MELZG554 HSCD LAB ASSIGNMENT 1

HSCD of SIMPLE ISA

Submitted by,

Anand S – 2021HT80003

Contents

Table of Figures

Table of source codes

Question

SystemC Assignment 1
Weightage: 10 marks **Deadline: 9th Nov, 2022**

On the System C assignment --> We can have the trivial instruction set as an example and the following to be implemented...
a> implement the ISA model in plain C and run the application as in (c) below.
b> implement the ISA in SystemC with full clock visibility and run the application as in (c) below.
The following to be noted

- It takes one clock cycle for a RAM or ROM model. i.e, if the address is given on the rising edge of clock '0', data will come out on clock '1' read cycle. For a write cycle, it takes one clock to actually complete the write.
- It takes one clock cycle for a Register Bank similar to RAM.
- It takes one clock cycle to add, subtract numbers
- Students can add JNZ (Jump on Not Zero) as a new instruction of their own to simplify the application sw.

c> Application sw is to do an addition of 10 numbers. say a0, a1, ... a9. Each number is 8bits wide.

Figure 2.6: A simple (trivial) instruction set.

Assembly instruct.	First byte		Second byte		Operation
MOV Rn, direct	0000	Rn	direct		Rn = M(direct)
MOV direct, Rn	0001	Rn	direct		M(direct) = Rn
MOV @Rn, Rm	0010		Rn	Rm	M(Rn) = Rm
MOV Rn, #immed.	0011	Rn	immediate		Rn = immediate
ADD Rn, Rm	0100		Rn	Rm	Rn = Rn + Rm
SUB Rn, Rm	0101		Rn	Rm	Rn = Rn - Rm
JZ Rn, relative	1000	Rn	relative		PC = PC+ relative (only if Rn is 0)

PART C: Application code to add ten numbers

```
0: MOV R0, 0xa   //# R0 is Loop iterator
1: MOV R1, 0x0   //# Clear R1 - R1 will have the numbers to be added up
2: MOV R2, 0x1   //# R2 has the value 0x1 for Increment and decrement ops
3: MOV R3, 0x0   //# R3 is the accumulator for the sum operations
4: ADD R3, R1    //# R3 = R3 + R1 - ac
5: ADD R1, R2    //# R1 = R1 + 1 - INCR R1 - get the next number
6: SUB R0, R2    //# R0 = R0 - 1 - DECR R0 - Next iteration of SUM loop
7: JNZ R0, -3    //# PC = PC + (-3) so that PC begins again at '4' and loops
if R0 != 0
```

Figure 1 Application code to add 10 numbers

The SW adds 10 numbers 0, 1, 2, 3, … 9 to get a sum of 45 (0x2d) = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9

PART A: Implement the ISA model in plain C and run the application code in PART C

```c
// ------------------------------------------------------------------
// HSCD assignent-1: Q1
// Author: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------
// vim: set sw=4: ts=4: set num: set expandtab:
// Online C compiler to run C program online
// C model for simple ISA
// Resources required to simulated:
// An instruction memory - say 128 half-words deep
#include <stdio.h>
#include <stdint.h>

// Typedefs ---------------------------------------------------------
typedef uint8_t  pc_t; // PC has address space of 256B
typedef uint16_t instruction_mem_t [256]; // PC has address space of 256B
typedef uint8_t  reg_file_t [16]; // R0-R15 registers, each 1B
typedef uint8_t  dmem_t [256]; // Data Mem is also addressed using 8bits
typedef uint8_t  opcode_t;
typedef uint8_t  op1_t;
typedef int16_t  op2_t; // This has to be signed for relative jumps

// Hardware resources (Global variables) ----------------------------
instruction_mem_t INSTR = {
    /* 0: MOV R0, 0xa */ 0x300a  // Setup
,   /* 1: MOV R1, 0x0 */ 0x3100  // Setup
,   /* 2: MOV R2, 0x1 */ 0x3201  // Setup
,   /* 3: MOV R3, 0x0 */ 0x3300  // Setup
,   /* 4: ADD R3, R1  */ 0x4031  // R3 = R3 + R1        -- R3 accumulates the sum
,   /* 5: ADD R1, R2  */ 0x4012  // R1 = R1 + 1; INC R1 -- New num created by incr of R1
,   /* 6: SUB R0, R2  */ 0x5002  // R0 = R0 - 1; DEC R1
,   /* 7: JNZ R0, -3  */ 0x90fd  // PC = PC + (-3) if R0 != 0
//    0x0010 // MOV Rn, direct; |0x0|Rn||dir|ect|; R0 = M(0x10)
//,   0x1100 // MOV direct, Rn; |0x1|Rn||dir|ect|; M(0x0) = R1
//,   0x2012 // MOV @Rn,Rm    ; |0x2|  ||Rn |Rm |; M(R1)  = R2
//,   0x33ff // MOV Rn, #immed; |0x3|Rn||Imm|edt|; R3 = 0xff
//,   0x4043 // ADD Rn, Rm    ; |0x4|  ||Rn |Rm |; R4 = R4 + R3
//,   0x5043 // SUB Rn, Rm    ; |0x5|  ||Rn |Rm |; R4 = R4 - R3
//,   0x84ff // JZ  Rn, reltiv; |0x8|R4||rel|tiv|; PC = PC + 4 (Only if R4 is 0)
//,   0x950f // JNZ Rn, reltiv; |0x8|R4||rel|tiv|; PC = PC + 4 (Only if R4 is != 0)
};
pc_t       PC;
reg_file_t R;
dmem_t     M;

// Hardware functions -----------------------------------------------
pc_t pc_update (pc_t PC, pc_t pc_incr, int st) {
    int orig_PC = PC;
    if (st == 0) {
        PC = PC + pc_incr;
    }
    printf("--> PC = PC + 0x%x = 0x%x + 0x%x = 0x%x\n", pc_incr, orig_PC, pc_incr, PC);
    return PC;
}
```

```
// Instruction parse operations - direct means 2nd byte is fully op2
opcode_t instr_opcode (int instr_addr) {
    return INSTR[instr_addr] >> 12;
}

op1_t instr_op1 (int instr_addr, int direct) {
    if (direct) {
        return (INSTR[instr_addr] >> 8) & 0x0f;
    } else {
        return ((INSTR[instr_addr] & 0xf0) >> 4);
    }
}

op2_t instr_op2 (int instr_addr, int direct) {
    if (direct) {
        return (INSTR[instr_addr] & 0xff);
    } else {
        return (INSTR[instr_addr] & 0x0f);
    }
}

int instr_decode (int instr_addr) {
    int st = 0;
    int orig_R_lhs;
    op1_t op1;
    op2_t op2;
    pc_t  pc_incr = 1; // Next instruction - points to next half-word
    opcode_t opcode = instr_opcode(instr_addr);
    printf ("--> PC: %d \n", PC);
    switch (opcode) {
        case 0x0:
            // MOV Rn, direct; |0x0|Rn||dir|ect|; Rn = M(direct)
            op1 = instr_op1(instr_addr, 1);
            op2 = instr_op2(instr_addr, 1);
            R[op1] = M[op2];
            printf("---> MOV R%d, 0x%x\n", op1, op2);
            printf("-------> R%d = M(0x%x) -> 0x%x = 0x%x\n", op1, op2, R[op1], M[op2]);
            break;
        case 0x1:
            // MOV direct, Rn; |0x1|Rn||dir|ect|; M(direct) = Rn
            op1 = instr_op1(instr_addr, 1);
            op2 = instr_op2(instr_addr, 1);
            M[op2] = R[op1];
            printf("---> MOV 0x%x, R%d\n", op2, op1);
            printf("-------> M(0x%x) = R%d -> 0x%x = 0x%x\n", op2, op1, M[op2], R[op1]);
            break;
        case 0x2:
            // MOV @Rn,Rm    ; |0x2|  ||Rn |Rm |; M(Rn) = Rm
            op1 = instr_op1(instr_addr,0);
            op2 = instr_op2(instr_addr,0);
            M[R[op1]] = R[op2];
            printf("---> MOV @R%d, R%d\n", op1, op2);
            printf("-------> M($R%d) = M(0x%x) = R%d -> 0x%x = 0x%x\n", op1, R[op1], op2,
M[R[op1]], R[op2]);
            break;
        case 0x3:
            // MOV Rn, #immed; |0x3|Rn||Imm|edt|; Rn = #immed
            op1 = instr_op1(instr_addr,1);
            op2 = instr_op2(instr_addr,1);
            R[op1] = op2;
            printf("---> MOV R%d, 0x%x\n", op1, op2);
            printf("-------> R%d = 0x%x -> 0x%x = 0x%x\n", op1, op2, R[op1], op2);
            break;
```

```
        case 0x4:
            // ADD Rn, Rm    ; |0x4|  ||Rn |Rm |; Rn = Rn + Rm
            op1 = instr_op1(instr_addr,0);
            op2 = instr_op2(instr_addr,0);
            orig_R_lhs = R[op1];
            R[op1] = R[op1] + R[op2];
            printf("---> ADD R%d, R%d\n", op1, op2);
            printf("-------> R%d = R%d + R%d -> 0x%x = 0x%x + 0x%x\n", op1, op1, op2, R[op1],
orig_R_lhs, R[op2]);
            break;
        case 0x5:
            // SUB Rn, Rm    ; |0x5|  ||Rn |Rm |; Rn = Rn - Rm
            op1 = instr_op1(instr_addr,0);
            op2 = instr_op2(instr_addr,0);
            orig_R_lhs = R[op1];
            R[op1] = R[op1] - R[op2];
            printf("---> SUB R%d, R%d\n", op1, op2);
            printf("-------> R%d = R%d - R%d -> 0x%x = 0x%x - 0x%x\n", op1, op1, op2, R[op1],
orig_R_lhs, R[op2]);
            break;
        case 0x6:
            break;
        case 0x8:
            // JZ  Rn, reltiv; |0x8|R4||rel|tiv|; PC = PC + relative
            op1 = instr_op1(instr_addr, 1);
            op2 = instr_op2(instr_addr, 1);
            if (R[op1] == 0) {
                pc_incr = op2;
                printf("------> JUMP succeeded\n");
            } else {
                printf("------> JUMP failed\n");
            }
            printf("---> JZ R%d, 0x%x\n", op1, op2);
            break;
        case 0x9:
            // JZ  Rn, reltiv; |0x8|R4||rel|tiv|; PC = PC + relative
            op1 = instr_op1(instr_addr, 1);
            op2 = instr_op2(instr_addr, 1);
            printf("---> JNZ R%d, 0x%x\n", op1, op2);
            if (R[op1] != 0) {
                pc_incr = op2;
                printf("------> JUMP succeeded\n");
            } else {
                printf("------> JUMP failed\n");
            }
            break;
        default:
            st = 1;
            printf("ERROR: Unsupported opcode %x\n", opcode);
            break;
    }
    PC = pc_update(PC, pc_incr, st);
    return st;
}

int main() {
    // Write C code here
    int er;
    printf("Hello world\n");
    printf("2nd element in instr mem is 0x%x \n", INSTR[1]);
    while (PC < 8) {
        er += instr_decode(PC);
    }
    return er;
}
```

