Hazards

October 21, 2024

1 Description

When we implement pipelining, we have to deal with the hazards thereof. Since the processor does many operations at the same time (electrons are always flowing through the transistors), pipeline registers and multiplexors are used to maintain order and produce predictable results. Writing a simulation of the processor can be a bit daunting because a computer program works in serial while the circuits on the processor all work in parallel. In other words, the five stage pipeline of a basic MIPS processor are operating simultaneously on the chip, while they have to operate in some order in your computer program.

Instructions proceed in stages from left to right: Instruction fetch – instruction decode – execute – memory access – writeback. Nevertheless, a sensible way to write a simulation is to process the stages in reverse order. You would write five methods, one for each stage in the pipeline, but call them in the order Writeback-Memory-Execute-Decode-Fetch. Of course, for the first several cycles, stages at the end of the pipeline wouldn't be doing anything useful, but once an instruction has migrated from the fetch stage through the rest of the pipeline, each stage will be doing something each time it is called. At the end of the program, of course, stages on the left will stop doing anything useful and the last stage that does anything significant would be the writeback stage. Pseuocode for this might look like:

```
int nCycle = 0;
while (not done)
{
Writeback();
Memory();
Execute();
  Decode();
  Fetch();
  nCycle++;
}
```

Pipeline registers are very important in synchronizing the operation of the stages. They hold the initial state for a stage, and they record the final state

when a stage is done with its operations. One of the reasons we call our stages "backwards" is so that the destination pipeline register can be written following a stage's execution without overwriting the initial conditions of another stage that hasn't executed yet this cycle. Of course, what information you persist in the pipeline register is up to you and is part of understanding how the processor works.

There will be four parts to this assignment:

1. Unrolled Simulation
2. Branch Simulation
3. Hazards Simulation
4. Dynamic Scheduling

Each part of this assignment will include encoding the instructions that are required to perform the simulation. Two reasons for requiring this should complement your coding environment. Binary data is read and written differently by the various built in file readers in programming languages. Writing the binary output allows for you to use a native format. Writing the binary will allow you to visualize how your program needs to shift the various values back out as the individual parts of the instruction: Rd, Rs1, Rs2, funct3, etc.

When encoding the instructions, assume that the first instruction starts at location 0x0. No memory or register file needs to be simulated, so only the instructions themeselves need to be stored so that they can be referenced again for the branch and hazards portions of the assignments. Plan ahead for this.

Each section will also include a comma separated value (csv) file with the following columns:

1. Cycle
2. Instr
3. Op
4. Fct3
5. Rd
6. Rs1
7. Rs2
8. RegWrite
9. ALUSrc
10. FwdA
11. FwdB

12. MemRd
13. MemWr
14. WBSel
15. bne

Two points from the deliverables will be assigned for case matching the output of the column names and order in your csv file. This will be the primary tool for grading.

The points assigned for this assignment are as follows:

Item	**Points**
Code with 5-Stages	15
Unrolled Simulation	20
Branch Simulation	20
Hazards Simulation	20
Dynamic Scheduling	20
Deliverables	5

1.1 Code with 5-Stages

You will need to design your code to include methods for each of the five stages:

1. Fetch
2. Decode
3. Execute
4. Memory
5. Write-Back

You may use a class or not, depending on your coding style. I highly suggest designing a class to be able to store a few items:

- A program counter
- A counter (for branching calculations)
- Last two instruction (for hazards)

Write your code to take the path to the binary file as an input.

1.2 Unrolled Simulation

Reference the following sequence of instructions for the unrolled simulation. Note that we perform this simulation before any branching to make the introduction to this assignment as simple as possible. To start, you do not need to consider any checks from the instructions, simply determine what is needed at each stage.

```
lw x7, 0(x10)
lw x6, 0(x7)
addi x6, x6, 1
sw x6, 0(x7)
lw x6, 4(x7)
addi x6, x6, 1
sw x6, 4(x7)
lw x6, 8(x7)
addi x6, x6, 1
sw x6, 8(x7)
```

Make sure to complete the remaining pipeline stages for the last instruction. You may leave columns FwdA and FwdB blank or asteriks.

1.3 Branch Simulation

In this section, you will have to determine which register to use as a counter and determine when the branch needs to occur. You may hardcode your logic for the register in question, so as to avoid having to simulate the alu ops for all instructions and also avoid keeping a register file.

When encoding your branch instruction, remember that the encoding removes the LSB bit 0, so your byte offset will need to be divided by two when encoding. Likewise, multiply by two when decoding.

You may leave columns FwdA and FwdB blank or asteriks.

```
lw x7, 0(x10)
addi x5, x0, 3
Loop:
lw x6, 0(x7)
addi x6, x6, 1
sw x6, 0(x7)
addi x7, x7, 4
addi x5, x5, -1
bne x5, x0, Loop
```

1.4 Hazards Simulation

Consider the following instructions for this simulation.

```
sub x2, x1, x3
and x12, x2, x5
or x13, x6, x2
and x2, x12, x13
add x14, x2, x2
```

Complete the encoding as before, and now display the output of your logic for the forward signals as part of your csv output.

1.5 Dynamic Scheduling

In this section, you will investigate eliminating data dependencies based on the unrolling example. Rewrite the code such that each of the group of three operations is performed using unique registers. Then, re-order the instructions in such a way that you can expect to have no stalls. This section proves that dynamic multiple issue can be achieved.

Provide your nine instructions with unique register locations as a text file, and perform the encoding, and output for a final time.

1.6 Deliverables

There will be a deliverable for each part of the assignment along with code. You will be performing encoding on your own, so this will be included as a deliverable.

1. Unrolled Simulation
    - `unrolled_instructions.bin`
    - `unrolled_simulation.csv`
2. Branch Simulation
    - `branch_instructions.bin`
    - `branch_simulation.csv`
3. Hazards Simulation
    - `hazards_instructions.bin`
    - `hazards_simulation.csv`
4. Dynamic Scheduling
    - `dynamic_re-order.txt`
    - `dynamic_instructions.bin`
    - `dynamic_simulation.bin`

Provide these files in a flat tar file (no directories) called `deliverables.tar` .