Building a 6-bit Microprocessor

Assignment no.2

Group of 3 Students

EE-222 Microprocessor Systems

1 Administrivia

1.1 Learning Objectives

This assignment will enable you to,

- Understand the basic principle of computing.
- Understand how computer processors are built with logic elements.
- Understand bare metal programming stack.
- Link Logic Design Concepts with this course.

1.2 Deliverables and Timeline

You are required to deliver and demonstrate a working hardware model of this assignment by April 1, 2022, at 4pm. You are also required to submit a one-page report on LMS showing the snapshot of your final design and the features implemented. A thorough understanding of the assignment is necessary which will help you in oral viva.

1.3 Marks Distribution

The following marks distribution is tentative and is subjected to change without any prior notice to the students.

Hardware Demo	Viva	Total Marks
20	10	30

2 Introduction

This assignment will introduce a design of a simple 4-bit number crunching machine (which you may also call as 4-bit Microprocessor) and a walk- through of its programming. **You must transform this design into aworking hardware model with the help of logic ICs for 6-bit processor**. The design is divided into three parts Arithmetic Logic Unit (ALU), Data Registers and Control Logic.

3 Components (Logic ICs) Used

- 4-bit D-type Registers
- 4-bit Binary Full Adder
- 4-bit Binary Counter
- Quad 2-Data Selectors
- 2-Line to 4-Line Decoder
- AND, OR, NOT and XOR Gates
- Parallel Address, Parallel 8-bit I/O EEPROM

4 Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is the most basic and important part of any computing machine. All the arithmetic or logical operations are performed through it. The ALU used in this assignment will be able to perform binary addition as well as 2's complement binary subtraction.

4.1 Combinational Logic

The circuit requirement is to add or subtract two 4-bit numbers and generate a carry. In order to choose between add and subtract operations, we will be using a selection bit. The Boolean equation for such a circuit can be realized as,

$$Out = (A + B)S' \ OR \ (A - B)S$$

Here "A" and "B" are the two inputs, "Out" is the output and "S", the selection bit.
Subtraction will be carried out by inverting the bits (i.e. taking 1's complement) of B and raising the "carry in" of adder to logic 1, in order to add an extra bit which will eventually generate 2's complement of B.

$$A - B = A + (1's\ complement\ of\ B) + 1(carry\ in)$$
$$A - B = A + (2's\ complement\ of\ B)$$

For inverting the bits we will use XOR gate with one input tied to the selection bit S, and the other to the input bit, such that when selection bit goes 1, the property of XOR, $B \oplus 1 = B'$ can be used and when it is 0 the input passes unaffected $B \oplus 0 = B$.

4.2 Circuit Assembly

The block diagram of ALU is shown in Figure 1.

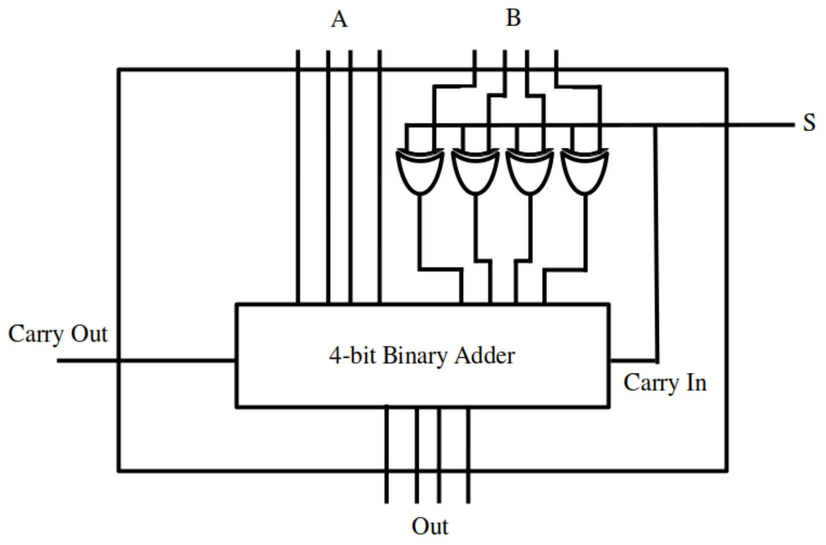


Figure 1: Arithmetic Logic Unit (ALU)

Selection bit (S)	**Function**
0	*Out = A +B*
1	*Out = A −B*

5 Data Registers

At least two 4-bit registers are required to hold the data for ALU. The output of these two registers, say R_A and R_B, are directly connected to the two inputs of the ALU, *A* and *B* respectively. The output of these registers is always enabled i.e., they are always channeling data into the ALU. However, the input to these registers is controlled and data can only enter into them when the input enable bit of R_A and R_B is at logic 1.

