Experiment 9: Designing and Implementing a Simple CPU

A. Objectives

- Familiarize students with partitioning digital system into a datapath and a control unit
- Familiarize students with datapath design
- Familiarize students with control unit design
- Design and Implement a simple CPU

B. Material

- Basys 2 board
- Xilinx ISE CAD software
- CloudV.io

C. Introduction

In this experiment, you will use Verilog HDL and ISE toolset to design and implement a simple CPU. In general, a CPU is a digital system with a datapath that supports several micro-operations. The CPU control unit sequences these micro-operations according to instructions stored in a program memory (external to the CPU). A program is made out of instructions. An instruction is stored in the program memory as a number (the instruction word). The bits that make the instruction word can be divided into fields. Each field tells something about the micro-operations that will be performed to execute the instruction. The CPU control unit is responsible for decoding the instruction word and generating the control signals required to enable micro-operations needed to execute the instruction.

The CPU we are targeting in this experiment is very simple RISC CPU and has small set of simple instructions (No memory load or store instructions). Each instruction can be executed in one clock cycle only. The CPU details are discussed in the following subsections.

C.1 The ALU

The ALU can perform 6 different operations, required to implement the supported instructions, on 8-bit operands. The ALU function Selection (**ALUSel**) is connected to the first 3 bits (least significant) of the instruction word op-code field. Please note that the ALU is not used for the IN or OUT instruction. For the **BEQ** and **BNE**, the ALU is used to subtract the instruction operands and ALU **ZFlag** output is used to indicate whether the condition is true or false. The following table summarizes the ALU functions

Table 1. ALU Functions

ALUSel	Function
000	Addition
001	Subtraction
010	Subtraction
011	Subtraction
100	Bitwise OR
101	Bitwise NOR
110	Bitwise AND
111	Bitwise XOR

An example ALU implementation in Verilog is outlined in Figure 1.

C.2 The Register File

The CPU has 16 8-bit registers. The registers are organized as a special multi-port memory called the register file. The register file has two read ports and one write port. Figure 1 shows a possible implementation of the CPU register file in Verilog HDL.

```
module ALU (A, B, Result, ALUSel, ZFlag);
        input [7:0]             A, B;
        input [2:0]             ALUSel;
        output reg [7:0]        Result;
        output                  ZFlag;

        assign ZFlag = (Result==8'b0);

        always @ (*) begin
         case (ALUSel)
               3'b000: Result = A + B;        // addition
               3'b001: Result = A - B;        // subtraction
               3'b010: Result = A - B;        // subtraction
               3'b011: Result = A - B;        // subtraction
               3'b100: Result = A | B;        // bitwise OR
               3'b101: Result = ~(A | B);     // bitwise NOR
               3'b110: Result = A & B;        // bitwise AND
               3'b111: Result = A ^ B;        // bitwise XOR
         endcase
        end
endmodule

module regFile(clk, rst, A_data, B_data, W_data, A_addr, B_addr, W_addr, wr);
        input           clk, rst;
        output[7:0]     A_data, B_data;
        input[7:0]      W_data;
        input[3:0]      A_addr, B_addr, W_addr;
        input           wr;

        reg[7:0]        RegFile[15:0]; // declaring array of 16 8-bit registers - Register
        File

        // Reading register
        assign A_data = RegFile[A_addr];
        assign B_data = RegFile[B_addr];

        // Writing to a register
        always @ (posedge clk) begin
         if(!rst)
               if(wr) RegFile[W_addr] <= W_data;     // writing data to register number
        W_addr
        end
endmodule
```

Figure 1. ALU and Register File Implementation in Verilog HDL

C3. Connecting the Register File to the ALU

If we connect the register file to the ALU as shown in Figure 2 we can perform several operations on the data stored in the register file. Table 2 lists all possible operations (usually called micro-operations) that can be performed by the hardware outlined in Figure 2. Fill the rest of Table 2 to show the values of c1, c2, …., c7 to enable each micro-operation. The hardware outlined in Figure 2 is called a datapath. In general, a datapath is made out of registers (to store data), combinational logic (to process the data) and the wires/busses (interconnect) that connect the registers

to the combinational logic. The operations that can be carried out by the datapath are called micro-operations and they are enabled by the control signals (c1, c2, …., c7).

