COMP311: *COMPUTER ORGANIZATION!*

Lecture 10: Adders/Subtractors, Shifters

tinyurl.com/comp311-sp26

Logistics

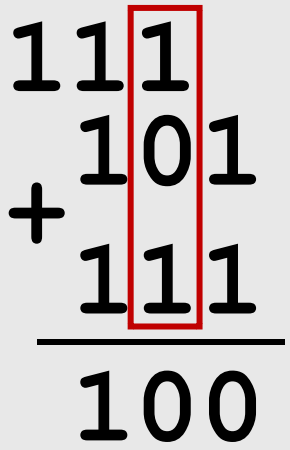
- Quiz next class! Review tomorrow 5pm in SN014. Topics:
  - *K-Maps: Reading them, filling them in, using them to find simplified Boolean equations*
  - *"Don't Cares" (in relation to to K-Maps, Truth Table Minimization)*
  - *Transistor Minimization (review lecture where we converted things into NAND/NOR gates)*
  - *Multiplexers: conceptually + circuit design w/ them*
  - *Reading + writing basic assembly. Instructions we have covered: ADD, ADDI, SUB, BEQ. Understanding how labels work.*
  - *Adder + subtractor circuits*
  - *Circuit design generally. Given an English specification, can you write down the truth table/draw the circuit*
- WA3 due Thurs
- Lab due Monday!

ADD, ADDI, SUB

ADDER CIRCUITS!

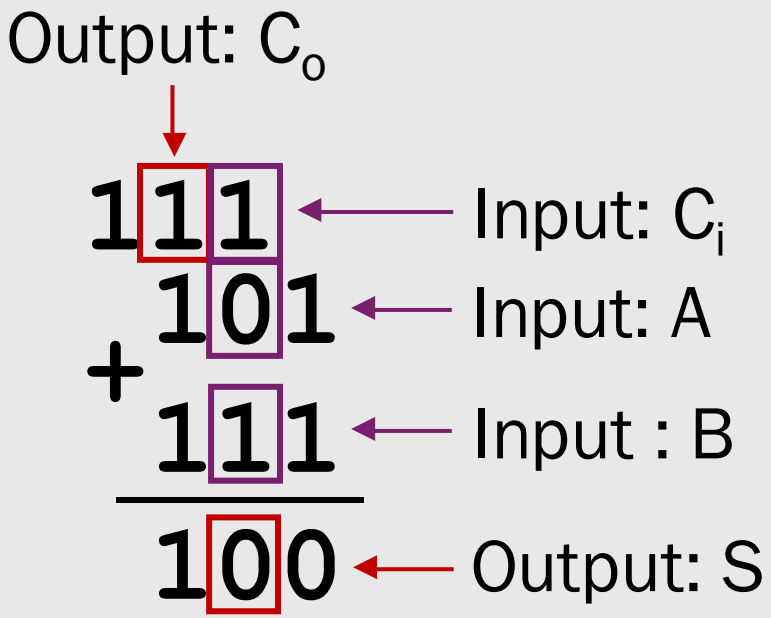
Full Adder

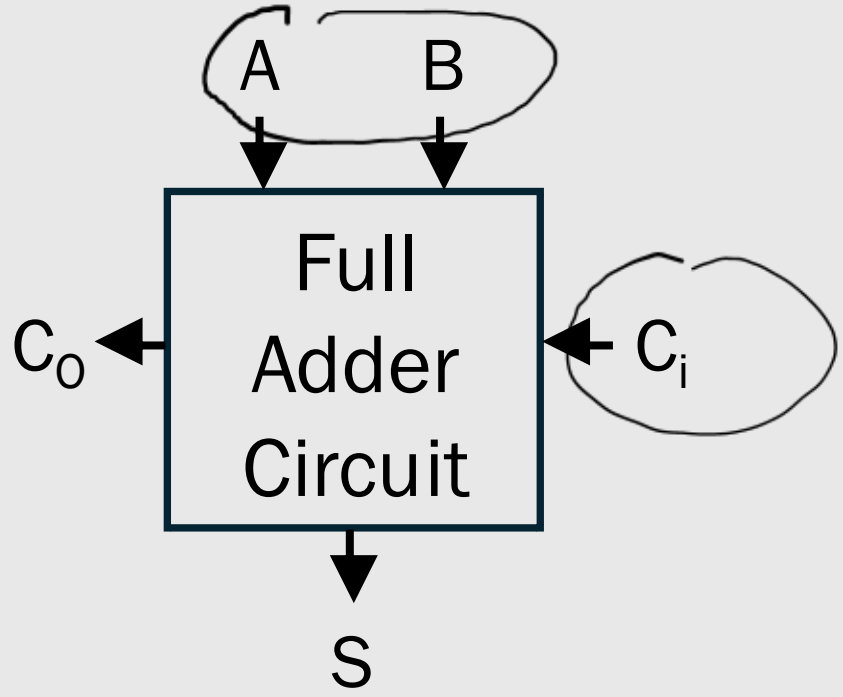
- We need to build a circuit that can add together three one-bit values



Full Adder

- This circuit will be able to add together the three bits in column 1





Full Adder

A	B	C_i	C_0	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

1. Fill in truth table
2. Find the equations
3. Draw the schematic
    1. Count the transistors in your design

$S = A \oplus B \oplus C$

A \ BC_i	00	01	11	10
0			1	
1				1

Hint: Maybe a K-Map would help for C_0

Full Adder

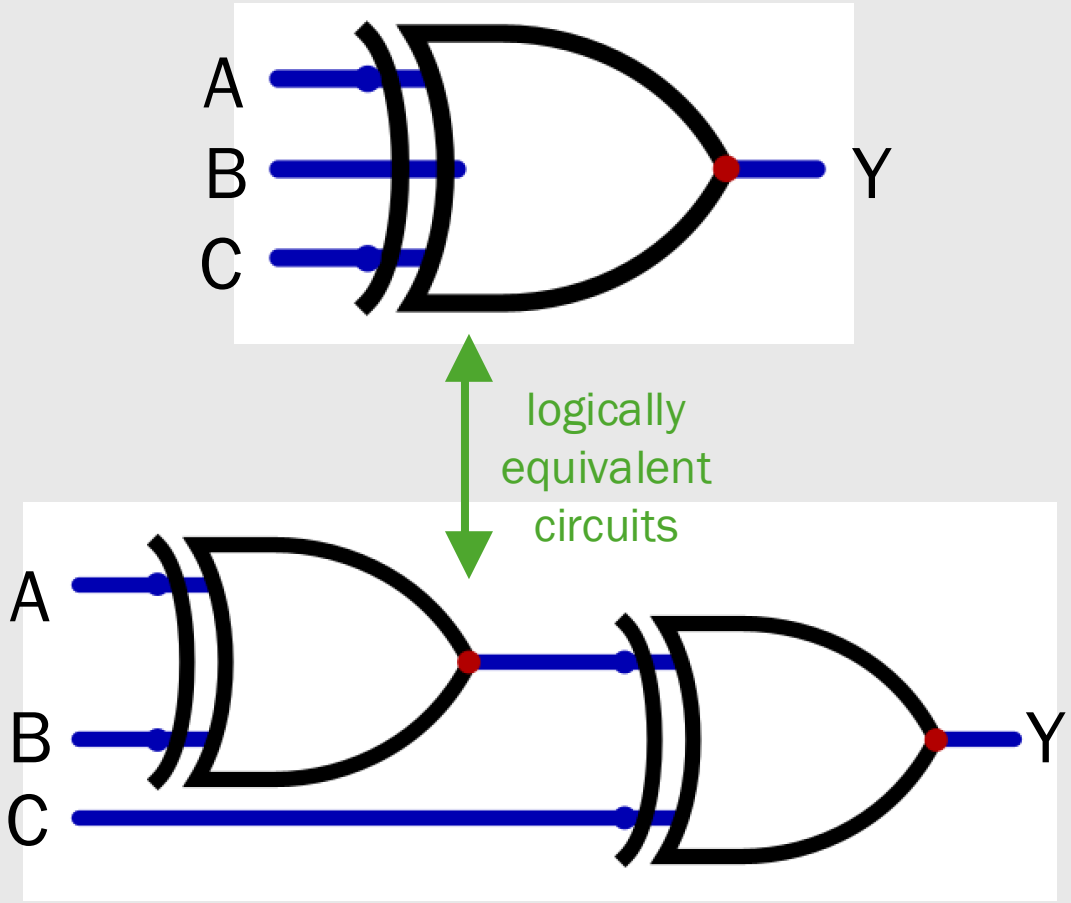
A	B	C_i	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus Ci$$

A \ BC_i	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$C_O = ACi + AB + BCi$$

Recall: 3-input XOR



A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$= S$

For any number of inputs, the output is 1 when an odd number of inputs is 1

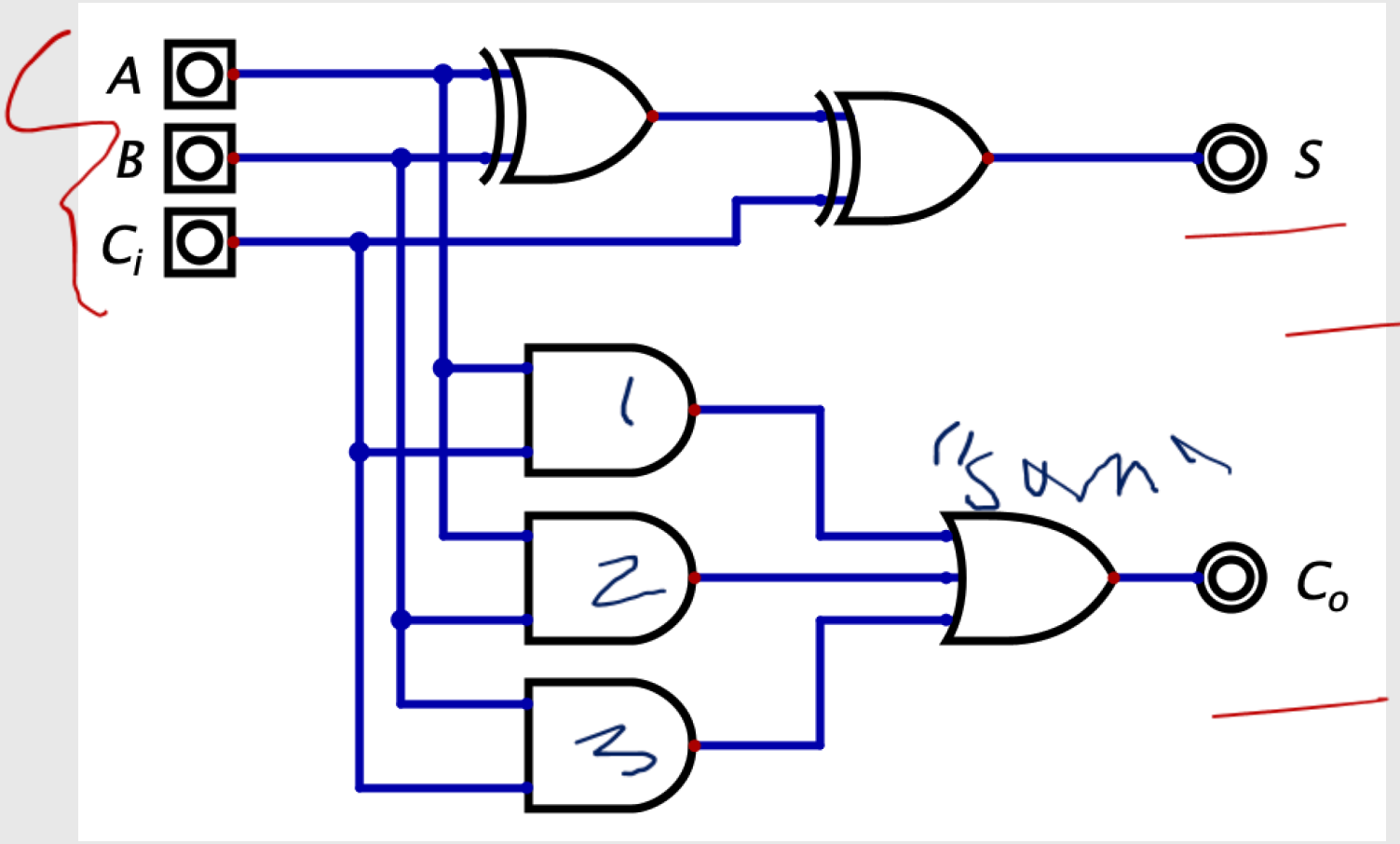
Transistor Count

Gate	# of Transistors
NOT	2
n-input AND	2n + 2
n-input OR	2n + 2
n-input NAND	2n
n-input NOR	2n
2-input XOR	12
3-input XOR	28
2-input XNOR	12
3-input XNOR	28

