Computer Architecture (CSE 422S)

Ken Wong
Washington University

kenw@wustl.edu
www.arl.wustl.edu/~kenw

Technology Trends

- CPU speed doubles every 18 months (Moore's Law)
- Memory speed doubles every 10 years
- But memory density quadruples every 2 years!
- Cache memories are an attempt to bridge CPU-memory gap

Potential parallelism between components
- Busses
  - Shared access
  - Connect different clock domains

Potential parallelism between components

Shared access

Connect different clock domains

Computer Organization

CPU
(Registers)

L1 Cache

L2 Cache

Processor Bus

Video

Bridge

Main Memory

I/O Bus

Controller

I/O Bus

Controller

Disk Drives

Network

Storage Hierarchy Properties

<table>
<thead>
<tr>
<th></th>
<th>Size*</th>
<th>Access Time</th>
<th>Cost/MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Integer Registers</td>
<td>256 B</td>
<td>1-4 ns</td>
<td>?</td>
</tr>
<tr>
<td>Primary (L1) Cache</td>
<td>36 KB</td>
<td>1-4 ns</td>
<td>?</td>
</tr>
<tr>
<td>Secondary (L2) Cache</td>
<td>1 MB</td>
<td>4-30 ns</td>
<td>?</td>
</tr>
<tr>
<td>Main Memory (DRAM)</td>
<td>512 MB</td>
<td>30-60 ns</td>
<td>$0.30</td>
</tr>
<tr>
<td>Disk Drive</td>
<td>160 GB</td>
<td>8-30 ms</td>
<td>$0.0008</td>
</tr>
</tbody>
</table>
Evolution of Intel Processor Features

<table>
<thead>
<tr>
<th>Processor</th>
<th>Date</th>
<th>Frequency</th>
<th>Transistors</th>
<th>Caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>60 MHz</td>
<td>3.1 M</td>
<td>L1: 16 KB</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1995</td>
<td>200 MHz</td>
<td>5.5 M</td>
<td>L1: 16 KB, L2: 256 KB</td>
</tr>
<tr>
<td>Pentium II</td>
<td>1997</td>
<td>266 MHz</td>
<td>7 M</td>
<td>L1: 32 KB, L2: 256 KB</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>500 MHz</td>
<td>8.2 M</td>
<td>L1: 32 KB, L2: 512 KB</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2000</td>
<td>1.5 GHz</td>
<td>42 M</td>
<td>*L1: 8 KB, L2: 256 KB</td>
</tr>
<tr>
<td>Xeon</td>
<td>2002</td>
<td>1.70 GHz</td>
<td>42 M</td>
<td>*L1: 8 KB, L2: 512 KB</td>
</tr>
<tr>
<td>Pentium M</td>
<td>2004</td>
<td>2.00 GHz</td>
<td>140 M</td>
<td>*L1: 64 KB, L2: 2 MB</td>
</tr>
</tbody>
</table>

* On-die caches

1 MHz = 10^6 cycles per sec, 1 GHz = 10^9

Clock Speeds

- CPU
  - 250 MHz SRAM
- Processor
  - 500 MHz Pentium
- Registers
  - 100 MHz, 32-Bit Processor Bus
- L1 Cache
- L2 Cache
- 4 nsec
- Video
- Bridge
- Main Memory
- 100 MHz SDRAM
- PCI Bus (32 or 64 bits), 33 MHz
- Controller
- Need for asynchrony between components
  - Buffers smooth out traffic
  - May still get "stalls"

Basic Instruction Execution

- Start
- Fetch Next Instruction
- Register Inst.
- Memory Inst.
- Execute
- Fetch Inst.
- Execute
- Fetch Data
- Execute
- Done

Processor Registers

What types of registers can be found in a simple integer RISC CPU?

