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 conditional 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?

ADDI C, X31, #9 // Const. 9
WHILE_TOP:
    <code 1>
    ADD i, X31, X31 // Init i
    B FOR_TEST
FOR_TOP:
    <code 2>
    ADDI i, i, #1 // i++
FOR_TEST:
    CMP i, C // i < 9?
    B.LT FOR_TOP
    <code 3>
    B 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:

ADDI X0, X0, #15
SUB  X2, X1, X0
EORI X3, X0, #15
ORR  X4, X3, X0

Constraint graph:
Complex ILP Example

Note: Assume no delay slots.

1: LDUR X0, [X3, #0]
2: ADD X1, X0, X2
3: SUB X2, X3, X4
4: ANDI X2, X5, #57
5: ORR X7, X5, X1
6: STUR X5, [X9, #0]
7: CBZ X7, LOOP
8: EOR X6, X8, X6
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
Hazard Minimization

Hardware and software techniques for improving performance
  Loop unrolling

  Register Renaming

  Predicated Instructions

Responsibility
  Hardware
  Software (Compiler)
Loop-Level Parallelism

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

CPU dynamically associates each register read with the most recent instruction issue that writes that register.

0: ADDI X0, X0, #8
1: LDUR X1, [X0, #0]
2: STUR X1, [X0, #100]
3: ADDI X0, X0, #8
4: LDUR X1, [X0, #0]
5: STUR X1, [X0, #100]
6: ADDI X0, X0, #8
7: LDUR X1, [X0, #0]
8: STUR X1, [X0, #100]
Hardware Register Renaming (cont.)
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 branches)

CMOVZ <dest>, <src>, <cond> // move src to dest if cond == 0
CMOVNZ <dest>, <src>, <cond> // move src to dest if cond != 0

Note: ARM actually uses CSEL, a Mux-like instruction instead. But, predication important enough we’ll pretend there’s a CMOV instruction…
d = a - b;
if (d<0)
    sum -= d;
else
    sum += d;

Predicated Instructions:

Normal:
ARM CSEL Instruction: the MUX

CSEL <dest>, <src1>, <src2>, <cond>
   <dest> = If <cond> then <src1> else <src2>
   <cond>: EQ, NE, LT, etc.

\[ d = a - b; \]
\[ \text{if } (d<0) \]
\[ \quad \text{sum } -= d; \]
\[ \text{else} \]
\[ \quad \text{sum } += d; \]
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

Pentium 4 Processor (“NetBurst”): 20 stages
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 still execute.

1: LDUR X6, [X0, #4]
2: ADD X7, X6, X0
3: LDUR X8, [X1, #8]
4: SUB X7, X8, X2
5: STUR X10, [X3, #0]
6: CBZ X4, FOO
7: AND X12, X4, X5
Superscalar

Dynamically schedule multiple instructions, based on hazard detection

```
Instr I + 7 | Instr I + 6 | Instr I + 5 | Instr I + 4 | Instr I + 3 | Instr I + 2 | Instr I + 1 | Instr I
```

Issues

Schedule/Dispatch
Hazard Detection

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 still execute

0:  LDUR X0, [X1, #8]
1:  ADDI X2, X0, #4
2:  STUR X2, [X1, #8]
3:  CBZ  X1, SKIP
4:  ADDI X3, X4, #0
   SKIP:
5:  ADDI X5, X5, #1
6:  EORI X0, X1, #5
7:  LSL  X0, X0, #2
8:  STUR X0, [X1, #16]
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</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>
<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</td>
</tr>
<tr>
<td>ALU/Load/Store</td>
<td>ALU/Branch</td>
</tr>
<tr>
<td>ALU/Load/Store</td>
<td>ALU/Branch</td>
</tr>
<tr>
<td>ALU/Load/Store</td>
<td>ALU/Branch</td>
</tr>
<tr>
<td>ALU/Load/Store</td>
<td>ALU/Branch</td>
</tr>
</tbody>
</table>
Symmetric Multi-Threading (SMT)

