Lecture 19: Instruction Level Parallelism

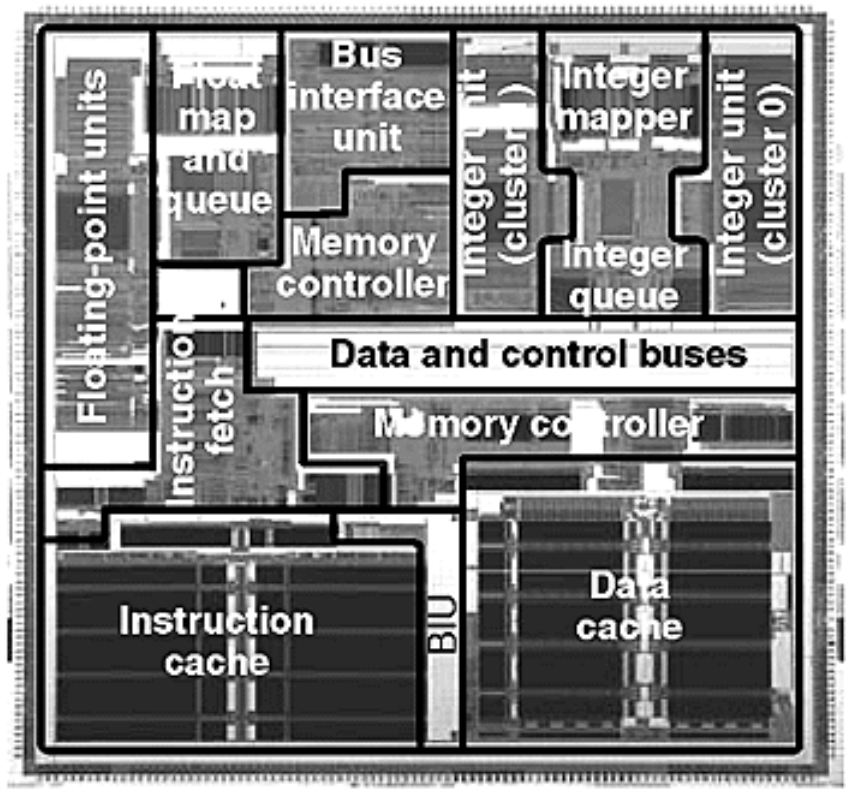
- Administrative:
  - Homework #5 due
  - Homework #6 handed out today
- Last Time:
  - DRAM organization and implementation
- Today
  - Static and Dynamic ILP
  - Instruction windows
  - Register renaming

Where Are We?

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

DEC Alpha 21064

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



DEC Alpha 21264

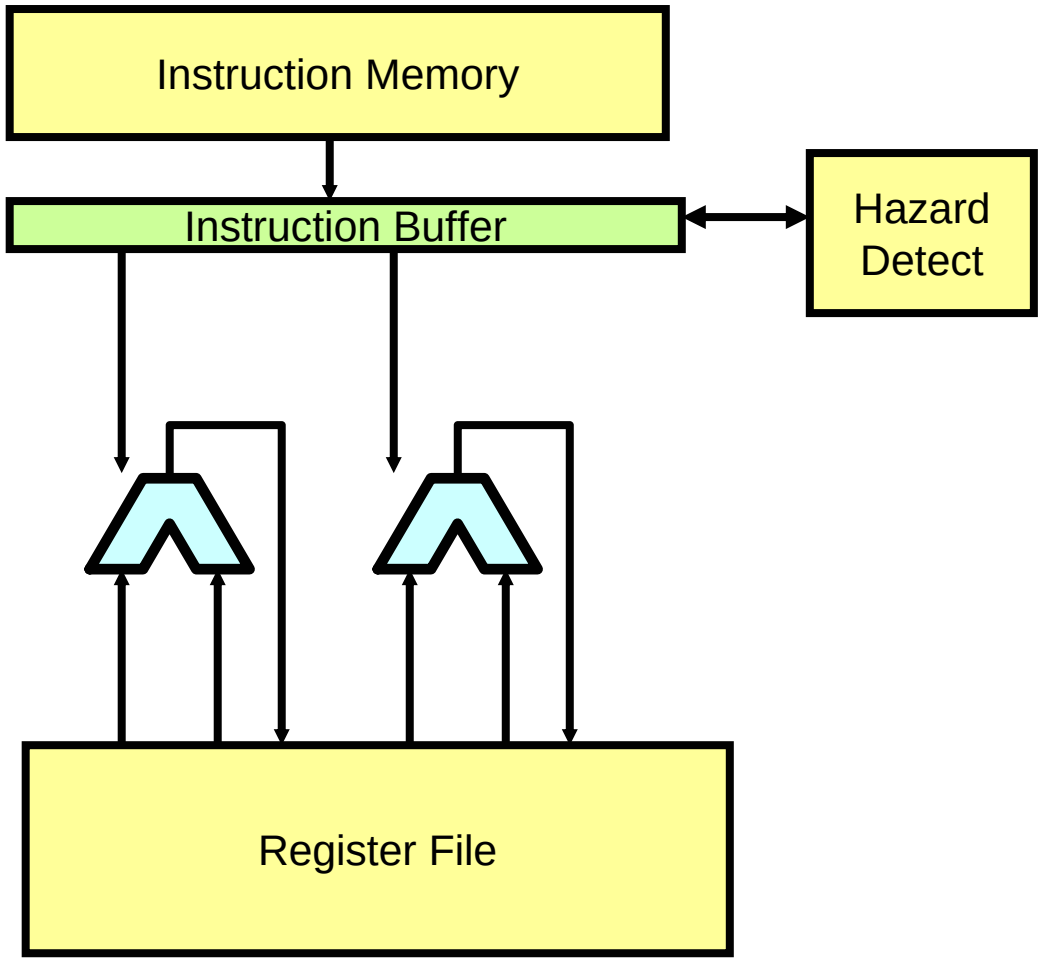
How Do We Speed up the Pipeline?

