CSE 4201
Computer Architecture

Prof. Mokhtar Aboelaze

Parts of these slides are taken from
Notes by Prof. David Patterson at UCB

Outline

- MIPS and instruction set
- Simple pipeline in MIPS
- Structural and data hazards
- Forwarding
- Branching
- Exception and interrupts
- Multicycle operations

MIPS Instruction set

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

Instruction Set

- Instruction Set Architecture
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- Meaning of each instruction is described by RTL on *architected registers* and memory
- Given technology constraints assemble adequate datapath
  - Architected storage mapped to actual storage
  - Function units to do all the required operations
  - Possible additional storage (eg. MAR, MDR, …)
  - Interconnect to move information among regs and FUs
- Map each instruction to sequence of RTLs
- Collate sequences into symbolic controller state transition diagram (STD)
- Lower symbolic STD to control points
- Implement controller

MIPS Instruction Set

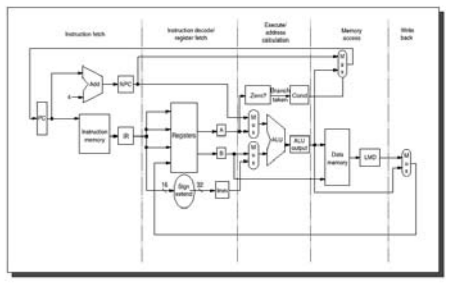
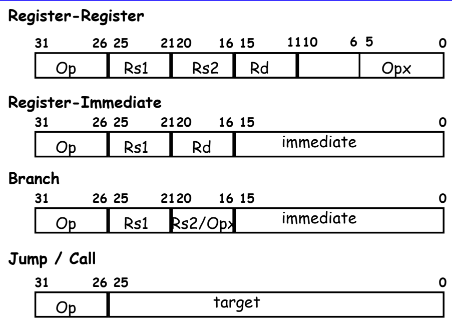
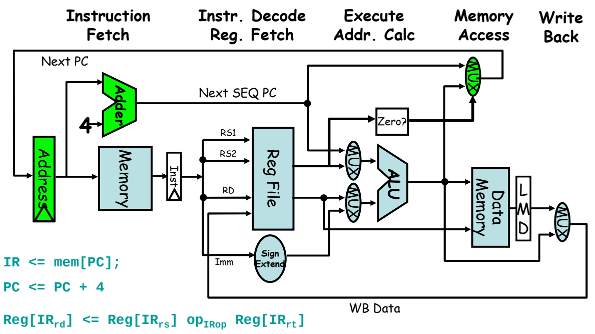


FIGURE 3.1 The implementation of the DLX datapath allows every instruction to be executed in four or five clock cycles.

MIPS Instruction Set

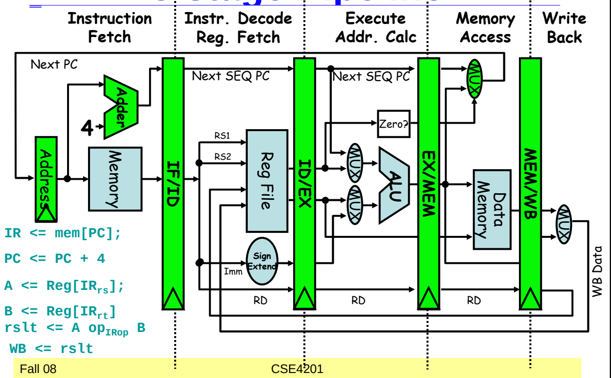


MIPS 5-Stage Pipeline



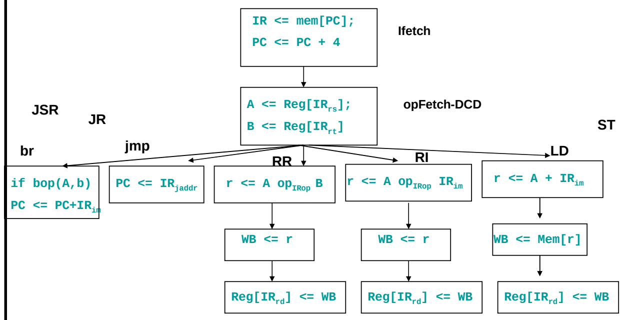
```
IR <= mem[PC];
PC <= PC + 4
Reg[IR_rd] <= Reg[IR_rs] op_IRop Reg[IR_rt]
```

