18-747 Lecture 5: Register Renaming and Dataflow

James C. Hoe
Dept of ECE, CMU
September 12 , 2001

Reading Assignments: S&L Ch3 pp57-82, MJ Ch6, today's handout

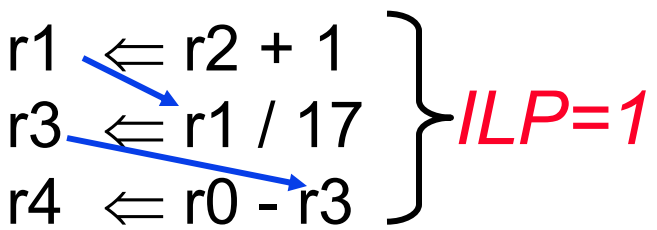
Announcements: Problem Set 1 and Project 0 due 2:30pm, 9/21

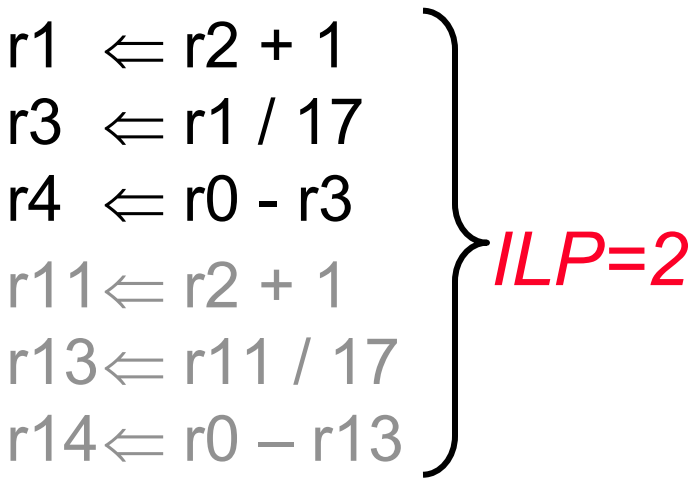
Handouts: *"The Metaflow Architecture"*

How to Increase Performance?

- Pipelining reduces cycle time
- Superscalar increases IPC (instruction per cycle)
- Both schemes need to find lots of ILP in the program

  Must simultaneously consider increasing number of instructions and allow increasing degree of not only parallel but also out-of-ordered operations





Flashback

- Scalar Pipeline
  - Inorder issue (RF read), inorder execute, inorder completion
  - Stall on RAW and Forwarding on RAW
- Diversified Pipelines: *Simple Scoreboard*
  - Inorder dispatch/issue, out-of-order completion
  - Stall dispatch on {structural, RAW, WAW}, *How about forwarding?*
- Diversified Pipelines: *Full Scoreboard*
  - Inorder dispatch, out-of-order issue, out-of-order completion
  - Stall dispatch on {structural, WAW}, stall issue on RAW, stall completion on WAR

Lessons:

- More out-of-orderness ⇒ more ILP ⇒ more hazards

- Don't forget compiler analysis and scheduling

- Stall is a generic technique to ensure sequencing

- RAW stall is a fundamental requirement (for now)

Limitations of Scoreboarding

- Consider a scoreboard processor with infinitely wide datapath

  In the best case, how many instructions can be simultaneously outstanding?

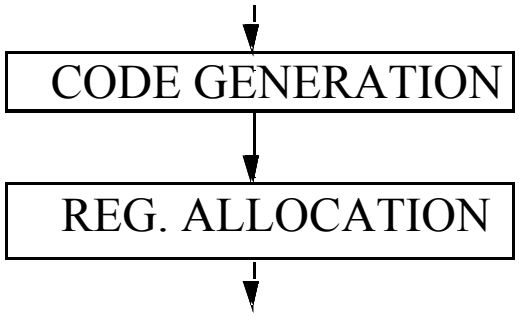
- Hints
  - no structural hazards
  - can always write a RAW-free code sequence

    addi r1,r0,1; addi r2,r0,1; addi r3,r0,1; …….
  - think about x86 ISA with only 8 registers

Contribution to Register Recycling:

Reasons for WAW and WAR

COMPILER REGISTER ALLOCATION



Single Assignment, Symbolic Reg.

Map Symbolic Reg. to Physical Reg.
Maximize Reuse of Reg.

INSTRUCTION LOOPS

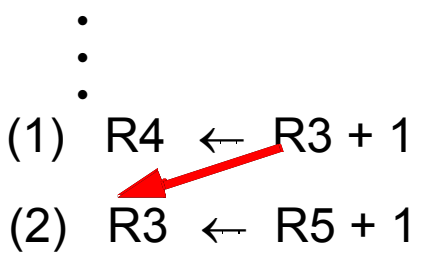
```
9   $34:  mul   $14,  $7,   40
10        addu  $15,  $4,   $14
11        mul   $24,  $9,   4
12        addu  $25,  $15,  $24
13        lw    $11,  0($25)
14        mul   $12,  $9,   40
15        addu  $13,  $5,   $12
16        mul   $14,  $8,   4
17        addu  $15,  $13,  $14
18        lw    $24,  0($15)
19        mul   $25,  $11,  $24
20        addu  $10,  $10,  $25
21        addu  $9,   $9,   1
22        ble   $9,   10,   $34
```

