CS 520 - Fall 2019 - Ghose

Programming Project 1: Simulator for APEX with in-order issue

DUE: Tuesday, November 12 by noon via Blackboard

All demos to be completed by Tuesday, Nov. 19

This is a project that has to be done INDIVIDUALLY.

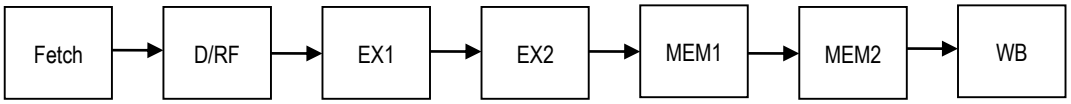
Soft copies of all documents to be submitted via Blackboard by noon, Nov. 12. You also need to demonstrate your simulator to one of the TAs. Instructions for scheduling the demos will be posted on the course page later.

Start working on this project as early as you can.

PROJECT DESCRIPTION

PART A:

This project requires you to implement a cycle-by-cycle simulator for an in-order APEX pipeline with 7 pipeline stages, each with a delay of one cycle, as shown below:



The execution of arithmetic and logical operations, as well as memory address calculations, is pipelined into two stages, EX1 and EX2. Likewise, memory accesses by load and store instructions is pipelined into two stages, MEM1 and MEM2. The arithmetic operations supported are integer operations on 32-bit operands and include addition (ADD, ADDL instructions), subtraction (SUB, SUBL instructions) and multiplication (MUL instruction). **Assume for simplicity that the two values to be multiplied have a product that fits into a single register.**

Instruction issues in this pipeline take place in program-order and a **simple interlocking logic** is used to handle dependencies in the D/RF stage. **For PART A of this project, this pipeline has no forwarding mechanism.**

Registers and Memory:

Assume that there are 16 architectural registers, R0 through R15. The code to be simulated is stored in a text file with one ASCII string representing an instruction (in the symbolic form, such as ADD R1, R4, R6 or ADDL R2, R1, #10) in each line of the file. Memory for data is viewed as a linear array of integer values (4 Bytes wide). Data memory ranges from 0 through 3999 and memory addresses correspond to a Byte address that begins the first Byte of the 4-Byte group that makes up a 4 Byte data item. Instructions are also 4 Bytes wide, but stored as strings in a file, as noted above for this project. Registers are also 4 Bytes wide.

Instruction Set:

The instructions supported are:

- Register-to-register instructions: ADD, ADDL (add register with a literal), SUB, SUBL (subtract literal from a register), MOVC (move constant or literal value into a register), AND, OR, EX-OR and MUL. As stated earlier, you can assume that the result of multiplying two registers will fit into a single register.
- MOVC <register> <literal>, moves literal value into specified register. The MOVC uses the EX1 and EX2 stages to add 0 to the literal and updates the destination register from the WB stage.
- Memory instructions: LOAD, LDR, STORE, STR: LOAD and STORE both include a literal value whose content is added to a register to compute the memory address, while LDR and STR instructions calculate a memory address by adding the contents of two registers. The semantics of LDR and STR are as follows, using the notation described in the class:

  LDR <dest>, <src1>, <src2>: <dest> ← Mem[<src1> + <src2>]

  STR <src1>, <src2>, <src3>: Mem[<src2> + <src3>] ← <src1>

- Control flow instructions: BZ, BNZ, JUMP and HALT. Instructions following a BZ, BNZ and JUMP instruction in the pipeline should be flushed on a taken branch. The zero flag (Z) is set only by arithmetic instructions when they are in the WB stage. **The dependency that the BZ or BNZ instruction has with the immediately prior arithmetic instruction (ADD, MUL, SUB) that sets the Z flag has to be implemented correctly.**

The semantics of BZ, BNZ, JUMP and HALT instructions are as follows:

- The BZ <literal> instruction cause in a control transfer to the address specified using PC-relative addressing if the Z flag is set. The decision to take a branch is made in the Integer ALU stage. BNZ <literal> is similar but causes branching if the Z flag is not set. BZ and BNZ target 4-Byte boundaries (see example below).
- JUMP specifies a register and a literal and transfers control to the address obtained by adding the contents of the register to the literal. The decision to take a branch is made in the Integer ALU stage. JUMP also targets a 4-Byte boundary.
- The HALT instruction stops instruction fetching as soon as it is decoded **but allows all prior instructions in the pipeline to complete** before returning to the command line for the simulator.

