Simulation of *minVAX* a CISC CPU

(A Little-Bitty Baby VAX)

Due Date: Monday, April 7th, 2014

Using the C++ *arch* package, described in separate handouts, you are to simulate a complex CISC CPU.

1. The Architecture of the CPU

You will be implementing a number of instructions using the hardware specified in the following diagram.

You may use any type of *StorageObject* for any of the registers specified.

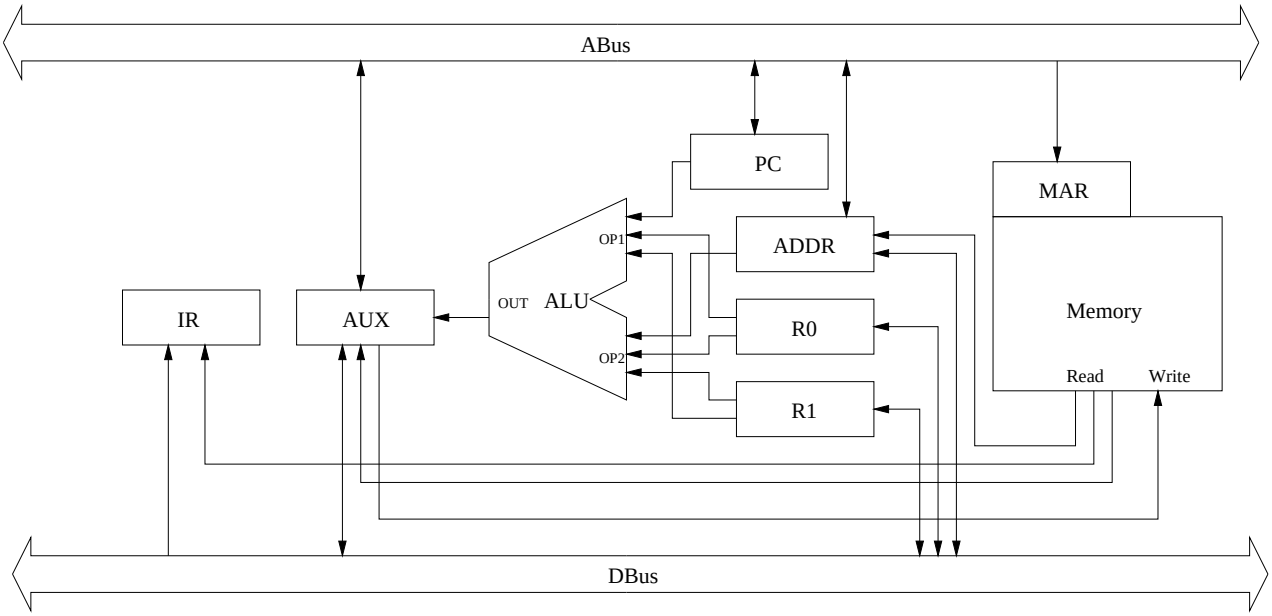
You may add any constants that you feel you need. This can include *arch* package constants (which are *OutFlows*) or constant registers (not an *arch* package object, just a register that you never change its value). You can connect your constants to any *Inflow* of any of the other components. You can not connect your new register to any *OutFlow*.

You may not store program data in any register not specified in the data paths diagram (i.e. you can not create a register *JUNK* and then do a $JUNK \leftarrow R0$).

You may not add any extra connections between the components specified in this data paths diagram.

You will need to document, in the **README** file, any extra components that you add to your data path.

1.1. The minVAX Computer Data Paths



CPU Data Paths

Version 1.0 03/28/09

1.2. Main Memory Requirements

The word size is 8 bits, and the address size is 8 bits. This means that there are 256 words of memory, hex 00 to FF.

1.3. Programmer-Usable Registers

There are two programmer visible registers.

Register	Name	Size	Notes
R0 & R1	Reg 0, Reg 1	8	These are the two registers that a user can directly manipulate with the assembly code

1.4. Internal CPU Registers

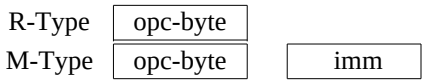
The **minVAX** hardware has a number of internal registers:

Register	Name	Size	Notes
PC	Program Counter	8	Location of instruction to execute
IR	Instruction Reg.	8	Register to hold the instruction while it is executing
ADDR	Address Register	8	Register to hold the 2nd instruction byte, needed by some instructions for the Address Mode calculations
AUX	Auxiliary Reg.	8	Auxiliary register used to hold the output of the ALU. In addition, this register is used to send and receive data from memory.

