Unit 18

Improving Performance
Caching and Pipelining
Review of Software Execution (1)

- Recall:
  1. High level code is compiled to assembly/machine code instructions
  2. When executed, CPU uses PC register to know what instruction to fetch/execute next
  3. PC is incremented and used to fetch the next instruction
  4. Registers are used as temp. storage of variables
  5. Variables "live" in memory & must be updated

```c
#define MAX 1000000;
int data[MAX];

void init()
{
    int i;
    for(i=0; i < MAX; i++)
    {
        data[i] = i;
    }
}

int main()
{
    init();
    /* More work */
    return 0;
}
```

Note this `init()` function initializes a million element array with values 0 to 999999

Next slide shows the execution of the 'init' for loop at several points in time (with some details omitted).
Review of Software Execution (2)

**Time 1: Store**

<table>
<thead>
<tr>
<th>Bus (Addr, Data, Control)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>PC 800a5</td>
</tr>
<tr>
<td>Regs</td>
</tr>
<tr>
<td>ALU</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>init:</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>L1: // i &lt;=&gt; s2</td>
</tr>
<tr>
<td>store s2,(sf)</td>
</tr>
<tr>
<td>add s2,1</td>
</tr>
<tr>
<td>compare s2,MAX</td>
</tr>
<tr>
<td>jump NZ,L1</td>
</tr>
</tbody>
</table>

**Time 2: Increment i**

<table>
<thead>
<tr>
<th>Bus (Addr, Data, Control)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>PC 800a6</td>
</tr>
<tr>
<td>Regs</td>
</tr>
<tr>
<td>ALU</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>init:</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>L1: // i &lt;=&gt; s2</td>
</tr>
<tr>
<td>store s2,(sf)</td>
</tr>
<tr>
<td>add s2,1</td>
</tr>
<tr>
<td>compare s2,MAX</td>
</tr>
<tr>
<td>jump NZ,L1</td>
</tr>
</tbody>
</table>

**Time 3: Loop & repeat**

<table>
<thead>
<tr>
<th>Bus (Addr, Data, Control)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>PC 800a5</td>
</tr>
<tr>
<td>Regs</td>
</tr>
<tr>
<td>ALU</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>init:</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>L1: // i &lt;=&gt; s2</td>
</tr>
<tr>
<td>store s2,(sf)</td>
</tr>
<tr>
<td>add s2,1</td>
</tr>
<tr>
<td>compare s2,MAX</td>
</tr>
<tr>
<td>jump NZ,L1</td>
</tr>
</tbody>
</table>

**Time 4**

<table>
<thead>
<tr>
<th>Bus (Addr, Data, Control)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>PC 800a6</td>
</tr>
<tr>
<td>Regs</td>
</tr>
<tr>
<td>ALU</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>init:</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>L1: // i &lt;=&gt; s2</td>
</tr>
<tr>
<td>store s2,(sf)</td>
</tr>
<tr>
<td>add s2,1</td>
</tr>
<tr>
<td>compare s2,MAX</td>
</tr>
<tr>
<td>jump NZ,L1</td>
</tr>
</tbody>
</table>

Some details have been left out (how does sf get initialized, incremented, etc.)
Improving Performance

• We want to improve the performance of our computation

• Question: What are we referring to when we say "performance"?
  – Speed
  – Energy consumption
  – Cost?

• We will primarily consider speed in this discussion
How Do We Measure Speed

• **Fundamental Measurement:** TIME
  – Absolute time from **start** to **finish**
  – To compare two alternative systems (HW + SW) and their performance, start a timer when you begin a task and stop it when the task ends
  – Do this for both systems and compare the resulting times

• We call this the **latency** of the system and it works great from the perspective of the single, overall task
  – If system A completes the task in 2 seconds and system B requires 3 seconds, then system A is clearly superior

• But when we dig deeper and realize that the single, overall task is likely made of many small tasks, we can consider more than just latency
Performance Depends on View Point?!

- What's faster to get from point A to point B?
  - A 747 Jumbo Airliner
  - An F-22 fighter jet

- If only 1 person needs to get from point A to point B, then the F-22
  - This is known as **latency [units of seconds]**
  - Time from the start of an operation until it completes

- If 200 people need to get from point A to point B, the 747 looks much better
  - This is known as **throughput [jobs/second]**

- The **overall** execution time (latency) may best be improved by increasing throughput and not the latency of individual tasks
Improving Latency and Throughput

CACHING AND PIPELINING
Hardware Techniques

• We can add hardware or reorganize our hardware to improve throughput and latency of individual tasks in an effort to reduce the total latency (time) to finish the overall task

• We will look at two examples:
  – Caching: Improves latency
  – Pipelining: Improves throughput
Caching

• **Cache (def.)** – "to store away in hiding or for future use"

• Primary idea
  – The first time you access or use something you expend the full amount of time to get it
  – However, store it someplace (i.e. in a cache) you can get it more quickly the next time you need it
  – The next time you need something check if it is in the cache first
  – If it is in the cache, you can get it quickly; else go get it expending the full amount of time (but then save it in the cache)

