# Computer Organization and Design

## Arithmetic & Logic Circuits

Henry Fuchs

Slides adapted from Montek Singh, who adapted them
from Leonard McMillan and from Gary Bishop
Back to McMillan & Chris Terman, MIT 6.004 1999

Tuesday, March 24, 2015

Lecture 11

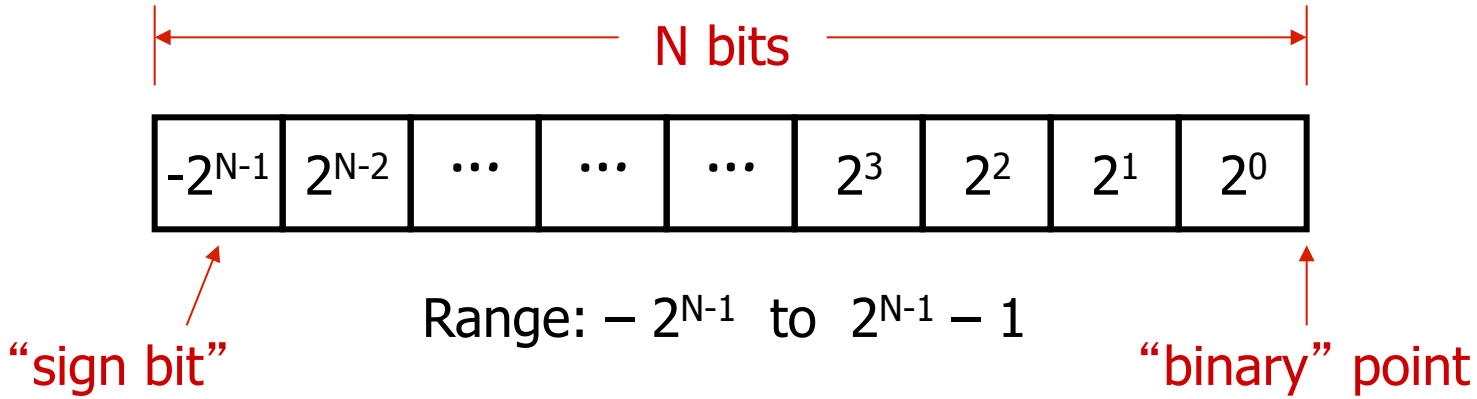
# Topics



- Arithmetic Circuits
  - adder
  - subtractor
- Logic Circuits
- Arithmetic & Logic Unit (ALU)
  - … putting it all together

# Review: 2's Complement






- 8-bit 2's complement example:

  11010110 = – 128 + 64 + 16 + 4 + 2 = – 42

- Beauty of 2's complement representation:
  - same binary addition procedure will work for adding both signed and unsigned numbers
    - as long as the leftmost carry out is properly handled/ignored
- Insert a "binary" point to represent fractions too:

  1101.0110 = – 8 + 4 + 1 + 0.25 + 0.125 = – 2.625

# Binary Addition



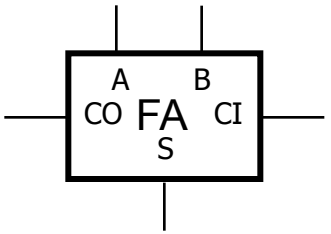
## ✱ Example of binary addition "by hand":

Carries from previous column

```
  1 1 0 1
A:  1101
B:+ 0101
  10010
```

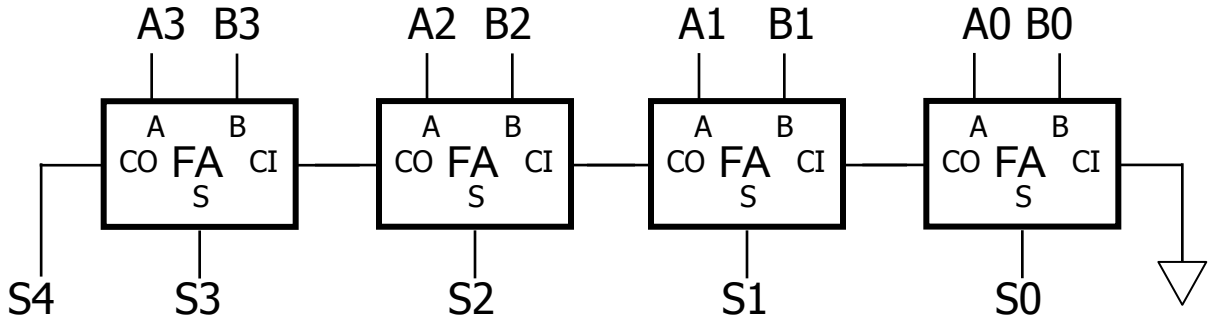
Adding two N-bit numbers produces an (N+1)-bit result