Full Adder

$$S = A \oplus B \oplus Ci$$

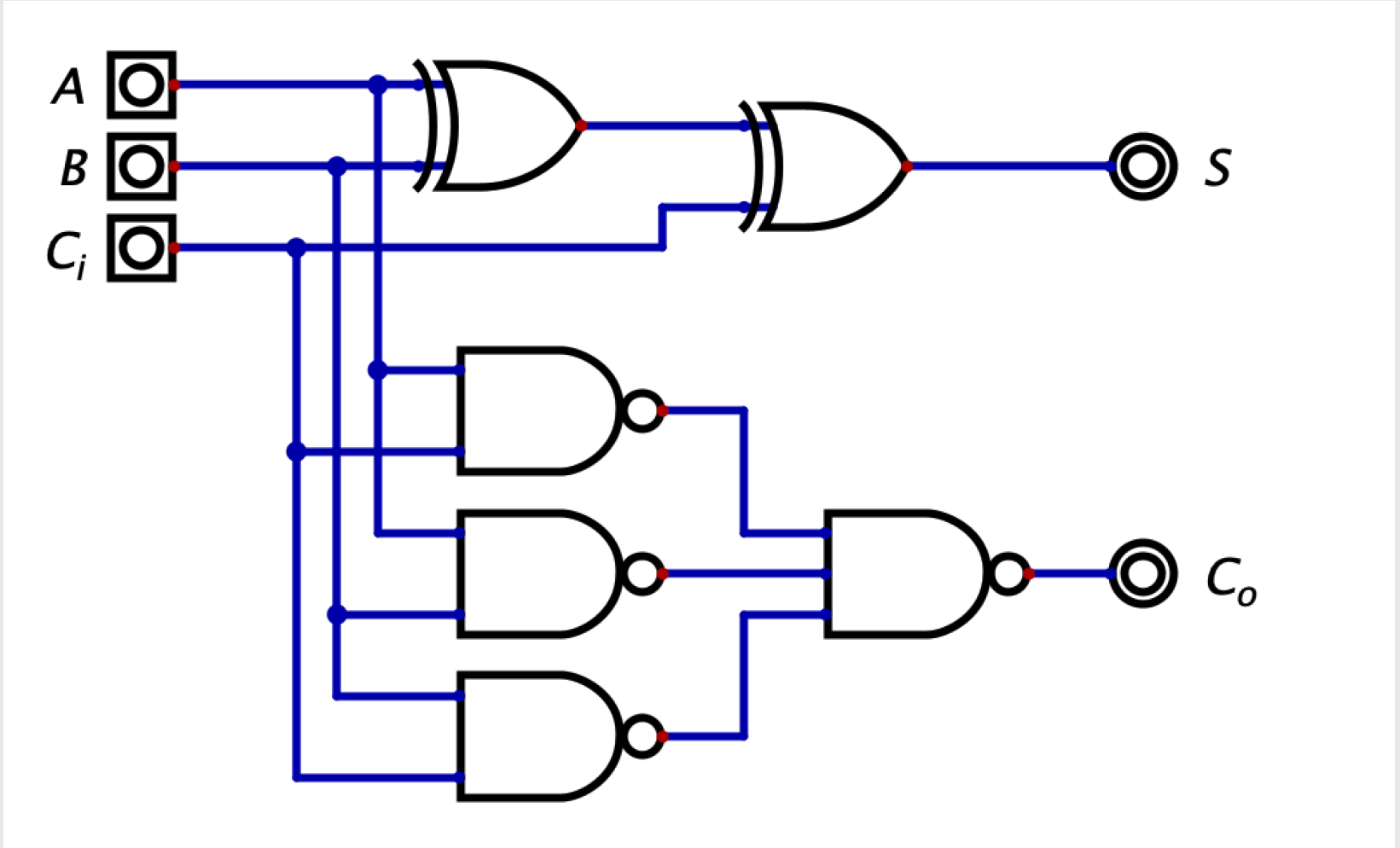
$$C_O = ACi + AB + BCi$$



Full Adder: transistor minimization

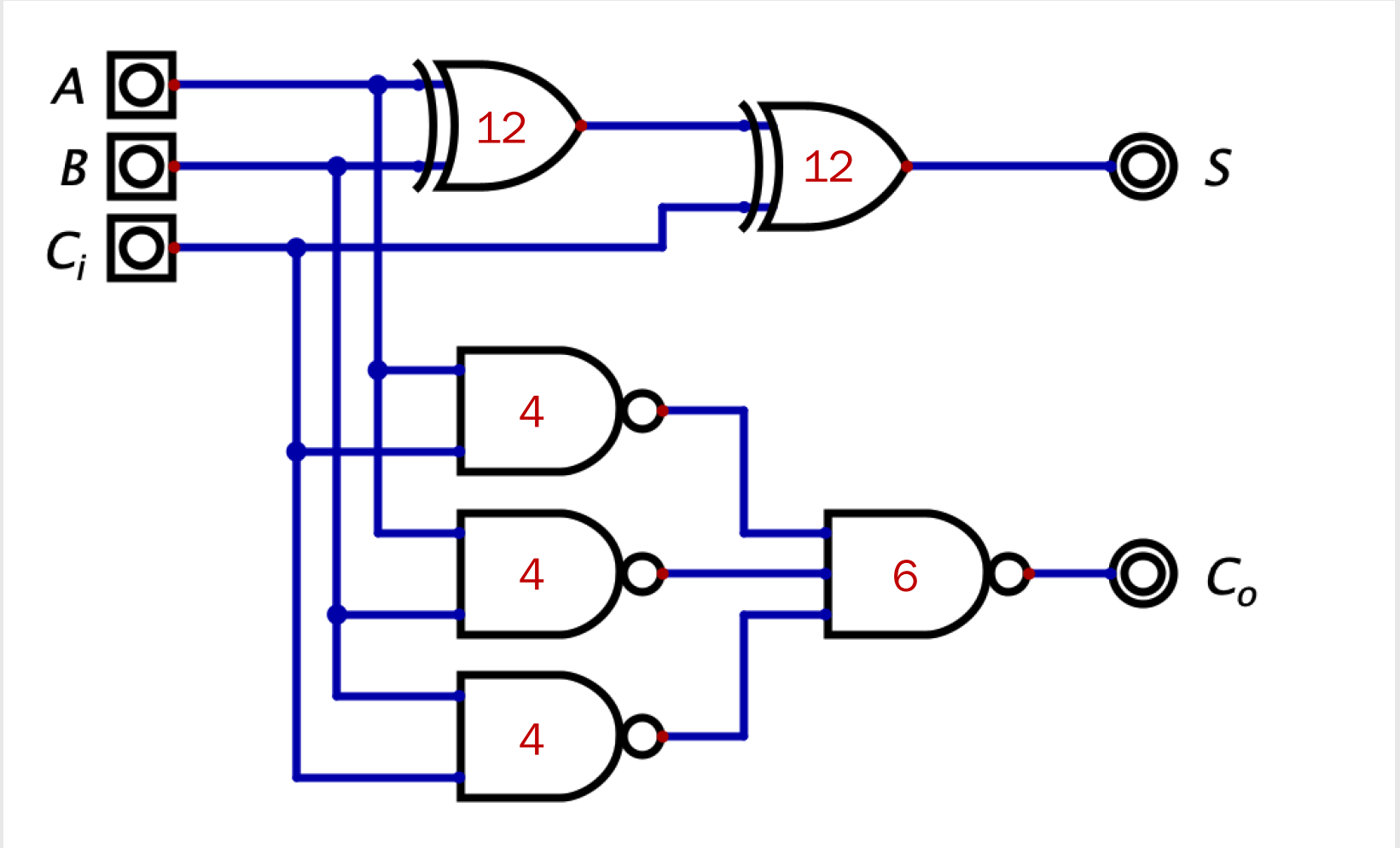
$$S = A \oplus B \oplus Ci$$

$$C_o = ACi + AB + BCi$$



Full Adder: transistor minimization

$$S = A \oplus B \oplus Ci \qquad C_o = ACi + AB + BCi$$



Transistor count = 42

Full Adder: Alternate implementation

A	B	C_i	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

$$S = A \oplus B \oplus C$$

$$C_o =$$

Full Adder: Alternate Implementation

A	B	C_i	C_0	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

$$S = A \oplus B \oplus C$$

Canonical SOP

$$C_0 = \bar{A}BC_i + A\bar{B}C_i + AB\bar{C}_i + ABCi$$

$$C_0 = \bar{A}BC_i + A\bar{B}C_i + AB(\bar{C}_i + Ci)$$

$$C_0 = \bar{A}BC_i + A\bar{B}C_i + AB$$

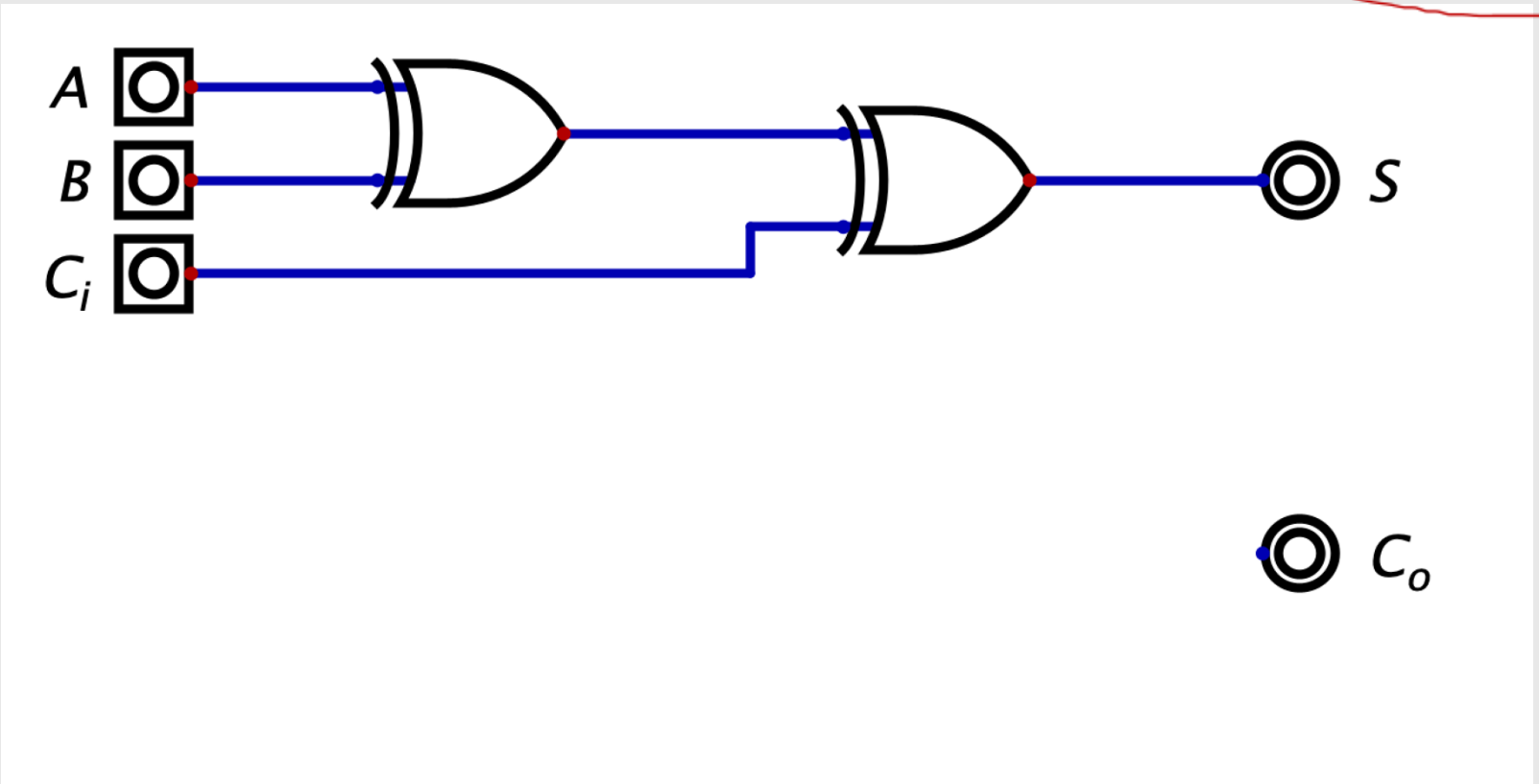
$$C_0 = C_i(\bar{A}B + A\bar{B}) + AB$$

$$C_0 = C_i(A \oplus B) + AB$$

Full Adder: Alternate Implementation

$$S = A \oplus B \oplus Ci$$

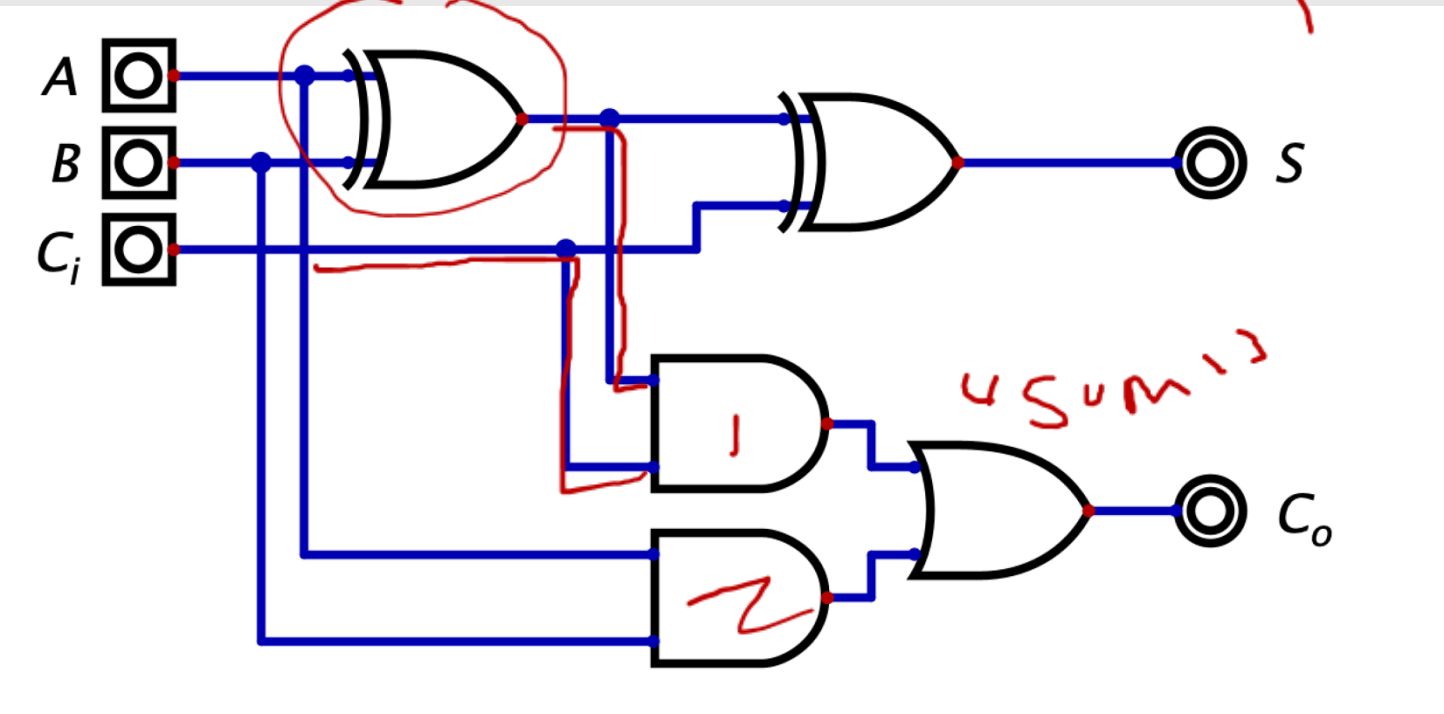
$$C_O = C_i(A \oplus B) + AB$$