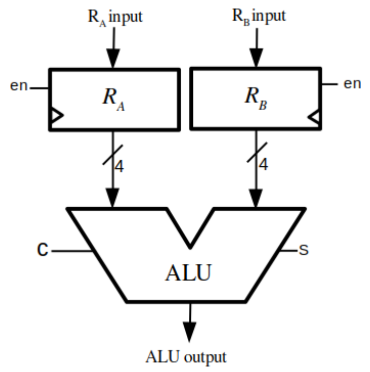


Figure 2: Register-ALU Assembly

5.1 Functionality and Time Synchronization

To start the computation, we need to load some initial values to R_A and R_B. Moreover, we would also like to utilize these registers to save the output of ALU too. In order to achieve this dual functionality, we need to add a 4-bit 1-out-of-2 data MUX before the input of registers. One of the MUX inputs would be connected to ALU's output and the other one to custom input. The selection bit of this MUX, say S_{reg} must be 0 to let the ALU's data pass through and 1 for custom value. The updated datapath is shown in Figure 3.

A small D flip-flop is also placed adjacent to ALU to store the carry bit into it on positive edge of clock so that we can examine the status of arithmetic operation performed by ALU.

Here the concept of clock is important. We know that registers are made of flip-flops that only store data on the positive edge of clock (assumingthe enable signal is 1, otherwise clock edges are in-effective). The small triangle on R_A and R_B in Figure 3 symbolizes input clock.

Let's see an example. Suppose through custom input (in past), 2 was stored in R_A and 3 in R_B. Now, enable of R_A is 1, enable of R_B is 0, selection bit (S) of ALU is 0 and S_{reg} is also 0. At the output of ALU, the sum 5 is present. As soon as the positive edge of clock comes, 5 got stored in R_A (as its enable pin was high) and appears at the output of R_A, the output of ALU becomes 8. But due to "internal gate delays", 8 appears a few nano-seconds after the positive edge had passed and now it cannot enter any register until next positive edge arrives. We can turn down the enable pins of registers to zero, to make clock edges in-effective so that the incoming 8 may not replace previously stored 5.

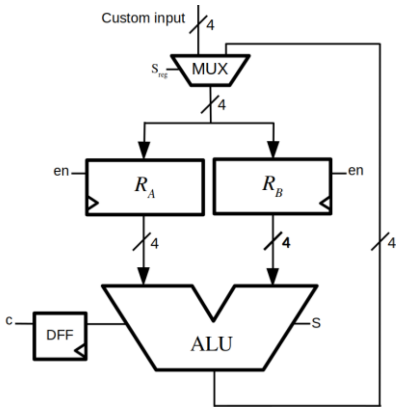


Figure 3: Modified Register-ALU Assembly

6 Testing

Let's test our design with a small "swapping" example. Usually when we swap the values of two variables, we use another temporary variable to hold data for some time,

```
Initial state: $x = a, y = b$
{
  $temp = x$
  $x = y$
  $y = temp$
}
Final state: $x = b, y = a$
```

However, there is another smart algorithm to swap data of two registers without using any third register,

```
Initial state: $x = a, y = b$
{
  $x = x + y$
  $y = x - y$
  $x = x - y$
}
Final State: $x = b, y = a$
```

There are four control signals (S_{reg}, en_{R_A}, en_{R_B} *and* S) and the 4-bit custom input in our hands. First, we shall load "a" to R_A and "b" to R_B, after which we will execute the algorithm. Here "a" and "b" are the 4-bit custom inputs.

