CSCI2500 Final Project

How do computers actually compute?

Team ID:

49

Authors:

Ethan Garcia

Joshua Sundararaman

Jimmy Chen

Course Coordinator:

Konstatin Kuzmin

8th December 2023

Contents

1 Requirements

- **Bit Width** - 16-bit
- **Main Memory Size** - 64Ki bytes
- **Main Memory Organization** - 8Ki x 8
- **Max number of bits to be used by L1 cache:** - 40,000
- **Additional Addressing Mode** - Indirect

1.0.1 Components Needed

Component	**Function**
AC or Accumulator	Intermediate data is stored within the AC
PC or Program Counter	As the name suggests it counts the current position of the code, each line has its own address; PC needs to be incremented after each instruction
MAR or Memory Access Register	Stores or fetches the 'data' at the given address
MBR or Memory Buffer Register	Stores the data when being transferred
IR or Instruction Register	Stores the instruction word of the currently executing instruction
ALU or Arithmetic and Logic Unit	Performs arithmetic and Boolean operations
Main memory	Stores data and instructions

2 Design 1

2.1 Instruction Set Architecture

Opcode	Operation
1000	ADD
1001	HALT
1010	LOAD
1011	STORE
1100	CLEAR
1101	SKIP
1110	JUMP

2.2 Components

2.2.1 ALU

Overview:

The ALU is a crucial component in a computer's architecture responsible for performing arithmetic and logic operations. In this 16-bit ALU, the operations include addition, subtraction, bitwise AND, bitwise OR, bitwise XOR, bitwise NOT for A, bitwise NOT for B, and clearing the output. The `ALU_Sel` input determines the operation to be performed.

Inputs:

- A and B: 16-bit operands on which operations are performed
- `ALU_Sel`: 4-bit input that specifies the operation to be executed.

Outputs:

- `ALU_Out`: 16-bit output that stores the result of the operation.
- `CarryOut`: 1-bit output that stores the carry bit of the operation.

Internal Signals:

- `ALU_Result`: 16-bit register storing the result of the selected operation.
- `tmp`: 17-bit wire used to calculate the carry-out during addition.

Operation Execution:

- The `tmp` wire is used to calculate the sum of A and B along with the carry bit (if any) during addition.
- The carry-out (`CarryOut`) is extracted from the 17th bit of `tmp`.
- The result of the selected operation is determined using a case statement based on the value of `ALU_Sel`.

Supported Operations:

- `0000`: Addition
- `0001`: Subtraction
- `0010`: Bitwise AND
- `0011`: Bitwise OR
- `0100`: Bitwise XOR
- `0101`: Bitwise NOT for A
- `0110`: Bitwise NOT for B
- `0111`: Clear output

Code:

```
`timescale 1ns / 1ps

module alu(
    input [15:0] A, B,
    input [3:0] ALU_Sel,
    output reg [15:0] ALU_Out,
    output reg CarryOut
    );

    reg [15:0] ALU_Result;
    wire [16:0] tmp;

    assign tmp = {1'b0, A} + {1'b0, B};
    assign CarryOut = tmp[16];

    always @* begin
        case (ALU_Sel)
            4'b0000: begin // ADD
                ALU_Result = A + B;
            end
            4'b0001: begin // SUB
                ALU_Result = A - B;
            end
            4'b0010: ALU_Result = A & B; // AND
            4'b0011: ALU_Result = A | B; // OR
            4'b0100: ALU_Result = A ^ B; // XOR
            4'b0101: ALU_Result = ~A;    // NOT for A
            4'b0110: ALU_Result = ~B;    // NOT for B
            4'b0111: ALU_Result = 0;     // Clear
            default: ALU_Result = A; // Default: A
        endcase
    end

    assign ALU_Out = ALU_Result;

endmodule
```

1. The addition operation is performed using both the assign statement and within the case statement to illustrate two different methods.
2. The case statement handles all possible operation cases based on the `ALU_Sel` input.
3. The default case is set to pass the value of A when an unrecognized operation is specified.
4. The `ALU_Out` is assigned the value of `ALU_Result` to provide the result of the selected operation.

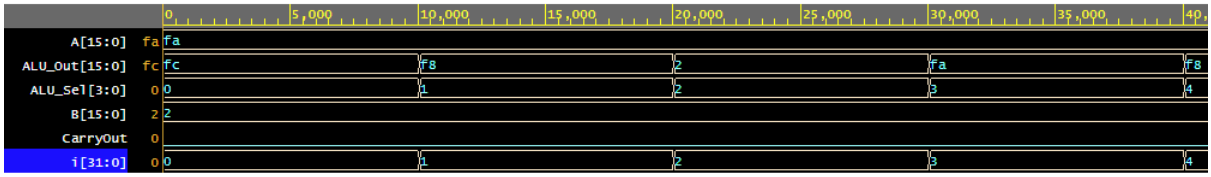
Testbench:

```
`timescale 1ns / 1ps

module test_alu;
    // Inputs
    reg[15:0] A, B;
    reg[3:0] ALU_Sel;
    // Outputs
    wire[15:0] ALU_Out;
    wire CarryOut;

    integer i;
    alu test_unit(A, B, ALU_Sel, ALU_Out, CarryOut);

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars(1, alu_testbench);
        A = 16'hFA;
        B = 16'h02;
        ALU_Sel = 4'h0;
        for (i = 0; i < 8; i = i + 1) begin
            #10;

            // Display operation name, inputs, and ALU_Out value after operation
            case (ALU_Sel)
                4'h0: $display("Operation: ADD, A = %h, B = %h, ALU_Out = %h, CarryOut = %b", A, B,
ALU_Out, CarryOut);
                4'h1: $display("Operation: SUB, A = %h, B = %h, ALU_Out = %h, CarryOut = %b", A, B,
ALU_Out, CarryOut);
                // Add cases for other operations similarly
            endcase

            // Increment ALU_Sel for the next operation
            ALU_Sel = ALU_Sel + 4'h1;
        end
        $finish; // End simulation after loop
    end
endmodule
```

Console Output:

```
Operation: ADD, A = 00fa, B = 0002, ALU_Out = 00fc, CarryOut = 0
Operation: SUB, A = 00fa, B = 0002, ALU_Out = 00f8, CarryOut = 0
```

EPWave:



2.2.2 Decoder

Overview:

A decoder is a digital circuit that converts an n-bit binary code into a 2^n line output. In this 16-bit decoder, the input in is an n-bit binary code, and the output out is a 2^n-bit vector where only one bit is set to '1' based on the input value.

