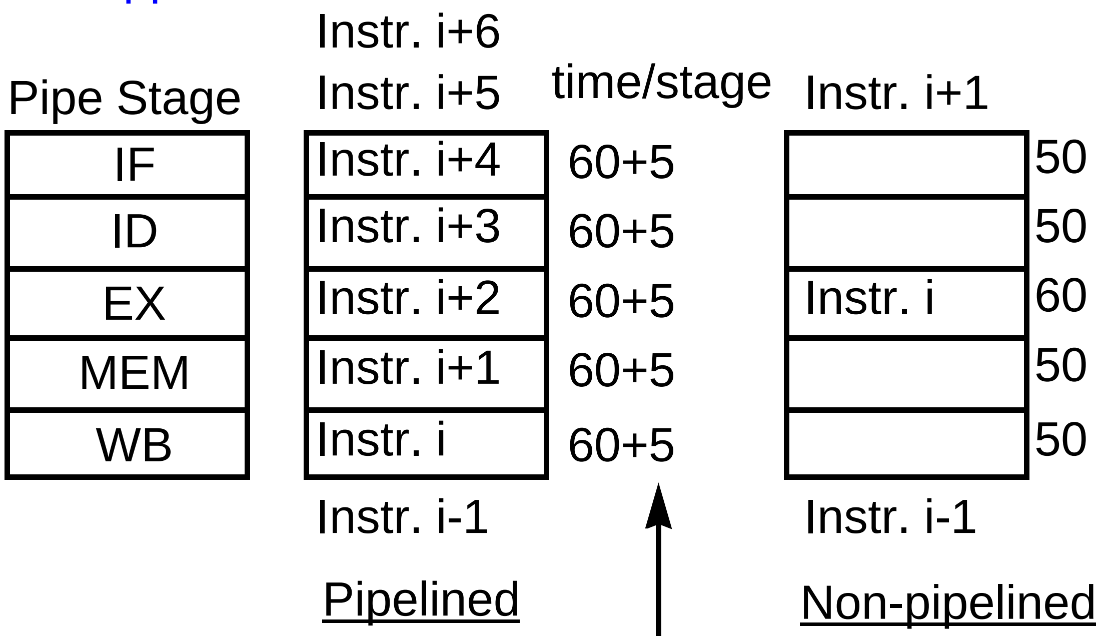
Pipelining

The key implementation technique used to make fast CPUs.

Multiple instructions are overlapped in execution.

All stages must be ready to proceed at the same time.

The time to move instruction one step down the pipeline (called machine cycle) is determined by the slowest pipe stage.

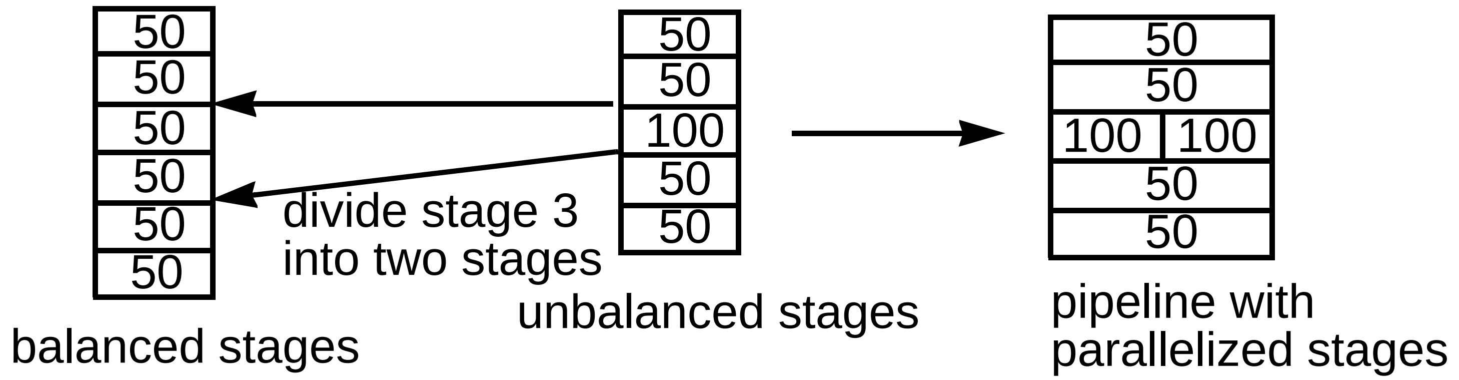


5 nsec pipeline overhead
due to synchronization among stages

$$\text{Speedup} = \frac{\text{AverageInstructionTimeWithoutPipeline}}{\text{AverageInstructionTimeWithPipline}} = \frac{260}{65} = 4$$

Pipeline Designer’s Goal and limits

- Goal $\longrightarrow$ Balance the length of the pipeline stages.



- More stages$\rightarrow$fewer operations/stage$\rightarrow$smaller clock cycle time/stage $\rightarrow$allow to maintain a low CPI.
- But clock cycle time > (latch overhead+clock skew)
  and there is a limited number of operations to performance per instruction.
  —Latches are used among stages to keep instruction’s intermediate values.
  —Clock signals reach different stages at different time$\equiv$clock skew.
- Pipelining increases the CPU instruction throughput (no. of instructions/sec) but does not reduce the execution time for an individual instruction.
  In fact, the execution time increases due to the pipeline overhead.
  (260 vs. 325 nsec).

DLX without Pipelining The five cycle

Instruction Fetch (IF) Cycle:

$IR \leftarrow Mem[PC]$

$NPC \leftarrow PC+4$

Instruction Decoding/Register Fetch (ID) Cycle:

$A \leftarrow Regs[IR_{6..10}]$

$B \leftarrow Regs[IR_{11..15}]$

$Imm \leftarrow ((IR_{16})^{16}\#\#IR16..31)$

A and B are wo temporary registers.
Decoding is done in parallel with the reading of registers A and B.

Execution/effective address (EX) cycle:

DLX without Pipelining
EX and MEM Instruction Cycle

Execution/effective address (EX) cycle: (depend on instruction type)

- Memory reference: (Load or Store instructions)
  ALUoutput $\leftarrow$ A + Imm; (compute the address)
- Register-Register ALU instructions: (op indicated by instruction decoding)
  ALUoutput $\leftarrow$ A op B;
- Register-Immediate ALU instructions: (LI R3, #3)
  ALUoutput $\leftarrow$ A op Imm;
- Bracnch:
  ALUoutput $\leftarrow$ NPC + Imm; (branch target address, NPC=PC+4)
  Cond $\leftarrow$ (A op 0); (here op can be EQ or NE)

Memory access/branch completion (MEM) cycle:

- Memory reference:
  LMD $\leftarrow$ Mem[ALUoutput] or (load instruction)
  Mem[ALUoutput] $\leftarrow$ B; (store instruction);
- Branch:
  if (cond) PC $\leftarrow$ ALUoutput else PC $\leftarrow$ NPC (next istruction)

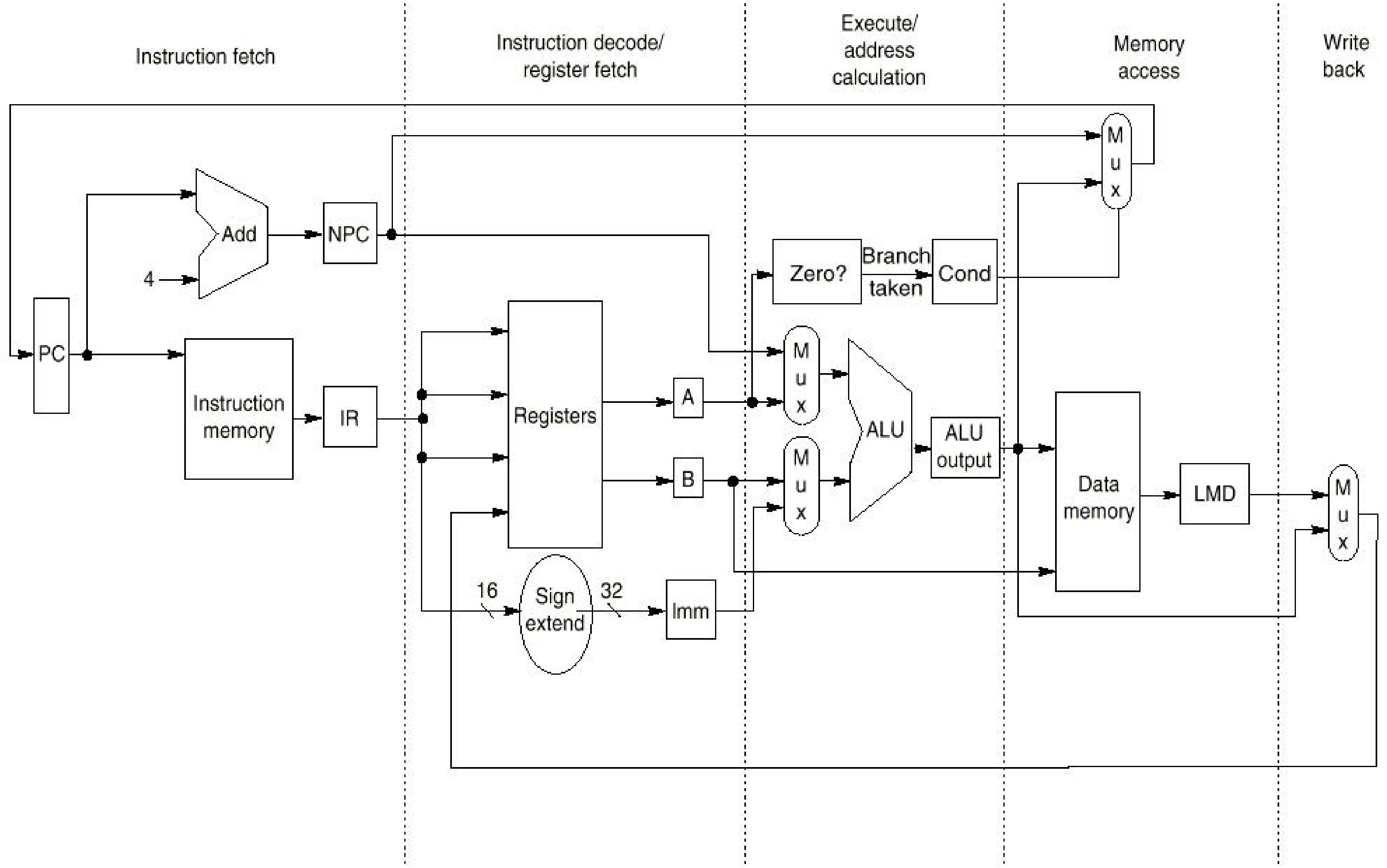
DLX without Pipelining
WB Instruction Cycle

Write-back (WB) cycle:

- Load instruction:
  $Regs[IR_{11..15}] \leftarrow LMD;$
- Register-Register ALU instructions:
  $Regs[IR_{16..20}] \leftarrow ALUoutput;$
- Register-Immediate ALU instructions: (LI R3, #3)
  $Regs[IR_{11..15}] \leftarrow ALUoutput;$

Note that in this implementation, branch instructions take 4 cycles, others take 5 cycles.

DLX Datapath



Basic DLX Pipeline

					Clock #				
instruction #	1	2	3	4	5	6	7	8	9
instruction i	IF	ID	EX	MEM	WB				
instruction i+1		IF	ID	EX	MEM	WB			
instruction i+2			IF	ID	EX	MEM	WB		
instruction i+3				IF	ID	EX	MEM	WB	
instruction i+4					IF	ID	EX	MEM	WB

IF: Instruction Fetch

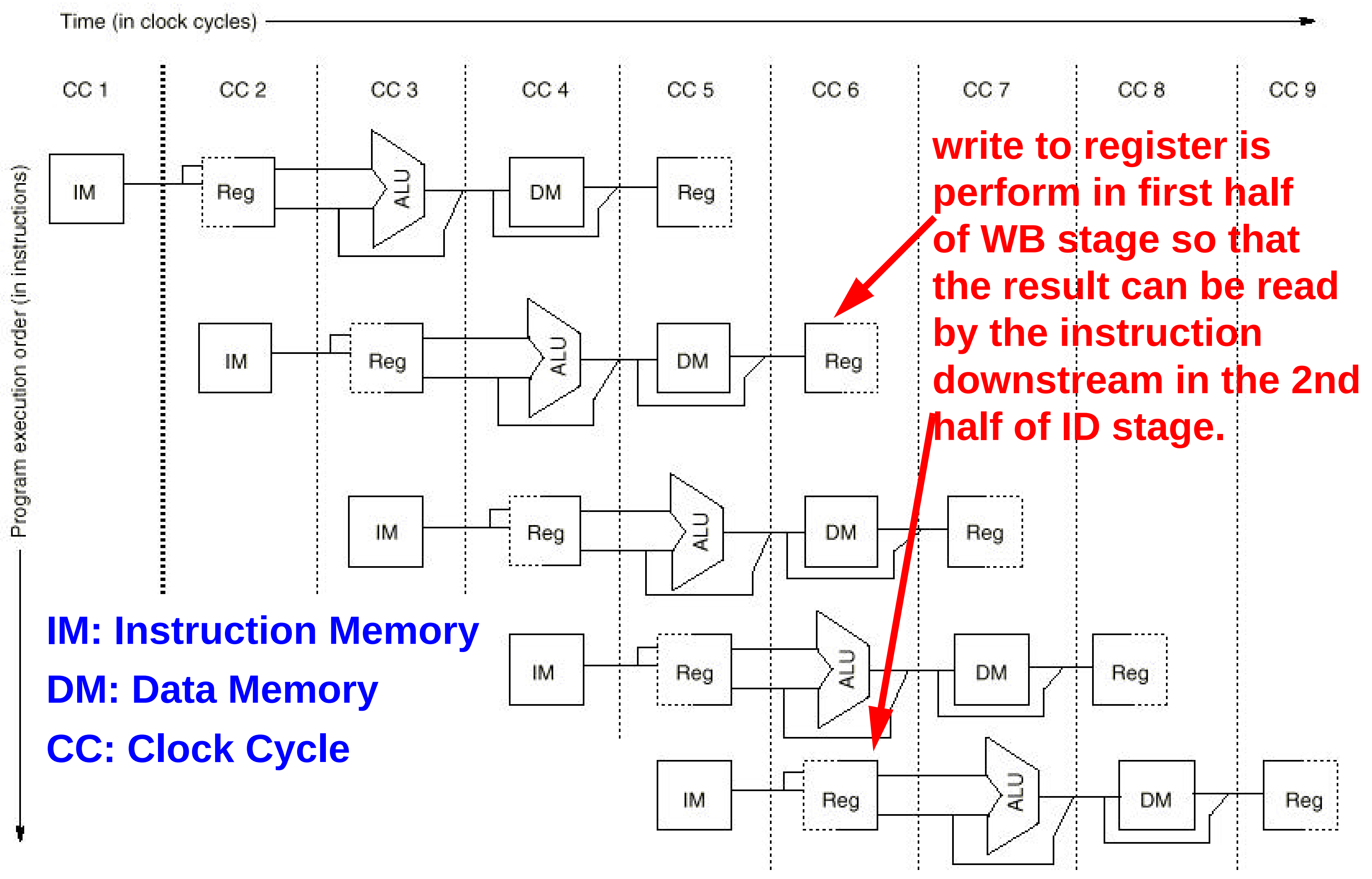
ID: Instruction Decode

EX: Execution stage

MEM: Memory Stage

WB: Write Back (to register)

Pipeline Stages and Their Resource Utilization



write to register is perform in first half of WB stage so that the result can be read by the instruction downstream in the 2nd half of ID stage.

