ARCHITECTURE OF COMPUTER SYSTEMS LECTURE 10 - COMPLEX PIPELINES, OUT-OF-ORDER ISSUE, REGISTER RENAMING

LAST TIME IN LECTURE 9

- Modern page-based virtual memory systems provide:
  - Translation, Protection, Virtual memory.
- Translation and protection information stored in page tables, held in main memory
- Translation and protection information cached in "translation-lookaside buffer" (TLB) to provide single-cycle translation+protection check in common case
- Virtual memory interacts with cache design
  - Physical cache tags require address translation before tag lookup, or use untranslated offset bits to index cache.
  - Virtual tags do not require translation before cache hit/miss determination, but need to be flushed or extended with ASID to cope with context swaps. Also, must deal with virtual address aliases (usually by disallowing copies in cache).

COMPLEX PIPELINING: MOTIVATION

Pipelining becomes complex when we want high performance in the presence of:

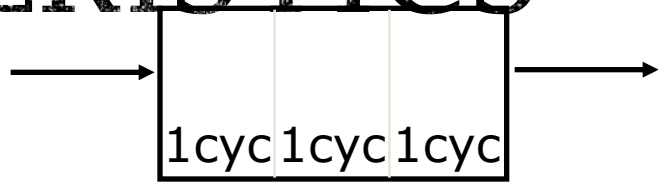
- Long latency or partially pipelined floating-point units
- Memory systems with variable access time
- Multiple arithmetic and memory units

FLOATING-POINT UNIT (FPU)

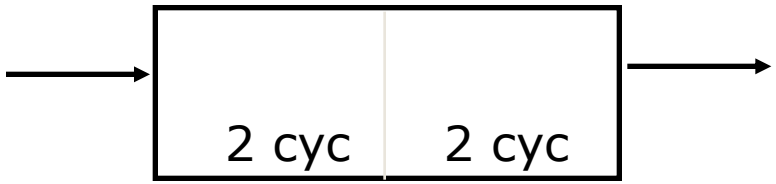
- Much more hardware than an integer unit
  - Single-cycle FPU is a bad idea – why?
- Common to have several FPU's
- Common to have different types of FPU's: Fadd, Fmul, Fdiv, …
- An FPU may be pipelined, partially pipelined or not pipelined
- To operate several FPU's concurrently the FP register file needs to have more read and write ports

FUNCTIONAL UNIT CHARACTERISTICS

fully pipelined



partially pipelined



Functional units have internal pipeline registers

⇒ operands are latched when an instruction enters a functional unit

⇒ following instructions are able to write register file during a long-latency operation

FLOATING-POINT ISA

- Interaction between floating point datapath and integer datapath is determined by ISA
- RISC-V ISA
  - separate register files for FP and Integer instructions
    - the only interaction is via a set of move/convert instructions (some ISA's don't even permit this)
  - separate load/store for FPR's and GPR's but both use GPR's for address calculation
  - FP compares write integer registers, then use integer branch

REALISTIC MEMORY SYSTEMS

Common approaches to improving memory performance:

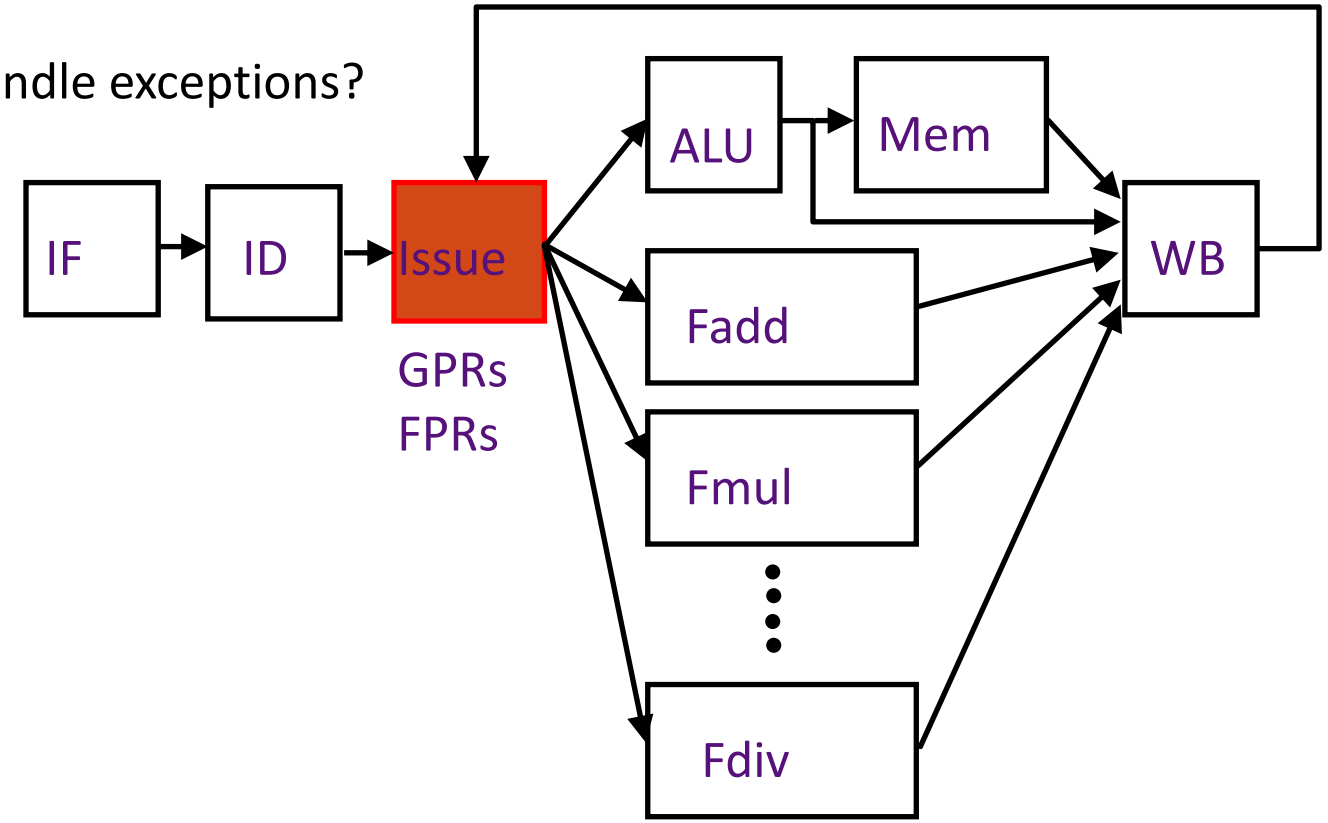
- Caches - single cycle except in case of a miss
  - stall
- Banked memory - multiple memory accesses
  - bank conflicts
- split-phase memory operations (separate memory request from response), many in flight
  - out-of-order responses

