Dependences and Hazards

Lecture 17
CS301
• Daily Review of today’s lecture
  ✷ Due tomorrow (10/30) at 8am
• HW #7 due today at 5pm
• HW #8 assigned
  ✷ Due 10/5 at 5pm
• Read Chapter 4.8–4.9
Data Dependencies

• We want to keep the pipeline completing an instruction every cycle
• When a later instruction depends on the result of an earlier instruction, stalls happen
• There are 3 types of data dependencies that we’ve been talking about:
  - RAW
  - WAR
  - WAW
add $t0, $s0, $s1
sub $t2, $t0, $s3
or $s3, $t7, $s2
mult $t2, $t7, $s0
add $t0, $s0, $s1
sub $t2, $t0, $s3
or  $s3, $t7, $s2
mult $t2, $t7, $s0
WAW – Write after Write

add $t0, $s0, $s1
sub $t2, $t0, $s3
or $s3, $t7, $s2
mult $t2, $t7, $s0
Identify all of the dependencies

add $t0, $s0, $s1
sub $t2, $t0, $s3
or $s3, $t7, $s2
mul $t2, $t7, $s0
Identify all of the dependencies

RAW
WAR
WAW

add $t0, $s0, $s1
sub $t2, $t0, $s3
or $s3, $t7, $s2
mul $t2, $t7, $s0
Identify all of the dependencies

RAW
WAR
WAW

add $t0, $s0, $s1
sub $t2, $t0, $s3
or $s3, $t7, $s2
mul $t2, $t7, $s0
Identify all of the dependences

RAW
WAR
WAW

add $t0, $s0, $s1
sub $t2, $t0, $s3
or $s3, $t7, $s2
mul $t2, $t7, $s0
Which dependences can cause hazards? (stalls)

RAW
WAR
WAW

add $t0, $s0, $s1
sub $t2, $t0, $s3
or $s3, $t7, $s2
mul $t2, $t7, $s0
Which dependences can cause hazards? (stalls)

RAW
WAR
WAW

add $t0, $s0, $s1
sub $t2, $t0, $s3
or $s3, $t7, $s2
mul $t2, $t7, $s0
Which dependences can cause hazards? (stalls)

RAW  Yes  True Dependency  WAR  No
WAR  
WAW  

add $t0, $s0, $s1
sub $t2, $t0, $s3
or $s3, $t7, $s2
mul $t2, $t7, $s0
Which dependences can cause hazards? (stalls)

RAW: Yes
WAR: No
WAW: No

add $t0, $s0, $s1
sub $t2, $t0, $s3
or $s3, $t7, $s2
mul $t2, $t7, $s0

True Dependency
Which dependences can cause hazards? (stalls)

- RAW: Yes
- WAR: No
- WAW: No

How do we solve data hazards?

- add $t0$, $s0$, $s1$
- sub $t2$, $t0$, $s3$
- or $s3$, $t7$, $s2$
- mul $t2$, $t7$, $s0$
Which dependences can cause hazards? (stalls)

- RAW: Yes
- WAR: No
- WAW: No

True Dependence

How do we solve data hazards? Instruction Reordering

```
add $t0, $s0, $s1
sub $t2, $t0, $s3
or $s3, $t7, $s2
mul $t2, $t7, $s0
```
Let’s reorder the or

RAW
WAR
WAW

add $t0, $s0, $s1
sub $t2, $t0, $s3
or $s3, $t7, $s2
mul $t2, $t7, $s0
Let’s reorder the or

RAW
WAR
WAW

add $t0, $s0, $s1
or $s3, $t7, $s2
sub $t2, $t0, $s3
mul $t2, $t7, $s0
Let’s reorder the or

Aaaaaaaah! The result of the or will be passed to the sub!!!!!!!
Let’s reorder the mul

RAW
WAR
WAW

add $t0, $s0, $s1
sub $t2, $t0, $s3
or $s3, $t7, $s2
mul $t2, $t7, $s0
Let’s reorder the mul

RAW
WAR
WAW

Let’s reorder the mul

RAW
WAR
WAW

add $t0, $s0, $s1
mul $t2, $t7, $s0
sub $t2, $t0, $s3
or $s3, $t7, $s2
Let’s reorder the mul

RAW
WAR
WAW

Aaaaaaah! $t2$ will be left with the result of the sub, not mult!

add $t0$, $s0$, $s1$

mul $t2$, $t7$, $s0$

sub $t2$, $t0$, $s3$

or $s3$, $t7$, $s2$
Why do we care about WAW,WAR?
Why do we care about WAW, WAR?

