Superscalar & RISC Architectures

COMP375 Computer Architecture and Organization
“The biggest risk is not taking any risk... In a world that’s changing really quickly, the only strategy that is guaranteed to fail is not taking risks.”

Mark Zuckerberg
Exam 3 Date Change

• The third exam in COMP375 will be on Friday, November 22, 2019
• This is the Friday before Thanksgiving instead of the Monday before Thanksgiving
Software and Hardware Solutions

• Pipeline hazards can be resolved by the hardware or the software

• Some processors (such as the Intel Pentium) produce correct results regardless of hazards. Hazards just slow execution

• Other processors assume that the software will avoid hazards. The compilers must prevent data hazards

• There is a general trend towards moving the intelligence to the software
Using A Pipeline

• Pipelining is *transparent* to high level language programmer

Disadvantages
• Assembler programmers who do not understand pipelines can produce inefficient code

Advantages
• Hardware automatically *stalls* the pipeline if items are not available, so programs are correct
• Compilers can rearrange code to avoid stalls
Pipeline Efficient Code

Inefficient

\[
\begin{align*}
w & += 7; \\
x & += 8; \\
y & += 9; \\
z & += 10; \\
\end{align*}
\]

Load R1, W
Add R1, 7
Store R1, W
Load R1, X
Add R1, 8
Store R1, X
Load R1, Y
Add R1, 9
Store R1, Y
Load R1, Z
Add R1, 10
Store R1, X

Pipeline Efficient (4 stage pipeline)

Load R1, W
Load R2, X
Load R3, Y
Load R4, Z
Add R1, 7
Add R2, 8
Add R3, 9
Add R4, 10
Store R1, W
Store R2, X
Store R3, Y
Store R4, Z

\[
\begin{align*}
w &= 7; \\
x &= 8; \\
y &= 9; \\
z &= 10; \\
\end{align*}
\]
Superscalar Processors

- Able to execute multiple instructions at a single time
- Uses multiple ALUs and execution resources
- Takes a sequential program and runs adjacent instructions in parallel if possible
- The Pentium Pro and following Intel processors are superscalar as are many other modern processors
Superscalar vs. Multiprocessor

• Superscalar machines execute regular sequential programs. The programmer is unaware of the parallelism.
• The programmer must explicitly code parallelism for multiprocessor systems.
• Simple instructions (arithmetic, load/store, conditional branch) can be initiated and executed independently.
• Equally applicable to RISC & CISC.
Superscalar Pipelining

Comparison of regular pipelining and superscalar pipelining
The superscalar processor described could run (at best)

A. same speed as simple processor at the same clock
B. 1.5 times as fast
C. twice as fast
D. three times as fast
E. faster than a speeding bullet
Instruction Level Parallelism

• As the processor is fetching and decoding instructions, it determines if they can be executed at the same time
• Instructions executed in parallel must be hazard free
• Some architectures, such as the Intel Itanium, explicitly define parallel instructions
Superscalar Hazards

- Data Dependency – Instructions executed in parallel cannot depend upon the results of the other instruction

\[
\begin{align*}
\text{Add} & \quad R1, \quad R2 \\
\text{Sub} & \quad R3, \quad \textcolor{red}{R1} \quad \text{Needs correct R1}
\end{align*}
\]

- These instructions cannot be executed in parallel
Superscalar Hazards

• Superscalar processors must have multiple execution resources. The number of resources (such as ALUs) limit the parallelism
• Control hazards impact the pipelining of both simple and superscalar processors
The impact of a control hazard is

A. greater with a superscalar processor than a non-superscalar processor
B. less with a superscalar processor than a non-superscalar processor
C. the same between superscalar and non-superscalar processors
D. All of the above
E. None of the above
Compiler Support

• An intelligent compiler can reorder the instructions so that adjacent instructions do not create data hazards
• Compilers for explicitly superscalar processors generate bundles of instructions that are executed in parallel
RISC Processors

- Modern processors take advantage of the architectural advances learned over the past decades
- Most new processors (i.e. Intel Itanium or ARM) are RISC processors
Design Alternatives

• **CISC** – Complex Instruction Set Computer
  • Pentium is the most popular example

• **RISC** – Reduced Instructions Set Computer
  • PowerPC, MIPS, SPARC, Intel Itanium, ARM

• No precise definition. The Intel Pentium 4 borrows many design ideas from RISC
Evolution of CISC Designs

• Motivation to efficiently use expensive resources
  – Processor
  – Memory
• High density code
  – Complex instructions
    • Hardware complexity is handled by *microprogramming*
    • Microprogramming is also helpful to
      – Reduce the impact of memory access latency
      – Offers flexibility, multiple members of the same family
  – Tailored to high-level language constructs
Simple Instructions

• Simple instructions are preferred
  – Complex instructions are mostly ignored by compilers
  – The Intel Pentium has several instructions that appear to be designed for Cobol programs
  – Most of the instructions used are very simple

• An implementation that supports complex instructions slows the execution of simple instructions
# Instruction Frequencies

