MP Out-Of-Order

Presented by Scott & Ian
Memes: Ethan/Udit/Jason

- CP1 deliverables expectations [Scott]
  - Design
    - Interface with memory
    - Each stage
    - Be able to trace every instruction, all arrows need high level field specifications
  - Queue
    - Queue can push & Pop
    - Queue indicates when full
  - Fetch
    - Magic memory, into a queue that stores instruction and PC
- How to make a Queue/Fetch [Ian]
  - How to make head/tail ptr queue, maybe say why it's better than shifting (power)
  - How to integrate it with fetch
  - Expected interface (push/pop)
- Tomasulo Design (briefly) [Scott]
  - Show a basic design (from lecture)
  - Explain verbally what is missing that they would want to add to their diagram
  - Control/Speculative execution
    - Both: wait until control instruction is head of ROB
    - Tomasulo: Flush everything
  - Nomenclature - dispatch vs issue
  - Memory - pretend it's another function unit, with just a single reservation station that doesn't deallocate until commit/retire
    - 2 steps
    - Stores commit in order when they are the head of the ROB
    - Loads get the correct data - youngest store older than the load to the same address
- ERR [Ian]
  - High level block description
  - Go through example instruction execution
  - Memory instructions - complications due to ERR - can't be unified with other instructions
  - Control Flush everything, copy RRF into RAT, reset Free List head/tail pointers.

Checkpoint 1 Logistics

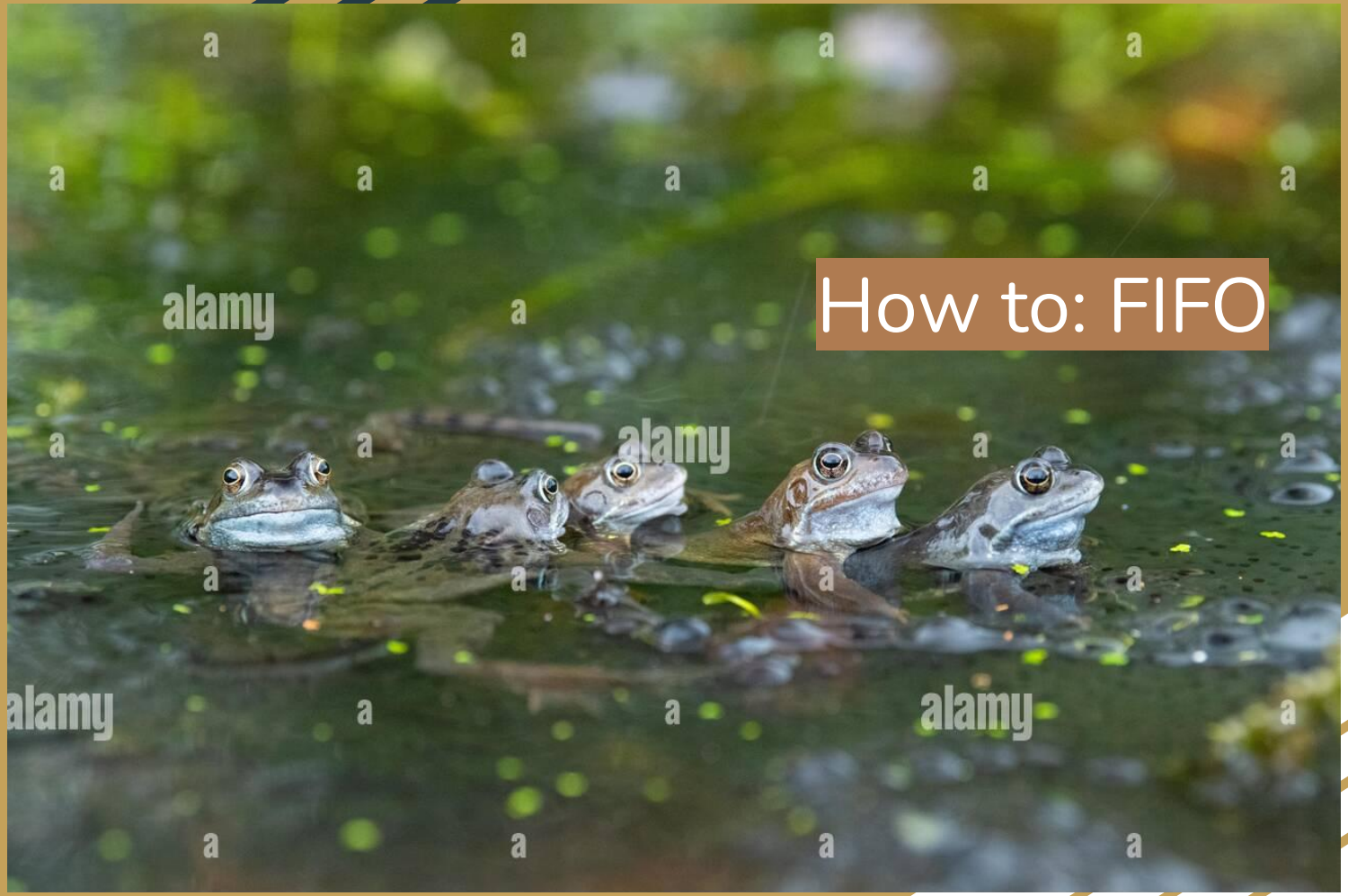
CP1 Deadline Information (3/25 11:59pm)

Requirements:

- *Design [5]*: Design a high-level block diagram detailing the datapath of your OoO
  - Must be digital - use draw.io or a similar tool. Cannot be handwritten
- *Hardware Submission [12.5]:*
  - Implement a parameterized FIFO with variable depth and width [5]
  - Implement the fetch stage. Must be integrated with magic memory and your parameterized FIFO [7.5]
    - PC initializes to 0x60000000
- Roadmap + Progress Report [2.5]

Total of 20 points.

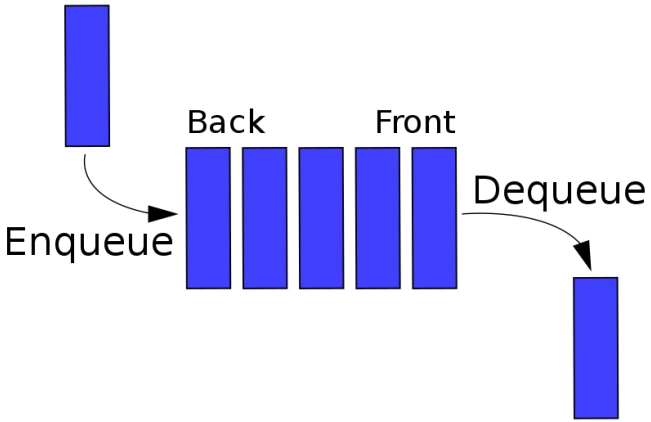
Schedule a meeting with your assigned mentor TA to demonstrate the above!!!!



How to: FIFO

What is a Queue/FIFO?

First **I**n, **F**irst **O**ut

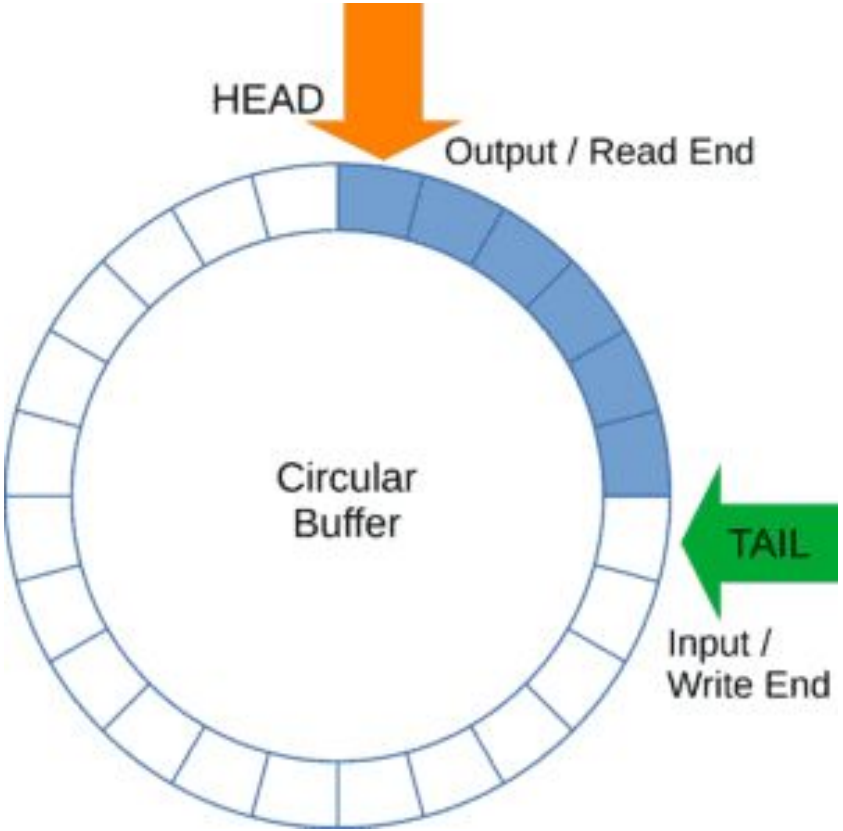


What does a FIFO look like in hardware?

Big buffer (registers or sram) store data

Internal head and tail pointers

- Read from head, increment head when dequeuing
- Write to tail, increment tail when enqueuing



How do we control this FIFO?

`input enqueue` - Control signal to enqueue **`enqueue_wdata`**

`input enqueue_wdata` - Input data to enqueue

`input dequeue` - Control signal to dequeue **`dequeue_rdata`**

`output dequeue_rdata` - Output data to dequeue

Are we missing anything?

When is the FIFO full or empty?

```
logic [DATA_WIDTH-1:0] entries [16];

logic [3:0] head_ptr;
logic [3:0] tail_ptr;
```

Head_ptr == Tail_ptr → We know that the FIFO is either full or empty

```
assign full = (head_ptr == tail_ptr);
assign empty = (head_ptr == tail_ptr);
```

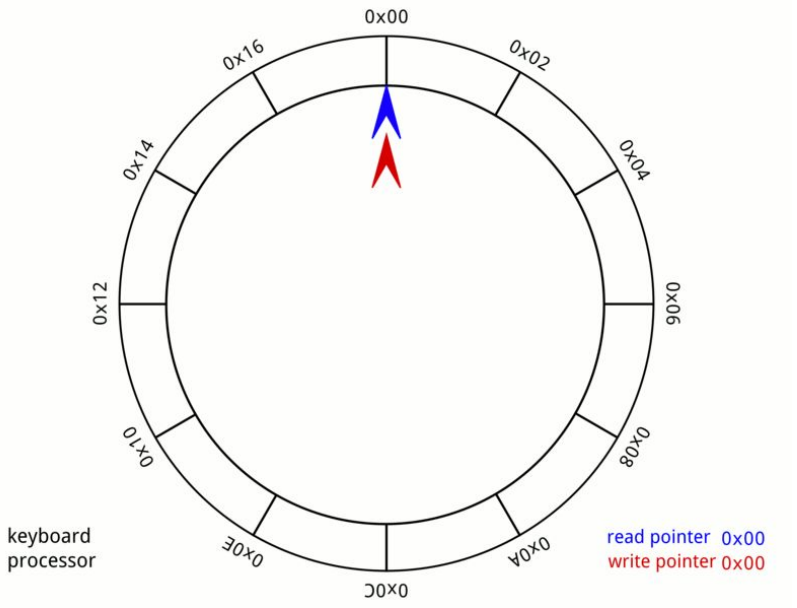
How do we distinguish between the two?

Full and Empty (cont)

Detect whether the tail “lapped” the head.

The tail_ptr would *overflow* before reaching the head pointer

Let’s add another bit!



Example for a circular buffer found on wikipedia. Head is read pointer, tail is write pointer.

Full and Empty (cont cont)

```
logic [DATA_WIDTH-1:0] entries [16];

logic [4:0] head_ptr;
logic [4:0] tail_ptr;
```

```
assign full = (
    (head_ptr[3:0] == tail_ptr[3:0]) &&
    (head_ptr[4] != tail_ptr[4])
);
```

With that additional bit, how do we detect full?

Fetch to Decode

Make sure fetch stalls when the FIFO is full!
Why did we do this?

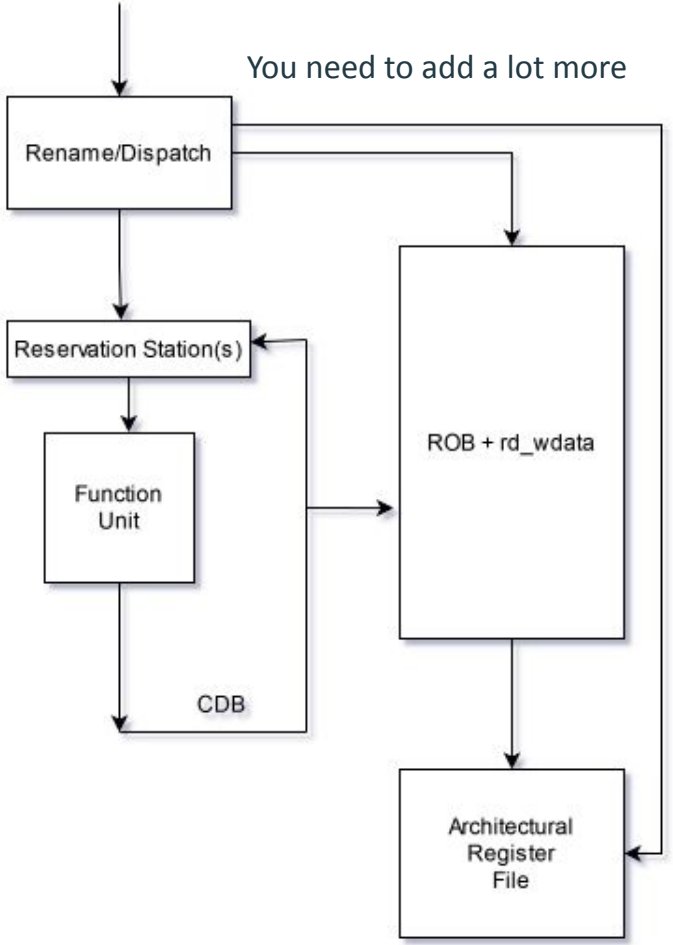
Tomasulo’s Algorithm

Tomasulo's Algorithm & OOO

Motivations:

- Tomasulo leverages instruction-level parallelism to achieve higher performance than in-order pipelines
  - Tomasulo's algorithm can schedule instructions without dependencies in parallel or otherwise out-of-order.
- Tomasulo is relatively simpler (conceptually) compared to Explicit Register Renaming (R10k style)
- Simpler designs can be developed faster, leaving more time for debugging and/or advanced features
  - More on the pros of ERR (and how it works) in later slides

You need to add a lot more



Some Nomenclature

Fetch: Grab instructions from memory

Decode: Parse the instructions and determine operands, immediates, etc.

Dispatch: Put instructions into ROB and RS. Wait for source registers to be ready.

