Module 2 Wrap-up and OoO

Colin Schmidt

CS 152 Section 7

3/3/2016

Agenda

- PS2 Review
- Quiz 2 Prep
- Out of Order Execution
- Lab 3 Info

Q1: Caches

- 1.{A,B}: Table questions?
- 1.C: Modeling a cache questions?
- 1.D: Average latency questions?

Q2: Coding for caches

- 2.{A,B,C}: Calculate cache misses questions?
  - Cache access pattern

Q3: New Cache Design

- 3.A: Same cycle time as before questions?
- 3.B: AMAT questions?
- 3.C: 3C’s questions?
- 3.D: Virtual aliasing questions?
- 3.E: Preventing aliasing questions?

Q4 Victim Cache

- 4.A: Access time questions?
- 4.B: Cache behavior questions?
- 4.C: AMAT questions?

Q5 3C’s

- doubling assoc?
- halving line size?
- doubling sets?
- adding prefetching

Q6 Memory Hierachy

- Hit Time
- Miss Rate
- Miss Penalty

Topics

- Caches
  - 3 C's
  - associativity
  - replacement policy
  - write policy
  - access time
  - AMAT
  - Writing good code

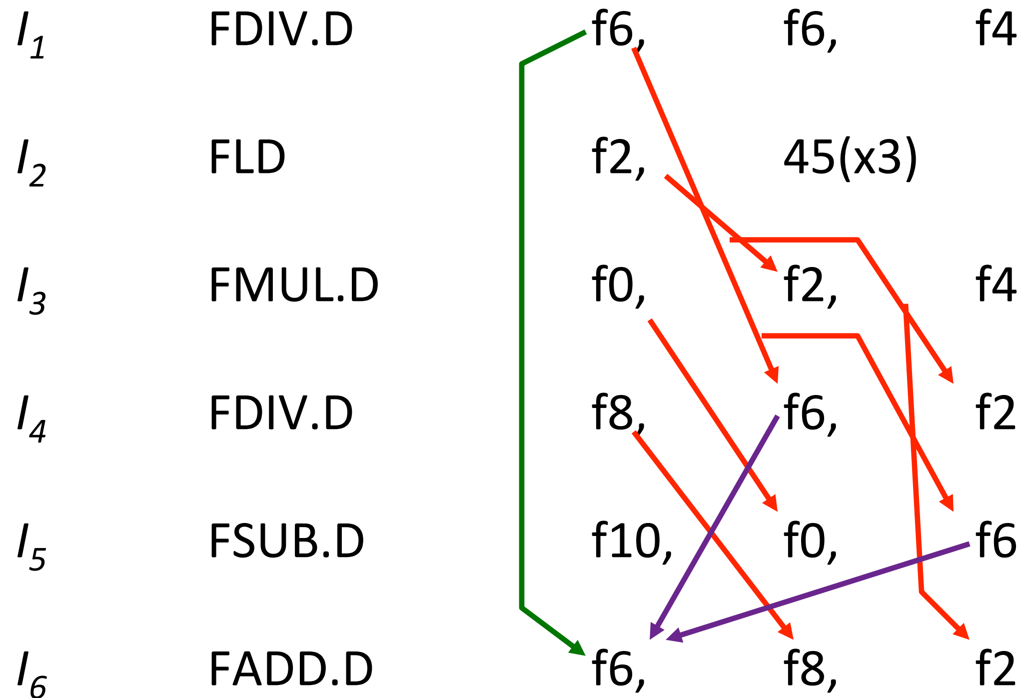
Translation/Protection

- Virtual Memory
- TLB
  - Cache interaction
  - Aliasing
- Page Tables
  - Entries
  - Organization
- Protection

Complex Pipelines

- Adding long latency operations is what causes complications
- Why wasn't this a problem with loads
- How do we prevent this?

Data Hazards: An Example



RAW Hazards
WAR Hazards
WAW Hazards

Scoreboard for In-order Issues

Busy[FU#] : a bit-vector to indicate FU's availability.
(FU = Int, Add, Mult, Div)
These bits are hardwired to FU's.

