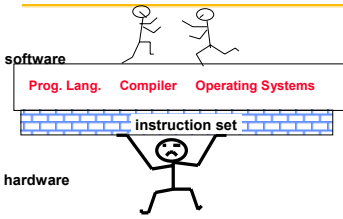
EECS 252 Graduate Computer Architecture

Lec 4 – Issues in Basic Pipelines (stalls, exceptions, branch prediction)

David Culler
Electrical Engineering and Computer Sciences
University of California, Berkeley

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

Debate Review: ISA -a Critical Interface



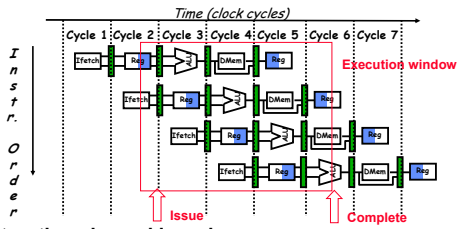
- Properties of a good abstraction
  - Lasts through many generations (portability)
  - Used in many different ways (generality)
  - Provides convenient functionality to higher levels
  - Permits an efficient implementation at lower levels

- Extremely well defined abstraction
- Huge, quantitative base of *usage data* for real applications filtered through SOA compiler technology
- Huge quantitative base of implementation *costs and performance*
- Convergence trend – enable optimizations, support HLL, OS support, contain complexity
- Lots of marketing (ignores, misuses, or selective use of established data)
- Worse when we get beyond scalar operations
- Translation / Interpretation boundary becoming less sharp

Discussion Exercise

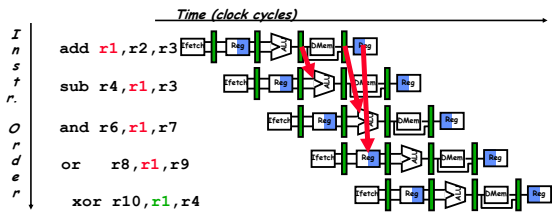
- In terms of the 'iron triangle" what are the performance implications of condition-codes?

Ordering Properties of basic inst. pipeline

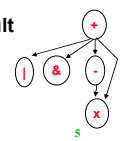


- Instructions issued in order
- Operand fetch is stage 2 => operand fetched in order
- Write back in stage 5 => no WAW, no WAR hazards
- Common pipeline flow => operands complete in order
- Stage changes only at "end of instruction"

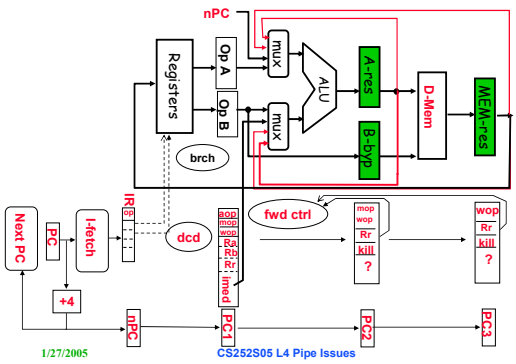
What does forwarding do?



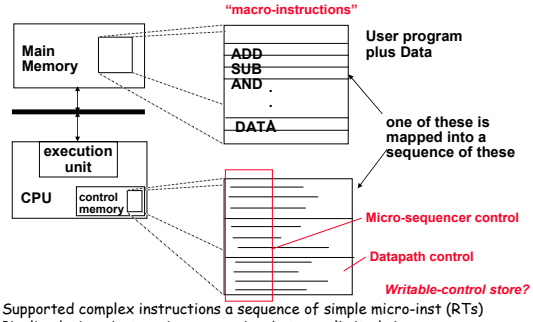
- Destination register is a name for instr's result
- Source registers are names for sources
- Forwarding logic builds data dependence (flow) graph for instructions in the execution window



