CS 520 - Fall 2023 - Ghose

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

DUE: Sunday, October 8 by midnight via Brightspace

All demos to be completed by Saturday, Oct. 14, by 5 pm

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

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 midnight on Sunday, Oct. 8. 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

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. All computations (addition, subtraction, multiplication), bitwise operations (and, or, ex-or) and all memory address calculations are done in the EX stage.

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.

Project 1 has two parts:

- For **PART 1** 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 2**, forwarding mechanisms are added to the simulator code developed for PART 1.

Detailed specifications of the ISA and pipeline for the processor follows.

A. The ISA Implemented by the Processor

1. **Registers defined in the ISA:** Assume that there are 32 architectural registers, R0 through R31. 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. There is a special register used for all the flags (**P** for positive, **N** for negative, **Z** for zero).

(Continues to the next page)

2. **Instruction Set:** The instructions supported are as follows:

- 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**, **STORE, LOADP and STOREP**: LOAD and STORE both include a literal value whose content is added to a register to compute the memory address and their semantics are as defined in the class notes/pdfs of the PowerPoints. The LOADP and STOREP instructions have the same semantics as the LOAD and the STORE, respectively, but in addition increment the source register used for memory address calculation by 4 once the original value of that source register is read out (which happens in this pipeline in the D/RF stage). The increment is implemented by a *dedicated* logic within the EX stage, that operates in parallel with the ALU used for address calculation. Updates to the source register used for memory calculation by the LOADP and STOREP are made when these instructions move into the WB stage. Assume that an extra write port exists on the register file for supporting the write for the update to the source register used for address calculation.
- A compare instruction, **CML** <src1>, <literal> is added to compare the contents of a register with a literal value and set the Z, N and P flags based on the result of the comparison. A second compare instruction, **CMP** <src1>, <src2> is added to compare the contents of two registers. Both of these instructions affect the flag values, as described later).
- A **HALT** instruction is added to the ISA. As soon as a HALT instruction enters the D/RF stage, further instruction fetching is suspended but all prior issued instructions are processed. The HALT instruction stops instruction fetching as soon as it is decoded **but allows all prior instructions in the pipeline to complete** and returns to the command line for the simulator (see later) when it enters the WB stage.
- A **NOP** instruction, which does nothing as it proceeds through the pipeline, all the way to the WB stage.
- The processor supports the following flag values:
    - The **Z**(ero), **N**(egative) and the **P**(ostive) flags. **These are stored in a special register called CC** (for **C**ondition **C**odes).
    - These flags are set by instructions that perform register-to-register ***arithmetic*** operations (specifically, by the ADD, ADDL, SUB, SUBL, MUL instructions), logical instructions (bitwise AND, OR, EX-OR) and the compare instructions. **Assume for simplicity that the two values to be multiplied have a product that fits into a single register.** The values of these flags are set only by a register-to-register ***arithmetic instruction and the any instruction that performs bitwise logical operation*** *when such an 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**. The special register containing these flags are thus quite different from register updates that take place from the WB stage.
    - **The instructions MOVC, LOAD, LOADP, STORE, STOREP do not affect the values of these flags.**
- The following 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, N 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.
- **BN**: branch if the negative flag is set.
- **BNN:** branch if the negative flag is not set.

ALL of these 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, so when the branch is taken, you have a total bubble of two cycles, since the two instructions following the BZ, BNZ, BN, BNN, BP and BNP are in the pipeline and need to be dismissed.

As an example, consider the branch instruction BNZ # -64. Assume that this instruction has the address 4096 – this is its PC value. When this BNZ 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 Z flag is tested. If the Z 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 (= 4096-64). If the Z flag is set when the BNZ 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 instruction that can set the flags (arithmetic or CML/CMP or any instruction performing bitwise logical operations**), and the flag the branch instruction has to check needs to be implemented correctly.**

- An instruction for implementing a function call, **JALR** (Jump-and-link register) with the following format:

  JALR <dest>, <src1>, <literal>

  This instruction transfers control to the memory location computed by adding the contents of the register src1 with the sign extended literal and saves the address of instruction that follows JALR in the register, dest. As in the case of the branches, the address to which control flows has to be 4-Byte aligned. The register <dest> is updated when JALR is in the WB stage. This instruction thus saves the return address in the register, dest.

- An unconditional control transfer instruction, **JUMP** with the following format:

  JUMP <src1>, <literal>

  This instruction transfers control to the 4-Byte aligned address computed by adding the contents of the register, src1, with the sign-extended literal in the instruction. Return from a function call is implemented by using:

  JUMP <src1>, #0

Where <src1> is the register which contains the return address that was saved by the JALR instruction that made a call to this function.

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.

A list of all the instructions to be implemented in the APEX simulator for the project is as follows:

Function of instruction(s)	Symbolic Name
Writing a literal value to a register	**MOVC**
Memory accesses	**LOAD, LOADP, STORE, STOREP**
Arithmetic operations	**ADD, ADDL, SUB, SUBL, MUL**
Bitwise logical operations	**AND, OR, EX-OR**
Comparing register value(s)	**CML, CMP**
Conditional branching	**BZ, BNZ, BP, BNP, BN, BNN**
Unconditional control transfer	**JUMP**
Function call	**JALR**
Miscellaneous	**NOP, HALT**

B. SIMULATED MEMORY

1. The Simulated Instruction Memory:

The instruction memory starts at Byte address 4000. You need to handle target addresses of BZ or BNZ instruction 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, 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 transferring control 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.

2. **The Simulated Data Memory:**

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.

C. OTHER HARDWARE DETAILS

Data dependencies are handled in the D/RF stage using a **simple scoreboarding** mechanism. Because of the in-order pipeline where out-of-order writes are not possible and register reads take place earlier than register writes within the pipeline, you have to implement only Condition (a) of simple scoreboarding.

All instructions, including the MUL, spend exactly one cycle in the EX stage.

Assume no cache misses occur, so that memory accesses for data and instructions take exactly one cycle.

PART 2:

Part 2 of the effort requires you to extend the simulator to add data forwarding support to forward data to waiting instructions in the D/RF stage. You need to implement the prioritization needed for forwarding correctly. Note that for LOADP and STOREP, the value of the updated register used for address calculation also needs to be forwarded. Make sure that you identify all possible instances of forwarding and implement them correctly.

D. OTHER SIMULATOR SPECIFICATIONS

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 returns to the command prompt. 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 <address>: Displays the content of a specific data 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 1 and another for PART 2, as follows:

PART 1: For PART1, 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.

Testing:

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