Module 3 – Frontend

(ECE 209) Secure and Advanced Computer Architecture

Nader Sehatbakhsh
Department of Electrical and Computer Engineering
University of California, Los Angeles

Last Time

- We reviewed basics topics in computer architecture.
- If you need more background, you can review things here (L1-L8): *https://ssysarch.github.io/ECE_M116C-CS_M151B/F23/schedule.html*

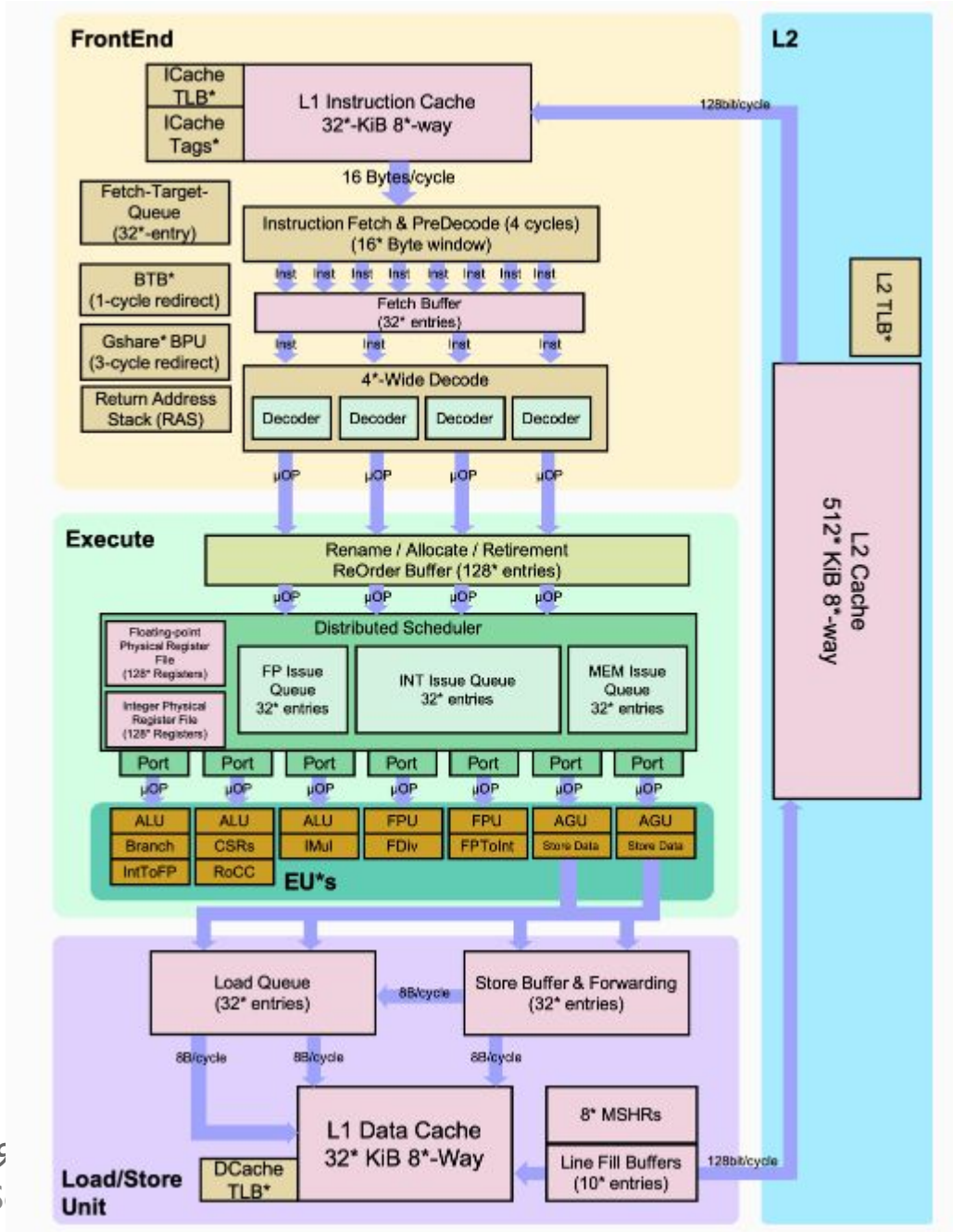
❖ Or read *David A. Patterson and John L. Hennessy, Computer Organization and Design: the Hardware/Software Interface: RISC-V Edition*

Today

- Discuss more advanced concepts in computer architecture.
  - Out-of-order execution
    - Front-end
    - Execution engine
    - Memory subsystem

BOOM Core

- Out-of-order execution



BOOM Core

- We will focus on third generation (v3) aka SonicBoom.

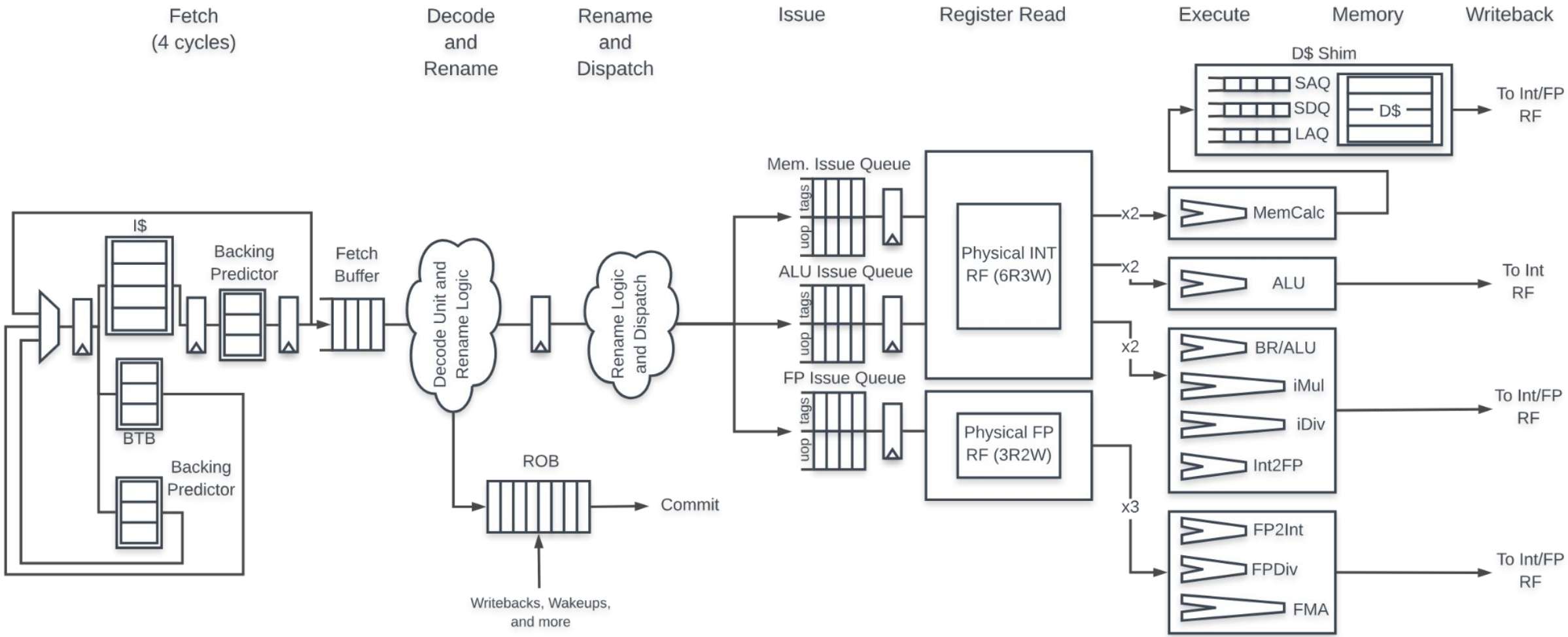
- More informations here:
  - https://docs.boom-core.org/en/latest/sections/intro-overview/boom.html
  - https://github.com/riscv-boom
  - https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf
  - https://boom-core.org/docs/HC30.Berkeley.Celio-Chiu.final.pdf

Frontend

- What is frontend?
  - Fetch and decode
    - Fetching
    - Caching
    - Branch prediction
    - Parallel decoding

- What are the main concern?
  - How to make things faster! (as always)

Simplified Pipeline



BOOM Core Pipeline Stages

- Conceptually, BOOM is broken up into 10 stages: *Fetch, Decode, Register Rename, Dispatch, Issue, Register Read, Execute, Memory, Writeback, and Commit.*

- More simplified design has seven stages: *Fetch, Decode/Rename, Rename/Dispatch, Issue/RegisterRead, Execute, Memory, and Writeback* (Commit occurs asynchronously, so it is not counted as part of the "pipeline")

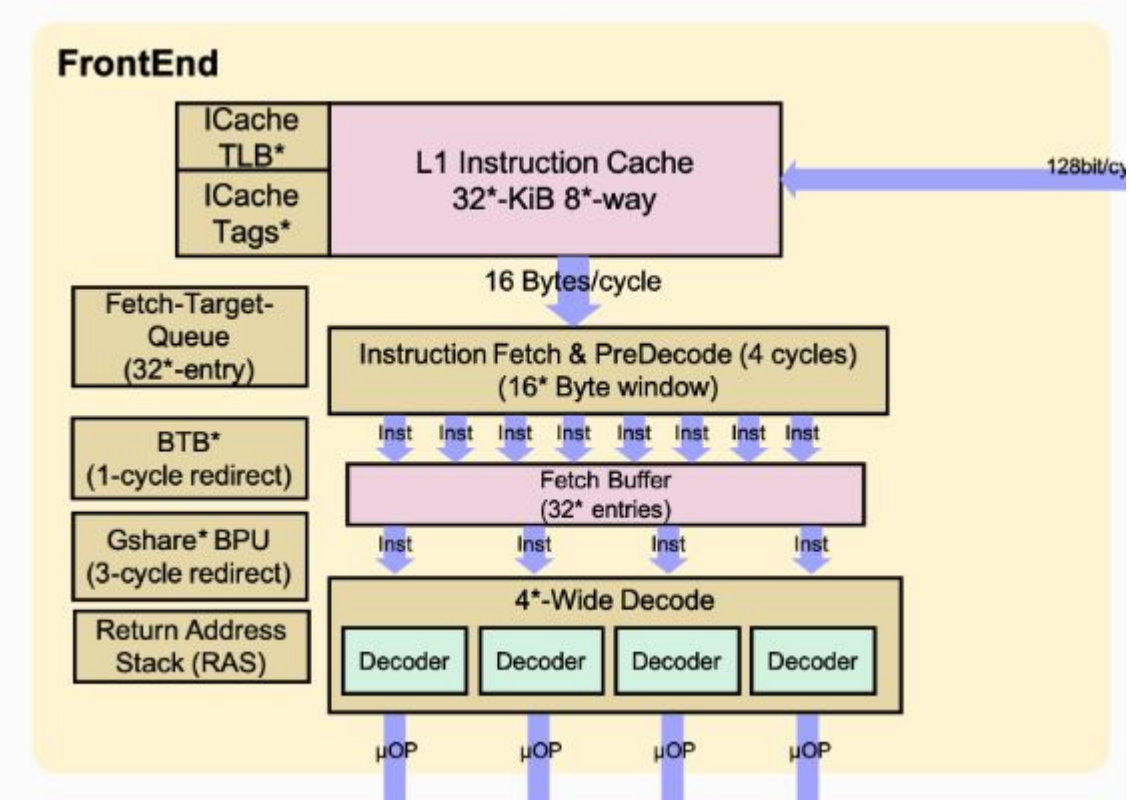
Frontend

- Fetch
- Decome
- Rename

Frontend - Basic Operations

- Fetch
  - Instructions are fetched from instruction memory and pushed into a FIFO queue, known as the Fetch Buffer. Branch prediction also occurs in this stage, redirecting the fetched instructions as necessary.
- Decode
  - Pulls instructions out of the Fetch Buffer and generates the appropriate Micro-Op(s) (UOPs) to place into the pipeline.
- Rename
  - The ISA, or "logical", register specifiers (e.g., x0-x31) are then renamed into "physical" register specifiers.

More Detailed View



Let's talk about each in more detail

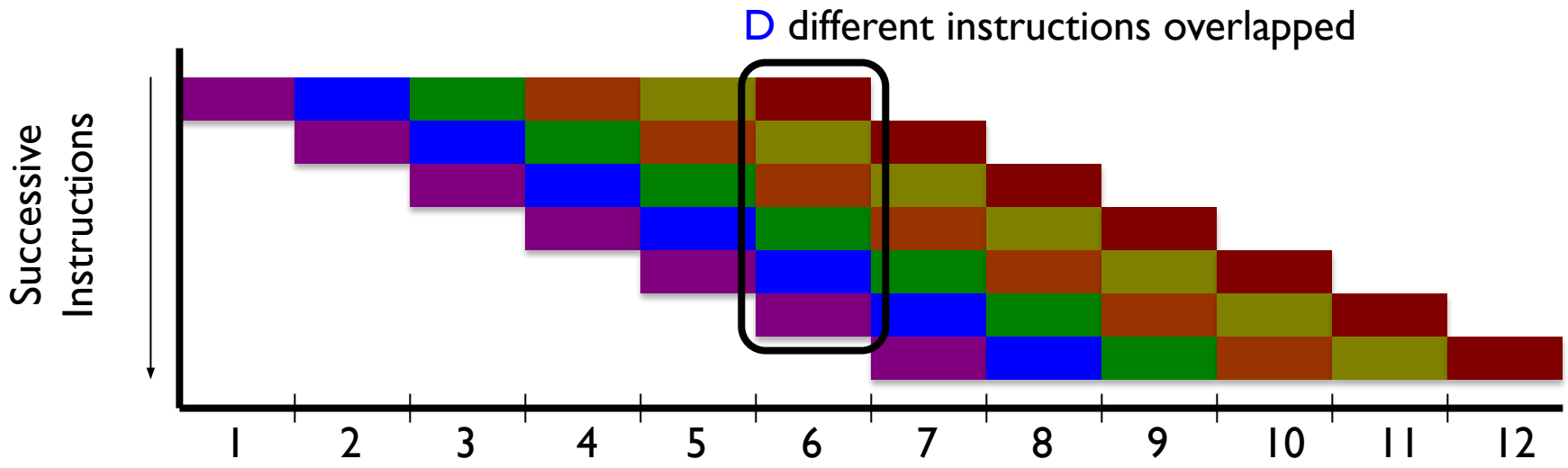
Improving Front-End Performance?

