# Unit 10: Arithmetic Logic Units

CSE 220: System Fundamentals I

Stony Brook University

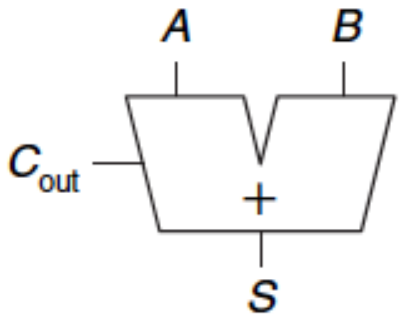
Joydeep Mitra

# Arithmetic Circuits

- Arithmetic circuits are fundamental building blocks of computers
- We will look at the implementations of a few basic arithmetic functions: addition, subtraction, and comparators
- We will then see how they can be combined to a single unit called the ALU

# Addition

- Let's consider the simple case of adding two bits
- This can be done by a *1-bit half adder*
  - Takes two inputs, `A` and `B`
  - Produces two outputs, `S` and $C_{out}$



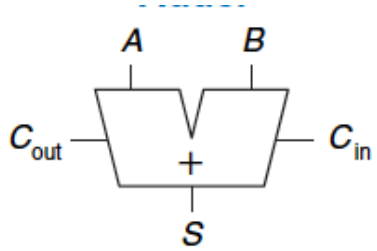

| A | B | $C_{out}$ | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

$$S = A \oplus B$$
$$C_{out} = AB$$

- The half adder can be built from an XOR gate and an AND gate

# Addition

- But what about adding more than 1 bit?
  - We need to build a multi-bit adder, where the carry out should propagate to the carry in of the next most significant bit
- This problem is solved by using the *full adder*
  - Takes three inputs, A, B, $C_{in}$
  - Produces two outputs, S and $C_{out}$
  - $C_{in}$ is $C_{out}$ of the previous bit



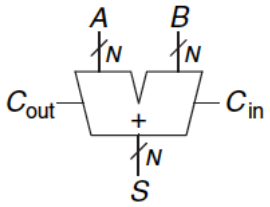

| $C_{in}$ | A | B | $C_{out}$ | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

- The *full adder* can be built from 2 half adders

# Addition

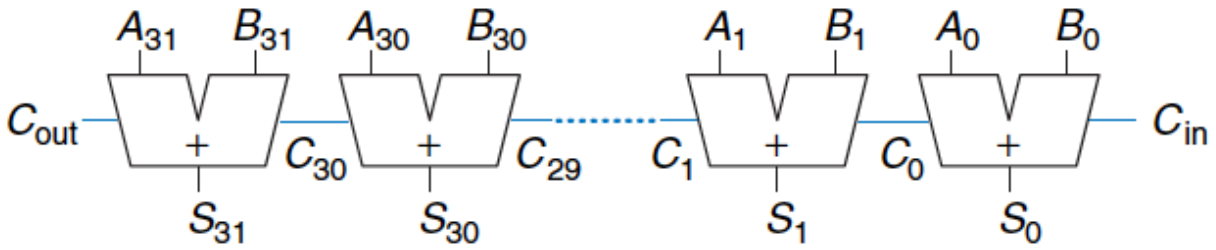
- An adder that performs multi-bit addition is called a *Carry Propagate Adder (CPA)*. Why?
  - Since the carry out of one bit propagates into the next bit as a carry in
  - It is represented as full adder, except that `A`,`B`, and `S` are actually buses and not single bits




- There are many ways of implementing a CPA; we will discuss one of them, the **ripple carry adder**

# Ripple Carry Adder

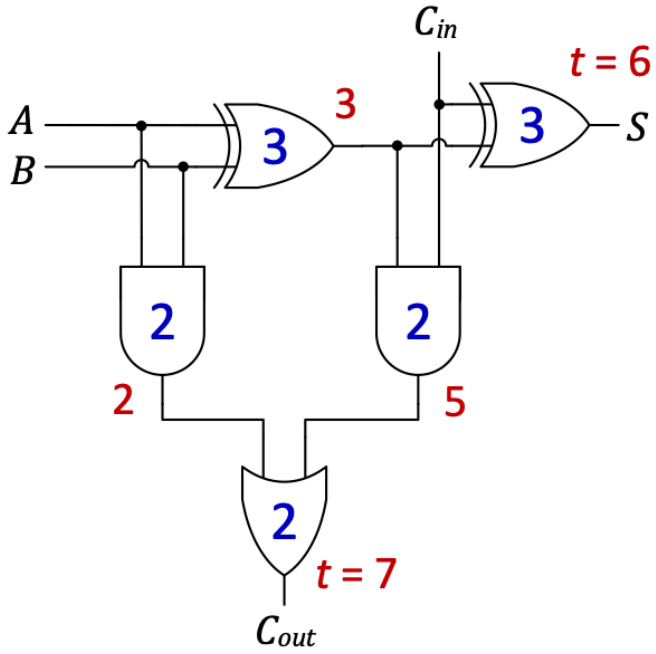
- A ripple carry adder is implemented by chaining together N full adders, each adder feeding a carry out to the carry in of the next adder
- The carry values ripple through from the lsb to the msb




- A limitation of this implementation is that it's slow. Why?
  - $S_{31}$ depends on $C_{30}$, which depends on $C_{29}$, and so on till $C_{in}$
  - Hence, the critical path grows with N, where N is the no. of full adders
  - Longer critical path implies longer (propagation) delays

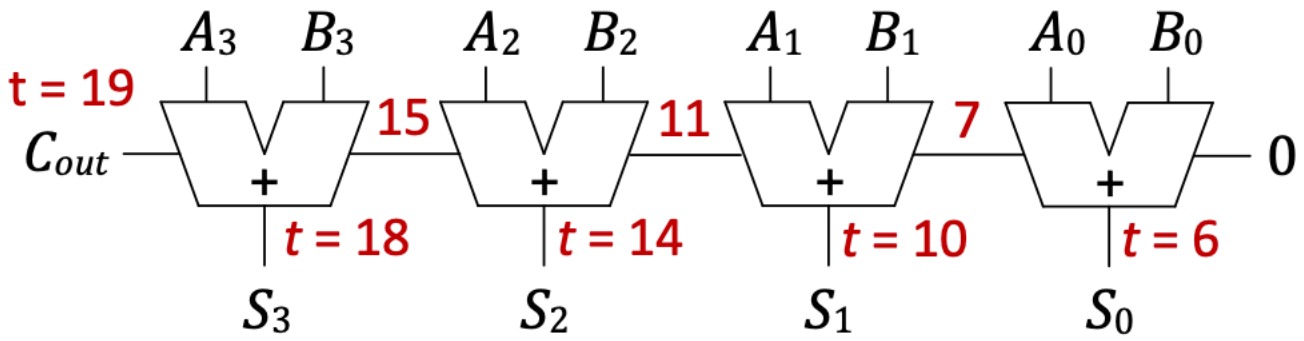
# Ripple Carry Adder

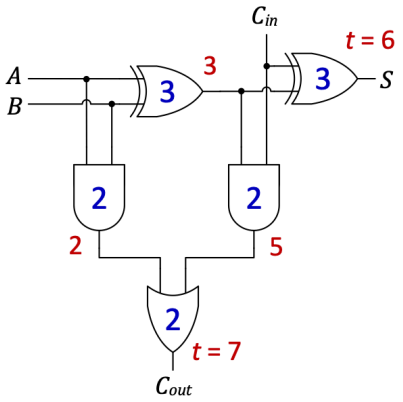
- If the propagation delays of the gates in a full adder are given to us, then what will be the propagation delay of one full adder circuit? Assume time in nanoseconds.