5-stage Pipeline



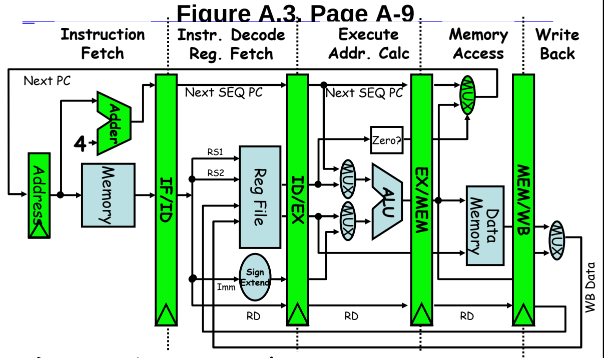
```
IR <= mem[PC];
PC <= PC + 4
A <= Reg[IR_rs];
B <= Reg[IR_rt]
rslt <= A op_IRop B
 WB <= rslt
Reg[IR_rd] <= WB
```

MIPS 5-Stage Pipeline



5 Steps of MIPS Datapath

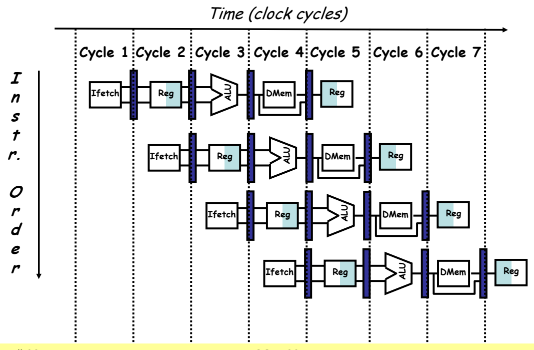
Figure A.3, Page A-9



- **Data stationary control**
  - local decode for each instruction phase / pipeline stage

Visualizing Pipelining

Figure A.2, Page A-8

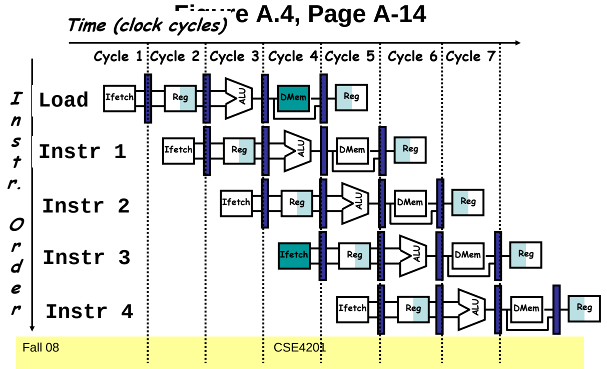


Pipelining is not quite that easy!

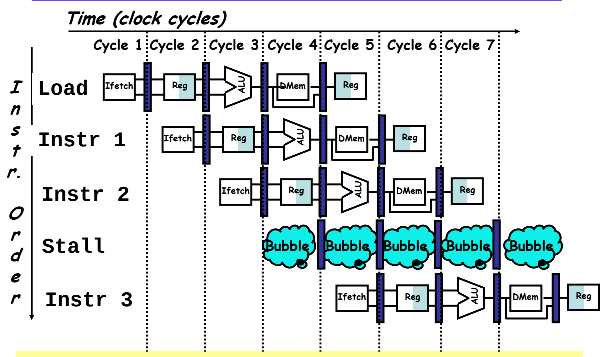
- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

One Memory Port/Structural Hazards

Figure A.4, Page A-14



One Memory Port/Structural Hazards



How do you “bubble” the pipe?

