Dynamic Branch Prediction

Readings: 4.8

Branches introduce control hazards
   Determine the right next instruction in time for instruction fetch

Previous solutions:
   Stall
   Statically predict not taken
   Branch Delay slot

Better:
   Branch-prediction buffers (caches)

Problems With Static Branch Predictors

Are all branches created equal?
**Branch Prediction Buffer**

Direct-mapped cache w/1-bit history
- Predict taken/not taken by previous execution
- If incorrect prediction, annul instructions incorrectly started

Tags?
Valid bits?

---

**Thought Experiment**

Consider the following code segment:

```
while (1) {
    <code 1>
    for(int i=0; i<9; i++) {
        <code 2>
    }
    <code 3>
}
```

1-bit prediction accuracy?

```assembly
addi C, $0, 9  # Const. 9
WHILE_TOP:
    <code 1>
    add i, $0, $0  # Init. i
    j FOR_TEST
FOR_TOP:
    <code 2>
    addi i, i, 1  # i++
    FOR_TEST:
        slt $t0, i, C  # i < 9?
        bneq $t0,$0,FOR_TOP
    <code 3>
    j WHILE_TOP  # Endwhile
```
2-bit Predictor

while (1) {
  if (normal_condition) {
    <code1>
  }
  for(int i=0; i<9; i++) {
    if (exception) {
      return (FALSE);
      <code 2>
    }
    <code 2>
  }
  if (random 50/50 chance) {
    <code 3>
  }
}
Instruction-Level Parallelism & Advanced Architectures

Readings: 4.10

Key to pipelining was dealing with hazards

Advanced processors require significant hazard avoidance/flexibility

ILP = Instruction-Level Parallelism

Why ILP

Advanced processors optimize two factors:

- Reduce clock period by heavy pipelining
  - Greater pipelining means more hazards, delay slots

- Reduce CPI
  - What if we want a CPI < 1.0?
**ILP Example**

Source code:

```assembly
addi $t0, $t0, 15
sub $t2, $t1, $t0
xori $t3, $t0, 15
or $t4, $t3, $t0
```

Constraint graph:

---

**Complex ILP Example**

Note: Assume no delay slots.

```assembly
1: lw $t0, 0($t3)
2: add $t1, $t0, $t2
3: sub $t2, $t3, $t4
4: andi $t2, $t5, 57
5: or $t7, $t5, $t1
6: sw $t8, 0($t9)
7: bne $t8, $t7, LOOP
8: slt $t0, $t8, $t0
```
### Types of Hazards

Data Hazards (multiple uses of the same location)
- RAW – Read after write
- WAW – Write after write
- WAR – Write after read
- RAR – Read after read

Memory

Control Hazards
- Branches/Jumps

### Hazard Minimization

Hardware and software techniques for improving performance
- Loop unrolling
- Register Renaming
- Predicated Instructions

Responsibility
- Hardware
- Software (Compiler)
Loop-Level Parallelism

```c
LLPex(char *src) {
    char *end = src + 1000;
    while(src!=end) {
        *src = (*src)+3; src++;
    }
}
```

Loop Unrolling

Better loop structure?
Compiler Register Renaming

Compiler reduces hazards by removing name dependences

Hardware Register Renaming

Add hardware to handle WAR and WAW conflicts

```
0: addi a0, a0, 4
1: lw t0, 0(a0)
2: sw t0, 100(a0)
3: addi a0, a0, 4
4: lw t0, 0(a0)
5: sw t0, 100(a0)
6: addi a0, a0, 4
7: lw t0, 0(a0)
8: sw t0, 100(a0)
```
Hardware Register Renaming

Control Hazards

if (c == 0) { t = s; }
if (a[0] == 0)
    a[0] = b[0];
else
    a[0] = a[0] + 4;

Branches introduce hazards that limit ILP
  Branch prediction
    Conditional/Predicated instructions
Predicated Instructions

if (c == 0) { t = s; }

Normal:

W/Conditional move (instructions with internal if-like operation – no jumps)

cmovz <dest>, <src>, <cond>  # move src to dest if cond == 0
cmovnz <dest>, <src>, <cond>  # move src to dest if cond != 0

Predicated Instructions (cont.)

d = a - b;
if (d < 0)
    sum -= d;
else
    sum += d;

Normal:  

Predicated Instructions:
Why ILP

Advanced processors optimize two factors:

- Reduce clock period by heavy pipelining
  Greater pipelining means more hazards, delay slots

- Reduce CPI
  What if we want a CPI < 1.0?