- WAR and WAW prevent instruction reordering
How to remove WAR, WAW dependences?

add $t0, $s0, $s1
sub $t2, $t0, $s3
or $s3, $t7, $s2
mul $t2, $t7, $s0
and $t4, $s3, $s5
add $s3, $s4, $s6
Register Renaming
use a different register for that result
(and all subsequent uses of that result)

add $t0, $s0, $s1
sub $t5, $t0, $s3
or $t6, $t7, $s2
mul $t2, $t7, $s0
and $t4, $t6, $s5
add $s3, $s4, $s6
Who renames registers?

- Static register renaming
- Dynamic register renaming
Who renames registers?

- Static register renaming
  - Compiler
  - Compiler is the one who makes assignments in the first place!
  - Number of registers limited by……..

- Dynamic register renaming
  - Hardware
  - Can offer more registers –
  - Number of registers limited by…..
Who renames registers?

- Static register renaming
  - Compiler
  - Compiler is the one who makes assignments in the first place!
  - Number of registers limited by Instruction format
- Dynamic register renaming
  - Hardware
  - Can offer more registers –
  - Number of registers limited by size of register file & clock rate
Minimizing Data Hazards
Minimizing Data Hazards

• Data Forwarding
Minimizing Data Hazards

- Data Forwarding
- Instruction Reordering
Summary

• What is the difference between a hazard and a dependence?

• How can we get rid of WAW/WAR dependences?

• What limits this solution?
Summary

• What is the difference between a hazard and a dependence?
  - A dependence prevents reordering
  - A hazard can cause a stall
  - Hazard $\rightarrow$ dependence, not always the converse

• How can we get rid of WAW/WAR dependences?

• What limits this solution?
Summary

- What is the difference between a hazard and a dependence?
  - A dependence prevents reordering
  - A hazard can cause a stall
  - Hazard $\rightarrow$ dependence, not always the converse

- How can we get rid of WAW/WAR dependences?
  - Register renaming

- What limits this solution?
Summary

- What is the difference between a hazard and a dependence?
  - A dependence prevents reordering
  - A hazard can cause a stall
  - Hazard $\rightarrow$ dependence, not always the converse

- How can we get rid of WAW/WAR dependences?
  - Register renaming

- What limits this solution?
  - The number of registers available (ISA or physical)
Control Dependences
In what cycle does the nextPC get calculated for the bne?
In what cycle does the or get fetched?

add $s5, $s4, $t1
bne $s0, $s1, end
or $s3, $s0, $t3
end: sw $s2, 0($t1)
In what cycle does the nextPC get calculated for the bne? End of 4
In what cycle does the or get fetched?

add $s5, $s4, $t1
bne $s0, $s1, end
or $s3, $s0, $t3
end: sw $s2, 0($t1)
In what cycle does the nextPC get calculated for the bne? End of 4
In what cycle does the or get fetched? Beginning of 3

add $s5, $s4, $t1
bne $s0, $s1, end
or $s3, $s0, $t3
end: sw $s2, 0($t1)
In what cycle does the nextPC get calculated for the bne? End of 4
In what cycle does the or get fetched? Beginning of 3

\[
\text{add } s5, s4, t1
\]
\[
bne s0, s1, end
\]
\[
or s3, s0, t3
\]
\[
end: sw s2, 0(t1)
\]
Barriers to Pipeline Performance

• Uneven stages
• Pipeline register delays
• Data Hazards
• Control Hazards
  ❖ *Whether* an instruction will execute depends on the outcome of a conditional branch still in the pipeline
In what cycle does the nextPC get calculated for the bne? End of 4
In what cycle does the or get fetched? Beginning of 3

add $s5, $s4, $t1
bne $s0, $s1, end
or $s3, $s0, $t3
end: sw $s2, 0($t1)
In what cycle does the nextPC get calculated for the bne? End of 4
In what cycle does the or get fetched? Beginning of 3

add $s5, $s4, $t1
bne $s0, $s1, end
or $s3, $s0, $t3
end: sw $s2, 0($t1)
In what cycle does the nextPC get calculated for the bne? 3
In what cycle does the or get fetched? 3

