Machine Problem 2: Pipelined Processor

2nd sem 2021-22

1 Introduction

We will extend the single-cycle processor implemented in the previous machine problem by employing concurrent execution through pipelining. Pipelining will allow multiple instructions to execute at the same time in the processor, increasing its performance by increasing its throughput.

2 Pipeline Description

The processor datapath will be divided into five pipeline stages. Each pipeline stage's tasks should be completed in one clock cycle, resulting in a total instruction execution latency of 5 clock cycles. Figure 1 shows a generic block diagram of the five-stage pipeline. Individual pipeline stages are separated by a pipeline register (highlighted in figure 1). The description of each stage is discussed below. Only the flow of data from one stage to another is described - specific control signals, especially their timing, is left for you to design and implement.

2.1 Instruction Fetch (IF)

The IF stage is responsible for fetching the instruction from Instruction Memory, which is external to the processor, and incrementing the Program Counter (PC) value. The instruction fetched from the processor (signal *inst*) is then sent to the IF/ID pipeline register, where it is saved and used in the next clock cycle by the ID stage.

2.2 Instruction Decode (ID)

The ID stage takes the saved instruction (signal *inst_IFID*) from the IF/ID pipeline register and performs several actions on it.

The first is to fetch necessary data from source registers in the register file. For arithmetic R-type instructions, this would be both operands. For loads and stores, this would be the base address of the effective address of the data memory location in question. Additionally for stores, the register operand to be stored is also fetched. For I-type arithmetic instructions, this would be the register operand. For branches, this would be the

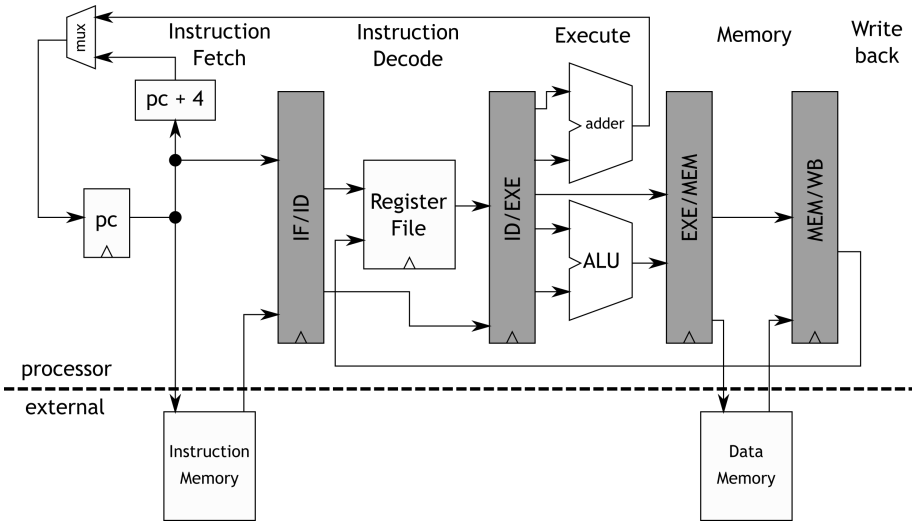

Figure 1: Generic five-stage pipelined datapath

operands we would compare. For JALR, this would be the register that contains the base address of the target program address.

The second action would be to produce immediate operands for instructions which need them. For load and stores, this would be the byte offset of the effective address. For I-type arithmetic instructions, this would be the immediate operand. For branches and jumps, this would be the byte offset of the branch target address.

The operands fetched or produced are then sent to the ID/EXE pipeline register, where they are saved and used in the next clock cycle by the EXE stage.

2.3 Execute (EXE)

The EXE stage is where the ALU is located. It takes the ALU operands saved in the ID/EXE pipeline register forward it to the ALU for calculation. For arithmetic instructions, the result is sent to the EXE/MEM pipeline register where it is saved and used in the next clock cycle by the MEM stage. For loads and stores, the ALU result would be the effective data memory address. This is also forwarded to the EXE/MEM pipeline register and saved for use in the next stage.

For branch instructions, the operands saved in the ID/EXE pipeline register are compared by the ALU. The branch target address is also computed from the associated PC value and immediate operand saved in the ID/EXE pipeline register. Branch target computation is done using a separate adder. Take note that the associated PC value is the PC of the branch instruction itself. The value of the actual PC would already be different from the associated PC value, so it would be necessary to save this value and use it once the branch instruction enters the EXE stage (hint: this can be done using the pipeline registers). The result of the ALU comparison along with the type of branch instruction are used to determine if the branch target address will be used to update the PC value.

Jump instructions use the EXE stage for computing the jump target address and using it to update the PC value. JAL will use the associated PC value (address of the JAL instruction) and the immediate operand saved in the ID/EXE pipeline register for this computation. Alternatively, JALR will use the base register value and immediate operand saved in the ID/EXE pipeline register.

