

forwarding idea

read wrong value (e.g. from register)

correct value is **already computed**

elsewhere in pipeline

maybe even after old value was read

substitute from wrong value

using MUX

quiz question: forwarding in IRMOVQ

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
<code>irmovq \$50, %r8</code>		F	D	E	M	W				
<code>addq %r11, %r8</code>			F	D	E	M	W			

quiz question: forwarding in IRMOVQ

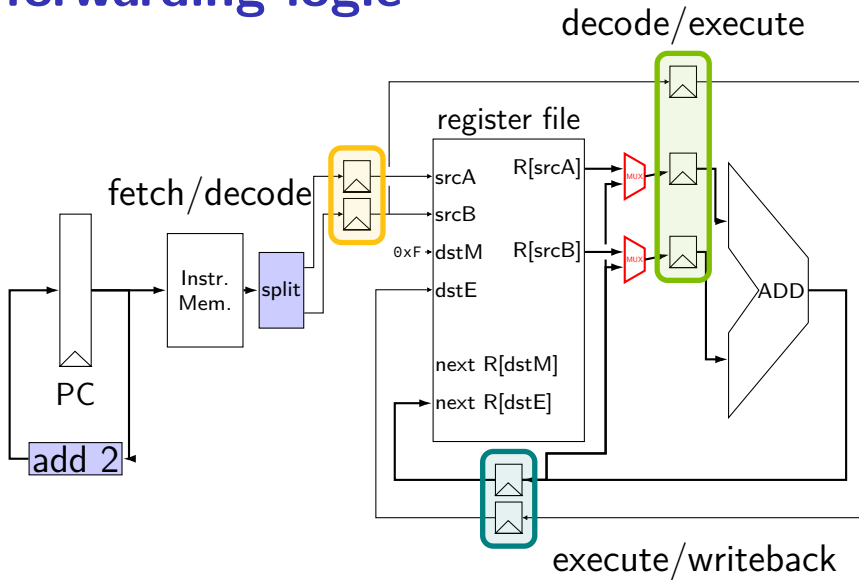
	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
<code>irmovq \$50, %r8</code>		F	D	E	M	W				
<code>addq %r11, %r8</code>			F	D	E	M	W			

output of **decode/execute** regs (`irmovq`)
(unchanged during execute stage)

input of **execute/memory** regs (`irmovq`)

input of **decode/execute** regs (`addq`)

forwarding logic



some forwarding paths

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
<code>addq %r8, %r9</code>		F	D	E	M	W				
<code>subq %r9, %r11</code>			F	D	E	M	W			
<code>mrmovq 4(%r11), %r10</code>				F	D	E	M	W		
<code>rmmovq %r9, 8(%r11)</code>					F	D	E	M	W	
<code>xorq %r10, %r9</code>						F	D	E	M	W

forwarding in HCL

```
register dE {
    valA : 64 = 0;
    dstE : 4 = 0;
};
...
/* was: d_valA = reg_outputA; */
d_valA = [
    reg_srcA == e_dstE : e_valE;
    ...
    1 : reg_outputA;
];
d_dstE = ...;
```

forwarding in HCL

```
register dE {  
    valA : 64 = 0;  
    dstE : 4 = 0;  
};  
...  
/* was: d_valA = reg_outputA; */  
d_valA = [  
    reg_srcA == e_dstE : e_valE;  
    ...  
    1 : reg_outputA;  
];  
d_dstE = ...;
```


forwarding in HCL

```
register dE {  
    valA : 64 = 0;  
    dstE : 4 = 0;  
};  
...  
/* was: d_valA = reg_outputA; */  
d_valA = [  
    reg_srcA == e_dstE : e_valE;  
    ...  
    1 : reg_outputA;  
];  
d_dstE = ...;
```

unsolved problem

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
mrmovq 0(%rax), %rbx		F	D	E	M	W				
subq %rbx, %rcx			F	D	E	M	W			

unsolved problem

	cycle #									
	0	1	2	3	4	5	6	7	8	
mrmovq 0(%rax), %rbx	F	D	E	M	W					
subq %rbx, %rcx		F	D	E	M	W				
subq %rbx, %rcx		F	F	D	E	M	W			

stall

multiple forwarding paths

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
addq %r10, %r8		F	D	E	M	W				
addq %r11, %r8			F	D	E	M	W			
addq %r12, %r8				F	D	E	M	W		

multiple forwarding paths


	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
<code>addq %r10, %r8</code>		F	D	E	M	W				
<code>addq %r11, %r8</code>			F	D	E	M	W			
<code>addq %r12, %r8</code>				F	D	E	M	W		

multiple forwarding HCL

```
d_valA = [  
    ...  
    reg_srcA == e_dstE : e_valE;  
    reg_srcA == m_dstE : m_valE;  
    ...  
    1 : reg_outputA;  
];
```


multiple forwarding paths (2)

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
addq %r10, %r8		F	D	E	M	W				
addq %r11, %r12			F	D	E	M	W			
addq %r12, %r8				F	D	E	M	W		




multiple forwarding paths (2)

	cycle #	0	1	2	3	4	5	6	7	8
<code>addq %r10, %r8</code>		F	D	E	M	W				
<code>addq %r11, %r12</code>			F	D	E	M	W			
<code>addq %r12, %r8</code>				F	D	E	M	W		



multiple forwarding paths (2)

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
<code>addq %r10, %r8</code>		F	D	E	M	W				
<code>addq %r11, %r12</code>			F	D	E	M	W			
<code>addq %r12, %r8</code>				F	D	E	M	W		



after forwarding/prediction

where do we still need to stall?

memory output needed in fetch
ret followed by anything

memory output needed in execute
mrmovq or popq + use
(in immediately following instruction)

overall CPU

5 stage pipeline

1 instruction completes every cycle — except hazards

most data hazards: solved by forwarding

load/use hazard: 1 cycle of stalling

jXX control hazard: branch prediction + squashing
2 cycle penalty for misprediction

ret control hazard: 3 cycles of stalling

pipelined control costs

how much faster than single-cycle processor?

at most five times faster

depends on hardware details

- does added logic make clock cycle slower?

depends on what programs we run:

- how many mispredicted jumps?

- how many rets?

- how many load/use hazards?

hazards versus dependencies

dependency — X needs result of instruction Y?

hazard — will it not work in some pipeline?

before extra work is done to “resolve” hazards
like forwarding or stalling or branch prediction

ex.: dependencies and hazards (1)

addq %rax, %rbx

subq %rax, %rcx

irmovq \$100, %rcx

addq %rcx, %r10

addq %rbx, %r10

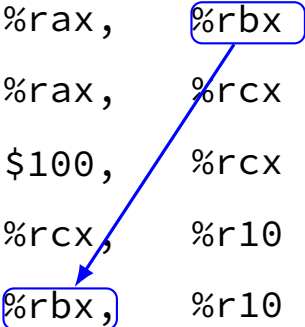
where are dependencies?

which are hazards in our pipeline?

which are resolved with forwarding?

ex.: dependencies and hazards (1)

addq	%rax,	%rbx
subq	%rax,	%rcx
irmovq	\$100,	%rcx
addq	%rcx,	%r10
addq	%rbx,	%r10



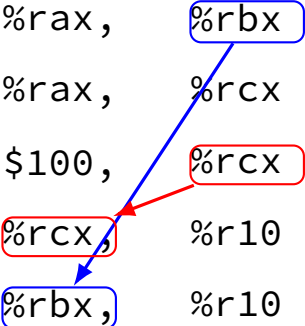
where are dependencies?

which are hazards in our pipeline?

which are resolved with forwarding?

ex.: dependencies and hazards (1)

addq	%rax,	%rbx
subq	%rax,	%rcx
irmovq	\$100,	%rcx
addq	%rcx,	%r10
addq	%rbx,	%r10



where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

ex.: dependencies and hazards (1)

addq	%rax,	%rbx
subq	%rax,	%rcx
irmovq	\$100,	%rcx
addq	%rcx,	%r10
addq	%rbx,	%r10

where are dependencies?
which are hazards in our pipeline?
which are resolved with forwarding?

ex.: dependencies and hazards (2)

mrmovq 0(%rax) %rbx

addq %rbx %rcx

jne foo

foo: **addq** %rcx %rdx

mrmovq (%rdx) %rcx

where are dependencies?

which are hazards in our pipeline?

which are resolved with forwarding?

pipeline with different hazards

example: 4-stage pipeline:

fetch/decode/execute+memory/writeback

	<i>// 4 stage</i>	<i>// 5 stage</i>
addq %rax, %r8	<i>//</i>	<i>// W</i>
subq %rax, %r9	<i>// W</i>	<i>// M</i>
xorq %rax, %r10	<i>// EM</i>	<i>// E</i>
andq %r8, %r11	<i>// D</i>	<i>// D</i>

pipeline with different hazards

example: 4-stage pipeline:

fetch/decode/execute+memory/writeback

	<i>// 4 stage</i>	<i>// 5 stage</i>
addq %rax, %r8	<i>//</i>	<i>// W</i>
subq %rax, %r9	<i>// W</i>	<i>// M</i>
xorq %rax, %r10	<i>// EM</i>	<i>// E</i>
andq %r8, %r11	<i>// D</i>	<i>// D</i>

addq/andq is hazard with 5-stage pipeline

addq/andq is **not** a hazard with 4-stage pipeline

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
addq %rcx, %r9		F	D	E1	E2	M	W			
addq %r9, %rbx										
addq %rax, %r9										

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

<i>cycle #</i>	0	1	2	3	4	5	6	7	8
addq %rcx, %r9	F	D	E1	E2	M	W			
addq %r9, %rbx		F	D	E1	E2	M	W		
addq %rax, %r9			F	D	E1	E2	M	W	

