CS 152
Computer Architecture and Engineering

Lecture 18 – Advanced Processors II

2006-10-31

John Lazzaro
(www.cs.berkeley.edu/~lazzaro)

TAs: Udam Saini and Jue Sun

www-inst.eecs.berkeley.edu/~cs152/
We update the PC based on the outputs of the branch predictor. If it is perfect, pipe stays full!

Dynamic Predictors: a cache of branch history

If we predicted incorrectly, these instructions MUST NOT complete!
Update BHT/BTB for next time, once true behavior known.

Branch Target Buffer (BTB)

<table>
<thead>
<tr>
<th>28-bit address tag</th>
<th>target address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0110[...]0100</td>
<td>PC + 4 + Loop</td>
</tr>
</tbody>
</table>

BNEZ R1 Loop

“Taken” or “Not Taken”

Must check prediction, kill instruction if needed.
Simple (“2-bit”) Branch History Table Entry

**Prediction for next branch.**
(1 = take, 0 = not take)
Initialize to 0.

**Was last prediction correct?**
(1 = yes, 0 = no)
Initialize to 1.

D  Q

After we “check” prediction ...

Flip bit if prediction is not correct and “last predict correct” bit is 0.

Set to 1 if prediction bit was correct.
Set to 0 if prediction bit was incorrect.
Set to 1 if prediction bit flips.

We do not change the prediction the first time it is incorrect. Why?

ADDI R4, R0, 11

**loop:** SUBI R4, R4, -1
BNE R4, R0, loop

This branch taken 10 times, then not taken once (end of loop). The next time we enter the loop, we would like to predict “take” the first time through.
Spatial enhancements: many BHTs ... 

Branch History Tables

(BHT00) (BHT01) (BHT10) (BHT11)

Adaptive function of history, state

"Taken" or "Not Taken"

Detects patterns in:

if (x < 12) [...]
if (x < 6) [...]

code.

Yeh and Patt, 1992.

BHT00/01/10/11 code the last four branches in the instruction stream

Yeh and Patt, 1992.
Superscalar \textit{R} machine

\begin{itemize}
  \item Instruction Issue Logic
  \item Data
  \item Instr
  \item Mem
  \item Addr
  \item PC and Sequencer
\end{itemize}
Sustaining Dual Instr Issues (no forwarding)

ADD R8, R0, R0
ADD R11, R0, R0
ADD R27, R26, R25
ADD R30, R29, R28
ADD R21, R20, R19
ADD R24, R23, R22
ADD R15, R14, R13
ADD R18, R17, R16
ADD R9, R8, R7
ADD R12, R11, R10

Best-case example ...
We add 12 forwarding buses (not shown). (6 to each ID from stages of both pipes).

Worst-Case Instruction Issue

ADD R8, R0, R0
ADD R9, R8, R0
ADD R10, R9, R0
ADD R11, R10, R0

Dependencies force "serialization"
Today: Out of Order Execution

Goal: Issue instructions out of program order

Example:

<table>
<thead>
<tr>
<th>Seconds Program</th>
<th>Instructions Program</th>
<th>Cycles Instruction</th>
<th>Seconds Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F2, 34(R2)</td>
<td>latency 1</td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>F4, 45(R3)</td>
<td>long</td>
<td></td>
</tr>
<tr>
<td>MULTD</td>
<td>F6, F4, F2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ADDD</td>
<td>F8, F2, F2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

... so let ADDD go first

MULTD waiting on F4 to load ...
Dynamic Scheduling: Enables Out-of-Order

**Goal:** Enable out-of-order by breaking pipeline in two: fetch and execution.

**Example:** IBM Power 5:

*Figure: CP compute 0, Instruction fetch and decode: like static pipelines*

*Today's focus: execution unit*
90 nm, 58 M transistors

L1 (64K Instruction) ↓ ↓ ↓ ↓
L2

L1 (32K Data) ↑ ↑ ↑ ↑

PowerPC 970 FX
Recall: WAR and WAW hazards ...

Write After Read (WAR) hazards. Instruction $I_2$ expects to write over a data value after an earlier instruction $I_1$ reads it. But instead, $I_2$ writes too early, and $I_1$ sees the new value.

Write After Write (WAW) hazards. Instruction $I_2$ writes over data an earlier instruction $I_1$ also writes. But instead, $I_1$ writes after $I_2$, and the final data value is incorrect.

Dynamic scheduling eliminates WAR and WAW hazards, making out-of-order execution tractable.
Dynamic Scheduling: A mix of 3 ideas

Imagine: an endless supply of registers ...

Top-down idea: Registers that may be written only once (but may be read many times) eliminate WAW and WAR hazards.

