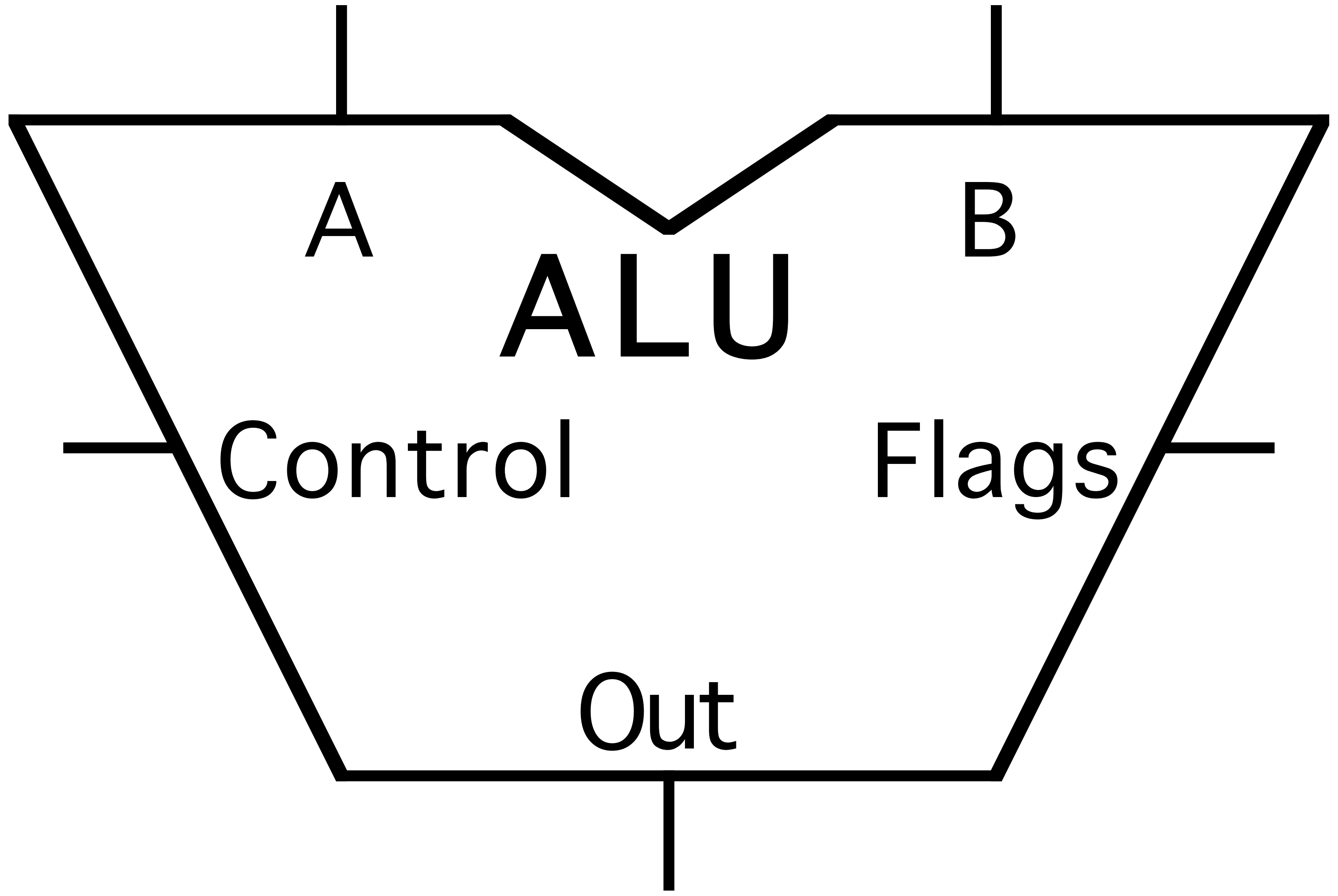
CS 201

Digital Logic

# Arithmetic and Logic

Mano chapter 3

# ALU




- Arithmetic Logic Unit
  - Arithmetic: ADD, SUB, MULT, DIV
  - Logic: AND, OR, NOT, XOR
- Mathematical "heart" of the computer
- We know how to built a circuit that can add, and do logic, but how to do other arithmetic?
  - how to subtract? how to represent negative numbers?
  - how to multiply?
  - how to divide? how to represent fractional numbers?

# Converting decimal to binary

- Binary to decimal is easy. Powers of two
- Decimal to binary is harder.
- Successive division by 2.
- Then read the remainders from bottom to top:
  - example: what is 307 in binary?

```
307 ÷ 2 = 153 r 1
153 ÷ 2 =  76 r 1
 76 ÷ 2 =  38 r 0
 38 ÷ 2 =  19 r 0
 19 ÷ 2 =   9 r 1
  9 ÷ 2 =   4 r 1
  4 ÷ 2 =   2 r 0
  2 ÷ 2 =   1 r 0
  1 ÷ 2 =   0 r 1
```

```
100110011 = 256 + 32 + 16 + 2 +
          = 307
```

# Converting decimal to binary

- convert $41_{10}$ to binary

$41 \div 2 = 20$ r 1
$20 \div 2 = 10$ r 0
$10 \div 2 = 5$ r 0
$5 \div 2 = 2$ r 1
$2 \div 2 = 1$ r 0
$1 \div 2 = 1$ r 1

101001

confirm: $32 + 8 + 1 = 41$

Practice: pick a random number between 0 and 100 and convert it. Then confirm.

# Some Terminology

- MSB: Most Significant Bit
  - The leftmost bit of a stream
- LSB: Least Significant Bit
  - The rightmost bit of a stream

- e.g. 10010100
  - MSB is 1, corresponding to $1 \times 2^7$
  - LSB is 0, corresponding to $0 \times 2^0$

# Binary arithmetic: remember the Full Adder

- Addition: Just like high school
  - Line up similar place values
  - Any result bigger than will fit in the column results in a carry into the next column
  - decimal: 5+7=12 -> 2 with 1 carried to the the 10s column
  - binary:   1+1=10 -> 0 with 1 carried to the 2s column

```
(1) (1) (1)          (1)       carries

     1   0   1          5

   + 1   1   1         +7
 ----------------    -----
 1   1   0   0       1  2
```

# Binary arithmetic: Subtraction

- Subtraction: Just like addition
  - Line up similar place values
  - Borrow from the next column if necessary

$$
\begin{array}{ccccc}
 & & & (1) & \\
 & & (10) & (\cancel{10}) & (10) \\
 & \cancel{1} & \cancel{1} & 0 & 0 \\
 & - & 1 & 1 & 1 \\
\hline
 & & 1 & 0 & 1
\end{array}
\qquad\qquad
\begin{array}{cc}
 & (10) \\
\cancel{1} & 2 \\
- & 7 \\
\hline
 & 5
\end{array}
\qquad \text{borrows}
$$

  - 1s column needs to borrow, but nothing in 2s column, so borrow from 4s column.

# Subtraction: Bitslice Truth Table D = X-Y

- (in this slide + is plus, not OR)
- $B_{in}$ means a column to the right has borrowed from this column,
  - so subtract 1 from X-Y
- $B_{out}$ is a request to borrow from the *next* column to the left
  - so add 2 (binary 10) to X-Y
- if $X < (Y + B_{in})$ then borrow from the next column ($B_{out}$ =1).
- then $D = 2 \times B_{out} + X - Y - B_{in}$

| X | Y | $B_{in}$ | $B_{out}$ | D |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# Subtraction: the bitslice functions

- $D = X \oplus Y \oplus B_{in}$,
- $B_{out} = \overline{X}\,\overline{Y}B_{in} + \overline{X}Y\overline{B_{in}} + \overline{X}YB_{in} + XYB_{in}$
  $= \overline{X}(Y \oplus B_{in}) + YB_{in}$
    - ➡ exercise: prove with k-maps
    - ➡ does it look familiar?
- But what happens if a borrow fails (i.e. what if X<Y)?
- if X<Y, the result will be negative
  - What does that mean in binary?
  - Tangent time…

# Number Representation in Binary

- Recall unsigned representation:
  - Each bit means '$+a \times 2^n$'
  - Always positive, since *a* can only be 0 or 1
- What about negative numbers?
  - We need ***signed numbers*** for subtraction.
- 3 common signed representation options, in order from obvious to useful:
    - Sign-magnitude
    - Complement
    - 2's complement

# Sign-Magnitude

- In decimal, we use extra symbols for the sign
  - ' – ' means negative, ' + ' means positive
- In binary, we can't just add extra symbols
  - only have "0" and "1"
  - So use an extra bit for the sign
  - Adding 0 at the beginning should not change the value
    - so 0 means positive, and 1 means negative.
- For example,
  - $0101_2 = +5_{10}$, and $1010_2 = -2_{10}$

# Subtraction with Sign-Magnitude

1. Check sign of both operands
2. Perform appropriate math
   - e.g.: $-x - y$ would be treated as $-(x + y)$,
   - add $x$ to $y$, then add "1" to make the result negative.

- Lots of logic to design
  - Check sign,
  - Logic for each sign combination and operation
    - 4 possibilities for add, 4 for subtract

# Complement

- if X is the binary encoding of the number, let $\overline{X}$ be the negative version of that
  - Negating is the equivalent inverting, seems intuitive
  - Need very different representations for negative and positive:
    - 5 = 0101
    - -5 = 1010
      - Each bit no longer corresponds to '$+a \times 2^n$'
      - Need a second step to figure out value

# 2's Complement

- Binary version of the "Radix Complement"
  - Subtract from the next higher power of the radix
- We'll do 10's complement first to get the feel.
- To form the 10's complement:
  - Subtract from the next higher power of 10
  - e.g.: 10's complement of 209 is **1000** − 209 = 791
  - so 791 means −209 in 10's complement
- The goal is to be able to subtract by adding
  - For example, $a - b = a + (-b)$

# 10's Complement subtraction by adding

- Use 10's complement for subtraction:

  $315 - 209$

  $= 315 + 1000 - 209 - 1000$

  - $1000 - 209 = 791$ is ten's complement for $-209$
  - the extra " $- 1000$" maintains the numerical value

  $= 315 + 791 - 1000$

  $= 1106 - 1000$

  $= 106$, which is the correct answer.

- Adding is easy, and dropping the extra 1000 is easy.
- Is finding 791 easy?

# Easily find the 10's complement

- to find the simple complement, subtract each bit from the biggest digit in the representation
  - in base 10, that value is 9
  - also called 9's complement
    - The 9's complement of 209 would be 790
- The 10's complement is just the 9's complement, plus 1

- So finding the 10's complement is easy, adding is easy, and dropping the next higher power of 10 is easy, so we can subtract without subtracting

# Now, in Binary

- Subtract from next higher power of 2.

  e.g. $43_{10} = 0101011_2$.
  - the two's complement representation of -43 is:
    - 10000000 – 0101011 = 1010101

```
 10000000
- 0101011
  1010101
```

- Can we form it without using subtraction, using the trick from 10's complement?
- Look at the subtraction table again:

# Forming the 2's complement

$X - Y = 10000000 - 0101011$

- Except for the MSB, the bits of X are always 0, which means repeated borrows.
- If we could make it so that *X* is always 1, there would be no borrows, implying $D = Y'$
- Notice that 1000000 – 1 = 111111

| X | Y | $B_{in}$ | $B_{out}$ | D |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# Forming the 2's complement

- 2's complement of $43_{10}$

  10000000 – 0101011 = 1010101

  (10000000 – 1) + 1 – 0101011 = 1010101

  1111111 – 0101011 + 1= 1010101

  (0101011)' + 1 = 1010101
- So, to form 2's complement, take the logical complement and add 1.
    - (just like in the 10's complement)
  - No subtraction necessary!

