CSE 2421

Integer Representation and Basic Operations

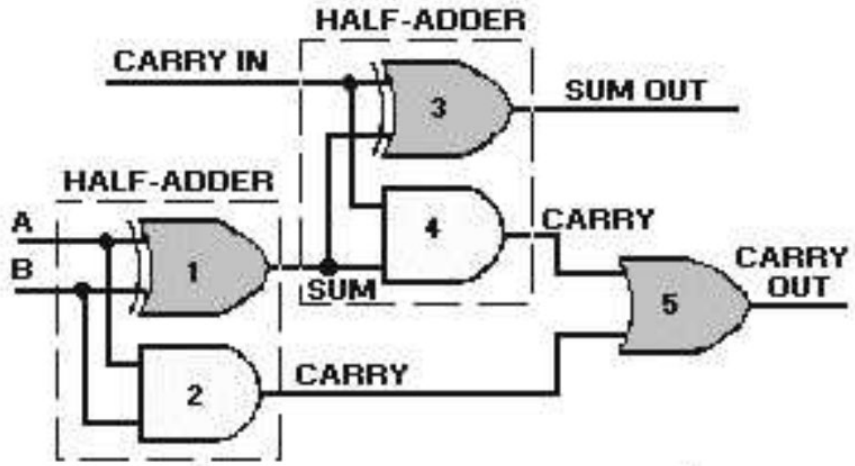
Required Reading: *Computer Systems: A Programmer's Perspective, 3rd Edition*

- Chapter 4, Sections 4.2 through 4.2.2
- Chapter 2, Sections 2.3 through 2.3.8
- Chapter 8, Section 8.2 through 8.2.4

Do you know how logic gates work?

- http://www.tutorialspoint.com/computer_logical_organization/logic_gates.htm

Bit operations



A	B	CARRY IN	SUM OUT	CARRY OUT
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

1 and 3 ➔ exclusive OR (^)
2 and 4 ➔ and (&)
5 ➔ or (|)

```
01100  carry*
 0110  a
 0111  b
-----
01101  a+b
```

* Always start with a carry-in of 0

Did it work?
What is a?
What is b?
What is a+b?
What if 8 bits instead of 4?

Integer Addition

- We will not worry about the intermediate outputs generated by the hardware; we will only be concerned with:
- The three input bits: 1) CARRY IN, 2) A, and 3) B.
- And the two output bits: 4) SUM OUT, and 5) CARRY OUT.
- Let's look at the bit by bit addition of the two 4 bit operands on the preceding slide: 0110 and 0111.

Example 4 bit addition

- CARRY IN 1100
- Operand A 0110
- Operand B 0111
- SUM OUT 1101
- CARRY OUT 0110
- Notes:
  - For addition, carry in to the least significant pair of bits is always 0.
  - Carry out from each pair of bits is propagated to carry in for the next most significant pair of bits.
  - For unsigned addition, if carry out from the most significant pair of bits is 0, there is no overflow.

Integer Representation

- Different encoding scheme than float
- Total number of distinct bit patterns (values): 2^w
  - where w is the bit width (number of bits) of the digital representation
- The left-most bit is the sign bit if using a signed data type
- Unsigned → non-neg numbers (>=0)
  - Minimum value: 0
  - Maximum value: 2^w-1
- Signed → neg, zero, and pos numbers
  - Minimum value: -2^{w-1} (2's complement – see below)
    - (Reminder: exponent before negative sign)
  - Maximum value: $2^{w-1}-1$

Acronyms for Decoding Schemes

- B2U – Binary to unsigned
- B2T – Binary to two's-complement
- B20 – Binary to ones'-complement
- B2S – Binary to sign-magnitude

Integer Decoding

- Binary to Decimal
- Unsigned = simple binary = B2U
  - 0101 = 5, 1111 = F, 1110 = E, 1001 = 9
- Signed = two's complement = B2T
  - 0 101 = positive number; same as B2U = 5
  - 1 111 = $-1*2^3 + 7 = -8 + 7 = -1$
  - 1 110 = $-1*2^3 + 6 = -8 + 6 = -2$
  - 1 001 = $-1*2^3 + 1 = -8 + 1 = -7$
  - Another way, if sign bit = 1, then it's a negative number and to get the magnitude of that number, you:
    - invert bits and add 1
  - Reminder: left-most bit is sign bit

CODE	B2U	B2T
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	**8**	**-8**
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

