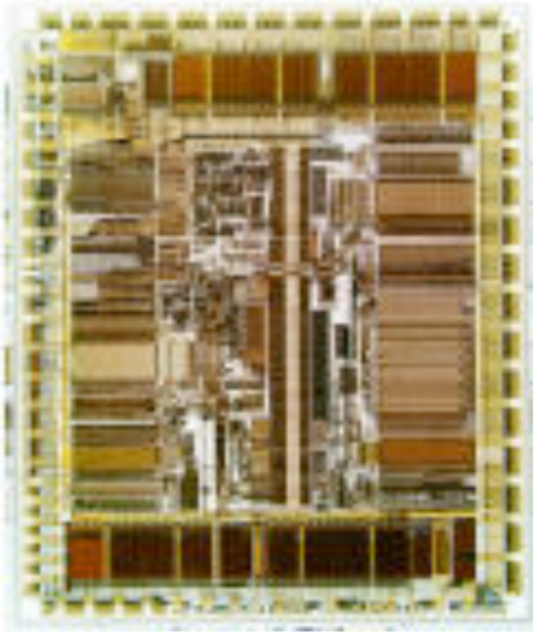
Lecture 14: Instruction Level Parallelism

- Last time
  - Pipelining in the real world
    - Control hazards
    - Other pipelines
- Today
  - Take QUIZ 10 over P&H 4.10-15, before 11:59pm today
  - Homework 5 due Thursday March 11, 2010
  - Instruction level parallelism
    - Multi-issue (Superscalar) and out-of-order execution

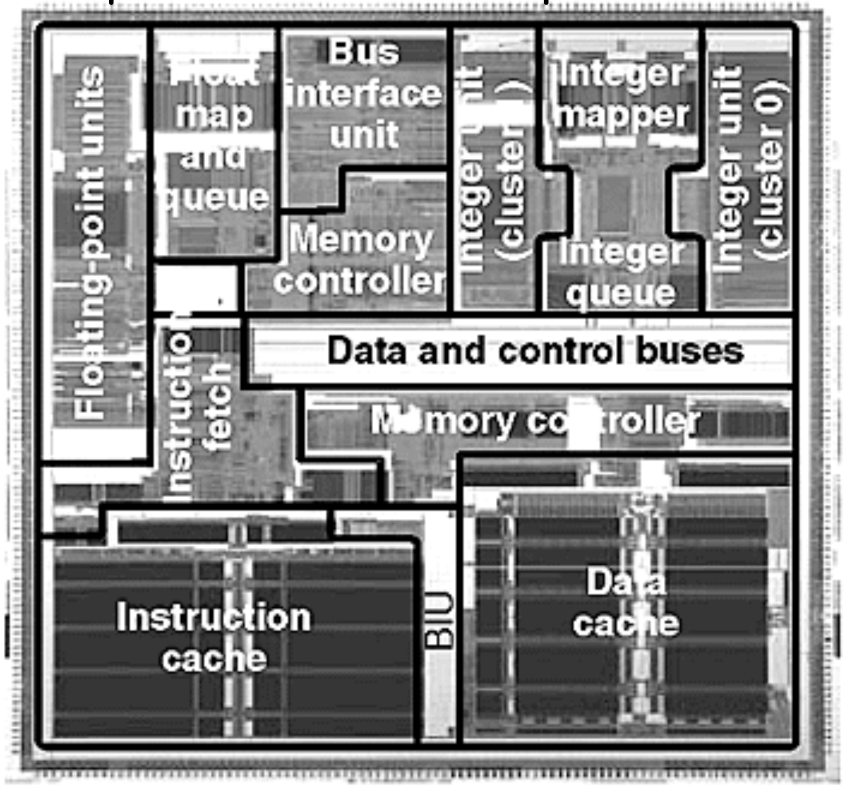
Where Are We?

