CSC 220: Computer Organization

Unit 10
Arithmetic-Logic Units

Prepared by:

Md Saiful Islam, PhD

Department of Computer Science

College of Computer and Information Sciences

Overview

- Arithmetic Unit Design
  - Primitive gates base implementation
  - MUX-based implementation
- Logic Unit Design
- Arithmetic-logic Unit Design
- Function Unit Design
  - Combinational Shifter

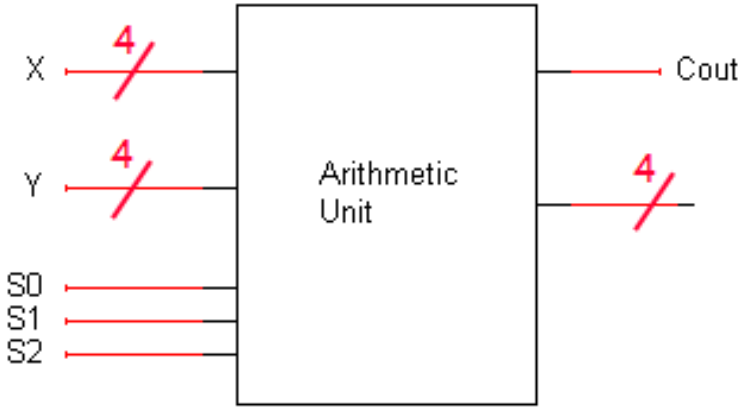
Chapter-8

M. Morris Mano, Charles R. Kime and Tom Martin, **Logic and Computer Design Fundamentals**, Global (5th) Edition, Pearson Education Limited, 2016. ISBN: 9781292096124

Arithmetic Unit Design

Designing a simple 4-bit AU

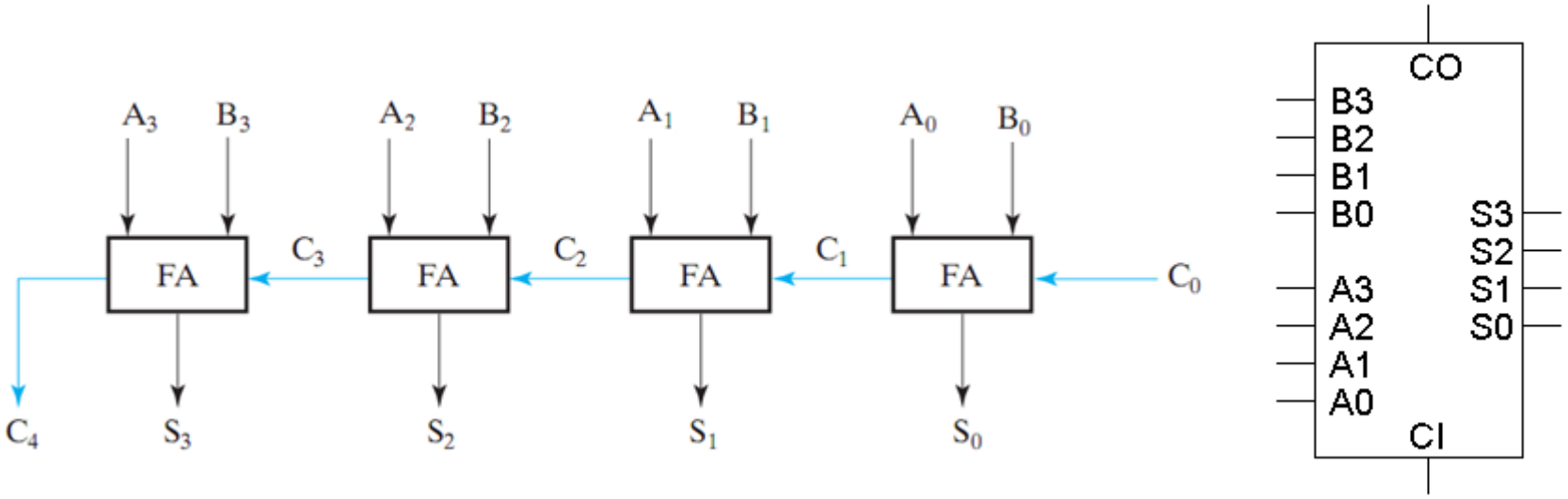
- **8 arithmetic operations**
- **Inputs:**
  - X (4 bits)
  - Y (4 bits)
  - S (3 bits)
- **Outputs:**
  - G (4 bits)
  - C_{out} (final carry)
- The / and 4 on a line indicate that it's actually *four* lines



S_2	S_1	S_0	Operation
0	0	0	G = X
0	0	1	G = X + 1
0	1	0	G = X + Y
0	1	1	G = X + Y + 1
1	0	0	G = X + Y'
1	0	1	G = X + Y' + 1
1	1	0	G = X – 1
1	1	1	G = X

The four-bit parallel adder

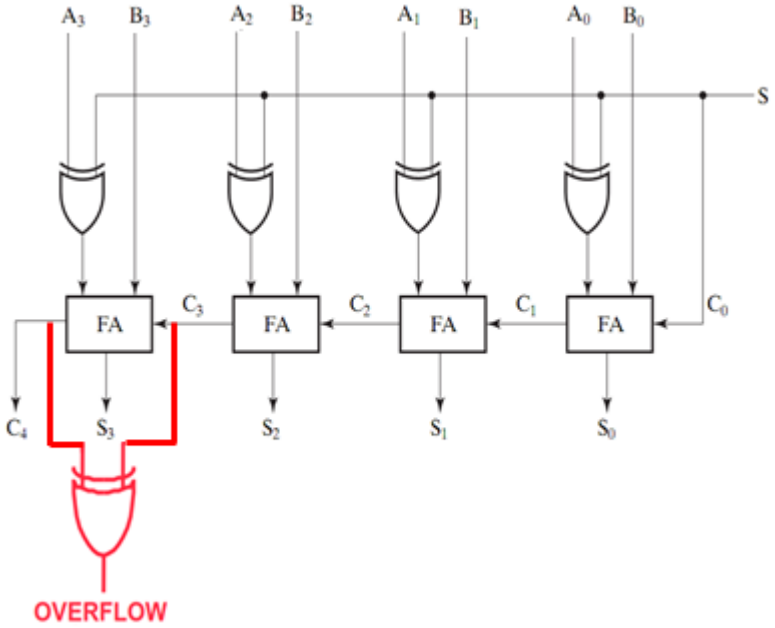
- **The basic four-bit adder *always* computes S = A + B + CI.**

