Introduction

- Rapidly changing field:
  - vacuum tube -> transistor -> IC -> VLSI
  - memory capacity and processor speed is doubling every 1.5 years:
- Things you'll be learning:
  - Foundation of computing, design methodologies, issues in design
  - how to analyze their performance (or how not to!)
- Why learn this stuff?
  - You want to design state-of-art system
  - you want to call yourself a "computer scientist or engineer"
  - you want to build software people use (need performance)
  - you need to make a decision or offer "expert" advice

1

What is a computing?

- In 1960, "computer" was still understood to be a *person*
  - A person who could compute
- By contrast, a recent dictionary begins the definition as
  - A "computer" is "An electronic machine..."
- But computing has had many abstraction
- We would learn about some of them today

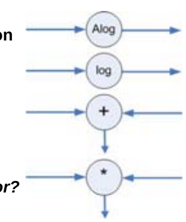
2

Consider An Example: Example 1

- Let us evaluate an expression
  - *A=B+C+D+E*F*
- *It can also be written as*
  - *A=(B+C)+D+E*F*
  - *A=(B+C+D)+E*F*
  - *A=(B+C+D)+(E*F)*
  - *A=B+(C+D)+E*F*
- *But are these correct?*
  - *A=(B+C+D+E)*F*
  - *A=B+C+(D+E*F)*
- *Depends on what are the rules for evaluating expressions*
- *What are we computing?*
- *What is the model?*

3

What is A Computing Abstraction?



- Consider computation a simple expression
  - A=B*C
- What do we need to do to compute?
  - *Need storage for B*
  - *Need storage for C*
  - *Multiply*
  - *Need storage for A*
  - *How would you do it on your calculator?*
- What if you do not have multiplier?
- But you have black boxes that compute, add, log/alog
  - Log A = Log B + Log C
- It is a functional transformation
- How do we achieve the computation? Put the blocks together

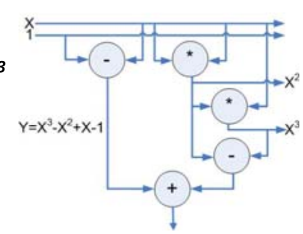
4

Consider Another Example: Example 2

- Consider the computation $Y = X^3 – X^2 + X - 1$
- How many operations?
  - *How many multiply?*
  - *How many adds/subs?*
  - *How many storage?*
- Is this the best we can do?
- How do we achieve efficiency is computation?

5

A Possible Solution: Example 2

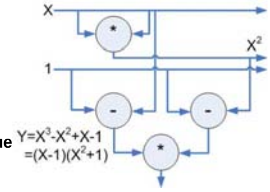


- How many operations?
  - *How many multiply? 2*
  - *How many adds/subs? 3*
  - *How many storage?*
  - *How much time?*
- Is this the best we can do?
  - For multiplication
  - Probably we can argue
  - What about adds/subs?
- This is not very efficient

6

Another Possible Solution: Example 2

- Simplify the function by factorization? $Y = X^3 - X^2 + X - 1 = (X-1)(X^2+1)$
  - *How many multiply? 2*
  - *How many adds/subs? 2*
  - *How many storage?*
  - *How much time?*
- Is this the best we can do?
  - For *, probably we can argue
  - What about adds/subs?



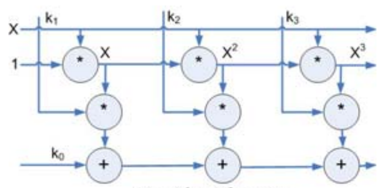
- Another factorization does not change number of operations

Consider One More Example: Example 3

- Consider the computation $Y = k_3 * X^3 + k_2 * X^2 + k_1 * X + k_0$
  - No constraints on values
- How many operations?
  - *How many multiply?*
  - *How many adds/subs?*
  - *How many storage?*
- Is this the best we can do?
- How do we achieve efficiency is computation?

A Possible Solution: Example 3

- How many operations?