IM: Instruction Memory

DM: Data Memory

CC: Clock Cycle

Pipeline Registers of DLX

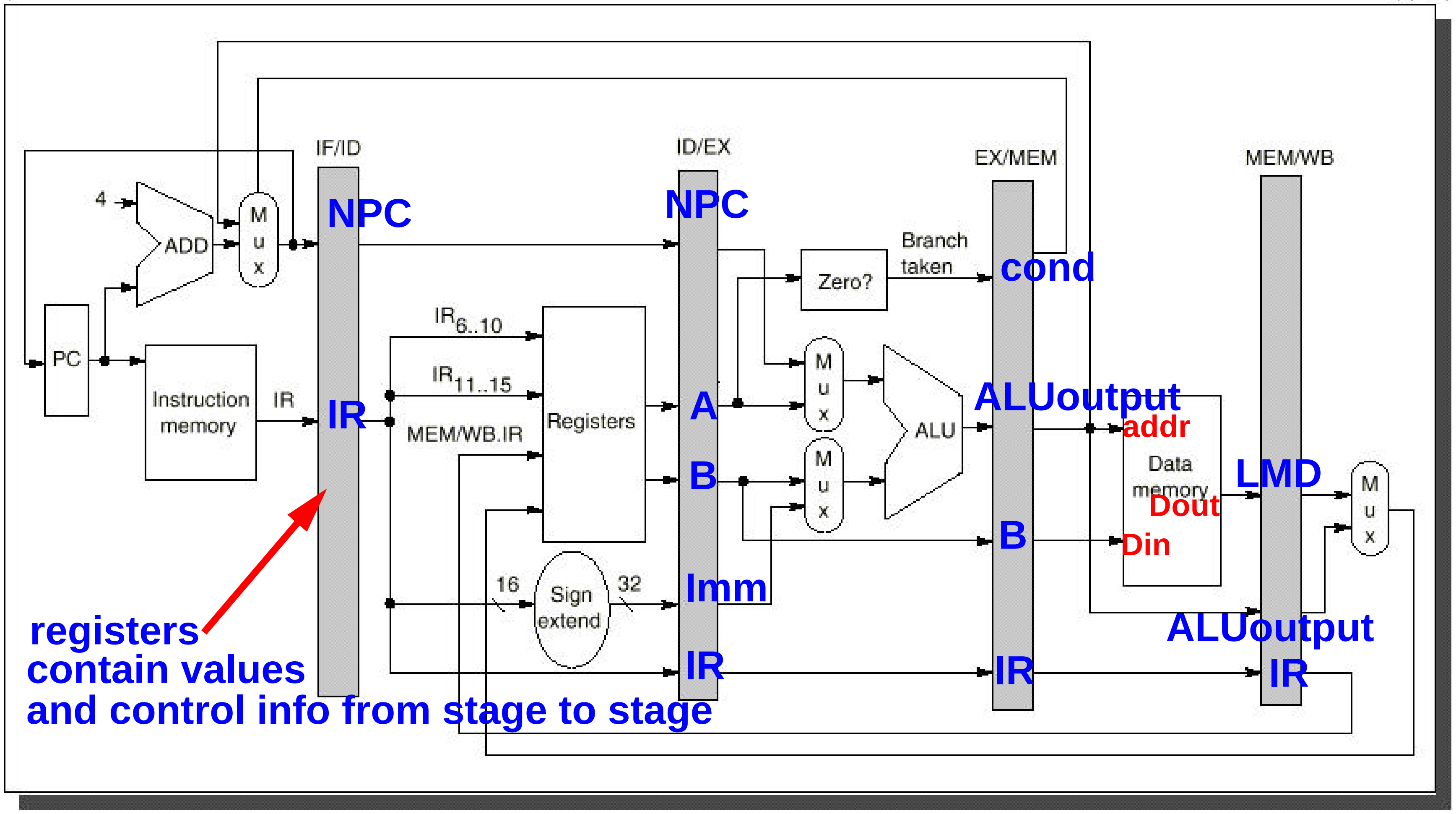


FIGURE 3.4 The datapath is pipelined by adding a set of registers, one between each pair of pipe stages.

Instruction Execution on DLX Pipeline
ADDI R1, R0, #1000

0	5 6	10 11	15 16	31
Opcode	rs1	rd	Imm	
ADD	0	1	1000	

IF: IF/ID.IR $\leftarrow$ Mem[PC];
 IF/ID.NPC, PC $\leftarrow$ (if EX/MEM.cond {EX/MEM.NPC} else {PC+4});
 (at the initialization of pipeline, EX/MEM.cond is set to 0.)

ID: ID/EX.NPC $\leftarrow$ IF/ID.NPC; ID/EX.IR $\leftarrow$ IF/ID.IR;
 ID/EX.A $\leftarrow$ Regs[IFI/ID.IR$_{6..10}$=0]; ID/EX.B $\leftarrow$Regs[IF/ID.IR$_{11..15}$=1](not used)
 ID/EX.Imm $\leftarrow$ (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$;

EX: EX/MEM.IR $\leftarrow$ ID/EX.IR;
 EX/MEM.ALUoutput $\leftarrow$ ID/EX.A + ID/EX.Imm
 EX/MEM.cond $\leftarrow$ 0; (indicating not a branch)

MEM: MEM/WB.IR $\leftarrow$ EX/MEM.IR;(why waste MEM step, not skip?)
 MEM/WB.ALUoutput $\leftarrow$ EX/MEM.ALUoutput;

WB: Regs[MEM/WB.IR$_{11..15}$] $\leftarrow$ MEM/WB.ALUoutput

Instruction Execution on DLX Pipeline
LW R2, 0(R1)

0 5	6 10	11 15	16 31
Opcode	rs1	rd	Imm
LW	1	2	0

IF: IF/IR.IR ← Mem[PC];

IF/ID.NPC, PC ← (if EX/MEM.cond {EX/MEM.NPC} else {PC+4});
(At the initialization of pipeline, EX/MEM.cond is set to 0.)

ID: ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR;

ID/EX.A ← Regs[IFI/ID.IR$_{6..10}$=1]; ID/EX.B ← Regs[IF/ID.IR$_{11..15}$];(not used)

ID/EX.Imm ← (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$;

EX: EX/MEM.IR ← ID/EX.IR;

EX/MEM.ALUoutput ← ID/EX.A + ID/EX.Imm; (compute the address)
EX/MEM.cond ← 0; (indicating not a branch)

MEM: MEM/WB.IR ← EX/MEM.IR;

MEM/WB.LMD ← Mem[EX/MEM.ALUoutput];

WB: Regs[MEM/WB.IR$_{11..15}$=2] ← MEM/WB.LMD

Instruction Execution on DLX Pipeline BEQZ R2, L

0 5	6 10	11 15	16 31
Opcode	rs1	rd	Imm
BEQZ	2	not used	4

IF: IF/IR.IR $\leftarrow$ Mem[PC];
IF/ID.NPC, PC $\leftarrow$ (if EX/MEM.cond {EX/MEM.NPC} else {PC+4});
(at the initialization of pipeline, EX/MEM.cond is set to 0.)

ID: ID/EX.NPC $\leftarrow$ IF/ID.NPC; ID/EX.IR $\leftarrow$ IF/ID.IR;
ID/EX.A $\leftarrow$ Regs[IFI/ID.IR$_{6..10}$=2]; ID/EX.B $\leftarrow$ Regs[IF/ID.IR$_{11..15}$];(not used)
ID/EX.Imm $\leftarrow$ (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$;

EX: EX/MEM.ALUoutput $\leftarrow$ ID/EX.NPC + ID/EX.Imm; (branch target address)
EX/MEM.cond $\leftarrow$ (ID/EX.A == 0)

Note that EX/MEM.cond affect the PC value, therefore the fetch of next instruction (the 3rd instruction after BEQZ). The 1st and 2nd instruction after BEQZ, in IF and ID stages, will have to be aborted if the branch is taken.

Instruction Execution on DLX Pipeline
SW 30(R1), R3

0 5	6 10	11 15	16 31
Opcode	rs1	rd	Imm
LW	1	3	30

IF: IF/IR.IR $\leftarrow$ Mem[PC];
IF/ID.NPC, PC $\leftarrow$ (if EX/MEM.cond {EX/MEM.NPC} else {PC+4});
(At the initialization of pipeline, EX/MEM.cond is set to 0.)

ID: ID/EX.NPC $\leftarrow$ IF/ID.NPC; ID/EX.IR $\leftarrow$ IF/ID.IR;
ID/EX.A $\leftarrow$ Regs[IFI/ID.IR$_{6..10}$=1]; ID/EX.B $\leftarrow$ Regs[IF/ID.IR$_{11..15}$=3];
ID/EX.Imm $\leftarrow$ (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$;

EX: EX/MEM.IR $\leftarrow$ ID/EX.IR; EX/MEM.B $\leftarrow$ ID/EX.B;
EX/MEM.ALUoutput $\leftarrow$ ID/EX.A + ID/EX.Imm (compute address)
EX/MEM.cond $\leftarrow$ 0; (indicating not a branch)

MEM: MEM/WB.IR $\leftarrow$ EX/MEM.IR;
Mem[EX/MEM.ALUoutput] $\leftarrow$ EX/MEM.B;

Pipeline Hazard—
The Major Hurdle of Pipelining

Hazards: Situations that prevent the next instruction from execution during its designated clock cycle.

Three classes of hazards:

- Structural hazards—arise from resource conflicts when the hardware can't support all possible combinations of instructions in simultaneous, overlapped execution.
- Data hazards—arise when an instruction depends on the result of a previous instruction currently in the pipeline.
- Control hazards—arise from changing the PC such as branch instructions
  For branch-taken situation, the instruction fetch is not in regular sequence, the target instruction is not available.

Simple solution to the hazards $\rightarrow$ stall the pipeline.
All instructions before the stalled instruction continue
All instructions after the stalled instruction are also stalled.

Stall the pipeline $\rightarrow$ Degrade the performance.