Pipelined in-order processor
Simple branch prediction
Instruction/data caches (on –chip)



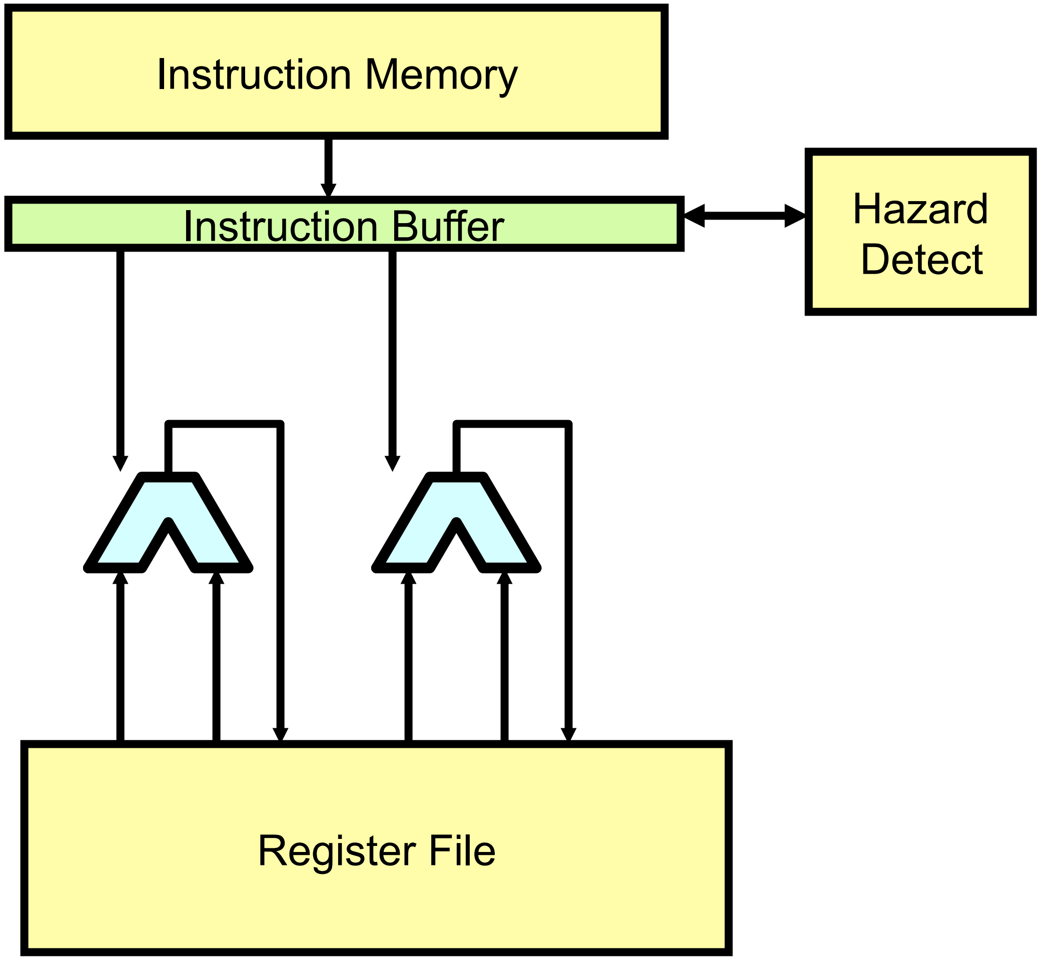
DEC Alpha 21064
Introduced in 1992

Out-of-order instruction execution
"Superscalar"
Sophisticated branch prediction



DEC Alpha 21264
Introduced 1998

Dynamic Multiple Issue (Superscalar)
No instruction reordering, Choose 0, 1 … N



MIPS with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

Hazards in Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet

    -
      ```
      add  $t0, $s0, $s1
      load $s2, 0($t0)
      ```
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

What Hardware Do We Need?

What Hardware Do We Need?

- Wider fetch i-cache bandwidth
- Multiported register file
- More ALUs
- Restrictions on issue of load/stores because N ports to the data cache slows it down too much

Multiple Issue (Details)

- Dependencies and structural hazards checked at run-time
- Can run existing binaries
  - Recompile for performance, not correctness
  - Example - Pentium

- More complex issue logic
  - Swizzle next N instructions into position
  - Check dependencies and resource needs
  - Issue M <= N instructions that can execute in parallel

Example Multiple Issue

Issue rules: at most 1 load/store, at most 1 floating op

Latency: load=1, int=1, float-mult = 2, float-add = 1

```
                                                          cycle
LOOP:    LD      F0, 0(R1)              // a[i]             1
         LD      F2, 0(R2)              // b[i]             2
         MULTD   F8, F0, F2             // a[i] * b[i]      4 (stall)
         ADDD    F12, F8, F16           // + c              5
       { SD      F12, 0(R3)             // d[i]             6
       { ADDI    R1, R1, 4
       { ADDI    R2, R2, 4                                  7
       { ADDI    R3, R3, 4
         ADDI    R4, R4, 1              // increment I      8
         SLT     R5, R4, R6             // i<n-1            9
         BNEQ    R5, R0, LOOP                               10
```

