Southern Luzon State University
College of Engineering
Lucban, Quezon

CPE23L- COMPUTER ARCHITECTURE & ORGANIZATION

Final Project Documentation: Water Refilling Box

Submitted by:

Bolo, Lyza April P.
Carreos, Clarice Mae G.
Corpuz, Reimarc G.

BSCPE IV - IE

Submitted to:

Engr. Carla May C. Ceribo

Submitted on:

January 15, 2024

INTRODUCTION

Verilog, as a hardware description language (HDL), plays a crucial role in designing and simulating digital circuits. In building a Central Processing Unit (CPU) using Verilog, registers, including instruction registers (IR), play a key role in storing and managing data during instruction execution. This Verilog code uses registers strategically to make the CPU work better by improving how the data moves around and stays organized. The instruction register specifically holds the binary representation of the current instruction fetched from memory. Its primary function is to facilitate the decoding of instructions, determining the operation the CPU needs to perform.

The Verilog code outlines the CPU architecture, defining the interconnections of components like Arithmetic and Logical Units (ALUs), multiplexers, and control units. Registers act as a temporary storage for different types of data movements, ensuring smooth instruction execution. The register-transfer level (RTL) design paradigm is applied to describe various data movements and control flow, connecting high-level functionality with low-level hardware implementation.

Clock and reset signals are incorporated to synchronize register and CPU component operations. This Verilog code forms the basis for simulating, testing, and compiling a fully operational CPU, highlighting Verilog's effectiveness in digital circuit design.

Understanding the data movement through registers, and to further test them, the project that we want to create is the Water Refilling Box. A water refilling box or machine, also known as a water vending machine or water dispenser, is a device designed to provide a convenient way for people to access purified or filtered water. Users may need to pay using coins, bills, or electronic payment methods to access the purified water. The time that the machine will work depends on the coins the user will insert.

METHODOLOGY & DISCUSSION

The Verilog code provided describes a simple processor module named REFILLING that performs various arithmetic, logical, load/store, jump, and branch operations based on the instruction set defined using macros. The processor consists of a program memory (inst_mem), data memory (data_mem), a set of general-purpose registers (GPR), a special register (SGPR), and condition flags (sign, zero, overflow, and carry). The processor follows a finite state machine (FSM) approach to execute instructions.

The decode_inst task decodes and executes the instructions based on the opcode (oper_type). Arithmetic and logical operations (add, sub, mul, etc.), bitwise operations (ror, rand, rxor, etc.), load/store instructions (storereg, storedin, senddout, sendreg), jump and branch instructions (jump, jcarry, jsign, etc.), and a halt instruction (halt) are implemented. The decode_condflag task updates the condition flags based on the executed instruction, such as sign, zero, overflow, and carry flags.

The main logic is implemented in the initial block, which reads instructions from a memory file (refilling.mem) and executes them in a sequential manner. The FSM manages the execution flow, transitioning between states such as idle, fetch_inst, dec_exec_inst, delay_next_inst, next_inst, and sense_halt. The count variable is used to introduce a delay (delay_next_inst) to ensure proper handling of conditional jumps.

ASSEMBLY CODE ANALYSIS METHODOLOGY

1. Instruction Overview

- *MOV R1, #3;: Move the immediate value 3 into register R1.*
- *MOV R2, #2;: Move the immediate value 2 into register R2.*
- *MOV R3, #0;: Move the immediate value 0 into register R3.*

- *MOV R4, #2;: Move the immediate value 2 into register R4.*
- *ADD R3, R3, R1;: Add the value in register R1 to the value in register R3 and store the result in register R3.*
- *SUB R4, R4, #1;: Subtract the immediate value 1 from the value in register R4 and store the result in register R4.*
- *JNZ @4;: Jump to the address 4 if the zero flag is not set (i.e., if R4 is not zero).*
- *MOV R5, R3;: Move the value in register R3 to register R5.*
- *MOV R6, #0;: Move the immediate value 0 into register R6.*
- *MOV R6, R5;: Move the value in register R5 to register R6.*
- *MOV R7, #0;: Move the immediate value 0 into register R7.*
- *MOV R7, R6;: Move the value in register R6 to register R7.*
- *MOV R6, R6, #1;: Move the value in register R6 incremented by 1 back to register R6.*
- *JNS @11;: Jump to the address 11 if the negative flag is not set (i.e., if R6 is not negative).*
- *HALT: Halt the execution.*

