## Problem M8.1: Fetch Pipelines

| PC | PC Generation |
|---|---|
| F1 | ICache Access |
| F2 | |
| D1 | Instruction Decode |
| D2 | |
| RN | Rename/Reorder |
| RF | Register File Read |
| EX | Integer Execute |

### Problem M8.1.A — Pipeline Subroutine Returns

Immediately after what pipeline stage does the processor know that it is executing a subroutine return instruction?
D2

Immediately after what pipeline stage does the processor know the subroutine return address?
RF

How many pipeline bubbles are required when executing a subroutine return? 6

### Problem M8.1.B — Adding a BTB

A subroutine can be called from many different locations and thus a single subroutine return can return to different locations. A BTB holds only the address of the last caller.

### Problem M8.1.C — Adding a Return Stack

Normally, instruction fetch needs to wait until the return instruction finishes the RF stage before the return address is known. With the return stack, as soon as the return instruction is decoded in D2, instruction fetch can begin fetching from the return address. This saves 2 cycles.

A return address is pushed after a JAL/JALR instruction is decoded in D2. A return address is popped after a JR r31 instruction is decoded in D2.

## Problem M8.1.D — Return Stack Operation

```
A: JAL B
A+1:
A+2:
…
B: JR r31
B+1:
B+2: …
```

| instruction | | | | | time→ | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | PC | F1 | F2 | D1 | D2 | RN | RF | EX | | | | | | | | | | |
| A+1 | | PC | F1 | F2 | D1 | ~~D2~~ | ~~RN~~ | ~~RF~~ | ~~EX~~ | | | | | | | | | |
| A+2 | | | PC | F1 | F2 | ~~D1~~ | ~~D2~~ | ~~RN~~ | ~~RF~~ | ~~EX~~ | | | | | | | | |
| A+3 | | | | PC | F1 | ~~F2~~ | ~~D1~~ | ~~D2~~ | ~~RN~~ | ~~RF~~ | ~~EX~~ | | | | | | | |
| A+4 | | | | | PC | ~~F1~~ | ~~F2~~ | ~~D1~~ | ~~D2~~ | ~~RN~~ | ~~RF~~ | ~~EX~~ | | | | | | |
| B | | | | | | PC | F1 | F2 | D1 | D2 | RN | RF | EX | | | | | |
| B+1 | | | | | | | PC | F1 | F2 | D1 | ~~D2~~ | ~~RN~~ | ~~RF~~ | ~~EX~~ | | | | |
| B+2 | | | | | | | | PC | F1 | F2 | ~~D1~~ | ~~D2~~ | ~~RN~~ | ~~RF~~ | ~~EX~~ | | | |
| B+3 | | | | | | | | | PC | F1 | ~~F2~~ | ~~D1~~ | ~~D2~~ | ~~RN~~ | ~~RF~~ | ~~EX~~ | | |
| B+4 | | | | | | | | | | PC | ~~F1~~ | ~~F2~~ | ~~D1~~ | ~~D2~~ | ~~RN~~ | ~~RF~~ | ~~EX~~ | |
| A+1 | | | | | | | | | | | PC | F1 | F2 | D1 | D2 | RN | RF | EX |

## Problem M8.1.E — Handling Return Address Mispredicts

When a value is popped off the return stack after D2, it is saved for two cycles as part of the pipeline state. After the RF stage of the return instruction, the actual r31 is compared against the predicted return address. If the addresses match, then we are done. Otherwise we mux in the correct program counter at the PC stage and kill the instructions in F1 and F2. Depending on how fast the address comparison is assumed to be, you might also kill the instruction in D1. So there is an additional 2 or 3 cycles lost on a return mispredict.

## Problem M8.1.F — Further Improving Performance

Ben should add a cache of the most recently encountered return instruction addresses. During F1, the contents of the cache are looked up to see if any entries match the current program counter. If so, then by the end of F1 (instead of D2) we know that we have a return instruction. We can then use the return stack to supply the return address.

# Problem M8.2: Managing Out-of-order Execution

## Problem M8.2.A

**Rename Table**

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

P9

**Physical Regs**

| | | |
|---|---|---|
| | 8016 | p |
| | ~~6823~~ | ~~p~~ |
| | ~~8000~~ | ~~p~~ |
| | 7 | p |
| P4 | 0 | p |
| P5 | 8004 | p |
| P6 | 8 | p |
| P7 | | |
| P8 | | |
| | | |

**Free List**

| |
|---|
| ~~P4~~ |
| ~~P5~~ |
| ~~P6~~ |
| ~~P7~~ |
| ~~P8~~ |
| P9 |
| P1 |
| P2 |
| |

**Reorder Buffer (ROB)**

next to commit → ; next available →

| use | ex | op | p1 | PR1 | p2 | PR2 | Rd | LPRd | PRd |
|---|---|---|---|---|---|---|---|---|---|
| ~~x~~ | x | lw | p | P2 | | | r1 | P1 | P4 |
| ~~x~~ | x | addi | p | P2 | | | r2 | P2 | P5 |
| x | | beqz | p | P4 | | | | | |
| x | x | addi | p | P3 | | | r3 | P3 | P6 |
| x | | bne | p | P5 | p | P0 | | | |
| x | | lw | p | P5 | | | r1 | P4 | P7 |
| x | | addi | p | P5 | | | r2 | P5 | P8 |
| x | | beqz | | P7 | | | | | |
| | | | | | | | | | |

**Problem M8.2.B**

### Rename Table

| | |
|---|---|
| R1 | P4 |
| R2 | P5 |
| R3 | P3 |
| R4 | P0 |

### Physical Regs

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

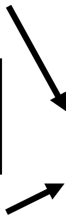
### Free List

| |
|---|
| P9 |
| P1 |
| P2 |
| P6 |
| P7 |
| P8 |
| |
| |
| |

### Reorder Buffer (ROB)

next to commit

next available



| use | ex | op | p1 | PR1 | p2 | PR2 | Rd | LPRd | PRd |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| x | | bne | p | P5 | p | P0 | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

**Problem M8.2.C**

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?

