Table of Contents

1 Misc Information
2 Parallel Hardware Archs. - Part 1
   - Von Neumann Computer
   - Von Neumann Model Modifications
   - Von Neumann Mods: Memory Caching
   - Von Neumann Mods: Virtual Memory
   - Von Neumann Mods: Instruction Level Parallelism
3 Next Time
HW #1 has been posted; due date is 02/03/15

HW #2 will be posted; due date is 02/10/15 (using statistics on HW1 code)

Student cluster tuckoo accounts have been created for enrolled students.

Reading: Ch2, Pacheco 2011 text:
"An Introduction to Parallel Programming."
HPC Hardware: Blue Gene/L Hardware
Von Neumann electronic digital computer

- Central processing unit:
  - arithmetic logic unit (ALU)
  - processor registers
- Control unit:
  - instruction register
  - program counter
- Memory unit:
  - data
  - instructions
- External mass storage
- Input and output mechanisms

The von Neumann Architecture

Figure 2.1
Main memory

- This is a collection of locations, each of which is capable of storing both instructions and data.

- Every location consists of an address, which is used to access the location, and the contents of the location.
Central processing unit (CPU)

- Divided into two parts.

- **Control unit** - responsible for deciding which instruction in a program should be executed. *(the boss)*

- **Arithmetic and logic unit (ALU)** - responsible for executing the actual instructions. *(the worker)*
Key terms

- **Register** – very fast storage, part of the CPU.

- **Program counter** – stores address of the next instruction to be executed.

- **Bus** – wires and hardware that connects the CPU and memory.
Parallel Hardware Archs. - Part 1

Von Neumann Computer

- Memory
- Fetch/Read
- CPU

- Memory
- Write/Store
- CPU
An operating system “process”

- An instance of a computer program that is being executed.
- Components of a process:
  - The executable machine language program.
  - A block of memory.
  - Descriptors of resources the OS has allocated to the process.
  - Security information.
  - Information about the state of the process.
Multitasking

- Gives the illusion that a single processor system is running multiple programs simultaneously.
- Each process takes turns running. (time slice)
- After its time is up, it waits until it has a turn again. (blocks)
Threading

- Threads are contained within processes.
- They allow programmers to divide their programs into (more or less) independent tasks.
- The hope is that when one thread blocks because it is waiting on a resource, another will have work to do and can run.
A process and two threads

- the "master" thread
- starting a thread is called **forking**
- terminating a thread is called **joining**

Figure 2.2
MODIFICATIONS TO THE VON NEUMANN MODEL
Basics of caching

■ A collection of memory locations that can be accessed in less time than some other memory locations.

■ A CPU cache is typically located on the same chip, or one that can be accessed much faster than ordinary memory.
Principle of locality

- Accessing one location is followed by an access of a nearby location.

- **Spatial locality** – accessing a nearby location.

- **Temporal locality** – accessing in the near future.
Principle of locality

```c
float z[1000];
...
sum = 0.0;
for (i = 0; i < 1000; i++)
    sum += z[i];
```
Levels of Cache

smallest & fastest

L1

L2

L3

largest & slowest
Cache hit

fetch x

L1 \ x \ \text{sum}

L2 \ y \ z \ \text{total}

L3 \ A[\text{radius r1 center}]

Cache miss

fetch x

L1 \ y \ \text{sum}

L2 \ \text{r1 z total}

L3 \ A[\text{radius center}]

Copyright © 2010, Elsevier Inc. All rights Reserved
Issues with cache

- When a CPU writes data to cache, the value in cache may be inconsistent with the value in main memory.

- **Write-through** caches handle this by updating the data in main memory at the time it is written to cache.

- **Write-back** caches mark data in the cache as *dirty*. When the cache line is replaced by a new cache line from memory, the *dirty* line is written to memory.
Cache mappings

- **Full associative** – a new line can be placed at any location in the cache.

- **Direct mapped** – each cache line has a unique location in the cache to which it will be assigned.

- **n-way set associative** – each cache line can be placed in one of \(n\) different locations in the cache.
**n-way set associative**

- When more than one line in memory can be mapped to several different locations in cache we also need to be able to decide which line should be replaced or evicted.
Example

Table 2.1: Assignments of a 16-line main memory to a 4-line cache
Caches and programs

```c
double A[MAX][MAX], x[MAX], y[MAX];

/* Initialize A and x, assign y = 0 */

/* First pair of loops */
for (i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++)
        y[i] += A[i][j]*x[j];

/* Second pair of loops */
for (j = 0; j < MAX; j++)
    for (i = 0; i < MAX; i++)
        y[i] += A[i][j]*x[j];
```

<table>
<thead>
<tr>
<th>Cache Line</th>
<th>Elements of A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A[0][0] A[0][1] A[0][2] A[0][3]</td>
</tr>
</tbody>
</table>
Virtual memory (1)

- If we run a very large program or a program that accesses very large data sets, all of the instructions and data may not fit into main memory.

- Virtual memory functions as a cache for secondary storage.
Virtual memory (2)

- It exploits the principle of spatial and temporal locality.
- It only keeps the active parts of running programs in main memory.
Virtual memory (3)

- **Swap space** - those parts that are idle are kept in a block of secondary storage.

