Out-of-Order Execution & Register Renaming

Krste Asanovic
Laboratory for Computer Science
M.I.T.

http://www.csg.lcs.mit.edu/6.823

Scoreboard for In-order Issue

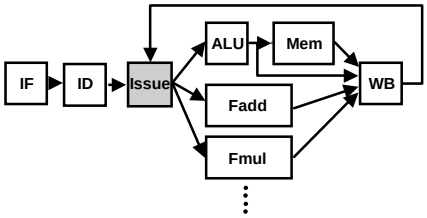
Busy[unit#] : a bit-vector to indicate unit's availability.
(unit = Int, Add, Mult, Div)
These bits are hardwired to FU's.

WP[reg#] : a bit-vector to record the registers for which writes are pending

Issue checks the instruction (opcode dest src1 src2) against the scoreboard (Busy & WP) to dispatch

FU available?	***not* Busy[FU#]**
RAW?	**WP[src1] or WP[src2]**
WAR?	***cannot arise***
WAW?	**WP[dest]**

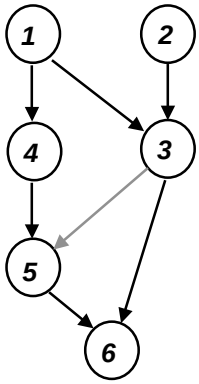
Out-of-Order Dispatch



- **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.**
- **Any instruction in buffer whose RAW hazards are satisfied can be dispatched *(for now, at most one dispatch per cycle)*. On a write back (WB), new instructions may get enabled.**

Out-of-Order Issue: *an example*

					latency
1	LD	F2,	34(R2)		*1*
2	LD	F4,	45(R3)		*long*
3	MULTD	F6,	F4,	F2	*3*
4	SUBD	F8,	F2,	F2	*1*
5	DIVD	F4,	F2,	F8	*4*
6	ADDD	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 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?

Which features of a program limit the number of instructions in the pipeline?

Overcoming the Lack of Register Names

Number of registers in an ISA limits the number of partially executed instructions in complex pipelines

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

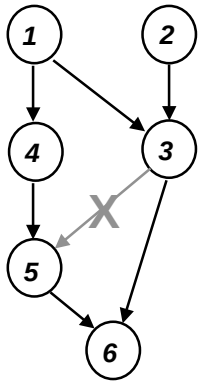
IBM 360 had only 4 Floating Point Registers

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 based on on-the-fly *register renaming*

Instruction-Level Parallelism with *Renaming*

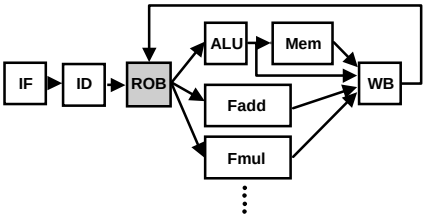
					latency
1	LD	F2,	34(R2)		*1*
2	LD	F4,	45(R3)		*long*
3	MULTD	F6,	F4,	F2	*3*
4	SUBD	F8,	F2,	F2	*1*
5	DIVD	F4',	F2,	F8	*4*
6	ADDD	F10,	F6,	F4'	*1*



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

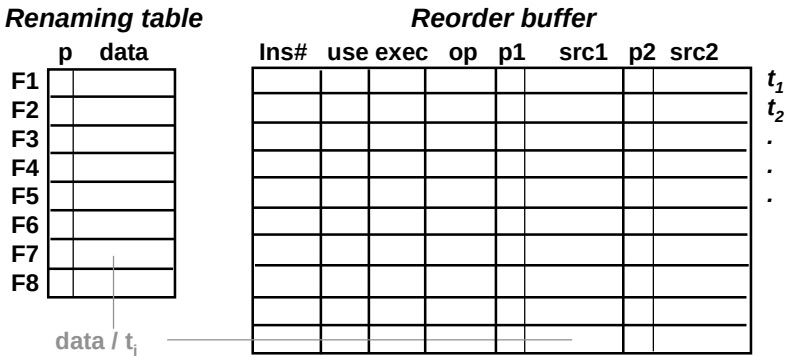


- **Decode does register renaming and adds instructions to the issue stage reorder buffer (ROB).**
  ⇒ *renaming makes WAR or WAW hazards impossible*

