VAX11 simulator

By Nir Adar and Rotem Grosman

Nir Adar Rotem Grosman underwar@hotmail.com rotemrpg@yahoo.com

2004

Welcome

VAX11 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.

The VAX11 Simulator makes it possible compile VAX-11 assembly programs and to test these programs.

The simulator comes with many features to make your work effective and productive:

Familiar interface:

- The environment looks like Visual Studio .Net
- · Advanced editor including syntax highlight, working on multiple documents
- Debug tools
- Significant documentation

Simulate many hardware aspects:

- Registers
- Virtual memory
- · Interrupts
- Clock cycles
- Machine language

And many other features...

The simulator was written by Software Laboratory - Technion, Faculty of Electrical Engineering.

Get Started

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.

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.

3. Start Working

When opening the working environment, the main application window will



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. 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.

W VAX11 Simulator - Version 1.00		
; Elle Edit Build Debug View Iools Help		
10660016944×60010 + • 9 81	In/Out	
fib.asm comisso (1)	× virtualment asm	11 × 11
#fibonachi series	# Assign values: 64 for memory size, 32 for	physical
<pre>.text .set LINES, 46 # start of the program main: .word 0 pushal todana calls \$1, .puts # print "To Dana:" pushi \$LINES # number of lines that s calls \$1, fib # call to fib function t pushi \$0 calls \$1, .exit # exit from the program # this function compute the fibonachi series and # needs an argument to know how much numbers to # takes it from the stark it down't charge art</pre>	.text .word 0 movb \$0, r0 movb \$0x21, r1 movb \$0x41, r2 movb (r0), r3 movb (r1), r4 movb (r2), r5 halt	
5	le.	3
Task List		φ×
Dumot P Task List		Line
Ready	Ln 10 Col	23

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.

io To Line		E
Line number	(1 - 59):	
25		
	ΟΚ	Cancel

Go To Line Window

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.

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
Labers Coror	on start of lines.
	The simulator supports some high-level functions,
OS Functions Color	as printf, getchar, etc.
OS Functions Color	These functions are known as operation system
	functions.
	Directives are commands meant for the assembler,

Directives Color	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	Select if the environment should highlight special
Hightlight	VAX11 words.
Show LST file after	If set, the LST file created during the compilation
compile	will be displayed after compile ends successfully.
Show agent on	The agent, Merlin, is welcome the users of
startup	VAX11 Simulator every time the program runs.



The working environment with custom colors settings

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.
	Select if the assembler should save LST file after
Save LST file after	successful compilation. LST file is text file
compile	containing the machine code and the source code
	of the compiled program.

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.
	If selected, the console window of debugged

Always on top on debug mode	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	
	If true, uninitalize memory cells will contain

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.

P VAX	11 Simulator - Version 1.00			6	6 🛛
E Be	Edit Build Debug Vers Look	Help			
1 in a	BOBANIS	6 E > # 93 A	I In/Out		
	fib.lat				11.0
Svishol	a Table:				~
Line	Symbol Name	Value (Dec)	Value (Be	x) Type	-
7	main	0	0	LABEL	
37	format	68	58	LADEL	
4	LINE:S	46	28	CONSTANT	
36	todana	78	4E	LABEL	
27	1000	49	31	LABEL	
19	fib	20	14	LASEL	
Adr	Machine Code	Source C	ode		
100		1 #fibonec	hi series		
:00		2			
:00		3 .text			
100		4 .set LIN	ES, 46		
:00		5			
100:		6 # start	of the progr	121	
100	00 00	7 maint	.word 0		
Error	(in line 0): Unrecognized	opcode name			
		8 pussh	al todana		
Error	(in line 9): Unrecognized	procedure			
	Mar 1969 states and states and	9 calls	\$1, .pputs	# print "To Dana:"	
02:	DD 2E	10 pushl	\$LINES.	# number of lines that will be printed must be greater then (0
Error	(in line 11); Undefined s	ymbol			
04:	FB 01 00 00 00 00 00	11 calls	\$1, ffib	# call to fib function that print the correct numbers	
OB:	DD 00	12 pushi	\$0		
OD:	FB 01 97 97 FF FF 00	13 calls	\$1, .exit	# exit from the program	10
Tack List	Sec				4 X
Desc	cription				Line
	25				
	🖃 🕄 Task List				
Build tale	4			1013 6454	-

LST File

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	
To Dana:	· · · · · · · · · · · · · · · · · · ·
1	
1	
2	
9	
5	
81	
13:	
21	
34	
5.5	
89	
1.4.4	
233	
377	
610	
987	
1597	
2584	
4181	
6765	
109.46	
Program end	ed successfully. Press any key

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.

Input File:	Rb-in. bd	
	Free	 _
Output File:	jout1.bt	 I Append

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.

The Basic structure of VAX-11 Computer



This section will describe the different parts of the computer.

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.

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.

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.

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. Bit 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.

31	30	29	21	20	16	15		8	7	6	5	4	3	2	1	0
СМ	TP	Othe	s	I	PL		Unused		DV	FU	IV	Т	Ν	Z	V	С

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.

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*.

*r*0...*r*11 are general purpose registers, while *r*12...*r*15 are special control registers.

Register	Alternative Name	Description
r12	АР	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	РС	Program Counter - contains the address of the next byte to be processed in the instruction stream.

Input/Output Devices

The simulator supports serial input / output devices.

Input: The keyboard (KBD) or input file.

Output: The monitor (CRT) or output file.

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.

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.

Addressing Modes

	Addressing Mode	Assembly Code	Machine Implementation		
1	Register	rNUMBER	50+Number		
2	Register Deferred	(rNUMBER)	60+Number		
3	Autoincrement	(rNUMBER)+	80+Number		
4	Autodecrement	-(rNUMBER)	70+Number		
5	Autoincrement Deferred	*(rNUMBER)+	90+Number		
6a		OFFSET(rNUMBER)	A0+Number	Byte Offset	
6b	Displacement	or	C0+Number	Word Offset	
6c		OFFSET[rNUMBER]	E0+Number	Long Offset	
7a			B0+Number	Byte Offset	
7b	Displacement Deferred	*OFFSET(rNUMBER)	D0+Number	Word Offset	
7c			F0+Number	Long Offset	
8a		(rBASE)[rINDEX]			
8b		(rBASE)+[rINDEX]		Depending on	
8c	Indov	-(rBASE)[rINDEX]	10+Indox	the BASE	
8d	muex	*(rBASE)+[rINDEX]	40+IIIuex	addressing	
8e		OFFSET(rBASE)[rINDEX]		mode.	
8f		*OFFSET(rBASE)[rINDEX]			
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	rNUMBER	r
2	Register Deferred	(rNUMBER)	*r
3	Autoincrement	(rNUMBER)+	*(r++)
4	Autodecrement	-(rNUMBER)	*(r)
5	Autoincrement Deferred	*(rNUMBER)+	**(r++)
6	Displacement	OFFSET(rNUMBER)	*((char*)r+OFFSET)
7	Displacement Deferred	*OFFSET(rNUMBER)	**((char*)r+OFFSET)
			*(rBASE+rINDEX)

