**Administrivia**

- Homework #6 due today
- Final
  - Tuesday May 1, 2pm to 5pm D106 LSRC
  - Open book, open note (but...)
  - Cumulative (weighted toward last third of class)
- Today: Review
- Consider independent studies (CSURF)

**The Big Picture**

- The Five Classic Components of a Computer

**System Organization**

**Data Representations**

- Binary integers
  - Complexity of arithmetic operations
  - Negative numbers
  - Maximum number you can represent
- ASCII code for characters
- Binary, Oct, Hex
  - Converting between representations
Binary Integers

- Unsigned Binary numbers (only positive)
  - Base 2 numbers, only two digits {0, 1}
  - \( i = 100101_2 = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \)
- Sign-Magnitude
  - Highest order bit is the sign bit
  - Example: \( 010110_2 = 22_{10} \); \( 110110_2 = -22_{10} \)
- 2's Complement
  - \( i = \sum_{n=0}^{k} a_n \times 2^n \)
  - Example: \( 010110_2 = 22_{10} \); \( 101010_2 = -22_{10} \) (6-bit 2's comp.)
  - Examples: \( 010 = 000000_2; 110 = 000001_2; -110 = 111111_2 \)
- Arithmetic

Floating Point Representation

Numbers are represented by:

\[
X = (-1)^S \times 2^{E-127} \times 1.M
\]

- \( S := 1 \)-bit field; Sign bit
- \( E := 8 \)-bit field; Exponent: Biased integer, \( 0 \leq E \leq 255 \)
- \( M := 23 \)-bit field; Mantissa: Normalized fraction with hidden 1.

A Program's View of Memory

- What is Memory? a bunch of bits
- Looks like a large linear array
- Find things by indexing into array
  - unsigned integer
  - Most computers support byte (8-bit) addressing
    - Each byte has a unique address (location).
    - Byte of data at address 0x100 and 0x101
    - Word of data at address 0x100 and 0x104
  - 32-bit v.s. 64-bit addresses
    - we will assume 32-bit for rest of course, unless otherwise stated

A Simple Program's Memory Layout

```plaintext
... int result;
main()
{
  int *x;
  ...
  result = x + result;
  ...
}
mem[0x208] = mem[0x400] + mem[0x208]
```

Instruction Set Architecture

- Instructions are bits with well defined fields
  - Like a floating point number has different fields
- Instruction Format
  - establishes a mapping from "instruction" to binary values
  - which bit positions correspond to which parts of the instruction (operation, operands, etc.)

Basic ISA Classes

Accumulator:

\[
1 \text{ address add A acc} \leftarrow \text{acc} + \text{mem[A]}
\]

Stack:

\[
0 \text{ address add tos} \leftarrow \text{tos} + \text{next (JAVA VM)}
\]

General Purpose Register:

\[
2 \text{ address add A B A} \leftarrow A + B
\]

Load/Store:

\[
3 \text{ address add Ra Rc Ra} \leftarrow \text{mem[Rc]}
\]

<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>add</td>
</tr>
<tr>
<td>1</td>
<td>address</td>
</tr>
<tr>
<td>2</td>
<td>address</td>
</tr>
<tr>
<td>3</td>
<td>address</td>
</tr>
<tr>
<td>4</td>
<td>address</td>
</tr>
<tr>
<td>5</td>
<td>load</td>
</tr>
<tr>
<td>6</td>
<td>store</td>
</tr>
</tbody>
</table>
MIPS Instruction Formats

R-type: Register-Register

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>6 5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs</td>
<td>Rt</td>
<td>Rd</td>
<td>imm11</td>
<td>Func</td>
<td></td>
</tr>
</tbody>
</table>

I-type: Register-Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs</td>
<td>Rt</td>
<td>immediate</td>
<td></td>
</tr>
</tbody>
</table>

J-type: Jump / Call

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>target</td>
<td></td>
</tr>
</tbody>
</table>