Full Adder: Alternate Implementation

$$S = A \oplus B \oplus Ci$$

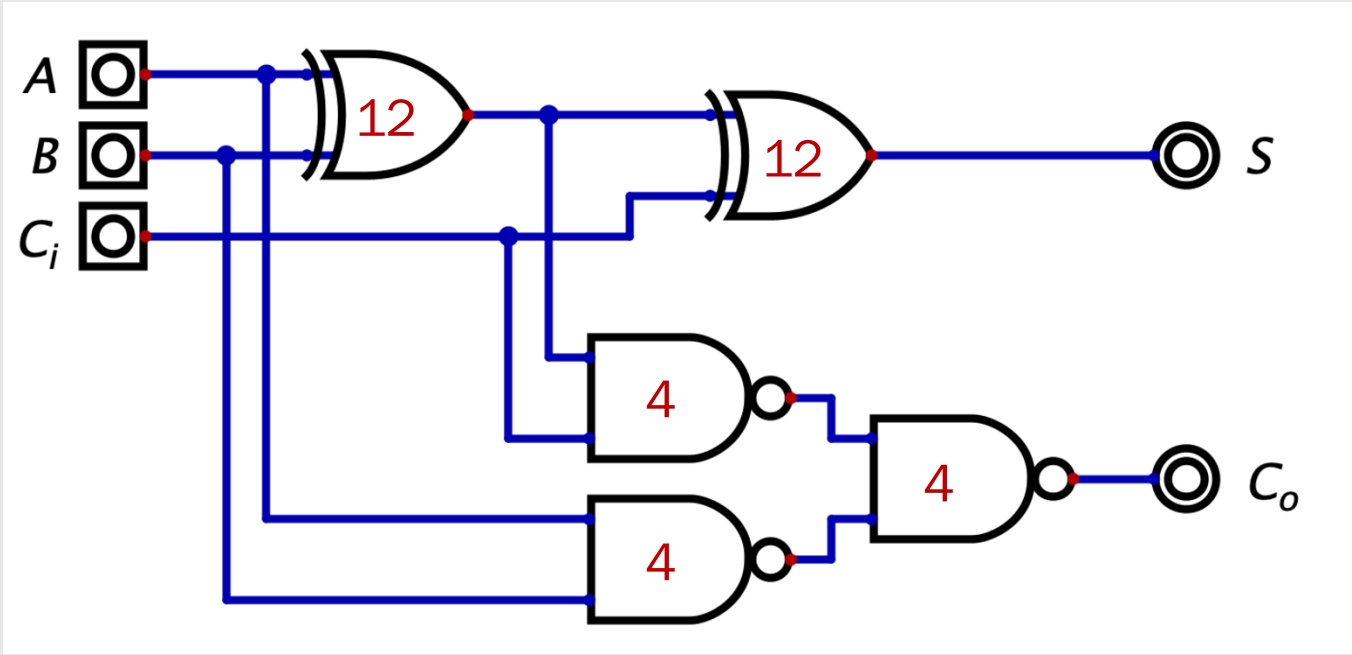
$$C_O = C_i(A \oplus B) + AB$$



Full Adder: Reducing Transistor Count

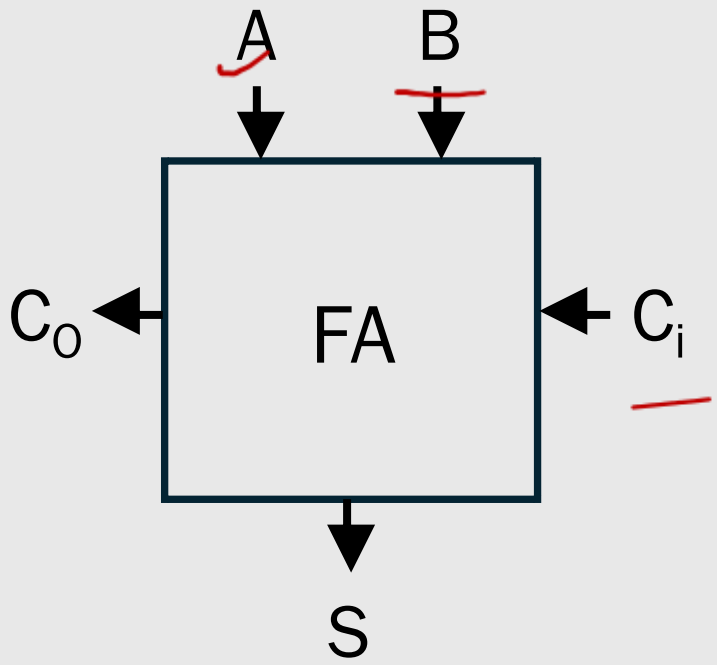
$$S = A \oplus B \oplus Ci$$

$$C_O = C_i(A \oplus B) + AB$$

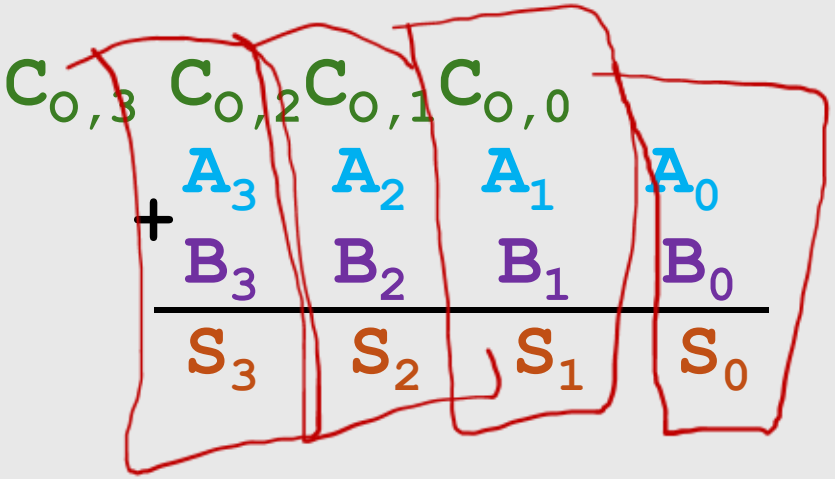


Transistor Count: 36

Full Adder Abstraction

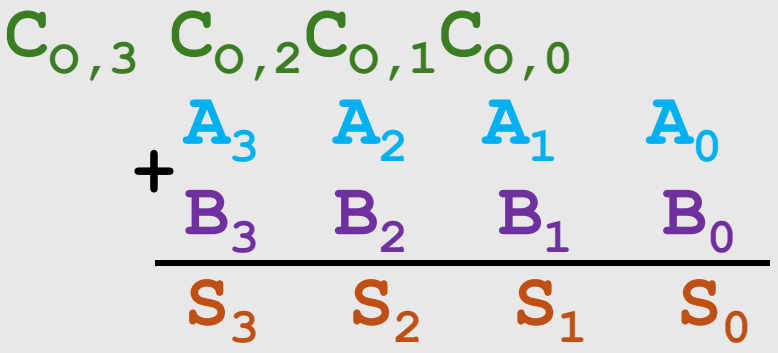


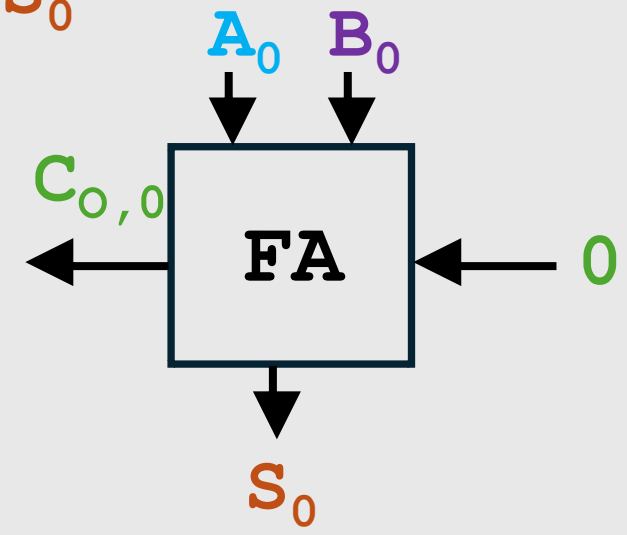
4-bit Adder



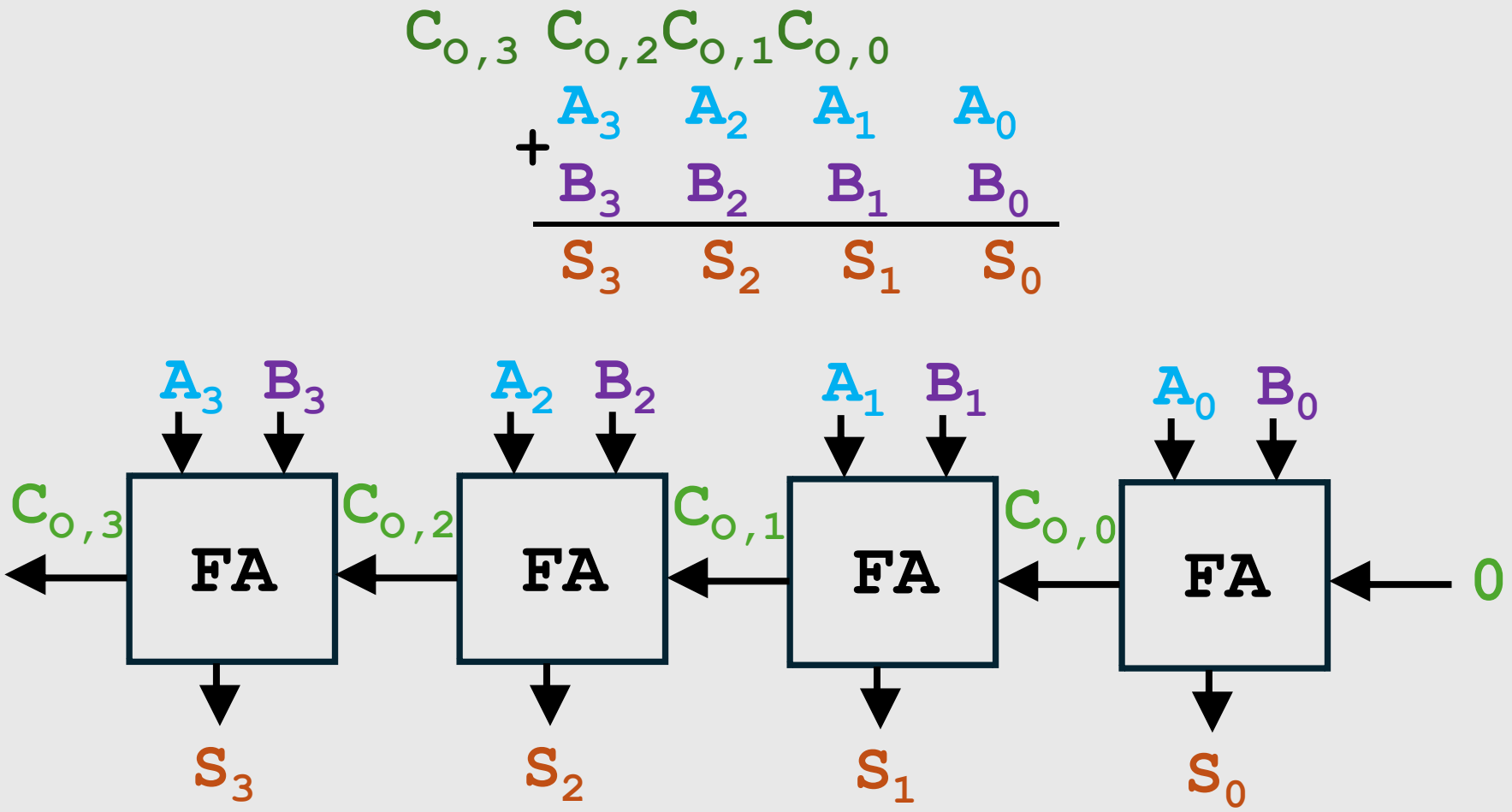
4-bit Adder

- How would you use 4 full-adders to add together A and B?





4-bit Adder


$C_{O,3}$ $C_{O,2}$ $C_{O,1}$ $C_{O,0}$
A_3 A_2 A_1 A_0
+
B_3 B_2 B_1 B_0
S_3 S_2 S_1 S_0
A_3 B_3
A_2 B_2
A_1 B_1
A_0 B_0
$C_{O,3}$
$C_{O,2}$
$C_{O,1}$
$C_{O,0}$
FA
FA
FA
FA
0
S_3
S_2
S_1
S_0

ADDER-SUBTRACTOR CIRCUIT

Recall: Two's Complement

Ex: Represent -15 in two's complement with 7 bits

- To represent a negative number, take the positive representation of the number, flip the bits, and add 1

Unsigned: `0b 000 1111` ⟹ 15

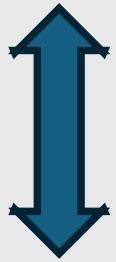
