Namal University, Mianwali
Department of Computer Science

IMPLEMENTING A PROGRAMMABLE MICROARCHITECTURE

Digital Logic and Design Project
Using Logisim Evolution

Made By: Manahil Mushtaq

1. Abstract:

A programmable micro-architecture can have various numbers of operations performed. In this semester Project of DLD Course that marks the completion of course, we must perform 6 of some basic operations. In the following part of report, I present the design of Instruction format and design of Program Counters that helped run the programs sequentially and at their desired times.

2. Problem Statement:

Implementing a functional prototype programmable microarchitecture which can execute 6 operations: 3 arithmetic and 3 logical.

3. Design Methodology:

We have adapted the following design methodology:

3.1 Functional Requirements:

1. Our implemented Programmable micro-architecture is required and limited to perform the following tasks:

3.1.1 Operations

3 Arithmetic Operations namely:

- Addition
- Subtraction
- Multiplication

And 3 Logical Operations which are comparison operations:

- Greater than
- Equal to
- Lesser than

2. It must contain 8 registers to support the microarchitecture.

3. The Instruction and Data memory must be stored separately.

Next, for memory storage, we adapted our design so that the memory is divided into two parts as required: Instruction Memory and Data Memory.

3.1.2 Instruction Memory

The Instruction Memory stores the instructions; one instruction is of 11 data bits. We will see that in detail in the next section.

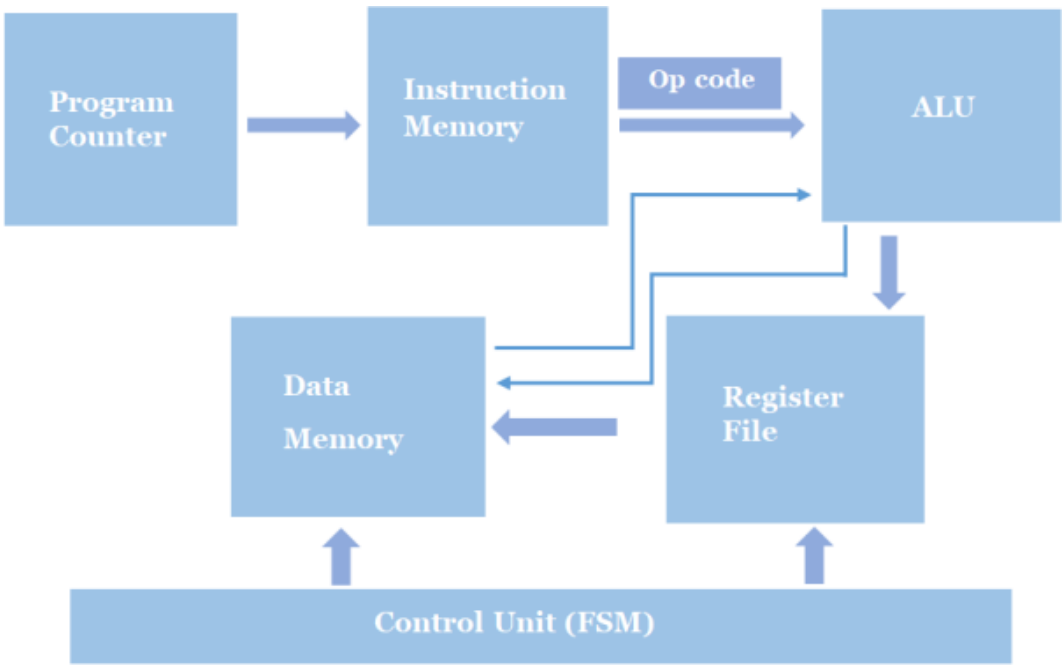
3.1.3 Data Memory

Data Memory stores the value of A, B and Result of the operations performed by ALU, for which the instruction was given in Instruction Memory. We will see the detailed functionality of this in the next section.