Pipeline Performance

$$\text{PipelineSpeedup} = \frac{\text{ClockCycleTimeWithoutPipelining} \times \text{CPIWithoutPipelining}}{\text{ClockCycleTimeWithPipelining} \times \text{CPIWithPipelining}}$$

$$\text{A pipelined machine's idealCPI} = \frac{\text{CPIWithoutPipelining}}{\text{PipelineDepth}}$$

$$\text{PipelineSpeedup} = \frac{\text{ClockCycleTimeWithoutPipelining} \times \text{idealCPI} \times \text{PipelineDepth}}{\text{ClockCycleTimeWithPipelining} \times \text{CPIWithPipelining}}$$

$$\text{CPIWithPipelining} = \text{idealCPI} + \text{PipelineStallClockCyclesPerInstruction}$$

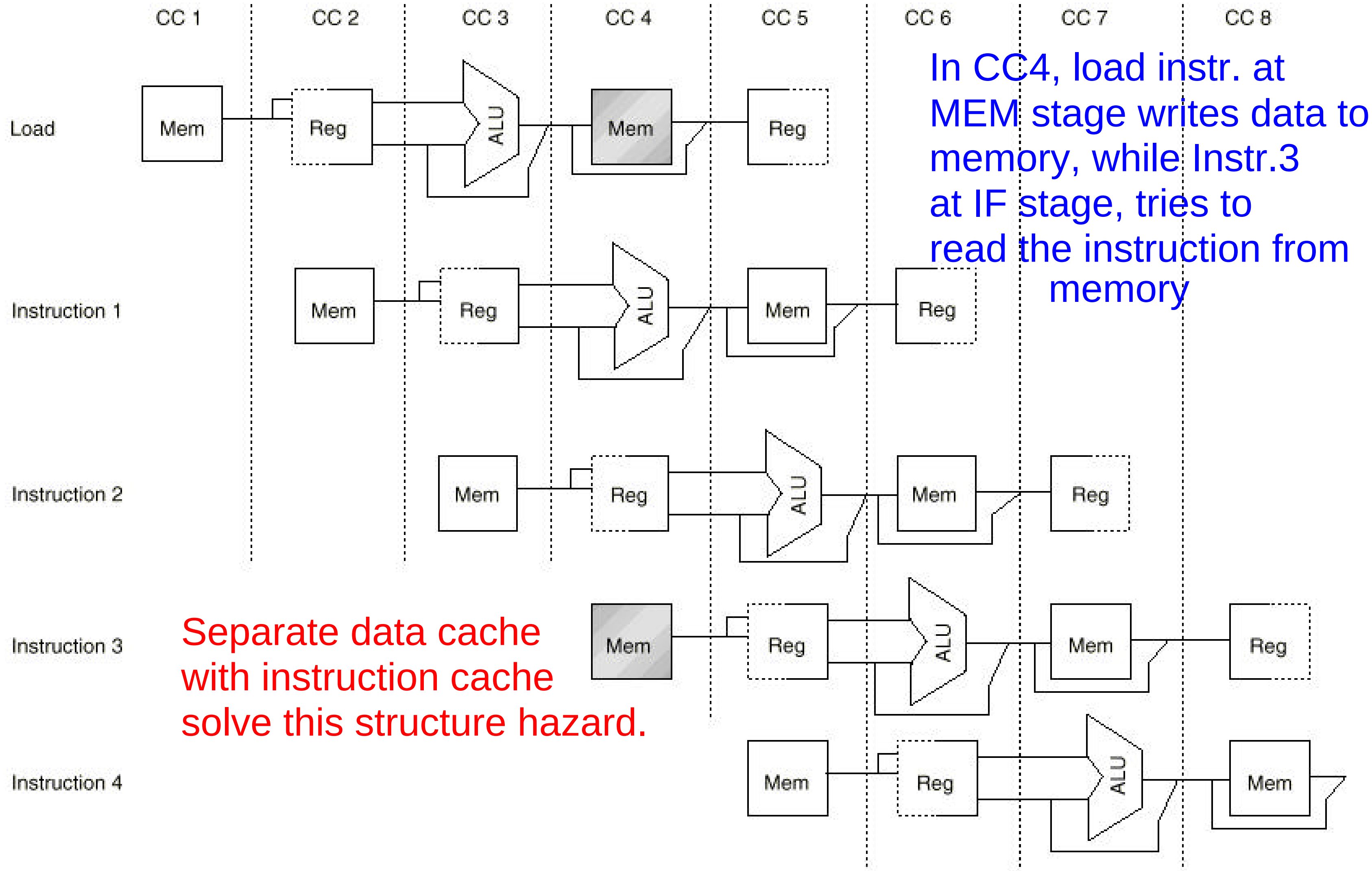
PipelineSpeedup=

$$\frac{\text{ClockCycleTimeWithoutPipelining} \times \text{idealCPI} \times \text{PipelineDepth}}{\text{ClockCycleTimeWithPipelining} \times (\text{idealCPI} + \text{PipelineStallClockCyclesPerInstruction})}$$

If we ignore the pipeline overhead, then

$$\text{PipelineSpeedup} = \frac{\text{idealCPI} \times \text{PipelineDepth}}{\text{idealCPI} + \text{PipelineStallClockCyclesPerInstruction}}$$

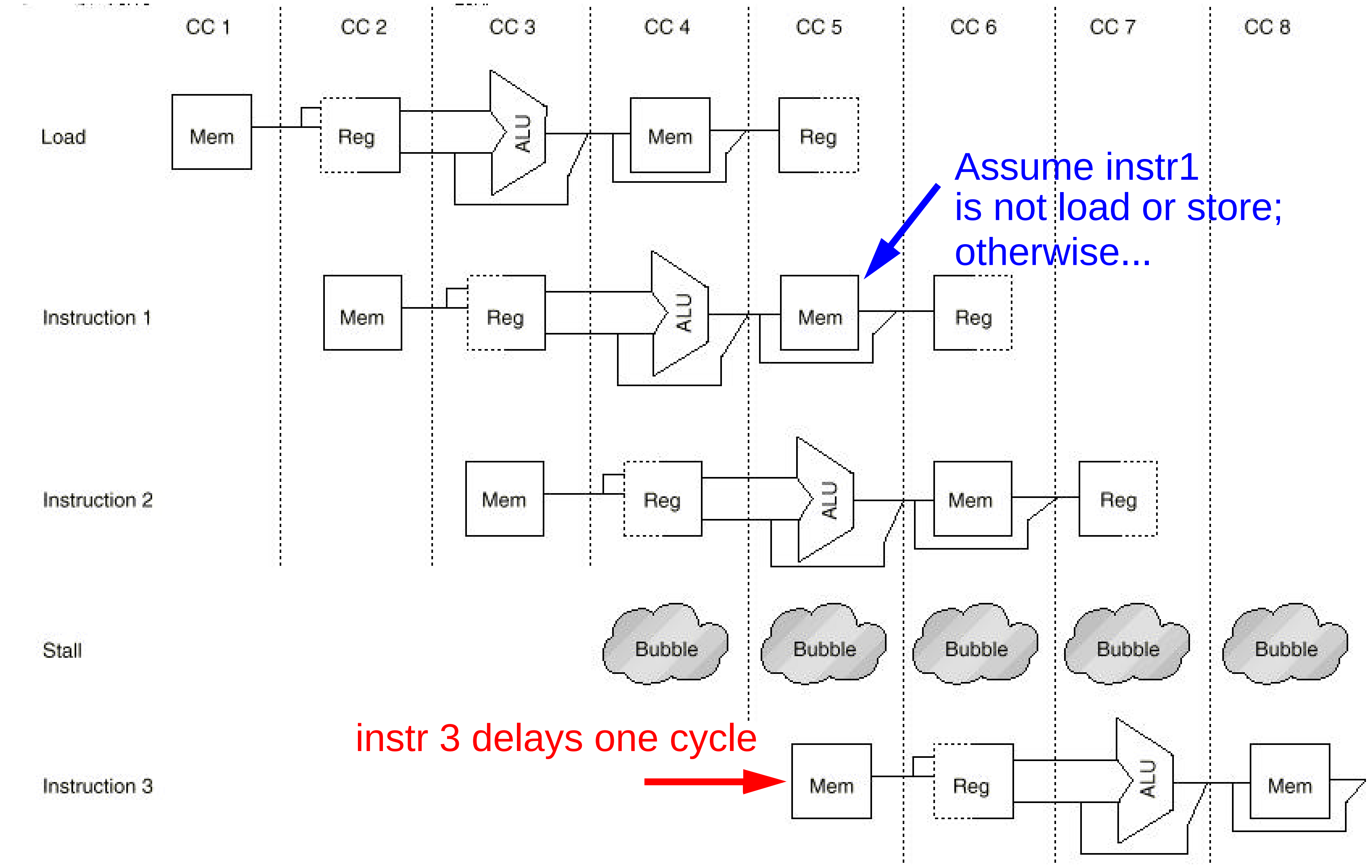
Structure Hazard: load with a single memory port



In CC4, load instr. at MEM stage writes data to memory, while Instr.3 at IF stage, tries to read the instruction from memory

Separate data cache with instruction cache solve this structure hazard.

Stall Instruction Execution



Why would a designer allow structural hazards?

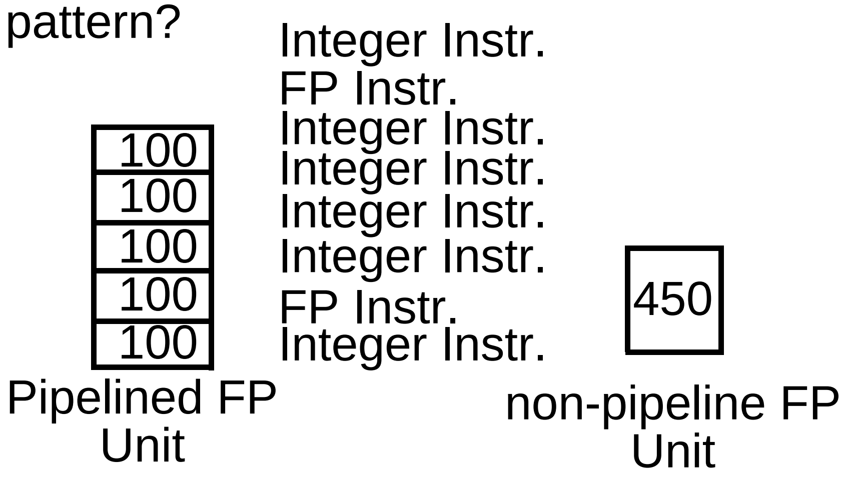
- Reduce cost
- Reduce the latency of the unit.

A non-pipelined or not fully pipelined unit has a shorter total delay than a fully pipelined unit.

Example. Floating point units in both CDC 7600 and MIPS R2010 choose shorter latency approach.

Note that unless there are high frequency or high concentration of consecutive floating point instructions, we will not be able to benefit from a fully pipelined floating point unit.

Which of the two FP units performs better with the following instruction stream pattern?



Data Hazards

The order of access to operands is changed by the pipeline versus the normal order encountered by sequentially executing instructions.

Example. A data hazard involving register operands

ADD	R1, R2, R3; R1$\leftarrow$ (R2)+(R3)
SUB	R4, R5, R1; R4 $\leftarrow$ (R5)-(R1)

ADD Writes the value of R1 in the WB pipe stage at cycle 5,
but SUB reads the value of R1 in the ID pipe stage at cycle 3.
If originally, (R1)=1, (R2)=2, (R3)=3, (R5)=5, what will be the value of R4, after the execution of the above instruction sequence on the DLX pipeline machine at page 136 that does not handle the data hazard?

Forwarding Technique for Solving Data Hazards

The forwarding technique is also called bypassing or short-circuiting.

- The ALU result is fed back to the ALU input latches for the next instruction.
- If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as ALU input rather than the value read from the register file.

Reduce the number of instructions that must be bypassed by

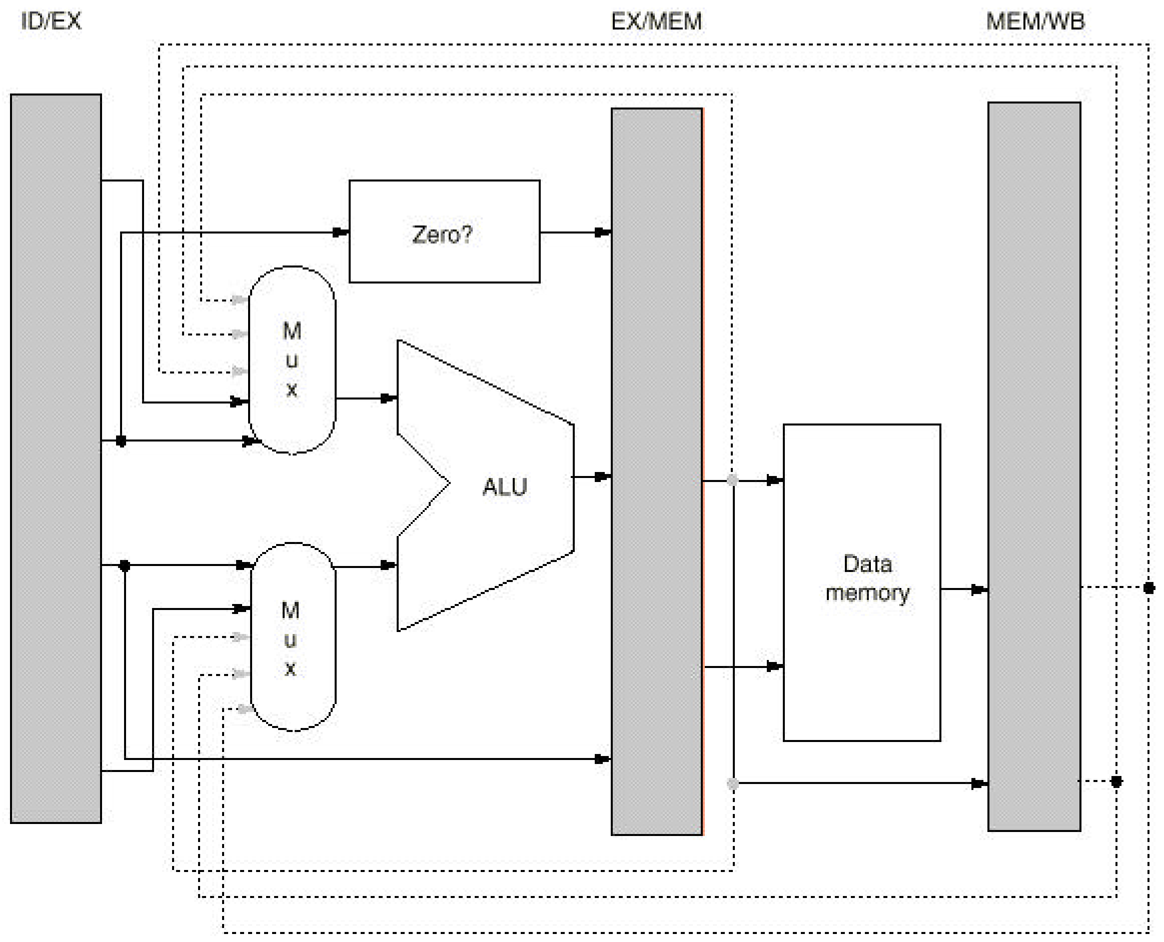
- Do the register writes in the first half of WB stage.
- Do the register reads in second half of ID stage.