Figure 2 C code for the simple ISA

```
~/uvmHelloWorld/design/simple_isa/c_model master ?2 ..... 11:14:36 AM
> make c_model_isa
cc   c_model_isa.o   -o c_model_isa
~/uvmHelloWorld/design/simple_isa/c_model master ?2 ..... 11:14:41 AM
> ./c_model_isa
Hello world
2nd element in instr mem is 0x3100
--> PC: 0
---> MOV R0, 0xa
-------> R0 = 0xa -> 0xa = 0xa
--> PC = PC + 0x1 = 0x0 + 0x1 = 0x1
--> PC: 1
---> MOV R1, 0x0
-------> R1 = 0x0 -> 0x0 = 0x0
--> PC = PC + 0x1 = 0x1 + 0x1 = 0x2
--> PC: 2
---> MOV R2, 0x1
-------> R2 = 0x1 -> 0x1 = 0x1
--> PC = PC + 0x1 = 0x2 + 0x1 = 0x3
--> PC: 3
---> MOV R3, 0x0
-------> R3 = 0x0 -> 0x0 = 0x0
--> PC = PC + 0x1 = 0x3 + 0x1 = 0x4
--> PC: 4
---> ADD R3, R1
-------> R3 = R3 + R1 -> 0x0 = 0x0 + 0x0
--> PC = PC + 0x1 = 0x4 + 0x1 = 0x5
--> PC: 5
---> ADD R1, R2
-------> R1 = R1 + R2 -> 0x1 = 0x0 + 0x1
--> PC = PC + 0x1 = 0x5 + 0x1 = 0x6
--> PC: 6
---> SUB R0, R2
-------> R0 = R0 - R2 -> 0x9 = 0xa - 0x1
--> PC = PC + 0x1 = 0x6 + 0x1 = 0x7
--> PC: 7
---> JNZ R0, 0xfd
------> JUMP succeeded
--> PC = PC + 0xfd = 0x7 + 0xfd = 0x4
--> PC: 4
---> ADD R3, R1
-------> R3 = R3 + R1 -> 0x1 = 0x0 + 0x1
--> PC = PC + 0x1 = 0x4 + 0x1 = 0x5
--> PC: 5
---> ADD R1, R2
-------> R1 = R1 + R2 -> 0x2 = 0x1 + 0x1
--> PC = PC + 0x1 = 0x5 + 0x1 = 0x6
--> PC: 6
---> SUB R0, R2
-------> R0 = R0 - R2 -> 0x8 = 0x9 - 0x1
```

```
---> SUB R0, R2
-------> R0 = R0 - R2 -> 0x8 = 0x9 - 0x1
--> PC = PC + 0x1 = 0x6 + 0x1 = 0x7
--> PC: 7
---> JNZ R0, 0xfd
------> JUMP succeeded
--> PC = PC + 0xfd = 0x7 + 0xfd = 0x4
--> PC: 4
---> ADD R3, R1
-------> R3 = R3 + R1 -> 0x3 = 0x1 + 0x2
--> PC = PC + 0x1 = 0x4 + 0x1 = 0x5
--> PC: 5
---> ADD R1, R2
-------> R1 = R1 + R2 -> 0x3 = 0x2 + 0x1
--> PC = PC + 0x1 = 0x5 + 0x1 = 0x6
--> PC: 6
---> SUB R0, R2
-------> R0 = R0 - R2 -> 0x7 = 0x8 - 0x1
--> PC = PC + 0x1 = 0x6 + 0x1 = 0x7
--> PC: 7
---> JNZ R0, 0xfd
------> JUMP succeeded
--> PC = PC + 0xfd = 0x7 + 0xfd = 0x4
--> PC: 4
---> ADD R3, R1
-------> R3 = R3 + R1 -> 0x6 = 0x3 + 0x3
--> PC = PC + 0x1 = 0x4 + 0x1 = 0x5
--> PC: 5
---> ADD R1, R2
-------> R1 = R1 + R2 -> 0x4 = 0x3 + 0x1
--> PC = PC + 0x1 = 0x5 + 0x1 = 0x6
--> PC: 6
---> SUB R0, R2
-------> R0 = R0 - R2 -> 0x6 = 0x7 - 0x1
--> PC = PC + 0x1 = 0x6 + 0x1 = 0x7
--> PC: 7
---> JNZ R0, 0xfd
------> JUMP succeeded
--> PC = PC + 0xfd = 0x7 + 0xfd = 0x4
--> PC: 4
---> ADD R3, R1
-------> R3 = R3 + R1 -> 0xa = 0x6 + 0x4
--> PC = PC + 0x1 = 0x4 + 0x1 = 0x5
--> PC: 5
---> ADD R1, R2
-------> R1 = R1 + R2 -> 0x5 = 0x4 + 0x1
--> PC = PC + 0x1 = 0x5 + 0x1 = 0x6
--> PC: 6
---> SUB R0, R2
-------> R0 = R0 - R2 -> 0x5 = 0x6 - 0x1
```

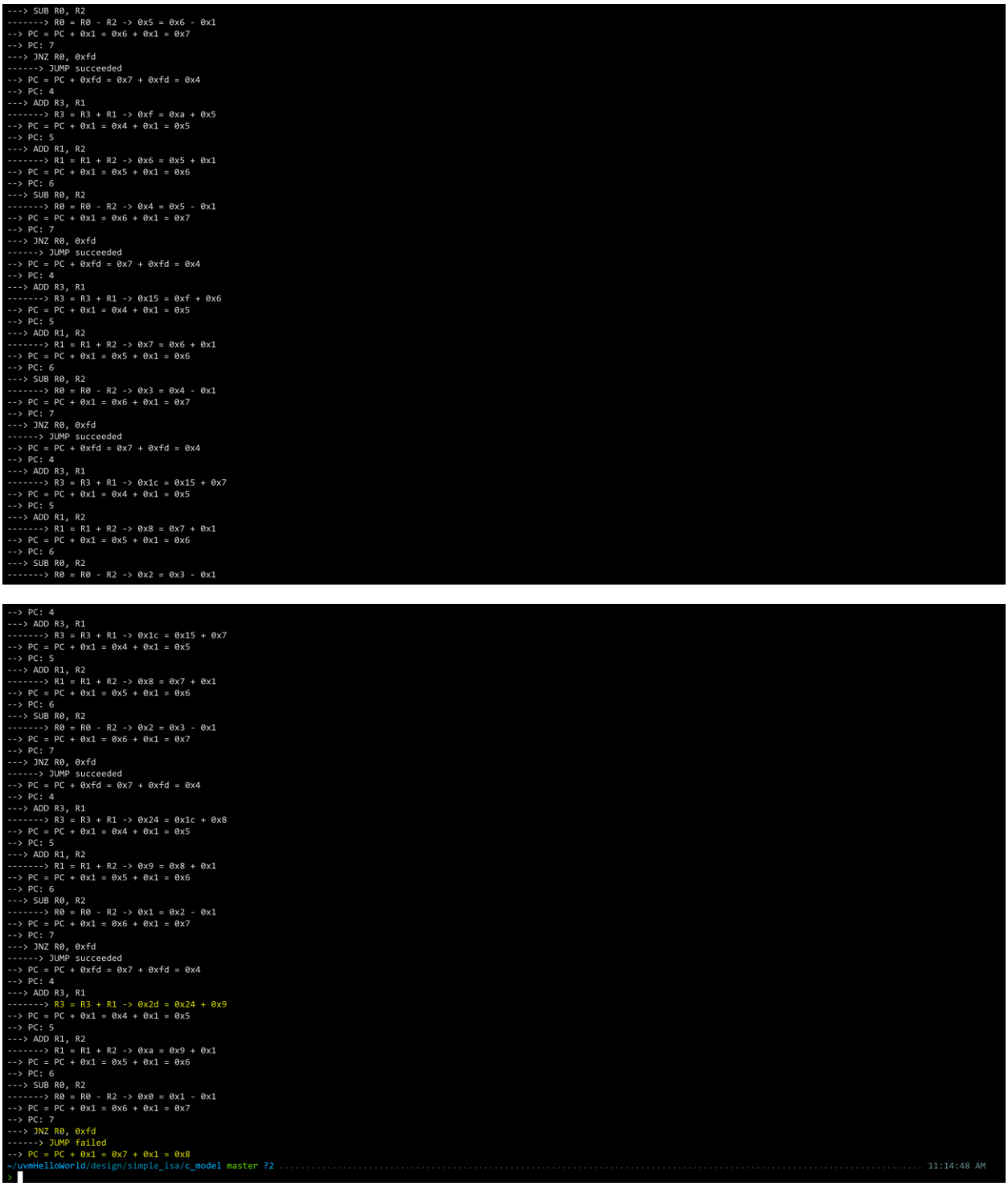


Figure 3 Execution log of the C model output

Thus the C code was executed for the application SW by coding the assembly into the instruction memory array. The execution of the code showed the final value of R3 = 8'd45 = 8'h2d, which is the expected output. Thus the execution was verified.