Control Pipeline

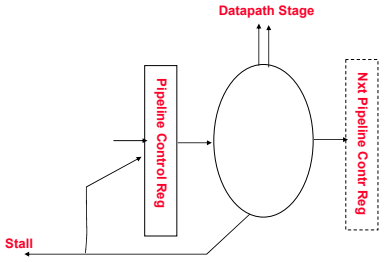


Historical Perspective: Microprogramming



Supported complex instructions a sequence of simple micro-inst (RTs)
Pipelined micro-instruction processing, but very limited view.
Could not reorganize macroinstructions to enable pipelining

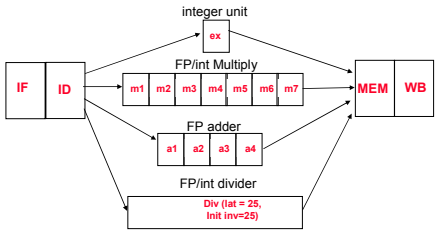
Multicycle stages



- Stage microsequencer spits micro-ops into the pipe

Typical "simple" Pipeline

- Example: MIPS R4000



Branch prediction

- Datapath parallelism only useful if you can keep it fed.
- Easy to fetch multiple (consecutive) instructions per cycle
  - essentially speculating on sequential flow
- Jump: unconditional change of control flow
  - Always taken
- Branch: conditional change of control flow
  - Taken about 50% of the time
  - Backward: 30% x 80% taken
  - Forward: 70% x 40% taken

A Big Idea for Today

- Reactive: past actions cause system to adapt use
  - do what you did before better
  - ex: caches
  - TCP windows
  - URL completion, ...
- Proactive: uses past actions to predict future actions
  - optimize speculatively, anticipate what you are about to do
  - branch prediction
  - long cache blocks
  - ???

Case for Branch Prediction when Issue N instructions per clock cycle

1. Branches will arrive up to *n* times faster in an *n*-issue processor
2. Amdahl's Law => relative impact of the control stalls will be larger with the lower potential CPI in an *n*-issue processor

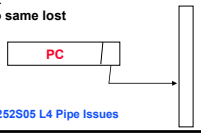
conversely, need branch prediction to 'see' potential parallelism

Branch Prediction Schemes

0. Static Branch Prediction
- 1-bit Branch-Prediction Buffer
- 2-bit Branch-Prediction Buffer
- Correlating Branch Prediction Buffer
- Tournament Branch Predictor
- Branch Target Buffer
- Integrated Instruction Fetch Units
- Return Address Predictors

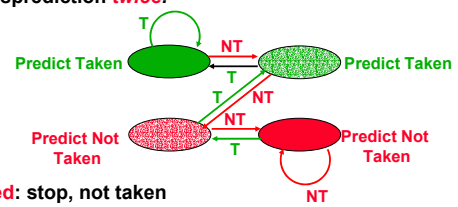
Dynamic Branch Prediction

- Performance = *ƒ*(accuracy, cost of misprediction)
- Branch History Table: Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check
    - » saves HW, but may not be right branch
  - If inst == BR, update table with outcome
- Problem: in a loop, 1-bit BHT will cause 2 mispredictions
  - 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
  - avg is 9 iterations before exit
  - Only 80% accuracy even if loop 90% of the time
- Local history
  - This particular branch inst
    - » Or one that maps into same lost



2-bit Dynamic Branch Prediction
(J. Smith, 1981)

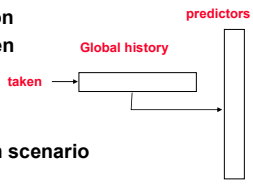
- 2-bit scheme where change prediction only if get misprediction *twice*:



- Red: stop, not taken
- Green: go, taken
- Adds *hysteresis* to decision making process
- Generalize to n-bit saturating counter

Consider 3 Scenarios

- Branch for loop test
- Check for error or exception
- Alternating taken / not-taken
  - example?