Latency of access to the main memory is usually much greater than one cycle and often unpredictable

Solving this problem is a central issue in computer architecture

ISSUES IN COMPLEX PIPELINE CONTROL

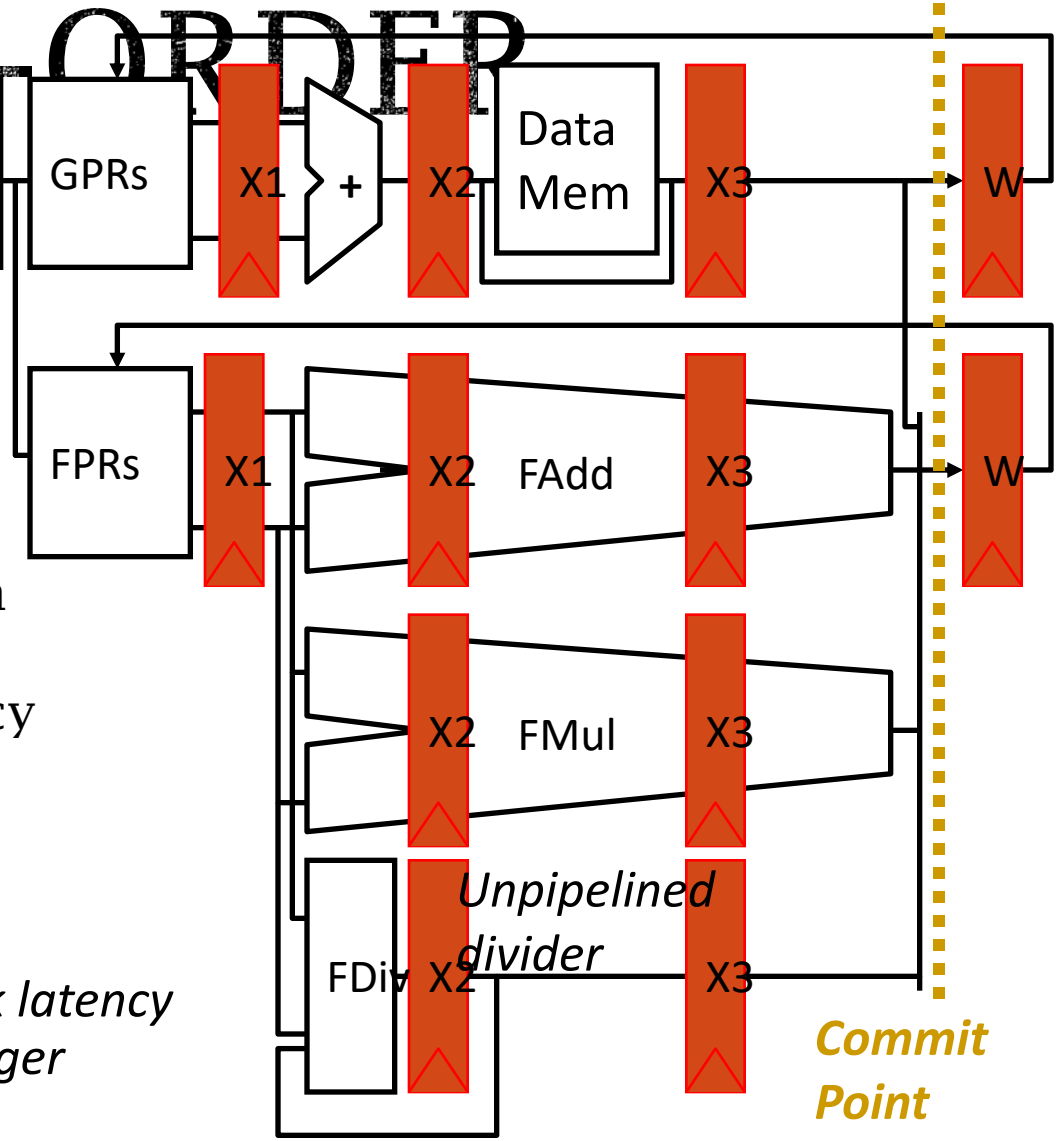
- Structural conflicts at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle
- Structural conflicts at the write-back stage due to variable latencies of different functional units
- Out-of-order write hazards due to variable latencies of different functional units
- How to handle exceptions?



