CS 520 - Fall 2021 - Ghose

Programming Project 1: Simulator for APEX with in-order issue
DUE: Wednesday, October 20 by noon via Brightspace

All demos to be completed by Wednesday, Oct. 27

****EARLY SUBMISSION IS ENCOURAGED****

THIS VERSION INCLUDES THE DESCRIPTION OF THE JUMP INSTRUCTION

This is a project that has to be done INDIVIDUALLY. DO YOUR OWN WORK.

Soft copies of all documents to be submitted via Brightspace by noon on Wednesday, Oct. 20. 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

The project has two parts:

PART A:

This project requires you to implement a cycle-by-cycle simulator for an in-order APEX pipeline with 5 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, are all implemented in the EX stage. Likewise, memory accesses by load and store instructions are performed in MEM. 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 scoreboarding logic** is used to handle dependencies to consumer instructions in the D/RF stage. **For PART A of this project, this, pipeline has the simple scoreboarding logic to stall an instruction in the D/RF stage till the source registers of the instruction are available. Further, the processor has no forwarding mechanism. For PART B, a forwarding mechanism is added to the simulator code developed for PART A.**

(OVER)

Registers:

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. Registers are 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.
- The instruction **MOVC** <register>, # <signed literal>, moves signed literal value into specified register. The MOVC uses the EX stage to add 0 to the signed literal and updates the destination register from the WB stage.
- Memory instructions include the **LOAD** and the **STORE**: LOAD and STORE both include a literal value whose content is added to a register to compute the memory address.
- **Two additional memory instructions are supported.** These are as follows:
    - **LDI** <dest> <src1> #<signed literal>
    - **STI** <src2> <src1> #<signed literal>

  These instructions are like the LOAD and the STORE but, **additionally**, both instructions increment the value of <src1> by 4 *after* the original value of <src1> is read out to compute a memory address in the D/RF stage. The actual update to the register <src1> takes place from the WB stage, like any other register update. A second write port is assumed to exist to enable two register values to be updated simultaneously from the WB stage. A separate adder exists in the EX stage to perform the increment operation to <src1> in parallel with the computation of the memory address by the ALU.
- The processor supports two flag values, the **Z**(ero), and **P**(ositive) flags that are set by instructions that perform register-to-register ***arithmetic*** operations (specifically, by the ADD, ADDL, SUB, SUBL, MUL instructions). The values of the two flags are set only by a register-to-register ***arithmetic instruction*** *when the instruction is in the EX stage*, towards the end of the clock cycle. **Put in other words, the flags are actually updated within the EX stage**. These are quite different from register updates that take place from the WB stage. The flags are set as follows:
    - The Z flag is set when the result of the arithmetic operation has the value zero.
    - The P flag is set when the result of the arithmetic instruction is **positive and higher than zero**.
- Two pairs of conditional control flow instructions (that is, ***branch instruction***s) are supported, one pair dedicated to a specific flag. These instructions perform conditional branching based on the values of the Z, and P flags and are follows:
    - **BZ**: branch if the zero flag is set; **BNZ**: branch if the zero flag is not set.
    - **BP**: branch if the positive flag is set; **BNP**: branch if the positive flag is not set.

  All of these four branch instructions use PC-relative addressing and specify a signed literal and the address of the location to branch to (called the "branch target") is computed in the EX stage by adding the signed literal to the address (PC-value) of the branch instruction. All target instructions begin at a 4-Byte boundary in memory. The decision whether to take branch or not is taken when the condition for branching is satisfied is made when the branch instruction is in the EX stage itself, towards the end of the clock cycle. If the branching condition is satisfied, all following instructions in the pipeline are

dismissed (these instructions with be in the D/RF stage and the F stage). In the following cycle, the instruction to be fetched comes from the address computed in the previous cycle for the branch target. If the branching condition is tested and not found to be valid, instruction execution continues as usual (and no instructions are dismissed from the earlier stages in the pipeline). "Dismissed" simply means that no further processing is done for these instructions which pass through the remaining pipeline changes without invoking any operations within the stages. In effect, each dismissed instruction is a single-cycle bubble.

As an example, consider the branch instruction BNP # -64. Assume that this instruction has the address 4096 – this is its PC value. When this BNP instruction is in the EX stage, the address of the target is computed by adding (-64) to 4096, to get the target address as 4032. At the same time, the P flag is tested. If the P flag is NOT set, the instructions in the D/RF and F are flushed and the next instruction fetched by the F stage comes from the address 4032. If the P flag is set when the BNP is in the EX stage, the instruction physically following the one in the F stage is fetched.

