CS152 Discussion Section

3/8/19

Administrivia

- Midterm grading moving along on schedule
- Today's discussion will introduce Lab 3
  - Won't officially be assigned until after Lab 2!
  - Due April 8th (one month from today)
  - Relies on local simulation like Lab 1
  - Instead of talking about the actual lab procedure, we'll take a higher-level look at what is actually being simulated

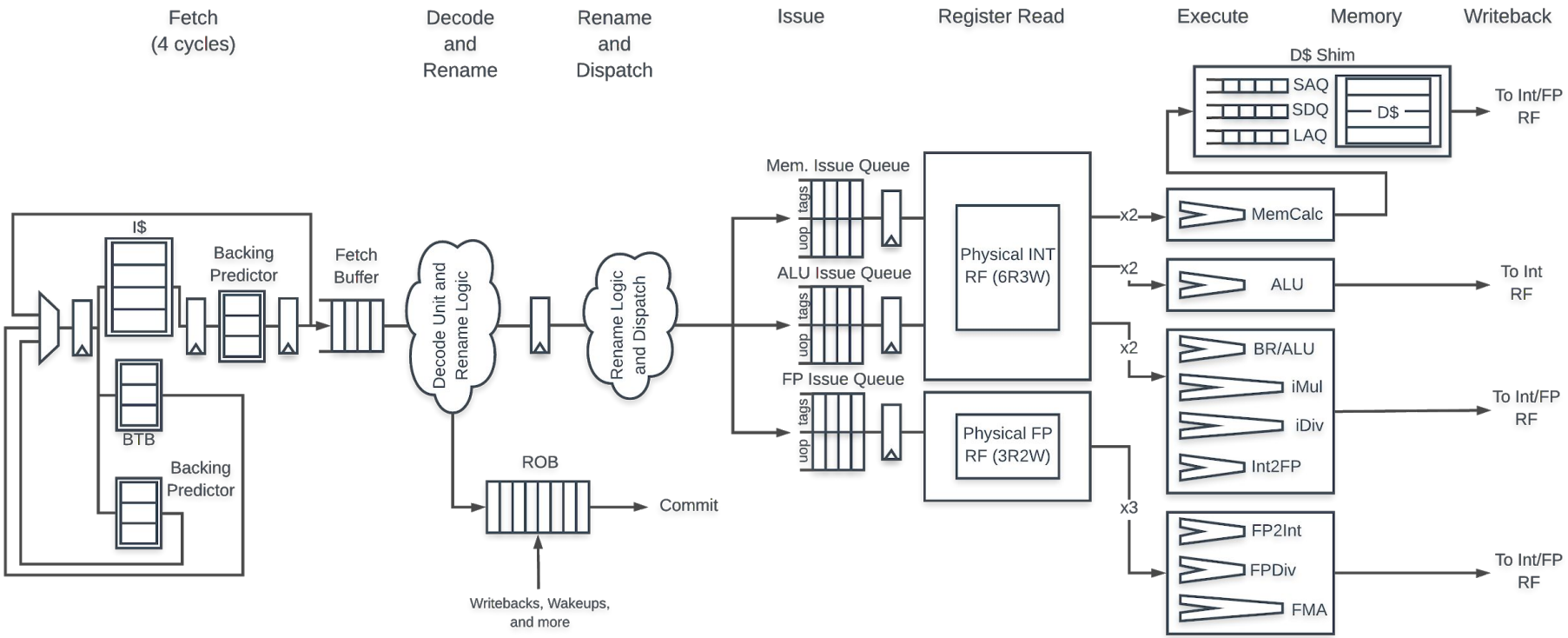
Lab 3 Goals

- Take an out-of-order core microarchitecture that has many parameters
- Vary those parameters and see how the performance is affected
  - Branch predictor
  - ROB size
  - Number of physical registers
  - Et cetera

The Berkeley Out-of-Order Machine (BOOM)