<table>
<thead>
<tr>
<th></th>
<th>Dynamic Occurrence</th>
<th>Memory-Reference Weighted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pascal</td>
<td>C</td>
</tr>
<tr>
<td>Assign</td>
<td>45%</td>
<td>38%</td>
</tr>
<tr>
<td>Loop</td>
<td>5%</td>
<td>3%</td>
</tr>
<tr>
<td>Call</td>
<td>15%</td>
<td>12%</td>
</tr>
<tr>
<td>IF</td>
<td>29%</td>
<td>43%</td>
</tr>
<tr>
<td>other</td>
<td>6%</td>
<td>1%</td>
</tr>
</tbody>
</table>
Cache vs. Microcode

• Simple instructions can be implemented without microcode
• Cache memory is about as fast as microcode memory
• Executing several simple instructions to perform something takes about as long as executing a complex instruction requiring lots of microcode
Addressing Modes

• Complex addressing modes lead to variable length instructions
  – Leads to inefficient instruction decoding and scheduling
• Complex addressing modes varies the time required to fetch an operand
  – Reduces pipeline efficiency
Microcode and L1 cache are located in the

<table>
<thead>
<tr>
<th>Microcode</th>
<th>L1 cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. BIOS,</td>
<td>RAM</td>
</tr>
<tr>
<td>B. CPU chip,</td>
<td>CPU chip</td>
</tr>
<tr>
<td>C. CPU chip,</td>
<td>RAM</td>
</tr>
<tr>
<td>D. BIOS,</td>
<td>CPU chip</td>
</tr>
</tbody>
</table>
Register Operands

- Almost all RISC instructions use register operands
- Usually only one Load and one Store instruction access RAM
- Many RISC systems use three register operands
  \[ \text{ADD} \ R1, R2, R3 \]
- Avoids data hazard of putting the result back in the source register
Function Calls

• Most function calls have few arguments
  – Only 1.25% of the calls have more than 6 arguments

• Most functions have few local variables
  – More than 93% have less than 6 local scalar variables

• Function call/return: ~15% of HLL statements
  – Constitute 31–33% of machine instructions
  – Generate nearly half (45%) of memory references
Avoiding the Stack

• Stack operations are memory intensive
• Many RISC processors avoid using a stack and use registers instead
• Arguments are passed in the registers
• The function’s return address is stored in a register
• Registers are used to hold local variables
Register Stacks

• Many RISC processors have a large number of registers, not all of which are visible at any one time
• The mapping of register X to a hardware register changes when a function is called
Before a Function Call

• Assume the assembly language programmer sees 32 registers

• Before a function call, arguments and the return address are put in registers R24 to R31
After a Function Call

• After a function call, the input arguments and the return address are available in registers R8 to R15
• R16 to R23 are used for local variables
• R24 to R31 contain arguments to next function
After another Function Call

- After another function call, the input arguments and the return address are again available in registers R8 to R15
- Return values are also put in R8 to R15 upon function return
After Function Return

- After the function return, the return values are available in registers R24 to R31
A difference between RISC and CICS is

A. CISC function calls typically store values on the stack in RAM
B. RISC functions use fewer parameters
C. CISC save parameters in the microcode store
D. RISC processors do not use call instructions
E. All the above
RISC Design Principles

• Simple operations
  – Simple instructions that can execute in one cycle

• Register-to-register operations
  – Only load and store operations access memory
  – Rest of the operations on a register-to-register basis

• Simple addressing modes
  – A few addressing modes (1 or 2)
RISC Design Principles

• Large number of registers
  – Needed to support register-to-register operations
  – Minimize the procedure call and return overhead

• Fixed-length instructions
  – Facilitates efficient instruction execution

• Simple instruction format
  – Fixed boundaries for various fields
RISC Design Principle

• Start an instruction every cycle

• Simple, fixed length instructions are easy to pipeline

• Only two instruction have memory operands all other operands are in registers

• Delayed branches
## Example Differences

<table>
<thead>
<tr>
<th></th>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VAX 11/780</td>
<td>Intel 486</td>
</tr>
<tr>
<td># instructions</td>
<td>303</td>
<td>235</td>
</tr>
<tr>
<td>Addr. modes</td>
<td>22</td>
<td>11</td>
</tr>
<tr>
<td>Inst. size (bytes)</td>
<td>2-57</td>
<td>1-12</td>
</tr>
<tr>
<td>GP registers</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>

chart © Dandamudi
RISC Traits

• Pipelined
• Simple uniform instructions
• Few instructions
• No microcode
• Few addressing modes
• Load/Store architecture
• Many identical general purpose registers
• Sliding register stack
• Delayed branches
• Fast
CICS advantages include:

A. Sliding register stack
B. Instructions designed to match high level language features
C. Load/Store architecture
D. Large microcode memory
Exam 3 Date Change

• The third exam in COMP375 will be on Friday, November 22, 2019
• This is the Friday before Thanksgiving instead of the Monday before Thanksgiving