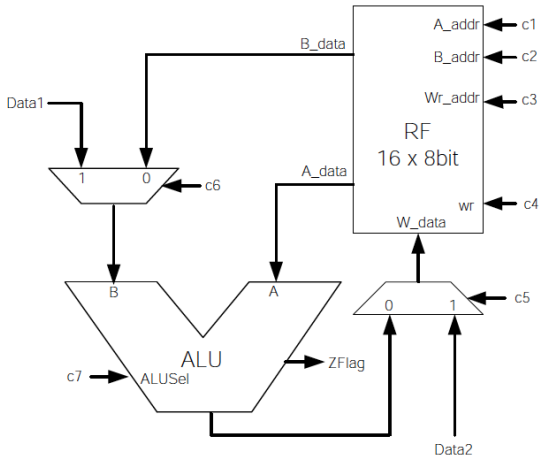


Figure 2. The register file connected to the ALU

Table 2. Micro-operations (µOP) of the data path of Figure 55

Operation	c1	c2	c3	c4	c5	c6	c7
RF[des] ← RF[src1] **op** RF[src2]	src1	src2	des	1	0	0	op
RF[des] ← RF[src1] **op** Data1							
RF[des] ← Data2							

The following is a possible Verilog implementation of the hardware outlined in Figure 2

```
module DataPath(clk, rst, Data1, Data2, c1, c2, c3, c4, c5, c6, c7);
        input           clk, rst;
        input[3:0]      c1, c2, c3;
        input           c4, c5, c6;
        input[2:0]      c7;
        input[7:0]      Data1, Data2;

        wire[7:0]       a_bus, b_bus, w_bus, b_mux, w_mux, alu_result;

        assign   b_mux = (c6) ? Data1 : b_bus;

        assign   w_mux = (c5) ? Data2 : alu_result;

        ALU alu(.A(a_bus), .B(b_mux), .Result(alu_result), .ALUSel(c7), .ZFlag());

        regFile RF( .clk(clk), .rst(rst), .A_data(a_bus), .B_data(b_bus), .W_data(w_mux),
                    .A_addr(c1), .B_addr(c2), .W_addr(c3), .wr(c4));

endmodule
```

To implement an algorithm (made out of a sequence of micro-operations), the values of the control signals should change in steps. Such stepping (or sequencing) can be automated if the control signals values are stored in a ROM memory then read one by one from the memory. A binary counter is used to generate the memory address. This is outlined in Figure 3. The sequence of micro-operations used to implement a specific operation is called a micro-program. The binary counter that provides the address to the ROM is called the Program Counter (PC).

