A Quick Start Guide to CS/COE 0447
Informal discussion about basic concepts

We'll refine and more formally define many of this material over the next few weeks!!!
Binary Numbers

- Computer operates on a “native” fixed-size binary quantity
  - 4-bit computer: Native size is a binary number with 4 bits (nibble)
  - 8-bit computer: Native size is a binary number with 8 bits (byte)
  - 16-bit computer: Native size is a binary number with 16 bits (halfword)
  - 32-bit computer: Native size is a binary number with 32 digits (word)
  - 64-bit computer: Native size is a binary number with 64 digits
  - 36-bit computer: Native size is a binary number with 36 digits

Binary Numbers

Really just a different representation for a quantity

How many ways can we represent these animals?
“Ten happy cows”
“Dix vaches heureux ”
“10 cows”
10 (decimal quantity)
1010b (base-2 quantity for 10 decimal)
110001b 110000b (base-2 quantity for characters ‘1’, ‘0’)
Ah (base-16 quantity for 10 decimal)
12o (base-8 quantity for 10 decimal)
Other Representations

- Base-10 numbers (decimal)
  - Each digit represents one of ten values
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

- Base-16 numbers (hexadecimal)
  - Each digit represents one of sixteen values
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

- Base-8 numbers (octal)
  - Each digit represents one of eight values
  - 0, 1, 2, 3, 4, 5, 6, 7

- Do you get the idea??? 😊

A bag o’ pennies!

Consider a bag of 100 pennies.

How can we represent it?
  - $1, 0.63 GBP, 0.76 Euro, 1 Susan B Anthony coin

OK, let’s try our bases…
  - Base-10: 100
  - Base-2: 1100100b
  - Base-16: 64h
  - Base-8: 144o
Why all this fuss?

- **Binary numbers**
  - Digital computers use "on/off" switches (transistors)
  - *On* is the binary digit 1
  - *Off* is the binary digit 0

- Have you written down a 32 digit binary number?
  1100010111101000110001001011b

- What a pain! Is there a better way?

  - **Base-16 (hexadecimal) is convenient notation**
    - Compact form
    - Easy and fast conversion binary <-> hexadecimal
    - Fixed sizes tend to be "chunkable" into 4s (consider slide 3)

Hexadecimal numbers

**Compare:** 1100010111101000110001001011b  
**To:** C2F51893h

What about a 64-bit number? Which would you rather write????

**How can we convert from binary to/from hex?**

- Simple, quick approach: A table approach (*we’ll be more formal later*)
- Construct a table that maps binary numbers to hex numbers

Hex digit is one of 16 numbers
Binary digit is one of 2 numbers

We need one hex digit for every four binary digits (4 digits, 2 per digit = $2^4 = 16$)
How to count in binary?

Suppose, we add two 1-bit binary numbers:

\[
\begin{align*}
0 + 0 &= 0 \\
0 + 1 &= 1 \\
1 + 0 &= 1 
\end{align*}
\]

But what about \(1 + 1\)?

It’s just like decimal numbers. We must carry the one:

\[
\begin{array}{cccc}
1 & 10 & 11 & 100 \\
+ 1 & + 1 & + 1 & + 1 \\
--- & --- & --- & --- \\
10 & 11 & 100 & 101 \\
\end{array}
\]

We have to carry the one since a single bit is only 0 or 1

---

Binary - Hex Table

Count from 0 to 15 in binary, then write hex digit

<table>
<thead>
<tr>
<th>binary</th>
<th>hex</th>
<th>binary</th>
<th>hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

remember how to add in binary, e.g.:

\[
\begin{array}{cccc}
0001 & 0101 & 1011 \\
+ 0001 & + 0001 & + 0001 \\
----- & ----- & ----- \\
0010 & 0110 & 1100 \\
\end{array}
\]
Use table to convert

For every four binary digits, look up the hex symbol
Write the hex symbol in corresponding position for four binary digits e.g., 0000111001111010000100011000100b

Grouped into “chunks” with four binary digits per chunk:

<table>
<thead>
<tr>
<th>0000</th>
<th>1110</th>
<th>0011</th>
<th>1101</th>
<th>0000</th>
<th>1000</th>
<th>1100</th>
<th>0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>E</td>
<td>3</td>
<td>D</td>
<td>0</td>
<td>8</td>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>

To go from hex to binary, simply do this process in reverse.