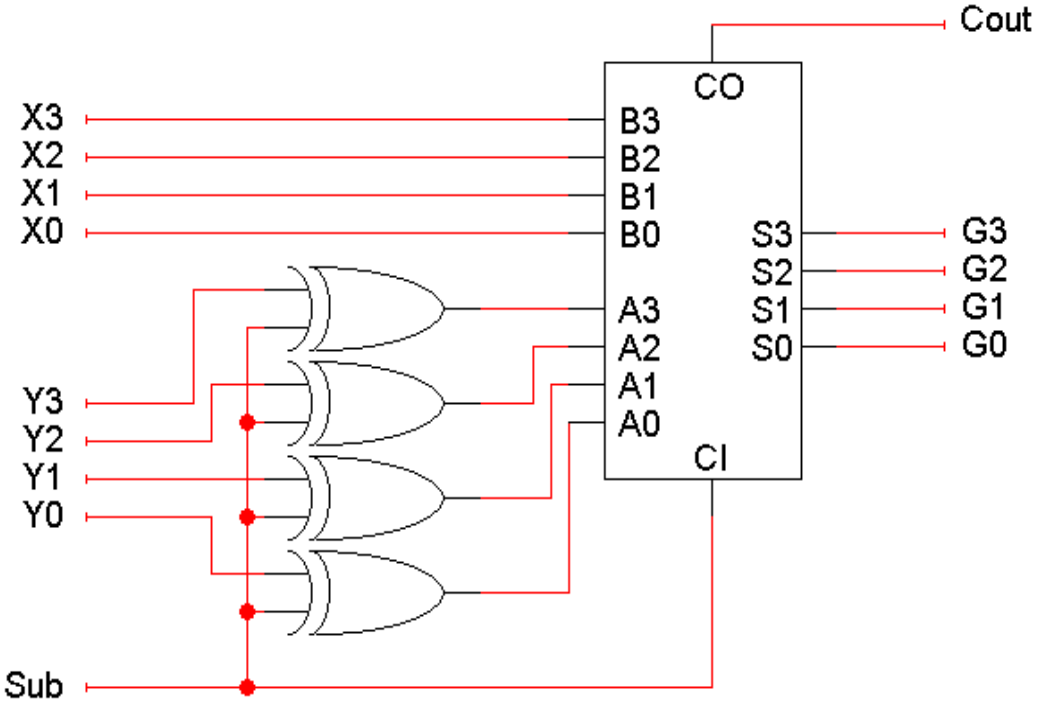


- **But by changing what goes into the adder inputs A, B and CI, we can change the adder output S.**
- **This is also what we did to build the combined adder-subtractor circuit.**

The adder-subtractor

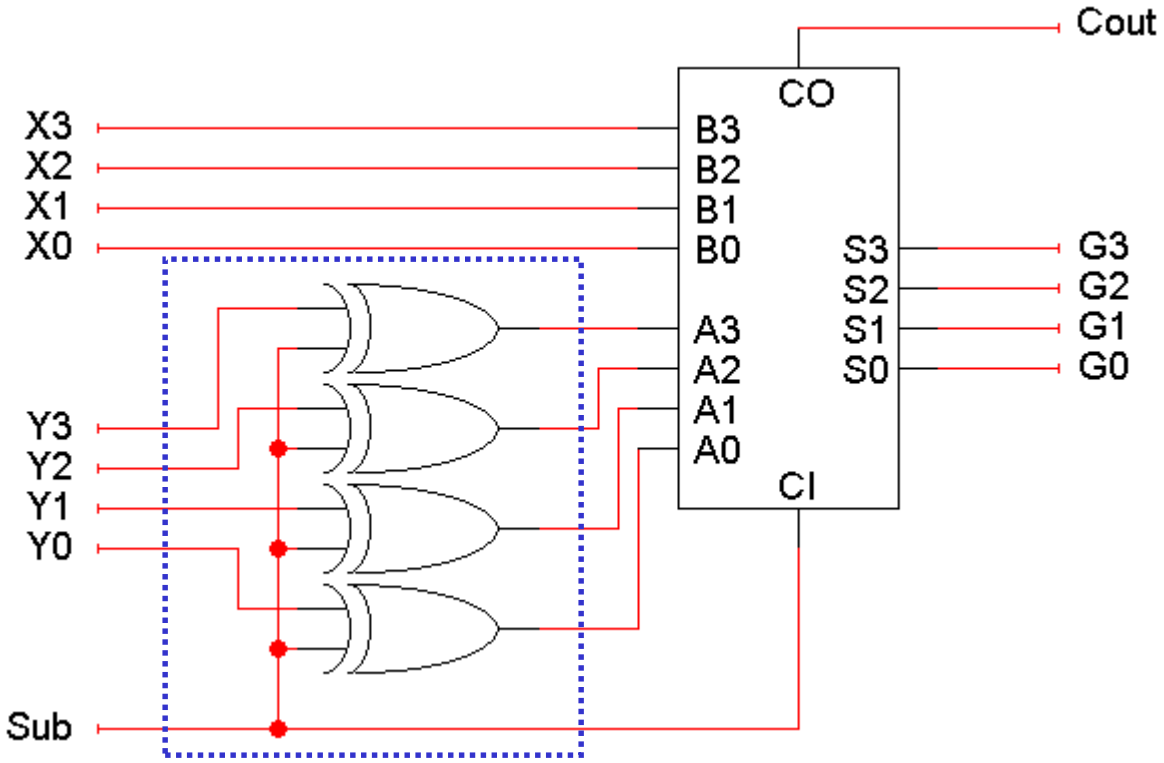
- Here the signal Sub and some XOR gates alter the adder inputs.
  - When Sub = 0, the adder inputs A, B, CI are Y, X, 0, so the adder produces G = X + Y + 0, or just X + Y.
  - When Sub = 1, the adder inputs are Y', X and 1, so the adder output is G = X + Y' + 1, or the two's complement operation X - Y.





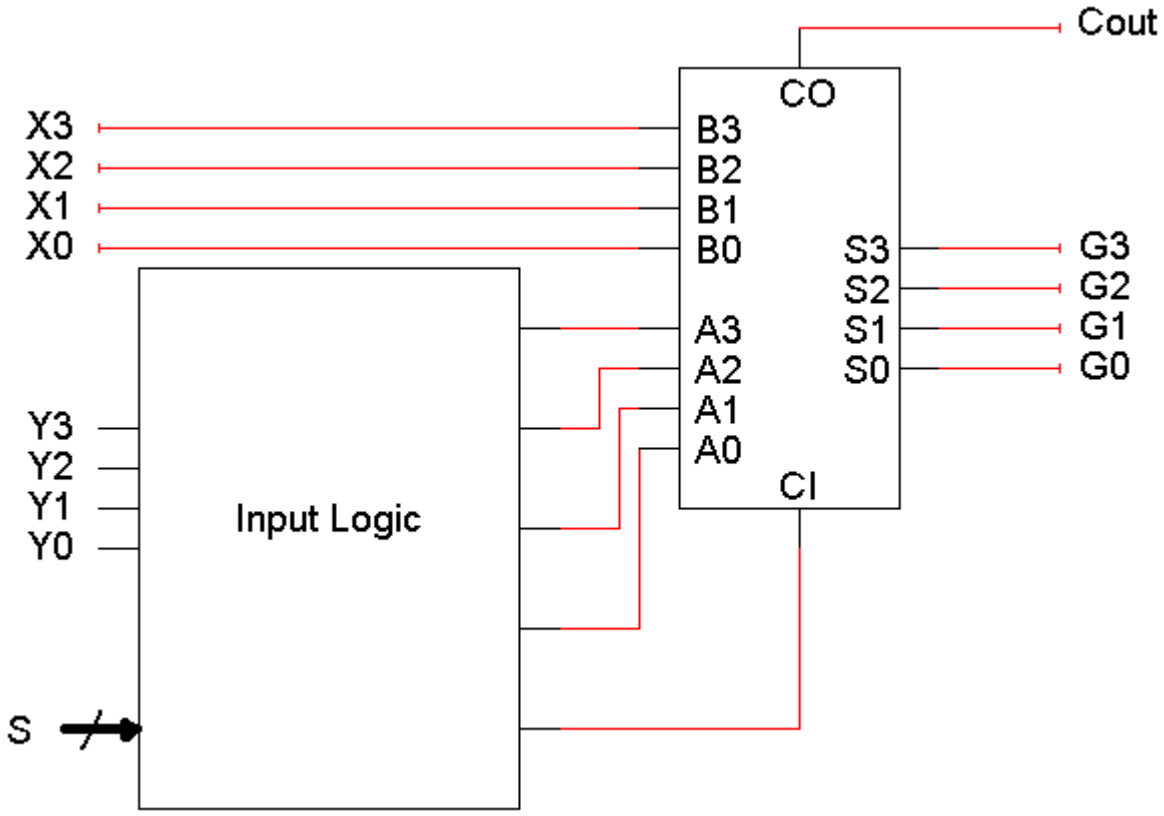
The multi-talented adder

- So we have one adder performing two separate functions.
- "Sub" acts like a function select input which determines whether the circuit performs addition or subtraction.
- "Sub" modifies the adder's inputs A and CI.



