Lecture 7: Arithmetic Logic Unit

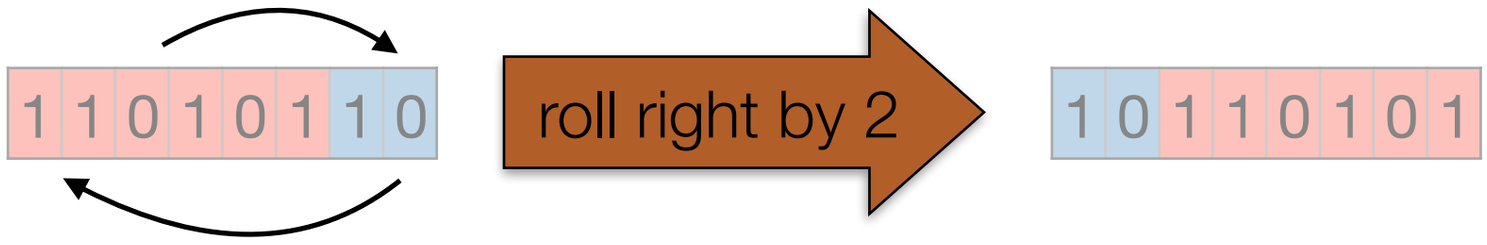
Spring 2024
Jason Tang

Topics

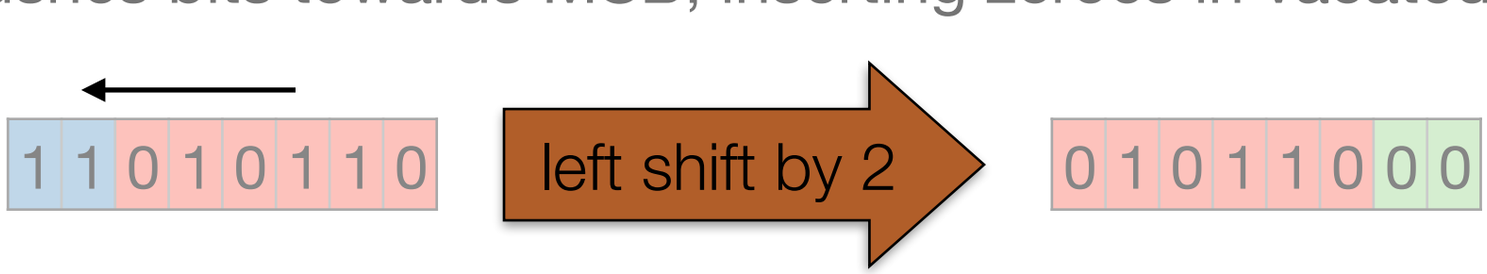
- Logical operations
- 1-bit adder (mostly review)
- Lookahead adder

Rolling and Shifting

- A roll (or rotate) pushes bits off of one end and reinserts them at the other end



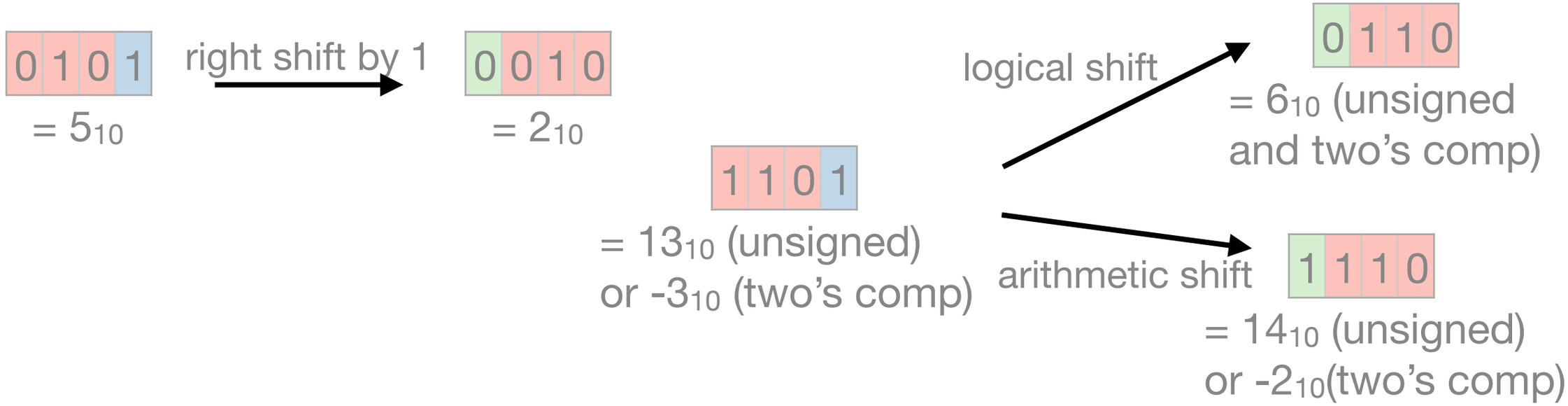
- A left shift pushes bits towards MSB, inserting zeroes in vacated bit positions



- Two different types of right shift, both pushing towards LSB:
  - Logical right shift: vacated bits are set to zero
  - Arithmetic right shift: vacated bits are signed extended

Shifting Dangers

- Left shifting can be used as a cheap way to multiply by a power of 2 (but beware of overflow)
    - Left shifting a two's complement number could result in flipping the sign bit
- Right-shifting **sometimes** results in dividing by a power of two, but only when original value was non-negative, or correct shift operation was used



C and Java Shift Operators

- In both C and Java, << is logical left shift
- In Java. >> is arithmetic right shift, >>> is logical right shift
- In C, usually, >> is arithmetic for signed values and logical for unsigned

```
#include <stdio.h>

int main(void) {
    signed char a = -42;
    printf("a is %02x, shifted: %02x\n”,
        a, (a >> 4));
    unsigned char b = (unsigned char) a;
    printf("b is %02x, shifted: %02x\n”,
        b, (b >> 4));
    return 0;
}
```

```
a is ffffffd6, shifted: fffffffd
b is d6, shifted: 0d
```