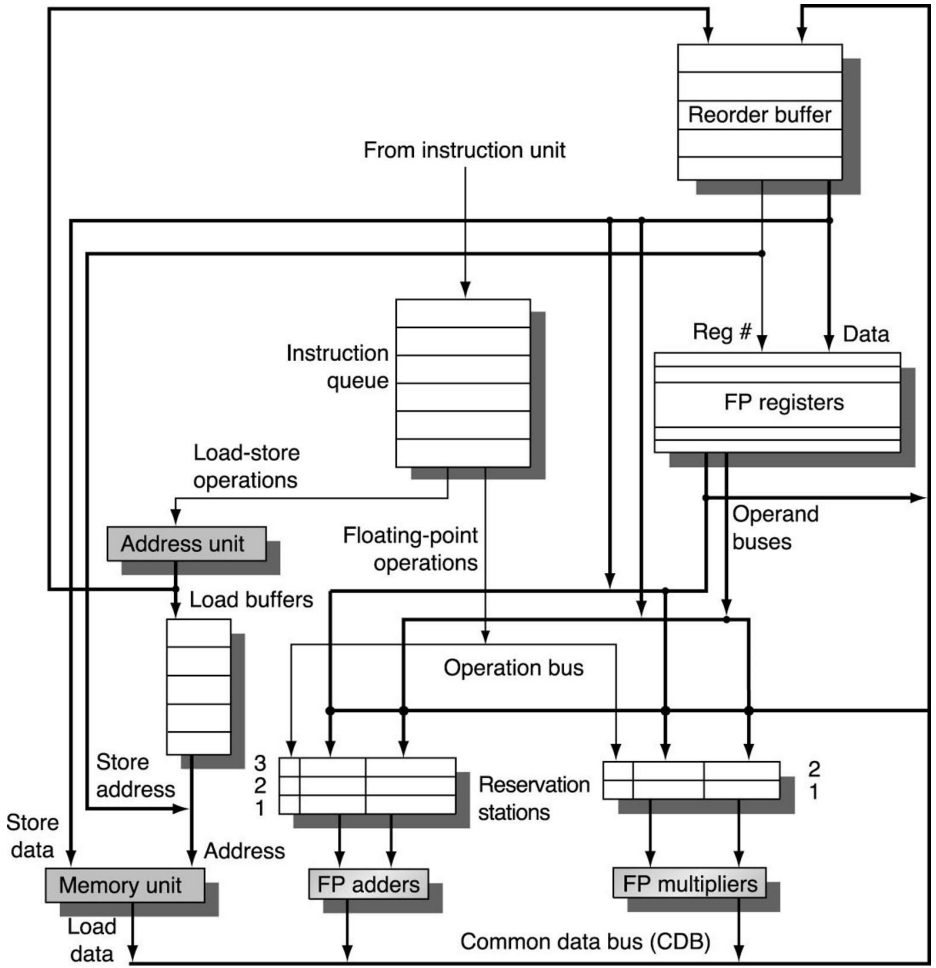
Issue: Once the source register values are ready, send the instruction to the appropriate execution unit

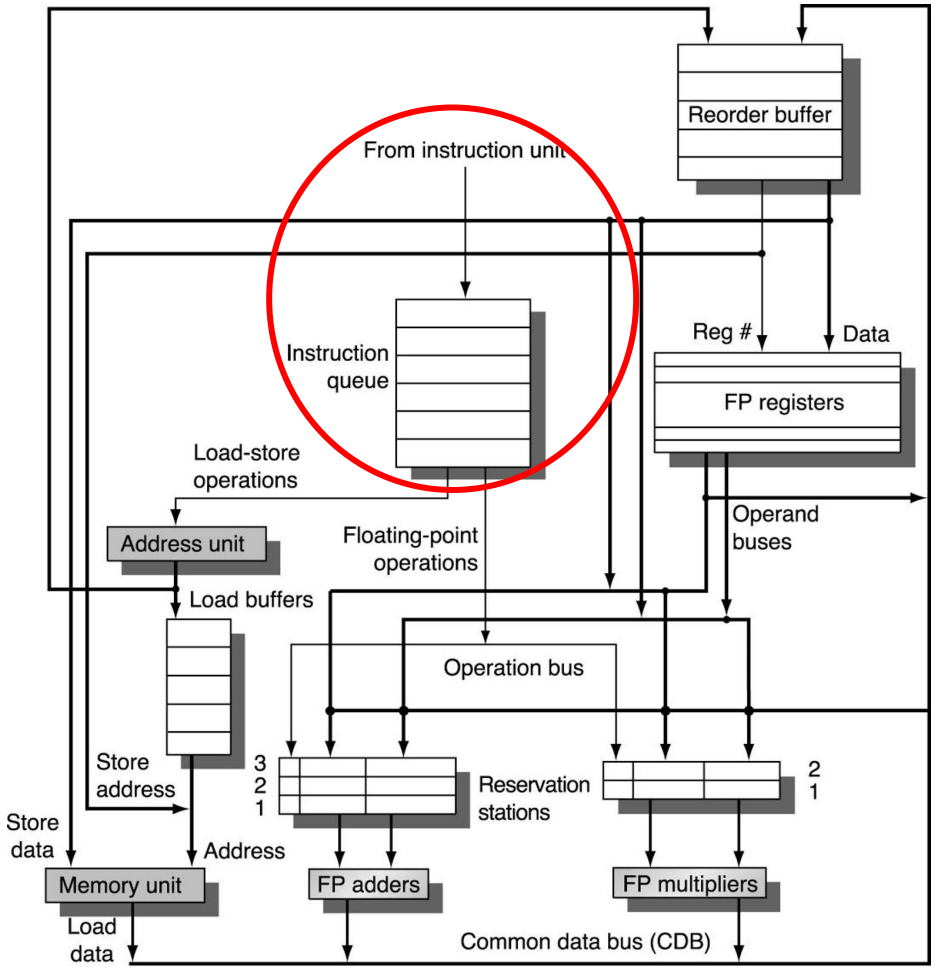
Execute: Do the execution (mults, adds, etc.)

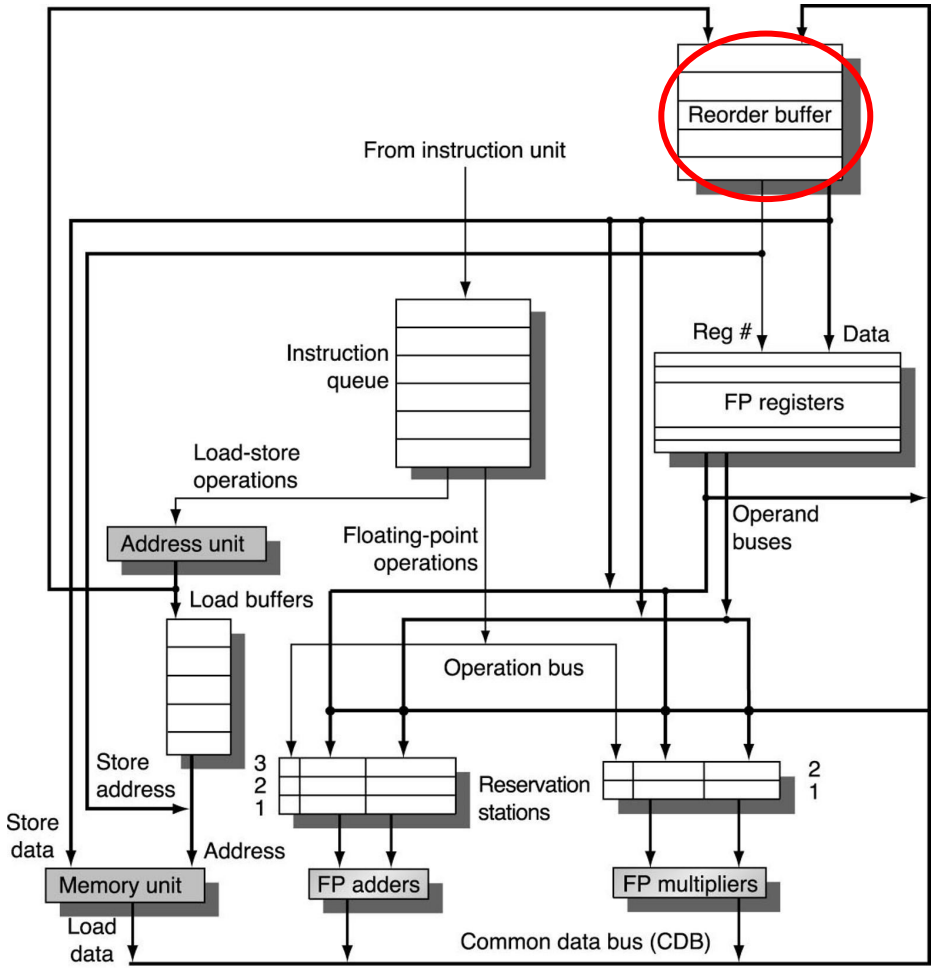
Writeback: Broadcast the value calculated during execute on the CDB

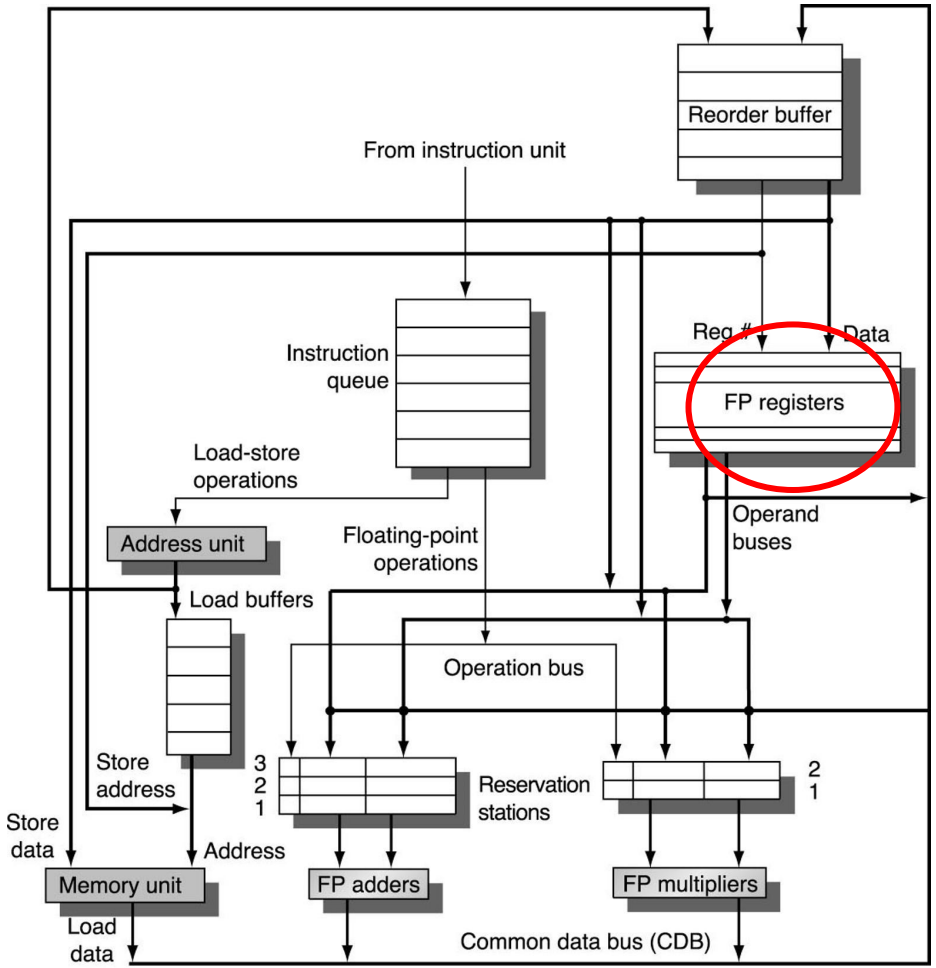
Commit/Retire: Write the oldest value in the ROB into the architectural register file (Tomasulo) or update the RRF (explicit renaming). Also store to memory if committing a store instruction.

You need to add a lot more

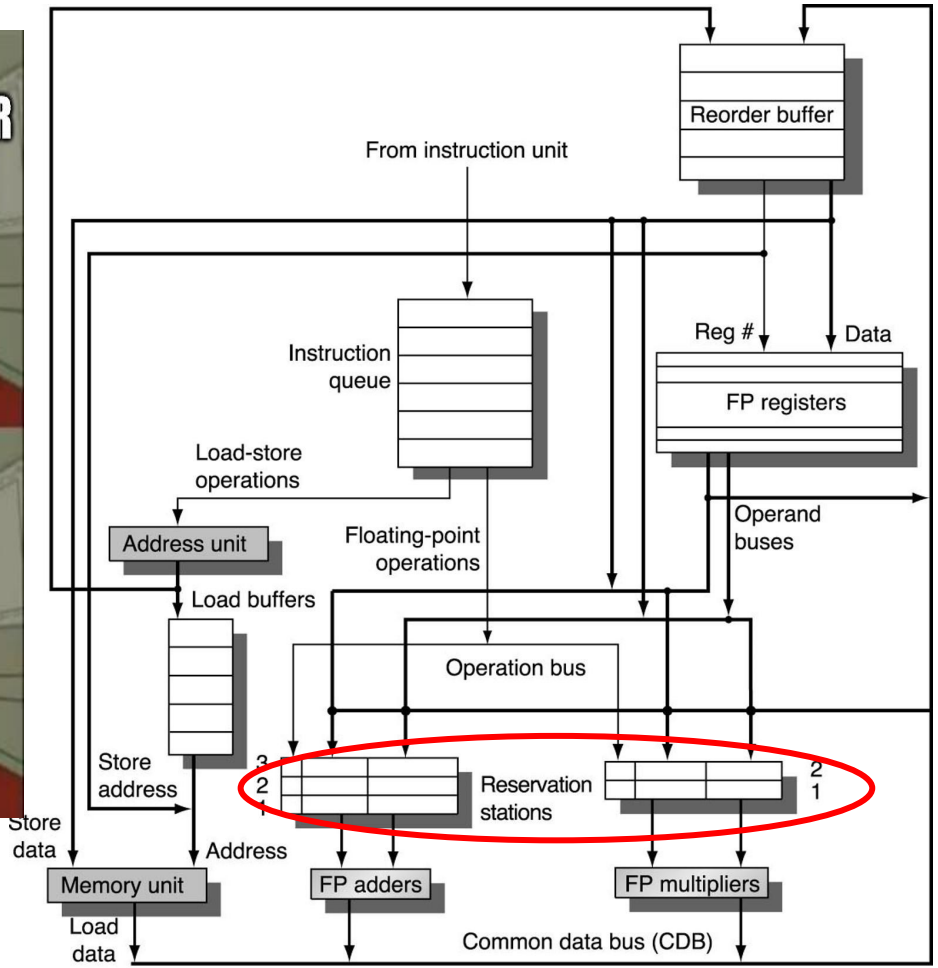


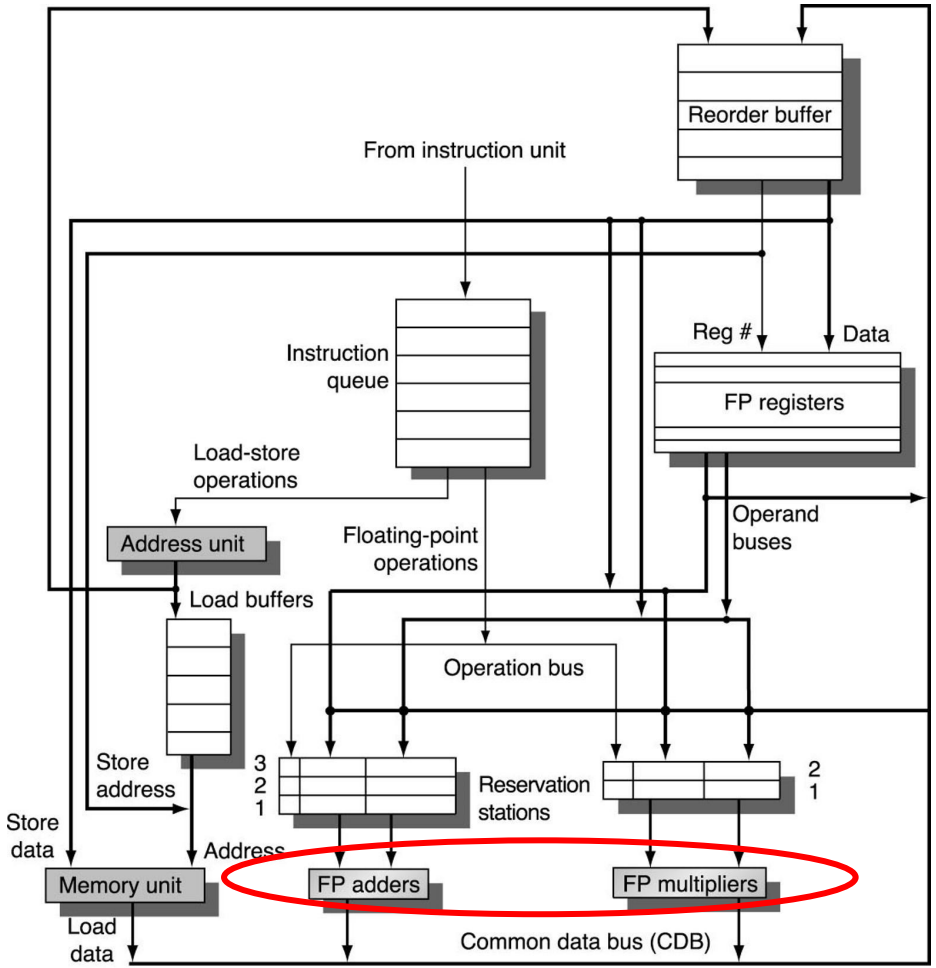

Reorder buffer
From instruction unit
Instruction queue
Reg #
Data
FP registers
Load-store operations
Address unit
Floating-point operations
Operand buses
Load buffers
Operation bus
Store address
3
2
1
Reservation stations
2
1
Store data
Address
Memory unit
FP adders
FP multipliers
Load data
Common data bus (CDB)

