EECS 252 Graduate Computer Architecture
Lecture 4

Control Flow (continued)
Interrupts

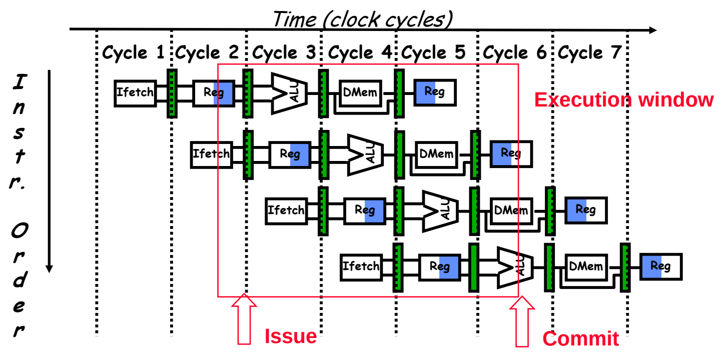
John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs252
http://www-inst.eecs.berkeley.edu/~cs252

Review: Summary of Technology Trends

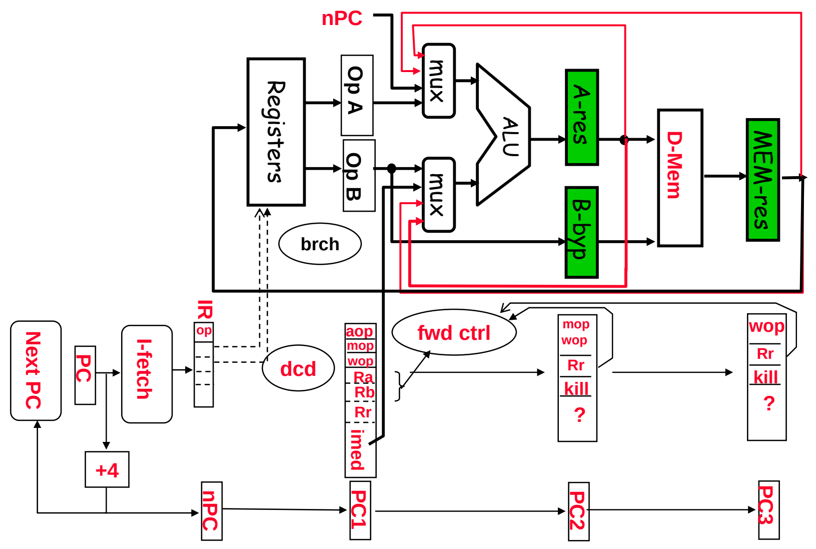
- For disk, LAN, memory, and microprocessor, bandwidth improves by square of latency improvement
  – In the time that bandwidth doubles, latency improves by no more than 1.2X to 1.4X
- Lag probably even larger in real systems, as bandwidth gains multiplied by replicated components
  – Multiple processors in a cluster or even in a chip
  – Multiple disks in a disk array
  – Multiple memory modules in a large memory
  – Simultaneous communication in switched LAN
- HW and SW developers should innovate assuming Latency Lags Bandwidth
  – If everything improves at the same rate, then nothing really changes
  – When rates vary, require real innovation

Review: Ordering Properties of basic inst. pipeline



- Instructions issued in order
- Operand fetch is stage 2 => operand fetched in order
- Write back in stage 5 => no WAW, no WAR hazards
- Common pipeline flow => operands complete in order
- Stage changes only at "end of instruction"