PART B: Implement the ISA as timing accurate HW model

Architecture of the simple_isa CPU (simple_cpu)

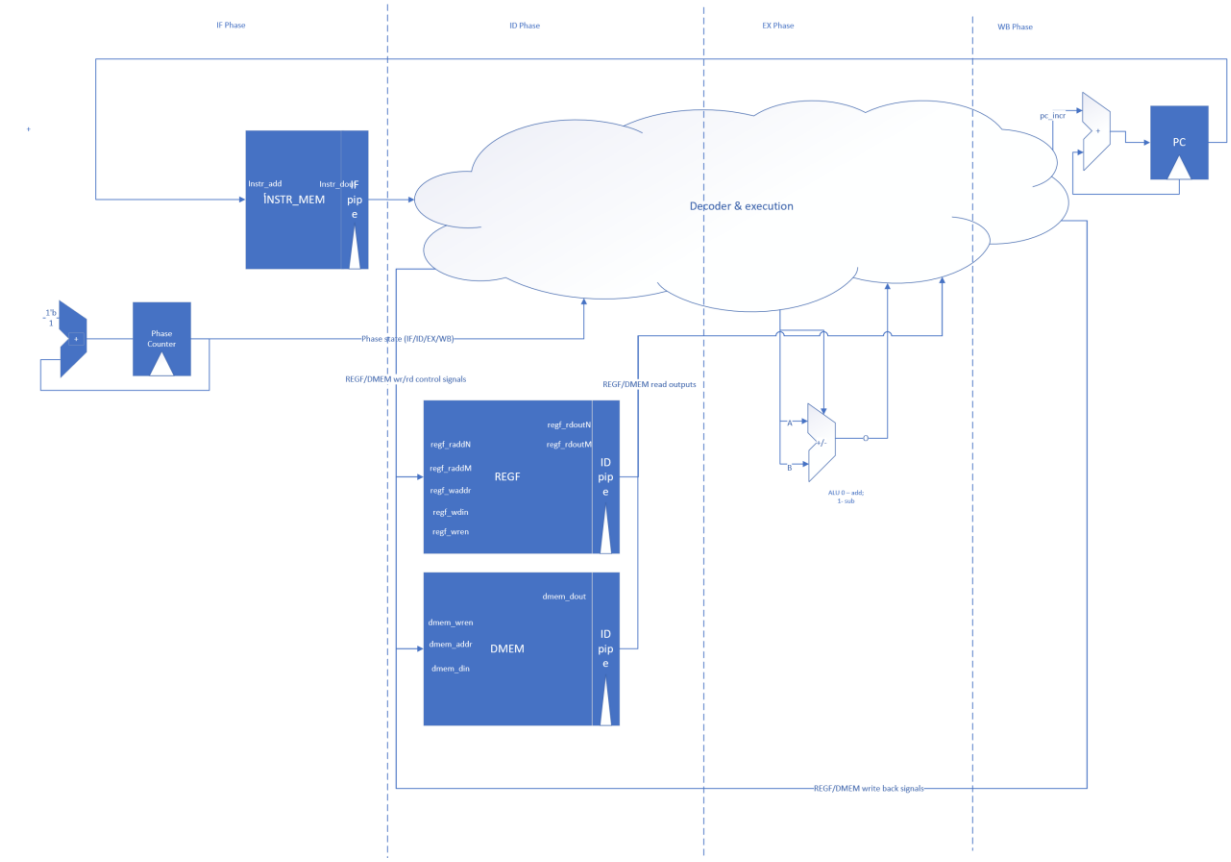


Figure 4 High level arch of the simple_isa CPU

The CPU is a 4 phase CPU with the phases IF (Instr Fetch), ID (Instr Decode), EX (Instr Execute), WB (Result Write Back). A dedicated Phase counter updates it's count every clock cycle, once out of reset.

The decode_ex combinational logic decides how to drive the H/W resources (REGF_MEM, DMEM, ALU)

During each phase for each of the implemented opcodes.

Implementing systemVerilog design for accurate timing

For trial and ease of implementation later-on with systemC, a timing accurate rough model was created for the SIMPLE-CPU design using systemVerilog. The detailed design is available in SystemVerilog code for the HDL design section.

simple_alu.sv simple_instr_mem.sv run.do simple_instr_mem.sv

```
// ------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------
// Specification : ROM with a sequential read port - 1 cycled delay

module simple_instr_mem (
    input   clk
,   input   resetn
,   input   logic   [7:0]  instr_addr // Instr mem is 512 bytes deep (256 Half-words)
,   output  logic   [15:0] INSTR      // Each instr is 2 bytes long
);

    // The instr mem
    logic [15:0] INSTR_MEM [256];

    always_comb begin
        INSTR_MEM[0] = 'h300a; /* 0: MOV R0, 0xa; Setup                                          */
        INSTR_MEM[1] = 'h3100; /* 1: MOV R1, 0x0; Setup                                          */
        INSTR_MEM[2] = 'h3201; /* 2: MOV R2, 0x1; Setup                                          */
        INSTR_MEM[3] = 'h3300; /* 3: MOV R3, 0x0; Setup                                          */
        INSTR_MEM[4] = 'h4031; /* 4: ADD R3, R1 ; R3 = R3 + R1        -- R3 accumulates the sum  */
        INSTR_MEM[5] = 'h4012; /* 5: ADD R1, R2 ; R1 = R1 + 1; INC R1 -- New num created by incr of R1*/
        INSTR_MEM[6] = 'h5002; /* 6: SUB R0, R2 ; R0 = R0 - 1; DEC R1                            */
        INSTR_MEM[7] = 'h90fd; /* 7: JNZ R0, -3 ; PC = PC + (-3) if R0 != 0                      */
    end

    // Sequential read
    always_ff @(posedge clk)
        INSTR <= INSTR_MEM[instr_addr];

endmodule: simple_instr_mem
```

Figure 5 Snapshot of simple_instr_mem module with the application SW loaded

The PC watchdog code ensures that the simulation completes when PC reaches value ‘8’. This is beyond the application code space, so this is a safe way to test the code.

simple_alu.sv simple_instr_mem.sv run.do simple_instr_mem.sv simple_pc.sv

```
 1 // ------------------------------------------------------------------
 2 // HSCD Assignment - 1
 3 // AUTHOR: Anand S
 4 // BITS ID: 2021HT80003
 5 // ------------------------------------------------------------------
 6
 7 module simple_pc (
 8     input   clk
 9 ,   input   resetn
10 ,   input   logic [1:0] phase
11 ,   input   logic signed [7:0] pc_incr
12
13 ,   output  logic [7:0] pc // PC can address 256 bytes
14
15 );
16
17     always_ff @(posedge clk, negedge resetn) begin: pc_incr_logic_seq
18         if (!resetn) begin
19             pc <= '0;
20         end else begin
21             if (phase == 2'b11)
22                 pc <= signed'(pc + pc_incr);
23         end
24     end: pc_incr_logic_seq
25     always_comb begin: pc_wdog
26         if (pc >= 8 && resetn === 1) begin
27             $info("PC value = %0d. Exiting.", pc);
28             $finish;
29         end
30     end: pc_wdog
31
32 endmodule: simple_pc
```

Figure 6 PC incrementing logic with an additional PC watchdog code added

Below is the obtained waveform. As shown at the bottom of the waves REGF[3] accumulates to a value of 0x2D, which is the expected result for the SW as obtained from the C model.

Simulation result

The simulation was done by loading the SW code with the simple ISA as shown below. This will act as a reference going forward for the systemC design.

