## Index

## Introduction

For this project, I created a general-purpose microprocessor. This project aimed to create a functional CPU different from the i281 CPU in Design and Instruction Architecture. The most important parts of the design are the Instruction Memory, Data Memory, Main Registers, Arithmetic and Logic Unit, Program Counter, and Control unit. My CPU's v1 can do most commands of a normal computer, but it is limited by instruction space and memory width.

## Abbreviations

| | | | |
|---|---|---|---|
| IMEM | Instruction Memory | DMEM | Data Memory |
| MReg | Main Registers | IX | Index Register |
| ALU | Arithmetic and Logic Unit | PC | Program Counter |

## Decoding the Instruction

One instruction is 16 bits divided into four parts, with the first 3 making up the instruction and the last part as the data. E.g., 0001_00_11_01111111. The first 4 bits specify the opcode or instruction type. The next two bits specify the addressing mode (Immediate, Direct, Indirect, or Indexed). The last two bits of the instruction specify which of the four registers in MReg will be used (A, B, C, or IX).

## OPCODEs

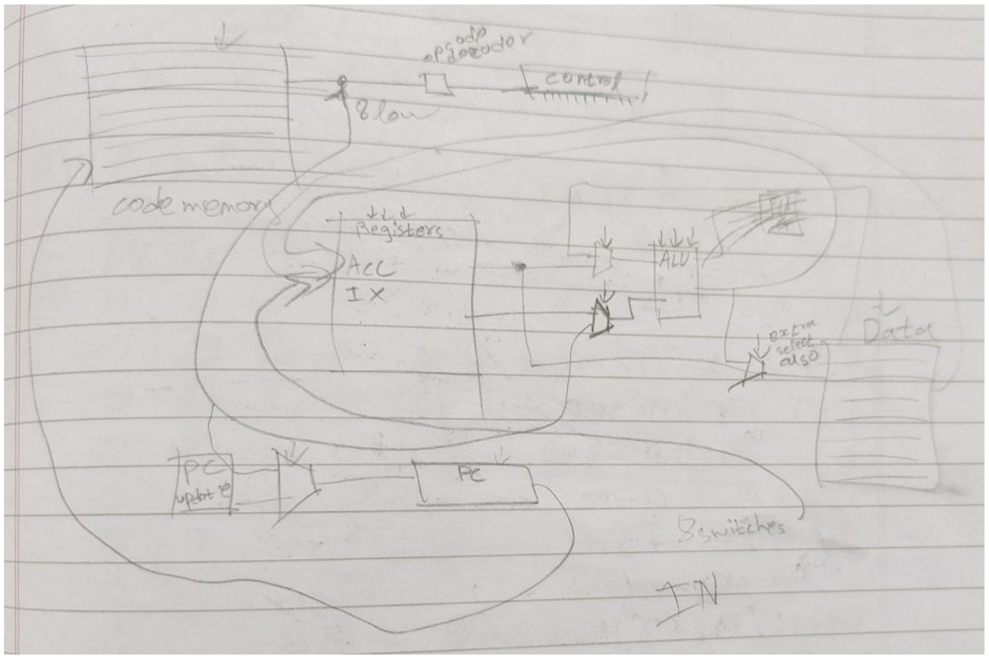
1. NOOP
    a. Description and Use: NO Operation. Used as the first command to allow the program to load. Also used to fill IMEM after the END
    b. In Assembly: 0000dddddddddddd
2. LDM
    a. Description and Use: LoaD value specified in the instruction
    b. In Assembly: 000100dddddddddd
3. LDD
    a. Description and Use: LoaD Data value specified in DMEM. Address provided in instruction data
    b. In Assembly: 000101dddddddddd
4. LDX
    a. Description and Use: LoaD a value from DMEM. Get the address by adding IX value to the value specified in the instruction
    b. In Assembly: 000111dddddddddd
5. STO
    a. Description and Use: STOre data in DMEM at the address provided in the instruction data
    b. In Assembly: 001001dddddddddd
6. STX
    a. Description and Use: STore a value in DMEM. Get the address by adding IX value to the value specified in the instruction
    b. In Assembly: 001011dddddddddd
7. ADD
    a. Description and Use: ADD data from MReg to data from the instruction
    b. In Assembly: 001100dddddddddd

