CS252
Graduate Computer Architecture
Lecture 4

Control flow and interrupts (cont'd)
Software Scheduling around hazards

September 13, 2000
Prof. John Kubiatowicz

Review: Control Flow and Exceptions

- RISC vs CISC was about the integrated system's view, not about removing instructions:
  - These names were a bit unfortunate in retrospect, since they caused some "religious" arguments
  - RISC ⇒ intelligent hardware-software tradeoffs driven by quantitative measurement with real benchmarks
  - End-to-end view point
- Control flow is the biggest problem for computer architects. This is getting worse:
  - Modern computer languages such as C++ and Java user many smaller procedure calls (method invocations)
  - Networked devices need to respond quickly to many external events.

Review: A "zero-cycle" jump

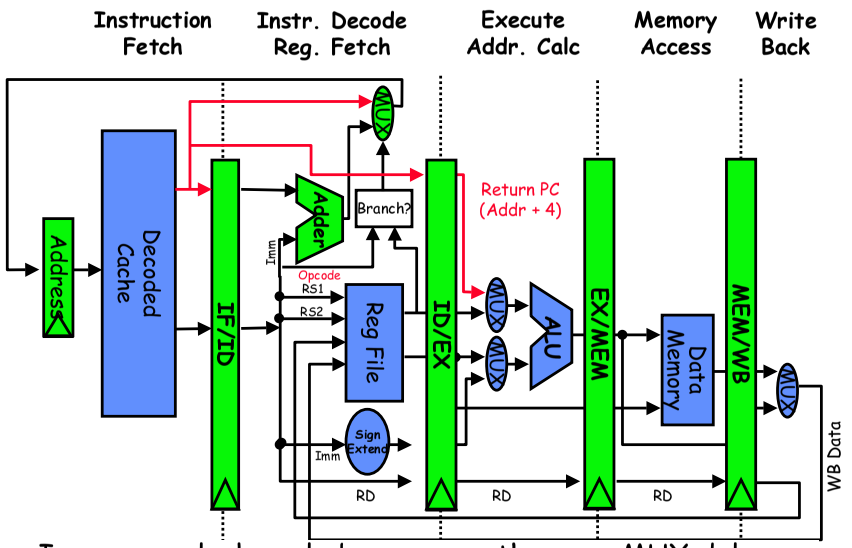
- What really has to be done at runtime?
  - Once an instruction has been detected as a jump or JAL, we might recode it in the internal cache.
  - Very limited form of dynamic compilation?

```
and   r3,r1,r5
addi  r2,r3,#4
sub   r4,r2,r1
jal   doit
subi   r1,r1,#1
```

Internal Cache state:

A:	and r3,r1,r5	N	A+4
	addi r2,r3,#4	N	A+8
	sub r4,r2,r1	L	doit
	---	--	---
	subi r1,r1,#1	N	A+20

- Use of "Pre-decoded" instruction cache
  - Called "branch folding" in the Bell-Labs CRISP processor.
  - Original CRISP cache had two addresses and could thus fold a complete branch into the previous instruction
  - Notice that JAL introduces a structural hazard on write



- Increases clock cycle by no more than one MUX delay
- Introduces structural hazard on write for JAL, however

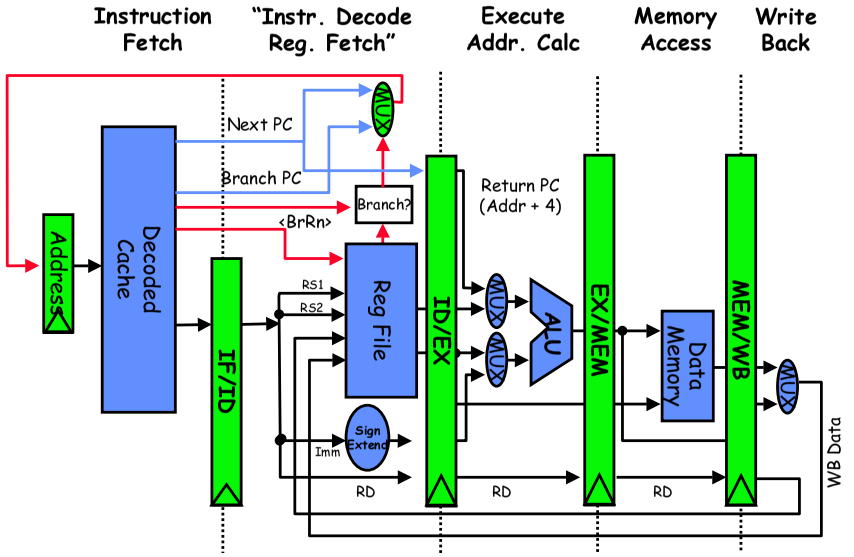
Why not do this for branches?
(original CRISP idea, applied to DLX)

```
and   r3,r1,r5
addi  r2,r3,#4
sub   r4,r2,r1
bne   r4,loop
subi  r1,r1,#1
```

