Pipelining: Basic and Intermediate Concepts

Slides by: Muhamed Mudawar
CS 282 – KAUST
Spring 2010

Outline:

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural Hazards
- Data Hazards & Forwarding
- Branch Hazards
- Handling Exceptions
- Handling Multicycle Operations

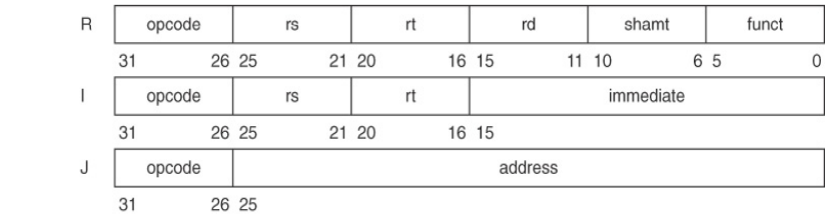
MIPS: Typical RISC ISA

- 32-bit fixed format instruction
- 32 GPR (R0 contains zero)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
- Simple branch conditions
- Delayed branch

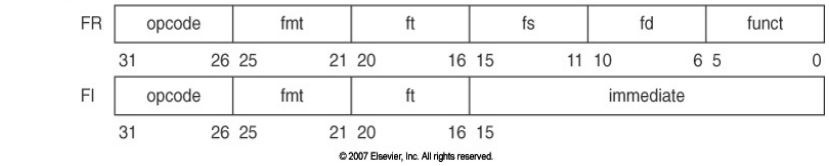
See Also: SPARC, IBM Power, and Itanium

Instruction Formats

Basic instruction formats

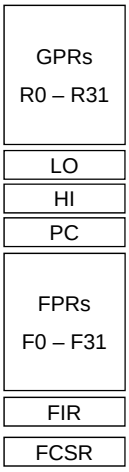


Floating-point instruction formats



Overview of the MIPS Registers

- 32 General Purpose Registers (GPRs)
  - 64-bit registers are used in MIPS64
  - Register 0 is always zero
  - Any value written to R0 is discarded
- Special-purpose registers LO and HI
  - Hold results of integer multiply and divide
- Special-purpose program counter PC
- 32 Floating Point Registers (FPRs)
  - 64-bit Floating Point registers in MIPS 64
- FIR: Floating-point Implementation Register
- FCSR: Floating-point Control & Status Register



Load and Store Instructions

Example instruction	Instruction name	Meaning
LD R1,30(R2)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[30+Regs[R2]]
LD R1,1000(R0)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[1000+0]
LW R1,60(R2)	Load word	Regs[R1] $\leftarrow_{64}$ (Mem[60+Regs[R2]]$_0$)^{32} ## Mem[60+Regs[R2]]
LB R1,40(R3)	Load byte	Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{56} ## Mem[40+Regs[R3]]
LBU R1,40(R3)	Load byte unsigned	Regs[R1] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[R3]]
LH R1,40(R3)	Load half word	Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{48} ## Mem[40+Regs[R3]] ## Mem[41+Regs[R3]]
L.S F0,50(R3)	Load FP single	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R3]] ## 0^{32}
L.D F0,50(R2)	Load FP double	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R2]]
SD R3,500(R4)	Store double word	Mem[500+Regs[R4]] $\leftarrow_{64}$ Regs[R3]
SW R3,500(R4)	Store word	Mem[500+Regs[R4]] $\leftarrow_{32}$ Regs[R3]$_{32..63}$
S.S F0,40(R3)	Store FP single	Mem[40+Regs[R3]] $\leftarrow_{32}$ Regs[F0]$_{0..31}$
S.D F0,40(R3)	Store FP double	Mem[40+Regs[R3]] $\leftarrow_{64}$ Regs[F0]
SH R3,502(R2)	Store half	Mem[502+Regs[R2]] $\leftarrow_{16}$ Regs[R3]$_{48..63}$
SB R2,41(R3)	Store byte	Mem[41+Regs[R3]] $\leftarrow_{8}$ Regs[R2]$_{56..63}$

Figure B.23 The load and store instructions in MIPS. All use a single addressing mode and require that the memory value be aligned. Of course, both loads and stores are available for all the data types shown.

Arithmetic / Logical Instructions

Example instruction	Instruction name	Meaning
DADDU R1,R2,R3	Add unsigned	Regs[R1]←Regs[R2]+Regs[R3]
DADDIU R1,R2,#3	Add immediate unsigned	Regs[R1]←Regs[R2]+3
LUI R1,#42	Load upper immediate	Regs[R1]←0^{32}##42##0^{16}
DSLL R1,R2,#5	Shift left logical	Regs[R1]←Regs[R2]<<5
SLT R1,R2,R3	Set less than	if (Regs[R2]<Regs[R3]) Regs[R1]←1 else Regs[R1]←0

Figure B.24 Examples of arithmetic/logical instructions on MIPS, both with and without immediates.

Control Flow Instructions

Example instruction	Instruction name	Meaning
J name	Jump	$PC_{36..63}$←name
JAL name	Jump and link	Regs[R31]←PC+8; $PC_{36..63}$←name; ((PC+4)−2^{27}) ≤ name < ((PC+4)+2^{27})
JALR R2	Jump and link register	Regs[R31]←PC+8; PC←Regs[R2]
JR R3	Jump register	PC←Regs[R3]
BEQZ R4,name	Branch equal zero	if (Regs[R4]==0) PC←name; ((PC+4)−2^{17}) ≤ name < ((PC+4)+2^{17})
BNE R3,R4,name	Branch not equal zero	if (Regs[R3]!= Regs[R4]) PC←name; ((PC+4)−2^{17}) ≤ name < ((PC+4)+2^{17})
MOVZ R1,R2,R3	Conditional move if zero	if (Regs[R3]==0) Regs[R1]←Regs[R2]

Figure B.25 Typical control flow instructions in MIPS. All control instructions, except jumps to an address in a register, are PC-relative. Note that the branch distances are longer than the address field would suggest; since MIPS instructions are all 32 bits long, the byte branch address is multiplied by 4 to get a longer distance.