Bitwise Operations

- Even though memory is (somewhat) cheap, one optimization is to pack different values into a single word
    - Example: store 8 separate boolean variables in a 8-bit bitfield
- Use logical AND, OR, and NOT (or NEGATE) instructions to isolate individual bits
    - AND often used to create a bitmask
- Other operators are NAND, NOR, XOR, and NXOR

C Bitwise Operations

- Combination of AND and shift are used to extract individual bits from an integer

- Combination of OR and shift are used to set individual bits in an integer

```c
#include <stdio.h>

int main(void) {
    unsigned char a = 0b11000101;
    printf("orig a: %02x\n", a);
    printf("middle 4: %02x\n", ((a >> 2) & 0xf));
    a = (0x03 << 4) | (a & 0x0f);
    printf("new a: %02x\n", a);
    unsigned char b = ~a;
    printf("b: %02x\n", b);
    return 0;
}
```

```
orig a: c5
middle 4: 01
new a: 35
b: ca
```

C Bitfields

- Combination of unions and bitfield structs can be used to manipulate individual bits
  - Almost always involves unsigned fixed-width integers
- Is **compiler dependent** as to packing and endianness order of bitfield
  - Bitfields are convenient if the code will only be compiled with a particular compiler and run on a particular architecture
  - Otherwise use bitwise operations to remain portable

http://c-faq.com/struct/bitfields.html

C Bitfield Example

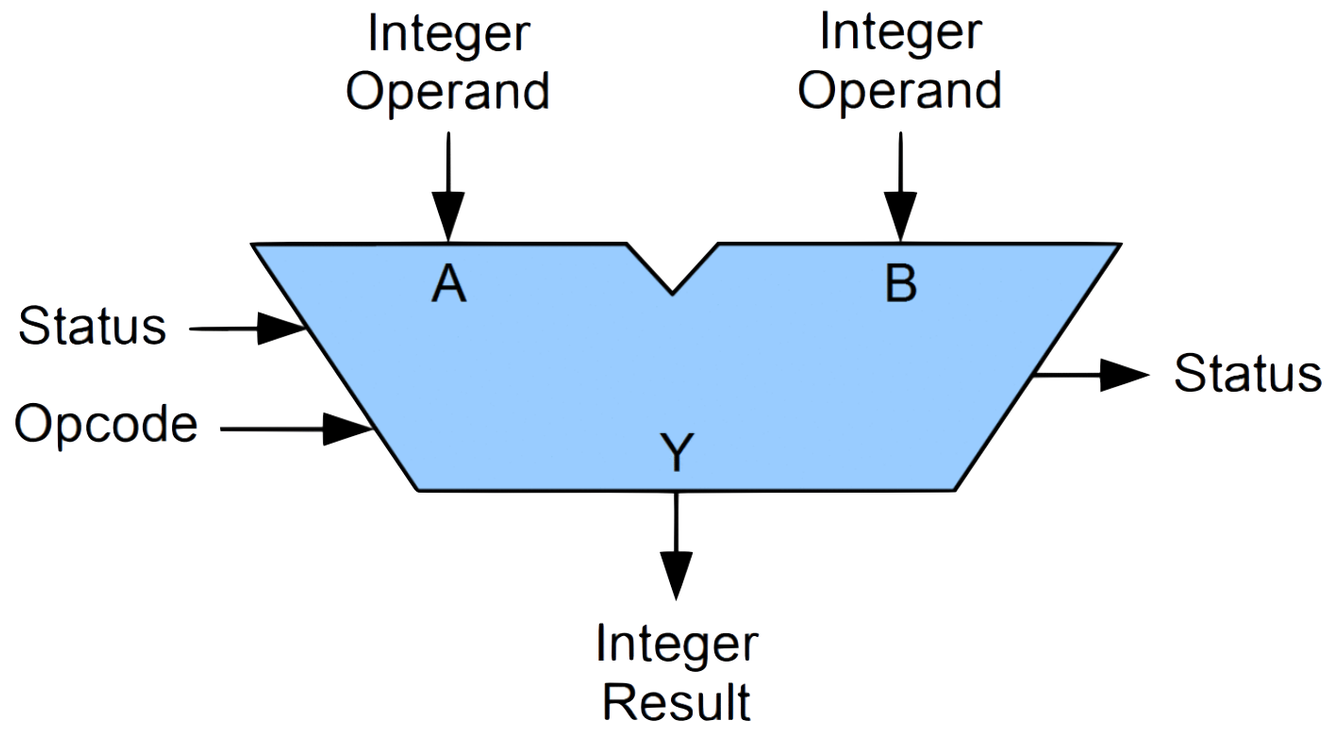
```c
#include <stdio.h>
#include <stdint.h>

union u {
    uint8_t val;
    struct {
        unsigned upper: 4;
        unsigned next: 2;
        unsigned flag1: 1;
        unsigned flag2: 1;
    } bits;
};

int main(void) {
    union u a;
    a.val = 0b11000101;
    printf("orig a: %02x\n", a.val);
    printf("upper 4: %02x\n", a.bits.upper);
    a.bits.flag1 = 0;
    printf("new a: %02x\n", a.val);
    return 0;
}
```

On macOS, with clang 900.0.39.2

```
orig a: c5
upper 4: 05
new a: 85
```

Arithmetic Logic Unit

- Hardware device that performs simple **integer** operations
- Handles up to two operands
- Has a selector to choose which operation to perform:
  - Add or subtract; usually logical operations like rotate, shift, and bitwise
  - Sometimes more complex operations like square root
- Typically, set condition code (also known as status) based upon operation



