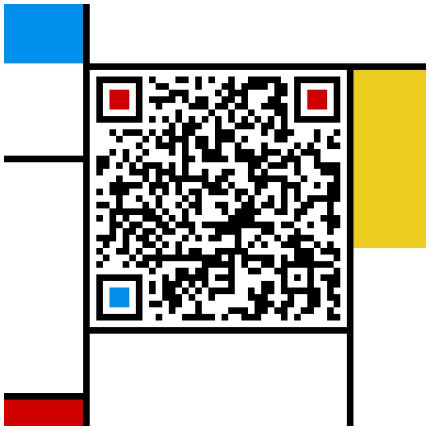
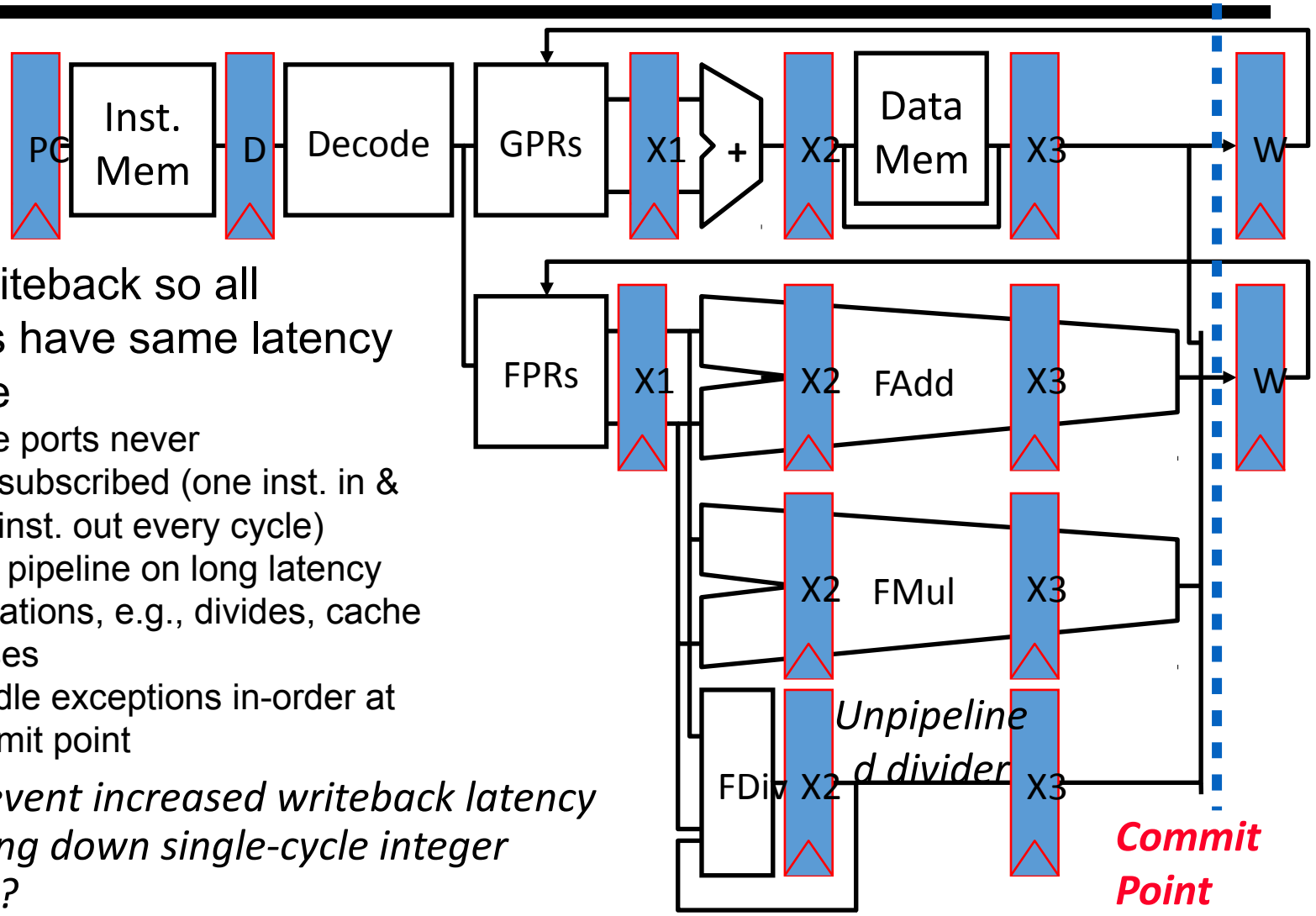
计算机组成与系统结构
Computer Organization & System Architecture

Huang Kejie (黄科杰) 百人计划研究员
Office: 玉泉校区老生仪楼 304
Email address: huangkejie@zju.edu.cn
HP: 17706443800



Recap: Complex In-Order Pipeline

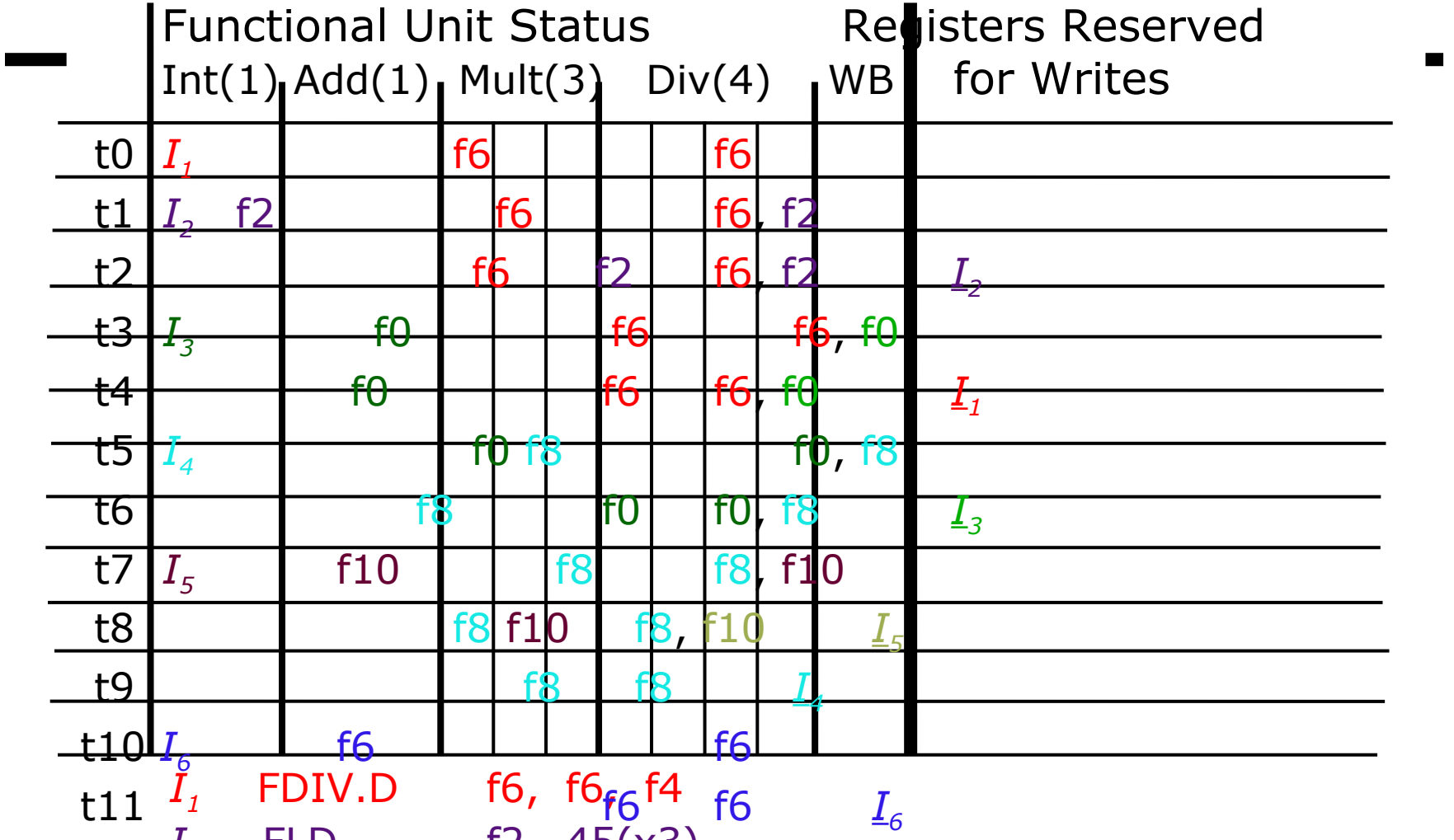


- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
  - Stall pipeline on long latency operations, e.g., divides, cache misses
  - Handle exceptions in-order at commit point

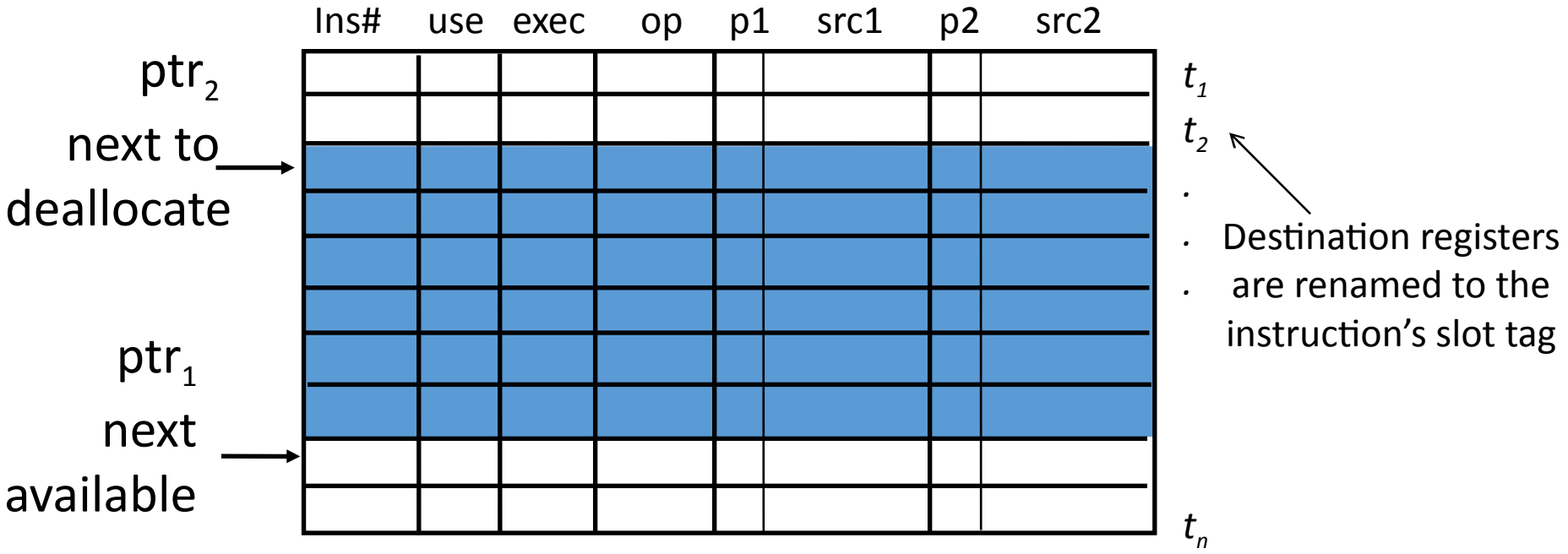
How to prevent increased writeback latency from slowing down single-cycle integer operations?

Bypassing

Scoreboard Dynamics



Reorder Buffer Management



ROB managed circularly

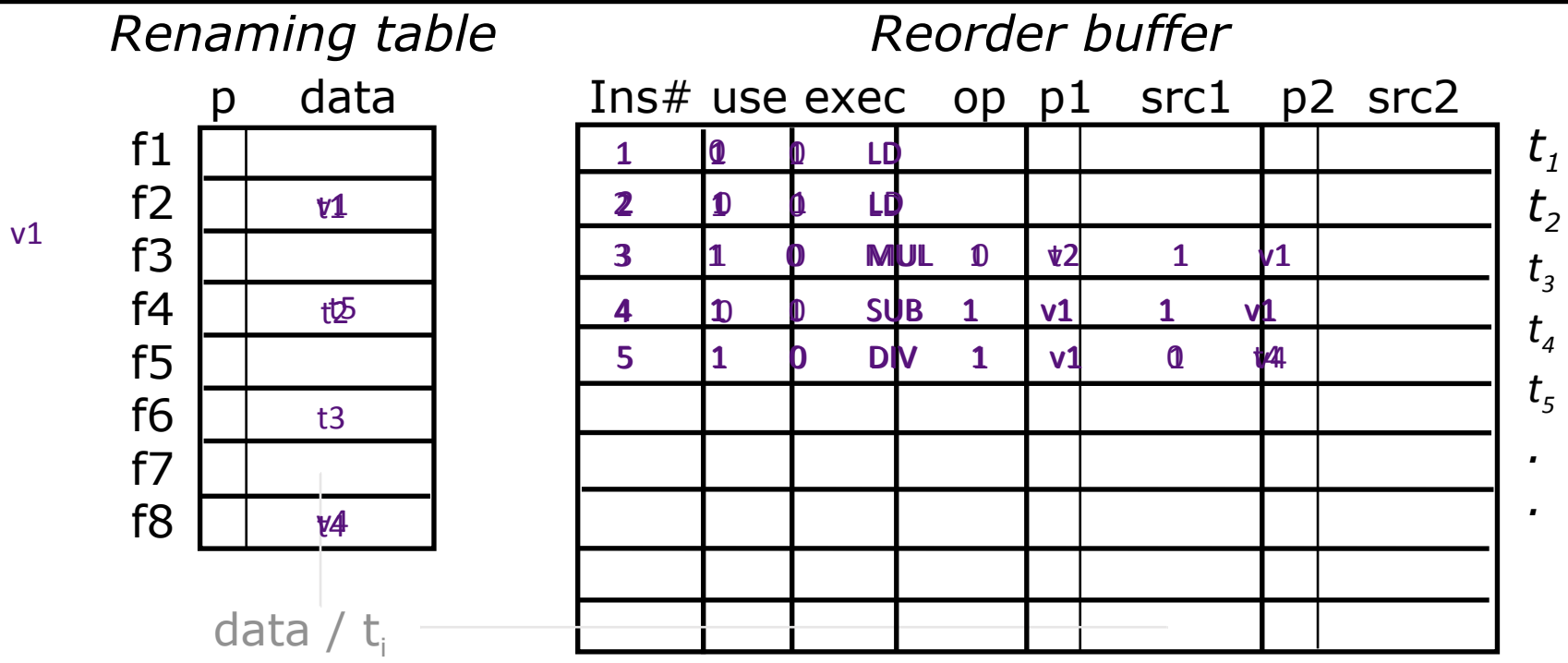
- "exec" bit is set when instruction begins execution
- When an instruction completes its "use" bit is marked free
- ptr_2 is incremented only if the "use" bit is marked free

Instruction slot is candidate for execution when:

- It holds a valid instruction ("use" bit is set)
- It has not already started execution ("exec" bit is clear)
- Both operands are available (p1 and p2 are set)