For (k=1;k<= 10; k++)
t += a [i] [k] * b [k] [j] ;

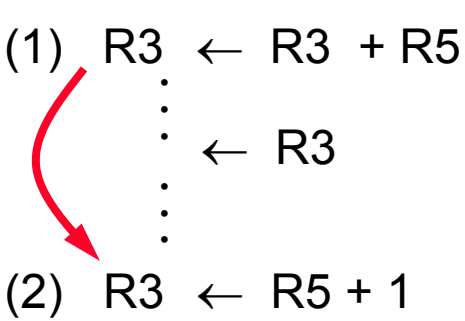
Reuse Same Set of Reg. in Each Iteration

Overlapped Execution of Different Iterations

Resolving False Dependencies



Must Prevent (2) from completing before (1) is dispatched



Must Prevent (2) from completing before (1) completes

Stalling: delay dispatching (or write back) of the later instruction

Copy Operands: Copy not-yet-used operand to prevent being overwritten (WAR)

Register Renaming: use a different register (WAW & WAR)

Register Renaming

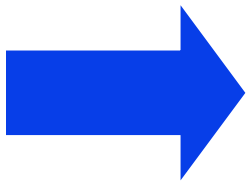
- Anti and output dependencies are false dependencies

$r_3 \leftarrow r_1$ op r_2
$r_5 \leftarrow r_3$ op r_4
$r_3 \leftarrow r_6$ op r_7

- The dependence is on name/location rather than data
- Given infinite number of registers, anti and output dependencies can always be eliminated