## ✱ Let's design a circuit to do it!




- Start by building a block that adds one column:
  - called a "full adder" (FA)
- Then cascade them to add two numbers of any size…

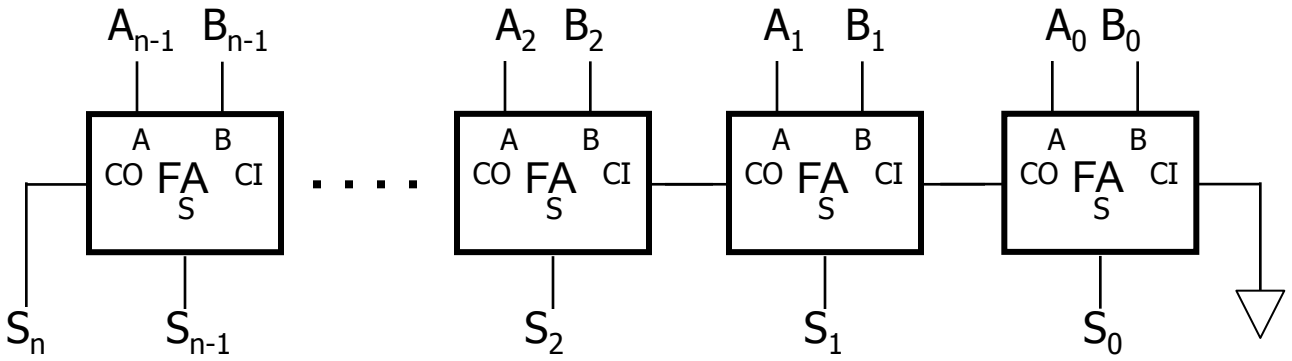
4-bit adder →



# Binary Addition



- Extend to arbitrary # of bits
  - *n* bits: cascade *n* full adders




  - Called "Ripple-Carry Adder"
    - carries ripple through from right to left
    - longest chain of carries has length *n*
      - how long does it take to add the numbers 0 and 1?
      - how long does it take to add the numbers -1 and 1?

# Designing a Full Adder (1-bit adder)



- Follow the step-by-step method
  1. Start with a truth table:
  2. Write down eqns for the "1" outputs:

$$C_o = \overline{C_i}AB + C_i\overline{A}B + C_iA\overline{B} + C_iAB$$

$$S = \overline{C_i}\,\overline{A}B + \overline{C_i}A\overline{B} + C_i\overline{A}\,\overline{B} + C_iAB$$

