COA Lab Project Report

Fall 2023

CSE-304L Computer Organization & Architecture Lab

Shahzad Bangash **(21PWCSE1980)**

Suleman Shah **(21PWCSE1983)**

Ali Asghar **(21PWCSE2059)**

Muhammad Shahab **(21PWCSE2074)**

"On our honor, as students of University of Engineering and Technology, we have neither given nor received unauthorized assistance on this academic work."

Submitted to:

Dr. Bilal Habib

January 31, 2024

Department of Computer Systems Engineering

University of Engineering and Technology, Peshawar

Sections:

- Introduction Page #3
- SAP-1 Page #3
- Tools Used Page #4
- SAP-1 Instruction Set Page #4
- Modules Used Page #4
- SAP-1 Operation Page #15
- Simulation Page #16
- Conclusion Page #20
- References Page #20

SAP-1 Implementation in Verilog

Introduction:

The SAP-1 (Simple As Possible-1) computer architecture serves as an excellent educational tool for understanding the fundamentals of computer architecture and organization. This project involves implementing the SAP-1 architecture in Verilog.

SAP-1:

The SAP-1 (Simple As Possible-1) serves as an entry-level model for understanding fundamental concepts of computer architecture. It consists of various modules, including a clock, program counter, registers, adder, memory, instruction register, bus, and controller. It provides a hands-on approach to learning the principles of computer organization and operation. This project is made with the help of Austin Morlan, you can check out his project in website given in references [1]. The overview of SAP-1 is shown in the figure 1.

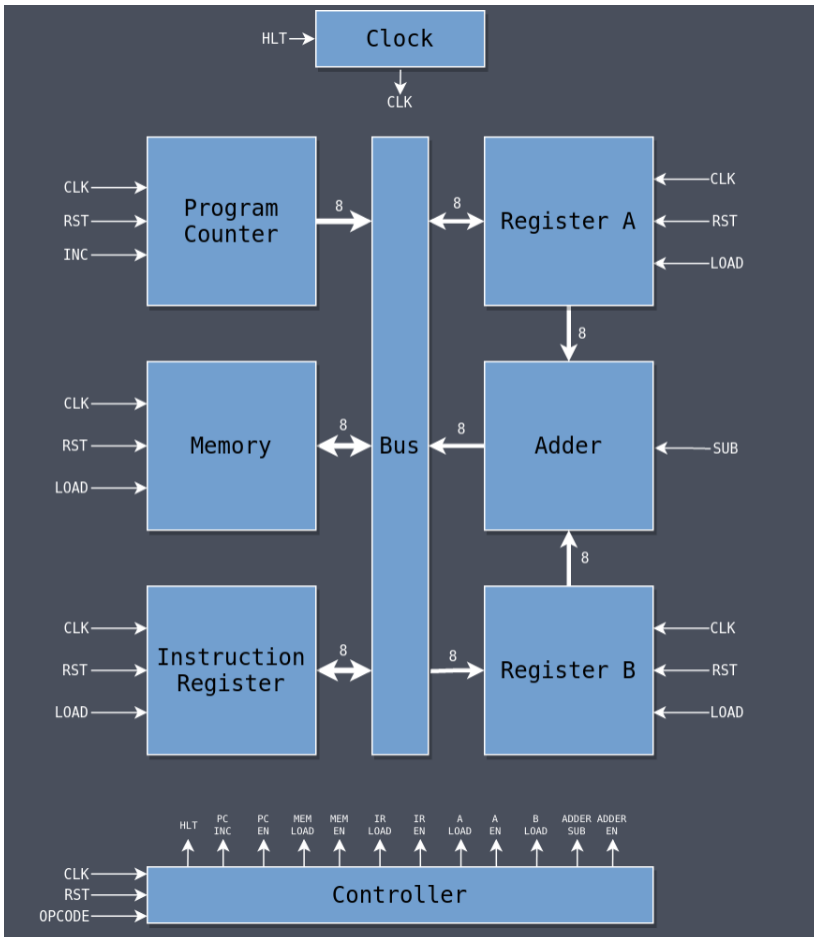

Figure 1, SAP-1 All modules

Tools Used:

1. **Verilog**

   A hardware descriptive language which we used to write code for SAP-1.

2. **Model Sim[2]**

   A tool for simulation and debugging tool often used in the field of digital electronics and integrated circuit design.

