CS472 Computer Architecture
FINAL LAB ASSIGNMENT
Due April 22, two weeks before final exam

Deductions for being late: 1st week - 2 % each day; 2nd week - 4% each day; Last day accepted: 3 days before exam! (This means 7 days before the final exam: 14% off, 6 days before exam: 18% off. 5 days before exam: 22% off. 4 days before exam: 26% off. 3 days before exam (hard deadline): 30% off.)
Submit via Blackboard.

****
* SHARE IDEAS, BUT NOT CODE. NOT *ONE* LINE! BASING YOUR WORK ON *
* SOMEONE ELSE'S (OR CHATGPT) WILL GET YOU AN "F" FOR THE COURSE. DON'T *
* USE OTHERS WORK "TO GET AN IDEA OF HOW TO DO IT" OR FOR ANY OTHER *
* REASON. NO EXCUSES. NO EXCEPTIONS. *DON'T DO IT!* *
****

OVERVIEW

This lab will be written in the language of your choice and will simulate how a pipelined datapath works. It **must** have a function (or procedure or method or whatever the term is for a code module in your language of choice) for each step in the pipeline: IF, ID, EX, MEM, and WB **with the function names shown below**. Your main program will have some initialization code and then will be one big loop, where each time through the loop is equivalent to one cycle in a pipeline. That loop will call those five functions, print out the appropriate information (the 32 registers and both READ and WRITE versions of the four pipeline registers) and then copy the WRITE version of the pipeline registers into the READ version for use the next cycle.

That is, your main program's loop will have the following sequence after initialization:

```
IF_stage();
ID_stage();
EX_stage();
MEM_stage();
WB_stage();
Print_out_everything();
Copy_write_to_read();
```

You must follow this order and include these exact function names for the five stages. Projects which go in the inverse order -- starting with WB, then MEM, then EX, then ID, then IF -- will get a zero!

Your program will use an array called **Main_Mem** to simulate a 1K Main Memory. It should be initialized as follows: Main_Mem[0]=0, Main_Mem[1]=1, …Main_Mem[0xFF]=0xFF, Main_Mem[0x100] = 0 and so on. (Note: 0xFF is the largest value that can be put in a byte, so after Main_Mem[0xFF]=0xFF, you start over with Main_Mem[0x100] = 0 and Main_Mem[0x101] = 1.) Your program will have simulated registers, which will just be an array of integers called **Regs**. For example, in C++ you could do: int Regs[32]; These registers are given initial values of x100 plus the register number except for register 0 which always has the value 0. (So $0=0, $1=0x101, $2=0x102, ... $10 = 0x10a, … $31=0x11f. (Reminder: register numbers are decimal.) **You MUST use the exact names Main_Mem and Regs for the arrays.**

DETAIL - PIPELINE REGISTERS

Your program will have four pipeline registers, which you will set up as structures (or something equivalent to structures in your language of choice) with a READ and a WRITE version for each. Each pipeline register's structure will be different and will be based on the information being passed from the previous stage (just like in the book and in the real world). You should model your structures on the pipeline registers in the book except that you do not need to include anything that is specific to branch instructions since you won't be implementing those. For control, you must use the control bits as we did in the detailed example. However, feel free to use as much space as makes the task easier for you (such as using an entire integer for a single one-bit control signal).

You will need a READ version of the pipeline registers and a WRITE version to make it so that you won't overwrite a pipeline register by one stage before the other stage has had a chance to read it. (Just like we do in our detailed example in class.) For example, on a given cycle the IF stage will write to the WRITE copy of the IFID pipeline register and the ID stage will read from the READ copy of that IFID pipeline register. This way, the information that the ID stage needs to read is not overwritten by what the IF writes during that cycle. At the end of each cycle, you'll copy the WRITE version into the READ version for use on the next cycle.

DETAIL - TEST INSTRUCTIONS

You will ONLY be implementing the following instructions: nop, add, sub, sb and lb. lb is a load byte that is just like a load word except that you only get one byte out of main memory instead of one word (which is four bytes or four memory addresses). This way you only have to get a single byte out of the MM array. The same thing holds for sb, which is a store byte and only writes to one byte in memory instead of the four that a sw does. (It writes the value of least significant byte in the register specified.) I simply have you using the byte versions of load/store so you don't have to concern yourself with accessing four addresses to handle the four bytes in a word. nop is a no-operation instruction. It is there to do nothing other than allow the other instructions to get through the pipeline. Do NOT add any other instructions. The opcodes are as follows: lb (x20) and sb (x28). The function codes are: add (x20), sub (x22) and nop (0).