If the out-of-order processor frequently mispredicts either of the branches, it is likely to execute the loop slower than the in-order processor. For this to be true, we must also assume that the branch misprediction penalty of the out-of-order processor is sufficiently longer than the branch resolution delay of the in-order processor, as is likely to be the case. The mispredictions may be due to deficiencies in the out-of-order processor's branch predictor, or the data-dependent branch may be fundamentally unpredictable in nature.

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 out-of-order processor predicts the branches with high enough accuracy, it can execute more than one instruction per cycle, and thereby execute the loop at a faster rate than the in-order processor.

## Problem M8.3: Exceptions and Register Renaming

### Problem M8.3.A — Recovering from Exceptions

By the definition of a precise exception, an exception that occurs in the middle of an x86 instruction should cause the machine state to revert to the state that previously existed right before the excepting instruction started executing. Thus a strategy to determine a precise state would be to take snapshots of the RAT only on x86 instruction boundaries (either when the last µop of an x86 instruction commits or right before the first µop of an x86 instruction is renamed).

### Problem M8.3.B — Minimizing Snapshots

Ben is correct. Since an exception causes the machine to revert to the state found on an x86 instruction boundary, all the temporary state used by the µops does not need to be kept. Thus, the RAT only has to hold the rename mappings for the architectural registers, and not for T0-T7.

### Problem M8.3.C — Renaming Registers

There must be at least 17 physical registers for the Bentium 4 to work properly. 16 registers are needed to hold the state of the machine at any given point in time (architectural and temporary register values), and an extra one is needed to rename an additional register using the given renaming algorithm to allow forward progress.

## Problem M8.4: Out-of-order Execution (Spring 2014 Quiz 2, Part C)

In this problem, we are going to update the state of the processor when different events happen. You are given an out-of-order processor in some initial state, as described by the registers (renaming table, physical registers, and free list), one-bit branch predictor, and re-order buffer. Your job is to show the changes that occur when some event occurs, starting from the same initial state except where noted. For partial credit, briefly describe what changes occur.

*Last updated:*
3/10/2020

**Problem M8.4.A**

Show the state of the processor if the first load completes (but does not commit).

INSTRUCTIONS

```
   00: LD R1, 0(R2)
   04: ADD R3, R1, R4
   08: ADD R2, R1, R2
   0c: BGEZ R4, A
   10: LD R3, 0(R2)
A: 14: SUB R1, R3, R2
   18: ADD R4, R3, R1
```

| BRANCH PREDICTOR | |
|---|---|
| 00 | 1 |
| 01 | 0 |
| 10 | 1 |
| 11 | 0 |

| RENAMING TABLE | |
|---|---|
| R1 | P4 |
| R2 | P6 |
| R3 | P5 |
| R4 | P3 |

| PHYS. REG. FILE | | |
|---|---|---|
| P0 | (R1) | p |
| P1 | (R2) | p |
| P2 | (R3) | p |
| P3 | (R4) | p |
| P4 | | p |
| P5 | | |
| P6 | | |
| P7 | | |

| FREE LIST |
|---|
| P7 |
| |
| |
| |
| |
| |
| |
| |

RE-ORDER BUFFER (ROB)

| | Use? | Ex | Op | P1 | PR1 | P2 | PR2 | Rd | LPRd | PRd |
|---|---|---|---|---|---|---|---|---|---|---|
| Next to commit → | X | X | LD | p | P1 | | | R1 | P0 | P4 |
| | X | | ADD | p | P4 | p | P3 | R3 | P2 | P5 |
| | X | | ADD | p | P4 | p | P1 | R2 | P1 | P6 |
| | X | | BGEZ | p | P3 | | | | | |
| Next available → | | | | | | | | | | |
| | | | | | | | | | | |

## Problem M8.4.B

Show the state of the processor after the next instruction is issued.

```
    00: LD R1, 0(R2)
    04: ADD R3, R1, R4
    08: ADD R2, R1, R2
    0c: BGEZ R4, A
    10: LD R3, 0(R2)
A:  14: SUB R1, R3, R2
    18: ADD R4, R3, R1
```