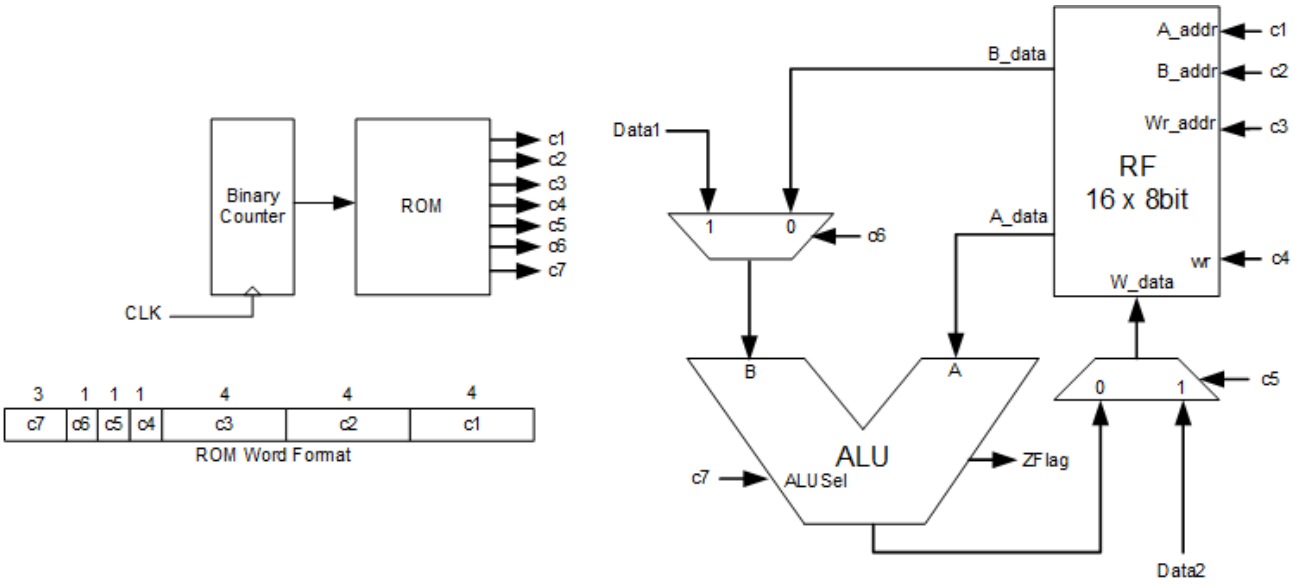


Figure 3. Datapath + ROM for the control signals

The outlined hardware in Figure 3 supports only sequential execution of micro-operations. To allow implementing control structures in the micro-programs (such as loops), the hardware should be modified to allow loading the binary counter (also called the Program Counter of "PC") with the address of the micro-operation word we want to execute next (*next_addr*). This micro-operation is usually called unconditional jump. Figure 4 outlines such modification. Now, the PC is a binary counter with a parallel load capability. Also, 2 more control signals are added (*c8* and *c9*). The ROM word is extended to include *c8*, *c9* and *next_addr*. The introduced