Original

r1 ← r2 / r3
r4 ← r1 * r5
r1 ← r3 + r6
r3 ← r1 - r4



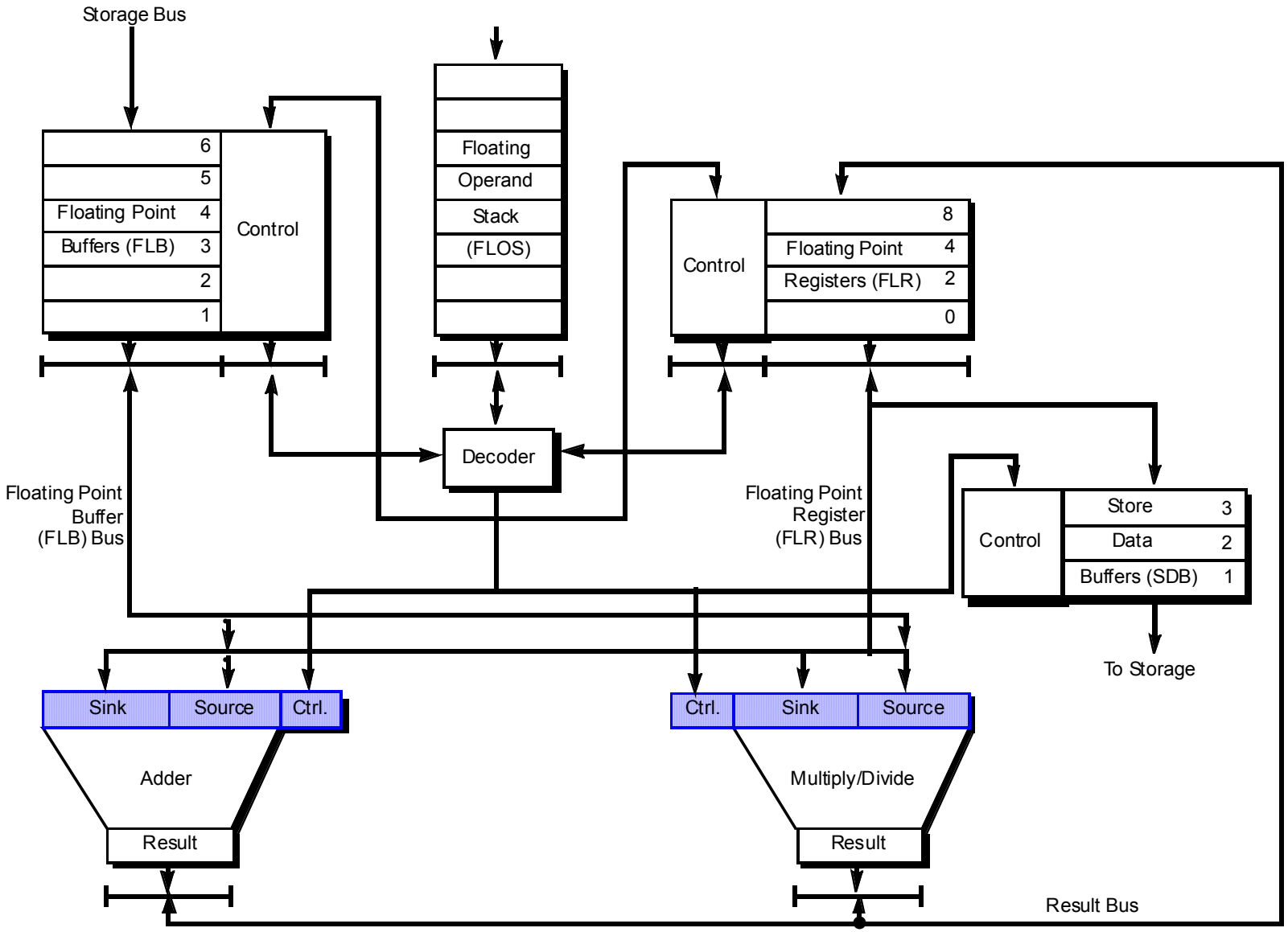
Renamed

r1 ← r2 / r3
r4 ← r1 * r5
r8 ← r3 + r6
r9 ← r8 - r4

Dataflow Order Execution

- Use data copying and/or hardware register renaming to eliminate WAR and WAW
  - register name refers to a temporary value produced by an earlier instruction *(ISA perspective)*
  - decouple register name from fixed storage location
  - disambiguate between register name reuse
- Maintain a window (or windows) of several pending instructions with only RAW dependence
- Issue instructions out-of-order
  - find instructions whose input operands are available
  - give preference to older instructions
  - A completing instruction's result can trigger other pending instructions *(RAW)*

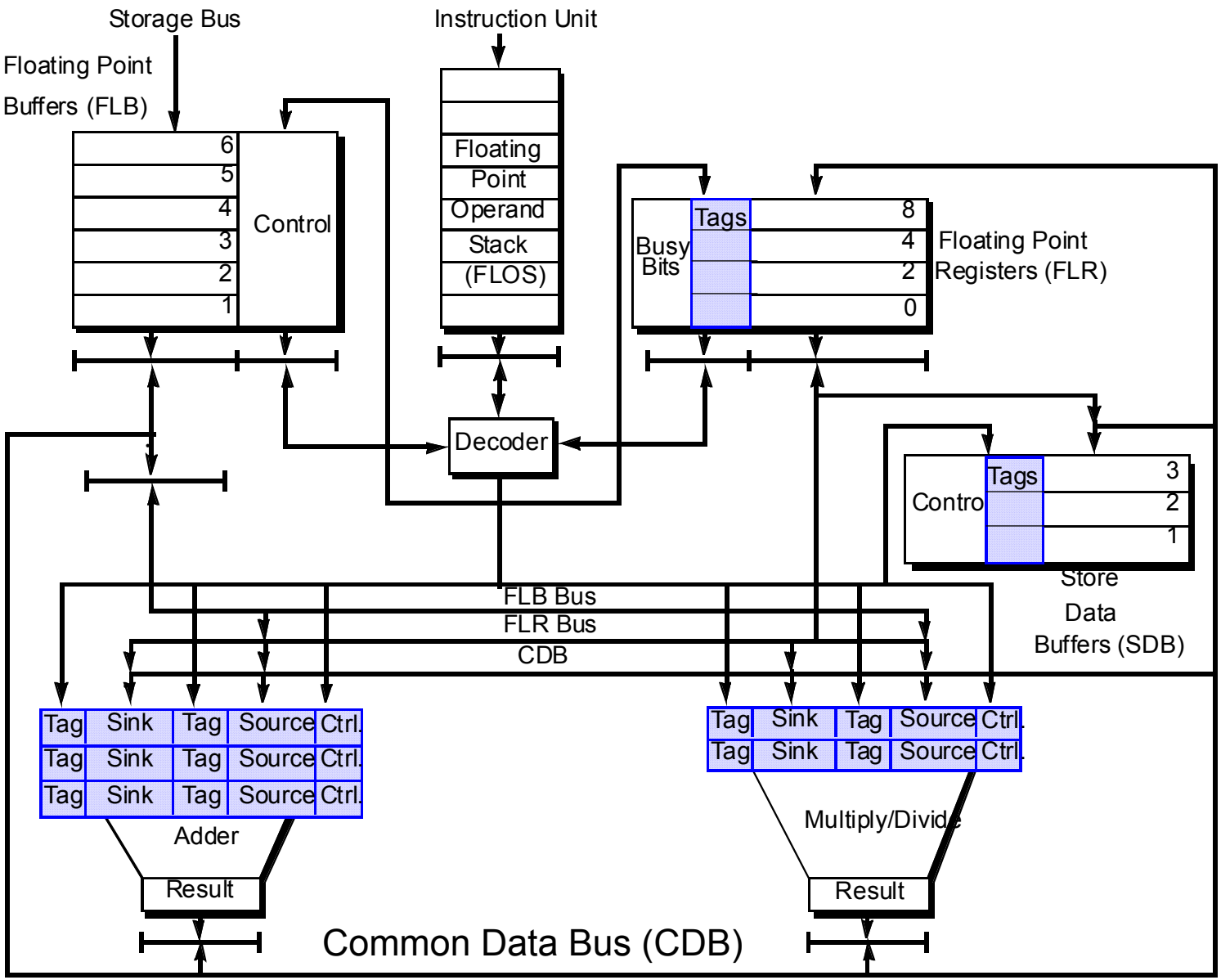
IBM 360 Floating Point Unit (Base Model)