Speed Up Equation for Pipelining

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

For simple RISC pipeline, CPI = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

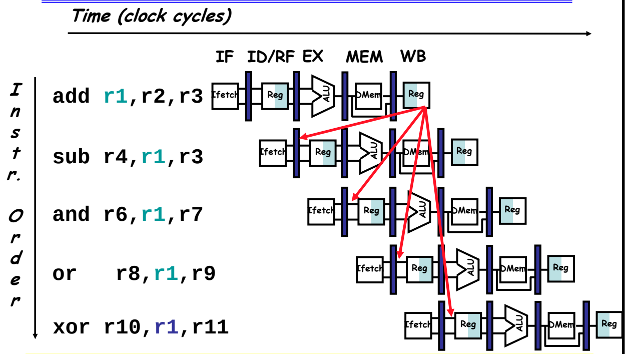
Example: Dual-port vs. Single-port

- Machine A: Dual ported memory (“Harvard Architecture”)
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

  SpeedUp_A = Pipeline Depth/(1 + 0) x ($\text{clock}_{unpipe}/\text{clock}_{pipe}$)
  = Pipeline Depth
  SpeedUp_B = Pipeline Depth/(1 + 0.4 x 1) x (clock_{unpipe}/(clock_{unpipe} / 1.05)
  = (Pipeline Depth/1.4) x 1.05
  = 0.75 x Pipeline Depth
  SpeedUp_A / SpeedUp_B = Pipeline Depth/(0.75 x Pipeline Depth) = 1.33
- Machine A is 1.33 times faster

Data Hazard on R1

Figure A.6, Page A-17



Three Generic Data Hazards

- Read After Write (RAW)
  Instr$_J$ tries to read operand before Instr$_I$ writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

Three Generic Data Hazards

- Write After Read (WAR)
  Instr$_J$ writes operand *before* Instr$_I$ reads it

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "anti-dependence" by compiler writers.
  This results from reuse of the name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

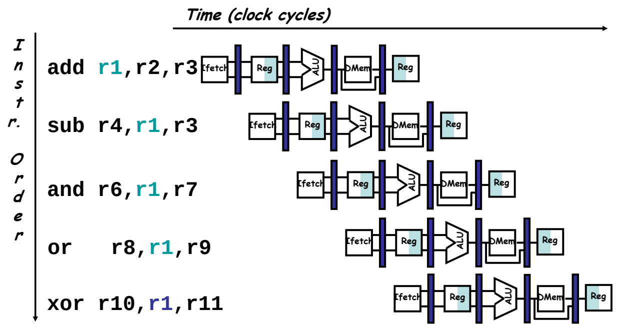
Three Generic Data Hazards

- Write After Write (WAW)
  Instr$_J$ writes operand *before* Instr$_I$ writes it.