# 2's complement for subtraction

- e.g.: $50_{10} - 43_{10}$ =0110010 – 0101011 = 0111 = $7_{10}$
- Using the 2's complement idea:

*complement and add 1*

```
        0110010 - 0101011
=       0110010 + 1010101 - 1000000
=            1000111      - 1000000
=               0111
```

- We still have to subtract out the next higher power of 2 to maintain numerical equivalence.

# 2's complement representation

- To make this idea workable, we need
  1. Some way to identify positive or negative numbers
  2. Some way to easily subtract out the next higher power of 2.

- Solution:
  1. Add another bit, same as in signed-magnitude, in the MSB location
  2. This bit will correspond to $-a \times 2^k$
  3. All other bits correspond to $+a \times 2^k$

# 2's complement representation

- e.g.: 7 bit representation: $-2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$
  - For small enough positive numbers:

    `0010101 = 16 + 4 + 1 = 21`

  - For negative numbers:

    $-43 = -1000000 + (1000000 - 101011)$

    $= -1000000 + 0010101$

    $= -2^6 + 2^4 + 2^2 + 2^0 = -64 + 16 + 4 + 1 = -43$

- So the complete representation is **1**010101 and can be formed simply by inverting the positive and adding 1:
  - 43 = 0101011 → – 43 = 1010100 + 1 = 1010101

# Limits of 7-bit 2's complement

- 0000000 is still $0_{10}$. That's good.
- The largest positive number is when the negative bit is 0, and all positive bits are 1:
  - 0111111, which is $63_{10}$.
- The largest negative number is when the negative bit is 1, and all positive bits are 0:
  - 1000000, which is $-64_{10}$.
- All 1's equals −1: 1111111 = −64 + 63 = −1

# Range of 2's complement representation

- Since $|-64| > |63|$,
  - All positive numbers have MSB = 0
  - All negative numbers have MSB = 1
  - So we can interpret the MSB as a sign bit!
- For 7 bits, range is $-64..+63$
- In general, for $n$ bits, range is
  - $-2^{n-1}$ to $2^{n-1}-1$

# Confirmation of 2's complement representation

- Positive numbers now require a leading zero.
  - e.g. 0111 = +7, 111 = −1
- The size of the representation is important
  - The MSB is negative and all other bits are positive
- −(−*N*) should equal +*N*
- e.g.: $-(-17_{10})$
  - +17 = 16+1 = 010001
  - −17 = 101110+1 = 101111
  - −(−17) = 010000+1 = 010001

*hooray!*
*wait, what?*

# The wierness of 2's complement

- If you flip the bits and add 1 to form the negative, you should have to subtract the bits before you flip, to undo that change.
- but flipping the bits and adding 1, twice, gets you back to the same value you started with
  - So for any X, $\overline{X}+1 = (X-1)'$
- Note: 2's complement is both a noun (the name of our representation) and a verb (to flip the bits and add 1)
- Remember, positive numbers are unchanged

# 2's complement: doing math

- 2's complement should allow us to subtract by adding (that was the goal)
- Try a bit-by-bit addition of a negative number
  - e.g. 50−43 = 50+(−43) = 7

| | | $-2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|
| 50 | $^1$ | $^1$0 | $^1$1 | 1 | 0 | 0 | 1 | 0 |
| -43 | + | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 7 | (1) | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

  - Note: Ignore the final carry out for the time being

# Some Practice

- Convert the following to 2's complement

  (use a 7–bit representation):

  25 =

  −11 =

- Do the following math using 2's complement

  25 + 11 =

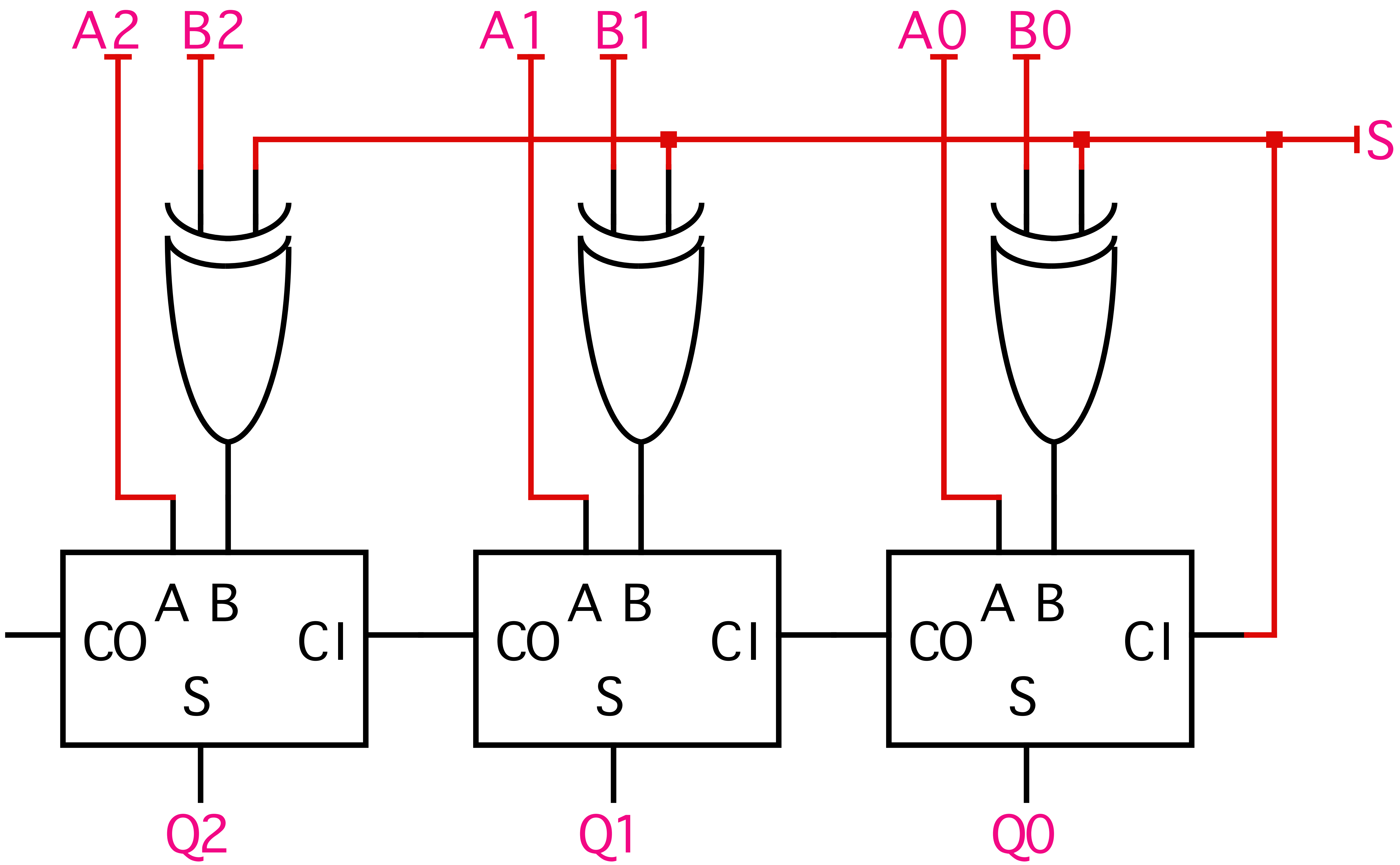
  25 – 11 =

  11 – 25 =

# 2's complement add/sub circuit

- To add, use full adder
- To subtract, add the complement + 1

S = 0:  Q = A + B
S = 1:  Q = A – B
XOR is used as a controlled complement (recall the truth table).
Ci = S adds the required "1" when subtracting.



# Overflow

- Using a 7-bit representation, consider 50 + 21 = 71

| | $-2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|
| 50 | $^{1}$0 | $^{1}$1 | 1 | 0 | 0 | 1 | 0 |
| +21 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 71 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |

- The result will be wrongly interpreted as –57
- Carry from the $+2^5$ column into the into the $-2^6$ column.
  - Doesn't make sense. columns mean different things
- ***Arithmetic Overflow***: The result of an operation is too large for the representation.
- NOT THE SAME AS carry out.

# Overflow

- Consider − 50 − 21= − 71

| | | | $-2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|---|
| −50 | | 1 | 1 | 0 | $^1$0 | $^1$1 | $^1$1 | 1 | 0 |
| −21 | | | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| −71 | | (1) | 0 | 1 | 1 | 1 | 0 | 0 | 1 |

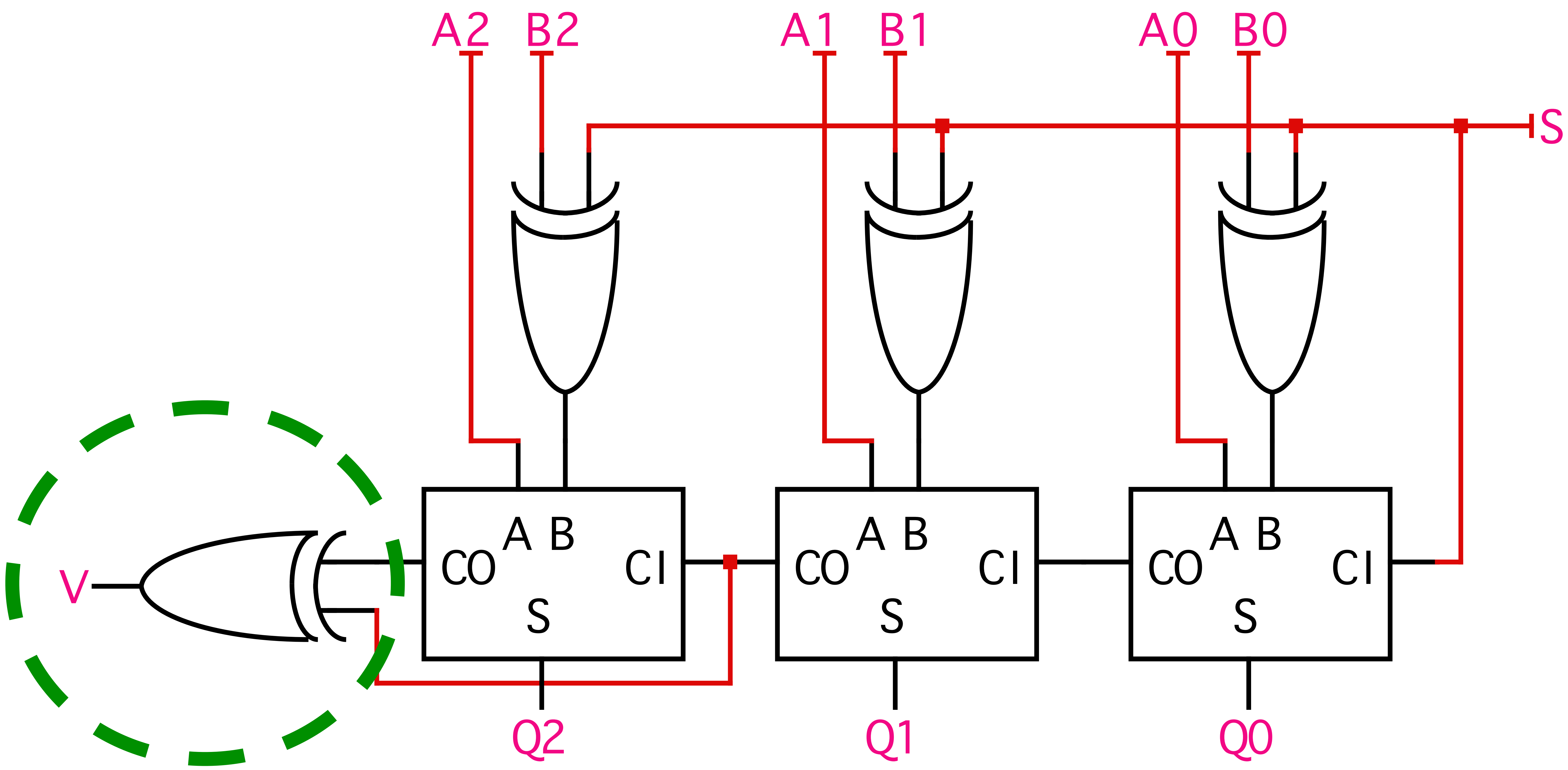
- Ignoring the final carry-out, as before.
- The result will be interpreted as +57.
- This is also arithmetic overflow.
  - in this case, carrying out of the $-2^6$ column into who knows what?