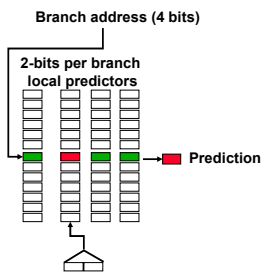


- Your worst-case prediction scenario
- How could HW predict "this loop will execute 3 times" using a simple mechanism?

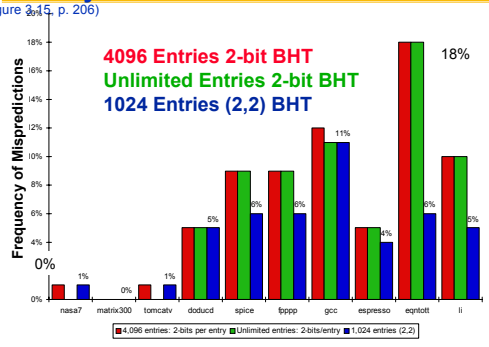
Correlating Branches

Idea: taken/not taken of recently executed branches is related to behavior of next branch (as well as the history of that branch behavior)
- Then behavior of recent branches selects between, say, 4 predictions of next branch, updating just that prediction

- (2,2) predictor: 2-bit global, 2-bit local



Accuracy of Different Schemes
(Figure 3.15, p. 206)



What's missing in this picture?

Re-evaluating Correlation

- Several of the SPEC benchmarks have less than a dozen branches responsible for 90% of taken branches:

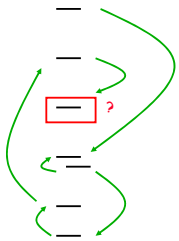
program	branch %	static	# = 90%
compress	14%	236	13
eqntott	25%	494	5
gcc	15%	9531	2020
mpeg	10%	5598	532
real gcc	13%	17361	3214

- Real programs + OS more like gcc
- Small benefits beyond benchmarks for correlation? problems with branch aliases?

BHT Accuracy

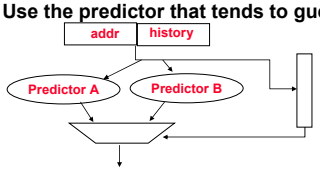
- Mispredict because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table
- 4096 entry table programs vary from 1% misprediction (nasa7, tomcatv) to 18% (eqntott), with spice at 9% and gcc at 12%
- For SPEC92, 4096 about as good as infinite table

Dynamically finding structure in Spaghetti



Tournament Predictors

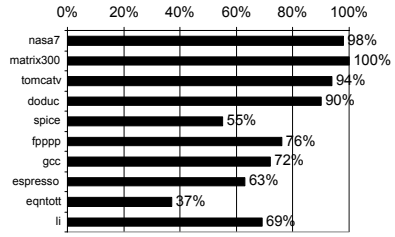
- Motivation for correlating branch predictors is 2-bit predictor failed on important branches; by adding global information, performance improved
- Tournament predictors: use 2 predictors, 1 based on global information and 1 based on local information, and combine with a selector
- Use the predictor that tends to guess correctly



Tournament Predictor in Alpha 21264

- 4K 2-bit counters to choose from among a global predictor and a local predictor
- Global predictor also has 4K entries and is indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor
  - 12-bit pattern: ith bit 0 => ith prior branch not taken; ith bit 1 => ith prior branch taken;
- Local predictor consists of a 2-level predictor:
  - Top level a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. 10-bit history allows patterns 10 branches to be discovered and predicted.
  - Next level Selected entry from the local history table is used to index a table of 1K entries consisting a 3-bit saturating counters, which provide the local prediction
- Total size: 4K*2 + 4K*2 + 1K*10 + 1K*3 = 29K bits! (~180,000 transistors)

