SAP-1

Computer Organization & Architecture Lab Project

BY:

Ali Asghar (21PWCSE2059)

Suleman Shah (21PWCSE1983)

Shahzad Bangash (21PWCSE1980)

Muhammad Shahab(21PWCSE2074)

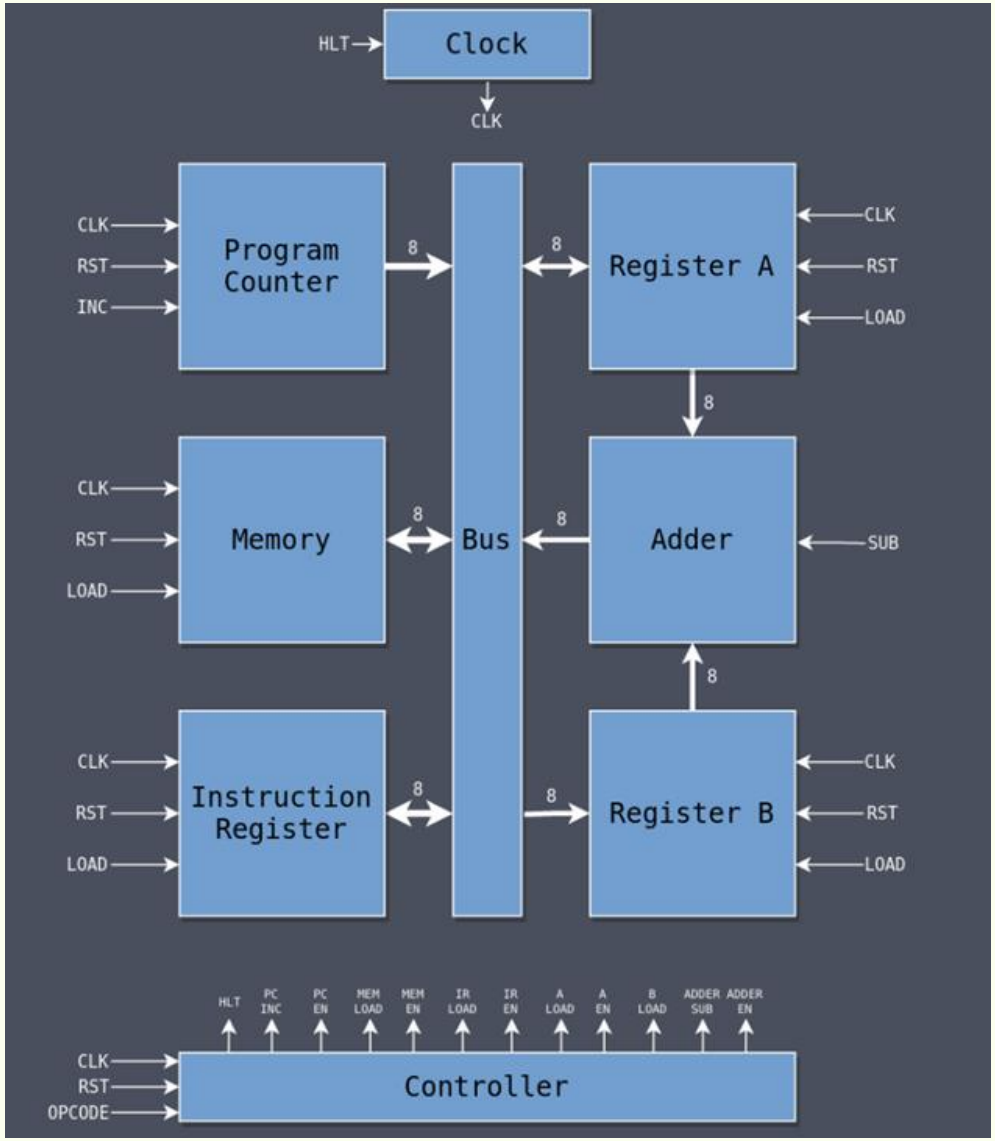
Introduction:

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 section [1].

