Electrical Engineering
& Computer Science
University of Missouri

ECE 4270/7270: Computer Organization, Spring 2024
LAB 3: Pipelined RISC-V Simulator Supporting Data Hazard Detection and Data Forwarding

Scope

In this lab assignment, you will extend the instruction-level RISC-V simulator that you have developed in Lab 1. The RISC-V simulator that you have developed in Lab 1 was unpipelined, namely, it could take one instruction at a time and finish the execution within a (long) single cycle. In the pipelined version, there are five stages where each stage takes one cycle (where the cycle time is about ~5x smaller than the unpipelined version). As you may recall, these five pipeline stages are:
- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execute (EX)
- Memory Access (MEM)
- Writeback (WB)

You will work on ALU and Load/Store instructions as a starting point, and exclude branch and jump instructions from the scope of this lab assignment. You should build your pipelined instruction-level RISC-V simulator assuming that you do not have any branches and jumps in your code (we exclude them for now. However, if you finish your lab earlier, you are welcome to add support for branches and jumps). The operations performed within each of these five stages are explained below.

1. Instruction Fetch (IF):

IR <= Mem[PC]
PC <= PC + 4

The instruction is fetched from memory into the instruction register (IR) by using the current program counter (PC). The PC is then incremented by 4 to address the next instruction. IR is used to hold the instruction (that is 32-bit) that will be needed in subsequent cycle during the instruction decode stage.

2. Instruction Decode (ID):

A <= REGS[rs]
B <= REGS[rt]
imm <= sign-extended immediate field of IR

In this stage, the instruction is decoded (i.e., opcode and operands are extracted), and the content of the register file is read (rs and rt are the register specifiers that indicate which registers to read from). The values read from register file are placed into two temporary registers called A and B. The values stored in A and B will be used in upcoming cycles by other stages (e.g., EX, or MEM). The lower 16 bits of the IR are sign-extended to 32-bit and stored in temporary register called imm. The value stored in imm register will be used in the next stage (i.e., EX).

3. Execution (EX)

In this stage, we have an ALU that operates on the operands that were read in the previous stage. We can perform one of three functions depending on the instruction type.

i) Memory Reference (load/store):
ALUOutput <= A + imm

ALU adds two operands to form the effective address and stores the result into register called ALUOutput.

ii) Register-register Operation
ALUOutput <= A op B

ALU performs the operation specified by the instruction on the values stored in temporary registers A and B and places the result into ALUOutput.

iii) Register-Immediate Operation
ALUOutput <= A op imm

ALU performs the operation specified by the instruction on the value stored in temporary register A and value in register imm and places the result into ALUOutput.

4. Memory Access (MEM):

for load: LMD <= MEM[ALUOutput]
for store: MEM[ALUOutput] <= B

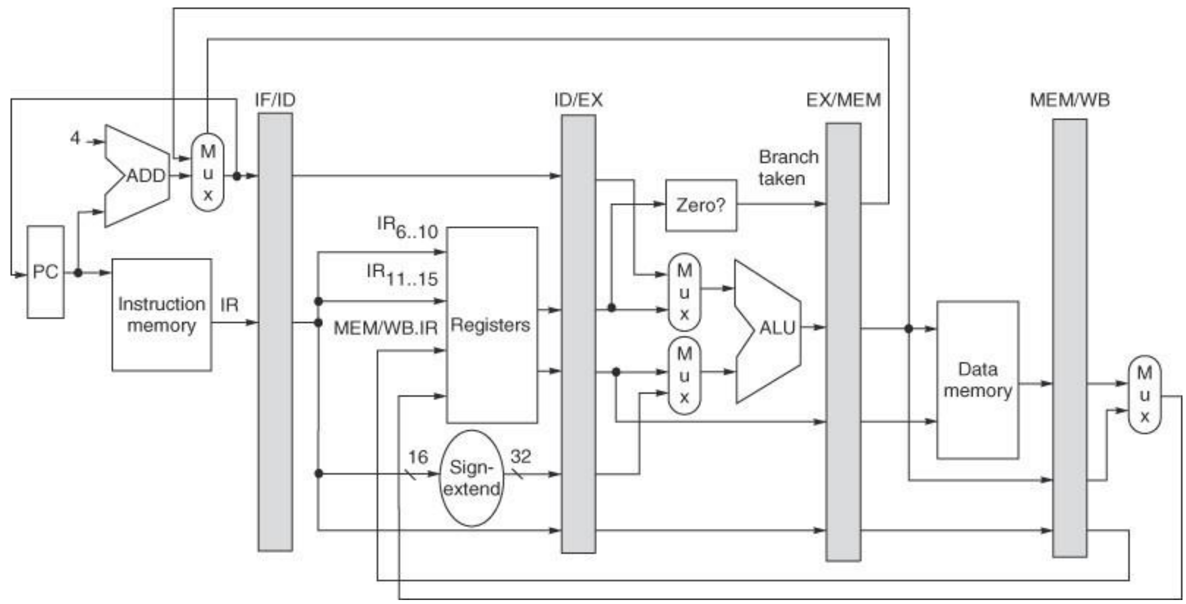
If the instruction is load, data is read from memory and stored in load memory data (LMD) register. If it is store, then the value stored in register B is written into memory. The address of memory to be accessed is the value computed in the previous stage and stored in ALUOutput register.