% of predictions from local predictor in Tournament Prediction Scheme



Accuracy of Branch Prediction

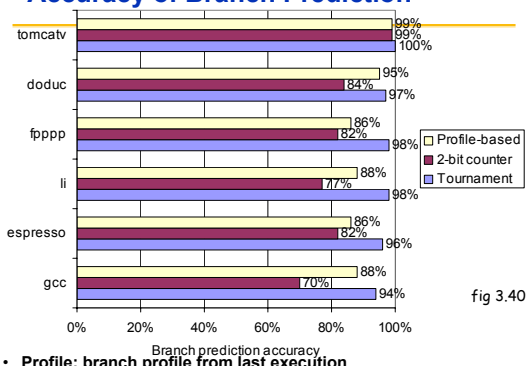
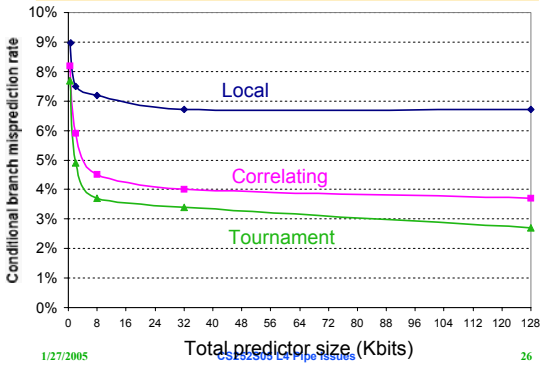


fig 3.40

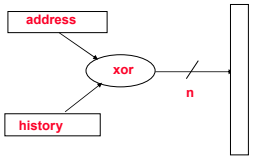
- Profile: branch profile from last execution (static in that in encoded in instruction, but profile)

Accuracy v. Size (SPEC89)



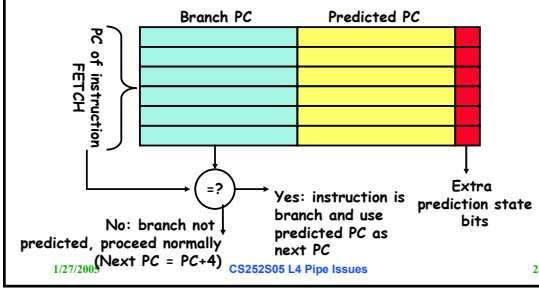
GSHARE

- A good simple predictor
- Sprays predictions for a given address across a large table for different histories



Need Address at Same Time as Prediction

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can't use wrong branch address (Figure 3.19, 3.20)

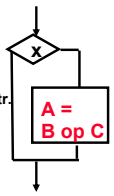


Predicated Execution

- Avoid branch prediction by turning branches into conditionally executed instructions:

  if (x) then A = B op C else NOP

  - If false, then neither store result nor cause exception
  - Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can annul any following instr.
  - IA-64: 64 1-bit condition fields selected so conditional execution of any instruction
  - This transformation is called "if-conversion"



- Drawbacks to conditional instructions
  - Still takes a clock even if "annulled"
  - Stall if condition evaluated late
  - Complex conditions reduce effectiveness; condition becomes known late in pipeline

Special Case Return Addresses

- Register Indirect branch hard to predict address
- SPEC89 85% such branches for procedure return
- Since stack discipline for procedures, save return address in small buffer that acts like a stack: 8 to 16 entries has small miss rate

Pitfall: Sometimes bigger and dumber is better

- 21264 uses tournament predictor (29 Kbits)
- Earlier 21164 uses a simple 2-bit predictor with 2K entries (or a total of 4 Kbits)
- SPEC95 benchmarks, 22264 outperforms
  - 21264 avg. 11.5 mispredictions per 1000 instructions
  - 21164 avg. 16.5 mispredictions per 1000 instructions
- Reversed for transaction processing (TP) !
  - 21264 avg. 17 mispredictions per 1000 instructions
  - 21164 avg. 15 mispredictions per 1000 instructions
- TP code much larger & 21164 hold 2X branch predictions based on local behavior (2K vs. 1K local predictor in the 21264)