SAP-1 All Modules:



Tools Used:

- **Verilog:**

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

- **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:

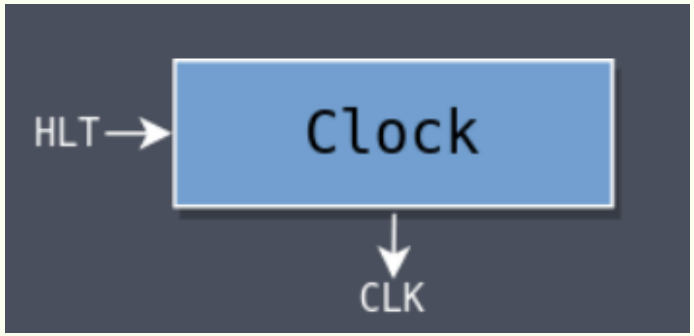
- **[0000] LDA $X:** Load the value at memory location $X into A.
- **[0001] ADD $X:** Add the value at memory location $X to A and store the sum in A.
- **[0010] SUB $X:** Subtract the value at memory location $X from A and store the difference in A.
- **[1111] HLT:** Halt execution of the program.

SAP-1 Modules:

1. Clock
2. Program Counter
3. Register A
4. Register B
5. Adder
6. Memory
7. Instruction Register
8. Bus
9. Controller

Clock Module:

- Generates the clock signal for the system and includes a mechanism to halt execution.
- Inputs: **halt** and **clk_in**.
- Output: **clk_out.**



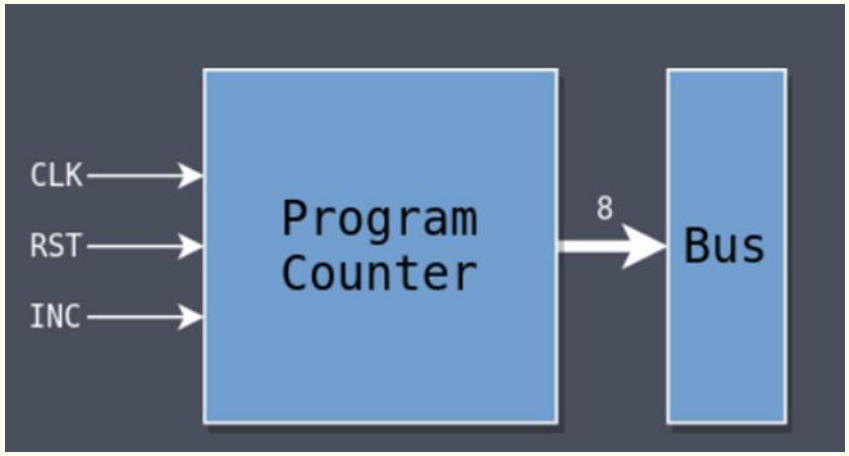
Code:

top.v | program.bin | adder.v | clock.v

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

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

endmodule
```

PC Module:

- Keeps track of the memory address of the next instruction.
- Inputs: **clk** (clock), **rst** (reset), **inc** (increment).
- Outputs: **out** (current program counter value).



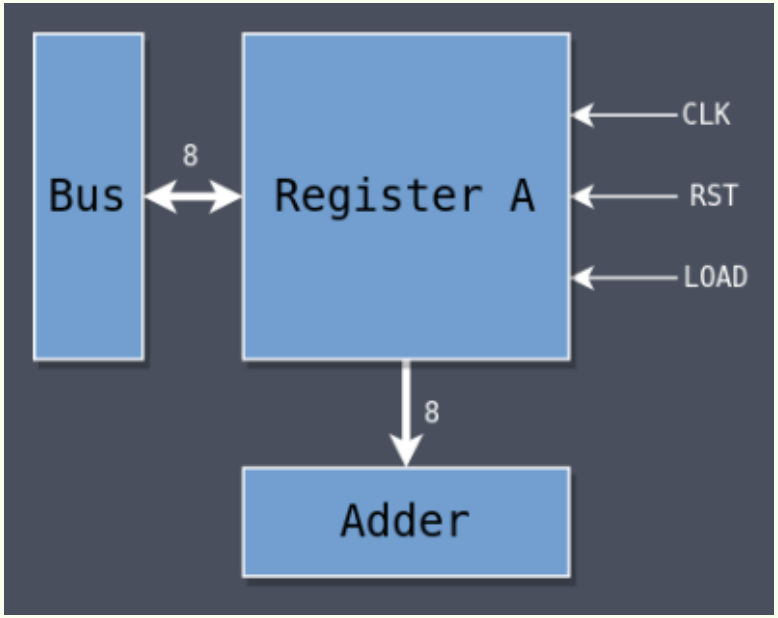
Code:

top.v | program.bin | adder.v | clock.v | top_tb.v | controller.v | ir.v | memory.v | pc.v