  3. Simplifying a bit (seems hard, but experienced designers are good at this art!)

| $C_i$ | A | B | $C_o$ | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

$$C_o = C_i\left(A + B\right) + AB \quad = C_i\left(A \oplus B\right) + AB$$

$$S = C_i \oplus A \oplus B$$

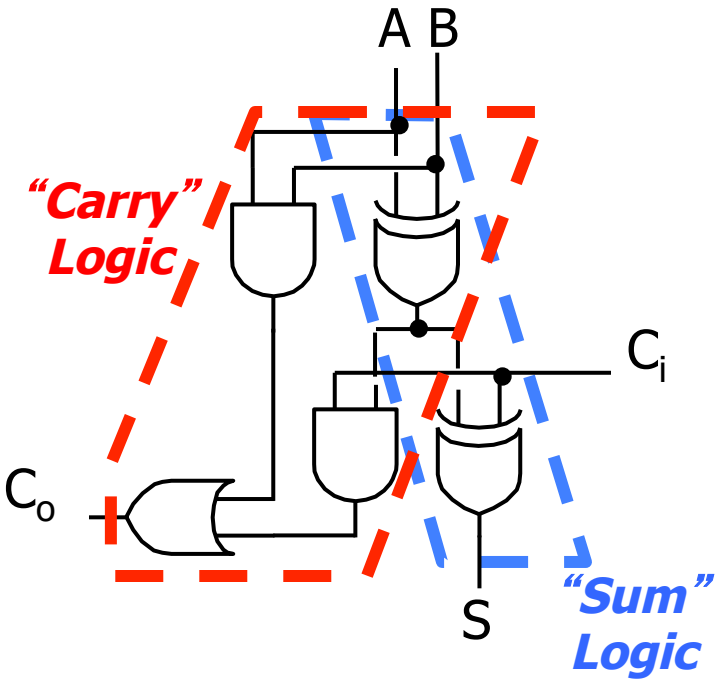
# For Those Who Prefer Logic Diagrams



- A little tricky, but only 5 gates/bit!

$$C_o = C_i \cdot \left(A \oplus B\right) + A \cdot B$$

$$S = C_i \oplus \left(A \oplus B\right)$$

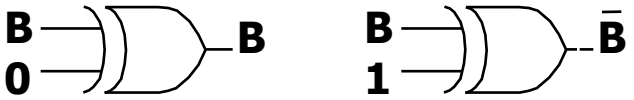


# Subtraction: A-B = A + (-B)

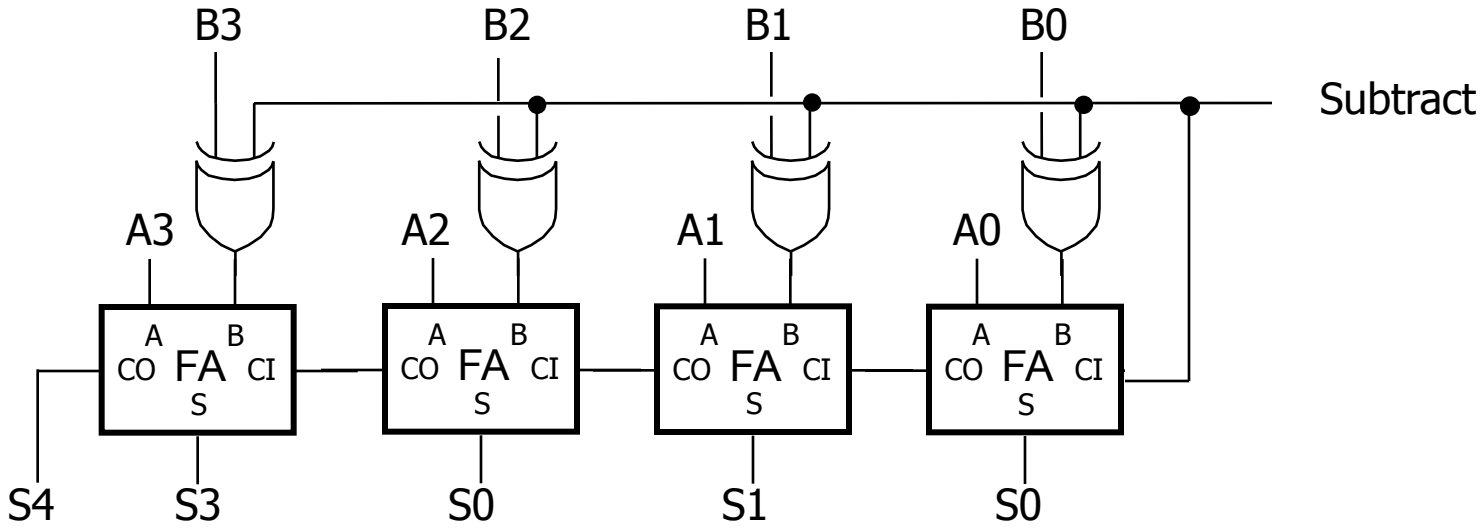


- Subtract B from A = add 2's complement of B to A
  - In 2's complement: –B = ~B + 1

~ = bit-wise complement




  - Let's build an arithmetic unit that does *both* add and sub
    - operation selected by *control input*:



# Condition Codes



- One often wants 4 other bits of information from an arith unit:
  - Z (zero): result is = 0
    - big NOR gate
  - N (negative): result is < 0
    - $S_{N-1}$
  - C (carry): indicates that most significant position produced a carry, e.g., "1 + (-1)"
    - Carry from last FA
  - V (overflow): indicates answer doesn't fit
    - precisely:

$$V = A_{n-1} B_{n-1} \overline{N} + \overline{A}_{n-1} \overline{B}_{n-1} N$$

-or-

$$V = Co_{n-1} \oplus Ci_{n-1}$$

To compare A and B, perform A–B and use condition codes:

Signed comparison:

| | |
|---|---|
| **LT** | **N⊕V** |
| **LE** | **Z+(N⊕V)** |
| **EQ** | **Z** |
| **NE** | **~Z** |
| **GE** | **~(N⊕V)** |
| **GT** | **~(Z+(N⊕V))** |

Unsigned comparison:

| | |
|---|---|
| **LTU** | **~C** |
| **LEU** | **~C+Z** |
| **GEU** | **C** |
| **GTU** | **~(~C+Z)** |

**Fixed 3/24/2015**

# Condition Codes



- Subtraction is useful also for *comparing* numbers

- To compare A and B:
  - First compute A – B
  - Then check the flags Z, N, C, V

- Examples:
  - LTU (less than for unsigned numbers)
    - A < B is given by ~C
  - LT (less than for signed numbers)
    - A < B is given by N⊕V
  - Others in table

To compare A and B, perform A–B and use condition codes:

Signed comparison:

| | |
|---|---|
| LT | N⊕V |
| LE | Z+(N⊕V) |
| EQ | Z |
| NE | ~Z |
| GE | ~(N⊕V) |
| GT | ~(Z+(N⊕V)) |

Unsigned comparison:

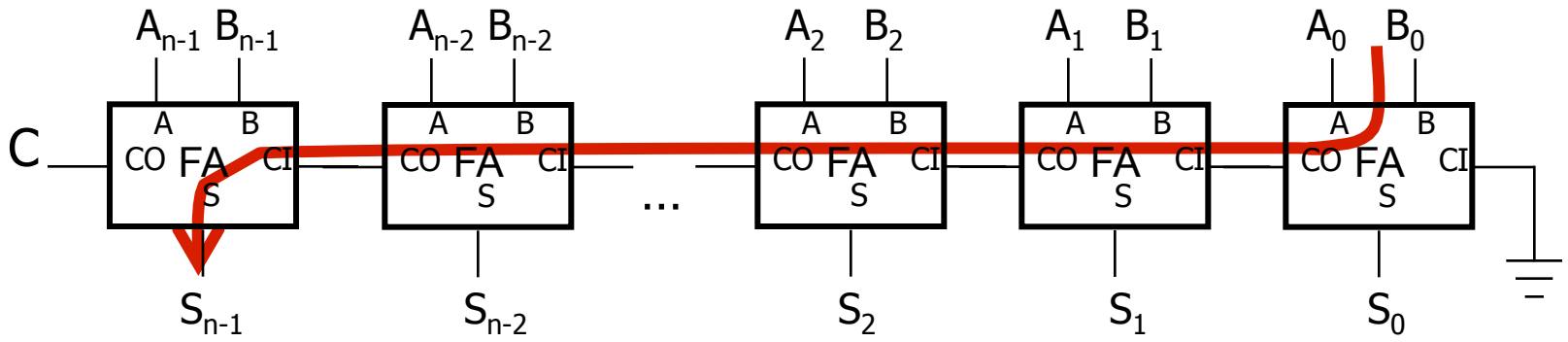
| | |
|---|---|
| LTU | ~C |
| LEU | ~C+Z |
| GEU | C |
| GTU | ~(~C+Z) |

Fixed 3/24/2015

# Latency of Ripple-Carry Adder



($T_{PD}$ = max propagation delay or latency of the entire adder)



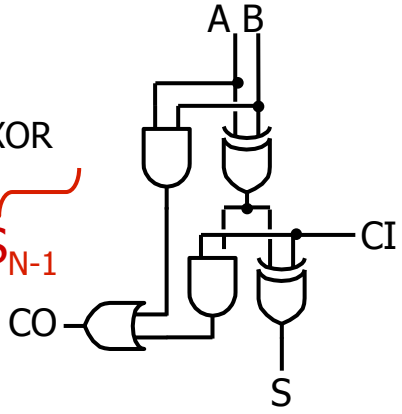

Worse-case path: carry propagation from LSB to MSB, e.g., when adding 11…111 to 00…001.