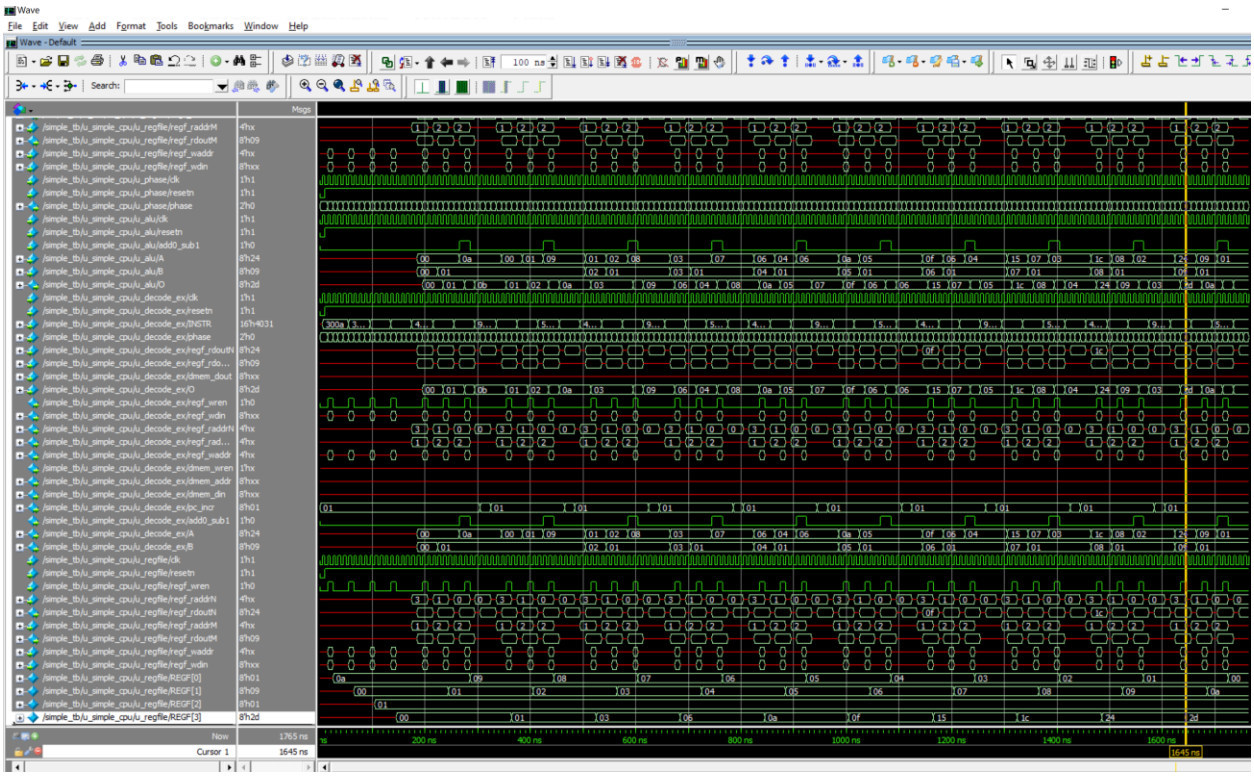


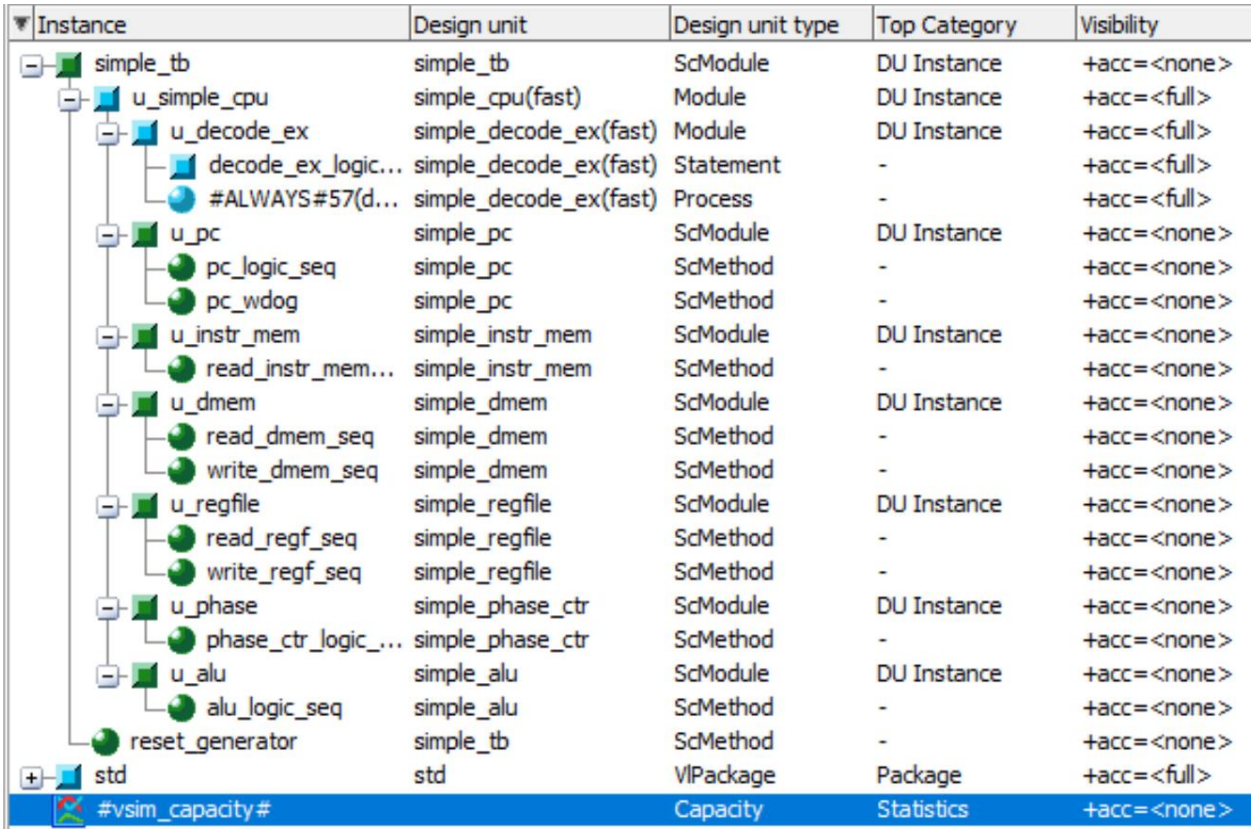
Figure 7 Final result obtained after running sim in ModelSim.

Note the highlight in the above figure.

It is clear from the waveform how the R0 register is updated to decrement in each loop, while the R1 register increments in each of the loop of the application SW.

SystemC code for simple_cpu hardware

Implementing the design as systemC code deriving from the systemVerilog model. Below is a tree graph of the hierarchical structure of the design.



Instance	Design unit	Design unit type	Top Category	Visibility
simple_tb	simple_tb	ScModule	DU Instance	+acc=<none>
u_simple_cpu	simple_cpu(fast)	Module	DU Instance	+acc=<full>
u_decode_ex	simple_decode_ex(fast)	Module	DU Instance	+acc=<full>
decode_ex_logic...	simple_decode_ex(fast)	Statement	-	+acc=<full>
#ALWAYS#57(d...	simple_decode_ex(fast)	Process	-	+acc=<full>
u_pc	simple_pc	ScModule	DU Instance	+acc=<none>
pc_logic_seq	simple_pc	ScMethod	-	+acc=<none>
pc_wdog	simple_pc	ScMethod	-	+acc=<none>
u_instr_mem	simple_instr_mem	ScModule	DU Instance	+acc=<none>
read_instr_mem...	simple_instr_mem	ScMethod	-	+acc=<none>
u_dmem	simple_dmem	ScModule	DU Instance	+acc=<none>
read_dmem_seq	simple_dmem	ScMethod	-	+acc=<none>
write_dmem_seq	simple_dmem	ScMethod	-	+acc=<none>
u_regfile	simple_regfile	ScModule	DU Instance	+acc=<none>
read_regf_seq	simple_regfile	ScMethod	-	+acc=<none>
write_regf_seq	simple_regfile	ScMethod	-	+acc=<none>
u_phase	simple_phase_ctr	ScModule	DU Instance	+acc=<none>
phase_ctr_logic_...	simple_phase_ctr	ScMethod	-	+acc=<none>
u_alu	simple_alu	ScModule	DU Instance	+acc=<none>
alu_logic_seq	simple_alu	ScMethod	-	+acc=<none>
reset_generator	simple_tb	ScMethod	-	+acc=<none>
std	std	VlPackage	Package	+acc=<full>
#vsim_capacity#		Capacity	Statistics	+acc=<none>

Figure 8 Tree graph of the hierarchies within the design

Figure 9 Diagrammatic representation of the hardware blocks in simple_cpu

Here, apart from the decode_ex and the cpu top-level, all the other modules, including the testbench are coded in systemC.

Below are the various source codes used for the systemC model of the design

```
// ------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------

#include "systemc.h"

SC_MODULE(simple_alu) {
    public:
    sc_in<bool>             clk;
    sc_in<bool>             resetn;
    sc_in<bool>             add0_sub1;
    sc_in<sc_uint<8> >      A;
    sc_in<sc_uint<8> >      B;
    sc_inout<sc_uint<8> >   O;

    // local vars

    void alu_logic_seq();

    // Constructor
    SC_CTOR (simple_alu)
        : clk("clk")
        , resetn("resetn")
        , A("A")
        , B("B")
        , O("O")
    {
        cout << "Executing new of simple_alu" << endl;
        SC_METHOD(alu_logic_seq)
        sensitive << clk.pos();
    }

    // Destructor
    ~simple_alu () {}
};

inline void simple_alu::alu_logic_seq() {
    if (add0_sub1.read() == 1) {
        O.write(A.read() - B.read());
    } else {
        O.write(A.read() + B.read());
    }
    cout << "@" << sc_time_stamp() <<":: ALU updated :: O = "
        << O.read() << " = A(" << A.read() << ") + B(" << B.read()
        << ")" << endl;
}

SC_MODULE_EXPORT(simple_alu);
```

Code 1 systemC code for simple_alu

```
// ------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------
// Specification : RAM with a sequential read port - 1 cycled delay
// Either reads or writes possible at any given time.


#include "systemc.h"

SC_MODULE(simple_dmem) {
    public:
    sc_in<bool>             clk;
    sc_in<bool>             resetn;
    sc_in<bool>             dmem_wren;
    sc_in<sc_uint<8> >      dmem_addr;  // dmem is 256 deep
    sc_inout<sc_uint<8> >   dmem_din;   // dmem is 2 Bytes wide
    sc_inout<sc_uint<8> >   dmem_dout;  // dmem is 2 Bytes wide

    // local vars
    sc_uint<8>   dmem [256];


    void read_dmem_seq();
    void write_dmem_seq();

    // Constructor
    SC_CTOR (simple_dmem)
        : clk("clk")
        , resetn("resetn")
        , dmem_wren("dmem_wren")
        , dmem_addr("dmem_addr")
        , dmem_din("dmem_din")
        , dmem_dout("dmem_dout")
    {
        cout << "Executing new of simple_dmem" << endl;

        SC_METHOD(read_dmem_seq)
        sensitive << clk.pos();
        SC_METHOD(write_dmem_seq)
        sensitive << clk.pos();
    }

    // Destructor
    ~simple_dmem () {}
};

inline void simple_dmem::read_dmem_seq() {
    dmem_dout.write(dmem[dmem_addr.read()]);
}

inline void simple_dmem::write_dmem_seq() {
    if (dmem_wren.read() == 1) {
        dmem[dmem_addr.read()] = dmem_din.read();
    }
}

SC_MODULE_EXPORT(simple_dmem);
```

Code 2 systemC code for the data memory

```
// ----------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ----------------------------------------------------------------------
// Specification : ROM with a sequential read port - 1 cycled delay

#include "systemc.h"

SC_MODULE(simple_instr_mem) {
    public:
    sc_in<bool>             clk;
    sc_in<bool>             resetn;
    sc_in<sc_uint<8> >      instr_addr; // INSTR mem is 256 deep
    sc_inout<sc_uint<16> >  INSTR;      // INSTR mem is 2 Bytes wide

    // local vars
    typedef sc_uint <16> instruction_mem_t [256];

    instruction_mem_t INSTR_MEM;

    void read_instr_mem_seq();

    // Constructor
    SC_CTOR (simple_instr_mem)
        : clk("clk")
        , resetn("resetn")
        , instr_addr("instr_addr")
        , INSTR("INSTR")
    {
        cout << "Executing new of simple_instr_mem" << endl;
        INSTR_MEM[0] = /* 0: MOV R0, 0xa */ 0x300a;// Setup
        INSTR_MEM[1] = /* 1: MOV R1, 0x0 */ 0x3100;// Setup
        INSTR_MEM[2] = /* 2: MOV R2, 0x1 */ 0x3201;// Setup
        INSTR_MEM[3] = /* 3: MOV R3, 0x0 */ 0x3300;// Setup
        INSTR_MEM[4] = /* 4: ADD R3, R1  */ 0x4031;// R3 = R3 + R1        -- R3 accumulates the sum
        INSTR_MEM[5] = /* 5: ADD R1, R2  */ 0x4012;// R1 = R1 + 1; INC R1 -- New num created by incr of R1
        INSTR_MEM[6] = /* 6: SUB R0, R2  */ 0x5002;// R0 = R0 - 1; DEC R1
        INSTR_MEM[7] = /* 7: JNZ R0, -3  */ 0x90fd;// PC = PC + (-3) if R0 != 0

        SC_METHOD(read_instr_mem_seq)
        sensitive << clk.pos();
    }

    // Destructor
    ~simple_instr_mem () {}
};

inline void simple_instr_mem::read_instr_mem_seq() {
    INSTR.write(INSTR_MEM[instr_addr.read()]);
}


SC_MODULE_EXPORT(simple_instr_mem);
```

