Lab 9: Building a Computer

Introduction

A conceptual block diagram of a simple computer is shown in Figure 1. In the previous labs, the Address Multiplexer (`Addr_Mux_Sel`), Arithmetic Logic Unit (`ALU`), Computer Control Unit (`CCU`) and required registers (8-and 1-bit) were already designed. In this lab all the components will be put together to build a computer. One of the missing blocks is the memory block, which can be found on the class webpage (NMT EE 231). To program the computer the initialization code must be modified. The last component remaining is a clock divider. The Spartan 7 has a built in clock, but it runs at 12MHz. Since this is too fast, the computer needs a counter that will divide the clock speed making the computer's operation visible. In this lab students will learn to use Vivado's graphical design tools to implement the computer (Processor) as shown in Figure 1.

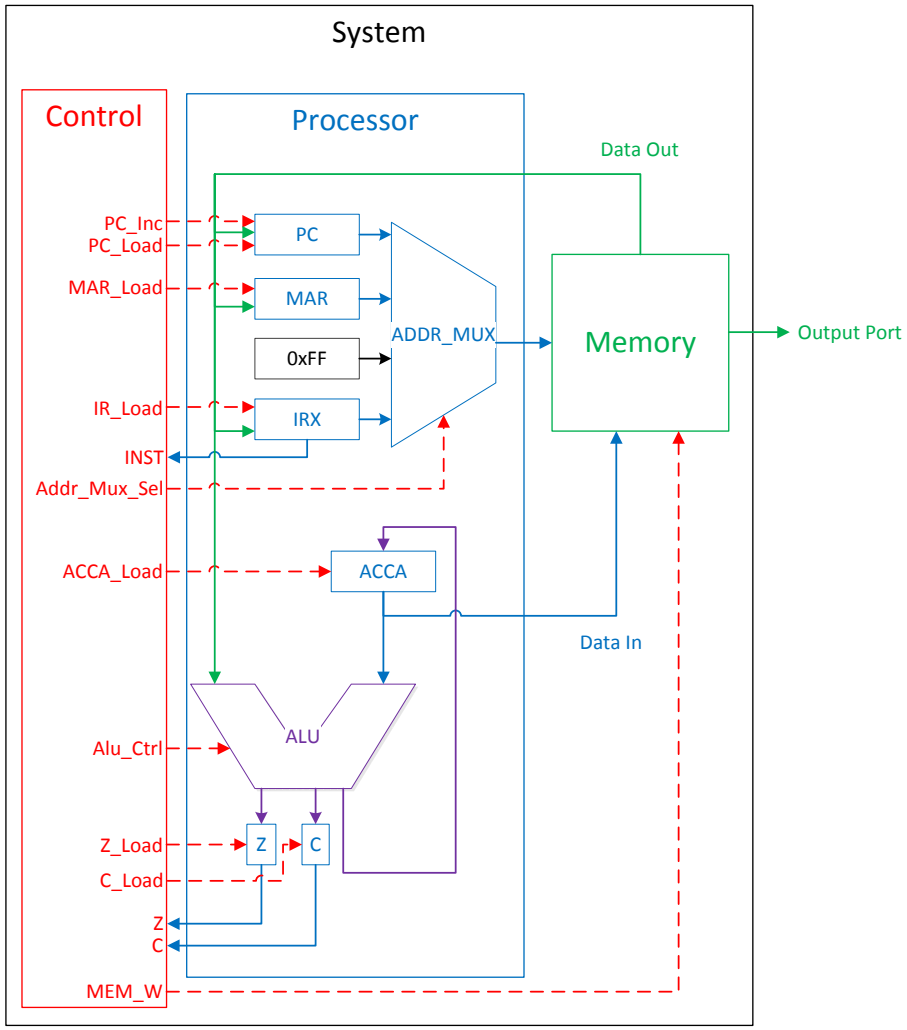

Figure 1: Processor Block Diagram

1 Prelab

Week 1:

1.1. Read the Prelab, Lab, and Supplementary material.

The Spartan 7 has a built in 12 MHz clock. If we were to use this clock for the "running lights" program, all of the LEDs would appear to be on at once because of the very high clock speed. To avoid this the computer needs a clock divider. The divider will be similar to a counter, in that it will count clock edges, but instead of just resetting or incrementing an output, it will toggle a new "clock" signal.