WP[reg#] : a bit-vector to record the registers for which writes are pending.
These bits are set by Issue stage and cleared by WB stage

Issue checks the instruction (opcode dest src1 src2) against the scoreboard (Busy & WP) to dispatch

FU available?	Busy[FU#]
RAW?	WP[src1] or WP[src2]
WAR?	*cannot arise*
WAW?	WP[dest]

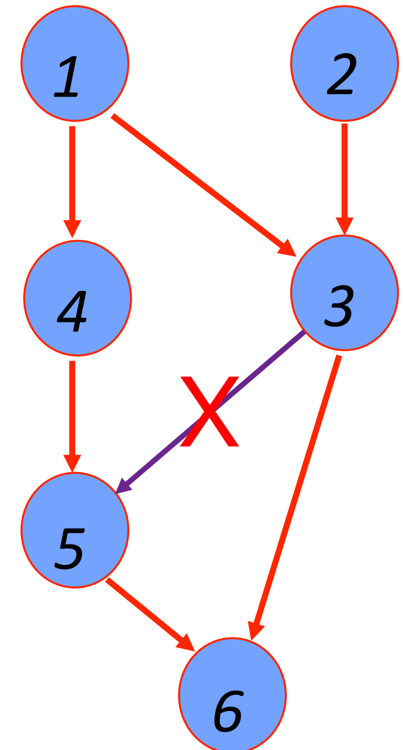
Scoreboard Dynamics

	Functional Unit Status										Registers Reserved for Writes	
	Int(1)	Add(1)	Mult(3)			Div(4)				WB		
t0	I_1					f6					f6	
t1	I_2 f2						f6				f6, f2	
t2								f6		f2	f6, f2	I_2
t3	I_3		f0						f6		f6, f0	
t4				f0						f6	f6, f0	I_1
t5	I_4				f0	f8					f0, f8	
t6							f8			f0	f0, f8	I_3
t7	I_5	f10						f8			f8, f10	
t8									f8	f10	f8, f10	I_5
t9										f8	f8	I_4
t10	I_6	f6									f6	
t11										f6	f6	I_6

I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(x3)	
I_3	FMULT.D	f0,	f2,	f4
I_4	FDIV.D	f8,	f6,	f2
I_5	FSUB.D	f10,	f0,	f6
I_6	FADD.D	f6,	f8,	f2

Issue Limitations: In-Order and Out-of-Order

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

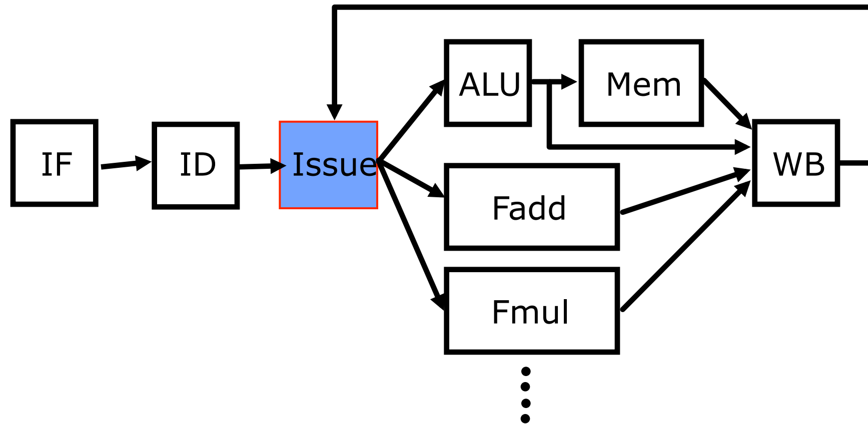


In-order: 1 (2,1) . . . . 2 3 4 4 3 5 . . . 5 6 6

Out-of-order: 1 (2,1) 4 4 5 . . . 2 (3,5) 3 6 6

Any antidependence can be eliminated by renaming.
(renaming => additional storage)
Can it be done in hardware? yes!

Register Renaming



- Decode does register renaming and adds instructions to the issue-stage instruction reorder buffer (ROB)