3.2 Approach

In our approach to its implementation, apart from separate Instruction and Data Memory, we have used **ALU** to carry out Arithmetic and Logical operations, Register File to store results of the said operations, **FSM** and **Counters** which are working as **Control Unit** for the instructions to execute.

4. Block Diagram



This block diagram represents our whole project. In this Block diagram the program counter is controlling the instruction memory. It decides after how many cycles the next instruction should be executed.

In instruction memory, we gave 11-bit data for instruction. Most significant 4-bits for the address of A, next 4 for the address of B and least significant 3-bits are OP-Code. The instruction memory we used has 8 addresses meaning it is 3 bits data width.

Next is ALU, this is an arithmetic and logic unit. As specified in design requirements it performs 3 arithmetic and 3 logic operations. The A and B are fed to the ALU through Data Memory. And the result of the ALU is again written in Data Memory. There is a MUX inside ALU that controls which operation is going to show its output. Instruction which is written in report of part ALU.

In the register file, we are writing the results from ALU. However, since A and B are coming from the same end of Data Memory so we need different registers with different enabler times so we can write their separate input values in them.

Data memory, we give the values of A and B manually. The values of A and B are read from the data memory then a function is performed on them in ALU, and its result is also saved in data memory. There is a counter which decides where the address at which result should be stored.

FSM is the control unit of the above shown CPU (Central Processing Unit). FSM controls when and how long the enabler of a register should be ON so that the value which that register was required to have been loaded into that. It also controls when the DM (Data Memory) Read is supposed to be ON, when the DM write is supposed to be ON and in how many clocks cycles a value is stored.

FSM works to automate the circuit and decides when, where and what should be loaded.

5. Circuit

The circuit has the following parts also mentioned above in the design methodology.

- Program Counter
- Instruction Memory
- Data Memory
- ALU
- Register File
- Control Unit (FSM and Counters)

5.1 Program Counter

Let's first talk about Program Counter. It keeps track of the instructions to be carried out. It decides the number of clock cycles required to carry out a single instruction. In our design, it uses 24 Clock cycles for that purpose. After 24 cycles are completed, it moves the instruction address to the next instruction and then carries out that instruction in 24 cycles.

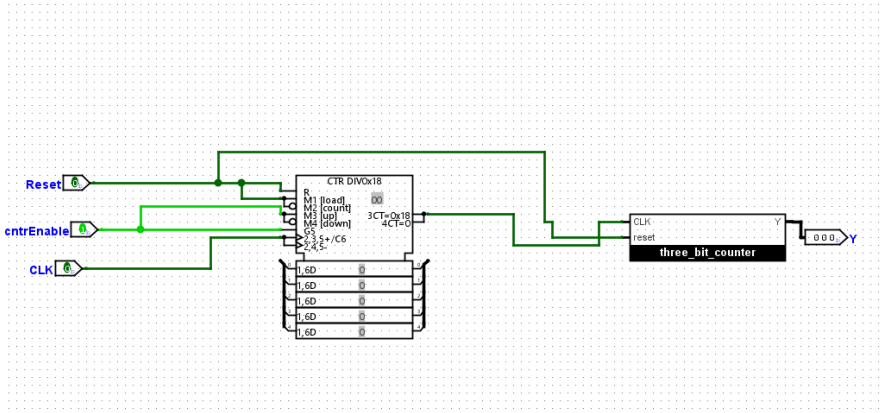


Figure 1: Program Counter

In the above Figure (1), we see the Program Counter.