Improving Front-End Performance

- How to improve?
  - Fetch more instructions at one time
  - Increase I-cache performance
  - Increase fetch-width
  - Increase useful instructions per fetch
  - Branch predictor
    - Improving accuracy
    - Improving the access time

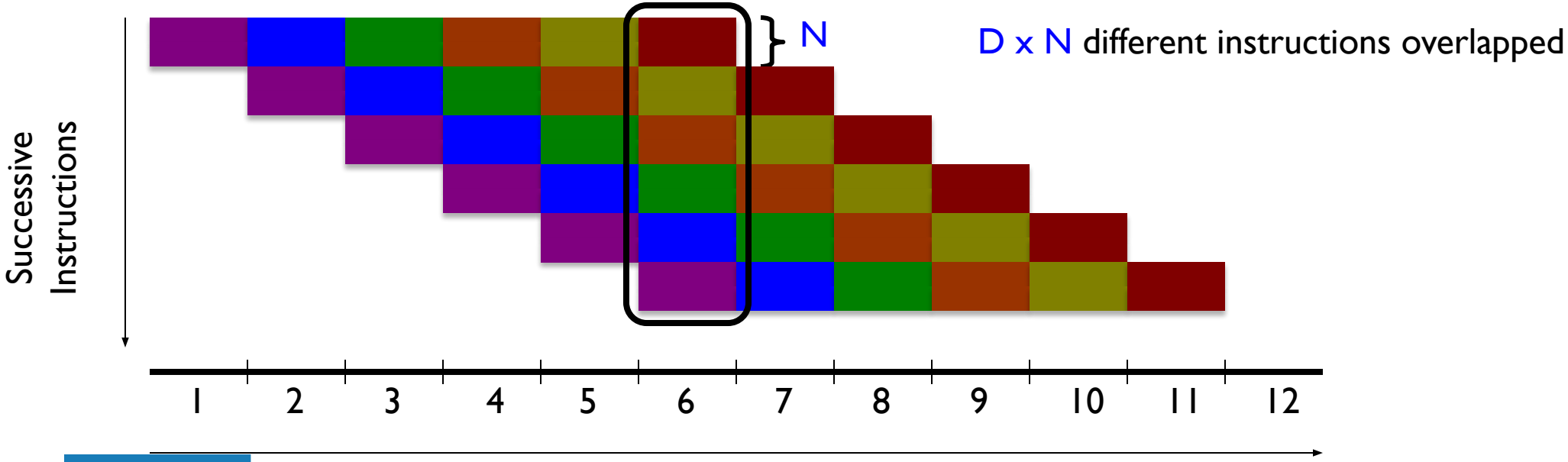
Scalar

- Scalar pipeline (baseline)
  - Instruction/overlap parallelism = D
  - Operation Latency = 1
  - Peak IPC = 1



SuperScalar

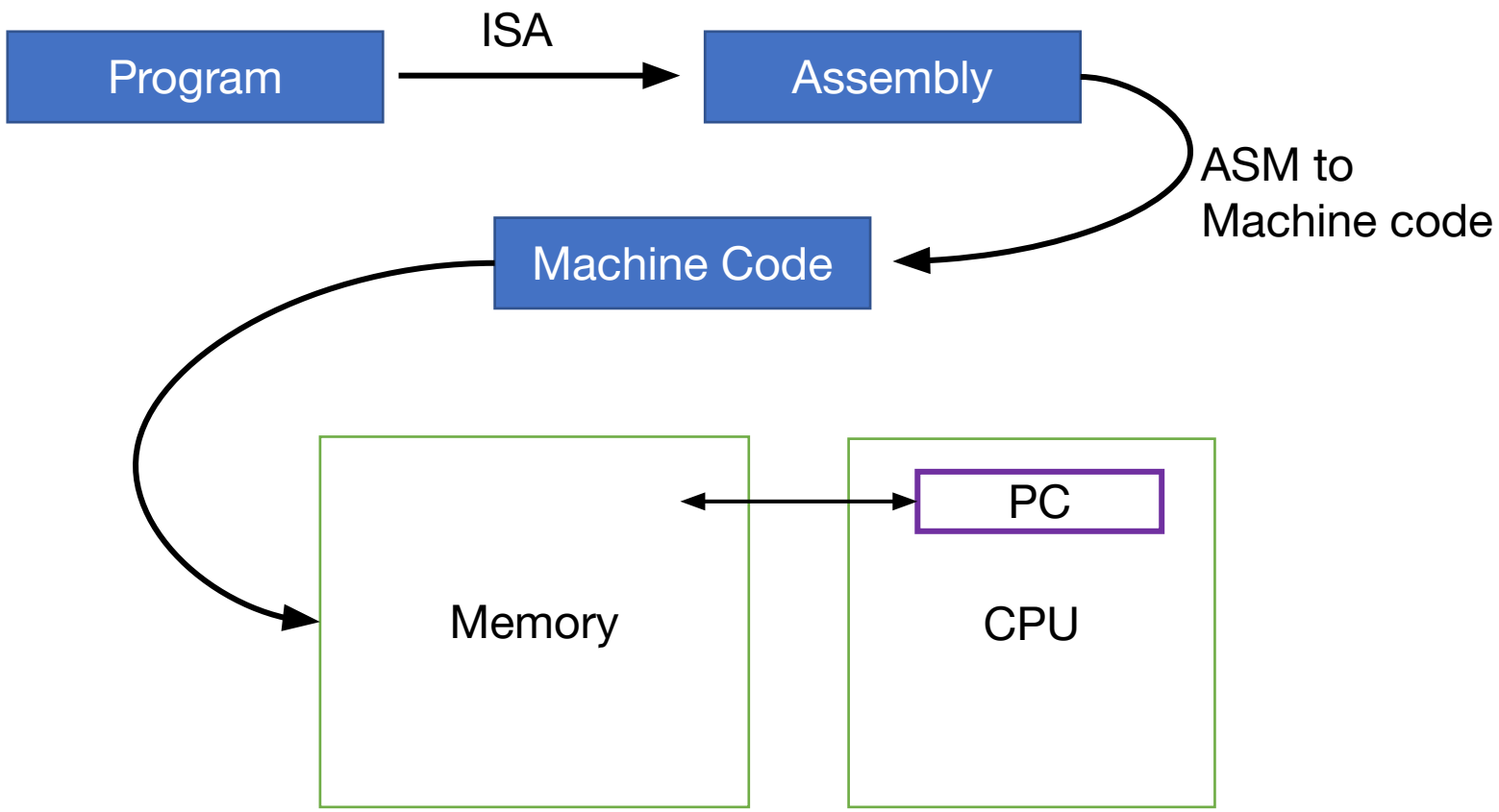
- Instruction parallelism = D x N
- Operation Latency = 1
- Peak IPC = N per cycle



Fetching

- We need to provide N instructions in each cycle to achieve IPC=N
- Do we get in reality? (why?)
- What are the tradeoffs?

Basic Operation of Fetching



PC

- Keep track of the program.
- Which instructions are fetched/executed are controlled by PC.
- Jumps affect PC.
- Normally, things are executed linearly.
- Memory is usually byte-addressable hence an N-byte instruction is stored in N consecutive lines in the memory → next address should be fetched from PC+N.
  - N = 32 or 64.
  - Instructions should be aligned.

Reading from Memory

- What is the main problem?
- How to solve it?
- Latency vs. bandwidth

Quick Detour

- Memory Hierarchy

Memory "Wall"

- CPU is much faster than memory.
  - Accessing DRAM takes 1000s of cycles in modern processors.
- CPU speed is ***growing*** much faster
  - The performance gap between memory and CPU is growing by around 50% per year.

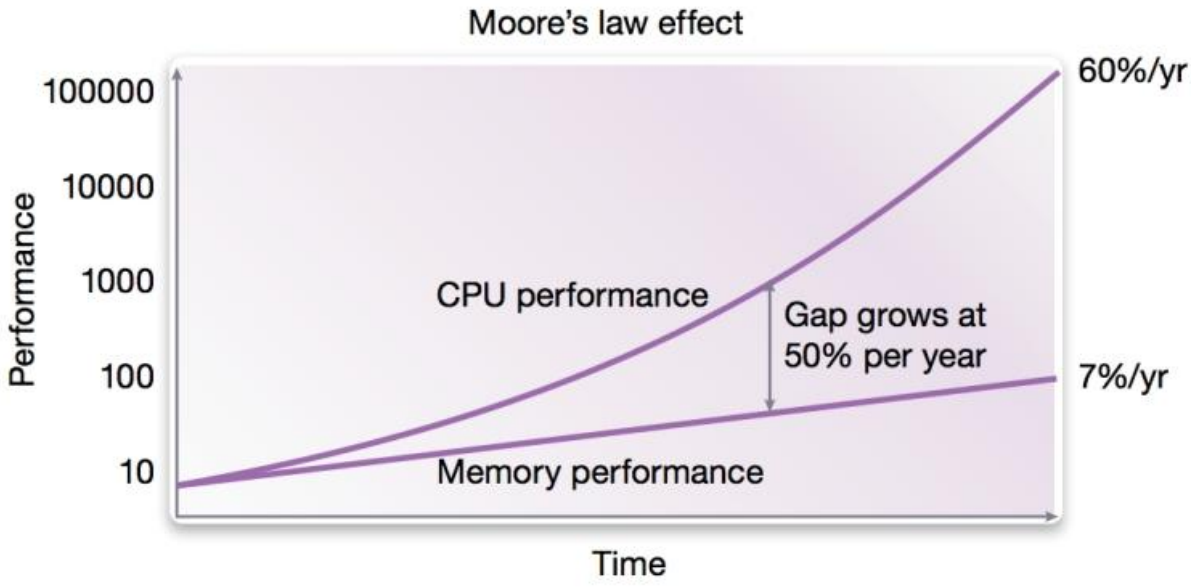


Image taken from:
https://www.extremetech.com/computing/233691-phase-change-memory-can-operate-thousands-of-times-faster-than-current-ram

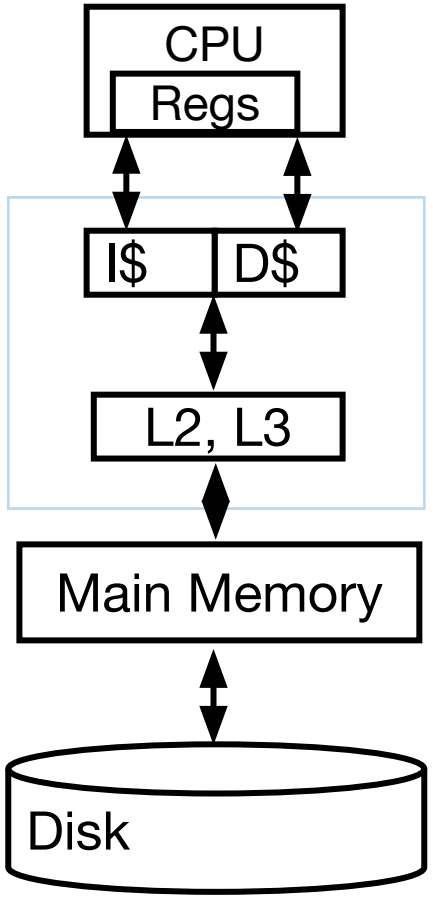
Fundamental Challenge

- Memories can be either large OR fast

Different Storage (Memory)

- Latches and Flip-Flops (aka Registers)
  - Very fast, parallel access.
  - Very expensive (one bit costs ***tens*** of transistors).
- Static RAM (SRAM)
  - Relatively fast, only one data word at a time.
  - Expensive (one bit costs **6+** transistors).
- Dynamic RAM (DRAM)
  - Slower, one data word at a time, reading *destroys* content (refresh), needs special process for manufacturing.
  - Cheap (one bit costs only ***one*** transistor plus one capacitor).
- DISK (*flash memory*, *hard disk*)
  - Much slower, access takes a long time, ***non-volatile.***
  - Very cheap.

Memory Hierarchy

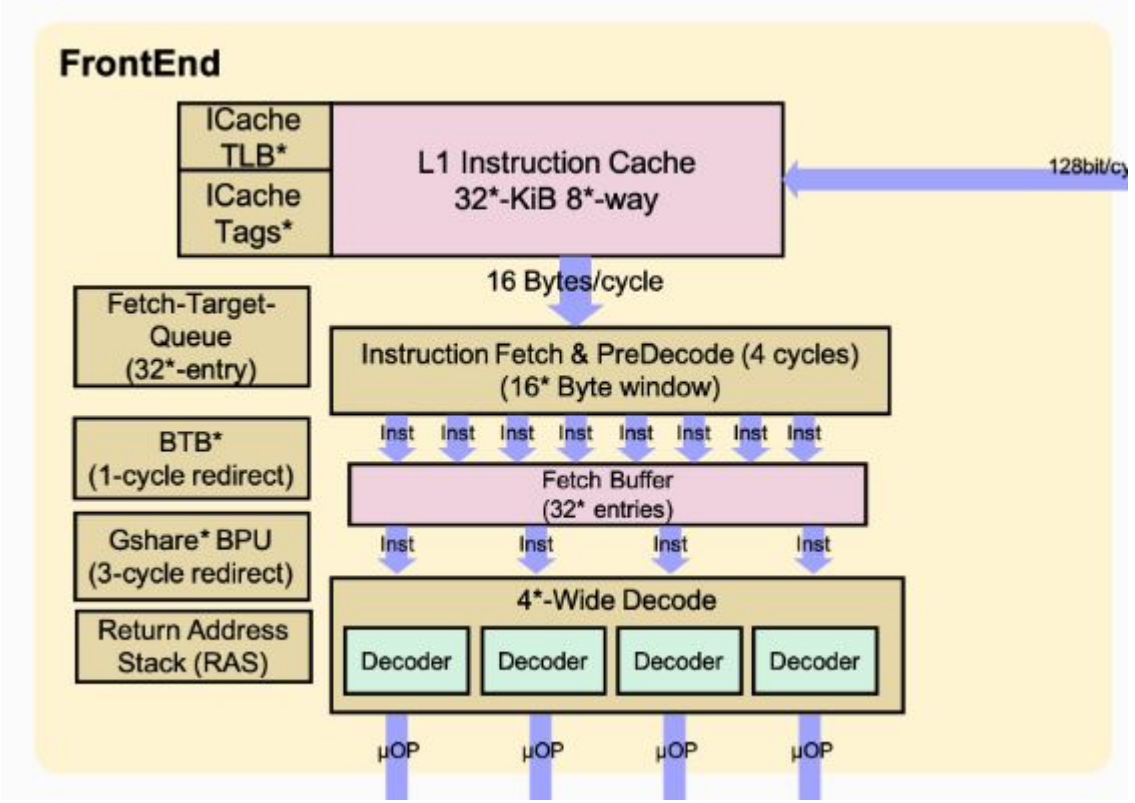


Cache Design

- We will get back to this later, for now assume that things are magically stored there.

Frontend

- Fetch
- Decome
- Rename



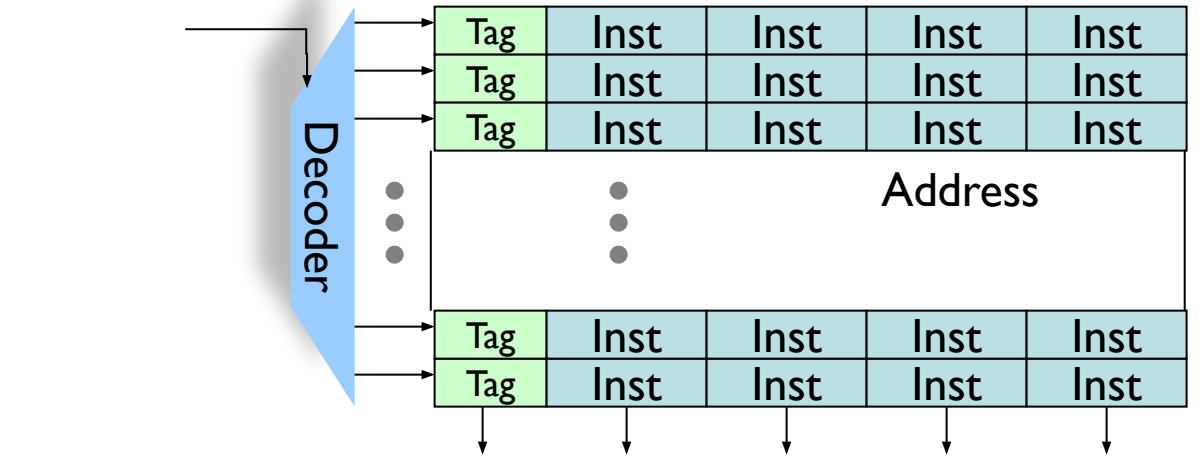
I-Cache Optimizations?