Flip bits: `0b 111 0000`

Add one: `0b 111 0001`

Recall Two's Complement Subtraction

$$A - B == A + (-B)$$

$$\begin{array}{r} -10 \\ -\ \ 4 \\ \hline -14 \end{array} \qquad \begin{array}{r} 10110_2 \\ -\ 00100_2 \\ \hline \end{array}$$



Equivalent Operations

$$\begin{array}{r} -10 \\ +\ -4 \\ \hline -14 \end{array} \qquad \begin{array}{r} 10110_2 \\ +\ 11100_2 \\ \hline 10010_2 \end{array}$$

Subtraction

- Instead of building a separate circuit for subtraction, we will modify the ripple carry adder such that it can perform addition and subtraction
- When we want to perform A – B
  - *We will actually perform A + (-B)*
  - *We need to modify our circuit such that it can negate B*

Subtraction

- Let's say that we want to perform the operation A – B where A = 5 and B = 1.
- We will modify the circuit such that it can flip each bit of B and add 1.

$$\begin{array}{r} 0101_2 \\ - \; 0001_2 \\ \hline \end{array}$$

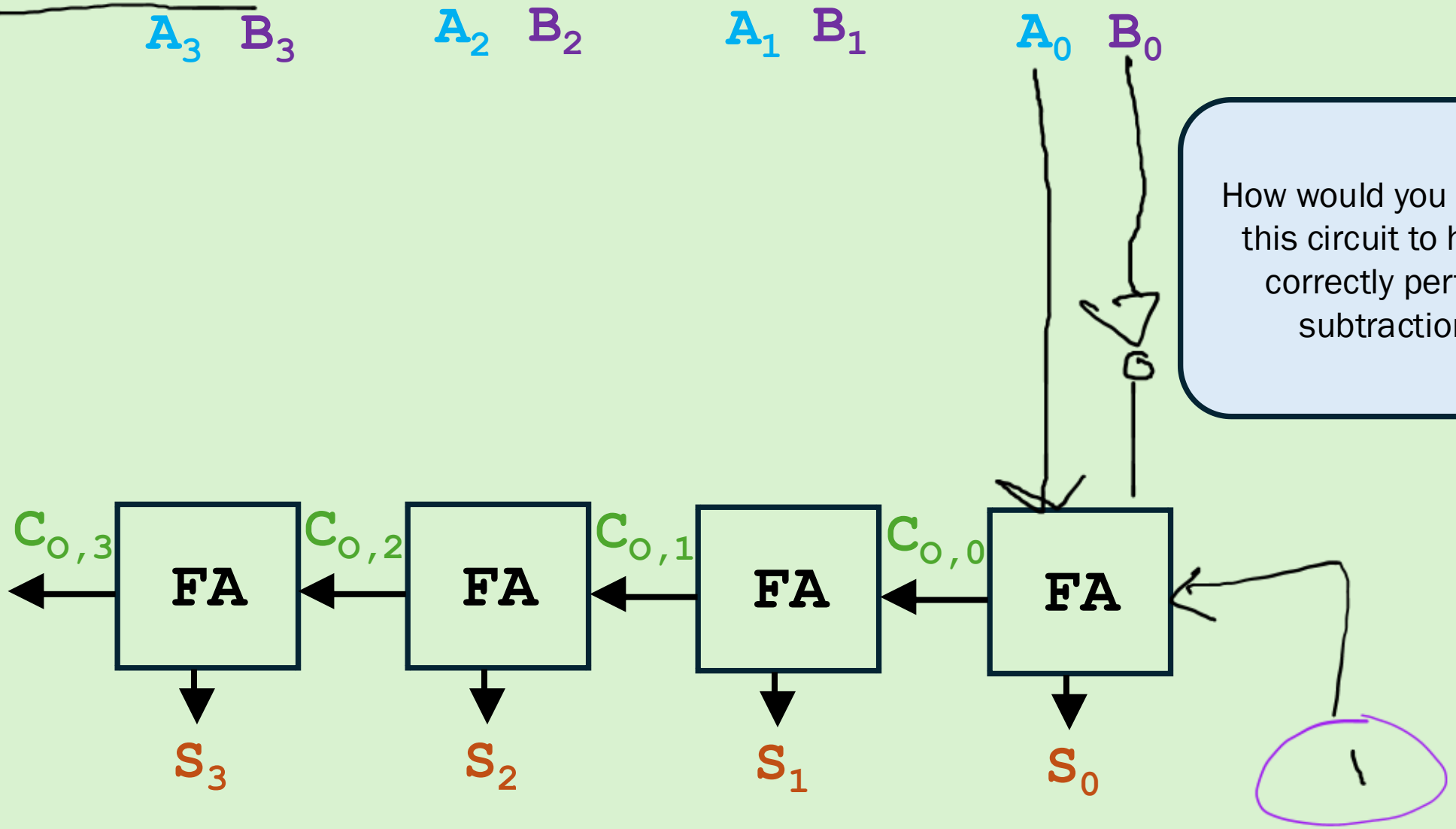
$$\begin{array}{r} 0101_2 \\ + \; 1110_2 \\ + \; \quad 1_2 \\ \hline 0100_2 \end{array}$$

