BAME2044 ASIC AND FPGA DESIGN (Practical Rubrics)

CLO3: Construct Combinational Logic Circuits using digital logic techniques based on SSI/ MSI/ LSI implementation. (P, P4) -20%

Name: _____ **Practical P__ TOTAL**: _____

Presentation Topic (Assignment/ Mini Project/ Lab): _____

PLO2: Practical Skills (Report)

No.	Aspect	Very Poor	Rating: 1-Very Poor, 2-Poor, 3-Average, 4-Good, 5-Excellent					Excellent	Marks
a.	Introduction (5%)	Limited understanding of practical work, superficial/ incorrect information.	1	2	3	4	5	Concise and precise, demonstrate in-depth knowledge of the subject matter (using various highly relevent references)	
b.	Methods (10%)	Superficial/ limited description of system. No evidence of systematic approach in problem solving. Info presentation is not organized.	1-2	3-4	5-6	7-8	9-10	In depth description of system components/ overall systems. Extensive / proficient use of proper guided techniques to solve problems. Method is fully reproducible. Presentation is clear and critically structured.	
c.	Results and Discussion (5%)	Superficial Tests, inaccurate/ ambiguos presentation of results. Irrelevant/ superficial discussion of system performance. No evidence of working solution that meets basic requirements.	1	2	3	4	5	Rigorous Testing for all key parameters, Acurate results, clearly presented & annotated, critical result analysis leading to rigorous discussion in meeting system performance, presents very good working solution.	

Methods of Assessment: 1 Concise Integrated/ Combined *Report*

Assessed by,

_____

Course Leader: MICHELLE LIM SERN MI

Date:

Total /20

CLO3: Construct Combinational Logic Circuits using digital logic techniques based on SSI/ MSI/ LSI implementation. (P, P4) -80%

PLO2: Practical Skills (Hands-on)

No.	Aspect	Poor	Rating: 1-Very Poor, 2-Poor, 3-Average, 4-Good, 5-Excellent					Excellent	(Rating x4) Marks
a.	Autonomous Learning (20%)	Very limited/ superficial understanding of work done (Able to answer BASIC questions only). Requires extensive referencing, a lot of prompting, very slow response. No confidence shown.	1	2	3	4	5	Demonstrate in-depth know-how on work done (able to answer any questions within the scope of work done precisely, concisely, and confidently). Display knowledge beyond what is required.	
b.	Proficiency in modern tool usage (20%)	Extremely limited understanding of developmet tools/ HDL code syntax. Requires extensive instructor assistance.	1	2	3	4	5	Highly Proficient, completes hands-on lab work within 1.5 hours. No assistance needed	
c.	System Design/ Construction (20%)	Could not/ Wrongly synthesized design/ Badly organized design modules, do not properly comment/ name interconnects in the design. No/ poor design planning.	1	2	3	4	5	Well synthesized design. Limited guidance by instructor. Properly organized design modules with proper naming conventions. Extensive/ Proper design planning done.	
d.	Testing/ Functionality (20%)	Unable to test the system. No/ limited proficiency to use any techniques/ create a test bench to test the synthesized system. Requires a lot of assistance. Design is non-functional.	1	2	3	4	5	Proper test bench/ testing techniques used with good proficiency. Proper testing procedures is followed. System is functioning well exactly as required of the specification with no ambiguity.	

Methods of Assessment: *Practical Lab Activities from P2, P3, P6, P7, P8*

Total /80

Practical P( ) TOTAL /100

Sample Title Page

DESIGN OF A SIMPLE PROCESSOR

Lim Sheng Yang (09WTD012345)

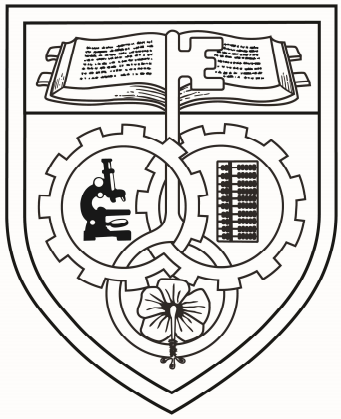
Assignment submitted in partial fulfillment of the requirements for
ASIC and FPGA Design (BAME2044) course of the Bachelor in Science
(Microelectronics with Embedded Technology)

Department of Computer Science
Faculty of Computing and Information Technology (FOCS)
TARC University College
Kuala Lumpur

OCT 2018

LABORATORY MANUAL

FACULTY OF COMPUTING AND INFORMATION TECHNOLOGY (FOCS)



TARC

TUNKU ABDUL RAHMAN UNIVERSITY COLLEGE

BAME 2044

ASIC AND FPGA DESIGN

Revised
Academic Year 2018/19

ASSIGNMENT FORMAT

!!! EVERY SENTENCE SHOULD BE PROVEN WITH A PROPER REFERENCE/ THEORY/ EQUATION/ CALCULATIONS (CITE ALL YOUR REFERENCES)

LAB REPORT (SUGGESTED FORMAT)

CONTENTS

ASSIGNMENT FORMAT (SAMPLE)

LAB REPORT (EXAMPLE FORMAT)

CONTENTS

1. INTRODUCTION

1.1 OBJECTIVES (POINT FORM)

1. To investigate several different topologies/ architectures/ design methods of …….. from past literatures.

2. To model, design and verify using HDL based test-bench all sub-modules of the …… using in Mentor Graphics environment.

3. To integrate, analyze and perform hardware implementation on all the integrated sub-modules of the ……design on the Cyclone II FPGA.

1.2 PROBLEM STATEMENT

1.3 BRIEF BACKGROUND (HISTORY, DEVELOPMENT, CURENT TRENDS, APPLICATION)

The first idea of the……. design was first suggested by……..(Sam,1922).....(…….History…… ……).

Currently, the trend of this …….. design is fast moving towards …… due to the progress in …….. (Reference, Year). (…… Talk about the current research trends of your topic……..). Table I shows a comparison table on past researcher's work regarding …….. From this table, we note that the …… design by Bandy achieves the lowest minimum startup voltage at ……. while Kim *et al.*'s design has …… The techniques used to achieve…… is ……..(…….continue discussing past techniques and topologies regarding ……. design…….).

TABLE I. PAST RESEARCHER'S WORK ON .....

Reference [Year]	*fsw*	Charge Pump Startup			Peak Efficiency	V_{OUT}	P_{MAX}
		Mechanism [Ext. V?]	V_{MIN}	Startup Time			
Bandy [2013]	12.8 Hz	One time Wireless Charging Scheme (Antenna) [No]	20mV	0.1 ms	89% pump efficiency	1.5V-1.9V	4 nW
Peng *et al.*, [2014]	87 kHz	6-stages dual branch with reverse control CTS with sub-threshold and body-biasing regime [No]	320 mV	0.1ms	89%	2.72V	12 mW
Kim *et al.*, [2015]	250 kHz	3 Stage Cross Coupled CP using DBB technique	150 mV	N/A	72.5% at $0.45V_{IN}$	0.619V at V_{IN}= 0.18V	21 µA at 0.18V

Fig. 1.1 on the other hand shows the past architecture of ……. which illustrates how a ……. operates. (…… …….. discuss past researcher's design on …….. topic……..).

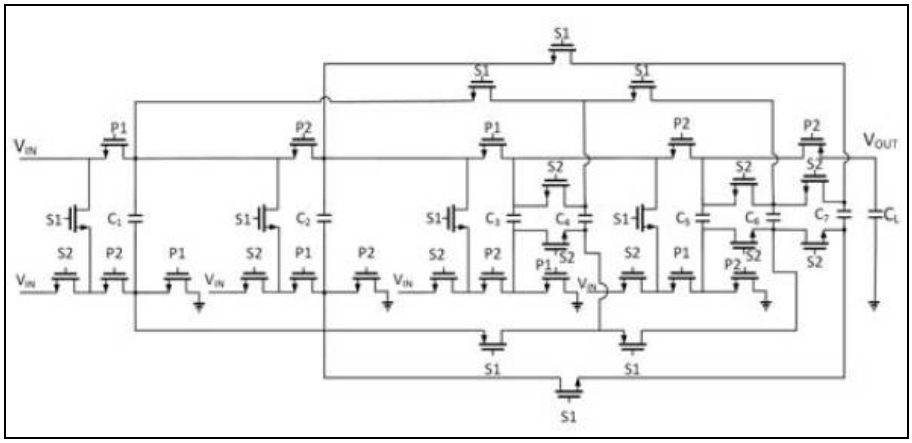


Fig. 1. Past researcher's design on ……. (Kumar, 2013)

In a nutshell, this report is a design and development of a……..implemented on……..It begins with an introduction to the background, past to current research on the topologies and strategies of …….. design in Chapter 1. Later, Chapter 2 proposes the design methodology while the results and discussion are summarized in Chapter 3 and finally conclude in Chapter 4.

2. METHODOLOGY

2.1 FLOW CHART (OVERALL PROCESS FLOW)

* SKETCH OF A FLOW CHART BASED ON METHODOLOGY FLOW
* BRIEF EXPLANATION OF FLOW CHART
* ASM CHART/ PROGRAM FLOW RELATED TO DESIGN
* TECHNIQUE/ APPROACH OF DESIGN (EQUATION/ VERIFICATION PLATFORM/ METHOD)
* STATE TABLE/ TRUTH TABLE ETC.
* CODING METHODOLOGY

2.3 INSTANTIATION OF SUB-MODULES (BLOCK/ HDL/ TESTBENCHCODE)

* PROPOSED BLOCK DIAGRAM OF DESIGN (OVERALL) TOP-DOWN DESIGN
* SHOW PROPOSED SUB-BLOCKS OF DESIGN
* RELATE HARDWARE BLOCK TO HDL BASED MODULES/ SUB-MODULES
* VERIFICATION PLAN (TABLE FORMAT)
* TEST BENCH CODING METHODOLOGY (LONG PROGRAMMES CAN BE ATTACHED AS APPENDIXES)

3. RESULTS & DISCUSSION

3.1 RESULTS AND DISCUSSION OF SUB-BLOCKS

3.2 RESULTS AND DISCUSSION OF INTERGATED SYSTEM

* USE FIGURES TO SHOW WAVEFORM RESULTS (FUNCTIONAL VERIFICATION)
* USE FIGURES TO SHOW WAVEFORM RESULTS (TIMING VERIFICATION)
* SUMMARIZE INFO INTO A PROPER TABLE.
* COMPARE PROPOSED DESIGN TO PAST DESIGNS OR AN ALTERNATIVE DESIGN
* COMPARE IN TERMS OF PARAMETERS RELATED TO THE SYSTEM
  ** SPEED/ POWER/ AREA/ FAN-OUT/ LOGIC ELEMENTS ETC.
* PERFORM CHANGES IN PARAMETERS (PARAMETRIC ANALYSIS) ON THE SYSTEM
  ** CHANGE OF FREQ., VOLTAGE SUPPLY, AREA VS SPEED OPTIMIZATION ETC.
* ANALYZE DATA FROM TABLE & PLOTS & PARAMETRIC ANALYSIS
* PROVIDE A CRITICAL DISCUSSION ON HOW YOUR DESIGN PERFORMED IN TERMS OF SPEED/ POWER/ AREA AS COMPARED TO PAST DESIGNS OR PEER'S DESIGN.
*END WITH A **SPECIFICATION TABLE** OF YOUR DESIGN.

4. CONCLUSION

* IN ONE PARAGRAPH CONCLUDE THE REPORT.
* THIS PARAGRAPH MUST INCLUDE THE FOLLOWING:-
  ** INTRODUCE YOUR DESIGN.
  ** A PROBLEM STATEMENT (IF APPLICAPLE)
  ** OBJECTIVES
  ** SURMMARIZED DESIGN METHODOLOGY
  ** FINAL RESULTS & CONCLUDING ANALYSIS
  ** APPLICATION/ FUTURE TREND/ RECOMMENDATION

EXAMPLE:

A near optimal (***your design)*** architecture has been proposed with ……(***critical description of your system design***)…….. . The main aim is to …….. (***State the main objective/ problem that motivates this design***)…….. The proposed …… will be modeled, designed and simulated …… in Mentor Graphics environment……..(***state your design methodology and main technique used)***……... Finally, this …….. result in a power reduction of at least 2 times (<70µW) conventional ……..with at least a 3.0-5.0 V regulated output at 4kHz operating frequency, 90% efficiency and a power of about 650 µW …… (***State the result of this design***)…… This ……. Can be applied to ……due to its……... (***state the most suitable application with justification***).

5. REFERENCES

* USE GOOGLE SCHOLAR TO HELP YOU
* PROVIDE PROPER REFERENCES
  **JOURNAL/CONFERENCES/ TEXT BOOKS

APPENDIX *(OPTIONAL)*

ASSIGNMENT ASSESMENT CRITERIA

ROUGH EVALUATION CRITERIA

CORRECTNESS OF SOLUTION 60%
INTRODUCTION 5%
SCHEMATIC/ LAYOUT PRESENTATION 5%
METHODOLOGY (SCHEMATIC/ LAYOUT/ CODE OPTIMIZATION & IDEAS) 15%
RESULTS 10%
DISCUSSION 20%
CONCLUSION 5%

UNDERSTANDING 20%
CONTENTS 10%
UNDERSTANDING 10%

PRESENTATION OF REPORT 20%
EFFORT 5%
PRESENTATION 5%
SCHEMATIC/ LAYOUT/ CODING CREATIVITY 10%

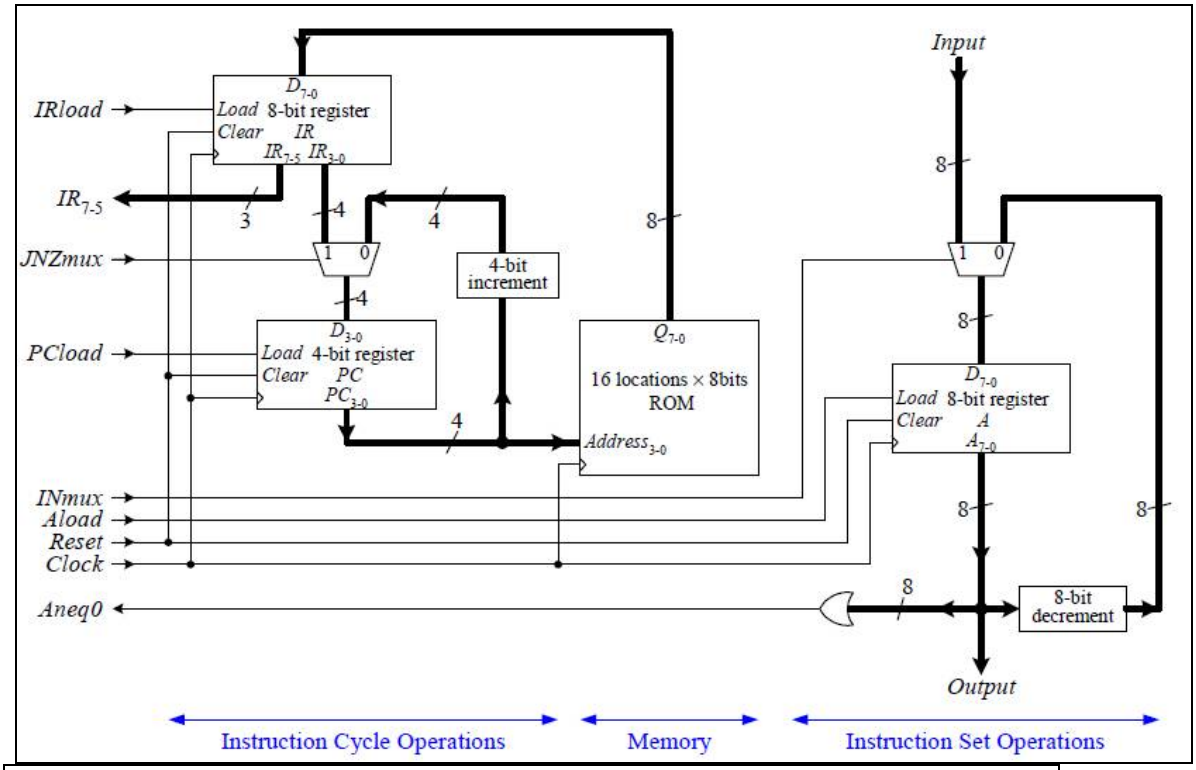
CONTENTS

LAB 1:

A SIMPLE PROCESSOR

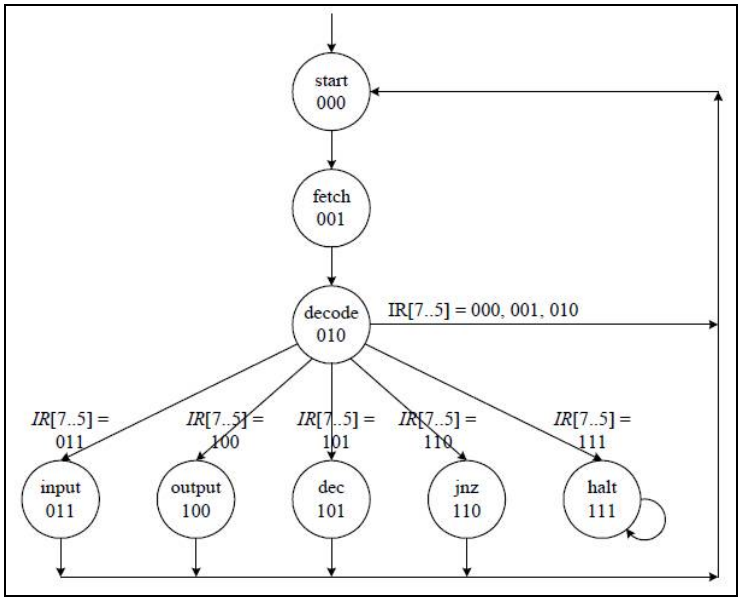
:: H/ W DEADLINE **WEEK 6**/ REPORT DEADLINE **WEEK 7**::
(Hardware 2-3 per group, REPORT 3-4 per group)

1. Use HDL Code to Synthesize the General Purpose Microprocessor (GPM) as shown in the Figure below (Datapath, Instruction Set, State Machine, and the Control Word Table has been given).
2. Write Design Code for :-
    a. Datapath (DP) of GPM
    b. Control Unit (CU) of GPM.
    c. Top Module of entire GPM (DP + CU).
3. Write Testbench code and verify waveforms in ModelSIM for 2(a-c) above.
4. Implement entire design in FPGA.
5. Implement the (Algorithm shown below) in FPGA/ Modelsim by manually compiling your assembly code/ instructions. Make sure your simulations are exactly like Figure 12.8.
6. Write a REPORT on your simple processor design (Refer to page 1 on Assignment Format).