Renaming & Out-of-order Issue

An example



1 FLD	f2,	34(x2)	
2 FLD	f4,	45(x3)	
3 FMULT.D	f6,	f4,	f2
4 FSUB.D	f8,	f2,	f2
5 FDIV.D	f4,	f2,	f8
6 FADD.D	f10,	f6,	f4

- *When are tags in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

Physical Register Management

Rename Table

x0	
x1	P8
x2	
x3	P7
x4	
x5	
x6	P5
x7	P6

Physical Regs

P0		
P1		
P2		
P3		
P4		
P5	<x6>	p
P6	<x7>	p
P7	<x3>	p
P8	<x1>	p
Pn		

Free List

P0
P1
P3
P2
P4

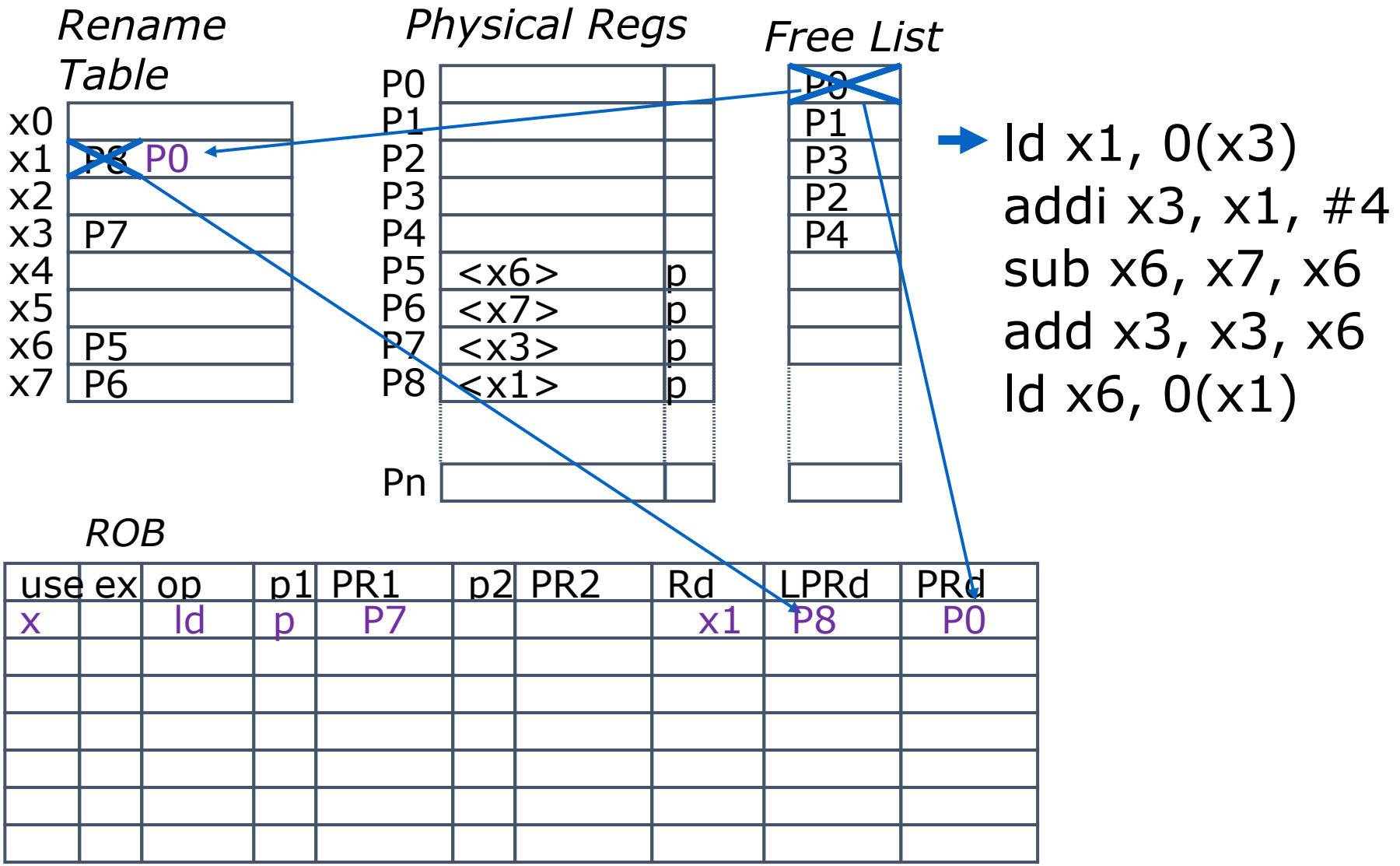
```
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)
```

ROB

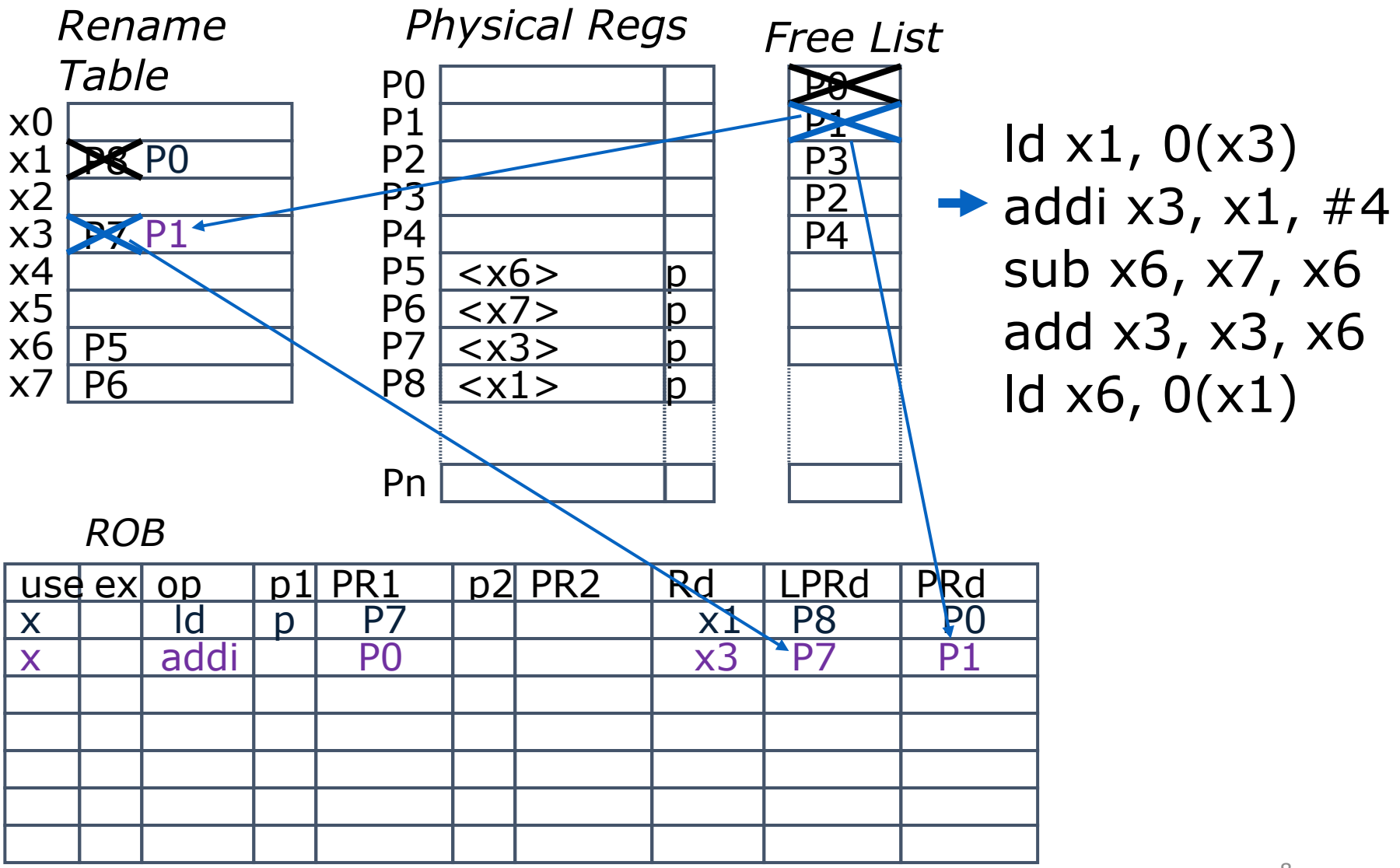
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

(LPRd requires third read port on Rename Table for each instruction)

Physical Register Management

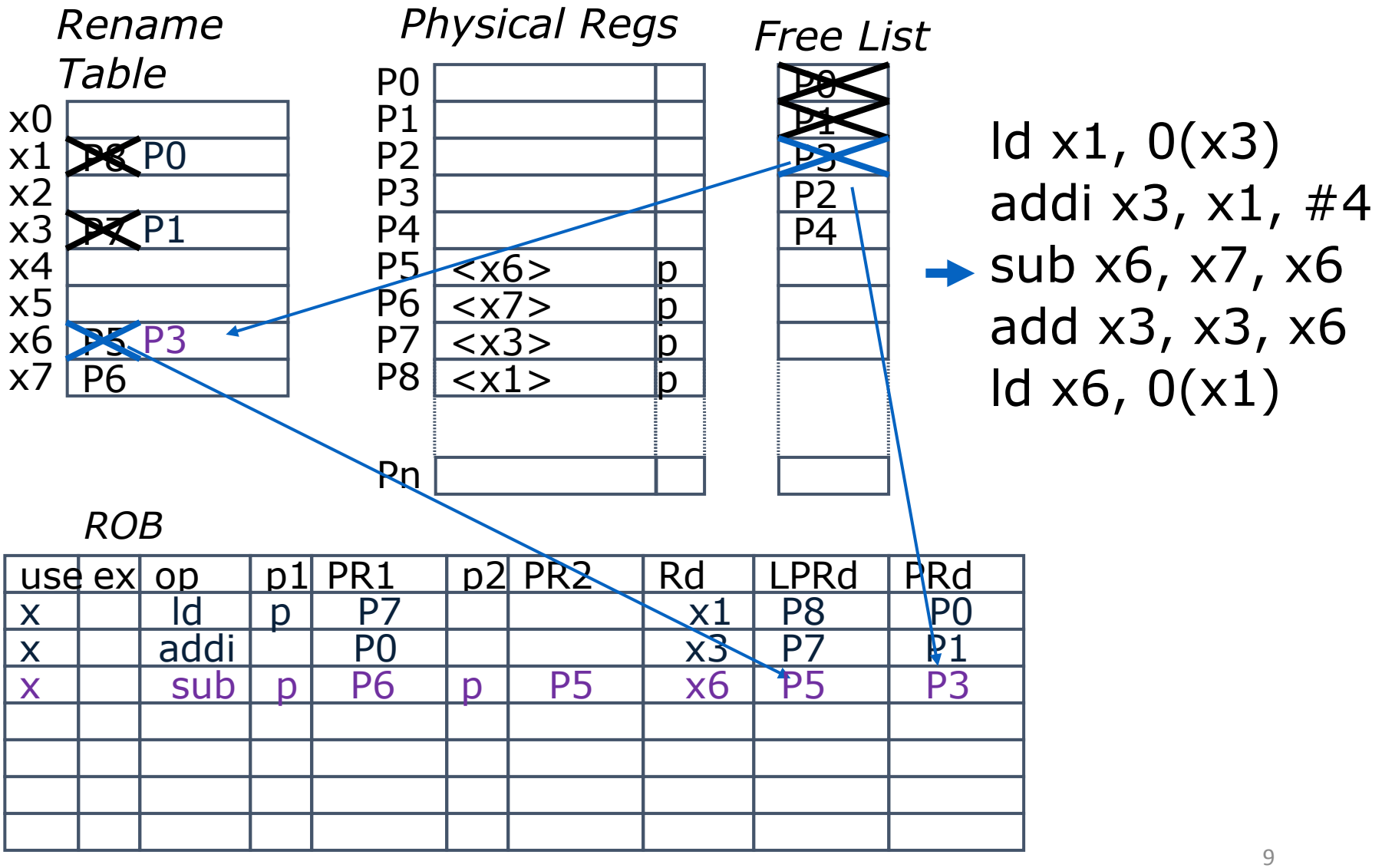


Physical Register Management



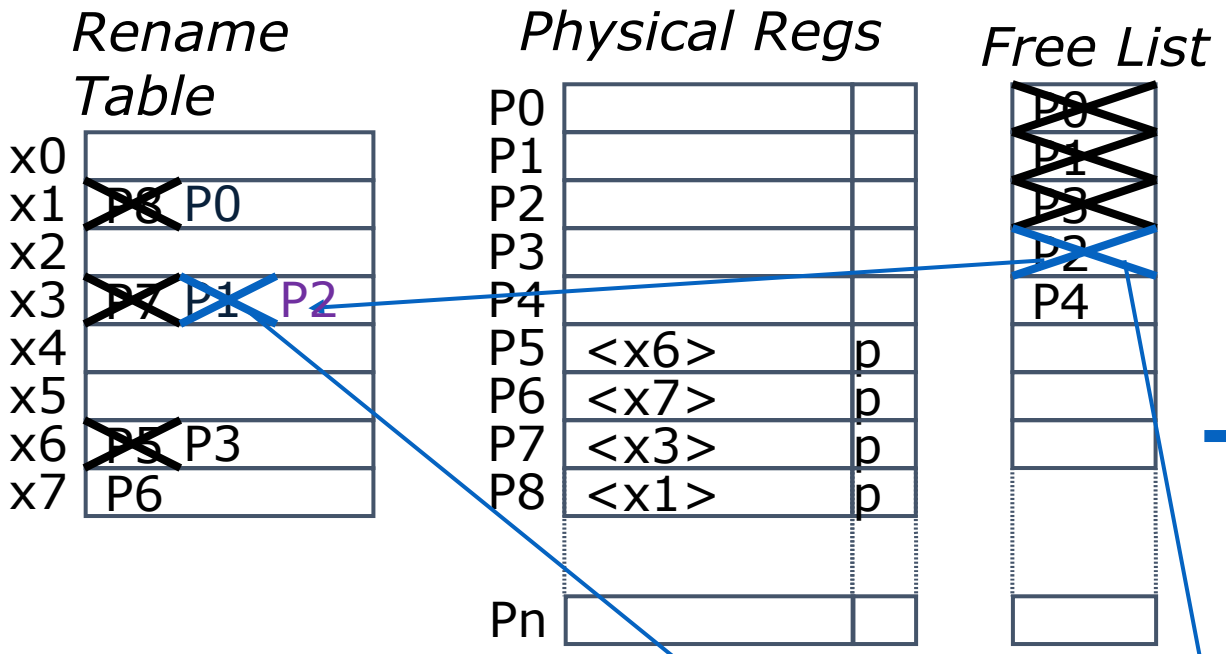
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1