In Fig. 3.10, this reduces the no. of instructions that must be bypassed from 3 to 2.

The Bypass Unit

It requires

- Latches, Multiplexers (MUX).
- Comparators which examine if adjacent instructions share a destination and a source.
- ALU result buffers


ID/EX
EX/MEM
MEM/WB
Zero?
Mux
ALU
Mux
Data memory

Other Types of Data Hazard

- The above data hazards involve register operands.
- The accesses to memory operands in some pipelined machines can also create data hazards.
- Will memory reference hazards happen in DLX?
- Cache misses could cause the memory references to get out of order if we allow the processor to continue working on later instructions.
  Give a scenario.
  $\rightarrow$ Solution: stall the pipeline until the missed data is accessed.

Forwarding results to more than one unit

```
Example:ADD         R1, R2, R3
        SW          25(R1), R1
```

We need to forward the value of R1 to ALU for the effective address 25(R1) and as data for preparing the storing.

Recall that the EX stage of the SW instruction includes the following operations:

- EX/MEM.ALUoutput$\leftarrow$ID/EX.A+(ID/EX.IR$_{16}$)^{16}##ID/EX.IR$_{16..31}$ and
- EX/MEM.B$\leftarrow$ID/EX.B

Three Types of Data Hazards

consider two instructions i and j, with i occurring before j.

- RAW (Read After Write)
  j tries to read a source before i writes it.
  $\rightarrow$ j incorrectly gets old value.
- WAW (Write After Write)
  j tries to write an operand before it is written by i.
  $\rightarrow$ leaving the operand with old value written by i.

Assume load memory stage takes two cycles						
LW R1,0(R2)	IF	ID	EX	MEM1	MEM2	WB
ADD R1, R2, R3		IF	ID	EX	WB	

- WAR (Write After Read)
  j tries to write a destination before it is read by i.
  $\rightarrow$ i incorrectly get new value.

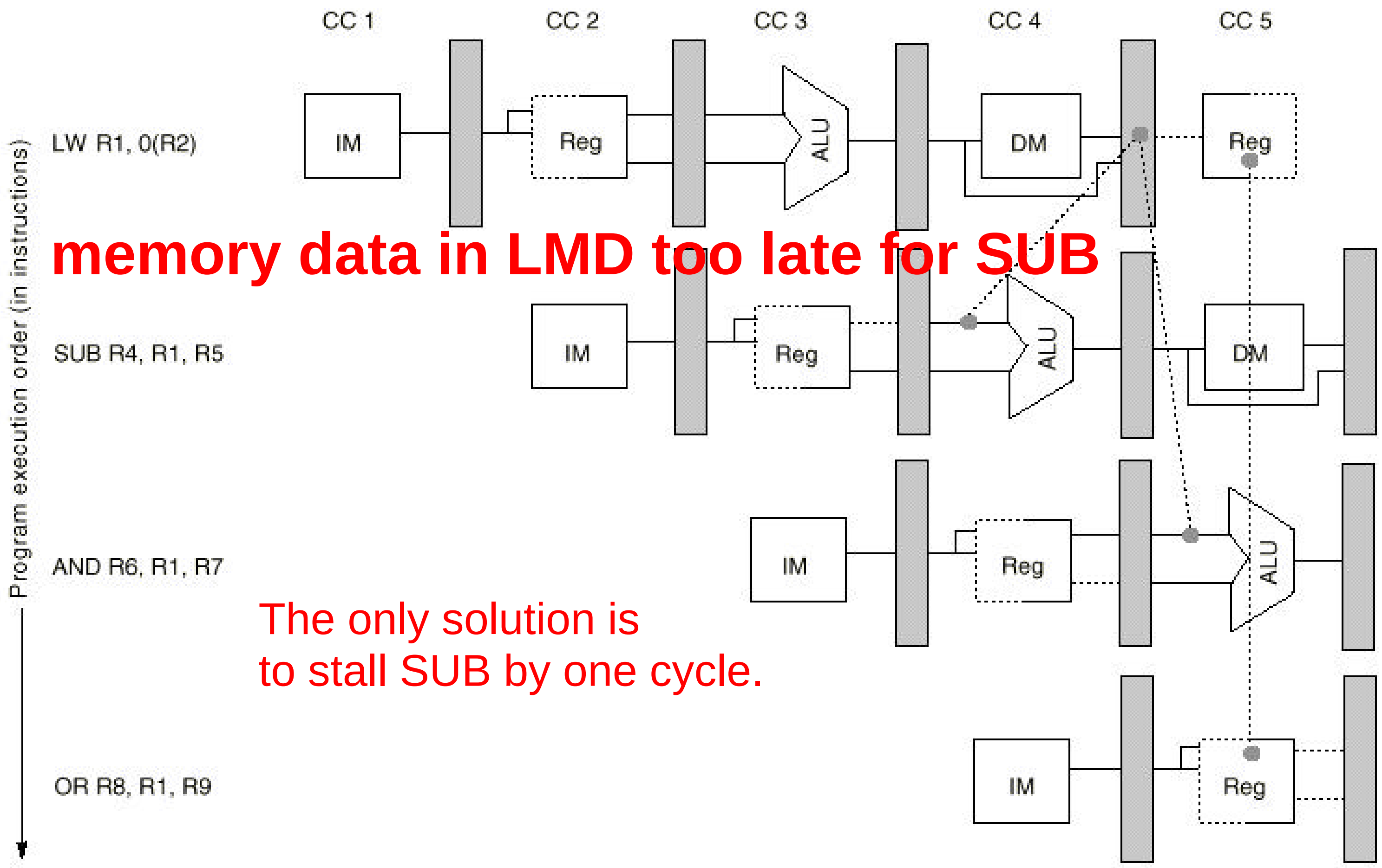
In DLX, which one will happen?

- DLX reads are early in the ID pipe stage and writes are late in the WB stage.
  $\rightarrow$ No WAR hazard.
- WAW happens only when there are more than one write in the pipeline.
  DLX writes registers only in the WB stage and stall pipeline when cache miss or long MEM cycles.
  $\rightarrow$ Therefore it avoids the WAW hazard.
- DLX only has RAW hazard.
- RAR (Read After Read) is not a hazard.

Pipeline Interlock for Load Data Hazard

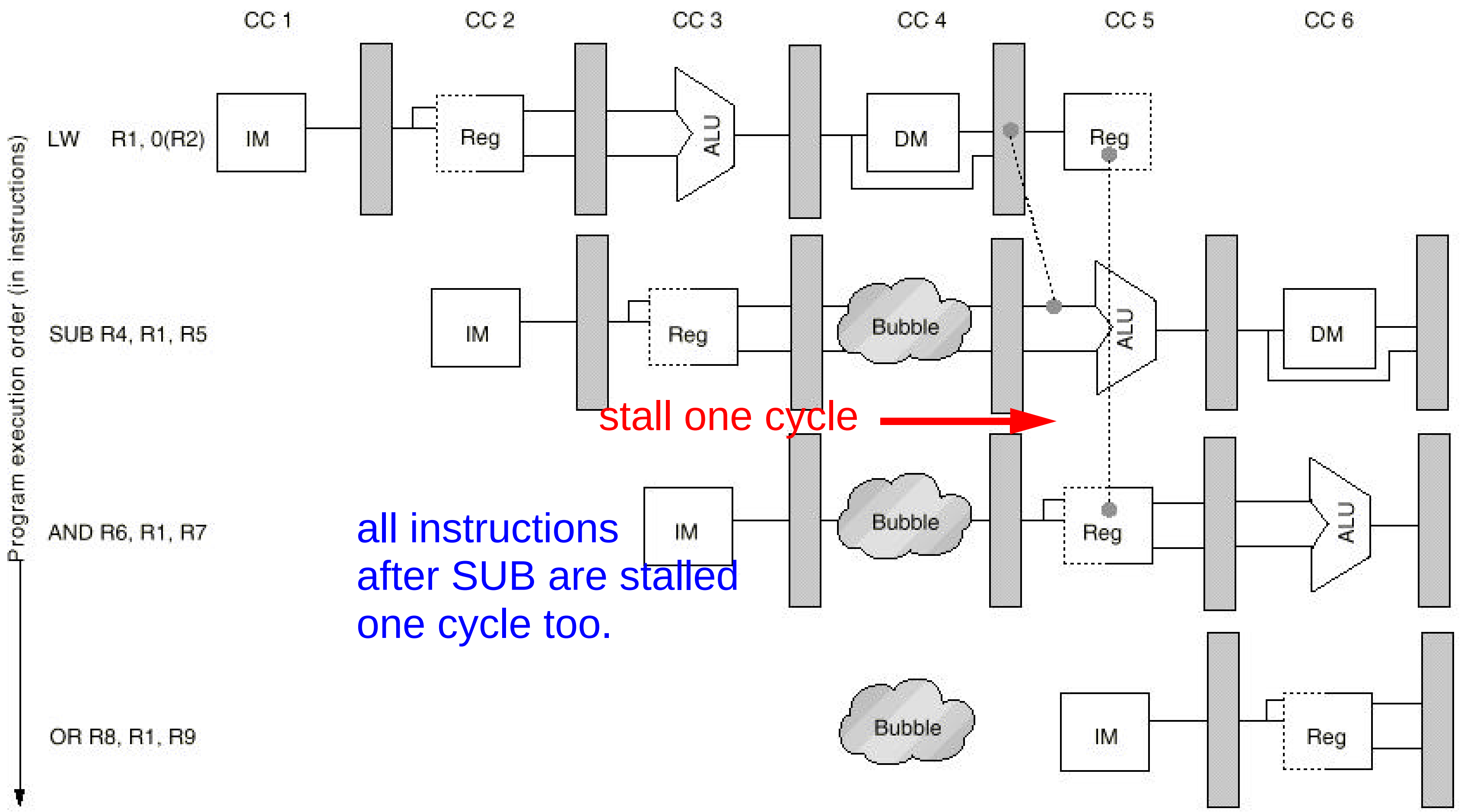
Load data hazard cannot be eliminated by forwarding alone.

Need a hardware, called pipeline interlock, to preserve the right execution order.



memory data in LMD too late for SUB

The only solution is
to stall SUB by one cycle.



all instructions after SUB are stalled one cycle too.

Pipeline Scheduling

The compiler rearranges the code sequence to avoid the pipeline stalls.
Assume there are instructions available for this rearrangement.

Example.

If A=B+C is followed by D=E-F, then the following scheduled code can avoid stall.

```
LW  Rb, B
LW  Rc, C
LW  Re, E  ; swapped with next instruction to avoid stall
ADDRa, Rb, Rc
LW  Rf, F
SW  A, Ra  ; store/load interchanged to avoid stall in SUB
SUBRd, Re, Rf
SW  D, Rd
```

Both load interlocks (LW Rc, C/ADD Ra, Rb, Rc and LW Rf, F/SUB Rd, Re, Rf) are eliminated. The ADD result is forwarded to SMDR for the EX stage of SW.

Control Hazards

Control hazard are problems caused by executing branching/jump instructions on a pipeline machine.

A branch that changes the PC to its target address is called a taken branch.

What will be the result of R2 after the execution of the following instruction sequence with the pipeline described in page 136?

```
2000    ADDI R2, R0, #5
2004    ADDI R1, R0, #10
2008    BNEZ R1, L2
200C    ADDI R2, R0, #4
2010    SUBI R3, R0, #5
2014    OR R5, R6, R7
2018    L2: ADD R4, R1,R0
```

Control Hazard

They can also cause a greater performance loss than data hazards

Example: If instr. i is a taken branch, the PC is not changed until the end of MEM stage. If we stall instr. i+1 after detecting that instr. i is a control instruction, this implies a stall of **three** cycles.