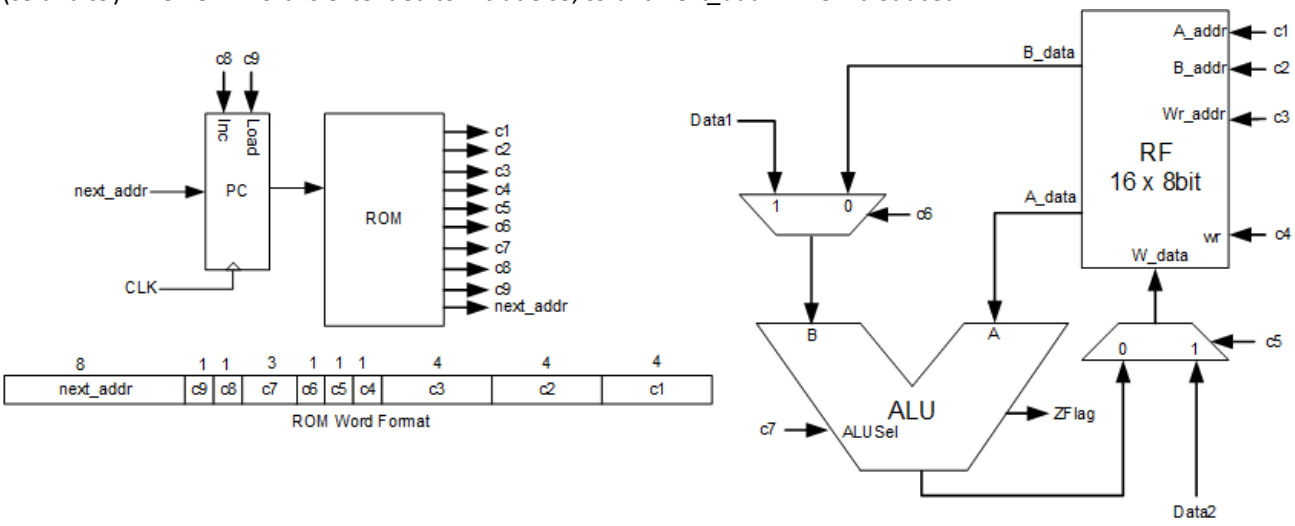


Figure 4. The updated Datapath to support control operations

In many cases, it is desirable to jump to the *next_addr* conditionally (conditional jump). This would enable the implementation of decision making structures (e.g., if else) in the micro-programs. A simple way to realize this, is to introduce a small combinational block to decide whether to load or increment the PC based on the value of the ALU zero flag (ZFlag). The ZFlag can be used to determine whether any two registers are equal or not (we subtract one from the other; if they are equal, ZFlag will be set to 1). This involves the addition of another control signal (c10) to indicate whether the jump micro-operation is conditional or unconditional. Figure 5 shows these updates.

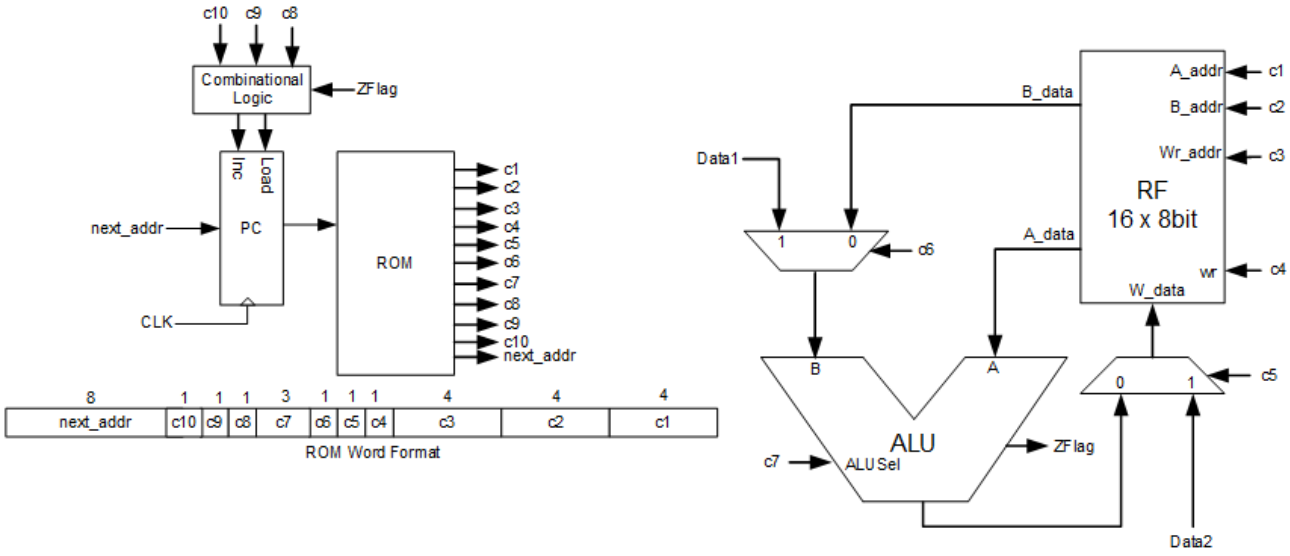


Figure 5. Updated datapath to support conditional jump micro-operations

C.4 The Program Memory (ROM)

There are two types of embedded memory in a Spartan-3 device: distributed RAM and block RAM. A distributed RAM is constructed from the logic cell's look-up table (LUT). The LUT can be configured as a 16-by-1 synchronous RAM, and multiple LUTs can be cascaded to form a wider and deeper memory module. The Spartan3E-100 device of the Basys 2 board can provide up to 15 Kbits of distributed memory, which is small, compared to a block RAM or external memory. Furthermore, since the distributed RAM uses the logic cells, it competes for resources with the normal logic. Thus, it is feasible only for applications that require relatively small storage.

