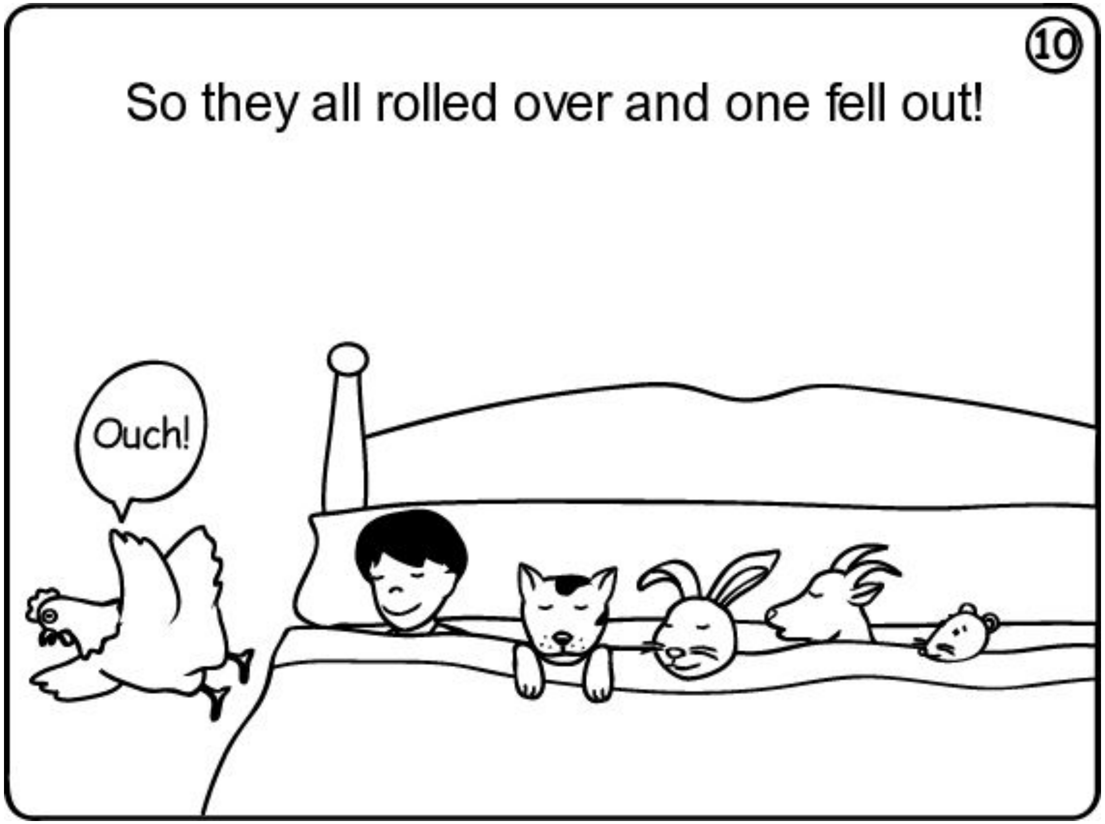
A problem with my gate Names

In the gate enumeration slide from two lectures ago I unfortunately gave bad names to two gates

INPUT AB	ZERO	AND	A>B	A	B>A	B	XOR	OR	NOR	XNOR	NOT 'B'	A>=B	NOT 'A'	B>=A	NAND	ONE
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

An Arithmetic Logic UNIT



- Shifts of shifts
- Boolean logic
- An ALU

Shifting Logic

Shifting is a common operation that is applied to groups of bits. Shifting is used for alignment, selecting parts of a word, as well as for arithmetic operations.