Solution 1: Add hardware to determine branch in decode stage

add $s5, $s4, $t1
bne $s0, $s1, end
or $s3, $s0, $t3
end: sw $s2, 0($t1)
Note

- For the rest of this course, the branches will be determined in the decode stage.
- All other optimizations will be in addition to moving branch calculation to decode stage.
Redefine the *semantics* of a branch: ALWAYS execute the instruction after the branch, regardless of the outcome of the branch.

```
add $s5, $s4, $t1
bne $s0, $s1, end
or $s3, $s0, $t3
end: sw $s2, 0($t1)
```

Solution 2: Branch Delay Slot
ALWAYS execute the instruction after the branch, regardless of the outcome of the branch. Try to fill that spot with an instruction from before the branch.

Solution 2: Also add Branch Delay Slot

```
bne $s0, $s1, end
or $s3, $s0, $t3
end: sw $s2, 0($t1)
```
Branch Delay Slot

- The hardware always executes instruction after a branch.
- The compiler tries to take an instruction from before branch and move it after branch.
- If it can find no instruction, it inserts a *nop* after the branch.
- If it forgets to place *nop* or *inst* there, you can get incorrect execution!!!!!
Branch Delay Slot – Limitations

- If you have a machine with 20 pipeline stages, and it takes 10 stages to calculate branch, how many branch delay slots are there?
- Can you move any instruction into branch delay slot?
- What happens as the pipeline gets deeper?
Branch Delay Slot – Limitations

• If you have a machine with 20 pipeline stages, and it takes 10 stages to calculate branch, how many branch delay slots are there? 9
• Can you move any instruction into branch delay slot?
• What happens as the pipeline gets deeper?
Branch Delay Slot – Limitations

- If you have a machine with 20 pipeline stages, and it takes 10 stages to calculate branch, how many branch delay slots are there? 9
- Can you move any instruction into branch delay slot? Only independent instructions
- What happens as the pipeline gets deeper?
Branch Delay Slot – Limitations

• If you have a machine with 20 pipeline stages, and it takes 10 stages to calculate branch, how many branch delay slots are there? 9

• Can you move any instruction into branch delay slot? Only independent instructions

• What happens as the pipeline gets deeper? More difficult to fill slots

• Branch delay slot is only used in short pipelines!
Solution 3: Branch Prediction

Guess which way the branch will go before calculation occurs. Clean up if predictor is wrong.

add $s5, $s4, $t1
bne $s0, $s1, end
or $s3, $s0, $t3
end: sw $s2, 0($t1)
First: Always predict not taken
If we are right, how many cycles do we stall?

add $s5, $s4, $t1
bne $s0, $s1, end
or $s3, $s0, $t3
end: sw $s2, 0($t1)
First: Always predict not taken
If we are right, how many cycles do we stall? 0

add $s5, $s4, $t1
bne $s0, $s1, end
or $s3, $s0, $t3
end: sw $s2, 0($t1)
Solution 3: Branch Prediction

First: Always predict not taken
If we are wrong, then flush incorrect instruction(s)

add $s5, $s4, $t1
bne $s0, $s1, end
or $s3, $s0, $t3
end: sw $s2, 0($t1)
First: Always predict not taken
If we are wrong, then flush incorrect instruction(s)
How many cycles do we stall?

add $s5, $s4, $t1
bne $s0, $s1, end
or $s3, $s0, $t3
end: sw $s2, 0($t1)
Solution 3: Branch Prediction

First: Always predict not taken
If we are wrong, then flush incorrect instruction(s)
How many cycles do we stall? 1

1. add $s5, $s4, $t1
2. bne $s0, $s1, end
3. or $s3, $s0, $t3
4. end: sw $s2, 0($t1)
Solution 3: Branch Prediction

First: Always predict taken
Why will this still result in a stall?

add $s5, $s4, $t1
bne $s0, $s1, end
end: sw $s2, 0($t1)
Branch Prediction

- If we’re going to predict taken, we need to know where to branch to earlier than when we determine where the branch actually goes to.
  - How?
Branch Prediction

- Understand the nature of programs
- Are branch directions random?
- If not, what will correlate?
  - Past behavior?
  - Previous branches’ behavior?
Branch Prediction

```assembly
slt $t1, $s2, $s3
beq $t1, $0, end

loop:  do some work
       addi $s2, $s2, 1
       slt $t1, $s2, $s3
       bne $t1, $0, loop

end:
```

