CIS 429/529

Complex Pipelining

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
- **Address Regs** 8 x 18-bit
- **Index Regs** 8 x 18-bit
- **Result**
- **10 Functional Units**
- **Inst. Stack** 8 x 60-bit
- **Operand**
- **IR**

Diagram showing the flow of data between these components.
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

  6 3 3 3  
  opcode  i j k  
  Ri ← (Rj) op (Rk)

• Only Load and Store instructions refer to memory!

  6 3 3 18  
  opcode  i j  disp  
  Ri ← [(Rj) + disp]
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

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

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

GPR’s
FPR’s

Fadd
Fmul
Fdiv
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)
Superscalar In-Order Pipeline

- 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

**Data-dependence**

\[
\begin{align*}
  r_3 &\leftarrow (r_1 \text{ op } r_2) & \text{Read-after-Write} \\
  r_5 &\leftarrow (r_3 \text{ op } r_4) & \text{(RAW) hazard}
\end{align*}
\]

**Anti-dependence**

\[
\begin{align*}
  r_3 &\leftarrow (r_1 \text{ op } r_2) & \text{Write-after-Read} \\
  r_1 &\leftarrow (r_4 \text{ op } r_5) & \text{(WAR) hazard}
\end{align*}
\]

**Output-dependence**

\[
\begin{align*}
  r_3 &\leftarrow (r_1 \text{ op } r_2) & \text{Write-after-Write} \\
  r_3 &\leftarrow (r_6 \text{ op } r_7) & \text{(WAW) hazard}
\end{align*}
\]
Detecting Data Hazards

- Range and Domain of instruction $i$
  - $R(i) =$ Registers (or other storage) modified by instruction $i$
  - $D(i) =$ 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 \((r1 + \text{disp1}) = (r4 + \text{disp2})\) ?
Data Hazards: An Example

$I_1$ DIVD $f_6, f_6, f_4$

$I_2$ LD $f_2, 45(r_3)$

$I_3$ MULTD $f_0, f_2, f_4$

$I_4$ DIVD $f_8, f_6, f_2$

$I_5$ SUBD $f_{10}, f_0, f_6$

$I_6$ ADDD $f_6, f_8, f_2$

RAW Hazards

WAR Hazards

WAW Hazards
Instruction Scheduling

Valid orderings:

**in-order**

\[ I_1 \quad I_2 \quad I_3 \quad I_4 \quad I_5 \quad I_6 \]

**out-of-order**

\[ I_2 \quad I_1 \quad I_3 \quad I_4 \quad I_5 \quad I_6 \]

\[ I_2 \quad I_1 \quad I_3 \quad I_4 \quad I_5 \quad I_6 \]
## Out-of-order Completion

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Inputs</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_1$ DIVD</td>
<td></td>
<td>f6, f6, f4</td>
<td>4</td>
</tr>
<tr>
<td>$I_2$ LD</td>
<td></td>
<td>f2, 45(r3)</td>
<td>1</td>
</tr>
<tr>
<td>$I_3$ MULTD</td>
<td></td>
<td>f0, f2, f4</td>
<td>3</td>
</tr>
<tr>
<td>$I_4$ DIVD</td>
<td></td>
<td>f8, f6, f2</td>
<td>4</td>
</tr>
<tr>
<td>$I_5$ SUBD</td>
<td></td>
<td>f10, f0, f6</td>
<td>1</td>
</tr>
<tr>
<td>$I_6$ ADDD</td>
<td></td>
<td>f6, f8, f2</td>
<td>1</td>
</tr>
</tbody>
</table>

*in-order comp*  

|  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 |  |  |  |  |  |  |  |  |

*out-of-order comp*  

|  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | 1 | 4 | 3 | 5 | 5 | 4 | 6 | 5 | 6 | 6 |