Old CPI = 12/11 = 1.09
New CPI = 10/11 = 0.91

Rescheduled for Multiple Issue

Issue rules: at most 1 LD/ST, at most 1 floating op

Latency: LD - 1, int-1, F*-2, F+-1

```
                                                             cycle
LOOP: LD     F0, 0(R1)              // a[i]                    1
      ADDI   R1, R1, 4
      LD     F2, 0(R2)              // b[i]                    2
      ADDI   R2, R2, 4
      MULTD  F8, F0, F2             // a[i] * b[i]             4
      ADDI    R4, R4, 1            // increment I
      ADDD   F12, F8, F16           // + c                     5
      SLT    R5, R4, R6             // i<n-1
      SD     F12, 0(R3)             // d[i]                    6
      ADDI    R3, R3, 4
      BNEQ   R5, R0, LOOP                                      7
```

Old CPI = 0.91
New CPI = 7/11 = 0.64

Given a two way issue processor, what's the best possible CPI? IPC?

The Problem with Static Scheduling

- In-order execution
  - an unexpected long latency blocks ready instructions from executing
  - binaries need to be rescheduled for each new implementation
  - small number of *named* registers becomes a bottleneck

```
LW    R1, C     //miss 50 cycles
LW    R2, D
MUL   R3, R1, R2
SW    R3, C
LW    R4, B     //ready
ADD   R5, R4, R9
SW    R5, A
LW    R6, F
LW    R7, G
ADD   R8, R6, R7
SW    R8, E
```

Dynamic Scheduling

- Determine execution order of instructions at *run time*
- Schedule with knowledge of run-time variable latency
  - cache misses
- Compatibility advantages
  - avoid need to recompile old binaries
  - avoid bottleneck of small *named* register sets
    - but still need to deal with spills
- Significant hardware complexity