Diversified Pipelined
Inorder Issue, Out-of-order Complete

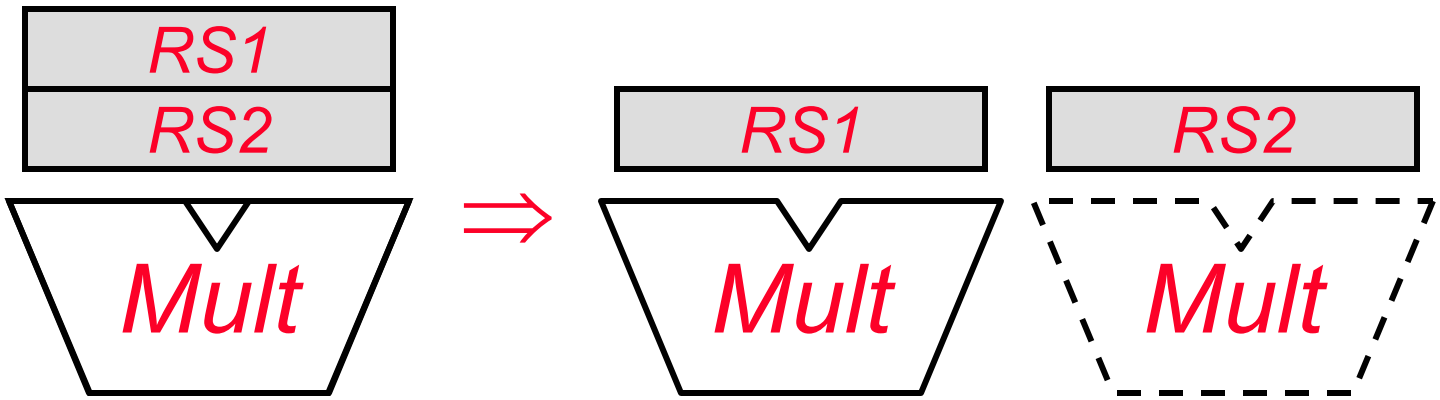
- Multiple functional units (FU's)
  - Floating-point add
  - Floating-point multiply/divide
- Three register files (pseudo reg-reg machine in FP unit)
  - (4) floating-point registers (FLR)
  - (6) floating-point buffers (FLB)
  - (3) store data buffers (SDB)
- Out of order instruction execution:
  - After decode the instruction unit passes all floating point instructions (in order) to the floating-point operation stack (FLOS).
  - In the floating point unit, instructions are then further decoded and issued from the FLOS to the two FU's
- Variable operation latencies (not pipelined):
  - Floating-point add: 2 cycles
  - Floating-point multiply: 3 cycles
  - Floating-point divide: 12 cycles

Tomasulo's Algorithm [IBM 360/91, 1967]



Reservation Station

- Buffers where instructions can wait for RAW hazard resolution and execution
- Associate more than one set of buffering registers (control, source, sink) with each FU ⇒ virtual FU's.
  - Add unit: three reservation stations
  - Multiply/divide unit: two reservation stations
- Pending (not yet executing) instructions can have either value operands or pseudo operands (aka. tags).



Rename Tags

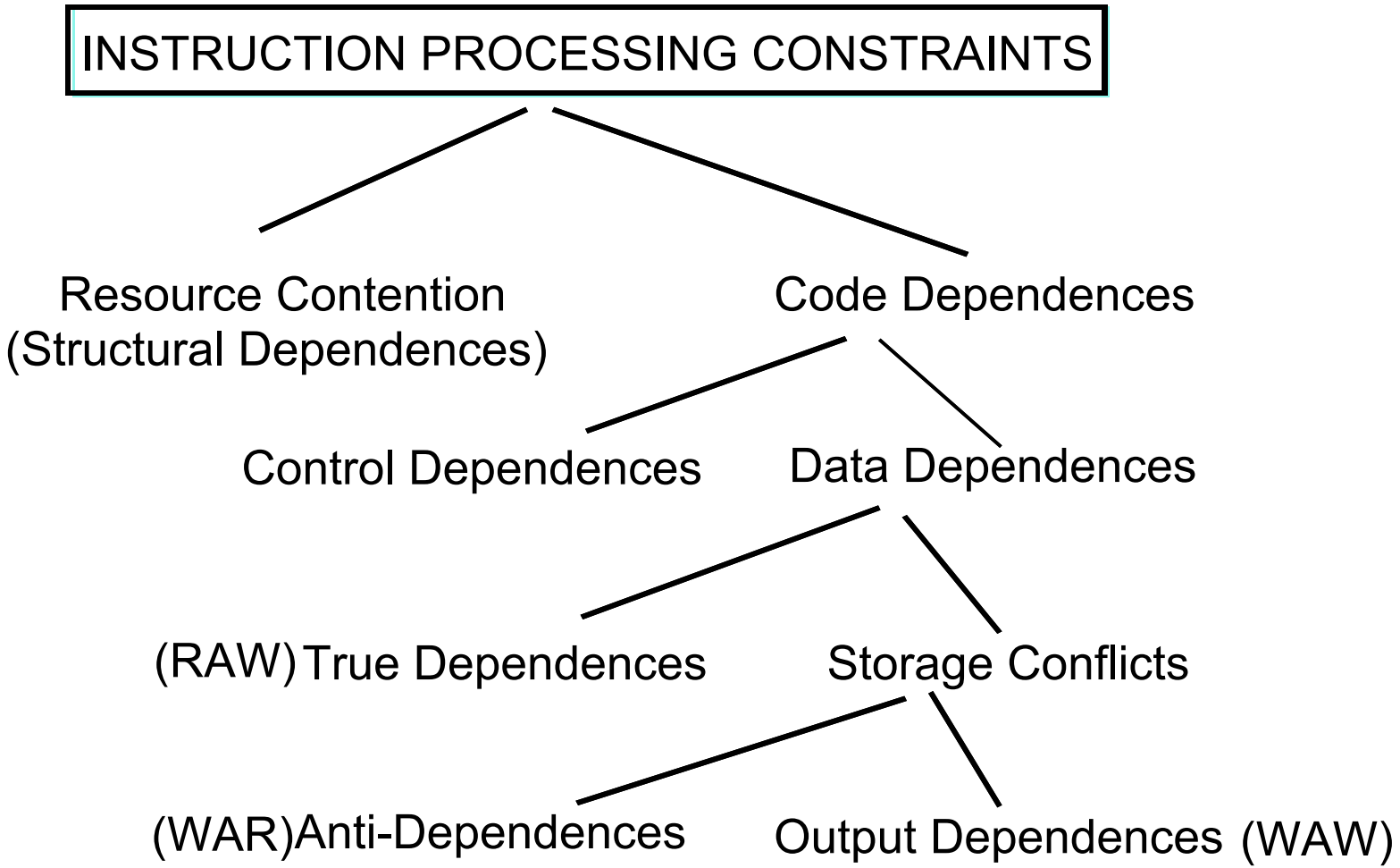
- Register names are normally bound to FLR registers
- When an FLR register is stale, the register "name" is bound to the pending-update instruction
- Tags are names to refer to these pending-update instructions
- In Tomasulo, A "tag" is statically bound to the buffer where a pending-update instruction waits.
  - 6 FLB's
  - 5 reservation stations (3 add RSs, 2 multiply/divide RSs)