A block RAM is a special memory module embedded in an FPGA device and is separated from the regular logic cells. It can be thought of as a fast SRAM wrapped by a synchronous, configurable interface. Each block RAM consists of 16K data bits plus optional 2K parity bits. It can be organized in different widths, from 16K by 1 to 512 by 32. The Spartan3E-100 device has 4 block RAMs, totaling 72K data bits. These block RAMs can be used for intermediate-sized applications, such as a FIFO, a large look-up table, or an intermediate-sized local memory.

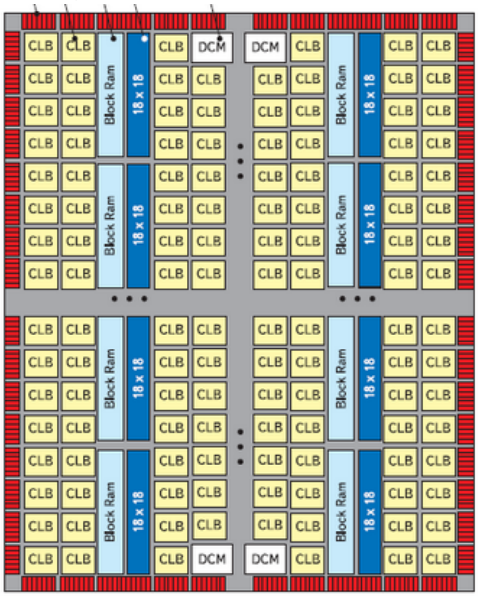


Figure 6. Spartan 3E FPGA Internals

The CORE Generator System is a design tool that delivers parameterized cores (also called IP's) optimized for Xilinx FPGAs. It provides you with a catalog of ready-made functions ranging in complexity from simple arithmetic operators such as adders, accumulators, and multipliers, to system-level building blocks such as filters, transforms, FIFOs, and memories. To specify the values for these modules, you must load a COE file when you create the IP core.

C.4.1 Generating a Coefficient File

To load a generated memory with data, a coefficient file (with the extension ".coe") can be used for this. To prepare a coefficient file containing the required data for a memory (for example 256x20-bit), follow the following steps.

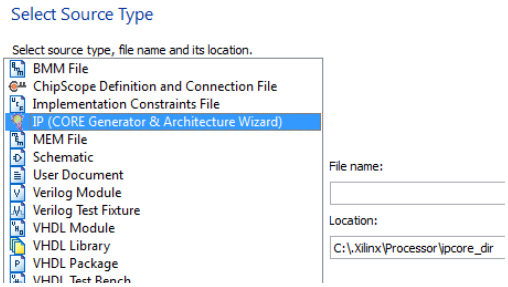
- Open Notepad (or any other text editor)
- Specify the radix used for the memory initialization values by typing:
  `memory_initialization_radix = 16;` (Hexadecimal base specified)
- Specify the values stored in memory by typing (the data elements are random ones):

```
memory_initialization_vector = 80001,
00120,
80201,
…….
etc
```

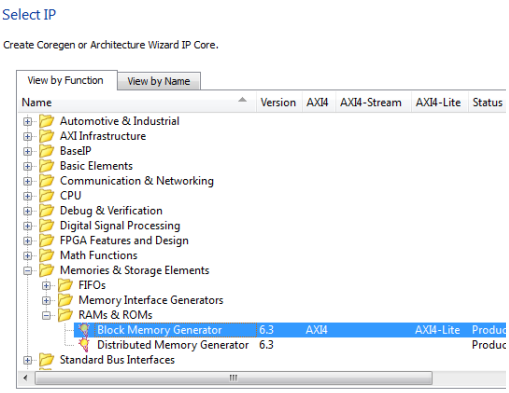
- When you finish, your file add **Semicolon** to your last entry
- Save the file as **"Mem_Init.coe"** in **the working directory** and close it.