Internal Cache state:

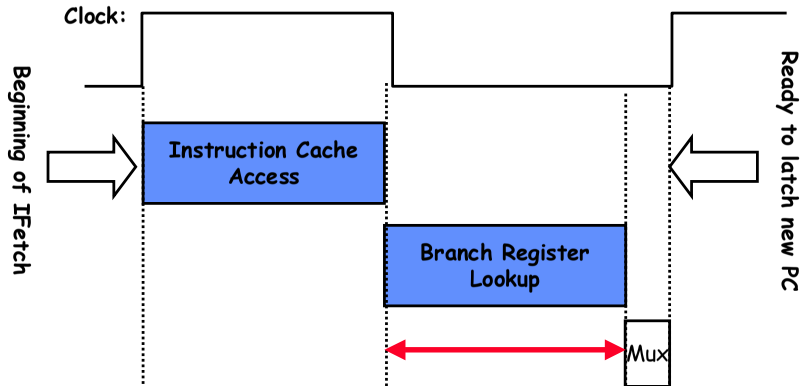
				Next	Branch
A:	and	r3,r1,r5	N	A+4	N/A
	addi	r2,r3,#4	N	A+8	N/A
	sub	r4,r2,r1	BnR4	A+16	loop
		---	--	---	---
A+16:	subi	r1,r1,#1	N	A+20	N/A

- Delay slot eliminated (good)
- Branch has been "folded" into sub instruction (good).
- Increases size of instruction cache (not so good)
- Requires another read port in register file (BAD)
- Potentially doubles clock period (Really BAD)



- Might double clock period -- must access cache and reg
- Could be better if had architecture with condition codes

Way of looking at timing:



Register file access time might be close to original clock period

However, one could use the first technique to reflect PREDICTIONS and remove delay slots

```
and   r3,r1,r5
addi  r2,r3,#4
sub   r4,r2,r1
bne   r4,loop
subi  r1,r1,#1
```

Internal Cache state:

				Next
A:	and	r3,r1,r5	N	A+4
	addi	r2,r3,#4	N	A+8
	sub	r4,r2,r1	N	A+12
	bne	loop	N	loop
A+16:	subi	r1,r1,#1	N	A+20

- This causes the next instruction to be immediately fetched from branch destination (predict taken)
- If branch ends up being not taking, then squash destination instruction and restart pipeline at address A+16

Book talks about R4000

(taken from page 204)

- On a *taken branch*, there is a one cycle delay slot, followed by two lost cycles (nullified insts).
  - *Recall from prereq quiz: delay slot is an instruction-set feature!*

	Clock Number								
Instruction	1	2	3	4	5	6	7	8	9
Branch inst	IF	IS	RF	EX	DF	DS	TC	WB	
Delay Slot		IF	IS	RF	EX	DF	DS	TC	WB
Branch Inst+8			IF	IS	null	null	null	null	null
Branch Inst+12				IF	null	null	null	null	null
Branch Targ					IF	IS	RF	EX	DF

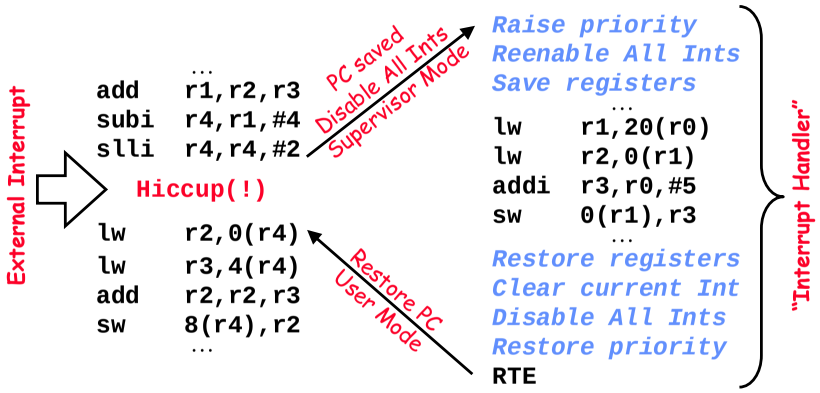
- On a *non-taken branch*, there is simply a delay slot (following two cycles *not* lost).
- This is bad for loops. We could:
  - Predict taken and keep delay slot.
  - Use our pre-decoded cache technique to completely remove DS.

