HW #1 has been posted; due date is 09/09/14
spots still available in class
check that you are getting emails from class mailing list:
par-comp-f14@mailman.sdsu.edu
if not, send me an email.
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.
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
- Largest & slowest: L3
**Cache hit**

- fetch x
  - L1: x sum
  - L2: y z total
  - L3: A[] radius r1 center

**Cache miss**

- fetch x
  - L1: y sum
  - L2: r1 z total
  - L3: A[] radius center
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

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];

/* 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];
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></th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Page Number</td>
<td>Byte Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31 30 … 13 12</td>
<td>11 10 … 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 … 1 1</td>
<td>0 0 … 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.2: 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.
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)

```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 (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>
<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</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>0</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>0</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>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</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></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 parallel adders](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)

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

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.
Flynn’s Taxonomy

SISD

MISD

SIMD

MIMD

Source:

http://en.wikipedia.org/wiki/Flynn’s_taxonomy
Single Instruction Single Data

![Diagram of SIMD architecture]

- **Control unit**
- **Processor**
- **Data (in)**
- **Data (out)**
Parallelism achieved by dividing data among the processors.

- Applies the same instruction to multiple data items.
- Called data parallelism
SIMD example

for \( i = 0; i < n; i++ \)

\[ x[i] += y[i]; \]
What if we don’t have as many ALUs as data items?

Divide the work and process iteratively.

Ex. \( m = 4 \) ALUs and \( n = 14 \) data items.

<table>
<thead>
<tr>
<th>Round</th>
<th>ALU1</th>
<th>ALU2</th>
<th>ALU3</th>
<th>ALU4</th>
</tr>
</thead>
</table>
**SIMD drawbacks**

- All ALUs are required to execute the same instruction, or remain idle.
- In classic design, they must also operate synchronously.
- The ALUs have no instruction storage.
- Efficient for large data parallel problems, but not other types of more complex parallel problems.
Vector processors (1)

- Operate on arrays or vectors of data while conventional CPU’s operate on individual data elements or scalars.

- Vector registers.
  - Capable of storing a vector of operands and operating simultaneously on their contents.
Vector processors (2)

- Vectorized and pipelined functional units.
  - The same operation is applied to each element in the vector (or pairs of elements).

- Vector instructions.
  - Operate on vectors rather than scalars.
Vector processors (3)

- Interleaved memory.
  - Multiple “banks” of memory, which can be accessed more or less independently.
  - Distribute elements of a vector across multiple banks, so reduce or eliminate delay in loading/storing successive elements.
- Strided memory access and hardware scatter/gather.
  - The program accesses elements of a vector located at fixed intervals.
**Vector processors - Pros**

- Fast.
- Easy to use.
- Vectorizing compilers are good at identifying code to exploit.
  - Compilers also can provide information about code that cannot be vectorized.
    - Helps the programmer re-evaluate code.
- High memory bandwidth.
- Uses every item in a cache line.
Vector processors - Cons

- They don’t handle irregular data structures as well as other parallel architectures.

- A very finite limit to their ability to handle ever larger problems. (scalability)
The Cray-1 Vector Computer:

- First vector machine (1975)
- $8.86 million
- appx 140 MFlops, for weather calculation!!
- load a lot of data into memory, perform a lot of ops on that data
- Freon liquid cooling
- 12 functional units (address, scalar, vector, and floating point)
Graphics Processing Units (GPU)

- Real time graphics application programming interfaces or API’s use points, lines, and triangles to internally represent the surface of an object.
GPUs

- A graphics processing pipeline converts the internal representation into an array of pixels that can be sent to a computer screen.

- Several stages of this pipeline (called shader functions) are programmable.
  - Typically just a few lines of C code.
GPUs

- Shader functions are also implicitly parallel, since they can be applied to multiple elements in the graphics stream.

- GPU’s can often optimize performance by using SIMD parallelism.

- The current generation of GPU’s use SIMD parallelism.
  - Although they are not pure SIMD systems.
NVIDIA GPU GF100
High-Level Block Diagram (2010)

- CPU is called the host and the cores in the GPU are called devices
- 4 "GPC" clusters
- Many SM (stream multiprocessors) each with SPs
- 512 CUDA stream processors (SPs) or cores
- SIMT (single instr. multiple thread)

Source: http://hothardware.com/Articles/NVIDIA-GF100-Architecture-and-Feature-Preview
NVIDIA GPU

- each SM core in each GPC is comprised of 32 CUDA cores
- 48/16KB of shared memory (3 x that of GT200),
- 16/48KB of L1 (there is no L1 cache on GT200),

Source: http://hothardware.com/Articles/NVIDIA-GF100-Architecture-and-Feature-Preview
MIMD

- Supports multiple simultaneous instruction streams operating on multiple data streams.

- Typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU.
Single Instruction Multiple Data

Multiple Instruction Multiple Data
Shared Memory System (1)

- A collection of autonomous processors is connected to a memory system via an interconnection network.
- Each processor can access each memory location.
- The processors usually communicate implicitly by accessing shared data structures.
Shared Memory System (2)

- Most widely available shared memory systems use one or more multicore processors.
  - (multiple CPU’s or cores on a single chip)
Shared Memory System

Figure 2.3
**UMA multicore system**

Time to access all the memory locations will be the same for all the cores.

*Figure 2.5*
NUMA multicore system

A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

Figure 2.6
Distributed Memory System

- **Clusters** (most popular)
  - A collection of commodity systems.
  - Connected by a commodity interconnection network.

- **Nodes** of a cluster are individual computations units joined by a communication network.

  a.k.a. hybrid systems
Distributed Memory System

Figure 2.4
Mac Book Pro - Intel Core

- Intel Core i7, Z77 chipset
- 4 cores, 8 hyperthreads
- 32 + 32 KB L1 cache for data and instructions (per core)
- 256 KByte L2 cache (per core)
- 8 MB L3 cache (split up between cores and GPU)

Source: http://www.notebookcheck.net/Review-Intel-Ivy-Bridge-Quad-Core-Processors.73624.0.html
Next Time

- Next class: Thursday, 09/04/14
- Finish Parallel Hardware
- Look at Parallel Software..