16-Bit Microprocessor Design

CpE 300 - Capstone Project

Nicholas Moya
CpE 300
Prof. Emma Regentova
December 2, 2015

Table of Contents

OVERVIEW

For the Capstone project in Digital System Architecture and Design, I was tasked with designing a SMP (Simple Microprocessor) on Quartus, written in Verilog, that consists of a control unit and data path which are used to operate 16-bit instructions on 8-bit numbers. The SMP has 16 different assembly instructions that it can use to manipulate an 8-bit number and store the result in either the accumulator register or the 64Kx8 SRAM memory module. The SMP is a 16-bit microprocessor because it uses a single 16-bit wide bus architecture for the purposes of moving and storing data. The architecture is explained in more detail in the next section. When the 16-bit instruction in sent to the memory, it is divided into the following form:

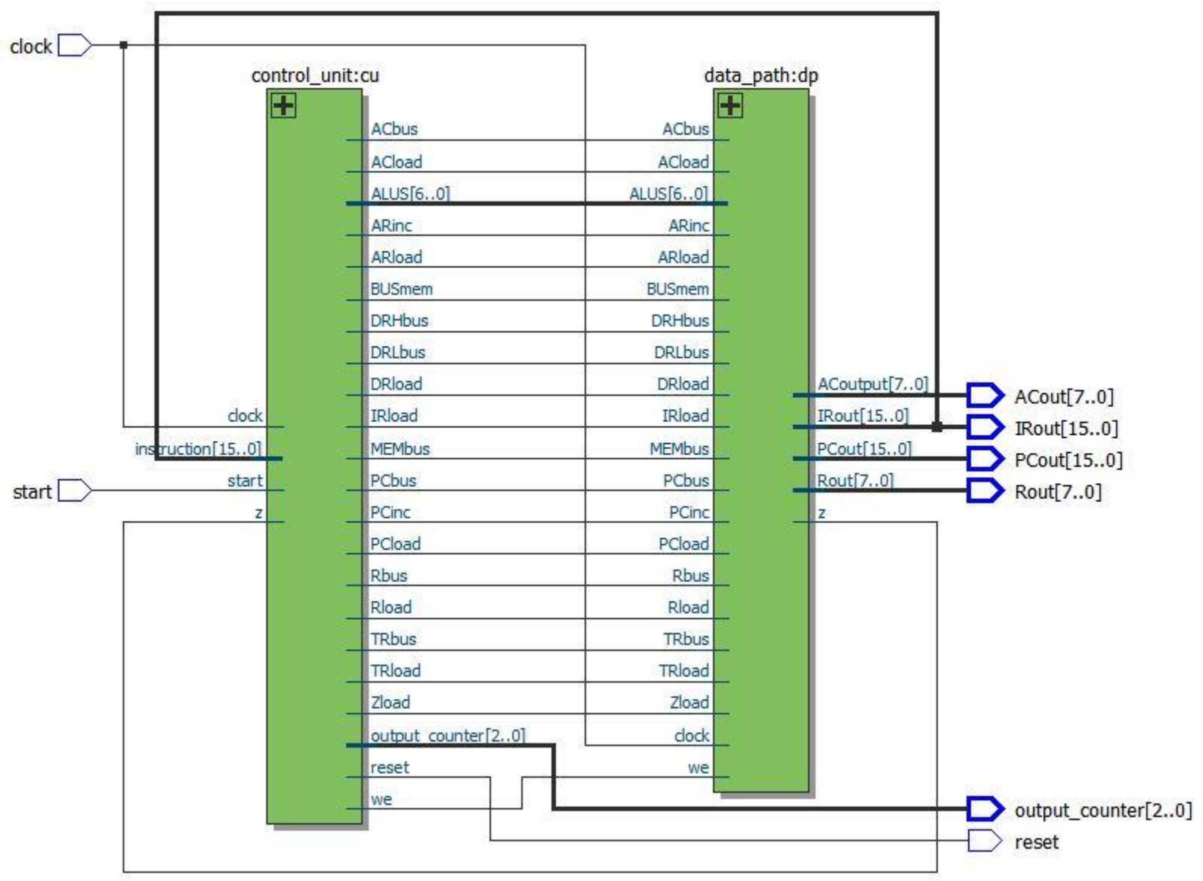
0000	Opcode of Instruction	LSB of Address of stored value	MSB of address of stored value
15 12	11 8	7 4	3 0

Visualization of 16-bit instructions as they are seen in memory

Thus, the SMP is programmed by writing this data (in hex) into the memory addresses, compiling the code, and observing the output using either a waveform file, test bench verification, or by programming the pins on an FPGA. Near the end of this report, I have included a section that details an example of programming the SMP with a set of test assembly instructions and the output is observed on a waveform file.

ARCHITECTURE

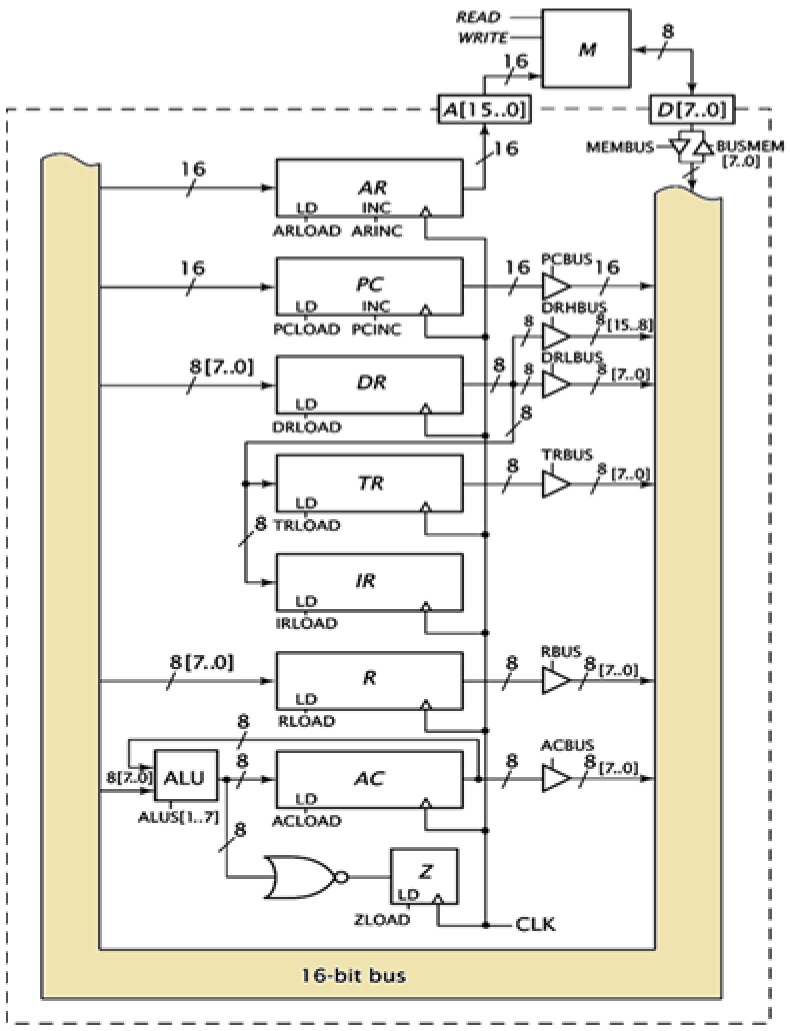
For the SMP to carry out basic assembly instructions, it needs various modules for decoding/decoding instructions, operating on 8-bit numbers, and storing data. Moreover, the various components need to be oriented in a logical way so that the SMP can accomplish instructions; this orientation is called our datapath. The datapath connects the registers, memory and ALU to each other via a 16-bit bus which is used for transferring data to accomplish an instruction. However, the datapath can only perform instructions based on the control signals sent from the control unit. As stated, the control unit sends control signals which are determined by the time cycle and the instruction trying to be performed. Moreover, the control unit is sent the instruction from the data path which it uses to generate the control signals needed to prepare the datapath for the next instruction. Thus, the architecture of this SMP is 16-bit instructions acting on 8-bit numbers which are manipulated in a data bath which itself is controlled by a control unit. All of the components and mechanics that go into the datapath and control unit will be detailed further; this is only an overview of the architecture that when into designing the SMP.



