CS520 Computer Architecture

Project 2 – Fall 2021

Due 12/07/2021, 11:59 pm

**1. RULES**

(1) All students must work alone. Cooperation is not allowed.

(2) Sharing of code between students is considered cheating and will receive appropriate action in accordance with University policy. The TA will scan source code through various tools available to us for detecting cheating. Source code that is flagged by these tools will be dealt with severely.

(3) You must do all your work in the C/C++.

(4) Your code must be compiled on remote.cs.binghamton.edu or the machines in the EB-G7 and EB-Q22. This is the platform where the TAs will compile and test your simulator. As I know, they all have the same software environment.

(5) There will be no extension in the due date.

**2. Project Description**

In this project, you will construct a superscalar pipeline.

**3. Baseline Pipeline**

Model simple pipeline with the following stages.

* 1 stage for fetch (IF)
* 1 stage for dispatch (ID)
* 1 stages for addition and subtraction (Execution unit A: EXa)
* 2 stages for multiplication and division (Execution unit B: EXb)
* 4 stages for memory operation (Memory unit: MEM)
* 1 stage for write back (WB)
* 16 x 4B registers
* 64KB memory (code for 0-999, data for 1000 – 65535)

memory.map is the memory map file for this project.

Each memory entry has 4B memory space (A space is in between memory entries)

* Instruction formats (4B fixed instruction)

set Rx, #Imm // set value to register Rx

ld Rx, #Addr // load into register Rx the data stored in the address #Addr. E.g.) ld R1, 0x0010

ld Rx, Ry // load into register Rx the data stored in the address at Ry

st Ry, #Addr // store the content of register Rx into the address #Addr. E.g.) st R1, 0x0010

st RX, Ry // store the content of register Rx into the address at Ry

add Rx, Ry, Rz // Compute Rx = Ry + Rz

add Rx, Ry, #Imm // Compute Rx = Ry + #Imm (an immediate valve)

sub Rx, Ry, Rz // Compute Rx = Ry – Rz

sub Rx, Ry, #Imm // Compute Rx = Ry - #Imm (an immediate valve)

mul Rx, Ry, Rz // Compute Rx = Ry \* Rz

mul Rx, Ry, #Imm // Compute Rx = Ry \* #Imm (an immediate valve)

div Rx, Ry, Rz // Compute Rx = Ry / Rz

div Rx, Ry, #Imm // Compute Rx = Ry / #Imm (an immediate valve)

bez Rx, #Imm // branch to #Imm if Rx==0

bgez Rx, #Imm // branch to #imm if Rx >= 0

blez Rx, #Imm // branch to #imm if Rx <= 0

bgtz Rx, #Imm // branch to #imm if Rx > 0

bltz Rx, #Imm // branch to #imm if Rx < 0

ret // exit the current program

This project will update your prior pipeline in project #1. As described in the previous project, the dependency check and register read are done at the ID stage.

**3.1. Simple Pipeline update from Project 1**

The register file allows the write operation in the first half of the clock cycle and the read operation in the second half of the cycle. The pipeline has bypassing/forwarding. The result of each instruction becomes ready after the last pipeline stage of the corresponding execution unit (e.g., if MEM unit needs 4 cycles to complete an instruction, your result is available after the 4th cycle in the Mem unit).

**3.2. Pipeline update: Parallel + Dynamic + Diversified pipeline**

**Pipeline Width & Register Files:** Your pipeline width is 2, meaning that all functional units are doubled in the pipeline. Here, we call one pipeline as pipeline P1 and another as pipeline P2. The pipeline supports the bandwidth to extend the wire width in the pipeline twice. Your register file also has four read ports and two write ports. Each register only allows one operation at a cycle. The register file allows the write operation in the first half of the clock cycle and the read operation in the second half of the cycle.

**Fetch & Dispatch:** Two instructions are fetched simultaneously. However, the ordering is maintained in the way that always the P1 fetch unit fetches an earlier instruction while the P2 fetch unit fetches the next instruction. The pipeline registers do not support reordering, meaning that two instructions at the IF stage must advance simultaneously. An instruction at the ID stage can advance alone **in-order**, meaning that only the instruction at the P1 ID stage can advance alone. The freed space cannot be used by a subsequent instruction until both instructions advance.

**Dependency:** The dependency check and register read are performed at the ID stage. Additionally, the pipeline has bypassing/forwarding. The result of each instruction becomes ready after the last pipeline stage of the corresponding execution unit (e.g., if MEM unit needs 4 cycles to complete an instruction, your result is available after the 4th cycle in the Mem unit). In the project codes, the forwarded values can be found in the cpu->forwarding variables.

**Execution units:** The pipeline has 2x EXa units (EXa1 and EXa2), 1x EXb unit, 1x Mem unit, and 2x Cond units. The Cond unit processes both conditional branch instructions (bez, bgez, blez, bgtz, and bltz) and unconditional branch instructions (ret). The set instruction only sets a register at the WB stage, but we define that it takes the path to the EXa. All other instructions take paths to the corresponding function units at the ID stage. The pipeline has two IF, ID, and WB units, processing two instructions per cycle. If there is a branch instruction, the PC is updated at Cond unit. After the branch instruction is executed, all wrongly fetched instructions must be squashed.

