



# Register Data Flow

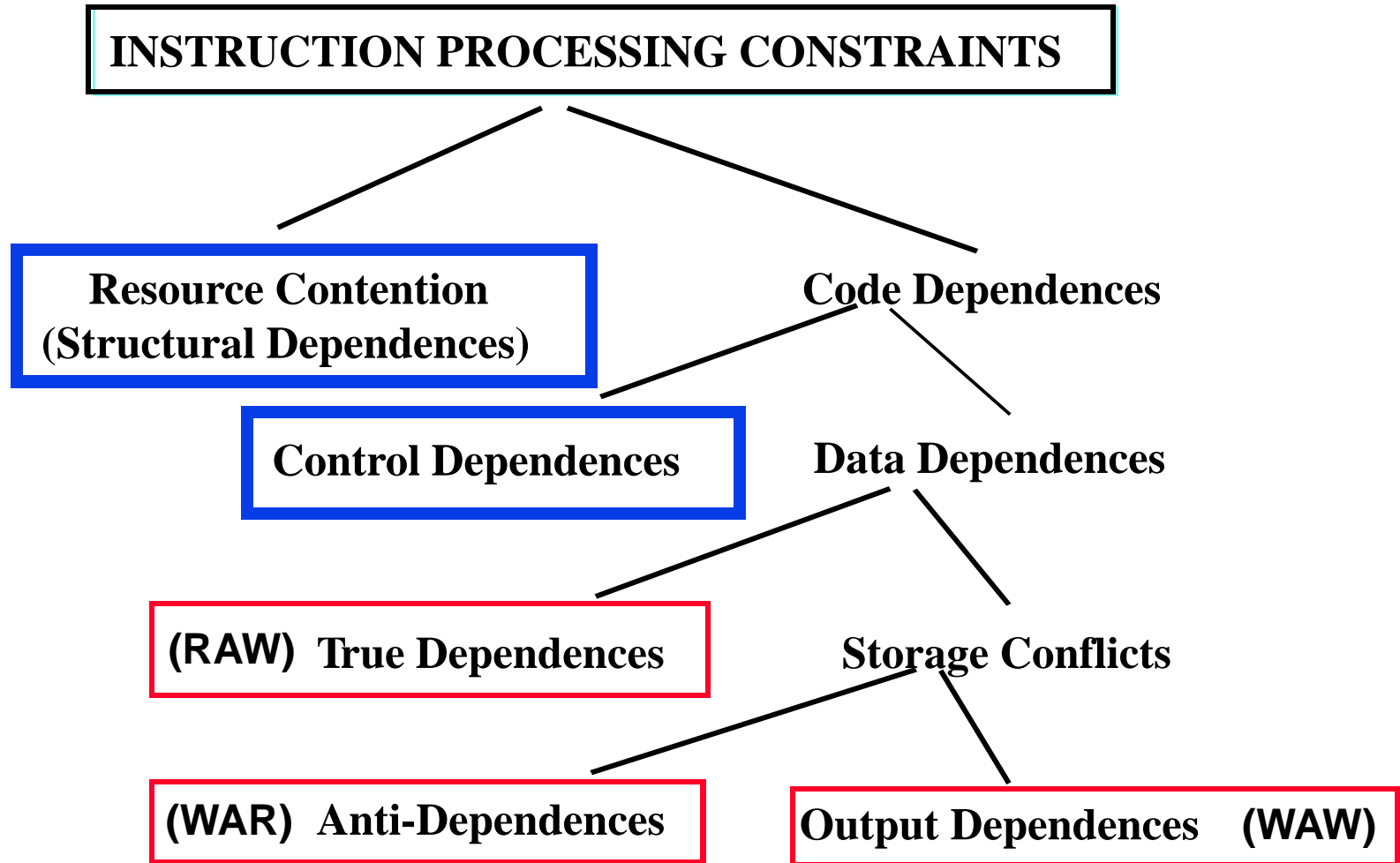
ECE/CS 752 Fall 2017

*Prof. Mikko H. Lipasti*  
*University of Wisconsin-Madison*

# Register Data Flow Techniques

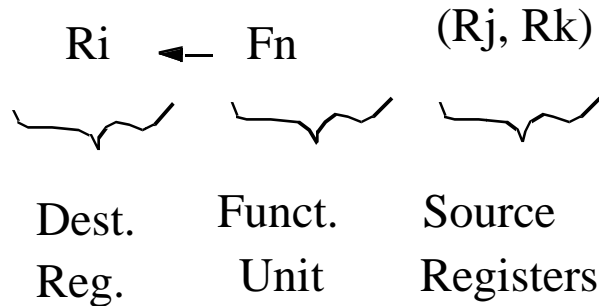
- Register Data Flow
  - Resolving Anti-dependences
  - Resolving Output Dependences
  - Resolving True Data Dependences
- Tomasulo's Algorithm [Tomasulo, 1967]
  - Modified IBM 360/91 Floating-point Unit
  - Reservation Stations
  - Common Data Bus
  - Register Tags
  - Operation of Dependency Mechanisms

# The Big Picture



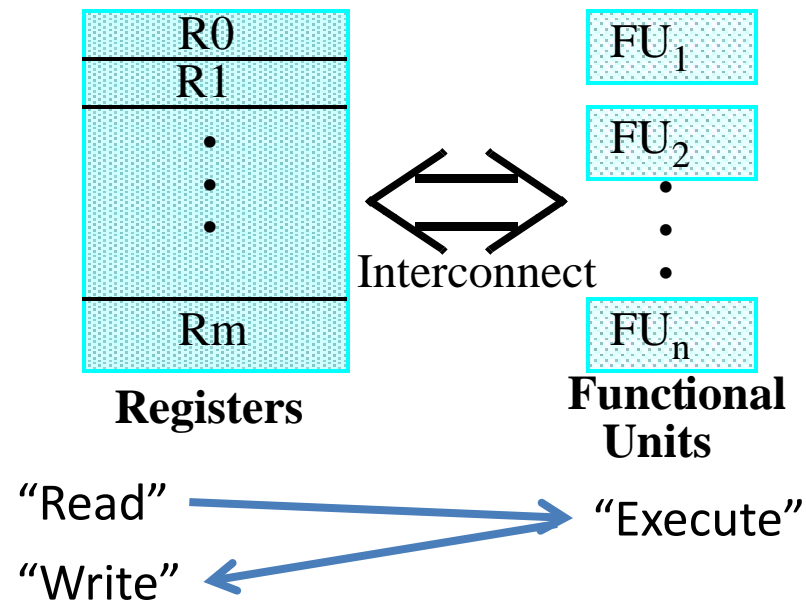
# Register Data Flow

Each ALU Instruction:



“Register Transfer”

## INSTRUCTION EXECUTION MODEL



**Need Availability of  $F_n$  (Structural Dependences)**

**Need Availability of  $R_j, R_k$  (True Data Dependences)**

**Need Availability of  $R_i$  (Anti-and output Dependences)**

# Causes of (Register) Storage Conflict

## REGISTER RECYCLING

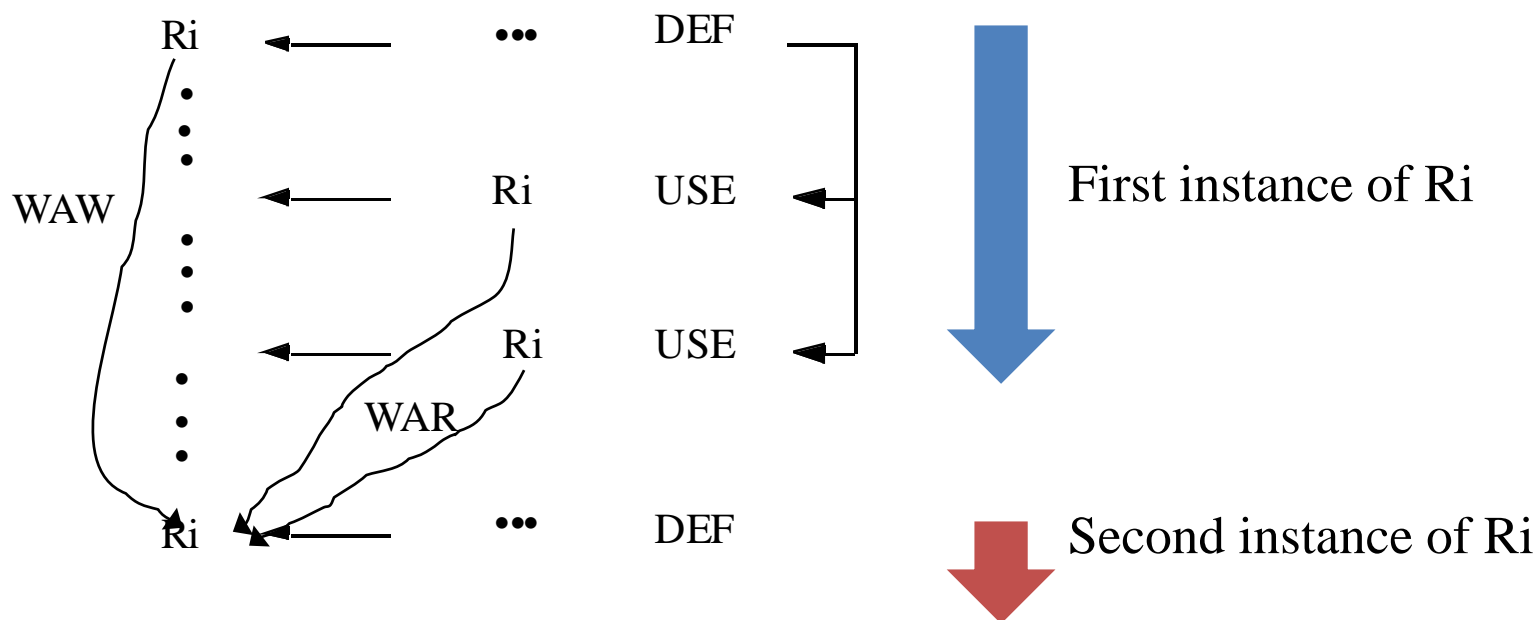
MAXIMIZE USE OF REGISTERS

MULTIPLE ASSIGNMENTS OF VALUES TO REGISTERS

## OUT OF ORDER ISSUING AND COMPLETION

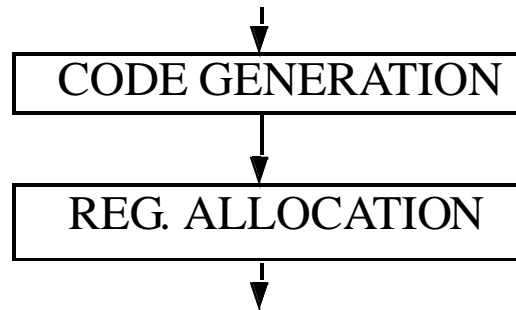
LOSE IMPLIED PRECEDENCE OF SEQUENTIAL CODE

LOSE 1-1 CORRESPONDENCE BETWEEN VALUES AND REGISTERS



# Contribution to Register Recycling

## COMPILER REGISTER ALLOCATION



Single Assignment, Symbolic Reg.

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

“Spill code”  
(if not  
enough  
registers)

## 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
```

A list of assembly instructions. A hand-drawn loop encloses instructions 9 through 21, indicating a loop body. Arrows point from the loop back to instruction 9, showing the loop structure.