RTL (Register Transfer Level) of the SMP

DATAPATH

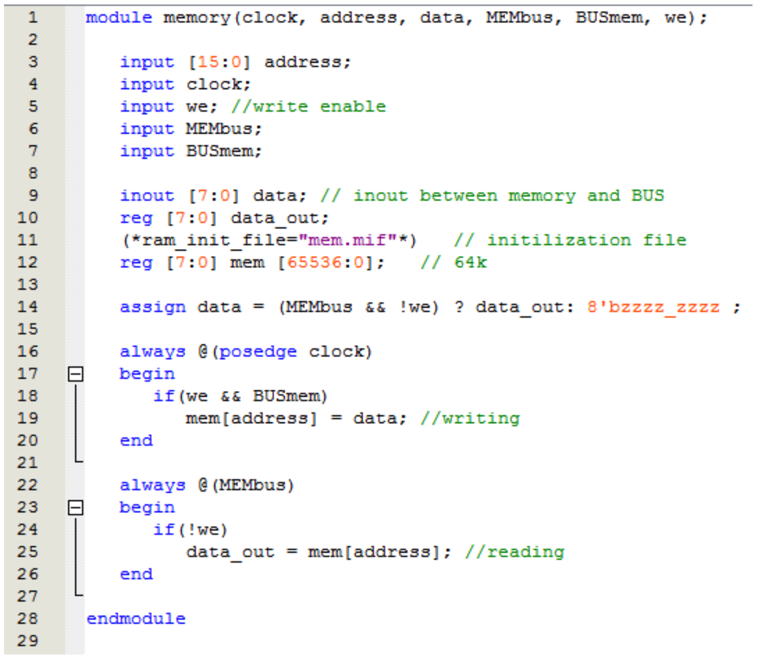
As stated previously, the datapath contains the registers, memory, and ALU and it uses a 16-bit bus to relay data between the various components. The registers are temporary storage units for 8-bit and 16-bit data, the ALU is used for computing arithmetic calculations, and the memory is used to program the SMP binary instruction information. The datapath also contains a latch which acts as a 'zero flag' and initializes to 1 if the AC register contains all zeros and 0 if the AC doesn't contain 0. Lastly, the registers and memory are connected to the bus via tristate buffers which are enabled by control signals sent from the control unit. A picture of the datapath is provided below. Now that the arrangement between the components is apparent, we will detail the individual function of each component.



Datapath of the SMP

MEMORY

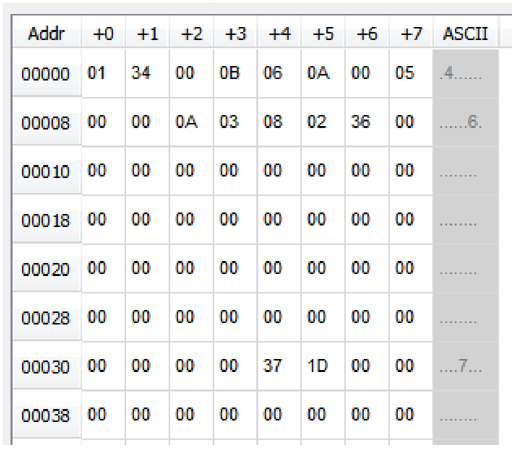
The SMP uses 64kx8 SRAM, meaning that it has 64,000 8-bit addresses that it uses to store assembly instructions in hex which are initialized by saving the instructions as a .mif file in the Quartus text editor. Memory is connected to the bus via two tristate busses and the AR register which contains the address supplied to the memory. The module has three control signals; we, MEMbus, and BUSmem. If we = 1, data is written into the memory address, if MEMbus = 1, data is written into the bus, and if BUSmem = 1, bus is written into the memory address. Below is an example of what the .mif initialized memory looks like and the code for the memory module.



```
module memory(clock, address, data, MEMbus, BUSmem, we);

   input [15:0] address;
   input clock;
   input we; //write enable
   input MEMbus;
   input BUSmem;

   inout [7:0] data; // inout between memory and BUS
   reg [7:0] data_out;
   (*ram_init_file="mem.mif"*)   // initilization file
   reg [7:0] mem [65536:0];   // 64k

   assign data = (MEMbus && !we) ? data_out: 8'bzzzz_zzzz ;

   always @(posedge clock)
   begin
      if(we && BUSmem)
         mem[address] = data; //writing
   end

   always @(MEMbus)
   begin
      if(!we)
         data_out = mem[address]; //reading
   end

endmodule
```

Memory Verilog Module

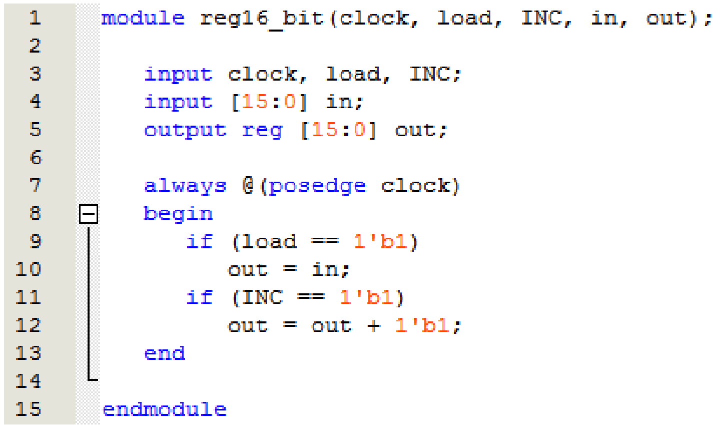
Mem.mif memory initilization file



Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
00000	01	34	00	0B	06	0A	00	05	.4......
00008	00	00	0A	03	08	02	36	00	......6.
00010	00	00	00	00	00	00	00	00	....
00018	00	00	00	00	00	00	00	00	....
00020	00	00	00	00	00	00	00	00	....
00028	00	00	00	00	00	00	00	00	....
00030	00	00	00	00	37	1D	00	00	....7...
00038	00	00	00	00	00	00	00	00	....

AR Register

The AR register is a 16-bit register which, as previously stated, is used to contain the address where the data will be stored in the memory. When the rising edge of the clock is enabled, the stored data is passed when Load = 1, and the stored data is incremented when INC = 1. This register is an initialization of the general 16-bit register module given below.



```
module reg16_bit(clock, load, INC, in, out);

   input clock, load, INC;
   input [15:0] in;
   output reg [15:0] out;

   always @(posedge clock)
   begin
      if (load == 1'b1)
         out = in;
      if (INC == 1'b1)
         out = out + 1'b1;
   end

endmodule
```

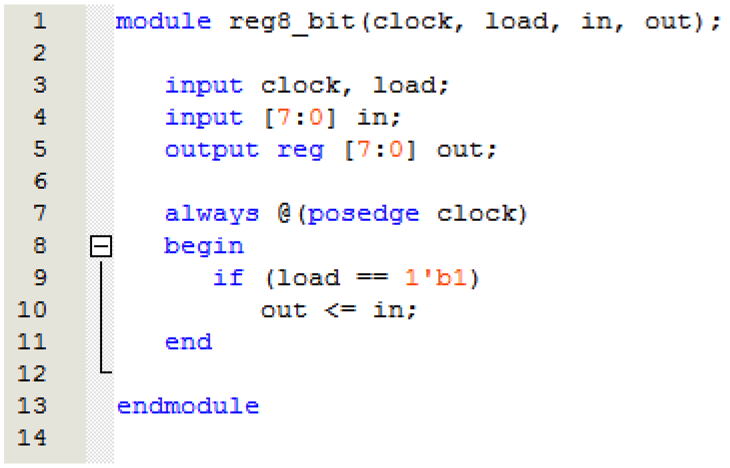
16-bit register module

PC Register

The PC register contains the address of the next instruction to be executed, that is, it contains the address of the next operand if the instruction. Generally, the PC increments after every cycle thus acting as a counter to display the current position in an instruction operation. When the rising edge of the clock is enabled, the stored data is passed when Load = 1, and the stored data is incremented when INC = 1. This register is an initialization of the general 16-bit register module which is previously displayed. In this microprocessor design, I set an output variable equal to the contents of the PC register so that I can observe the current position of an instruction operation.