Superpipelining

Divide datapath into **multiple** pipeline stages

![Superpipelining Diagram]

Pentium 4 Processor (“NetBurst”): 20 stages

- IF: Instruction Fetch
- RF: Register Fetch
- EX: Execute
- MEM: Data Memory
- WB: Writeback

<table>
<thead>
<tr>
<th>IF</th>
<th>RF</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr. Memory</td>
<td>Register File</td>
<td>Execute</td>
<td>Data Memory</td>
<td>Register File</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TC inet IP</th>
<th>TC fetch</th>
<th>Drive Alloc</th>
<th>Rename</th>
<th>Queue</th>
<th>Schedule</th>
<th>Dispatch</th>
<th>Reg File</th>
<th>Exec Flags</th>
<th>Branch</th>
<th>Ck Drive</th>
</tr>
</thead>
<tbody>
<tr>
<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>

198

199
Multiple Issue

Replicate Execution Units

ALU
Shifter
Floating Point
Register File
Memory
Ifetch/Branch

VLIW

Very Long Instruction Word

Max CPI?

Concerns
VLIW Scheduling

Schedule the code for a 4-way VLIW. Assume no delay slots, and all instructions in parallel with a branch/jump still execute.

1: lw $t1, 4($a0)
2: add $t2, $t1, $a0
3: lw $t3, 8($a1)
4: sub $t2, $t3, $a2
5: sw $t5, 0($a3)
6: beq $s0, $s1, FOO
7: and $t7, $s0, $s1

Superscalar

Dynamically schedule multiple instructions, based on hazard detection

Schedule/Dispatch
Hazard Detection

Issues

Mem  PC
Example Execution on Modern Processors

Bringing everything together, how would this code be run on an advanced CPU? No delay slots, instructions in parallel with a branch/jump still execute

0: lw $t0, 4($t1)
1: addi $t2, $t0, 4
2: sw $t2, 4($t1)
3: beq $t1, $0, SKIP
4: addi $t3, $t4, 0
SKIP:
5: addi $t5, $t5, 1
6: xori $t0, $t1, 5
7: sll $t0, $t0, 2
8: sw $t0, 8($t1)

Example on VLIW

How would this be scheduled on a 2-way VLIW? Assume no delay slots.

<table>
<thead>
<tr>
<th>ALU/Load/Store</th>
<th>ALU/Branch/Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example on Superscalar

How would this execute on a 2-way superscalar? Assume it schedules the earliest ready instruction first, and cache miss only stalls dependent instructions.

<table>
<thead>
<tr>
<th>No miss</th>
<th>2-cycle miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU/Load/Store</td>
<td>ALU/Branch/Jump</td>
</tr>
<tr>
<td>ALU/Load/Store</td>
<td>ALU/Branch/Jump</td>
</tr>
<tr>
<td>ALU/Load/Store</td>
<td>ALU/Branch/Jump</td>
</tr>
<tr>
<td>ALU/Load/Store</td>
<td>ALU/Branch/Jump</td>
</tr>
<tr>
<td>ALU/Load/Store</td>
<td>ALU/Branch/Jump</td>
</tr>
<tr>
<td>ALU/Load/Store</td>
<td>ALU/Branch/Jump</td>
</tr>
<tr>
<td>ALU/Load/Store</td>
<td>ALU/Branch/Jump</td>
</tr>
<tr>
<td>ALU/Load/Store</td>
<td>ALU/Branch/Jump</td>
</tr>
<tr>
<td>ALU/Load/Store</td>
<td>ALU/Branch/Jump</td>
</tr>
</tbody>
</table>

Symmetric Multi-Threading (SMT)

~Superscalar drawing from multiple programs or program threads

Issues
Multicore

Add multiple processors (separate control & datapath)

**Issues**

Multicore Example

Using a multicore is significantly more complex.

**Example: Uniprocessor MAX**

```c
int max(int vals[], int len) {
    int result = -infinity;
    for (int i=0; i<len; i++) {
        if (vals[i] > result)
            result = vals[i];
    }
    return result;
}
```

**Multicore MAX**

```c
int max(int vals[], int len) {
    int global_result = -infinity;
    int lenT = len/num_procs;
    for (int i=0; i<num_procs; i++)
        process maxT(&vals[i*lenT], lenT);
}
```

```c
void maxT(int vals[], int len) {
```