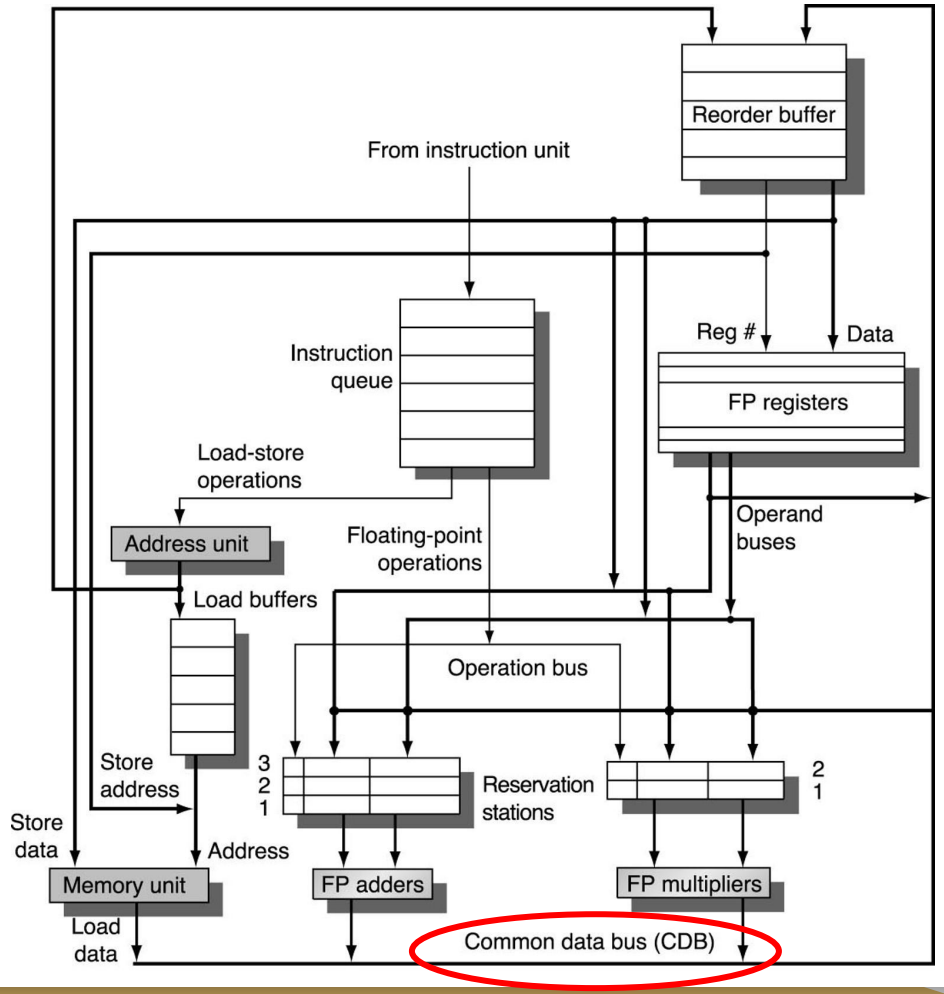

Reorder buffer
From instruction unit
Instruction queue
Reg #
Data
FP registers
Load-store operations
Address unit
Floating-point operations
Operand buses
Load buffers
Operation bus
Store address
3
2
1
Reservation stations
2
1
Store data
Address
Memory unit
FP adders
FP multipliers
Load data
Common data bus (CDB)

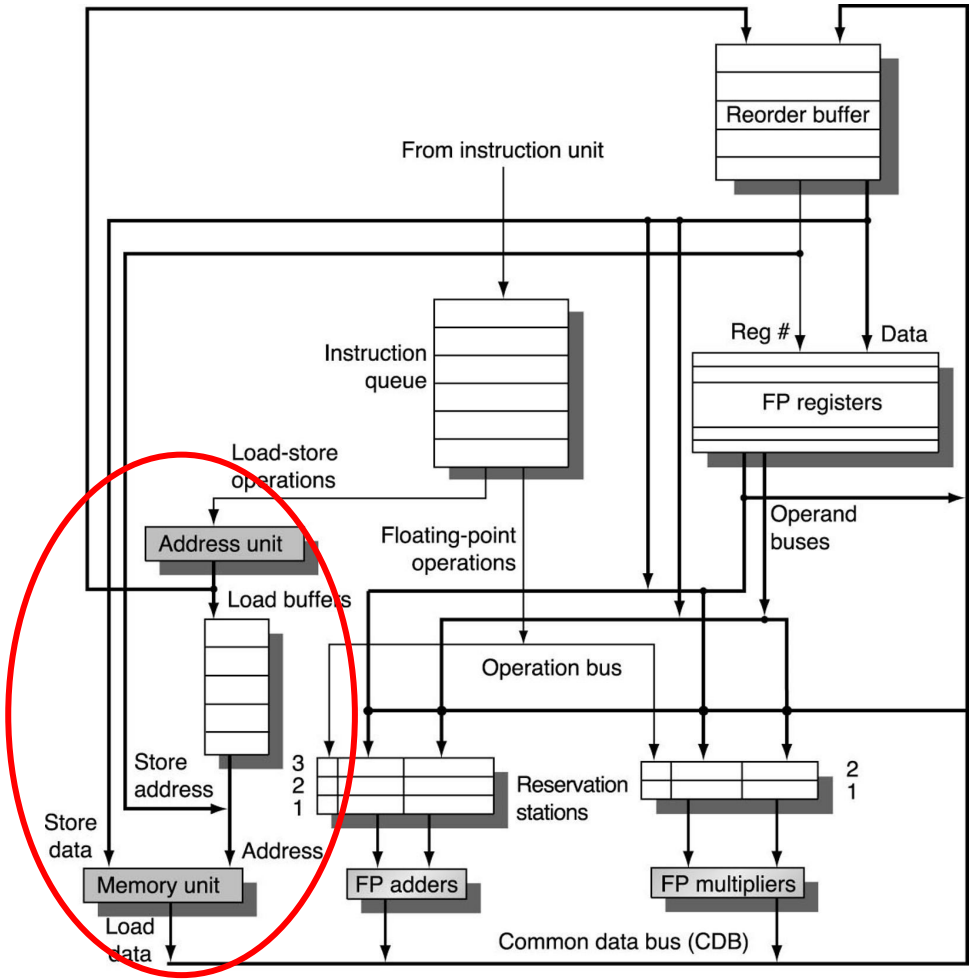

Reorder buffer
From instruction unit
Instruction queue
Reg #
Data
FP registers
Load-store operations
Address unit
Floating-point operations
Operand buses
Load buffers
Operation bus
Store address
3
2
1
Reservation stations
2
1
Store data
Address
Memory unit
FP adders
FP multipliers
Load data
Common data bus (CDB)


THIS IS WHERE I'D PUT MY SOURCE REGISTER
RESERVATION STATION
IF IT WAS READY
imgflip.com

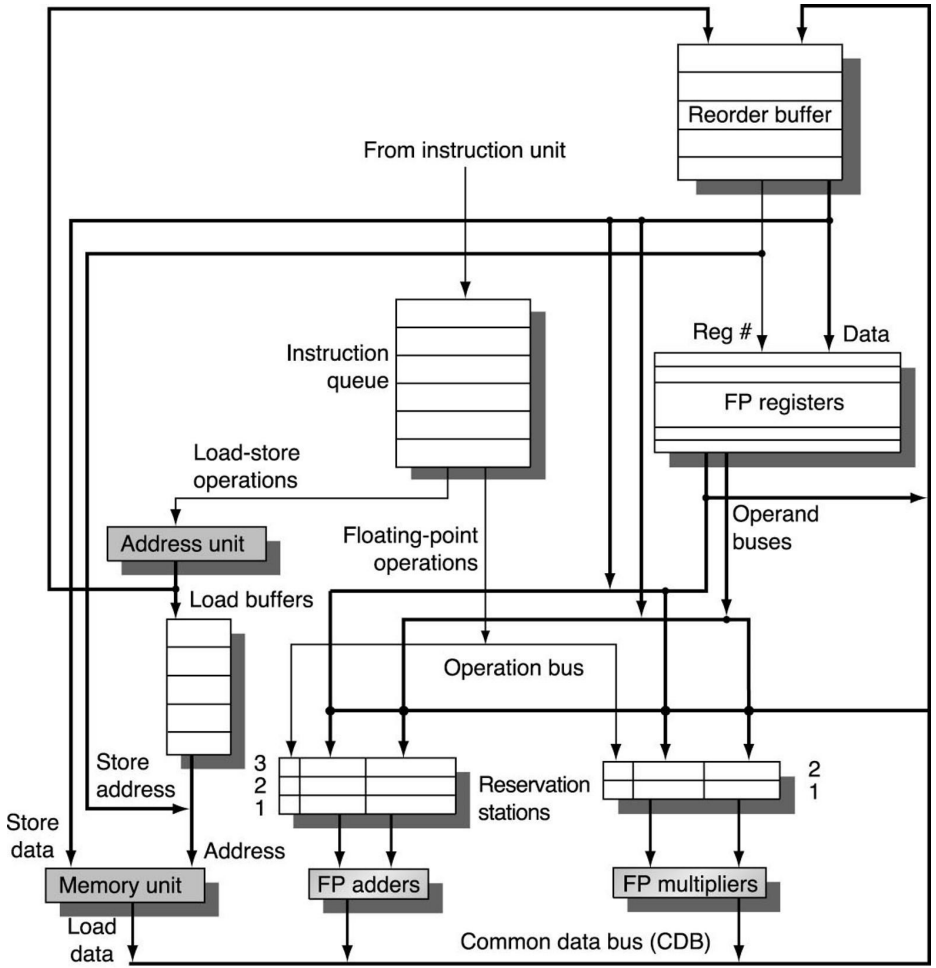

Reorder buffer
From instruction unit
Instruction queue
Reg #
Data
FP registers
Load-store operations
Address unit
Floating-point operations
Operand buses
Load buffers
Operation bus
Store address
Reservation stations
Store data
Address
Memory unit
FP adders
FP multipliers
Load data
Common data bus (CDB)


Reorder buffer
From instruction unit
Instruction queue
Reg #
Data
FP registers
Load-store operations
Address unit
Floating-point operations
Operand buses
Load buffers
Operation bus
Store address
3
2
1
Reservation stations
2
1
Store data
Address
Memory unit
FP adders
FP multipliers
Load data
Common data bus (CDB)


Reorder buffer
From instruction unit
Instruction queue
Reg #
Data
FP registers
Load-store operations
Address unit
Floating-point operations
Operand buses
Load buffers
Operation bus
Store address
3
2
1
Reservation stations
2
1
Store data
Address
Memory unit
FP adders
FP multipliers
Load data
Common data bus (CDB)


Reorder buffer
From instruction unit
Instruction queue
Reg #
Data
FP registers
Load-store operations
Address unit
Floating-point operations
Operand buses
Load buffers
Operation bus
Store address
3
2
1
Reservation stations
2
1
Store data
Address
Memory unit
FP adders
FP multipliers
Load data
Common data bus (CDB)

What's missing?



What's missing?