8	Index	(rBASE)[rINDEX] (rBASE)+[rINDEX] -(rBASE)[rINDEX] *(rBASE)+[rINDEX] OFFSET(rBASE)[rINDEX] *OFFSET(rBASE)[rINDEX]	*((rBASE++)+rINDEX) *((rBASE)+rINDEX) **(rBASE+rINDEX) *((OFFSET+ (char*)rBASE)+rINDEX) *(*(OFFSET+ (char*)rBASE)+rINDEX)
9	Literal	\$VALUE (<64)	VALUE

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>

	0		ßsp(r14),
F E D C B A 9 8 7 6 5 4 3 2 1 0	0		fp(r13)
Flags: N, Z, V, C, T, More	Misc. PSW		
SPA S 0 Register Mask for r0-r11	ap(r12)		
	r0 r1 		
for <i>calls</i> .			
	r11		
ns:	Fill (0-3		
Iller Duches parameters	bytes,		ßap(r12)
inci i usites parameters.	to SPA)		
sh parameters number (N). Max: 255		Ν	
e 2 lsb bits of SP goes to SPA. Also tempßsp	arg1		Parameters List
sing SPA, we decide to fill 0-3 bytes, to align	arg2		Liot
stack.	 argN		
shing r0-r11 according to the mask.			
shing pc, fp, ap (Returning Address).			
esetting PSW Flags.			
sh PSW+Misc.			
sh 0 to mark the frame's end.			
'ßSP, APßtemp			
ßFunction 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

Stack structure after CALLG:

0	0	0	Ν
	ລາ	g1	
arg2			
		•••	
argN			

		()
DS	F E D C B A 9 8 7 6 5 4 3 2 1 0 W Elegge N Z V C T More 1 0	Misc.	PSW
M	isc. SPA S 0 Register Mask for r0-r11	ap(ı	:12)
		fp(r	13)
S i	s 0 for <i>calla.</i>	pc(ı	:15)
01	o o for carry.	r0	
Δ		r	1
AC	tions:	•	·•
0.	Preparing parameters list on the global	r1	.1
]	memory.	Fill (0- bytes	.3
1.	Including parameters number (N). Max: 255	accordi	ng
2.	2. The 2 lsb bits of SP goes to SPA.		
3.	3. Using SPA, we decide to fill 0-3 bytes, to align		
1	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.	FPßSP, APßglobal section address.		
10.	PCßFunction 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. spßfp+4
- 2. tempßMisc. + 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...

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 + 1]'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

NOTATION

is given the value
= 1, if is TRUE;
= 0, if is FALSE
is pushed onto the stack
is popped off the stack
addition
subtraction, or unary minus
multiplication
division
exponentiation
contents of
the value of , sign extended to a longer
structure
the value of , zero extended to a longer
structure
the bit field of a structure, consisting of bits n
1, m - 2,, n + 1, n
bit n of the structure x
bits m:n of the structure x
the address a indexed by the value b
the maximum of the numbers x and y
the minimum of the numbers x and y
equal to, signed or unsigned
greater than or equal to, signed
greater than or equal to, unsigned
greater than, signed
greater than, unsigned
less than or equal to, signed
less than or equal to, unsigned
less than, signed
less than, unsigned
not equal to, signed or unsigned

ACB ADD COMPARE AND BRANCH

Purpose	maintain loop count and loop		
Format	opcode limit.rx, add.rx, index.mx, displ.bw		
Operation	index ß index + add; if {{add GEQ 0} AND {index LEQ limit}} OR {{add LSS 0} AND {index GEQ limit}} then		
Condition codes	<pre>PC & PC + SEXT (displ); N ß index LSS 0; Z ß index EQL 0; V ß {integer or floating overflow}; C ß C;</pre>		
Exceptions	integer overflow floating overflow floating underflow reserved operand		
Opcodes	9D 3D Fl 4F Float 6F	ACBB ACBW ACBL ACBF ing ACBD	Add Compare and Branch Byte Add Compare and Branch Word Add Compare and Branch Long Add Compare and Branch 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 ß 1; Z ß 0 V ß 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.

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 "

ADAWI ADD ALIGNED WORD INTERLOCKED

Purpose	maintain operating system resource usage counts		
Format	opcode add.rw, sum.mw		
	tmp ß add;		
	{set interlock};		
Operation	sum ß sum + tmp;		
	{released interlock};		
	N ß sum LSS 0;		
	Z ß sum EQL 0;		
Condition codes	Vß {integer overflow};		
	C ß {carry from most significant bit};		
	reserved operand fault		
Exceptions	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	 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. If the addend and the sum operand overlap, the result and the condition codes are UNPREDICTABLE. 		

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

.text main: .word 0 \$4, r0 movw \$5, r1 movw adawi r0, r1 r1 pushl format pushal \$2, .printf calls pushl \$0 \$1, .exit calls .data **format:** .asciz "R1 is %d\n"

ADD ADD

Purpose	perform arithmetic addition			
Tarma t	opcode add.rx, sum.mx2 operandopcode addl.rx, add2.rx, sum.wx3 operand			
Format				3 operand
	sum ß	Ssum + add;		2 operand
Operation	sum ß add1 + add2; 3 operand			3 operand
	N ß sum LSS 0;			
	Z ß sum EQL 0;			
Condition codes	Vßov	erflow;		
	C ß carry from most significant bit (integer);			
	C ß 0 (floating);			
	Intege	er overflow		
	Floating overflow			
Exceptions	Floating underflow			
	Reserved operand			
	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	
Opcodes	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	
	In 2 operand format, the addend operand is added to the sum operand and the sum operand is replaced by the			

Description	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.		
	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.		
Notes	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 ß1; Z ß 0; V ß 1; and C ß 0.		

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"

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
               # N = 1, V = 1
addb2 $1, r0
               \# N = 0, V = 0
addl2 $1, r0
               \# N = 1, V = 0
addb2 $1, r0
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
```

Another flags example:

.text		
main: .word 0 movb \$0x7C, r0 addb3 \$-1, \$1, r0	# Z = 1, C = 1	
pushl \$0 calls \$1, .exit		

ADWC ADD WITH CARRY

Purpose	perform extended-precision addition		
Format	opcode add.rl, sum.ml		
Operation	sum ß sum + add + C;		
	N ß sum LSS 0;		
	Z ß sum EQL 0;		
Condition codes	V ß {integer overlow};		
	C ß {carry from most significant bit};		
Exceptions	Integer overflow		
Opcodes	D8 ADWC A	dd 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 i low order bits of the results.	s replaced by the	
	2. The two additions in the operation simultaneously.	on are performed	

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 $0x7FFFFFF, r1 # V = 1, N = 1

pushl $0

calls $1, .exit
```

AOB ADD ONE AND BRANCH

Purpose	increment integer loop count and loop		
Format	opcode limit.rl, index.ml, displ.bb		
	index ß index + 1; if index LSS limit AOBLSS		
Operation	then PC ß PC + SEXT (displ); if index LEQ limit AOBLEQ		
	then PC & PC + SEXT (displ);		
	N ß index LSS 0;		
	Z ß index equal 0;		
Condition codes	C ß {integer overflow};		
	C ß C;		
Exceptions	integer overflow		
	F2 AOBLSSAdd One and Branch Less Than		
Opcodes	F3 AOBLEQAdd One and Branch Less Than orEqual		
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.		