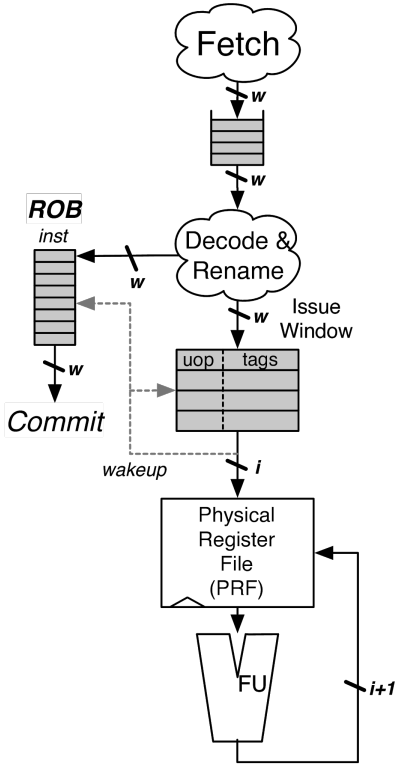
BOOM Pipeline Overview

DefaultBoomConfig

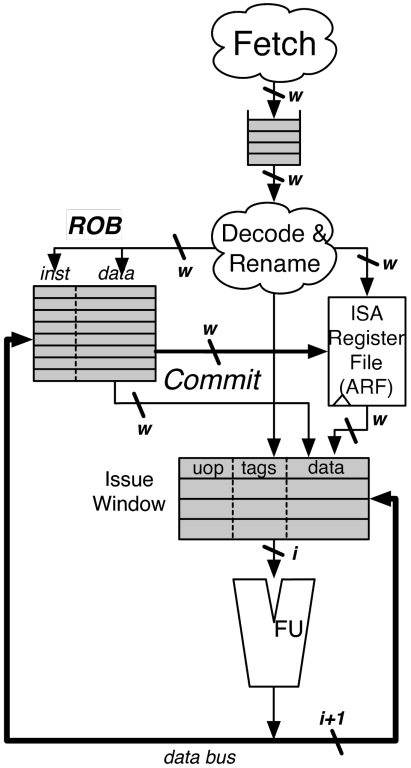


Type of Machine

BOOM



Unified Physical Register File



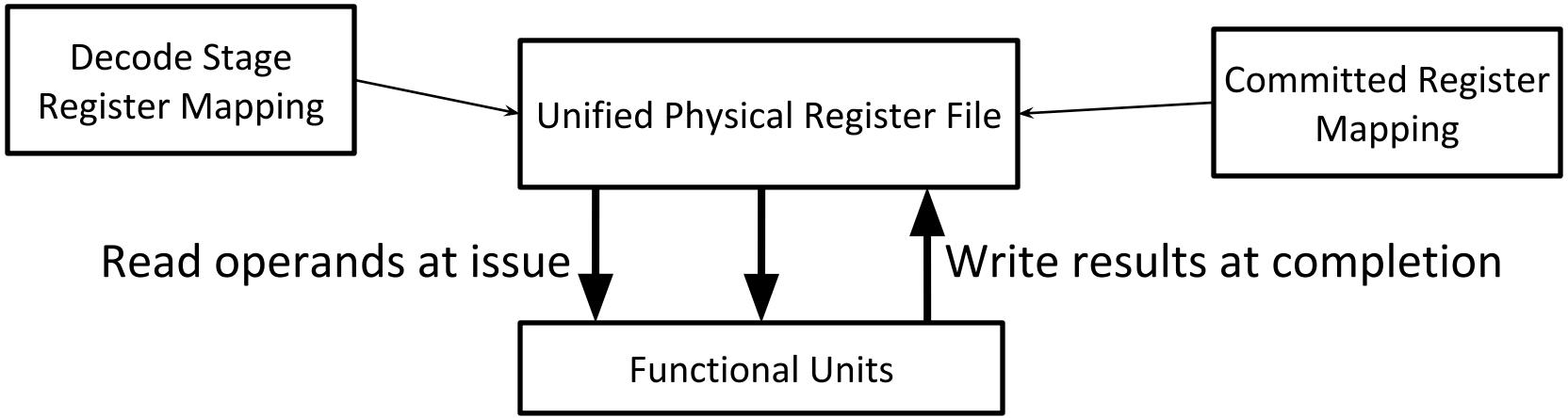
Data in ROB

Out-of-Order Design Space

Unified Physical Register File

(MIPS R10K, Alpha 21264, Intel Pentium 4 & Sandy/Ivy Bridge)

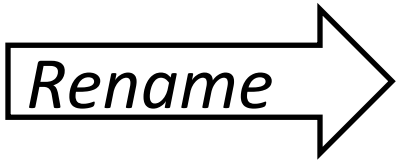
- Rename all architectural registers into a single *physical* register file during decode, no register values read
- Functional units read and write from single unified register file holding committed and temporary registers in execute
- Commit only updates mapping of architectural register to physical register, no data movement



Lifetime of Physical Registers

- Physical regfile holds committed and speculative values
- Physical registers decoupled from ROB entries *(no data in ROB)*

```
ld x1, (x3)
addi x3, x1, #4
sub x6, x7, x9
add x3, x3, x6
ld x6, (x1)
add x6, x6, x3
sd x6, (x1)
ld x6, (x11)
```

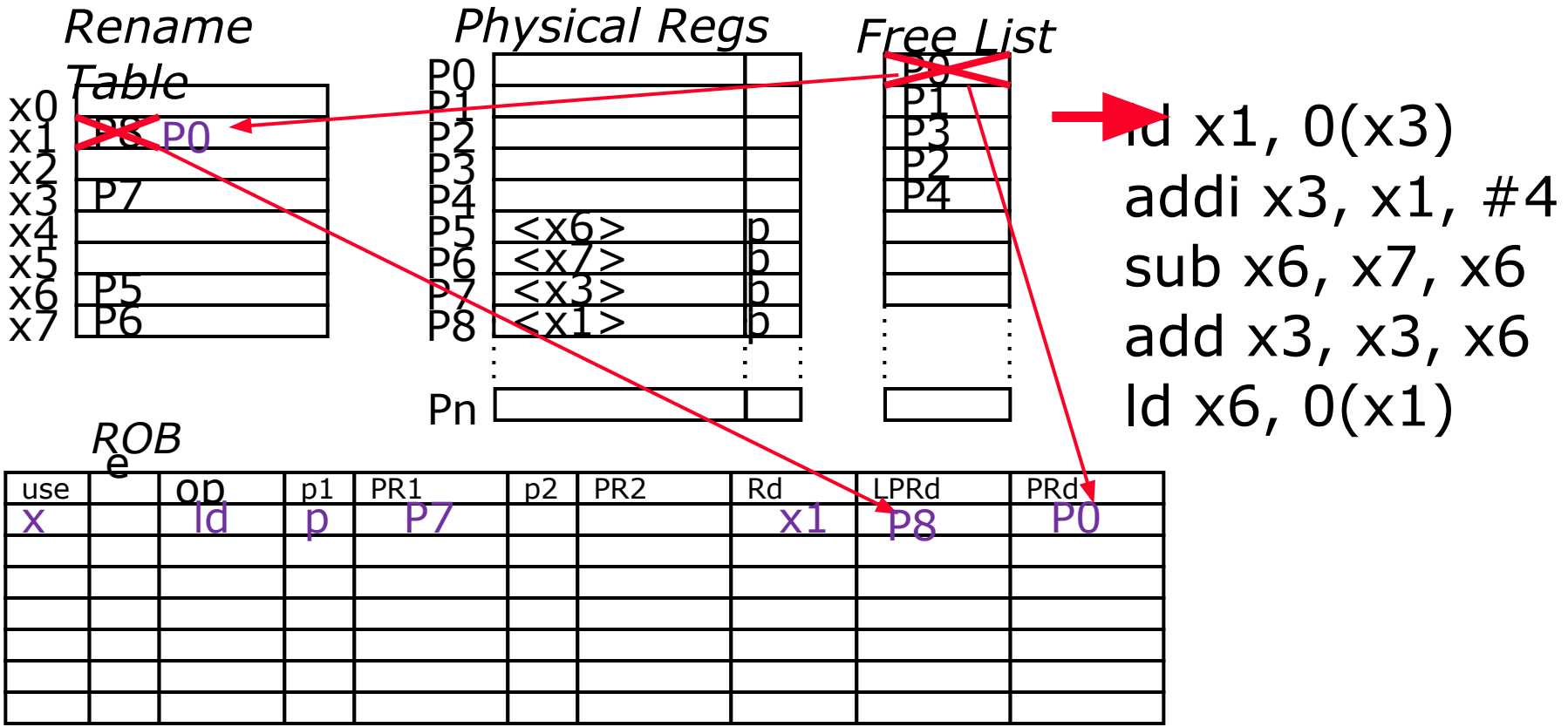



```
ld P1, (Px)
addi P2, P1, #4
sub P3, Py, Pz
add P4, P2, P3
ld P5, (P1)
add P6, P5, P4
sd P6, (P1)
ld P7, (Pw)
```

When can we reuse a physical register?

