MSM-8
8-Bit Arithmetic Logic Unit

MSM
TECHNOLOGY

"Experience what's outside!"

Created and Designed by
Gregory **Magallanes**, Kefin **Sajan**, and Bryan **Morfe**

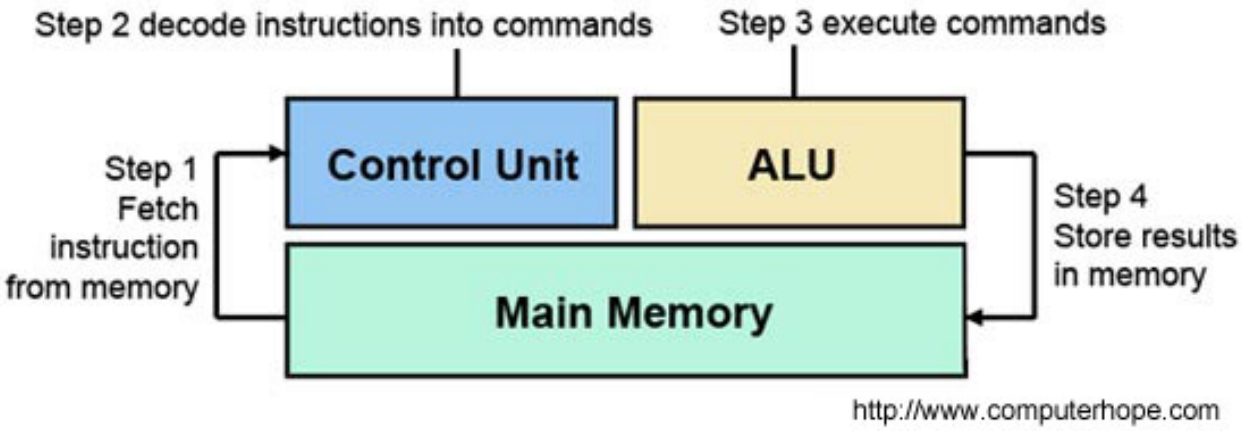
Final Project
CS 3410 FALL 2018

This page was left blank intentionally

What is an Arithmetic logic unit?

a) An Arithmetic logic unit (ALU) is a digit circuit inside a CPU (Central Processing Unit) with the objective of perform all mathematical computation for the CPU. The ALU is placed inside the CPU to get data from registers and to store in the output register.

b) The first concept was introduced by John Von Neumann in 1945, and the idea was implemented in 1948 as a single-bit ALU.



More information about ALU: https://en.wikipedia.org/wiki/Arithmetic_logic_unit

Our Mission

Our target was to create an ALU (Really an AU) that could add, subtract and multiply 8-bit 2's complement binary numbers.

Our Design

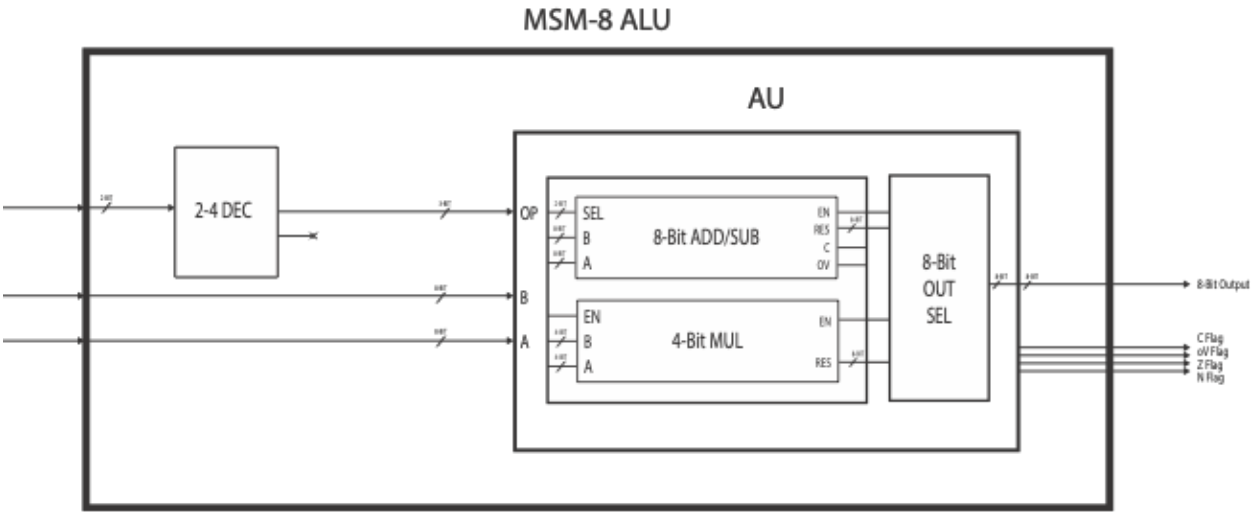
Our Arithmetic Unit had to be capable of adding/subtracting and at the same time multiply–which adds a parallelism aspect to it. At the same time, we just wanted one output (8-bit 2's complement). This meant that we had to implement a way to choose the operation our AU was to perform–decoding–and as such, we had to find a way to choose one output based on an operation code. Our choice for operation selection was a decoder. Since we had three possible operations, a 2-bit input operation code could be used along with a decoder that would act as an enable for each operation was used. A 2-bit to 4-bit decoder would be perfect where one operation code would be left unused.

