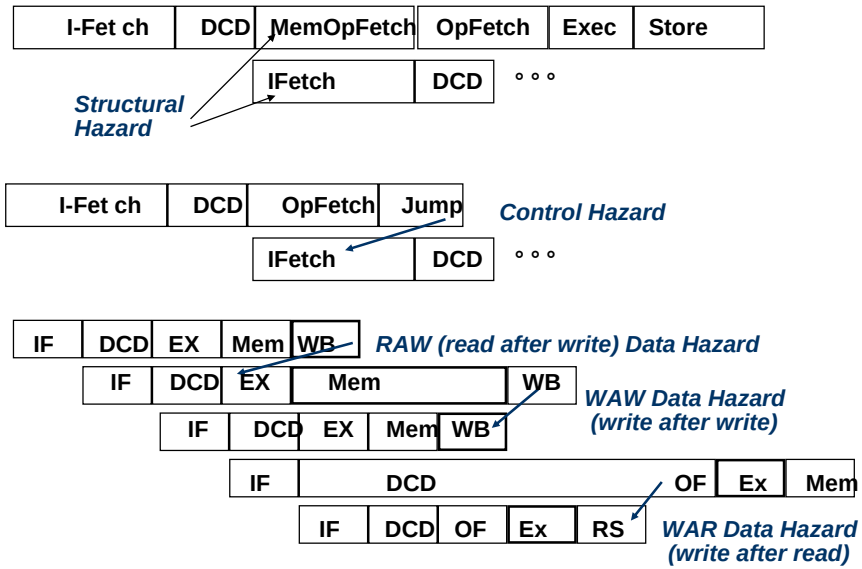
Computer System Architecture

Pipelining Part III

Chalermek Intanagonwiwat

Slides courtesy of David Patterson

Pipeline Hazards Again



Data Hazards

- Avoid some "by design"
  - eliminate WAR by always fetching operands early (DCD) in pipe
  - eleminate WAW by doing all WBs in order (last stage, static)
- Detect and resolve remaining ones
  - stall or forward (if possible)

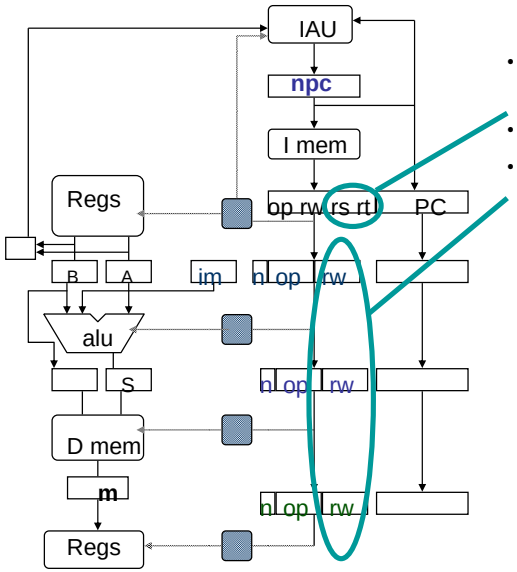
Hazard Detection

- Suppose instruction i is about to be issued and a predecessor instruction j is in the instruction pipeline.
- A RAW hazard exists on register ρ if $\rho \in$ Rregs(i) $\cap$ Wregs(j)
  - Keep a record of pending writes (for inst's in the pipe) and compare with operand regs of current instruction.
  - When instruction issues, reserve its result register.
  - When on operation completes, remove its write reservation.

Hazard Detection (cont.)

