 Name: Vishnu Varma Venkata

ECE4100/ECE6100/CS4290/CS6290
Advanced Computer Architecture
**Homework 1**
**Due:** Sunday, September 22nd 2019 (11:55 pm)

---

## Part A: Pipelining

---

Consider the 5-stage pipeline discussed in class.
The critical path for each stage is shown below


| | | | | | |
|---|---|---|---|---|---|
| **Critical path** | **(0.9 ns)** | **(0.8ns)** | **(0.9ns)** | **(1ns)** | **(0.7ns)** |

**Problem A.1:**
What is the highest frequency at which this pipeline can operate correctly?

Max latency = 1ns → critical

⇒ highest frequency = $\frac{1}{1\times10^{-9}}$ = 1GHz  CPI = 1

The stall time of a pipelined design can be reduced by Data Bypassing/Forwarding. Performance Evaluations showed that implementing data forwarding reduces the CPI (Clock cycle Per Instruction) by 15%. However, forwarding adds an additional mux to the ID stage or the EX stage that adds 0.4ns to the critical path.

**Problem A.2:**
If we implement data forwarding by adding the mux to the ID stage, what will be the new operating frequency, and overall speedup?
[Speedup = Old Run Time / New Run Time]

CPI = 0.85

critical path = 0.8 + 0.4 = 1.2ns

operating frequency = $\frac{1}{1.2\times10^{-9}} = \frac{10^9}{1.2}$ = 833.33 MHz

Speedup = $\frac{1\times1}{1.2\times0.85}$ = 0.98

**Problem A.3:**
If we implement data forwarding by adding the mux to the EX stage, what will be the new operating frequency, and overall speedup?

critical path: 1.3ns

operating frequency = $\frac{1}{1.3 \times 10^{-9}}$ = 769.23 MHz

Speedup = $\frac{1 \times 1}{1.3 \times 0.85}$ = 0.9049

**Problem A.4:**
Select one recommendation from below based on the answers above
(a) Implement data forwarding by adding the mux to the ID stage
(b) Implement data forwarding by adding the mux to the EX stage
(c) Do not implement data forwarding

without forwarding - lowest total time

**Problem A.5:**
How would your answers for A.2, A.3 and A.4 change if the mux delay was 0.2 ns?

Speedup with mux in ID stage: $\frac{1 \times 1}{(0.8+0.2) \times 0.85}$ = 1.17647

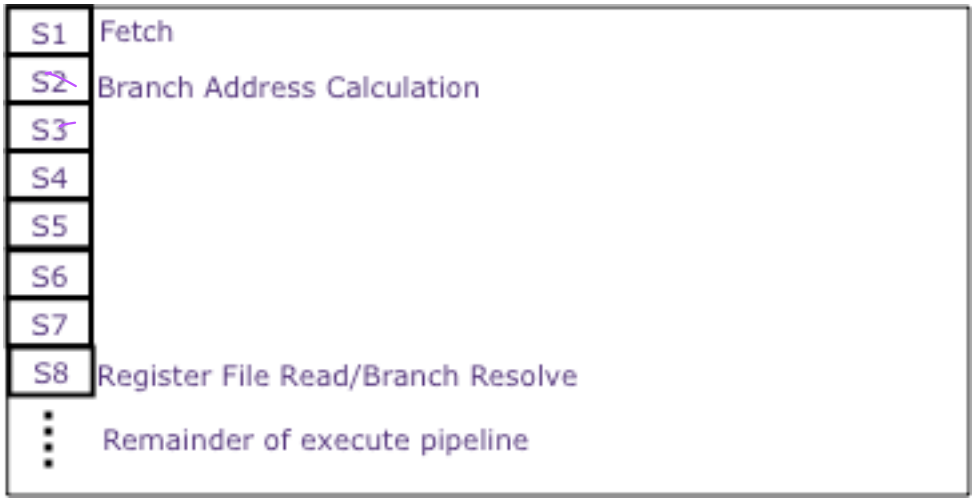
Speedup with mux in EX stage: $\frac{1 \times 1}{(0.9+0.2) \times 0.85}$ = 1.0695

