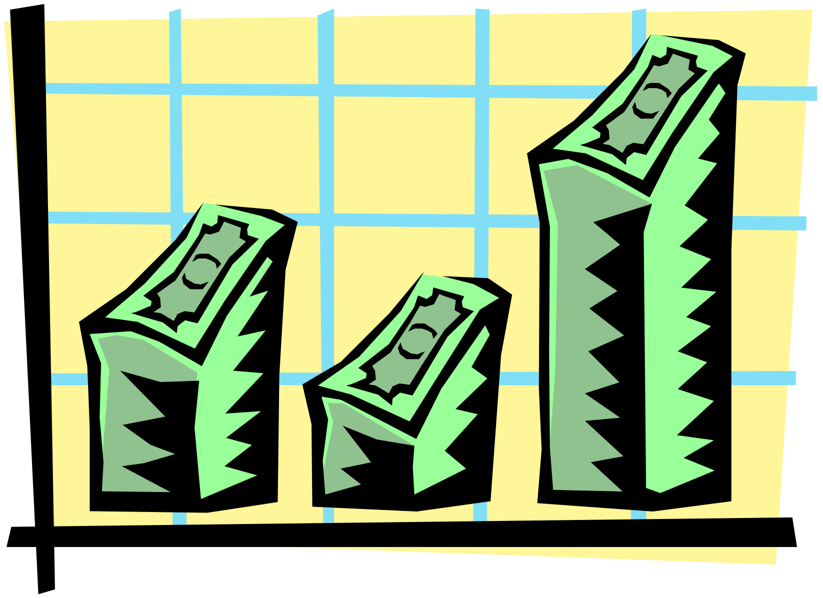
# Computer Arithmetic

- ALU design
- Multiplication – Booth Encoding
- Floating-Point Numbers



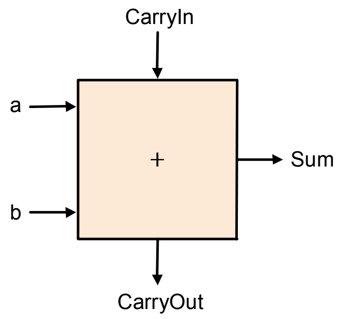
## Full Adder

review

Recall

Adding 2's Complement Binary Numbers

$$\begin{array}{l} P_3\,P_2\,\,P_1\,P_0 \\ Y_3\,Y_2\,Y_1Y_0 \\ +\text{------------} \end{array}$$



$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$

$sum = a \text{ xor } b \text{ xor } c_{in}$

**Note 2-gate delays**

- N-bit ripple-carry adder: a series connection of n FA's
- Sign-extension: repeating sign bit to fill up higher bits,
  - e.g. 16 to 32bits
- Overflow:

  Carry into MSB(Most Significant Bit) is a sign-bit

  Carry out of MSB is also a sign-bit for (N+1) bit representation

  They should be the same, otherwise OVERFLOW.

## Multiplication

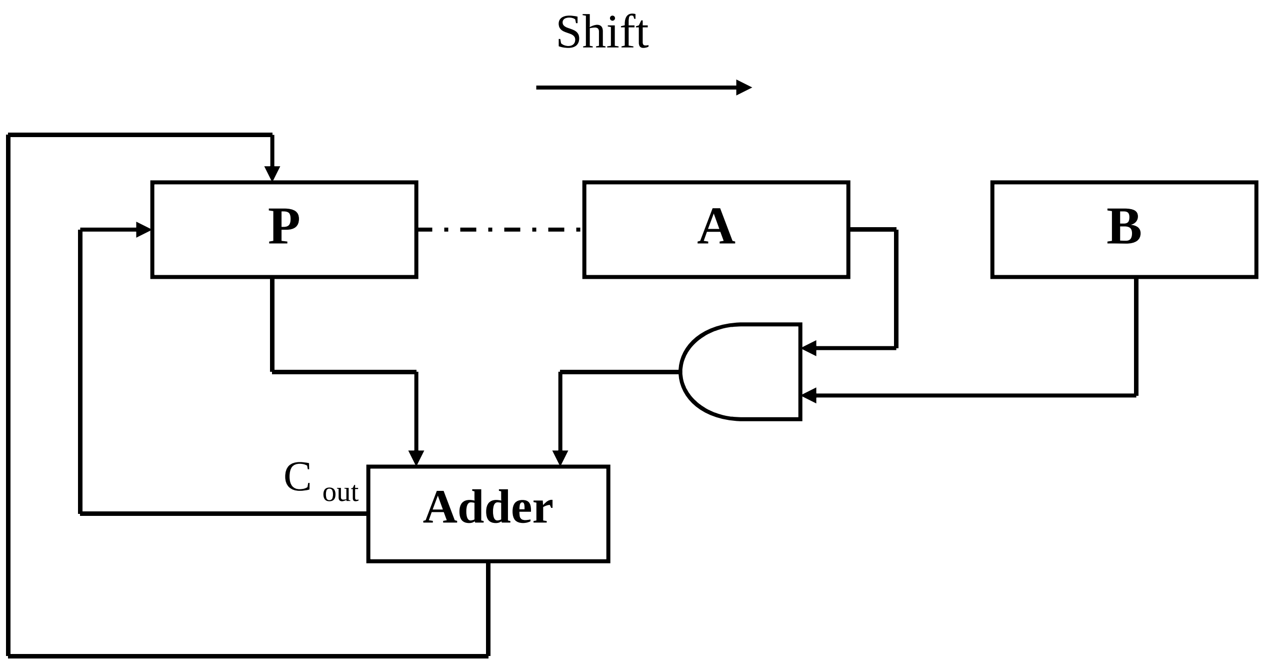
- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at a version based on grade-school algorithm

```
     0010   (multiplicand)
___x_1011   (multiplier)
```