# Overflow Detection

- It is important to know when overflow occurs
  - We can design other hardware to handle it later.
- Seems to occur when carrying into or out of the MSB
  - MSB Column is negative, carry-in to it would be positive.
  - but if we carry in AND carry out:
    - the representation is correct again
      - (recall 50 – 43)
    - although final carry out is still wrong.

# Overflow Detection

- In general: if we carry in but don't carry out, or if we carry out but don't carry in, there's a problem.
- Overflow when Ci=1 and Co=0, or Ci=0 and Co=1
- Ci≠Co, which is the same as Ci⊕Co
- Final circuit:



# Representation Size

- Overflow is a consequence of representation size
  - If the answer can't fit, you get overflow.
- Can we change the size of the representation?
  - make a bigger 2's complement that would fit?
- eg: Start with a 4-bit representation: $-2^3 + 2^2 + 2^1 + 2^0$
  - 1000 = -8, 0111 = +7, 1111 = -1

- To switch to a 5-bit representation: $-2^4 + 2^3 + 2^2 + 2^1 + 2^0$
  - -8 = -(01000) = 10111+1 = 11000
  - +7 = 00111

# Sign Extension

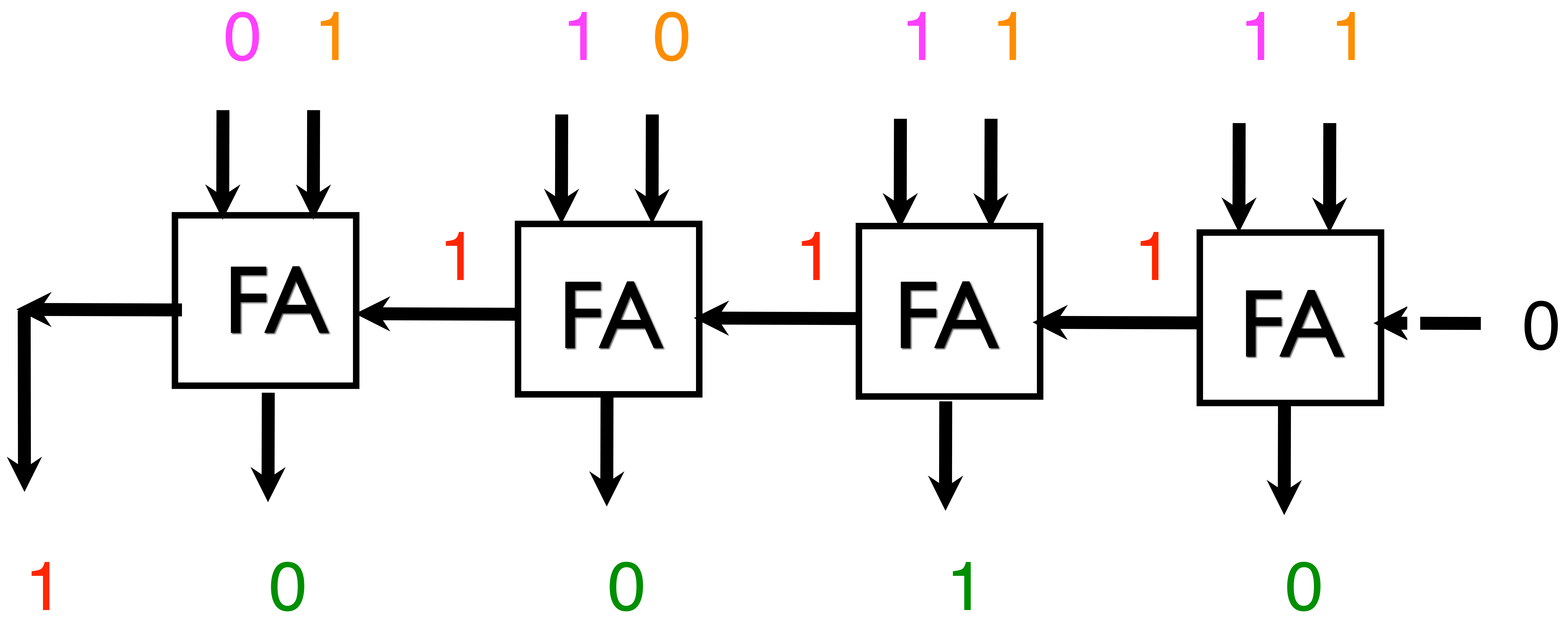
- We can add lots of 0s to the left of a positive number without changing its value
- Similarity, we can add lots of 1s to the left of a two's complement negative number without changing its value
  - subtract $2^n$ for the new negative MSB,
  - $2^{(n-1)}$ was the negative MSB, but is now positive
  - so we must add $2^{(n-1)}$ back to the value, twice
  - since $2^n = 2^{(n-1)} + 2^{(n-1)}$, the value is unchanged

| | |
|---|---|
| 101 = 1101 = 11101 = 11111111101 | $-2^4 + 2^3 + 2^2 + 2^1 + 2^0$ |
| 010 = 0010 = 00010 = 00000000010 | $-2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$ |

# Faster addition

- Recall: full adder has big gate delay
- Also called "Ripple Carry" adder
  - Carry ripples from one adder to the next
  - Total circuit delay depends on the word size.
  - 64 bit adder has ***64 full-adders*** worth of gate delay.
    - we know any circuit can be built with at most 3 levels of gate delay
- Carry lookahead
  - To design some logic so adder can be faster
  - Recognize relationship between Ci and Co

# Ripple-carry adder



# Recall full adder expressions

$s_i = x_i \oplus y_i \oplus ci_i \qquad co_i = x_i y_i + x_i ci_i + y_i ci_i$

- Carry-out is the carry-in for the next full adder

$co_i = ci_{i+1}$ So we'll just use "$c$"

$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$

$= x_i y_i + (x_i + y_i)\, c_i$

- Both $x_i y_i$ and $(x_i + y_i)$ take only 1 gate delay.
- $c_i$ is what takes time so let's get rid of it.

# Carry Look-ahead

- Starting from the LSB

  $c_1 = x_0y_0 + (x_0 + y_0)\ c_0$

  $c_2 = x_1y_1 + (x_1 + y_1)\ c_1$ (then replace $c_1$ as above)

  $c_2 = x_1y_1 + (x_1 + y_1)\ [x_0y_0 + (x_0 + y_0)\ c_0]$
- next do $c_3$. This will get big quickly.
- Let's rename parts of these expressions

  $x_iy_i$ = Carry Generate ($G_i$)

  $x_i + y_i$ = Carry Propogate ($P_i$)
- These are both available in 2 gate delays.
  - Neither depends on the carry-in.
- Why these names?…

# Carry Look-ahead

- *Carry Generate*: Start a carry regardless of the other inputs.
- *Carry Propagate*: If a carry comes in, pass it along, but don't generate a new carry.
- Have a look at the functions, and this makes sense:

$$c_{i+1} = x_i y_i + (x_i + y_i)\, c_i$$

$x_i y_i$ : if $x_i$ and $y_i$ are both 1, there would be a carry out of that bitslice regardless of the other inputs.

$x_i + y_i$ : if $x_i$ or $y_i$ are 1, there would be a carry out of that bitslice only if there was a carry in to that bitslice.

# Carry Look-ahead

- Using the generate and propagate functions:

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 c_1 = g_1 + p_1(g_0 + p_0 c_0)$$

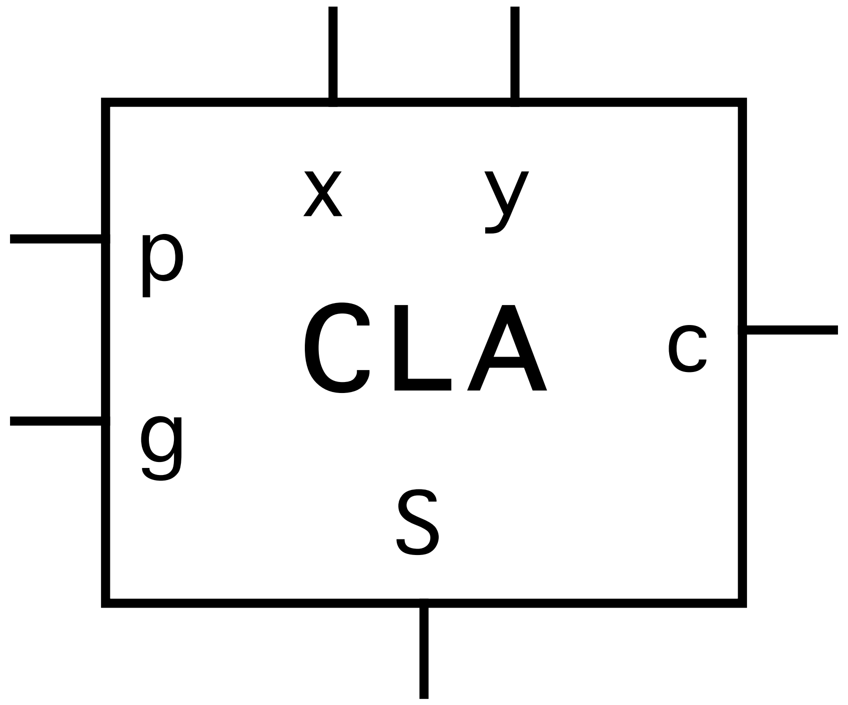
$$= g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \ldots + p_i p_{i-1} \ldots p_0 c_0$$