Recommendation (a/b/c) : a is better (speedup)

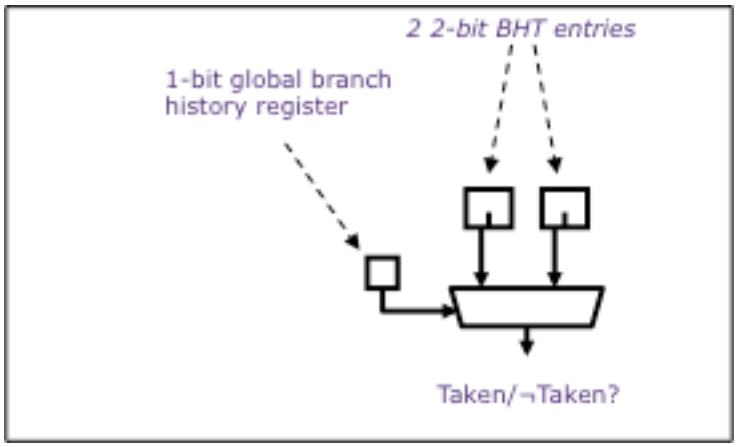
## Part B: Branch Prediction

---

Consider a CPU with a deep pipeline pictured below.




The first stage of the pipeline fetches the instruction. The **second stage** of the pipeline recognizes branch instructions and decodes the branch target (which is present as part of the instruction). The second stage also has a Global History Branch Predictor. *If the branch is predicted to be Taken*, it forwards the decoded target of the branch to the first stage, and kills the instruction in the first stage. The **eighth stage** of the pipeline reads the registers and **resolves the correct direction of the branch**. If the branch direction was mispredicted, the correct direction is forwarded to the first stage, and all instructions in between are killed. The remaining stages finish the computation of the instruction.




The processor uses a **single global history bit** to remember whether the last branch was taken or not. The global history bit is used to index into the **BHT** (the address of the branch is not used for selecting the BHT entry).
The BHT has 2 entries. Each entry is a 2-bit saturating counter.
In state **1X**, we will guess **Taken**; in state **0X**, we will guess **Not Taken**.

**Problem B.1:**

Fill out the following Table on the number of bubbles in the pipeline based on the predicted and actual directions of the branch.

| Branch Prediction | Actual Direction | Pipeline Bubbles |
|---|---|---|
| T | T | 1 |
| T | NT | 7 |
| NT | T | 7 |
| NT | NT | 0 |

In the next two problems we will study execution of the following loop.

| Instruction Label | Address | Instruction |
|---|---|---|
| LOOP | I1 | BEQ R2, R5, NEXT ← NT |
| | I2 | ADD R4, R4, 1 |
| NEXT | I3 | MULT R2, R2, 3847 |
| | I4 | BNEZ R4, LOOP ← T |
| | I5 | NOP |
| | I6 | NOP |
| | I7 | NOP |
| | I8 | NOP |
| | I9 | NOP |
| | I10 | NOP |
| | I11 | NOP |

This processor has **no** branch delay slots. You should assume that branch I1 is **never** taken, and that the branch I4 is **always** taken.
**Thus the correct instruction execution sequence will be:**
**I1, I2, I3, I4, I1, I2, I3, I4, I1, I2, I3, I4, …**

You should also **disregard any possible structural hazards**. The processor always runs at full speed, and there are **no pipeline bubbles** (except for those created by the branches).

**Problem B.2:**

We study how well the history bit works, when it is being **updated by the eighth stage** of the processor (i.e., upon branch resolution).

The eighth stage also updates the BHT based on the result of a branch. The same BHT entry that was used to make the original prediction is updated.

Please fill in the entries of the table below from cycle #9 to 14. An instruction has to be fetched very cycle. Only enter those entries that change. Cycles 0 to 9 are already filled for you.