- Program Counter (PC): Address of next instruction
- Instruction Register (IR): The most recently fetched instruction
- Status Register (SR): Results of comparisons, errors, etc. (sometimes called Processor Status Word (PSW))
- Stack Pointer (SP): Address of the top stack element
- General Registers (R[0]..R[31]): Operands

How are these registers used during program execution?
Sources Of CPU Speed

- **RISC (Reduced Instruction Set Computer)**
  » Simpler instructions allow shorter clock cycle times
- **Shorter Wires**
  » Signal gets to destination faster: Smaller cycle time
- **Instruction Pipelining**
  » Stages: Fetch, Decode, Get Source, Execute, Complete
- **Superscalar (Multiple instruction stream exec)**
  » Out-of-Order Instruction Execution/Completion
- **Branch Prediction**
  » Avoid instruction stall preceding comparison result by executing code based on predicted comparison

Pipelining

- **Time**
- **Fetch I**
- **Fetch D**
- **Execute**

<table>
<thead>
<tr>
<th>Time</th>
<th>Fetch I</th>
<th>Fetch D</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I2</td>
<td>I1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>I3</td>
<td>I2</td>
<td>I1</td>
</tr>
</tbody>
</table>

- Max speed of 1 MHz pipeline: 1 MIPS (1,000,000 Instructions Per Second)

Pipelining Gantt Chart

- **A Gantt Chart is a space-time diagram**

Example

- **Pseudo-Code:**
  ```
  int A[N][N], B[N];
  for (int i=0; i < N; i++)
    for (j=0; j < N; j++)
  ```

- **Machine Pseudo-Code:**
  ```
  // R1 contains address of A[i][j]
  L1: if (outer loop done) Exit outer loop;
  Initialize pointer R2;
  L2: if (inner loop done) Exit inner loop;
  R3 <- *R1;
  R4 <- *R2;
  R5 <- R3 * R4;
  *R1 <- R5;
  R1 = R1+4;
  R2 = R2+4;
  goto L2;
  // End of inner loop
  goto L1;
  // End of outer loop
  ```
  ```
  3N² arithmetic ops
  2N² memory reads
  N² memory writes
  ```
General Comments

- Arithmetic (and logic) operations are very simple
  - Requires bringing data from memory to the ALU (Arithmetic and Logic Unit)
  - Operations are actually done in general-purpose registers
- There are $3N^2$ arithmetic operations and at least $2N^2$ memory reads, $N^2$ memory writes
- Large N Execution Time (Naive): $3N^2 t_1 + 3N^2 t_2$
  - $t_1$: Time for 1 arithmetic operation (~1 nsec)
  - $t_2$: Time for 1 memory operation (~40 nsec; i.e., ~40 $t_1$)
  - **Bottleneck**: Main memory accesses
    - Execution time will be dominated by memory access pattern and memory speed!!!
    - **AND** CPU-Memory speed gap has continued to grow
- Need memory architecture improvement

Cache Memory Operation

- Cache Memory
  - Copy data to storage that is closer (and faster) to its place of use
  - Decreases access time to the cached data on subsequent accesses
- Read Operation
  ```
  if (X is in cache) { // Hit: T1
    Read from cache;
  } else { // Miss: T2 + T1
    Read from memory into cache;
    Read from cache;
  }
  ```
- Cache Design???

Cache Memory

- ALU – General Registers
- Inst, Data – L1 Cache
- L2 Cache – Main Memory

- L1 cache typically split between instructions and data
- Arithmetic operations are only done in general registers ➔ Must read $x$ from main memory into cache(s) and into a general register

EMAT

- **Effective Memory Access Time**
- $EMAT = \left[ \frac{H \times T_1 + M \times (T_1 + T_2)}{H + M} \right]$
  - $H$: Number of cache hits
  - $M$: Number of cache misses
  - $T_1$: Cache access time
  - $T_2$: Memory access time
  - This form is closer to the operational steps of caching
- $EMAT = h \times T_1 + (1-h) \times (T_1 + T_2)$
  - $h$: Hit ratio ($H / (H + M)$)
  - This form is easier to use
- $EMAT = T_1 + m \times T_2$ where $m = (1-h)$
  - $m$: Miss ratio
  - This form indicates that you must always pay the cost of a cache access
EMAT Example

Consider:
- Memory access time $T_2 = 50$ ns
- Cache access time $T_1 = 4$ ns

What is the EMAT if $h = 0.90$?
- $EMAT = 0.90 \times 4 + 0.10 \times 54 = 9.0$ ns

What is the EMAT if $h = 0.95$?
- $EMAT = 0.95 \times 4 + 0.05 \times 54 = 6.5$ ns

What size data cache would be required in our example to get an EMAT of 8 ns?
- Depends on organization of cache, memory layout, ...