SAP-1 Instruction Set:

All instruction in our SAP-1 consists of opcode and operand. The opcode is 4 bits and operand are 4 bits making total of each instruction 8 bits. There are a total of **four** instructions in our SAP-1 Implementation. They are described below one by one.

1. **LDA:**

   This instruction loads the data from a specific memory location into the accumulator. The memory address is specified in the operand.

2. **ADD:**

   This instruction adds the data from a specific memory location to the data in the accumulator. The result is stored in the accumulator. The memory address is specified in the operand.

3. **SUB:**

   This instruction subtracts the data at a specific memory location from the data in the accumulator. The result is stored in the accumulator. The memory address is specified in the operand.

4. **HLT:**

   This instruction stops the execution of the program.

Modules Used:

The subsequent sections provide detailed explanations of each module.

1. Clock:

The clock module creates the clock signal and can stop the system when necessary. It takes an input called **clk_in** and produces an output named **clk_out**. The output mirrors the input unless the **hlt** signal is activated, in which case the output becomes zero. This feature is crucial for the **HLT** instruction, allowing the computer to halt its operations. If a program doesn't need to run indefinitely, the **HLT** instruction can be used to stop all further execution by halting the clock.

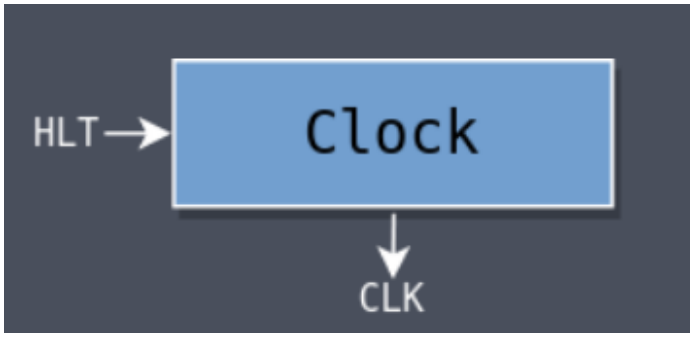

Figure 2, clock module

Code:

```
module clock(
    input hlt,
    input clk_in,
    output clk_out
);

assign clk_out = (hlt) ? 1'b0 : clk_in;

endmodule
```

Figure 3, clock module code

2. Program Counter (PC):

The program counter stores the address of the next instruction to be executed. This module increments its value from **0x0 (0)** to **0xF (15).** When the clock rises and **inc** is asserted, the PC is increased by one otherwise, it remains unchanged. The **rst** signal resets PC to zero when triggered.

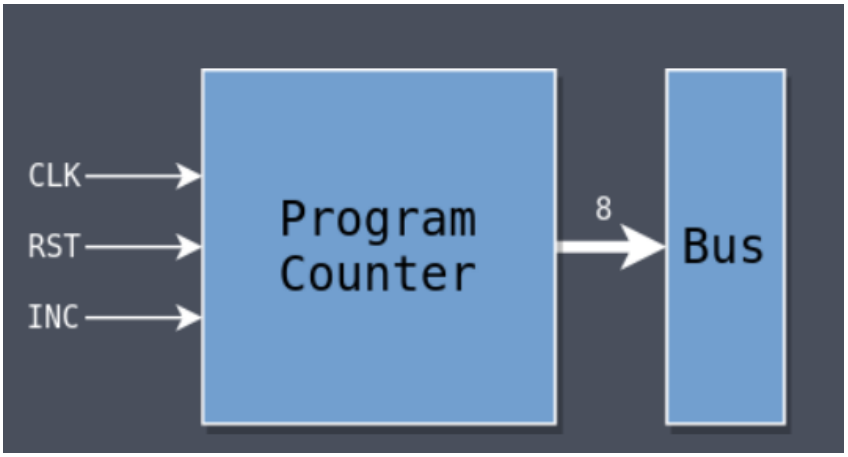

Figure 4, Program Counter module

Code:

```
module pc(
    input clk,
    input rst,
    input inc,
    output[7:0] out
);

reg[3:0] pc;

always @(posedge clk, posedge rst) begin
    if (rst) begin
        pc <= 4'b0;
    end else if (inc) begin
        pc <= pc + 1;
    end
end

assign out = pc;

endmodule
```

Figure 5, Program Counter module code

3. Register A:

Register A is basically accumulator which is the main register of the computer and many of the instructions depend upon it. Its internals look similar to some of the things seen previously: a **clk**, a **rst**, and an **out**. Bus is an input which is driven by some other module and Register A can read from it when it needs to load which happens when **load** is asserted.