```
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 Anti-Dependences

.  
 .  
 .  
 (1) R4  $\leftarrow$  R3 + 1  
 (2) R3  $\leftarrow$  R5 + 1



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

## STALL DISPATCHING

DELAY DISPATCHING OF (2)

REQUIRE RECHECKING AND REACCESSING

WAR  
only

## COPY OPERAND

COPY NOT-YET-USED OPERAND TO PREVENT BEING  
OVERWRITTEN

MUST USE TAG IF ACTUAL OPERAND NOT-YET-AVAILABLE

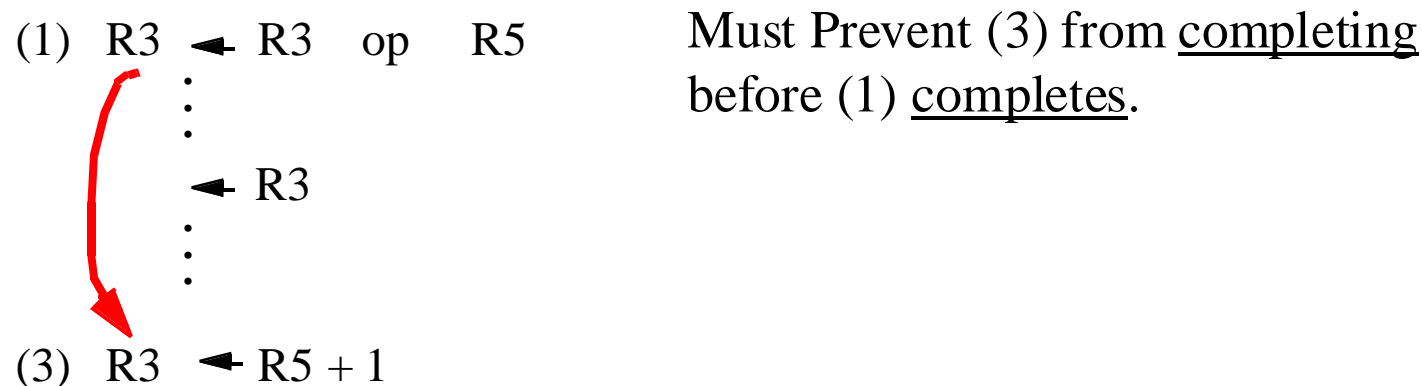
WAR  
and  
WAW

## RENAME REGISTER

HARDWARE ALLOCATION

R3  $\leftarrow$  ...  
 $\leftarrow$  R3  
 R3'  $\leftarrow$  ...  
 $\leftarrow$  R3'

# Resolving Output Dependences



## STALL DISPATCHING/ISSUING

DENOTE OUTPUT DEPENDENCE

HOLD DISPATCHING UNTIL RESOLUTION OF DEPENDENCE

ALLOW DECODING OF SUBSEQUENT INSTRUCTIONS

## RENAME REGISTER

HARDWARE ALLOCATION

$R3$	$\leftarrow \dots$
	$\leftarrow R3$
$R3'$	$\leftarrow \dots$
	$\leftarrow R3'$



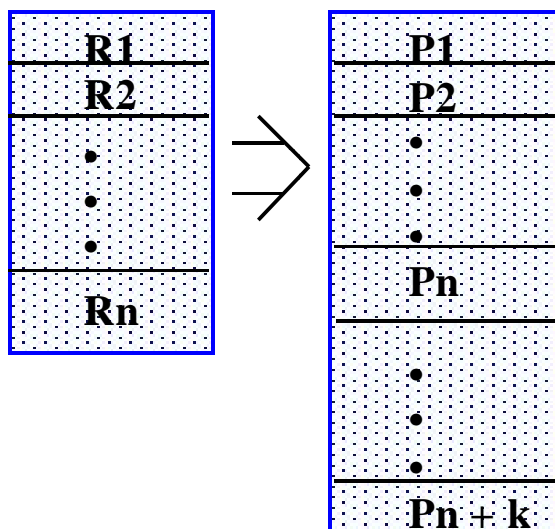
# Register Renaming

## Register Renaming Resolves:

**Anti-Dependences**  
**Output Dependences**

**Architected  
Registers**

**Physical  
Registers**



## Design of Redundant Registers

**Number:**

**One**

**Multiple**

**Allocation:**

**Fixed for Each Register**

**Pooled for all Registers**

**Location:**

**Attached to Register File  
(Centralized)**

**Attached to functional units  
(Distributed)**

# Register Renaming in the RIOS-I FPU



Fload R7 <= Mem[] (P32 freed)

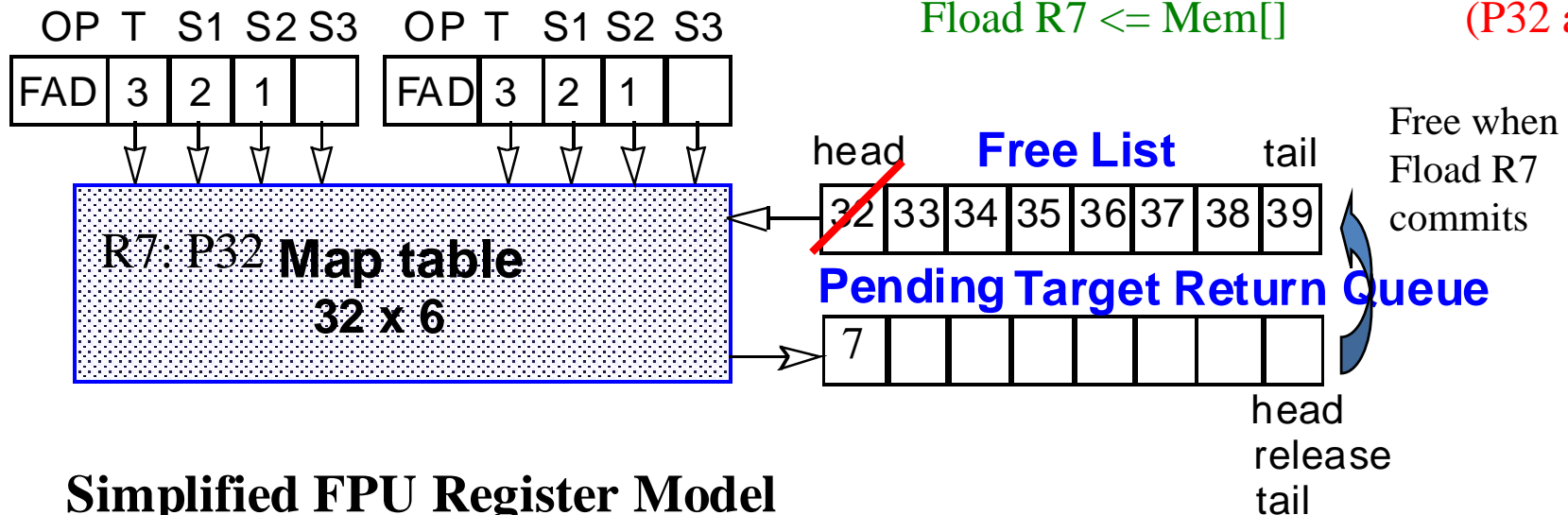
...

<= R7 (actual last use) (P32)

...

Fload R7 <= Mem[] (P32 alloc)

## FPU Register Renaming



## Simplified FPU Register Model

Incoming FPU instructions pass through a renaming table prior to decode

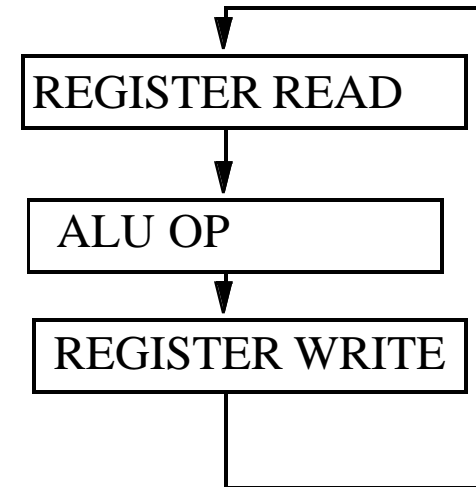
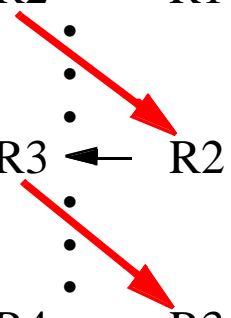
The 32 architectural registers are remapped to 40 physical registers

Physical register names are used within the FPU

Complex control logic maintains active register mapping

# Resolving True Data Dependences

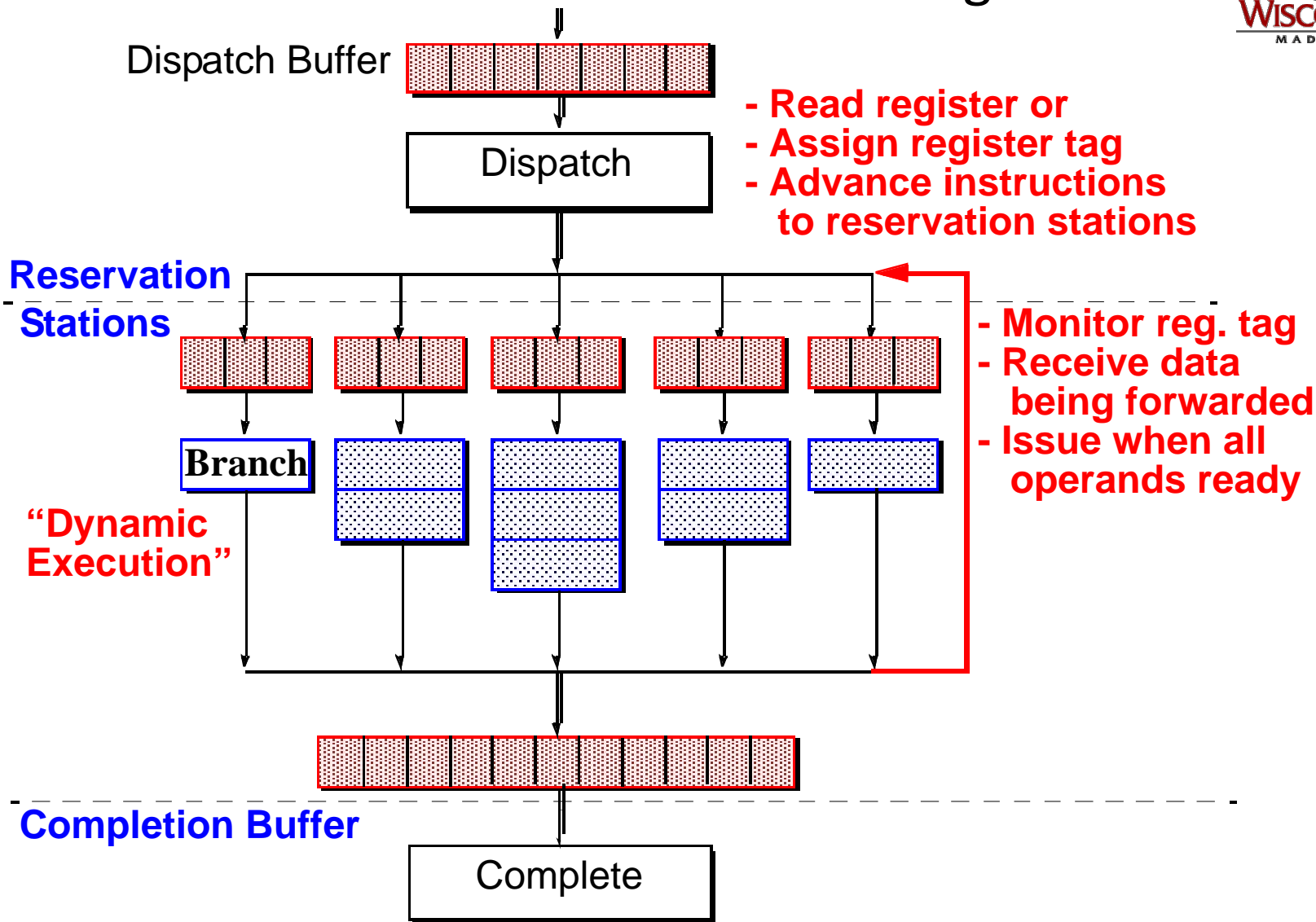
(1)  $R2 \leftarrow R1 + 1$   
     $\vdots$   
(2)  $R3 \leftarrow R2$   
     $\vdots$   
(3)  $R4 \leftarrow R3$