INSTRUCTIONS

| BRANCH PREDICTOR | |
|---|---|
| 00 | 1 |
| 01 | 0 |
| 10 | 1 |
| 11 | 0 |

| RENAMING TABLE | |
|---|---|
| R1 | P4 |
| R2 | P6 |
| R3 | ~~P5~~ P7 |
| R4 | P3 |

| PHYS. REG. FILE | | |
|---|---|---|
| P0 | (R1) | p |
| P1 | (R2) | p |
| P2 | (R3) | p |
| P3 | (R4) | p |
| P4 | | |
| P5 | | |
| P6 | | |
| P7 | | |

| FREE LIST |
|---|
| ~~P7~~ |
| |
| |
| |
| |
| |
| |
| |

RE-ORDER BUFFER (ROB)

| | Use? | Ex | Op | P1 | PR1 | P2 | PR2 | Rd | LPRd | PRd |
|---|---|---|---|---|---|---|---|---|---|---|
| Next to commit → | X | | LD | p | P1 | | | R1 | P0 | P4 |
| | X | | ADD | | P4 | p | P3 | R3 | P2 | P5 |
| | X | | ADD | | P4 | p | P1 | R2 | P1 | P6 |
| | X | | BGEZ | p | P3 | | | | | |
| | X | | LD | | P6 | | | R3 | P5 | P7 |
| Next available → | | | | | | | | | | |

**Problem M8.4.C**

From the state at the end of Question 2, as the next action can the processor issue (not execute) another instruction?

No. There are no physical registers on the free list.

In one or two sentences, what does this say about our design? How can we improve it?

We didn’t solve Little’s Law correctly when we sized our physical register file. We need to make it bigger so it can support the number of instructions we have in flight in the ROB.

**Problem M8.4.D**

Show the state of the processor if the first `LD` triggers a page fault and after abort finishes.

INSTRUCTIONS

```
    00: LD R1, 0(R2)
    04: ADD R3, R1, R4
    08: ADD R2, R1, R2
    0c: BGEZ R4, A
    10: LD R3, 0(R2)
A:  14: SUB R1, R3, R2
    18: ADD R4, R3, R1
```

| BRANCH PREDICTOR | |
|---|---|
| 00 | 1 |
| 01 | 0 |
| 10 | 1 |
| 11 | 0 |

| RENAMING TABLE | |
|---|---|
| R1 | ~~P4~~ P0 |
| R2 | ~~P6~~ P1 |
| R3 | ~~P5~~ P2 |
| R4 | P3 |

| PHYS. REG. FILE | | |
|---|---|---|
| P0 | (R1) | p |
| P1 | (R2) | p |
| P2 | (R3) | p |
| P3 | (R4) | p |
| P4 | | |
| P5 | | |
| P6 | | |
| P7 | | |

| FREE LIST |
|---|
| P7 |
| P4 |
| P5 |
| P6 |
| |
| |
| |
| |

Next available →
Next to commit →

| RE-ORDER BUFFER (ROB) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Use? | Ex | Op | P1 | PR1 | P2 | PR2 | Rd | LPRd | PRd |
| ~~X~~ | | ~~LD~~ | ~~p~~ | ~~P1~~ | | | ~~R1~~ | ~~P0~~ | ~~P4~~ |
| ~~X~~ | | ~~ADD~~ | | ~~P4~~ | ~~p~~ | ~~P3~~ | ~~R3~~ | ~~P2~~ | ~~P5~~ |
| ~~X~~ | | ~~ADD~~ | | ~~P4~~ | ~~p~~ | ~~P1~~ | ~~R2~~ | ~~P1~~ | ~~P6~~ |
| ~~X~~ | | ~~BGEZ~~ | ~~p~~ | ~~P3~~ | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

## Problem M8.5 (Spring 2015 Quiz 2, Part B)

You are given an out-of-order processor that

- Issues at most one instruction per cycle
- Commits at most one instruction per cycle
- Uses an unified physical register file

### Problem M8.5.A

Consider the following code sequence:

```
        Addr
I0      (0x24)        lw      r2, (r4), #0
I1      (0x28)        addi    r2, r2,   #16
I2      (0x2C)        lw      r3, (r4), #4
I3      (0x30)        blez    r3, L1
I4      (0x34)        addi    r4, r2, #8
I5      (0x38)        mul     r1, r2, r3
I6      (0x3C)        addi    r3, r2, #8
I7      (0x40)    L1: add     r2, r1, r3
```

Assume the branch instruction (blez) is not taken. 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.

Older Instruction (columns); Younger Instruction (rows)

| | I0 | I1 | I2 | I3 | I4 | I5 | I6 | I7 |
|---|---|---|---|---|---|---|---|---|
| I0 | - | | | | | | | |
| I1 | WAW<br>RAW | - | | | | | | |
| I2 | | | - | | | | | |
| I3 | | | RAW | - | | | | |
| I4 | WAR | RAW | WAR | | - | | | |
| I5 | | RAW | RAW | | | - | | |
| I6 | | RAW | WAW | WAR | | WAR | - | |
| I7 | | WAW | | | WAR | WAR<br>RAW | WAR<br>RAW | - |

In Problems M8.5.B to M8.5.D, you should update the state of the processor when different events happen. The starting state in each question is the same, and the event specified in each question is the ONLY event that takes place for that question. The starting state is shown in the different structures: renaming table, physical registers, free list, two-bit branch predictor, global history buffer, and reorder buffer (ROB).

Note the following conventions:

- The valid bit for any entry is represented by "1".
- The valid bit can be cleared by crossing it out.
- In the ROB, the "ex" field should be marked with "1" when an instruction starts execution, and the "use" field should be cleared when it commits. Be sure to update the "next to commit" and "next available" pointers, if necessary.
- Fill out the "after" fields in all the tables. Write new values in these boxes if the values change due to the event specified in the question. You do not have to repeat the values if they do not change due to the event.