$$t_{PD} = (t_{PD,XOR} + t_{PD,AND} + t_{PD,OR}) + (N-2)*(t_{PD,OR} + t_{PD,AND}) + t_{PD,XOR}$$

A,B to CO

CI to CO

$CI_{N-1}$ to $S_{N-1}$




$\approx \Theta(N)$

$\Theta(N)$ is read “order N” and tells us that the latency of our adder grows in proportion to the number of bits in the operands.

# Can we add faster?



- Yes,
  there are many sophisticated designs that are faster
  - Carry-Lookahead Adders (CLA)
  - Carry-Skip Adders
  - Carry-Select Adders

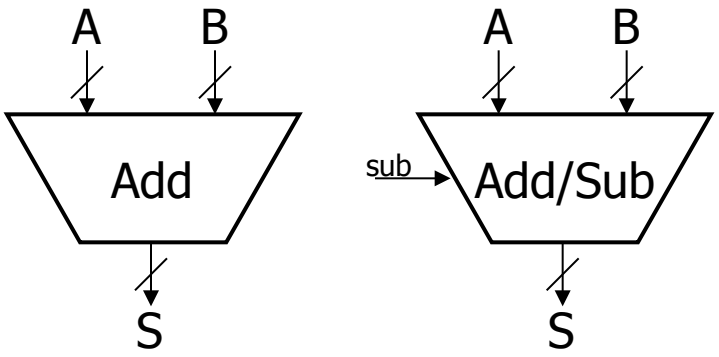
  ➔ See textbook for details

# Adder Summary



- Adding is not only common, but it is also tends to be one of the most time-critical of operations
  - As a result, a wide range of adder architectures have been developed that allow a designer to tradeoff complexity (in terms of the number of gates) for performance.


At this point we' ll define a high-level functional unit for an adder, and specify the details of the implementation as necessary.

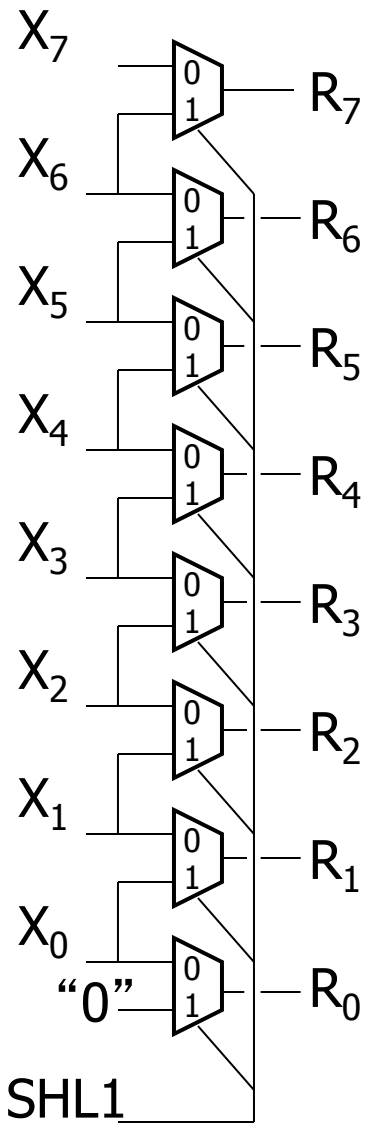


# Shifting and Logical Operations

# Shifting Logic



- Shifting is a common operation
  - applied to groups of bits
  - used for alignment
  - used for "short cut" arithmetic operations
    - X << 1 is often the same as 2*X
    - X >> 1 can be the same as X/2
- For example:
  - $X = 20_{10} = 00010100_2$
  - Left Shift:
    - $(X << 1) = 00101000_2 = 40_{10}$
  - Right Shift:
    - $(X >> 1) = 00001010_2 = 10_{10}$
  - Signed or "Arithmetic" Right Shift:
    - $(-X >>> 1) = (11101100_2 >>> 1) = 11110110_2 = -10_{10}$