Inputs:

`in`: An n-bit input that represents the binary code to be decoded. **Outputs:**

`out`: A 2^n-bit vector where only the bit corresponding to the input value is set to '1'.

Parameters:

`ENCODE_WIDTH`: A parameter that determines the width of the input (in) and the number of outputs (out). **Local Parameters:**

`latency`: A local parameter set to 1, indicating the latency of the assignment operation.

Operation Execution:

The assignment statement uses the shift-left (<<) operator to set the bit at the position specified by the value of in to '1'. All other bits remain '0'.

Code:

```
`timescale 1 ns / 1 ps

module decoder #(parameter ENCODE_WIDTH = 4) (
    input  [ENCODE_WIDTH-1:0] in,
    output [2**ENCODE_WIDTH-1:0] out
);

    localparam latency = 1;

    assign #latency out = 'b1 << in;

endmodule
```

1. The localparam statement defines a local parameter latency set to 1, indicating a single

time unit of delay for the assignment operation.

2. The assign statement performs a bitwise shift operation (¡¡) to set the output bit corresponding to the binary value of in to '1'. This effectively decodes the binary input into a one-hot representation.

3. The parameter `ENCODE_WIDTH` determines the width of the input and the number of output bits. If `ENCODE_WIDTH` is 4, then the output vector will have $2^4 = 16$ bits.

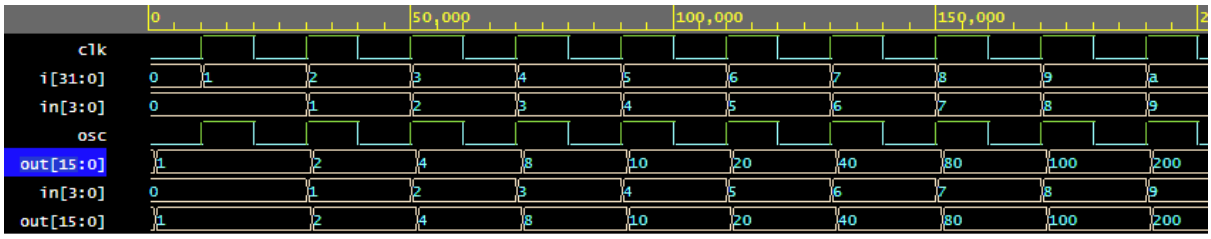
Testbench:

```
`timescale 1 ns / 1 ps

module test_decoder;

  parameter ENCODE_WIDTH = 4;
  parameter DECODE_WIDTH = 2**ENCODE_WIDTH;

  reg osc;
  reg [ENCODE_WIDTH-1:0] in;
  reg [DECODE_WIDTH-1:0] out;

  localparam period = 10;

  wire clk;
  assign clk = osc;

  decoder #(.ENCODE_WIDTH(ENCODE_WIDTH)) u0 (
      .in(in),
      .out(out)
  );

  integer i;

  always begin  // Clock wave
     #period  osc = ~osc;
  end

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    {osc, in} <= 0;

    for (i = 0; i < 16; i = i + 1) begin
      @(posedge clk) in = i;
      $display("Input in = %0d, Output out = %0d", in, out);
    end

    #(period * 16) $finish;
  end

endmodule
```

Console Output:

```
Input in = 0, Output out = 1
Input in = 1, Output out = 1
Input in = 2, Output out = 2
Input in = 3, Output out = 4
Input in = 4, Output out = 8
Input in = 5, Output out = 16
Input in = 6, Output out = 32
Input in = 7, Output out = 64
Input in = 8, Output out = 128
Input in = 9, Output out = 256
Input in = 10, Output out = 512
Input in = 11, Output out = 1024
Input in = 12, Output out = 2048
Input in = 13, Output out = 4096
Input in = 14, Output out = 8192
Input in = 15, Output out = 16384
```

EPWave:



2.2.3 Ram

Overview:

A single-port synchronous RAM module is designed to store and retrieve data based on the address provided. It is synchronous, meaning that data is read and written on clock edges. This RAM module is parameterized with options for address width (`ADDR_WIDTH`), data width (`DATA_WIDTH`), and length (`LENGTH`) of the memory.

Parameters:

- `ADDR_WIDTH`: Parameter specifying the width of the address bus.
- `DATA_WIDTH`: Parameter specifying the width of the data bus.
- `LENGTH`: Parameter specifying the length of the memory, calculated as $2^{(}$ADDR_WIDTH).

Inputs:

- `clk`: Clock input for synchronous operation.
- `addr`: Address bus indicating the location in memory.
- `data`: Bidirectional data bus for read and write operations.
- `cs`: Chip select signal.
- `we`: Write enable signal.
- `oe`: Output enable signal.

Outputs:

- `tmp_data`: Temporary storage for data during read operations.
- `mem`: Memory array to store data.

Write Operation:

On the positive edge of the clock (posedge clk), if the chip select (cs) and write enable (we) signals are active, the data at the specified address (addr) is updated with the input data.

Read Operation:

On the negative edge of the clock (negedge clk), if the chip select (cs) is active and write enable (we) is inactive, the data at the specified address (addr) is loaded into the temporary storage (`tmp_data`).