2.4 Memory (MEM)

Data memory accesses are performed in the MEM stage. The effective address computed by the EXE stage is fetched from the EXE/MEM pipeline register and forwarded to external data memory for loads and stores. Along with this address, data to be stored which was fetched during the ID stage will also be forwarded to data memory at this stage. A corresponding data memory write enable will be generated for store instructions as well.

Data fetched from memory for load instructions will be sent to the MEM/WB pipeline register and saved for later use by the WB stage. Results for arithmetic instructions previously saved to the EXE/MEM pipeline registers are also forwarded to the MEM/WB pipeline register in the same manner.

Jumps need to eventually write the address of the instruction after the jump instruction (i.e. PC + 4 where PC is the address of the jump instruction). This value passes through the same process as data to be saved to the register file during the WB stage, sending it to the MEM/WB pipeline register as well.

2.5 Write-back (WB)

Data (in the case of loads and arithmetic instructions) and addresses (in the case of jumps) that needed to be written to the register file is fetched from the MEM/WB pipeline register and sent to the write port of the register file. Along with this data, a corresponding write enable to the register file is also sent during the WB stage.

3 Top-level module and interface

To ensure that your design adheres to the data flow constraints mentioned in the previous section, we will be routing internal pipeline stage signals to the output of the processor for observation purposes. These signals are mostly connected to pipeline register outputs.

- ALUop1 - ALU operand 1 (64 bits). Output of ID/EXE pipeline register.
- ALUop2 - ALU operand 2 (64 bits). Output of ID/EXE pipeline register.
- ALUres - ALU result (64 bits) after being sampled by pipeline register. Output of EXE/MEM pipeline register.

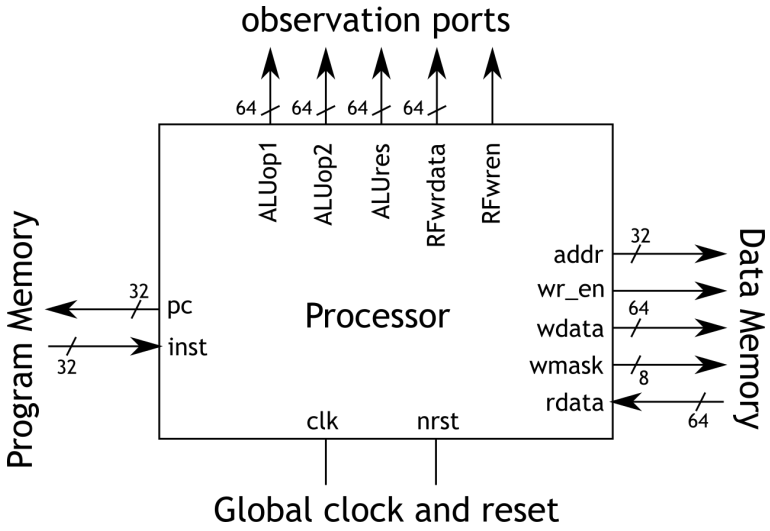

Figure 2: Top-level module ports

- RFwren - Register file write enable.
- Rfwrdata - Register file write data (64 bits).

Figure 2 shows the top-level ports of the processor. All unconnected ports will be used for observation in the testbench. To illustrate the timing constraints of processes within the pipelined processor, several timing diagrams depicting the values of the signals discussed above will be shown for different classes of instructions in the next subsections . Other internal signals not included in the list above will also be shown for clarity.

3.1 Loads

Figure 3 shows an *ld* instruction at address 100 being executed. The instruction is fetched during the clock cycle where it is in the IF stage. In the next clock cycle, the instruction is now in the ID stage and its saved value in the IF/ID register (inst_IFID) is used for generating the base and offset values of the load address. In the third clock cycle the instruction would be in the EXE stage, where the saved values of the base and offset (ALUop1 and ALUop2) in the ID/EXE pipeline registers are sent to the ALU to compute the effective address. One clock cycle after, this effective address now resides in the ALUres signal output of the EXE/MEM pipeline register and is sent to data memory to fetch the required value (val). The value is saved to the MEM/WB pipeline register, where it is used in the fifth clock cycle during the WB stage to write the value back to the register file. Assuming the destination register is x10, its value is only updated in the clock cycle after the *ld* instruction was in the WB stage.

3.2 Stores

The first three stages of stores are similar to loads. In the MEM stage, the effective address is still forwarded to data memory. However, instead of reading the data memory the value to be stored is written to it. This is shown in figure 4 by the wr_en being asserted during the MEM stage.