2. The Instruction Set

2.1. Instruction Layouts

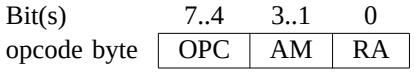
The instructions in the **minVAX** computer come in two formats:



Where both the *opc* and the *imm* are 8-bits long.

The *imm* is an optional byte that contains information used by the address mode of the instruction to compute where one of the operands of the instruction is located.

For each of these formats, the *opc-byte* is further broken down as follows:



Where:

Field	Size	Notes
OPC	4-bits	The opcode for this instruction.
AM	3-bits	Address Mode for this instruction.
RA	1-bit	The primary register for this instruction. The RA register is used as the destination, and one of the source registers for all data manipulation instructions

2.2. Address Modes Supported

The following Address Modes (AM) are supported by the *minVAX* CPU:

AM encoding	Address Mode	Effective Address	Notes
000	Register 0	data in $R0$	The 2nd operand is in $R0$. This can only be used for ALU operations.
001	Register 1	data in $R1$	The 2nd operand is in $R1$. See AM `000` above.
010	Displacement 0	$EA = R0 + imm$	Computes a memory address as a constant offset to register $R0$.
011	Displacement 1	$EA = R1 + imm$	See AM `010` above.
100	Immediate	data in imm	The 2nd operand is in the imm byte. This can only be used for ALU operations.
101	Absolute	$EA = imm$	The immediate byte contains the memory address.
110	PC Relative	$EA = PC + imm$	Computes a memory address as a constant offset to PC register. Note that the PC should be pointing to the address after the imm byte.
111	illegal		Not a legal address mode.

Note that for instructions that need data, then the Effective Address either specifies the data, or the memory location where the data is found (so the data is at $MEM[EA]$).

For instructions that need a memory location, then only the address modes that generate an EA can be used (other address modes will generate an error).

For instructions that don't use a 2nd operand, These instructions will check for illegal AM, but otherwise they will ignore the AM information. This means that all these instructions are in R-Type format, no matter what AM they claim to use.

Note that if the PC-relative or displacement address mode generate an address that is outside the address space, you should just use the truncated location.

2.3. List of Instructions

The *minVAX* CPU supports the following instructions.

OPCODE	MNEMONIC	DESCRIPTION	RTL
		Following inst. use the AM to get the data	
0000	NOP	No operation	
0001	ADD	Add data to register RA	$RA = RA + data(AM)$
0010	AND	Bitwise And data to register RA	$RA = RA \ \& \ data(AM)$
0011	SRA	Shift Right Arithmetic by number of bits specified by data	$RA = RA \gg_a data(AM)$
0100	SLL	Shift Left Logical by number of bits specified by data	$RA = RA \ll data(AM)$
		Following inst. use the AM to get an EA	
0101	LDR	Load reg. RA from memory at the EA specified by the AM	$RA \leftarrow Mem[\ EA\ ]$
0110	STR	Store reg. RA to memory at the EA specified by the AM	$Mem[\ EA\ ] \leftarrow RA$
0111	JMP	Jump to memory location specified by the EA	$PC = EA$
1000	BEZ	Branch to location EA if RA is zero	if $RA == 0$ then $PC = EA$
1001	BLT	Branch to location EA if RA is less than zero	if $RA < 0$ then $PC = EA$
1010	NOP	No operation	
		Following inst. ignore the AM	
1011	CLR	Clear register RA	$RA = 0$
1100	CMP	Complement register RA	$RA = \overline{RA}$
1101	INC	Increment register RA	$RA = RA + 1$
1110	DMP	Dump (print) the value of reg. RA	
1111	HLT	Halt the computer	

Note that the *RTL* field of the table above does not contain the steps needed to execute the instruction, instead it contains the final result of the operation.

