argument /pointer pushed) Last Argument pushed into Stack is a pointer to the "Format-string".

scanf may stop reading a particular input field before it reaches a space character because:

- the specified width was reached
- the next character cannot be converted as specified
- the next character conflicts with a character in the control string

When any of these situations occur, the next input field is considered to begin at the first unread character.

3.15.7. printf - Write Formatted String to stdout

printf formats and prints a series of characters and values to 'stdout'.
'Format-string' determines what is to be printed and how it is to be printed out.
'Format-string' consists of ordinary characters, escape sequences, and format specifications.
The 'Format-string' is read left to right. When the first format specification is encountered, the value of the first argument after the 'Format-string' is converted and output according to format specifications. The second format specification causes the second argument to be converted and output, and so on.

Escape sequences:
Escape sequences are special character combinations that can represent whitespace and non-graphic characters. They are used to specify actions such as carriage returns and tab movements. Escape sequences consist of a backslash ('\') followed by a character or combination of characters:

\n	new-line
\t	tab
\b	backspace
\r	carriage return
\\	backslash
""	Reverse-commas

If there are arguments following 'Format-string', then 'Format-string' must contain format specifications that determine the output format for these arguments.
Format specifications always begin with a percent sign (%) and have the following form:

% [width] type

Each field of the format specification is a single character or number signifying a particular format option.
The following describes each field.

Type:
The 'Type' character determines whether the associated argument is interpreted as a character, string, or number. The simplest format specification contains only a percent sign and a 'Type' character. (For example: %s prints a string.)
The 'Type' characters are:

d	Decimal	Integer
o	Octal	Integer
x	Hex	Integer
X	Hex	Integer (Capital Letters output)
c		Character
s		String
		- Characters printed up to the first null character ('\0')
ld		Long Integer
lx		Long Hex

Width:
The optional width specifier is a non-negative decimal integer specifying the minimum number of characters to print, padding with blanks and zeros.
Width never causes a value to be truncated.

Returns:

R0 = '0' if successful , '-1' on error.

Notes:

Arguments are : pointers to 'String' variable Value of 'Char',numbers etc. (push address for %s, push values for %c %x %d %o ...)

Arguments are pushed in reversed order. (Last argument is printed first)

Last Argument pushed into Stack is a pointer to the "Format-string".

If there are more arguments than there are format specifications, the extra arguments are ignored.

The results are undefined if there are not enough arguments for all the format specifications.

Ordinary characters are simply copied in the order of their appearance.

If the percent sign is followed by a character that has no meaning as a format field, the character is copied to 'stdout'.

Example:

```
        pushl     x
        pushal    Str
        pushl     y
        pushal    Form1
        calls     $4,.printf
        -
        -

.data
x:              .word     1234
y:              .word     5678
Str:            .asciz    "Greater than"
Form1:          .asciz    "%d is %s %d"
```

--> 1234 is Greater than 5678

3.15.8. malloc - Allocate Memory Block

malloc allocates a block of 'Size' bytes.

Returns: On R0: Pointer to allocated space.

Returns NULL if the space cannot be allocated.

Notes:

One argument pushed into Stack = number of bytes to be allocated.

Use free to deallocate block allocated with malloc.

Example:

```
.text
main: .word 0

      pushl $0x100
      calls $1, .malloc
      pushl r0
      pushal format
      calls $2, .printf

      pushl $0x100
      calls $1, .malloc
      pushl r0
      pushal format
      calls $2, .printf

      pushl $0x100
      calls $1, .malloc
      pushl r0
      pushal format
      calls $2, .printf

      pushl $0x1119
      calls $1, .free

      pushl $0x100
```

```
      calls $1, .malloc
      pushl r0
      pushal format
      calls $2, .printf

      pushl $0
      calls $1, .exit

.org 0x1000
.data
format: .asciz "Address: 0x%08X\n"
```

3.15.9. free - Deallocate Memory Block

free deallocates the previously allocated memory block pointed to by 'Ptr'.

The block must have been allocated by malloc.

Returns: Nothing.

Notes:

One argument is pushed into Stack = pointer to deallocated area.

free deallocates the number of bytes that were allocated in the call to malloc.

Example:

```
.text
main: .word 0

      pushl $0x100
      calls $1, .malloc
      pushl r0
      pushal format
      calls $2, .printf

      pushl $0x100
      calls $1, .malloc
      pushl r0
      pushal format
      calls $2, .printf

      pushl $0x100
      calls $1, .malloc
```

```
        pushl r0
        pushal format
        calls $2, .printf

        pushl $0x1119
        calls $1, .free

        pushl $0x100
        calls $1, .malloc
        pushl r0
        pushal format
        calls $2, .printf

        pushl $0
        calls $1, .exit

.org 0x1000
.data
format: .asciz "Address: 0x%08X\n"
```

3.15.10. sprintf - Write Formatted String to string

sprintf formats and prints a series of characters and values to a string - a buffer that located in the memory.
'Format-string' determines what is to be printed and how it is to be printed out.
'Format-string' consists of ordinary characters, escape sequences, and format specifications.
The 'Format-string' is read left to right. When the first format specification is encountered, the value of the first argument after the 'Format-string' is converted and output according to format specifications. The second format specification causes the second argument to be converted and output, and so on.

This function identical to printf except it sends the output to string and not to stdout.

Escape sequences:
Escape sequences are special character combinations that can represent whitespace and non-graphic characters. They are used to specify actions such as carriage returns and tab movements. Escape sequences consist of a backslash ('\') followed by a character or combination of characters:

\n	new-line
\t	tab
\b	backspace
\r	carriage return
\\	backslash
""	Reverse-commas

If there are arguments following 'Format-string', then 'Format-string' must contain format specifications that determine the output format for these arguments.
Format specifications always begin with a percent sign (%) and have the following form:

% [width] type

Each field of the format specification is a single character or number signifying a particular format option.
The following describes each field.

Type:
The 'Type' character determines whether the associated argument is interpreted as a character, string, or number. The simplest format specification contains only a percent sign and a 'Type' character. (For example: %s prints a string.)
The 'Type' characters are:

d	Decimal	Integer
o	Octal	Integer
x	Hex	Integer
X	Hex	Integer (Capital Letters output)
c		Character
s		String
		- Characters printed up to the first null character ('\0')
ld		Long Integer
lx		Long Hex

Width:
The optional width specifier is a non-negative decimal integer specifying the minimum number of characters to print, padding with blanks and zeros.
Width never causes a value to be truncated.

Returns:

R0 = '0' if successful , '-1' on error.

Notes:

Arguments are: pointers to 'String' variable Value of 'Char', numbers etc. (push address for %s, push values for %c %x %d %o ...)

Arguments are pushed in reversed order. (Last argument is printed first)

Last Argument pushed into Stack is a pointer to the "Format-string".

If there are more arguments than there are format specifications, the extra arguments are ignored.

The results are undefined if there are not enough arguments for all the format specifications.

Ordinary characters are simply copied in the order of their appearance.

If the percent sign is followed by a character that has no meaning as a format field, the character is copied to the buffer.

Example:

```
.text
main: .word 0
      pushl $10
      pushl $7
      pushal format
      pushal buffer
      calls $4, .sprintf

      pushal buffer
      calls $1, .puts

      halt

format: .asciz "%d -- %d\n"
buffer: .space 10
```

3.15.11. Initgraph - Initializes the graphics system

Initializes the graphics system

Gets: Nothing

Returns: On r0: 0, -1 on failure

Remarks: To start the graphics system, you must first call initgraph.

3.15.12. closegraph - Shuts down the graphics system

Shuts down the graphics system

Gets: Nothing

Returns: On r0: 0, -1 on failure

Remarks: closegraph deallocates all memory allocated by the graphics system. It then restores the screen to the mode it was in before you called initgraph.

3.15.13. cleardevice - clears the graphics screen

Clears the graphics screen

Gets: Nothing

Returns: On r0: 0, -1 on failure

3.15.14. line - draws line

line draws a line between two specified points.

line draws a line from (x1, y1) to (x2, y2) using the current color.

Gets: x1, y1, x2, y2 on stack.

Returns: On r0 - 0 on success, -1 on error

Example:

```
.text
.word 0
calls $0, .initgraph
pushl $0
pushl $0
pushl $720
pushl $424
calls $4, .line
```

3.15.15. rectangle - Draws a rectangle

Draws a rectangle (graphics mode)

Gets: Left, Top, Right, Bottom

(left,top) is the upper left corner of the rectangle, and (right,bottom) is its lower right corner.

Returns: On r0: 0 on success, -1 on failure

Example:

```
.text
.word 0
calls $0, .initgraph
pushl $10
pushl $10
pushl $490
pushl $390
calls $4, .rectangle
```

3.15.16. setcolor - sets the current drawing color

Description: setcolor sets the current drawing color. It gets color in RGB format and sets the current color to that color.

Gets: On stack: Red, Green, Blue

Returns: On r0: 0 on success, -1 on failure

Example:

```
.text
.word 0
calls $0, .initgraph
pushl $100
pushl $200
pushl $100
calls $3, .setcolor
```

3.15.17. getmaxx, getmaxy

Returns maximum x or y screen coordinate

Gets: Nothing
Returns: On r0: maximum x or y screen coordinate

3.15.18. putpixel

putpixel plots a pixel at a specified point.

Gets: X, Y on stack
Returns: On r0: 0 on success, -1 on failure

3.15.19. circle

circle draws a circle in the current drawing color.

Gets: OnStack: X, Y - Center point of the circle
radius - Radius of the circle
Returns: On r0: 0 on success, -1 on failure

3.15.20. outtextxy

outtextxy displays a string at the specified location (graphics mode)
outtextxy displays textstring in the viewport at the position (x, y)

Gets: Text, X, Y
Returns: On r0: 0 on success, -1 on failure

3.15.21. setfont

Sets the active font.

Gets: Font Name (String), Font Size
Returns: On r0: 0 on success, -1 on failure

3.16. Instruction Set Summary

Regular opcodes		Cycles
mov	move	1
add	add	1
sub	subtract	1
neg	negate	1
cmp	compare	1
bit	bit test	1
mul	multiply	4-11
div	divide	14-38
rem	reminder	13-38
sft	shift logical	4
ash	shift	3
rot	rotate	3
and	bitwise and	1
or	bitwise or	1
xor	bitwise xor	1
not	bitwise not	1

Control Flow		Cycles
br	branch always	3
be	branch equal	3
bne	branch not equal	3
bg	branch greater	3
bge	branch greater equal	3
bl	branch less	3
ble	branch less equal	3
jmp	jump	4
call	function call	6
ret	function return	4
trap	trap	7
rei	return form interrupt	5

Addressing Mode		Cycles
r*N*	Register	2
(r*N*)	Register Deferred	4
(r*N*)+	Autoincrement	5
-(r*N*)	Autodecrement	5
*(r*N*)+	Autoincrement Deferred	7
OFF(r*N*)	Displacement	4
*OFF(r*N*)	Displacement Deferred	6
(r*B*)[r*IDX*] (r*B*)+[r*IDX*] -(r*B*)[r*IDX*] *(r*B*)+[r*IDX*] OFF(r*B*)[r*IDX*] *OFF(r*B*)[r*IDX*]	Index	Base + 4
$VALUE (<64)	Literal	1
$VALUE	Immediate	5
*$ADDRESS	Absolute	7
ADDRESS	Relative	4
*ADDRESS	Relative Deferred	6

- Not all opcodes are included

4. VAX11 Simulator - User Guide

4.1. Introduction

The working environment contains several connected components: Text Editor, Assembler and Simulator. Using those components we are able to write VAX11 code, compile it and run it. The environment also gives us the ability to run the program step by step and debug it.
The interface is simulator to Visual Studio .Net and to make it easy for new users to work with the simulator.

4.2. Main Features

Text Editor:

- Contains Syntax Highlight option for writing VAX11 code.
- Allows working on several files simultaneously

Assembler:

- Detailed Error Messages
- Code optimization option - resulting in shorter code than the old simulator.
- Can generate output file containing the user code and the machine code.

Simulator:

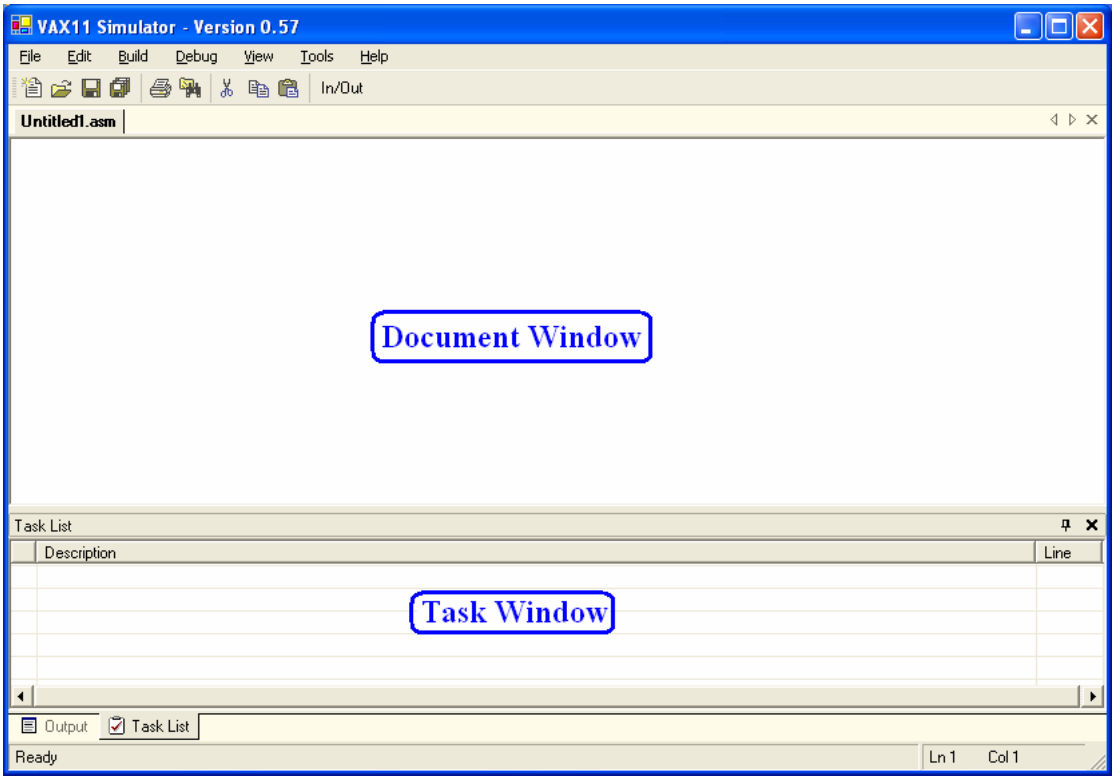
- Supports many of the VAX11 opcodes.
- Supports many VAX system-calls.
- Supports Interrupts and Exceptions
- Virtual memory support, giving 4GB addresses space.
- Contains simulation for physical memory and page faults.
- Option to analyze the program running time.

Debugger:

- Running the program in Step-By-Step Mode
- Supports Break-Points
- Displays registers status, memory and stack.

4.3. Start Working

When opening the working environment, the main application window will appear.



VAX11 Simulator Opening Screen

Document Window is the place where the user writes the program's code. This window is operating as simple text editor.

The **Task Window** is used by the assembler. If it finds errors in the user's code, the task window will contain it, and will give the user to jump directly to that line.

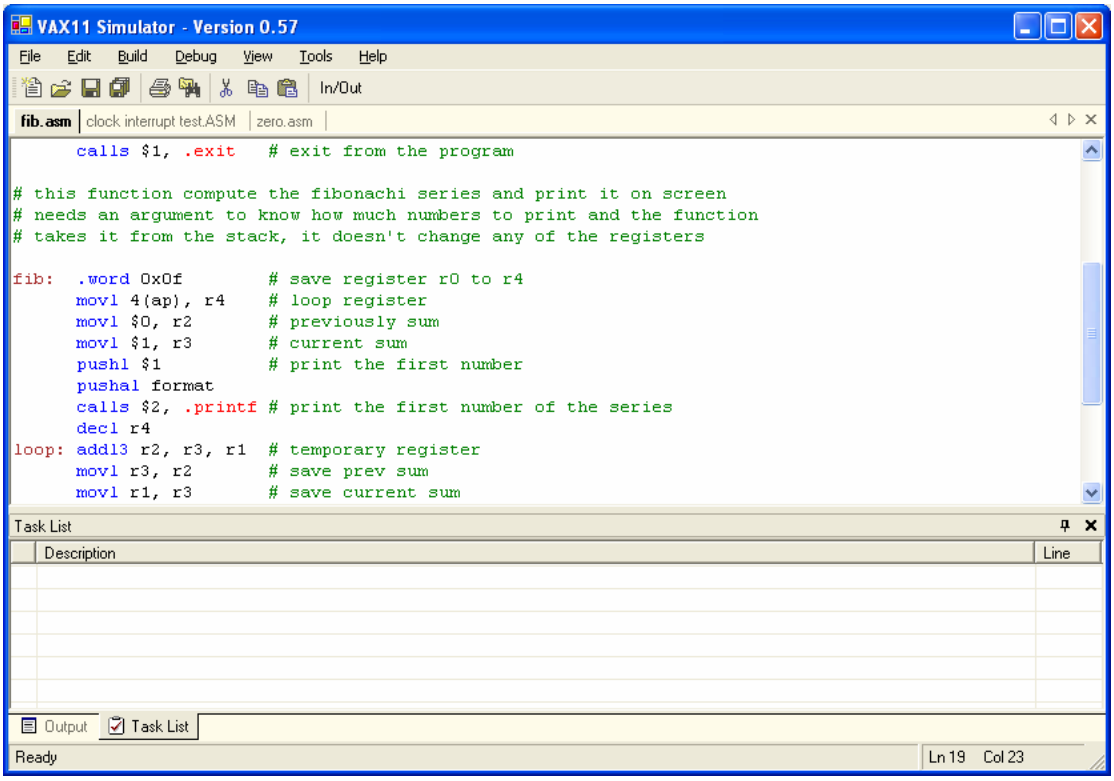
Near the task list we can find the **Output Window**, where the assembler and the simulator sends information about the compilation status or the running status.

4.4. Text Editor

The working environment allows the user to write text files which contains VAX code. Every **document** (program) appears on its own window, and the user is able to switch between the open documents.

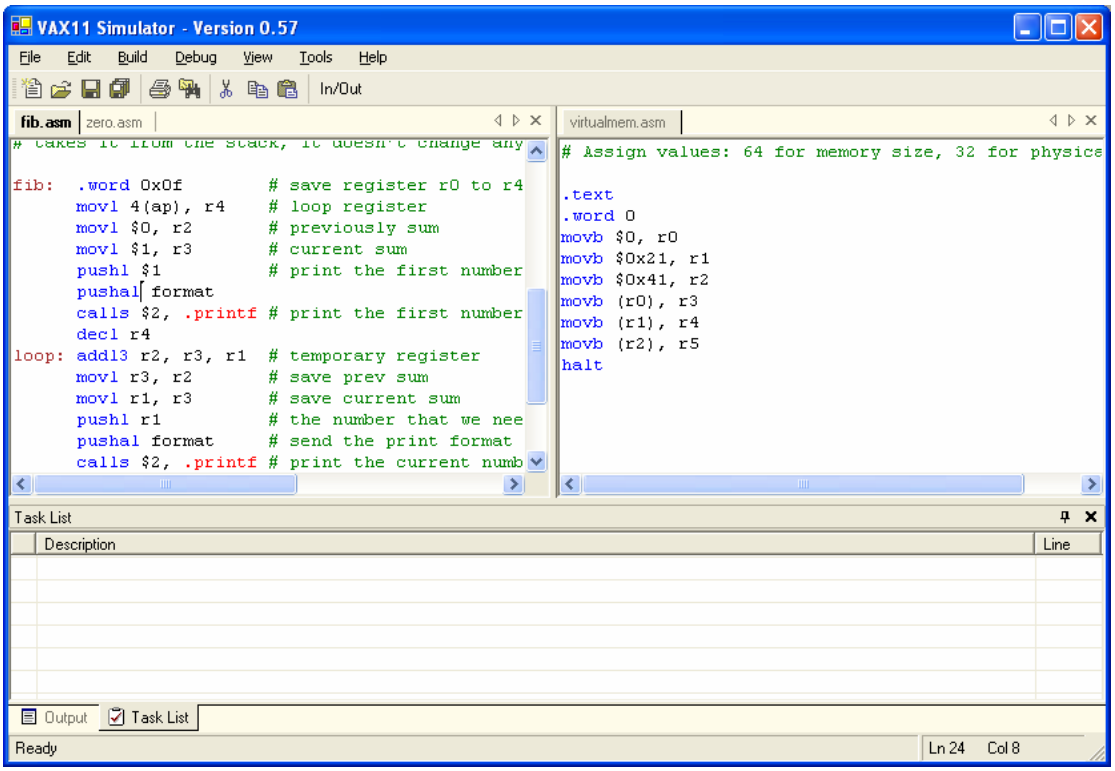
The editor colors the code as we type it, to make it easier to read the source.

We are able to customize the colors via the environment menus.



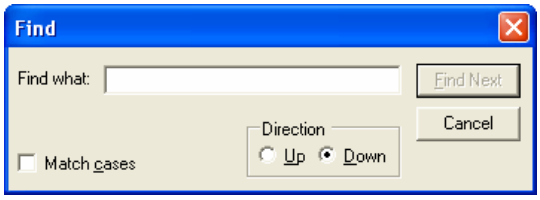
Three open documents

The working environment also let us to split the open documents and view each one of it on part of the screen. Splitting the view is done by dragging the document name from the documents tab to the desire place.



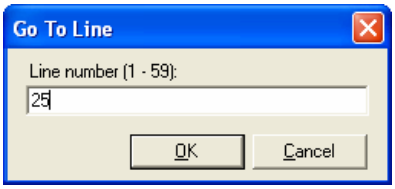
Splitting the windows

The editor contains finding tool in the text. To use it, select "Find" under the "Edit" menu, or press Ctrl+F. Then enter the text to be searched.



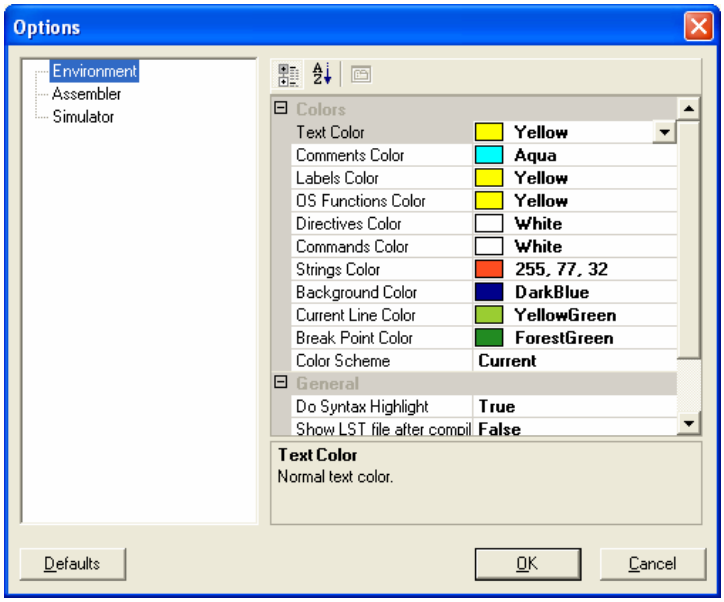
Find Window

Jumping to specific line in the text can be done using "Go To" option in the "Edit" window, or using the Ctrl+G keys.



Go To Line Window

4.5. Options Window



Options Window – Environment Settings

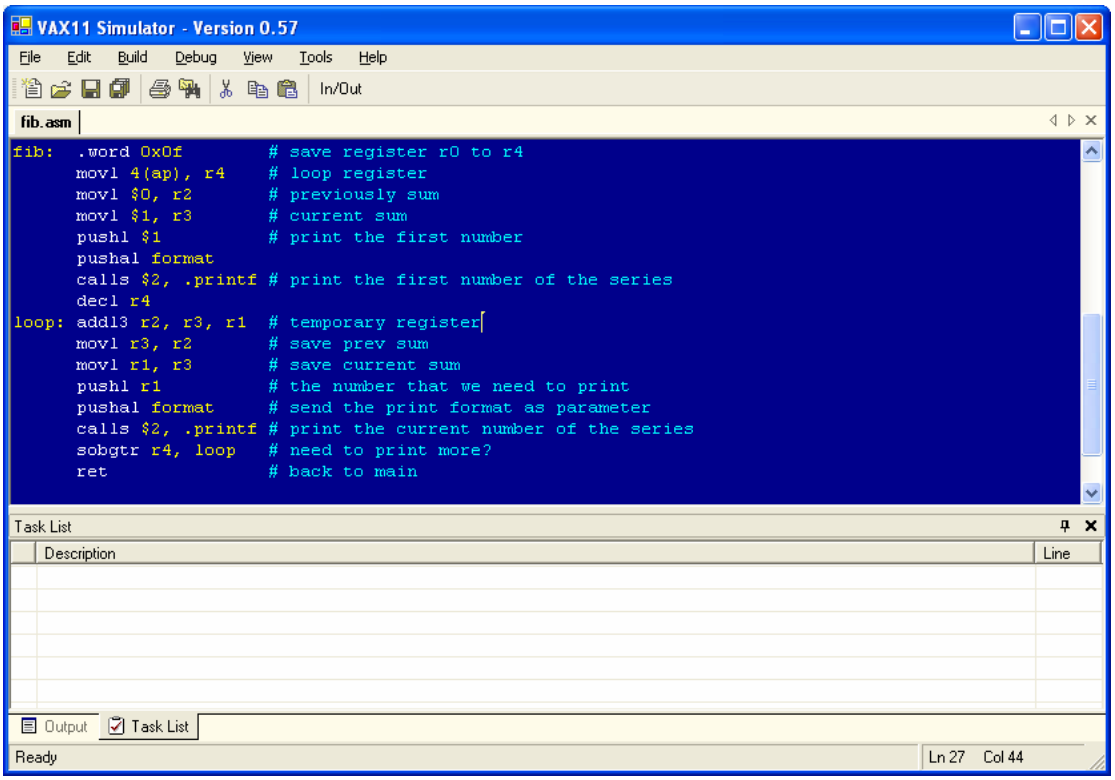
Many aspects of the environment, assembler and simulator can be personalize.

The options window let the user change the different settings of the system.

4.5.1. Working Environment

Colors	
Parameter	**Meaning**
Text Color	Normal text color
Comments Color	Comments start after # sign.
Labels Color	Labels are identifiers following by : that appears on start of lines.
OS Functions Color	The simulator supports some high-level functions, as printf, getchar, etc. These functions are known as operation system functions.
Directives Color	Directives are commands meant for the assembler, that doesn't appear on the final machine code. Examples: .word, .space
Commands Color	VAX11 commands (opcodes) color
Strings Color	Strings are text appear betweens "".
Background Color	Documents Background Color

Current Line Color	Color of the next line that will be executed (debug mode)
BreakPoint Color	BreakPoint Color
Color Scheme	Select one of pre-defined colors sets
General	
Parameter	**Meaning**
Do Syntax Hightlight	Select if the environment should highlight special VAX11 words.
Show LST file after compile	If set, the LST file created during the compilation will be displayed after compile ends successfully.
Show agent on startup	The agent, Merlin, is welcome the users of VAX11 Simulator every time the program runs.



The working environment with custom colors settings

4.5.2. Assembler

General	
Parameter	**Meaning**
Optimize Code	VAX11 Simulator generates smaller code than the old SIM simulator used by the Technion. Set this option to false if you wish the assembler to generate code as SIM does, without its enhancements.
Save LST file after compile	Select if the assembler should save LST file after successful compilation. LST file is text file containing the machine code and the source code of the compiled program.

4.5.3. Simulator

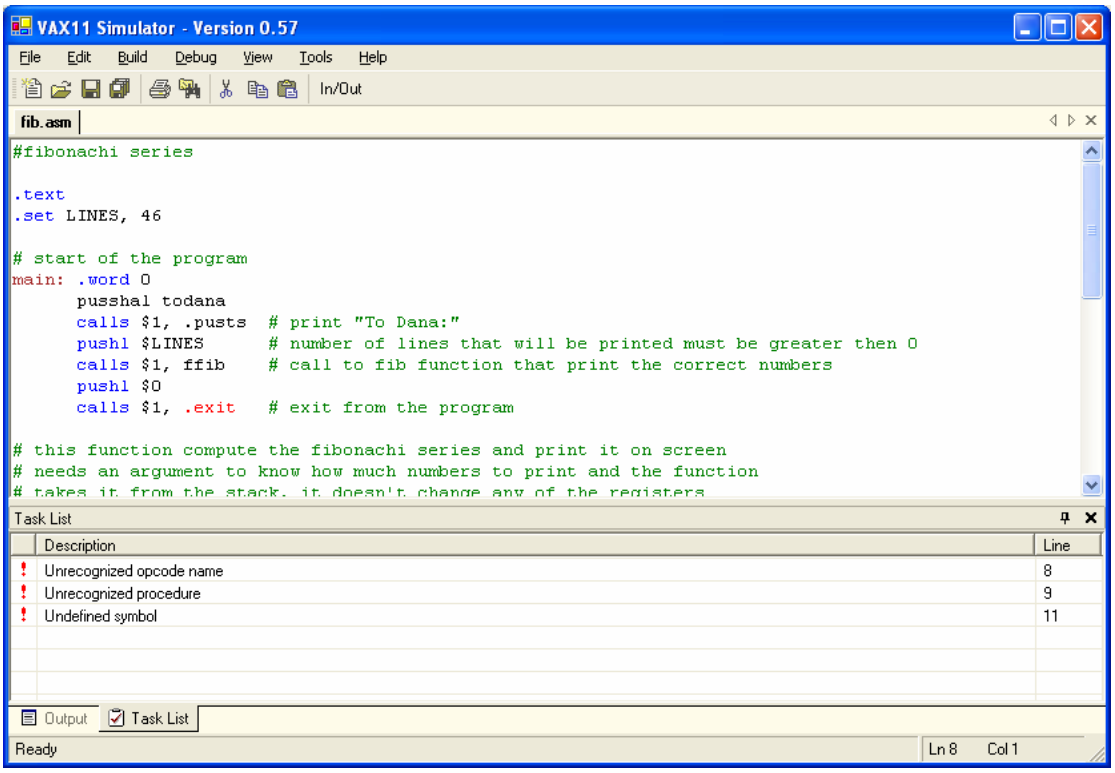
The simulator simulate a physical memory that divided to pages and supports virtual memory. The settings that related to memory effects the simulation of that memory. The console is the input/output window of each running program.

Console	
Parameter	**Meaning**
Text Color	The color of the console's output.
Background Color	Background color for the console.
Always on top on debug mode	If selected, the console window of debugged application will be above all other windows, even when deactivate.
General	
Parameter	**Meaning**
Show Registers in Hex	If true, while debugging, the registers values will be displayed in Hex basis. Else it will be displayed as decimal numbers.
Show Special Registers	If true, while debugging, the special VAX11 registers will be displayed among the general registers.
Show Debug Information	If selected, the simulator will generate detailed information about the simulator state in case of errors in the user's program.

Memory	
Parameter	**Meaning**
Page Size	Memory Page Size
Physical Memory Size	Physical Memory Size. The memory size should be multiple of the page size.
Show Memory Accesses	If selected, the simulator will display information about accesses to memory after each command.
Show Physical Addresses	If selected, the simulator will show "physical" addressing for every virtual address.
Show Page Faults	If selected, the simulator will display message when page-fault occurs.
Fill Memory With Garbage	If true, uninitalize memory cells will contain garbage. If false, it will contain zeros

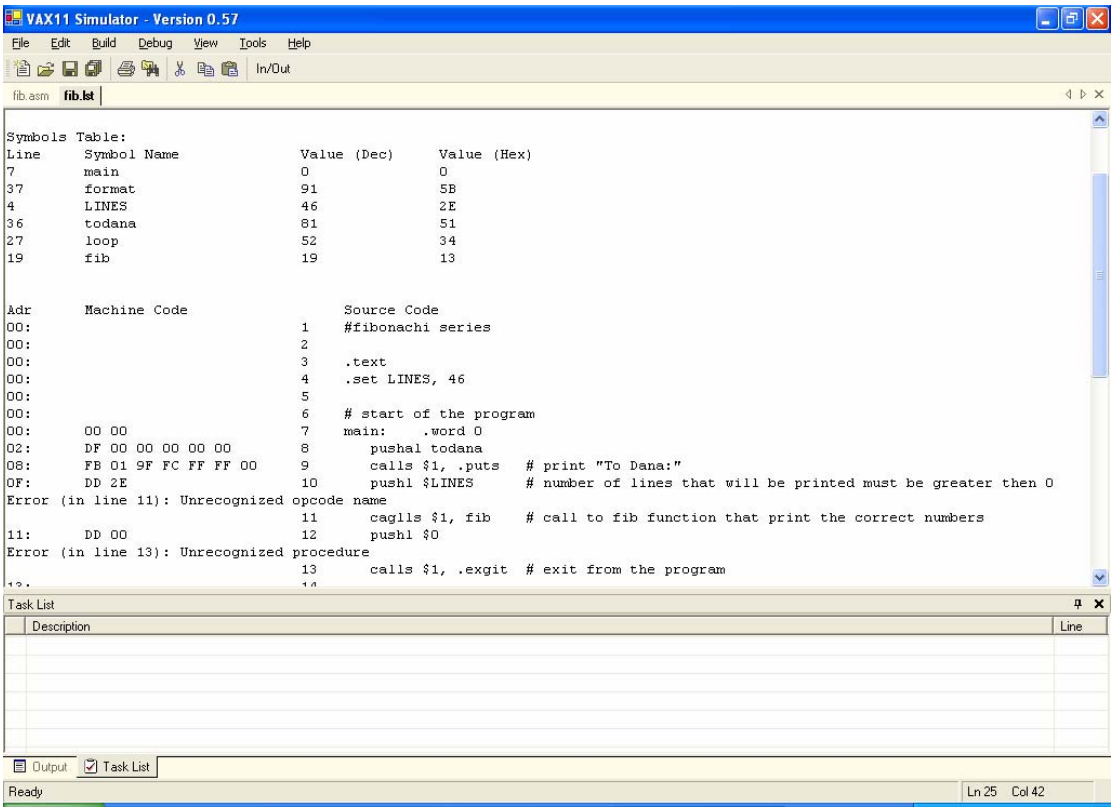
4.6. The Assembler

After we wrote an assembler code, we can compile it using the "Compile" option under "Build" menu. In case we have errors in our code, list of the errors will appear on the task window, and we will be able to fix it.