Branch instr.	IF	ID	EX	MEM	WB					
Branch successor		IF	stall	stall	IF	ID	EX	MEM	WB	
Branch successor+1						IF	ID	EX	MEM	WB
Branch successor+2							IF	ID	EX	MEM
Branch successor+3								IF	ID	EX
Branch successor+4									IF	ID
Branch successor+5										IF

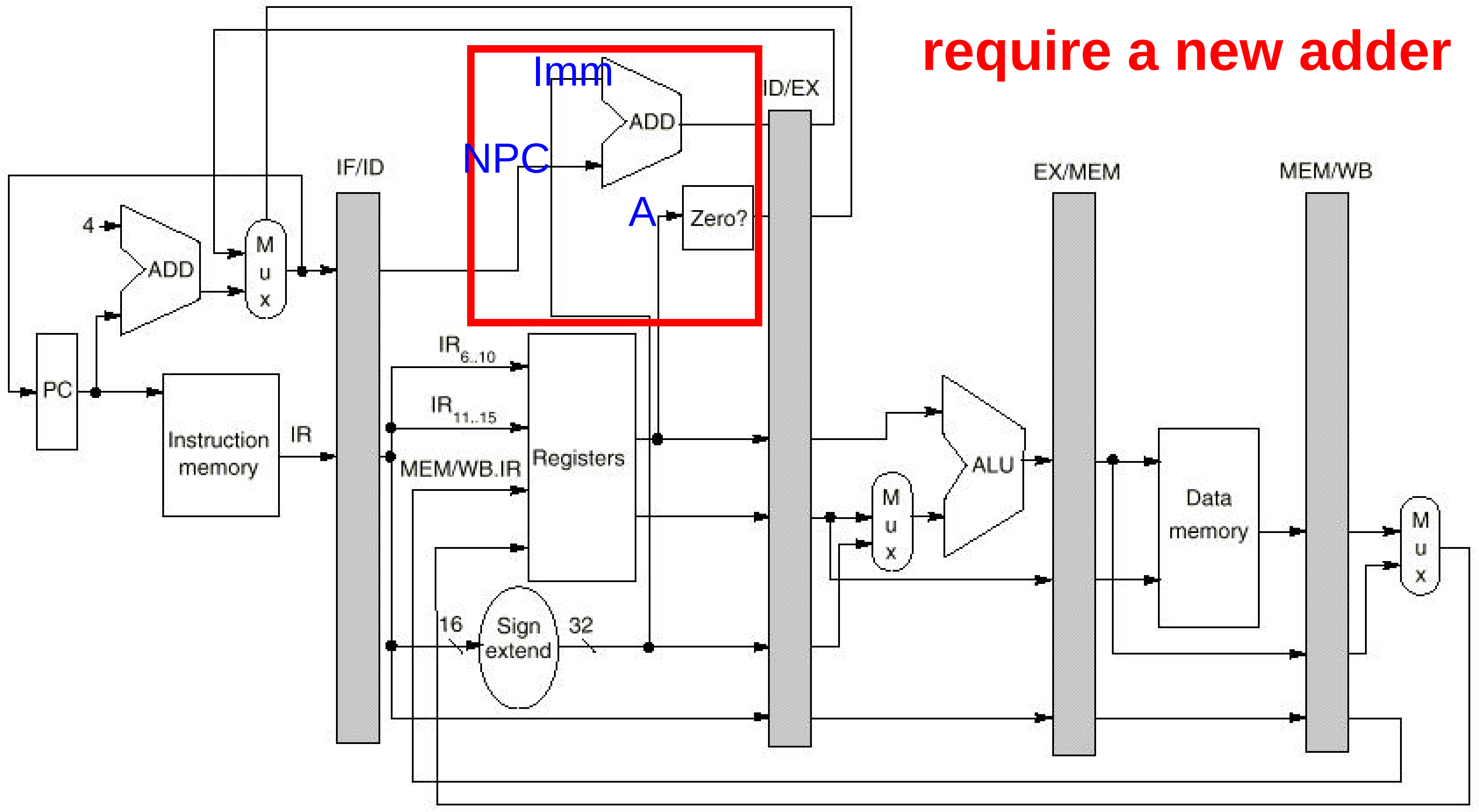
Assume 14%of instr. are control instr. and an ideal CPI of 1$\rightarrow$

$$\frac{\textbf{Speedup}_{real}}{\textbf{Speedup}_{ideal}} = \frac{\frac{1 \times \textbf{PipelineDepth}}{1 + 0.14 \times 3}}{\frac{1 \times \textbf{PipelineDepth}}{1}} = 0.70$$, a significant loss.

Reduce Two Stalled Cycles in a Branch

- Find out whether the branch is taken or not (zero test), earlier in the pipeline.
- Compute the branch taken address earlier. Move from EX stage to ID stage.

require a new adder



Frequency of Control Instructions

- For SPEC subset integer benchmarks, on average, 13% forward condition branches, 3% backward conditional branches, and 4% unconditional branches.
- For SPEC subset FP benchmarks, on average, 7%, forward condition branches, 2% backward conditional branches, and 1% unconditional branches. FP programs have fewer branches.

Reducing Pipeline Branch Penalties

1. Predict-not-taken scheme.

Untaken branch	IF	ID	EX	MEM	WB				
instr. i+1		IF	ID	EX	MEM	WB			
instr. i+2			IF	ID	EX	MEM	WB		
instr. i+3				IF	ID	EX	MEM	WB	
instr. i+4					IF	ID	EX	MEM	WB
Taken branch	IF	ID	EX	MEM	WB				
instr. i+1		IF	idle	idle	idle	idle			
branch target			IF	ID	EX	MEM	WB		
branch target+1				IF	ID	EX	MEM	WB	
branch target+2					IF	ID	EX	MEM	WB

2. Predict-taken scheme. In DLX, it is not useful since we need to generate target address at ID stage. Work for machines which set condition code and generate target address earlier.
3. Delay-branch-scheduling scheme.

Predict-not-taken

- If the branch is untaken $\rightarrow$ no stall.
- If the branch is taken $\rightarrow$ one clock-cycle stall; stop the pipeline and restart the fetch.

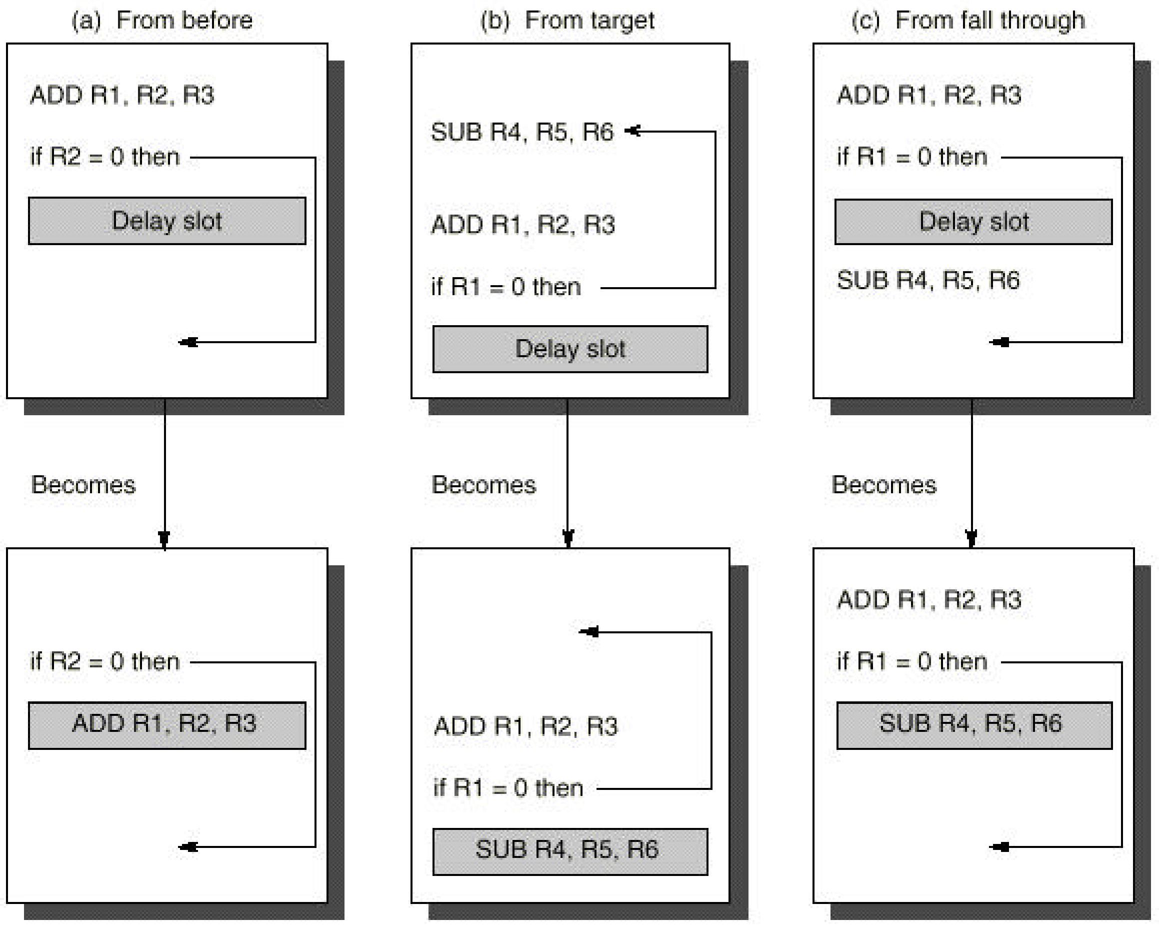
Predict-taken

- Apply to situations where the target address is known before the branch outcome.
- For DLX pipeline, the target address and branch outcome are known at the same stage. Therefore, there is no advantage using this scheme.
- There is always one clock-cycle stall, if predict-taken scheme is used in DLX pipeline.

Delay branch slots: The sequential instructions between branch instruction and branch target instruction are called in delay branch slots. They will be executed no matter what. Therefore better no affect the computation if branch is taken.

Delayed-branch-scheduling

Rearrange the code sequence just like pipeline-scheduling for data hazards.



Criteria for Delay-branch-scheduling

- 50% of branch delays are filled with instructions "*from before branch*".

Scheduling strategy	Requirements	Improves performance when?
From before branch	Branch must not depend on the results of resched-uled instructions	Always
From targe	Must be OK to execute rescheduled instruction if branch is not taken. May need to duplicate instruc-tion	When branch is taken. May enlarge program if instructions are dupli-cated
From fall through	Must be OK to execute instructions if branch is taken	When branch is not taken.

Reducing Pipeline Control Hazard

Original Code

I1	ADDI	R1, R0, #1
I2	LW	R2, 1500(R0)
I3	LW	R7, 2500(R0)
I4	ADDI	R3, R0,#200
I5 L1:	SLLI	R5, R1, #2
I6	LW	R6,5000(R5)
I7	ADD	R6,R6,R2
I8	ADD	R6,R6,R7
I9	SW	0(R5), R6
I10	ADDI	R1,R1,#1
I11	SLE	R4,R1, R3
I12	BNE	R4, L1
I13 L2:	SW	2000(R0), R1

Delay Branch scheduling From “before”

I1	ADDI	R1, R0, #1
I2	LW	R2, 1500(R0)
I3	LW	R7, 2500(R0)
I4	ADDI	R3, R0,#200
I13 L2:	SW	2000(R0), R1

Delay Branch Scheduling From “targe”

I1	ADDI	R1, R0, #1
I2	LW	R2, 1500(R0)
I3	LW	R7, 2500(R0)
I4	ADDI	R3, R0,#200
I14 L2:	SW	2000(R0), R1

Reducing Pipeline Control Hazard

Original Code

I1	ADDI	R1, R0, #1
I2	LW	R2, 1500(R0)
I3	LW	R7, 2500(R0)
I4	ADDI	R3, R0,#200
I5 L1:	SLLI	R5, R1, #2
I6	LW	R6,5000(R5)
I7	ADD	R6,R6,R2
I8	ADD	R6,R6,R7
I9	SW	0(R5), R6
I10	ADDI	R1,R1,#1
I11	SLE	R4,R1, R3
I12	BNE	R4, L1
I13 L2:	SW	2000(R0), R1

Delay Branch scheduling From “before”

I1	ADDI	R1, R0, #1
I2	LW	R2, 1500(R0)
I3	LW	R7, 2500(R0)
I4	ADDI	R3, R0,#200
I5 L1:	SLLI	R5, R1, #2
I6	LW	R6,5000(R5)
I7	ADD	R6,R6,R2
I8	ADD	R6,R6,R7
I9	ADDI	R1,R1,#1
I10	SLE	R4,R1, R3
I11	BNE	R4, L1
I12	SW	0(R5), R6
I13 L2:	SW	2000(R0), R1

Delay Branch Scheduling From “targe”

I1	ADDI	R1, R0, #1
I2	LW	R2, 1500(R0)
I3	LW	R7, 2500(R0)
I4	ADDI	R3, R0,#200
I5 L1:	SLLI	R5, R1, #2
I6	LW	R6,5000(R5)
I7	ADD	R6,R6,R2
I8	ADD	R6,R6,R7
I9	SW	0(R5), R6
I10	ADDI	R1,R1,#1
I11	SLE	R4,R1, R3
I12	BNE	R4, L1+4
I13	SLLI	R5, R1, #2
I14 L2:	SW	2000(R0), R1

Type of Exceptions

Exceptions are harder to handle in pipeline machines due to overlapping of instructions. It may be difficult to judge which one should be allowed to finish or what pipeline stage information to be saved.

- Page fault
- Integer arithmetic overflow or underflow
- FP arithmetic anomaly.
- Misaligned memory access (if alignment is required).
- Memory-protection violation.
- Undefined instruction.
- Power failure
- Hardware malfunction.
- Invoke OS server (trap)
- Tracing instruction execution.
- Breakpoint
- I/O device request

Dealing with Interrupts

When an interrupt occurs, the pipeline must be safely shut down and the state saved so that the instruction can be restarted in the correct state.

Three basic steps to save the pipeline state safely:

- Force a trap instruction into the pipeline on the next IF.
- Until the trap is taken, turn off all writes for the faulting instruction and for all the instructions that follow in the pipeline.
  $\longrightarrow$ This prevents any state changes from instructions that will not be completed before the interrupt is handled.
- The interrupt handling routine saves the PC of the faulting instructions immediately.

Problem: What if the faulting instruction is in a branch delay slot and the branch was taken? Which instructions need to be restarted?

Solution: We need to save addresses of the instructions in the branch delay slots and the branch target. (Is that right?)

- Note that the addresses of the instructions in the branch delay slots and the branch target are not sequential $\longrightarrow$ can not just save one address.

