Computer Science 210

Computer Systems 1

Lecture Notes

Lecture 4

Arithmetic & Other Operations

Credits: Adapted from slides prepared by Gregory T. Byrd, North Carolina State University

Unsigned Integers

- Non-positional notation
  - could represent a number ("5") with a string of ones ("11111")
  - problems?
- Weighted positional notation
  - like decimal numbers: "329"
  - "3" is worth 300, because of its position, while "9" is only worth 9

329: 10^2 10^1 10^0

3x100 + 2x10 + 9x1 = 329

most significant → 101 ← least significant: 2^2 2^1 2^0

1x4 + 0x2 + 1x1 = 5

2-2

Unsigned Integers (cont.)

- An *n*-bit unsigned integer represents any of 2^n (integer) values:
  from 0 to 2^n-1.

2^2	2^1	2^0	Value
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

2-3

Unsigned Binary Arithmetic

- Base-2 addition – just like base-10!
  - add from right to left, propagating carry

```
                       carry
  10010          10010          1111
+  1001        +  1011        +    1
  11011          11101         10000
```

Subtraction, multiplication, division,...

2-4

Signed Integers

- With n bits, we can distinguish 2^n unique values
  - assign about half to positive integers (1 through 2^{n-1}) and about half to negative (-2^{n-1} through -1)
  - that leaves two values: one for 0, and one extra
- Positive integers
  - just like unsigned, but zero in *most significant* (MS) bit
    00101 = 5
- Negative integers
  - Sign-Magnitude (or Signed-Magnitude) – set MS bit to show negative, other bits are the same as unsigned
    10101 = -5
  - One's complement – flip every bit to represent negative
    11010 = -5
  - In either case, MS bit indicates sign: 0=positive, 1=negative

2-5

Two's Complement

- Problems with sign-magnitude and 1's complement
  - two representations of zero (+0 and –0)
  - arithmetic circuits are complex
    - How to add two sign-magnitude numbers?
      - e.g., try 2 + (-3)
    - How to add two one's complement numbers?
      - e.g., try 4 + (-3)
- *Two's complement* representation developed to make circuits easy for arithmetic.
  - for each positive number (X), assign value to its negative (-X), such that X + (-X) = 0 with "normal" addition, ignoring carry out

```
   00101  (5)          01001  (9)
+  11011  (-5)      +  10111  (-9)
   00000  (0)       (1)00000  (0)
```

2-6

Two's Complement Representation

- If number is positive or zero,
  - normal binary representation, zeroes in upper bit(s)
- If number is negative,
  - start with positive number
  - flip every bit (i.e., take the one's complement)
  - then add one

```
  00101  (5)           01001  (9)
  11010  (1's comp)    10110  (1's comp)
+     1              +     1
  11011  (-5)          10111  (-9)
```

2-7

Two's Complement Signed Integers

- MS bit is sign bit – it has weight -2^{n-1}.
- Range of an *n*-bit number: -2^{n-1} through $2^{n-1} - 1$.
  - The most negative number (-2^{n-1}) has no positive counterpart.

-2^3	2^2	2^1	2^0	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7

-2^3	2^2	2^1	2^0	
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

2-8

Converting Binary (2's C) to Decimal

1. If leading bit is zero (a positive number) just convert as normal

```
X  =  01101000
   =  2^6+2^5+2^3
   =  64+32+8
X  =  104
```

n	2^n
0	1
1	2
2	4
3	**8**
4	16
5	**32**
6	**64**
7	128

Assuming 8-bit 2's complement numbers.

2-9

Converting Binary (2's C) to Decimal

- If the number is negative (MS bit is a 1)
- Same as before **EXCEPT** the MS bit is negative

```
X= 11100110
 = -2^7+2^6+2^5+2^2+2^1
 = -128+64+32+4+2
X=-26
```

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	-128

Watch a video http://www.youtube.com/watch?v=NUASXiqazc8

Assuming 8-bit 2's complement numbers.