Instruction	Encoding	Operation	Comment
IN A	011 ×××××	A ← input	Input to A
OUT A	100 ×××××	output ← A	Output from A
DEC A	101 ×××××	A ← A – 1	Decrement A
JNZ address	110 ×aaaa	if (A != 0) then PC = aaaa	Jump to address if A is not zero
HALT	111 ×××××	Halt	Halt execution

Notations:
A = accumulator.
PC = program counter.
aaaa = four bits for specifying a memory address.
× = don't cares.



Control Word	State $Q_2Q_1Q_0$	*IRload*	*PCload*	*INmux*	*Aload*	*JNZmux*	*Halt*
0	000 start	0	0	0	0	0	0
1	001 fetch	1	1	0	0	0	0
2	010 decode	0	0	0	0	0	0
3	011 input	0	0	1	1	0	0
4	100 output	0	0	0	0	0	0
5	101 dec	0	0	0	1	0	0
6	110 jnz	0	if (*Aneq0* = 1) then 1 else 0	0	0	1	0
7	111 halt	0	0	0	0	0	1

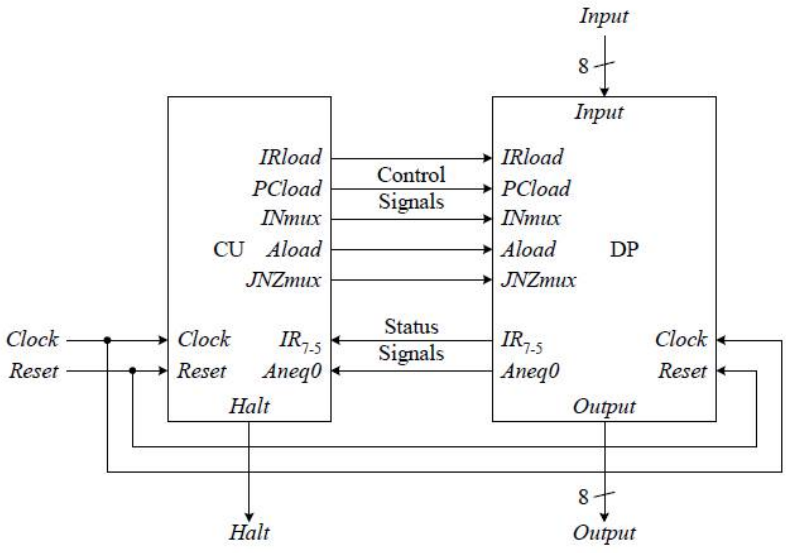


Figure 12.5. Complete circuit for the EC-1 general-purpose microprocessor.

```
        IN A          -- input a value into the A register
loop:   OUT A         -- output the value from the A register
        DEC A         -- decrement A by one
        JNZ loop      -- go back to loop if A is not zero
        HALT          -- halt
```

memory address	instruction encoding	
0000	01100000;	-- IN A
0001	10000000;	-- OUT A
0010	10100000;	-- DEC A
0011	11000001;	-- JNZ loop
0100	11111111;	-- HALT

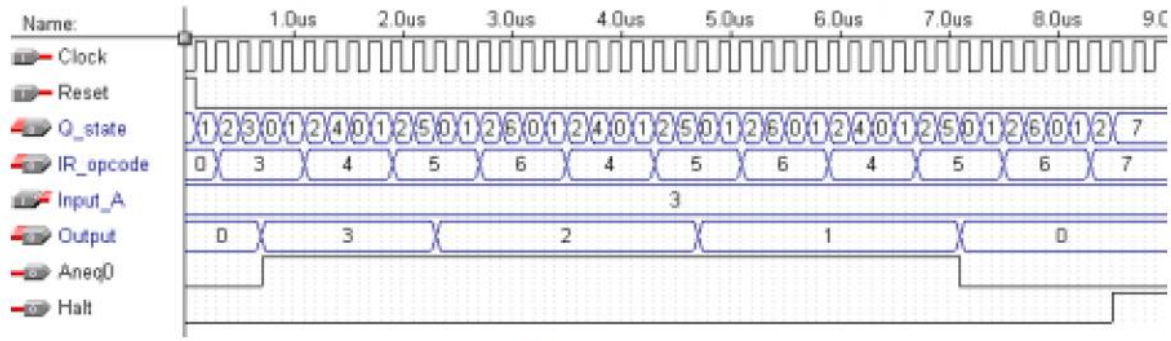


Figure 12.8. A sample simulation trace of the countdown program running on the EC-1 starting at the input 3.

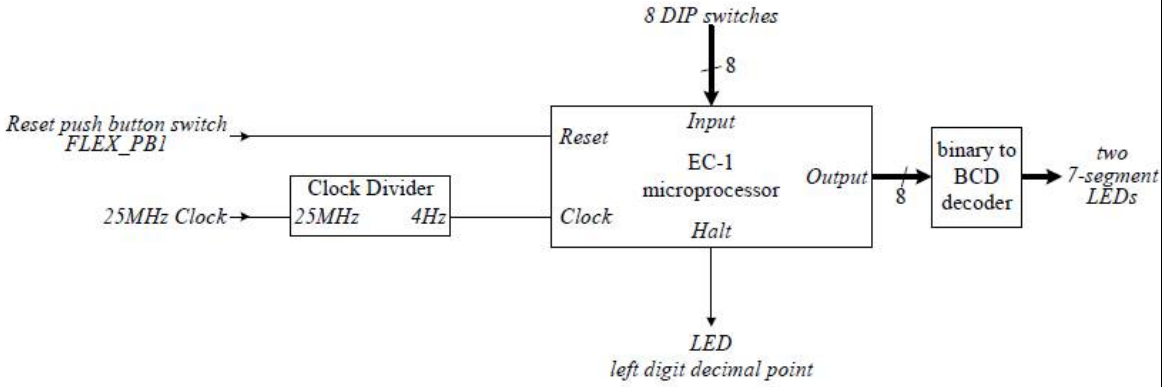


Figure 12.9. Hardware implementation of the EC-1 computer.

LAB 1B:

A Simple Processor

Figure 1 shows a digital system that contains a number of 9-bit registers, a multiplexer, an adder/subtracter unit, and a control unit(finite state machine). Data is input to this system via the 9-bit *DIN* input. This data can be loaded through the 9-bit wide multiplexer into the various registers, such as $R0, \ldots, R7$ and A. The multiplexer also allows data to be transferred from one register to another. The multiplexer's output wires are called a *bus* in the figure because this term is often used for wiring that allows data to be transferred from one location in a system to another.

Addition or subtraction is performed by using the multiplexer to first place one 9-bit number onto the bus wires and loading this number into register A. Once this is done, a second 9-bit number is placed onto the bus, the adder/subtracter unit performs the required operation, and the result is loaded into register G. The data in G can then be transferred to one of the other registers as required.

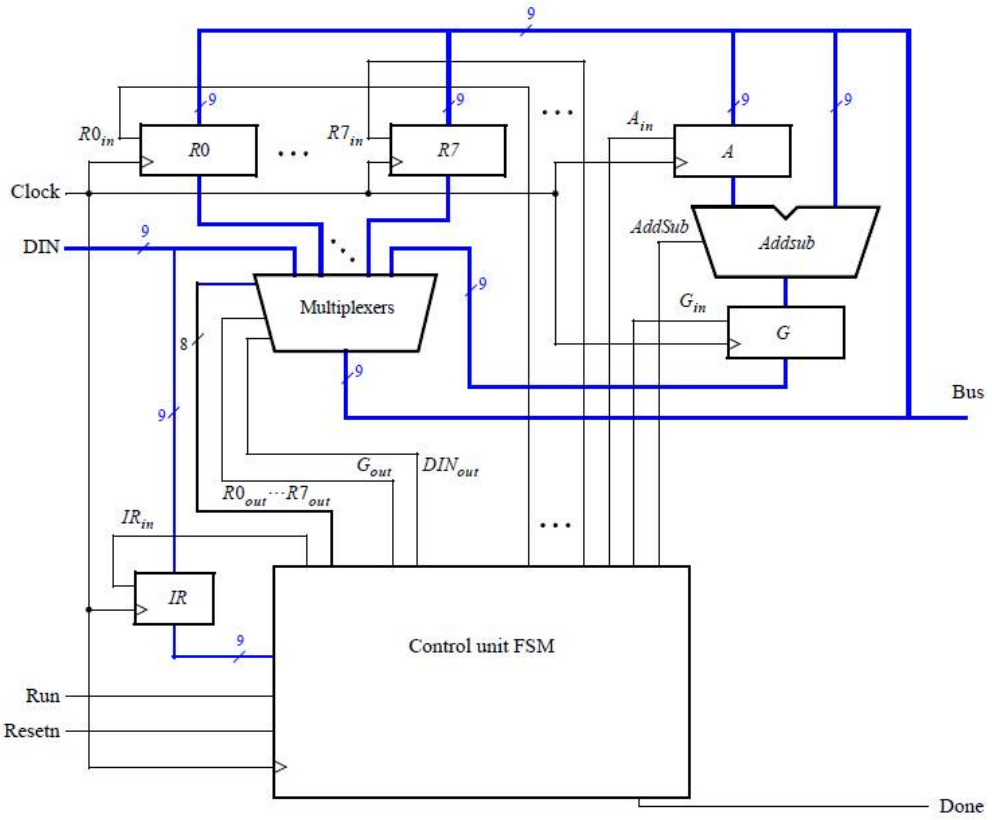


Figure 1: A digital system.

The system can perform different operations in each clock cycle, as governed by the *control unit*. This unit determines when particular data is placed onto the bus wires and it controls which of the registers is to be loaded with this data. For example, if the control unit asserts the signals $R0_{out}$ and A_{in}, then the multiplexer will place the contents of register $R0$ onto the bus and this data will be loaded by the next active clock edge into register A.

A system like this is often called a *processor*. It executes operations specified in the form of instructions.

Table 1 lists the instructions that the processor has to support for this exercise. The left column shows the name of an instruction and its operand. The meaning of the syntax RX ← [RY] is that the contents of register RY are loaded into register RX. The **mv** (move) instruction allows data to be copied from one register to another. For the **mvi** (move immediate) instruction the expression RX ← D indicates that the 9-bit constant D is loaded into register RX.

Operation	Function performed
mv Rx,Ry	$Rx \leftarrow [Ry]$
mvi Rx,#D	$Rx \leftarrow D$
add Rx, Ry	$Rx \leftarrow [Rx] + [Ry]$
sub Rx, Ry	$Rx \leftarrow [Rx] - [Ry]$

Table 1. Instructions performed in the processor.

Each instruction can be encoded and stored in the *IR* register using the 9-bit format IIIXXXYYY, where III represents the instruction, XXX gives the RX register, and YYY gives the RY register. Although only two bits are needed to encode our four instructions, we are using three bits because other instructions will be added to the processor in later parts of this exercise. Hence *IR* has to be connected to the nine bits of the *DIN* input, as indicated in Figure 1. For the **mvi** instruction the YYY field has no meaning, and the immediate data #D has to be supplied on the 9-bit *DIN* input after the **mvi** instruction word is stored into *IR*.

Some instructions, such as an addition or subtraction, take more than one clock cycle to complete, because multiple transfers have to be performed across the bus. The finite state machine in the control unit "steps through" such instructions, asserting the control signals needed in successive clock cycles until the instruction has completed. The processor starts executing the instruction on the *DIN* input when the *Run* signal is asserted and the processor asserts the *Done* output when the instruction is finished. Table 2 indicates the control signals that can be asserted in each time step to implement the instructions in Table 1. Note that the only control signal asserted in time step 0 is IR_{in}, so this time step is not shown in the table.

	T_1	T_2	T_3
(**mv**): I_0	RY_{out}, RX_{in}, *Done*		
(**mvi**): I_1	DIN_{out}, RX_{in}, *Done*		
(**add**): I_2	RX_{out}, A_{in}	RY_{out}, G_{in}	G_{out}, RX_{in}, *Done*
(**sub**): I_3	RX_{out}, A_{in}	RY_{out}, G_{in}, *AddSub*	G_{out}, RX_{in}, *Done*

Table 2. Control signals asserted in each instruction/time step.

Part I

Design and implement the processor shown in Figure 1 using Verilog code as follows:

1. Create a new Quartus II project for this exercise.
2. Generate the required Verilog file, include it in your project, and compile the circuit. A suggested skeletc of the Verilog code is shown in parts a and b of Figure 2, and some subcircuit modules that can be used i this code appear in Figure 2c.
3. Use functional simulation to verify that your code is correct. An example of the output produced by functional simulation for a correctly-designed circuit is given in Figure 3. It shows the value $(040)_{16}$ bein loaded into *IR* from *DIN* at time 30 ns. This pattern (the leftmost bits of *DIN* are connected to *IR*) represen the instruction **mvi** R0,#D, where the value $D = 5$ is loaded into $R0$ on the clock edge at 50 ns. Th simulation then shows the instruction **mv** R1,R0 at 90 ns, **add** R0,R1 at 110 ns, and **sub** R0,R0 at 190 n Note that the simulation output shows *DIN* as a 3-digit hexadecimal number, and it shows the contents c *IR* as a 3-digit octal number.
4. Create a new Quartus II project which will be used for implementation of the circuit on the Altera DE board. This project should consist of a top-level module that contains the appropriate input and output por for the Altera board. Instantiate your processor in this top-level module. Use switches SW_{8-0} to drive th *DIN* input port of the processor and use switch SW_9 to drive the *Run* input. Also, use push button KEY_0 fc *Resetn* and KEY_1 for *Clock*. Connect the processor bus wires to $LEDR_{8-0}$ and connect the *Done* signal t $LEDR_9$.
5. Add to your project the necessary pin assignments for the DE1 board. Compile the circuit and download into the FPGA chip.
6. Test the functionality of your design by toggling the switches and observing the LEDs. Since the processor clock input is controlled by a push button switch, it is easy to step through the execution of instructions an observe the behavior of the circuit.

```
module  proc (DIN, Resetn, Clock, Run, Done, BusWires);
    input [8:0] DIN;
    input Resetn, Clock, Run;
    output Done;
    output [8:0] BusWires;

    parameter T0 = 2'b00, T1 = 2'b01, T2 = 2'b10, T3 = 2'b11;
    . . . declare variables

    assign I = IR[1:3];
    dec3to8 decX (IR[4:6], 1'b1, Xreg);
    dec3to8 decY (IR[7:9], 1'b1, Yreg);
```

Figure 2a. Skeleton Verilog code for the processor.

```
// Control FSM state table
always @(Tstep_Q, Run, Done)
begin
    case (Tstep_Q)
        T0: // data is loaded into IR in this time step
            if (!Run) Tstep_D = T0;
            else Tstep_D = T1;
        T1: ...
    endcase
end

// Control FSM outputs
always @(Tstep_Q or I or Xreg or Yreg)
begin
    ... specify initial values
    case (Tstep_Q)
        T0: // store DIN in IR in time step 0
        begin
            IRin = 1'b1;
        end
        T1: //define signals in time step 1
            case (I)
                ...
            endcase
        T2: //define signals in time step 2
            case (I)
                ...
            endcase
        T3: //define signals in time step 3
            case (I)
                ...
            endcase
    endcase
end

// Control FSM flip-flops
always @(posedge Clock, negedge Resetn)
    if (!Resetn)
    ...

regn reg_0 (BusWires, Rin[0], Clock, R0);
... instantiate other registers and the adder/subtracter unit

... define the bus

endmodule
```

Figure 2*b*. Skeleton Verilog code for the processor.

```
module  dec3to8(W, En, Y);
    input [2:0] W;
    input En;
    output [0:7] Y;
    reg [0:7] Y;

    always @(W or En)
    begin
        if (En == 1)
            case (W)
                3'b000: Y = 8'b10000000;
                3'b001: Y = 8'b01000000;
                3'b010: Y = 8'b00100000;
                3'b011: Y = 8'b00010000;
                3'b100: Y = 8'b00001000;
                3'b101: Y = 8'b00000100;
                3'b110: Y = 8'b00000010;
                3'b111: Y = 8'b00000001;
            endcase
        else
            Y = 8'b00000000;
    end
endmodule

module  regn(R, Rin, Clock, Q);
    parameter n = 9;
    input [n-1:0] R;
    input Rin, Clock;
    output [n-1:0] Q;
    reg [n-1:0] Q;

    always @(posedge Clock)
        if (Rin)
            Q <= R;
endmodule
```

Figure 2*c*. Subcircuit modules for use in the processor.

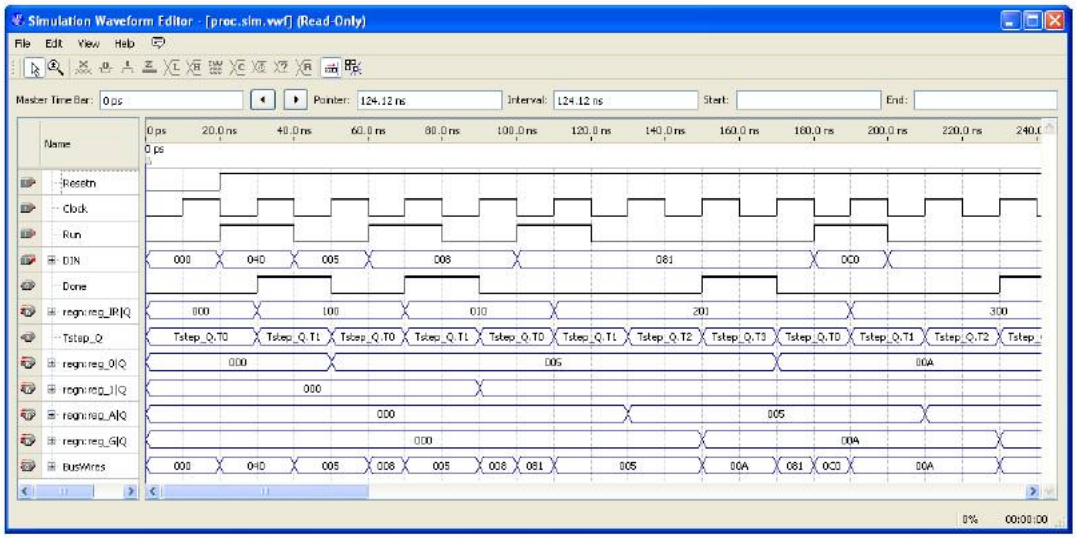


Figure 3. Simulation of the processor.