**The circled instructions are those which will be committed (i.e., are non-speculative).**
The rest of the fetched instructions will become NOPs / pipeline bubbles.

| Cycle | Instruction Fetched | Branch Prediction | Prediction Correct? | Branch Predictor State | | |
|---|---|---|---|---|---|---|
| | | | | Branch History | Last Branch Not Taken Predictor | Last Branch Taken Predictor |
| 0 | - | - | | T | 10 | 01 |
| 1 | I1 | | | | | |
| 2 | I2 | NT | Yes | | | |
| 3 | I3 | | | | | |
| 4 | I4 | | | | | |
| 5 | I5 | NT | No | | | |
| 6 | I6 | | | | | |
| 7 | I7 | | | | | |
| 8 | I8 | | | NT | 10 | 00 |
| 9 | I9 | | | | | |
| 10 | I10 | | | | | |
| 11 | I11 | | | T | 10 | 01 |
| 12 | I1 | | | | | |
| 13 | I2 | NT | Yes | | | |
| 14 | I3 | | | | | |

**Problem B.3**

Now we study how well the branch **history bit** works, when it **is being updated speculatively** by the second stage of the processor. If the branch is mispredicted, the eighth stage sets the branch history bit to the correct value.

*If an instruction in the second stage, and one in the eighth stage both want to update the History Bit, the one in the eighth stage gets higher priority (since it is non-speculative).*

Finally, the eighth stage also updates the BHT based on the result of a branch. The same BHT entry that was used to make the original prediction is updated.

Please fill in the entries of the table below from cycle #5 to 13. An instruction has to be fetched very cycle. Only enter those entries that change. Cycles 0 to 5 are already filled for you.

**The circled instructions are the ones which will be committed (i.e., are non speculative).**
The rest of the fetched instructions will become NOPs / pipeline bubbles.

| Cycle | Instruction Fetched | Branch Prediction | Prediction Correct? | Branch Predictor State | | |
|---|---|---|---|---|---|---|
| | | | | Branch History | Last Branch Not Taken Predictor | Last Branch Taken Predictor |
| 0 | - | - | | T | 10 | 01 |
| 1 | I1 | | | | | |
| 2 | I2 | NT | Yes | NT | 10 | 01 |
| 3 | I3 | | | | | |
| 4 | I4 | | | | | |
| 5 | I5 | T | Yes | T | 10 | 01 |
| 6 | I1 | | | | | |
| 7 | I2 | NT | Yes | NT | 10 | 01 |
| 8 | I3 | | | NT | 10 | 00 |
| 9 | I4 | | | | | |
| 10 | I5 | T | Yes | T | 10 | 00 |
| 11 | I1 | | | T | 11 | 00 |
| 12 | I2 | NT | Yes | NT | 11 | 00 |
| 13 | I3 | | | NT | 11 | 00 |

## Part C: Dependencies and Register Renaming

---

**Problem C.1**

Consider the following instruction sequence. An equivalent sequence of C-like pseudocode is also provided.

```
I1:  L.D     F1, 0 (R1)        ;    F1 = *r1;
I2:  MUL.D   F2, F0, F2        ;    F2 = F0*F2;
I3:  ADD.D   F1, F2, F2        ;    F1 = F2 + F2;
I4:  L.D     F2, 0 (R2)        ;    F2 = *r2;
I5:  ADD.D   F3, F1, F2        ;    F3 = F1 + F2;
I6:  S.D     F3, 0 (R3)        ;    *r3 = F3;
……
```

Fill out the table below to identify all Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW) dependencies in the above sequence. Do not worry about memory dependencies for this question. The dependency between I2 and I3 is already filled in for you.

| | | **Earlier (Older) Instruction** | | | | | |
|---|---|---|---|---|---|---|---|
| | | I1 | I2 | I3 | I4 | I5 | I6 |
| **Current Instruction** | I1 | - | | | | | |
| | I2 | - | - | | | | |
| | I3 | WAW | RAW | - | | | |
| | I4 | - | WAW/ WAR | WAR | - | | |
| | I5 | - | - | RAW | RAW | - | |
| | I6 | - | - | - | - | RAW | - |