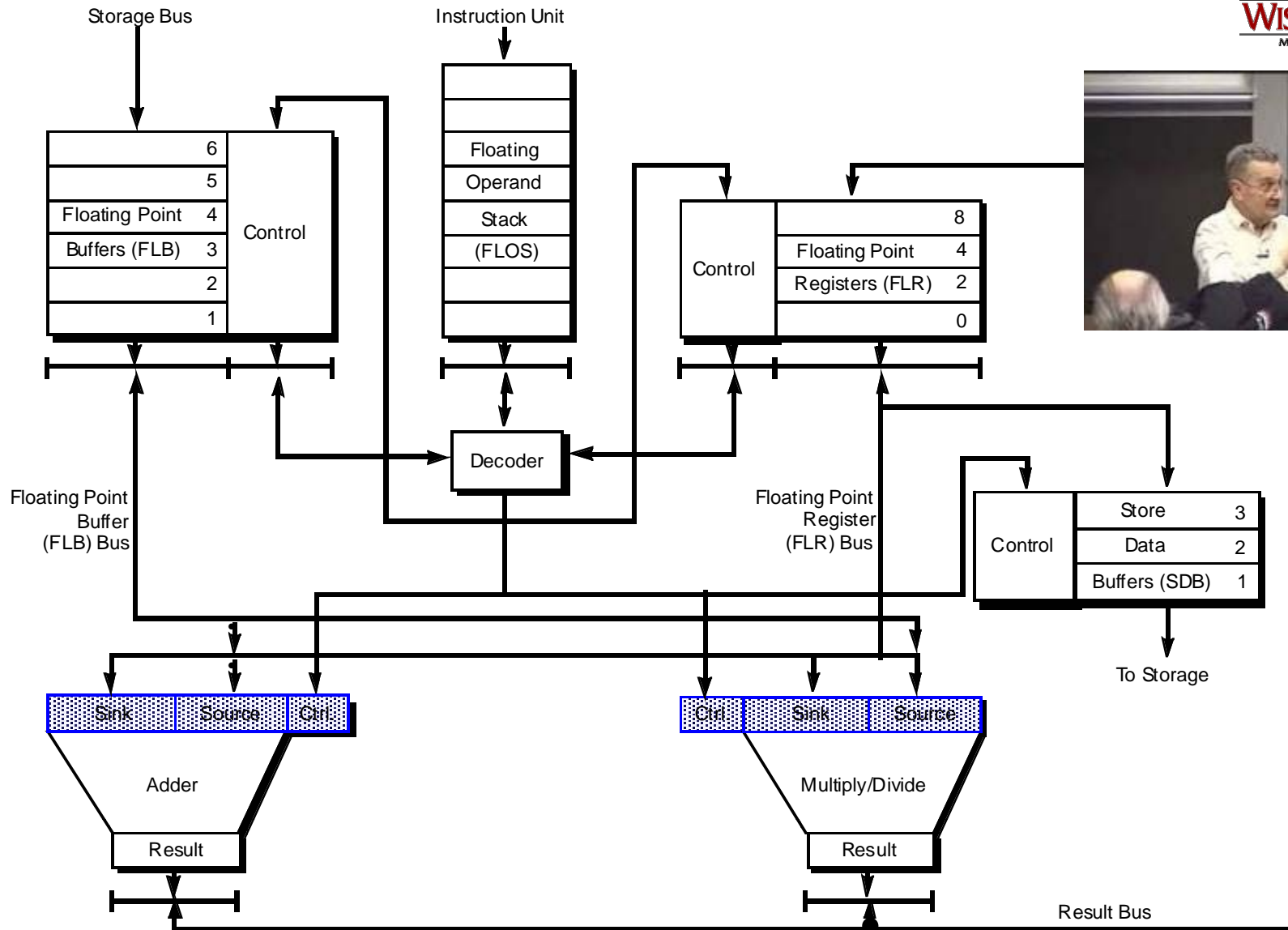
**STALL DISPATCHING**  
**ADVANCE INSTRUCTIONS**

- 1) Read register(s), get “IOU” if not ready
- 2) Advance to reservation station
- 3) Wait for “IOU” to show up
- 4) Execute

# Embedded “Data Flow” Engine



# Tomasulo's Algorithm [Tomasulo, 1967]



# IBM 360/91 FPU

- **Multiple functional units (FU's)**
  - Floating-point add
  - Floating-point multiply/divide
- **Three register files (pseudo reg-reg machine in floating-point 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) [actually a queue, not a stack]
  - In the floating point unit, instructions are then further decoded and issued from the FLOS to the two FU's
- **Variable operation latencies:**
  - Floating-point add: 2 cycles
  - Floating-point multiply: 3 cycles
  - Floating-point divide: 12 cycles
- **Goal: achieve concurrent execution of multiple floating-point instructions, in addition to achieving one instruction per cycle in instruction pipeline**

# Dependence Mechanisms

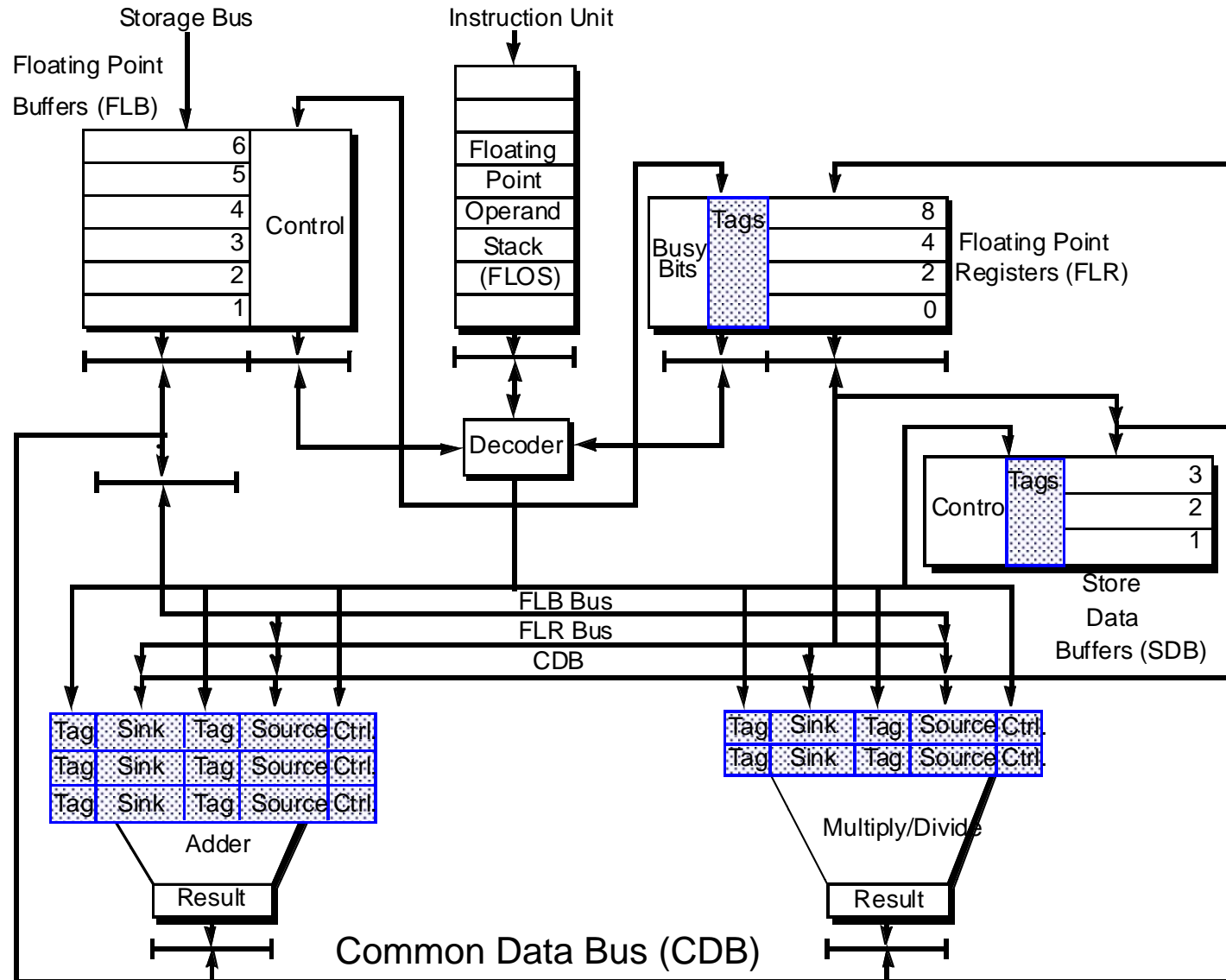
**Two Address IBM 360 Instruction Format:**

**$R1 \leftarrow R1 \text{ op } R2$**

**Major dependence mechanisms:**

- **Structural (FU) dependence = > virtual FU's**
  - Reservation stations
- **True dependence = > pseudo operands + result forwarding**
  - Register tags
  - Reservation stations
  - Common data bus (CDB)
- **Anti-dependence = > operand copying**
  - Reservation stations
- **Output dependence = > register renaming + result forwarding**
  - Register tags
  - Reservation stations
  - Common data bus (CDB)

# IBM 360/91 FPU





# Reservation Stations

- Used to collect operands or pseudo operands (tags).
- 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

Tag	Sink	Tag	Source
0 implies valid data	Source value	0 implies valid data	Source value

## Common Data Bus (CDB)

- **CDB is fed by all units that can alter a register (or supply register values) and it feeds all units which can have a register as an operand.**
- Sources of CDB:
  - Floating-point buffers (FLB)
  - Two FU's (add unit and the multiply/divide unit)
  - $6 \text{ FLB} + 3 \text{ addRS} + 2 \text{ muldivRS} = 11$  unique sources
  - 3 physical sources (FLB, adder, mul/div)
- Destinations of CDB:
  - Reservation stations
  - Floating-point registers (FLR)
  - Store data buffers (SDB)
  - $(5 \text{ RS} \times 2) + 4 \text{ FLR} + 3 \text{ SDB} : \text{CDB has 17 destinations}$
- Circuit design very challenging
  - 3 physical sources must arbitrate for access to CDB
  - Tag + data must be driven to 17 destinations

# Register Tags

- **Every source of a register value must be uniquely identified by its own tag value.**
  - (6) FLB's
  - (5) reservation stations (3 with add unit, 2 with multiply/divide unit)  
= = > 4-bit tag is needed to identify the 11 potential sources
- **Every destination of a register value must carry a tag field.**
  - (5) “sink” entries of the reservation stations
  - (5) “source” entries of the reservation stations
  - (4) FLR's
  - (3) SDB's
  - = = > a total of 17 tag fields are needed (i.e. 17 places that need tags)