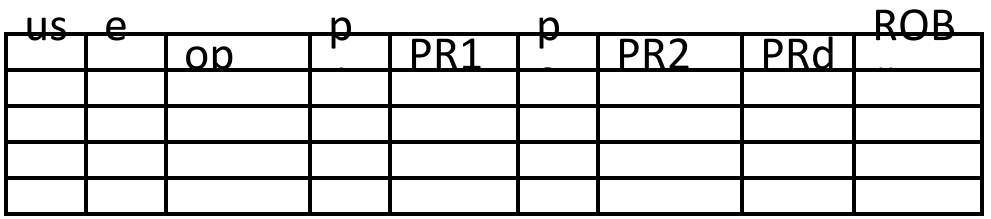
When next writer of same architectural register commits

Physical Register Management

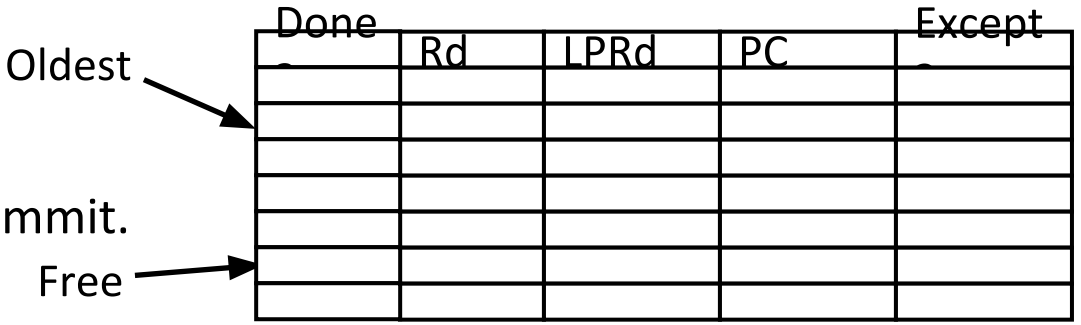


Separate Issue Window from ROB

The issue window holds only instructions that have been decoded and renamed but not issued into execution. Has register tags and presence bits, and pointer to ROB entry.



Reorder buffer used to hold exception information for commit.



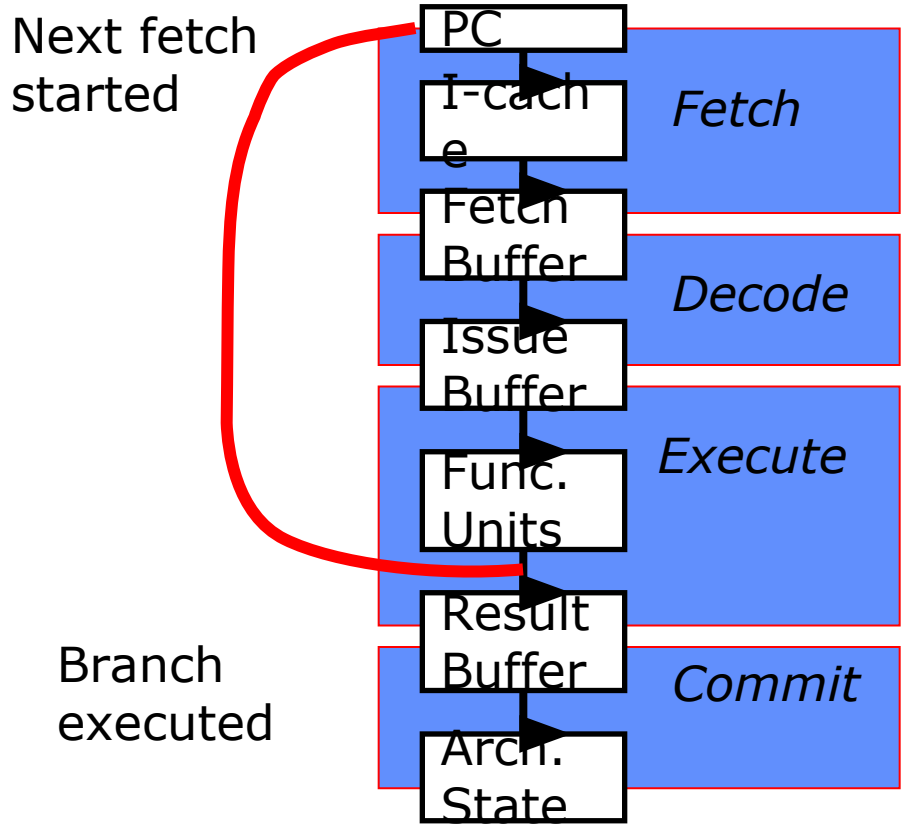
ROB is usually several times larger than issue window – why?

Control Flow Penalty

Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution !

How much work is lost if pipeline doesn't follow correct instruction flow?

~ Loop length x pipeline width + buffers



Reducing Control Flow Penalty

- Software solutions
  - Eliminate branches - loop unrolling
    - Increases the run length
  - Reduce resolution time - instruction scheduling
    - Compute the branch condition as early as possible (of limited value because branches often in critical path through code)
- Hardware solutions
  - Find something else to do (delay slots)
    - Replaces pipeline bubbles with useful work (requires software cooperation) – quickly see diminishing returns
  - Speculate - branch prediction
    - Speculative execution of instructions beyond the branch
    - Many advances in accuracy, widely used

Branch Prediction

Motivation:

Branch penalties limit performance of deeply pipelined processors
Modern branch predictors have high accuracy
(>95%) and can reduce branch penalties significantly

Required hardware support:

Prediction structures:

- Branch history tables, branch target buffers, etc.

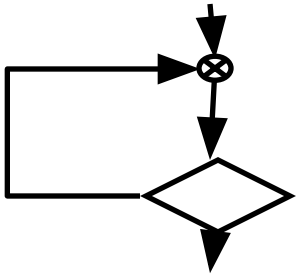
Mispredict recovery mechanisms:

- *Keep result computation separate from commit*
- Kill instructions following branch in pipeline
- Restore state to that following branch

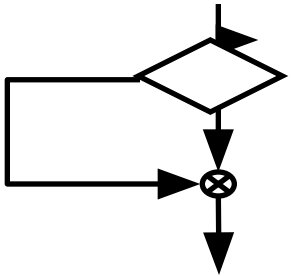
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

backward
90%



forward
50%



A basic “null predictor” can always pick PC+4!

Dynamic Branch Prediction: learning based on past behavior

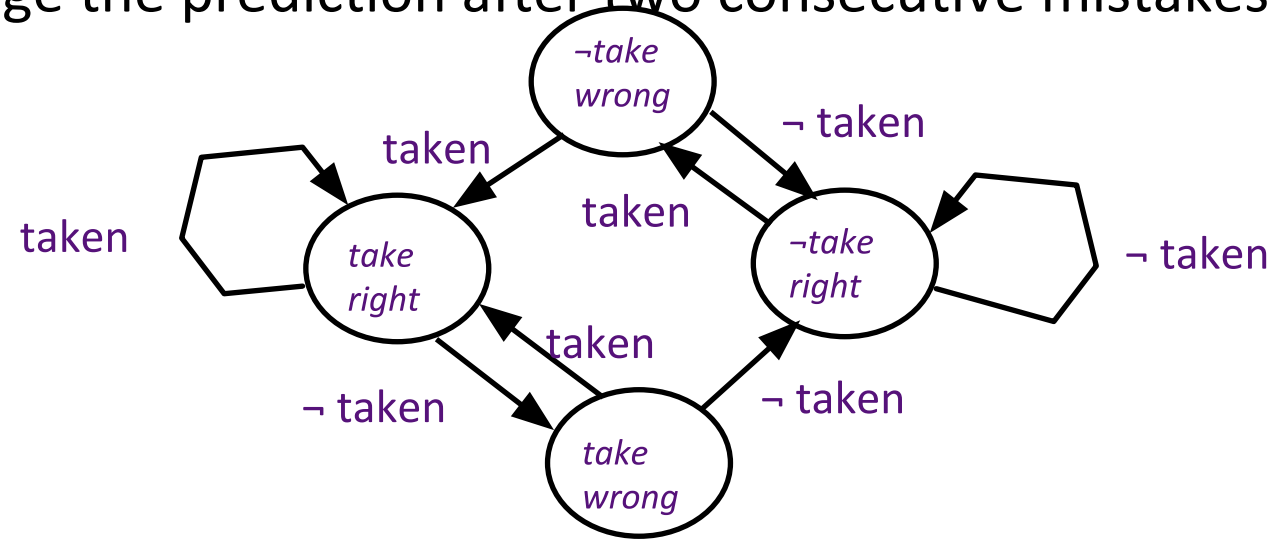
- Temporal correlation
  - The way a branch resolves may be a good predictor of the way it will resolve at the next execution
- Spatial correlation
  - Several branches may resolve in a highly correlated manner (a preferred path of execution)