**Problem C.2**

Your task is to rewrite the code in C.1 using register renaming to remove as many dependencies as you can. You may use an unlimited number of registers. Do not change the order of instructions. For your convenience, the code in C.1 is repeated below.

```
I1:  L.D     F1, 0 (R1)
I2:  MUL.D   F2, F0, F2
I3:  ADD.D   F1, F2, F2
I4:  L.D     F2, 0 (R2)
I5:  ADD.D   F3, F1, F2
I6:  S.D     F3, 0 (R3)
```

**[After register renaming]**

```
I1: LD    S1 , 0(R1)
I2: MUL.D S2 , F0 , F2
I3: ADD.D S3 , S2 , S2
I4:  L.D  S4 , 0(R2)
I5: ADD.D S5 , S3 , S4
I6:  S.D  S5 , 0(R3)
```

**C.3:**

Suppose we want to run the same code sequence under the system which is 2-wide superscalar with the units shown in the table below.

| | |
|---|---|
| One Memory Unit | 1 cycle latency |
| One Floating-point Adder | 2 cycle latency |
| One Floating-point Multiplier | 8 cycle latency |

All of these functional units are fully pipelined, but there is no bypassing so they have to stall to deal with dependencies. Note that the issue stage and the writeback stage also take one cycle to complete.

The system is an **out-of-order issue, out-of-order completion** machine, **with register renaming**. Many instructions can be issued and completed at one cycle and assume that it can rename as many registers as necessary as in C.2. Also assume that the Issue Queue contains all 6 instructions at the beginning. Fill in the table to show which instruction(s) is being dispatch, executed at each unit, and written back, for each cycle.

Assume that I1 and I2 are issued at cycle 0, as indicated in the table. Remember it has to deal with hazards by stalling (You may not need all the columns in the table).

| Cycle: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dispatch (1) | I1 I2 | I4 | | | | | | | | | I3 | | | | I5 | | | | I6 | | |
| Memory Unit (1) | | I1 | I4 | | | | | | | | | | | | | | | | | I6 | |
| F-Adder (2) | | | | | | | | | | | | I3 | I3 | | | I5 | I5 | | | | |
| F-Mult (8) | | I2 | I2 | I2 | I2 | I2 | I2 | I2 | I2 | | | | | | | | | | | | |
| WriteBack (1) | | | I1 | I4 | | | | | | I2 | | | | I3 | | | | I5 | | | I6 |

## Part D: Out-of-Order Execution, Speculative Execution, and Recovery

---

This problem investigates the operation of a superscalar processor with branch prediction, register renaming, and out-of-order execution. The processor holds all data values in a **physical register file (PRF)**, and uses a **register alias table (RAT)** to map from architectural to physical register names. A **free list** is used to track which physical registers are available for use. A **reorder buffer (ROB)** contains the bookkeeping information for managing the out-of-order execution (but, it does not contain any register data values).

When a branch instruction is encountered, the processor predicts the outcome and takes a snapshot of the rename table. If a misprediction is detected when the branch instruction later executes, the processor recovers by flushing the incorrect instructions from the ROB, rolling back the "next available" pointer, updating the free list, and restoring the earlier rename table snapshot.

We will investigate the execution of the following code sequence (assume that there is **no** branch-delay slot):

```
loop:  lw    r1, 0(r2)     # load r1 from address in r2
       addi  r2, r2, 4     # increment r2 pointer
       beqz  r1, skip      # branch to "skip" if r1 is 0
       addi  r3, r3, 1     # increment r3
skip:  bne   r2, r4, loop  # loop until r2 equals r4
```

The diagram on the next page shows the state of the processor during the execution of the given code sequence. An instance of each instruction in the loop has been issued into the ROB (the beqz instruction has been predicted not-taken), but none of the instructions have begun execution. In the diagram, old values which are no longer valid are shown in the following format: P1. The rename table snapshots and other bookkeeping information for branch misprediction recovery are not shown.