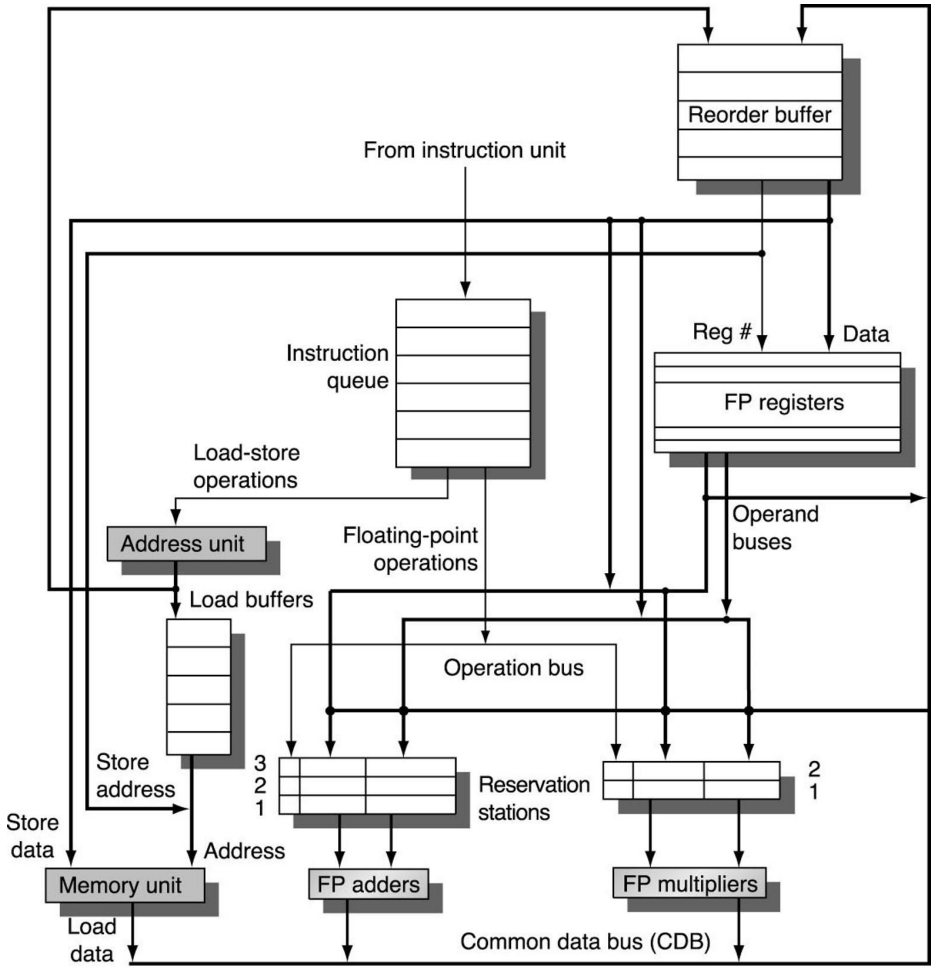
-I cache

-D cache

-Branch prediction and recovery

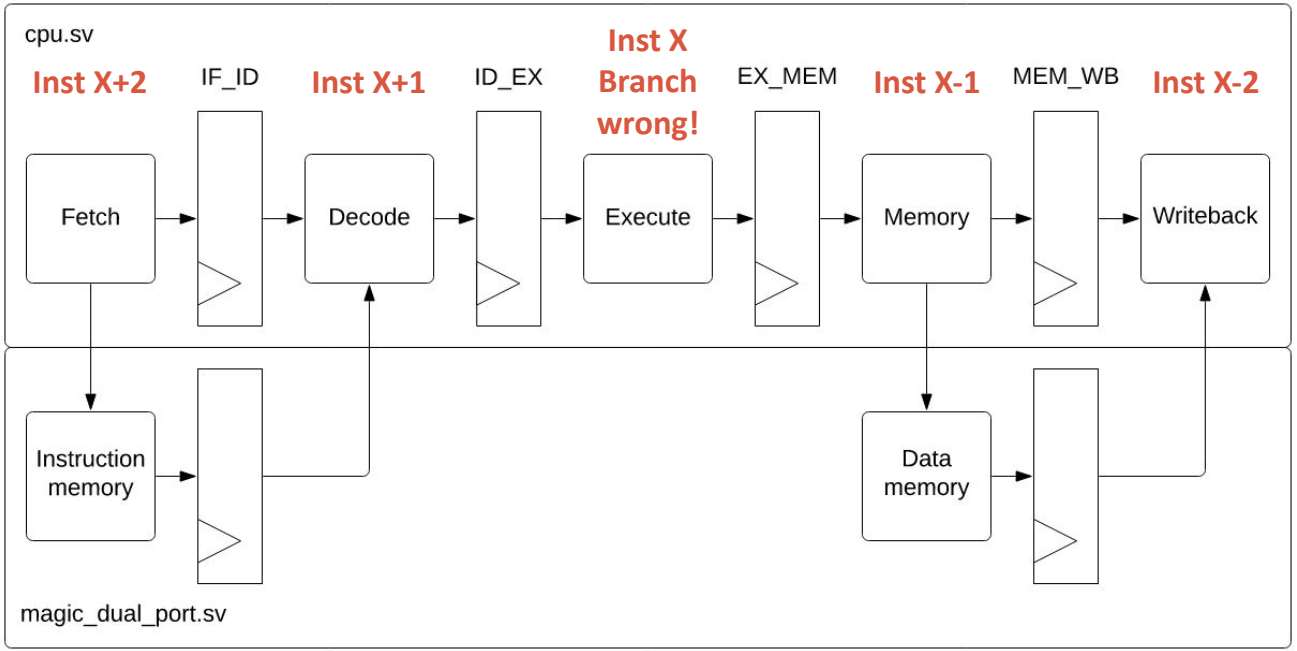
- *(& you are using integer not floating point so reg file and execution units will be different)*

- Load/Store logic is flexible and you probably will want something different.

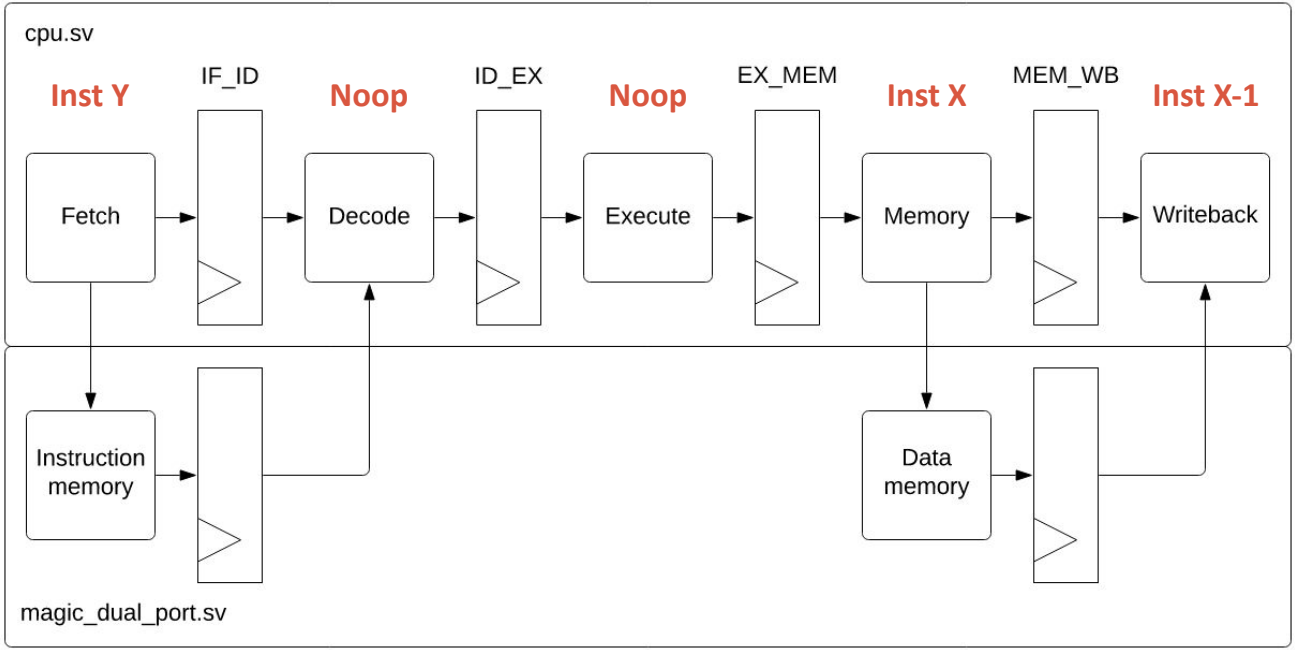


Branch Misprediction (Speculative Execution)

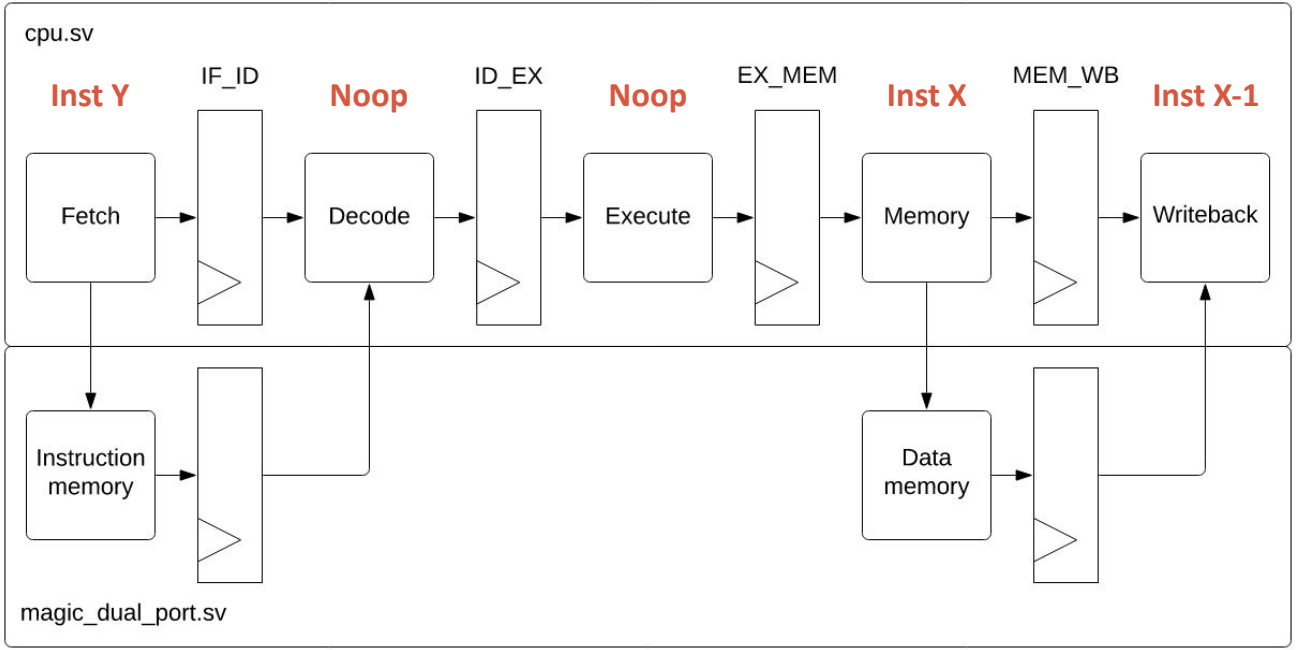
Speculative Execution: Pipelined Design



Speculative Execution: Pipelined Design



Speculative Execution: Pipelined Design



Q: But how do we do this in an out of order processor?
Hint: Which structure maintains program order?

Speculative Execution: Pipelined Design

Q: But how do we do this in an out of order processor?

V1: On commit, check if the branch was mispredicted. If it was, squash everything younger than the branch (every other instruction).
[Minimum implementation]

Requires flush signals sent to all of your hardware blocks. These are triggered by the ROB commit logic. This is a little trickier in Explicit Register Renaming (see later slide)

Speculative Execution: Pipelined Design

Q: But how do we do this in an out of order processor?

V2: Squash branches immediately once you figure out they were mispredicted.
[Advanced Feature: Early Branch Recovery]

Requires branch dependency logic to be tagged on every instruction
Also requires checkpointing processor state before every branch so that it can be recovered on a mispredict.

For more details, speak to your mentor TA.

Memory

Memory Operations: Correctness

Rules for memory correctness:

- The contents of memory (including cache) must never be speculative.
- The contents of memory (including cache) must be updated in order
- Loads to address X must reflect the result of the youngest store that modifies address X that is older than the load.

Memory Operations: Challenges

Handling memory instructions is non-trivial in an out-of-order processor

Q: You tell me - why are memory instructions challenging?

Memory Operations: Challenges

Handling memory instructions is non-trivial in an out-of-order processor

Q: You tell me - why are memory instructions challenging?

A: Memory disambiguation. Memory operations have register dependencies *and* memory address dependencies! Furthermore, you can't rollback a write to memory once it occurs!

Memory Operations: Example

Program:

```
SW: x10, 12(x15)  # Store x10 ->[12 + x15]
ADD: x5, x0, x2
LW: x1, 12(x15) # Load [12 + x15] -> x1
```

Memory Operations: Example

Program:

SW: x10, 12(x15) # Store x10 ->[12 + x15]

ADD: x5, x0, x2

LW: x1, 12(x15) # Load [12 + x15] -> x1

Memory Operations: Example

Program:

```
SW: x10, 12(x15)  # Store x10 ->[12 + x15]
ADD: x5, x0, x2
LW: x1, 12(x14) # Load [12 + x14] -> x1
```

Memory Operations: Example

Program:

SW: x10, 12(x15) # Store x10 ->[12 + x15]

ADD: x5, x0, x2

LW: x1, 12(x14) # Load [12 + x14] -> x1

?

Memory Operations: Example

Program:

SW: x10, 12(x15) # Store x10 ->[12 + x15]

ADD: x5, x0, x2

LW: x1, 12(x14) # Load [12 + x14] -> x1

Need memory disambiguation!

Loads/Stores Require Two Steps:

1. Determine source/destination address
2. Read/Write memory

Memory Operations: Details

When Can a Store Execute (Write to memory?)

When Can a Load Execute (Read from memory)?

Memory Operations: Details

When Can a Store Execute (Write to memory?)
A: On Commit

When Can a Load Execute (Read from memory)?
V1: Once all older stores are committed
[Minimum implementation]

Easiest implementation is to stall dispatch

- After a store is dispatched, stall dispatching all subsequent memory instructions until that store commits.

Memory Operations: Details

When Can a Store Execute (Write to memory?)
A: On Commit

When Can a Load Execute (Read from memory)?
V2: Once all older store addresses are known and conflicting stores are committed
[Advanced Feature]

Need to keep track of relative ordering of loads & stores. Requires complex load-store queue(s)

Memory Operations: Details

When Can a Store Execute (Write to memory?)
A: On Commit

When Can a Load Execute (Read from memory)?
V3: Once all older store addresses are known, with store values forwarded to the load before the store commits.
[Bigger advanced feature]

Requires loads to grab values from store queue in addition to load/store queue(s).

Memory Operations: Details

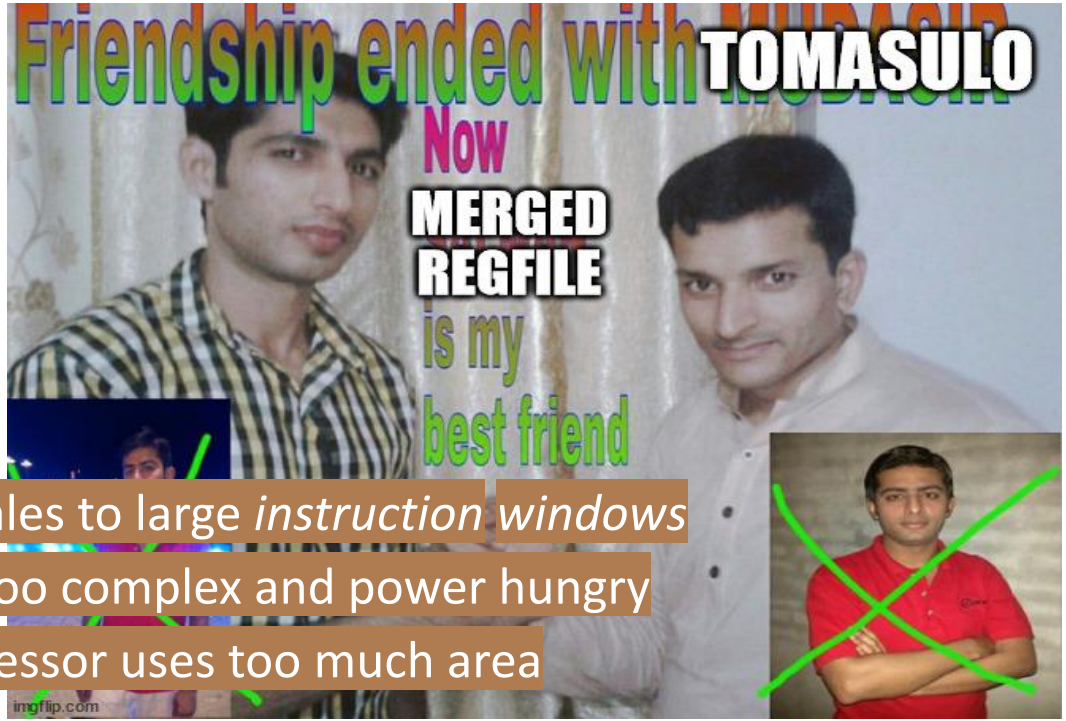
When Can a Store Execute (Write to memory?)
A: On Commit

When Can a Load Execute (Read from memory)?
V4: Speculatively decide (before the addresses are known) whether store values should be forwarded to loads and *rollback* if wrong
[Hardest implementation & worth the most advanced feature points.]

See Fire & Forget paper or create a custom load-store queue implementation with data dependency predictors and squashing logic.