DR Register

The DR register is an 8-bit register which receives instructions and data from the memory and moves data to the memory via the bus and by separating its data into an 8-bit high stream and an 8-bit low stream. When the rising edge of the clock is enabled, the stored data is passed when Load = 1. This register is an initialization of the general 8-bit register module given below.



```
module reg8_bit(clock, load, in, out);

   input clock, load;
   input [7:0] in;
   output reg [7:0] out;

   always @(posedge clock)
   begin
      if (load == 1'b1)
         out <= in;
   end

endmodule
```

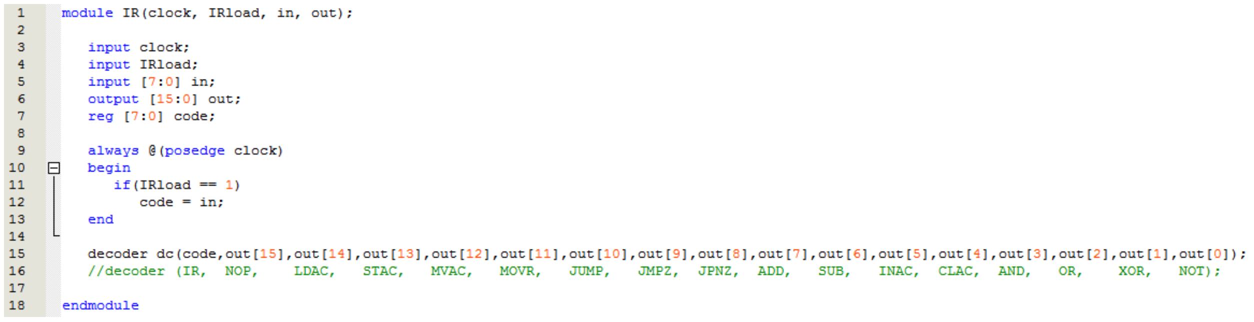
8-bit register module

TR Register

The 8-bit TR register acts as a buffer which temporarily stores data or addresses during instruction execution. When the rising edge of the clock is enabled, the stored data is passed when Load = 1. This register is an initialization of the general 8-bit register module previously displayed.

IR Resister

The IR register loads 8-bit data from the DR register on rising clock edges when Load = 1 and it transfers that data to a decoder module which decodes the data into one of the 16 assembly instructions. It then passes that decoded 16-bit data out to the control unit. The IR module is displayed below.



```
module IR(clock, IRload, in, out);

   input clock;
   input IRload;
   input [7:0] in;
   output [15:0] out;
   reg [7:0] code;

   always @(posedge clock)
   begin
      if(IRload == 1)
         code = in;
   end

   decoder dc(code,out[15],out[14],out[13],out[12],out[11],out[10],out[9],out[8],out[7],out[6],out[5],out[4],out[3],out[2],out[1],out[0]);
   //decoder (IR,  NOP,    LDAC,   STAC,   MVAC,   MOVR,   JUMP,   JMPZ,  JPNZ,  ADD,   SUB,   INAC,  CLAC,  AND,   OR,    XOR,   NOT);

endmodule
```

IR module

R Register

The b-bit R register is a general purpose register that is used to hold nonspecific 8-bit values. This has a similar function to the AC register in that they both are general purpose registers which hold nonspecific data. When the rising edge of the clock is enabled, the stored data is passed when Load = 1. This register is an initialization of the general 8-bit register module previously displayed. In this microprocessor design, I set an output variable equal to the contents of the R register so that I can observe the current contents of the R register.

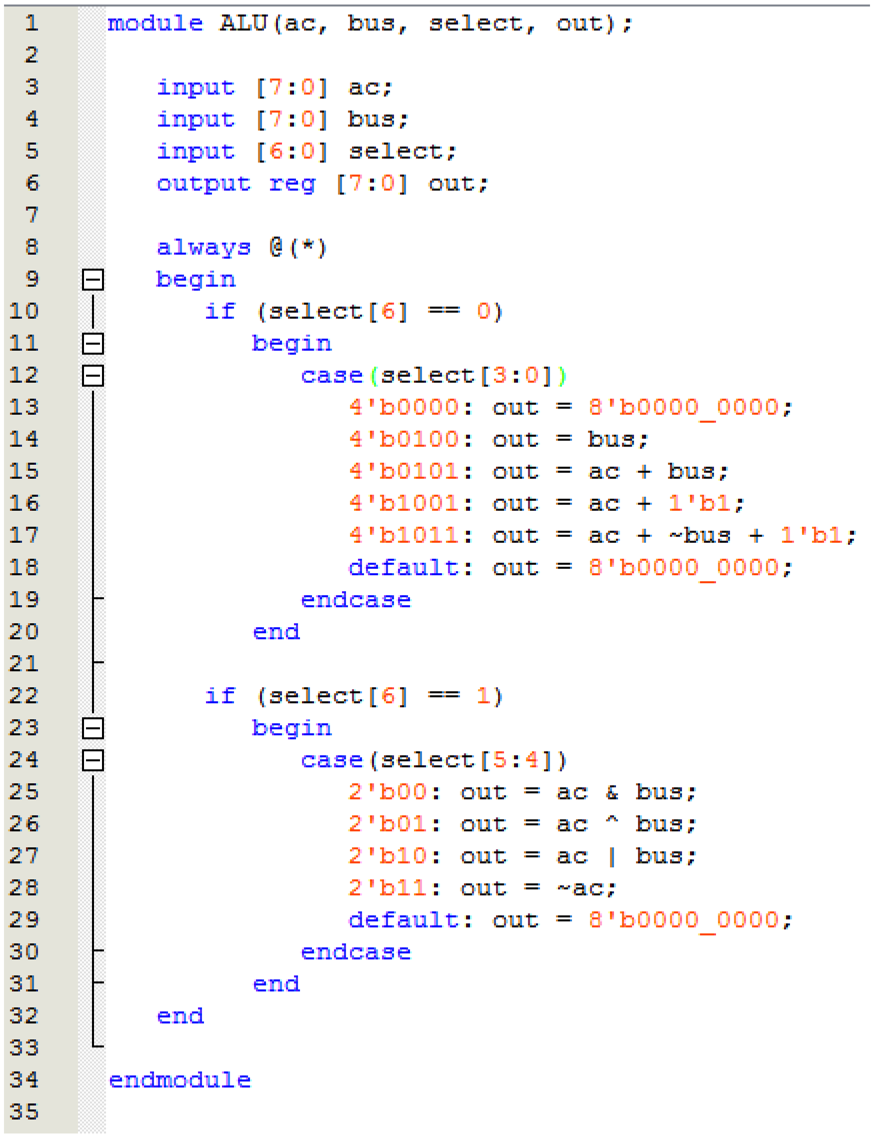
AC Register

The8-bit AC register is an accumulator register and acts as a general register used to contain nonspecific values, usually sums from instructions or ALU calculations. When the rising edge of the clock is enabled, the stored data is passed when Load = 1. This register is an initialization of the general 8-bit register module previously displayed. In this microprocessor design, I set an output variable equal to the contents of the PC register so that I can observe the current contents of the AC register.

ALU

The ALU is used to perform arithmetic operations on 8-bit data fed into it from the AC register and the bus which is determined by the 7-bit control signal ALUS. The 8-bit output of the ALU is sent to a tristate buffer connected to the bus and a z latch which sets z = 1, if the output of ALU = 0 and z = 0 if the output of the ALU isn’t 0. The operations of the ALU are listed in the table below as well as the Verilog module.