We see another sub-circuit of counter in this. Following is an image of that:

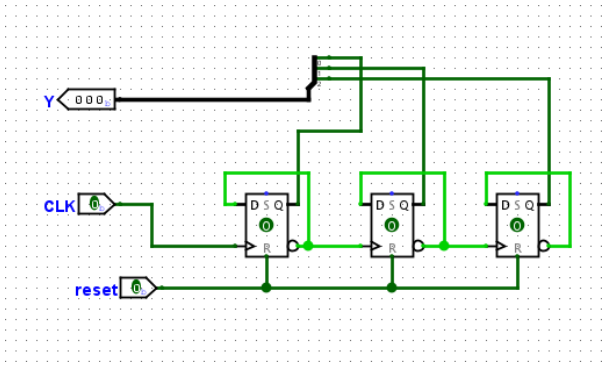


Figure 2: 3-bit Up Counter

This is a simple 3-bit Up counter. On every rising edge of clock, the counter increments by 1. It keeps incrementing till it reaches 111 (Highest binary number that can be represented in 3 bits), for which the decimal representation is 7. This 3-bit clock is further used in the overall program counter. As we will see later that our Instruction Memory is 3 bits and that means 2^3 (8) instructions can be executed in it. And a 3-bit clock is sufficient to carry this number of instructions.

This clock performs the task of executing all the instructions that are written in Instruction Memory. Meanwhile, the other clock, which I mentioned before this one in Figure (1), carries out the clock cycles for one such instruction (all tasks to be done in 1 instruction).

The clock in Figure (1), is integrated with the clock in Figure (2) such that once it reaches the maximum value which is 24 (as in 24 clock cycles), the instruction clock in Figure (2) increments by 1 and this keeps going on for the rest of the instructions.

This program counter is then integrated with the Instruction Memory in the circuit.

5.2 Instruction Memory

Instruction memory, as briefly discussed, is used to store the instructions. It is a RAM module taken from Logisim Evolution. It has 3 Address Bit Width that stores a number of 8 (2^3 = 8) instructions and data bit width of 11 bits. That means that every instruction is of 11 bits. The most significant **4 bits** are used to define the address at which **A** (input 1) will be stored in Data Memory and the next **4 bits** after that define the address at which **B** (input 2) will be stored in Data Memory. The remaining least significant **3-bits** are **OP-Code.**

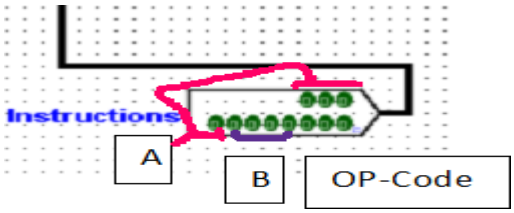


Figure 3: Instruction bits

This is how we can give the instruction that will be fed into Instruction Memory.

OP-Code, also regarded as operation code, is a 3-bit (3-bits in our case however the number of bits vary with the number of operations ALU is performing) binary number which decides the result of which operation will be shown or given as output. Every operation is assigned an OP-Code which goes as one of the inputs (along with **A** and **B**) to the ALU and the result for that operation is given as output.

An image of Instruction Memory that is integrated in the complete circuit:

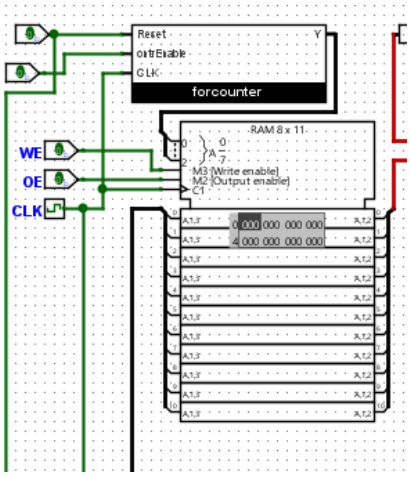


Figure 4: Instruction Memory

The above is an image of Instruction Memory, and we can also see the Program Counter. Both share the same clock which moves to execute the next instruction after 24 clock cycles are completed.

At WE (Write Enable) the instructions are read and written into Instruction Memory from the instruction input where these are originally written. However, there is another way of writing these instructions, which is to write these instructions manually. We can manually enter all the instructions which will then be executed sequentially.