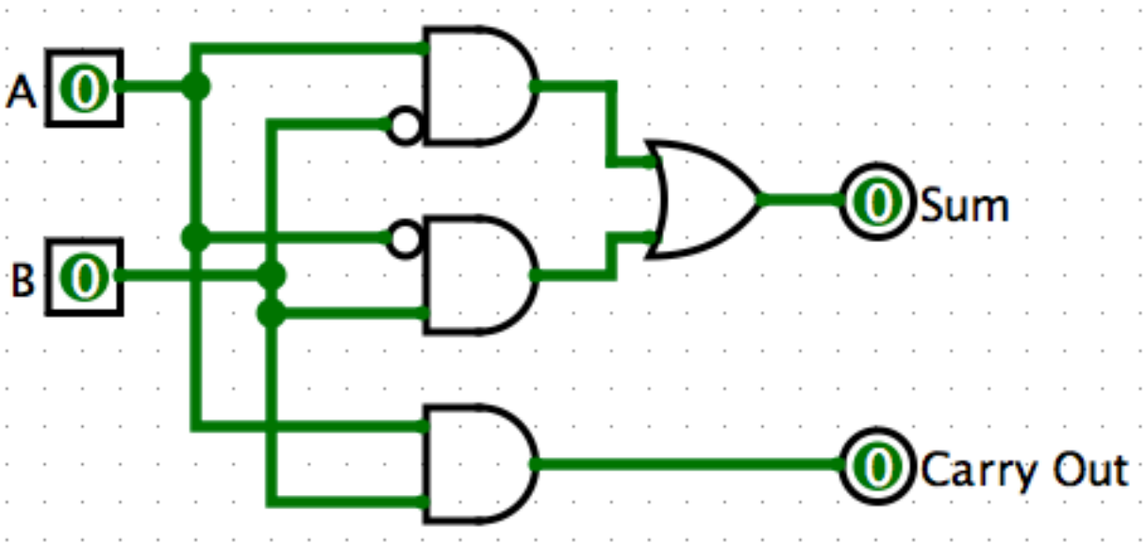
https://en.wikipedia.org/wiki/Arithmetic_logic_unit

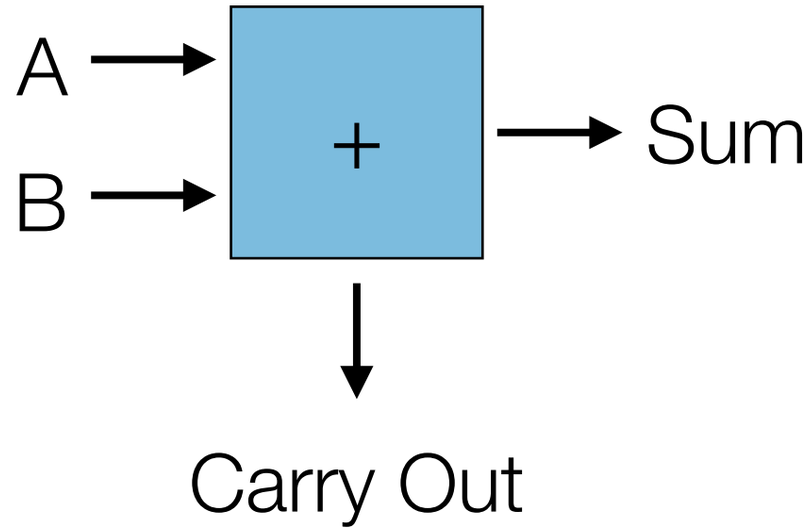
1-Bit Half Addition

- In simplest case, a half-adder (also known as a (2, 2) adder) adds two bits together, and calculates a sum and a carry-out

$$Sum = A \oplus B = A \cdot \overline{B} + \overline{A} \cdot B \qquad CarryOut = A \cdot B$$

Input		Outputs	
A	B	Sum	Carry Out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1





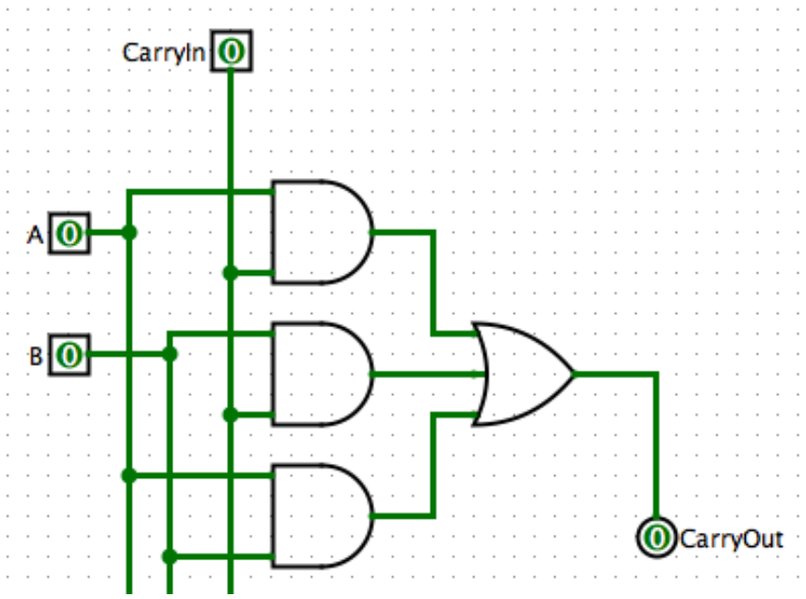
1-Bit Full Adder

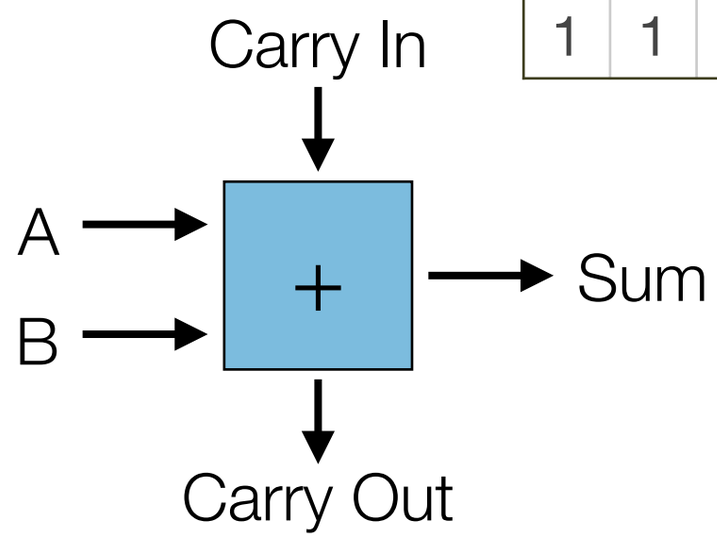
- A full-adder (also known as (3, 2) adder) includes a carry-in bit as well

$$Sum = (A \cdot \overline{B} \cdot \overline{C_{in}}) + (\overline{A} \cdot B \cdot \overline{C_{in}}) + (\overline{A} \cdot \overline{B} \cdot C_{in}) + (A \cdot B \cdot C_{in})$$

$$C_{out} = (B \cdot C_{in}) + (A \cdot C_{in}) + (A \cdot B) + (A \cdot B \cdot C_{in}) = (B \cdot C_{in}) + (A \cdot C_{in}) + (A \cdot B)$$