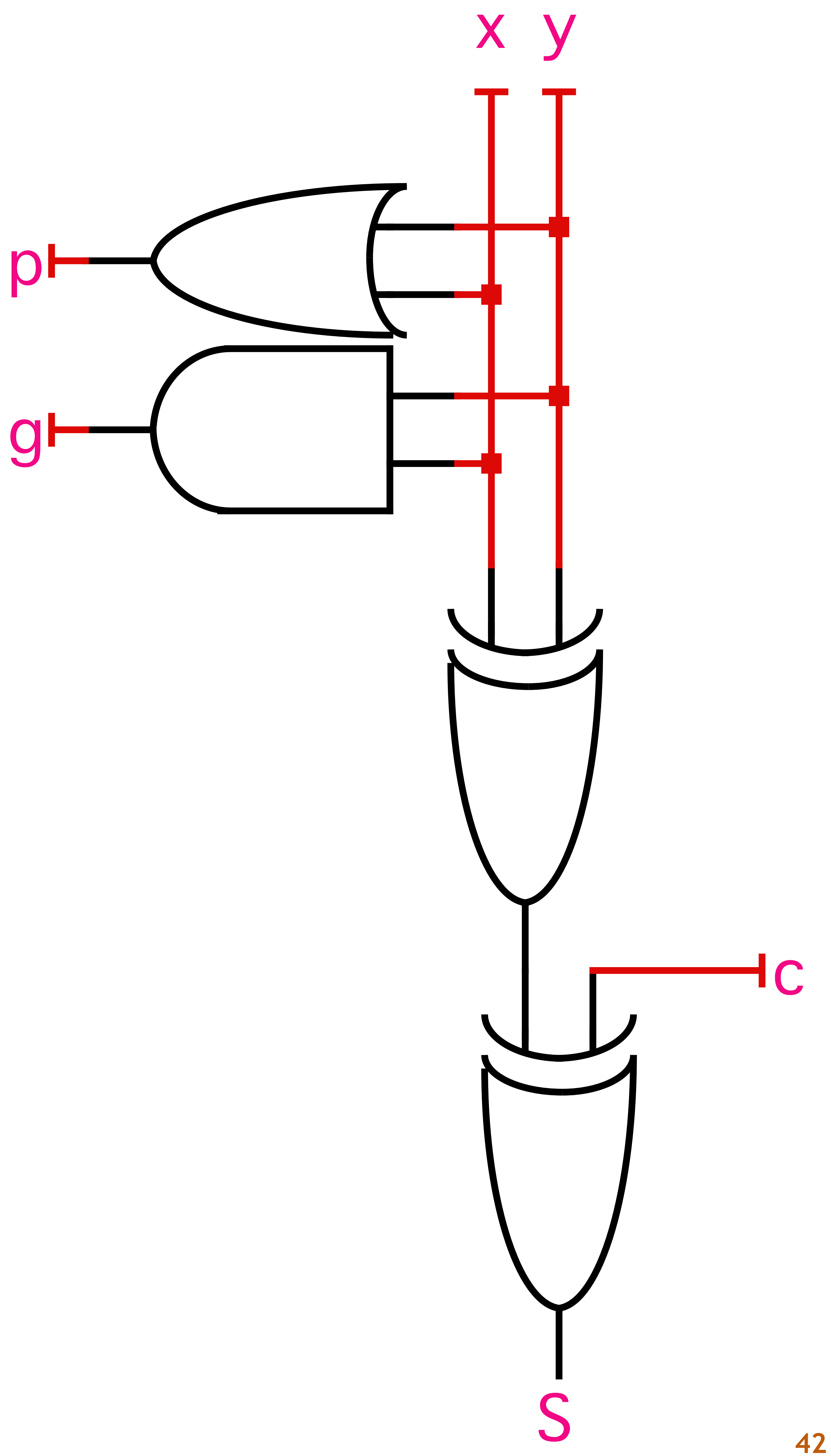
- This will require *lots* more hardware, but
- Guaranteed to execute in 4 gate delays.
  - $p_i$ and $g_i$ are generated in one gate delay
  - $c_i$ takes two gate delays from $p_i$ and $g_i$
  - sum takes one more delay

# CLA Hardware

- Modify the Full Adder:
  - output pi and gi, not ci




- Connect them to create a 4-bit carry-look ahead adder.
  - Lots of additional logic is still required to work with the new P and G signals



# How do we generate c for each bitslice?

- Carry in at bitslice i is true if there is a generate from i-1, or a propogate passing along a carry from a previous slice
- for each bitslice, $c_i = g_i + pc_{i-1}$
- The hardware is constructed recursively, duplicating and building on the logic for calculating carry from generation and propagation from each previous bitslice
- 64 bit adder will have hundreds of gates, thousands of inputs
  - *Trade complexity for speed*

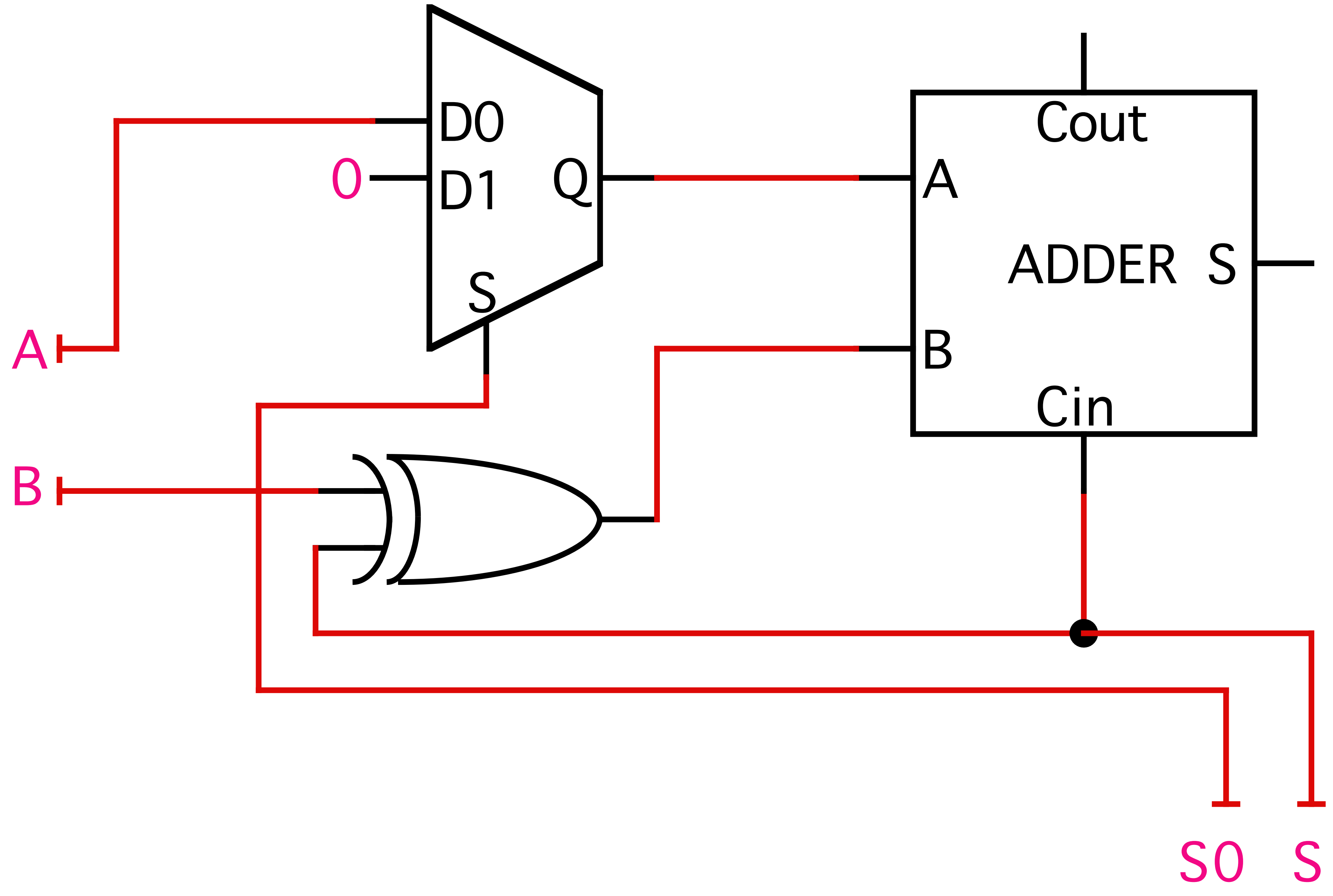
# 4-bit CLA

# ALU: Putting it all together

- Building each individual component of the ALU is done
- To put it all together, we just need multiplexers
  - Generate all possible answers
  - use MUX to choose which answer we will use
  - control signals to ALU are used to select output
- Flags are output signals that give useful information
  - was the result zero? was the result negative? was there overflow?
- Add, subtract, logic, and shifting are all useful

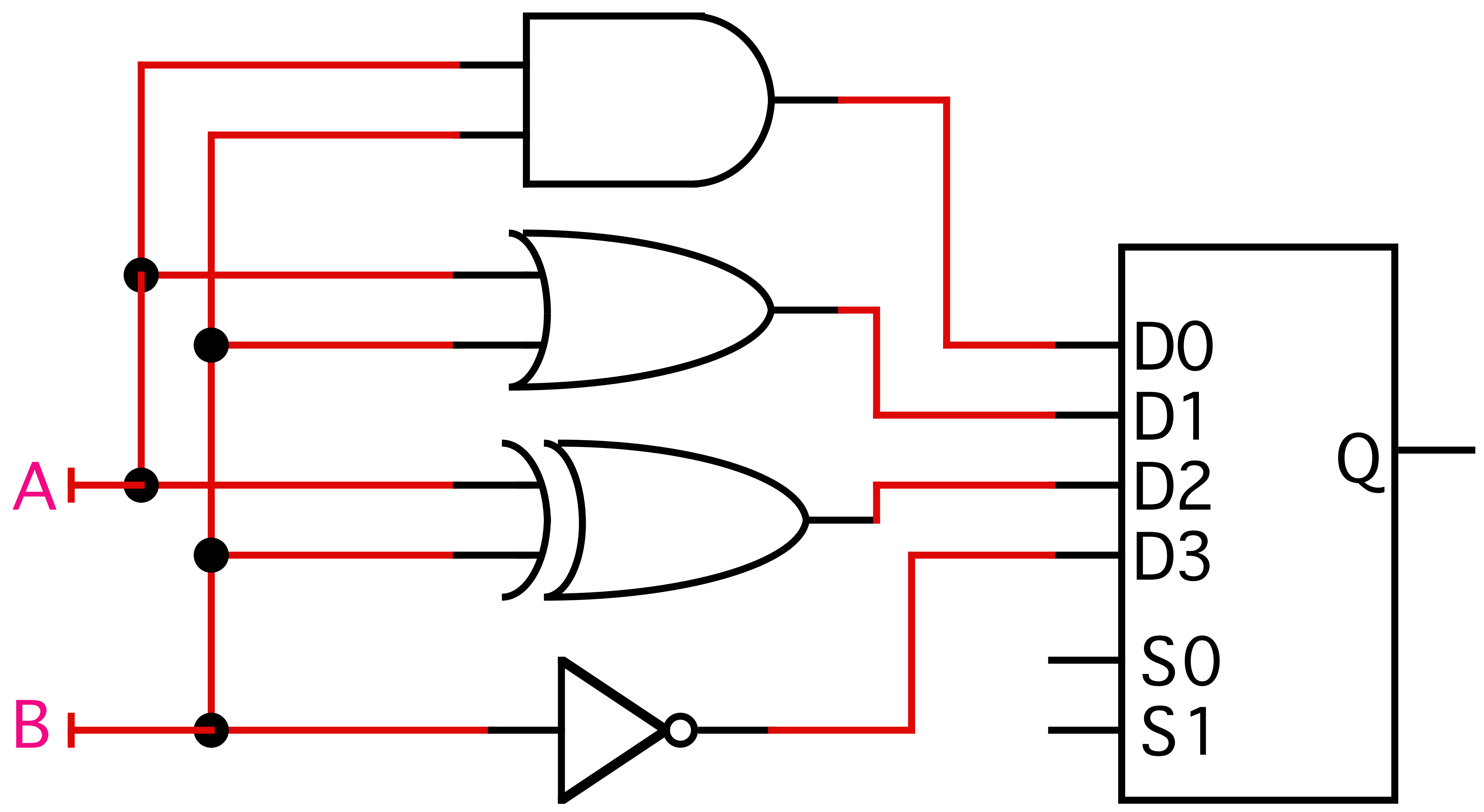
# ALU: Arithmetic

- Desired functions: add, subtract, negative, pass-through
  - or other functions
- Start with adder
- add XOR to B to allow ±B
- Add MUX to A to allow A±B or 0±B
- 2 control signals needed
  - S0: S/$\overline{A}$
  - S1: A=0