- Negative numbers:  convert and multiply

## Unsigned Integer Multiplication

- Add shift.                B x A => P || A.
- LSB of A = 1 $\rightarrow$ P = P + B.
- Shift right   Carry & P & A   by 1 bit.



# Unsigned Integer Multiplication - Example

- 0010x1011: BxA=P

```
     P         A        B
     0000      1011     0010
   + 0010
     0010      1011 (and then shift right)
     0001      0101
   + 0010
     0011      0101 (and then shift right)
     0001      1010
   + 0000
     0001      1010 (and then shift right)
     0000      1101
   + 0010
     0010      1101 (and then shift right)
     0001      0110     final answer in P||A.
```

```
      0010
x     1011
----------
      0010
     0010
    0000
   0010
-----------
   0010110
```

## Unsigned Integer Multiplication

- How about negative number multiplication?

$\rightarrow$ X . Y ; |X| . |Y|. g1

Check sign(X) $\neq$ sign(Y) $\rightarrow$ - |X| . |Y|.

- Checking sign is overhead!

g1 just do the previous slide algorithm with 2's complement, then it's OK

Booth is to reduce overhead not just sign bit check but also reducing additions.
ghlee, 2015-03-31

## Booth's Algorithm for 2's Complement. Numbers

- Consider $Y \cdot X$ , where X = 0111

Since, 0111 = 1000 – 0001,

$Y \cdot X = Y \cdot (1000) - Y \cdot (0001)$

- Subtract Y and Add 8 times Y (i.e. shift 3 times) . This req. 1 add, 1 sub. & 4 shifts.
- Recording the above 0111 = 1000 – 0001= 100 $\bar{\mathbf{1}}$ is called **Booth Recording** (in radix 2)

## Booth Recording

- Let $X = X_{n-1} \cdots X_0$ for Y. X

At i th step of multiplication , $( X_{-1} = 0)$

| $X_I$ | $X_{I-1}$ | Recording | Algorithm | Note |
|---|---|---|---|---|
| 1 | 0 | $\bar{1}$ | -Y & shift | Beginning of 1's |
| 1 | 1 | 0 | Shift | Middle of 1's |
| 0 | 1 | 1 | +Y & shift | End of 1's |
| 0 | 0 | 0 | shift | Middle of 0's |

## Example

Y. X $\rightarrow$ P || X   with Y = -5 ,  X = -6   in 4 bit

| P | X | |
|---|---|---|
| 0 0 0 0 | 1 0 1 $\underline{0\ 0}$ | ; $X_0 = X_{-1} = 0$ ( shift right ) |
| 0 0 0 0 | 0 1 0 $\underline{1\ 0}$ | ; $X_1 = 1 \neq X_0 = 0$ ( subtract) |
| 0 1 0 1 | 0 1 0 1 | ; & shift right |
| 0 0 1 0 | 1 0 1 $\underline{0\ 1}$ | ; $X_2 = 0$ , $X_1 = 1$ : Add |
| 1 0 1 1 | | |
| + ------------------------ | | |
| 1 1 0 1 | 1 0 1 0 | ; shift right |
| 1 1 1 0 | 1 1 0 $\underline{1\ 0}$ | ; $X_3 = 1$ ,  $X_2$ =0 : (subtract) |
| 0 0 1 1 | 1 1 0 1 | ; & shift right |
| 0 0 0 1 | 1 1 1 0 | |
| -------------------------- | | ; 30 = -5 x -6 |

