Today: Dynamic Scheduling Overview

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

Example: IBM Power 5:

1. Fetch and decode: like static pipelines
2. Out-of-order processing

Today’s focus: execution unit

Dynamic Scheduling: A mix of 3 ideas

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

A common bus == long wires == slow?

Pipelines in theory: Wires are short, so clock periods can be short. “wiring by abutment”

Pipelines in practice:

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.

Last Time: Superpipelining & Superscalar

Q. Could adding pipeline stages reduce CPI for an application?
A. Yes, due to these problems:

- Extra branch delays: Branch prediction
- Extra load delays: Optimize code
- Structural hazards: Optimize code, add hardware

Possible Solution

Pipe with a barrier

Shared queues

Write hold:

Write

RF: L2 cache

Fig. 2. Possible Solution

Possible Solution

Pipe with a barrier

Shared queues

Write hold:

Write

RF: L2 cache

Fig. 2. Possible Solution
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

(1) Only one unit writes at a time (one source).
(2) All units may read the written values (many destinations).

Register Renaming

Consider this simple loop ...

Loop: LD F0, 0(R1) ; F0 = vector element
       ADD F4, F0, F2 ; add scalar from F2
       SD F4, 0(R1) ; store result
       SUBI R1, R1, 8 ; decrement pointer 8B (DW)
       BNEI 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.

...
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 ...

Interface between instruction fetch and execution.

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

ISS = Instruction Issue

Threads: A closer look

Next instr to “commit”, (complete).

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 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>1</td>
<td>14</td>
<td>87</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>1</td>
<td>14</td>
<td>87</td>
<td>100</td>
</tr>
</tbody>
</table>

ALU does the add, writing 100 to PR23.

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).

Final thought: Branch prediction required

Challenge: Precise Interrupt / Exception

Definition:
It must appear as if an interrupt is taken between two instructions (say I and I_{i+1})

- the effect of all instructions up to and including I, is totally complete
- no effect of any instruction after 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 ...

Dynamic scheduling and exceptions ...

Key observation: Only the architect 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.

Finally 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.