

Computer Systems Architecture CMSC 411 Unit 3 – Instruction Pipelining

Alan Sussman
February 13, 2003

Administrivia

- HW #2 due today – solution posted soon
- Quiz on Tuesday – Units 1 & 2
- HW #1 problem 1.17d
 - MFLOPs with coprocessor
 - answer shows that MFLOPs is computed as
$$\frac{(\# \text{ fp ops})/(\text{time for fp ops})}{(\# \text{ fp ops})/(\text{total time} - \text{time for integer ops})}$$
 - that is correct – don't count integer ops against MFLOPs
 - but both are counted in MIPS (both integer and fp ops are instructions!)

CMSC 411 - Alan Sussman

2

Last time

- Compiler/architecture interaction
 - providing a good target for the compiler can make a huge difference in performance – up to a factor of 10 on an f.p. intensive application
 - provide regularity, primitives, make costs of code sequences easy to determine
- MIPS/MIPS64 architectures
 - load/store, 64 bits (with 32-bit ops), 3 instruction formats for MIPS64 (all 32 bits), immediate and displacement addressing modes

CMSC 411 - Alan Sussman

3

So far

- What we mean by computer performance
- How to measure it
- How instruction sets are designed
- How the design influences performance

CMSC 411 - Alan Sussman

4

What's next

- A variety of hardware and compiler techniques to speed the execution of programs
 - What is pipelining? (Section A.1)
 - How does MIPS divide instructions into stages or cycles? (A.1)
 - What kinds of overheads are there in pipelining? (A.1)
 - How much speedup do we get? (A.1)
 - What are structural hazards, data hazards, and control hazards? (A.2)
 - How are these techniques used to reduce stalls:
 - data forwarding? (A.2)
 - instruction reordering? (A.2)
 - compiler approaches to reduce branch delays? (A.2)

CMSC 411 - Alan Sussman

5

What is pipelining?

- **Pipelining** is an implementation technique whereby multiple instructions are overlapped in execution
- In other words, at any given moment in the execution of a computer program, many different instructions are at various stages of completion!
- Example: Car wash

CMSC 411 - Alan Sussman

6

Throughput

- The number of instructions that *complete* per unit time
- Instructions take many clock cycles
Ideally, every clock cycle, we want a new instruction to begin (and end)
- This is how we will improve throughput

CMSC 411 - Alan Sussman

7

A MIPS implementation without pipelining

- Recall from CMSC 311 that instructions execute in different stages or cycles
 - **Instruction fetch cycle (IF)**: fetch the instruction from memory and update the program counter (PC) to point to the next instruction. Note: We're not using the *NPC* register that the book introduces.

$$IR \leftarrow \text{Mem}[PC]$$

$$PC \leftarrow PC + 4$$

CMSC 411 - Alan Sussman

8

MIPS w/o pipelining (cont.)

- **Instruction decode cycle (ID)**: Put the operands in pipeline registers *A* and *B*. Sign-extend the low order 16 bits of the IR and store in pipeline register *Imm*. (This sometimes holds the "immediate" constant.)

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

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

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

CMSC 411 - Alan Sussman

9

MIPS w/o pipelining (cont.)

- **Execution cycle (EC)**: Use the ALU
- If memory reference:

$$\text{ALUOutput} \leftarrow A + \text{Imm}$$
- If register-register ALU instruction:

$$\text{ALUOutput} \leftarrow A \text{ op } B$$
- If register-immediate ALU instruction:

$$\text{ALUOutput} \leftarrow A \text{ op } \text{Imm}$$
- If branch instruction: compute the branch address and check the branch condition:

$$\text{ALUOutput} \leftarrow PC + (\text{Imm} \ll 2)$$

$$\text{Cond} \leftarrow (A \text{ op } 0)$$

(but *PC* or *Imm* should be adjusted down by 4 to make this work right).

CMSC 411 - Alan Sussman

10

MIPS w/o pipelining (cont.)

- **Memory access cycle (MEM)**: finish loads, stores, and branches:
 - Load: $\text{LMD} \leftarrow \text{Mem}[\text{ALUOutput}]$
 - Store: $\text{Mem}[\text{ALUOutput}] \leftarrow B$
 - Branch: *if* Cond *then* $PC \leftarrow \text{ALUOutput}$
else PC is ok

CMSC 411 - Alan Sussman

11

MIPS w/o pipelining (cont.)

