CIS 429/529
Complex Pipelining: Superscalar

Prof. Michel A. Kinsky
Complexity Pipelining

• Pipelining becomes complex when we want high performance in the presence of:
  ▸ Long latency or partially pipelined floating-point units
  ▸ Multiple function and memory units
  ▸ Memory systems with variable access time
CDC 6600 by Seymour Cray

- Year 1963
- A fast pipelined machine with 60-bit words
  - 128K word main memory capacity, 32 banks
- Ten functional units (parallel, unpipelined)
  - Floating Point: adder, 2 multipliers, divider
  - Integer: adder, 2 incrementers, ...
CDC 6600 by Seymour Cray

- Hardwired control (no microcoding)
  - Dynamic scheduling of instructions using a scoreboard
- Ten Peripheral Processors for Input/Output
  - A fast multi-threaded 12-bit integer ALU
- Very fast clock, 10 MHz (FP add in 4 clocks)
- >400,000 transistors, 750 sq. ft., 5 tons, 150 kW, novel freon-based technology for cooling
- Fastest machine in world for 5 years (until 7600)
  - Over 100 sold ($7-10M each)
CDC 6600: Datapath

Central Memory

Operand Regs
8 x 60-bit

Operand

Result

Address Regs
8 x 18-bit

Oprnd addr

Result addr

Index Regs
8 x 18-bit

10 Functional Units

IR

Inst. Stack
8 x 60-bit
A Load/Store Architecture

- Separate instructions to manipulate three types of reg:
  - 8 60-bit data registers (X)
  - 8 18-bit address registers (A)
  - 8 18-bit index registers (B)

- All arithmetic and logic instructions are reg-to-reg

- Only Load and Store instructions refer to memory!
CDC6600: Vector Addition

- Implicit operations:
  - Touching address registers 1 to 5 initiates a load
  - 6 to 7 initiates a store
  - Very useful for vector operations

\[
\begin{align*}
B0 & \leftarrow -n \\
\text{loop: } & \text{JZE B0, exit} \\
A0 & \leftarrow B0 + a0 \\
A1 & \leftarrow B0 + b0 \\
X6 & \leftarrow X0 + X1 \\
A6 & \leftarrow B0 + c0 \\
B0 & \leftarrow B0 + 1 \\
\text{jump loop}
\end{align*}
\]

Ai = address register
Bi = index register
Xi = data register

load X0
load X1
store X6
Floating Point ISA

• Interaction between the Floating point datapath and the Integer datapath is determined largely by the ISA

• MIPS ISA
  ‣ separate register files for FP and Integer instructions the only interaction is via a set of move instructions (some ISA’s don’t even permit this)
  ‣ separate load/store for FPR’s and GPR’s but both
  ‣ use GPR’s for address calculation
  ‣ separate conditions for branches
    • FP branches are defined in terms of condition codes
Floating Point Unit

• Much more hardware than an integer unit

• Single-cycle floating point unit is a bad idea - why?
  ‣ it is common to have several floating point units
  ‣ it is common to have different types of FPU's
    • Fadd, Fmul, Fdiv, ...
  ‣ an FPU may be pipelined, partially pipelined or not pipelined
  ‣ To operate several FPU’s concurrently the register file needs to have more read and write ports
Function Unit Characteristics

- Function units have internal pipeline registers
  - Operands are latched when an instruction enters a function unit
  - Inputs to a function unit (e.g., register file) can change during a long latency operation
Complex Pipeline Structure

IF → ID → Issue

GPR’s
FPR’s

ALU
Mem

Fadd
Fmul
Fdiv

WB
Complex Pipeline Control Issues

- Structural conflicts at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle

- Structural conflicts at the write-back stage due to variable latencies of different function units

- Out-of-order write hazards due to variable latencies of different function units
Complex In-Order Pipeline

- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
• How should we handle data hazards for very long latency operations?
Superscalar In-Order Pipeline

- Fetch two instructions per cycle; issue both simultaneously if one is integer/memory and other is floating-point (dependences?)
- Inexpensive way of increasing throughput
- The idea can be extended to wider issue but register file ports and bypassing costs grow quickly
  - Example 4-issue UltraSPARC
### Types of Data Hazards

- Consider executing a sequence of

  \[ r_k \leftarrow (r_i) \text{ op } (r_j) \]

  type of instructions

<table>
<thead>
<tr>
<th>Data-dependence</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[ r_3 \leftarrow (r_1) \text{ op } (r_2) ]</td>
<td>Read-after-Write</td>
</tr>
<tr>
<td>[ r_5 \leftarrow (r_3) \text{ op } (r_4) ]</td>
<td>(RAW) hazard</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Anti-dependence</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[ r_3 \leftarrow (r_1) \text{ op } (r_2) ]</td>
<td>Write-after-Read</td>
</tr>
<tr>
<td>[ r_1 \leftarrow (r_4) \text{ op } (r_5) ]</td>
<td>(WAR) hazard</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output-dependence</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[ r_3 \leftarrow (r_1) \text{ op } (r_2) ]</td>
<td>Write-after-Write</td>
</tr>
<tr>
<td>[ r_3 \leftarrow (r_6) \text{ op } (r_7) ]</td>
<td>(WAW) hazard</td>
</tr>
</tbody>
</table>
Detecting Data Hazards

- **Range and Domain of instruction i**
  - $R(i) = \text{Registers (or other storage) modified by instruction } i$
  - $D(i) = \text{Registers (or other storage) read by instruction } i$

- **Suppose instruction j follows instruction i in the program order. Executing instruction j before the effect of instruction i has taken place can cause a**
  - RAW hazard if $R(i) \cap D(j) \neq \emptyset$
  - WAR hazard if $D(i) \cap R(j) \neq \emptyset$
  - WAW hazard if $R(i) \cap R(j) \neq \emptyset$
Register vs. Memory Data Dependence

• Data hazards due to register operands can be determined at the decode stage

• Data hazards due to memory operands can be determined only after computing the effective address

  › store \( M[(r1) + \text{disp1}] \leftarrow (r2) \)
  › load \( r3 \leftarrow M[(r4) + \text{disp2}] \)
  › Does \((r1 + \text{disp1}) = (r4 + \text{disp2})\)?