Name of Candidate: _____

| Assessment of Practical/ Test/ Presentation (100%) | | | | | |
|---|---|---|---|---|---|
| | | 1-4 | 5-8 | 9-12 | 13-16 |
| 1. | Overall comprehension /knowledge of the assignment carried out. | Does not demonstrate adequate comprehension of the assignment carried out (unable to answer any question). | Demonstrates poor comprehension of the assignment carried out (able to answer a few basic questions with prompting). | Demonstrates satisfactory comprehension of the assignment carried out (able to answer some key questions). | Demonstrates go comprehension o assignment carrie (able to answer m questions well). |
| 2. | Quality of actual work done/carried out. | Work done insufficient to meet assignment objectives. | Minimal work done, covers some key aspect(s) with errors. | Work done is adequate to cover all key aspects with minor errors (some objectives may not be met). | Work done meets assignment objec with minor errors. |
| 3 | Candidate's ability to use proper tools and techniques for intended tasks. | Unable to use proper tools and techniques for intended tasks, unresponsive to Lecturer. | Unable to use proper workshop tools and techniques for intended tasks, major guidance from Lecturer is needed. | Able to use proper tools and techniques with minimal guidance from Lecturer. | Able to use prope and techniques a techniques witho guidance. |
| 4. | Functionality & Correctness of Schematic/ Layout drawing with size optimization | Non functional & incorrect Schematic/ Layout drawing. Without optimizations | Functional with inaccurate Schematic/ Layout drawing. Without optimizations | Functional with minimal. Correct Schematic/ Layout drawing. Without optimizations. | Functional withou Correct Schemat Layout. With mini optimizations. |
| 5. | Candidate's ability to use logical troubleshooting procedures to isolate faulty stage/ ability to resolve problems | Unable to perform troubleshooting /resolve problems, unresponsive to Lecturer. | Unable to perform troubleshooting /resolve problems, major guidance from Lecturer is needed. | Able to perform some troubleshooting /resolve some of the problems encountered, with guidance from Lecturer. | Demonstrates go of logical trouble procedures to iso faulty stage/resol problems using lo systematic metho to resolve most o problems encoun |

**Comment of Practical:**

**Signature of Lecturer:** **Date:**

| Assessment of Report (100%) | | | | | |
|---|---|---|---|---|---|
| | | 1-4 | 5-8 | 9-12 | 13-16 |
| 1. | Overall comprehension /knowledge of the assignment carried out. | Does not demonstrate adequate comprehension of the assignment carried out (unable to answer any question). | Demonstrates poor comprehension of the assignment carried out (able to answer a few basic questions with prompting). | Demonstrates satisfactory comprehension of the assignment carried out (able to answer some key questions). | Demonstrates go comprehension o assignment carrie (able to answer m questions well). |
| 2. | Candidate's ability to define assignment objective, conclude and present results/ outcome. | Unable to formulate assignment objective & conclude or produce results. Unresponsive to Lecturer. | Unable to formulate assignment objective & conclude sufficiently or produce incorrect results appropriately. Merely following instructions from Lecturer. | Formulate assignment objective, conclusions and produce results with guidance from Lecturer. | Able to formula appropriate assignment obj conclusion and produce correc results. Minor corrections nee Resources nee clearly identifie |
| 3 | Candidate's ability to define assignment methodology (with schematic/ layout & optimizations) and scheduling. | Unable to formulate methodology and unresponsive to Lecturer. Non functional & incorrect Schematic/ Layout drawing. Without optimizations | Unable to formulate methodology, merely following instruction from Lecturer. Functional with inaccurate Schematic/ Layout drawing. Without optimizations | Formulation of methodology and scheduling with guidance from Lecturer. Functional with minimal. Correct Schematic/ Layout drawing. Without optimizations. | Able to formula appropriate methodology w scheduling. Mi guidance & corrections. Fu without error. Co Schematic & Lay minimal optimiza |

Name of Candidate: _____

| | | | | | |
|---|---|---|---|---|---|
| 4. | Candidate's ability to perform relevant discussions. | Unable to perform discussion or analysis on the results obtained/ unable to resolve problems. unresponsive to Lecturer. | Unable to perform discussion or analysis on the results obtained/ unable to resolve problems adequately, major guidance from Lecturer is needed. | Able to perform some discussion or analysis on the results obtained/ /resolve some of the problems encountered, with guidance from Lecturer. | Performs good di or analysis on the obtained/ Able to most of the probl encountered with much guidance fr lecturer. |
| 5. | Presentation of Report (Assignment format and contents, effort & creativity) | No formatting, effort or creativity shown, contents are mostly irrelevant to assignment | Unable to write assignment with proper format and contents, merely following instruction from Lecturer. Little effort & creativity. | Able to write assignment with correct format and appropriate contents with guidance from Lecturer. Some effort with little or no creativity shown. | Able to write cle concise assignr with correct forn appropriate cor Minor correctio needed. Good and slight creat shown. |

**Comment of Practical:**

**Signature of Lecturer:** **Date:**

Name of Candidate: _____

**Sample Title Page**

# DESIGN OF A CMOS 0.35µm SIMPLE PROCESSOR

Lim Sheng Yang (09WTD012345)

Assignment submitted in partial fulfillment of the requirements for
Digital System Design (BAME2004) course of the Bachelor in Science
(Microelectronics with Embedded Technology)

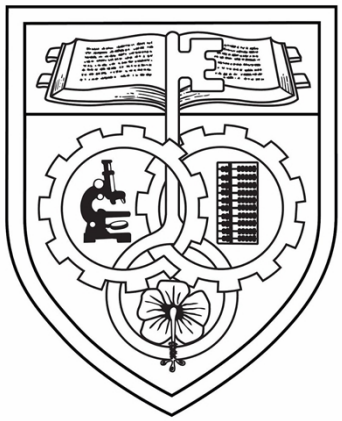
Department of Physical Sciences
Faculty of Applied Sciences and Computing (FASC)
TARC University College
Kuala Lumpur

September 2018

Name of Candidate: _____

# LABORATORY MANUAL

## FACULTY OF
## APPLIED SCIENCES AND COMPUTING



TARC

TUNKU ABDUL RAHMAN
UNIVERSITY COLLEGE

## BAME 2004

# DIGITAL SYSTEM DESIGN

Revised
Academic Year 2018/ 19

Name of Candidate: _____

# ASSIGNMENT FORMAT

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

## LAB REPORT (SUGGESTED FORMAT)

**CONTENTS**


**1. INTRODUCTION**

1.1 OBJECTIVES (POINT FORM)

1.2 PROBLEM STATEMENT

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

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

2.2 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.

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

5 **REFERENCES**

**6 APPENDIX**

Name of Candidate: _____

# 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.

Name of Candidate: _____

(…… …….. discuss past researcher's design on …….. topic……..).

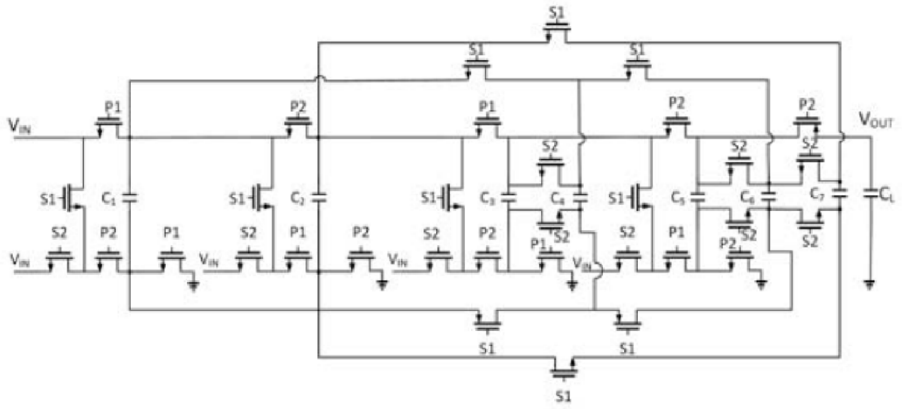



Fig. 1. Past researcher's design on ……. (Kumar, 2013)

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:-

Name of Candidate: _____

** 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

Name of Candidate: _____


| | PAGE |
|---|---|
| **ASSIGNMENT FORMAT** | 2-5 |
| **ASSESSMENT CRITERIA** | 5 |
| **CONTENTS** | 6 |
| **LAB PLANS** | 7 |
| **LAB 1 : Switches, Light and Multiplexers** | 8-13 |
| **LAB 2 : Numbers and Displays** | 14-18 |
| **LAB 2B : 74585 & Test Bench** | 19-24 |
| **LAB 3 : Latches, Flip-Flop and Registers** | 25-29 |
| **LAB 3B : Dedicated Datapath** | 30 |
| **LAB 3C : General Purpose Datapath** | 31 |
| **LAB 3D : Control Unit** | 33-35 |
| **LAB 4 : Counters** | 36 |
| **LAB 4B : Timing Circuits** | 38 |
| **LAB 5 : Timers and Real Time Clock** | 39-41 |
| **LAB 6 : Adders, Subtractors and Multiplexers** | 42- 46 |
| **LAB 7 : FSM** | 47-52 |
| **LAB 7B : Traffic Light (FSM)** | 53 |
| **LAB 7C : Electric Train Controller (FSM)** | 54-68 |
| **LAB 8 : Register Files** | 69 |
| **LAB 8B : 4 x 4 RAM** | 70 |
| **LAB 9 : GCD (Dedicated µP)** | 72 |
| **LAB 10 : High Low Game (Dedicated µP)** | 74 |
| **LAB 11 : Finding Largest Number (Dedicated µP)** | 76 |
| **APPENDIXES** | 78-91 |

Name of Candidate: _____

# LAB PLANS

**MAY 2018 (10 WEEKS x 2P) TENTATIVE**

WEEK 2 : LAB 1: Switches, Lights, Multiplexers

WEEK 3 : LAB 2: Numbers and Displays + Test Bench (Comparator)

WEEK 4-5 : LAB 4: Counters and Timing Circuits/ Barrel Shifter

WEEK 6-7 : LAB 7B: FSM (Traffic Light/ RC Controller) + Test Bench + FPGA

WEEK 8-9 : LAB 3B: Datapaths (dedicated/ General + Control Unit) + Report

WEEK 10 : LAB 8: Memories/ Register Files + **REPORT 1 (FSM) DEADLINE**

WEEK 11 : LAB 9/10/11: Dedicated microprocessor (TB + FPGA)

WEEK 12 : FINAL DEMO (Final Mini Project) LAB 9/10/11/self-titled

# LAB 1:

## Switches, Lights and Multiplexers

:: WEEK 1::
DURATION: 2 HOURS

Name of Candidate: _____

The purpose of this exercise is to learn how to connect simple input and output devices to an FPGA chip and implement a circuit that uses these devices. We will use the switches $SW_{9-0}$ on the DE1 board as inputs to the circuit. We will use light emitting diodes (LEDs) and 7-segment displays as output devices.

**Part I**

The DE1 board provides 10 toggle switches, called $SW_{9-0}$, that can be used as inputs to a circuit, and 10 red lights, called $LEDR_{9-0}$, that can be used to display output values. Figure 1 shows a simple Verilog module that uses these switches and shows their states on the LEDs. Since there are 10 switches and lights it is convenient to represent them as vectors in the Verilog code, as shown. We have used a single assignment statement for all 10 *LEDR* outputs, which is equivalent to the individual assignments

```
assign LEDR[9] = SW[9];
assign LEDR[8] = SW[8];
...
assign LEDR[0] = SW[0];
```

The DE1 board has hardwired connections between its FPGA chip and the switches and lights. To use $SW_{9-0}$ and $LEDR_{9-0}$ it is necessary to include in your Quartus II project the correct pin assignments, which are given in the *DE1 User Manual*. For example, the manual specifies that $SW_0$ is connected to the FPGA pin *L22* and $LEDR_0$ is connected to pin *R20*. A good way to make the required pin assignments is to import into the Quartus II software the file called *DE1_pin_assignments.qsf*, which is provided on the *DE1 System CD* and in the University Program section of Altera's web site. The procedure for making pin assignments is described in the tutorial *Quartus II Introduction using Verilog Design*, which is also available from Altera.

It is important to realize that the pin assignments in the *DE1_pin_assignments.qsf* file are useful only if the pin names given in the file are exactly the same as the port names used in your Verilog module. The file uses the names *SW*[0] ... *SW*[9] and *LEDR*[0] ... *LEDR*[9] for the switches and lights, which is the reason we used these names in Figure 1.

```
// Simple module that connects the SW switches to the LEDR lights
module part1 (SW, LEDR);
    input [9:0] SW;         // toggle switches
    output [9:0] LEDR;      // red LEDs

    assign LEDR = SW;
endmodule
```

Figure 1. Verilog code that uses the DE1 board switches and lights.

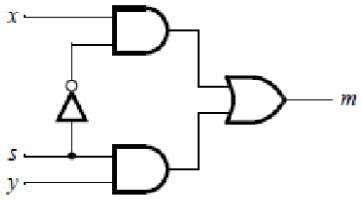
Perform the following steps to implement a circuit corresponding to the code in Figure 1 on the DE1 board.

1. Create a new Quartus II project for your circuit. Select Cyclone II EP2C20F484C7 as the target chip, which is the FPGA chip on the Altera DE1 board.
2. Create a Verilog module for the code in Figure 1 and include it in your project.
3. Include in your project the required pin assignments for the DE1 board, as discussed above. Compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the switches and observing the LEDs.

Name of Candidate: _____

**Part II**

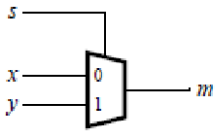
Figure 2$a$ shows a sum-of-products circuit that implements a 2-to-1 *multiplexer* with a select input $s$. If $s = 0$ the multiplexer's output $m$ is equal to the input $x$, and if $s = 1$ the output is equal to $y$. Part $b$ of the figure gives a truth table for this multiplexer, and part $c$ shows its circuit symbol.




a) Circuit

| $s$ | $m$ |
|---|---|
| 0 | $x$ |
| 1 | $y$ |

b) Truth table




c) Symbol

Figure 2. A 2-to-1 multiplexer.

The multiplexer can be described by the following Verilog statement:

```
assign m = (~s & x) | (s & y);
```

You are to write a Verilog module that includes four assignment statements like the one shown above to describe the circuit given in Figure 3$a$. This circuit has two four-bit inputs, $X$ and $Y$, and produces the four-bit output $M$. If $s = 0$ then $M = X$, while if $s = 1$ then $M = Y$. We refer to this circuit as a four-bit wide 2-to-1 multiplexer. It has the circuit symbol shown in Figure 3$b$, in which $X$, $Y$, and $M$ are depicted as four-bit wires. Perform the steps shown below.

1. Create a new Quartus II project for your circuit.
2. Include your Verilog file for the four-bit wide 2-to-1 multiplexer in your project. Use switch $SW_9$ on the DE1 board as the $s$ input, switches $SW_{3-0}$ as the $X$ input and $SW_{7-4}$ as the $Y$ input. Connect the *SW* switches to the red lights *LEDR* and connect the output $M$ to the green lights $LEDG_{3-0}$.
3. Include in your project the required pin assignments for the DE1 board. As discussed in Part I, these assignments ensure that the input ports of your Verilog code will use the pins on the Cyclone II FPGA that are connected to the *SW* switches, and the output ports of your Verilog code will use the FPGA pins connected to the *LEDR* and *LEDG* lights.
4. Compile the project.
5. Download the compiled circuit into the FPGA chip. Test the functionality of the four-bit wide 2-to-1 multiplexer by toggling the switches and observing the LEDs.

Name of Candidate: _____

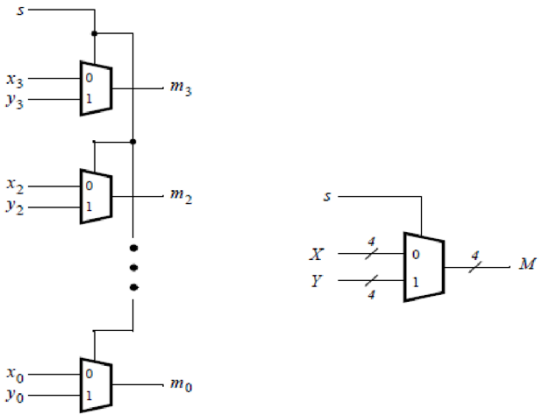
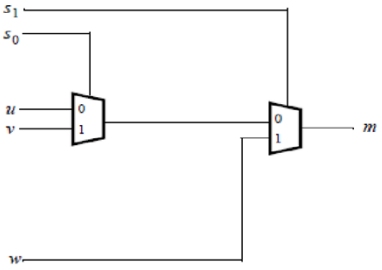



Figure 3. A four-bit wide 2-to-1 multiplexer.

**Part III**

In Figure 2 we showed a 2-to-1 multiplexer that selects between the two inputs $x$ and $y$. For this part consider a circuit in which the output $m$ has to be selected from three inputs $u$, $v$, and $w$. Part $a$ of Figure 4 shows how we can build the required 3-to-1 multiplexer by using two 2-to-1 multiplexers. The circuit uses a 2-bit select input $s_1 s_0$ and implements the truth table shown in Figure 4$b$. A circuit symbol for this multiplexer is given in part $c$ of the figure.

Recall from Figure 3 that a four-bit wide 2-to-1 multiplexer can be built by using four instances of a 2-to-1 multiplexer. Figure 5 applies this concept to define a two-bit wide 3-to-1 multiplexer. It contains two instances of the circuit in Figure 4$a$.




| $s_1$ | $s_0$ | $m$ |
|---|---|---|
| 0 | 0 | $u$ |
| 0 | 1 | $v$ |
| 1 | 0 | $w$ |
| 1 | 1 | $w$ |

b) Truth table

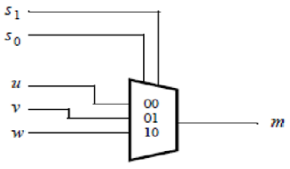



Figure 4. A 3-to-1 multiplexer.

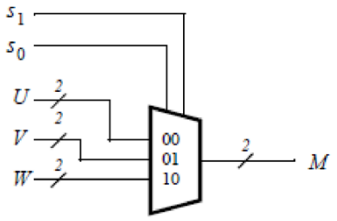



Figure 5. A two-bit wide 3-to-1 multiplexer.

Perform the following steps to implement the two-bit wide 3-to-1 multiplexer.

1. Create a new Quartus II project for your circuit.
2. Create a Verilog module for the two-bit wide 3-to-1 multiplexer. Connect its select inputs to switches $SW_{9-8}$, and use switches $SW_{5-0}$ to provide the three 2-bit inputs $U$ to $W$. Connect the $SW$ switches to the red lights $LEDR$ and connect the output $M$ to the green lights $LEDG_{1-0}$.
3. Include in your project the required pin assignments for the DE1 board. Compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the two-bit wide 3-to-1 multiplexer by toggling the switches and observing the LEDs. Ensure that each of the inputs $U$ to $W$ can be properly selected as the output $M$.

Name of Candidate: _____

**Part IV**

Figure 6 shows a *7-segment* decoder module that has the two-bit input $c_1c_0$. This decoder produces seven outputs that are used to display a character on a 7-segment display. Table 1 lists the characters that should be displayed for each valuation of $c_1c_0$. To keep the design simple, only three characters are included in the table (plus the 'blank' character, which is selected for codes 11).

The seven segments in the display are identified by the indices 0 to 6 shown in the figure. Each segment is illuminated by driving it to the logic value 0. You are to write a Verilog module that implements logic functions that represent circuits needed to activate each of the seven segments. Use only simple Verilog **assign** statements in your code to specify each logic function using a Boolean expression.

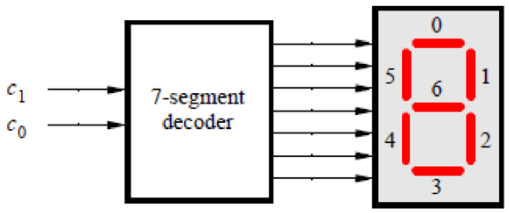



Figure 6. A 7-segment decoder.

| $c_1c_0$ | Character |
|---|---|
| 00 | d |
| 01 | E |
| 10 | 1 |
| 11 | |

Table 1. Character codes.

Perform the following steps:

1. Create a new Quartus II project for your circuit.
2. Create a Verilog module for the 7-segment decoder. Connect the $c_1c_0$ inputs to switches $SW_{1-0}$, and connect the outputs of the decoder to the *HEX0* display on the DE1 board. The segments in this display are called $HEX0_0$, $HEX0_1$, . . ., $HEX0_6$, corresponding to Figure 6. You should declare the 7-bit port

   **output** [0:6] HEX0;

   in your Verilog code so that the names of these outputs match the corresponding names in the *DE1 User Manual* and the *DE1_pin_assignments.qsf* file.
3. After making the required DE1 board pin assignments, compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the $SW_{1-0}$ switches and observing the 7-segment display.

**Part V**

Consider the circuit shown in Figure 7. It uses a two-bit wide 3-to-1 multiplexer to enable the selection of three characters that are displayed on a 7-segment display. Using the 7-segment decoder from Part IV this circuit can display any of the characters D, E, 1, and 'blank'. The character codes are set according to Table 1 by using the switches $SW_{5-0}$, and a specific character is selected for display by setting the switches $SW_{9-8}$.

An outline of the Verilog code that represents this circuit is provided in Figure 8. Note that we have used the circuits from Parts III and IV as subcircuits in this code. You are to extend the code in Figure 8 so that it uses three 7-segment displays rather than just one. You will need to use three instances of each of the subcircuits. The purpose of your circuit is to display any word on the four displays that is composed of the characters in Table 1, and be able to rotate this word in a circular fashion across the displays when the switches $SW_{9-8}$ are toggled. As an example, if the displayed word is dE1, then your circuit should produce the output patterns illustrated in Table 2.

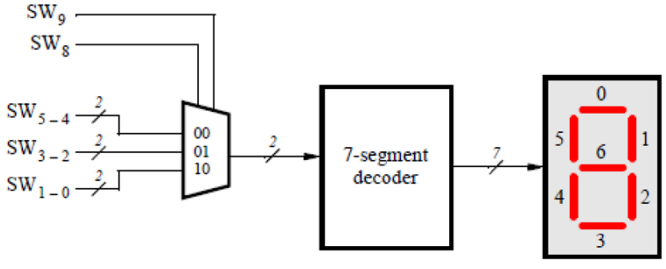



Figure 7. A circuit that can select and display one of three characters.

Name of Candidate: _____

```
module part5 (SW, HEX0);
    input [9:0] SW;          // toggle switches
    output [0:6] HEX0;       // 7-seg displays

    wire [1:0] M;

    mux_2bit_3to1 M0 (SW[9:8], SW[5:4], SW[3:2], SW[1:0], M);
    char_7seg H0 (M, HEX0);
endmodule

// implements a 2-bit wide 3-to-1 multiplexer
module mux_2bit_3to1 (S, U, V, W, M);
    input [1:0] S, U, V, W;
    output [1:0] M;

    ... code not shown

endmodule

// implements a 7-segment decoder for d, E, 1 and 'blank'
module char_7seg (C, Display);
    input [1:0] C;           // input code
    output [0:6] Display;    // output 7-seg code

    ... code not shown

endmodule
```

Figure 8. Verilog code for the circuit in Figure 7.

| $SW_9$ $SW_8$ | Character pattern |
|---|---|
| 00 | d E 1 |
| 01 | E 1 d |
| 10 | 1 d E |

Table 2. Rotating the word dE1 on three displays.

