CPU Performance Equation

Now assume that we can calculate average number of cycles per instruction – CPI. IC is the number of instructions in the program and CC is cycle time.

\[
CPU\_time = IC \times CPI \times CC
\]

Now assume that we can calculate average number of cycles per instruction for each type of instructions \(i\) – CPI\(_i\). IC is the number of instructions of this type in the program.

\[
CPU\_time = \sum_{i=1}^{\text{IC}} IC_i \times CPI_i \times CC
\]

Locality Principles

- Temporal: recently accessed data will be accessed in the future
  - Can you think of a piece of code that exhibits temporal locality?
- Spatial: adjacent data will be accessed
  - Can you think of a piece of code that exhibits spatial locality?

Parallelism

If we can break the code into \(N\) pieces and execute them in parallel we can reduce the overall execution time by at most \(N\) times

- Pipelining
  - Instructions are divided into smaller steps
  - Units inside the CPU are dedicated to one step – think of assembly line in a factory
- Multiple processors
  - Each working on a set of instructions
- Associative memory
  - Look for data in all entries simultaneously

Exercise

Suppose that we have made the following measurements:

- Frequency of FPSQR instructions = 2% of IC
- Frequency of all FP instructions (including FPSQR) = 25% of IC
- \(CPI_{FP} = 4.0\)
- \(CPI_{FPSQR} = 20\)
- \(CPI_{NONFP} = 1.33\)

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? Can we apply Amdahl’s law to this?

From Source to Assembly Code

\[
a+b=c
\]

\[
\text{compiler}
\]

\[
\text{LOAD R1, a}
\]

\[
\text{LOAD R2, b}
\]

\[
\text{ADD R3, R1, R2}
\]

\[
\text{STORE R3, c}
\]
What Is an Instruction?

```plaintext
opcode: 
- location (memory or register) 
- which operation (ADD, MULT) 
- type (INT, FP)

ADD R1, R3, R4
ADD F1, F2, F3
SUB R1, R2, (100)
FADD R1, R2, R3
```

Instruction Set Architecture

- Which operations will a processor support?
  - ADD, MULT, SUB ...
- Where can ALU operands reside?
  - Memory, registers, stack
- Where can ALU result be stored?
  - Memory, registers, stack
- How many operands in each instruction?
  - Fixed or as many as we want
- Range of operands
- Length of an instruction
  - Fixed or variable

Goals for Instruction Set Design

- Short instructions
  - Minimize instruction width – minimize program size
- Good instruction density
  - Minimize instruction count – minimize program size
- Fast operations
  - ADD (100), (200), (40) longer than ADD R1, R2, R3
- Simple circuitry
- Compiler optimisation

Instruction Classification:
By Type of Internal Storage (Where are ALU operands/result?)

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

Stack Architecture

```
stack

C=A+B
Push A
Push B
Add
Pop C
```

Inside a CPU
Stack Architecture

- Special instructions to access memory
  - push, pop
- Operands 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

- C = A + B
- Load A
- Add B
- Store C
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

- 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
Any operation can access memory. First operand is loaded from 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

- 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

- Memory-memory architecture: (obsolete)
  - Operation is performed on the memory locations
  - Result is written into the memory

Which Architecture Is the Best?

- Early computers used stack, accumulator, register-memory and memory-memory
- Current computers use load-store:
  - Register access is faster
  - Registers can be named with fewer bits than memory
  - 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
Exercise
Write the code segment

\[ x \times y = z \]
\[ z - w = z \]

- In stack architecture
- In accumulator architecture
- In register-memory architecture
- In load-store architecture
- In memory-memory architecture

GPR Architecture Classification: By Number Of Operands

- Three
  - All three in memory (3,3)
  - memory-memory
  - All three in registers (0,3)
  - load-store
  - One operand in memory (1,3)
  - register-memory

- Two
  - Both in memory (2,2)
  - memory-memory architecture
  - One operand in memory (1,2)
  - register-memory

GPR Architecture Classification: By Number Of Memory References

- Three operands
  - All three in memory (3,3)
  - memory-memory
  - All three in registers (0,3)
  - load-store
  - One operand in memory (1,3)
  - register-memory

- Two operands
  - 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, fixed length instruction, similar CPI, compiler optimizations</td>
<td>High instruction count, longer programs</td>
</tr>
<tr>
<td>Register-memory</td>
<td>Better instruction density</td>
<td>Source operand is destroyed, longer instructions, CPI vary by operand location</td>
</tr>
<tr>
<td>Memory-memory</td>
<td>Best instruction density</td>
<td>Longest instructions, CPI vary by operand location, memory bottleneck</td>
</tr>
<tr>
<td>Memory-memory (2,2) or (3,3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>