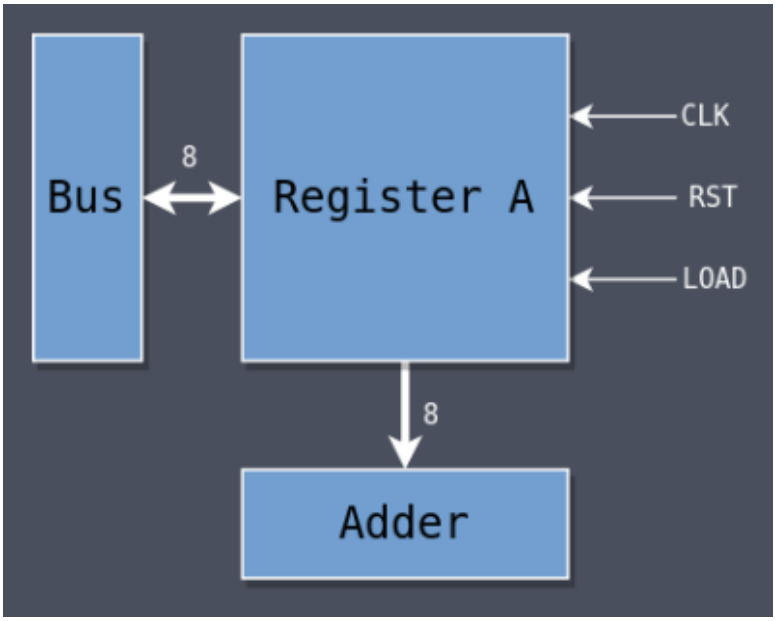

Figure 6, Register A module

Code:

```
module reg_a(
    input clk,
    input rst,
    input load,
    input[7:0] bus,
    output[7:0] out
);

reg[7:0] reg_a;

always @(posedge clk, posedge rst) begin
    if (rst) begin
        reg_a <= 8'b0;
    end else if (load) begin
        reg_a <= bus;
    end
end

assign out = reg_a;

endmodule
```

Figure 7, Register A module code

4. Register B:

Register B is identical to Register A in design but when it is used, it never drives the bus directly, its output is fed to the Adder only. The SAP-1 is designed so that Register A is where the main action occurs and Register B supports it.

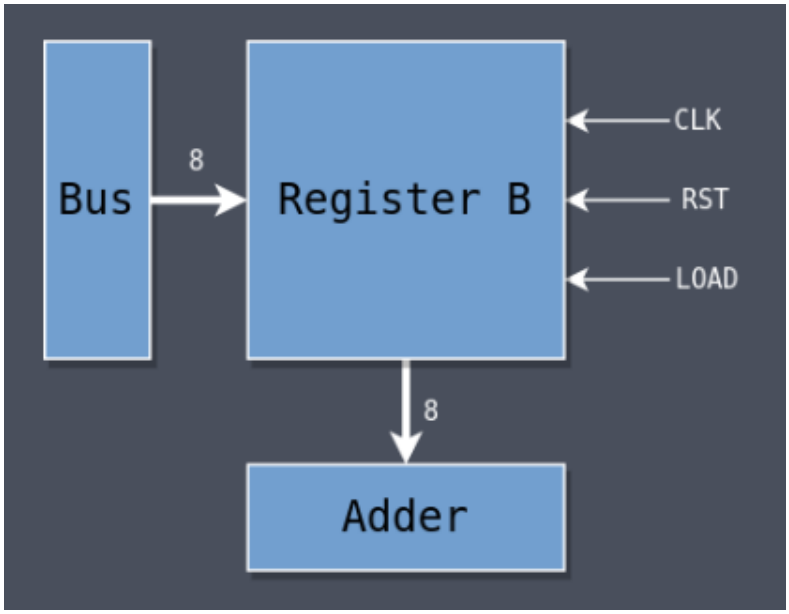

Figure 8, Register B module

Code:

```
module reg_b(
    input clk,
    input rst,
    input load,
    input[7:0] bus,
    output[7:0] out
);

reg[7:0] reg_b;

always @(posedge clk, posedge rst) begin
    if (rst) begin
        reg_b <= 8'b0;
    end else if (load) begin
        reg_b <= bus;
    end
end

assign out = reg_b;

endmodule
```

Figure 9, Register B module code

5. Adder:

The SAP-1 can only do addition and subtraction. The arithmetic module is called the Adder even though it also does subtraction (subtraction is just addition of a negative number after all). Notice the lack of a clock signal. The adder is constantly calculating either addition or subtraction based on the values in a and b and being placed directly onto its output **out**.