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

<i>cycle #</i>	0	1	2	3	4	5	6	7	8
addq %rcx, %r9	F	D	E1	E2	M	W			
addq %r9, %rbx		F	D	E1	E2	M	W		
addq %r9, %rbx		F	D	D	E1	E2	M	W	
addq %rax, %r9			F	D	E1	E2	M	W	
addq %rax, %r9			F	F	D	E1	E2	M	W

exercise: different pipeline

split execute into two stages: F/D/E1/E2/M/W

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
addq %rcx, %r9		F	D	E1	E2	M	W			
addq %r9, %rbx			F	D	E1	E2	M	W		
addq %r9, %rbx			F	D	D	E1	E2	M	W	
addq %rax, %r9				F	D	E1	E2	M	W	
addq %rax, %r9				F	F	D	E1	E2	M	W

exercise: forwarding paths

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
<code>addq %rcx, %r9</code>		F	D	E	M	W				
<code>rmmovq %r9, 8(%r8)</code>			F	D	E	M	W			
<code>popq %r10</code>				F	D	E	M	W		
<code>mrmovq 8(%r9), %r11</code>					F	D	E	M	W	
<code>pushq %r11</code>						F	D	E	M	W

exercise: forwarding paths

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
<code>addq %rcx, %r9</code>		F	D	E	M	W				
<code>rmmovq %r9, 8(%r8)</code>			F	D	E	M	W			
<code>popq %r10</code>				F	D	E	M	W		
<code>mrmovq 8(%r9), %r11</code>					F	D	E	M	W	
<code>pushq %r11</code>						F	D	E	M	W

exercise: forwarding paths

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
<code>addq %rcx, %r9</code>		F	D	E	M	W				
<code>rmmovq %r9, 8(%r8)</code>			F	D	E	M	W			
<code>popq %r10</code>				F	D	E	M	W		
<code>mrmovq 8(%r9), %r11</code>					F	D	E	M	W	
<code>pushq %r11</code>						F	D	E	M	W

exercise: forwarding paths (alt pipe)

suppose four-stage pipeline:

fetch/decode+execute/memory/writeback

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
addq %rcx, %r9		F	DE	M	W					
rmmovq %r9, 8(%r8)			F	DE	M	W				
popq %r10				F	DE	M	W			
mrmovq 8(%r9), %r11					F	DE	M	W		
pushq %r11						F	DE	M	W	

exercise: forwarding paths (alt pipe)

suppose four-stage pipeline:

fetch/decode+execute/memory/writeback

	<i>cycle #</i>	0	1	2	3	4	5	6	7	8
addq %rcx, %r9		F	DE	M	W					
rmmovq %r9, 8(%r8)			F	DE	M	W				
popq %r10				F	DE	M	W			
mrmovq 8(%r9), %r11					F	DE	M	W		
pushq %r11						F	DE	M	W	

overall CPU

5 stage pipeline

1 instruction completes every cycle — except hazards

most data hazards: solved by forwarding

load/use hazard: 1 cycle of stalling

jXX control hazard: branch prediction + squashing
2 cycle penalty for misprediction

ret control hazard: 3 cycles of stalling

pipelined control costs

how much faster than single-cycle processor?

at most five times faster

depends on HW details:

- how expensive is forwarding logic? (new MUXes on critical path)

- how well balanced are the stages?

depends on what programs we run:

- how many mispredicted jumps?

- how many rets?

- how many load/use hazards?

HCL2D pipeline registers

```
register xF {
    pc : 64 = 0;
};
/* Fetch+PC Update*/
register fD {
    rA : 4 = REG_NONE; rB : 4 = REG_NONE;
};
/* Decode */
register dE {
    valA : 64 = 0; valB : 64 = E; dstE : 4 = REG_NONE;
}
/* Execute */
register eW {
    valE : 64 = 0; dstE : 4 = REG_NONE;
}
/* Writeback */
```


HCL2D: Fetch/Decode

unpipelined

```
/* Fetch+PC Update*/  
pc = F_pc;  
x_pc = pc + 2;  
rA = i10bytes[12..16];  
rB = i10bytes[8..12];  
/* Decode */  
reg_srcA = rA;  
reg_srcB = rB;  
dstE = rB;  
valA = reg_outputA;  
valB = reg_outputB;
```

pipelined

```
/* Fetch+PC Update*/  
pc = F_pc;  
x_pc = pc + 2;  
f_rA = i10bytes[12..16];  
f_rB = i10bytes[8..12];  
/* Decode */  
reg_srcA = D_rA;  
reg_srcB = D_rB;  
dstE = D_rB;  
d_valA = reg_outputA;  
d_valB = reg_outputB;
```

HCL2D pipelining debugging: intro

debugging pipelines is consistently one of the biggest sources of difficulty in this class

notably: big drain on TA time

HCL2D pipeline debugging (1)

draw a picture of the state of the instructions

get -d output

 redirect to a file

 cpu.exe -d input.yo >output.txt

check **each stage** of the broken instruction

expect forwarding/hazard-handling problems

HCL2D pipeline debugging (2)






write assembly — not just supplied test cases
remove anything not involved in the error
find a **minimal** test case
don't spend time looking at irrelevant instructions

draw the pipeline stages
what instructions are in fetch/decode/etc. when






HCL2D addq unpipelined

```
wire rA : 4, rB : 4, dstE : 4;
wire valA : 64, valB : 64, valE : 64;
register xF {
    pc : 64 = 0;
};
/* Fetch+PC Update*/
pc = F_pc; x_pc = pc + 2;
rA = i10bytes[12..16]; rB = i10bytes[8..12];
/* Decode */
reg_srcA = rA; reg_srcB = rB; dstE = rB;
valA = reg_outputA; valB = reg_outputB;
/* Execute */
valE = valA + valB;
/* Writeback */
reg_dstE = dstE; reg_inputE = valE;
```






addq pipeline registers

stage	addq rA, rB	
fetch	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$	
PC update	$\text{PC} \leftarrow \text{valP}$	 PC
decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	 icode
execute	$\text{valE} \leftarrow \text{valB} + \text{valB}$	 icode
memory		 icode
write back	$R[\text{rB}] \leftarrow \text{valE}$	 icode






addq pipeline registers

stage	addq rA, rB	
fetch	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$	
PC update	$\text{PC} \leftarrow \text{valP}$	 PC
decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	 icode, rA, rB
execute	$\text{valE} \leftarrow \text{valB} + \text{valB}$	 icode, rB
memory		 icode, rB
write back	$R[\text{rB}] \leftarrow \text{valE}$	 icode, rB






addq pipeline registers

stage	addq rA, rB	
fetch	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$	
PC update	$\text{PC} \leftarrow \text{valP}$	 PC
decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	 icode, rA, rB
execute	$\text{valE} \leftarrow \text{valB} + \text{valB}$	 icode, rB, valA, valB
memory		 icode, rB
write back	$R[\text{rB}] \leftarrow \text{valE}$	 icode, rB

addq pipeline registers

stage	addq rA, rB	
fetch	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$	
PC update	$\text{PC} \leftarrow \text{valP}$	 PC
decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	 icode, rA, rB
execute	$\text{valE} \leftarrow \text{valB} + \text{valB}$	 icode, rB, valA, valB
memory		 icode, rB, valE
write back	$R[\text{rB}] \leftarrow \text{valE}$	 icode, rB, valE

addq pipeline registers

stage	addq rA, rB	
fetch	$\text{icode} : \text{ifun} \leftarrow M_1[\text{PC}]$ $\text{rA} : \text{rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$	
PC update	$\text{PC} \leftarrow \text{valP}$	 PC
decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$ $\text{dstE} \leftarrow \text{rB}$	 icode, rA, rB
execute	$\text{valE} \leftarrow \text{valB} + \text{valB}$	 icode, dstE, valA, valB
memory		 icode, dstE, valE
write back	$R[\text{dstE}] \leftarrow \text{valE}$	 icode, dstE, valE


addq pipeline registers

stage addq rA, rB

fetch icode : ifun $\leftarrow M_1[PC]$
 rA : rB $\leftarrow M_1[PC+1]$
 valP $\leftarrow PC + 2$

 PC

PC update PC \leftarrow valP

 icode, rA, rB

decode valA $\leftarrow R[rA]$
 valB $\leftarrow R[rB]$
 dstE \leftarrow rB

 icode, **dstE**, valA, valB

execute valE \leftarrow valA + valB

redundant with rB + icode e, **dstE**, valE ---
 but will make handling data hazards easier e, **dstE**, valE ---

write back **R[dstE]** \leftarrow valE

Precise Exceptions and Out-of-Order Execution

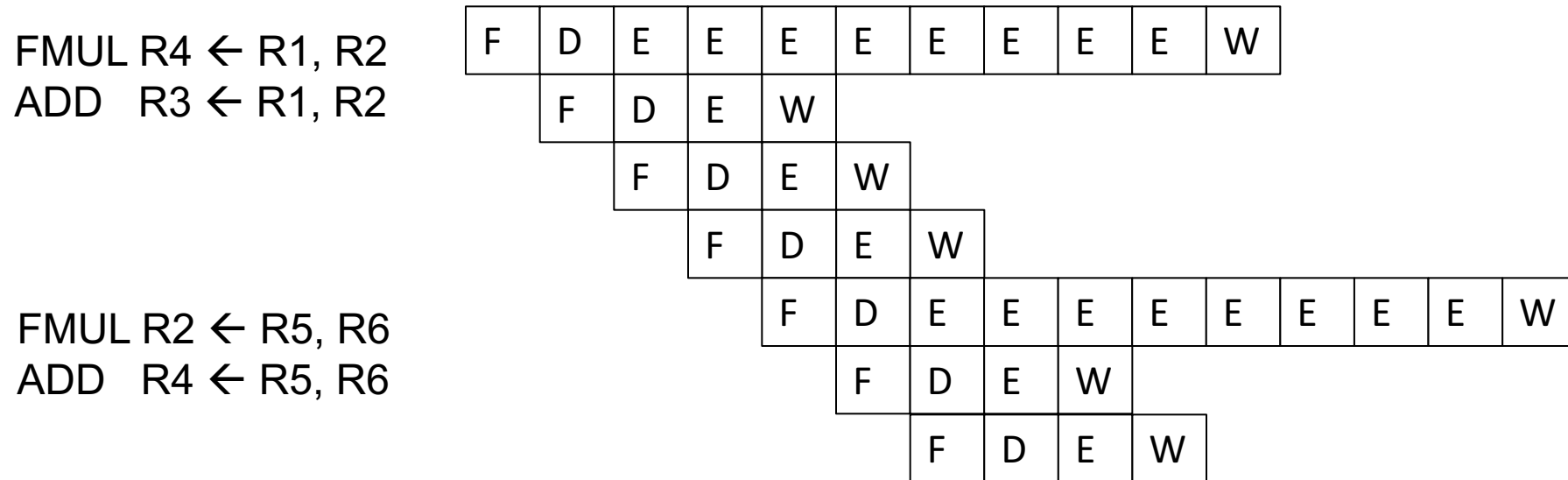
Samira Khan

Multi-Cycle Execution

- Not all instructions take the same amount of time for “execution”
- Idea: Have multiple different functional units that take different number of cycles
 - Can be pipelined or not pipelined
 - Can let independent instructions start execution on a different functional unit before a previous long-latency instruction finishes execution

ISSUES IN PIPELINING: MULTI-CYCLE EXECUTE

- Instructions can take different number of cycles in EXECUTE stage
 - Integer ADD versus FP Multiply



- What is wrong with this picture?
 - What if FMUL incurs an exception?
 - Sequential semantics of the ISA NOT preserved!