Perform the following steps.

1. Create a new Quartus II project for your circuit.
2. Include your Verilog module in the Quartus II project. Connect the switches $SW_{9-8}$ to the select inputs of each of the three instances of the three-bit wide 3-to-1 multiplexers. Also connect $SW_{5-0}$ to each instance of the multiplexers as required to produce the patterns of characters shown in Table 2. Connect the outputs of the three multiplexers to the 7-segment displays *HEX2*, *HEX1*, and *HEX0*.
3. Include the required pin assignments for the DE1 board for all switches, LEDs, and 7-segment displays. Compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by setting the proper character codes on the switches $SW_{5-0}$ and then toggling $SW_{9-8}$ to observe the rotation of the characters.

Name of Candidate: _____

**Part VI**

Extend your design from Part V so that is uses all four 7-segment displays on the DE1 board. Your circuit should be able to display words with three (or fewer) characters on the four displays, and rotate the displayed word when the switches $SW_{9-8}$ are toggled. If the displayed word is dE1, then your circuit should produce the patterns shown in Table 3.

| $SW_9$ $SW_8$ | Character pattern | | | |
|---|---|---|---|---|
| 00 | | d | E | 1 |
| 01 | d | E | 1 | |
| 10 | E | 1 | | d |
| 11 | 1 | | d | E |

Table 3. Rotating the word dE1 on eight displays.

Perform the following steps:

1. Create a new Quartus II project for your circuit and select the target chip as Cyclone II EP2C20F484C7.
2. Include your Verilog module in the Quartus II project. Connect the switches $SW_{9-8}$ to the select inputs of each instance of the multiplexers in your circuit. Also connect $SW_{5-0}$ to each instance of the multiplexers as required to produce the patterns of characters shown in Table 3. (Hint: for some inputs of the multiplexers you will want to select the 'blank' character.) Connect the outputs of your multiplexers to the 7-segment displays *HEX3*, . . ., *HEX0*.
3. Include the required pin assignments for the DE1 board for all switches, LEDs, and 7-segment displays. Compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by setting the proper character codes on the switches $SW_{5-0}$ and then toggling $SW_{9-8}$ to observe the rotation of the characters.

Name of Candidate: _____

# LAB 2:

## Numbers and Displays

:: WEEK 2 ::
DURATION: 2 HOURS

This is an exercise in designing combinational circuits that can perform binary-to-decimal number conversion and binary-coded-decimal (BCD) addition.

**Part I**

We wish to display on the 7-segment displays *HEX1* to *HEX0* the values set by the switches $SW_{9-0}$. Let the values denoted by $SW_{7-4}$ and $SW_{3-0}$ be displayed on *HEX1* and *HEX0*, respectively. Your circuit should be able to display the digits from 0 to 9, and should treat the valuations 1010 to 1111 as don't-cares.

1. Create a new project which will be used to implement the desired circuit on the Altera DE1 board. The intent of this exercise is to manually derive the logic functions needed for the 7-segment displays. You should use only simple Verilog **assign** statements in your code and specify each logic function as a Boolean expression.

2. Write a Verilog file that provides the necessary functionality. Include this file in your project and assign the pins on the FPGA to connect to the switches and 7-segment displays, as indicated in the User Manual for the DE1 board. The procedure for making pin assignments is described in the tutorial *Quartus II Introduction using Verilog Design*, which is available on the *DE1 System CD* and in the University Program section of Altera's web site.

3. Compile the project and download the compiled circuit into the FPGA chip.

4. Test the functionality of your design by toggling the switches and observing the displays.

**Part II**

You are to design a circuit that converts a four-bit binary number $V = v_3v_2v_1v_0$ into its two-digit decimal equivalent $D = d_1d_0$. Table 1 shows the required output values. A partial design of this circuit is given in Figure 1. It includes a comparator that checks when the value of $V$ is greater than 9, and uses the output of this comparator in the control of the 7-segment displays. You are to complete the design of this circuit by creating a Verilog module which includes the comparator, multiplexers, and circuit *A* (do not include circuit *B* or the 7-segment decoder at this point). Your Verilog module should have the four-bit input $V$, the four-bit output $M$ and the output $z$. The intent of this exercise is to use simple Verilog **assign** statements to specify the required logic functions using Boolean expressions. Your Verilog code should not include any **if-else**, **case**, or similar statements.

| Binary value | Decimal digits | |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 0 | 1 |
| 0010 | 0 | 2 |
| ... | ... | ... |
| 1001 | 0 | 9 |
| 1010 | 1 | 0 |
| 1011 | 1 | 1 |
| 1100 | 1 | 2 |
| 1101 | 1 | 3 |
| 1110 | 1 | 4 |
| 1111 | 1 | 5 |

Table 1. Binary-to-decimal conversion values.

Name of Candidate: _____

Perform the following steps:

1. Make a Quartus II project for your Verilog module.
2. Compile the circuit and use functional simulation to verify the correct operation of your comparator, multiplexers, and circuit *A*.
3. Augment your Verilog code to include circuit *B* in Figure 1 as well as the 7-segment decoder. Change the inputs and outputs of your code to use switches $SW_{3-0}$ on the DE1 board to represent the binary number $V$, and the displays *HEX1* and *HEX0* to show the values of decimal digits $d_1$ and $d_0$. Make sure to include in your project the required pin assignments for the DE1 board.
4. Recompile the project, and then download the circuit into the FPGA chip.
5. Test your circuit by trying all possible values of $V$ and observing the output displays.

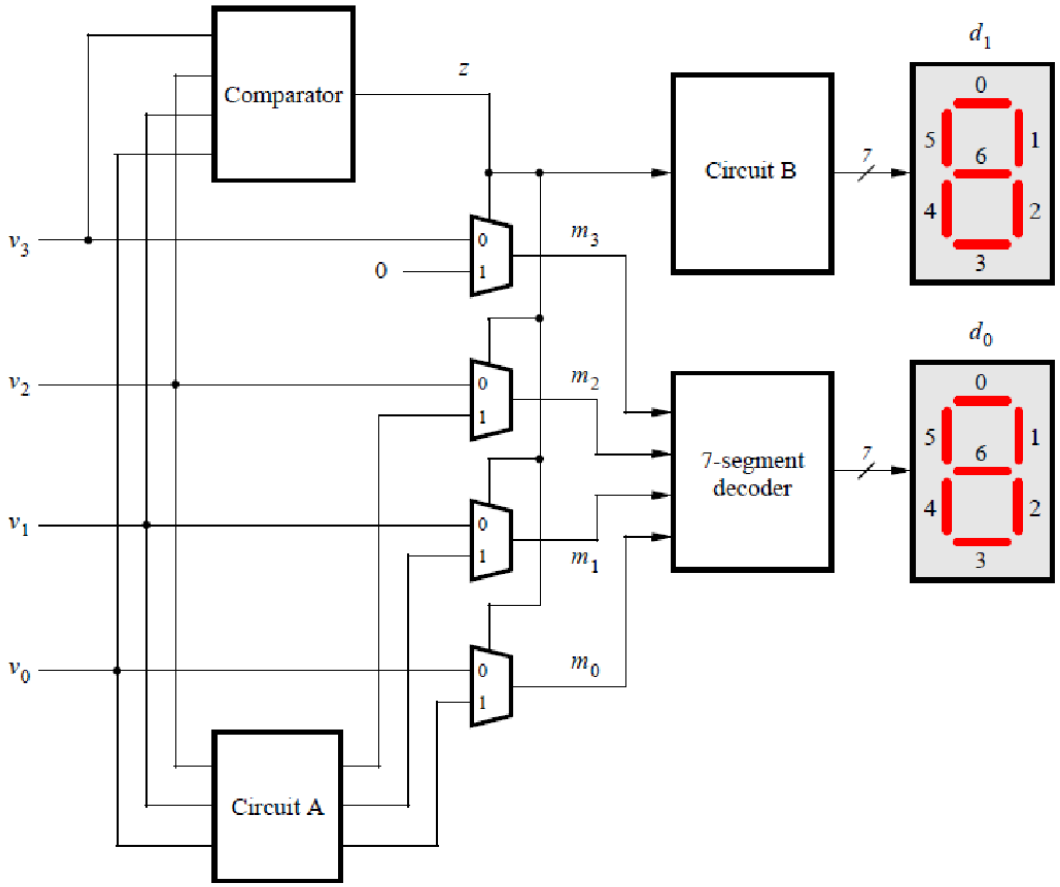
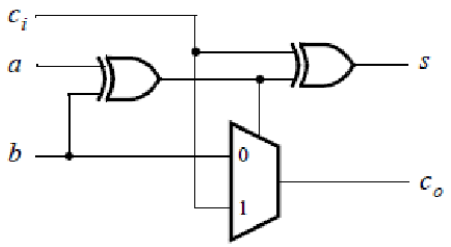



Figure 1: Partial design of the binary-to-decimal conversion circuit.

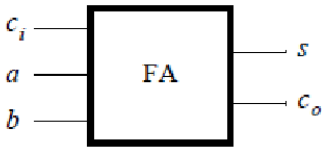
Name of Candidate: _____

**Part III**

Figure 2$a$ shows a circuit for a *full adder*, which has the inputs $a$, $b$, and $c_i$, and produces the outputs $s$ and $c_o$. Parts $b$ and $c$ of the figure show a circuit symbol and truth table for the full adder, which produces the two-bit binary sum $c_o s = a + b + c_i$. Figure 2$d$ shows how four instances of this full adder module can be used to design a circuit that adds two four-bit numbers. This type of circuit is usually called a *ripple-carry* adder, because of the way that the carry signals are passed from one full adder to the next. Write Verilog code that implements this circuit, as described below.



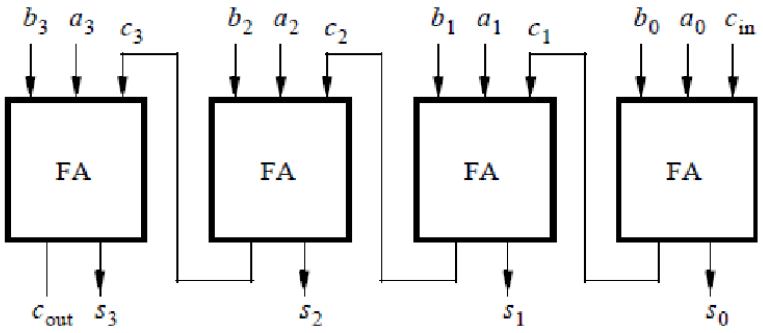

a) Full adder circuit




b) Full adder symbol

| $b$ | $a$ | $c_i$ | $c_o$ | $s$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

c) Full adder truth table




d) Four-bit ripple-carry adder circuit

Figure 2: A ripple-carry adder circuit.

1. Create a new Quartus II project for the adder circuit. Write a Verilog module for the full adder subcircuit and write a top-level Verilog module that instantiates four instances of this full adder.
2. Use switches $SW_{7-4}$ and $SW_{3-0}$ to represent the inputs $A$ and $B$, respectively. Use $SW_8$ for the carry-in $c_{in}$ of the adder. Connect the *SW* switches to their corresponding red lights LEDR, and connect the outputs of the adder, $c_{out}$ and $S$, to the green lights LEDG.
3. Include the necessary pin assignments for the DE1 board, compile the circuit, and download it into the FPGA chip.
4. Test your circuit by trying different values for numbers $A$, $B$, and $c_{in}$.

Name of Candidate: _____

**Part IV**

In part II we discussed the conversion of binary numbers into decimal digits. It is sometimes useful to build circuits that use this method of representing decimal numbers, in which each decimal digit is represented using four bits. This scheme is known as the *binary coded decimal* (BCD) representation. As an example, the decimal value 59 is encoded in BCD form as 0101 1001.

You are to design a circuit that adds two BCD digits. The inputs to the circuit are BCD numbers $A$ and $B$, plus a carry-in, $c_{in}$. The output should be a two-digit BCD sum $S_1S_0$. Note that the largest sum that needs to be handled by this circuit is $S_1S_0 = 9 + 9 + 1 = 19$. Perform the steps given below.

1. Create a new Quartus II project for your BCD adder. You should use the four-bit adder circuit from part III to produce a four-bit sum and carry-out for the operation $A + B$. A circuit that converts this five-bit result, which has the maximum value 19, into two BCD digits $S_1S_0$ can be designed in a very similar way as the binary-to-decimal converter from part II. Write your Verilog code using simple **assign** statements to specify the required logic functions–do not use other types of Verilog statements such as **if-else** or **case** statements for this part of the exercise.

2. Use switches $SW_{7-4}$ and $SW_{3-0}$ for the inputs $A$ and $B$, respectively, and use $SW_8$ for the carry-in. Connect the *SW* switches to their corresponding red lights LEDR, and connect the four-bit sum and carry-out produced by the operation $A + B$ to the green lights LEDG. Display the BCD values of $A$ and $B$ on the 7-segment displays *HEX3* and *HEX2*, and display the result $S_1S_0$ on *HEX1* and *HEX0*.

3. Since your circuit handles only BCD digits, check for the cases when the input $A$ or $B$ is greater than nine. If this occurs, indicate an error by turning on the green light $LEDG_7$.

4. Include the necessary pin assignments for the DE1 board, compile the circuit, and download it into the FPGA chip.

5. Test your circuit by trying different values for numbers $A$, $B$, and $c_{in}$.

**Part V**

In part IV you created Verilog code for a BCD adder. A different approach for describing the adder in Verilog code is to specify an algorithm like the one represented by the following pseudo-code:

```
1   T0 = A + B + c0
2   if (T0 > 9) then
3       Z0 = 10;
4       c1 = 1;
5   else
6       Z0 = 0;
7       c1 = 0;
8   end if
9   S0 = T0 − Z0
10  S1 = c1
```

It is reasonably straightforward to see what circuit could be used to implement this pseudo-code. Lines 1 and 9 represent adders, lines 2-8 correspond to multiplexers, and testing for the condition $T_0 > 9$ requires comparators. You are to write Verilog code that corresponds to this pseudo-code. Note that you can perform addition operations in your Verilog code instead of the subtractions shown in line 9. The intent of this part of the exercise is to examine the effects of relying more on the Verilog compiler to design the circuit by using **if-else** statements along with the Verilog $>$ and $+$ operators. Perform the following steps:

1. Create a new Quartus II project for your Verilog code. Use switches $SW_{7-4}$ and $SW_{3-0}$ for the inputs $A$ and $B$, respectively, and use $SW_8$ for the carry-in. The value of $A$ should be displayed on the 7-segment displays *HEX3*, while $B$ should be on *HEX2*. Display the BCD sum, $S_1S_0$, on the 7-segment displays *HEX1* and *HEX0*.

2. Use the Quartus II RTL Viewer tool to examine the circuit produced by compiling your Verilog code. Compare the circuit to the one you designed in Part IV.

3. Download your circuit onto the DE1 board and test it by trying different values for numbers $A$ and $B$.

## Part VI

Name of Candidate: _____

Design a combinational circuit that converts a 6-bit binary number into a 2-digit decimal number represented in the BCD form. Use switches $SW_{5-0}$ to input the binary number and 7-segment displays *HEX1* and *HEX0* to display the decimal number. Implement your circuit on the DE1 board and demonstrate its functionality.

Name of Candidate: _____

# LAB 2B:

## 74F85: HDL BASED TEST BENCH PRACTICE

:: WEEK 2 ::
DURATION: 2 HOURS

1. Design a HDL based 4-bit magnitude comparator (74F85) based on the provided datasheet.
2. List a verification plan for your design.
3. Design a HDL based test bench to verify the design in Model SIM.
4. Show your progress at the end of two hours.

### 4-bit magnitude comparator — 74F85

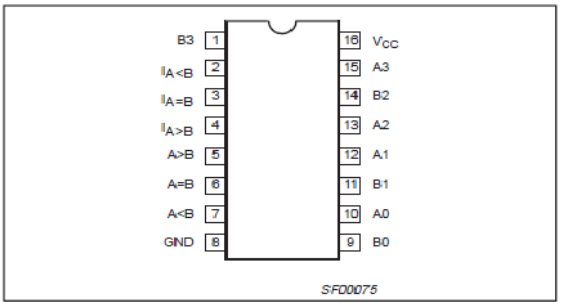
#### FEATURES

- High-impedance NPN base inputs for reduced loading (20µA in High and Low states)
- Magnitude comparison of any binary words
- Serial or parallel expansion without extra gating

#### DESCRIPTION

The 74F85 is a 4-bit magnitude comparator that can be expanded to almost any length. It compares two 4-bit binary, BCD, or other monotonic codes and presents the three possible magnitude results at the outputs. The 4-bit inputs are weighted (A0–A3) and (B0–B3) where A3 and B3 are the most significant bits. The operation of the 74F85 is described in the Function Table, showing all possible logic conditions. The upper part of the table describes the normal operation under all conditions that will occur in a single device or in a series expansion scheme. In the upper part of the table the three outputs are mutually exclusive. In the lower part of the table, the outputs reflect the feed-forward conditions that exist in the parallel expansion scheme. The expansion inputs $I_{A>B}$, and $I_{A=B}$ and $I_{A<B}$ are the least significant bit positions. When used for series expansion, the A>B, A=B and A<B outputs of the lease significant word are connected to the corresponding $I_{A>B}$, $I_{A=B}$ and $I_{A<B}$ inputs of the next higher stage. Stages can be added in this manner to any length, but a propagation delay penalty of about 15ns is added with each additional stage. For proper operation, the expansion inputs of the least significant word should be tied as follows: $I_{A>B}$ = Low, $I_{A=B}$ = High, and $I_{A<B}$ = Low.

#### PIN CONFIGURATION




| TYPE | TYPICAL PROPAGATION DELAY | TYPICAL SUPPLY CURRENT (TOTAL) |
|---|---|---|
| 74F85 | 7.0ns | 40mA |

#### ORDERING INFORMATION

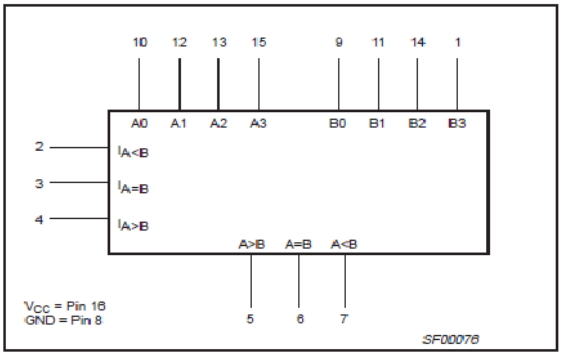
| DESCRIPTION | COMMERCIAL RANGE $V_{CC}$ = 5V ±10%, $T_{amb}$ = 0°C to +70°C | PKG DWG # |
|---|---|---|
| 16-pin plastic DIP | N74F85N | SOT38-4 |
| 16-pin plastic SO | N74F85D | SOT162-1 |

#### INPUT AND OUTPUT LOADING AND FAN OUT TABLE

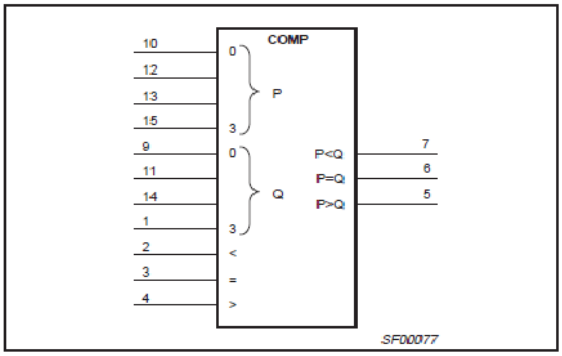
| PINS | DESCRIPTION | 74F (U.L.) HIGH/LOW | LOAD VALUE HIGH/LOW |
|---|---|---|---|
| A0–A3 | Comparing inputs | 1.0/0.033 | 20µA/20µA |
| B0–B3 | Comparing inputs | 1.0/0.033 | 20µA/20µA |
| $I_{A<B}$, $I_{A=B}$, $I_{A>B}$ | Expansion inputs (active High) | 1.0/0.033 | 20µA/20µA |
| A<B, A=B, A>B | Data outputs (active High) | 50/33 | 1.0mA/20mA |

NOTE: One (1.0) FAST unit load is defined as: 20µA in the High state and 0.6mA in the Low state.

