Lecture 3: Control flow, interrupts and exceptions

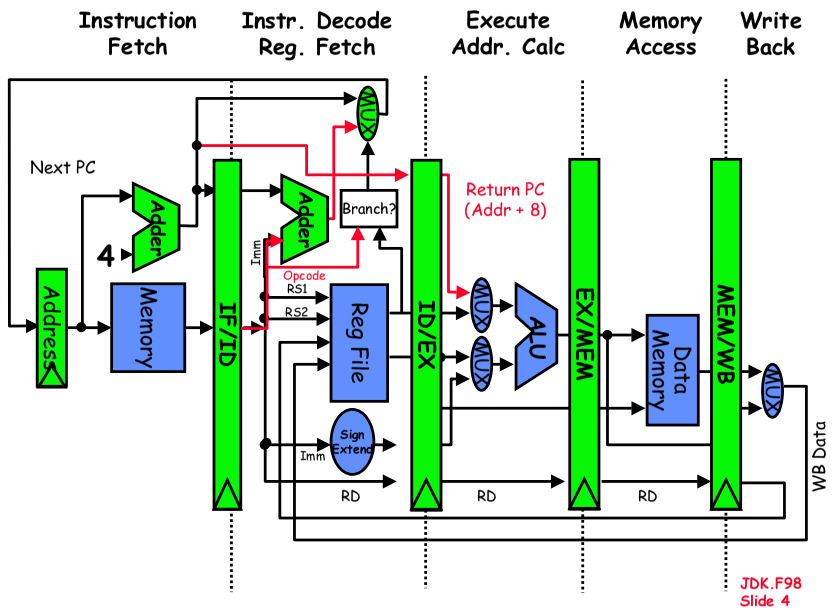
Prof. John Kubiatowicz
Computer Science 252
Fall 1998

Changes in the flow of instructions make pipelining difficult

- Must avoid adding too much overhead in pipeline startup and drain.
- Branches and Jumps cause fast alteration of PC. Things that get in the way:
  - Instructions take time to decode, introducing delay slots.
  - The next PC takes time to compute
  - For conditional branches, the branch direction takes time to compute.
- Interrupts and Exceptions also cause problems
  - Must make decisions about when to interrupt flow of instructions
  - Must preserve sufficient pipeline state to resume execution

Jumps and Calls (JAL) (unconditional branches)

- Even though we know that we will change PC, still require delay slot because of:
  - Instruction Decode -- Pretty hard and fast
  - PC Computation -- Could fix with absolute jumps/calls (not necessarily a good solution)
- Basically, there is a decision being made, which takes time.
- This suggests single delay slot:
  - I.e. next instruction after jump or JAL is *always* executed