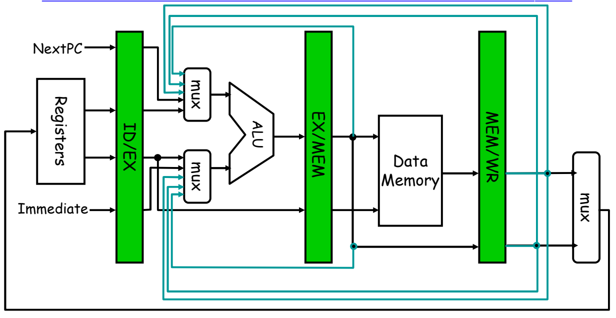
```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- Will see WAR and WAW in more complicated pipes

Forwarding to Avoid Data Hazard

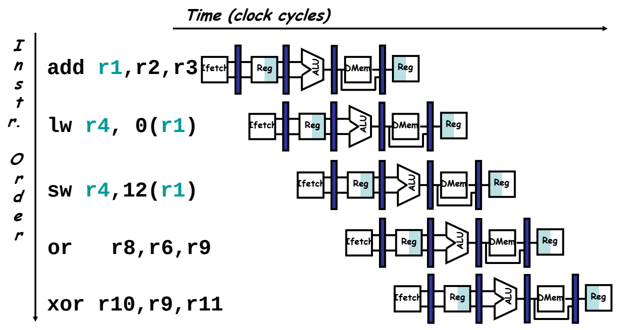


HW Change for Forwarding

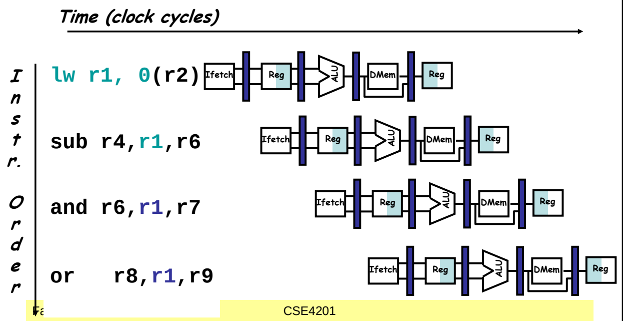


What circuit detects and resolves this hazard?

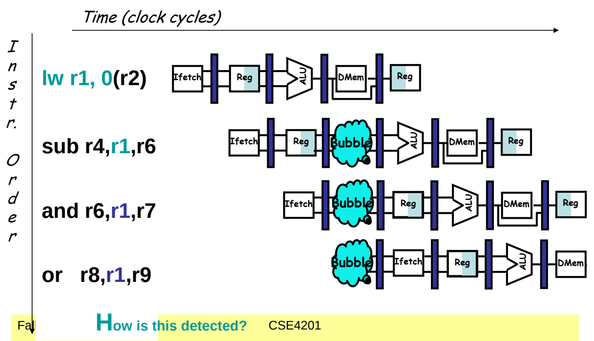
Forwarding to Avoid LW-SW Data Hazard



Data Hazard Even with Forwarding



Data Hazard Even with Forwarding



How is this detected?

Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

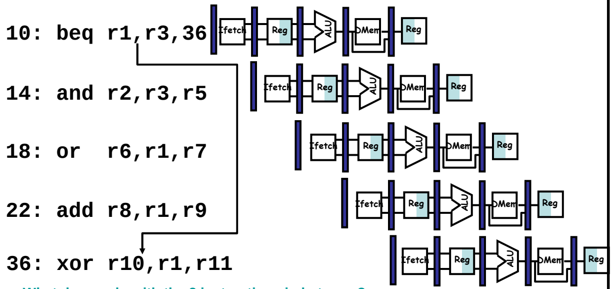
d = e – f;

assuming a, b, c, d ,e, and f in memory.

Slow code:		Fast code:	
LW	Rb,b	LW	Rb,b
LW	Rc,c	LW	Rc,c
ADD	Ra,Rb,Rc	LW	Re,e
SW	a,Ra	ADD	Ra,Rb,Rc
LW	Re,e	LW	Rf,f
LW	Rf,f	SW	a,Ra
SUB	Rd,Re,Rf	SUB	Rd,Re,Rf
SW	d,Rd	SW	d,Rd

Compiler optimizes for performance. Hardware checks for safety.

Control Hazard on Branches Three Stage Stall



What do you do with the 3 instructions in between?

How do you do it?

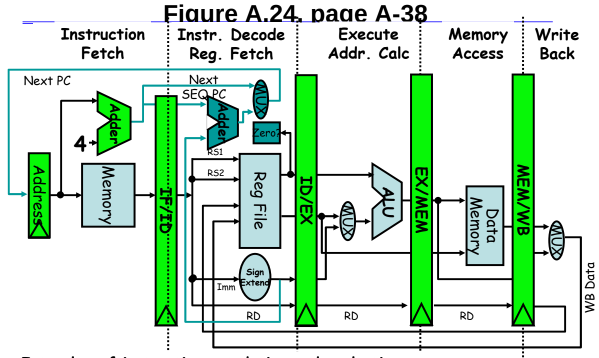
Where is the “commit”?

Branch Stall Impact

- If CPI = 1, 30% branch,
  Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Pipelined MIPS Datapath

Figure A.24, page A-38



• Interplay of instruction set design and cycle time.

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
  - MIPS still incurs 1 cycle branch penalty
  - Other machines: branch target known before outcome

Four Branch Hazard Alternatives

#4: Delayed Branch

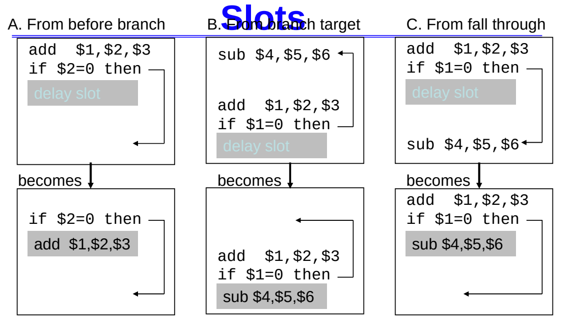
- Define branch to take place AFTER a following instruction