- **Pages** – blocks of data and instructions.
  - Usually these are relatively large.
  - Most systems have a fixed page size that currently ranges from 4 to 16 kilobytes.
Virtual memory (4)

program A

program B

program C

main memory
Virtual page numbers

- When a program is compiled its pages are assigned *virtual* page numbers.

- When the program is run, a table is created that maps the virtual page numbers to physical addresses.

- A page table is used to translate the virtual address into a physical address.
Page table

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th></th>
<th></th>
<th></th>
<th>Byte Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Page Number</td>
<td>31</td>
<td>30</td>
<td>...</td>
<td>13</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Translation-lookaside buffer (TLB)

- Using a page table has the potential to significantly increase each program’s overall run-time.

- A special address translation cache in the processor.
Translation-lookaside buffer (2)

- It caches a small number of entries (typically 16–512) from the page table in very fast memory.

- Page fault – attempting to access a valid physical address for a page in the page table but the page is only stored on disk.
Instruction Level Parallelism (2)

- **Pipelining** - functional units are arranged in stages.

- **Multiple issue** - multiple instructions can be simultaneously initiated.
Pipelining
Pipelining example (1)

<table>
<thead>
<tr>
<th>Time</th>
<th>Operation</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fetch operands</td>
<td>$9.87 \times 10^4$</td>
<td>$6.54 \times 10^5$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Compare exponents</td>
<td>$9.87 \times 10^4$</td>
<td>$6.54 \times 10^3$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Shift one operand</td>
<td>$9.87 \times 10^4$</td>
<td>$0.654 \times 10^4$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Add</td>
<td>$9.87 \times 10^4$</td>
<td>$0.654 \times 10^4$</td>
<td>$10.524 \times 10^4$</td>
</tr>
<tr>
<td>5</td>
<td>Normalize result</td>
<td>$9.87 \times 10^4$</td>
<td>$0.654 \times 10^4$</td>
<td>$1.0524 \times 10^5$</td>
</tr>
<tr>
<td>6</td>
<td>Round result</td>
<td>$9.87 \times 10^4$</td>
<td>$0.654 \times 10^4$</td>
<td>$1.05 \times 10^3$</td>
</tr>
<tr>
<td>7</td>
<td>Store result</td>
<td>$9.87 \times 10^4$</td>
<td>$0.654 \times 10^4$</td>
<td>$1.05 \times 10^5$</td>
</tr>
</tbody>
</table>

Add the floating point numbers $9.87 \times 10^4$ and $6.54 \times 10^3$
Pipelining example (2)

float x[1000], y[1000], z[1000];

for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];

- Assume each operation takes one nanosecond ($10^{-9}$ seconds).
- This for loop takes about 7000 nanoseconds.
Pipelining (3)

- Divide the floating point adder into 7 separate pieces of hardware or functional units.
- First unit fetches two operands, second unit compares exponents, etc.
- Output of one functional unit is input to the next.
Pipelining (4)

Table 2.3: Pipelined Addition.
Numbers in the table are subscripts of operands/results.
Pipelining (5)

- One floating point addition still takes 7 nanoseconds.

- But 1000 floating point additions now takes 1006 nanoseconds!
Multiple Issue (1)

- Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program.

```plaintext
for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];
```

![Diagram of adders and functional units](Image)
Multiple Issue (2)

- **static** multiple issue - functional units are scheduled at compile time.

- **dynamic** multiple issue – functional units are scheduled at run-time.

*superscalar*
Speculation (1)

- In order to make use of multiple issue, the system must find instructions that can be executed simultaneously.

- In speculation, the compiler or the processor makes a guess about an instruction, and then executes the instruction on the basis of the guess.
Speculation (2)

\[
z = x + y; \\
\text{if} (z > 0) \\
\quad w = x; \\
\text{else} \\
\quad w = y;
\]

Z will be positive

If the system speculates incorrectly, it must go back and recalculate \( w = y \).
Hardware multithreading (1)

- There aren’t always good opportunities for simultaneous execution of different threads.

- Hardware multithreading provides a means for systems to continue doing useful work when the task being currently executed has stalled.
  - Ex., the current task has to wait for data to be loaded from memory.
Hardware multithreading (2)

- **Fine-grained** - the processor switches between threads after each instruction, skipping threads that are stalled.

  - **Pros**: potential to avoid wasted machine time due to stalls.
  - **Cons**: a thread that’s ready to execute a long sequence of instructions may have to wait to execute every instruction.
Hardware multithreading (3)

- **Coarse-grained** - only switches threads that are stalled waiting for a time-consuming operation to complete.

  - **Pros**: switching threads doesn’t need to be nearly instantaneous.
  - **Cons**: the processor can be idled on shorter stalls, and thread switching will also cause delays.
Hardware multithreading (3)

- **Simultaneous multithreading (SMT)** - a variation on fine-grained multithreading.

- Allows multiple threads to make use of the multiple functional units.
Next class: 02/03/15
HW #1 Due: Thursday, 02/12/15 at start of class
HW #2 Due: Tuesday, 02/17/15 at start of class
Continue Reading: Ch2, Pacheco 2011 text: "An Introduction to Parallel Programming."