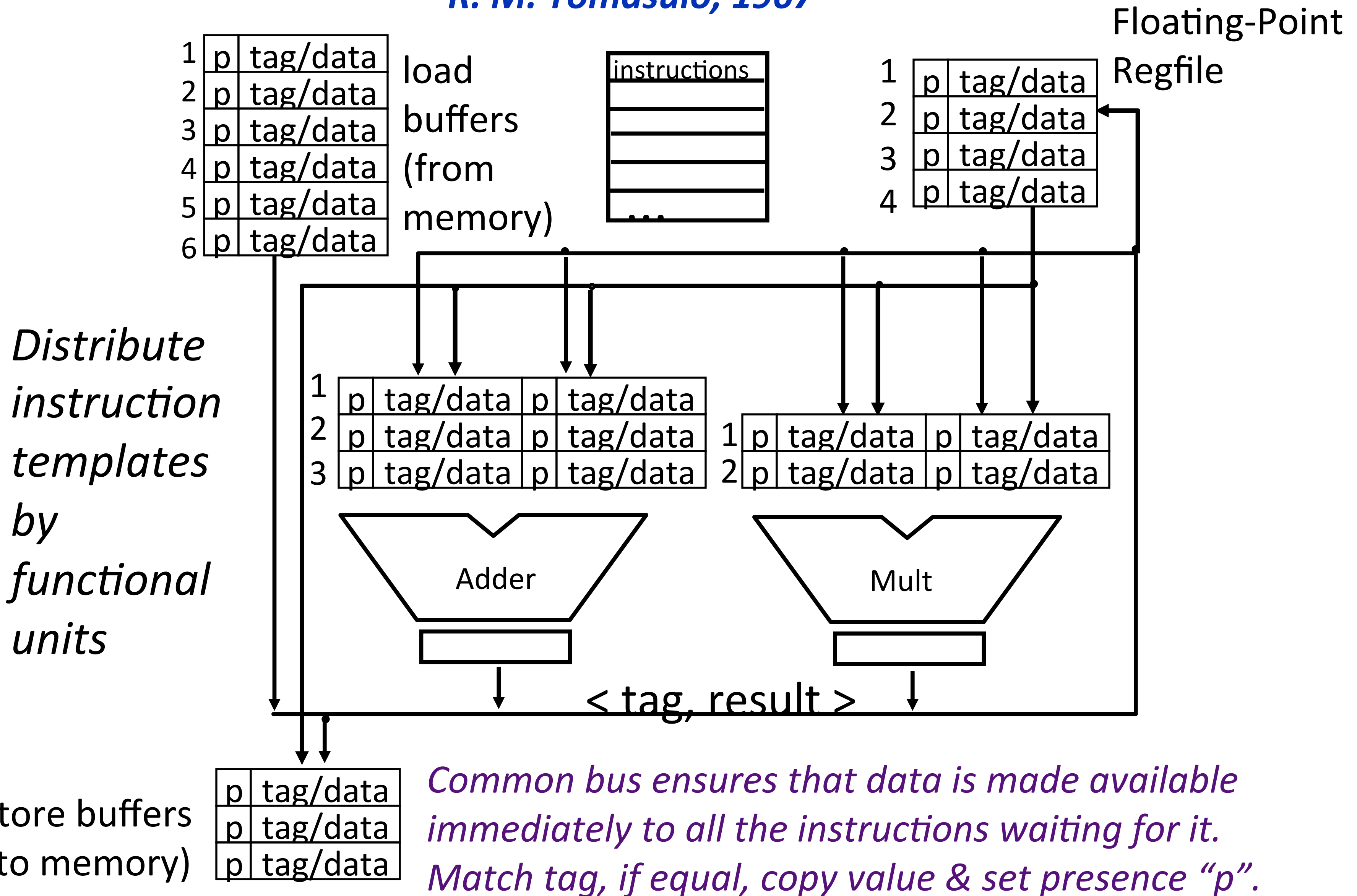
  ⇒ renaming makes WAR or WAW hazards impossible

- Any instruction in ROB whose RAW hazards have been satisfied can be issued.