One-Bit Branch History Predictor

- For each branch, remember last way branch went
- Has problem with loop-closing backward branches, as two mispredicts occur on every loop execution
  1. first iteration predicts loop backwards branch not-taken (loop was exited last time)
  2. last iteration predicts loop backwards branch taken (loop continued last time)

Branch Prediction Bits

- Assume 2 BP bits per instruction for Finite State Machine
- Change the prediction after two consecutive mistakes!



BP state:

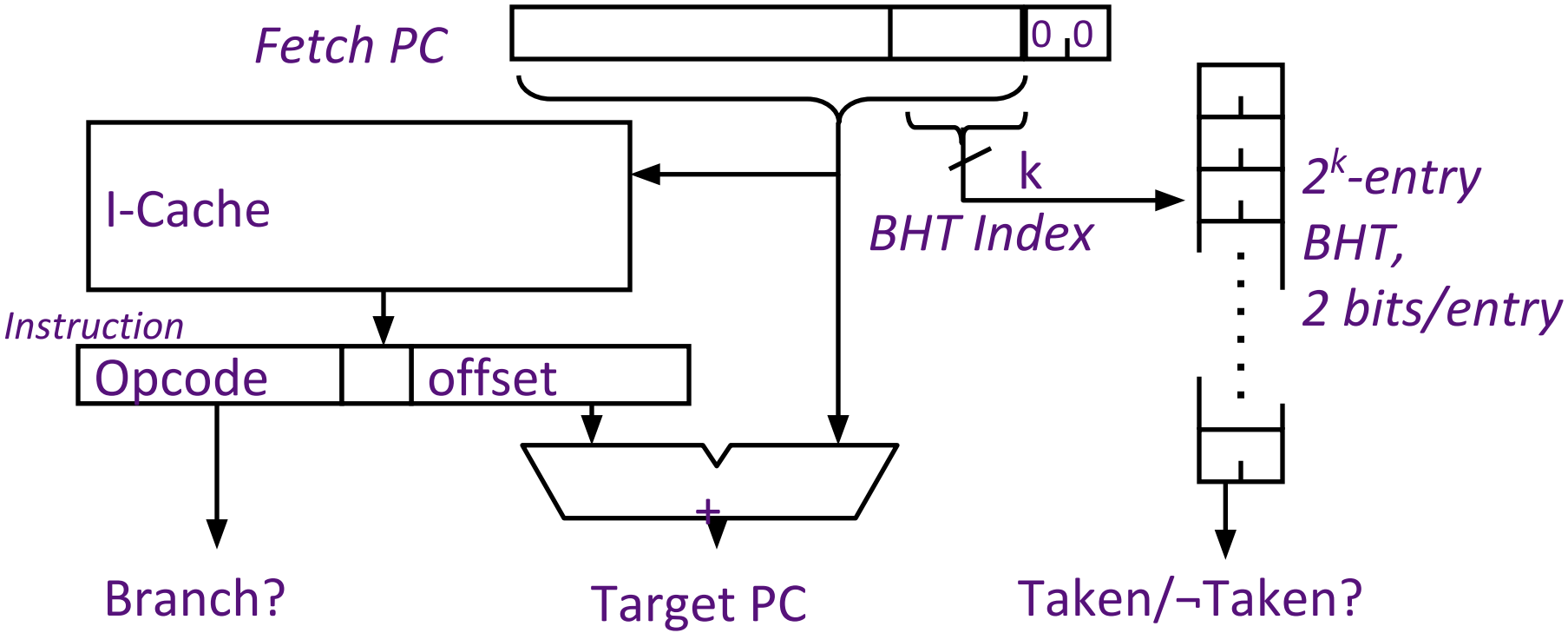
(*predict* take/¬take) x (*last prediction* right/wrong)

Branch Prediction Bits

- Why not store a copy of this FSM for every instruction?

Must keep a table that is direct-mapped by some of the PC bits, with each row holding a copy of FSM state.

Branch History Table (BHT)



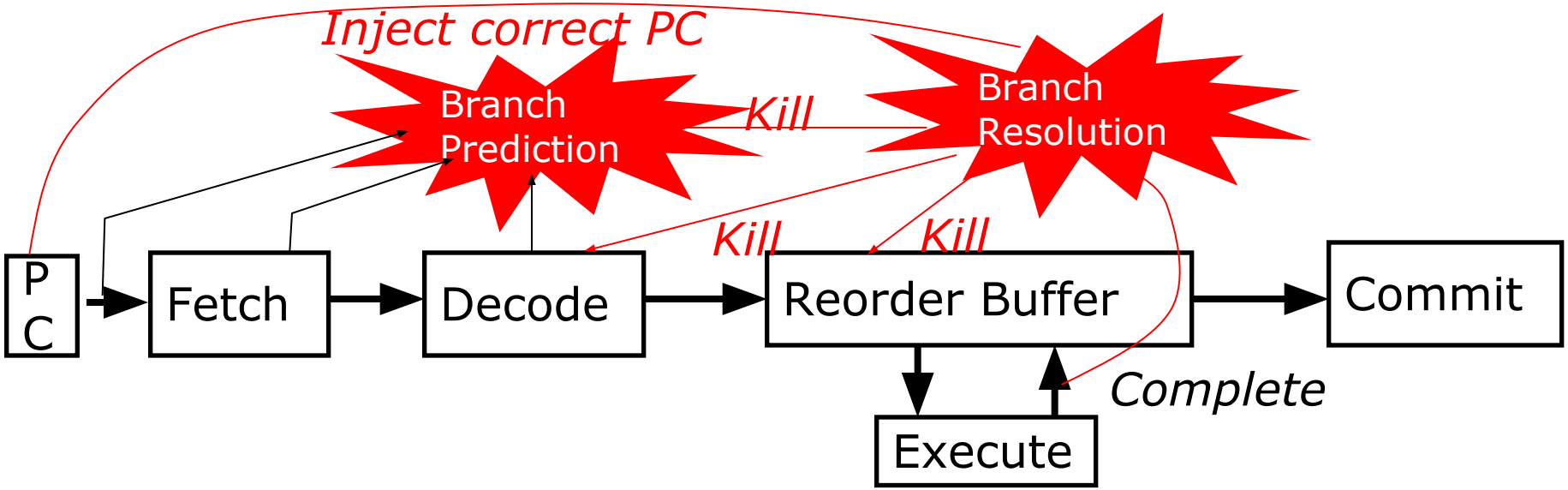
4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

Branch Prediction Penalties

- Even if you have perfect prediction, when does it happen?

