ECE 486/586: Computer Architecture

Spring 2020 TERM PROJECT

The project will be carried out in groups of students. Final project report is due on Thursday, June 11 11:59:59 PM.

Project Objective:

The goal of the project is to model a simplified version of the MIPS ISA called MIPS-lite and the in-order 5-stage pipeline to be discussed in class. You will write your own simulator in a high-level language of your choice (such as C, C++, JAVA, Verilog, VHDL, etc.). The simulator will take the provided memory image as its input. It will implement two key features: (i) A functional simulator which simulates the MIPS-lite ISA and captures the impact of instruction execution on machine state, (ii) A timing simulator which models the timing details for the 5-stage pipeline. The output of the simulator will include: (a) A breakdown of instruction frequencies in the instruction trace into different instruction types, (b) Final machine state (register values, memory contents etc.), (c) Execution time (in cycles) of the instruction trace on the 5-stage pipeline.

Note that there is no need to model the internal details of processor hardware, such as pipeline latches or wires. The goal is NOT to build a simulator that is hardware accurate for each bit of the pipeline. You only need to make modifications that will capture the changes to machine state and count the total clock cycles a program takes to execute. You will have to **visualize** the 5 stage pipeline and the instructions in every stage, and then program your simulator with that in mind. For example, when there is a data hazard, you will ascertain how many stall cycles would this kind of hazard cause in a real processor pipeline. Then you have to account for the impact of these stall cycles on the execution time.

Project Details

Please read the following details carefully and follow them, when implementing your simulator:

1. **Instruction Set:** Your simulator should be capable of simulating the following instructions:
   a. Arithmetic Instructions:
      i. ADD Rd Rs Rt (Add the contents of registers Rs and Rt, transfer the result to register Rd). Opcode: 000000
      ii. ADDI Rt Rs Imm (Add the contents of register Rs to the immediate value "Imm", transfer the result to register Rt). Opcode: 000001
      iii. SUB Rd Rs Rt (Subtract the contents of register Rt from Rs, transfer the result to register Rd). Opcode: 000010
      iv. SUBI Rt Rs Imm (Subtract the immediate value "Imm" from the contents of register Rs, transfer the result to register Rt). Opcode: 000011
      v. MUL Rd Rs Rt (Multiply the contents of registers Rs and Rt, transfer the result to register Rd). Opcode: 000100
      vi. MULI Rt Rs Imm (Multiply the contents of registers Rs with the immediate value "Imm", transfer the result to register Rt). Opcode: 000101

   b. Logical Instructions
      i. OR Rd Rs Rt (Take a bitwise OR of the contents of registers Rs and Rt, transfer the result to register Rd). Opcode: 000110
      ii. ORI Rt Rs Imm (Take a bitwise OR of the contents of registers Rs and the immediate value "Imm", transfer the result to register Rt). Opcode: 000111

   iii. AND Rd Rs Rt (Take a bitwise AND of the contents of registers Rs and Rt, transfer the result to register Rd). Opcode: 001000
   iv. ANDI Rt Rs Imm (Take a bitwise AND of the contents of registers Rs and the immediate value “Imm”, transfer the result to register Rt). Opcode: 001001
   v. XOR Rd Rs Rt (Take a bitwise XOR of the contents of registers Rs and Rt, transfer the result to register Rd). Opcode: 001010
   vi. XORI Rt Rs Imm (Take a bitwise XOR of the contents of registers Rs and the immediate value “Imm”, transfer the result to register Rt). Opcode: 001011

c. Memory Access Instructions
   i. LDW Rt Rs Imm (Add the contents of Rs and the immediate value “Imm” to generate the effective address “A”, load the contents (32-bits) of the memory location at address “A” into register Rt). Opcode: 001100
   ii. STW Rt Rs Imm (Add the contents of Rs and the immediate value “Imm” to generate the effective address “A”, store the contents of register Rt (32-bits) at the memory address “A”). Opcode: 001101

d. Control Flow Instructions
   i. BZ Rs x (If the contents of register Rs are zero, then branch to the “x”th instruction from the current instruction). Opcode: 001110
   ii. BEQ Rs Rt x (Compare the contents of registers Rs and Rt. If they are equal, then branch to the “x”th instruction from the current instruction). Opcode: 001111
   iii. JR Rs (Load the PC [program counter] with the contents of register Rs. Jump to the new PC). Opcode: 010000
   iv. HALT (Stop executing the program). Opcode: 010001