The test "program" that will flow through your pipeline will be:

```
0xa1020000
0x810AFFFC
0x00831820
0x01263820
0x01224820
0x81180000
0x81510010
0x00624022
0x00000000      # This is a nop, used just to allow the "real" instructions finish in the pipeline
0x00000000      # This is a nop, used just to allow the "real" instructions finish in the pipeline
0x00000000      # This is a nop, used just to allow the "real" instructions finish in the pipeline
0x00000000      # This is a nop, used just to allow the "real" instructions finish in the pipeline
```

HOW TO PROCEED

You should be able to reuse most of your first project to take an instruction and figure out what it is supposed to do (i.e., decode it). Don't reinvent the wheel. Use that first project, but it all must be incorporated into a single program. (You can't just run the first program, take the output from that and run that as input to a second program. It must all be contained in a single executable.)

Put these instructions into an array, something like "int InstructionCache[12] and maintain an index into that array just like a program counter. Note that there are no branch operations so you will just start with the first instruction and then continue all the way down until the last nop. The instructions that do something are followed by nops so the last few instructions can finish up in the pipeline (i.e., while the last non-nop instruction makes it way through the WB stage).

You will be simulating a five-stage pipeline comparable to the one in the book. Each stage will pass information into the next stage's pipeline register. The five stages will be as follows:

- IF You will fetch the next instruction out of the Instruction Cache. Put it in the WRITE version of the IF/ID pipeline register.

- ID Here you'll read an instruction from the READ version of IF/ID pipeline register, do the decoding and register fetching and write the values to the WRITE version of the ID/EX pipeline register.

- EX Here you'll perform the requested instruction on the specific operands you read out of the READ version of the IDEX pipeline register and then write the appropriate values to the WRITE version of the EX/MEM pipeline register. For example, an “add” operation will take the two operands out of the ID/EX pipeline register and add them together like this:

  EX_MEM_WRITE.ALU_Result = ID_EX_READ.Reg_Val1 + ID_EX_READ.Reg_Val2;

- MEM If the instruction is a lb, then use the address you calculated in the EX stage as an index into your Main Memory array and get the value that is there. Otherwise, just pass information from the READ version of the EX_MEM pipeline register to the WRITE version of MEM_WB.

- WB Write to the registers based on information you read out of the READ version of MEM_WB.

Rather than writing code to specially handle the first few cycles (during which the pipeline gets filled with the instructions), you can initialize your pipeline to have all NOPs in it before you begin with the first real instruction. If the control indicates a NOP, then nothing will happen for that instruction even if there is junk in all the other values for the pipeline register. This way, when you start accepting the actual instructions, you already have NOPs in the other stages so you can just treat the very first cycles as the same as all the others.

No handling of data hazards is expected.

WHAT I WANT FOR OUTPUT

After each cycle I want you to **print out the Regs values (not the physical registers on your computer, the Regs array that you use to simulate the physical registers) and both the READ and WRITE versions of all four pipeline registers (giving me labels so I know what you're showing**). Do this BEFORE you copy the WRITE version into the READ version at the very end of the cycle. Show ALL the detail in your pipeline registers. **IF YOU DON'T SHOW ALL THIS INFORMATION, DON'T EXPECT ANYTHING CLOSE TO FULL CREDIT**. Since there are eight instructions followed by the NOPs, you should be showing me 12 cycles of your pipeline.

Feel free to waste as much space as you want within the pipeline register. For example, you do NOT have to compress all the control bits into a single variable that you then examine bit-by-bit. Instead, you can just define a boolean variable (or just an int that will have either 1 or 0) for every control signal.

You don't need to include things in the pipeline registers that aren't necessary to accomplish your task (e.g., the Program Counter or the Branch Target Address since you aren't implementing branches). **Other than leaving out details regarding branch instructions, however, stay true to how pipelines are supposed to work. It is very important that your project behaves like a pipeline as far as the restrictions of simulation realistically allow.**

Use all that output detail of the pipeline registers to your debugging advantage.