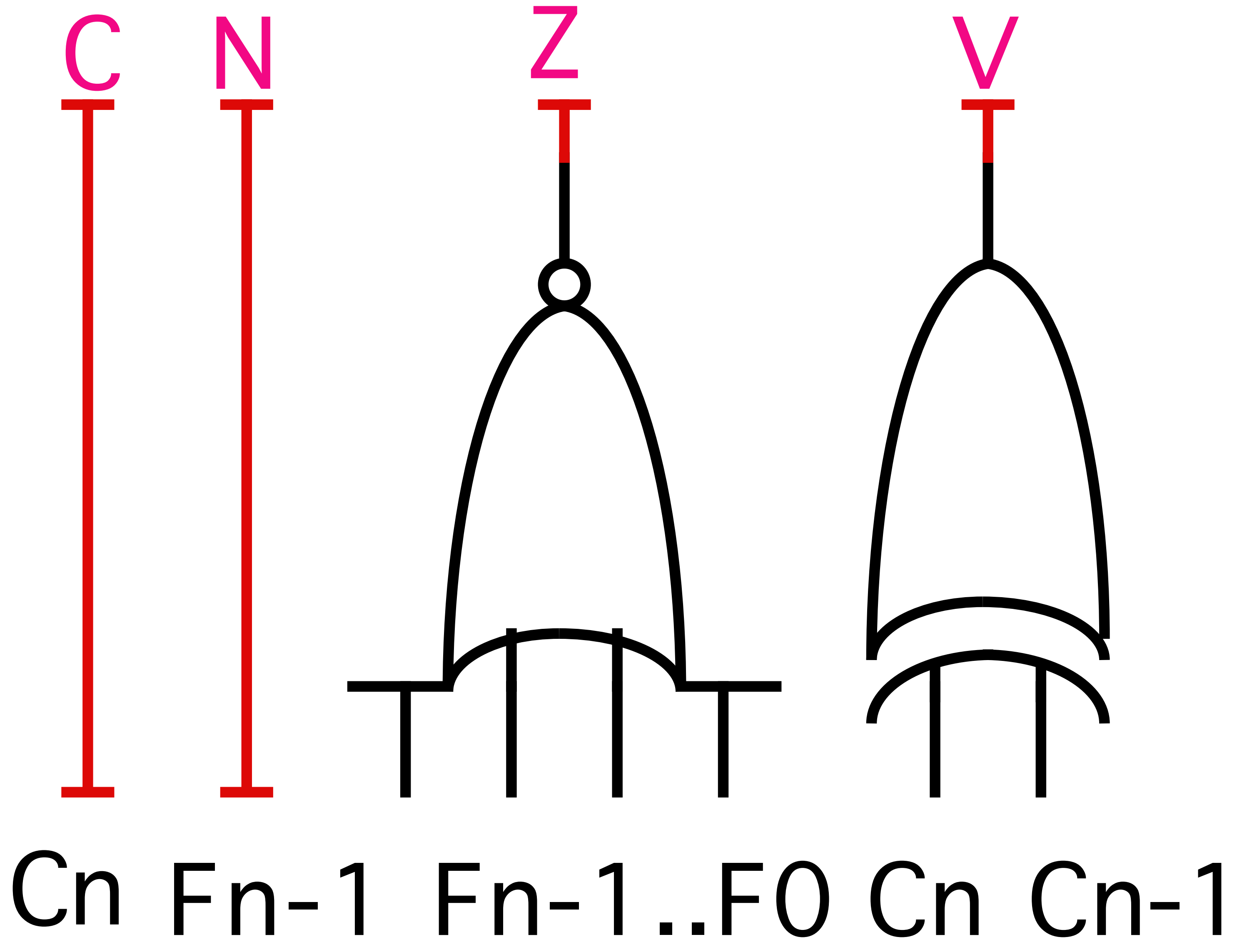
# ALU: Logic

- Pick desired logic functions
  - AND, OR, NOT, XOR
- Select between them with a MUX
  - Again, two control signals
  - Can use the same 2 signals as for Arithmetic
- Select either Arithmetic or Logic with a final MUX and a third control signal (S2)



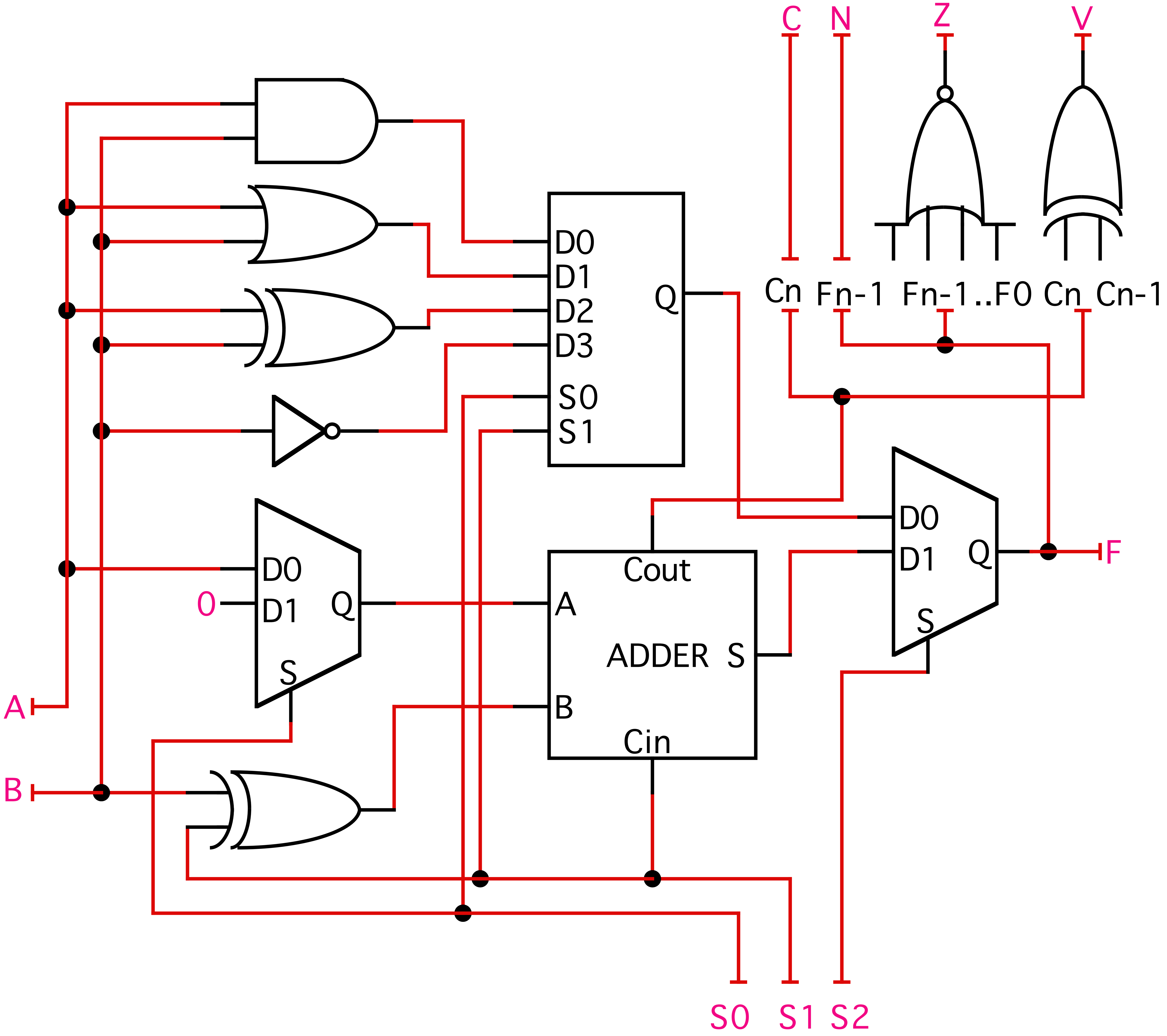
# ALU: FLAGS

- Output signals that indicate certain conditions
- Typical flags: NZVC
  - N = Negative = MSB of output ($F_{n-1}$)
  - Z = Zero = NOR of output $(F_{n-1}+F_{n-2}+..+F_0)'$
  - V = Overflow = $C_{n-1} \oplus C_{n-2}$
  - C = Carry = $C_{n-1}$




- NOTE: different ALU will have different functions, different flags
- NOTE: ALU is entirely combinational

# Complete ALU and function table




| S2 | S1 | S0 | F |
|---|---|---|---|
| 0 | 0 | 0 | AND |
| 0 | 0 | 1 | OR |
| 0 | 1 | 0 | XOR |
| 0 | 1 | 1 | $\overline{B}$ |
| 1 | 0 | 0 | A+B |
| 1 | 0 | 1 | +B |
| 1 | 1 | 0 | A-B |
| 1 | 1 | 1 | -B |

# Sequential Math

- ALU is fine for combinational arithmetic
- Some arithmetic will require multiple steps, decision making, state machines etc
    - Multiplication and Division, to start
- These could be done combinationally, since the answer is the same for the same input
    - but the logic would be very large and complex
    - In practice, sometimes done with look-up tables to speed up the process
- Other common sequential math: graphics processing

# Sequential Math: Binary Multiplication

Recall grade 3 math. eg 14×10=140

$$
\begin{array}{cccccccclr}
 & & & & 1 & 1 & 1 & 0 & & \text{(Multiplicand)} \\
 & & & \times & 1 & 0 & 1 & 0 & & \text{(Multiplier)} \\
\cline{4-8}
 & & & & 0 & 0 & 0 & 0 & = 0 \times 1110 \times 2^0 & \\
 & & & 1 & 1 & 1 & 0 & 0 & = 1 \times 1110 \times 2^1 & \text{(Partial Products)} \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & = 0 \times 1110 \times 2^2 & \\
 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & = 1 \times 1110 \times 2^3 & \\
\cline{1-8}
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & = 140 & \text{(Product)}
\end{array}
$$