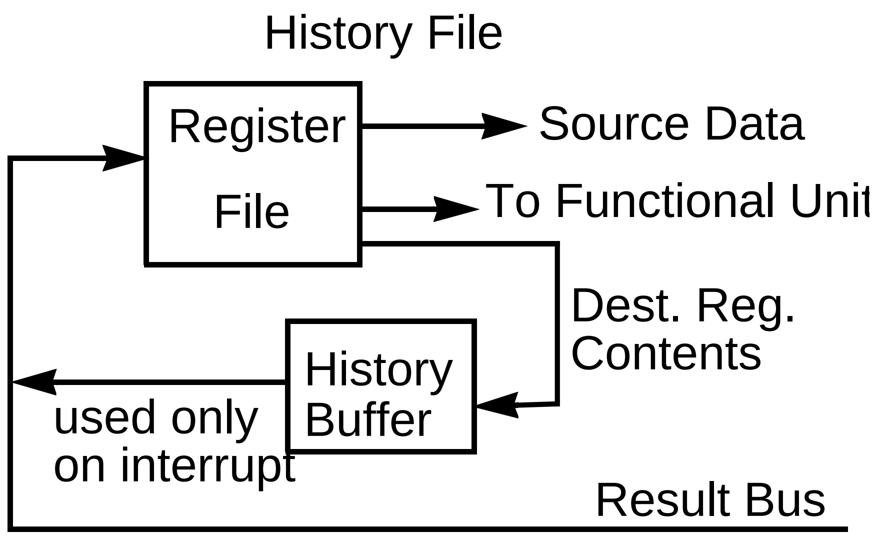
Precise Interrupt

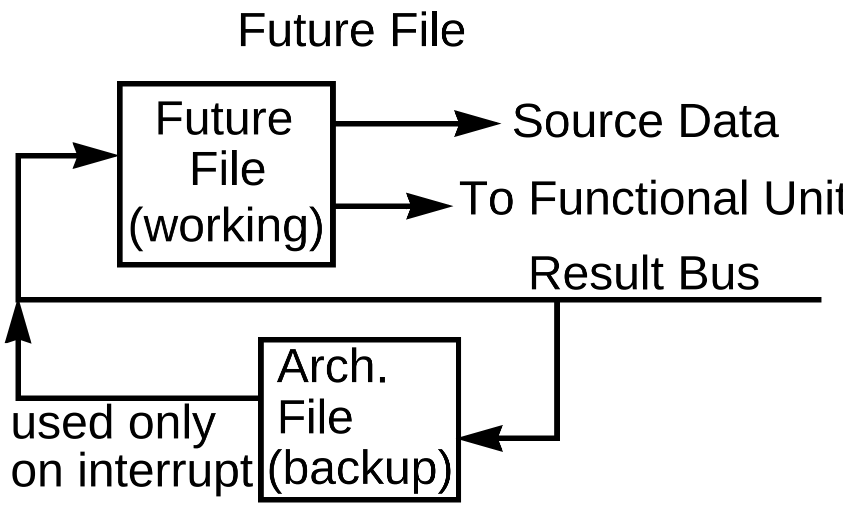
If the pipeline can be stopped so that the instructions just before the faulting instruction are completed and those after it can be restarted from scratch, the pipeline is said to have **precise interrupts**.

Problems: On some machines, the faulting instruction such as FP exceptions, may writes its result before the interrupt is handled.

Supporting precise interrupts is a requirement in many systems.

Solutions: Save old (new) register contents in additional register files to be used for recovering the state after interrupt.





Multiple Interrupts

LW	IF	ID	EX	MEM(1)	WB	
ADD		IF	ID	EX(2)	MEM	WB

(1): A data page fault interrupt

(2): An Arithmetic interrupt They occur at the same time.

IF stage: Page fault; misalignment memory access; memory-protection violation

ID stage: Undefined or illegal opcode

EX stage: Arithmetic exception

MEM stage: Page fault on data accessl; misalignment memory access; memory-protection violation.

WB stage: none.

How to handle this case?

- First, deal with the data page fault.
- Then, restart the execution.

Interrupt may occur out of order

LW	IF	ID	EX	MEM(1)	WB	
ADD		IF(2)	ID	EX	MEM	WB

(1): A data page fault interrupt

(2): An instruction page fault interrupt.

(2) occurs earlier than (1), even though (2) is a later instruction.

How to solve this problem?

Approach 1: Complete precise

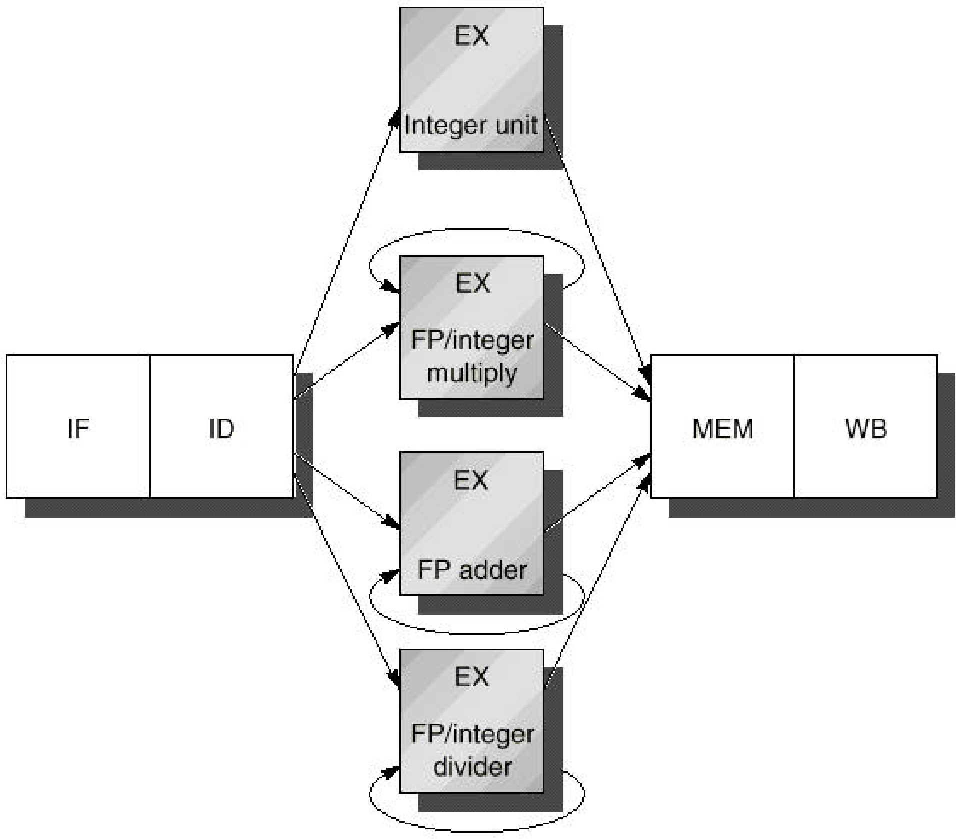
- Each instruction carries a status vector and each interrupt is posted in the status vector.
- The interrupt status vector is checked when an instruction enters WB stage.
- If any interrupts are posted, they are handled in the order
  i.e., the interrupt corresponding to the earliest pipe stage is handled first.
- In DLX machine, *no state* is changed until WB. Therefore DLX has precise interrupts if this approach is used.

Approach 2: handle an interrupt as soon as it appears.

Handle Multicycle Operations

FP operations, integer multiply, and integer divide take multicycle in execution.

New DLX Pipeline allows EX cycle to be repeated as many times as needed and use multiple FP functional units.



Latency and Initiation Interval of Functional Unit

Latency: the number of intervening cycles between an instruction that produces a result and an instruction that uses the results.

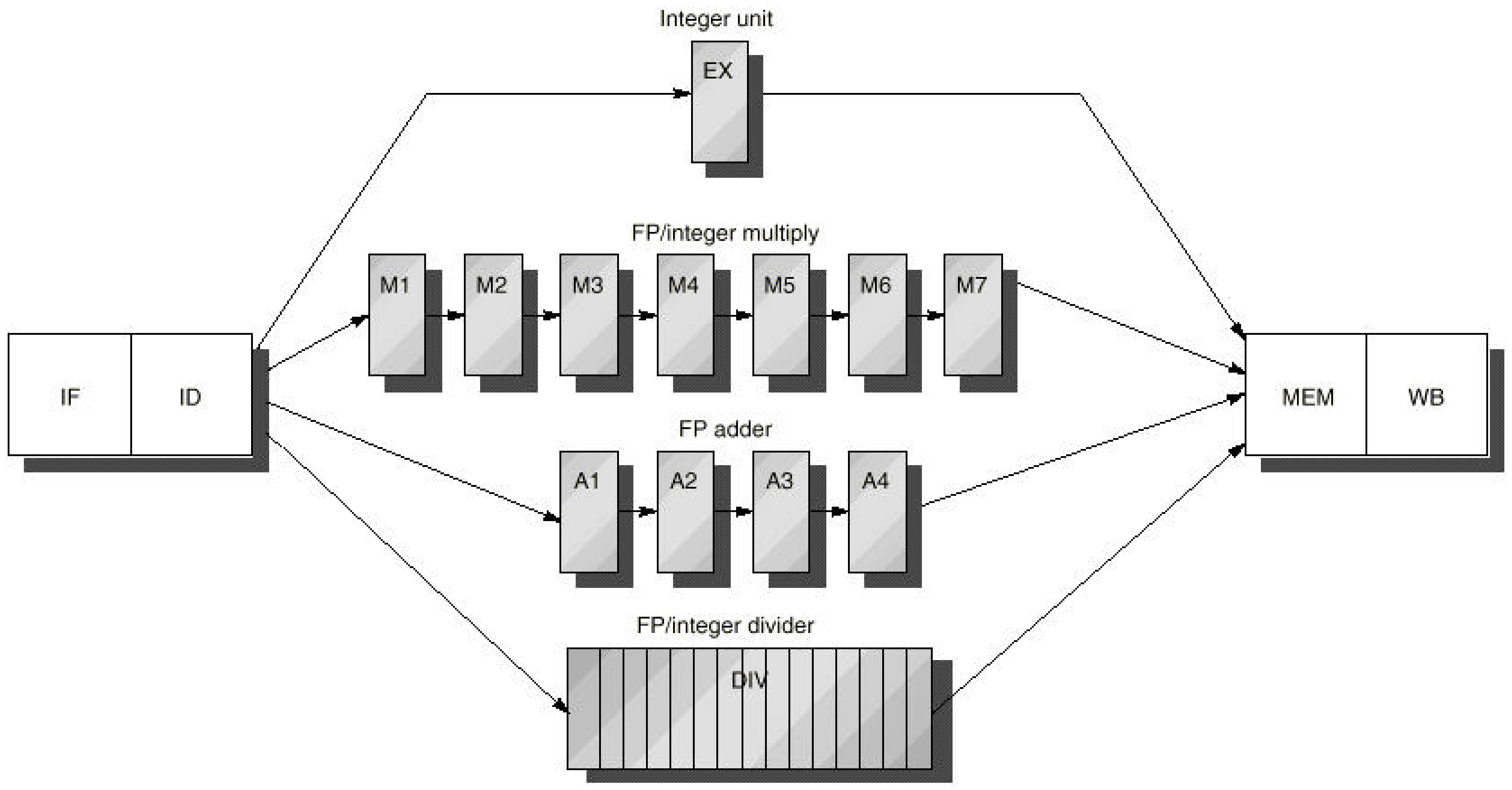
Initiation interval: the number of cycles that must elapse between issuing two operations of a given type.

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply	6	1
FP divide(also integer divide and FP sqrt)	24	24

Latency = the number of pipeline stages of the function unit - 1.

Note that the instructions that follow the current instruction will be issued at least one cycle later.Why FP divide has 24 cycles latency?

DLX with pipelined FP Multiplier and FP Adder



Deal with Pipeline Hazards

When we want to issue a new FP instruction, we take the following steps:

- Check for structural hazard—Wait until the required functional unit is not busy
- Check for a RAW data hazard—Wait until the source registers are not listed as destinations by any of the active EX stages in the functional units.
- Check for forwarding—test if the destination register of an instruction in MEM or WB is one of the source registers of the FP instruction.

In additional to the above hazard prevention steps, we must

- handle the situation where both FP loads and FP operations reach WB simultaneously and compete for FP register file for write.
  $\rightarrow$ allow a single instruction to enter MEM stage and stall others;
  give high priority to the instruction with longest latency.
  (since it most likely to cause bottleneck)
- The above scheme stalls instructions after ID stage (different than the integer pipeline).
- handle WAR and WAW hazards, which could happen because instructions with the different execution cycles.

WAW Hazards

```
DIVF    F0, F2, F4  ; FP divide takes 24 clock cycles
SUBF    F0, F8, F10; FP sub takes less than 4 clock cycles
```

How could this code sequence happen?

How to resolve this WAW hazard?

- Delay the issues of SUBF until DIVF enter MEM.
- Stamp out the result of DIVF by detecting the WAW hazard.

Is there a WAW hazard in the following code sequence?

```
DIVF    F0, F2, F4
MULTF   F5, F0, F1
SUBF    F0, F8, F10
```

RAW Hazard

instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
LD F4,0(R2)	IF	ID	EX	MEM	WB											
MULTD F0, F4, F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB			
ADDD F2, F0, F8			IF	ID	stall	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM
SD F2, 0(R2)				IF	ID	stall	stall	stall	stall	stall	stall	stall	stall	stall	stall	MEM

Structural Hazard

instruction	1	2	3	4	5	6	7	8	9	10	11
MULTD F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
…		IF	ID	EX	MEM	WB					
…			IF	ID	EX	MEM	WB				
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
…					IF	ID	EX	MEM	WB		
…						IF	ID	EX	MEM	WB	
LD F8, 0(R2)							IF	ID	EX	MEM	WB

- At clock cycle 11, all three instructions try to write to the regiester file. If there is only one port, this is a structural hazard.