Physical Register Management



use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3

Physical Register Management

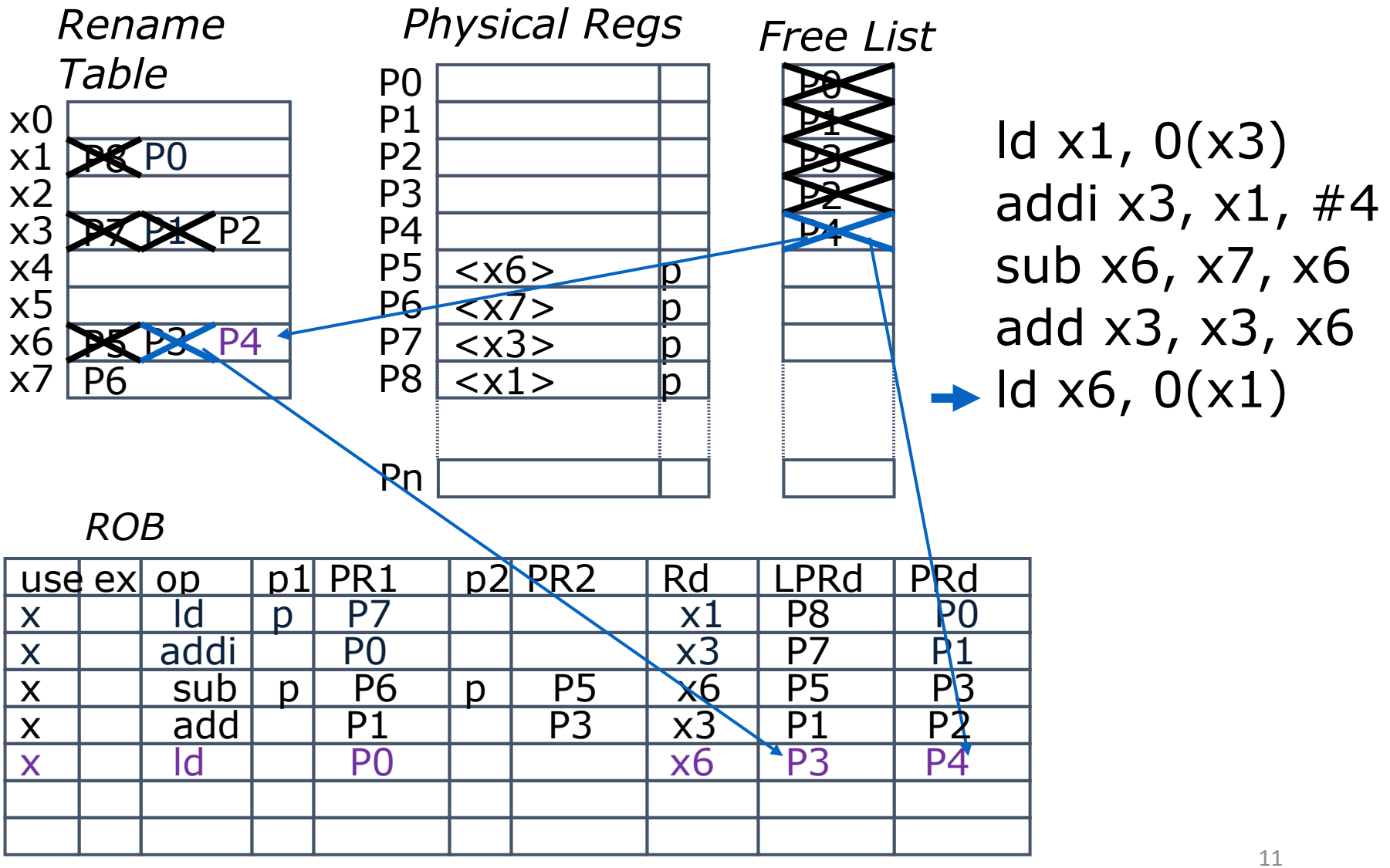


ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
→ add x3, x3, x6
ld x6, 0(x1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2

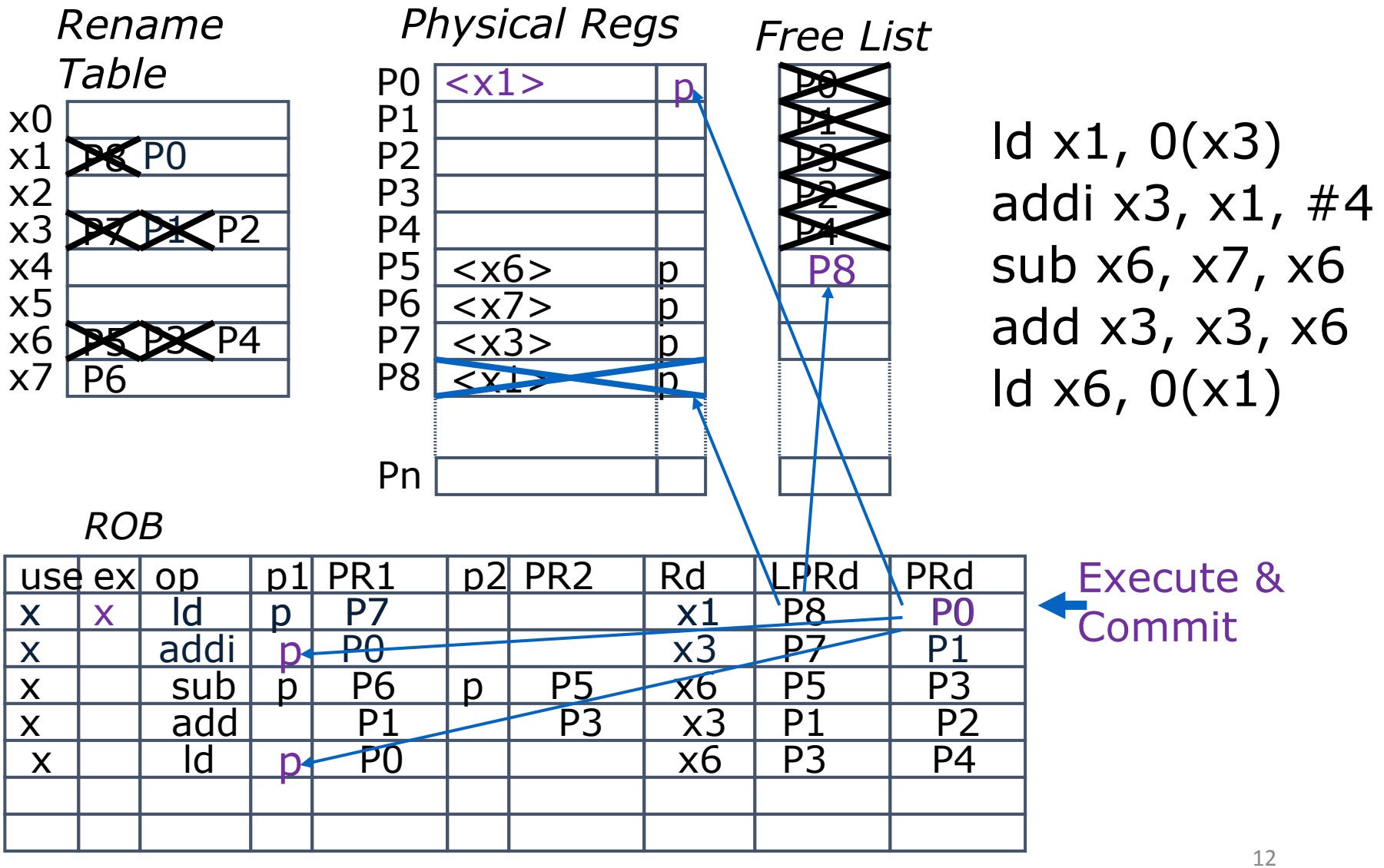
Physical Register Management



ROB

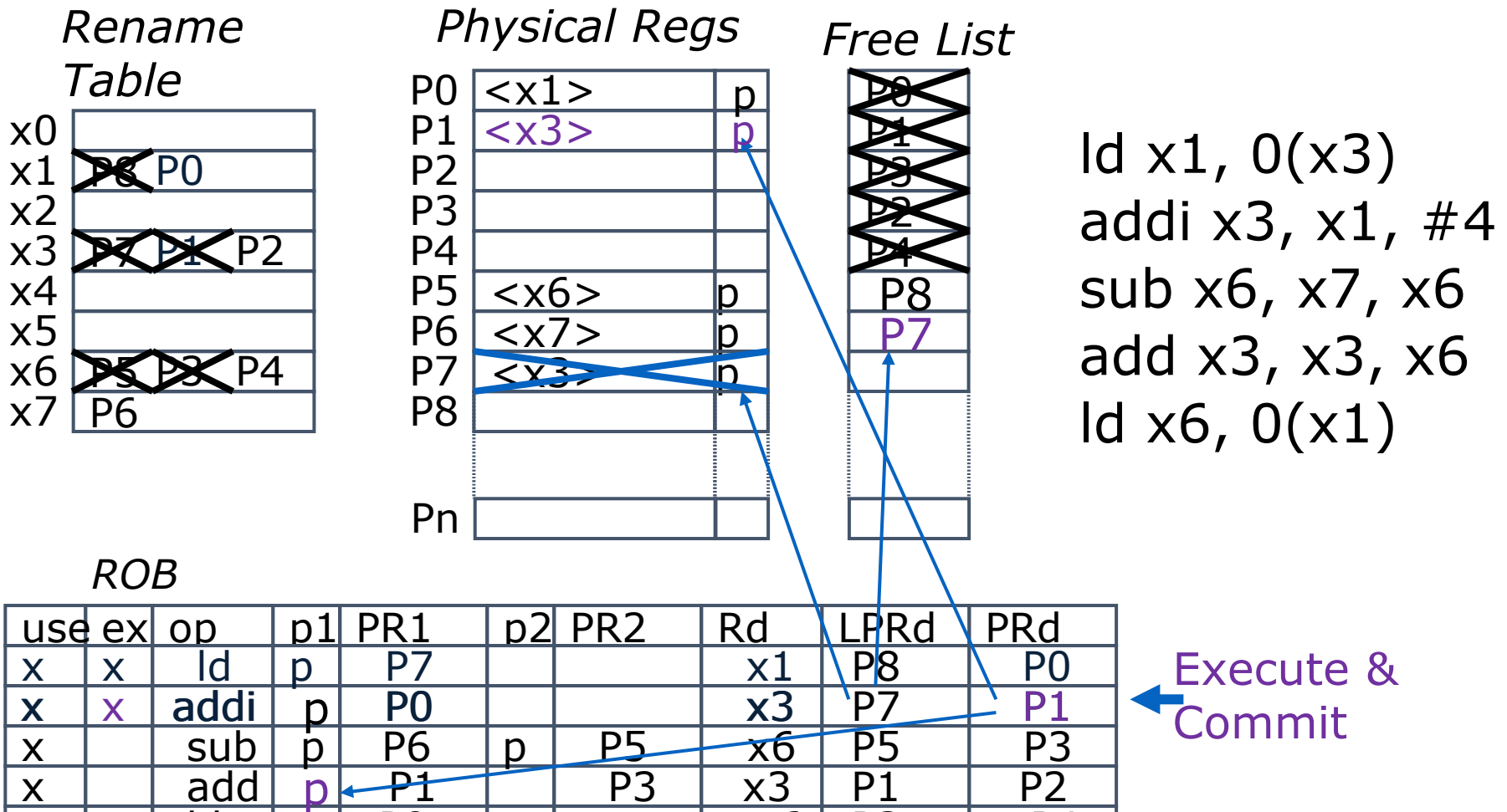
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2
x		ld		P0			x6	P3	P4

Physical Register Management



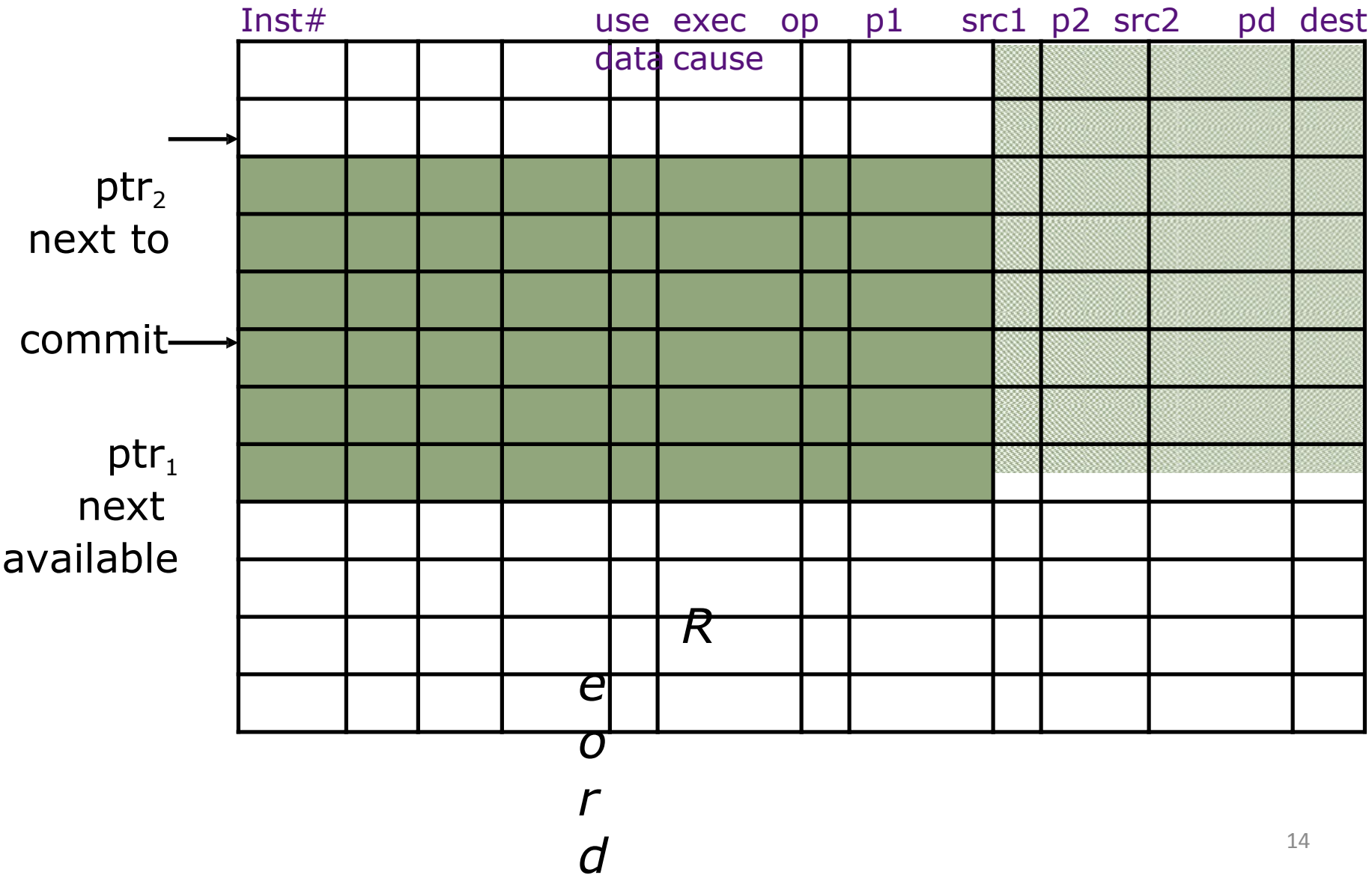
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			x1	P8	P0
x		addi	p	P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2
x		ld	p	P0			x6	P3	P4