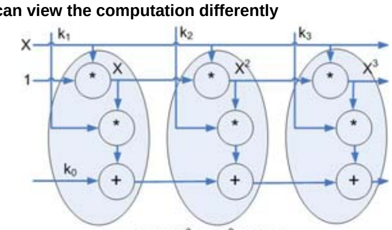


$Y = k_3 * X^3 + k_2 * X^2 + k_1 * X + k_0$

  - *How many multiply? 6, but one can be saved easily*
  - *How many adds/subs? 3*
  - *How many storage?*
  - *How much time?*

Another Way to Solution: Example 3

- We can view the computation differently

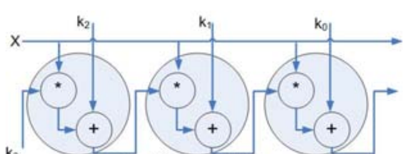


$Y = k_3 * X^3 + k_2 * X^2 + k_1 * X + k_0$

- Why this form?
  - Provides a building block for computation
- But has each block big, 4 inputs, 2 outputs

Yet Another Way to Solution: Example 3

- We can look at the expression differently



$Y = k_3 * X^3 + k_2 * X^2 + k_1 * X + k_0$
$= ((k_3 * X + k_2) * X + k_1) * X + k_0$

- How many operations?
- Why this form?
  - Provides a way to optimize and provides a building block

Point of Discussion

- A good computation structure require some thinking
- Optimize on hardware design cost
- Optimize on time for computation
- There may be a tradeoff that needs to be explored
- Identify common building blocks that can be implemented and used to realize interesting computations
- Always consider
  - How many operations?
  - How many time steps?
  - What is the tradeoff?
  - Solutions may not be obvious

Computing with a designed Machine

- Consider computation in example 1 (An user may like to directly say this as is)
  - A=B*C
- A given machine has facility to load variables and perform arithmetic and complex functions (who designed it?)
- So how do we compute?
- Here is a conceptual program
  - Load B, mem1
  - Load C, mem2
  - Multiply mem1, mem2, mem3
  - Store A, mem3
- On your simple calculator
  - Key in value of B
  - Press multiply
  - Key in value of C
  - Press = and Read A out

Program Example 2

- Required computation is $Y=X^3-X^2+X-1$
- A complex program may look like
  - X = value
  - $Y = X^3 – X^2 + X – 1$
- A simple program may look like
  - Load X, mem1
  - Multiply mem1, mem1, mem2
  - Multiple mem1, mem2, mem3
  - Sub mem3, mem2, mem4
  - Add mem4, mem1, mem5
  - Load #1, mem6
  - Sub mem5, mem6, mem7
  - Store Y, mem7
- Do we need all these memory locations?

Program Example 2 Differently

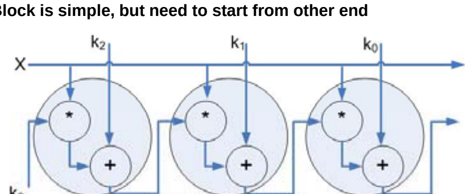
- Factorized function is $Y=X^3-X^2+X-1$ $=(X-1)(X^2+1)$
- A simple program may look like

Now Consider Our Complex Example – 3

- Required computation is $Y=k_3*X^3+k_2*X^2+k_1*X+k_0$
- How do we approach this
- We need some structure to store variables
  - An array structure k[i], i = 0, 1, 2, 3, ...
  - A variable name X
  - Store powers of X, i.e., X^i in xpower[i], i = 0, 1, 2, 3, …
  - A result location Y
  - X is given by user
  - K[i] is filled in by user
  - Y is initially zero
  - Partial Y computation is, Y = k[0]
  - Also, xpower[0] = 1
  - At each step i = 1, 2, 3, …, we have 3 inputs and 2 outputs
    - we take xpower[i-1], partial result Y, and k[i]
    - And compute xpower[i] and a new partial result Y