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

text
nain: .word 0 movl \$1, r1
orLoop: pushl r1 pushal format calls \$2, .printf aoblss \$10, r1, forLoop
pushl \$0 calls \$1, .exit
data
ormat: .asciz "%d "

ASH ARITHMETIC SHIFT

Purpose	shift of integer				
Format	opcode cnt.rb, src.rx, dst.wx				
Operation	dst ß src shifted cnt bits;				
	N ß dst LSS 0;				
	Z ß dst EQL 0;				
Condition codes	V ß {integer overflow};				
	C ß 0;				
Exceptions	Integer overflow				
	78 ASHL Arithmetic Shift Long				
Opcodes	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.				
	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				
Notes	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.

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 \$0xFFFFFFF, r1 ashl \$4, r1, r2 pushl r2 pushal format2 calls \$2, .printf # Prints FFFFFF0 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 FFFFF0

pushl \$0 calls \$1, .exit

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

B BRANCH ON (CONDTION)

Purpose	test condition code				
Format	opcode displ.bb				
Operation	if condition then PC ß PC + SEXT (displ);				
	NßN;				
	Z ß Z;				
Condition codes	VßV;				
	CßC;				
Exceptions	none				
	CONDITION				
	12 Z EQL 0 Equal (signed)	BNEQ,	Branch on Not		
		BNEQU, Br	anch on Not Equal		
	Unsigned				
	13 Z EQL 1BEQL, Branch on Equal (signed)				
	BEOLU, Branch on Equal				
	Unsigned				
	14 {N OR Z}EQL 0 (signed)	BGTR, Bran	ich on Greater Than		
	15 {N OR Z}EQL 1 Equal	BLEQ, Bran (signed)	ich on Less Than or		
	18 N EQL 0 or Equal	BGEQ, Brar (signe	ıch on Greater Than ed)		
Opcodes	19 N EQL 1 (signed)	BLSS, Bran	ich on Less Than		
	1A {C OR Z}EQL 0	BGTRU, Br Unsigne	anch on Greater Than ed		
	1B {C OR Z}EQL 1 Equal	BLEQU, Br Unsigned	anch Less Than or		

	1C V EQL 0	BVC, Branch on Overflow Clear			
	1D V EQL 1	BVS, Branch on Overflow Set			
	1E C EQL 0 or Equal	BGEQU, Branch on Greater Than unsigned			
		BCC Branch on Carry Clear			
	1F C EQL 1 Unsigned	BLSSU, Branch on Less Than			
		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.				
	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			
Notes	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
```

<pre>jmp end_prog eq2: calls \$0, prn_eq</pre>
end_prog: pushl \$0 calls \$1, .exit
<pre>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</pre>
.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
<pre># 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</pre>
<pre>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</pre>
end_prog: pushl \$0 calls \$1, .exit
<pre>prn_not_eq: .word 0 pushal not_eq pushal format calls \$2, .printf ret prn_eq: .word 0 pushal eq pushal format</pre>
calls \$2, .printf ret
ed. 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"
```

BB BRANCH ON BIT

Purpose	test selected bit				
Format	opcode pos.rl, base.vb, displ.bb				
	teststate = if {BBS} then 1 else 0;				
Operation	if FIELD (pos, 1, base) EQL teststate then				
	PC ß I	PC + SEXT	(displ);		
	NßN	•			
	ZßZ;				
Condition codes	VßV;				
	C ß C;				
Exceptions	reserv	ed operand			
	E0	BBS	Branch on Bit Set		
Opcodes	El	BBC	Branch on Bit Clear		
	The single bit field specified by the position and base				
Description	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.				
	1. A reserved operand fault occurs if pos GTRU 31 and the bit is contained in a register.				
Notes	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 instruction				

.text
main: .word 0
movb \$0x80, r1
bbs \$7 r1 prp true
colle \$0, pro false
cans ao, prin_raise
jmp end_prog
eq2: calls \$0, prn_false
end prog:
colle \$1 ovit
prn_false: .word 0
pushal lbl_false
pushal format
calls \$2printf
rot
pm_ude
pushal lbl_true
pushal format
calls \$2, .printf
imp end prog
J-r
data
Ibl true: asciz "True"
Ibl_falce. asciz "Ealco"
IDI_IAISE: .asciz Faise
tormat: .asciz "%s\n"

BB BRANCH ON BIT (AND MODIFY WITHOUT INTERLOCKED)

Purpose	test and modify selected bit			
Format	opcode pos.rl, base.vb, displ.bb			
	teststate = if {BBSS or BBSC} then 1 else 0;			
	newstate = if {BBSS or BBCS} then 1 else 0;			
	temp ß FIELD (pos, 1, base);			
Operation	FIELD (pos, 1, base) ß newstate;			

	if tmp EQL teststate then			
	PC ß PC + SEXT (displ);			
	N ß N:			
Condition codes	Z IS Z; V ß V;			
	CßC;			
Exceptions	reserved operand			
	E2	BBSS	Branch on Bit Set and Set	
	E3 Set	BBCS	Branch on Bit Clear and	
Opcodes	E4 Clear	BBSC	Branch on Bit Set and	
	E5 Clear	BBCC	Branch on Bit Clear and	
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.			
	1. A reserved operand fault occurs if 0 BTRU 31 and the bit is contained in a register.			
Notes	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.			

BB BRANCH ON BIT INTERLOCKED

Purpose	test and modify selected bit under memory interlock				
Format	opcode pos.rl, base.vb, displ.bb				
	teststate = if {BBSSI} the 1 else 0;				
Operation Condition codes	newstafe = teststate;				
	{set interlock};				
	temp ß FIELD (pos, 1, base);				
	FIELD (pos, 1, base) ß newstate;				
	{release interlock};				
	if tmp EQL teststate then				
	PC & PC + SEXT (displ);				
	N ß N;				
	Z ß Z;				
	VßV;				
	CßC;				
Exceptions	reserved operand				
Opcodes	E6 BBSSI Branch on Bit Set and Set				
	Interlocked				
	E7 BBCCI Branch on Bit Clear and Clear				
	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.				
Description	tested bit is put in the new state as indicated by the				
Description	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.		