Physical Register Management

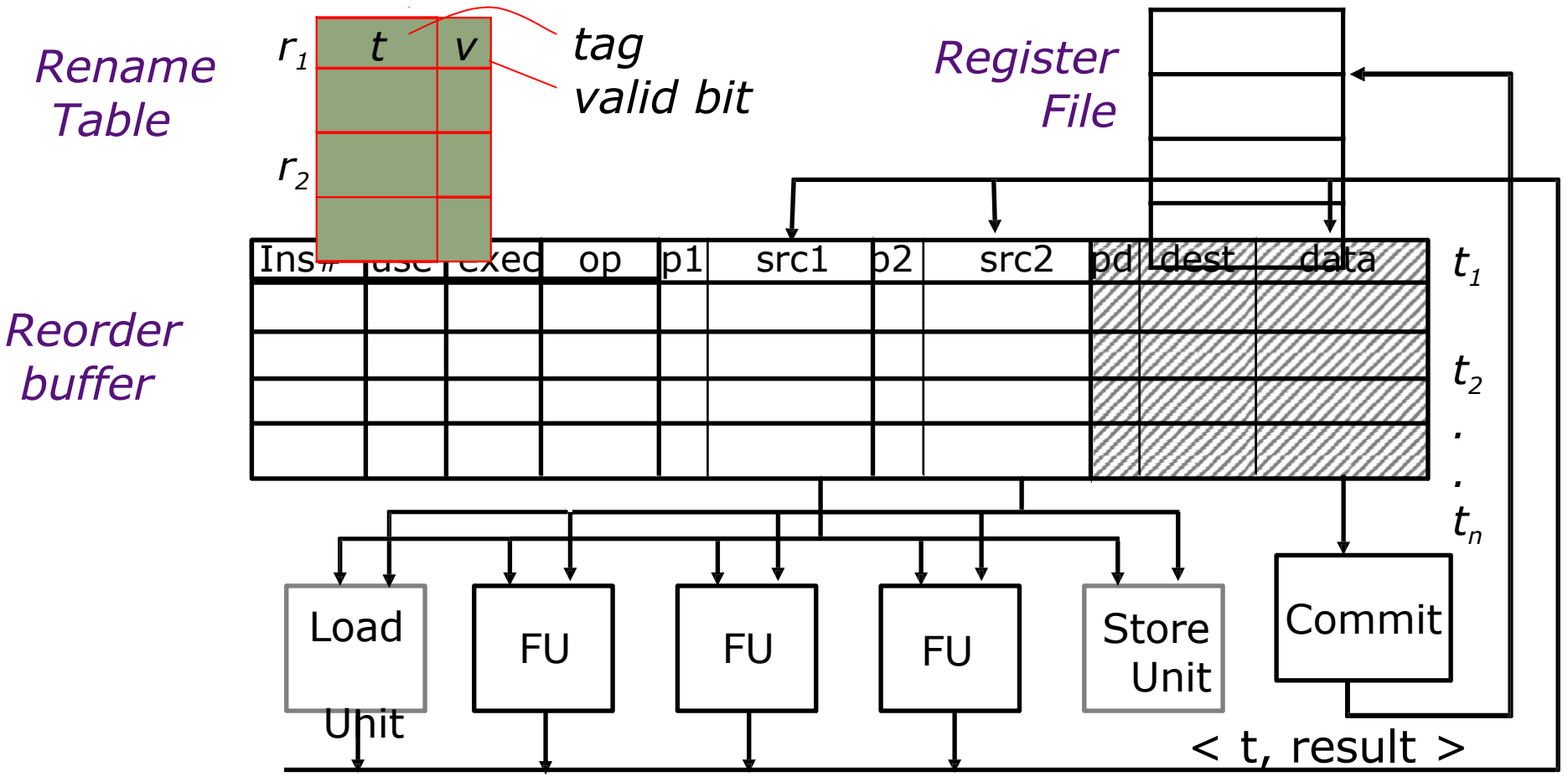


use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			x1	P8	P0
x	x	addi	p	P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add	p	P1		P3	x3	P1	P2
x		ld	p	P0			x6	P3	P4

Extensions for Precise Exceptions



Renaming Table

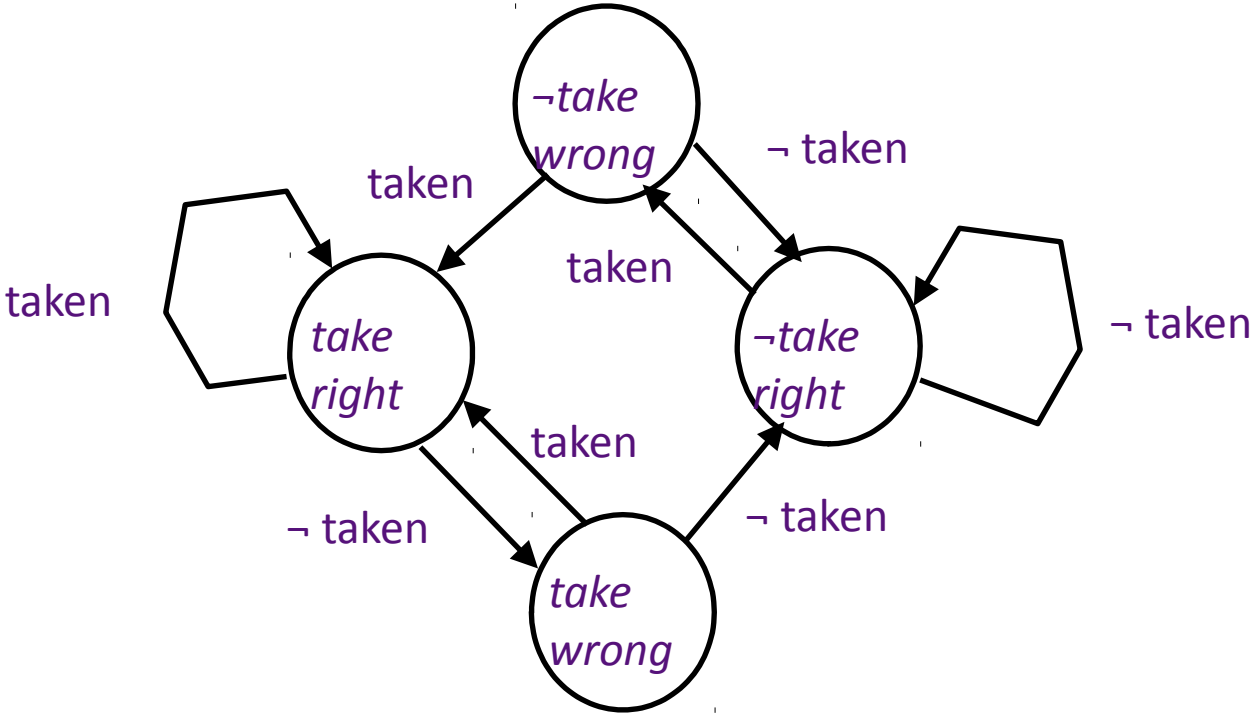


Renaming table is a cache to speed up register name look up. It needs to be cleared after each exception taken.

When else are valid bits cleared? *Control transfers*

Branch Prediction Bits

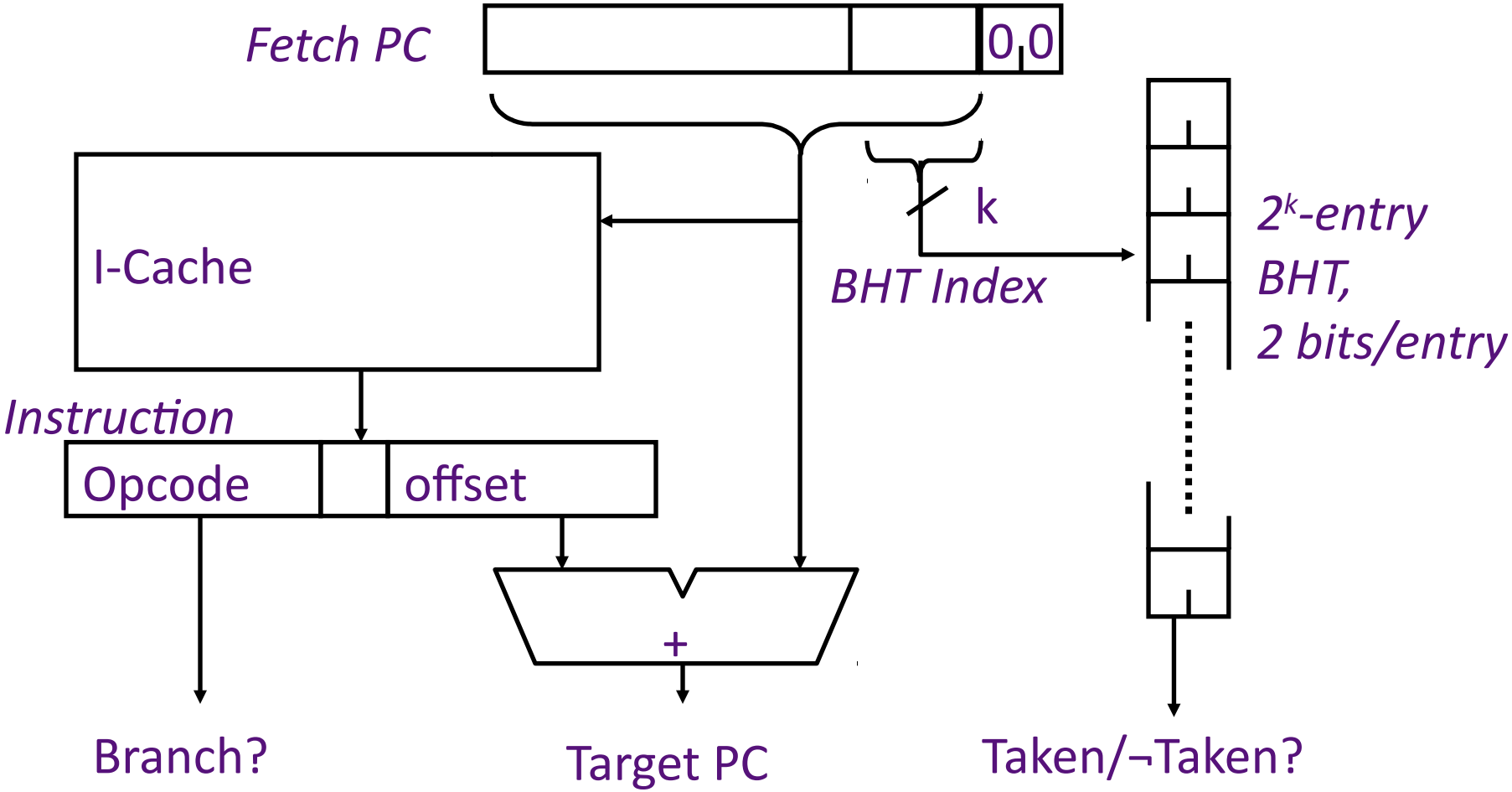
- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!



BP state:

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

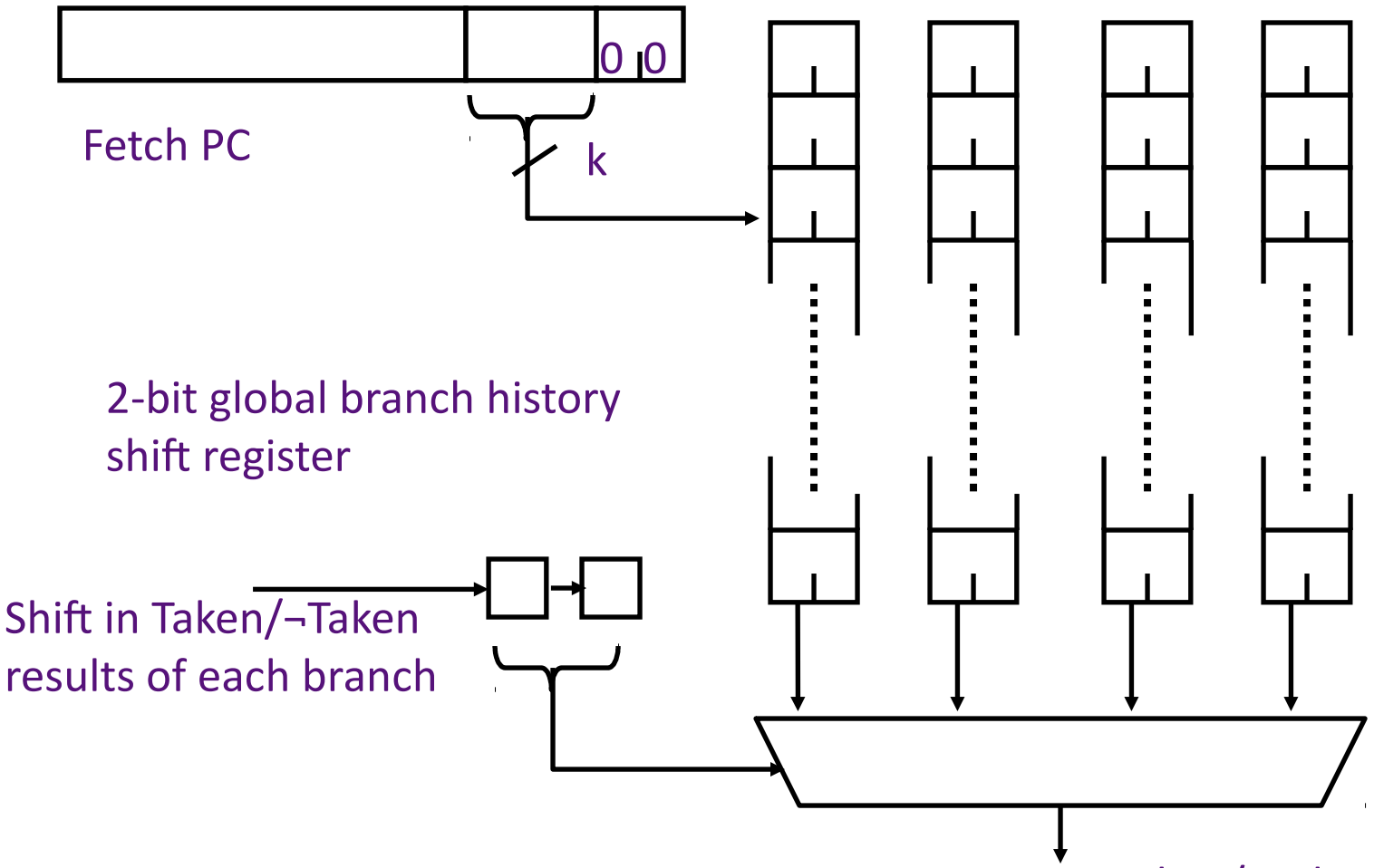
Branch History Table (BHT)