ALUS	**Output (Out)**
0XX0000	0000 0000
0XX0100	Bus
0XX0101	ac + bus
0XX1001	ac + 1
0XX1011	ac + NOT (bus) + 1
100XXXX	ac AND bus
101XXXX	ac XOR bus
110XXXX	ac OR bus
111XXXX	NOT (ac)

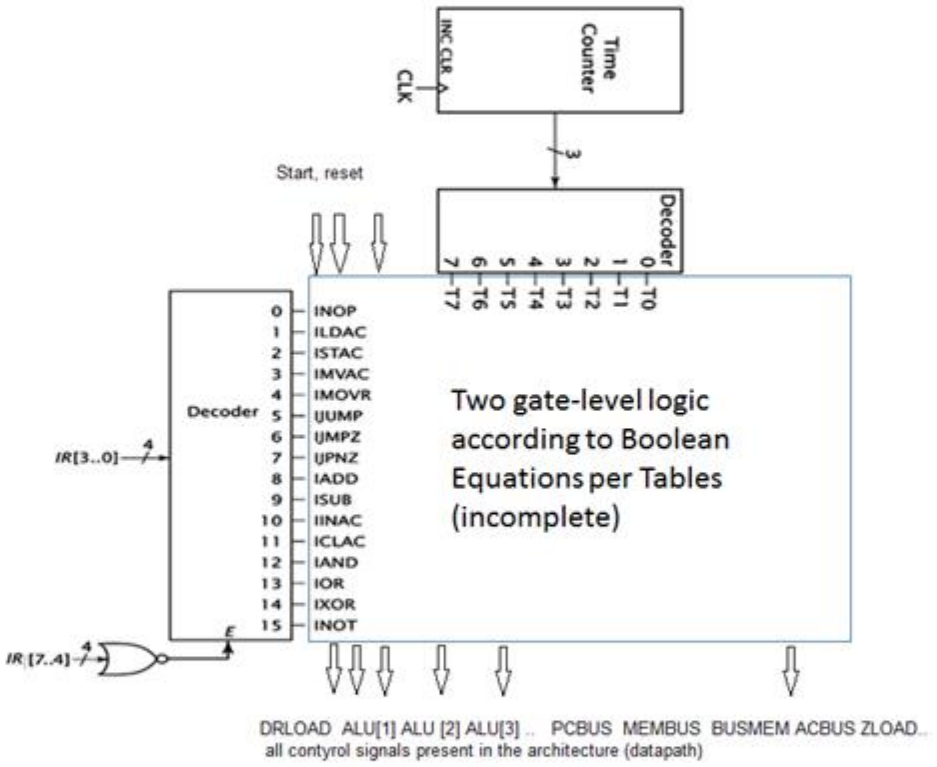


```
module ALU(ac, bus, select, out);

   input [7:0] ac;
   input [7:0] bus;
   input [6:0] select;
   output reg [7:0] out;

   always @(*)
   begin
      if (select[6] == 0)
         begin
            case(select[3:0])
               4'b0000: out = 8'b0000_0000;
               4'b0100: out = bus;
               4'b0101: out = ac + bus;
               4'b1001: out = ac + 1'b1;
               4'b1011: out = ac + ~bus + 1'b1;
               default: out = 8'b0000_0000;
            endcase
         end

      if (select[6] == 1)
         begin
            case(select[5:4])
               2'b00: out = ac & bus;
               2'b01: out = ac ^ bus;
               2'b10: out = ac | bus;
               2'b11: out = ~ac;
               default: out = 8'b0000_0000;
            endcase
         end
   end

endmodule
```

ALU Module

Control Unit

As stated earlier, the control unit sends control signals the datapath which controls how data is manipulated within the registers. The control signals are determined by two factors; the cycle time, and the decoded opcode of the instruction to be done. The cycle time comes from a counter which counts from T[0] to T[7] and represents the number of cycles needed to complete an instruction. The instruction comes from a 4 to 16 decoder which takes the 4 LSB of the content in the IR register and converts them to one of the 16 assembly instructions. This process is pictured below.



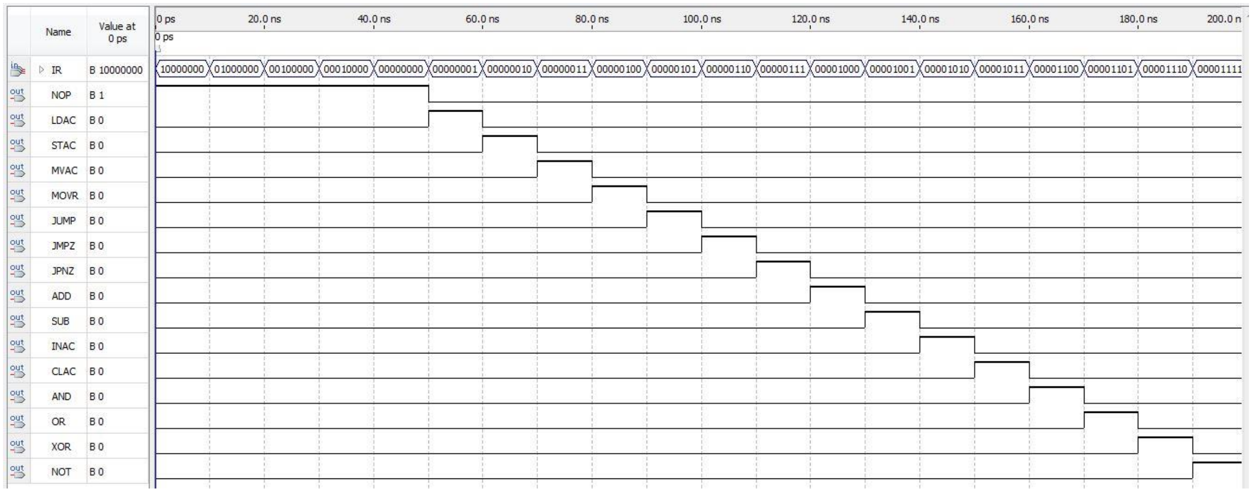
After the cycle time and the instruction is determined, the control signals are calculated by Boolean equations which come from derived tables that specify all control signals that must = 1 for a specific instruction to be done. After making the tables for a 16 instructions, at their respective cycles within the instruction, we multiply (ANDed) the cycle time, T with the instruction opcode and added (ORed) to every other instruction which in turn is multiplied (ANDed) to its respective cycle time, T, in the same method. After doing this for each control signal, we have derived the Boolean equations that will go in our control unit. The control signals are listed below.

```
we <= T[7]&instruction[13];
MEMbus <= T[1]|instruction[14]&(T[3]|T[4]|T[6])|instruction[10]&(T[3]|T[4])|instruction[8]&(T[3]|T[4])&(~z)|instruction[13]&(T[3]|T[4])|instruction[9]&(T[3]|T
BUSmem <= T[7]&instruction[13];
ARload <= T[0]|T[2]|instruction[14]&T[5]|instruction[13]&T[5];
ARinc  <= T[3]&(instruction[14]|instruction[10]|instruction[13])|instruction[9]&T[3]&z|instruction[8]&T[3]&(~z);
PCload <= instruction[10]&T[5]|instruction[9]&T[5]&z|(~z)&instruction[8]&T[5];
PCinc  <= T[1] | instruction[14]&(T[3]|T[4])|instruction[13]&(T[3]|T[4])|instruction[9]&(~z)&(T[3]|T[4])|instruction[8]&z&(T[3]|T[4]);
PCbus  <= T[0]|T[2];
DRload <= T[1]|instruction[14]&(T[3]|T[4]|T[6])|instruction[10]&(T[3]|T[4])|instruction[13]&(T[3]|T[4]|T[6])|z&instruction[9]&(T[3]|T[4])|instruction[8]&(T[3]
DRHbus <= T[5]&(instruction[14]|instruction[10]|instruction[13])|instruction[9]&T[5]&z|instruction[8]&(~z)&T[5];
DRLbus <= instruction[14]&T[7]|instruction[13]&T[7];
TRload <= T[4]&(instruction[14]|instruction[10]|instruction[13])|instruction[9]&z&T[4]|instruction[8]&(~z)&T[4];
TRbus  <= T[5]&(instruction[14]|instruction[10]|instruction[13])|instruction[9]&T[5]&z|instruction[8]&(~z)&T[5];
IRload <= T[2];
Rload  <= instruction[12]&T[3];
Rbus   <= T[3]&(instruction[11]|instruction[7]|instruction[6]|instruction[3]|instruction[2]|instruction[1]);
ACload <= instruction[14]&T[7]|T[3]&(instruction[11]|instruction[7]|instruction[6]|instruction[5]|instruction[4]|instruction[3]|instruction[2]|instruction[1]|
ACbus  <= instruction[12]&T[3]|instruction[13]&T[6];
ALUS[6]<= T[3]&(instruction[3]|instruction[2]|instruction[1]|instruction[0]);
ALUS[5]<= T[3]&(instruction[2]|instruction[0]);
ALUS[4]<= T[3]&(instruction[1]|instruction[0]);
ALUS[3]<= T[3]&(instruction[6]|instruction[5]);
ALUS[2]<= T[3]&instruction[7]|T[7]&instruction[14]|T[3]&instruction[11];
ALUS[1]<= T[3]&instruction[6];
ALUS[0]<= T[3]&(instruction[7]|instruction[6]|instruction[5]);
Zload  <= T[3]&(instruction[0]|instruction[1]|instruction[2]|instruction[3]|instruction[4]|instruction[5]|instruction[6]|instruction[7]);
```