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 ope			3 operand	
Operation	dst ßdst AND {NOT mask}; 2 operand				
	dst ß src AND {NOT mask}; 3 operand				
Condition codes	N ß dst LSS 0;				
	Z ß dst EQL 0;				
	V ß 0;				
	С ß С;				
Exceptions	None				
Opcodes	8A	BICB2	Bit Clear By	te; 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 \$0xFFFFFFF, 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 "%lX\n"		

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 = 0xFbisb2 \$0xf0, r4 # N = 1, r4 = 0xFF bisl2 \$0xf0, r4 # All flags are zero, r4 = 0xFFbicb2 \$0xff, r4 # Z = 1, r4 = 0x00 bicb2 \$0xff, r4 # Z = 1, r4 = 0x00movb \$0xFF , r1 incb r1 bisl2 \$0x80000000 , r4 # N = 1, C = 1 bisw2 \$0x0F0F , r4 # N = 0, C = 1, r4 = 0x80000F0F **bicl2** \$0xFFFFFFF, r4 # Z = 1, C = 1 halt

BIS BIT SET

Purpose	perform logical inclusive OR of two integers				
	opcod	opcode mask.rx, dst.mx			
Format	орсос	le mask.rx, si	3 operand		
	dst . ſ	dst OR mas	k;	2 operand	
Operation	dst ß src OR mask;			3 operand	
	N ß d	st LSS 0;			
	Zßd	st EQL 0;			
Condition codes	Vß0	•			
	CßC	•			
Exceptions	None	None			
	88	BISB2	Bit Set Byt	e 2 Operand	
	89	BISB3	Bit Set Byt	e 3 Operand	
	A8	BISW2	Bit Set Wor	d 2 Operand	
Opcodes	A9	BISW3	Bit Set Wor	d 3 Operand	
	C8	BISL2	Bit Set Long 2 Operand		
	C9	BISL3	Bit Set Lon	g 3 Operand	
	In 2 c	perand forma	at, the mask ope	erand is ORed with the	
Description	destination operand and the destination operand is				
Description	operand is ORed with the source operand and the				
	destir	ation operan	d 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 "%lX\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 "%lX\n"		

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 = 0xFbisb2 \$0xf0, r4 # N = 1, r4 = 0xFF bisl2 \$0xf0, r4 # All flags are zero, r4 = 0xFFbicb2 \$0xff, r4 # Z = 1, r4 = 0x00 bicb2 \$0xff, r4 # Z = 1, r4 = 0x00movb \$0xFF , r1 incb r1 bisl2 \$0x80000000 , r4 # N = 1, C = 1 bisw2 \$0x0F0F , r4 # N = 0, C = 1, r4 = 0x80000F0F **bicl2** \$0xFFFFFFF, r4 # Z = 1, C = 1 halt

BISPSW, BICPSW BIT SET PSW, BIT CLEAR PSW PSL

Purpose	set or clear trap enables			
Format	opcode mask.rw			
Operation	PSW ß PSW OR mask; BISPSW PSW ß PSW AND {NOT mask};			
	N ß N OR mask <3>; BISPSW			
	Z β Z OR mask <2>; V β V OR mask <1>:			
Condition codes	C β C OR mask <0>;			
Condition codes	N ß N AND {NOT mask} <3>; BICPSW			
	Z ß Z AND {NOT mask} <2>;			
	V ß VAND {NOT mask} <1>;			
	C ß C AND {NOT mask} <0>;			
Exceptions	Reserved Operand			
	B8 BISPSW Bit set PSW			
Opcodes	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.			

text	
nain: .word 0 bispsw \$0x55 bicpsw \$5	
halt	

BIT BIT TEST

Purpose	test a set of bits for all zero			
Format	opcode mask.rx, src.rx			
Operation	tmp ß src AND mask;			
	N ß tmp LSS 0;			
	Z ß tmp EQL 0;			
Condition codes	V ß 0;			
	C ß C;			
Exceptions	None			
	93 BITB Bit Test Byte			
Opcodes	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"
```

BLB BRANCH ON LOW BIT

Purpose	test bi	t			
Format	opcod	opcode src.rl, displ.bb			
	teststa	te = if {BLE	3S} then 1 else 0;		
Operation	if src<0> EQL teststate then				
	PC ß I	PC + SEXT	(displ);		
	NßN	,			
	Z ß Z;				
Condition codes	VßV;				
	CßC	• •			
Exceptions	none				
On an Inc	E8	BLBS	Branch on Low Bit Set		
Opcodes	E9	BLBC	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 so althou	ource operan gh only one	d is taken with longword context 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"		

BR, JMP BRANCH, JUMP

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

BPT BREAKPOINT FAULT

Purpose	stop for debugging				
Format	opcode				
Operation	PSL <tp>ß0;</tp>				
	N ß 0;				
	Z ß O;				
Condition codes	V ß 0;				
	Сß0;				
Exceptions	none				
Opcodes	03 BPT Breakpoint Fault				
Description	This instruction is used, together with the T-bit, to implement debugging facilities.				

BSB, JSB SUBROUTINE INSTRUCTIONS JUMP, BRANCH TO SUBROUTINE

Purpose	transfe	transfer control to subroutine			
.	opcod subrou	e displ.bx itine	b	ranch to	
Format	opcod subrou	e dst.ab itine	ju	mp to	
	-(SP)	ß PC;			
Operation	PC ß I subrou	PC + SEXT utine	'(displ); bra	anch to	
	PC ß o subrou	lst; ıtine	jun	np to	
	NßN	;			
Condition codes	ZßZ;				
Condition codes	VßV	;			
	C ß C	,			
Exceptions	none				
	10 Displa	BSBB acement	Branch to Subroutine with B	Byte	
Opcodes	30 Displa	BSBW acement	Branch to Subroutine with V	Vord	
	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 evalua JSB ca	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)+.

CALLG CALL PROCEDURE WITH GENERAL ARGUMENT LIST

Purpose	invoke a procedure with actual arguments from anywhere in memory				
Format	opcode arglist.ab, dst.ab				
	{align stack};				
	{create stack frame);				
Operation	{set arithmetic trap enables};				
	{set new values of AP, EP, PC};				
	N ß 0;				
	Z ß 0;				
Condition codes	V ß 0;				
	C ß 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.
	 If bits 13:12 of the entry mask are not 0, a reserved operand fault occurs. On a reserved operand fault, condition codes are
	unpredictable.
Notes	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.

CALLS CALL PROCEDURE WITH STACK ARGUMENT LIST

Purpose	invoke a procedure with actual arguments or addresses on the			
, r	stack			
Format	opcode numarg.rl, dst.ab			
	{push arg count};			
	{align stack};			
Operation	{create stack frame);			
	{set arithmetic trap enables};			
	{set new values of AP, EP, PC};			
	N ß 0;			
	Z ß 0;			
Condition codes	V ß 0;			
	Сß0;			
Exceptions	reserved operand			
Opcodes	FBCALLSCall Procedure With StackArgument List			
	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,			

Description	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	 If bits 13:12 of the entry mask are not 0, a reserved operand fault occurs. 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.

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"

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				
	tmp ß selector - base;				
Operation	PC ß PC + if tmp LEQU limit then				
	SEXT	(displ [tmp]) else {2	+ 2* ZEST (limit)};		
	N ß te	emp LSS limit;			
	Z ß te	mp EQL limit;			
Condition codes	V ß 0;	•			
	C ß temp LSSU limit;				
Exceptions	none				
	8F	CASEB	Case Byte		
Opcodes	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.				
	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].				

Notes	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.

CLR CLEAR

Purpose	clear a scalar quantity				
Format	opcode dst.wx				
Operation	dst ß 0;				
	N ß 0:				
	Z ß 1;				
Condition codes	V ß 0;				
	C ß C;				
Exceptions	None				
	94 CLRB Clear Byte				
	B4 CLRW Clear Word				
	D4 CLRL Clear Long				
Opcodes	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 "%lX %lX\n"

CMP COMPARE

Purpose	arithmetic comparison between two scalar quantities				
Format	opcode srcl.rx, src2.rx				
Operation	srcl - src2;				
	Nßs	rcl LSS src2;			
	Zßs	rcl EQL src2;			
Condition codes	V ß 0;				
	C ß srcl LSSU src2 (integer);				
	C ß C (floating);				
Exceptions	None (integer); reserved operand (floating point)				
	91	CMPB	Compare Byte		
	Bi	CMPW	Compare Word		
Opcodes	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"
```