Part II

In this part you are to design the circuit depicted in Figure 4, in which a memory module and counter are connected to the processor from Part I. The counter is used to read the contents of successive addresses in the memory, and this data is provided to the processor as a stream of instructions. To simplify the design and testing of this circuit we have used separate clock signals, *PClock* and *MClock*, for the processor and memory.

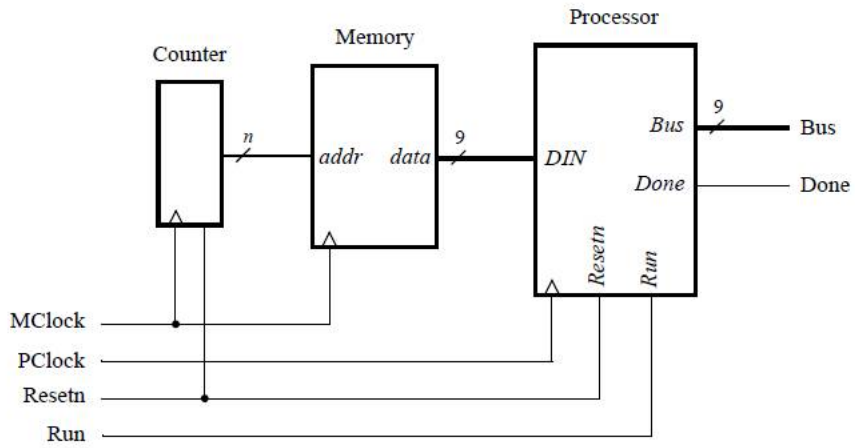


Figure 4. Connecting the processor to a memory and counter.

1. Create a new Quartus II project which will be used to test your circuit.

2. Generate a top-level Verilog file that instantiates the processor, memory, and counter. Use the Quartus II MegaWizard Plug-In Manager tool to create the memory module from the Altera library of parameterized modules (LPMs). The correct LPM is found under the *Memory Compiler* category and is called *ROM: 1-PORT*. Follow the instructions provided by the wizard to create a memory that has one 9-bit wide read data port and is 32 words deep. Page 3 of the wizard is shown in Figure 5. Advance to the subsequent page and *deselect* the setting called 'q' output port under the category Which ports should be registered?. Since this memory has only a read port, and no write port, it is called a *synchronous read-only memory*

(synchronous ROM). Note that the memory includes a register for synchronously loading addresses. This register is required due to the design of the memory resources on the Cyclone II FPGA; account for the clocking of this address register in your design.

To place processor instructions into the memory, you need to specify *initial values* that should be stored in the memory once your circuit has been programmed into the FPGA chip. This can be done by telling the wizard to initialize the memory using the contents of a *memory initialization file (MIF)*. The appropriate screen of the MegaWizard Plug-In Manager tool is illustrated in Figure 6. We have specified a file named *inst_mem.mif*, which then has to be created in the directory that contains the Quartus II project. Use the Quartus II on-line Help to learn about the format of the *MIF* file and create a file that has enough processor instructions to test your circuit.

3. Use functional simulation to test the circuit. Ensure that data is read properly out of the ROM and executed by the processor.

4. Make sure your project includes the necessary port names and pin location assignments to implement the circuit on the DE1 board. Use switch SW_9 to drive the processor's *Run* input, use KEY_0 for *Resetn*, use KEY_1 for *MClock*, and use KEY_2 for *PClock*. Connect the processor bus wires to $LEDR_{8-0}$ and connect the *Done* signal to $LEDR_9$.

5. Compile the circuit and download it into the FPGA chip.

6. Test the functionality of your design by toggling the switches and observing the LEDs. Since the circuit's clock inputs are controlled by push button switches, it is easy to step through the execution of instructions and observe the behavior of the circuit.

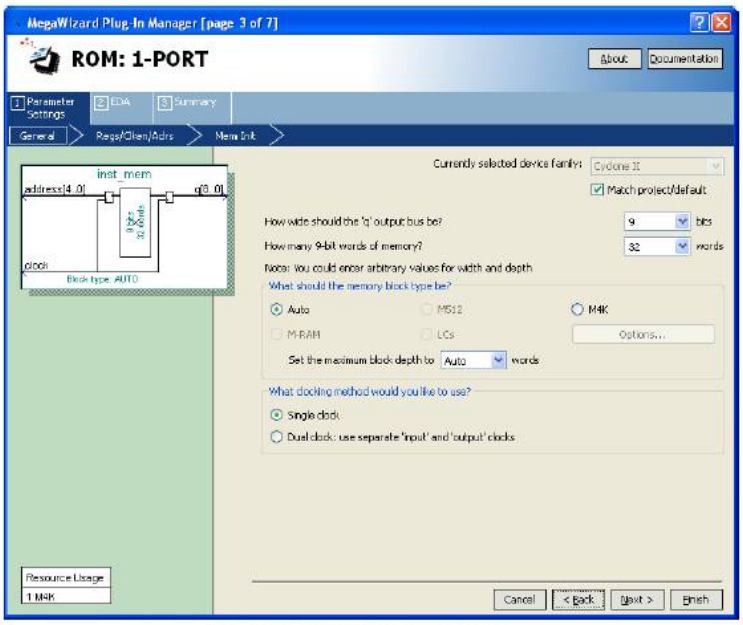


Figure 5. 1-PORT configuration.

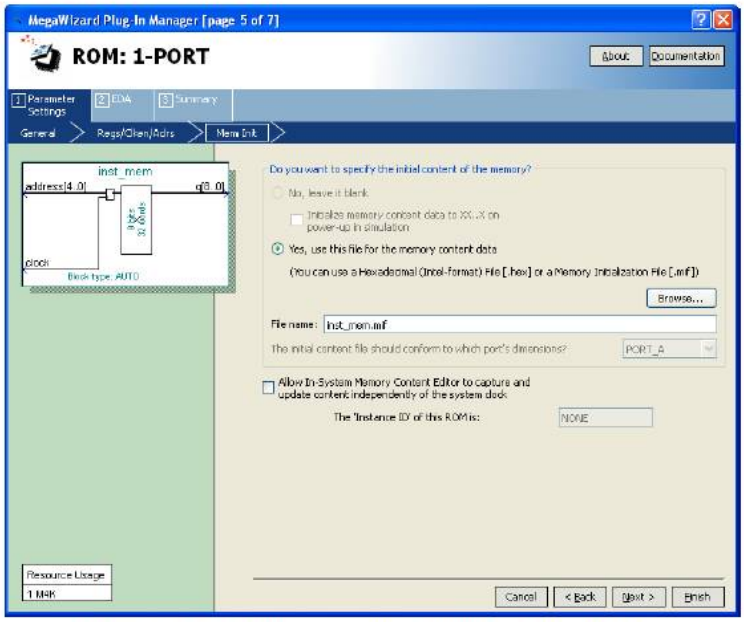


Figure 6. Specifying a memory initialization file (MIF).

Enhanced Processor

It is possible to enhance the capability of the processor so that the counter in Figure 4 is no longer needed, and so that the processor has the ability to perform read and write operations using memory or other devices. These enhancements involve adding new instructions to the processor and the programs that the processor executes are therefore more complex; they are described in a subsequent lab exercise available from Altera.

========== EXTRACTED FROM ALTERA CORP, 2011==========

LAB 2:

AN ENHANCED PROCESSOR

:: H/ W DEADLINE **WEEK 6**/ REPORT DEADLINE **WEEK 7**::
(Hardware 2-3 per group, REPORT 3-4 per group)

1. Use HDL Code to Synthesize the General Purpose Microprocessor (GPM) as shown in the Figure below (Datapath, Instruction Set, State Machine, and the Control Word Table has been given).
2. Write Design Code for :-
   a. Datapath (DP) of GPM
   b. Control Unit (CU) of GPM.
   c. Top Module of entire GPM (DP + CU).
3. Write Testbench code and verify waveforms in ModelSIM for 2(a-c) above.
4. Implement entire design in FPGA.
5. Implement the (Algorithm shown below) in FPGA/ Modelsim by manually compiling your assembly code/ instructions.
6. Write a REPORT on your simple processor design (Refer to page 1 on Assignment Format).

Instruction	Encoding	Operation	Comment
LOAD A, address	000 aaaaa	A ← M[aaaaa]	Load A with content of memory location aaaaa
STORE A, address	001 aaaaa	M[aaaaa] ← A	Store A into memory location aaaaa
ADD A, address	010 aaaaa	A ← A + M[aaaaa]	Add A with M[aaaaa] and store the result back into A
SUB A, address	011 aaaaa	A ← A – M[aaaaa]	Subtract A with M[aaaaa] and store result back into A
IN A	100 ×××××	A ← input	Input to A
JZ address	101 aaaaa	if (A = 0) then PC = aaaaa	Jump to address if A is zero
JPOS address	110 aaaaa	if (A > 0) then PC = aaaaa	Jump to address if A is positive
HALT	111 ×××××	Halt	Halt execution

Notations:
A = accumulator.
M = memory.
PC = program counter.
aaaaa = four bits for specifying a memory address.
× = don't cares.

Figure 12.10. Instruction set for the EC-2.

State $Q_3Q_2Q_1Q_0$	*IRload*	*JMPmux*	*PCload*	*Meminst*	*MemWr*	$Asel_{1,0}$	*Aload*	*Sub*	*Halt*
0000 start	0	0	0	0	0	00	0	0	0
0001 fetch	1	0	1	0	0	00	0	0	0
0010 decode	0	0	0	1	0	00	0	0	0
1000 load	0	0	0	0	0	10	1	0	0
1001 store	0	0	0	1	1	00	0	0	0
1010 add	0	0	0	0	0	00	1	0	0
1011 sub	0	0	0	0	0	00	1	1	0
1100 input	0	0	0	0	0	01	1	0	0
1101 jz	0	1	*Aeq0*	0	0	00	0	0	0
1110 jpos	0	1	*Apos*	0	0	00	0	0	0
1111 halt	0	0	0	0	0	00	0	0	1

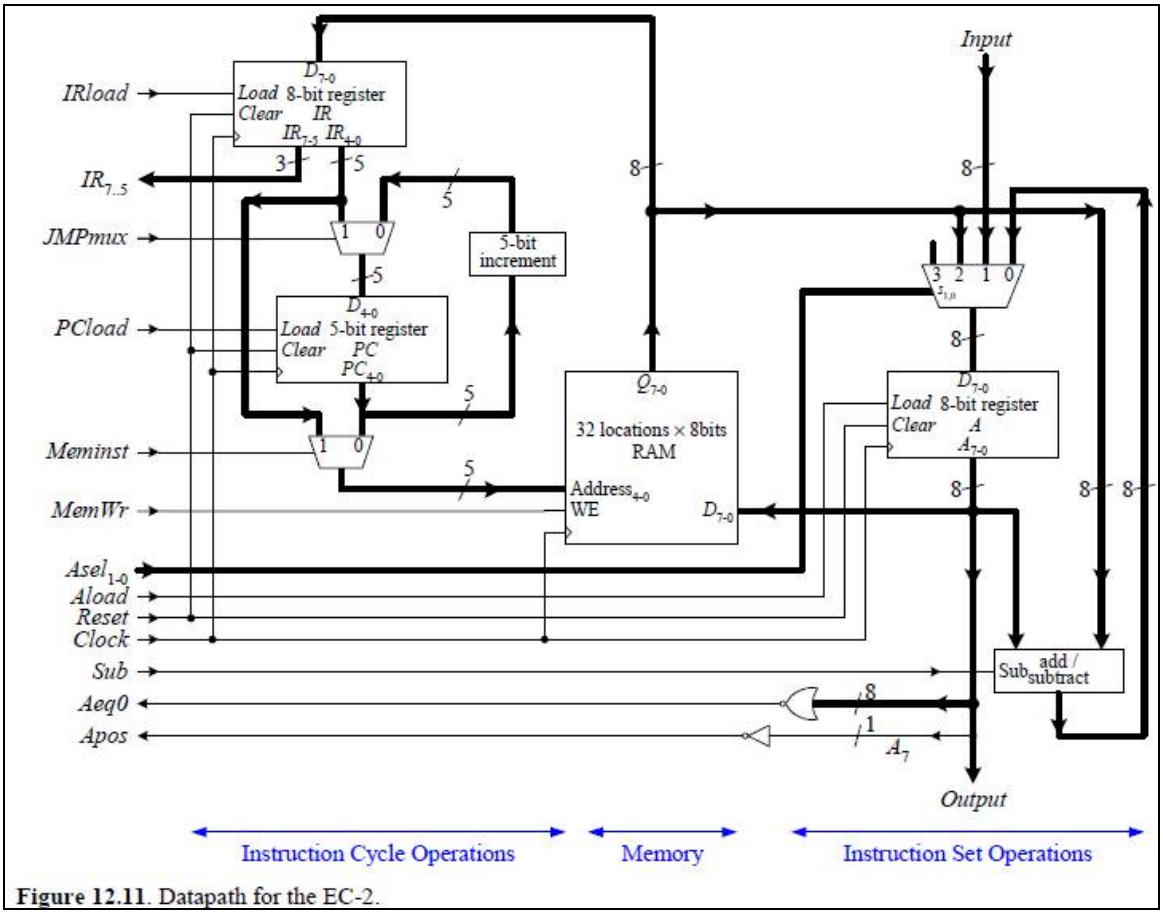
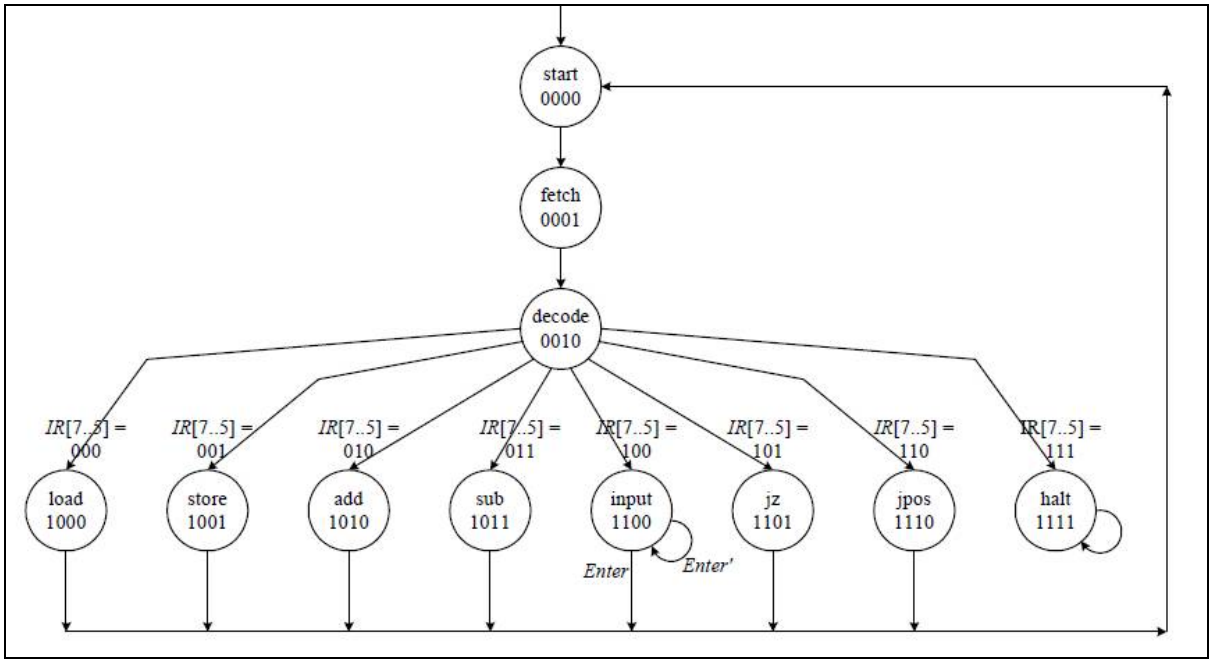


Figure 12.11. Datapath for the EC-2.