# Operation of Dependence Mechanisms

## 1. **Structural (FU) 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 dispatching.

## 2. **True dependence** = > **pseudo operands + result forwarding**

- If an operand is available in FLR, it is copied to a res. station entry.
- If an operand is not available (i.e. there is pending write), then a tag is copied to the reservation station entry instead. This tag identifies the source of the pending write. This instruction then waits in its reservation station for the true dependence to be resolved.
- When the operand is finally produced by the source (ID of source = tag value), this source unit asserts its ID, i.e. its tag value, on the CDB followed by broadcasting of the operand on the CDB.
- All the reservation station entries and the FLR entries and SDB entries carrying this tag value in their tag fields will detect a match of tag values and latch in the broadcasted operand from the CDB.
- Hence, true dependence does not block subsequent independent instructions and does not stall a physical FU. Forwarding also minimizes delay due to true dependence.

i:  $R2 \leq R0 + R4$

# Example 1

j:  $R8 \leq R0 + R2$  (RAW on R2)

Cycle #1

DISPATCHED INSTRUCTION(S):

i

	ID	Tag	Sink	Tag	Source
i	1	0	6.0	0	10.0
	2				
	3				

Adder i

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

	Busy	Tag	Data
0			6.0
2	x	1	3.5
4			10.0
8			7.8

Cycle #2

DISPATCHED INSTRUCTION(S):

j

	ID	Tag	Sink	Tag	Source
i	1	0	6.0	0	10.0
j	2	0	6.0	1	16.0
	3				

Adder i

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

	Busy	Tag	Data
0			6.0
2	x	1	16.0
4			10.0
8	x	2	7.8

Cycle #3

DISPATCHED INSTRUCTION(S):

	ID	Tag	Sink	Tag	Source
	1				
j	2	0	6.0	0	16.0
	3				

Adder j

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

	Busy	Tag	Data
0			6.0
2			16.0
4			10.0
8	x	2	7.8

# Operation of Dependence Mechanisms

## 3. **Anti-dependence** = > operand copying

- If an operand is available in FLR, it is copied to a reservation station entry.
- By copying this operand to the reservation station, all anti-dependences due to future writes to this same register are resolved.
- Hence, the reading of an operand is not delayed, possibly due to other dependences, and subsequent writes are also not delayed.

i:  $R4 \leq R0 * R8$

j:  $R0 \leq R4 * R2$  (RAW on R4)

k:  $R2 \leq R2 + R8$  (WAR on R2)

## Example 2

Cycle #1

DISPATCHED INSTRUCTION(S):

i, j

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
i 4	0	6.0	0	7.8
j 5	4		0	3.5

Mult/Div i

Busy	Tag	Data
0 x	5	6.0
2		3.5
4 x	4	10.0
8		7.8

Cycle #2

DISPATCHED INSTRUCTION(S):

k

ID	Tag	Sink	Tag	Source
k 1	0	3.5	0	7.8
2				
3				

Adder k

ID	Tag	Sink	Tag	Source
i 4	0	6.0	0	7.8
j 5	4		0	3.5

Mult/Div i

Busy	Tag	Data
0 x	5	6.0
2 x	1	3.5
4 x	4	10.0
8		7.8

Cycle #3

DISPATCHED INSTRUCTION(S):

ID	Tag	Sink	Tag	Source
k 1	0	3.5	0	7.8
2				
3				

Adder k

ID	Tag	Sink	Tag	Source
i 4	0	6.0	0	7.8
j 5	4		0	3.5

Mult/Div i

Busy	Tag	Data
0 x	5	6.0
2 x	1	3.5
4 x	4	10.0
8		7.8

# Operation of Dependence Mechanisms

## 3. **Output dependence** = > register renaming + result forwarding

- If a register is waiting for a pending write, its tag field will contain the ID, or tag value, of the source for that pending write.
- When that source eventually produces the result, that result will be written into the register via the CDB.
- It is possible that prior to the completion of the pending write, another instruction can come along and also has that same register as its destination register.
- If this occurs, the operands (or pseudo operands) needed by this instruction are still copied to an available reservation station. In addition, the tag field of the destination register of this instruction is updated with the ID of this new reservation station, i.e. the old tag value is overwritten. This will ensure that the said register will get the latest value, i.e. the late completing earlier write cannot overwrite a later write.
- Hence, the output dependence is resolved without stalling a physical functional unit, not requiring additional buffers to ensure sequential write back to the register file.



What if j causes FP overflow exception?

- where is R4?
- it is lost => imprecise exceptions!

i:  $R4 \leq R0 * R8$

j:  $R2 \leq R0 + R4$  (RAW on R4)

k:  $R4 \leq R0 + R8$  (WAW on R4)

l:  $R8 \leq R4 * R8$  (RAW on R4)

## Example 3

Cycle #1

DISPATCHED INSTRUCTION(S):

i, j

	ID	Tag	Sink	Tag	Source
j	1	0	6.0	4	
	2				
	3				

Adder

	ID	Tag	Sink	Tag	Source
i	4	0	6.0	0	7.8
	5				

Mult/Div i

	Busy	Tag	Data
0			6.0
2	x	1	3.5
4	x	4	10.0
8			7.8

Cycle #2

DISPATCHED INSTRUCTION(S):

k, l

	ID	Tag	Sink	Tag	Source
j	1	0	6.0	4	
k	2	0	6.0	0	7.8
	3				

Adder k

	ID	Tag	Sink	Tag	Source
i	4	0	6.0	0	7.8
l	5	2		0	7.8

Mult/Div i

	Busy	Tag	Data
0			6.0
2	x	1	3.5
4	x	2	10.0
8	x	5	7.8

Cycle #3

DISPATCHED INSTRUCTION(S):

	ID	Tag	Sink	Tag	Source
j	1	0	6.0	4	
k	2	0	6.0	0	7.8
	3				

Adder k

	ID	Tag	Sink	Tag	Source
i	4	0	6.0	0	7.8
l	5	2		0	7.8

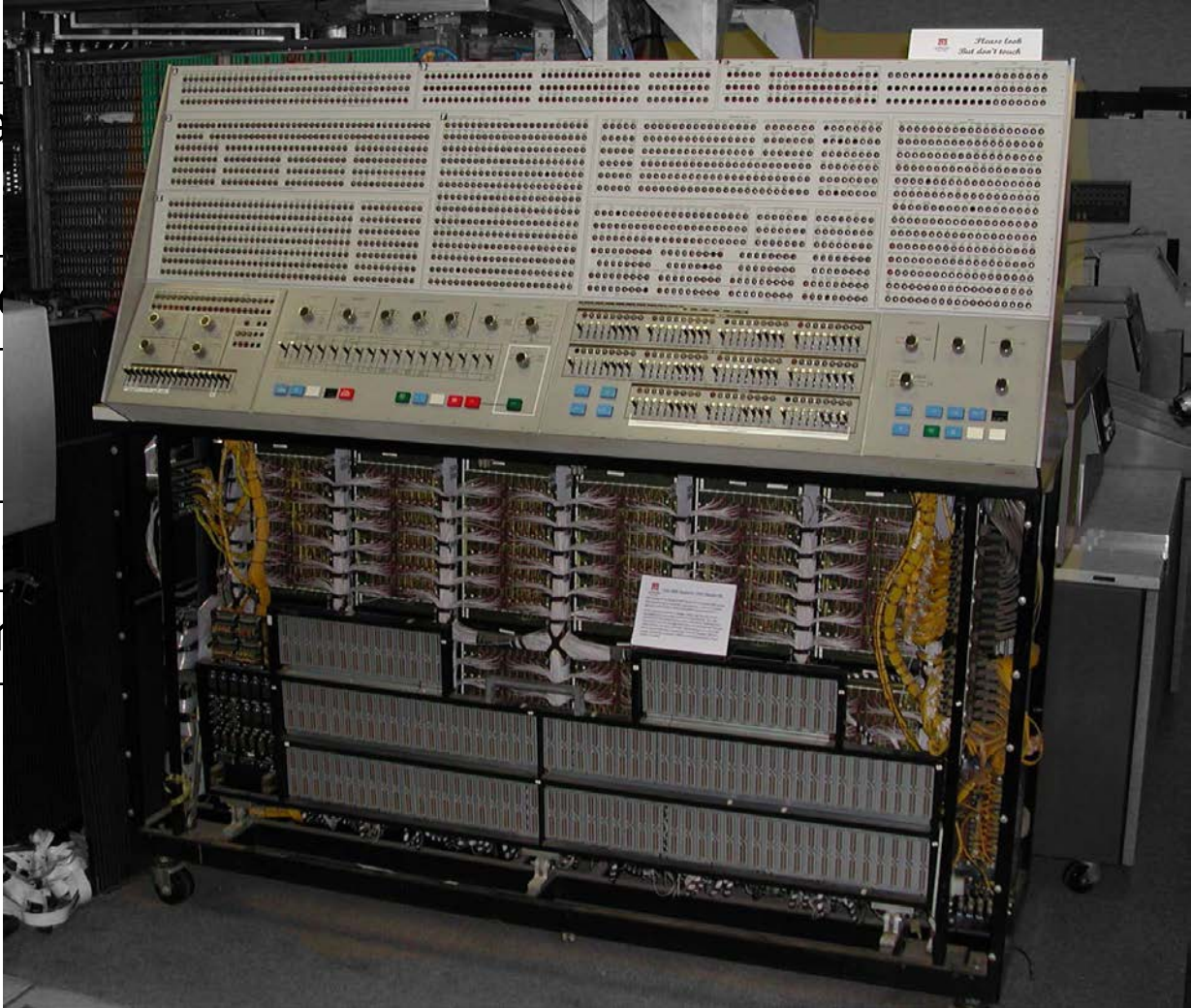
Mult/Div i

	Busy	Tag	Data
0			6.0
2	x	1	3.5
4	x	2	13.8
8			7.8

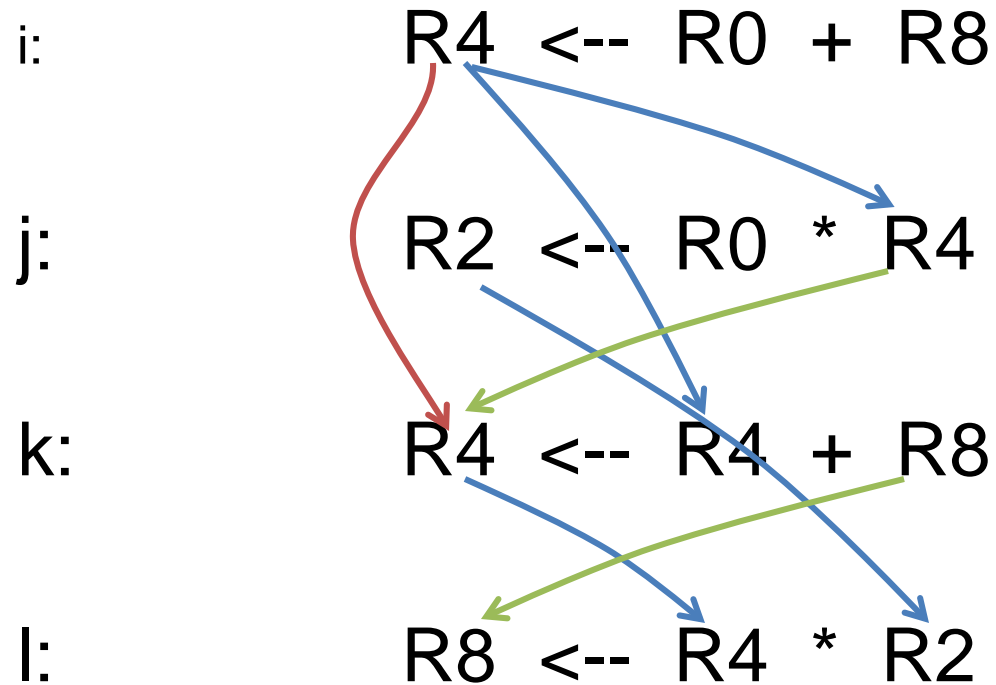
# Summary of Tomasulo's Algorithm

- Supports out of order execution of instructions.
- Resolves dependences dynamically using hardware.
- Attempts to delay the resolution of dependencies as late as possible.
- **Structural dependence** does not stall issuing; virtual FU's in the form of reservation stations are used.
- **Output dependence** does not stall issuing; copying of old tag to reservation station and updating of tag field of the register with pending write with the new tag.
- **True dependence** with a pending write operand does not stall the reading of operands; pseudo operand (tag) is copied to reservation station.
- **Anti-dependence** does not stall write back; earlier copying of operand awaiting read to the reservation station.
- Can support sequence of multiple output dependences.
- Forwarding from FU's to reservation stations bypasses the register file.