- A WAW hazard exists on register ρ if ρ ∈ Wregs(*i*) ∩ Wregs(*j*)
- A WAR hazard exists on register ρ if ρ ∈ Wregs(*i*) ∩ Rregs(*j*)

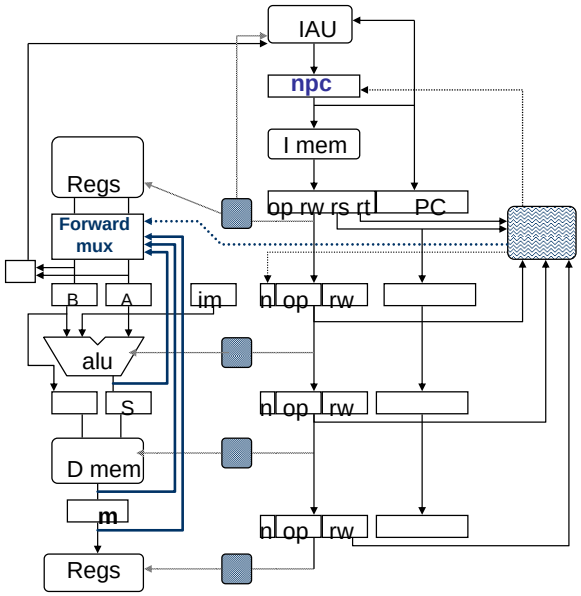
Record of Pending Writes



- Current operand registers
- Pending writes
- hazard <=

  ((rs == rw_{ex}) & $regW_{ex}$) OR
  ((rs == rw_{mem}) & $regW_{me}$) OR
  ((rs == rw_{wb}) & $regW_{wb}$) OR
  ((rt == rw_{ex}) & $regW_{ex}$) OR
  ((rt == rw_{mem}) & $regW_{me}$) OR
  ((rt == rw_{wb}) & $regW_{wb}$)

Resolve RAW by forwarding



- Detect nearest valid write op operand register and forward into op latches, bypassing remainder of the pipe
- Increase muxes to add paths from pipeline registers
- Data Forwarding = Data Bypassing

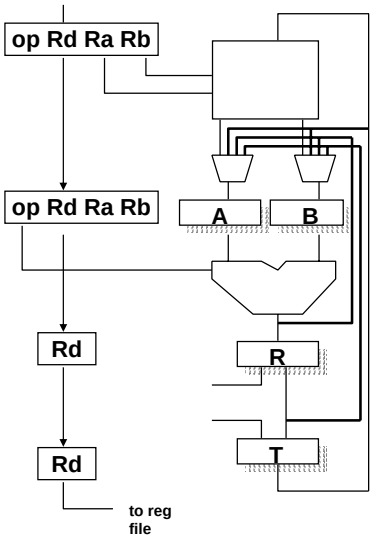
What about memory operations?

° If instructions are initiated in order and operations always occur in the same stage, there can be no hazards between memory operations!

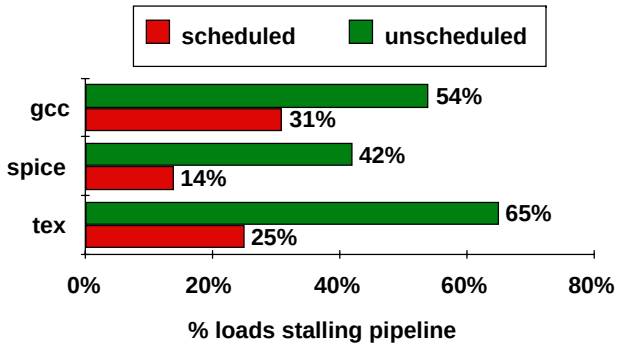
° What does delaying WB on arithmetic operations cost?
  - cycles ?
  - hardware ?

° What about data dependence on loads?
  R1 <- R4 + R5
  R2 <- Mem[R2 + I]
  R3 <- R2 + R1
=>

"Delayed Loads"



Compiler Avoiding Load Stalls:

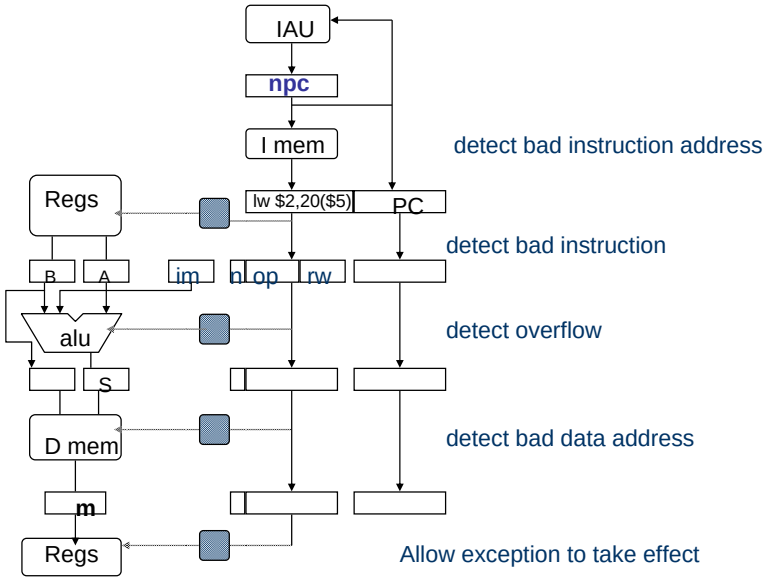


What about Interrupts, Traps, Faults?

- External Interrupts:
  - Allow pipeline to drain,
  - Load PC with interrupt address
- Faults (within instruction, restartable)
  - Force trap instruction into IF
  - disable writes till trap hits WB
  - must save multiple PCs or PC + state

Refer to MIPS solution

Exception Handling



Exception Problem

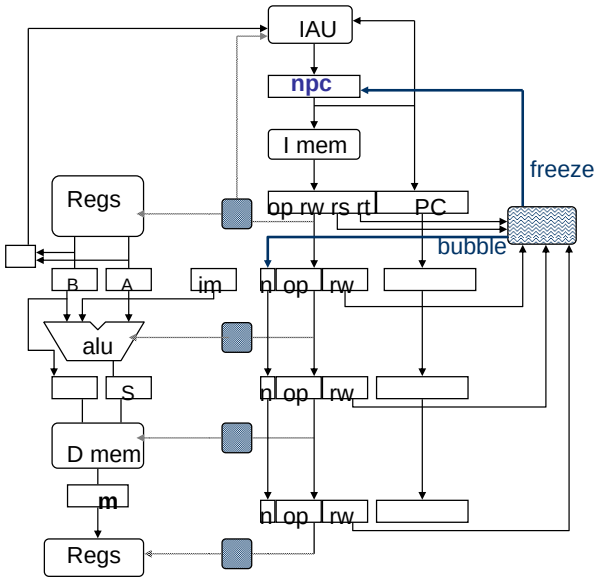
- Exceptions/Interrupts: 5 instructions executing in 5 stage pipeline
  - How to stop the pipeline?
  - Restart?
  - Who caused the interrupt?

Exception Problem (cont.)

Stage Problem interrupts occurring

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; memory error

Resolution: Freeze above & Bubble Below

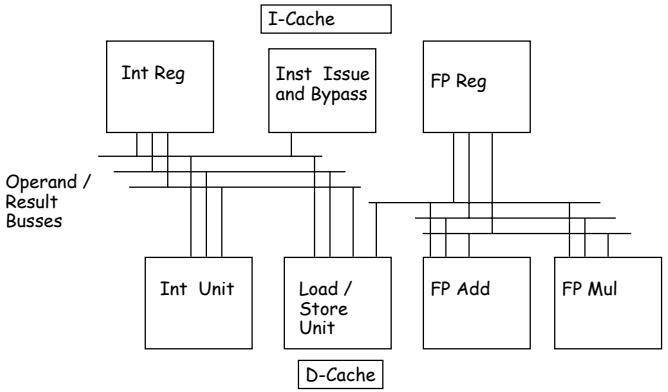


Summary

- Pipelines pass control information down the pipe just as data moves down pipe
- Forwarding/Stalls handled by local control
- Exceptions stop the pipeline
- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)
- More performance from deeper pipelines, parallelism