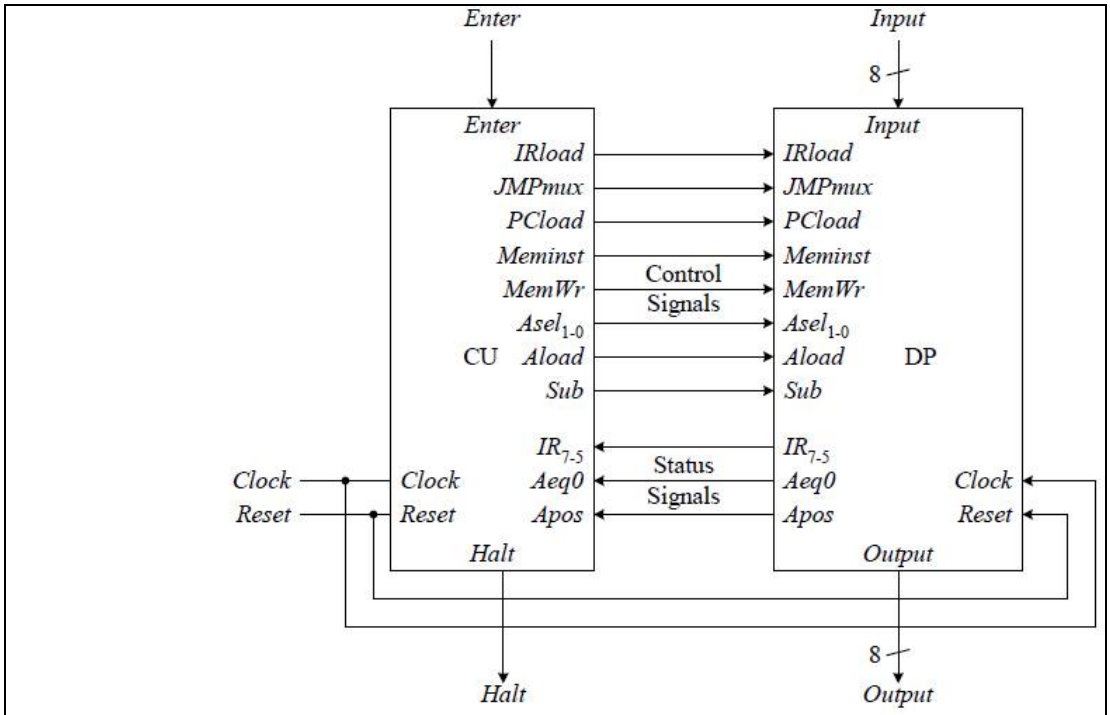


Figure 12.13. Complete circuit for the EC-2 general-purpose microprocessor.

```
-----------------------------------------------------------
-- COUNT
-- Program to countdown from input N to 0

  00000 : 10000000;  -- input
  00001 : 01111111;  -- Sub A,11111
  00010 : 10100100;  -- jz 00100
  00011 : 11000001;  -- jp 00001
  00100 : 11111111;  -- halt
  11111 : 00000001;  -- constant 1

END;
```

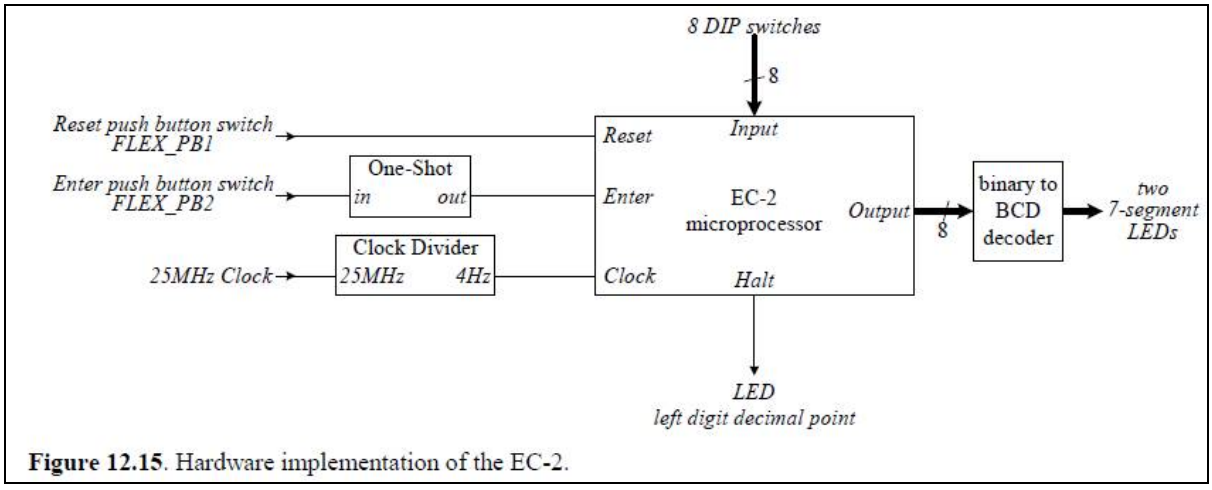


Figure 12.15. Hardware implementation of the EC-2.

LAB 2B:

An Enhanced Processor

In Laboratory Exercise 9 we described a simple processor. In Part I of that exercise the processor itself was designed, and in Part II the processor was connected to an external counter and a memory unit. This exercise describes subsequent parts of the processor design. Note that the numbering of figures and tables in this exercise are continued from those in Parts I and II in the preceding lab exercise.

Part III

In this part you will extend the capability of the processor so that the external counter is no longer needed, and so that the processor has the ability to perform read and write operations using memory or other devices. You will add three new types of instructions to the processor, as displayed in Table 3. The **ld** (load) instruction loads data into register RX from the external memory address specified in register RY. The **st** (store) instruction stores the data contained in register RX into the memory address found in RY. Finally, the instruction **mvnz** (move if not zero) allows a **mv** operation to be executed only under a certain condition; the condition is that the current contents of register G are not equal to 0.

Operation	Function performed
ld Rx,[Ry]	$Rx \leftarrow [[Ry]]$
st Rx,[Ry]	$[Ry] \leftarrow [Rx]$
mvnz Rx, Ry	if G != 0, $Rx \leftarrow [Ry]$

Table 3. New instructions performed in the processor.

A schematic of the enhanced processor is given in Figure 7. In this figure, registers $R0$ to $R6$ are the same as in Figure 1 of Laboratory Exercise 9, but register $R7$ has been changed to a counter. This counter is used to provide the addresses in the memory from which the processor's instructions are read; in the preceding lab exercise, a counter external to the processor was used for this purpose. We will refer to $R7$ as the processor's *program counter* (*PC*), because this terminology is common for real processors available in the industry. When the processor is reset, *PC* is set to address 0. At the start of each instruction (in time step 0) the contents of *PC* are used as an address to read an instruction from the memory. The instruction is stored in IR and the *PC* is automatically incremented to point to the next instruction (in the case of **mvi** the *PC* provides the address of the immediate data and is then incremented again).

The processor's control unit increments *PC* by using the *incr_PC* signal, which is just an enable on this counter. It is also possible to directly load an address into *PC* ($R7$) by having the processor execute a **mv** or **mvi** instruction in which the destination register is specified as $R7$. In this case the control unit uses the signal $R7_{in}$ to perform a parallel load of the counter. In this way, the processor can execute instructions at any address in memory, as opposed to only being able to execute instructions that are stored in successive addresses. Similarly, the current contents of *PC* can be copied into another register by using a **mv** instruction. An example of code that uses the *PC* register to implement a loop is shown below, where the text after the % on each line is just a comment. The instruction **mv** R5,R7 places into *R5* the address in memory of the instruction **sub** R4,R2. Then, the instruction **mvnz** R7,R5 causes the **sub** instruction to be executed repeatedly until R4 becomes 0. This type of loop could be used in a larger program as a way of creating a delay.

```
mvi   R2,#1
mvi   R4,#10000000   % binary delay value
mv    R5,R7          % save address of next instruction
sub   R4,R2          % decrement delay count
mvnz  R7,R5          % continue subtracting until delay count gets to 0
```

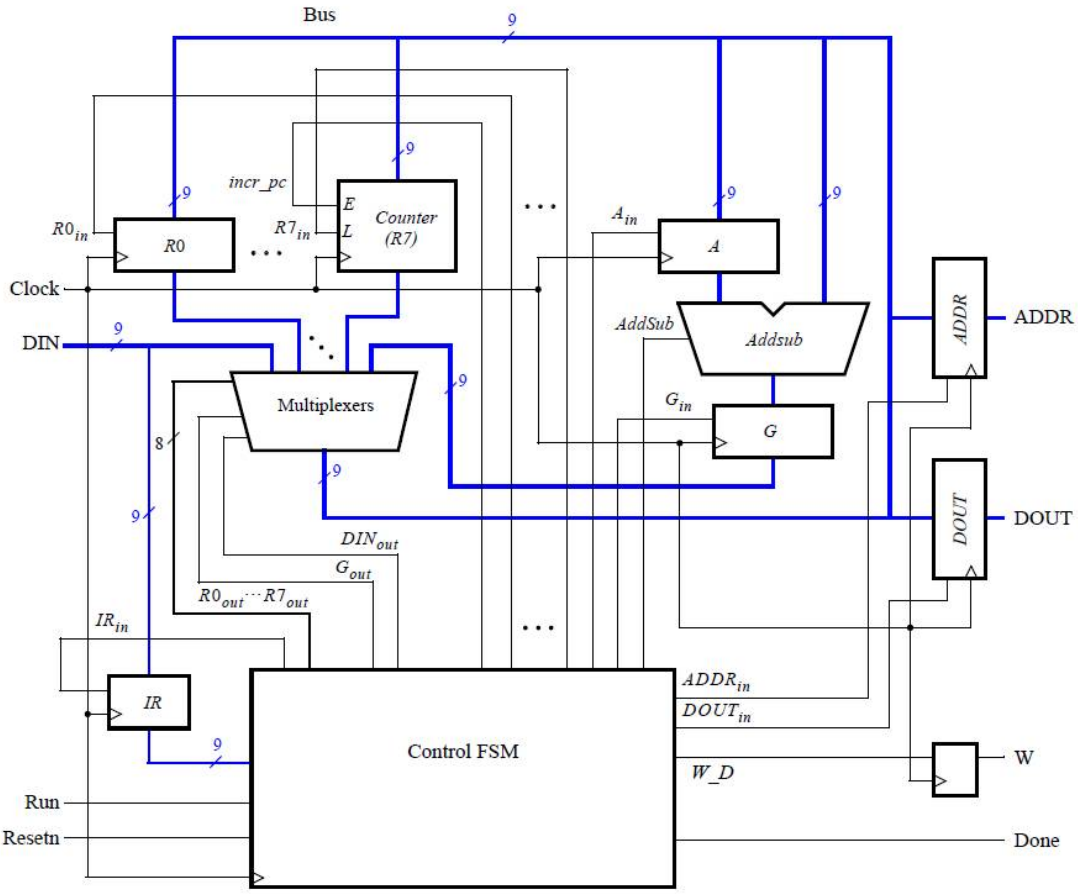


Figure 7. An enhanced version of the processor.

Figure 7 shows two registers in the processor that are used for data transfers. The *ADDR* register is used to send addresses to an external device, such as a memory module, and the *DOUT* register is used by the processor to provide data that can be stored outside the processor. One use of the *ADDR* register is for reading, or *fetching*, instructions from memory; when the processor wants to fetch an instruction, the contents of *PC* ($R7$) are transferred across the bus and loaded into *ADDR*. This address is provided to memory. In addition to fetching instructions, the processor can read data at any address by using the *ADDR* register. Both data and instructions are read into the processor on the *DIN* input port. The processor can write data for storage at an external address by placing this address into the *ADDR* register, placing the data to be stored into its *DOUT* register, and asserting the output of the *W* (write) flip-flop to 1.

Figure 8 illustrates how the enhanced processor is connected to memory and other devices. The memory unit in the figure supports both read and write operations and therefore has both address and data inputs, as well as a write enable input. The memory also has a clock input, because the address, data, and write enable inputs must be loaded into the memory on an active clock edge. This type of memory unit is usually called a *synchronous random access memory (synchronous RAM)*. Figure 8 also includes a 9-bit register that can be used to store data from the processor; this register might be connected to a set of LEDs to allow display of data on the DE1 board. To allow the processor to select either the memory unit or register when performing a write operation, the circuit includes some logic gates that perform *address decoding*: if the upper address lines are $A_8A_7 = 00$, then the memory module will be written at the address given on the lower address lines. Figure 8 shows n lower address lines connected to the memory; for this exercise a memory with 128 words is probably sufficient, which implies that $n = 7$ and the memory address port is driven by $A_6 \ldots A_0$. For addresses in which $A_8A_7 = 01$, the data

written by the processor is loaded into the register whose outputs are called *LEDs* in Figure 8.

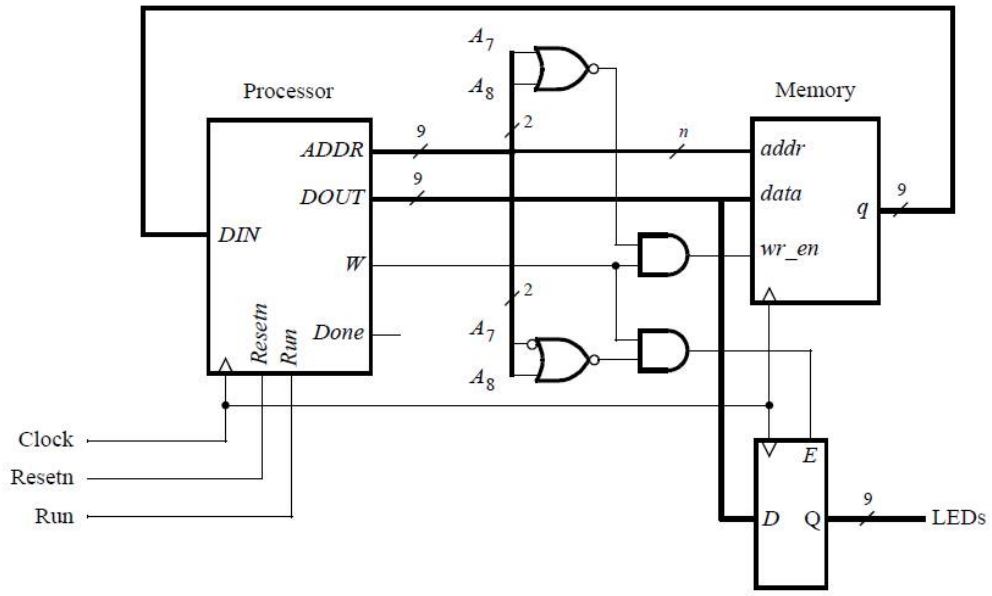


Figure 8. Connecting the enhanced processor to a memory and output register.

1. Create a new Quartus II project for the enhanced version of the processor.

2. Write Verilog code for the processor and test your circuit by using functional simulation: apply instructions to the *DIN* port and observe the internal processor signals as the instructions are executed. Pay careful attention to the timing of signals between your processor and external memory; account for the fact that the memory has registered input ports, as we discussed for Figure 8.

3. Create another Quartus II project that instantiates the processor, memory module, and register shown in Figure 8. Use the Quartus II MegaWizard Plug-In Manager tool to create the RAM: 1-PORT memory module. Follow the instructions provided by the wizard to create a memory that has one 9-bit wide read/write data port and is 128 words deep. Use a MIF file to store instructions in the memory that are to be executed by your processor.

4. Use functional simulation to test the circuit. Ensure that data is read properly from the RAM and executed by the processor.

5. Include in your project the necessary pin assignments to implement your circuit on the DE1 board. Use switch SW_9 to drive the processor's *Run* input, use KEY_0 for *Resetn*, and use the board's 50 MHz clock signal as the *Clock* input. Since the circuit needs to run properly at 50 MHz, make sure that a timing constraint is set in Quartus II to constrain the circuit's clock to this frequency. Read the Report produced by the Quartus II Timing Analyzer to ensure that your circuit operates at this speed; if not, use the Quartus II tools to analyze your circuit and modify your Verilog code to make a more efficient design that meets the 50-MHz speed requirement. Also note that the *Run* input is asynchronous to the clock signal, so make sure to synchronize this input using flip-flops.

   Connect the *LEDs* register in Figure 8 to $LEDR_{8-0}$ so that you can observe the output produced by the processor.

6. Compile the circuit and download it into the FPGA chip.

7. Test the functionality of your design by executing code from the RAM and observing the LEDs.

Part IV

In this part you are to connect an additional I/O module to your circuit from Part III and write code that is executed by your processor.

Add a module called *seg7_scroll* to your circuit. This module should contain one register for each 7-segment display on the DE1 board. Each register should directly drive the segment lights for one 7-segment display, so that the processor can write characters onto these displays. Create the necessary address decoding to allow the processor to write to the registers in the *seg7_scroll* module.

1. Create a Quartus II project for your circuit and write the Verilog code that includes the circuit from Figure 8 in addition to your *seg7_scroll* module.
2. Use functional simulation to test the circuit.
3. Add appropriate timing constraints and pin assignments to your project, and write a MIF file that allows the processor to write characters to the 7-segment displays. A simple program would write a word to the displays and then terminate, but a more interesting program could scroll a message across the displays, or scroll a word across the displays in the left, right, or both directions.
4. Test the functionality of your design by executing code from the RAM and observing the 7-segment displays.

Part V

Add to your circuit from Part IV another module, called *port_n*, that allows the processor to read the state of some switches on the board. The switch values should be stored into a register, and the processor should be able to read this register by using a **ld** instruction. You will have to use address decoding and multiplexers to allow the processor to read from either the RAM or *port_n* units, according to the address used.

1. Draw a circuit diagram that shows how the *port_n* unit is incorporated into the system.
2. Create a Quartus II project for your circuit, write the Verilog code, and write a MIF file that demonstrates use of the *port_n* module. One interesting application is to have the processor scroll a message across the 7-segment displays and use the values read from the *port_n* module to change the speed at which the message is scrolled.
3. Test your circuit both by using functional simulation and by downloading it and executing your processor code on the DE1 board.

Suggested Bonus Parts

The following are suggested bonus parts for this exercise.

1. Use the Quartus II tools to identify the critical paths in the processor circuit. Modify the processor design so that the circuit will operate at the highest clock frequency that you can achieve.
2. Extend the instructions supported by your processor to make it more flexible. Some suggested instruction types are logic instructions (AND, OR, etc), shift instructions, and branch instructions. You may also wish to add support for logical conditions other than "not zero" , as supported by **mvnz**, and the like.
3. Write an Assembler program for your processor. It should automatically produces a MIF file from assembler code.

LAB 3:
IMPLEMENTING ALGORITHMS IN H/W

1. Implement Basic Algorithm as shown in Figure 3(a) into the simple processor in LAB 1.
2. Implement the Algorithms in Figure 3(b) into the enhanced processor in LAB 2.
3. Use HDL based testbench code to implement your manually compiled algorithm in 1 and 2.

```
        IN A        -- input a value into the A register
loop:   OUT A       -- output the value from the A register
        DEC A       -- decrement A by one
        JNZ loop    -- go back to loop if A is not zero
        HALT        -- halt
```

```
memory      instruction
address     encoding
0000        01100000;   -- IN A
0001        10000000;   -- OUT A
0010        10100000;   -- DEC A
0011        11000001;   -- JNZ loop
0100        11111111;   -- HALT
```

Figure 3 (a)

```
-- GCD
-- Program to calculate the GCD of two numbers

  00000 : 10000000;  -- input A
  00001 : 00111110;  -- store A,x
  00010 : 10000000;  -- input A
  00011 : 00111111;  -- store A,y

  00100 : 00011110;  -- loop: load A,x    -- x = y?
  00101 : 01111111;  -- sub A,y
  00110 : 10110000;  -- jz out            -- x=y
  00111 : 11001100;  -- jp xgty           -- x>y

  01000 : 00011111;  -- load A,y          -- y>x
  01001 : 01111110;  -- sub A,x           -- y-x
  01010 : 00111111;  -- store A,y
  01011 : 11000100;  -- jp loop

  01100 : 00011110;  -- xgty: load A,x    -- x>y
  01101 : 01111111;  -- sub A,y           -- x-y
  01110 : 00111110;  -- store A,x
  01111 : 11000100;  -- jp loop

  10000 : 00011110;  -- load A,x
  10001 : 11111111;  -- halt

  11110 : 00000000;  -- x
  11111 : 00000000;  -- y
```

```
1     input X
2     input Y
3     while (X ≠ Y){
4        if (X > Y) then
5           X = X - Y
6        else
7           Y = Y - X
8        end if
9        }
10    output X
```

Figure 3 (b)

LAB 3B:

Implementing Algorithms in Hardware

This is an exercise in using algorithmic state machine charts to implement algorithms as hardware circuits.