#### LOGIC SYMBOL



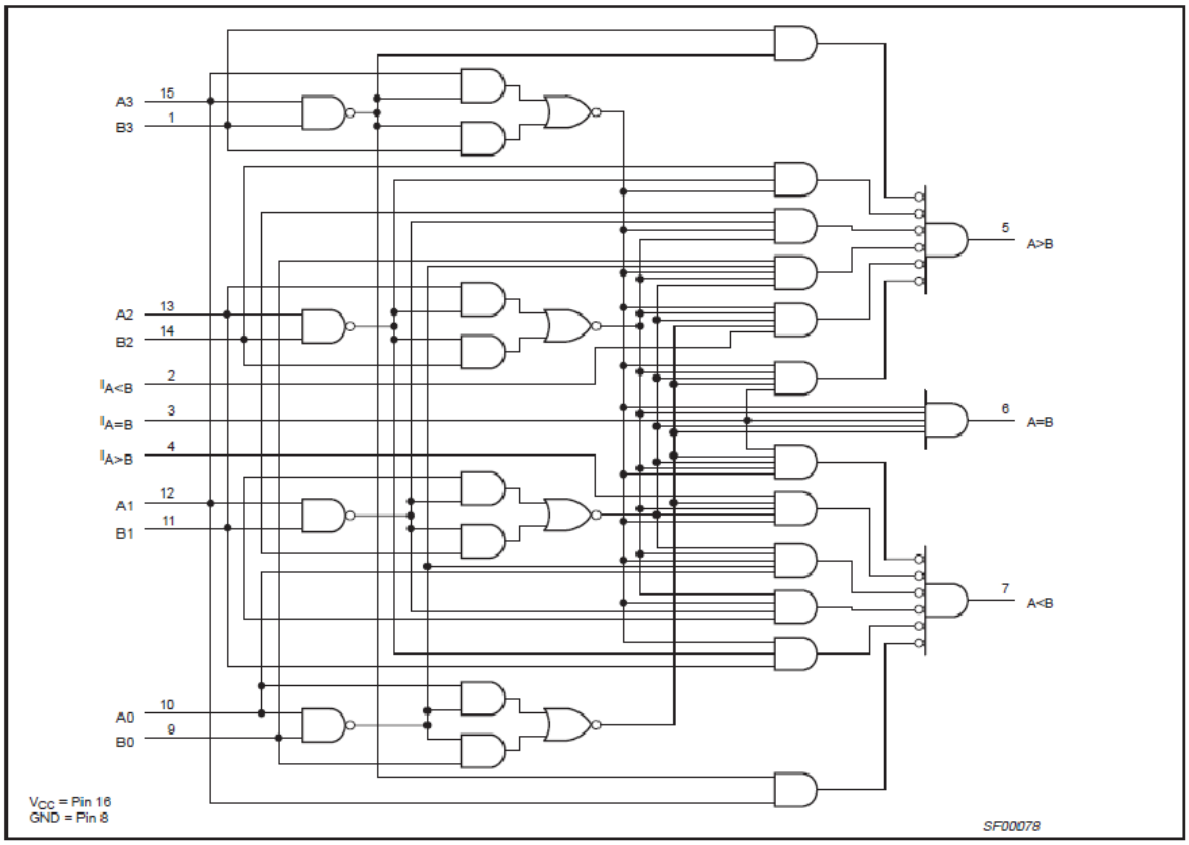

#### IEC/IEEE SYMBOL



Name of Candidate: _____



## LOGIC DIAGRAM




## FUNCTION TABLE

| COMPARING INPUTS | | | | EXPANSION INPUTS | | | OUTPUTS | | |
|---|---|---|---|---|---|---|---|---|---|
| A3,B3 | A2,B2 | A1,B1 | A0,B0 | $I_{A>B}$ | $I_{A<B}$ | $I_{A=B}$ | A>B | A<B | A=B |
| A3>B3 | X | X | X | X | X | X | H | L | L |
| A3<B3 | X | X | X | X | X | X | L | H | L |
| A3=B3 | A2>B2 | X | X | X | X | X | H | L | L |
| A3=B3 | A2<B2 | X | X | X | X | X | L | H | L |
| A3=B3 | A2=B2 | A1>B1 | X | X | X | X | H | L | L |
| A3=B3 | A2=B2 | A1<B1 | X | X | X | X | L | H | L |
| A3=B3 | A2=B2 | A1=B1 | A0>B0 | X | X | X | H | L | L |
| A3=B3 | A2=B2 | A1=B1 | A0<B0 | X | X | X | L | H | L |
| A3=B3 | A2=B2 | A1=B1 | A0=B0 | H | L | L | H | L | L |
| A3=B3 | A2=B2 | A1=B1 | A0=B0 | L | H | L | L | H | L |
| A3=B3 | A2=B2 | A1=B1 | A0=B0 | L | L | H | L | L | H |
| A3=B3 | A2=B2 | A1=B1 | A0=B0 | X | X | H | L | L | H |
| A3=B3 | A2=B2 | A1=B1 | A0=B0 | H | H | L | L | L | L |
| A3=B3 | A2=B2 | A1=B1 | A0=B0 | L | L | L | H | H | L |

H = High voltage level
L = Low voltage level
X = Don't care

Name of Candidate: _____



## APPLICATION

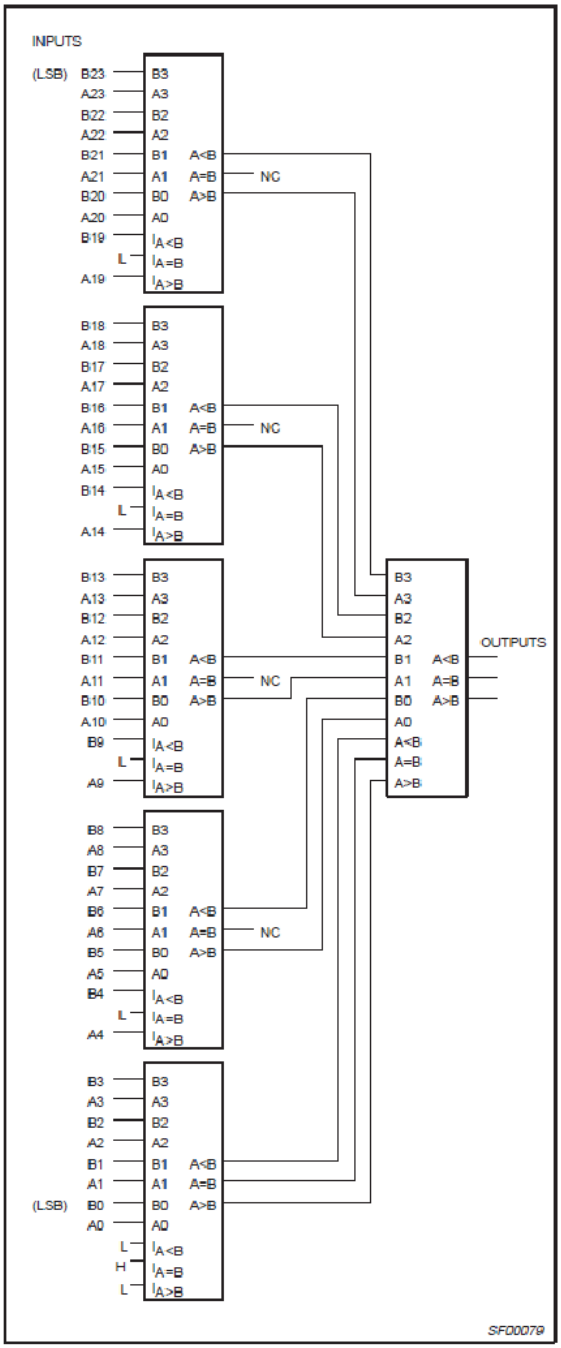



Figure 1. Comparison of Two 24-Bit Words

The parallel expansion scheme shown in Figure 1 demonstrates the most efficient general use of these comparators. The expansion inputs can be used as a fifth input bit position except on the least significant device, which must be connected as in the serial scheme. The expansion inputs used by labeling $I_{A>B}$ as an "A" input, $I_{A<B}$ as a "B" input and setting $I_{A=B}$ = Low. The 74F85 can be used as a 5-bit comparator only when the outputs are used to drive the (A0–A3) and (B0–B3) inputs of another 74F85 device. The parallel technique can be expanded to any number of bits as shown in Table 1.

**Table 1.**

| WORD LENGTH | NUMBER OF PACKAGES | TYPICAL SPEEDS 74F |
|---|---|---|
| 1–4 bits | 1 | 12ns |
| 5–24 bits | 2–6 | 22ns |
| 25–120 bits | 8–31 | 34ns |

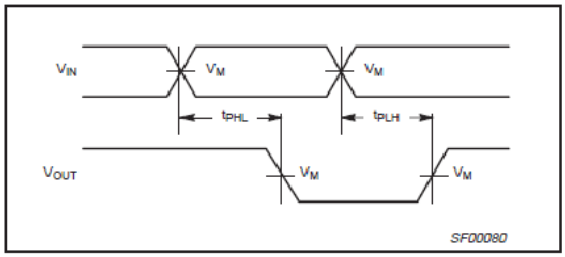
Name of Candidate: _____



## AC ELECTRICAL CHARACTERISTICS

| SYMBOL | PARAMETER | TEST CONDITION | LIMITS | | | | | UNIT |
|---|---|---|---|---|---|---|---|---|
| | | | $V_{CC}$ = +5.0V<br>$T_{amb}$ = +25°C<br>$C_L$ = 50pF, $R_L$ = 500Ω | | | $V_{CC}$ = +5.0V ± 10%<br>$T_{amb}$ = 0°C to +70°C<br>$C_L$ = 50pF, $R_L$ = 500Ω | | |
| | | | MIN | TYP | MAX | MIN | MAX | |
| $t_{PLH}$<br>$t_{PHL}$ | Propagation delay<br>A or B to A<B, A>B | Waveform 1<br>3 logic levels | 6.0<br>7.0 | 8.5<br>9.5 | 11.0<br>14.0 | 5.5<br>6.5 | 13.0<br>15.5 | ns |
| $t_{PLH}$<br>$t_{PHL}$ | Propagation delay<br>A or B to A=B | Waveform 1<br>4 logic levels | 6.5<br>7.0 | 9.0<br>9.5 | 11.5<br>14.0 | 6.0<br>6.5 | 14.0<br>14.5 | ns |
| $t_{PLH}$<br>$t_{PHL}$ | Propagation delay<br>$I_{A<B}$ and $I_{A=B}$ to A>B | Waveform 1<br>1 logic level | 3.0<br>3.0 | 5.0<br>6.0 | 7.5<br>9.0 | 2.5<br>2.5 | 9.0<br>10.0 | ns |
| $t_{PLH}$<br>$t_{PHL}$ | Propagation delay<br>$I_{A=B}$ to A=B | Waveform 1<br>2 logic levels | 2.5<br>3.5 | 4.5<br>7.5 | 7.0<br>10.0 | 2.0<br>2.5 | 9.0<br>12.0 | ns |
| $t_{PLH}$<br>$t_{PHL}$ | Propagation delay<br>$I_{A>B}$ and $I_{A=B}$ to A<B | Waveform 1<br>1 logic level | 3.0<br>3.0 | 5.0<br>6.0 | 8.0<br>9.0 | 3.0<br>2.0 | 9.5<br>9.5 | ns |

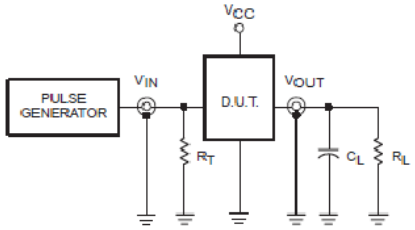
## AC WAVEFORMS



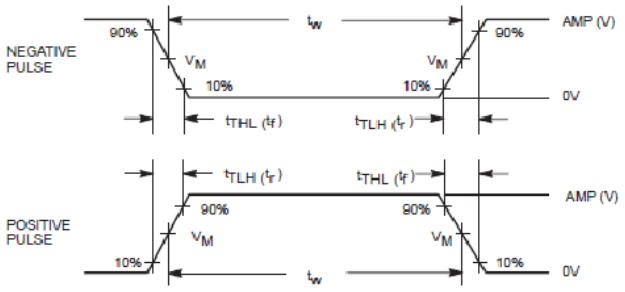

**Waveform 1. Propagation Delay Input to Output**

**NOTE:**
For all waveforms, $V_M$ = 1.5V.

## TEST CIRCUIT AND WAVEFORMS




Test Circuit for Totem-Pole Outputs




**DEFINITIONS:**

$R_L$ = Load resistor; see AC ELECTRICAL CHARACTERISTICS for value.
$C_L$ = Load capacitance includes jig and probe capacitance; see AC ELECTRICAL CHARACTERISTICS for value.
$R_T$ = Termination resistance should be equal to $Z_{OUT}$ of pulse generators.

**Input Pulse Definition**

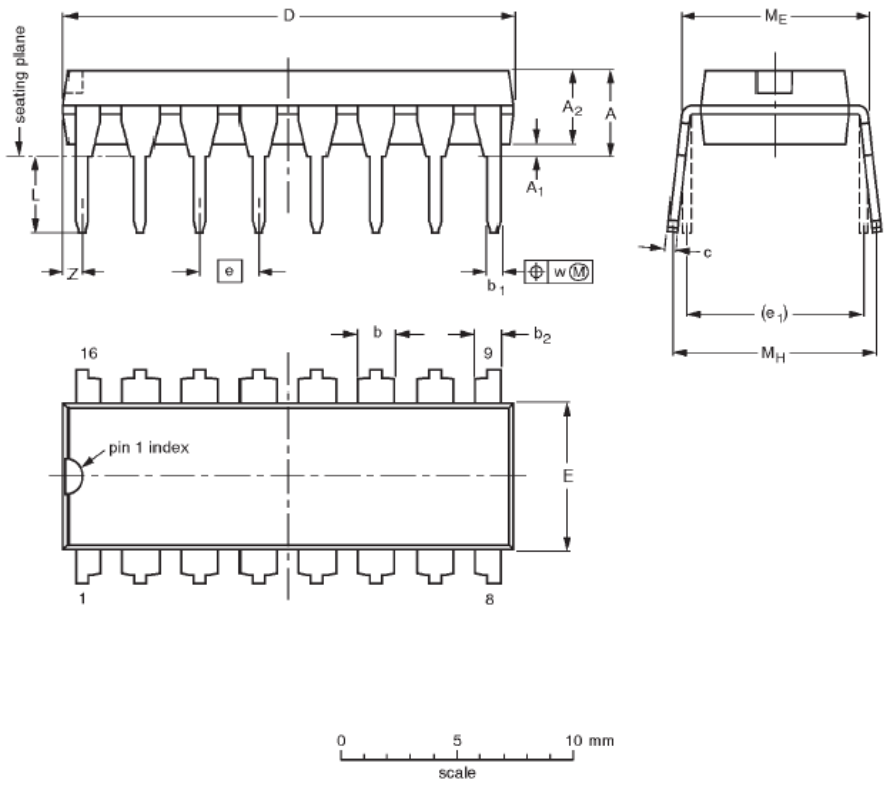
| family | INPUT PULSE REQUIREMENTS | | | | | |
|---|---|---|---|---|---|---|
| | amplitude | $V_M$ | rep. rate | $t_w$ | $t_{TLH}$ | $t_{THL}$ |
| 74F | 3.0V | 1.5V | 1MHz | 500ns | 2.5ns | 2.5ns |

SF00008

Name of Candidate: _____



**DIP16: plastic dual in-line package; 16 leads (300 mil)** **SOT38-4**




**DIMENSIONS (inch dimensions are derived from the original mm dimensions)**

| UNIT | A max. | $A_1$ min. | $A_2$ max. | b | $b_1$ | $b_2$ | c | D (1) | E (1) | e | $e_1$ | L | $M_E$ | $M_H$ | w | Z (1) max. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| mm | 4.2 | 0.51 | 3.2 | 1.73<br>1.30 | 0.53<br>0.38 | 1.25<br>0.85 | 0.36<br>0.23 | 19.50<br>18.55 | 6.48<br>6.20 | 2.54 | 7.62 | 3.60<br>3.05 | 8.25<br>7.80 | 10.0<br>8.3 | 0.254 | 0.76 |
| inches | 0.17 | 0.020 | 0.13 | 0.068<br>0.051 | 0.021<br>0.015 | 0.049<br>0.033 | 0.014<br>0.009 | 0.77<br>0.73 | 0.26<br>0.24 | 0.10 | 0.30 | 0.14<br>0.12 | 0.32<br>0.31 | 0.39<br>0.33 | 0.01 | 0.030 |

**Note**

1. Plastic or metal protrusions of 0.25 mm maximum per side are not included.

| OUTLINE VERSION | REFERENCES | | | | EUROPEAN PROJECTION | ISSUE DATE |
|---|---|---|---|---|---|---|
| | IEC | JEDEC | EIAJ | | | |
| SOT38-4 | | | | | | ~~92-11-17~~<br>95-01-14 |

Name of Candidate: _____



## Data sheet status

| Data sheet status | Product status | Definition [1] |
|---|---|---|
| Objective specification | Development | This data sheet contains the design target or goal specifications for product development. Specification may change in any manner without notice. |
| Preliminary specification | Qualification | This data sheet contains preliminary data, and supplementary data will be published at a later date. Philips Semiconductors reserves the right to make chages at any time without notice in order to improve design and supply the best possible product. |
| Product specification | Production | This data sheet contains final specifications. Philips Semiconductors reserves the right to make changes at any time without notice in order to improve design and supply the best possible product. |

[1] Please consult the most recently issued datasheet before initiating or completing a design.

## Definitions

**Short-form specification** — The data in a short-form specification is extracted from a full data sheet with the same type number and title. For detailed information see the relevant data sheet or data handbook.

**Limiting values definition** — Limiting values given are in accordance with the Absolute Maximum Rating System (IEC 134). Stress above one or more of the limiting values may cause permanent damage to the device. These are stress ratings only and operation of the device at these or at any other conditions above those given in the Characteristics sections of the specification is not implied. Exposure to limiting values for extended periods may affect device reliability.

**Application information** — Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors make no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

## Disclaimers

**Life support** — These products are not designed for use in life support appliances, devices or systems where malfunction of these products can reasonably be expected to result in personal injury. Philips Semiconductors customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors for any damages resulting from such application.

**Right to make changes** — Philips Semiconductors reserves the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified.

**Philips Semiconductors**
811 East Arques Avenue
P.O. Box 3409
Sunnyvale, California 94088–3409
Telephone 800-234-7381



print code Date of release: 10-98
Document order number: 9397-750-05067

*Let's make things better.*

**Philips Semiconductors**

PHILIPS

# LAB 3:

Name of Candidate: _____

# Latches, Flip-Flops and Registers

:: WEEK 3 ::
DURATION: 2 HOURS

The purpose of this exercise is to investigate latches, flip-flops, and registers.

**Part I**

Altera FPGAs include flip-flops that are available for implementing a user's circuit. We will show how to make use of these flip-flops in Part IV of this exercise. But first we will show how storage elements can be created in an FPGA without using its dedicated flip-flops.

Figure 1 depicts a gated RS latch circuit. Two styles of Verilog code that can be used to describe this circuit are given in Figure 2. Part *a* of the figure specifies the latch by instantiating logic gates, and part *b* uses logic expressions to create the same circuit. If this latch is implemented in an FPGA that has 4-input lookup tables (LUTs), then only one lookup table is needed, as shown in Figure 3*a*.

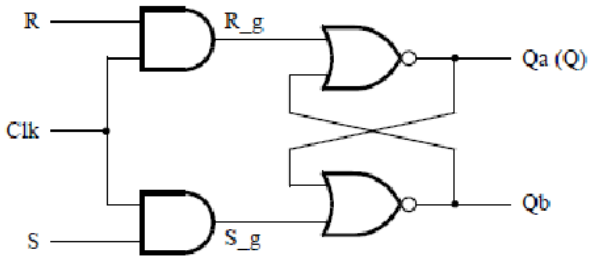



Figure 1: A gated RS latch circuit.

```
// A gated RS latch
module part1 (Clk, R, S, Q);
    input Clk, R, S;
    output Q;

    wire R_g, S_g, Qa, Qb /* synthesis keep */ ;

    and (R_g, R, Clk);
    and (S_g, S, Clk);
    nor (Qa, R_g, Qb);
    nor (Qb, S_g, Qa);

    assign Q = Qa;

endmodule
```

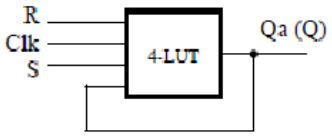
Figure 2*a*. Instantiating logic gates for the RS latch.

Name of Candidate: _____

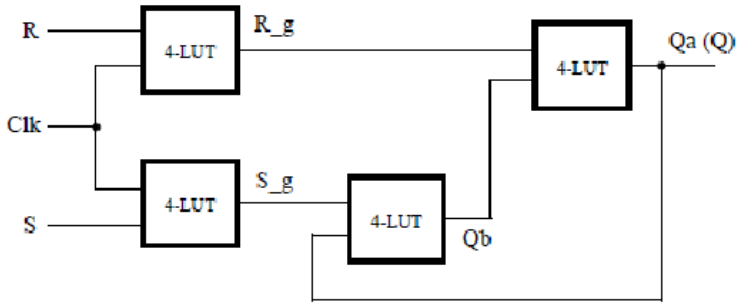
```
// A gated RS latch
module part1 (Clk, R, S, Q);
    input Clk, R, S;
    output Q;

    wire R_g, S_g, Qa, Qb /* synthesis keep */ ;

    assign R_g = R & Clk;
    assign S_g = S & Clk;
    assign Qa = ~(R_g | Qb);
    assign Qb = ~(S_g | Qa);

    assign Q = Qa;

endmodule
```

Figure 2*b*. Specifying the RS latch by using logic expressions.

Although the latch can be correctly realized in one 4-input LUT, this implementation does not allow its internal signals, such as $R\_g$ and $S\_g$, to be observed, because they are not provided as outputs from the LUT. To preserve these internal signals in the implemented circuit, it is necessary to include a *compiler directive* in the code. In Figure 2 the directive /* synthesis keep */ is included to instruct the Quartus II compiler to use separate logic elements for each of the signals $R\_g$, $S\_g$, $Qa$, and $Qb$. Compiling the code produces the circuit with four 4-LUTs depicted in Figure 3*b*.




(a) Using one 4-input lookup table for the RS latch.




(b) Using four 4-input lookup tables for the RS latch.

Figure 3. Implementation of the RS latch from Figure 1.

Create a Quartus II project for the RS latch circuit as follows:

Name of Candidate: _____

1. Create a new project for the RS latch. Select as the target chip the Cyclone II EP2C20F484C7, which is the FPGA chip on the Altera DE1 board.

2. Generate a Verilog file with the code in either part *a* or *b* of Figure 2 (both versions of the code should produce the same circuit) and include it in the project.

3. Compile the code. Use the Quartus II RTL Viewer tool to examine the gate-level circuit produced from the code, and use the Technology Viewer tool to verify that the latch is implemented as shown in Figure 3*b*.

4. In QSim, create a Vector Waveform File (.vwf) which specifies the inputs and outputs of the circuit. Draw waveforms for the $R$ and $S$ inputs and use QSim to produce the corresponding waveforms for $R\_g$, $S\_g$, $Qa$, and $Qb$. Verify that the latch works as expected using both functional and timing simulation.

**Part II**

Figure 4 shows the circuit for a gated D latch.

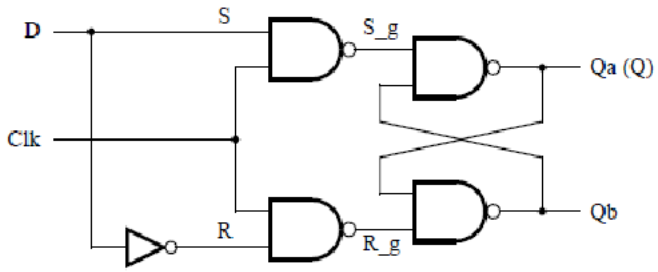



Figure 4. Circuit for a gated D latch.

Perform the following steps:

1. Create a new Quartus II project. Generate a Verilog file using the style of code in Figure 2*b* for the gated D latch. Use the /* synthesis keep */ directive to ensure that separate logic elements are used to implement the signals $R, S\_g, R\_g, Qa$, and $Qb$.

2. Select as the target chip the Cyclone II EP2C20F484C7 and compile the code. Use the Technology Viewer tool to examine the implemented circuit.

3. Verify that the latch works properly for all input conditions by using functional simulation. Examine the timing characteristics of the circuit by using timing simulation.

4. Create a new Quartus II project which will be used for implementation of the gated D latch on the DE1 board. This project should consist of a top-level module that contains the appropriate input and output ports (pins) for the DE1 board. Instantiate your latch in this top-level module. Use switch $SW_0$ to drive the $D$ input of the latch, and use $SW_1$ as the $Clk$ input. Connect the Q output to $LEDR_0$.

5. Recompile your project and download the compiled circuit onto the DE1 board.

6. Test the functionality of your circuit by toggling the $D$ and $Clk$ switches and observing the Q output.

Name of Candidate: _____

**Part III**

Figure 5 shows the circuit for a master-slave D flip-flop.

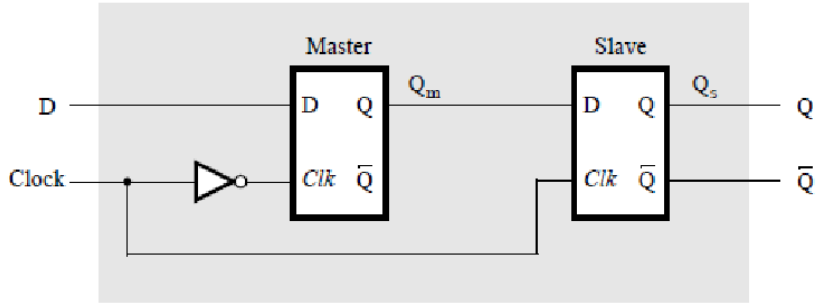



Figure 5. Circuit for a master-slave D flip-flop.