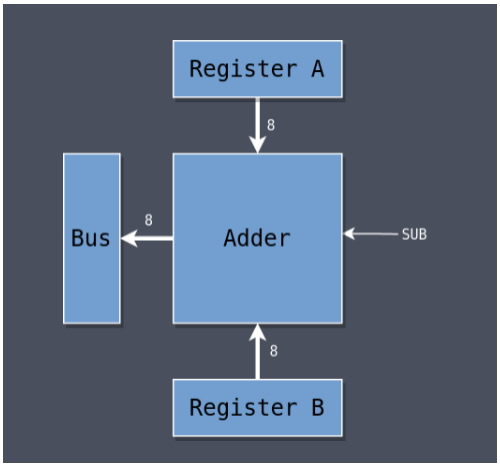

Figure 10, Adder module

Code:

```
module adder(
    input[7:0] a,
    input[7:0] b,
    input sub,
    output[7:0] out
);

assign out = (sub) ? a-b : a+b;

endmodulex
```

Figure 11, Adder module code

6. Memory:

The SAP-1 has **16 bytes** of memory. There is a 4-bit register called the **Memory Address Register (MAR)** which is used to store a memory address. The SAP-1 takes two clock cycles to read from memory, one cycle loads an address from the bus into the MAR (using the **load** signal) and the second cycle uses the value in the MAR to address into **ram** and output that value onto the bus.

The **initial** block is used to initialize the memory by loading its contents from a file which is an easy way to set the memory. The file has sixteen lines where each line represents a byte of memory.

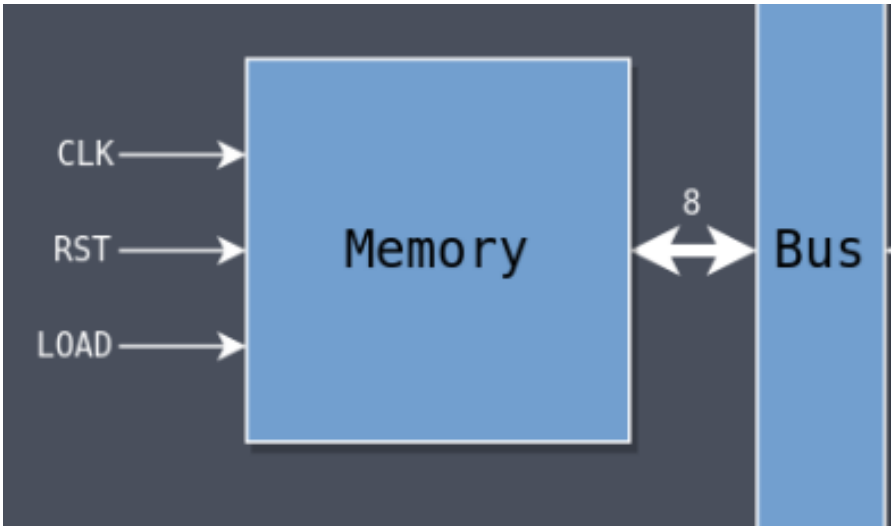

Figure 12, Memory module

Code:

```
module memory(
    input clk,
    input rst,
    input load,
    input[7:0] bus,
    output[7:0] out
);

initial begin
    $readmemh("program.bin", ram);
end

reg[3:0] mar;
reg[7:0] ram[0:15];

always @(posedge clk, posedge rst) begin
    if (rst) begin
        mar <= 4'b0;
    end else if (load) begin
        mar <= bus[3:0];
    end
end
```

Figure 13, Memory module code

7. Instruction Register (IR):

Before an instruction can be interpreted and acted upon, it needs to be loaded from memory into a module that can separate the opcode from the data. That's the job of the **Instruction Register (IR).**

An instruction has two components: the upper four bits are the **opcode** and the lower four bits are the **operand**. Some instructions use an operand and some don't in which case it will be ignored.

The **rst** and **load** signals do what they've done in other modules and the entire instruction is driven onto the output **out**. Later on when its used its divided into its two 4-bit components: the opcode and operand.