# Ripple Carry Adder

- Now, imagine the delay for a ripple carry adder for 4 bit addition

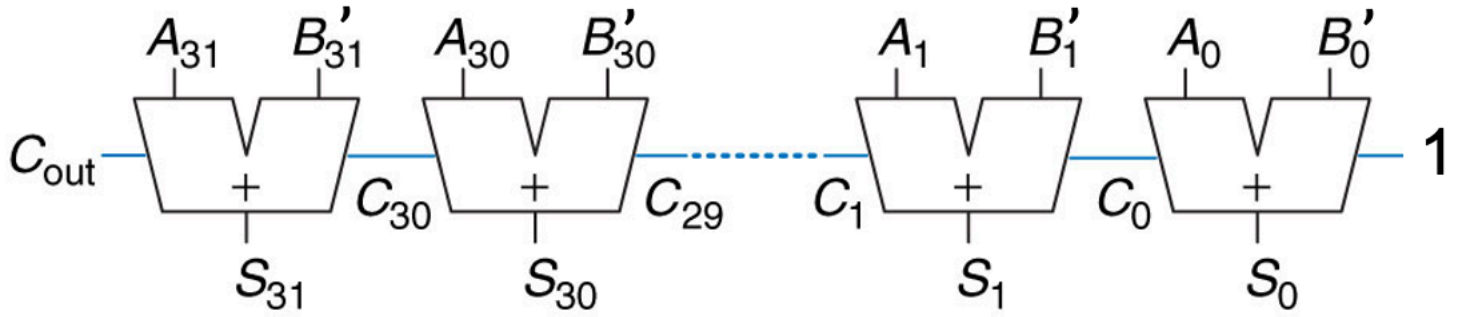



- The propagation delay is 19 ns
- An alternative implementation optimizes the delay by looking ahead at certain adders and predicting their sum and carry outs.

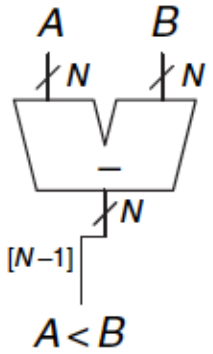
# Subtraction

- Subtraction is just addition of a positive number with a negative number
- We know that the binary negative number is represented as two's complement
- So, subtracting B from A is the same as adding the two's complement of B to A
  - `A – B = A + B′ + 1`



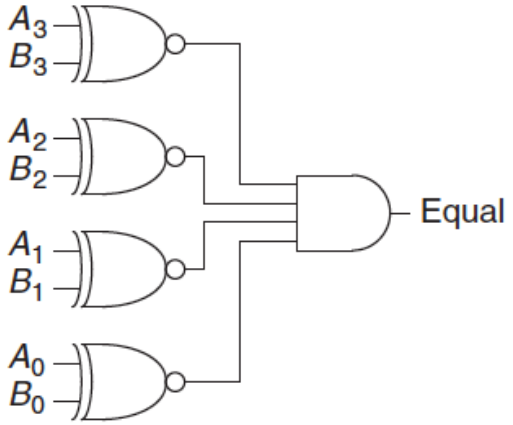
# Magnitude Comparison

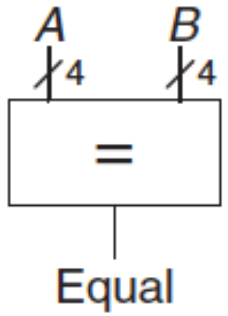
- Comparing two numbers A and B is the same as performing subtraction and checking the sign bit. E.g.,
  - `A > B` <=> 1 if sign bit of A – B is 0; 0 if sign bit of A – B is 1
  - `A < B` <=> 1 if sign bit of A – B is 1; 0 if sign bit of A – B is 0
- Basically, the output of a magnitude comparison is the value of the sign bit or its complement; depends on your requirement specification



# Equality Comparison

- Equality comparison between A and B requires us to compare every bit in A and B, a bit like addition
  - A = B if all bits are equal; if at least one is unequal, then A ≠ B
- Note that the XNOR gate can be used to identify if the two inputs are equal
- Every bit of A and B can be XNOR-ed and the output fed to an AND gate



# Building An ALU

- Until now, we have seen individual operations
- Building an ALU involves combining numerous operations (arithmetic and logical) into one unit
- General Idea:
  - Every operation in an ALU is identified by an **opcode**
  - A multiplexer is used to select between the different opcodes
  - Each operation is implemented by building an ALU for each bit of the operands in the operation
    - For N-bit operands, we will have N ALUs chained together; the output of 1 ALU may be the input of another ALU

# Example #1

- Consider the set of operations/functions in the table for which we will build an ALU
- $F_{2:0}$ is a 3-bit signal, which indicates the opcode for each operation
  - E.g., `000` indicates the ALU output should be `A AND B`
- Assume we have an N-bit adder, a two input AND gate and OR gate, and a NOT gate. We also have a 2:1 MUX and a 4:1 MUX

| $F_{2:0}$ | Function |
|---|---|
| 000 | A AND B |
| 001 | A OR B |
| 010 | A + B |
| 011 | not used |
| 100 | A AND $\overline{B}$ |
| 101 | A OR $\overline{B}$ |
| 110 | A – B |
| 111 | SLT |

# Example #1

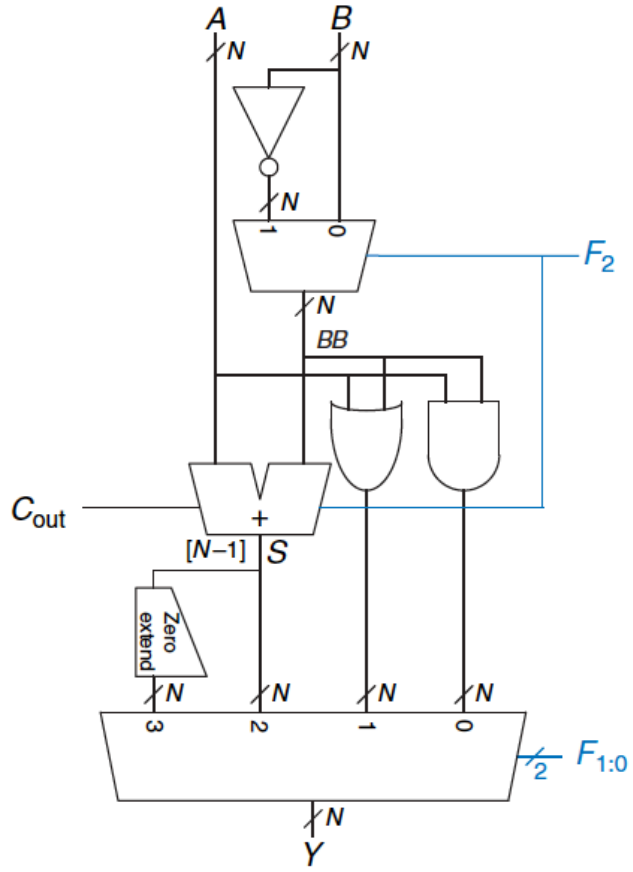
- We see that few operations need an adder, which needs an input $C_{in}$
- We also observe that the operations are similar (with difference in operands) when $F_2 = 0$ and $F_2 = 1$
- We re-arrange the table according to our observations

| $F_2$ | $F_{1:0}$ | Function | $C_{in}$(Bit 0) |
|---|---|---|---|
| 0 | 00 | A AND B | X |
| 0 | 01 | A OR B | X |
| 0 | 10 | A + B | 0 |
| 0 | 11 | Not used | X |

| $F_2$ | $F_{1:0}$ | Function | $C_{in}$(Bit 0) |
|---|---|---|---|
| 1 | 00 | A AND B’ | X |
| 1 | 01 | A OR B’ | X |
| 1 | 10 | A – B (A+B’+1) | 1 |
| 1 | 11 | SLT (A+B’+1) | 1 |

# Example #1

- We use the 4:1 MUX with $F_{1:0}$ control signals to select between the functions -- AND, OR, +, -
- We use the 2:1 MUX with $F_2$ to select between `B` and `B'`
- Note that $F_2$ and $C_{in}$ have the same values
- For the SLT function, we just need the *msb* of `A+B+1` as the *zero-extended* output