In Questions 2 through 4, we will use the same code sequence as in Question 1:

```
     Addr
I0   (0x24)       lw     r2, (r4), #0
I1   (0x28)       addi   r2, r2,   #16
I2   (0x2C)       lw     r3, (r4), #4
I3   (0x30)       blez   r3, L1
I4   (0x34)       addi   r4, r2, #8
I5   (0x38)       mul    r1, r2, r3
I6   (0x3C)       addi   r3, r2, #8
I7   (0x40)   L1: add    r2, r1, r3
```

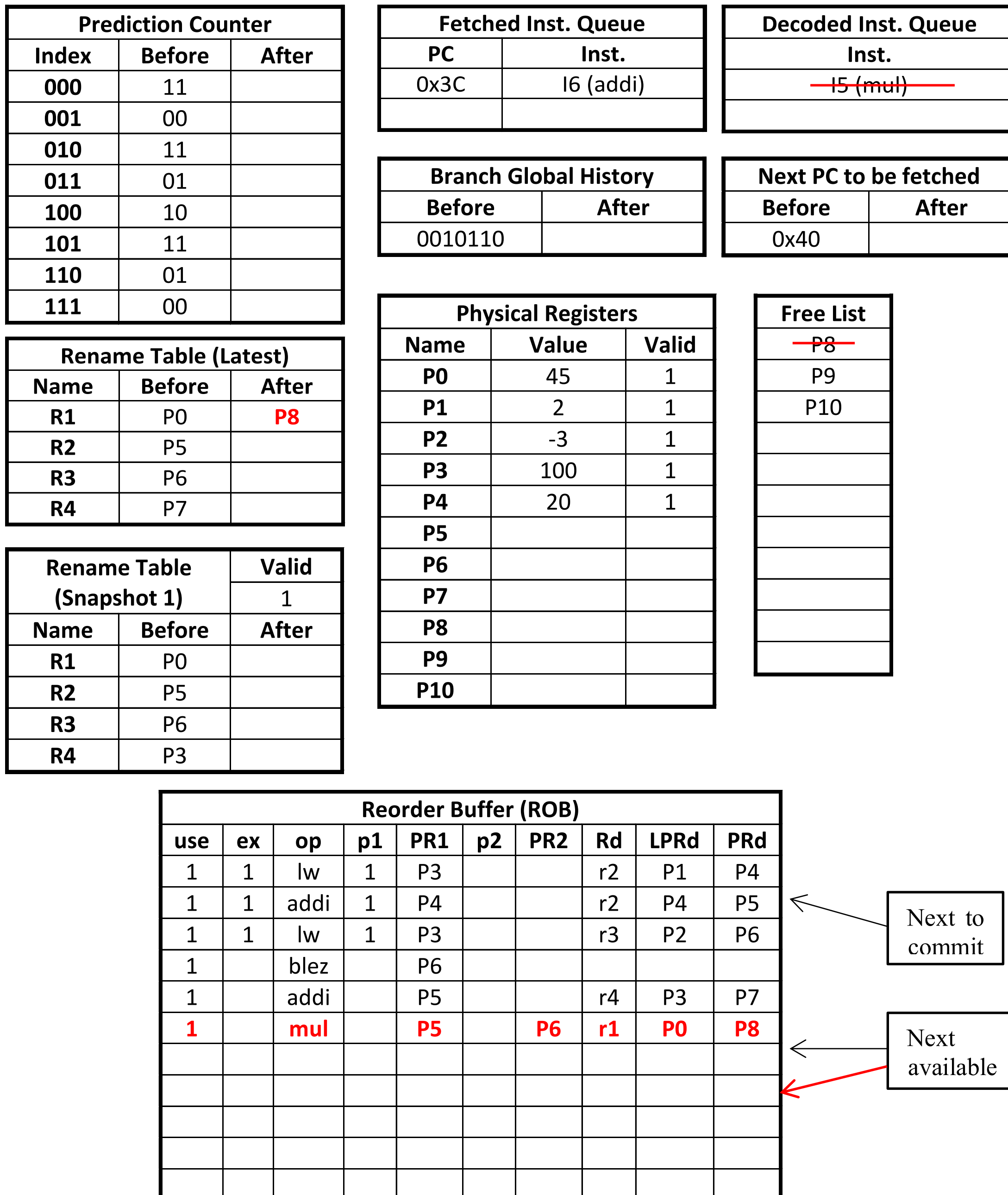
The starting state of the processor is as follows:

- Instructions I0-I4 are already in the ROB.
- I0 (lw) has already finished execution.
- I1 (addi) and I2 (lw) have started executing but have not finished yet.
- I3 (blez) has been predicted to be Not-Taken by the branch predictor.
- I5 (mul) has completed the decode stage.
- I6 (addi) has completed the Fetch Stage.
- The next PC is set to 0x40, which is the PC of I7 (add).

