Processor Design — Pipelined Processor (4)

Hung-Wei Tseng
Announcement

• Midterm this Thursday
  • We will talk about the detail next Wednesday
  • No cheat-sheet allowed/required
  • You need to bring a calculator

• Homework 3 & Reading quizzes due next Monday
  • Please don’t use online solution
Recap: Pipelining

- Break up the logic with “pipeline registers” into pipeline stages
  - Each pipeline registers is clocked
  - Each pipeline stage takes one cycle
  - Pipeline registers only updates at the end of a clock
- Each stage acts on a different instruction simultaneously
- Each stage can only work on an instruction once
- States/control signals of instructions are hold in pipeline registers
Recap: Pipeline hazards

• Even though we perfectly divide pipeline stages, it’s still hard to achieve $\text{CPI} == 1$.

• 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
Recap: Stall for Control hazard

- The processor cannot determine the next PC to fetch

```
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)
```

7 cycles per loop
Recap: 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
Recap: always predict not-taken

- Always predict the next PC is PC+4

```assembly
LOOP:  lw    $t3, 0($s0)  [IF]  [ID]  [EXE]  [MEM]  [WB]
     addi  $t0, $t0, 1   [IF]  [ID]  [EXE]  [MEM]  [WB]
     add   $v0, $v0, $t3  [IF]  [ID]  [EXE]  [MEM]  [WB]
     addi  $s0, $s0, 4    [IF]  [ID]  [EXE]  [MEM]  [WB]
     bne   $t1, $t0, LOOP [IF]  [ID]  [EXE]  [MEM]  [WB]
     sw    $v0, 0($s1)    [IF]  [ID]  [EXE]  [MEM]  [WB]
     add   $t4, $t3, $t5  [IF]  [ID]  [EXE]  [MEM]  [WB]
     lw    $t3, 0($s0)    [IF]  [ID]  [EXE]  [MEM]  [WB]
```

If branch is not taken: no stalls!
If branch is taken: no hurt!

7 cycles per loop

flush the instructions fetched incorrectly
Recap: always predict taken

Consult BTB in fetch stage

Branch Target Buffer
Recap: always predict taken

- Always predict taken with the help of BTB

```assembly
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?
Recap: static branch predictions

• How many of the following about static branch prediction method is correct?
  • Comparing with stalls, static branch prediction mechanisms are never doing worse in our current MIPS 5-stage pipeline
  • A static branch prediction mechanism never changes the prediction result during program execution
  • “Flush” occurs only after the processor detects an incorrect branch prediction
  • “Always predict taken” cannot fetch a taken instruction during the ID stage of the branch instruction without the help of BTB

A. 0
B. 1
C. 2
D. 3
E. 4
Dynamic branch prediction
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

Branch Target Buffer
Recap: Accuracy of 1-bit counter

- Consider the following code:

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

What is the prediction accuracy of branch Y using 1-bit predictors (if all counters start with 0/not taken). Choose the most close one.

Assume unlimited BTB entries.

A. 0%
B. 33%
C. 67%
D. 100%

---

<table>
<thead>
<tr>
<th>i</th>
<th>branch</th>
<th>predict</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>T</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>T</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>T</td>
<td>NT</td>
</tr>
</tbody>
</table>
2-bit counter

- A 2-bit counter for each branch
- If the prediction in taken states, fetch from target PC, otherwise, use PC+4

Branch Target Buffer

<table>
<thead>
<tr>
<th>PC</th>
<th>Target Address</th>
<th>Prediction</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>

PC = 0x400420

Taken!
Performance of 2-bit counter

- 2-bit state machine for each branch

```c
for(i = 0; i < 10; i++) {
    sum += a[i];
}
```

90% accuracy!

<table>
<thead>
<tr>
<th>i</th>
<th>state</th>
<th>predict</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4-9</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>T</td>
<td>NT</td>
</tr>
</tbody>
</table>

- Application: 80% ALU, 20% Branch, and branch resolved in EX stage, average CPI?

- $1 + 20\% \times (1-90\%) \times 2 = 1.04$
Accuracy of 2-bit counter

• 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
```

What is the prediction accuracy of branch Y using 2-bit predictors (if all counters start with 00). Choose the closest one. Assume unlimited BTB entries.

A. 0%
B. 33%
C. 67%
D. 100%

---

<table>
<thead>
<tr>
<th>i</th>
<th>branch</th>
<th>state</th>
<th>predict</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
</tbody>
</table>
• Consider the following code:

```c
int i = 0;

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

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.

\[ n \text{-bit GHR} = 101 \text{ (T, NT, T)} \]

\[ \text{2}^n \text{ entries} \]
• Consider the following code:

```java
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.

<table>
<thead>
<tr>
<th>i</th>
<th>?</th>
<th>GHR</th>
<th>BHT</th>
<th>prediction</th>
<th>actual</th>
<th>New BHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y</td>
<td>0000</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>0001</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>0011</td>
<td>10</td>
<td>T</td>
<td>NT</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0110</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>1101</td>
<td>10</td>
<td>T</td>
<td>NT</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>1010</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>0101</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>1011</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>0111</td>
<td>10</td>
<td>T</td>
<td>NT</td>
<td>01</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>1110</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>1101</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
<td>00</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>1010</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>0101</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>1011</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>0111</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
<td>00</td>
</tr>
<tr>
<td>7</td>
<td>X</td>
<td>1110</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>Y</td>
<td>1101</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
<td>00</td>
</tr>
<tr>
<td>8</td>
<td>X</td>
<td>1010</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>Y</td>
<td>0101</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>X</td>
<td>1011</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>Y</td>
<td>0111</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
<td>00</td>
</tr>
</tbody>
</table>

Nearly perfect after this.
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:

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

Which of 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
Deep pipeline and branch prediction
Pentium 4

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

• 31 stages in Prescott
• 103W (3.6GHz, 65nm)
• Reference
  • The Microarchitecture of the Pentium 4 Processor
Athlon 64

- **12 stage pipeline**

|---|---------------------|-----------------|------------------|-------|------|---------------------|--------------|------------|-------------|-------------|------------------|-----------------|

- **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?

\[
\text{CPI}_{\text{P4}} = 0.8 \times 1 + 0.2 \times 0.9 \times 1 + 0.2 \times 0.1 \times 19 = 1.36
\]
\[
\text{CPI}_{\text{Athlon64}} = 0.8 \times 1 + 0.2 \times 0.9 \times 1 + 0.2 \times 0.1 \times 10 = 1.18
\]

At least 15% faster clock rate to achieve the same performance
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)    IF ID EXE MEM WB
lw    $a0, 0($t1)    IF ID ID EXE MEM WB
bne   $a0, $zero, 0  IF IF ID ID EX MEM WB
```

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
Case Study
Pentium 4 Microarch.
Athlon 64

AMD K8 Architecture

Diagram showing the architecture of the Athlon 64 processor, including components like the ITLB, Level 1 Instruction Cache, L2-TLB, Level 2 Cache, L2 ECC, L2 Tags, L2 Tag ECC, System Request Queue (SRQ), Cross Bar (XBAR), Memory Controller, Hyper Transport, Load/Store Queue (44-entry), Data TLB 40-entry, Level 1 Data Cache, 64 kByte, 2-way ECC, 2K Branch Targets, 16k History Counter, RAS & Target Address.
Demo revisited

- Why the sorting the array speed up the code by 2.75x despite the increased instruction count?

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

for (unsigned i = 0; i < 100000; ++i) {
    for (unsigned c = 0; c < arraySize; ++c) {
        if (data[c] >= 128)
            sum += data[c];
    }
}
```
Q & A