# Boolean Operations

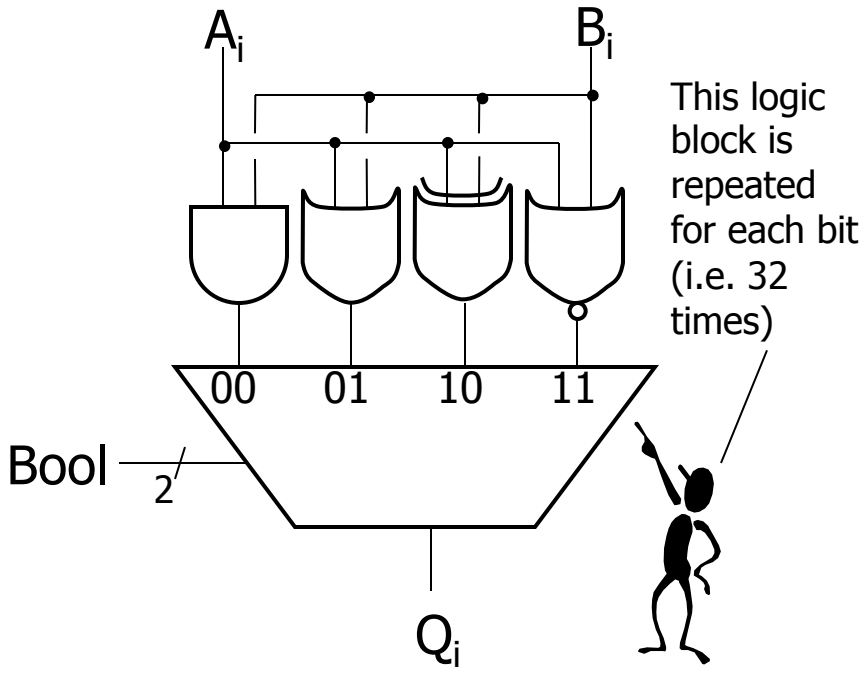


- It will also be useful to perform logical operations on groups of bits. Which ones?
  - ANDing is useful for "masking" off groups of bits.
    - ex. 10101110 & 00001111 = 00001110 (mask selects last 4 bits)
  - ANDing is also useful for "clearing" groups of bits.
    - ex. 10101110 & 00001111 = 00001110 (0's clear first 4 bits)
  - ORing is useful for "setting" groups of bits.
    - ex. 10101110 | 00001111 = 10101111 (1's set last 4 bits)
  - XORing is useful for "complementing" groups of bits.
    - ex. 10101110 ^ 00001111 = 10100001 (1's invert last 4 bits)
  - NORing is useful for.. uhm…
    - ex. 10101110 # 00001111 = 01010000 (0's invert, 1's clear)

# Boolean Unit



- It is simple to build up a Boolean unit using primitive gates and a mux to select the function.
  - Since there is no interconnection between bits, this unit can be simply replicated at each position.
  - The cost is about 7 gates per bit. One for each primitive function, and approx 3 for the 4-input mux.