Function	**Control(S_{reg} en_{R_A} en_{R_B} S)**	**Result(at posedge clk)**
Load a to R_A	110x	a stored in R_A
Load b to R_B	101x	b stored in R_B
$R_A = R_A + R_B$	0100	a+b stored in R_A
$R_B = R_A - R_B$	0011	a stored in R_B
$R_A = R_A - R_B$	0101	b stored in R_A

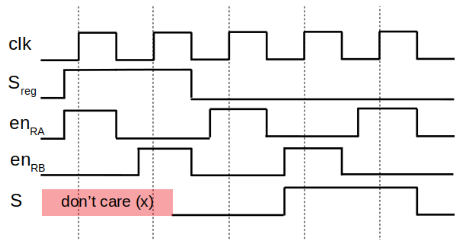
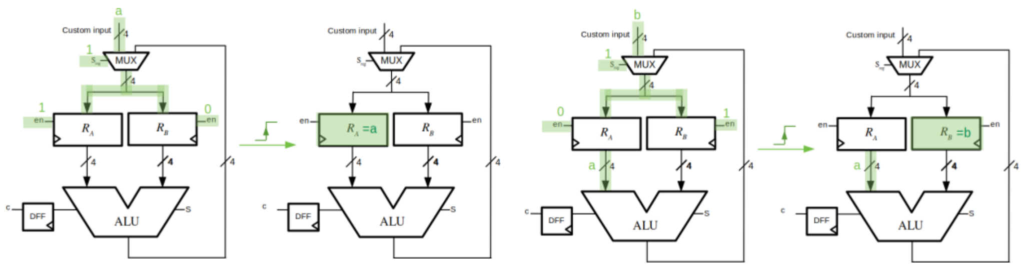
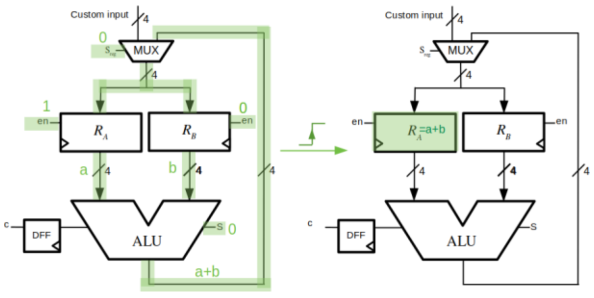


Figure 4: "Swapping" Control Signals Timing diagram

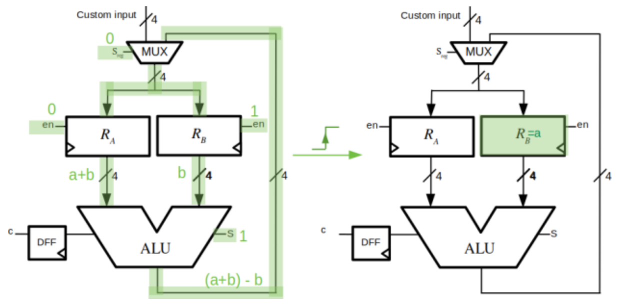


Step 1

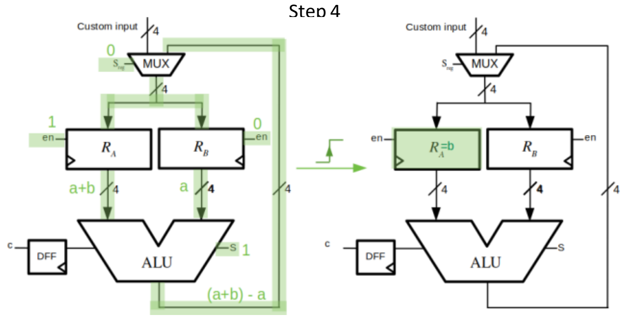
Step 2



Step 3



Step 4



Step 5

7 Output Register

A small addition to our design, the output register R_O. We shall use this register to store the final result after all the processing for the user to refer. The register is directly connected with R_A.