- **Write-back cycle (WB)**: update the registers
 - Register-register ALU instruction:

$$\text{Regs}[IR_{16..20}] \leftarrow \text{ALUOutput}$$
 - Register-immediate ALU instruction:

$$\text{Regs}[IR_{11..15}] \leftarrow \text{ALUOutput}$$
 - Load instruction:

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

CMSC 411 - Alan Sussman

12

Some notes

- All instructions take 4-5 cycles
- Some of the temporary registers could be eliminated, but they become convenient in a minute for pipelining
- Could build the architecture so that these 5 cycles are one clock cycle, not 5
- We assume that there are separate data paths for instruction memory and for data memory (so loads and stores don't interfere with instruction fetch), usually implemented via separate caches

CMSC 411 - Alan Sussman

13

A pipelined implementation of MIPS

Suppose we have 5 load/store instructions to execute, all involving different registers and memory locations. Fig. A.1

Inst. #	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
$i+1$		IF	ID	EX	MEM	WB			
$i+2$			IF	ID	EX	MEM	WB		
$i+3$				IF	ID	EX	MEM	WB	
$i+4$					IF	ID	EX	MEM	WB

CMSC 411 - Alan Sussman

14

With pipeline registers

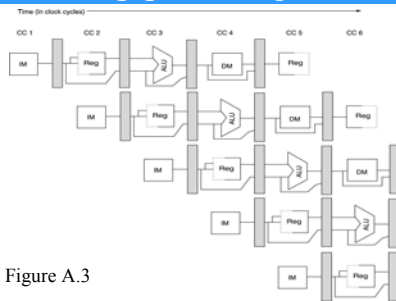


Figure A.3

© 2003 Elsevier Science (USA). All rights reserved.
CMSC 411 - Alan Sussman

15

Communications paths and timing

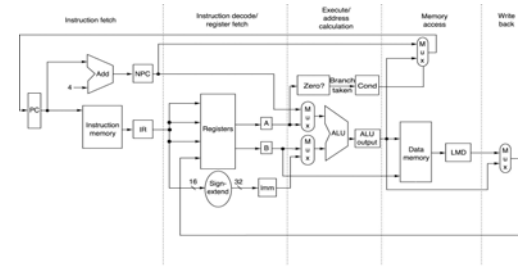


Figure A.17

© 2003 Elsevier Science (USA). All rights reserved.
CMSC 411 - Alan Sussman

16

Ideal throughput from pipelining

- Example 1: Suppose we have 100 load/store instructions to execute, and we arrange to have no register conflicts
- Time for original MIPS implementation: $100 \text{ instructions} \times 5 \text{ cycles per instruction} = 500 \text{ cycles}$
- Time for pipelined MIPS implementation: 1st instruction takes 5 cycles. The others each finish 1 cycle later than the preceding one.
Time = $5 + 99 = 104 \text{ cycles}$
- Speedup = $500/104 \approx 5$
- This is the "ideal case" - life is not that simple

CMSC 411 - Alan Sussman

17

A more realistic case

- Example 2: Suppose that in our original MIPS implementation, we could run the stages this fast:
 - IF - 10ns
 - ID - 8ns
 - EX - 7ns
 - MEM - 10ns
 - WB - 5ns
- Then time for original MIPS implementation of 100 load/store instructions:
 - $100 \text{ instructions} \times 40 \text{ ns per instruction} = 4000 \text{ ns}$

CMSC 411 - Alan Sussman

18

Example 2 (cont.)

- Time for pipelined MIPS implementation:
We have to synchronize the stages, so we need to run the clock at 10 ns
- 1st instruction takes 50 ns. The others each finish 1 cycle later than the preceding one.
 - Time = 50 ns + 99*10 ns = 1040 ns
- Speedup = 4000/1040 $\approx$ 3.85

CMSC 411 - Alan Sussman

19

Even more realistic case

- Example 3: The original MIPS implementation doesn't always need to use the MEM cycle
 - IF - 10ns
 - ID - 8ns
 - EX - 7ns
 - MEM - 10ns
 - WB - 5ns
- Suppose that only 30% of instructions use memory access. So, on average, for every 100 instructions, we have about 70 that use 4 stages and 30 that use 5.

CMSC 411 - Alan Sussman

20

Example 3 (cont.)

- Time for original MIPS implementation:
 - 70 instructions $\times$ 30 ns per instruction + 30 instructions $\times$ 40 ns per instruction = 3300 ns
- Time for pipelined MIPS implementation: We have to synchronize the stages, so we need to run the clock at 10 ns, and we need 5 cycles for **every** instruction.
 - 1st instruction takes 50 ns. The others each finish 1 cycle later than the preceding one
 - Time = 50 ns + 99*10 ns = 1040 ns
- Speedup = 3300/1040 $\approx$ 3.17

CMSC 411 - Alan Sussman

21

Overhead of pipelining

- We just summarized the two major overhead costs in pipelining:
 - making the time for every stage equal the time for the longest stage
 - making the time for every instruction equal the time for the longest instruction (not quite true, but true for a wide range of instructions)
- Unfortunately, the speedup of pipelining is reduced even further by **hazards** that cause "bubbles" in the pipeline

