CS 221B Computer Architecture

Week 2: Information Representation

Fall 2001

Course Schedule

W1	Sep 11- Sep 14	Introduction	
W2	Sep 18- Sep 21	Information Representation (1)	(Chapter 3)
W3	Sep 25- Sep 28	Information Representation (2)	(Chapter 3)
W4	Oct 2- Oct 5	Computer Architecture Pep/6	(Chapter 4)
W5	Oct 9- Oct 12	Assembly Language	(Chapter 5)
W6*	**Oct 16- Oct 19**	**High-Level Languages (1)**	**(Chapter 6)**
W7*	**Oct 23- Oct 26**	**High-Level Languages (2)**	**(Chapter 6)**
	Midterm*		
W8	Oct 30- Nov 2	Instruction Sets	(Tanenbaum)
W9	Nov 6 - Nov 9	Devices & Interrupts	(Chapter 8)
W10*	**Nov 13- Nov 16**	**Storage Management**	**(Chapter 9)**
W11	Nov 20- Nov 23	Combinational Logic	(Chapter 10)
W12	Nov 27- Nov 30	Sequential Logic & Review	(Chapter 11)

**Marked classes will be taught by Dave Dove, same place same slot*

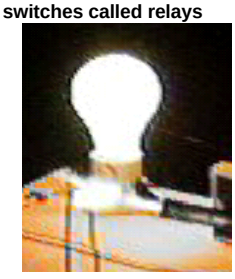
Week 2: Plan

- **Binary Storage**
- **Number Representations**
- **Positive Numbers: Unsigned Binary Integers**
- **Negative Numbers: Two's Complement Integers**
- **Floating Point Representations**
- **Binary Logic and Arithmetic**

I. Positive Binary Numbers and Addition

Binary Storage

- **Electronic computer memories cannot store numbers & characters directly**
- **Electromagnetic computers of the 40s used electrical switches called relays**



Binary Storage

- **Relays work like light switches**
  - except they can be turned on or off by a current.
- **Electronic computers use transistors**
  - do the same thing without any mechanical action.
- **When a light bulb is on, one can read the charge on the bulb as a voltage: 110 V.**
  - When it is off this voltage is 0 V.
- **Electronic computers can read the voltages in transistor switches: on + 5 V off 0 V.**
  - This is what constitutes their memory
  - There is no halfway between voltage readings: it is on or off!

Binary vs. Digital

- **Digital means that a signal stored in memory can have only a fixed number of values.**
- **Binary means it can only have two values.**
  - On or off.
- **The word "bit" comes from binary digit.**
- **A bit can only hold a 1 or a 0 nothing else**
  - Not A, Z, 3, or nothing.
  - Just 0 or 1.
- **Bits in computer memory are grouped in cells.**
  - Typically 8 bits (or a byte) long.
  - Although this number is arbitrary, it is typically a power of 2: *8, 16, 32, 64*

Bits in Memory

- **Below you see 4 cells of 8 bits.**
- **Which of these represents a byte in binary computer memory?**

0	1	1	1	1	0	1	0
J	A	N	U	A	R	Y	!
			1	0	0	1	0
8	9	4	2	5	3	4	1

- **So numbers or letters must be coded first!**

Coding Integer Numbers in Binary

- **Integers can be positive or negative: -1, 0, 1, 2, 3**
  - Let's consider positive or unsigned binary integers first.
- **But before we do that, how do humans count integers?**
  - Using our hands!


Coding Integer Numbers in Binary

- **This will only get us to 10. Now what do we do?**
- **Ask someone to count the number of 10s!**

20!

What's Going on Here?

0	10	20 ...	90	*ask someone to count 100s*
1	11	21	91	100 200 300 … 999
2	12	22	92	*ask someone to count 1000s*
3	13	23	93	1000 2000 3000 … 9999
4	14	24	94	
5	15	25	95	etc...
6	16	26	96	
7	17	27	97	This is called the decimal
8	18	28	98	system of counting or base 10
9	19	29	99	101 =

$1 \times 10^2 + 0 \times 10^1 + 1 \times 10^0 = 101$

What's Going on Here?

- **So if you have a memory (fingers) that counts to 10.**
- **When you run out of memory, add extra memory to count the 10s.**
- **When you run out of memory again, add extra memory to count the 100s**
- **Etc…**
- **The only reason we count to 10 is because that's where our finger memory stops!!!**
- **This means we can do this trick with any number...**

So what if we had 8 fingers?