The Von Neumann Model/Architecture

- Also called *stored program computer* (instructions in memory). Two key properties:
- Stored program
 - Instructions stored in a linear memory array
 - Memory is unified between instructions and data
 - The interpretation of a stored value depends on the control signals
- Sequential instruction processing
 - One instruction processed (fetched, executed, and completed) at a time
 - Program counter (instruction pointer) identifies the current instr.
 - Program counter is advanced sequentially except for control transfer instructions

HANDLING EXCEPTIONS IN PIPELINING

- Exceptions versus interrupts
- Cause
 - Exceptions: internal to the running thread
 - Interrupts: external to the running thread
- When to Handle
 - Exceptions: when detected (and known to be non-speculative)
 - Interrupts: when convenient
 - Except for very high priority ones
 - Power failure
 - Machine check
- Priority: process (exception), depends (interrupt)
- Handling Context: process (exception), system (interrupt)

PRECISE EXCEPTIONS/INTERRUPTS

- The architectural state should be consistent when the exception/interrupt is ready to be handled
 1. All previous instructions should be completely retired.
 2. No later instruction should be retired.

Retire = commit = finish execution and update arch. state

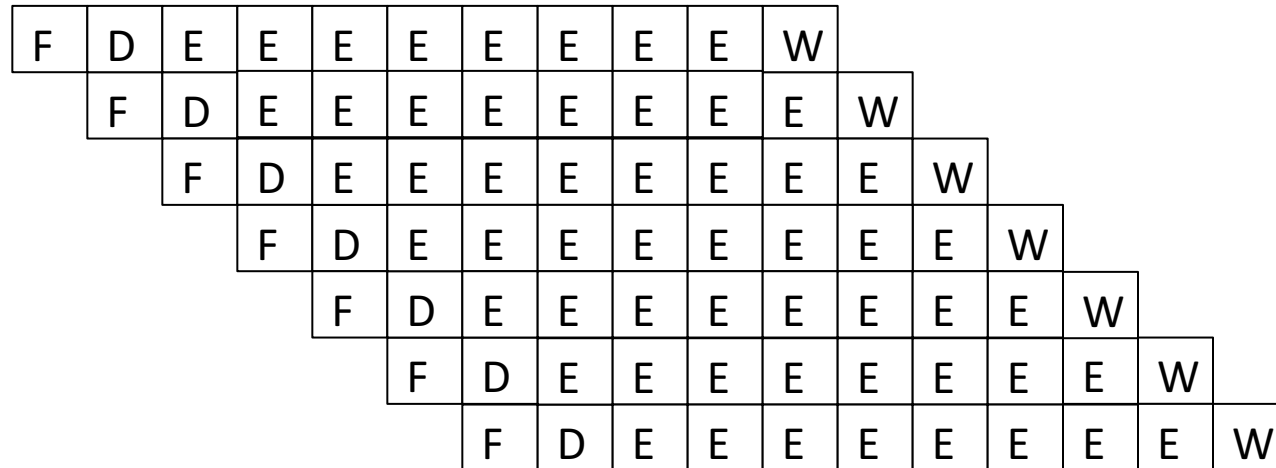
WHY DO WE WANT PRECISE EXCEPTIONS?

- Aid software debugging
- Enable (easy) recovery from exceptions, e.g. page faults
- Enable (easily) restartable processes

ENSURING PRECISE EXCEPTIONS IN PIPELINING

- Idea: Make each operation take the same amount of time

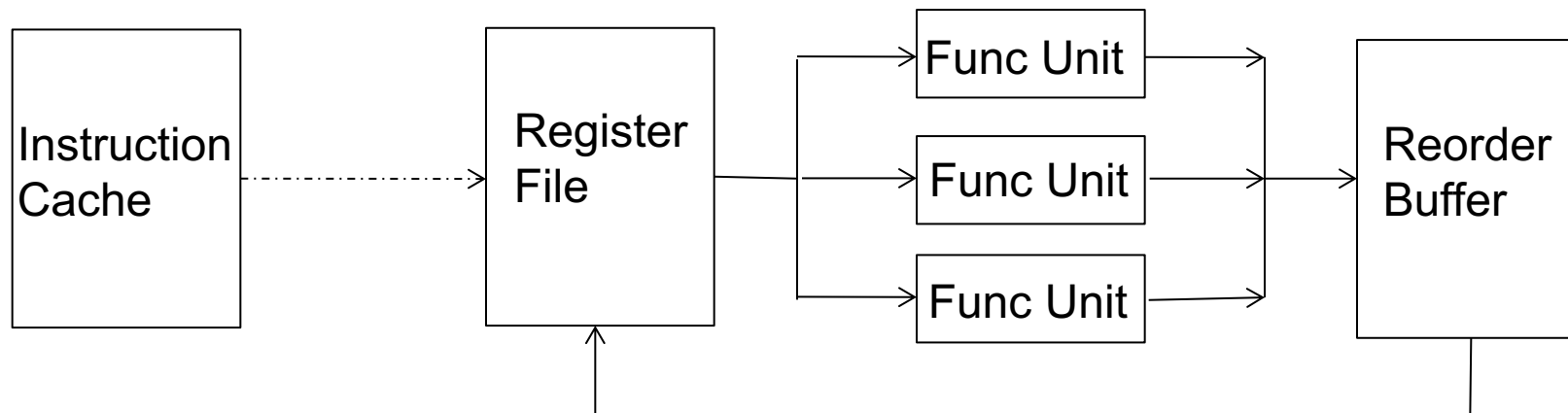
FMUL R3 ← R1, R2
ADD R4 ← R1, R2



- Downside
 - What about memory operations?
 - Each functional unit takes 500 cycles?

SOLUTION: REORDER BUFFER (ROB)

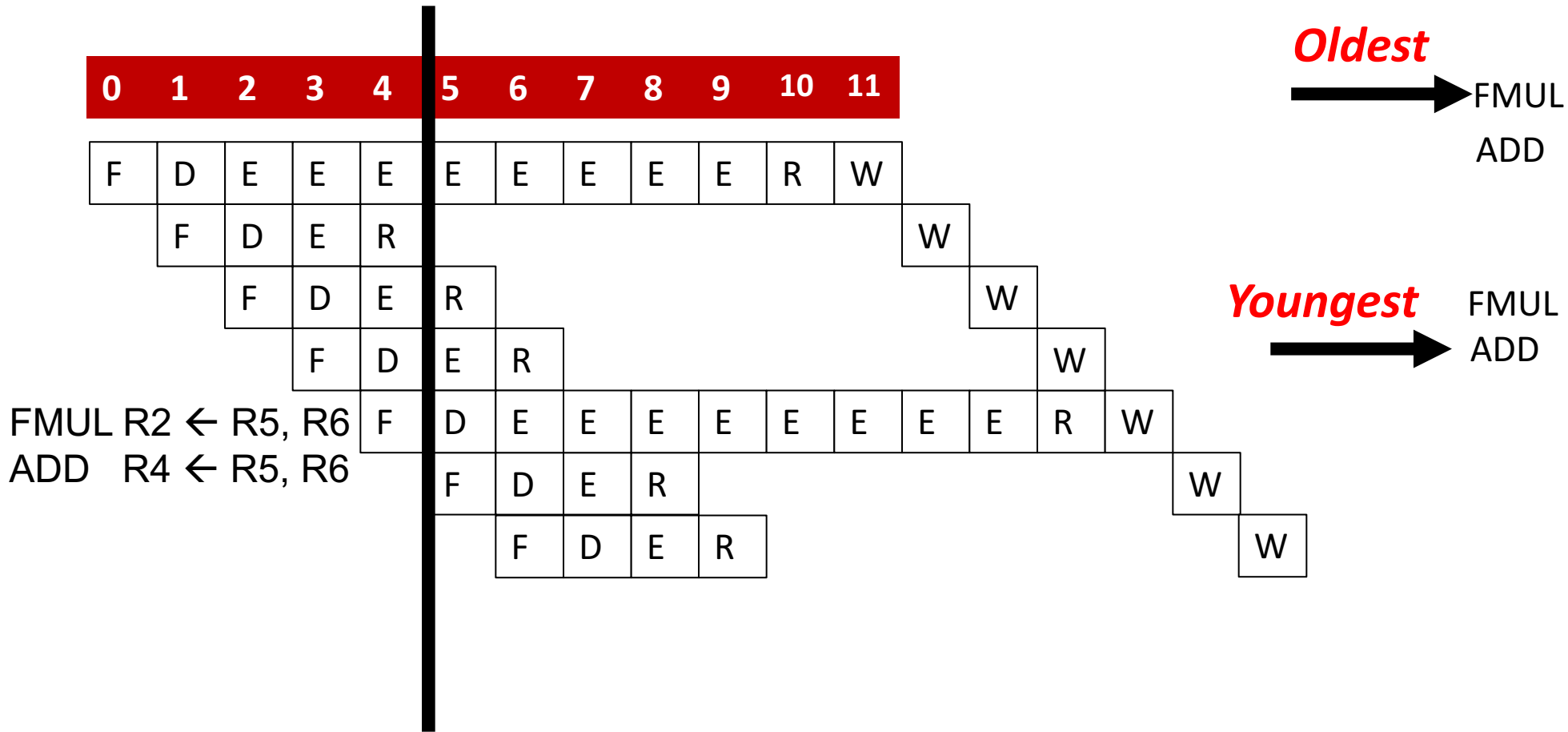
- Idea: Complete instructions out-of-order, but reorder them before making results visible to architectural state
- When instruction is decoded it reserves an entry in the ROB
- When instruction completes, it writes result into ROB entry
- When instruction oldest in ROB and it has completed, its result moved to reg. file or memory