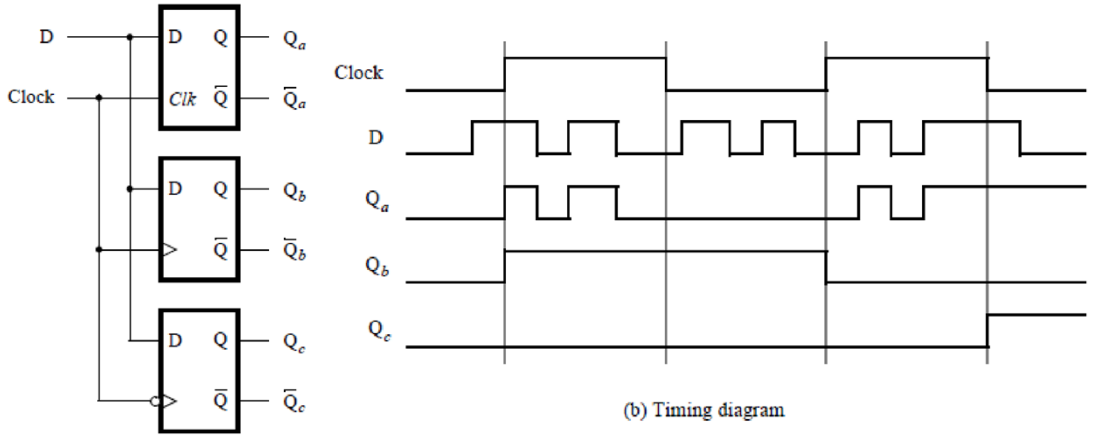
Perform the following:

1. Create a new Quartus II project. Generate a Verilog file that instantiates two copies of your gated D latch module from Part II to implement the master-slave flip-flop.
2. Include in your project the appropriate input and output ports for the Altera DE1 board. Use switch $SW_0$ to drive the D input of the flip-flop, and use $SW_1$ as the *Clock* input. Connect the Q output to $LEDR_0$.
3. Compile your project.
4. Use the Technology Viewer to examine the D flip-flop circuit, and use simulation to verify its correct operation.
5. Download the circuit onto the DE1 board and test its functionality by toggling the *D* and *Clock* switches and observing the Q output.

Name of Candidate: _____

**Part IV**

Figure 6 shows a circuit with three different storage elements: a gated D latch, a positive-edge triggered D flip-flop, and a negative-edge triggered D flip-flop.




(a) Circuit

(b) Timing diagram

Figure 6. Circuit and waveforms for Part IV.

```
module D_latch  (D, Clk, Q);
   input D, Clk;
   output reg Q;

   always @ (D, Clk)
      if (Clk)
         Q = D;
endmodule
```

Figure 7. A behavioral style of Verilog code that specifies a gated D latch.

**Part V**

We wish to display the hexadecimal value of a 8-bit number $A$ on the two 7-segment displays, $HEX3-2$. We also wish to display the hex value of a 8-bit number $B$ on the two 7-segment displays, $HEX1-0$. The values of $A$ and $B$ are inputs to the circuit which are provided by means of switches $SW_{7-0}$. This is to be done by first setting the switches to the value of $A$ and then setting the switches to the value of $B$; therefore, the value of $A$ must be stored in the circuit.

1. Create a new Quartus II project which will be used to implement the desired circuit on the Altera DE1 board.
2. Write a Verilog file that provides the necessary functionality. Use $KEY_0$ as an active-low asynchronous reset, and use $KEY_1$ as a clock input.
3. Include the Verilog file in your project and compile the circuit.
4. Assign the pins on the FPGA to connect to the switches and 7-segment displays, as indicated in the User Manual for the DE1 board.
5. Recompile the circuit and download it into the FPGA chip.
6. Test the functionality of your design by toggling the switches and observing the output displays.

Name of Candidate: _____

# LAB 3B:

## DEDICATED DATAPATHS

:: WEEK 3::
DURATION: 2 HOURS

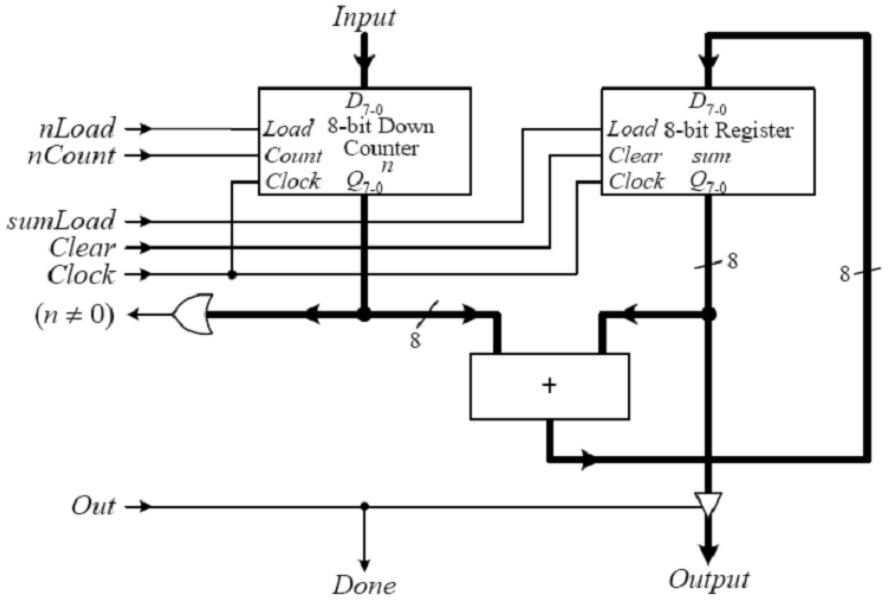
Design a dedicated datapath and control word table to perform the summation from n down to 1 as shown in the following algorithm.

```
1          sum = 0
2          INPUT n
3          WHILE (n ≠ 0){
4             sum = sum + n
5             n = n - 1
6          }
7          OUTPUT sum
```

Fig. 3B: Summation Algorithm

Task 3B:

1. Derive the control word table.
2. Sketch the required hardware to perform the algorithm shown in Fig. 3B.
3. Design the HDL based dedicated datapath in using Altera's Quartus II software.
4. Design a HDL based test bench to verify your design by simulating with appropriate control words. **Lecturer's Check Pont**
5. Implement your design on the FPGA. **Lecturer's Check Pont**




| Control Word | Instruction | *nLoad* | *nCount* | *sumLoad* | *Clear* | *Out* |
|---|---|---|---|---|---|---|
| 1 | *sum* = 0 | 0 | 0 | 0 | 1 | 0 |
| 2 | INPUT *n* | 1 | 0 | 0 | 0 | 0 |
| 3 | *sum* = *sum* + *n* | 0 | 0 | 1 | 0 | 0 |
| 4 | *n* = *n* – 1 | 0 | 1 | 0 | 0 | 0 |
| 5 | OUTPUT *sum* | 0 | 0 | 0 | 0 | 1 |

# LAB 3C:

Name of Candidate: _____

# GENERAL DATAPATH

DURATION: 2 HOURS

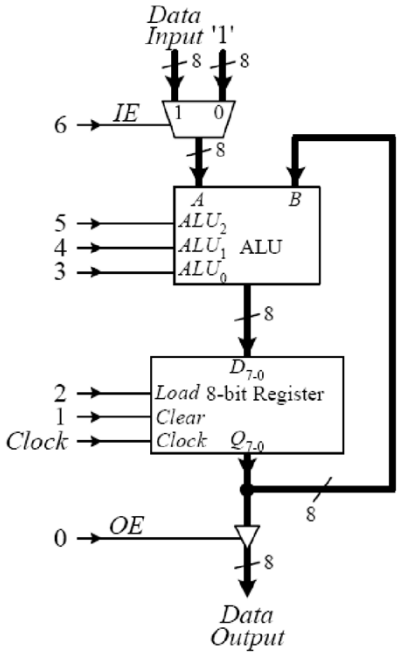



**Figure 9.27** A simple, general datapath circuit.

| $ALU_2$ | $ALU_1$ | $ALU_0$ | Operation |
|---|---|---|---|
| 0 | 0 | 0 | Pass through $A$ |
| 0 | 0 | 1 | $A$ AND $B$ |
| 0 | 1 | 0 | $A$ OR $B$ |
| 0 | 1 | 1 | NOT $A$ |
| 1 | 0 | 0 | $A + B$ |
| 1 | 0 | 1 | $A - B$ |
| 1 | 1 | 0 | $A + 1$ |
| 1 | 1 | 1 | $A - 1$ |

1. Construct a simple general datapath design as shown in Figure 9.27.
2. Test your general datapath by first deriving the control word table from the algorithm as shown below:-

```
1          i = 0
2          WHILE (i ≠ 10){
3             i = i + 1
4             OUTPUT i
5             }
```

You should obtain the following simulation trace on your waveform using a HDL based test bench.

Name of Candidate: _____

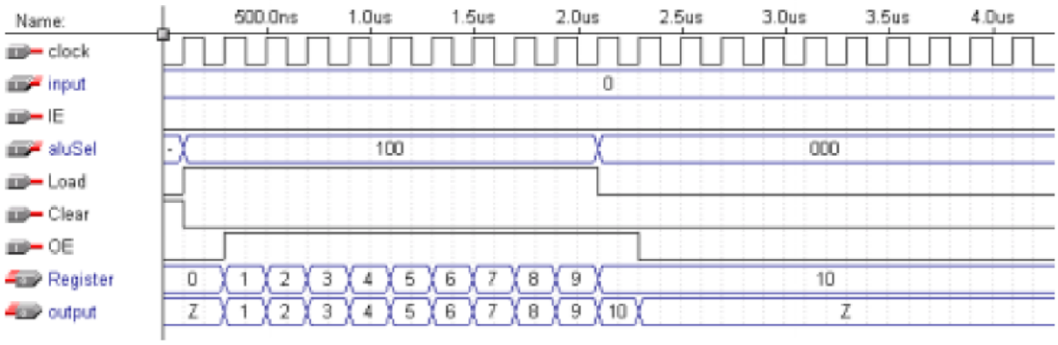



**Figure 9.41** Corrected simulation trace for using the three control words from Figure 9.40.

3. Implement on your FPGA.

SOLUTION 3C:

| Control Word | Instruction | *IE* 6 | $ALU_2\,ALU_1\,ALU_0$ 5–3 | *Load* 2 | *Clear* 1 | *OE* 0 |
|---|---|---|---|---|---|---|
| 1 | $i = 0$ | × | ××× | 0 | 1 | 0 |
| 2 | $i = i + 1$ | 0 | 100 (add) | 1 | 0 | 0 |
| 3 | OUTPUT $i$ | × | ××× | 0 | 0 | 1 |

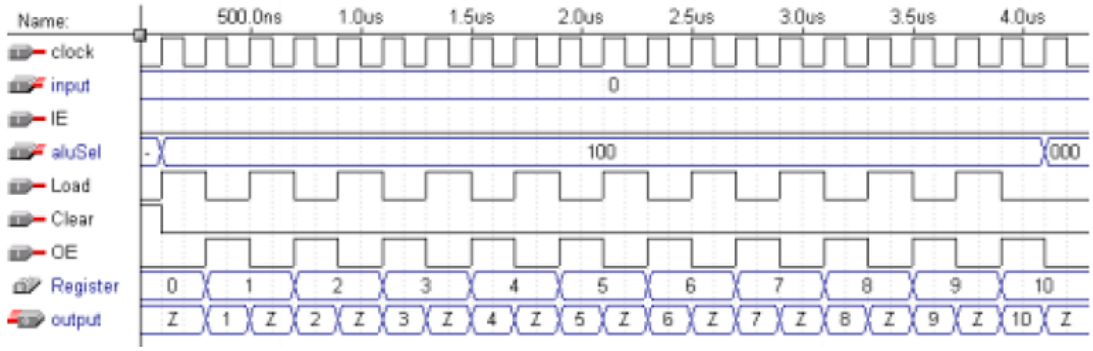



**Figure 9.30** Simulation trace for Example 9.8 using the three control words shown in Figure 9.28(b).

| Control Word | Instruction | *IE* 6 | $ALU_2\,ALU_1\,ALU_0$ 5–3 | *Load* 2 | *Clear* 1 | *OE* 0 |
|---|---|---|---|---|---|---|
| 1 | $i = 0$ | × | ××× | 0 | 1 | 0 |
| 2 | $i = i + 1$ & OUTPUT $i$ | 0 | 100 (add) | 1 | 0 | 1 |

**Figure 9.38** Counting algorithm using two control words for the datapath in Figure 9.27(a).

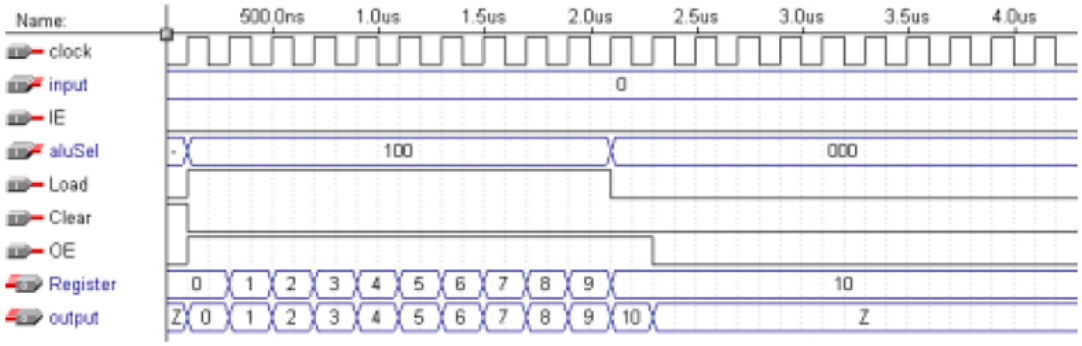



**Figure 9.39** Simulation trace for using the two control words from Figure 9.38.

Name of Candidate: _____

| Control Word | Instruction | *IE* 6 | $ALU_2\,ALU_1\,ALU_0$ 5–3 | *Load* 2 | *Clear* 1 | *OE* 0 |
|---|---|---|---|---|---|---|
| 1 | $i = 0$ | × | ××× | 0 | 1 | 0 |
| 2 | $i = i + 1$ | 0 | 100 (add) | 1 | 0 | 0 |
| 3 | $i = i + 1$ & OUTPUT $i$ | 0 | 100 (add) | 1 | 0 | 1 |

**Figure 9.40**. Optimized control words for the counting algorithm using the datapath in Figure 9.27(a).

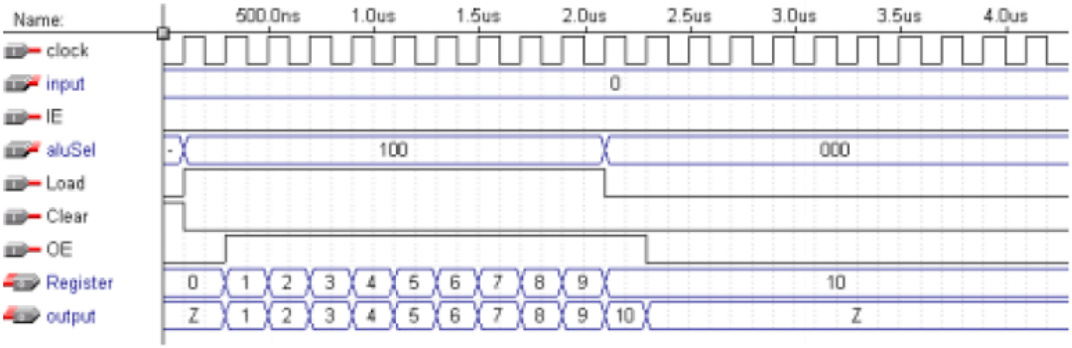



**Figure 9.41** Corrected simulation trace for using the three control words from Figure 9.40.

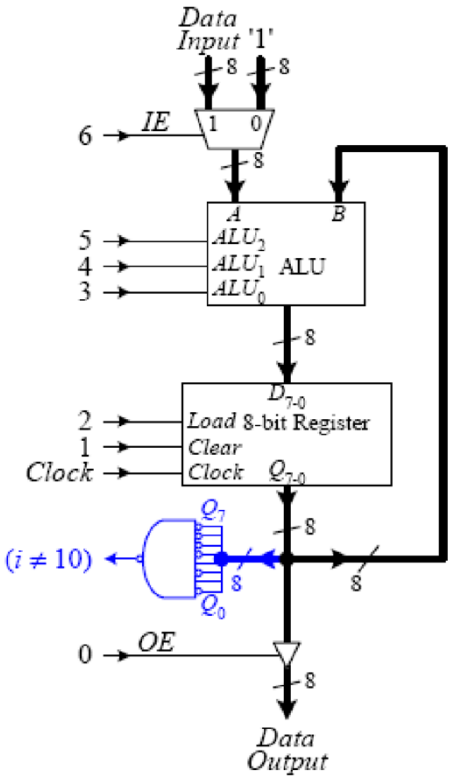
# LAB 3D:

## CONTROL UNIT & INTEGRATION

DURATION: 2 HOURS

Design, derive, test, simulate and implement on your FPGA the control unit of an ALU as shown in Figure 9.

Name of Candidate: _____




```
1          i = 0
2          WHILE (i ≠ 10){
3             i = i + 1
4             OUTPUT i
5          }
```

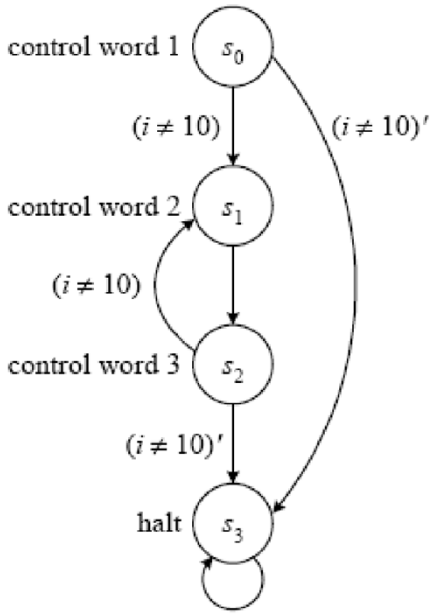
| Control Word | Instruction | *IE* 6 | $ALU_2\,ALU_1\,ALU_0$ 5–3 | *Load* 2 | *Clear* 1 | *OE* 0 |
|---|---|---|---|---|---|---|
| 1 | $i = 0$ | × | ××× | 0 | 1 | 0 |
| 2 | $i = i + 1$ | 0 | 100 (add) | 1 | 0 | 0 |
| 3 | OUTPUT $i$ | × | ××× | 0 | 0 | 1 |

Figure 9 Data path, algorithm & Control Word table

1. Derive your Control Unit (FSM/ state diagram, Excitation Table, Implementation Table, Excitation Equations, Output Equations/ Control Signal Equations, Circuitry)
2. Design the HDL based Control Unit in Quartus II.
3. Verify your state machine using the RTL view.
4. Test your Control Unit using a HDL based Testbench.
5. Integrate your Control Unit & Datapath.
6. Implement integrated design with automation on your FPGA.
7. Log ALL your methodology, observations and results neatly.

Name of Candidate: _____

**Answer**




(a)

| Current State $Q_1Q_0$ | Next State $Q_{1next}\,Q_{0next}$ | |
|---|---|---|
| | $(i \neq 10)'$ | $(i \neq 10)$ |
| $s_0$ 00 | $s_3$ 11 | $s_1$ 01 |
| $s_1$ 01 | $s_2$ 10 | $s_2$ 10 |
| $s_2$ 10 | $s_3$ 11 | $s_1$ 01 |
| $s_3$ 11 | $s_3$ 11 | $s_3$ 11 |

(b)

| Current State $Q_1Q_0$ | Implementation $D_1\,D_0$ | |
|---|---|---|
| | $(i \neq 10)'$ | $(i \neq 10)$ |
| 00 | 11 | 01 |
| 01 | 10 | 10 |
| 10 | 11 | 01 |
| 11 | 11 | 11 |

(c)

$D_0 = Q_1 + Q_0'$

$D_1 = (i \neq 10)' + Q_0$

(d)

| $Q_1Q_0$ | *Clear* |
|---|---|
| 00 | 1 |
| 01 | 0 |
| 10 | 0 |
| 11 | × |

$Clear = Q_1'Q_0'$

(e)

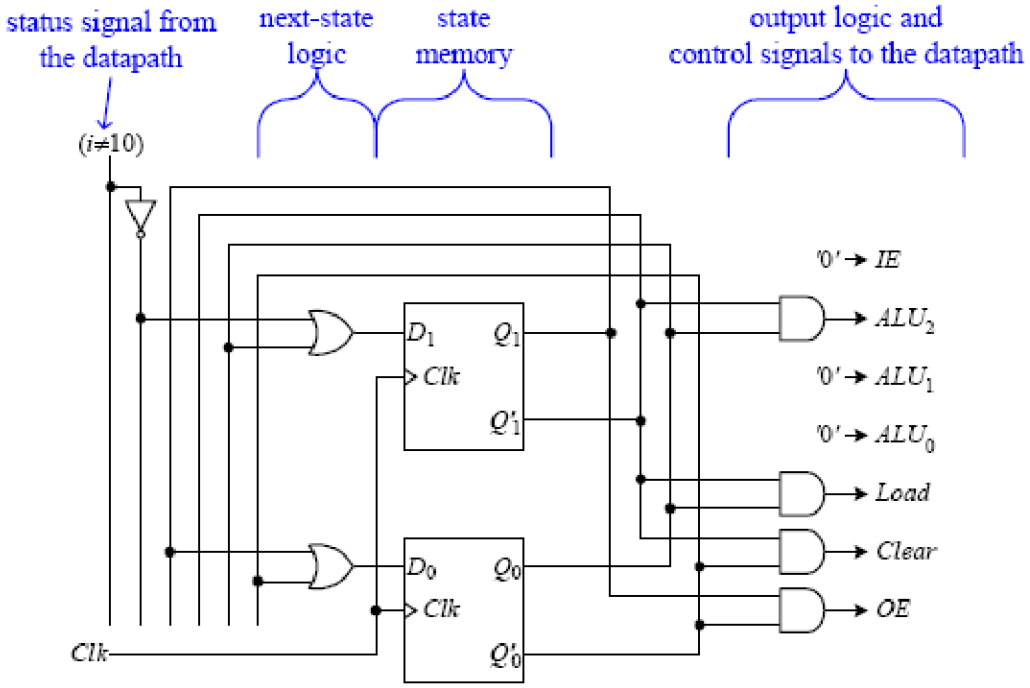
| $Q_1Q_0$ | *IE* | $ALU_2$ | $ALU_1$ | $ALU_0$ | *Load* | *Clear* | *OE* |
|---|---|---|---|---|---|---|---|
| 00 | × | × | × | × | 0 | 1 | 0 |
| 01 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 10 | × | × | × | × | 0 | 0 | 1 |
| 11 | × | × | × | × | × | × | 0 |

(f)

Name of Candidate: _____

$$
\begin{aligned}
IE &= 0 \\
ALU_2 &= Q_1'Q_0 \\
ALU_1 &= 0 \\
ALU_0 &= 0 \\
Load &= Q_1'Q_0 \\
Clear &= Q_1'Q_0' \\
OE &= Q_1Q_0'
\end{aligned}
$$

(g)




(h)

# LAB 4:

## COUNTERS

:: DURATION: 2 HOURS ::

Name of Candidate: _____

The purpose of this exercise is to build and use counters. The designed circuits are to be implemented on an Altera DE1 Board.