1.1. How high would a counter need to go in order to slow the 12 MHz clock signal down to 10 Hz?

1.2. How many bits does that number require?

1.3. Create a clock divider.

Week 2:

1.1. Using the instruction set provided in the Table 1, write a computer program to generate the two "running light" sequences (two separate program routines, you do not have to have both simultaneously, just one at a time).

One way to accomplish this is to start with an 8-bit number 1111 1110 (where the zero represents the LED that would be off). Then left shift that number and increment the sequence to back-fill with ones, then repeat; once you have reached the end, jump back to the beginning of the program. There are many different solutions, differing in requiring more processing/clock cycles, or memory usage. To save the desired output to the LEDs (Output_port) save the value to the address 0x80. Figures 2 and 3 show the expected output at different time steps for each of the two sequences to be generated.

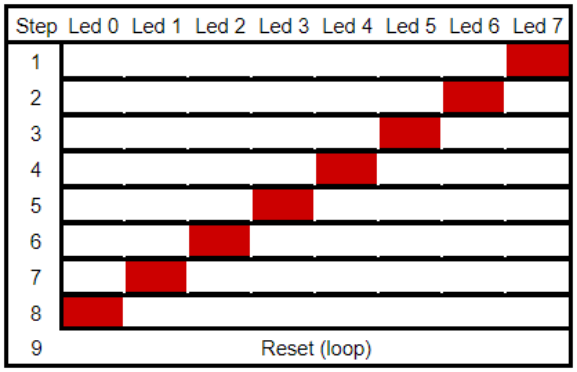

Figure 2: Running '1' Left

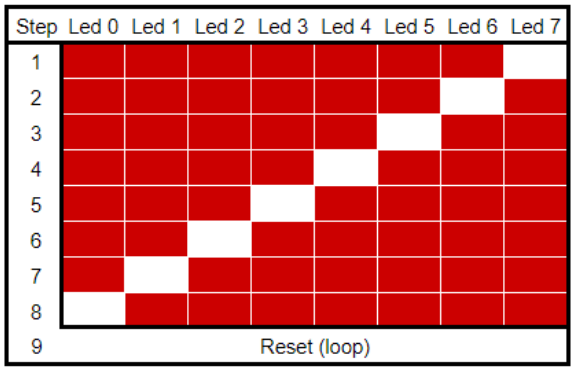

Figure 3: Running '0' Left

2 Lab

2.1. Using the graphical method as shown in Section 3, design and build the entire simple computer.

In the CCU lab you, created Operational codes, or "Opcodes".These codes are basic instructions that can be written into memory for the computer to later read as Operational instructions. As outlined in the CCU lab, the computer will first fetch the Opcode instruction, and then execute either 1 or 2 execution cycles depending on that Opcode. By writing values into memory, we can tell the computer to do what we want in this case running lights, therefore "Programming" the computer. The programming technique of using Opcodes is a primitive form of programming simple computers and microcontrollers called **Assembly**.

2.1. Enter your "running lights" program into the `Memory_Block.v` file.

2.2. ⋆⋆ Simulate the computer you have created in the prelab (*Week 2*).

2.3. ⋆⋆ Run your code on your board.

3 Supplement: Graphical Design of Processor

3.1. Start a new project, and include all of the necessary modules (`Addr_Mux_Sel`, `ALU`, `CCU`, `1-bit Registers`, `8-bit Registers`), and create a new .v file for `Memory_Block.v` and your `Clock_Divider.v`

3.2. On the top bar, select **Flow** → **Create Block Design** → and name and **create** the block design.

3.3. Now **right click** on the empty Block Diagram, then select **add module...**

3.4. Select the module that you desire to add, and it will be placed into the Block Diagram. Continue adding all of the required modules.