~Superscalar drawing from multiple programs or program threads

<table>
<thead>
<tr>
<th>Instr L + 7</th>
<th>Instr L + 6</th>
<th>Instr L + 5</th>
<th>Instr L + 4</th>
<th>Instr L + 3</th>
<th>Instr L + 2</th>
<th>Instr L + 1</th>
<th>Instr L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr J + 7</td>
<td>Instr J + 6</td>
<td>Instr J + 5</td>
<td>Instr J + 4</td>
<td>Instr J + 3</td>
<td>Instr J + 2</td>
<td>Instr J + 1</td>
<td>Instr J</td>
</tr>
<tr>
<td>Instr K + 7</td>
<td>Instr K + 6</td>
<td>Instr K + 5</td>
<td>Instr K + 4</td>
<td>Instr K + 3</td>
<td>Instr K + 2</td>
<td>Instr K + 1</td>
<td>Instr K</td>
</tr>
</tbody>
</table>

Schedule/Dispatch
Hazard Detection

Issues

Mem
PC
Multicore

Add multiple processors (separate control & datapath)

![Diagram showing a computer with processors 1 to 4, memory, and devices (input and output).]

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_proc; i++)
        process maxT(&vals[i*lenT], lenT);
}

void maxT(int vals[], int len) {
```
Flynn’s Taxonomy

Readings: 6.3, 6.6

Categorize models of parallel computation

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Data Streams</th>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>SISD</td>
<td>![SISD Diagram]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>![SIMD Diagram]</td>
<td>MIMD</td>
</tr>
<tr>
<td>Multiple</td>
<td></td>
<td>![MIMD Diagram]</td>
<td>![MIMD Diagram]</td>
</tr>
</tbody>
</table>
Early SIMD: Vector Processors

Add vector registers, each holding say 32 ints: V0..V7
Add vector operations. E.g. ADDV V0, V1, V2  // V0i = V1i + V2i

Example: Compute Y = aX+Y. X0 = a, X19 = &X, X20 = &Y

// Vectorized version
LDURV V1, [X19, #0] // Get X
MULVS V2, V1, X0 // a*X
LDURV V3, [X20, #0] // Get Y
ADDV V4, V3, V2 // a*X+Y
STURV V4, [X20, #0] // Save Y

// Standard ARM assembly
ADDI X1, X19, 32*8 // &X[32]
LOOP:
LDUR X2, [X19, #0] // Get X[i]
MUL X2, X2, X1 // a*X[i]
LDUR X3, [X20, #0] // Get Y[i]
ADD X4, X3, X2 // a*X[i]+Y[i]
STUR X4, [X20, #0] // Save Y
ADDI X19, X19, #8 // X++
ADDI X20, X20, #8 // Y++
CMP X1, X19 // X == end?
B.NE LOOP
Vector Lanes

1 lane
A[0]  B[0]

2 lane
Vector Considerations

Good:
- Repetitive computations
- Block-based LDUR/STUR moves (like cache blocks)
- Remove loop overhead
- Easily parallelized

Bad:
- “Jumpy” code: if-then, case
  - Predicated instructions, but still wasteful
- Working on data that isn’t “packed” together.
  - Scatter/Gather instructions
- Can’t accelerate non-vectorized code
Graphics Processing Units (GPUs)

Processor optimized for graphics processing, coprocessor to main CPU

GPUs, vs. CPUs

- CPU coprocessor, so don’t have to support everything.
- GPU datasets smaller than CPUs (4-6GB vs. 32-256GB)
- Skip multilevel caches, use high-bandwidth memory
  - Hide latency via multi-threading
- Provide many parallel processors (MIMD), that are internally SIMD.

Example

- 8-16 SIMD Processors
- 8-16 Lanes/SIMD Processor
- Largest cache: 0.75MB (comparable CPU might be 8MB)
- DRAM: 4-6GB (comparable CPU might be 8-256GB)
GPU SIMD Processor