Students are expected to have a basic understanding of counters and sufficient familiarity with Verilog hardware description language to implement various types of flip-flops.

**Part I**

Consider the circuit in Figure 1. It is a 4-bit synchronous counter which uses four T-type flip-flops. The counter increments its value on each positive edge of the clock if the *Enable* signal is asserted. The counter is reset to 0 by setting the *Clear* signal low. You are to implement a 8-bit counter of this type.

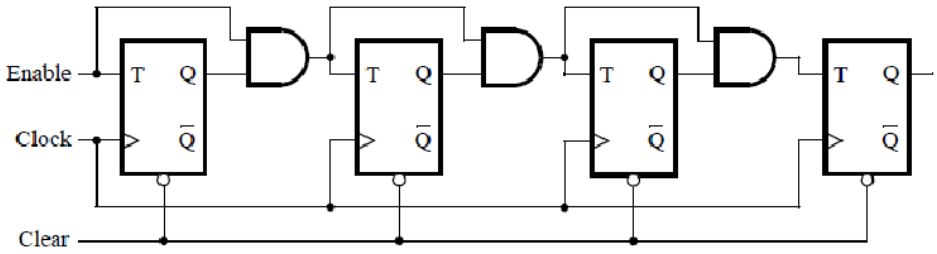



Figure 1: A 4-bit counter.

1. Write a Verilog file that defines a 8-bit counter by using the structure depicted in Figure 1. Your code should include a T flip-flop module that is instantiated 8 times to create the counter. Compile the circuit. How many logic elements (LEs) are used to implement your circuit? What is the maximum frequency, $F_{max}$, at which your circuit can be operated?
2. Simulate your circuit to verify its correctness.
3. Augment your Verilog file to use the pushbutton $KEY_0$ as the *Clock* input, switches $SW_1$ and $SW_0$ as *Enable* and *Clear* inputs, and 7-segment displays *HEX1-0* to display the hexadecimal count as your circuit operates. Make the necessary pin assignments needed to implement the circuit on the DE1 board, and compile the circuit.
4. Download your circuit into the FPGA chip and test its functionality by operating the implemented switches.
5. Implement a 4-bit version of your circuit and use the Quartus II RTL Viewer to see how Quartus II software synthesized your circuit. What are the differences in comparison with Figure 1?

**Part II**

Another way to specify a counter is by using a register and adding 1 to its value. This can be accomplished using the following Verilog statement:

$$Q <= Q + 1;$$

Name of Candidate: _____

Compile a 16-bit version of this counter and determine the number of LEs needed and the $F_{max}$ that is attainable. Use the RTL Viewer to see the structure of this implementation and comment on the differences with the design from Part I.

**Part III**

Use an LPM from the Library of Parameterized modules to implement a 16-bit counter. Choose the LPM options to be consistent with the above design, i.e. with enable and synchronous clear. How does this version compare with the previous designs?

**Note:** The tutorial *Using the Library of Parameterized Modules (LPM)* explains the use of LPMs. It can be found on the Altera University Program website.

**Part IV**

Design and implement a circuit that successively flashes digits 0 through 9 on the 7-segment display *HEX0*. Each digit should be displayed for about one second. Use a counter to determine the one second intervals. The counter should be incremented by the 50-MHz clock signal provided on the DE1 board. Do not derive any other clock signals in your design–make sure that all flip-flops in your circuit are clocked directly by the 50-MHz clock signal.

**Part V**

Design and implement a circuit that displays the word dE1, in ticker tape fashion, on the four 7-segment displays $HEX3-0$. Make the letters move from right to left in intervals of about one second. The patterns that should be displayed in successive clock intervals are given in Table 1.

| Clock cycle | Displayed pattern |
|---|---|
| 0 | d E 1 |
| 1 | d E 1 |
| 2 | E 1 d |
| 3 | 1 d E |
| 4 | d E 1 |
| ... | and so on |

Table 1. Scrolling the word dE1 in ticker-tape fashion.

**Preparation**

The recommended preparation for this laboratory exercise includes:

1. Verilog code for **Part I**
2. Simulation of the Verilog code for **Part I**
3. Verilog code for **Part II**
4. Verilog code for **Part III**

In addition, a module that displays a hex digit on seven segment display the students designed in a previous lab would be an asset.

# LAB 4B:

## TIMING CIRCUITS

DURATION: 2 HOURS

Name of Candidate: _____

1. **1 Hz Clock Generator**
   With an input clock frequency of 50 MHz, create a slower clock of 1 Hz. Simulate the design to verify. Implement your design on the DE1 FPGA board.

2. Design, Synthesize and Implement on the Cyclone II DE1 Board a Clock Divider Selector that allows generating **1 Hz, 2 Hz, 5 Hz** and **10 Hz** clocks from the on-board 50 MHz clock.

   a) Use the given physical design constraint file for pin assignment.

   b) Use the on-board LEDs to show the generated clock.

   c) Use the on-board dip-switches to select the clock divider.

3. **BCD Decoder**
   Write Verilog code to decode a 7-segment display. The inputs are 4-bits binary value. Use Behavioral coding method

4. **Design a 1 Digit BCD up counter**
   Design a 1 Digit BCD up counter to count decimal values from 0 to 9 that wraps back to 0. Display the count on HEX0 of the Cyclone II DE1 board.

5. **Design a 2 Digit BCD up counter**
   Design a 2 Digit BCD up counter to count decimal values from 00 to 99 that wraps back to 00 and repeat counting. The BCD counter should count the internal signals Digit10 and Digit1, which are then decoded to HEX1 and HEX0 outputs. The decoder has been implemented in the previous code. The counting rate should be in seconds. Make KEY[0] as reset counter to "00".

6. **Design a 2 Digit BCD up-down counter**
   Based on the Verilog code from the above design, Modify it to become a 2-Digit up down counter. Use SW[9] as the direction input signal, which controls the direction of counter, i.e, '1' for counting up and '0' for counting down. The counting rate should be in seconds. Design, compile, and implement the design on the DE1 Board.

7. **BCD Up-Down counter with synchronous Load**
   Based on previous design. Write 2 Digit BCD up-down counter to count decimal values. with two additional input signals. It should stop counting up/ down when it reaches 99/ 00 respectively.
   **LOAD (KEY [1]):**
   A synchronous control signal to cause the BCD counter to reload a new value to the counter when it is high and start counting down when it goes low.
   **LOADVALUE (SW[7] to SW[0]):** An 8-bit value to be loaded to the counter after the LOAD signal is asserted.

8. **REAL TIME CLOCK**
   Design and implement a circuit on the DE1 board that acts as a real-time clock. It should display the minute (from 0 to 60) on *HEX3−2* and the second (from 0 to 60) on *HEX1−0*. Use the switches *SW*7−0 to preset the minute part of the time displayed by the clock.

# LAB 5:

## Timers and Real-time Clock

DURATION: 2 HOURS

Name of Candidate: _____

The purpose of this exercise is to study the use of clocks in timed circuits. The designed circuits are to be implemented on an Altera DE1 board.

**Background**

In Verilog hardware description language we can describe a variable-size counter by using a parameter declaration. An example of an *n*-bit counter is shown in Figure 1.

```
module counter(clock, reset_n, Q);
    parameter n = 4;

    input  clock, reset_n;
    output [n-1:0] Q;
    reg    [n-1:0] Q;

    always @(posedge clock or negedge reset_n)
    begin
        if (~reset_n)
            Q <= 'd0;
        else
            Q <= Q + 1'b1;
    end
endmodule
```

Figure 1: A Verilog description of an *n*-bit counter.

The parameter *n* specifies the number of bits in the counter. A particular value of this parameter is defined by using a **defparam** statement. For example, an 8-bit counter can be specified as:

```
counter eight_bit(clock, reset_n, Q);
    defparam eight_bit.N = 8;
```

By using parameters we can instantiate counters of different sizes in a logic circuit, without having to create a new module for each counter.

**Part I**

Create a modulo-k counter by modifying the design of an 8-bit counter to contain an additional parameter. The counter should count from 0 to $k-1$. When the counter reaches the value $k-1$ the value that follows should be 0.

Your circuit should use pushbutton $KEY_0$ as an asynchronous reset, $KEY_1$ as a manual clock input. The contents of the counter should be displayed on red LEDs. Compile your design with Quartus II software, download your design onto a DE1 board, and test its operation. Perform the following steps:

1. Create a new Quartus II project which will be used to implement the desired circuit on the DE1 board.

Name of Candidate: _____

2. Write a Verilog file that specifies the desired circuit.
3. Include the Verilog file in your project and compile the circuit.
4. Simulate the designed circuit to verify its functionality.
5. Assign the pins on the FPGA to connect to the lights and pushbutton switches, by importing the pin-assignment file *DE1_pin_assignments.qsf*.
6. Recompile the circuit and download it into the FPGA chip.
7. Verify that your circuit works correctly by observing the display.

**Part II**

Implement a 3-digit BCD counter. Display the contents of the counter on the 7-segment displays, $HEX2{-}0$. Derive a control signal, from the 50-MHz clock signal provided on the DE1 board, to increment the contents of the counter at one-second intervals. Use the pushbutton switch $KEY_0$ to reset the counter to 0.

**Part III**

Design and implement a circuit on the DE1 board that acts as a real-time clock. It should display the minutes (from 0 to 60) on $HEX3{-}2$ and the seconds (from 0 to 60) on $HEX1{-}0$. Use the switches $SW_{7-0}$ to preset the minute part of the time displayed by the clock.

**Part IV**

An early method of telegraph communication was based on the Morse code. This code uses patterns of short and long pulses to represent a message. Each letter is represented as a sequence of dots (a short pulse), and dashes (a long pulse). For example, the first eight letters of the alphabet have the following representation:

| | |
|---|---|
| A | • — |
| B | — • • • |
| C | — • — • |
| D | — • • |
| E | • |
| F | • • — • |
| G | — — • |
| H | • • • • |

Design and implement a circuit that takes as input one of the first eight letters of the alphabet and displays the Morse code for it on a red LED. Your circuit should use switches $SW_{2-0}$ and pushbuttons $KEY_{1-0}$ as inputs. When a user presses $KEY_1$, the circuit should display the Morse code for a letter specified by $SW_{2-0}$ (000 for A, 001 for B, etc.), using 0.5-second pulses to represent dots, and 1.5-second pulses to represent dashes. Pushbutton $KEY_0$ should function as an asynchronous reset. A high-level schematic diagram of the circuit is shown in Figure 2.

**Hint:** Use a counter to generate 0.5-second pulses, and another counter to keep the $LEDR_0$ light on for either 0.5 or 1.5 seconds.

Name of Candidate: _____

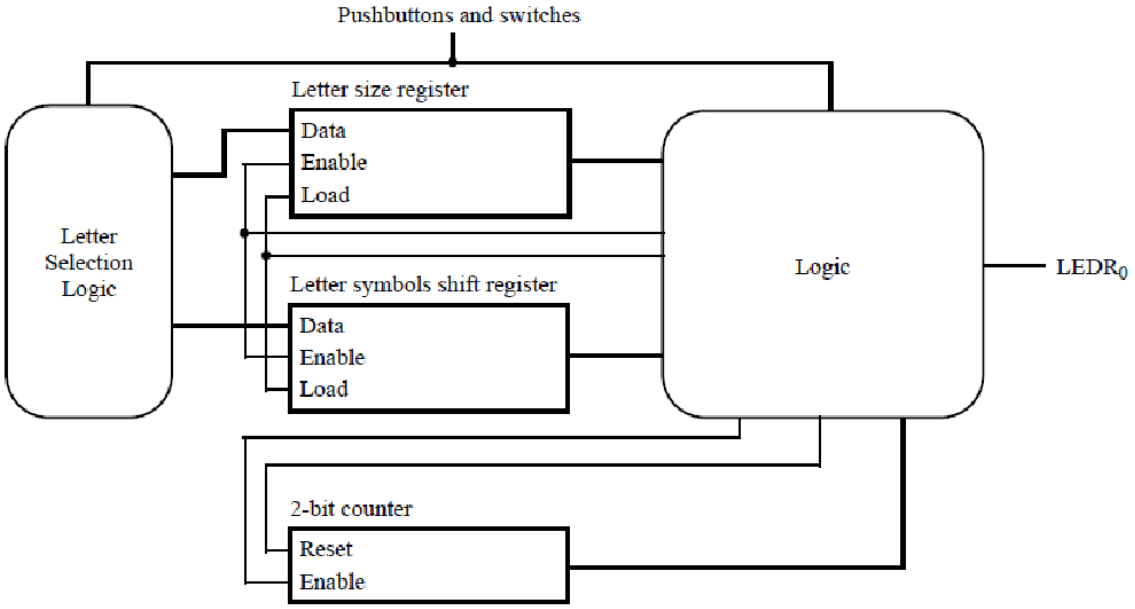



Figure 2: High-level schematic diagram of the circuit for part IV.

**Preparation**

The recommended preparation for this laboratory exercise includes:

1. Verilog code for **Part I**
2. Simulation of the Verilog code for **Part I**
3. Verilog code for **Part II**
4. Verilog code for **Part III**

Name of Candidate: _____

# LAB 6:

## Adders, Subtractors, and Multiplexers

DURATION: 2 HOURS

The purpose of this exercise is to examine arithmetic circuits that add, subtract, and multiply numbers. Each circuit will be described in Verilog and implemented on an Altera DE1 board.

**Part I**

Consider again the four-bit ripple-carry adder circuit used in lab exercise 2; its diagram is reproduced in Figure 1.

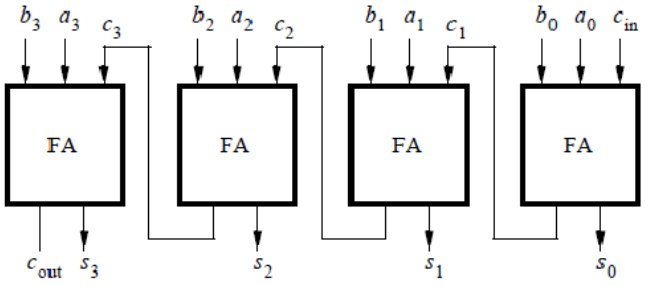



Figure 1: A four-bit ripple carry adder.

This circuit can be implemented using a '+' sign in Verilog. For example, the following code fragment adds $n$-bit numbers $A$ and $B$ to produce outputs $sum$ and $carry$:

```
wire [n-1:0] sum;
wire carry;
...
assign {carry, sum} = A + B;
```

Use this construct to implement a circuit shown in Figure 2.

Design and compile your circuit with Quartus II software, download it onto a DE1 board, and test its operation as follows:

1. Create a new Quartus II project. Select as the target chip the Cyclone II EP2C20F484C7, which is the FPGA chip on the Altera DE1 board. Select Cyclone II EP2C20F484C7 device to implement the designed circuit on the DE1 board.
2. Write Verilog code that describes the circuit in Figure 2.
3. Connect input $A$ to switches $SW_{7-0}$, and use $KEY_0$ as an active-low asynchronous reset and $KEY_1$ as a manual clock input. The sum output should be displayed on red $LEDR_{7-0}$ lights and the carry-out should be displayed on the red $LEDR_0$ light.
4. Assign the pins on the FPGA to connect to the switches and 7-segment displays by importing the *DE1_pin_assignments.qsf* file.
5. Compile your design and use timing simulation to verify the correct operation of the circuit. Once the simulation works properly, download the circuit onto the DE1 board and test it by using different values of $A$. Be sure to check that the *Overflow* output works correctly.

Name of Candidate: _____

6. Open the Quartus II Compilation Report and examine the results reported by the Timing Analyzer. What is the maximum operation frequency, $f_{max}$, of your circuit? What is the longest path in the circuit in terms of delay?

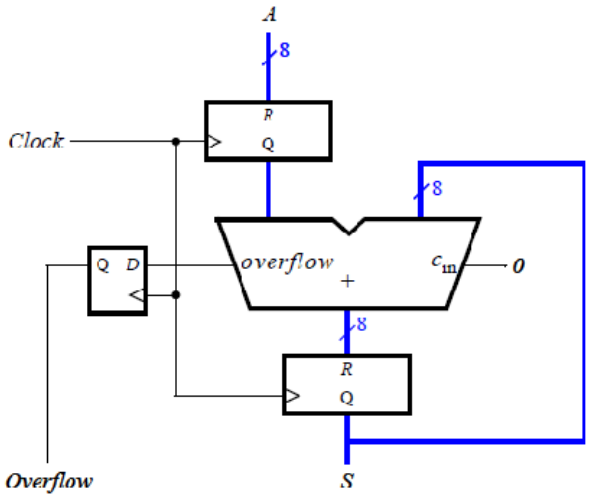



Figure 2: An eight-bit accumulator circuit.

**Part II**

Extend the circuit from Part I to be able to both add and subtract numbers. To do so, add an *add_sub* input to your circuit. When *add_sub* is 1, your circuit should subtract $A$ from $S$, and add $A$ and $S$ as in Part I otherwise.

**Part III**

Figure 3$a$ gives an example of paper-and-pencil multiplication $P = A \times B$, where $A = 11$ and $B = 12$.

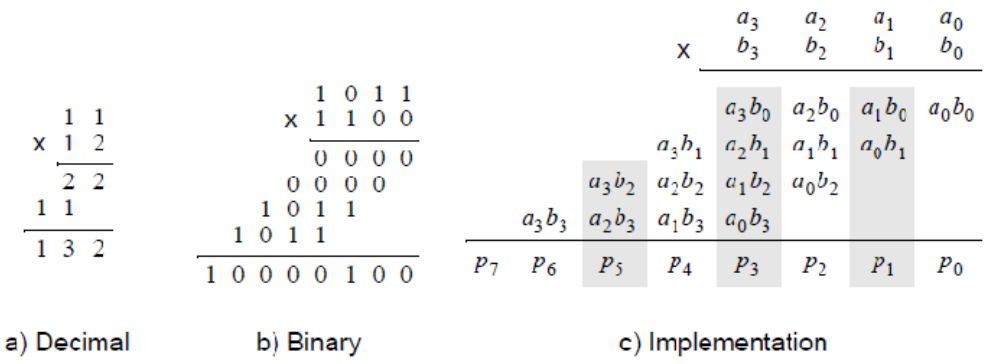



Figure 3: Multiplication of binary numbers.

We compute $P = A \times B$ as an addition of summands. The first summand is equal to $A$ times the ones digit of $B$. The second summand is $A$ times the tens digit of $B$, shifted one position to the left. We add the two summands to form the product $P = 132$.

Part $b$ of the figure shows the same example using four-bit binary numbers. To compute $P = A \times B$, we first form summands by multiplying $A$ by each digit of $B$. Since each digit of $B$ is either 1 or 0, the summands are either shifted versions of $A$ or 0000. Figure 3$c$ shows how each summand can be formed by using the Boolean AND operation of $A$ with the appropriate bit of $B$.

Name of Candidate: _____

A four-bit circuit that implements $P = A \times B$ is illustrated in Figure 4. Because of its regular structure, this type of multiplier circuit is called an *array multiplier*. The shaded areas correspond to the shaded columns in Figure 3*c*. In each row of the multiplier AND gates are used to produce the summands, and full adder modules are used to generate the required sums.

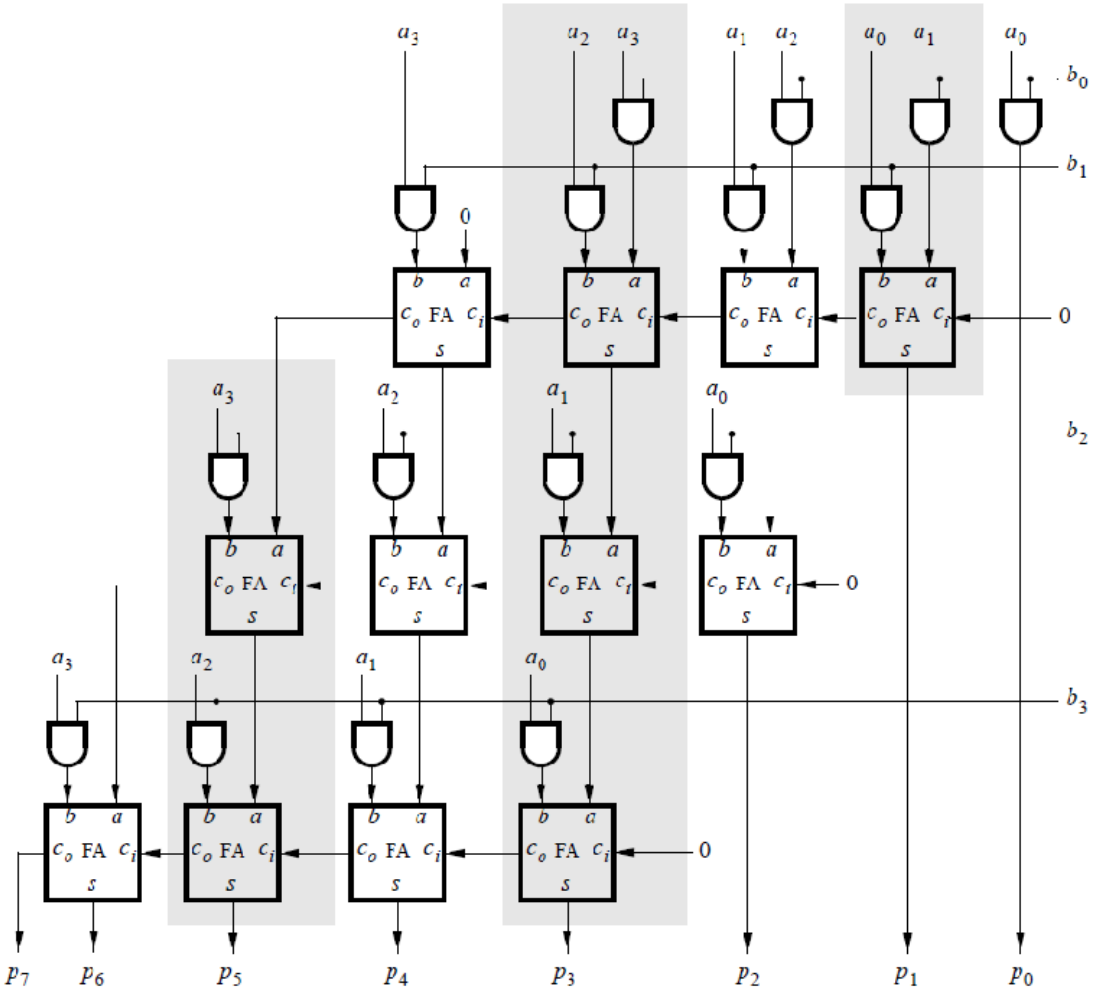



Figure 4: An array multiplier circuit.