## Example

Y = 5 , X = -6 in 4 bit

| P | X | |
|---|---|---|
| **0 0 0 0** | **1 0 1 $\underline{0\ 0}$** | **; $X_0 = X_{-1} = 0$ ( shift right )** |
| **0 0 0 0** | **0 1 0 $\underline{1\ 0}$** | **; $X_1 = 1 \neq X_0 = 0$ ( subtract)** |
| **1 0 1 1** | **0 1 0 1** | **; & shift right (with sign-bit)** |
| **1 1 0 1** | **1 0 1 $\underline{0\ 1}$** | **; $X_2 = 0$ , $X_1 = 1$ : Add** |
| **0 1 0 1** | | |
| **+ ------------------------** | | |
| **0 0 1 0** | **1 0 1 0** | **; shift right** |
| **0 0 0 1** | **0 1 0 $\underline{1\ 0}$** | **; $X_3 = 1$ , $X_2 =0$ : (subtract)** |
| **1 0 1 1** | **0 1 0 1** | **;** |
| **1 1 0 0** | **0 1 0 1** | **; shift right** |
| **--------------------------** | | |
| **1 1 1 0** | **0 0 1 0** | **; -30 = 5 x -6** |

## Booth Recording

- Booth recording takes care of negative multiplier
- In the example
- X=-6=1010 $\longrightarrow$ $\bar{1}1\ \bar{1}0$

Subtract 2 times

Add 4 times

Subtract 8 times

Note

Hope to have less add/sub:

alternating 0 and 1 makes worst case g2

## 슬라이드 11

| | |
|---|---|
| g2 | so -6=1010 is the worst case!<br>ghlee, 2015-03-31 |

## Faster Array Multiplier

Instead of doing the adding partial products linearly,
i.e. one after another partial product,
can be done in parallel by grouping partial products

Each of group has three partial products

Wallace Tree Multiplier

Each group of two partial products

Binary Tree Multiplier

$$\begin{array}{cccccccc}
 & & & & m^0{}_3 & m^0{}_2 & m^0{}_1 & m^0{}_0 \\
 & & & m^1{}_3 & m^1{}_2 & m^1{}_1 & m^1{}_0 & \\
 & & m^2{}_3 & m^2{}_2 & m^2{}_1 & m^2{}_0 & & \\
 & m^3{}_3 & m^3{}_2 & m^3{}_1 & m^3{}_0 & & & \\
+ & & & & & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}$$

## Division

- Can be done in reverse of multiplication of shift and add

  shift-left and subtract

Faster Division uses

- the fact x/y = x*(1/y) iterative guess of 1/y
- multiple quotient bits with precalculated table: SRT division

read Text 3.4(p.178~182)

## Floating Point Numbers

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., 31,557,600,000,000
- Normalized Representation:
  - No leading zeros and one significant digit before decimal point: (sign)m.xyz…. X $10^e$

e.g. +3.15576 x $10^9$
sign: +
significand: 3.15576
exponent: 9
number base: 10, i.e decimal

# Floating Point  Numbers

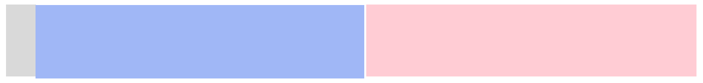
For computer floating point:

Number base: 2 ($2^n$)

Sign: 0 (+) or 1(-)

Exponent: actual exponent + bias

Significand:   binary number (with Sign bit)



$$(-1)^{sign} \bullet \text{ significand} \bullet 2^{exponent}$$

- o  bias : to make smallest exponent 0
  comparing exponents and checking 0 more
  convenient with biased representation
- o more bits for significand gives more accuracy
- o more bits for exponent increases range
- o Mantissa (or Fraction): normalized with no significant

digit left to the decimal point $0.315576 \times 10^{10}$

## Floating Point Numbers

- $(-1)^{\text{sign}} x (\text{significand}) x 2^{\text{exponent}}$
- Issues
  - Format : which is which and how many bits, number base ($2^n$)
  - Handling of exceptions
    - Overflow, Underflow, very small numbers
  - Rounding
    - How to round up numbers?