Inputs			Outputs	
A	B	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

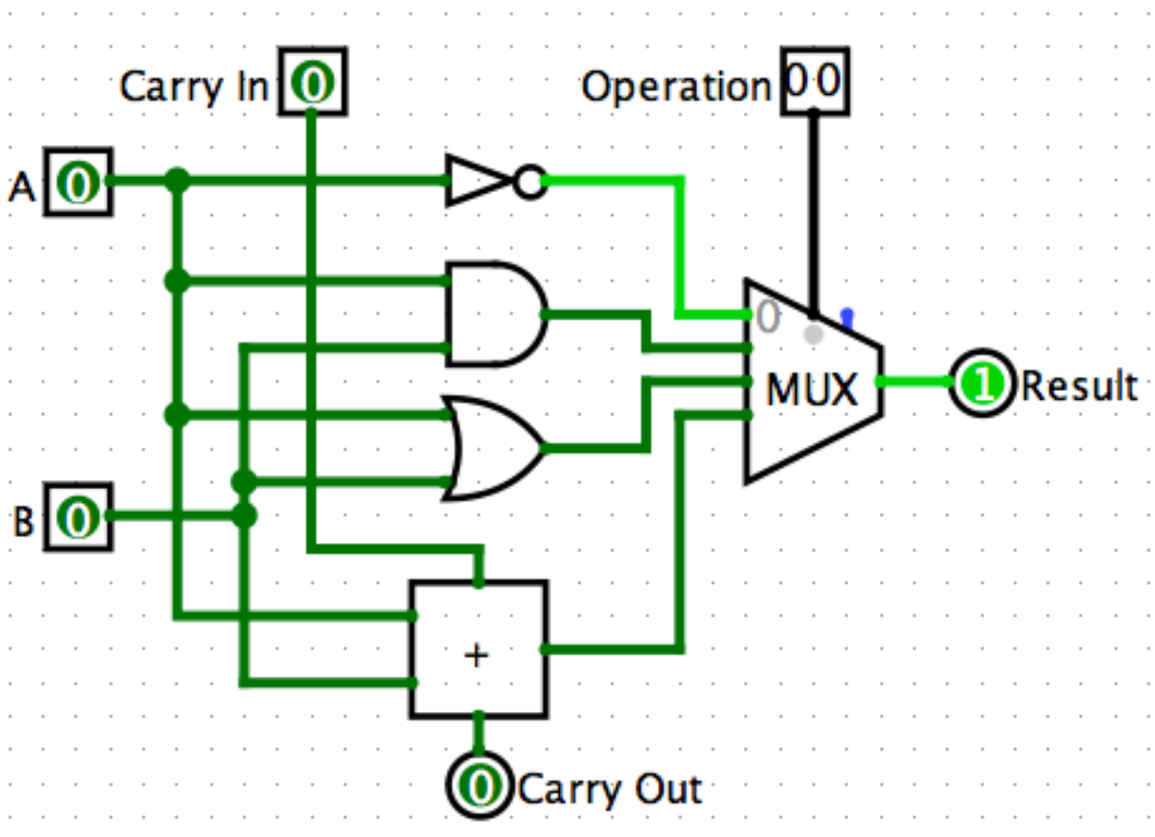




Sum calculation is left as an exercise to the reader

ALU Selector

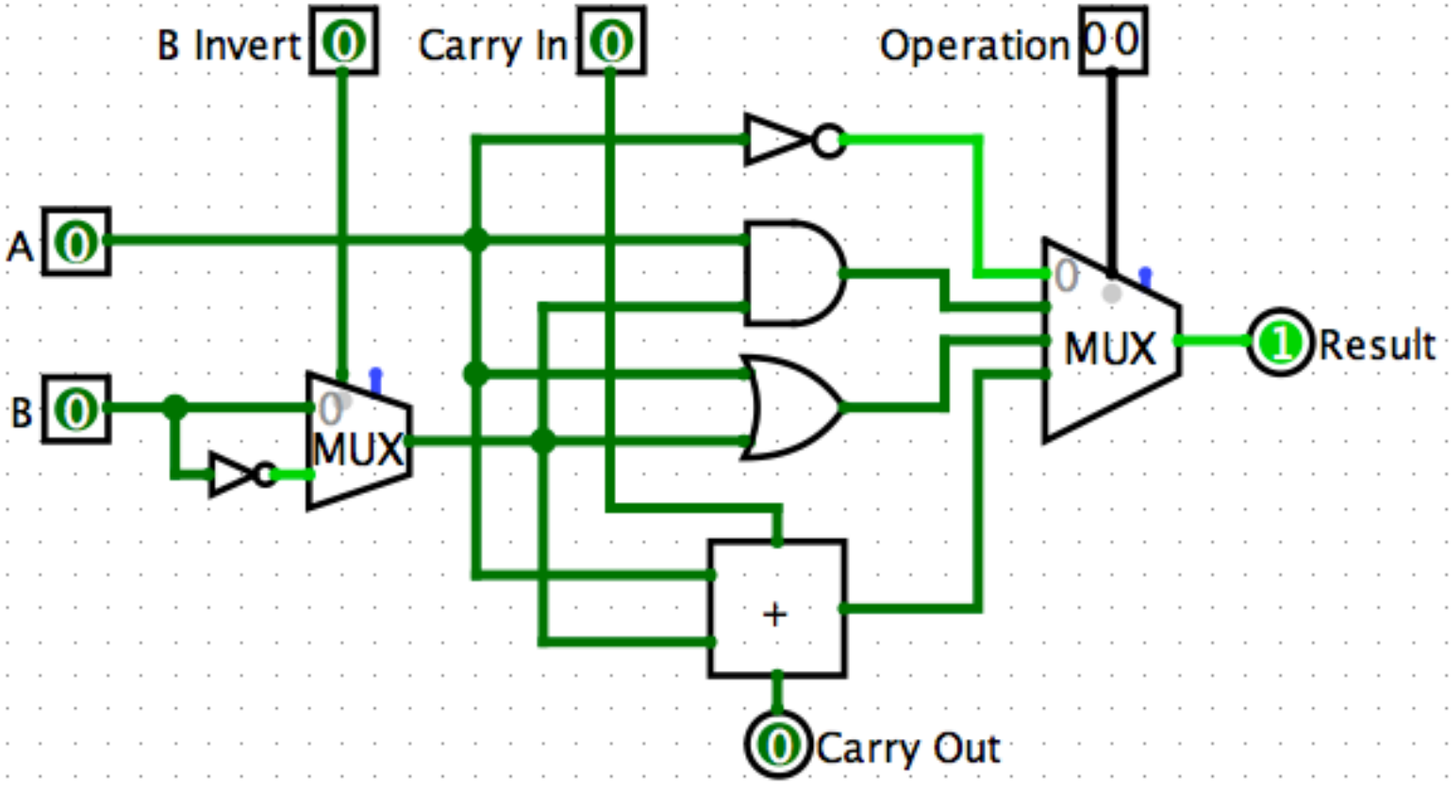
- ALU has a selector to choose which operation to perform
- Example ALU that supports 4 operations:



Operation		Usage
0	0	¬ A
0	1	A ∧ B
1	0	A ∨ B
1	1	A + B

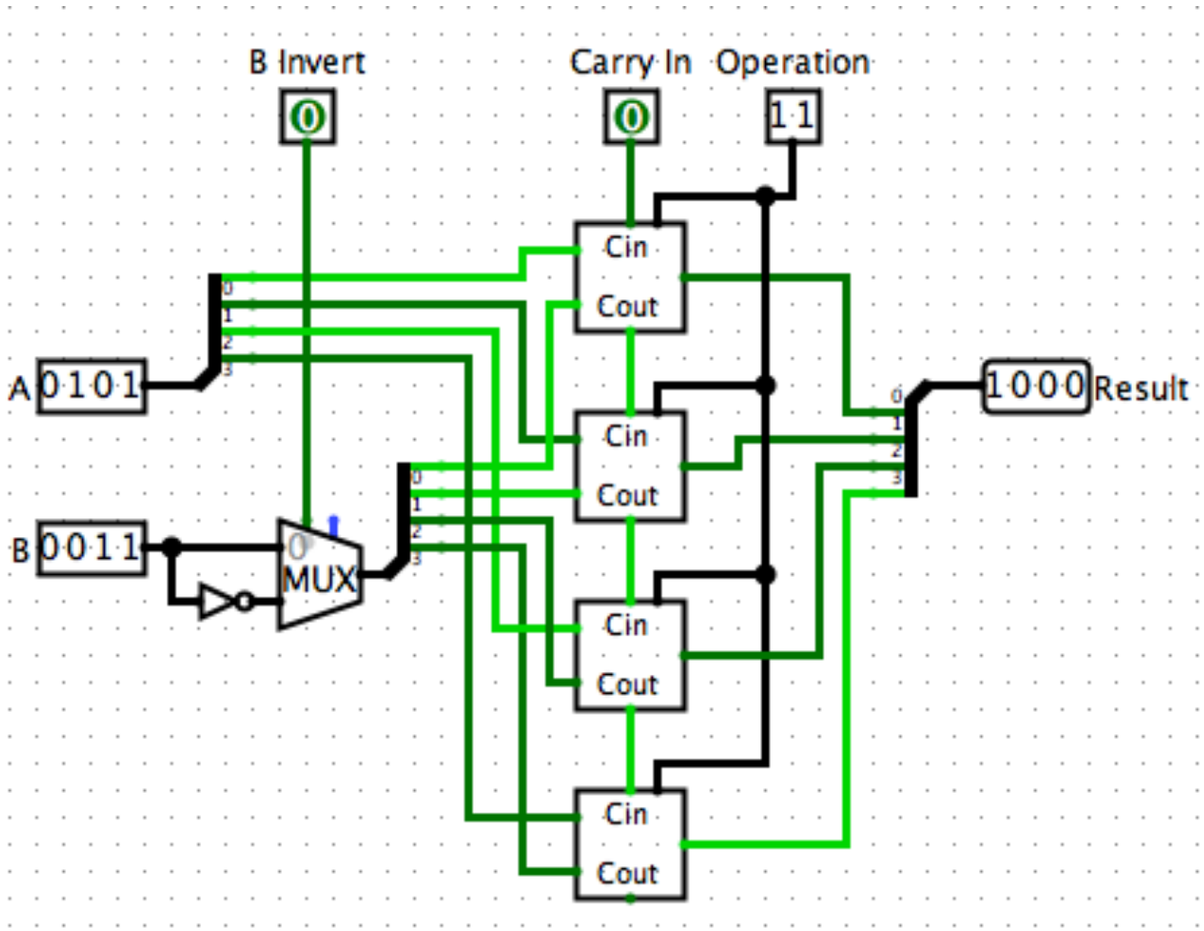
Supporting Subtraction

- Subtraction means adding the negative, and the negative in two's complement is a bit inversion, plus one: $A - B = A + \overline{B} + 1$
  - By adding a selector to **B** operand, the same adder is used for addition and subtraction