5. Writeback (WB)

for register-register instruction: REGS[rd] <= ALUOutput
for register-immediate instruction: REGS[rt] <= ALUOutput
for load instruction: REGS[rt] <= LMD

In this stage, we write the result back into the destination register in register file. The result may come from LMD or ALUOutput depending on the instruction.

Notice that, the temporary registers that we used (i.e. IR, A, B, imm, ALUOpt, LMD) will be overwritten by the next instruction during the next cycle. Since these values have to be passed from one pipeline stage to next, we utilize pipeline registers to do so. The pipeline registers carry both data and control signals from one pipeline stage to another. Any value needed on a later pipeline stage should be placed in pipeline register and forwarded from one stage to another until it is no longer needed (i.e., not used in the upcoming stages). Below is the pipelined datapath that you are going to build. For now, you can omit the branch related links and mux, and assume that PC is incremented by 4 at every cycle.



As you can see, there are 4 pipeline registers and each of them has fields associated with proper temporary registers that we mentioned above (i.e. IR, A, B, imm, ALUOpt, LMD). The pipeline register called IF/ID has the following fields:

IF/ID.IR
IF/ID.PC

On the other hand, pipeline register called ID/EX has the following fields:

ID/EX.IR
ID/EX.A
ID/EX.B
ID/EX.imm

Similarly, pipeline register called EX/MEM has the following fields:

EX/MEM.IR
EX/MEM.A
EX/MEM.B
EX/MEM.ALUOutput

And finally, pipeline register called MEM/WB has the following fields:

MEM/WB.IR
MEM/WB.ALUOutput
MEM/WB.LMD
The events on pipeline stages and which values are forwarded from one stage to another through the pipeline registers are summarized below.

Stage	Operation	
IF	IF/ID.IR <= MEM [PC] IF/ID.PC <= PC+4	
ID	ID/EX.IR <= IF/ID.IR ID/EX.A <= REGS[IF/ID.IR[rs]] ID/EX.B <= REGS[IF/ID.IR[rt]] ID/EX.imm <= sign-extend(IF/ID.IR[imm. Field])	
	ALU Instruction	Load/Store Instruction
EX	EX/MEM.IR <= ID/EX.IR EX/MEM.ALUOpt <= ID/EX.A op ID/EX.B (for reg-reg) or EX/MEM.ALUOpt <= ID/EX.A op ID/EX.imm (for reg-imm)	EX/MEM.IR <= ID/EX.IR EX/MEM.ALUOpt <= ID/EX.A + ID/EX.imm EX/MEM.B <= ID/EX.B
MEM	MEM/WB.IR <= EX/MEM.IR MEM/WB.ALUOutput <= EX/MEM.ALUOutput	MEM/WB.IR <= EX/MEM.IR MEM/WB.LMD <= MEM[EX/MEM.ALUOutput] (for load) or MEM[EX/MEM.ALUOutput] <= EX/MEM.B (for store)
WB	REGS[MEM/WB.IR[rd]] <= MEM/WB.ALUOutput (for reg-reg) or REGS[MEM/WB.IR[rt]] <= MEM/WB.ALUOutput (for reg-imm)	REGS[MEM/WB.IR[rt]] <= MEM/WB.LMD (for load)

Implementation

You are given a skeleton of the pipelined RISC-V simulator (you can download it through the canvas), similar to one provided in Lab 1. However, instead of handle_instruction(), you now have handle_pipeline() function. This function is called every cycle. You will have more than one instruction on the fly (on different pipeline stages) in a given cycle. Within handle_pipeline() function, there are five functions corresponding to each pipeline stage. handle_pipeline() function looks like as follows:

```
void handle_instruction(){
        WB();
        MEM();
        EX();
        ID();
        IF();
}
```