3. Simulator Requirements

3.1. Tracing

Every time your simulator fetches a new instruction to run, it should print the address and digital code of the instructions *opc–byte*, its mnemonic, its base register (*RA*) and the address mode, each printed separately. If the instruction/address mode specify the instruction is in M-type, then you should print out the *imm* value. Finally, you should print out the result of the instruction which is either the value of a register that changed (or a register that was dumped), the location and value of any data written to memory, or a message telling if the branch was/was not taken. The format of this should be:

Result	Trace Output	Notes
R*?*	R*?=##*	*?* is reg. number and ## is the new value of the reg
Memory	MEM[@@]=##	@@ is the EA and ## is the value being written
branch taken	BRANCH TAKEN	
branch not taken	BRANCH NOT TAKEN	

Note that the value of the register/memory location is the value after the instruction has completed.

All values are to be shown in hexadecimal, with the appropriate number of leading zeros to make all the trace fields line up.

Here is an example:

```
00:  b0 = CLR 0 0    R0=00
01:  00 = NOP 0 0
02:  b1 = CLR 1 0    R1=00
03:  5b = LDR 1 5 a0 R1=fb
05:  d1 = INC 1 0    R1=fc
06:  5a = LDR 0 5 90 R0=10
08:  12 = ADD 0 1    R0=0c
09:  66 = STR 0 3 84 MEM[80]=0c
0b:  9d = BLT 1 6 05 BRANCH TAKEN
12:  01 = NOP 1 0
13:  7a = JMP 0 5 3e BRANCH TAKEN
3e:  8c = BEZ 0 6 01 BRANCH NOT TAKEN
40:  e0 = DMP 0 0    R0=0c
41:  c0 = CMP 0 0    R0=f3
42:  c4 = CMP 0 2    R0=0c
43:  21 = AND 1 0    R1=0c
44:  b1 = CLR 1 0    R1=00
45:  d1 = INC 1 0    R1=01
46:  b0 = CLR 0 0    R0=00
47:  45 = SLL 1 2 a1 R1=08
49:  39 = SRA 1 4 02 R1=02
4b:  aa = NOP 0 5
4c:  f0 = HLT 0 0

MACHINE HALTED due to halt instruction
```

Make sure you follow the formatting precisely. You can look at the sample output files, discussed below, to see the exact format.

3.2. Command Line

You should call your program "minVAX".

The object code file is named as an argument to the program, e.g., "`minVAX test1.obj`". Since the **Memory** class in the arch package automatically reads the file when you invoke its **load** function, you need not do any error checking on its contents.

3.3. Halting

Besides after executing a HALT instruction, your CPU will need to stop when:

1) it attempts to execute past the end of the legal memory addresses (check this by checking PC overflow)
2) when an illegal address mode is found in an instruction
3) when an invalid address mode is used (instructions that need a memory location can't use AMs that produce a data value

When one of these happens, your simulator should print the message "MACHINE HALTED due to", plus the reason (with the following message):

Reason	Message
PC overflow	MACHINE HALTED due to PC overflow
illegal address mode	MACHINE HALTED due to unknown address mode
invalid address mode	MACHINE HALTED due to invalid address mode
halt instruction	MACHINE HALTED due to halt instruction

4. README

You must submit a file named README with your project. This file should contain your name, and a description of the project. It should also contain a list of all the files submitted describing what the files contain, and why they were submitted.

You should also include a list of any extra hardware components that you added to minVAX's data paths, with a description of what the extra hardware is used for and why you needed it.

In addition, you should include any notes you think would be useful while I grade the assignment.

5. Handing In the Assignment

By the end of the day (i.e. 11:59:59pm on the CS system clock) the program is due. Submit your solution using the command below; you must submit your source files (all `.cpp` and `.h` files you write) and the `README` file. The try script will generate a `Makefile` (using `~csci453/pub/misc/header.mak` and the gmakemake program as discussed in class) and recompile your submission for testing.

To submit your solution, use a command such as

try grd-453 minVAX minVAX.cpp README **OTHER_PROGRAM_FILES**

where **OTHER_PROGRAM_FILES** contains the names of all the other program files (`.cpp` and `.h` files) you are submitting as your solution.

NOTE:

I do not want object code or executable files from you.

Do not submit a directory, just submit all files pertaining to the project.

6. Supplied Software

`header.mak` used by the gmakemake program in ~csci453/pub/misc/header.mak

An executable solution is in ~csci453/pub/bin/ubuntu86/minVAX

Test programs are in ~csci453/pub/minVAX/*.obj

The output for the test programs are in ~csci453/pub/minVAX/*.out