Control signals and their Boolean equations

Decoder

In order to initialize the memory, we use the decoder to decode the 4 LSB of the contents in the IR register into 1 of 16 values which tells the control unit which instruction it should perform. The instruction will translate to an instruction[x] signal which is used to calculate the control signals needed to accomplish the instruction. Below, I have included a waveform of the decoder to show what the opcode for a given instruction is.



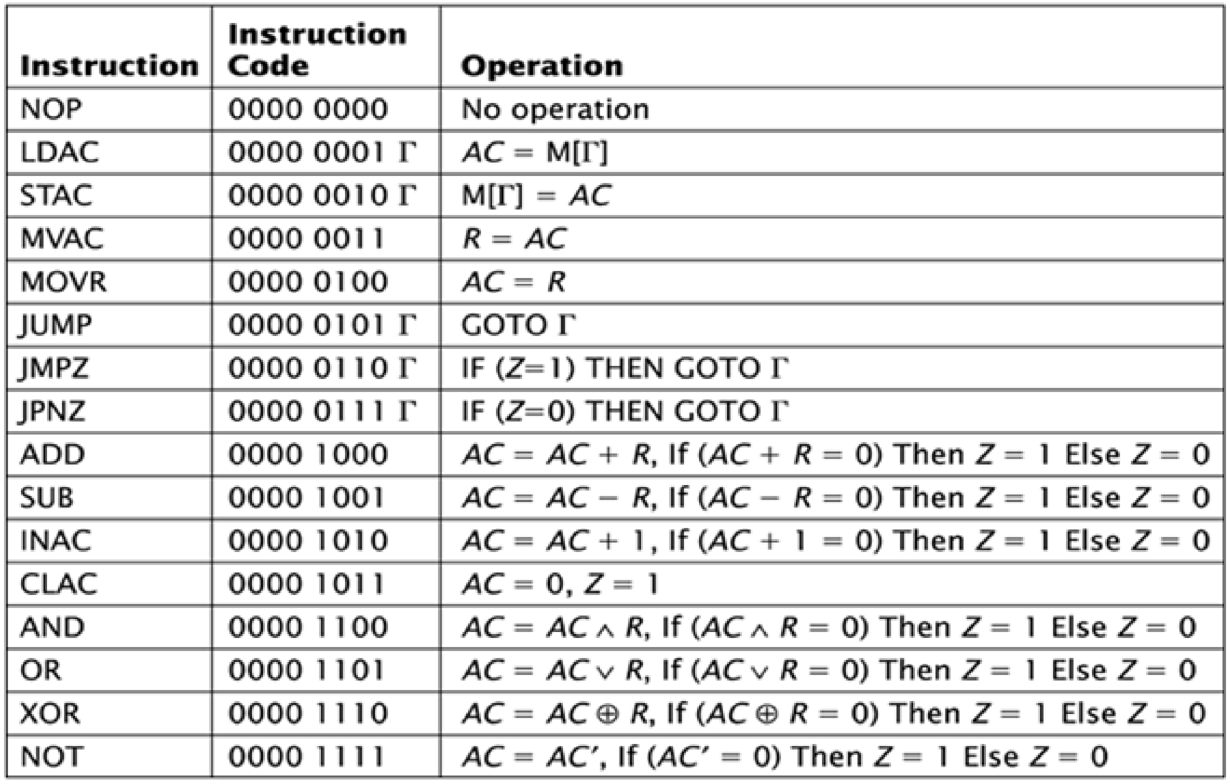
Control Signals

Here is a list of the control signals and a description of their functions.

Control Signal	**Function**
We	Enables data to be written into memory
MEMbus	Enables data to be feed from memory into the bus
BUSmem	Enables data to be feed from the bus into memory
ARLoad	Loads new data into the AR register
ARInc	Increments the contents of the AR register
PCLoad	Loads new data into the PC register
PCInc	Increments the contents of the PC register
PCBus	Enables data to be sent from the PC register to the bus
DRLoad	Enables new data into the DR register
DRLBus	Enables the 8 LSB from the DR register to be sent from the DR register to the bus
DRHBus	Enables the 8 MSB from the DR register to be sent from the DR register to the bus
TRLoad	Enables new data to be read into the TR register
TRBus	Enables data to be sent from the TR register to the bus
IRload	Enables new data into the TR register
Rload	Enables new data into the R register
RBus	Enables data to be sent from the R register to the bus
ACload	Enables new data into the AC register
ACBus	Enables data to be sent from the AC register to the bus
ALUS [6:0]	Control signals that determine which operation is performed in the ALU
Zload	Enables new data into the Z register

Assembly Instructions

These are the assembly instructions, their respective instruction code, and their functions.

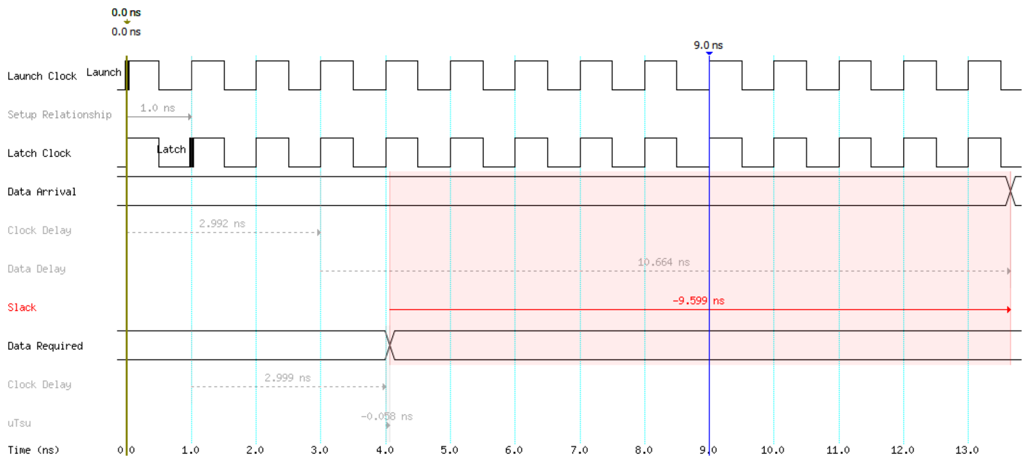


Instruction	Instruction Code	Operation
NOP	0000 0000	No operation
LDAC	0000 0001 Γ	$AC = M[\Gamma]$
STAC	0000 0010 Γ	$M[\Gamma] = AC$
MVAC	0000 0011	$R = AC$
MOVR	0000 0100	$AC = R$
JUMP	0000 0101 Γ	GOTO Γ
JMPZ	0000 0110 Γ	IF (Z=1) THEN GOTO Γ
JPNZ	0000 0111 Γ	IF (Z=0) THEN GOTO Γ
ADD	0000 1000	$AC = AC + R$, If $(AC + R = 0)$ Then $Z = 1$ Else $Z = 0$
SUB	0000 1001	$AC = AC - R$, If $(AC - R = 0)$ Then $Z = 1$ Else $Z = 0$
INAC	0000 1010	$AC = AC + 1$, If $(AC + 1 = 0)$ Then $Z = 1$ Else $Z = 0$
CLAC	0000 1011	$AC = 0$, $Z = 1$
AND	0000 1100	$AC = AC \wedge R$, If $(AC \wedge R = 0)$ Then $Z = 1$ Else $Z = 0$
OR	0000 1101	$AC = AC \vee R$, If $(AC \vee R = 0)$ Then $Z = 1$ Else $Z = 0$
XOR	0000 1110	$AC = AC \oplus R$, If $(AC \oplus R = 0)$ Then $Z = 1$ Else $Z = 0$
NOT	0000 1111	$AC = AC'$, If $(AC' = 0)$ Then $Z = 1$ Else $Z = 0$