Code 3 systemC code for instruction memory

```
// ------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------

#include "systemc.h"

SC_MODULE(simple_pc) {
    public:
    sc_in<bool>             clk;
    sc_in<bool>             resetn;
    sc_in<sc_uint<2> >      phase;
    sc_in<sc_int<8> >       pc_incr;
    sc_inout<sc_uint<8> >   pc;

    // local vars

    void pc_logic_seq();
    void pc_wdog();

    // Constructor
    SC_CTOR (simple_pc)
        : clk("clk")
        , resetn("resetn")
        , phase("phase")
        , pc_incr("pc_incr")
        , pc("pc")
    {
        cout << "Executing new of simple_pc" << endl;

        SC_METHOD(pc_logic_seq)
        sensitive << resetn.neg();
        sensitive << clk.pos();

        SC_METHOD(pc_wdog)
        sensitive << pc;
    }

    // Destructor
    ~simple_pc () {}
};

inline void simple_pc::pc_logic_seq() {
    if (resetn.read() == 0) {
        pc.write(0);
    } else {
        if (phase.read() == 3) {
            pc.write(pc.read() + pc_incr.read());
        }
    }
}

inline void simple_pc::pc_wdog() {
    if (pc.read() >= 8 && (resetn.read() == 1)) {
        cout << "@" << sc_time_stamp() << ":: PC overflowing to 8. Exiting."
            << endl;
        sc_stop();
    }
}

SC_MODULE_EXPORT(simple_pc);
```

Code 4 systemC code for pc update logic

```
// ------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------
// Has 4 phases - IF, ID, EX, WB
// The FSM uses this ctr to sequence the signals

#include "systemc.h"

SC_MODULE(simple_phase_ctr) {
    public:
    sc_in<bool>         clk;
    sc_in<bool>         resetn;
    sc_inout<sc_uint<2> >  phase;

    // local vars
    sc_uint<2> phase_int;

    void phase_ctr_logic_seq () {
        if (resetn.read() == 0) {
            phase_int = 0;
            phase.write(phase_int);
        } else {
            phase_int = phase_int + 1;
            phase.write(phase_int);
            cout << "@" << sc_time_stamp() <<":: Incremented phase = "
                << phase.read() << endl;
        }
    } // endfunction phase_ctr_logic_seq

    // Constructor
    SC_CTOR (simple_phase_ctr)
        : clk   ("clk")
        , resetn("resetn")
        , phase ("phase")
    {
        cout << "Executing new of simple_phase_ctr" << endl;
        SC_METHOD (phase_ctr_logic_seq);
        sensitive << resetn.neg();
        sensitive << clk.pos();
    } // endconstructore
    // Destructor
    ~simple_phase_ctr() {}
}; // endmodule simple_phase_ctr

SC_MODULE_EXPORT(simple_phase_ctr);
```

Code 5 systemC code for the phase counter logic

```
// ----------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ----------------------------------------------------------------------
// Specification : ROM with a sequential read port - 1 cycled delay

#include "systemc.h"

SC_MODULE(simple_regfile) {
    public:
    sc_in<bool>             clk;
    sc_in<bool>             resetn;
    sc_in<bool>             regf_wren;
    // Rd port - N
    sc in<sc uint<4> >      regf raddrN;  // regf is 256 deep
    sc_inout<sc_int<8> >    regf_rdoutN;  // regf is 1 Byte wide

    // Rd port - M
    sc_in<sc_uint<4> >      regf_raddrM;  // regf is 256 deep
    sc_inout<sc_int<8> >    regf_rdoutM;  // regf is 1 Byte wide

    // Wr port - common
    sc_in<sc_uint<4> >      regf_waddr;
    sc_in<sc_uint<8> >      regf_wdin;

    // local vars
    //typedef sc_int<8> regf_t [256];
    //sc_signal<regf_t>   REGF;
    //sc_vector<sc_signal<sc_int<9> >> REGF{"REGF", 256};
   sc_int<8>   REGF [256];


    void read_regf_seq();
    void write_regf_seq();

    // Constructor
    SC_CTOR (simple_regfile)
        : clk("clk")
        , resetn("resetn")
        , regf_wren("regf_wren")
        , regf_raddrN("regf_raddrN")
        , regf_rdoutN("regf_rdoutN")
        , regf_raddrM("regf_raddrM")
        , regf_rdoutM("regf_rdoutM")
        , regf_waddr("regf_waddr")
        , regf_wdin("regf_wdin")
    {
        cout << "Executing new of simple_regfile" << endl;

        SC_METHOD(read_regf_seq)
        sensitive << clk.pos();
        SC_METHOD(write_regf_seq)
        sensitive << clk.pos();
    }

    // Destructor
    ~simple_regfile () {}
};

inline void simple_regfile::read_regf_seq() {
    regf_rdoutN.write(REGF[regf_raddrN.read()]);
    regf_rdoutM.write(REGF[regf_raddrM.read()]);
}

inline void simple_regfile::write_regf_seq() {
    if (regf_wren.read() == 1) {
        int i = int(regf_waddr.read());
        REGF[i] = regf_wdin.read();
        // REGF[regf_waddr.read()] = regf_wdin.read();
    }
}

SC_MODULE_EXPORT(simple_regfile);
```

Code 6 systemC code for the register file block

```
// ----------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ----------------------------------------------------------------------
// Specification : RAM with two sequential read port - 1 cycle delay
// And 1 sequential write port.
// Either reads or writes possible at any given time.

module simple_decode_ex (
    input   clk
,   input   resetn

// Ctrl inputs
// -- instr
,   input   logic   [15:0]INSTR
// -- phase
,   input   logic   [1:0] phase
// -- regf
,   input   logic   [7:0] regf_rdoutN
,   input   logic   [7:0] regf_rdoutM
// -- dmem
,   input   logic   [7:0] dmem_dout
// -- alu
,   input   logic   [7:0] O

// Ctrl outputs
// -- regf
,   output  logic         regf_wren
,   output  logic   [7:0] regf_wdin
,   output  logic   [3:0] regf_raddrN
,                         regf_raddrM
,                         regf_waddr
// -- dmem
,   output  logic         dmem_wren
,   output  logic  [7:0]  dmem_addr
,   output  logic  [7:0]  dmem_din
// -- pc
,   output  logic  [7:0]  pc_incr
// -- alu
,   output  logic         add0_sub1
,   output  logic  [7:0]  A
,                         B
);
    `define OPCODE 15:12
    `define OP1_DIRECT 11:8
    `define OP2_DIRECT 7:0
    `define OP1 7:4
    `define OP2 3:0
    `define IF 0
    `define ID 1
    `define EX 2
    `define WB 3

    // This is a pure combinational block - only drives the output as required
    // for a given phase based on extensive muxing on the phase and the opcode
    always_comb begin: decode_ex_logic_comb
        // defaults
        regf_wren   = 0;
        regf_wdin   = 'x;
        regf_raddrN = 'x;
        regf_raddrM = 'x;
        regf_waddr  = 'x;

        dmem_wren   = 'x;
        dmem_addr   = 'x;
        dmem_din    = 'x;

        pc_incr     = 1;
        add0_sub1   = 0;

        // conditional updates
        case (INSTR[`OPCODE])
            0: begin // MOV Rn, direct; |0x0|Rn||dir|ect|; Rn = M(direct)
                case (phase)
                    `ID: begin
                        // Read M(direct)
                        dmem_wren = 0;
                        dmem_addr = INSTR[`OP2_DIRECT];
                    end
                    `EX: begin
                        // Read M(direct)
                        dmem_wren = 0;
                        dmem_addr = INSTR[`OP2_DIRECT];
                    end
                    `WB: begin
                        // Read M(direct)
                        dmem_wren = 0;
                        dmem_addr = INSTR[`OP2_DIRECT];
                        // Write to Rn
                        regf_wren = 1;
                        regf_wdin = dmem_dout;
                        regf_waddr= INSTR[`OP1_DIRECT];
                    end
                endcase
            end
            1: begin // MOV direct, Rn; |0x1|Rn||dir|ect|; M(direct) = Rn
                case (phase)
                    `ID: begin
                        // Read Rn
                        regf_wren = 0;
                        regf_raddrN = INSTR[`OP1_DIRECT];
                    end
                    `EX: begin
                        // Read Rn
                        regf_wren = 0;
                        regf_raddrN = INSTR[`OP1_DIRECT];
                        // write back early
                        dmem_wren = 1;
                        dmem_addr = INSTR[`OP2_DIRECT];
                        dmem_din  = regf_rdoutN;
                    end
                    `WB: begin
                        // Read Rn
```

```
            regf_wren = 0;
            regf_raddrN = INSTR[`OP1_DIRECT];
            // write back early
            dmem wren = 1;
            dmem addr = INSTR[`OP2 DIRECT];
            dmem din  = regf rdoutN;
        end
    endcase
end
2: begin // MOV @Rn,Rm    ; |0x2|  ||Rn |Rm |; M(Rn) = Rm
    case (phase)
        `ID: begin
            // Read Rn
            regf_wren = 0;
            regf raddrN = INSTR[`OP1];
            regf raddrM = INSTR[`OP2];
        end
        `EX: begin
            // Read Rn
            regf wren = 1;
            regf raddrN = INSTR[`OP1];
            // write back early
            dmem_wren  = 1;
            dmem_addr  = regf_rdoutN;
            dmem_din   = regf_rdoutM;
        end
        `WB: begin
            // Read Rn
            regf wren = 1;
            regf raddrN = INSTR[`OP1];
            // write back early
            dmem wren  = 1;
            dmem_addr  = regf_rdoutN;
            dmem_din   = regf_rdoutM;
        end
    endcase
end
3: begin // MOV Rn, #immed; |0x3|Rn||Imm|edt|; Rn = #immed
    case (phase)
        `ID: begin
            // Directly write to Rn and be done with it
            regf wren = 1;
            regf wdin = INSTR[`OP2 DIRECT];
            regf waddr= INSTR[`OP1 DIRECT];
        end
    endcase
end
4: begin // ADD Rn, Rm    ; |0x4|  ||Rn |Rm |; Rn = Rn + Rm
    case (phase)
        `ID: begin
            // Read Rn and Rm
            regf_wren   = 0;
            regf_raddrN = INSTR[`OP1];
            regf raddrM = INSTR[`OP2];
        end
        `EX: begin
            // Read Rn and Rm
            regf_wren   = 0;
            regf_raddrN = INSTR[`OP1];
            regf_raddrM = INSTR[`OP2];
            // Select add
            add0_sub1   = 0;
            A           = regf_rdoutN;
            B           = regf rdoutM;
        end
        `WB: begin
            // Read Rn and Rm
            regf wren   = 0;
            regf raddrN = INSTR[`OP1];
            regf raddrM = INSTR[`OP2];
            // Select add
            add0_sub1   = 0;
            A           = regf_rdoutN;
            B           = regf rdoutM;
            // Update Rn (OP1) in regfile
            regf wren   = 1;
            regf waddr  = INSTR[`OP1];
            regf wdin   = O;
        end
    endcase
end
5: begin // SUB Rn, Rm    ; |0x5|  ||Rn |Rm |; Rn = Rn - Rm
    case (phase)
        `ID: begin
            // Read Rn and Rm
            regf_wren   = 0;
            regf_raddrN = INSTR[`OP1];
            regf_raddrM = INSTR[`OP2];
        end
        `EX: begin
            // Read Rn and Rm
            regf wren   = 0;
            regf_raddrN = INSTR[`OP1];
            regf_raddrM = INSTR[`OP2];
            // Select add
            add0_sub1   = 1;
            A           = regf_rdoutN;
            B           = regf_rdoutM;
        end
        `WB: begin
            // Read Rn and Rm
            regf wren   = 0;
            regf raddrN = INSTR[`OP1];
            regf raddrM = INSTR[`OP2];
            // Select add
            add0 sub1   = 1;
            A           = regf_rdoutN;
            B           = regf_rdoutM;
            // Update Rn (OP1) in regfile
            regf wren   = 1;
            regf waddr  = INSTR[`OP1];
            regf wdin   = O;
        end
    endcase
```

```
            end
            8: begin // JZ  Rn, reltiv; |0x8|R4||rel|tiv|; PC = PC + relative
                case (phase)
                    `ID: begin
                        // Read Rn and relative value (DIRCT format)
                        regf_wren   = 0;
                        regf_raddrN = INSTR[`OP1_DIRECT];
                    end
                    `EX: begin
                        // Read Rn and relative value (DIRCT format)
                        regf_wren   = 0;
                        regf_raddrN = INSTR[`OP1_DIRECT];
                        if (regf_rdoutN != 0) begin
                            pc_incr = INSTR[`OP2_DIRECT];
                        end
                    end
                    `WB: begin
                        // Read Rn and relative value (DIRCT format)
                        regf_wren   = 0;
                        regf_raddrN = INSTR[`OP1_DIRECT];
                        if (regf_rdoutN == 0) begin
                            pc_incr = INSTR[`OP2_DIRECT];
                        end
                    end
                endcase
            end
            9: begin // JZ  Rn, reltiv; |0x8|R4||rel|tiv|; PC = PC + relative
                case (phase)
                    `ID: begin
                        // Read Rn and relative value (DIRCT format)
                        regf_wren   = 0;
                        regf_raddrN = INSTR[`OP1_DIRECT];
                    end
                    `EX: begin
                        // Read Rn and relative value (DIRCT format)
                        regf_wren   = 0;
                        regf_raddrN = INSTR[`OP1_DIRECT];
                        if (regf_rdoutN != 0) begin
                            pc_incr = INSTR[`OP2_DIRECT];
                        end
                    end
                    `WB: begin
                        // Read Rn and relative value (DIRCT format)
                        regf_wren   = 0;
                        regf_raddrN = INSTR[`OP1_DIRECT];
                        if (regf_rdoutN != 0) begin
                            pc_incr = INSTR[`OP2_DIRECT];
                        end
                    end
                endcase
            end
            default: begin
                case (phase)
                    `ID, `EX, `WB: begin
                        $error("Invalid OPCODE used!!\n");
                        $finish;
                    end
                endcase
            end
        endcase
    end: decode_ex_logic_comb

endmodule: simple_decode_ex
```