For output, the obvious choice was a multiplexer. We decided to breakdown the multiplexer and implement a slightly modified version of it that would make the design of our AU slightly more intuitive and efficient.

Our AU also has four flags that include a Carry bit flag (C), and Overflow flag (oV), a Zero flag (Z), and a Negative flag (N), that indicate whether the computation resulted in a carry bit, overflow, was zero, and/or was negative, respectively.

Block Diagram

Below is a block diagram of our design.



Operations

As shown in the Block Diagram, our system contains four major components:

A. 2-4 Decoder with the following operation codes:

i) 00 - Left unused
ii) 01 - Multiplication
iii) 10 - Subtraction
iv) 11 - Addition

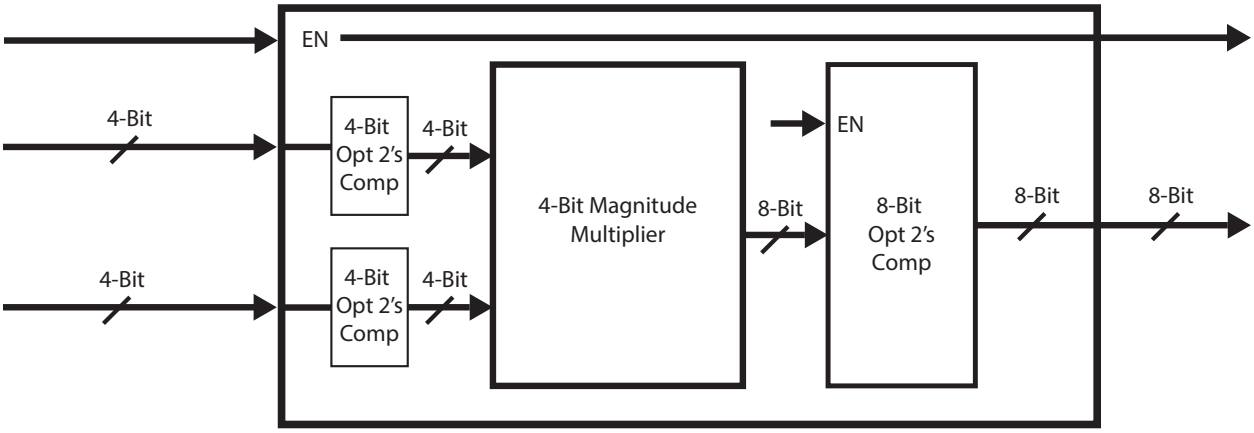
B. 2's Complement 8-Bit Adder/Subtractor

Our design uses the same circuit for addition and subtraction. We simply take the 2's complement of the B operand if subtraction is selected, and add A to the 2's complement of B. This results is the operation A – B.

C. 2's Complement 4-Bit Multiplier

This is perhaps the most complicated part of our design, yet simple and intuitive.

Let's take a look at our multiplication block diagram:



As can be observed in our design, the inputs of our multiplication circuit is 4-bit as opposed to 8-bit. However, the inputs for our AU are really 8 bits, so to get over this we simply took the *least significant four bits* of each operand as the input of our multiplication circuit.

Additionally, we have an enable bit which corresponds to the second least significant of our decoder (the one that would be 1 if the operation code is 01). This enable bit helps in the output section aspect of our circuit, and as such is irrelevant in our multiplication circuit. It is shown here purely for illustration purposes.