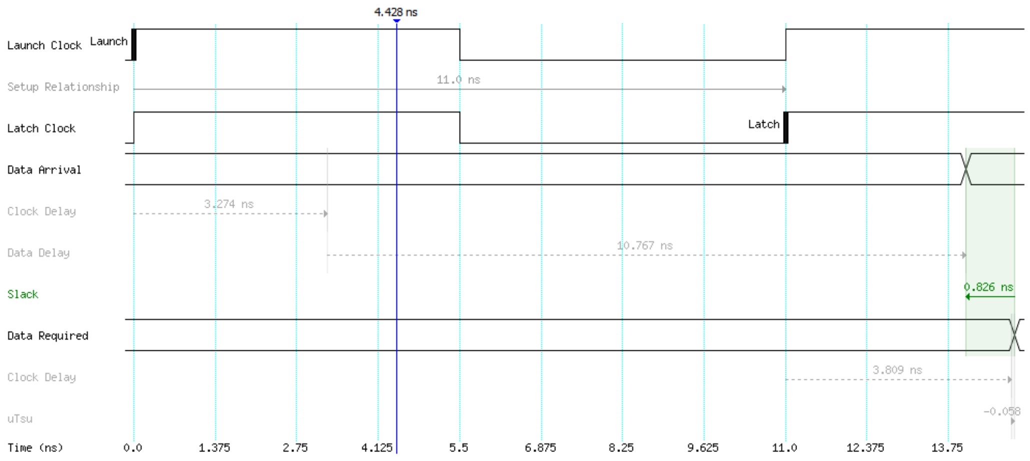
Timing Analysis

By doing a timing analysis in Quartus, we can see that out clock period is too fast to accurately operate our SMP. If we increase the clock period from 1ns to 9.947ns (round to 10ns), the clock will be synchronous with the operation cycles of the SMP, this effectively makes the frequency of the microprocessor:

$$f_c = \frac{1}{Tc} = \frac{1}{10ns} = 100MHz$$



Timing Error when clock period = 1ns



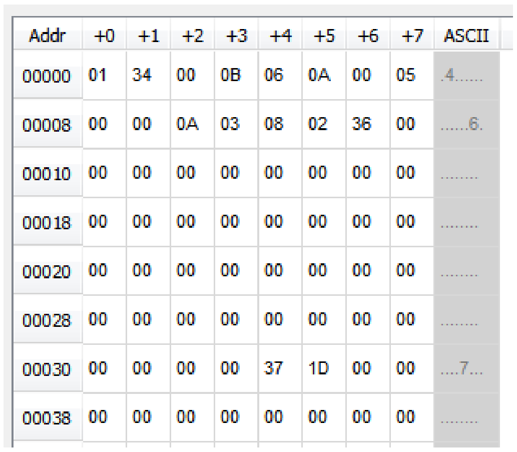
No timing error when clock period = 10ns

Assembly Programming Example

As a demonstration of the SMP, I have included an example where we program the SMP with a test set of assembly instructions and observe the output. In this example, we want to program the following assembly instructions into the SMP:

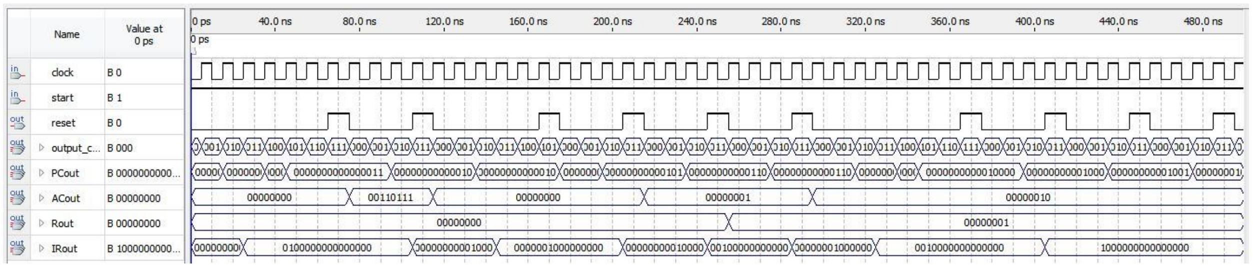
```
0:      LDAC 0034
3:      CLAC
4:      JMPZ 000A
7:      JUMP 0000
A:      INAC
B:      MVAC
C:      ADD
D:      STAC 0036

........................
0034:   55  //constant 55
0035 :  29  // constant 29
```

To do this, we simply program the .mif file with the appropriate hex equivalent of the binary opcodes of the instructions and registers. When we are done, the memory will look like this:



Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
00000	01	34	00	0B	06	0A	00	05	.4......
00008	00	00	0A	03	08	02	36	00	......6.
00010	00	00	00	00	00	00	00	00	....
00018	00	00	00	00	00	00	00	00	....
00020	00	00	00	00	00	00	00	00	....
00028	00	00	00	00	00	00	00	00	....
00030	00	00	00	00	37	1D	00	00	....7...
00038	00	00	00	00	00	00	00	00	....

Where 01 is the opcode for LDAC, 34 is the MSB of the address, 00 is the LSM, 0B is the opcode of CLAC, 06 is the opcode of JUMPZ and so one. After we compile the SMP with this newly initialized memory file, we observe the following output on the waveform:



The assembly code has the purpose of storing 55 into the AC, clearing the AC, incrementing the AC, storing this result in R, adding R and AC and storing that result in AC. Comparing this result with the waveform, we see that the AC has decimal value 55 stored in it, then 0, then 1 and finally 2. Similarly, the R register starts with value 0 and then 1. This is the exact result as described by the assembly programming, thus we can verify that the SMP is fully operational.

Verilog Files for SMP

Lastly, I have included the .v files for each module used to build the SMP in Quartus as the last pages of this report. This is provided so that the contents of a module can be seen without having to include .v file attachments.