```
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
```

Register A:

- Primary register for storing data also known as accumulator.
- Inputs: **clk** (clock), **rst** (reset), **load** (load data), **bus** (data from bus).
- Outputs: **out** (data stored in Register A).



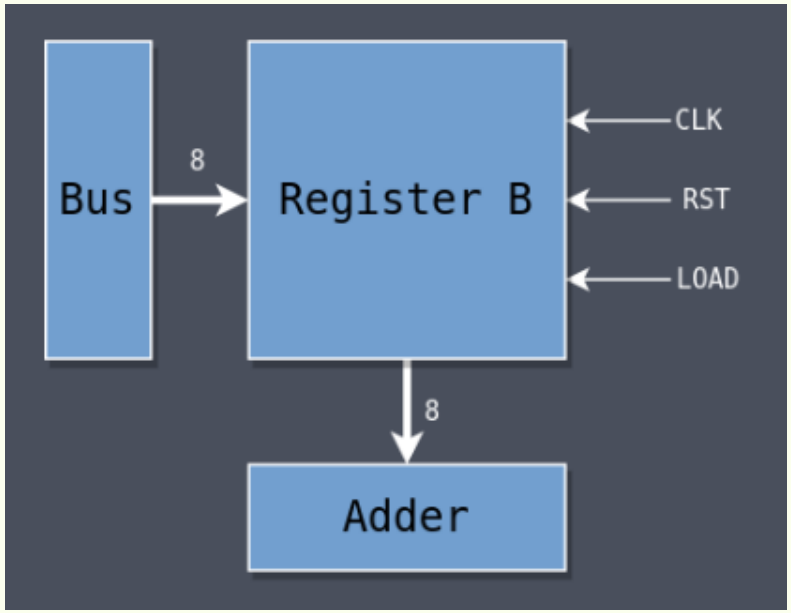
Code:

top.v | program.bin | adder.v | clock.v | top_tb.v | controller.v | ir.v | memory.v

```
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
```

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
- Inputs: **clk** (clock), **rst** (reset), **load** (load data), **bus** (data from bus).
- Outputls: **out** (data stored in Register B).

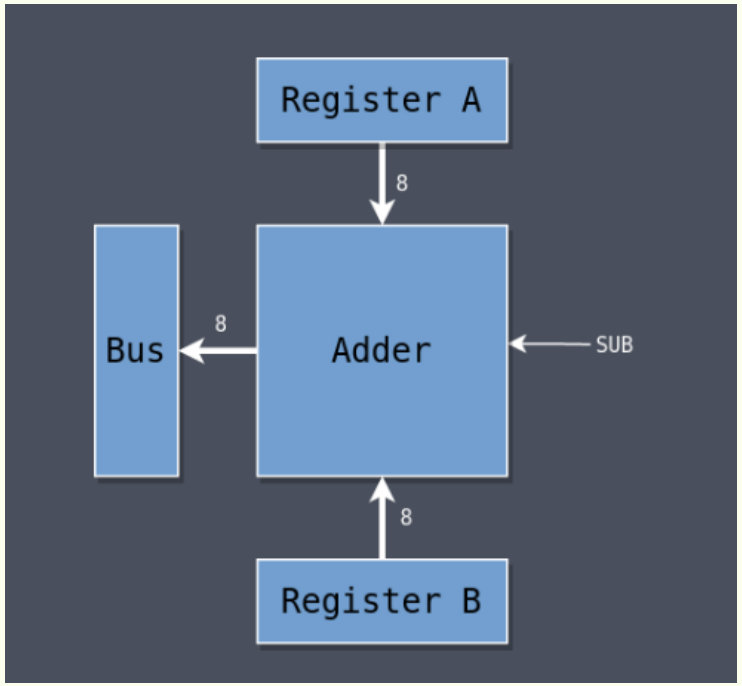


Code:

```
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
```

Adder:

- Performs addition and subtraction operations.
- Inputs: data from **reg A** and **B**, **sub** (subtraction signal).
- Outputs: **out** (result of addition/subtraction)



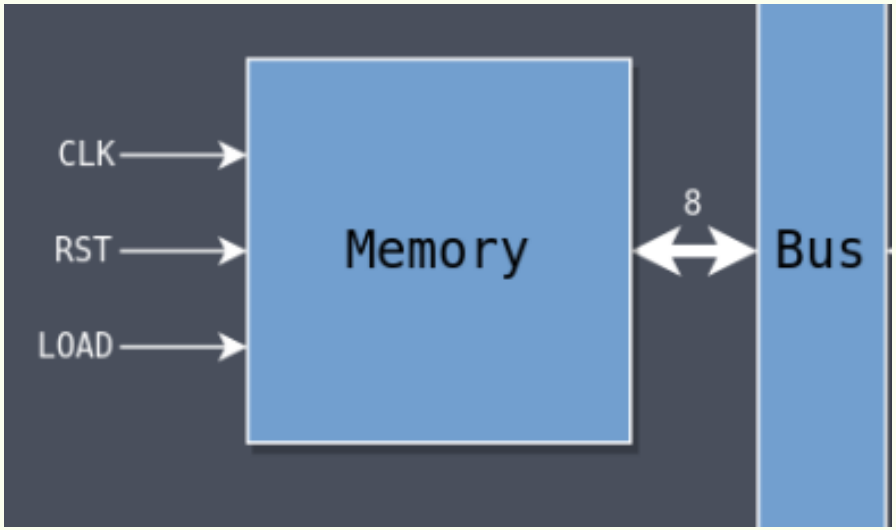
Code:

top.v | program.bin | adder.v

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

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

endmodulex
```

Memory:

- Represents the RAM of the computer.
- Inputs: **clk** (clock), **rst** (reset), **load** (load memory address), **bus** (data from bus).
- Outputs: **out**



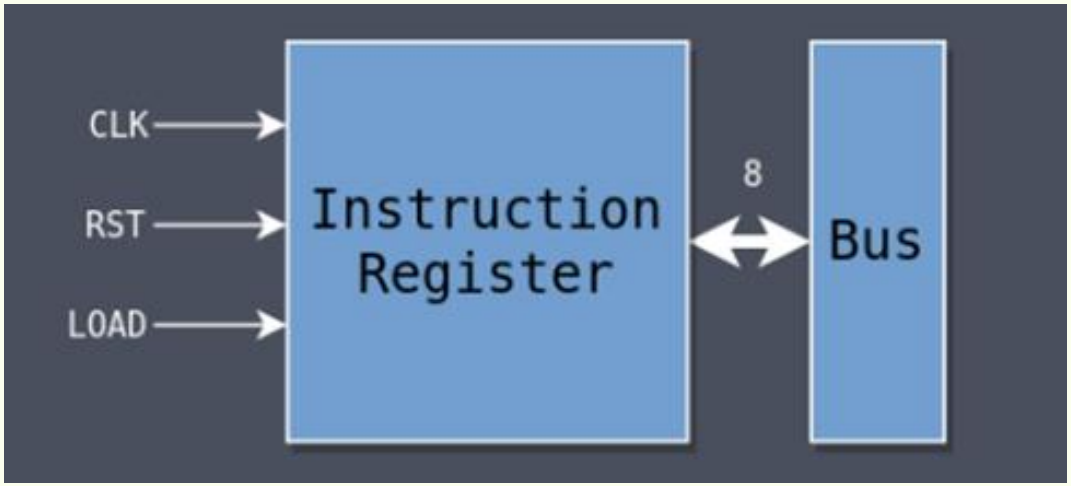
Code:

top.v | program.bin | adder.v | clock.v | top_tb.v | controller.v | ir.v | memory.v

```
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
```

IR:

- Decodes the current instruction.
- Inputs: **clk** (clock), **rst** (reset), **load** (load instruction), **bus** (data from bus).
- Outputs: **out**



Code:

top.v | program.bin | adder.v | clock.v | top_tb.v | controller.v | ir.v

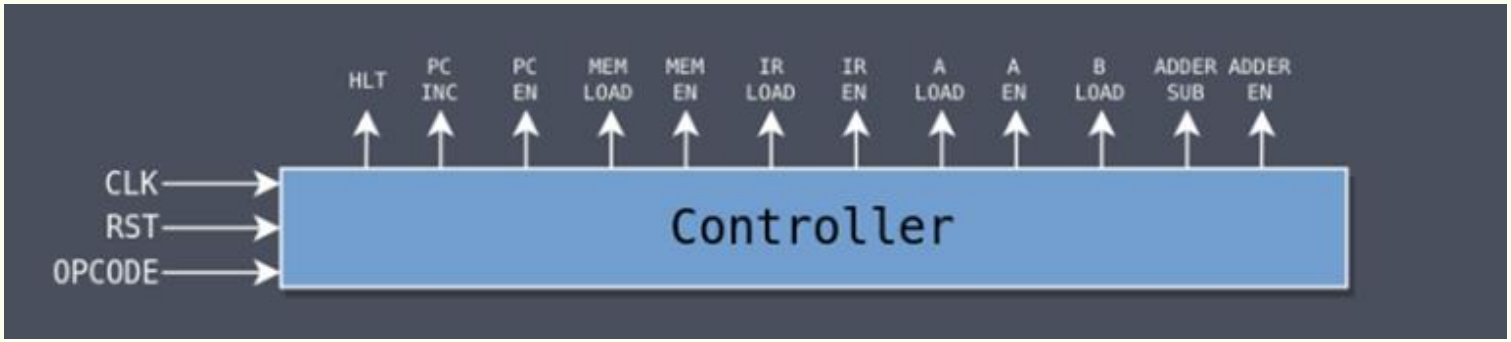
```
module ir(
    input clk,
    input rst,
    input load,
    input[7:0] bus,
    output[7:0] out
);

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
```