**Branch Instructions:** The branch instruction is now executed (update PC) in the ID stage **in-order**. The branch instruction issued to the EXa stage but does nothing in the stage. All the following instructions are squashed until the branch instruction is resolved in the ID stage (data hazard). At the next cycle after PC is updated by a branch instruction, new instructions can be fetched. The pipeline has no branch prediction.

**Out-of-order Execution:** The pipeline supports out-of-order processing with renaming. The renaming is performed at the ID (dispatch) stage. The pipeline has two new buffers, a reservation station and a reorder buffer. The size of each buffer is as follows.

Reservation Station: 6 entries

Reorder Buffer: 10 entries

If an instruction is dispatched, an entry in the reservation station and an entry in the reorder buffer are allocated for the instruction. When the instruction is issued, the entry in the reservation station is freed. Once the instruction is committed (retire), the entry in the reorder buffer is freed. The instructions stay in the reservation station until they become ready.

The instructions are dispatched in-order and are executed out-of-order. They retire (commit) in-order (i.e., the pipeline supports precise exception). To maintain the in-order processing, the reorder buffer has the first-in, first-out (FIFO) policy. The issue bandwidth is unlimited so that instructions can be issued as long as corresponding functional units are available. The reorder buffer also has unlimited bandwidth so that instruction can update the reorder buffer when its execution completes. However, the WB stage still has a bandwidth limitation, allowing up to two instructions every cycle.

If two ready instructions need the same functional unit at the same cycle, an earlier instruction in the program order is always issued first, while the next instruction must wait its turn at the next cycle. Although this stall is a structural hazard, you do not need to count it for the results. In this pipeline, you only need to count structural hazards (fetch stalls) when fetched instructions at the IF stage cannot advance due to the full fetch queue. Additionally, you need to count the number of the ID stage stalls due to the full reservation station and the full reorder buffer. If the instruction is stalled due to both full buffers, it is counted as a stall due to the full reservation station.

The data is forwarded to both the reservation station and the beginning of the issue stage (the beginning of the execution stages). Due to the out-of-order execution, the pipeline now must monitor WAR hazard. Both WAW and WAR hazards are solved by the renaming.

**Note #1.** You should refer to the course slides for further detail on the buffers and out-of-order execution.

**Note #2.** The return instruction may not be the last completed instruction. For example, the return instruction is at the Cond stage while an earlier instruction is still at the 3rd Mem stage. However, your simulator must measure the pipeline performance until the time when all the instructions complete.



Print the following simulation results.

Start...

Number of ID stage stalls due to the full reservation station:

Number of ID stage stalls due to the full reorder buffer:

Number of fetch stalls:

Total execution cycles:

IPC:

**Tips.**

1) You need good understanding of register renaming, reservation stations, and re-order buffer.

2) You need to check the data dependency at the ID stage, which requires the stage needs to check physical registers, forwarded values (cpu->forwarding), and re-order buffer.

3) A branch instruction can be dispatched to the reservation station only when the branch is resolved.

4) The pipeline must be stalled when the reservation station or the ROB becomes full.

**4. Validation and Other Requirements**

**4.1. Validation requirements**

Sample simulation outputs will be provided on the website. You must run your simulator and debug it until it matches the simulation outputs. (the format is already coded).

Also, the content in the memory map file (memory\_map.txt) must be correct too. Your output must match both numerically and in terms of formatting, because the TAs will “diff” your output with the correct output. You must confirm correctness of your simulator by following these two steps for each program:

1) Redirect the console output of your simulator to a temporary file. This can be achieved by placing “> your\_output\_file” after the simulator command.

2) Test whether or not your outputs match properly, by running this unix command:

“diff –iw <your\_output\_file> <posted\_output\_file>”

The –iw flags tell “diff” to treat upper-case and lower-case as equivalent and to ignore the amount of whitespace between words. Therefore, you do not need to worry about the exact number of spaces or tabs as long as there is some whitespace where the sample outputs have whitespace. Both your outputs and final memory\_map.txt must be the same as the solution.

3) Your simulator must run correctly not only with the given programs. Note that TA will validate your simulator with hidden programs.

**4.2. Compiling and running simulator**

We will provide you the simulator codes. But you have to complete the fetch and dispatch stages (functions) in the given project2.c file.

You will hand in source code and the TA will compile and run your simulator. As such, you must be able to compile and run your simulator on machines in EB-G7 and EB-Q22. This is required so that the TAs can compile and run your simulator. You also can access the machine with the same environment remotely at remote.cs.binghamton.edu via SSH.

The pipeline receives a program name.

e.g. sim test\_01.asm

**5. What to submit**

You must hand in project2.c. Also, you must submit a cover page with the project title, the Honor Pledge, and your full name as electronic signature of the Honor Pledge.

**6. Penalties**

**-10 points** for each date late

**Cheating**: Source code that is flagged by tools available to us will be dealt with according to University Policy. This includes a 0 for the project and other disciplinary actions.