2. Control Flow

- *The code uses conditional jumps based on the result of the subtraction operation (JNZ and JNS) to control the flow of execution.*
- *It enters a loop at address 4 (@4) until the value in register R4 becomes zero.*
- *It enters a loop at address 11 (@11) until the value in register R6 becomes zero.*
- *The code halts the execution when it reaches the HALT instruction.*

3. Registers and Variables

Registers and temporary values are used and stored in:

- *R1*
- *R2*
- *R3*
- *R4*
- *R5*
- *R6*
- *R7*

4. Data Flow

- *The code manipulates data using arithmetic operations (ADD, SUB) and transfers data between registers using move operations (MOV).*

5. Loop Structure

- *The loop at address 4 (@4) involves adding the value in R1 to R3, decrementing R4, and jumping back to address 4 if R4 is not zero.*
- *The loop at address 11 (@11) involves subtracting the value in R6 to R7. R6 is decrementing as it subtract by 1 until 0 product.*

6. Halt Condition

- *The program halts when the HALT instruction is encountered.*

7. Output

- *The final result or output is not explicitly mentioned. It might be the value stored in R7 or the state of registers at the halt point.*

The execution example provided states that if 3 pesos are inserted, and 1 peso is equivalent to 2 seconds, the time duration is 6 seconds. The desired output is a countdown sequence: 6, 5, 4, 3, 2, 1, 0.