Out Of Order Completion

```
DIVF    F0, F2, F4
ADDF    F10, F10, F8
SUBF    F12, F12, F14
```

ADDF and SUBF expect to complete before DIVF completes

If

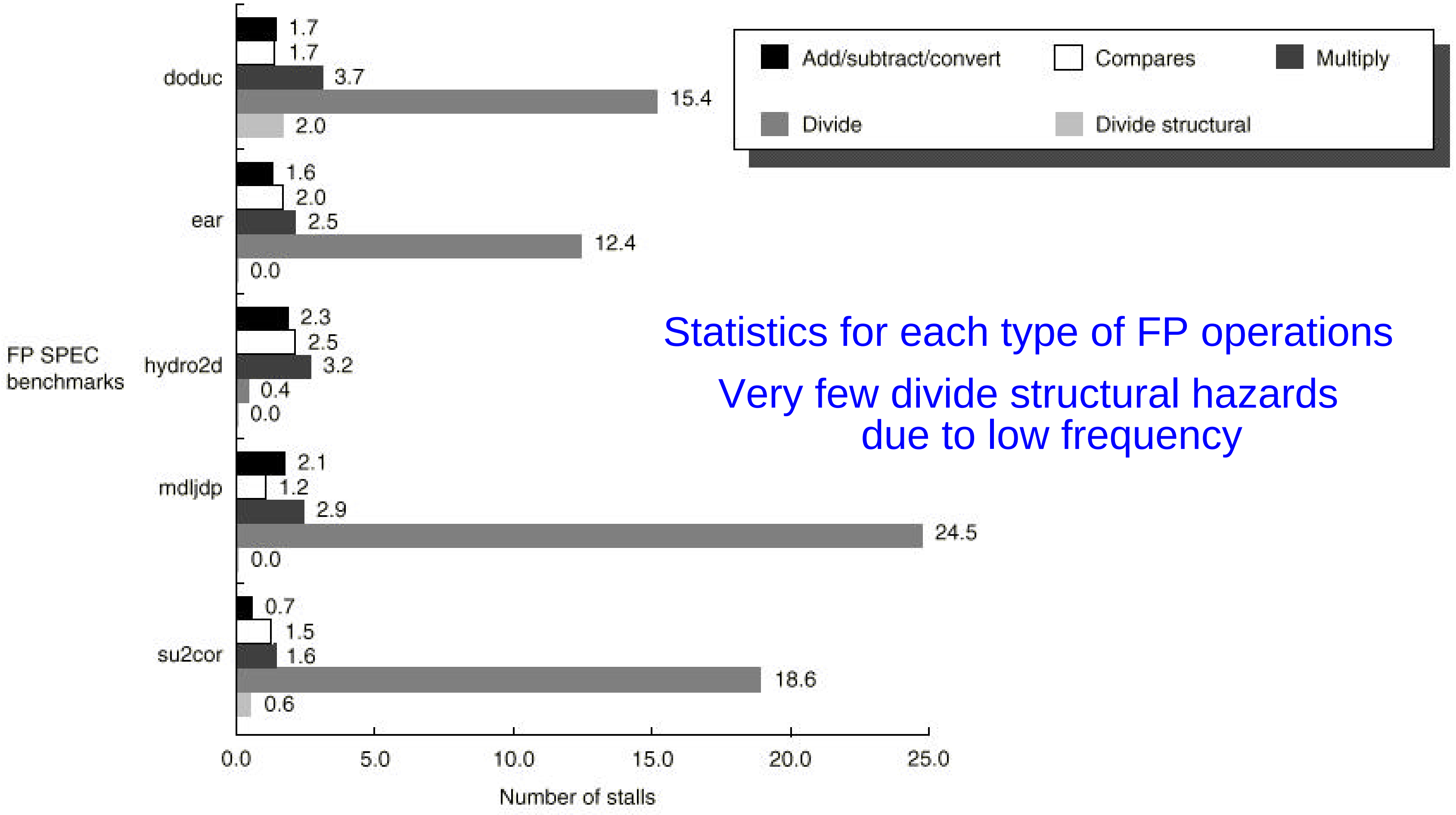
- DIVF causes a FP-arithmetic interrupt,
- ADDF has already completed and destroy the value of F10,
- SUB has not completed,

then we can not restore the state to what is was before the DIVF;
$\rightarrow$ this is an imprecise interrupt.

To have a precise interrupt:

- use history file as that used in CYBER 180/900, VAXes.
- use future file.
- keep enough information for the trap-handling routine to restart.
- A hybrid scheme—allow the instr. issue to continue only if it is certain that all instructions before the **issuing** instr. will complete without causing an interrupt.

Performance of a DLX FP Pipeline Page 190

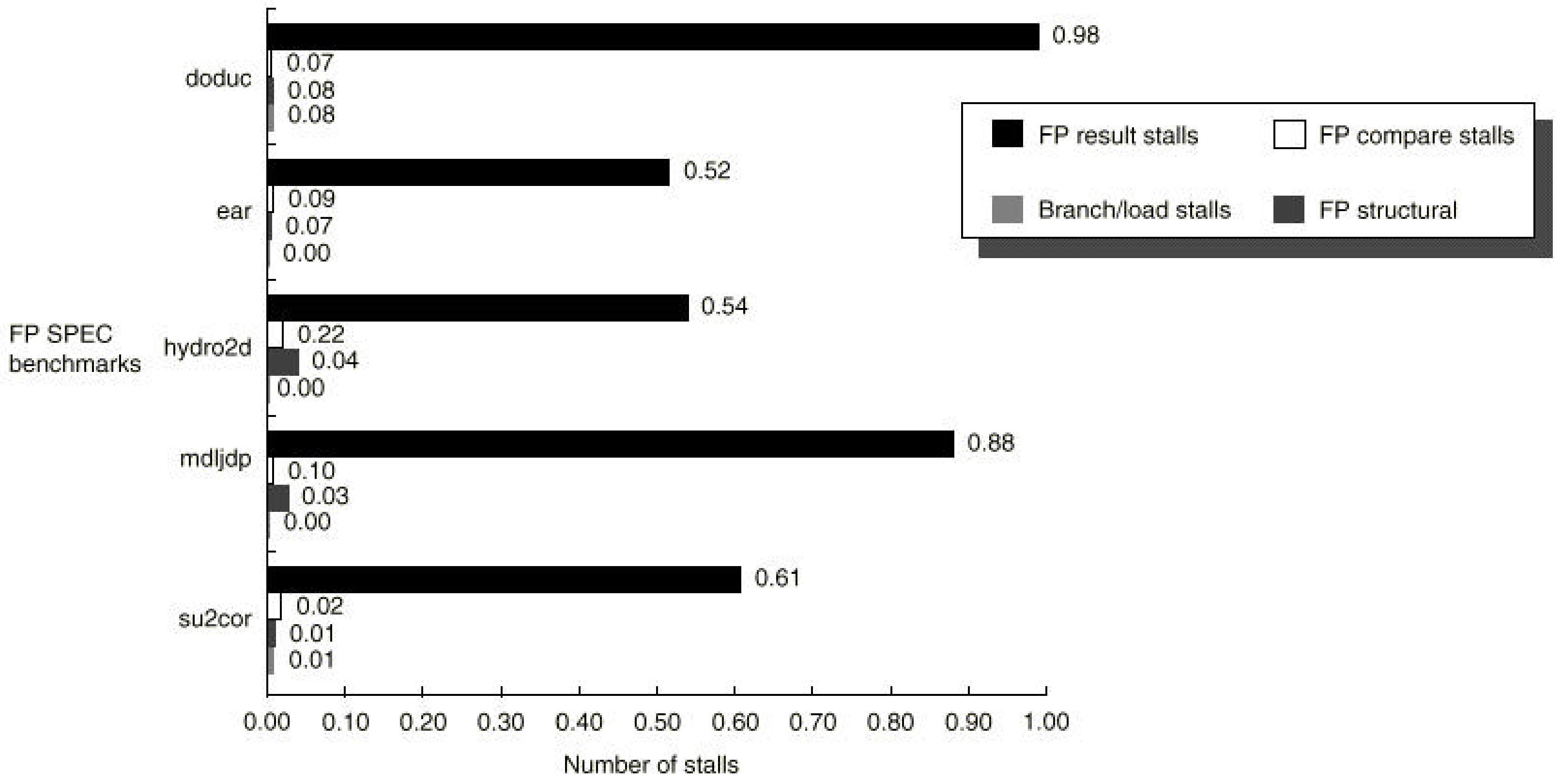


Statistics for each type of FP operations

Very few divide structural hazards due to low frequency

# of stall cycles for each types of FP operations. It is about 46% to 59% of the latency of the functional units. It is generated by running 5 FP SPEC benchmarks.

# of Stalls per Instruction



The compiler tries to schedule both load and FP delays before it schedules branch delays.Why?

Exercise #6

Given the following DLX code,

```
      ADDI   R1, R0, #1        ;; keep the value of i in register R1
      LW     R2, 1500(R0)      ;; keep the value of C in R2
      LW     R7, 2500(R0)      ;; keep the value of D in R7
      ADDI   R3, R0, #200      ;; keep the loop count, 200, in R3
L1:   SLLI   R5, R1, #2        ;; multiply i by 4
      LW     R6, 5000(R5)      ;; calculate address of B[i]
      ADD    R6, R6, R2        ;; B[i]+C
      ADD    R6, R6, R7        ;; B[i]+C+D
      SW     0(R5), R6         ;; A[i]=B[i]+C+D
      ADDI   R1, R1, #1        ;; i++
      SLE    R4, R1, R3
      BNEZ   R4, L1
L2:   SW     2000(R0), R1      ;; save final value of i to memory
```

a) Identify the data hazards that can be solved by the forwarding techniques.

b) Are there stalls due to data hazard even though the forwarding techniques is used? How would you improve that?

c) Are there control hazards? How would you resolve that?

d) How many clock cycles are needed for the execution of the above code if the pipeline in Figure 3.22 with the forwarding hardware?
(Note that the loop count is 200.)

e) Show the improved code after the pipeline-scheduling technique is applied to avoid the stalls caused by the pipeline hazards?

f) How many clock cycles are needed for the execution of your improved code using the pipeline mentioned in part d)?

Solution to Exercise #6

a) ADD R6,R6, R7 has a source operand R6 and is waiting for the result of ADD R6, R6, R2.
SW 0(R5), R6 has a source operand R6 and is waiting for the result of ADD R6,R6, R7.
SLE R4,R1,R3 has a source operand R1 and is waiting for the result of ADDI R1,R1,#1.
BNEZ R4,L1 has a source operand R4 and is waiting for the result of SLE R4,R1,R3.

b) As it can be seen in the pipeline execution pattern table above, the ADD R6,R6, R7 will have a stall cycle at Cycle 9 to wait for the data from memory 5000(R5). This can not be eliminated by the forwarding technique. The forwarding technique did, however, shorten the stall from two cycles to one. The best way to improve this situation is to do pipeline scheduling by moving an independent instruction, such as ADDI R1,R1,R3, to fill this load delay slot.

c) Yes, there is control hazard. The instruction SLLI after the BNEZ will have to be stalled for at least one cycle. We solve this problem by pipeline scheduling. We can either move an independent instruction from before BNEZ such as SW o(R5), R6 to fill the branch delay slot, or we can move an independent instruction from after BNEZ such as SLLI. Note that in latter case, we need to modify BNEZ R4, L1 to BNEZ R4, L1+4 and add one additional instruction SLLI R5,R1,#2 to the program. Compared with the two approaches, the latter is one instruction longer and take one more cycle to finish the program (the execution of the last SLLI is a waste).

d) It takes 4 cycles before the loop, takes 10 cycles including 2 stall cycles in the loop, and takes 5 cycles to finish the execution of SW 2000(R0), R1. In total, it take 4+10*200+5=2009 cycles.

Homework #6

Problem. Pipeline hazards and Pipeline Scheduling.

Given the following DLX code that computes **Y**=**X**-a***Y** where **X** and **Y** are integer vectors of 100 elements.

```
        ADDI    R1, R0, #1       ;; keep the value of i in register R1
        LW      R2, 1500(R0)     ;; keep the value of a in R2
        ADDI    R4, R0, #100     ;; keep the loop count, 100, in R4
L1:     SLLI    R5, R1, #2       ;; multiply i by 4
        LW      R6, 5000(R5)     ;; load X[i] with its address = 5000+R5
        LW      R7, 6000(R5)     ;; load Y[i] with its address = 6000+R5
        MULT    R7, R2, R7       ;; b*Y[i]
        SUB     R7, R6, R7       ;; X[i]-b*Y[i]
        SW      6000(R5), R7     ;; Y[i]=X[i]-b*Y[i]
        ADDI    R1, R1, #1       ;; i++
        SLE     R8, R1, R3
        BNEZ    R8, L1
L2:     SW      2000(R0), R1     ;; save final value of i to memory
```

a) Is there a pipeline hazard between MULT R7, R3, R7 and SUB R7, R6, R7? If there is a pipeline hazard, what is its type and how to solve it?

Homework #6

b) Is there a pipeline hazard between SUB R7, R6, R7 and SW 6000(R5), R7? If there is a pipeline hazard, what is its name and how to solve it?

c) The ADDI R1, R1, #1 can be scheduled to be executed after LW R6, 5000(R5) to fill the delay slot and avoid one stall cycle. But there is another instruction that is also a good candidate to fill that delay slot (probably a better candidate.) Please identify the instruction.

d) Show the improved code after pipeline-scheduling is applied to avoid all possible pipeline hazards.

Problem 2. Assume that same pipeline on Page 190. For the following code

```
LW     F6, 5000(R5)     ;; load X[i] with its address = 5000+R5
LW     F7, 6000(R5)     ;; load Y[i] with its address = 6000+R5
MULTF  F7, F2, F7       ;; a*Y[i]
SUBF   F7, F6, F7       ;; X[i]-a*Y[i]
SW     6000(R5), F7     ;; Y[i]=X[i]-a*Y[i]
```

a) Show the pipeline stages for the execution of the above code segment. Note that here MULTF is single precision FP multiply and SUBF is a single precision FP subtraction.

b) Idenfy all pipeline hazards.

Solution to Homework #6

Problem 1. Pipeline hazards and Pipeline Scheduling.