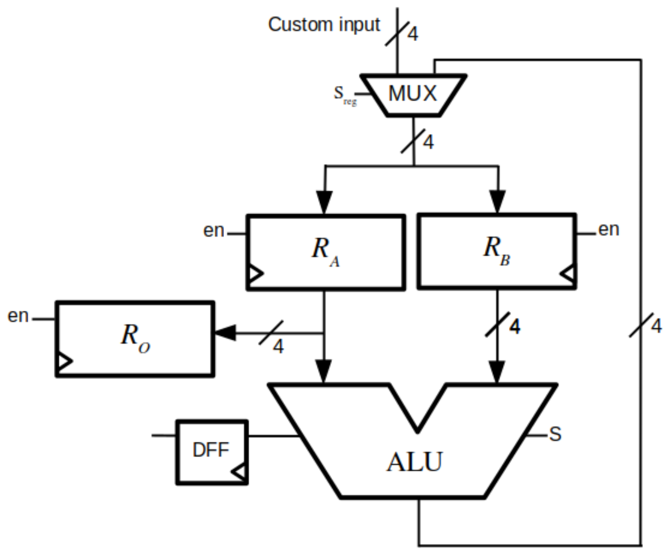


Figure 5: Output Register

7.1 Register De-multiplexing

There are now three signals associated with the registers, en_A, en_B and en_O. In order to reduce these signals we will use a 2-to-4 decoder (two inputs and four outputs) with a truth table shown below,

Input (D_0, D_1)	**Output** (O_0, O_1, O_2, O_3)
00	1000
01	0100
10	0010
11	0001

Now the outputs of decoder are connected such that O_0 to en_A, O_1 is to en_B and O_2 to en_O,

Input (D_0, D_1)	**Output**
00	R_A input enabled
01	R_B input enabled
10	R_O input enabled
11	no operation

Keep in mind, from now onward we will call the value of D_0D_1, "**address**" of the corresponding register.

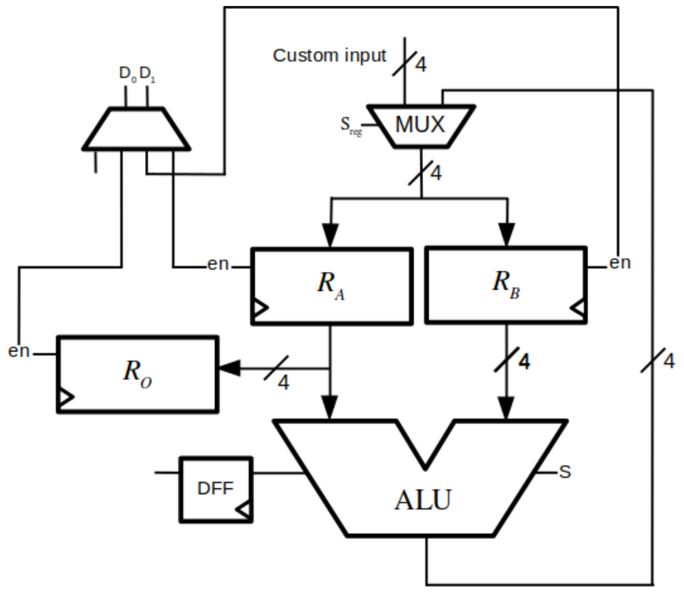


Figure 6: Design with Decoder Incorporated

8 Control Logic

We cannot apply logic values on the control signals manually, all the time. To avoid this inconvenience, we can store the values of control signals in a non-volatile memory (ROM or EEPROM) and connect its output lines to our design. Let’s call the values of the control signal stored in the memory as Instructions.

8.1 Instruction Memory

Memory is an array (having rows and columns) of cells, which can store binary data (1 or 0). For our design we will talk about those memories having 8 cells (or bits) in each row. Each 8-bit wide row can be accessed by its address starting from zero to (length of array - 1). We will store the set of control signals one in each row. For example in “swapping” test, the set of control signals was 110x, 101x, 0100, 0011, 0101 . We can store 110x in row(0), 101x in row(1) and so on. For an 8-bit row, each set off signals (also called **“instruction”**) will take only four bits, the rest of bits, we don’t care.

8.2 Program Counter

In order to set the signals (or instruction) on the control lines of our design before the positive edge of clock arrives, we will be using a 4-bit counter connected to address line of instruction memory, such that initially the value of counter is zero with zeroth row activated. As soon as the positive edge of clock comes, zeroth instruction is executed by our design as well as the counter gets incremented. Now the value of counter is one and row 1 of instruction memory is activated. The incoming positive edge will execute this instruction and the process goes on.