- **Any instruction in ROB whose RAW hazards have been satisfied can be dispatched.**
  ⇒ Out-of order or dataflow execution

Renaming & Out-of-order Issue

An example



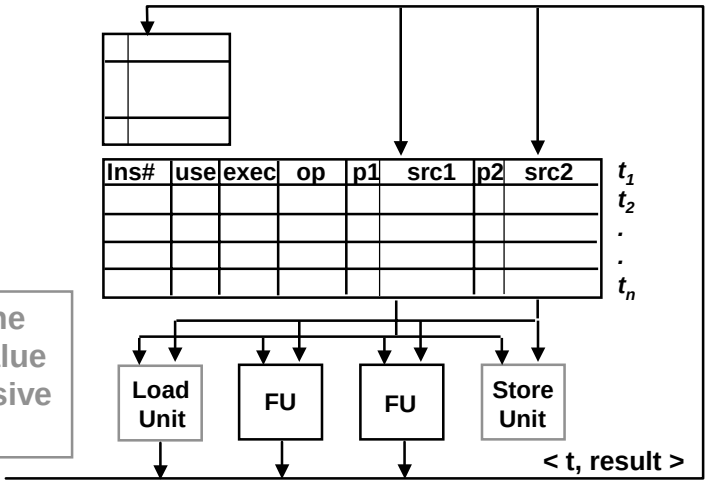
1	LD	F2,	34(R2)	
2	LD	F4,	45(R3)	
3	MULTD	F6,	F4,	F2
4	SUBD	F8,	F2,	F2
5	DIVD	F4,	F2,	F8
6	ADDD	F10,	F6,	F4

- ***When are names in sources replaced by data?***
- ***When can a name be reused?***

Data-Driven Execution

Renaming table & reg file

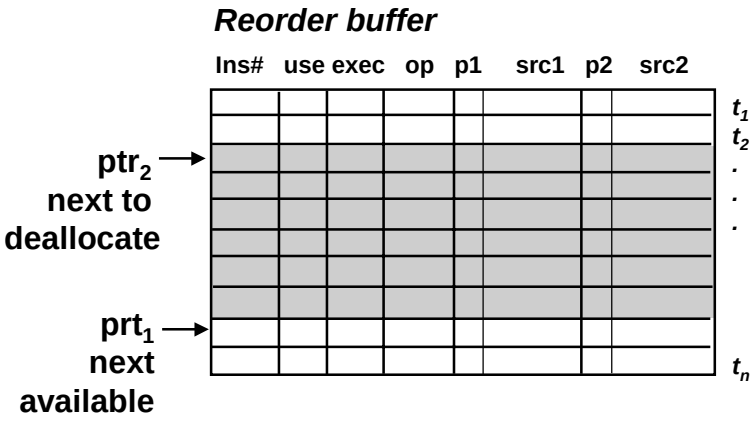
Reorder buffer



Replacing the tag by its value is an expensive operation

- **Instruction template (i.e., tag t) is allocated by the Decode stage, which also stores the tag in the reg file**
- **When an instruction completes, its tag is deallocated**