0	**10 (8)**	**...**	**70 (7*8 = 56)**	**… 100 (8*8 = 64)**
1	**11**		**71**	
2	**12**		**72**	
3	**13**		**73**	
4	**14**		**74**	
5	**15**		**75**	
6	**16**		**76**	
7	**17 (15)**		**77**	**This is called the octal system of counting or base 8**

101 =

$1 \times 8^2 + 0 \times 8^1 + 1 \times 8^0 = 65$

So what if we had 2 fingers?

0	**10 (2)**	**100 (2*2 = 4)**	**1000 (2*2*2 = 8)**
1	**11**	**101**	
		110	
		111	

This is called the binary system of counting or base 2

101 =

$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$

Radix-weighted Positional Number System

- **Counting is a radix-weighted positional system**
  A series of n digits *d*:
  $d_{n-1}d_{n-2} \ldots d_2d_1d_0$
- **Represent a polynomial:**
  $d_{n-1} * B^{n-1} + d_{n-2} * B^{n-2} + \ldots + d_2 * B^2 + d_1 * B^1 + d_0 * B^0$

  where B is the base or radix for the number system, and each digit, d_i, has a value between 0 and B - 1.

For Example: Base 2 or Binary

- **When the base is 2 and n equals 4:**

  1 0 1 0

  $= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$

  $= 8 + 0 + 2 + 0 = 10$ decimal

- **What we just did is called Base Conversion**
  - From base 2 to base 10

From Decimal Back to Binary

- **This is a little trickier.**
- **(1) find the largest power of two that is less than or equal to the number to be converted**
- **(2) subtract that power of two from the number to be converted and repeat step 1 with the result.**
- **(3) Write a 1 in the bit positions corresponding to the powers of 2 you subtracted, or else 0**
  - $22 - 2^4 = 6$ -> 1
  - $6 - 2^{3=8}$ is not possible -> 0
  - $6 - 2^2 = 2$ -> 1
  - $2 - 2^1 = 0$ -> 1
  - $0 - 2^0$ is not possible -> 0

  = 1 0 1 1 0

From Decimal Back to Binary

- **Another method uses remainders, but is the same.**
- **Divide the number by 2, the base, and write down the remainder.**

22 / 2	**remainder**	
11	**0**	
5	**1**	by now we have divided by 4
2	**1**	by now we have divided by 8
1	**0**	by now we have divided by 16
0	**1**	by now we have divided by 32 = 2^n

- **Now read back in reverse order: 10110!**

Range

- **Remember?**
- **Computer memory is divided up into cells with a finite number of bits per memory location**
  - The range of numbers which fit into one memory location is limited.
- **The range of numbers that can fit in an n-bit cell**
  - from 00...000 (n zeros) to 11...111 (n ones),
  - Representing the numbers from 0 to 2^n - 1
  - It can hold 2^n numbers
  - However, the largest number is 2^n - 1!!!!
  - So with 8 bits, the largest number is 255 and the total is 256 (255 and 0)

So what happens when we add?

- **Unsigned Addition**
  - Works like addition with decimal numbers, but is easier
  - because you only need to know the rules for 0 and 1
  - Note that these rules are easy to implement using the simple logic gates we discussed last week.

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = oops!

- **We need to carry a bit forward to the next memory location here:**

1 + 1 = 10

So what happens when we add?

- **Note that the same thing happens in decimal:**

9 + 1 = oops!
10
99 + 1 = oops!
100

6
8
+
14

- **We add until we reach ten for that digit, then we carry 1 and leave the remainder (4).**

Binary Example

- **010110 (22)**
- **011100 (28)**
- **---------- +**
- **110010 (50)**

- **It is clear why binary numbers were chosen for computers: it is so easy to implement!**

The Carry Bit

- **So what happens when we can no longer carry because our memory cell is full?**
  - E.g., our memory cell is 8 bits
  - What happens when we add 128?
- **For this, the ALU in the computer, which does all the additions, has an extra bit set aside**
- **This bit is called the carry bit.**
- **Is set when the most significant bit overflows**
  - The bit on the left in most computers
  - The bit on the right is called “least significant bit”
  - The carry bit can then be used for the least sign. bit of the next cell to form a 16-bit “word”.

6-Bit Example (Range 64)

•010110 (22) **110110 (54)**
•011100 (28) **011100 (28)**
•---------- + **---------- +**
•C= 0 110010 (50) **C=1 010010 (64 + 18)**

•No carrying is necessary for larger bits, as the ALU cannot operate on numbers larger than the data path and register size allows.
•In the above example, would be a 6 bit register size.
  - In modern computers: 32 - 64 bits!