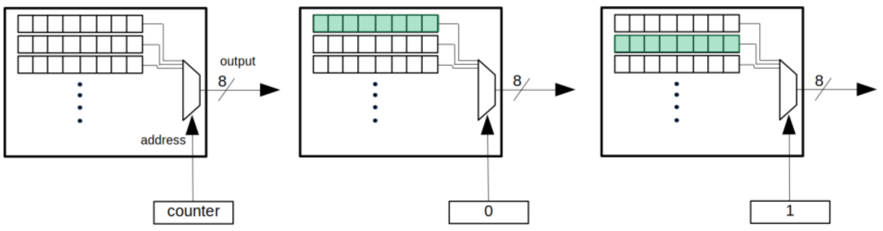


Figure 7: Memory and Program Counter

The program counter must have a 4-bit input and a load control such that when some custom input is applied to 4-bit input of counter and load control is kept high on the arrival of positive edge, the counter stores that custom input into it and start counting from that specific value on the next edge of clock. We wish to design a combinational logic around this feature of counter.

$$Load = J \quad OR \quad (C \quad AND \quad \text{“Carry out”})$$

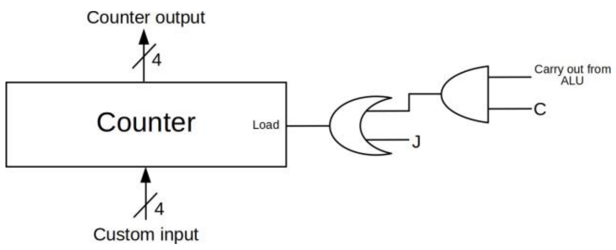


Figure 8: Jump Logic

Here “J” and “C” are control signals in our hand such that when we apply some custom input to counter and raise J to 1, the counter starts counting from custom input. Similarly, when the carry out from ALU is 1 and we raise C to 1, the counter starts counting from custom input. This phenomenon is known as **“JUMP”** or **“BRANCH”** in computing language.

8.3 Instruction Encoding

Since the rows of instruction memory are 8-bits wide, we wish to adjust our all control signals to these 8-bits. Moreover, our custom input is also supposed to be a part of this instruction. One scheme to adjust all these signals is,

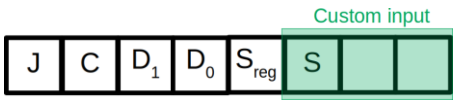


Figure 9: 8-bit wide instruction

1. **Jump (J):** The seventh bit or MSB of the instruction is fixed for Jump. Whenever we set it to 1, the custom input will be loaded in the program counter.

2. **Carry jump (C):** The sixth bit is fixed for “jump if ALU operation produces a carry”. Whenever we set it to 1, the custom input will be loaded in the program counter.

3. **Register Address (D_1D_0):** The fifth and fourth bit of instruction is fixed for register address. The values 00, 01, 10 and 11 corresponds to R_A, R_B, R_O and no register, respectively.

4. **ALU or Custom input (S_{reg}):** The third bit is reserved for multi- plexing ALU or custom input. When it is 0, ALU’s output is directed to registers’ input otherwise, the custom input.

5. **Custom input (immediate):** The last three bits, zeroth, first and second are reserved for custom input. Due to adjustment problem, we have to compromise custom input length (to 3-bits only). The fourth bit is hard wired to zero (ground), which means, we can only load values ranging from 0 to 7 (either in program counter or registers).

6. **ADD or SUB (S):** The second bit of the instruction has dual func- tion. Apart from being part of custom input, it is also connected to ALU selection bit S. This decision is taken based on the observation that

when ALU is performing either addition or subtraction, there is no interference of custom input. Similarly, when we are loading some custom input to either registers or counter, we don’t care about ALU processing.