Code With Errors – The Errors appears on the Task Window

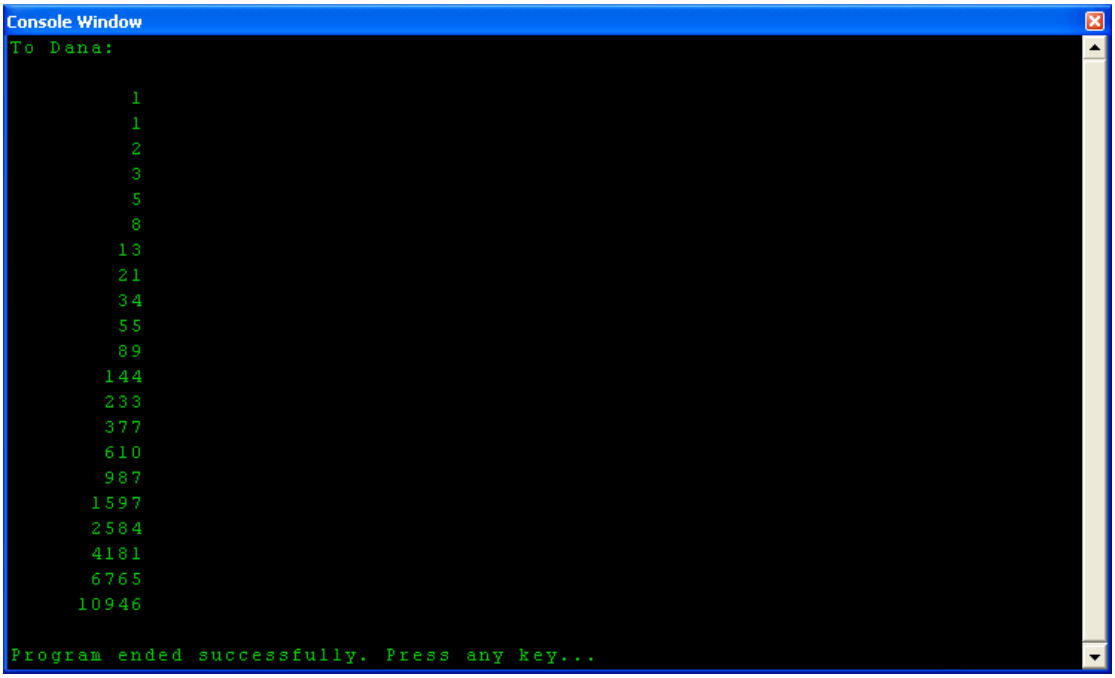
LST files are files contain the user code and the machine code of the program.
The working environment lets the user watch and save the LST files.
To do so, we need to press on "View LST File" option under the "Build" menu.
Note that we can view that file even if we have errors on our code.


VAX11 Simulator - Version 0.57
File Edit Build Debug View Tools Help
In/Out
fib.asm fib.lst
Symbols Table:
Line Symbol Name Value (Dec) Value (Hex)
7 main 0 0
37 format 91 5B
4 LINES 46 2E
36 todana 81 51
27 loop 52 34
19 fib 19 13
Adr Machine Code Source Code
00: 1 #fibonachi series
00: 2
00: 3 .text
00: 4 .set LINES, 46
00: 5
00: 6 # start of the program
00: 00 00 7 main: .word 0
02: DF 00 00 00 00 00 8 pushal todana
08: FB 01 9F FC FF FF 00 9 calls $1, .puts # print "To Dana:"
0F: DD 2E 10 pushl $LINES # number of lines that will be printed must be greater then 0
Error (in line 11): Unrecognized opcode name
11 caglls $1, fib # call to fib function that print the correct numbers
11: DD 00 12 pushl $0
Error (in line 13): Unrecognized procedure
13 calls $1, .exgit # exit from the program
Task List
Description
Line
Output
Task List
Ready
Ln 25 Col 42

LST File

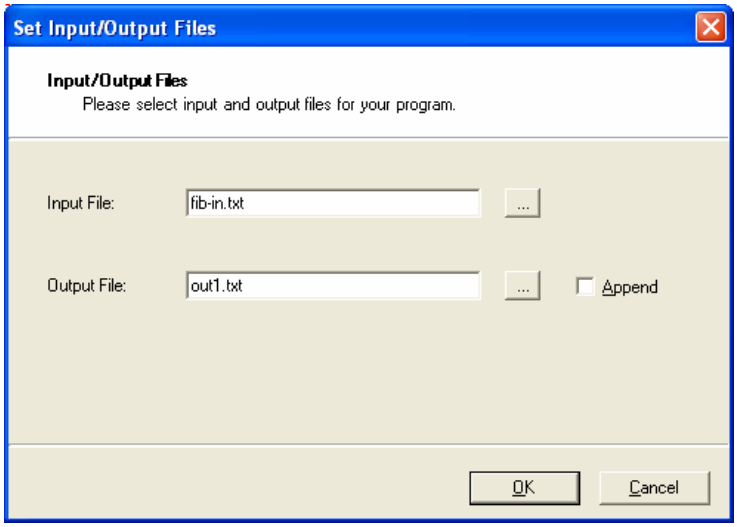
4.7. The Simulator

The simulator is able to execute VAX11 programs. It simulate many hardware aspects, including memory, registers, interrupts, exceptions and more.
In order to execute our program, we need to select "Execute" option from the "Build" menu. After we will press that option, a console window will appear and our program will start.



Console window containing program that displays Fibonacci numbers

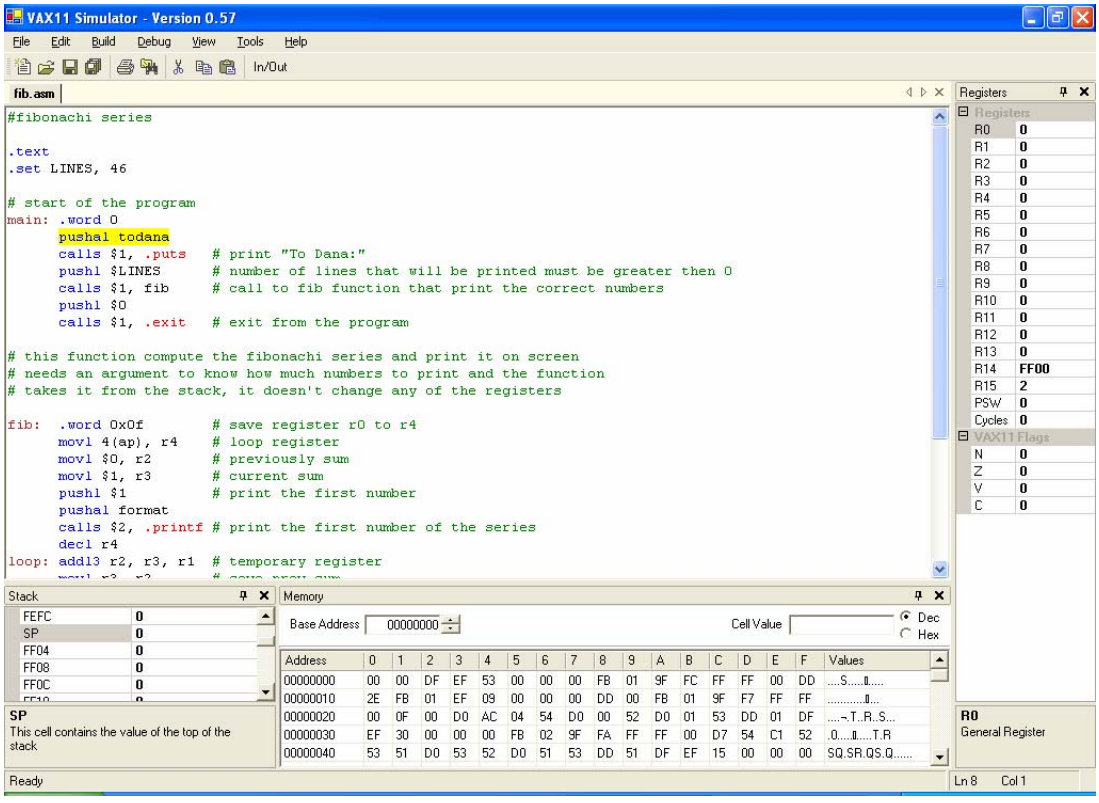
The simulator let us selecting input and output files for out program, using " Set Input/Output Files" in the "Build" menu.
Pay attention that the input and output files can be set for each open document separately.
Append option let us adding output to existing file.



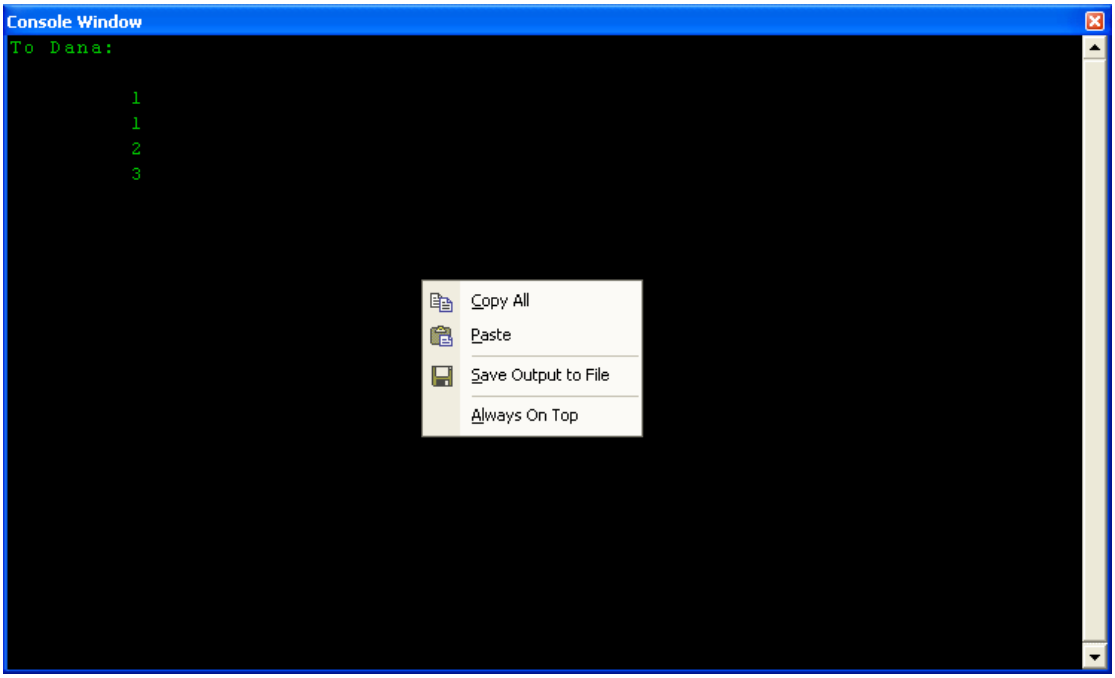
Set Input/Output Files Window

The debug menu allows us running the program in Step-By-Step mode, in order to fix problems in our program. Starting the program in Step-By-Step mode is done by pressing on "Step" option under the "Debug" menu.

When the debug starts several windows will appears to displays the system status: registers windows, memory and stack windows. Also the next line to be performed will be marked using a color.



A console window will be opened too to display the program's output.

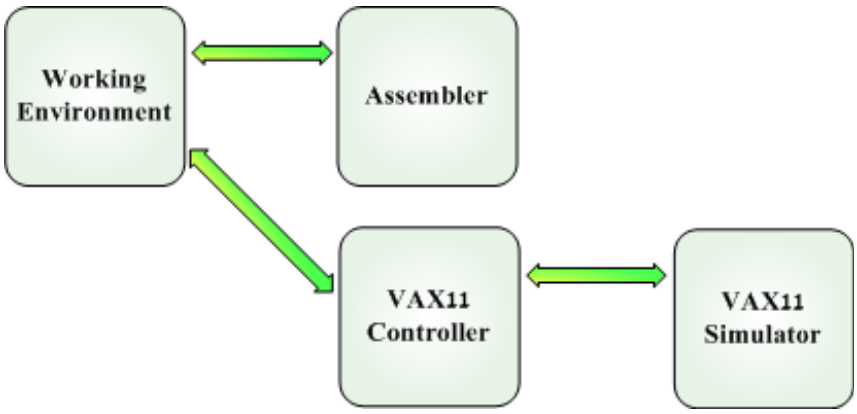


Another useful option is right-clicking on the console. A popup menu will appear with several options: Setting the window as "Always in Top", saving the program's output, and pasting data to the running program.

5. Our Project - Overview

5.1. Project Modules

The project is divided to three main modules: Working Environment, Assembler, and simulator.



5.1.1. The working environment

The working environment contains a source editor. It uses the other modules to provide complete IDE for writing, compiling and testing programs for the VAX-11. The environment is also responsible to manage all the running VAX-11 programs.

5.1.2. The assembler

The assembler module provides services to the environment.
It gets assembly code from the environment, processes it, and returns machine code to the environment. It also manages the symbols table of the program.

5.1.3. The Simulator & the Controller

The simulator and the controller modules manage the running programs.
The simulator module contains all the VAX-11 data - registers, memory, and the current state. The simulator can read opcode from the memory, and analyze it, including all side-effects.

The controller makes the simulator to run. It decides when to command the simulator to execute a command; it sends the simulator information about interrupts. In debug mode, it responsible to give the control to the environment when the simulator reaches a break-point.

6. Settings Class

6.1. Description

Setting class is helping class that contains all the setting of VAX-11 simulator, assembler and working environment.
When the environment is loaded, all the settings are read from the XML file to this class, for use of all the other modules in the project.
The class has several sub classes that indeed to organize the settings.

6.2. Class Structure

Settings class contains three sub classes; each contains setting and constants that relevant to other module in the project:

- Assembler
- Simulator
- Environment

Assembler Settings:

Setting	Values	Meaning
GenerateLstFile	Yes/No	Should the assembler generate LST files while compiling the code?
LstFileAppend	Over/Append	When generating LST file, if there is already file with the same name, should we override it or append the new LST file to it?

Simulator Settings:

Setting	Values	Meaning
StartAddress	int	Address where to load the user's code
TextColor	Colors	The color of the console's output.
BackgroundColor	Colors	Background color for the console.
PageSize	int	Memory Page Size
PhysicalMemorySize	int	Physical Memory Size. The memory size should be multiple of the page size.
ShowMemoryAccesses	True/False	If true, the simulator will display information about accesses to memory after each command.
ShowPhysicalAddresses	True/False	If selected, the simulator will show "physical" addressing for every virtual address.
ShowPageFaults	True/False	If selected, the simulator will display message when page-fault occurs.
FillMemorywithGarbage	True/False	If true, uninitalize memory cells will contain garbage. If false, it will contain zeros

Environment Settings:

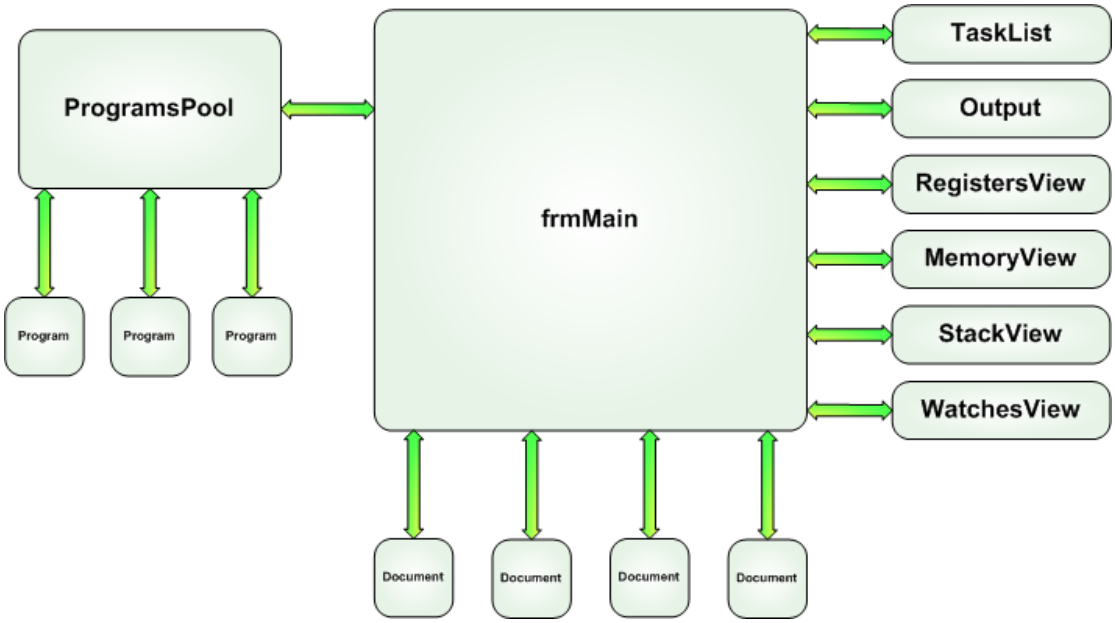
Setting	Values	Meaning
HighlightCode	Yes/No	Should the environment highlight the user's code?
EditorFont	Font	Font size and type that will be used by the environment
HighlightColors	Colors	In what colors to highlight the different text parts
OpenLSTFileAfterCompile	Yes/No	Should the environment open the LST for viewing after the compilation?
TextColor	Colors	Normal Text Color
CommentsColor	Colors	Comments start after # sign. The setting let the user define its colors
LabelsColor	Colors	Labels are identifiers following by : that appears on start of lines.
SystemCallsColor	Colors	The simulator supports some high-level functions, as printf, getchar, etc. These functions are known as operation system functions.
DirectiveColor	Colors	Directives are commands meant for the assembler, that doesn't appear on the final machine code. Examples: .word, .space
CommandsColor	Colors	VAX11 commands (opcodes) color.
StringsColor	Colors	Strings are text appear betweens "".
BackgroundColor	Colors	Documents Background Color
CurrentLineColor	Colors	Color of the next line that will be executed (debug mode)

Environment Constants

Constant	Value	Meaning
FILE_FILTERS	string	Filters for file saving / loading
MESSAGEBOXS_TITLE	string	Define the title of the different message boxes of the environment
SIM_VERSION	string	Text representing the simulator current version

7. Working Environment

7.1. Overview



frmMain

- The program begins operating from frmMain. This is the primary class of the environment.
- Contains the status bar, the menus, the toolbar, and the container.
- Displays the entire user interface.

TaskList

- Contains warnings and errors massages generated during compile.

Output

- Contains the compile massages.
- Displays messages during the compile process.
- Documents
- Contains the source files of the user.

- Communicate with the main application via events.
- The application might contain multiply instances of this class at the same time.

RegistersView

- Displays the regular registers and their current values.
- Displays the relevant privilege registers.
- Displays the PSW.
- Displays the internal clock counter, if relevant.
- Lets the user change the registers' values during the debug process.

MemoryView

- Contains the simulator's memory content.
- Lets the user change the memory content during the debug process.
- Displays address + 16 bytes in each line.

StackView

- Contains the simulator's stack content.
- Lets the user change the stack content during the debug process.
- Displays the address + 4 bytes in each line

WatchesView

- Contains the user's watches.
- Appears on debug mode, letting the user examine the program.

ProgramsPool

- The simulator allows many programs to run at the same time. ProgramsPool is container class that saves references to all the running processes.

Program

- Manages running program.
- Contains VAX-11 Simulator Object and VAX-11 Controller.

7.2. Environment Classes

7.2.1. frmMain

This is the primary class of the project. The class is responsible for displaying the main application window and managing the user interface.
It has no public methods/fields aside from its constructor.

The following points are related to the internal structure of this class.
In order to minimize dependencies between classes, we access other classes using covering functions. Doing so, we will be able to replace in the future parts of the GUI/Assembler/Simulator without major changes to the code.

Dealing with Task List

Function	Gets	Returns	Throws
AddTask	TaskType, Message, Line Number	Nothing	Nothing
ClearTasks	Nothing	Nothing	Nothing
updateTasksLinesNumber	FromLine, Offset	Nothing	Nothing

AddTask - Add task to the task list.
ClearTasks - Clear all tasks from the task list
updateTasksLinesNumber - Change all line numbers of tasks by offset, starting from given line number.

Event	Gets	Returns	Throws
OnTaskListClick	LineNumber	Nothing	Nothing

OnTaskListClick - Function to handle clicks on tasks list. Not to be called by the user, only by events.

Dealing with Output window

Function	Gets	Returns	Throws
ClearOutputWindow	Nothing	Nothing	Nothing
AppendToOutputWindow	StringtoAppend	Nothing	Nothing

ClearOutputWindow - Clears the output window

AppendToOutputWindow - Append text to output window

Dealing with Documents

Function	Gets	Returns	Throws
GetActiveDocumentIndex	Nothing	int	NoActivePageException
OnDocumentPosition	Point	Nothing	Nothing

GetActiveDocumentIndex - Get the page index of the active document

OnDocumentPosition - Update the status bar location when caret changes position

Dealing with Register List

Function	Gets	Returns	Throws
AddRegister	RegName, Value	Nothing	Nothing
ClearAll	Nothing	Nothing	Nothing
updateReg	RegName, Value	Nothing	Nothing

AddRegister - Add register to the list.

ClearAll - Clear all from the register list

updateReg - Change the value of the register

Event	Gets	Returns	Throws
OnRegListClick	LineNumber	Nothing	Nothing
OnRegListChange	LineNumber	Nothing	Nothing

OnRegListClick - Function to handle clicks on Reglist. Not to be called by the user, only by events.

OnRegListChange - Function to handle changes on Reglist. Not to be called by the user, only by events.

Dealing with MemoryView window

Function	Gets	Returns	Throws
ClearAll	Nothing	Nothing	Nothing
Update	Nothing	Nothing	Nothing

ClearAll - Clears the Memory window

Update - Display the correct output depend on the position.

Event	Gets	Returns	Throws
OnMemChange	LineNumber	Nothing	Nothing

OnMemChange - Function to handle changes on Memory. Not to be called by the user, only by events.

Dealing with StackView

Function	Gets	Returns	Throws
ClearAll	Nothing	Nothing	Nothing
Update	Nothing	Nothing	Nothing

ClearAll - Clears the Stack window

Update - Display the correct output.

Event	Gets	Returns	Throws
OnStackChange	LineNumber	Nothing	Nothing

OnStackChange - Function to handle changes on Stack. Not to be called by the user, only by events.

Interface - Helping Functions

Function	Gets	Returns	Throws
StatusBarMessage	string	Nothing	Nothing
GiveFocusToTheActiveDocument	Nothing	Nothing	Nothing
ArrangeControls	Nothing	Nothing	Nothing
ClearDocumentInfoStatusBar	Nothing	Nothing	Nothing

StatusBarMessage - Helping function for sending messages to status bar

GiveFocusToTheActiveDocument - Gives the focus to the active document

ArrangeControls - Update controls when the window size is changing

ClearDocumentInfoStatusBar - Clears document info form status bar

Files - Creating, loading, saving and closing

Function	Gets	Returns	Throws
CreateNewDocument	Nothing	Nothing	Nothing
DoOpen	Nothing	Nothing	Nothing
PrepareDocumentForClosing	frmToClose	bool	Nothing
DoSave	frmToSave	bool	Nothing
InterfaceDoSave	Nothing	bool	Nothing
InterfaceDoSaveAs	Nothing	bool	Nothing
InterfaceDoClose	Nothing	bool	Nothing
CloseAllDocuments	Nothing	bool	Nothing

CreateNewDocument - Create new document

DoOpen - Open assembly file

PrepareDocumentForClosing - Preparing document to be close. After calling to this function, we need to remove the page tab from the documents list.

DoSave - Save a document.

InterfaceDoSave - Interface Save - Actions to do when the user press on "Save" button

InterfaceDoSaveAs - Interface Save As - Actions to do when the user press on "Save As" button

InterfaceDoClose - Close the active document

CloseAllDocuments - Close all open documents

7.2.2. frmTaskList

This class is responsible to display the task list. It contains the errors and warnings the user received during the code compilation.
The class raise event - *OnClickEvent*, when user double click on list item, announcing the code line that related to the clicked entry.

Public Functions:

Function	Gets	Returns	Throws
AddTask	TaskType, Message, Line Number	Nothing	Nothing
ClearTasks	Nothing	Nothing	Nothing
updateTasksLinesNumber	FromLine, Offset	Nothing	Nothing

AddTask - Add task to the task list.
ClearTasks - Clear all tasks from the task list
updateTasksLinesNumber - Change all line numbers of tasks by offset, starting from given line number.

7.2.3. frmCompileMessages

This class is responsible to display the output window - contains messages arrived during compile time.

Public Properties:

Output - Get/Set the text in the output window

7.2.4. frmEditor

This class represents a single document. It actually contains the document.
It responsible to display the document and giving interface functions for the main program to deal with it.
In the main program there may be many instances of this class.
The class raise event - *OnPositionChange* - when the caret position is changing.

Public Functions:

Function	Gets	Returns	Throws
Cut	Nothing	Nothing	Nothing
Copy	Nothing	Nothing	Nothing
Paste	Nothing	Nothing	Nothing
SelectAll	Nothing	Nothing	Nothing
FocusDocument	Nothing	Nothing	Nothing

Cut - Cut text
Copy - Copy text
Paste - Paste text
SelectAll - Select all text
FocusDocument - Give focus to the document

Public Properties:

bDocumentSaved - Did we save the current document
sFileName - File Name (including path)

Public Events:

OnPositionChange - Occurs when caret's position is changing

Private Functions:

Function	Gets	Returns	Throws
GetLocationOnRTB	RichTextBox	Point	Nothing

GetLocationOnRTB - Returns the line and the column of the caret.

7.2.5. RegistersView

The class is responsible to display the registers and their current values. It lets the user change the registers' values during the debug process.
The class sends event to the environment - *OnUserChangeRegister*, to notify the environment if the user decided to change the value of a register.
The class listens to registers changes, and updates it display according to it.
Most of the class work is internal:
When register changes, the class automatically updates its display, and color the register that changed.
The class let the user change a register value, and makes all the relevant updates.

Public Functions:

Function	Gets	Returns	Throws
AddRegister	Register Reference	Nothing	Nothing
ColorAllRegisters	Color	Nothing	Nothing

AddRegister - Add register to the display.

ColorAllRegisters - Change the color of all the registers. We use it after executing of every command, to erase the changes of the previous command.

7.2.6. MemoryView

The class is responsible to display the simulator's memory contents.
It let the user change the memory content during the debug process.
It display address + 16 bytes in each line
The class sends event to the environment - *OnUserChangeMemory*, to notify the environment if the user decided to change the value of a memory cell.

Public Functions:

Function	Gets	Returns	Throws
SetMemorySource	Memory Reference	Nothing	Nothing
DisplayMemory	Starting Address, End Address	Nothing	Nothing

SetMemorySource - Gets reference to Memory object, which the memory's data is in.

DisplayMemory - Displays specific address of the memory.

7.2.7. StackView

The class displays the simulator's stack content. It let the user change the stack content during the debug process.
It displays address + 4 bytes in each line.
The class listens to SP changes, and updates it display according to it.

Public Functions:

Function	Gets	Returns	Throws
SetMemorySource	Memory Reference	Nothing	Nothing
SetSPSource	Regiser Reference	Nothing	Nothing

SetMemorySource - Gets reference to Memory object, which the memory's data is in.

SetSPSource - Gets SP register which relevent to the memory object.

7.2.8. WatchesView

The class contains the user's watches. Appears on debug mode, letting the user examine the program.
Communicate with the controller, to evaluate the watches after each command.

Public Functions:

Function	Gets	Returns	Throws
AddWatch	Watch	Nothing	Nothing
DelAllWatchs	Nothing	Nothing	Nothing

AddWatch - Add watch to the list

DelAllWatchs - Del all watches

Event	Gets	Returns	Throws
OnWatchesClick	LineNumber	Nothing	Nothing
OnWatchesChange	LineNumber	Nothing	Nothing

OnWatchesClick - Function to handle user changes like add watch or delete one.

OnWatchesChange - Function to handle updates on watches. Not to be called by the user, only by events.

7.2.9. ProgramsPool

The simulator allows many programs to run at the same time.
ProgramsPool is container class the saves references to all the running processes.
Programs in the programs pool might enter debug mode, but only one application at time can be in debug mode.
ProgramsPool maintains all the simulators running. It responsible to create new programs and run it.

Public Functions:

Function	Gets	Returns	Throws
AddApplicationToDebugMode	CodeBlock, Start Location	Program Reference	AlreadyInDebugMode
AddApplicationToPool	CodeBlock, Start Location	Nothing	Nothing
KillAllPrograms	Nothing	Nothing	Nothing
StopDebug	Nothing	Nothing	NoApplicationInDebugMode

AddApplicationToDebugMode - Create new Program class. Connect between the application to the environment to display its status, and to give the environment control on its running.

AddApplicationToPool - Create new Program class, run it and add it to the pool. The program runs as separate thread

KillAllPrograms - Kill all running programs.

StopDebug - Stop the debugged program. Free its memory.

7.2.10. Program

Manage running program.

Public Functions:

Function	Gets	Returns	Throws
Constructor	CodeBlock, Start Address, IsDebug.	Nothing	Nothing
GetController	Nothing	VAX11Controller	Nothing
Run	Nothing	Nothing	Nothing

Constructor - Create new program, open the console. Create new controller.

GetController - Returns the VAX-11 Controller of the program.

Run - Run the application.

Public Events:

ProgramEnd - - Program is about to end.

7.2.11. BreakPointList, BreakPointEntry (Class)

Field	Type	Comments
Position	Int	Absolute value
TimeToLive	int	Times before removing the breakpoint

Class Methods:

Operation	Gets	Returns	Throws
EmptyList	Nothing	Nothing	Nothing
AddEntry	New BreakPointEntry	Nothing	Breakpoint exists
RemoveEntry	BreakPointEntry	Nothing	unknown breakpoint
GetEntry	Position	BreakPointEntry	unknown breakpoint

EmptyList - Creates new breakpoint list.

AddEntry - Add new breakpoint to the table.

RemoveEntry - remove breakpoint from list

GetEntry - Gets position and returns its his entry if exists.

7.3. Simulator Debug Mode's Design

Debug mode offers several options to the user: Step by Step, Run to cursor, etc.
When user select one of those commands, if we didn't start debug mode yet, we doing it by creating special process for debugging, and set its status.
After the process exists, the command that the user requested is appealed on the debugged program.

7.3.1. Flags

The Program class contains several flags for working with debug mode. The value of the flags might be change during the debug.

bInStepMode - True if we're in step mode. If this flag is true, the program will halt after each command and will give the control back to the environment.

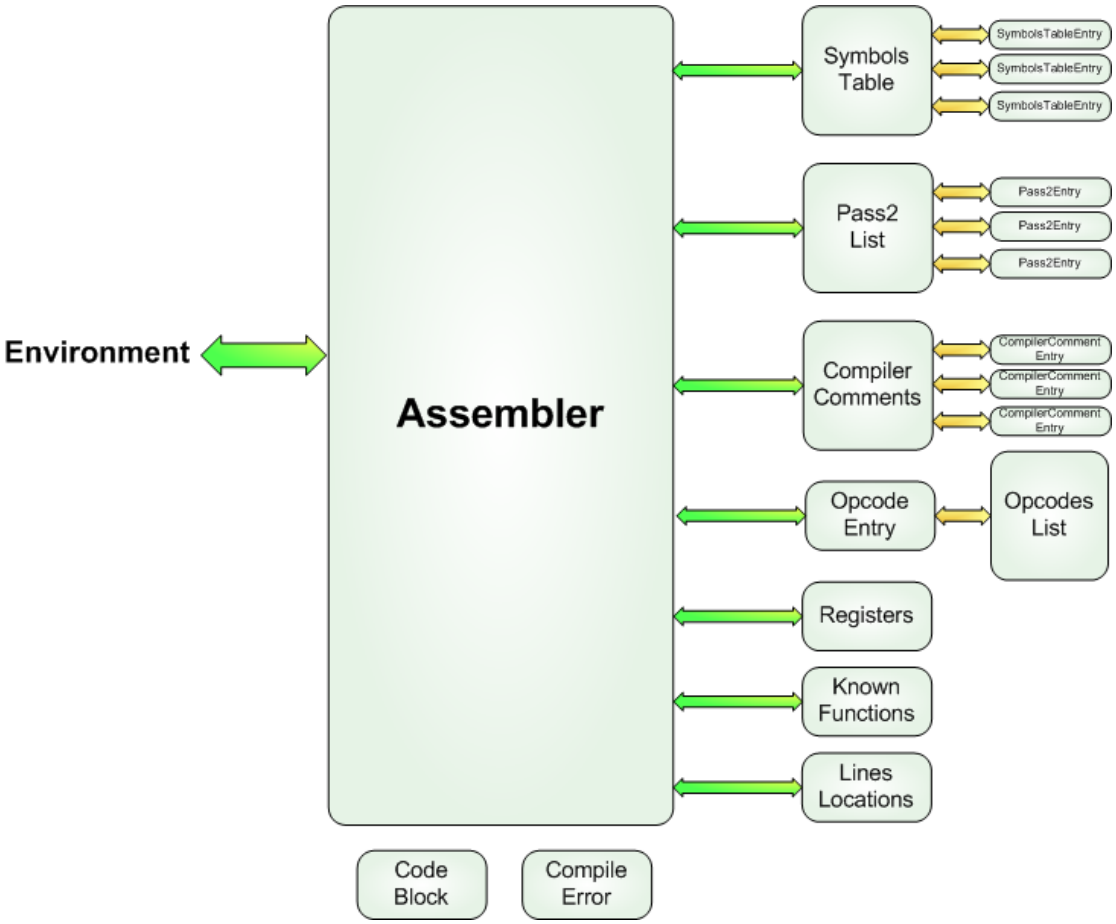
bInDebugMode - True if the current program is the debugged program, false elsewhere.

7.3.2. Semaphores

BreakPointWait - The program waits on this semaphore when it reaches break point or when it needs to wait in step mode.

8. Assembler's Class & Algorithms

8.1. Overview



8.1.1. Data Structures

- Several classes in the project contain VAX-11 specific information. The information includes registers names, known functions addresses and opcodes names and operands.
- The idea is to split the source analyze process to few categories, making the analyzing of each category simple.

KnownFunctions, Registers, Opcode Table

- Registers class contains the VAX-11 registers names.
- KnownFunctions class contains known operation system functions.
- The Opcode Table contains information about all the known opcodes: the opname of each command, its machine code, the number of operands the command needs to get, and also the expected type of each operand.
- Implemented using Hash Table to achieve O(1).

