Lecture 19: Pipelining

• Today’s topics:
  - Hazards and instruction scheduling
  - Branch prediction
  - Out-of-order execution
Problem 0

add $1, $2, $3
add $5, $1, $4

Without bypassing:
add $1, $2, $3: IF DR AL DM RW
add $5, $1, $4: IF DR DR DR AL DM RW

With bypassing:
add $1, $2, $3: IF DR AL DM RW
add $5, $1, $4: IF DR AL DM RW

Point of Production
Point of Consumption
Problem 1

add $1, $2, $3

lw $4, 8($1)
Problem 1

add $1, $2, $3

lw $4, 8($1)
Problem 2

lw $1, 8($2)

lw $4, 8($1)
Problem 2

lw $1, 8($2)

lw $4, 8($1)
Problem 3

lw  $1, 8($2)

sw  $1, 8($3)
Problem 3

lw $1, 8($2)

sw $1, 8($3)
Problem 4

A 7 or 9 stage pipeline, RR and RW take an entire stage

lw $1, 8($2)

add $4, $1, $3
Problem 4

A 7 or 9 stage pipeline, RR and RW take an entire stage

lw $1, 8($2)
add $4, $1, $3
Problem 4

Without bypassing: 4 stalls

With bypassing: 2 stalls

lw $1, 8($2)
add $4, $1, $3
Control Hazards

• Simple techniques to handle control hazard stalls:
   for every branch, introduce a stall cycle (note: every 6th instruction is a branch!)
   assume the branch is not taken and start fetching the next instruction – if the branch is taken, need hardware to cancel the effect of the wrong-path instruction
   fetch the next instruction (branch delay slot) and execute it anyway – if the instruction turns out to be on the correct path, useful work was done – if the instruction turns out to be on the wrong path, hopefully program state is not lost
   make a smarter guess and fetch instructions from the expected target
Branch Delay Slots

a. From before

```
add $s1, $s2, $s3
if $s2 = 0 then
  Delay slot

Becomes

if $s2 = 0 then
  add $s1, $s2, $s3
```

b. From target

```
sub $t4, $t5, $t6
...
add $s1, $s2, $s3
if $s1 = 0 then
  Delay slot

sub $t4, $t5, $t6
```

Source: H&P textbook
Pipeline without Branch Predictor

PC \rightarrow IF (br) \rightarrow Reg Read Compare Br-target \rightarrow PC + 4
Pipeline with Branch Predictor

PC -> IF (br) -> Branch Predictor -> Reg Read Compare Br-target
2-Bit Prediction

- For each branch, maintain a 2-bit saturating counter:
  - if the branch is taken: counter = min(3,counter+1)
  - if the branch is not taken: counter = max(0,counter-1)
  ... sound familiar?

- If (counter >= 2), predict taken, else predict not taken

- The counter attempts to capture the common case for each branch