Is beq often taken or not taken?
Is bne often taken or not taken?

for(i; i<n;i++)
do some work
Branch Prediction

```assembly
slt $t1, $s2, $s3
beq $t1, $0, end

loop: do some work

addi $s2, $s2, 1
slt $t1, $s2, $s3
bne $t1, $0, loop

end:
```

for(i; i<n;i++)
  do some work

Is `beq` often taken or not taken?  
Not Taken

Is `bne` often taken or not taken?
**Branch Prediction**

```assembly
slt $t1, $s2, $s3
beq $t1, $0, end
loop:  do some work

addi $s2, $s2, 1
slt $t1, $s2, $s3
bne $t1, $0, loop

end:
```

for(i; i<n;i++)
    do some work

**Is beq often taken or not taken?**  Not Taken
**Is bne often taken or not taken?**  Taken

**Conclusion:** We want a prediction that is unique to each branch. Look up prediction by PC
First Branch Predictor

Predict whatever happened last time
Update the predictor for next time

Predict Taken  |  Predict Not Taken
First Branch Predictor

Predict whatever happened last time
Update the predictor for next time

T
Predict Taken

NT
Predict Not Taken
First Branch Predictor

Predict whatever happened last time
Update the predictor for next time

Predict Taken Predict Not Taken

T NT NT

1 0
Branch Prediction

```
slt $t1, $s2, $s3
beq $t1, $0, end
loop: do some work
    addi $s2, $s2, 1
    slt $t1, $s2, $s3
    bne $t1, $0, loop
end:
```

```
for(i; i<n;i++)
    do some work
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NextState</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branch Prediction

```
slt $t1, $s2, $s3
beq $t1, $0, end
loop:    do some work
        addi $s2, $s2, 1
        slt $t1, $s2, $s3
        bne $t1, $0, loop
end:
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction</td>
<td>NT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reality</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NextState</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branch Prediction

```
slt $t1, $s2, $s3
beq $t1, $0, end
loop:  do some work
       addi $s2, $s2, 1
       slt $t1, $s2, $s3
       bne $t1, $0, loop
end:
for(i; i<n;i++)
   do some work
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reality</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NextState</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
slt $t1, $s2, $s3
beq $t1, $0, end
loop: do some work
addi $s2, $s2, 1
slt $t1, $s2, $s3
bne $t1, $0, loop
end:

for(i; i<n;i++)
do some work

Iteration     1  2  ...  x  1  2  ...  y
CurState      0  1  1  0
Prediction   NT T  T
Reality       T  T  NT
NextState     1  1  0
```
Branch Prediction

```
slt $t1, $s2, $s3
beq $t1, $0, end

loop:
do some work
addi $s2, $s2, 1
slt $t1, $s2, $s3
bne $t1, $0, loop

end:
```

```
for(i; i<n;i++)
do some work
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction</td>
<td>NT</td>
<td>T</td>
<td></td>
<td>T</td>
<td>NT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reality</td>
<td>T</td>
<td>T</td>
<td></td>
<td>NT</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NextState</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Branch Prediction

```
    slt $t1, $s2, $s3
    beq $t1, $0, end
    loop: do some work
        addi $s2, $s2, 1
        slt $t1, $s2, $s3
        bne $t1, $0, loop
    end:
```

```
for(i; i<n;i++)
do some work
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Prediction</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>Reality</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>NextState</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Branch Prediction

When are we wrong?????

```assembly
slt $t1, $s2, $s3
beq $t1, $0, end
loop: do some work
addi $s2, $s2, 1
slt $t1, $s2, $s3
bne $t1, $0, loop
end:
```

```plaintext
for(i; i<n;i++)
do some work
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>…</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>…</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Prediction</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>Reality</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NextState</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
### Branch Prediction

When are we wrong????? First and last iteration of each loop

```assembly
slt $t1, $s2, $s3
beq $t1, $0, end
loop:
  do some work
  addi $s2, $s2, 1
  slt $t1, $s2, $s3
  bne $t1, $0, loop
end:
```

```python
for(i; i<n;i++)
    do some work
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>…</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>…</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Prediction</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>Reality</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td></td>
</tr>
<tr>
<td>NextState</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

When are we wrong????? First and last iteration of each loop

---

When are we wrong?????