B2O & B2S

- One's complement – bit complement of B2U for negatives
- Signed Magnitude – left most bit for sign, B2U for the remaining bits
- Both include neg values
- Min/max = $-(2^{w-1}-1)$ to $2^{w-1}-1$
- *Positive and negative zero*
- Difficulties with arithmetic operations (that's why these encodings are not used for integers anymore)

CODE	B2U	B2T	B2O	B2S
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	**8**	**-8**	**-7**	**-0**
1001	9	-7	-6	-1
1010	10	-6	-5	-2
1011	11	-5	-4	-3
1100	12	-4	-3	-4
1101	13	-3	-2	-5
1110	14	-2	-1	-6
1111	15	-1	-0	-7

Signed vs Unsigned

- Casting – signed to unsigned, unsigned to signed…
- Changes **the meaning or interpretation** of the value, but ***not*** the bit representation

CODE	B2U	B2T
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	**15**	**-1**

Signed vs Unsigned (cont)

- When an operation is performed where one operand is signed and the other is unsigned, C implicitly casts the signed operand to unsigned (remember the type conversion hierarchy), then performs the operations
- Arithmetic operations in the machine are not affected, since bit representation doesn't change
- Relational operators could be affected (Force unsigned immediate value with suffixed 'u' in C)

Signed vs. Unsigned

- 0 == 0u → True
  - Unsigned relational operator
- -1 < 1 → True
  - Signed relational operator
- -1 < 1u → False
  - Unsigned relational operator
  - -1 is stored as 0xFFFFFFFF (4 byte int)
    - Interpreted as an unsigned value, this is the largest integer that fits in 32 bits!
- NOTE: For integers, w = 32 on stdlinux

Sign Extend

- Sign Extension
  - For unsigned fill to left with zero
  - For signed repeat sign bit (MSB, or most significant bit)
- `char x = -27; /*w = 8 in C*/`
- `short y; /*w = 16 on stdlinux for short*/`
  - `27 = 0001 1011`
  - `-27 = 1110 0101  /* invert +1 to get 2's * complement */`
- `y = (short)x;`
- `x is sign-extended to 16 bits (on stdlinux), and this sign-extended bit pattern is assigned to y:`
  - `-27 = 1111 1111 1110 0101`

Truncation

- Drops the high order w-k bits when truncating a w-bit number to a k-bit number
- `short y = -27; /*w = 16 on stdlinux*/`
  - `27 = 0000 0000 0001 1011`
  - `-27 = 1111 1111 1110 0101`
    - `/*invert + 1 to get 2's complement*/`
- `char x = (char)y; /*w = 8 for char in C*/`
- `x = (char)y;`
  - `-27 = 1110 0101 /* truncate high-order w - k, or 16 - 8, bits */`

Truncation – loss of information

- When the high order w-k bits are dropped when truncating a w-bit number to a k-bit number, does the value change?

Truncation – loss of information

- When the high order w-k bits are dropped when truncating a w-bit number to a k-bit number, does the value change?
- Answer:
  - Unsigned:
    - If all of the truncated bits are 0, value is preserved.
  - Signed:
    - If the most significant bit (msb) in the truncated number is the same as each of the truncated bits, value is preserved.

Truncation

HEX		UNSIGNED – B2U		TWO'S COMP – B2T	
orig	trunc	orig	trunc	orig	trunc
0 (0000)	0 (000)	0	0	0	0
2 (0010)	2 (010)	2	2	2	2
9 (1001)	1 (001)	9	1	–7	1
B (1011)	3 (011)	11	3	–5	3
F (1111)	7 (111)	15	7	–1	–1

Integer Addition

- Unsigned
- Overflow when $x+y > 2^w - 1$

- Example:
- Unsigned 4-bit BTU
- The processor only needs to check carry-out from most significant bits
- Overflow > 15

x	y	x+y	result
8 1000	5 0101	13 1101	13 ok
8 1000	7 0111	15 1111	15 ok
12 1100	5 0101	17 10001	1 OF

Integer Addition

- Signed
- Negative overflow when $x+y < -2^{w-1}$
- Positive overflow when $x+y > 2^{w-1}-1$
- Result incorrect if
  carry-in != carry out
  of top bit (msb)