IEEE 754 floating point standard

## IEEE 754 floating point standard

- single precision:  8 bit exponent, 24 bit significand (23+1)
- double precision:  11 bit exponent, 53 bit significand (52+1)
- Do not represent 1 before the binary point: It's always 1!
- Exponent Bias: to make smallest negative exponent 0
  - 127 for single precision; 0 ~ 255 with 255 (all 1's) reserved
  - 1023 for double precision
    - all 1's for exponent: reserved for special cases

Note: some specials in IEEE754

- zero: zero significand and zero exponent
- 0/0: NaN (nozero significand, all 1's for exponent)
- n/0: infinity (zero significand, all 1's for exponent)

- Denormal: close to zero (nonzero significand with 0 exponent)
  - Without denormal, more possibility of Underflow

See Figure 3.13 Text p.187

# IEEE 754 floating-point standard

- Example:
  - decimal: -.75 = -3/4 = $-3/2^2$
  - binary: -.11 = $-1.1 \times 2^{-1}$
  - floating point: exponent = -1 + bias (= 127) = 126 = 01111110
  - IEEE single precision:

```
1 01111110 10000000000000000000000
↑    ↑            ↑
sign exponent(w.bias) fraction(+1=significand)
```

# Floating Point Complexities

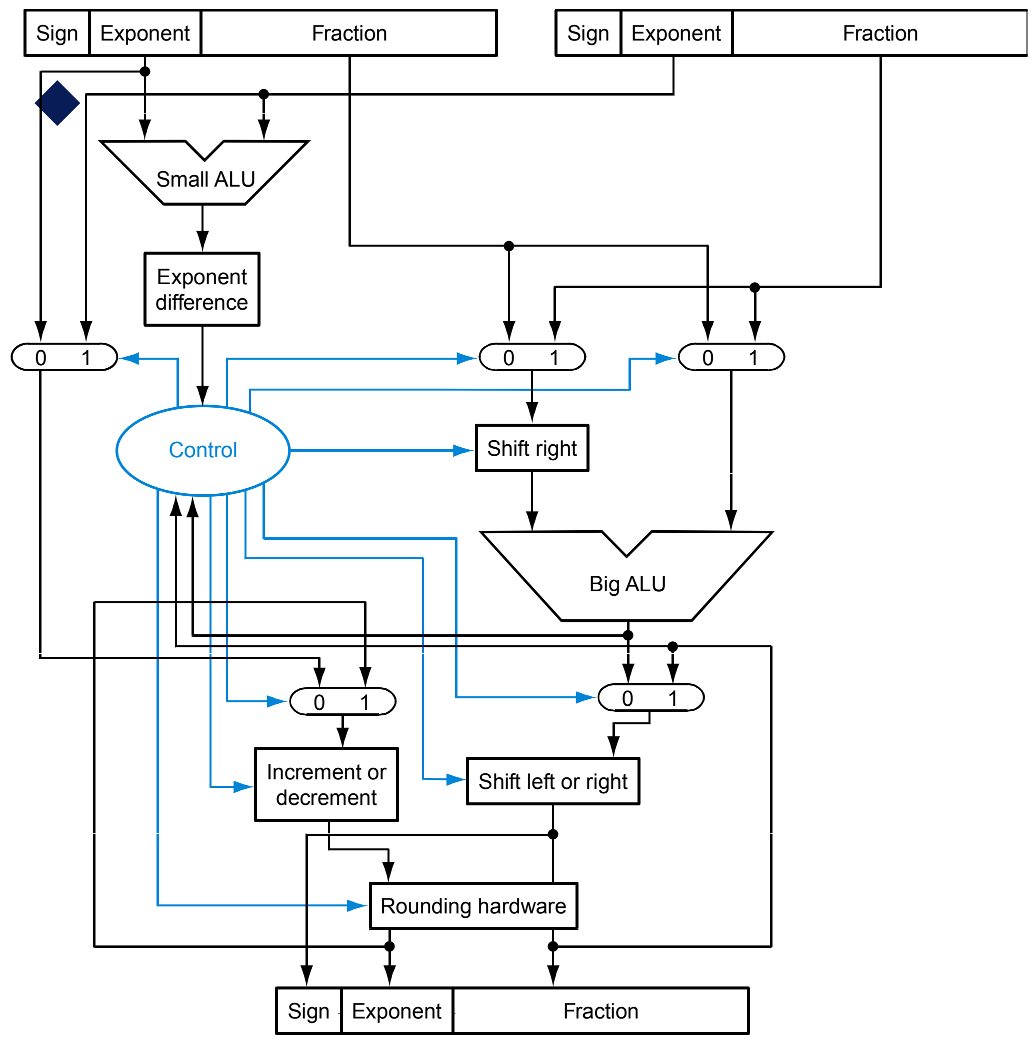
- Operations are somewhat more complicated
- Needs to compare exponents and align them by shifting significand
- Needs to work on significand and exponent separately
- Needs to normalize the result for correct representation