• Examples:
  – Web-browser
  – Checking out a book from the library
  – Your refrigerator
Cache Overview

- Remember what register are used for?
  - Quick access to copies of data
  - Only a few (32 or 64) so that we can access really quickly
  - Controlled by the software/compiler

- Cache memory is a small-ish, (kilobytes to a few megabytes) "fast" memory usually built onto the processor chip

- Will hold copies of the latest data & instructions accessed by the processor

- Managed by the HW
  - Transparent to the software
Cache Operation (1)

- When processor wants data or instructions it always checks in the cache first
- If it is there, fast access
- If not, get it from memory
- Memory will also supply surrounding data since it is likely to be needed soon
  - Why?
  - Things like arrays & code (instructions) are commonly accessed sequentially
Cache Operation (2)

- When processor asks for the data again or for the next data value in the array (or instruction of the code) the cache will likely have it
- Questions?

**Main point:** Caching reduces the latency of memory accesses which improves overall program performance.
Memory Hierarchy & Caching

- Use several levels of faster and faster memory to hide latency of larger levels

- Registers
  - L1 Cache: ~ 1ns
  - L2 Cache: ~ 10ns
  - Main Memory: ~ 100 ns

- Unit of Transfer: 8-64 bytes

- Smaller → Faster, Less Expensive → Larger

- 8-64 bits of transfer
Pipelining

• We'll now look at a hardware technique called **pipelining** to improve **throughput**

• The key idea is to **overlap** the processing of multiple "items" (either data or instructions)
Example

• Suppose you are asked to build dedicated hardware to perform some operation on all 100 elements of some arrays
• Suppose the operation \((A[i]+B[i])/4\) takes 10 ns to perform
• How long would it take to process the entire arrays: **1000 ns**
  – Can we improve?

```plaintext
for(i=0; i < 100; i++)
    C[i] = (A[i] + B[i]) / 4;
```

Clock Freq. = \(1/10\text{ns} = 100 \text{ MHz}\) (longest path from register to register)
Pipelining Example

- Pipelining refers to insertion of registers to split combinational logic into smaller stages that can be overlapped in time (i.e. create an assembly line)

<table>
<thead>
<tr>
<th></th>
<th>Stage 1</th>
<th>Stage 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Cycle 0</td>
<td>A[0] + B[0]</td>
<td></td>
</tr>
</tbody>
</table>

for(i=0; i < 100; i++)
C[i] = (A[i] + B[i]) / 4;

Time for 0th elements to complete: 10ns
Time between each of the remaining 99 element completing: 5ns
Total: 10 + 99*5 = 505ns

Define:

\[
\text{speedup} = \frac{\text{Original time}}{\text{Improved Time}}
\]

\[
\text{speedup} = \frac{1000\text{ns}}{505\text{ns}} \approx 2x
\]

Clock freq: = 200MHz = 1/5ns
Need for Registers

• Provides separation between combinational functions
  – Without registers, fast signals could “catch-up” to data values in the next operation stage

Performing an operation yields signals with different paths and delays

We don’t want signals from two different data values mixing. Therefore we must collect and synchronize the values from the previous operation before passing them on to the next.
Pipelining Example

• By adding more pipelined stages we can improve throughput
• Have we affected the latency of processing individual elements? No!!!
• Questions/Issues?
  – Balancing stage delays
  – Overhead of registers (Not free to split stages)
    • This limits how much we can split our logic

```c
for(i=0; i < 100; i++)
  C[i] = (A[i] + B[i]) / 4;
```

Time for 0th elements to complete: 10 ns
Time between each of the remaining 99 element completing: 2.5 ns
Total: 257.5 ns

\[
\text{speedup} = \frac{1000\text{ns}}{257.5\text{ns}} \approx 4x
\]
Non-Pipelined Processors

• Currently we know our processors execute software 1 instruction at a time

• 3 steps/stages of work for each instruction are:
  – Fetch
  – Decode
  – Execute
Pipelined Processors

- By breaking our processor hardware for instruction execution into stages we can overlap these stages of work.
- Latency for a single instruction is the same.
- Overall throughput, and thus total latency, are greatly improved.
More and More Stages

- We can break the basic stages of work into substages to get better performance.
- In doing so our clock period goes down; frequency goes up.
- All kinds of interesting issues come up though when we overlap instructions and our discussed in future CENG courses.

![Diagram showing overlapping instructions and clock periods](image)

Clock freq. = 1/10ns = 100MHz

Clock freq. = 1/5ns = 200MHz
Summary

• By investing extra hardware we can improve the overall latency of computation

• Measures of performance:
  – Latency is start to finish time
  – Throughput is tasks completed per unit time (measure of parallelism)

• Caching reduces latency by holding data we will use in the future in quickly accessible memory

• Pipelining improves throughput by overlapping processing of multiple items (i.e. an assembly line)