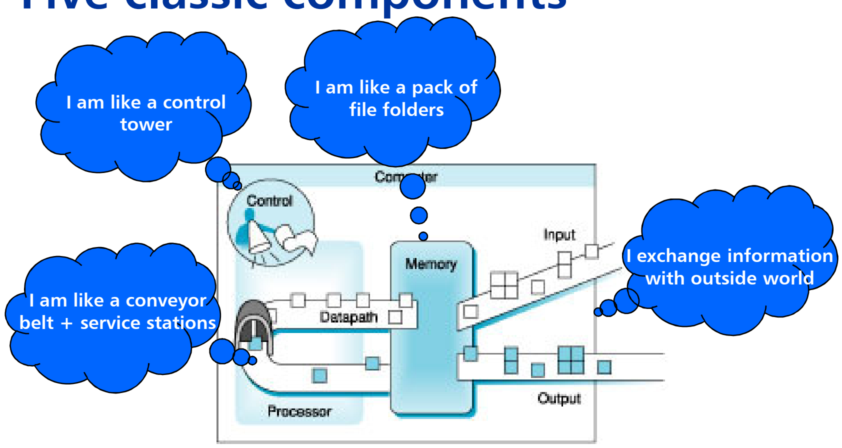
CS/COE0447: Computer Organization and Assembly Language

Chapter 3

modified by Bruce Childers
original slides by Sangyeun Cho

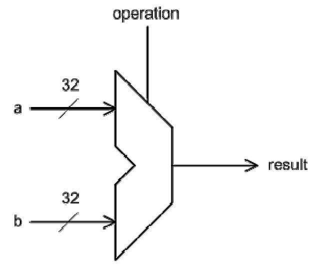
Dept. of Computer Science
University of Pittsburgh

Five classic components



Binary arithmetic

- (Sounds scary)
- So far we studied
  - Instruction set architecture basic
  - MIPS architecture & assembly language
- We will review binary arithmetic algorithms and their implementations
- Binary arithmetic will form the basis for CPU's datapath design



Binary number representations

- We looked at how to represent a number (in fact the value represented by a number) in binary
  - Unsigned numbers – everything is positive
- We will deal with more complicated cases
  - Negative numbers
  - Real numbers (a.k.a. floating-point numbers)

Unsigned Binary Numbers

- Limited number of binary numbers (patterns of 0s and 1s)
  - 8-bit number: 256 patterns, 00000000 to 11111111
  - in general, there are 2^N bit patterns, where N is bit width

    16 bit: $2^{16} = 65,536$ bit patterns

    32 bit: $2^{32} = 4,294,967,296$ bit patterns

- Unsigned numbers use patterns for ***0*** and ***positive numbers***
  - 8-bit number range [0..255] corresponds to

00000000	0
00000001	1
...	...
11111111	255

  - 32-bit number range [0..4294,967,295]
  - in general, the range is $[0..2^N-1]$

Unsigned Binary Numbers

- Binary addition
  - **`0 + 0 = 0, carry = 0 (no carry)`**
  - **`1 + 0 = 1, carry = 0`**
  - **`0 + 1 = 1, carry = 0`**
  - **`1 + 1 = 0, carry = 1`**

- Binary subtraction
  - **`0 – 0 = 0, borrow = 0 (no borrow)`**
  - **`1 – 0 = 1, borrow = 0`**
  - **`0 – 1 = 1, borrow = 1`**
  - **`1 – 1 = 0, borrow = 0`**

Unsigned Binary Numbers

- Binary arithmetic is straightforward
- Addition: Just add numbers and carry as necessary
- Consider adding 8-bit numbers:

carry

```
 01001111x
  01101011        107d
+ 01001101         77d
----------        ----
  10111000        184d
```

legal number: betw. 0 and 255

carry overflowed

```
 11101111x
  11101011        235d
+ 01001101         77d
----------        ----
 100111000        312d
```

illegal number: overflowed 8 bits

Unsigned Binary Numbers

- Binary arithmetic is straightforward
- Subtraction: Just subtract and borrow as necessary
- Consider subtracting 8-bit numbers:

borrow

```
    111
  01101011        107d
- 01001101         77d
----------        ----
  00011110         30d
```

legal number: betw. 0 and 255

```
 111111
  01101011        107d
- 01101101        109d
----------        ----
 111111110        -2d
```

illegal number: underflowed 8 bits
(i.e., “borrow overflow”)