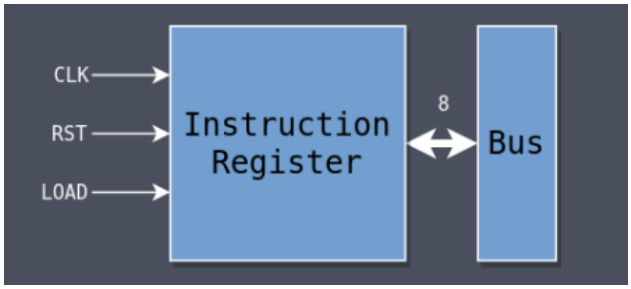

Figure 14, Instruction register module

Code:

```
module ir(
    input clk,
    input rst,
    input load,
    input[7:0] bus,
    output[7:0] out
);

reg[7:0] ir;

always @(posedge clk, posedge rst) begin
    if (rst) begin
        ir <= 8'b0;
    end else if (load) begin
        ir <= bus;
    end
end

assign out = ir;

endmodule
```

Figure 15, Instruction register module code

8. Bus:

The **bus** is how all of the modules send data between themselves. When one module needs to send data to another, it puts it on the bus. When one module needs to receive data from another, it reads it from the bus. All is coordinated by certain signals being asserted at certain times: a load signal reads from the bus and an enable signal outputs onto the bus.

The bus is eight bits and nothing more than wires that go between every component in the computer. It's eight bits wide because it's an 8-bit computer. All data operations occur in units of eight bits.

As shown in the module descriptions above, every module has an output called out which is always being driven by whatever value/logic the module contains. The controller then asserts an enable signal for whichever module's output is needed on the bus.

To select the proper module to be the only one driving the bus, I multiplexed the five module outputs **(adder_out**, **a_out**, **ir_out**, **mem_out**, **pc_out)** using the five enable signals **(adder_en**, **a_en**, **ir_en**, **mem_en**, **pc_en)** as the select. When no enable signals are asserted, the bus is driven with zero.

9. Controller:

The controller is the most complicated part of the computer and is where all of the interesting stuff happens. It decides what the computer will do next by asserting the different **control signals** that have gone into each of the modules.

Those control signals are:

- **hlt** Halt execution of the computer
- **pc_inc** Increment the Program Counter
- **pc_en** Put the value in the Program Counter onto the bus
- **mar_load** Load an address into the Memory Address Register
- **mem_en** Put a value from memory onto the bus
- **ir_load** Load a value from the bus into the Instruction Register
- **ir_en** Put the value in the Instruction Register onto the bus
- **a_load** Load a value from the bus into A
- **a_en** Put the value in A onto the bus
- **b_load** Load a value from the bus into B
- **adder_sub** Subtract the value in B from A
- **adder_en** Put the adder's value onto the bus

The controller module controls the behavior of the computer by asserting those signals at different times according to different stimuli.

Instruction execution occurs in a series of stages where each stage takes one clock cycle. The SAP-1 has six stages, starting at Stage 0 and counting to Stage 5, at which point it returns back to Stage 0 again. It continues on like that forever with every tick of the clock: 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, etc.

There is a 3-bit stage register (allowing values from 0 to 7) and with each tick of the clock the stage increases by one. Once it hits 5, it goes back to 0. It changes stage on the negative clock edge so that the signals will be set up properly before the modules need them on the next positive clock edge.

opcode is passed from the IR into the controller module to do different things based on what instruction is currently executing. What it does depends on the instruction and the stage of execution.

Finally, the output of the controller is the twelve control signals listed above. Different stages of different instructions will assert different signals to accomplish different things. Rather than pass the signals in individually, We pass them all in a single 12-bit value where each bit represents one of the signals. That keeps the code cleaner and makes it easier to set all the bits to zero before setting the ones that need to be asserted at that time.