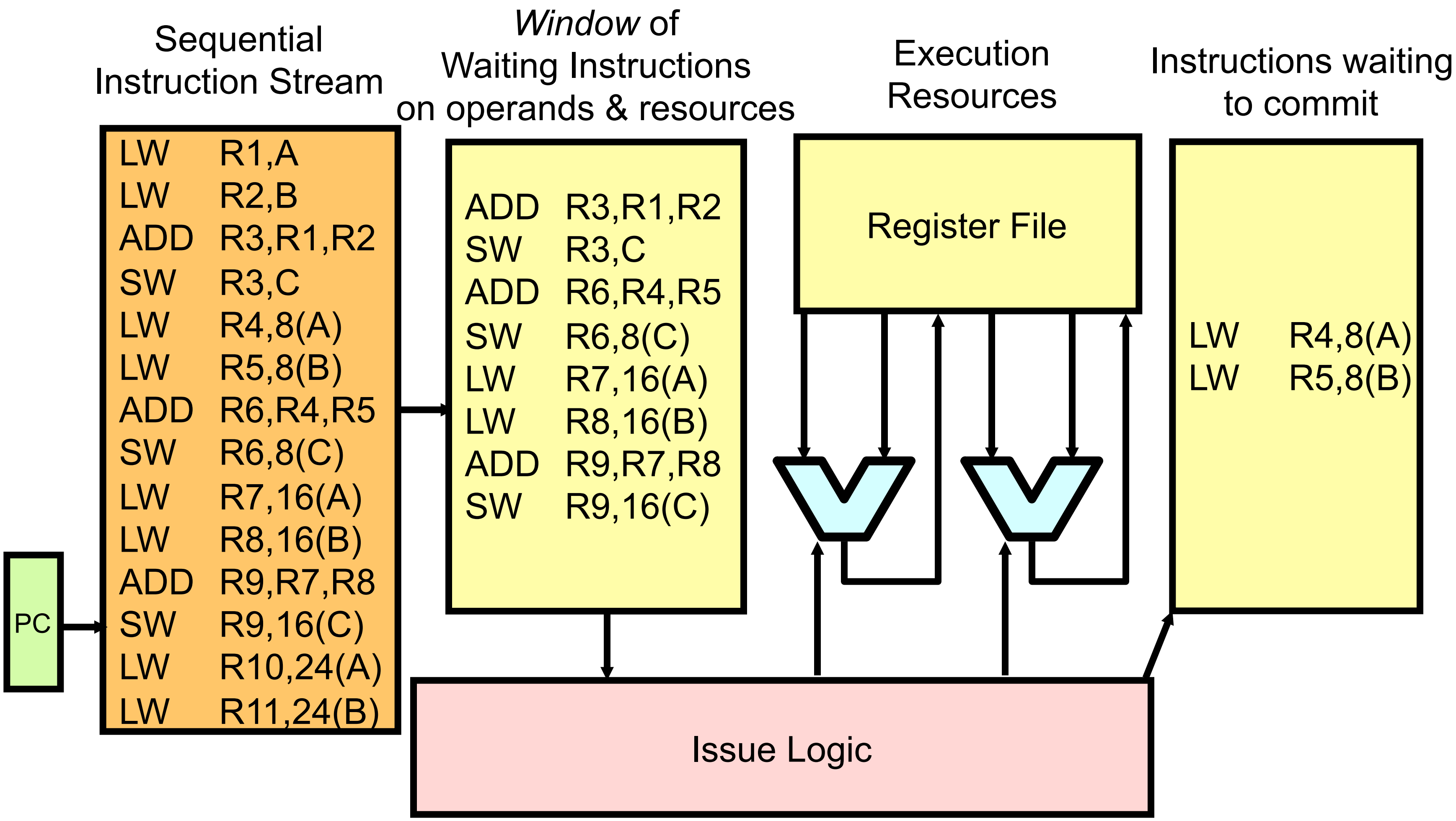
Example

Top = without dynamic scheduling, Bottom = with dynamic scheduling

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
LD R1,A	I	M	X	X	X	X	X	X	X	X	X	C																		
LD R2,B		I	C																											
MUL R3,R1,R2												I	X	X	C															
LD R4,C													I	M	X	X	X	X	X	X	X	X	X	C						
LD R5,D														I	C															
MUL R6,R5,R4																								I	X	X	C			
ADD R7,R3,R6																											I	X	C	
SD R7,E																													I	C
LD R1,A	I	M	X	X	X	X	X	X	X	X	X	C																		
LD R2,B		I	C																											
MUL R3,R1,R2												I	X	X	C															
LD R4,C			I	M	X	X	X	X	X	X	X	X	X	C																
LD R5,D				I	C																									
MUL R6,R5,R4														I	X	X	C													
ADD R7,R3,R6																	I	X	C											
SD R7,E																			I	C										

- **10 cycle data memory (cache) miss**
- **3 cycle MUL latency**
- **2 cycle add latency**

Dynamic Scheduling
Basic Concept



Sequential Instruction Stream

```
LW   R1,A
LW   R2,B
ADD  R3,R1,R2
SW   R3,C
LW   R4,8(A)
LW   R5,8(B)
ADD  R6,R4,R5
SW   R6,8(C)
LW   R7,16(A)
LW   R8,16(B)
ADD  R9,R7,R8
SW   R9,16(C)
LW   R10,24(A)
LW   R11,24(B)
```

PC

Window of Waiting Instructions on operands & resources

```
ADD  R3,R1,R2
SW   R3,C
ADD  R6,R4,R5
SW   R6,8(C)
LW   R7,16(A)
LW   R8,16(B)
ADD  R9,R7,R8
SW   R9,16(C)
```

Execution Resources

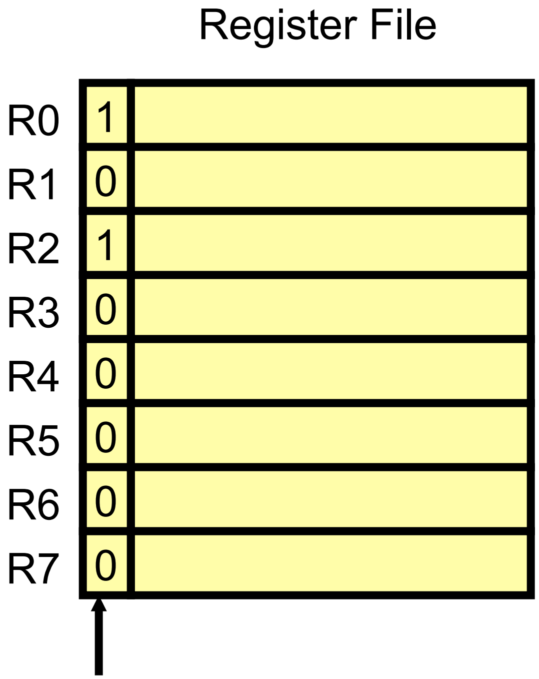
Register File

Instructions waiting to commit

```
LW   R4,8(A)
LW   R5,8(B)
```

Issue Logic

Implementation I - Register Scoreboard



Register	Valid bit
R0	1
R1	0
R2	1
R3	0
R4	0
R5	0
R6	0
R7	0

valid bit
(= 0 if write “pending”)