* means completion
Little’s Law

Parallelism = Throughput \times Latency

or

\bar{N} = \bar{T} \times \bar{L}
Pipelined ILP Machine

Max Throughput, Six Instructions per Cycle

One Pipeline Stage

Latency in Cycles

Two Integer Units, Single Cycle Latency

Two Load/Store Units, Three Cycle Latency

Two Floating-Point Units, Four Cycle Latency

• How much instruction-level parallelism (ILP) required to keep machine pipelines busy?
Superscalar Control Logic Scaling

• Each issued instructions must make interlock checks against $W*L$ instructions, i.e., growth in interlocks $\propto W^*(W*L)$

• For in-order machines, $L$ is related to pipeline latencies

• For out-of-order machines, $L$ also includes time spent in instruction buffers (instruction window or ROB)

• As $W$ increases, larger instruction window is needed to find enough parallelism to keep machine busy $\rightarrow$ greater $L$
  
  ➔ Out-of-order control logic grows faster than $W^2$ ($\sim W^3$)
Out-of-Order Control Complexity

[ SGI/MIPS Technologies Inc., 1995 ]
VLIW: Very Long Instruction

- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
  - Parallelism within an instruction → no x-operation RAW check
  - No data use before data ready → no data interlocks
Early VLIW Machines

• **FPS API20B (1976)**
  ‣ scientific attached array processor
  ‣ first commercial wide instruction machine

• **Multiflow Trace (1987)**
  ‣ available in configurations with 7, 14, or 28 operations/instruction
  ‣ 28 operations packed into a 1024-bit instruction word

• **Cydrome Cydra-5 (1987)**
  ‣ 7 operations encoded in 256-bit instruction word
  ‣ rotating register file
VLIW Compiler

• The compiler:
  ‣ Schedules to maximize parallel execution
  ‣ Guarantees intra-instruction parallelism
  ‣ Schedules to avoid data hazards (no interlocks)
    • Typically separates operations with explicit NOPs
Intel EPIC IA-64

• EPIC is the style of architecture
  ‣ Explicitly Parallel Instruction Computing

• IA-64 is Intel’s chosen ISA
  ‣ IA-64 = Intel Architecture 64-bit
  ‣ An object-code compatible VLIW

• Itanium (aka Merced) is first implementation (cf. 8086)
  ‣ First customer shipment expected 1997 (actually 2001)
  ‣ McKinley, second implementation shipped in 2002
### IA-64 Instruction Format

- **Template bits describe grouping of these instructions with others in adjacent bundles**
- **Each group contains instructions that can execute in parallel**

<table>
<thead>
<tr>
<th>Instruction 2</th>
<th>Instruction 1</th>
<th>Instruction 0</th>
<th>Template</th>
</tr>
</thead>
</table>

#### 128-bit instruction bundle

- `bundle j-1`  
- `bundle j`  
- `bundle j+1`  
- `bundle j+2`  

- `group i-1`  
- `group i`  
- `group i+1`  
- `group i+2`
Problems with “Classic” VLIW

- Knowing branch probabilities
  - Profiling requires an significant extra step in build process
- Object code size
  - instruction padding wastes instruction memory/cache
  - loop unrolling/software pipelining replicates code
- Scheduling variable latency memory operations
  - caches and/or memory bank conflicts impose statically unpredictable variability
Problems with “Classical” VLIW

- Scheduling for statically unpredictable branches
  - optimal schedule varies with branch path
- Object-code compatibility
  - have to recompile all code for every machine, even for two machines in the same generation
Loop Unrolling

- Unroll inner loop to perform 4 iterations at once

  ▶ Is this code correct?

```c
for (i=0; i<N; i++)
```

```c
for (i=0; i<N; i+=4)
{
}
```
Superscalar and VLIW Machines

• Superscalar architecture implements instruction-level parallelism
  ‣ Single Instructions-Single Data (SISD) format

• VLIW machines show the advantages and limitations of instruction-level parallelism
  ‣ Multiple Instructions-Multiple Data (MIMD)

• We will further explore MIMD types of execution with multicore processors

• Single Instructions-Multiple Data (SIMD) execution with vector processor (next week)
Next Class

• SIMD and Vector Processors