II.
Signed Binary Numbers and Subtraction

Representing Negative Numbers

- **We are used to the following addition rule for negative numbers: *a + -a = 0***
  - So *a* negated, then added makes 0
  - However, computers cannot represent minus signs.
  - You would have a ternary computer
  - You don't want to add hardware for this...
- **Let's use the most significant bit to indicate a number is positive (0) or negative (1).**
  - But if 1000 0000 is - 0 and 0000 0000 is + 0
  - Programmers have to make extra checks
  - Addition of negative numbers wouldn't work the same as addition of positive numbers.

For Example

- **If we would do this, we would get**

00 0101 (5)
10 0101 (-5)
---------- +
C=0 10 1010 (-10????)

- **That's not zero!**

Let's Try the Inverse (NOT)

- **Also called a one's complement**
- **If we would do this, we would get**

00 0101 (5)
11 1010 (-5, inverse)
---------- +
C=0 11 1111 (-0 with all 1s?)

- **That's not a real zero either!**
- **We'd like zero to be 0000 0000!**

Let's Cheat and *Make* it Zero!

00 0101 (5)
11 1010 (-5)
---------- +
C=0 11 1111 (- 0 with all 1s?)
00 0001
---------- +
C=1 00 0000

Now that's a real zero, if we forget the carry

Two's Complement or Negation

- **This is called *two's complement representation.***
- **We negate (NEG) by inversing (NOT) and adding 1**
- **We still have the most sign. bit indicating sign!**
  - 1000 0000 (-128)
  - 1111 1111 (-1)
  - 0000 0000 (0)
  - 0111 1111 (127)
- **But normal addition rules apply!**
- **This does mean we now have only half the range.**
  - For 8-bit numbers 127 (2^{n-1} -1) to -128 (2^{n-1})
  - Zero is 0000 0000, loose one positive number

Another example

- **Let's take 23, negate its binary number**

```
          00010111    (23)
          11101001    (-23)
          ----------  +
C=1       00000000    (0)
```

- **If we now forget the carry bit…we have 0!**

So how does this work?

- **Well, positive numbers is straightforward**
  - 0000 0000 to 0111 1111: 0 to 127
- **There is two ways of looking at negative numbers:**

  (1) We just count using zeros instead of ones
  - 1111 1111 -1 NOT = 0000 0000 0
  - 1111 1110 -2 NOT = 0000 0001 1
  - 1111 1101 -3 NOT = 0000 0010 2

  (2) We just count from -128 up
  - 1000 0000 -128 0000 0000 0
  - 1000 0001 -127 0000 0001 1
  - 1000 0010 -126 0000 0010 2

  Either way always fill unused most sign. bits with 1!

Subtraction in Two's Complement

- **The problem of "borrow" is similar in binary subtraction to that in decimal:**

		A	B	A - B	
1008		1	0	1	C=1 11 1101 (1d)
9		1	1	0	00 0010 (2d)
-------	-	0	0	0	---------- -
999		borrow 1… 0	1	1	11 1111 (-1d)

borrow

- **We can construct a subtraction table that has two parts.**
  - Three cases of subtracting without borrow
  - One case where we have to borrow a digit, no matter how far to the left is the next available 1.

Subtraction in Two's Complement

- **However, we do not need any special circuitry for subtraction**
  - First you negate the number to be subtracted
  - Then you perform an add operation

```
001010 (10 dec.)        001010
000100 (4 dec.)         111100
---------- -            ---------- +
000110          C= 1    000110
```

- **However, the carry bit no longer indicates an overflow!**

The Overflow Bit

- **To signal a sign change in two's complement interpretation of integer arithmetic**
  - The processor has an extra bit: the overflow bit
- **The overflow bit (V) is set when two positive numbers turn into a negative one**
  - or two negative numbers turn into a positive one
  - Processor detects this by comparing the bit carried into the sign bit with the carry bit.
  - If they are different, overflow has occurred
- **Carry bit is signals overflow for unsigned interpret.**
- **Overflow bit signals overflow for signed interpret.**
  - Both are always set independently of interpretation
  - It's up to the software to check the C and V bits.

Some Examples with Addition

```
0 0                  0 1 Carry in but not out   0 0
00 0011              010110                     00 0101
01 0101              001100                     11 0111
-----------+         ---------- +               ---------- +
011000               100010                     111100
C= 0 V=0             C=0 V=1                    C=0 V=0

1 1                  1 1                        1 0 Carry out but not in
00 1000              111010                     10 0110
11 1010              110111                     10 0010
---------- +         ---------- +               ----------+
00 0010              110001                     00 1000
C=1 V=0              C=1 V=0                    C=1 V=1
```

Overflow in Subtraction

- **Overflow in subtraction can occur during borrowing.**
  - Remember, the carry bit can be set when borrowing
  - However, one can also borrow from the sign bit
- **Overflow occurs when number we are subtracting from is smaller than the number subtracted**
  - When there is a borrow from the carry bit into the sign bit, but no borrow from the sign bit
  - Or when there is a borrow from the sign bit, but not from the carry bit.
- **So basically C and V signal the error that occurs!**

Some Examples with Subtraction

1. Subtraction of negative number from positive (using signed interpretation).

		unsigned interp.	signed interp.	
	0111	7	7	Unsigned:
-	1000	8	-8	Check C should be -1 (not possible)
				Signed:
C=1,V=1	1111	15	-1	check V this should be 15!