BUS:

- Facilitates data exchange between modules.
- Inputs: **ir_en**, **adder_en**, a**_en**, **mem_en**, **pc_en** (module enable signals).
- Outputs: **bus** (data on the bus).

Controller:

- Controls the behavior of the computer based on the instruction and stage.
- Inputs: **clk** (clock), **rst** (reset), **opcode** (instruction opcode).
- Outputs: Various control signals (**hlt**, **pc_inc**, **pc_en**, **mar_load**, **mem_en, ir_load, ir_en, a_load, a_en, b_load**, **adder_sub**, **adder_en**).



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
```

Code:

```
                if (stage == 5) begin
                        stage <= 0;
                end else begin
                        stage <= stage + 1;
                end
        end
end

always @(*) begin
        ctrl_word = 12'b0;

        case (stage)
                0: begin
                        ctrl_word[SIG_PC_EN] = 1;
                        ctrl_word[SIG_MEM_LOAD] = 1;
                end
                1: begin
                        ctrl_word[SIG_PC_INC] = 1;
                end
                2: begin
                        ctrl_word[SIG_MEM_EN] = 1;
                        ctrl_word[SIG_IR_LOAD] = 1;
                end
                3: begin
                        case (opcode)
                                OP_LDA: begin
                                        ctrl_word[SIG_IR_EN] = 1;
                                        ctrl_word[SIG_MEM_LOAD] = 1;
                                end
                                OP_ADD: begin
                                        ctrl_word[SIG_IR_EN] = 1;
                                        ctrl_word[SIG_MEM_LOAD] = 1;
                                end
```

Code:

```
                        OP_SUB: begin
                                ctrl_word[SIG_IR_EN] = 1;
                                ctrl_word[SIG_MEM_LOAD] = 1;
                        end
                        OP_HLT: begin
                                ctrl_word[SIG_HLT] = 1;
                        end
                endcase
        end
        4: begin
                case (opcode)
                        OP_LDA: begin
                                ctrl_word[SIG_MEM_EN] = 1;
                                ctrl_word[SIG_A_LOAD] = 1;
                        end
                        OP_ADD: begin
                                ctrl_word[SIG_MEM_EN] = 1;
                                ctrl_word[SIG_B_LOAD] = 1;
                        end
                        OP_SUB: begin
                                ctrl_word[SIG_MEM_EN] = 1;
                                ctrl_word[SIG_B_LOAD] = 1;
                        end
                endcase
        end
        5: begin
                case (opcode)
                        OP_ADD: begin
                                ctrl_word[SIG_ADDER_EN] = 1;
                                ctrl_word[SIG_A_LOAD] = 1;
                        end
```

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
```

SAP-1 Operation

- All Instructions have **three** stages same
- 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.

SAP-1 Operation

LDA Instruction:

- 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

SAP-1 Operation

ADD Instruction:

- 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)

SAP-1 Operation

SUB Instruction:

- 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)

SAP-1 Operation

HLT Instruction:

- 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),
```

Code:

```
23          .inc(pc_inc),
24          .out(pc_out)
25      );
26
27
28      wire mar_load;
29      wire mem_en;
30      wire[7:0] mem_out;
31      memory mem(
32          .clk(clk),
33          .rst(rst),
34          .load(mar_load),
35          .bus(bus),
36          .out(mem_out)
37      );
38
39
40      wire a_load;
41      wire a_en;
42      wire[7:0] a_out;
43      reg_a reg_a(
44          .clk(clk),
45          .rst(rst),
```

Code:

```
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),
```

Code:

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          {
```