2-10

Operations: Arithmetic and Logical

- Recall: a data type includes *representation* and *operations*.
- We now have a good representation for signed integers, so let's look at some arithmetic operations:
  - Addition
  - Subtraction
  - Sign Extension
- We'll also look at overflow conditions for addition.
- Multiplication, division, etc., can be built from these basic operations.
- Logical operations are also useful:
  - AND
  - OR
  - NOT

2-11

Addition

- As we've discussed, 2's comp addition is just binary addition.
  - assume all integers have the same number of bits
  - ignore carry out
  - for now, assume that sum fits in n-bit 2's comp. representation

```
   01101000 (104)        11110110 (-10)
+  11110000 (-16)     +  11110111 (-9)
   01011000 (88)      (1)11101101 (-19)
```

Assuming 8-bit 2's complement numbers.

2-12

Subtraction

- Negate subtrahend* (2nd no.) and add
- 10 – 4 = 6
- 10 + (-4) = 6
- We can do all subtraction using addition!!!
- Our computer will only need an adder circuit
- Much simpler

* minuend (c) – subtrahend (b) = difference (a)

- First let's look at why we **don't** want to subtract like we do with decimal numbers
- http://www.youtube.com/watch?v=jBY3iPYyzUY

2-13

Subtraction

- 104 – 16 = ?
- 104 + (-16) =?

```
  01101000 (104)
+ 11110000 (-16)
  01011000 (88)
```

2-14

Subtraction

- (-10) –(-9) = ?
- (-10) + 9 = ?

```
  11110110 (-10)
+ 00001001 (9)
  11111111 (-1)
```

2-15

Overflow

- If operands are too big, their sum cannot be represented as an *n*-bit 2's comp number
- 5 bits can represent 2^5 or 32 unsigned integers
- Or 0 to 15 positive and -1 to -16 as signed integers

unsigned		signed	
01110	(14)	01110	(14)
+ 01000	(8)	+ 01000	(8)
10110	(22)	10110	(-10)

- We have overflow in signed binary if:
  - signs of both operands are the same, and
  - sign of sum is different.
- Another test -- easy for hardware:
  - carry into MS bit does not equal carry out

2-16

Overflow

- If operands are too big, their sum cannot be represented as an *n*-bit 2's comp number
- 5 bits can represent 2^5 or 32 unsigned integers
- Or 0 to 15 positive and -1 to -16 as signed integers

signed	
10111	(-8)
+ 10111	(-9)
01110	(+14)

- We have overflow if:
  - signs of both operands are the same, and
  - sign of sum is different.
- Easy for hardware to test

2-17

Overflow

- Example: 4-bit signed
  - Two's complement - 8 <= x <= 7
  - Examples:

6 + 7 = 13 (outside the range)

0110 ← 6
+ 0111 ← 7
01100 carries
01101 (4-bit) => 1101

Answer = -3 Invalid answer

-6 + -7 = -13 (outside the range)

1010 ← -6
+ 1001 ← -7
10000 carries
10011(4-bit)= 0011

Answer = 3 Invalid answer

-2 + 3 = 1

1110 ← -2
+ 0011 ← 3
11100 carries
10001 (4-bit)=0001

Answer = 1 valid answer

2 + 3 = 5

0010 ← 2
+0011 ← 3
0100 carries
0101

Answer = 5 Valid answer

Addition/Subtraction with 2's Complement

- Two's complement representation allows addition and subtraction from a single simple adder.