ALU

```
module ALU(ac, bus, select, out);

      input [7:0] ac;
      input [7:0] bus;
      input [6:0] select;
      output reg [7:0] out;

      always @(*)
      begin
            if (select[6] == 0)
                  begin
                        case(select[3:0])
                              4'b0000: out = 8'b0000_0000;
                              4'b0100: out = bus;
                              4'b0101: out = ac + bus;
                              4'b1001: out = ac + 1'b1;
                              4'b1011: out = ac + ~bus + 1'b1;
                              default: out = 8'b0000_0000;
                        endcase
                  end

            if (select[6] == 1)
                  begin
                        case(select[5:4])
                              2'b00: out = ac & bus;
                              2'b01: out = ac ^ bus;
                              2'b10: out = ac | bus;
                              2'b11: out = ~ac;
                              default: out = 8'b0000_0000;
                        endcase
                  end
      end

endmodule
```

CONTROL UNIT

```
module control_unit(clock, start, instruction, z, we, MEMbus, BUSmem,
                                        ARload, ARinc, PCload, PCinc,
PCbus, DRload,
                                        DRHbus, DRLbus, TRload, TRbus,
IRload, Rload,
                                        Rbus, ACload, ACbus, ALUS, Zload,
output_counter,
                                        reset);

      input clock;
      input start;
      input z;
      input [15:0] instruction;

      output reg we, MEMbus, BUSmem, ARload, ARinc, PCload, PCinc, PCbus;
      output reg DRload, DRHbus, DRLbus, TRload, TRbus, IRload, Rload;
      output reg Rbus, ACload, ACbus, Zload;
      output reg [6:0] ALUS;
      output [2:0] output_counter;
      output reset;

      wire [7:0] T;
      wire [2:0] counter_out;

      counter   c(clock, start, reset, counter_out);
      // counter (clock, start, reset, out);

      decoder_time   d(counter_out, T);
      // decoder_time (in,          out);

      assign output_counter = counter_out; // observe counter clock
      assign reset =
instruction[14]&T[7]|instruction[10]&T[5]|T[3]&(instruction[11]|instructio
n[12])|instruction[13]&T[7]|(~z)&instruction[9]&T[4]|instruction[9]&z&T[5]
|instruction[8]&z&T[4]|instruction[8]&(~z)&T[5]|instruction[15]&T[3]|T[3]&
(instruction[7]|instruction[6]|instruction[5]|instruction[4]|instruction[3
]|instruction[2]|instruction[1]|instruction[0]);

      always @(*)
      begin
            we <= T[7]&instruction[13];
            MEMbus <=
T[1]|instruction[14]&(T[3]|T[4]|T[6])|instruction[10]&(T[3]|T[4])|instruct
ion[8]&(T[3]|T[4])&(~z)|instruction[13]&(T[3]|T[4])|instruction[9]&(T[3]|T
[4])&z;
            BUSmem <= T[7]&instruction[13];
            ARload <= T[0]|T[2]|instruction[14]&T[5]|instruction[13]&T[5];
            ARinc  <=
T[3]&(instruction[14]|instruction[10]|instruction[13])|instruction[9]&T[3]
&z|instruction[8]&T[3]&(~z);
            PCload <=
instruction[10]&T[5]|instruction[9]&T[5]&z|(~z)&instruction[8]&T[5];
            PCinc  <= T[1] |
instruction[14]&(T[3]|T[4])|instruction[13]&(T[3]|T[4])|instruction[9]&(~z
)&(T[3]|T[4])|instruction[8]&z&(T[3]|T[4]);
            PCbus  <= T[0]|T[2];
            DRload <=
T[1]|instruction[14]&(T[3]|T[4]|T[6])|instruction[10]&(T[3]|T[4])|instruct
ion[13]&(T[3]|T[4]|T[6])|z&instruction[9]&(T[3]|T[4])|instruction[8]&(T[3]
|T[4])&(~z);
            DRHbus <=
T[5]&(instruction[14]|instruction[10]|instruction[13])|instruction[9]&T[5]
&z|instruction[8]&(~z)&T[5];
            DRLbus <= instruction[14]&T[7]|instruction[13]&T[7];
            TRload <=
T[4]&(instruction[14]|instruction[10]|instruction[13])|instruction[9]&z&T[
4]|instruction[8]&(~z)&T[4];
            TRbus  <=
T[5]&(instruction[14]|instruction[10]|instruction[13])|instruction[9]&T[5]
&z|instruction[8]&(~z)&T[5];
            IRload <= T[2];
            Rload  <= instruction[12]&T[3];
            Rbus   <=
T[3]&(instruction[11]|instruction[7]|instruction[6]|instruction[3]|instruc
tion[2]|instruction[1]);
            ACload <=
instruction[14]&T[7]|T[3]&(instruction[11]|instruction[7]|instruction[6]|i
nstruction[5]|instruction[4]|instruction[3]|instruction[2]|instruction[1]|
instruction[0]);
            ACbus  <= instruction[12]&T[3]|instruction[13]&T[6];
            ALUS[6]<=
T[3]&(instruction[3]|instruction[2]|instruction[1]|instruction[0]);
            ALUS[5]<= T[3]&(instruction[2]|instruction[0]);
            ALUS[4]<= T[3]&(instruction[1]|instruction[0]);
            ALUS[3]<= T[3]&(instruction[6]|instruction[5]);
            ALUS[2]<=
T[3]&instruction[7]|T[7]&instruction[14]|T[3]&instruction[11];
            ALUS[1]<= T[3]&instruction[6];
            ALUS[0]<= T[3]&(instruction[7]|instruction[6]|instruction[5]);
            Zload  <=
T[3]&(instruction[0]|instruction[1]|instruction[2]|instruction[3]|instruct
ion[4]|instruction[5]|instruction[6]|instruction[7]);
      end

endmodule
```

COUNTER

```
module counter(clock, start, reset, out);

      input clock;
      input start;
      input reset;
      output reg [2:0] out;

      always @(posedge clock)
      begin
            if(reset == 0 & start == 1)
                  out <= out + 1'b1;
            else
                  out <= 0;
      end

endmodule
```

DATA PATH

```
module data_path(clock, we, MEMbus, BUSmem, ARload, ARinc,
                                   PCload, PCinc, PCbus, DRload, DRHbus,
DRLbus, TRload,
                                   TRbus, IRload, Rload, Rbus, ALUS,
ACload, ACbus,
                                   Zload, PCout, ACoutput, Rout, IRout,
z);

      input clock, we, MEMbus, BUSmem, ARload, ARinc;
      input PCload, PCinc, PCbus, DRload, DRHbus, DRLbus, TRload;
      input TRbus, IRload, Rload, Rbus, ACload, ACbus, Zload;
      input [6:0] ALUS;

      output [15:0] IRout;
      output [15:0] PCout;
      output [7:0] ACoutput;
      output [7:0] Rout;
      output z;

      wire [15:0] BUS;
      wire [15:0] ARout_MEMin;
      wire [15:0] PCout_TSin;
      wire [7:0] DRout;
      wire [7:0] TRout_TSin;
      wire [7:0] Rout_TSin;
      wire [7:0] ACout;
      wire [7:0] ALUout;
      wire ZeroFlag;   // zero flag (if AC = 0, Z = 1, else, Z = 0)

      assign PCout = PCout_TSin;   // observe contents of PC register
      assign ACoutput = ACout;     // observe contents of AC register
      assign Rout = Rout_TSin;     // observe contents of R register

      reg16_bit AR(clock, ARload, ARinc, BUS, ARout_MEMin);
      // reg16_bit(clock, load,   INC,   in,  out);

      memory SRAM(clock, ARout_MEMin, BUS[7:0], MEMbus, BUSmem,  we);
      //   memory(clock, address,     data,     MEMbus, BUSmem,  we);

      reg16_bit PC(clock, PCload, PCinc, BUS, PCout_TSin);
      //        PC(clock, load,   INC,   in,  out);

      tristate16_bit PCTS(PCout_TSin, PCbus, BUS);
      //   tristate16_bit(in,         enable, out);

      reg8_bit DR(clock, DRload, BUS[7:0], DRout);
      // reg8_bit(clock, load,   in,       out);

      tristate8_bit DRH(DRout, DRHbus, BUS[15:8]);
      //  tristate8_bit(in,    enable, out);

      tristate8_bit DRL(DRout, DRLbus, BUS[7:0]);
      //  tristate8_bit(in,    enable, out);

      reg8_bit TR(clock, TRload, DRout, TRout_TSin);
      // reg8_bit(clock, load,   in,    out);

      tristate8_bit TRTS(TRout_TSin, TRbus, BUS[7:0]);
      //   tristate8_bit(in,         enable, out);

      IR reg8_bit(clock, IRload, DRout, IRout);
      //       IR(clock, load,   in,    out);

      reg8_bit R(clock, Rload, BUS[7:0], Rout_TSin);
      //reg8_bit(clock, load,  in,       out);

      tristate8_bit RTS(Rout_TSin, Rbus,   BUS[7:0]);
      //  tristate8_bit(in,        enable, out);

      reg8_bit AC(clock, ACload, ALUout, ACout);
      // reg8_bit(clock, load,   in,     out);

      ALU ALU8_bit(ACout, BUS[7:0], ALUS, ALUout);
      //       ALU(AC,    BUS,      ALUS, out);

      tristate8_bit ACTS(ACout, ACbus, BUS[7:0]);
      //   tristate8_bit(in,    enable, out)

      z_latch zl(clock, ZeroFlag, Zload, z);
      // z_latch(clock, in,   load,  out);

   assign ZeroFlag = ~(ALUout[7] | ALUout[6] | ALUout[5] | ALUout[4] |
ALUout[3] |ALUout[2] | ALUout[1] | ALUout[0]);

endmodule
```