Terminology

Op = opcode
Rs, Rt, Rd = register specifier

Instruction Sequencing

- Fetch Execute Cycle
- Program Counter
- Jumps vs. branch

Assembly Programming

- Identifiers
- Labels
- Pseudo-instructions
- Allocating & accessing memory
- From C/C++ to ASM
- What happens during the execution of a given ASM program

Procedure Call and Return

```c
int equal(int a1, int a2) {
    int tsame;
    tsame = 0;
    if (a1 == a2)
        tsame = 1;
    return(tsame);
}
main() {
    int x, y, same;
    x = 43;
    y = 2;
    same = equal(x, y);
    // other computation
}
```

Procedure Calls

- Procedure Call Gap
  - jal, jr -> function invocation, arguments, return value
  - recursion (local name space)
- Stack is good data structure for this
- Save/restore values
- Calling conventions
  - callee, caller saved regs

Basics of Logic Design

- Boolean functions
- Logic gates (AND, OR, NOT, etc)
- Multiplexors
- Decoders
- Adder
- Arithmetic Logic Unit (ALU)
Boolean Functions, Expressions, Truth Table

\[ F(A, B, C) = (A \cdot B) + (\neg A \cdot C) \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Can write any Boolean function as Sum of Products

Logic Gates

- **Gates** are electronics devices that implement simple Boolean functions

Examples

\[ \begin{align*}
\text{AND}(a, b) & : a \cdot b \\
\text{OR}(a, b) & : a + b \\
\text{XOR}(a, b) & : a \oplus b \\
\text{NAND}(a, b) & : \neg(a \cdot b) \\
\text{NOR}(a, b) & : \neg(a + b) \\
\text{XNOR}(a, b) & : \neg(a \oplus b)
\end{align*} \]

Boolean Functions, Gates and Circuits

- **Circuits** are made from a network of gates. (function compositions).

\[ a \oplus b = \neg a \cdot b + \neg b \cdot a \]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>(a \oplus b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The ALU

- **Overflow** = Zero

Memory Elements

- **Store State**
- **Output depends on Input and current state**
- **Used to create sequential circuits**
- **Previous circuits called combinational**

- **SR-Latch, D Flip-Flop**
- **Register File**
  - D Flip-Flop + Address Decoder

Arithmetic

- **Multiplication**
- **Booth’s algorithm**
- **FP Addition**
- **FP multiplication**
- **Accuracy (rounding errors)**
Data Path

• What parts of the data path are used to implement a specific instruction

An Abstract View of the Implementation

Datapath for Store Operations

• Mem[R[rs]] + SignExt[imm16] <- R[rt]
  Example:
  sw rt, rs, imm16

Instruction Fetch Unit

• j target
  - PC<31:2> <- PC<31:2> concat target<25:0>

The Single Cycle Datapath during Or Immediate

• R[rt] <- R[rs] or ZeroExt[imm16]

Finite State Machine

• Finite State Machine is:
  - A machine with a finite number of possible states.
  - A machine with a finite number of possible inputs.
  - A machine with a finite number of possible different outputs.
  - At each period (Clock cycle) the machine receives an input and it produces an output.
  - The output is a function of the machine input and current state.
  - After each period the machine changes state.
  - The new state is a function of the input and current state.
Traffic Light Controller: Coded State Diagram

Traffic Controller FSM implementation

Multicycle Processor Control

System Organization

Caches

Example: 1K Direct Mapped Cache

- Capacity
- Associativity
- Block size
- Multiple levels of cache
  - CPU time = (CPU execution clock cycles + Memory stall clock cycles) x clock cycle time
  - Memory stall clock cycles = Memory accesses x Miss rate x Miss penalty
  - Software techniques to improve cache performance
Exceptions & Interrupts

- Execution Context
- Context Switch
- Transfer control to OS
- Execution Mode (kernel vs. user)

Concurrency

- Multiple things happening simultaneously
  - logically or physically
- Causes
  - Interrupts
  - Voluntary context switch (system call/trap)
  - Hyperthreading / Shared memory multiprocessor