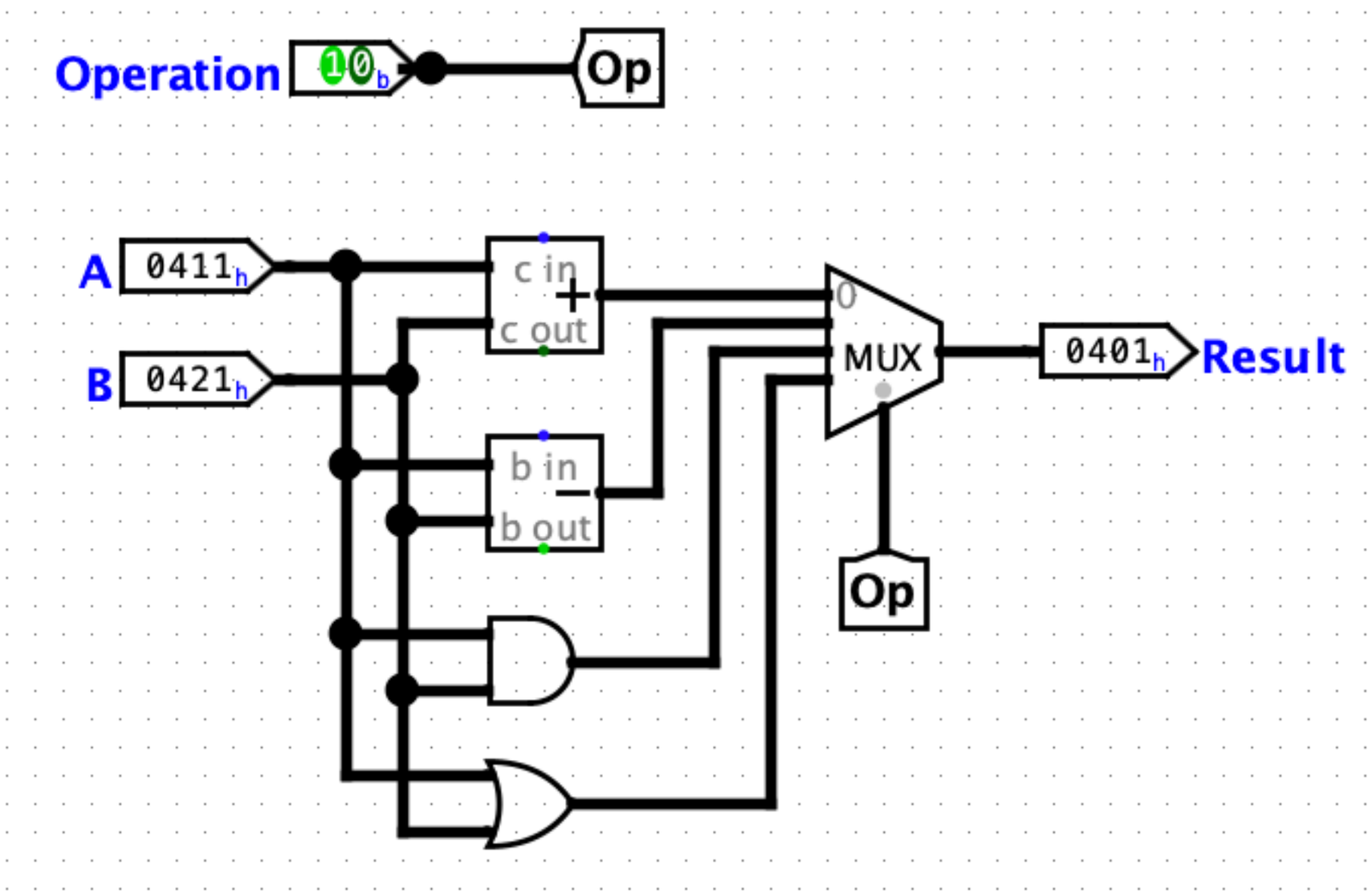


Multibit ALU Design

- A full ALU (16-bit, 32-bit, etc) can be created by connecting adjacent 1-bit ALUs, using Carry In and Carry Out lines
  - Chain Carry Out from one adder to Carry In of next adder (a ripple carry adder)
  - Slow due to gate propagation delay
- Perform subtraction by inverting B and set first Carry In to 1