Important Note: For arithmetic computations (such as addition, subtraction, multiplication) carried out in the above instructions, the operands should be treated as **signed** integers and the computations should be carried out in 2’s complement binary arithmetic. Specifically, any register contents or immediate values used in arithmetic instructions, effective address calculations or branch target computations should be treated as signed integers. For example, an immediate value of $(1000000000000001)_2$ is equal to $(-32767)_{10}$ and not $(+32769)_{10}$

2. Instruction Formats: The instructions mentioned above are specified in one of the following two instruction formats:

(i) R-type format:

6	5	5	5	11
Opcode	Rs	Rt	Rd	Unused

This format is used by the following instructions: ADD, SUB, MUL, OR, AND, XOR

(ii) **I-type format:**

6	5	5	16
Opcode	Rs	Rt	Immediate

This format is used by all the remaining instructions: ADDI, SUBI, MULI, ORI, ANDI, XORI, LDW, STW, BZ, BEQ, JR, HALT. Note that some of the instructions (BZ, JR and HALT) do not use all the fields in the I-type format.

3. **Memory Image:** The simulator will take a memory image as its input. This memory image represents the initial state of the simulated program. The image will be provided to you as a text file which contains the initial contents of both the code segment (for instructions) and the data segment (for data) in a shared address space. Each line in the text file represents one word (4 bytes) of memory shown in the **hexadecimal** (base-16) format. The addresses start at "0", from the first line in the image and then increase by "4" at every next line. The addresses will not be specified in the memory image and are implicitly obvious from the line number in the image file. For example, if you want to read the contents of memory address $(20)_{10}$, then those contents can be found on the sixth line in the memory image. Also note that all the data in the trace file is represented in the "Big Endian" format.

   Example: Let us assume that the first line in the memory image is as follows: 00853000. What instruction does this line represent?
   Answer: As the image is in hexadecimal (base-16) format, you will first need to convert the information to binary (base-2) representation so that it is easier to decode. The binary representation for this line is as follows: 00000000100001010011000000000000. Since the opcode is in the 6 most significant bits, we can infer that opcode = 000000, which represents the ADD instruction (R-type format). By analyzing other instruction fields, we can conclude that Rs = $(00100)_2 = (4)_{10}$, Rt = $(00101)_2 = (5)_{10}$ and Rd = $(00110)_2 = (6)_{10}$. Therefore this line represents the instruction: ADD R6, R4, R5

   Note: Even though MIPS-lite uses 32-bit addresses, it is impractical to provide a memory image for the entire 2^{32} bytes of memory. Therefore, the provided images will be limited to 4kB memory state (1024 lines). The instructor will ensure that none of the simulated programs accesses any data outside the 4kB. If your simulator detects that the processor is attempting to access data outside the 4kB range, it should be treated as a sign that the simulator is not operating correctly.

4. **Functional Simulator:** You'll need to develop a functional simulator which captures the effect of running the simulated program on the simulated machine state. The machine state includes the following three components: (i) Program counter (PC), (ii) General purpose registers (R1 to R31), (iii) Memory. The initial state of the memory is contained in the memory image. You will assume that the PC and all the general purpose registers are initialized to "0". As you simulate each instruction in the program, you will keep track of the machine state (PC, GPRs and memory) and make necessary changes to the state based on the impact of the simulated instruction. For example, if an instruction writes a new value to register R6, then you will update the register R6 with the instruction result. Each instruction will also update the PC, which will in turn cause a new instruction to be fetched and simulated. You will continue the simulation until you encounter a "HALT" instruction.

   The memory images needed for your final project report will be provided two weeks before the project report deadline. To test your simulator in the meantime, you can write your own test images (short segments of code and data) which can be analyzed (by hand) to determine expected outcomes without running any simulations. You can then test your simulator by simulating these test images and comparing the simulator output with the expected outcome.