CMSC 411 - Alan Sussman

22

Pipeline hazards cause stalls

- When some instruction is unable to complete on schedule, we must
 - finish the earlier instructions on schedule
 - delay the later instructions
- This is called **stalling** the pipeline

CMSC 411 - Alan Sussman

23

Pipeline hazards

- What causes delays in instruction completion?
 - **Structural hazards** are hardware delays
Example: memory does not respond to a request as fast as it is expected to
 - **Data hazards** arise when data can be predicted to be unready at the time it is needed
Example: an instruction needs a register that a previous instruction is still modifying
 - **Control hazards** arise when we need to do something other than incrementing the PC by 4
Example: conditional branch, jump

CMSC 411 - Alan Sussman

24

Pipeline hazards (cont.)

Pipeline hazards reduce throughput and speedup even more! Fig. A.5
Structural hazard – a load with 1 memory port for data/instructions

Inst #	1	2	3	4	5	6	7	8	9	10
Load	IF	ID	EX	MEM	WB					
$i+1$		IF	ID	EX	MEM	WB				
$i+2$			IF	ID	EX	MEM	WB			
$i+3$				stall	IF	ID	EX	MEM	WB	
$i+4$						IF	ID	EX	MEM	WB
$i+5$							IF	ID	EX	MEM
$i+6$								IF	ID	EX

CMSC 411 - Alan Sussman

25

Pipeline hazards (cont.)

- Example 4: In Example 3, had on average, 70 instructions that use 4 stages and 30 that use 5
- Time for original MIPS implementation = 3300 ns
- Suppose that 5 of those instructions involve branches. So 5 times, need to wait until the ID cycle of one instruction is complete before start the IF cycle of the next instruction.
- Therefore, the next instruction will start 2 cycles later, not 1. So add 5 cycles to the time.

CMSC 411 - Alan Sussman

26

Example 4 (cont.)

- Time for pipelined MIPS implementation:
 - 1st instruction takes 50 ns. The others each finish 1 cycle later than the preceding one, but there is a 5 cycle hazard penalty
 - Time = $50 \text{ ns} + 99 \cdot 10 \text{ ns} + 5 \cdot 10 \text{ ns} = 1090 \text{ ns}$
- Speedup = $3300/1090 \approx 3.03$

CMSC 411 - Alan Sussman

27

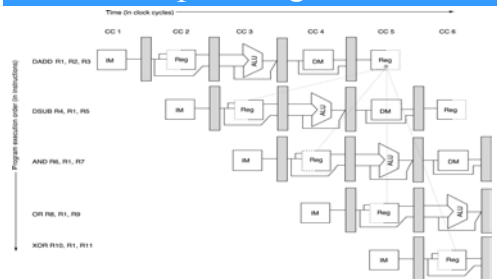
Data hazards

- A **data hazard** occurs when a piece of data is not available when it is needed
 - Perhaps there was a *cache miss*: we expected the value to be in cache, but instead we need to find it in memory
 - Perhaps it is involved in a previous computation that has not yet completed

CMSC 411 - Alan Sussman

28

Example – Figure A.6



© 2003 Elsevier Science (USA). All rights reserved.
CMSC 411 - Alan Sussman

29

Types of data hazards

- **RAW** : read after write
 - One instruction writes a value. A later instruction reads it. Problem: an old value may be read.
- **WAW** : write after write
 - One instruction writes a value. A later instruction writes in the same location. Problem: the final value may be the first, rather than the second.
- **WAR** : write after read
 - One instruction reads a value. A later instruction writes in the same location. Problem: the value read may be the changed value rather than the original. This ordinarily cannot happen.

CMSC 411 - Alan Sussman

30

How to avoid data hazard stalls: forwarding

- Need to move data more quickly from the ALU output to the ALU inputs
- So **forward** the output by designing the circuits so that the ALU output is always fed back (immediately) into the ALU input latches (pipeline registers), adding a circuit to choose between the ALU output and the other registers

CMSC 411 - Alan Sussman

31

Computer Systems Architecture CMSC 411 Unit 3 – Instruction Pipelining

Alan Sussman
February 20, 2003

Administrivia

- HW #3 available on web page
- Quiz postponed until Tuesday
- I'm out of town Tuesday – lecture and quiz given by guest lecturer Henrique Andrade
 - no office hours Tuesday or Wednesday either
 - make an appointment for Monday, or after Wednesday

CMSC 411 - Alan Sussman

33

Last time

- Pipelining – overlap execution of multiple instructions, to improve *throughput*, not latency
- 5 basic pipe stages
 - IF, ID, EX, MEM, WB
 - with *pipeline registers* between stages
- Costs of pipelining include
 - having to run all stages at same clock rate
 - all instructions have to go through all stages
- Hazards cause *stalls*
 - structural – resource conflicts
 - data – RAW, WAW, WAR – *forwarding* helps
 - control – from branches and jumps