Before we go into details of our design, let's illustrate how our multiplier works in pseudocode:

```
A = 4 bit input
B = 4 bit input
if A is negative
      A = A * -1 // make A positive if was negative
      A_neg = TRUE
if B is negative
      B = B * -1 // make B positive if was negative
      B_neg = TRUE
C = A * B
```

```
if B_neg == TRUE AND A_neg == FALSE OR
   A_neg == TRUE AND B_neg == FALSE
      C = C * -1
```

As can be seen in the pseudocode, we only multiply positive numbers, and make the result negative if the signs of A and B were different. We accomplish this with optional two's complements and magnitude multipliers.

Next, are our two 4-Bit Optional 2's Complement Taker. That's how we call them. Basically, we only take the two's complement if the number is negative–the most significant bit is 1. That bit serves an enable bit for our two's complement taker, making it optional, and only enabling it if the number is negative. To take the two's complement based on that bit we first need to invert every bit in its 4-bit input by XORing *only the magnitude* bits. We can safely take the sign bit out of the equation and only use it as the enable bit. The reason why this works is because if the sign bit is 1, then we have to invert it, and it becomes 0. If the sign bit is 0, we don't invert it, and it stays 0, so if we take the two's complement or not does not change the fact that the sign bit should become 0. Finally, the second step to taking the two's complement is to add 1, which means that we can simply add the sign bit; if the sign bit is 1, we add 1, otherwise we add 0, which works perfectly for our purposes.

Technically, the output of our 4-bit Optional 2's Complement Taker is 4-bits, however, there is only one case where that is true. In all other cases, the output is only 3 bits. To illustrate what we mean let's take as input any 4-bit number and pass it through the machine:

0110 => 0110	Sign bit is zero, so the machine does nothing
1011 => 0100 => 0101	Fourth bit is zero, and can be ignored.
1001 => 0110 => 0111	Fourth bit is zero, and can be ignored.

The one case where that doesn't happen is if the input is -8d or 1000b:

1000 => 0111 => 1000

The only case where the fourth bit is 1 is the one above. We can take advantage of this by making a much simpler 3-bit magnitude multiplier circuit and making a special case for the input -8d. This will make our design much more efficient as

we are able to ignore over 170 cases that will never happen by taking them completely off the equation. We will talk about that case and how we solved it later.

For now, let's ignore the case where the output of the two's complement taker is 1, and consider the others where it is 0. If the fourth bit is always zero, we can just implement a 3-bit multiplier. Inside the 4-bit magnitude multiplier there really is a 3-bit magnitude multiplier and a circuit that takes care of the -8d case. The 3-bit magnitude multiplier was implemented with a truth table and 6-bit input variable, 6-bit output variable K-Map that is illustrated in the next section. This resulted in many and and or gates for each bit of the output.

Now, how did we take care of the case -8d (or 1000b)? We thought about it in a mathematical way:

$$-8x = -8x; \qquad \textit{is equivalent to} \qquad -7x - x = -8x;$$

Now, we have to remember that our multiplier circuit only multiplies *magnitudes*, therefore the equation looks more like:

$$8x = 8x; \qquad \textit{is equivalent to} \qquad 7x + x = 8x;$$

Additionally, we have to take care of an additional case. If the inputs are both 8d, the above equation won't work:

$$8 * 8 = 8 * 8; \qquad \textit{is equivalent to} \qquad 7 * 7 + 7 + 7 + 1 = 8 * 8$$

In the above equation we see a slight change, but the goal is to have numbers that can be represented in 3 bits. So we can see a common thing in both equations; whenever we see an 8d, we have to subtract 1, or making it 7. Also, at the end we always add the other operand to the final result. The only difference in the case where both numbers is 8d is that we must add 1 at the end. This happens because we are subtracting 1 to both operands. This makes our 3-bit magnitude multiplier be able to multiply 8d as well, something that would normally require a 4-bit magnitude multiplier. Here is the pseudocode for the more complete multiplier:

```
A = 4 bit input

B = 4 bit input
if A is negative
```

```
      A = A * -1 // make A positive if was negative
      A_neg = TRUE
if B is negative
      B = B * -1 // make B positive if was negative
      B_neg = TRUE

if A is 8
      A = 7
      A_was_eight = TRUE

if B is 8
      B = 7
      B_was_eight = TRUE

C = A * B

if A_was_eight
      C = C + B

if B_was_eight
      C = C + A

if A_was_eight AND B_was_eight
      C = C + 1

if B_neg == TRUE AND A_neg == FALSE OR
   A_neg == TRUE AND B_neg == FALSE
      C = C * -1
```

If the circuit follows the above pseudocode, it can multiply *any* 4-bit two's complement number. Finally, the 8-bit Optional 2's Complement Taker uses the same concept as the 8-bit Optional 2's Complement Taker, but uses as the enable bit is a 2-bit XOR gate of the sign bit of both operands. The output of this gate is 1 if the inputs are different, so it will take the two's complement of the result of the inputs have different signs. This works because the result of our 4-bit magnitude multiplier is at most 7-bits (8 * 8), so the output never causes Carry, Overflow, or a Negative result, so you can safely take the two's complement if the output is *supposed* to be negative. This produces our final result.