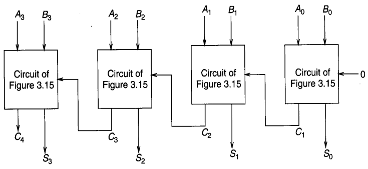

Figure 3.16 A circuit for adding two 4-bit binary numbers

- Circuit to add : S = A + B
- To subtract A – B : invert B and enable carry in

1-19

Logical Operations

- Operations on logical TRUE or FALSE
  - two states -- takes one bit to represent: TRUE=1, FALSE=0

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A
0	1
1	0

- View *n*-bit number as a collection of *n* logical values
  - operation applied to each bit independently

2-20

Examples of Logical Operations

- AND
  - useful for clearing bits
    - AND with zero = 0
    - AND with one = no change

```
     11000101
AND  00001111
     00000101
```

- OR
  - useful for setting bits
    - OR with zero = no change
    - OR with one = 1

```
     11000101
OR   00001111
     11001111
```

- NOT
  - unary operation one argument flips every bit

```
NOT  11000101
     00111010
```

2-21

Sign Extension

- Sometimes we want to convert a small number of bits into a larger number of bits
- If we just pad with zeroes on the left:

4-bit		8-bit	
0100	(4)	**00000100**	(still 4)
1100	(-4)	**00001100**	(12, not -4)

- Instead,
  - propagate the MS bit (the sign bit):

4-bit		8-bit	
0100	(4)	**00000100**	(still 4)
1100	(-4)	**11111100**	(still -4)

2-22

Hexadecimal Notation
(not a representation)

- It is often convenient to write binary (base-2) numbers using hexadecimal (base-16) notation instead.
  - fewer digits -- four bits per hex digit
  - less error prone -- easy to corrupt long string of 1's and 0's

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

Binary	Hex	Decimal
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

2-23

Converting from Binary Notation to Hexadecimal Notation

- Every four bits is a hex digit.
  - start grouping from right-hand side

011	1010	1000	1111	0100	1101	0111
⇩	⇩	⇩	⇩	⇩	⇩	⇩
3	A	8	F	4	D	7

This is not a new machine representation, just a convenient way to write the number.

This video shows you how to convert binary to hex
http://www.youtube.com/watch?v=W_NpD248CdE
(with binary to octal thrown in)

2-24

Representing Text

- American Standard Code for Information Interchange (ASCII)
  – Developed from telegraph codes, alternative to IBM's *Extended Binary Coded Decimal Interchange Code* (*EBCDIC*) in 1960s
  – Printable and non-printable (ESC, DEL, …) characters (127)
  – Limited set of characters – many character missing, especially language-specific
  – Many national "standards" developed

2-25

Text: ASCII Characters

- ASCII: Maps 128 characters to 7-bit code.
  – both printable and non-printable (ESC, DEL, …) characters
  – "ASCIIbetical" order

00	nul	10	dle	20	sp	30	0	40	@	50	P	60	`	70	p
01	soh	11	dc1	21	!	31	1	41	A	51	Q	61	a	71	q
02	stx	12	dc2	22	"	32	2	42	B	52	R	62	b	72	r
03	etx	13	dc3	23	#	33	3	43	C	53	S	63	c	73	s
04	eot	14	dc4	24	$	34	4	44	D	54	T	64	d	74	t
05	enq	15	nak	25	%	35	5	45	E	55	U	65	e	75	u
06	ack	16	syn	26	&	36	6	46	F	56	V	66	f	76	v
07	bel	17	etb	27	'	37	7	47	G	57	W	67	g	77	w
08	bs	18	can	28	(	38	8	48	H	58	X	68	h	78	x
09	ht	19	em	29	)	39	9	49	I	59	Y	69	i	79	y
0a	nl	1a	sub	2a	*	3a	:	4a	J	5a	Z	6a	j	7a	z
0b	vt	1b	esc	2b	+	3b	;	4b	K	5b	[	6b	k	7b	{
0c	np	1c	fs	2c	,	3c	<	4c	L	5c	\	6c	l	7c	|
0d	cr	1d	gs	2d	-	3d	=	4d	M	5d	]	6d	m	7d	}
0e	so	1e	rs	2e	.	3e	>	4e	N	5e	^	6e	n	7e	~
0f	si	1f	us	2f	/	3f	?	4f	O	5f	_	6f	o	7f	del

2-26