Exceptions and Interrupts

(Hardware)

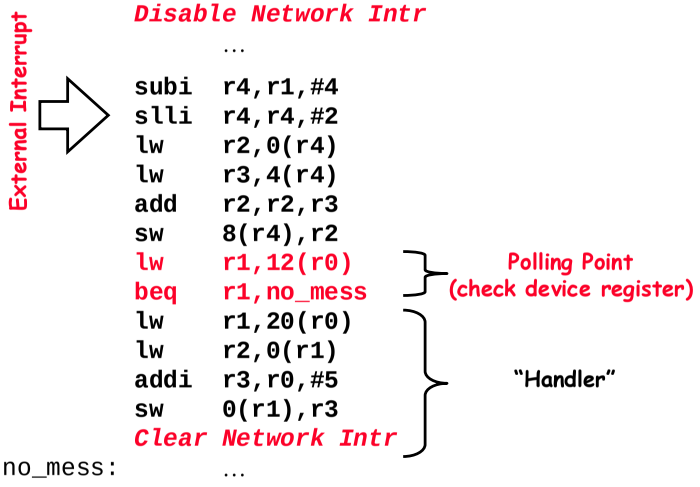
Example: Device Interrupt

(Say, arrival of network message)



Alternative: Polling

(again, for arrival of network message)



Polling is faster/slower than Interrupts.

- Polling is faster than interrupts because
  - Compiler knows which registers in use at polling point. Hence, do not need to save and restore registers (or not as many).
  - Other interrupt overhead avoided (pipeline flush, trap priorities, etc).
- Polling is slower than interrupts because
  - Overhead of polling instructions is incurred regardless of whether or not handler is run. This could add to inner-loop delay.
  - Device may have to wait for service for a long time.
- When to use one or the other?
  - Multi-axis tradeoff
    - » Frequent/regular events good for polling, *as long as device can be controlled at user level.*
    - » Interrupts good for infrequent/irregular events
    - » Interrupts good for ensuring regular/predictable service of events.

Exception/Interrupt classifications

- *Exceptions:* relevant to the current process
  - Faults, arithmetic traps, and synchronous traps
  - Invoke software on behalf of the currently executing process
- *Interrupts:* caused by asynchronous, outside events
  - I/O devices requiring service (DISK, network)
  - Clock interrupts (real time scheduling)
- *Machine Checks:* caused by serious hardware failure
  - Not always restartable
  - Indicate that bad things have happened.
    - » Non-recoverable ECC error
    - » Machine room fire
    - » Power outage

A related classification: Synchronous vs. Asynchronous

- *Synchronous:* means related to the instruction stream, i.e. during the execution of an instruction
  - Must stop an instruction that is currently executing
  - Page fault on load or store instruction
  - Arithmetic exception
  - Software Trap Instructions
- *Asynchronous:* means unrelated to the instruction stream, i.e. caused by an outside event.
  - Does not have to disrupt instructions that are already executing
  - Interrupts are asynchronous
  - Machine checks are asynchronous
- *SemiSynchronous (or high-availability interrupts):*
  - Caused by external event but may have to disrupt current instructions in order to guarantee service

Interrupt controller hardware and mask levels

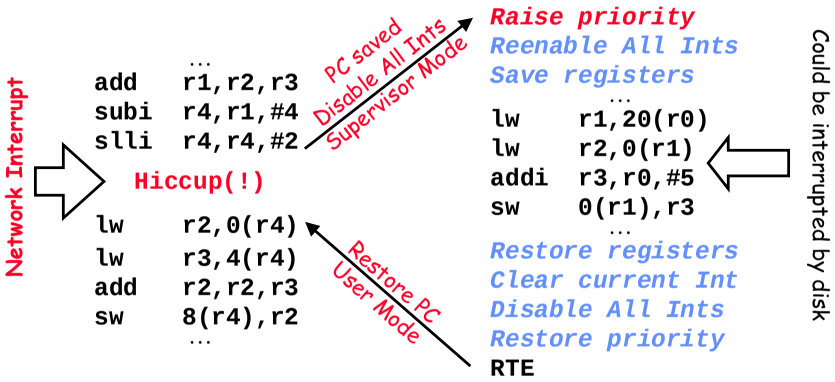
- Operating system constructs a hierarchy of masks that reflects some form of interrupt priority.
- For instance:

Priority	Examples
0	Software interrupts
2	Network Interrupts
4	Sound card
5	Disk Interrupt
6	Real Time clock
≡	Non-Maskable Ints (power)

  - This reflects the an order of urgency to interrupts
  - For instance, this ordering says that disk events can interrupt the interrupt handlers for network interrupts.