- Example:
- Signed 4-bit B2T
- Positive overflow > 7
- Negative overflow < -8

x	y	x+y	result
-8 1000	-5 1011	-13 1 0011	3 Neg OF
-8 1000	-8 1000	-16 1 0000	0 Neg OF
-8 1000	5 0101	-3 1101	-3 ok
2 0010	5 0101	7 0111	7 ok
5 0101	5 0101	10 0 1010	-6 Pos OF

B2T integer negation

- How to determine a negative value in B2T?
  - Reminder: B2U = B2T (for positive values)
  - To get B2T representation of negative value of a B2U bit pattern → invert the B2U bit pattern and add 1
- Two's complement negation (for a w bit representation):
  - -2^{w-1} is *its own additive inverse*
    - *Additive inverse is the value added to a given value to get 0.*
  - To get additive inverse of other values, use integer negation (invert the bits, and add 1) [this also works for -2^{w-1}]

B2T integer negation

GIVEN			NEGATION		
HEX	binary	base 10	base 10	binary*	HEX
0x00	0b00000000	0	0	0b00000000	0x00
0x40	0b01000000	64	-64	0b11000000	0xC0
0x80	0b10000000	-128	-128	0b10000000	0x80
0x83	0b10000011	-125	125	0b01111101	0x7D
0xFD	0b11111101	-3	3	0b00000011	0x03
0xFF	0b11111111	-1	1	0b00000001	0x01
			*binary = invert the bits and add 1		

Sign/Unsign+Negation example

```
#include <stdio.h>
#include <limits.h>
void main()    {
  int n = 0;
  printf("neg of %d is %d ",n,-n);
  n = 64;
  printf("\nneg of %d is %d ",n,-n);
  n = -64;
  printf("\nneg of %d is %d ",n,-n);
  n = INT_MIN;  /* INT_MIN is defined in limits.h */
  printf("\nneg of %d is %d ",n,-n);

  unsigned int a = 0;
  printf("\n\n0 - 1 unsigned is %u ",a-1);
  printf("\n unsigned max is %u ", UINT_MAX);
  a = 5;
  printf("\nnegof unsigned %d is %u",a, -a);   }
```

Output?

Sign/Unsign+Negation example

```
#include <stdio.h>
#include <limits.h>
void main()    {
  int n = 0;
  printf("neg of %d is %d ",n,-n);     /*negation of 0 is 0*/
  n = 64;
  printf("\nneg of %d is %d ",n,-n);   /*negation of 64 is -64*/
  n = -64;
  printf("\nneg of %d is %d ",n,-n);   /*negation of -64 is 64*/
  n = INT_MIN;
  printf("\nneg of %d is %d ",n,-n);   /*negation of
                                         -2147483648 is -2147483648 */
  unsigned int a = 0;
  printf("\n\n0 - 1 unsigned is %u ",a-1);
                                        /*0 - 1  unsigned is 4294967295 */
  printf("\n unsigned max is %u ", UINT_MAX);
                                        7*unsigned max is 4294967295*/
  a = 5;
  printf("\nnegof unsigned %d is %u",a, -a);
                                    /*negation of unsigned 5 is 4294967291 */
}
```

Rounding

y	round down (towards −∞)	round up (towards +∞)	round towards zero	round away from zero	round to nearest
+23.67	+23	+24	+23	+24	+24
+23.50	+23	+24	+23	+24	+24
+23.35	+23	+24	+23	+24	+23
+23.00	+23	+23	+23	+23	+23
0	0	0	0	0	0
−23.00	−23	−23	−23	−23	−23
−23.35	−24	−23	−23	−24	−23
−23.50	−24	−23	−23	−24	−24
−23.67	−24	−23	−23	−24	−24

Rounding – our system

```
#include <stdio.h>
void main(){
    float y[9]={23.67, 23.50, 23.35, 23.00, 0, -23, -23.35, -23.5, -23.67};
    int i;
    for (i=0;i<9;i++) {
         printf("y = %.4f %.2f %.1f %.0f\n",
                        y[i], y[i],y[i], y[i]);
    }
}
```

/*OUTPUT ROUND TO NEAREST*/

y = 23.6700 23.67 23.7 24
y = 23.5000 23.50 23.5 24
y = 23.3500 23.35 23.4 23
y = 23.0000 23.00 23.0 23
y = 0.0000 0.00 0.0 0
y = -23.0000 -23.00 -23.0 -23
y = -23.3500 -23.35 -23.4 -23
y = -23.5000 -23.50 -23.5 -24
y = -23.6700 -23.67 -23.7 -24