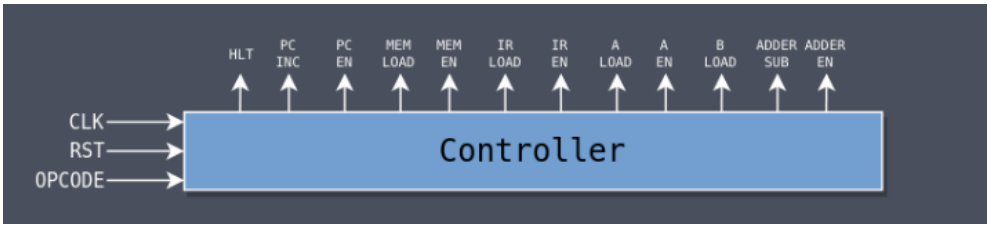

Figure 16, Controller module

Code:

```
Ln#
1  module controller(
2          input clk,
3          input rst,
4          input[3:0] opcode,
5          output[11:0] out
6  );
7
8  localparam SIG_HLT       = 11;
9  localparam SIG_PC_INC    = 10;
10 localparam SIG_PC_EN     = 9;
11 localparam SIG_MEM_LOAD  = 8;
12 localparam SIG_MEM_EN    = 7;
13 localparam SIG_IR_LOAD   = 6;
14 localparam SIG_IR_EN     = 5;
15 localparam SIG_A_LOAD    = 4;
16 localparam SIG_A_EN      = 3;
17 localparam SIG_B_LOAD    = 2;
18 localparam SIG_ADDER_SUB = 1;
19 localparam SIG_ADDER_EN  = 0;
20
21 localparam OP_LDA = 4'b0000;
22 localparam OP_ADD = 4'b0001;
23 localparam OP_SUB = 4'b0010;
24 localparam OP_HLT = 4'b1111;
25
26 reg[2:0]  stage;
27 reg[11:0] ctrl_word;
28
29 always @(negedge clk, posedge rst) begin
30         if (rst) begin
31                 stage <= 0;
32         end else begin
```

Figure 17.1 Controller module code

```
33                  if (stage == 5) begin
34                          stage <= 0;
35                  end else begin
36                          stage <= stage + 1;
37                  end
38          end
39  end
40
41  always @(*) begin
42          ctrl_word = 12'b0;
43
44          case (stage)
45                  0: begin
46                          ctrl_word[SIG_PC_EN] = 1;
47                          ctrl_word[SIG_MEM_LOAD] = 1;
48                  end
49                  1: begin
50                          ctrl_word[SIG_PC_INC] = 1;
51                  end
52                  2: begin
53                          ctrl_word[SIG_MEM_EN] = 1;
54                          ctrl_word[SIG_IR_LOAD] = 1;
55                  end
56                  3: begin
57                          case (opcode)
58                                  OP_LDA: begin
59                                          ctrl_word[SIG_IR_EN] = 1;
60                                          ctrl_word[SIG_MEM_LOAD] = 1;
61                                  end
62                                  OP_ADD: begin
63                                          ctrl_word[SIG_IR_EN] = 1;
64                                          ctrl_word[SIG_MEM_LOAD] = 1;
65                                  end
```

Figure 17.2 Controller module code

```
66                                  OP_SUB: begin
67                                          ctrl_word[SIG_IR_EN] = 1;
68                                          ctrl_word[SIG_MEM_LOAD] = 1;
69                                  end
70                                  OP_HLT: begin
71                                          ctrl_word[SIG_HLT] = 1;
72                                  end
73                          endcase
74                  end
75                  4: begin
76                          case (opcode)
77                                  OP_LDA: begin
78                                          ctrl_word[SIG_MEM_EN] = 1;
79                                          ctrl_word[SIG_A_LOAD] = 1;
80                                  end
81                                  OP_ADD: begin
82                                          ctrl_word[SIG_MEM_EN] = 1;
83                                          ctrl_word[SIG_B_LOAD] = 1;
84                                  end
85                                  OP_SUB: begin
86                                          ctrl_word[SIG_MEM_EN] = 1;
87                                          ctrl_word[SIG_B_LOAD] = 1;
88                                  end
89                          endcase
90                  end
91                  5: begin
92                          case (opcode)
93                                  OP_ADD: begin
94                                          ctrl_word[SIG_ADDER_EN] = 1;
95                                          ctrl_word[SIG_A_LOAD] = 1;
96                                  end
```

Figure 17.3 Controller module code

```
                                OP_SUB: begin
                                        ctrl_word[SIG_ADDER_SUB] = 1;
                                        ctrl_word[SIG_ADDER_EN] = 1;
                                        ctrl_word[SIG_A_LOAD] = 1;
                                end
                        endcase
                end
        endcase
end

assign out = ctrl_word;

endmodule
```

Figure 17.4 Controller module code

SAP-1 Operations:

Our SAP-1 has **four** operations that it can perform:

- **[0000] LDA $X** Load the value at memory location $X into A.
- **[0001] ADD $X** Add the value at memory location $X to A and store the sum in A.
- **[0010] SUB $X** Subtract the value at memory location $X from A and store the difference in A.
- **[1111] HLT** Halt execution of the program.