Keep A the same

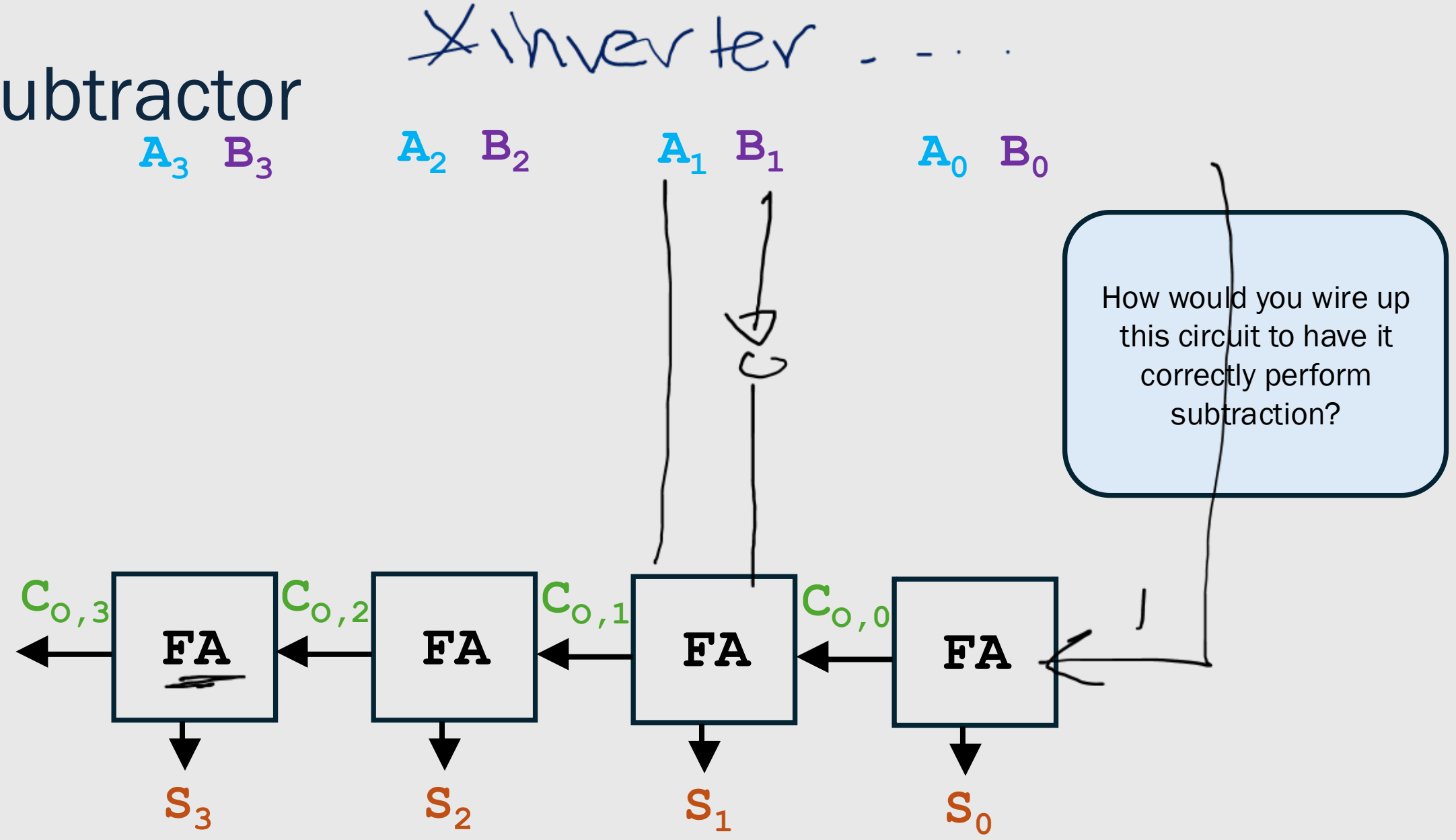
Flip the bits of B

Add 1

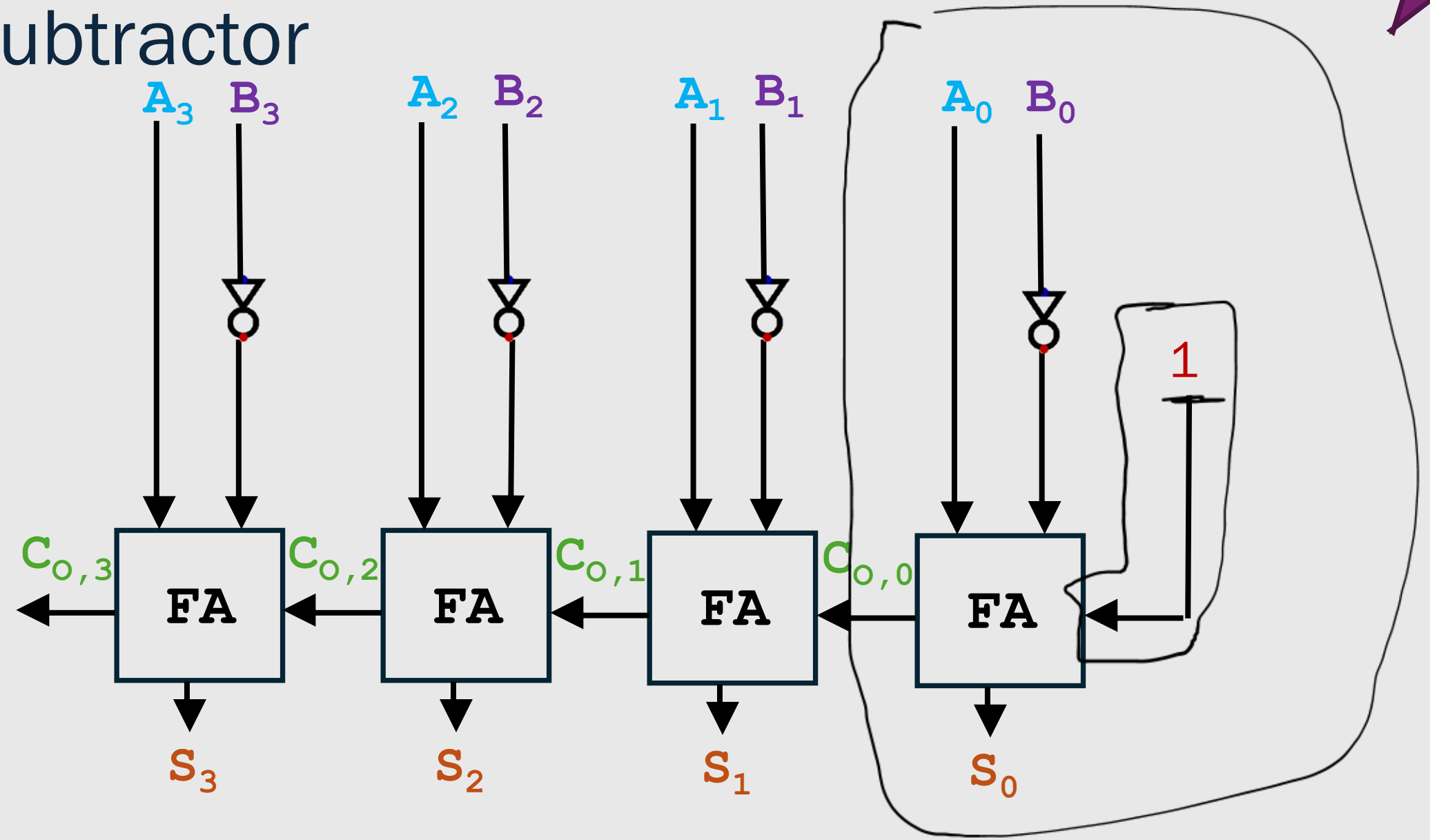
Subtractor


A3 B3
A2 B2
A1 B1
A0 B0
How would you wire up this circuit to have it correctly perform subtraction?
Co,3
Co,2
Co,1
Co,0
FA
FA
FA
FA
S3
S2
S1
S0
1

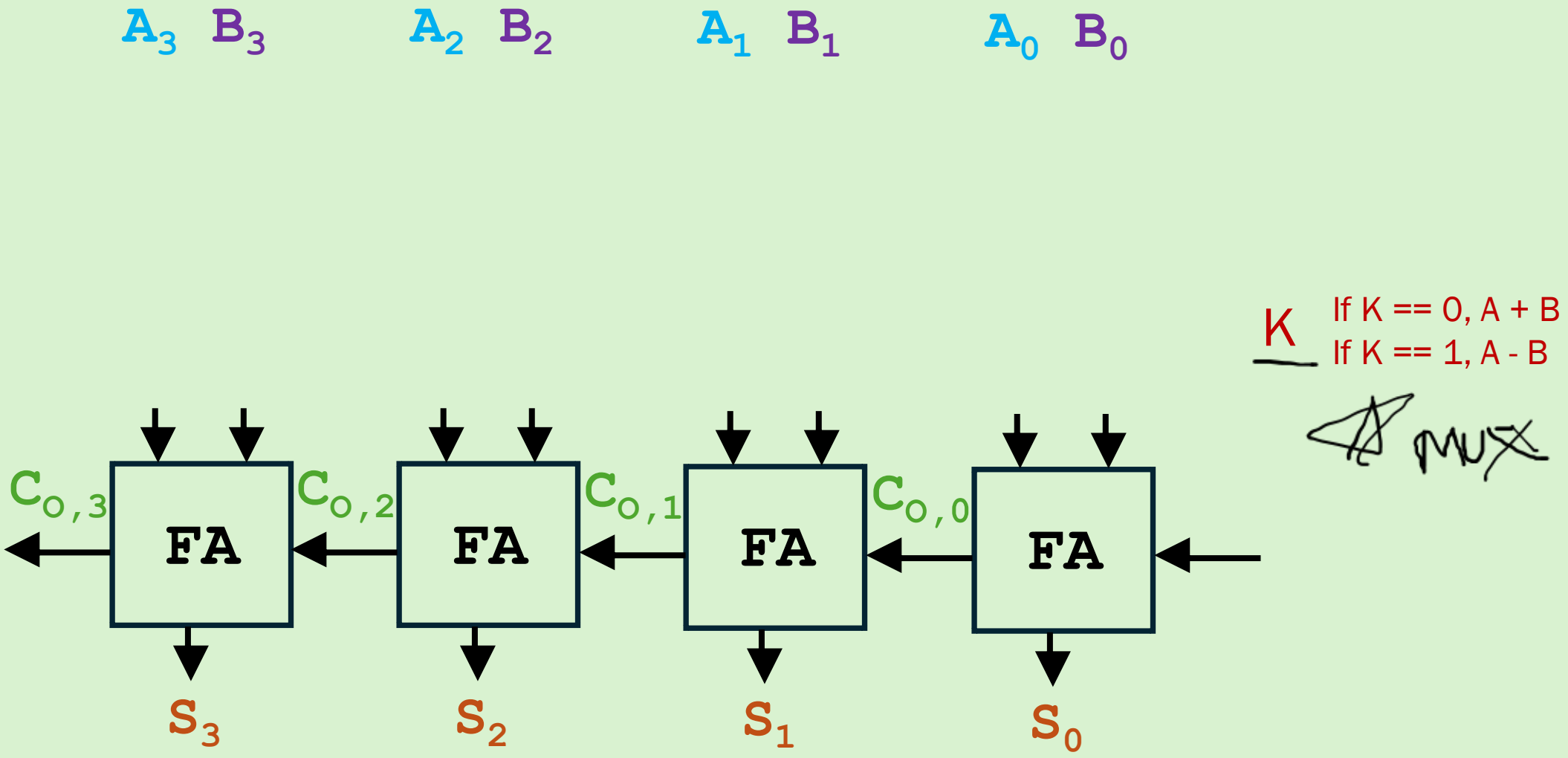
Subtractor


Inverter
A3 B3
A2 B2
A1 B1
A0 B0
How would you wire up this circuit to have it correctly perform subtraction?
Co,3
Co,2
Co,1
Co,0
FA
FA
FA
FA
S3
S2
S1
S0
1