Achieving "zero-cycle" jump

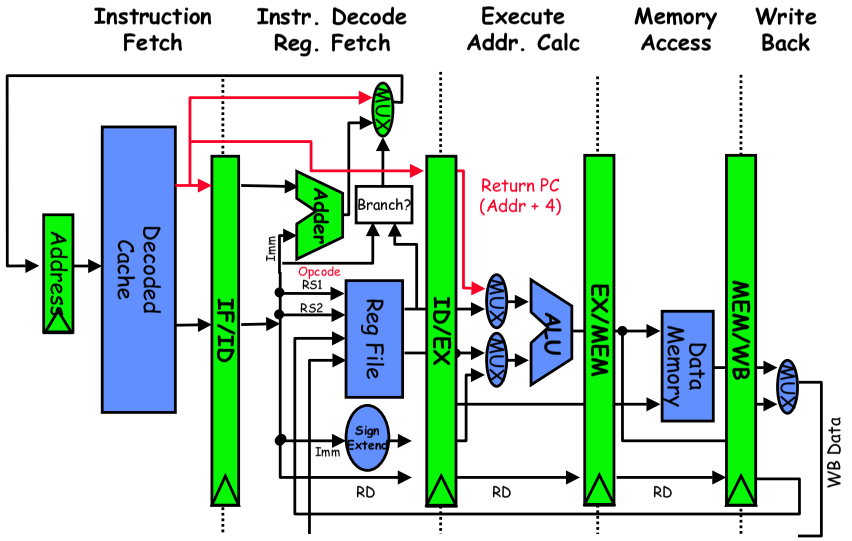
- However, what really has to be done at runtime?
  - Once an instruction has been detected as a jump or JAL, we might recode it in the internal cache.
  - Very limited form of dynamic compilation?

```
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

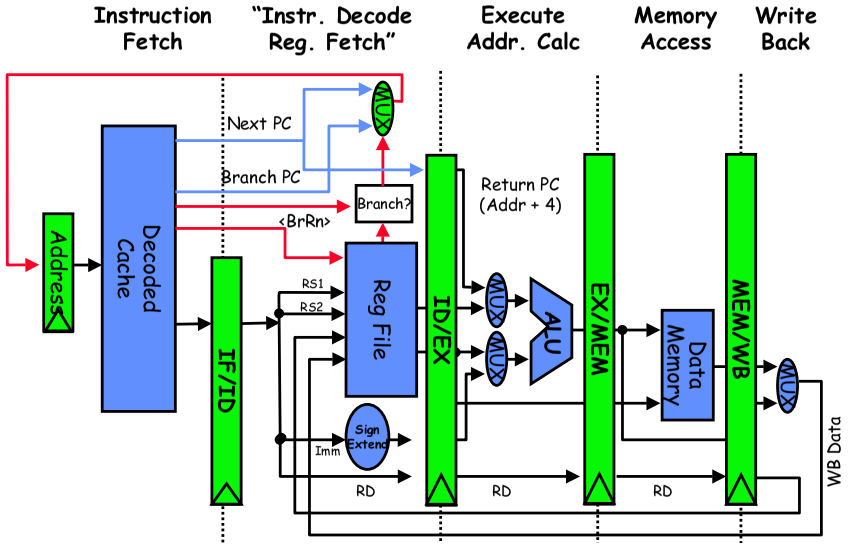
Why not do this for branches?
(original CRISP idea, applied to DLX)

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

Internal Cache state:

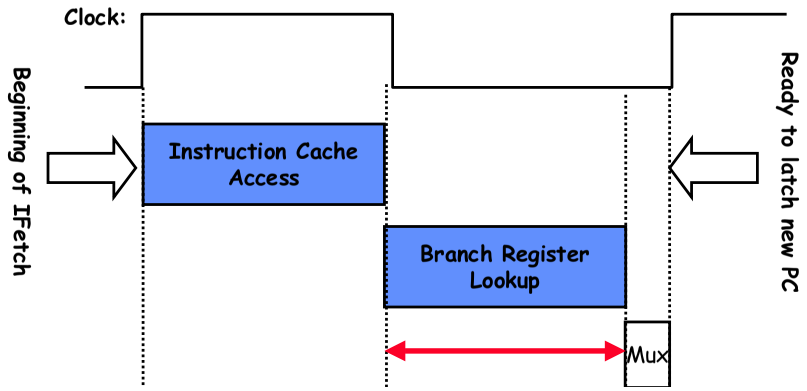
				Next	Branch
A:	and r3,r1,r5	N	A+4	N/A
	addi r2,r3,#4	N	A+8	N/A
	sub r4,r2,r1	BnR4	A+16	loop
	---	--	---	---
A+16:	subi r1,r1,#1	N	A+20	N/A

- 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).

	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

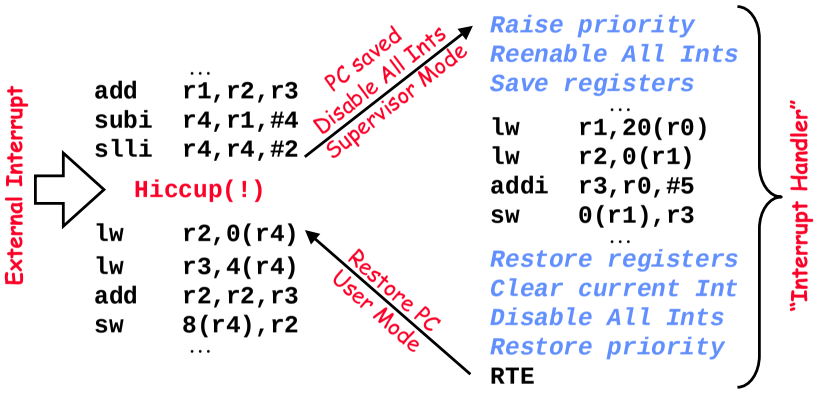
- On a *non-taken branch*, there is simply a delay slot (following two cycles *not* lost).
- This is bad for loops. We could reverse this behavior with our pre-decoded cache technique.

Exceptions and Interrupts

(Hardware)

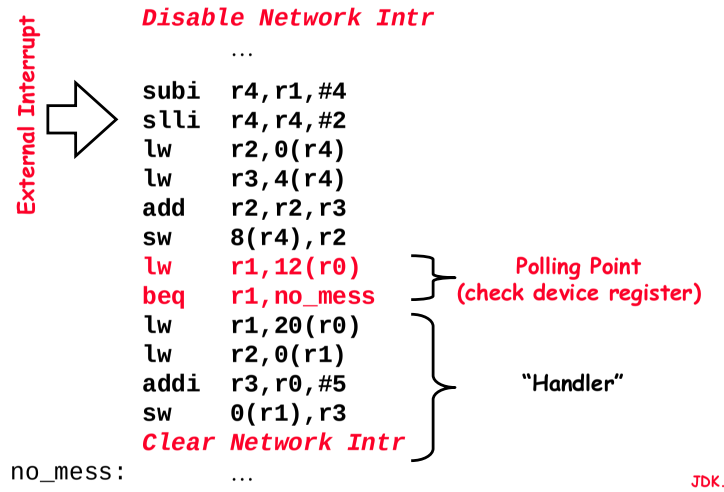
Example: Device Interrupt

(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

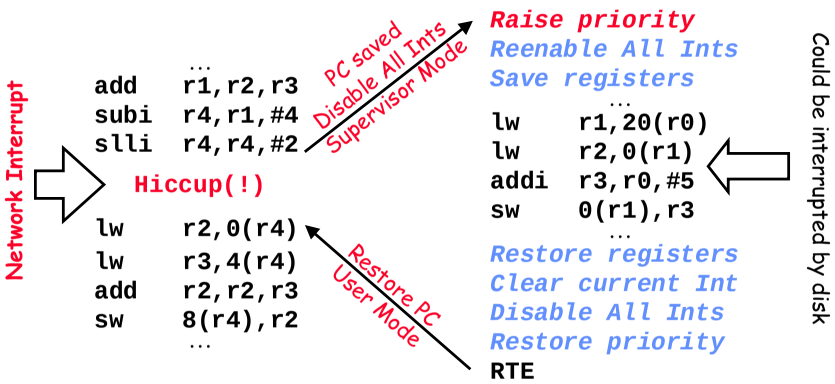
- Interrupt disable mask may be multi-bit word accessed through some special memory address
- Operating system constructs a hierarchy of masks that reflects some form of interrupt priority.
- For instance:

Priority	Examples
0	Software interrupts
2	Network Interrupts
4	Sound card
5	Disk Interrupt
6	Real Time clock

  - 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)