Data Output:

The assign statement determines the output data based on chip select (cs), output enable (oe), and write enable (we). If the chip select and output enable are active while write enable is inactive, the output data is set to the temporary data (`tmp_data`). Otherwise, it is set to 'bz' (high-impedance).

Code:

```
`timescale 1 ns / 1 ps

module single_port_sync_ram
  # (parameter ADDR_WIDTH = 16,
     parameter DATA_WIDTH = 8,
     parameter LENGTH = (1<<ADDR_WIDTH)
    )

  (    input clk,
       input [ADDR_WIDTH-1:0] addr,
       inout [DATA_WIDTH-1:0] data,
       input cs,
       input we,
       input oe
  );

  reg [DATA_WIDTH-1:0] tmp_data;
  reg [DATA_WIDTH-1:0] mem[LENGTH];

  always @ (posedge clk) begin
    // Used when debugging a multiplication program
    // if (addr=='h10D) $display("Product is %d", mem[addr]);
    if (cs & we)
      mem[addr] <= data;
  end

  always @ (negedge clk) begin
    if (cs & !we)
      tmp_data <= mem[addr];
  end

  assign data = cs & oe & !we ? tmp_data : 'bz; //changed from hz to bz


endmodule
```

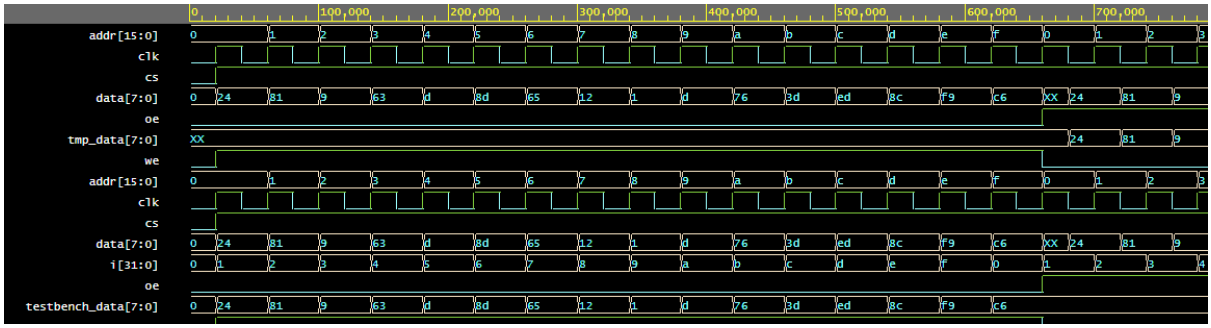
Testbench:

```
`timescale 1 ns / 1 ps

module test_ram;

  reg clk;
  reg cs;
  reg we;
  reg oe;
  reg [15:0] addr;
  wire [7:0] data;
  reg [7:0] testbench_data;

  single_port_sync_ram #(.ADDR_WIDTH(16), .DATA_WIDTH(8)) test
  (
    .clk(clk),
    .addr(addr),
    .data(data),
    .cs(cs),
    .we(we),
    .oe(oe)
  );


  always #20 clk = ~clk;
  assign data = !oe ? testbench_data : 'bz;

  integer i;
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    {clk, cs, we, addr, testbench_data, oe} <= 0;

    for(i = 0; i < 16; i = i+1) begin
        repeat (1) @(posedge clk) begin
        addr <= i;
        we <= 1;
        cs <= 1;
        oe <= 0;
        testbench_data <= $random;
        $display("Writing data %h to address %d", testbench_data, i);  // Show the write Operation (we =
1)
      end
    end

    for (i = 0; i < 16; i = i+1) begin
      repeat (1) @(posedge clk) begin
        addr <= i;
        we <= 0;
        cs <= 1;
        oe <= 1;
        $display("Reading data %h from address %d", data, i); // Show the read operation (we = 0)
      end
    end

    @(posedge clk) cs <= 0;

    #30 $finish;
  end
endmodule
```

Console Output:

```
Writing data 00 to address          0
Writing data 24 to address          1
Writing data 81 to address          2
Writing data 09 to address          3
Writing data 63 to address          4
Writing data 0d to address          5
Writing data 8d to address          6
Writing data 65 to address          7
Writing data 12 to address          8
Writing data 01 to address          9
Writing data 0d to address         10
Writing data 76 to address         11
Writing data 3d to address         12
Writing data ed to address         13
Writing data 8c to address         14
Writing data f9 to address         15
Reading data c6 from address          0
Reading data 24 from address          1
Reading data 81 from address          2
Reading data 09 from address          3
Reading data 63 from address          4
Reading data 0d from address          5
Reading data 8d from address          6
Reading data 65 from address          7
Reading data 12 from address          8
Reading data 01 from address          9
Reading data 0d from address         10
Reading data 76 from address         11
Reading data 3d from address         12
Reading data ed from address         13
Reading data 8c from address         14
Reading data f9 from address         15
```

EPWave:


addr[15:0]
clk
cs
data[7:0]
oe
tmp_data[7:0]
we
addr[15:0]
clk
cs
data[7:0]
i[31:0]
oe
testbench_data[7:0]

2.2.4 Large Ram

Overview:

This module is designed for a 16-bit system and includes a larger single-port synchronous RAM composed of multiple smaller RAMs. It uses a decoder to generate chip select signals for each smaller RAM module based on a subset of the address bits. **Parameters:**

- `ADDR_WIDTH`: Parameter specifying the width of the address bus.
- `DATA_WIDTH`: Parameter specifying the width of the data bus.
- `DATA_WIDTH_SHIFT`: Parameter determining the shift amount for splitting the data bus into two parts.

Inputs:

- `clk`: Clock input for synchronous operation.
- `addr`: Address bus indicating the location in memory.
- `data`: Bidirectional data bus for read and write operations.
- `cs`: Chip select signal.
- `we`: Write enable signal.
- `oe`: Output enable signal.

Internal Wires:

`cs`: 4-bit wire representing chip select signals generated by the decoder.