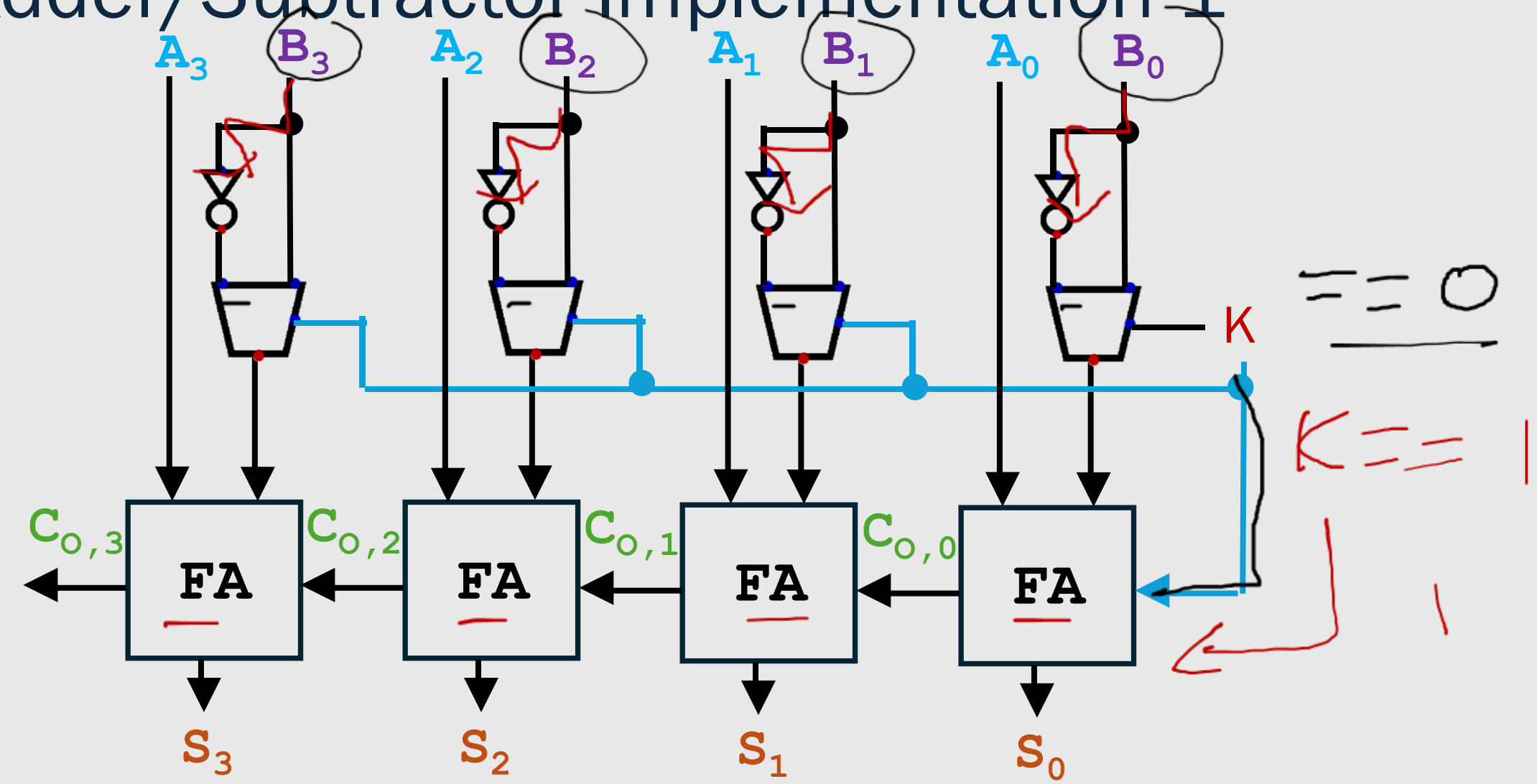
Subtractor



Adder/subtractor



Adder/Subtractor Implementation 1



XOR

$$N \oplus 1 = \overline{N}$$

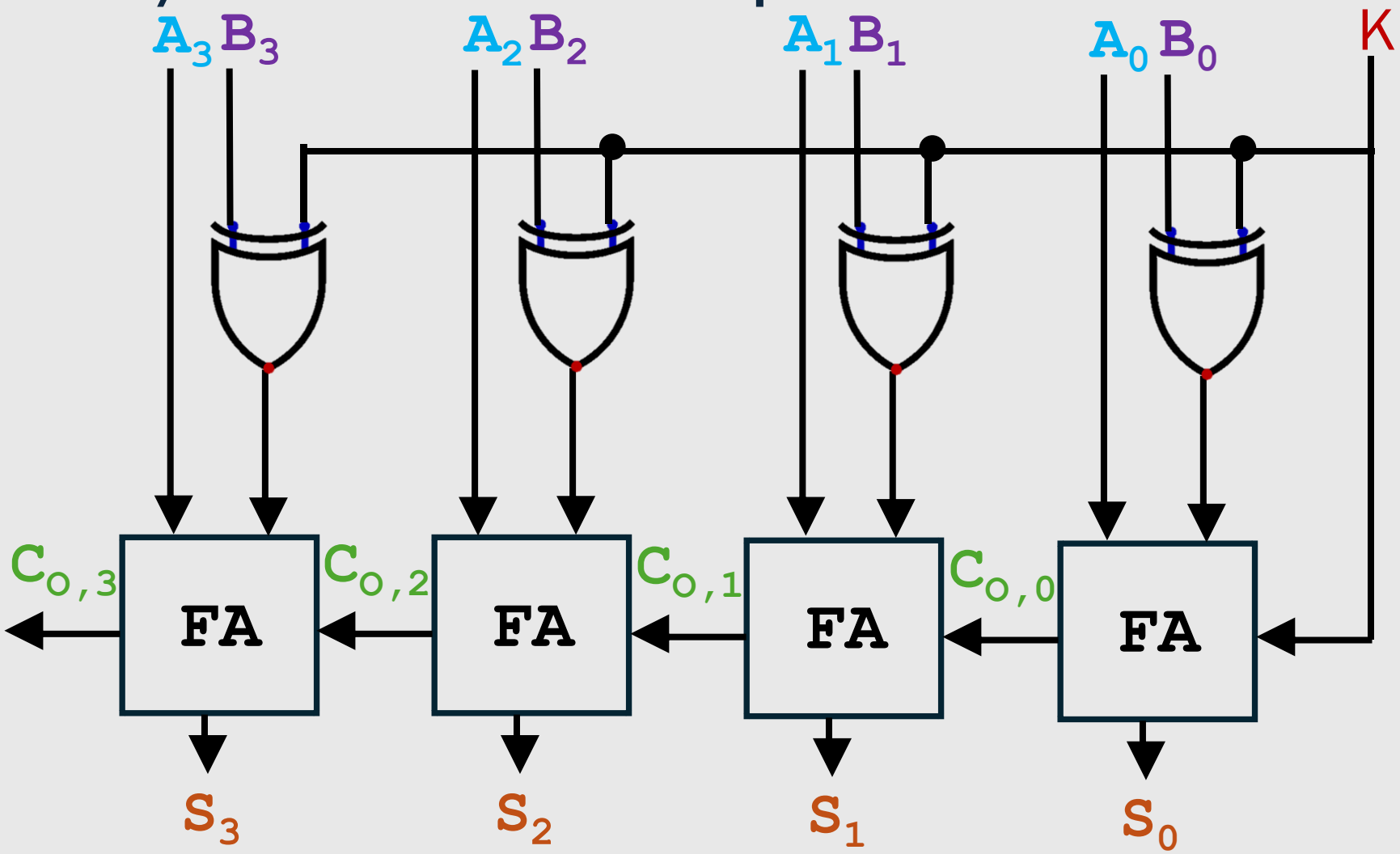
$$N \oplus 0 = N$$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

If we XOR B_i with K

- When K = 0, $B_i \oplus K = B_i$
- When K = 1, $B_i \oplus K = \overline{B_i}$

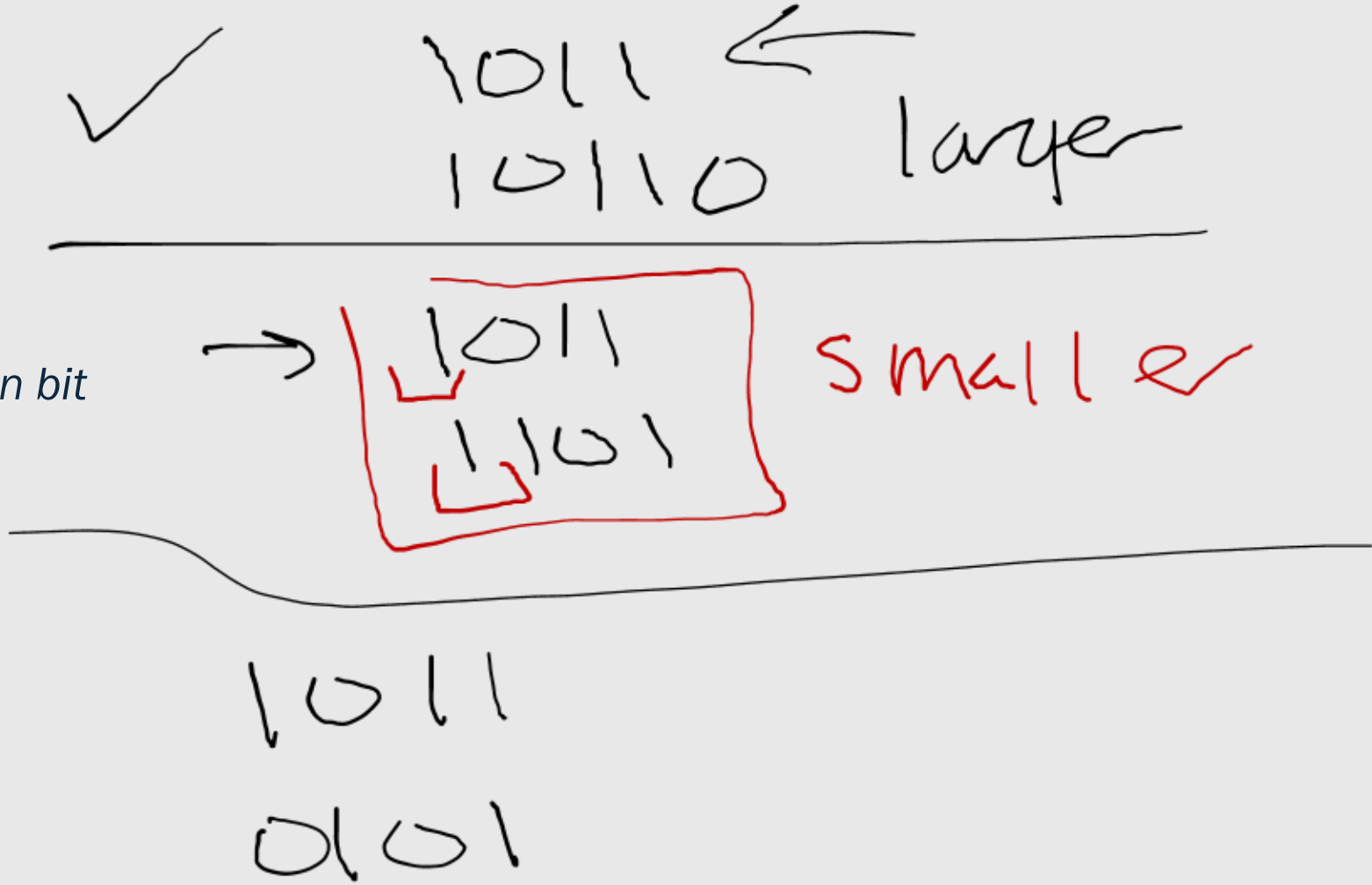
Adder/Subtractor Implementation 2



SHIFTER CIRCUITS

Reminder: Types of Shifts

- A << B
  - *logical shift left*
  - *Fill right side with 0s*
- A >>> B
  - *arithmetic shift right*
  - *Fill left side with the sign bit*
- A >> B
  - *logical shift right*
  - *Fill left side with 0s*



Left Shifting

- Let's say my input, A, is a 4-bit number: 0b1101

Shift Amount	Result
0	
1	
2	
3	
4	
5	
6	
7	
8	

Left Shifting

- Let's say my input, A, is a 4-bit number: 0b1101

Shift Amount	Result
0	0b1101
1	0b1010
2	0b0100
3	0b1000
4	0b0000
5	0b0000
6	0b0000
7	0b0000
8	0b0000

Shift Amount

- Let's say my input, A, is a 4-bit number: 0b1101

Shift Amount	Result
0	0b1101
1	0b1010
2	0b0100
3	0b1000
4	0b0000
5	0b0000
6	0b0000
7	0b0000
8	0b0000

If we shift A by more than 3 bits, the result will always be 0. We want to make our shifter circuit as simple as possible, so we will only support a shift amount range of 0-3.

