Chapter 2
Parallel Hardware and Parallel Software

The Von Neuman Architecture

- **Control unit**: responsible for deciding which instruction in a program should be executed. (the boss)
- **ALU** (Arithmetic and logic unit): responsible for executing the actual instructions. (the worker)
- **Memory**:
  - A collection of locations, each of which is capable of storing instructions or data.
  - Every location has an address, which is used to access the location.
A Process (a task)

- An instance of a computer program that is being executed.
- Components of a process:
  - Memory space.
  - Program
  - data
  - Security information.
  - State of the process
    - Values of registers
    - Program counter, stack pointer, …
    - Allocated resources

Multitasking (multiprogramming): Gives the illusion that multiple processes are running simultaneously.

- Each process takes turns running. (time slice)
- Switch context (process) when blocked or time is up

The Memory wall

- Memory system is often the bottleneck for many applications.
- Memory performance metrics:
  - latency
  - bandwidth.
**Memory Latency: An Example**

- 1 GHz processor (1 ns clock) with a multiply-add units capable of executing a multiply/add in each cycle
  - The peak processor rating is 2 GFLOPS (2 * 10⁹ floating point operation per second).
- We want to compute the dot product of two vectors \( x[0], \ldots, x[n-1] \) and \( y[0], \ldots, y[n-1] \).
  \[
  DP = \sum_{i=0}^{n-1} x[i] \times y[i]
  \]
- Need two operands every ns for peak processor performance
- If DRAM has a latency of 100 ns, then it can supply only one operand every 100 cycle. Hence, can do a multiply/add (2 FLOPS) every 200 ns. That is one FLOP every 100 ns.
- Actual performance = 10 MFLOPS

**Overcoming the memory wall**

- **Use caches**: take advantage of spatial and temporal locality

- **Use prefetching**: Can improve performance by pre-fetching. In this case, performance depends on memory bandwidth and not its latency. In the example above, if bandwidth is such that one operands can be fetched every 5ns, then can do a multiply/add every 10 ns. That is one FLOP every 5ns (200 MFLOP) – still not enough to match the 2GFLOP processor.

- **Use multithreading**
Review of cache operation

- **Issues to deal with:**
  - How do we know if a data item is in the cache?
  - If it is, how do we find it?
  - If it is not, what do we do?

- **It boils down to**
  - where do we put an item in the cache?
  - how do we identify the items in the cache?

- **Two solutions**
  - put item anywhere in cache (associative cache)
  - associate specific locations to specific items (direct mapped cache)

---

**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 place in one of \( n \) different locations in the cache.

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

<table>
<thead>
<tr>
<th>Memory Index</th>
<th>Cache Location</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fully Assoc</td>
</tr>
<tr>
<td>0</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>1</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>2</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>3</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>4</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>5</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>6</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>7</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>8</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>9</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>10</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>11</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>12</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>13</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>14</td>
<td>0, 1, 2, or 3</td>
</tr>
<tr>
<td>15</td>
<td>0, 1, 2, or 3</td>
</tr>
</tbody>
</table>

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

Effect of cache line size

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++)
  y[i] += A[i][i]*x[i];

/* Assign y = 0 */

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

Direct mapped cache with line size = 4 words

Performance of second pair of loops is bad if cache can hold only 2 lines
**Effect of data layout: An example**

Consider the following code fragment:

```c
for (j = 0; j < 1000; j++)
    column_sum[j] = 0.0;
for (i = 0; i < 1000; i++)
    column_sum[j] += b[i][j];
```

The code fragment sums columns of the matrix `b` into a vector `column_sum`.

- The vector `column_sum` is small and easily fits into the cache
- The matrix `b` is accessed in a column order
- With row major storage, strided access results in very poor performance.

**What is row major storage?**

```
\[ b_{1,1} \ b_{1,2} \cdots \ b_{1,10} \\
\ b_{2,1} \ b_{2,2} \cdots \ b_{2,10} \\
\vdots \ \ \ \ \ \vdots \\
\ b_{10,1} \ b_{10,2} \cdots \ b_{10,10} \\
\]
```

Stored as:

```
\[ b_{1,1} \ b_{1,2} \cdots b_{1,10} \ b_{2,1} \ b_{2,2} \cdots b_{2,10} \ b_{3,1} \ b_{3,2} \cdots b_{3,10} \cdots \\
\]
```

**Change the access pattern**

We can fix the above code as follows:

```c
for (j = 0; j < 1000; j++)
    column_sum[j] = 0.0;
for (i = 0; i < 1000; i++)
    for (j = 0; j < 1000; j++)
        column_sum[j] += b[i][j];
```

In this case, the matrix is traversed in a row-order and performance can be expected to be significantly better --- Why?

**Memory layouts and order of computation can have a significant impact on the spatial and temporal locality.**
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.

Multiple levels of caches in multi-core processors

- **L1** - smallest & fastest
- **L2**
- **L3** - largest & slowest
- Off-chip memory
Multithreading: originally designed as an alternate Approach for Hiding Memory Latency

We illustrate threads with the simple example of dot product of two vectors, \(x\) and \(y\):

\[
\begin{align*}
\text{dp} &= 0; \\
\text{for} \ (i = 0; \ i < n; \ i++) \\
\text{dp} &= \text{dp} + x[i] \times y[i]
\end{align*}
\]

Instead of calling the function \textit{partial\_product()}, create a new thread that will execute that function.

\[
\begin{align*}
\text{dp} &= 0; \\
\text{for} \ (k = 0; \ k < 4; \ k++) \\
\text{partial\_product} \ (k \times n/4, \ n/4); \\
\text{dp} &= \text{dp} + \text{pdp}[k]; \\
\text{void partial\_product} \ (a, \ b); \\
\{ \\
\text{pdp}[k] &= 0; \\
\text{for} \ (i = a \ ; \ i < a+b; \ i++) \\
\text{pdp}[k] &= \text{pdp}[k] + x[i] \times y[i]; \\
\text{return}
\}
\end{align*}
\]
A Thread is a light weight process