Review: Control Pipeline

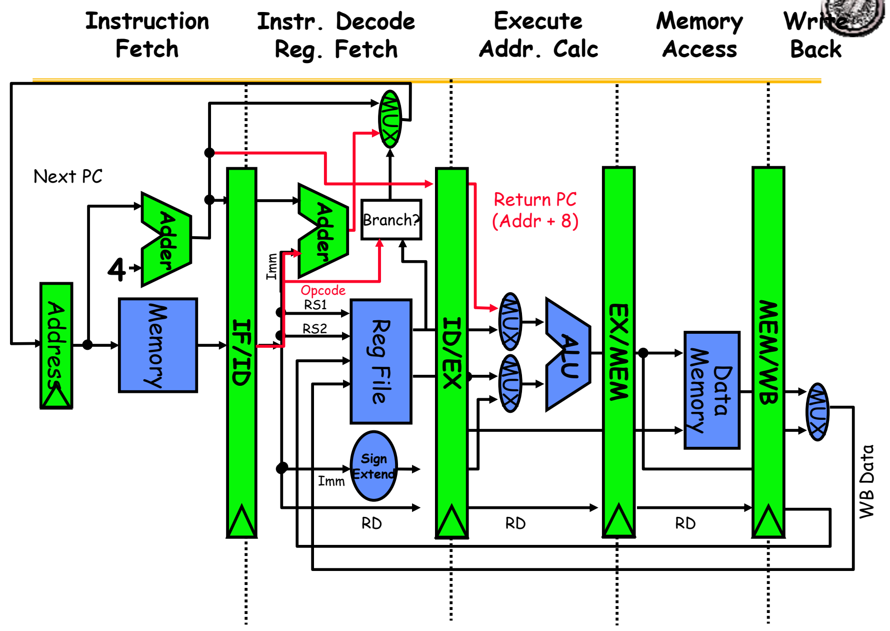


Changes in the flow of instructions make pipelining difficult

- 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

- 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?

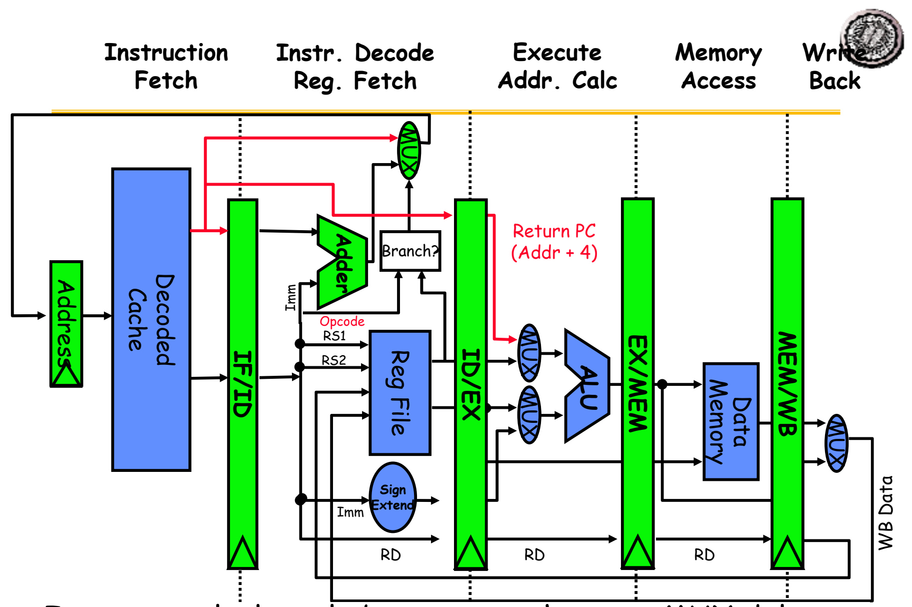
Internal Cache state:

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

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

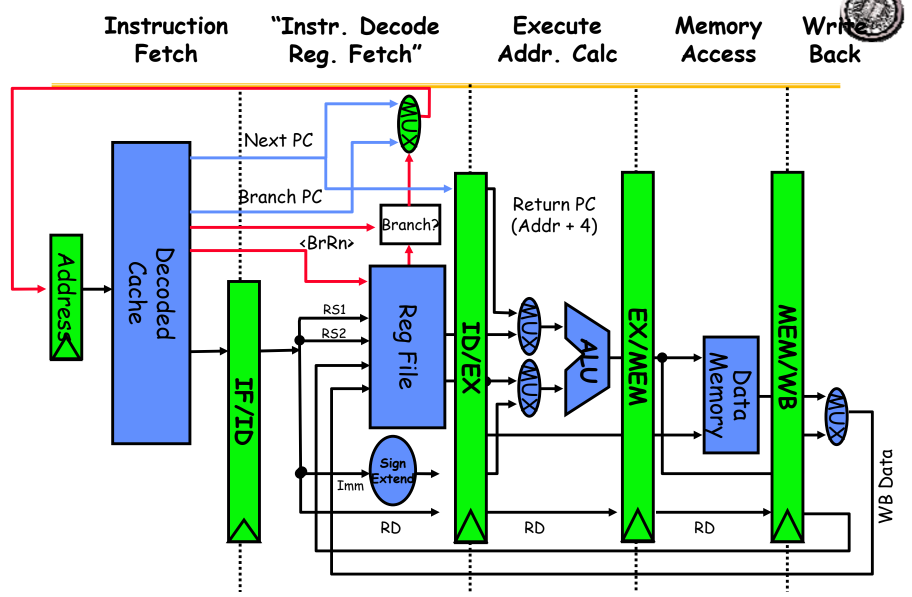
Why not do this for branches?
(original CRISP idea, applied to DLX)

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

Internal Cache state:

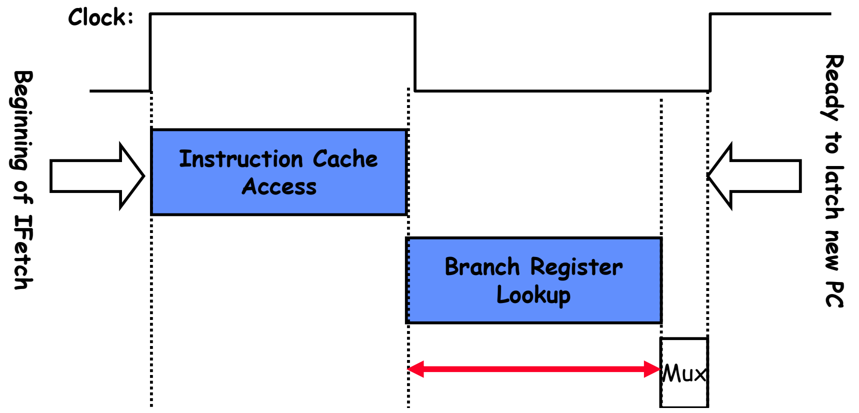
			Next	Branch
A:	and r3,r1,r5	N	A+4	N/A
	addi r2,r3,#4	N	A+8	N/A
	sub r4,r2,r1	BnR4	A+16	loop
	---	--	---	---
A+16:	subi r1,r1,#1	N	A+20	N/A

- **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

R4000 Pipeline

- 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.

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

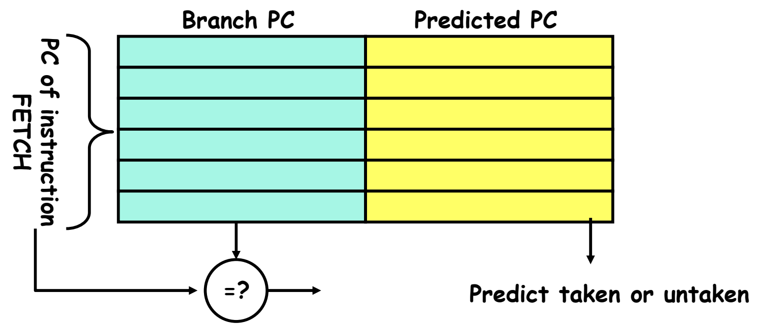
- This causes the next instruction to be immediately fetched from branch destination (predict taken)
- If branch ends up being not taking, then squash destination instruction and restart pipeline at address A+16