Shift Amount

Shift Amount	Result
0	0b1101
1	0b1010
2	0b0100
3	0b1000
4	0b0000
5	0b0000
6	0b0000
7	0b0000
8	0b0000

We need to be able to shift by up to 3 bits.

To represent the numbers 0-3, we need 2 bits.

Shift Amount

Shift Amount	Result
0	0b10101
1	0b01010
2	0b10100
3	0b01000
4	0b10000
5	0b00000
6	0b00000
7	0b00000
8	0b00000

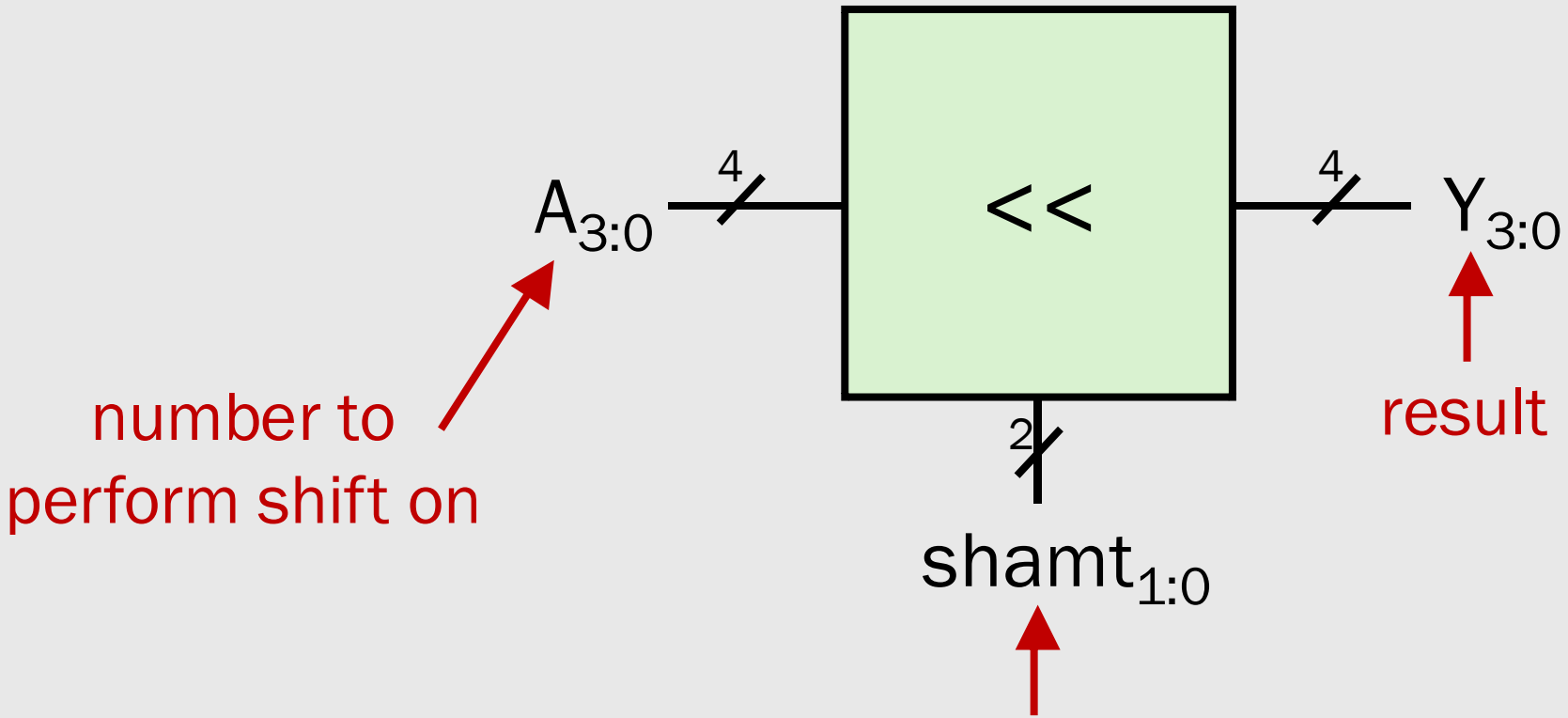
We need to be able to shift by up to 4 bits.

To represent the numbers 0-4, we need 3 bits.

For an n-bit number, need ceil($\log_2 n$) bits for the shift amount

Left Shift

- Design a circuit that can perform a left shift operation on a 4-bit number.



Left Shift

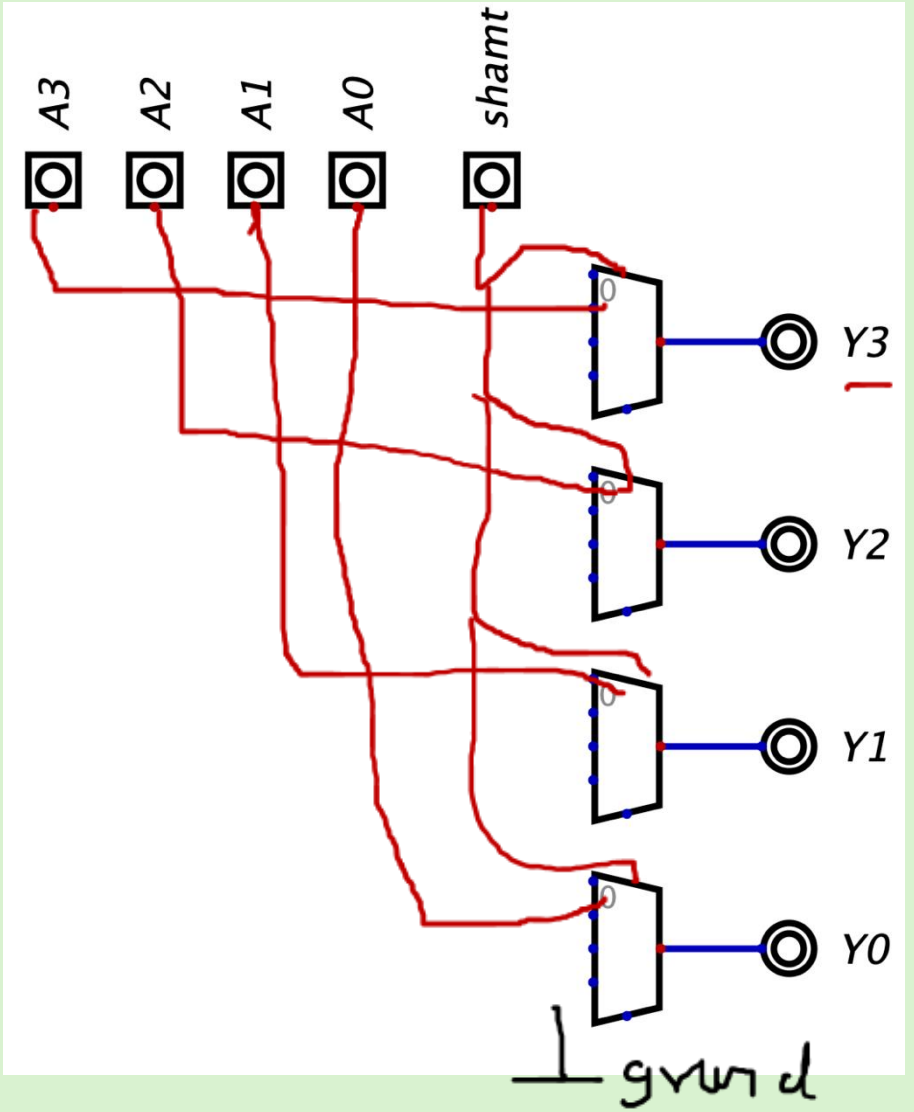
Output Value

Shift Amount

	Y_3	Y_2	Y_1	Y_0
0	A_3	A_2	A_1	A_0
1	A_2	A_1	A_0	0
2	A_1	A_0	0	0
3	A_0	0	0	0

Left Shift

- Design a circuit that can perform a variable left shift on a 4-bit number.
  - *shamt is a 2-bit input*
  - *you may use ground for 0*

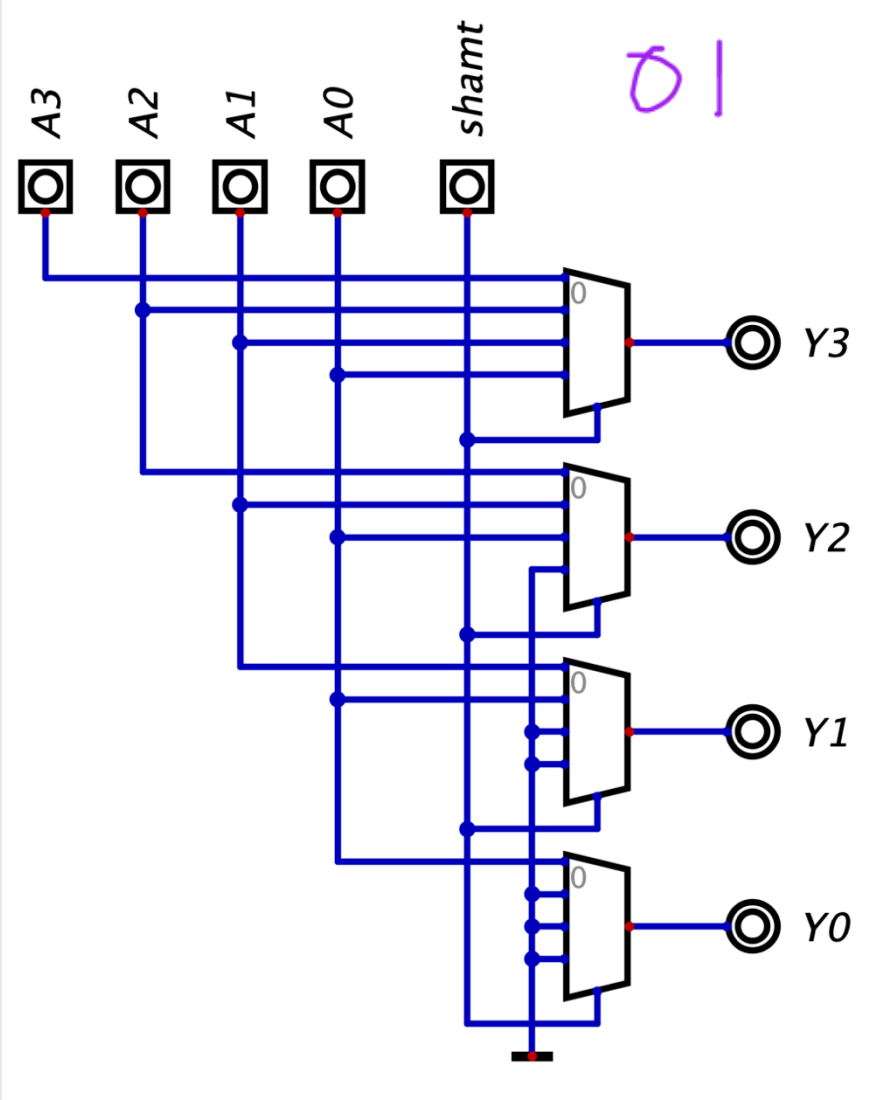


Left Shift

- Design a circuit that can perform a variable left shift on a 4-bit number.