Data Transfer / Arithmetic / Logical

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR*
LB,LBU,SB	Load byte, load byte unsigned, store byte (to/from integer registers)
LH,LHU,SH	Load half word, load half word unsigned, store half word (to/from integer registers)
LW,LWU,SW	Load word, load word unsigned, store word (to/from integer registers)
LD,SD	Load double word, store double word (to/from integer registers)
L.S,L.D,S.S,S.D	Load SP float, load DP float, store SP float, store DP float
MFC0,MTC0	Copy from/to GPR to/from a special register
MOV.S,MOV.D	Copy one SP or DP FP register to another FP register
MFC1,MTC1	Copy 32 bits to/from FP registers from/to integer registers
Arithmetic/logical	*Operations on integer or logical data in GPRs; signed arithmetic trap on overflow*
DADD,DADDI,DADDU,DADDIU	Add, add immediate (all immediates are 16 bits); signed and unsigned
DSUB,DSUBU	Subtract; signed and unsigned
DMUL,DMULU,DDIV, DDIVU,MADD	Multiply and divide, signed and unsigned; multiply-add; all operations take and yield 64-bit values
AND,ANDI	And, and immediate
OR,ORI,XOR,XORI	Or, or immediate, exclusive or, exclusive or immediate
LUI	Load upper immediate; loads bits 32 to 47 of register with immediate, then sign-extends
DSLL,DSRL,DSRA,DSLLV, DSRLV,DSRAV	Shifts: both immediate (DS__) and variable form (DS__V); shifts are shift left logical, right logical, right arithmetic
SLT,SLTI,SLTU,SLTIU	Set less than, set less than immediate; signed and unsigned

Control and Floating Point

Control	*Conditional branches and jumps; PC-relative or through register*
BEQZ,BNEZ	Branch GPRs equal/not equal to zero; 16-bit offset from PC + 4
BEQ,BNE	Branch GPR equal/not equal; 16-bit offset from PC + 4
BC1T,BC1F	Test comparison bit in the FP status register and branch; 16-bit offset from PC + 4
MOVN,MOVZ	Copy GPR to another GPR if third GPR is negative, zero
J,JR	Jumps: 26-bit offset from PC + 4 (J) or target in register (JR)
JAL,JALR	Jump and link: save PC + 4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
ERET	Return to user code from an exception; restore user mode
Floating point	*FP operations on DP and SP formats*
ADD.D,ADD.S,ADD.PS	Add DP, SP numbers, and pairs of SP numbers
SUB.D,SUB.S,SUB.PS	Subtract DP, SP numbers, and pairs of SP numbers
MUL.D,MUL.S,MUL.PS	Multiply DP, SP floating point, and pairs of SP numbers
MADD.D,MADD.S,MADD.PS	Multiply-add DP, SP numbers and pairs of SP numbers
DIV.D,DIV.S,DIV.PS	Divide DP, SP floating point, and pairs of SP numbers
CVT._._	Convert instructions: CVT.x.y converts from type x to type y, where x and y are L (64-bit integer), W (32-bit integer), D (DP), or S (SP). Both operands are FPRs.
C.__.D,C.__.S	DP and SP compares: "__" = LT,GT,LE,GE,EQ,NE; sets bit in FP status register

Figure B.26 Subset of the instructions in MIPS64. Figure B.22 lists the formats of these instructions. SP = single precision; DP = double precision. This list can also be found on the back inside cover.

Instruction Mix for SPECint2000

Instruction	gap	gcc	gzip	mcf	perlbmk	Integer average
load	26.5%	25.1%	20.1%	30.3%	28.7%	26%
store	10.3%	13.2%	5.1%	4.3%	16.2%	10%
add	21.1%	19.0%	26.9%	10.1%	16.7%	19%
sub	1.7%	2.2%	5.1%	3.7%	2.5%	3%
mul	1.4%	0.1%				0%
compare	2.8%	6.1%	6.6%	6.3%	3.8%	5%
load imm	4.8%	2.5%	1.5%	0.1%	1.7%	2%
cond branch	9.3%	12.1%	11.0%	17.5%	10.9%	12%
cond move	0.4%	0.6%	1.1%	0.1%	1.9%	1%
jump	0.8%	0.7%	0.8%	0.7%	1.7%	1%
call	1.6%	0.6%	0.4%	3.2%	1.1%	1%
return	1.6%	0.6%	0.4%	3.2%	1.1%	1%
shift	3.8%	1.1%	2.1%	1.1%	0.5%	2%
and	4.3%	4.6%	9.4%	0.2%	1.2%	4%
or	7.9%	8.5%	4.8%	17.6%	8.7%	9%
xor	1.8%	2.1%	4.4%	1.5%	2.8%	3%
other logical	0.1%	0.4%	0.1%	0.1%	0.3%	0%