VERILOG CODE

```
`timescale 1ns / 1ps

//////////fields of IR
`define oper_type    IR[31:27]
`define rdst         IR[26:22]
`define rsrc1        IR[21:17]
`define imm_mode     IR[16]
`define rsrc2        IR[15:11]
`define isrc         IR[15:0]


////////////////arithmetic operation
`define movsgpr      5'b00000
`define mov          5'b00001
`define add          5'b00010
`define sub          5'b00011
`define mul          5'b00100

////////////////logical operations : and or xor xnor nand nor not

`define ror          5'b00101
`define rand         5'b00110
`define rxor         5'b00111
`define rxnor        5'b01000
`define rnand        5'b01001
`define rnor         5'b01010
`define rnot         5'b01011

////////////////////// load & store instructions

`define storereg     5'b01101   //////store content of register in data memory
`define storedin     5'b01110   ////// store content of din bus in data memory
`define senddout      5'b01111   /////send data from DM to dout bus
`define sendreg       5'b10001   ////// send data from DM to register

////////////////////////////// Jump and branch instructions
`define jump          5'b10010  ////jump to address
`define jcarry        5'b10011  ////jump if carry
`define jnocarry      5'b10100
`define jsign         5'b10101  ////jump if sign
`define jnosign       5'b10110
`define jzero         5'b10111  //// jump if zero
`define jnozero       5'b11000
`define joverflow     5'b11001 ////jump if overflow
`define jnooverflow   5'b11010

//////////////////////////halt
`define halt          5'b11011
```

```
module REFILLING(
input clk,sys_rst,
input [15:0] din,
output reg [15:0] dout
);

////////////////adding program and data memory
reg [31:0] inst_mem [15:0]; ////program memory
reg [15:0] data_mem [15:0]; ////data memory


reg [31:0] IR;            ////// instruction register
<--ir[31:27]--><--ir[26:22]--><--ir[21:17]--><--ir[16]--><--ir[15:11]--><--ir[10:0]-->
                      //////fields               <---  oper  --><--   rdest --><--   rsrc1
--><--modesel--><--  rsrc2 --><--unused  -->
                      //////fields               <---  oper  --><--   rdest --><--   rsrc1
--><--modesel--><--  immediate_date      -->

reg [15:0] GPR [31:0] ;   ///////general purpose register gpr[0] ..... gpr[31]
reg [15:0] SGPR ;           ///// msb of multiplication --> special register
reg [31:0] mul_res;

reg sign = 0, zero = 0, overflow = 0, carry = 0; ///condition flag
reg [16:0] temp_sum;

reg jmp_flag = 0;
reg stop = 0;

task decode_inst();
 begin

jmp_flag = 1'b0;
stop     = 1'b0;

case(`oper_type)
////////////////////////////////
`movsgpr: begin
   GPR[`rdst] = SGPR;
end

////////////////////////////////
`mov : begin
   if(`imm_mode)
       GPR[`rdst]  = `isrc;
   else
      GPR[`rdst]   = GPR[`rsrc1];
end
```

```
//////////////////////////////////////////////////

`add : begin
     if(`imm_mode)
       GPR[`rdst]   = GPR[`rsrc1] + `isrc;
    else
       GPR[`rdst]   = GPR[`rsrc1] + GPR[`rsrc2];
end

//////////////////////////////////////////////////////

`sub : begin
     if(`imm_mode)
       GPR[`rdst]  = GPR[`rsrc1] - `isrc;
    else
      GPR[`rdst]   = GPR[`rsrc1] - GPR[`rsrc2];
end

//////////////////////////////////////////////////////////

`mul : begin
     if(`imm_mode)
       mul_res   = GPR[`rsrc1] * `isrc;
    else
       mul_res   = GPR[`rsrc1] * GPR[`rsrc2];

    GPR[`rdst]   =  mul_res[15:0];
    SGPR         =  mul_res[31:16];

end

////////////////////////////////////////////////////////////// bitwise or

`ror : begin
     if(`imm_mode)
       GPR[`rdst]  = GPR[`rsrc1] | `isrc;
    else
      GPR[`rdst]   = GPR[`rsrc1] | GPR[`rsrc2];
end

//////////////////////////////////////////////////////////////bitwise and

`rand : begin
     if(`imm_mode)
       GPR[`rdst]  = GPR[`rsrc1] & `isrc;
    else
      GPR[`rdst]   = GPR[`rsrc1] & GPR[`rsrc2];
end

////////////////////////////////////////////////////////// bitwise xor
```

```
`rxor : begin
     if(`imm_mode)
       GPR[`rdst]  = GPR[`rsrc1] ^ `isrc;
    else
      GPR[`rdst]   = GPR[`rsrc1] ^ GPR[`rsrc2];
end

////////////////////////////////////////////////////////////// bitwise xnor

`rxnor : begin
     if(`imm_mode)
       GPR[`rdst]  = GPR[`rsrc1] ~^ `isrc;
    else
       GPR[`rdst]   = GPR[`rsrc1] ~^ GPR[`rsrc2];
end

////////////////////////////////////////////////////////////// bitwise nand

`rnand : begin
     if(`imm_mode)
       GPR[`rdst]  = ~(GPR[`rsrc1] & `isrc);
    else
      GPR[`rdst]   = ~(GPR[`rsrc1] & GPR[`rsrc2]);
end

//////////////////////////////////////////////////////////////bitwise nor

`rnor : begin
     if(`imm_mode)
       GPR[`rdst]  = ~(GPR[`rsrc1] | `isrc);
    else
      GPR[`rdst]   = ~(GPR[`rsrc1] | GPR[`rsrc2]);
end

//////////////////////////////////////////////////////////////not

`rnot : begin
     if(`imm_mode)
       GPR[`rdst]  = ~(`isrc);
    else