The dependency that the branch instruction has with the immediately prior arithmetic or CMP (see below) and the flag the branch instruction has to check needs to be implemented correctly.

- A compare instruction, **CMP** <src1>, <src2> is added to compare the contents of two registers and set the Z and P flags based on the result of the comparison. Specifically, the Zero flag is set if the values stored in the two source registers are equal, the Positive flag is set when the contents of <src1> has a value greater than the value of the contents of <src2>.
- A **HALT** instruction is added to the ISA. As soon as a HALT instruction enters the D/RF stage, further instruction fetching is suspended and all prior instructions are processed. When the HALT instruction enters the WB stage, the simulator returns to the command prompt (see later).
- A **NOP** instruction, which does nothing as it proceeds through the pipeline.
- 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.

The implementation of an ADD, BZ, BNZ, HALT and a LOAD are already provided. You are to implement the code for all other instructions described above. The **JUMP** instruction has the following syntax:

JUMP <src1> <signed literal> (example: JUMP R4 #-24)

The semantics of the JUMP instruction is as follows:

JUMP calculates the address to which control has to be transferred by adding the value of the signed literal to the contents of the specified register (rsrc1) in the EX stage in the cycle the JUMP instruction is in the EX stage. This is the address of the memory location that will contain the next instruction to be processed by the pipeline. At the end of that cycle in which the memory address for the instruction to be processed following the JUMP is calculated, when JUMP is in the EX stage, the following things are done:

a) The value of the PC in the F(etch) stage containing the address of the instruction to be fetched in the following clock cycle is set to the address computed in the EX stage.
b) Any further processing of the two instructions currently in the Fetch stage and the D/RF stage are abandoned.

As an example, consider the code fragment:

JUMP R4, #8
ADD R1, R2, R3
LOAD R3, R5, #16

At the end of the clock cycle, say Cycle T, when JUMP is in the EX stage, the address of the next instruction to be processed following this JUMP in the sequential execution model is calculated by adding the contents of R4 with a literal value of 8. Let's assume that there is a **SUB** R1, R2, R6 instruction stored at this memory address. This is the address that is used to overwrite the PC in the F(etch) stage at the end of Cycle T. Simultaneously, stapes are taken at the end of Cycle T ensure that the ADD and the LOAD are not processed any further. In the sequential execution model, the sequence of instructions processed is thus:

JUMP R4, #8
SUB R1, R2, R6

Note that like the JUMP, the instructions BZ, BNZ, BP, BNP instructions all require further processing of the two instructions following them to be abandoned when the branching condition tested by each of these instructions is satisfied.

The Simulated Instructions, Instructions and Data Memory:

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 BZ 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><space><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. To implement the other instructions described earlier, you will need to modify the code file parser.

Memory for data is viewed as a linear array of integer values that are each 4 Bytes wide. The memory space for data is Byte-addressable. All Byte addresses used by the loads and stores are required to start at a 4-Byte boundary. Thus, data items fetched by the loads and written to by the stores are assumed to start at 4-Byte boundaries. The data memory space ranges from Byte Address 0 through Byte Address 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 accessed. Instructions are also 4 Bytes wide, but stored as strings in a file, as noted above for this project.

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.

Single_step: Advances the simulation by one cycle. Implemented using a simple <return> at the command prompt.

Display: Displays the contents of each stage in the pipeline, all registers, the flag register and the contents of the first 10 memory locations containing data, starting with address 0.

ShowMem: Displays the content of a specific memory location, with the address of the memory location specified as an argument to this command.

A skeleton code in C for this simulator is included in the zipped file uploaded to Brightspace earlier. 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.

Simulator Versions to Develop and Submit:

You have to develop two separate simulator versions of the simulator, one for PART A and another for PART B, as follows:

PART A: For PARTA, you will have to implement:

- The code for all other instructions specified above.
- The code for any simulator command that does not exist in the skeleton code given.
- The code to implement interlocking using a **simple** scoreboarding logic, as described earlier.

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 EX and MEM 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 EX 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.

Testing:

You should test the simulator versions you develop with you own test code. Use small code fragments. We will also supply you with two different code segments at a later time.