Lecture # 6

****
Review -- 1 min
****
<good news/bad news>

Pipeline Speedup Equation

/* # stages / (1+stalls) * change in clock rate) */

3 Hazards Limit Performance

structural -- just build more hardware

data -- fundamental limit -- data dependencies

control -- branches, etc

Data Hazards

RAW – instruction i writes register then instruction j reads it
(j had better see i's write!)

-- an issue in DLX pipeline --> result forwarding, load stalls

aka *true dependence*

WAW -- instruction i writes register then instruction j writes it
(j's write had better win)

Not a problem in DLX pipeline (comes up later)

Not a fundamental problem (if we had infinite registers,
we would never have to put 2 results in same register)

aka *output dependence*

WAR -- instruction i reads register then instruction j writes it
(i had better see old value)

Again, not a fundamental problem (if we had infinite
registers)

aka *antidependence*

****
Outline - 1 min

****
Pipeline Complications:

Control Hazards/Branches

Interrupts

****
Lecture - 20 min
****
Control Hazards/Branches

<picture of basic DLX data path>

Note critical path on branch

branch	IF	ID	EX	MEM	WB					
i+1		IF	stall	stall	IF	ID	EX	MEM	WB	
i+2						IF	ID	EX	MEM	WB
...										

3 cycle stall!

If CPI = 1
30% branch
Stall 3 cycles --> new CPI = 1.9

Solution
----

1. Determine branch taken/not taken sooner

and 2. compute taken branch address earlier

DLX Implementation
----

1. Determine branch taken/not taken sooner
    --> dlx branches test register against 0
    --> add comparitor to ID/RF stage
2. Determine branch address earlier
    --> adder to calculate new address in ID/RF stage

<picture of new DLX data path>

New Result

Branch	IF	ID	EX	MEM	WB		
i+1		IF	**IF**	ID	EX	MEM	WB

Penalty: 1 cycle per branch

We can do better
----
4 options
1. Stall (as above)
2. Predict Branch Not Taken
    execute successor instructions in sequence
    "Squash" instructions in pipeline if taken
        (Works B/C update state in WB stage -- late)

    47% of DLX branches not taken ("avg program")

    Key idea: PC+4 already known, so use it

3. Predict Taken
    53% taken on avg

    BUT -- doesn't help us
        (don't know target address until ID completes)
        --> DLX still has 1 cycle penalty

    In some machines, branch target known before outcome

(e.g. complex branch conditions)

4. Delayed Branch

Define branch to take place AFTER a following instruction

branch instruction
successor in delay slot ALWAYS EXECUTED
next instruction (executed only if branch not taken)

In general N delay slots
(N = 1 always, though)
DLX uses N=1

If we can fill delay slot with SAFE instruction
then branch penalty = 0

Problem -- hides branch that costs 1 instruction
What if we build a machine where branch costs 3 instructions?

(kinda weird)

****
Admin - 3 min
****
leading discussion
HW2
Project topic

****
Lecture - 24 min
****
Delayed branch -- compiler issues

Must fill delay slot with SAFE instruction

(always OK to execute)

Get instruction from:
o before branch instruction
o target address
(only useful if branch taken
must be safe if not taken)
o from fall through
(only useful if not taken;
must be safe if taken)
o NO OP
(always useless, always safe)

2 useful ideas:

Cancelling branch -- allows squashing
of "unsafe, useless" instruction
--> makes it easier to schedule branch

compiler "prediction" taken/not taken
suppose you have 2 branch candidates
one from fall through, one from target

if branch is backwards, predict taken --> take target
(think loops)

if branch is forwards, predict not taken --> take dest

Also, profile-based compilers -- run program and
keep statistics at each branch.

Bottom line -- branches

$$\text{Pipeline Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Branch Fequency} * \text{Branch Penalty}}$$

QUESTION: Assumes clock frequency unchanged. Fair?

branch frequency = 14%
Fraction taken = 65%
half of delay slots usefully filled

Scheduling Scheme	Branch penalty (taken/not)	CPI	speedup v. stall
Stall (orig)	3/3	1.42	1.0
stall(new)/predict taken	1/1	1.14	1.26
predict not taken	1/0	1.09	1.29
delayed branch	~0.5	1.07	1.31

Interrupts and Exceptions

example -- page fault, arithmetic exception

Goal
- save state of processor
- handle exception
- restore state of processor

In DLX
1. force trap "instruction" into pipeline
   store PC of faulting instruction
   "squash" faulting instruction and all later instructions in pipeline

--> make them NO-OPs
--> not allowed to change state

2. when trap reaches WB, jump to exception-handling routine
3. exception handling routine saves PC of faulting instruction
4. handle exception
5. load PC with PC of faulting instruction and resume

Subtleties

1. with delayed branch PC of faulting instruction not enough
   (what if the instruction in delay slot faults?)
   --> need N PC’s for N delay slots

2. works b/c state changes in WB
   what if update state in different stages

   Ugh!
   1. "roll out" of changes
      e.g. history buffer
   2. re-start in middle of instruction
      e.g. VAX move string instruction
   3. get wrong answer
      ("It’s OK b/c we’re getting an exception
      and aborting computation anyway")
      e.g. imprecise floating point exceptions
      (Alpha 21064, Power-2, MIPS R8000)

   4. serialize state changes
      e.g. "precise mode" on modern RISCs
      can be > 10x slower than "imprecise mode"

3. Handle Exceptions in program order

ex (WARNING: somewhat contrived to illustrate point in DLX pipeline)

BNE	IF	ID	EX	MEM	WB			<branch instruction>
LW		IF	ID	EX	MEM	WB		<delay slot>
ADD			IF	ID	EX	MEM	WB	<target instruction>

Suppose I am debugging

and I’ve got a lousy program

The branch is, in fact to an illegal address (page fault -- branch to non-existant page)

The load word is loading an illegal address

ADD instruction page faults on cycle 3

LW page faults on cycle 5

--> confuse me when I’m debugging!

Solution -- keep status vector with all instructions in pipe

handle exceptions as instructions leave pipeline

****

Summay - 1 min

****