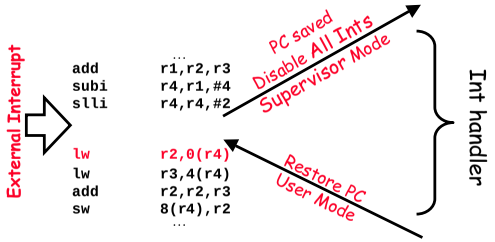
Note that priority must be raised to avoid recursive interrupts!

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

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 instruction have committed their state and no following instructions including the interrupting instruction have modified any state.
  - This means, effectively, 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?

- Simplify the task of the operating system a lot
  - Quick to get restart point (making for fast interrupts)
  - Small amount of state needs to be saved away if unloading process.
- Many types of interrupts/exceptions need to be restartable:
  - I.e. TLB faults,IEEE gradual underflow, etc.

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.
  - Done by emulating 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

⇨

```
<float 1>
<int 1>
<int 2>
<int 3>
<float 2>
<int 4>
<int 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.

How to achieve precise interrupts
(In-Class discussion of Jim Smith's paper)

- In-order instruction issue
- Several methods of getting sequential state:
  - in-order instruction completion
  - Reorder buffer
  - History buffer

Summary

- Changes in control flow cause the most trouble with pipelining
- Some pre-decode techniques can transform dynamic decisions into static ones (VLIW-like)
- Interrupts and Exceptions either interrupt the current instruction or happen between instructions
- 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 interrupts even in the face of out-of-order execution!