		V	DEST REG	DEST VAL	COMPLETE
<i>Oldest</i> →	FMUL	1	R4	--	0
	ADD	1	R3	--	0
		1			0
	FMUL	1			0
<i>Youngest</i> →	ADD	1			0

Reorder File

REORDER BUFFER: INDEPENDENT OPERATIONS

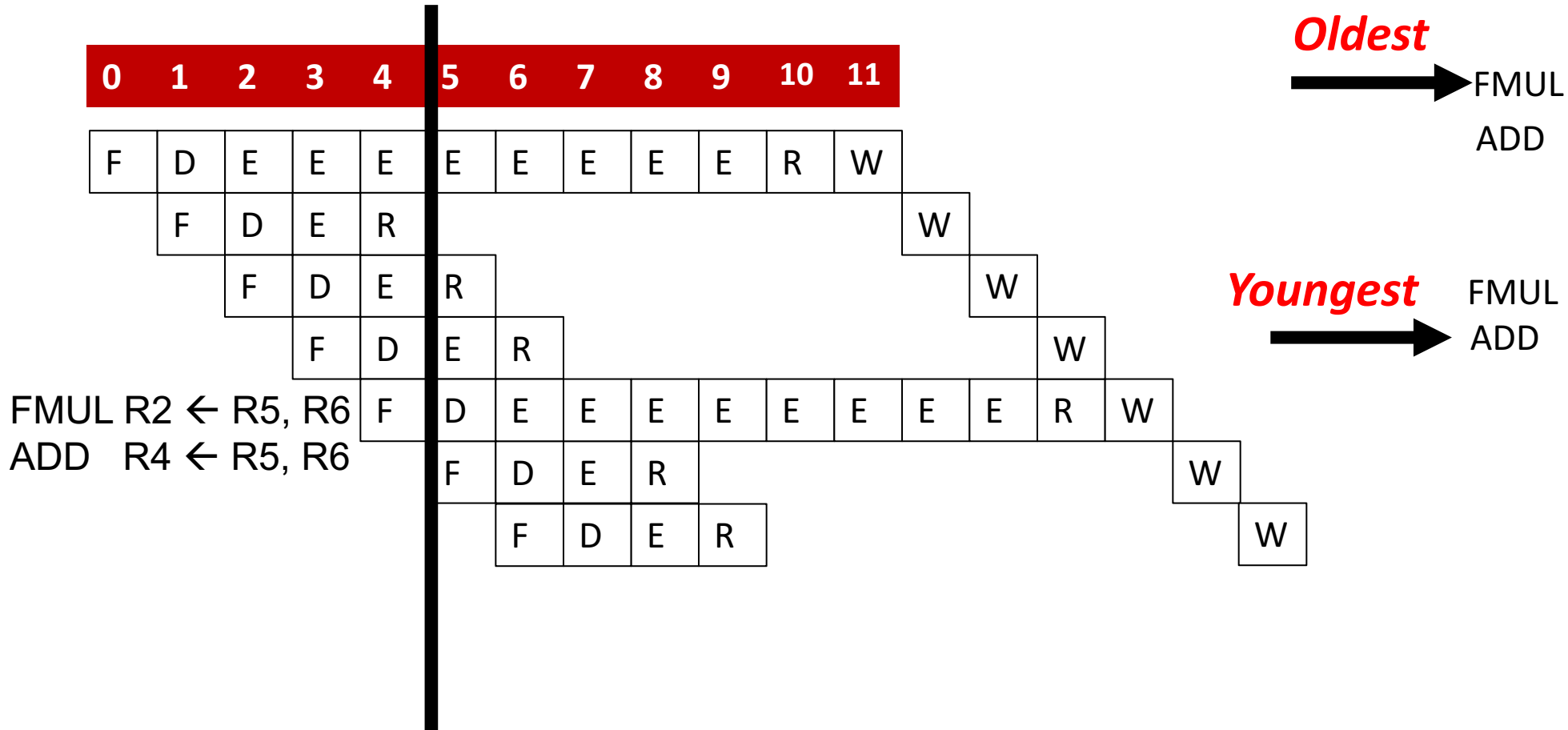


CYCLE 5

V	DEST REG	DEST VAL	CO MPL ETE
1	R4	--	0
1	R3	1000	1
1			0
1			0
1	R2	--	0

Reorder File

REORDER BUFFER: INDEPENDENT OPERATIONS

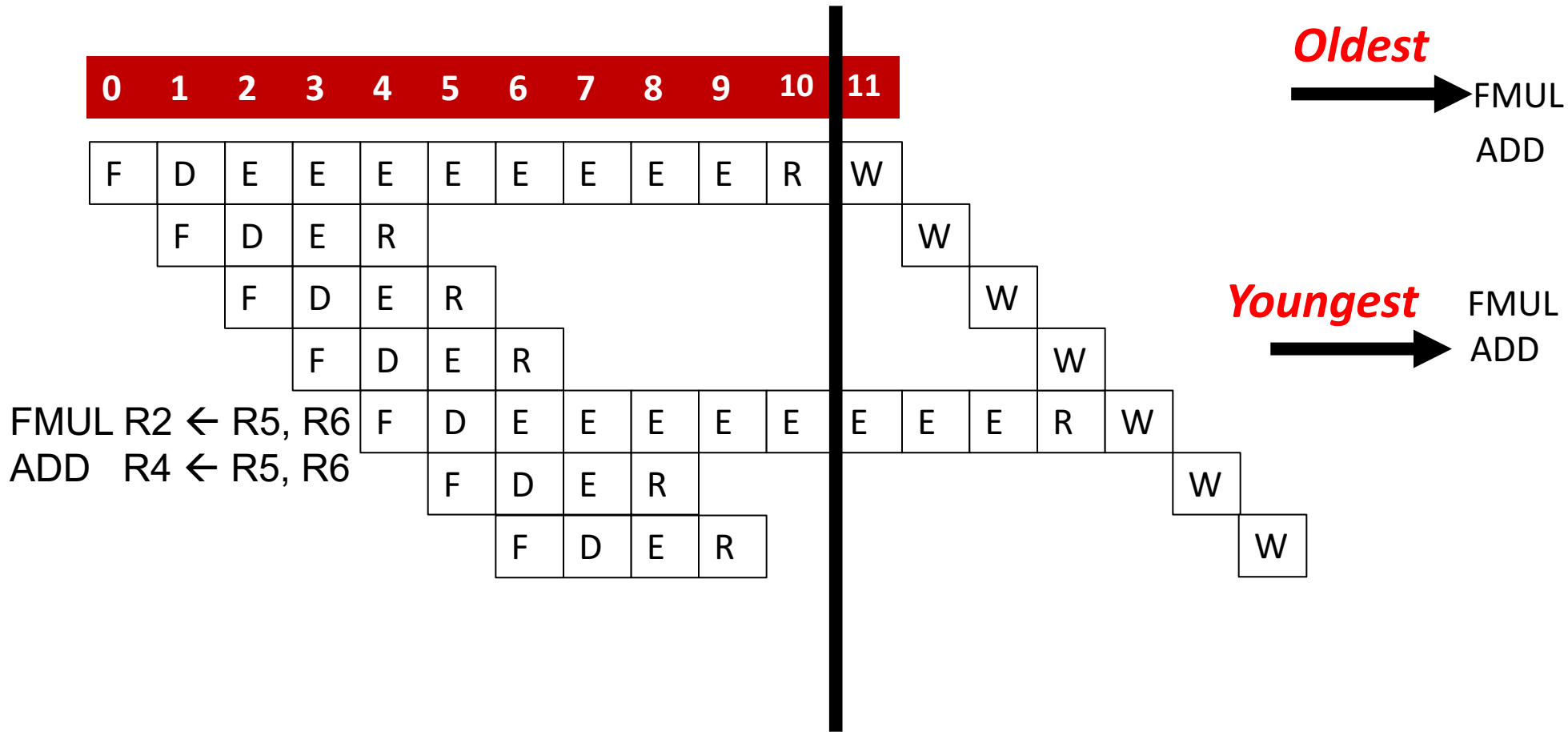


CYCLE 5

V	DEST REG	DEST VAL	CO MPL ETE
1	R4	--	0
1	R3	1000	1
1			0
1			0
1	R2	--	0
1	R4	--	0

Reorder File

REORDER BUFFER: INDEPENDENT OPERATIONS

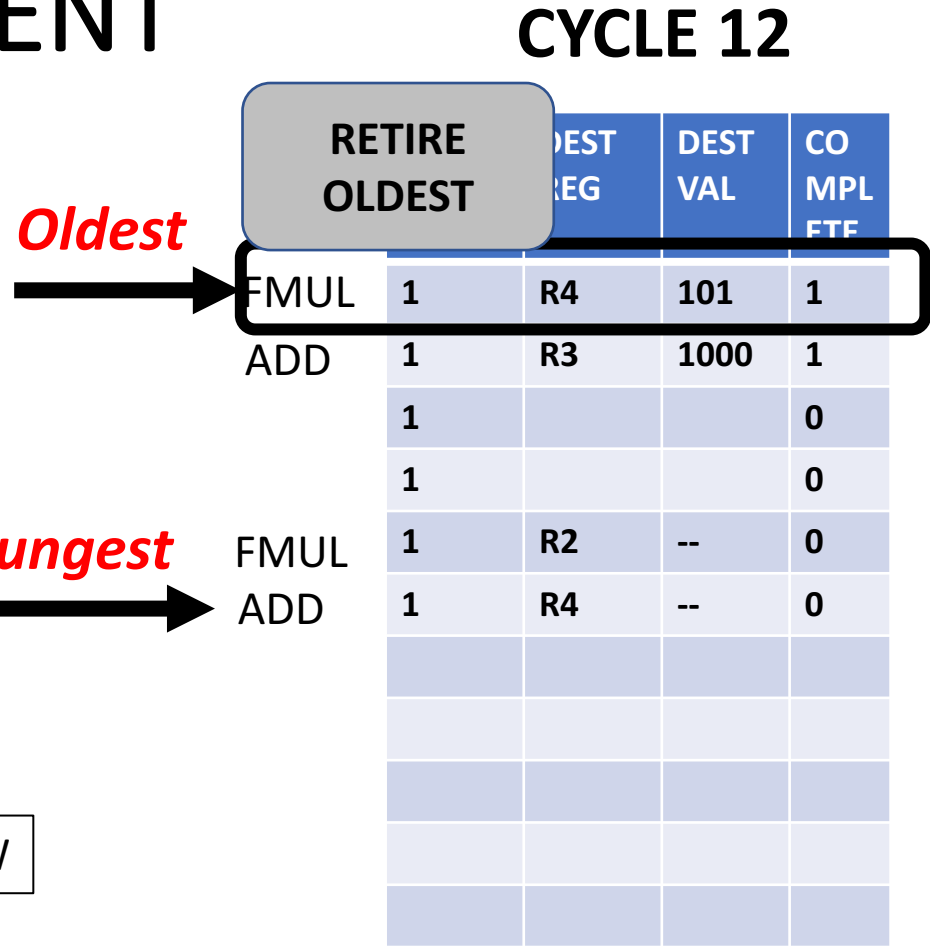
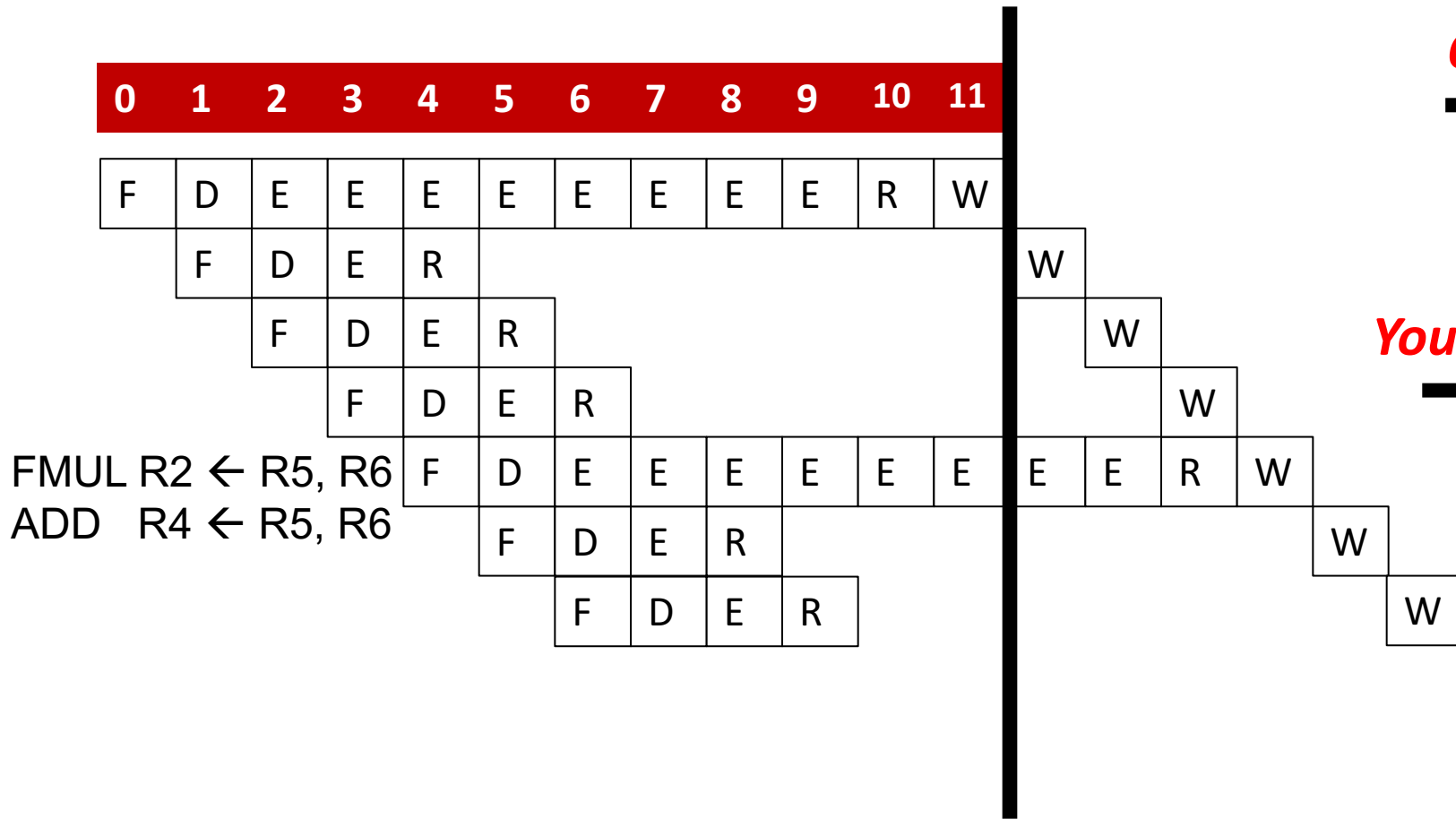