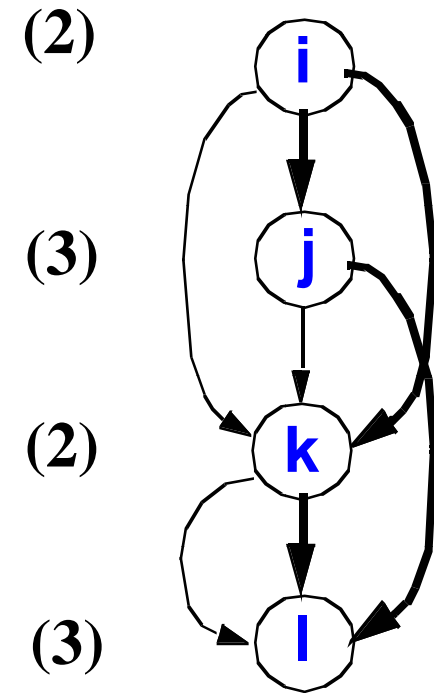
# Tomasulo vs. Modern OOO

	IBM 360/91	Modern
Width		
Structural		
Anti-depend		s
Output depende		ing
True dep		ng
Exception		orw.
Impleme		B)
	>\$1 million	

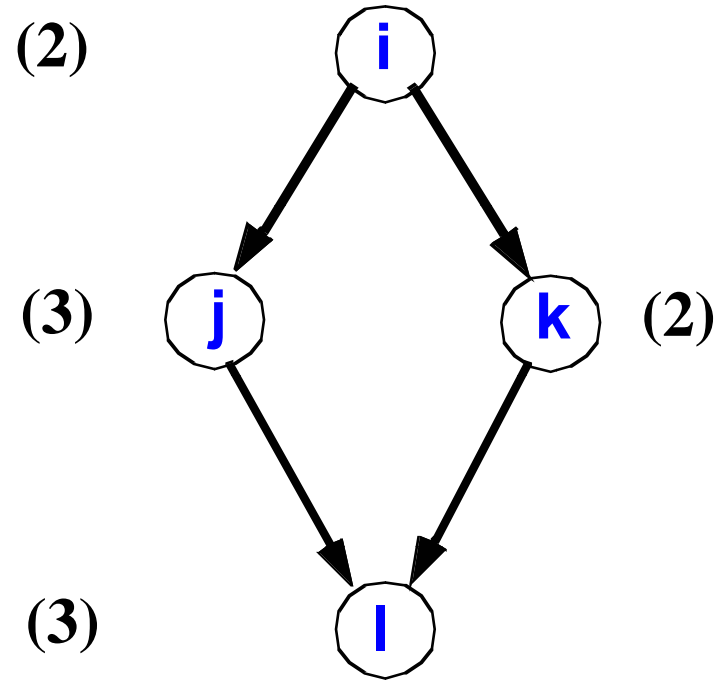
## Example 4



## Example 4



(10)



(8)

Can Tomasulo's algorithm reach dataflow limit of 8?

# Example 4

CYCLE #1

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

BusyTag	Data
0	6.0
2	3.5
4	10.0
8	7.8

DISPATCHED INSTRUCTION(S): \_\_\_\_\_

CYCLE #2

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

BusyTag	Data
0	
2	
4	
8	

DISPATCHED INSTRUCTION(S): \_\_\_\_\_

CYCLE #3

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

BusyTag	Data
0	
2	
4	
8	

DISPATCHED INSTRUCTION(S): \_\_\_\_\_

# Example 4

CYCLE #4

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

BusyTag	Data
0	
2	
4	
8	

DISPATCHED INSTRUCTION(S): \_\_\_\_\_

CYCLE #5

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

BusyTag	Data
0	
2	
4	
8	

DISPATCHED INSTRUCTION(S): \_\_\_\_\_

CYCLE #6

ID	Tag	Sink	Tag	Source
1				
2				
3				

Adder

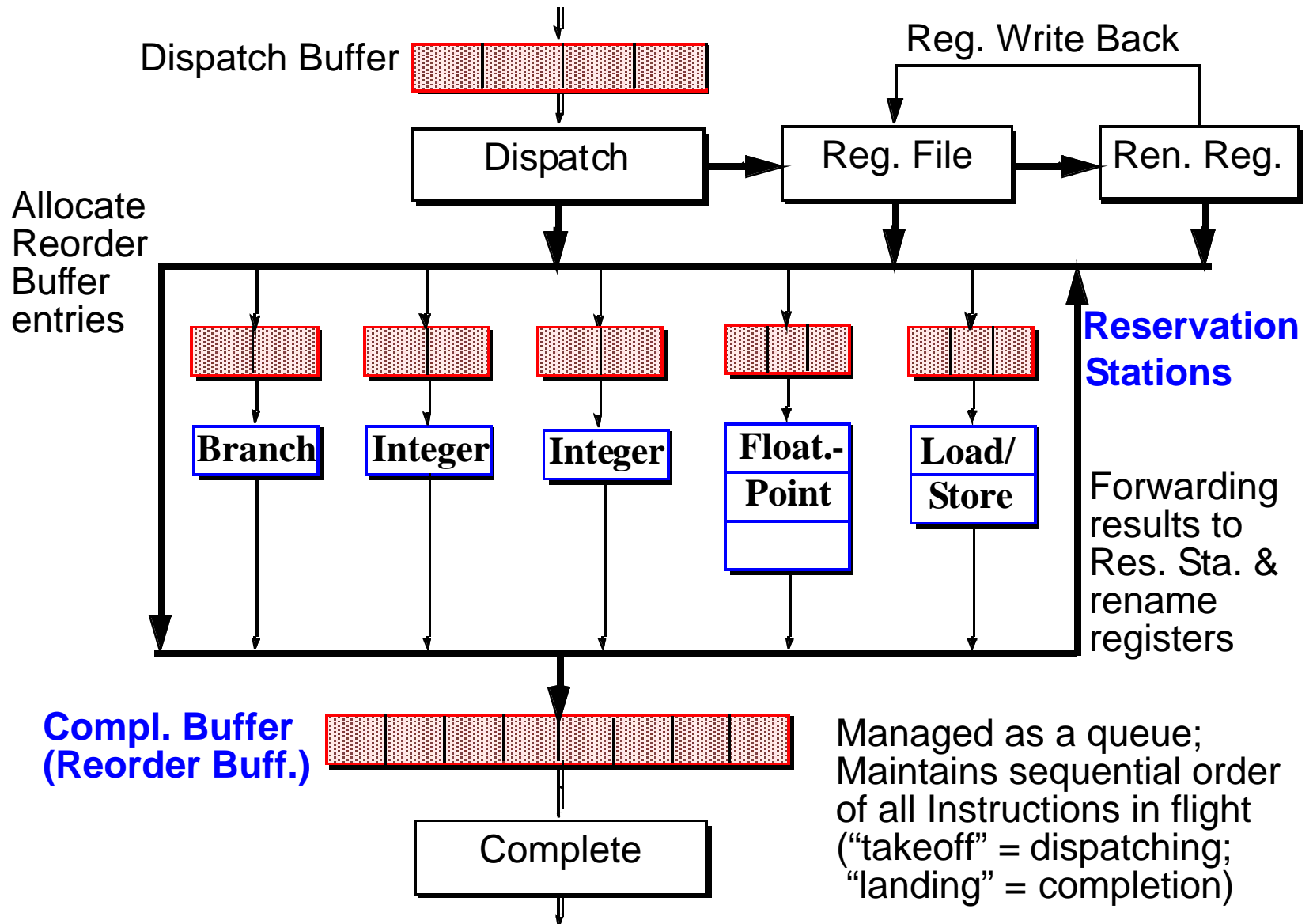
ID	Tag	Sink	Tag	Source
4				
5				

Mult/Div

BusyTag	Data
0	
2	
4	
8	

DISPATCHED INSTRUCTION(S): \_\_\_\_\_

# “Dataflow Engine” for Dynamic Execution





# Instruction Processing Steps

## •DISPATCH:

- Read operands from Register File (RF) and/or Rename Buffers (RRB)
- Rename destination register and allocate RRB entry
- Allocate Reorder Buffer (ROB) entry
- Advance instruction to appropriate Reservation Station (RS)

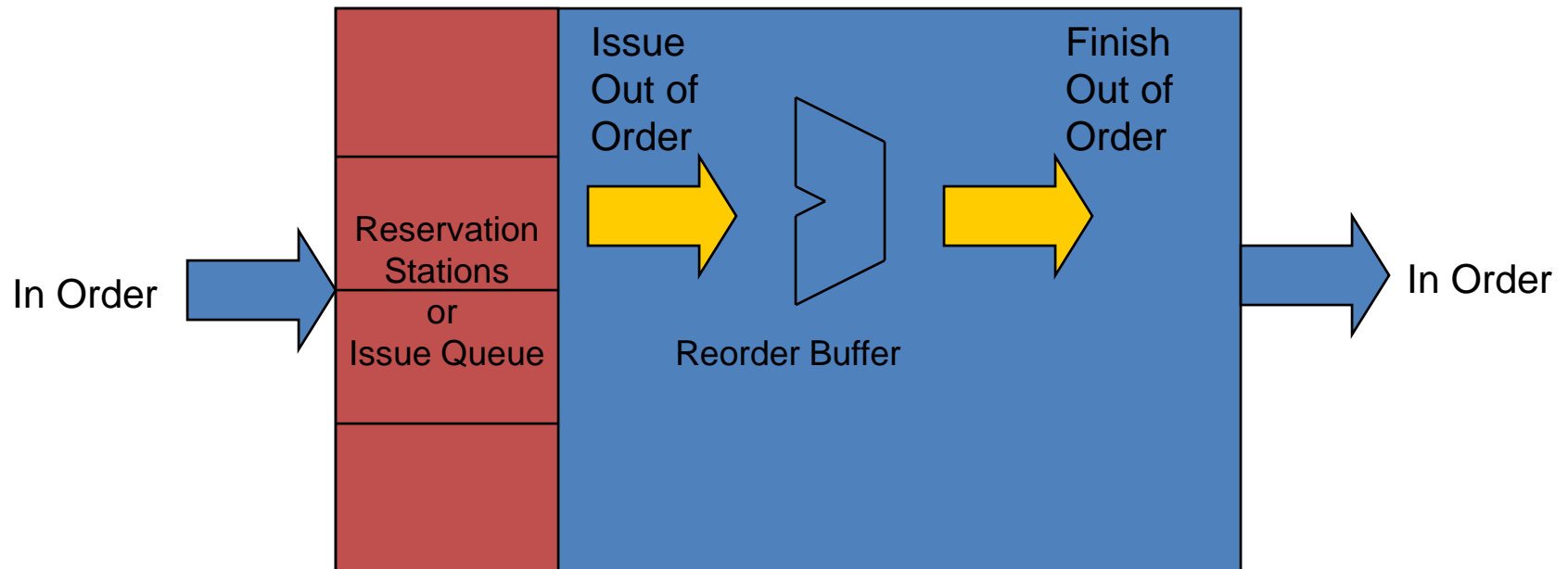
## •EXECUTE:

- RS entry monitors bus for register Tag(s) to latch in pending operand(s)
- When all operands ready, issue instruction into Functional Unit (FU) and deallocate RS entry (no further stalling in execution pipe)
- When execution finishes, broadcast result to waiting RS entries, RRB entry, and ROB entry

## •COMPLETE:

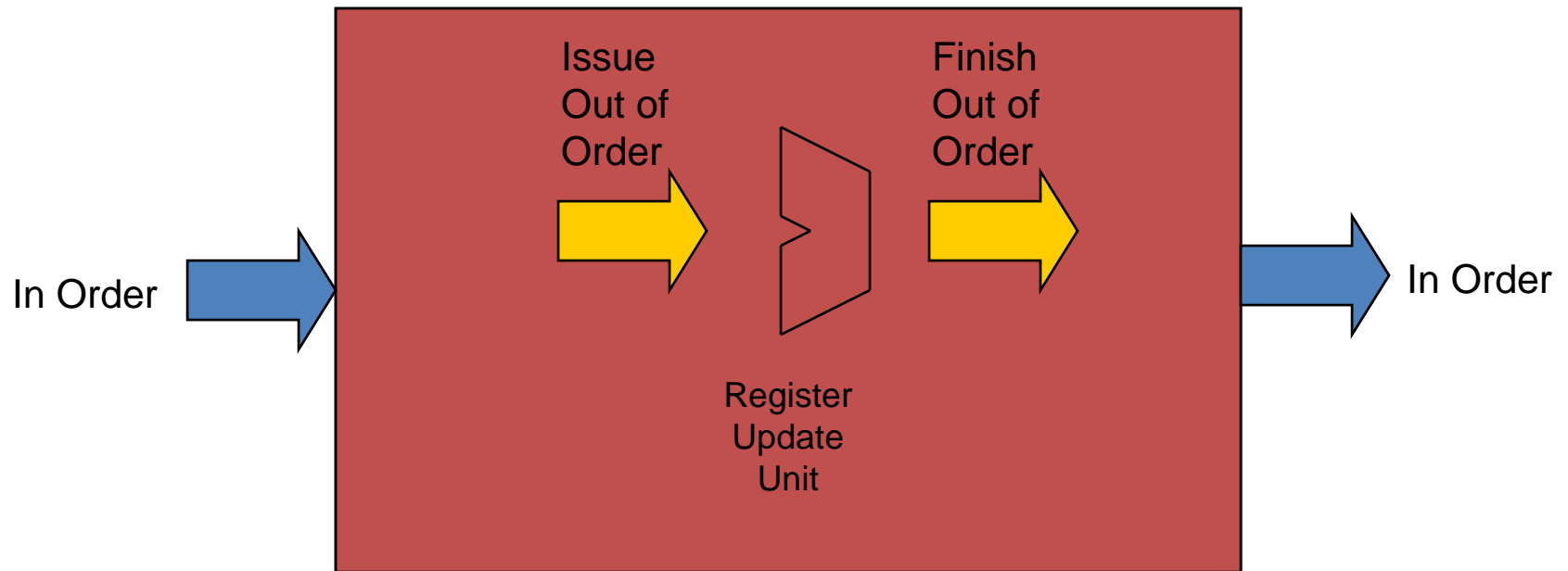
- Update architected register from RRB entry, deallocate RRB entry, and if it is a store instruction, advance it to Store Buffer
- Deallocate ROB entry and instruction is considered architecturally completed

# Reservation Station Implementation



- **Reservation Stations: distributed vs. centralized**
  - Wakeup: benefit to partition across data types
  - Select: much easier with partitioned scheme
    - Select 1 of  $n/4$  vs. 4 of  $n$

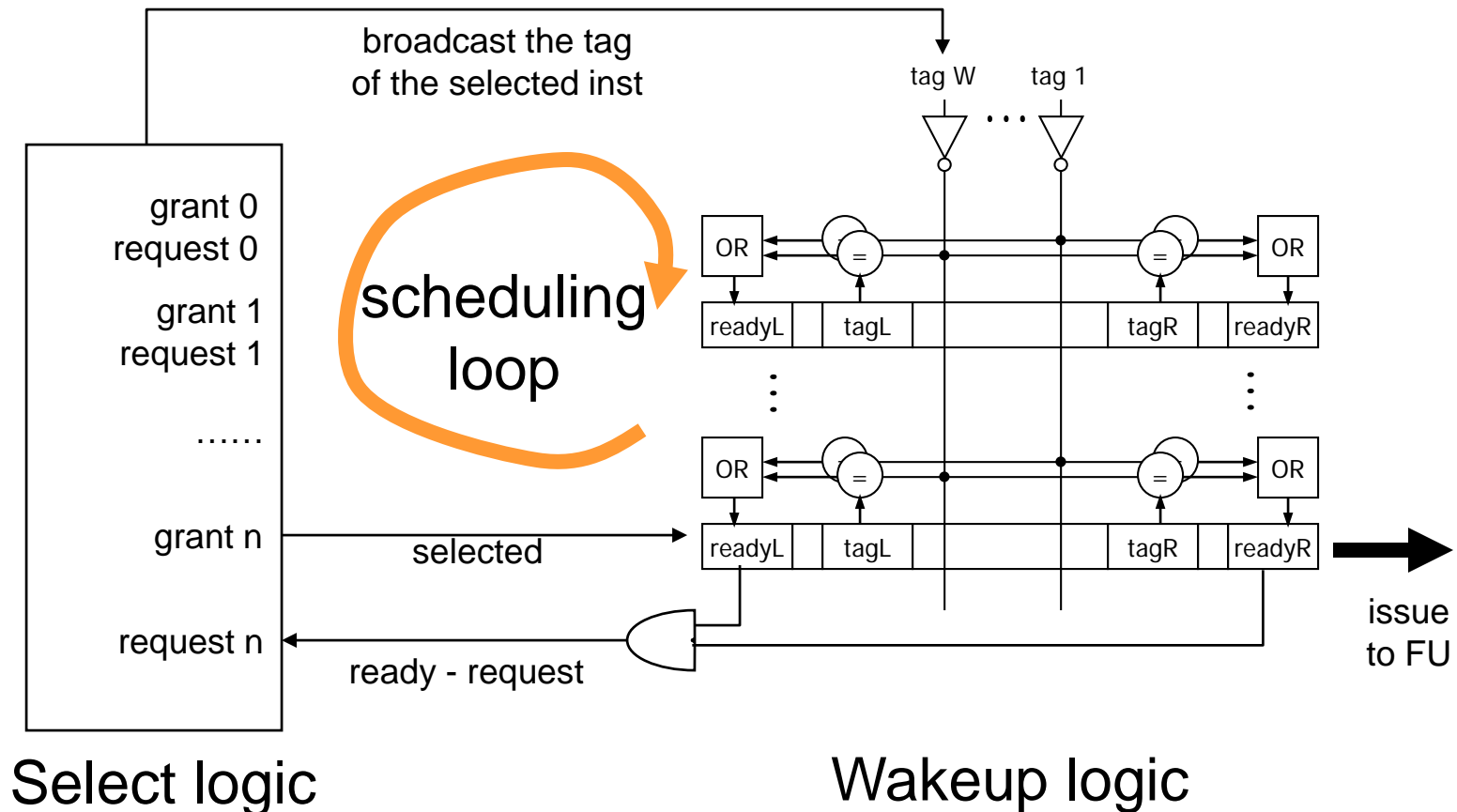
# Reorder Buffer Implementation



- **Merge RS and ROB => Register Update Unit (RUU)**
  - Inefficient, hard to scale
  - Perhaps of interest only to historians

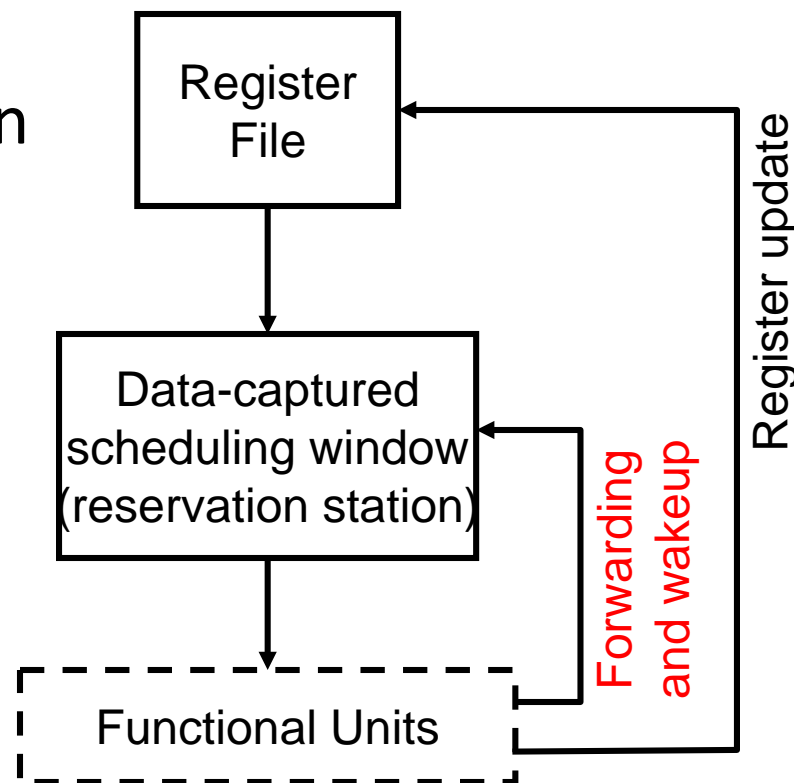
# Scheduling loop

- Wakeup dominated by wire delay
  - More compact RS => higher frequency



# Scheduler Designs

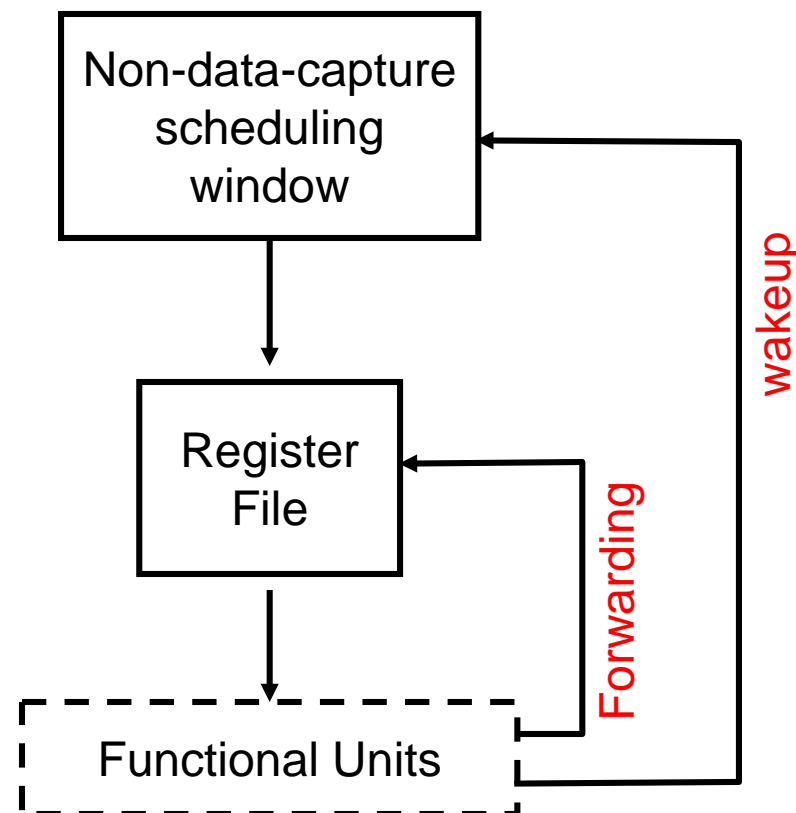
- Data-Capture Scheduler
  - keep the most recent register value in reservation stations
  - Data forwarding and wakeup are combined