Simplifying Allocation/Deallocation

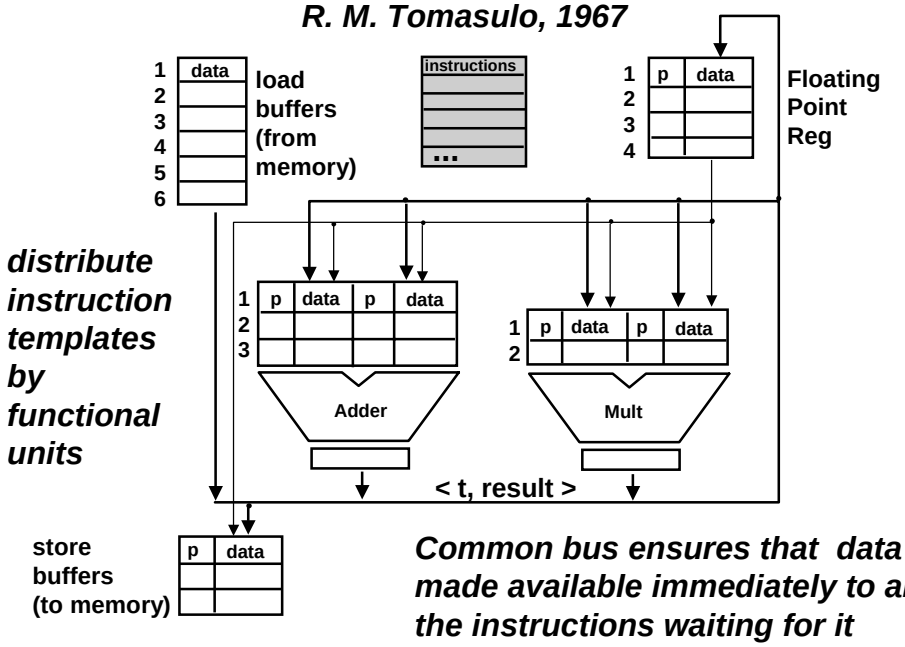


Instruction buffer is managed circularly

- When an instruction completes its "use" bit is marked free
- ptr_2 is incremented only if the "use" bit is marked free

IBM 360/91 Floating Point Unit

R. M. Tomasulo, 1967



Common bus ensures that data is made available immediately to all the instructions waiting for it

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. **Exceptions not precise!**
2. **Effective on a very small class of programs**

One more problem needed to be solved

Precise Interrupts

It must appear as if an interrupt is taken between two instructions **(say I_i and I_{i+1})**

- **the effect of all instructions up to and including I_i is totally complete**
- **no effect of any instruction after I_i has taken place**

The interrupt handler either aborts the program or restarts it at I_{i+1} .

Effect on Interrupts

Out-of-order Completion

I_1	DIVD	f6,	f6,	f4
I_2	LD	f2,	45(r3)	
I_3	MULTD	f0,	f2,	f4
I_4	DIVD	f8,	f6,	f2
I_5	SUBD	f10,	f0,	f6
I_6	ADDD	f6,	f8,	f2

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

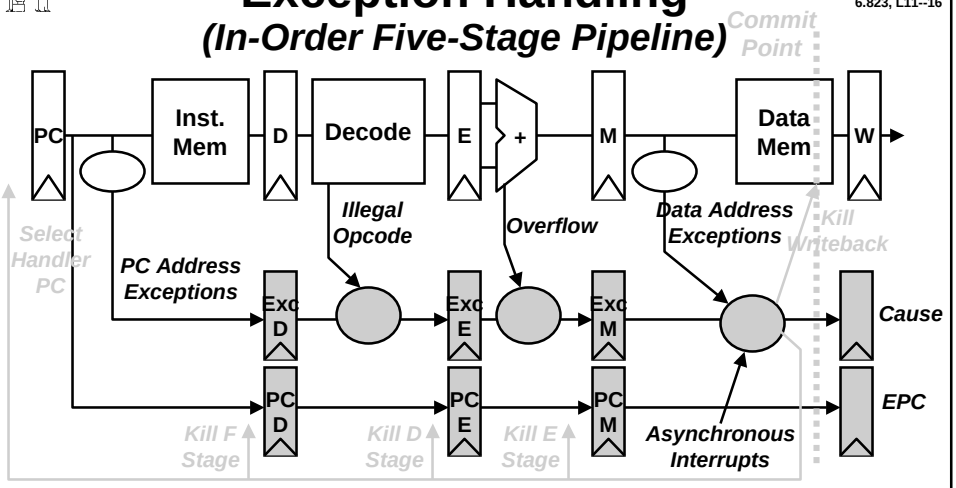
restore f2 ***restore f10***

Consider interrupts

Precise interrupts are difficult to implement at high speed - want to start execution of later instructions before exception checks finished on earlier instructions