- What happens if you mispredict?

Branch Misprediction in Pipeline



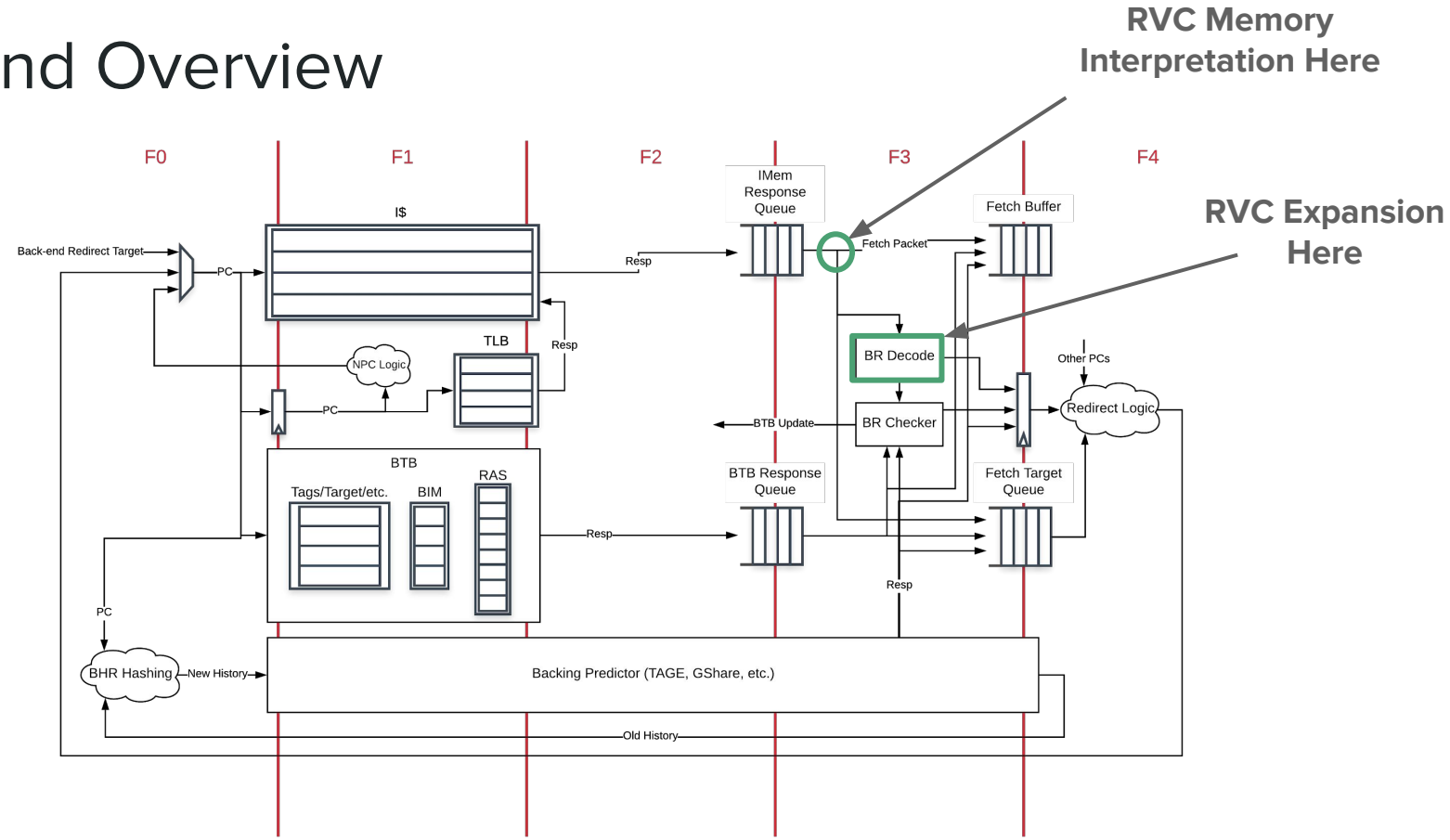
- Can have multiple unresolved branches in ROB
- Can resolve branches out-of-order by killing all the instructions in ROB that follow a mispredicted branch
- MIPS R10K uses 4 mask bits to tag instructions that are dependent on up to 4 speculative branches
- Mask bits cleared as branch resolves, and reused for next branch

Rename Table Recovery

- Have to quickly recover rename table on branch mispredicts
- BOOM has a parameterizable number of snapshots for each of four outstanding speculative branches