3.5. Once all of the modules have been loaded, click and drag each of the inputs to their outputs, completing the diagram. If desired, you can also drag around individual module blocks for better placement.

3.6. To add an output of input, **right click** the Block Diagram, and select **create port**. After selecting any options you may click and drag the port to the desired connection.

Hint: You may want to add additional output ports during simulation, to assist in debugging and finding any trouble. (`Alu_ctrl, Address, Data_out, Acca...`)

3.1. After finalizing the block diagram, in Sources, **right click** the Block design, and select **Generate Output Products**. Then, in the same menu, select **Create HDL Wrapper**. This will write an equivalent of a `main.v` module, that you can assign pins, write a testbench (simulation) for, and ultimately upload to the Spartan 7.

Table 1: Computer Instructions

Op.Code	Instruction	Operation (Mnemonic)
	`nop`	Do nothing. (No Operation)
	`LDDA addr`	Loads `ACCA` with the value in memory at address `addr`. `C` stays the same, `Z` changes. (Load `ACCA` from memory)
	`LDDA_IMM #num`	Loads `ACCA` with `num`, the value in memory at the address immediately following the `LDAA #num` command. `C` stays the same, `Z` changes. (Load `ACCA` with an immediate)
	`STAA addr`	Stores the value in `ACCA` at memory address `addr`. `C` stays the same, Z changes. (Store `ACCA` in memory)
	`ADDA addr`	Adds the value in memory location `addr` to the value in `ACCA` and saves the result in `ACCA`. `C` and `Z` change. (Add `ACCA` and value in memory)
	`SUBA addr`	Subtracts the value in memory location `addr` from the value in `ACCA` and saves the result in `ACCA`. `C` and `Z` change. (Subtract value in memory from `ACCA`)
	`ANDA addr`	Perform a logical `AND` of the value in memory location `addr` with the value in `ACCA`. Save the result in `ACCA`. `C` stays the same, `Z` changes. (Logical `AND` of `ACCA` and value in memory)
	`ORAA addr`	Perform a logical `OR` of the value in memory location `addr` with the value in `ACCA`. Save the result in `ACCA`. `C` stays the same, `Z` changes. (Logical `OR` of `ACCA` and value in memory)
	`CMPA addr`	Compare `ACCA` to value in `addr`. This is done by subtracting the value in `addr` from `ACCA`. `ACCA` does not change. `C` and `Z` change. (Compares `ACCA` to the value in `addr`)
	`COMA`	Replace the value in `ACCA` with its one's complement. `C` is set to 1 and `Z` changes. (Compliment `ACCA`)
	`INCA`	Increment value in `ACCA`. `C` stays the same and `Z` changes. (`INCA ACCA`)
	`LSLA`	Logical shift left of `ACCA`. `C` and `Z` change. (Logical shift left `ACCA`)
	`LSRA`	Logical shift right of `ACCA`. `C` and `Z` change. (Logical shift right `ACCA`)
	`ASRA`	Arithmetic shift right of `ACCA`. `C` and `Z` change. (Arithmetic shift right `ACCA`)
	`JMP addr`	Jumps to the instruction stored in address `addr`. The `PC` is replaced with `addr`. `C` and `Z` stay the same. (Jump)
	`JCS addr`	Jumps to the instruction stored in address `addr` if $C = 1$. If `C` is not set, continue with next instruction. `C` and `Z` stay the same. (Jump if carry set)
	`JCC addr`	Jumps to the instruction stored in address `addr` if $C = 0$. If `C` is set, continue with next instruction. `C` and `Z` stay the same. (Jump if carry not set)
	`JEQ addr`	Jumps to the instruction stored in address `addr` if $Z = 1$. If `Z` is not set, continue with next instruction. `C` and `Z` stay the same. (Jump if `Z` set)