Now Program Example – 3 Using Alternate

- What is the big difference?
- Block is simple, but need to start from other end



$Y=k_3*X^3-k_2*X^2+k_1*X-k_0$
$=((k_3*X+k_2)*X+k_1)*X+k_0$

Differences Between the two Programs

- First approaches computes a 3-inputs, 2-outputs function
- The second one uses a 3-input, 1-output function
  - Mathematically that is how we prefer to write functions
- First method can be used for successive addition of term
- The second method requires us to know how many terms

Computing Functions: Difference Engine

- Consider the computation $Y = X^3 - X^2 + X - 1$
- Consider the table
- What is going on each row
- Can you name each row?
- Can you tell how an entry in a row is computed?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	
-1	0	5	20	51	104	185	300	455	656	909	1220	1595	2040	
1	5	15	31	53	81	115	155	201	253	311	375	445		
4	10	16	22	28	34	40	46	52	58	64	70			
6	6	6	6	6	6	6	6	6	6	6				

Difference Engine Abstraction

- Suppose you want to calculate y = Sin (x)
- Need a Sin calculator
  - Looks cheap on your calculator, it is expensive computation
  - How would you go about it?
- Consider a Taylor series expansion
  - $y = \sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots\ldots$
- Based on computing differences, a finite n-th order polynomial can be differentiated n times, which can be represented by a difference
- What degree polynomial is sufficient?
  - Depends on accuracy needed (we will visit that many times)
- Let us consider only two terms:
  - $y = \sin(x) = x - x^3/3!$

Calculating using Difference Engine

- To compute value of sin(x) at x(0), x(1), x(2), x(3), x(4), x(5), ……… such that difference in two consecutive values of x is small
  - $\Delta x = x(i+1) - x(i)$
  - $y(x(i)) = \sin(x(i)) = x(i) - x(i)^3/3!$
- For simplicity, we will drop () and denote the corresponding values of y also as y0, y1, y2, y3, …..
- We can calculate y0, y1, y2, and y3 by hand and also call them Δ^0y0, Δ^0y1, Δ^0y2, and Δ^0y3, respectively
- Why are we doing it?
- That forms the basis of difference engine abstraction

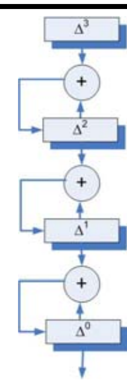
Difference Engine (cond.)

- If we differentiate the function, forth differentiation will yield a 0
- What about the third differentiation?
  - A constant (value is -1 in this case)
  - And others can be calculated as well
- First order difference can be written as
  - Δ^1y0 = y1-y0; Δ^1y1 = y2-y1; Δ^1y2 = y3-y2
- Second order difference can be written as
  - Δ^2y0 = Δ^1y1 - Δ^1y0 = y2-2y1+y0
  - Δ^2y1 = Δ^1y2 – Δ^1y1 = y3-2y2+y1
- Third order difference can be written as
  - Δ^3y0 = Δ^2y1 - Δ^2y0 = y3-3y2+3y1-y0
- And the forth order difference is Δ^4y0 = 0
- Suppose we know Δ^3y0, Δ^2y0, Δ^1y0, and Δ^0y0
- Using this we can recursively compute Δ^3y1, Δ^2y1, and Δ^1y1, and Δ^0y1
- And then all y2 and y3, and y4………