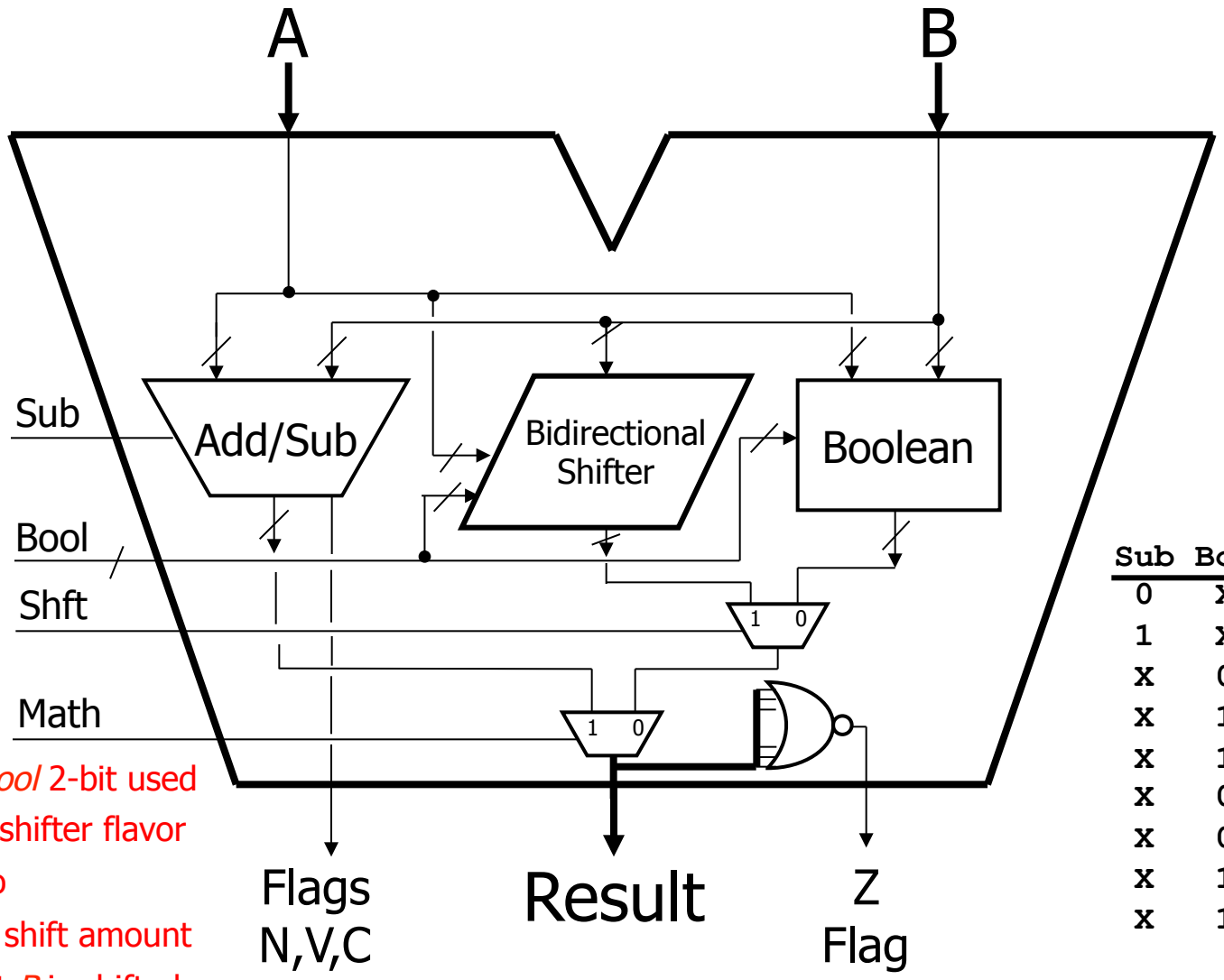



- This is a straightforward, but not too elegant of a design.

# An ALU, at Last (without comparisons)



- Combine add/sub, shift and Boolean units




- *Bool* 2-bit used for shifter flavor also
- *A* shift amount that *B* is shifted