8.1.2. Classes

Assembler

- The main class of this module.
- The only class that communicate with the working environment
- Gets assembly source code and generate machine code
- Communicate with the environment using events to inform it when it found errors in the code.

SymbolsTable, SymbolsTableEntry

- VAX-11 Assembler allows the users to use symbols - labels and constants, to make the code more readable.
- While analyzing the code, the compiler collects all the symbols it found in the Symbols Table.

- When the user uses a symbol, the compiler is using the symbols table to find the value of the symbol.
- Implemented using Hash Table to achieve O(1).

Pass2List, Pass2Entry

- The user may use symbols that will appear later in the code. In this case, the compiler cannot find the value that the symbol represents.
- In such cases, the compiler adds all the unknown symbols to Pass2List. After the first pass on the code, the compiler moves on this list, and checks if it can solve the symbols again.
- Implemented using Hash Table to achieve O(1).

CompilerComment, CompilerCommentEntry

- While compiling the user's code, the compiler might find syntax errors.
- CompilerComment class is class that maintains the list of all the compile errors.

8.1.3. Algorithm

The compiling process is divided to several parts:

- PreCompiler - In order to make the analyzing of the code more simple - we first do some preparing processing on the user's code. The PreCompiler cleans the user's comments, and clean white spaces from the beginning of each line, from its end and between the words.
- Pass 1 - First analyzing of the code. Creates symbols table, Creates table of the missing symbols for the second pass. Convert all the opnames and the operands to machine code.
- Pass 2 - After the first pass, the compiler moves again on all the expressions it had problems to translate in the first pass, and tries to translate it again.
  If it succeeds to translate all the expressions, then the compile process is over.

8.2. Data Structures

8.2.1. Opcodes Table

The opcodes table contains information about the different operands the assembler recognizes.
The opcodes table is constant list - it never changes and all its content is known.

Each entry in the table has the following fields:

Field	Type	Comments
OpName	string	Name of the operation
OpCode	int	Opcode of the operation
Operands	int	Number of expected operands
OpType	string	Operands types

The format of the OpType field is 2 characters per operand (max 6 operands).

1st character:

- a - address
- b - immediate value (of the 2nd type)
- r - read access
- w - write access
- p - privilege register

2nd character:

- 1 - byte
- 2 - word
- 4 - long
- 8 - quad

8.2.2. Known Procedures Table

The known procedures table contains the name and the simulator addresses of the known procedures of VAX-11 - printf, malloc, exit, etc.
Each entry in the table has the following fields:

Field	Type	Comments
Name	string	Procedure's Name
Address	int	Address of the symbol

8.2.3. Known Registers Table

The known registers table contains the name and the numbers of the known registers of VAX-11.
Each entry in the table has the following fields:

Field	Type	Comments
Name	string	Register's Name
Number	int	Register Number

8.2.4. Assembler Messages

The following table presents the different error messages our assembler generates.

Message Number	Message
0	Compile Succeed
1	Compile Failed
2	Number expected
3	Unexpected end of line
4	Label already defined
5	Double label in line
6	Illegal addressing mode
7	Number is too big
8	Register expected
9	Symbol expected
10	Register is not allowed here
11	Unrecognized procedure
12	) expected
13	Unrecognized opcode name
14	Immediate addressing mode cannot be destination
15	] expected
16	Rn must be different than Rx
17	End of operand expected
18	End of line or comment expected
19	Using of Literal/Immediate addressing mode is not allowed here. The opcode require using of other addressing modes
20	Unrecognized Directive
21	Defined symbol or number expected
22	" expected
23	Value (number or symbol) expected
24	, expected
25	PC cannot be used here
26	Displacement too big
27	Procedure not allowed here
28	Divide by zero
29	Undefined symbol
30	Program must begin with .TEXT
31	Privileged register cant be used here
32	Illegal expression
33	Expected: Privileged register
34	Illegal string format - \ used with wrong control char
35	Illegal string format - Single " appears in the middle of the string
36	ORG directive can't be used to jump backward
37	Negative numbers can't be assigned to .WORD. Consider using of .INT instead
38	The given value is out of .INT maximum range (-32,768 .. 32,767). Consider using of .LONG instead
39	Can't declare strings with more than 255 characters using .ASCIC
40	The given value is out of signed .BYTE maximum range (-128 .. 127). Consider using of .INT instead
41	Number is too small
42	Can't define two entry points for the program. Please remove one of the definitions
43	The entry point specific isn't valid label or address

8.2.5. Known VAX-11 Functions:

Function Name	Address
GETCHAR	0xffff
PUTCHAR	0xfffe
GETS	0xfffd
PUTS	0xfffc
SCANF	0xfffb
PRINTF	0xfffa
MALLOC	0xfff9
EXIT	0xfff8
FREE	0xfff7
UNGETCHAR	0xfff6

8.2.6. VAX-11 Register List

Register Name	Register Number
R1	1
R2	2
R3	3
R4	4
R5	5
R6	6
R7	7
R8	8
R9	9
R10	10
R11	11
R12	12
R13	13
R14	14
R15	15
AP	12
FP	13
SP	14
PC	15
SCBB	17
IPL	18
SIRR	20
SISR	21
ICCS	24
NICR	25
ICR	26
OSC	30
RXCS	32
RXDB	33
TXCS	34
TXDB	35

8.2.7. VAX-11 commands

Opname	Number of Operands	OpType	Opcode
ACBA	4	r1r1w1b2	0x9D
ACBB	4	r1r1w1b1	0x9D
ACBL	4	r4r4w4b2	0xF1
ACBW	4	r2r2w2b2	0x3D
ADDB2	2	r1w1	0x80
ADDB3	3	r1r1w1	0x81
ADDL2	2	r4w4	0xC0
ADDL3	3	r4r4w4	0xC1
ADDW2	2	r2w2	0xA0
ADDW3	3	r2r2w2	0xA1
ADWC	2	r4w4	0xD8
AOBLEQ	3	r4w4b1	0xF3
AOBLSS	3	r4w4b1	0xF2
ASHL	3	r1r4w4	0x78
BBC	3	r4a1b1	0xE1
BBCC	3	r4a1b1	0xE5
BBCS	3	r4a1b1	0xE3
BBS	3	r4a1b1	0xE0
BBSC	3	r4a1b1	0xE4
BBSS	3	r4a1b1	0xE2
BCC	1	b1	0x1E
BCS	1	b1	0x1F
BEQL	1	b1	0x13
BEQLU	1	b1	0x13
BGEQ	1	b1	0x18
BGEQU	1	b1	0x1E
BGTR	1	b1	0x14
BGTRU	1	b1	0x1A
BICB2	2	r1w1	0x8A
BICB3	3	r1r1w1	0x8B
BICL2	2	r4w4	0xCA
BICL3	3	r4r4w4	0xCB
BICPSW	1	r2	0xB9
BICW2	2	r2w2	0xAA
BICW3	3	r2r2w2	0xAB
BISB2	2	r1w1	0x88
BISB3	3	r1r1w1	0x89
BISL2	2	r4w4	0xC8
BISL3	3	r4r4w4	0xC9
BISPSW	1	r2	0xB8
BISW2	2	r2w2	0xA8
BISW3	3	r2r2w2	0xA9
BITB	2	r1w1	0x93
BITL	2	r4w4	0xD3
BITW	2	r2w2	0xB3

BLBC	2	r4b1	0xE9
BLBS	2	r4b1	0xE8
BLEQ	1	b1	0x15
BLEQU	1	b1	0x1B
BLSS	1	b1	0x19
BLSSU	1	b1	0x1F
BNEQ	1	b1	0x12
BNEQU	1	b1	0x12
BPT	0		0x03
BRB	1	b1	0x11
BRW	1	b2	0x31
BSBB	1	b1	0x10
BSBW	1	b2	0x30
BVC	1	b1	0x1C
BVS	1	b1	0x1D
CALLG	2	a1a1	0xFA
CALLS	2	r4a1	0xFB
CASEB	3	r1r1r1	0x8F
CASEL	3	r4r4r4	0xCF
CASEW	3	r2r2r2	0xAF
CLRB	1	w1	0x94
CLRL	1	w4	0xD4
CLRW	1	w2	0xB4
CMPB	2	r1r1	0x91
CMPC3	3	r2a1a1	0x29
CMPC5	5	r2a1r1r2a1	0x2D
CMPL	2	r4r4	0xD1
CMPW	2	r2r2	0xB1
CVTBL	2	r1w4	0x98
CVTBW	2	r1w2	0x99
CVTLB	2	r4w1	0xF6
CVTLW	2	r4w2	0xF7
CVTWB	2	r2w1	0x33
CVTWL	2	r2w4	0x32
DECB	1	w1	0x97
DECL	1	w4	0xD7
DECW	1	w2	0xB7
DIVB2	2	r1w1	0x86
DIVB3	3	r1r1w1	0x87
DIVL2	2	r4w4	0xC6
DIVL3	3	r4r4w4	0xC7
DIVW2	2	r2w2	0xA6
DIVW3	3	r2r2w2	0xA7
EDIV	4	r4r4w4w4	0x7B
HALT	0		0x00
INCB	1	m1	0x96
INCL	1	m4	0xD6
INCW	1	m2	0xB6
INSQUE	2	a1a1	0x0E
JCC	1	b2	0x31031F
JCS	1	b2	0x31031E
JEQL	1	b2	0x310312

JEQLU	1	b2	0x310312
JGEQ	1	b2	0x310319
JGEQU	1	b2	0x31031F
JGTR	1	b2	0x310315
JGTRU	1	b2	0x31031B
JLBC	2	r4b2	0x3103E8
JLBS	2	r4b2	0x3103E9
JLEQ	1	b2	0x310314
JLEQU	1	b2	0x31031A
JLSS	1	b2	0x310318
JLSSU	1	b2	0x31031E
JMP	1	a1	0x17
JNEQ	1	b2	0x310313
JNEQU	1	b2	0x310313
JSB	1	a1	0x16
JVC	1	b2	0x31031D
JVS	1	b2	0x31031C
LOCC	3	r1r2a1	0x3A
MATCHC	4	r2a1r2a1	0x39
MCOMB	2	r1w1	0x92
MCOML	2	r4w4	0xD2
MCOMW	2	r2w2	0xB2
MFPR	2	p4w4	0xDB
MNEGB	2	r1w1	0x8E
MNEGL	2	r4w4	0xCE
MNEGW	2	r2w2	0xAE
MOVAB	2	a1w1	0x9E
MOVAL	2	a4w4	0xDE
MOVAW	2	a2w2	0x3E
MOVB	2	r1w1	0x90
MOVC3	3	r2a1a1	0x28
MOVC5	5	r2a1r1r2a1	0x2C
MOVL	2	r4w4	0xD0
MOVPSL	1	w4	0xDC
MOVTC	6	r2a1r1a1r2a1	0x2E
MOVTUC	6	r2a1r1a1r2a1	0x2F
MOVW	2	r2w2	0xB0
MOVZBL	2	r1w4	0x9A
MOVZBW	2	r1w2	0x9B
MOVZWL	2	r2w4	0x3C
MTPR	2	r4p4	0xDA
MULB2	2	r1w1	0x84
MULB3	3	r1r1w1	0x85
MULL2	2	r4w4	0xC4
MULL3	3	r4r4w4	0xC5
MULW2	2	r2r2	0xA4
MULW3	3	r2r2w2	0xA5
NOP	0		0x01
POPR	1	r2	0xBA
PUSHAB	1	a1	0x9F
PUSHAL	1	a4	0xDF
PUSHAW	1	a2	0x3F

PUSHL	1	r4	0xDD
PUSHR	1	r2	0xBB
REI	0		0x02
REMQUE	2	a1w4	0x0F
RET	0		0x04
ROTL	3	r1r4w4	0x9C
RSB	0		0x05
SBWC	2	r4w4	0xD9
SCANC	4	r2a1a1r1	0x2A
SKPC	3	r1r2a1	0x3B
SOBGEQ	2	w4b1	0xF4
SOBGTR	2	w4b1	0xF5
SPANC	4	r2a1a1r1	0x2B
SUBB2	2	r1w1	0x82
SUBB3	3	r1r1w1	0x83
SUBL2	2	r4w4	0xC2
SUBL3	3	r4r4w4	0xC3
SUBW2	2	r2w2	0xA2
SUBW3	3	r2r2w2	0xA3
TSTB	1	r1	0x95
TSTL	1	r4	0xD5
TSTW	1	r2	0xB5
XORB2	2	r1w1	0x8C
XORB3	3	r1r1w1	0x8D
XORL2	2	r4w4	0xCC
XORL3	3	r4r4w4	0xCD
XORW2	2	r2w2	0xAC
XORW3	3	r2r2w2	0xAD

8.3. Assembler Class and Internal Data Structures

8.3.1. Assembler Class Interface

The following table presents the public methods of the assembler class:

Operation	Gets	Returns	Throws
Constructor	Nothing	Nothing	Nothing
GetSymbolsTable	Nothing	Code's symbols table	Nothing
GetCompileMessages	Nothing	Compile Comments	Nothing
CompileCode	Assembly Code	Machine Code	CompileError
GetMachineCode	Nothing	Machine Code	Nothing
GetLSTFile	Nothing	The LST File	Nothing

Constructor - Initialize the assembler.

CompileCode - Gets code and compile it. This is the main function of this module.

GetSymbolsTable - Returns the symbols table generated during the compiling process.

GetCompileMessages - Returns the compiling errors.

GetMachineCode - Gets the machine code that the assembler created.

GetLSTFile - Generate and create LST file for the compiled code. LST file is file that mix the source code with the machine code, allowing the user to see the compiler output.

8.3.2. CodeBlock (Class)

In order to pass code of blocks from the different functions to the main compile process, we use CodeBlock objects, that contains the machine code we want to pass and its size. CodeBlock is sub-class of Assembler class.

Class Properties:

Property	Type	Comments	Gets / Sets
MachineCode	Byte[]	MachineCode	Get / Set
Size	int	Size of code	Get

MachineCode - Contains machine code, as array of bytes

Size - Machine Code size, in bytes.

Methods:

Operation	Gets	Returns	Throws
Operator+	CodeBlock	CodeBlock	Nothing
Byte Casting	Byte	CodeBlock	Nothing
Int Casting	Int	CodeBlock	Nothing
Constructor 1	Nothing	Nothing	Nothing
Constructor 2	Number, BlockSize	Nothing	IllegalBlockSize
ChangeCodeblock	Position, CodeBlock	CodeBlock	IndexOutOfRange

Constructor 1 - Creates an empty CodeBlock

Constructor 2 - Creates CodeBlock that contains Number, in size of BlockSize
Example: CodeBlock(15,4) → 0F 00 00 00

Operator+ - Overloaded operator +. Allowing us to sum several BlockCode, to create bigger CodeBlock that contains them.

Byte Casting - Allows casting from byte to CodeBlock

Int Casting - Allows casting from int to CodeBlock

ChangeCodeblock - Changes given CodeBlock

8.3.3. Opcode Entry (Class)

When the program wants to get information from the opcodes table, it creates object from Opcode Entry, which contains the request information from the table.

Class Fields: Same as those of the opcodes table.

Class Methods:

Operation	Gets	Returns	Throws
Constructor	OpName	Nothing	UnrecognizedCommand

Constructor - Gets opname and gets all the information related to that opname.

Class Properties

Property	Type	Comments	Gets / Sets
OpCode	int	Opcode of the operation	Get
Operands	int	Number of expected operands	Get
OpType	string	Operands types	Get

8.3.4. Symbols Table (Class)

The symbols table contains information about all the labels defined in the source file. The information is entered to the table during pass 1 of the assembler. Each entry in the table has the following fields:

Field	Type	Comments
Name	string	Symbol's Name
Value	int	Address of the symbol
Type	enum	Constant / Label
Line	int	The line number the symbol defined at

Class Methods:

Operation	Gets	Returns	Throws
Constructor	Nothing	Nothing	Nothing
AddEntry	New symbol table entry	Nothing	LabelAlreadyExists
ResetTable	Nothing	Nothing	Nothing
SymbolValue	string - name	int - value	unknown symbol

Constructor - Creates new symbol table.

AddEntry - Add new symbol to the table.

ResetTable - Resets all symbols table entries.

SymbolValue - Gets symbols and returns its value.

8.3.5. LinesLocations (Class)

For debug (breakpoints) support and for making the LST file, the assembler saves the starting address of each line in the code.

Class Methods:

Operation	Gets	Returns	Throws
Constructor	Nothing	Nothing	Nothing
ResetLines	Nothing	Nothing	Nothing
AddLine	Line, Address	Nothing	Nothing
GetStartingAddress	Line Number	Starting Address	NoSuchLine

Constructor - Initialize the lines list.

AddEntry - Add new entry to the list.

ResetLines - Resets all list entries.

GetStartingAddress - Gets line and returns its starting address.

8.3.6. CompilerComment (Class)

When returning compile messages, the assembler returns array of CompilerComment objects.

Class fields:

Field	Type	Comments
MessageNumber	int	Number of message
Line	int	The relevant line number (or -1)

8.3.7. Pass2List (Class)

We create the list on Pass1. The list is being used on Pass2.
The list contains the locations that pass2 need to change and some information about the needed changes.

Class fields:

Field	Type	Comments
Where	int	LC - Location where the update need to take place
Size	short int	How many bytes are allowed to be change
Expression	string	The all line of the expression
negative	boolean	True if the number is allowed to be negative

8.4. Assembler Algorithms

8.4.1. General

When we create assembler object, the constructor is responsible to make it usable.
After we use the object to compile code, we can use *GetSymbolsTable*() and *GetCompileMessages*() to get information about the compiling process.
The *CompileCode*() function starts the analyzing of the user's code. During its execution, it calls to *DoPass1*() and *DoPass2*() that do the compiling process.
To get the compiler's results, we use *GetMachineCode*() and *GetLSTFile*().

8.4.2. Constructor

Gets: Nothing
Returns: Nothing
Throws: Nothing
Description: Initialize assembler object

Algorithm:

1) **Reset** Symbols Table
2) **Reset** Compile Messages
3) **Reset** Machine Code
4) **Reset** Pass2 List

8.4.3. GetSymbolsTable

Gets: Nothing

Returns: Symbols Table

Throws: Nothing

Description: Returns the assembler symbols table

Algorithm:

1) **Return** the internal Symbols Table.

8.4.4. GetCompileMessages

Gets: Nothing

Returns: Compile Comments

Throws: Nothing

Description: Returns the compile time messages

Algorithm:

1) **Return** the internal list of complier comments.

8.4.5. GetLstFile

Gets: Nothing

Returns: The LST File

Throws: Nothing

Description: Using the symbols table, the assembly code and the machine code, the function generate LST file for the code and returns it.

Algorithm:

1) RetFile ← Empty Block
2) Add header line to RetFile: Address, Machine Code, Line and Line Source
3) **For** Line = 1 **to** TotalNumberOfLines
4) Use LinesLocationTable to find the machine code and the address that releated to the current line.
5) Add the current line with all the relevant fields to RetFile.

8.4.6. GetMachineCode

Gets: Nothing

Returns: Machine Code

Throws: Nothing

Description: Returns the machine code generated by the compiler

Algorithm:

1) **Return** the machine code.

8.4.7. CompileCode

Gets: Assembly Code

Returns: Nothing

Throws: CompileError

Description: Compile the code, returns the machine code

Algorithm:

1) **Reset** all internal data structures.
2) **Save** original code, for future use
3) **If** there is no code, **then return** empty block.
4) **Try**:
5) sCleanCode ← PreCompiler(Assembly Code)
6) TempBlockCode ← DoPass1(sCleanCode)
7) MachineCode ← DoPass2(TempBlockCode)
8) **Catch:** (Compile Errors)
9) Create CompileMessages array contains the errors.
10) **Throw** CompileError

8.4.8. PreCompiler

Gets: Assembly Code

Returns: Clean Assembly Code

Throws: Nothing

Description: Clean the code from comments and spaces

Comment: Be careful not to clean empty lines, so line number will stay untouched.

Algorithm:

1) **While** not end of source
2) CurLine ← Read Line
3) **While** not end of line
4) **If** found '#' on line, **then** cut from it till end of line.
5) Clean spaces between the words.
6) Add the line to the block we return
7) **Return** ReturnedBlock

8.4.9. DoPass1

Gets: CodeBlock

Returns: CodeBlock

Throws: CompileError

Description: Move first time on the code, build symbols table

Algorithm:

```
1)  LC ← 0
2)  LineNumber ← 0
3)  ErrFlag ← false
4)  FinalCode ← Empty Block
5)  CurWord ← ReadNextWord(LinePointer)
6)  If CurWord is not ".TEXT" then
7)          add CompileError("Program must begin with .TEXT",
                LineNumber)
8)          Throw CompileError
9)  While not reach end of code
10)         LineNumber ← LineNumber + 1
11)         LinePointer ← 0
12)         LabelDefined ← false
13)         Call AddToLinesList(LineNumber, LC)
14)         If reached end of line, then continue
15)         CurWord ← ReadNextWord(LinePointer)
16)         While CurWord ends with ":" (label) do
17)                 If the LabelDefined = True then
18)                         add CompileError("Double label in line",
                                LineNumber)
19)                         ErrFlag ← true
20)                 Else
21)                         LabelDefined ← true
```

```
22)                         Try
23)                             UpdateSymbolsTable(CurWord, LC,
    Lbl)
24)                         Catch (LabelAlreadyExists)
25)                             add CompileError("Label already
    defined",
                                                LineNumber)
26)                             ErrFlag ← true
27)                 CurWord ← ReadNextWord(LinePointer)
28)             If CurWord starts with "." then
29)                 Try
30)
                CodeBlock ← AnalyzeDirective(CurWord,
                                            LC, LinePointer)
31)                     FinalCode ← FinalCode + CodeBlock
32)                     LC ← LC+CodeBlock.Size
33)                 Catch (CompileError)
34)                     add CompileError(CompileError.Error,
                        LineNumber)
35)                     ErrFlag ← true
36)             Else
37)                 Try
38)                     CodeBlock ← AnalyzeCommand(CurWord,
                                    LC,LinePointer)
39)                     FinalCode ← FinalCode + CodeBlock
40)                     LC ← LC+CodeBlock.Size
41)                 Catch (CompileError)
42)                     add CompileError(CompileError.Error,
                        LineNumber)
43)                     ErrFlag ← true
44)             If line not ended then
45)                 add CompileError("End of line or comment expected",
                        LineNumber)
```

```
46)                 ErrFlag ← true
47)   If ErrFlag = true then throw CompileError
48)   Return FinalCode
```

8.4.10. AnalyzeCommand

Gets: WordToAnalyze , LC

Returns: CodeBlock

Throws: CompileError (including error field)

Description: Analyze the command that found at WordToAnalyze,
Returns the machine opcode of the operation in the code block.
Analyze the operands and add it to the block.

Algorithm:

```
1)   CurCommand ← New OpcodeEntry(WordToAnalyze)
     (Might throw exception)
2)   ReturnedCodeBlock.MachineCode ← CurCommand.OpCode
3)   For Counter = 0 to CurCommands.Operands-1 do
4)
     CurOperand ← FetchOperand(CurCommand.OpType[2*Counter],
                  CurCommand.OpType[2*Counter+1]-'0')
          (Might throw CompileError)
5)        ReturnedCodeBlock ← ReturnedCodeBlock +CurOperand
6)        LC ← LC+CodeBlock.Size
7)        If Counter = CurCommand.Operands then break
8)        If ReadNextChar(WordToAnalzye) is not ',' then
9)             Throw CompileError(", expected")
```

8.4.11. FetchOperand

Gets: Expected Operand Type (OpType), Expected Operand Size (OpLen), LC, IsPrivilegedCommand

Returns: CodeBlock

Throws: CompileError (including error field)

Description: Analyze the operand where the given LC is. Returns the machine code for the operand.

8.4.12. AnalyzeDirective

Gets: WordToAnalyze, LC

Returns: CodeBlock

Throws: CompileError

Description: Analyze the directive from WordToAnalyze.

Algorithm:

1) TempBlock ← Empty CodeBlock
2) WordToAnalyze ← MakeLower(WordToAnalyze)
3) **If** WordToAnalyze = ".data" or ".text" or ".org" **then Return** TempBlock
4) **If** WordToAnalyze = ".space" **then**
5) TempNum ← ReadNextNumber(LinePointer)
6) **Return** CodeBlock(0,TempNum)
7) **If** WordToAnalyze = ".set" **then**
8) TempName ← ReadNextWord(LinePointer)
9) **If** TempName is empty **then throw** CompileError("symbol expected")

```
10)         If ReadNextChar isn't "," then throw CompileError (",
    expected")
11)         SymbolTable.AddEntry(TempName,CalcExpersion(WordToA
    nalyze))
12)         Return TempBlock
13)     If WordToAnalyze = ".ascii" then
14)         Return CodeBlock(AnalyzeString(WordToAnalyze))
15)     If WordToAnalyze = ".asciz" then
16)         Return CodeBlock(AnalyzeString(WordToAnalyze) +
    ascii(0))
17)     If WordToAnalyze = ".byte" then
18)         BlockSize ← 8
19)     Else if WordToAnalyze = ".word" then
20)         BlockSize ← 16
21)     Else if WordToAnalyze = ".int" then
22)         BlockSize ← 32
23)     Else if WordToAnalyze = ".long" then
24)         BlockSize ← 32
25)     Else if WordToAnalyze = ".quad" then
26)         BlockSize ← 64
27)     Else
28)         Throw CompileError ("Unrecognized Directive")
29)     Do
30)         TempNum ← CalcNextExpersion(LinePointer)
31)         TempNum ← TempNum modulo 2^BlockSize
32)
    TempBlock ← TempBlock +CodeBlock(TempNum,BlockSize/8)
33)     While (ReadNextChar(WordToAnalyze) = ",")
34)     Return TempBlock
```

8.4.13. ReadNextNumber

Gets: Expression

Returns: A number.

Throws: CompileError("Number expected")

Description: The function reads the next word and convert it to number
The function convert hex/decimal numbers.
If the next word cannot be converted then the function throw exception.

8.4.14. ReadNextWord

Gets: Expression

Returns: A string

Throws: Nothing

Description: The function returns the next word.
Word may contain only big letters, small letters and numbers.

8.4.15. ReadNextChar

Gets: Expression

Returns: A char

Throws: Nothing

Description: The function returns the next char on Expression that is not white spaces.

8.4.16. CalcExpression

Gets: Expression

Returns: A number.

Throws: CompileErrors (Defined symbol or number expected, Value expected, number too big)

Description: This function Gets string with expression contains labels, numbers and operators and solves it. If the string doesn't contains ecuation, it throws exception.
If one of the labels isn't defined, it throw exception, and it
is the caller responsability to add the expression to Pass2List.
If the user tries to divide by zero, it throws exception.
If one of the numbers in the equation is too big, it throws an exception.

8.4.17. AnalyzeString

Gets: Expression

Returns: A string

Throws: CompileError(""\"" expected")

Description: The function read string from "\"" to the first "\"" that don't have before him slash "\\" and replace the codes like "\n" to their meaning. If there is not "\"" at the end or at the beginning of the string then the function throws CompileError(""\"" expected").

8.4.18. DoPass2

Gets: CodeBlock

Returns: CodeBlock

Throws: CompileError

Description: Second pass on the code, generate final machine code
We assume in this point that all the definitions of the constants
(Those who need to be in the symbol table) already defined
on AnalyzeDirective, and now we need to update only expressions
with the constants that need update on the memory

Algorithm:

1) **For each** record on Pass2List do
2) **Try**

```
3)                  value<-- CalcNextExpression (Pass2List.Expression)
4)          Catch ()
5)                  add CompileError(unknown symbol,LineNumber)
6)                  wascatch<--true
7)                  continue //foreach
8)          If overRange(value,Pass2List.negative,Pass2List.size) then
9)                  If Pass2List.negative
10)                         add CompileError(number too big,LineNumber)
11)                 else
12)                         add CompileError(displacement too
    big,LineNumber)
13)         else ChangeCodeblock

            (Pass2List.where,CodeBlock(value,Pass2List.size))
14)  If was catch then Throw ("unknown symbol")
```

8.4.19. overRange

Gets: value, negative allowed, num of bytes

Returns: Boolean

Throws: nothing

Description this function check if the value is over the allowed range that determine from negativity and number of bytes

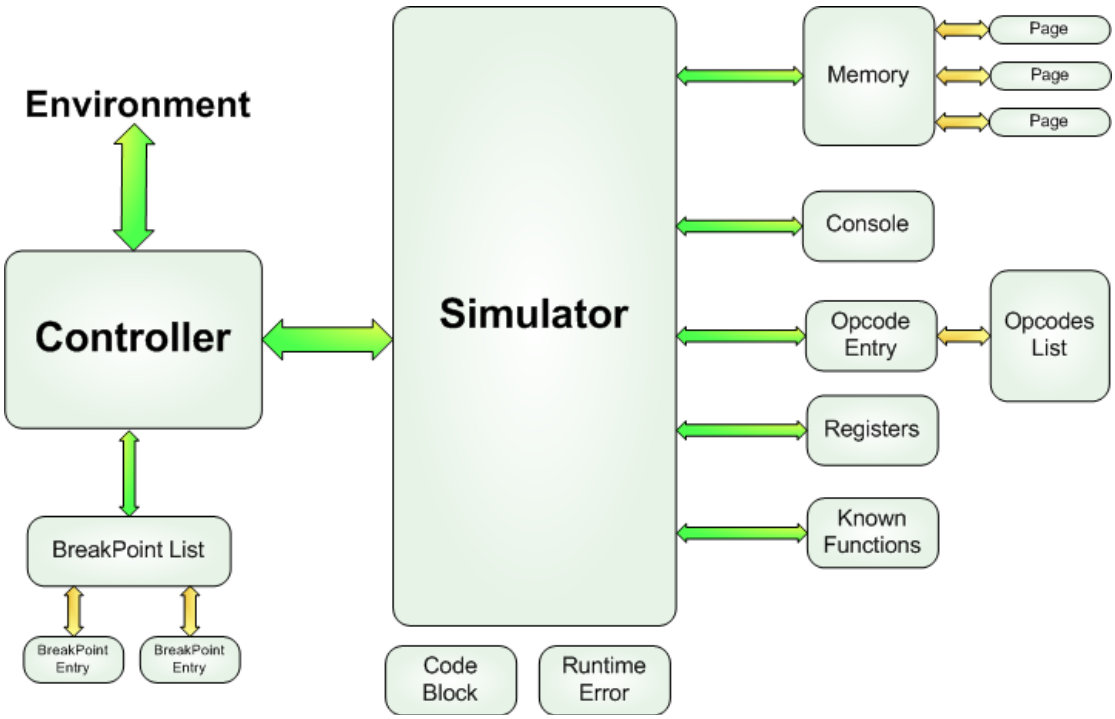
Algorithm:

1) **If** negative allowed and abs(value)< 2^(8*num bytes) **then** return **True**
2) **If** negative not allowed and abs(value)<2^(8*num bytes-1) return **True**
3) **return False**

9. VAX-11 Simulator

9.1. Simulator Class and Internal Data Structures

9.1.1. Classes Overview



Vax11Simulator

- The main class of this module.
- This class contains the other simulator classes and connects between them.
- The class communicates with the Vax11Controller class.
- Gets assembly machine code, where position the code in the memory, breakpoints list, and start running the code.

Vax11Controller

- The only class that communicate with the working environment.
- The class communicate with the environment, with the simulator
- The class communicates with the console, receives events from it and sends it to the simulator.

Console window

- I/O window for running the code.
- The class is handle all the VAX-11 interrupts and sends it to the controller.

Memory

- This class contains the code program, the stack and free memory for allocating new variables.
- Allowing virtual memory of 4GB.
- Work in unit of byte size.

Registers

- This class contains all of the known registers and their values.
- Size of every register is Long, but can work with the first byte or first word of the register

9.1.2. Memory Class

Memory class is class that managed the VAX-11 memory.
It provides up to 4GB virtual memory. From the point-view of the user, the memory is consecutive. The class works with 32 bit addresses.

Class interface:

Methods:

Method	Gets	Returns	Throws
Constructor	Fill, Capacity	Nothing	IllegalSize
Read	Address, Size	CodeBlock	IllegalAddress, IllegalSize
Write	Address, Data (CodeBlock)	Nothing	IllegalAddress

Constructor - Initialize the memory system. *Fill* is the value of all the unallocated locations.
Capacity is the size of the memory (up to 4096MB)
Read - Reads information from the memory, and return it.
Write - Write information to the memory.

Events:

Event	Description
MemoryAccessed	Raised when the memory is accessed. The event returns object contains the address, the data, and if it was read/write operation.

9.1.3. Simulator Class

Description

The simulator class is the heart of the simulator. It contains all of the simulator's data: The memory, the registers and all other information that related to the machine state. To allow debugging, the simulator exposes all that information, and let the user view it and change it.
The main methods of this class are "*PerformNextCommand*" and "*ReceiveInterrupt*". *PerformNextCommand* execute the command where the PC is, including all side effects and operations.
ReceiveInterrupt tells the simulator that specific interrupt was occurred. The simulator can accept or reject the interrupt.

Class methods:

Method	Gets	Returns	Throws
Constructor	CodeBlock	Nothing	
LoadCodeBlockIntoMemory	CodeBlock	Nothing	
PerformNextCommand	Nothing	Nothing	
ReceiveInterrupt	Interrupt	Nothing	

Constructor - Initialize the simulator's data. Make some operations in order to make the analyzing of commands faster. Loads the program's code into the memory.

LoadCodeBlockIntoMemory - Load new program to the memory; Replaces any existing data.

PerformNextCommand - Executes the next opcode; Updates all the relevant machine data.

ReceiveInterrupt - Sends interrupt to the simulator

Property	Type	Get/Set	Comments
Memory	Memory	Get, Set	
Registers	Registers	Get, Set	

9.1.4. Registers (Class)

The class contains the VAX-11 registers. It allows reading and modifying the value of each one of it.

Class Methods:

Operation	Gets	Returns	Throws
GetReg	Register Number	int	nothing
SetReg	Register Number, value	nothing	nothing

GetReg - Gets register and return his value.

SetReg - Set register value.

Event	Description
OnRegisterChanged	Raised when register is changed.

9.1.5. Console

Console is one of the main classes of the simulator. It supports Text & Graphics modes.
It displays the entire VAX-11 Simulator user interface, it receive events and sends it to the controller.

It contains methods for using at graphics mode and methods for using in text mode. If we use the wrong type of command for the current console state, the command is ignored.

9.1.5.1. Text Mode

The screen size on text mode is 80×25.

We saved the previous BUFFER_SIZE lines, and able to scroll to see it.