Explicit Register Renaming

Explicit Register Renaming (ERR)



Motivations:

- Tomasulo poorly scales to large *instruction windows*
  - ROB becomes too complex and power hungry
  - The entire processor uses too much area

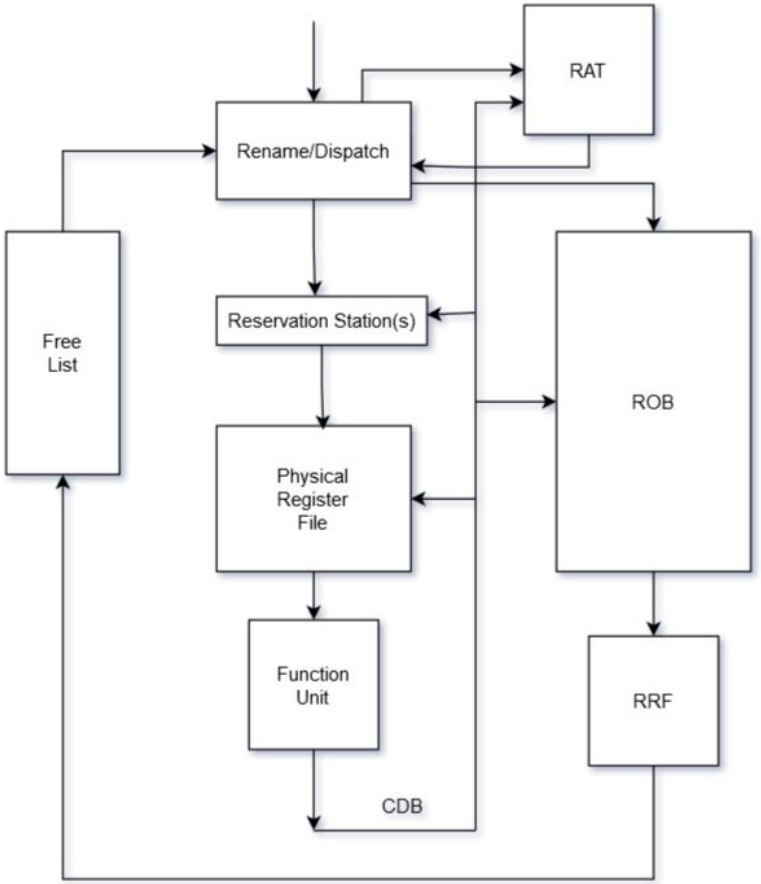
Can we be more efficient?

Datapath Diagram

Only the Physical Register File stores data

It is only read *after* an instruction leaves a reservation station

If you decide to do ERR, you will need to include much more detail!



Free List

It’s just a FIFO*!

*with a bit of fanciness for control instructions

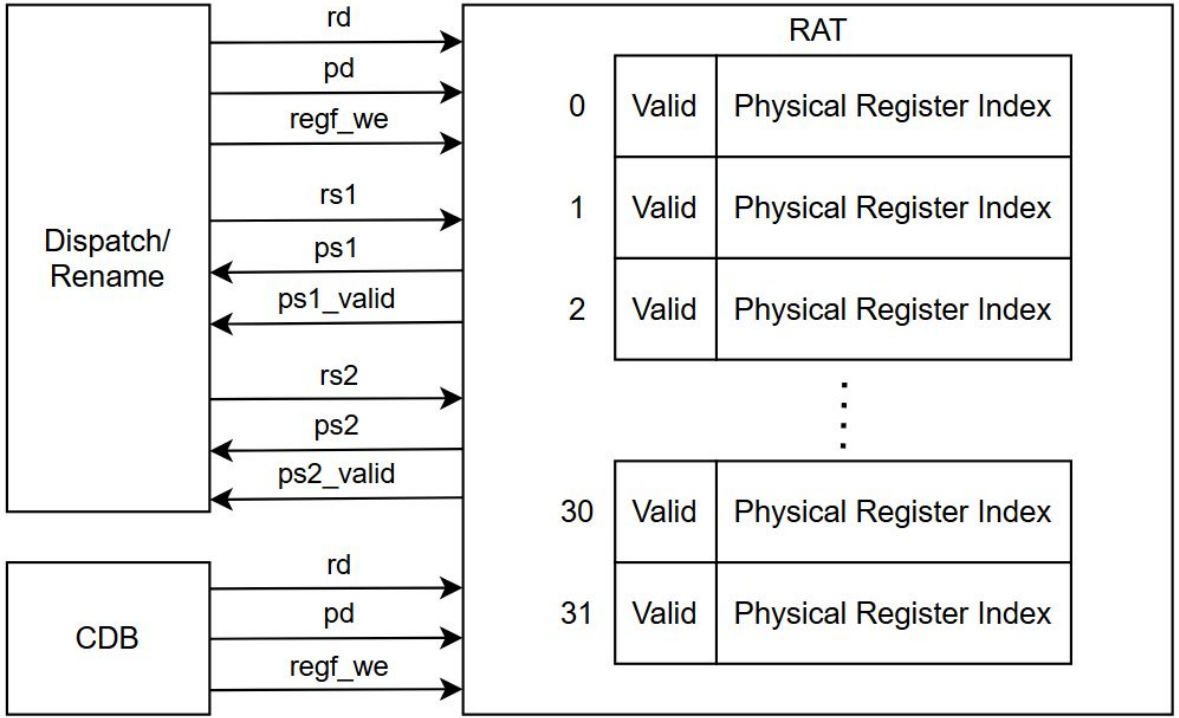
RAT - Register Alias Table

Dispatch/Rename:

- Maps arch sources (rs1/rs2) to phys sources (ps1/ps2)
- Renames rd to pd, marking invalid

CDB:

- Sets entry rd to valid if it still maps to pd.

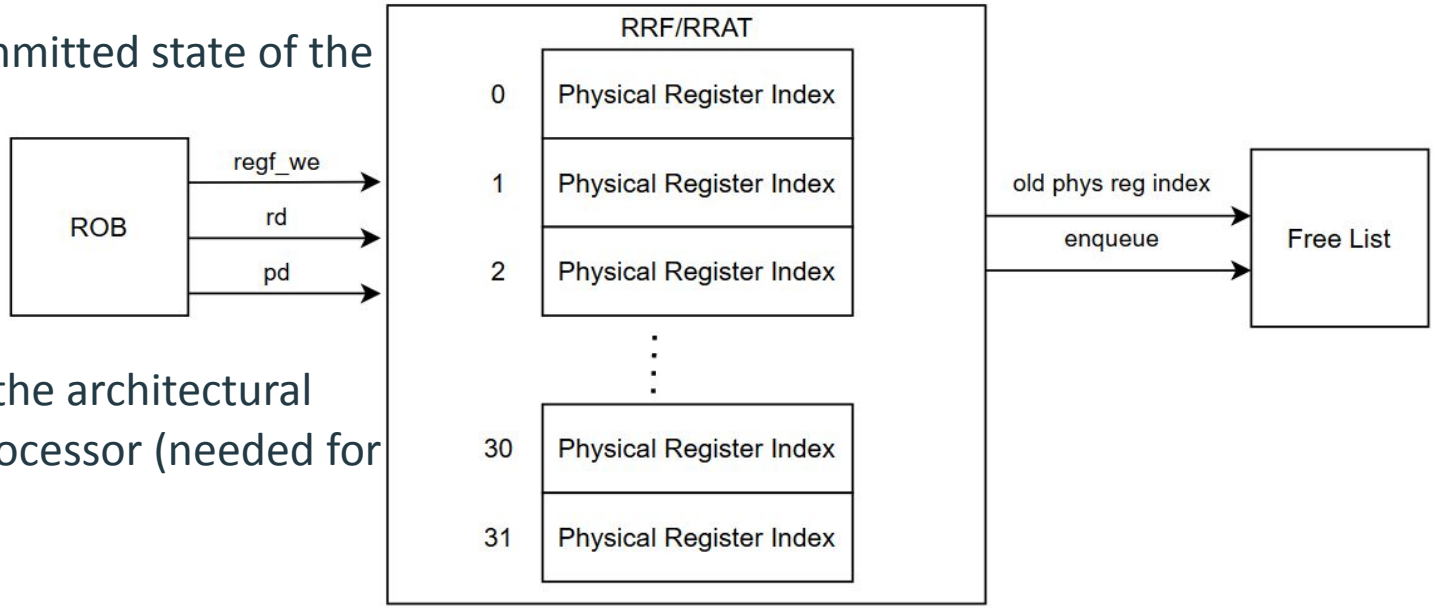


RRF/RRAT (Retirement RF/RAT)

Similar to RAT - just no valid field

Stores the committed state of the processor

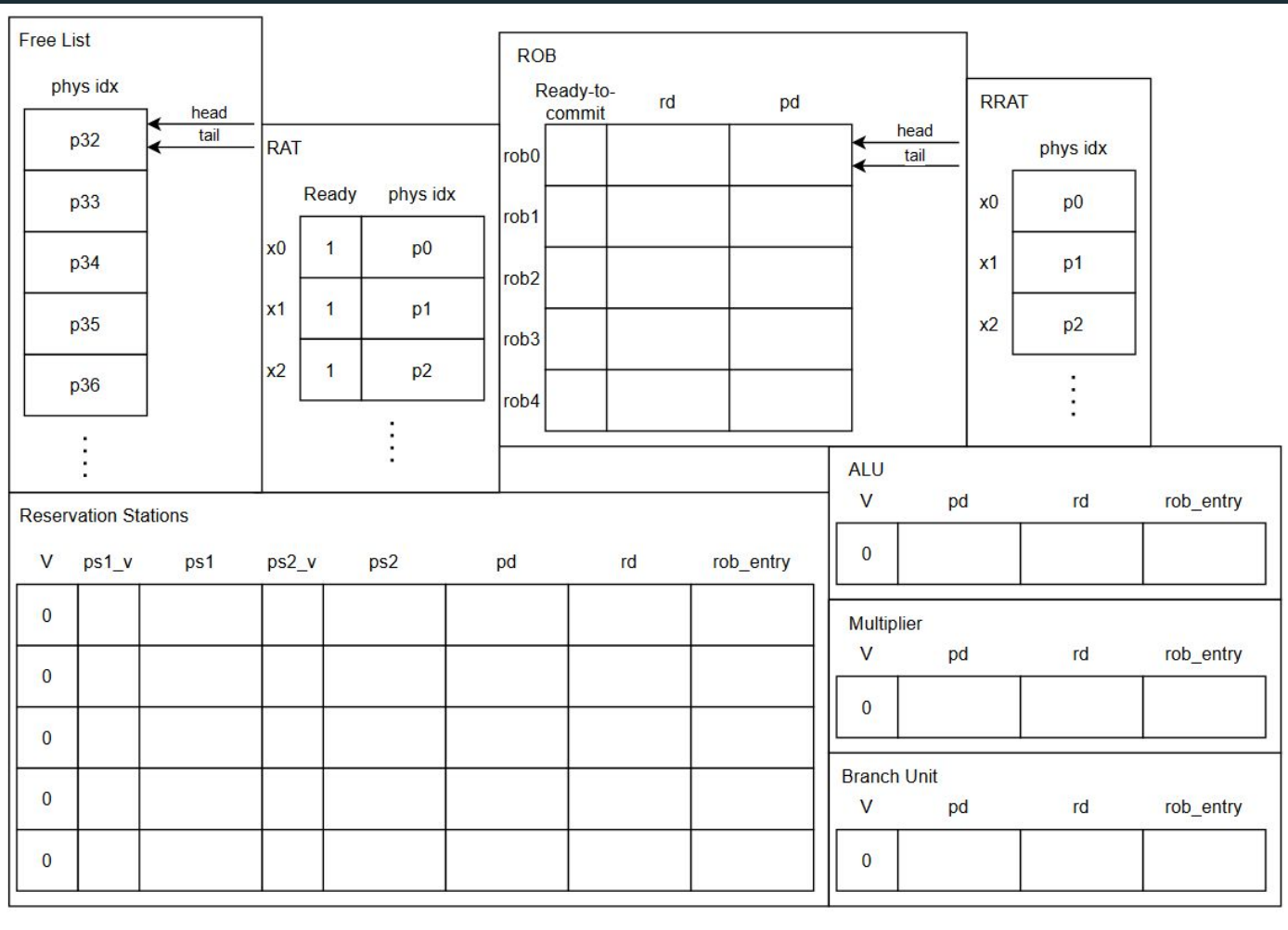
Used to store the architectural state of the processor (needed for speculation)



Cycle 0:

Program:
Addi x1, x0, 4
Add x2, x1, x0
Mul x2, x2, x1
Bge x2, x1, label
Add x2, x2, x0
Add x1, x1, x1

Mul: 3 cycles
ALU: 1 cycles
BR: 1 cycles



Cycle 1:

Program:

Addi x1, x0, 4

Add x2, x1, x0

Mul x2, x2, x1

Bge x2, x1, label

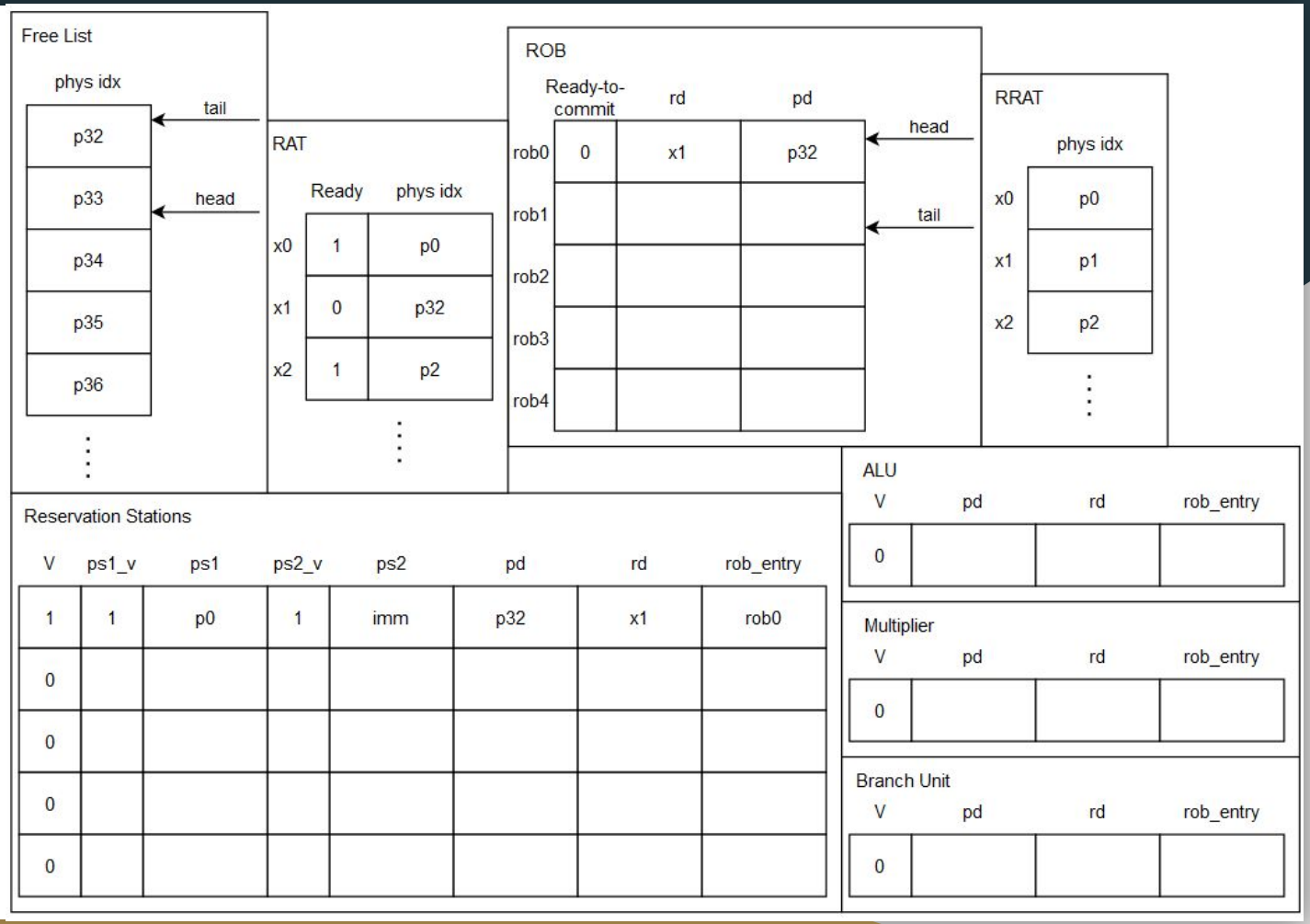
Add x2, x2, x0

Add x1, x1, x1

Mul: 3 cycles

ALU: 1 cycles

BR: 1 cycles



Cycle 2:

Program:

Addi x1, x0, 4

Add x2, x1, x0

Mul x2, x2, x1

Bge x2, x1, label

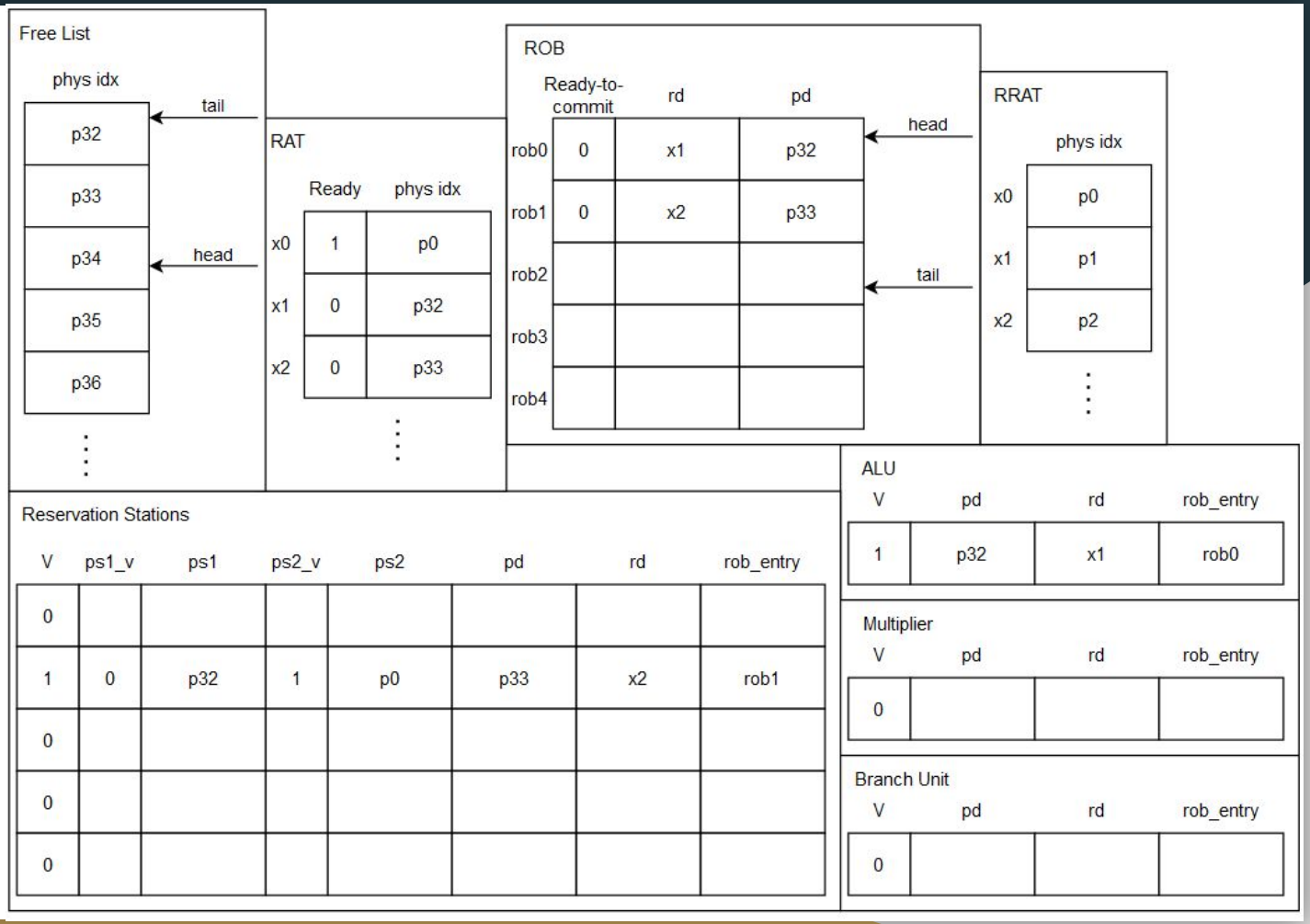
Add x2, x2, x0

Add x1, x1, x1

Mul: 3 cycles

ALU: 1 cycles

BR: 1 cycles



Cycle 3:

Program:

Addi x1, x0, 4

Add x2, x1, x0

Mul x2, x2, x1

Bge x2, x1, label

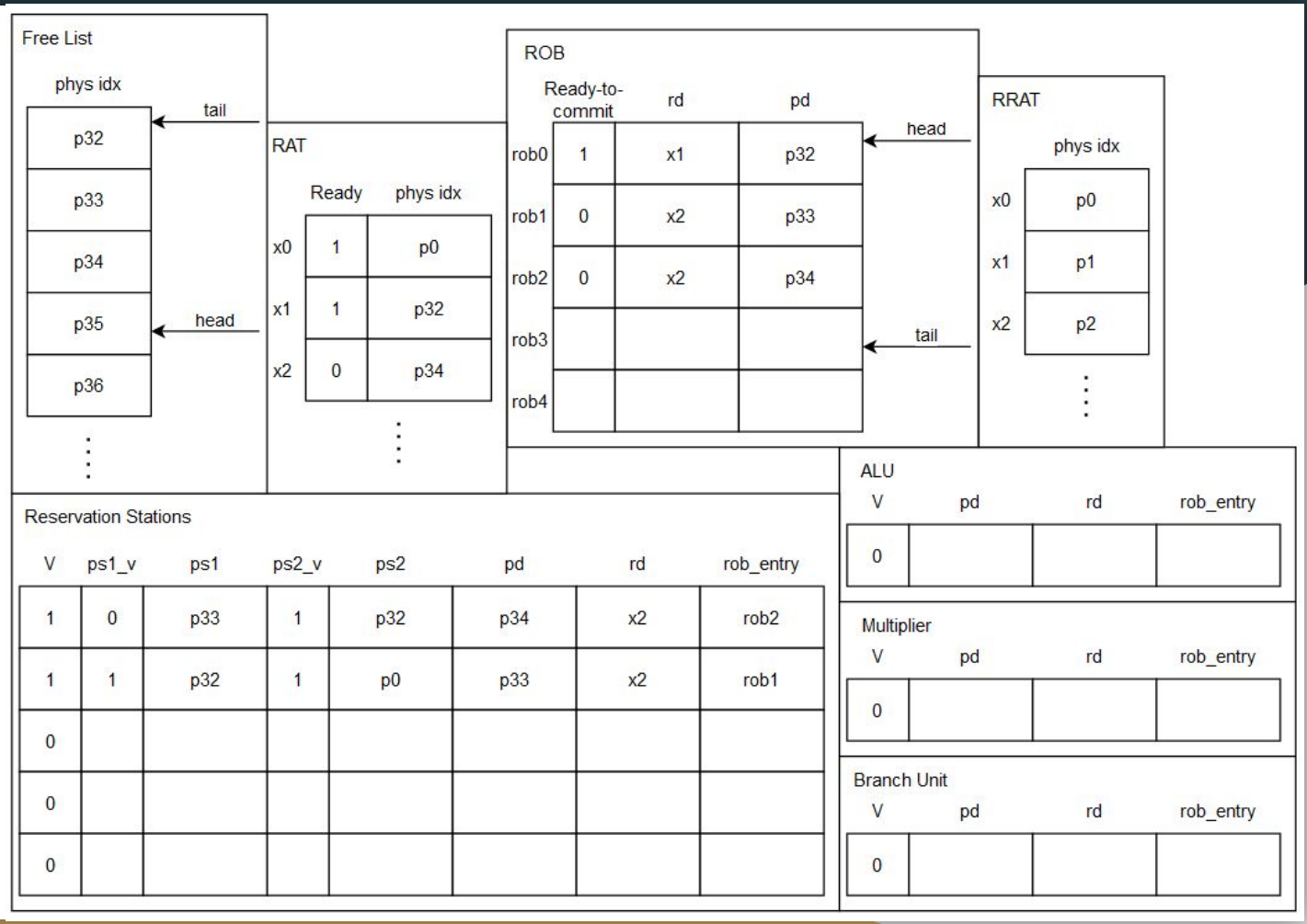
Add x2, x2, x0

Add x1, x1, x1

Mul: 3 cycles

ALU: 1 cycles

BR: 1 cycles



Cycle 4:

Program:

Addi x1, x0, 4

Add x2, x1, x0

Mul x2, x2, x1

Bge x2, x1, label

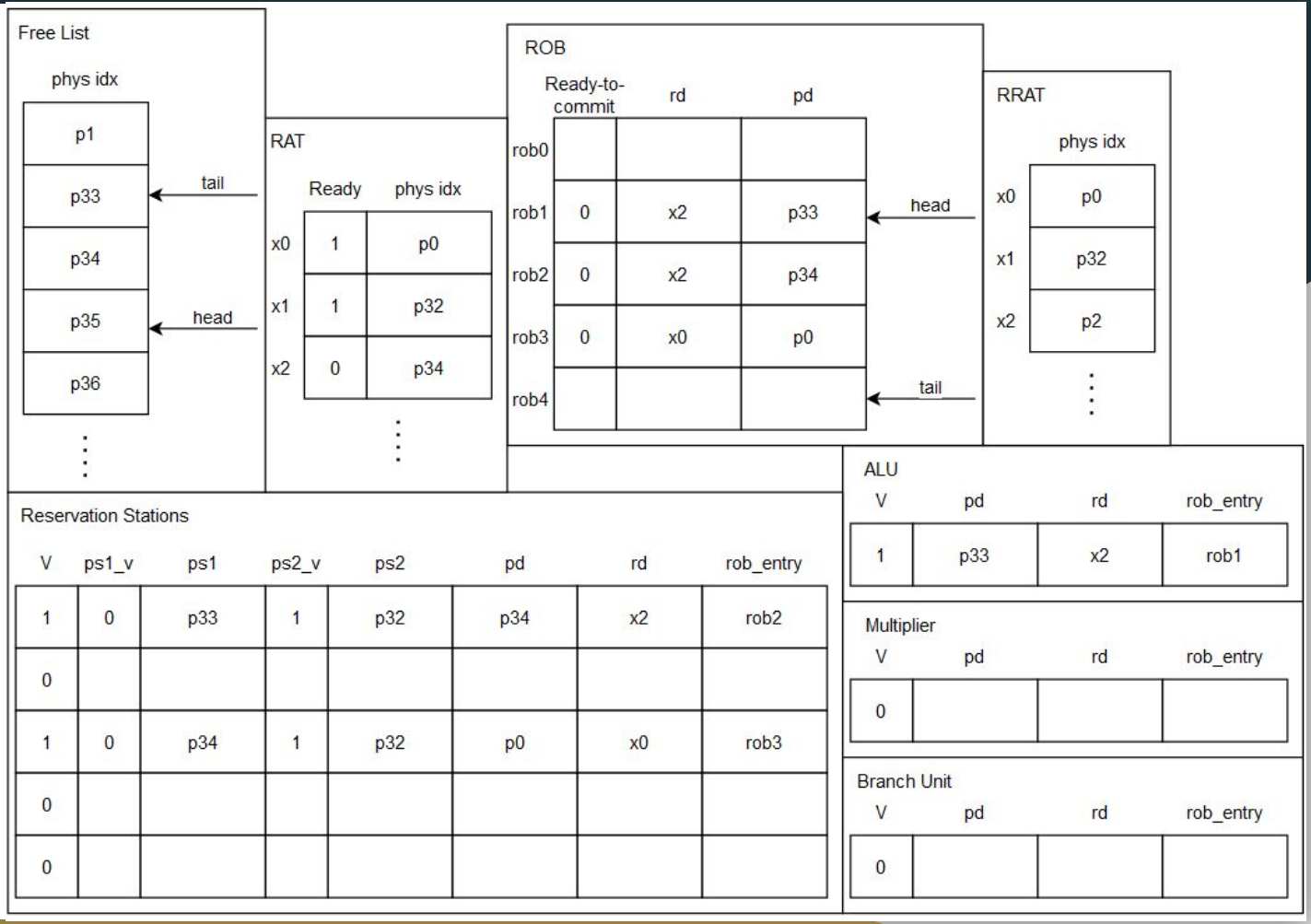
Add x2, x2, x0

Add x1, x1, x1

Mul: 3 cycles

ALU: 1 cycles

BR: 1 cycles



Cycle 5:

Program:

Addi x1, x0, 4

Add x2, x1, x0

Mul x2, x2, x1

Bge x2, x1, label

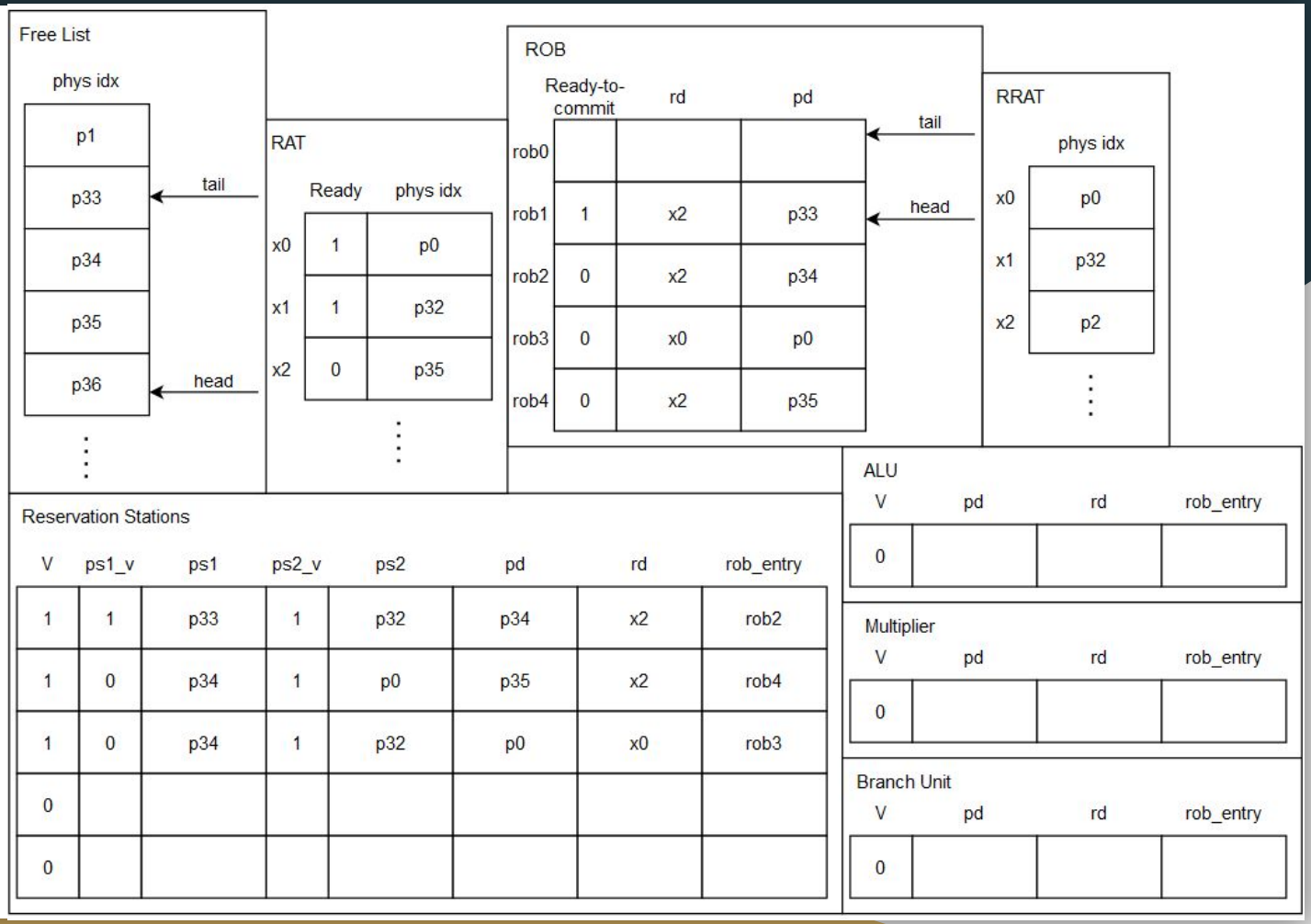
Add x2, x2, x0

Add x1, x1, x1

Mul: 3 cycles

ALU: 1 cycles

BR: 1 cycles

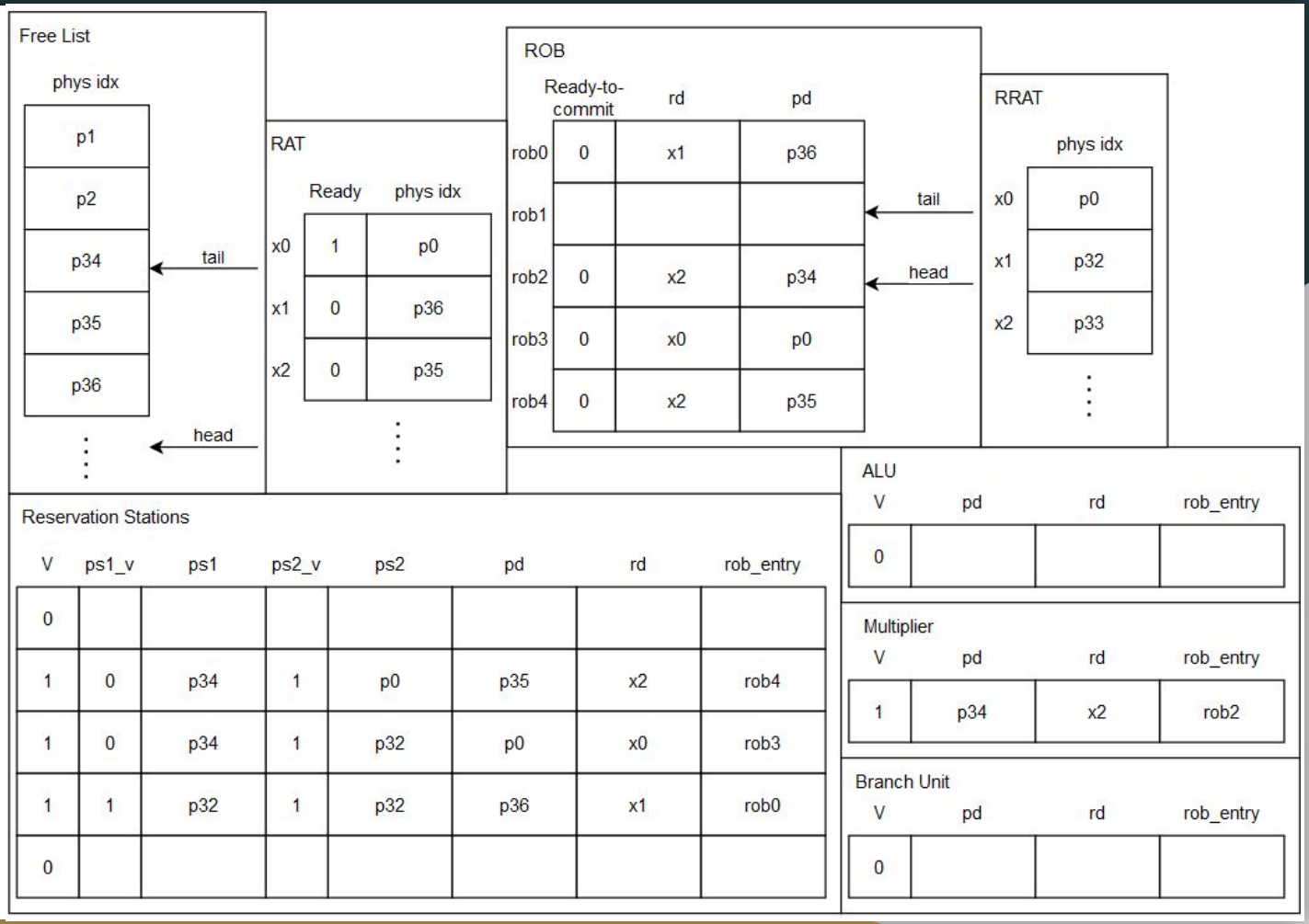


Cycle 6:

Program:

Addi x1, x0, 4
Add x2, x1, x0
Mul x2, x2, x1
Bge x2, x1, label
Add x2, x2, x0
Add x1, x1, x1

Mul: 3 cycles
ALU: 1 cycles
BR: 1 cycles

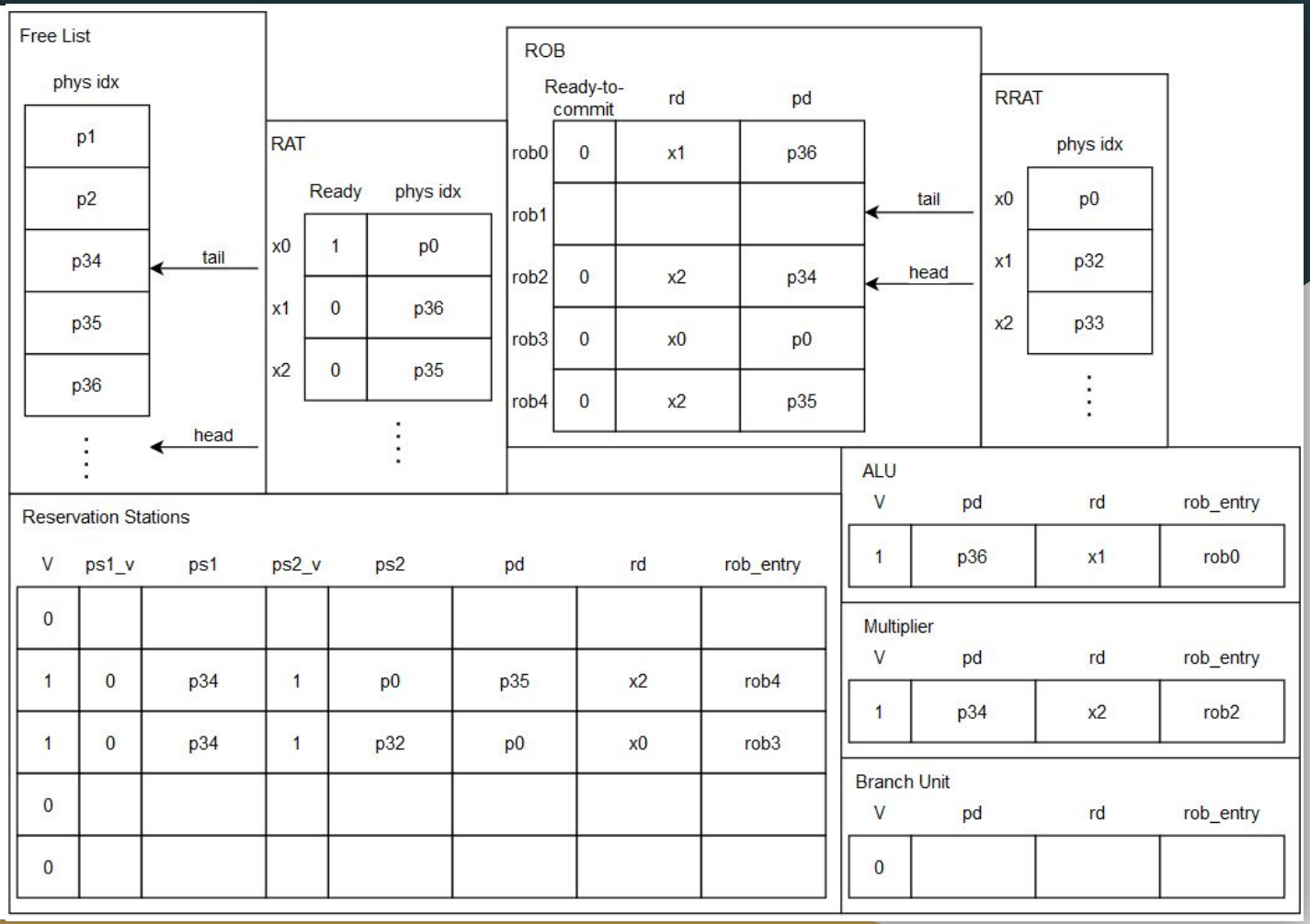


Cycle 7:

Program:

Addi x1, x0, 4
Add x2, x1, x0
Mul x2, x2, x1
Bge x2, x1, label
Add x2, x2, x0
Add x1, x1, x1

Mul: 3 cycles
ALU: 1 cycles
BR: 1 cycles

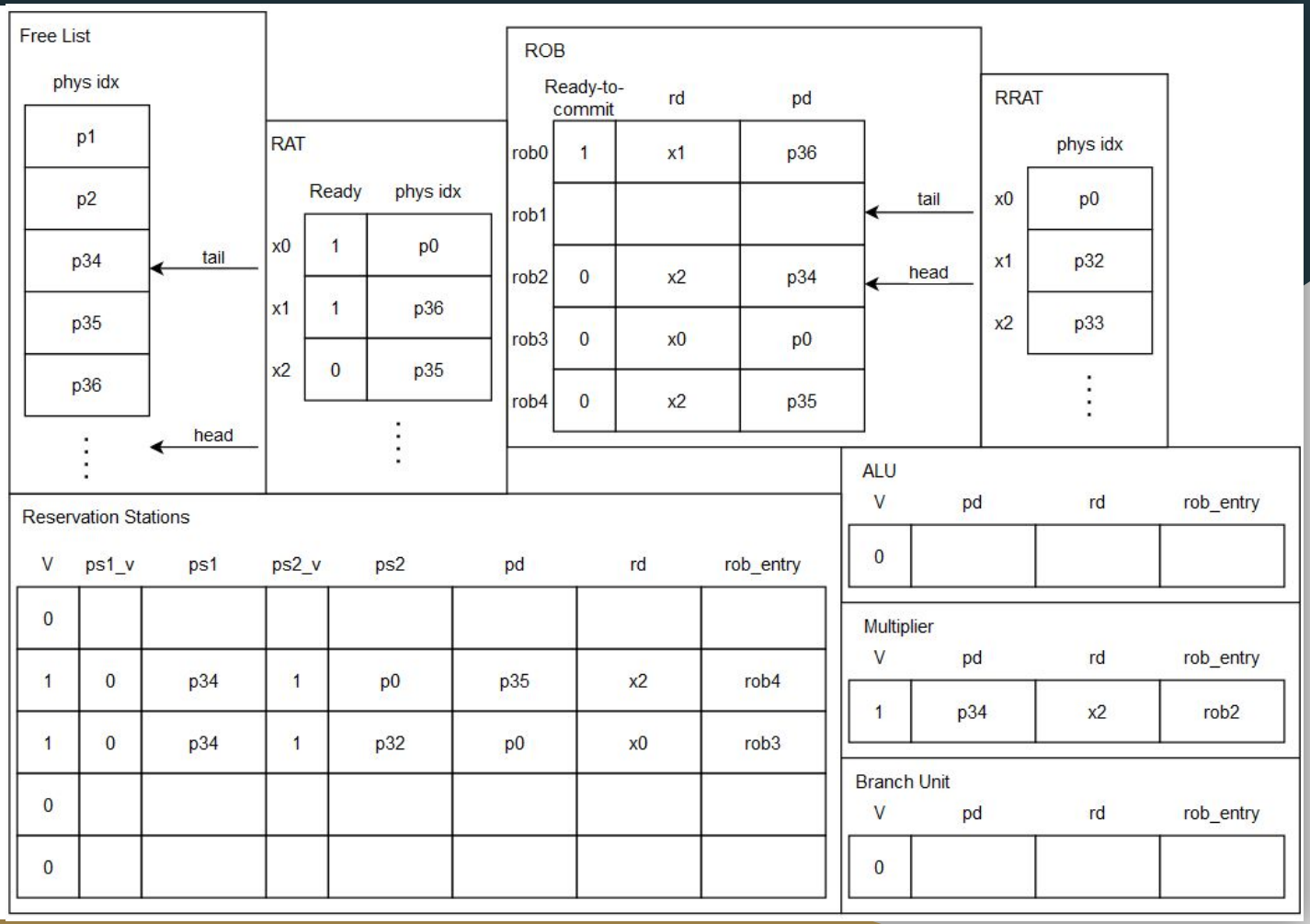


Cycle 8:

Program:

Addi x1, x0, 4
Add x2, x1, x0
Mul x2, x2, x1
Bge x2, x1, label
Add x2, x2, x0
Add x1, x1, x1