Modifying the adder inputs

- By following the same approach, we can use an adder to compute *other* functions as well.
- We just have to figure out which functions we want, and then put the right circuitry into the "Input Logic" box .



Some more possible functions

- We already saw how to set adder inputs A, B and CI to compute either X + Y or X - Y.
- How can we produce the increment function G = X + 1?

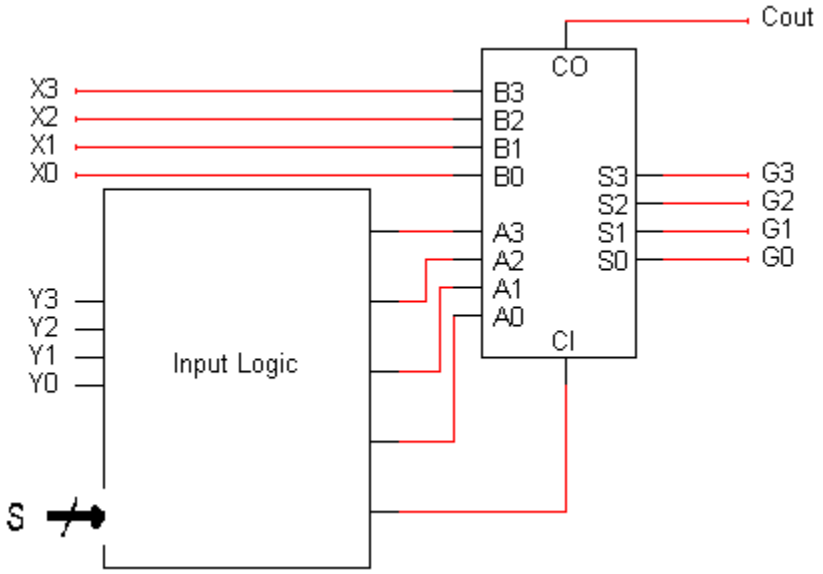
One way: Set A = 0000, B = X, and CI = 1

- How about decrement: G = X - 1?

A = 1111 (-1), B = X, CI = 0

- How about transfer: G = X? (This can be useful.)

A = 0000, B = X, CI = 0



This is almost the same as the increment function!

The role of CI

- The transfer and increment operations have the same A and B inputs, and differ only in the CI input.
- In general we can get additional functions (not all of them useful) by using both CI = 0 and CI = 1.
- Another example:
  - Two's-complement subtraction is obtained by setting A = Y', B = X, and CI = 1, so G = X + Y' + 1.
  - If we keep A = Y' and B = X, but set CI to 0, we get G = X + Y'. This turns out to be a ones' complement subtraction operation.

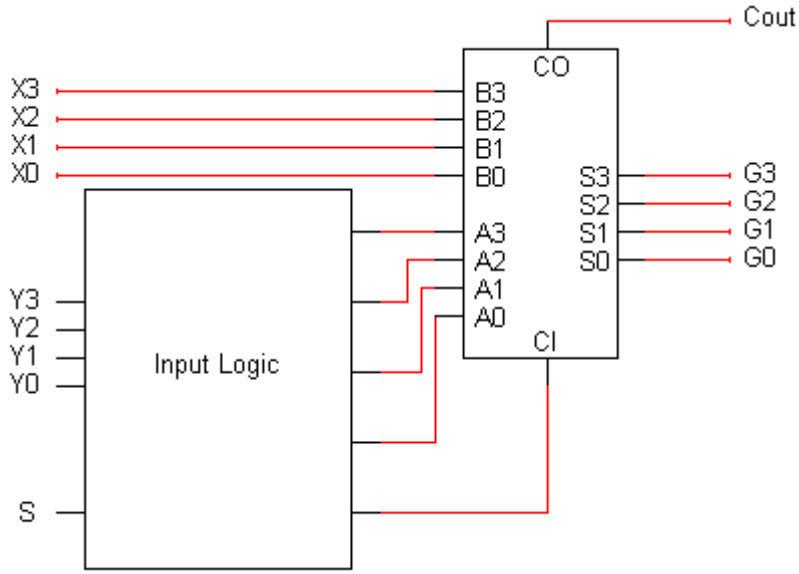
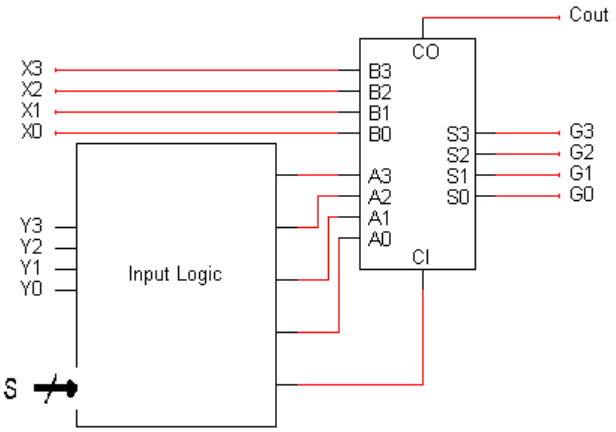


Table of arithmetic functions

- Here are some of the different possible arithmetic operations.
- We'll need some way to specify which function we're interested in, so we've *randomly assigned* a selection code to each operation.

S_2	S_1	S_0	Arithmetic operation	
0	0	0	X	(transfer)
0	0	1	X + 1	(increment)
0	1	0	X + Y	(add)
0	1	1	X + Y + 1	
1	0	0	X + Y'	(1C subtraction)
1	0	1	X + Y' + 1	(2C subtraction)
1	1	0	X - 1	(decrement)
1	1	1	X	(transfer)



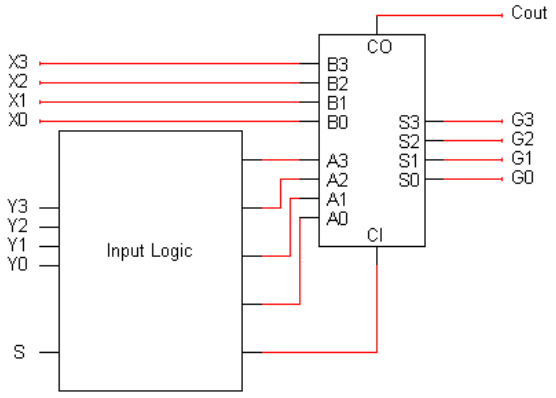
Mapping the table to an adder

- **This second table shows what the adder's inputs should be for each of our eight desired arithmetic operations.**

Selection code S_2	S_1	S_0	Desired arithmetic operation G (A + B + CI)		Required adder inputs A	B	CI
0	0	0	X	(transfer)	0000	X	0
0	0	1	X + 1	(increment)	0000	X	1
0	1	0	X + Y	(add)	Y	X	0
0	1	1	X + Y + 1		Y	X	1
1	0	0	X + Y′	(1C subtraction)	Y′	X	0
1	0	1	X + Y′ + 1	(2C subtraction)	Y′	X	1
1	1	0	X – 1	(decrement)	1111	X	0
1	1	1	X	(transfer)	1111	X	1