       GPR[`rdst]   = ~(GPR[`rsrc1]);
end

//////////////////////////////////////////////////////////////

`storedin: begin
  data_mem[`isrc] = din;
end
```

```
////////////////////////////////////////////////////////////

`storereg: begin
   data_mem[`isrc] = GPR[`rsrc1];
end

////////////////////////////////////////////////////////////

`senddout: begin
   dout  = data_mem[`isrc];
end

////////////////////////////////////////////////////////////

`sendreg: begin
  GPR[`rdst] =  data_mem[`isrc];
end

////////////////////////////////////////////////////////////

`jump: begin
 jmp_flag = 1'b1;
end

`jcarry: begin
  if(carry == 1'b1)
     jmp_flag = 1'b1;
   else
     jmp_flag = 1'b0;
end

`jsign: begin
  if(sign == 1'b1)
     jmp_flag = 1'b1;
   else
     jmp_flag = 1'b0;
end

`jzero: begin
  if(zero == 1'b1)
     jmp_flag = 1'b1;
   else
     jmp_flag = 1'b0;
end

`joverflow: begin
  if(overflow == 1'b1)
     jmp_flag = 1'b1;
   else
     jmp_flag = 1'b0;
```

```
end

`jnocarry: begin
  if(carry == 1'b0)
     jmp_flag = 1'b1;
   else
     jmp_flag = 1'b0;
end

`jnosign: begin
  if(sign == 1'b0)
     jmp_flag = 1'b1;
   else
     jmp_flag = 1'b0;
end

`jnozero: begin
  if(zero == 1'b0)
     jmp_flag = 1'b1;
   else
     jmp_flag = 1'b0;
end

`jnooverflow: begin
  if(overflow == 1'b0)
     jmp_flag = 1'b1;
   else
     jmp_flag = 1'b0;
end

//////////////////////////////////////////////////////////
`halt : begin
stop = 1'b1;
end

endcase
end
endtask


//////////////////////logic for condition flag
task decode_condflag();
begin

/////////////////sign bit
if(`oper_type == `mul)
  sign = SGPR[15];
else
  sign = GPR[`rdst][15];
```

```
////////////////carry bit
if(`oper_type == `add)
  begin
    if(`imm_mode)
      begin
      temp_sum = GPR[`rsrc1] + `isrc;
      carry    = temp_sum[16];
      end
    else
      begin
      temp_sum = GPR[`rsrc1] + GPR[`rsrc2];
      carry    = temp_sum[16];
      end   end
  else
   begin
      carry  = 1'b0;
   end

//////////////////// zero bit

if(`oper_type == `mul)
  zero = ~((|SGPR[15:0]) | (|GPR[`rdst]));
else
  zero = ~(|GPR[`rdst]);

//////////////////////overflow bit

if(`oper_type == `add)
    begin
      if(`imm_mode)
        overflow = ( (~GPR[`rsrc1][15] & ~IR[15] & GPR[`rdst][15] ) | (GPR[`rsrc1][15] &
IR[15] & ~GPR[`rdst][15]) );
      else
        overflow = ( (~GPR[`rsrc1][15] & ~GPR[`rsrc2][15] & GPR[`rdst][15]) |
(GPR[`rsrc1][15] & GPR[`rsrc2][15] & ~GPR[`rdst][15]));
    end
  else if(`oper_type == `sub)
   begin
      if(`imm_mode)
        overflow = ( (~GPR[`rsrc1][15] & IR[15] & GPR[`rdst][15] ) | (GPR[`rsrc1][15] &
~IR[15] & ~GPR[`rdst][15]) );
      else
        overflow = ( (~GPR[`rsrc1][15] & GPR[`rsrc2][15] & GPR[`rdst][15]) |
(GPR[`rsrc1][15] & ~GPR[`rsrc2][15] & ~GPR[`rdst][15]));
   end
  else
    begin
    overflow = 1'b0;
    end
```

```
end
endtask

//////////reading program
initial begin
$readmemb("C:/Users/Reimarc/Downloads/refilling.mem",inst_mem);
end

//////////reading instructions one after another
reg [2:0] count = 0;
integer PC = 0;

////////////////////////////////// fsm states

parameter idle = 0, fetch_inst = 1, dec_exec_inst = 2, next_inst = 3, sense_halt = 4,
delay_next_inst = 5;
//////idle : check reset state
///// fetch_inst : load instruction from Program memory
///// dec_exec_inst : execute instruction + update condition flag
///// next_inst : next instruction to be fetched
reg [2:0] state = idle, next_state = idle;
////////////////////////////////// fsm states

//////////////////reset decoder

always@(posedge clk)
begin
 if(sys_rst)
   state <= idle;
 else
   state <= next_state;
end

//////////////////next state decoder + output decoder

always@(*)
begin
  case(state)
   idle: begin
     IR         = 32'h0;
     PC          = 0;
     next_state = fetch_inst;
   end

  fetch_inst: begin
    IR          =  inst_mem[PC];
    next_state  = dec_exec_inst;
  end

  dec_exec_inst: begin
```

```
   decode_inst();
   decode_condflag();
   next_state  = delay_next_inst;
  end


  delay_next_inst:begin
  if(count < 4)
      next_state  = delay_next_inst;
    else
      next_state  = next_inst;
  end

  next_inst: begin
     next_state = sense_halt;
     if(jmp_flag == 1'b1)
       PC = `isrc;
     else
       PC = PC + 1;
  end


 sense_halt: begin
    if(stop == 1'b0)
      next_state = fetch_inst;
    else if(sys_rst == 1'b1)
      next_state = idle;
    else
      next_state = sense_halt;
  end

  default : next_state = idle;

  endcase

end

//////////////////////////////////// count update

always@(posedge clk)
begin
case(state)

 idle : begin
    count <= 0;
 end

 fetch_inst: begin
   count <= 0;
 end
```

```
 dec_exec_inst : begin
   count <= 0;
 end

 delay_next_inst: begin
   count  <= count + 1;
 end

  next_inst : begin
    count <= 0;
 end

  sense_halt : begin
    count <= 0;
 end

 default : count <= 0;

endcase
end


endmodule
```