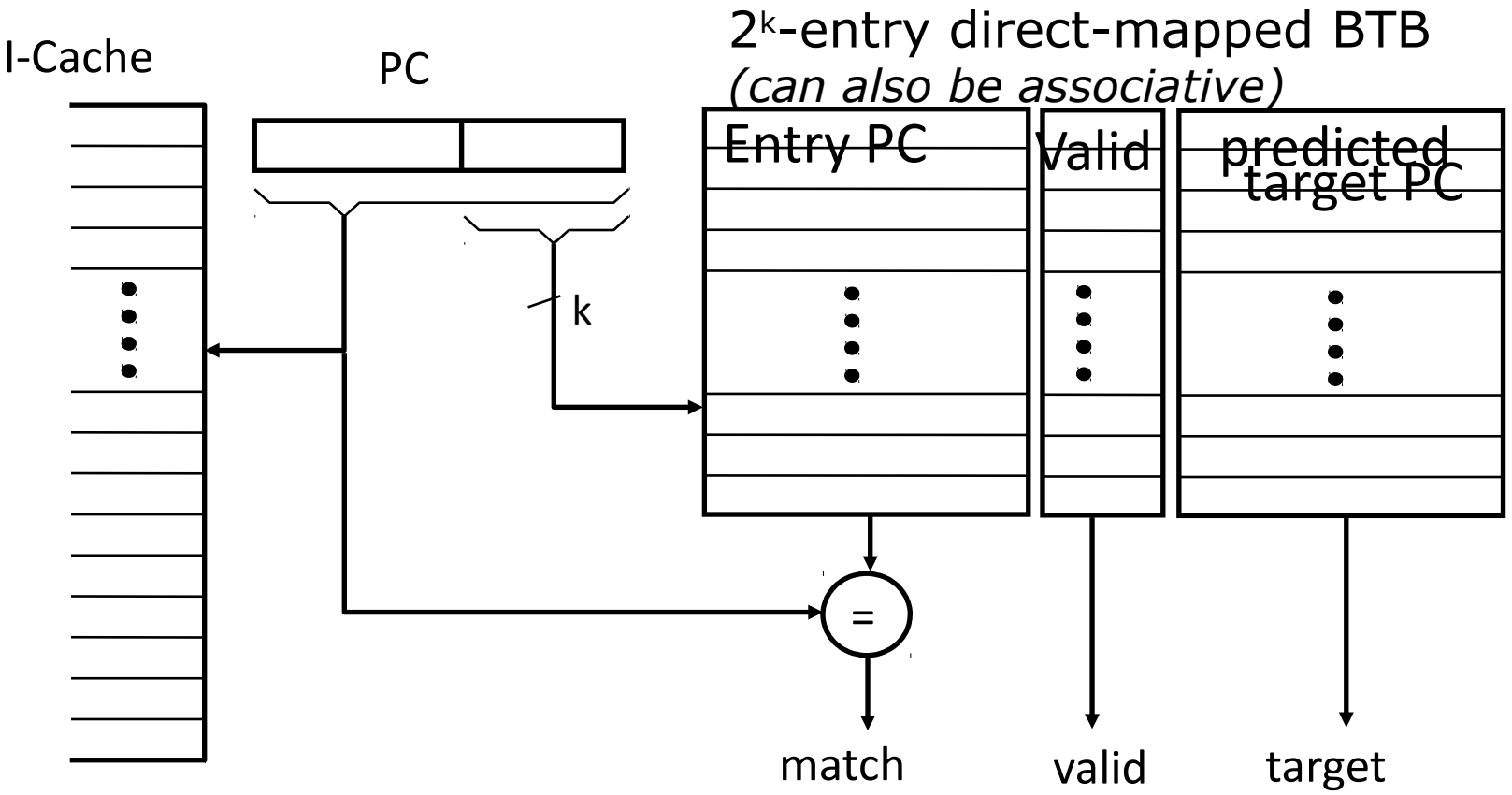
4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

Two-Level Branch Predictor

Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)



Branch Target Buffer (BTB)

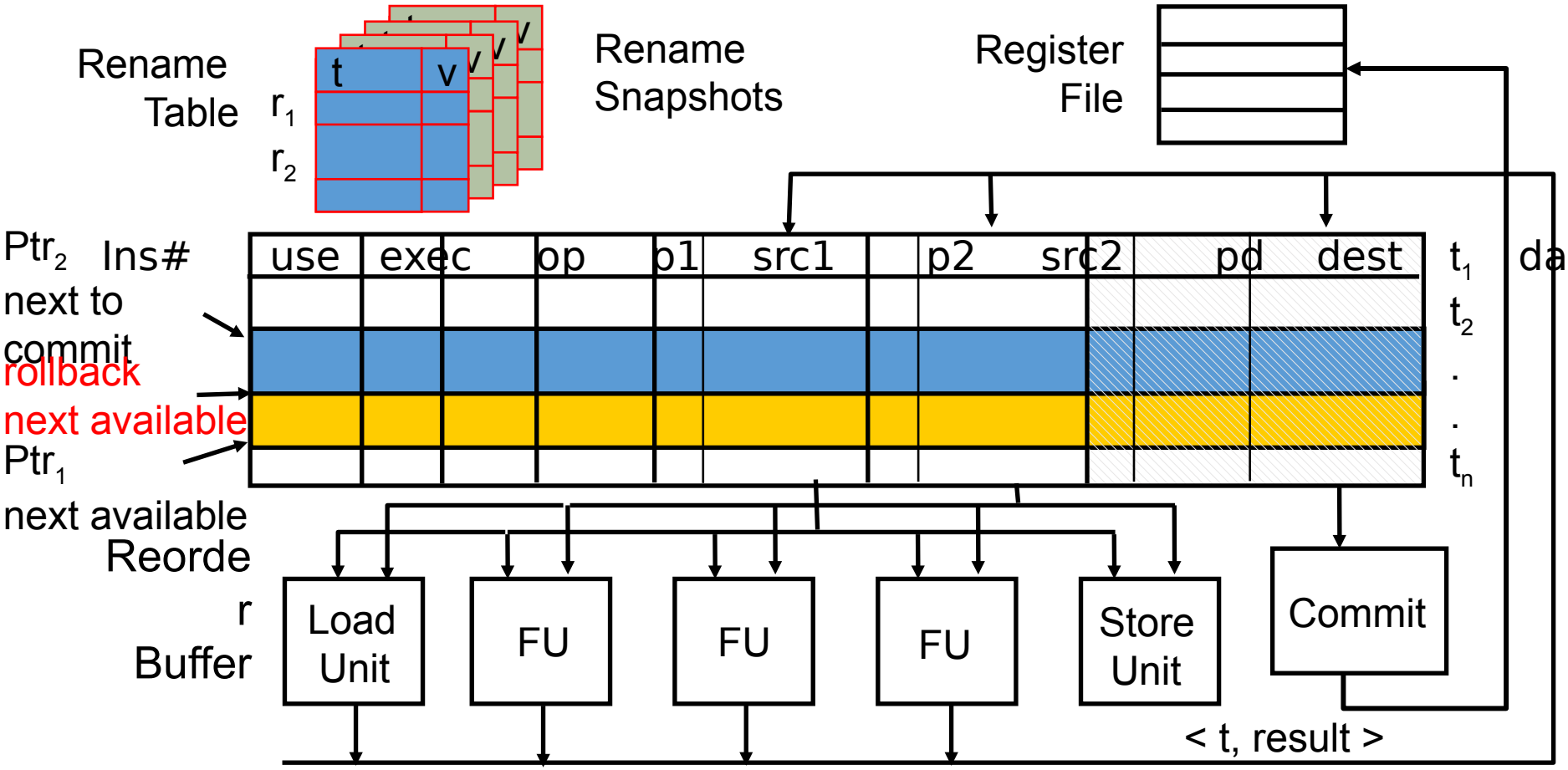


- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded

Rename Table Recovery

- Have to quickly recover rename table on branch mispredicts
- MIPS R10K only has four snapshots for each of four outstanding speculative branches
- Alpha 21264 has 80 snapshots, one per ROB instruction

Mispredict Recovery

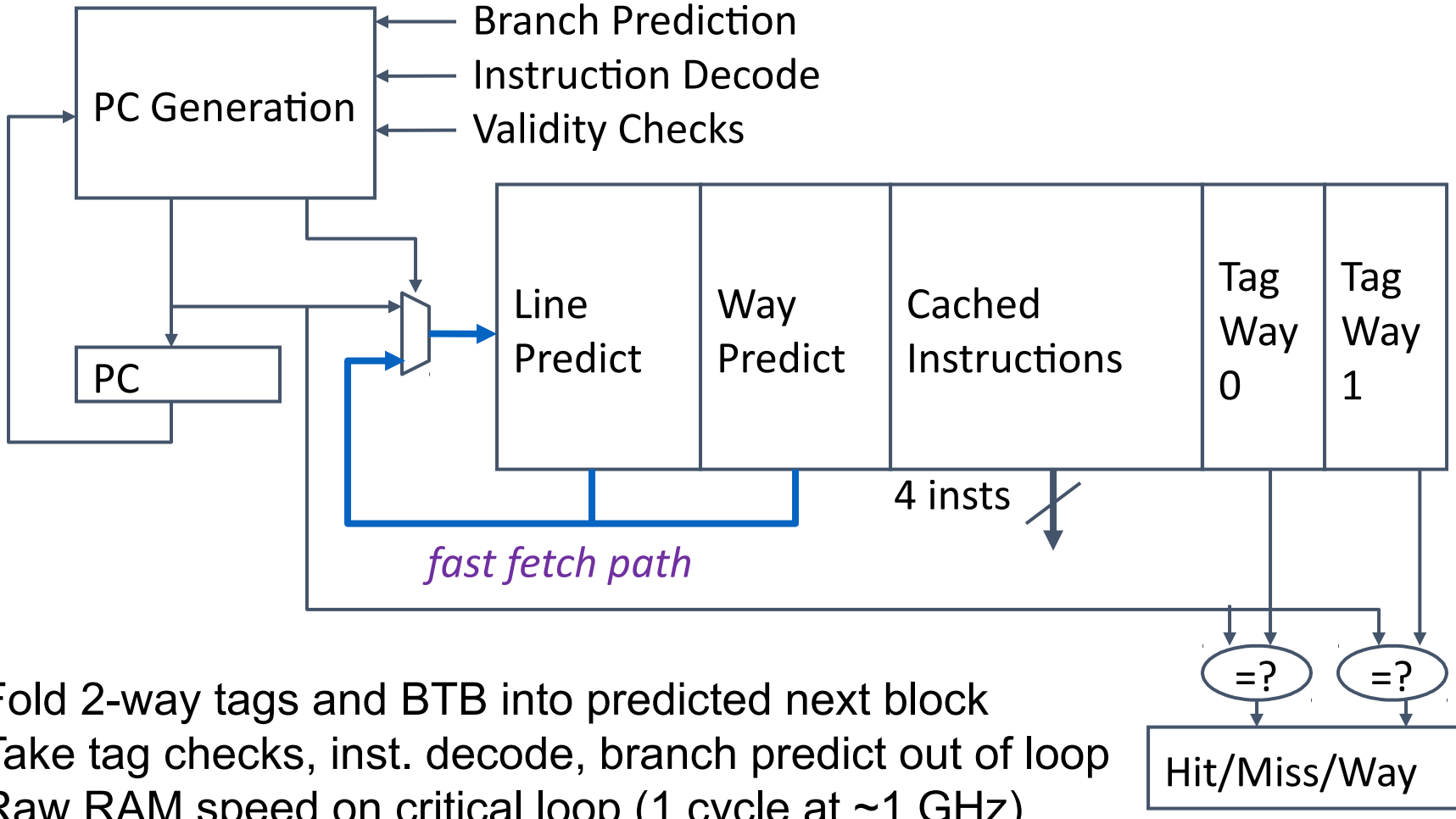


- Take snapshot of register rename table at each predicted branch, recover earlier snapshot if branch mispredicted

Improving Instruction Fetch

- Performance of speculative out-of-order machines often limited by instruction fetch bandwidth
  - speculative execution can fetch 2-3x more instructions than are committed
  - mispredict penalties dominated by time to refill instruction window
  - taken branches are particularly troublesome

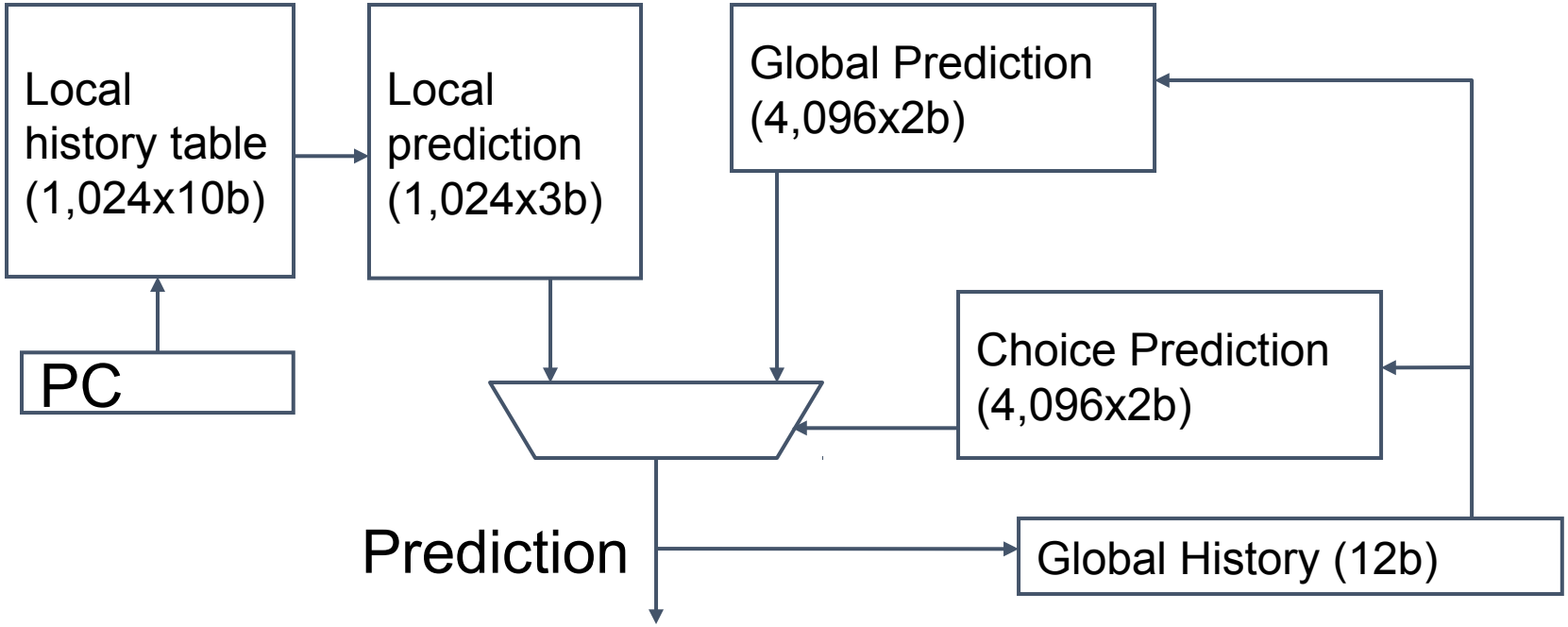
Increasing Taken Branch Bandwidth (Alpha 21264 I-Cache)



- Fold 2-way tags and BTB into predicted next block
- Take tag checks, inst. decode, branch predict out of loop
- Raw RAM speed on critical loop (1 cycle at ~1 GHz)
- 2-bit hysteresis counter per block prevents overtraining

Tournament Branch Predictor (Alpha 21264)

- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications



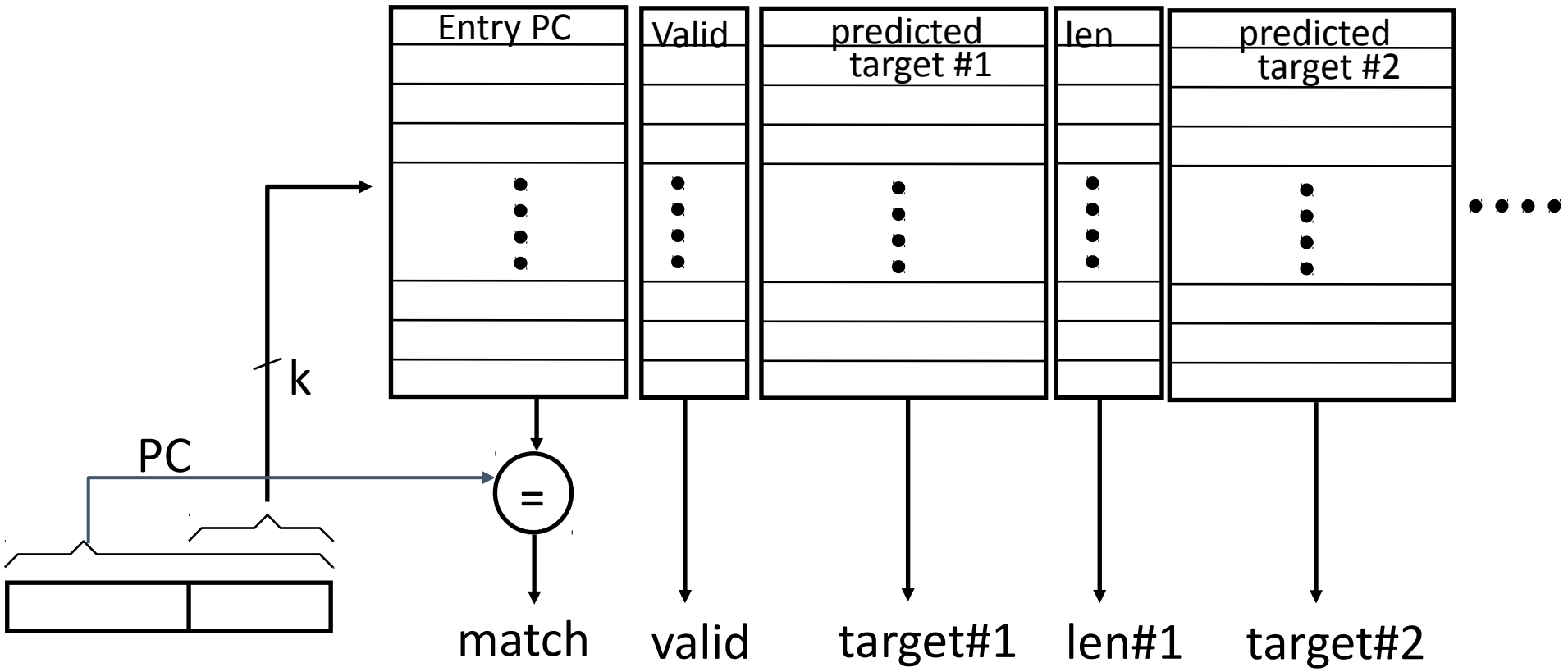
Taken Branch Limit

- Integer codes have a taken branch every 6-9 instructions

- To avoid fetch bottleneck, must execute multiple taken branches per cycle when increasing performance

- This implies:
  - predicting multiple branches per cycle
  - fetching multiple non-contiguous blocks per cycle

Branch Address Cache (Yeh, Marr, Patt)



Extend BTB to return multiple branch predictions per cycle

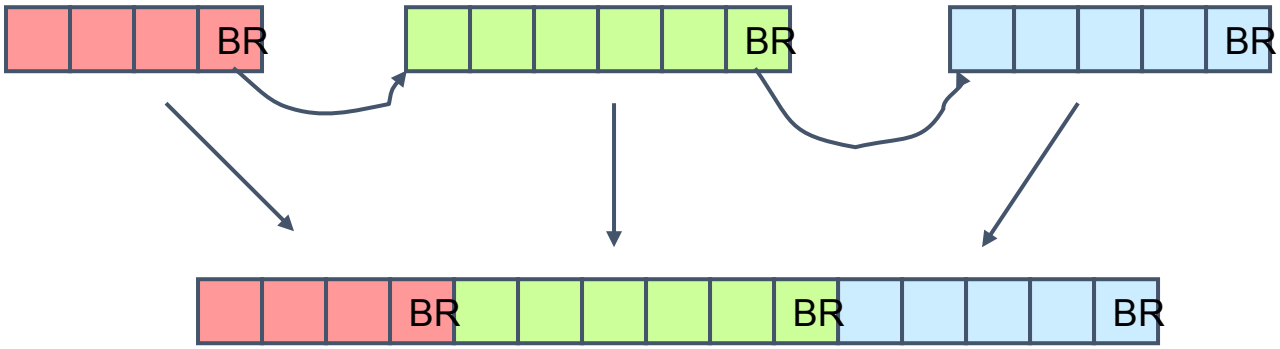
Fetching Multiple Basic Blocks

- Requires either
  - multiported cache: expensive
  - interleaving: bank conflicts will occur

- Merging multiple blocks to feed to decoders adds latency, increasing mispredict penalty and reducing branch throughput

Trace Cache

- Key Idea: Pack multiple non-contiguous basic blocks into one contiguous trace cache line



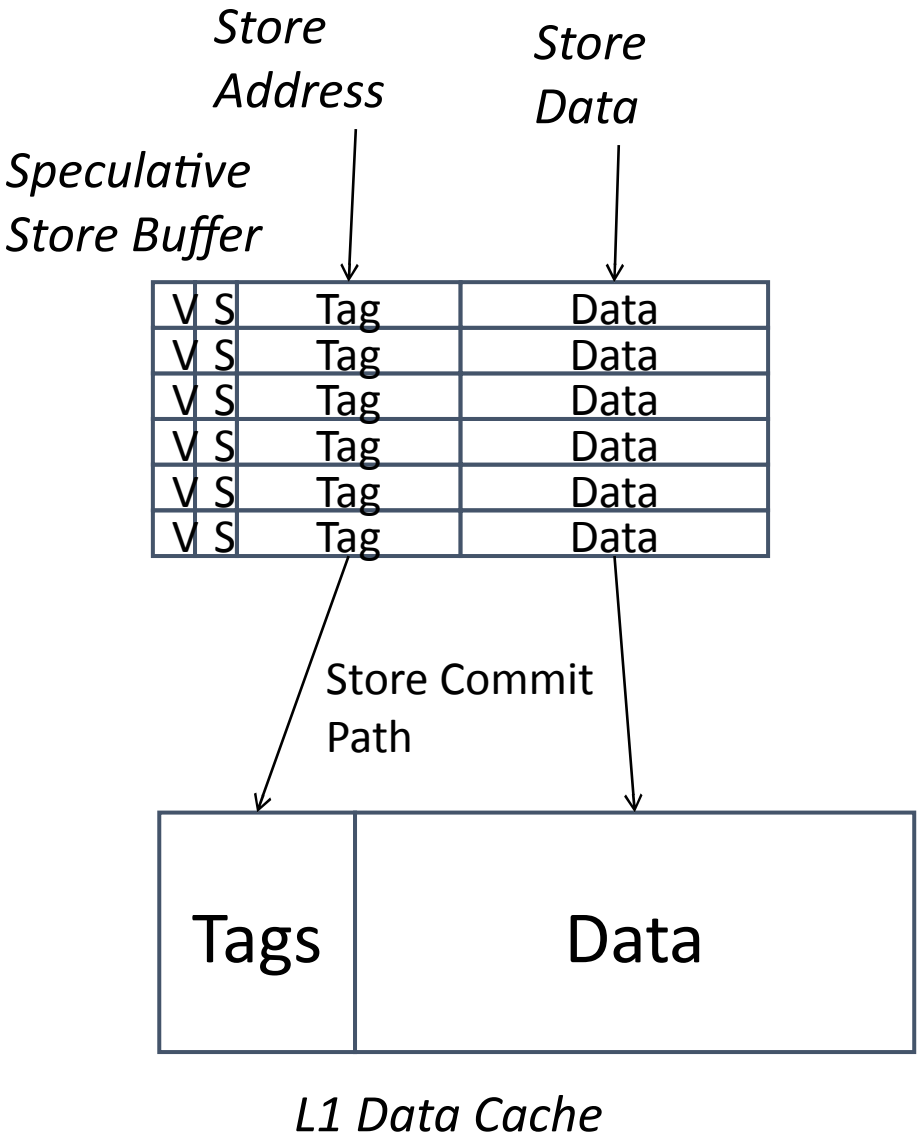
- Single fetch brings in multiple basic blocks
- Trace cache indexed by start address and next n branch predictions
- Used in Intel Pentium-4 processor to hold decoded uops

Load-Store Queue Design

- After control hazards, data hazards through memory are probably next most important bottleneck to superscalar performance

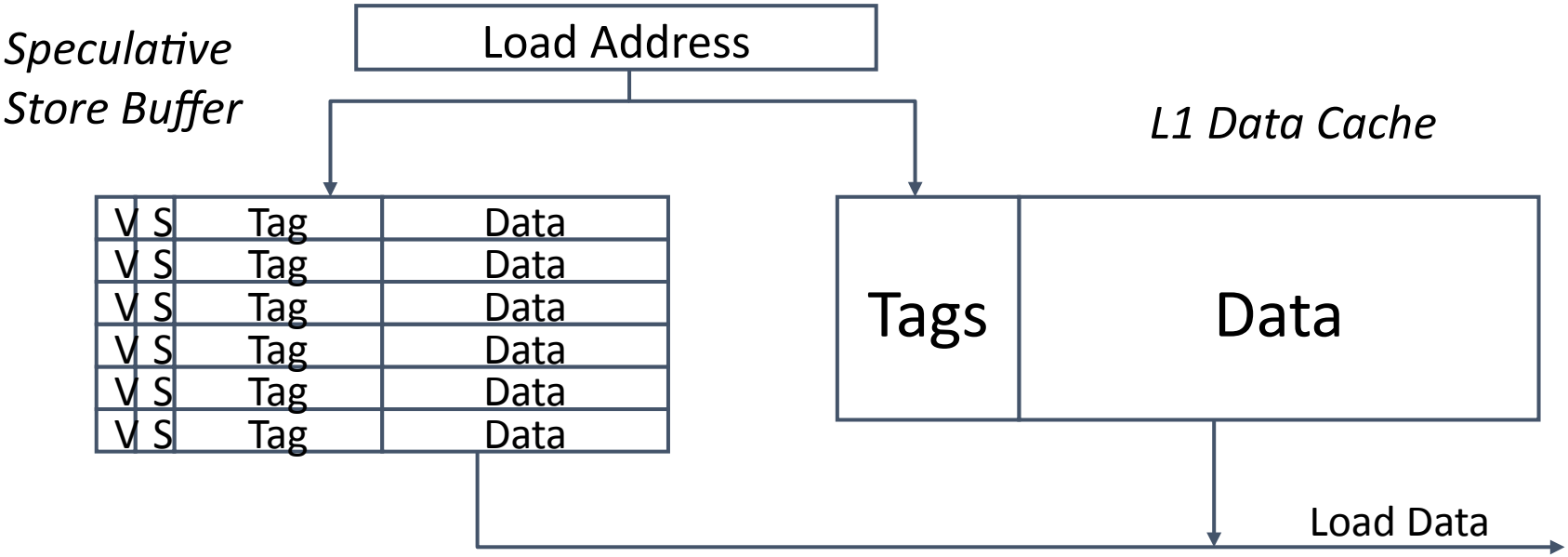
- Modern superscalars use very sophisticated load-store reordering techniques to reduce effective memory latency by allowing loads to be speculatively issued

Speculative Store Buffer



- Just like register updates, stores should not modify the memory until after the instruction is committed. A speculative store buffer is a structure introduced to hold speculative store data.
- During decode, store buffer slot allocated in program order
- Stores split into "store address" and "store data" micro-operations
- "Store address" execution writes tag
- "Store data" execution writes data
- Store commits when oldest instruction and both address and data available:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit

Load bypass from speculative store buffer



- If data in both store buffer and cache, which should we use?
  Speculative store buffer
- If same address in store buffer twice, which should we use?
  Youngest store older than load

Memory Dependencies

```
sd x1, (x2)
ld x3, (x4)
```

- When can we execute the load?

In-Order Memory Queue

- Execute all loads and stores in program order
  - => Load and store cannot leave ROB for execution until all previous loads and stores have completed execution
- Can still execute loads and stores speculatively, and out-of-order with respect to other instructions
- Need a structure to handle memory ordering…

Conservative O-o-O Load Execution

```
sd x1, (x2)
ld x3, (x4)
```

- Can execute load before store, if addresses known and `x4 != x2`

- Each load address compared with addresses of all previous uncommitted stores
  - can use partial conservative check i.e., bottom 12 bits of address, to save hardware

- Don't execute load if any previous store address not known

- (MIPS R10K, 16-entry address queue)

Address Speculation

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that **`x4 != x2`**
- Execute load before store address known
- Need to hold all completed but uncommitted load/store addresses in program order
- If subsequently find **`x4==x2`**, squash load and all following instructions
- => Large penalty for inaccurate address speculation

Memory Dependence Prediction (Alpha 21264)

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that **`x4 != x2`** and execute load before store
- If later find **`x4==x2`**, squash load and all following instructions, but mark load instruction as store-wait
- Subsequent executions of the same load instruction will wait for all previous stores to complete
- Periodically clear store-wait bits

Open-source Berkeley RISC-V Processors

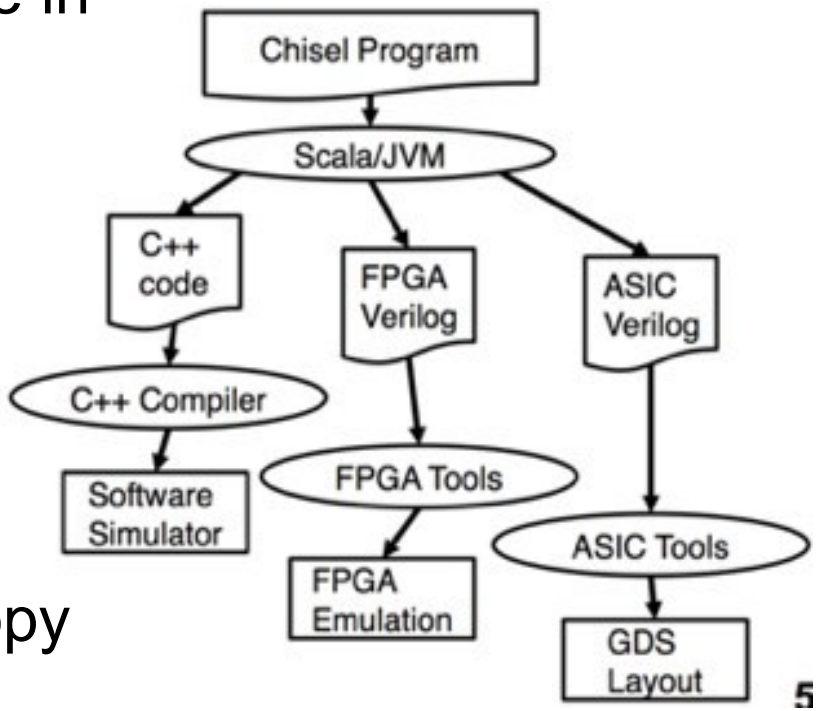
- Sodor Collection
  - RV32I - Entry, educational, not synthesizable
- Rocket-chip SoC generator
- Z-scale
  - RV32IM - micro-controller
- Rocket
  - RV64G - in-order, single-issue application core
- BOOM
  - RV64G - out-of-order, superscalar application core

What is BOOM?