Alternative Multibit ALU Design

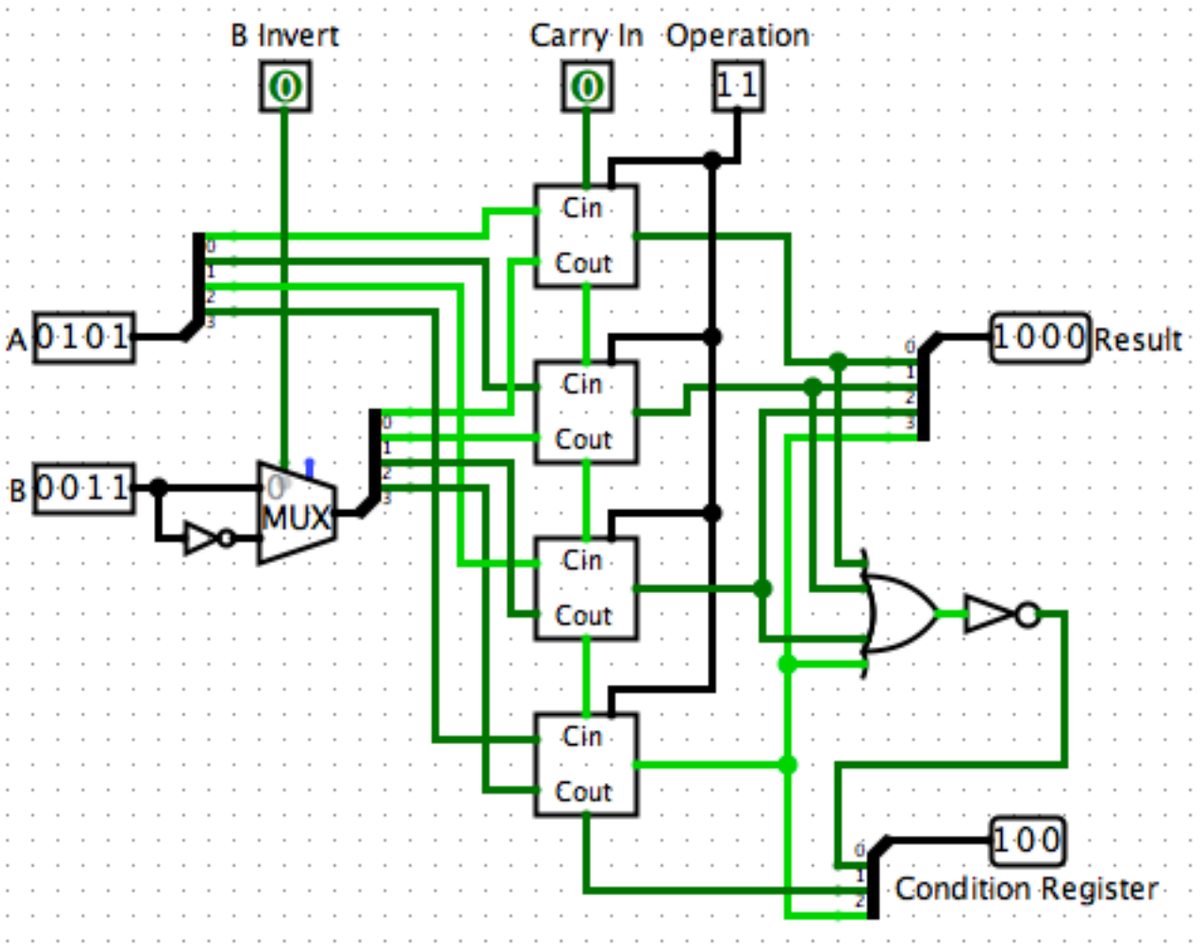


Setting Condition Codes

- Recall that condition code register bits are set automatically as a result of some operations

condition	set by
zero (Z)	result is zero
carry (C)	carry out of MSB
negative (N)	MSB was 1
overflow (V)	an overflow

- By examining the Carry Out bit of MSB adder and resulting value, ALU sets condition bits



Overflow calculation is left as an exercise to the reader

Propagation Delays in Ripple ALUs

- Carry In of a 1-bit adder depends upon result of previous 1-bit adder
- Result of adding most significant bits is only available after all bits (i.e., after **n**-1 single bit additions)
  - Too slow in time-critical hardware
- Carry Lookahead ALU anticipates value of Carry Out
  - Takes many more gates to anticipate carry
  - Worst case scenario is $\log_2(\mathbf{n})$, where **n** is number bits in the adder