CMSC 411 - Alan Sussman

34

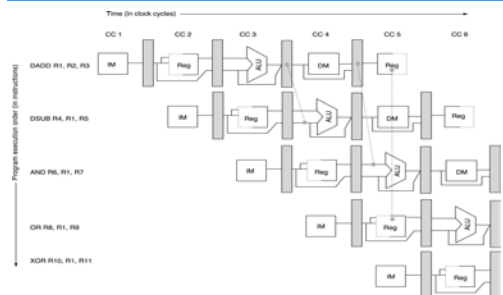
How to avoid data hazard stalls: forwarding

- Need to move data more quickly from the ALU output to the ALU inputs
- So **forward** the output by designing the circuits so that the ALU output is always fed back (immediately) into the ALU input latches (pipeline registers), adding a circuit to choose between the ALU output and the other registers

CMSC 411 - Alan Sussman

35

Forwarding – Fig. A.7

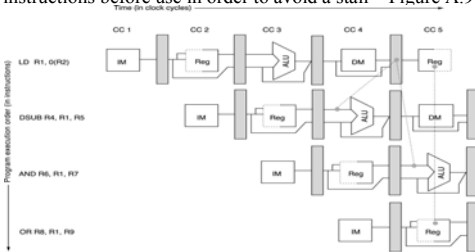


© 2003 Elsevier Science (USA). All rights reserved.
CMSC 411 - Alan Sussman

36

Sometimes forwarding not enough

- Example:** Data needs to be loaded from memory at least two instructions before use in order to avoid a stall – Figure A.9



© 2003 Elsevier Science (USA). All rights reserved.
CMSC 411 - Alan Sussman

37

Forwarding (cont.)

- Compilers need to be smart enough to prevent stalls when possible

```
LD R1, b
LD R2, c
LD R3, d    ADD can't be done yet
DADD R4,R1,R2
DADD R4,R3,R4  ok by forwarding
LD R5, f    need to start this before
              a = b + c + d completes
```

Example:

```
a = b + c + d;
e = d - f;
```

- Need to make sure that the first ADD operation delays until b and c are loaded

```
SD a, R4
DSUB R6,R3,R5
SD e, R6    ok by forwarding
```

CMSC 411 - Alan Sussman

38

Forwarding (cont.)

- Rules for interchanging instructions:
 - must be in same block (i.e., no branches between them)
 - must check graph of dependencies to make sure they are independent

CMSC 411 - Alan Sussman

39

How the MIPS pipeline introduces stalls

- Data hazards are checked during instruction decode (ID) - if a hazard exists, the EX cycle is delayed (i.e., the instruction is not **issued**), a "no-op" is issued instead
- The ID cycle also determines whether data forwarding is needed

CMSC 411 - Alan Sussman

40

Control hazards

- Question:** When do we find out that the PC needs to be modified?
- Answer:** In pipeline stage ID of a branch instruction
- So, if a branch is taken (i.e., if the PC is modified), then have to wait until the *next* cycle before can fetch the correct instruction

CMSC 411 - Alan Sussman

41

Control hazards (cont.)

Branch inst.	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Successor + 1				IF	ID	EX	MEM
Successor + 2					IF	ID	EX

Wastes 1 clock cycle

CMSC 411 - Alan Sussman

42

Example

- If branch in 30% of instructions, then instead of executing 1 instruction per cycle, have 70% of instructions executing in 1 cycle and 30% of instructions executing in 2 cycles
- An average of $.7 + .6 = 1.3$ cycles per instruction
 - Worse by 30%

CMSC 411 - Alan Sussman

43

Compiler approaches to branch delays

- Freeze or flush** the pipeline when determine that a branch is taken - refer back to Figure A.11 (a stall is inserted)
- Predict not taken:** continue to begin execution of instructions as if the branch is not taken, but change them to a "no-op" if the branch is taken

CMSC 411 - Alan Sussman

44

Predict not taken scheme – Fig. A.12

Untaken branch	IF	ID	EX	MEM	WB				
Inst. i+1		IF	ID	EX	MEM	WB			
Inst. i+2			IF	ID	EX	MEM	WB		
Inst. i+3				IF	ID	EX	MEM	WB	
Inst. i+4					IF	ID	EX	MEM	WB

Taken branch	IF	ID	EX	MEM	WB				
Inst. i+1		IF	idle	idle	idle	idle			
Branch target			IF	ID	EX	MEM	WB		
B.t. + 1				IF	ID	EX	MEM	WB	
B.t. + 2					IF	ID	EX	MEM	WB

CMSC 411 - Alan Sussman

45

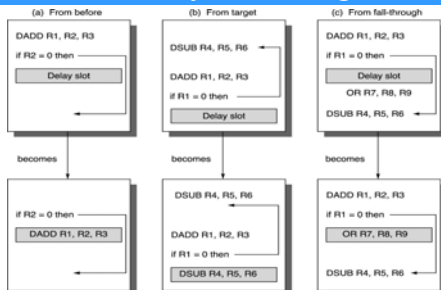
Compiler approaches (cont.)