- Tracks register writes
  - busy = pending write
- Detect hazards for scheduler

```
ADD R3,R1,R2
```

- **Wait until R1 is *valid***
- **Mark R3 valid when complete**

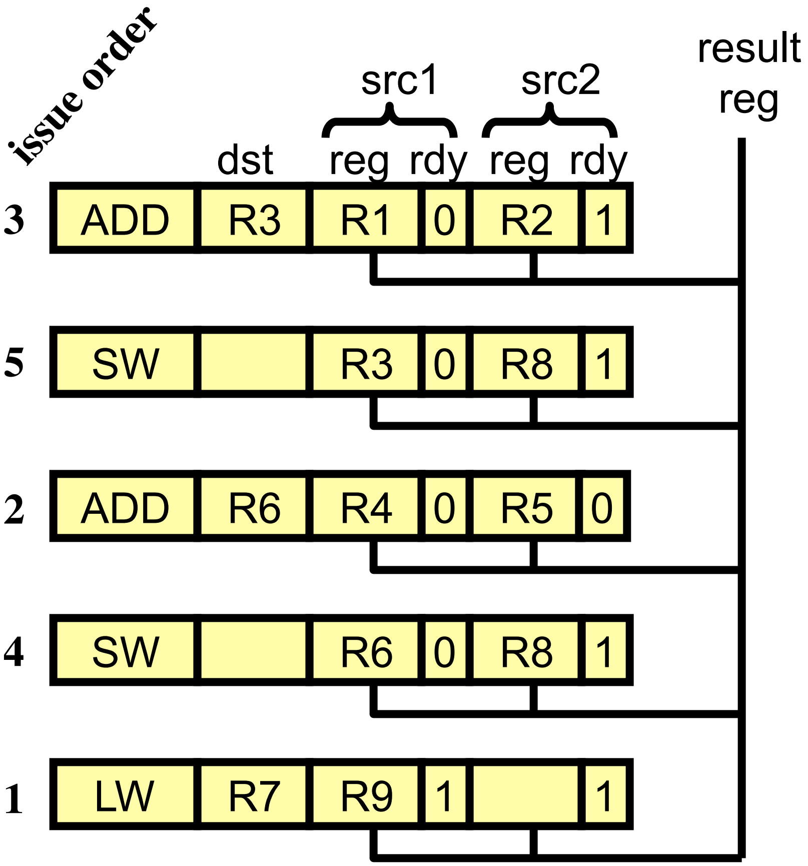
```
SUB R4,R0,R3
```

- **Wait for R3**

What about:

```
LD  R3,(0)R7
ADD R4,R3,R5
LD  R3,(4)R7
```

Implementing A Simple Instruction Window



			src1		src2	
issue order		dst	reg	rdy	reg	rdy
3	ADD	R3	R1	0	R2	1
5	SW		R3	0	R8	1
2	ADD	R6	R4	0	R5	0
4	SW		R6	0	R8	1
1	LW	R7	R9	1		1

```
ADD     R3,R1,R2
SW      R3,0(R8)
ADD     R6,R4,R5
SW      R6,8(R8)
LW      R7,16(R9)
```

R0	1	
R1	0	
R2	1	
R3	0	
R4	0	
R5	0	
R6	0	
R7	0	

Often called *reservation stations*
reg = name, value

Result sequence: R4, R7, R5, R1, R6, R3

Instruction Window Policies

- Add an instruction to the window
  - only when dest register is not busy
  - mark destination register busy
  - check status of source registers and set ready bits
- When each result is generated
  - compare dest register field to all waiting instruction source register fields
  - update ready bits
  - mark dest register not busy
- Issue an instruction when
  - execution resource is available
  - all source operands are ready
- Result
  - issues instructions out of order as soon as source registers are available
  - allows only one operation in the window per destination register

Register Renaming (1)