```
branch instruction
 sequential successor1
 sequential successor2
 ........
 sequential successorn
branch target if taken
```

Branch delay of length *n*

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Scheduling Branch Delay Slots



A. From before branch

```
add  $1,$2,$3
if $2=0 then
   delay slot
```

becomes

```
if $2=0 then
   add $1,$2,$3
```

B. From branch target

```
sub $4,$5,$6

add  $1,$2,$3
if $1=0 then
   delay slot
```

becomes

```
add  $1,$2,$3
if $1=0 then
   sub $4,$5,$6
```

C. From fall through

```
add  $1,$2,$3
if $1=0 then
   delay slot

sub $4,$5,$6
```

becomes

```
add  $1,$2,$3
if $1=0 then
   sub $4,$5,$6
```

- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the sub instruction may need to be copied, increasing IC
- In B and C, must be okay to execute sub when branch fails

Delayed Branch

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Assume 4% unconditional branch, 6% conditional branch- untaken, 10% conditional branch-taken

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. unpipelined*	*speedup v. stall*
Stall pipeline	3	1.60	3.1	1.0
Predict taken	1	1.20	4.2	1.33
Predict not taken	1	1.14	4.4	1.40
Delayed branch	0.5	1.10	4.5	1.45

Problems with Pipelining

- Exception: An unusual event happens to an instruction during its execution
  - Examples: divide by zero, undefined opcode
- Interrupt: Hardware signal to switch the processor to a new instruction stream
  - Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)
- Problem: It must appear that the exception or interrupt must appear between 2 instructions (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 can take place
- The interrupt (exception) handler either aborts program or restarts at instruction I_{i+1}

And In Conclusion: Control and Pipelining

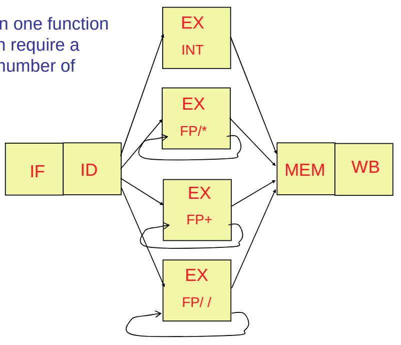
- Quantify and summarize performance
  - Ratios, Geometric Mean, Multiplicative Standard Deviation
- F&P: Benchmarks age, disks fail,1 point fail danger
- Control VIA State Machines and Microprogramming
- Just overlap tasks; easy if tasks are independent
- Speed Up ≤ Pipeline Depth; if ideal CPI is 1, then:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

- Hazards limit performance on computers:
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
  - Control: delayed branch, prediction
- Exceptions, Interrupts add complexity

Multicycle operations

More than one function unit, each require a variable number of cycles.



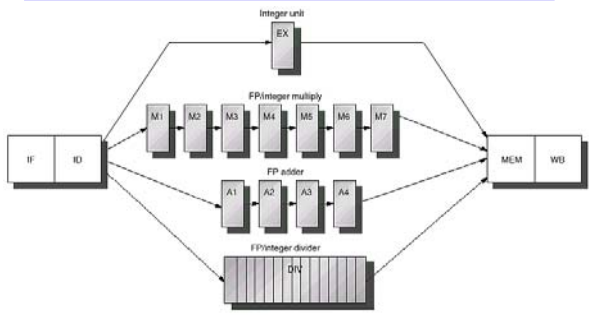
Multi cycles operations

- Assuming the following

Function	Unit latency	initiation period
Integer ALU	0	1
Data Memory	1	1
FP add	3	1
FP Multiply	6	1
FP Divide	24	24

Notice that FP add and multiply are pipelined (4 and 7 stages pipeline respectively).

Latency is the number of cycles between an instruction that produces a result and another one that uses the result.



Multicycle operations