Code 7 code for decode and execution logic

```
// ------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------
// Top level hookup

module simple_cpu (
    input   clk
,   input   resetn
);
    // Interconnect wires
    // PC block
    wire signed [7:0] pc_incr;
    wire [7:0] pc; // PC can address 256 bytes;
    // Instr mem
    wire   [7:0]  instr_addr;
    wire   [15:0] INSTR     ;
    // Dmem
    wire   dmem_wren      ;
    wire   [7:0] dmem_addr;// common addr port for read and write
    wire   [7:0] dmem_din ;
    wire   [7:0] dmem_dout;
    // regfile
    wire regf_wren;
    wire [3:0] regf_raddrN;
    wire signed [7:0] regf_rdoutN;
    wire [3:0] regf_raddrM;
    wire signed [7:0] regf_rdoutM;
    wire [3:0] regf_waddr;
    wire signed [7:0] regf_wdin;
    // Phase_ctr
    wire [1:0] phase;
    // Alu
    wire [7:0] A, B, O;
    wire       add0_sub1;

    // PC block
    simple_pc u_pc (
        .clk    (clk)
    ,   .resetn (resetn)
    ,   .phase  (phase)
    ,   .pc_incr(pc_incr)
    ,   .pc     (pc));

    // Instr mem
    simple_instr_mem u_instr_mem (
        .clk        (clk)
    ,   .resetn     (resetn)
    ,   .instr_addr (pc)
    ,   .INSTR      (INSTR));

    // Dmem
    simple_dmem u_dmem (
        .clk        (clk)
    ,   .resetn     (resetn)
    ,   .dmem_wren  (dmem_wren)
    ,   .dmem_addr  (dmem_addr)
    ,   .dmem_din   (dmem_din)
    ,   .dmem_dout  (dmem_dout));

    // Regfile
    simple_regfile u_regfile (
        .clk        (clk)
    ,   .resetn     (resetn)
    ,   .regf_wren  (regf_wren)
    ,   .regf_raddrN(regf_raddrN)
    ,   .regf_rdoutN(regf_rdoutN)
    ,   .regf_raddrM(regf_raddrM)
    ,   .regf_rdoutM(regf_rdoutM)
    ,   .regf_waddr (regf_waddr)
    ,   .regf_wdin  (regf_wdin));

    // decode-ex
    simple_decode_ex u_decode_ex (.*);

    // Phase
    simple_phase_ctr u_phase (
        .clk    (clk)
    ,   .resetn (resetn)
    ,   .phase  (phase));

    // Alu
    simple_alu u_alu (
        .clk (clk)
    ,   .resetn(resetn)
    ,   .add0_sub1(add0_sub1)
    ,   .A(A)
    ,   .B(B)
    ,   .O(O));

endmodule: simple_cpu
```

Code 8 code for simple_cpu top-level glue block

```
#ifndef _SCGENMOD_simple_cpu_
#define _SCGENMOD_simple_cpu_

#include "systemc.h"

class simple_cpu : public sc_foreign_module
{
public:
    sc_in<bool> clk;
    sc_in<bool> resetn;


    simple_cpu(sc_module_name nm, const char* hdl_name)
     : sc_foreign_module(nm),
       clk("clk"),
       resetn("resetn")
    {
        elaborate_foreign_module(hdl_name);
    }
    ~simple_cpu()
    {}

};

#endif
```

Code 9 systemC code for the header file of simple_cpu in c++ for the sc_foreign_module simple_cpu

```
#include "systemc.h"
#include <iostream>
#include "simple_cpu.h"

SC_MODULE(simple_tb) {
    sc_clock clk;
    sc_event reset_deactivation_event;
    sc_signal<bool> resetn;

    int counter;

    // module instances
    simple_cpu* u_simple_cpu;

    void reset_generator();

    // Constructor
    SC_CTOR(simple_tb)
        : clk ("clock", 10, SC_NS, 0.5, 0.0, SC_NS, false)
        , resetn("resetn")
    {
        // Create instances
        u_simple_cpu = new simple_cpu("u_simple_cpu", "simple_cpu");
        u_simple_cpu->clk(clk);
        u_simple_cpu->resetn(resetn);

        SC_METHOD(reset_generator);
        sensitive << reset_deactivation_event;
    }

    // Destructor
    ~simple_tb() {
        delete u_simple_cpu; u_simple_cpu = 0;
    }
};

inline void simple_tb::reset_generator() {
    static bool first = true;
    if (first) {
        first = false;
        resetn.write(0);
        reset_deactivation_event.notify(20, SC_NS);
    } else {
        resetn.write(1);
    }
}

SC_MODULE_EXPORT(simple_tb);
```

Code 10 systemC code for the simple_tb testbench module

Simulation result

The code was simulated using ModelSim SE 2019.2 version. And the behaviour of the module was found to match with that of the systemVerilog model exactly