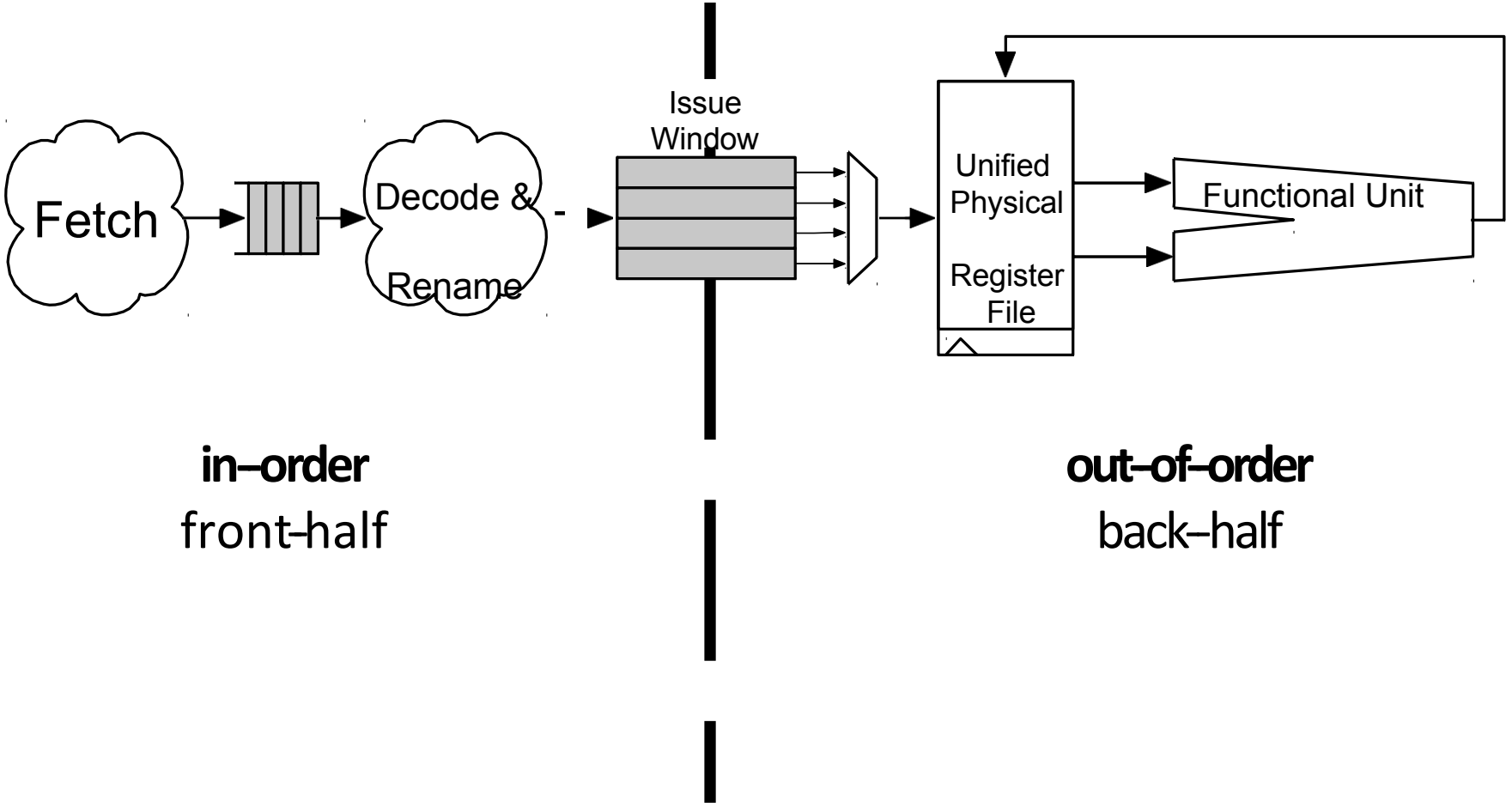
- superscalar, out-of-order processor written in Berkeley's Chisel RTL
- It is synthesizable
- It is parameterizable
- We hope to use it as a plaOorm for architecture research
- It is open-source!
  - BOOM is a work-in-progress

Chisel

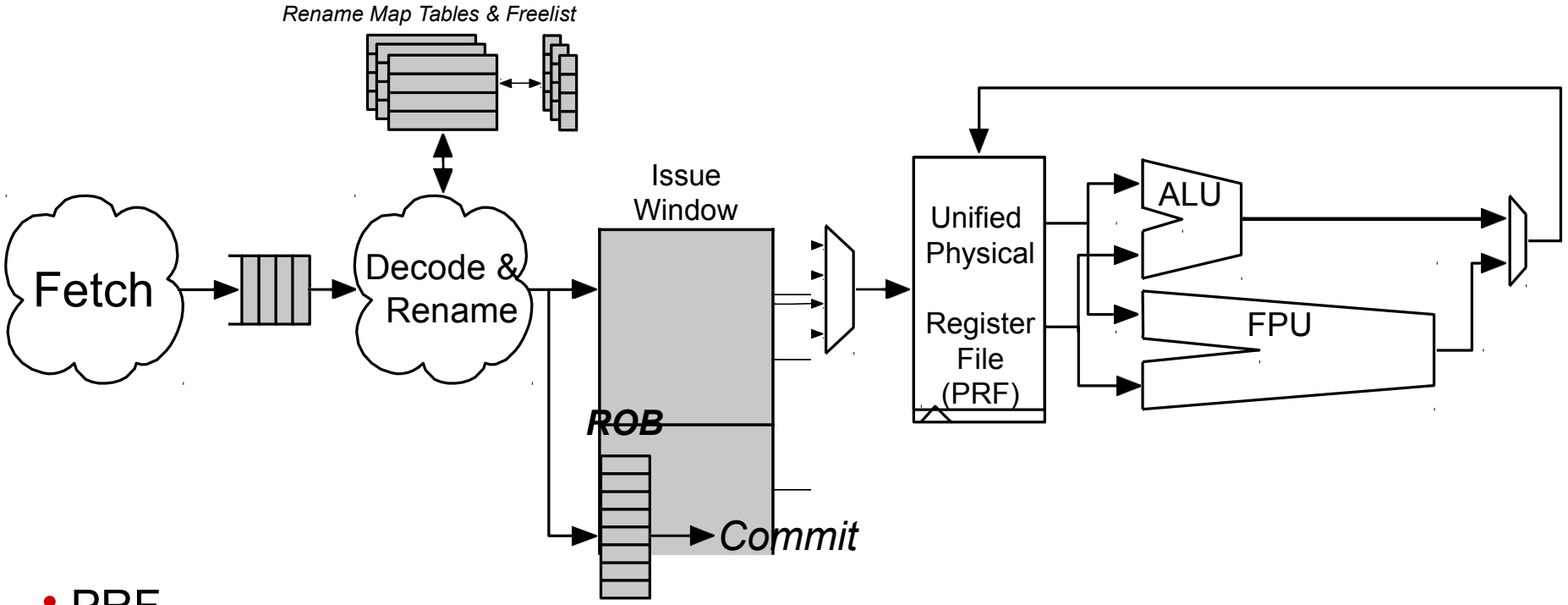
- Hardware Construction Language embedded in Scala
- ***not*** a high-level synthesis language hardware module is a data structure in Scala
- Full power of Scala for writing generators
  - object-oriented programming
  - factory objects, traits, overloading
  - functional programming
  - high-order funs, anonymous funcs, currying
- generated C++ simulator is 1:1 copy of Verilog designs
- version 3.0 coming soon!
  - uses an IR called FIRRTL



BOOM Pipeline

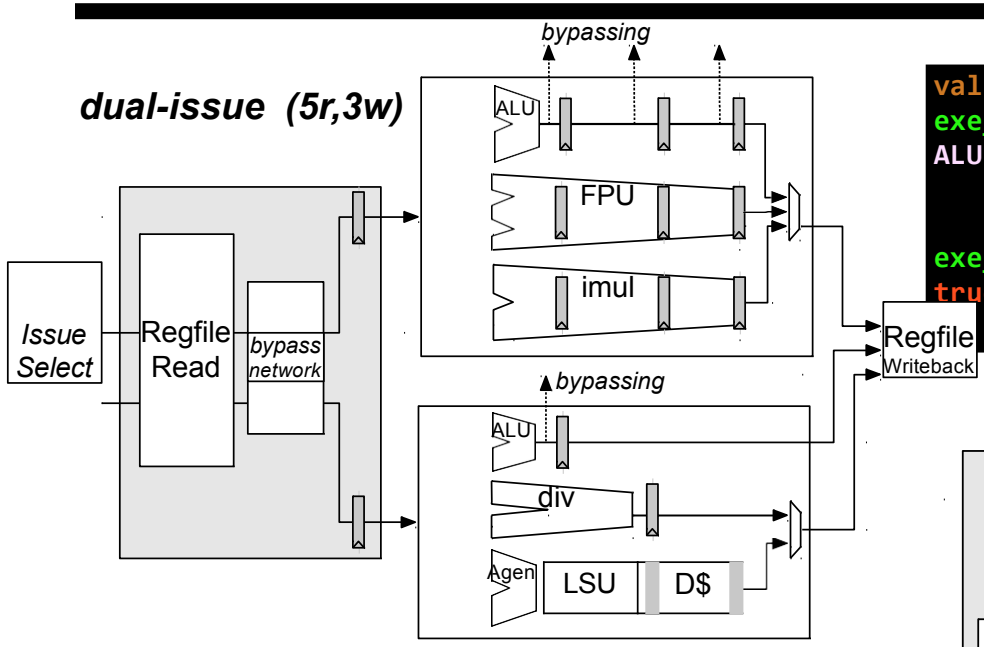


BOOM Pipeline



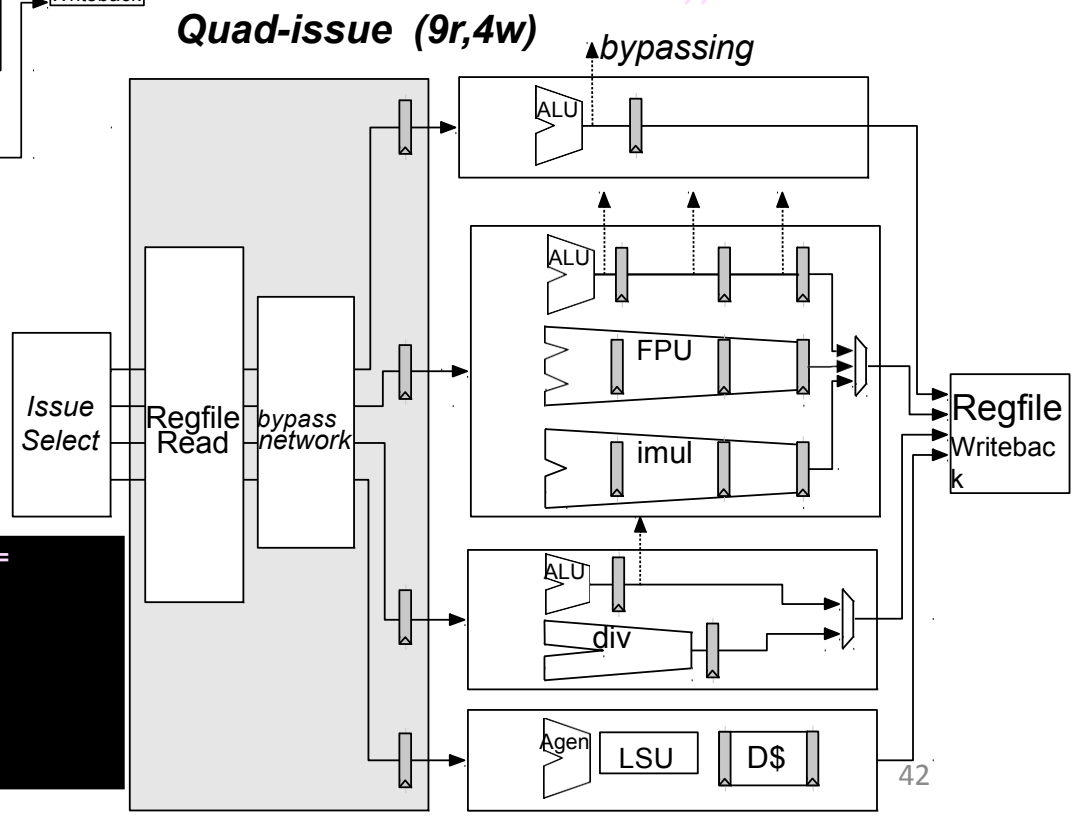
- PRF
  - explicit renaming
  - holds speculative and commi6ed data
  - holds both x-regs, f-regs
- split ROB/issue window design
- Unified Issue Window
  - holds all instructions

Parameterized Superscalar




```
val exe_units = ArrayBuffer[ExecutionUnit]()
exe_units += Module(new                          = true
ALUExeUnit(is_branch_unit                        = true
                                  , has_fpu      = true
                                  , has_mul
exe_units += Module(new ALUMemExeUnit(fp_mem_support =
true                              , has_div      =
                                  true
                                  ))
```

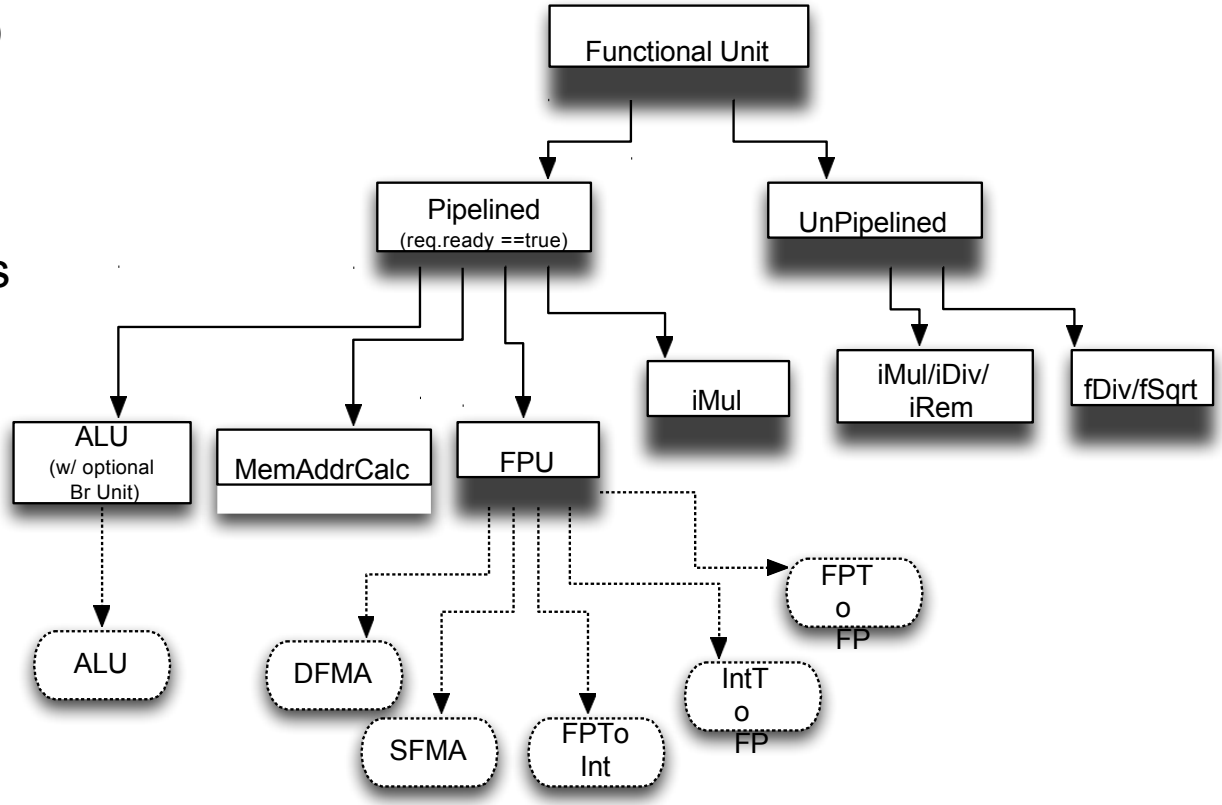
OR




```
exe_units += Module(new ALUExeUnit(is_branch_unit =
true))
exe_units += Module(new ALUExeUnit(has_fpu = true
                                  , has_mul = true
                                  ))
exe_units += Module(new ALUExeUnit(has_div =
true))  exe_units += Module(new MemExeUnit())
```