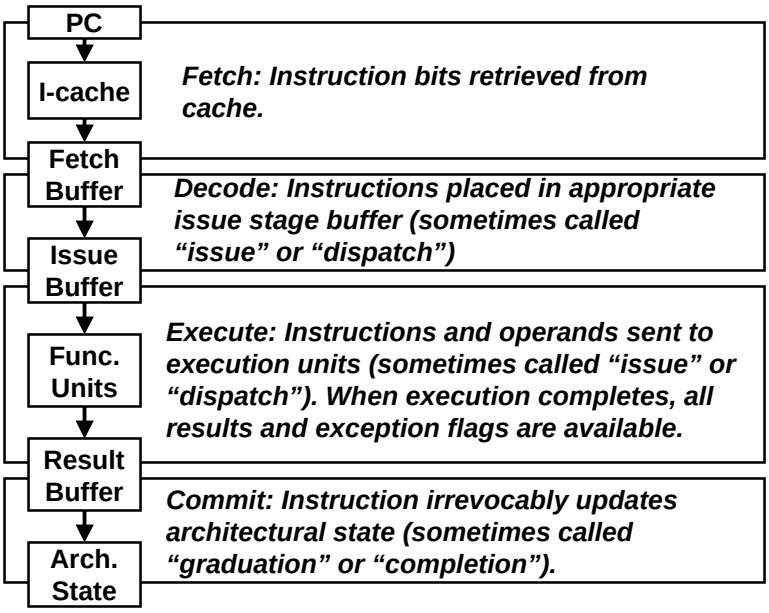
Exception Handling

(In-Order Five-Stage Pipeline)



- **Hold exception flags in pipeline until commit point (M stage)**
- **Exceptions in earlier pipe stages override later exceptions**
- **Inject external interrupts at commit point (override others)**
- **If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage**

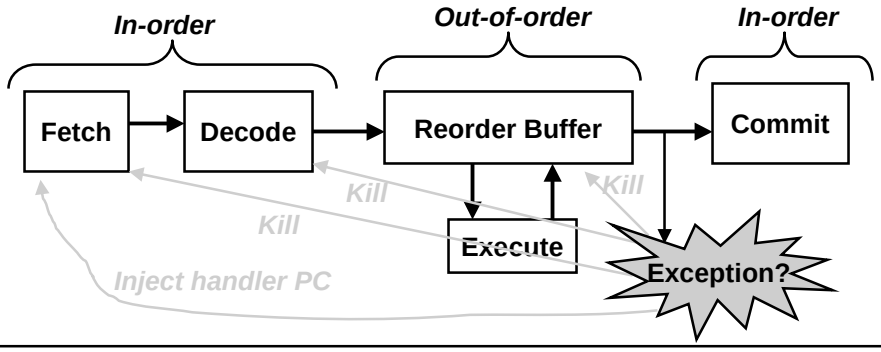
Phases of Instruction Execution



In-Order Commit for Precise Exceptions

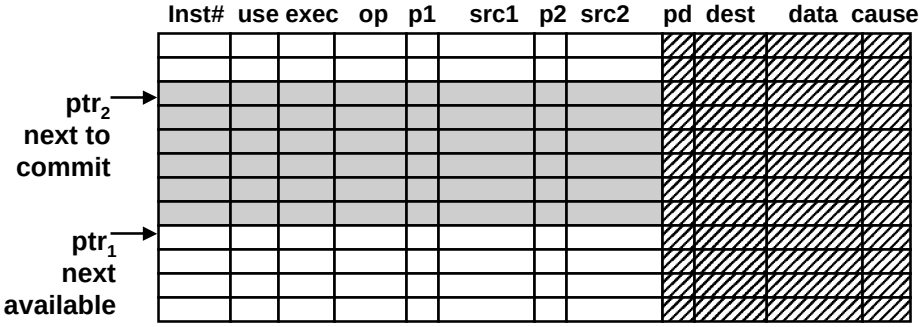
- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order ($\Rightarrow$ out-of-order completion)
- *Commit* (write-back to architectural state, regfile+memory) is in-order

