CS 152
Computer Architecture and Engineering
Lecture 18 -- Dynamic Scheduling I

2014-4-1
John Lazzaro
(not a prof - “John” is always OK)

TA: Eric Love

www-inst.eecs.berkeley.edu/~cs152/
Today: Out of Order Execution

Goal: Issue instructions out of program order

Example:

... so let ADDD go first

Also: Speculate through branches, aim for CPI < 1
Dynamic Scheduling: Enables Out-of-Order

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

Example: IBM Power 5:

l-fetch and decode: like static pipelines

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

L1 (64K Instruction) ↓ ↓ ↓ ↓

512K

L2

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.

(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.

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Registers</th>
<th>Destination Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI R1, R0, 64</td>
<td></td>
<td>ADDI PR01, PR00, 64</td>
</tr>
<tr>
<td>Loop: LD F0, 0(R1)</td>
<td>R1→ PR01</td>
<td></td>
</tr>
<tr>
<td>ADDDD F4, F0, F2</td>
<td></td>
<td>ADDDD PF04, PF00, PF02</td>
</tr>
<tr>
<td>SD F4, 0(R1)</td>
<td>F0→ PF00</td>
<td></td>
</tr>
<tr>
<td>SUBI R1, R1, 8</td>
<td></td>
<td>SD PF04, 0(PR01)</td>
</tr>
<tr>
<td>BNEZ R1, Loop</td>
<td></td>
<td>SUBI PR11, PR01, 8</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td>BEQZ PR11 ENDLOOP</td>
</tr>
</tbody>
</table>
```

What was gained?

An instruction may execute once all of its source registers have been written.
Modified to include additional detail on register renaming and physical register reclamation.
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
Each line holds physical \(<\text{src1, src2, dest}\)> registers for an instruction, and controls when it executes.

Execution engine works on the physical registers, not the architecture registers.
### Circular Reorder Buffer: A closer look

Next instr to “commit”, (complete).

**Instruction opcode**
- Use bit (1 if line is in use)
- Execute bit (0 if waiting ...)

<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>0</td>
<td>0</td>
<td>0</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>0</td>
<td>0</td>
<td>0</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>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Add next inst, in program order.

Physical register numbers

Valid bits for values

Copies of physical register values

Next instruction to “commit”, (complete).

Thursday, April 3, 14
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 \texttt{< 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

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

Load Unit

ALU #1

ALU #2

Store Unit

Common Data Bus: <reg #, reg val>

To Memory

Table:

<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>

Example: Load/Store Disambiguation

Load Unit

Store Unit

Common Data Bus: <reg #, reg val>

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

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.
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!
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.

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: Dynamic Scheduling I
Moore’s Law

2.6 Billion

Synchronous logic on a single clock domain is not practical for a 276 million transistor design.

Power 5: 276 million transistors
GALS: Globally Asynchronous, Locally Synchronous

Both clocks. The basic GALS method focuses on point-to-point communication between blocks. Another approach to interfacing locally synchronous blocks is using specially designed asynchronous FIFO buffers and hiding the system synchronization problem within the FIFO buffers. Such a system can tolerate very large interconnect delays and is also robust with regard to metastability. Designers can use this method to interconnect asynchronous and synchronous systems and also to construct synchronous-synchronous and asynchronous-asynchronous interfaces. Figure 2 diagrams a typical FIFO interface, which achieves an acceptable data throughput.

In addition to the data cells, the FIFO structure includes an empty/full detector and a special deadlock detector. The advantage of FIFO synchronizers is that they don't affect the locally synchronous module's operation. However, with very wide interconnect data buses, FIFO structures can be costly in silicon area. Also, they require specialized complex cells to generate the empty/full flags used for flow control. The introduced latency might be significant and unacceptable for high-speed applications.

As an alternative, Beigne and Vivet designed a synchronous-asynchronous FIFO based on the bisynchronous classical FIFO design using gray code, for the specific case of an asynchronous network-on-chip (NoC) interface.

Boundary synchronization

A third solution is to perform data synchronization at the borders of the locally synchronous island, without affecting the inner operation of locally synchronous blocks and without relying on FIFO buffers. For this purpose, designers can use standard two-flop, one-flop, predictive, or adaptive synchronizers for mesochronous systems, or locally delayed latching.

This method can achieve very reliable data transfer between locally synchronous blocks. On the other hand, such solutions generally increase latency and reduce data throughput, resulting in limited applicability for high-speed systems. Table 1 summarizes the properties of GALS systems' synchronization methods.

Advantages and limitations of GALS solutions

The scientific community has shown great interest in GALS solutions and architectures in the past two decades. However, this interest hasn't culminated in many commercial applications, despite all reported advantages. There are several reasons why standard design practice has not adopted GALS techniques.

Design and system integration issues

Many proposed solutions require programmable ring oscillators. This is an inexpensive solution that allows full control of the local clock. However, it has significant drawbacks. Ring oscillators are impractical for industrial use. They need careful calibration because they are very sensitive to process, voltage, and temperature variations. Moreover, embedded ring oscillators consume additional power through continuous switching of the chained inverters.

On the other hand, careful design of the delay line can reduce its power consumption to a level below that of a corresponding clock tree. In addition, this method can achieve very reliable data transfer between locally synchronous blocks. On the other hand, such solutions generally increase latency and reduce data throughput, resulting in limited applicability for high-speed systems.

Synchronous modules typically 50K-1M gates, so that the synchronous logic approach works well without requiring heroics. Examples ...
Stars denote FIFOs that create separate synchronous domains. An example of how architecture and circuits work together.
Recap: 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.
On Thursday

To be continued ...

Have fun in section!