CMPC COMPARE CHARACTERS

Purpose	to compare two character strings				
	opcode len.rw, src1addr.ab, src2addr.ab 3 operand				
Format	opcode src1len.rw, src1addr.ab, fill.rb, src2len.rw 5 operand				
	src2addr.ab				
	Compare bytes in order from start of string.				
Operation	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 ß {first byte} LSS {second byte};				
Condition codes	Z ß {first byte} EQL {second byte};				
	V IS U;				
	C is {Illist byte} LSSU {second byte};				
Exceptions	None				
Opendas	29CMPC3Compare Characters 3 Operand				
Opcodes	2D CMPC5 Compare Characters 5 Operand				
	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				

Description	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.		
	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:		
Notes	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		

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.

CVT CONVERT

Purpose	convert a signed quantity to a different signed data type			
Format	opcode src.rx, dst.wy			
Operation	dst ß conversion of src;			
	Nßds	st LSS 0;		
Condition codes	Z IS dst EQL 0;			
Condition codes	Vß {s	src cannot be re	presented in dst};	
	Сß0;			
	Intege	er overflow		
Exceptions	Floati	ng overflow		
	Reserved operand			
	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	
Operation codes	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 Long	CVTRFL Convert Rounded F		Convert Rounded Floating to		
	6A	CVTDL		Convert Double to Long		
	6B Long	CVTRDL		Convert Rounded Double to		
	56	CVTFD		Convert Floating to Double		
	76 CVTD			Convert Double to Floating		
	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:					
	CVTE trunca	SF (ted	exact	CVTFW		
	CVTE trunca	SD e ted	exact	CVTDW		
Description	CVTV trunca	VF e ted	exact	CVTFL		
	CVTV trunca	VD e ted	exact	CVTRFL		
	CVTL trunca	F ا ted	rounde	ed CVTDL		
	CVTL rounde	،D و ed	exact	CVTRDL		
	CVTF exact	B 1	truncat	ted CVTFD		
	CVTE rounde)B t ed	truncat	ted CVTDF		
	1. Inte source	ger overf operand	low oco are not	ccurs if any truncated bits of the t 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.			
Notes	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 ß 1; Z ß 0; V ß 1; and C ß 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.			

```
.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
```

DEC DECREMENT

Purpose	subtract 1 from an integer				
Format	opcode dif.mx				
Operation	dif ß dif - 1;				
	N ß dif LSS 0;				
Condition codes	Z ß dif EQL 0;				
	V ß {integer overflow};				
	C ß {borrow from most significant bit};				
Exceptions	Integer overflow				
	97	DECB	Decrement Byte		
Opcodes	B7	DECW	Decrement Word		
	D7	DECL	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"	

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 $0x0000000, r0
   decl r0
               # N = 1, C = 1
   movl $0x8000000, r0
   decl r0
               # V = 1
```

movw \$0x8	3000, r0		
decw r0	# V = 1		
movb \$0x8	0, r0		
decb r0	# V = 1		
pushl \$0			
calls \$1, .ex	xit		

DIV DIVIDE

Purpose	perform arithmetic division					
	opcode divr.rx, quo.mx 2 operand					
Format	opcode divr.rx, divd.rx, quo.wx 3 operand					
	quo ß	quo / divr;	2 operand			
Operation	quo ß	divd / divr;	3 operand			
	N ß quo LSS 0;					
	Z ß quo EQL 0;					
Condition codes	V ß {overflow} OR {divr EQL 0};					
	C ß 0	,				
	Intege	er overflow				
Exceptions	Divid	Divide by zero				
	Floating overflow					
	Floating underflow					
	Reserved operand					
	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			
Opcodes	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			
	In 2 operand format, the quotient operand is divided by the divisor operand and the quotient operand is replaced					
Description	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 ß 1; Z ß 0; V ß 1; and C ß 0.					

EDIV EXTENDED DIVIDE

Purpose	perform extended-precision division				
Format	opcode divr.rl, divd.rq, quo.wl, rem.wl				
Operation	quo ß divd/divr;				
	rem ß REM{dvid, divr};				
	N ß quo LSS 0;				
	Z ß que EQL 0;				
Condition codes	V ß {integer overflow} OR {divr EQL 0};				
	C ß 0;				
D	Integer overflow				
Exceptions	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.				

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			
Oracratica	<pre>int ß integer part of muld* {mulr'mulrx};</pre>			
Operation	<pre>frac & fractional part of muld * {mulr'mulrx);</pre>			
	N ß fract LSS 0;			
Condition order	Z ß fract EQL 0;			
Condition codes	V ß {integer overflow};			
	Сß0;			
	Integer overflow			
Exceptions	Floating underflow			
	Reserved operand			
	54 EMODF Extended Multiply and Integrate Floating			
Opcodes	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.					

EMUL EXTENDED MULTIPLY

Purpose	perform extended-precision multiplication			
Format	opcode mulr.rl, muld.rl, add.rl, prod.wq			
Operation	prod ß muld * mulr + SEXT(add);			
	N ß prod LSS 0;			
	Z ß prod EQL 0;			
Condition codes	V ß 0;			
	Сß0;			
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.			

HALT

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

INC INCREMENT

Purpose	add 1 to an integer			
Format	opcode sum.mx			
Operation	sum ß sum +1;			
	N ß sum LSS 0;			
Condition codes	Z ß sum EQL 0;			
	V ß {integer overflow};			
	C ß {carry from most significant bit};			
Exceptions	Integer overflow			
	96 Increment Byte			
Opcodes	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"
```