The values in the brackets represent the **opcode** and all but HLT have an operand. LDA, for example, has the opcode **0000** and its operand is the address of the value to be loaded into A.

Every instruction has the same **first three stages** which fetch the next instruction from memory based on the current value in the PC.

- **All Instructions**
  - **Stage 0**
    - Put the PC onto the bus (**pc_en**)
    - Load that value into the MAR (**mar_load**)
  - **Stage 1**
    - Increment the PC (**pc_inc**)
  - **Stage 2**
    - Put whatever is in memory at the MAR address onto the bus (**mem_en**)
      Load it into the IR (**ir_load**)

After the first **three stages**, the actions performed during the next three differ depending on the instruction, and some of the instructions do nothing at all.

- **LDA**
  - **Stage 3**
    - Put the instruction operand onto the bus (**ir_en**)
    - Load that value into the MAR (**mar_load**)

    - **Stage 4**
        - Put whatever is in memory at the MAR address onto the bus (**mem_en**)
        - Load that value into Register A (**a_load**)
    - **Stage 5**
        - Idle
- **ADD**
    - **Stage 3**
        - Put the instruction operand onto the bus (**ir_en**)
        - Load that value into the MAR (**mar_load**)
    - **Stage 4**
        - Put whatever is in memory at the MAR address onto the bus (**mem_en**)
        - Load that value into Register B (**b_load**)
    - **Stage 5**
        - Put the value in the adder onto the bus (**adder_en**)
        - Load that value into Register A (**a_load**)
- **SUB**
    - **Stage 3**
        - Put the instruction operand onto the bus (**ir_en**)
        - Load that value into the MAR (**mar_load**)
    - **Stage 4**
        - Put whatever is in memory at the MAR address onto the bus (**mem_en**)
        - Load that value into Register B (**b_load**)
    - **Stage 5**
        - Do subtraction rather than addition (**adder_sub**)
        - Put the value in the adder onto the bus (**adder_en**)
        - Load that value into Register A (**a_load)**
- **HLT**
    - **Stage 3**
        - Halt the clock (**hlt**)
    - **Stage 4**
        - Idle
    - **Stage 5**
        - Idle

Simulation:

All the modules is tested in **top** testbench. It instantiates all of the modules in the computer and connects them to each other. The initial block at the beginning runs once at the start of simulation to create a file called top_tb.v which contains all of the simulation data.

Code:

```
module top(
    input CLK
);


reg[7:0] bus;

wire rst;
wire hlt;
wire clk;
clock clock(
    .hlt(hlt),
    .clk_in(CLK),
    .clk_out(clk)
);

wire pc_inc;
wire pc_en;
wire[7:0] pc_out;
pc pc(
    .clk(clk),
    .rst(rst),
    .inc(pc_inc),
```

Figure 18.1, Top test bench

```
    .inc(pc_inc),
    .out(pc_out)
);


wire mar_load;
wire mem_en;
wire[7:0] mem_out;
memory mem(
    .clk(clk),
    .rst(rst),
    .load(mar_load),
    .bus(bus),
    .out(mem_out)
);


wire a_load;
wire a_en;
wire[7:0] a_out;
reg_a reg_a(
    .clk(clk),
    .rst(rst),
```

Figure 18.2, Top test bench

```
45          .rst(rst),
46          .load(a_load),
47          .bus(bus),
48          .out(a_out)
49      );
50
51
52      wire b_load;
53      wire[7:0] b_out;
54      reg_b reg_b(
55          .clk(clk),
56          .rst(rst),
57          .load(b_load),
58          .bus(bus),
59          .out(b_out)
60      );
61
62
63      wire adder_sub;
64      wire adder_en;
65      wire[7:0] adder_out;
66      adder adder(
67          .a(a_out),
```

Figure 18.3, Top test bench

```
top.v   program.bin   adder.v   clock.v   top_tb.v
68          .b(b_out),
69          .sub(adder_sub),
70          .out(adder_out)
71      );
72
73
74      wire ir_load;
75      wire ir_en;
76      wire[7:0] ir_out;
77      ir ir(
78          .clk(clk),
79          .rst(rst),
80          .load(ir_load),
81          .bus(bus),
82          .out(ir_out)
83      );
84
85      controller controller(
86          .clk(clk),
87          .rst(rst),
88          .opcode(ir_out[7:4]),
89          .out(
90          {
```