I-Cache Optimizations

- Improve cache performance itself
  - Large cache, improve associativity
  - Code compression
  - Conflict misses less of a problem than in data caches
- Improve spatial locality of cache
  - Large line size
  - Spatial locality inherent in sequential program I-stream
  - Cache prefetching
  - Next-line, streaming buffer
- Improve useful fetch instructions
  - Code layout
  - Branch target (even if not taken)
  - Trace cache

How many instructions should we fetch?

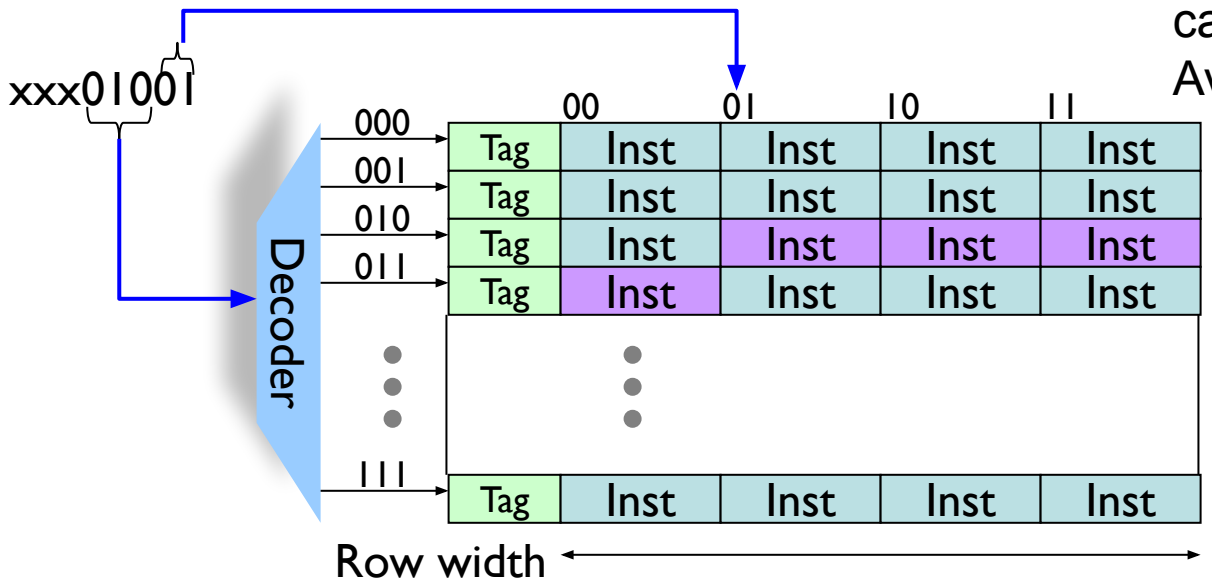
N-fetch Problem



Fetch group: how many instructions are fetched in one cycle

PC = 0xxx 01001 and fetch 4 instructions

N = 4, and 4 instructions can fit into 1 cache line
Average fetch rate is ?

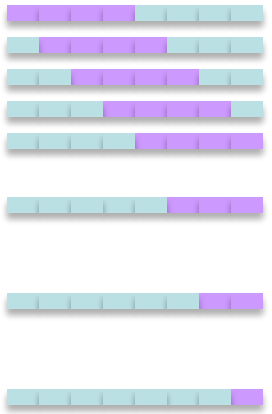


1/4 x 4 + ¼ x 3 + 1/4x 2 + 1/4x 1 = 2.5

How about a bigger cacheline?

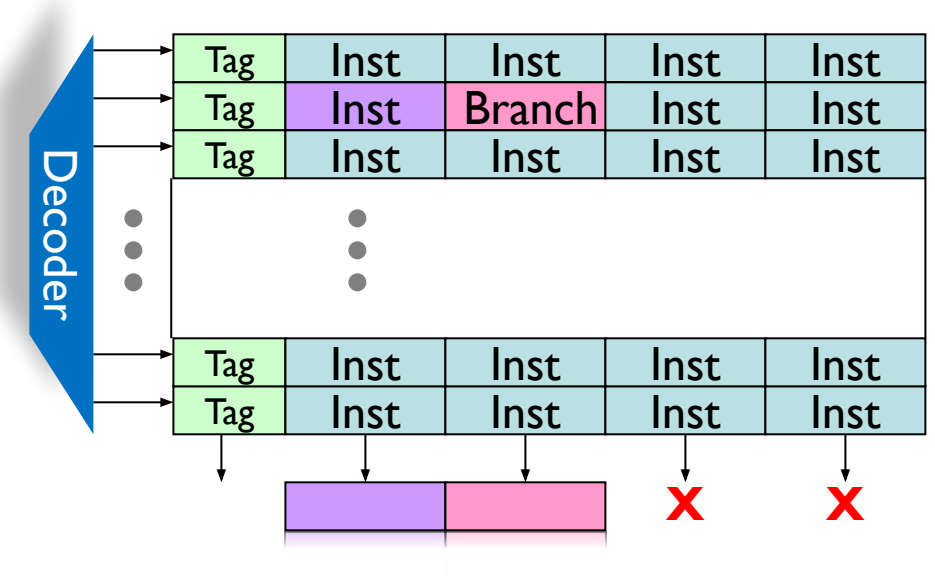
- Let N=4, cache line size = 8
- Then fetch rate = $^{5}/_{8}$ x 4
  + $^{1}/_{8}$ x 3
  + $^{1}/_{8}$ x 2
  + $^{1}/_{8}$ x 1

  = 3.25 instructions per cycle



Branch Problem

- Even if fetch group is aligned, and/or cache line size > than fetch group, taken branches disrupt fetch.



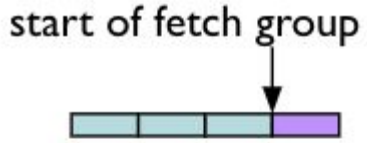
Fetch Rate with Branch

- Let N=4
- Branch every 5 instructions on average
- Assume branch always taken
- Assume branch target may start at any offset in a cache row

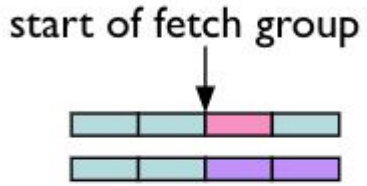
→ What would be the rate?

Fetch Rate with Branch

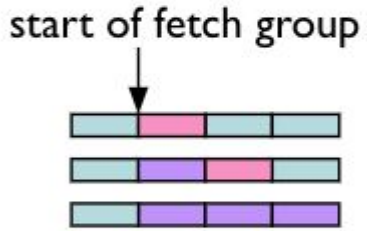
¼ x 1 instruction



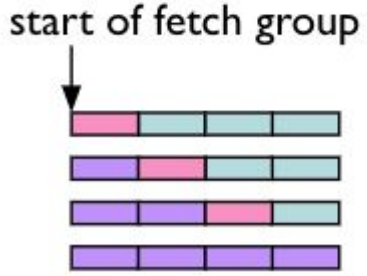
¼ x (0.2 x 1 + 0.8 x 2)



¼ x (0.2 x 1 + 0.8 x (0.2 x 2 + 0.8 x 3))



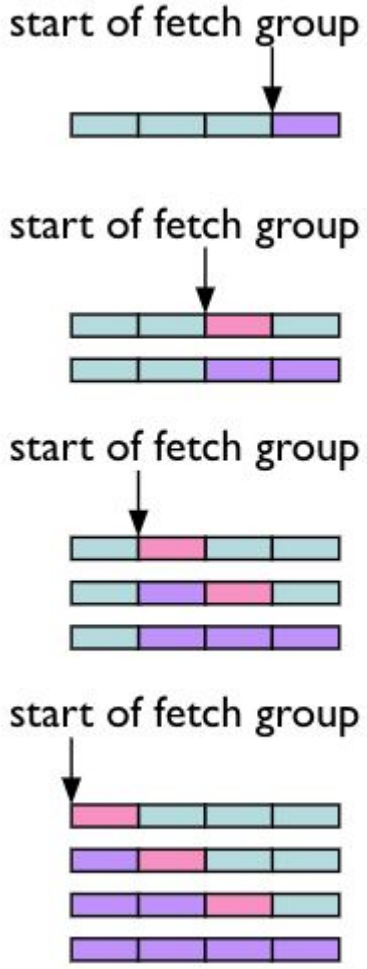
¼ x (0.2 x 1 + 0.8 x (0.2 x 2 + 0.8 x (0.2 x 3 + 0.8 x 4)))



Fetch Rate with Branch

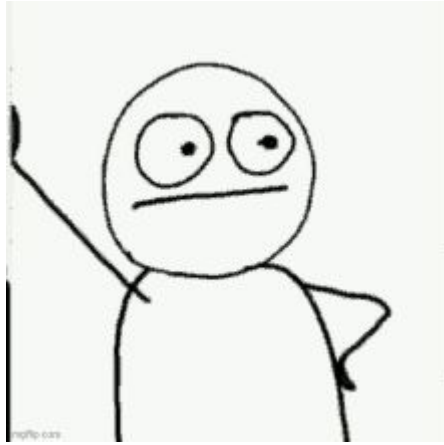
- Fetch rate = 2.048

→ One solution to reduce the overhead from compile time: basic block starts at the beginning of cache blocks



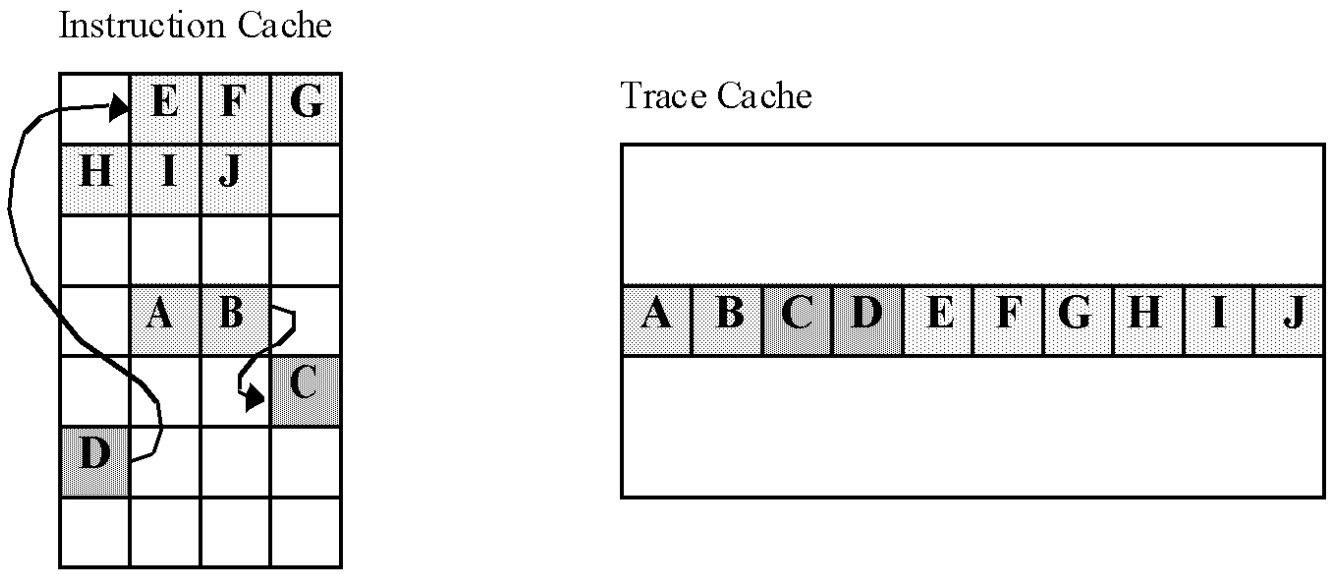
Can we do better?

- What else can we do instead of fetching multiple lines from the caches?



Trace Cache

- What if we store useful instructions together?
- Programs have strong temporal locality!



Eric Rotenberg, S. Bennett, and James E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. MICRO, December 1996.

Trace Cache

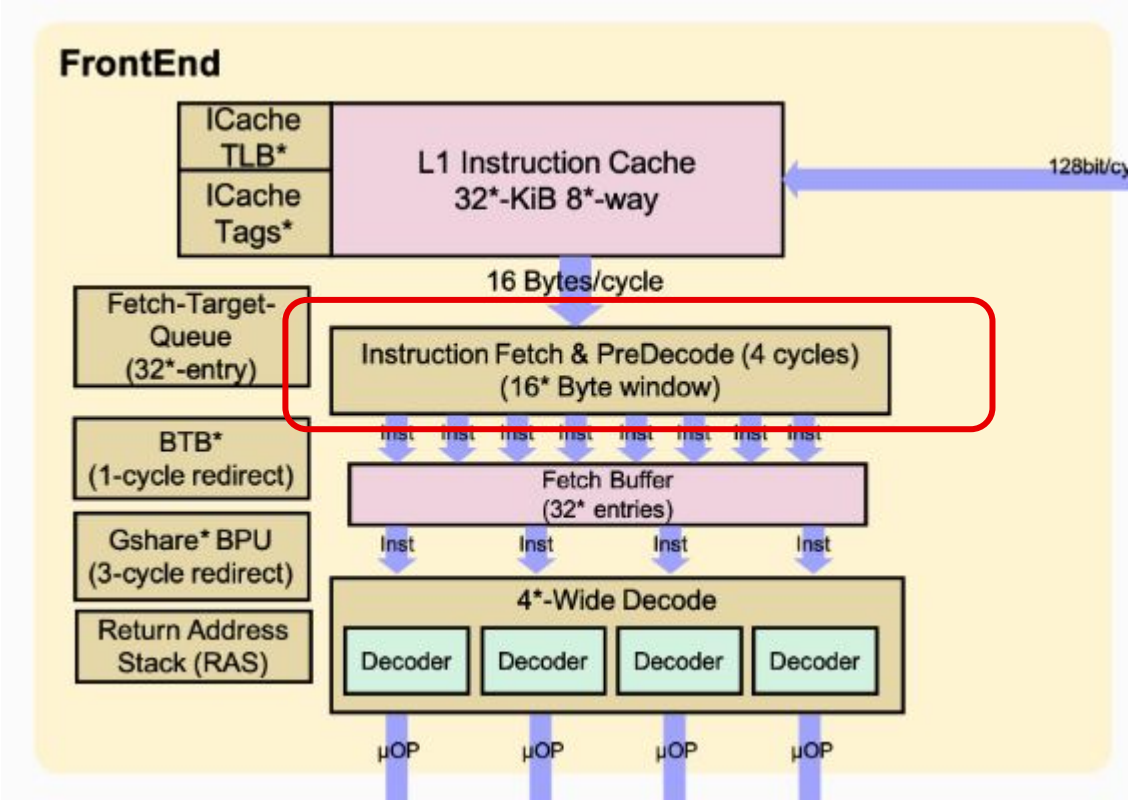
- Observe the sequence of instructions and create a mechanism to save that!

- How to update?

- How to select between regular cache and trace cache?