C.4.2 Generating Memory core using Xilinx Core Generator

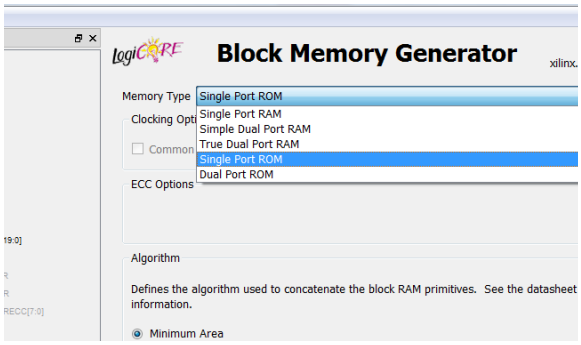
- Right click on your device name, select New Source
- Select Source Type: **IP core (CORE Generator and Architecture wizard)**
- Name your file
- Click Next



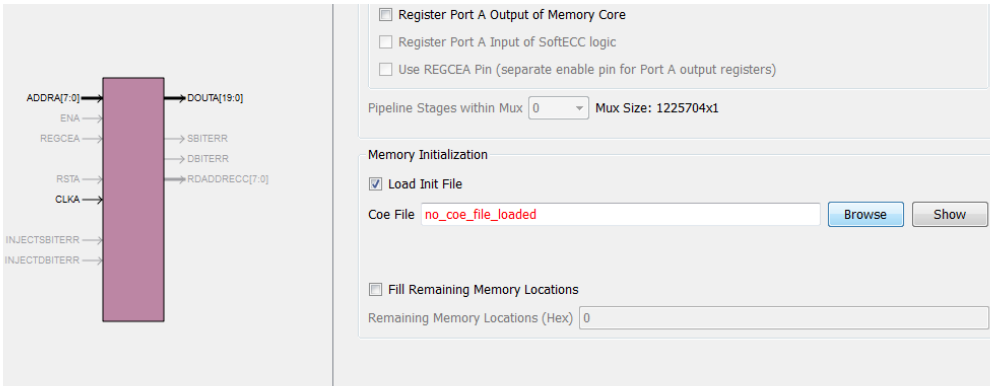
- In the "New Source" Wizard – select IP, expand Memories and Storage Elements, then expand RAMs and ROMs and then select **Block Memory Generator**.
- Click "Next" then click "Finish" to open IP customization window.



- In the IP customization window. Click Next to customize the core.
- Select Memory Type: **Single Port ROM**
- Click Next specify the **Read width: 20**, **Read depth: 256.** Change these 2 values as required.
- Click Next



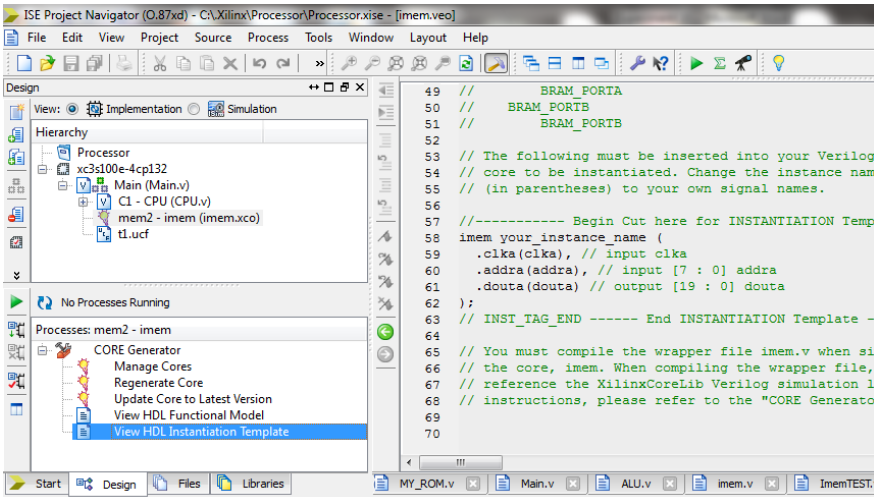
- Check the **Load Init File** Box
- Browse the **"Mem_Init.coe"** file
- Click Show to verify the memory contents
- Then Click Generate