DECODER

```
module decoder(IR, NOP, LDAC, STAC, MVAC, MOVR, JUMP, JMPZ, JPNZ, ADD,
                              SUB, INAC, CLAC, AND, OR, XOR, NOT);

      input [7:0] IR;

      output reg NOP;
      output reg LDAC;
      output reg STAC;
      output reg MVAC;
      output reg MOVR;
      output reg JUMP;
      output reg JMPZ;
      output reg JPNZ;
      output reg ADD;
      output reg SUB;
      output reg INAC;
      output reg CLAC;
      output reg AND;
      output reg OR;
      output reg XOR;
      output reg NOT;

      always @(*)
      begin
            if ( ~(IR[7]|IR[6]|IR[5]|IR[4]) == 0)
                        {NOP, LDAC, STAC, MVAC, MOVR, JUMP, JMPZ, JPNZ,
ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} = 16'b0000_0000_0000_0000;
            else
                  begin
                        case(IR[3:0])
                              4'b0000: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b1000_0000_0000_0000;
                              4'b0001: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0100_0000_0000_0000;
                              4'b0010: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0010_0000_0000_0000;
                              4'b0011: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0001_0000_0000_0000;
                              4'b0100: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0000_1000_0000_0000;
                              4'b0101: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0000_0100_0000_0000;
```

```
                                4'b0110: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0000_0010_0000_0000;
                                4'b0111: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0000_0001_0000_0000;
                                4'b1000: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0000_0000_1000_0000;
                                4'b1001: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0000_0000_0100_0000;
                                4'b1010: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0000_0000_0010_0000;
                                4'b1011: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0000_0000_0001_0000;
                                4'b1100: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0000_0000_0000_1000;
                                4'b1101: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0000_0000_0000_0100;
                                4'b1110: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0000_0000_0000_0010;
                                4'b1111: {NOP, LDAC, STAC, MVAC, MOVR, JUMP,
JMPZ, JPNZ, ADD, SUB, INAC, CLAC, AND, OR, XOR, NOT} =
16'b0000_0000_0000_0001;
                          endcase
                    end
            end

endmodule
```

Time Cycle Decoder

```
module decoder_time(in, out);

      input [2:0] in;
      output reg [7:0] out;

      always @(*)
      begin
            case(in)
                  3'b000: out = 8'b0000_0001;
                  3'b001: out = 8'b0000_0010;
                  3'b010: out = 8'b0000_0100;
                  3'b011: out = 8'b0000_1000;
                  3'b100: out = 8'b0001_0000;
                  3'b101: out = 8'b0010_0000;
                  3'b110: out = 8'b0100_0000;
                  3'b111: out = 8'b1000_0000;
                  default: out = 8'b0000_0000;
            endcase
      end

endmodule
```

IR Register

```
module IR(clock, IRload, in, out);

      input clock;
      input IRload;
      input [7:0] in;
      output [15:0] out;
      reg [7:0] code;

      always @(posedge clock)
      begin
            if(IRload == 1)
                  code = in;
      end

      decoder
dc(code,out[15],out[14],out[13],out[12],out[11],out[10],out[9],out[8],out[
7],out[6],out[5],out[4],out[3],out[2],out[1],out[0]);
      //decoder (IR,    NOP,    LDAC,    STAC,    MVAC,    MOVR,    JUMP,
JMPZ,  JPNZ,  ADD,   SUB,   INAC,  CLAC,  AND,   OR,    XOR,   NOT);

endmodule
```

MEMORY

```
module memory(clock, address, data, MEMbus, BUSmem, we);

      input [15:0] address;
      input clock;
      input we; //write enable
      input MEMbus;
      input BUSmem;

      inout [7:0] data;// inout between memory and BUS
      reg [7:0] data_out;
      (*ram_init_file="mem.mif"*)  // initilization file
      reg [7:0] mem [65536:0];     // 64k

      assign data = (MEMbus && !we) ? data_out: 8'bzzzz_zzzz ;

      always @(posedge clock)
      begin
            if(we && BUSmem)
                  mem[address] = data; //writing
      end

      always @(MEMbus)
      begin
            if(!we)
                  data_out = mem[address]; //reading
      end

endmodule
```

PC REGISTER

```
module PC(clock, PCload, INC, in, out);

      input clock, INC, PCload;
      input [15:0] in;
      output reg [15:0] out;

      always @(posedge clock)
      begin
            if(INC == 1)
                  out = out + 1'b1;
            if(PCload == 1)
                  out = in;

      end

endmodule
```

8-BIT REGISTER

```
module reg8_bit(clock, load, in, out);

      input clock, load;
      input [7:0] in;
      output reg [7:0] out;

      always @(posedge clock)
      begin
            if (load == 1'b1)
                  out <= in;
      end

endmodule
```

16-BIT REGISTER

```
module reg16_bit(clock, load, INC, in, out);

      input clock, load, INC;
      input [15:0] in;
      output reg [15:0] out;

      always @(posedge clock)
      begin
            if (load == 1'b1)
                  out = in;
            if (INC == 1'b1)
                  out = out + 1'b1;
      end

endmodule
```

SMP

```
module SMP(clock, start, reset, output_counter, PCout, ACout, Rout, 
IRout);


      input clock;
      input start;
      output reset;
      output [2:0] output_counter; // display current counter
      output [15:0] PCout;   // output contents in PC register
      output [15:0] IRout; // output contents in IR register
      output [7:0] ACout;    // output contents in AC register
      output [7:0] Rout;     // output contents in R register

      wire we, MEMbus, BUSmem, ARload, ARinc, PCload, PCinc, PCbus;
      wire DRload, DRHbus, DRLbus, TRload, TRbus, IRload, Rload, Rbus;
      wire ACload, ACbus, Zload, z;
      wire [6:0] ALUS;

      control_unit cu(clock, start, IRout, z, we, MEMbus, BUSmem, ARload, 
                                        ARinc, PCload, PCinc, PCbus, 
DRload, DRHbus, 
                                        DRLbus, TRload, TRbus, IRload, 
Rload, Rbus, 
                                        ACload, ACbus, ALUS, Zload, 
output_counter, 
                                        reset);

      data_path dp(clock, we, MEMbus, BUSmem, ARload, ARinc, PCload, 
                                  PCinc, PCbus, DRload, DRHbus, DRLbus, 
TRload, TRbus, 
                                  IRload, Rload, Rbus, ALUS, ACload, 
ACbus, Zload, 
                                  PCout, ACout, Rout, IRout, z);

endmodule
```

8-BIT TRISTATE BUFFER

```
module tristate8_bit(in, enable, out);

      input [7:0] in;
      input enable;
      output reg [7:0] out;

      always @(*)
      begin
      out = (enable) ? in : 8'bzzzz_zzzz;
      end

endmodule
```

16-BIT TRISTATE BUFFER

```
module tristate16_bit(in, enable, out);

      input [15:0] in;
      input enable;
      output reg [15:0] out;

      always @(*)
      begin
      out = (enable) ? in : 16'bzzzz_zzzz_zzzz_zzzz;
      end

endmodule
```

Z LATCH

```
module z_latch(clock, in, load, out);

      input clock;
      input in;
      input load;
      output reg out;

      always @(posedge clock)
      begin
            if(load == 1)
                  out <= in;
      end

endmodule
```