- Predict taken:** Good if most of the branches are from loops
- Schedule using **branch delay slots**, reordering the code to test the branch earlier

CMSC 411 - Alan Sussman

46

Branch delay slot – Fig. A.14



© 2003 Elsevier Science (USA). All rights reserved.
CMSC 411 - Alan Sussman

47

Scheduling branch delay slot

- If taken from before branch
 - branch must not depend on rescheduled instruction
 - always improves performance
- If taken from branch target
 - must be OK to execute rescheduled instructions if branch not taken, and may need to duplicate insts.
 - performance improved when branch taken
- If taken from fall through
 - must be OK to execute insts. if branch taken
 - improves performance when branch not taken

CMSC 411 - Alan Sussman

48

Compiler approaches (cont.)

- Some machines have **cancelling** branches, to change the next instruction to a ``no-op" when necessary - then there are no requirements on the scheduling strategy
- In a set of benchmarks not shown in book, 70% of the branch hazards in simpler version of MIPS can be eliminated by branch scheduling

CMSC 411 - Alan Sussman

49

Summary

- Pipelining can speed instruction execution...
- But need to deal with structural hazards, data hazards, and control hazards
- Next
 - How to handle exceptions?
 - How to handle long instructions, such as floating point arithmetic?

CMSC 411 - Alan Sussman

50

The problem

- Question: What makes pipelining hard to implement?
- **Answer:** Surprises
- Technical names for surprises:
 - exceptions
 - faults
 - interrupts

CMSC 411 - Alan Sussman

51

Some examples of exceptions

- Request for I/O
- Arithmetic troubles: overflow or underflow
- Page fault: data not in (physical) memory
- Illegal address, giving a memory protection violation
- Hardware failure

CMSC 411 - Alan Sussman

52

Classifying exceptions

- **Synchronous:** repeatable every time
Example: DIV R2, R2, R0
Asynchronous: caused by external events like hardware failure and devices external to processor and memory
- **User requested:** user task asks for it (example: breakpoint)
Coerced: cannot be predicted by user
- **User maskable:** can be disabled by user task
Example: arithmetic exception
Nonmaskable: cannot be turned off
Example: hardware failure

CMSC 411 - Alan Sussman

53

Classifying exceptions (cont.)

- **Within instruction:** prevents instruction from completing
Between instructions: no instruction prevented
- **Terminating:** stops the task
Resuming: task can continue
- Machines that handle exceptions, save the state, and then restart correctly are said to be **restartable**

CMSC 411 - Alan Sussman

54

Categorizing exceptions – Fig. A. 27

Exception type	Synch. vs. asynch.	User request vs. coerce	User maskable vs. not	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynch	Coerced	Not	Between	Resume
Invoke OS	Synch	User req.	Not	Between	Resume
Tracing instructions	Synch	User req.	Maskable	Between	Resume
Breakpoint	Synch	User req.	Maskable	Between	Resume
Integer overflow	Synch	Coerced	Maskable	Within	Resume
Floating pt. overflow/underflow	Synch	Coerced	Maskable	Within	Resume

CMSC 411 - Alan Sussman

55

Categorizing exceptions (cont.)

Exception type	Synch. vs. asynch.	User request vs. coerce	User maskable vs. not	Within vs. between instructions	Resume vs. terminate
Page fault	Synch	Coerced	Not	Within	Resume
Misaligned memory access	Synch	Coerced	Maskable	Within	Resume
Mem. prot. violation	Synch	Coerced	Not	Within	Resume
Undefined instruction	Synch	Coerced	Not	Within	Terminate
Hardware malfunction	Asynch	Coerced	Not	Within	Terminate
Power failure	Asynch	Coerced	Not	Within	Terminate

CMSC 411 - Alan Sussman

56

The most difficult exceptions...

- ... are those that occur within EX or MEM stages and need to be handled in a restartable way
- Why difficult? Handling one includes:
 - the next IF gets a "trap instruction"
 - until the trap is taken, turn off all "writes" for the faulting instruction and those that follow it.
 - what does the trap do?
 - The trap transfers control to the exception handling routine in the operating system, which saves the PC of the faulting instruction and handles the fault
 - the task is then resumed, using the saved PC and the MIPS instruction RFE or something like it
- Note:** May need to save several PCs if delayed branches are involved

CMSC 411 - Alan Sussman

57

Exceptions (cont.)