Difference Engine Example

- IN: x0 x1 x2 x3 x4 x5 x6
- OUT: y0 y1 y2 y3 y4 y5 y67
- 0^{th} Diff: Δ^0y0 Δ^0y1 Δ^0y2 Δ^0y3 Δ^0y4 Δ^0y5 Δ^0y6
- 1^{st} Diff: Δ^1y0 Δ^1y1 Δ^1y2 Δ^1y3 Δ^1y4 Δ^1y5 Δ^1y6
- 2^{nd} Diff: Δ^2y0 Δ^2y1 Δ^2y2 Δ^2y3 Δ^2y4 Δ^2y5 Δ^2y6
- 3^{rd} Diff: Δ^3y0 Δ^3y1 Δ^3y2 Δ^3y3 Δ^3y4 Δ^3y5 Δ^3y6
- In general
  - $\Delta^n y(i+1) = \Delta^n y(i)$ for nth order function and
  - $\Delta^{j+1} y(i) = \Delta^j y(i+1) - \Delta^j y(i)$ for j = 0, 1, 2, … n-1, and i = 0, 1, 2, …
  - Or $\Delta^j y(i+1) = \Delta^j y(i) + \Delta^{j+1} y(i)$ for j = 0, 1, 2, … n-1
- So if we know the values in the first column, we can compute second column and so on
- The structure need n+1 memories (to store a column) and n adders
- One can also write a C program to compute a column at a time
  - And the first column is obtained by calculating values by hand

Difference Engine Organization



Decimal System

- We are all familiar with decimal numbers
- Consider a number 2375
- What digits representing thousand, hundred, ten and one's place
- How did you get it?
- Give me an algorithm
  - Divide by 1000, result is thousand place value
  - Subtract 1000*thousand place value
  - Divide by 100, result is hundred place value
  - Subtract 100*hundred place value
  - Divide by 10, result is ten place value
  - Subtract 10*ten place value
  - Remainder is one place value
- What is good about this algorithm
- What is bad about it?

An Easier Algorithm

- Divide by 10
- Remainder is one place value
- Divide the result by 10
- Remainder is ten place value
- Divide the result by 10
- Remainder is hundred place value
- Divide the result by 10
- Remainder is thousand place value

- Any time result is zero, that means no more value

- Division is always by 10

- We always need result and remainder

Any Base b Algorithm

- Divide by b
- Remainder is one place value
- Divide the result by b
- Remainder is ten place value
- Divide the result by b
- Remainder is hundred place value
- Divide the result by b
- Remainder is thousand place value

- Any time result is zero, that means no more value

- Division is always by b

- Remainder is always between 0 and b-1

Information Representation

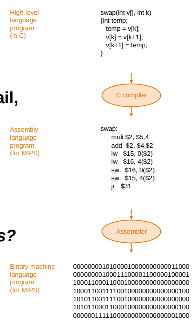
- Information theory: discusses how to deal with information
- We only deal with some aspects of it
- Virtually all computers now store information in binary form
- A binary number system has two digits, 0 and 1
- Combination of binary digits represent various kind of information
- Examples
  - 01001011
  - It can be interpreted as an integer value, a character code, a floating point number….
- Non binary numbers are also possible
- How do we represent negative numbers?
  i.e., which bit patterns will represent which numbers?

Abstraction

- Delving into the depths reveals more information

- An abstraction omits unneeded detail, helps us cope with complexity

What are some of the details that appear in these familiar abstractions?



Historical Perspective

- 1642 Pascal: Mechanical Computer
- 1671: Gottfried Leibniz ADD/SUB/MUL/DIV
- 1801: Automatic Control of Weaving Process
- 1827 The Difference Engine by Charles Babbage
- 1936: Zuse Z1: electromechanical computers
- 1941: Zuse Z2
- 1943: Zuse Z3
- 1944: Aiken: Ark 1 at Harvard
- 1942-45: ABC at Iowa State (Atanasoff-Berry Computer)
- 1946: ENIAC: Eckert and Mauchley: Vacuum Tube
- 1945 EDVAC by von-Neumann machine, father of modern computing

Why Binary?