Solution: Atomic Sequence of Instructions

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>Time</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Atomic Sequence
  - Appears to execute to completion without any intervening operations

HW Support for Atomic Operations

- Could provide direct support in HW
  - Atomic increment
  - Insert node into sorted list??
- Just provide low level primitives to construct atomic sequences
  - called synchronization primitives
    - LOCK(counter->lock);
    - counter->value = counter->value + 1;
    - UNLOCK(counter->lock);
  - test&set (x) instruction: returns previous value of x and sets x to “1”
    - LOCK(x) → while (test&set(x));
    - UNLOCK(x) → x = 0;

Virtual Memory

- Process = virtual address space + thread of control
- Translation
  - VA → PA
    - What physical address does virtual address A map to
  - Is VA in physical memory?
- Protection (access control)
  - Do you have permission to access it?

Virtual and Physical Memories

- Disk
- Frame 0
- Frame 1
- Frame 2
- Frame 3
- Frame 4
- Frame 5
- Page 0
- Page 1
- Page 2
- Page 3
- Page N-1
- Page N
Fast Translation: Translation Buffer

- Cache of translated addresses
- 64 entry fully associative

Virtual Address

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Page Offset</th>
<th>phys frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Physical Address

- 64x1 mux

I/O: Device Drivers

- top-half
  - API (open, close, read, write, ioctl)
  - I/O Control (ioctl, device specific arguments)
- bottom-half
  - interrupt handler
  - communicates with device
  - resumes process
- Must have access to user address space and device control registers => runs in kernel mode.

Device Controllers

Controller deals with mundane control (e.g., position head, error detection/correction)

Processor communicates with Controller

Memory Mapped I/O

- Issue command through store instruction
- Check status with load instruction
- Caches?

Communicating with the processor

- Polling
  - can waste time waiting for slow I/O device
  - busy wait
  - can interleave with useful work
- Interrupts
  - interrupt overhead
  - interrupt could happen anytime - asynchronous
  - no busy wait

Data Movement

- Programmed I/O
  - processor has to touch all the data
  - too much processor overhead
  - for high bandwidth devices (disk, network)
- DMA
  - processor sets up transfer(s) (source/destination addresses, length)
  - DMA controller transfers data
  - complicates memory system
**Disk Access**

- **Access time** = queue + seek + rotational + transfer + overhead
- **Seek time**
  > move arm over track
  > average is confusing (startup, slowdown, locality of accesses)
- **Rotational latency**
  > wait for sector to rotate under head
  > average = 0.5/(3600 RPM) = 8.3ms
- **Transfer Time**
  > \( f(\text{size}, \text{BW bytes/sec}) \)

---

**Multiple Potential Bus Masters: the Need for Arbitration**

- **Bus arbitration scheme:**
  - A bus master wanting to use the bus asserts the bus request
  - A bus master cannot use the bus until its request is granted
  - A bus master must signal to the arbiter after finish using the bus
- **Bus arbitration schemes usually try to balance two factors:**
  - **Bus priority:** the highest priority device should be serviced first
  - **Fairness:** Even the lowest priority device should never be completely locked out from the bus
- **Bus arbitration schemes can be divided into four broad classes:**
  - Distributed arbitration by self-selection: each device wanting the bus places a code indicating its identity on the bus.
  - Distributed arbitration by collision detection: Ethernet uses this.
  - Daisy chain arbitration: single device with all request lines.
  - Centralized, parallel arbitration: see next-next slide

---

**Designing an I/O System**

- **CPU** \( 3 \times 10^9 \) inst/sec, average 100,000 insts in OS per I/O
- **Memory bus** 1000 MB/sec
- **SCSI Ultra320 controller** 320MB/sec, up to 7 disks
- **Disk R/W BW** 75 MB/sec, average seek + rotational 6ms
- **Workload:**
  - 64KB reads (sequential)
  - 200,000 user insts per I/O
- **Find:**
  - max sustainable I/O rate
  - # of disks & controllers required

---

**Summary**

- You’ve learned a lot this semester!
- Good luck on your finals
- Study...