Converting Decimal to/from Binary

• How can we convert between decimal and binary?

• Binary numbers represent “sums of powers of 2”
• We can exploit this property to easily convert

• From decimal to binary
  – You are given the binary number size (e.g., 8 bits, 16 bits, 32 bits)
  – Write decimal number as a sum of powers of 2
  – Each exponent value determines a “position” in the binary number
  – Positions numbered 0 to number size – 1
  – Positions ordered right to left (0 is rightmost, number size – 1 is left)
    • 8-bit number: \[ b_7 \ b_6 \ b_5 \ b_4 \ b_3 \ b_2 \ b_1 \ b_0 \]
  – Write “1” in exponent position for each power of 2 in sum
  – Write “0” in all other positions
Example: Decimal to Binary

- Suppose 8 bit number
- Consider 63 decimal

- What are the powers of 2 for 63?
- How many “positions” are in the binary number?
- Which positions have 1s and which ones have 0s?
- Write down the 1s and 0s for the number?

- Answers:
  - $63 = 32 + 16 + 8 + 4 + 2 + 1 = 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$
  - 8 positions b/c it’s an 8-bit binary number
  - 1s in positions 0, 1, 2, 3, 4, 5 and 0s in positions 6, 7
  - 00111111

From Binary to Decimal

- There’s a “1” in any position with a power of 2 value
- Thus, we can sum the powers of 2s that correspond to the “1s positions” to get the decimal number!

- Consider the 8-bit number: 00101101b

- What positions (powers of 2) have 1s?
- What is the sum of the powers of 2?

- Answer:
  - Positions 0, 2, 3, 5
  - Sum is: $2^0 + 2^2 + 2^3 + 2^5 = 45$ decimal
System layers

- Computer system consists of several “layers”
- It’s stack of components that work together

Applications written in a high-level programming language (e.g., Java, C, C++)
System layers

- Computer system consists of several “layers”
- It’s stack of components that work together

<table>
<thead>
<tr>
<th>App 1</th>
<th>App 2</th>
<th>App 3</th>
<th>App 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Software (Compiler, Assembler, Linker)</td>
<td>Operating System</td>
<td>Instruction Set Architecture (ISA)</td>
<td>Processor Implementation</td>
</tr>
</tbody>
</table>

- Converts high-level language programs into a form that can run on the computer system
- Provides common functionality needed when program executes and has a way to share computer resources among programs
System layers

- Computer system consists of several “layers”
- It’s stack of components that work together

<table>
<thead>
<tr>
<th>App 1</th>
<th>App 2</th>
<th>App 3</th>
<th>App 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>System Software (Compiler, Assembler, Linker)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Operating System</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Instruction Set Architecture (ISA)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Processor Implementation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Gives an interface (a “contract”) between the software & the hardware; Hardware needs to know what to do. (0447!) Different ISAs are possible!

System layers

- Computer system consists of several “layers”
- It’s stack of components that work together

<table>
<thead>
<tr>
<th>App 1</th>
<th>App 2</th>
<th>App 3</th>
<th>App 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>System Software (Compiler, Assembler, Linker)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Operating System</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Instruction Set Architecture (ISA)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Processor Implementation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A specific hardware circuit (set of transistors & wires) that implements the ISA. You can implement it different ways!
From a C (C++ or Java) program to running it

- Software tools: Compiler, Assembler, Linker
  - Convert the HLL program into machine code
  - Processor (computer system) fetches & executes machine code
  - It doesn't "know" C, C++ or Java!

- Once program converted into machine code:
  - Program is stored in memory
  - Processor fetches one instruction at a time (the conveyor belt)
  - Processor executes the instruction until done (the control tower)
  - Continues executing instructions until program is done

- This is the "fetch-execute cycle". Fetch an instruction, execute it.
Assembly language and machine code

- Instruction set architecture
  - Describes the "machine instructions" that the processor can execute
  - Instructions say what computation to do (e.g., add two numbers)
  - Gives other details, such as temporary storage locations
  - Registers are "temporary storage" that can be used in computations
  - Many different instruction set architectures (Intel x86, Sun SPARC, etc)

- Assembly language
  - Simply a human readable form to write instructions
  - It's a programming language, much like Java, C++ or C
  - Except it's at the lowest level of the software stack
  - Many different assembly languages, usually one for each ISA

- Machine instructions (a.k.a., “machine code”)
  - Simply a machine readable form of instructions
  - A machine instruction is simply a binary number!