Methods:

Method	Gets	Returns	Throws
SetCursorLocation	x, y	Nothing	IllegalPosition
Write	Text	Nothing	Nothing
Getchar	Nothing	char	Nothing
Gets	Max Size (or -1)	string	Nothing
ClearScreen	Nothing	Nothing	Nothing
TextColor	Color (0-15)	Nothing	Nothing

9.1.5.2. Graphics Mode

Default screen size is 800×600 pixels. The user can change the size at choice.

Methods:

Method	Gets	Returns	Throws
ClearDevice	Nothing	Nothing	Nothing
Line	x1, y1, x2, y2	Nothing	IllegalPosition
PutPixel	x, y	Nothing	IllegalPosition
Circle	x, y, radius	Nothing	IllegalPosition
Rectangle	x1, y1, x2, y2	Nothing	IllegalPosition
SetColor	color	Nothing	Nothing
Fill	x, y, Color	Nothing	IllegalPosition
GetMaxX	Nothing	Integer	Nothing
GetMaxY	Nothing	Integer	Nothing
OutTextXY	x, y, text	Nothing	IllegalPosition
SetFont	Font, Size, Properties	Result	Nothing

9.1.5.3. General

The following information applied for both graphics and text mode.

Methods:

Method	Gets	Returns	Throws
InitGraph	Nothing	Nothing	Nothing
CloseGraph	Nothing	Nothing	Nothing

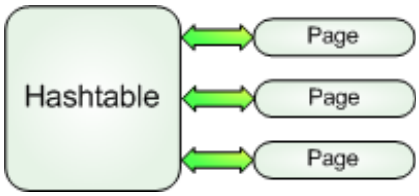
Events:

OnKeyPressed:	Raised when user press on the keyboard.
OnTimer:	Raised every microsecond or tics.
OnPrintCompleted:	Raised when print of character has been completed

9.2. Simulator Algorithms

9.2.1. Memory

We designed the simulator's memory system similar to the way UNIX system manages its memory.
Memory is divided to *pages*. Each page contains 1024 bytes.
We hold hashtable of used pages.
When user reads data, we check if the data is in one of the pages. If so, we return it. Else, we return the memory fill value (0 in our implementation).
When user writes data, we write it on the appropriate page. If the page is not in the hashtable, we creates new one and adds it to the table.
In our implementation, we use CodeBlock objects as pages, and their starting address is their key on the hashtable.



10. Test Programs

In addition to the tests for each opcodes, we test our simulator using "real-life" programs.
The testing programs are included below. Some of the programs were written by us, and some by the "Software System" Technion course's staff.

The programs are the basic tests of our environment. We used it to check our assembler against the old one as well as real-life tests for our simulator.

10.1. Program 1 – “Hello, World”

Official Program written by the "Software System" Technion course's staff.
The program copy chars from keyboard and print it on screen.
It prints "hello world" upon startup.

```
# mtfcopy.asm - copy char from keyboard and print it on screen
# includes printing "Hello, World" at start

      .text                    # program starts here
main: .word 1                  # register mask word
      pushal s
      calls $1, .puts
loop: calls $0, .getchar       # read char returned in r0
      movl r0,ch
      blss fini                # jump to fini when ch<0 - end of input
      pushl ch
      calls $1, .putchar
      jmp loop
fini: pushl $0
      calls $1, .exit

      .data                    # data starts here
s:    .asciz "Hello, World"
ch:   .long 0                  # 32 bit
```

Modified version of the program - including extra testing to .getchar: the program prints on the end the number of chars that were pressed. If the user pressed on 5 chars, and then deleted 2, and then pressed enter, it should display 3.

```
# mtfcopy2.asm - copy char from keyboard and print it on screen
# includes printing "hello world" at start

      .text                    # program starts here
main: .word 1                  # register mask word
      pushal s
      calls $1, .puts
      movl $0, r7
loop: calls $0, .getchar       # read char returned in r0
      movl r0,ch
      blss fini                # jump to fini when ch<0 - end of input
      incl r7
      pushl ch
      calls $1, .putchar
      jmp loop
fini:
      pushl r7
      pushal format
      calls $2, .printf

      calls $0, .getchar

      pushl $0
      calls $1, .exit

      .data                    # data starts here
s:    .asciz "Hello World"
ch:   .long 0                  # 32 bit
format:     .asciz "\n%d\n"
```

10.2. Program 2 – Dump Memory

```
# dump.asm - display the memory contents from address 00 to 32
# with formatting

      .text
main: .word 0
      pushl $0
      pushl $32
      calls $2, dump    #call dump with 2 param: 32 and 0
      pushl $0
      calls $1, .exit   #call .exit with 1 param: 0
#============================================================
dump: .word 0xF         #mask r0-r3
      movl 8(AP), R2    #start address in r2
      movl 4(AP), R3    #total memory dump length in r3
row:  movl $7, R1       #row length counter
      movl $0, R0
col:  movb (R2)[R1], R0
      pushl R0
      sobgeq R1, col
      pushl R2
      pushal format
      calls $10, .printf
      addl2 $8, R2
      subl2 $8, R3
      bgtr row
      ret

      .data
format:     .asciz "%04lx: %02x %02x %02x %02x %02x %02x %02x %02x\n"
```

10.3. Program 3 – Analyzing User Input

```
# Converts all input chars to uppercase and centers the line

      .text
      .set MAXLINE,80               # maximum row length
main: .word 0
loop: pushal inline
      calls $1, .gets               # read a line into inline
      tstl r0                       #r0 = -1 means EOF
      blss exit
      pushal inline
      calls $1, strlen              # returns string length into r1
      movl $MAXLINE,r2              # r2 = max line length
      subl2 r1, r2                  # r1 = margins length
      divl2 $2, r2                  # left margin length
      clrl r0                       # r0 index
margin:
      movb $' ,outline(r0)          # fill the outline with spaces
      incl r0                  # according to the left margin length
      sobgtr r2, margin
      pushal inline
      addl2 $outline, r0
      pushl r0
      calls $2, strcpy
      pushal outline
      calls $1, upcase
      pushal outline
      calls $1, .puts
      jmp loop
exit: pushl $0
      calls $1, .exit

      .data
inline:     .space MAXLINE
outline:    .space MAXLINE

      .text
upcase:     .word 0x00
      movl 4(ap),r1
upcase1:    movb (r1)+,r0           #read a char into r0
      beql upcase2                  # 0 means end of string
      cmpb r0,$97                   # $97='a
      blss upcase1
      cmpb r0,$122                  # $122 = 'z
      bgtr upcase1
      subl2 $32,r0
      movb r0,-(r1)
      incl r1
      jmp upcase1
upcase2:    ret

strlen:     .word 0x00
      movl 4(ap),r1
strlen1:    tstb (r1)+
      bneq strlen1
      subl2 4(ap),r1
      decl r1
```

```
      ret

strcpy:     .word 0x00
      movl 4(ap),r1
      movl 8(ap),r0
strcpy1:    movb (r0)+,(r1)+
      bneq strcpy1
      ret
```

10.4. Program 4 – Prime Numbers

The following program prints on the screen all the prime numbers from a specific range.

```
.text
.set maxprimes, 100
.set primesize, 4*maxprimes
main: .word   0
      movl    $2,r6 # value
      clrl    r5
test: movl    $2,r7 # divisor
loop: cmpl    r6,r7 # last divisor?
      bneq    divide
      pushl   r6
      pushal  format
      calls   $2,.printf
      incl    r5
      movl    r6,prime[r5]
      cmpl    r5,$maxprimes
      bneq    next
      pushl   $0
      calls   $1,.exit
divide:
      ediv    r7,r6,r8,r9
      tstl    r9    # remainder
      beql    next  # not prime
      incl    r7
      jmp     loop  # next divisor
next: incl    r6   # next value
      jmp     test

format:     .asciz  "%10d"
prime:            .space primesize
```

10.5. Program 5 – String Processing

```
# Gets lines of text from the user. When input is done,
# We display how many times each of the English letters has appeared.

.text
main: .word 0


# Read line
readLine:
      pushal theString
      calls $1, .gets
      tstl r0
      blss end

#Analyze each one of the chars in the line
analyzeChar:
      cmpb (r0), $'a
      blss nextChar
      cmpb (r0), $'z
      bgtr nextChar
      movb (r0), r1
      subl2 $97, r1
      incb charsCount(r1)

nextChar:
      incl r0
      tstl (r0)
      jneq analyzeChar

      brb readLine
end:
      movl $0, r3
      clrl r2

# Print the results
printLoop:
      moval charsCount, r0
      movb (r0)[r3], r2
      pushl r2
      pushl $97
      addl2 r3, (sp)
      pushal format
      calls $3, .printf
      incl r3
      cmpl r3, $26
      jneq printLoop

      pushl $0
      calls $1, .exit

.data
theString:  .space 80
charsCount: .space 26
format:     .asciz "%c=%d  "
```

10.6. Program 6 - Fibonacci Series - Version 1

We provide here two implementations of functions that calculate Fibonacci numbers. The following program displays Fibonacci numbers iteratively. It displays the first LINES Fibonacci numbers.

```
#fibonachi series

.text
.set LINES, 10

# start of the program
main: .word 0
      pushl $LINES      # number of lines that will be printed must
be grater then 0
      calls $1, fib     # call to fib function that print the correct
numbers
      pushl $0
      calls $1, .exit   # exit from the program

# this function compute the fibonachi series and prints it on screen
# needs an argument to know how much numbers to print and the
function
# takes it from the stack, it doesn't change any of the registers

fib:  .word 0x0f        # save register r0 to r4
      movl 4(ap), r4    # loop register
      movl $0, r2       # previously sum
      movl $1, r3       # current sum
      pushl $1          # print the first number
      pushal format
      calls $2, .printf # print the first number of the series
      decl r4
loop: addl3 r2,r3,r1    # temporary register
      movl r3, r2       # save prev sum
      movl r1, r3       # save current sum
      pushl r1          # the number that we need to print
      pushal format     # send the print format as parameter
      calls $2, .printf # print the current number of the series
      sobgtr r4, loop   # need to print more?
      ret               # back to main

format: .asciz "%d\n"
```

10.7. Program 7 - Fibonacci Series - Version 2

The following implementation is recursive function that returns the Fibonacci series N-th member. This example uses other opcodes and addressing modes than the previous example.

```
# fib.asm - display the Fibonacci series N-th member, recursive
implementation

      .text
      .word 0
      .set N, 10

      callg arglist, fib
      pushl r0
      pushal format
      calls $2, .printf
      pushl $0
      calls $1, .exit


# receives the number N, and returns the fibonnaci series N-th member
in r0

fib:  .word 0x02
      movl 4(ap), r1
      cmpl r1, $2
      bgtr body
      movl $1, r0
      ret
body: subl2 $4, sp
      movab -(r1), -(sp)
      calls $1, fib
      movl r0, -4(fp)
      movab -(r1), -(sp)
      calls $1, fib
      addl2 -4(fp), r0
      ret

.data
arglist:    .long 1, N
format:     .asciz "%d\n"
```

10.8. Program 8 - Copy a String

The following program copies a string from one location to another.

```
# The following program takes a string and copy it to free location
# in the memory

.text

main: .word 0
      moval str, r0     # source string
      moval dup, r1     # duplicate string
loop1:
      movb (r0)+, (r1)+
      bneq loop1

      pushal dup
      calls $1, .puts
end:
      pushl $0
      calls $1,.exit
      .data
str:  .asciz "any string"
dup:  .space  50
```

10.9. Program 9 - Analyze input - number of chars, words and lines

```
            .text
            .set    LF,10
            .set    SP,0x20
main:       .word   0
loop:       calls   $0,.getchar
            movb    r0,ch
            blss    eof
            incl    chars
            cmpb    ch,$SP
            bleq    space
nonspace:   locc    ch,delimits,delimit1
            beql    space
            incw    nonspaces
            incw    linechars
            cmpw    nonspaces,$1
            beql    newword
            jmp     loop
delimits:   .byte   3 #length
delimit1:   .ascii  ".,:"
newword:    incw    words
            jmp     loop
space:      clrw    nonspaces
            incw    linechars
            cmpb    ch,$LF
            beql    newline
            jmp     loop
newline:    incl    lines
            clrw    linechars
            jmp     loop
eof:        tstw    linechars
            beql    output
            incw    lines
output:     pushl   lines
            pushl   words
            pushl   chars
            pushal  format
            calls   $4,.printf
            pushl   $0
            calls   $1,.exit
           .data
ch:        .word 0
nonspaces: .word 0
linechars: .word 0
format:    .asciz "Chars: %ld\tWords: %ld\tLines: %ld\n"
chars:     .long 0
words:     .long 0
lines:     .long 0
```

11. VAX-11 Opcodes

11.1. OPERAND DESCRIPTORS

An operand descriptor is a string of characters consisting of three components:

<name>.<access type><datatype context>

The three components and their possible values are as follows.

(a) name

The name component of a descriptor is any word or abbreviation which is descriptive of the operand involved. Names such as src (“source”), dst (“destination”), pos (‘position”), and so on, are used to give some indication as to the significance of the operand.

(b) access type

The access type component of a descriptor can take on a number of possible values.

r read-only. The operand is a structure (byte, word, and so on, depending on the operations context) whose location is specified by any general register or program counter addressing mode. The operand is read by the processor, but is not written by it.

w written-only. The operand is a structure (byte, word, and so on, depending on the operation’s context) whose location is specified by any addressing mode except 0, 1, 2, or 3 (short literal) or program counter mode 8 (immediate). The operand is written by the processor, but is not read by it.

m modified. The operand is a structure (byte, word, and so on, depending on the operation’s context) whose location is specified by any addressing mode except 0, 1,2, or 3 (short literal) or program counter mode 8 (immediate). The operand is read by the processor, and it may be modified and its new value written back to its location.

b branch displacement. There is no structure reference. Rather, the operand is a program counter displacement, whose size is determined by the operation’s context, and whose value is sign extended to a longword upon execution.

a address access. The operand is the address of a structure (byte, word, arid so on, depending on the operation’s context). The operand may be specified by any addressing mode except 0, 1, 2, 3 (short literal); general register mode 5 (register); and program counter mode 8 (immediate). Regardless of the operation’s context, the operand is always a longword (since it is an address). The context of the address calculation is determined by the operation’s context.

v bit field. The operand is one of the following.

1. The address of a structure (byte, word, and so on, depending on the operation's ontext). The operand may be specified by any addressing mode except 0, 1, 2, 3 (short literal); general register mode 5 (register); and program counter mode 8 (immediate). Regardless of the operations context, the operand is always a longword (since it is an address). The context of the address calculation is determined by the operation's context.
2. The contents of a register, the operand being specified by a general register mode 5 construction—Rn. The operand is the contents of Rn, or of R[n + l]'R[n]

(c) datatype context

The datatype context component of the descriptor specifies the operation's context and is used to determine side effects, to calculate addresses, and to determine the factor by which register contents are multiplied in index mode instruction. The possible datatype descriptors are the following.

b	byte
w	word
l	longword
q	quadword
o	octaword
x	datatype of the first (or only) operand specified by the operation
y	datatype of the second operand specified by the operation

11.2. NOTATION

<—	is given the value
{. . .}	= 1, if... is TRUE; = 0, if ... is FALSE
—(SP) <—	is pushed onto the stack
<— (SP)+	is popped off the stack
+	addition
—	subtraction, or unary minus
*	multiplication
/	division
**	exponentiation
c(. . .)	contents of
SEXT(. . .)	the value of . . . , sign extended to a longer structure
ZEXT(. . .)	the value of . . . , zero extended to a longer structure
m: n	the bit field of a structure, consisting of bits m, m - 1, m - 2, . . ., n + 1, n
x<n>	bit n of the structure x
x<m : n>	bits m:n of the structure x
a[b]	the address a indexed by the value b
MAX(x,y)	the maximum of the numbers x and y
MI N (x ,y)	the minimum of the numbers x and y
EQL	equal to, signed or unsigned
GEQ	greater than or equal to, signed
GEQU	greater than or equal to, unsigned
GTR	greater than, signed
GTRU	greater than, unsigned
LEQ	less than or equal to, signed
LEQU	less than or equal to, unsigned
LSS	less than, signed
LSSU	less than, unsigned
NEQL	not equal to, signed or unsigned

11.3. VAX-11 Opcodes and Examples

The following pages contain all the VAX-11 opcodes we implemented, including our test program for each one of them.
The examples are our main test program for the simulator.

11.3.1. ACB ADD COMPARE AND BRANCH

Purpose	maintain loop count and loop
Format	opcode limit.rx, add.rx, index.mx, displ.bw
Operation	index ← index + add; if {{add GEQ 0} AND {index LEQ limit}} OR {{add LSS 0} AND {index GEQ limit}} then PC ← PC + SEXT (displ);
Condition codes	N ← index LSS 0; Z ← index EQL 0; V ← {integer or floating overflow}; C ← C;
Exceptions	integer overflow floating overflow floating underflow reserved operand
Opcodes	9D ACBB Add Compare and Branch Byte 3D ACBW Add Compare and Branch Word Fl ACBL Add Compare and Branch Long 4F ACBF Add Compare and Branch Floating 6F ACBD Add Compare and Branch Double
Description	The addend operand is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. If the addend operand is positive (or 0) and the comparison is less than or equal or if the addend is negative and the comparison is greater than or equal, the sign-extended branch displacement is added to PC and PC

	is replaced by the result.
Notes	1. ACB efficiently implements the general FOR or DO loops in high-level languages since the sense of the comparison between index and limit is dependent on the sign of the addend. 2. On integer overflow, the index operand is replaced by the low order bits of the true result. Comparison and branch determination proceed normally on the updated index operand. 3. On floating underflow, the index operand is replaced by 0. Comparison and branch determination proceed normally. 4. On floating overflow, the index operand is replaced by an operand of all bits 0 except for a sign bit of 1 (reserved operand). N ← 1; Z ← 0 V ← 1. The branch is not taken. 5. On a reserved operand fault, the index operand is unaffected and the condition codes are unpredictable 6. Except for 5, above, the C-bit is unaffected 7. On a trap, the branch condition will be tested and the PC potentially updated before the exception is taken Thus the PC might point to the start of the loop and not the next consecutive instruction.

Example 1

The program prints the numbers 0 to 10 on the screen.

```
.text

main: .word 0
      movl $10, r1
      movl $0, r2

forLoop:
      pushl r2
      pushal format
      calls $2, .printf
      acbl r1, $1, r2, forLoop

      pushl $0
      calls $1, .exit
.data
format: .asciz "%d "
```

11.3.2. ADD ADD

Purpose	perform arithmetic addition		
Format	opcode add.rx, sum.mx		2 operand
	opcode addl.rx, add2.rx, sum.wx		3 operand
Operation	sum ←sum + add;		2 operand
	sum ← add1 + add2;		3 operand
Condition codes	N ← sum LSS 0; Z ← sum EQL 0; V←overflow; C ← carry from most significant bit (integer); C ← 0 (floating);		
Exceptions	Integer overflow Floating overflow Floating underflow Reserved operand		
Opcodes	80	ADDB2	Add Byte 2 Operand
	81	ADDB3	Add Byte 3 Operand
	A0	ADDW2	Add Word 2 Operand
	Al	ADDW3	Add Word 3 Operand
	C0	ADDL2	Add Long 2 Operand
	C1	ADDL3	Add Long 3 Operand
	40	ADDF2	Add Floating 2 Operand
	41	ADDF3	Add Floating 3 Operand
	60	ADDD2	Add Double 2 Operand
	61	ADDD3	Add Double 3 Operand
Description	In 2 operand format, the addend operand is added to the sum operand and the sum operand is replaced by the result. In 3 operand format, the addend 1 operand is added to the addend 2 operand and the sum operand is replaced by the result. In floating point format, the result is rounded.		
Notes	1. Integer overflow occurs if the input operands to the add have the same sign and the result has the opposite sign. On overflow, the sum operand is replaced by the low order bits of the true result. 2. On a floating reserved operand fault, the sum operand		

	is unaffected and the condition codes are unpredictable. 3. On floating underflow, the sum operand is replaced by 0. 4. On floating overflow, the sum operand is replaced by an operand of all bits 0 except for a sign bit of 1 (a reserved operand). N ←1; Z ← 0; V ← 1; and C ← 0.

Example 1

The following program puts the value 3 in R1 and 4 in R2, and then sums it and prints the result. At the end of this program, R2 is 7.

```
.text
main: .word 0
      movl $3, r1
      movl $4, r2
      addl2 r1, r2
      pushl r2
      pushal format
      calls $2, .printf

      pushl $0
      calls $1, .exit
.data
format: .asciz "R2 is %d"
```

Example 2

The following program puts the value 3 in R1 and 4 in R2, and then sums it and puts the value in R3. We then print the content of R3.

```
.text
main: .word 0
      movl $3, r1
      movl $4, r2
      addl3 r1, r2, r3
      pushl r3
      pushal format
      calls $2, .printf

      pushl $0
      calls $1, .exit
.data
format: .asciz "R3 is %d"
```

Example 3

This example comes to present the changes of the flags during calls to add opcode.

```
.text

main: .word 0
movb $0x7C, r0
addb2 $1, r0        # N = 0, V = 0
addb2 $1, r0
addb2 $1, r0
addb2 $1, r0        # N = 1, V = 1
addl2 $1, r0        # N = 0, V = 0
addb2 $1, r0        # N = 1, V = 0
addb2 $1, r0
addb2 $1, r0
movb $0xFD, r0
addb2 $1, r0        # N = 1
addb2 $1, r0
addb2 $1, r0        # Z = 1, C = 1
addb2 $1, r0
pushl $0
calls $1, .exit
```

Example 4

Another flags example:

```
.text

main: .word 0
movb $0x7C, r0
addb3 $-1, $1, r0        # Z = 1, C = 1

pushl $0
calls $1, .exit
```

11.3.3. ADWC ADD WITH CARRY

Purpose	perform extended-precision addition
Format	opcode add.rl, sum.ml
Operation	sum ← sum + add + C;
Condition codes	N ← sum LSS 0; Z ← sum EQL 0; V ← {integer overlow}; C ← {carry from most significant bit};
Exceptions	Integer overflow
Opcodes	D8 ADWC Add with Carry
Description	The contents of the condition code C bit and the addend operand are added to the sum operand and the sum operand is replaced by the result.
Notes	1. On overflow, the sum operand is replaced by the low order bits of the results. 2. The two additions in the operation are performed simultaneously.

Example 1

This example shows how ADWC changes the flags of VAX11.

```
.text
main: .word 0
      movb $0x7C, r0
      addb3 $-1, $1, r0        # Z = 1, C = 1
      adwc $0x7FFFFFFF, r1     # V = 1, N = 1

      pushl $0
      calls $1, .exit
```

11.3.4. ADAWI ADD ALIGNED WORD INTERLOCKED

Purpose	maintain operating system resource usage counts
Format	opcode add.rw, sum.mw
Operation	tmp ← add; {set interlock}; sum ← sum + tmp; {released interlock};
Condition codes	N ← sum LSS 0; Z ← sum EQL 0; V ← {integer overflow}; C ← {carry from most significant bit};
Exceptions	reserved operand fault integer overflow
Opcodes	58 ADAWI Add Aligned Word Interlocked
Description	The addend operand is added to the sum operand and the sum operand is replaced by the result. The operation is interlocked against ADAWI operations by other processors or devices in the system. The destination must be aligned on a word boundary i.e., bit zero of the sum operand address must be zero. If it is not, a reserved operand fault is taken.
Notes	1. Integer overflow occurs if the input operands to the add have the same sign and the result has the opposite sign. On overflow, the sum operand is replaced by the low order bits of the true result. 2. If the addend and the sum operand overlap, the result and the condition codes are UNPREDICTABLE.

Example 1

The example shows how we sum two registers using this opcode.

```
.text
main: .word 0
      movw        $4, r0
      movw        $5, r1
      adawi       r0, r1
      pushl       r1
      pushal      format
      calls       $2, .printf

      pushl       $0
      calls       $1, .exit

.data
format: .asciz "R1 is %d\n"
```

11.3.5. AOB ADD ONE AND BRANCH

Purpose	increment integer loop count and loop
Format	opcode limit.rl, index.ml, displ.bb
Operation	index ← index + 1; if index LSS limit AOBLSS then PC ← PC + SEXT (displ); if index LEQ limit AOBLEQ then PC ← PC + SEXT (displ);
Condition codes	N ← index LSS 0; Z ← index equal 0; C ← {integer overflow}; C ← C;
Exceptions	integer overflow
Opcodes	F2 AOBLSS Add One and Branch Less Than F3 AOBLEQ Add One and Branch Less Than or Equal
Description	One is added to the index operand and the index operand is replaced by the result. The index operand is compared with the limit operand. On AOBLSS, if it is less than, the branch is taken. On AOBLEQ, if it is less than or equal, the branch is taken. If the branch is taken, the sign extended branch displacement is added to the PC and the PC is replaced by the result.
Notes	1. Integer overflow occurs if the index operand before addition is the largest positive integer. On overflow, the index operand is replaced by the largest negative integer, and thus (unless the limit operand is the largest negative integer on AOBLSS) the branch is taken. 2. The C-bit is unaffected.

Example 1

The following prints the number 1 to 9 on the screen:

```
.text

main: .word 0
      movl $1, r1

forLoop:
      pushl r1
      pushal format
      calls $2, .printf
      aoblss $10, r1, forLoop

      pushl $0
      calls $1, .exit
.data
format: .asciz "%d "
```

11.3.6. ASH ARITHMETIC SHIFT

Purpose	shift of integer
Format	opcode cnt.rb, src.rx, dst.wx
Operation	dst ← src shifted cnt bits;
Condition codes	N ← dst LSS 0; Z ← dst EQL 0; V ← {integer overflow}; C ← 0;
Exceptions	Integer overflow
Opcodes	78 ASHL Arithmetic Shift Long 79 ASHQ Arithmetic Shift Quad
Description	The source operand is arithmetically shifted by the number of bits specified by the count operand and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand shifts to the left bringing Os into the least significant bit. A negative count operand shifts to the right bringing in copies of the most significant (sign) bit into the most significant bit position. A zero count operand replaces the destination operand with the unshifted source operand.
Notes	1. Integer overflow occurs on a left shift if any bit shifted into the sign bit position differs from the sign bit of the source operand. On overflow, the destination operand is replaced by the low order bits of the true result. 2. If cnt GEQ 32 (ASHL) or cnt GEQ 64 (ASHQ); the destination operand is replaced by 0 3. If cnt LEQ -32 (ASHL) or cnt LEQ -63 (ASHQ); all the bits of the destination operand are copies of the sign bit of the source operand. 4. A left shift is equivalent to a multiply by the corresponding power of two. A right shift is not, however, equivalent to a divide because negative numbers are rounded away from zero.

Example 1

The program demonstrates several usages of ASHL opcode.

```
.text
main: .word 0

      movb $1, r1
      ashl $4, r1, r2
      pushl r2
      pushal format
      calls $2, .printf       # Prints 16


      movb $0x10, r1
      ashl $-4, r1, r2
      pushl r2
      pushal format
      calls $2, .printf       # Prints 1

      movl $0xFFFFFFFF, r1
      ashl $4, r1, r2
      pushl r2
      pushal format2
      calls $2, .printf       # Prints FFFFFFF0


      movl $0xFFF00FFF, r1
      ashl $-4, r1, r2
      pushl r2
      pushal format2
      calls $2, .printf       # Prints FFFF00FF

      movl $0xF0FFFFFF, r1
      ashl $4, r1, r2
      pushl r2
      pushal format2
      calls $2, .printf       # Prints FFFFFF0

      pushl $0
      calls $1, .exit

.data
format: .asciz "R2 is %d\n"
format2: .asciz "R2 is %8X\n"
```

11.3.7. B BRANCH ON (CONDTION)

Purpose	test condition code
Format	opcode displ.bb
Operation	if condition then PC ← PC + SEXT (displ);
Condition codes	N ← N; Z ← Z; V ← V; C ← C;
Exceptions	none
Opcodes	CONDITION 12 Z EQL 0 BNEQ,Branch on Not Equal (signed) BNEQU, Branch on Not Equal Unsigned 13 Z EQL 1 BEQL, Branch on Equal (signed) BEOLU, Branch on Equal Unsigned 14 {N OR Z}EQL 0 BGTR, Branch on Greater Than (signed) 15 {N OR Z}EQL 1 BLEQ, Branch on Less Than or Equal (signed) 18 N EQL 0 BGEQ, Branch on Greater Than or Equal (signed) 19 N EQL 1 BLSS, Branch on Less Than (signed) 1A {C OR Z}EQL 0 BGTRU, Branch on Greater Than Unsigned 1B {C OR Z}EQL 1 BLEQU, Branch Less Than or Equal Unsigned 1C V EQL 0 BVC, Branch on Overflow Clear 1D V EQL 1 BVS, Branch on Overflow Set 1E C EQL 0 BGEQU, Branch on Greater Than or Equal unsigned BCC Branch on Carry Clear 1F C EQL 1 BLSSU, Branch on Less Than Unsigned BSC Branch on Carry Set
Description	The condition codes are tested and if the condition indicated by the instruction is met, the sign-extended branch

	displacement is added to the PC and PC is replaced by the result.
Notes	The VAX- conditional branch instructions permit considerable flexibility in branching but require care in choosing the correct branch instruction. The conditional branch instructions are divided into 3 overlapping groups: 1. Overflow and Carry Group BVS V EQL 1 BVC V EQL 0 BCS C EQL 1 BCC C EQL 0 These instructions are typically used to check for overflow (when overflow traps are not enabled), for multiprecision arithmetic, and for other special purposes. 2. Unsigned Group BLSSU C EQL 1 BLEQU {C or Z} EQL 1 BEQLU Z EQL 1 BNEQU Z EQL 0 BGEQU C EOL 0 BGTRU {C OR Z} EQL 0 These instructions typically follow integer and field in structions where the operands are treated as unsigned integers, addressed instructions, and character string in structions. 3. Signed Group BLSS N EQL 1 BLEQ {N OR Z} EQL 1 BEQL Z EQL 1 BNEQ Z EQL 0 BGEQ N EQL 0 BGTR {N OR Z} EQL 0 These instructions typically follow integer and field instructions where the operands are being treated as signed integers, floating point instructions, and decimal string instructions.

Example 1 - BEQL

The following program shows the usage of BEQL opcode.

```
.text

main: .word 0

      # first case - Z should be 1 (Equal)
      movb $0, r1
      beql eq1
      calls $0, prn_not_eq
      jmp next_stage
eq1:  calls $0, prn_eq

      # second case - Z should be 0 (Not Equal)
next_stage:
      movb $1, r1
      beql eq2
      calls $0, prn_not_eq
      jmp end_prog
eq2:  calls $0, prn_eq

end_prog:
      pushl $0
      calls $1, .exit

prn_not_eq: .word 0
      pushal not_eq
      pushal format
      calls $2, .printf
      ret
prn_eq: .word 0
      pushal eq
      pushal format
      calls $2, .printf
      ret

.data

eq:         .asciz "Equal"
not_eq:     .asciz "Not Equal"
format:     .asciz "%s\n"
```

Example 2 – BNEQ, BNEQU

Almost the same as the previous example, this program shows the usage of BNEQ, BNEQU opcodes.

```
.text

main: .word 0

      # first case - Z should be 0 (Not Equal)
      movb $0, r1
      bneq eq1
      calls $0, prn_not_eq
      jmp next_stage
eq1:  calls $0, prn_eq

      # second case - Z should be 1 (Equal)
next_stage:
      movb $1, r1
      bnequ eq2
      calls $0, prn_not_eq
      jmp end_prog
eq2:  calls $0, prn_eq

end_prog:
      pushl $0
      calls $1, .exit

prn_not_eq: .word 0
      pushal not_eq
      pushal format
      calls $2, .printf
      ret
prn_eq: .word 0
      pushal eq
      pushal format
      calls $2, .printf
      ret

.data

eq:         .asciz "Equal"
not_eq:     .asciz "Not Equal"
format:     .asciz "%s\n"
```

Example 3 – BGTR

```
.text

main: .word 0

      movb $10, r1
      movb $5, r2

      # first case - should be True
      cmpb r1, r2
      bgtr eq1
      calls $0, prn_false
      jmp next_stage
eq1:  calls $0, prn_true

      # second case - should be False
next_stage:
      cmpb r2, r1
      bgtr eq2
      calls $0, prn_false
      jmp end_prog
eq2:  calls $0, prn_true

end_prog:
      pushl $0
      calls $1, .exit

prn_false: .word 0
      pushal lbl_false
      pushal format
      calls $2, .printf
      ret
prn_true: .word 0
      pushal lbl_true
      pushal format
      calls $2, .printf
      ret

.data

lbl_true:   .asciz "True"
lbl_false:  .asciz "False"
format:     .asciz "%s\n"
```

Example 4 – BGEQ

```
.text

main: .word 0

      movb $10, r1
      movb $5, r2

      # first case - should be True
      cmpb r1, r2
      bgeq eq1
      calls $0, prn_false
      jmp next_stage
eq1:  calls $0, prn_true

      # second case - should be False
next_stage:
      cmpb r2, r1
      bgeq eq2
      calls $0, prn_false
      jmp end_prog
eq2:  calls $0, prn_true

end_prog:
      pushl $0
      calls $1, .exit

prn_false: .word 0
      pushal lbl_false
      pushal format
      calls $2, .printf
      ret
prn_true: .word 0
      pushal lbl_true
      pushal format
      calls $2, .printf
      ret

.data

lbl_true:   .asciz "True"
lbl_false:  .asciz "False"
format:     .asciz "%s\n"
```

11.3.8. BB BRANCH ON BIT

Purpose	test selected bit
Format	opcode pos.rl, base.vb, displ.bb

Operation	teststate = if {BBS} then 1 else 0;
	if FIELD (pos, 1, base) EQL teststate then
	PC ← PC + SEXT (displ);

Condition codes	N ← N; Z ← Z; V ← V; C ← C;
Exceptions	reserved operand
Opcodes	E0 BBS Branch on Bit Set El BBC Branch on Bit Clear
Description	The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to PC and PC is replaced by the result.
Notes	1. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in a register. 2. On a reserved operand fault, the condition codes are unpredictable. 3. The modification of the bit is not an interlocked operation. See BBSSII and BBCCI for interlocking instructions.