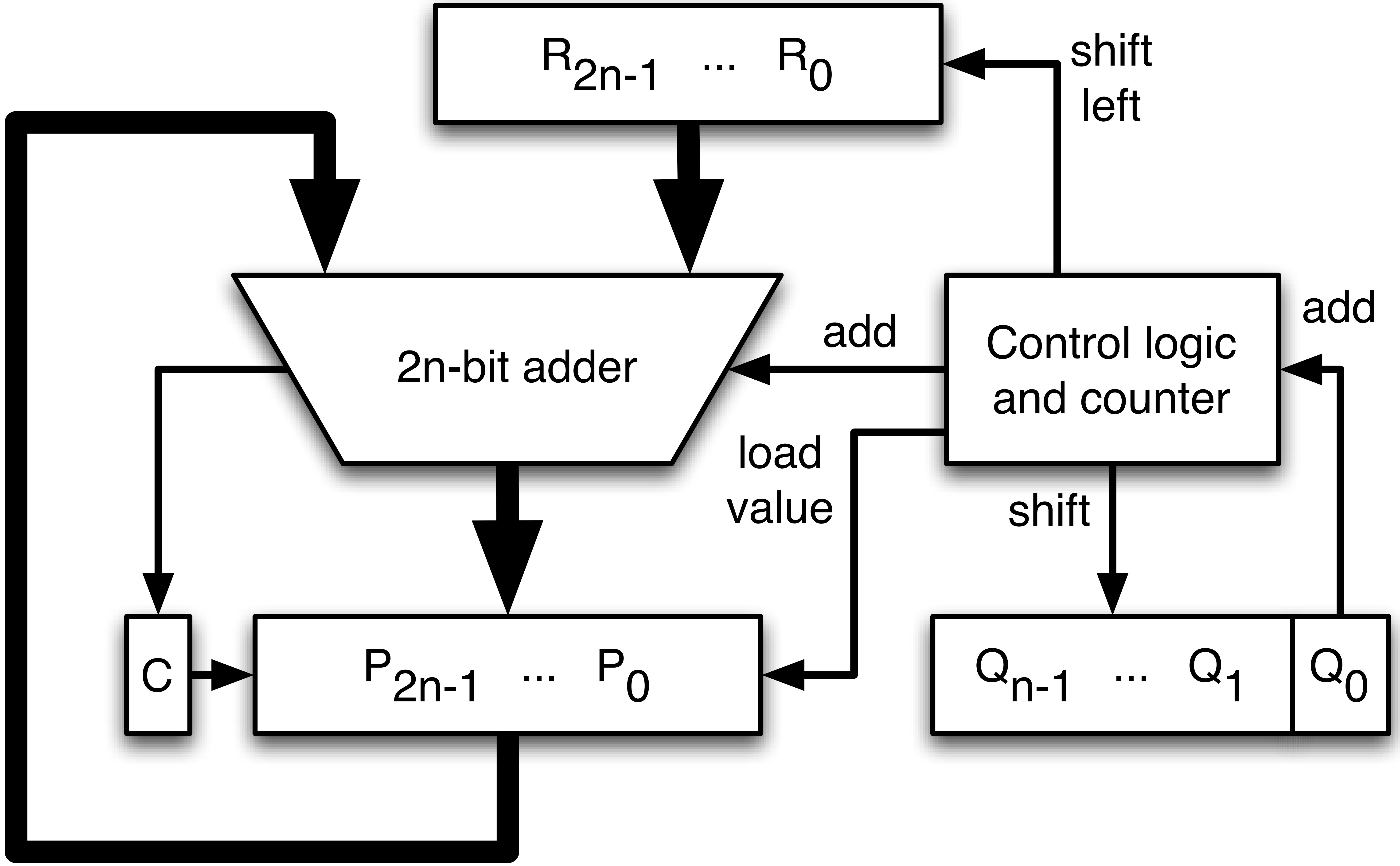
Note: the trailing zeros are usually implied.

Note: assume positive numbers.

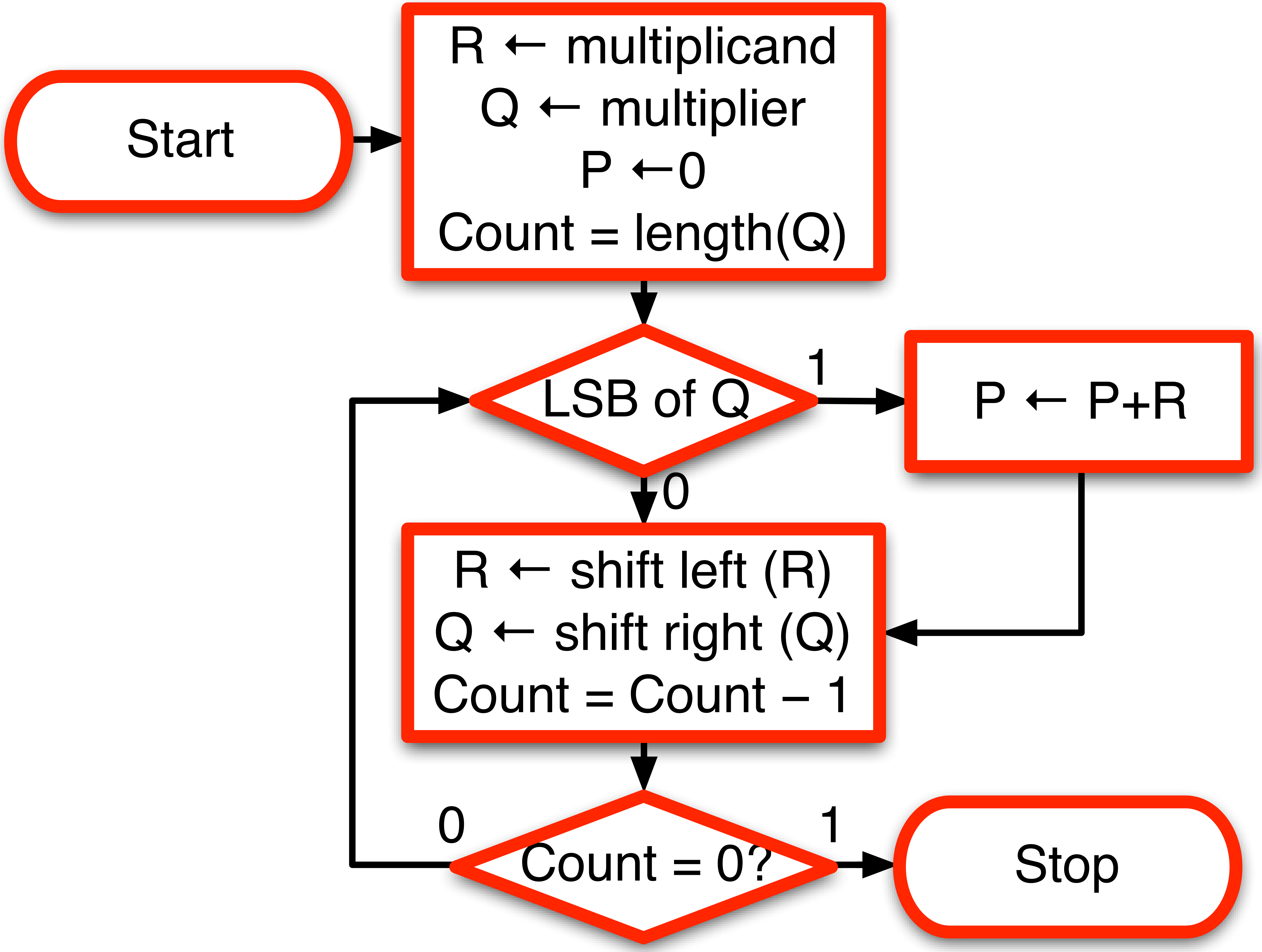
# Binary Multiplication

- You've seen this in the lab, I think
- Registers, with shift
- Adding with carry
- Repeated shift and add, accumulating result
- Shifting a 0 into the register will maintain the partial products
- Add or not add the multiplicand, based on the current bit of the multiplier
- We'll go into more detail, build a couple variants

# Binary multiply implementation



# Binary multiply flowchart



# Binary multiply example

- e.g.: 14 × 10 = 140 Step by step:

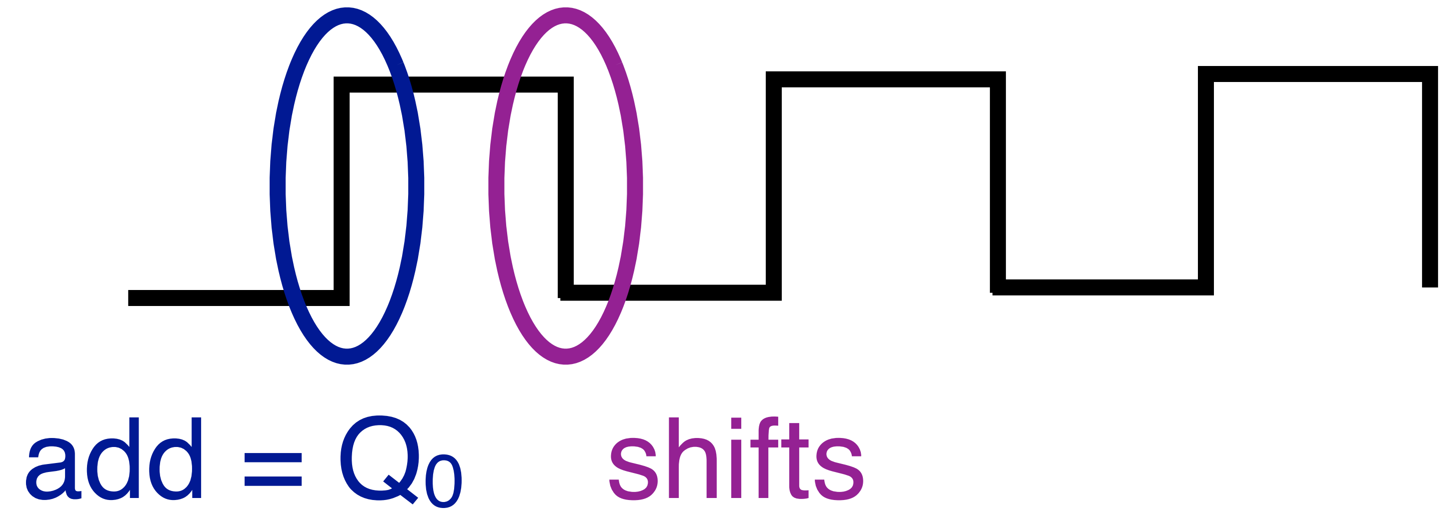
| count | R | Q | P |
|---|---|---|---|
| 4 | 1110 | 1 0 1 0 | 0 |
| 3 | 11100 | 1 0 1 | 11100 |
| 2 | 111000 | 1 0 | 11100 |
| 1 | 1110000 | 1 | 10001100 |
| 0 | Done. P=10001100 | | |

# Binary multiply algorithm

- Use a down-counter to decide when done
  - Start with c = length(Q)
  - done when c = 0
- Store Q, R and P in shift registers
  - size(Q) = number of bits in Q
  - size(R) = 2 × size(Q) (room for left shifts)
  - size(P) = 2 × size(Q)
- Other logic to control the circuit

# Control Logic and Counter

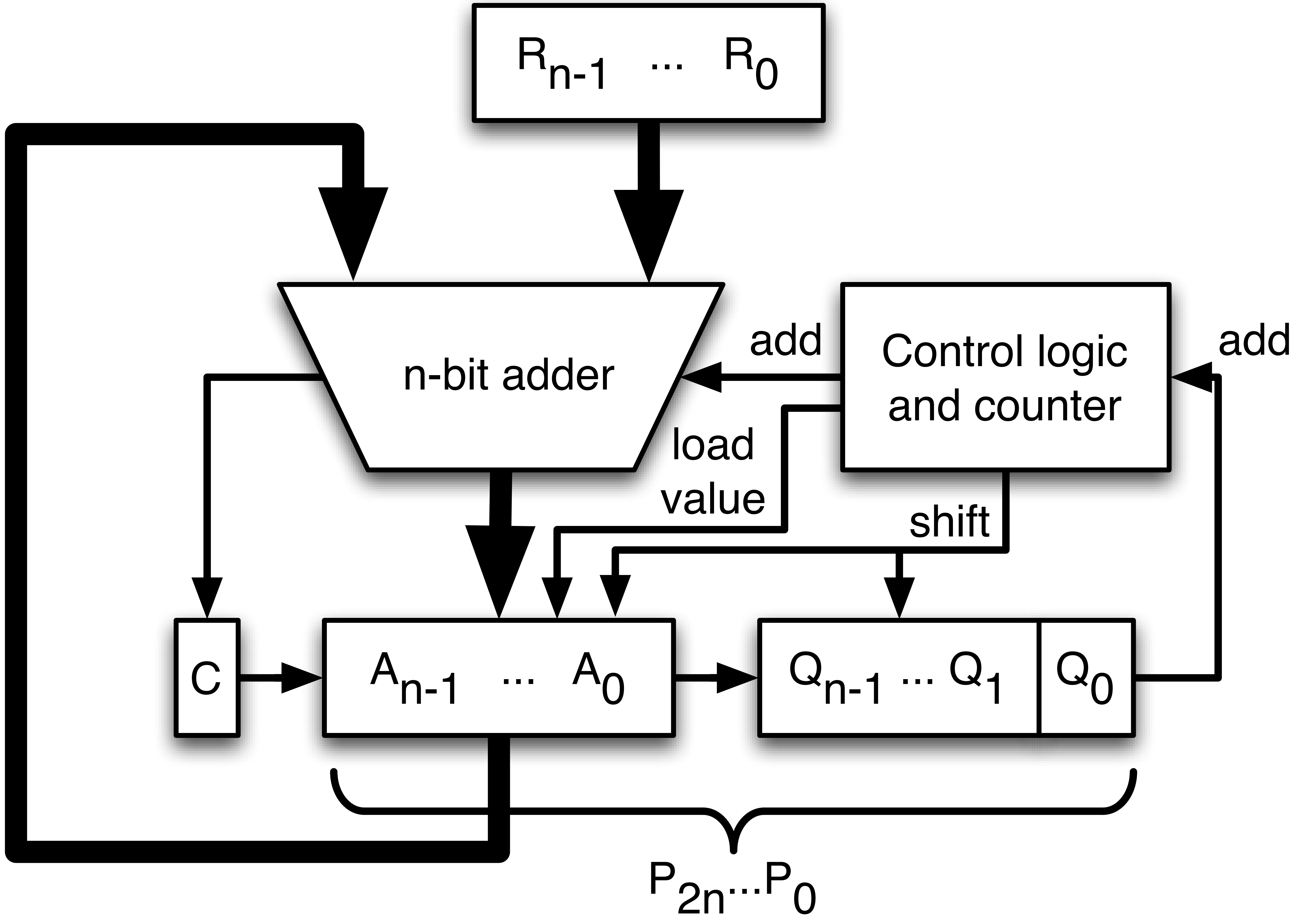
- Finite state machine, designed as we have done in class
- Separate into counter (which we can design) and control logic
  - Counter: set to *n*, decrement to 0 (NOR = done)
  - Control logic: if $Q_0 = 1$, add and load new value
    - route $Q_0$ to add and load signals
    - Shifts - always shift.