Mul: 3 cycles
ALU: 1 cycles
BR: 1 cycles

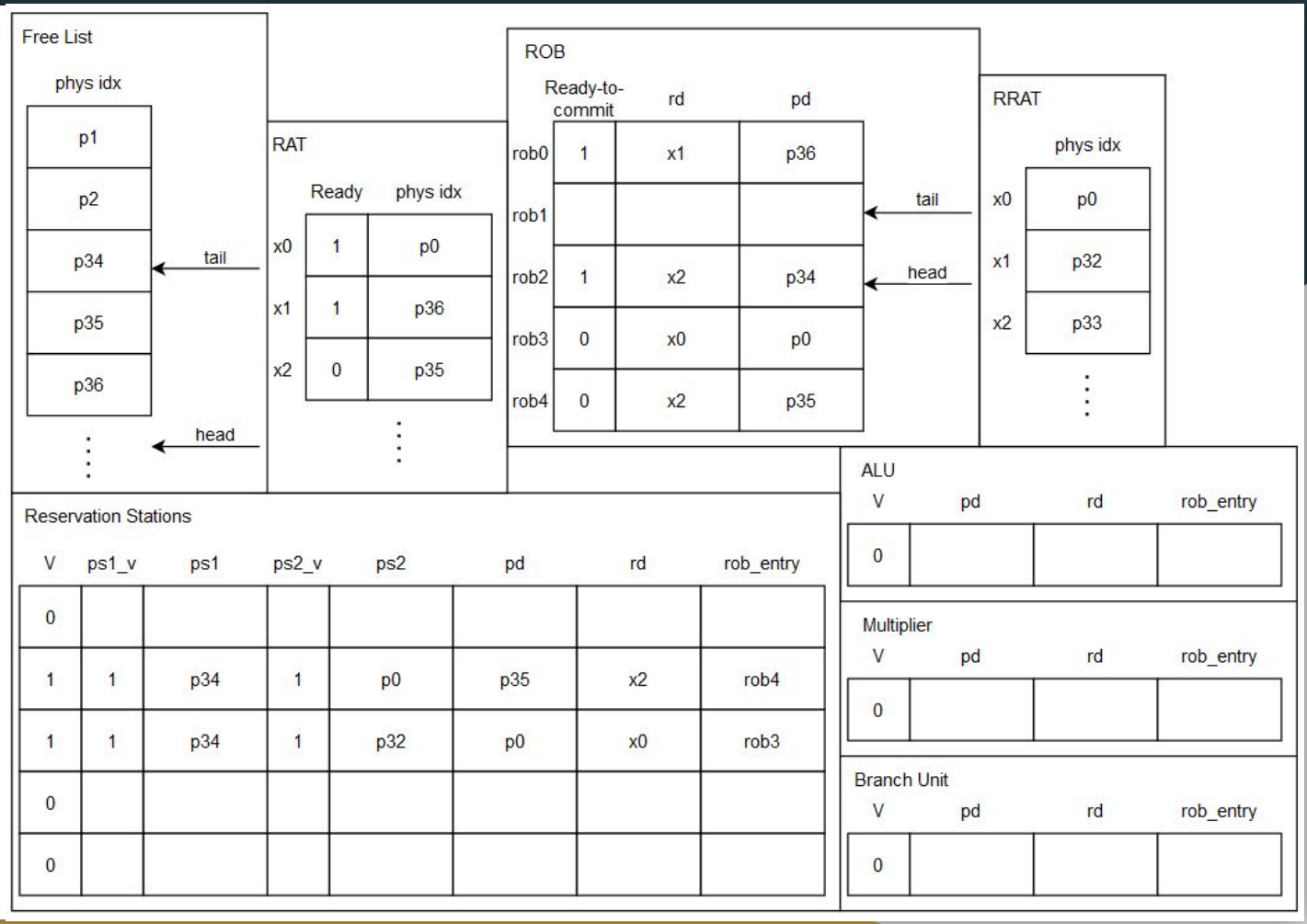


Cycle 9:

Program:

Addi x1, x0, 4
Add x2, x1, x0
Mul x2, x2, x1
Bge x2, x1, label
Add x2, x2, x0
Add x1, x1, x1

Mul: 3 cycles
ALU: 1 cycles
BR: 1 cycles



Cycle 10:

Program:

Addi x1, x0, 4

Add x2, x1, x0

Mul x2, x2, x1

Bge x2, x1, label

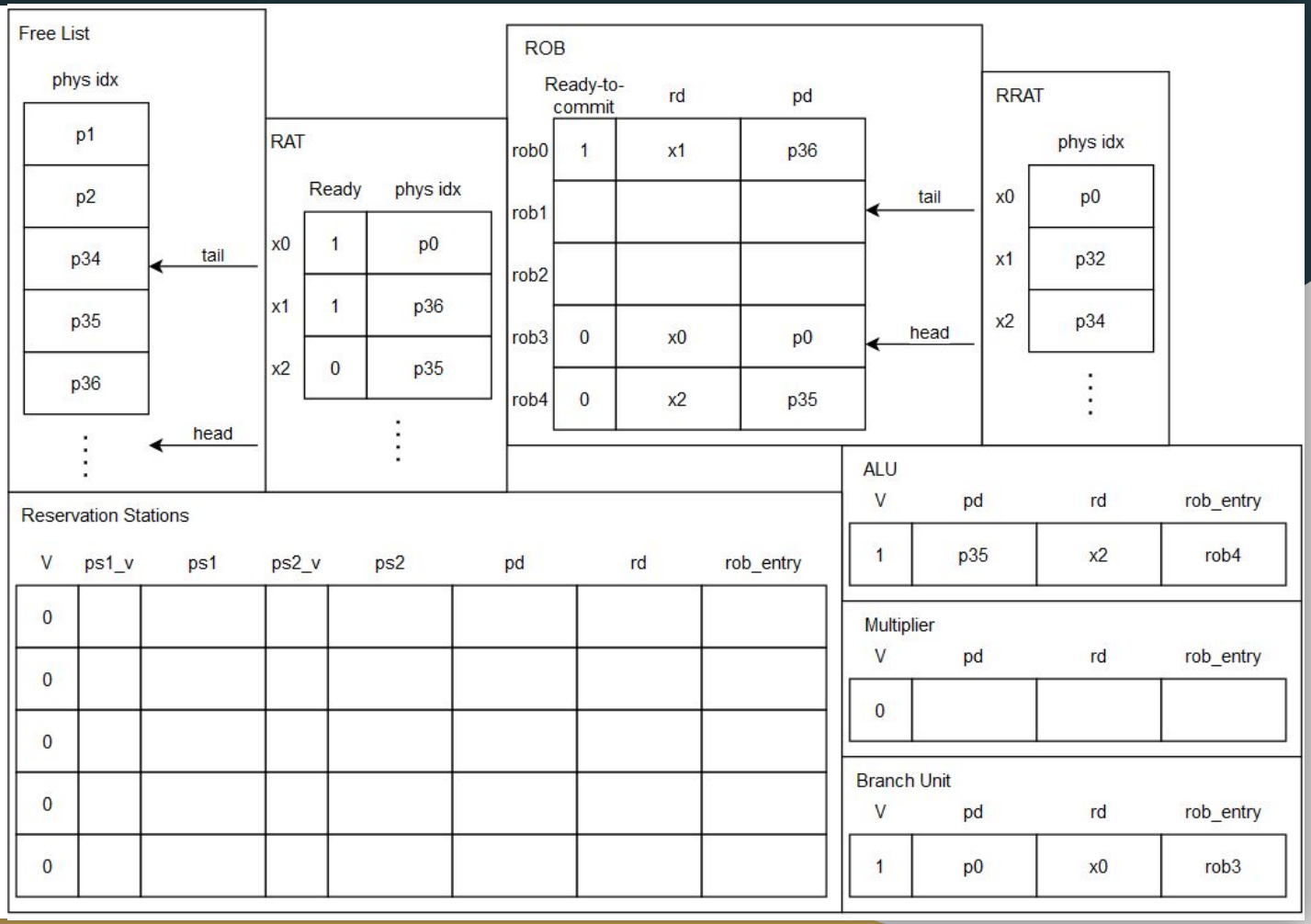
Add x2, x2, x0

Add x1, x1, x1

Mul: 3 cycles

ALU: 1 cycles

BR: 1 cycles

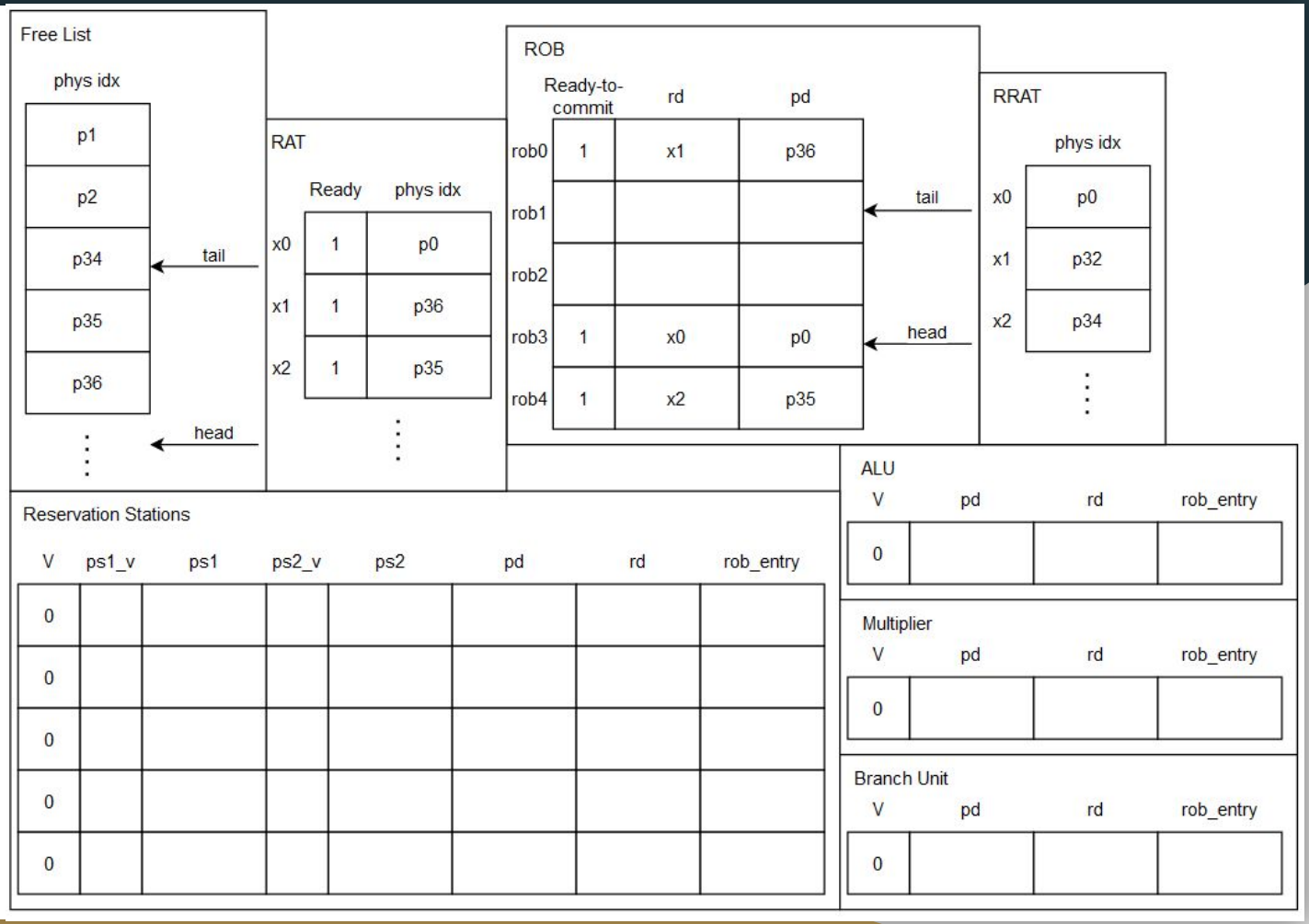


Cycle 11:

Program:

Addi x1, x0, 4
Add x2, x1, x0
Mul x2, x2, x1
Bge x2, x1, label
Add x2, x2, x0
Add x1, x1, x1

Mul: 3 cycles
ALU: 1 cycles
BR: 1 cycles

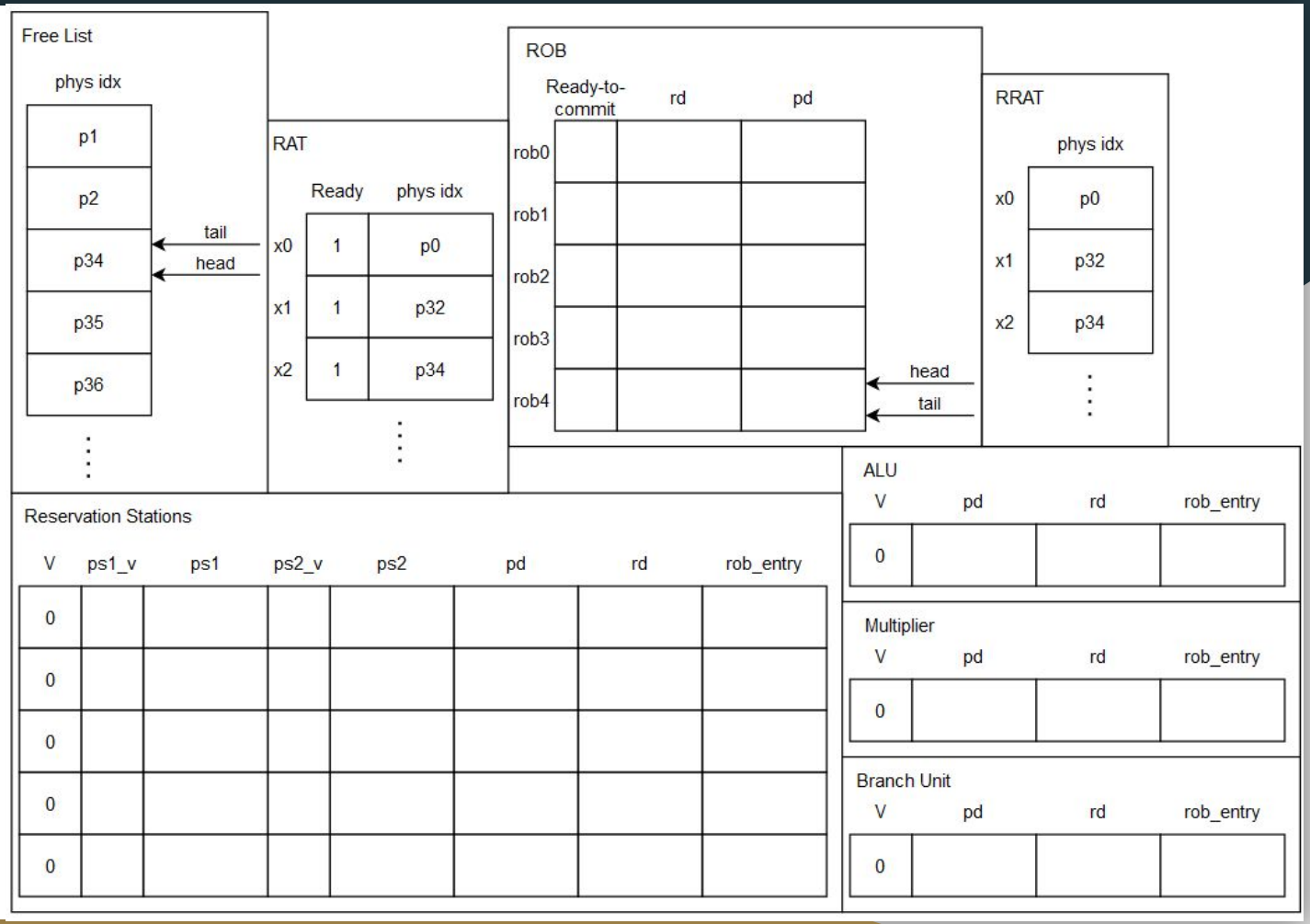


Cycle 12:

Program:

Addi x1, x0, 4
Add x2, x1, x0
Mul x2, x2, x1
Bge x2, x1, label
Add x2, x2, x0
Add x1, x1, x1

Mul: 3 cycles
ALU: 1 cycles
BR: 1 cycles



Memory Instructions in ERR

You cannot mix memory instructions with normal instructions inside the reservation stations

You need a separate structure (load-store queue, etc) for memory instructions.

GOOD LUCK!

You’re smart, you’ve got this, and we’re here to help.

Alternate Verification Tooling

What is Verilator?

- Like VCS, converts your hardware to C++. But it's open source!
  - Can run on your local machines as a result, and is cross-platform
- Drawbacks: there's a lot.
  - Cycle-level simulation
  - *Only supports dual-state simulation*
  - Also pickier about the quality of your HDL (stricter linting)
  - Incomplete SV support compared to VCS
  - Harder to support threaded models
  - Can't access internal signals by default
  - Traces are way bigger compared to VCS (15GB for all of Coremark with struct extraction)
- Why bother using it at all?
  - ***Speed* - "all of CoreMark within seconds" type speed.**

Verilator Testbenches

- Overall simulation logic is no longer managed by an SV initial block
- Need to write C++ logic to interface with the DUT, manage CLK, etc.
    - Can still write performance monitoring in SV
- State management in C++ -> powerful software models in a familiar language
    - Lack of bitslices can sometimes make memory access challenging
- Many other features, so definitely look at the documentation on their website!