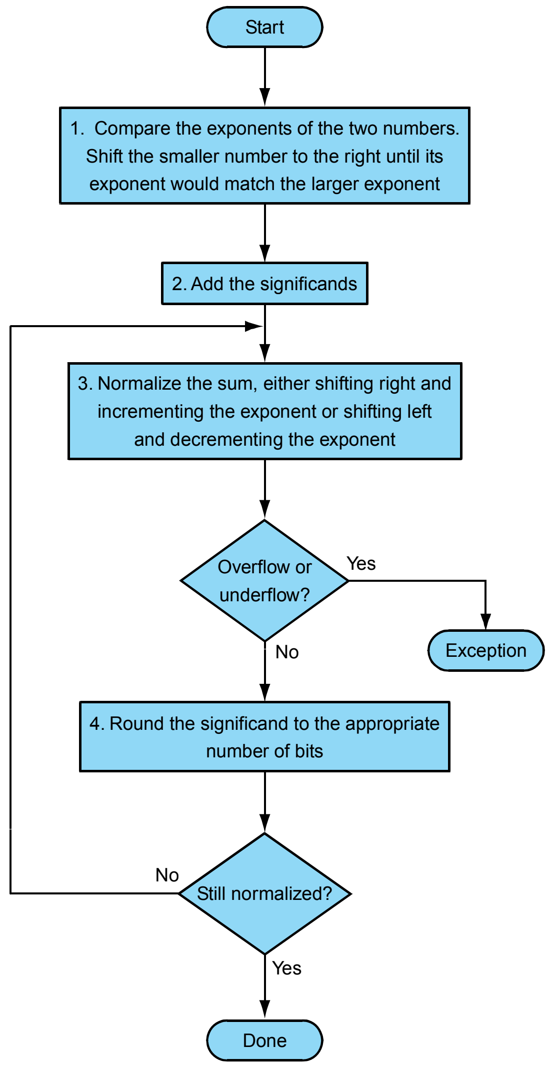
e.g. (read Text p. 198)
Add 0.5 and –0.4375 in binary Floating Point with 4-bit significand

$0.5 = 1.000 \times 2^{-1}$
$-0.4375 = -7/16 = -1.110 \times 2^{-2}$

1. Shift lesser exponent operand: $-0.111 \times 2^{-1}$
2. Add significands: $1.000 - 0.111 = 0.001$
3. Normalize $\underline{1.000 \times 2^{-3}} \times 2^{-1} = 1.000 \times 2^{-4}$

# Floating point addition




Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

## Floating Point Complexities

- In addition to overflow we can have "underflow"
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round – representing two additional bits for better precision plus one sticky bit (the bit moved out when shifted for exponent alignment) representing any non-zero bit beyond the precision (read Text p.206&208)
  - four rounding modes: up/down, truncate, nearest even.
  - positive divided by zero yields "infinity"
  - zero divide by zero yields "not a number"
  - other complexities

0.5
0.51
0.501
0.5000003

- Implementing the standard can be tricky
  - see text for description of 80x86 and Pentium bug!