1011

01



R-TYPE & I-TYPE INSTRUCTIONS

What is the course about?

High-level programming languages

```
// High-level (C)
int add3(int a, int b)
{
    return a + b + 3;
}
```

Easy for humans to read/write

Assembly
Low-level languages

```
# Matching RISC-V assembly
# a0 = a, a1 = b;  return in a0

add   a0, a0, a1   # a0 = a0 + a1
addi  a0, a0, 3    # a0 = a0 + 3
ret                # return
```

Human readable

Machine code

```
0000000 01011 01010
000 01010 0110011

000000000011 01010
000 01010 0010011

000000000000 00001
000 00000 1100111
```

Machine-readable

Square in different assembly languages

MIPS

x86

ARM

RISC-V

```
square(int):
    addi    sp, sp, -16       # make stack frame
    sw      ra, 12(sp)        # save return address
    sw      s0, 8(sp)         # save frame pointer
    addi    s0, sp, 16        # set up new frame pointer
    sw      a0, -4(s0)        # save argument x

    lw      t0, -4(s0)        # t0 = x
    mul     t0, t0, t0        # t0 = x * x
    mv      a0, t0            # move result to a0

    lw      ra, 12(sp)        # restore return address
    lw      s0, 8(sp)         # restore frame pointer
    addi    sp, sp, 16        # pop stack frame
    jr      ra                # return
```

```
square(int):
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        ldr     r3, [r7, #4]
        mul     r3, r3, r3
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
```

Square in different assembly languages

MIPS

```
square(int):
        addiu   $sp,$sp,-8
        sw      $fp,4($sp)
        move    $fp,$sp
        sw      $4,8($fp)
        lw      $2,8($fp)
        nop
        mult    $2,$2
        mflo    $2
        move    $sp,$fp
        lw      $fp,4($sp)
        addiu   $sp,$sp,8
        jr      $31
        nop
```

x86

```
square(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, eax
        pop     rbp
        ret
```

ARM

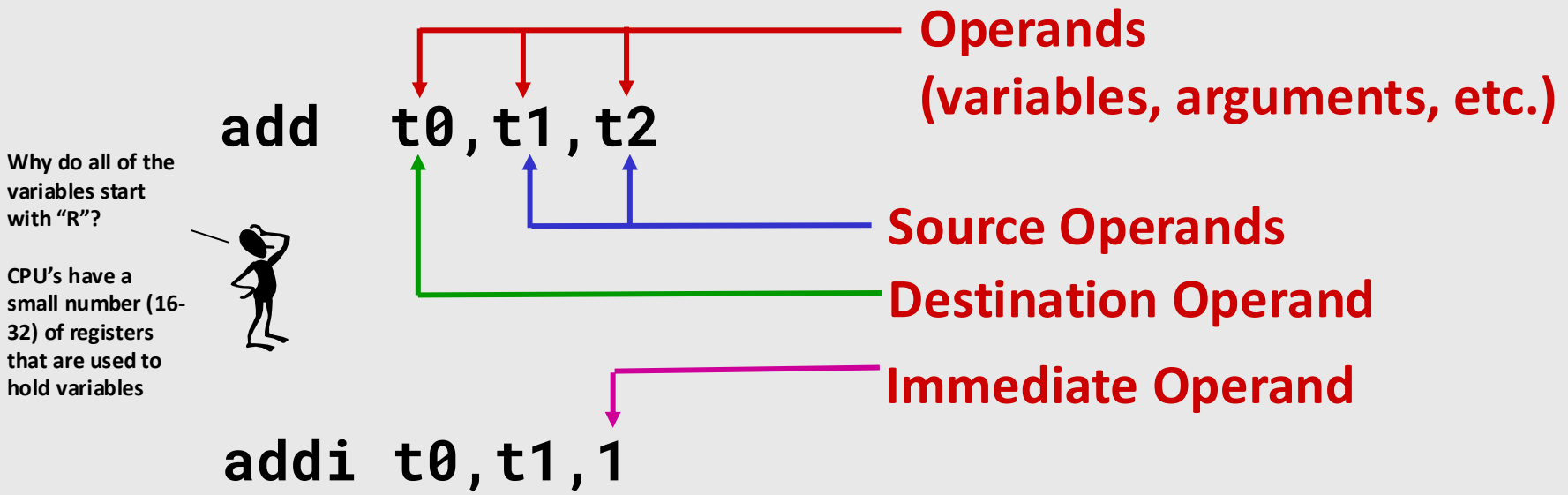
```
square(int):
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        ldr     r3, [r7, #4]
        mul     r3, r3, r3
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
```

Instruction Set Architecture (ISA)

- A set of rules that define the interface between the hardware and software.

Anatomy of an Instruction

- Computers execute a set of primitive operations called instructions
- Instructions specify an operation and its operands (arguments of the operation)
- Types of operands: destination, source, and immediate



Instruction Set Architecture (ISA)

Encoding of instructions raises some interesting choices…

- Trade Offs: performance, compactness, programmability
- Uniformity. Should different instructions
  - *Be the same size (number of bits)?*
  - *Take the same amount of time to execute?*
  - *Trend: Uniformity. Affords simplicity, speed, pipelining.*
- Complexity. How many different instructions? What level operations?
  - *Level of support for particular software operations: array indexing, procedure calls, "polynomial evaluate", etc*
  - ***"Reduced Instruction Set Computer"***
    (RISC) philosophy: simple instructions, optimized for speed
- Mix of Engineering & Art…

RISC-V 32

RISC-V Register Details

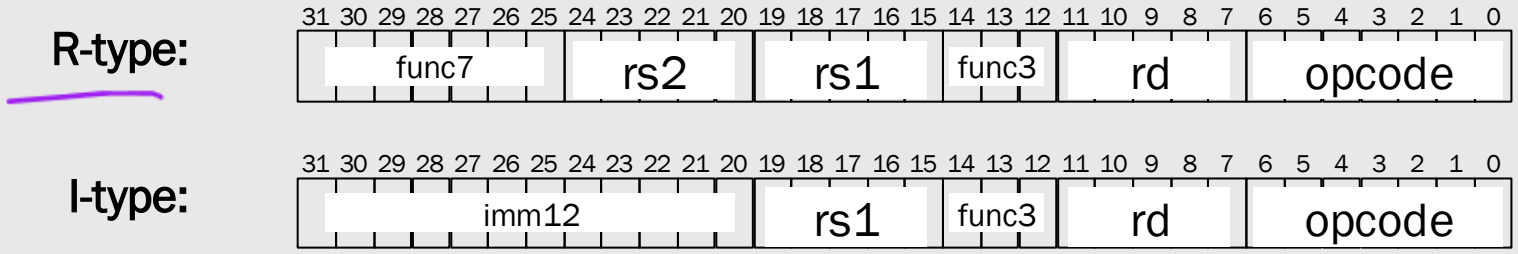
- There are 32 named registers [x0, x1, …. x31]
- x0 is special. It always contains "0" and, when used as a destination, the result is ignored
- The operands of most instructions are registers
- This means to operate on a variables in memory you must:
  - *Load the value/values from memory into a register*
  - *Perform the instruction*
  - *Store the result back into memory*
- Going to and from memory can be expensive (4x to 20x slower than operating on a register)
- Net effect: Keep variables in registers as much as possible!
- By convention, most registers are dedicated to specific tasks

A.K.A a "Load-Store Architecture"

Basic RISC-V Instructions

- Instructions include various “fields” that encode combinations of OPCODES and arguments
- special fields enable extended functions
- several 5-bit OPERAND fields, for specifying the sources and destination of the operation, usually one of the 32 registers
- Embedded constants (“immediate” values) of various sizes,

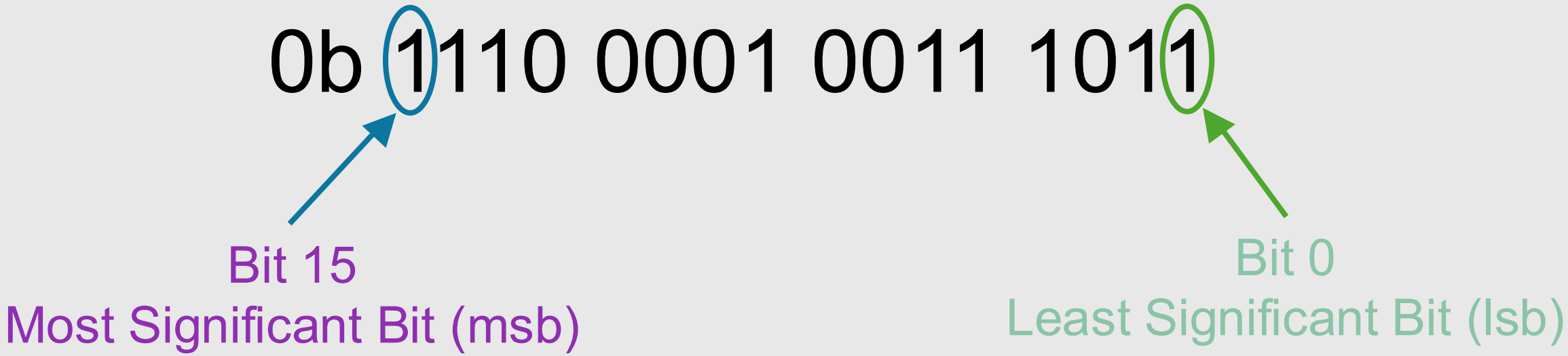
The basic data-processing instruction formats:



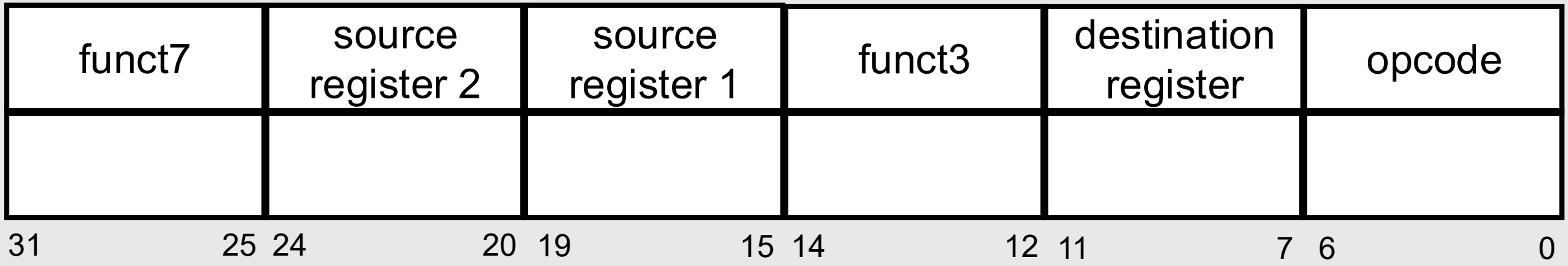
Reading Binary

0b 1110 0001 0011 1011

Reading Binary



r1 = r2 + r3; add x1, x3, x2



r1 = r2 + r3; add x3, x2, x1

add x1, x3, x2

funct7	source register 2	source register 1	funct3	destination register	opcode
0000000	00011	00010	000	00001	0110011
31 25	24 20	19 15	14 12	11 7	6 0

3 2

0000000 00011 00010 000 00001 0110011

3 2 1

Opcode is always 0110011 for integer R-type ALU instructions
funct3 is always the primary operation code group (e.g. 000 = add/sub. 100 = xor, 111 = and)
funct7 is used to further specify the operation (e.g. distinguishes add vs. sub. srl vs. sra)