X << 1 is approx the same as 2*X

X >> 1 can be the same as X/2

For example:

$X = 00010100_2 = 20_{10}$

Left Shift:

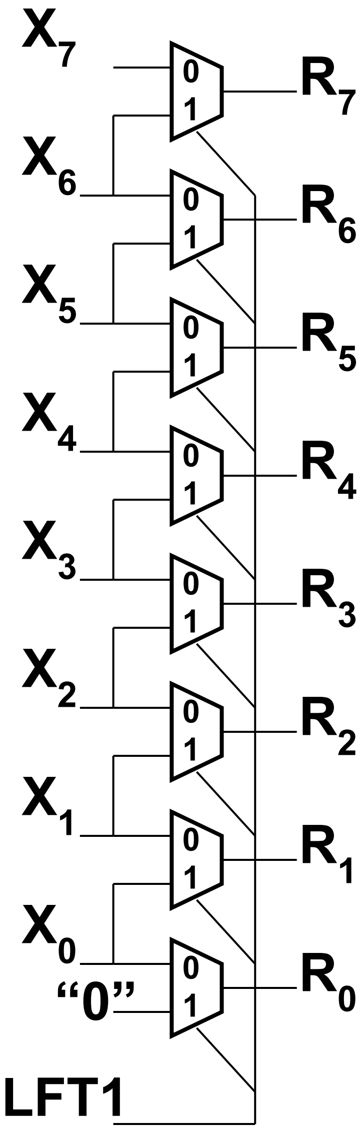
$(X << 1) = 00101000_2 = 40_{10}$

Right Shift:

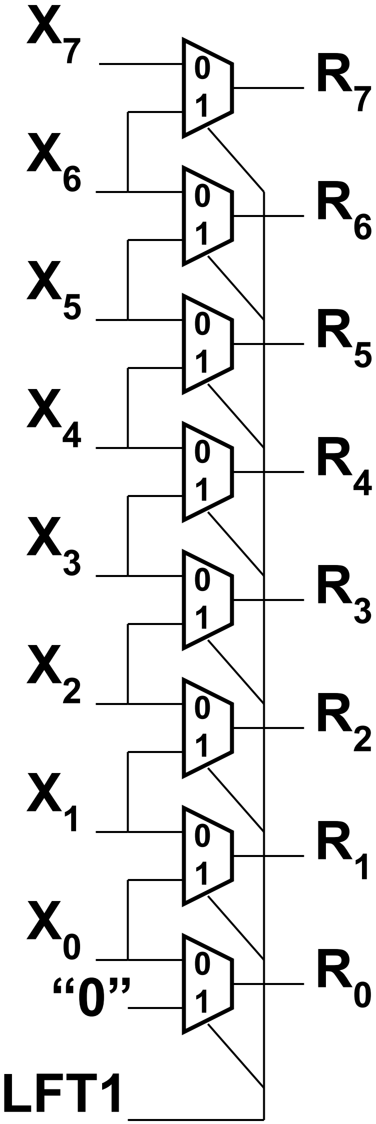
$(X >> 1) = 00001010_2 = 10_{10}$

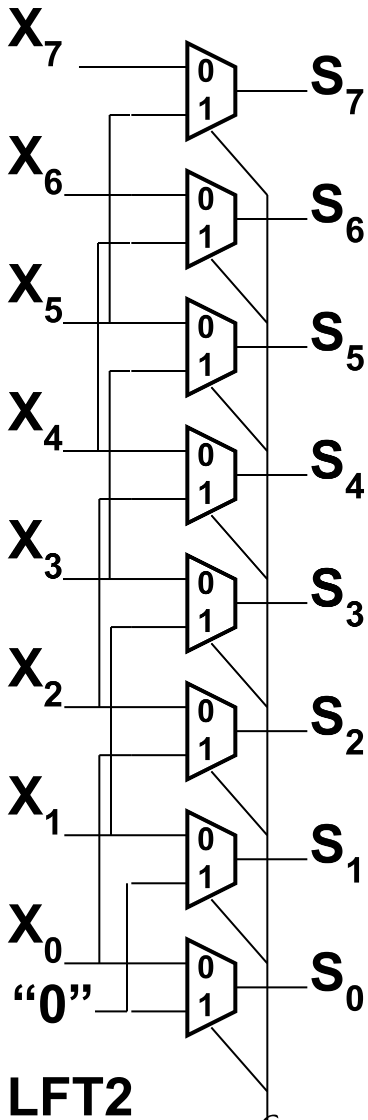
Signed or "Arithmetic" Right Shift:

$(-X >> 1) = (11101100_2 >> 1) = 11110110_2 = -10_{10}$



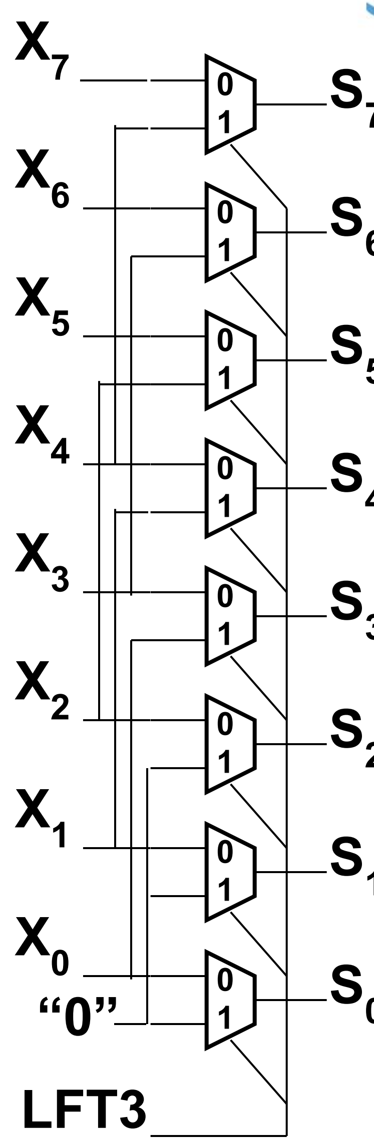
More Shifting



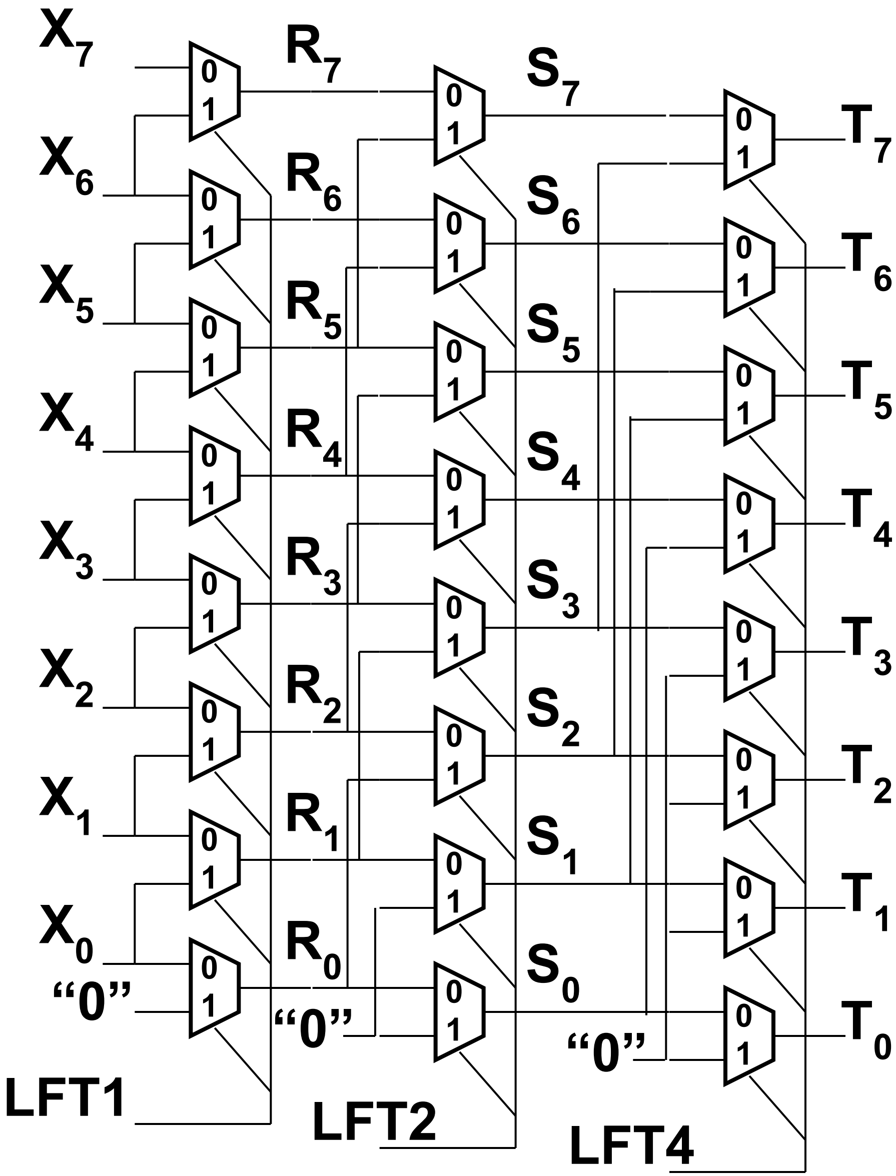


Using the same basic idea we can build left