Overview of Pipelining

Venkatesh Akella

EEC 270

Winter 2005

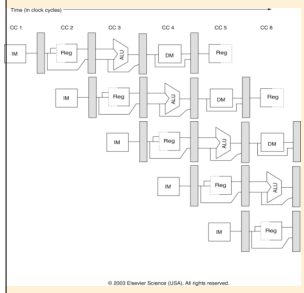
Outline

- Overview
- Hazards and how to eliminate them?
- Performance Evaluation with Hazards
- Precise Interrupts
- Multicycle Operations
- MIPS R4000 - 8-stage pipelined processor, A Case Study
- Out of order Execution - An introduction

What is pipelining?

- It is an implementation technique.
- Think Assembly Line. A Car Wash for example
- Overlapped execution of multiple instructions
- Better utilization of the resources
- Reduces average execution time per instruction
- Reduces clock period (Tc) of the performance equation
- If original program takes multiple clock cycles, pipelining can be viewed as reducing CPI
- Latency of an instruction remains same or increases slightly
- Throughput increases

Classic 5 Stage MIPS Pipeline



- Stage 1 (IF)

NPC = PC + 4 || IR = MEM(PC)

- Stage 2 (ID)

Read registers; decode; resolve branch and compute branch target addr (BTA)

Why? Fixed field instr. Format otherwise you need to decode first and then resolve branches

- Stage 3 (EX)

Effective address || ALU operation

(works for LS arch, otherwise you need another pipe stage)

- Stage 4 (MEM)

Read Data | Store Data

- Stage 5 (WB)

Complete ALU op or Load Operation

Implementation/Performance

- Unpipelined Processor - 2 to 5 cycles per instruction
- Pipelined Processor
  - Add pipeline registers, so that multiple copies of the state is maintained corresponding to multiple simultaneously executing instructions
  - Replicate resources to avoid structural hazards or pipeline the functional units/memory
  - Potential (Ideal) Speed up = N where n is number of stages but
  - Pipelining requires 5x memory bandwidth compared to unpipelined
  - Clock skew introduces overhead - Watch out for Amadahl's law, limits how many stages you can have
  - Branches ruin the party!

Hazards

- Structural
- Data Hazards
- Control Hazards

Stall till the hazards clears - simple strategy

Speed up = $CPI_{unpipelined}$ / (1 + Pipeline Stalls/instr)

So, we need to eliminate hazards:

Structural Hazards

- Duplicate resources
- Pipeline functional units

Data Hazards

DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11

What is the value of produced by DSUB if you don't do anything? Non deterministic

<u>What do we do?</u>

R1 is not needed till cycle 4; but it is produced in cycle 3, so in the case of a hazard, the DSUB instruction can read from the output oF ALU

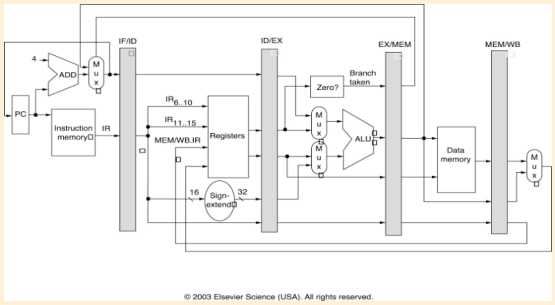
<u>Called Forwarding or Bypass</u>

Should the Forwarding logic be enabled during an interrupt - Why?

LD R1, 0(R2)

DSUB R4, R1, R1 - will forwarding work? Load use hazard :-(

Basic Pipeline Datapath



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

Implement control by carrying the control signals,
Branch Penalty is 3 cycles, can we do better

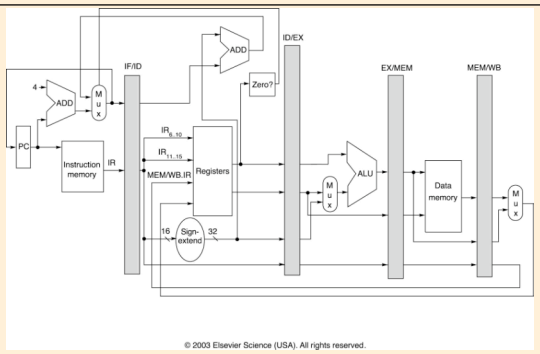
Control Hazards

- Branch is resolved at end of cycle 4, so the instruction fetch in cycle 2,3,4 is "speculative"
- Easiest solution - Fetch again in cycle 3
- Branch penalty is 3
- Problem every branch instructions takes 4 cycles :-(
- Supposing branch is not taken, you had the right instruction in cycle 2
- Predict Not Taken and flush if you are wrong

(What's the branch penalty now?)

- Branch Delay slot
- Predict Taken - Will it work with our pipeline? Why?

Reducing Branch Penalty to 1 cycle



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

What makes pipelining difficult?