*Last updated:*
3/10/2020

**Problem M8.5.B**

The following figure shows the starting state of the processor. Suppose the decoded instruction I5 (mul) is now inserted into the ROB. Update the diagram to reflect the processor state after this event has occurred.



| Prediction Counter | | |
|---|---|---|
| **Index** | **Before** | **After** |
| **000** | 11 | |
| **001** | 00 | |
| **010** | 11 | |
| **011** | 01 | |
| **100** | 10 | |
| **101** | 11 | |
| **110** | 01 | |
| **111** | 00 | |

| Fetched Inst. Queue | |
|---|---|
| **PC** | **Inst.** |
| 0x3C | I6 (addi) |
| | |

| Decoded Inst. Queue |
|---|
| **Inst.** |
| ~~I5 (mul)~~ |
| |

| Branch Global History | |
|---|---|
| **Before** | **After** |
| 0010110 | |

| Next PC to be fetched | |
|---|---|
| **Before** | **After** |
| 0x40 | |

| Rename Table (Latest) | | |
|---|---|---|
| **Name** | **Before** | **After** |
| **R1** | P0 | P8 |
| **R2** | P5 | |
| **R3** | P6 | |
| **R4** | P7 | |

| Physical Registers | | |
|---|---|---|
| **Name** | **Value** | **Valid** |
| **P0** | 45 | 1 |
| **P1** | 2 | 1 |
| **P2** | -3 | 1 |
| **P3** | 100 | 1 |
| **P4** | 20 | 1 |
| **P5** | | |
| **P6** | | |
| **P7** | | |
| **P8** | | |
| **P9** | | |
| **P10** | | |

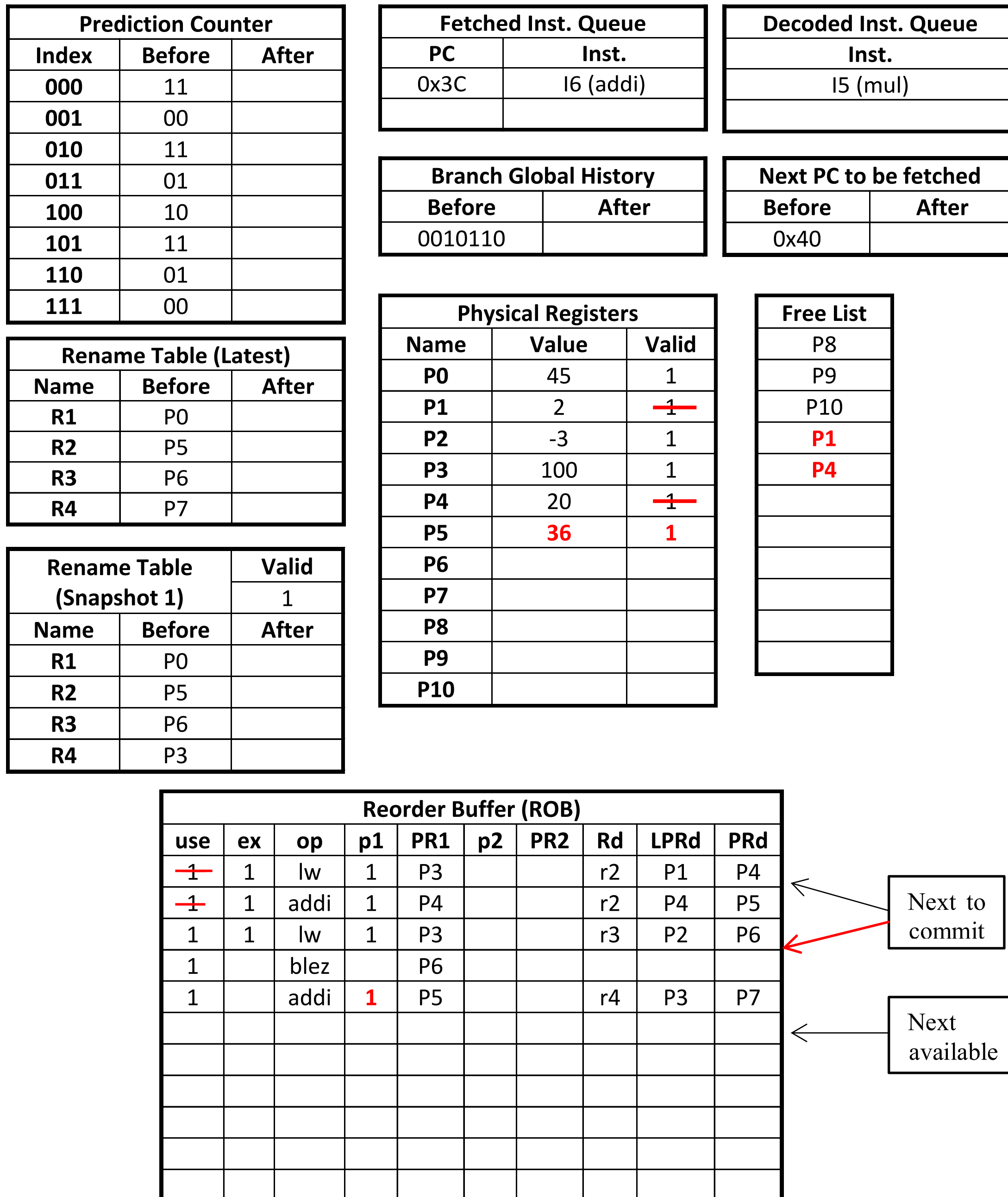
| Free List |
|---|
| ~~P8~~ |
| P9 |
| P10 |
| |
| |
| |
| |
| |
| |
| |
| |