Below are the waveforms for proof

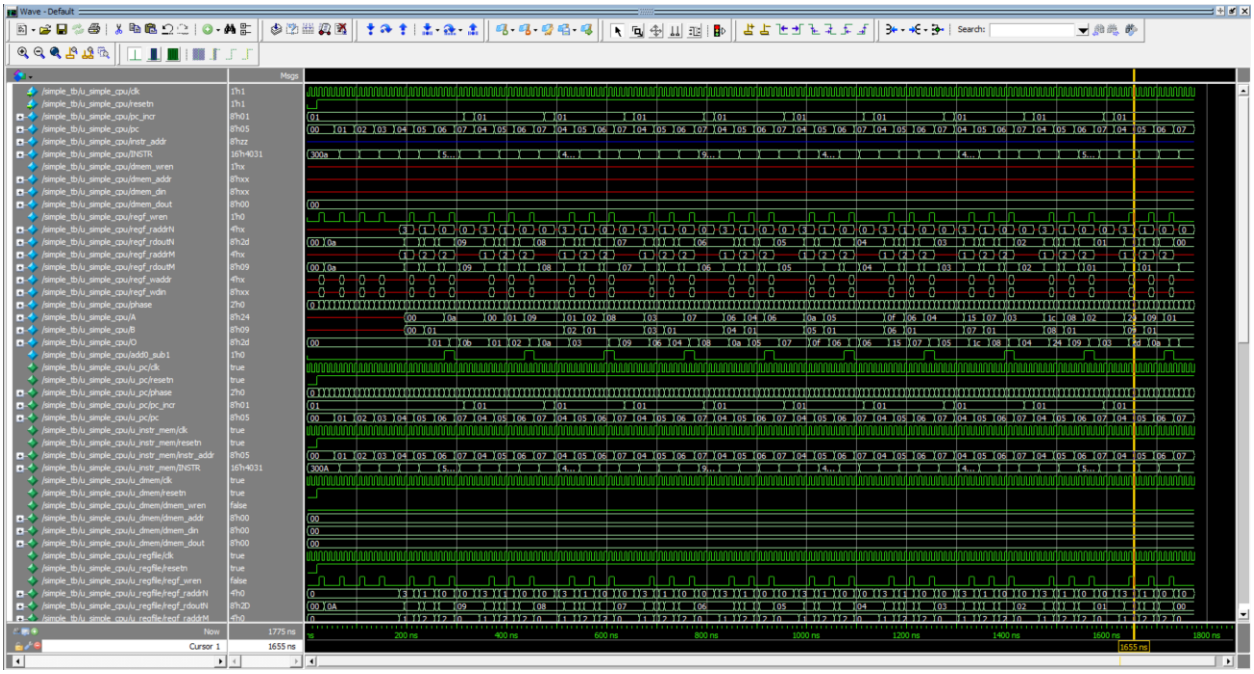


Figure 10 Waveform (part 1) of the systemC design of simple_cpu

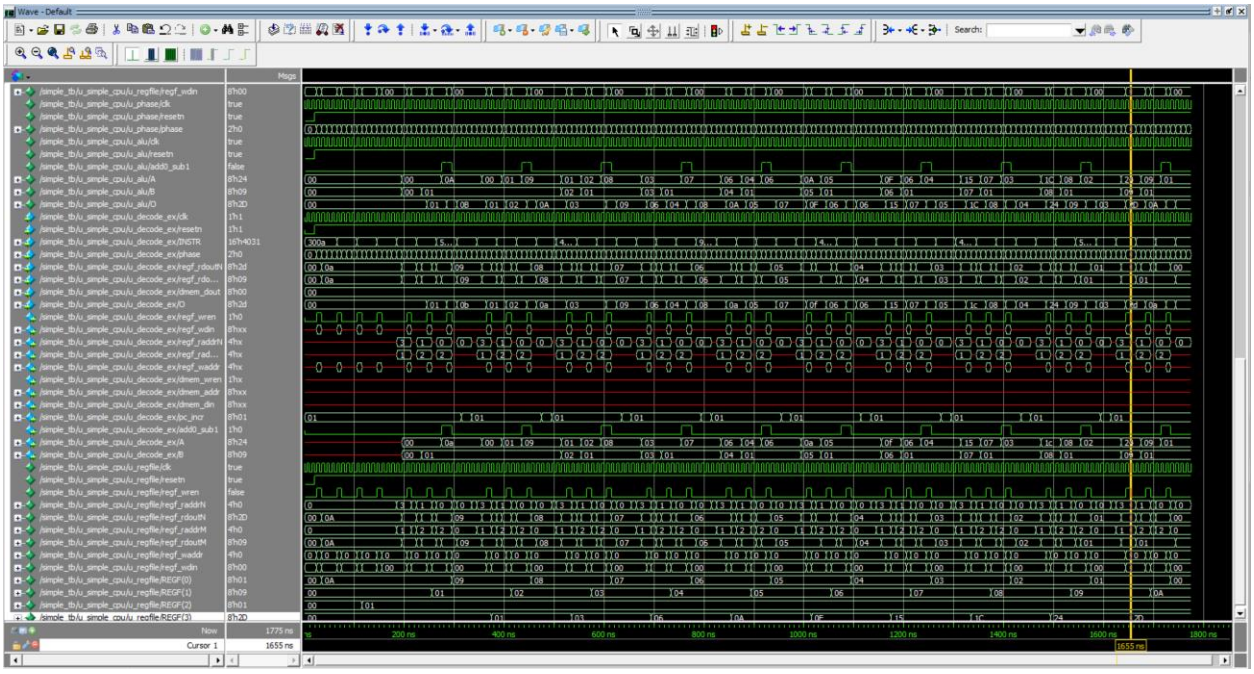


Figure 11 Waveform (part 2) of the systemC design of simple_cpu

Below is the run.do file used to simulate the code

```
# THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
# WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION
# OR ITS LICENSORS AND IS SUBJECT TO LICENSE TERMS.

# Use this run.do file to run this example.
# Either bring up ModelSim and type the following at the "ModelSim>" prompt:
#     do run.do
# or, to run from a shell, type the following at the shell prompt:
#     vsim -do run.do -c
# (omit the "-c" to see the GUI while running from the shell)

onbreak {resume}

# create library
if [file exists work] {
    vdel -all
}
vlib work

# compile all Verilog source files
vlog -sv ../sv/flist.v -define SC

# compile sc files
sccom -g simple_phase_ctr.cpp
sccom -g simple_alu.cpp
sccom -g simple_pc.cpp
sccom -g simple_instr_mem.cpp
sccom -g simple_dmem.cpp
sccom -g simple_regfile.cpp
# create simple_cpu.h
scgenmod -bool simple_cpu > simple_cpu.h
sccom -g simple_tb.cpp
sccom -link

# open debugging windows
quietly view *

# start and run simulation
vsim -vopt work.simple_tb -voptargs="+acc" -vv

# Add waves
do wave.do
# add wave -r /simple_tb/u_simple_cpu/*
# add wave -r /simple_tb/u_phase/*

# Run command
run 500000 ns
```

Figure 12 run.do file for the systemC simulation in modelsim

APPENDIX

SystemVerilog code for the HDL design

```
// ------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------

module simple_alu (
    input   clk
,   input   resetn
,   input   logic add0_sub1
,   input   logic [7:0] A
,                       B
,   output  logic [7:0] O
);

    always@(posedge clk) begin
        if (add0_sub1) begin
            O <= A - B;
        end else begin
            O <= A + B;
        end
    end
endmodule: simple_alu
```

Code 11 SV code for ALU of the simple_cpu

```
// ------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------
// Specification : RAM with a sequential read port - 1 cycled delay
// Either reads or writes possible at any given time.

module simple_dmem (
    input   clk
,   input   resetn
,   input   logic   dmem_wren
,   input   logic   [7:0] dmem_addr // common addr port for read and write
,   input   logic   [7:0] dmem_din
,   output  logic   [7:0] dmem_dout
);

    logic [7:0] dmem [256];

    always_ff @(posedge clk) begin
        dmem_dout           <= dmem[dmem_addr];
        if (dmem_wren)
            dmem[dmem_addr] <= dmem_din;
    end

endmodule: simple_dmem
```

Code 12 SV code for simple_dmem: The 8x256 data memory

```
// ------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------
// Specification : ROM with a sequential read port - 1 cycled delay

module simple_instr_mem (
    input   clk
,   input   resetn
,   input   logic   [7:0]  instr_addr // Instr mem is 512 bytes deep (256 Half-words)
,   output  logic   [15:0] INSTR      // Each instr is 2 bytes long
);

    // The instr mem
    logic [15:0] INSTR_MEM [256];

    always_comb begin
        INSTR_MEM[0] = 'h300a; /* 0: MOV R0, 0xa; Setup                                          */
        INSTR_MEM[1] = 'h3100; /* 1: MOV R1, 0x0; Setup                                          */
        INSTR_MEM[2] = 'h3201; /* 2: MOV R2, 0x1; Setup                                          */
        INSTR_MEM[3] = 'h3300; /* 3: MOV R3, 0x0; Setup                                          */
        INSTR_MEM[4] = 'h4031; /* 4: ADD R3, R1 ; R3 = R3 + R1        -- R3 accumulates the sum  */
        INSTR_MEM[5] = 'h4012; /* 5: ADD R1, R2 ; R1 = R1 + 1; INC R1 -- New num created by incr of R1*/
        INSTR_MEM[6] = 'h5002; /* 6: SUB R0, R2 ; R0 = R0 - 1; DEC R1                            */
        INSTR_MEM[7] = 'h90fd; /* 7: JNZ R0, -3 ; PC = PC + (-3) if R0 != 0                      */
    end

    // Sequential read
    always_ff @(posedge clk)
        INSTR <= INSTR_MEM[instr_addr];

endmodule: simple_instr_mem
```

Code 13 SV code for the instruction memory, with application SW of PartC loaded

```
// ------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------

module simple_pc (
    input   clk
,   input   resetn
,   input   logic [1:0] phase
,   input   logic signed [7:0] pc_incr

,   output  logic [7:0] pc // PC can address 256 bytes

);

    always_ff @(posedge clk, negedge resetn) begin: pc_incr_logic_seq
        if (!resetn) begin
            pc <= '0;
        end else begin
            if (phase == 2'b11)
                pc <= signed'(pc + pc_incr);
        end
    end: pc_incr_logic_seq
    always_comb begin: pc_wdog
        if (pc >= 8 && resetn === 1) begin
            $info("PC value = %0d. Exiting.", pc);
            $finish;
        end
    end: pc_wdog

endmodule: simple_pc
```

Code 14 SV code for the PC register and incrementing logic

```
// ------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------
// Has 4 phases - IF, ID, EX, WB
// The FSM uses this ctr to sequence the signals

module simple_phase_ctr (
    input   clk
,   input   resetn
,   output  logic [1:0] phase
);

    always_ff @(posedge clk, negedge resetn)
        if (!resetn)
            phase <= 0;
        else
            phase <= phase + 1;

endmodule: simple_phase_ctr
```

Code 15 SV code for the phase incrementor part of the FSM

```
// ------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------
// Specification : RAM with two sequential read port - 1 cycle delay
// And 1 sequential write port.
// Either reads or writes possible at any given time.

module simple_regfile (
    input   clk
,   input   resetn
,   input   logic regf_wren

// Rd port - N
,   input   logic [3:0] regf_raddrN
,   output  logic signed [7:0] regf_rdoutN

// Rd port - M
,   input   logic [3:0] regf_raddrM
,   output  logic signed [7:0] regf_rdoutM

// Wr port - common
,   input   logic [3:0] regf_waddr
,   input   logic signed [7:0] regf_wdin
);

    logic [7:0] REGF [16];

    always_ff @(posedge clk)
        if (regf_wren)
            REGF[regf_waddr] <= regf_wdin;

    always_ff @(posedge clk) begin
        regf_rdoutN <= REGF[regf_raddrN];
        regf_rdoutM <= REGF[regf_raddrM];
    end

endmodule: simple_regfile
```

