Pipeline hazards

- Even though we perfectly divide pipeline stages, it’s still hard to achieve CPI == 1.

- Pipeline hazards:
  - Structural hazard
    - The hardware does not allow two pipeline stages to work concurrently
  - Data hazard
    - A later instruction in a pipeline stage depends on the outcome of an earlier instruction in the pipeline
  - Control hazard
    - The processor is not clear about what’s the next instruction to fetch
Sol. of data hazard II: Forwarding

• Take the values, where ever they are!

```assembly
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```

10 cycles! CPI == 2 (Not optimal, but much better!)
Solution I: Delayed branches

LOOP: lw $t3, 0($s0)
     addi $t0, $t0, 1
     add  $v0, $v0, $t3
     addi $s0, $s0, 4
     bne $t1, $t0, LOOP

branch delay slot

6 cycles per loop
Solution II: always predict not-taken

- Always predict the next PC is PC+4

```assembly
LOOP: lw   $t3, 0($s0)  
      addi $t0, $t0, 1   
      add  $v0, $v0, $t3  
      addi $s0, $s0, 4    
      bne  $t1, $t0, LOOP  
      sw   $v0, 0($s1)    
      add  $t4, $t3, $t5  
      lw   $t3, 0($s0)   
```

If branch is not taken: no stalls!
If branch is taken: doesn’t hurt!

7 cycles per loop
Solution III: always predict taken

Consult BTB in fetch stage
Solution III: always predict taken

- Always predict taken with the help of BTB

LOOP: `lw $t3, 0($s0)`
- `addi $t0, $t0, 1`
- `add $v0, $v0, $t3`
- `addi $s0, $s0, 4`
- `bne $t1, $t0, LOOP`
- `lw $t3, 0($s0)`
- `addi $t0, $t0, 1`
- `add $v0, $v0, $t3`

5 cycles per loop
(CPI == 1 !!!)

But what if the branch is not always taken?
Local 1-bit counter

- Predict this branch will go the same way as the result of the last time this branch executed
  - 1 for taken, 0 for not taken

PC = 0x400420

<table>
<thead>
<tr>
<th>Address</th>
<th>Target Address</th>
<th>Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400420</td>
<td>0x8048324</td>
<td>1</td>
</tr>
<tr>
<td>0x400464</td>
<td>0x8048392</td>
<td>1</td>
</tr>
<tr>
<td>0x400578</td>
<td>0x804850a</td>
<td>0</td>
</tr>
<tr>
<td>0x41000C</td>
<td>0x8049624</td>
<td>1</td>
</tr>
</tbody>
</table>

Branch Target Buffer

Taken!
Local 2-bit counter

- A 2-bit counter for each branch
- Predict taken if the counter value \( \geq 2 \)
- If the prediction in taken states, fetch from target PC, otherwise, use PC+4

<table>
<thead>
<tr>
<th>Branch Target Buffer</th>
<th>PC</th>
<th>Branch Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400420</td>
<td>0x8048324</td>
<td>11</td>
</tr>
<tr>
<td>0x400464</td>
<td>0x8048392</td>
<td>10</td>
</tr>
<tr>
<td>0x400578</td>
<td>0x804850a</td>
<td>00</td>
</tr>
<tr>
<td>0x41000C</td>
<td>0x8049624</td>
<td>01</td>
</tr>
</tbody>
</table>

Branch Target Buffer

- Taken! 0x400420

Diagram:

- Taken 3 (11)
- Taken 2 (10)
- Not Taken 0 (00)
- Not Taken 1 (01)
Make the prediction better

- Consider the following code:

```c
i = 0;
do {
    if( i % 3 != 0 ) // Branch Y,
        taken if i % 3 == 0
        a[i] *= 2;
        a[i] += i;
} while ( ++i < 100 ) // Branch X
```

Can we capture the pattern?
Predict using history

- Instead of using the PC to choose the predictor, use a bit vector (global history register, GHR) made up of the previous branch outcomes.
- Each entry in the history table has its own counter.

3-bit GHR = 101 (T, NT, T)

2<sup>3</sup> entries

index

history table

Taken!
Consider the following code:

```c
i = 0;
do {
    if( i % 3 != 0) // Branch Y,
    taken if i % 3 == 0
    a[i] *= 2;
    a[i] += i;
    // Branch Y
} while ( ++i < 100) // Branch X
```

Assume that we start with a 4-bit GHR= 0, all counters are 10.

Nearly perfect after this.
Announcement

• Homework #3 is up
  • Due Sunday (7/17) at noon
  • Will release solution right after the deadline
  • No late submission accepted

• No office hour of Hung-Wei this Friday
  • Will have office hour this Thursday 11a-12p instead
  • You still have TA’s support this Friday
Accuracy of global history predictor

Consider the following code:

```c
sum = 0;
i = 0;
do {
    if(i % 2 == 0)  // Branch Y, taken if i % 2 != 0
        sum+=a[i];
} while ( ++i < 100)  // Branch X
```

Which of predictor performs the best?

A. Predict always taken
B. Predict alway not-taken
C. 1-bit counter for each branch
D. 2-bit counter for each branch
E. 4-bit global history with 2-bit counters
Accuracy of global history predictor

- Consider the following code:

```cpp
sum = 0;
i = 0;
do {
    if(i % 10 == 0) // Branch Y, taken if i % 10 != 0
        sum+=a[i];
} while ( ++i < 100) // Branch X
```

If all counters are initialized as 0s, for branch Y, which predictor performs the best?

A. Predict always taken
B. 1-bit counter for each branch
C. 2-bit counter for each branch
D. 4-bit global history with 2-bit counters

The pattern is longer than GHR
Branch prediction and modern processors
Deep pipeline

- Higher frequencies by shortening the pipeline stages
- Higher marketing values since consumers usually link performance with frequencies
Power

- **Dynamic power:** $P = aCV^2f$
  - $a$: switches per cycle
  - $C$: capacitance
  - $V$: voltage
  - $f$: frequency, usually linear with $V$
  - Doubling the clock rate consumes more power than a quad-core processor!
- **Static/Leakage power** becomes the dominant factor in the most advanced process technologies.
- **Power** is the direct contributor of “heat”
  - Packaging of the chip
  - Heat dissipation cost
Energy

- Energy = \( P \times ET \)
- The electricity bill and battery life is related to energy!
- Lower power does not necessarily mean better battery life if the processor slows down the application too much
Deep Pipeline & Energy

• Increases the power consumption since we increase the clock rate
• You may still save “energy” if you can reduce the “execution time”
Case Study
Intel Pentium 4 Microarch.
**Intel Pentium 4**

- Very deep pipeline: in order to achieve high frequency! (start from 1.5GHz)
  - 20 stages in Netburst

| 1 | TC Nxt IP | 2 | TC Fetch | 3 | Drive | 4 | Alloc | 5 | Rename | 6 | Que | 7 | Sch | 8 | Sch | 9 | Sch | 10 | Sch | 11 | Sch | 12 | Disp | 13 | Disp | 14 | RF | 15 | RF | 16 | Ex | 17 | Flgs | 18 | Br Ck | 19 | Drive |

- 31 stages in Prescott
- 103W (3.6GHz, 65nm)

**Reference**
- [The Microarchitecture of the Pentium 4 Processor](#)
AMD Athlon 64

AMD K8 Architecture
AMD Athlon 64

- **12 stage pipeline**

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>ID and Pack</td>
<td>8</td>
<td>Dispatch</td>
<td>9</td>
<td>Scheduling</td>
<td>10</td>
<td>Execution</td>
<td>11</td>
<td>D-Cache Address</td>
<td>12</td>
<td>D-cache Access</td>
</tr>
</tbody>
</table>

- **89W TDP (Opteron 2.2GHz 90nm)**
Pentium 4 v.s. Athlon 64

- Application: 80% ALU, 20% Branch, 90% prediction accuracy, consider the two machines:
  - Pentium 4 with 20 pipeline stages, branch resolved in stage 19, running at 3 GHz
  - Athlon 64 with 12 pipeline stages, branch resolved in stage 10, running at 2.7 GHz (11% longer cycle time)

which one is faster?

A. Athlon 64
B. Pentium 4
Pentium 4 v.s. Athlon 64

- Application: 80% ALU, 20% Branch, 90% prediction accuracy, consider the two machines:
  - Pentium 4 with 20 pipeline stages, branch resolved in stage 19, running at 3 GHz
  - Athlon 64 with 12 pipeline stages, branch resolved in stage 10, running at 2.7 GHz (11% longer cycle time)

Which one is faster?

\[
\text{CPI}_{\text{P4}} = 0.80 \times 1 + 0.20 \times 0.90 \times 1 + 0.20 \times 0.10 \times 19 = 1.36
\]

\[
\text{CPI}_{\text{Athlon64}} = 0.80 \times 1 + 0.20 \times 0.90 \times 1 + 0.20 \times 0.10 \times 10 = 1.18
\]

At least 15% faster clock rate to achieve the same performance
Demo revisited

• Why the sorting the array speed up the code despite the increased instruction count?

```cpp
if (option)
    std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
```
Deep pipelining and data hazards
Data hazard revisited

• How many cycles it takes to execute the following code?
• Draw the pipeline execution diagram
  • assume that we have full data forwarding.

```
lw   $t1, 0($a0)  
lw   $a0, 0($t1)  
bne  $a0, $zero, 0
```

9 cycles
Data hazards on a different pipeline design

• If we split the “MEM” stage into two stages, ME1 and ME2, and data is available after ME2, how many cycles it takes to execute the following code?

```
lw   $t1, 0($a0)
lw   $a0, 0($t1)
bne  $a0, $zero
```

A. 9  
B. 10  
C. 11  
D. 12  
E. 13