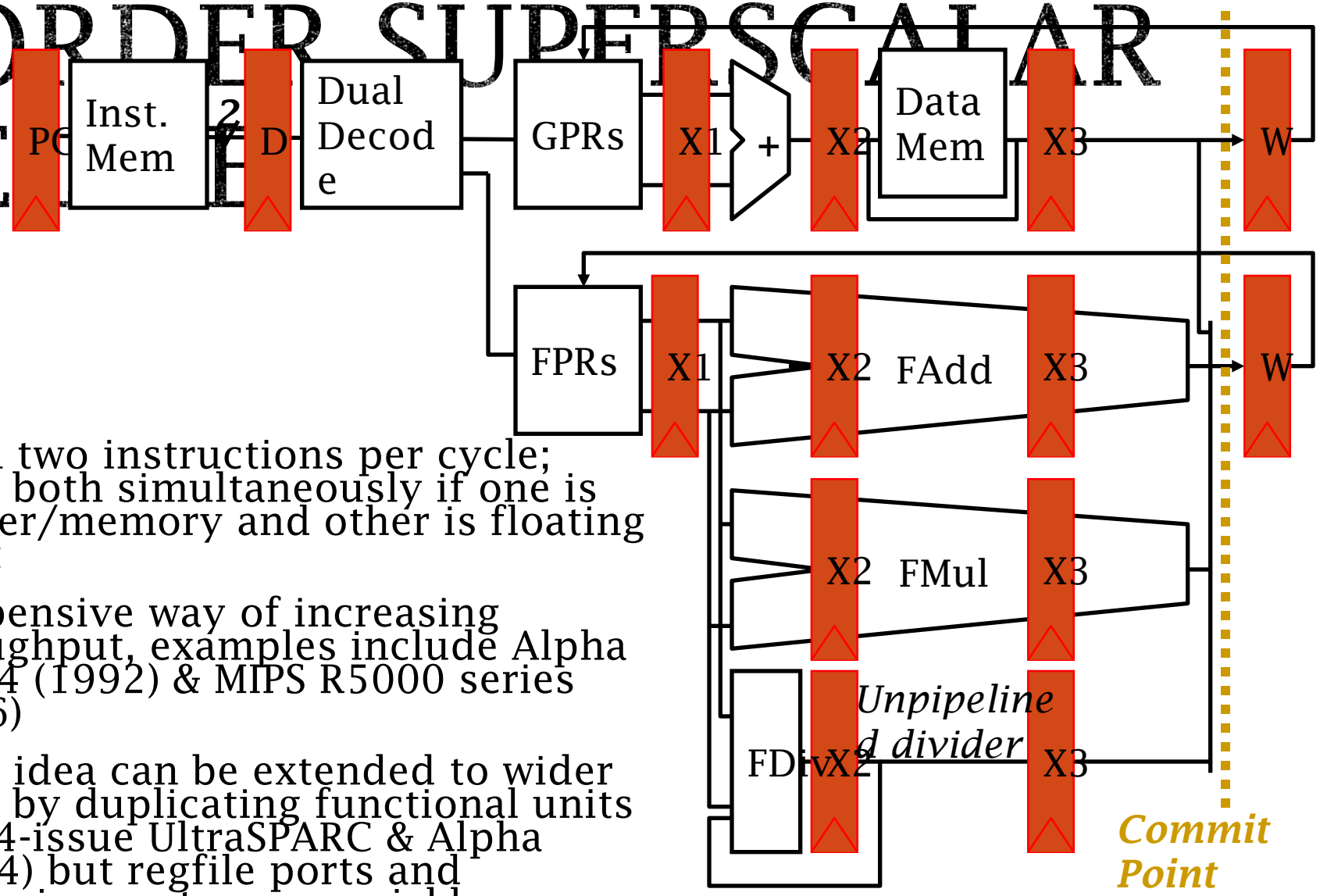
COMPLEX IN-ORDER PIPELINE

- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
  - Stall pipeline on long latency operations, e.g., divides, cache misses
  - Handle exceptions in-order at commit point

How to prevent increased writeback latency from slowing down single cycle integer operations? Bypassing



IN-ORDER SUPERSCALAR PIPELINE



- Fetch two instructions per cycle; issue both simultaneously if one is integer/memory and other is floating point
- Inexpensive way of increasing throughput, examples include Alpha 21064 (1992) & MIPS R5000 series (1996)
- Same idea can be extended to wider issue by duplicating functional units (e.g. 4-issue UltraSPARC & Alpha 21164) but regfile ports and bypassing costs grow quickly

TYPES OF DATA HAZARDS

Consider executing a sequence of

$r_k \leftarrow r_i \ op \ r_j$

type of instructions

Data-dependence

$r_3 \leftarrow r_1$ op r_2 Read-after-Write

$r_5 \leftarrow r_3$ op r_4 (RAW) hazard

Anti-dependence

$r_3 \leftarrow r_1$ op r_2 Write-after-Read

$r_1 \leftarrow r_4$ op r_5 (WAR) hazard

Output-dependence

$r_3 \leftarrow r_1$ op r_2 Write-after-Write

$r_3 \leftarrow r_6$ op r_7 (WAW) hazard

REGISTER VS. MEMORY DEPENDENCE

Data hazards due to register operands can be determined at the decode stage, but data hazards due to memory operands can be determined only after computing the effective address

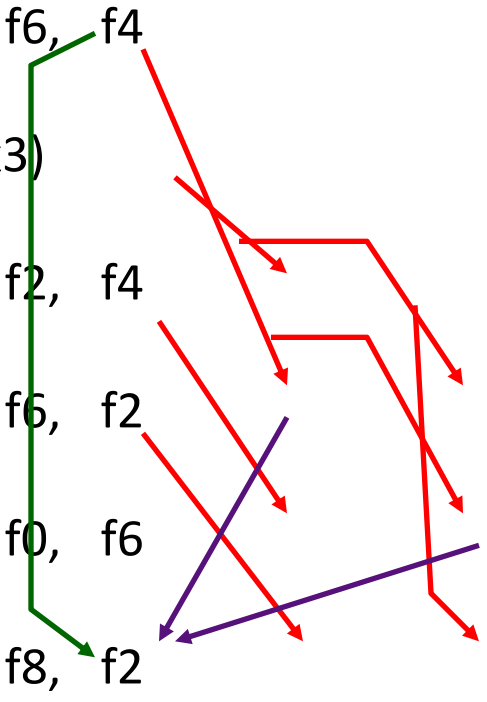
Store: **M[r1 + disp1] □ r2**

Load: **r3 □ M[r4 + disp2]**

Does **(r1 + disp1) = (r4 + disp2)** ?

