Simulation of *simpISA* a Simple CPU

(Dummest Little Computer grows up)

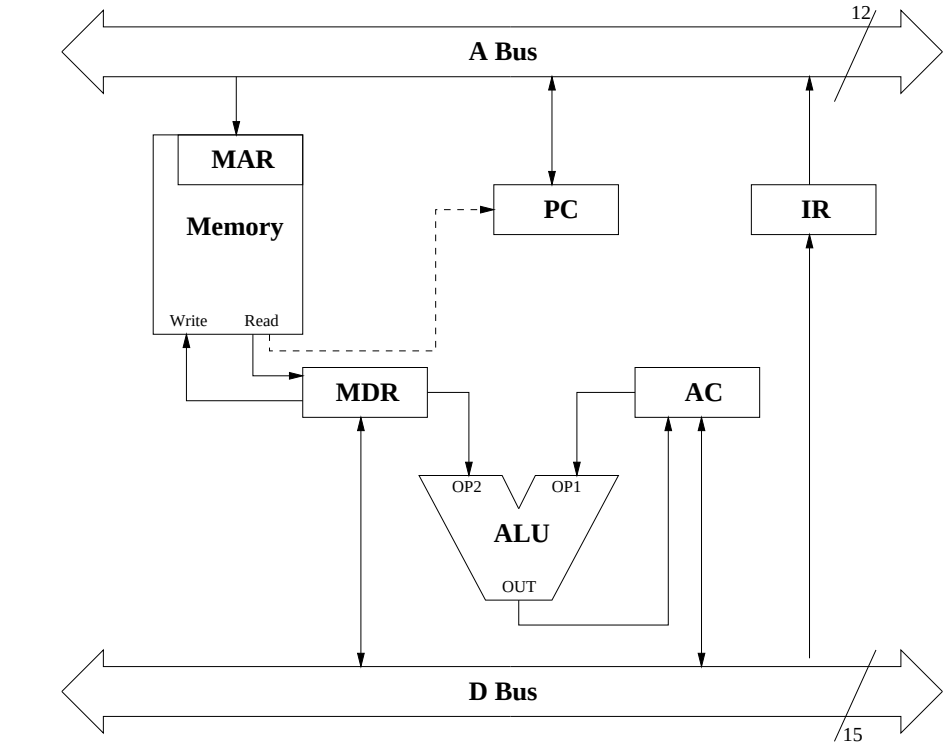
Due Date: **Wednesday, March 5th**

Using the C++ *arch* package, described in separate handouts, you are to simulate a simple 8-instruction CPU.

1. The Architecture of the CPU

You will be implementing a number instructions using the hardware specified in the *Dummest Little Computer* demo program. You should not modify the hardware in any way, instead you will be modifying the control to execute a different instructions set, specified below.

1.1. The Dummest Little Computer Data Paths



dummest1.fig

1.2. Main Memory Requirements

The word size is 15 bits, and the address size is 12 bits. This means that there are 4,096 words of memory, hex 000 to FFF.

1.3. Programmer-Usable Registers

The only register available directly to the programmer is a 15-bit accumulator, abbreviated as *AC*.

1.4. Internal CPU Registers

The Dummest hardware has three internal registers:

Register	Name	Size	Notes
PC	Program Counter	12	Location of instruction to execute
MDR	Memory Data Reg.	15	Temporary register used as the second input to the ALU, and as the location to read-to, and write-from
IR	Instruction Reg.	15	Register to hold the instruction while it is executing

2. The Instruction Set

2.1. Instruction Layouts

The high order three bits of the instruction, (bits 14, 13, and 12), contain the opcode. The low order 12 bits, (bits 11 through 0), contain an address used by the instruction.