D. 8-bit Output Selector

Our circuit always performs addition/subtraction and multiplication on every operation. However, the Output Selector selects the appropriate result based on the Operation Code.

The way it works is simple and can be illustrated with the pseudocode below:

```
add_sub_result = add_sub_result * enable_bit_add_sub
mul_result = mul_result * enable_bit_mul
final_result = add_sub_result + mul_result
```

As we can see in the code above, if the addition OR subtraction operation was selected, then its enable bit will be 1, otherwise it will be 0. The same is applicable to the multiplication. If enable bit is 0, the whatever the result was, it will become 0, and adding it to anything else will be that anything else. In binary, you can just AND each bit of the output with its enable bit. If the enable bit is 1, then the result is unaffected, otherwise the result is 0. To choose between add/sub and mul, we just OR both outputs, and the was that is not 0 (the one whose enable bit was not 0) will pass through, and that result will be selected.

Truth Tables & K-Maps

3-Bit Magnitude Multiplier

n	A_2	A_1	A_0	B_2	B_1	B_0	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	0	0	0	0
2	0	0	0	0	1	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0	0	0	0
5	0	0	0	1	0	1	0	0	0	0	0	0
6	0	0	0	1	1	0	0	0	0	0	0	0
7	0	0	0	1	1	1	0	0	0	0	0	0
8	0	0	1	0	0	0	0	0	0	0	0	0
9	0	0	1	0	0	1	0	0	0	0	0	1
10	0	0	1	0	1	0	0	0	0	0	1	0
11	0	0	1	0	1	1	0	0	0	0	1	1
12	0	0	1	1	0	0	0	0	0	1	0	0
13	0	0	1	1	0	1	0	0	0	1	0	1
14	0	0	1	1	1	0	0	0	0	1	1	0
15	0	0	1	1	1	1	0	0	0	1	1	1
16	0	1	0	0	0	0	0	0	0	0	0	0
17	0	1	0	0	0	1	0	0	0	0	1	0
18	0	1	0	0	1	0	0	0	0	1	0	0
19	0	1	0	0	1	1	0	0	0	1	1	0
20	0	1	0	1	0	0	0	0	1	0	0	0
21	0	1	0	1	0	1	0	0	1	0	1	0
22	0	1	0	1	1	0	0	0	1	1	0	0
23	0	1	0	1	1	1	0	0	1	1	1	0
24	0	1	1	0	0	0	0	0	0	0	0	0
25	0	1	1	0	0	1	0	0	0	0	1	1
26	0	1	1	0	1	0	0	0	0	1	1	0
27	0	1	1	0	1	1	0	0	1	0	0	1

n	A_2	A_1	A_0	B_2	B_1	B_0	O_5	O_4	O_3	O_2	O_1	O_0
28	0	1	1	1	0	0	0	0	1	1	0	0
29	0	1	1	1	0	1	0	0	1	1	1	1
30	0	1	1	1	1	0	0	1	0	0	1	0
31	0	1	1	1	1	1	0	1	0	1	0	1
32	1	0	0	0	0	0	0	0	0	0	0	0
33	1	0	0	0	0	1	0	0	0	1	0	0
34	1	0	0	0	1	0	0	0	1	0	0	0
35	1	0	0	0	1	1	0	0	1	1	0	0
36	1	0	0	1	0	0	0	1	0	0	0	0
37	1	0	0	1	0	1	0	1	0	1	0	0
38	1	0	0	1	1	0	0	1	1	0	0	0
39	1	0	0	1	1	1	0	1	1	1	0	0
40	1	0	1	0	0	0	0	0	0	0	0	0
41	1	0	1	0	0	1	0	0	0	1	0	1
42	1	0	1	0	1	0	0	0	1	0	1	0
43	1	0	1	0	1	1	0	0	1	1	1	1
44	1	0	1	1	0	0	0	1	0	1	0	0
45	1	0	1	1	0	1	0	1	1	0	0	1
46	1	0	1	1	1	0	0	1	1	1	1	0
47	1	0	1	1	1	1	1	0	0	0	1	1
48	1	1	0	0	0	0	0	0	0	0	0	0
49	1	1	0	0	0	1	0	0	0	1	1	0
50	1	1	0	0	1	0	0	0	1	1	0	0
51	1	1	0	0	1	1	0	1	0	0	1	0
52	1	1	0	1	0	0	0	1	1	0	0	0
53	1	1	0	1	0	1	0	1	1	1	1	0
54	1	1	0	1	1	0	1	0	0	1	0	0
55	1	1	0	1	1	1	1	0	1	0	1	0
56	1	1	1	0	0	0	0	0	0	0	0	0
57	1	1	1	0	0	1	0	0	0	1	1	1