Mid-level idea: An instruction waiting for an operand to execute may trigger on the (single) write to the associated register. (eliminates RAW hazards)

Bottom-up idea: To support “snooping” on register writes, attach all machine elements to a common bus.

Robert Tomasulo, IBM, 1967. FP unit for IBM 360/91
Register Renaming

Imagine: an endless supply of registers??
How???
Consider this simple loop ...

Loop:    LD  F0,0(R1) ;F0= array element
        ADDD  F4,F0,F2 ;add scalar from F2
        SD  F4,0(R1) ;store result
        SUBI  R1,R1,8 ;decrement pointer 8B (DW)
        BNEZ  R1,Loop ;branch R1!=zero
        NOP ;delayed branch slot

Every pass through the loop introduces the potential for WAW and/or WAR hazards for F0, F4, and R1.

(Note: F registers are floating point registers. F0 is not equal to the constant 0, but instead is a normal register just like F1, F2, ...).
Given an endless supply of registers ...

Rename “architected registers” (Ri, Fi) to new “physical registers” (PRi, PFi) on each write.

Loop:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Old Registers</th>
<th>New Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>R1, R0, 64</td>
<td>PRO1, PR00, 64</td>
</tr>
<tr>
<td>LD</td>
<td>F0, 0(R1)</td>
<td>PF00</td>
</tr>
<tr>
<td>ADDD</td>
<td>F4, F0, F2</td>
<td>PF04, PF00, PF02</td>
</tr>
<tr>
<td>SD</td>
<td>F4, 0(R1)</td>
<td>PF04, 0(PR01)</td>
</tr>
<tr>
<td>SUBI</td>
<td>R1, R1, 8</td>
<td>PR11, PR01, 8</td>
</tr>
<tr>
<td>BNEZ</td>
<td>R1, Loop</td>
<td>PR21</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td>ENDLOOP</td>
</tr>
</tbody>
</table>

What was gained?

An instruction may execute once all of its source registers have been written.
Bus-Based CPUs
A common bus == long wires == slow?

Pipelines in theory

Wires are short, so clock periods can be short. "wiring by abutment"

Long wires are the price we paid to avoid stalls

Conjecture: If processor speed is limited by long wires, lets do a design that fully uses the semantics of long wires by using a bus.
If we add too many functional units, one bus is too long, too slow. Solutions: more buses, faster electrical signalling.

1. Only one unit writes at a time (one source).
2. All units may read the written values (many destinations), if interested in id#.

Common Data Bus <data id#, data value>

If we add too many functional units, one bus is too long, too slow. Solutions: more buses, faster electrical signalling.

1. Only one unit writes at a time (one source).
2. All units may read the written values (many destinations), if interested in id#.
Data-Driven Execution
(Associative Control)

Caveat: In comparison to static pipelines, there is great diversity in dynamic scheduling implementations. Presentation that follows is a composite, and does not reflect any specific machine.
Recall: IBM Power 5 block diagram ...

Queues between instruction fetch and execution.

Branch redirects

Instruction fetch

Interrupts and flushes

Out-of-order processing

Branch pipeline

Load/store pipeline

Fixed-point pipeline

Floating-point pipeline

MP = “Mapping” from architected registers to physical registers (renaming).

ISS = Instruction Issue

MP = “Mapping” from architected registers to physical registers (renaming).
Instructions placed in “Reorder Buffer”

Each line holds physical <src1, src2, dest> registers for an instruction, and controls when it executes.

Execution engine works on the physical registers, not the architecture registers.

Reorder Buffer

<table>
<thead>
<tr>
<th>Inst #</th>
<th>[...]</th>
<th>src1 #</th>
<th>src1 val</th>
<th>src2 #</th>
<th>src2 val</th>
<th>dest #</th>
<th>dest val</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</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>
</tr>
<tr>
<td>[...]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Common Data Bus: <reg #, reg val>

From Memory

Load Unit

ALU #1

ALU #2

Store Unit

To Memory
Circular Reorder Buffer: A closer look

Next instr to “commit”, (complete).

<table>
<thead>
<tr>
<th>Inst #</th>
<th>Op</th>
<th>U</th>
<th>E</th>
<th>#1</th>
<th>#2</th>
<th>#d</th>
<th>P1</th>
<th>P2</th>
<th>Pd</th>
<th>P1 value</th>
<th>P2 value</th>
<th>Pd value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ADD</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>OR</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>SUB</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction opcode

Use bit (1 if line is in use)

Execute bit (0 if waiting ...)

Add next inst, in program order.

Physical register numbers

Valid bits for values

Copies of physical register values
Example: The life of \texttt{ADD R3,R1,R2}

Issue: R1 “renamed” to PR21, whose value (13) was set by an earlier instruction. R2 renamed to PR22; it has not been written. R3 renamed to PR23.