- Exceptions
- Instruction set complications
- Multicycle operations

Exceptions => I/O device request, breakpoint, Page Fault, Mem Protection, system call, overflow, misaligned, hardware malfunction

- Sync vs Async
- Predictable vs Unpredictable. Eg: user requested I/O, syscall are predictable
- Maskable vs Unmaskable => can the user disable the interrupt?
- Within instructions or between instructions => cache miss, page fault
- Terminate vs resume - Terminate is obviously easier.
- Restartable - if a pipeline can handle the exceptions, save state and resume without affecting the execution of the program the processor is said to be restartable

PRECISE INTERRUPTS

- If the pipeline can be stopped so that all instructions before the offending instruction can be completed and all instructions following the offending instruction can be started from scratch

So, what's the problem?

a) Sometimes the faulting instr may overwrite the source operands (FP mult), before the exception you need to extract them and save them

b) An instr after the faulty inst may complete and update state eg: DIVF R1, R2, F3

DADD R2, R2 0

-- assume out-of-order completion

So, what do you do?

- Do not update the state of the instruction and its successor unless you know the instruction cannot cause an exception.
- Allow 2 modes of operations - Precise and Imprecise
- Imprecise Mode is 10x faster than Precise mode? Why?

Eg: you can disallow out-of-order completion in the precise mode

Why precise?

Makes OS interface simple; why? hw is dealing the cleanup

With Virtual Memory you do not have a choice.

Handling Exceptions

- Force a trap instruction into the pipeline on the next instruction fetch
- Until the trap is taken, turn off all the writes for the faulting instruction - this prevents state changes for instructions that will not be completed
- Jump to exception handler, which will save the PC of the offending instruction
- Delayed Branches make this difficult - more than one PC may have to be saved

Multiple Exceptions can occur in same cycle

LOAD	IF	ID	EX	MEM	WB	
DADD		IF	ID	EX	MEM	WB

Assume LD causes page fault, DADD has an overflow

Address page fault first

Overflow exception will occur again. Handle it then as it will be only exception

What if DADD causes Icache miss?

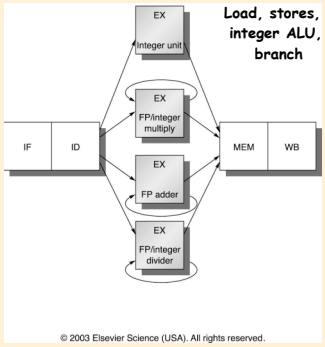
What do you do? You have to handle the page fault first?

Mark the eception in DADD; disable all state updates and deal with the exceptions in WB stages in instruction order (in order completion)

Instruction Set Complications?

- What is an instruction wrote in 2 stags instead of one? Eg: auto increment/decrement
- Eg: IA 32 - need hardware support if precise exceptions are required
- Multiple memory accesses -> need to hold the values and a mask register to tell what has been copied. Example memory block copy
- Conditional code or Program Status Word - nee to save them as part of the state
- Multicycle operations - complicate exceptions further - convert CISC instructions into internal microoperations and pipelinethem - so that all instructions take the same number of cycles -

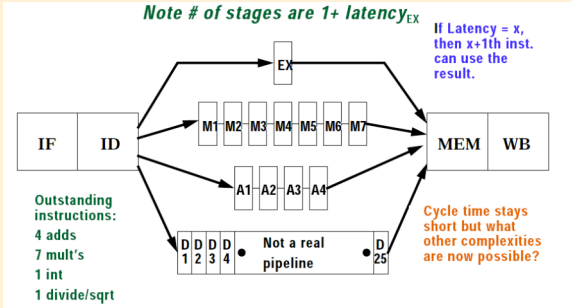
Multicycle Instructions



Load, stores, integer ALU, branch

	LATENCY	Initiation Interval
INTEGER ALU	0	1
LOAD	1	1
FP ADD	3	1
FP MULT	6	1
FP DIVIDE (non-pipelined)	24	25

FP PIPELINE



Hazard Classification

Instr(i) OPi Rdi, Rs1i, Rs2i

…….. … … … … . .

Instr (j) Opj Rdj, Rs1j, Rs2j

- Read After Write (RAW) :j tries to read source before i writes to it
- Write After Read (WAR): j writes the destination before I read it - in general it is not a problem if reads are done early (2nd stage) and writes are done late - however, it could be a problem if writes are done early and reads are done late or when instructions are re-ordered
- Write after Write (WAW) : Instr j tries to write an operand before it is written by instruction I
  - » This occurs if instructions WRITE in more than one stage
  - » Out of order execution and completion

What are the new Problems?

- Structural Hazards - eg: there is only one divider (these need to be detected and stalled)
- Varying completion times => conflicts on the register file write port - need interlock hardware
- Out-of-order completion
- Lots of data Hazards (RAW) as more instructions in flight
- WAW Hazards due to out-of-order completion

ADD F2, F4, F6

LD F2, 0(R2)

Can you have WAR Hazards? Why?