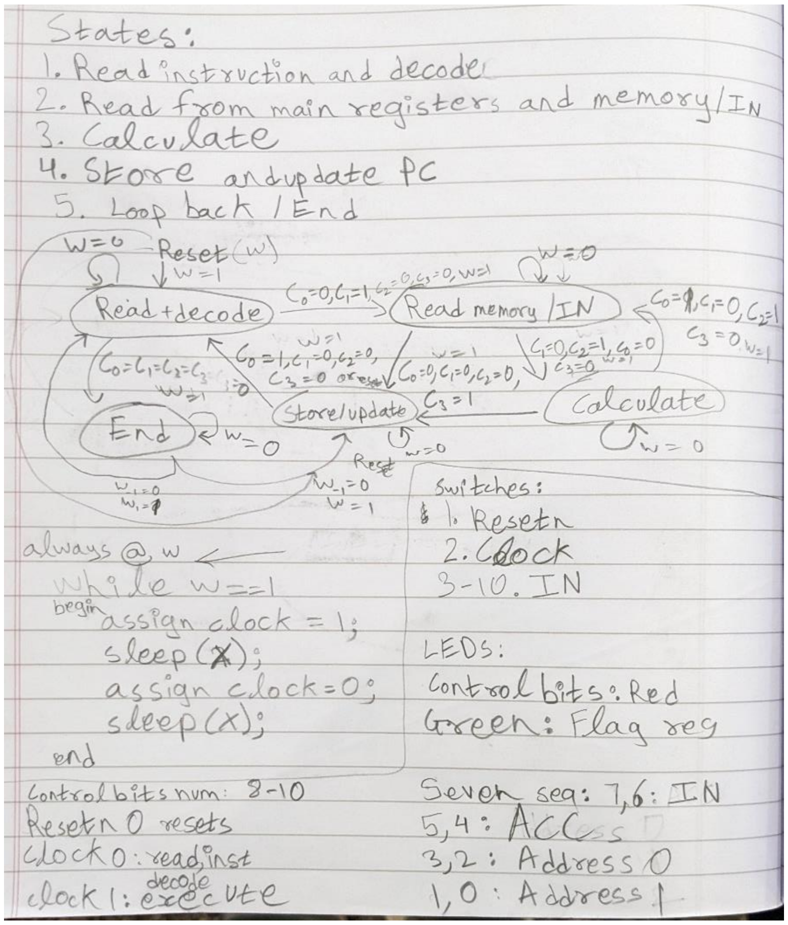
8. ADD
    a. Description and Use: ADD data from MReg to data at the address provided in the instruction data
    b. In Assembly: 001101dddddddddd
9. SUB
    a. Description and Use: SUBtract data in MReg from data from the instruction
    b. In Assembly: 010000dddddddddd
10. SUB
    a. Description and Use SUBtract data in MReg from data at the address provided in the instruction data
    b. In Assembly: 010001dddddddddd
11. INC
    a. Description and Use: INCrement the value of specified MReg by 1
    b. In Assembly: 0101dddddddddddd
12. MOV
    a. Description and Use: MOVe data from MReg A to specified MReg
    b. In Assembly: 0110dddddddddddd
13. IN
    a. Description and Use: Input a value from the board
    b. In Assembly: 0111dddddddddddd
14. OUT
    a. Description and Use: OUTput the decimal value of one MReg onto the LCD Display
    b. In Assembly: 1000dddddddddddd
15. CMP
    a. Description and Use: CoMPare data in one MReg to the value given in the instruction data
    b. In Assembly: 100100dddddddddd
16. CMD
    a. Description and Use: CoMPare data in one MReg to the data at the address provided in the instruction data
    b. In Assembly: 100101dddddddddd
17. JMP
    a. Description and Use: JuMP to the instruction address provided in the instruction data
    b. In Assembly: 1010dddddddddddd
18. JPN
    a. Description and Use: JumP to the instruction address provided in the instruction data if Not equal
    b. In Assembly: 10110ddddddddddd
19. JPE
    a. Description and Use: JumP to the instruction address provided in the instruction data if Equal
    b. In Assembly: 10111ddddddddddd
20. AND
    a. Description and Use: Bitwise AND of data from one MReg with the value in instruction data
    b. In Assembly: 110000dddddddddd

21. AND
    a. Description and Use: Bitwise AND of data from one MReg with the data at the address provided in the instruction data
    b. In Assembly: 110001dddddddddd
22. OR
    a. Description and Use: Bitwise OR of data from one MReg with the value in instruction data
    b. In Assembly: 110100dddddddddd
23. OR
    a. Description and Use: Bitwise OR of data from one MReg with the data at the address provided in the instruction data
    b. In Assembly: 110101dddddddddd
24. XOR
    a. Description and Use: Bitwise XOR of data from one MReg with the value in instruction data
    b. In Assembly: 111000dddddddddd
25. XOR
    a. Description and Use: Bitwise XOR of data from one MReg with the data at the address provided in the instruction data
    b. In Assembly: 111001dddddddddd
26. END
    a. Description and Use: END execution
    b. In Assembly: 1111dddddddddddd
27. Instructions currently not implemented but possible with the current Instruction set:
    a. LDI
    b. STI
    c. ADDI
    d. ADDX
    e. SUBI
    f. SUBX
    g. CMI
    h. CMX
    i. JPGT
    j. JPGE
    k. ANDI
    l. ANDX
    m. ORI
    n. ORX
    o. XORI
    p. XORX

**Design progress**

Early Concepts:





States:

1. Read instruction and decode
2. Read from main registers and memory/IN
3. Calculate
4. Store and update PC
5. Loop back / End

w=0
Reset (w)
w=1
Read + decode
$C_0=0, C_1=1, C_2=0, C_3=0, w=1$
w=0
Read memory / IN
$C_0=1, C_1=0, C_2=1$
$C_3=0$ w=1
$C_0=C_1=C_2=C_3=0$
w=1
$C_0=1, C_1=0, C_2=0,$
$C_3=0$ or reset
w=1
w=1
$C_0=0, C_1=0, C_2=0,$
$C_1=0, C_2=1, C_0=0$
$C_3=0$ w=1
End
w=0
Store/update
$C_3=1$
Calculate
w=0
w=0
Reset
$w_{-1}=0$
w=1
$w_{-1}=0$
$w_1=1$

```
always @ w
while w==1
begin assign clock = 1;
      sleep(X);
      assign clock=0;
      sleep(X);
end
```