- Easy to represent
  - Off and On
  - Open and close switch
  - Head and tail on a coin
  - Polarity of magnetization
  - 0 and nonzero voltage levels
- How to represent information in binary?
- Say we want to represent positive number 0 and 1
  - 0 is 0 and 1 is 1
- say we want to represent red and green colors
  - 0 is red and 1 is green (or vice versa)
- Say we want to represent fall and spring semesters
  - 0 is fall and 1 is spring (or vice versa)

More Complicated Examples

- Numbers 0 to 7
  - We use combination of digits
    - 1 digits gives us two combination
    - 2 will yield four
    - 3 will yield 8
  - Need three bits (binary digits)
- What if we want to represent 16 alphabets - Need four bits
- What if we want to represents numbers from 11 to 25?
- Homework Problem:
  - For each part below devise a scheme to represent, in binary, each set of symbols
    - (A) Numbers: 0, 1, 2, 3, 4, 5, 6, 7
    - (B) Alphabets: A, B, C, D, E, F
    - (C) Integers from 21 to 36

Bits and Combinations

# of Bits	# of quantities
1	2
2	4
3	8
4	16
..	..
..	..
..	..
n	2^n

- What happens in other number systems?
- In base b, n digits give b^n combinations
- Base 10: decimal
- Base 8: Octal
- Base 16: Hexadecimal

Representation of Positive Numbers

- Positional value
- Binary digits are numbered
- Right most digit is 0
- Next to that is a 1
- And so on up to n-1 in an n-bit representation
- Decimal point is implied at the right of bit 0
- Each bit is assigned a weight
- The weight of i^{th} bit is 2^i
- Using this notation
  - The value of an n bit sequence is
  - $2^{n-1} x_{n-1} + 2^{n-2} x_{n-2} + \ldots + 2^1 x_1 + 2^0 x_0$
  - $= \sum_{i=0}^{i=n-1} 2^i x_i$

x x x x x x x x

7 6 5 4 3 2 1 0

Bit #	Weight
0	2^0
1	2^1
2	2^2
3	2^3

Some Examples

- Convert 0101 into decimal
  - Position: 3 2 1 0
  - Weight: 8 4 2 1
  - Digits: 0 1 0 1
  - Decimal value: 8 * 0 + 4 * 1 + 2 * 0 + 1 * 1 = 5
- Convert 10110101 into decimal
  - Position: 7 6 5 4 3 2 1 0
  - Weight: 128 64 32 16 8 4 2 1
  - Digits: 1 0 1 1 0 1 0 1
  - Decimal value: 128*1+64*0+32*1+16*1+8*0+4*1+2*0+1*1=181
- Now try 10000000
- And try 01111111

And What About the Reverse Operation?

- First see the largest weight of a binary positional digit contained in the number
- Put that binary digit = 1 and subtract weight
- Then try subtracting the next bit's weight
- If successful
  - next bit is 1, else next bit is 0 (and restore the value)
- Repeat the last two steps until done
- Convert decimal number 181 into binary
- Largest weight is 128, subtract 128 and set bit 7 = 1
- Try subtracting 64 out of remainder 53 (181-128)
- No successful, so the next digit is 0
- Try weight 32, 16, 8, 4, 2, and 1 successively
- Number is 1 0 1 1 0 1 0 1

A Simpler Method

- Convert decimal number 181 into binary
- Start dividing by 2
- Successive remainders are digits from right
- 181/2 = 90 remainder 1
- 90/2 = 45 remainder 0
- 45/2 = 22 remainder 1
- 22/2 = 11 remainder 0
- 11/2 = 5 remainder 1
- 5/2 = 2 remainder 1
- 2/2 = 1 remainder 0
- 1/2 = 0 remainder 1
- Number is 1 0 1 1 0 1 0 1

And Now Try Some Problems

- Suppose you want to represent positive integers in binary.
- Indicate how many bits are required to represent each of the following sets of integers:
  – (1) The integers from 0 to 127 inclusive
  – (2) The integers from 0 to 2,048 inclusive
  – (3) The integers from 0 to 32,500 inclusive
  – (4) The integers from 0 to 1,500,345 inclusive
- Indicate how large a value can be represented by each of the binary quantities: A (1) 4-bit, (2) 12-bit, and (3) 24-bit quantity.
- Convert each of the following binary digits into decimal. Assume these quantities represent unsigned integers.
  – (1) 1010; (2) 10010; (3) 0111110; (4) 10000000; (5) 0111111
- Convert each of the following decimal numbers into binary.
  – (1) 6; (2) 13; (3) 111; (4) 147; (5) 511