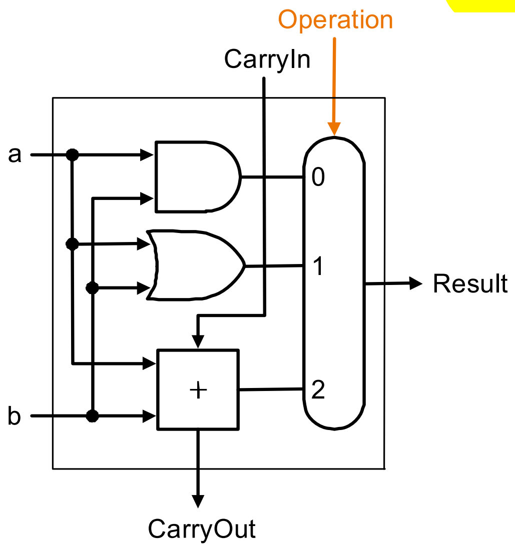
## MIPS ALU

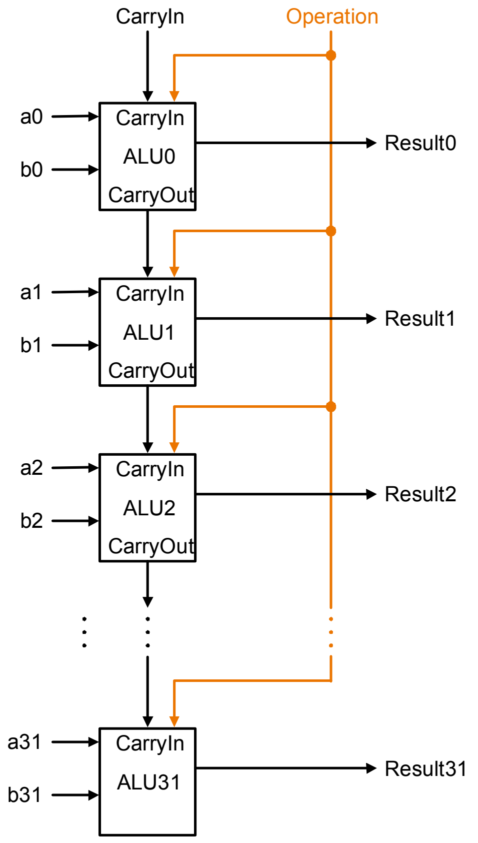
- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")
- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)
  - we'll look at examples for addition and multiplication

# Building a 32 bit ALU

Support AND, OR, Add/Sub

Key: Mux





## Tailoring the ALU to the MIPS

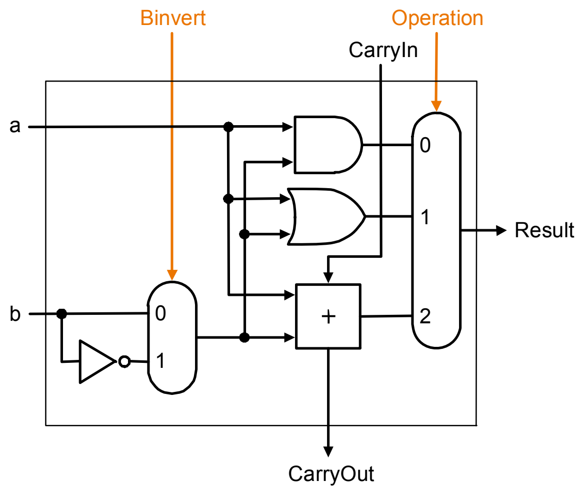
overview

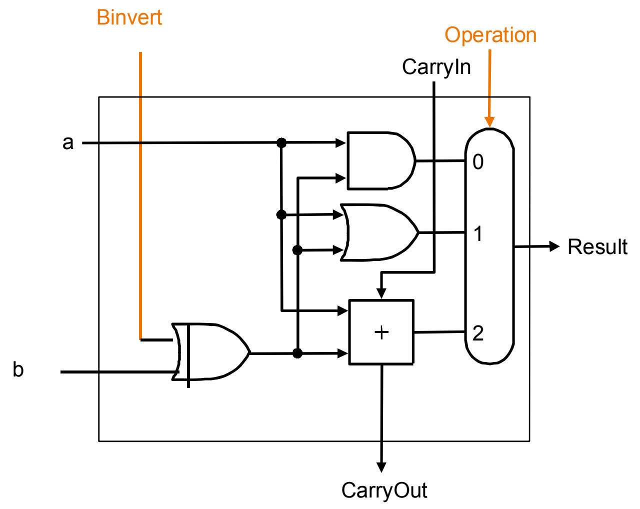
- Adding more and select through Mux per opcode
  - Need to support test for equality (beq \$t5, \$t6, 0x100)
    - use subtraction: (a-b) = 0 implies a = b

| Name | Format | Example | | | | | | Comments |
|---|---|---|---|---|---|---|---|---|
| | | 6bits | 5bits | 5 bits | 5 bits | 5bits | 6bits | |
| add | R | 0 | 2 | 3 | 1 | 0 | 32 | add \$1,\$2,\$3 |
| sub | R | 0 | 2 | 3 | 1 | 0 | 34 | sub \$1,\$2,\$3 |
| addi | I | 8 | 2 | 1 | 100 | | | addi \$1,\$2,100 |
| addu | R | 0 | 2 | 3 | 1 | 0 | 35 | subu \$1,\$2, \$3 |
| and | R | 0 | 2 | 3 | 1 | 0 | 36 | and \$1,\$2, \$3 |
| or | R | 0 | 2 | 3 | 1 | 0 | 37 | or \$1, \$2, \$3 |
| lw | I | 35 | 2 | 1 | 100 | | | lw \$1, 100 (\$2) |
| sw | I | 43 | 2 | 1 | 100 | | | sw \$1, 100 (\$2) |
| beq | I | 4 | 1 | 2 | 25 | | | beq \$1, \$2, 100 |
| j | J | 2 | 2500 | | | | | j 10000 |