Decoder:

An instance of the decoder module is used to decode a subset of address bits (`addr[ADDR_WIDTH-1:ADDR_WIDT`
into chip select signals (`cs[3:0]`).

Smaller RAM Modules:

Four instances each of smaller single-port synchronous RAM modules (`u00, u01, ..., u31`) are instantiated to form the larger RAM. Each instance corresponds to a different subset of address bits and has its own chip select signal (`cs[0], cs[1], cs[2], cs[3]`).

Code:

```
`include "ram.sv"
`include "decoder.sv"

`timescale 1 ns / 1 ps

module single_port_sync_ram_large
  # ( parameter ADDR_WIDTH = 18,
      parameter DATA_WIDTH = 16,
      parameter DATA_WIDTH_SHIFT = 1
    )

  (   input clk,
      input [ADDR_WIDTH-1:0] addr,
      inout [DATA_WIDTH-1:0] data,
      input cs_input,
      input we,
      input oe
  );

  wire [3:0] cs;

  decoder #(.ENCODE_WIDTH(2)) dec
  (   .in(addr[ADDR_WIDTH-1:ADDR_WIDTH-2]),
      .out(cs)
  );

  single_port_sync_ram  #(.DATA_WIDTH(DATA_WIDTH/2)) u00
  (   .clk(clk),
      .addr(addr[ADDR_WIDTH-3:0]),
      .data(data[(DATA_WIDTH>>DATA_WIDTH_SHIFT)-1:0]),
      .cs(cs[0]),
      .we(we),
      .oe(oe)
  );
  single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH>>DATA_WIDTH_SHIFT)) u01
  (   .clk(clk),
      .addr(addr[ADDR_WIDTH-3:0]),
      .data(data[DATA_WIDTH-1:DATA_WIDTH>>DATA_WIDTH_SHIFT]),
      .cs(cs[0]),
      .we(we),
      .oe(oe)
  );

  single_port_sync_ram  #(.DATA_WIDTH(DATA_WIDTH/2)) u10
  (   .clk(clk),
      .addr(addr[ADDR_WIDTH-3:0]),
      .data(data[(DATA_WIDTH>>DATA_WIDTH_SHIFT)-1:0]),
      .cs(cs[1]),
      .we(we),
      .oe(oe)
  );
  single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH>>DATA_WIDTH_SHIFT)) u11
  (   .clk(clk),
      .addr(addr[ADDR_WIDTH-3:0]),
      .data(data[DATA_WIDTH-1:DATA_WIDTH>>DATA_WIDTH_SHIFT]),
      .cs(cs[1]),
      .we(we),
      .oe(oe)
  );

  single_port_sync_ram  #(.DATA_WIDTH(DATA_WIDTH/2)) u20
  (   .clk(clk),
      .addr(addr[ADDR_WIDTH-3:0]),
      .data(data[(DATA_WIDTH>>DATA_WIDTH_SHIFT)-1:0]),
      .cs(cs[2]),
      .we(we),
      .oe(oe)
  );
  single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH>>DATA_WIDTH_SHIFT)) u21
  (   .clk(clk),
      .addr(addr[ADDR_WIDTH-3:0]),
      .data(data[DATA_WIDTH-1:DATA_WIDTH>>DATA_WIDTH_SHIFT]),
      .cs(cs[2]),
      .we(we),
      .oe(oe)
  );

  single_port_sync_ram  #(.DATA_WIDTH(DATA_WIDTH/2)) u30
  (   .clk(clk),
      .addr(addr[ADDR_WIDTH-3:0]),
      .data(data[(DATA_WIDTH>>DATA_WIDTH_SHIFT)-1:0]),
      .cs(cs[3]),
      .we(we),
      .oe(oe)
  );
  single_port_sync_ram #(.DATA_WIDTH(DATA_WIDTH>>DATA_WIDTH_SHIFT)) u31
  (   .clk(clk),
      .addr(addr[ADDR_WIDTH-3:0]),
      .data(data[DATA_WIDTH-1:DATA_WIDTH>>DATA_WIDTH_SHIFT]),
      .cs(cs[3]),
      .we(we),
      .oe(oe)
  );

endmodule
```

Testbench:

```
`timescale 1 ns / 1 ps

module test_ram_large;
  parameter ADDR_WIDTH = 18;
  parameter DATA_WIDTH = 16;

  reg clk;
  reg cs;
  reg we;
  reg oe;
  reg [ADDR_WIDTH-1:0] addr;
  wire [DATA_WIDTH-1:0] data;
  reg [DATA_WIDTH-1:0] testbench_data;

  single_port_sync_ram_large  #(.DATA_WIDTH(DATA_WIDTH)) u0
  (   .clk(clk),
      .addr(addr),
      .data(data[DATA_WIDTH-1:0]),
      .cs_input(cs),
      .we(we),
      .oe(oe)
  );


  always #20 clk = ~clk;
  assign data = !oe ? testbench_data : 'hz;

  integer i;
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    {clk, cs, we, addr, testbench_data, oe} <= 0;

    repeat (2) @ (posedge clk);

    // Write
    for (i = 2**(ADDR_WIDTH-2)-4; i < 2**(ADDR_WIDTH-2); i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 1; cs <= 1; oe <= 0; testbench_data <= $random;
    end

    for (i = 2**(ADDR_WIDTH-1)-4; i < 2**(ADDR_WIDTH-1); i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 1; cs <= 1; oe <= 0; testbench_data <= $random;
    end

    for (i = 2**(ADDR_WIDTH-1)+2**(ADDR_WIDTH-2)-4; i < 2**(ADDR_WIDTH-1)+2**(ADDR_WIDTH-2); i = i+1)
begin
      repeat (1) @(posedge clk) addr <= i; we <= 1; cs <= 1; oe <= 0; testbench_data <= $random;
    end

    for (i = 2**ADDR_WIDTH-4; i < 2**ADDR_WIDTH-4; i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 1; cs <= 1; oe <= 0; testbench_data <= $random;
    end

    // Read
    for (i = 2**(ADDR_WIDTH-2)-4; i < 2**(ADDR_WIDTH-2); i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 0; cs <= 1; oe <= 1;
    end

    for (i = 2**(ADDR_WIDTH-1)-4; i < 2**(ADDR_WIDTH-1); i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 0; cs <= 1; oe <= 1;
    end

    for (i = 2**(ADDR_WIDTH-1)+2**(ADDR_WIDTH-2)-4; i < 2**(ADDR_WIDTH-1)+2**(ADDR_WIDTH-2); i = i+1)
begin
      repeat (1) @(posedge clk) addr <= i; we <= 0; cs <= 1; oe <= 1;
    end

    for (i = 2**ADDR_WIDTH-4; i < 2**ADDR_WIDTH-4; i = i+1) begin
      repeat (1) @(posedge clk) addr <= i; we <= 0; cs <= 1; oe <= 1;
    end

    #40 $finish;
  end
endmodule
```