Example 1

```
.text
main: .word 0
      movb $0x80, r1
      bbs $7, r1, prn_true
      calls $0, prn_false
      jmp end_prog
eq2:  calls $0, prn_false

end_prog:
      pushl $0
      calls $1, .exit

prn_false: .word 0
      pushal lbl_false
      pushal format
      calls $2, .printf
      ret
prn_true:
      pushal lbl_true
      pushal format
      calls $2, .printf
      jmp end_prog

.data

lbl_true:   .asciz "True"
lbl_false:  .asciz "False"
format:     .asciz "%s\n"
```

11.3.9. BB BRANCH ON BIT (AND MODIFY WITHOUT INTERLOCKED)

Purpose	test and modify selected bit
Format	opcode pos.rl, base.vb, displ.bb

Operation	teststate = if {BBSS or BBSC} then 1 else 0;
	newstate = if {BBSS or BBCS} then 1 else 0;
	temp ← FIELD (pos, 1, base);
	FIELD (pos, 1, base) ← newstate;
	if tmp EQL teststate then
	PC ← PC + SEXT (displ);

Condition codes	N ← N: Z ← Z; V ← V; C ← C;
Exceptions	reserved operand
Opcodes	E2 BBSS Branch on Bit Set and Set E3 BBCS Branch on Bit Clear and Set E4 BBSC Branch on Bit Set and Clear E5 BBCC Branch on Bit Clear and Clear
Description	The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to PC and PC is replaced by the result. Regardless of whether the branch is taken or not, the tested bit is put in the new state as indicated by the instruction.
Notes	1. A reserved operand fault occurs if 0 BTRU 31 and the bit is contained in a register. 2. On a reserved operand fault, the field is unaffected and the condition codes are unpredictable. 3. The modification of the bit is not an interlocked operation. See BBSSI and BBCCI for interlocking instructions.

11.3.10. BB BRANCH ON BIT INTERLOCKED

Purpose	test and modify selected bit under memory interlock
Format	opcode pos.rl, base.vb, displ.bb

Operation	teststate = if {BBSSI} the 1 else 0;
	newstafe = teststate;
	{set interlock};
	temp ← FIELD (pos, 1, base);
	FIELD (pos, 1, base) ← newstate;
	{release interlock};
	if tmp EQL teststate then
	PC ← PC + SEXT (displ);

Condition codes	N ← N; Z ← Z; V ← V; C ← C;
Exceptions	reserved operand
Opcodes	E6 BBSSI Branch on Bit Set and Set Interlocked E7 BBCCI Branch on Bit Clear and Clear Interlocked
Description	The single bit field specified by the position and base operands is tested. If it is in the test state indicated by the instruction, the sign-extended branch displacement is added to the PC and PC is replaced by the result. Regardless of whether the branch is effected or not, the tested bit is put in the new state as indicated by the instruction. If the bit is contained in memory, the reading of the state of the bit and the setting of it to the new state is an interlocked operation. The operation is interlocked against similar operations by other processors or devices in the system.
Notes	1. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in registers. 2. On a reserved operand fault, the field is unaffected and the condition codes are unpredictable. 3. Except for memory interlocking BBSSI is equivalent to BBSS and BBCCI is equivalent to BBCC.

11.3.11. BIC BIT CLEAR

Purpose	perform complemented AND of two integers		
Format	opcode mask.rx, dst.mx		2 operand
	opcode mask.rx, src.rx, dst.wx		3 operand
Operation	dst ←dst AND {NOT mask};		2 operand
	dst ← src AND {NOT mask};		3 operand
Condition codes	N ← dst LSS 0; Z ← dst EQL 0; V ← 0; C ← C;		
Exceptions	None		
Opcodes	8A	BICB2	Bit Clear Byte; 2 operand
	8B	BICB3	Bit Clear Byte; 3 operand
	AA	BICW2	Bit Clear Word; 2 operand
	AB	BICW3	Bit Clear Word; 3 operand
	CA	BICL2	Bit Clear Long; 2 operand
	GB	BICL3	Bit Clear Long; 3 operand
Description	In 2 operand format, the destination operand is ANDed with the ones complement of the mask operand and the destination operand is replaced by the result. In 3 operand format, the source operand is ANDed with the l's ones complement of the mask operand and the destination operand is replaced by the result.		
Notes			

Example 1

The following program reset the first byte of r5 using bicl2.

```
.text

main: .word 0
movl $0xFFFFFFFF, r5
bicl2 $0xFF, r5
pushl r5
pushal format
calls $2, .printf

pushl $0
calls $1, .exit

.data
format: .asciz "%X\n"
```

Example 2

The following program resets the first byte of r5 using bicl3 and stores the result on r6.

```
.text

main: .word 0
movl $0xFFFFFFFF, r5
bicl3 $0xFF, r5, r6
pushl r6
pushal format
calls $2, .printf

pushl $0
calls $1, .exit

.data
format: .asciz "%X\n"
```

Example 3

The following example demonstrates the effects of BIS and BIC on VAX11 flags.

```
.text

main: .word 0

      movb  $0, r4
      bisb2 $0xf  , r4        # All flags are zero, r4 = 0xF
      bisb2 $0xf0 , r4        # N = 1, r4 = 0xFF
      bisl2 $0xf0 , r4        # All flags are zero, r4 = 0xFF
      bicb2 $0xff , r4        # Z = 1, r4 = 0x00
      bicb2 $0xff , r4        # Z = 1, r4 = 0x00

      movb $0xFF  , r1
      incb r1
      bisl2 $0x80000000 , r4  # N = 1, C = 1

      bisw2 $0x0F0F     , r4  # N = 0, C = 1, r4 = 0x80000F0F

      bicl2 $0xFFFFFFFF, r4   # Z = 1, C = 1

      halt
```

11.3.12. BIS BIT SET

Purpose	perform logical inclusive OR of two integers		
Format	opcode mask.rx, dst.mx		2 operand
	opcode mask.rx, src.rx, dst.wx		3 operand
Operation	dst . ← dst OR mask;		2 operand
	dst ← src OR mask;		3 operand
Condition codes	N ← dst LSS 0; Z ← dst EQL 0; V ← 0; C ← C;		
Exceptions	None		
Opcodes	88	BISB2	Bit Set Byte 2 Operand
	89	BISB3	Bit Set Byte 3 Operand
	A8	BISW2	Bit Set Word 2 Operand
	A9	BISW3	Bit Set Word 3 Operand
	C8	BISL2	Bit Set Long 2 Operand
	C9	BISL3	Bit Set Long 3 Operand
Description	In 2 operand format, the mask operand is ORed with the destination operand and the destination operand is replaced by the result. In 3 operand format, the mask operand is ORed with the source operand and the destination operand is replaced by the result.		
Notes			

Example 1

Example of using the bisl2 opcode:

```
.text

main: .word 0
movl  $0xF0F0F0F0, r5
bisl2 $0x0A0B0C0D, r5
pushl r5
pushal format
calls $2, .printf

pushl $0
calls $1, .exit

.data
format: .asciz "%X\n"
```

The output of the program is FAFBFCFD.

Example 2

Example of using the bisl3 opcode. The output is the same as the previous example.

```
.text

main: .word 0
movl  $0xF0F0F0F0, r5
bisl3 $0x0A0B0C0D, r5, r6
pushl r6
pushal format
calls $2, .printf

pushl $0
calls $1, .exit

.data
format: .asciz "%X\n"
```

Example 3

The following example demonstrates the effects of BIS and BIC on VAX11 flags.

```
.text

main: .word 0

      movb  $0, r4
      bisb2 $0xf  , r4        # All flags are zero, r4 = 0xF
      bisb2 $0xf0 , r4        # N = 1, r4 = 0xFF
      bisl2 $0xf0 , r4        # All flags are zero, r4 = 0xFF
      bicb2 $0xff , r4        # Z = 1, r4 = 0x00
      bicb2 $0xff , r4        # Z = 1, r4 = 0x00

      movb $0xFF  , r1
      incb r1
      bisl2 $0x80000000 , r4  # N = 1, C = 1

      bisw2 $0x0F0F     , r4  # N = 0, C = 1, r4 = 0x80000F0F

      bicl2 $0xFFFFFFFF, r4   # Z = 1, C = 1

      halt
```

11.3.13. BISPSW, BICPSW BIT SET PSW, BIT CLEAR PSW PSL

Purpose	set or clear trap enables	
Format	opcode mask.rw	
Operation	PSW ← PSW OR mask;	BISPSW
	PSW ← PSW AND {NOT mask};	BICPSW
Condition codes	N ← N OR mask <3>;	BISPSW
	Z ← Z OR mask <2>;	
	V ← V OR mask <1>;	
	C ← C OR mask <0>;	
	N ← N AND {NOT mask} <3>;	BICPSW
	Z ← Z AND {NOT mask} <2>;	
	V ← VAND {NOT mask} <1>;	
	C ← C AND {NOT mask} <0>;	
Exceptions	Reserved Operand	
Opcodes	B8 BISPSW Bit set PSW	
	B9 BICPSW Bit clear PSW	
Description	On BISPSW, the processor status longword is ORed with the 16-bit mask operand and the PSW is replaced by the result. On BICPSW, the processor status longword is ANDed with the l's complement of the 16-bit mask operand and the PSW is replaced by the result.	
Notes	A reserved operand fault occurs if mask <15:8> is not zero. On a reserved operand fault, the PSW is not affected.	

Example 1

```
.text

main: .word 0
      bispsw $0x55
      bicpsw $5

      halt
```

11.3.14. BIT BIT TEST

Purpose	test a set of bits for all zero
Format	opcode mask.rx, src.rx
Operation	tmp ← src AND mask;
Condition codes	N ← tmp LSS 0; Z ← tmp EQL 0; V ← 0; C ← C;
Exceptions	None
Opcodes	93 BITB Bit Test Byte B3 BITW Bit Test Word D3 BITL Bit Test Long
Description	The mask operand is ANDed with the source operand. Both operands are unaffected. The only action is to affect condition codes.
Notes	

Example 1

The program displays message on the screen, which effected from the result of BITL command.

```
.text

main: .word 0

      movl $0xFF, r2
      movl $0xFF, r3

      # first case - should be True
      bitl r2, r3
      bneq eq1
      calls $0, prn_false
      jmp next_stage
eq1:  calls $0, prn_true

      # second case - should be False
```

```
next_stage:
      movl $0x0F, r2
      movl $0xF0, r3

      bitl r2, r3
      bneq eq2
      calls $0, prn_false
      jmp end_prog
eq2:  calls $0, prn_true

end_prog:
      pushl $0
      calls $1, .exit

prn_false: .word 0
      pushal lbl_false
      pushal format
      calls $2, .printf
      ret
prn_true: .word 0
      pushal lbl_true
      pushal format
      calls $2, .printf
      ret

.data

lbl_true:   .asciz "True"
lbl_false:  .asciz "False"
format:     .asciz "%s\n"
```

11.3.15. BLB BRANCH ON LOW BIT

Purpose	test bit		
Format	opcode src.rl, displ.bb		

Operation	teststate = if {BLBS} then 1 else 0;
	if src<0> EQL teststate then
	PC ← PC + SEXT (displ);		

Condition codes	N ← N; Z ← Z; V ← V; C ← C;		
Exceptions	none		
Opcodes	E8 E9	BLBS BLBC	Branch on Low Bit Set Branch on Low Bit Clear
Description	The low bit (bit 0) of the source operand is tested and if it is equal to the test state indicated by the instruction, the sign-extended branch displacement is added to PC and PC os replaced by the result.		
Notes	The source operand is taken with longword context although only one bit is tested.		

Example 1

The example checks the first bit of r1 and jump to prn_true, as it contains 1.

```
.text
main: .word 0
      movb $1, r1
      blbs r1, prn_true
prn_false:
      pushal lbl_false
      pushal format
      calls $2, .printf
      halt

prn_true:
      pushal lbl_true
      pushal format
      calls $2, .printf
      halt

.data

lbl_true:   .asciz "True"
lbl_false:  .asciz "False"
format:     .asciz "%s\n"
```

11.3.16. BPT BREAKPOINT FAULT

Purpose	stop for debugging
Format	opcode
Operation	PSL<TP>←0;
Condition codes	N ← 0; Z ← 0; V ← 0; C ← 0;
Exceptions	none
Opcodes	03 BPT Breakpoint Fault
Description	This instruction is used, together with the T-bit, to implement debugging facilities.

11.3.17. BR, JMP BRANCH, JUMP

Purpose	transfer control	
Format	opcode displ.bx	Branch
	opcode dst.ab	Jump
Operation	PC ← PC +SEXT (displ);	Branch
	PC ← dst;	Jump
Condition codes	N ← N; Z ← Z; V ← V; C ← C;	
Exceptions	none	
Opcodes	11 BRB Branch With Byte Displacement 31 BRW Branch With Word Displacement 17 JMP Jump	
Description	For branch, the sign-extended branch displacement is added to PC and PC is replaced by the result. For Jump, the PC is replaced by the destination operand.	

11.3.18. BSB, JSB SUBROUTINE INSTRUCTIONS JUMP, BRANCH TO SUBROUTINE

Purpose	transfer control to subroutine
Format	opcode displ.bx branch to subroutine opcode dst.ab jump to subroutine
Operation	-(SP) ← PC; PC ← PC + SEXT (displ); branch to subroutine PC ← dst; jump to subroutine
Condition codes	N ← N; Z ← Z; V ← V; C ← C;
Exceptions	none
Opcodes	10 BSBB Branch to Subroutine with Byte Displacement 30 BSBW Branch to Subroutine with Word Displacement 16 JSB Jump to Subroutine
Description	PC is pushed on the stack as a longword. For branch, the sign-extended branch displacement is added to PC and PC is replaced by the result. For jump, PC is replaced by the destination operand.
Notes	Since the operand specifier conventions cause the evaluation of the destination operand before saving PC, JSB can be used for co routine calls with the stack used for linkage. The form of such a call is JSB *(SP)+.

11.3.19. CALLG CALL PROCEDURE WITH GENERAL ARGUMENT LIST

Purpose	invoke a procedure with actual arguments from anywhere in memory
Format	opcode arglist.ab, dst.ab
Operation	{align stack}; {create stack frame); {set arithmetic trap enables}; {set new values of AP, EP, PC};
Condition codes	N ← 0; Z ← 0; V ← 0; C ← 0;
Exceptions	reserved operand
Opcodes	FA CALLG Call Procedure with General Argument List
Description	SP is saved in a temporary and then bits 1:0 are replaced by 0 so that the stack is longword aligned. The procedure entry mask is scanned from bit 11 to 0 and the contents of registers whose number corresponds to set bits in the mask are pushed on the stack as longwords. PC, FP, and AP are pushed on the stack as longwords. The condition codes are cleared. A long-word containing the saved two low bits of SP in bits 31:30, a 0 in bit 29 and bit 28. The low 12 bits of the procedure entry mask in bits 27:16, and the PSW in bits 15:0 with T cleared is pushed on the stack. A longword 0 is pushed on the stack. FP is r placed by SP. AP is replaced by the arglist operand which specifies the address of the actual argument list. The trap enables in the PSW are set to a known state. Integer overflow, and decimal overflow are affected according to bits 14 and 15 of the entry mask respectively; floating underflow is cleared. 7- bit is unaffected. PC is replaced by the sum of destination operand plus 2 which transfers control to the called procedure at the byte beyond the entry mask.
Notes	1. If bits 13:12 of the entry mask are not 0, a reserved operand fault occurs. 2. On a reserved operand fault, condition codes are unpredictable. The procedure calling standard and the condition handling

	facility require the following register saving conventions. R0 and R1 are always available for function return values and are never saved in the entry mask. All registers R2 through R11 which are modified in the called procedure must be preserved in the mask.

11.3.20. CALLS CALL PROCEDURE WITH STACK ARGUMENT LIST

Purpose	invoke a procedure with actual arguments or addresses on the stack
Format	opcode numarg.rl, dst.ab
Operation	{push arg count}; {align stack}; {create stack frame); {set arithmetic trap enables}; {set new values of AP, EP, PC};
Condition codes	N ← 0; Z ← 0; V ← 0; C ← 0;
Exceptions	reserved operand
Opcodes	FB CALLS Call Procedure With Stack Argument List
Description	The numarg operand is pushed on the stack as a longword (byte 0 contains the number of arguments high order 24 bits are used by DIGITAL software). SP is saved in a temporary and then bits 1:0 of SP are replaced by 0 so that the stack is long- word aligned. The procedure entry mask is scanned from bit 11 to bit 0 and the contents of register whose number corresponds to set bits in the mask are pushed on the stack. PC, FP, and AP are pushed on the stack as longwords. The condition codes are cleared. A longword containing the saved two low bits of SP in bits 31:30, a 1 in bit 29, a 0 in bit 28, the low 12 bits of the procedure entry mask in bits 27:16, and the PSW in bits 15:0 with T cleared is pushed on the stack. A longword 0 is pushed on the stack. FP is replaced by SP. AP is set to the saved SP (the value of the stack pointer after the number of arguments operand was pushed on the stack). The trap enables in the PSW are set to a known state. Integer overflow and decimal overflow are affected according to bits 14 and 15 of the entry mask, respectively; floating underflow is cleared. T bit is unaffected. AP is replaced by the saved SP. PC is replaced by the sum of destination operand plus 2 which transfers control to the called procedure at the byte beyond the entry mask.

Notes	1. If bits 13:12 of the entry mask are not 0, a reserved operand fault occurs. 2. On a reserved operand fault, the condition codes are unpredictable. 3. Normal use is to push the arglist onto the stack in reverse order prior to the CALLS. On return, the arglist is removed from the stack automatically. 4. The procedure calling standard and the condition handling facility require the following register saving conventions. R0 and R1 are always available for function return values and are never saved in the entry mask. All registers R2 and R11 which are modified in the called procedure must be preserved in the entry mask.

Example 1

The following program demonstrate 3 functions calls.

```
.text
main: .word 0
      calls $0, func1
      calls $0, func1
      calls $0, func1

      pushl $0
      calls $1, .exit

func1:      .word 0

      movl $99, r1
      pushl r1
      pushal format
      calls $2, .printf
      ret
.data
format: .asciz "R1 is %d\n"
```

11.3.21. CASE CASE INSTRUCTIONS

Purpose	perform multi-way branching depending on arithmetic input
Format	opcode selector.rx, base.rx, limit.rx, displ[0].bw,...,displ[limit].bw
Operation	tmp ← selector - base; PC ← PC + if tmp LEQU limit then SEXT (displ [tmp]) else {2 + 2* ZEST (limit)};
Condition codes	N ← temp LSS limit; Z ← temp EQL limit; V ← 0; C ← temp LSSU limit;
Exceptions	none
Opcodes	8F CASEB Case Byte AF CASEW Case Word CF CASEL Case Long
Description	The base operand is subtracted from the selector operand and a temporary is replaced by the result. The temporary is compared with the limit operand and if it is less than or equal unsigned, a branch displacement selected by the temporary value is added to PC and PC is replaced by the result. Otherwise, 2 times the sum of the limit operand plus 1 is added to PC and PC is replaced by the result. This causes PC to be moved past the array of branch displacements. Regardless of the branch taken, the condition codes are affected by the comparison of the temporary operand with the limit operand.
Notes	1. After operand evaluation, PC is pointing at displ [0] not the next instruction. The branch displacements are relative to the address of displ [0]. 2. The selector and base operands can both be considered either as signed or unsigned integers. 3. The limit is {the number of choices}-1.

11.3.22. CLR CLEAR

Purpose	clear a scalar quantity
Format	opcode dst.wx
Operation	dst ← 0;
Condition codes	N ← 0: Z ← 1; V ← 0; C ← C;
Exceptions	None
Opcodes	94 CLRB Clear Byte B4 CLRW Clear Word D4 CLRL Clear Long D4 CLRF Clear Floating 7C CLRQ Clear Quad 7C CLRD Clear Double
Description	The destination operand is replaced by 0.
Notes	CLRx dst is equivalent to MOVx 0,dst, but is shorter.

Example 1:

The following program's output is "56780000 0".

```
.text
main: .word 0
movl $0x12345678, r0
movl $0x56781234, r1
clrl r0
clrw r1
pushl r0
pushl r1
pushal format
calls $3, .printf
halt
.data
format: .asciz "%X %X\n"
```

11.3.23. CMP COMPARE

Purpose	arithmetic comparison between two scalar quantities
Format	opcode srcl.rx, src2.rx
Operation	srcl – src2;
Condition codes	N ← srcl LSS src2; Z ← srcl EQL src2; V ← 0; C ← srcl LSSU src2 (integer); C ← C (floating);
Exceptions	None (integer); reserved operand (floating point)
Opcodes	91 CMPB Compare Byte Bi CMPW Compare Word Dl CMPL Compare Long 51 CMPF Compare Floating 71 CMPD Compare Double
Description	The source 1 operand is compared with the source 2 operand. The only action is to affect the condition codes.
Notes	On a floating reserved operand fault, the condition codes are unpredictable.

Example 1

The following program demonstrates condition branches that effected from CMP results.

```
.text

main: .word 0

      movb $10, r1
      movb $5, r2

      # first case - should be True
      cmpb r1, r2
      bgtr eq1
      calls $0, prn_false
      jmp next_stage
eq1:  calls $0, prn_true
```

```
      # second case - should be False
next_stage:
      cmpb r2, r1
      bgtr eq2
      calls $0, prn_false
      jmp end_prog
eq2:  calls $0, prn_true

end_prog:
      pushl $0
      calls $1, .exit

prn_false: .word 0
      pushal lbl_false
      pushal format
      calls $2, .printf
      ret
prn_true: .word 0
      pushal lbl_true
      pushal format
      calls $2, .printf
      ret

.data

lbl_true:   .asciz "True"
lbl_false:  .asciz "False"
format:     .asciz "%s\n"
```

11.3.24. CMPC COMPARE CHARACTERS

Purpose	to compare two character strings
Format	opcode len.rw, src1addr.ab, src2addr.ab 3 operand opcode src1len.rw, src1addr.ab, fill.rb, src2len.rw 5 operand src2addr.ab
Operation	Compare bytes in order from start of string. On 5 operand opcode, if one of the strings is shorter than the second one we use fill to compare to the rest of the characters of the second string.
Condition codes	Final Condition codes reflect last affecting of Condition Codes in Operation above. N ← {first byte} LSS {second byte}; Z ← {first byte} EQL {second byte}; V ← 0; C ← {first byte} LSSU {second byte};
Exceptions	None
Opcodes	29 CMPC3 Compare Characters 3 Operand 2D CMPC5 Compare Characters 5 Operand
Description	In 3 operand format, the bytes of string 1 specified by the length and address 1 operands are compared with the bytes of string 2 specified by the length and address 2 operands. Comparison proceeds until inequality is detected or all the bytes of the strings hare been examined. Condition codes are affected by the result of the last byte comparison. In 5 operand format, the bytes of the string 1 specified by the length 1 and address operands are compared with the bytes of string 2 specified by the length 2 and address 2 operands. If one string is longer than the other, the shorter string is conceptually extended to the length of the longer by appending (at higher addresses) bytes equal to the fill operand. Comparison proceeds until inequality is detected or all the bytes of the strings have been examined. Condition codes are affected by the result of the last byte comparison.
Notes	1. After execution of CMPC3;

	R0 = number of bytes remaining in string 1 (including byte which terminated comparison); R0 is zero only if strings are equal. Rl = address of the byte in string 1 which terminated comparison; if strings are equal, Al = address of one byte beyond string 1. R2 = R0 R3 = address of the byte in string 2 which terminated comparison: if strings are equal, R3 = address of one byte beyond string 2. 2. After execution of CMPC5: R0 = number of bytes remaining in string 1 (including byte which terminated comparison); R0 is zero Only if string 1 and string 2 are of equal length and equal or string 1 was exhausted before comparison terminated. Rl = address of the byte in string 1 which terminated comparison; if comparison did not terminate before string 1 exhausted, R1 = address of one byte beyond string 1. R2 = number of bytes remaining in string 2 (including byte which terminated comparison): R2 is zero only if string 2 and string 1 are of equal length or string 2 was exhausted before comparison terminated. R3 = address of the byte in string 2 which terminated comparison; if comparison did not terminate before string 2 was exhausted, R3 = address of one byte beyond string. 3. If both strings have zero length, Z is set and N and C are cleared just as in the case of two equal strings.

11.3.25. CVT CONVERT

Purpose	convert a signed quantity to a different signed data type		
Format	opcode src.rx, dst.wy		
Operation	dst ← conversion of src;		
Condition codes	N ← dst LSS 0; Z ← dst EQL 0; V ← {src cannot be represented in dst}; C ← 0;		
Exceptions	Integer overflow Floating overflow Reserved operand		
Operation codes	99	CVTBW	Convert Byte to Word
	98	CVTBL	Convert Byte to Long
	33	CVTWB	Convert Word to Byte
	32	CVTWL	Convert Word to Long
	F6	CVTLB	Convert Long to Byte
	F7	CVTLW	Convert Long to Word
	4C	CVTBF	Convert Byte to Floating
	6C	CVTBD	Convert Byte to Double.
	4D	CVTWF	Convert Word to Floating
	6D	CVTWD	Convert Word to Double
	4E	CVTLF	Convert Long to Floating
	6E	CVTLD	Convert Long to Double
	48	CVTFB	Convert Floating to Byte
	68	CVTDB	Convert Double to Byte
	49	CVTFW	Convert Floating to Word
	69	CVTDW	Convert Double to Word
	4A	CVTFL	Convert Floating to Long
	4B	CVTRFL	Convert Rounded Floating to Long
	6A	CVTDL	Convert Double to Long
	6B	CVTRDL	Convert Rounded Double to Long
	56	CVTFD	Convert Floating to Double
	76	CVTDF	Convert Double to Floating
Description	The source operand is converted to the data type of the		

	destination operand and the destination operand is replaced by the result. For integer format, conversion of a shorter data type to a longer is done by sign extension; conversion of longer to a shorter is done by truncation of the higher numbered (most significant) bits. For floating format, the form of the con version is as follows: CVTBF exact CVTFW truncated CVTBD exact CVTDW truncated CVTWF exact CVTFL truncated CVTWD exact CVTRFL truncated CVTLF rounded CVTDL truncated CVTLD exact CVTRDL rounded CVTFB truncated CVTFD exact CVTDB truncated CVTDF rounded
Notes	1. Integer overflow occurs if any truncated bits of the source operand are not equal to the sign bit of the destination operand. 2. Only converts with an integer destination operand can result in integer overflow. On integer overflow, the destination operand is replaced by the low order bits of the true results. 3. Only CVTDF can result in floating overflow. On floating overflow, the destination operand is replaced by an operand of all 0 bits except for a sign bit of 1 (a reserved operand). N ← 1; Z ← 0; V ← 1; and C ← 0. 4. Only converts with a floating point source operand can result in a reserved operand fault. On a reserved operand fault, the destination operand in unaffected and the condition codes are unpredictable.

Example 1

```
.text
main: .word 0

      movb $-1, r1
      cvtbw r1, r2      # r2 will contain 0xFFFF

      clrl r2

      movl $0x123, r1
      cvtlb r1   , r2   # r2 will contain 0x23. V = 1

      halt
```

11.3.26. DEC DECREMENT

Purpose	subtract 1 from an integer		
Format	opcode dif.mx		
Operation	dif ← dif - 1;		
Condition codes	N ← dif LSS 0; Z ← dif EQL 0; V ← {integer overflow}; C ← {borrow from most significant bit};		
Exceptions	Integer overflow		
Opcodes	97 B7 D7	DECB DECW DECL	Decrement Byte Decrement Word Decrement Long
Description	One is subtracted from the difference operand and the difference operand is replaced by the result.		
Notes	1. Integer overflow occurs if the largest negative integer is decremented. On overflow, the difference operand is re placed by the largest positive integer. 2. DECx dif is equivalent to SUBx2 $1, dif, but is shorter.		

Example 1:

```
.text
      .word 0
      movl $7, r0
      decl r0
      pushl r0
      pushal format
      calls $2, .printf
      pushl $0
      calls $1, .exit
format: .asciz "%d\n"
```

Example 2

Flags example - the following program demonstrate the different values for flags while performing DEC commands: We can say that N and Z are set as always. V is set if 80..0 is decremented to 7F..F. C is set if 0 is decremented to FF..F.

```
.text
      .word 0
      movl $3, r0
      decl r0
      decl r0
      decl r0      # Z = 1
      decl r0      # N = 1, C = 1

      movb $0x81, r0
      decb r0      # N = 1
      decb r0      # N = 0, C = 0, V = 1
      decb r0      # N = 0, C = 0, V = 0

      pushl $0
      calls $1, .exit
```

Example 3

Another flags example

```
.text
main: .word 0

      movb $0, r0
      decb r0            # N = 1, C = 1
      movb $0xFF, r0
      decb r0            # N = 1
      movw $0x0000, r0
      decw r0            # N = 1, C = 1
      movl $0x00000000, r0
      decl r0            # N = 1, C = 1
      movl $0x80000000, r0
      decl r0            # V = 1
      movw $0x8000, r0
      decw r0            # V = 1
      movb $0x80, r0
      decb r0            # V = 1

      pushl $0
      calls $1, .exit
```

11.3.27. DIV DIVIDE

Purpose	perform arithmetic division
Format	opcode divr.rx, quo.mx 2 operand opcode divr.rx, divd.rx, quo.wx 3 operand
Operation	quo ← quo / divr; 2 operand quo ← divd / divr; 3 operand
Condition codes	N ← quo LSS 0; Z ← quo EQL 0; V ← {overflow} OR {divr EQL 0}; C ← 0;
Exceptions	Integer overflow Divide by zero Floating overflow Floating underflow Reserved operand
Opcodes	86 DIVB2 Divide Byte 2 Operand 87 DIVB3 Divide Byte 3 Operand A6 DIVW2 Divide Word 2 Operand A7 DIVW3 Divide Word 3 Operand C6 DIVL2 Divide Long 2 Operand 07 DIVL3 Divide Long 3 Operand 46 DIVF2 Divide Floating 2 Operand 47 DIVF3 Divide Floating 3 Operand 66 DIVD2 Divide Double 2 Operand 67 DIVD3 Divide Double 3 Operand
Description	In 2 operand format, the quotient operand is divided by the divisor operand and the quotient operand is replaced by the result. In 3 operand format, the dividend operand is divided by the divisor operand and the quotient operand is replaced by the result. In floating format, the quotient operand result is rounded for both 2 and 3 operand format.
Notes	1. Integer division is performed such that the remainder (unless it is zero) has the same sign as the dividend; i.e., the result is truncated towards 0. 2. Integer overflow occurs if and only if the largest negative

	integer is divided by -1. On overflow, operands are affected as in 3 below. 3. In the integer divisor operand is 0, then in 2 operand integer format, the quotient operand is not affected; in 3 operand format the quotient operand is replaced by the dividend operand. 4. On a floating reserved operand fault, the quotient operand is unaffected and the condition codes are un predictable. 5. On floating underflow, the quotient operand is replaced by 0. 6. On floating divide by zero or on floating overflow the quotient operand is replaced by an operand of all bits 0 except for a sign bit of 1 (a reserved operand). N ← 1; Z ← 0; V ← 1; and C ← 0.

11.3.28. EDIV EXTENDED DIVIDE

Purpose	perform extended-precision division
Format	opcode divr.rl, divd.rq, quo.wl, rem.wl
Operation	quo ← divd/divr; rem ← REM{dvid, divr};
Condition codes	N ← quo LSS 0; Z ← que EQL 0; V ← {integer overflow} OR {divr EQL 0}; C ← 0;
Exceptions	Integer overflow Divide by zero
Opcodes	7B EDIV Extended Divide
Description	The dividend operand is divided by the divisor operand; the quotient operand is replaced by the quotient and the remainder operand is replaced by the remainder.
Notes	1. The division is performed such that the remainder operand (unless it is 0) has the same sign as the dividend operand. 2. On overflow, or if the divisor operand is 0, then the quotient operand is replaced by bits 31:0 of the dividend operand, and the remainder is replaced by 0.

11.3.29. EMOD EXTENDED MULTIPLY AND INTEGERIZE

Purpose	perform accurate range reduction of math function arguments
Format	opcode mulr.rx, mulrx.rb, mud.rx, int.wl, fract.wx
Operation	int ← integer part of muld* {mulr'mulrx}; frac ← fractional part of muld * {mulr'mulrx);
Condition codes	N ← fract LSS 0; Z ← fract EQL 0; V ← {integer overflow}; C ← 0;
Exceptions	Integer overflow Floating underflow Reserved operand
Opcodes	54 EMODF Extended Multiply and Integrate Floating 74 EMODD Extended Multiply and Integrate Double
Description	The floating point multiplier extension operand (second operand) is concatenated with the floating point multiplier (first operand) to gain eight additional low order fraction bits. The multiplicand operand is multiplied by the extended multiplier operand. After multiplication, the integer portion is extracted and a 32-bit (EMODF) or 64-bit (EMODD) floating point number is formed from the fractional part of the product by truncating extra bits. The multiplication is such that the result is equivalent to the exact product truncated (before normalization) to a fraction field of 32 bits in floating and 64 bits in double. Regarding the result as the sum of an integer and fraction of the same sign, the integer operand is replaced by the integer part of the result and the fraction operand is replaced by the rounded fractional part of the result.
Notes	1. On a reserved operand fault, the integer operand and the fraction operand are unaffected. The condition codes are unpredictable. 2. On floating underflow, the integer and fraction operands are replaced by zero. 3. On integer overflow, the integer operand is replaced by the low order bits of the true result. 4. Floating overflow is indicated by integer overflow; how ever, integer overflow is possible in the absence of floating

	overflow.

11.3.30. EMUL EXTENDED MULTIPLY

Purpose	perform extended-precision multiplication
Format	opcode mulr.rl, muld.rl, add.rl, prod.wq
Operation	prod ← muld * mulr + SEXT(add);
Condition codes	N ← prod LSS 0; Z ← prod EQL 0; V ← 0; C ← 0;
Exceptions	None
Opcodes	7A EMUL Extended Multiply
Description	The multiplicand operand is multiplied by the multiplier operand giving a double length result. The addend operand is sign extended to double length and added to the result. The product operand is replaced by the final result.

11.3.31. HALT

Purpose	stop processor operation
Format	opcode
Operation	It PSL<current_mode> NEQU kernel then {reserved to DIGITAL opcode fault} else {halt the processor};
Condition codes	N ← N; Z ← Z; V ← V; C ← C;
Exceptions	none
Opcodes	00 HALT Halt
Description	If the process is running in kernel mode, the processor is halted.

11.3.32. INC INCREMENT

Purpose	add 1 to an integer
Format	opcode sum.mx
Operation	sum ← sum +1;
Condition codes	N ← sum LSS 0; Z ← sum EQL 0; V ← {integer overflow}; C ← {carry from most significant bit};
Exceptions	Integer overflow
Opcodes	96 Increment Byte 66 Increment Word D6 Increment Long
Description	One is added to the sum operand and the sum operand is replaced by the result.
Notes	1. Arithmetic overflow occurs if the largest positive integer is incremented. On overflow, the sum operand is replaced by the largest negative integer. 2. INCx sum is equivalent to ADDx2 $1, sum, but is shorter.