Dynamic Branch Prediction

- Prediction could be "Static" (at compile time) or "Dynamic" (at runtime)
  - For our example, if we were to statically predict "taken", we would only be wrong once each pass through loop
- Is dynamic branch prediction better than static branch prediction?
  - Seems to be. Still some debate to this effect
  - Today, lots of hardware being devoted to dynamic branch predictors.
- Does branch prediction make sense for 5-stage, in-order pipeline? What about 8-stage pipeline?
  - Perhaps: eliminate branch delay slots/then predict branches

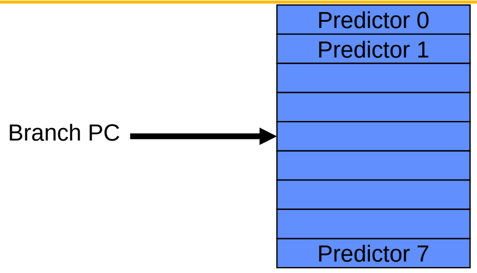
Simple dynamic prediction: Branch Target Buffer (BTB)

- Address of branch index to get prediction AND branch address (if taken)
  - Must check for branch match now, since can't use wrong branch address
  - Grab predicted PC from table since may take several cycles to compute



- Update predicted PC when branch is actually resolved
- Return instruction addresses predicted with stack

Branch History Table (BHT)



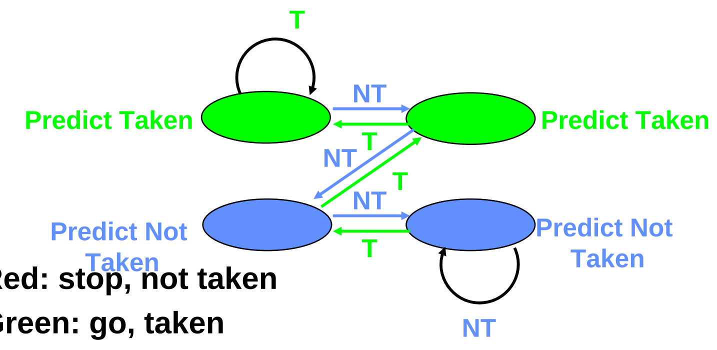
- BHT is a table of “Predictors”
  - Could be 1-bit, could be complete state machine
  - Indexed by PC address of Branch – without tags
- In Fetch state of branch:
  - BTB identifies branch
  - Predictor from BHT used to make prediction
- When branch completes
  - Update corresponding Predictor

Dynamic Branch Prediction: Usual Division

- Branch Target Buffer (BTB): identify branches and hold *taken addresses*
  - *Trick: identify branch before fetching instruction!*
- Branch History Table(BHT)
  - Table makes prediction by keeping long-term history
    - » Example: Simple 1-bit BHT: keep last direction of branch
  - No address check: Can be good, can be bad….
- Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iteratios before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts exit instead of looping
- Performance = *ƒ*(accuracy, cost of misprediction)
  - Misprediction ⇒ Flush Reorder Buffer

Dynamic Branch Prediction: 2-bit predictor

- Solution: 2-bit scheme where change prediction only if get misprediction *twice:* (Figure 4.13, p. 264)

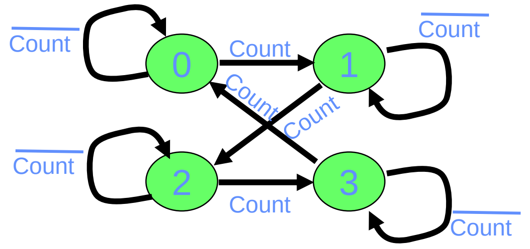


- Red: stop, not taken
- Green: go, taken
- Adds *hysteresis* to decision making process