EPWave:

2.2.5 CPU

Overview:

The module represents a basic 16-bit CPU with a simple instruction set architecture. The CPU fetches, decodes, and executes instructions stored in memory. The memory is implemented using the `single_port_sync_ram_large module`, and arithmetic/logic operations are performed using the alu module. The CPU executes a Fibonacci sequence calculation as an example program. **Parameters:**

- `ADDR_WIDTH`: Parameter specifying the width of the address bus.
- `DATA_WIDTH`: Parameter specifying the width of the data bus.

Clock Generation: The module includes a clock (clk) derived from an oscillator (osc). The clock waveform is generated with a period of 10 time units, using a toggling osc signal. **Memory (RAM):** An instance of the `single_port_sync_ram_large` module is instantiated to represent the memory (RAM) for storing program instructions and data. The RAM is 16 bits wide, and the address width is 18 bits. **ALU (Arithmetic Logic Unit):** An instance of the alu module (alu16) is instantiated to perform arithmetic and logic operations. It takes two 16-bit inputs (A and B) and produces a 16-bit output (`ALU_Out`). The ALU selection (`ALU_Sel`) is controlled by the CPU instructions. **Registers:** Several registers are defined for the CPU, including:

- MAR (Memory Address Register)
- data (Data Bus)
- `testbench_data` (Testbench Data)
- A and B (ALU Inputs)
- `ALU_Out` (ALU Output)
- PC (Program Counter)
- IR (Instruction Register)
- MBR (Memory Buffer Register)

- AC (Accumulator)

Instruction Execution Loop: The CPU includes an initial block that initializes the clock waveform and sets up the program for Fibonacci sequence calculation. A loop is implemented for instruction execution, where instructions are fetched, decoded, and executed in a sequential manner. The program uses a simple instruction set, where each instruction is a 16-bit word.

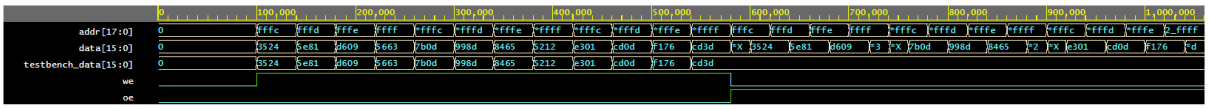
Instruction Set: The instruction set includes operations such as load, store, add, subtract, logical AND, logical OR, halt, skip, jump, and clear accumulator.

Code:

```
`timescale 1 ns / 1 ps

module test_cpu;
  parameter ADDR_WIDTH = 18;
  parameter DATA_WIDTH = 16;

  reg osc;
  localparam period = 10;

  wire clk;
  assign clk = osc;

  reg cs;
  reg we;
  reg oe;
  integer i;
  reg [ADDR_WIDTH-1:0] MAR;
  wire [DATA_WIDTH-1:0] data;
  reg [DATA_WIDTH-1:0] testbench_data;
  assign data = !oe ? testbench_data : 'hz;

  single_port_sync_ram_large  #(.DATA_WIDTH(DATA_WIDTH)) ram
  (    .clk(clk),
   .addr(MAR),
      .data(data[DATA_WIDTH-1:0]),
      .cs_input(cs),
      .we(we),
      .oe(oe)
  );

  reg [15:0] A;
  reg [15:0] B;
  reg [15:0] ALU_Out;
  reg [1:0] ALU_Sel;
  alu alu16(
    .A(A),
    .B(B),  // ALU 16-bit Inputs
    .ALU_Sel(ALU_Sel),// ALU Selection
    .ALU_Out(ALU_Out) // ALU 16-bit Output
     );

  reg [15:0] PC = 'h100;
  reg [15:0] IR = 'h0;
  reg [15:0] MBR = 'h0;
  reg [15:0] AC = 'h0;

  initial osc = 1;  //init clk = 1 for positive-edge triggered
  always begin  // Clock wave
     #period  osc = ~osc;
  end

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    // Fibonacci 11 program
    @(posedge clk) MAR <= 'h0000100; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000005;
    @(posedge clk) MAR <= 'h0000102; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000006;
    @(posedge clk) MAR <= 'h0000104; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10110007;
    @(posedge clk) MAR <= 'h0000106; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000006;
    @(posedge clk) MAR <= 'h0000108; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10110004;
    @(posedge clk) MAR <= 'h000010A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000007;
    @(posedge clk) MAR <= 'h000010C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10110006;
    @(posedge clk) MAR <= 'h000010E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000008;
    @(posedge clk) MAR <= 'h0000110; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000000F;
    @(posedge clk) MAR <= 'h0000112; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10110008;
    @(posedge clk) MAR <= 'h0000114; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h11004000;
    @(posedge clk) MAR <= 'h0000116; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000000;
    @(posedge clk) MAR <= 'h0000118; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10010000;
    @(posedge clk) MAR <= 'h000011A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000000;
    @(posedge clk) MAR <= 'h000011C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000001;
    @(posedge clk) MAR <= 'h000011E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000000;
    @(posedge clk) MAR <= 'h0000120; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h0000000A;
    @(posedge clk) MAR <= 'h0000122; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'hFFFF00FF;
```

```
@(posedge clk) PC <= 'h0000100;

    for (i = 0; i < 62; i = i+1) begin
          // Fetch
          @(posedge clk) MAR <= PC; we <= 0; cs <= 1; oe <= 1;
          @(posedge clk) IR <= data;
          @(posedge clk) PC <= PC + 1;
          // Decode and execute
      case(IR[15:12])
        4'b0001: begin//load
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) AC <= MBR;
        end
            4'b0010: begin//store
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= AC;
              @(posedge clk) we <= 1; oe <= 0; testbench_data <= MBR;
              #20;
              #1;
        end
        4'b0011: begin//add
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) ALU_Sel <= 'b01; A <= AC; B <= MBR;
              @(posedge clk) AC <= ALU_Out;
        end
        4'b0100: begin//subtract
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) ALU_Sel <= 'b10; A <= AC; B <= MBR;
              @(posedge clk) AC <= ALU_Out;
        end
        4'b0101: begin//and
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) ALU_Sel <= 'b11; A <= AC; B <= MBR;
              @(posedge clk) AC <= ALU_Out;
        end
        4'b0110: begin//or
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) ALU_Sel <= 'b00; A <= AC; B <= MBR;
              @(posedge clk) AC <= ALU_Out;
        end
        4'b0111: begin//halt
              @(posedge clk) PC <= PC - 1;
        end
        4'b1000: begin//skip
          @(posedge clk)
          if(IR[11:10]==2'b01 && AC == 0) PC <= PC + 1;
          else if(IR[11:10]==2'b00 && AC < 0) PC <= PC + 1;
          else if(IR[11:10]==2'b10 && AC > 0) PC <= PC + 1;
        end
        4'b1001: begin //jump
              @(posedge clk) PC <= IR[11:0];
        end
        4'b1010: begin
          @(posedge clk) AC <= 0;
        end

      endcase
    end

    @(posedge clk) MAR  <= 'h10110004; we <= 0; cs <= 1; oe <= 1;
    @(posedge clk);

    #20 $finish;

  end

endmodule
```