# ALU Flags

- Some ALUs produce extra outputs, called *flags*
- *Flags* indicate additional information about the output. E,g,.
  - An *overflow* flag that result has overflowed
  - A *zero* flag indicates that the output is 0
- The additional information from flags can be used to makes decisions about the final output or to select certain inputs for the next ALUs in the circuit

# Example #2

- Design a 3-bit ALU for the following functions:
  - We can only use 8:1 MUX, 2:1 MUX, half adders, and NOT gates inside an ALU

| C2 | C1 | C0 | Function |
|---|---|---|---|
| 0 | 0 | 0 | A XOR B |
| 0 | 0 | 1 | A AND B |
| 0 | 1 | 0 | A + B |
| 0 | 1 | 1 | A NAND B |
| 1 | 0 | 0 | SGTE (Set to 1 if A >= B) |
| 1 | 0 | 1 | SLT (Set to 1 if A < B) |
| 1 | 1 | 0 | A - B |
| 1 | 1 | 1 | A NOR B |

# Example #2

- First reason about the operands
  - Specifically, whether function operands need to be A,B,A', or B'
- Next, reason how to express each function with the available hardware

| C2 C1 C0 | Function | Translated Function |
|---|---|---|
| 0 0 0 | A XOR B | Sum output of half adder ( A + B) |
| 0 0 1 | A AND B | Carry out of half adder (A + B) |
| 0 1 0 | A + B | Sum of full adder (A+B; Carry out is Carry in of next bit) |
| 0 1 1 | A NAND B | (A AND B) XOR 1 |
| 1 0 0 | SGTE (Set to 1 if A >= B) | NOT SLT |
| 1 0 1 | SLT (Set to 1 if A < B) | A + B' + 1 (consider only MSB) |
| 1 1 0 | A - B | A + B' + 1 |
| 1 1 1 | A NOR B | A'B' (Carry out of half adder) |

# Example #2

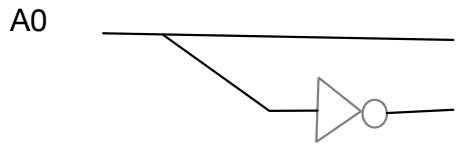
- We need control/select signals to select between operands that use the same hardware
  - **Ainv** selects between A and A'
  - **Binv** selects between B and B'
  - **K** selects the input for the 2nd half adder
  - **Less** selects between SGTE and SLT

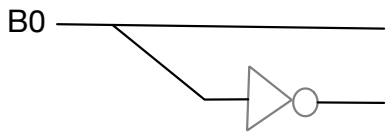
| C2 | C1 | C0 | Function | Ainv | Binv | K | Less | Cin (LSB) |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | A XOR B | 0 | 0 | X | X | X |
| 0 | 0 | 1 | A AND B | 0 | 0 | X | X | X |
| 0 | 1 | 0 | A + B | 0 | 0 | 0 | X | 0 |
| 0 | 1 | 1 | A NAND B | 0 | 0 | 1 | X | X |
| 1 | 0 | 0 | SGTE (Set to 1 if A >= B) | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | SLT (Set to 1 if A < B) | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | A - B | 0 | 1 | 0 | X | 1 |
| 1 | 1 | 1 | A NOR B | 1 | 1 | X | X | X |

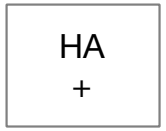
$$Ainv = C_2C_1C_0 \qquad Binv = C_2$$
$$K = C_1C_0 \qquad Less = C_0$$
$$Cin\ (LSB) = C_2$$

