CS 520 - Fall 2016 - Ghose

Programming Project 1: Simulator for in-order version of APEX with two separate FUs

DUE: Monday, November 14 in Class

All demos to be completed by Friday, Nov. 18

This is a team project to be done with a two-member team. The same team will work on extending this simulator as the final project for this course. You need to find yourself a team member.

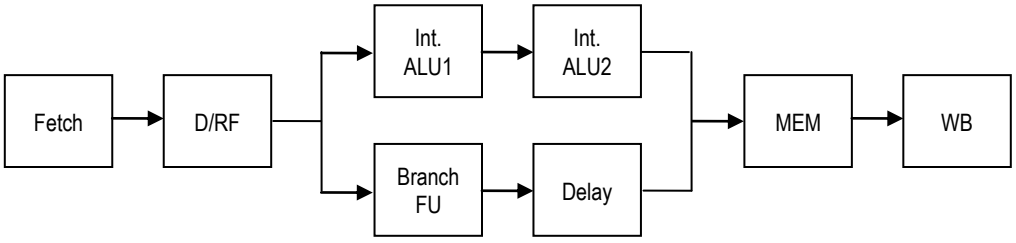
You are required to submit a hardcopy design documentation, the weekly activity log (see below), a CD of all your code and example results. Each team also needs to demonstrate your simulator to one of the TAs. Instructions for scheduling the demos will be posted on Blackboard later.

To ensure that team members are contributing fairly to the project, keep a weekly (or daily) activity log to document progress on the project, clearly stating who did what, how issues were identified by the team and addressed. This activity log is part of the documentation to be turned in.

Start working on this project as early as you can.

PROJECT DESCRIPTION

This project requires you to implement a cycle-by-cycle simulator for an in-order APEX pipeline with two different function units, as shown below:



For all instructions, excepting the branches, BAL (see below) and JUMP, the operations are performed on a 2-stage integer ALU (with stages Int. ALU1 and Int. ALU2, with a delay of one cycle for each stage). The BZ, BNZ, BAL and JUMP instructions have their target addresses computed in the Branch FU and are all completed in this stage. The decision to take a branch is also made in this stage. The BAL instruction saves the return address into a special register X, which is updated from the WB stage. The Branch FU is followed by a one cycle delay stage, so this is an in-order pipeline where execution begins in order and all registers are updated in order from WB.

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) 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.

Instruction Set:

The instructions supported are:

- Register-to-register instructions: ADD, SUB, MOVC, MUL, AND, OR, EX-OR (all done on the Integer ALU in two cycles). 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 ALU stages to add 0 to the literal and updates the destination register from the WB stage.
- Memory instructions: LOAD, STORE - both using a literal offset.
- Control flow instructions: BZ, BNZ, JUMP, BAL, HALT. Instructions following a BZ, BNZ, JUMP and BAL instruction in the pipeline should be flushed on a taken branch. The zero flag (Z) is set only by arithmetic instructions. You can assume that this flag is stored as an extension of the destination register for the instruction that generates the result. (This will simplify coding for Project 2!).

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

- BAL <register>, literal: saves address of the ***next*** instruction in a special register X and then sets fetch PC to contents of <reg> plus the literal.
- JUMP specifies a register and a literal and transfers control to the address obtained by adding the contents of the register to the literal. A return from a function call is coded as JUMP X, #0.
- The HALT instruction stops execution.

You need to handle target addresses of JUMP and BAL 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 accessed using a computed address. To get the instruction at the target of a BZ, BNZ, JUMP or BAL, a fixed mapping is defined between a instruction address and a line number in the text file containing ALL instructions:

Address 4000 is mapped to line 0 in the text file

Address 4001 is mapped to line 1 in the text file

Address 4002 is mapped to line 2 in the text file etc.

So when you simulate a JUMP instruction whose computed target has the address 4025, you are jumping to the instruction at line 25 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.

Forwarding:

The processor to be simulated incorporates forwarding. Forwarding is used to deliver a result to a flow-dependent instruction, which advances as much as possible in the pipeline and needs the forwarded data when it enters the stage where that data is needed or earlier. Make sure that you ***also*** implement the forwarding from a LOAD to an immediately following STORE that requires the value as the register

value being stored. All other cases of forwarding need to be implemented, including ***the forwarding of the value of register X*** from the output of the Branch FU and the output of the delay stage. Most importantly, you need to make sure ***flag values can also be forwarded to their consumers*** (the BZ and BNZ in this case).

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 4001, 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 (including X) and the contents of the first 100 memory locations containing data, starting with address 0.

Any one of the following languages can be used to implement BOTH parts: C, C++ or Java. ***You will be required to extend the simulator for Project 1 as part of the final project.***