There was a borrow into the most significant bit, but not out of it.

1. Subtraction of positive number from negative (using signed interpretation).

		unsigned interp.	signed interp.	
	1000	8	-8	
-	0111	7	7	
C=0,V=1	0001	1	1	check V this should be -15!

There was no borrow into the most significant bit, but there was borrow out of it.

Decimal to Signed Binary

- **Convert a negative number from decimal to binary**
  - First, convert its magnitude from decimal to unsigned binary
  - Then negate by taking two's compliment: NOT + 1
  - **-7 (dec) in 8 bit cell**
  - **+ 7 (dec) = 0000 0111 (bin)**
  - **NOT 0000 0111 = 1111 1000**
    0000 0001
    ---------- +
    1111 1001 (-7 dec)

Signed Binary to Decimal

- **First check the sign bit.**
  - If 0 the number is positive and you are done
  - If 1 the number is negative you negate (NOT + 1) and convert as you would unsigned binary

NOT 1101 1010 = 0010 0101
0000 0001
---------- +
0010 0110
32 + 4+ 2 = 38 (decimal)
thus -38

Another method

- **Add place values of zeros and subtract 1**

(1) We just count using zeros instead of ones
- 1111 1111 -1 NOT = 0000 0000 0
- 1111 1110 -2 NOT = 0000 0001 1
- 1111 1101 -3 NOT = 0000 0010 2

1101 1010
- (32 + 4 + 1) - 1 = -38

III.
Binary Logic

Logical Operators: AND

- **We have seen how we can do simple math with binary numbers: ADD, NEG, SUB (NEG+ADD)**
  - These are called arithmetic operations
  - Bits are interpreted as numbers
- **But because bits can represent true or false we can also do logic.**
  - NOT is an example of this!
  - Bits are interpreted true or false
- **These are called logical operators**
  - Logical operations do not affect carry and overflow

Logical Operators: AND

- **AND means if both bits are true, the result is true**
  - If ape and naked then man
  - In C: &&

p	*q*	*p* AND *q*
0	*0*	*0*
0	*1*	*0*
1	*0*	*0*
1	*1*	*1*

Truth Table for AND

Logical Operators: OR

- **OR means if one of the bits is true, result is true**
  - If sun or moon there is light
  - In C: II

p	*q*	*p* OR *q*
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*1*

Truth Table for OR

Logical Operators: XOR

- **Exclusive OR (XOR) means if ONLY one of the bits is true, result is true**
  - If sun or moon there is light
  - But during eclipse they cancel each other out!

p	*q*	*p* XOR *q*
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*0*

Truth Table for XOR

Other Operations

- **Arithmetic Left Shift**
  - Each bit in cell shifts one place to left
  - Equivalent to multiplying by 2
  - affects Carry and oVerflow
- **Arithmetic Right Shift**
  - Each bit in cell shifts one place to right
  - Equivalent to dividing by 2
  - affects Carry only
- **Logical Left Shift and Logical Right Shift**
  - Same as arithmetic left shift but leaves C and V alone
- **Note that these operations, like NOT and NEG do not require two groups of bits: they are *unary* operations.**

Arithmetic Shift Left

- **ASL 111100** **C=1 111000** **unsigned: 60 => 56 (120)**
  V=0 **signed: -4 => -8**
- **ASL 000011** **C=0 000110** **unsigned: 3 => 6**
  V=0 **signed: 3 => 6**
- **ASL 010110** **C=0 101100** **unsigned: 22 => 44**
  V=1 **signed: 22 => -20 (44)**

- **So it simply doubles the number, regardless of sign!!!**
- **Again, the C bit detects overflow of unsigned values, V bit detects overflow for signed values.**

Arithmetic Shift Right

- **ASR 010100** **001010 C=0** **signed:** **20 => 10**
- **ASR 010111** **001011 C=1** **signed:** **23 => 11 (11.5)**
- **ASR 110010** **111001 C=0** **signed:** **-14 => -7**
- **So it simply halves the number**
  - It will round down when number not even
- **ASR is specifically designed for signed numbers**
  - Use logical shift right for unsigned, as it will not borrow
  - Two's complement oVerflow is impossible!!!