Given the following DLX code that computes **Y**=**X**-a***Y** where **X** and **Y** are integer vectors of 100 elements.

```
        ADDI    R1, R0, #1        ;; keep the value of i in register R1
        LW      F2, 1500(R0)      ;; keep the value of a in F2
        ADDI    R3, R0, #100      ;; keep the loop count, 100, in R3
L1:     SLLI    R5, R1, #2        ;; multiply i by 4
        LW      F6, 5000(R5)      ;; load X[i] with its address = 5000+R5
        LW      F7, 6000(R5)      ;; load Y[i] with its address = 6000+R5
        MULTF   F7, F2, F7        ;; a*Y[i]
        SUBF    F7, F6, F7        ;; X[i]-a*Y[i]
        SW      6000(R5), F7      ;; Y[i]=X[i]-a*Y[i]
        ADDI    R1, R1, #1        ;; i++
        SLE     R8, R1, R3
        BNEZ    R8, L1
L2:     SW      2000(R0), R1      ;; save final value of i to memory
```

a) (5pts) Is there a pipeline hazard between MULTF F7, F2, F7 and SUBF F7, F6, F7? If there is a pipeline hazard, what is its type and how to solve it?

Ans: There is a RAW hazard since SUBF is waiting MULTF to generate its second source operand. It will take 7 cycles for the floating point multiplier to generate the value. The pipeline need to be stalled for 6 cycles. The speed up the execution the floating pointer multiplier will **forward** the result to the floating point adder that executes the SUBF.

There is also a WAW hazard since SUBF also generates the result that overwrite F7 value. The solution to the RAW hazard mentioned above also solves this WAW hazard.

b) (5pts) Is there a pipeline hazard between SUBF F7, F6, F7 and SW 6000(R5), F7? If there is a pipeline hazard, what is its name and how to solve it?

Ans: There is a RAW hazard since SW is waiting SUBF to generate its second source operand. It can be solved by stalling the execution of SW for 3 cycles.

c) (5pts) The ADDI R1, R1, #1 can be scheduled to be executed after LW F7, 6000(R5) to fill the delay slot and avoid one stall cycle. But there is another instruction that is also a good candidate to fill that delay slot (probably a better candidate.) Please identify the instruction.

Ans: We can schedule LW F6, 5000(R5) to be executed after LW F7.

d) (15 pts) Show the improved code after pipeline-scheduling is applied to avoid all possible pipeline hazards. Here we assume that there are 6 cycle latency on the multiplication. Take that the into your consideration when scheduling the code.

Ans: It turns out that without loop unrolling we can not avoid the stall of the pipeline. Here is the best we can do using pipeline scheduling.

```
        ADDI    R1, R0, #1          ;; keep the value of i in register R1
        LW      F2, 1500(R0)        ;; keep the value of a in F2
        ADDI    R3, R0, #100        ;; keep the loop count, 100, in R3
L1:     SLLI    R5, R1, #2          ;; multiply i by 4
        LW      F7, 6000(R5)        ;; load Y[i] with its address = 6000+R5
        LW      F6, 5000(R5)        ;; load X[i] with its address = 5000+R5
        MULTF F7, F2, F7         ;; a*Y[i]
        ADDI  R1, R1, #1         ;; i++
        SLE   R8, R1, R3
        SUBF  F7, F6, F7         ;; X[i]-a*Y[i]
        BNEZ  R8, L1
        SW    6000(R5), F7       ;; Y[i]=X[i]-a*Y[i]
L2:     SW    2000(R0), R1       ;; save final value of i to memory
```

ADDIR1, R0, #1	F	D	X	M	B																
LWF2, 1500(R0)		F	D	X	M	B															
ADDI R3, R0, #100			F	D	X	M	B														
SLLIR5, R1, #2				F	D	X	M	B													
LWF7, 6000(R5)					F	D	X	M	B												
LWF6, 5000(R5)						F	D	X	M	B											
MULTFF7, F2, F7							F	D	X1	X2	X3	X4	X5	X6	X7	M	B				
ADDIR1, R1, #1								F	D	X	M	B									
SLER8, R1, R3									F	D	X	M	B								
SUBFF7, F6, F7										F	D	S	S	S	S	X1	X2	X3	X4	M	B
BNEZR8, L1											F	S	S	S	S	D	X	M	B		
SW6000(R5), F7																F	D	S	S	X	M

```
ADDI R1, R0, #1;; keep the value of i in register R1
     LW     F2, 1500(R0)    ;; keep the value of a in F2
     ADDI   R3, R0, #100    ;; keep the loop count, 100, in R3
L1:  SLLI   R5, R1, #2      ;; multiply i by 4
     LW     F7, 6000(R5)    ;; load Y[i] with its address = 6000+R5
     LW     F6, 5000(R5)    ;; load X[i] with its address = 5000+R5
     MULTF F7, F2, F7       ;; a*Y[i]
     SUBF   F7, F6, F7      ;; X[i]-a*Y[i]
     SW     6000(R5), F7    ;; Y[i]=X[i]-a*Y[i]
     LW     F9, 6004(R5)    ;; load Y[i] with its address = 6000+R5
     LW     F8, 5004(R5)    ;; load X[i] with its address = 5000+R5
     MULTF F9, F2, F9       ;; a*Y[i]
     SUBF   F9, F8, F9      ;; X[i]-a*Y[i]
     SW     6004(R5), F9    ;; Y[i]=X[i]-a*Y[i]
     ADDI   R1, R1, #2      ;; i++
     SLE    R8, R1, R3
     BNEZ   R8, L1
```

```
ADDI R1, R0, #1;; keep the value of i in register R1
     LW     F2, 1500(R0)    ;; keep the value of a in F2
     ADDI   R3, R0, #100    ;; keep the loop count, 100, in R3
L1:  SLLI   R5, R1, #2      ;; multiply i by 4
     LW     F7, 6000(R5)    ;; load Y[i] with its address = 6000+R5
     LW     F6, 5000(R5)    ;; load X[i] with its address = 5000+R5
     MULTF F7, F2, F7       ;; a*Y[i]
     LW     F9, 6004(R5)    ;; load Y[i] with its address = 6000+R5
     LW     F8, 5004(R5)    ;; load X[i] with its address = 5000+R5
     ADDI   R1, R1, #2      ;; i++
     SLE    R8, R1, R3
     MULTF F9, F2, F9       ;; a*Y[i]
     SUBF   F7, F6, F7      ;; X[i]-a*Y[i]
     SW     6000(R5), F7    ;; Y[i]=X[i]-a*Y[i]
     SUBF   F9, F8, F9      ;; X[i]-a*Y[i]
     SW     6004(R5), F9    ;; Y[i]=X[i]-a*Y[i]
     BNEZ   R8, L1
```

SLLIR5, R1, #2																	F	S	S	D	X

e) How many time we have to unroll the loop to avoid the stall comletely?

Ans:

Homework #6

Problem 1. Pipeline hazards and Pipeline Scheduling.

Given the following DLX code that computes **Y**=**X**-a***Y** where **X** and **Y** are integer vectors of 100 elements.

```
     ADDI   R1, R0, #1       ;; keep the value of i in register R1
     LW     F2, 1500(R0)     ;; keep the value of a in F2
     ADDI   R3, R0, #100     ;; keep the loop count, 100, in R3
L1:  SLLI   R5, R1, #2       ;; multiply i by 4
     LW     F6, 5000(R5)     ;; load X[i] with its address = 5000+R5
     LW     F7, 6000(R5)     ;; load Y[i] with its address = 6000+R5
     MULTF  F7, F2, F7       ;; a*Y[i]
     SUBF   F7, F6, F7       ;; X[i]-a*Y[i]
     SW     6000(R5), F7     ;; Y[i]=X[i]-a*Y[i]
     ADDI   R1, R1, #1       ;; i++
     SLE    R8, R1, R3
     BNEZ   R8, L1
L2:  SW     2000(R0), R1     ;; save final value of i to memory
```

a) (5pts) Is there a pipeline hazard between MULTF F7, F2, F7 and SUBF F7, F6, F7? If there is a pipeline hazard, what is its type and how to solve it?

Homework #6

b) (5pts) Is there a pipeline hazard between SUBF F7, F6, F7 and SW 6000(R5), F7? If there is a pipeline hazard, what is its name and how to solve it?

c) (5pts) The ADDI R1, R1, #1 can be scheduled to be executed after LW F7, 5000(R5) to fill the delay slot and avoid one stall cycle. But there is another instruction that is also a good candidate to fill that delay slot (probably a better candidate.) Please identify the instruction.

d) (15 pts) Show the improved code after pipeline-scheduling is applied to avoid all possible pipeline hazards. Here we assume that there are 6 cycle latency on the multiplication. Take that the into your consideration when scheduling the code.

Homework #7

Problem 3. DLX pipeline with multiple cycle functional units.

Assume we have the same DLX pipeline as that shown in Figure 6.28. The multiplier takes 10 cycles, the FP adder takes 5 cycles, and the divider takes 20 cycles to complete its computation.

a) (10pts) Identify all the pipeline hazards in the following sequence of code.

```
DIVF    F0, F2, F4
DIVF    F6, F8, F10
ADDF    F0, F0, F6
SUBF    F6, F4, F10
```

b) (10pts) For each pipeline hazard, explain how it can be handled.

c) (10pts) After mechanisms are installed to avoid the pipeline hazards, how many clock cycles are needed to execute the above code, including all the pipe stages?

Problem 4. (10 points) Select and view any of the following tapes in CS420/520 library reserve.

David Patterson, “The design & development of SPARC”

Phil Hester, “Superscalar RISC Concepts and the RS6000”

Michael Mahon, “HP precision architecture”

I suggest that you watch a tape in group so that you can discuss, but you can also watch it individually.

a) What topic in the tape impresses you most?

b) Write one paragraph on the new thing that you learn from the speaker.

c) Your opinions/suggestions on the use of these video tapes. (This is a feedback question.)

Solution to Homework #7

1.a) There is a RAW data hazard. It can be solved by forwarding the result of MULT to replace the Rs2 operand for the SUB operation. It can also be solved by pipeline scheduling by moving ADDI and SLE instructions between MULT and SUB.

1.b) There is a RAW data hazard. It can be solved by forwarding the result of SUB to replace the Rd operand for the SW operation. It can also be solved by pipeline scheduling by moving ADDI and SLE instructions between SUB and SW.

1.c) LW R7, 6000(R5).

1.d)

```
1.d) ADDI   R1, R0, #1       ;; keep the value of i in register R1
     LW     R2, 1500(R0)     ;; keep the value of a in R2
     LW     R3, 2500(R0)     ;; keep the value of b in R3
     ADDI   R4, R0, #100     ;; keep the loop count, 100, in R4
L1:  SLLI   R5, R1, #2       ;; multiply i by 4
     LW     R6, 5000(R5)     ;; load X[i] with its address = 5000+R5
     LW     R7, 6000(R5)     ;; load Y[i] with its address = 6000+R5
     MULT   R6, R2, R6       ;; a*X[i]
     MULT   R7, R3, R7       ;; b*Y[i]
     ADDI   R1, R1, #1       ;; i++
     SLE     R8, R1, R3
     SUB     R7, R6, R7       ;; a*X[i]-b*Y[i]
     BNEZ    R8, L1
     SW      6000(R5), R7     ;; Y[i]=a*X[i]-b*Y[i]
L2:     SW      2000(R0), R1     ;; save final value of i to memory
```

2.a) Possible Interrupts

```
        ADDI    R1, R0, #1       IF page fault
        LW      R2, 1500(R0)     MEM page fault
        LW      R3, 2500(R0)     MEM page fault
        ADDI    R4, R0, #100
L1:     SLLI    R5, R1, #2
        LW      R6, 5000(R5)     MEM page fault
        MULT    R6, R2, R6       Arithmetic fault
        LW      R7, 6000(R5)     MEM page fault
        MULT    R7, R3, R7       Arithmetic fault
        SUB     R7, R6, R7       Arithmetic fault
        SW      6000(R5), R7     MEM page fault
        ADDI    R1, R1, #1
        SLE     R8, R1, R3
        BNEZ    R8, L1
L2:     SW      2000(R0), R1     MEM page fault
```

2.b) There are three instances where an LW or SW is followed by an ALU instruction. But the pipeline interlock mechanism will delay the MULTs for one cycle therefore no two interrupts can happen in the same cycle. The ADDI after SW will not overflow.

2.c) If there is a multiple interrupt in the same cycle, the MEM page fault should be served before the EX stage arithmetic fault to be taken care of.

3.a&b) There is a structural hazard between 1st and 2nd DIVF since there is only one FP division unit. The 2nd DIVF has to be stalled.
There is a RAW data hazard between 2nd DIVF and ADDF via F6. ADDF has to be stalled until F6 is ready to be read.
There is a structure hazard between ADDF and SUBF. SUBF has to be stalled.
There is a potential WAW data hazard between 2nd DIVF and SUBF via F6 but it won’t happen due to the stall to avoid the structural hazard.
There is a potential WAR data hazard between ADDF and SUBF via F6 but it won’t happen due to the stall to avoid the structural hazard.
There is a potential WAW data hazard between 1st DIVF and ADDF via F0 but it is avoided due to the stall.
There is also a RAW hazard between 1st DIVF and ADDF via F0.

3.c) 54 cycles are needed.

	1	2	3	4		22	23	24		42	43	44		47	48	49		52	53	54
DIVF F0,F2,F4	IF	ID	EX	EX		EX	MEM	WB												
DIVF F6,F8,F10		IF	ID	S		S	EX	EX		EX	MEM	WB								
ADDF F0, F0, F6			IF	S		S	ID	S		S	EX	EX		EX	MEM	WB				
SUBF F6, F4, F10				S		S	IF	S		S	ID	S		S	EX	EX		EX	MEM	WB