Principles Of Computer Design

1. Make the Common Case Fast – more frequent code dominates the execution

\[
\text{speedup}_{\text{overall}} = \frac{\text{Performance}_{\text{new}}}{\text{Performance}_{\text{old}}} \quad \text{speedup}_{\text{overall}} = \frac{\text{Time}_{\text{old}}}{\text{Time}_{\text{new}}}
\]

Amdahl’s law

\[
\text{Time}_{\text{new}} = \text{Time}_{\text{old}}(1-f) + \frac{\text{Time}_{\text{old}}}{\text{speedup}_{\text{enhanced}}} \times f
\]

Example

We are considering a new CPU that makes Web server applications run 10 times faster. Original CPU serves Web pages 40% of time, the rest is waiting for I/O. (Answer: 1.56)

CPU Performance Equation

\[
\text{CPU } \_ \_ \text{time} = \text{clock } \_ \_ \text{cycles } \_ \_ \text{for } \_ \_ \text{a } \_ \_ \text{program } \_ \_ \text{cycle } \_ \_ \text{time}
\]

\[
\text{CPU } \_ \_ \text{time} = \text{IC } \times \text{CPI } \_ \_ \text{cycle } \_ \_ \text{time}
\]

\[
\text{CPU } \_ \_ \text{time} = \sum_{i=1}^{\text{IC}} \text{IC}_i \times \text{CPI}_i \times \text{cycle } \_ \_ \text{time}
\]

\[
\text{CPI} = \frac{\sum_{i=1}^{\text{IC}} \text{IC}_i \times \text{CPI}_i}{\text{IC}}
\]

Example

We are considering two alternatives for improving graphics engine performance:

1. Speeding up floating-point square root (FPSQR) operation, which is used 20% of time, by a factor of 10, and
2. Speeding up all FP instructions, which are used 50% of time, by a factor 1.6

Both alternatives cost the same. Which one is better? (Answer: the second one)

Example

Suppose we have made the following measurements:

- Frequency of FPSQR = 2%
- Frequency of all FP operations = 25%
- CPI_{FP} = 4.0
- CPI_{FPSQR} = 20
- CPI_{OTHER} = 1.33

First design alternative is to decrease CPI_{FPSQR} to 2, and the second is to decrease CPI_{FP} to 2.5. Which one is better? (Answer: the second)

Principles Of Computer Design

2. Principle of locality:
   - Temporal: recently accessed data will be accessed in the future
   - Spatial: adjacent data will be accessed

3. Take advantage of parallelism:
   - Pipelining, multiple processors, associative memory
What is an Instruction?

- **Opcode:** Operation (ADD, MULT, …)
- **Type of operands:** 
- **Result**
- **Location type:**
- **Location:**

What is Instruction Set Architecture?

- Set of operations that processor will support: ADD, MULT, SUB …
- Location of operands: memory, registers, stack …
- Location of the result
- Number of operands in each instruction
- Range of operands
- Length of an instruction

Goals for Instruction Set Design

- Short instructions: minimize program size
- Good instruction density: minimize program size
- Fast operations
- Simple circuitry
- Compiler optimisation

Classification by Type of Internal Storage

- Stack
- Accumulator
- General purpose register
  - Register-memory
  - Register-register (load-store)
  - Memory-memory

Stack Architecture

```
stack  C=A+B
memory

TOS
Push A
Push B
Add
Pop C

ALU
```

```
stack  C=A+B
memory

TOS
Push A
Push B
Add
Pop C

ALU
```
Stack Architecture

- Stack:\[ C = A + B \]
- Push A
- Push B
- Add
- Pop C

Classification by Type of Internal Storage

- Stack architecture:
  - Special instructions to access memory: push, pop
  - Operands are loaded from memory onto the stack
  - ALU performs operation upon the last two elements on the stack
  - Both operands and location of result are implicit
  - First operand is removed from the stack, result is written in the place of the second operand
  - Result has to be explicitly stored back into memory

Accumulator Architecture

- Accumulator:\[ C = A + B \]
- Load A
- Add B
- Store C
Accumulator Architecture