Perform the following steps to implement the array multiplier circuit:

1. Create a new Quartus II project to implement the desired circuit on the Altera DE1 board.
2. Generate the required Verilog file, include it in your project, and compile the circuit.
3. Use functional simulation to verify your design.
4. Augment your design to use switches $SW_{11-8}$ to represent the number $A$ and switches $SW_{3-0}$ to represent $B$. The hexadecimal values of $A$ and $B$ are to be displayed on the 7-segment displays *HEX6* and *HEX4*, respectively. The result $P = A \times B$ is to be displayed on *HEX1* and *HEX0*.
5. Assign the pins on the FPGA to connect to the switches and 7-segment displays by importing the *DE1_pin_assignments.qsf* file.
6. Recompile the circuit and download it into the FPGA chip.
7. Test the functionality of your circuit by toggling the switches and observing the 7-segment displays.

Name of Candidate: _____

**Part IV**

In Part III, an array multiplier was implemented using full adder modules. At a higher level, a row of full adders functions as an $n$-bit adder and the array multiplier circuit can be represented as shown in Figure 5.

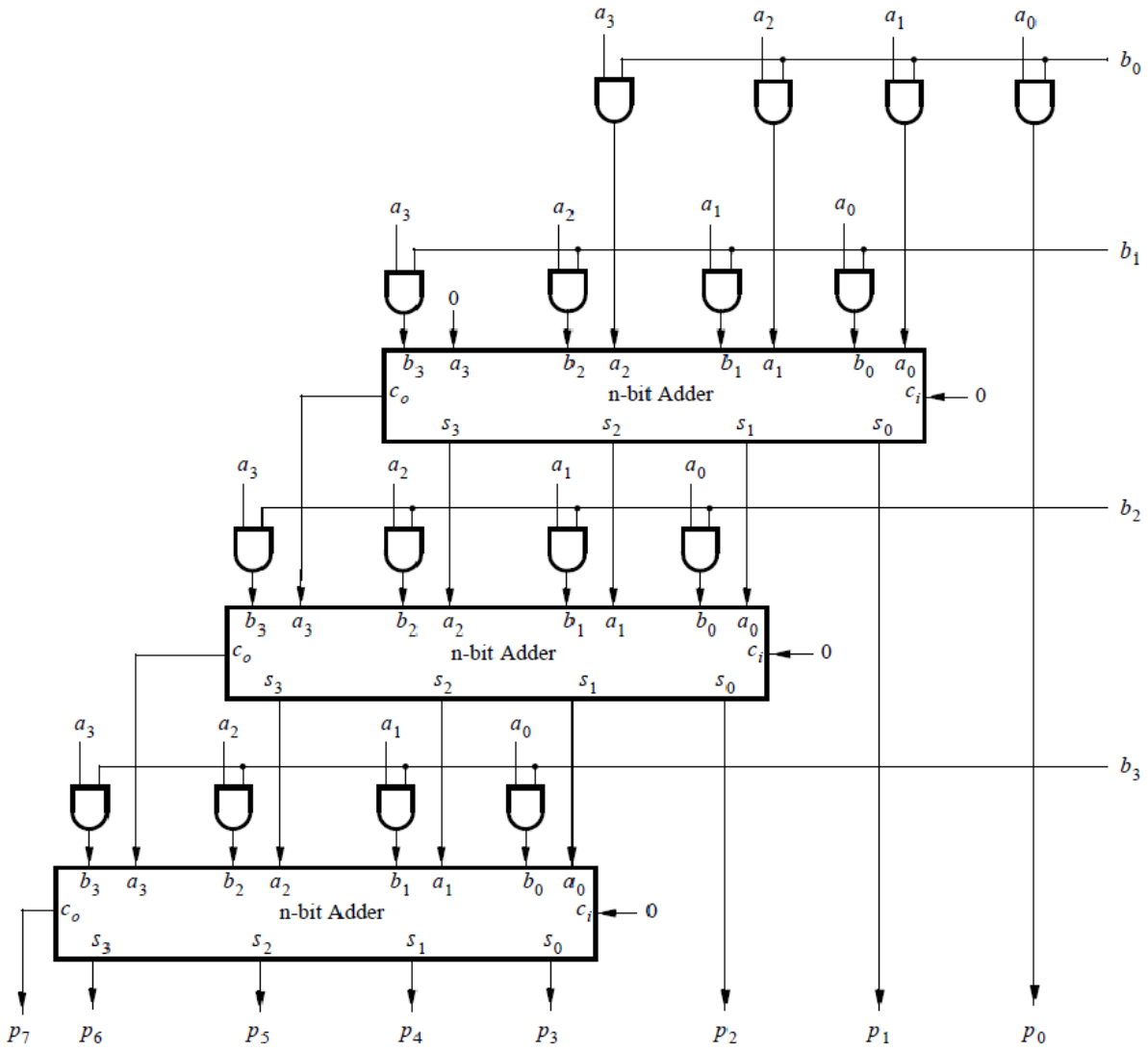



Figure 5: An array multiplier implemented using $n$-bit adders.

Each $n$-bit adder adds a shifted version of $A$ for a given row and the partial sum of the row above. Abstracting the multiplier circuit as a sequence of additions allows us to build larger multipliers. The multiplier should consist of n-bit adders arranged in a structure shown in Figure 5. Use this approach to implement a 5x5 multiplier circuit with registered inputs and outputs, as shown in Figure 6.

Name of Candidate: _____

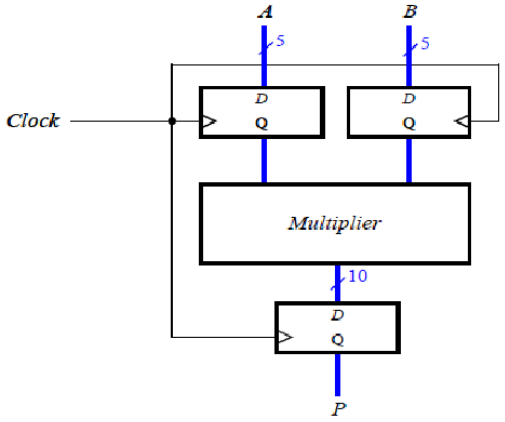



Figure 6: A registered multiplier circuit.

Perform the following steps:

1. Create a new Quartus II project.
2. Write the required Verilog file, include it in your project, and compile the circuit.
3. Use functional simulation to verify your design.
4. Augment your design to use switches $SW_{9-5}$ to represent the number $A$ and switches $SW_{4-0}$ to represent $B$. The result $P = A \times B$ is to be displayed on *HEX3-0*.
5. Assign the pins on the FPGA to connect to the switches and 7-segment displays.
6. Recompile the circuit and download it into the FPGA chip.
7. Test the functionality of your design by toggling the switches and observing the 7-segment displays.
8. How large is the circuit in terms of the number of logic elements?
9. What is the $f_{max}$ for this circuit?

## Part V

Part IV showed how to implement multiplication $A \times B$ as a sequence of additions, by accumulating the shifted versions of $A$ one row at a time. Another way to implement this circuit is to perform addition using an adder tree.

An adder tree is a method of adding several numbers together in a parallel fashion. This idea is illustrated in Figure 7. In the figure, numbers $A$, $B$, $C$, $D$, $E$, $F$, $G$, and $H$ are added together in parallel. The addition $A + B$ happens simultaneously with $C + D$, $E + F$ and $G + H$. The result of these operations are then added in parallel again, until the final sum $P$ is computed.

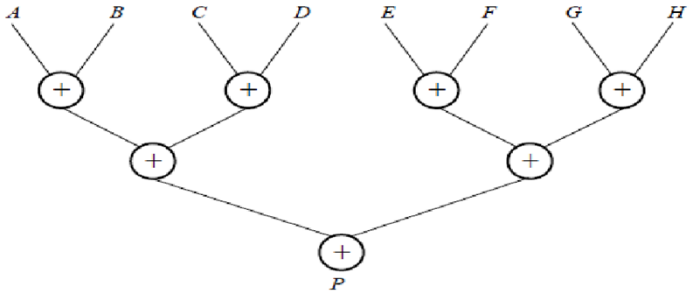



Figure 7: An example of adding 8 numbers using an adder tree.

In this part you are to implement a 5x5 array multiplier that computes $P = A \times B$. Use an adder tree structure to implement operations shown in Figure 5. Inputs $A$ and $B$, as well as the output $P$ should be registered as in Part IV. What is the $f_{max}$ for this circuit?

**Preparation**

The recommended preparation for this laboratory exercise includes Verilog code for Parts I through V.

Name of Candidate: _____

# LAB 7:

## Finite State Machine

DURATION : 2 HOURS

This is an exercise in using finite state machines.

**Part I**

We wish to implement a finite state machine (FSM) that recognizes two specific sequences of applied input symbols, namely four consecutive 1s or four consecutive 0s. There is an input $w$ and an output $z$. Whenever $w = 1$ or $w = 0$ for four consecutive clock pulses the value of $z$ has to be 1; otherwise, $z = 0$. Overlapping sequences are allowed, so that if $w = 1$ for five consecutive clock pulses the output $z$ will be equal to 1 after the fourth and fifth pulses. Figure 1 illustrates the required relationship between $w$ and $z$.

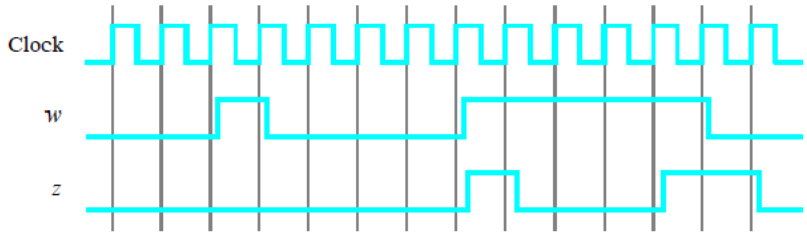



Figure 1: Required timing for the output $z$.

A state diagram for this FSM is shown in Figure 2. For this part you are to manually derive an FSM circuit that implements this state diagram, including the logic expressions that feed each of the state flip-flops. To implement the FSM use nine state flip-flops called $y_8, \ldots, y_0$ and the one-hot state assignment given in Table 1.

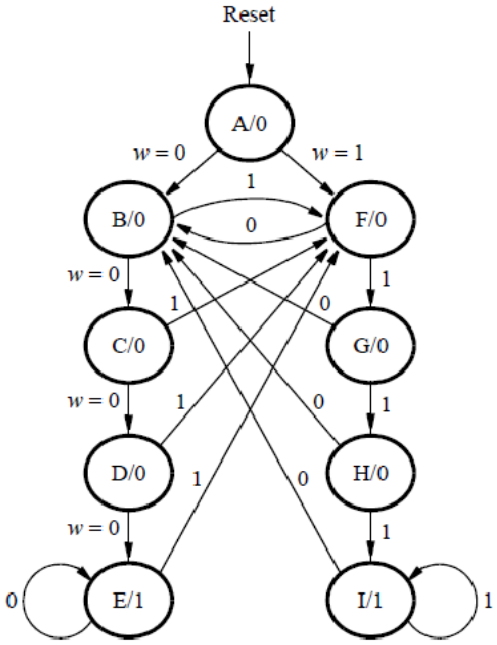



Figure 2: A state diagram for the FSM.

Name of Candidate: _____

| Name | State Code $y_8y_7y_6y_5y_4y_3y_2y_1y_0$ |
|---|---|
| **A** | 000000001 |
| **B** | 000000010 |
| **C** | 000000100 |
| **D** | 000001000 |
| **E** | 000010000 |
| **F** | 000100000 |
| **G** | 001000000 |
| **H** | 010000000 |
| **I** | 100000000 |

Table 1: One-hot codes for the FSM.

Design and implement your circuit on the DE1 board as follows:

1. Create a new Quartus II project for the FSM circuit. Select as the target chip the Cyclone II EP2C20F484C7, which is the FPGA chip on the Altera DE1 board.

2. Write a Verilog file that instantiates the nine flip-flops in the circuit and which specifies the logic expressions that drive the flip-flop input ports. Use only simple **assign** statements in your Verilog code to specify the logic feeding the flip-flops. Note that the one-hot code enables you to derive these expressions by inspection.

   Use the toggle switch $SW_0$ on the DE1 board as an active-low synchronous reset input for the FSM, use $SW_1$ as the $w$ input, and the pushbutton $KEY_0$ as the clock input which is applied manually. Use the green light $LEDG_0$ as the output $z$, and assign the state flip-flop outputs to the red lights $LEDR_8$ to $LEDR_0$.

3. Include the Verilog file in your project, and assign the pins on the FPGA to connect to the switches and the LEDs, as indicated in the User Manual for the DE1 board. Compile the circuit.

4. Simulate the behavior of your circuit.

5. Once you are confident that the circuit works properly as a result of your simulation, download the circuit into the FPGA chip. Test the functionality of your design by applying the input sequences and observing the output LEDs. Make sure that the FSM properly transitions between states as displayed on the red LEDs, and that it produces the correct output values on $LEDG_0$.

6. Finally, consider a modification of the one-hot code given in Table 1. When an FSM is going to be implemented in an FPGA, the circuit can often be simplified if all flip-flop outputs are 0 when the FSM is in the reset state. This approach is preferable because the FPGA's flip-flops usually include a *clear* input, which can be conveniently used to realize the reset state, but the flip-flops often do not include a *set* input.

   Table 2 shows a modified one-hot state assignment in which the reset state, $A$, uses all 0s. This is accomplished by inverting the state variable $y_0$. Create a modified version of your Verilog code that implements this state assignment. (*Hint*: you should need to make very few changes to the logic expressions in your circuit to implement the modified state assignment.) Compile your new circuit and test it both through simulation and by downloading it onto the DE1 board.

Name of Candidate: _____

| Name | State Code $y_8y_7y_6y_5y_4y_3y_2y_1y_0$ |
|---|---|
| **A** | 000000000 |
| **B** | 000000011 |
| **C** | 000000101 |
| **D** | 000001001 |
| **E** | 000010001 |
| **F** | 000100001 |
| **G** | 001000001 |
| **H** | 010000001 |
| **I** | 100000001 |

Table 2: Modified one-hot codes for the FSM.

**Part II**

For this part you are to write another style of Verilog code for the FSM in Figure 2. In this version of the code you should not manually derive the logic expressions needed for each state flip-flop. Instead, describe the state table for the FSM by using a Verilog **case** statement in an **always** block, and use another **always** block to instantiate the state flip-flops. You can use a third **always** block or simple assignment statements to specify the output $z$. To implement the FSM, use four state flip-flops $y_3, \ldots, y_0$ and binary codes, as shown in Table 3.

| Name | State Code $y_3y_2y_1y_0$ |
|---|---|
| **A** | 0000 |
| **B** | 0001 |
| **C** | 0010 |
| **D** | 0011 |
| **E** | 0100 |
| **F** | 0101 |
| **G** | 0110 |
| **H** | 0111 |
| **I** | 1000 |

Table 3: Binary codes for the FSM.

A suggested skeleton of the Verilog code is given in Figure 3.

Name of Candidate: _____

```
module part2 ( . . . );
   . . . define input and output ports

   . . . define signals
   reg [3:0] y_Q, Y_D;     // y_Q represents current state, Y_D represents next state
   parameter A = 4'b0000, B = 4'b0001, C = 4'b0010, D = 4'b0011, E = 4'b0100,
      F = 4'b0101, G = 4'b0110, H = 4'b0111, I = 4'b1000;

   always @(w, y_Q)
   begin: state_table
      case (y_Q)
         A: if (!w) Y_D = B;
            else Y_D = F;
         . . . remainder of state table
         default: Y_D = 4'bxxxx;
      endcase
   end // state_table

   always @(posedge Clock)
   begin: state_FFs
      . . .
   end // state_FFS

   . . . assignments for output z and the LEDs
endmodule
```

Figure 3: Skeleton Verilog code for the FSM.

Implement your circuit as follows.

1. Create a new project for the FSM.
2. Include in the project your Verilog file that uses the style of code in Figure 3. Use the toggle switch $SW_0$ on the DE1 board as an active-low synchronous reset input for the FSM, use $SW_1$ as the $w$ input, and the pushbutton $KEY_0$ as the clock input which is applied manually. Use the green light $LEDG_0$ as the output $z$, and assign the state flip-flop outputs to the red lights $LEDR_3$ to $LEDR_0$. Assign the pins on the FPGA to connect to the switches and the LEDs, as indicated in the User Manual for the DE1 board.
3. Before compiling your code it is necessary to explicitly tell the Synthesis tool in Quartus II that you wish to have the finite state machine implemented using the state assignment specified in your Verilog code. If you do not explicitly give this setting to Quartus II, the Synthesis tool will automatically use a state assignment of its own choosing, and it will ignore the state codes specified in your Verilog code. To make this setting, choose Assignments > Settings in Quartus II, and click on the Analysis and Synthesis item on the left side of the window, then click on the More Setting button. As indicated in Figure 4, change the parameter State Machine Processing to the setting User-Encoded.
4. To examine the circuit produced by Quartus II open the RTL Viewer tool. Double-click on the box shown in the circuit that represents the finite state machine, and determine whether the state diagram that it shows properly corresponds to the one in Figure 2. To see the state codes used for your FSM, open the Compilation Report, select the Analysis and Synthesis section of the report, and click on State Machines.
5. Simulate the behavior of your circuit.
6. Once you are confident that the circuit works properly as a result of your simulation, download the circuit into the FPGA chip. Test the functionality of your design by applying the input sequences and observing the output LEDs. Make sure that the FSM properly transitions between states as displayed on the red LEDs, and that it produces the correct output values on $LEDG_0$.

Name of Candidate: _____

7. In step 3 you instructed the Quartus II Synthesis tool to use the state assignment given in your Verilog code. To see the result of removing this setting, open again the Quartus II settings window by choosing Assignments > Settings, and click on the Analysis and Synthesis item, then click on the More Setting button. Change the setting for State Machine Processing from User-Encoded to One-Hot. Recompile the circuit and then open the report file, select the Analysis and Synthesis section of the report, and click on State Machines. Compare the state codes shown to those given in Table 2, and discuss any differences that you observe.

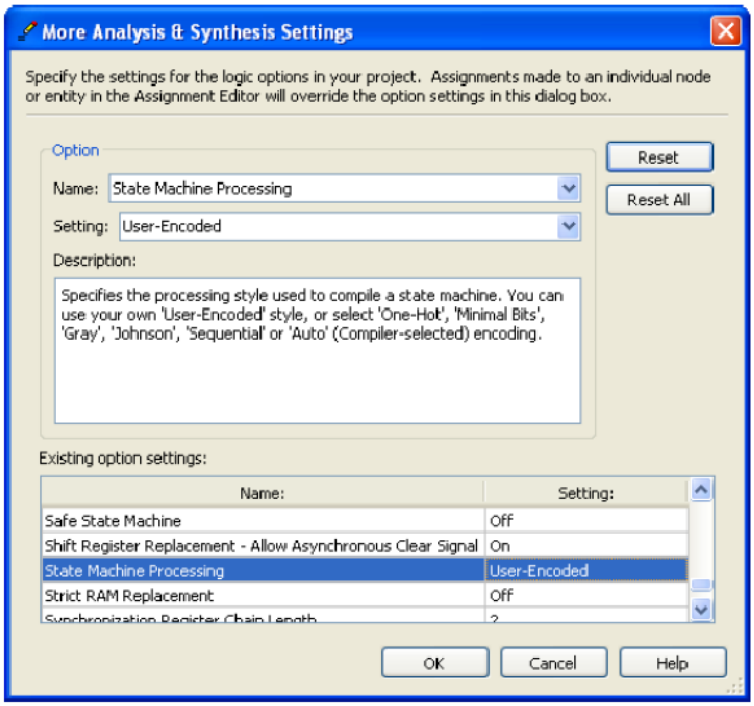



Figure 4: Specifying the state assignment method in Quartus II.

**Part III**

The sequence detector can be implemented in a straightforward manner using shift registers, instead of using the more formal approach described above. Create Verilog code that instantiates two 4-bit shift registers; one is for recognizing a sequence of four 0s, and the other for four 1s. Include the appropriate logic expressions in your design to produce the output $z$. Make a Quartus II project for your design and implement the circuit on the DE1 board. Use the switches and LEDs on the board in a similar way as you did for Parts I and II and observe the behavior of your shift registers and the output $z$. Answer the following question: could you use just one 4-bit shift register, rather than two? Explain your answer.

Name of Candidate: _____

**Part IV**

In this part of the exercise you are to implement a Morse-code encoder using an FSM. The Morse code uses patterns of short and long pulses to represent a message. Each letter is represented as a sequence of dots (a short pulse), and dashes (a long pulse). For example, the first eight letters of the alphabet have the following representation:

| | |
|---|---|
| A | •— |
| B | —••• |
| C | —•—• |
| D | —•• |
| E | • |
| F | ••—• |
| G | ——• |
| H | •••• |

Design and implement a Morse-code encoder circuit using an FSM. Your circuit should take as input one of the first eight letters of the alphabet and display the Morse code for it on a red LED. Use switches $SW_{2-0}$ and pushbuttons $KEY_{1-0}$ as inputs. When a user presses $KEY_1$, the circuit should display the Morse code for a letter specified by $SW_{2-0}$ (000 for A, 001 for B, etc.), using 0.5-second pulses to represent dots, and 1.5-second pulses to represent dashes. Pushbutton $KEY_0$ should function as an asynchronous reset.

A high-level schematic diagram of a possible circuit for the Morse-code encoder is shown in Figure 5.

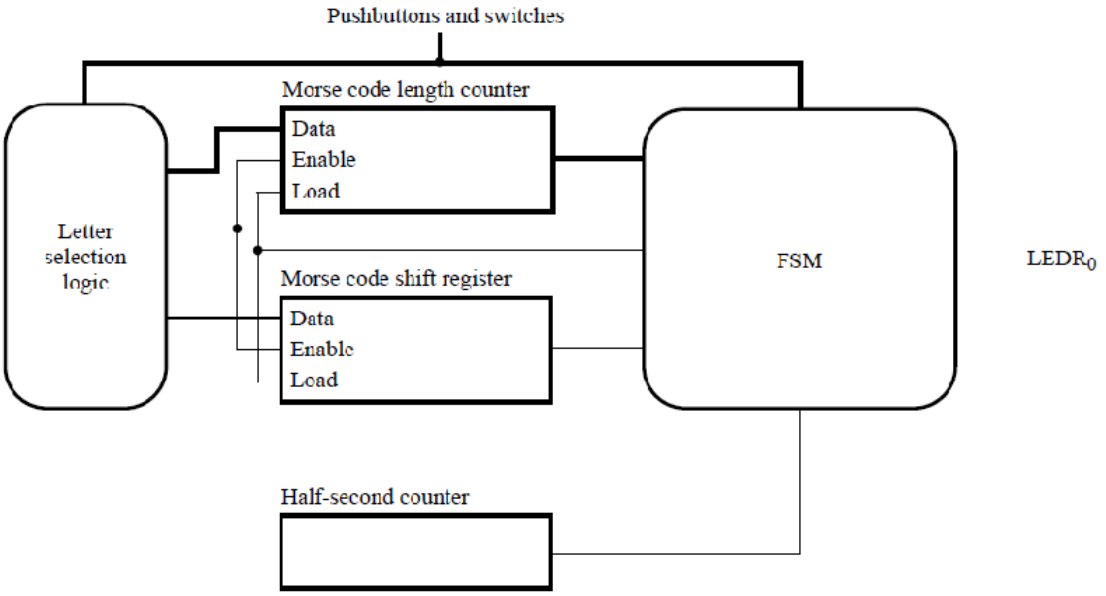



Figure 5: High-level schematic diagram of the circuit for Part IV.