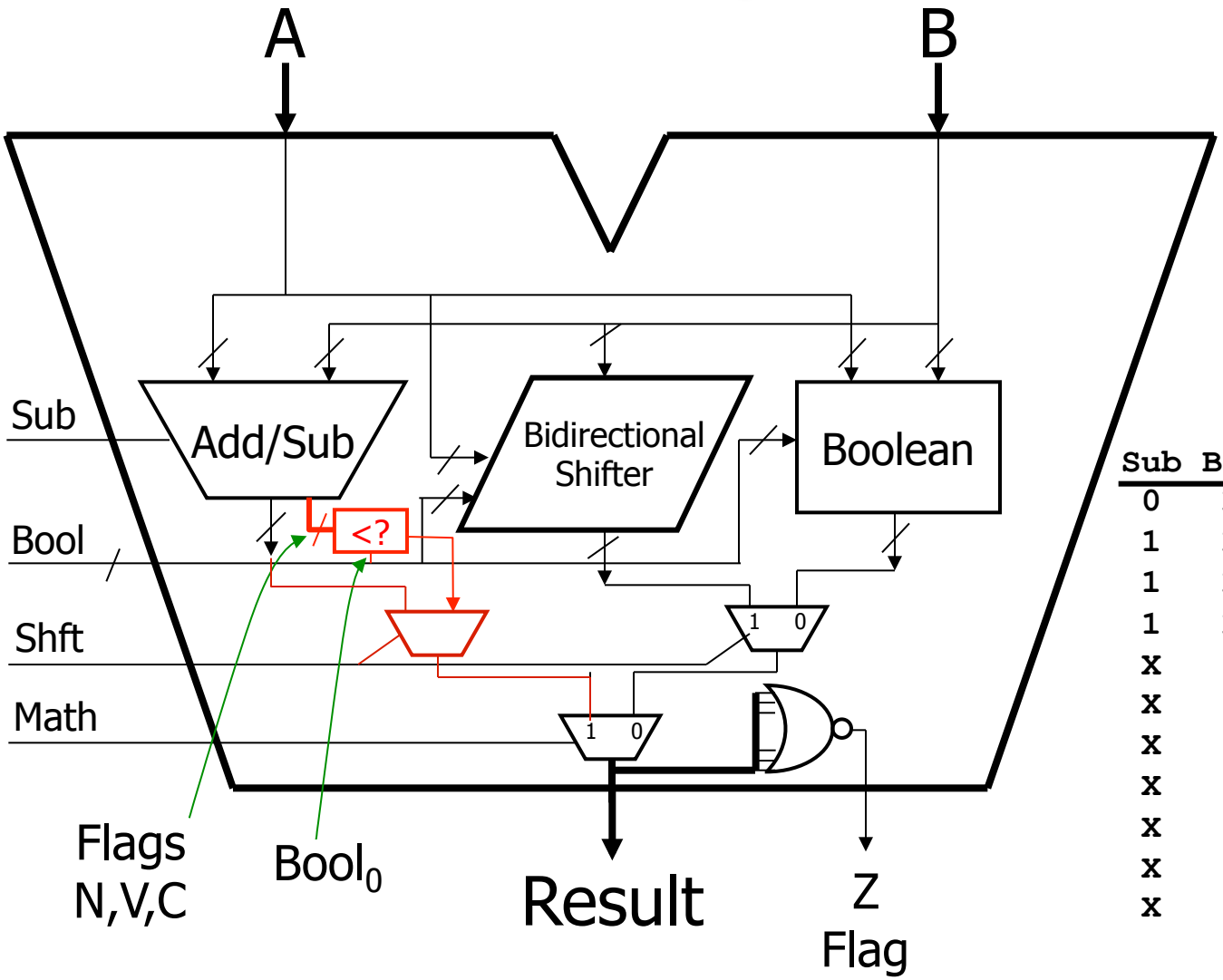
5-bit ALUFN

| Sub | Bool | Shft | Math | OP |
|---|---|---|---|---|
| 0 | XX | 0 | 1 | A+B |
| 1 | XX | 0 | 1 | A-B |
| X | 00 | 1 | 0 | B<<A |
| X | 10 | 1 | 0 | B>>A |
| X | 11 | 1 | 0 | B>>>A |
| X | 00 | 0 | 0 | A & B |
| X | 01 | 0 | 0 | A \| B |
| X | 10 | 0 | 0 | A ^ B |
| X | 11 | 0 | 0 | $\overline{A \mid B}$ |

# A Complete ALU



- With support for comparisons (LT and LTU)
  - LTU = ~C and LT = N⊕V




5-bit ALUFN

| Sub | Bool | Shft | Math | OP |
|---|---|---|---|---|
| 0 | XX | 0 | 1 | A+B |
| 1 | XX | 0 | 1 | A-B |
| 1 | X0 | 1 | 1 | A LT B |
| 1 | X1 | 1 | 1 | A LTU B |
| X | 00 | 1 | 0 | B<<A |
| X | 10 | 1 | 0 | B>>A |
| X | 11 | 1 | 0 | B>>>A |
| X | 00 | 0 | 0 | A & B |
| X | 01 | 0 | 0 | A \| B |
| X | 10 | 0 | 0 | A ^ B |
| X | 11 | 0 | 0 | $\overline{A \mid B}$ |

# Next



- Circuits for
  - Multiplication
  - Division
  - Floating-Point Operations