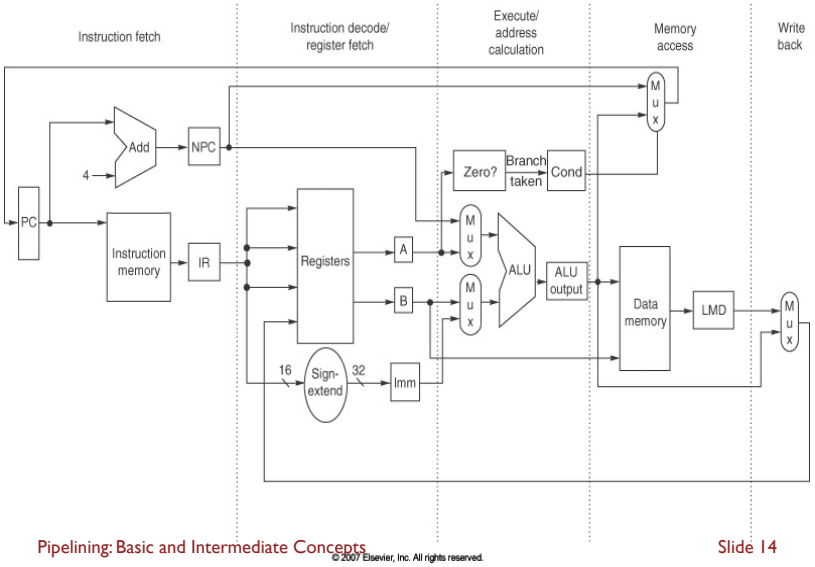
Instruction Mix for SPECfp2000

Instruction	applu	art	equake	lucas	swim	FP average
load	13.8%	18.1%	22.3%	10.6%	9.1%	15%
store	2.9%		0.8%	3.4%	1.3%	2%
add	30.4%	30.1%	17.4%	11.1%	24.4%	23%
sub	2.5%		0.1%	2.1%	3.8%	2%
mul	2.3%			1.2%		1%
compare		7.4%	2.1%			2%
load imm	13.7%		1.0%	1.8%	9.4%	5%
cond branch	2.5%	11.5%	2.9%	0.6%	1.3%	4%
cond mov		0.3%	0.1%			0%
jump			0.1%			0%
call			0.7%			0%
return			0.7%			0%
shift	0.7%		0.2%	1.9%		1%
and			0.2%	1.8%		0%
or	0.8%	1.1%	2.3%	1.0%	7.2%	2%
xor		3.2%	0.1%			1%
other logical			0.1%			0%
load FP	11.4%	12.0%	19.7%	16.2%	16.8%	15%
store FP	4.2%	4.5%	2.7%	18.2%	5.0%	7%
add FP	2.3%	4.5%	9.8%	8.2%	9.0%	7%
sub FP	2.9%		1.3%	7.6%	4.7%	3%
mul FP	8.6%	4.1%	12.9%	9.4%	6.9%	8%
div FP	0.3%	0.6%	0.5%		0.3%	0%
mov reg-reg FP	0.7%	0.9%	1.2%	1.8%	0.9%	1%
compare FP		0.9%	0.6%	0.8%		0%
cond mov FP		0.6%		0.8%		0%
other FP				1.6%		0%

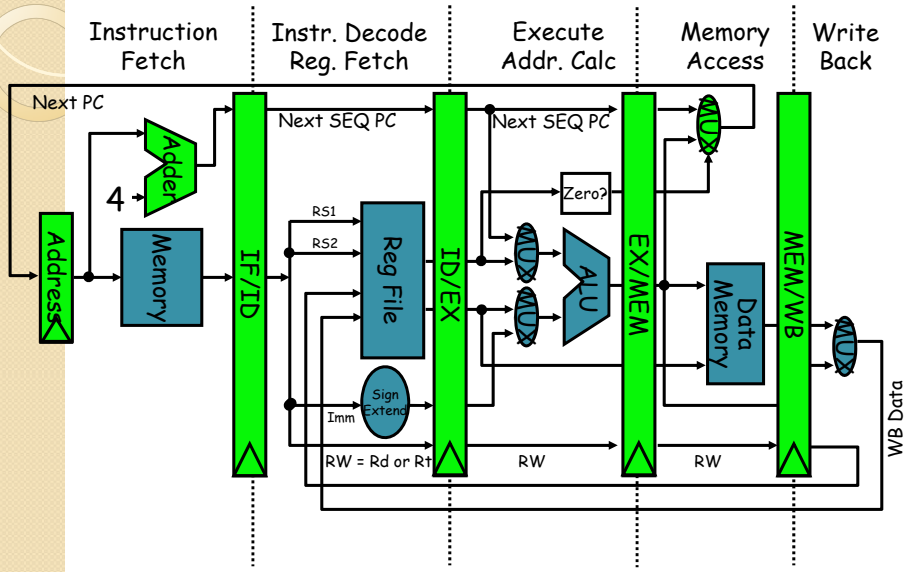
Next:

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural Hazards
- Data Hazards & Forwarding
- Branch Hazards
- Handling Exceptions
- Handling Multicycle Operations

5 Steps of Instruction Execution



Pipelined MIPS Datapath & Stage Registers

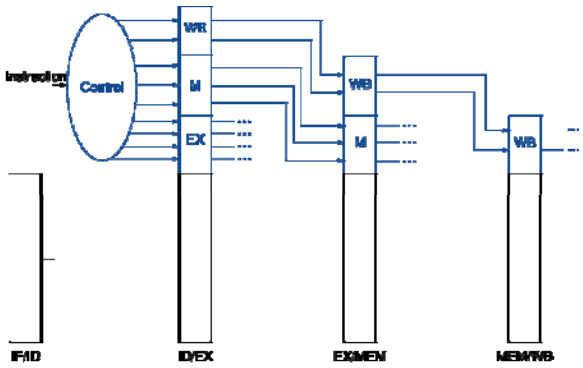


Events on Every Pipe Stage

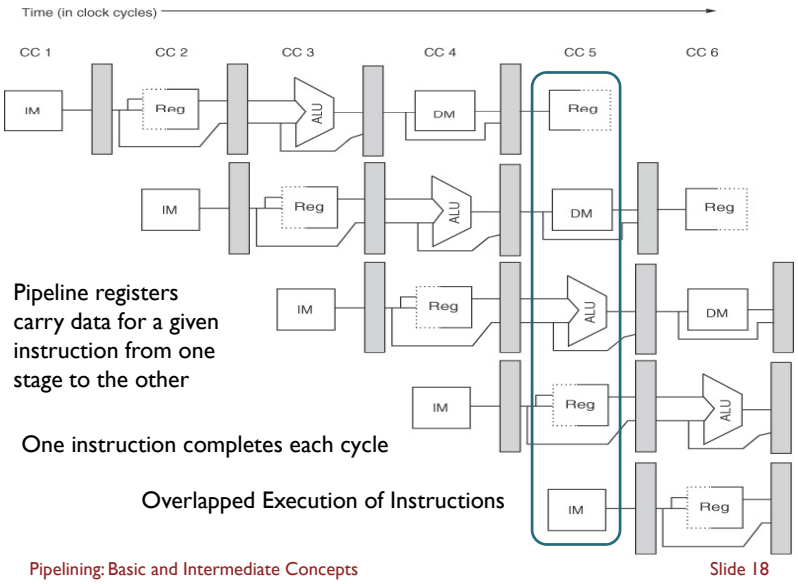
Stage	Any Instruction		
IF	IF/ID.IR ← MEM[PC]; IF/ID.NPC ← PC+4 PC ← if ((EX/MEM.opcode=branch) & EX/MEM.cond) {EX/MEM.ALUoutput} else {PC + 4}		
ID	ID/EX.A ← Regs[IF/ID.IR[Rs]]; ID/EX.B ← Regs[IF/ID.IR[Rt]] ID/EX.NPC ← IF/ID.NPC; ID/EX.Imm ← extend(IF/ID.IR[Imm]); ID/EX.Rw ← IF/ID.IR[Rt or Rd]		
	ALU Instruction	Load / Store	Branch
EX	EX/MEM.ALUoutput ← ID/EX.A func ID/EX.B, or EX/MEM.ALUoutput ← ID/EX.A op ID/EX.Imm	EX/MEM.ALUoutput ← ID/EX.A + ID/EX.Imm EX/MEM.B ← ID/EX.B	EX/MEM.ALUoutput ← ID/EX.NPC + (ID/EX.Imm << 2) EX/MEM.cond ← br condition
MEM	MEM/WB.ALUoutput ← EX/MEM.ALUoutput	MEM/WB.LMD ← MEM[EX/MEM.ALUoutput] or MEM[EX/MEM.ALUoutput] ← EX/MEM.B	
WB	Regs[MEM/WB.Rw] ← MEM/WB.ALUOutput	For load only: Regs[MEM/WB.Rw] ← MEM/WB.LMD	