As their names suggest WB(), MEM(), EX(), ID() and IF() functions should implement what is needed for writeback, memory access, execution, instruction decode, and instruction fetch stage of the pipeline, respectively. You are responsible to implement these functions. Note that, each stage should perform what it has to perform, nothing more, nothing less. While you implement these functions, you need to maintain the pipeline registers that we mentioned above. For this purpose, you are given CPU_Pipeline_Reg data structure in mu-RISC-V.h file. There are four instances created already for the pipeline registers and named as ID_IF, IF_EX, EX_MEM, MEM_WB (see mu-RISC-V.h for details).

These functions appear in reverse order within the code because you should use the outcome of particular stage as input to next stage in the next cycle, not in the current cycle. As an example, if we had IF() first, then ID() what would happen is the following. IF() would fetch new instruction and update its pipeline register IF/ID. Then, ID() function would try to read the content of the IF/ID fields which have been updated very recently (within the same cycle).

Notice that this is not the intended behavior; ID() function should be able to read IF/ID fields that were updated in the previous cycle. To maintain this correct timing of consumer/producer relationship, the functions for pipeline stages appear on reverse order in the code. To elaborate on that more, for the first 4 cycles WB() function should do nothing since no instruction have reached that stage until the fifth cycle. Similarly, MEM() function should do nothing for the very first 3 cycles since no instruction have reached MEM stage until the fourth cycle. Similarly, EX stage should do nothing for the very first 2 cycles, and ID stage it for the very first cycle. However, if you change the order of these functions, you can see that all stages become active in the very first cycle: IF() would fill ID/IF that would be consumed by ID(), then ID() would fill ID/EX that would be consumed by EX(), and EX() would fill EX/MEM that would be consumed by MEM(), and MEM() would fill MEM/WB that would be consumed by WB(). Notice that all these stages would become active in the very first cycle. This is not correct behavior. The output of a particular pipeline stage should be consumed by the following pipeline stage in the next cycle. In short, do not change the order of functions that appear in handle_pipeline(); this order will make your coding a lot easier.

You should also implement show_pipeline() function that print out the content of pipeline registers for a given cycle. It will be used for debugging purposes. The outcome of this function should be similar to the following.

```
Current PC:    value
IF/ID.IR       value instruction( e.g. add $1, $2, $3)
IF/ID.PC       value //notice that it contains the next PC

ID/EX.IR       value instruction
ID/EX.A        value
ID/EX.B        value
ID/EX.imm      value

EX/MEM.IR      value
EX/MEM.A       value
EX/MEM.B       value
EX/MEM.ALUOutput         value

MEM/WB.IR value
MEM/WB.ALUOutput         value
MEM/WB.LMD        value
```

Along with pipeline registers, you may still need to maintain CPU_State (i.e. CURRENT_STATE, NEXT_STATE) since CURRENT_STATE has the REGS field that contains the values stored in the registers (you need to read them in ID stage). Similarly, NEXT_STATE has REGS field where you should store the result of the instruction back into the corresponding register (in WB stage).

Once again, in the scope of this lab assignment, you do not work with branch and jump instructions. So the test files provided do not contain branch and jump instructions.

Grading Rubric

Code: Pipelined RISC-V simulator (75)
Report: 25 points

Pipelined RISC-V simulator (75 points):

To get a full credit for the code, your pipelined RISC-V simulator should work correctly (i.e., the instructions should be fetched, decoded, executed correctly. The data being forwarded among pipeline registers should be correct. Your code will be debugged step by step to see if the transitions between pipeline stages are handled correctly).

Lab report (25 points):

Your report should give details about the work distribution within the group (who did what), milestones in your work and your implementation decisions (why did you choose the way you did it, and/or how did you do that). Explain any difficulty you observed, any optimizations you have made.

Submission

Please pay attention to this guideline here: You should submit the lab report along with the pipelined RISC-V simulator code you developed (provide makefile, as well). Please, generate a pdf file for your report and name it lab3_report_groupX.pdf, then place it into the folder called lab3_groupX (where X is your group number). The folder lab3_groupX should also contain the src/ folder that contains the code for pipelined RISC-V simulator (i.e. mu-RISC-V.h, mu-RISC-V.c and Makefile). Then, please compress the lab3_groupX folder as lab3_groupX.tar.gz and submit it through Canvas.