  ⇒ Out-of-order or dataflow execution

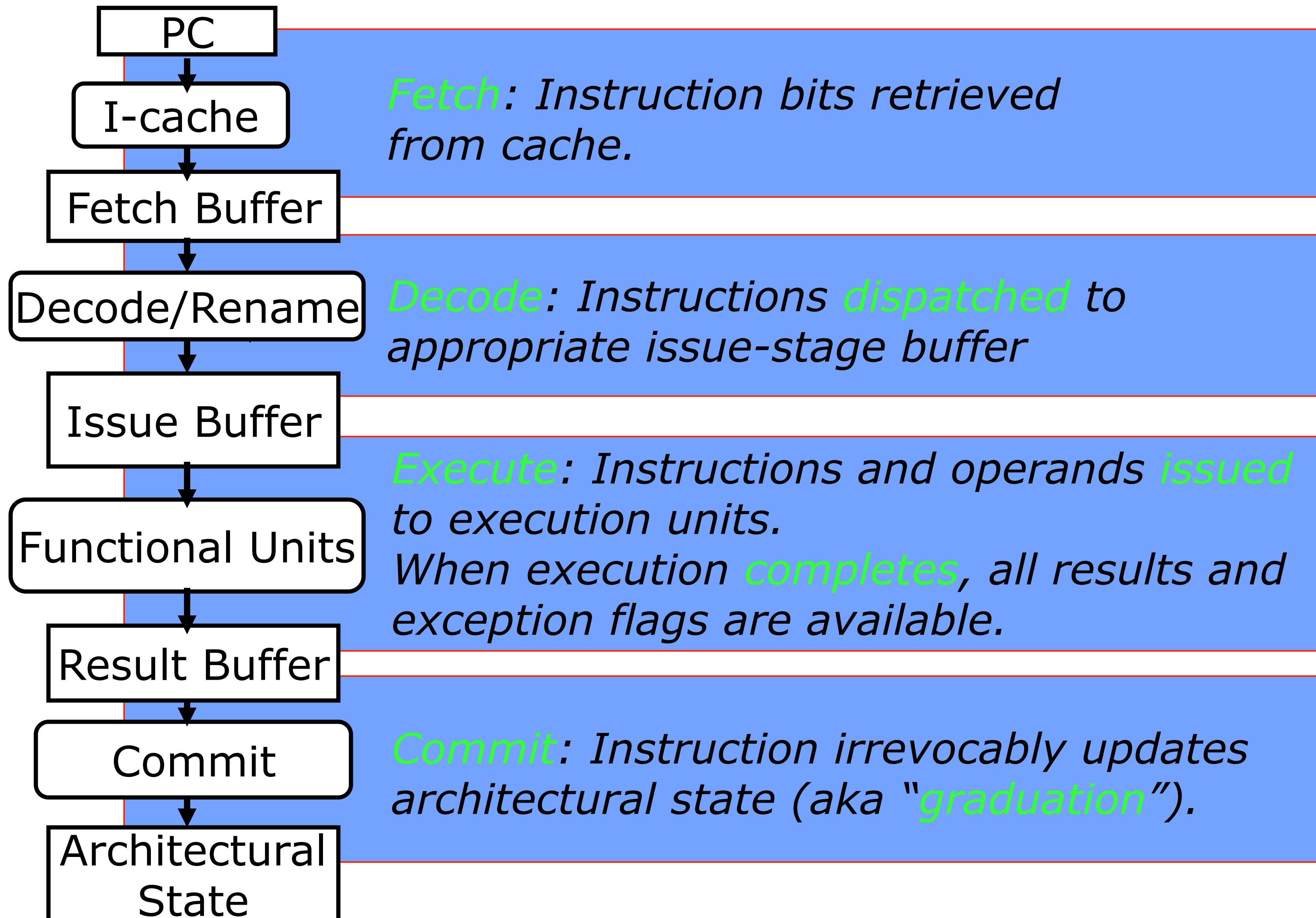
IBM 360/91 Floating-Point Unit

R. M. Tomasulo, 1967



Common bus ensures that data is made available immediately to all the instructions waiting for it. Match tag, if equal, copy value & set presence "p".