Spatial and Temporal Locality

Consider the following program:

```
S = 0;
for (i=0; i < N; i++) { S = S + x[i]; }
T = 0;
for (i=0; i < N; i++) { T = T + x[i] * x[(i%4)*100]; }
```

Would cache memory speed up execution?
- The inner loops are small pieces of code that can be fetched repeatedly from cache memory (Spatial Locality) as it is executed.
- For small enough $N$, the data array $x[]$ can be stored entirely in cache and can be accessed from cache in the second loop (Spatial Locality)
- The inner loops are small pieces of code that can be fetched repeatedly from cache memory (Spatial Locality) as it is executed.

Simple Interrupts (1)

Interrupt (Trap, Exception)
- A vectored transfer of control to the supervisor
- Through a trap table
  - One entry for each trap type
  - PC: A branch table with 256 addresses stored in first 1KB of memory

Examples
- User requests OS service (system call via software trap)
- I/O device request completion
- Arithmetic overflow or underflow
- Page fault (virtual address not in main memory)
- Memory-protection violation (segmentation fault)
- Undefined instruction

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority

Simple Interrupts (2)

Disable all interrupts while processing an interrupt
- Ignore new interrupt until after reenabling interrupts
  - i.e., New interrupt is pending
- Interrupts usually have an interrupt priority
**Life Of A System Call**

- User Process
  - `read(fd, buf, nb)`
  - File Descriptor, Buffer Address, Number of Bytes
  - Software Initiated Trap

Top Half of Kernel (may block)

- I/O Wait Queue
- I/O Buffer
- Process State
- CPU State
- Hardware Interrupt

Bottom Half of Kernel (never blocks)

- Interrupt Handler
- I/O controller buffer

**Life Of A Device Read Request (1)**

- **CPU**
  - `read()` ultimately executes `syscall()` which traps into the kernel
    - Machine code for `read()` code comes from a library
  - `Enter kernel mode from user mode`
    - Kernel Mode: Privileged instructions and access to all of memory
  - `Queue request if I/O device is not ready`
  - `Send read request to controller of I/O device`
  - `Put process on queue while waiting for I/O to complete`
  - `Perform context switch`
    - Save process state and switch control (give CPU) to another process

- **Device Controller**
  - `Initiate operation on I/O device`

**Life Of A Device Read Request (2)**

- **I/O Device**
  - Transmit data to the controller’s buffer

- **Device Controller**
  - Request use of bus
  - Transfer data to main memory after bus grant
    - Controller “shares” memory bus usage with CPU and other devices
  - `Interrupt CPU` when read request has finished

- **CPU**
  - Interrupt transfers CPU control to the interrupt service routine
  - Save state of interrupted process
  - Disable lower priority interrupts while handling interrupt
  - Give control of CPU to a process selected by the scheduler
    - Restore CPU state and copy data from kernel buffer to user buffer
  - Reenable interrupts, restore user-mode and return from syscall

**Privileged Instructions**

- **Examples**
  - `Perform I/O`
  - `Change virtual memory protection bits`
  - `Disable/enable interrupt`
  - `Load timer registers`

- **Two (2) Modes of Operation**
  - **User Mode**: Execute instructions in user program
  - **Supervisor (Kernel) Mode**: Execute instructions in operating system
  - `Mode bit in status register` indicates the execution mode
  - `Causes of mode switching`
    - `Interrupt`, `System call` (software trap)

- **Privileged instructions can only be used in supervisor mode**
Multiple Interrupts

- Define interrupt priorities
  - Allow higher priority interrupt to interrupt a lower priority interrupt handler
  - e.g., Disk I/O interrupt > Serial line interrupt

---

Main Ideas

- Technology Trends (CPU-memory gap)
- Time and rate units (msec, usec, nsec, MHz, GHz)
- Parallel Execution ➔ Higher Speed
- Pipelining
  - Startup latency ➔ #stages / (clock rate)
  - Maximum throughput (output rate) ➔ clock rate
- Cache Hierarchy
  - Hide effect of slower memory
  - Increase effective memory speed
  - (Min, Max) access time = (cache, cache+memory) time
- Trap Processing
  - Control structure (branching, save/restore state)
  - Application to system call