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 # r0 contains 0, C = 1, V = 0, Z = 1 incb r0 movw \$0xFFFF, r0 incw r0 # r0 contains 0, C = 1, V = 0, Z = 1 movl \$0xFFFFFFF, r0 # r0 contains 0, C = 1, V = 0, Z = 1 incl r0 movl \$0x7FFFFFFF, r0 # r0 contains 0x80000000, C = 0, V = 1, N = 1 incl r0 movw \$0x7FFF, r0 # r0 contains 0x8000, C = 0, V = 1, N = 1 incw r0 movb \$0x7F, r0 incb r0 # r0 contains 0x80, C = 0, V = 1, N = 1 halt

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 ß {indexin + subscript} *size;			
Operation	{subscript range trap};			
	N ß indexout LSS 0;			
	Z ß indexout EQL 0;			
Condition codes	V ß 0;			
	Сß0;			
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.

INSQUE INSERT ENTRY IN QUEUE

Purpose	add entry to head or tail of queue				
Format	opcode	opcode entry.ab, pred.ab			
	If (all memory accesses can be completed) then begin				
Operation	entry	(entry) ß (pred);	forward link of		
	entry	(entry+4) ß pred;	backward link of		
	succes	((pred)+4) ß entry; sor	backward link of		
	predec	(pred) ß entry; essor	forward link of		
	end;				
	else				
	begin				
	{backup instruction};				
		{initiate fault}			
	end;				
	Nß (ei	ntry) LSS (entry+4);			
	Zß (er	ntry) EQL (entry+4);	first entry in queue		
Condition codes	V ß 0;				
	C ß (entry) LSSU (entry+4);				
Exceptions	None				
Opcodes	OE	INSQUE	Insert Entry in Queue		
	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				