Partitioned Instruction Issue (simple Superscalar)

independent int and FP issue to separate pipelines



Single Issue Total Time = Int Time + FP Time

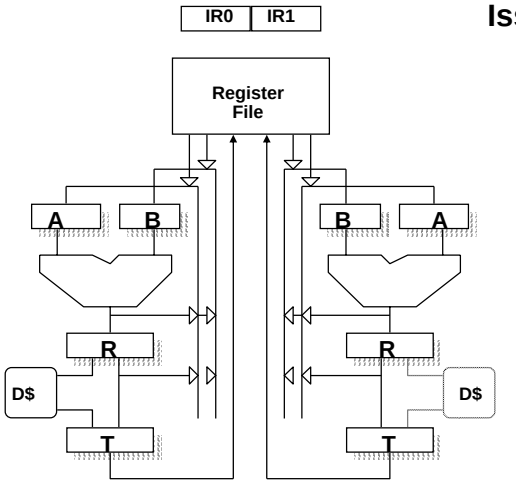
Max Speedup: $\frac{\text{Total Time}}{\text{MAX(Int Time, FP Time)}}$

Example

Basic Loop:	Cycles
load Ra <- Ai	1
load Ry <- Yi	1
fmult Rm <- Ra*Rx	7
faddRs <- Rm+Ry	5
store Ai <- Rs	1
inc Yi	1
dec i	1
inc Ai	1
branch	1

Total Single Issue Cycles: 19 (7 integer, 12 floating point)
Minimum with Dual Issue: 12
Potential Speedup: 1.6 !!!

Multiple Pipes/ Harder Superscalar



Issues:

Reg. File ports

Detecting Data Dependences

Bypassing

RAW Hazard

WAR Hazard

Multiple load/store ops?

Branches

Getting CPI < 1: Issuing Multiple Instructions/Cycle

Type	*PipeStages*
Int. instruction	IF ID EXMEMWB
FP instruction	IF ID EXMEMWB
Int. instruction	IF ID EXMEMWB
FP instruction	IF ID EXMEMWB
Int. instruction	IF ID EX MEM WB
FP instruction	IF ID EXMEMWB

Unrolled Loop that Minimizes Stalls for Scalar

```
1. Loop: LD F0, 0(R1)
2.       ADDD F4, F0, F2
3.       SD  0(R1), F4
4.       SUBI R1, R1, #8
5.       BNEZ R1, Loop
```

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

Delayed Branch

```
1 Loop: LD     F0,0(R1)
2       LD     F6,-8(R1)
3       LD     F10,-16(R1)
4       LD     F14,-24(R1)
5       ADDD   F4,F0,F2
6       ADDD   F8,F6,F2
7       ADDD   F12,F10,F2
8       ADDD   F16,F14,F2
9       SD     0(R1),F4
10      SD     -8(R1),F8
11      SD     -16(R1),F12
12      SUBI   R1,R1,#32
13      BNEZ   R1,LOOP
14      SD     8(R1),F16  ; 8-32 = -24
```

14 clock cycles, or 3.5 per iteration

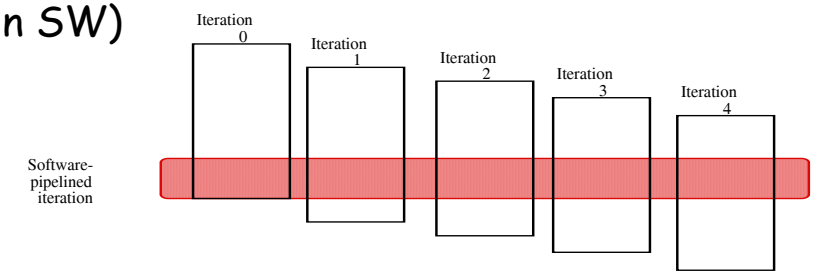
Loop Unrolling in Superscalar

	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	LD F0,0(R1)		1
	LD F6,-8(R1)		2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12		8
	SD -24(R1),F16		9
	SUBI R1,R1,#40		10
	BNEZ R1,LOOP		11
	SD -32(R1),F20		12