# Scheduler Designs

- Non-Data-Capture Scheduler

- keep the most recent register value in RF (physical registers)
- Data forwarding and wakeup are decoupled

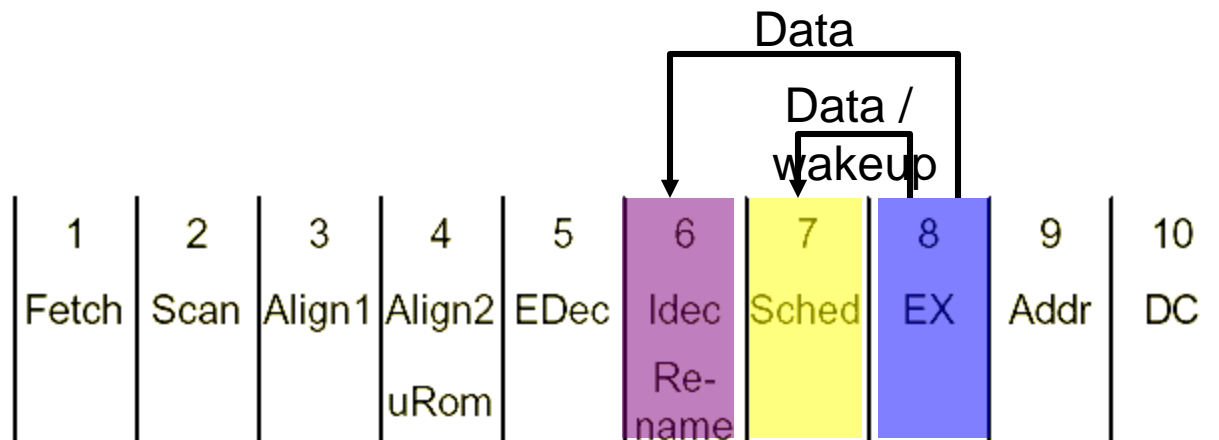


- Complexity benefits

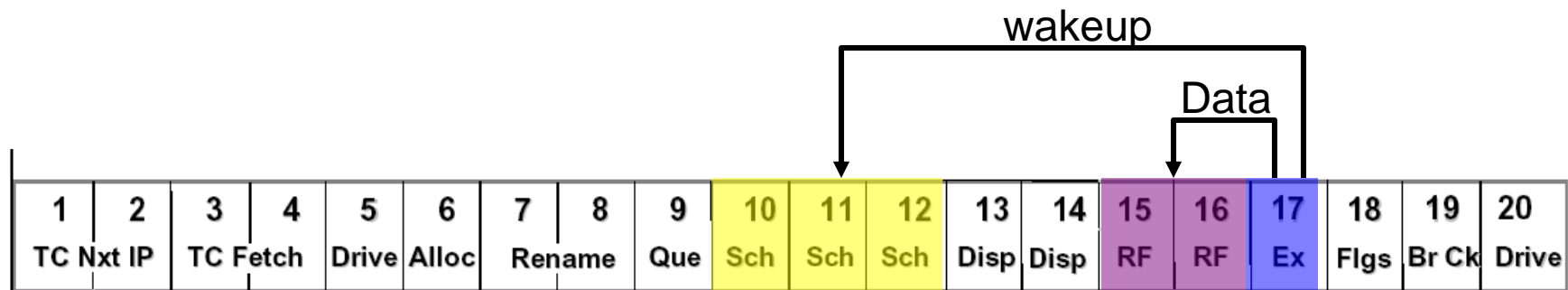
- smaller RS / faster wakeup path

# Mapping to pipeline stages

- AMD K7 (data-capture)

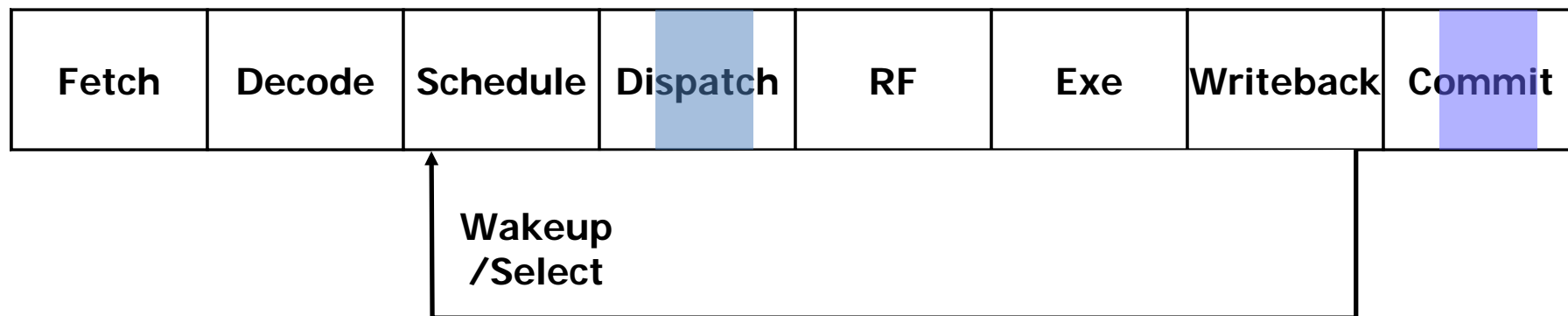


- Pentium 4 (non-data-capture)



# Scheduling atomicity & non-data-capture scheduler

- Multi-cycle scheduling loop



- Scheduling atomicity is not maintained
  - Separated by extra pipeline stages (Disp, RF)
  - Unable to issue dependent instructions consecutively

➔ solution: **speculative scheduling**

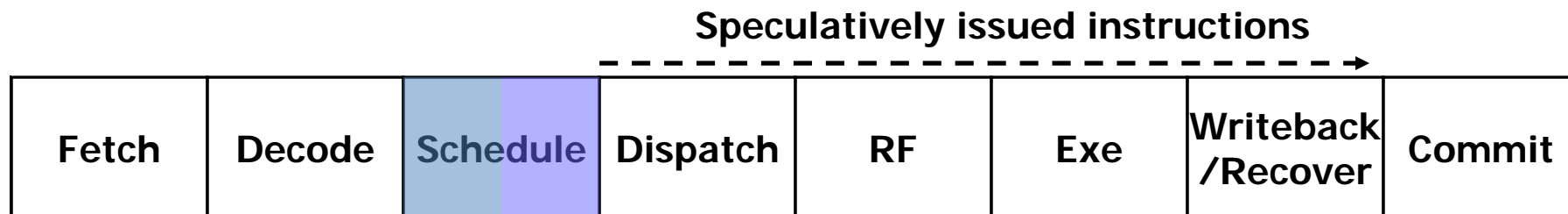


# Speculative Scheduling

- Speculatively wakeup dependent instructions even before the parent instruction starts execution
  - Keep the scheduling loop within a single clock cycle
- But, nobody knows what will happen in the future
- Source of uncertainty in instruction scheduling: loads
  - Cache hit / miss
  - Store-to-load aliasing
  - ➔ eventually affects timing decisions
- Scheduler assumes that all types of instructions have pre-determined fixed latencies
  - Load instructions are assumed to have a common case (over 90% in general) \$DL1 hit latency
  - If incorrect, subsequent (dependent) instructions are replayed

# Speculative Scheduling

- Overview



- Unlike the original Tomasulo's algorithm
  - Instructions are scheduled BEFORE actual execution occurs
  - Assumes instructions have pre-determined fixed latencies
    - ALU operations: fixed latency
    - Load operations: assumes \$DL1 latency (common case)

# Selection Logic

- Single FU of each type: arbiter
  - Choose 1:n ready instructions
  - Fairly straightforward logic
- Multiple FUs of a given type: allocator
  - Choose m:n ready instructions
  - Must solve classic “matching” problem
    - Using e.g. *iSLIP* [McKeown, IEEE Trans. On Networking, 1999]
  - Not feasible in cycle-time critical loop
  - Instead: instructions bound to FU when inserted

# Selection Logic

- Arbiter must guarantee forward progress
  - Select oldest instruction first
  - Algorithm: mask remembers who was waiting when you arrived, defer to them

Entry	Valid	Mask
0	1	0100
1	1	0000
2	1	1100
3	0	0000

1) Entries 2 & 3 wake up: 0011  
 2:  $|(0011 \& 1101)| \Rightarrow \text{true}$   
 3:  $|(0011 \& 0100)| \Rightarrow \text{false}$

2) 3 is selected, flash clears col 3

3) 2 is selected in next cycle, etc.