- Threads within the same process
- Share the memory address space
- Each has its own registers, program counter and stack pointer
- The OS schedules processes but a thread library function schedules threads within a process

Multithreading for Latency Hiding: Example

- In the example of slide 16, the first thread accesses a pair of vector elements and waits for them.
- In the meantime, the second thread can access two other vector elements in the next cycle, and so on.
- After \( L \) cycles, where \( L \) is the latency of the memory system, the first thread gets the requested data from memory and can perform \( B \) computations, where \( B \) is the number of words in a cache block size.
- When done, the data items for the next thread arrive, and so on. If there are \( L/B \) threads, then, we can perform a computation in every clock cycle.

Assumptions made:
- The memory system is capable of servicing multiple outstanding requests, and its bandwidth can keep up with the processing rate.
- No time penalty for switching threads.
Virtual memory

- 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 is a cache for secondary storage (swap space).

- Exploits the principle of spatial and temporal locality.

- It only keeps the active parts of running programs in main memory (unit of storage = a page).

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.

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Virtual Page Number</th>
<th>Byte Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 ... 13 12</td>
<td>11 10 ... 1 0</td>
<td>1 0</td>
</tr>
<tr>
<td>0 0 ... 1 1</td>
<td>0 0 ... 1 1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

Virtual Address Divided into Virtual Page Number and Byte Offset
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.
- 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 (ILP)

- Attempts to improve processor performance by having multiple processor components or functional units simultaneously executing instructions.
- Pipelining - functional units are arranged in stages.
- Multiple issue - multiple instructions can be simultaneously initiated.
Add the floating point numbers 9.87×10⁴ and 6.54×10³

<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>9.87×10⁴</td>
<td>6.54×10³</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Compare exponents</td>
<td>9.87×10⁴</td>
<td>6.54×10³</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Shift one operand</td>
<td>9.87×10⁴</td>
<td>0.654×10⁴</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Add</td>
<td>9.87×10⁴</td>
<td>0.654×10⁴</td>
<td>10.524×10⁴</td>
</tr>
<tr>
<td>5</td>
<td>Normalize result</td>
<td>9.87×10⁴</td>
<td>0.654×10⁴</td>
<td>1.0524×10⁵</td>
</tr>
<tr>
<td>6</td>
<td>Round result</td>
<td>9.87×10⁴</td>
<td>0.654×10⁴</td>
<td>1.05×10⁵</td>
</tr>
<tr>
<td>7</td>
<td>Store result</td>
<td>9.87×10⁴</td>
<td>0.654×10⁴</td>
<td>1.05×10⁵</td>
</tr>
</tbody>
</table>

Add the floating point numbers 9.87×10⁴ and 6.54×10³

Pipelining example

```c
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.
- This loop takes about 7000 nanoseconds.
- Divide the floating point adder into 7 separate pieces of hardware or functional units.
- Output of one functional unit is input to the next
- Loop takes 1006 nanoseconds.
Pipelining

<table>
<thead>
<tr>
<th>Time</th>
<th>Fetch</th>
<th>Compare</th>
<th>Shift</th>
<th>Add</th>
<th>Normalize</th>
<th>Round</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>999</td>
<td>999</td>
<td>998</td>
<td>997</td>
<td>996</td>
<td>995</td>
<td>994</td>
<td>993</td>
</tr>
<tr>
<td>1000</td>
<td>999</td>
<td>998</td>
<td>997</td>
<td>996</td>
<td>995</td>
<td>994</td>
<td>994</td>
</tr>
<tr>
<td>1001</td>
<td>999</td>
<td>998</td>
<td>997</td>
<td>996</td>
<td>995</td>
<td>994</td>
<td>995</td>
</tr>
<tr>
<td>1002</td>
<td>999</td>
<td>998</td>
<td>997</td>
<td>996</td>
<td>995</td>
<td>994</td>
<td>996</td>
</tr>
<tr>
<td>1003</td>
<td>999</td>
<td>998</td>
<td>997</td>
<td>996</td>
<td>995</td>
<td>994</td>
<td>997</td>
</tr>
<tr>
<td>1004</td>
<td>999</td>
<td>998</td>
<td>997</td>
<td>996</td>
<td>995</td>
<td>994</td>
<td>998</td>
</tr>
<tr>
<td>1005</td>
<td>999</td>
<td>998</td>
<td>997</td>
<td>996</td>
<td>995</td>
<td>994</td>
<td>999</td>
</tr>
</tbody>
</table>

Pipelined Addition.
Numbers in the table are subscripts of operands/results.

Multiple Issue

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

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

- **VLIW**: functional units are scheduled at compile time (static scheduling).
- **Superscalers**: functional units are scheduled at run-time (dynamic scheduling).
Speculation

- 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.

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

If the system speculates incorrectly, it must go back and recalculate \( w \).

Multi-threading

Provides a means to continue doing useful work when the currently executing task has stalled (ex. wait for long memory latency)

- **Software-based** multithreading (Posix Threads)
  - Hardware traps on a long-latency operation
  - Software then switches the running process to another (context switching)
  - Issues with the overhead

- **Hardware-based** multithreading
  - For user defined threads or compiler generated threads
  - Hardware support for context switching
  - Replicate registers (including PC and stack pointer)
- Example: IBM Power5 and Pentium-4 supports SMT
Hardware multi-threading

- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed

- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn’t hide short stalls (such as the stalls resulting from data hazards)

- SMT in multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available