9 Programming

There are 9 functions we can perform with this machine. The functions and the set of control signals for them are,

Function	**J**	**C**	**D_1**	**D_0**	**S_{reg}**	**S**		
						Custom Input		
$R_A = R_A + R_B$	0	0	0	0	0	0	0	0
$R_B = R_A + R_B$	0	0	0	1	0	0	0	0
$R_A = R_A - R_B$	0	0	0	0	0	1	0	0
$R_B = R_A - R_B$	0	0	0	1	0	1	0	0
$R_O = R_A$	0	0	1	0	0	0	0	0
$R_A = imm$	0	0	0	0	1	imm[2]	imm[1]	imm[0]
$R_B = imm$	0	0	0	1	1	imm[2]	imm[1]	imm[0]
Jump to imm if carry out	0	1	1	1	0	imm[2]	imm[1]	imm[0]
Jump to imm	1	0	1	1	0	imm[2]	imm[1]	imm[0]

9.1 Fibonacci Numbers

Below is a Program that will produce Fibonacci sequence in the output register.

0	$R_A = 0$	00001000	load zero to A
1	$R_B = 1$	00011001	load 1 to B
2	$R_O = R_A$	00100000	push A to output
3	$R_B = R_A + R_B$	00010000	Add A to B
4	jump if carry	01110000	If A+B produce carry jump to start
5	$R_A = R_A + R_B$	00000000	Swap A and B
6	$R_B = R_A - R_B$	00010100	Swap A and B
7	$R_A = R_A - R$	00000100	Swap A and B
8	jump to 2	10110010	jump to 3rd instruction

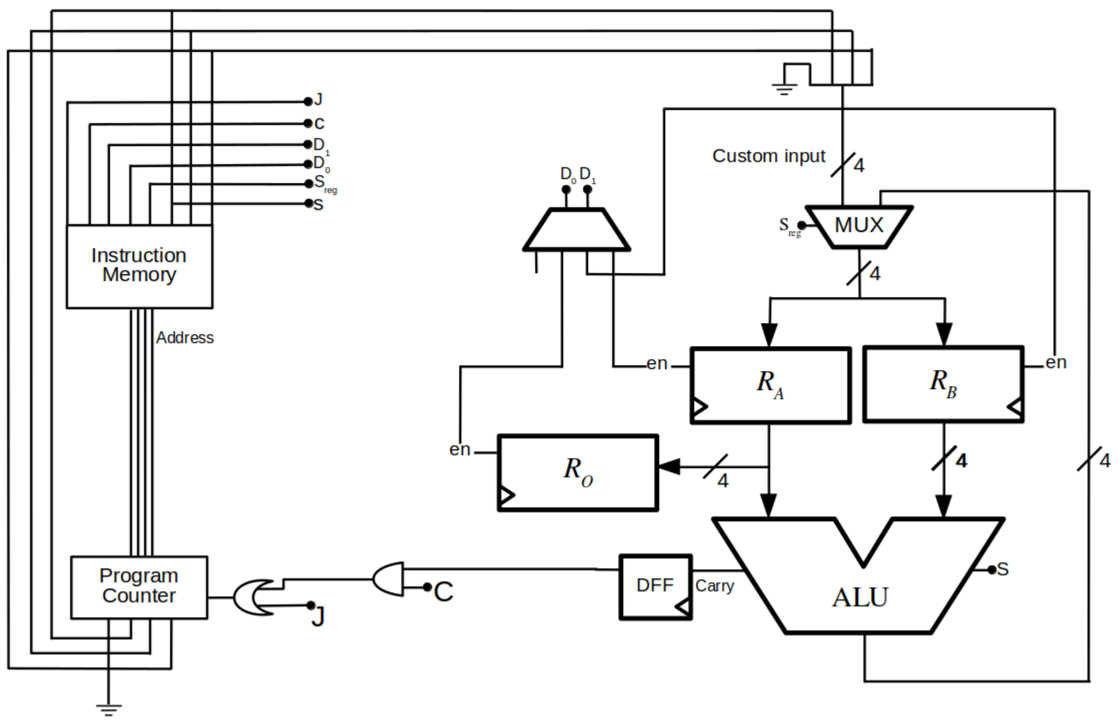


Figure 10: 4-bit Number Crunching Machine

10 Concluding Remarks

The design of a number crunching machine presented in this document is actually a baby processor core. All the processing units integrated into your laptops, mobiles and embedded devices are based on the same basic principle. You can implement some other interesting programs like multiplying two numbers or generating tables of 1, 2, 3, . . ., etc.

You have to extend the above design and build a hardware prototype of a 6-bit wide processor. If you are able to successfully complete this assignment and can extend or adapt it according to your need, you have gained a strong hold on the basics of Processors.