Figure 18.4, Top test bench

```
90      {
91          hlt,
92          pc_inc,
93          pc_en,
94          mar_load,
95          mem_en,
96          ir_load,
97          ir_en,
98          a_load,
99          a_en,
100         b_load,
101         adder_sub,
102         adder_en
103     })
104 );
105
106
107 always @(*) begin
108     if (ir_en) begin
109         bus = ir_out;
110     end else if (adder_en) begin
111         bus = adder_out;
112     end else if (a_en) begin
```

Figure 18.5, Top test bench

```
102         adder_en
103     })
104 );
105
106
107 always @(*) begin
108     if (ir_en) begin
109         bus = ir_out;
110     end else if (adder_en) begin
111         bus = adder_out;
112     end else if (a_en) begin
113         bus = a_out;
114     end else if (mem_en) begin
115         bus = mem_out;
116     end else if (pc_en) begin
117         bus = pc_out;
118     end else begin
119         bus = 8'b0;
120     end
121 end
122
123 endmodule
```

Figure 18.6, Top test bench

initial blocks aren't synthesizable; they're purely used for testing. In this case, one is used as a way of simulating a clock by toggling clk_in 128 times. With each iteration of the loop, clk_in is set to its inverse ~clk_in. So its state will be 0 -> 1 -> 0 -> 1 -> 0 -> 1, etc.

The **#1** is a time delay, which again is non-synthesizable and used only for testing. The test waits one time unit, toggles the clock, loops, waits one time unit, toggles the clock, loops, **128** times.
After configuring it a bit to nicely show the signals that are important, it displays this

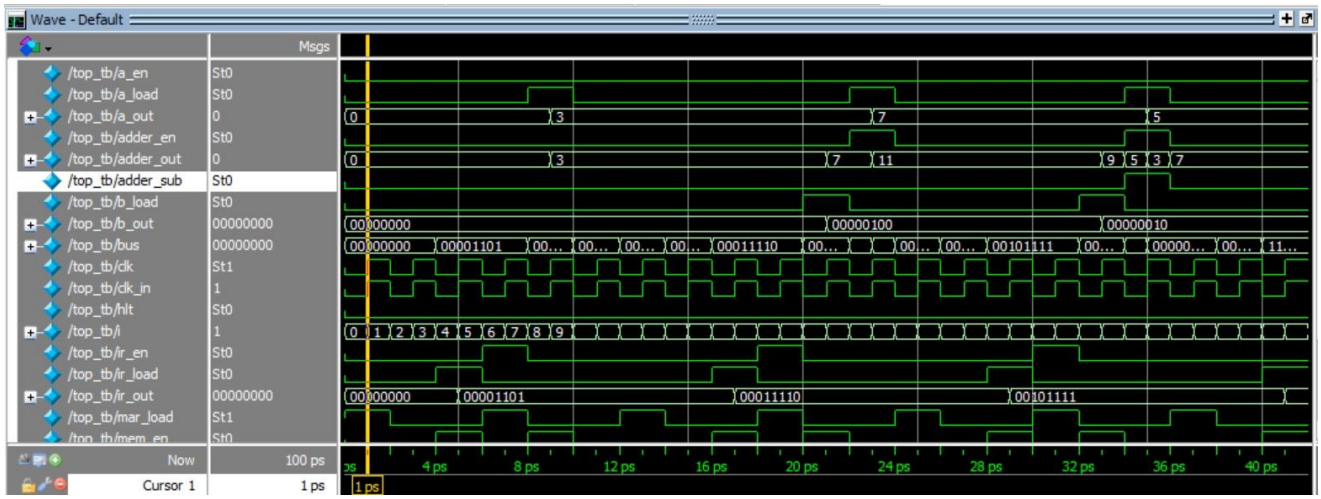

Figure 19, Top test bench output

Conclusion:

The Verilog implementation of the SAP-1 computer is successfully implemented and can be update to SAP-2. The modules work together in a synchronized manner, executing instructions and performing arithmetic operations. This project provides valuable experience in Verilog laying the foundation for more advanced computer architectures.

References:

[1]AUSTIN MORLAN. Building an FPGA Computer: SAP-1. https://austinmorlan.com/posts/fpga_computer_sap1/

[2]ModelSim. Intel. https://www.intel.com/content/www/us/en/software-kit/750368/modelsim-intel-fpgas-standard-edition-software-version-18-1.html