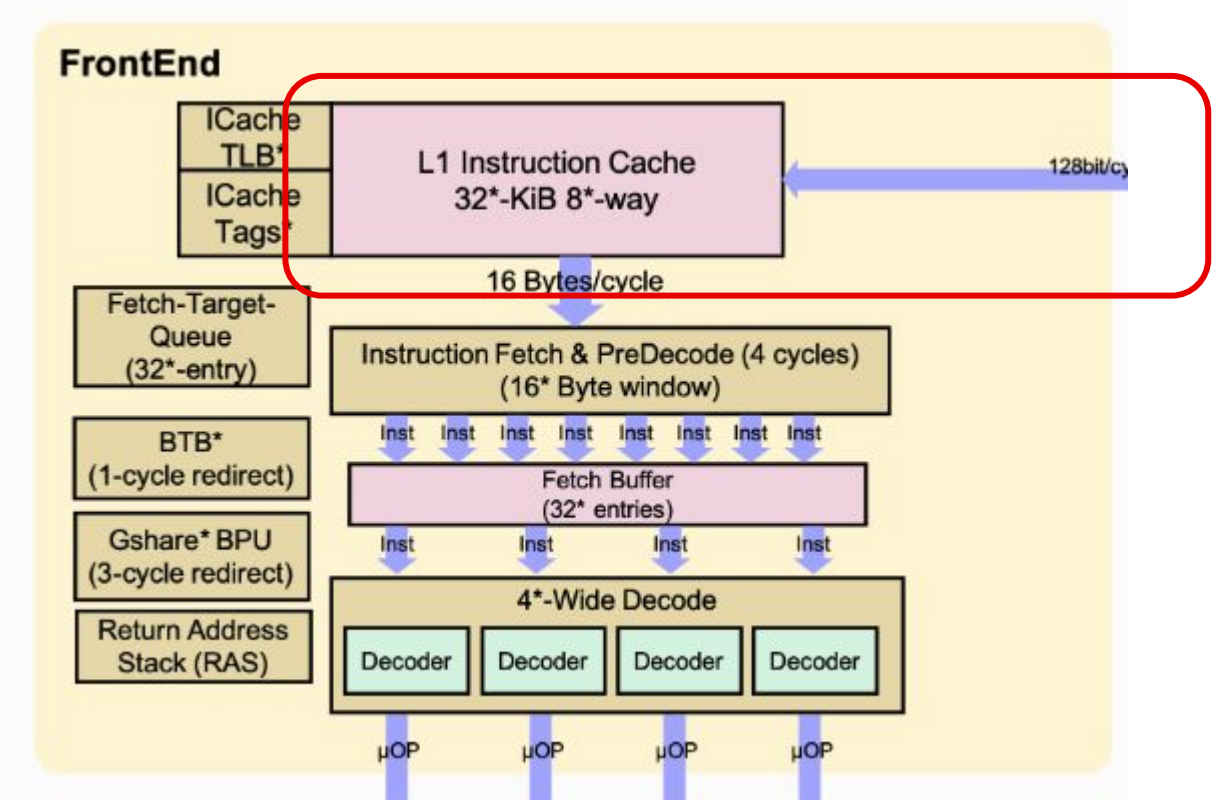
Frontend

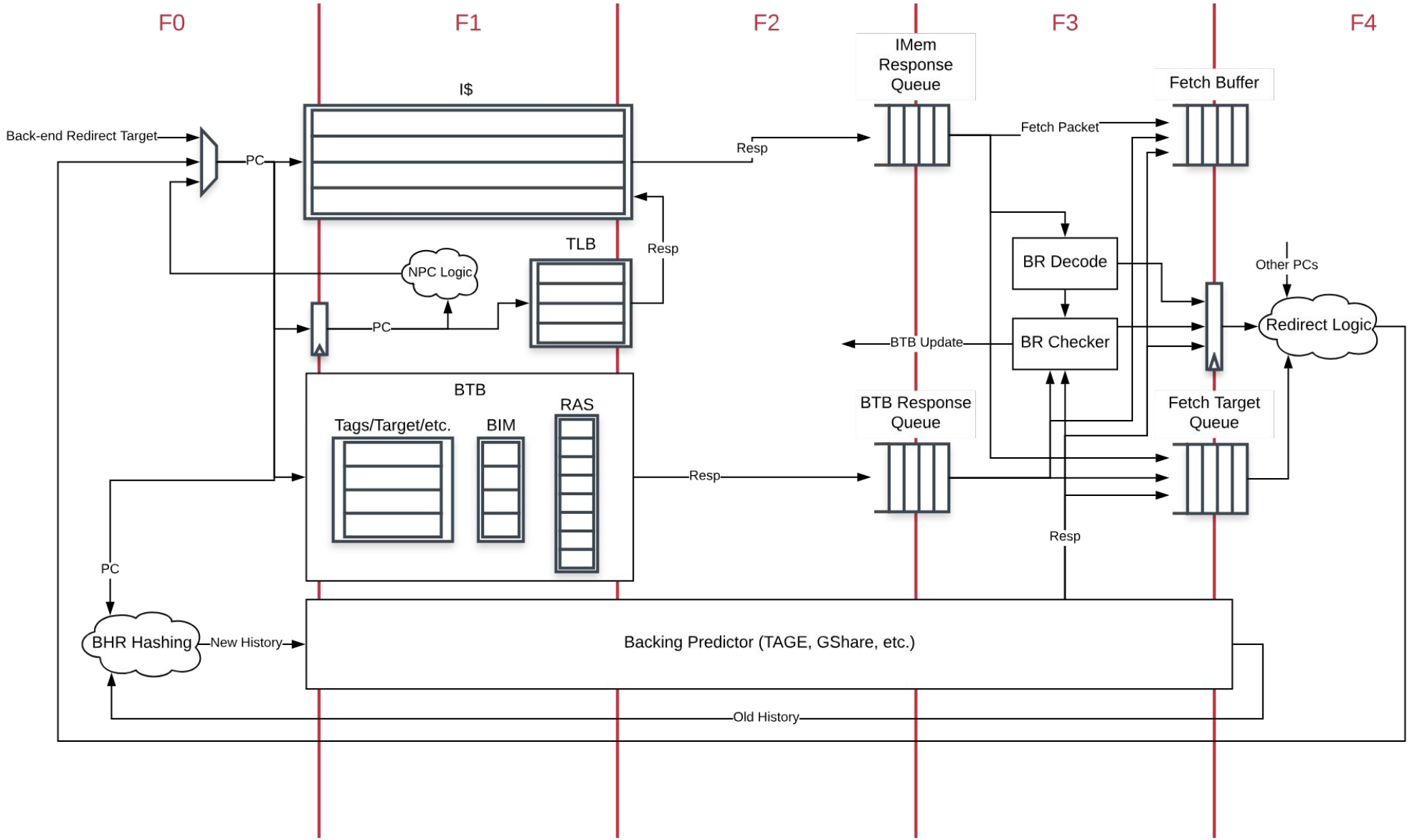
- Fetch
- Decome
- Rename



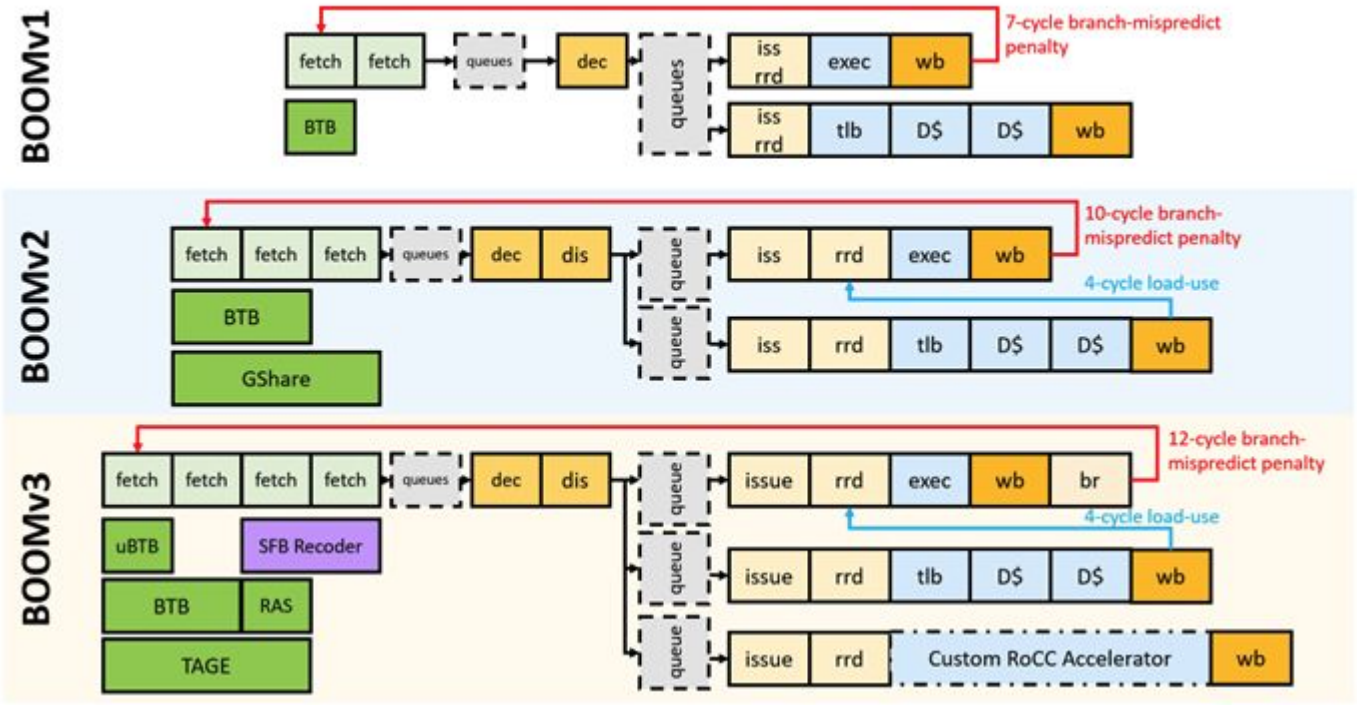
Frontend

- Fetch
- Decome
- Rename




F0
F1
F2
F3
F4
I$
IMem Response Queue
Fetch Buffer
Back-end Redirect Target
PC
Resp
Fetch Packet
TLB
NPC Logic
Resp
BR Decode
Other PCs
PC
BTB Update
BR Checker
Redirect Logic
BTB
RAS
Tags/Target/etc.
BIM
BTB Response Queue
Fetch Target Queue
Resp
Resp
PC
BHR Hashing
New History
Backing Predictor (TAGE, GShare, etc.)
Old History

Fetch in BOOM



Fetch in BOOM

- Four stages. The additional stages were mainly added to support more sophisticated branch prediction.

- Multiple branches can be predicted/resolved simultaneously.

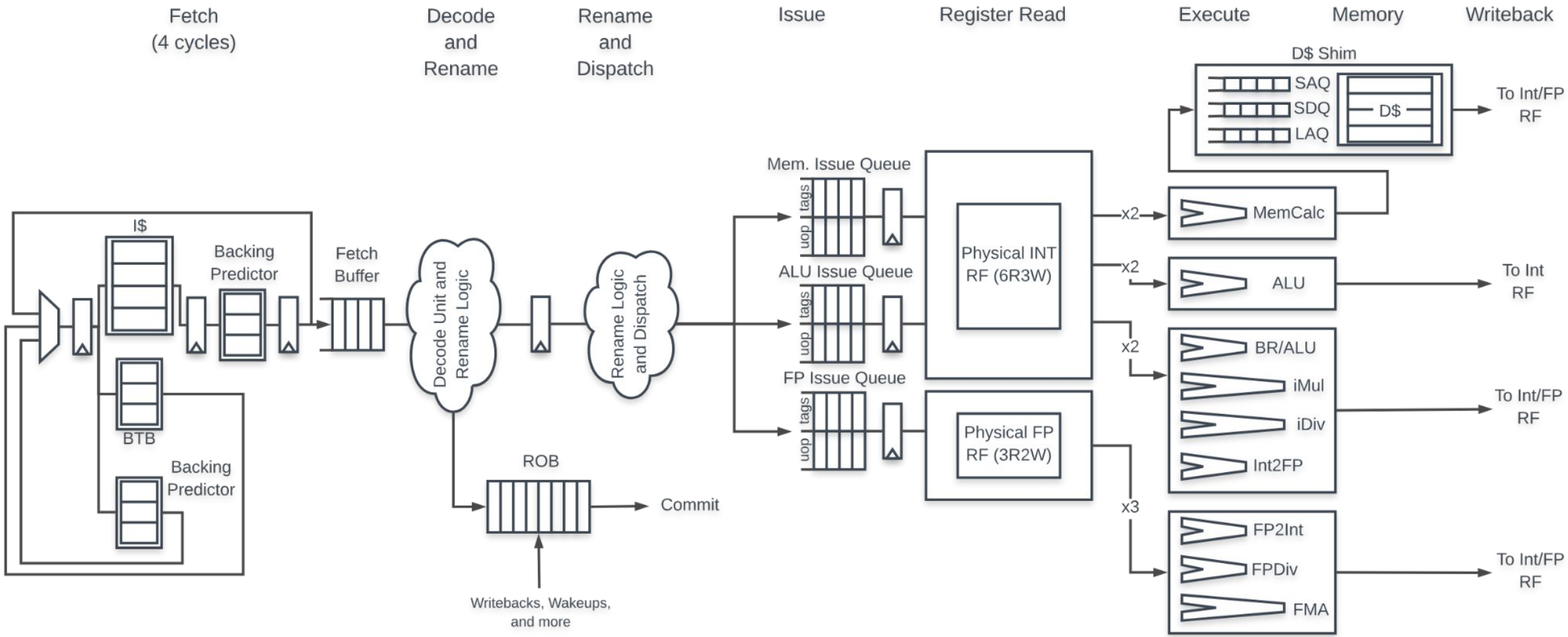
Branch Prediction

- We will discuss this extensively in the next lecture.

