Review

Computer Science 104
Machine Organization & Programming

Administrative

- Y86 Simulator Due today
- Final Friday May 6, 7-10pm in D106
- Today
  - high-level review
  - Q&A
- Marisabel review Wednesday (?) Watch for email
What is Computer Architecture?

- Coordination of levels of abstraction

  - Application
  - Operating System
  - Compiler
  - Firmware
  - CPU
  - Memory
  - I/O system
  - Digital Design
  - Circuit Design

  Software
  Interface Between HW and SW
  Instruction Set Architecture, Memory, I/O
  Hardware

- Under a set of rapidly changing Forces
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\}
  - \( i = 100101_2 \); \( i = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \)
- Sign-Magnitude := Highest order bit is the sign bit
  - Example: \( 010110_2 = 22_{10} \); \( 110110_2 = -22_{10} \)
- 2’s Complement
  - \( i = -a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \ldots + a_0 \times 2^0 \)
  - Example: \( 010110_2 = 22_{10} \); \( 101010_2 = -22_{10} \) (6-bit 2’s comp.)
  - Examples: \( 0_{10} = 000000_2 \); \( 1_{10} = 000001_2 \); \( -1_{10} = 111111_2 \)
- Arithmetic
Floating Point Representation

Numbers are represented by:

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

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

Single precision floating point number

```
  31 30 22 0
  s  exp  Mantissa
```

---

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

```c
... 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 address  add A  acc ← acc + mem[A]
1+x address addx A  acc ← acc + mem[A + x]

Stack:

0 address  add  tos ← tos + next (JAVA VM)

General Purpose Register:

2 address  add A B  A ← A + B (x86)
3 address  add A B C  A ← B + C

Load/Store:

3 address  add Ra Rb Rc  Ra ← Rb + Rc (ARM,MIPS)
load Ra Rb  Ra ← mem[Rb]
store Ra Rb  mem[Rb] ← Ra

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

---

Review: 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
}
```

- MOV $43, %ECX
- MOV $2, %EDX
- CALL 0x30408
- MOV $0, %eax
- CMPL %ecx, %edx
- JNE 0x30418
- MOV $1, %eax
- ADDL %edx, %ecx
- RET

---

PC | M[%esp]
---|---
0x10000 | ??
0x10004 | ??
0x10008 | ??
0x30408 | 0x1000c
0x3040c | 0x1000c
0x30410 | 0x1000c
0x30414 | 0x1000c
0x30418 | 0x1000c
0x3041c | 0x1000c
0x1000c | ??
Procedure Calls

- Procedure Call Gap
  - call, ret \rightarrow \text{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

- \( AND(a, b) \)
- \( OR(a, b) \)
- \( NOT(a) \)
- \( XOR(a, b) \)
- \( NAND(a, b) \)
- \( NOR(a, b) \)
- \( XNOR(a, b) \)
Boolean Functions, Gates and Circuits

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

\[
F = \overline{a} \overline{b} + \overline{b} \overline{a}
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>XOR(a, 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

<table>
<thead>
<tr>
<th>a_{n-1}</th>
<th>a_{n-2}</th>
<th>b_{n-1}</th>
<th>b_{n-2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_{n-1}</td>
<td>a_{n-2}</td>
<td>b_{n-1}</td>
<td>a_{n-2}</td>
</tr>
<tr>
<td>b_{n-1}</td>
<td>a_{n-2}</td>
<td>b_{n-1}</td>
<td>a_{n-2}</td>
</tr>
<tr>
<td>b_{n-1}</td>
<td>a_{n-2}</td>
<td>b_{n-1}</td>
<td>a_{n-2}</td>
</tr>
</tbody>
</table>

© Alvin R. Lebeck
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

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.
Data Path

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

Y86 Instructions

• Recall Memory is a Bunch of Bits!
• How do we know if it is an instruction or not?
• How do we know which instruction, which operands, etc.
• Format
  ➢ 1--6 bytes of information read from memory
    » Can determine instruction length from first byte
    » Not as many instruction types, and simpler encoding than with IA32
  ➢ Each accesses and modifies some part(s) of the program state
## Y86 Instruction Set

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>halt</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rrmovl rA, rB</td>
<td>2</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>irmovl V, rB</td>
<td>3</td>
<td>0</td>
<td>rB</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rrmovl rA, D(rB)</td>
<td>4</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>rrmovl D(rB), rA</td>
<td>5</td>
<td>0</td>
<td>rA</td>
<td>D</td>
<td>rB</td>
<td></td>
</tr>
<tr>
<td>OPl rA, rB</td>
<td>6</td>
<td>0</td>
<td>rA</td>
<td>rB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td>Dest</td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
<td>Dest</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pushl rA</td>
<td>10</td>
<td>0</td>
<td>rA</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>popl rA</td>
<td>11</td>
<td>0</td>
<td>rA</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## SEQ Stages

- **Fetch**
  - Read instruction from instruction memory
  - If PC points to it, we view it as instruction
- **Decode**
  - Read program registers
- **Execute**
  - Compute value or address
- **Memory**
  - Read or write data
- **Write Back**
  - Write program registers
- **PC**
  - Update program counter
Computation Steps