    $\Rightarrow$ 4-bit tag is needed to identify the 11 potential sources
- Instructions can be dispatched to RSs with either value operands or just tags.
  - Tag operand $\Rightarrow$ unfulfilled RAW dependence
  - the instruction in the RS corresponding to the Tag will produce the actual value eventually

Common Data Bus (CDB)

- CDB is driven by all units that can update FLR
  - When an instruction finishes, it broadcasts both its "tag" and its result on the CDB.
  - *Why don't we need the destination register name?*
- Sources of CDB:
  - Floating-point buffers (FLB)
  - Two FU's (add unit and the multiply/divide unit)
- The CDB is monitored by all units that was left holding a tag instead of a value operand
  - Listens for tag broadcast on the CDB
  - If a tag matches, grab the value
- Destinations of CDB:
  - Reservation stations
  - Store data buffers (SDB)
  - Floating-point registers (FLR)

Superscalar Execution Check List



Structural Dependence Resolution

- Structural dependence: virtual FU's
  - FLOS can hold and decode up to 8 instructions.
  - Instructions are dispatched to the 5 reservation stations (virtual FU's) even though there are only two physical FU's.
  - Hence, structural dependence does not stall decoding

Why is this useful?

Resolving True-Dependence

- True dependence: Tags + CDB
  - If an operand is available in FLR, it is copied to RS
  - If an operand is not available then a tag is copied to the RS instead. This tag identifies the source (RS/instruction) of the pending write
  - Eventually the source instruction completes and broadcasts its tag and value on the CDB
  - Any reservation station entry, FLR entry or SDB entry that holds a matching tag as operand will latch in the broadcasted value from the CDB.

RAW dependence does not block subsequent independent instructions and does not block an FU

RAW Example:

i: R2 ← R0 + R4

j: R8 ← R0 + R2

Cyc #1:

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			6.0
2			3.5
4			10.0
8			7.8

Cyc #2:

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

Cyc #3:

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

RAW Example:

i: R2 ← R0 + R4
j: R8 ← R0 + R2

Cyc #1: dispatch i

RS	Tag	Sink	Tag	Src
1	0	6.0	0	10.0
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			6.0
2	X	1	--
4			10.0
8			7.8

Cyc #2: dispatch j

RS	Tag	Sink	Tag	Src
1	0	6.0	0	10.0
2	0	6.0	1	--
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			6.0
2	X	1	--
4			10.0
8	X	2	--

*Cyc #3: i in **RS 1** broadcasts tag and result: CBD=<<1,16.0>>*

RS	Tag	Sink	Tag	Src
1				
2	0	6.0	0	16.0
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			6.0
2			16.0
4			10.0
8	X	2	--

Resolving Anti-Dependence

- Anti-dependence: Operand Copying
  - If an operand is available in FLR, it is copied to RS with the issuing instruction
  - By copying this operand to RS, all WAR dependencies due to future writes to this same register are resolved

Hence, the reading of an operand is not delayed, possibly due to other dependencies, and subsequent writes are also not delayed.