- Instruction Level Parallelism (ILP)
  - Multi-issue (in-order execution to multiple pipes)
  - Dynamic scheduling ("Superscalar")
  - Compiler schedules ILP (VLIW/EPIC)
- Undetermined dependencies at compile time ⇒ dynamic scheduling
  - Object code compatibility
  - Simplify compiler
- WAR/WAW hazards ⇒ register renaming

```
ADD R1,R2,R3      ⇒     ADD R1,R2,R3
SUB R1,R4,R5            SUB R1',R4,R5
```

- Too many branches ⇒ better branch prediction
  - Or use predication to eliminate branches
- Unknown dependencies (control/data) ⇒ speculate

Multiple Issue – No instruction reordering



Multiple Issue (Details)

- Dependencies and structural hazards checked at run-time
- Can run existing binaries
  - Recompiler 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 = 1, 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*-1, 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

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

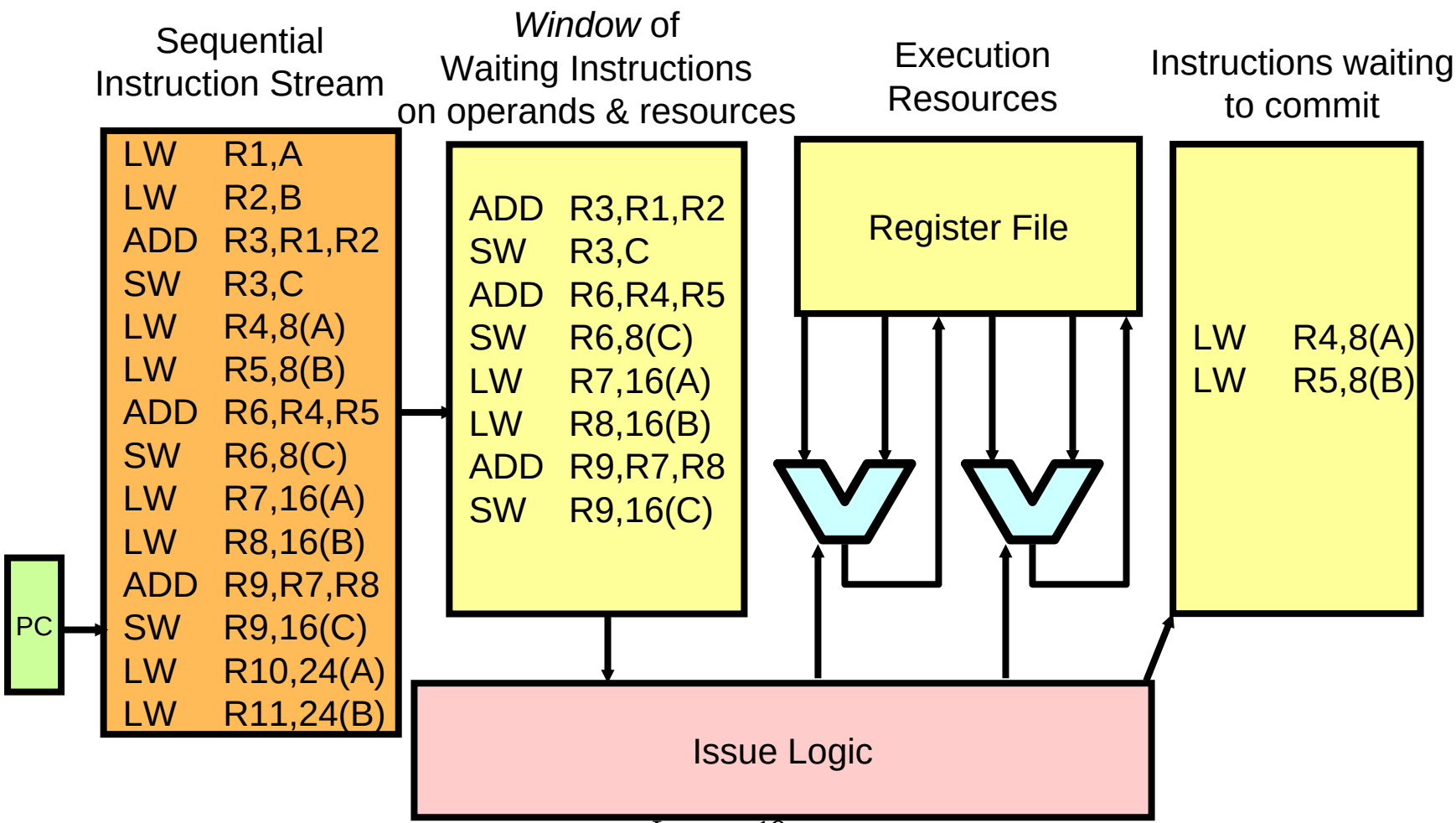
Example

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**

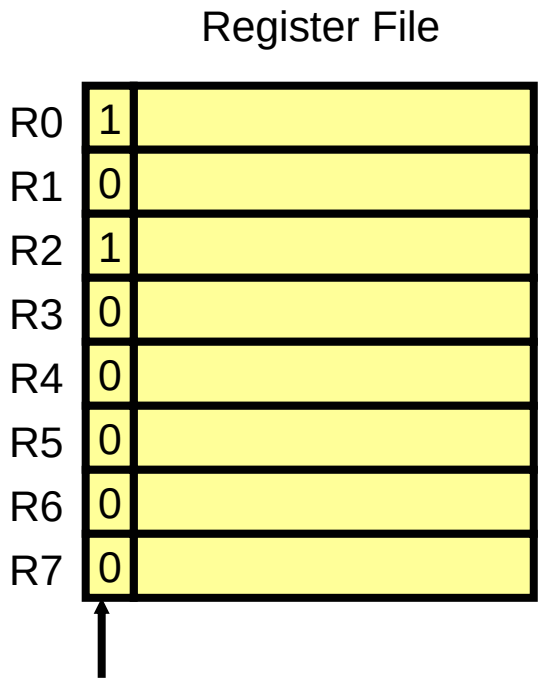
Dynamic Scheduling Basic Concept



Implementation Issues

- Instruction fetch
  - fixed number of instruction window *slots* (e.g., 32)
    - generic
    - partitioned over execution units
  - fetch next sequential instruction whenever a slot is free
- Instruction decode
  - mark input and output registers *busy*
  - slots monitor register status and execution unit reservation tables
- Instruction Issue
  - all input operands available
  - output operand (register) not *busy* (WAW, WAR) due to earlier instruction
  - execution unit is available
- Instruction commit
  - all previous instructions have committed
  - why?