Testbench: EPWave:



3 Design 1a

CPU with indirect addressing.

```
`timescale 1 ns / 1 ps

module test_cpu;
  parameter ADDR_WIDTH = 18;
  parameter DATA_WIDTH = 16;

  reg osc;
  localparam period = 10;

  wire clk;
  assign clk = osc;

  reg cs;
  reg we;
  reg oe;
  integer i;
  reg [ADDR_WIDTH-1:0] MAR;
  wire [DATA_WIDTH-1:0] data;
  reg [DATA_WIDTH-1:0] testbench_data;
  assign data = !oe ? testbench_data : 'hz;

  single_port_sync_ram_large  #(.DATA_WIDTH(DATA_WIDTH)) ram
  (    .clk(clk),
   .addr(MAR),
      .data(data[DATA_WIDTH-1:0]),
      .cs_input(cs),
      .we(we),
      .oe(oe)
  );

  reg [15:0] A;
  reg [15:0] B;
  reg [15:0] ALU_Out;
  reg [1:0] ALU_Sel;
  alu alu16(
    .A(A),
    .B(B),  // ALU 16-bit Inputs
    .ALU_Sel(ALU_Sel),// ALU Selection
    .ALU_Out(ALU_Out) // ALU 16-bit Output
     );

  reg [15:0] PC = 'h100;
  reg [15:0] IR = 'h0;
  reg [15:0] MBR = 'h0;
  reg [15:0] AC = 'h0;

  initial osc = 1;  //init clk = 1 for positive-edge triggered
  always begin  // Clock wave
     #period  osc = ~osc;
  end

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    // Fibonacci 11 program with indirect addressing
    @(posedge clk) MAR <= 'h0000100; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000005;
    @(posedge clk) MAR <= 'h0000102; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000006;
    @(posedge clk) MAR <= 'h0000104; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10110007;
    @(posedge clk) MAR <= 'h0000106; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000006;
    @(posedge clk) MAR <= 'h0000108; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10110004;
    @(posedge clk) MAR <= 'h000010A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000007;
    @(posedge clk) MAR <= 'h000010C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10110006;
    @(posedge clk) MAR <= 'h000010E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000008;
    @(posedge clk) MAR <= 'h0000110; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000000F;
    @(posedge clk) MAR <= 'h0000112; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10110008;
    @(posedge clk) MAR <= 'h0000114; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h11004000;
    @(posedge clk) MAR <= 'h0000116; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000000;
    @(posedge clk) MAR <= 'h0000118; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10010000;
    @(posedge clk) MAR <= 'h000011A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h0000124;
    @(posedge clk) MAR <= 'h000011C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h0000126;
    @(posedge clk) MAR <= 'h000011E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h0000128;
    @(posedge clk) MAR <= 'h0000120; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h000012A;
    @(posedge clk) MAR <= 'h0000122; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h000012C;
```

```
@(posedge clk) MAR <= 'h0000124; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000000;
    @(posedge clk) MAR <= 'h0000126; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000001;
    @(posedge clk) MAR <= 'h0000128; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000000;
    @(posedge clk) MAR <= 'h000012A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00001010;
    @(posedge clk) MAR <= 'h000012C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h11111111;

    @(posedge clk) PC <= 'h0000100;

    for (i = 0; i < 62; i = i+1) begin
          // Fetch
          @(posedge clk) MAR <= PC; we <= 0; cs <= 1; oe <= 1;
          @(posedge clk) IR <= data;
          @(posedge clk) PC <= PC + 1;
          // Decode and execute
      case(IR[15:12])
        4'b0001: begin//load
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) AC <= MBR;
        end
            4'b0010: begin//store
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= AC;
              @(posedge clk) we <= 1; oe <= 0; testbench_data <= MBR;
              #20;
              #1;
        end
        4'b0011: begin//add
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) ALU_Sel <= 'b01; A <= AC; B <= MBR;
              @(posedge clk) AC <= ALU_Out;
        end
        4'b0100: begin//subtract
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) ALU_Sel <= 'b10; A <= AC; B <= MBR;
              @(posedge clk) AC <= ALU_Out;
        end
        4'b0101: begin//and
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) ALU_Sel <= 'b11; A <= AC; B <= MBR;
              @(posedge clk) AC <= ALU_Out;
        end
        4'b0110: begin//or
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) ALU_Sel <= 'b00; A <= AC; B <= MBR;
              @(posedge clk) AC <= ALU_Out;
        end
        4'b0111: begin//halt
              @(posedge clk) PC <= PC - 1;
        end
        4'b1000: begin//skip
          @(posedge clk)
          if(IR[11:10]==2'b01 && AC == 0) PC <= PC + 1;
          else if(IR[11:10]==2'b00 && AC < 0) PC <= PC + 1;
          else if(IR[11:10]==2'b10 && AC > 0) PC <= PC + 1;
        end
        4'b1001: begin //jump
              @(posedge clk) PC <= IR[11:0];
        end
        4'b1010: begin
          @(posedge clk) AC <= 0;
        end

      endcase
    end

    @(posedge clk) MAR  <= 'h10110004; we <= 0; cs <= 1; oe <= 1;
    @(posedge clk);

    #20 $finish;

  end

endmodule
```