Recap: Device Interrupt

(Say, arrival of network message)



Note that priority must be raised to avoid recursive interrupts!

SPARC (and RISC I) had register windows

- On interrupt or procedure call, simply switch to a different set of registers
- Really saves on interrupt overhead
  - Interrupts can happen at any point in the execution, so compiler cannot help with knowledge of live registers.
  - Conservative handlers must save all registers
  - Short handlers might be able to save only a few, but this analysis is compilcated
- Not as big a deal with procedure calls
  - Original statement by Patterson was that Berkeley didn't have a compiler team, so they used a hardware solution
  - Good compilers can allocate registers across procedure boundaries
  - Good compilers know what registers are live at any one time
- However, register windows have returned!
  - IA64 has them
  - Many other processors have shadow registers for interrupts

Supervisor State

- Typically, processors have some amount of state that user programs are not allowed to touch.
  - Page mapping hardware/TLB
    - » TLB prevents one user from accessing memory of another
    - » TLB protection prevents user from modifying mappings
  - Interrupt controllers -- User code prevented from crashing machine by disabling interrupts. Ignoring device interrupts, etc.
  - Real-time clock interrupts ensure that users cannot lockup/crash machine even if they run code that goes into a loop:
    - » "Preemptive Multitasking" vs "non-preemptive multitasking"
- Access to hardware devices restricted
  - Prevents malicious user from stealing network packets
  - Prevents user from writing over disk blocks
- Distinction made with at least two-levels: USER/SYSTEM (one hardware mode-bit)
  - x86 architectures actually provide 4 different levels, only two usually used by OS (or only 1 in older Microsoft OSs)

Entry into Supervisor Mode

- Entry into supervisor mode typically happens on interrupts, exceptions, and special trap instructions.
- Entry goes through kernel instructions:
  - interrupts, exceptions, and trap instructions change to supervisor mode, then jump (indirectly) through table of instructions in kernel

```
intvec:  j      handle_int0
         j      handle_int1
                ...
         j      handle_fp_except0
                ...
         j      handle_trap0
         j      handle_trap1
```

  - OS "System Calls" are just trap instructions:

```
read(fd,buffer,count)  =>   st     20(r0),r1
                            st     24(r0),r2
                            st     28(r0),r3
                            trap   $READ
```

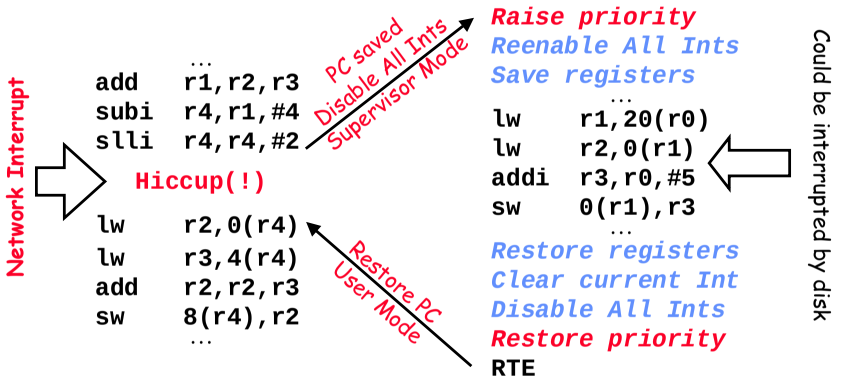
- OS overhead can be serious concern for achieving fast interrupt behavior.

Administrative

- Final class size: ?
  - 26 people took exam. Number of people still taking class???
  - Will be telling Michael-David that everyone is in. Make sure that you at least put yourself on the waiting list.
  - People in the <50 category should contact me.
  - Want to get electronic photos of everyone in class
    - » Will bring digital camera next time
- Paper summaries should be summaries!
  - Single paragraphs
  - You are supposed to read through and extract the key ideas (as you see them).
- OK, Ok. I didn't put up the electronic submission mechanism yet.

Review: Device Interrupt

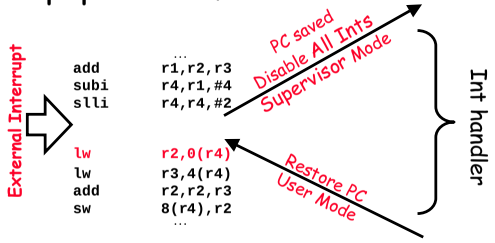
(Say, arrival of network message)



Note that priority must be raised to avoid recursive interrupts!