Background

Algorithmic State Machine (ASM) charts are an alternative representation for finite state machines, which allow designers to express larger state machines and circuits in a manner similar to a flow chart. An example of an ASM chart is shown in Figure 1. It represents a circuit that counts the number of bits set to 1 in an n-bit input A ($A = a_{n-1}a_{n-2}..a_1a_0$).

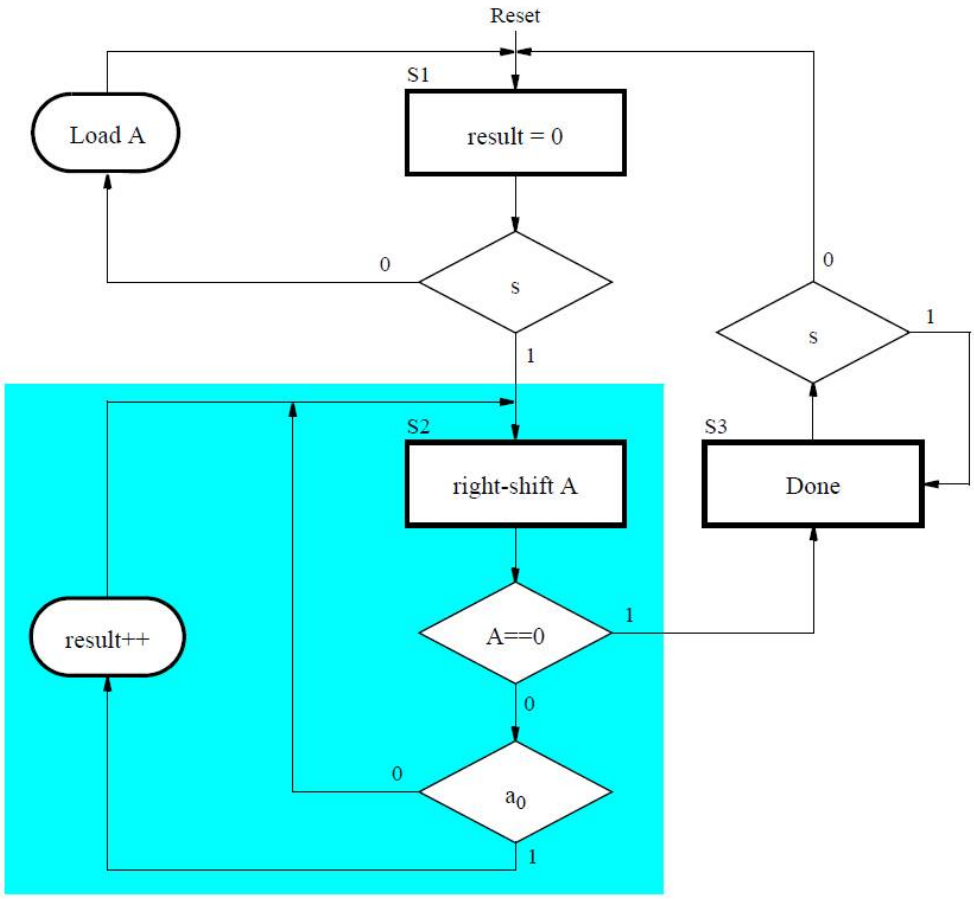


Figure 1: ASM chart for a bit counting circuit.

In this ASM chart, state S1 is the initial state where we load input into shirt register *A* and wait for the start (*s*) signal to begin operation. We then counter the number of 1's in *A* in state S2, and wait in state S3 when counting is completed.

The key distinction between ASM and flow charts is in what is known as *implied timing*. In contrast to a flow

chart, events that stem from a single state (rectangle) box in an ASM chart are considered to happen in the same clock cycle. Any synchronous elements, such as counters or registers, update their value when the next state is reached. Thus, the correct way to interpret the highlighted state in Figure 1 is as follows.

In state S2, the shift register A is enabled to shift contents at the next positive edge of the clock. Simultaneously, its present value is tested to check if it is equal to 0. If A is not 0, then we check if the least-significant bit of A (a_0) is 1. If it is, then the counter named *result* will be incremented at the next positive edge of the clock. If A is 0, then we proceed to state S3.

The implementation of the bit counting circuit consists of components controlled by an FSM that functions according to the ASM chart - we refer to these components as the *datapath*. The datapath components include a counter to store *result* and a shift register A.

In this exercise you will design and implement several circuits using ASM charts.

Part I

Implement the bit-counting circuit using the ASM chart shown in Figure 1 on a DE1 board. The inputs to your circuit should consist of an 8-bit input connected to slider switches SW_{7-0}, an asynchronous reset connected to KEY_0, and a start signal (s) connected to switch SW_8. Your circuit should display the number of 1s in the given 8-bit input value using red LEDs, and signal that the algorithm is finished by lighting up a green LED.

Part II

We wish to implement a binary search algorithm, which searches through an array to locate an 8-bit value A specified via switches SW_{7-0}. A block diagram for the circuit is shown in Figure 2.

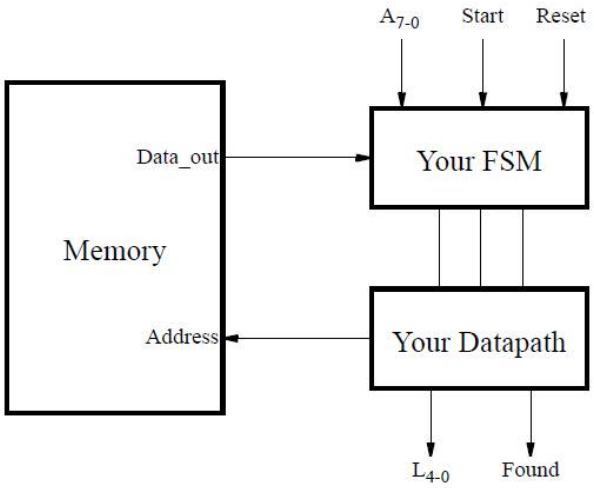


Figure 2: A block diagram for a circuit that performs a binary search.

The binary search algorithm works on a sorted array. Rather than comparing each value in the array to the one being sought, we first look at the middle element and compare the sought value to the middle element. If the middle element has a greater value, then we know that the element we seek must be in the first half of the array. Otherwise, the value we seek must be in the other half of the array. By applying this approach recursively, we can locate the sought element in only a few steps.

In this circuit, the array is stored in an on-chip memory instantiated using MegaWizard Plug-In Manager. To create the approriate memory block, use the the RAM: 1-PORT module from the MegaWizard Plug-In Manager as shown in Figure 3.

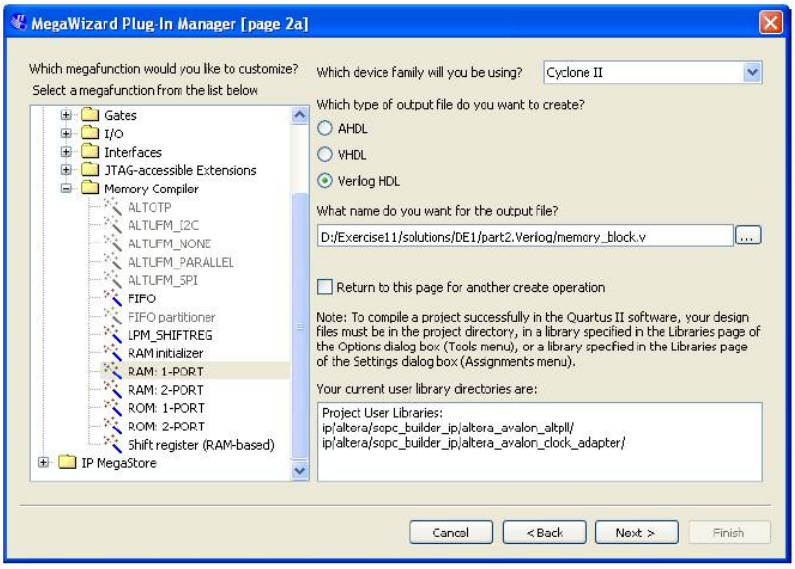


Figure 3: Single-port memory selection using MegaWizard Plug-In Manager.

In the window in Figure 3, specify the Verilog HDL output file to be memory_block.v. When creating the memory block, you should also specify a memory initilization file to be my_array.mif, so that the memory contents can be set to contain an ordered array of numbers.

The circuit should produce a 5-bit value L that specifies the address in the memory where the number A is located. In addition, a signal *Found* should be set high to indicate that the number A was found in the memory, and set low otherwise.

Perform the following steps:

1. Create an ASM chart for the binary search algorithm. Keep in mind that it takes two clock cycles for the data to be read from memory. You may assume that the array has a fixed size of 32 elements.

2. Implement the FSM and the datapath for your circuit.

3. Connect your FSM and datapath to the memory block as shown in Figure 2.

4. Include in your project the necessary pin assignments to implement your circuit on the DE1 board. Use switch SW_8 to drive the processor's *Run* input, use SW_7 to SW_0 to specify the value to be searched, use KEY_0 for *Resetn*, and use the board's 50 MHz clock signal as the *Clock* input. Connect $LEDR_4$ to $LEDR_0$ to show the address in memory of the number A, and $LEDG_0$ for the *Found* signal.

5. Create a file called my_array.mif and fill it with an ordered set of 32 eight-bit integer numbers. You can do this in Quartus II by choosing File > New... from the main menu and selecting Memory Initialization File. This will open a memory file editor, where the contents of the memory may be specified. After this file is created and/or modified, your design needs to be fully recompiled, and downloaded onto the DE1 board for the changes to take effect.

Preparation

The recommended preparation for this exercise is to write Verilog code for Parts I and II.

LAB 4:
BASIC DIGITAL SIGNAL PROCESSING

:: H/ W DEADLINE **WEEK 10**/ PRESENTATION DEADLINE **WEEK 10**::
(Hardware 3-4 per group)

Basic Digital Signal Processing

This exercise is suggested for students who want to use the Audio CODEC on the Altera DE1 board for their project in an introductory digital logic course. Students will design circuit that takes input from an Audio CODEC, alters the sound from the microphone by filtering out noise, and produces the resulting sound through the speakers. In addition to a DE1 board, a microphone and speakers will be required.

Background

Sounds, such as speech and music, are signals that change over time. The amplitude of a signal determines the volume at which we hear it. The way the signal changes over time determines the type of sounds we hear. For example, an 'ah' sound is represented by a waveform shown in Figure 1.

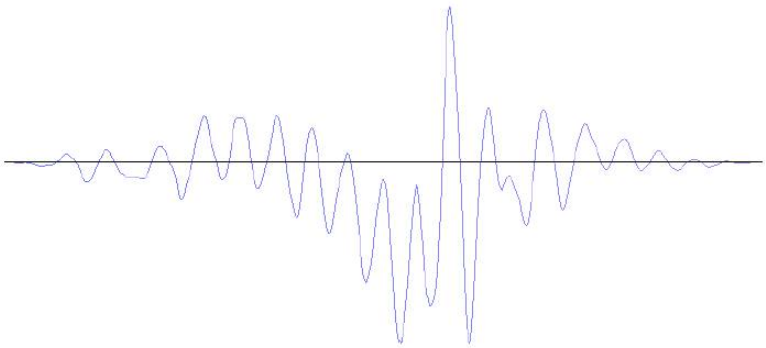


Figure 1: A waveform for an 'ah' sound.

The waveform is an analog signal, which can be stored in a digital form by using a relatively small number of samples that represent the analog values at certain points in time. The process of producing such digital signals is called *sampling*.

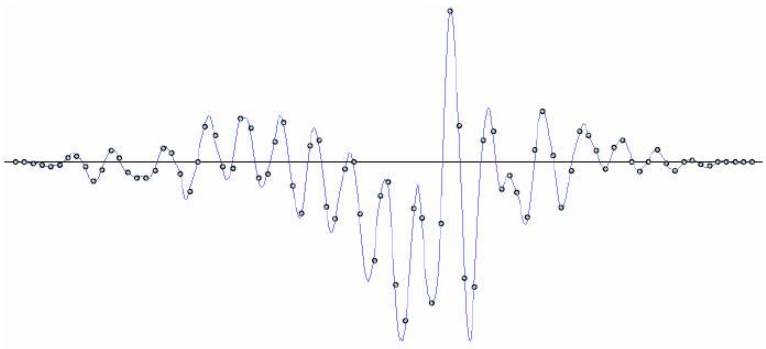


Figure 2: A sampled waveform for an 'ah' sound.

The points in Figure 2 provide a sampled waveform. All points are spaced equally in time and they trace the original waveform.

The Altera DE1 board is equipped with an audio CODEC capable of sampling sound from a microphone and providing it as input to a circuit. By default, the CODEC provides 48000 samples per second, which is sufficient to accurately represent audible sounds.

In this exercise you will create several designs that take input from an Audio CODEC on the Altera DE1 board, record and process the sound from a microphone, and play it back through the speakers. To simplify your task, a simple system that can record and playback sounds on an Altera DE1 board is provided for you. The system, shown in Figure 3, comprises a Clock Generator, an Audio CODEC Interface, and an Audio/Video Configuration modules. This interface is a simplified version of the Altera University Program Audio IP Cores you can find at http://university.altera.com.

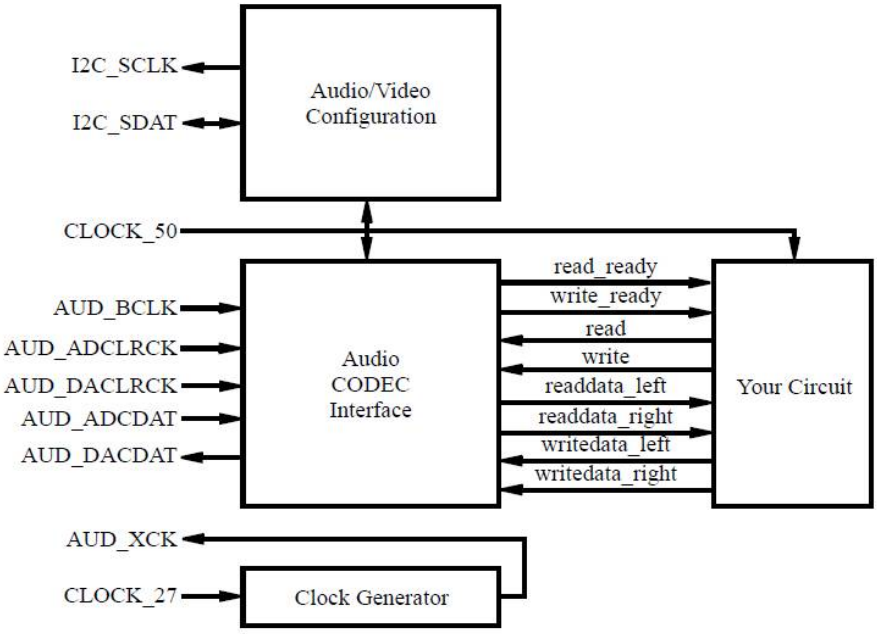


Figure 3: Audio System for this exercise.

The left-hand side of Figure 3 shows the inputs and outputs of the system. These I/O ports supply the clock inputs, as well as connect the Audio CODEC and Audio/Video Configuration modules to the corresponding peripheral devices on the Altera DE1 board. In the middle, a set of signals to and from the Audio CODEC Interface module is shown. These signals allow your circuit, depicted on the right-hand side, to record sounds from a microphone and play them back via speakers.

The system works as follows. Upon reset, the Audio/Video Configuration begins an autoinitialization sequence. The sequence sets up the audio device on the Altera DE1 board to sample microphone input at a rate of 48kHz and produce output through the speakers at the same rate. Once the autoinitialization is complete, the Audio CODEC begins reading the data from the microphone once every 48000^{th} of a second, and sends it to the Audio CODEC Interface core in the system. Once received, the sample is stored in a 128-element buffer in the Audio CODEC Interface core. The first element of the buffer is always visible on the readdata_left and readdata_right outputs when the read_ready signal is asserted. The next element can be read by asserting the read signal, which ejects the current sample and a new one appears one or more clock cycles later, if read_ready signal is asserted.

To output sound through the speakers a similar procedure is followed. Your circuit should observe the write_ready signal, and if asserted write a sample to the Audio CODEC by providing it at the writedata_left and writedata_right inputs and asserting the write signal. This operation stores a sample in a buffer inside of the Audio CODEC Interface, which will then send the sample to the speakers at the right time.

A starter kit that contains this design is provided in a starterkit as part of this exercise.

Part I

In this part of the exercise, you are to make a simple modification to the provided circuit to pass the input from the microphone to the speakers. You should take care to read data from and write data to the Audio CODEC Interface only when its ready signals are asserted.

Compile your circuit and download it onto the Altera DE1 board. Connect microphone and speakers to the Mic and Line Out ports of the DE1 board and speak to the microphone to hear your voice through the speakers. After resetting the circuit, you should hear your own voice through the speakers when you talk to the microphone.

Part II

In this part, you will learn a basic signal processing technique known as *filtering*. Filtering is a process of adjusting a signal - for example, removing noise. Noise in a sound waveform is represented by small, but frequent changes to the amplitude of the signal. A simple logic circuit that achieves the task of noise-filtering is an averaging Finite Impulse Response (FIR) filter. The schematic diagram of the filter is shown in Figure 4.

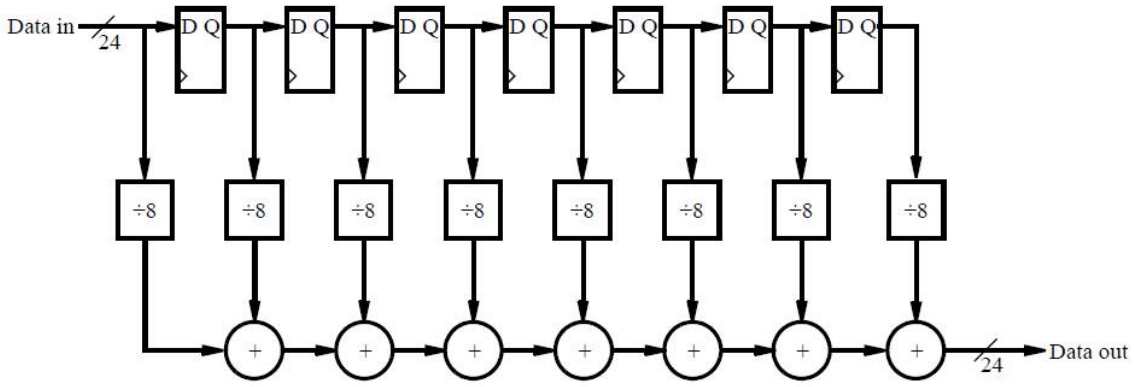


Figure 4: A simple averaging FIR filter.