Administrivia

- Prerequisite Quiz
  - Still looking over them
  - I believe that I will let everyone in, but will be contacting some of you about prerequisites
    - » Make sure to read over solutions. If you don’t understand something, ASK!
  - Want to get electronic photos of everyone in class
    - » Can everyone send me one? I will keep them private
- Paper summaries should be summaries!
  - Single paragraphs!!!
  - You are supposed to read through and extract the key ideas (as you see them).
- WebSite signup
  - Now works – only 9 of you currently signed up.
  - You should get a confirmation request message from mailman that requires you to click on a link and confirm addition to mailing list

Prereq: Brief Recap of FSMs



S_1	S_0	C	S_1'	S_0'
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

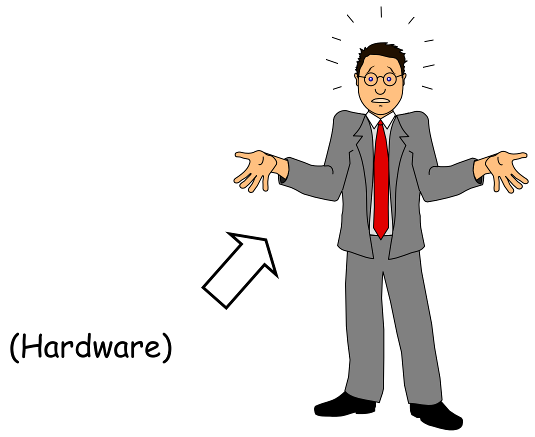
Equation production: can use "Karnaugh" maps

S_0	00	01	11	10
0	0	1	1	0
1	1	0	0	1

S_1	00	01	11	10
0	0	0	1	1
1	0	1	0	1

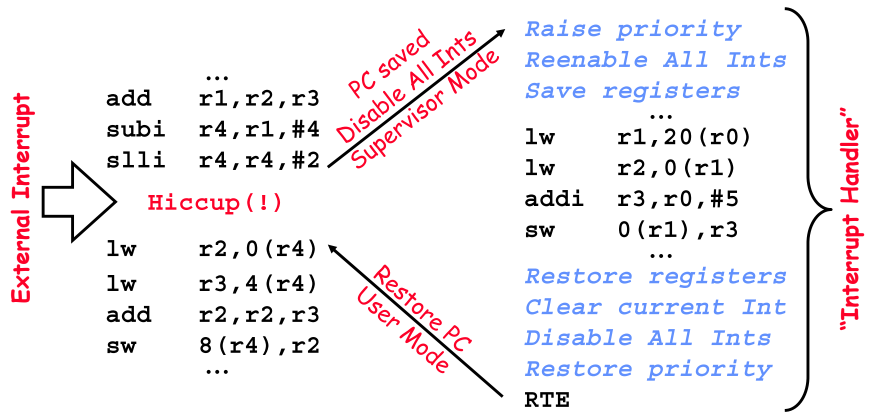
$$S_0' = (\overline{S_0} \cdot C) + (S_0 \cdot \overline{C}) \qquad S_1' = (\overline{S_1} \cdot S_0 \cdot C) + (S_1 \cdot \overline{C}) + (S_1 \cdot \overline{S_0})$$

Exceptions and Interrupts



Example: Device Interrupt

(Say, arrival of network message)