- All instructions follow same general pattern
- Differ in what gets computed on each step

### Computation Steps Table
- **Fetch**
  - icode,ifun \( \rightarrow \) Read instruction byte
  - rA,rB \( \rightarrow \) Read register byte
  - valC \( \rightarrow \) [Read constant word]
  - valP \( \rightarrow \) Compute next PC

- **Decode**
  - valA, srcA \( \rightarrow \) Read operand A
  - valB, srcB \( \rightarrow \) Read operand B

- **Execute**
  - valE \( \leftarrow \) Perform ALU operation
  - Cond code \( \leftarrow \) Set condition code register

- **Memory**
  - valM \( \leftarrow \) [Memory read/write]

- **Write**
  - dstE \( \leftarrow \) Write back ALU result
  - dstM \( \leftarrow \) Write back memory result

- **PC update**
  - PC \( \leftarrow \) Update PC

---

Pipeline Stages

- ** Fetch**
  - Select current PC
  - Read instruction
  - Compute incremented PC

- **Decode**
  - Read program registers

- **Execute**
  - Operate ALU

- **Memory**
  - Read or write data memory

- **Write Back**
  - Update register file
Pipelining the mrmovl Instruction

- The five independent functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File’s Read ports for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the Mem stage
  - Register File’s Write port for the WrB stage
- One instruction enters the pipeline every cycle
  - One instruction comes out of the pipeline (completed) every cycle
  - The “Effective” Cycles per Instruction (CPI) is 1; ~1/5 cycle time

Pipeline Summary

- Concept
  - Break instruction execution into 5 stages
  - Run instructions through in pipelined mode
  - Latency of one instruction is ~ same, but throughput increases
- Limitations
  - Can’t handle dependencies between instructions when instructions follow too closely
  - Data dependencies
    - One instruction writes register, later one reads it
  - Control dependency
    - Instruction sets PC in way that pipeline did not predict correctly
    - Mispredicted branch and return
Caches

- Capacity
- Associativity
- Block size
- Multiple levels of cache
  - \( \text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory stall clock cycles}) \times \text{clock cycle time} \)
  - \( \text{Memory stall clock cycles} = \text{Memory accesses} \times \text{Miss rate} \times \text{Miss penalty} \)
  - 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)
  - Multithreading / Shared memory multiprocessor

![Concurrency Diagram]

**Solution: Atomic Sequence of Instructions**

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

```plaintext
begin atomic
ld (count)
add
switch
st (count+1)
end atomic
switch
```

```
begin atomic
ld (count+1)
wait
st (count+2)
end atomic
```

© Alvin R. Lebeck
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

Virtual Memory

Page 0
Page 1
Page 2
Page 3

Physical Memory

Frame 0
Frame 1
Frame 2
Frame 3
Frame 4
Frame 5

Disk

Page 2

Virtual and Physical Memories

Page Tables

- A page table is an array of page table entries (PTEs) that maps virtual pages to physical pages.
  - Per-process kernel data structure in DRAM
Speeding up Translation with a TLB

- Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay
- Solution: **Translation Lookaside Buffer** (TLB)
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

TLB Miss

A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?
Intel Core i7 Memory System

Processor package
Core x4
- Registers
- Instruction fetch
- MMU (addr translation)

- L1 d-cache
  - 32 KB, 8-way
- L1 i-cache
  - 32 KB, 8-way
- L1 d-TLB
  - 64 entries, 4-way
- L1 i-TLB
  - 128 entries, 4-way

- L2 unified cache
  - 256 KB, 8-way
- L2 unified TLB
  - 512 entries, 4-way

- L3 unified cache
  - 8 MB, 16-way
  - (shared by all cores)
- DDR3 Memory controller
  - 3 x 64 bit @ 10.66 GB/s
  - 32 GB/s total (shared by all cores)

QuickPath interconnect
- 4 links @ 25.6 GB/s each

To other cores
To I/O bridge

Main memory

Device Drivers

- Software that provides access to devices (mp3, camera, phone, FLASH drive)
  - top-half
    - Application Programming Interface (API): e.g., open, close, read, write, ioctl
    - I/O Control: e.g., 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?**

- Physical Address
  - ROM
  - RAM
  - I/O
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 Time

- Average time to access some target sector approximated by:
  - $T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}}$

- Seek time ($T_{\text{avg seek}}$)
  - Time to position heads over cylinder containing target sector.
  - Typical $T_{\text{avg seek}}$ is 3—9 ms

- Rotational latency ($T_{\text{avg rotation}}$)
  - Time waiting for first bit of target sector to pass under r/w head.
  - $T_{\text{avg rotation}} = 1/2 \times 1/\text{RPMs} \times 60 \text{ sec}/1 \text{ min}$
  - Typical $T_{\text{avg rotation}} = 7200 \text{ RPMs}$

- Transfer time ($T_{\text{avg transfer}}$)
  - Time to read the bits in the target sector.
  - $T_{\text{avg transfer}} = 1/\text{RPM} \times 1/(\text{avg # sectors/track}) \times 60 \text{ secs}/1 \text{ min}$

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
Summary

• You’ve learned a lot this semester!
• Good luck on your finals
• Study…