- Ideally, pipeline can be interrupted so that instructions before the fault complete. Then want to restart execution just after the faulting instruction - **precise exception handling**
- This is the right way to do it, but sometimes architects/manufacturers take shortcuts

CMSC 411 - Alan Sussman

58

When do MIPS exceptions occur?

- IF**
 - page fault on instruction fetch
 - misaligned memory access
 - memory protection violation
- ID**
 - undefined or illegal opcode
- EX**
 - arithmetic exception
- MEM**
 - page fault on data fetch/store
 - misaligned memory access
 - memory protection violation
- WB:** None!

CMSC 411 - Alan Sussman

59

Computer Systems Architecture CMSC 411 Unit 3 – Instruction Pipelining

Alan Sussman
February 25, 2003

Administrivia

- HW #3 due next Tuesday, March 4
- Quiz today – last 25 minutes of class
- No office hours today or tomorrow
 - see TA or try Thursday

CMSC 411 - Alan Sussman

61

Last time

- Forwarding
 - to use ALU or load result before WB
 - compiler reorders instructions to prevent stalls, use forwarding
- Control hazards lead to *branch delays*
 - because branch target isn't computed until ID
 - one (partial) solution is for compiler to schedule branch delay slots
- Exceptions
 - machine must save pipeline state, handle exception (with OS), and restart where exception occurred
 - precise vs. imprecise exception handling
 - program generated ones can occur in all pipe stages except WB

CMSC 411 - Alan Sussman

62

Examples of exception handling

LD	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

- Handle the MEM fault first, then restart

LD	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

- IF fault occurs first, even though LD will fault later
- But for precise exceptions, must handle LD fault first

CMSC 411 - Alan Sussman

63

How is this done?

- **Answer:** Don't handle exceptions until the WB stage
 - each instruction has an associated status vector that keeps track of faults
 - any bit set in the status vector turns off register writes and memory writes
 - in WB stage, the status vector is checked and any fault is handled
 - So, since instructions reach WB in proper order, faults for earlier instructions are handled *before* faults for later instructions
 - Unfortunately, will need to violate this later (for instructions that don't reach WB in proper order)

CMSC 411 - Alan Sussman

64

Commitment

- When an instruction is guaranteed to complete, it is **committed**
- Life is easier if no instruction changes the machine state before it is committed
- In MIPS, commitment occurs at the end of the MEM stage - that's why register update occurs in the stage after that
- Some machines muddy the state before commitment, and the exception handler must do its best to restore the state that existed before the instruction started

CMSC 411 - Alan Sussman

65

Complications caused by long instructions

- So far, all MIPS instructions take 5 cycles
- But haven't talked yet about the floating point instructions
- Take it on faith that floating point instructions are inherently slower than integer arithmetic instructions
 - doubters may consult Appendix H in H&P online

CMSC 411 - Alan Sussman

66

How slow is slow?

- Some typical times:
 - latency** is the number of cycles between an instruction that produces a result and one that uses it
 - initiation interval** is the number of cycles between two instructions of the same kind (for example, two ADD.Fs)

Figure A.30

Instruction	Latency	Initiation
ALU uses	0	1
Load/store	1	1
ADD.F, SUB.F	3	1
DIV.F	24	25

CMSC 411 - Alan Sussman

67

Examples

- If have a string of instructions:
 - DADD
 - DSUB
 - AND
 - OR
 - SLLI
- then there are no delays in the pipeline, because initiation=1 means can start one of these instructions every cycle, and latency=0 means that results from one instruction will be available when the next instruction needs them.

CMSC 411 - Alan Sussman

68

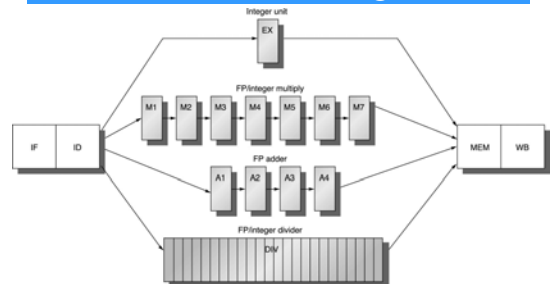
Examples (cont.)