Pipelined Control

- Control signals derived from instruction opcode
- Control signals are pipelined just like data



Visualizing Pipelining



Pipeline registers carry data for a given instruction from one stage to the other

One instruction completes each cycle

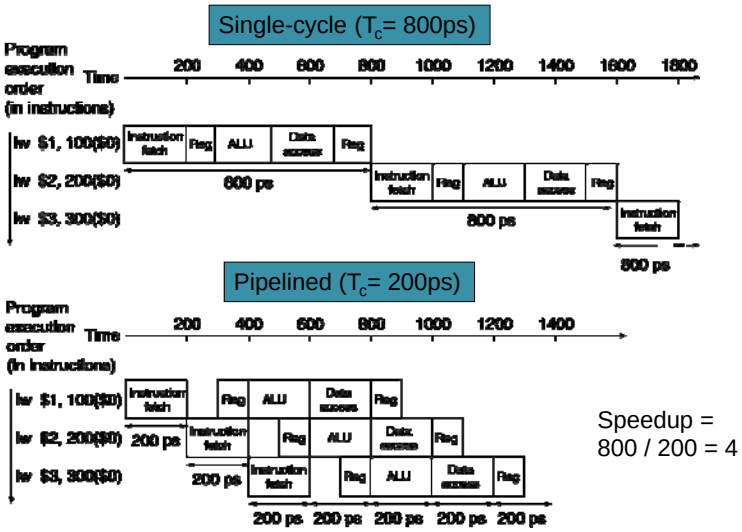
Overlapped Execution of Instructions

Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined versus non-pipelined datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
load	200ps	100 ps	200ps	200ps	100 ps	800ps
store	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
branch	200ps	100 ps	200ps			500ps

Pipeline Performance



Pipeline Speedup

- If all stages are balanced
  - All stages take the same time
  - Time between instructions$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - Compare with Intel x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Calculate address in 3rd stage, access memory in 4th
  - Alignment of memory operands
    - Memory access takes only one cycle

Pipelining is not quite that easy!

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot allow two instructions to use same resource during same cycle
  - 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)

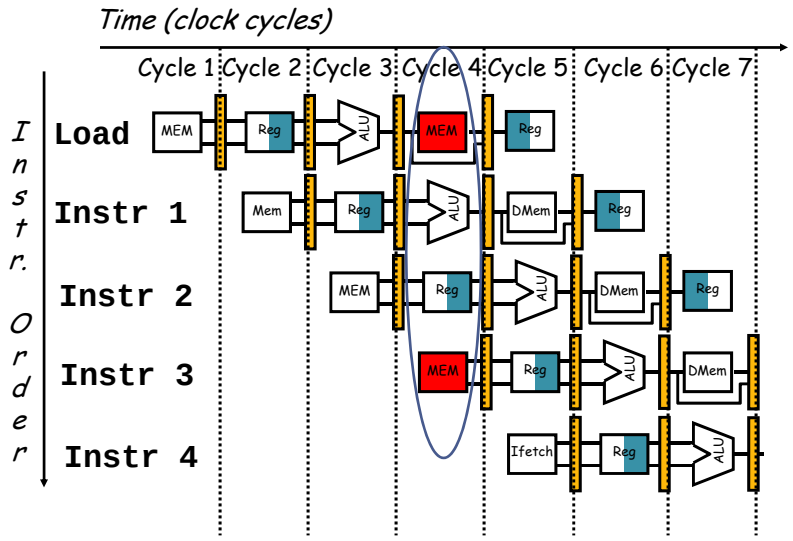
Next:

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural Hazards
- Data Hazards & Forwarding
- Branch Hazards
- Handling Exceptions
- Handling Multicycle Operations

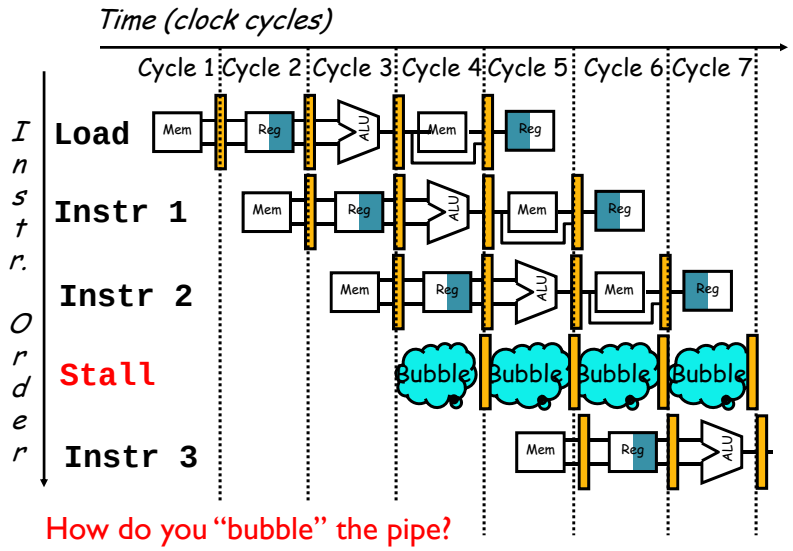
Structure Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for a cycle
    - Causes a pipeline "bubble"
- Hence, pipelined datapaths require separate Instruction and Data memories
  - Or separate Instruction and Data caches

One Memory Port/Structural Hazards



One Memory Port/Structural Hazards



How do you "bubble" the pipe?