- Unrolled 5 times to avoid delays (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration

Software Pipelining

- Observation: if iterations from loops are independent, then can get ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (- Tomasulo in SW)



Software Pipelining Example

Before: Unrolled 3 times

```
1  LD   F0,0(R1)
2  ADDD F4,F0,F2
3  SD   0(R1),F4

4  LD   F6,-8(R1)
5  ADDD F8,F6,F2
6  SD   -8(R1),F8

7  LD   F10,-16(R1)
8  ADDD F12,F10,F2
9  SD   -16(R1),F12
10 SUBI R1,R1,#24
11 BNEZ R1,LOOP
```

After: Software Pipelined

```
1  SD   0(R1),F4 ; Stores M[i]
2  ADDD F4,F0,F2 ; Adds to M[i-1]
3  LD   F0,-16(R1); Loads M[i-2]
4  SUBI R1,R1,#8
5  BNEZ R1,LOOP
```

Limits of Superscalar

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards
- If more instructions issue at same time, greater difficulty of decode and issue
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue

HW Schemes: Instruction Parallelism

- Why in HW at run time?
  - Works when can't know real dependence at compile time
  - Compiler simpler
  - Code for one machine runs well on another
- Key idea: Allow instructions behind stall to proceed

```
DIVD  F0,F2,F4
ADDD  F10,F0,F8
SUBD  F12,F8,F14
```

  - Enables out-of-order execution => out-of-order completion
  - ID stage checked both for structural & data dependencies

HW Schemes: Instruction Parallelism (cont.)

- Out-of-order execution divides ID stage:
  1. Issue—decode instructions, check for structural hazards
  2. Read operands—wait until no data hazards, then read operands
- Scoreboards allow instruction to execute whenever 1 & 2 hold, not waiting for prior instructions
- CDC 6600: In order issue, out of order execution, out of order commit (also called completion)

Performance of Dynamic SS

Iteration no.	*Instructions*	*Issues*	*Executes*	*Writes result*
			clock-cycle number	
1	LD F0,0(R1)	1	2	4
1	ADDD F4,F0,F2	1	5	8
1	SD 0(R1),F4	2	9	
1	SUBI R1,R1,#8	3	4	5
1	BNEZ R1,LOOP	4	5	
2	LD F0,0(R1)	5	6	8
2	ADDD F4,F0,F2	5	9	12
2	SD 0(R1),F4	6	13	
2	SUBI R1,R1,#8	7	8	9
2	BNEZ R1,LOOP	8	9	

- 4 clocks per iteration

Branches, Decrements still take 1 clock cycle

HW support for More ILP

- *Speculation*: allow an instruction to issue that is dependent on branch predicted to be taken *without* any consequences (including exceptions) if branch is not actually taken ("HW undo")
- Often try to combine with dynamic scheduling
- Separate *speculative* bypassing of results from real bypassing of results
  - When instruction no longer speculative, write results (*instruction commit*)
  - execute out-of-order but commit in order

Dynamic Scheduling in Pentium Pro

- PPro doesn't pipeline 80x86 instructions
- PPro decode unit translates the Intel instructions into 72-bit micro-operations (- MIPS)
- Takes 1 clock cycle to determine length of 80x86 instructions + 2 more to create the micro-operations

Dynamic Scheduling in Pentium Pro (cont.)

- Most instructions translate to 1 to 4 micro-operations
- Complex 80x86 instructions are executed by a conventional microprogram (8K x 72 bits) that issues long sequences of micro-operations

Limits to Multi-Issue Machines

- Difficulties in building HW
  - Duplicate FUs to get parallel execution
  - Increase ports to Register File
  - Increase ports to memory

Summary

- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)
- More performance from deeper pipelines, parallelism
- Superscalar
  - CPI < 1
  - Dynamic issue vs. Static issue
  - More instructions issue at same time, larger the penalty of hazards

Summary (cont.)

- SW Pipelining
  - Symbolic Loop Unrolling to get most from pipeline with little code expansion, little overhead