Processor Design – Pipelined Processor (IV)

Hung-Wei Tseng
Announcement

• Homework #3 due next Tuesday
• Midterm next Thursday
  • Will talk more about our midterm next Tuesday
  • Focus on the slides, your homework, clicker questions, readings & reading quizzes
• Permanently change my office hours to MTu 2p-3p
  • You still can make an appointment with me if you cannot meet at these time slots
  • Check the calendar all the time
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)  # IF
addi $t0, $t0, 1      # ID
addi $s0, $s0, 4      # MEM
bne $t1, $t0, LOOP    # WB
sw $v0, 0($s1)        # MEM
addi $s0, $s0, 4      # WB
add $v0, $v0, $t3     # WB
```

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

```c
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?
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

<table>
<thead>
<tr>
<th>PC</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>

Taken!
Accuracy of 1-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 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%
Outline

• Dynamic branch predictions
• Branch and modern processors
• Deep pipelines and data hazards
Other dynamic branch predictors
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
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%

The table below shows the branch prediction states for each iteration:

<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>
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.

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

2^n entries

- Taken!
Performance of global history predictor

• 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.

<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
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
Branch prediction and modern processors
Deeper pipeline

- Higher frequencies by shortening the pipeline stages
- Higher marketing values since consumers usually link performance with frequencies
- Potentially higher power consumption as dynamic/active power $= aCV^2f$
- If the execution time is better, still consume less energy
Case Study
Intel Pentium 4 Microarch.
• Very deep pipeline: in order to achieve high frequency! (start from 1.5GHz)
  • 20 stages in Netburst

<table>
<thead>
<tr>
<th></th>
<th>TC Nxt IP</th>
<th>TC Fetch</th>
<th>Drive</th>
<th>Alloc</th>
<th>Rename</th>
<th>Que</th>
<th>Sch</th>
<th>Sch</th>
<th>Shch</th>
<th>Disp</th>
<th>RF</th>
<th>RF</th>
<th>Ex</th>
<th>Flgs</th>
<th>Br Ck</th>
<th>Drive</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• 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

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

- 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
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)  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
Intel’s latest SkyLake

The Skylake microarchitecture builds on the successes of the Haswell and Broadwell microarchitectures. The basic pipeline functionality of the Skylake microarchitecture is depicted in Figure 2-1.

The Skylake microarchitecture offers the following enhancements:

- Larger internal buffers to enable deeper OOO execution and higher cache bandwidth.
- Improved front end throughput.
- Improved branch predictor.
- Improved divider throughput and latency.
- Lower power consumption.
- Improved SMT performance with Hyper-Threading Technology.
- Balanced floating-point ADD, MUL, FMA throughput and latency.

The microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of a number of components including a ring interconnect to multiple slices of L3 (an off-die L4 is optional), processor graphics, integrated memory controller, interconnect fabrics, etc. A four-core configuration can be supported similar to the arrangement shown in Figure 2-3.

Figure 2-1. CPU Core Pipeline Functionality of the Skylake Microarchitecture

Good reference for intel microarchitectures:
Q & A