Resolving Structural Hazards

- Serious Hazard:
  - Hazard cannot be ignored
- Solution 1: Delay Access to Resource
  - Must have mechanism to delay access to resource
  - Stall the pipeline until resource is available
- Solution 2: Add more hardware resources
  - Add more hardware to eliminate the structural hazard
  - Redesign the memory to have two ports
  - Or have two memories, each with a single port
    - One memory for instructions and the second for data
    - Harvard Architecture

Speedup Equation for Pipelining

$$\text{Speedup} = \frac{\text{CPI}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

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

For simple single-issue pipeline, Ideal CPI = 1

$$\text{Speedup} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

If stages are balances, $\text{Cycle}_{\text{unpipelined}}/\text{Cycle}_{\text{pipelined}}$ = Pipeline Depth

$$\text{Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

Example: Dual-port vs. Single-port Memory

- Machine A: Two memories ("Harvard Architecture")
- Machine B: Single ported memory, but it is pipelined
- B has a clock rate 1.05 times faster than clock rate of A
- Ideal pipelined CPI = 1 for both
- Loads are 40% of instructions executed

Stall cycles per instruction due to structural hazards

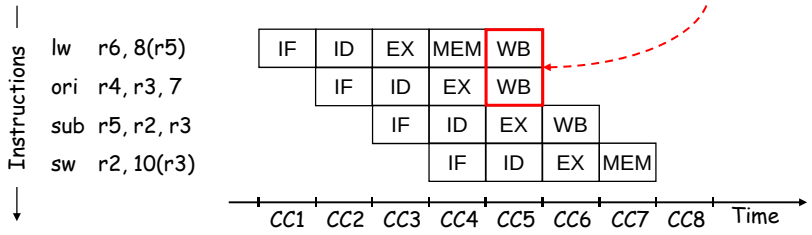
$$\text{Speedup}_{A/B} = \frac{\text{CPI}_B}{\text{CPI}_A} \times \frac{\text{Clock rate}_A}{\text{Clock rate}_B} = \frac{1 + 0.4}{1} \times \frac{1}{1.05} = 1.33$$

- Machine A is 1.33 times faster than B

Writing ALU result in Stage 4

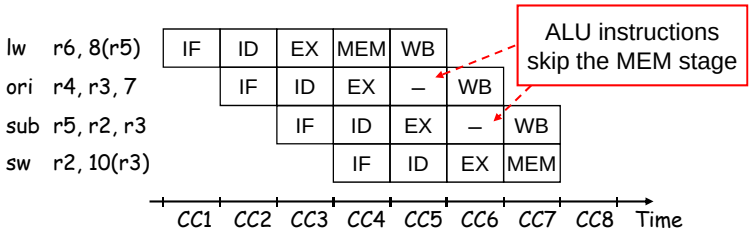
- Problem
  - Writing back ALU result in stage 4
  - Conflict with writing load data in stage 5

Structural Hazard
Two instructions are attempting to write the register file during same cycle



Resolving Write-Back Structure Hazard

- Solution 1
  - Add a second write port (costly solution)
  - Can do two writes during same cycle
- Solution 2 (better for single-issue pipeline)
  - Delay all write backs to the register file to stage 5
  - ALU instructions bypass stage 4 doing nothing



ALU instructions skip the MEM stage

Diagram shows instruction use of stages at each clock cycle

Next:

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural Hazards
- Data Hazards & Forwarding
- Branch Hazards
- Handling Exceptions
- Handling Multicycle Operations

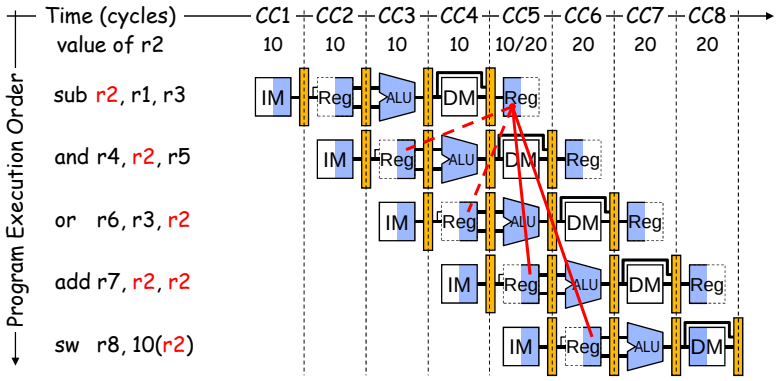
Data Hazards

- Data Dependence can cause a data hazard
- The dependent instructions are close to each other
  - Pipelined execution changes the order of operand access
- Read After Write – RAW Hazard
  - Given two instructions *I* and *J*, where *I* comes before *J* …
  - Instruction *J* should read an operand after it is written by *I*
  - Called a data dependence in compiler terminology

```
I: add r1, r2, r3     # r1 is written
J: sub r4, r1, r3     # r1 is read
```

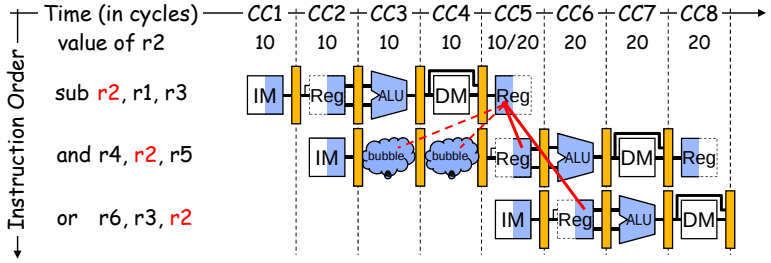
  - Hazard occurs when *J* reads the operand before *I* writes it

Example of a RAW Data Hazard



- Result of sub is needed by and, or, add, & sw instructions
- Instructions and & or will read old value of r2 from reg file
- During CC5, r2 is written and read – new value is read