  - **Adder input CI is always the same as selection code bit S_0.**
  - **B is always set to X.**
  - **A depends only on S_2 and S_1.**

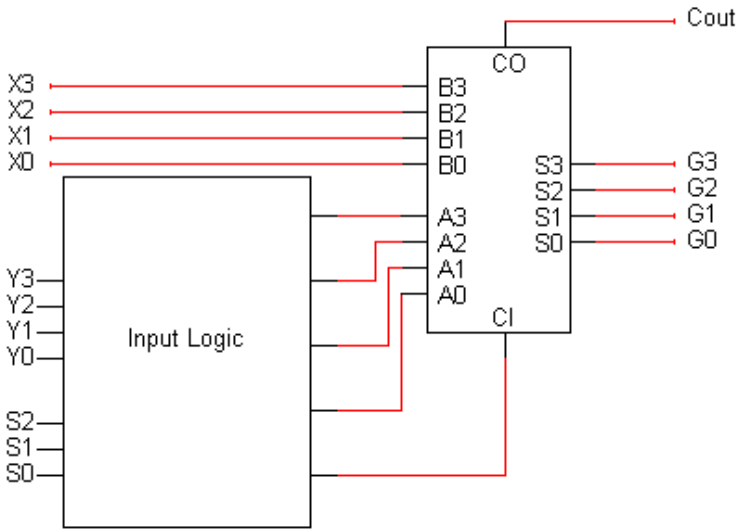
- **These equations depend on both the desired operations and the assignment of selection codes.**



Building the input logic

- **All we need to do is compute the adder input A, given the arithmetic unit input Y and the function select code S (actually just S_2 and S_1).**
- **Here is an abbreviated truth table:**

S_2	S_1	A
0	0	0000
0	1	Y
1	0	Y′
1	1	1111

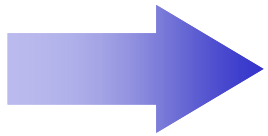


- **We want to pick one of these four possible values for A, depending on S_2 and S_1.**

Primitive gate-based input logic

- We could build this circuit using primitive gates.
- If we want to use K-maps for simplification, then we should first expand out the abbreviated truth table.
  - The Y that appears in the output column (A) is actually an input.
  - We make that explicit in the table on the right.
- Remember A and Y are each 4 bits long!

S_2	S_1	A
0	0	0000
0	1	Y
1	0	Y'
1	1	1111



inputs output

S_2	S_1	Y_i	A_i
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Primitive gate implementation

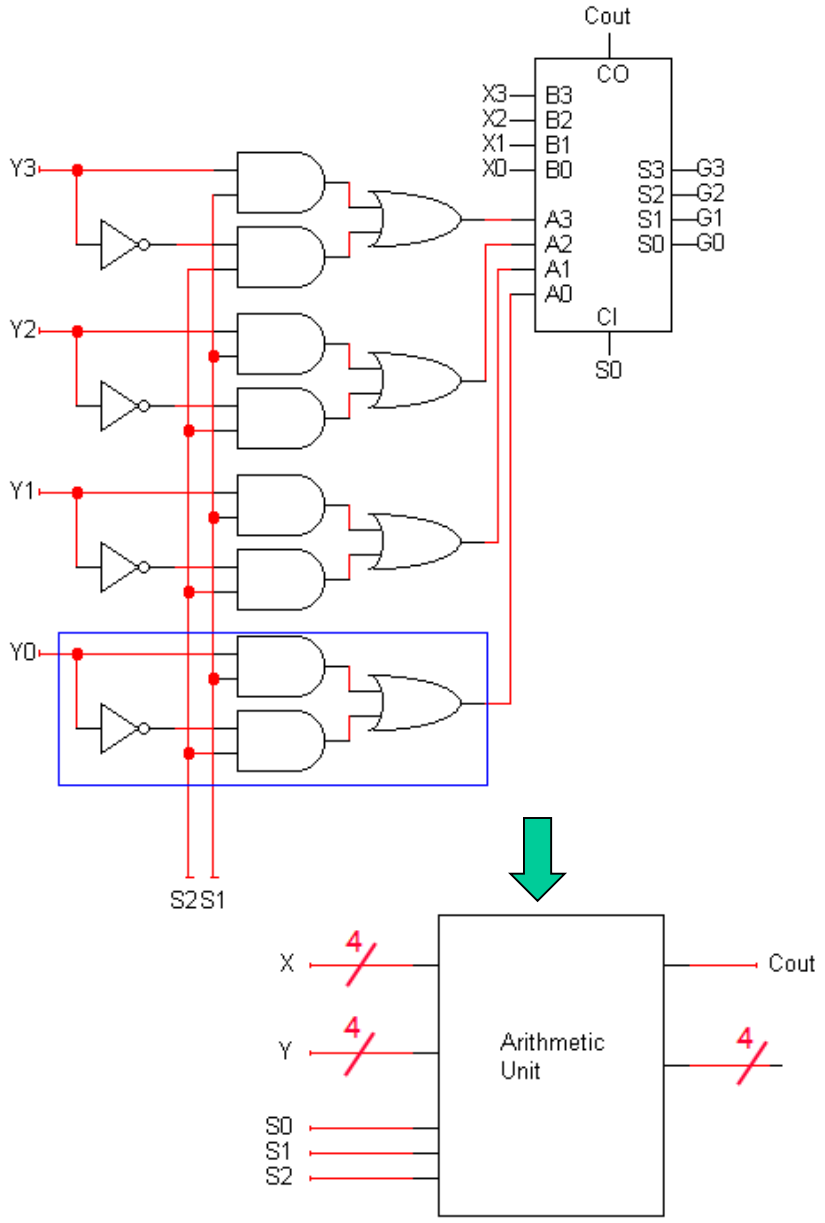
- **From the truth table, we can find an MSP:**

			S_1	
	0	0	1	0
S_2	1	0	1	1
		Y_i		

$$A_i = S_2 Y_i' + S_1 Y_i$$

- **Again, we have to repeat this once for each bit Y3-Y0, connecting to the adder inputs A3-A0.**