Example 1

Simple use of INCL opcode:

```
.text
main: .word 0

movl $5, r1
pushl r1
pushal format
calls $3, .printf       # R1 is 5
incl r1
pushl r1
pushal format
calls $3, .printf       # R1 is 6

pushl $0
calls $1, .exit

.data
format: .asciz "R1 is %d\n"
```

Example 2

The following example shows the different flags rise while using INC opcodes:

```
.text

main: .word 0

movb $0, r0
incb r0             # r0 contains 1, all flags are 0

movb $0xFF, r0
incb r0             # r0 contains 0, C = 1, V = 0, Z = 1

movw $0xFFFF, r0
incw r0             # r0 contains 0, C = 1, V = 0, Z = 1

movl $0xFFFFFFFF, r0
incl r0             # r0 contains 0, C = 1, V = 0, Z = 1

movl $0x7FFFFFFF, r0
incl r0             # r0 contains 0x80000000, C = 0, V = 1, N = 1

movw $0x7FFF, r0
incw r0             # r0 contains 0x8000, C = 0, V = 1, N = 1

movb $0x7F, r0
incb r0             # r0 contains 0x80, C = 0, V = 1, N = 1

halt
```

11.3.33. INDEX COMPUTE INDEX

Purpose	calculation of arrays of fixed length data, bit fields, and strings
Format	opcode subscript.rl, low.rl, high.rl, size.rl, indexin.rl, indexout.wl
Operation	indexout ← {indexin + subscript} *size; if {subscript LSS low} or {subscript GTR high} then {subscript range trap};
Condition codes	N ← indexout LSS 0; Z ← indexout EQL 0; V ← 0; C ← 0;
Exceptions	subscript range
Opcodes	OA INDEX Index
Description	The indexin operand is added to the subscript operand and the sum is multiplied by the size operand. The indexout operand is replaced by the result. If the subscript operand is less than the low operand or greater than the high operand, a sub script range trap is taken.
Notes	1. No arithmetic exception other than subscript range can result from this instruction. Thus no indication is given if overflow occurs in either the add or multiply steps. If overflow occurs on the add step the sum is the low order 32 bits of the true result. It overflow occurs on the multiply step the indexout operand is replaced by the low order 32 bits of the true product of the sum and the subscript operand. In the normal use of this instruction, overflow cannot occur without a subscript range trap occurring. 2. The index instruction is useful in index calculations for arrays of the fixed length data types (integer and floating) and for index calculations for arrays of bit fields, character strings, and decimal strings. The indexin operand permits cascading INDEX instructions for multidimensional arrays. For one-dimensional bit field arrays it also permits introduction of the constant portion of an index calculation which is not readily absorbed by address arithmetic.

11.3.34. INSQUE INSERT ENTRY IN QUEUE

Purpose	add entry to head or tail of queue
Format	opcode entry.ab, pred.ab

Operation	If (all memory accesses can be completed) then
	begin
	(entry) ← (pred); forward link of entry
	(entry+4) ← pred; backward link of entry
	((pred)+4) ← entry; backward link of successor
	(pred) ← entry; forward link of predecessor
	end;
	else
	begin
	{backup instruction};
	{initiate fault}
	end;

Condition codes	N ← (entry) LSS (entry+4); Z ← (entry) EQL (entry+4); first entry in queue V ← 0; C ← (entry) LSSU (entry+4);
Exceptions	None
Opcodes	OE INSQUE Insert Entry in Queue
Description	The entry specified by the entry operand is inserted into the queue following the entry specified by the predecessor operand. If the entry inserted was the first one in the queue, the condition code Z-bit is set; otherwise it is cleared. The insertion is a non-interruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state.
Notes	1.Because the insertion is non-interruptible, processes running in kernel mode can share queues with interrupt service routines. 2. The INSQUE and REMQUE instructions are implemented

	such that cooperating software processes in a single processor may access a shared list without additional synchronization if the insertions and removals are only at the head or trail of the queue. 3. During access validation, any access which cannot be completed results in a memory management exception even though the queue insertion is not started. 4. The instruction is similar to the interlocked sequence MOVL pred, temp ref MOVAB pred, entry + 4 MOVAB entry, 4(tmp reg) MOVL temp reg, entry MOVAB entry, pred

Example 1

```
.text
main: .word 0
      remque head,temp  #removing an member from empty queue works!
      insque mem1,head
      insque mem2,head
      insque mem3,mem2  # now the queue should be mem2,mem3,mem1
      remque mem3,temp  # now queue should ne mem2,mem1
      remque mem2,temp  # only mem1 left
      remque mem1,temp  # empty again
      pushl $0
      calls $1,.exit


.data
# queue head. first long is the head pointer, second long is the tail
pointer.
# initialize to empty queue, i.e. both pointers point to the head.
head: .long head,head
# queue menebers. first long is NEXT pointer, second long is the PREV
pointer
mem1: .long 0,0
mem2: .long 0,0
mem3: .long 0,0

temp: .long 0,0
```

11.3.35. LOCC SKPC LOCATE CHARACTER, SKIP CHARACTER

Purpose	to find or skip character in character string
Format	opcode char.rb, len.rw, addr.ab
Operation	Compare each character until equal (LOCC) or not equal (SKPC). Z set if condition not satisified.
Condition codes	N ← 0; Z ←R0 EQL 0; V ← 0; C ← 0;
Exceptions	None
Opcodes	3A LOCC Locate Character 3B SKPC Skip Character
Description	The character operand is compared with the bytes of the string specified by the length and address operands. Comparison continues until equality is detected for the Locate Character instruction or inequality for the Skip Character instruction or until all bytes of the string have been compared. If equality is detected for the Locate Character instruction, the condition code Z bit is cleared; otherwise the Z bit is set. If inequality is detected for the Skip Character instruction, the condition code Z bit is cleared: otherwise the Z bit is set.
Notes	1. After execution: R0 = number of bytes remaining in the string (including located one) if byte located; otherwise R0 = 0. R1 = address of the byte located if byte located; other R1 = address of one byte beyond the string. 2. It the string has zero length, condition code Z is set just as though each byte of the entire string were equal (unequal) to the character.

Example 1

```
.text
main: .word 0
      locc $32, $len, str
      pushl r0
      pushal format
      calls $2, .printf

      pushl $0
      calls $1, .exit
.data
.set len, 11
str:  .asciz "abc def ghi"
format: .asciz "Characters left: %d\n"
```

11.3.36. MATCHC Match Characters

Purpose	to find substring (object) in character string
Format	opcode objlen.rw, objaddr.ab, srclen.rw, srcaddr.ab
Operation	search the string located on srcaddr for full string match of object
Condition codes	N ← 0; V ←R0 EQL 0; V ← 0; C ← 0;
Exceptions	None
Opcodes	39 MATCHC Match Characters
Description	The source string specified by the source length and source address operands is searched for a substring which matches the object string specified by the object length and object ad dress operands. If the substring is found, the condition code Z bit is set; otherwise, it is cleared.
Notes	1. After Execution: R0 = if a match occurred 0; otherwise the number of bytes in the object string. Rl = if a match occurred, the address of one byte beyond the object string; otherwise the address of the object string. R2 = if a match occurred, the number of bytes remaining in the source string after the match; otherwise 0. R3 = if a match occurred, the address of 1 byte beyond the last byte matched; otherwise the address of 1 byte beyond the source 2. If both strings have zero length or if the object string has zero length, condition code Z is set just as though the substring were found. 3. If the source string has zero length and the object string has non-zero length, condition code Z is cleared just a though the substring were not found.

11.3.37. MCOM MOVE COMPLEMENTED

Purpose	move the logical complement of an integer
Format	opcode src.rx, dst.wx
Operation	dst ← NOT src
Condition codes	N ← dst LSS 0; Z ← dst EQL 0; V ← 0; C ← C;
Exceptions	None
Opcodes	92 MCOMB Move Complemented Byte B2 MCOMW Move Complemented Word D2 MCOML Move Complemented Long
Description	The destination operand is replaced by the ones complement of the source operand.

Example 1

```
.text
main: .word 0
      mcomb r1, r2       # r2 = 0xFF
      mcomw r2, r3       # r3 = 0xFF00
      mcoml r3, r4       # r4 = 0xFFFF00FF
      halt
```

11.3.38. MNEG MOVE NEGATED

Purpose	move the arithmetic negation of a scalar quantity
Format	opcode src.rx, dst.wx
Operation	dst ← -scr;
Condition codes	N ← dst LSS 0; Z ← dst EOL 0; V ← overflow (Integer); V ← 0 (floating); C ← dst NEQ 0 (integer); C ← 0 (floating);
Exceptions	Integer overflow; reserved operand (floating)
Opcodes	8E MNEGB Move Negated Byte AE MNEGW Move Negated Word CE MNEGL Move Negated Long 52 MNEGF Move Negated Floating 72 MNEGD Move Negated Double
Description	The destination operand is replaced by the negative of the source operand.
Notes	1. Integer overflow occurs if the source Operand is the largest negative integer (which has no positive counterpart). On overflow, the destination operand is replaced by the source operand. 2. On floating reserved operand fault, the destination operand is unaffected and the condition codes are unpredictable. 3. If source is positive zero, result is positive zero. If source is reserved operand (minus zero), a reserved operand fault occurs. For all other floating point source values, bit 15 (sign bit) is complemented.

Example 1

```
.text
main: .word 0
      movb $1, r1         # r1 contains 1
      mnegw r1, r2        # r2 contains 0xFFFF
      mnegw r2, r3        # r3 contains 1
      mnegl r3, r4        # r4 contains 0xFFFFFFFF
      halt
```

11.3.39. MOV

Purpose	move a scalar quantity
Format	opcode src.rx, dst.wx
Operation	dst ← src
Condition codes	N ← dst LSS 0 Z ← dst EQL 0 V ← 0 C ← C
Exceptions	None (integer); Reserved operand (floating point)
Operation codes	90 MOVB Move Byte BO MOVW Move Word DO MOVL Move Long 7D MOVQ Mode Quad 50 MOVF Move Floating 70 MOVD Move Double
Description	The destination operand is replaced by the source operand. The source operand is unaffected.
Notes	1. On a floating reserved operand fault, the destination operand is unaffected and the condition codes are unpredictable. 2. Unlike the POP-11, but like the other VAX-11 instructions, MOVB and MOVW do not modify the high order bytes of a register destination. Refer to the MOVZxL and CVTxL instructions to update the full register contents.

Example 1

In the following program we put 0 in r0, and then printing it, and printing the PSW. Then we put -1 in r0 and print the PSW again. PSW will be 4 and then 8 (On the beginning the third bit will be 1, and on the end the fourth bit will be set).

```
.text

main: .word 0
      # put 0 in r0. Zero flag will raised
      movl $0, r0
      movpsl r1   # r1 will contains the PSL
```

```
        movw r1, r2 # save only the PSW to r2
        pushl r0
        pushl r2
        pushal format
        calls $3, .printf

        # put negative number in r0. Neg flag should rised
        movl $-1, r0
        movpsl r1   # r1 will contains the PSL
        movw r1, r2 # save only the PSW to r2
        pushl r0
        pushl r2
        pushal format
        calls $3, .printf

        pushl $0
        calls $1, .exit


format: .asciz "PSW is %d. R0 is %d\n"
```

Example 2

The programs test the different types of mov commands: movl, movw and movb.

It will print 12345678, then 5678 and then 78.

```
.text

main: .word 0
        movl $0x12345678, r1
        movw r1, r2
        movb r2, r3
        pushl r3
        pushl r2
        pushl r1
        pushal format
        calls $4, .printf

        pushl $0
        calls $1, .exit


format: .asciz "R1 is %X, R2 is %X, R3 is %X\n"
```

Example 3

The following program demonstrates the using of the movq opcode:

```
.text
main: .word 0
movl $1000, r0
movl $1004, r1
movl $0x12345678, (r0)
movl $0x54321234, (r1)
movq (r0), r4      # R4 contains 0x12345678, R5 contains 0x54321234

halt
```

11.3.40. MOVA, PUSHA MOVE ADDRESS, PUSH ADDRESS

Purpose	calculate address of quantity		
Format	opcode src.ax, dst.wl		MOVA
	opcode src.ax		PUSHA
Operation	dst ← src;		MOVA
	-(SP) ← src;		PUSHA
Condition codes	N ← result LSS 0; Z ← result EQL 0; V ← 0; C ← C;		
Exceptions	None		
Opcodes	9E	MOVAB	Move Address Byte
	3E	MOVAW	Move Address Word
	DE	MOVAL	Move Address Long
	DE	MOVAF	Move Address Floating
	7E	MOVAQ	Move Address Quad
	7E	MOVAD	Move Address Double
	9F	PUSHAB	Push Address Byte
	3F	PUSHAW	Push Address Word
	DF	PUSHAL	Push Address Long
	DF	PUSHAF	Push Address Floating
	7F	PUSHAQ	Push Address Quad
	7F	PUSHAD	Push Address Double
Description	For MOVA, the destination operand is replaced by the source operand which is an address. For PUSHA, the source operand is pushed on the stack. The context in which the source operand is evaluated is given by the data type of the instruction. The operand whose address replaces the destination operand is not referenced.		
Notes	1. The source operand is of address access type which causes the address of the specified operand to be moved. 2. PUSHAx is equivalent to MOVAx src, -(SP), but is shorter.		

Example 1

```
.text

main: .word 0
      moval myString, r0
      pushl r0
      calls $1, .puts

      pushl $0
      calls $1, .exit

.data
myString: .asciz "Hello, World"
```

11.3.41. MOVC MOVE CHARACTER

Purpose	to move character string or block of memory
Format	3 operands: opcode len.rw srcaddr.ab, dstaddr.ab 5 operands: opcode srclen.rw, srcaddr.ab, fill.rb, dstlen.rw, dstaddr.ab
Operation	Copy len bytes from srcaddr to dstaddr MOVC3 Copy min(srclen, dstlen) bytes from srcaddr to dstaddr MOVC5 If dstlen > srclen then fill the rest of dstaddr with fill
Condition codes	N ← srclen LSS dstlen; Z ← srclen EQL dstlen; V ← 0; C ← srclen LSSU dstlen;
Exceptions	None
Opcodes	28 MOVC3 Move Character 3 Operand 2C MOVC5 Move Character 5

	Operand
Description	In 3 operand format, the destination string specified by the length and destination address operands is replaced by the source string specified by the length and source address operands. In 5 operand format, the destination string specified by the destination length and destination address operands is replaced by the source string specified by the source length and source address operands. If the destination string is longer than the source string, the highest address bytes of the destination are replaced by the fill operand. If the destination string is shorter that the source string, the highest addressed bytes of the source string are not moved. The operation of the instruction is such that overlap of the source and destination strings does not affect the result.
Notes	1. After execution of MOVC3: R0 = 0 R1 = address of one byte beyond the source string R2 = 0 R3 = address of one byte beyond the destination string R4 = 0 R5 = 0 2. After execution of MOVC5: R0 = number of unmoved bytes remaining in source string. R0 is non-zero only if source string is longer than destination string R1 address of one byte beyond the last byte in source string that was moved R2 = 0 R3 = address of one byte beyond the destination string R4 = 0 R5 = 0 3. MOVC3 is the preferred way to copy one block of memory to another. 4. MOVC5 with a 0 source length operand is the preferred way to fill a block of memory with the fill character. 5. On MOVC3, or if the MOVC5 and the strings are of equal length, then Z is set and N, V, and C are cleared.

Example 1

```
.text
main: .word 0
      movc3 $13, strHello, strBuffer

      pushal strBuffer
      calls $1, .puts

      pushl $0
      calls $1, .exit

.data
strHello:   .asciz "Hello, World"
strBuffer:  .space 80
```

Example 2

```
.text
main: .word 0
      movc5 $0, strBuffer, $'a, $79, strBuffer

      pushal strBuffer
      calls $1, .puts

      pushl $0
      calls $1, .exit

.data
strBuffer:  .space 80
```

11.3.42. MOVPSL MOVE FROM PSL

Purpose	obtain processor status
Format	opcode dst.wl
Operation	dst ← PSL;
Condition codes	N ← N; Z ← Z; V ← V; C ← C;
Exceptions	none
Opcodes	DC MOVPSL Move from PSL
Description	The destination operand is replaced by the processor status longword
Notes	

Example 1

In the following program we put 0 in r0, and then printing it, and printing the PSW. Then we put -1 in r0 and print the PSW again. PSW will be 4 and then 8 (On the beginning the third bit will be 1, and on the end the fourth bit will be set).

```
.text

main: .word 0
      # put 0 in r0. Zero flag will raised
      movl $0, r0
      movpsl r1   # r1 will contains the PSL
      movw r1, r2 # save only the PSW to r2
      pushl r0
      pushl r2
      pushal format
      calls $3, .printf

      # put negative number in r0. Neg flag should rised
      movl $-1, r0
      movpsl r1   # r1 will contains the PSL
      movw r1, r2 # save only the PSW to r2
      pushl r0
      pushl r2
      pushal format
      calls $3, .printf
```

```
        pushl $0
        calls $1, .exit


format: .asciz "PSW is %d. R0 is %d\n"
```

11.3.43. MOVTC MOVE TRANSLATED CHARACTERS

Purpose	to move and translate character string
Format	opcode srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab, dstlen.rw, dstaddr.ab
Operation	
Condition codes	N ← srclen LSS dstlen; Z ← srclen EQL dstlen; V ← 0; C ← srclen LSSU dstlen;
Exceptions	None
Opcodes	2E MOVTC Move Translated Characters
Description	The source string specified by the source length and source address operands is translated and replaces the destination string specified by the destination length and destination address operands. Translation is accomplished by using each byte of the source string as an index into a 256-byte table whose zeros entry address is specified by the table address operand. The byte selected replaces the byte of the destination string. If the destination string is longer than the source string, the highest addressed bytes of the destination string are replaced by the fill operand. If the destination string shorter than the source string, the highest addressed bytes of the source string are not translated and moved. The operation of the instruction is such that overlap of the source and destination strings does not affect the result. It the destination string overlaps the translation table, the destination string is unpredictable.
Notes	1 After execution: R0 = number of translated bytes remaining in source string, R0 is non-zero only if source string is longer than destination string. R1 = address of one byte beyond the last byte in Source string that was translated.

	R2 = 0 R3 = address of the translation table. R4 = 0 R5 = address on one byte beyond the destination string

Example 1

```
.text

main: .word 0

      movtc $9, myString, $0, TranslateTable, $20, dstString

      pushal dstString
      calls $1, .puts
      pushl $0
      calls $1, .exit

.data
      myString: .asciz "abcd abcd"
      dstString: .space 20

TranslateTable:
      .space 32
      .byte 32
      .space 97-33
      .byte 'b, 'c, 'd, 'e
```

11.3.44. MOVTUC MOVE TRANSLATED UNTIL CHARACTER

Purpose	to move and translate character string, handling escape codes
Format	opcode srclen.rw, srcaddr.ab, esc.rb, tbladdr.ab, dstlen.rw, dstaddr.ab
Operation	
Condition codes	N ← srclen LSS dstlen; Z ← srclen EQL dstlen; V ← {terminated by escape}; C ← srclen LSSU dstlen;
Exceptions	None
Opcodes	2F MOVTUC Move Translated Until Character
Description	The source string specified by the source length and source address operands is translated and replaces the destination/s specified by the destination length and destination address operands. Translation is accomplished by using each byte of the source string as index into a 256-byte table Whose zeros entry address is specified by the table address operand. The byte selected replaces the byte of the destination string. Translation continues until a translated byte is equal to the escape byte or until the source string or destination string is exhausted. if translation is terminated because of escape the condition code V-bit is set; otherwise, his cleared. if the destination string overlaps the source string or the table, destination string and R0 through R5 are unpredictable.
Notes	1. After execution: R0 = number of bytes remaining in source string (including the byte which caused the escape). R0 is zero only if the entire source string was translated and moved without escape. R1 = address of the byte which resulted in destination string exhaustion or escape; or if no exhaustion or escape, Rl = address of one byte beyond the source string. R2 = 0 R3 = address of the table. R4 = number of bytes remaining in the destination string. R5 = address of the byte in the destination string which would have received the translated byte that caused the escape or would have received a translated byte if the source string were not exhausted; or if no exhaustion or escape, R1 = address of

	one byte beyond the destination string. 2. V should be tested before the V and C condition codes to make sure that an escape is detected on the last character of the source string.

11.3.45. MOVZ MOVE ZERO-EXTENDED

Purpose	convert an unsigned integer to a wider unsigned integer
Format	opcode src.rx, dst.wy
Operation	dst ← ZEXT (src);
Condition codes	N ← 0; Z ← dst EQL 0; V ← 0; C ← C;
Exceptions	None
Opcodes	9B MOVZBW Move Zero-Extended Byte to Word 9A MOVZBL Move Zero-Extended Byte to Long 3C MOVZWL Move Zero-Extended Word to Long
Description	For MOVZBW, bits 7:0 of the destination operand are replaced by the source operand; bits 15:8 are replaced by zero. For MOVZBL, bits 7:0 of the destination operand are replaced by the source operand; bits 31:8 are replaced by 0. For MOVZWL, bits 15:0 of the destination operand are replaced by the source operand; bits 31:16 are replaced by 0.

Example 1

```
.text
main: .word 0
      movb $255, r0
      movzbl r0, r1
      pushl r1
      pushal format
      calls $2, .printf
      pushl $0
      calls $1, .exit

.data
format: .asciz "R1 is %d"
```

11.3.46. MUL MULTIPLY

Purpose	perform arithmetic multiplication		
Format	opcode mulr.rx, prod.mx		2 operand
	opcode mulr.rx, muld.rx, prod.wx		3 operand
Operation	prod ← prod * mulr;		2 operand
	prod ← muld * mulr;		3 operand
Condition codes	N ← prod LSS 0; Z ← prod EQL 0; V ← overflow; C ← 0;		
Exceptions	Integer overflow Floating overflow Floating underflow Reserved operand		
Opcodes	84	MULB2	Multiply Byte 2 Operand
	85	MULB3	Multiply Byte 3 Operand
	A4	MULW2	Multiply Word 2 Operand
	A5	MULW3	Multiply Word 3 Operand
	C4	MULL2	Multiply Long 2 Operand
	C5	MULL3	Multiply Long 3 Operand
	44	MUIF2	Multiply Floating 2 Operand
	45	MULF3	Multiply Floating 3 Operand
	64	MULD2	Multiply Double 2 Operand
	65	MULD3	Multiply Double 3 Operand
Description	In 2 operand format, the product operand is multiplied by the multiplier operand and the product operand is replaced by the result. In 3 operand format, the multiplicand operand is multiplied by the multiplier operand and the product operand is replaced by the result. In floating format, the product operand result is rounded for both 2 and 3 operand format.		
Notes	1. Integer overflow occurs if the high half of the double length result is not equal to the sign extension of the low half. On integer overflow, the product operand is replaced by the low order bits of the true result. 2. On a floating reserved operand fault, the product operand is		

	unaffected and the condition codes are unpredictable. 3. On floating underflow, the product operand is replaced by 0. 4. On floating overflow, the product operand is replaced by an operand of all bits 0 except for a sign bit of 1 (a re served operand). N← 1: Z ← 0; V ← 1; and C ← 0.

Example 1

The program multiplies the values stored in R1 and R2 and put the result in R3.

```
.text

main: .word 0
      movl  $3, r1
      movl  $2, r2
      mull3 r1, r2, r3
      pushl r3
      pushal format
      calls $2, .printf
      pushl $0
      calls $1, .exit
.data
format: .asciz "R3 is %d"
```

11.3.47. POLY POLYNOMINAL EVALUATION

Purpose	allows fast calculation of math functions
Format	opcode arg.rx, degree.rw, tbladdr.ab
Operation	result C ← degree; For degree times, loop result ← arg * result; Perform multiply, and retain an extended floating traction of 31 bits (POLYF) or 63 bits (POLYD) (the fraction is truncated before normalization) use this result in the following step result ← result + next coefficient; normalize, round, and check for over/underflow only after the combined multiply/add sequence if overflow then trap; if underflow then clear result, remember underflow and continue looping;
Condition codes	N ← R0 LSS 0; Z ← R0 EQL 0; V ← {floating overflow}; C ← 0;
Exceptions	Floating overflow Floating underflow Reserved operand
Opcodes	55 POLYF Polynomial Evaluation Floating 75 POLYD Polynomial Evaluation Double
Description	The table address operand points to a table of polynomial coefficients. The coefficient of the highest order term of the polynomial is pointed to by the table address operand. The table is specified with lower order coefficients stored at in creasing addresses. The data type of the coefficients is the same as the data type of the argument operand. The evaluation is carried out by Homer's method and the contents of R0 (R1'R0 for POLYD) are replaced by the result. The result computed is: if d = degree and x = arg

	result = C[0] + x*(C[1] + x*(C[2] +...x*C[d])))
	The unsigned word degree operand specifies the highest numbered coefficient to participate in the evaluation.
Notes	1. After execution:
	POLYF
	R0 = result
	R1 = 0
	R2 = 0
	R3 = table address + degree*4 + 4
	POLYD
	R0 = high order part of result
	R1 = low order part of result
	R2 = 0
	R3 = table address + degree *8 + 8
	R4 = 0
	R5 = 0
	2. The multiplication is performed such that the precision of the product is equivalent to a floating point datum having a 31-bit (63-bit for POLYD) fraction.
	3. If the unsigned word degree operand is 0, the result is C0.
	4. If the unsigned word degree operand is greater than 31, a reserved operand exception occurs.
	5. On a reserved operand exception:
	• If PSL<FPD> = 0, the reserved operand is either the degree operand (greater than 31), or the argument operand, or some coefficient.
	• If PSL<FPD> = 1, the reserved operand is a coefficient, and R3 is pointing at the value which caused the exception.
	• The state of the saved condition codes and the other registers is unpredictable. If the reserved operand is changed and the contents of the condition codes and all registers are preserved, the fault is continuable.
	6. On floating underflow after the rounding operation, the temporary result is replaced by zero, and the operation continues. A floating underflow trap occurs at the end of the instruction it underflow occurred during any iteration of the computation loop. Note that the final result may be non zero if underflow occurred before the last iteration.
	7. On floating overflow after the rounding operation at any iteration of the computation loop, the instruction terminates

	and causes a trap. On overflow the contents of R2 and R3 (R2 through R5 for POLYD) are unpredictable. R0 contains the reserved operand (minus 0) and R1 = 0. 8. POLY can have both overflow and underflow in the same instruction. If both occur, overflow trap is taken; underflow is lost. 9. If the argument is zero and one of the coefficients in the table is the reserved operand, whether a reserved operand fault occurs is unpredictable.

11.3.48. POPR POP REGISTERS

Purpose	restore multiple registers from stack
Format	opcode mask.rw
Operation	
Condition codes	N ← N; Z ← Z; V ← V; C ← C;
Exceptions	None
Opcodes	BA POPR Pop Registers
Description	The contents of registers whose number corresponds to set bits in the mask operand are replaced by longwords popped from the stack. R[n] is replaced if mask <n> is set. The mask is scanned from bit 0 to bit 14. Bit 15 is ignored.
Notes	This instruction is similar to the sequence MOVL(SP)+,RO MOVL(SP)+,R1 … MOVL(SP)+,R14 where only the masked registers are popped.

11.3.49. PUSHL PUSH LONG

Purpose	push source operand onto stack
Format	opcode src.rl
Operation	-(SP) ← src;
Condition codes	N ← src LSS 0; Z ← src EQL 0; V ← 0: C ← C;
Exceptions	None
Operation codes	DD PUSHL Push Long
Description	The long word source operand is pushed on the stack.
Notes	PUSHL is equivalent to MOVL src, -(SP), but is shorter.

Example 1

The following program pushes the number 100 to the stack, and then reads it from there. r1 will contain 100 at the end of this program.

```
.text

main: .word 0
      pushl $100
      movl (sp), r1
      pushl r1
      pushal format
      calls $2, .printf

      pushl $0
      calls $1, .exit


format: .asciz "R1 is %d\n"
```

11.3.50. PUSHR PUSH REGISTERS

Purpose	save multiple registers or stack
Format	opcode mask.rw
Operation	
Condition codes	N ← N; Z ← Z; V ← V; C ← C;
Exceptions	None
Opcodes	BB PUSHR Push Registers
Description	The contents of registers whose number corresponds to set bits in the mask operand are pushed on the stack as long words. R[n] is pushed if mask <n> is set. The mask is scanned from bit 14 to bit 0. Bit 15 is ignored.
Notes	1. The order of pushing is specified so that the contents of higher numbered registers are stored at higher memory addresses. This results in a double floating datum stored in adjacent registers being stored by PUSHR in memory in the correct order 2. This instruction is similar to the sequence PUSHL R14 PUSHL R13 ... PUSHL R0 where only the masked registers are pushed.

Example 1:

The following program pushes r0 and r1 to the stack, and then print it.

```
.text
main: .word 0
      movl $1, r0
      movl $2, r1
      pushr $3
      pushal format
      calls $3, .printf
      pushl $0
      calls $1, .exit
.data
format: .asciz "%d %d\n"
```

11.3.51. REI - RETURN FROM EXCEPTION OR INTERRUPT

Purpose	exit from an exception or interrupt service routine
Format	opcode
Operation	tmp1←(SP)+; tmp2←(SP)+; if { tmp2 <current_mode> LSSU <current_mode> } or { tmp2<IS> EQLU 1 and PSL<IS> EQLU 0} or { tmp2<IS> EQLU 1 and tmp2<current_mode> NEQU 0 } or { tmp2<IS> EQLU 1 and tmp2<IPI> EQLU 0} or { tmp2<IPL> GRTU 0 and tmp2<current_mode> NEQU 0 } or { tmp2 <prev_mode> LSSU <current_mode> } or { tmp2<IPL> GRTU PSL<IPL> } or { tmp2<PSL_MBZ> NEQU 0} then {reserved operand fault}; if { tmp2<CM> EQLU 1} and { tmp2<FPD, IS, DV, FU, IV> NEQU 0 } or { tmp2<current_mode> NEQU 3}} then {reserved operand fault}; { disallow interrupts}; if PSL<IS> EQLU 1 then ISP←SP else PSL<current_mode>_SP←SP; if PSL<TP> EQLU 1 then tmp<TP>←1 PC←tmp1; PSL←tmp2;
Condition codes	N ← save PSL<3>; Z ← save PSL<2>; V ← save PSL<1>; C ← save PSL<0>;
Exceptions	reserved operand
Opcodes	02 REI Return from Exception or Interrupt
Description	A longword is popped from the current stack and held in a temporary PC. A second longword is popped from the current stack and held in a temporary PSL. Validity of the popped PSL is checked. The current stack pointer is saved and a new stack

	pointer is selected according to the new PSL current mode and IS fields. The level of the highest privilege AST is checked against the current access mode to see whether a pending AST can be delivered. Execution resumes with the instruction being executed at the time of the exception or interrupt. Any instruction look ahead in the processor is reinitialized.
Notes	1. The exception or interrupt service routine is responsible for restoring any registers saved and removing any parameters from the stack. 2. As usual for faults, any access violation or translation not valid conditions on the stack pops restore the stack pointer and fault. 3. The non-interrupt stack pointers may be fetched and stored by hardware either in internal registers or in their allocated slots in the Process Control Block. Only LDPCTX and SVPCTX always fetch and store in the Process Control Block. MFPR and MTPR always fetch and store the pointers whether in registers or the Process Control Block.

11.3.52. REMQUE REMOVE ENTRY IN QUEUE

Purpose	remove entry from head or tail of queue
Format	opcode entry.ab, addr.wl

Operation	If (all memory accesses can be completed) then
	begin
	((entry+4)) ← (entry); forward link of predecessor
	((entry)+4) ← (entry+4); backward link of successor
	addr ← entry;
	end;
	else
	begin
	{backup instruction};
	{initiate fault}
	end;

Condition codes	N ← (entry) LSS (entry+4); Z ← (entry) EQL (entry+4); removed last entry V ← entry EQL (entry+4); no entry to remove C ← (entry) LSSU (entry+4);
Exceptions	None
Opcodes	OF REMQUE Remove Entry from Queue
Description	The queue entry specified by the entry operand is removed from the queue. The address operand is replaced by the ad dress of the entry removed. If there was no entry in the queue to be removed, the condition code V bit is set; otherwise it is cleared, If the queue is empty at the end of this instruction, the condition code Z-bit is set; otherwise it is cleared. The removal is a non-interruptible operation. Before performing any part of the operation, the processor validates that the entire operation can be completed. This ensures that if a memory management exception occurs, the queue is left in a consistent state.
Notes	1. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt ser vice routines. 2. The INSQUE and REMQUE instructions are implemented such that cooperating software processes in a single processor

	may access a shared list without additional synchronization if insertions and removals are only at the head or tail of the queue. 3. During access validation, any access which cannot be completed results in a memory management exception even though the queue removal is not started.

Example 1

```
.text
main: .word 0
      remque head,temp  #removing an member from empty queue works!
      insque mem1,head
      insque mem2,head
      insque mem3,mem2  # now the queue should be mem2,mem3,mem1
      remque mem3,temp  # now queue should ne mem2,mem1
      remque mem2,temp  # only mem1 left
      remque mem1,temp  # empty again
      pushl $0
      calls $1,.exit


.data
# queue head. first long is the head pointer, second long is the tail
pointer.
# initialize to empty queue, i.e. both pointers point to the head.
head: .long head,head
# queue menebers. first long is NEXT pointer, second long is the PREV
pointer
mem1: .long 0,0
mem2: .long 0,0
mem3: .long 0,0

temp: .long 0,0
```

11.3.53. RET RETURN FROM PROCEDURE

Purpose	transfer control from a procedure back to calling program
Format	opcode
Operation	{restore SP from FP}; {restore registers}; {drop stack alignment}; {restore PSW}; {If CALLS, remove arglist};
Condition codes	N ← restored PSW <3>; Z ← restored PSW <2>; V ← restored PSW <1>; C ← restored PSW <0>;
Exceptions	reserved operand
Opcodes	SP is replaced by FP plus 4. A longword containing stack alignment bits in bits 31:30, a CALLS/CALLG flag in bit 29, the low 12 bits of the procedure entry mask in bits 27:16, and a saved in a temporary PC, FP, and AP are replaced by long-words popped from the stack. A register restore mask is formed from bits 27:16 of the temporary. Scanning from bit 0 to bit 11 of the restore mask, the contents of registers whose number is indicated by set bits in the mask are replaced by longwords popped from the stack. SP is replaced by the sum of SP and bits 31:30 of the temporary. PSW is replaced by bits 15:0 of the temporary. If bit 29 in the temporary is 1 (indicating that the procedure was called by CALLS), a longword containing the number of arguments is popped from the stack. Four times the unsigned value of the low byte of this longword is added to SP and SP is replaced by the result.
Description	1. A reserved operand fault occurs if temporary1 <15:8> NEG 0. 2. On a reserved operand fault, the condition codes are Unpredictable. The value of temporary1 <28> is ignored. 3. The procedure calling standard and condition handling facility assume that procedures which return a function value or a status code do so in R0 or R0 and R1.