First and last iteration of each loop

```assembly
slt $t1, $s2, $s3
beq $t1, $0, end
loop:
  do some work
  addi $s2, $s2, 1
  slt $t1, $s2, $s3
  bne $t1, $0, loop
end:
```
Two-bit Branch Predictor

Must be wrong twice in a row to switch prediction
Update the predictor for next time
One wrong→ state 1 or 2, No wrong → state 0 or 3

Predict Taken

Predict Not Taken
Two-bit Branch Predictor

Must be wrong *twice in a row* to switch prediction
Update the predictor for next time
One wrong-> state 1 or 2, No wrong -> state 0 or 3

Predict Taken

Predict Not Taken
Two-bit Branch Predictor

Must be wrong twice in a row to switch prediction.
Update the predictor for next time.
One wrong -> state 1 or 2, No wrong -> state 0 or 3.

Predict Taken

Predict Not Taken
Two-bit Branch Predictor

Must be wrong twice in a row to switch prediction
Update the predictor for next time
One wrong -> state 1 or 2, No wrong -> state 0 or 3

Predict Taken

Predict Not Taken
Second Branch Predictor

Must be wrong **twice in a row** to switch prediction
Update the predictor for next time

Predict Taken

Predict Not Taken
Branch Prediction

```assembly
slt $t1, $s2, $s3
beq $t1, $0, end

loop: do some work
addi $s2, $s2, 1
slt $t1, $s2, $s3
bne $t1, $0, loop

end:
```

for(i; i<n;i++)
do some work

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>…</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>…</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NextState</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branch Prediction

```
slt $t1, $s2, $s3  for(i; i<n;i++)
beq $t1, $0, end
    do some work
loop: do some work
    addi $s2, $s2, 1
    slt $t1, $s2, $s3
    bne $t1, $0, loop
end:
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>…</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>…</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reality</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NextState</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branch Prediction

```
slt $t1, $s2, $s3
beq $t1, $0, end

loop:  do some work
  addi $s2, $s2, 1
  slt $t1, $s2, $s3
  bne $t1, $0, loop

end:
```

```
for(i; i<n;i++)
  do some work
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>2</td>
<td>3</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction</td>
<td>T</td>
<td>T</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reality</td>
<td>T</td>
<td>T</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NextState</td>
<td>3</td>
<td>3</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Branch Prediction

```assembly
slt $t1, $s2, $s3
beq $t1, $0, end

loop: do some work

addi $s2, $s2, 1
slt $t1, $s2, $s3
bne $t1, $0, loop

end:
```

```c
for(i; i<n;i++)
do some work
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>…</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>…</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>2</td>
<td>3</td>
<td></td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction</td>
<td>T</td>
<td>T</td>
<td></td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reality</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NextState</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branch Prediction

```
slt $t1, $s2, $s3
beq $t1, $0, end

loop: do some work
    addi $s2, $s2, 1
    slt $t1, $s2, $s3
    bne $t1, $0, loop

end:
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>2</td>
<td>3</td>
<td></td>
<td>3</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prediction</td>
<td>T</td>
<td>T</td>
<td></td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reality</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NextState</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branch Prediction

\[
\text{slt } \$t1, \$s2, \$s3 \\
\text{beq } \$t1, 0, \text{end}
\]

loop: do some work

\[
\text{addi } \$s2, \$s2, 1 \\
\text{slt } \$t1, \$s2, \$s3 \\
\text{bne } \$t1, 0, \text{loop}
\]

end:

\[
\text{for(i; i<n;i++)} \\
\text{do some work}
\]

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>2</td>
<td>3</td>
<td></td>
<td>3</td>
<td>2</td>
<td>3</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Prediction</td>
<td>T</td>
<td>T</td>
<td></td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>Reality</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>NextState</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
When are we wrong?????

```
slt $t1, $s2, $s3
beq $t1, $0, end
loop: do some work

addi $s2, $s2, 1
slt $t1, $s2, $s3
bne $t1, $0, loop
end:
```

```
for(i; i<n; i++)
do some work
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Prediction</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Reality</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td></td>
</tr>
<tr>
<td>NextState</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
When are we wrong????? Only when we exit the loop
for(i; i<n;i++)
do some work

```
slt $t1, $s2, $s3
beq $t1, $0, end
loop:  do some work

addi $s2, $s2, 1
slt $t1, $s2, $s3
bne $t1, $0, loop

end:
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>x</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurState</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Prediction</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Reality</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td></td>
</tr>
<tr>
<td>NextState</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Simplest Branch Predictors