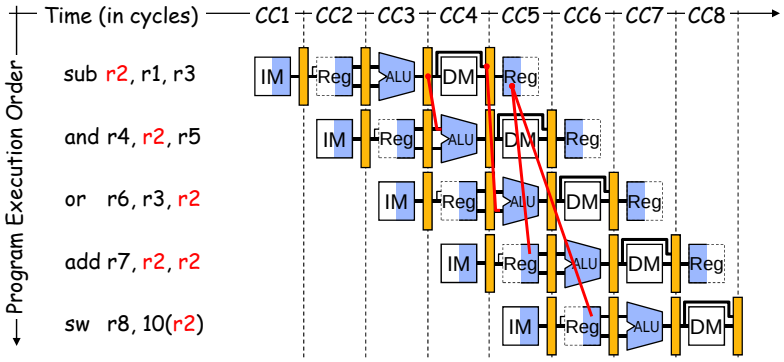
Solution 1: Stalling the Pipeline



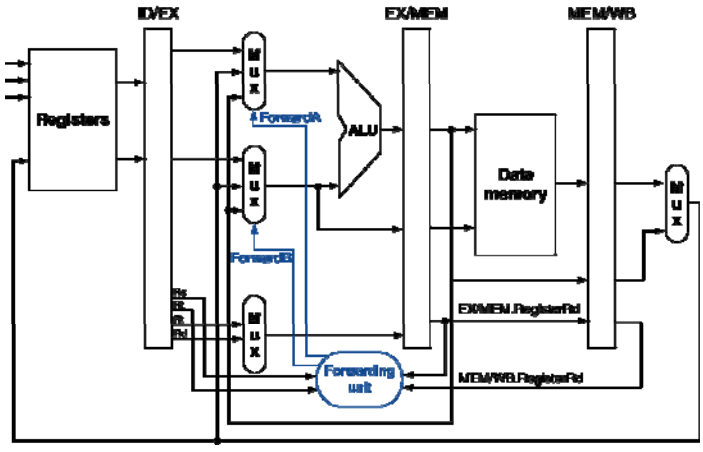
- The and instruction cannot fetch r2 until CC5
  - The and instruction remains in the IF/ID register until CC5
- Two bubbles are inserted into ID/EX at end of CC3 & CC4
  - Bubbles are NOP instructions: do not modify registers or memory
  - Bubbles delay instruction execution and waste clock cycles

Solution 2: Forwarding ALU Result

- The ALU result is forwarded (fed back) to the ALU input
  - No bubbles are inserted into the pipeline and no cycles are wasted
- ALU result exists in either EX/MEM or MEM/WB register



Hardware Support for Forwarding



Detecting RAW Hazards

- Pass register numbers along pipeline
  - ID/EX.RegisterRs = register number for Rs in ID/EX
  - ID/EX.RegisterRt = register number for Rt in ID/EX
  - ID/EX.RegisterRd = register number for Rd in ID/EX
- Current instruction being executed in ID/EX register
- Previous instruction is in the EX/MEM register
- Second previous is in the MEM/WB register
- RAW Data hazards when

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
} Fwd from EX/MEM pipeline reg

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
} Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not R0
  - EX/MEM.RegisterRd ≠ 0
  - MEM/WB.RegisterRd ≠ 0

Forwarding Conditions

- Detecting RAW hazard with Previous Instruction
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01 (Forward from EX/MEM pipe stage)
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01 (Forward from EX/MEM pipe stage)
- Detecting RAW hazard with Second Previous
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10 (Forward from MEM/WB pipe stage)
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10 (Forward from MEM/WB pipe stage)

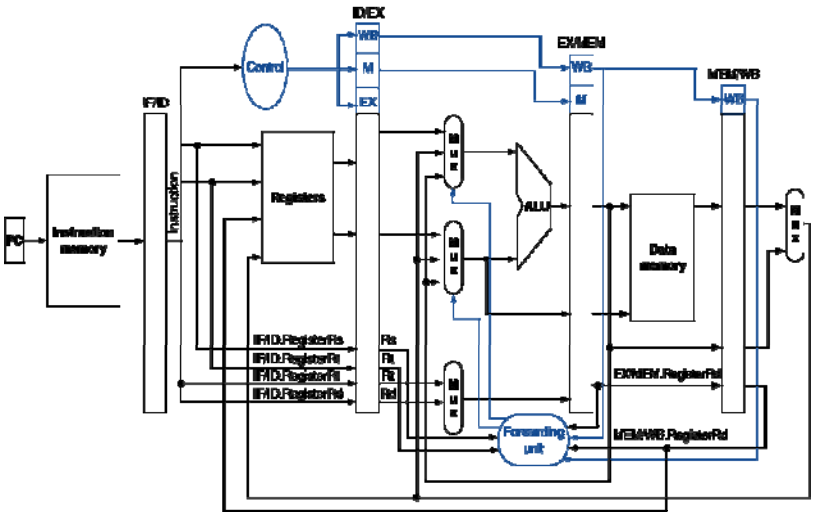
Double Data Hazard

- Consider the sequence:

```
add r1,r1,r2
sub r1,r1,r3
and r1,r1,r4
```

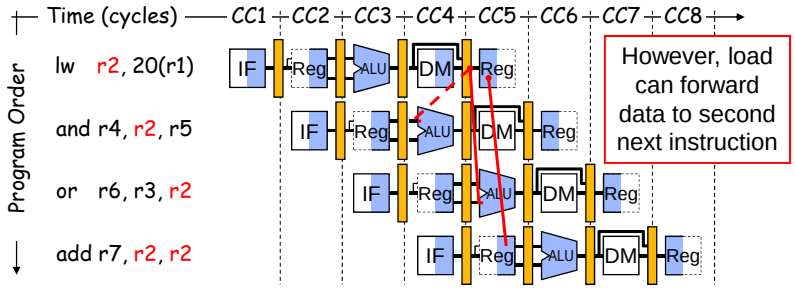
- Both hazards occur
  - Want to use the most recent
  - When executing AND, forward result of SUB
    - ForwardA = 01 (from the EX/MEM pipe stage)

Datapath with Forwarding



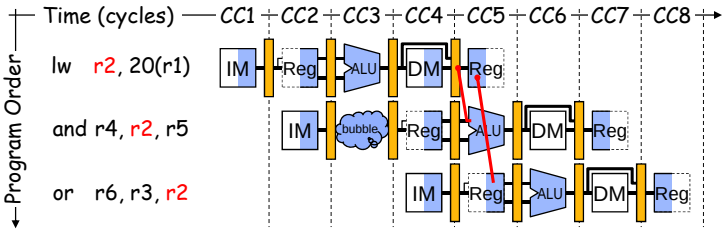
Load Delay

- Not all RAW data hazards can be forwarded
  - Load has a delay that cannot be eliminated by forwarding
- In the example shown below …
  - The LW instruction does not have data until end of CC4
  - AND wants data at beginning of CC4 - NOT possible



Stall the Pipeline for one Cycle

- Freeze the PC and the IF/ID registers
  - No new instruction is fetched and instruction after load is stalled
- Allow the Load in ID/EX register to proceed
- Introduce a bubble into the ID/EX register
- Load can forward data after stalling next instruction



Compiler Scheduling

- Compilers can schedule code in a way to avoid load stalls
- Consider the following statements:

  a = b + c; d = e – f;