DATA HAZARDS: AN EXAMPLE

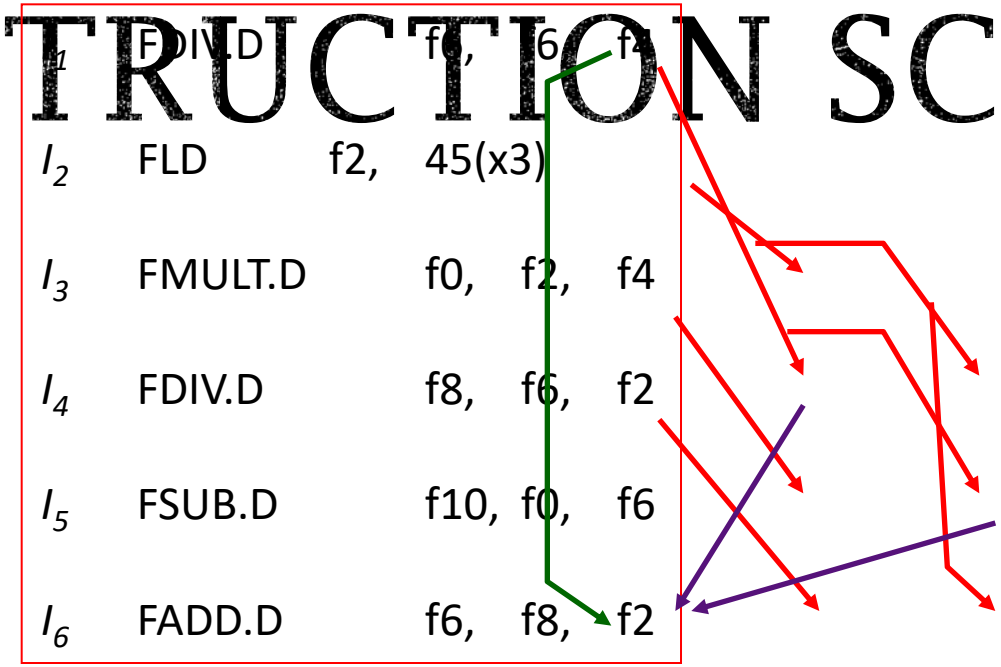
```
I1   FDIV.D    f6,   f6,   f4
I2   FLD       f2,   45(x3)
I3   FMUL.D    f0,   f2,   f4
I4   FDIV.D    f8,   f6,   f2
I5   FSUB.D    f10,  f0,   f6
I6   FADD.D    f6,   f8,   f2
```

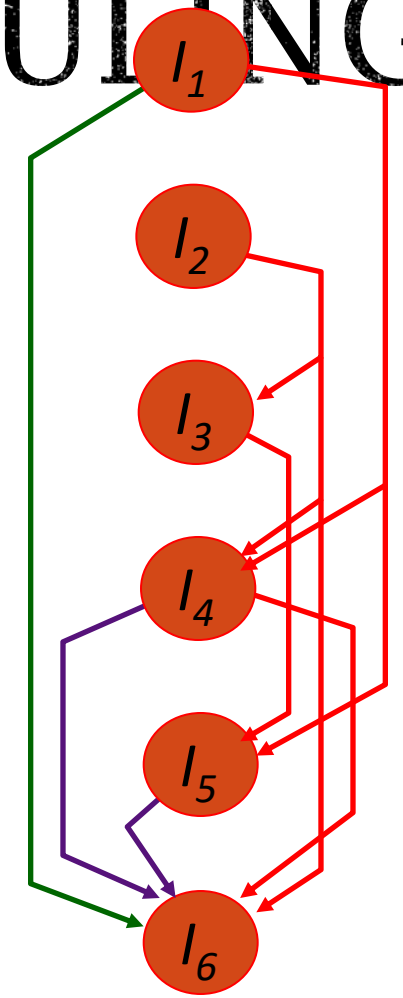


RAW Hazards
WAR Hazards
WAW Hazards

INSTRUCTION SCHEDULING

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





Valid orderings:

in-order I_1 I_2 I_3 I_4 I_5 I_6

out-of-order I_2 I_1 I_3 I_4 I_5 I_6

out-of-order I_1 I_2 I_3 I_5 I_4 I_6

OUT-OF-ORDER COMPLETION

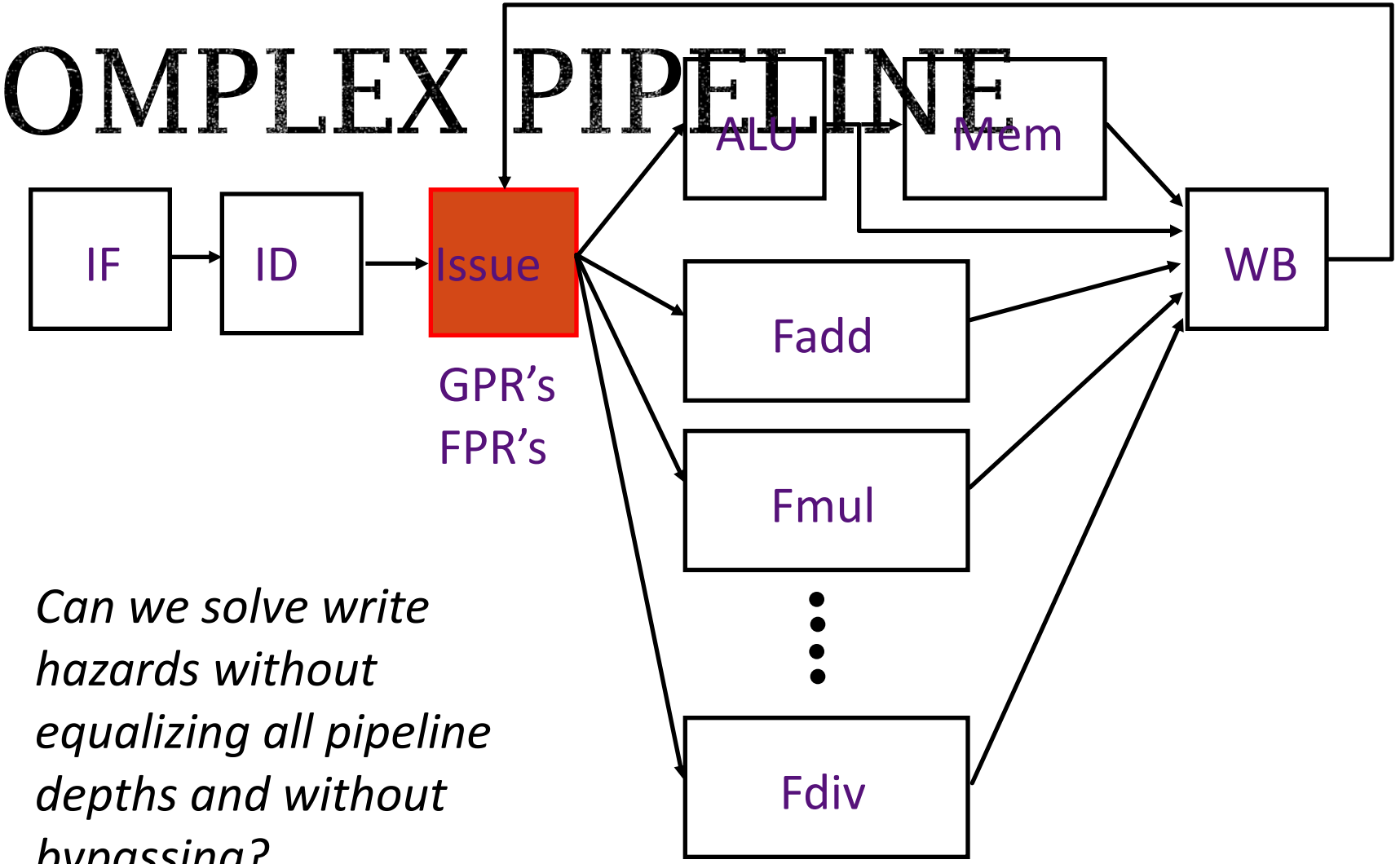
IN-ORDER ISSUE

					Latency
I_1	FDIV.D	f6,	f6,	f4	*4*
I_2	FLD	f2,	45(x3)		*1*
I_3	FMULT.D	f0,	f2,	f4	*3*
I_4	FDIV.D	f8,	f6,	f2	*4*
I_5	FSUB.D	f10,	f0,	f6	*1*
I_6	FADD.D	f6,	f8,	f2	*1*

in-order comp 1 2 1 2 3 4 3 5 4 6 5 6

out-of-order comp 1 2 2 3 1 4 3 5 5 4 6 6

COMPLEX PIPELINE



Can we solve write hazards without equalizing all pipeline depths and without bypassing?

WHEN IS IT SAFE TO ISSUE AN INSTRUCTION?

Suppose a data structure keeps track of all the instructions in all the functional units

The following checks need to be made before the Issue stage can dispatch an instruction

- Is the required function unit available?
- Is the input data available? □□□ RAW?
- Is it safe to write the destination? □□□WAR?□ WAW?
- Is there a structural conflict at the WB stage?

A DATA STRUCTURE FOR CORRECT ISSUES

KEEPS TRACK OF THE STATUS OF FUNCTIONAL UNITS

Name	*Busy*	*Op*	*Dest*	*Src1*	*Src2*
Int					
Mem					
Add1					
Add2					
Add3					
Mult1					
Mult2					
Div					

The instruction i at the Issue stage consults this table

FU available? check the busy column
RAW? search the dest column for i's sources
WAR? search the source columns for i's destination
WAW? search the dest column for i's destination

An entry is added to the table if no hazard is detected;
An entry is removed from the table after Write-Back

STRUCTURE ASSUMING IN-ORDER ISSUE

Suppose the instruction is not dispatched by the Issue stage if a RAW hazard exists or the required FU is busy, and that operands are latched by functional unit on issue:

Can the dispatched instruction cause a

WAR hazard ?

NO: Operands read at issue

WAW hazard ?

YES: Out-of-order completion

SIMPLIFYING THE DATA STRUCTURE …

- No WAR hazard
  - no need to keep src1 and src2
- The Issue stage does not dispatch an instruction in
- case of a WAW hazard
  - a register name can occur at most once in the dest column
- WP[reg#] : a bit-vector to record the registers for which writes are pending
  - These bits are set by the Issue stage and cleared by the WB stage

  Each pipeline stage in the FU's must carry the dest field and a flag to indicate if it is valid “the (we, ws) pair”

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]