CYCLE 11

V	DEST REG	DEST VAL	CO MPL ETE
1	R4	101	0
1	R3	1000	1
1			0
1			0
1	R2	--	0
1	R4	--	0

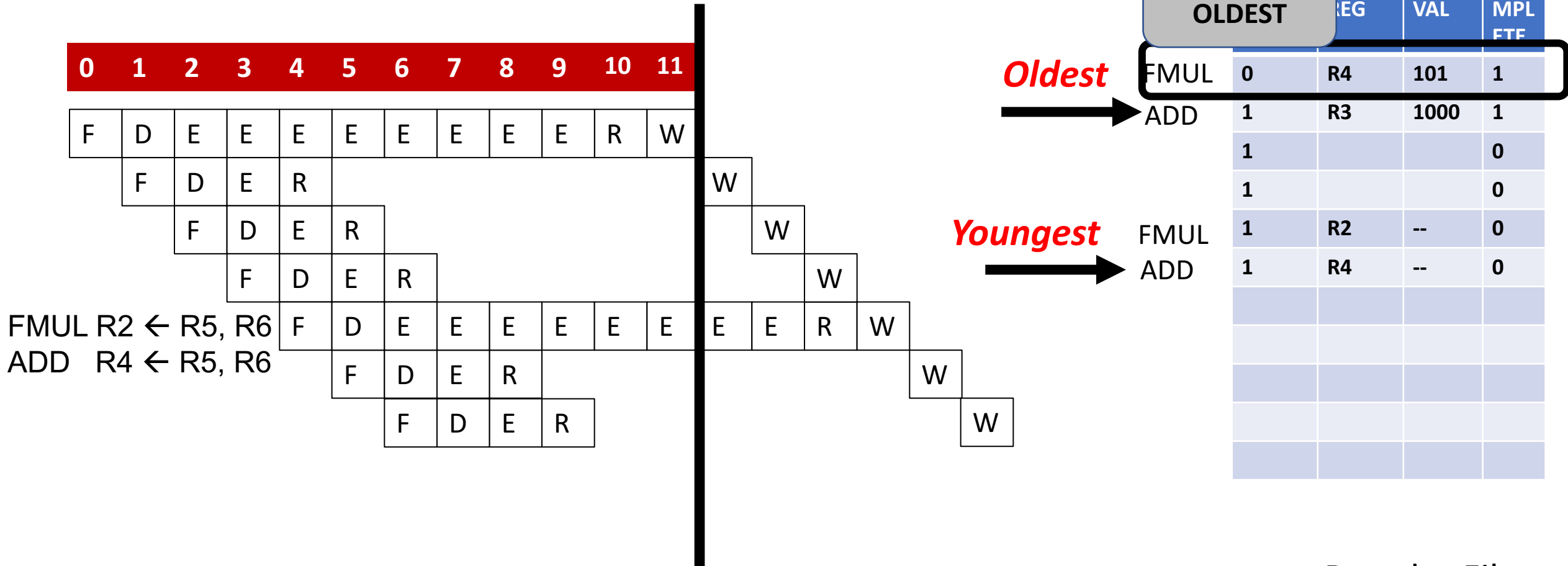
Reorder File

REORDER BUFFER: INDEPENDENT OPERATIONS

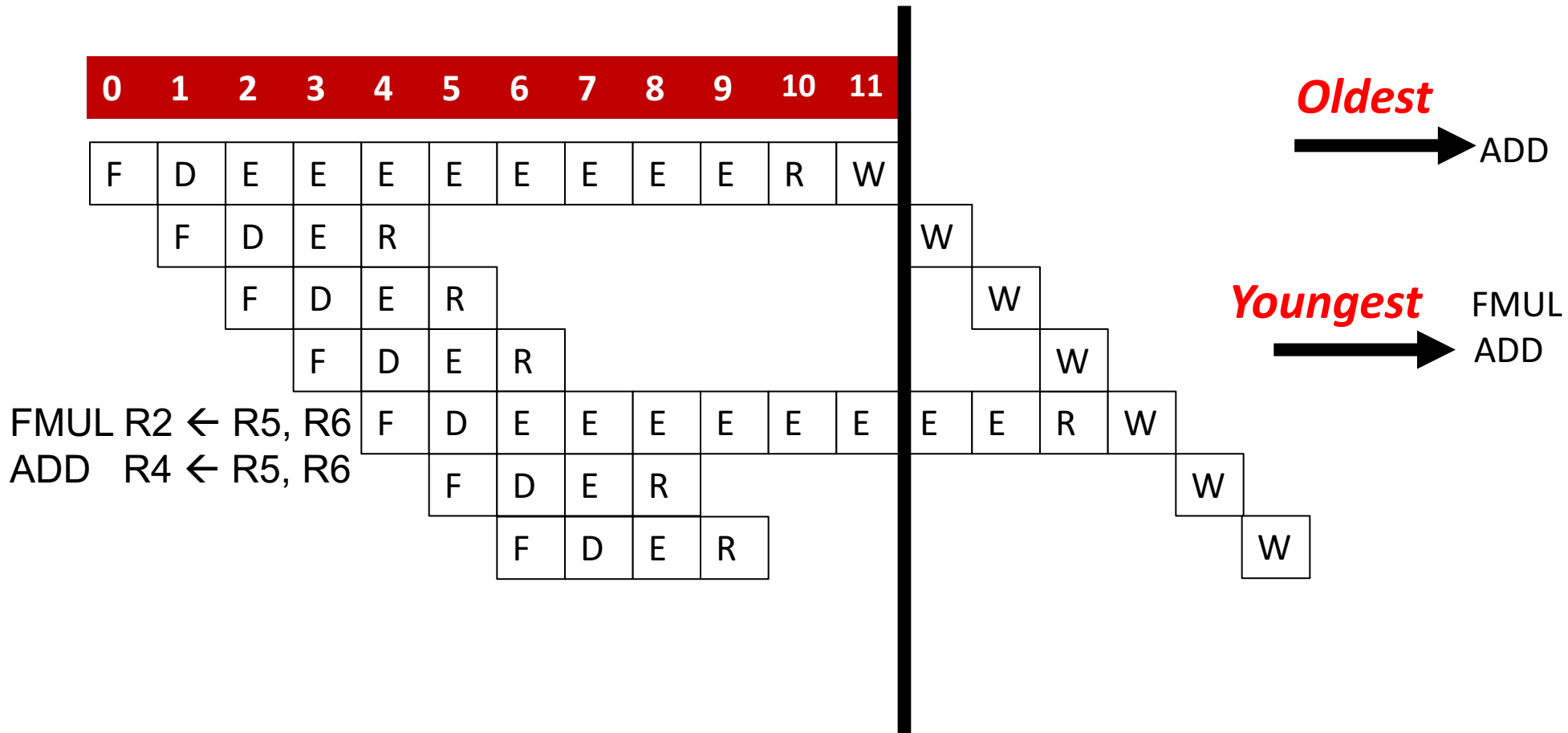


Reorder File

REORDER BUFFER: INDEPENDENT OPERATIONS



REORDER BUFFER: INDEPENDENT OPERATIONS



CYCLE 12

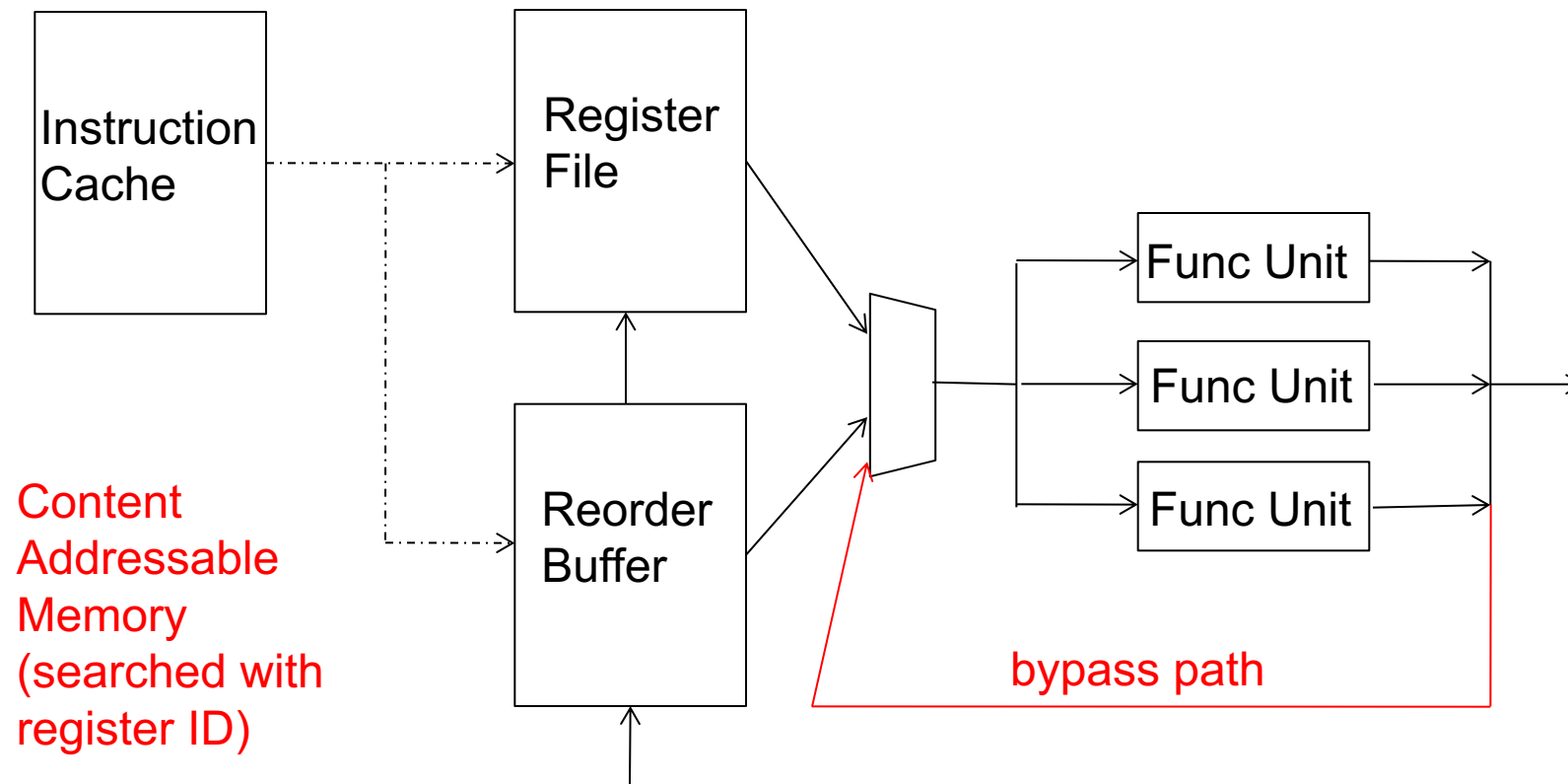
V	DEST REG	DEST VAL	CO MPL ETE
0			
1	R3	1000	1
1			0
1			0
1	R2	--	0
1	R4	--	0

Reorder File

What if a later operation needs a value in the reorder buffer?
Read reorder buffer in parallel with the register file. **How?**

REORDER BUFFER: HOW TO ACCESS?