An averaging filter, like the one shown in Figure 4, removes noise from a sound by averaging the values of adjacent samples. In this particular case, it removes small deviations in sound by looking at changes in the adjacent 8 samples. When using low-quality microphones, this filter should remove the noise produced when you speak to the microphone, making your voice sound clearer.

You are to implement the circuit shown in Figure 4 to process the sound from the microphone, and output the filtered sound through the speakers. Do you notice any difference between the quality of sound in this part as compared to Part I?

NOTE:
It is possible to obtain high-quality microphone with noise-cancelling capabilities. In such circumstances, you are unlikely to hear any effect from using this filter. If this is the case, we suggest introducing some noise into the sound by adding the output of the circuit in Figure 5 to the sample produced by the Audio CODEC.

The circuit is a simple counter, whose value should be interpreted as a signed value. The circuit should be clocked by a 50MHz clock, and the enable signal should be driven high when the Audio CODEC module can both produce and accept a new sample.

To hear the effect of the noise generator, add the values produced by the circuit to each sample of sound from the Audio CODEC in the circuit in part I.

Part III

The implementation of the averaging filter in part II may have been effective in removing some of the noise, and all of the noise produced by the noise generator. However, if your microphone is of low-quality or you increate

```
module noise_generator (clk, enable, Q);
    input clk, enable;
    output [23:0] Q;
    reg [2:0] counter;

    always@(posedge clk)
        if (enable)
            counter = counter + 1'b1;

    assign Q = {{10{counter[2]}}, counter, 11'd0};
endmodule
```

Figure 5: Circuit to generate some noise.

the width of the counter in the noise generator, the filter in part II would be insufficient to remove the noise. The reason for this is because the filter in part II only looked at a very small time frame over which the sound waveform was changing. This can be remedied by making the filter larger, taking an average of more samples.

In this part, you are to experiment with the size of the filter to determine the number of samples over which you have to average sound input to remove background noise. To do this more effectively, use the design of an averaging FIR filter shown in Figure 6.

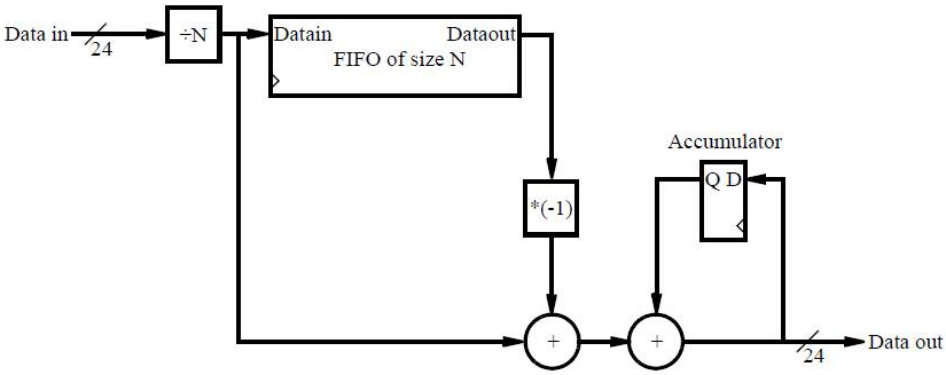


Figure 6: N-sample averaging FIR filter.

To compute the average of the last N samples, this circuit first divides the input sample by N. Then, the resulting value is stored it in a First-In First-out (FIFO) buffer of length N and added to the accumulator. To make sure the value in the accumulator is the average of the last N samples, the circuit subtracts the value that comes out of the FIFO, which represents the $(n+1)^{th}$ sample.

Implement, compule and download the circuit onto Altera DE1 board. Connect microphone and speakers to the Mic and Line Out ports of the DE1 board and speak to the microphone to hear your voice through the speakers. Experiment with different values of N to see what happens to your voice and any background noise, remembering to divide the samples by appropriate value. We recommend experimenting with values of N that are a power of 2, to make the division easier.

If you have a portable music player, with a connector such that you can supply input to your circuit through the Mic port, try experimenting with different sizes of the filter and its effect on the song you play.

LAB 5:

VGA/ AUDIO CODEC/ PS_2 KEYBOARD/ SD_CARD (CASE STUDY)

ASSIGNED : WEEK 7
EVALUATED : WEEK 9/10
:: H/ W DEADLINE **WEEK 6**/ PRESENTATION DEADLINE **WEEK 6**::
(Hardware 3-4 per group)

(1) VGA x3
(2) Audio Codec for DSP x3
(3) PS_2 Keyboard x2
(4) SD_CARD x3

1. You will be given some pre-designed HDL code. You are required to understand, modify and present your case study on week 6 using power point to guide you along your 15 minutes presentation.
2. FPGA implementation together with your associated peripherals must work accordingly and explanation must be given on how you implement each of the above peripherals using FPGA after 2-3 weeks of HDL code study/ modification and implementation.
3. FORMAT OF YOUR POWERPOINT SLIDES ARE:-
    a. TITLE
    b. GROUP MEMBER'S NAME
    c. INTRODUCTION
    d. OBJECTIVES
    e. GENERAL BLOCK DIAGRAM OF HOW YOU CONNECT AND INTEGRATE EACH PERIPHERALS.
    f. METHODOLOGY (HOW YOU IMPLEMENT/ MODIFY YOUR DESIGN, THE CODING METHOD?)
    g. RESULTS AND DISCUSSION (RESULT AND OUTCOME OF YOUR CASE STUDY AND HARDWARE IMPLEMENTATION)
    h. CONCLUSION (WRAP UP/ IMPROVEMENT/ FUTURE TRENDS)
4. Your presentation is 15 minutes with 5 minutes of Q and A.
5. You MUST NOT EXCEED 15 slides.
6. You may show a video of your implementation.

LAB 6:
IMPLEMENTING BIST OR BILBO TESTER

A. FOR BILBO

1. Use HDL Code to implement a combinatorial circuit of your choice (e.g. FA4).
2. Use HDL code to synthesize a BILBO circuit as shown in Figure A2.
3. Split Combinational Circuit into two separate network and perform BILBO test on it (A1).
4. Verify your BILBO tester by means of a HDL based testbench in ModelSIM.
5. Implement your BILBO tester in FPGA.
6. Make sure all 4 modes of BILBO work flawlessly.

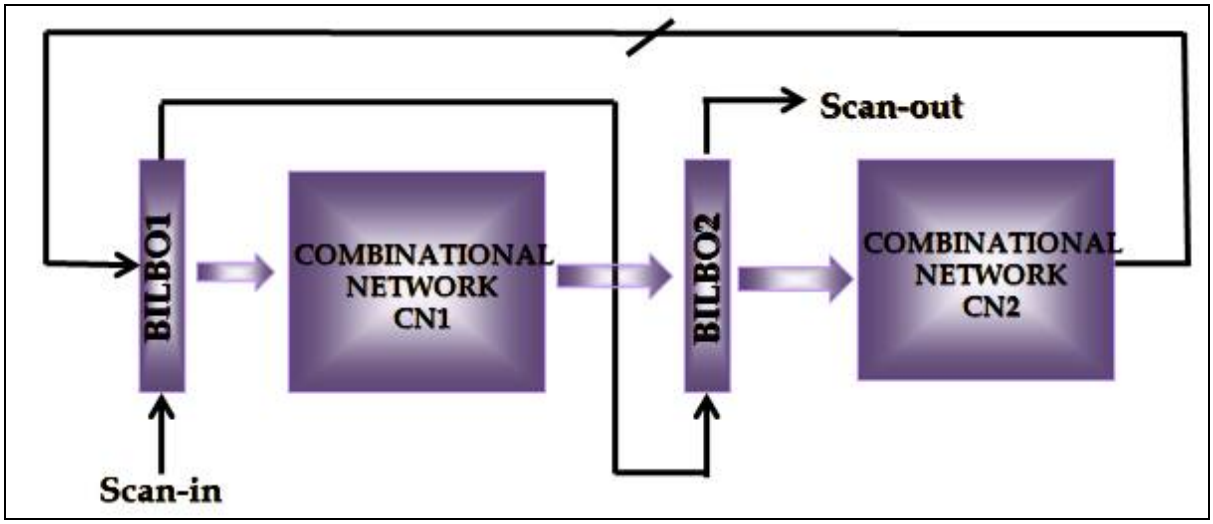


Figure A1: BILBO tester Block diagram

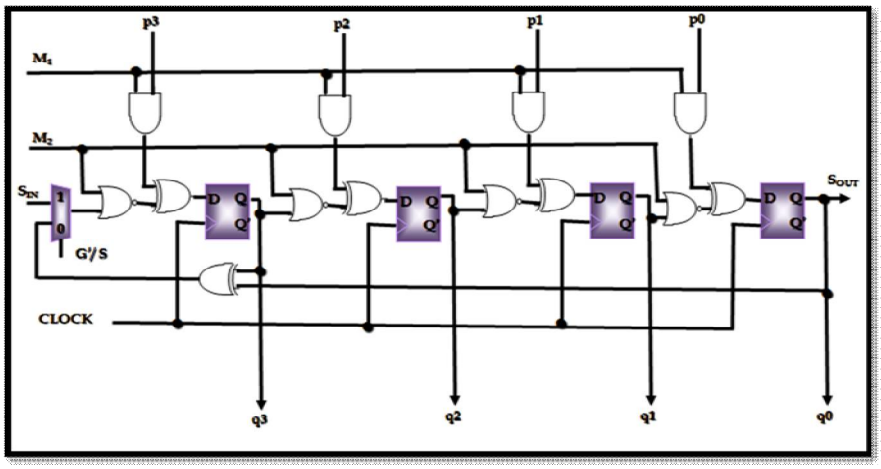


Figure A2: BILBO tester circuit

B. FOR BIST

1. Use HDL Code to implement a sequential circuit of your choice (e.g. a simple FSM)
2. Use HDL code to synthesize a Built In Self Test (BIST) circuit as shown in Figure A3.
3. Verify your BIST tester by means of a HDL based testbench in ModelSIM.
4. Implement your BIST tester on FPGA.
5. Make sure all 4 modes of BILBO work flawlessly.

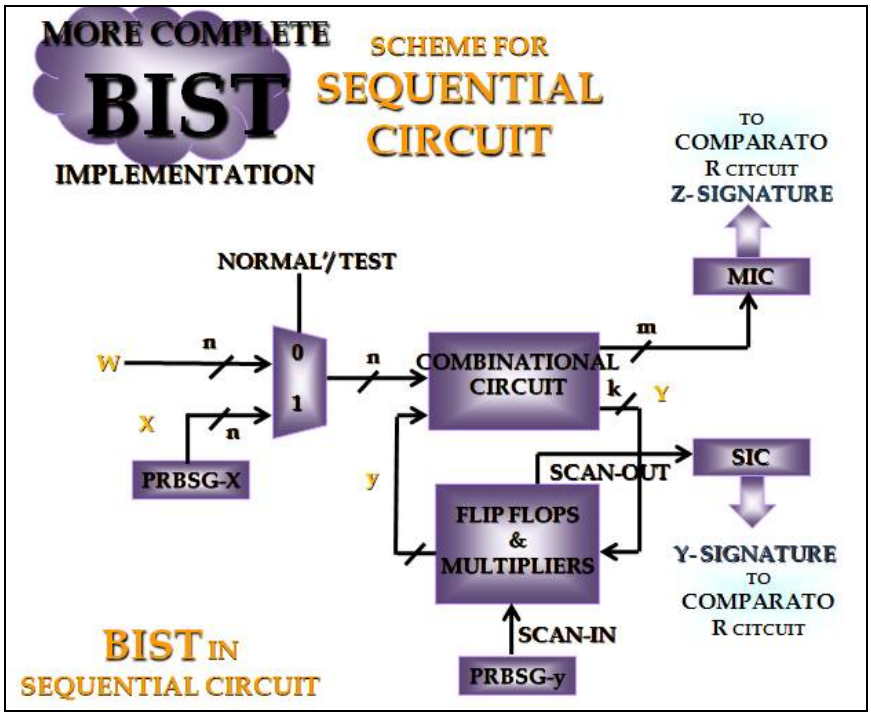


Figure A3: BIST Implementation

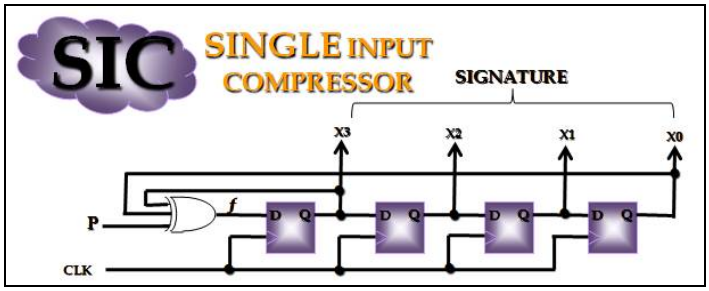


Figure A4: SIC

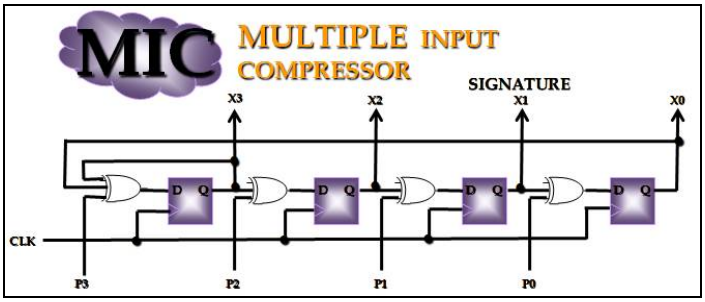


Figure A5: MIC

APPENDIX

:: A_SAMPLE TEST BENCH ::
:: B_HOW TO USE MODELSIM ::
:: C_KEY VERILOG FEATURES ::

APPENDIX A: SAMPLE TEST BENCH CODE

TESTBENCH CODE

```
//...Testbench to check the divide_by_10 logic...
`include "a_div.v"  //include module to be tested
module a_div_tb();

//Declare input as regs and output as wires
reg clock_1, clock_2;
reg load_b, clear_b;
reg  [2:0] data;
reg data_A;
wire [2:0] Q;
wire QA;

initial begin
clock_1 = 0;                    //initial clock1 value
clock_2 = 0;                    //initial clock2 value
clear_b = 0;                    //initial value of clear_b
load_b  = 0;                    //initial value of load_b
data    = 3'b000;               //initial data value (D,C,B)
data_A  = 0;                    //initial data value (A)
#2 clear_b = 1; load_b = 0;     //Asynchronous Load
#2 data = 3'b101; data_A =1;    //data value
#2 data = 3'b010;
#4 clear_b = 0;                 //Asynchronous Clear
#4 clear_b = 1; load_b = 1;     //Asynchronous Counter
#100 $finish;                   //Terminate simulation
end

//.............clock pulse generator..............
always #1 clock_1 = ~clock_1;
always #2 clock_2 = ~clock_2;
//Connect module to test bench
a_div two_five (Q, QA, data, data_A,
load_b, clear_b, clock_1, clock_2);
endmodule
```

DESIGN CODE

```
module a_div (Q, QA, data, data_A,
load_b, clear_b, clock_1, clock_2);
//.............port declaration.................
output [2:0] Q;
output QA;
input  [2:0] data;
input data_A, clock_1,clock_2, load_b, clear_b;
//...........port data type declaration.........
wire [2:0] data;
wire data_A, clock_1, clock_2, load_b, clear_b;
reg [2:0] Q;
reg QA;
//...............code begins here...............
always @ (clear_b or load_b or negedge clock_1 or data_A)
if(!clear_b) begin
    QA <= 0;                                  //Asynchronous Clear
end else if ((clear_b)&&(load_b)) begin
    QA <= !QA;                                //Asynchronous div_2 Counter
end else if (clear_b &&(!load_b)) begin
    QA <= data_A;                             //Asynchronous Load
end
always @ (clear_b or load_b or negedge clock_2 or data)
if(!clear_b) begin
Q <= 3'b0;                                   //Asynchronous Clear
end else if (clear_b && load_b &&(Q==3'b100)) begin
Q <= 3'b0;                                   //Back to 0 after decimal 4 = 5 counts
end else if ((clear_b)&&(load_b)) begin
Q <= Q+1;                                    //Asynchronous div_5 Counter
end else if (clear_b &&(!load_b)) begin
Q <= data;                                   //Asynchronous Load
end
endmodule
```

APPENDIX B: HOW TO USE MODELSIM

Introduction to Simulation of Verilog Designs Using ModelSim Graphical Waveform Editor

For Quartus II 12.0

1 Introduction

This tutorial provides an introduction to simulation of logic circuits using the Graphical Waveform Editor in the ModelSim Simulator. It shows how the simulator can be used to perform functional simulation of a circuit specified in Verilog HDL. It is intended for a student in an introductory course on logic circuits, who has just started learning this material and needs to acquire quickly a rudimentary understanding of simulation.

Contents:

2 Background

ModelSim is a powerful simulator that can be used to simulate the behavior and performance of logic circuits. This tutorial gives a rudimentary introduction to functional simulation of circuits, using the graphical waveform editing capability of ModelSim. It discusses only a small subset of ModelSim features.

The simulator allows the user to apply inputs to the designed circuit, usually referred to as *test vectors*, and to observe the outputs generated in response. The user can use the Waveform Editor to represent the input signals as waveforms.

In this tutorial, the reader will learn about:

- Test vectors needed to test the designed circuit
- Using the ModelSim Graphical Waveform Editor to draw test vectors
- Functional simulation, which is used to verify the functional correctness of a synthesized circuit

This tutorial is aimed at the reader who wishes to simulate circuits defined by using the Verilog hardware description language. An equivalent tutorial is available for the user who prefers the VHDL language.

PREREQUISITE

The reader is expected to have access to a computer that has ModelSim-SE software installed.