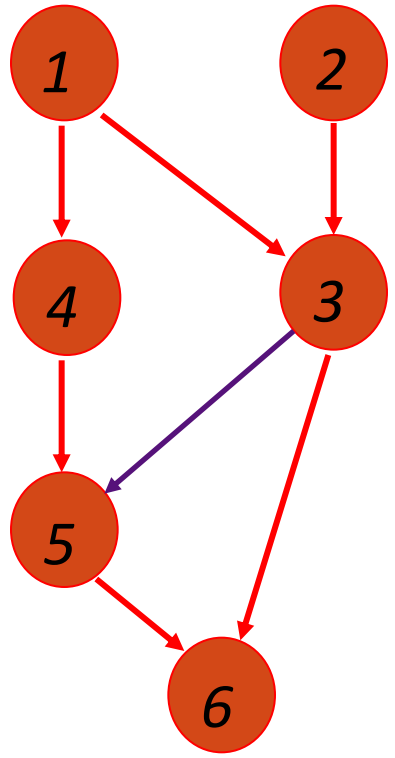
SCOREBOARD DYNAMICS

	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

IN-ORDER ISSUE LIMITATIONS: *AN EXAMPLE*

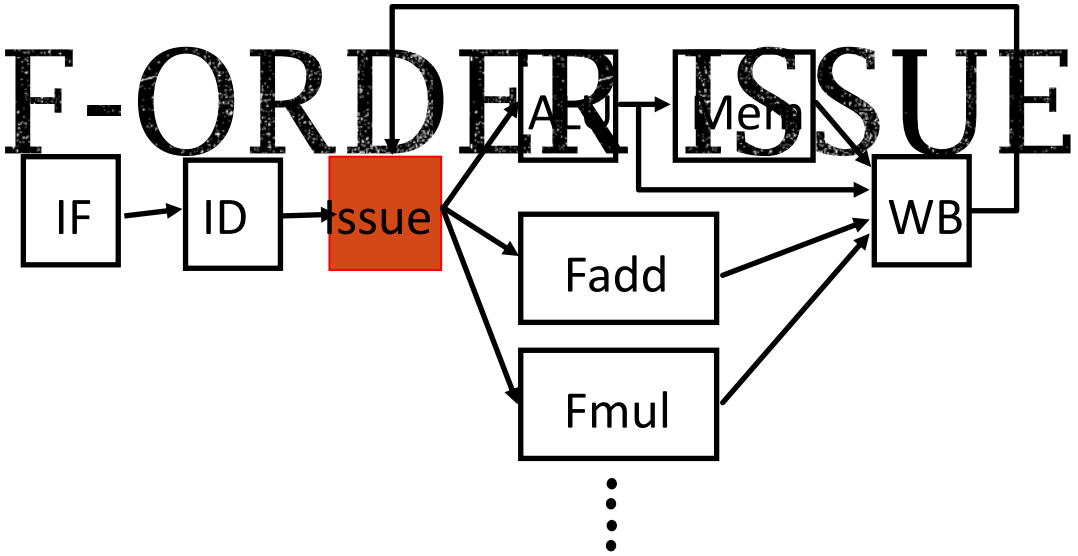
					latency
1	FLD	f2,	34(x2)		*1*
2	FLD	f4,	45(x3)		*long*
3	FMULT.D	f6,	f4,	f2	*3*
4	FSUB.D	f8,	f2,	f2	*1*
5	FDIV.D	f4,	f2,	f8	*4*
6	FADD.D	f10,	f6,	f4	*1*



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

In-order issue restriction prevents instruction 4 from being dispatched

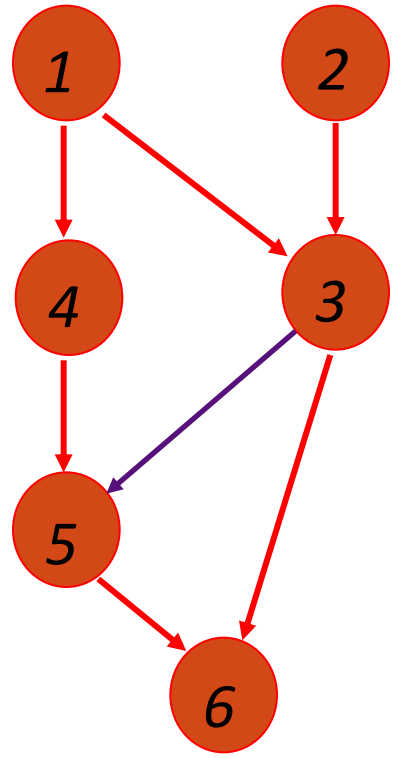
OUT-OF-ORDER ISSUE



- Issue stage buffer holds multiple instructions waiting to issue.
- Decode adds next instruction to buffer if there is space and the instruction does not cause a WAR or WAW hazard.
  - Note: WAR possible again because issue is out-of-order (WAR not possible with in-order issue and latching of input operands at functional unit)
- Any instruction in buffer whose RAW hazards are satisfied can be issued (for now at most one dispatch per cycle). On a write back (WB), new instructions may get enabled.

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 . . . . 2 3 . . 3 5 . . . 5 6 6

Out-of-order execution did not allow any significant improvement!

HOW MANY INSTRUCTIONS CAN BE IN THE PIPELINE?

Which features of an ISA limit the number of instructions in the pipeline?

Number of Registers

Out-of-order dispatch by itself does not provide any significant performance improvement!

OVERCOMING THE LACK OF REGISTER NAMES

Floating Point pipelines often cannot be kept filled with small number of registers.

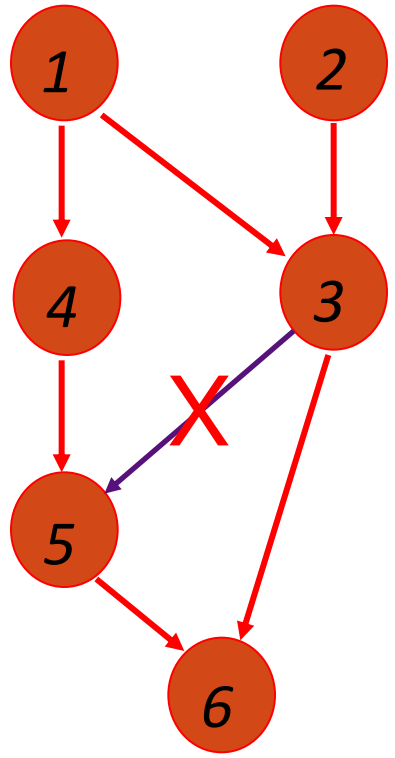
IBM 360 had only 4 floating-point registers

Can a microarchitecture use more registers than specified by the ISA without loss of ISA compatibility ?

Robert Tomasulo of IBM suggested an ingenious solution in 1967 using on-the-fly *register renaming*

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

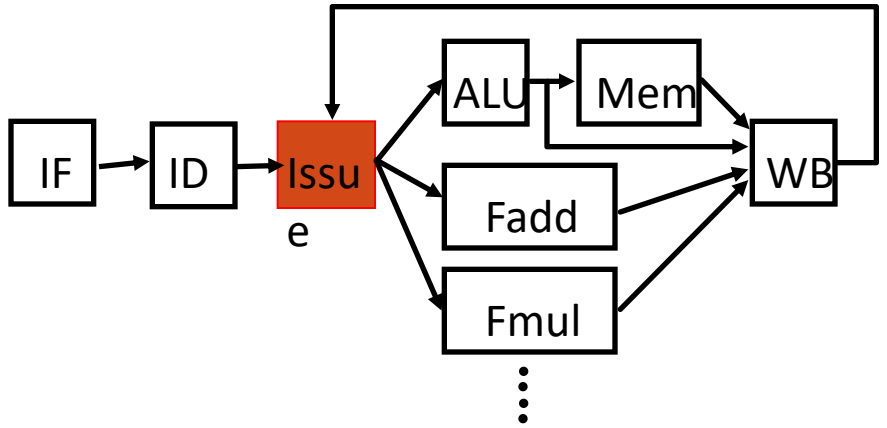
Out-of-order: 1 (2,1) 4 4 5 . . . 2 (3,5) 3 6 6