Precise Interrupts/Exceptions

- An interrupt or exception is considered *precise* if there is a single instruction (or interrupt point) for which:
  - All instructions before that have committed their state
  - No following instructions (including the interrupting instruction) have modified any state.
- This means, that you can restart execution at the interrupt point and "get the right answer"
  - Implicit in our previous example of a device interrupt:
    - » Interrupt point is at first lw instruction



Precise interrupt point requires multiple PCs to describe in presence of delayed branches

```
      addi  r4,r3,#4
      sub   r1,r2,r3
PC:   bne   r1,there
PC+4: and   r2,r3,r5
      <other insts>
```

⇐ Interrupt point described as <PC,PC+4>

```
      addi  r4,r3,#4
      sub   r1,r2,r3
PC:   bne   r1,there
PC+4: and   r2,r3,r5
      <other insts>
```

Interrupt point described as:

<PC+4,there> (branch was taken)
or
<PC+4,PC+8> (branch was not taken)

Why are precise interrupts desirable?

- Many types of interrupts/exceptions need to be restartable. Easier to figure out what actually happened:
  - I.e. TLB faults. Need to fix translation, then restart load/store
  - IEEE gradual underflow, illegal operation, etc:

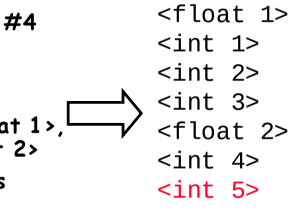
    e.g. Suppose you are computing: $f(x) = \frac{\sin(x)}{x}$
    Then, for $x \to 0$,

    $$f(0) = \frac{0}{0} \Rightarrow NaN + illegal_operation$$

    Want to take exception, replace *NaN* with 1, then restart.
- Restartability doesn't *require* preciseness. However, preciseness makes it *a lot easier* to restart.
- Simplify the task of the operating system a lot
  - Less state needs to be saved away if unloading process.
  - Quick to restart (making for fast interrupts)

Approximations to precise interrupts

- Hardware has imprecise state at time of interrupt
- Exception handler must figure out how to find a precise PC at which to restart program.
  - Emulate instructions that may remain in pipeline
  - Example: SPARC allows limited parallelism between FP and integer core:
    - » possible that integer instructions #1 - #4 have already executed at time that the first floating instruction gets a recoverable exception
    - » Interrupt handler code must fixup <float 1>, then emulate both <float 1> and <float 2>
    - » At that point, precise interrupt point is integer instruction #5.

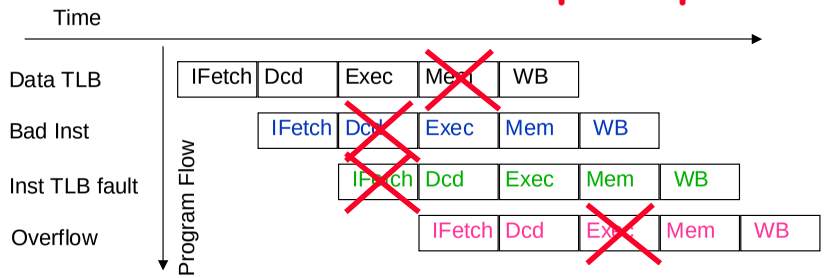


- Vax had string move instructions that could be in middle at time that page-fault occurred.
- Could be arbitrary processor state that needs to be restored to restart execution.

Precise Exceptions in simple 5-stage pipeline:

- Exceptions may occur at different stages in pipeline (I.e. out of order):
  - Arithmetic exceptions occur in execution stage
  - TLB faults can occur in instruction fetch or memory stage
- What about interrupts? The doctor's mandate of "do no harm" applies here: try to interrupt the pipeline as little as possible
- All of this solved by tagging instructions in pipeline as "cause exception or not" and wait until end of memory stage to flag exception
  - Interrupts become marked NOPs (like bubbles) that are placed into pipeline instead of an instruction.
  - Assume that interrupt condition persists in case NOP flushed
  - Clever instruction fetch might start fetching instructions from interrupt vector, but this is complicated by need for supervisor mode switch, saving of one or more PCs, etc

Another look at the exception problem



- Use pipeline to sort this out!
  - Pass exception status along with instruction.
  - Keep track of PCs for every instruction in pipeline.
  - Don't act on exception until it reache WB stage
- Handle interrupts through "faulting noop" in IF stage
- When instruction reaches WB stage:
  - Save PC ⇒ EPC, Interrupt vector addr ⇒ PC
  - Turn all instructions in earlier stages into noops!