2.2. List of Instructions

OPCODE	MNEMONIC	DESCRIPTION	RTL
0	LOAD	Load accumulator from memory address given	$MAR \leftarrow IR_{11\text{–}0}$ $MDR \leftarrow Mem[MAR]$ $AC \leftarrow MDR$
1	STORE	Store accumulator to memory address given	$MAR \leftarrow IR_{11\text{–}0}$, $MDR \leftarrow AC$ $Mem[MAR] \leftarrow MDR$
2	ISZ	Increment the contents of the memory address given. If the result is zero, skip execution of the instruction following this one. Note that the AC should not be changed by this instruction.	$MAR \leftarrow IR_{11\text{–}0}$ $MDR \leftarrow Mem[MAR]$ $MDR \leftarrow MDR + 1$ $Mem[MAR] \leftarrow MDR$, if $MDR == 0$ then $PC \leftarrow PC + 1$
3	JUMP	Make the address given in the instruction the location of the next instruction to execute.	$PC \leftarrow IR_{11\text{–}0}$
4	HALT	Halt the machine	
5	BZAC	Conditional branch on AC equal to zero. If the AC has a value of zero, branch to the address given in the instruction.	if $AC == 0$ then $PC \leftarrow IR_{11\text{–}0}$
6	ADD	Add memory to AC	$MAR \leftarrow IR_{11\text{–}0}$ $MDR \leftarrow Mem[MAR]$ $AC \leftarrow AC + MDR$
7	SWP	Swap memory with the AC.	$MAR \leftarrow IR_{11\text{–}0}$ $MDR \leftarrow Mem[MAR]$ $AC \leftarrow MDR$,$MDR \leftarrow AC$ $Mem[MAR] \leftarrow MDR$

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 entire instruction, and then its mnemonic and address printed separately. Finally, the value of the accumulator (before executing the instruction) should be printed. All values are to be shown in hexadecimal, with the appropriate number of leading zeros to make all the trace lines line up (i.e. address values should be printed with 3 digits, and data values should be printed with 4 digits).

Here is an example:

```
9c4:  0101 =  LOAD 101   AC=0000
9c5:  1102 = STORE 102   AC=0300
9c6:  2102 =   ISZ 102   AC=0300
9c7:  3050 =  JUMP 050   AC=0300
050:  6103 =   ADD 103   AC=0300
051:  7104 =   SWP 104   AC=02ff
052:  50ff =  BZAC 0ff   AC=0002
053:  4000 =  HALT 000   AC=0002

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 "**simpISA**".

The object code file is named as an argument to the program, e.g., "`simpISA 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. The next section is just here to show you the format of the load files, so that you can read/create them yourself if you wish to do so.

3.3. Load File

Each line in the machine code file contains **word**-sized hexadecimal numbers separated by spaces. For all but the last line, the first number is the address where the third value in the line should be stored. The second number contains the number of values to be stored, i.e., the total number of values on the input line, minus two. The last line contains but one value, which is the starting address of the program.

Example

```
7 8 2 3 4c 5 d6a 7 10 11
3f 1 100
7
```

This loads memory as follows (numbers in hex):

mem[7]=2	mem[8]=3	mem[9]=4c
mem[a]=5	mem[b]=d6a	mem[c]=7
mem[d]=10	mem[e]=11	mem[3f]=100

(Start program at 7)

3.4. Halting

Besides after executing a HALT instruction, your CPU will need to stop when it attempts to execute past the end of the legal memory addresses (check this by checking PC overflow), or when an illegal op code is found in an instruction. When one of these happens, your simulator should print a blank line, then the message "MACHINE HALTED due to", plus the reason (with the following message):

Reason	**Message**
PC overflow	MACHINE HALTED due to PC overflow
illegal op code	MACHINE HALTED due to unknown op code
halt instruction	MACHINE HALTED due to halt instruction

Note regarding halting on **PC overflow**, memory location `fff` is a legal address, and an instruction at that location should be executed. The halt should happen if the CPU tries to advance, and execute a sequencial instruction after location `fff`.

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

5. Registering Your Account

As this is the first assignment this quarter, you must begin by registering your account with the grader account. Do this with the following command:

```
try grd-453 register /dev/null
```

6. 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 simpISA simpISA.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.

7. Supplied Software

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

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

Test program(s) are in ~csci453/pub/simpISA/*.obj

The output for the test program(s) are in ~csci453/pub/simpISA/*.out