Code 16 SV code for the register file hardware

```
// ------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------
// Specification : RAM with two sequential read port - 1 cycle delay
// And 1 sequential write port.
// Either reads or writes possible at any given time.

module simple_decode_ex (
    input   clk
,   input   resetn

// Ctrl inputs
// -- instr
,   input   logic   [15:0]INSTR
// -- phase
,   input   logic   [1:0] phase
// -- regf
,   input   logic   [7:0] regf_rdoutN
,   input   logic   [7:0] regf_rdoutM
// -- dmem
,   input   logic   [7:0] dmem_dout
// -- alu
,   input   logic   [7:0] O

// Ctrl outputs
// -- regf
,   output  logic         regf_wren
,   output  logic   [7:0] regf_wdin
,   output  logic   [3:0] regf_raddrN
,                         regf_raddrM
,                         regf_waddr
// -- dmem
,   output  logic         dmem_wren
,   output  logic  [7:0]  dmem_addr
,   output  logic  [7:0]  dmem_din
// -- pc
,   output  logic  [7:0]  pc_incr
// -- alu
,   output  logic         add0_sub1
,   output  logic  [7:0]  A
,                         B
);
    `define OPCODE 15:12
    `define OP1_DIRECT 11:8
    `define OP2_DIRECT 7:0
    `define OP1 7:4
    `define OP2 3:0
    `define IF 0
    `define ID 1
    `define EX 2
    `define WB 3

    // This is a pure combinational block - only drives the output as required
    // for a given phase based on extensive muxing on the phase and the opcode
    always comb begin: decode ex logic comb
        // defaults
        regf_wren   = 0;
        regf_wdin   = 'x;
        regf_raddrN = 'x;
        regf_raddrM = 'x;
        regf_waddr  = 'x;

        dmem_wren   = 'x;
        dmem_addr   = 'x;
        dmem_din    = 'x;

        pc_incr     = 1;
        add0_sub1   = 0;

        // conditional updates
        case (INSTR[`OPCODE])
            0: begin // MOV Rn, direct; |0x0|Rn||dir|ect|; Rn = M(direct)
                case (phase)
                    `ID: begin
                        // Read M(direct)
                        dmem_wren = 0;
                        dmem_addr = INSTR[`OP2_DIRECT];
                    end
                    `EX: begin
                        // Read M(direct)
                        dmem_wren = 0;
                        dmem_addr = INSTR[`OP2_DIRECT];
                    end
                    `WB: begin
                        // Read M(direct)
                        dmem_wren = 0;
                        dmem_addr = INSTR[`OP2_DIRECT];
                        // Write to Rn
                        regf_wren = 1;
                        regf_wdin = dmem_dout;
                        regf_waddr= INSTR[`OP1_DIRECT];
                    end
                endcase
            end
            1: begin // MOV direct, Rn; |0x1|Rn||dir|ect|; M(direct) = Rn
            end
            2: begin // MOV @Rn,Rm    ; |0x2|  ||Rn |Rm |; M(Rn) = Rm
            end
            3: begin // MOV Rn, #immed; |0x3|Rn||Imm|edt|; Rn = #immed
                case (phase)
                    `ID: begin
                        // Directly write to Rn and be done with it
                        regf_wren = 1;
                        regf_wdin = INSTR[`OP2_DIRECT];
                        regf_waddr= INSTR[`OP1_DIRECT];
                    end
                endcase
            end
            4: begin // ADD Rn, Rm    ; |0x4|  ||Rn |Rm |; Rn = Rn + Rm
                case (phase)
                    `ID: begin
                        // Read Rn and Rm
                        regf wren   = 0;
                        regf raddrN = INSTR[`OP1];
                        regf_raddrM = INSTR[`OP2];
                    end
                    `EX: begin
                        // Read Rn and Rm
                        regf_wren   = 0;
                        regf_raddrN = INSTR[`OP1];
                        regf_raddrM = INSTR[`OP2];
                        // Select add
                        add0_sub1   = 0;
                        A           = regf rdoutN;
                        B           = regf_rdoutM;
                    end
                    `WB: begin
                        // Read Rn and Rm
                        regf_wren   = 0;
                        regf_raddrN = INSTR[`OP1];
                        regf_raddrM = INSTR[`OP2];
                        // Select add
                        add0_sub1   = 0;
                        A           = regf_rdoutN;
                        B           = regf_rdoutM;
                        // Update Rn (OP1) in regfile
                        regf_wren   = 1;
                        regf_waddr  = INSTR[`OP1];
                        regf_wdin   = O;
                    end
                endcase
```

```
            end
            5: begin // SUB Rn, Rm    ; |0x5|  ||Rn |Rm |; Rn = Rn - Rm
                case (phase)
                    `ID: begin
                        // Read Rn and Rm
                        regf_wren   = 0;
                        regf_raddrN = INSTR[`OP1];
                        regf_raddrM = INSTR[`OP2];
                    end
                    `EX: begin
                        // Read Rn and Rm
                        regf wren   = 0;
                        regf_raddrN = INSTR[`OP1];
                        regf_raddrM = INSTR[`OP2];
                        // Select add
                        add0_sub1   = 1;
                        A           = regf_rdoutN;
                        B           = regf_rdoutM;
                    end
                    `WB: begin
                        // Read Rn and Rm
                        regf_wren   = 0;
                        regf_raddrN = INSTR[`OP1];
                        regf_raddrM = INSTR[`OP2];
                        // Select add
                        add0_sub1   = 1;
                        A           = regf_rdoutN;
                        B           = regf rdoutM;
                        // Update Rn (OP1) in regfile
                        regf_wren   = 1;
                        regf_waddr  = INSTR[`OP1];
                        regf_wdin   = O;
                    end
                endcase
            end
            8: begin // JZ  Rn, reltiv; |0x8|R4||rel|tiv|; PC = PC + relative
            end
            9: begin // JZ  Rn, reltiv; |0x8|R4||rel|tiv|; PC = PC + relative
                case (phase)
                    `ID: begin
                        // Read Rn and relative value (DIRCT format)
                        regf_wren   = 0;
                        regf_raddrN = INSTR[`OP1_DIRECT];
                    end
                    `EX: begin
                        // Read Rn and relative value (DIRCT format)
                        regf_wren   = 0;
                        regf_raddrN = INSTR[`OP1_DIRECT];
                        if (regf_rdoutN != 0) begin
                            pc_incr = INSTR[`OP2_DIRECT];
                        end
                    end
                    `WB: begin
                        // Read Rn and relative value (DIRCT format)
                        regf wren   = 0;
                        regf_raddrN = INSTR[`OP1_DIRECT];
                        if (regf rdoutN != 0) begin
                            pc_incr = INSTR[`OP2_DIRECT];
                        end
                    end
                endcase
            end
            default: begin
                case (phase)
                    `ID, `EX, `WB: begin
                        $error("Invalid OPCODE used!!\n");
                        $finish;
                    end
                endcase
            end
        endcase
    end: decode_ex_logic_comb

endmodule: simple_decode_ex
```

Code 17 SV code for the instruction decode and control logic

```
// ------------------------------------------------------------------
// HSCD Assignment - 1
// AUTHOR: Anand S
// BITS ID: 2021HT80003
// ------------------------------------------------------------------
// Top level hookup

module simple_cpu (
    input   clk
,   input   resetn
);
    // Interconnect wires
    // PC block
    wire signed [7:0] pc_incr;
    wire [7:0] pc; // PC can address 256 bytes;
    // Instr mem
    wire   [7:0]  instr_addr;
    wire   [15:0] INSTR     ;
    // Dmem
    wire   dmem_wren      ;
    wire   [7:0] dmem_addr;// common addr port for read and write
    wire   [7:0] dmem_din ;
    wire   [7:0] dmem_dout;
    // regfile
    wire regf_wren;
    wire [3:0] regf_raddrN;
    wire signed [7:0] regf_rdoutN;
    wire [3:0] regf_raddrM;
    wire signed [7:0] regf_rdoutM;
    wire [3:0] regf_waddr;
    wire signed [7:0] regf_wdin;
    // Phase_ctr
    wire [1:0] phase;
    // Alu
    wire [7:0] A, B, O;
    wire       add0_sub1;

    // PC block
    simple_pc u_pc (
        .clk    (clk)
    ,   .resetn (resetn)
    ,   .phase  (phase)
    ,   .pc_incr(pc_incr)
    ,   .pc     (pc));

    // Instr mem
    simple_instr_mem u_instr_mem (
        .clk        (clk)
    ,   .resetn     (resetn)
    ,   .instr_addr (pc)
    ,   .INSTR      (INSTR));

    // Dmem
    simple_dmem u_dmem (
        .clk        (clk)
    ,   .resetn     (resetn)
    ,   .dmem_wren  (dmem_wren)
    ,   .dmem_addr  (dmem_addr)
    ,   .dmem_din   (dmem_din)
    ,   .dmem_dout  (dmem_dout));

    // Regfile
    simple_regfile u_regfile (
        .clk        (clk)
    ,   .resetn     (resetn)
    ,   .regf_wren  (regf_wren)
    ,   .regf_raddrN(regf_raddrN)
    ,   .regf_rdoutN(regf_rdoutN)
    ,   .regf_raddrM(regf_raddrM)
    ,   .regf_rdoutM(regf_rdoutM)
    ,   .regf_waddr (regf_waddr)
    ,   .regf_wdin  (regf_wdin));

    // decode-ex
    simple_decode_ex u_decode_ex (.*);

    // Phase
    simple_phase_ctr u_phase (
        .clk    (clk)
    ,   .resetn (resetn)
    ,   .phase  (phase));

    // Alu
    simple_alu u_alu (
        .clk (clk)
    ,   .resetn(resetn)
    ,   .add0_sub1(add0_sub1)
    ,   .A(A)
    ,   .B(B)
    ,   .O(O));

endmodule: simple_cpu
```

Code 18 SV code for the top-level module for the simple_cpu

```
module simple_tb;

    // Wires
    reg clk, resetn;
    wire [1:0] phase;
    // DUT
    simple_cpu u_simple_cpu (.*);

    initial begin
        clk <= 0;
        forever #5 clk = ~ clk;
    end

    initial begin: seq
        resetn = 0;
        #10;
        resetn = 1;
    end: seq

    initial begin: finish
        #2000;
        $finish;
    end: finish
endmodule: simple_tb
```

Code 19 SV code for the testbench

THE END.

ANAND S

2021HT80003