3.3 Arithmetic

Figure 5 shows the execution of an *add* instruction. The IF and ID stages are similar to loads, except that the operands op1 and op2 are generated. These are saved in the ID/EXE pipeline registers and are used as inputs to the ALU (ALUop1 and ALUop2) in the EXE stage. The sum is saved in the EXE/MEM pipeline register. During the MEM stage, this saved result (ALUres) is simply forwarded from the EXE/MEM to the MEM/WB pipeline register. Finally, the sum is written back to the register file during the WB stage, as depicted by the assertion of RFwrdata. Similar to loads, if the destination register of the *add* instruction is x10, its value is only updated in the clock cycle after *add* was in the WB stage.

3.4 Branches

The timing of taken branches is shown in figure 6. A *bne* instruction whose branch target address is 500 is executed. The operands are 13 and 14, respectively. This results in the branch being taken. Since the ALU

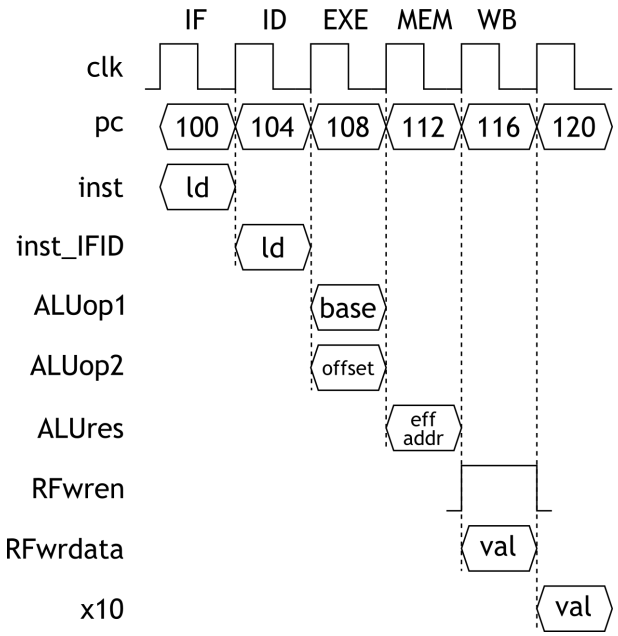

Figure 3: Timing diagram for loads

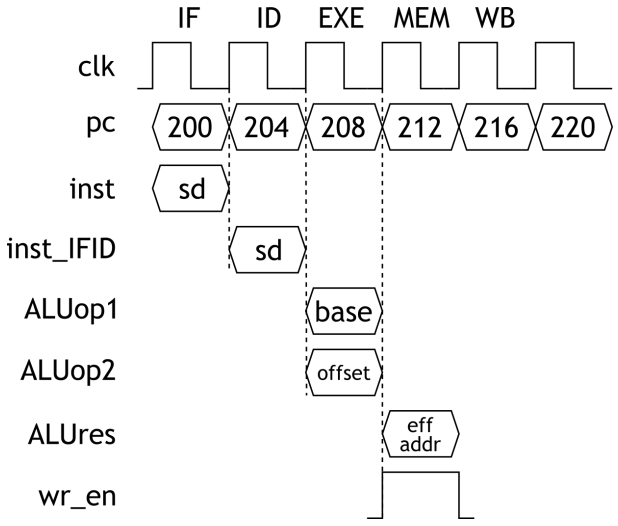

Figure 4: Timing diagram for stores

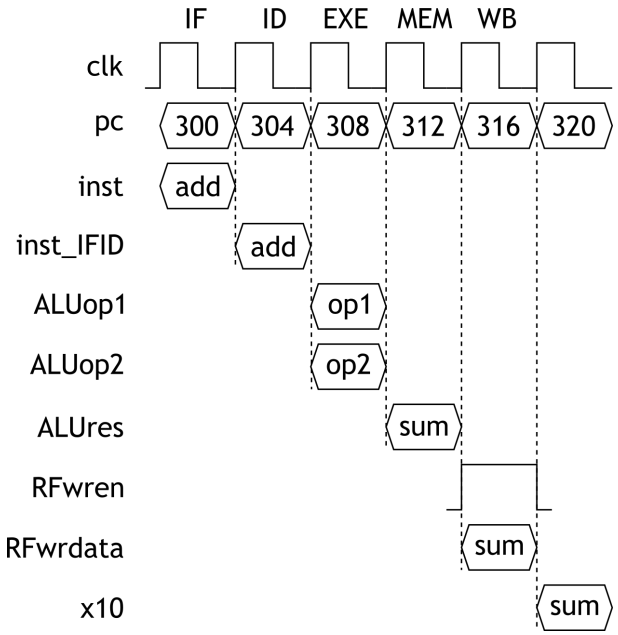

Figure 5: Timing diagram for arithmetic instructions

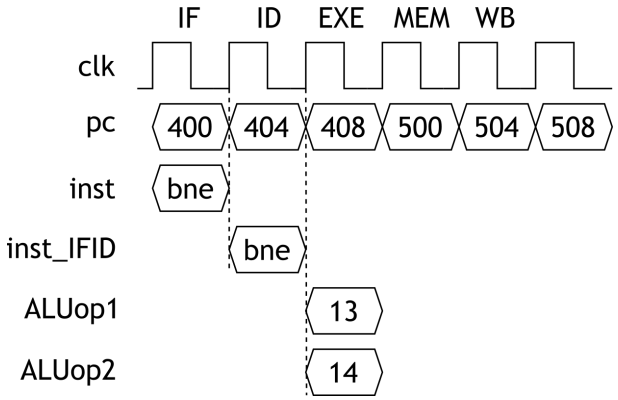

Figure 6: Timing diagram for branches

performs the comparison during the EXE stage, the pc is updated to the branch target address in the following clock cycle (when the branch instruction is in the MEM stage). In cases where the branch is not taken, the pc is incremented normally. Branches also don't do anything during the MEM and WB stages.

3.5 Jumps

Figure 7 shows the execution of a *jal* instruction whose target address is 700. The pc is updated at the same time as with branches, when the instruction is at the MEM stage. This is done because for the other jump instruction, *jalr*, the target address must be computed from a source register and an immediate offset. This computation is achieved using the ALU. Unlike branches, jumps save the value of address of the instruction after the jump (in this case, 604) to a destination register (in this case x10) during the WB stage.

4 Hazards

Overlapped execution of instructions results in possible hazards that may occur within the pipelined processor.

Data hazards result from the late writes made to the destination register. As seen in figures 3, 5, and 7, writes to the register file are made only in the final stage of the pipeline. Therefore, any instructions that use the written result as operands will fetch incorrect (non-updated) values if they perform the operand fetch before the write is committed. Operand fetches are done during the ID stage, while writes to the register file only reflect after the WB stage. Thus, a dependent instruction should be in the ID stage no earlier than the clock cycle after the WB of the instruction that produces the operand. In the example shown in figure 3, a instruction