Base 'b' number

- In general a number system can have any base b
- the digit used are 0, 1, … , b-1
- The weight of i^{th} place is b^i
- The conversion formula from base b into decimal number is

$$\sum_{i=0}^{i=n-1} b^i x_i \qquad \text{for i = 0 to n – 1} \qquad \text{for an n digit quantity}$$

- Commonly used base are 2, 3, 8, 10, 16, ...

Bases 2, 8, and 16 are related

Binary	Decimal	Octal	Hexadecimal
0000	00	00	0
0001	01	01	1
0010	02	02	2
0011	03	03	3
0100	04	04	4
0101	05	05	5
0110	06	06	6
0111	07	07	7
1000	08	10	8
1001	09	11	9
1010	10	12	A
1011	11	13	B
1100	12	14	C
1101	13	15	D
1110	14	16	E
1111	15	17	F

Conversion

- From binary to octal
  – make groups of 3 bits from right to left

  $01\ 110\ 110_2 \Rightarrow 166_8$
- From octal to binary
  – make each digit as 3 bits sequence

  $276_8 \Rightarrow 010\ 111\ 110_2$
- From binary to hexadecimal
  – make groups of 4 bits from right to left

  $0111\ 0110_2 \Rightarrow 76_{16}$
- From hexadecimal to binary
  – make each digit as 4 bits sequence

  $37_{16} \Rightarrow 0011\ 0111_2$

Signed numbers

- Positive numbers are well understood
- An n-bit number represents numbers from 0 to 2^n-1
- n+m bits can be used to represent n-bit integer and m-bit fraction of a number
- However negative numbers cause another problem
- In all solutions, one bit is needed to represent the sign, + or -
- MSB can be used for that purpose, i.e., represent sign
- Remaining bits can be interpreted differently
  – They can represent magnitude as a positive number
  – They can be complemented (represent 0 by 1 and 1 by 0)
  – Or manipulate in some other way

Example

Consider the bit string 1010:

- Sign and Magnitude
  - 1 010
    -ve 2
  - So it represents -2
- 1's Complement
  - 1 010 → 1 101
    -ve 5
  - So it represents -5
- 2's Complement
  - 1 010 → 1 101 + 1
    -ve 6
  - So it represents -6

Interpretation

- Sign and Magnitude
  - Out of n bits, one is reserved for sign
  - Remaining bits represent the value of number as positive
  - It is equivalent of representing it as $(1 - 2x_{n-1})\sum_{i=0}^{n-2} 2^i x_i$
- 1's Complement
  - Convert the magnitude of number as a binary string
  - Then complement every bit (replace 1 by 0 and 0 by 1)
  - This is equivalent of having the weight of MSB as $-(2^{n-1}-1)$
- 2's Complement
  - Convert the magnitude of number as a binary string
  - Complement every bit (replace 1 by 0 and 0 by 1) and add 1
  - This is equivalent of having the weight of MSB as -2^{n-1}

Sign Magnitude, 1's, and 2's complement

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

Maximum and Minimum values in n-bits

- We use 2's complement as it makes add/sub)simple
- n-bits uses only n-1 bits to store the value
- Largest positive value is $2^{n-1}-1$
- Largest negative value is -2^{n-1}
- For n=4, these values are from + and -
- For n=8, these values are from + and -
- If we need larger or smaller values to be stored, we have problem -- leads to overflow and underflow

- For MULT/DIV, sign and magnitude is better
  - But we cannot keep switching

Negation

- To change sign of a number
- In Sign and Magnitude
  - Just complement the sign
- 1's Complement
  - Complement all bits
- 2's Complement
  - Complement all bits and add 1
- Adding 1 is expensive operation (Example: Add 1 to 0111)
- Alternate 2's complement method
  - Scan the string from right
  - Retain all bits up to the first 1
  - Then complement the remaining bits

Example:
6 = 0110
-6 = 1010

Negation Examples

- Negate the following 4-bit 2's Complement Binary Values:

0011	1111	0111	1010
1100+1	0000+1	1000+1	0101+1
-> 1101	-> 0001	-> 1001	-> 0110

- What is the negation of 1000 in 4-bit 2's complement?