<table>
<thead>
<tr>
<th>Inst#</th>
<th>Op</th>
<th>U</th>
<th>E</th>
<th>#1</th>
<th>#2</th>
<th>#d</th>
<th>P1</th>
<th>P2</th>
<th>Pd</th>
<th>P1 value</th>
<th>P2 value</th>
<th>Pd value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Add</td>
<td>1</td>
<td>0</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

A write to PR22 appears on the bus, value 87. Both operands are now known, so 13 and 87 sent to ALU.

<table>
<thead>
<tr>
<th>Inst#</th>
<th>Op</th>
<th>U</th>
<th>E</th>
<th>#1</th>
<th>#2</th>
<th>#d</th>
<th>P1</th>
<th>P2</th>
<th>Pd</th>
<th>P1 value</th>
<th>P2 value</th>
<th>Pd value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Add</td>
<td>1</td>
<td>1</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>13</td>
<td>87</td>
<td>-</td>
</tr>
</tbody>
</table>

ALU does the add, writing \(<\text{PR23}, 100\>) onto the bus.

<table>
<thead>
<tr>
<th>Inst#</th>
<th>Op</th>
<th>U</th>
<th>E</th>
<th>#1</th>
<th>#2</th>
<th>#d</th>
<th>P1</th>
<th>P2</th>
<th>Pd</th>
<th>P1 value</th>
<th>P2 value</th>
<th>Pd value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Add</td>
<td>1</td>
<td>1</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td>87</td>
<td>100</td>
</tr>
</tbody>
</table>
More details (many are still overlooked)

Q. Why are we storing each physical register value several times in the reorder buffer? Quick access.

Example: Load/Store Disambiguation

<table>
<thead>
<tr>
<th>Inst #</th>
<th>src1 #</th>
<th>src1 val</th>
<th>src2 #</th>
<th>src2 val</th>
<th>dest #</th>
<th>dest val</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</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>
</tr>
<tr>
<td>[...]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Issue logic monitors bus to maintain a physical register file, so that it can fill in <val> fields during issue.

Reorder buffer: a state machine triggered by reg# bus comparisons

From Memory

Common Data Bus: <reg #, reg val>

To Memory
Exceptions and Interrupts

Exception: An unusual event happens to an instruction during its execution. Examples: divide by zero, undefined opcode.

Interrupt: Hardware signal to switch the processor to a new instruction stream. Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting).
Challenge: Precise Interrupt / Exception

Definition:

*It must appear as if an interrupt is taken between two instructions* (say $I_i$ and $I_{i+1}$)

- the effect of all instructions up to and including $I_i$ is totally complete
- no effect of any instruction after $I_i$ has taken place

The interrupt handler either aborts the program orrestarts it at $I_{i+1}$.

Follows from the “contract” between the architect and the programmer ...
Precise Exceptions in Static Pipelines

Key observation: architected state only change in memory and register write stages.
Dynamic scheduling and exceptions ...

Key observation: Only the architected state needs to be precise, not the physical register state. So, we delay removing instructions from the reorder buffer until we are ready to "commit" to that state changing the architected registers.
Add completion logic to data path ...

To sustain CPI < 1, must be able to do multiple issues, commits, and reorder buffer execution launches and writes per cycle.

Not surprising design and validation teams are so large.
Power 5: By the numbers ...

Fetch up to 8 instructions per cycle.

Dispatch up to 5 instructions per cycle

Execute up to 8 instructions per cycle

Branch redirects

Instruction fetch

Interrupts and flushes

Out-of-order processing

Branch pipeline

Load/store pipeline

Fixed-point pipeline

Floating-point pipeline

MP

ISS

RF

EX

DC

F6

CP

WB

Xfer

WB

Xfer

WB

Xfer

Up to 200 instructions “in flight”.

240 physical registers (120 int + 120 FP)

A thread may commit up to 5 instructions per cycle.

Up to 200 instructions “in flight”.

240 physical registers (120 int + 120 FP)

A thread may commit up to 5 instructions per cycle.

CS 152 L18: Advanced Processors II
Note: Good branch prediction required
Because so many stages between predict and result!

BP = Branch prediction. On IBM Power 5, quite complex ... uses a predictor to predict the best branch prediction algorithm!
Conclusions: Dynamic Scheduling

Three big ideas: register renaming, data-driven detection of RAW resolution, bus-based architecture.

Very complex, but enables many things: out-of-order execution, multiple issue, loop unrolling, etc.

Has saved architectures that have a small number of registers: IBM 360 floating-point ISA, Intel x86 ISA.
This Friday: Memory System Checkoff

Run your test vector suite on the Calinx board, display results on LEDs