Fetch Details

- The Front-end retrieves a Fetch Packet of instructions from instruction memory and puts them into the Fetch Buffer.
- The Fetch Packet also contains other meta-data, such as a valid mask (valid instr) and branch prediction information.
- Additionally, the PC and branch prediction information is stored inside of the Fetch Target Queue

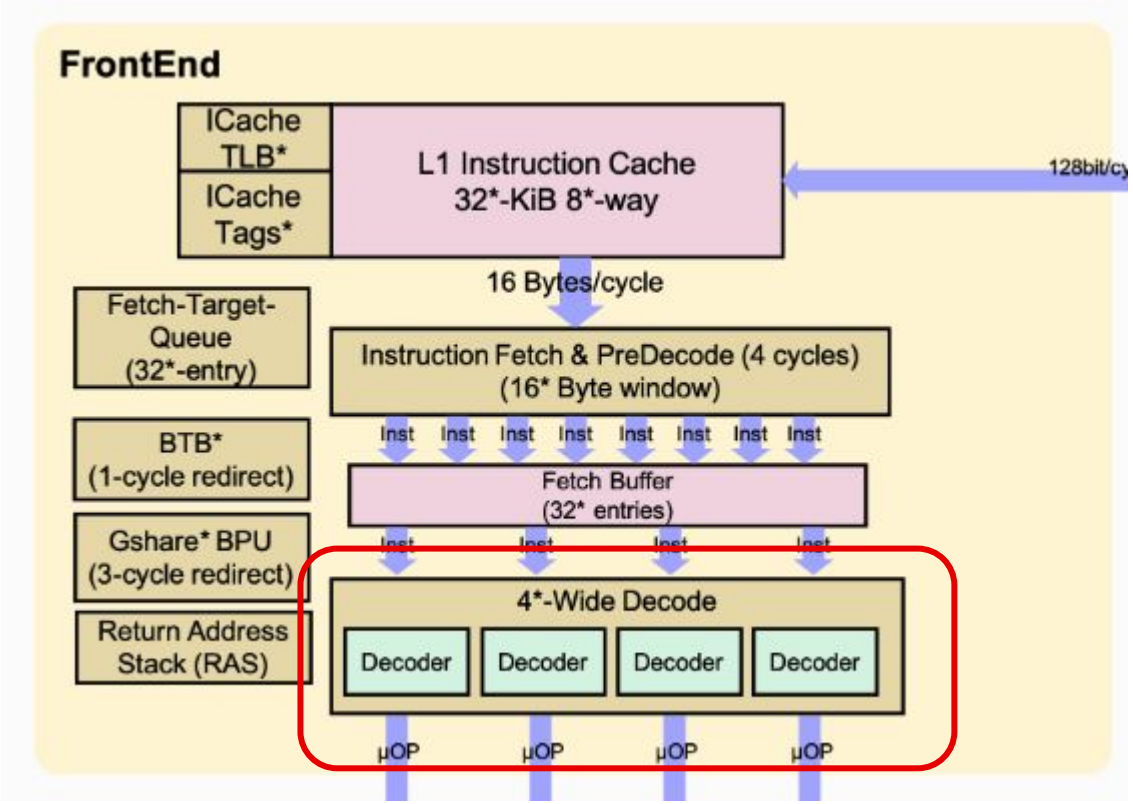
Simplified Pipeline



Decode

BOOM Decoder

- 4-wide: consuming from the fetch buffer.
- Try to answer these three things:
  - What operation
  - What resources
  - What operand



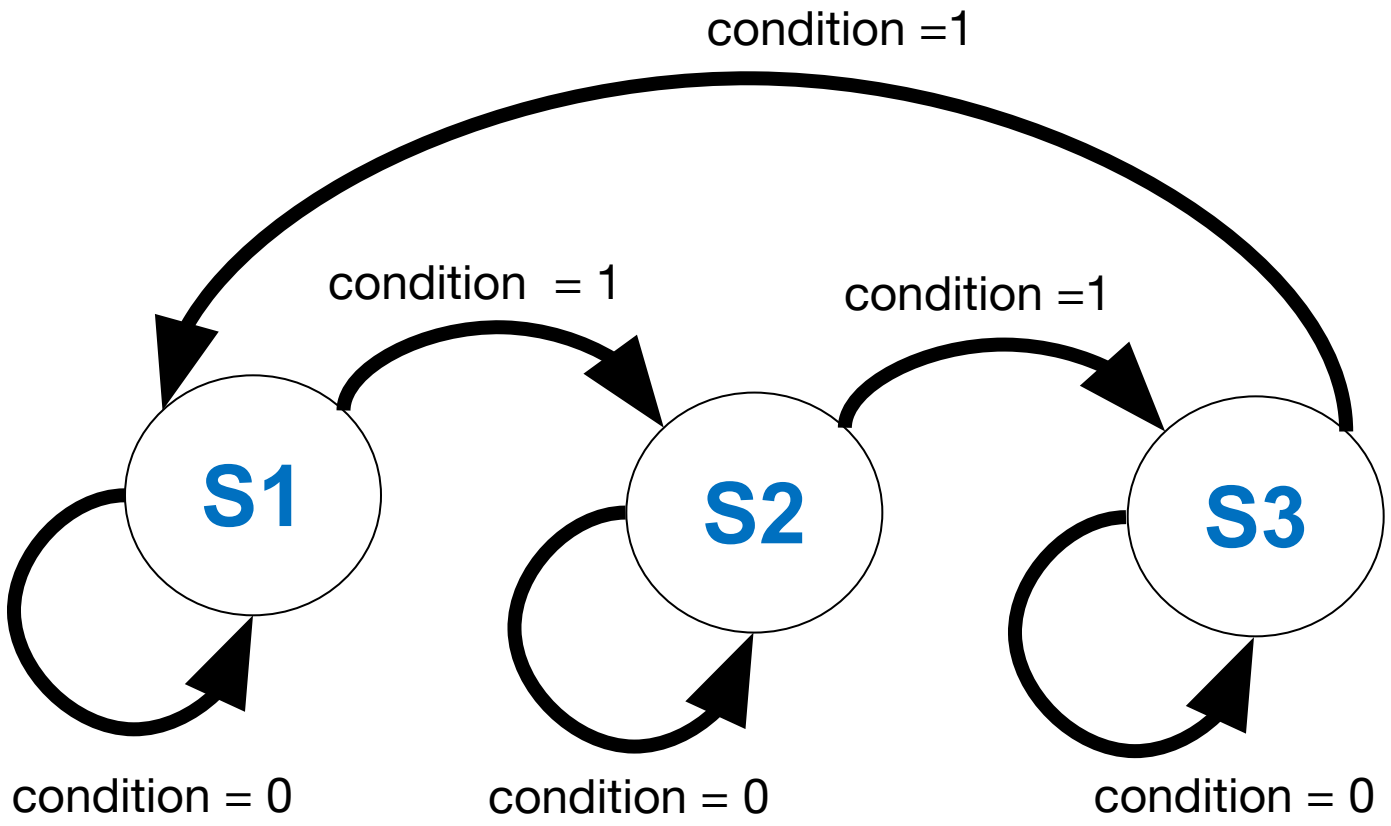
How to build a controller?

– *Use finite state machines!*

Fine-State Machine (FSM)

- A *mathematical* model of computation.
- At each point, the system *can be only at one* of the several, but ***finite***, states.
- FSM shows all the states and how they transit to each other.
- FSM also includes finite number of inputs and outputs.

State Machine Diagram



FSM/Controller for a Processor

- A giant FSM with many states!

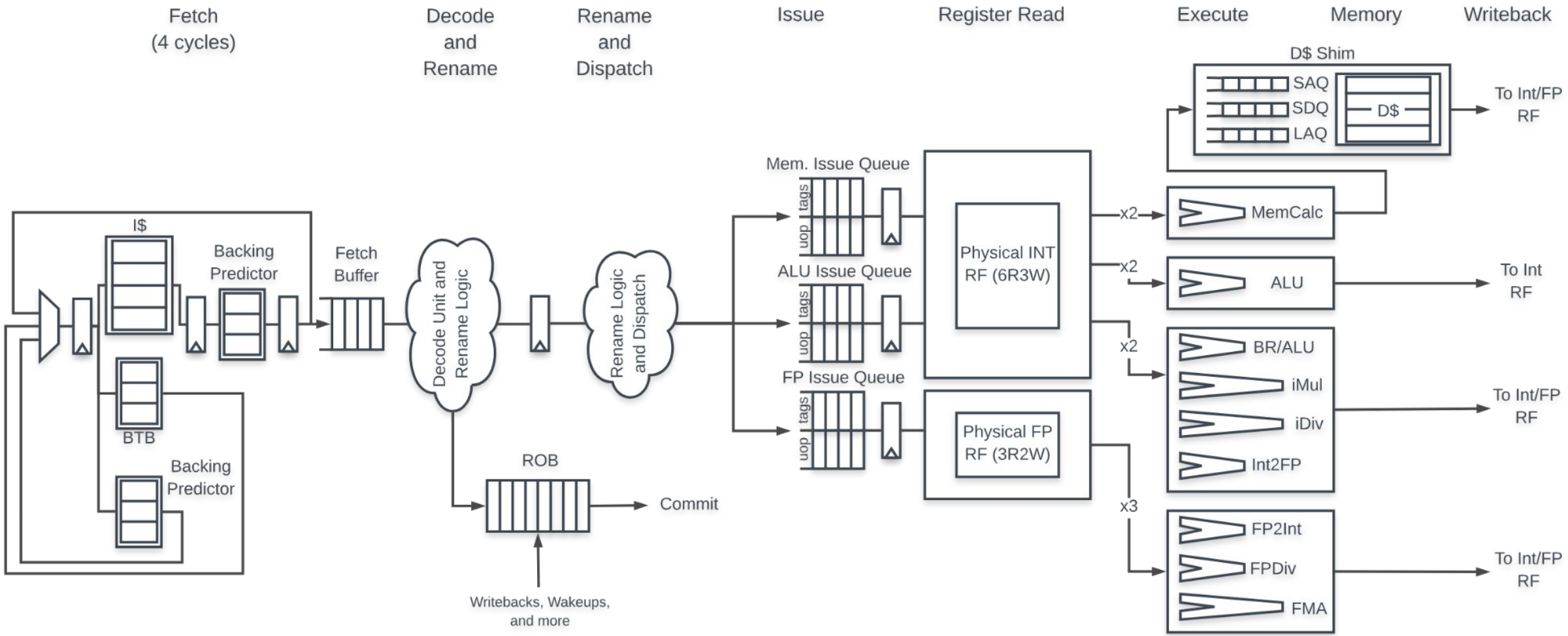
```
11
12  package boom.v3.common
13
14  import chisel3._
15  import chisel3.util._
16
17  import org.chipsalliance.cde.config.Parameters
18
19  import boom.v3.exu.FUConstants
20
21  /**
22   * Extension to BoomBundle to add a MicroOp
23   */
24  abstract trait HasBoomUOP extends BoomBundle
25  {
26    val uop = new MicroOp()
27  }
28
29  /**
30   * MicroOp passing through the pipeline
31   */
32  class MicroOp(implicit p: Parameters) extends BoomBundle
33    with freechips.rocketchip.rocket.constants.MemoryOpConstants
34    with freechips.rocketchip.rocket.constants.ScalarOpConstants
35  {
36    val uopc             = UInt(UOPC_SZ.W)       // micro-op code
37    val inst             = UInt(32.W)
38    val debug_inst       = UInt(32.W)
39    val is_rvc           = Bool()
40    val debug_pc         = UInt(coreMaxAddrBits.W)
41    val iq_type          = UInt(IQT_SZ.W)        // which issue unit do we use?
42    val fu_code          = UInt(FUConstants.FUC_SZ.W) // which functional unit do we use?
43    val ctrl             = new CtrlSignals
44
45    // What is the next state of this uop in the issue window? useful
46    // for the compacting queue.
47    val iw_state         = UInt(2.W)
48    // Has operand 1 or 2 been waken speculatively by a load?
49    // Only integer operands are speculaively woken up,
50    // so we can ignore p3.
51    val iw_p1_poisoned   = Bool()
52    val iw_p2_poisoned   = Bool()
53
54    val is_br            = Bool()                    // is this micro-op a (branch) vs a regular PC+4 inst?
55    val is_jalr          = Bool()                    // is this a jump? (jal or jalr)
56    val is_jal           = Bool()                    // is this a JAL (doesn't include JR)? used for branch unit
57    val is_sfb           = Bool()                    // is this a sfb or in the shadow of a sfb
58
59    val br_mask          = UInt(maxBrCount.W)  // which branches are we being speculated under?
60    val br_tag           = UInt(brTagSz.W)
61
62    // Index into FTQ to figure out our fetch PC.
63    val ftq_idx          = UInt(log2Ceil(ftqSz).W)
64    // This inst straddles two fetch packets
65    val edge_inst        = Bool()
```

Micro-op vs. Instruction?

- The basic operations are called micro-op.
- An instruction might be multiple micro-op.
- For RISC architectures: micro-op and instructions are mostly the same.
- Compress and unaligned instructions.