**Register Alias Table**

| | | | |
|---|---|---|---|
| R1 | ~~P1~~ | ~~P4~~ | ~~P7~~ P4 |
| R2 | ~~P2~~ | ~~P5~~ | ~~P8~~ P5 |
| R3 | ~~P3~~ | ~~P6~~ | P3 |
| R4 | P0 | | |

```
lw r1, 0(r2)
addi r2, r2, 4
beqz r1, skip
addi r3, r3, 1
bne r2, r4, loop
```

**Physical Register File**

| | | |
|---|---|---|
| P0 | 8016 | p |
| P1 | ~~6823~~ | ~~p~~ |
| P2 | ~~8000~~ | ~~p~~ |
| P3 | 7 | p |
| P4 | 0 | p |
| P5 | 8004 | p |
| P6 | ~~8~~ | ~~p~~ |
| P7 | | |
| P8 | | |
| P9 | | |

**Free List**

~~P4~~
~~P5~~
~~P6~~
~~P7~~
~~P8~~
P9
P1
P2
P8
⋮
P7
P6

**Reorder Buffer (ROB)**

| | use | ex | op | p1 | PR1 | p2 | PR2 | Rd | LPRd | PRd |
|---|---|---|---|---|---|---|---|---|---|---|
| next to commit → | ~~x~~ | ✓ | ~~lw~~ | ~~p~~ | ~~P2~~ | | | ~~r1~~ | ~~P1~~ | ~~P4~~ |
| | ~~x~~ | ✓ | ~~addi~~ | ~~p~~ | ~~P2~~ | | | ~~r2~~ | ~~P2~~ | ~~P5~~ |
| next to commit / next available → | x | | beqz | p | P4 | | | | | |
| | ~~x~~ | ✓ | addi | p | P3 | | | r3 | P3 | P6 |
| next available → | ~~x~~ | | bne | p | P5 | p | P0 | | | |
| | ~~X~~ | | lw | p | P5 | | | r1 | P4 | P7 |
| | ~~X~~ | | addi | p | P5 | | | r2 | P5 | P8 |
| next available → | ~~X~~ | | ~~beqz~~ | | P7 | | | | | |
| | | | | | | | | | | |

**Problem D.1:**

Assume that the following events occur in order (though not necessarily in a single cycle):

**Step 1.** The first three instructions from the next loop iteration (lw, addi, beqz) are written into the ROB (note that the bne instruction has been predicted taken).

**Step 2.** All instructions which are ready after Step 1 execute, write their result to the physical register file, and update the ROB. Note that this step only occurs **once**.

**Step 3.** As many instructions as possible commit.

**Step 4.** The processor detects that the beqz instruction has mispredicted the branch outcome, and recovery action is taken to repair the processor state.

**Update the diagram to reflect the processor state after these events have occurred**. **Cross out** any entries which are no longer valid. Note that the "**ex**" field should be **marked** when an instruction executes, and the "**use**" field should be **cleared** when it commits. Be sure to update the "**next to commit**" and "**next available**" pointers. If the **load** executes, assume that the data value it retrieves is **0**.

**Problem D.2:**

Consider (1) a single-issue, in-order processor with no branch prediction and (2) a multiple-issue, out-of-order processor with branch prediction. Assume that both processors have the same clock frequency. Consider how fast the given loop executes on each processor, assuming that it executes for many iterations.

Under what conditions, if any, might the loop execute at a faster rate on the in-order processor compared to the out-of-order processor?

there are two branches in the code, as in the above case if the branch is mispredicted then all the further instructions have to be flushed - loss of cycles. so if the branch predictor is not accurate enough and the loss of cycles due to flushing > branch resolution cycles then in order might have better performance.

Under what conditions, if any, might the loop execute at a faster rate on the out-of-order processor compared to the in-order processor?

If the branch prediction accuracy is good then out of order performs better because it executes many possible instructions parallely.