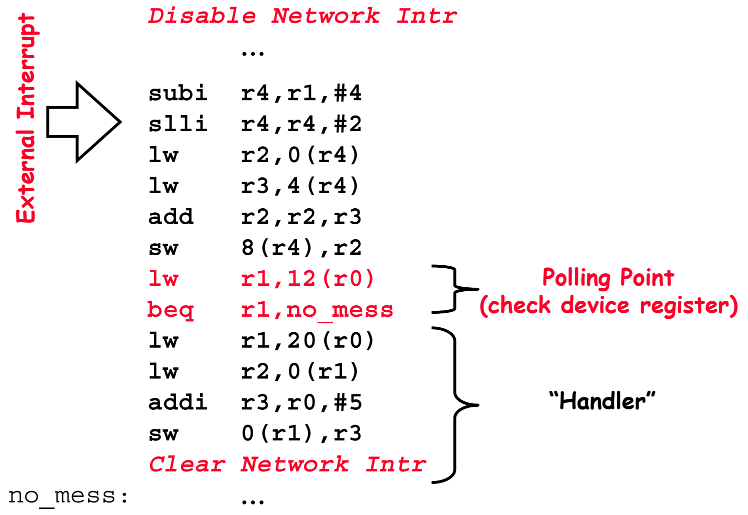


```
        …
add     r1,r2,r3
subi    r4,r1,#4
slli    r4,r4,#2
  Hiccup(!)
lw      r2,0(r4)
lw      r3,4(r4)
add     r2,r2,r3
sw      8(r4),r2
        …
```

```
Raise priority
Reenable All Ints
Save registers
        …
lw      r1,20(r0)
lw      r2,0(r1)
addi    r3,r0,#5
sw      0(r1),r3
        …
Restore registers
Clear current Int
Disable All Ints
Restore priority
RTE
```

Alternative: Polling

(again, for arrival of network message)



```
             Disable Network Intr
                     …
             subi    r4,r1,#4
             slli    r4,r4,#2
             lw      r2,0(r4)
             lw      r3,4(r4)
             add     r2,r2,r3
             sw      8(r4),r2
             lw      r1,12(r0)
             beq     r1,no_mess
             lw      r1,20(r0)
             lw      r2,0(r1)
             addi    r3,r0,#5
             sw      0(r1),r3
             Clear Network Intr
no_mess:             …
```

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.

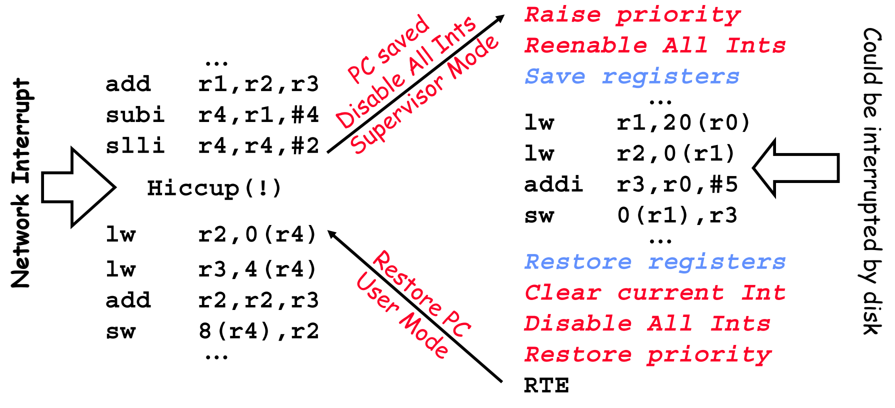
Exception/Interrupt classifications

- *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 Priorities Must be Handled



```
      ...
add   r1,r2,r3
subi  r4,r1,#4
slli  r4,r4,#2
  Hiccup(!)
lw    r2,0(r4)
lw    r3,4(r4)
add   r2,r2,r3
sw    8(r4),r2
      ...
```

```
Raise priority
Reenable All Ints
Save registers
      ...
lw    r1,20(r0)
lw    r2,0(r1)
addi  r3,r0,#5
sw    0(r1),r3
      ...
Restore registers
Clear current Int
Disable All Ints
Restore priority
RTE
```

Note that priority must be raised to avoid recursive interrupts!