- **This completes our arithmetic unit.**

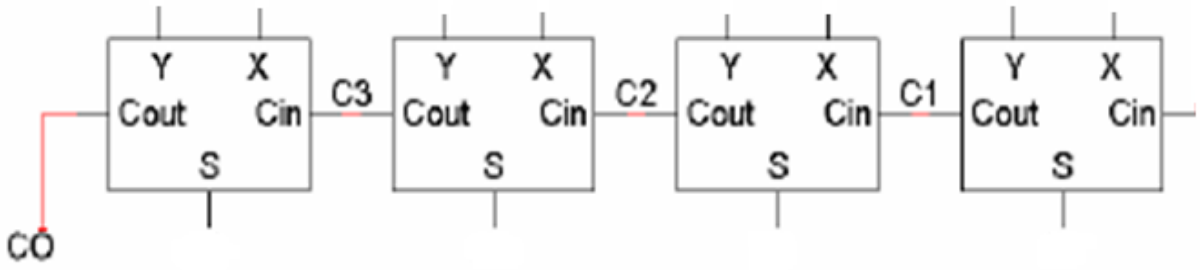


Multiplexer-based implementation

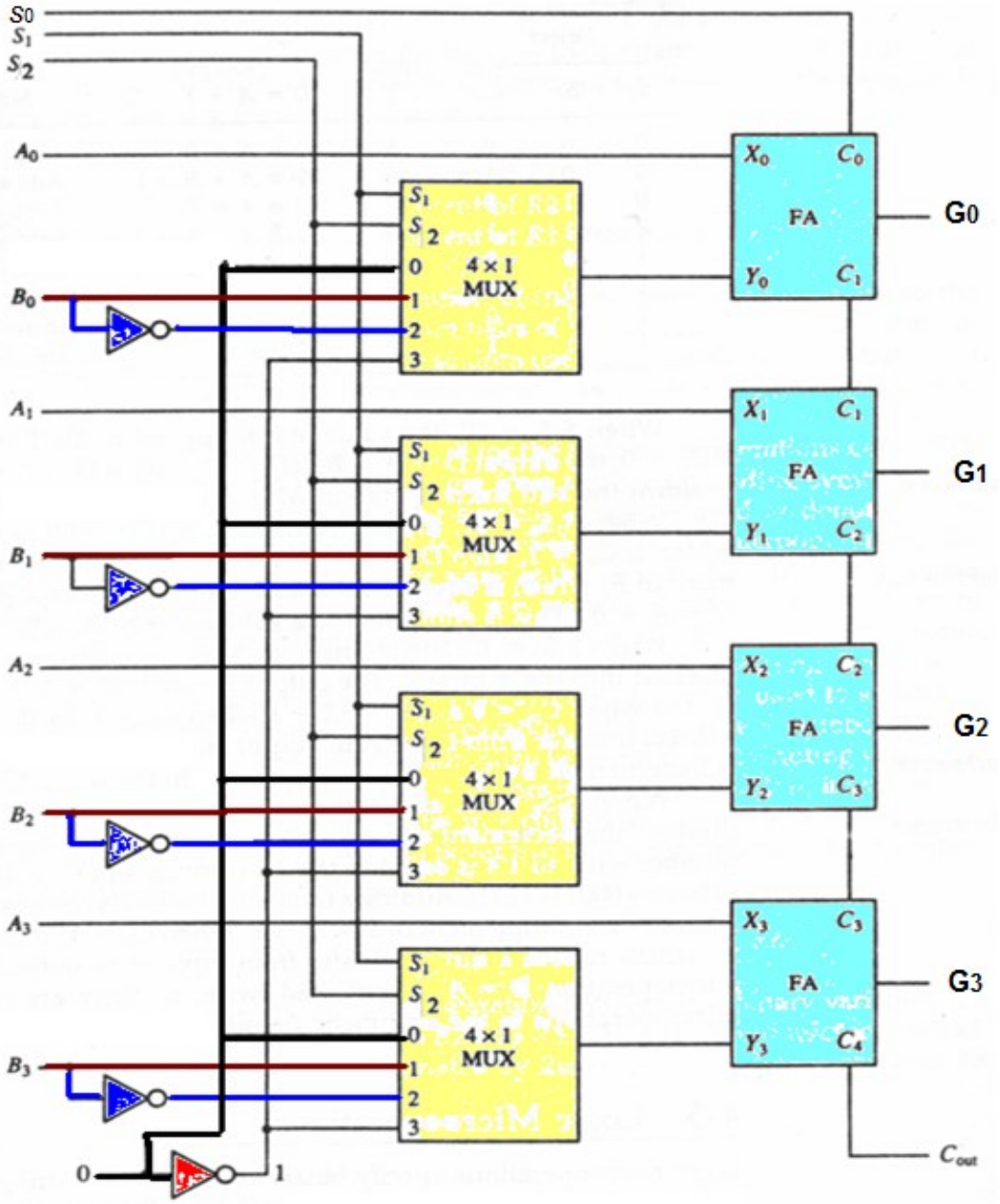
Alternative Implementation using 4 bit adder circuit and multiplexers

Selection code			Desired arithmetic operation		Required adder inputs		
S_2	S_1	S_0	G (X + Y + CI)		Y	X	CI
0	0	0	A	(transfer)	0000	A	0
0	0	1	A + 1	(increment)	0000	A	1
0	1	0	A + B	(add)	B	A	0
0	1	1	A + B + 1		B	A	1
1	0	0	A + B'	(1C subtraction)	B'	A	0
1	0	1	A + B' + 1	(2C subtraction)	B'	A	1
1	1	0	A - 1	(decrement)	1111	A	0
1	1	1	A	(transfer)	1111	A	1

S_2	S_1	Y
0	0	0000
0	1	B
1	0	B'
1	1	1111



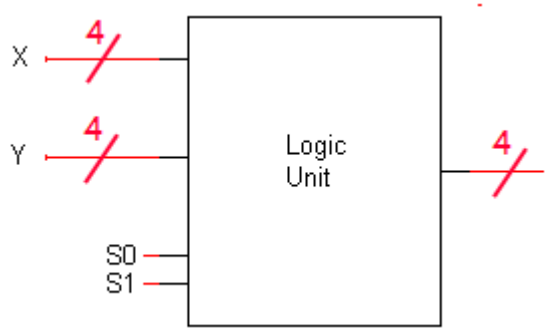
Multiplexer-based implementation



S_2	S_1	y
0	0	0000
0	1	B
1	0	B'
1	1	1111

Logic Unit Design

- **Most computers also support logical operations like AND, OR, XOR and NOT, but extended to multi-bit words instead of just single bits.**



- **Inputs:**
  - **X (4 bits)**
  - **Y (4 bits)**
  - **S (2 bits)**
- **Outputs:**
  - **G (4 bits)**

S_1	S_0	Output
0	0	$G = X \wedge Y$
0	1	$G = X \vee Y$
1	0	$G = X \oplus Y$
1	1	$G = X'$

- **Bitwise operations: To apply a logical operation to two words X and Y, apply the operation on each pair of bits X_i and Y_i:**

```
      1 0 1 1          1 0 1 1          1 0 1 1
AND   1 1 1 0    OR    1 1 1 0    XOR   1 1 1 0
-------------    -------------    -------------
      1 0 1 0          1 1 1 1          0 1 0 1
```

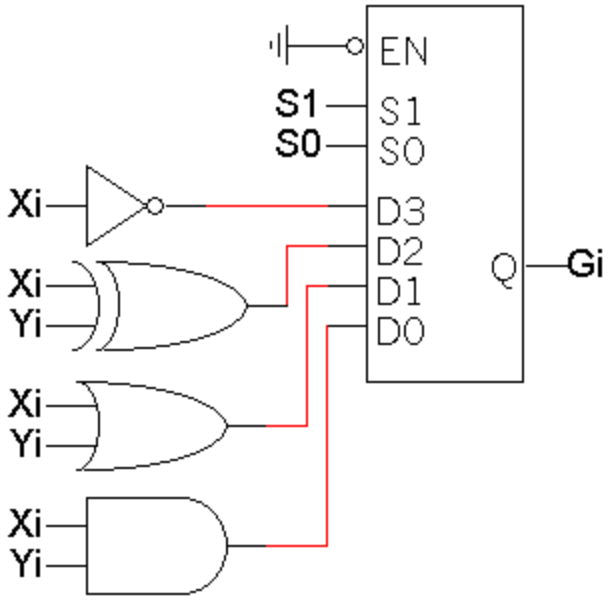
- **Single operand logical operation: "complementing" all the bits in a number.**

```
NOT   1 0 1 1
-------------
      0 1 0 0
```

Defining a logic unit

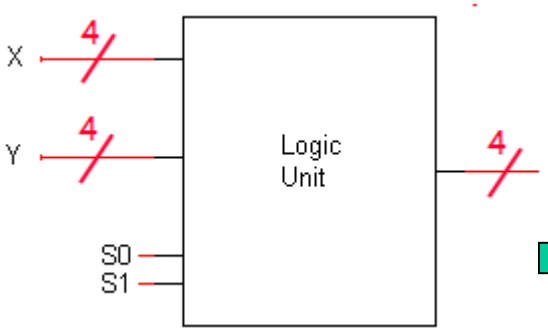
- A logic unit supports different logical functions on two multi-bit inputs X and Y, producing an output G.
- This abbreviated table shows four possible functions and assigns a selection code S to each.