A little example

Let's add three numbers: 10, 20, 30
Here's a C program:
```c
int main(void) {
    int sum;
    sum = 10 + 20 + 30; // sum is 10+20+30=60
}
```
Here's a MIPS program (assembly language):
```assembly
addiu $5,$0,10 ; $5 is register, $5=0+10=10
addiu $5,$5,20 ; adds 20 to $5, $5=10+20=30
addiu $5,$5,30 ; adds 30 to $5, $5=30+30=60
```

The addition is done as three separate computations (0+10, 10+20, 30+30)
Register $5 is needed as temporary storage to hold sum between computations
A little example

Here’s a MIPS program (assembly language):

```
addiu $5,$0,10 ; $5 is register, $5=0+10=10
addiu $5,$5,20 ; adds 20 to $5, $5=10+20=30
addiu $5,$5,30 ; adds 30 to $5, $5=30+30=60
```

How about the machine code?
Assembler converts the above form into binary numbers

```
0x2405000a  binary for addiu $5,$0,10
0x24a50014  binary for addiu $5,$5,20
0x24a5001e  binary for addiu $5,$5,30
```

Note: 0x notation means the number that follows is hexadecimal

---

A Loop on Two Architectures

How much does the assembly language differ between ISAs?

```
int main(void)
{
    int i, sum;
    sum = 0;
    for (i = 0; i < 10; i++) // sum numbers 0 to 9
        sum = sum + i;
}
```

The assembly language and machine instructions for this program were created for two different instruction sets:

Sun SPARC processor
MIPS processor
A Loop on Two Architectures

**C program**

```c
int main(void) {
    int i, sum;
    sum = 0;
    for (i=0; i<10; i++)
        sum = sum + i;
}
```

Let's see this loop on the SPARC and MIPS processors.

Careful look at the assembly and the machine code!

A Loop on Two Architectures

**C program**

```c
int main(void) {
    int i, sum;
    sum = 0;
    for (i=0; i<10; i++)
        sum = sum + i;
}
```

**SPARC Assembly**

```
mov 0,%01
mov 0,%g1
L2:  add %01,%g1,%01
     add %g1,1,%g1
     cmp %g1,10
     blt  L2
```

Compiler translates the C program into a sequence of assembly language instructions for the Sun SPARC processor.
The same C program can be translated by a compiler for the MIPS processor as well.

Do you see differences?

The assembler for the SPARC converts the human readable assembly into actual machine code that the processor is able to execute (binary 32-bit numbers).
A Loop on Two Architectures

**C program**

```c
int main(void) {
    int i, sum;
    sum = 0;
    for (i=0;i<10; i++)
        sum = sum+i;
}
```

**SPARC Assembly**

```
mov 0,%o1
mov 0,%g1
L2: add %o1,%g1,%o1
    addi $3,%g1,1
    slti $2,%g1,10
    bne $2,%o1,L2
```

**MIPS Assembly**

```
move $5,$0
move $3,$0
L2: add $5,%g1,3
    addi $3,$3,1
    slti $2,$3,10
    bne $2,%0,L2
```

Similarly, the assembler for MIPS turns the assembly into machine code.

MIPS also has 32-bit instr.

Do you see differences?

---

A Loop on Two Architectures

What is the correct value for `sum`?

- **45 in decimal**
- **101101 in binary**
- **2D in hexadecimal**
- **55 in octal (base-8 ☺️)**
Try Running the Assembly Program with MARS

The assembly file is on the CS 0447 web site:

Steps to run the program
1. Save this file to a local directory on your computer (say “foo”)
2. Download the MARS simulator (see CS0447 web site).
3. Double click on the simulator icon (the jar file)
4. Select “File” -> “Open”, then navigate to directory “foo”, select the file “loop-mars.asm”.
5. Select “Run” -> “Assemble”. This assembles your program.
   - You are now in the Execute window, which shows the machine code.
6. Select “Run” -> “Go”. This runs your program. It stops with:
   -- program is finished running –
7. Look in the Registers window. The value of $a1 should be 45. This is the total sum of the numbers 0 to 10.