Unsigned Binary to Decimal

- How to convert binary number?
  - First, each digit is position *i*, numbered right to left
  - e.g., for 8-bit number: $b_7 b_6 b_5 b_4\ b_3 b_2 b_1 b_0$
- Now, we just add up powers of 2
  - $b_0 \times 2^0 + b_1 \times 2^1 + b_2 \times 2^2 + \ldots + b_7 \times 2^7$
- An example

  1011 0111

  $= 1\times2^0 + 1\times2^1 + 1\times2^2 + 0\times2^3 + 1\times2^4 + 1\times2^5 + 0\times2^6 + 1\times2^7$

  $= 1 + 2 + 4 + 0 + 16 + 32 + 0 + 128$

  = 183d
- $v = \sum (b_i \times 2^i)$, where $0 \leq i \leq K$-1, where K=# bits, i is bit posn

Unsigned Binary Numbers in MIPS

- MIPS instruction set provides support
  - addu \$1,\$2,\$3 - adds two unsigned numbers (\$2,\$3)
  - addiu \$1,\$2,10 - adds unsigned number with **signed** immediate
  - subu \$1,\$2,\$3 - subtracts two unsigned numbers
  - etc.
- Primary issue: **The carry/borrow out is ignored**
  - Overflow is possible, but it is ignored
  - Signed versions take special action on overflow (we'll see shortly!)
- Unsigned memory accesses: lbu, lhu
  - Loaded value is treated as unsigned number
  - Convert from smaller bit width (8 or 16) to a 32-bit number
  - **Upper bits in the 32-bit destination register are set to 0s**

Important 7-bit Unsigned Numbers

- American Standard Code for Information Interchange (ASCII)
  - Developed in early 60s, rooted in telecomm
  - Maps 128 bit patterns (2^7) into control, alphabet, numbers, graphics
  - Provides control values present in other important codes (at the time)
  - 8^{th} bit might be present and used for error detection (parity)

- Control: Null (0), Bell (7), BS (8), LF (0A), CR (0D), DEL (7F)
- Numbers: (30-39)
- Alphabet: Uppercase (41-5A), Lowercase (61-7A)
- Other (punctuation, etc): 20-2F, 3A-40, 5E-60, 7B-7E

- Unicode: A larger (8,16,32 bit) encoding; backward compatible with ASCII

Signed Numbers

- How to represent positive and ***negative*** numbers?

- We still have a limited number of bit patterns
  - 8-bit: 256 bit patterns
  - 16 bit: 2^{16} = 65,536 bit patterns
  - 32 bit: 2^{32} = 4,294,967,296 bit patterns

- Re-assign bit patterns differently
  - Some patterns are assigned to negative numbers, some to positive

- Three ways
  - Sign magnitude, 1's complement, 2's complement

Method 1: sign-magnitude

- Same method we use for decimal numbers
- {sign bit, absolute value (magnitude)}
  - Sign bit (msb): 0 – positive, 1 – negative
  - Examples, assume 4-bit representation
    - 0000 +0
    - 0011 +3
    - 1001 -1
    - 1111 -7
    - 1000 -0 (two 0's???)
- Properties
  - Two 0s – a positive 0 and a negative 0?
  - Equal # of positive and negative numbers
  - A + (-A) does not give zero!
  - Consider sign during arithmetic

Sign-magnitude

- Let's check A + (-A) is not zero
- Consider N = 5 bits number. Zero is 00000 or 10000.
- Try this: -4 + 4 = ?????

```
-4 is 10100
 4 is 00100

so, let's add them together:
  10100  -4d
+ 00100   4d
-------  ---
  11000  -8d  YIKES!
```

Method 2: one's complement

- Negation of +X is $((2^N – 1) – X)$, where N is number of bits
  - A + (-A) = $2^N – 1$ (i.e., -0)
  - Given a number A, it's negation is done by (1111…1111 – A)
  - In fact, simple bit-by-bit inversion will give the same-magnitude number with a different sign
  - Examples, assume 4-bit representation
    - 0000
    - 0011
    - 1001
    - 1111 -0
    - 1000
- Properties
  - There are two 0s
  - There are equal # of positive and negative numbers
  - A+(-A) = 0 (whew!) but… A+0=A only works for +0 (try it with -0!)
  - 2 step process for subtraction (accounts for "carry out")

One's Complement

- Negation of X $(2^N – 1) – X)$, positive are usual value
- Consider N=4

Binary	One's	Binary	One's
0000	0	1000	-7
0001	1	1001	-6
0010	2	1010	-5
0011	3	1011	-4
0100	4	1100	-3
0101	5	1101	-2
0110	6	1110	-1
0111	7	1111	-0

notice how the counting works: 1111 is -0… then -1… -2… etc.