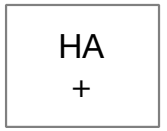
# Example #2

Available Hardware



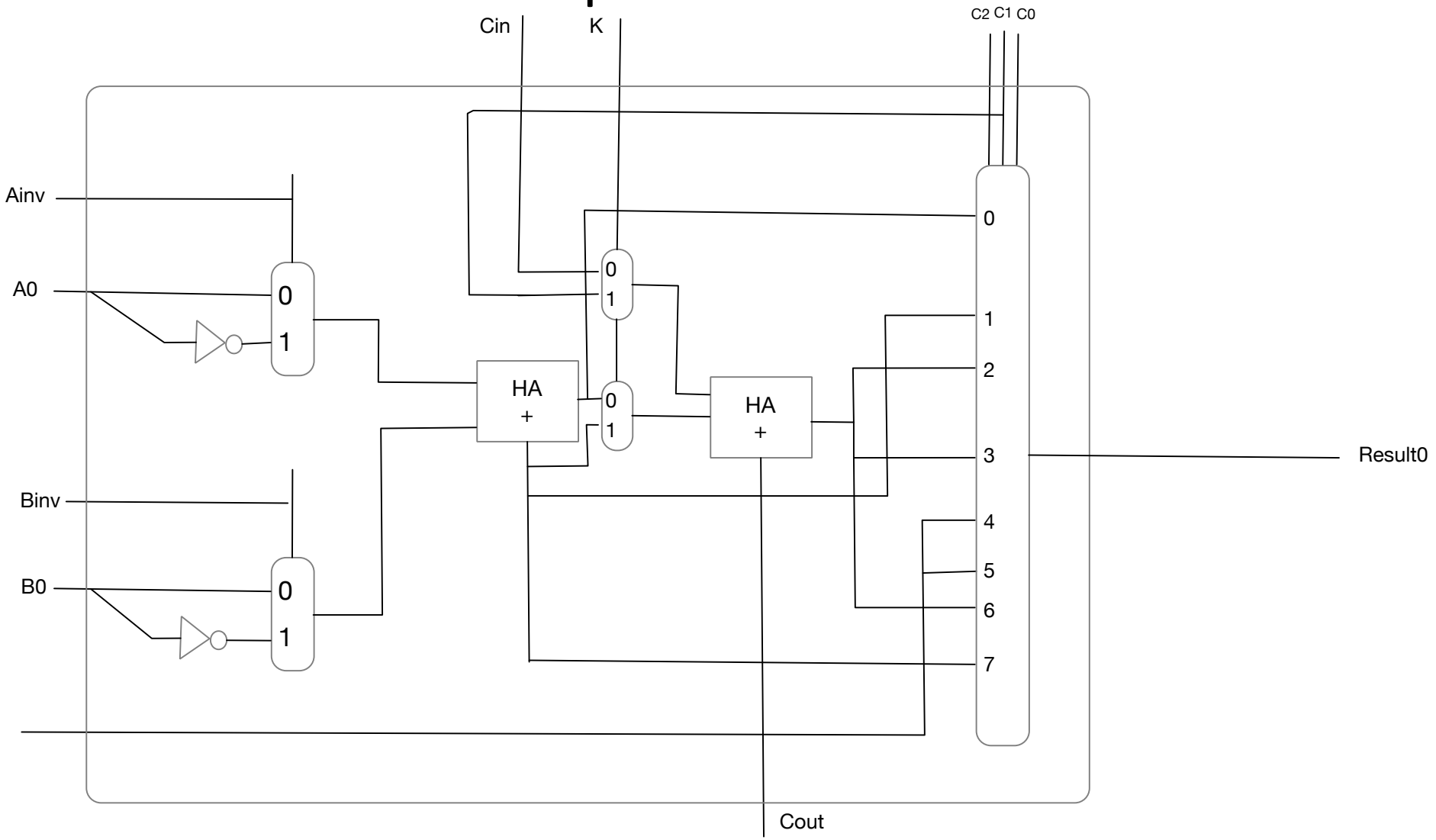



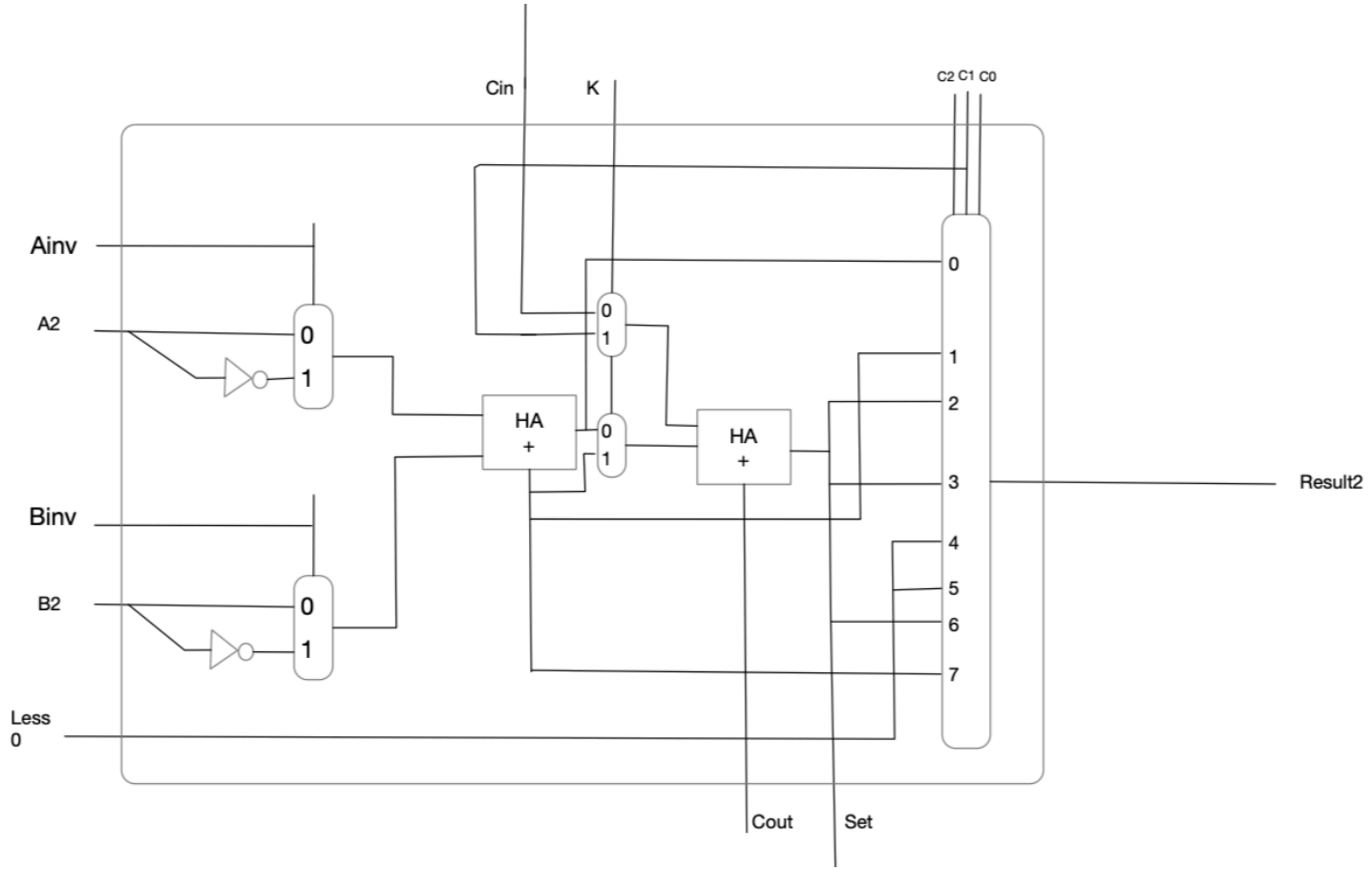

# Example #2

- LSB ALU



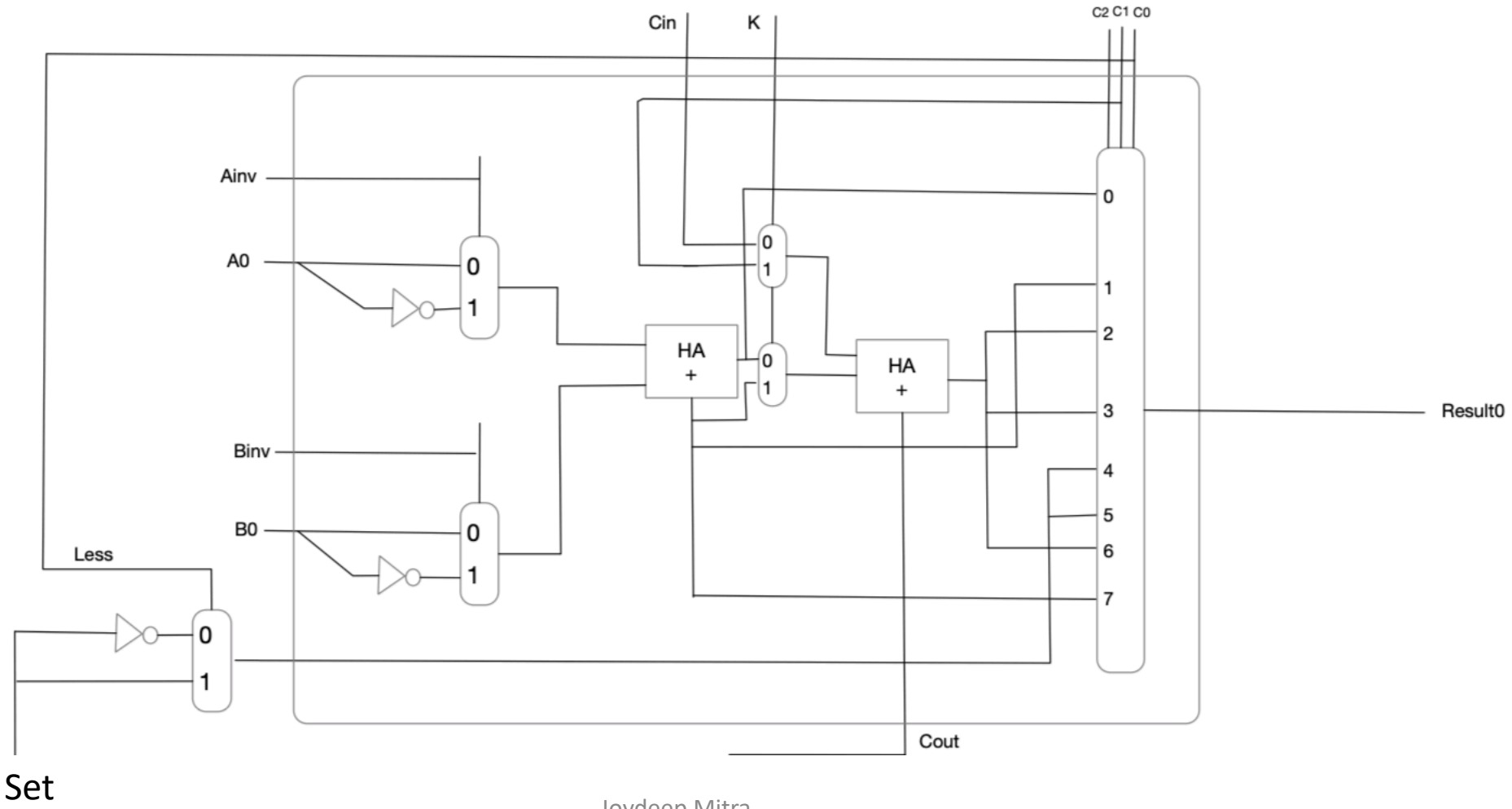
# Example #2

- The MSB will be the same as LSB except we will have to wire the comparator functions appropriately