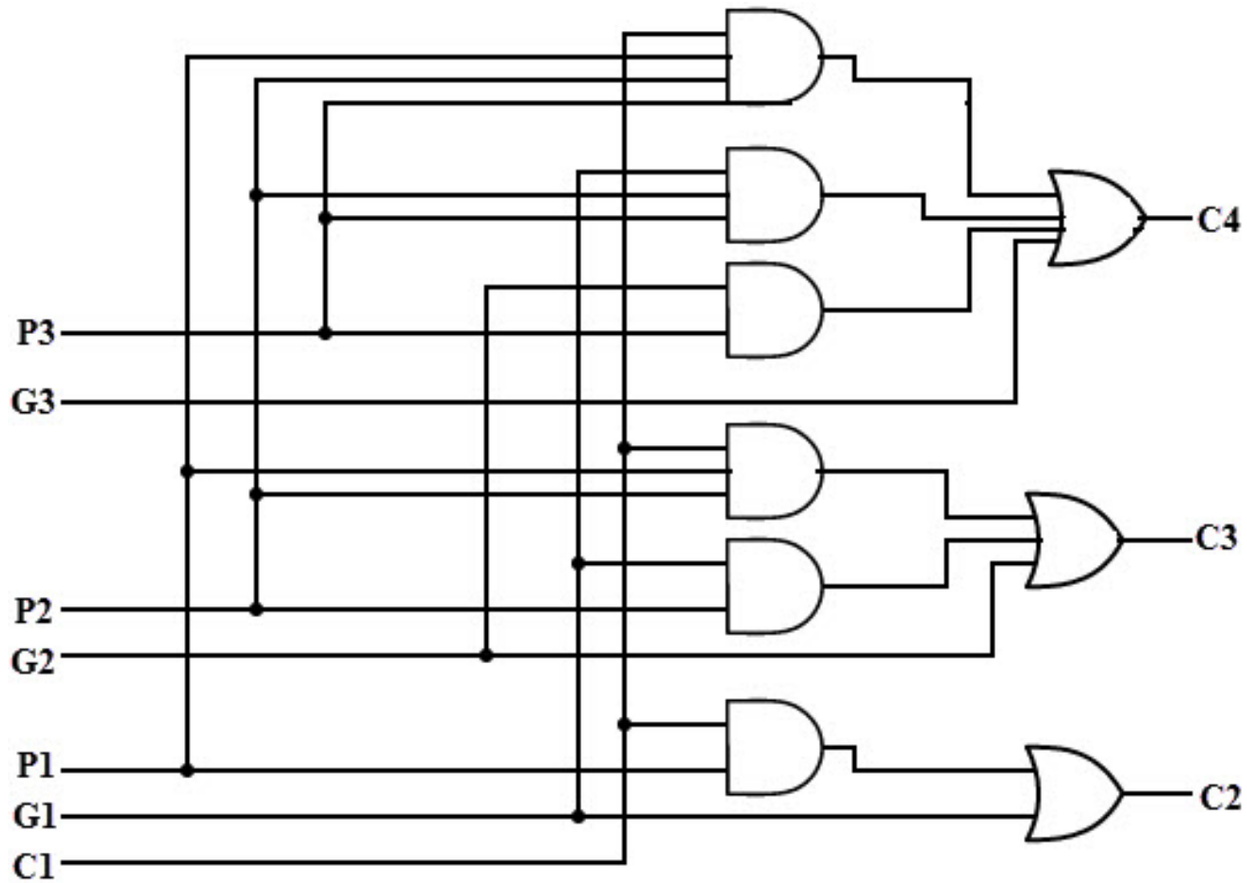
Carry Lookahead Theory

- Let the generate function **G(A, B)** be 1 if **A** plus **B** will generate a Carry Out
  - In binary arithmetic, **G(A, B)** = **A** · **B**, regardless of Carry In ($\mathbf{C_{in}}$)
- Let the propagate function **P(A, B)** be 1 if **A** plus **B** will generate a Carry Out, but only when $\mathbf{C_{in}}$ is 1
  - **P(A, B)** = **A** + **B**; *or* **P(A, B)** = **A** ⊕ **B** because of **G(A,B)** will be **1**
- Then the Carry Out for bit **i** is $\mathbf{C_i} = \mathbf{G_i} + (\mathbf{P_i} \cdot \mathbf{C_{i-1}})$
- For a carry lookahead group with size 2, $\mathbf{C_i} = \mathbf{G_i} + (\mathbf{P_i} \cdot (\mathbf{G_{i-1}} + (\mathbf{P_{i-1}} \cdot \mathbf{C_{i-2}}))$
  - By the Distributive Law, $\mathbf{C_i} = \mathbf{G_i} + (\mathbf{P_i} \cdot \mathbf{G_{i-1}}) + (\mathbf{P_i} \cdot \mathbf{P_{i-1}} \cdot \mathbf{C_{i-2}})$

Using Carry Lookahead Groups

- For a carry lookahead group size 4, $C_4 = G_3 + (P_3 \cdot G_2) + (P_3 \cdot P_2 \cdot G_1) + (P_3 \cdot P_2 \cdot P_1 \cdot G_0) + (P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_{in})$

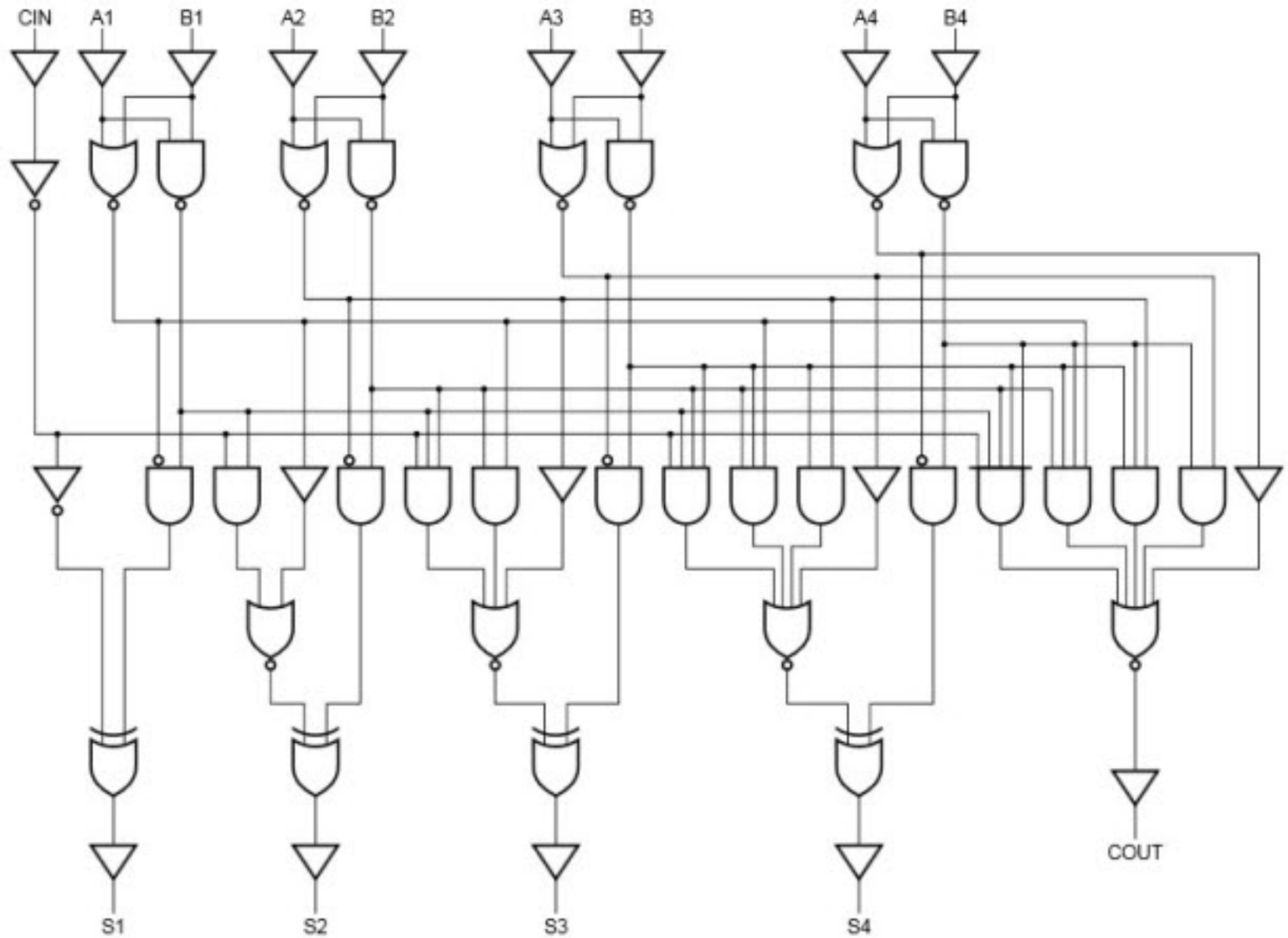
- Therefore, all Carry Outs for the group can be calculated in parallel:



https://www.electronicshub.org/carry-look-ahead-adder/

Cascading Carry Lookahead

- For this circuit, S2 and S4 have equal gate delays
- Carry Out of one group is propagated to next carry lookahead group
  - 16-bit adder can be implemented as 4 groups of 4-bit adders
  - Can create supergroups of carry lookahead groups



Speed of Carry Generation

- For *this* 1-bit adder, the gate delays are 2 for the Carry Out and 3 for the Sum

- If this 1-bit adder were chained together into a 16-bit ripple adder, the gate delays are (16×2) = 32 for the last Carry Out, (15×2+3) = 33 for last Sum bit

- If instead the adder had 4 sets of 4-bit groups, the delays are (4×2) = 8 for last Carry Out, (3×2+3) = 9 for last sum bit

    - Larger groups would be faster, but use more gates