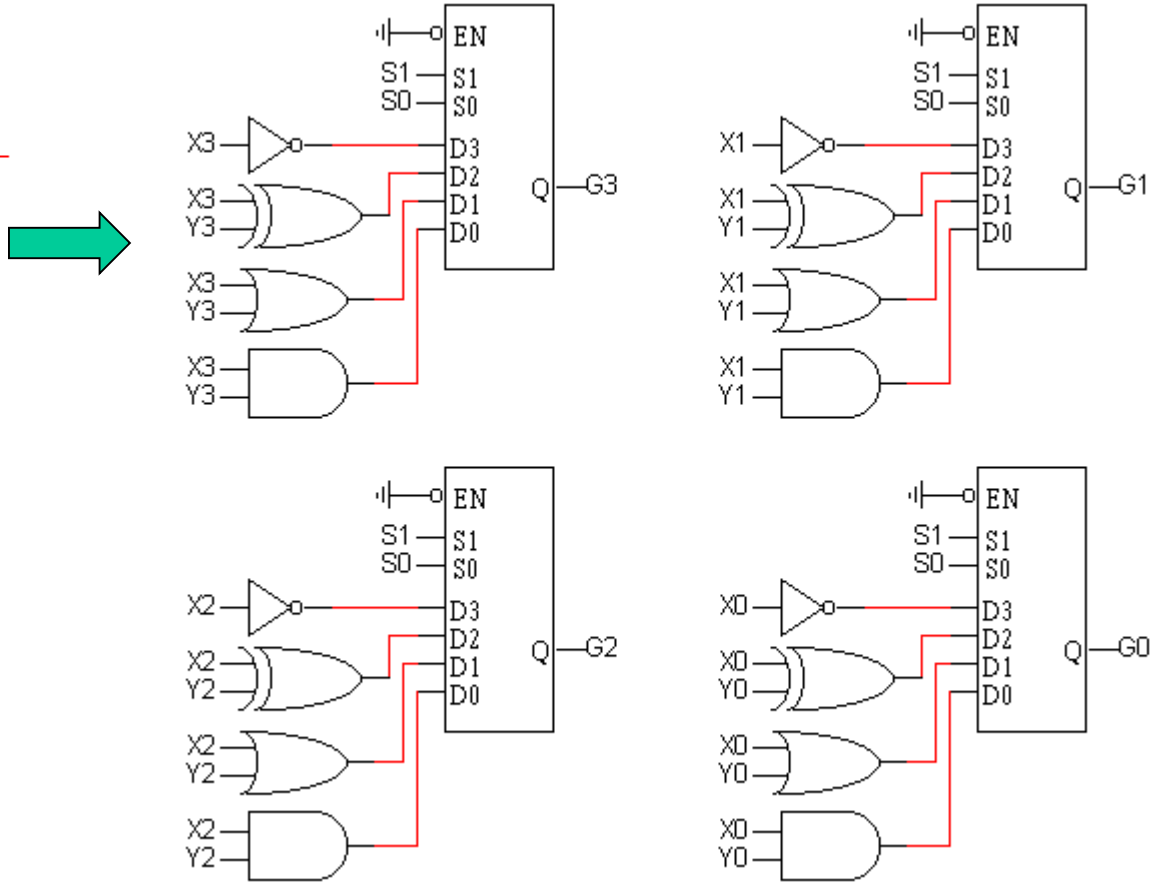
S_1	S_0	Output
0	0	$G_i = X_i Y_i$
0	1	$G_i = X_i + Y_i$
1	0	$G_i = X_i \oplus Y_i$
1	1	$G_i = X_i'$



- We'll just use multiplexers and some primitive gates to implement this.
- Again, we need one multiplexer for *each bit* of X and Y.

Our simple logic unit

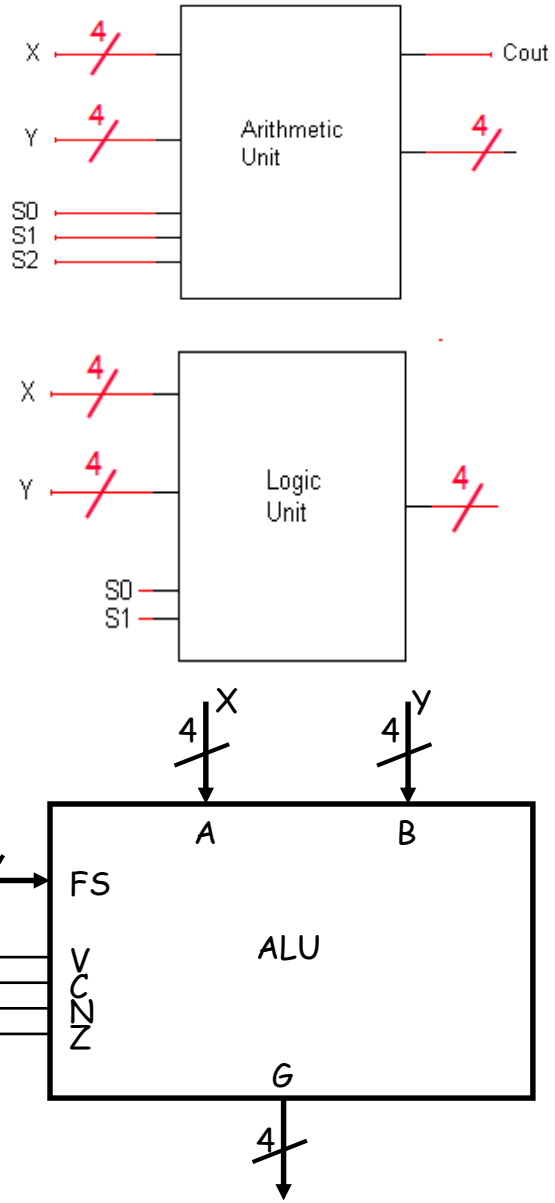




- **Inputs:**
  - **X (4 bits)**
  - **Y (4 bits)**
  - **S (2 bits)**
- **Outputs:**
  - **G (4 bits)**

The arithmetic and logic units

- **Now we have two pieces of the puzzle:**
  - **An arithmetic unit that can compute eight functions on 4-bit inputs.**
  - **A logic unit that can perform four functions on 4-bit inputs.**
- **We can combine these together into a single circuit, an arithmetic-logic unit (ALU).**