More BOOM Detail

Frontend Overview



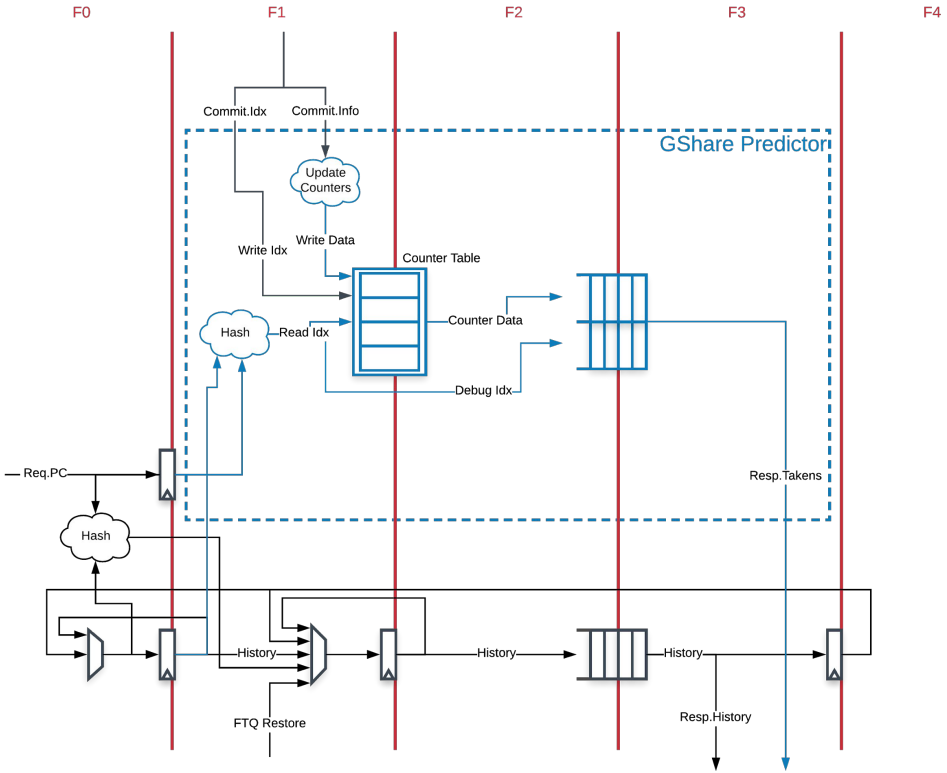
Frontend Description

- Cycle 0
  - Get the PC to pass into the BTB/BPD and I$
  - Hash the GHR outside the “backing predictor”
- Cycle 1
  - I$ is getting a line
  - BTB makes a prediction on the PC
  - 1st stage of “backing predictor”
    - Ex. GShare accesses counter tables
- Cycle 2
  - 2nd stage of “backing predictor”
    - Ex. GShare puts counter output into queue
- Cycle 3
  - BR decode I$ result (not “true” decode)
  - BR check to update BTB and redirect
  - Enqueue the Fetch Buffer and Fetch Target Queue with instructions and BR info
- Fetch Buffer
  - Connects Front-end to Back-end
  - Passes Fetch Packet of instructions to the Decode stage
- Fetch Target Queue
  - Previously called the Branch Reorder Buffer
  - Holds branch prediction information used to update the BPD on commit, mispredictions, speculation
  - Dequeued on commit once all instructions in an entry are committed
- RVC in Cycle 3
  - Puts a full Fetch Packet into the Fetch Buffer depending on where instruction is aligned
  - Used in Decoder to “expand” instructions

Branch Predictor Pipeline

- GHR management is kept outside the “abstract” predictor
  - Keeps track of global branch state
  - Keep “snapshot” or copy of GHR per “fetch packet”
- Abstract Predictor
  - TAGE
  - GShare
  - BaseOnly
    - Only use BIM from BTB to predict
  - Random
    - Using LSFR to create random predictions
  - Null predictor
    - Don’t predict anything

GShare Example



Backend

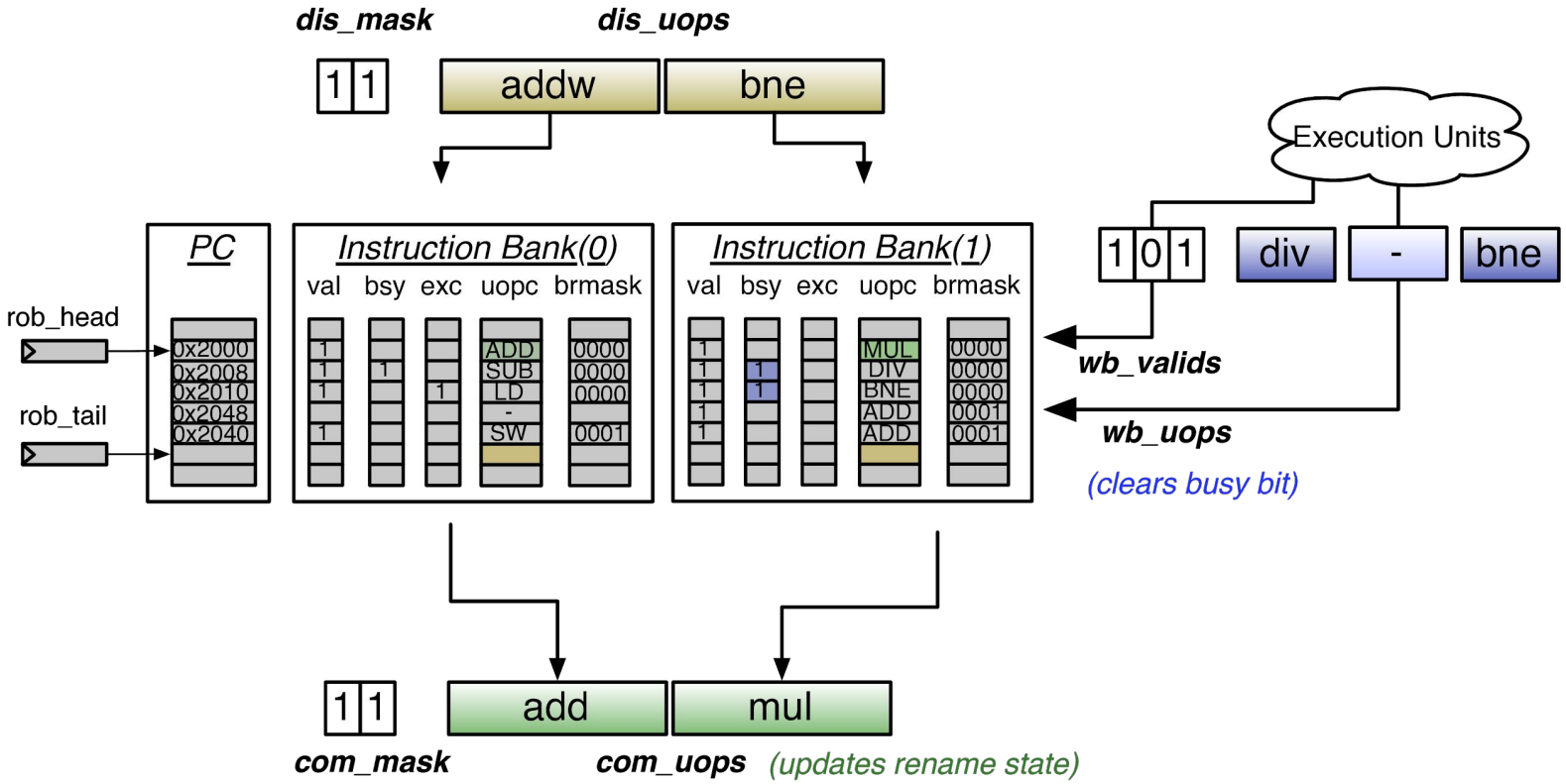
Decode and Rename

- Simple Decode
  - Thanks RISC-V!
  - RVC instructions are expanded here to go through the normal decode
- Rename
  - Split into FP and INT Rename
    - Both are coupled by the Decode Width
  - Map Table - maps logical registers (x0,...x31) to physical registers
    - Otherwise known as the “rename table”
  - Busy Table - marks physical registers as “busy” until operands are ready
  - Free List - running list of physical registers that are free
- Complete Map table snapshots are kept for each BR instruction in the ROB
  - Can recover in a single cycle