5. **Timing Simulator:** You'll need to develop a pipeline timing model which captures the flow of instructions through the 5-stage pipeline. Your model should be capable of detecting pipeline hazards and then accounting for any resultant stall cycles or flushing of instructions from the pipeline.
   The throughput of instructions through the pipeline should be 1 instruction per clock cycle, except when any of the following events occur: (i) RAW hazard, (ii) Taken branch.

i. <u>RAW hazards:</u> You will have to account for stall cycles due to RAW hazards. You will need a scheme in which you can match the source operands of the recently decoded instruction with the destination operands of the other instructions down the pipe. If there is a dependence, you will add stall cycles according to how many stages apart the instructions are. Note that these stall cycles would be different depending on whether forwarding (bypassing) is present or not. A good idea would be to enlist all possible hazard combinations and calculate the stall cycles for each. Assume all instructions take 5 cycles to finish. So there are no WAR/WAW hazards in this instruction set and pipeline combination. Also assume that when a register read and register write of the same register occur in the same clock cycle, write data is written in the first half cycle and the new contents of the register can be read in the second half of the cycle. Keep in mind that if you assume no forwarding you have to time instructions one way and if you assume forwarding you do your timing another way.

ii. <u>Mispredicted Branches:</u> Your simulator will assume an always-not-taken branch predictor. This implies that you will fetch the next sequential instruction every cycle, irrespective of whether the previous instruction was a branch, a jump or any other instruction. When you encounter a "taken" branch or a "jump", your fetch unit will have fetched instructions from the wrong path. Assume a pipeline timing where you can figure out the branch target and branch condition at the end of EX stage. This means that you would have fetched two instructions from the wrong path. You need to ensure that these instructions are flushed from the pipeline, otherwise they would corrupt the machine state.

<u>Implementation Methodology:</u>

The following methodology describes how you can implement the stalling of the pipe and flushing of instructions. It is not necessary that you follow this idea, you may think up of a better scheme if you wish to as long as it achieves correct clock cycle counts.

You will maintain a global clock counter, an array of length 5, and a circular buffer of 5 entries. Each entry in the circular buffer will correspond to an instruction in the pipeline. The entry could be a struct that contains all information regarding that instruction i.e. the source and destination operands etc. The array will simply record what each stage contains at that point, e.g. maintain a pointer to the appropriate struct. Note that we only need 5 struct entities and they can be recycled, because for every new instruction fetched, at least one of the structs can be reclaimed as the instruction it held has flown out of the pipe. For normal operation, you will increment the clock by 1 and throw out the instruction in the last stage. Update the array contents in a manner that it reflects a flow of 1 stage in the pipeline. This makes space for a new instruction which you can now fetch into that slot.

When you encounter a data hazard, then you must first determine how many cycles the pipeline needs to stall. Note that this stall penalty would vary depending on which stage the producer instruction is going through when the consumer instruction gets decoded. Then, you lock the consumer instruction and any following instruction into their stages for the duration of the stall penalty, while the instructions in the later stages keep moving forward. A simple way to do that is to insert special NOP instructions in the EX stage every cycle until the duration of the stall penalty. You will also need to stop fetching any new instructions for the stall duration.

When you encounter a "taken" branch or a "jump" in the EX stage, , then the instructions upstream of the branch must be made to point to NOP instruction, the PC should be updated with the branch/jump target and instruction fetching should proceed from the new PC value in the next cycle.

You can figure out how to deal with other case, such as forwarding in a similar fashion. The underlying scheme is to identify which instruction is to be stalled, all those downstream are allowed to flow ahead, all those upstream are locked. When a flush is required, the upstream instructions will be made to point to NOP instructions.

6. **Results:** Once you have understood all the above details, you are ready to implement the simulator. You will be implementing three different versions of the simulator:

   A. Functional simulator only
   B. Functional simulator + Timing simulator assuming no pipeline forwarding
   C. Functional simulator + Timing simulator with pipeline forwarding

7. **Project Report:** Your project report must include the following results for the provided memory image. Some of these results can be shown in the form of a table or graph, whichever you prefer.
   i. Total number of instructions and a breakdown of instruction frequencies for the following instruction types: Arithmetic, Logical, Memory Access, Control Transfer.
   ii. Final state of program counter, general purpose registers and memory (You only need to include the register and memory locations whose state has changed during the program execution)
   iii. Describe the stall conditions in both the "no forwarding" and "forwarding" cases and how long you stalled the pipeline for each stall condition (e.g., in the "no forwarding" case, if a consumer instruction comes right after a producer instruction, then the stall penalty is 2 cycles)
   iv. In the case of "no forwarding", the number of data hazards and average stall penalty per hazard.
   v. In the case of "forwarding", the number of data hazards which could not be fully eliminated by forwarding.
   vi. Execution time in terms of number of clock cycles for the "no forwarding" and the "forwarding" scenarios.
   vii. Speedup achieved by "forwarding" as compared to "no forwarding".

   You should also include a copy of your source files with your project report.