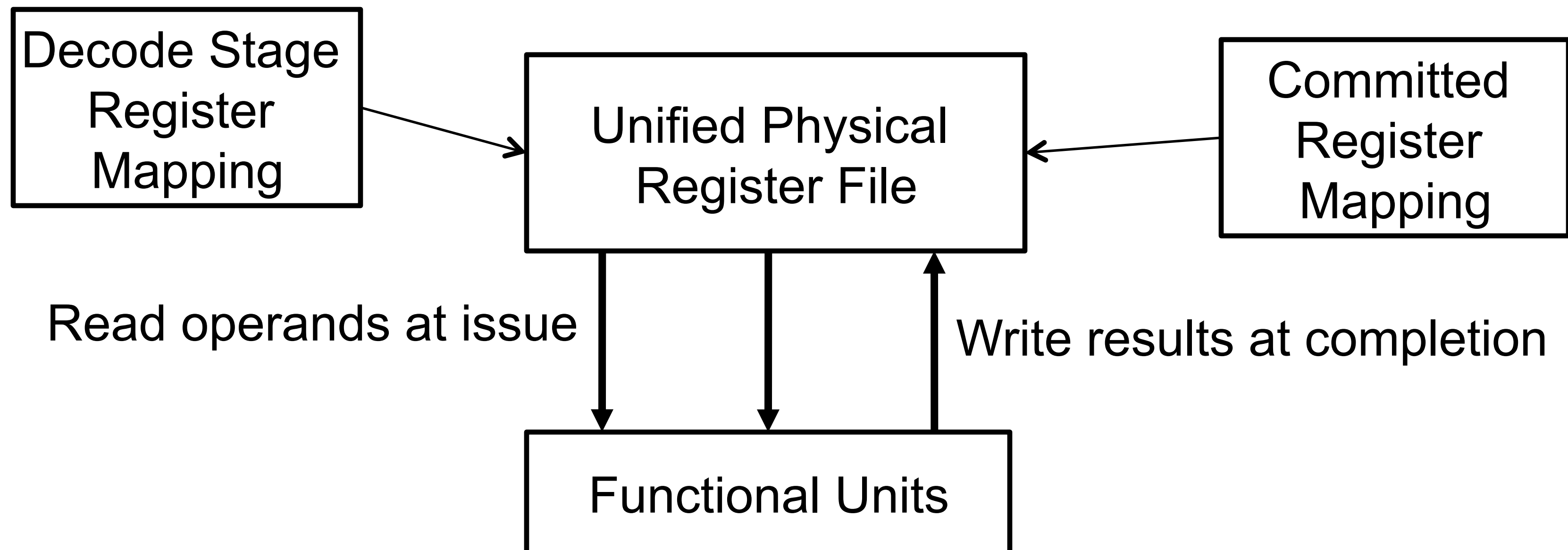
Phases of Instruction Execution



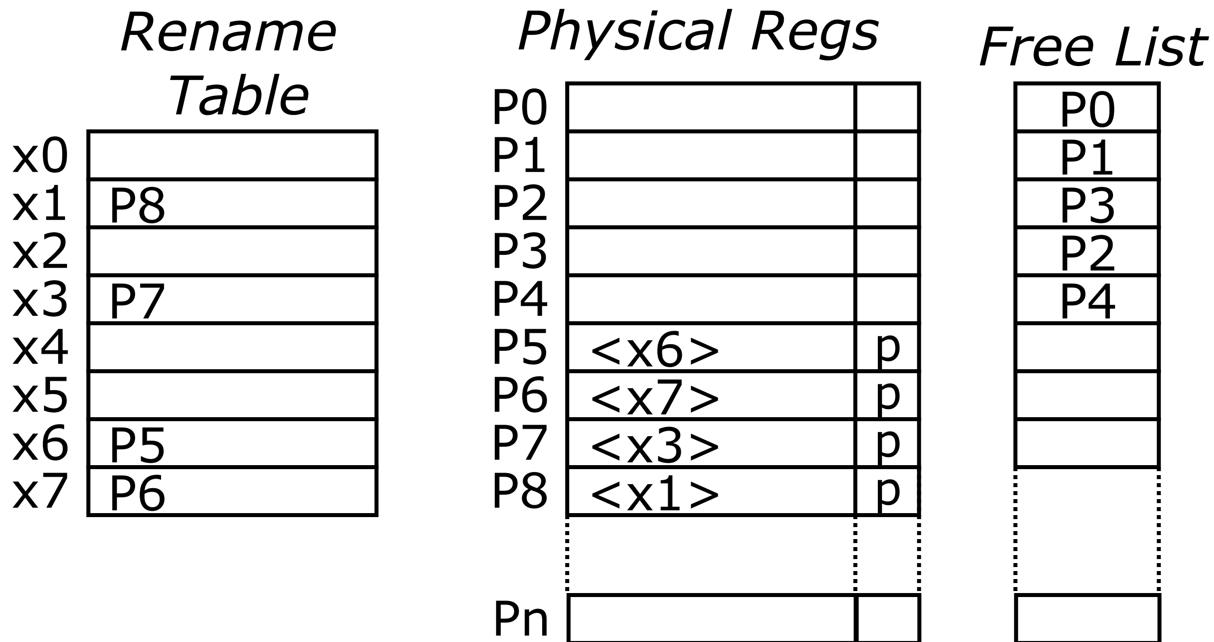
Unified Physical Register File

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

- Rename all architectural registers into a single *physical* register file during decode, no register values read
  - x1 -> P1
- 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



Physical Register Management




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

ROB

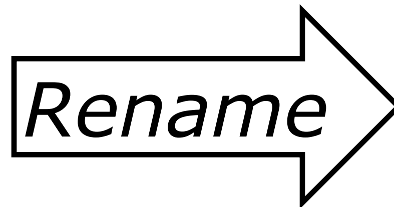
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

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

Lifetime of Physical Registers

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

```
ld x1, (x3)              ld P1, (Px)
addi x3, x1, #4          addi P2, P1, #4
sub x6, x7, x9           sub P3, Py, Pz
add x3, x3, x6           add P4, P2, P3
ld x6, (x1)              ld P5, (P1)
add x6, x6, x3           add P6, P5, P4
sd x6, (x1)              sd P6, (P1)
ld x6, (x11)             ld P7, (Pw)
```



When can we reuse a physical register?

When next write of same architectural register commits

Lab 3

- Not ready yet…
- Later this week/weekend
- Info at http://ccelio.github.io/riscv-boom-doc/

Questions