C.4.3 Using the Generated memory

- In the Sources window, Select the memory file you generated
- In the Processes window, Expand **CORE Generator** then double click on **VIEW HDL Instantiation Template**

- Copy the instantiation snippet in your source code and connect the required signals to memory as mentioned in the Instantiation Template



D. Pre-lab work

- Complete Table 9.
- Model and verify the H/W outlined in Figure 58 using CloudV.

E. Experiment

1) Using Verilog HDL and ISE toolset, implement the datapath outlined in Figure 5.
2) Modify the datapath to include "*data1*" into the ROM word. What is the size of the ROM word after this modification? (Hint: Try to re-use existing fields).
3) Modify the datapath so that *c9* is not connected directly to the "load" control of the PC. Instead, the PC is loaded with *next_addr* when both *c9* and *ZFlag* are high.
4) Modify the datapath as per Figure 7. The modification involves the addition of an 8-bit register (*out_reg*) with its input connected to *A_data* (register file read port A). The control signal *c11* is connected to the register load control (`c11: out_reg ← RF[c1]`). Add *c11* to the ROM word.
5) Develop a microprogram to add the numbers from 1 to 10 and place the sum into the register *out_reg*.
6) Test your design on the Basys 2 board. Connect the LED's (8 of them) to the register *out_reg*.

F. Homework

1) Connect *Data2* to the 8 toggle switches. Modify the program to sum the numbers between 1 and a number you enter through the toggle switches.
2) Can we reduce the size of the ROM word? Note that: a micro-operation does not use all the word fields. Also, some control signals cannot be active at the same time (e.g., c8 and c9).

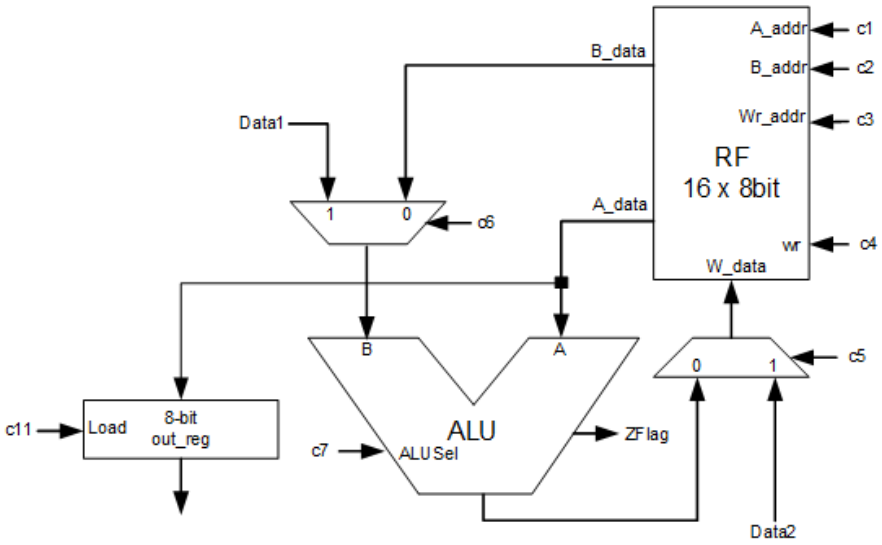


Figure 7. The Updated Datapath to support I/O