Last Time: Superpipelining & Superscalar

Address of BNEZ instruction
0b0110[...]01001000

BNEZ R1 Loop

Branch Target Buffer (BTB)

Branch History Table (BHT)

Update BHT/BTB for next time, once true behavior known

“Taken” Address
Must check prediction, kill instructions if needed.

“Taken” or “Not Taken”

Hit

28-bit address tag

28 bits

2 bits

Target address

0b0110[...]0100

PC + 4 + Loop

UC Regents Spring 2005 © UCB
Today: Dynamic Scheduling Overview

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

Example: IBM Power 5:

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

L1 (64K Instruction) ↓ ↓ ↓ ↓

L1 (32K Data) ↑ ↑ ↑ ↑

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

Write After Read (WAR) hazards. Instruction I2 expects to write over a data value after an earlier instruction I1 reads it. But instead, I2 writes too early, and I1 sees the new value.

Write After Write (WAW) hazards. Instruction I2 writes over data an earlier instruction I1 also writes. But instead, I1 writes after I2, 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.
Given an endless supply of registers ...

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

**Loop:**

ADDI R1, R0, 64

LD F0, 0(R1)

ADDD F4, F0, F2

SD F4, 0(R1)

SUBI R1, R1, 8

BNEZ R1, Loop

NOP

**ADDI PR01, PR00, 64**

**LD PF00 0(PR01)**

**ADDD PF04, PF00, PF02**

**SD PF04, 0(PR01)**

**SUBI PR11, PR01, 8**

**BEQZ PR11 ENDLOOP**

**ITER2: LD PF10 0(PR11)**

**ADDD PF14, PF10, PF02**

**SD PF14, 0(PR11)**

**SUBI PR21, PR11, 8**

**BEQZ PR21 ENDLOOP**

**ITER3: LD PF20 0(PR21)**

[...]

**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"

Pipelines in practice

Long wires are the price we paid to avoid stalls

Conjecture: If processor speed is limited by long wires, let's do a design that fully uses the semantics of long wires by using a bus.
A bus-based multi-cycle computer

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

Common Data Bus <data id#, data value>
(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.

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

ISS = Instruction Issue
Instructions placed in “Reorder Buffer”

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

Common Data Bus: <reg #, reg val>

Execution engine works on the physical registers, not the architecture registers.
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>8</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</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>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 100 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>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)

**Example: Load/Store Disambiguation**

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.

<table>
<thead>
<tr>
<th>Inst #</th>
<th>src #</th>
<th>src val</th>
<th>src #</th>
<th>src 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>

Load Unit → ALU #1 → ALU #2 → Store Unit → To Memory

Common Data Bus: `<reg #, reg val>`

Q. Why are we storing each physical register value several times in the reorder buffer? See next topic...
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:** (or exception)

*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 or restarts 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 ...

Reorder Buffer

<table>
<thead>
<tr>
<th>Inst #</th>
<th>[...]</th>
<th>src #</th>
<th>src val</th>
<th>src #</th>
<th>src 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>

Load Unit

ALU #1

ALU #2

Store Unit

Commit

ISA Registers

From Memory

To Memory
Final thought: 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.
Reminder: Friday Test Bench Checkoff

<table>
<thead>
<tr>
<th>F 4/8</th>
<th>Final Project: Test Bench Checkoff</th>
</tr>
</thead>
</table>

---

**Test vector suite for Week 3/4/5 checkoffs, running in ModelSim (3/4) and SPIM (5).**

**Detailed block diagrams, state machines, and Lab 3 CPU changes**