- Slow code: 2 stall cycles

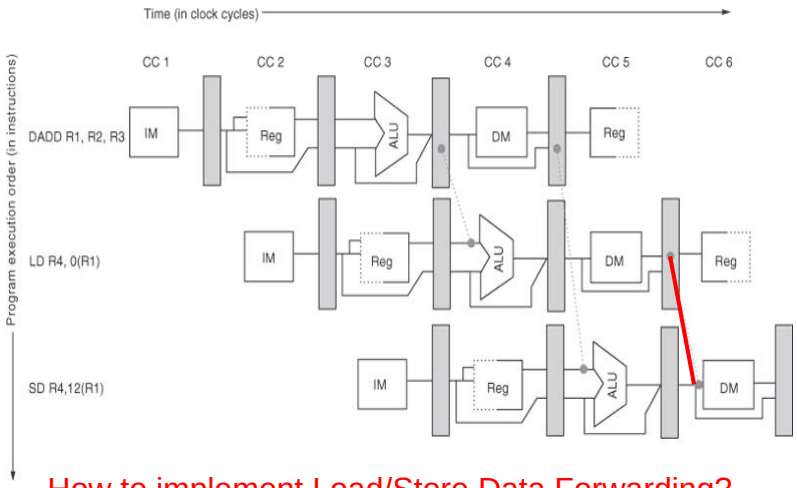
```
lw   r10, (r1)      # r1 = addr b
lw   r11, (r2)      # r2 = addr c
add  r12, r10, $11  # stall
sw   r12, (r3)      # r3 = addr a
lw   r13, (r4)      # r4 = addr e
lw   r14, (r5)      # r5 = addr f
sub  r15, r13, r14  # stall
sw   r15, (r6)      # r6 = addr d
```

Fast code: No Stalls

```
lw   r10, 0(r1)
lw   r11, 0(r2)
lw   r13, 0(r4)
lw   r14, 0(r5)
add  r12, r10, r11
sw   r12, 0(r3)
sub  r15, r13, r14
sw   r14, 0(r6)
```

Compiler optimizes for performance. Hardware checks for safety.

Load/Store Data Forwarding



How to implement Load/Store Data Forwarding?

Write After Read

- Instr **J** should write its result after it is read by **I**
- Called an anti-dependence by compiler writers

```
I: sub r4, r1, r3 # r1 is read
J: add r1, r2, r3 # r1 is written
```

- Results from reuse of the name r1
- Hazard occurs when **J** writes r1 before **I** reads it
- Cannot occur in the basic 5-stage pipeline because:
  - Reads are always in stage 2, and
  - Writes are always in stage 5
  - Instructions are processed in order

Write After Write

- Inst **J** should write its result after **I**
- Called output-dependence in compiler terminology

```
I: sub r1, r4, r3 # r1 is written
J: add r1, r2, r3 # r1 is written again
```

- This hazard also results from the reuse of name r1
- Hazard when writes occur in the wrong order
- Can't happen in our basic 5-stage pipeline because:
  - All writes are ordered and take place in stage 5
- WAR and WAW hazards occur in complex pipelines
- Notice that Read After Read – RAR is NOT a hazard

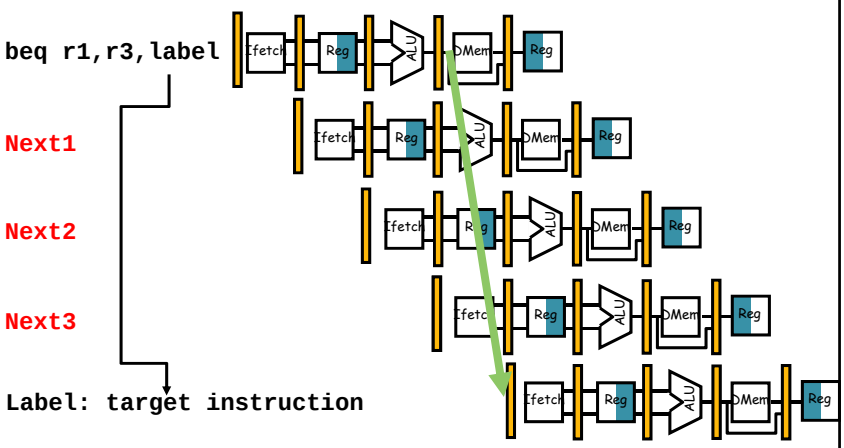
Next:

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural Hazards
- Data Hazards & Forwarding
- Branch Hazards
- Handling Exceptions
- Handling Multicycle Operations

Control Hazards

- Branch instructions can cause great performance loss
- Branch instructions need two things:
  - Branch Result — Taken or Not Taken
  - Branch Target
    - PC + 4 — If Branch is NOT taken
    - PC + 4 + 4 × imm — If Branch is Taken
- For our pipeline: 3-cycle branch delay
  - PC is updated 3 cycles after fetching branch instruction
  - Branch target address is calculated in the ALU stage
  - Branch result is also computed in the ALU stage
  - What to do with the next 3 instructions after branch?

3-Cycle Branch Delay

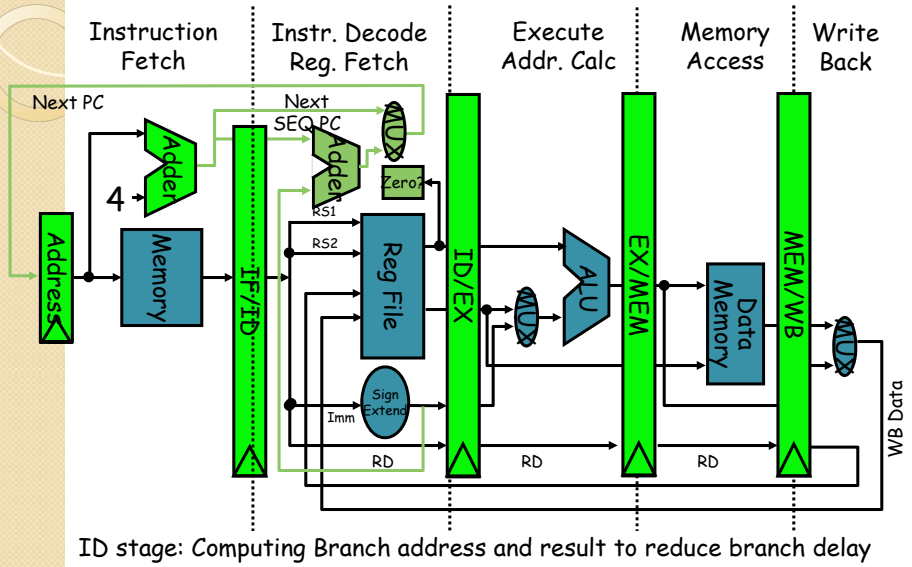


- Next1 thru Next3 instructions will be fetched anyway
- Pipeline should flush Next1 - Next3 if branch is taken