WAR Example:

i: R4 ← R0 x R8
j: R0 ← R4 x R2
k: R2 ← R2 + R8

Cyc #1:

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			6.0
2			3.5
4			10.0
8			7.8

Cyc #2:

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

Cyc #3:

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

WAR Example:

i: R4 ← R0 x R8
j: R0 ← R4 x R2
k: R2 ← R2 + R8

Cyc #1: dispatch i

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4	0	6.0	0	7.8
5				

Mult/Div

FLR	Busy	Tag	Data
0			6.0
2			3.5
4	X	4	--
8			7.8

Cyc #2: dispatch j & k (assume dual issue)

RS	Tag	Sink	Tag	Src
1	0	3.5	0	7.8
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4	0	6.0	0	7.8
5	4	--	0	3.5

Mult/Div

FLR	Busy	Tag	Data
0	X	5	--
2	X	1	--
4	X	4	--
8			7.8

Cyc #3:

RS	Tag	Sink	Tag	Src
1	0	3.5	0	7.8
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4	0	6.0	0	7.8
5	4	--	0	3.5

Mult/Div

FLR	Busy	Tag	Data
0	X	5	--
2	X	1	--
4	X	4	--
8			7.8

WAR Example:

i: R4 ← R0 x R8
j: R0 ← R4 x R2
k: R2 ← R2 + R8

Cyc #4: RS 1 and 4 completes CBD=<<1,11.3>> & <<4,46,8>>

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5	0	46.8	0	3.5

Mult/Div

FLR	Busy	Tag	Data
0	X	5	--
2			11.3
4			46.8
8			7.8

Resolving Output-Dependence

- Output dependence: "register renaming" + result forwarding
  - If a FLR is waiting for a pending write, it's tag field will contain the tag of the source instruction
  - If a 2nd instruction comes along and want to write the same register
    - the register can be renamed to the 2nd instruction *(i.e. new tag)*
    - Any instruction that needs the value of the 1st pending write has the tag of the 1st instruction. Hence, the correct value will be forwarded from the 1st instruction directly
    - any subsequent instruction that reads the register will get the tag, or eventually the result, of the 2nd instruction

WAW dependence is resolved without stalling a physical functional unit and does not require additional buffers to ensure sequential write back to the register file.

WAW Example:

i: R4 ← R0 x R8
j: R2 ← R0 + R4
k: R4 ← R0 + R8
l: R8 ← R4 x R8

Cyc #1:

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			6.0
2			3.5
4			10.0
8			7.8

Cyc #2:

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

Cyc #3:

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

WAW Example:

i: R4 ← R0 x R8
j: R2 ← R0 + R4
k: R4 ← R0 + R8
l: R8 ← R4 x R8

Cyc #4:

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

Cyc #5:

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

Cyc #6:

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

WAW Example:

i: R4 ← R0 x R8
j: R2 ← R0 + R4
k: R4 ← R0 + R8
l: R8 ← R4 x R8

Cyc #1: dispatch i and j

Adder

RS	Tag	Sink	Tag	Src
1	0	6.0	4	--
2				
3				

Mult/Div

RS	Tag	Sink	Tag	Src
4	0	6.0	0	7.8
5				

FLR	Busy	Tag	Data
0			6.0
2	X	1	--
4	X	4	--
8			7.8

Cyc #2: dispatch k and l

Adder

RS	Tag	Sink	Tag	Src
1	0	6.0	4	--
2	0	6.0	0	7.8
3				

Mult/Div

RS	Tag	Sink	Tag	Src
4	0	6.0	0	7.8
5	2	--	0	7.8

FLR	Busy	Tag	Data
0			6.0
2	X	1	--
4	X	2	--
8	X	5	--

Cyc #3:

Adder

RS	Tag	Sink	Tag	Src
1	0	6.0	4	--
2	0	6.0	0	7.8
3				

Mult/Div

RS	Tag	Sink	Tag	Src
4	0	6.0	0	7.8
5	2	--	0	7.8

FLR	Busy	Tag	Data
0			6.0
2	X	1	--
4	X	2	--
8	X	5	--

WAW Example:

i: R4 ← R0 x R8
j: R2 ← R0 + R4
k: R4 ← R0 + R8
l: R8 ← R4 x R8

Cyc #4: RS 2 and 4 completes: CBD=<<2,13k8>> & <<4,46,8>>

RS	Tag	Sink	Tag	Src
1	0	6.0	0	46.2
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5	0	13.8	0	7.8

Mult/Div

FLR	Busy	Tag	Data
0			6.0
2	X	1	--
4			13.8
8	X	5	--

Cyc #5:

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

Cyc #6:

RS	Tag	Sink	Tag	Src
1				
2				
3				

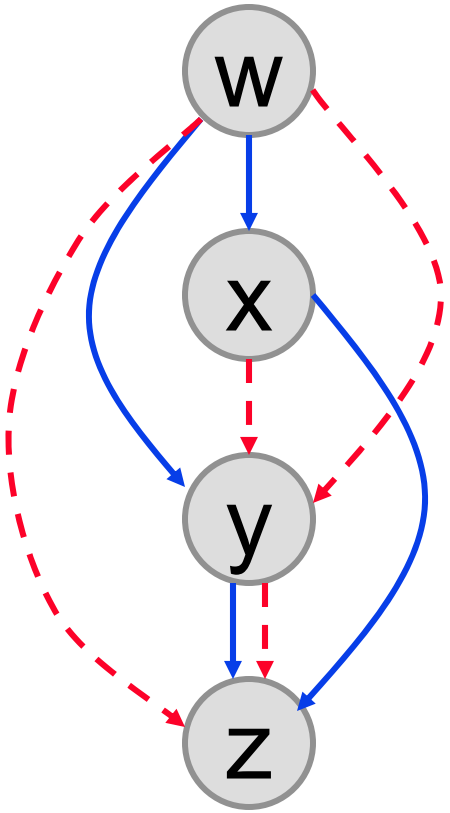
Adder

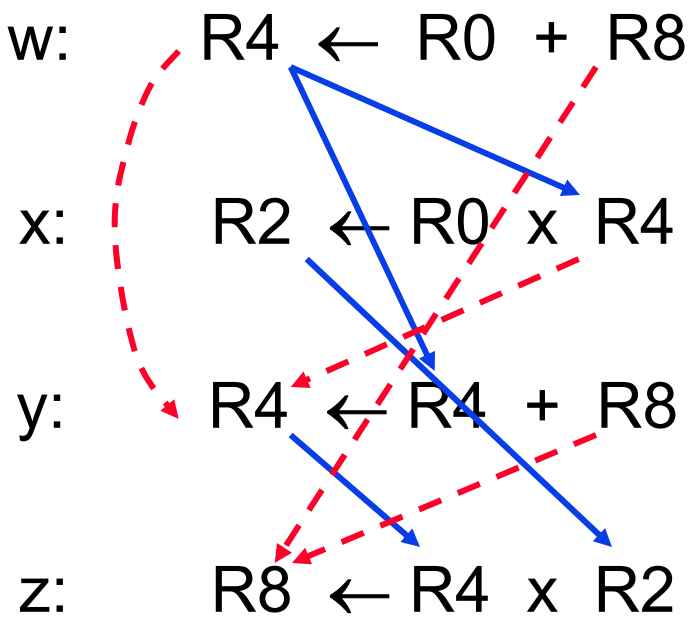
RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

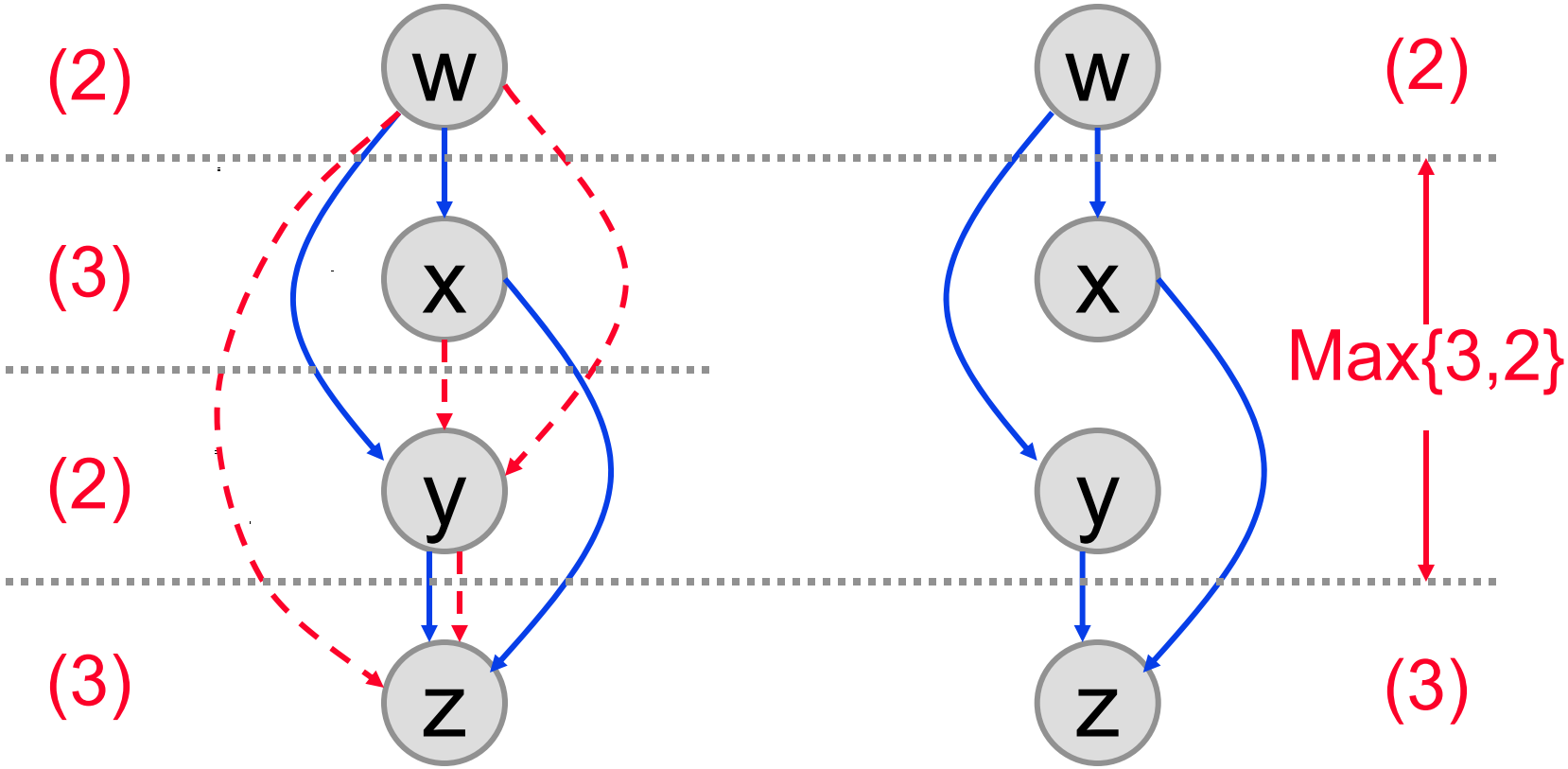
FLR	Busy	Tag	Data
0			
2			
4			
8			

Code Sequence for Example 4





Critical Path Analysis



In-order State and Precise Interrupt

- If an IBM 360/91 instruction causes an exception, can we stop the processor in a precise state?