| Rename Table (Snapshot 1) | | Valid |
|---|---|---|
| | | 1 |
| **Name** | **Before** | **After** |
| **R1** | P0 | |
| **R2** | P5 | |
| **R3** | P6 | |
| **R4** | P3 | |

| Reorder Buffer (ROB) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **use** | **ex** | **op** | **p1** | **PR1** | **p2** | **PR2** | **Rd** | **LPRd** | **PRd** |
| 1 | 1 | lw | 1 | P3 | | | r2 | P1 | P4 |
| 1 | 1 | addi | 1 | P4 | | | r2 | P4 | P5 |
| 1 | 1 | lw | 1 | P3 | | | r3 | P2 | P6 |
| 1 | | blez | | P6 | | | | | |
| 1 | | addi | | P5 | | | r4 | P3 | P7 |
| 1 | | mul | | P5 | | P6 | r1 | P0 | P8 |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

**Problem M8.5.C**

Start from the same processor state, shown below. Suppose now I1 (addi) has completed execution. Commit as many instructions as possible. Update the diagram to reflect the processor state after I1 execution completes and as many instructions as possible have committed. Again, assume no other events take place.



| Prediction Counter | | |
|---|---|---|
| **Index** | **Before** | **After** |
| **000** | 11 | |
| **001** | 00 | |
| **010** | 11 | |
| **011** | 01 | |
| **100** | 10 | |
| **101** | 11 | |
| **110** | 01 | |
| **111** | 00 | |

| Fetched Inst. Queue | |
|---|---|
| **PC** | **Inst.** |
| 0x3C | I6 (addi) |
| | |

| Decoded Inst. Queue |
|---|
| **Inst.** |
| I5 (mul) |
| |

| Branch Global History | |
|---|---|
| **Before** | **After** |
| 0010110 | |

| Next PC to be fetched | |
|---|---|
| **Before** | **After** |
| 0x40 | |

| Rename Table (Latest) | | |
|---|---|---|
| **Name** | **Before** | **After** |
| **R1** | P0 | |
| **R2** | P5 | |
| **R3** | P6 | |
| **R4** | P7 | |

| Physical Registers | | |
|---|---|---|
| **Name** | **Value** | **Valid** |
| **P0** | 45 | 1 |
| **P1** | 2 | ~~1~~ |
| **P2** | -3 | 1 |
| **P3** | 100 | 1 |
| **P4** | 20 | ~~1~~ |
| **P5** | 36 | 1 |
| **P6** | | |
| **P7** | | |
| **P8** | | |
| **P9** | | |
| **P10** | | |

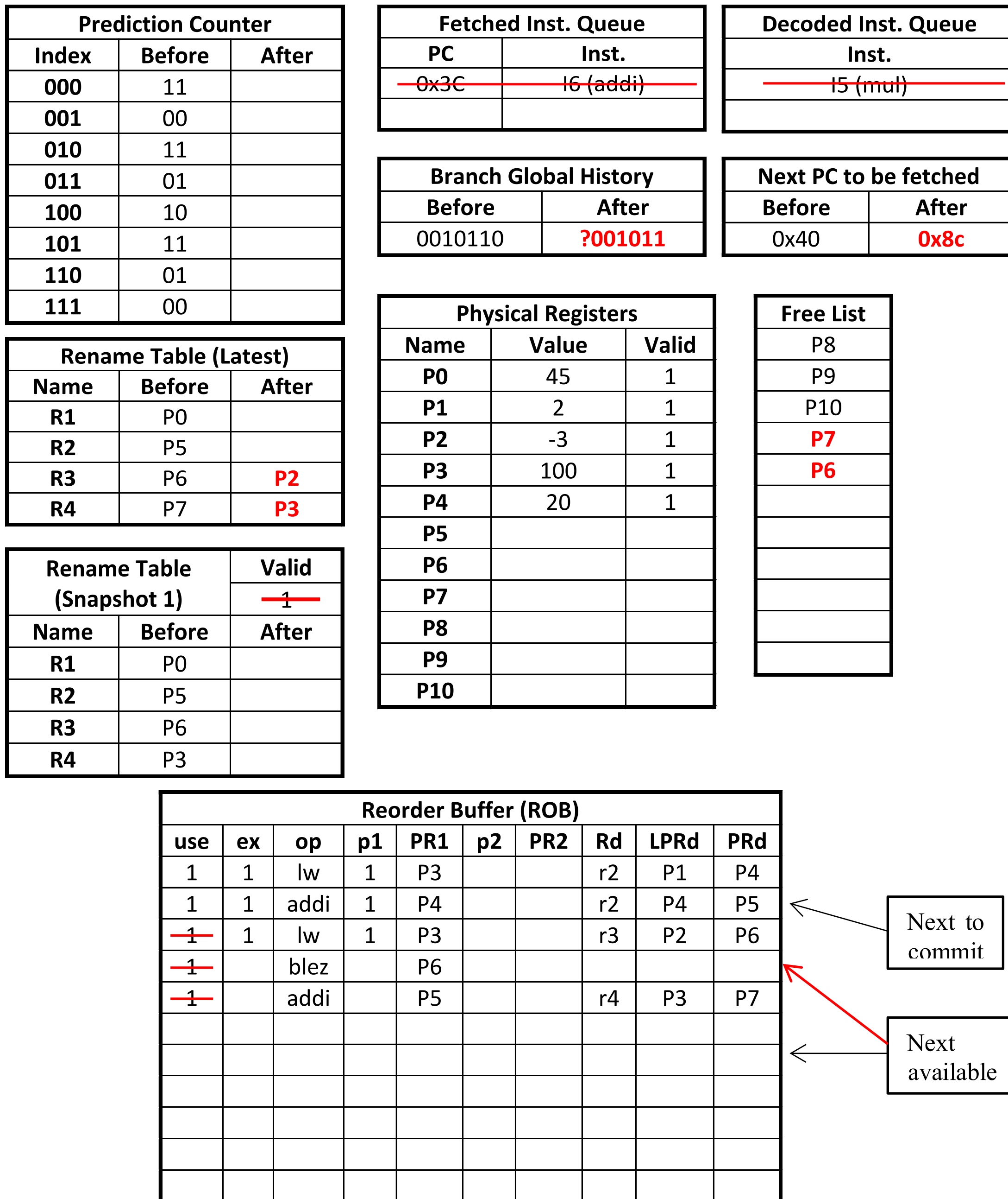
| Free List |
|---|
| P8 |
| P9 |
| P10 |
| P1 |
| P4 |
| |
| |
| |
| |
| |
| |