- Suppose have a string of instructions
 - ADD.F
 - SUB.F
- Then initiation=1 means that can start **SUB.F** one cycle behind **ADD.F**
- But latency=3 means that this will work right *only if* **SUB.F** doesn't need **ADD.F**'s results
- If it does need the results, then need two instructions in between **ADD.F** and **SUB.F** to prevent bubbles in the pipeline

CMSC 411 - Alan Sussman

69

Functional units - Fig. A.31



© 2003 Elsevier Science (USA). All rights reserved.
CMSC 411 - Alan Sussman

70

Examples (cont.) - Fig. A.32

MUL.D	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
ADD.D		IF	ID	A1	A2	A3	A4	MEM	WB		
L.D			IF	ID	EX	MEM	WB				
S.D				IF	ID	EX	MEM	WB			

Italics shows where data is needed, **bold** where a result is available

CMSC 411 - Alan Sussman

71

Hazards caused by long instructions

- The floating point adder and multiplier are pipelined, but the divider is not - that is why the initiation interval for divide is 25
 - A program will run **very** slowly if it does too many of these!
- It will also run slowly if the results of the divide are needed too soon

CMSC 411 - Alan Sussman

72

FP stalls from RAW hazards – Fig. A.33

Inst.	1	2	3	4	5	6	7	8	9
L.D F4,0(R2)	IF	ID	EX	MEM	WB				
MUL.D F0,F4,F6		IF	ID	stall	M1	M2	M3	M4	M5
ADD.D F2,F0,F8			IF	stall	ID	stall	stall	stall	stall
S.D F2,0(R2)				IF	stall	stall	stall	stall	stall

Inst.	10	11	12	13	14	15	16	17
L.D								
MUL.D	M6	M7	MEM	WB				
ADD.D	stall	stall	A1	A2	A3	A4	MEM	
S.D	stall	stall	ID	EX	stall	stall	MEM	

CMSC 411 - Alan Sussman

73

Long instructions (cont.)

- It is possible that two instructions enter the WB stage at the same time

ADD.D	IF	ID	A1	A2	A3	A4	MEM	WB
LD		IF	ID	ALU	MEM	WB		
DADD			IF	ID	ALU	MEM	WB	
DADD				IF	ID	ALU	MEM	WB

- A structural hazard

CMSC 411 - Alan Sussman

74

Long instructions (cont.)

- Instructions can finish in the wrong order
- This can cause WAW hazards
 - see p. A-52 of H&P for an example
- This violation of WB ordering defeats the previous strategy for precise exception handling

CMSC 411 - Alan Sussman

75

WAW structural hazard – Fig. A.34

MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
L.D F2,0(R2)							IF	ID	EX	MEM	WB

CMSC 411 - Alan Sussman

76

How to detect hazards in ID

- Early detection would prevent trouble
- Check for structural hazards:**
 - will the divide unit clear in time?
 - will WB be possible when we need it?
- Check for RAW data hazards:**
 - will all source registers be available when needed?
- Check for WAW data hazards:**
 - Is the destination register for any ADD.D, multiply or divide instructions the same register as the destination for this instruction?
- If anything dangerous could happen, delay the execute cycle so no conflict occurs

CMSC 411 - Alan Sussman

77

Precise exception handling for long instructions

Example:

```
DIV.D F0, F2, F4
ADD.D F10, F10, F8
SUB.D F12, F12, F14
```

- Suppose
 - ADD.D completes,
 - then SUB.D has a floating-point exception,
 - then DIV.D detects an exception
- Big trouble, because ADD.D has destroyed register F10

CMSC 411 - Alan Sussman

78

Possible fixes

- Give up and just do **imprecise exception handling**
 - tempting, but very annoying to users
- **Delay WB** until all previous instructions complete
 - since so many instructions can be active, this is expensive - requires a lot of supporting hardware
- Write, to memory, a **history file** of register and memory changes so can undo instructions if necessary
 - or keep a **future file** of computed results that are waiting for MEM or WB

CMSC 411 - Alan Sussman

79

Possible fixes (cont.)

- Let the **exception handler** finish the instructions in the pipeline and then restart the pipe at the next instruction
- Have the floating point units **diagnose exceptions in their first or second stages**, so can handle them by methods that work well for handling integer exceptions

CMSC 411 - Alan Sussman

80

Computer Systems Architecture CMSC 411 Unit 3 – Instruction Pipelining

Alan Sussman
February 27, 2003

Administrivia

- Quiz returned Tuesday
 - answers posted on web page
- Read Chapter 3 – Unit 4 on instruction-level parallelism
- HW #3 due Tuesday, March 4
- HW #4 posted soon

CMSC 411 - Alan Sussman

82

Last time

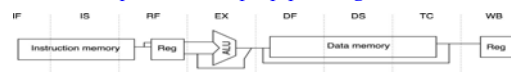
- Exception handling
 - for 5 stage pipeline, handle them in WB stage, to keep proper order
 - after the instruction *commits*
- Long instructions – generally means f.p. instructions
 - higher latency, and initiation interval, than other (integer) instructions
 - typically means the EX stage is multiple cycles, and sometimes not pipelined (e.g., divider)
 - can get 2 (or more) instructions trying to enter WB at same time – *structural hazard*
 - can get instructions finishing in wrong order – may cause WAW hazards, messing up precise exception handling
 - detect hazards in ID stage, and delay EX if a conflict occurs
 - finally, several ways to handle exceptions – history/future file, software (OS exception handler), diagnose early in pipeline, ...