This memory will give the output i.e., the instruction that is fed when **OE** (Output Enable) is turned ON. In our circuit, we have used 2 splitters to separate the output in such a way that the most significant 4-bits (address for A) and the next 4-bits (address for B) go into a MUX which then sends these addresses into Data Memory and the OP-Code goes into ALU to designate the operation required to perform.

The output of ALU is then sent to the register file where the outputs are stored in registers, and this is only the 4 bits of the result. However, when A and B both are 4-bit inputs then the result of multiplication is in 2n (where n is the number of bits), i.e., in 8 bits. The most significant 4 bits are part of C-out, which is the carry out, the one that overflows.

Now as we are aware that the result of multiplication is stored in 8 bits, so we have used the most significant bit that is stored in register for **C-out** and stored that in the memory and in the next state, we have stored the output value from the Register File, which consists of the least 4-bits of result for multiplication and the total 4-bit answer for the rest of the arithmetic and logical operations.

5.3 Data Memory

Data memory is also a RAM module that comes with Logisim Evolution. It has 4-bit data width and 4 address bit width. It stores 4 bits of data in hexadecimal notation.

However, we have designed it to work differently than the instruction memory. In Data Memory, the addresses of inputs A and B are designated by the instructions that are coming as inputs from MUX used earlier and the MUX is controlled by the **Control Logic** (**FSM**). The value of A and B are thus stored in 2 separately which are named Register 1 and Register 2, respectively.

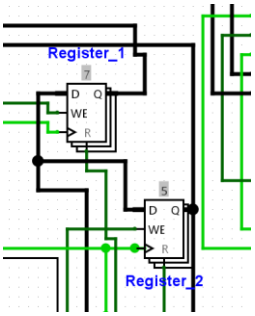


Figure 5: Register 1 and Register 2

The output from Register File and C-out registers is then going into another MUX which is also controlled by the FSM. Now what happens is that, as described above, firstly, the value from the C-out register is stored and then the output from Register File is stored. However, we do not always need to

store the value of C-out since that is only for multiplication, so we maintain it at 0 in the rest of the operations.

5.4 Limitations of storing result in Data Memory

While working on our design we faced a limitation that whenever a function is executed it will always store 8 bits worth of data, irrespective of the fact that C-out being 0 in case of operations other than multiplication.

Although the data bit width is only 4 bits and since the result of multiplication must be stored in 8 bits, we used 2 locations of 4 bits to store that result, this happens in case of multiplication. In other operations, C-out is 0 and it stores a 0 in one location in Data Memory and the 4-bit result in other.

Let's look at the figure for better understanding.

Let's take for an example that our Instruction memory consists of the following instructions.

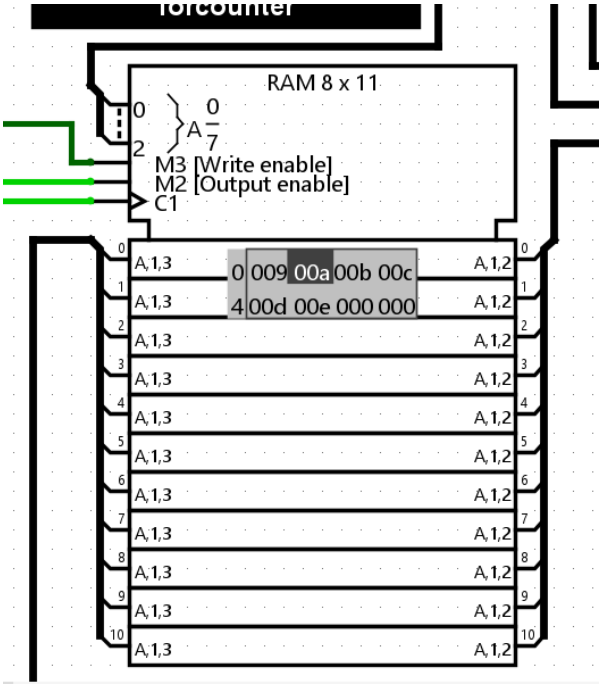


Figure 6: Instruction Memory with sequential instruction writte

Instruction No.	Address of A	Address of B	OP-Code
1	0000	0001	001 (ADD)
2	0000	0001	010 (Subtract)
3	0000	0001	011 (Greater than)
4	0000	0001	100 (Equal to)
5	0000	0001	101 (Smaller than)
6	0000	0001	110 (Multiply)

Now we will see how results for these instructions will be stored in Data Memory.