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

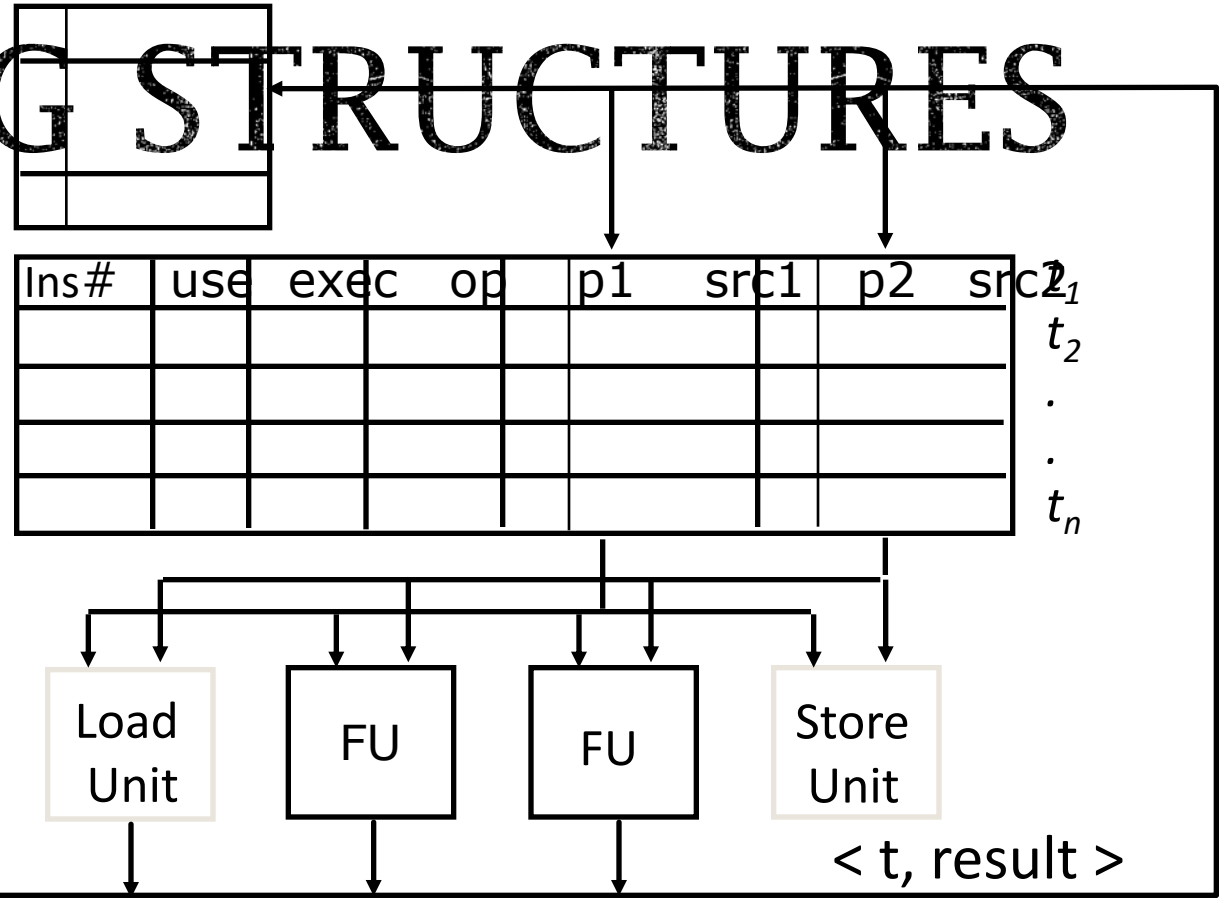
- Any instruction in ROB whose RAW hazards have been satisfied can be dispatched.

  → Out-of-order or dataflow execution

RENAMING STRUCTURES

Renaming table & regfile

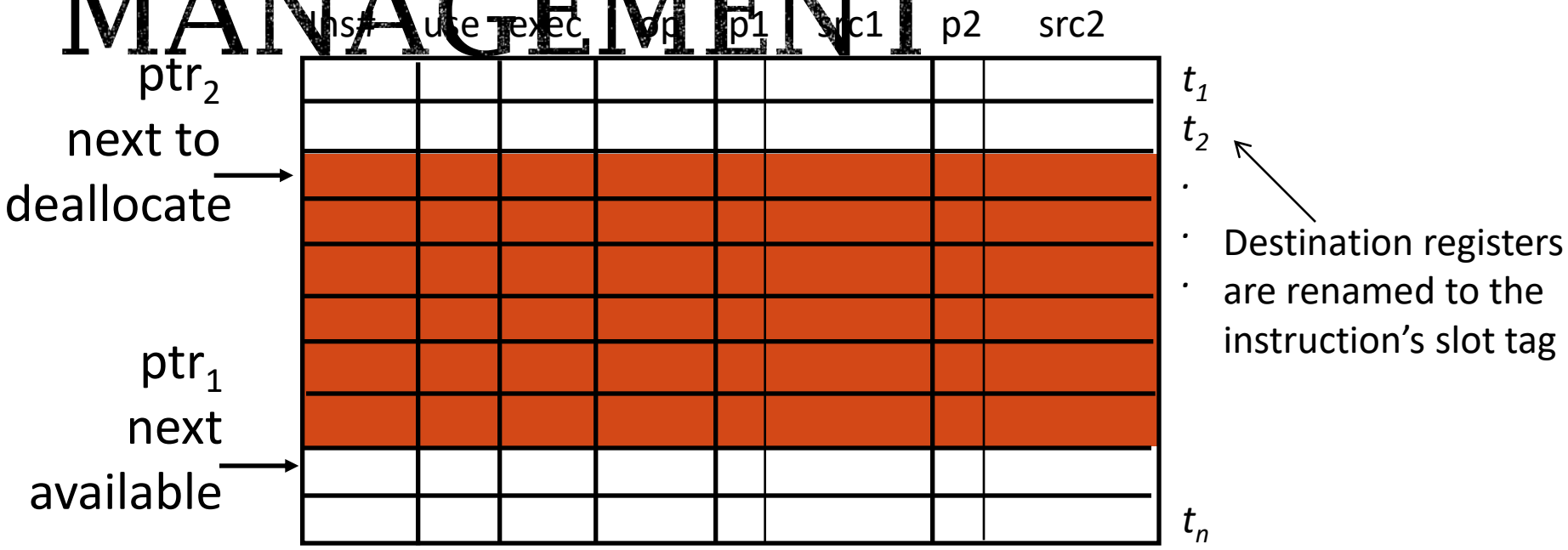
Reorder buffer



Replacing the tag by its value is an expensive operation

- Instruction template (i.e., tag t) is allocated by the Decode stage, which also associates tag with register in regfile
- When an instruction completes, its tag is deallocated