- Memory indexed by lower portion of address
- Entry contains few bits specifying prediction
- Accessed in IF stage so fetching of target occurs in next cycle
Real Branch Predictors

- TargetPC saved with predictor
- Limited space, so different branches may map to the same predictor
  - errors?
- Prediction based on past behavior of several branches
Advantages of Branch Prediction

- No extra instructions
- Highly predictable branches have no stalls
- Works well with loops.
- All hardware – no compiler necessary
Disadvantages/Limits of Branch Prediction

• Large penalty when wrong
  - Badly behaved branches kill performance
• Only a few can be performed each cycle (only a problem in multi-issue machines)
Minimizing Control Hazards
Minimizing Control Hazards

- Calculate branch in decode stage
Minimizing Control Hazards

- Calculate branch in decode stage
- Branch delay slot
Minimizing Control Hazards

- Calculate branch in decode stage
- Branch delay slot
- Branch prediction
CPI

- CPI = Σ((% instr) × (cycles))
- How do hazards affect CPI?
- How do branches affect CPI?
CPI

• CPI = \( \sum (\% \text{ instr} \times \text{cycles}) \)

• How do hazards affect CPI?
  - Arithmetic instructions’ cycle time increases

• How do branches affect CPI?
CPI

- CPI = \( \sum (\% \text{ instr} \times \text{cycles}) \)
- How do hazards affect CPI?
  - Arithmetic instructions’ cycle time increases
- How do branches affect CPI?
  - Branches’ cycle time increases
Summary of Optimizing Instruction Schedule

- Identify dependencies
- Draw timing diagram with data forwarding
- Move instructions between stalled instructions
  - This is reordering. You may need to do register renaming to do this.
- Reduce impact of control hazards if possible
  - Branch delay slot
Exceptions
What is an Exception?

- When there is an unexpected change in control flow, control switches to OS to handle
  - Examples: Divide by zero, arithmetic overflow, undefined instruction
Steps for Exceptions
Steps for Exceptions

- Detect exception
Steps for Exceptions

- Detect exception
- Place processor in state before offending instruction
Steps for Exceptions

- Detect exception
- Place processor in state before offending instruction
- Record exception type
Steps for Exceptions

- Detect exception
- Place processor in state before offending instruction
- Record exception type
- Record instruction’s PC in EPC
Steps for Exceptions

- Detect exception
- Place processor in state before offending instruction
- Record exception type
- Record instruction’s PC in EPC
- Transfer control to OS
How does pipelining affect exception-handling?
What happens if the third instruction is undefined?

In what stage is it detected?
In what cycle?

1. Detection

```
add $s0, $0, $0
lw $s1, 0($t0)
undefined
or $s3, $s4, $t3
```
What happens if the third instruction is undefined?

In what stage is it detected? **Decode**
In what cycle? 4

add $s0, $0, $0
lw $s1, 0($t0)
undefined
or $s3, $s4, $t3
1. Detection

- Must associate exception with proper instruction
- What happens if multiple exceptions happen in the same cycle?
  - Prioritize exceptions (earliest instructions have priority)
2. Preserve state before instruction

What? What does that mean?!?

- add $s0, $0, $0
- lw $s1, 0($t0)
- undefined
- or $s3, $s4, $t3
2. Preserve state before instruction

What? What does that mean?!? **Complete previous instructions, flush following instructions and do not let current write back**

- `add $s0, $0, $0`
- `lw $s1, 0($t0)`
- undefined
- or `$s3, $s4, $t3`
2. Preserve state before instruction

- add $s0, $0, $0
- lw $s1, 0($t0)
- undefined
- or $s3, $s4, $t3
3. Record exception type

- Place value in *cause register* or
- Use vectored interrupts
  - (exception routine address dependent on exception type)
4. Record nPC in EPC
Machine in detection cycle

or

Undef

lw

add
4. Record nPC in EPC

- Non-trivial because PC changes each cycle, and exceptions can be detected in several stages (decode, execute, memory)
- Precise exceptions
- Imprecise exceptions
4. Record PC in EPC

- Non-trivial because PC changes each cycle, and exceptions can be detected in several stages (decode, execute, memory)
- Precise exceptions figure out PC in hardware
- Imprecise exceptions let OS figure it out
5. Transfer control to OS

- Same as before