shifters of arbitrary shift amounts using muxes.

Each shift amount requires its own set of muxes.



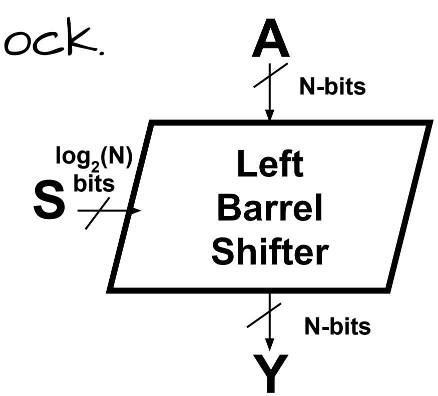
Barrel Shifting



If we connect our "shift-left-two" shifter to the output of our "shift-left-one" we can shift by 0, 1, 2, or 3 bits.

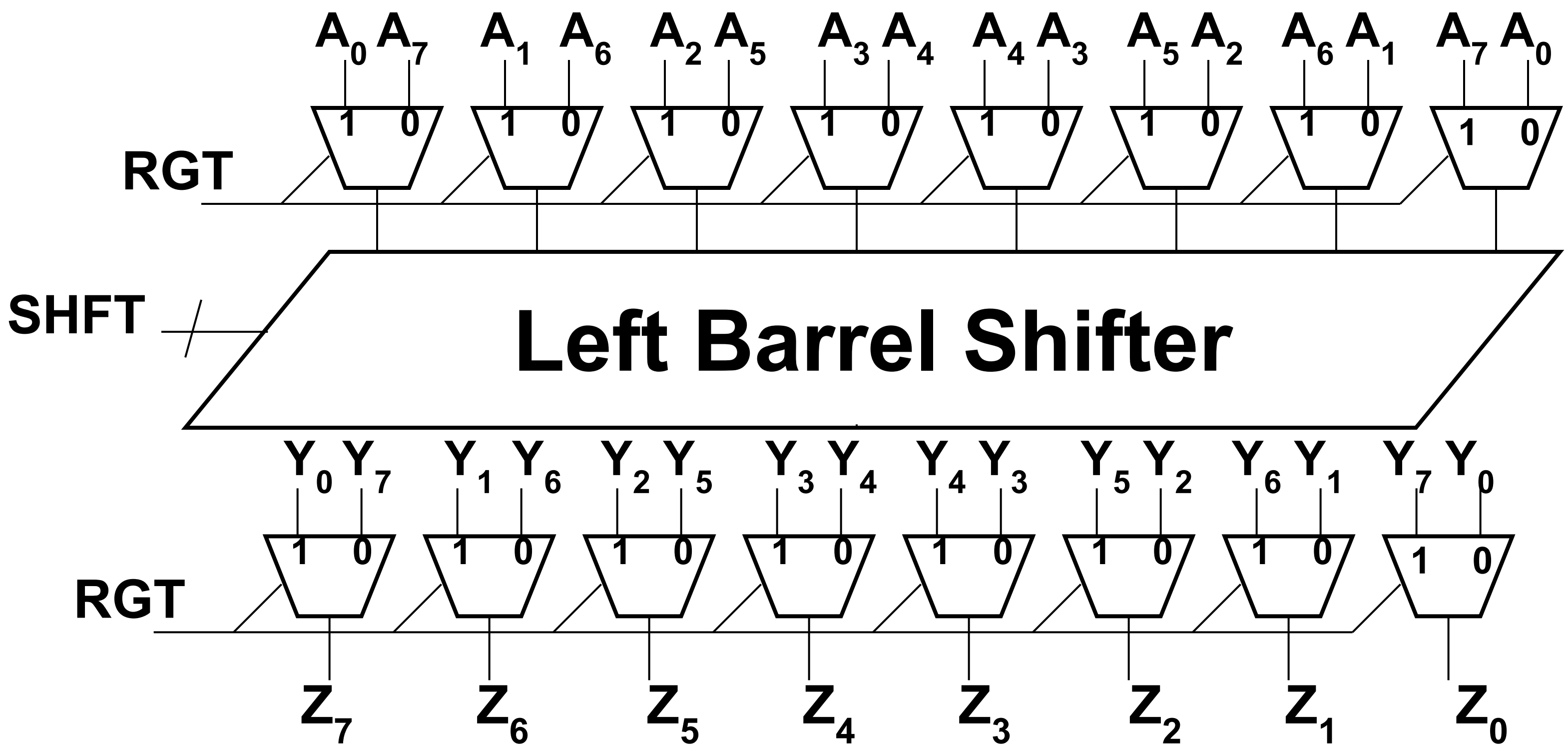
And, if we add one more "shift-left-4" shifter we can do any shift up to 7 bits!