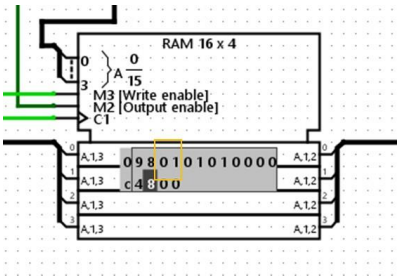


Figure 7: Result of ADD stored in Data Memory

Here the first 2 addresses store the values of inputs A (value: 9) and B (value: 8) which we have entered manually. All the functions are going to be performed on these two inputs.

Let's look at the **ADD** operation (Instruction for which is **009** (hexa-decimal for 0000 0001 001) and its result is circled). Its result is going to be stored in the next 2 memory addresses after addresses of A and B inputs. 3rd address (represented in binary by 0010 since the addresses start from 0000 – 1st address) is where C-out (which is 0 for ADD) is stored. In the 4th address (0011) the output of result from ALU followed by Register File is stored. Now we see the 3rd address is 0.

This is same for subtraction and for Logical operations too. Now we know this is a limitation in our design, that despite the data memory being 4 bits width, we must store our result in 2 locations of 8 bits.

On the brighter side, our output for multiplication is stored perfectly. Let's see the figure again this time for instruction **00e** (hexa-decimal for 0000 0001 110)

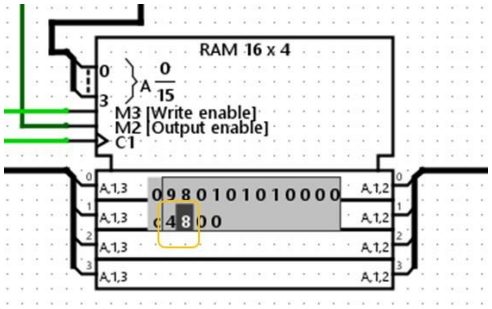


Figure 8: Result of Multiplication stored in Data Memory

The output for multiplication of 9 and 8 is 48 (which is in fact represented in hexa-decimal value for convenience)

There is no problem when it comes to see whether the result stored is correct or not, the only problem is that it takes up an extra 4 bits of location in operations other than multiplication.

Now let's look at an example program to conclude:

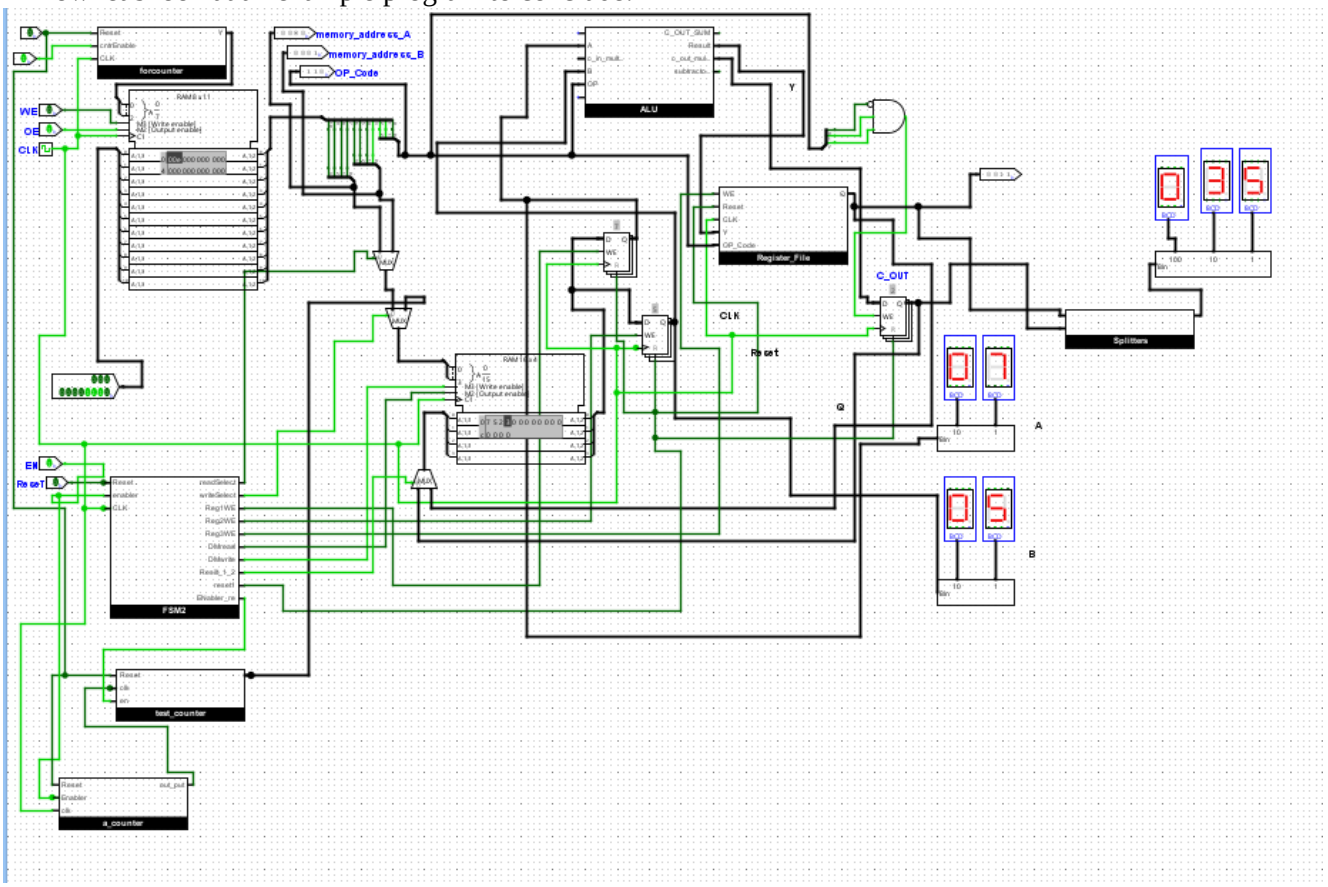


Figure 9: Circuit carrying out a multiplication of two inputs

Input A = **7**

Address of Input A = **0000**

Input B = **5**

Address of Input B = **0001**

OP-Code = **110 (for multiplication)**

Result = **23 (in hexadecimal, 35 in decimal as shown by BCD)**

6. Improvements:

The circuit can be improved and only the result of multiplication be stored in 2 locations (8 bits) and the result of other arithmetic and logic operations should be only taking up 1 location (4 bits).

It will be even better if we can manage to store the results of logic comparison into only 1 bit since its result is only 1 bit (1 for True and 0 for False).

7. Conclusion:

The complete process of making a functional prototype programmable computer taught us multiple things. In the non-technical domain, we understood the importance of teamwork and consistency. In the technical domain we got a lot better at handling circuits and understanding the working of different components and then the complete circuit.

Furthermore, we integrated all the different components namely Program Counter, Instruction Memory, ALU, Register File, Data Memory into the complete circuit. We ensured the working of all components separately, which can also be found in other parts of the report. This resulted in complete microarchitecture of a programmable computer.