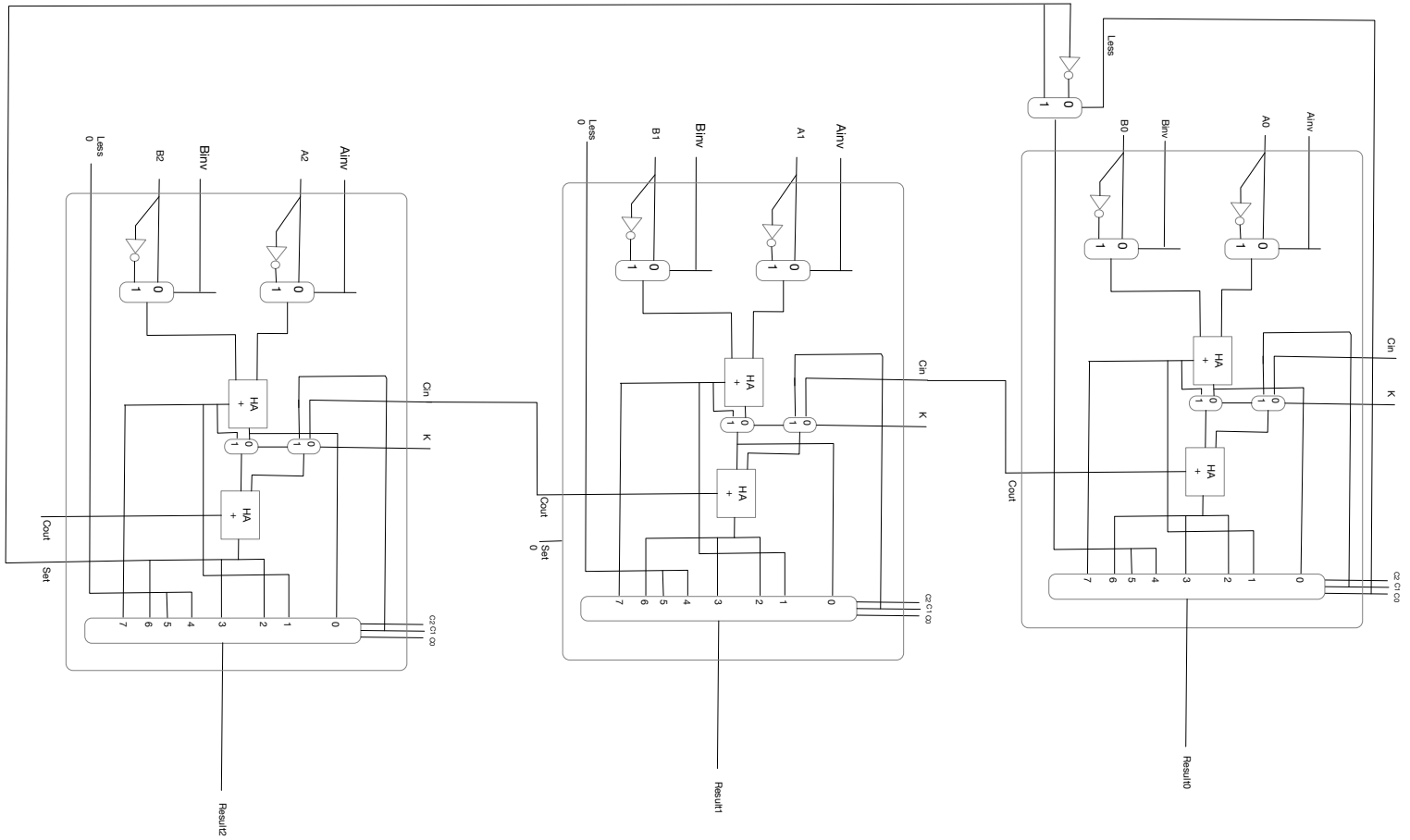
# Example #2

- The LSB ALU needs to be modified to account for the Set bit from MSB



# Example #2

- A snapshot of the 3-bit ALU



# ALU Recap

- Arithmetic circuits are fundamental building blocks of computers
- We have seen implementations of a few basic arithmetic functions: addition, subtraction, and comparators
- We have seen examples of how ALUs can be used to implement a multitude of operations

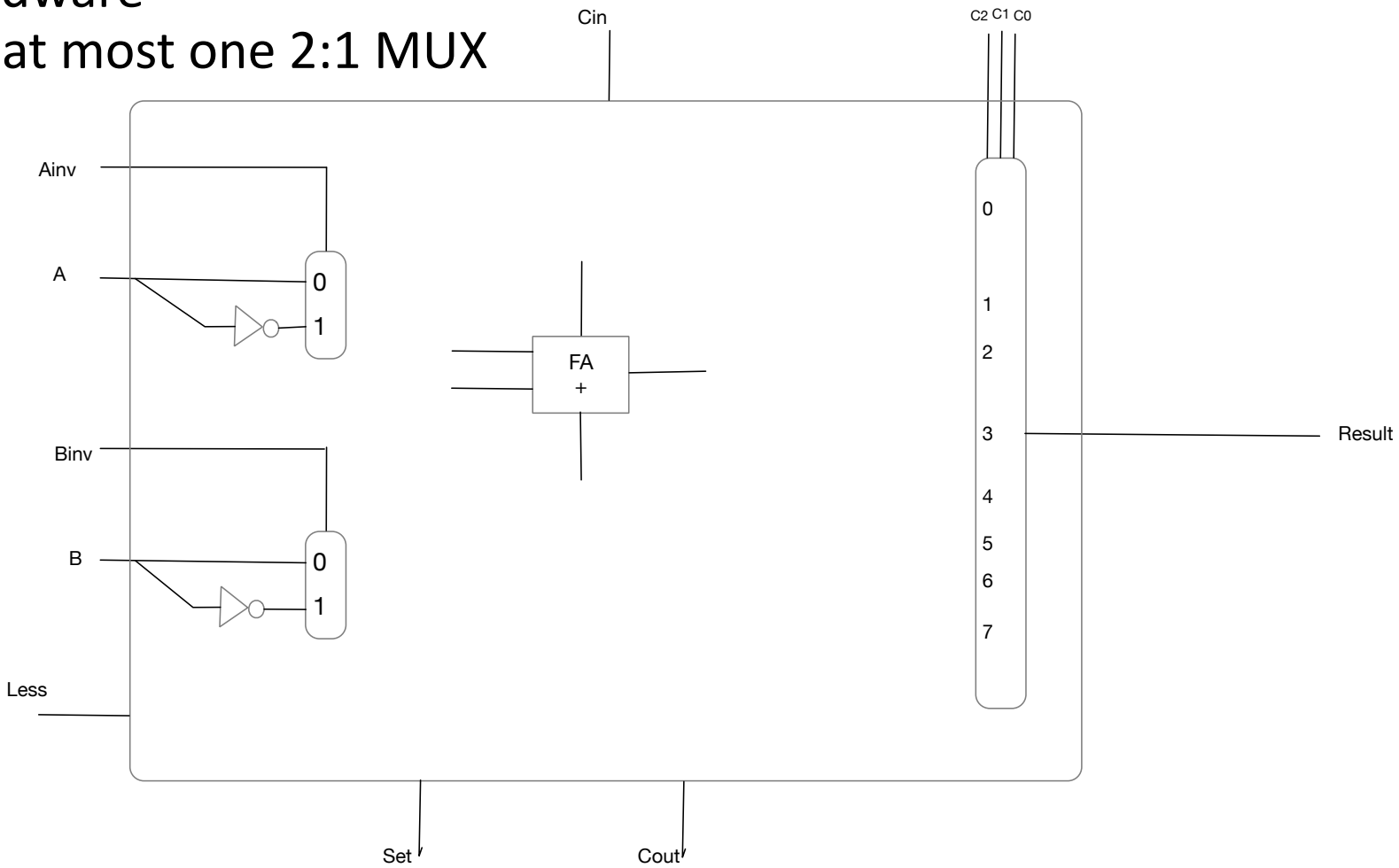
# Example #3

- Design the LSB and MSB ALU for the following functions

| C2 | C1 | C0 | Function |
|---|---|---|---|
| 0 | 0 | 0 | A + B |
| 0 | 0 | 1 | A NOR B |
| 0 | 1 | 0 | SGT (A > B) |
| 0 | 1 | 1 | unused |
| 1 | 0 | 0 | A AND B |
| 1 | 0 | 1 | A XOR B |
| 1 | 1 | 0 | A - B |
| 1 | 1 | 1 | Set Even ( A is even) |