```
MULTD   IF    ID    M1    M2    M3    M4    M5    M6    M7    MEM   WB
ADDD          IF    ID    A1    A2    A3    A4    MEM   WB
LD             IF    ID    EX    MEM   WB
SD                  IF    ID    EX    MEM   WB
```

Stages in red indicates when data are needed, in blue indicates when data are produced

Need to introduce more pipeline registers A1/A2, ..

Hazards and Forwarding

- Because the divide unit is not pipelined, structural hazards may arise
- Because of different running times. We may need to do more than one register write in a single cycle
- WAW hazard is now possible, WAR is not since they all read in one stage
- Instructions can complete in different order, more complicated exception handling
- Because of the longer latency, more RAW hazard

Hazards and Forwarding

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

Substantially longer stall and forwarding

Hazard and Forwarding

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
….											
….											
ADDD F2,F4,F6				IF	ID	A1	A2	A3	A4	MEM	WB
…											
…											
LD F8,0(R2)							IF	ID	EX	MEM	WB

Three different instruction writing in the same cycle, IF LD is issued one cycle earlier, with destination of F2, that will lead to WAW hazard

Hazards and Forwarding

- One way to deal with multiple writes is to have multiple write ports, but it may be rarely used.
- Another way is to detect the structural hazard by using an interlock
  1. We can track the use of the write port before it is issued (ID stage) and stall
  2. Or, we can detect this hazard at entering the MEM stage, it is easier to detect, and we can choose which instruction to proceed (the one with the longest latency?)

Maintaining Precise Exception

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

- This is known as out of order completion
- What if SUBF causes an Exception after ADDF is completed but before DIVF is, or if DIVF caused an exception after both ADDF and SUBF completed, there is no way to maintain a precise exception since ADDF destroys one of its operands

Maintaining Precise Exception (sol 1)

- Early solution is to ignore the problem
- More recent ones, are to introduce two modes of operations, fast but with imprecise exception, and slow with precise exception.
- DEC Alpha 2104, Power1 and Power-2, MIPS R8000

Maintaining Precise Exception (sol 2)

- Buffer the results of an operation until all the operations before it are completed.
- Costly, especially with long pipes.
- One variation is called history file, old values are stored in the history file and can be restored in case of exception
- Or, we an use future file, where new values are stored until all proceeding instructions are completed.

Maintaining Precise Exception (sol 3)

- Allow the exception to become imprecise, but we have to keep enough information so that the exception handling routine can recover.
- These information are usually the PC addresses of the instructions that were in the pipe during the exception, who finished, and who not.

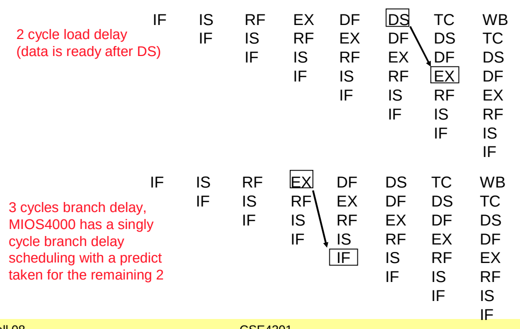
Maintaining Precise Exception (sol 4)

- A hybrid technique, Allow the instructions to be issued only if we are certain that all the instructions before the issuing instruction will complete without causing an exception
- That guarantees that no instruction after the interrupting one will be completed, and all instructions before it will complete.
- Must check for exception early in the EX stage.

MIPS4000

- 8 Stage Pipeline:
  - IF–first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  - IS–second half of access to instruction cache.
  - RF–instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  - EX–execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  - DF–data fetch, first half of access to data cache.
  - DS–second half of access to data cache.
  - TC–tag check, determine whether the data cache access hit.
  - WB–write back for loads and register-register operations.
- 8 Stages: What is impact on Load delay? Branch delay? Why?

MIPS4000


2 cycle load delay
(data is ready after DS)