- A register value can be in the register file, reorder buffer, (or bypass paths)



SIMPLIFYING REORDER BUFFER ACCESS

- Idea: Use indirection
- Access register file first
 - If register not valid, register file stores the ID of the reorder buffer entry that contains (or will contain) the value of the register
 - Mapping of the register to a ROB entry
- Access reorder buffer next
- What is in a reorder buffer entry?

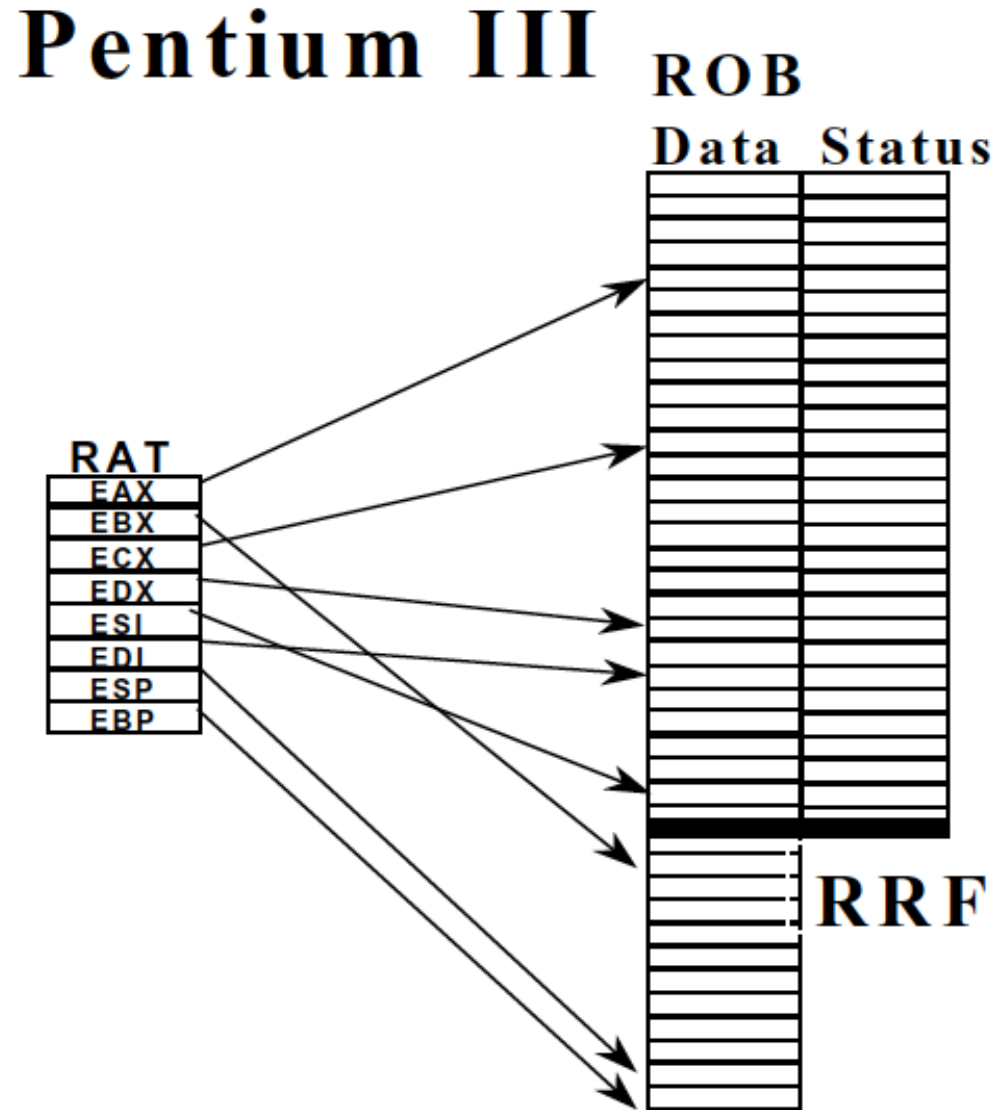
V	DestRegID	DestRegVal	StoreAddr	StoreData	BranchTarget	PC/IP	Control/valid bits
---	-----------	------------	-----------	-----------	--------------	-------	--------------------

- Can it be simplified further?