One's Complement

- Let's check the "0 property": A + (-A) = 0
- Suppose A = 5

```
5 is 0101
negation of 5 is (2⁴-1)-5 = (16-1) - 5 = 15 - 5 = 10
10 (unsigned) is 1010
check the table: 1010 is -5 in 1's complement
now, let's try 5 + (-5) in 1's complement

  0101          5          1010
+ 1010         -5        + 0000  (+0)        11    0)
------       ----        ------            ---
  1111         -0          1010  (-5)            (-6)
```


Method 3: two's complement

- Negation is (2^N – X)
  - A + (-A) = 2^N
  - Given a number A, it's negation is done by (1111…1111 – A) + 1
  - In fact, simple bit-by-bit inversion followed by adding 1 will give the same-magnitude number with a different sign
  - Examples, assume 4-bit representation
    - 0000
    - 0011
    - 1001
    - 1111
    - 1000 ?
- Properties
  - There is a single 0
  - There are unequal # of positive and negative numbers
  - Subtraction is simplified - one step based on addition (we'll see! ☺)

Two's Complement

- Negation of X (2^N – X), positive are usual value
- Consider N=4

Binary	One's	Binary	One's
0000	0	1000	-8
0001	1	1001	-7
0010	2	1010	-6
0011	3	1011	-5
0100	4	1100	-4
0101	5	1101	-3
0110	6	1110	-2
0111	7	1111	-1

notice how the counting works: 1000 is -8… 1001 is -7… etc.

Two's Complement

- Let's check the "0 property": A + (-A) = 0
- Suppose A = 5

```
5 is 0101
negation of 5 is 2⁴ - 5 = 16 - 5 = 11
11(unsigned) is 1011
check the table: 1011 is -5 in 2's complement
now, let's try 5 + (-5) in 2's complement

  0101          5         1011            0111  (7)
+ 1011         -5       + 0000  (0)     + 0001  (1)
------       ----       ------          ------
1 0000          0         1011  (-5)      1000  (-8)
```

Two's Complement

- Negation: (2^8 – X) vs. (11111111 - X) + 1
- Note 2^8 needs 9 bits:
  - 2^8 is 256, from earlier conversion process: 1 0000 0000 = 1 * 2^8
- Whereas the other form has only 8 bits. Let's try it!
  - Consider X = 10 and we want to find -10

```
  1111 1111
– 0000 1010  (10d)
  ---------
  1111 0101 (–11d)
+         1
  ---------
  1111 0110 (–10d)
```

Oh, cool!
That's just flipping bits!

Two's Complement

- How to convert binary 2's complement number?
  - Same as before, except most significant bit is "sign"
- Consider an 8-bit 2's complement number
  - $b_0 \times 2^0 + b_1 \times 2^1 + b_2 \times 2^2 + \ldots + \mathbf{b_7 \times (-2^7)}$
- An example
  1011 0111
  = $1\times2^0 + 1\times2^1 + 1\times2^2 + 0\times2^3 + 1\times2^4 + 1\times2^5 + 0\times2^6 + \mathbf{1\times(-2^7)}$
  = 1 + 2 + 4 + 0 + 16 + 32 + 0 + **(-128)**
  = -73d
- What is 73d in 2's complement binary number?
- $v = (\sum (b_i \times 2^i)) + b_{K-1} \times -2^{K-1}$,
  where $0 \le i < K-1$, where K=# bits, i is bit posn

Summary

Code	Sign-Magnitude	1's Complement	2's Complement
000	+0	+0	+0
001	+1	+1	+1
010	+2	+2	+2
011	+3	+3	+3
100	-0	-3	-4
101	-1	-2	-3
110	-2	-1	-2
111	-3	-0	-1

- Issues
  - # of zeros
  - Balance
  - Arithmetic algorithm implementation