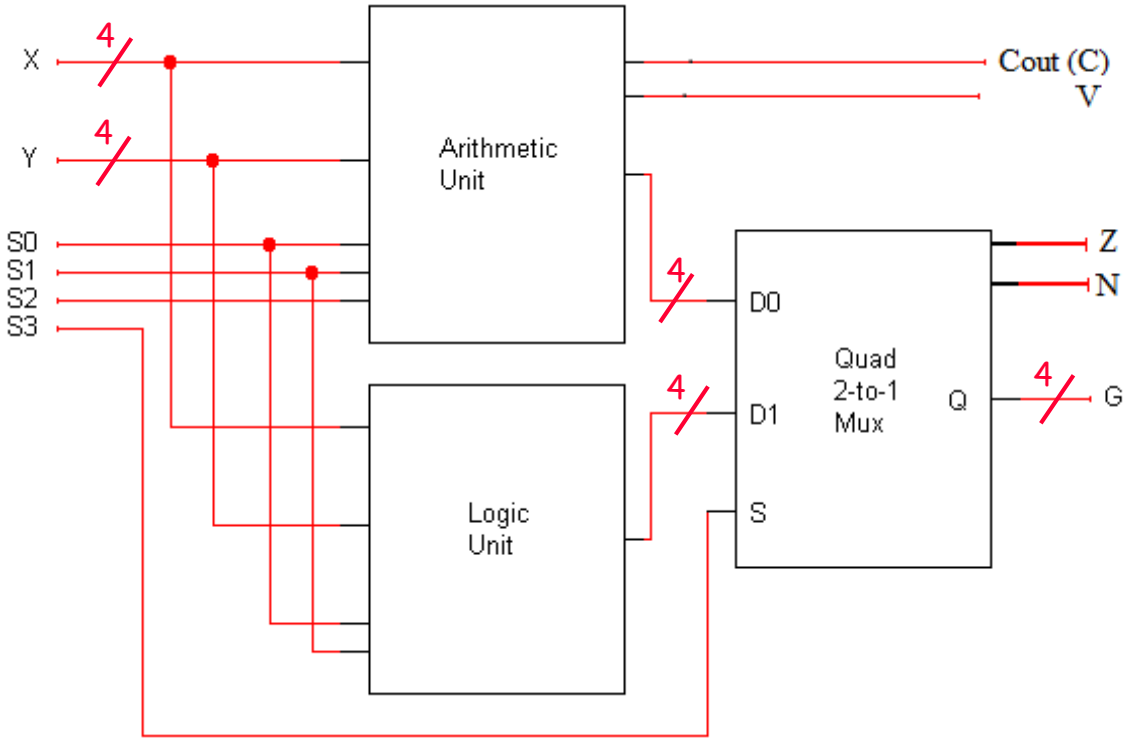


Our ALU function table

- **This table shows a sample function table for an ALU.**
- **All of the arithmetic operations have S_3=0, and all of the logical operations have S_3=1.**
- **These are the same functions we saw when we built our arithmetic and logic units a few minutes ago.**
- **Since our ALU only has 4 logical operations, we don't need S_2. The operation done by the logic unit depends only on S_1 and S_0.**

S_3	S_2	S_1	S_0	Operation
0	0	0	0	G = X
0	0	0	1	G = X + 1
0	0	1	0	G = X + Y
0	0	1	1	G = X + Y + 1
0	1	0	0	G = X + Y′
0	1	0	1	G = X + Y′ + 1
0	1	1	0	G = X – 1
0	1	1	1	G = X
1	x	0	0	G = X and Y
1	x	0	1	G = X or Y
1	x	1	0	G = X ⊕ Y
1	x	1	1	G = X′

A complete ALU circuit



G is the final ALU output.

- When S3 = 0, the final output comes from the arithmetic unit.
- When S3 = 1, the output comes from the logic unit.

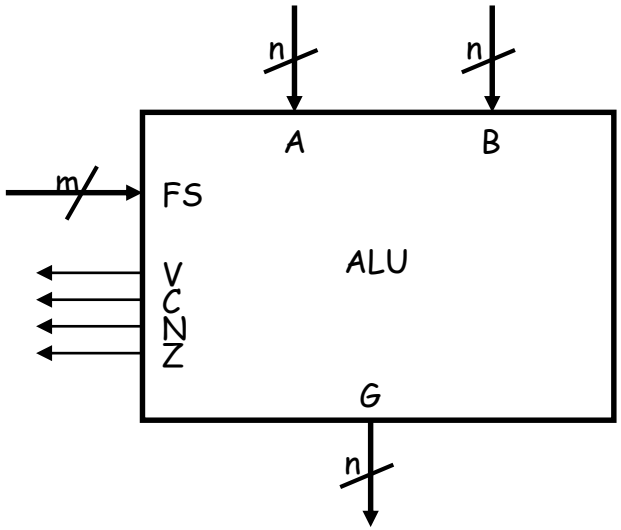
Status bits: Additional outputs

- C_{out} (C)
- Over-flow (V)
- Zero (Z)
- Negative (N)

- **The arithmetic and logic units share the select inputs S1 and S0, but only the arithmetic unit uses S2.**
- ***Both* the arithmetic unit and the logic unit are "active" and produce outputs.**
  - The mux determines whether the final result comes from the arithmetic or logic unit.
  - The output of the other one is effectively ignored.
- **Our hardware scheme may seem like wasted effort, but it's not really.**
  - "Deactivating" one or the other wouldn't save that much time.
  - We have to build hardware for both units anyway, so we might as well run them together.

The all-important ALU

- We'll use the following general block symbol for the ALU.
  - A and B are two n-bit numeric inputs.
  - FS is an m-bit function select code, which picks one of 2^m functions.
  - The n-bit result is called G.
  - Several status bits provide more information about the output G:
    - V = 1 in case of signed overflow.
    - C is the carry out.
    - N = 1 if the result is negative.
    - Z = 1 if the result is 0.