Example 1

The following program demonstrate 3 functions calls.

```
.text
main: .word 0
      calls $0, func1
      calls $0, func1
      calls $0, func1

      pushl $0
      calls $1, .exit

func1:      .word 0

      movl $99, r1
      pushl r1
      pushal format
      calls $2, .printf
      ret
.data
format: .asciz "R1 is %d\n"
```

11.3.54. ROTL ROTATE LONG

Purpose	rotate of integer
Format	opcode cnt.rb, src.rl, dst.wl
Operation	dst ← src rotated cnt bits;
Condition codes	N ← dst LSS 0; Z ← dst EQL 0; V ← 0; C ← C;
Exceptions	None
Opcodes	9C ROTL Rotate Long
Description	The source operand is rotated logically by the number of bits specified by the court operand and the destination operand is replaced by the result. The source operand is unaffected. A positive count operand rotates to the left. A negative count operand rotates to the right. A 0 count operand replaces the destination operand with the source operand.
Notes	

Example 1

```
.text
.word 0
movl $0xff, r1
rotl $8, r1, r2          # r2 is 0xff00

movl $0xff000000, r1
rotl $8, r1, r2          # r2 is 0xff

movl $0xff, r1
rotl $-8, r1, r2         # r2 is $0xff000000

halt
```

11.3.55. RSB RETURN FROM SUBROUTINE

Purpose	return control from subroutine
Format	opcode
Operation	PC ← (SP) +;
Condition codes	N ← N; Z ← Z; V ← V; C ← C;
Exceptions	none
Opcodes	05 RSB Return from Subroutine
Description	PC is replaced by a longword popped from the stack.
Notes	1. RSB is used to return from subroutines called by the BSBB, BSBW and JSB instructions. 2. RSB is equivalent to JMP *(SP) +, but is one byte shorter.

11.3.56. SBWC SUBTRACT WITH CARRY

Purpose	perform extended-precision subtraction
Format	opcode sub.rl, dif.ml
Operation	dif ← dif - sub - C;
Condition codes	N ← dif LSS 0; Z ← dif EQL 0; V ← {integer overflow}; C ← {borrow from most significant bit};
Exceptions	Integer overflow
Opcodes	D9 SBWC Subtract with Carry
Description	The subtrahend operand and the contents of the condition code C bit are subtracted from the difference operand and the difference operand is replaced by the result.
Notes	1. On overflow, the difference operand is replaced by the low order bits of the true result.

	2. The two subtractions in the operation are performed simultaneously.

Example 1

```
.text

main: .word 0

      movl $10, r1
      movl $3, r2
      sbwc r2, r1 # r1 is 7

      movl $10, r1
      movl $3, r2
      bispsw $1
      sbwc r2, r1 # r1 is 6

      pushl $0
      calls $1, .exit
```

Example 2

```
.text

main: .word 0

      movl $0x80000000, r1
      movl $1, r2
      sbwc r2, r1       # V = 1

      movl $0x80000001, r1
      movl $1, r2
      bispsw $1
      sbwc r2, r1 # V = 1

      pushl $0
      calls $1, .exit
```

11.3.57. SCANC SPANC SCAN CHARACTERS, SPAN CHARACTERS

Purpose	to find or skip a set of characters in character string
Format	opcode len.rw, addr.ab, tbladdr.ab, mask.rb
Operation	Mask test each character until zero (SPANC) or nonzero (SCANC).
Condition codes	N ← 0; V ←R0 EQL 0; V ← 0; C ← 0;
Exceptions	None
Opcodes	2A SCANC Scan Characters 2B SPANC Span Characters
Description	The bytes of the string specified by the length and address operands are successively used to index into a 256 byte table whose zeroth entry address is specified by the table address operand. The byte selected from the table is ANDed with the mask operand. The operation continues until the result of the AND is non-zero for the SCANC instruction or zero for the SPANC instruction or until all the bytes of the string have been exhausted. If a non-zero AND result for the SCANC or a zero result for the SPANC is detected, the condition code Z-bit is cleared; otherwise, the Z-bit is set.
Notes	1. After execution: R0 = number of bytes remaining in the string (include the byte which produced the non-zero AND result for SCANC or zero result for SPANC). R0 is zero only if there was a zero AND result for SCANC or a non-zero result for SPANC. R1 = address of the byte which produced non-zero AND result for SCANC or a zero AND result for SPANC; Or, if zero result., R1 = address of one byte beyond the string. R2 = O R3 = address of the table 2. If the string has zero length, condition code z is set just as though the entire string were scanned (spanned).

11.3.58. SOB SUETRACT ONE AND BRANCH

Purpose	decrement integer loop count and loop
Format	opcode index.ml, displ.bb

Operation	index ← index -1: SOBGEQ
	If index GEQ 0 then
	PC ← PC + SEXT (displ);
	index ← index-1: SOBGTR
	If index GTR 0 then
	PC ← PC + SEXT (displ);

Condition codes	N ← index LSS 0; Z ← index EQL 0; V ← {integer overflow}; C ← C;
Exceptions	integer overflow
Opcodes	F4 SOBGEO Subtract One and Branch Greater Than or Equal F5 SOBGTR Subtract One and Branch Greater Than
Description	One is subtracted from the index operand and the index operand is replaced by the result. On SOBGEQ, If the index operand is greater than or equal to 0, the branch is taken. On SOBGTR, if the index operand is greater than 0, the branch is taken, if the branch is taken, the sign-extended branch displacement is added to the PC and the PC is replaced by the result.
Notes	1. Integer overflow occurs if the index operand before subtraction is the largest negative integer. On overflow, the index operand is replaced by the largest positive integer, and thus the branch is taken. 2. The C-bit is unaffected.

Example 1

The program makes a loop that prints the numbers 10 down to 1 on the screen.

```
.text
main: .word 0
      movl $10, r1

prnLoop:
      pushl r1
      pushal format
      calls $2, .printf
      sobgtr r1, prnLoop

      pushl $0
      calls $1, .exit

.data
format: .asciz "%d\n"
```

Example 2

The example shows the case when overflow occurs on SOBGEQ:

```
.text
main: .word 0
      movl $0x80000000, r2

prnLoop:
      pushl r2
      pushal format
      calls $2, .printf
      sobgeq r2, prnLoop # overflow on the first time we reach this
                         # line

      pushl $0
      calls $1, .exit

.data
format: .asciz "%ld\n"
```

11.3.59. SUB SUBTRACT

Purpose	perform arithmetic subtraction		
Format	opcode sub.rx, dif.mx		2 operand
	opcode sub.rx, min.rx, dif.wx		3 operand
Operation	dif ← dif - sub;		2 operand
	dif ← min - sub;		3 operand
Condition codes	N ← dif LSS 0; Z ← dif EQL 0; V ← overflow; C ← {borrow from most significant bit}(integer); C ← 0 (floating);		
Exceptions	Integer overflow Floating overflow Floating underflow Reserved operand		
Opcodes	82	SUBB2	Subtract Byte 2 Operand
	83	SUBB3	Subtract Byte 3 Operand
	A2	SUBW2	Subtract Word 2 Operand
	A3	SUBW3	Subtract Word 3 Operand
	C2	SUBL2	Subtract Long 2 Operand
	C3	SUBL3	Subtract Long 3 Operand
	42	SUBF2	Subtract Floating 2 Operand
	43	SUBF3	Subtract Floating 3 Operand
	62	SUBD2	Subtract Double 2 Operand
	63	SUBD3	Subtract Double 3 Operand
Description	In 2 operand format, the subtrahend operand is subtracted from the difference operand and the difference operand is replaced by the result. In 3 operand format, the subtrahend operand is subtracted from the minuend operand and the difference operand is replaced by the result. In floating format, the result is rounded.		
Notes	1. Integer overflow occurs if the input operands to the subtract are of different signs and the sign of the result is the sign of the subtrahend. On overflow, the difference operand is replaced by		

	the low order bits of the true result. 2. On a floating reserved operand fault, the difference operand unaffected and the condition codes are unpredictable. 3. On floating underflow, the difference operand is replaced by 0. 4. On floating overflow, the difference is replaced by an operand of all 0 bits except for a sign bit of 1 (a reserved operand). N ← 1; Z ← 0; V ← 1; and C ← 0.

Example 1

Flags Example:

```
.text

main: .word 0
movb $0x82, r0
subb2 $10, r0      # N = 0, V = 1

movw $0x8002, r0
subw2 $10, r0      # N = 0, V = 1

movw $0x8002, r0
subl2 $10, r0      # N = 0, V = 0

movl $0x80000002, r0
subl2 $10, r0      # N = 0, V = 1

movb $0x0, r0
subb2 $10, r0      # N = 1, C = 1

movw $0x0, r0
subw2 $10, r0      # N = 1, C = 1

movl $0x0, r0
subl2 $10, r0      # N = 1, C = 1

pushl $0
calls $1, .exit
```

11.3.60. TST TEST

Purpose	arithmetic compare of a scalar to 0.
Format	opcode src.rx
Operation	src ← 0;
Condition codes	N ← src LSS 0; Z ← src EQL 0; V ← 0; C ← 0;
Exceptions	None (integer); Reserved operand (floating point)
Opcodes	95 TSTB Test Byte B5 TSTW Test Word D5 TSTL Test Long 53 TSTF Test Floating 73 TSTD Test Double
Description	The condition codes are affected according to the value of the source operand.
Notes	1. TSTx src is equivalent to CMPx src, $0, but is shorter. 2. On a floating reserved operand, the condition codes are unpredictable.

Example 1

```
.text

main:
	.word 0
	movl $0, r0
	tstl r0
	movpsl r1
	pushl r1
	pushal format
	calls $2, .printf

	pushl $0
	calls $1, .exit

.data
format: .asciz "PSL is %d\n"
```

11.3.61. XOR EXCLUSIVE OR

Purpose	perform logical exclusive OR of two integers		
Format	opcode mask.rx, dst.mx		2 operand
	opcode mask.rx, src.rx, dst.wx		3 operand
Operation	dst ← dst XOR mask;		2 operand
	dst ← src XOR mask;		3 operand
Condition codes	N ← dst LSS 0;		
	Z ← dst EQL 0;		
	V ← 0;		
	C ← C;		
Exceptions	None		
Opcodes	8C	XORB2	Exclusive OR Byte 2 Operand
	8D	XORB3	Exclusive OR Byte 3 Operand
	AC	XORW2	Exclusive OR Word 2 Operand
	AD	XORW3	Exclusive OR Word 3 Operand
	CC	XORL2	Exclusive OR Long 2 Operand
	CD	XORL3	Exclusive Or Long 3 Operand
Description	In 2 operand format, the mask operand is XORed with the destination operand and the destination operand is replaced by the result. In 3 operand format, the mask operand is XORed with the source operand and the destination operand is re placed by the result.		

Example 1

Simple example of XORL3:

```
.text

main: .word 0
movl  $0xF0F0F0F0, r5
xorl3 $0x0A0B0C0D, r5, r6
pushl r6
pushal format
calls $2, .printf

pushl $0
calls $1, .exit

.data
format: .asciz "%X\n"
```

The program's output is FAFBFCFD.

Example 2

XORW3 example:

```
.text

main: .word 0
movw  $0xF0F0, r5
xorw3 $0x0FF0, r5, r6
pushl r6
pushal format
calls $2, .printf

pushl $0
calls $1, .exit

.data
format: .asciz "%X\n"
```

Program's output is FF00.

Example 3

Flags raised by XOR commands:

```
.text

main: .word 0
movw  $0xF0F0, r5
xorw3 $0x0FF0, r5, r6          # N = 1

movl  $0xF0F00000, r5
xorl3 $0x0FF00000, r5, r6      # N = 1

movb  $0xF0, r5
xorb3 $0x0F, r5, r6            # N = 1

movb  $0x00, r5
xorb3 $0x0F, r5, r6            # N = 0

movb  $0xFF, r5
xorb3 $0xFF, r5, r6            # N = 0, Z = 1

movw  $0xF0F0, r5
xorw2 $0x0FF0, r5              # N = 1

movl  $0xF0F00000, r5
xorl2 $0x0FF00000, r5          # N = 1

movb  $0xF0, r5
xorb2 $0x0F, r5                # N = 1

movb  $0x00, r5
xorb2 $0x0F, r5                # N = 0

movb  $0xFF, r5
xorb2 $0xFF, r5                # N = 0, Z = 1

pushl $0
calls $1, .exit

.data
format: .asciz "%X\n"
```

12. Old Simulator Bugs

12.1. Introduction

The following pages demonstrate the bugs we found on the old simulator, while writing the new one. Most of the bugs were found as we tried our testing program on both our simulator and the old simulator, and compare the results.
We wanted to demonstrate the importance of good testing program to a product.
Some of the bugs we found in our tests were unknown for around 20 years.

12.2. Assembler's bugs

While most of the bugs were found on the simulator, also the assembler has some bugs:

12.2.1. Problems with long labels & constants

When declaring long names (more than 19 characters) for variables or constants, the assembler cuts it and takes only the first 19 characters.
If we declare 2 variables that their first 19 characters are identical, the assembler declares 2 variables with the same name.
Example:

```
.text
.set AAAAAAAAAAAAAAAAAAAA, 2
.set AAAAAAAAAAAAAAAAAABBB, 3
```

12.2.2. Variables can be redefined

The assembler allows us to declare the same variable/constant several times.

Example:

```
.text
.set a, 2
.set a, 4
```

12.2.3. CALLS Parameter's bug

CALLS command fails to limit the number of function parameters to 255 as VAX-11 manual specific - it accepts any number. Example:

```
.text
main: .word 0
calls $1000, my_func

my_func:
      # ...
```

12.2.4. ASCIZ doesn't support "" as described on the manual

In the following example the first and the third lines should be legal, while the second and the forth lines supposed to be illegal. The old assembler takes all the four lines as illegal.

```
data:  .asciz "a""""""a"      # legal
data2: .asciz "a""""""a""     # illegal
data3: .asciz "a""""""""""a"  # legal
data5: .asciz """             # illegal
```

12.2.5. .WORD directive accepts negative numbers

As described in the manual, .WORD should accept only numbers from the range 0 .. 65,535. Yet the old simulator allows assigning negative values to .WORD directive.

Example:

```
.word -6
```

12.2.6. .INT directive accept illegal values

.INT directive shouldn't accept number bigger than 32,767 or smaller than -32,768, while the old simulator allows bigger number to pass compile.

Example:

```
.int 43457
```

12.2.7. .BYTE directive accept illegal values

.BYTE directive shouldn't accept number bigger than 127 or smaller than -128, while the old simulator allows bigger number to pass compile.

Example:

```
.byte -250
```

12.2.8. Immediate Addressing Mode accept illegal values

Immediate addressing mode allows the user to enter values that are out of the opcode range.
Example:

```
.text
movb $-130, r4
```

12.2.9. BBS, BBC, BBCC, BBCS, BBSS, BBSC Opcodes not always pass compile

The opcodes format, as specific by the manuals, of all of those opcodes is: "opcode pos.rl, base.vb, displ.bb", meaning the second operand can be either address or general register.
Yet, the old simulator accepts only addresses and doesn't allow 'base' to be register.

The following example results in compile-error:

```
.text
main: .word 0
      movb $0x80, r1
      bbs $8, r1, prn_true
      calls $0, prn_false
      jmp end_prog
eq2:  calls $0, prn_false

end_prog:
      pushl $0
      calls $1, .exit

prn_false: .word 0
      pushal lbl_false
      pushal format
      calls $2, .printf
      ret
```

```
prn_true:
      pushal lbl_true
      pushal format
      calls $2, .printf
      jmp end_prog

.data

lbl_true:   .asciz "True"
lbl_false:  .asciz "False"
format:     .asciz "%s\n"
```

12.2.10. ASCIZ generate garbage sometimes

The following code with send garbage to the screen:

```
.text

.word 0
pushal man
calls $1, .puts
halt
man: .asciz "#'
```

12.2.11. .SPACE Accept negative numbers

.SPACE accepts negative numbers, and generates millions of zeroes as response.

Example:

```
.text
.space -3
```

12.3. Simulator's bugs

12.3.1. CMPC5

CMPC5 len1, addr1, fillchar, len2, addr2

when len1 is equal to len2 the results in R0 and R2 are always zero, while it should be zero only if both strings are identical and of equal length.
(R1 and R3 act as described in the manual)

12.3.2. PRINTF

12.3.2.1. printf clears R1

The function .printf modifies not only R0, but R1 as well - it turned it to 0.

Example:

```
.text
main:      .word 0
           movl $10, r1
           pushal format
           calls $1, .printf
           pushl r1
           pushal format2
           calls $2, .printf
           pushl $0
           calls $1, .exit
.data
format:   .asciz "blah blah\n"
format2: .asciz "r1 = %d\n"
```

12.3.2.2. %X isn't working as expected

The function printf supports %x control char, but %X isn't working correctly and producing garbage.

Example:

```
.text
main: .word 0
movl  $0xF0F0F0F0, r5
pushl r5
pushal format
calls $2, .printf
pushl $0
calls $1, .exit

.data
format: .asciz "%X\n"
```

12.3.2.3. 11.3.2.3. %?d isn't working as expected

%0Xd should print a number with at least X digits. for example, when we use %02d it should print 01 instead of 1. Yet it actually cut the number and takes only 2 digits from it if it is bigger than the specific length.

Example:

```
      .text
main: .word 0x00

      pushl $123
      pushal format
      calls $2, .printf

      pushl $0
      calls $1, .exit

.data
format:     .asciz "\r%02d"
```

The output is 23 instead of 123 as expected.

12.3.3. ACBL

The instruction "ACBL" doesn't work in the old simulator

Example:

```
.text
main:       .word 0
            clrl  r0
loop:       pushl r0
            pushal format
            calls $2, .printf
            acbl $30, $5, r0, loop
            pushl $0
            calls $1, .exit
            .data
format:     .asciz "%d\n"
```

12.3.4. DECB, DECW, DECL

DEC commands: DECB, DECW, DECL sets the machine flags incorrectly.

The rule it need to obey: N and Z are set as always. V is set if 80..0 is decremented to 7F..F. C is set if 0 is decremented to FF..F.

Example:

```
.text
      .word 0
      movl $3, r0
      decl r0
      decl r0
      decl r0      # Z = 1
      decl r0      # On Old Sim: N = 1, C = 0 (Need to be: C = 1)

      movb $0x81, r0
      decb r0      # N = 1
      decb r0      # On Old Sim: C = 1, V = 1 (Need to be: C = 0)
      decb r0      # N = 0, C = 0, V = 0

      pushl $0
      calls $1, .exit
```

12.3.5. INCB, INCW, INCL

INC commands: INCB, INCW, INCL sets the machine flags incorrectly.
The rule it need to obey: N and Z are set as always. V is set if 7F..F is incremented to 80..0. C is set if FF..F is incremented to 0.

Example:

```
.text

main: .word 0

movb $0xFF, r0
incb r0             # On Old Sim: C = 0, V = 0 (Need to be: C = 1)

movw $0xFFFF, r0
incw r0             # On Old Sim: C = 0, V = 0 (Need to be: C = 1)

movl $0xFFFFFFFF, r0
incl r0             # On Old Sim: C = 0, V = 0 (Need to be: C = 1)

movl $0x7FFFFFFF, r0
incl r0             # On Old Sim: C = 0, V = 0 (Need to be: V = 1)

movw $0x7FFF, r0
incw r0             # r0 contains 0x8000, C = 0, V = 1, N = 1

movb $0x7F, r0
incb r0             # r0 contains 0x80, C = 0, V = 1, N = 1

halt
```

We can see here two different bugs:

1. INC opcodes doesn't raise C flag when needed.
2. INCL doesn't work well on edge case - when we give it the maximum possible negative number.

12.3.6. ADWC

ADWC sets flags incorrectly.

Example:

```
.text
.word 0
movb $0x7C, r0
addb3 $-1, $1, r0        # Z = 1, C = 1
adwc $0x7FFFFFFF, r1     # On Old Sim:
                         # N = 1, V = 1, C = 1 (Need To Be: C = 0)
halt
```

12.3.7. SUBB2, SUBW2, SUBL2

The opcodes set flags incorrectly.

Example:

```
.text

main: .word 0
movb $0x82, r0
subb2 $10, r0      # N = 0, V = 0 (Should be V = 1)

movw $0x8002, r0
subw2 $10, r0      # N = 0, V = 0 (Should be V = 1)

movw $0x8002, r0
subl2 $10, r0      # N = 0, V = 0

movl $0x80000002, r0
subl2 $10, r0      # N = 0, V = 0, C = 1 (Should be V = 1, C = 0)

movb $0x0, r0
subb2 $10, r0      # N = 1, C = 1

movw $0x0, r0
subw2 $10, r0      # N = 1, C = 1

movl $0x0, r0
subl2 $10, r0      # N = 1, C = 1

pushl $0
calls $1, .exit
```

12.3.8. SUBB3, SUBW3, SUBL3

The opcodes set flags incorrectly.

Example:

```
.text

main: .word 0
movb $0x82, r0
subb3 $10, r0, r0 # N = 0, V = 0 (Should be V = 1)

movw $0x8002, r0
subw3 $10, r0, r0 # N = 0, V = 0 (Should be V = 1)

movw $0x8002, r0
subl3 $10, r0, r0 # N = 0, V = 0

movl $0x80000002, r0
subl3 $10, r0, r0 # N = 0, V = 0, C = 1 (Should be V = 1, C = 0)

movb $0x0, r0
subb3 $10, r0, r0 # N = 1, C = 1

movw $0x0, r0
subw3 $10, r0, r0 # N = 1, C = 1

movl $0x0, r0
subl3 $10, r0, r0 # N = 1, C = 1

pushl $0
calls $1, .exit
```

12.3.9. SOBGEQ

The opcode set the overflow flag incorrectly.
Integer overflow occurs if the index operand before subtraction is the largest negative integer. Yet SOBGEQ opcode doesn't raise the flag.
The SOBGTR opcode, which do almost the same action as SOBGEQ do raise the flag.

Example:

```
.text
main: .word 0
      movl $0x80000000, r2

prnLoop:
      pushl r2
      pushal format
      calls $2, .printf
      sobgeq r2, prnLoop # overflow flag should rise on the first
                         # time we reach this line
      pushl $0
      calls $1, .exit

.data
format: .asciz "%ld\n"
```

12.3.10. EDIV

EDIV gets long operand instead of quad operand on the old simulator. The format of the opcode is "opcode divr.rl, divd.rq, quo.wl, rem.wl", but the old simulator takes also the second operand as long operand.

12.3.11. DIVB2, DIVW2, DIVB3, DIVW3

The divide opcodes doesn't work when the divider is negative number.

Example:

```
.text
main: .word 0
      movb $0x80, r0
      divb2 $-2, r0      # r0 should contains 40 (Old sim: r0=0)

      movw $0x8000, r1
      divw2 $-2, r1      # r1 should contains 4000 (Old sim: r1=0)

      movl $0x80000000, r2
      divl2 $-2, r2      # r2 contains 40000000

      movb $0x80, r3
      divb3 $-2, r3, r3 # r3 should contains 40 (Old sim: r3=0)

      movw $0x8000, r4
      divw3 $-2, r4, r4 # r4 should contains 4000 (Old sim: r4=0)

      movl $0x80000000, r5
      divl3 $-2, r5, r5 # r5 contains 40000000

      halt
```

12.3.12. GETCHAR, GETS

When the user presses Ctrl+Z (EOF), the functions stop immediately without waiting for the new line character.

12.3.13. SBWC

SBWC changes flags incorrectly. Example:

```
.text

main: .word 0

      movl $0x80000000, r1
      movl $1, r2
      sbwc r2, r1       # C = 1, V = 0 (Need to be V = 1, C = 0)

      movl $0x80000001, r1
      movl $1, r2
      bispsw $1
      sbwc r2, r1       # C = 1, V = 0 (Need to be V = 1, C = 0)

      pushl $0
      calls $1, .exit
```

12.3.14. free

Free function doesn't check if it really allocated the memory.

The function gets a block and marks it as free memory.

The old simulator didn't check if it was already free memory, or if it was allocated earlier by the simulator.

13. VAX11 User Manual – Hebrew

הקדמה

חבילת התוכנה מורכבת ממספר רכיבים המקושרים ביניהם: עורך טקסט, אסמבלר ודמיין. באמצעות רכיבים אלו אנו מסוגלים לכתוב קוד בשפת האסמבלר של VAX-11, להפוך אותו לקוד בשפת מכונה ולהריץ אותו. כמו כן, סביבת העבודה כוללת כלים להרצת התוכנית צעד אחד צעד ולניפוי שגיאות. הממשק עוצב בדומה לרוב תוכניות Windows הקיימות, על מנת לאפשר למשתמשים להתחיל לעבוד עימו במהירות ובנוחות.

תכונות עיקריות

עורך הטקסט

- כולל Syntax Highlight המותאם לכתיבת קוד עבור VAX-11.
- מאפשר עריכה של מספר קבצים בו זמנית.

אסמבלר

- אסמבלר מתוחכם המספק מידע מפורט במקרה של שגיאות.
- כולל אפשרות לאופטימיזציה של הקוד.
- מאפשר להציג קובץ פלט הכולל את שפת המכונה במקביל לקוד.

דמיין

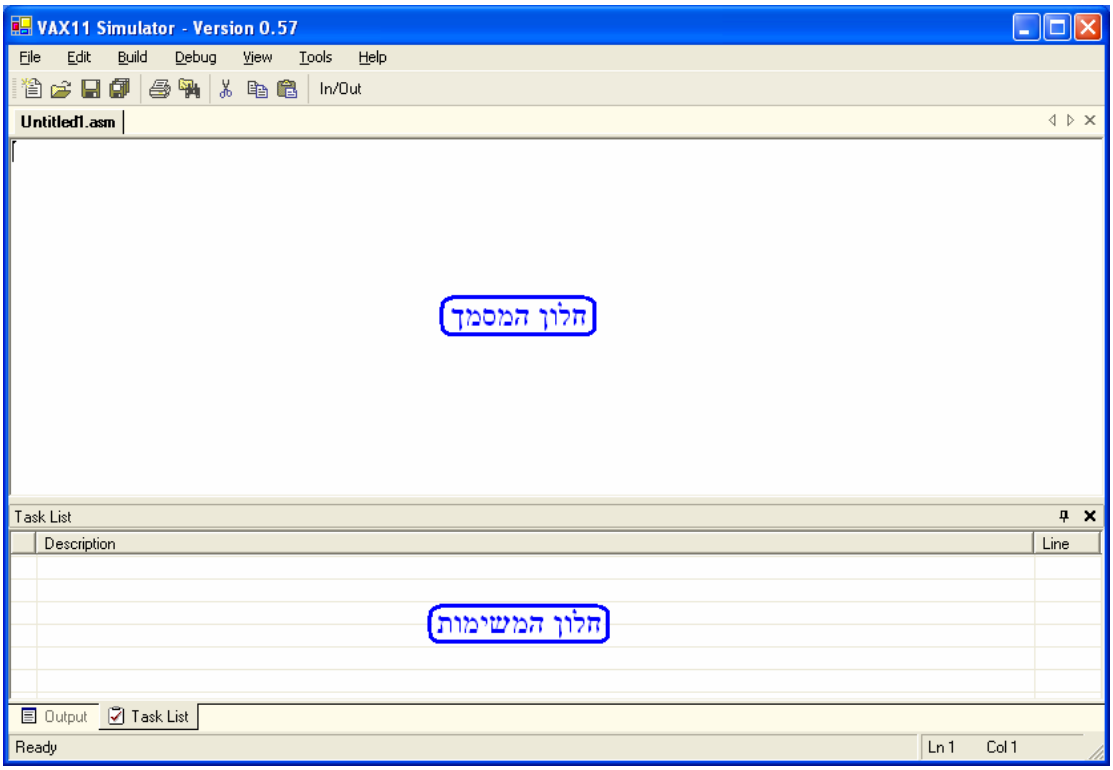
- דמיין התומך ברוב ה-Opcodes של מחשב ה-VAX11
- תומך בעשרות שגרות מערכת של VAX.
- תומך בפסיקות ובחריגות.
- תומך בזיכרון וירטואלי, מאפשר מרחב כתובות של 4GB בתים.
- מכיל הדמיה של זיכרון פיסי ו-Page Faults.
- מאפשר לעקוב אחרי זמן ביצוע התוכנית.

מנפה שגיאות:

- מאפשר הרצת התוכנית שורה אחר שורה, תוך כדי שהסביבה מסמנת את השורה הפעילה.
- תומך בהוספת נקודות עצירה לתוכנית.
- מאפשר לצפות במצב הרגיסטרים, המחסנית והזיכרון בכל רגע.

התחלת העבודה

כאשר נפתח את סביבת העבודה ייפתח החלון הראשי של התוכנית.



מסך הפתיחה של הסימולטור

חלון המסמך הוא המקום בו כותב המשתמש את הקוד של התוכנית. חלון זה מתפקד כעורך טקסט פשוט.

חלון המשימות משמש את האסמבלר. במידה וישנן שגיאות בקוד המשתמש תופיע רשימת השגיאות בחלון זה. כל שגיאה מופיעה בשורה נפרדת, כאשר לחיצה כפולה על שורת השגיאה מקפיצה את חלון המסמך אל השורה המתאימה בקוד.

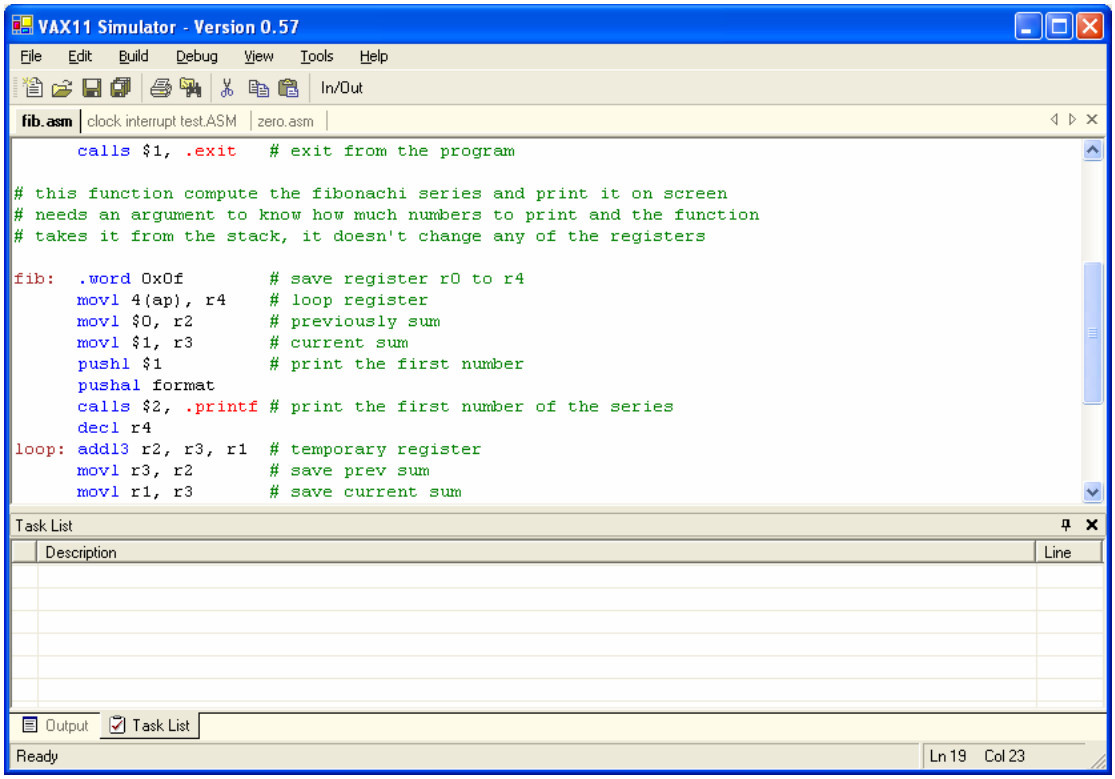
לצד חלון המשימות נמצא **חלון הפלט**, אליו האסמבלר והדמיין שולחים מידע בזמן פעולתם על מצב ההידור או מצב הריצה של התוכנית.

עורך הטקסט

סביבת העבודה מאפשרת למשתמש לערוך קבצי טקסט, בהם ייכתבו תוכניות VAX.

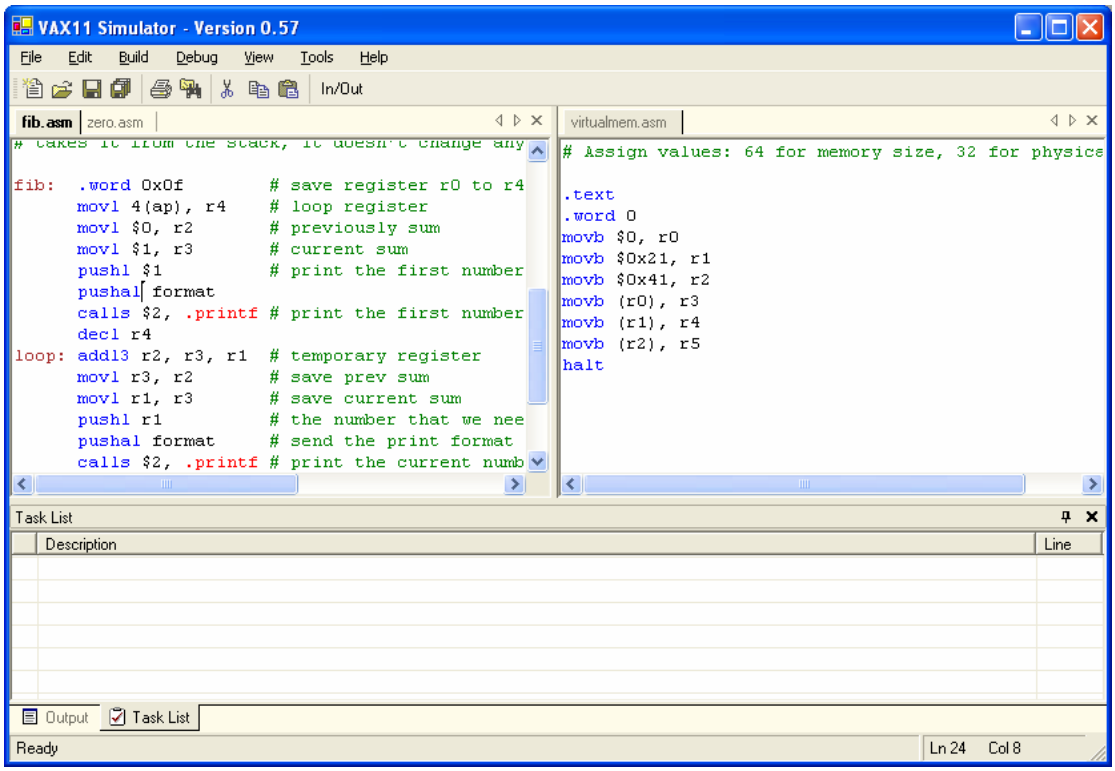
כל **מסמך** (תוכנית) של המשתמש נמצא בחלון מסמך משלו, כאשר ניתן לעבור בין המסמכים השונים הפעילים.