Converting negative number to Binary

- Convert a negative decimal number to binary in 2's complement
- Method 1:
  - Convert the magnitude to an n-bit string
  - Negate the number
  - Example: -5   Magnitude in binary: 0101   Negation: 0011
- Method 2:
  - The magnitude of number must be less than or equal to 2^{n-1}
  - Add 2^n to the number
  - Convert this number as an n-bit unsigned integer
  - Example: -4 + (16) = 12 (decimal) = 1100 (binary)
    -7 + (16) = 9 (decimal) = 1001 (binary)

Computer Arithmetic for one bit

- ADD and SUB are fundamental
- Adding one digit to another gives result (R) and carry (C) bit
- Subtracting a digit from another gives result (R) and borrow (B)
- Examples of adding/subtracting two digits

X	0	0	1	1	X	0	0	1	1
Y	+0	+1	+0	+1	Y	-0	-1	-0	-1
R	0	1	1	0	R	0	1	1	0
C	0	0	0	1	B	0	1	0	0

- Add/sub of two digits with carry/borrow also gives two digits
- That is adding/subtracting two digits with carry/borrow

Previous →	C	1	1	1	1	B	1	1	1	1
	X	0	0	1	1	X	0	0	1	1
	Y	+0	+1	+0	+1	Y	-0	-1	-0	-1
	R	1	0	0	1	R	1	0	0	1
Current →	C	0	1	1	1	B	1	1	0	1

ADD/SUB with more than one bit

- Follow rules of decimal arithmetic
- Add carry to/sub borrow from the next digit
- In 2's complement, if we simply add or subtract without regard to sign, we get correct result if there is no overflow/underflow
- Overflow/Underflow occurs when the carry into and the carry out of the sign bit position are different.
- Examples

C/B	00010		01000		11010		10000	
X	0101	0101	0101	1001	0010	1011	0101	1011
Y	+0001	+1011	+0100	+1010	-0101	-1001	-1101	-0100
Res	0110		1001		1101		1000	
	Corr	Corr	Over	Under	Corr	Corr	Over	Under

ADD/SUB revisited

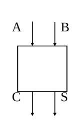
- Understand the examples again
- Overflow
  - When two positive numbers added together or a negative number subtracted from a positive number yields negative
- Underflow
  - When two negative numbers added together or a positive number subtracted from a negative number yields positive

C/B	00010	11110	01000	10000	11010	00000	10000	01000
X	0101	0101	0101	1001	0010	1011	0101	1011
Y	+0001	+1011	+0100	+1010	-0101	-1001	-1101	-0100
Res	0110	0000	1001	0011	1101	0010	1000	0111
	Corr	Corr	Over	Under	Corr	Corr	Over	Under

Using 1-bit building blocks to make n-bit circuit

- Design a 1-bit circuit with proper "glue logic" to use it for n-bits
  - It is called a bit slice
  - The basic idea of bit slicing is to design a 1-bit circuit and then piece together n of these to get an n-bit component
- Example:
- A half-adder adds two 1-bit inputs
- Two half adders can be used to add 3 bits
- A 3-bit adder is a full adder
- A full adder can be a bit slice to construct an n-bit adder

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Half Adder

Full adder and multi-bit ripple-carry adder

- Two half adders can be used to add 3 bits
- n-bit adder can be built by full adders
- n can be arbitrary large