How to achieve precise interrupts when instructions executing in arbitrary order?

- Jim Smith's classic paper (you read for this time) discusses several methods for getting precise interrupts:
  - In-order instruction completion
  - Reorder buffer
  - History buffer
- We will discuss these after we see the advantages of out-of-order execution.

Impact of Hazards on Performance

Case Study: MIPS R4000 (200 MHz)

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

Case Study: MIPS R4000

TWO Cycle Load Latency	IF	IS	RF	EX	DF	[DS]	TC	WB
		IF	IS	RF	EX	DF	DS	TC
			IF	IS	RF	EX	[EX]	DS
				IF	IS	RF	IS	DF
					IF	IS	RF	EX
						IF	IS	RF
							IF	IS
								IF
THREE Cycle Branch Latency (conditions evaluated during EX phase)	IF	IS	RF	[EX]	DF	DS	TC	WB
		IF	IS	RF	EX	DF	DS	TC
			IF	IS	RF	EX	DF	DS
				IF	IS	RF	EX	DF
					[IF]	IS	RF	EX
						IF	IS	RF
							IF	IS
								IF

Delay slot plus two stalls
Branch likely cancels delay slot if not taken

MIPS R4000 Floating Point

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

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

MIPS FP Pipe Stages

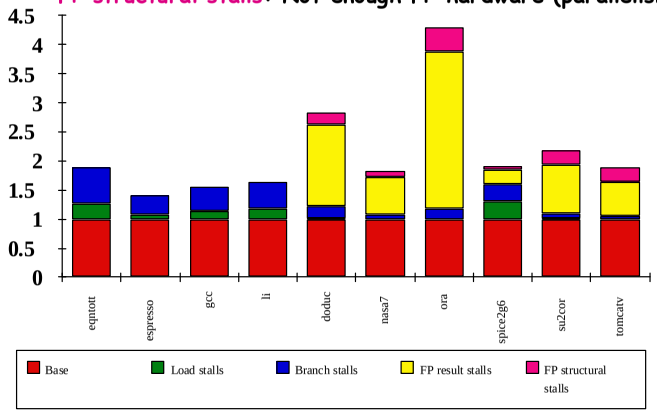
FP Instr	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	…
Add, Subtract	U	S+A	A+R	R+S					
Multiply	U	E+M	M	M	M	N	N+A	R	
Divide	U	A	R	D^{28}	…	D+A	D+R,	D+R,	D+A, D+R, A, R
Square root	U	E	$(A+R)^{108}$	…	A	R			
Negate	U	S							
Absolute value	U	S							
FP compare	U	A	R						

Stages:

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

R4000 Performance

- Not ideal CPI of 1:
  - Load stalls (1 or 2 clock cycles)
  - Branch stalls (2 cycles + unfilled slots)
  - FP result stalls: RAW data hazard (latency)
  - FP structural stalls: Not enough FP hardware (parallelism)



Advanced Pipelining and Instruction Level Parallelism (ILP)

- ILP: Overlap execution of unrelated instructions
- gcc 17% control transfer
  - 5 instructions + 1 branch
  - Beyond single block to get more instruction level parallelism
- Loop level parallelism one opportunity
  - First SW, then HW approaches
- DLX Floating Point as example
  - Measurements suggests R4000 performance FP execution has room for improvement

FP Loop: Where are the Hazards?

```
Loop:  LD    F0,0(R1)   ;F0=vector element
       ADDD  F4,F0,F2   ;add scalar from F2
       SD    0(R1),F4   ;store result
       SUBI  R1,R1,8    ;decrement pointer 8B (DW)
       BNEZ  R1,Loop    ;branch R1!=zero
       NOP              ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Execution in cycles*	*Latency in cycles*
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	1	1
Load double	Store double	1	0
Integer op	Integer op	1	0

Where are the stalls?

FP Loop Showing Stalls

```
1 Loop: LD    F0,0(R1)   ;F0=vector element
2       stall
3       ADDD  F4,F0,F2   ;add scalar in F2
4       stall
5       stall
6       SD    0(R1),F4   ;store result
7       SUBI  R1,R1,8    ;decrement pointer 8B (DW)
8       BNEZ  R1,Loop    ;branch R1!=zero
9       stall            ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- 9 clocks: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

```
1 Loop: LD    F0,0(R1)
2       stall
3       ADDD  F4,F0,F2
4       SUBI  R1,R1,8
5       BNEZ  R1,Loop    ;delayed branch
6       SD    8(R1),F4   ;altered when move past SUBI
```

Swap BNEZ and SD by changing address of SD

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

6 clocks: Unroll loop 4 times code to make faster?

