# *Lab 9: Build a Computer*

## Introduction

A conceptual block diagram of a simple computer is shown in Figure 1. In previous labs the `Addr_Mux`, the `ALU`, the Control Unit (`CCU`) and the required registers were already designed. In this lab all the components will be put together to build a computer. The only missing block is the memory block which can be found here. Also needed is, the mem_init.v file in which program code is to be inserted. Use graphical design to implement the computer as shown in Figure 1. Instructions on how to do such are provided.

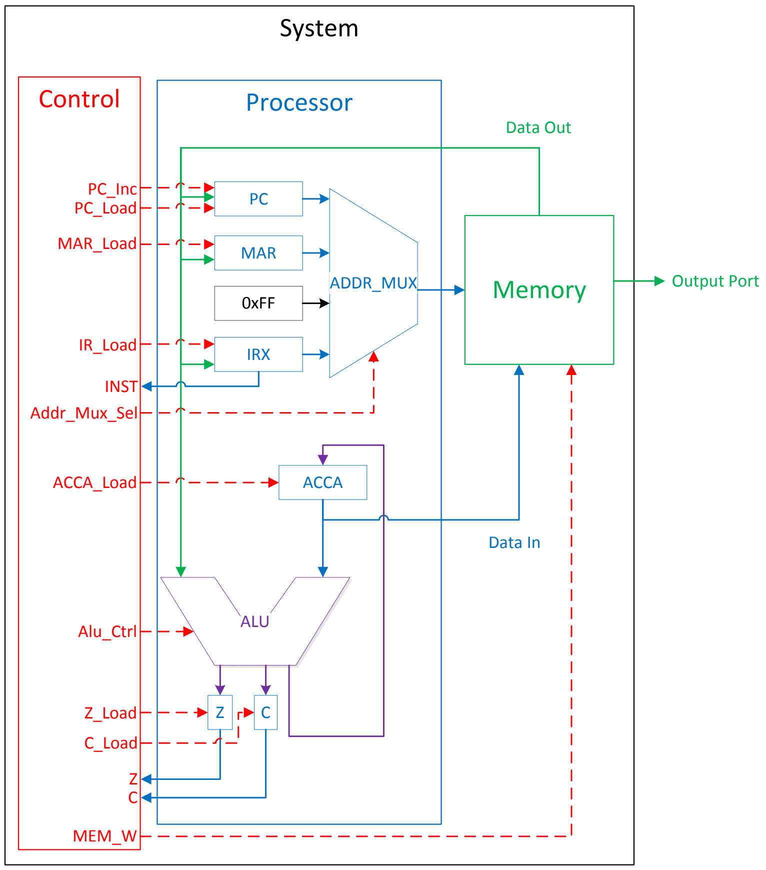
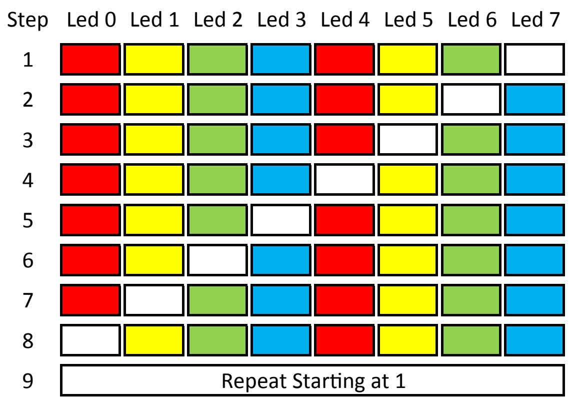


Figure 1: Processor Block Diagram

## 1 Prelab

1.1. Using the instruction set provided in the Table 1, write a computer program to generate running lights. One way to accomplish this is to start with an 8-bit number `0000 0001` (where the one represents the LED that would be off). Then left shift that number; once you have reached the end, jump back to the beginning of the program. Figure 2 shows the expected output at different time steps.

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



| Step | Led 0 | Led 1 | Led 2 | Led 3 | Led 4 | Led 5 | Led 6 | Led 7 |
|---|---|---|---|---|---|---|---|---|
| 1 | 🟥 | 🟨 | 🟩 | 🟦 | 🟥 | 🟨 | 🟩 | ⬜ |
| 2 | 🟥 | 🟨 | 🟩 | 🟦 | 🟥 | 🟨 | ⬜ | 🟦 |
| 3 | 🟥 | 🟨 | 🟩 | 🟦 | 🟥 | ⬜ | 🟩 | 🟦 |
| 4 | 🟥 | 🟨 | 🟩 | 🟦 | ⬜ | 🟨 | 🟩 | 🟦 |
| 5 | 🟥 | 🟨 | 🟩 | ⬜ | 🟥 | 🟨 | 🟩 | 🟦 |
| 6 | 🟥 | 🟨 | ⬜ | 🟦 | 🟥 | 🟨 | 🟩 | 🟦 |
| 7 | 🟥 | ⬜ | 🟩 | 🟦 | 🟥 | 🟨 | 🟩 | 🟦 |
| 8 | ⬜ | 🟨 | 🟩 | 🟦 | 🟥 | 🟨 | 🟩 | 🟦 |
| 9 | Repeat Starting at 1 | | | | | | | |

Figure 2: Running Lights LED Sequence

## 2 Lab

2.1. Enter your running light program into the mem_init.v file.

2.2. Simulate the computer you have created in the prelab.

2.3. Run your code on your board.

## 3 Supplement: Graphical Design of Processor

3.1. Start a new project and include the mem_block.v.

3.2. Include the other modules you need for this lab (e.g. the `ALU`.)

3.3. Create graphical symbols for each module

3.3.1. While in the "implementation" view, and on the "Design" tab, click on your desired module (Listed in the "Hierarchy" panel).

3.3.2. Then, In the "Processes" panel, below, expand the "Design Utilities" tree.

3.3.3. Under this tree is the "Create Schematic Symbol" menu item, double-click it.

3.3.4. **Only** the first module will generate a symbol, but all the subsequent modules will show as created. Select the missing module(s) and right-click "Create Schematic Symbol" and choose "ReRun."

3.4. Create a new file, as you would create a new Verilog file, and choose "Schematic." Once the file is created, the Schematic editor should load.

3.5. Navigate the left-panel to the "Symbols" tab, and choose the category labelled as your project directory.

3.6. Click on modules as needed, then place them in the schematic by clicking where you want them.

3.7. When ready, find the "Add wire" tool in the vertical toolbar and connect the modules.

## Appendix

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) |