# What about subtraction (a – b) ?

- Two's complement approach: just negate b and add.
- A clever solution: negate+1 = 2's complement
  negate signal = carry-in for least significant bit position





B xor 1 = not B

## Tailoring the ALU to the MIPS

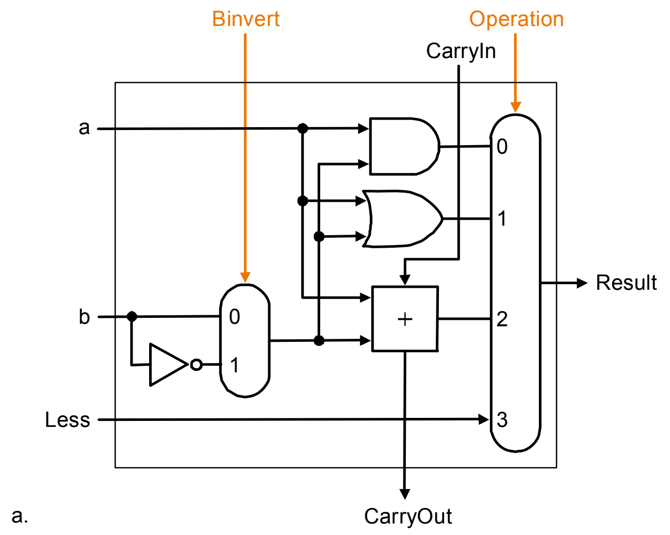
- Need to support test for equality (beq \$t5, \$t6, L)
  - use subtraction: (a-b) = 0 implies a = b

- to support the set-on-less-than instruction

  (slt \$r1, \$r2, \$r3)
  - remember: slt is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b

## Supporting slt

- Can we figure out the idea?



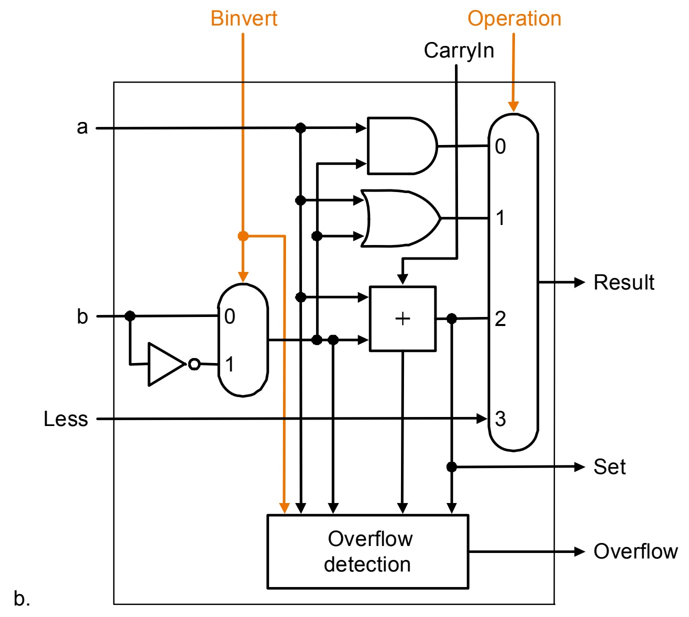
- If a<b then (a-b)<0

So, if after subtraction, the result is Negative, i.e. MSB=1

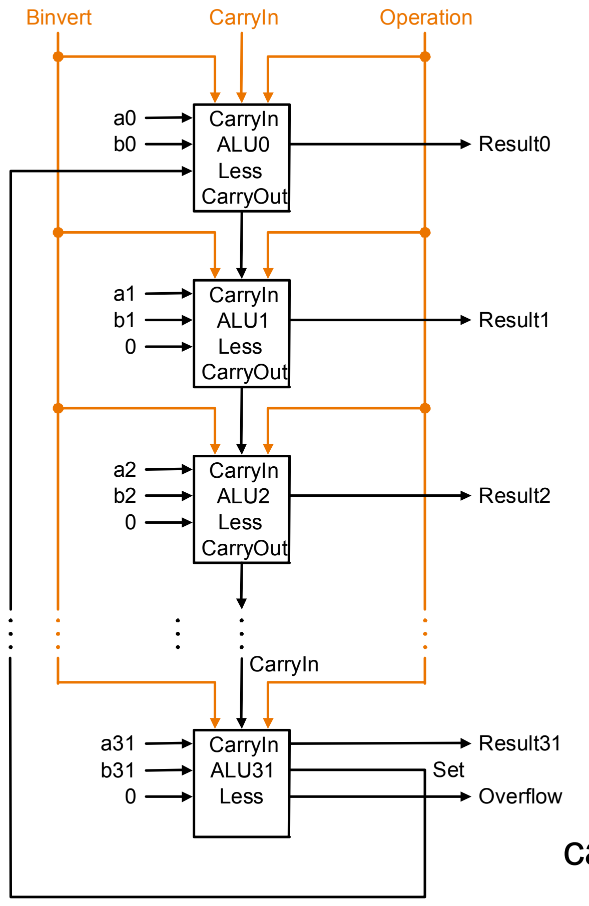
Then

LSB of Result =MSB

and other bits of Result = 0



# Supporting slt & overflow



for sub

for 2's compl.