Unroll Loop Four Times (straightforward way)

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

1 cycle stall

2 cycles stall

Rewrite loop to minimize stalls?

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration

Assumes R1 is multiple of 4

Unrolled Loop That Minimizes Stalls

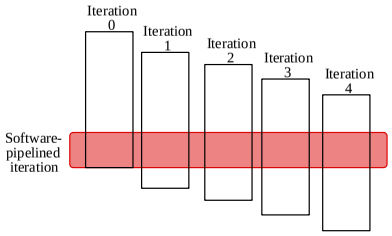
```
1 Loop: LD     F0,0(R1)
2       LD     F6,-8(R1)
3       LD     F10,-16(R1)
4       LD     F14,-24(R1)
5       ADDD   F4,F0,F2
6       ADDD   F8,F6,F2
7       ADDD   F12,F10,F2
8       ADDD   F16,F14,F2
9       SD     0(R1),F4
10      SD     -8(R1),F8
11      SD     -16(R1),F12
12      SUBI   R1,R1,#32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

- What assumptions made when moved code?
  - OK to move store past SUBI even though changes register
  - OK to move loads before stores: get right data?
  - When is it safe for compiler to do such changes?

14 clock cycles, or 3.5 per iteration

Another possibility: Software Pipelining

- Observation: if iterations from loops are independent, then can get more 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)



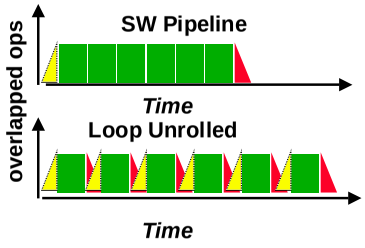
Software Pipelining Example

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
```



- Symbolic Loop Unrolling
  - Maximize result-use distance
  - Less code space than unrolling
  - Fill & drain pipe only once per loop vs. once per each unrolled iteration in loop unrolling

5 cycles per iteration

Compiler Perspectives on Code Movement

- Compiler concerned about dependencies in program
- Whether or not a HW hazard depends on pipeline
- Try to schedule to avoid hazards that cause performance losses
- (True) Data dependencies (RAW if a hazard for HW)
  - Instruction i produces a result used by instruction j, or
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
- If dependent, can't execute in parallel
- Easy to determine for registers (fixed names)
- Hard for memory ("memory disambiguation" problem):
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?

Where are the data dependencies?

```
1 Loop: LD    F0,0(R1)
2       ADDD  F4,F0,F2
3       SUBI  R1,R1,8
4       BNEZ  R1,Loop    ;delayed branch
5       SD    8(R1),F4   ;altered when move past SUBI
```


Compiler Perspectives on Code Movement

- Another kind of dependence called name dependence: two instructions use same name (register or memory location) but don't exchange data
- Antidependence (WAR if a hazard for HW)
  - Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first
- Output dependence (WAW if a hazard for HW)
  - Instruction i and instruction j write the same register or memory location; ordering between instructions must be preserved.

Where are the name dependencies?

```
1 Loop:LD     F0,0(R1)
2      ADDD   F4,F0,F2
3      SD     0(R1),F4     ;drop SUBI & BNEZ
4      LD     F0,-8(R1)
5      ADDD   F4,F0,F2
6      SD     -8(R1),F4    ;drop SUBI & BNEZ
7      LD     F0,-16(R1)
8      ADDD   F4,F0,F2
9      SD     -16(R1),F4   ;drop SUBI & BNEZ
10     LD     F0,-24(R1)
11     ADDD   F4,F0,F2
12     SD     -24(R1),F4
13     SUBI   R1,R1,#32    ;alter to 4*8
14     BNEZ   R1,LOOP
15     NOP
```

How can remove them?

Where are the name dependencies?

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

Called "register renaming"

Compiler Perspectives on Code Movement

- Name Dependencies are Hard to discover for Memory Accesses
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?
- Our example required compiler to know that if R1 doesn't change then:

  `0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)`

  There were no dependencies between some loads and stores so they could be moved by each other

Compiler Perspectives on Code Movement

- Final kind of dependence called control dependence
- Example

```
if p1 {S1;};
if p2 {S2;};
```

S1 is control dependent on p1 and S2 is control dependent on p2 but not on p1.