So, let's put a box around it and call it a new functional block.

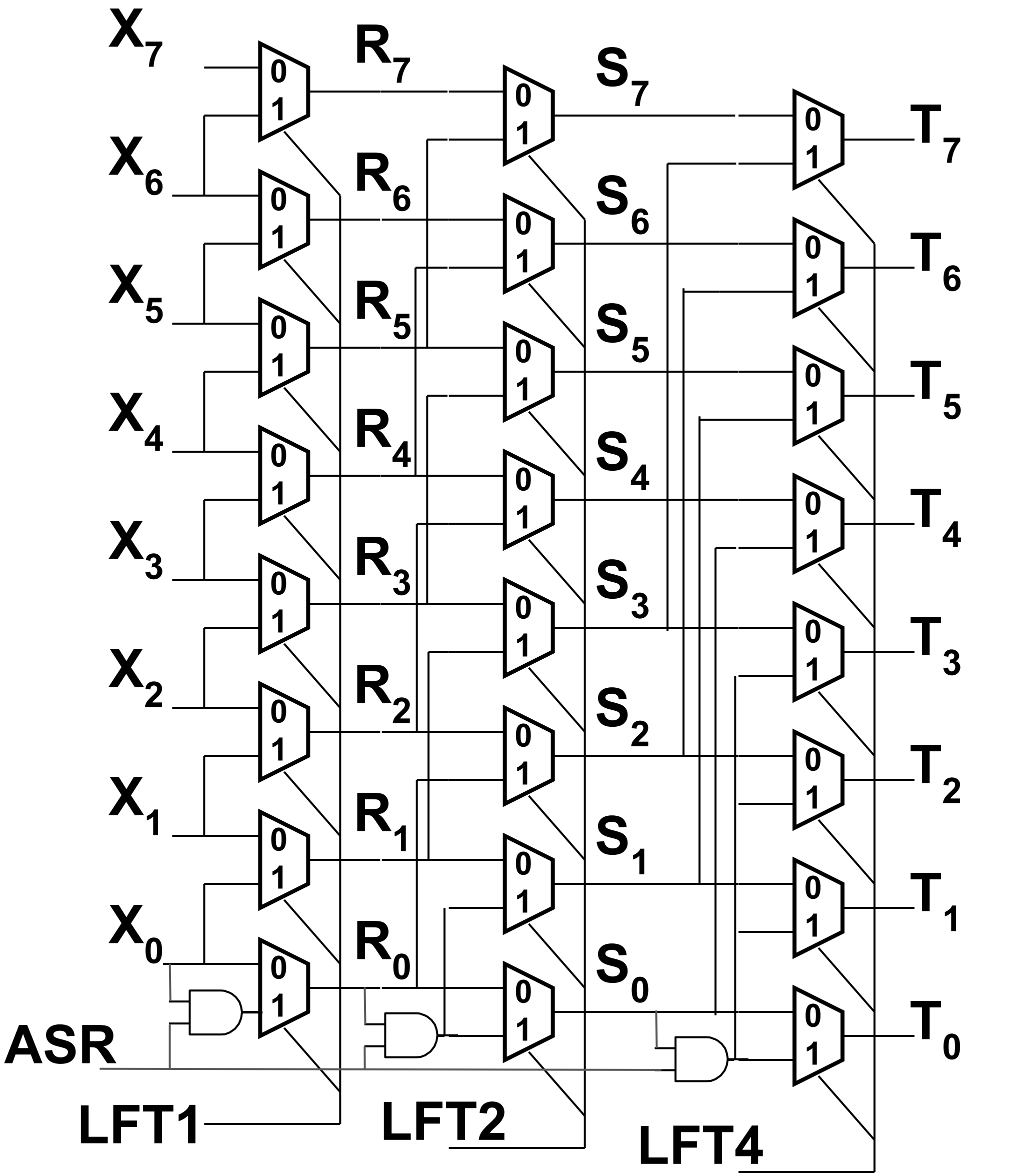


Adding a Twist

It would be straightforward to construct a "right barrel shifter" unit. However, a simple trick that enables a "left barrel shifter" to do both.



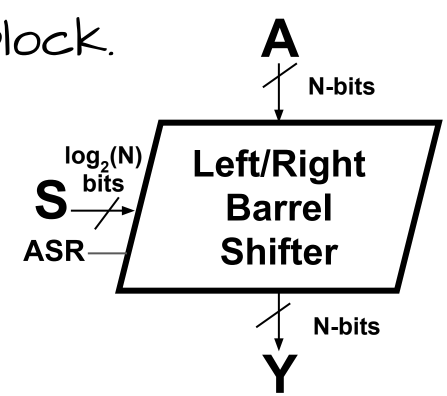
One LAST Detail



If we connect our "shift-left-two" shifter to the output of our "shift-left-one" we can shift by 0, 1, 2, or 3 bits.

And, if we add one more "shift-left-4" shifter we can do any shift up to 7 bits!

So, let's put a box around it and call it a new functional block.



Bitwise Logical Operations

We need to perform logical operations, or **Booleans**, on groups of bits. Which ones?

ANDing is used for "masking" off groups of bits.
ex. 10101110 & 00001111 = 00001110 (mask selects last 4 bits)

ORing is used for "setting" groups of bits.
ex. 10101110 | 00001111 = 10101111 (1's set last 4 bits)