# Binary multiply: a better circuit?

- We're only really adding *n* bits at a time
  - Can we get away with an *n* bit adder?
- As the number of bits in P increases, the number of bits in Q decreases
  - Perhaps we can utilize this as well.
- Consider the following hardware:

# Binary multiply: a better circuit?



# Binary multiply: a better circuit?

- All registers, and the adder, are now *n* bit
- New register A holds partial product.
- Carry bit is shifted into MSB of A
- LSB of A is shifted into MSB of Q
- A and Q together make partial products P

# Signed Multiplication

- Check sign of R and Q
- Make positive, if necessary
- Perform multiplication as before
- Change sign of result P if necessary
  - i.e. if $R_{n-1} \oplus Q_{n-1} = 1$, result should be negative.
- Easier way: Booth algorithm
  - Takes advantage of some cool math

# The booth algorithm

- Consider a number containing a string of 1s
  - eg. 011110
- Recall (from 2's complement work):

  1111 = 10000 − 1
- It is also true that

  011110 = 100000 − 10

  $2^4 + 2^3 + 2^2 + 2^1 = 16 + 8 + 4 + 2 = 30$

  $= 32 - 2$

  $= 2^5 - 2^1$

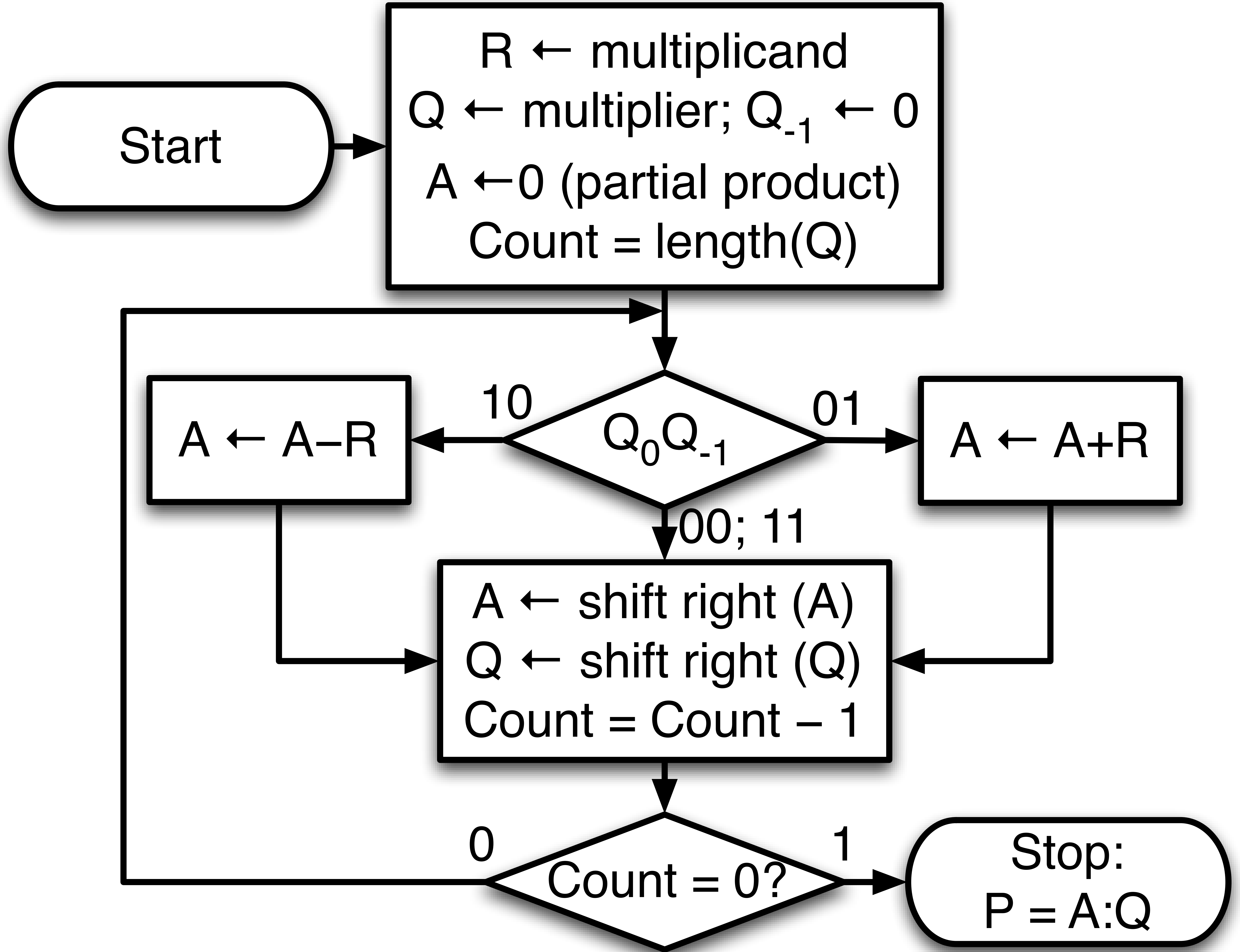
# The booth algorithm

When the multiplier has a string of 1s,
- add at the multiplicand at the start of the string
- subtract the multiplicand at the end of the string

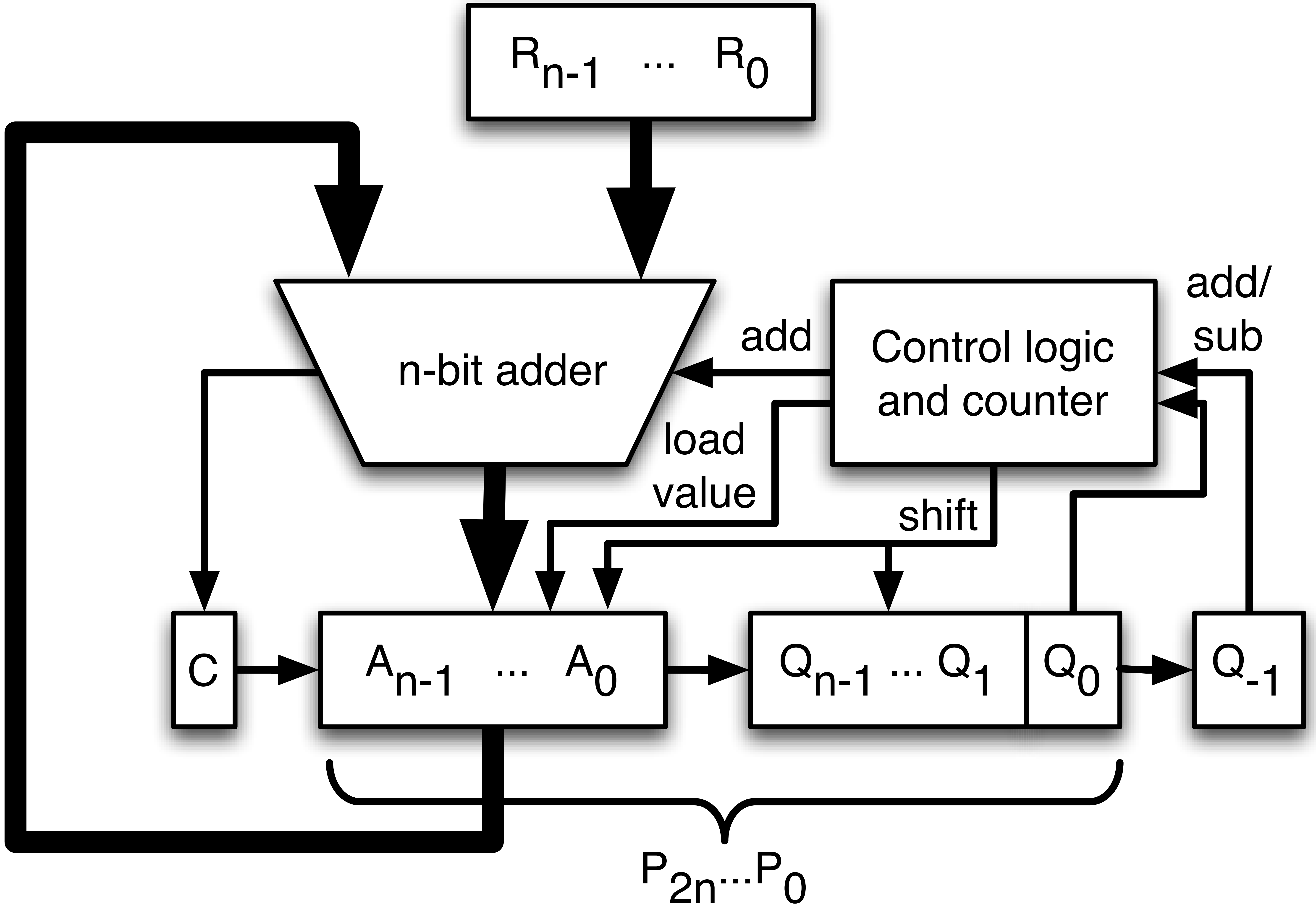
Identifying the ends of a string of 1s
- Starts with "01", ends with "10"

This also allows us to multiply signed numbers
- because we are subtracting.
- e.g.: $100001 = -100000 + 10$

# The booth algorithm



# The booth algorithm implementation



# But wait...

- This algorithm needs to handle positive AND negative numbers, but
- Shift right could turn negative into positive!

  1100 →shift right→ 0110; −4 → 6

- We want to retain the sign when we shift
- *Arithmetic* shift right: replicate the sign bit.