CMSC 411 - Alan Sussman

83

A case study: MIPS R4000 pipeline design

- MIPS64 architecture, with deeper 8 stage pipeline
 - to get higher clock rates
 - extra stages come from memory accesses
 - techniques called *superpipelining*



© 2003 Elsevier Science (USA). All rights reserved.
CMSC 411 - Alan Sussman

84

MIPS R4000 pipeline stages

- **IF** – 1st half instruction fetch
 - PC selection and start instruction cache access
- **IS** – 2nd half instruction fetch
 - complete instruction cache access
- **RF** – instruction decode, register fetch, hazard checking, instruction cache hit detection
- **EX** – execution
 - includes effective address computation, ALU operation, branch target computation and condition evaluation

CMSC 411 - Alan Sussman

85

MIPS R4000 pipeline (cont.)

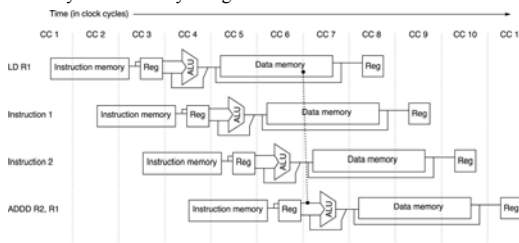
- **DF** – 1st half data fetch
 - 1st half of data cache access
- **DS** – 2nd half data fetch
 - complete data cache access
- **TC** – tag check
 - determine whether data cache access *hit*
- **WB** – write back for loads and ALU operations

CMSC 411 - Alan Sussman

86

MIPS R4000 pipeline (cont.)

A 2 cycle load delay – Fig. A.38

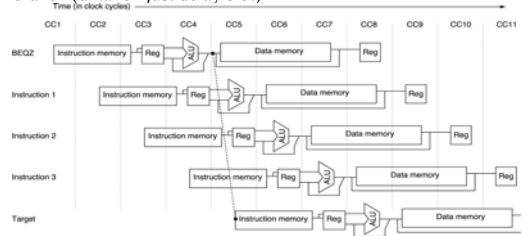


© 2003 Elsevier Science (USA). All rights reserved.
CMSC 411 - Alan Sussman

87

MIPS R4000 pipeline (cont.)

A 3 cycle branch delay – 1 delay slot + 2 cycle stall for taken branch (untaken just delay slot)



© 2003 Elsevier Science (USA). All rights reserved.
CMSC 411 - Alan Sussman

88

Forwarding

- Deeper pipeline increases number of levels of forwarding for ALU operations
 - 4 possible sources for an ALU bypass – EX/DF, DF/DS, DS/TC, TC/WB

CMSC 411 - Alan Sussman

89

Floating point pipeline

- 3 functional units
 - divider, multiplier, adder
- Double precision FP ops take from 2 (negate) up to 112 cycles (square root)
- Effectively 8 stages, combined in different orders for various FP operations
 - one copy of each stage, and some instructions use a stage zero or more times, and in different orders
- Overall, rather complicated ...
 - see H&P for more details

CMSC 411 - Alan Sussman

90

R4000 pipeline performance

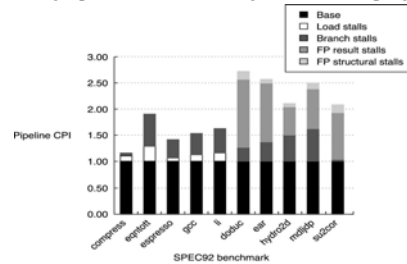
- 4 major causes of pipeline stalls
 - **load stalls** – from using load result 1 or 2 cycles after load
 - **branch stalls** – 2 cycles on every taken branch, or empty branch delay slot
 - **FP result stalls** – RAW hazards for an FP operand
 - **FP structural stalls** – from conflicts for functional units in FP pipeline

CMSC 411 - Alan Sussman

91

SPEC92 benchmarks

Assuming a perfect cache – 5 integer and five FP programs



© 2003 Elsevier Science (USA). All rights reserved.

92

Dynamically scheduled pipelines

- We'll cover this, and the scoreboard technique, in Unit 4
 - need some general background first

CMSC 411 - Alan Sussman

93

Pitfalls

- **Unexpected hazards do occur ...**
 - for example, when a branch is taken before a previous instruction finishes
- **Extensive pipelining can slow a machine down, or lead to worse cost-performance**
 - more complex hardware can cause a longer clock cycle, killing the benefits of more pipelining

CMSC 411 - Alan Sussman

94

Pitfalls (cont.)

- **A poor compiler can make a good machine look bad**
 - compiler writers need to understand the architecture in order to
 - optimize efficiently and
 - avoid hazards
 - better to eliminate useless instructions, than make them run faster

CMSC 411 - Alan Sussman

95