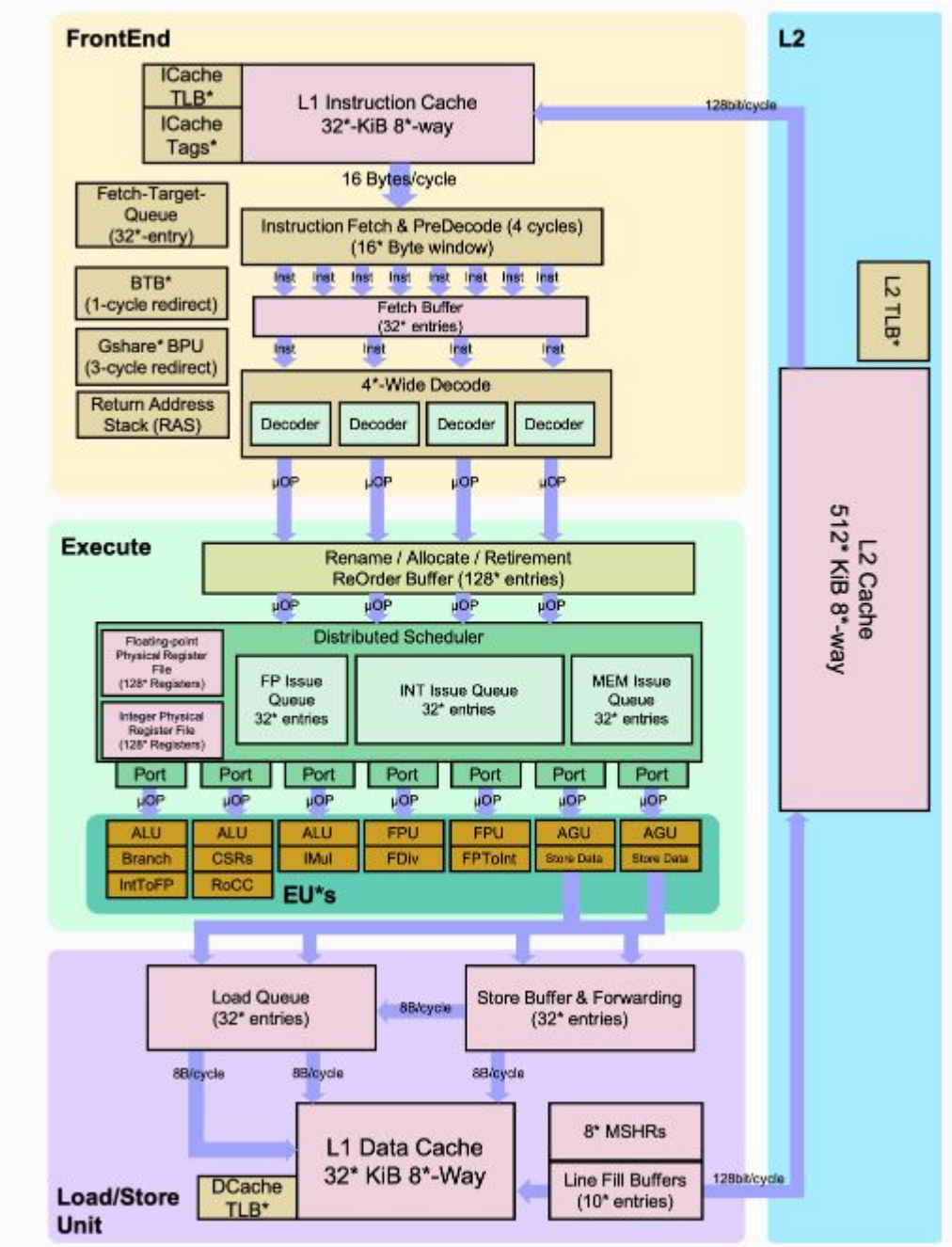
Rename

Simplified Pipeline

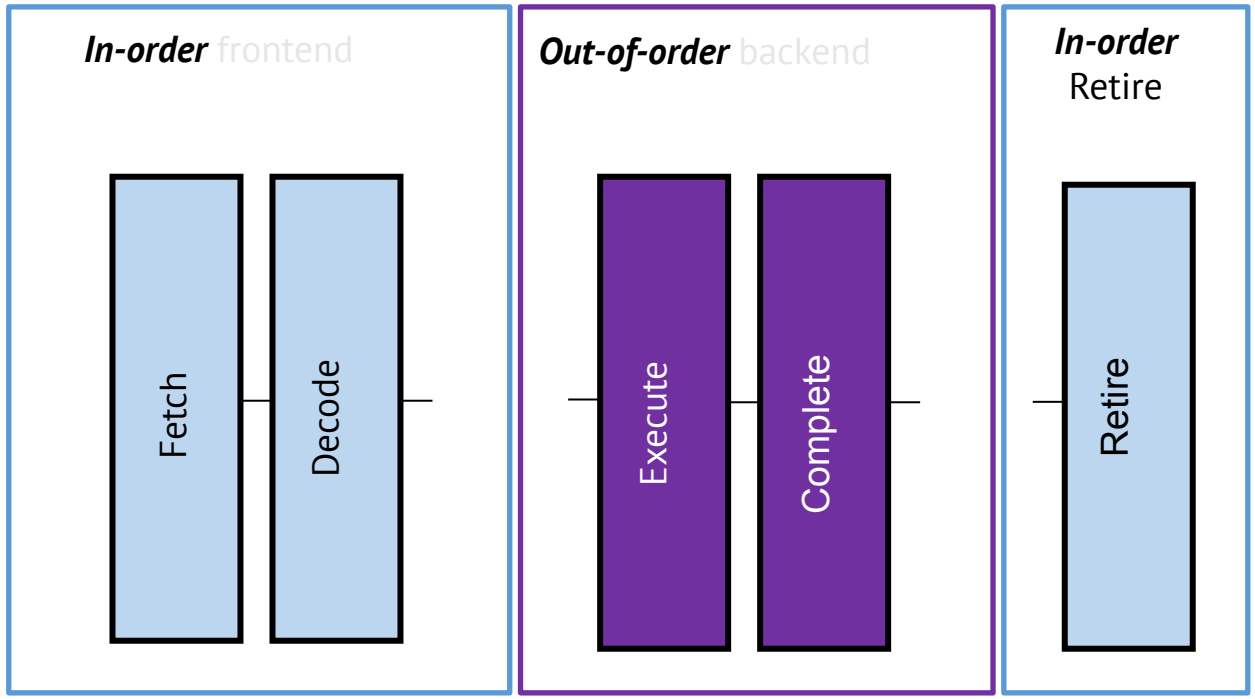


In-Order vs. OoO

- We fetch and decode in-order but execute things out-of-order!
- Why? Example:

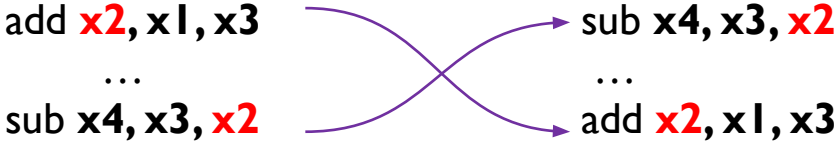


Out-of-Order Design



What would happen if we start executing out-of-order?

- ***Example:***
  - **RAW** (Read After Write) = "true dependence"



What would happen if we start executing out-of-order?

- ***Example:***
  - **RAW** (Read After Write) = "true dependence" -- ***true***

```
add x2, x1, x3
…
sub x4, x3, x2
```

What would happen if we start executing out-of-order?

- ***Example:***
  - **RAW** (Read After Write) = "true dependence" -- ***true***

    add **x2, x1, x3**

    …

    sub **x4, x3, x2**
  - **WAW** (Write After Write) = "output dependence"

    add **x2, x1, x3**

    …

    sub **x2, x3, x4**

What would happen if we start executing out-of-order?

- ***Example:***
  - **RAW** (Read After Write) = "true dependence" -- ***true***

    add **x2, x1, x3**

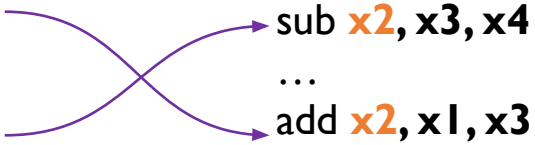
    …

    sub **x4, x3, x2**
  - **WAW** (Write After Write) = "output dependence"

    add **x2, x1, x3**

    …

    sub **x2, x3, x4**

    sub **x2, x3, x4**

    …

    add **x2, x1, x3**

What would happen if we start executing out-of-order?

- ***Example:***
  - **RAW** (Read After Write) = "true dependence" -- ***true***

```
add x2, x1, x3
…
sub x4, x3, x2
```

  - **WAW** (Write After Write) = "output dependence"

```
add x2, x1, x3          sub x2, x3, x4
      …                       …
sub x2, x3, x4          add x2, x1, x3
```

  - **WAR** (Write After Read) = "anti-dependence"

```
add x3, x1, x2
…
sub x2, x3, x4
```

What would happen if we start executing out-of-order?

- ***Example:***
  - **RAW** (Read After Write) = "true dependence" -- ***true***

```
add x2, x1, x3
…
sub x4, x3, x2
```

  - **WAW** (Write After Write) = "output dependence"

```
add x2, x1, x3          sub x2, x3, x4
…                         …
sub x2, x3, x4          add x2, x1, x3
```

  - **WAR** (Write After Read) = "anti-dependence"

```
add x3, x1, x2          sub x2, x3, x4
…                         …
sub x2, x3, x4          add x3, x1, x2
```

False vs. True Dependency

- ***Example:***
  - **RAW** (Read After Write) = "true dependence"

    add **x2, x1, x3**
    …
    sub **x4, x3, x2**

  - **WAW** (Write After Write) = "output dependence"

    add **x2, x1, x3**
    …
    sub **x2, x3, x4**

    sub **~~x2~~ x5, x3, x4**
    …
    add **x2, x1, x3**

  - **WAR** (Write After Read) = "anti-dependence"

    add **x3, x1, x2**
    …
    sub **x2, x3, x4**

    sub **~~x2~~ x5, x3, x4**
    …
    add **x3, x1, x2**

Why false dependencies?

- Finite number of registers
  - At some point, you're forced to overwrite somewhere
  - Most RISC: 32 registers, x86: only 8, x86-64: 16
- Loops, Code Reuse
  - If you write a value to R1 in a loop body, then R1 will be reused every iteration → induces many false dep's
  - Loop unrolling can help a little
    - Will run out of registers at some point anyway
    - Trade off with code bloat
  - Short function calls can result in similar register reuse
    - Inlining can help a little

False vs. True Dependency

- We need a mechanism for fixing existing RAW data hazards.
  - We will use forwarding as we will show later.

- We need a new mechanism to avoid false dependencies
  - *Renaming can fix this!*

Register Renaming

- *Limited* architectural registers (ISA registers, e.g., 32 in RISC-V)
- *Much more* physical registers (e.g., 128 registers)
- One architectural register (A-reg) can be assigned to ***multiple*** physical registers (P-reg).

Renaming Strategy Components

1. Mapping mechanism

2. Physical registers
   a. Allocated vs. free registers
   b. Allocation/deallocation mechanism

3. State maintenance (commit, mispredictions, etc.)

How to do renaming?

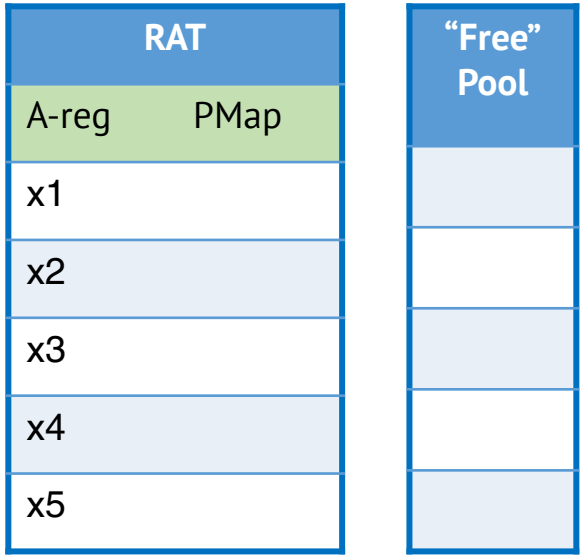
- *Register Alias/Map Table (RAT)*
  - *One* entry per architectural register.
  - Each entry stores the *physical* location of *the most recent* version of the architectural (logical) register.

★ ***Algorithm:***

- For each *destination* A-reg, the renaming algorithm assigns a new P-reg from a *pool of free (physical) registers.*
- For each *source* A-reg, the renaming algorithm accesses RAT and finds the corresponding P-reg.

Renaming Example

- Let's assume:
  - We have 5 architectural registers.
  - 10 physical registers.



Renaming Example

- Let's assume:
  - We have 5 architectural registers.
  - 10 physical registers.
  - The *initial* mapping is like this

RAT	
A-reg	PMap
x1	p1
x2	p2
x3	p3
x4	p4
x5	p5

"Free" Pool
p6
p7
p8
p9
p10

Renaming Example

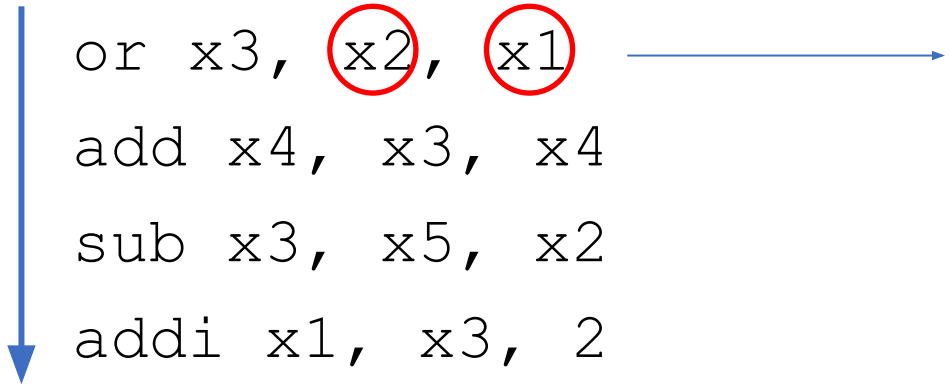
```
or x3, x2, x1
add x4, x3, x4
sub x3, x5, x2
addi x1, x3, 2
```

RAT	
A-reg	PMap
x1	p1
x2	p2
x3	p3
x4	p4
x5	p5

"Free" Pool
p6
p7
p8
p9
p10

Renaming Example

```
or x3, x2, x1
add x4, x3, x4
sub x3, x5, x2
addi x1, x3, 2
```



RAT	
A-reg	PMap
x1	p1
x2	p2
x3	p3
x4	p4
x5	p5

"Free" Pool
p6
p7
p8
p9
p10

Renaming Example

```
or x3, x2, x1      →   or ?, p2, p1
add x4, x3, x4
sub x3, x5, x2
addi x1, x3, 2
```

RAT	
A-reg	PMap
x1	p1
x2	p2
x3	p3
x4	p4
x5	p5

"Free" Pool
p6
p7
p8
p9
p10

Renaming Example

```
or x3, x2, x1
add x4, x3, x4
sub x3, x5, x2
addi x1, x3, 2
```