4 Design 1b

```
`timescale 1 ns / 1 ps

module test_cpu;
  parameter ADDR_WIDTH = 18;
  parameter DATA_WIDTH = 16;

  reg osc;
  localparam period = 10;

  wire clk;
  assign clk = osc;

  reg cs;
  reg we;
  reg oe;
  integer i;
  reg [ADDR_WIDTH-1:0] MAR;
  wire [DATA_WIDTH-1:0] data;
  reg [DATA_WIDTH-1:0] testbench_data;
  assign data = !oe ? testbench_data : 'hz;

  single_port_sync_ram_large  #(.DATA_WIDTH(DATA_WIDTH)) ram
  (    .clk(clk),
   .addr(MAR),
      .data(data[DATA_WIDTH-1:0]),
      .cs_input(cs),
      .we(we),
      .oe(oe)
  );

  Cache cache(
        .clk(clk),
        .addr(cache_addr),
        .write(write),
        .data_in(cache_data),
        .found(cache_out),
        .hit(hit)
  );

  reg [15:0] A;
  reg [15:0] B;
  reg [15:0] ALU_Out;
  reg [1:0] ALU_Sel;
  alu alu16(
    .A(A),
    .B(B),  // ALU 16-bit Inputs
    .ALU_Sel(ALU_Sel),// ALU Selection
    .ALU_Out(ALU_Out) // ALU 16-bit Output
     );

  reg [15:0] PC = 'h100;
  reg [15:0] IR = 'h0;
  reg [15:0] MBR = 'h0;
  reg [15:0] AC = 'h0;

  initial osc = 1;  //init clk = 1 for positive-edge triggered
  always begin  // Clock wave
     #period  osc = ~osc;
  end

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    // Fibonacci 11 program
    @(posedge clk) MAR <= 'h0000100; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000005;
    @(posedge clk) MAR <= 'h0000102; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000006;
    @(posedge clk) MAR <= 'h0000104; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10110007;
    @(posedge clk) MAR <= 'h0000106; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000006;
    @(posedge clk) MAR <= 'h0000108; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10110004;
    @(posedge clk) MAR <= 'h000010A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000007;
    @(posedge clk) MAR <= 'h000010C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10110006;
    @(posedge clk) MAR <= 'h000010E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000008;
    @(posedge clk) MAR <= 'h0000110; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h1000000F;
    @(posedge clk) MAR <= 'h0000112; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10110008;
    @(posedge clk) MAR <= 'h0000114; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h11004000;
    @(posedge clk) MAR <= 'h0000116; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10000000;
    @(posedge clk) MAR <= 'h0000118; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h10010000;
    @(posedge clk) MAR <= 'h000011A; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000000;
    @(posedge clk) MAR <= 'h000011C; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000001;
    @(posedge clk) MAR <= 'h000011E; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h00000000;
    @(posedge clk) MAR <= 'h0000120; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'h0000000A;
    @(posedge clk) MAR <= 'h0000122; we <= 1; cs <= 1; oe <= 0; testbench_data <= 'hFFFF00FF;
```

```
@(posedge clk) PC <= 'h0000100;

    for (i = 0; i < 62; i = i+1) begin
          // Fetch
          @(posedge clk) MAR <= PC; we <= 0; cs <= 1; oe <= 1;
          @(posedge clk) IR <= data;
          @(posedge clk) PC <= PC + 1;
          // Decode and execute
      case(IR[15:12])
        4'b0001: begin//load
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) AC <= MBR;
        end
            4'b0010: begin//store
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= AC;
              @(posedge clk) we <= 1; oe <= 0; testbench_data <= MBR;
              #20;
              #1;
        end
        4'b0011: begin//add
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) ALU_Sel <= 'b01; A <= AC; B <= MBR;
              @(posedge clk) AC <= ALU_Out;
        end
        4'b0100: begin//subtract
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) ALU_Sel <= 'b10; A <= AC; B <= MBR;
              @(posedge clk) AC <= ALU_Out;
        end
        4'b0101: begin//and
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) ALU_Sel <= 'b11; A <= AC; B <= MBR;
              @(posedge clk) AC <= ALU_Out;
        end
        4'b0110: begin//or
              @(posedge clk) MAR <= IR[11:0];
              @(posedge clk) MBR <= data;
              #20;
              #1;
              @(posedge clk) ALU_Sel <= 'b00; A <= AC; B <= MBR;
              @(posedge clk) AC <= ALU_Out;
        end
        4'b0111: begin//halt
              @(posedge clk) PC <= PC - 1;
        end
        4'b1000: begin//skip
          @(posedge clk)
          if(IR[11:10]==2'b01 && AC == 0) PC <= PC + 1;
          else if(IR[11:10]==2'b00 && AC < 0) PC <= PC + 1;
          else if(IR[11:10]==2'b10 && AC > 0) PC <= PC + 1;
        end
        4'b1001: begin //jump
              @(posedge clk) PC <= IR[11:0];
        end
        4'b1010: begin
          @(posedge clk) AC <= 0;
        end

      endcase
    end

    @(posedge clk) MAR  <= 'h10110004; we <= 0; cs <= 1; oe <= 1;
    @(posedge clk);

    #20 $finish;

  end

endmodule
```