Description	one in the queue, the condition otherwise it is cleared. The in interruptible operation. Before operation, the processor valid operation can be completed. The memory management exception in a consistent state.	on code Z-bit is set; sertion is a non- e performing any part of the lates that the entire This ensures that if a ton occurs, the queue is left			
	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.				
Notes	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			

```
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

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			
	N ß 0;			
	Z ßR0 EQL 0;			
Condition codes	V ß 0;			
	Сß0;			
Exceptions	None			
	3A LOCC Locate Character			
Opcodes	3BSKPCSkip 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.			
	1. After execution:			
	R0 = number of bytes remaining in the string (including located one) if byte located; otherwise $R0$ = 0.			
Notes	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.

.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"

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				
	N ß 0;				
	V ßR0 EQL 0;				
Condition codes	V ß 0;				
	C ß 0;				
Exceptions	None				
Opcodes	39MATCHCMatch 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.				
	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.				
Notes	R3 = if a match occurred, the address of 1 byte beyond				
Notes	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.

MCOM MOVE COMPLEMENTED

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

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

MNEG MOVE NEGATED

Purpose	move the arithmetic negation of a scalar quantity			
Format	opcode src.rx, dst.wx			
Operation	dst ß -scr;			
	N ß dst LSS 0;			
	Z ß dst EOL 0;			
	V ß overflow (Integer);			
Condition codes	Vß0(floating);			
	C ß dst NEQ 0 (integer);			
	C ß 0 (floating);			
Exceptions	Integer overflow; reserved operand (floating)			
	8E MNEGB Move Negated Byte			
	AE MNEGW Move Negated Word			
Opcodes	CE MNEGL Move Negated Long			
•	52 MNEGF Move Negated Floating			
	72 MNEGD Move Negated Double			
	The destination operand is replaced by the negative of			
Description	the			
	source operand.			
	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.			
Notes	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.

.text	
main: .word 0	
movb \$1, r1	# r1 contains 1
mnegw r1, r2	<pre># r2 contains 0xFFFF</pre>
mnegw r2, r3	# r3 contains 1
mnegl r3, r4	<pre># r4 contains 0xFFFFFFFF</pre>
halt	

MOV

Purpose	move a scalar quantity			
Format	opcode src.rx, dst.wx			
Operation	dst ß s	dst ß src		
	N ß dst LSS 0			
	ZßdstEQL0			
Condition codes	V ß 0			
	CßC			
Exceptions	None	(integer); Rese	rved operand (floating point)	
	90	MOVB	Move Byte	
	BO	MOVW	Move Word	
	DO	MOVL	Move Long	
Operation codes	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.			
	1. On a floating reserved operand fault, the			
	are			
	unpredictable.			
Notes	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.			

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

MOVA, PUSHA MOVE ADDRESS, PUSH ADDRESS

Purpose	calcu	calculate address of quantity		
	opcode src.ax, dst.wl		MOVA	
Format	opcoc	le src.ax	PUSHA	
	dst ß :	src;	MOVA	
Operation	-(SP)	ß src;	PUSHA	
	N ß re	esult LSS 0;		
	Z ß re	esult EQL 0;		
Condition codes	VßO	•		
	CßC	•		
Exceptions	None			
	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	
Opcodes	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	
	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			
Description				

	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.

text	
nain: .word 0 moval myString, r0 pushl r0 calls \$1, .puts	
pushl \$0 calls \$1, .exit	
data nyString: .asciz "Hello, World"	

MOVC MOVE CHARACTER

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

Description	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.	
	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	
Notes	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	

.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

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	
	N ß srclen LSS dstlen;
Condition order	Z ß srclen EQL dstlen;
Condition codes	V ß 0;
	C ß srclen LSSU dstlen;
Exceptions	None
Opcodes	2EMOVTCMove TranslatedCharacters
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 is unpredictable.
	1 After execution: R0 = number of translated bytes remaining in source string, R0 is non-zero only if source string is

	longer tha	an destination string.
Notes	R Source	1 = address of one byte beyond the last byte in string that was translated.
	R	2 = 0
	R	3 = address of the translation table.
	R	4 = 0
	R	5 = address on one byte beyond the destination
	string	

.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

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		
	N ß srclen LSS dstlen;	
Condition or dee	Z ß srclen EQL dstlen;	
Condition codes	V ß {terminated by escape};	
	C ß srclen LSSU dstlen;	
Exceptions	None	
Opcodes	2FMOVTUCMove Translated UntilCharacter	
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.	
	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
Notes	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.

MOVPSL MOVE FROM PSL

Purpose	obtain processor status		
Format	opcode dst.wl		
Operation	dst ß PSL;		
Condition codes	NßN;		
	Z ß Z;		
	VßV;		
	C ß 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
<pre># 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</pre>
pushl \$0 calls \$1, .exit
format: .asciz "PSW is %d. R0 is %d\n"

MOVZ MOVE ZERO-EXTENDED

Purpose	convert an unsigned integer to a wider unsigned integer	
Format	opcode src.rx, dst.wy	
Operation	dst ß ZEXT (src);	
	N ß 0;	
	Z ß dst EQL 0;	
Condition codes	V ß 0;	
	CßC;	
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"
MUL MULTIPLY

Purpose	perform arithmetic multiplication				
T	opcod	e mulr.rx, prod.	2 operand		
Format	opcod	e mulr.rx, muld	.rx, prod.wx	3 operand	
	prod ß prod * mulr;			2 operand	
Operation	prod ß muld * mulr;			3 operand	
	N ß pr	rod LSS 0;			
	Z ß prod EQL 0;				
Condition codes	Vßov	verflow;			
	Сß0;				
	Intege	r overflow			
—	Floating overflow				
Exceptions	Floating underflow				
	Reserved operand				
	84	MULB2	Multiply Byte 2 Op	erand	
	85	MULB3	Multiply Byte 3 Op	erand	
	A4	MULW2	Multiply Word 2 O	perand	
	A5	MULW3	Multiply Word 3 O	perand	
	C4	MULL2	Multiply Long 2 Op	berand	
Opcodes	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	
	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				

Description	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.		
	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.		
Notes	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). Nß 1: Z ß 0; V ß 1; and C ß 0.		

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"

POLY POLYNOMINAL EVALUATION

Purpose	allows fast calculation of math functions			
Format	opcode arg.rx, degree.rw, tbladdr.ab			
	result C ß degree; For degree times, loop result ß arg * result; Perform multiply, and retain an extended floating traction of 31 bits (POLYF) or 63 bits			
Operation	(the fraction is truncated before normalization) use this result in the following step			
	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	55POLYFPolynomial Evaluation Floating75POLYDPolynomial Evaluation Double			
	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.			
Description	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.			
	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:
Notes	• If PSL <fpd> = 0, the reserved operand is either the degree operand (greater than 31), or the argument operand, or some coefficient.</fpd>
	• If PSL <fpd> = 1, the reserved operand is a coefficient, and R3 is pointing at the value which caused the exception.</fpd>
	• 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.

POPR POP REGISTERS

Purpose	restore multiple registers from stack				
Format	opcode mask.rw				
Operation					
	NßN;				
	Z ß Z;				
Condition codes	VßV;				
	C ß C;				
Exceptions	None				
Opcodes	BAPOPRPop 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.</n>				
	This instruction is similar to the sequence				
	MOVL (SP)+,RO				
Notes	MOVL (SP)+,R1				
	MOVL (SP)+,R14				
	where only the masked registers are popped.				

PUSHL PUSH LONG

Purpose	push source operand onto stack				
Format	opcode src.rl				
Operation	-(SP) ß src;				
Condition codes	N ß src LSS 0;				
	Z ß src EQL 0;				
	V ß 0:				
	СßС;				
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.

ext
hain: .word 0 pushl \$100 movl (sp), r1 pushl r1 pushal format calls \$2, .printf
pushl \$0 calls \$1, .exit
ormat: .asciz "R1 is %d\n"

PUSHR PUSH REGISTERS

Purpose	save multiple registers or stack				
Format	opcode mask.rw				
Operation					
	NßN;				
	Z ß Z;				
Condition codes	VßV;				
	CßC;				
Exceptions	None				
Opcodes	BB	PUSH	R	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.</n>				
	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				
Netze	2. This	instruc	ilar to the sequence		
Notes	PUSH	Ĺ	R14		
	PUSH	Ĺ	R13		
	•••				
	PUSH	Ĺ	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
nain: .word 0
movl \$1, r0
movl \$2, r1
pushr \$3
pushal format
calls \$3, .printf
pushl \$0
calls \$1, .exit
data
ormat: .asciz "%d %d\n"

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)) ß (entry); forward link of predecessor ((entry)+4) ß (entry+4); backward link of successor addr ß entry; end; else begin {backup instruction}; {initiate fault} end;			
Condition codes	N ß (entry) LSS (entry+4);Z ß (entry) EQL (entry+4);removed last entryV ß entry EQL (entry+4);no entry to removeC ß (entry) LSSU (entry+4);			
Exceptions	None			
Opcodes	OF REMQUE Remove Entry from Queue			
	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-			

Description	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.	
	1. Because the removal is non-interruptible, processes running in kernel mode can share queues with interrupt ser vice routines.	
Notes	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.	

.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

REI - RETURN FROM EXCEPTION OR INTERRUPT

Purpose	exit from an exception or interrupt service routine		
Format	opcode		
Operation	<pre>tmp1ß(SP)+; tmp2ß(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, dv,="" fu,<br="" is,="">IV> NEQU 0 } or { tmp2<current_mode> NEQU 3} then { reserved operand fault}; 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 N & save PSL<3>;</tp></tp></current_mode></is></current_mode></fpd,></cm></psl_mbz></ipl></ipl></current_mode></prev_mode></current_mode></ipl></ipi></is></current_mode></is></is></is></current_mode></current_mode></pre>		

Condition codes	Z ß save PSL<2>;			
	V ß save PSL<1>;			