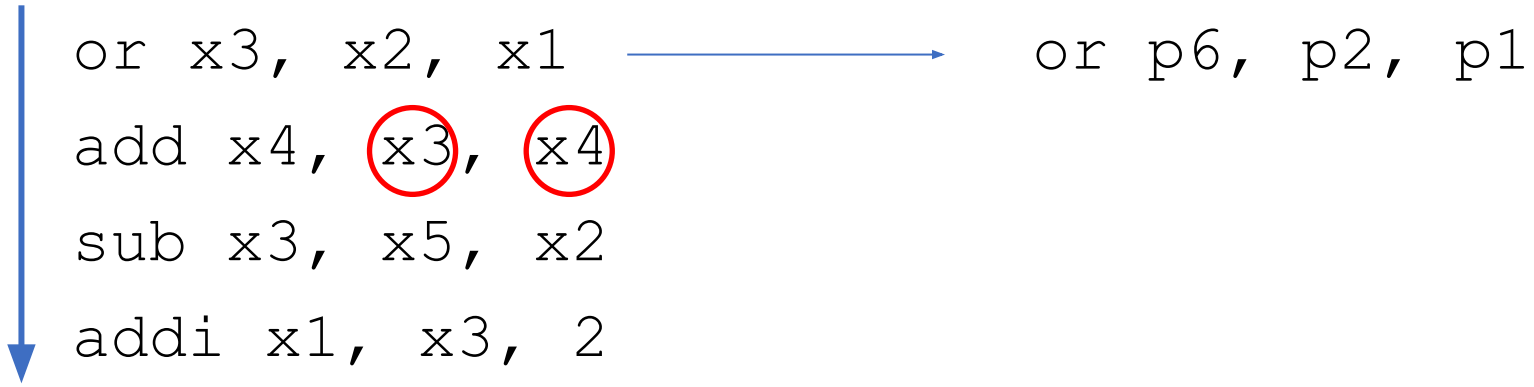
→

```
or p6, p2, p1
```

RAT	
A-reg	PMap
x1	p1
x2	p2
x3	~~p3~~ p6
x4	p4
x5	p5

"Free" Pool
~~p6~~
p7
p8
p9
p10

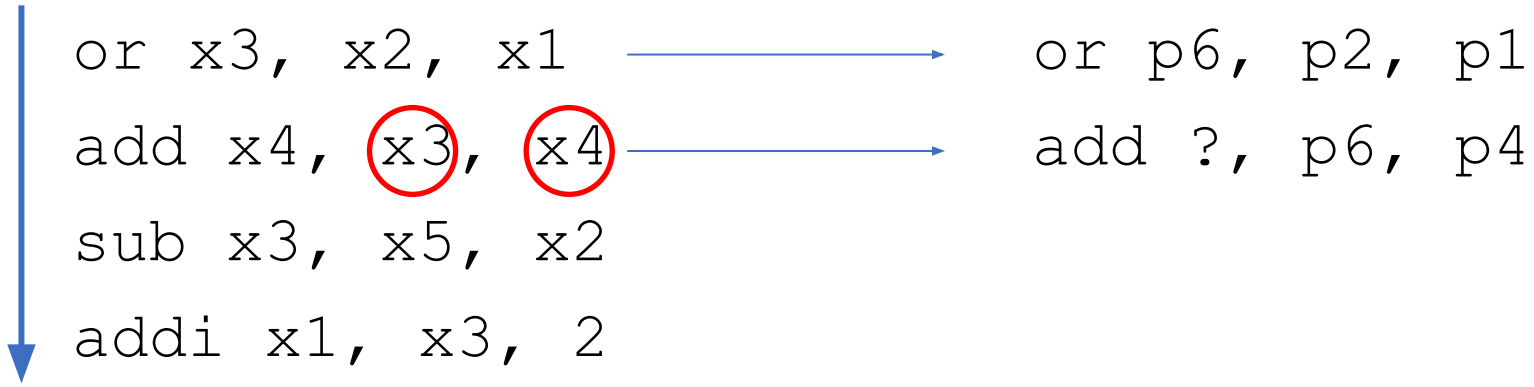
Renaming Example



RAT	
A-reg	PMap
x1	p1
x2	p2
x3	p6
x4	p4
x5	p5

"Free" Pool
p7
p8
p9
p10

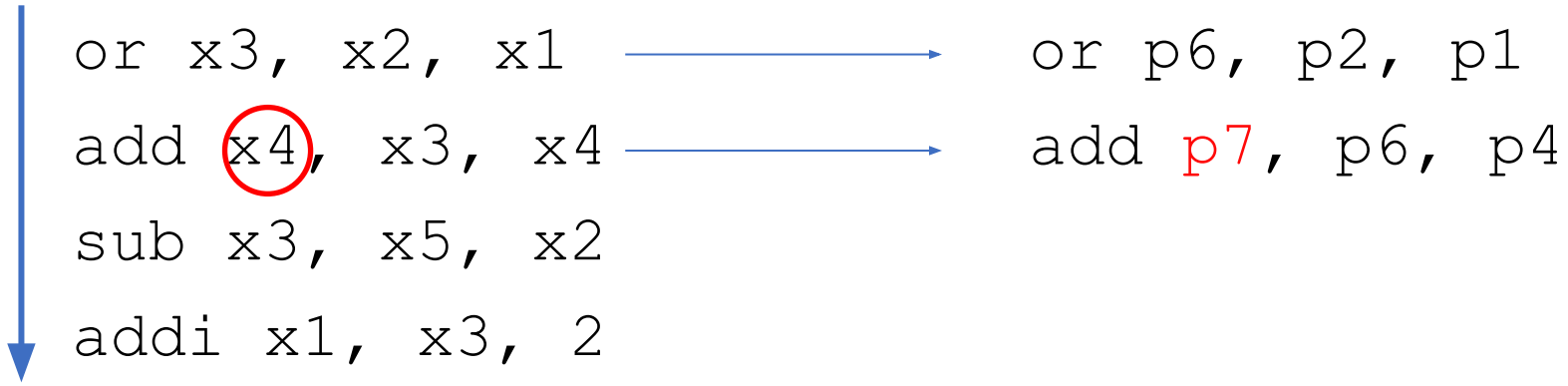
Renaming Example



RAT	
A-reg	PMap
x1	p1
x2	p2
x3	p6
x4	p4
x5	p5

"Free" Pool
p7
p8
p9
p10

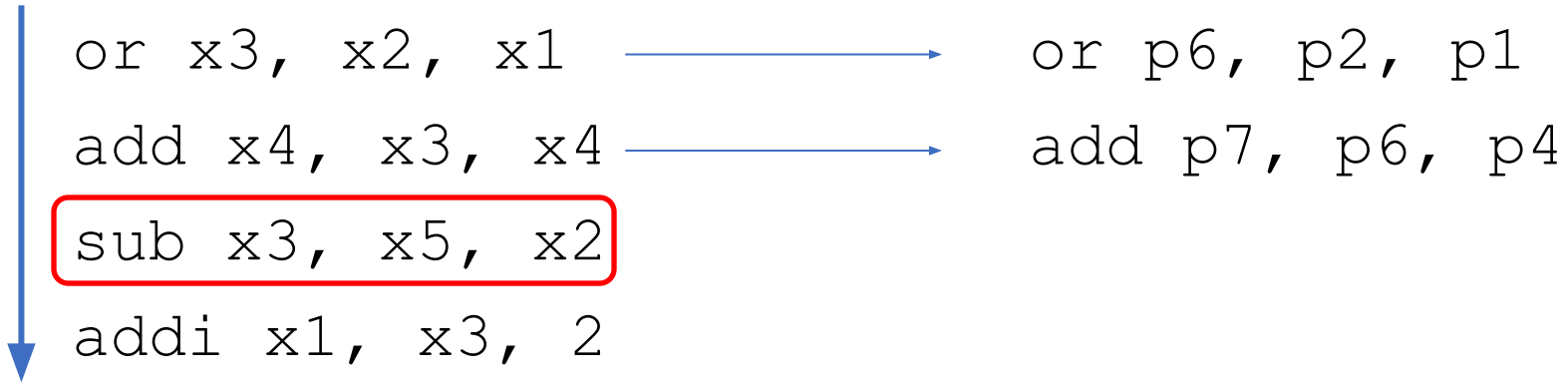
Renaming Example



RAT	
A-reg	PMap
x1	p1
x2	p2
x3	p6
x4	~~p4~~ p7
x5	p5

"Free" Pool
~~p7~~
p8
p9
p10

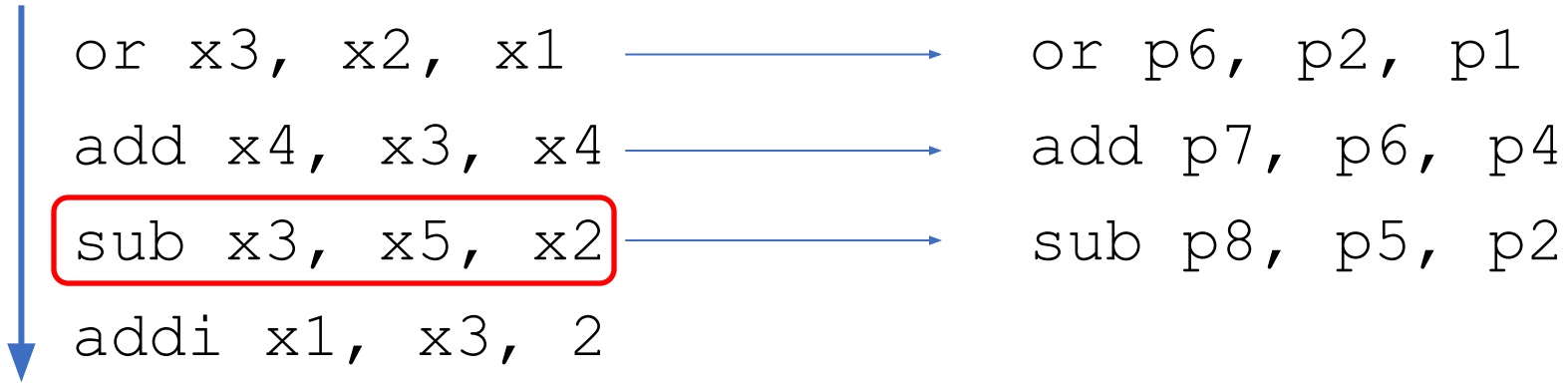
Renaming Example



RAT	
A-reg	PMap
x1	p1
x2	p2
x3	p6
x4	p7
x5	p5

"Free" Pool
p8
p9
p10

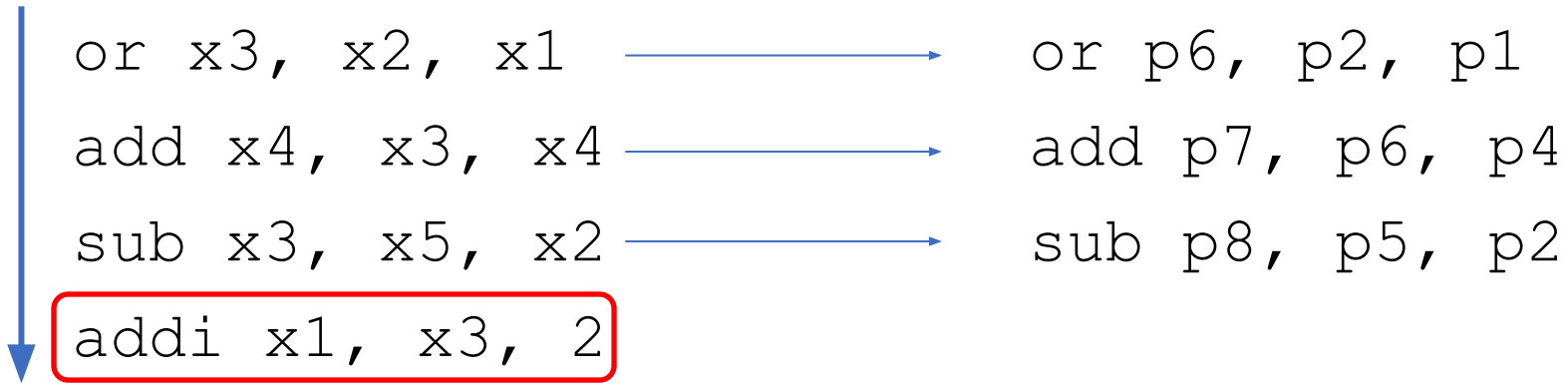
Renaming Example



RAT	
A-reg	PMap
x1	p1
x2	p2
x3	~~p6~~ p8
x4	p7
x5	p5

"Free" Pool
p8
p9
p10

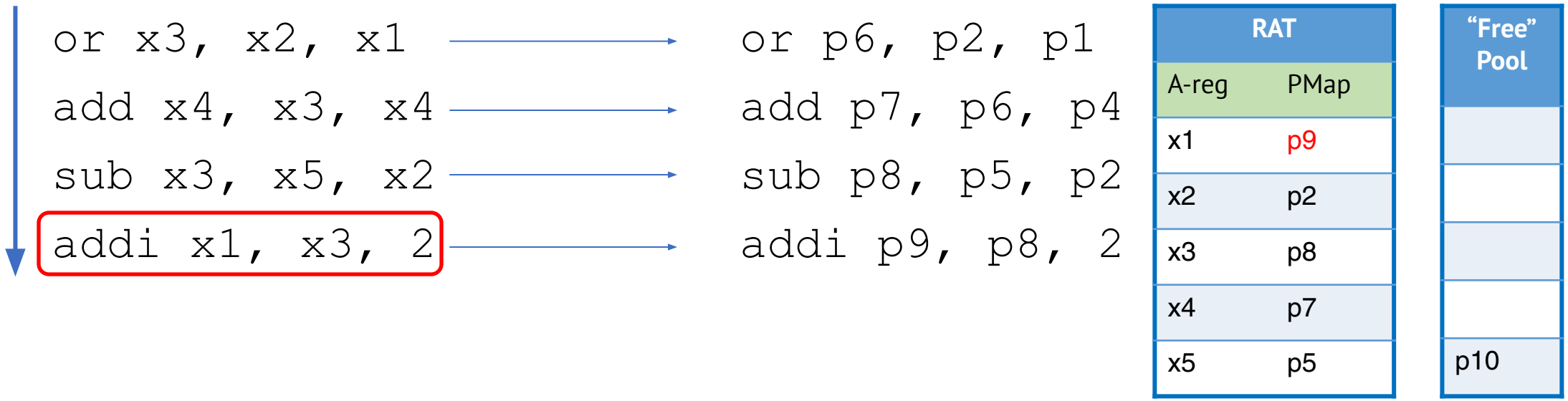
Renaming Example



RAT	
A-reg	PMap
x1	p1
x2	p2
x3	p8
x4	p7
x5	p5

"Free" Pool
p9
p10

Renaming Example

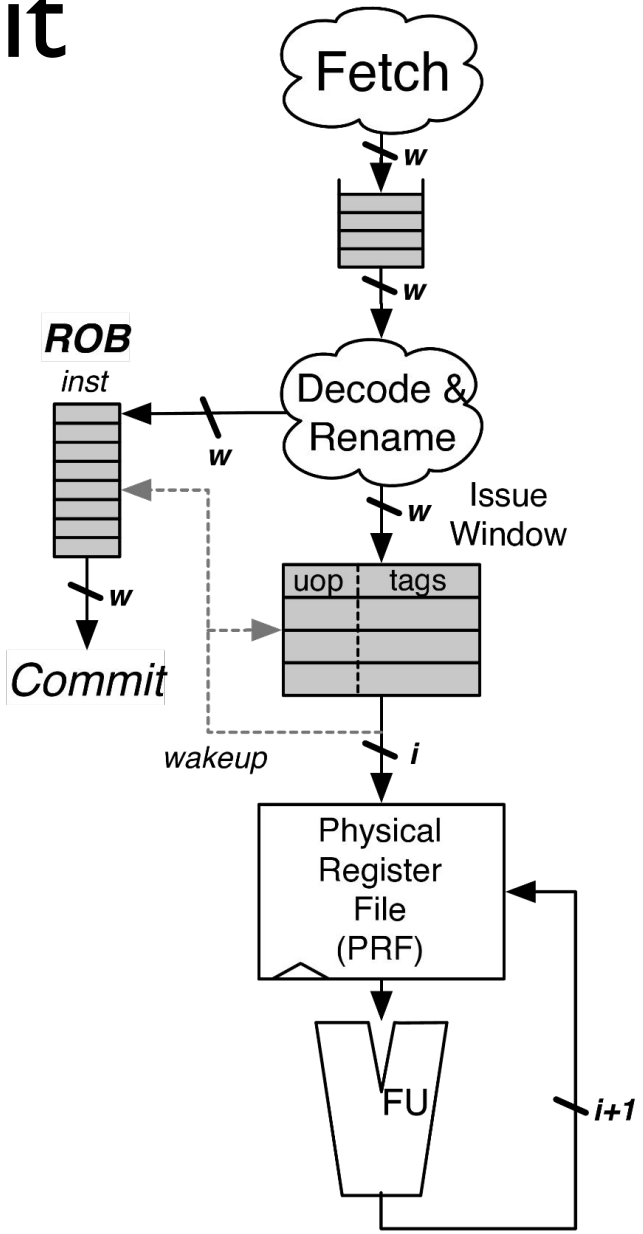


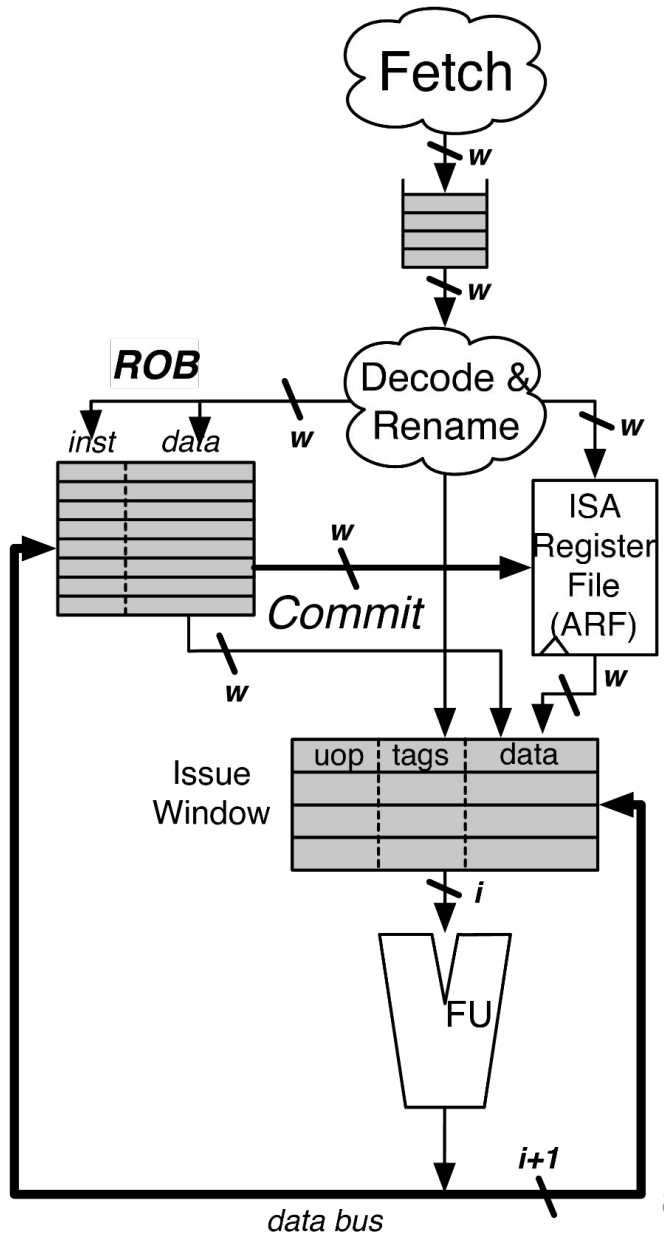
Main Renaming Methods

- "Explicit" or physical register file design (what BOOM uses)
- "Implicit" or data-in-ROB design