Dispatch and Issue

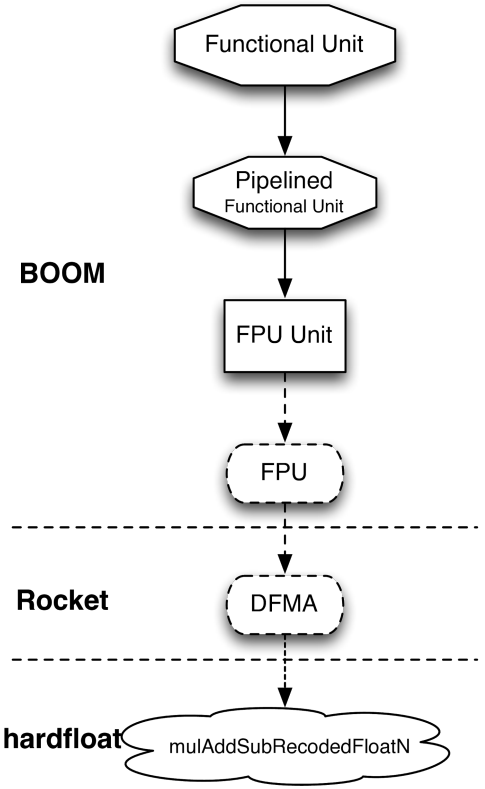
- Multiple Issue Queues for each type of micro-operation (uop)
  - Memory
  - Integer (ALU)
  - Floating Point
  - Note: Now you can have a monolithic Memory + Integer issue queue
- Once operands are ready, then the uop is issued (this means reading the necessary registers and moving to the execution stage)
- Wakeup (ready) ports coming from the Exe Units are automatically wired to the issue queues

ROB

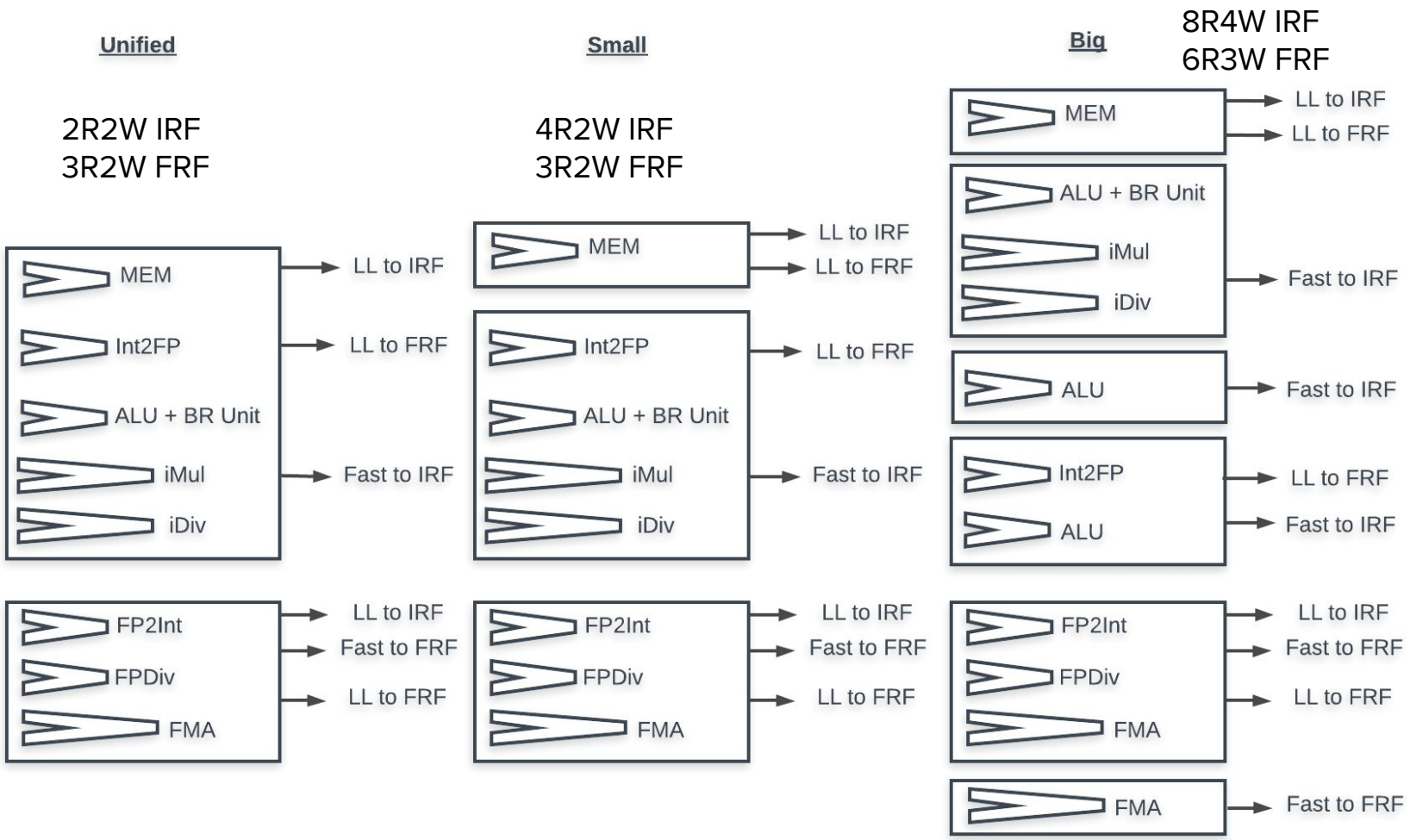

dis_mask
dis_uops
1 1
addw
bne
Execution Units
PC
Instruction Bank(0)
Instruction Bank(1)
val bsy exc uopc brmask
rob_head
rob_tail
0x2000
0x2008
0x2010
0x2048
0x2040
ADD
SUB
LD
-
SW
0000
0000
0000
0001
MUL
DIV
BNE
ADD
ADD
0000
0000
0000
0001
0001
1 0 1
div
-
bne
wb_valids
wb_uops
(clears busy bit)
1 1
add
mul
com_mask
com_uops
(updates rename state)

Execution Units

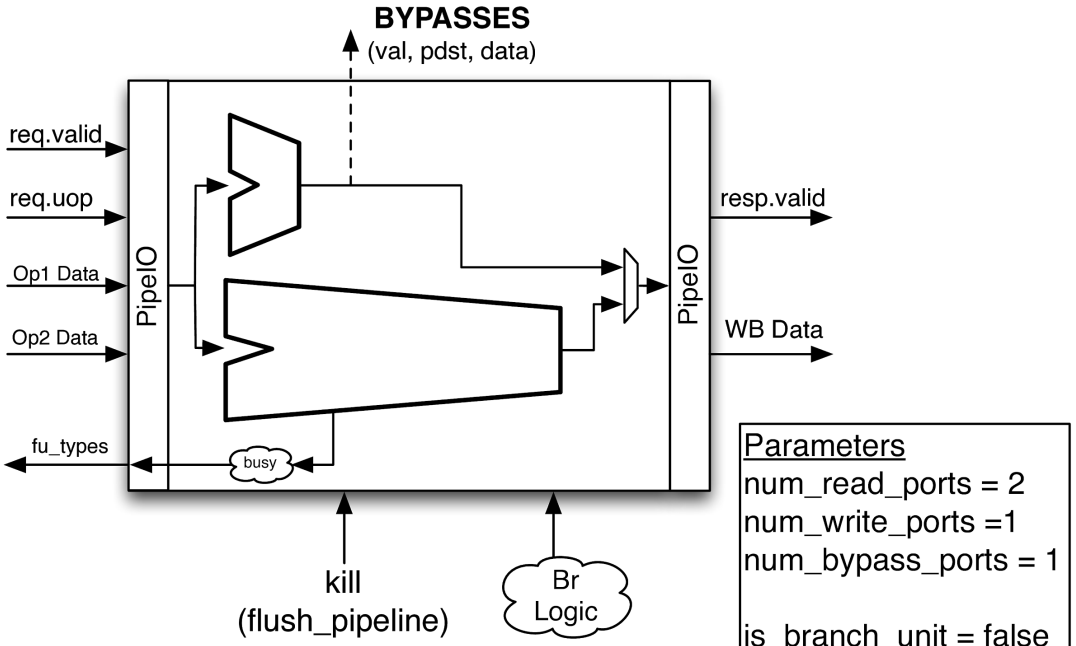
- Execution Unit
  - Top level execution unit that shares read port and write ports
  - Can be composed of multiple “smaller” functional units
  - May reserve a dedicated RF write port, or share the long-latency writeback with memory
- Functional Unit
  - Wraps external units into pipelined functional units
  - Auto-generates pipeline regs for uop metadata
  - Auto-generates misspeculation kill logic