```
IF  IS  RF  EX  DF  DS  TC  WB
    IF  IS  RF  EX  DF  DS  TC
        IF  IS  RF  EX  DF  DS
            IF  IS  RF  EX  DF
                IF  IS  RF  EX
                    IF  IS  RF
                        IF  IS
                            IF
```

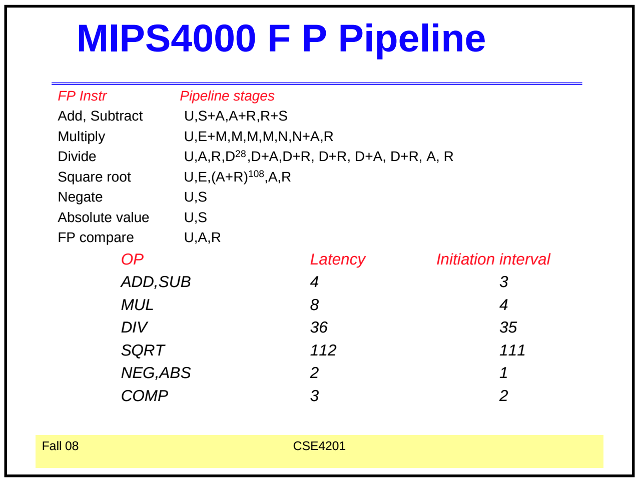
3 cycles branch delay, MIOS4000 has a singly cycle branch delay scheduling with a predict taken for the remaining 2

```
IF  IS  RF  EX  DF  DS  TC  WB
    IF  IS  RF  EX  DF  DS  TC
        IF  IS  RF  EX  DF  DS
            IF  IS  RF  EX  DF
                IF  IS  RF  EX
                    IF  IS  RF
                        IF  IS
                            IF
```

MIPS4000 FP Pipeline

- FP Adder, FP Multiplier, FP Divider
- Last step of FP Multiplier/Divider uses FP Adder HW
- 8 kinds of stages in FP units:

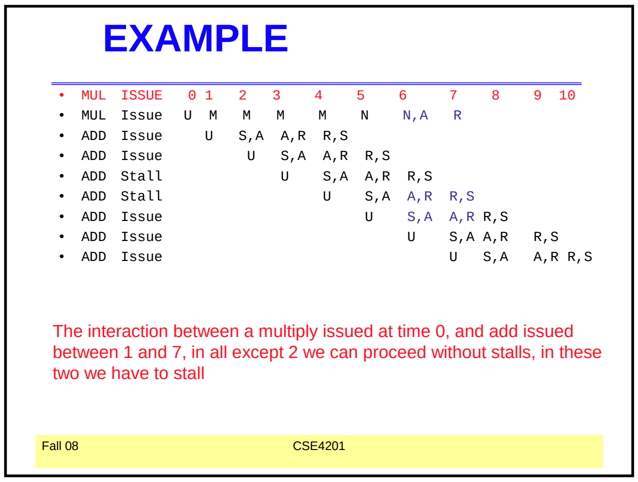
Stage	*Functional unit*	*Description*
A	FP adder	Mantissa ADD stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers



MIPS4000 F P Pipeline

FP Instr	*Pipeline stages*
Add, Subtract	U,S+A,A+R,R+S
Multiply	U,E+M,M,M,M,N,N+A,R
Divide	U,A,R,D^{28},D+A,D+R, D+R, D+A, D+R, A, R
Square root	U,E,(A+R)^{108},A,R
Negate	U,S
Absolute value	U,S
FP compare	U,A,R

OP	*Latency*	*Initiation interval*
ADD,SUB	*4*	*3*
MUL	*8*	*4*
DIV	*36*	*35*
SQRT	*112*	*111*
NEG,ABS	*2*	*1*
COMP	*3*	*2*



EXAMPLE

MUL	ISSUE	0	1	2	3	4	5	6	7	8	9	10
MUL	Issue	U	M	M	M	M	N	N,A	R			
ADD	Issue		U	S,A	A,R	R,S						
ADD	Issue			U	S,A	A,R	R,S					
ADD	Stall				U	S,A	A,R	R,S				
ADD	Stall					U	S,A	A,R	R,S			
ADD	Issue						U	S,A	A,R	R,S		
ADD	Issue							U	S,A	A,R	R,S	
ADD	Issue								U	S,A	A,R	R,S

The interaction between a multiply issued at time 0, and add issued between 1 and 7, in all except 2 we can proceed without stalls, in these two we have to stall