	C ß 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	 and the or the exception of interrupt. Thy instruction look ahead in the processor is reinitialized. 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 			

RET RETURN FROM PROCEDURE

Purpose	transfer control from a procedure back to calling		
	program		
Format	opcode		
	{restore SP from FP};		
	{restore registers};		
Operation	{drop stack alignment};		
	{restore PSW};		
	{If CALLS, remove arglist};		
	N ß restored PSW <3>;		
	Z ß restored PSW <2>;		
Condition codes	V ß restored PSW <1>;		
	C ß 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	 A reserved operand fault occurs if temporary1 <15:8> NEG 0. 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.

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 .word 0 func1: movl \$99, r1 pushl r1 pushal format calls \$2, .printf ret .data format: .asciz "R1 is %d\n"

ROTL ROTATE LONG

Purpose	rotate of integer		
Format	opcode cnt.rb, src.rl, dst.wl		
Operation	dst ß src rotated cnt bits;		
	N ß dst LSS 0;		
	Z ß dst EQL 0;		
Condition codes	V ß 0;		
	C ß 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

1.	- 1	1.
n	а	IT.
11	u.	ιι

RSB RETURN FROM SUBROUTINE

Purpose	return control from subroutine			
Format	opcode			
Operation	PC ß (SP) +;			
	N ß N;			
	Z ß Z;			
Condition codes	VßV;			
	C ß 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.			

SBWC SUBTRACT WITH CARRY

Purpose	perform extended-precision subtraction		
Format	opcode sub.rl, dif.ml		
Operation	dif ß dif - sub - C;		
	N ß dif LSS 0;		
	Z ß dif EQL 0;		
Condition codes	V ß {integer overflow};		
	C ß {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 \$0x8000 movl \$1, r2 sbwc r2, r1	0000, r1 # V = 1		
movl \$0x8000 movl \$1, r2 bispsw \$1 sbwc r2, r1 # V	0001, r1 7 = 1		
pushl \$0 calls \$1, .exit			

SOB SUETRACT ONE AND BRANCH

Purpose	decrement integer loop count and loop			
Format	opcode index.ml, displ.bb			
	index ß index -1: SOBGEQ			
	If index GEQ 0 then			
Onevation	PC ß PC + SEXT (displ);			
Operation	index ß index-1: SOBGTR			
	If index GTR 0 then			
	PC ß PC + SEXT (displ);			
	N ß index LSS 0;			
	Z ß index EQL 0;			
Condition codes	Vß {integer overflow};			
	C ß C;			
Exceptions	integer overflow			
Opcodes	F4SOBGEOSubtract One and BranchGreaterThan or Equal			
	F5SOBGTRSubtract One and BranchGreaterThan			
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.			
	1. Integer overflow occurs if the index operand before subtraction is the largest negative integer. On overflow,			

Notes	the index operand is replaced by the largest positive integer, and thus the branch is taken.
	2. The C-bit is unaffected.

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"

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).		
	N ß 0;		
	V ßR0 EC	QL 0;	
Condition codes	V ß 0;		
	Сß0;		
Exceptions	None		
	2A SC	CANC	Scan Characters
Opcodes	2B SI	PANC	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.		
	 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.		
Notes	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).		

SUB SUBTRACT

Purpose	perform arithmetic subtraction				
Format	opcode sub.rx, dif.mx			2 operand	
Format	орсоо	de sub.rx, min.	rx, dif.wx	3 operand	
	dif ß	dif - sub;	2 operand	1	
Operation	dif ß	min - sub;	3	operand	
	N ß d	if LSS 0;			
	Zßd	if EQL 0;			
Condition codes	Vßo	verflow;			
	C ß {	borrow from n	nost significant bit}(integer	;);	
	C ß 0	(floating);			
	Integ	er overflow			
	Floating overflow				
Exceptions	Floating underflow				
	Reserved operand				
	82	SUBB2	Subtract Byte 2 Operand	d	
	83	SUBB3	Subtract Byte 3 Operand	d	
	A2	SUBW2	Subtract Word 2 Operan	nd	
	A3	SUBW3	Subtract Word 3 Operan	nd	
	C2	SUBL2	Subtract Long 2 Operand	d	
Opcodes	C3	SUBL3	Subtract Long 3 Operand	d	
	42	SUBF2	Subtract Floating 2 Oper	rand	
	43	SUBF3	Subtract Floating 3 Oper	rand	
	62	SUBD2	Subtract Double 2 Oper	and	
	63	SUBD3	Subtract Double 3 Oper	and	
	In 2 operand format, the subtrahend operand is				

Description	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 ß 1; Z ß 0; V ß 1; and C ß 0.			

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 $0x8000002, 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

TST TEST

Purpose	arithmetic compare of a scalar to 0.			
Format	opcode src.rx			
Operation	src ß 0;			
	N ß src LSS 0;			
	Z ß src EQL 0;			
Condition codes	V ß 0;			
	Сß0;			
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.			
	1. TSTx src is equivalent to CMPx src, \$0, but is shorter.			
Notes	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"

XOR EXCLUSIVE OR

Purpose	perform logical exclusive OR of two integers					
Format	opcode mask.rx, dst.mx		mx 2 operand			
	opcod	e mask.rx, src.	rx, dst.wx 3 operand			
Operation	dst ß d	lst XOR mask;	2 operand			
	dst ß s	src XOR mask;	3 operand			
	N ß ds	st LSS 0;				
	Z ß ds	t EQL 0;				
Condition codes	V ß 0;					
	CßC	C ß C;				
Exceptions	None					
	8C	XORB2	Exclusive OR Byte 2 Operand			
	8D	XORB3	Exclusive OR Byte 3 Operand			
	AC	XORW2	Exclusive OR Word 2 Operand			
Opcodes	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 "%lX\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 "%IX\n"

Program's output is FF00.

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"	

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.
.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

.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

.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

.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

.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.

.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.

.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

.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

.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

.QUAD - Quadword storage directive

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

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

.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.

.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.

.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.

.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

circle

circle draws a circle in the current drawing color.

Gets:	OnStack:	Χ, Υ	- Center point of the circle
		radius	- Radius of the circle
Retur	ns: On r): 0 on succ	ess, -1 on failure

cleardevice - clears the graphics screen

Clears the graphics screen

Gets:NothingReturns:On r0: 0, -1 on failure

closegraph - Shuts down the graphics system

Shuts down the graphics system

Nothing
On r0: 0, -1 on failure
closegraph deallocates all memory allocated by the graphics
system. It
then restores the screen to the mode it was in before you called
initgraph.

getmaxx, getmaxy

Returns maximum x or y screen coordinate

Gets: Nothing

Returns: On r0: maximum x or y screen coordinate

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.

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:

t de la constant de la const
ord 0
ls \$0, .initgraph
hl \$0
hl \$0
hl \$720
hl \$424
ls \$4, .line
hl \$424 Is \$4, . <mark>line</mark>

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

putpixel

putpixel plots a pixel at a specified point.

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

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

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

Example:

text
word 0
calls \$0, .initgraph
bushl \$100
bushl \$200
bushl \$100
calls \$3, .setcolor

setfont

Sets the active font.

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

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 movb r0,	# Get char from the keyboard	

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			

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
0	Octal	Integer
Х	Hex	Integer
Х	Hex	Integer (Capital Letters output)
С		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:

1
pushl x
pushal Str
push v
pushal Form1
pushar Formit
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

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

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	
<pre>pushal szWelcomeMessage calls \$1, .puts # Ask for the user's name</pre>	
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, "	

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.

<u>Format specifications</u>: 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.

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
0	Octal	Integer
Х	Hex	Integer
Х	Hex	Integer (Capital Letters output)
С		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.

.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

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.	

.text	
<pre># User program here # pushl \$0 calls \$1, .exit # Exit with error code 0</pre>	

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.

.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"

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.

.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"

Interrupts

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.

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 ¹⁰₁₆ to ¹⁷₁₆ are for use by devices and controllers. Interrupt levels ¹⁸₁₆ to ^{1F}₁₆ are used by urgent conditions, including the interval clock, serious errors, and power fail.

Some known IPLs:

- System Clock: 18₁₆
- Terminal: 14₁₆

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 exceptioncausing 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.

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.
	Loads it, but this is not the normal way of making
"MTPR src,	software interrupt requests. It is useful for clearing
SISR"	the software interrupt system and for reloading it
	after a power failure, for example.

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.

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.

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:

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

Interrupt vectors:

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

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.

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	
			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	
25	NICD	WO	bit7=1 Int	
25	ICR		Next Interval Count Register	
20		ĸo	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	

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.

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.

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.

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.

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.

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

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.

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 Address: 0x%X\n"

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.