Open-□source Berkeley RISC-V Processors

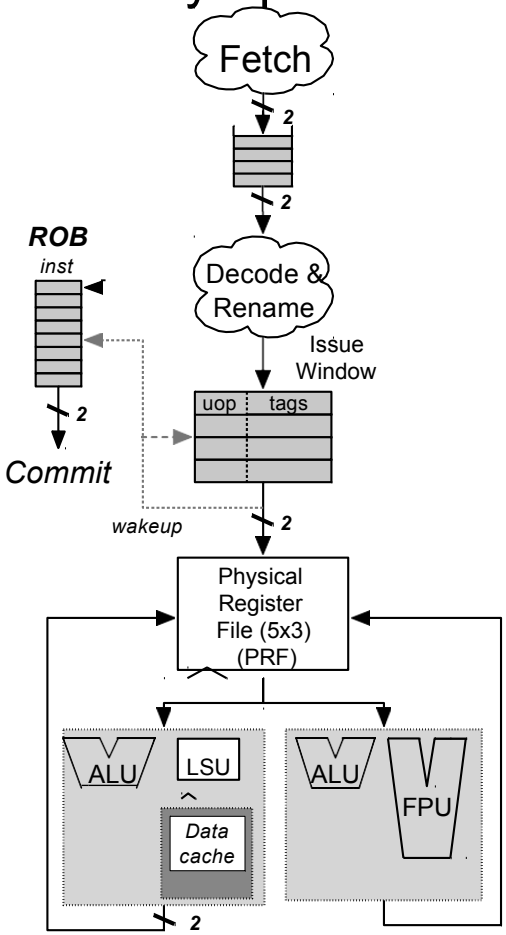
- Abstract FunctionalUnit
  - describes common IO
- Pipelined/Unpipelined
  - handles storing uop metadata, branch resolution, branch kills
- Concrete Subclasses
  - instantiates the actual expert-written FU
  - no modifications required to get FU working with speculative OoO
  - allows easy "stealing" of external code



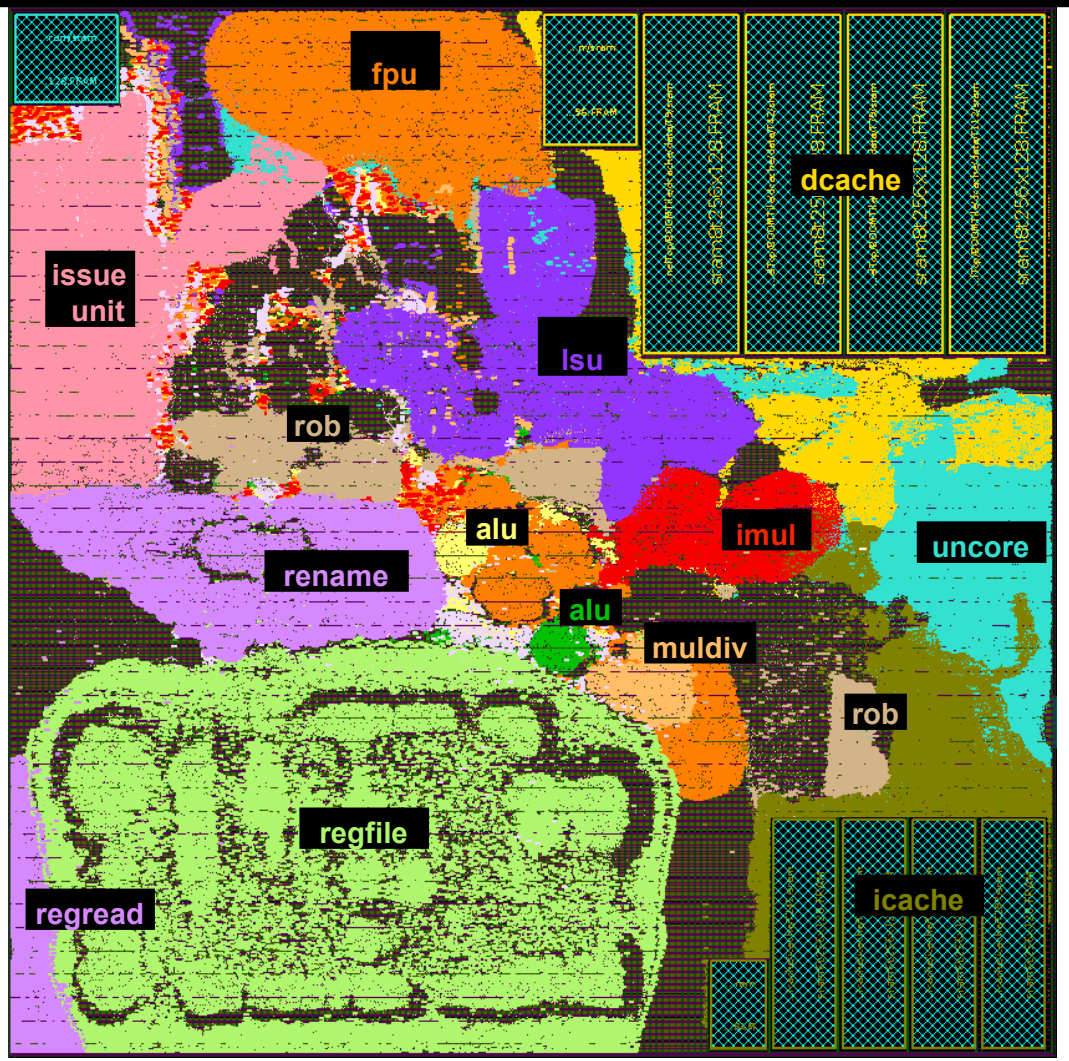
Synthesizable

preliminary results

- Targeting ASIC
- Runs on FPGA
  - Zynq zc706



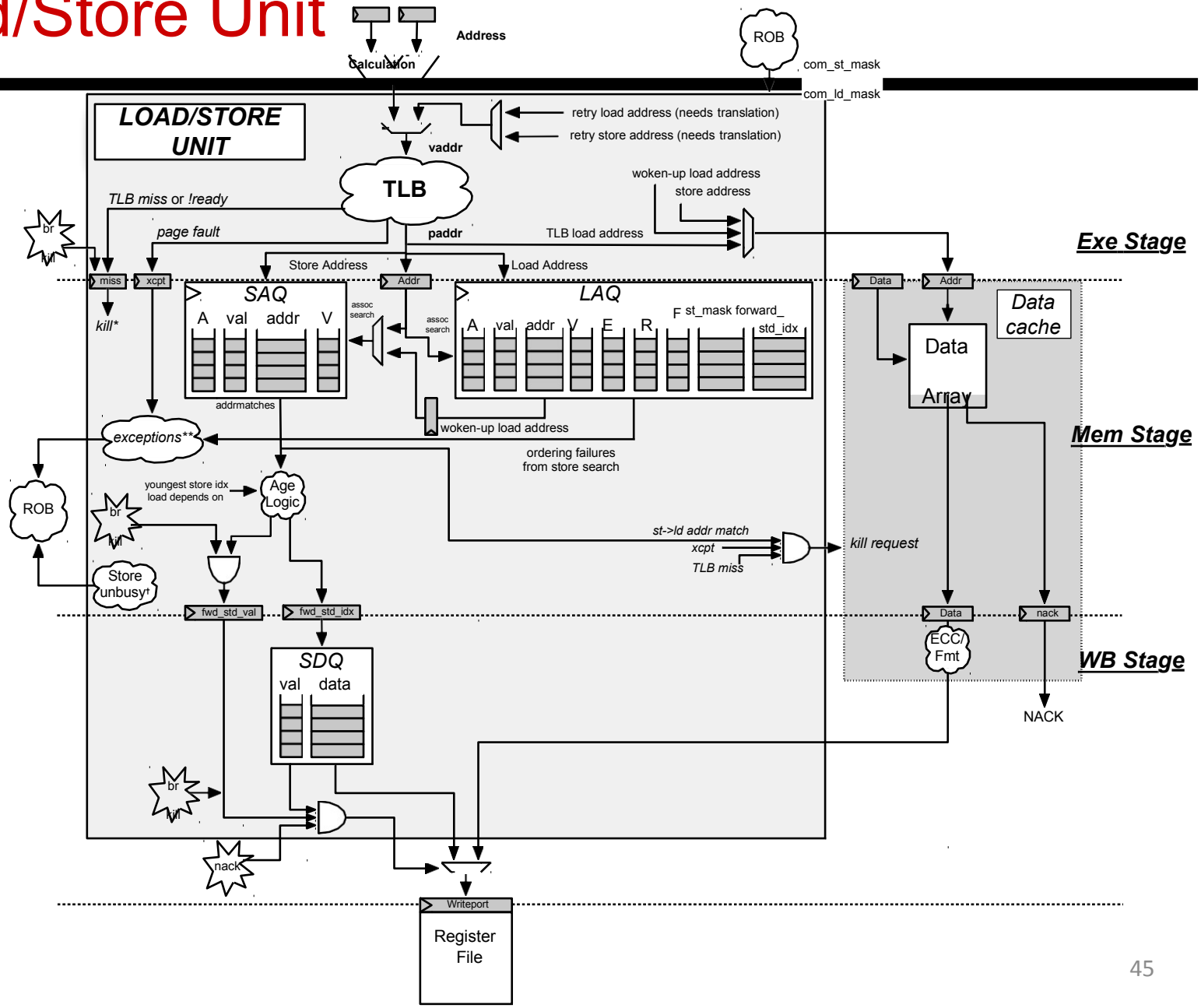
TSMC 45nm floorplan



2-wide BOOM, 16kB L1 caches

1.2 mm²

Load/Store Unit

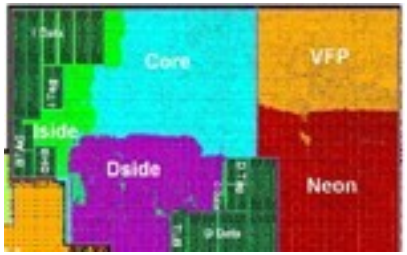


Comparison against ARM

Category	ARM Cortex-A9	RISC-V BOOM-2w
ISA	32-bit ARM v7	64-bit RISC-V v2 (RV64G)
Architecture	2 wide, 3+1 issue Out-of-Order 8-stage	2 wide, 3 issue Out-of-Order 6-stage
Performance	**3.59** CoreMarks/MHz	**3.91** CoreMarks/MHz
Process	TSMC 40GPLUS	TSMC 40GPLUS
Area with 32K caches	~2.5 mm^2	~1.00 mm^2
Area efficiency	**1.4** CoreMarks/MHz/mm^2	**3.9** CoreMarks/MHz/mm^2
Frequency	1.4 GHz	1.5 GHz
Power	**0.5-1.9 W** (2 cores + L2) @ TSMC 40nm, 0.8-2.0 GHz	**0.25 W** (1 core + L1) @ TSMC 45nm, 1 GHz

+9%!

note:
not to scale



Repositories

- BOOM
  - https://github.com/ucb-bar/riscv-boom
  - just the BOOM core code

- Rocket-chip
  - https://github.com/ucb-bar/rocket-chip
  - the rest of the SoC
  - chisel, rocket, uncore, junctions, riscv-tools, fpga-zynq, etc.
  - generates C++ emulator, verilog for FPGA
  - http://riscv.org/tutorial-hpca2015/riscv-rocket-chip-tutorial-bootcamp-hpca2015.pdf (for more information)

BOOM Quick-start

```
$ export ROCKETCHIP_ADDONS="boom"
$  git clone https://github.com/ucb-bar/rocket-chip.git
$ cd rocket-chip
$  git checkout boom
$ git submodule update --init
$ cd riscv-tools
$ git submodule update --init --recursive riscv-tests
$ cd ../emulator; make run CONFIG=BOOMCPPConfig
```

- "BOOM-chip" is (currently) a branch of rocket-chip
- "make run" builds and runs riscv-tests suite
- there are many different CONFIGs available!
  - rocket-chip/src/main/scala/PrivateConfigs.scala
  - rocket-chip/boom/src/main/scala/configs.scala
  - slightly different configs for different targets (CPP, FPGA, VLSI)

How do you verify and debug BOOM?

- parameterization makes this tough!
- tested with:
  - riscv-tests
    - (assembly functional tests + bare-metal micro-benchmarks)
  - CoreMark + riscv-pk
  - SPEC + Linux
  - riscv-torture

riscv-torture

- now open-source!
  - https://github.com/ucb-bar/riscv-torture
- torture generates random tests to stress the core pipeline
- runs test on Spike and your processor
  - architectural register state dumped to memory on program termination
  - diff state
  - if error found, finds smallest version of test that exhibits an error

riscv-torture quick-start

Run Rocket-chip (BOOM-chip?)

```
$ git clone https://github.com/ucb-bar/rocket-chip.git
$ cd rocket--chip
$ git submodule update --init
$ cd riscv--tools
$ git submodule update --init --recursive riscv-tests
$ cd ../emulator; make run CONFIG=BOOMCPPConfig
```

Run Torture

```
$ cd rocket-chip
$ git clone https://github.com/ucb-bar/riscv-torture.git
$ cd riscv-torture
$ git submodule update --init
$ vim Makefile  # change RTL_CONFIG=BOOMCPPConfig
$ make igentest # test that torture works, gen a single test
$ make cnight# run C++ emulator overnight
```

Commit Logging

- BOOM can generate commit logs
  - (priv level, PC, inst, wb raddr, wb data)
- Spike supports generating commit logs!
  - configure Spike with "--enable-commitlog" flag
  - outputs to stderr
- only semi-automated
  - valid differences from Spike and hw
  - e.g., spinning on LR+SC

```
# modify the build.sh in riscv-tools
build_project riscv-isa-sim --prefix=$RISCV --with-fesvr=$RISCV --enable-commitlog
```

What documentation is available?

- A design document is in progress
  - https://github.com/ccelio/riscv-boom-doc
- Wiki
  - https://github.com/ucb-bar/riscv-boom/wiki

Conclusion

- BOOM is RV64G, runs SPEC on Linux on an FPGA
- BOOM is ~10k loc and 4 person-years of work
- excellent platform for prototyping new ideas
- now open-source!