בזמן כתיבת המסמך סביבת העבודה צובעת באופן אוטומטי פקודות הקשורות אל ה-VAX לנוחות המתכנת. הצבעים ניתנים להתאמה אישית מלאה דרך תפריטי סביבת העבודה.



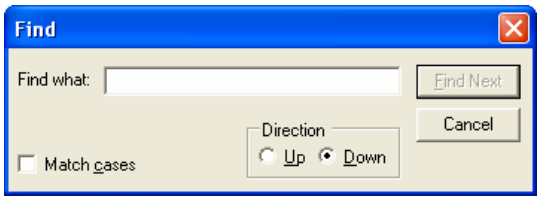
שלושה מסמכים פתוחים

סביבת העבודה מאפשרת גם לפצל את המסמכים הפעילים, כך שכל אחד מהם יתפוס חלק משטח העבודה. פיצול המסמכים נעשה על ידי לחיצה על שם המסמך בסימניית המסמכים, וגרירתו הצידה.



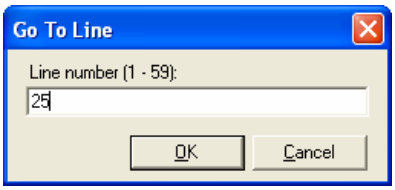
פיצול מסמכים לחלונות מקבילים

עורך הטקסט מאפשר לבצע חיפוש בטקסט. לחיצה על האפשרות Find בתפריט Edit או לחיצה על המקשים Ctrl+F תפתח את חלון החיפוש. ניתן להקליד טקסט ולמצוא את כל המופעים של אותו הטקסט במסמך הפעיל.

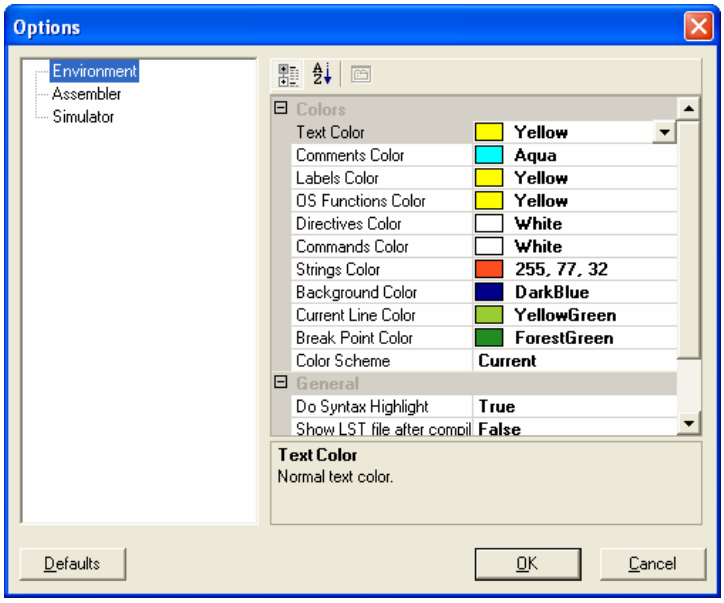


חלון החיפוש

ניתן גם לקפוץ לשורה ספציפית בקובץ, וזאת על ידי בחירת התפריט Go To שנמצא תחת Edit, או על ידי לחיצה על צירוף המקשים Ctrl+G.



חלון האפשרויות



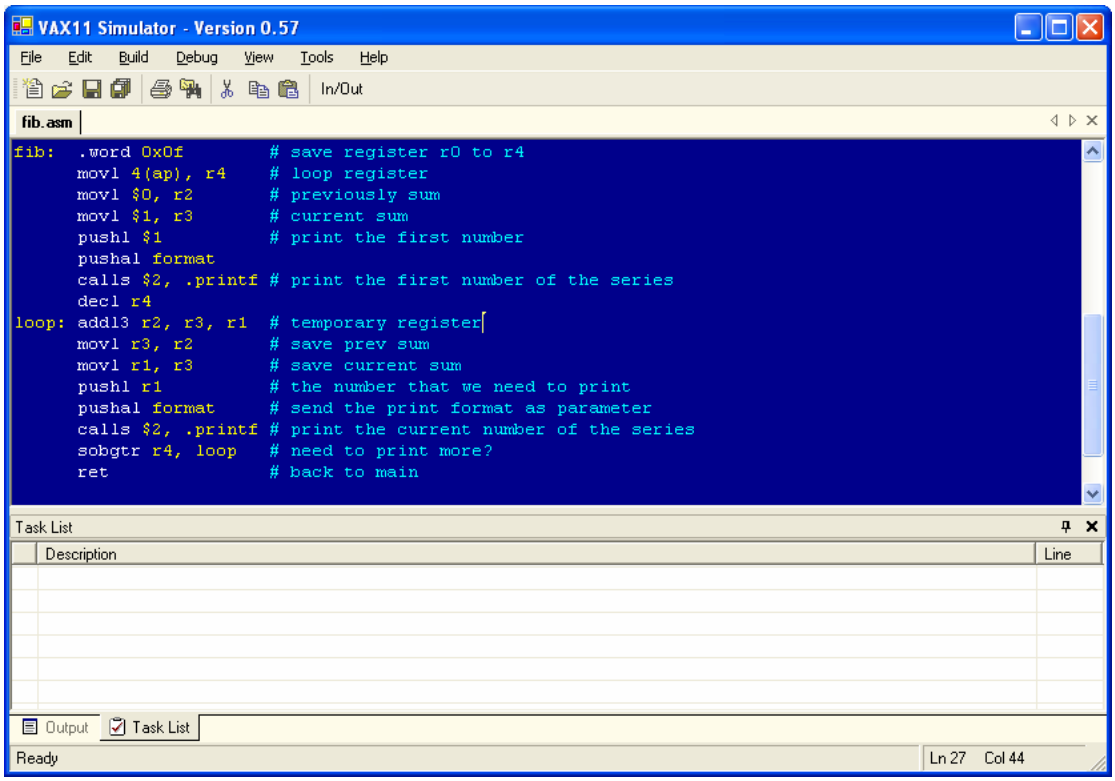
חלון האפשרויות – אפשרויות סביבת העבודה

אספקטים רבים בסביבת העבודה, באסמבלר ובסימולטור ניתנים להתאמה אישית עבור המשתמש.

חלון האפשרויות מאפשר לקבוע את ההגדרות השונות עבור סביבת העבודה.

סביבת העבודה

צבעים	
פרמטר	**משמעות**
Text Color	הצבע של כל טקסט שאינו בעל משמעות מיוחדת עבור VAX
Comments Color	צבע ההערות בקוד
Labels Color	צבע התגיות
OS Functions Color	צבע קריאות מערכת
Directives Color	צבע ההנחיות למהדר
Commands Color	צבע ה-opcodes של ה-VAX
Strings Color	צבע קבועי מחרוזות
Background Color	צבע הרקע של המסמך
Current Line Color	צבע השורה הנוכחית (רלוונטי בזמן ניפוי שגיאות)
Breakpoint Color	צבע שורות בהן יש נקודות עצירה
Color Scheme	בחירה של ערכת צבעים מבין הערכות הבאות יחד עם הסימולטור.
כללי	
פרמטר	**משמעות**
Do Syntax Highlight	קובע האם הטקסט ייצבע או לא
Show LST file after compile	במידה ואפשרות זו נבחרת, לאחר כל הידור מוצלח יוצג קובץ LST המציג את תוצאת ההידור
Show agent on startup	קובע האם מרלין הקוסם יברך את המשתמש עם הפעלת הסימולטור



סביבת העבודה עם צבעים מותאמים אישית

אסמבלר

כללי	
פרמטר	**משמעות**
Optimize Code	במידה ואפשרות זו נבחרת, האסמבלר ייצור את הקוד הקטן ביותר האפשרי עבור קוד המשתמש. על מנת להשוות את האסמבלר הנוכחי לאסמבלר הישן יש לבטל אפשרות זאת, מכיוון שהאסמבלר הישן לא תמך באופטימיזציה של הקוד.
Save LST file after compile	אם אפשרות זו נבחרת, האסמבלר ייצר קובץ פלט עם תוצאת ההידור לאחר כל הידור מוצלח.

סימולטור

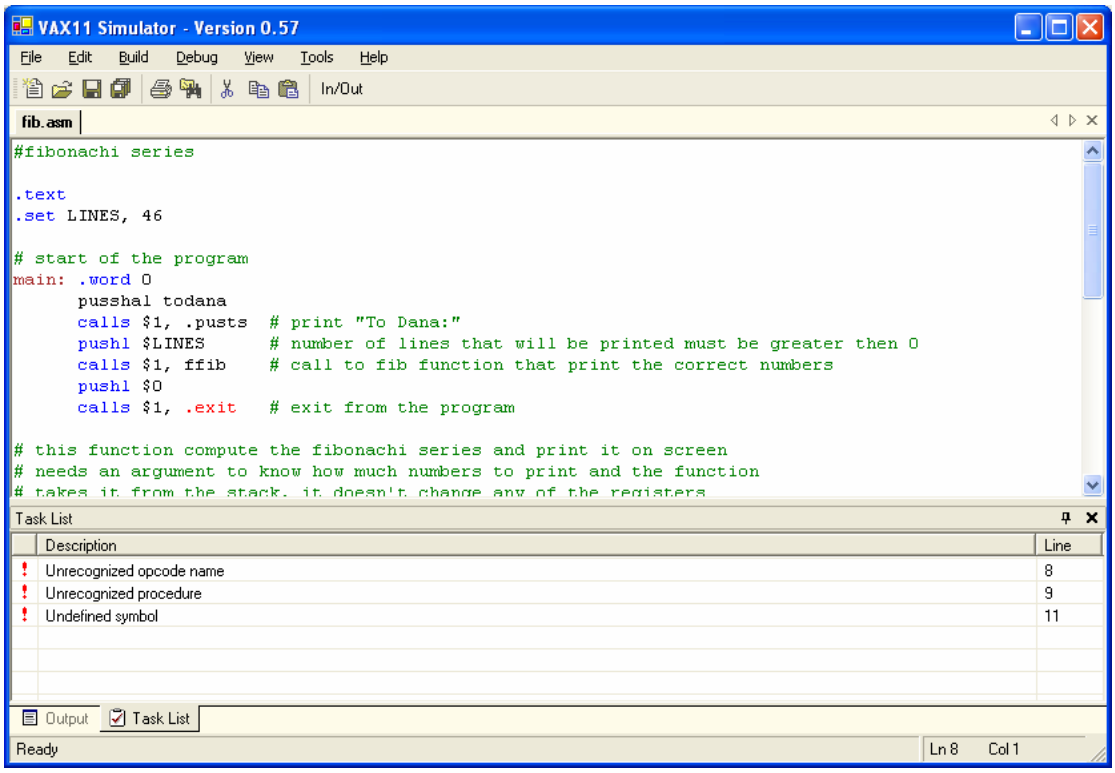
הסימולטור מכיל הדמיה של זיכרון פיסי – המחולק לדפים והתומך בזיכרון וירטואלי. ההגדרות הנוגעות לזיכרון מתייחסות אל ההדמיה הקיימת בסימולטור. ה-Console הוא חלון הקלט/פלט של הסימולטור.

Console	
פרמטר	**משמעות**
Text Color	צבע הטקסט של חלון ה-Console
Background Color	צבע הרקע של חלון ה-Console
Always on top on debug mode	קובע הם חלון ה-Console יהיה Always On Top, אם אנחנו במצב ניפוי שגיאות
כללי	
פרמטר	**משמעות**
Show Registers in Hex	במידה ונבחר, ערכי הרגיסטרים במצב ניפוי יופיעו כמספרים הקסדצימליים
Show Special Registers	במידה ונבחר, רגיסטרים מיוחדים הנגישים בדרך כלל במצב גרעין (kernel) בלבד יוצגו במצב ניפוי
Show Debug Information	במידה ונבחר, הסימולטור ייצר פלט מפורט על מצב הסימולטור במידה ומתרחשת שגיאה הגורמת לסיום התוכנית
זיכרון	
פרמטר	**משמעות**
Page Size	גודל דף בזיכרון של הסימולטור
Physical Memory Size	גודל הזיכרון הפיסי
Show Memory Accesses	במידה ואפשרות זו נבחרת, יוצגו למשתמש כל הגישות לזיכרון שמבצעת התוכנית
Show Physical Addresses	קובע האם להציג את הכתובת הפיסית במקביל לכתובת הוירטואלית כאשר אנו מציגים את הגישות לזיכרון
Show Page Faults	קובע האם להציג הודעה כאשר מתרחש Page-Fault
Fill Memory With Garbage	במידה ונבחר, הזיכרון יתמלא באופן רנדומלי בזבל, ולא באפסים.

האסמבלר

לאחר שנכתוב קוד נוכל להדר אותו דרך התפריט Build ובחירת האפשרות Compile.

במידה והיו שגיאות בקוד, תופיע רשימת השגיאות בחלון המשימות ותינתן לנו אפשרות לתקנן.



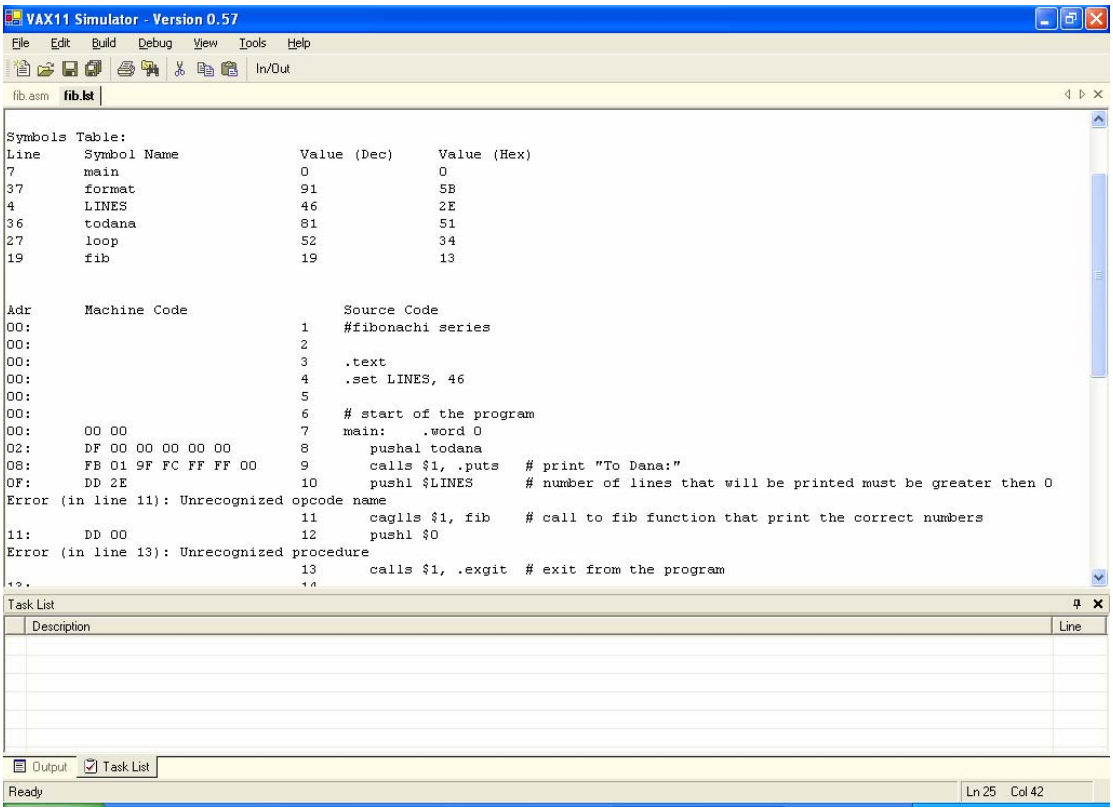
קוד עם שגיאות – השגיאות מופיעות בחלון המשימות

קבצי LST הם קבצים המציגים את קוד המשתמש במקביל לקוד המכונה אותו ייצר האסמבלר.

סביבת העבודה מאפשרת למשתמש להביט ולשמור את קבצי ה-LST.

על מנת לצפות בקובץ ה-LST, נלך לתפריט Build ושם נבחר View LST File.

יש לציין כי ניתן לצפות בקובץ ה-LST גם במידה והיו שגיאות בזמן ההידור, ובמקרה זה יופיעו השגיאות ב-LST לצד הקוד.

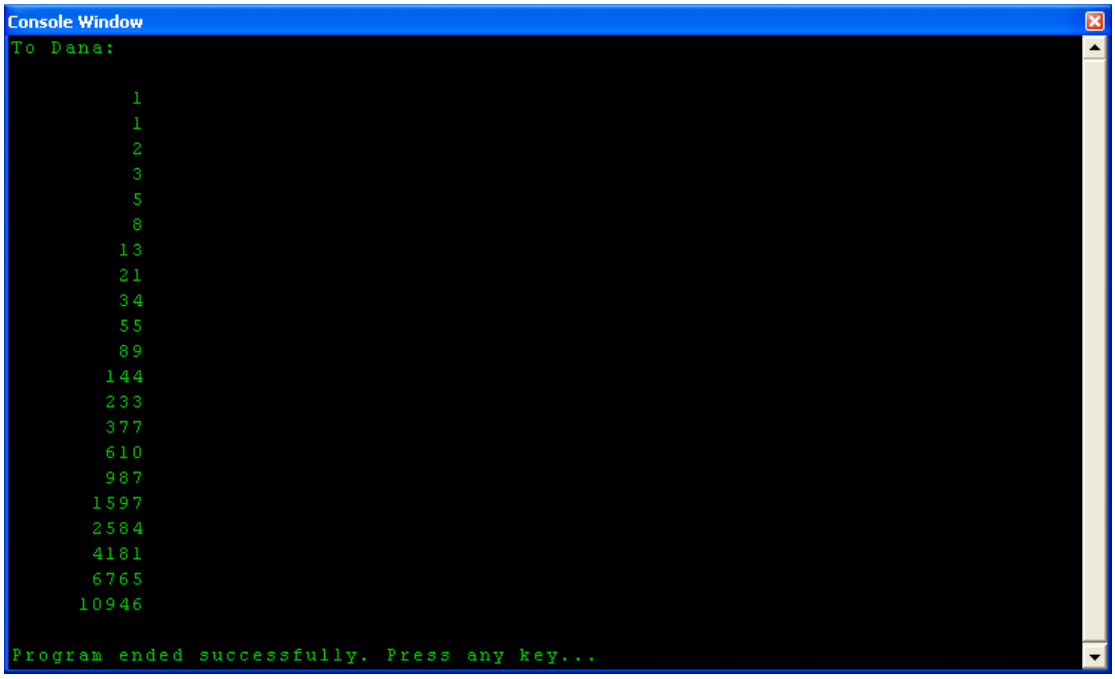

VAX11 Simulator - Version 0.57
File Edit Build Debug View Tools Help
In/Out
fib.asm fib.lst
Symbols Table:
Line Symbol Name Value (Dec) Value (Hex)
7 main 0 0
37 format 91 5B
4 LINES 46 2E
36 todana 81 51
27 loop 52 34
19 fib 19 13
Adr Machine Code Source Code
00: 1 #fibonachi series
00: 2
00: 3 .text
00: 4 .set LINES, 46
00: 5
00: 6 # start of the program
00: 00 00 7 main: .word 0
02: DF 00 00 00 00 00 8 pushal todana
08: FB 01 9F FC FF FF 00 9 calls $1, .puts # print "To Dana:"
0F: DD 2E 10 pushl $LINES # number of lines that will be printed must be greater then 0
Error (in line 11): Unrecognized opcode name
11 caglls $1, fib # call to fib function that print the correct numbers
11: DD 00 12 pushl $0
Error (in line 13): Unrecognized procedure
13 calls $1, .exgit # exit from the program
Task List
Description
Line
Output
Task List
Ready
Ln 25 Col 42

LST קובץ

הדמיין

הדמיין מאפשר הרצת תוכניות VAX. הדמיין מחקה אספקטים רבים של החומרה, כולל זיכרון, רגיסטרים, פסיקות, חריגות ועוד.

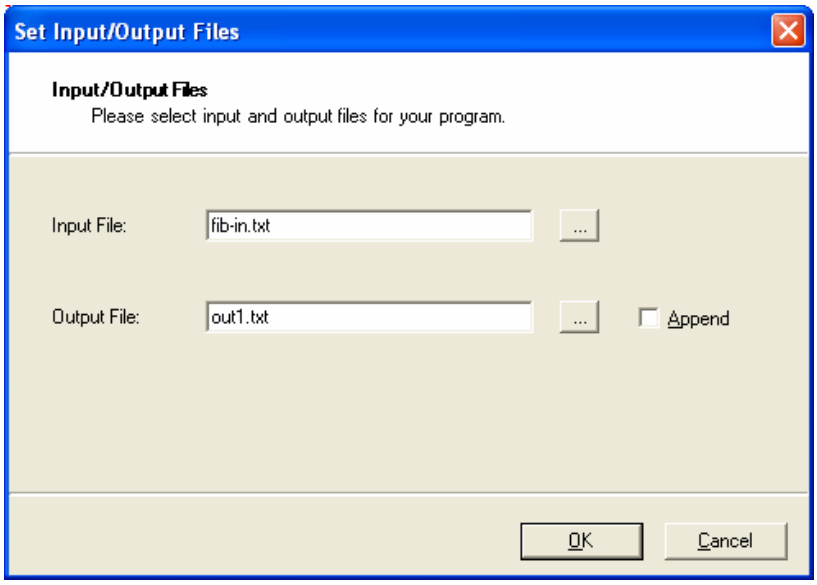
על מנת להריץ תוכנית שכתבנו, נלך לתפריט Build ונבחר באפשרות Execute. ייפתח חלון Console בו יוצג הפלט של התוכנית שלנו, ובו נוכל להכניס קלט במידה והתוכנית שכתבנו דורשת זאת.



חלון Console עם תוכנית המציגה מספרי פיבונצ'י

הדמיין מאפשר לנו לקבוע קבצי קלט ופלט, על מנת שנוכל לקרוא נתונים מהם במקום מהמקלדת ואל המסך.

על מנת להגדיר קבצי קלט/פלט עבור תוכנית מסוימת, יש ללחוץ על התפריט Build, ושם על Set Input/Output Files ולבחור את קבצי קלט הפלט הרצויים.
יש לשים לב שהגדרת קבצי הקלט-פלט היא אישית עבור כל אחד מהמסמכים הפתוחים.
האפשרות Append מאפשרת להוסיף את הפלט בסוף קובץ קיים.

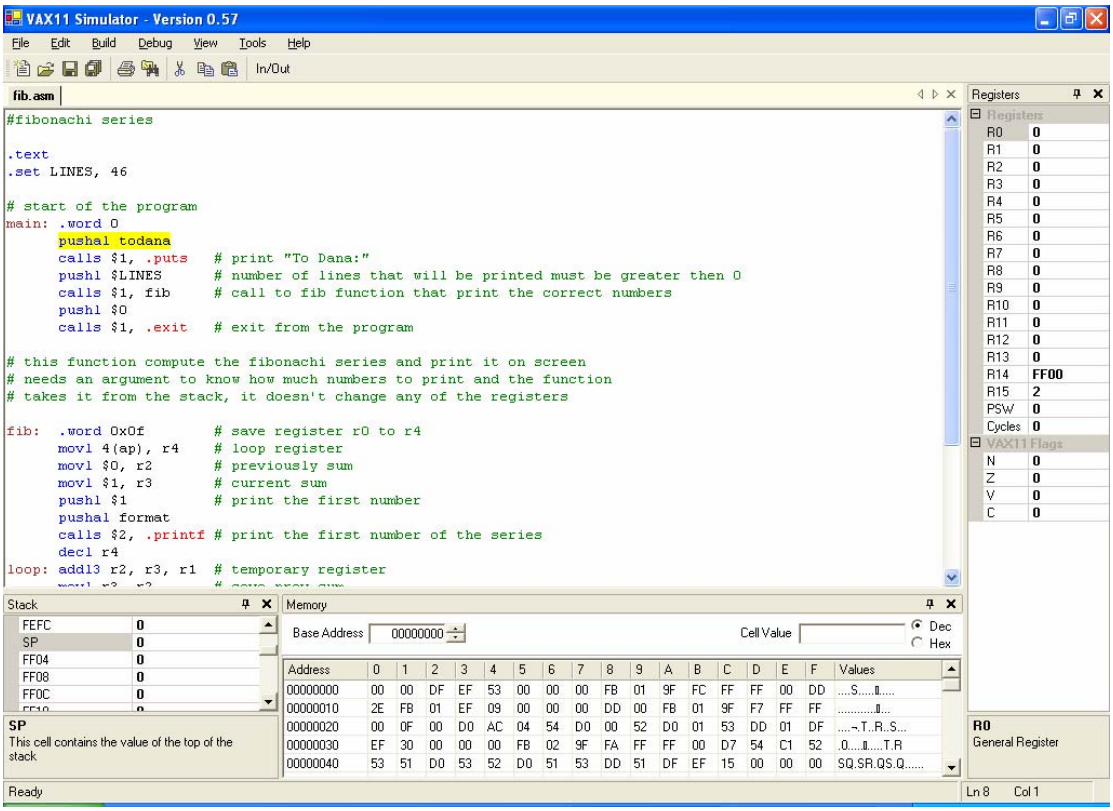


חלון בחירת קבצי קלט/פלט

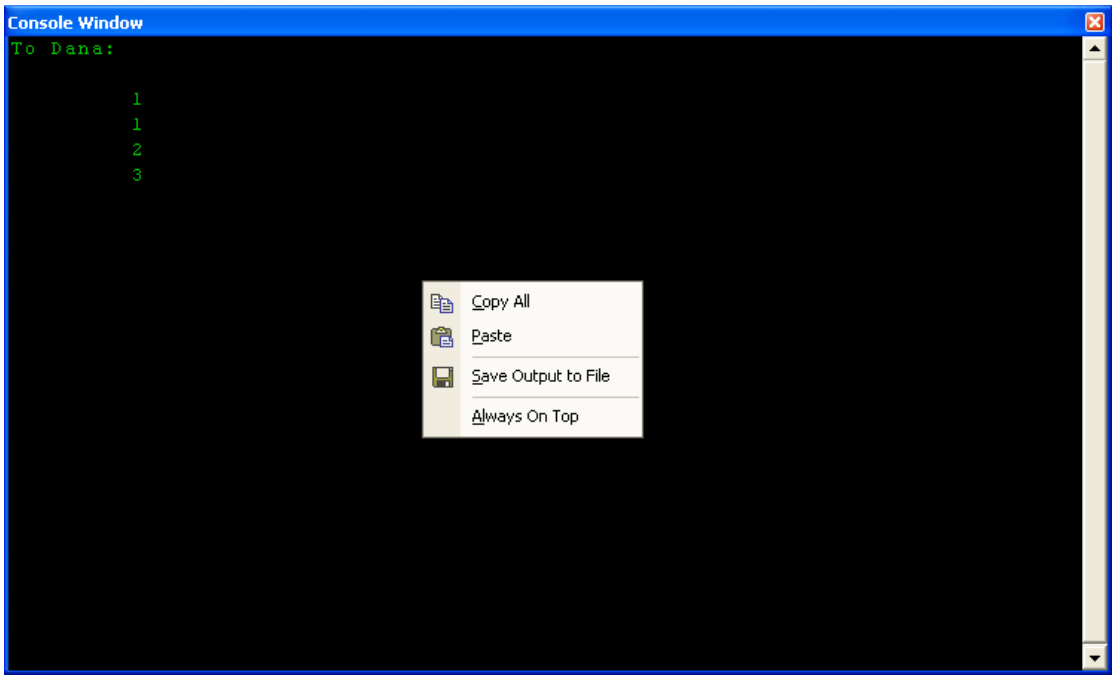
תפריט ה-Debug מאפשר הפעלת התוכנית במצב ריצה צעד אחד צעד, על מנת לתקן שגיאות בתוכנית.

הפעלת התוכנית במצב ריצה צעד אחר צעד נעשה על ידי האפשרות Step בתפריט Debug.

ייפתחו בסביבת העבודה חלונות נוספים, המציגים את מצב המערכת: רגיסטרים, זיכרון ומחסנית. כמו כן, השורה הבאה לביצוע תודגש כל רגע בצבע.



בנוסף ייפתח חלון ה-Console בו יופיע פלט התוכנית בכל רגע ורגע.



אפשרות שימושית נוספת הקיימת בחלון ה-Console היא לחיצה על המקש ימני של העכבר בנקודה כלשהי ב-Console. לחיצה על המקש הימני תפתח תפריט בו מספר אפשרויות: נוכל לקבוע שהחלון יהיה מעל כל החלונות האחרים, נוכל להעתיק את פלט התוכנית שלנו, ונוכל להדביק מידע שישמש כקלט לתוכנית.

התוכנית הראשונה

נציג תוכנית פשוטה המדפיסה על המסך את המחרוזת "Hello, World", ונתחה את התוכנית.

```
.text

.org 0x1000
main: .word 0

      # Print a message
      pushal strHello
      calls $1, .puts

      # Exit Program
      pushl $0
      calls $1, .exit

.data

strHello:   .asciz "Hello, World\n"
```

כל תוכנית חייבת להתחיל בהנחיה ".text" המציינת את תחילת קוד התוכנית.

ההנחיה ".org" אומרת לסימולטור למקם את הקוד החל מכתובת המצוינת כפרמטר. במקרה שלנו הקוד ימוקם החל מהכתובת 0x1000. אין כל הגבלה למקם את התוכנית בכל מקום בזיכרון, אם כי לא מקובל למקם את התוכנית בכתובות הנמוכות, עקב העובדה שבמערכת אמיתית מערכת ההפעלה היא הנמצאת בכתובות הזיכרון הנמוכות.

התגית main היא נקודת ההתחלה של התוכנית. הסימולטור מחפש תמיד את תגית זו, ובמידה ונמצאה היא משמשת כנקודת ההתחלה של התוכנית. במידה ונרצה להתחיל את התוכנית בנקודה אחרת ולא ב-main נשתמש בהנחיה ".entrypoint".

לאחר התגית main מופיעה מילת המסכה, ולאחריה קוד התוכנית.

כל תוכנית המסתיימת בהצלחה מסתיימת בשורות:

```
      pushl $0
      calls $1, .exit
```

שורות אלו מבצעות קריאה לשגרת מערכת שתפקידה לסיים את התוכנית.

התגית ".data" מציינת את התחלת אזור הנתונים של התוכנית.

14. Bibliography

14.1. Books

1. Digital Equipment Corporation (1979), **VAX Architecture Handbook**, USA.
2. Digital Equipment Corporation (1980), **VAX Hardware Handbook**, USA.
3. Digital Equipment Corporation (1980), **VAX Software Handbook**, USA.
4. Digital Equipment Corporation (1986), **Supermicrosystems Handbook**, USA.
5. Gerber, I., (2002), **C# and .NET Technology**, Israel.
6. J. Jair and Lev I., (1996), **Exercises Book - Software System Course**, The Technion, Israel.
7. Levy, H.M., Eckhouse, R.H. (1980), **Computer programming and architecture: The VAX-11**, Digital Press, USA.

14.2. Summaries

8. Adar N., (2003), **"C# for C++ programmers"**, http://underwar.livedns.co.il/.
9. Ahuja S., (lparam@hotmail.com) (2002), **"XML Parser"**.
10. Allowatt T., (2002), **"Bending the .NET PropertyGrid to Your Will"**.

11. A.M. Kuchling (2000), **"Regular Expression"**.

12. Apple Computer, Inc (1995), Introduction to RISC Technology, **"Two representative CISC designs"**

13. Bornstein N. (2002), **"Learning C# XML"**.

14. Brejeon A. (2001), **"Read and Write application parameters in XML"**.

15. Ceddia D. (2003), **"Drag and Drop Files with C#"**.

16. Char J. (2003), **"HTML Help Workshop Tutorial"**.

17. Crowley J., (2003), **"Colour HTML Tags"**.

18. Dario F. Gomes (2001), **"Learning to Use Regular Expressions"**, http://ibm.com/developer/.

19. Deitel P., Deitel H., Listfield J., Nieto T., Yaeger C., Zlatkina M. (2003), **"Multithreading in C#"**.

20. Dimri A. (2002), **"Threading in C#"**.

21. Ewing G. (2002), **"Using Threads"**, Clarity Consulting Inc.

22. Farber A. (2001), **"Worker Threads in C#"**.

23. Gold M., (2003), **"Simple Color Syntax Code Editor for PHP"**, (mtechsupport@microgold.com).

24. Hinrichs R. (2003), **"Managed MessageBeep() in C#"**

25. J. T. Johnson (2002), **".NET XML Serialization"**.

26. Krikorian R. (2001), **"Multithreading with C#"**.

27. Kurniawan, B., (2002), **"C# Key Processing Techniques"**, Published on The O'Reilly Network (http://www.oreillynet.com).

28. Kurniawan B. (2002), **"C# Object Serialization"**, Published on The O'Reilly Network (http://www.oreillynet.com).

29. Merrill B., (2001), **"C# Regular Expressions"**, http://www.oreilly.com/.

30. Microsoft Corp. (2003), **"Manipulating Controls from Threads"**

31. MSDN Communities (2001), **"Printing from a RichTextBox"**.

14.3. Open Sources

32. Magic - The User Interface Library for .Net - Manual Pages (2003), http://www.dotnetmagic.com.