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

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 Summary

- Prediction becoming important part of scalar execution
- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch.
  - Either different branches
  - Or different executions of same branches
- Tournament Predictor: more resources to competitive solutions and pick between them
- Branch Target Buffer: include branch address & prediction
- Predicated Execution can reduce number of branches, number of mispredicted branches
- Return address stack for prediction of indirect jump

Adminstrative Issues

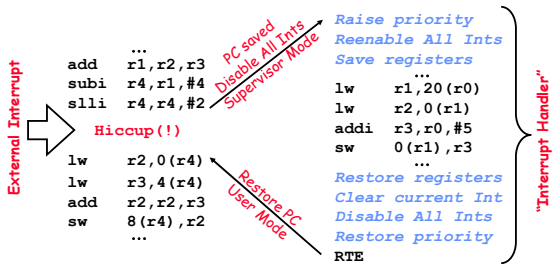
- HW 1 posted, Due in a week. Work in Pairs
- Readings for Tuesday: Cray 1, CDC 6600
  - Bring a question on each

Exceptions and Interrupts

(Hardware)

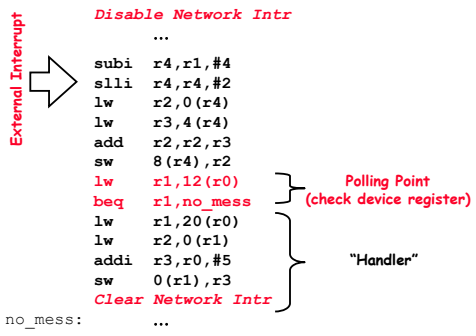
Example: Device Interrupt

(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.

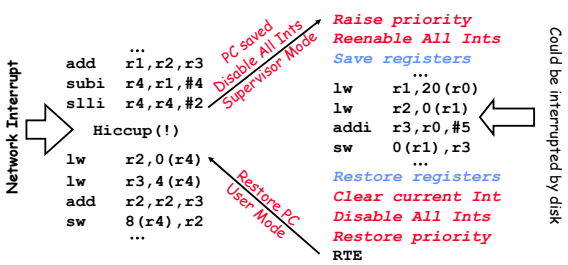
Exception/Interrupt classifications

- *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 Priorities Must be Handled



Note that priority must be raised to avoid recursive interrupts!

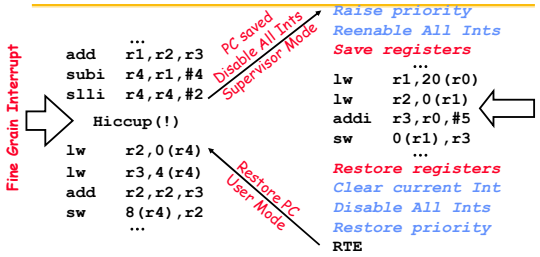
Interrupt controller hardware and mask levels

- 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?



- 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
```

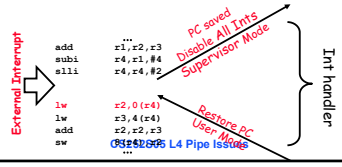
  - OS "System Calls" are just trap instructions:

```
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



Precise interrupt point may require multiple PCs

```
      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)

- 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 $x \rightarrow 0$
    $$f(0) = \frac{0}{0} \Rightarrow NaN + illegal\_operation$$

    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.

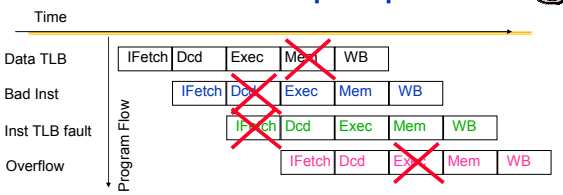
```
<float 1>
<int 1>
<int 2>
<int 3>
<float 2>
<int 4>
<int 5>
```

- 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 ⇒ 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 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.

Summary

- 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