# Example #3

Available Hardware
- we can add at most one 2:1 MUX



# Example #3

- Express the functions in available hardware

| C2 C1 C0 | Function | Translated Function |
|---|---|---|
| 0 0 0 | A + B | Full Adder with A, B, Cin (Sum) |
| 0 0 1 | A NOR B | Full Adder with A', B', Cin = 0 (Carry out) |
| 0 1 0 | SGT (A > B) | Full Adder with A', B, Cin = 1 (MSB of Sum) |
| 0 1 1 | unused | Ground |
| 1 0 0 | A AND B | Full Adder with A, B, Cin = 0 (Carry out) |
| 1 0 1 | A XOR B | Full Adder with A, B, Cin =0 (Sum) |
| 1 1 0 | A - B | Full Adder with A, B, Cin = 1 (Sum) |
| 1 1 1 | Set Even ( A is even) | LSB of A' |

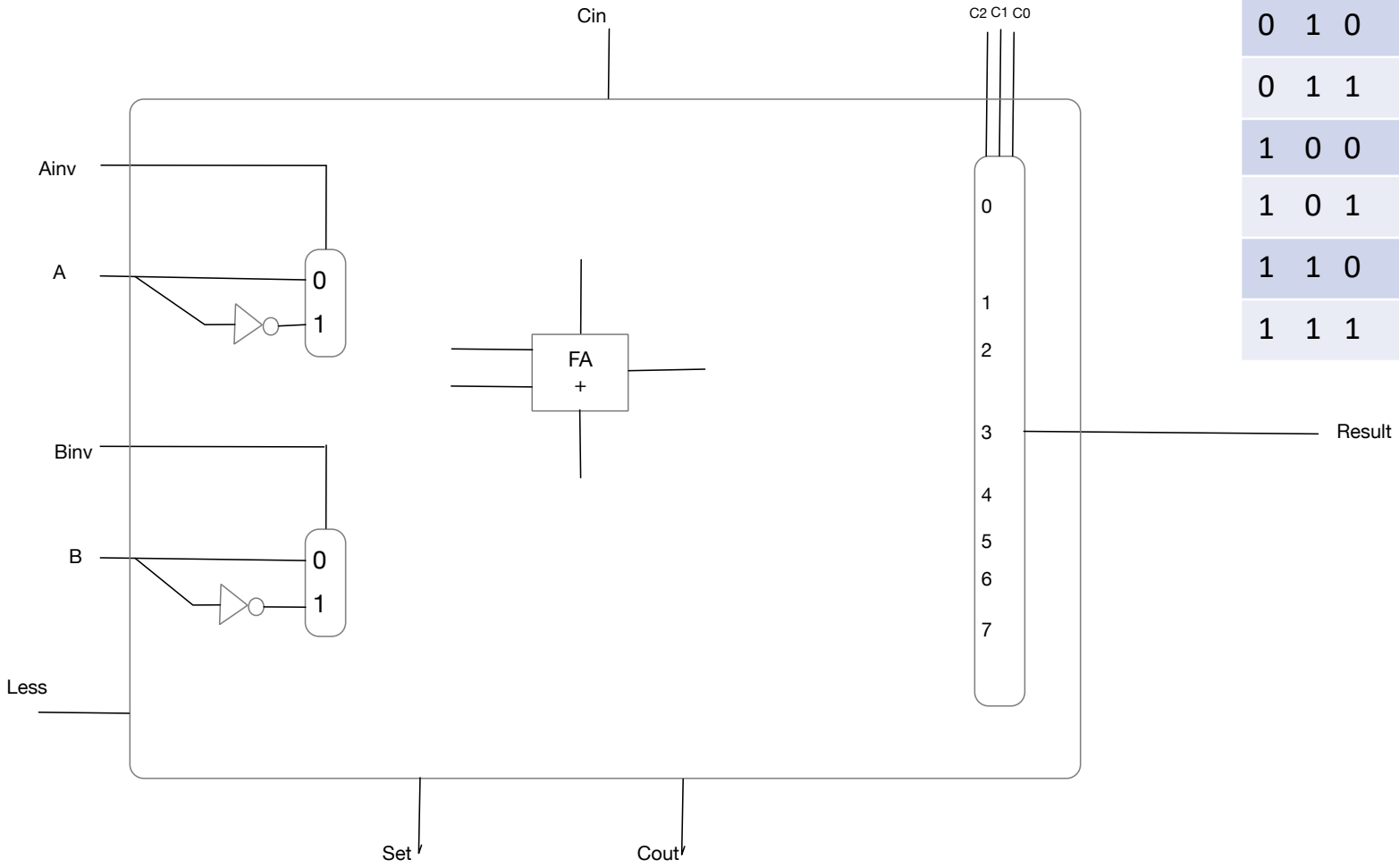
# Example #3

- We need the following control/select signals to select between operands that use the same hardware
  - **Ainv** to select between A and A'
  - **Binv** to select between B and B'
  - **S** to select between arithmetic and logical functions

| C2 | C1 | C0 | Function | Ainv | Binv | S | Cin (LSB) |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | A + B | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | A NOR B | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | SGT (A > B) | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | unused | X | X | X | X |
| 1 | 0 | 0 | A AND B | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | A XOR B | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | A - B | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | Set Even (A is even) | 1 | X | X | X |

$$\mathbf{Ainv} = C_2'(C_0 + C_1) + C_1C_0$$
$$\mathbf{Binv} = C_2'C_0 + C_2C_1$$
$$\mathbf{S} = C_2C_1' + C_0$$
$$\mathbf{Cin} = C_1$$