Compiler Perspectives on Code Movement

- Two (obvious?) constraints on control dependences:
  - An instruction that is control dependent on a branch cannot be moved before the branch.
  - An instruction that is not control dependent on a branch cannot be moved to after the branch (or its execution will be controlled by the branch).
- Control dependencies relaxed to get parallelism; get same effect if preserve order of exceptions (address in register checked by branch before use) and data flow (value in register depends on branch)

Where are the control dependencies?

```
1 Loop:  LD     F0,0(R1)
2        ADDD   F4,F0,F2
3        SD     0(R1),F4
4        SUBI   R1,R1,8
5        BEQZ   R1,exit
6        LD     F0,0(R1)
7        ADDD   F4,F0,F2
8        SD     0(R1),F4
9        SUBI   R1,R1,8
10       BEQZ   R1,exit
11       LD     F0,0(R1)
12       ADDD   F4,F0,F2
13       SD     0(R1),F4
14       SUBI   R1,R1,8
15       BEQZ   R1,exit
....
```

When Safe to Unroll Loop?

- Example: Where are data dependencies? (A,B,C distinct & nonoverlapping)

```
for (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i];     /* S1 */
      B[i+1] = B[i] + A[i+1];   /* S2 */
}
```

1. S2 uses the value, A[i+1], computed by S1 in the same iteration.

2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].
This is a "loop-carried dependence": between iterations

- For our prior example, each iteration was distinct
- Implies that iterations can't be executed in parallel, Right????

Does a loop-carried dependence mean there is no parallelism???

- Consider:

```
for (i=0; i< 8; i=i+1) {
      A = A + C[i];     /* S1 */
}
```

Could compute:

```
"Cycle 1":  temp0 = C[0] + C[1];
            temp1 = C[2] + C[3];
            temp2 = C[4] + C[5];
            temp3 = C[6] + C[7];
"Cycle 2":  temp4 = temp0 + temp1;
            temp5 = temp2 + temp3;
"Cycle 3":  A = temp4 + temp5;
```

- Relies on associative nature of "+".
- See "Parallelizing Complex Scans and Reductions" by Allan Fisher and Anwar Ghuloum (handed out next week)

Can HW get CPI closer to 1?

- 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 #1:
Allow instructions behind stall to proceed

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

Out-of-order execution ⇒ out-of-order completion?

- Key idea #2: Register Renaming

```
DIVD  F0,F2,F4          DIVD F0,F2,F4
ADDD  F10,F0,F8         ADDD F10,F0,F8
SUBD  F0,F8,F14   ⇒     SUBD F100,F8,F14
MULD  F6,F10,F0         MULD F6,F10,F100
```

Totally removes WAR and WAW hazards.

Next time: Advanced pipelining

- How do we prevent WAR and WAW hazards?
- How do we deal with variable latency?
  - Forwarding for RAW hazards harder.

Instruction		Clock Cycle Number																
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD	F6,34(R2)	IF	ID	EX	MEM	WB												
LD	F2,45(R3)		IF	ID	EX	MEM	WB											
MULTD	F0,F2,F4			IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	MEM	WB
SUBD	F8,F6,F2				IF	ID	A1	A2	MEM	WB								
DIVD	F10,F0,F6					IF	ID	stall	stall	stall	stall	stall	stall	stall	stall	stall	D1	D2
ADDD	F6,F8,F2						IF	ID	A1	A2	MEM	WB						

RAW

WAR

Summary #1

- Control flow causes lots of trouble with pipelining
  - Other hazards can be "fixed" with more transistors or forwarding
  - We will spend a lot of time on branch prediction techniques
- Some pre-decode techniques can transform dynamic decisions into static ones (VLIW-like)
  - Beginnings of dynamic compilation techniques
- Interrupts and Exceptions either interrupt the current instruction or happen between instructions
  - Possibly large quantities of state must be saved before interrupting
- Machines with *precise exceptions* provide one single point in the program to restart execution
  - All instructions before that point have completed
  - No instructions after or including that point have completed
- Hardware techniques exist for precise exceptions even in the face of out-of-order execution!
  - Important enabling factor for out-of-order execution

Summary #2: Software Scheduling

- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
- Increasing length of pipe increases impact of hazards
  - pipelining helps instruction bandwidth, not latency!
- Instruction Level Parallelism (ILP) found either by compiler or hardware.
- Loop level parallelism is easiest to see
  - SW dependencies/compiler sophistication determine if compiler can unroll loops
  - Memory dependencies hardest to determine => Memory disambiguation
  - Very sophisticated transformations available
- Next time: HW exploiting ILP
  - Works when can't know dependence at compile time.
  - Code for one machine runs well on another