  1100 →arithmetic shift right→ 1110; −4 → −2

# Shifts

- Logical shift: Shift in a "0" from either direction
- Arithmetic shift: maintain sign bit
  - Shifting right: replicate the sign bit, drop the LSB
    - $\boxed{Q_n \rightarrow Q_{n-1} \rightarrow \ldots \rightarrow Q_1 \rightarrow Q_0}$
    - becomes $Q_n\ Q_n\ Q_{n-1} \ldots Q_1$
  - Shifting left: Retain the sign bit, shift in a "0" from the right.
    - $\boxed{Q_n \quad Q_{n-1} \leftarrow Q_{n-2} \ \ldots \leftarrow Q_0} \leftarrow 0$
    - becomes $Q_n\ Q_{n-2} \ldots Q_1\ 0$

# The booth algorithm : Example

10111 × 10011, (−9)×(−13)= +117 = 01110101

| C | A | Q | Q−1 | Next Function |
|---|---|---|---|---|
| 5 | 00000 | 10011 | 0 | Subtract |
| | 01001 | 10011 | 0 | Arith. shift |
| 4 | 00100 | 11001 | 1 | Arith. shift |
| 3 | 00010 | 01100 | 1 | Add |
| | 11001 | 01100 | 0 | Arith. shift |
| 2 | 11100 | 10110 | 0 | Arith. shift |
| 1 | 11110 | 01011 | 0 | Subtract |
| | 00111 | 01011 | 0 | Arith. shift |
| 0 | 00011 | 10101 | 1 | Done |

# Unsigned division

Again, go back to grade 3:

147 ÷ 11 = 13, remainder 4

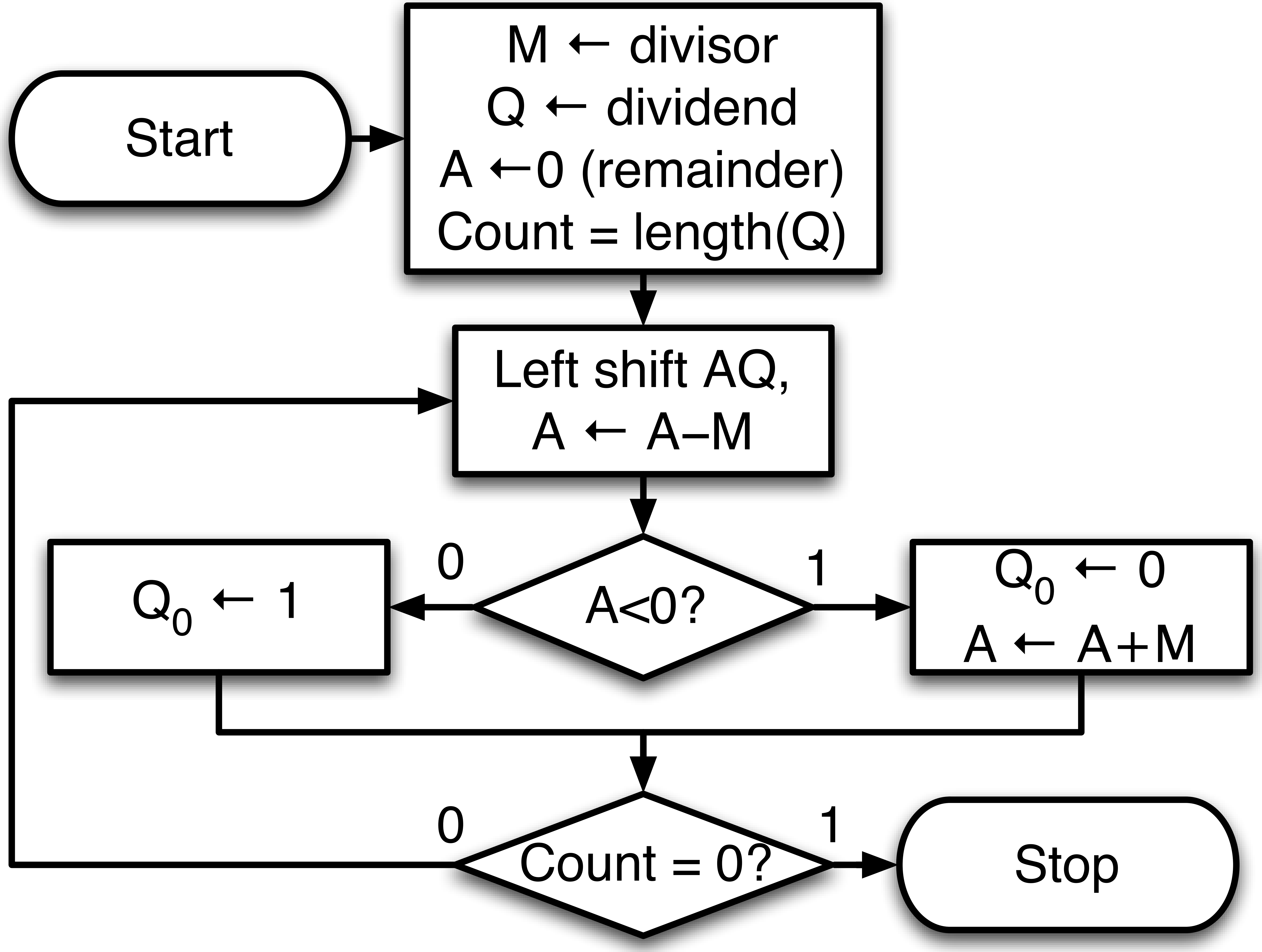
```
Divisor              00001101     Quotient
                ________
                1011)10010011     Dividend
                      1011↓||
                      001110||
                        1011↓↓
                        001111
                          1011    Remainder
                           100
```

Partial Remainders

# Division

- multiplication is repeated addition of the multiplicand based on whether or not the multiplier is 0 or 1
- division is repeated subtraction of the divisor from the dividend
  - If the result is negative you took away too much and the quotient should be 0 at that bit position
  - add back the divisor
  - If the result is positive, you didn't subtract too much so the quotient should be 1 at that bit position

# Procedure: Unsigned division



# Things to consider

- This algorithm assumes M > Q
- A is n+1 bits long
- Shifting A and Q together
  - same idea as in multiply
- Successive subtraction
  - Each time, if the result is negative, subtracted too much so add back again.
- Use a counter, start at "n"
- "Done" when counter reaches 0
  - Quotient is in "Q", Remainder is in "A"

# Example: 1000/11=10r10 (8/3=2r2)

| C | A | Q | Function |
|---|---|---|---|
| 4 | 00000 | 1000 | Start |
| | 00001 ← | 000☐ | shift AQ |
| | 11110 | 000☐ | $A \leftarrow A - M$ |
| 3 | 00001 | 0000 | $Q_0 \leftarrow 0$, $A \leftarrow A + M$ |
| | 00010 ← | 000☐ | shift AQ |
| | 11111 | 000☐ | $A \leftarrow A - M$ |
| 2 | 00010 | 0000 | $Q_0 \leftarrow 0$, $A \leftarrow A + M$ |
| | 00100 ← | 000☐ | shift AQ |
| | 00001 | 000☐ | $A \leftarrow A - M$ |
| 1 | 00001 | 0001 | $Q_0 \leftarrow 1$ |
| | 00010 ← | 001☐ | shift AQ |
| | 11111 | 001☐ | $A \leftarrow A - M$ |
| 0 | 00010 | 0010 | $Q_0 \leftarrow 0$, $A \leftarrow A + M$ |

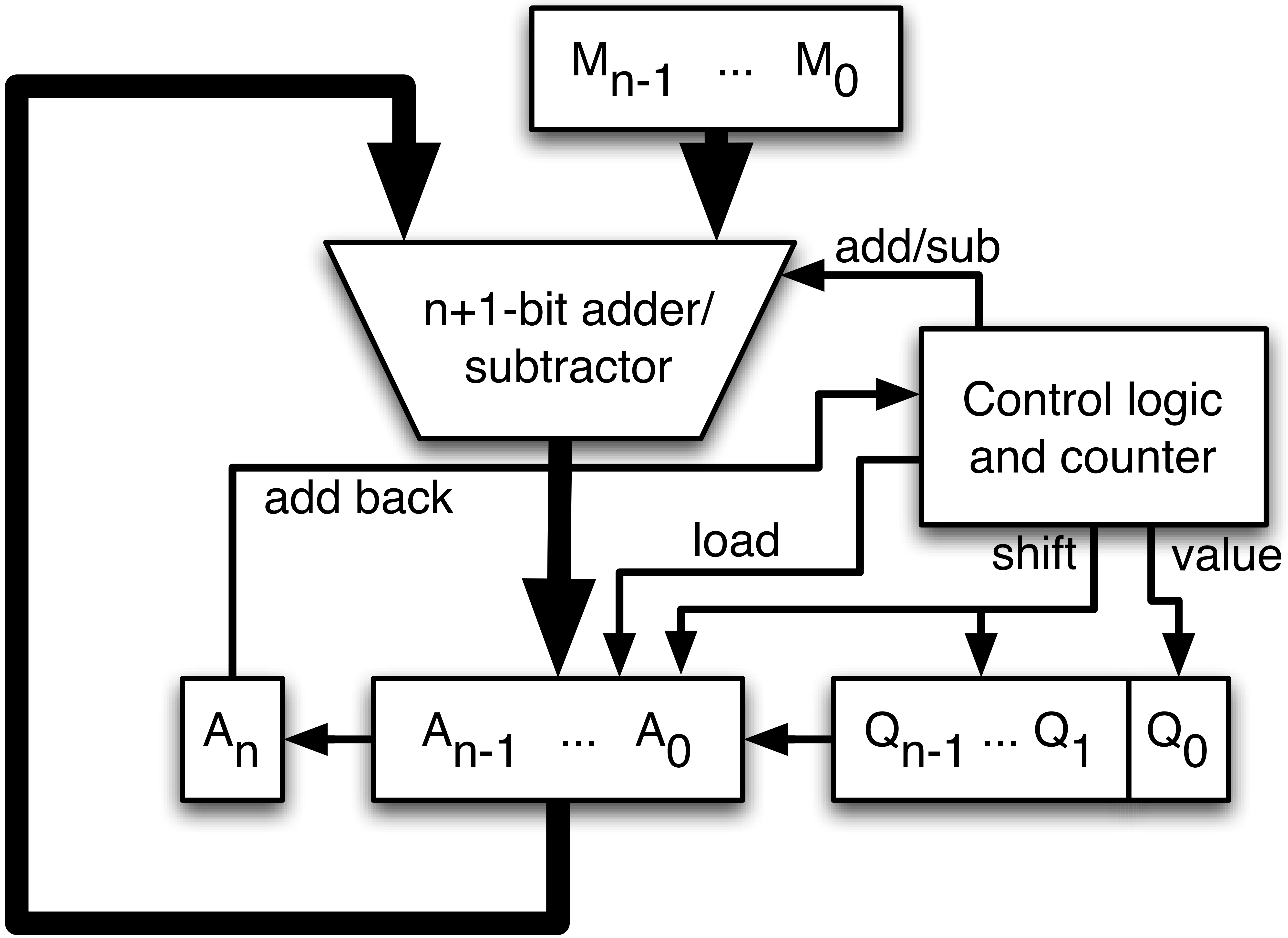
00001 − 11 = 11110

00010 − 11 = 11111

00100 − 11 = 00001

00010 − 11 = 11111

# Hardware for division



# Floating Point Numbers

- So far we've dealt with integers
- Need to allow computers to handle decimals, big numbers and small numbers
- We'll cover this later in the course, somewhat abstractly

# Where are we now?

- We’ve built all the parts we need to start assembling them into a computer
  - Registers
  - Memory (sort of - we’ll do more)
  - ALU and other math bits
- We need to do some work before we start assembling
  - What will this computer do, and how?
  - How to move data around in the computer?
- Next: Assembly Language.