n	A_2	A_1	A_0	B_2	B_1	B_0	O_5	O_4	O_3	O_2	O_1	O_0
58	1	1	1	0	1	0	0	0	1	1	1	0
59	1	1	1	0	1	1	0	1	0	1	0	1
60	1	1	1	1	0	0	0	1	1	1	0	0
61	1	1	1	1	0	1	1	0	0	0	1	1
62	1	1	1	1	1	0	1	0	1	0	1	0
63	1	1	1	1	1	1	1	1	0	0	0	1

O_5

$A_2A_1A_0$ \ $B_2B_1B_0$	000	001	011	010	100	101	111	110
000								
001								
011								
010								
100								
101							47	
111						61	63	62
110							55	54

O_4

$A_2A_1A_0$ \ $B_2B_1B_0$	000	001	011	010	100	101	111	110
000								
001								
011							31	30
010								
100					36	37	39	38
101					44	45		46
111			59		60		63	
110			51		52	53		

O_3

$A_2A_1A_0$ \ $B_2B_1B_0$	000	001	011	010	100	101	111	110
000								
001								
011			27		28	29		
010					20	21	23	22
100			35	34			39	38
101			43	42		45		46
111				58	60			62
110				50	52	53	55	

O_2

$A_2A_1A_0$ \ $B_2B_1B_0$	000	001	011	010	100	101	111	110
000								
001					12	13	15	14
011				26	28	29	31	
010			19	18			23	22
100		33	35			37	39	
101		41	43		44			46
111		57	59	58	60			
110		49		50		53		54

O_1

$A_2A_1A_0$ \ $B_2B_1B_0$	000	001	011	010	100	101	111	110
000								
001			11	10			15	14
011		25		26		29	31	30
010		17	19			21	23	
100								
101			43	42			47	46
111				58		61		62
110		49	51			53	55	

O_0

$A_2A_1A_0$ \ $B_2B_1B_0$	000	001	011	010	100	101	111	110
000								
001		9	11			13	15	
011		25	27			29		
010								
100								
101		41	43			45	47	
111		57	59			61	63	
110								

$O_5 = A_2A_1B_2B_1 + A_2A_0B_2B_1B_0 + A_2A_1A_0B_2B_0$

$O_4 = A_2A_1'A_0'B_2 + A_2A_1'B_2B_1' + A_2A_0'B_2B_1' + A_2A_1'B_2B_0' + A_2B_2B_1'B_0' + A_2'A_1A_0B_2B_1 + A_1A_0B_2B_1B_0 + A_2A_1B_2'B_1B_0$

$O_3 = A_2'A_1A_0'B_2 + A_2'A_1B_2B_1' + A_1B_2B_1'B_0' + A_1A_0'B_2B_0 + A_2A_1'A_0'B_1 + A_2A_1'B_2'B_1 + A_2B_2'B_1B_0' + A_2A_0B_1B_0' + A_2'A_1A_0B_2'B_1B_0 + A_2A_1'A_0B_2B_1'B_0$

$O_2 = A_1'A_0B_2B_0' + A_0B_2B_1'B_0' + A_2'A_0B_2B_0 + A_2'A_1A_0'B_1 + A_1A_0'B_1B_0' + A_1B_2'B_1B_0' + A_2A_1'A_0'B_0 + A_2A_0'B_1'B_0 + A_2A_0B_2'B_0$

$O_1 = A_1'A_0B_1 + A_0B_1B_0' + A_1B_1'B_0 + A_2'A_1A_0'B_0 + A_1A_0'B_2'B_0$

$O_0 = A_0B_0$

Additional

For additional details, refer to the CCT file that contains the circuit implementation.