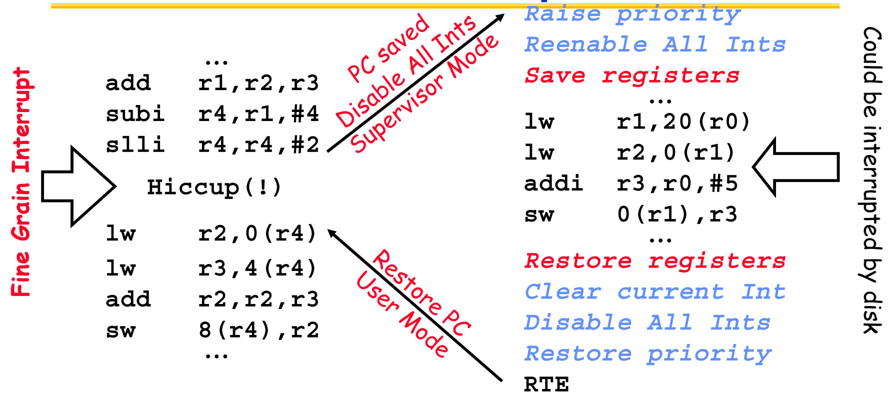
Interrupt controller hardware and mask levels

- 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.

Can we have fast interrupts?



```
        …
add   r1,r2,r3
subi  r4,r1,#4
slli  r4,r4,#2
   Hiccup(!)
lw    r2,0(r4)
lw    r3,4(r4)
add   r2,r2,r3
sw    8(r4),r2
        …
```

```
Raise priority
Reenable All Ints
Save registers
        …
lw    r1,20(r0)
lw    r2,0(r1)
addi  r3,r0,#5
sw    0(r1),r3
        …
Restore registers
Clear current Int
Disable All Ints
Restore priority
RTE
```

- Pipeline Drain: Can be very Expensive
- Priority Manipulations
- Register Save/Restore
  – 128 registers + cache misses + etc.

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
```

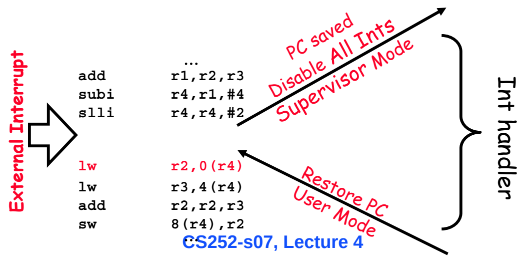
  - OS “System Calls” are just trap instructions:

```
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.

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



```
        …
add     r1,r2,r3
subi    r4,r1,#4
slli    r4,r4,#2

lw      r2,0(r4)
lw      r3,4(r4)
add     r2,r2,r3
sw      8(r4),r2
        …
```

Precise interrupt point may require multiple PCs

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

      addi   r4,r3,#4
      sub    r1,r2,r3        Interrupt point described as:
PC:   bne    r1,there
PC+4: and    r2,r3,r5     ⇐   <PC+4,there> (branch was taken)
      <other insts>                    or
                             <PC+4,PC+8> (branch was not taken)
```

- On SPARC, interrupt hardware produces “pc” and “npc” (next pc)
- On MIPS, only “pc” – must fix point in software

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.

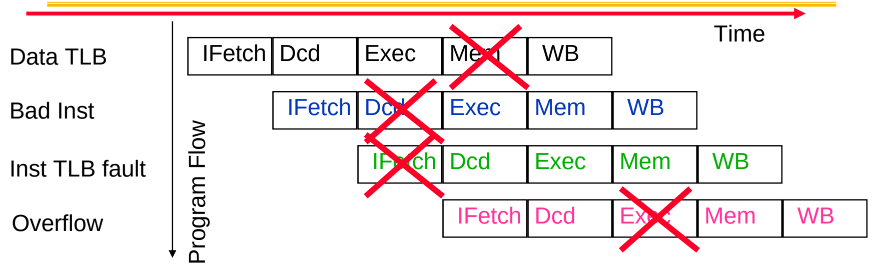
```
<float 1>
<int 1>
<int 2>
<int 3>
<float 2>
<int 4>
<int 5>
```

- 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.

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 $\Rightarrow$ EPC, Interrupt vector addr $\Rightarrow$ PC
  - Turn all instructions in earlier stages into noops!

Precise interrupts when instructions executing in arbitrary order?

- Jim Smith's classic paper (you will read this) 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.

And in conclusion …

- 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