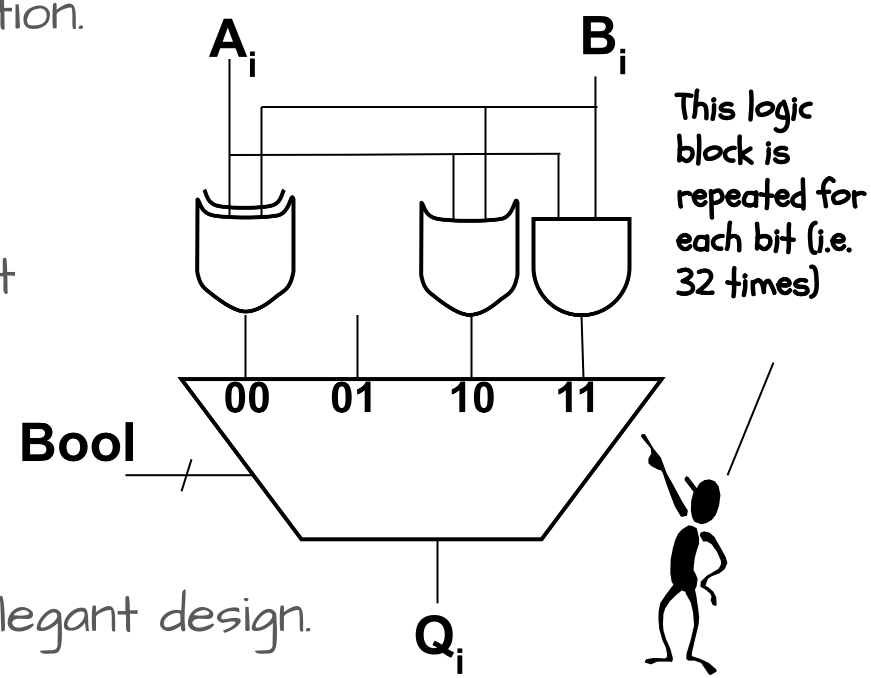
EORing is used for "complementing" groups of bits.
ex. 10101110 ^ 00001111 = 10100001 (complement last 4 bits)

Boolean Unit (The obvious way)

It is simple to build up a Boolean unit using primitive gates and a mux to select the function.

Since there is no interconnection between bits, this unit can be simply replicated at each position. The cost is about 6 gates per bit. One for each primitive function, and approx 3 for the 4-input mux.

This is a straightforward, but not elegant design.



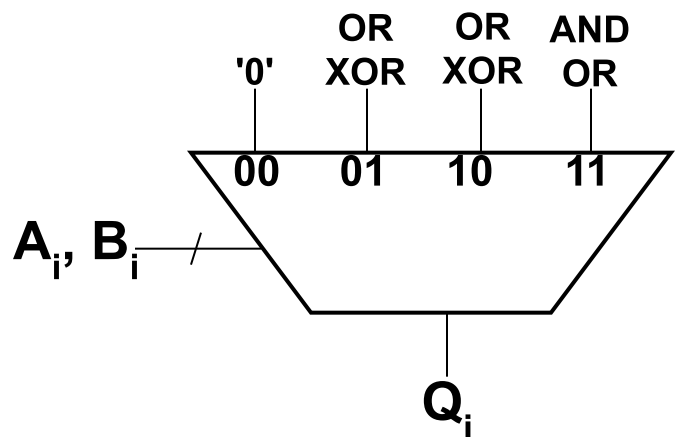
Cooler Bools

We can better leverage a MUX's capabilities in our Boolean unit design, by connecting the bits to the select lines.

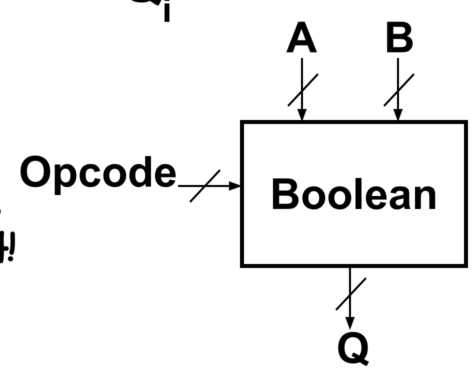
Why is this better?

While it might take a little logic to decode the truth table inputs, you only have to do it once, independent of the number of bits.

BTW, it also handles the MOV and MVN cases.



Which ever way makes the most sense to you. Let's get a box around it!



AN ALU, AT LAST

We give the "Math Center" of a computer a special name--the Arithmetic Logic Unit (ALU). For us, it just a big box of gates! Well need to decode a few control lines, Sub, Math, Shift, b00,b01,b10,b11 from the instruction.