TESTBENCH

```
`timescale 1ns / 1ps

//////////fields of IR
`define oper_type     IR[31:27]
`define rdst          IR[26:22]
`define rsrc1         IR[21:17]
`define imm_mode    IR[16]
`define rsrc2         IR[15:11]
`define isrc          IR[15:0]


////////////////arithmetic operation
`define movsgpr     5'b00000
`define mov         5'b00001
`define add         5'b00010
`define sub         5'b00011
`define mul         5'b00100

////////////////logical operations : and or xor xnor nand nor not

`define ror         5'b00101
`define rand        5'b00110
```

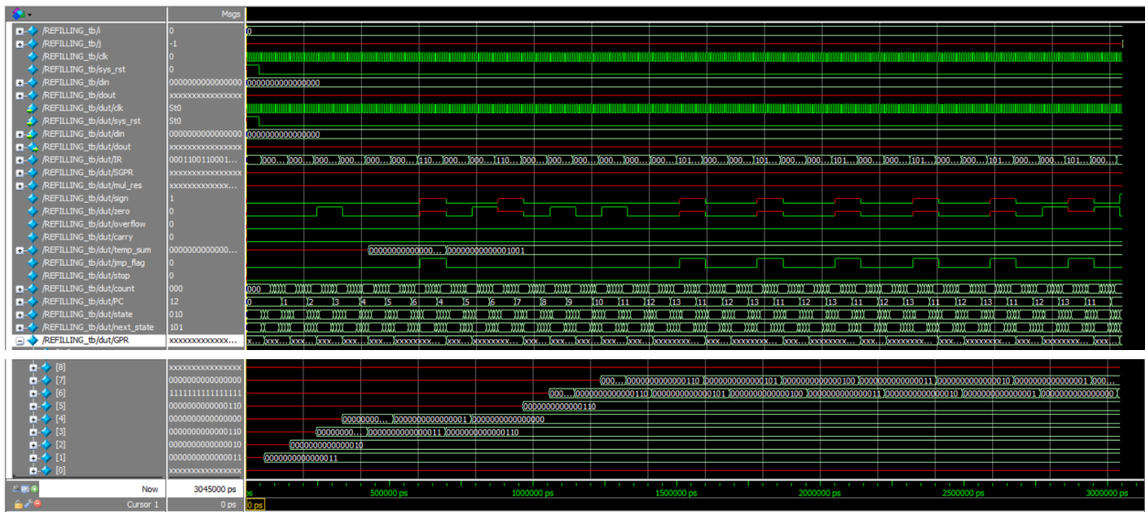
```
`define rxor          5'b00111
`define rxnor         5'b01000
`define rnand         5'b01001
`define rnor          5'b01010
`define rnot          5'b01011

////////////////////// load & store instructions

`define storereg       5'b01101   //////store content of register in data memory
`define storedin       5'b01110   ////// store content of din bus in data memory
`define senddout       5'b01111   /////send data from DM to dout bus
`define sendreg        5'b10001   ////// send data from DM to register

////////////////////////////// Jump and branch instructions
`define jump           5'b10010  ////jump to address
`define jcarry         5'b10011  ////jump if carry
`define jnocarry       5'b10100
`define jsign          5'b10101  ////jump if sign
`define jnosign        5'b10110
`define jzero          5'b10111  //// jump if zero
`define jnozero        5'b11000
`define joverflow      5'b11001 ////jump if overflow
`define jnooverflow    5'b11010

//////////////////////////halt
`define halt          5'b11011

module REFILLING_tb;

integer i = 0;
integer j;

reg clk = 0,sys_rst = 0;
reg [15:0] din = 0;
wire [15:0] dout;

REFILLING dut(clk, sys_rst, din, dout);

always #5 clk = ~clk;

initial begin
sys_rst = 1'b1;
repeat(5) @(posedge clk);
sys_rst = 1'b0;
#3000;
$display("--------------------------------------------------FIRST CUSTOMER");
$display("WATER REFILLING BOX");
$display("1 pesos =  %0d seconds", dut.GPR[2]);
$display("INSERTED: %0d pesos", dut.GPR[1]);
```

```
$display("TIME DURATION: From %0d seconds to %0d second", dut.GPR[5],
dut.GPR[7]);
$display("----------------------------------------------------------------");

for (j = dut.GPR[5]; j >= 0; j = j - 1)
begin
    $display("%d seconds", j);
end

$stop;
end

endmodule
```