to select a particular op

Sign-bit, MSB
goes to LSB for slt

carry-in ≠ carry-out

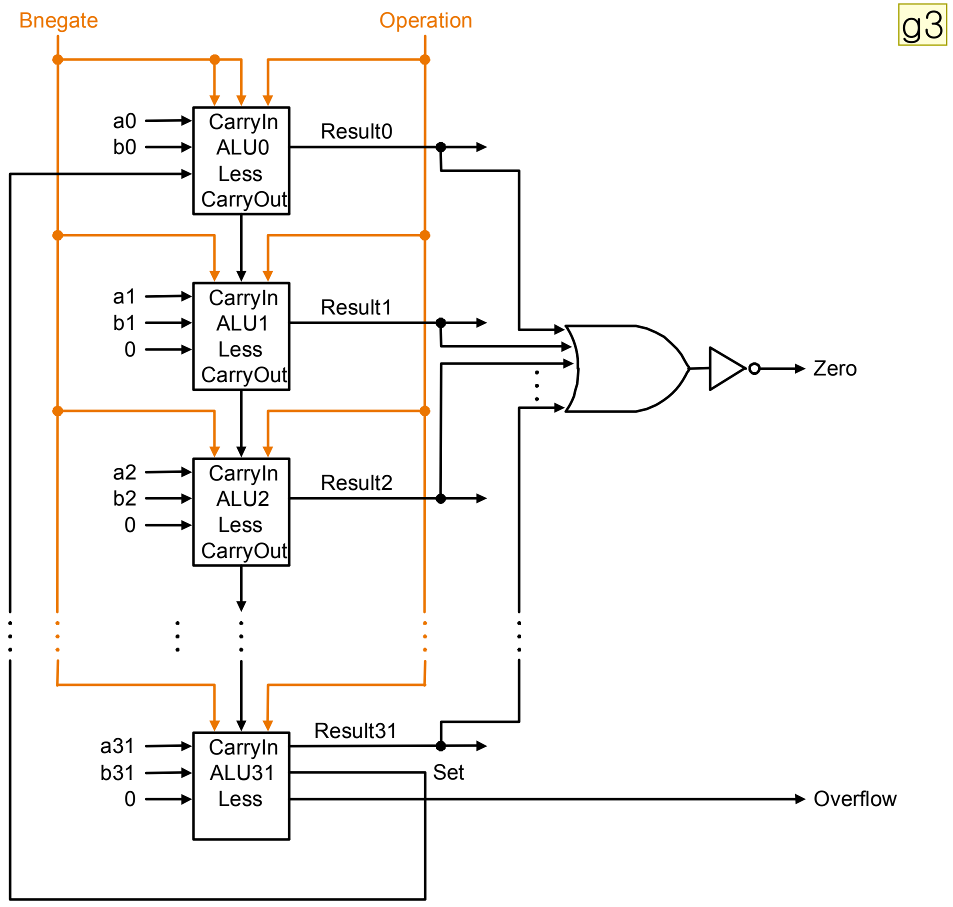
## Test for equality

•***Note: zero is 1 when the result is zero!***

◆ Notice control lines:

```
0000 = and
0001 = or
0010 = add
0110 = subtract
0111 = slt
```

(see Sec. 4.4. p.247)



g3 for beq, subtraction yields 0. all bit positions will be o. so negating them makes 1.
for sub and slt we need subtraction, so Bnegate will be 1 and it will feed to the lsb as a carry-in to have two's comlement.
ghlee, 2015-03-31

## Chapter Three Summary

- Computer arithmetic is constrained by limited precision
    - May Not be  asscociative $(a+b)+c \neq a+(b+c)$
- Bit patterns have no inherent meaning but standards do exist
    - two’s complement
    - IEEE 754 floating point
- Computer instructions determine “meaning” of  the bit patterns

For Fast Adder, read the supplemental slides posted!

**We are ready to move on (and implement the processor)!**