| Rename Table (Snapshot 1) | | Valid |
|---|---|---|
| | | 1 |
| **Name** | **Before** | **After** |
| **R1** | P0 | |
| **R2** | P5 | |
| **R3** | P6 | |
| **R4** | P3 | |

| Reorder Buffer (ROB) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **use** | **ex** | **op** | **p1** | **PR1** | **p2** | **PR2** | **Rd** | **LPRd** | **PRd** |
| ~~1~~ | 1 | lw | 1 | P3 | | | r2 | P1 | P4 |
| ~~1~~ | 1 | addi | 1 | P4 | | | r2 | P4 | P5 |
| 1 | 1 | lw | 1 | P3 | | | r3 | P2 | P6 |
| 1 | | blez | | P6 | | | | | |
| 1 | | addi | 1 | P5 | | | r4 | P3 | P7 |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

## Problem M8.5.D

Start from the same processor state, shown below. Suppose instruction I2 (lw) triggers an ALU overflow exception. Restore the architectural and microarchitectural state to recover from misspeculation. The exception handler for the processor is at address 0x8C (control is transferred to the exception handler after recovery). You do not need to worry about the number of cycles taken by recovery. Show the processor state after recovery.



| Prediction Counter | | |
|---|---|---|
| **Index** | **Before** | **After** |
| **000** | 11 | |
| **001** | 00 | |
| **010** | 11 | |
| **011** | 01 | |
| **100** | 10 | |
| **101** | 11 | |
| **110** | 01 | |
| **111** | 00 | |

| Fetched Inst. Queue | |
|---|---|
| **PC** | **Inst.** |
| ~~0x3C~~ | ~~I6 (addi)~~ |
| | |

| Decoded Inst. Queue |
|---|
| **Inst.** |
| ~~I5 (mul)~~ |
| |

| Branch Global History | |
|---|---|
| **Before** | **After** |
| 0010110 | ?001011 |

| Next PC to be fetched | |
|---|---|
| **Before** | **After** |
| 0x40 | 0x8c |

| Rename Table (Latest) | | |
|---|---|---|
| **Name** | **Before** | **After** |
| **R1** | P0 | |
| **R2** | P5 | |
| **R3** | P6 | P2 |
| **R4** | P7 | P3 |

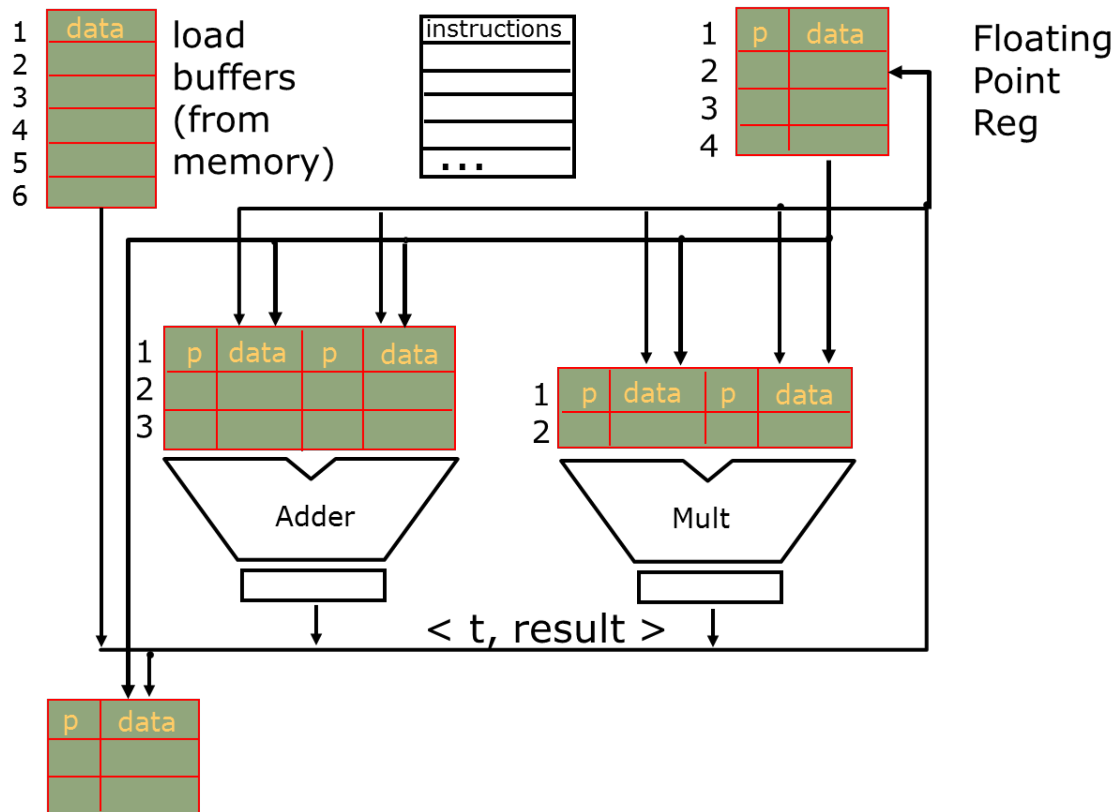
| Rename Table (Snapshot 1) | | **Valid** ~~1~~ |
|---|---|---|
| **Name** | **Before** | **After** |
| **R1** | P0 | |
| **R2** | P5 | |
| **R3** | P6 | |
| **R4** | P3 | |