New Structural Hazards Sources

Instruction	1	2	3	4	5	6	7	8	9	10	11
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	
LD F8, 0(R2)							IF	ID	EX	MEM	WB

- Scan columns for common resource requirements
  - at cycle 10: 3 requirements for MEM
    - doesn't matter since MULTD and ADD don't do anything in this stage
  - at cycle 11: we have a problem however - ideas??

Increased RAW Hazards

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

How do you handle these problems?

- Issue stage becomes more complete. Before issuing Check the following and stall if necessary
- Check for structural hazards
- Check for RAW hazards and stall or forward = deeper pipelines means lots of checks and more complex forwarding paths
- Check for WAW hazards

Maintaining Precise Interrupts

DIV F0, F2, F4 --- 25 cycles

ADD F10, F19,F8 -- 4 cycles

SUB F12, F12, F14 -- 4 cycles

What's the problem?

Out-of-order completion - ADD and SUB may finish before divide finishes

Divide produces an exception after F12 is written by SUB

Can you have precise interrupts? No. You cannot restart SUB :-(

4 solutions

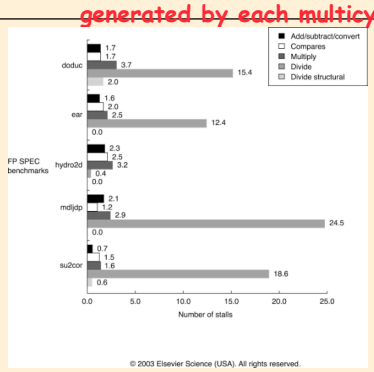
a) No Precise Interrupts or Precise/Imprecise modes - disallow some interrupts to be precise, or limit the overlap by allowing only one outstanding FP operation eg: 21064, 21164,Power1,2

Making Interrupts Precise

- Buffer the intermediate results with Result queue
  - History File : Keep original values, Bypassing becomes a pain
  - Future File: Keep new values in a buffer and update the Register file only after instructions before it have finished. In order completion
- Let the SW reconstruct the state by keeping the PCs of the instructions in flight; software can finish the preceding instructions
- Hybrid Schemes - allow an instruction to continue on if it is guaranteed that the preceeding instruction will complete with an exception.

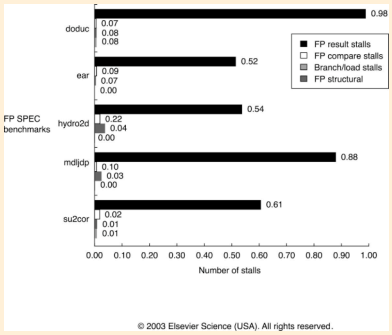
Eg: detect the div/zero exception early

Performance of FP Pipeline - stalls generated by each multicycle FP op



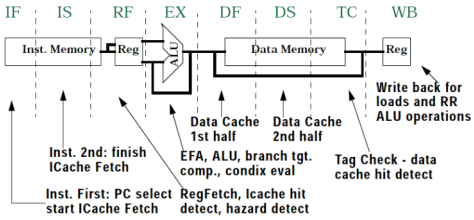
How would you Reduce these stalls?

MIPS Floating Point Performance - stalls/instructions



MIPS R4000 Pipeline

- Real 64-bit machine
  - ran between 100MHz and 200MHz
  - deeper pipelining (sometimes called *superpipelining*)
  - instruction set very similar to the MIPS64



Things to Notice

- **Load Delay is 2 cycles**
  - 2 instructions or bubbles between a load instruction and an instruction that uses the load value
  - At end of DS stage but before the tag check?
  - Yes, if the tag check fails, it is a cache miss and hence you need to stall
- **Branch Penalty is 3 cycles**
  - First cycle is a delay slot
  - The next two cycles are predict-not taken I.e. if not taken the delay is one otherwise it is 2, assuming the branch delay slot is filled with an useful instruction

MIPS R4000 Performance on SPEC92

Benchmark	Pipe CPI	Load Stalls	Branch Stalls	FP Res. Stalls	FP Struc. Stalls
compress	1.20	0.14	0.06	0.00	0.00
eqntott	1.88	0.27	0.61	0.00	0.00
espresso	1.42	0.07	0.35	0.00	0.00
gcc	1.56	0.13	0.43	0.00	0.00
li	1.64	0.18	0.46	0.00	0.00
INTEGER AVERAGE	1.54	0.16	0.39	0.00	0.00
doduc	2.84	0.01	0.22	1.39	0.22
mkljdp2	2.66	0.01	0.31	1.20	0.15
ear	2.17	0.00	0.46	0.59	0.12
hydro2d	2.53	0.00	0.62	0.75	0.17
su2cor	2.18	0.02	0.07	0.84	0.26
FP AVERAGE	2.48	0.01	0.33	0.95	0.18
OVERALL AVERAGE	2.00	0.10	0.36	0.46	0.09