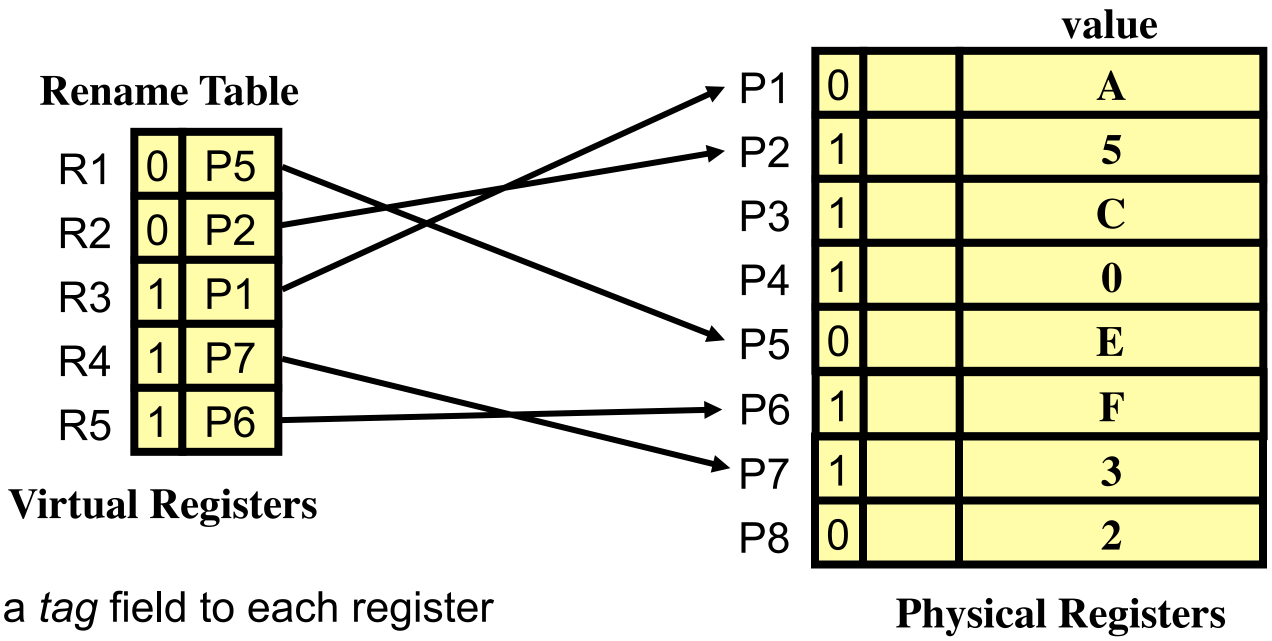
What about this sequence?

```
LW   R1, 0(R4)
ADD  R2, R1, R3
LW   R1, 4(R4)
ADD  R5, R1, R3
```

Can't add 3 to the window since R1 is already busy

Need 2 R1s!

Register Renaming (2)



Rename Table

R1	0	P5
R2	0	P2
R3	1	P1
R4	1	P7
R5	1	P6

Virtual Registers

			value
P1	0		A
P2	1		5
P3	1		C
P4	1		0
P5	0		E
P6	1		F
P7	1		3
P8	0		2

Physical Registers

Add a *tag* field to each register
- translates from virtual to physical register name

In window

LW R1, 0(R4)
ADD R2, R1, R3

Next instruction

LW R1, 4(R4)

Register Renaming (3)

S1	LW	P5	P7	1		1
S2	ADD	P2	P5	0	P1	1
S3	LW	P4	P7	1		1
S4	ADD	P6	P4	0	P1	1

	Before			After	
R1	0	P5	R1	0	P4
R2	0	P2	R2	0	P2
R3	1	P1	R3	1	P1
R4	1	P7	R4	1	P7
R5	1	P6	R5	0	P6

When result is generated:

compare *tag* of result to not-ready source fields

grab data if match

Add instruction to window even if dest register is busy

When adding instruction to window

read data of non-busy source registers and retain

read tags of busy source registers and retain

write tag of destination register with slot number

```
LW    R1,0(R4)
ADD   R2,R1,R3
LW    R1,4(R4)
ADD   R5,R1,R3
```

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

Summary

- Summary
  - Pipelining is simple, but a correct high performance implementation is complex
  - Dynamic multiple issue
  - Static multiple issue (VLIW)
  - Out-of-order execution – dependencies, renaming, etc.
- Next Time
  - Caches (new topic!)
  - Homework 5 due Thursday March 11, 2010
  - Read: P&H 5.1–5