| Physical Registers | | |
|---|---|---|
| **Name** | **Value** | **Valid** |
| **P0** | 45 | 1 |
| **P1** | 2 | 1 |
| **P2** | -3 | 1 |
| **P3** | 100 | 1 |
| **P4** | 20 | 1 |
| **P5** | | |
| **P6** | | |
| **P7** | | |
| **P8** | | |
| **P9** | | |
| **P10** | | |

| Free List |
|---|
| P8 |
| P9 |
| P10 |
| P7 |
| P6 |
| |
| |
| |
| |
| |
| |

| Reorder Buffer (ROB) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **use** | **ex** | **op** | **p1** | **PR1** | **p2** | **PR2** | **Rd** | **LPRd** | **PRd** |
| 1 | 1 | lw | 1 | P3 | | | r2 | P1 | P4 |
| 1 | 1 | addi | 1 | P4 | | | r2 | P4 | P5 |
| ~~1~~ | 1 | lw | 1 | P3 | | | r3 | P2 | P6 |
| ~~1~~ | | blez | | P6 | | | | | |
| ~~1~~ | | addi | | P5 | | | r4 | P3 | P7 |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

# Problem M8.6: Out-of-order Processor Design (Spring 2014 Quiz 2, Part D)




You are designing an out-of-order processor similar to the IBM 360/91 Tomasulo design shown above. This design distributes the re-order buffer around the processor, placing entries near their associated functional units. In such a design, the distributed ROB entries are called "reservation stations". Entries are allocated when the instruction is decoded and freed when the instruction is dispatched to the functional unit.

Your design achieves an average throughput of 1.5 instructions per cycle. Two-thirds of instructions are adds, and one-third are multiplies. The latency of each instruction type *from allocation to completion* is 5 cycles for adds and 14 cycles for multiplies.

| | | |
|---|---|---|
| **Add** | 2/3 | 5 |
| **Multiply** | 1/3 | 14 |

The adder and multiplier are each fully pipelined with full bypassing. *Once an instruction is dispatched to the FU*, the adder takes 2 cycles and the multiplier takes 5 cycles.

| | | |
|---|---|---|
| 1.5 | 2 | 5 |

**Problem M8.6.A**

How many entries are in use, on average, in the reservation station at each functional unit (adder, multiplier) in the steady state? Assume there are infinite entries available if needed. What is the average latency of an instruction in this machine? *For partial credit, feel free to give any formulae you believe may be important to answer this question.*

This is a Little's Law question: $T = N / L$.

From the fraction of instructions and the machine's total throughput, we can get the throughput of each type of instruction.

$T_{add} = 2/3 * 3/2 = 1$
$T_{mul} = 1/3 * 3/2 = 1/2$

To solve for the number of entries in use, we need to know the average latency an instruction spends in the reservation station. From the problem description, reservation stations are in use from allocation until the instruction is dispatched to the functional unit. So the latency in the reservation station itself is the end-to-end latency minus the latency of the functional unit.

$L_{r,add} = L_{add} – L_{fu,add} = 5 – 2 = 3$ cycles
$L_{r,mul} = L_{mul} – L_{fu,mul} = 14 – 5 = 9$ cycles

Thus the number of entries in use is on average:

$N_{add} = T_{add} * L_{r,add} = 3$
$N_{mul} = T_{mul} * L_{r,mul} = 9 / 2 = 4.5$

The average latency can be computed from the frequency of instructions directly:

$L = 2/3\ L_{add} + 1/3\ L_{mul} = 2/3 * 5 + 1/3 * 14 = 8$

Or from Little's Law, but this is more complicated. We now want to know the number of adds and multiplies in flight. This is the number of entries plus the number of instructions in the FU themselves. The adder has an issue rate of 1, so the adder is always full. The multiplier has an issue rate of ½, so it is half full. Therefore:

$L = N / T = (3 + 2 + 4.5 + 5/2) / 1.5 = 8$

It's nice to see that they agree, but really the first formulation is much e