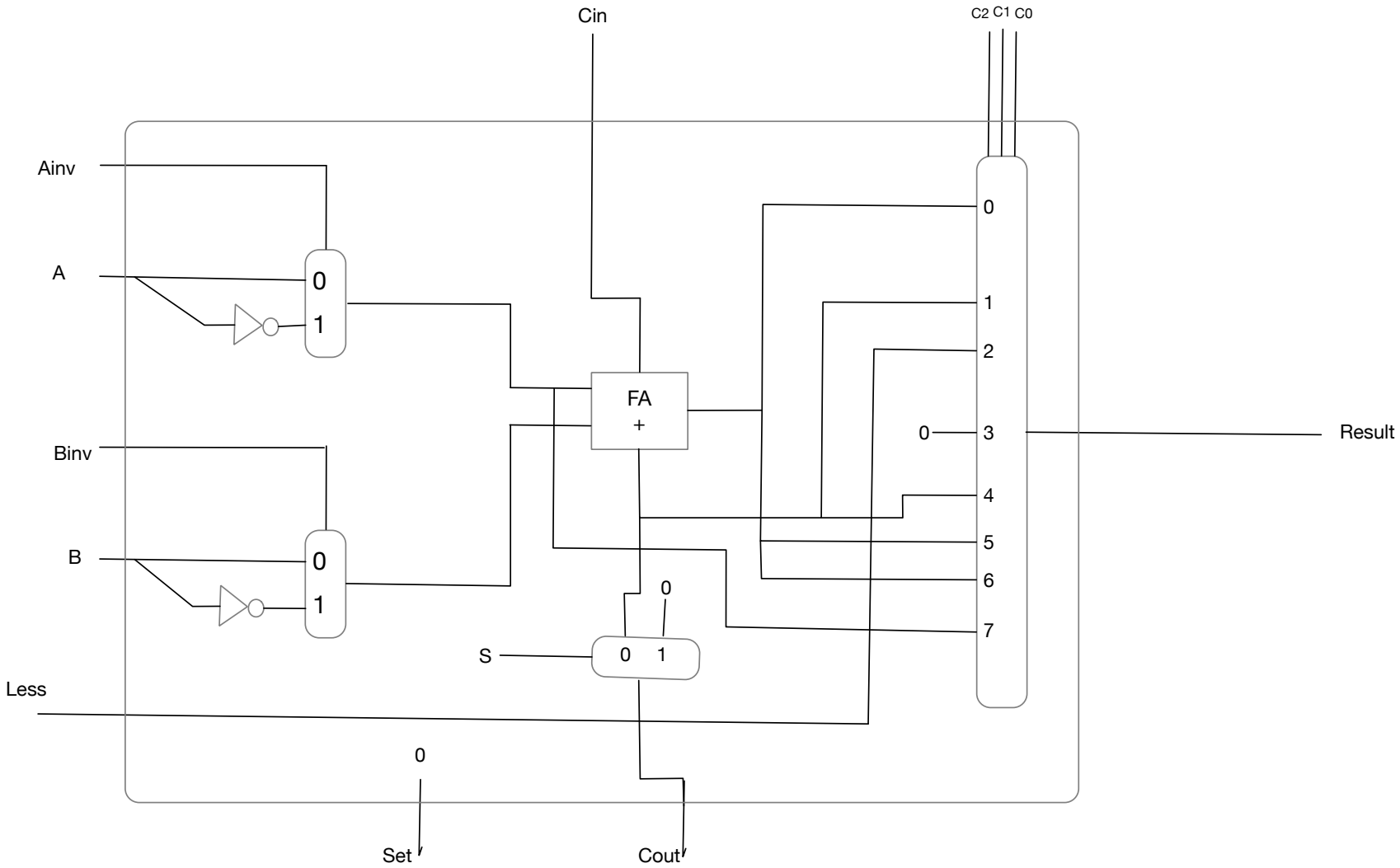
# Example #3




| C2 C1 C0 | Function | Ainv | Binv | S | Cin |
|---|---|---|---|---|---|
| 0 0 0 | A + B | 0 | 0 | 0 | 0 |
| 0 0 1 | A NOR B | 1 | 1 | 1 | 0 |
| 0 1 0 | SGT (A > B) | 1 | 0 | 0 | 1 |
| 0 1 1 | unused | X | X | X | X |
| 1 0 0 | A AND B | 0 | 0 | 1 | 0 |
| 1 0 1 | A XOR B | 0 | 0 | 1 | 0 |
| 1 1 0 | A - B | 0 | 1 | 0 | 1 |
| 1 1 1 | Set Even (A is even) | 1 | X | X | X |

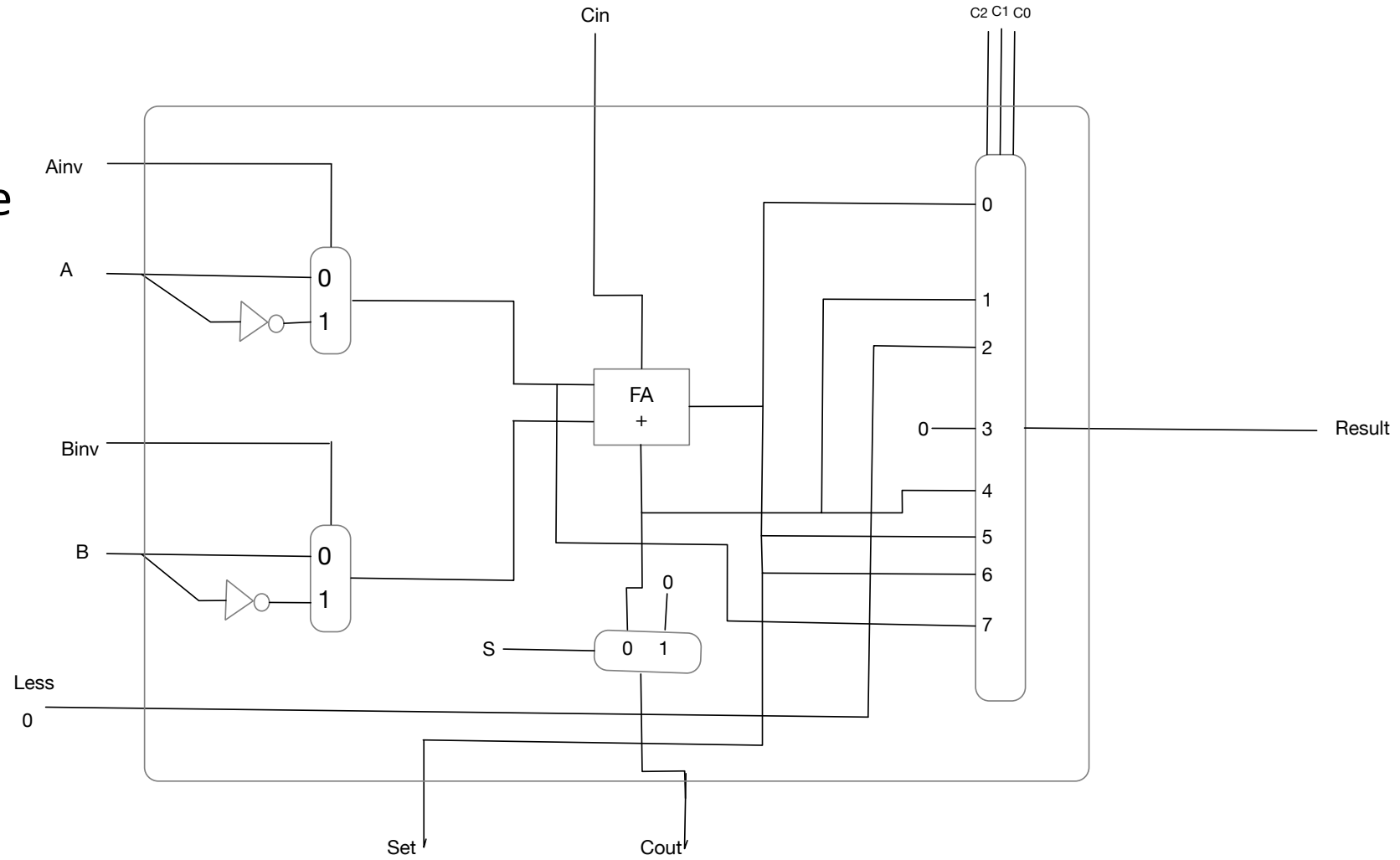
# Example #3

The LSB ALU



# Example #3

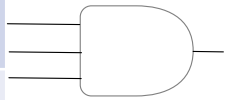
- The MSB ALU is the same as LSB except the **Set** signal is connected to the Sum output of the full adder
- The Set output is connected back to the LSB **Less** signal
- The MSB **Less** signal is 0