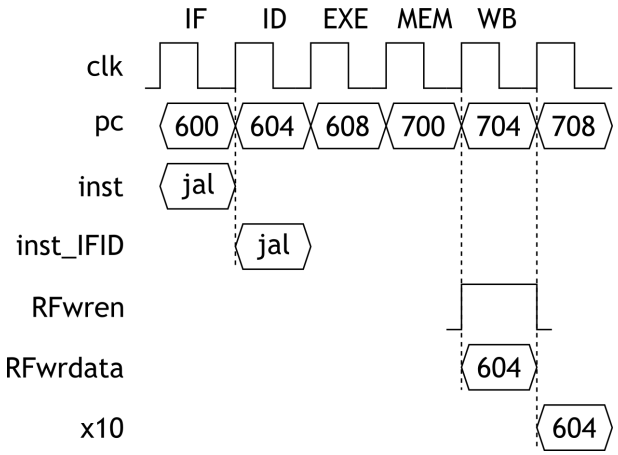

Figure 7: Timing diagram for jumps

dependent on *ld* (address 100) should be placed in instruction memory address 116 onwards to prevent data hazards from occuring.

In this machine problem, correct handling of data hazards is not required (this will be the topic of the next machine problem). For purposes of testing, always assume that the ordering of instructions in any given program will not result in a data hazard (which can be achieved by following the rule previously stated).

Control hazards occur whenever the processor deviates from a linear pattern of fetching instructions from memory. This occurs in taken branches and jumps since the target address only reflects in the pc during the EXE stage. At the time the branch or jump instruction has determined to update the pc to the target address, two other instructions after the branch/jump have already been fetched. In figure 6, these are instructions at addresses 404 and 408. If the branch/jump is taken, these instructions should not have been fetched in the first place, so any effect they have on the processor state (i.e. pc, memory and register file contents) should not be committed.

In this machine problem, correct handling of control hazards is also not required. For purposes of testing, you can place two *nop* instructions after each branch and jump instruction to preserve processor state even if they are fetched unnecessarily.

5 Grading

Grading will be based on the functionality of instructions. Criteria for grading this machine problem will be as follows:

- Loads and Stores - 40 points
- Arithmetic Instructions - 30 points
- Branches - 15 points
- Jumps - 15 points

Full points for a category will be given only if the synthesized model works for that category. Half points will be given instead if the RTL model functions correctly. Testing for all non load/store categories are dependent on the functionality of ld and sd, so make sure that your ld and sd instructions work perfectly.

Deadline for submission will be on May 21, 2022, through an UVLE submission bin. Submissions beyond the deadline will be allowed until the end of the semester but will have a deduction of 10 points.

Upload all source files in an archived file. Do not upload the entire Vivado project directory. Allowed archive formats are: zip, tar, tar.gz. Failure to comply to these rules will automatically merit a 20 point deduction.