Explicit vs. Implicit

- Tradeoffs

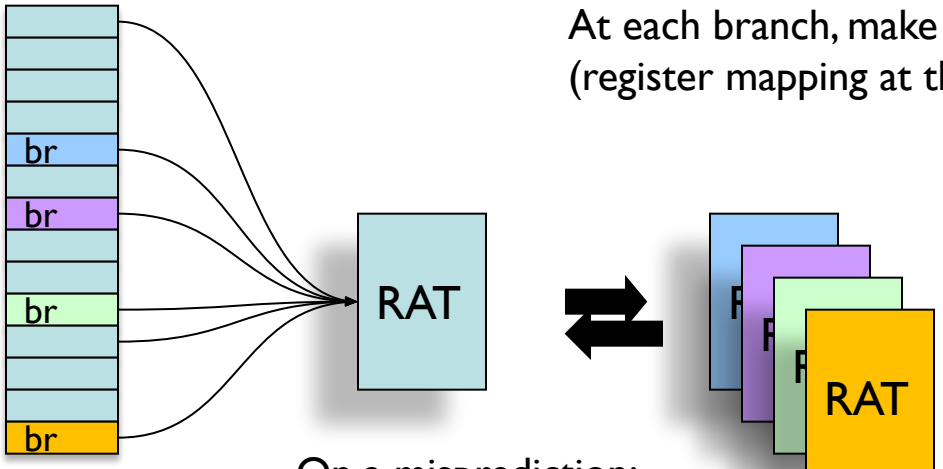




Explicit

- Have one unified register file called PRF.
- Have a Map table to keep track of instructions and their operands and their mapping.
- For each branch, create a new "copy" of the RAT (checkpointing).
- On mispredcition/exception/recovery, check the corresponding table and revert changes.

Recovery



At each branch, make a copy of the RAT
(register mapping at the time of the branch)

Checkpoint
Free Pool

On a misprediction:

1. flush wrong-path instructions
2. deallocate RAT checkpoints
3. recover RAT from checkpoint
4. resume renaming

Challenges

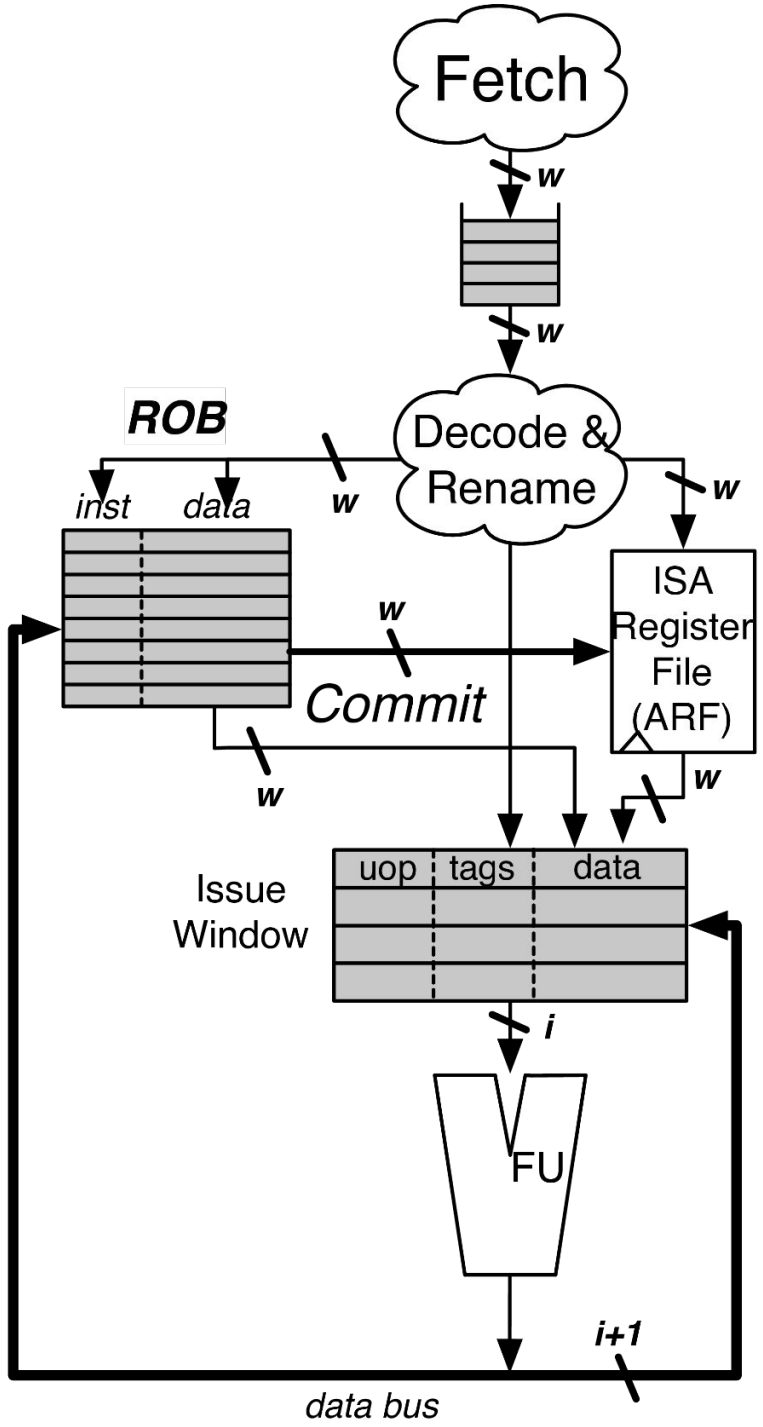
- Need one checkpoint per branch.
- What if the code has nothing but branches?
  - Worst case needs one checkpoint per ROB entry.
  - Can assign one checkpoint per branch color.
  - Stall front-end when out of branch colors/checkpoints.

➢ Possible solutions: Create checkpoint only for low-confident branches

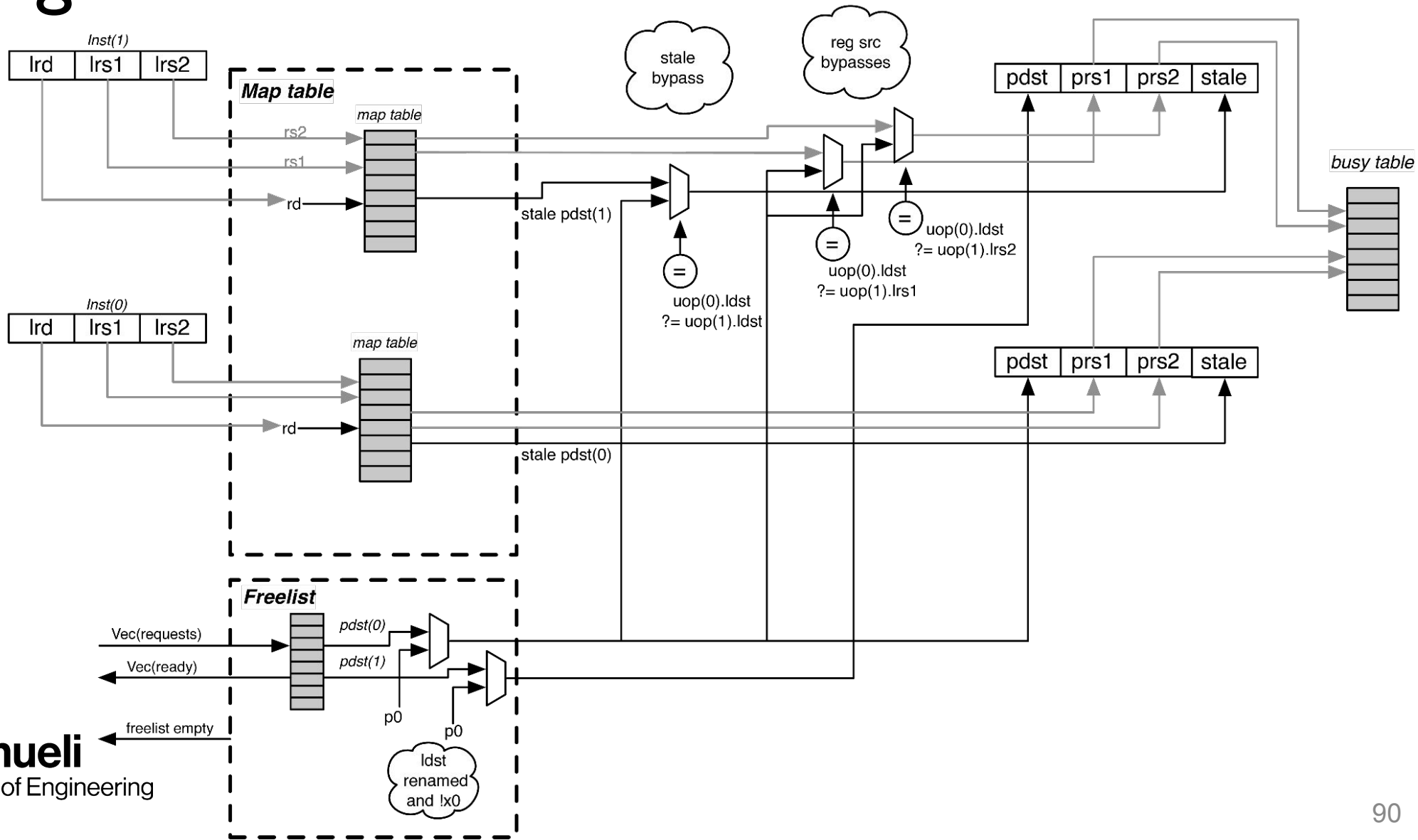
Implicit

- Keep track of data in ROB/Buffer. Pass data along with the instruction.
- Need only one RAT.

➢ Pros and cons?



Renaming in BOOM



UCLA Samueli School of Engineering

Renaming in BOOM

- The Busy Table tracks the readiness status of each physical register. If all physical operands are ready, the instruction will be ready to be issued. (Will talk about this later.)

Renaming in BOOM

- The Free List tracks the physical registers that are currently un-used and is used to allocate new physical registers to instructions passing through the Rename stage.

- The Free List is implemented as a bit-vector. A priority decoder can then be used to find the first free register. BOOM uses a cascading priority decoder to allocate multiple registers per cycle.

- On every branch, the Rename Map Tables are snapshotted to allow single-cycle recovery on a branch misprediction. Likewise, the Free List also sets aside a new "Allocation List", initialized to zero. As new physical registers are allocated, the Allocation List for each branch is updated to track all of the physical registers that have been allocated after the branch. If a misspeculation occurs, its Allocation List is added back to the Free List by OR'ing the branch's Allocation List with the Free List.

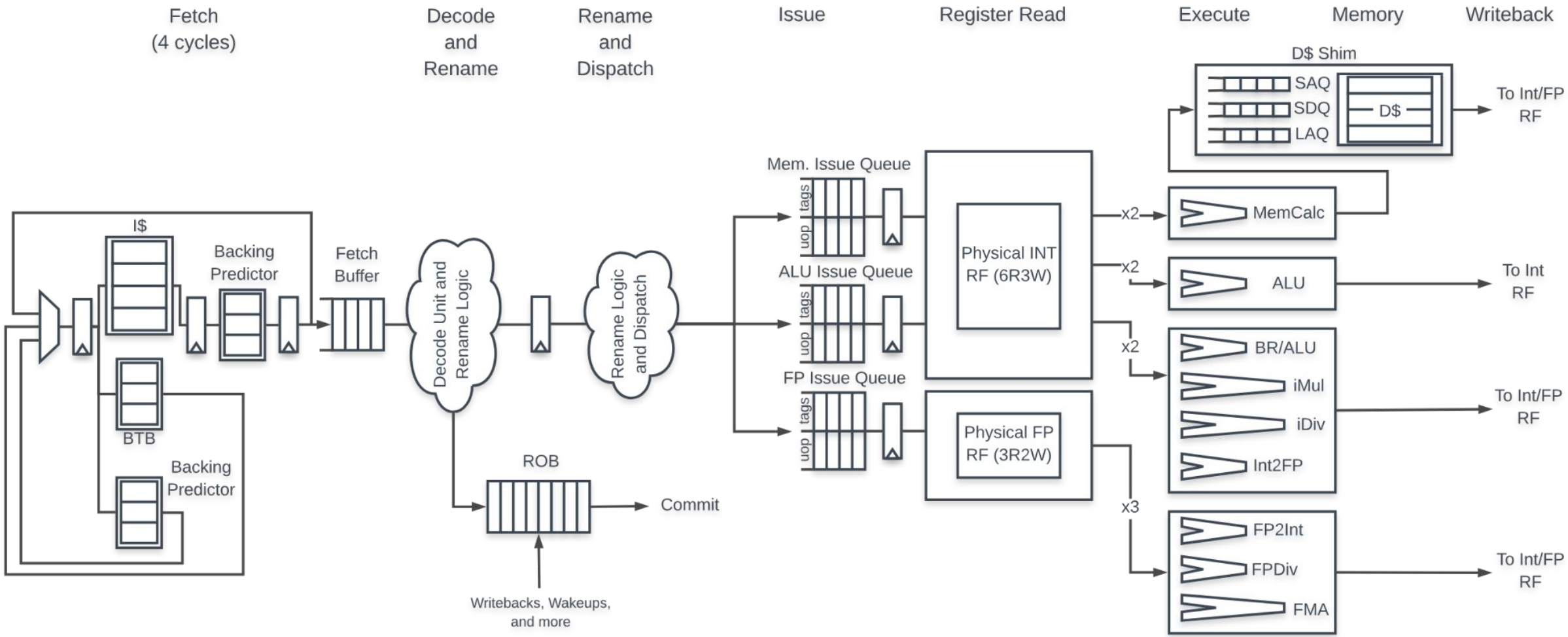
Recovery in BOOM

- An additional, optional "Committed Map Table" holds the rename map for the committed architectural state. If enabled, this allows single-cycle reset of the pipeline during flushes and exceptions (the current map table is reset to the Committed Map Table).

When to free?

- For instructions that will write a register, the Map Table is read to get the stale physical destination specifier ("stale pdst"). Once the instruction commits, the stale pdst is returned to the Free List, as no future instructions will read it.

Simplified Pipeline



What's next?

- Details of branch prediction
- Execution engine

Acknowledgement

- This course is partly inspired by the following courses created by my colleagues:
  - 6.888 MIT (Yan)
  - CS 598 UIUC (Fletcher)
  - CS 7290 Georgia Tech (Hyesoon Kim)

End of Presentation