Temporary storage needed to hold results before commit (shadow registers and store buffers)



Extensions for Precise Exceptions

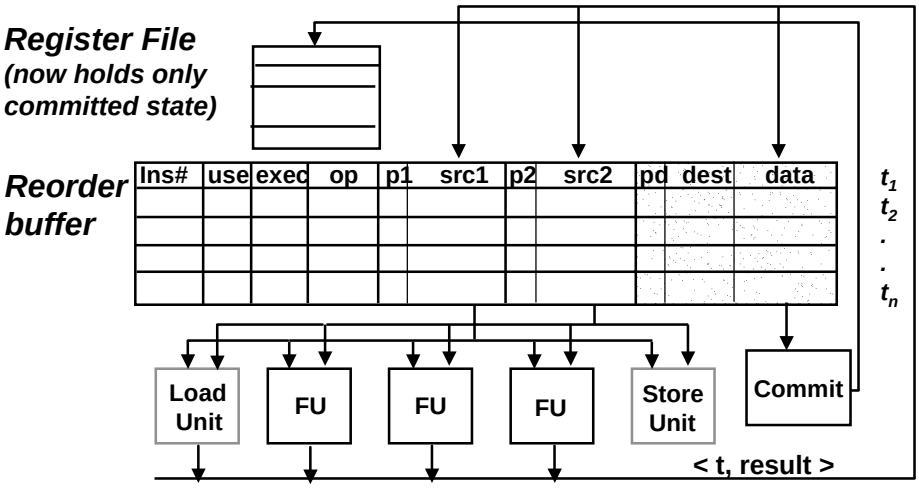
Instruction reorder buffer



- add <pd, dest, data, cause> fields in the instruction template
- commit instructions to reg file and memory in program order ⇒ buffers can be maintained circularly
- on exception, clear reorder buffer by resetting ptr_1=ptr_2

(stores must wait for commit before updating memory)

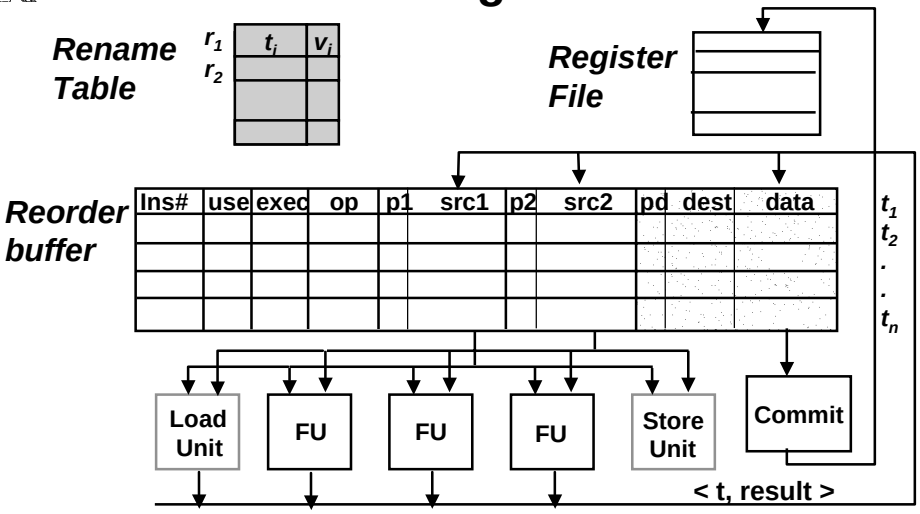
Rollback and Renaming



Register file does not contain renaming tags any more.