**Preparation**

The recommended preparation for this exercise is to write Verilog code for Parts I through IV.

Name of Candidate: _____

# LAB 7B:

## Traffic Light Controller (FSM)

DURATION: 2 HOURS
:: ASSIGNMENT 1::

Design using a Finite State Machine Approach (FSM) a Traffic Signal Controller.

**Specification**

Consider a controller for traffic at the intersection of a main highway and a country road as shown in Fig. 7 below:-

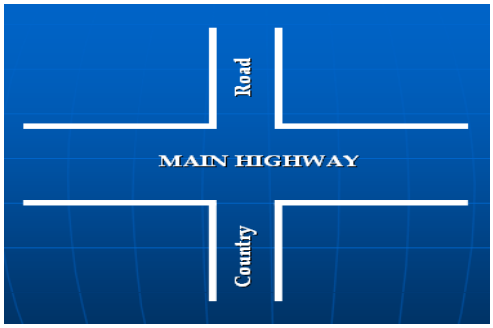
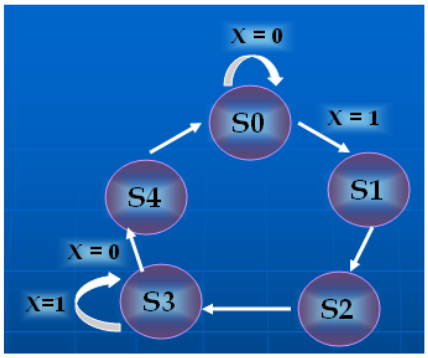



Fig. 7 Cross Section of Highway and Country Road

The following specifications must be considered:-

1. The traffic signal for the main highway gets highest priority because cars are continuously present on the main highway. Thus, the main highway signal remains green by default.
2. Occasionally, cars from the country road arrive at the traffic signal. The traffic signal for the country road must turn green only long enough to let the cars on the country road go.
3. As soon as there are no cars on the country road, the country road traffic signal turns yellow and then red and the traffic signal on the main highway turns green again.
4. There is a sensor to detect cars waiting on the country road. The sensor sends a signal X as input to the controller. X = 1 if there are cars on the country road; otherwise, X = 0.
5. There are delays on transitions from S1 to S2, from S2 to S3, and from S4 to S0. The delays must be controllable. Let S1 to S2 =10s delay, S2 to S3 = 20s delay. S4 to S0 = 10s delay.
6. Display a count-down timer on two HEX for all delays stated in 5.
7. Display “Hr GO” during state S0 and “Cr GO” during state S3.
8. Display “STOP” on HEX [3:0] for 5s during S2 to S3. Blinking the word “STOP” ON and OFF.
9. State diagram and state definitions for the traffic signal controller are as shown in Fig. 7B below:-




| State | Signals |
|---|---|
| *S0* | *Hwy = G Cntry = R* |
| *S1* | *Hwy = Y Cntry = R* |
| *S2* | *Hwy = R Cntry = R* |
| *S3* | *Hwy = R Cntry = G* |
| *S4* | *Hwy = R Cntry = Y* |

Name of Candidate: _____

# LAB 7C:

## Train Controller (FSM)

DURATION: 2 HOURS
:: ASSIGNMENT 1::

## 8 State Machine Design: The Electric Train Controller

### 8.1 The Train Control Problem

The track layout of a small electric train system is shown in Figure 8.1. Two trains, we'll call A and B, run on the tracks, hopefully without colliding. To avoid collisions, the trains require a safety controller that allows trains to move in and out of intersections without mishap.

For safe operation, only one train at a time can be present on any given track segment. The track layout seen in Figure 8.1 is divided into four track segments. Each track segment has sensors that are used to detect trains at the entry and exit points.

In Figure 8.1, there are two Trains A and B. As an example, assume Train A always runs on the outer track loop and Train B on the inner track loop. Assume for a moment that Train A has just passed Sensor 4 and is near Switch 3 moving counterclockwise. Let's also assume that Train B is moving counterclockwise and approaching Sensor 2. Since Train B is entering the common track (Track 2), Train A must be stopped when it reaches Sensor 1, and must wait until Train B has passed Sensor 3 (i.e., Train B is out of the common track). At this point, the track switches should switch for Train A, Train A will be allowed to enter Track 2, and Train B will continue moving toward Sensor 2.

The controller is a state machine that uses the sensors as inputs. The controller's outputs control the direction of the trains and the position of the switches. However, the state machine does not control the speed of the train. This means that the system controller must function correctly independent of the speed of the two trains.

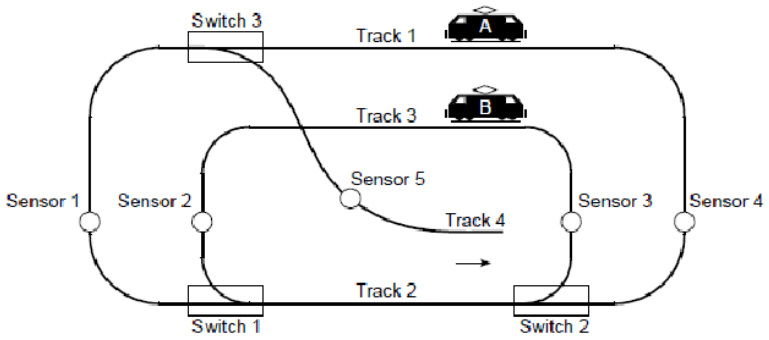



**Figure 8.1** Track layout with input sensors and output switches and output tracks.

Name of Candidate: _____

An FPGA-based "virtual" train simulation will be used that emulates this setup and provides video output. Since there are no actual power circuits connected to a train on the FPGA board, it is only intended to give you a visual indication of how the output signals work in the real system. The following sections describe how the state machine should control each signal to operate the trains properly.

## 8.2 Train Direction Outputs (DA1-DA0, and DB1-DB0)

The direction for each train is controlled by four output signals (two for each train), DA (DA1-DA0) for train A, and DB (DB1-DB0) for train B[2]. When these signals indicate forward "01" for a particular train, a train will move counterclockwise (on track 4, the train moves toward the outer track). When the signals imply reverse "10", the train(s) will move clockwise. The "11" value is illegal and should not be used. When these signals are set to "00", a train will stop. (See Figure 8.2.)

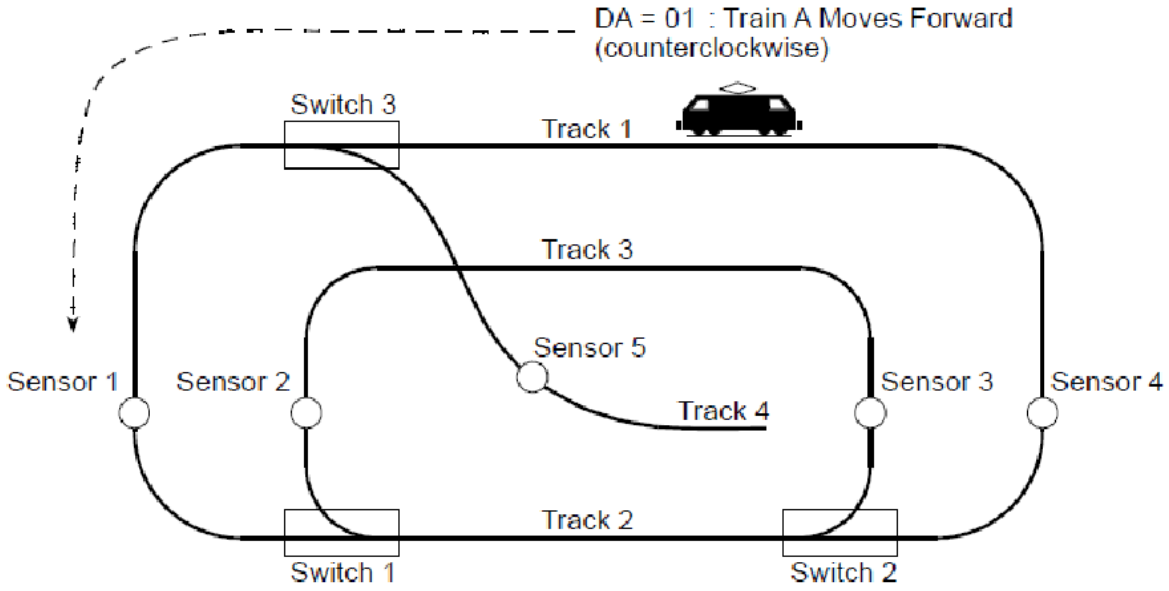



**Figure 8.2** Controlling the train's motion with the train direction signals.

Name of Candidate: _____

## 8.3 Switch Direction Outputs (SW1, SW2, and SW3)

Switch directions are controlled by asserting the SW1, SW2, and SW3 output signals either high (outside connected with inside track) or low (outside tracks connected). That is, anytime all of the switches are set to 1, the tracks are setup such that the outside tracks are connected to the inside tracks. (See Figure 8.3.)

If a train moves the wrong direction through an open switch it will derail. Be careful. If a train is at the point labeled "Track 1" in Figure 8.3 and is moving to the left, it will derail at Switch 3. To keep it from derailing, SW3 would need to be set to 0.

Also, note that Tracks 3 and 4 cross at an intersection and care must be taken to avoid a crash at this point.

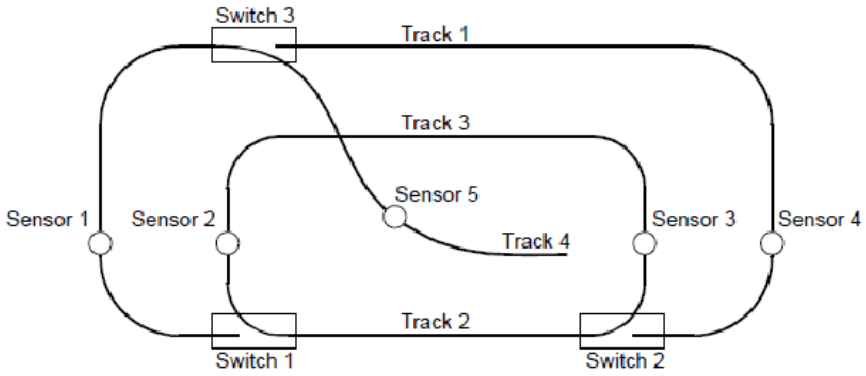



**Figure 8.3** Track direction if all switches are asserted (SW1 = SW2 = SW3 = 1)

## 8.4 Train Sensor Input Signals (S1, S2, S3, S4, and S5)

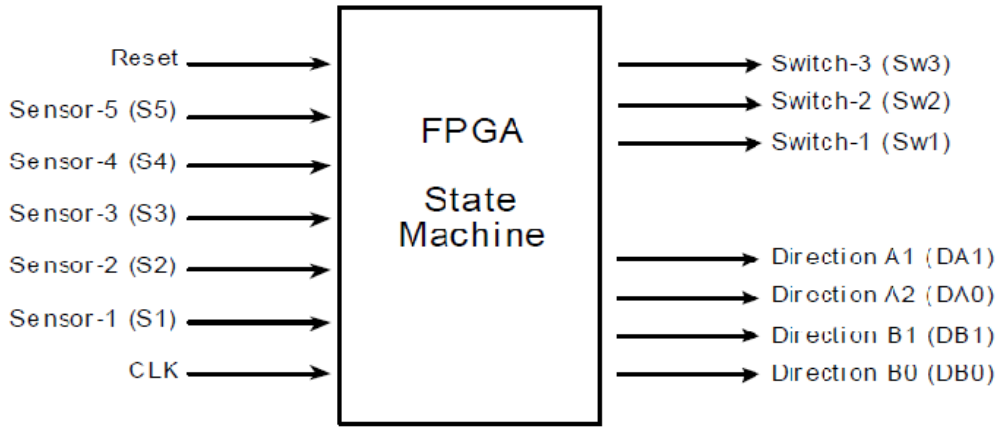
The five train sensor input signals (S1, S2, S3, S4, and S5) go high when a train is near the sensor location. It should be noted that sensors (S1, S2, S3, S4, and S5) *do not go high for only one clock cycle.* In fact, the sensors fire continuously for *many* clock cycles per passage of a train. This means that if your design is testing the same sensor from one state to another, you must wait for the signal to change from high to low.

As an example, if you wanted to count how many times that a train passes Sensor 1, you can not just have an "IF S1 GOTO count-one state" followed by "IF S1 GOTO count-two state." You would need to have a state that sees S1='1', then S1='0', then S1='1' again before you can be sure that it has passed S1 twice. If your state machine has two concurrent states that look for S1='1', the state machine will pass through both states in two consecutive clock cycles although the train will have passed S1 only once.

Another way would be to detect S1='1', then S4='1', then S1='1' if, in fact, the train was traversing the outside loop continuously. Either method will ensure that the train passed S1 twice.

Name of Candidate: _____

The state machine's signal inputs and outputs have been summarized in the following figure:




Sensor (S1, S2, S3, S4, S5) = 1 Train Present

= 0 Train not Present

Switches (SW1, SW2, SW3) = 0 Connected to Outside Track

= 1 Connected to Inside Track

Train Direction (DA1-DA0) and (DB1-DB0) = 00 Stop

= 01 Forward (Counterclockwise)

= 10 Backward (Clockwise)

**Figure 8.4** Train Control State Machine I/O Configuration.

## 8.5 An Example Controller Design

We will now examine a working example of a train controller state machine. For this controller, two trains run counterclockwise at various speeds and avoid collisions. One Train (A) runs on the outer track and the other train (B) runs on the inner track. Only one train at a time is allowed to occupy the common track.

Both an ASM chart and a classic state bubble diagram are illustrated in Figures 8.5 and 8.6 respectively. In the ASM chart, state names, ABout, Ain, Bin, Bstop, and Astop indicate the active and possible states. The rectangles contain

Name of Candidate: _____

the active (High) outputs for the given state. Outputs not listed are always assumed to be inactive (Low).

The diamond shapes in the ASM chart indicate where the state machine tests the condition of the inputs (S1, S2, etc.). When two signals are shown in a diamond, they are both tested at the same time for the indicated values.

A state machine classic bubble diagram is shown in Figure 8.6. Both Figures 8.5 and 8.6 contain the same information. They are simply different styles of representing a state diagram. The track diagrams in Figure 8.7 show the states visually. In the state names, "in" and "out" refer to the state of track 2, the track that is common to both loops.

## Description of States in Example State Machine

**All States**

- All signals that are not "Asserted" are zero and imply a logical result as described.

**ABout:** "Trains A and B Outside"

- DA0 Asserted: Train A is on the outside track and moving counterclockwise (forward).
- DB0 Asserted: Train B is on the inner track (not the common track) and also moving forward.
- Note that by NOT Asserting DA1, it is automatically zero -- same for DB1. Hence, the outputs are DA – "01" and DB – "01".

**Ain:** "Train A moves to Common Track"

- Sensor 1 has fired either first or at the same time as Sensor 2.
- Either Train A is trying to move towards the common track, or
- Both trains are attempting to move towards the common track.
- Both trains are allowed to enter here; however, state Bstop will stop B if both have entered.
- DA0 Asserted: Train A is on the outside track and moving counterclockwise (forward).
- DB0 Asserted: Train B is on the inner track (not the common track) and also moving forward.

**Bstop:** "Train B stopped at S2 waiting for Train A to clear common track"

- DA0 Asserted: Train A is moving from the outside track to the common track.
- Train B has arrived at Sensor 2 and is stopped and waits until Sensor 4 fires.
- SW1 and SW2 are NOT Asserted to allow the outside track to connect to common track.

**Bin:** "Train B has reached Sensor 2 before Train A reaches Sensor 1"

- Train B is allowed to enter the common track. Train A is approaching Sensor 1.
- DA0 Asserted: Train A is on the outside track and moving counterclockwise (forward).
- DB0 Asserted: Train B is on the inner track moving towards the common track.
- SW1 Asserted: Switch 1 is set to let the inner track connect to the common track.
- SW2 Asserted: Switch 2 is set to let the inner track connect to the common track.

**Astop:** "Train A stopped at S1 waiting for Train B to clear the common track"

- DB0 Asserted: Train B is on the inner track moving towards the common track.
- SW1 and SW2 Asserted: Switches 1 and 2 are set to connect the inner track to the common track.

Name of Candidate: _____

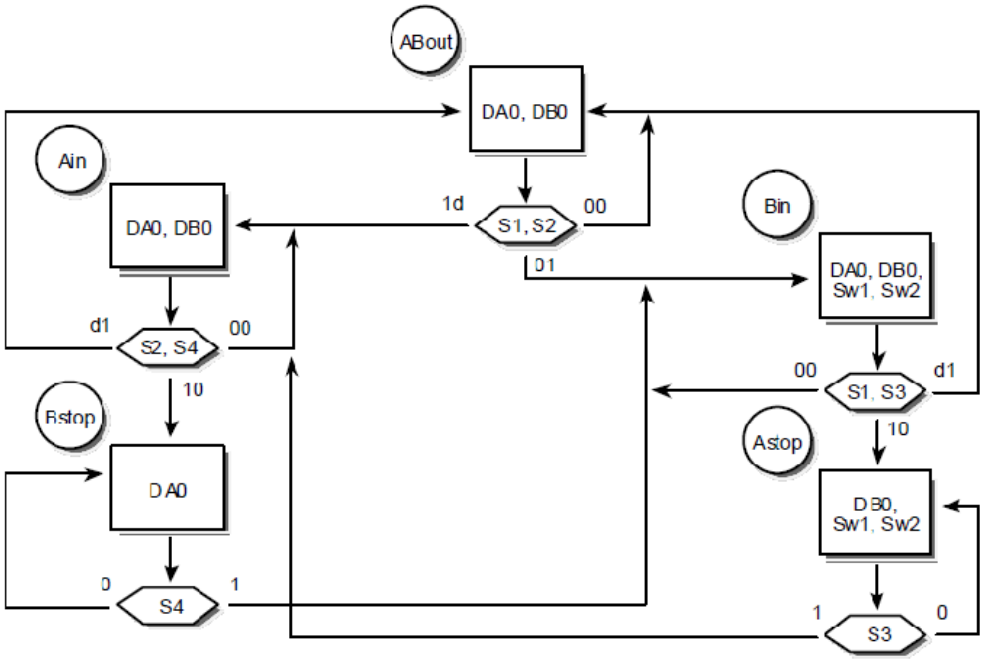



**Figure 8.5** Example Train Controller ASM Chart.

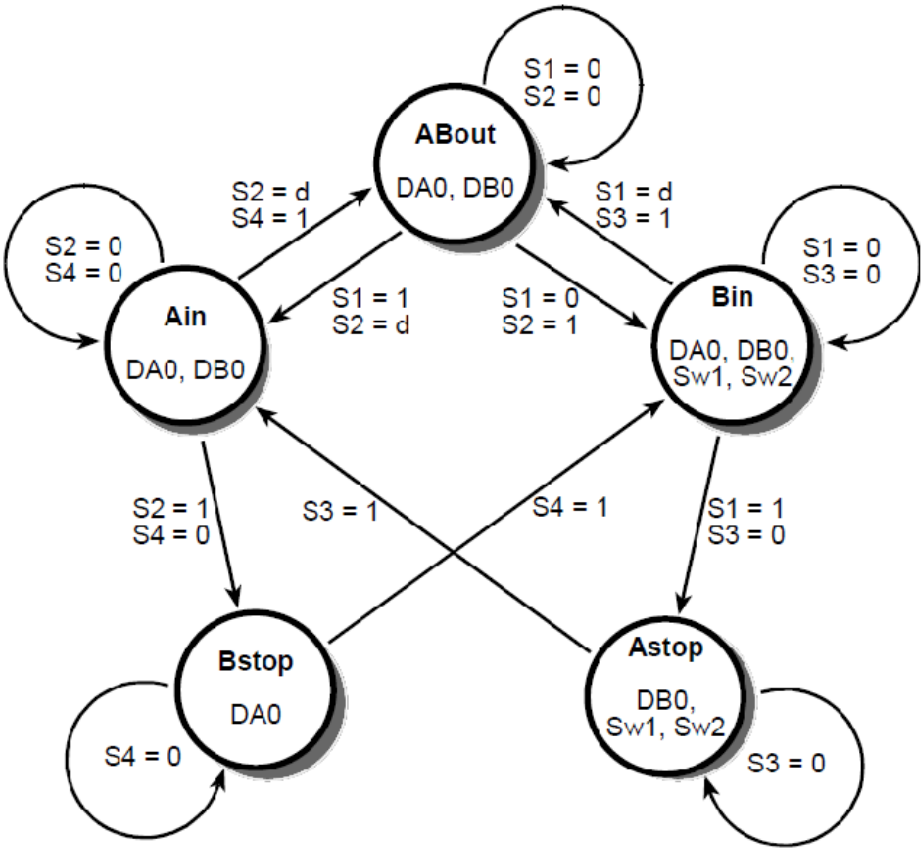



**Figure 8.6** Example Train Controller State Diagram.

Name of Candidate: _____

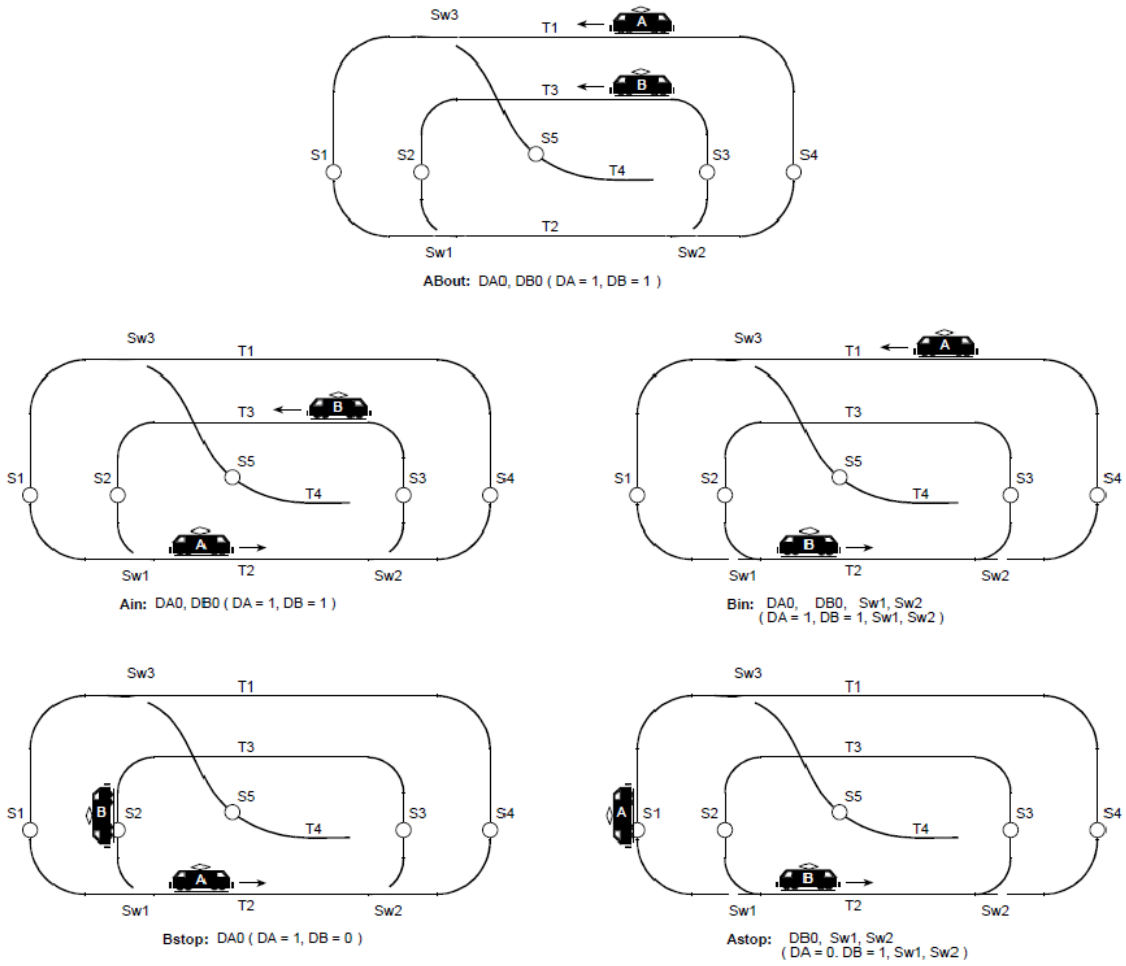



**Figure 8.7** Working diagrams of train positions for each state.

Table 8.1 Outputs corresponding to states.

| State | ABout | Ain | Astop | Bin | Bstop |
|---|---|---|---|---|---|
| Sw1 | 0 | 0 | 1 | 1 | 0 |
| Sw2 | 0 | 0 | 1 | 1 | 0 |
| Sw3 | 0 | 0 | 0 | 0 | 0 |
| DA(1-0) | 01 | 01 | 00 | 01 | 01 |
| DB(1-0) | 01 | 01 | 01 | 01 | 00 |

Name of Candidate: _____

## 8.8 Automatically Generating a State Diagram of a Design

You can use **Tools ⇨Netlist Viewers ⇨State Diagram Viewer** to automatically generate a state diagram and state table of a VHDL or Verilog based state machine after it has been compiled successfully as seen in Figure 8.8. The encoding tab at the bottom will also display the state encodings which typically use the one-hot encoding scheme (i.e., one flip-flop is used per state and the active flip-flop indicates the current state).