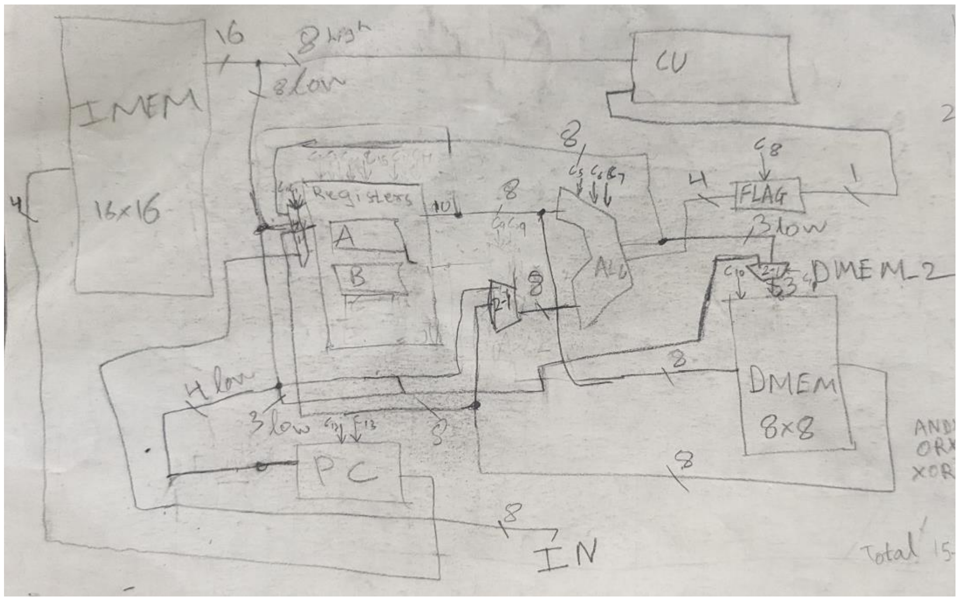
Switches:
1. Resetn
2. Clock
3-10. IN

LEDs:
Control bits: Red
Green: Flag reg

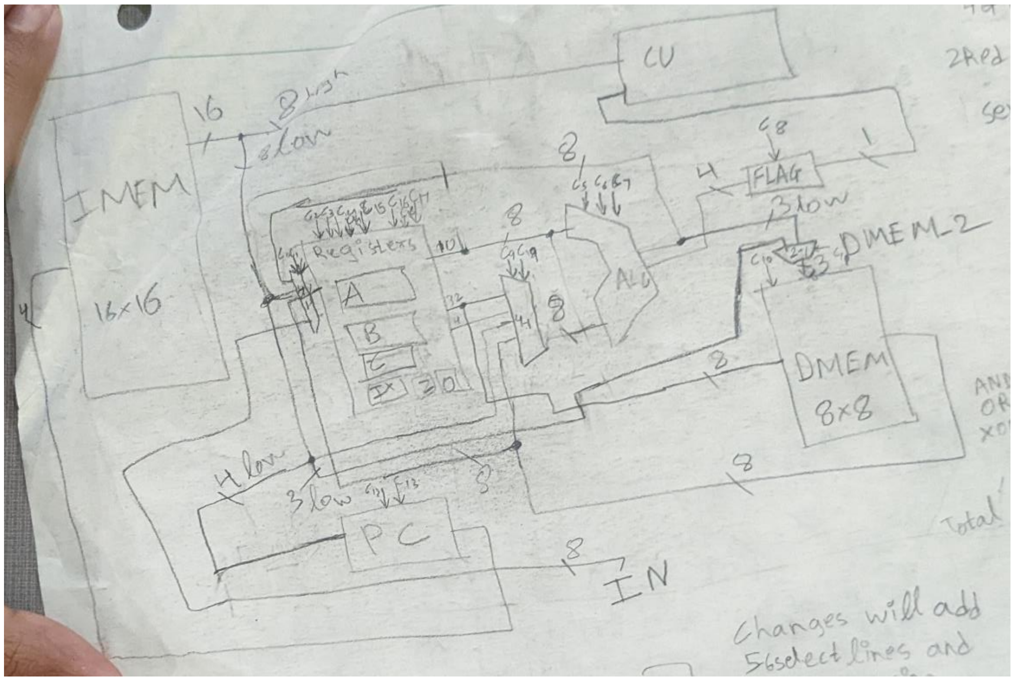
Control bits num: 8-10
Resetn 0 resets
Clock 0: read inst
Clock 1: decode execute

Seven seg: 7,6: IN
5,4: ACC
3,2: Address 0
1,0: Address 1

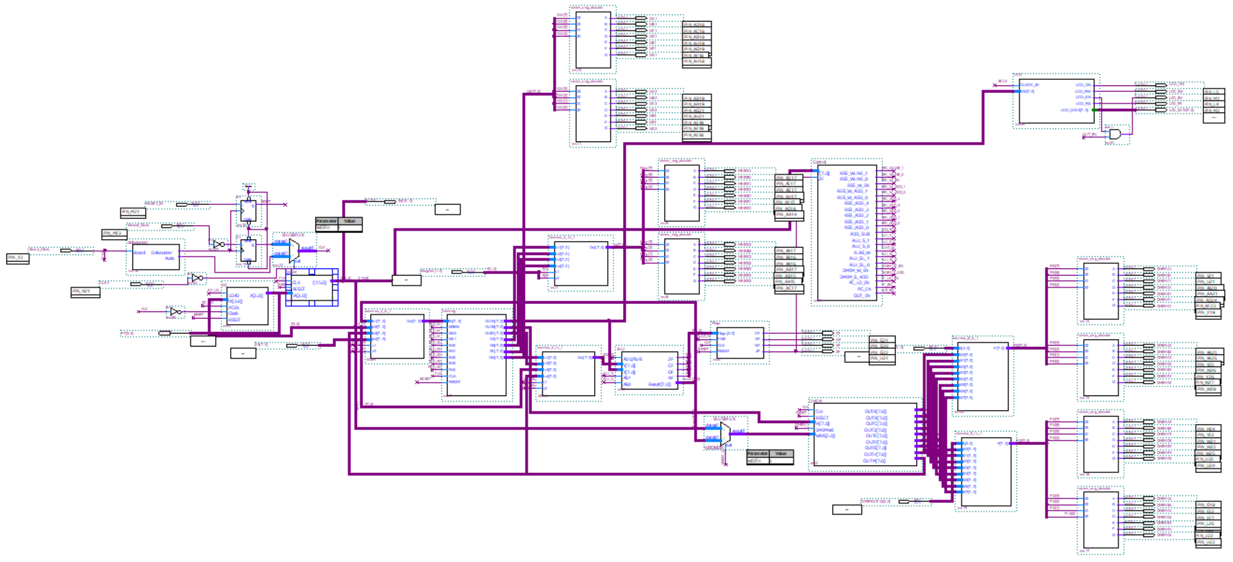
Initial Design:



Second Design:



Final Design:



**Design Phases:**

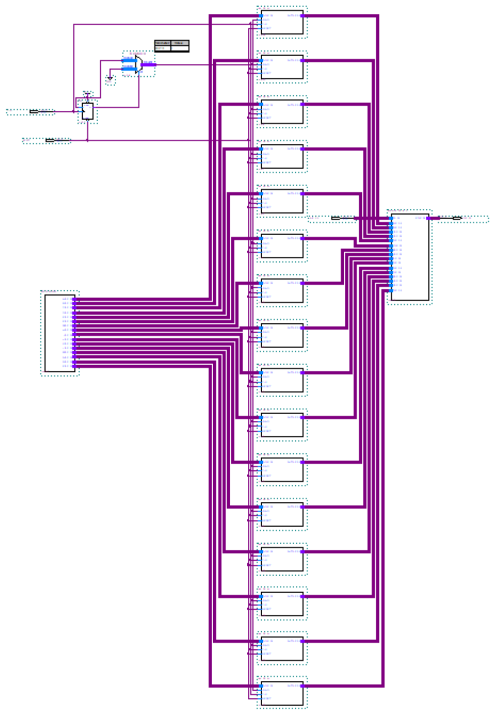
1. ALU creation
   I started with creating my ALU. Since I knew my ALU would have to allow a control system to run, I designed it to have three 8-bit bitwise modules for AND, OR, and XOR, respectively. I also created an 8-bit Carry Look-Ahead (CLA) adder for addition and subtraction. I put everything together with a 4-to-1 multiplexor, five inputs, and five outputs.
2. Early design concepts
   I created a rough design for the CPU using inspiration from the i281 CPU, making slight changes according to the instruction set of my CPU. I created a state diagram that extended one instruction over several clock cycles and started to look at my I/O assignments.
3. First design
   Once I clarified the overall specifications, I started creating a concrete design for the CPU. I added the memory registers (IMEM, MReg, and DMEM) and other required components and wires.
4. Modifications for additional functionality
   I further refined the CPU and added extra control bits, registers, and wires so that certain important instructions could be run on the CPU.
5. Memory
   Using instructions given in Lab 12, I created 16-bit and 8-bit registers for my purpose. I also created modules to output 8 bits of all ones and all zeros. Using the 16-bit registers, I created the IMEM, and I created a module to input program memory in it. Similarly, I designed DMEM using 8-bit registers and a data memory file. I used the register file instructions in Lab 12 to create the MReg.
6. Assembly
   I wired all the components per my modified diagram and added the inputs.
7. Outputs
   I added the outputs using the LEDs and seven-segment displays. A total of 24 LEDs and eight seven-segment displays were used.

8. Control
   To create the control module, I started by evaluating the number of control bits required and then created an excel file to map out the bits against the 8 bits of the opcodes. Following this, I analyzed each control bit separately and optimized them to create logic functions for each of them. This led to the creation of the module with 88 logic units.
9. LCD
   I found a few files online to run the LCD. I adapted those files to my purpose, added them to my project, and updated my control module.

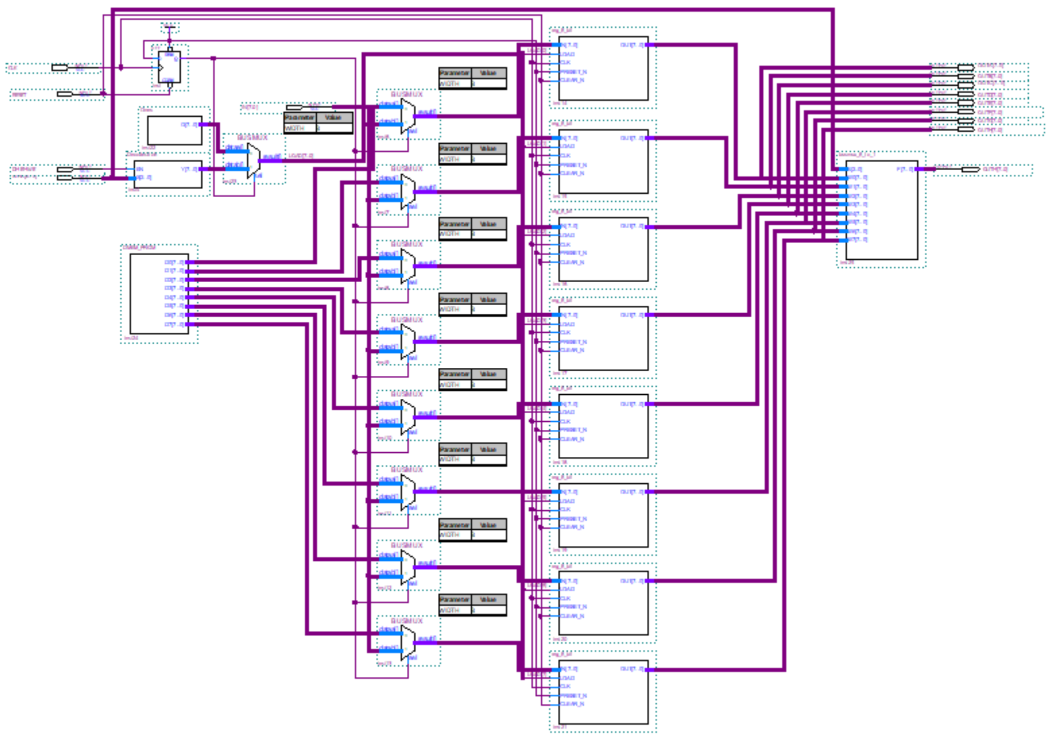
**Design Components**

High Level:

IMEM:



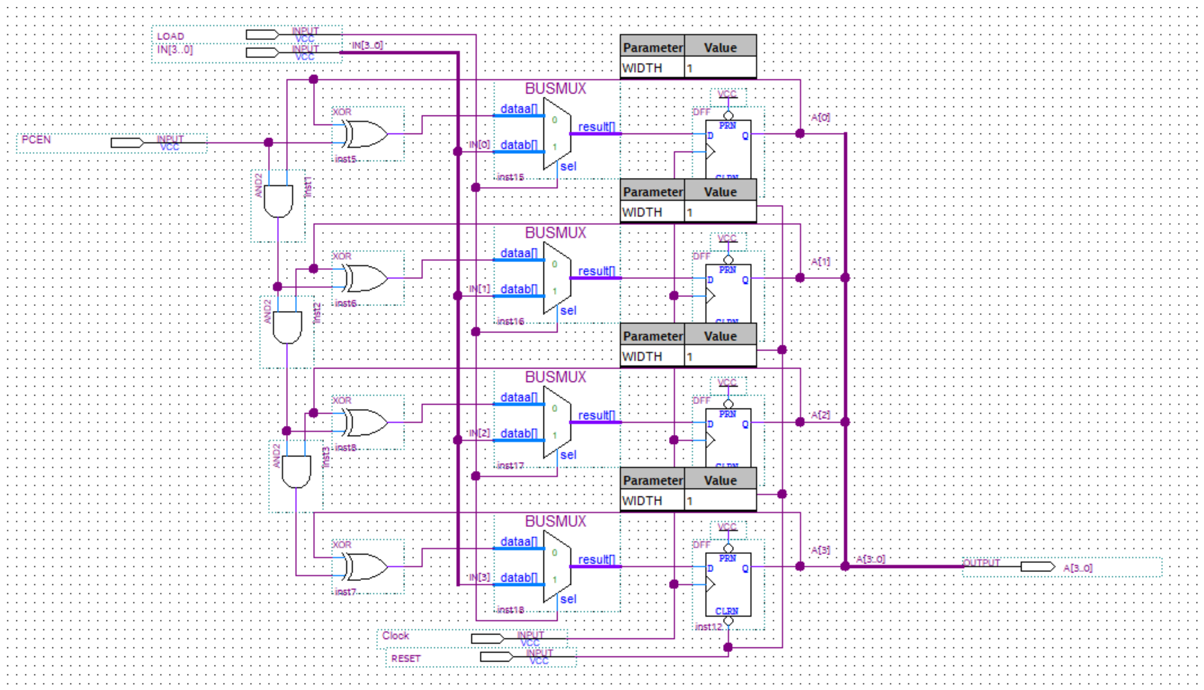
DMEM:



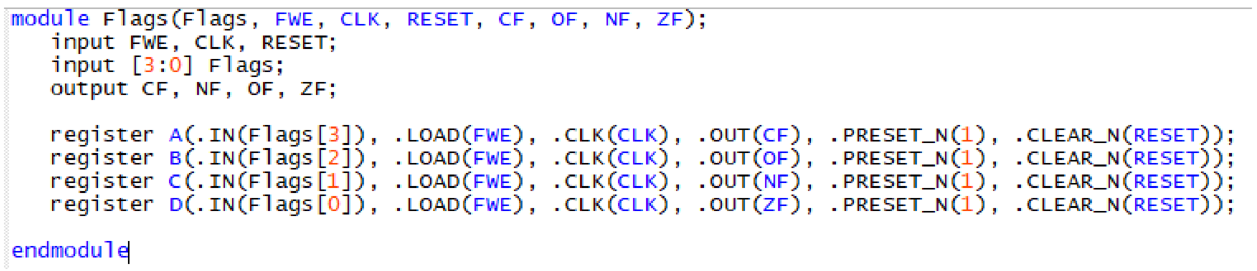
MReg:

```
module mainreg(IN, MRWE, WA0, WA1, RA0, RA1, RA2, RA3, RA4, CLK, RESET, OUTA, OUTB, OUTC, OA, OB, OC, OIX);
   input MRWE, WA0, WA1, RA0, RA1, RA2, RA3, RA4, CLK, RESET;
   input [7:0] IN;
   output [7:0] OUTA, OUTB, OUTC, OA, OB, OC, OIX;

   wire [7:0] OZ, OO;

   Decoder2to4 decoder(.S1(WA1), .S0(WA0), .EN(MRWE), .Y0(Y0), .Y1(Y1), .Y2(Y2), .Y3(Y3));

   reg_8_bit A(.IN(IN), .LOAD(Y0), .CLK(CLK), .OUT(OA), .PRESET_N(1), .CLEAR_N(RESET));
   reg_8_bit B(.IN(IN), .LOAD(Y1), .CLK(CLK), .OUT(OB), .PRESET_N(1), .CLEAR_N(RESET));
   reg_8_bit C(.IN(IN), .LOAD(Y2), .CLK(CLK), .OUT(OC), .PRESET_N(1), .CLEAR_N(RESET));
   reg_8_bit IX(.IN(IN), .LOAD(Y3), .CLK(CLK), .OUT(OIX), .PRESET_N(1), .CLEAR_N(RESET));
   Zeros Z(.Z(OZ));
   Ones O(.O(OO));

   busmux_4_to_1 opa(.In1(OA), .In2(OB), .In3(OC), .In4(OIX), .s1(RA1), .s0(RA0), .Out(OUTA));
   busmux_4_to_1 opb(.In1(OA), .In2(OB), .In3(OC), .In4(OIX), .s1(RA3), .s0(RA2), .Out(OUTB));
   busmux_4_to_1 opc(.In1(OZ), .In2(OO), .In3(OZ), .In4(OO), .s1(1), .s0(RA4), .Out(OUTC));
endmodule
```

PC:

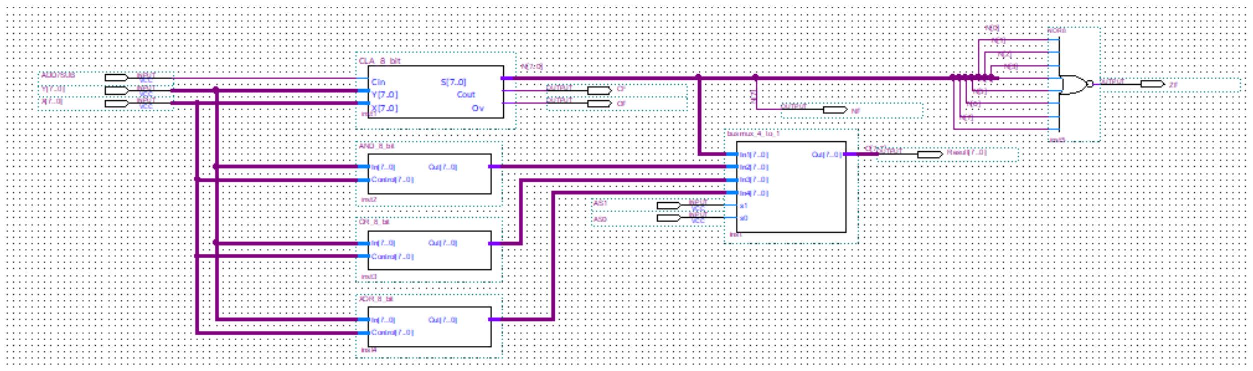


Flags:

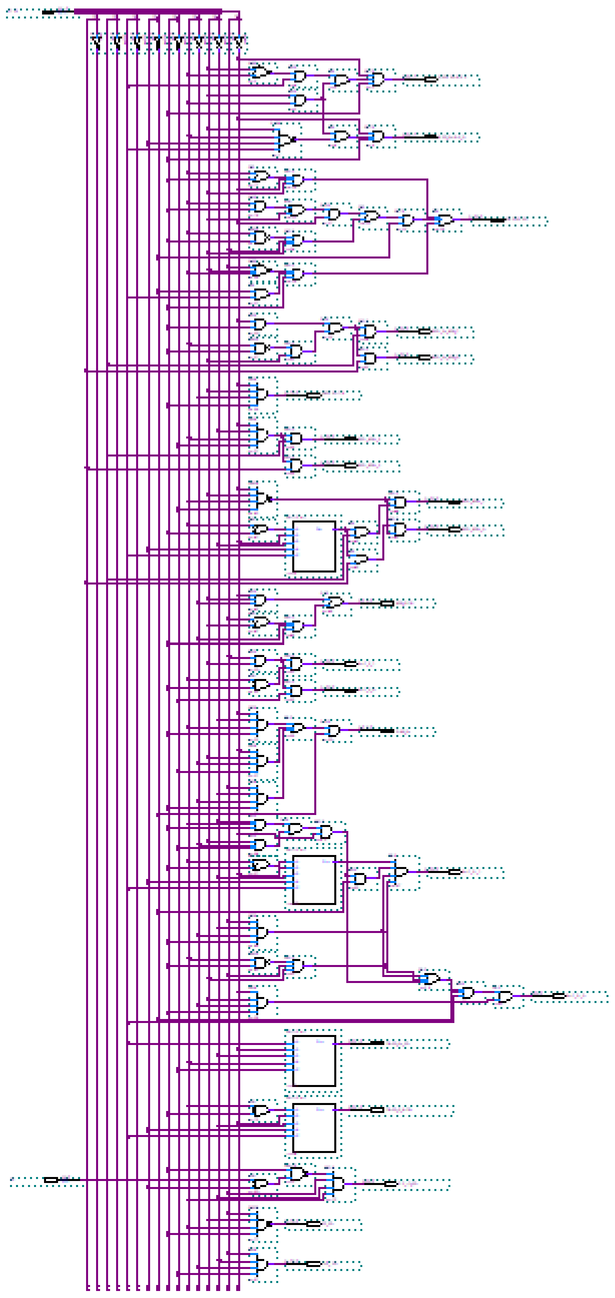


```
module Flags(Flags, FWE, CLK, RESET, CF, OF, NF, ZF);
   input FWE, CLK, RESET;
   input [3:0] Flags;
   output CF, NF, OF, ZF;

   register A(.IN(Flags[3]), .LOAD(FWE), .CLK(CLK), .OUT(CF), .PRESET_N(1), .CLEAR_N(RESET));
   register B(.IN(Flags[2]), .LOAD(FWE), .CLK(CLK), .OUT(OF), .PRESET_N(1), .CLEAR_N(RESET));
   register C(.IN(Flags[1]), .LOAD(FWE), .CLK(CLK), .OUT(NF), .PRESET_N(1), .CLEAR_N(RESET));
   register D(.IN(Flags[0]), .LOAD(FWE), .CLK(CLK), .OUT(ZF), .PRESET_N(1), .CLEAR_N(RESET));

endmodule
```

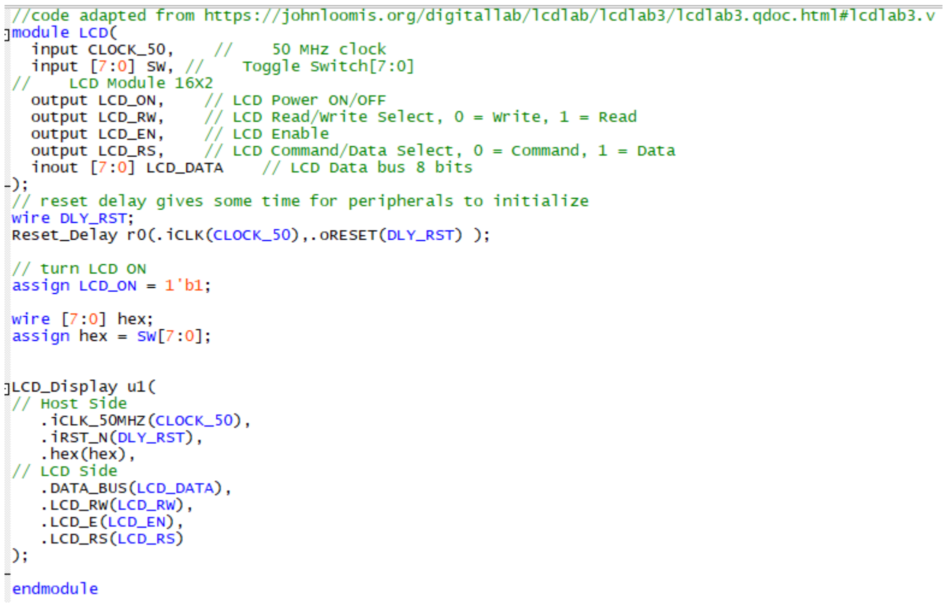
ALU:



Control:

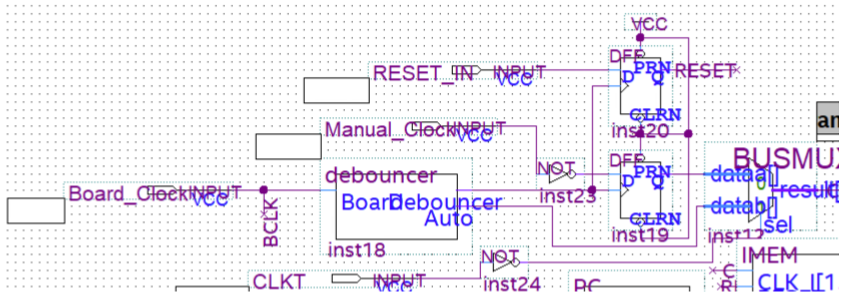


LCD:



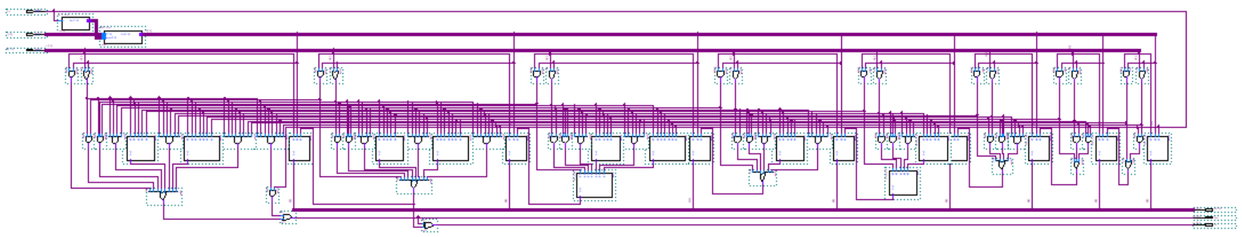
```
//code adapted from https://johnloomis.org/digitallab/lcdlab/lcdlab3/lcdlab3.qdoc.html#lcdlab3.v
module LCD(
  input CLOCK_50,    //    50 MHz clock
  input [7:0] SW, //    Toggle Switch[7:0]
//    LCD Module 16X2
  output LCD_ON,    // LCD Power ON/OFF
  output LCD_RW,    // LCD Read/Write Select, 0 = Write, 1 = Read
  output LCD_EN,    // LCD Enable
  output LCD_RS,    // LCD Command/Data Select, 0 = Command, 1 = Data
  inout [7:0] LCD_DATA    // LCD Data bus 8 bits
);
// reset delay gives some time for peripherals to initialize
wire DLY_RST;
Reset_Delay r0(.iCLK(CLOCK_50),.oRESET(DLY_RST) );

// turn LCD ON
assign LCD_ON = 1'b1;

wire [7:0] hex;
assign hex = SW[7:0];


LCD_Display u1(
// Host Side
   .iCLK_50MHZ(CLOCK_50),
   .iRST_N(DLY_RST),
   .hex(hex),
// LCD Side
   .DATA_BUS(LCD_DATA),
   .LCD_RW(LCD_RW),
   .LCD_E(LCD_EN),
   .LCD_RS(LCD_RS)
);

endmodule
```

Clock and reset:



Low Level:

8-bit CLA:

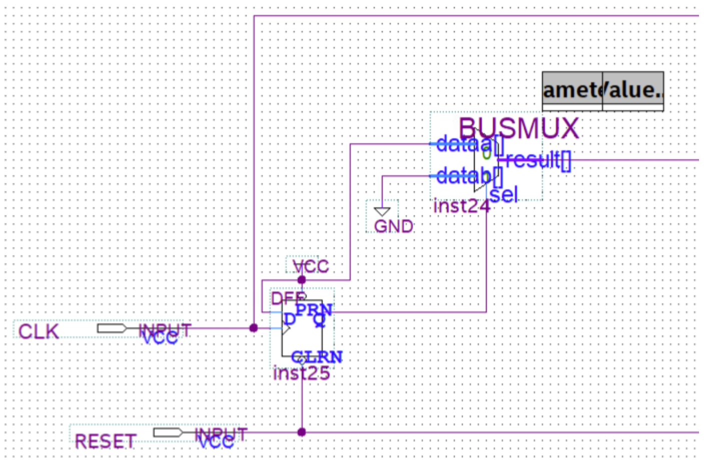


8-bit bitwise AND:

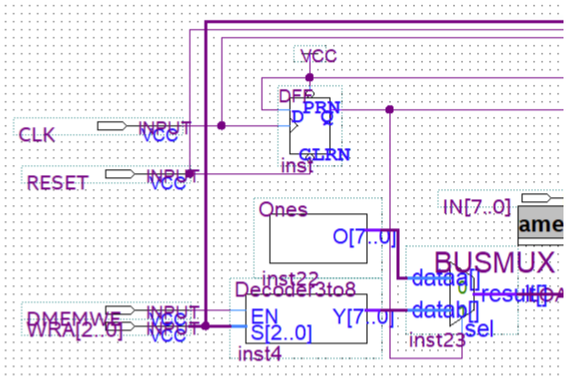
```
module AND_8_bit(In, Control, Out);
   integer k;
   input [7:0] In;
   input [7:0] Control;
   output reg [7:0] Out;

   always @*
   begin
      for(k = 0; k <= 7; k = k + 1)
      begin
         Out[k] = In[k] & Control[k];
      end
   end
endmodule
```

The 8-bit bitwise OR and 8-bit bitwise XOR have been built similarly.

IMEM reset loop:



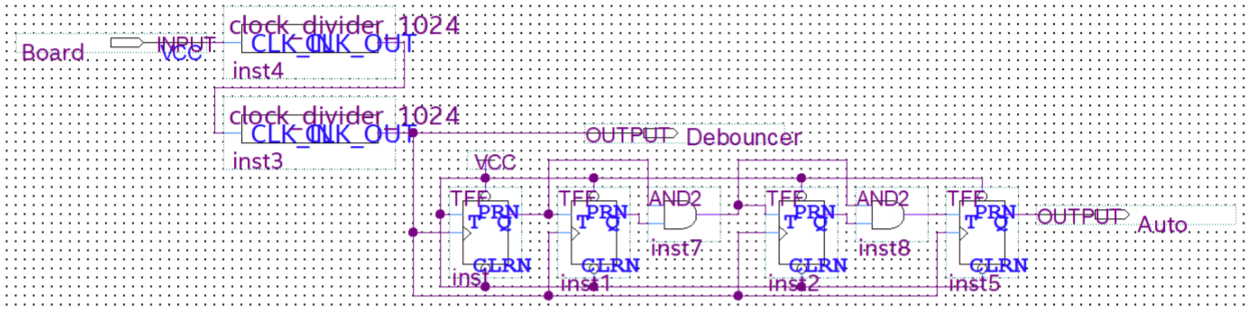
DMEM reset loop:



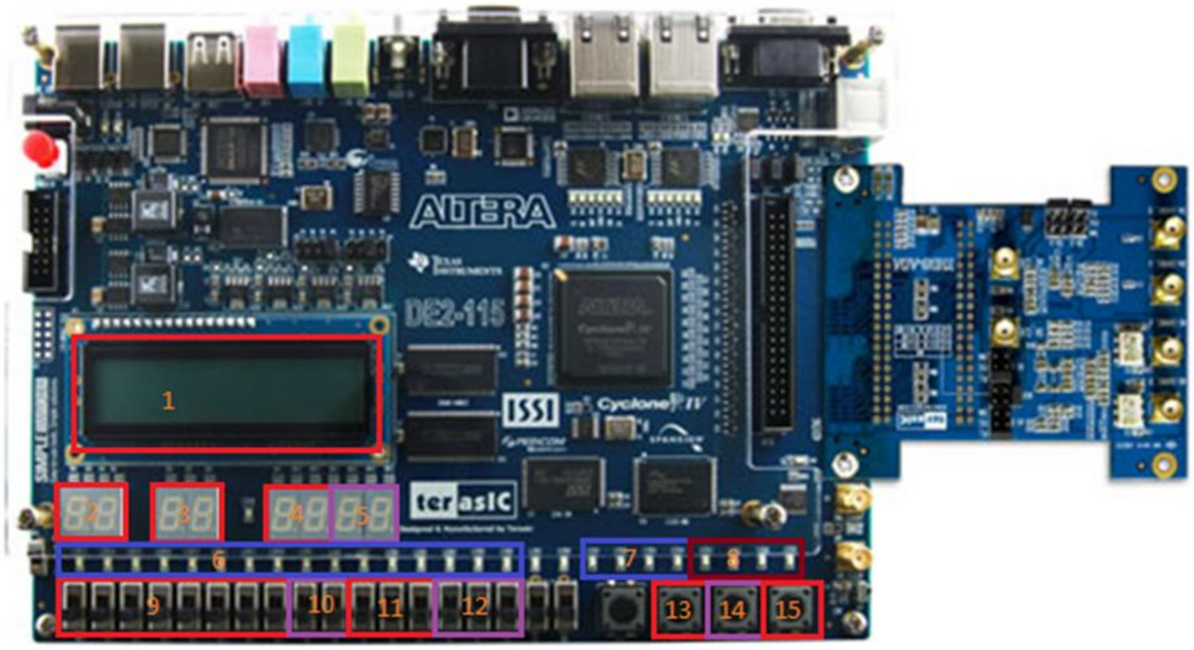
Binary Coded Decimal converter:

This module is unlike the conventional BCDs. From the start, it considers the sign bit. Since this module is only used with the LCD module, it produces hexadecimal ASCII codes representing a positive or negative sign, the required numbers, and spaces. Due to its custom design, it only outputs the necessary digits. For example, $FF_{16}$ will translate to $-1_{10}$ instead of the conventional $255_{10}$ or $-001_{10}$.

Debouncer and Auto clock:



**I/O placements**



| | | | | | |
|---|---|---|---|---|---|
| **1** | **LCD Display** | **6** | **Instruction** | **11** | **DMEM32 Select** |
| **2** | **MRegOut** | **7** | **FLAGs (CF, NF, OF, ZF)** | **12** | **DMEM10 Select** |
| **3** | **IX** | **8** | **Program Counter** | **13** | **Auto Clock** |
| **4** | **DMEM32** | **9** | **Input** | **14** | **Reset** |
| **5** | **DMEM10** | **10** | **MReg Select** | **15** | **Manual Clock** |

**Instructions to run:**

1. Loading the program
    a. Open IMEM_PRGM.v and comment all lines containing assign statements.
    b. Uncomment lines for the program you want to run and save. Program Structure
    c. Open DMEM_PRGM.v and comment all lines containing assign statements.
    d. Uncomment data lines for the program you want to run and save. Data Structure
    e. Compile and run
2. Running the program
    a. On the board, there are two clocks you can use: Manual and Auto. For the Manual clock, press Key0 to go to the next instruction. For Auto clock, hold Key2. The reset is on Key1.
    b. For programs requiring input, use switches SW17 to SW10.
    c. To visualize different main and data registers, the switches SW9 and SW8 can be used for one of the main registers, and the data can be seen on HEX7 and HEX6. For data memory, use switches SW7 to SW5 to control HEX3 and HEX2 and switches SW4 to SW2 to control HEX1 and HEX0. HEX5 and HEX4 visualize the Index Register (Also available on SW9 = 1 and SW8 = 1)
    d. The red LEDs LEDR[17] to LEDR[2] visualize the current Instruction. The green LEDs LEDG[7] to LEDG[4] visualize the Flags register. The green LEDs LEDG[3] to LEDG[0] visualize the Program Counter.
    e. If the program uses the LCD, note that loading a new program or pressing reset will not clear the LCD.

**NOTE:** A recording of each program has been provided in the 'Program Videos' folder

**Problems Faced**

The first problem was faced during the early concepts phase, as the specifications for the CPU were uncertain, and the FSM was too complicated. This was slowly resolved to get the first design, presenting the problem of not supporting some vital instructions. Further developments were made to produce the second design, which supported the instructions. Another problem came with the Control module, which produces 21 bits through various logic functions using 8 bits of input. This was debugged to give a functional circuit diagram. The final problem came with loading data into IMEM and DMEM, and for this, I had to change the loading mechanism and flip the 2-to-1 multiplexor inputs in the memories.

**Lessons Learnt**

I learned a lot about Verilog and microprocessor design during the design phase. This includes the different net types and loops and specifying positive or negative edges of variables. I also learned how to display on the LCD of the DE2-115 board.

**Future Prospects**

1. Assembler v1
2. v2: Add support for multi-clock cycle instructions to run in one clock cycle
3. Assembler v2
4. v3: Increase the number of possible instructions in a 4-bit opcode
5. Assembler v3
6. v4: Increase the Instruction and PC width, and increase IMEM and DMEM length
7. Assembler v4

**Appendices**

1. **Creating your own programs**
   a. Use program structure as a base. A will always be NOOP. Write your program lines by referring to the opcodes. The last instruction of every program is END. If your program is less than 16 lines (including the NOOP), fill the extra lines with the NOOP instruction. Paste the assign statements into IMEM_PRGM.v and save.
   b. Similarly, you can create the data and save it in DMEM_PRGM.v
   c. Compile and run.
   d. For further instructions, look at 'Running the program'
2. **IMEM program structure:**

```
//program name
assign A = 16'b0000000000000000;
assign B = 16'b0001010000000000;
assign C = 16'b1100000000000000;
assign D = 16'b0010010000000000;
assign E = 16'b0001001100000001;
assign F = 16'b0111000000000000;
assign G = 16'b0010110000000000;
assign H = 16'b0001010000000000;
assign I = 16'b1101000000000001;
assign J = 16'b0010010000000000;
assign K = 16'b1101000001111111;
assign L = 16'b0010010000000000;
assign M = 16'b1001000001111111;
assign N = 16'b1011000000001110;
assign O = 16'b1110000010000000;
assign P = 16'b1111000000000000;
```

3. **DMEM program structure:**

```
//program names separated by commas
assign D0 = 8'b00000000;
assign D1 = 8'b00000000;
assign D2 = 8'b00000000;
assign D3 = 8'b00000000;
assign D4 = 8'b00000000;
assign D5 = 8'b00000000;
assign D6 = 8'b00000000;
assign D7 = 8'b00000000;
```