TRANSCRIPT DISPLAY

```
# ------------------------------------------------------FIRST CUSTOMER
# WATER REFILLING BOX
# 1 pesos =  2 seconds
# INSERTED: 3 pesos
# TIME DURATION: From 6 seconds to 0 second
# ------------------------------------------------------------------
#            6 seconds
#            5 seconds
#            4 seconds
#            3 seconds
#            2 seconds
#            1 seconds
#            0 seconds
# Break in Module REFILLING_tb at C:/altera/13.0sp1/REFILLING_tb.v line 82
```

WAVEFORM DISPLAY



MEMORY FILE

```
00001_00001_00000_1_0000_0000_0000_0011
00001_00010_00000_1_0000_0000_0000_0010
00001_00011_00000_1_0000_0000_0000_0000
00001_00100_00000_1_0000_0000_0000_0010
00010_00011_00011_0_0000_1000_0000_0000
00011_00100_00100_1_0000_0000_0000_0001
11000_00000_00000_0_0000_0000_0000_0100
00001_00101_00011_0_0000_0000_0000_0000
00001_00110_00000_1_0000_0000_0000_0000
00001_00110_00101_0_0000_0000_0000_0000
00001_00111_00000_1_0000_0000_0000_0000
00001_00111_00110_0_0000_0000_0000_0000
00011_00110_00110_1_0000_0000_0000_0001
10110_00000_00000_0_0000_0000_0000_1011
11011_00000_00000_0_0000_0000_0000_0000
```