3 Design Project

To illustrate the simulation process, we will use a very simple logic circuit that implements the majority function of three inputs, x_1, x_2 and x_3. The circuit is defined by the expression

$$f(x_1, x_2, x_3) = x_1x_2 + x_1x_3 + x_2x_3$$

In Verilog, this circuit can be specified as follows:

```
module majority3 (x1, x2, x3, f);
   input  x1, x2, x3;
   output f;
   assign f = (x1 & x2) | (x1 & x3) | (x2 & x3);
endmodule
```

Enter this code into a file called *majority.v*.

ModelSim performs simulation in the context of *projects* – one project at a time. A project includes the design files that specify the circuit to be simulated. We will first create a directory (folder) to hold the project used in the tutorial. Create a new directory and call it *modelsim_intro*. Copy the file *majority.v* into this directory.

Open the ModelSim simulator. In the displayed window select File > New > Project, as shown in Figure 1.

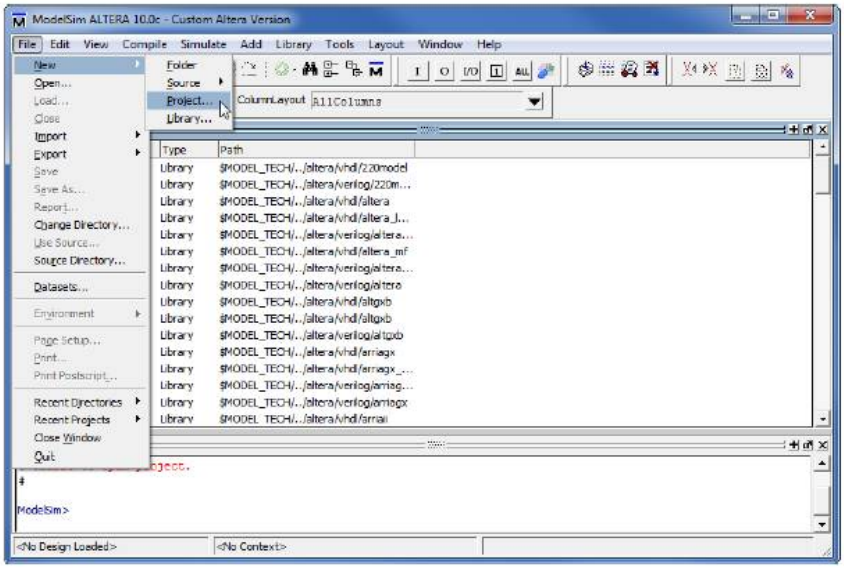


Figure 1. The ModelSim window.

A Create Project pop-up box will appear, as illustrated in Figure 2. Specify the name of the project; we chose the name *majority*. Use the Browse button in the Project Location box to specify the location of the directory that you created for the project. ModelSim uses a working library to contain the information on the design in progress; in the Default Library Name field we used the name *work*. Click OK.

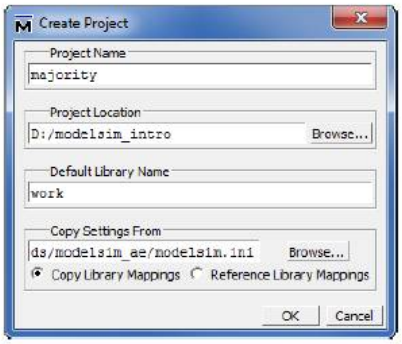


Figure 2. Created Project window.

In the pop-up window in Figure 3, click on Add Existing File and add the file *majority.v* to the project as shown in Figure 4. Click OK, then close the windows.

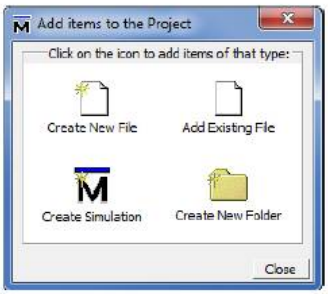


Figure 3. Add Items window.

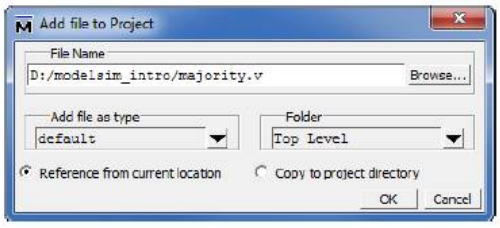


Figure 4. Add Items window.

At this point, the main Modelsim window will include the file as indicated in Figure 5. Observe that there is a question mark in the Status column. Now, select Compile > Compile All, which leads to the window in Figure 6 indicating in the Transcript window (at the bottom) that the circuit in the *majority.v* file was successfully compiled. Note that this is also indicated by a check mark in the Status column. The circuit is now ready for simulation.

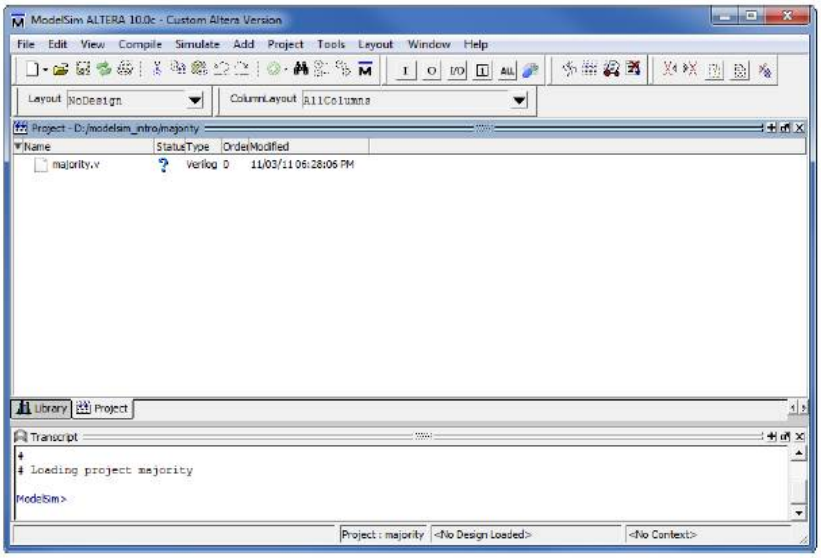


Figure 5. Updated ModelSim window.

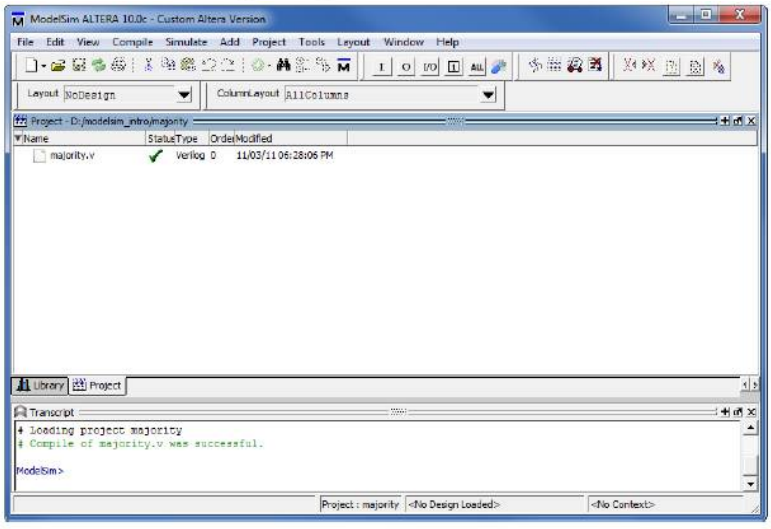


Figure 6. ModelSim window after compilation.

4 Creating Waveforms for Simulation

To perform simulation of the designed circuit, it is necessary to enter the simulation mode by selecting Simulate > Start Simulation. This leads to the window in Figure 7.

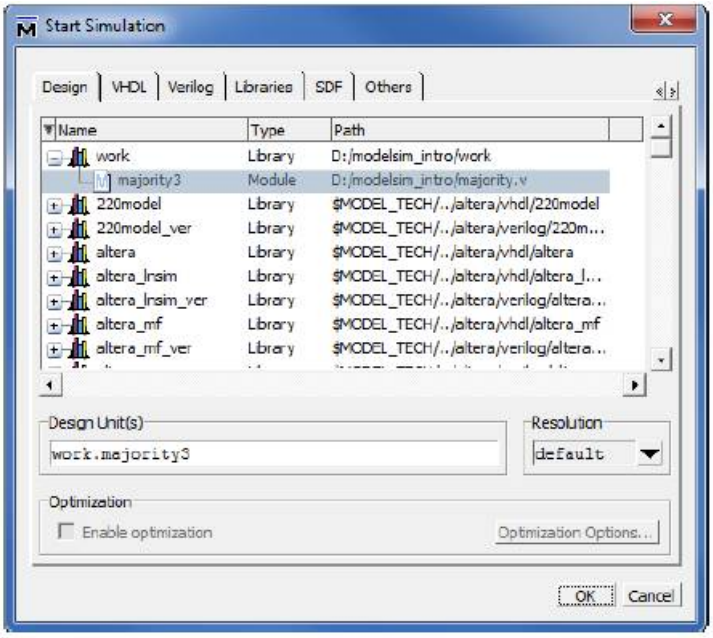


Figure 7. Start Simulation window.

Expand the *work* directory and select the design called *majority*, as shown in the figure. Then click OK. Now, an Objects window appears in the main ModelSim window. It shows the input and output signals of the designed circuit, as depicted in Figure 8.

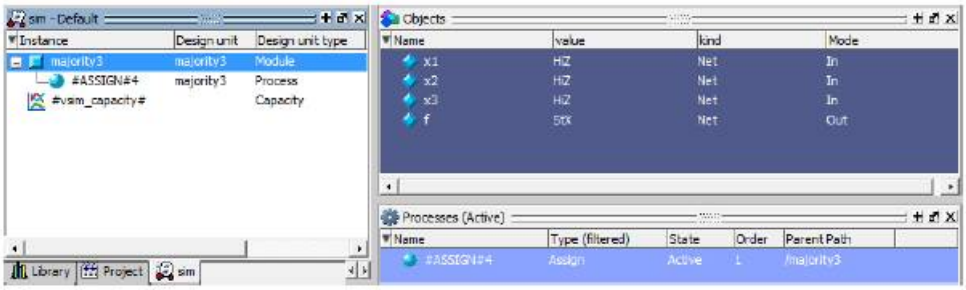


Figure 8. Signals in the Objects window.

To simulate the circuit we must first specify the values of input signals, which can be done by drawing the input waveforms using the Graphical Waveform Editor. Select View > Wave which will open the Wave window depicted in Figure 9. The Wave window may appear as a part of the main ModelSim window; in this case undock it by

clicking on the Dock/Undock icon in the top right corner of the window and resize it to a suitable size. If the Wave window does not appear after undocking, then select View > Wave in the main ModelSim window.

We will run the simulation for 800 ns; so, select View > Zoom > Zoom Range and in the pop-up window that will appear specify the range from 0 to 800 ns. This should produce the image in Figure 9. To change the units, right-click on the scale and select Grid & Timeline Properties..., then select ns in the time units drop-down menu.

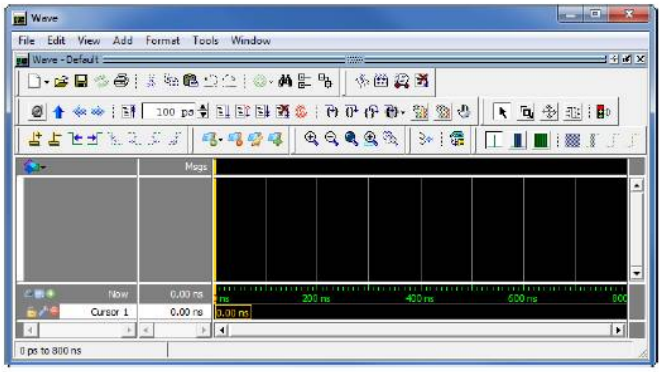


Figure 9. The Wave window.

For our simple circuit, we can do a complete simulation by applying all eight possible valuations of the input signals x_1, x_2 and x_3. The output f should then display the logic values defined by the truth table for the majority function. We will first draw the waveform for the x_1 input. In the Objects window, right-click on $x1$. Then, choose Create Wave in the drop-down box that appears, as shown in Figure 10. This leads to the window in Figure 11, which makes it possible to specify the value of the selected signal in a time period that has to be defined. Choose Constant as the desired pattern, zero as the start time, and 400 ns as the end time. Click Next. In the window in Figure 12, enter 0 as the desired logic value. Click Finish. Now, the specified signal appears in the Wave window, as indicated in Figure 13.

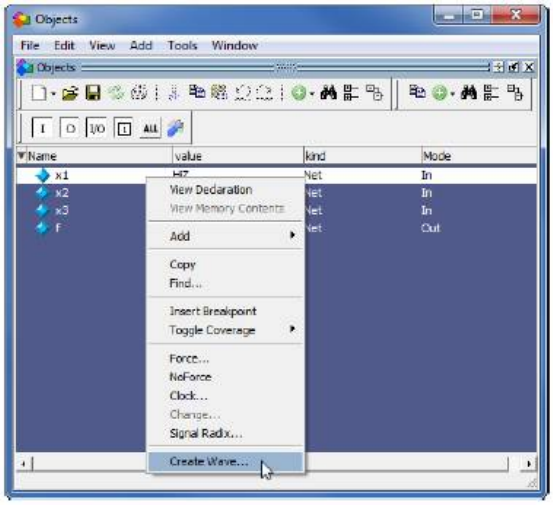


Figure 10. Selecting a signal in the Objects window.

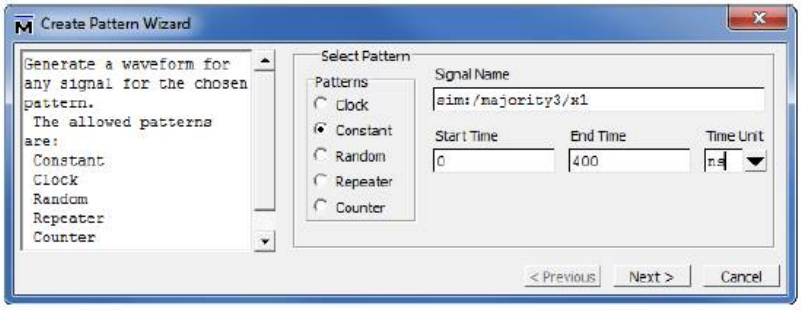


Figure 11. Specifying the type and duration of a signal.

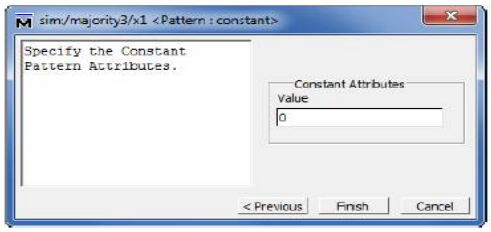


Figure 12. Specifying the value of a signal.

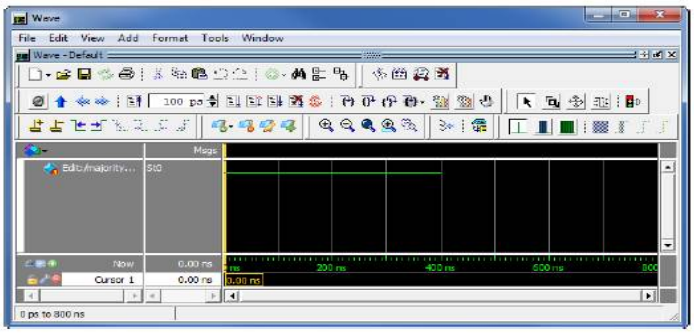


Figure 13. The updated Wave window.

To draw the rest of the x_1 signal, right-click on its name in the Wave window. In the drop-down window that appears, select Edit > Create/Modify Waveform. This leads again to the window in Figure 11. Now, specify 400 ns as the start time and 800 ns as the end time. Click Next. In the window in Figure 12, specify 1 as the required logic value. Click Finish. This completes the waveform for x_1, as displayed in Figure 14.

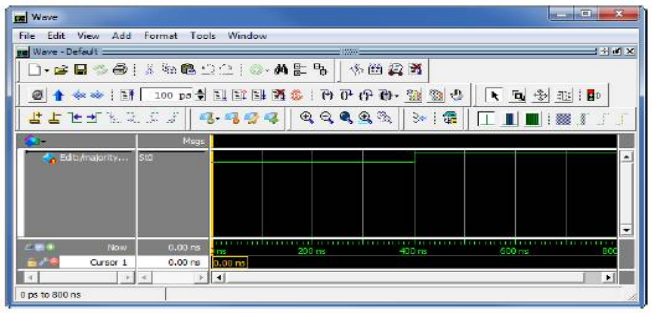


Figure 14. The completed waveform for x_1 input.

ModelSim provides different possibilities for creating and editing waveforms. To illustrate another approach, we will specify the waveform for x_2 by first creating it to have a 0 value throughout the simulation period, and then editing it to produce the required waveform. Repeat the above procedure, by right-clicking on $x2$ in the Objects window, to create a waveform for x_2 that has the value 0 in the interval 0 to 800 ns. So far, we used the Wave window in the Select Mode which is indicated by the highlighted icon . Now, click on the Edit Mode icon , and then right-click to reach the drop-down menu shown in Figure 15 and select Wave Edit. Note that this causes the toolbar menu to include new icons for use in the editing process.

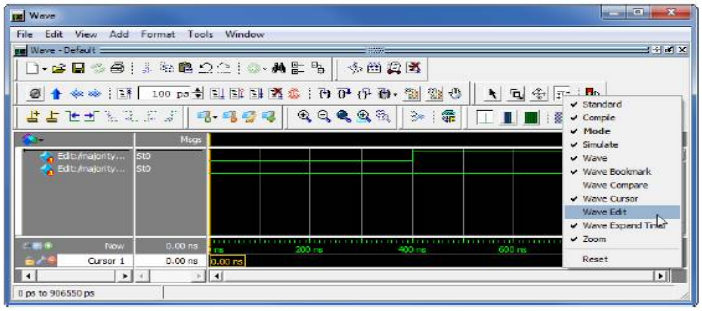


Figure 15. Selecting the Wave Edit mode.