Branch Stall Impact

- If CPI = 1 without branch stalls, and 30% branch
- If stalling 3 cycles per branch
- => new CPI = 1+0.3×3 = 1.9
- Two part solution:
  - Determine branch taken or not sooner, and
  - Compute taken branch address earlier
- MIPS Solution:
  - Move branch test to ID stage (second stage)
  - Adder to calculate new PC in ID stage
  - Branch delay is reduced from 3 to just 1 clock cycle

Modified Pipelined MIPS Datapath



ID stage: Computing Branch address and result to reduce branch delay

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
- 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 until ID stage
  - MIPS still incurs 1 cycle branch penalty
  - Other machines: branch target known before branch outcome

Four Branch Hazard Alternatives

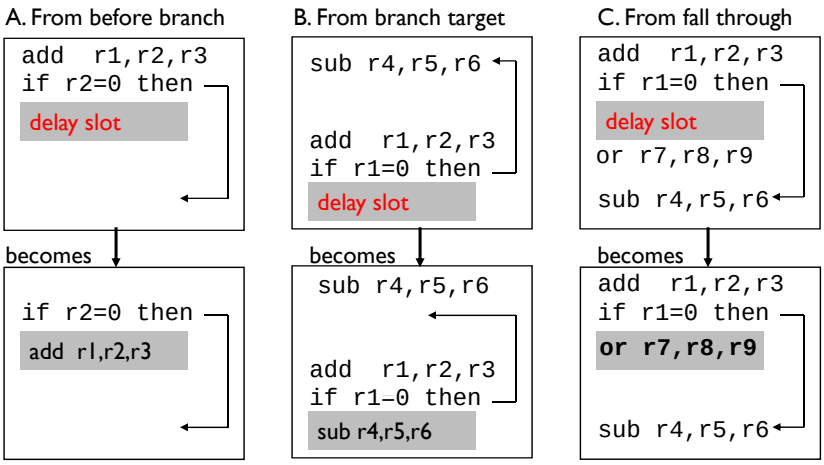
#4: Delayed Branch

- Define branch to take place AFTER following instruction

```
branch instruction
  sequential successor1
  sequential successor2
  ........                    } Branch delay of length n
  sequential successorn
branch target if taken
```

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

Scheduling Branch Delay Slots



- 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 & C, must be okay to execute instruction in delay slot in all cases

Effectiveness of 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

Performance of Branch Schemes

Assuming an ideal CPI = 1 without counting branch stalls

Pipeline speedup over non-pipelined datapath:

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch Frequency} \times \text{Branch Penalty}}$$

$$\text{Pipeline CPI}_{\text{branch stalls}} = \text{Ideal CPI}_{\text{no stalls}} + \text{Branch Freq} \times \text{Branch Penalty}$$

Evaluating Branch Alternatives

Branch Scheme	Penalty Unconditional	Penalty Untaken	Penalty Taken
Stall always	2	3	3
Predict taken	2	3	2
Predict not taken	2	0	3
Delayed branch	1	0	2

Assume 4% unconditional branch, 6% conditional branch- untaken, and 10% conditional branch-taken. What is the impact on the CPI?

Branch Scheme	Unconditional Branches	Untaken Branches	Taken Branches	All Branches
	4%	6%	10%	20%
Stall always	0.08	0.18	0.30	CPI+0.56
Predict taken	0.08	0.18	0.20	CPI+0.46
Predict not taken	0.08	0	0.30	CPI+0.38
Delayed branch	0.04	0	0.20	CPI+0.24

Next:

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural Hazards
- Data Hazards & Forwarding
- Branch Hazards
- Handling Exceptions
- Handling Multicycle Operations

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the execution of an instruction
    - e.g., undefined opcode, overflow, syscall, …
- Interrupt
  - An external I/O device controller is requesting processor
- Exceptions and Interrupts complicate the implementation and control of the pipeline

Types of Exceptions

- I/O device request (hardware interrupt)
- Invoking the OS (system call)
- Tracing instruction execution
- Breakpoint (programmer requested)
- Integer arithmetic overflow
- Floating Point arithmetic anomaly
- Page fault (requested page is not in memory)
- Misaligned memory access
- Memory protection violation
- Undefined instruction
- Hardware malfunction and Power failure

Handling Exceptions

- In MIPS, exceptions are managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
- Jump to handler at a fixed address

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If program can be restarted
  - Take corrective action
  - Use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, Cause, …

Alternative Approach

- Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - …: C000 0040
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

Exceptions in MIPS 5-stage pipeline

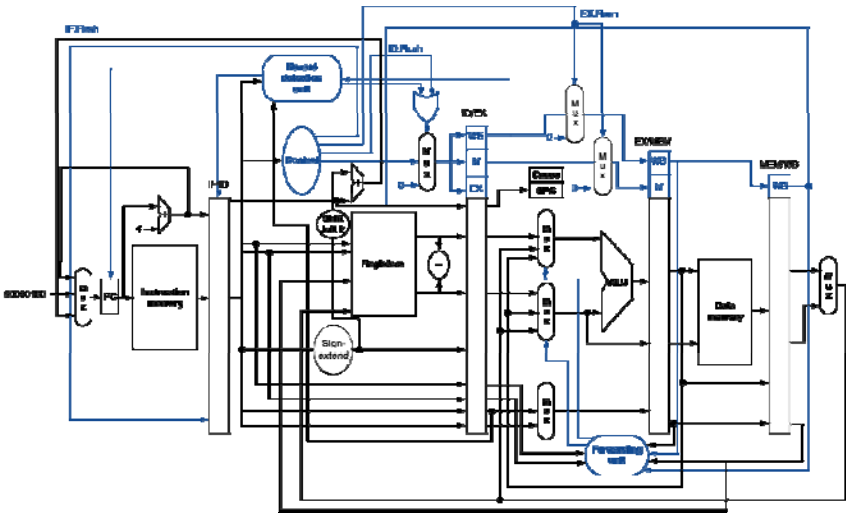
Stage	Exceptions that may occur
IF	Page fault on instruction fetch, misaligned memory access, memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch, misaligned memory access, memory protection violation
WB	None

Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage

  `add r1, r2, r1`
  - Prevent `r1` from being written
  - Complete previous instructions
  - Flush `add` and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

5-Stage Pipeline with Exceptions



Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - Precise exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is more difficult!

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require "manual" completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

Next:

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural Hazards
- Data Hazards & Forwarding
- Branch Hazards
- Handling Exceptions
- Handling Multicycle Operations

Pipeline with Multiple Functional Units