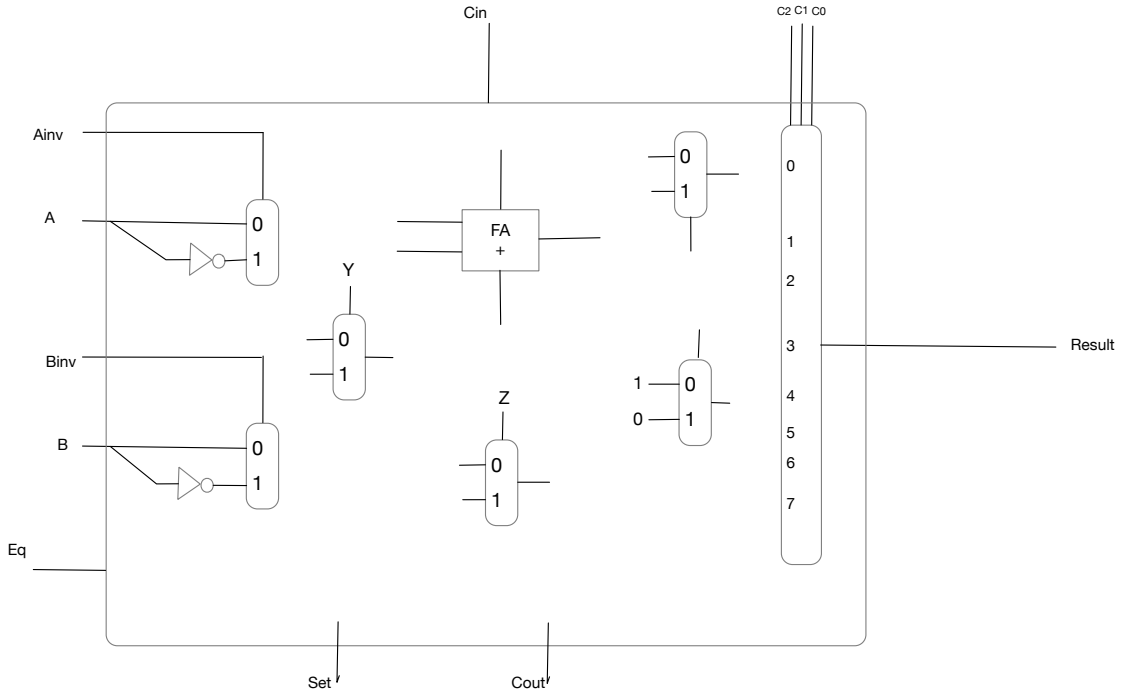


# Example #4

- Design a 3-bit ALU for the following functions

| C2 | C1 | C0 | Function |
|---|---|---|---|
| 0 | 0 | 0 | SEQ (A = B) |
| 0 | 0 | 1 | A NOR B |
| 0 | 1 | 0 | A OR B |
| 0 | 1 | 1 | A AND B |
| 1 | 0 | 0 | A - B |
| 1 | 0 | 1 | A + B |
| 1 | 1 | 0 | A XOR B |
| 1 | 1 | 1 | 2A |






**Available Hardware**

# Example #4

- Boolean Equations for Control/Select signals

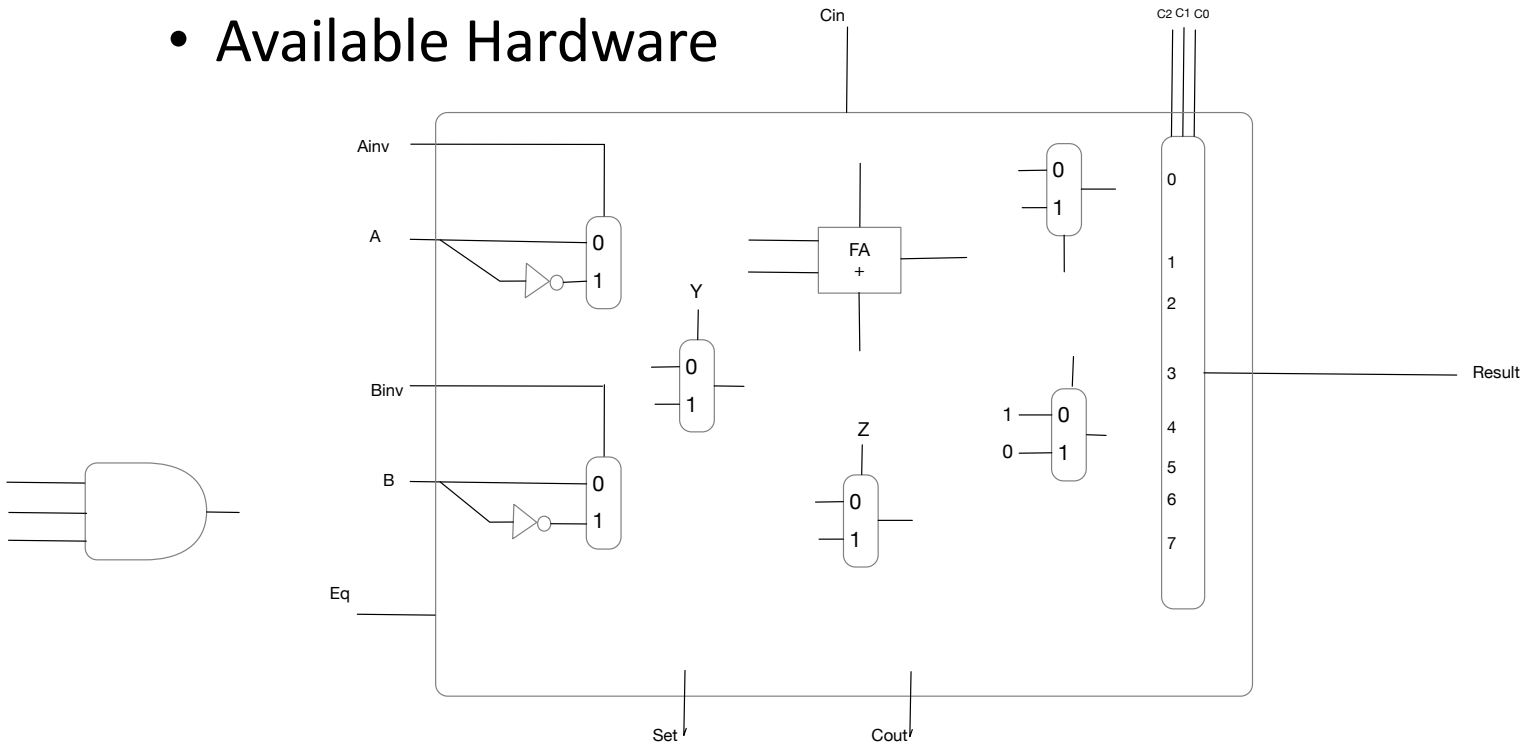
| $C_2$ $C_1$ $C_0$ | Function | $A_{inv}$ | $B_{inv}$ | Y | Z | $C_{in}$(LSB) |
|---|---|---|---|---|---|---|
| 0 0 0 | SEQ (A = B) | 0 | 0 | 0 | 1 | 0 |
| 0 0 1 | A NOR B | 1 | 1 | 0 | 1 | 0 |
| 0 1 0 | A OR B | 1 | 1 | 0 | 1 | 0 |
| 0 1 1 | A AND B | 0 | 0 | 0 | 1 | 0 |
| 1 0 0 | A - B | 0 | 1 | 0 | 0 | 1 |
| 1 0 1 | A + B | 0 | 0 | 0 | 0 | 0 |
| 1 1 0 | A XOR B | 0 | 0 | 0 | 1 | 0 |
| 1 1 1 | 2A | 0 | X | 1 | 0 | 0 |

$$A_{inv} = C_2'(C_1 \text{ XOR } C_0)$$
$$B_{inv} = C_2'(C_1 \text{ XOR } C_0) + C_2C_1'C_0'$$
$$Y = C_2C_1C_0$$
$$Z = C_2' + C_1C_0'$$
$$Cin = C_2C_1'C_0'$$

- Y is used to select between A and B
- Z is used to select between arithmetic and logical operations

# Example #4

- Available Hardware




| C2 C1 C0 | Function | Ainv | Binv | Y | Z | Cin |
|---|---|---|---|---|---|---|
| 0 0 0 | SEQ (A = B) | 0 | 0 | 0 | 1 | 0 |
| 0 0 1 | A NOR B | 1 | 1 | 0 | 1 | 0 |
| 0 1 0 | A OR B | 1 | 1 | 0 | 1 | 0 |
| 0 1 1 | A AND B | 0 | 0 | 0 | 1 | 0 |
| 1 0 0 | A - B | 0 | 1 | 0 | 0 | 1 |
| 1 0 1 | A + B | 0 | 0 | 0 | 0 | 0 |
| 1 1 0 | A XOR B | 0 | 0 | 0 | 1 | 0 |
| 1 1 1 | 2A | 0 | X | 1 | 0 | 0 |

# Example #4

- LSB ALU
- The middle bit ALU and MSB ALU are the same as the LSB ALU