How does the decode stage find the tag of a source register?

Renaming Table



Renaming table is like a cache to speed up register name look up (rename tag + valid bit per entry). It needs to be cleared after each exception taken . When else are valid bits cleared? _____

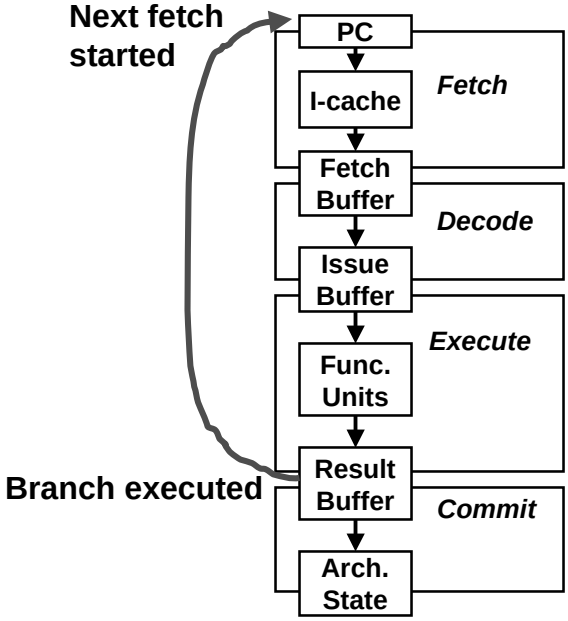
Effect of Control Transfer on Pipelined Execution

Control transfer instructions require insertion of bubbles in the pipeline.

The number of bubbles depends upon the number of cycles it takes

- ***to determine the next instruction address, and***
- ***to fetch the next instruction***

Branch Penalty



Branch Penalties in Modern Pipelines

UltraSPARC-III instruction fetch pipeline stages (in-order issue, 4-way superscalar, 750MHz, 2000)

Fetch	A	PC Generation/Mux
	P	Instruction Fetch Stage 1
	F	Instruction Fetch Stage 2
Decode	B	Branch Address Calc/Begin Decode
	I	Complete Decode
	J	Steer Instructions to Functional units
Execute	R	Register File Read
	E	Integer Execute
	⋮	Remainder of execute pipeline (+another 6 stages)

Branch penalty: Cycles?_____ Instructions?_____

Average Run-Length between Branches

Average dynamic instruction mix from SPEC92:

	SPECint92	SPECfp92
ALU	39 %	13 %
FPU Add		20 %
FPU Mult		13 %
load	26 %	23 %
store	9 %	9 %
branch	16 %	8 %
other	10 %	12 %

SPECint92: ***compress, eqntott, espresso, gcc , li***
SPECfp92: ***doduc, ear, hydro2d, mdijdp2, su2cor***

What is the average *run length* between branches?

Reducing Control Transfer Penalties

Software solution

- ***loop unrolling***
  Increases the run length
- ***instruction scheduling***
  Compute the branch condition as early as possible
  (limited)

Hardware solution

- ***delay slots***
  replaces pipeline bubbles with useful work
  (requires software cooperation)
- ***branch prediction & speculative execution***
  of instructions beyond the branch

Branch Prediction

Motivation: **branch penalties limit performance of deeply pipelined processors**

Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

Required hardware support:

Prediction structures: **branch history tables, branch target buffers, etc.**

Mispredict recovery mechanisms:

- **In-order machines: kill instructions following branch in pipeline**
- **Out-of-order machines: shadow registers and memory buffers for each speculated branch**

DLX Branches and Jumps

Instruction	*Taken known?*	*Target known?*
BEQZ/BNEZ	**After Reg. Fetch**	**After Inst. Fetch**
J	**Always Taken**	**After Inst. Fetch**
JR	**Always Taken**	**After Reg. Fetch**

Must know (or guess) both target address and whether taken to execute branch/jump.