REORDER BUFFER PROS AND CONS

- Pro
 - Conceptually simple for supporting precise exceptions
- Con
 - Reorder buffer needs to be accessed to get the results that are yet to be written to the register file
 - CAM or indirection → increased latency and complexity

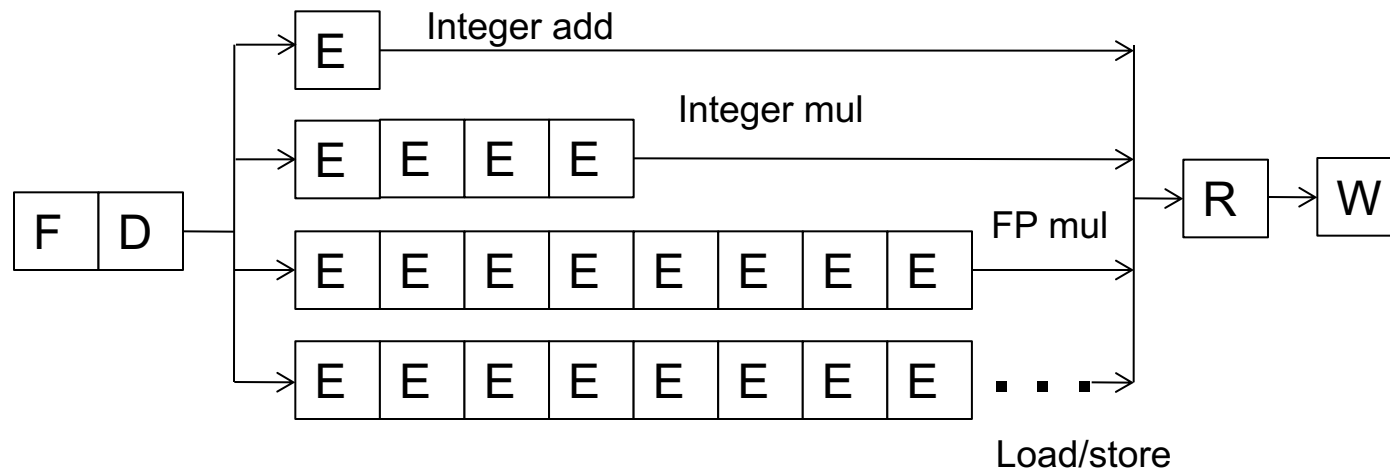
Reorder Buffer in Intel Pentium III



Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

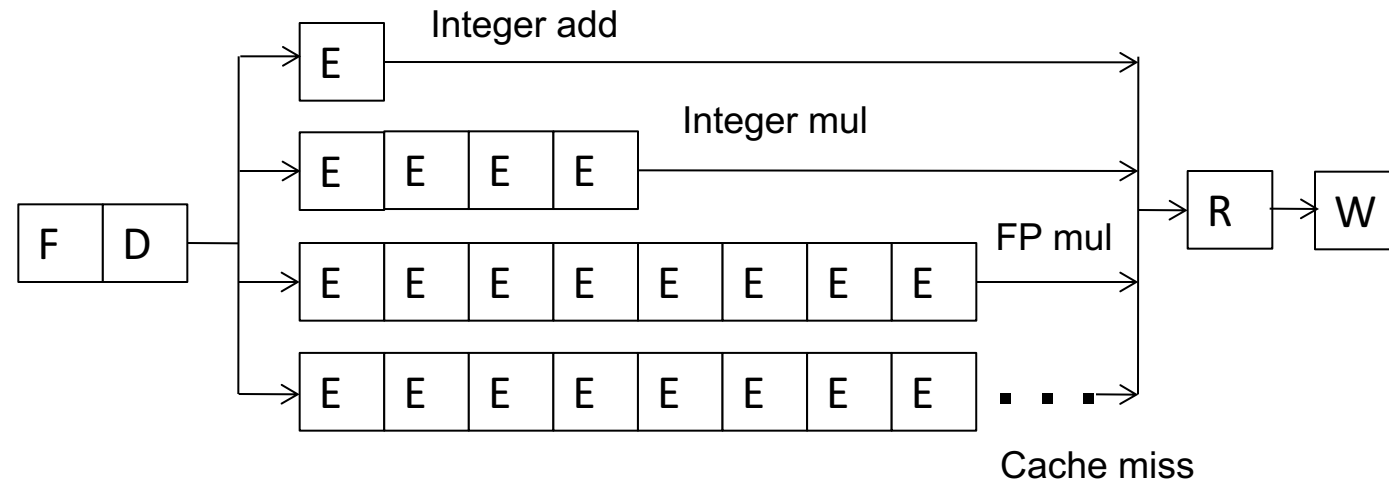
In-Order Pipeline with Reorder Buffer

- **Decode (D)**: Access regfile/ROB, allocate entry in ROB, check if instruction can execute, if so **dispatch** instruction
- **Execute (E)**: Instructions can complete out-of-order
- **Completion (R)**: Write result **to reorder buffer**
- **Retirement/Commit (W)**: Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler
- **In-order dispatch/execution, out-of-order completion, in-order retirement**



Out-of-Order Execution (Dynamic Instruction Scheduling)

AN IN-ORDER PIPELINE



- Problem: A true data dependency stalls dispatch of younger instructions into functional (execution) units
- Dispatch: Act of sending an instruction to a functional unit

CAN WE DO BETTER?

- What do the following two pieces of code have in common (with respect to execution in the previous design)?

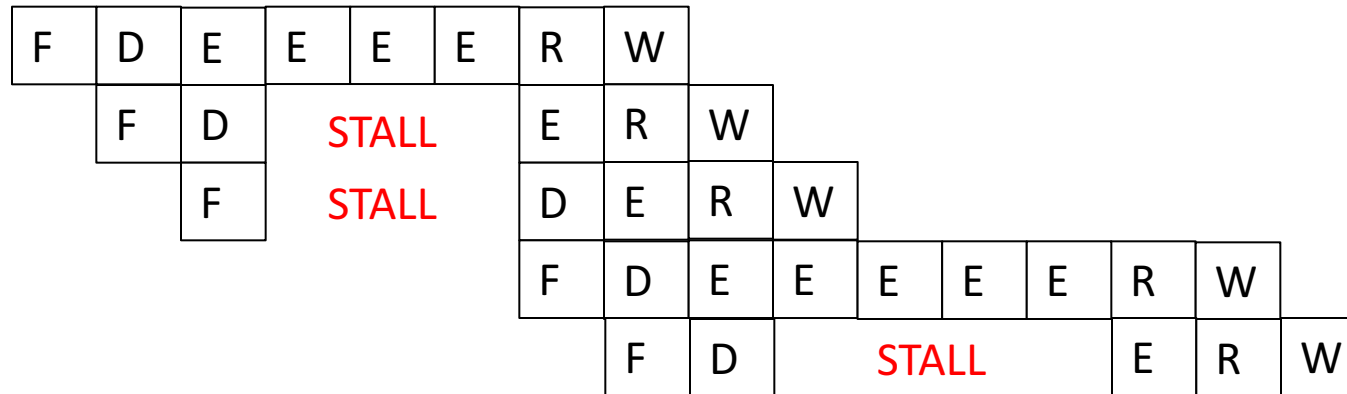
```
IMUL R3 ← R1, R2
ADD  R3 ← R3, R1
ADD  R1 ← R6, R7
IMUL R5 ← R6, R8
ADD  R7 ← R9, R9
```

```
LD   R3 ← R1 (0)
ADD  R3 ← R3, R1
ADD  R1 ← R6, R7
IMUL R5 ← R6, R8
ADD  R7 ← R9, R9
```

- **Answer: First ADD stalls the whole pipeline!**
 - ADD cannot dispatch because its source registers unavailable
 - Later **independent** instructions cannot get executed
- How are the above code portions different?
 - **Answer: Load latency is variable (unknown until runtime)**
 - What does this affect? Think compiler vs. microarchitecture