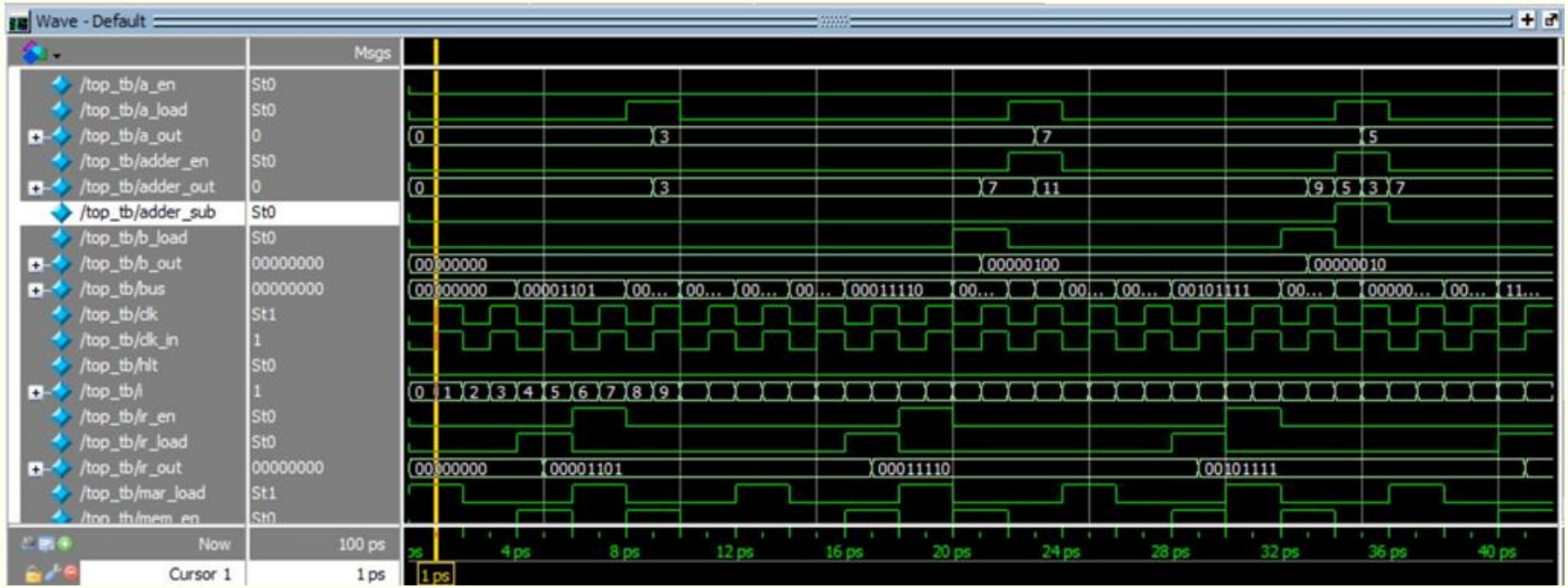
Code:

```
    {
        hlt,
        pc_inc,
        pc_en,
        mar_load,
        mem_en,
        ir_load,
        ir_en,
        a_load,
        a_en,
        b_load,
        adder_sub,
        adder_en
    })
);


always @(*) begin
    if (ir_en) begin
        bus = ir_out;
    end else if (adder_en) begin
        bus = adder_out;
    end else if (a_en) begin
```

Code:

```
            adder_en
        })
);


always @(*) begin
    if (ir_en) begin
        bus = ir_out;
    end else if (adder_en) begin
        bus = adder_out;
    end else if (a_en) begin
        bus = a_out;
    end else if (mem_en) begin
        bus = mem_out;
    end else if (pc_en) begin
        bus = pc_out;
    end else begin
        bus = 8'b0;
    end
end

endmodule
```

Output:


Wave - Default
Msgs
/top_tb/a_en St0
/top_tb/a_load St0
/top_tb/a_out 0
/top_tb/adder_en St0
/top_tb/adder_out 0
/top_tb/adder_sub St0
/top_tb/b_load St0
/top_tb/b_out 00000000
/top_tb/bus 00000000
/top_tb/clk St1
/top_tb/clk_in 1
/top_tb/hlt St0
/top_tb/i 1
/top_tb/ir_en St0
/top_tb/ir_load St0
/top_tb/ir_out 00000000
/top_tb/mar_load St1
Now 100 ps
Cursor 1 1 ps
4 ps
8 ps
12 ps
16 ps
20 ps
24 ps
28 ps
32 ps
36 ps
40 ps

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

THANK YOU FOR YOUR TIME