4.1 Data Cache Implementation

In the updated CPU design, a data cache has been seamlessly integrated using a dedicated module called ‘Cache‘. This cache module acts as an intermediary between the CPU and the

main memory (‘ram‘ module), offering a faster data access mechanism for frequently used information. The cache is equipped with input and output ports, such as ‘clk‘ (clock), ‘addr‘ (address), ‘write‘ (write enable), ‘`data_in`‘ (input data), ‘found‘ (cache hit signal), and ‘hit‘ (hit line). The CPU leverages the cache by fetching data through it, and the resultant information is stored in the ‘data‘ signal for subsequent processing. The cache implementation includes hit detection logic (‘found‘ signal) to determine whether the requested data is present in the cache, optimizing data access by reducing latency associated with memory retrieval.

Within the CPU execution loop, the ‘data‘ signal is dynamically assigned based on the presence or absence of a cache hit. If a cache hit is detected, the data is directly sourced from the cache (‘`data_in`‘). On the other hand, in the event of a cache miss, the CPU fetches the required data from the main memory (‘ram‘ module). Additionally, the cache module facilitates write operations back to memory, controlled by the ‘write‘ signal. Overall, the integration of the data cache enhances the CPU’s performance by minimizing memory access times and streamlining the retrieval of frequently accessed data, contributing to improved overall system efficiency.

5 Testbench

5.1 Fibonacci F11

Code:

```
ORG 100             ; Fibonacci sequence calculation

    Load    FibPrev    ; Load the previous Fibonacci number into AC
    Add     FibCurr    ; Add the current Fibonacci number
    Store   FibNext    ; Store the new Fibonacci number

    Load    FibCurr    ; Load the current Fibonacci number into AC
    Store   FibPrev    ; Update the previous Fibonacci number

    Load    FibNext    ; Load the new Fibonacci number into AC
    Store   FibCurr    ; Update the current Fibonacci number

    Load    Ctr        ; Load the loop control variable
    Add     Neg1       ; Decrement the loop control variable by one
    Store   Ctr        ; Store the new value of the loop control variable

    Skipcond  400       ; If the control variable = 0, skip next instruction to terminate the loop
    Jump    Loop       ; Otherwise, go to Loop

    Halt               ; Terminate program

FibPrev, Dec   0       ; Previous Fibonacci number (initialized to 0)
FibCurr, Dec   1       ; Current Fibonacci number (initialized to 1)
FibNext, Dec   0       ; Next Fibonacci number
Ctr,      Dec   10      ; Loop control variable (for the 11th Fibonacci number)
Neg1,     Dec   -1      ; Used to increment and decrement by 1
```

Machine Code:

```
Address Byte1 Byte 2    Code
                      ORG 100              / Fibonnaci sequence

0100    1000   0005     Load FibPrev        / Address 105
0102    1000   0006     Add FibCurr         / Address 106
0104    1011   0007     Store FibNext       / Address 107
0106    1000   0006     Load FibCurr        / Address 106
0108    1011   0004     Store FibPrev       / Address 104
010A    1000   0007     Load FibNext        / Address 107
010C    1011   0006     Store FibCurr       / Address 106
010E    1000   0008     Load Ctr            / Address 108
0110    1000   000F     Add Neg1            / Address 10F
0112    1011   0008     Store Ctr           / Address 108
0114    1100   4000     Skipcond 400        / Skip if AC = 0
0116    1000   0000     Jump Loop           / Address 100
0118    1001   0000     Halt                / End program
011A    0000   0000     Dec 0               / FibPrev initialized to 0
011C    0000   0001     Dec 1               / FibCurr initialized to 1
011E    0000   0000     Dec 0               / FibNext initialized to 0
0120    0000   000A     Dec 10              / Loop control variable initialized to 10
0122    FFFF   00FF     Dec -1              / Neg1 initialized to -1
```

To translate the assembly code down into machine code, we used the opcode from our ISA for each instruction to assign them an address. Then we used the address to assign each instruction a binary value.

For example, using the first four lines of the assembly code this is how they were translated:

1. `ORG 100`: This instruction sets the origin of the program to address 100. The machine code at address 100 is 1000 0000 (1000 for the opcode ”ORG” and 0000 for the operand ”100”).

2. `Load FibPrev`: In the provided assembly language, the Load instruction is represented by the opcode 1000. The operand 0005 represents the address of the variable FibPrev. Therefore, the machine code for this line is 1000 0005 (1000 for ”Load” and 0005 for the operand).

3. `Add FibCurr`: The Add instruction is represented by the opcode 1000, and the operand 0006 corresponds to the address of the variable FibCurr. Thus, the machine code is 1000 0006.

4. `Store FibNext`: The Store instruction is represented by the opcode 1011, and the operand 0007 corresponds to the address of the variable FibNext. Therefore, the machine code is 1011 0007.

This process is repeated for each line of the assembly code, with each mnemonic and operand being translated into the corresponding machine code.