# Reorder Buffer Implementation

Busy	Issued	Finished	Instruction address	Rename register	Speculative	Valid
------	--------	----------	------------------------	--------------------	-------------	-------

(a)

	Next entry to be allocated (tail pointer)					Next instruction to complete (head pointer)						
	↓					↓						
B	0	0	0	0	0	1	1	1	1	1	1	1
I												
F												
IA												
RR												
S												
V												

Reorder buffer

(b)

- **Reorder Buffer**

- “Bookkeeping,” physically distributed, many read/write ports
- Can be instruction-grained, or block-grained (4-5 ops)

# Register File Speculation & Recovery

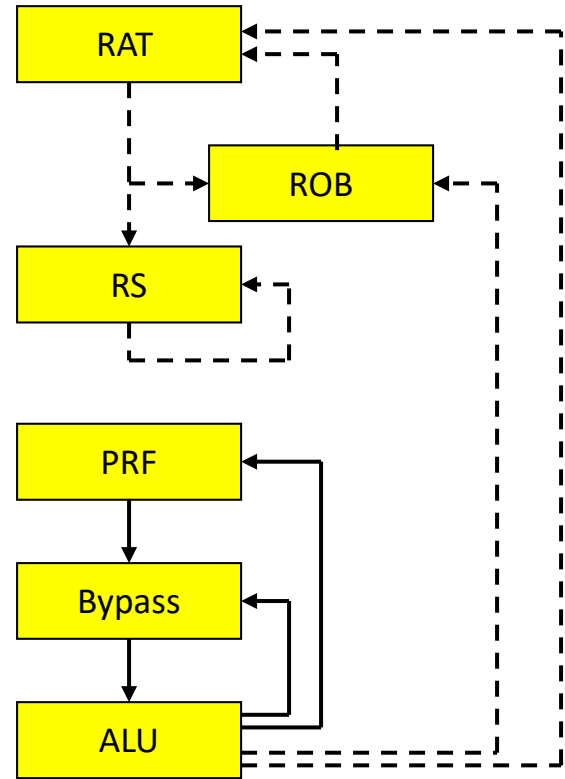
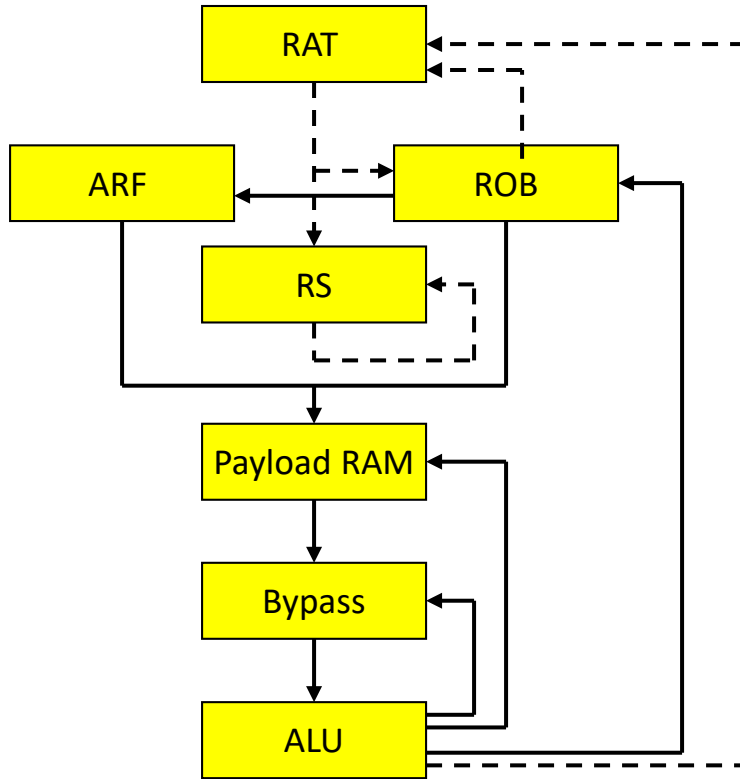
- **History file (similar to checkpointing)**
  - Copy previous value from ARF to HF at dispatch
  - Use HF to reconstruct precise state if needed
- **Future file: separate ARF & RRF (lecture notes, PPC 620, Pentium Pro, Core i7, AMD K8, ARM A15)**
  - Copy committed value from RRF to ARF
  - Update rename table mapping
- **Physical Register File: merged ARF & RRF (MIPS R10000 , Pentium 4, Alpha 21264, Power 4-7, Nehalem, Sandybridge, Bulldozer, Bobcat)**
  - No copy; simpler datapath (operand always in PRF)
  - Simply “commit” rename table mapping as branches resolve

# Register File Alternatives

Register Lifetime	Status	Duration (cycles)	Result stored where?		
			Future File	History File	Phys. RF
Dispatch	Unavail	$\geq 1$	N/A	N/A	N/A
Finish execution	Speculative	$\geq 0$	FF	ARF	PRF
Commit	Committed	$\geq 0$	ARF	ARF	PRF
Next def. Dispatched	Committed	$\geq 1$	ARF	HF	PRF
Next def. Committed	Discarded	$\geq 0$	Overwritten	Discarded	Reclaimed

- Rename register organization
  - Future file (future updates buffered, later committed)
    - Rename register file
  - History file (old versions buffered, later discarded)
  - **Merged (single physical register file)**

# ARF vs. PRF



## Architected Register File -style

- We showed that PRF is better [ISLPED 07] – nearly everyone now agrees!
- P6 thru Core 2 Duo (Merom): ARF
- Pentium4/Nehalem/Sandybridge, AMD Bulldozer & Bobcat: PRF
- Recent regression: Intel Silvermont, Cortex A15 use ARF ... design team?

## Physical Register File - style



# Misprediction Recovery

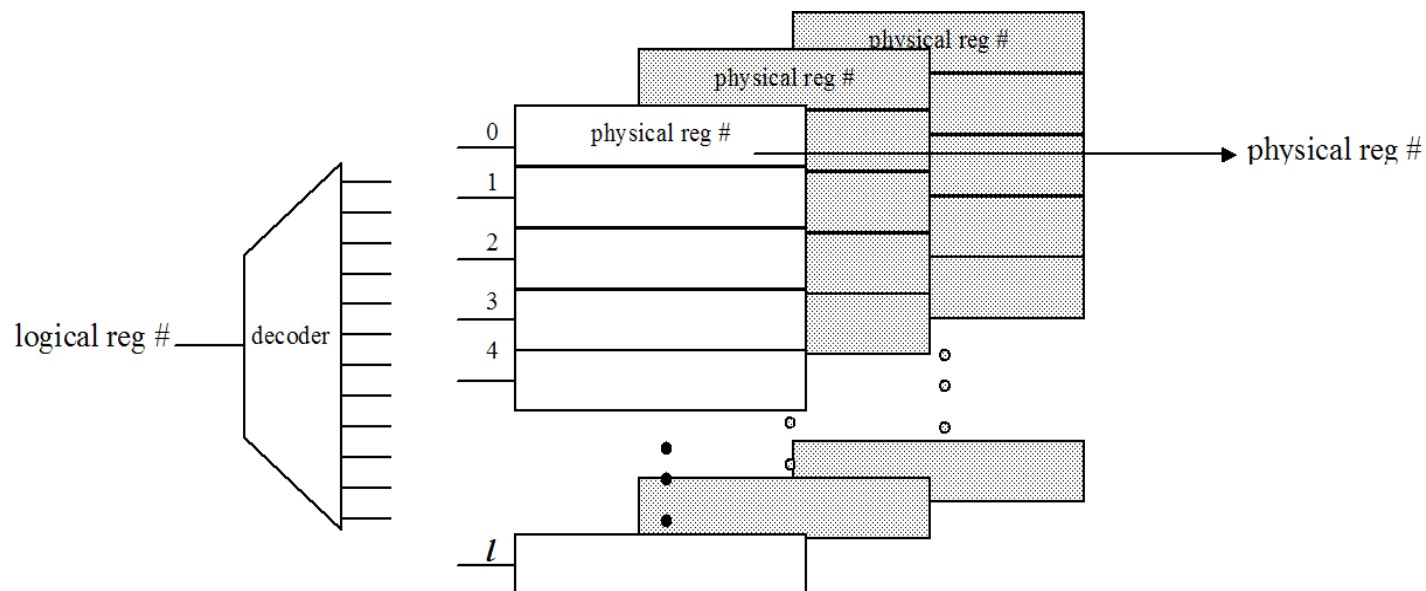
Valid	PC	Dest PR	Prev PR	Src1	Src2	Imm/target	Issued	Executed	Exception	T/NT Pred
1	x400C	P13	P17	P25	n/a	X80	Y	Y	N	
1	X4008	P14	P22	P31	P5		Y	N	N	
1	X4004					x4020				NT

- Branch mispredicts, exceptions: must reclaim allocated resources
  - Load queue, store queue/color, branch color, ROB entry, rename register
- Can reclaim implicitly
  - Tag broadcast: all entities match & release
  - Too expensive for physical register file (PRF)
- Or reclaim explicitly
  - Walk through ROB which contains pointers
  - Follow pointers to release resources
- Also, recover rename mappings
  - Read previous mappings (pending release) and repair map table

# Rename Table Implementation

- MAP checkpointing
  - Performance optimization
    - Recovery from branches, exceptions
  - Checkpoint granularity
    - Every instruction (costly)
    - Every branch, play back ROB to get to exception boundary
- RAM vs CAM Map Table

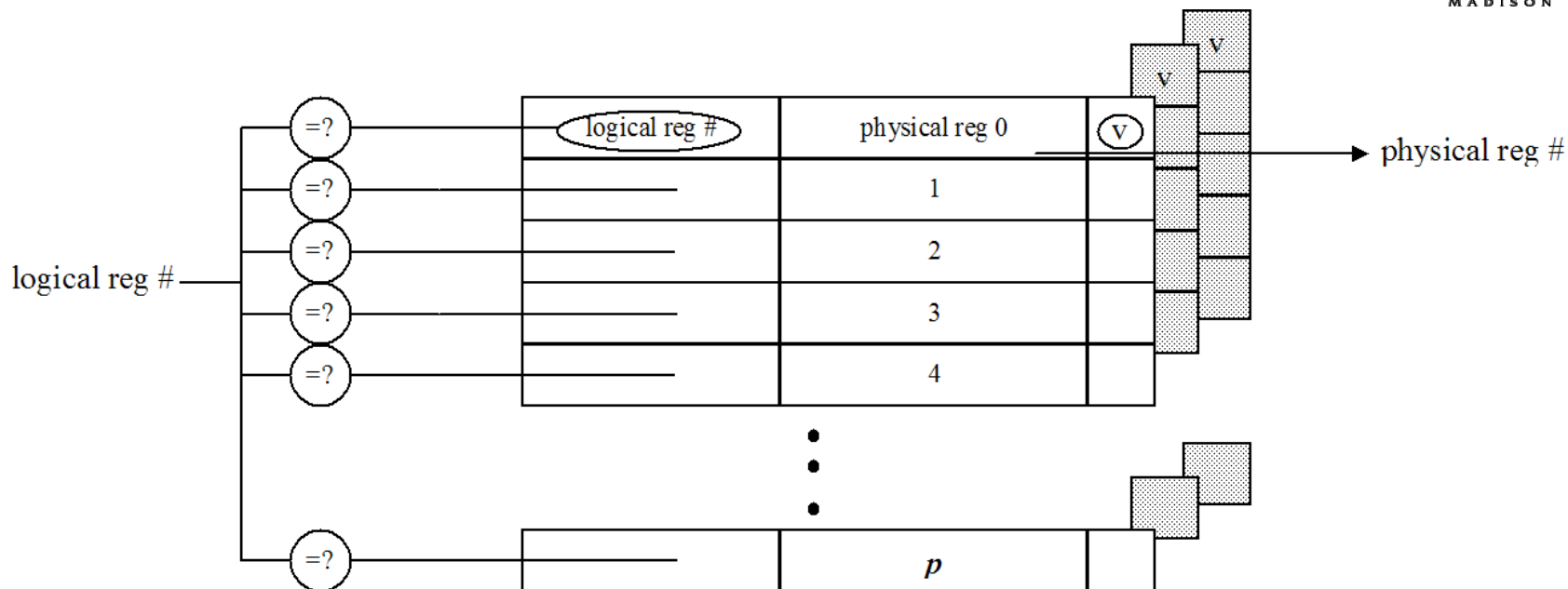
# RAM Map Table



- Just a lookup table

Checkpoint size:  $n$  (# arch reg)  $\times \log_2(\text{phys reg})$

# CAM Map Table



- CAM search for mappings
  - # rows == number of physical registers
  - Checkpoint only the valid bit column
- Used in Alpha 21264

# Summary

- Register dependences
  - True dependences
  - Antidependences
  - Output dependences
- Register Renaming
- Tomasulo's Algorithm
- Reservation Station Implementation
- Reorder Buffer Implementation
- Register File Implementation
  - History file
  - Future file
  - Physical register file
- Rename Table Implementation