As soon as a target memory address is calculated, instruction following a taken branch or a JUMP are squashed immediately, in the same cycle as the address calculation is completed in the EX2 stage.

The instruction memory starts at Byte address 4000. You need to handle target addresses of JUMP correctly - what these instructions compute is a memory address. However, all your instructions are stored as ASCII strings, one instruction per line in a SINGLE text file and there is no concept of an instruction memory that can be directly accessed using a computed address. To get the instruction at the target of a BZ, BNZ or JUMP, a fixed mapping is defined between an instruction address and a line number in the text file containing ALL instructions:

- Physical Line 1 (the very first line) in the text file contains a 4 Byte instruction that is addressed with the Byte address 4000 and occupies Bytes 4000, 4001, 4002, 4003.
- Physical Line 2 in the text file contains a 4 Byte instruction that is addressed with the Byte address 4004 and occupies Bytes 4004, 4005, 4006, 4007.
- Physical Line 3 in the text file contains a 4 Byte instruction that is addressed with the Byte address 4008 and occupies Bytes 4008, 4009, 4010, 4011 etc.

The targets of all control flow instructions thus have to target a 4_byte boundary. So when you simulate a JUMP instruction whose computed target has the address 4012, you are jumping to the instruction at physical line 4 in the text file for the code to be simulated. Register contents and literals used for computing the target of a branch should therefore target one of the lines in the text file. Your text input file should also be designed to have instructions at the target to start on the appropriate line in the text file.

Instructions are stored in the following format in the text file, one per line:

<OPCODE characters><comma><argument1><comma><argument2> <comma><argument3>

Where arguments are registers or literals. Registers are specified using two or three characters (for example, R5 or R14). Literal operands, if any, appear at the end, preceded by an optional negative sign. This format is different from the notation used elsewhere (and in this problem description), as it uses commas to separate arguments.

Simulator Commands:

Your simulator is invoked by specifying the name of the executable file for the simulator and the name of the ASCII file that contains the code to be simulated. Your simulator should have a command interface that allows users to execute the following commands:

Initialize: Initializes the simulator state, sets the PC of the fetch stage to point to the first instruction in the ASCII code file, which is assumed to be at address 4000. Each instruction takes 4 bytes of space, so the next instruction is at address 4004, as memory words are 4 Bytes long, just like the integer data items.

Simulate <n>: simulates the number of cycles specified as <n> and waits. Simulation can stop earlier if a HALT instruction is encountered and when the HALT instruction is in the WB stage.

Display: Displays the contents of each stage in the pipeline, all registers, the flag register and the contents of the first 100 memory locations containing data, starting with address 0.

A skeleton code in C for this simulator that simulates only two instructions is included in a separate file. This simulator has most of the major data structures that you will need to use and also implements the simulated memory for instructions and data. You can extend this code by modifying it appropriately for this assignment.

PART B:

For the simulated pipeline, add the complete set of necessary forwarding logic to forward register values. Assume that forwarding can take place only from the output of EX2 and MEM2 stages and that any dependency stall causes the dependent instruction to wait in the D/RF stage, however, a stalled instruction can pick up a forwarded value as it is entering the EX1 stage.

Please document your code appropriately and check it out by using test code sequences. For PART A, it may be easier to implement one instruction at a time and then test the simulator. You can write simple test code sequences yourself. For PART B, a similar approach may be used by adding code to forward data between one specific pair of stages at a time. Again, use you own test code sequence. Some sample test code will be posted later.