C = A + B

Load A
Add B
Store C

Classification by Type of Internal Storage

Accumulator architecture:
- Any operation can access memory
- First operand is loaded from the memory into accumulator
- Operation is performed on the accumulator and the second operand (from the memory)
- First operand and location of result are implicit
- Result is written into accumulator
- Result has to be explicitly stored back into memory

Register-Memory Architecture

C = A + B

Load R1, A
Add R3, R1, B
Store R3, C

Register-Memory Architecture

C = A + B

Load R1, A
Add R3, R1, B
Store R3, C
Register-Memory Architecture

\[ C = A + B \]

- Load R1, A
- Add R3, R1, B
- Store R3, C

Classification by Type of Internal Storage

- Register-memory architecture:
  - Any operation can access memory
  - First operand is loaded from the memory into a register
  - Operation is performed on the register and the second operand (from the memory)
  - Both operands and location of result are **explicit**
  - Result is written into a register
  - Result has to be explicitly stored back into memory

Load-Store Architecture

\[ C = A + B \]

- Load R1, A
- Load R2, B
- Add R3, R1, R2
- Store R3, C

Load-Store Architecture

\[ C = A + B \]

- Load R1, A
- Load R2, B
- Add R3, R1, R2
- Store R3, C
Load-Store Architecture

CIS 662 – Computer Architecture – Fall 2004 - Class 2 – 9/9/04

ALU

C=A+B
Load R1, A
Load R2, B
Add R3, R1, R2
Store R3, C

Memory-Memory Architecture

CIS 662 – Computer Architecture – Fall 2004 - Class 2 – 9/9/04

ALU

C=A+B
Add C, A, B

Which Architecture is the Best?

CIS 662 – Computer Architecture – Fall 2004 - Class 2 – 9/9/04

➢ Early computers used stack, accumulator, register-memory and memory-memory
➢ Current computers use load-store:
   ➢ Register access is faster
   ➢ Registers allow for compiler optimisations (out of order execution)
   ➢ Registers can be used to hold all the variables relevant for a specific code segment – all operations are faster
   ➢ Registers can be named with fewer bits than memory

Classification by Type of Internal Storage

➢ Register-register (load-store) architecture:
   ➢ Special instructions to access memory: load, store
   ➢ First operand is loaded from the memory into a register
   ➢ Second operand is loaded from the memory into a register
   ➢ Operation is performed on the registers
   ➢ Both operands and location of result are explicit
   ➢ Result is written into a register, and has to be explicitly stored back into memory

Memory-memory architecture: (obsolete)

➢ Operation is performed on the memory locations
➢ Result is written into the memory

Classification of GPR Architectures by Number of Operands

➢ Three: result, operand₁, and operand₂
➢ Two: result = operand₁, and operand₂
Classification of GPR Architectures by Number of Memory References

- Maximum number of operands = 3:
  - All three in memory (3,3) – memory-memory
  - All three in registers (0,3) – load-store
  - One operand in memory (1,3) – register-memory

- Maximum number of operands = 2:
  - Both in memory (2,2) – memory-memory architecture
  - One operand in memory (1,2) – register-memory

Which Architecture is the Best?

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-register</td>
<td>Simple</td>
<td>Higher instruction count</td>
</tr>
<tr>
<td>(0, 3)</td>
<td>Fixed-length instruction</td>
<td>Longer programs</td>
</tr>
<tr>
<td></td>
<td>Similar CPI</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Compiler optimizations</td>
<td></td>
</tr>
<tr>
<td>Register-memory</td>
<td>Better instruction density</td>
<td>Source operand destroyed</td>
</tr>
<tr>
<td>(1, 2)</td>
<td></td>
<td>Longer instructions</td>
</tr>
<tr>
<td>Memory-memory</td>
<td>Best instruction density</td>
<td>CPI vary by operand location</td>
</tr>
<tr>
<td>(2, 2) or (3, 3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary

- Support GPR architecture
- Register-register to facilitate pipelining

Homework

- Due Thursday, 9/16 by the end of the class
- Do exercises 1.2, 1.3, 1.14 (assume that the base ratio for machine M is calculated as \(\text{time}(M)/\text{time(Ref)}\)), 1.25