ASSEMBLY LANGUAGE CODE

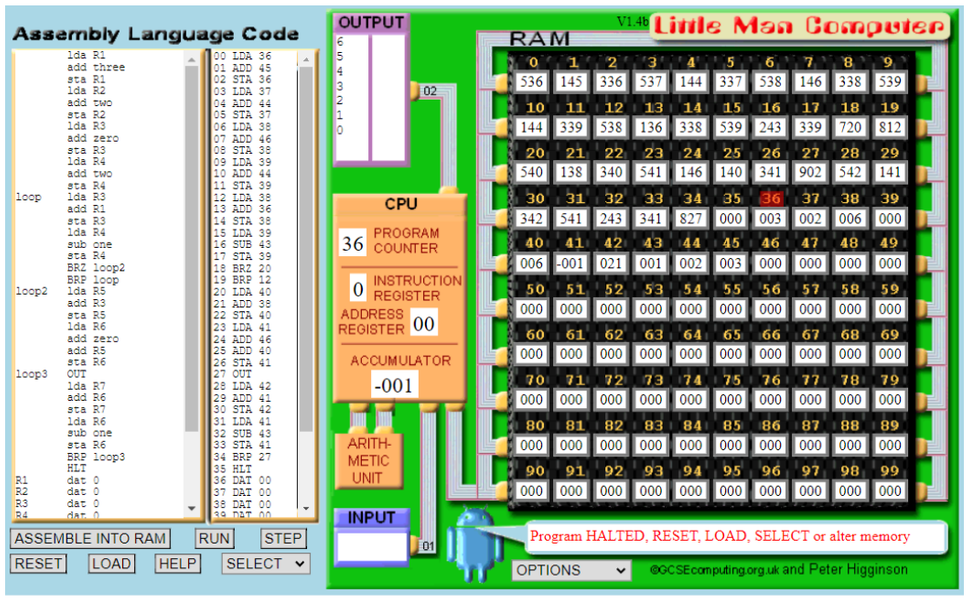
```
MOV R1, #3;
MOV R2, #2;
MOV R3, #0;
MOV R4, #2;
ADD R3, R3, R1;
SUB R4, R4, #1;
JNZ @4;
MOV R5, R3;
MOV R6, #0;
MOV R6, R5;
MOV R7, #0;
MOV R7, R6;
SUB R6, R6, #1;
JNS @11;
HALT
```

LITTLE MAN COMPUTER

```
        lda R1
        add three
        sta R1
        lda R2
        add two
        sta R2
        lda R3
        add zero
        sta R3
        lda R4
        add two
        sta R4
loop    lda R3
        add R1
        sta R3
        lda R4
```

```
        sub one
        sta R4
        BRZ loop2
        BRP loop
loop2   lda R5
        add R3
        sta R5
        lda R6
        add zero
        add R5
        sta R6
loop3   OUT
        lda R7
        add R6
        sta R7
        lda R6
        sub one
        sta R6
        BRP loop3
        HLT
R1      dat 0
R2      dat 0
R3      dat 0
R4      dat 0
R5      dat 0
R6      dat 0
R7      dat 0
one     dat 1
two     dat 2
three   dat 3
zero    dat 0
```

LMC DISPLAY OUTPUT



CONCLUSION

In this project, we utilized the provided Verilog CPU design modules to understand data operations within CPU architecture. To test our understanding we have a water refilling box as the sample. We then make an assembly language code to serve as the foundation on how we can make the problem functions. We also used this to further understand how data transfer occurs on the registers. We also use this as a basis to make our mem file. Subsequently, we constructed our own testbench to evaluate our model. And with these we are able to observe it through its codes, inputs like mem file, and through its outputs such as the waveforms and the transcript.

Through this project, we developed a strong understanding of how registers, such as the instruction register (IR) and general-purpose registers (GPRs), function within a CPU. The IR holds 32 binary bits and stores the current instruction, while the GPRs also hold 32 bits and store memory data. We also gained knowledge of how different operations like addition, subtraction, multiplication, move, and jump work inside the CPU. Particularly this project primarily focused on using addition and subtraction operations. This project enabled us to apply our gained knowledge in a practical application, such as the water refilling box. In conclusion, using Verilog codes helped us realize how data works within the CPU architecture.