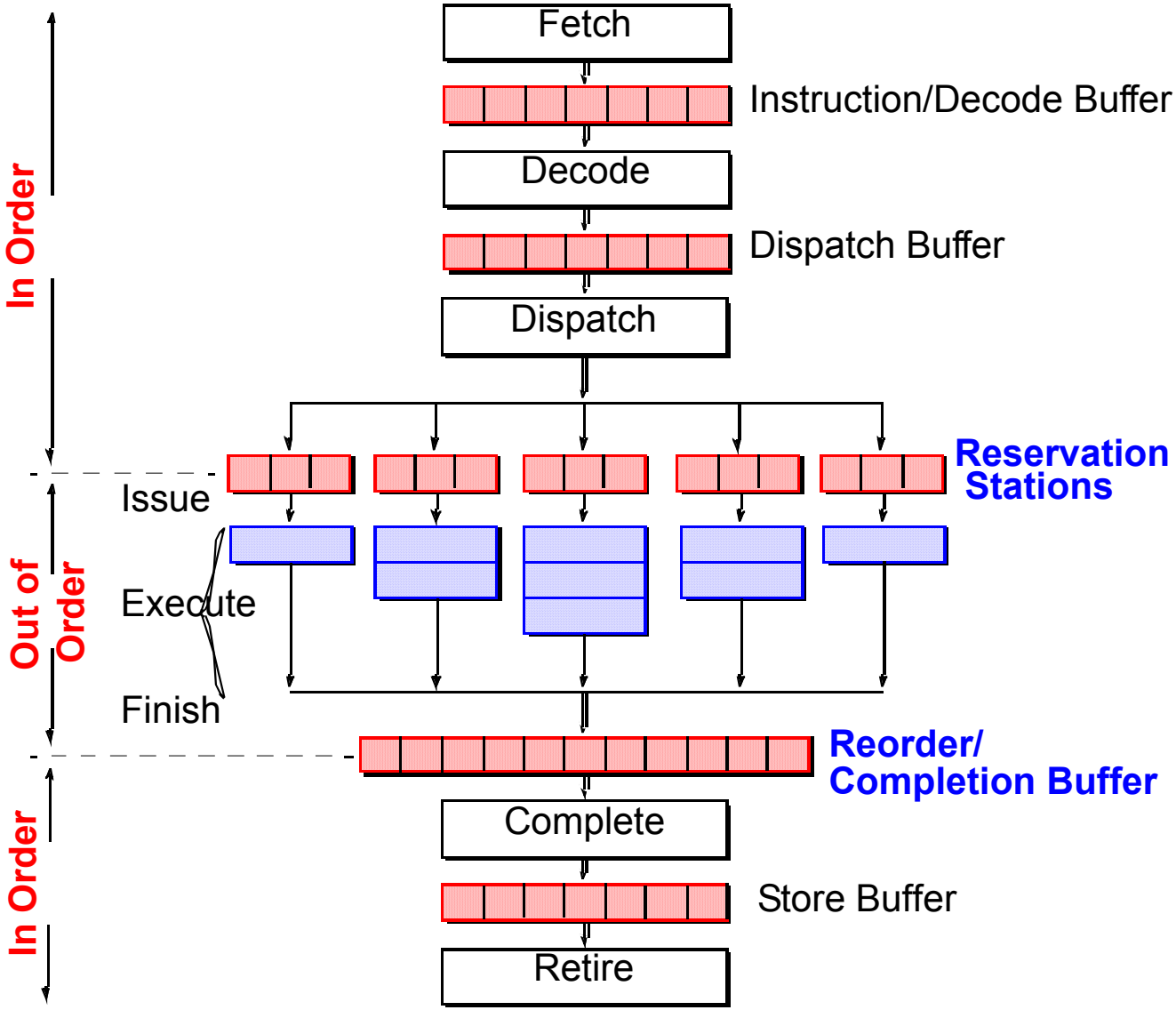
i: R4 ← R0 x R8
j: R2 ← R0 + R4 Exception!!
k: R4 ← R0 + R8
l: R8 ← R4 x R8

- By the time *j* executes, *k* has already updated *R4? How do you rewind the register file to the state just after i?*

Recall, i never even got to update R4!!

- Next time, how to maintain an "in-order" state of the machine. (In-order state = the machine state as viewed by the first not-yet-completed instruction.)

A Modern Superscalar Processor



Example 4

Cyc #1

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

Cyc #2

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

```
w:   R4 ← R0 + R8
x:   R2 ← R0 x R4
y:   R4 ← R4 + R8
z:   R8 ← R4 x R2
```

Example 4

Cyc #3

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

Cyc #4

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

Cyc #5

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

Example 4

Cyc #6

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

Cyc #7

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			

Cyc #8

RS	Tag	Sink	Tag	Src
1				
2				
3				

Adder

RS	Tag	Sink	Tag	Src
4				
5				

Mult/Div

FLR	Busy	Tag	Data
0			
2			
4			
8			