The waveform for x_2 should change from 0 to 1 at 200 ns, then back to 0 at 400 ns, and again to 1 at 600 ns. Select $x2$ for editing by clicking on it. Then, click just to the right of the 200-ns point, hold the mouse button down and sweep to the right until you reach the 400-ns point. The chosen interval will be highlighted in white, as shown in Figure 16. Observe that the yellow cursor line appears and moves as you sweep along the time interval. To change the value of the waveform in the selected interval, click on the Invert icon as illustrated in the figure. A pop-up box in Figure 17 will appear, showing the start and end times of the selected interval. If the displayed times are not exactly 200 and 400 ns, then correct them accordingly and click OK. The modified waveform is displayed in Figure 18. Use the same approach to change the value of x_2 to 1 in the interval from 600 to 800 ns, which should yield the result in Figure 19.

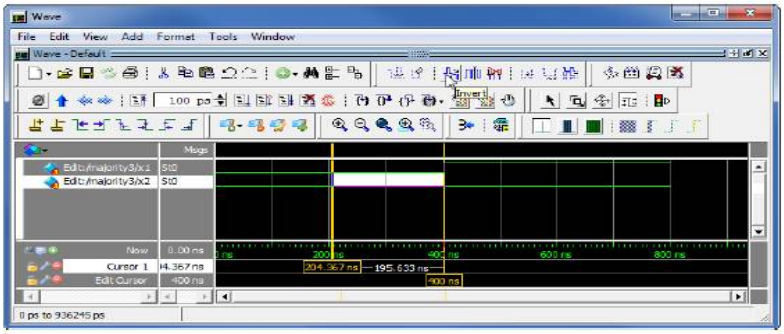


Figure 16. Editing the waveform.

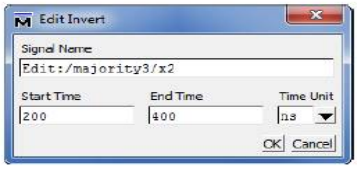


Figure 17. Specifying the exact time interval.

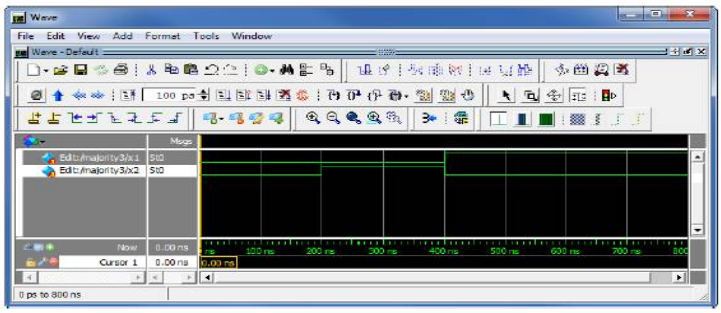


Figure 18. The modified waveform.

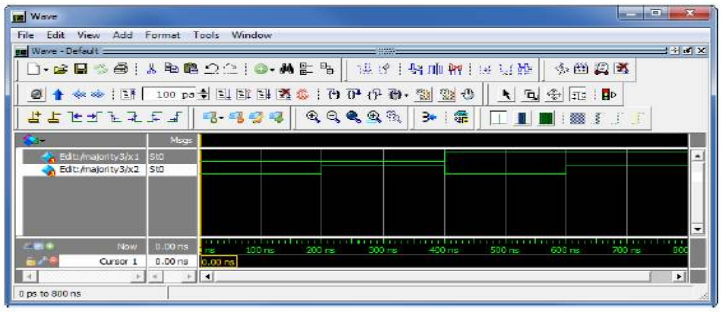


Figure 19. Completed waveforms for x_1 and x_2.

We will use a third approach to draw the waveform for x_3. This signal should alternate between 0 and 1 logic values at each 100-ns interval. Such a regular pattern is indicative of a *clock* signal that is used in many logic circuits. To illustrate how a clock signal can be defined, we will specify x_3 in this manner. Right-click on the x_3 input in the Objects window and select Create Wave. In the Create Pattern Wizard window, select Clock as the required pattern, and specify 0 and 800 ns as the start and end times, respectively, as indicated in Figure 20. Click Next, which leads to the window in Figure 21. Here, specify 0 as the initial value, 200 ns as the clock period, and 50 as the duty cycle. Click Finish. Now, the waveform for x_3 is included in the Wave window.

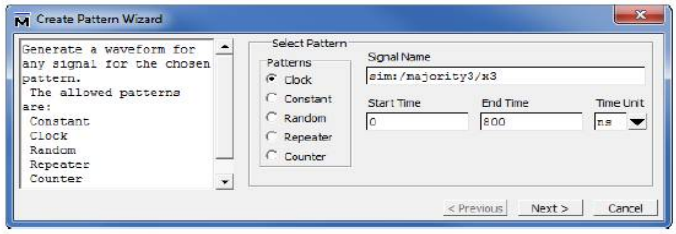


Figure 20. Selecting a signal of clock type.

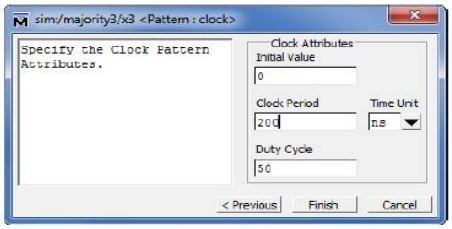


Figure 21. Defining the characteristics of a clock signal.

Lastly, it is necessary to include the output signal f. Right-click on f in the Objects window. In the drop-down menu that appears, select Add > To Wave > Selected Signals as shown in Figure 22. The result is the image in Figure 23.

Figure 22. Adding a signal to the Wave window.

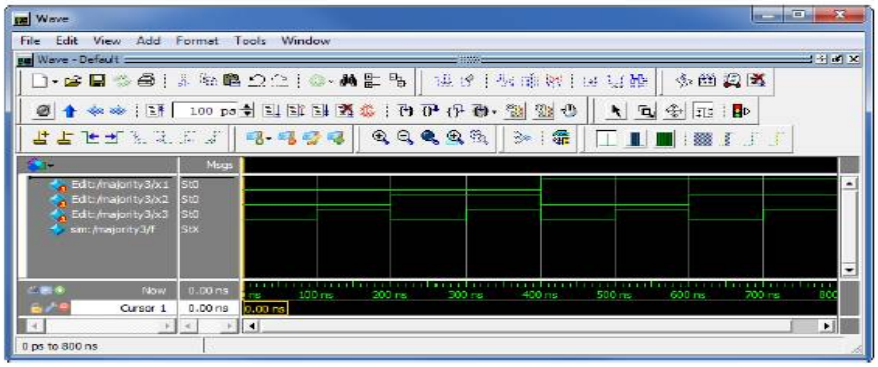


Figure 23. The completed Wave window.

Save the created waveforms as *majority.do* file, as indicated in Figure 24.

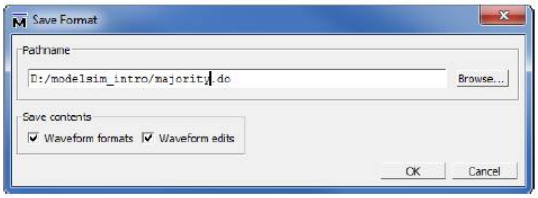


Figure 24. Saving the waveform file.

5 Simulation

To perform the simulation, open the Wave window and specify that the simulation should run for 800 ns, as indicated in Figure 25. Then, click on the Run-All icon, as shown in Figure 26. The result of the simulation will be displayed as presented in Figure 27. Observe that the output f is equal to 1 whenever two or three inputs have the value 1, which verifies the correctness of our design.

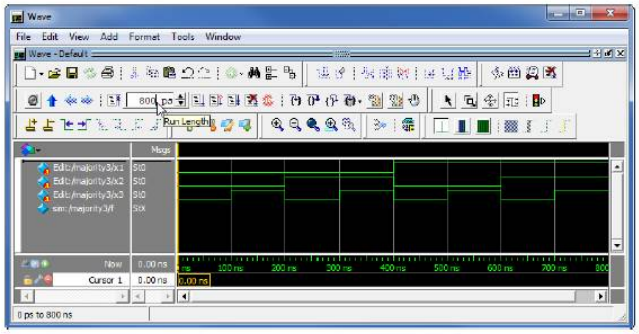


Figure 25. Setting the simulation interval.

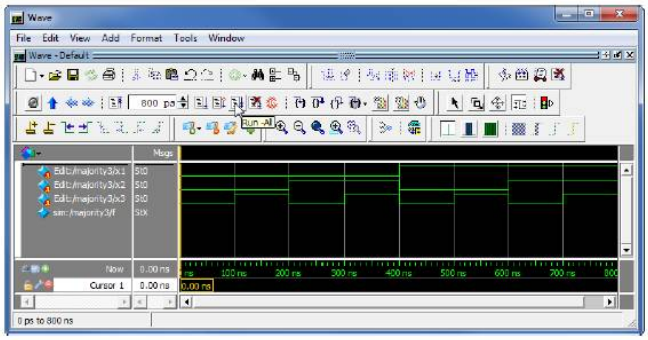


Figure 26. Running the simulation.

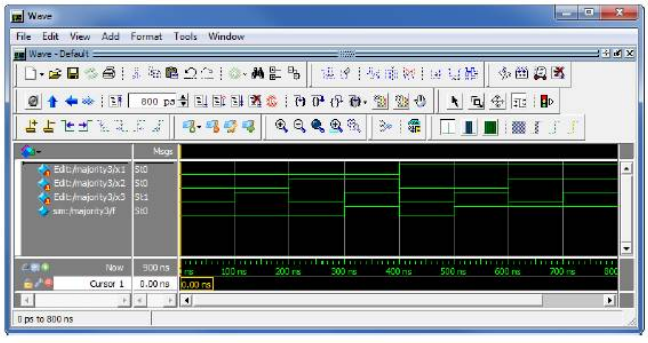


Figure 27. Result of the simulation.

6 Making Changes and Resimulating

Changes in the input waveforms can be made using the approaches explained above. Then, it is necessary to resimulate the circuit using the altered waveforms. For example, change the waveform for x_1 to have the logic value 1 in the interval from 0 to 200 ns, as indicated in Figure 28. Now, click on the Restart icon shown in the figure. A pop-up box in Figure 29 will appear. Leave the default entries and click OK. Upon returning to the Wave window,

simulate the design again by clicking on the Run-All icon. The result is given in Figure 30.

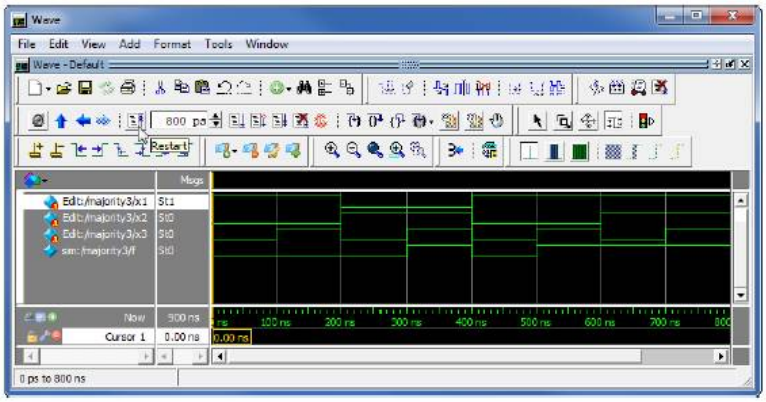


Figure 28. Changed input waveforms.

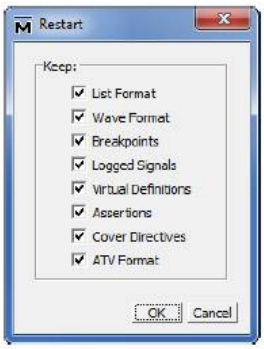


Figure 29. The Restart box.

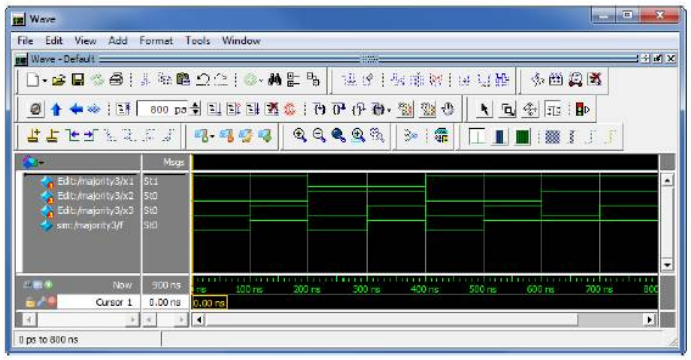


Figure 30. Result of the new simulation.

Simulation is a continuous process. It can be stopped by selecting Simulate > End Simulation in the main ModelSim window.

7 Concluding Remarks

The purpose of this tutorial is to provide a quick introduction to ModelSim, explaining only the rudimentary aspects of functional simulation that can be performed using the ModelSim Graphical User Interface. More details about the ModelSim GUI and its use in simulation can be found in the *Generating Stimulus with Waveform Editor* chapter of *ModelSim SE User's Manual*, which is available as part of an installed ModelSim-SE simulator.

A more extensive discussion of simulation using the ModelSim simulator is provided in the tutorial *Using ModelSim to Simulate Logic Circuits in Verilog Designs* and *Using ModelSim to Simulate Logic Circuits in VHDL Designs*, which are available on Altera's University Program Web site.

APPENDIX C: KEY VERILOG FEATURES

Summary of Key Verilog Features (IEEE 1364)

Module

Encapsulates functionality; may be nested to any depth.

module module_name (list of ports);
Declarations
Port modes: **input**, **output**, **inout** identifier;
Nets (e.g., **wire** A[3:0];)
Register variable (e.g., **reg** B[31: 0];)
Constants: (e.g. **parameter** size = 8;)
Named events
Continuous assignments
(e.g. **assign** sum = A + B;)
Behaviors **always** (cyclic), **initial** (single-pass)
specify ... **endspecify**
function ... **endfunction**
task ... **endtask**
Instantiations
primitives
modules
endmodule

Multi Input Primitives

(Each input is a scalar)

and (out, in_1, in_2, ... in_N);
nand (out, in_1, in_2, ... in_N);
or (out, in_1, in_2, ... in_N);
nor (out, in_1, in_2, ... in_N);
xor (out, in_1, in_2, ... in_N);
xnor (out, in_1, in_2, ... in_N);

Multi-Output Primitives

buf (out_1, out_2, ..., out_N, in); // buffer
not (out_1, out_2, ..., out_N, in); // inverter

Three-State Multi-Output Primitives

bufif0 (out, in, control); **bufif1** (out, in, control);
notif0 (out, in, control); **notif1** (out, in, control);

Pullups and Pulldowns

pullup (out_y); **pulldown** (out_y);

Propagation Delays

Single delay: **and** #3 G1 (y, a, b, c);
Rise/fall: **and** #(3, 6) G2 (y, a, b, c);
Rise/fall/turnoff: **bufif0** #(3, 6, 5) (y, x_in, en);
Min:typ:Max: **bufif1** #(3:4:5, 4:5:6, 7,8:9) (y, x_in, en);

Command line options for single delay value simulation:
+maxdelays, **+typdelays**, **+mindelays**

Example: verilog +mindelays testbench.v

Concurrent Behavioral Statements

May execute a level-sensitive assignment of value to a net (keyword: **assign**), or may execute the statements of a cyclic (keyword: **always**) or single-pass (keyword: **initial**) behavior. The statements execute sequentially, subject to level-sensitive or edge-sensitive event control expressions.

Syntax:

assign net_name = [expression];
always begin [procedural statements] **end**
initial begin [procedural statements] **end**

Cyclic (**always**) and single-pass (**initial**) behaviors may be level sensitive and/or edge sensitive.

Edge sensitive:

always @(posedge clock)
q <= data;

Level sensitive:

always @ (enable or data)
if (enable) q = data

Data Types: Nets and Registers

Nets: Establish structural connectivity between instantiated primitives and/or modules; may be target of a continuous assignment; e.g., **wire**, **tri**, **wand**, **wor**.

Value is determined during simulation by the driver of the net; e.g., a primitive or a continuous assignment. (Example: **wire** Y = A + B.)

Registers: Store information and retain value until reassigned.

Value is determined by an assignment made by a procedural statement.

Value is retained until a new assignment is made; e.g., **reg**, **integer**, **real**, **realtime**, **time**.

Example:

```
always @ (posedge clock)
  if (reset) q_out <=0;
    else q_out <= data_in;
```

Procedural Statements

Describe logic abstractly; statements execute sequentially to assign value to variables.

```
if (expression_is_true) statement_1; else
statement_2;
case (case_expression)
case_item: statement;
...
default: statement;
endcase
for (conditions ) statement;
repeat constant_expression statement;
while (expression_is_true) statement;
forever statement;
fork statements join  // execute in parallel
```

Assignments

Continuous: Continuously assigns the value of an expression to a net.

Procedural (Blocked): Uses the = operator; executes statements sequentially; a statement cannot execute until the preceding statement completes execution. Value is assigned immediately.

Procedural (Nonblocking): Uses the <= operator; executes statements concurrently, independent of the order in which they are listed. Values are assigned concurrently.

Procedural (Continuous):

assign ... **deassign** overrides procedural assignments to a net.

force ... **release** overrides all other assignments to a net or a register.

Operators

{ }, {{ }}	concatenation
+ − * /	arithmetic
%	modulus
> >= < <=	relational
!	logical negation
&&	logical and
||	logical or
==	logical equality
!=	logical inequality
===	case equality
!==	case inequality
~	bitwise negation
&	bitwise and
|	bitwise or
^	bitwise exclusive-or
^~ or ~^	bitwise equivalence
&	reduction and
~&	reduction nand
!	or
~|	reduction nor
^	reduction exclusive-or
~^ or ^~	reduction xnor
<<	left shift
>>	right shift
?:	conditional
or	Event or

Specify Block

Example: Module Path Delays

```
specify
// specparam declarations (min: typ: max)
specparam t_r = 3:4:5, t_f = 4:5:6;
(A, B) *> Y) =  (t_r, t_f);          // full
(Bus_1 => Bus_1) = (t_r, t_f); // parallel
if (state == S0) (a, b *> y) = 2; // state dep
(posedge clk => (y -: d_in)) = (3. 4); // edge
endspecify
```

Example: Timing Checks

```
specify
specparam t_setup = 3:4:5, t_hold = 4:5:6;

$setup (data, posedge clock, t_setup);
$hold (posedge clock, data, t_hold);
endspecify
```

Memory

Declares an array of words.

Example: Memory declaration and readout

```
module memory_read_display();
  reg [31: 0] mem_array [1: 1024];
  integer k;
```