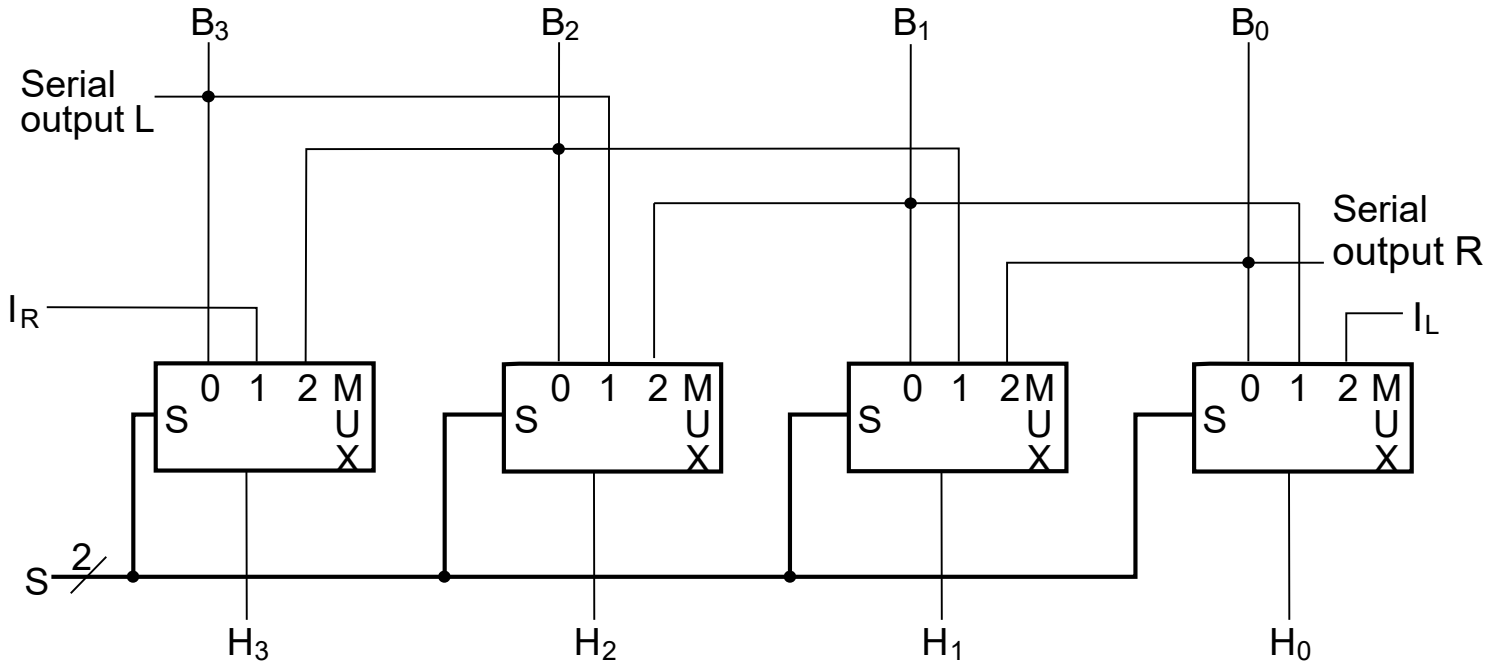


S_3	S_2	S_1	S_0	Operation
0	0	0	0	G = X
0	0	0	1	G = X + 1
0	0	1	0	G = X + Y
0	0	1	1	G = X + Y + 1
0	1	0	0	G = X + Y'
0	1	0	1	G = X + Y' + 1
0	1	1	0	G = X - 1
0	1	1	1	G = X
1	x	0	0	G = X and Y
1	x	0	1	G = X or Y
1	x	1	0	G = X ⊕ Y
1	x	1	1	G = X'

Combinational Shifter

- **Bidirectional shift register with parallel load**
  - Disadvantage: 3 clock pulses required
  - Ex: R1← sr R2
- **Combinational Shifter**
  - Transfer from a source to destination register
  - One clock cycle
- **Operations:**
  - Transfer
  - Shift Left,
  - Shift Right

4-Bit Basic Left/Right Shifter



- **Serial Inputs:**
  - I_R for right shift
  - I_L for left shift
- Logic Shift (**zero**) will be used
- Many options depending on instruction set

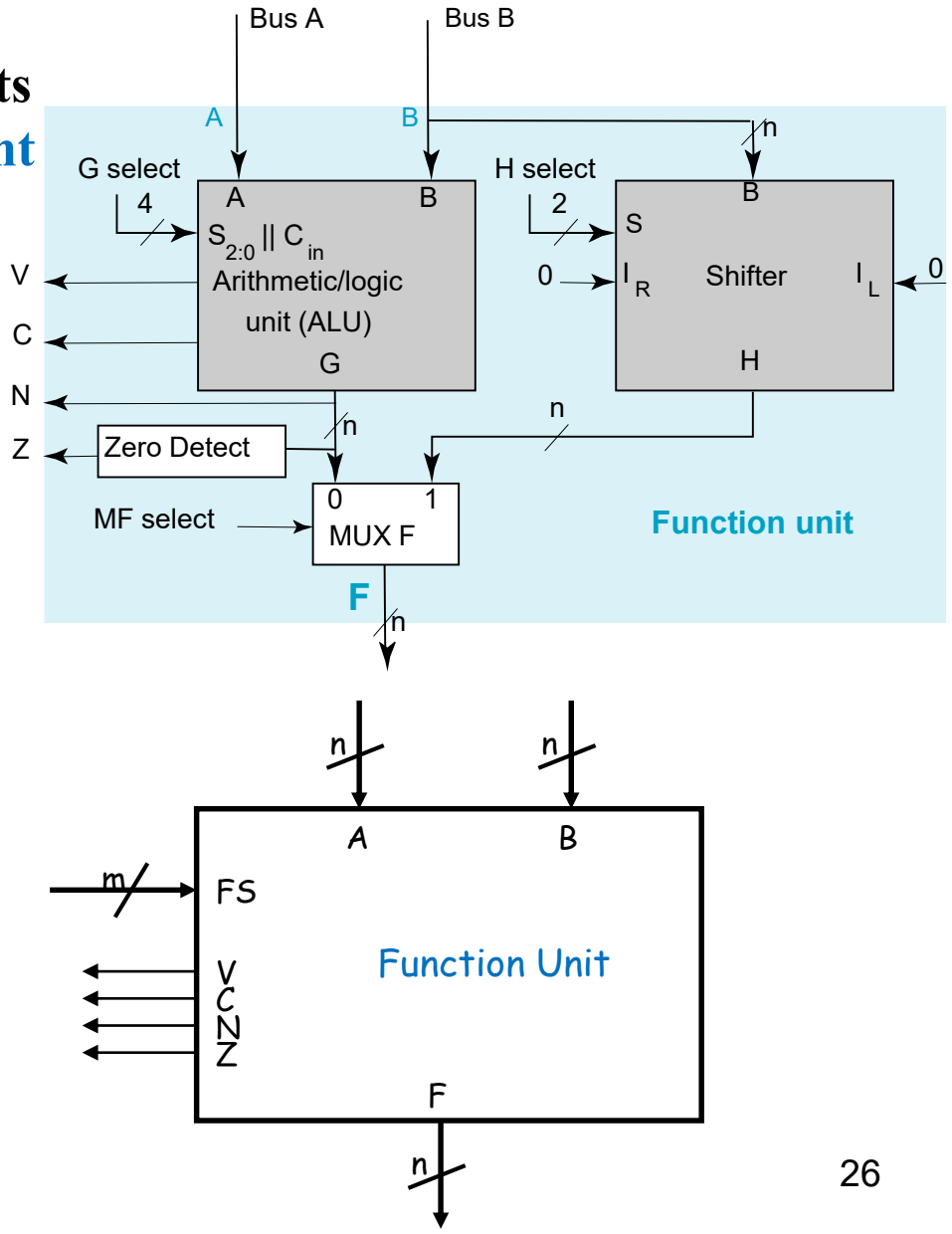
- **Shift Functions:**
  (S1, S0) = 00 Pass B unchanged
  01 Right shift
  10 Left shift
  11 Unused
- **Serial Outputs** (we will ignore)
  - R for right shift (Same as MSB input)
  - L for left shift (Same as LSB input)

Function Unit Design

Function Unit = ALU + Shifter

- **The function select code FS is 4 bits long, but there are only 15 different functions here.**

FS	Operation
0000	F = A
0001	F = A + 1
0010	F = A + B
0011	F = A + B + 1
0100	F = A + B'
0101	F = A + B' + 1
0110	F = A - 1
0111	F = A
1000	F = A ∧ B (AND)
1001	F = A ∨ B (OR)
1010	F = A ⊕ B (XOR)
1011	F = A'
1100	F = B
1101	F = sr B (shift right)
1110	F = sl B (shift left)



Definition of Function Unit Select (FS) Codes

FS(3:0)	MF Select	G Select(3:0)	H Select(3:0)	Micr ooperation
0000	0	0000	XX	$F \leftarrow A$
0001	0	0001	XX	$F \leftarrow A + 1$
0010	0	0010	XX	$F \leftarrow A + B$
0011	0	0011	XX	$F \leftarrow A + B + 1$
0100	0	0100	XX	$F \leftarrow A + \overline{B}$
0101	0	0101	XX	$F \leftarrow A + \overline{B} + 1$
0110	0	0110	XX	$F \leftarrow A - 1$
0111	0	0111	XX	$F \leftarrow A$
1000	0	1X00	XX	$F \leftarrow A \wedge B$
1001	0	1X01	XX	$F \leftarrow A \vee B$
1010	0	1X10	XX	$F \leftarrow A \oplus B$
1011	0	1X11	XX	$F \leftarrow \overline{A}$
1100	1	XXXX	00	$F \leftarrow B$
1101	1	XXXX	01	$F \leftarrow$ sr B
1110	1	XXXX	10	$F \leftarrow$ sl B

Boolean Equations:

$$MF = F_3 F_2$$

$$G_i = F_i$$

$$H_i = F_i$$