REORDER BUFFER MANAGEMENT



ROB managed circularly

- "exec" bit is set when instruction begins execution
- When an instruction completes its "use" bit is marked free
- ptr_2 is incremented only if the "use" bit is marked free

Instruction slot is candidate for execution when:

- It holds a valid instruction ("use" bit is set)
- It has not already started execution ("exec" bit is clear)
- Both operands are available (p1 and p2 are set)

RENAMING & OUT-OF-ORDER ISSUE

AN EXAMPLE

Renaming table

	p	data
f1		
f2		v1
f3		
f4		t5
f5		
f6		t3
f7		
f8		t4

v1

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2	
1	0	0	LD					t_1
2	1	0	LD					t_2
3	1	0	MUL	0	t2	1	v1	t_3
4	1	0	SUB	1	v1	1	v1	t_4
5	1	0	DIV	1	v1	0	t4	t_5
								.
								.

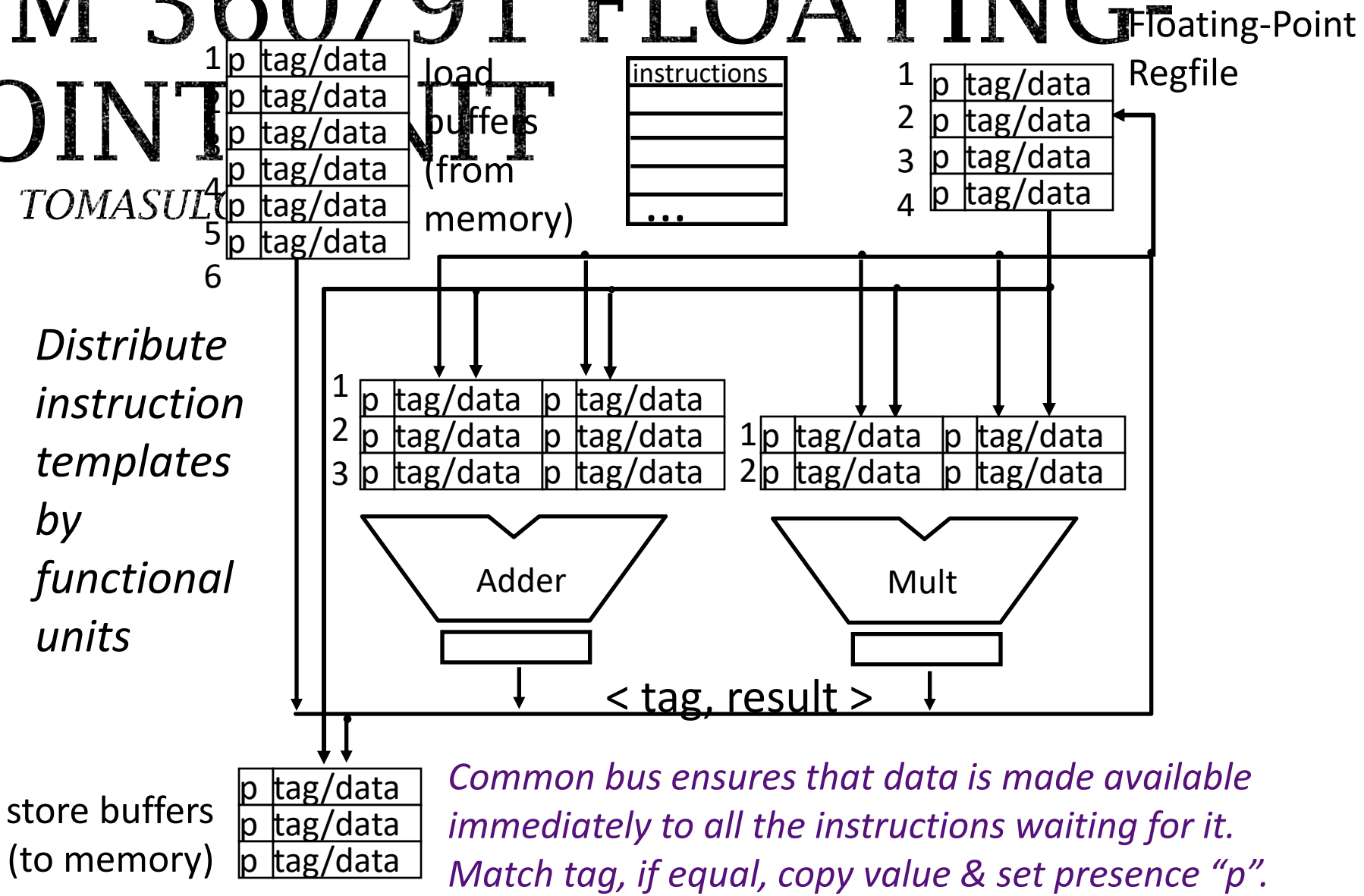
data / t_i

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

- *When are tags in sources replaced by data?*
  Whenever an FU produces data
- *When can a name be reused?*
  Whenever an instruction completes

IBM 360/91 FLOATING-POINT UNIT

R. M. TOMASULO



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".

EFFECTIVENESS?

Renaming and Out-of-order execution was first implemented in 1969 in IBM 360/91 but did not show up in the subsequent models until mid-Nineties.

Why ?

Reasons

1. Effective on a very small class of programs
2. Memory latency a much bigger problem
3. Exceptions not precise!

One more problem needed to be solved

Control transfers

ACKNOWLEDGEMENTS

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252