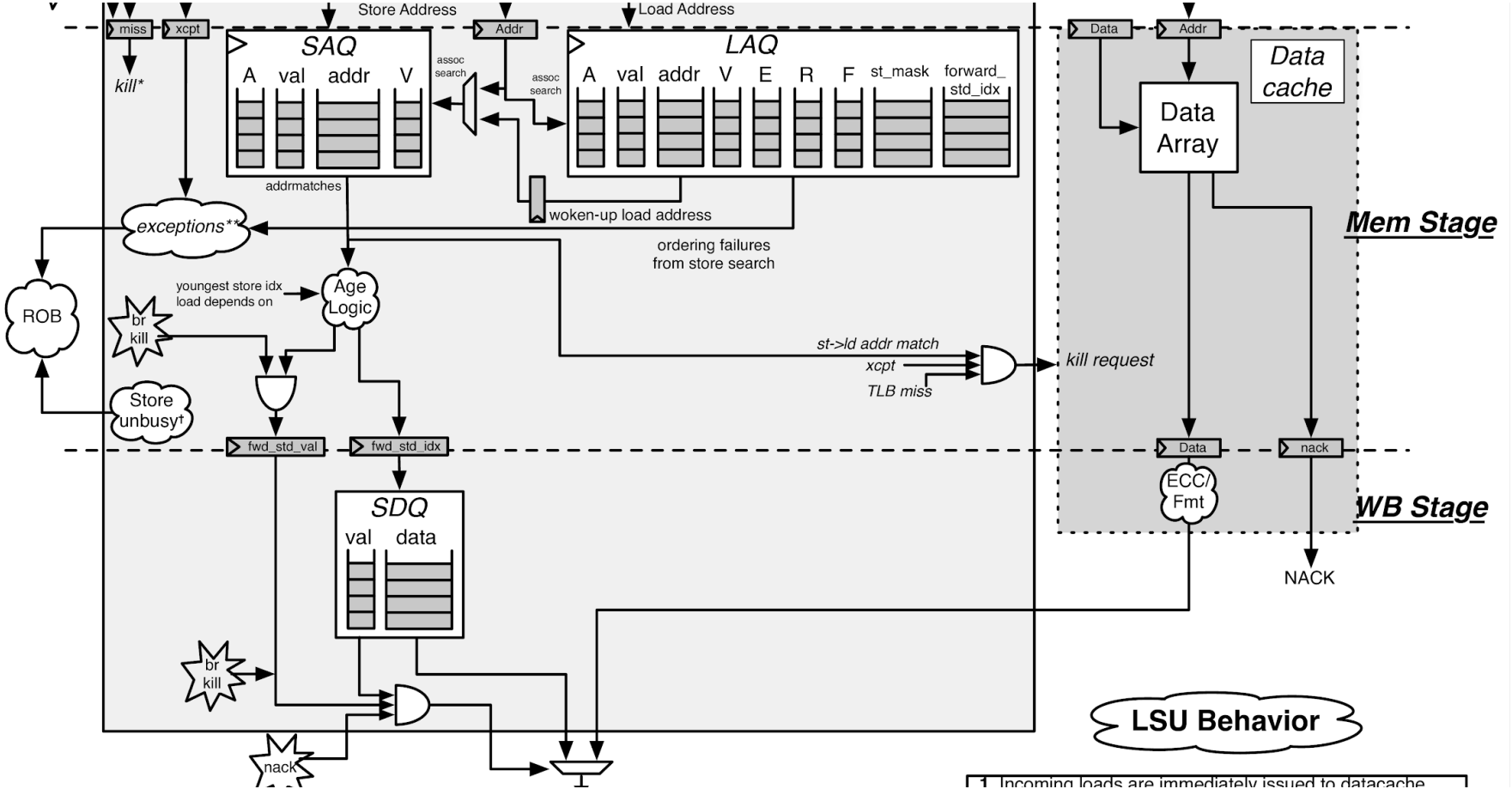
Functional Unit Hierarchy


Unified
2R2W IRF
3R2W FRF
MEM
Int2FP
ALU + BR Unit
iMul
iDiv
LL to IRF
LL to FRF
Fast to IRF
FP2Int
FPDiv
FMA
LL to IRF
Fast to FRF
LL to FRF
Small
4R2W IRF
3R2W FRF
MEM
Int2FP
ALU + BR Unit
iMul
iDiv
LL to IRF
LL to FRF
Fast to IRF
FP2Int
FPDiv
FMA
LL to IRF
Fast to FRF
LL to FRF
Big
8R4W IRF
6R3W FRF
MEM
LL to IRF
LL to FRF
ALU + BR Unit
iMul
iDiv
Fast to IRF
ALU
Fast to IRF
Int2FP
ALU
LL to FRF
Fast to IRF
FP2Int
FPDiv
FMA
LL to IRF
Fast to FRF
LL to FRF
FMA
Fast to FRF

Execution Units



Pipelined Functional Unit


Store Address
Load Address
miss
xcpt
Addr
SAQ
A val addr V
assoc search
LAQ
A val addr V E R F st_mask forward_ std_idx
kill*
addrmatches
woken-up load address
exceptions**
ordering failures from store search
youngest store idx load depends on
Age Logic
ROB
br kill
Store unbusy†
st->ld addr match
xcpt
TLB miss
kill request
Data
Addr
Data cache
Data Array
Mem Stage
fwd_std_val
fwd_std_idx
SDQ
val data
Data
nack
ECC/ Fmt
WB Stage
NACK
br kill
nack
LSU Behavior
1. Incoming loads are immediately issued to datacache

Memory System

- SAQ
  - Holds store addresses
  - Content Addressable Memory (CAM) to associatively search for conflicts
  - Incoming loads must search for dependent stores
- SDQ
  - Hold store data
- LAQ
  - Holds load addresses
  - CAM for conflict detection
  - Incoming loads must search for earlier loads
- D$ Shim
  - Wraps the underlying rocketchip L1 cache
- Rocket non-blocking L1 cache