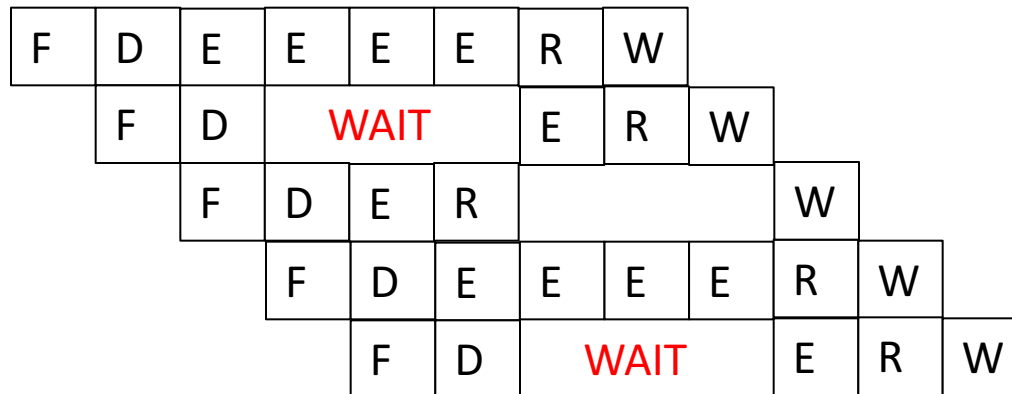
IN-ORDER VS. OUT-OF-ORDER DISPATCH

- In order dispatch + precise exceptions:



IMUL R3 ← R1, R2
 ADD R3 ← R3, R1
 ADD R1 ← R6, R7
 IMUL R5 ← R6, R8
 ADD R7 ← R3, R5

- Out-of-order dispatch + precise exceptions:



- 16 vs. 12 cycles

PREVENTING DISPATCH STALLS

- Any way to prevent dispatch stalls?
 - Dataflow: fetch and “fire” an instruction when its inputs are ready
 - Problem: in-order dispatch (scheduling, or execution)
 - Solution: out-of-order dispatch (scheduling, or execution)

TOMASULO'S ALGORITHM

- OoO with register renaming invented by Robert Tomasulo
 - Used in IBM 360/91 Floating Point Units
 - **Read:** Tomasulo, “**An Efficient Algorithm for Exploiting Multiple Arithmetic Units,**” IBM Journal of R&D, Jan. 1967.
- What is the major difference today?
 - **Precise exceptions:** IBM 360/91 did NOT have this
 - Patt, Hwu, Shebanow, “**HPS, a new microarchitecture: rationale and introduction,**” MICRO 1985.
 - Patt et al., “**Critical issues regarding HPS, a high performance microarchitecture,**” MICRO 1985.
- Variants are used in most high-performance processors
 - Initially in Intel Pentium Pro, AMD K5
 - Alpha 21264, MIPS R10000, IBM POWER5, IBM z196, Oracle UltraSPARC T4, ARM Cortex A15

- These slides are not covered in the class
- These are for students who want to know more

- What is the insight of OOO execution?

OUT-OF-ORDER EXECUTION (DYNAMIC SCHEDULING)

- Idea: Move the dependent instructions out of the way of independent ones (s.t. independent ones can execute)
 - Rest areas for dependent instructions: Reservation stations
- Monitor the source “values” of each instruction in the resting area
- When all source “values” of an instruction are available, “fire” (i.e. dispatch) the instruction
 - Instructions dispatched in dataflow (not control-flow) order
- Benefit:
 - **Latency tolerance**: Allows independent instructions to execute and complete in the presence of a long latency operation

The Von Neumann Model/Architecture

- Also called *stored program computer* (instructions in memory). Two key properties:
- Stored program
 - Instructions stored in a linear memory array
 - Memory is unified between instructions and data
 - The interpretation of a stored value depends on the control signals

When is a value interpreted as an instruction?
- Sequential instruction processing
 - One instruction processed (fetched, executed, and completed) at a time
 - Program counter (instruction pointer) identifies the current instr.
 - Program counter is advanced sequentially except for control transfer instructions

The Dataflow Model (of a Computer)

- Von Neumann model: An instruction is fetched and executed in **control flow order**
 - As specified by the **instruction pointer**
 - Sequential unless explicit control flow instruction
- Dataflow model: An instruction is fetched and executed in **data flow order**
 - i.e., when its operands are ready
 - i.e., there is **no instruction pointer**
 - Instruction ordering specified by data flow dependence
 - Each instruction specifies “who” should receive the result
 - An instruction can “fire” whenever all operands are received
 - Potentially many instructions can execute at the same time
 - Inherently more parallel

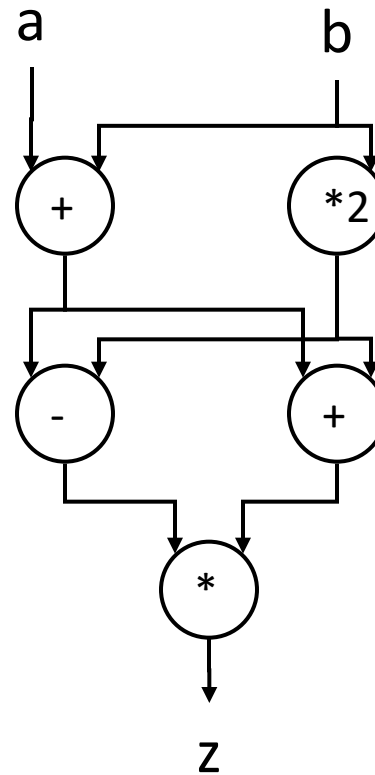
Von Neumann vs Dataflow

■ Consider a Von Neumann program

- What is the significance of the program order?
- What is the significance of the storage locations?

```
v <= a + b;  
w <= b * 2;  
x <= v - w  
y <= v + w  
z <= x * y
```

Sequential



Dataflow

■ Which model is more natural to you as a programmer?

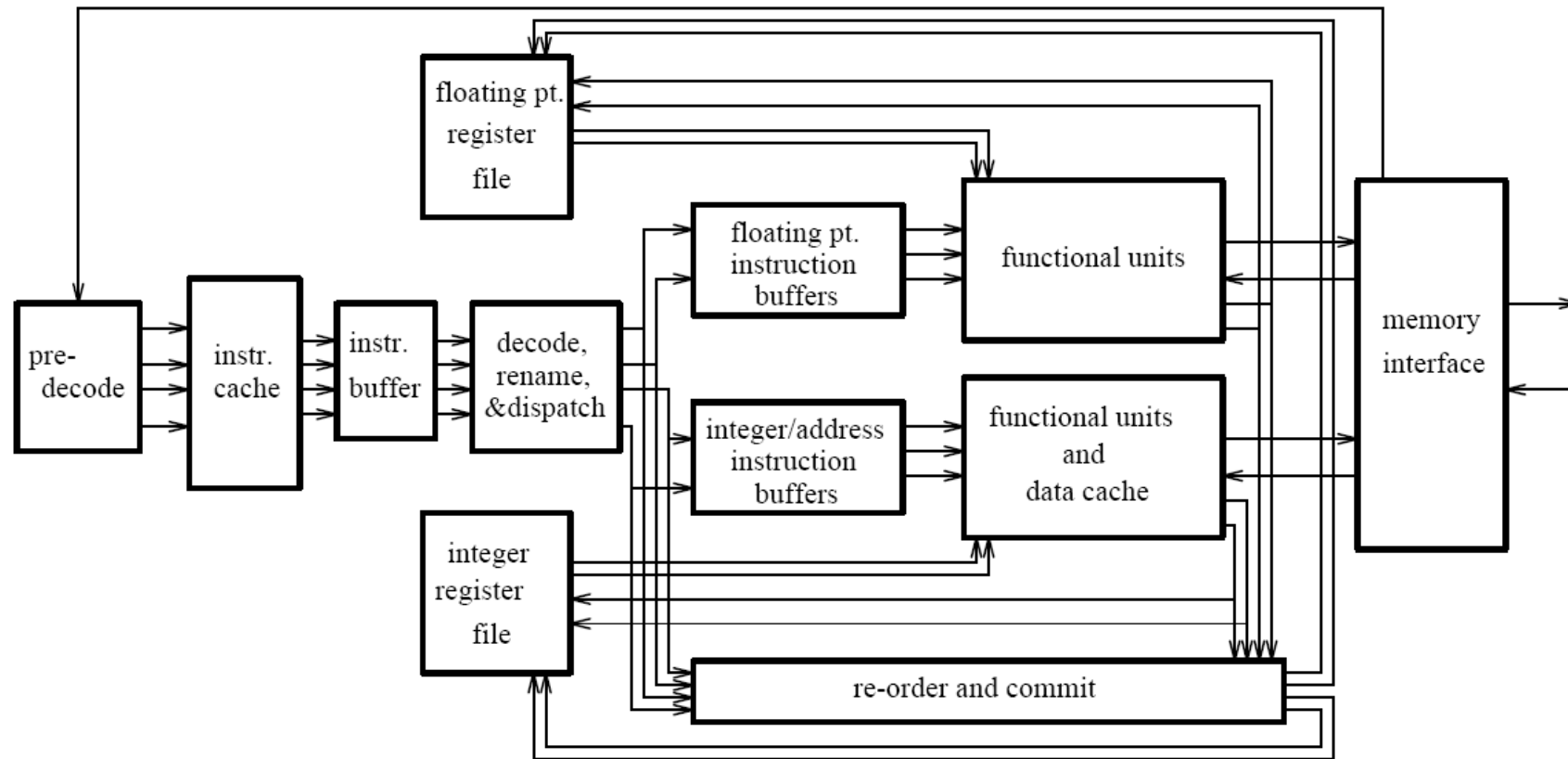
Data Flow Advantages/Disadvantages

- Advantages
 - Very good at exploiting **irregular parallelism**
 - Only real dependencies constrain processing
- Disadvantages
 - Debugging difficult (no precise state)
 - Interrupt/exception handling is difficult (what is precise state semantics?)
 - Too much parallelism? (Parallelism control needed)
 - High bookkeeping overhead (tag matching, data storage)
 - Memory locality is not exploited

OOO EXECUTION: RESTRICTED DATAFLOW

- An out-of-order engine dynamically builds the dataflow graph of a piece of the program
 - which piece?
- The dataflow graph is limited to the instruction window
 - Instruction window: all decoded but not yet retired instructions
- Can we do it for the whole program?
- Why would we like to?
- In other words, how can we have a large instruction window?

GENERAL ORGANIZATION OF AN OOO PROCESSOR



- Smith and Sohi, “[The Microarchitecture of Superscalar Processors](#),” Proc. IEEE, Dec. 1995.

TOMASULO'S MACHINE: IBM 360/91