Implementation I - Register Scoreboard



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

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

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

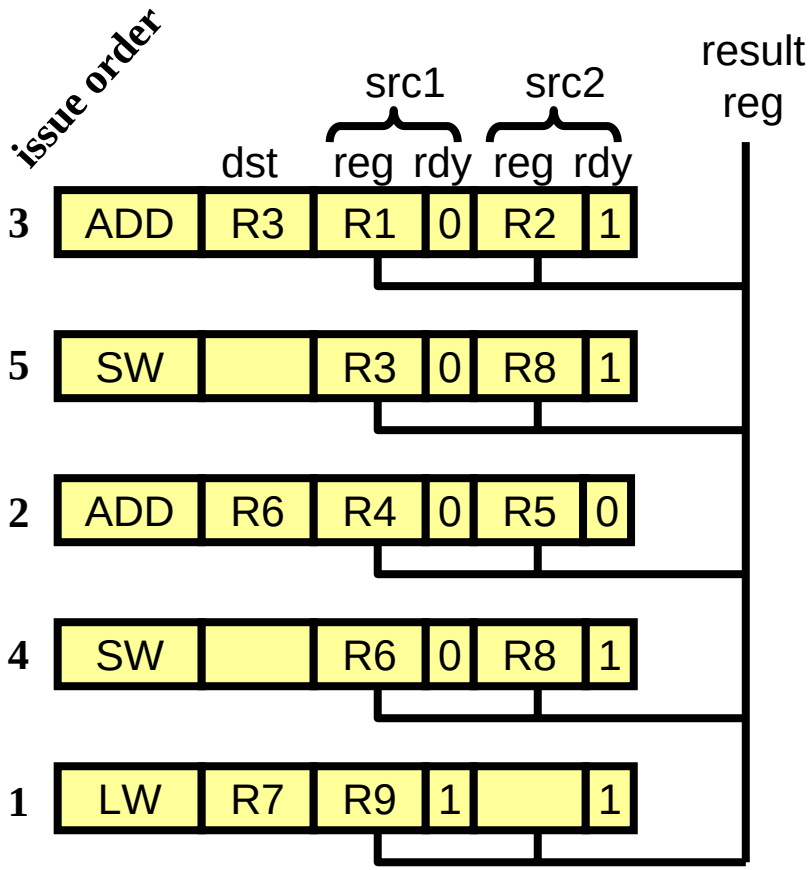
```
SUB R4,R0,R3
```

- **Wait for R3**

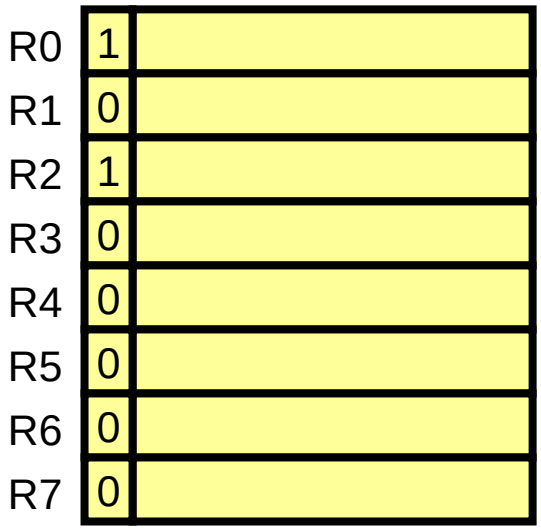
What about:

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

Implementing A Simple Instruction Window




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



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)

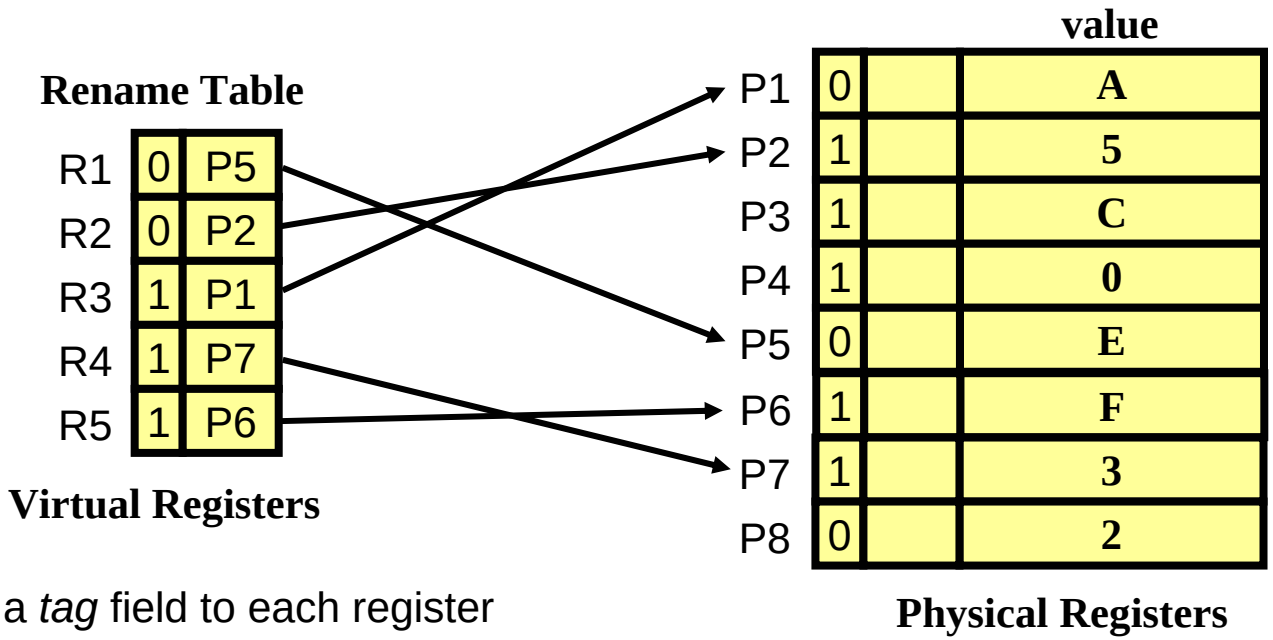
What about this sequence?

```
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)



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	0	P4
R2	0	P2	0	P2
R3	1	P1	1	P1
R4	1	P7	1	P7
R5	1	P6	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
```

Summary

- Today:
  - Basics of out-of-order instruction execution
    - Hardware to keep track of dependencies
    - Mechanisms to reduce false dependences (WAR, WAW)
- Next Time
  - Examples and more details