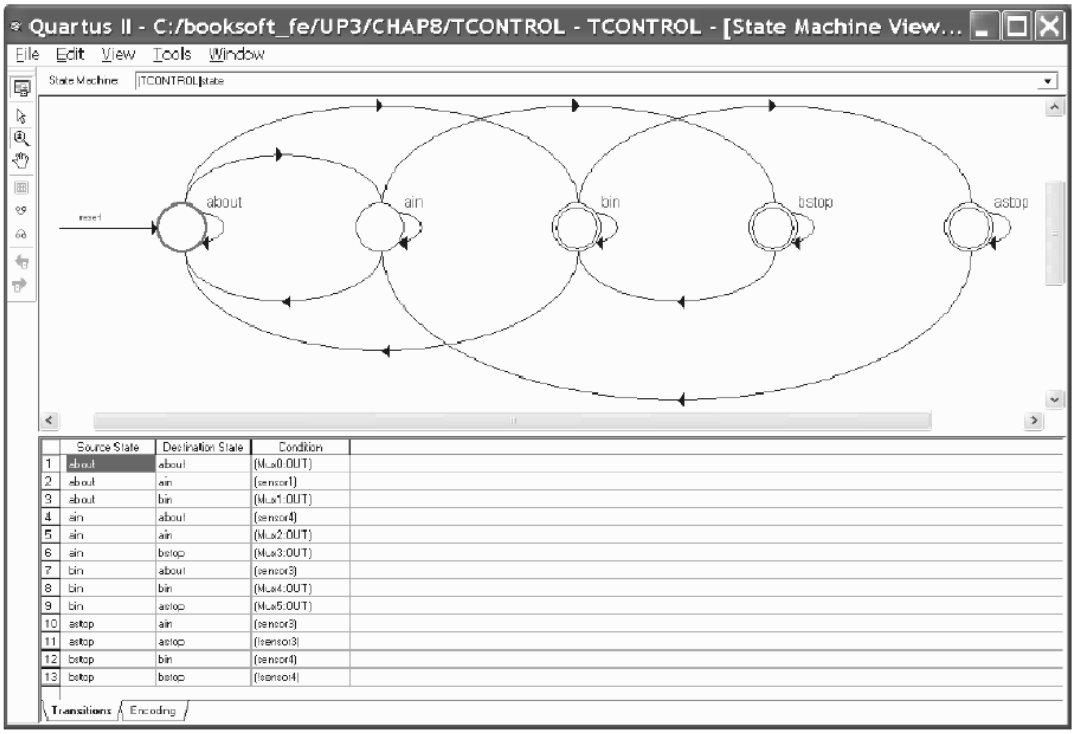



**Figure 8.8** Automatically generated state diagram of Tcontrol.vhd.

Name of Candidate: _____

## 8.9 Simulation Vector file for State Machine Simulation

The vector waveform file, *tcontrol.vwf*, seen in Figure 8.9 controls the simulation and tests the state machine. A vector waveform file specifies the simulation stimulus and display. This file sets up a 40ns clock and specifies sensor patterns (inputs to the state machine), which will be used to test the state machine. These patterns were chosen by picking a path in the state diagram that moves to all of the different states.

The sensor-input patterns will need to be changed if you change to a different train pattern, and therefore, the state machine. Sensor inputs should not change faster than the clock cycle time of 40ns. As a minimum, try to test all of the states and arcs in your state machine simulation.

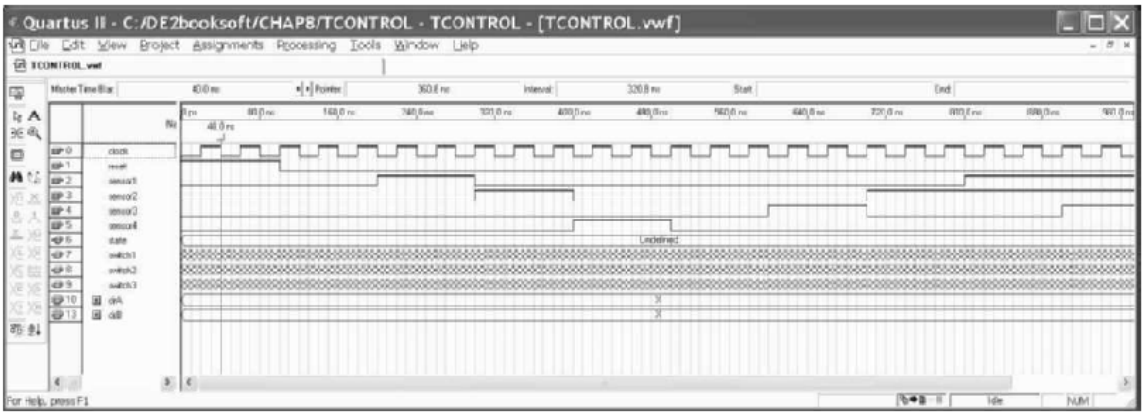



**Figure 8.9** Tcontrol.vwf vector waveform file for simulation.

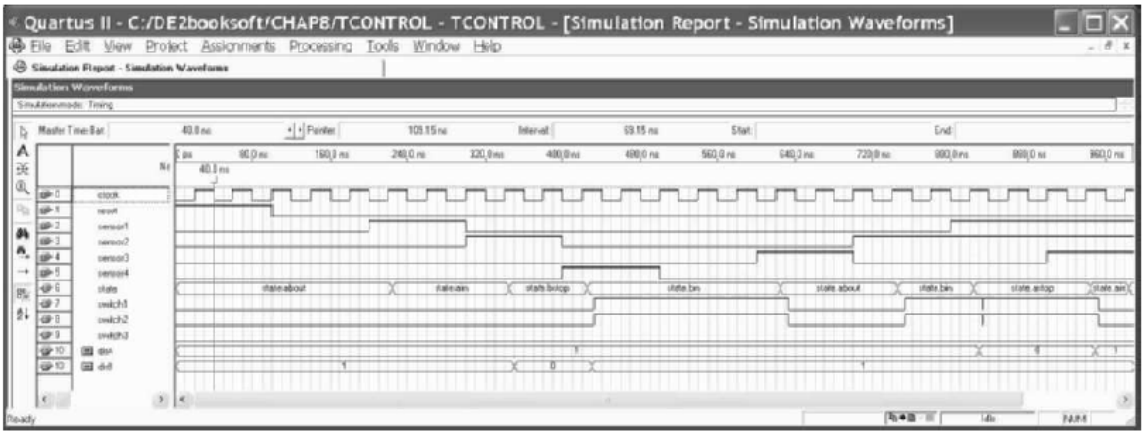



**Figure 8.10** Simulation of Tcontrol.vhd using the Tcontrol.vwf vector waveform file in Figure 8.9.

Name of Candidate: _____

## 8.10 Running the Train Control Simulation

Follow these steps to compile and simulate the state machine for the electric train controller.

### Select Current Project

Make Tcontrol.vhd the current project with **File ⇨Open Project ⇨*Name***

Then find and select **Tcontrol.vhd**.

### Compile and Simulate

Select **Processing ⇨Start Compilation and Simulation**. The simulator will run automatically if there are no compile errors. Select **Processing ⇨Simulation Report** to see the timing diagram display of your simulation as seen in Figure 8.10. Whenever you change your VHDL (or Verilog) source you will need to repeat this step. If you get compile errors, clicking on the error will move the text editor to the error location. The Altera software has extensive online help including HDL syntax examples.

Make any text changes to Tcontrol.vhd or Tcontrol.vwf (test vector waveform file) with **File ⇨Open**. This brings up a special editor window. Note that the menus at the top of the screen change depending on which window is currently open.

### Updating new Simulation Test Vectors

To update the simulation with new test vectors from a modified Tcontrol.vwf, select **Processing ⇨Start Simulation**. The simulation will then run with the new test vectors. If you modify Tcontrol.vhd, you will need to recompile first.

## 8.11 Running the Video Train System (After Successful Simulation)

A simulated or "virtual" train system is provided to test the controller without putting trains and people at risk. The simulation runs on the FPGA chip. The output of the simulation is displayed on a VGA monitor connected directly to the FPGA board. A typical video output display is seen in Figure 8.11. This module is also written in VHDL and it provides the sensor inputs and uses the outputs from the state machine to control the trains. The module tcontrol.vhd is automatically connected to the train simulation.

Here are the steps to run the virtual train system simulation:

### Select the top-level project

Make Train.vhd the current project with **File ⇨Open Project ⇨*Name***

Then find and select **Train.qpf**. Train.qsf must be in the project directory since it contains the FPGA chip pin assignment information needed for video outputs and switch inputs. Double check that your FPGA Device type is correct.

Name of Candidate: _____

**Compile the Project**

Select **Processing ⇨Start Compilation**. Train.vhd will link in your tcontrol.vhd file if it is in the same directory, when compiled. This is a large program, so it will take a few seconds to compile.

**Download the Video Train Simulation**

Select **Tools ⇨Programmer**. When the programmer window opens click on the Program/Configure box if it is not already selected. In case of problems, see the FPGA board download tutorials in Chapter 1 for more details. The FPGA board must be turned on the power supply must be connected, and the Byteblaster* cable must be plugged into the PC. When everything is setup, the start button in the programming window should highlight. If the start button is not highlighted, try closing and reopening the programmer window. Under Hardware setup the Byteblaster should be selected. To download the board, click on the highlighted **start** button. Attach a VGA monitor to the FPGA board.

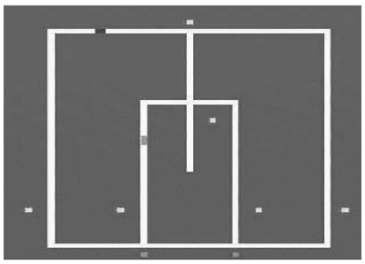


**Figure 8.11** Video Image from Train System Simulation.

**Viewing the Video Train Simulation**

Train output should appear on the VGA monitor after downloading is complete as seen in Figure 8.11. On the DE1 and DE2, FPGA KEY2 is run/step/stop and FPGA KEY1 is the reset. Train A is displayed in black and Train B is displayed

Name of Candidate: _____

in red. On the DE1 and DE2, hit KEY2 once to start the train simulation running. Hitting KEY2 again will stop the simulation. If you hit KEY2 twice quickly while trains are stopped, it will single step to the next track sensor state change. Other boards also use two pushbuttons for these functions.

Sensor and switch values are indicated with a green or red square on the display. Switch values are the squares next to each switch location. Green indicates no train present on a sensor and it indicates switch connected to outside track for a switch.

On the DE2 and UP3 board's, the LCD display top line shows the values of the sensor (s), and switch (sw) signals in binary and the bottom line indicates the values of DirA and DirB in binary. The most significant bit in each field is the highest numbered bit.

If a possible train wreck is detected by two trains running on the same track segment, the simulation halts and the monitor will flash. The FPGA board's slide or DIP switches control the speed of Train A (low 2 bits) and B (high 2 bits). Be sure to check operation with different train speeds. Many problems occur more often with a fast moving and a slow moving train.

## 8.12 A Hardware Implementation of the Train System Layout

Using the new Digital Command Control (DCC) model trains[3], a model train system with a similar track layout and control scheme can be setup and controlled by the FPGA board[4]. In DCC model trains, the train speed, direction, and other special features are controlled via a bipolar bit stream that is transmitted on the train tracks along with the power. A DCC decoder is located inside each train's engine that interprets the DCC signals and initiates the desired action (i.e., change in speed, direction, or feature status).

On a DCC system, trains are individually addressable. As seen in Figure 8.12, the DCC signal's frequency or zero crossing rate is changed in the DCC signal to transmit the data bits used for a command. A DCC command contains both a train address and a speed command. Each train engine is assigned a unique address. The electric motors in the train's engine are powered by a simple diode circuit that provides full–wave rectification of the bipolar DCC signal that is present on the track. In this way, the two metal train tracks can simultaneously provide both direct current (DC) power and speed control commands for the trains.

The output voltages and current levels provided by an FPGA output pin cannot drive the DCC train signals directly, but an FPGA can send the DCC data streams to the train track with the addition of a higher current H–bridge circuit that controls the train's power supply. An H-bridge contains four large power transistors that provide the higher drive current needed for DC motors and they

[3] DCC standards are approved by the National Model Railroad Association and are available online at http://www.nmra.org/standards/DCC/standards_rps/DCCStds.html.

[4] Additional details on using FPGAs for DCC can be found in "Using the Using FPGAs to Simulate and Implement Digital Design Systems in the Classroom", by T. S. Hall and J. O. Hamblen in the *Proceedings of the 2006 ASEE Southeast Section Conference*.

Name of Candidate: _____

can also reverse the motor. Integrated H–bridge modules are available that can minimize the number of discrete components used. One such example is the National Semiconductor LMD18200 integrated H–bridge module. The LMD18200 supports TTL and CMOS compatible inputs allowing the FPGA board's output pins to be connected directly to the H-bridge inputs. A H-bridge typically requires two digital input control pins (i.e., forward, reverse, and stop). The H-bridge switches the train's power supply and in addition to the FPGA output pins that drive the H-bridge inputs, a ground connection is required between the train's power supply and the FPGA power supply.

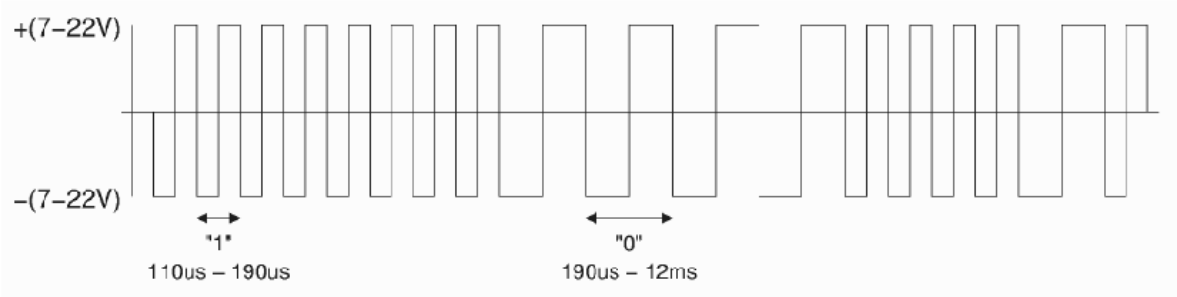



**Figure 8.12** A portion of a DCC train signal is seen above. The zero crossing rate of a DCC signal is used to send data bits for train speed commands. The DCC signal is also rectified in each train's engine to provide 7-22V DC power for the train's electric motor and decoder circuits.

For the train sensors, Sharp GP2L26 infrared (IR) photointerrupter sensors can be used to detect when a train passes each sensor point. These sensors emit IR light from an LED and detect when the light is reflected back with an IR detector circuit. These sensors are very small (3mm x 4mm) and can fit between the rails on the track. Wires can be run down through the roadbed to a central protoboard where the discrete components needed for interfacing to this sensor are connected. Many model trains have dark underbodies, and the IR photo sensors can not always detect the trains passing over them. To increase the visibility of the trains to the photo sensors, pieces of reflective tape can be taped to the bottom of the trains.

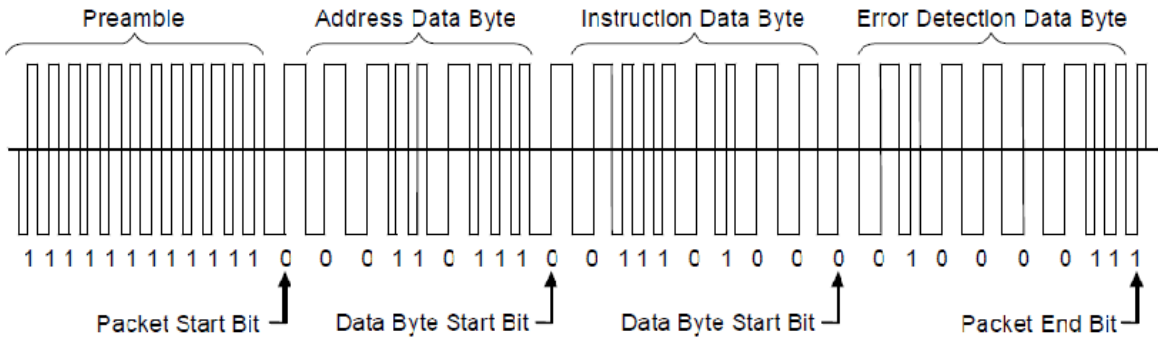



**Figure 8.13** An example DCC model train speed and direction command packet.

Name of Candidate: _____

Another version of the train.VHD program, dcc_train.zip is available on the book's DVD that will control a DCC train setup. It replaces the video train simulation module and produces the outputs needed to run the real DCC train system. It contains a DCC IP core, DCC_low_level.vhd, which generates the appropriate DCC signal packets as seen in Figure 8.13. The DCC data stream is generated by combining the Speed switch inputs and the Train direction signals (i.e., DA, DB) from the Tcontrol module. The appropriate DCC command packet is created from these signals and then saved in a register. The registered command is shifted out to produce a serial bit stream. The DCC standard only provides for one–way communication, and thus, no transmission guarantee can be made (i.e., no acknowledgement is sent back by the train). Therefore, a given DCC command is repeatedly shifted out until another command is received to ensure transmission of each command. Continuous transmission also insures a consistent power level on the tracks.

Additional construction details for anyone building the FPGA-based DCC model train setup are available at the book's website. The FPGA uses five input pins to read in from the IR photointerrupter track sensors, two output bits to send DCC commands, and two output bits to control each of the three track switches. Each track switch has two solenoid drivers that open and close a switch. The proper solenoid must be briefly turned on or pulsed to move the switch and then turned off. Leaving the solenoid turned on continuously will overheat and eventually burn out the solenoid. The 50ms timed pulse required to briefly energize a track switch's solenoid is already provided in the IP core.

To connect the train setup to all of the FPGA I/O pins, a ribbon cable can be attached to one of the I/O expansion headers on the FPGA board with the other end attached to the train interface circuitry on a protoboard or custom printed circuit board (PCB). The FPGA device type and I/O pin assignments for the train.VHD project will need to be changed depending on each user's custom train interface circuitry and the FPGA board I/O expansion connector used. Consult each FPGA board's reference manual for complete details on the I/O expansion header's FPGA pin numbers.

## 8.13 Laboratory Exercises

1. Assuming that train A now runs clockwise and B remains counterclockwise, draw a new state diagram and implement the new controller. If you use VHDL to design the new controller, you can modify the code presented in section 8.7. Simulate the controller and then run the video train simulation.

2. Design a state machine to operate the two trains avoiding collisions but minimizing their idle time. Trains must not crash by moving the wrong direction into an open switch. Develop a simulation to verify your state machine is operating correctly before running the video train system. The trains are assumed to be in the initial positions as shown in Figure 8.14. Train A is to move counterclockwise around the outside track until it comes to Sensor 1, then move to the inside track stopping at Sensor 5 and waiting for B to pass Sensor 3 twice. Trains can move at different speeds, so no assumption should be made

Name of Candidate: _____

about the train speeds. A train hitting a sensor can be stopped before entering the switch area.

Once B has passed Sensor 3 twice, Train A moves to the outside track and continues around counterclockwise until it picks up where it left off at the starting position as shown in Figure 8.15. Train B is to move as designated only stopping at a sensor to avoid collisions with A. Train B will then continue as soon as there is no potential collision and continue as designated. Trains A and B should run continuously, stopping only to avoid a potential collision.

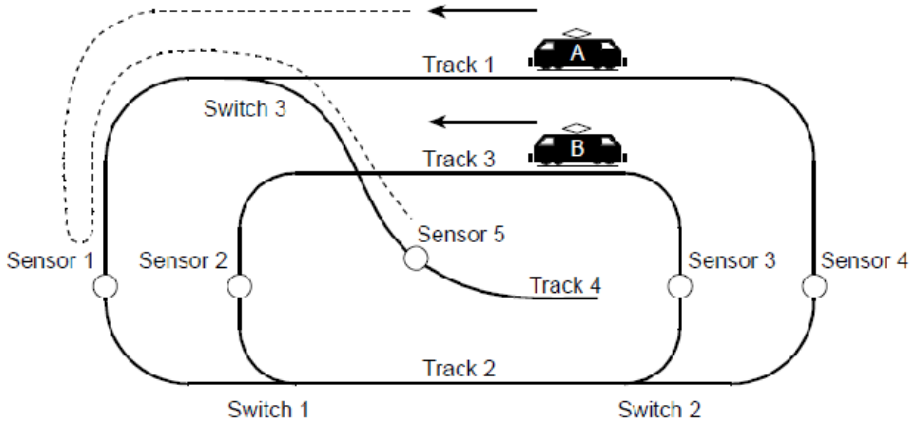



**Figure 8.14** Initial Positions of Trains at State Machine Reset with Initial Paths Designated.

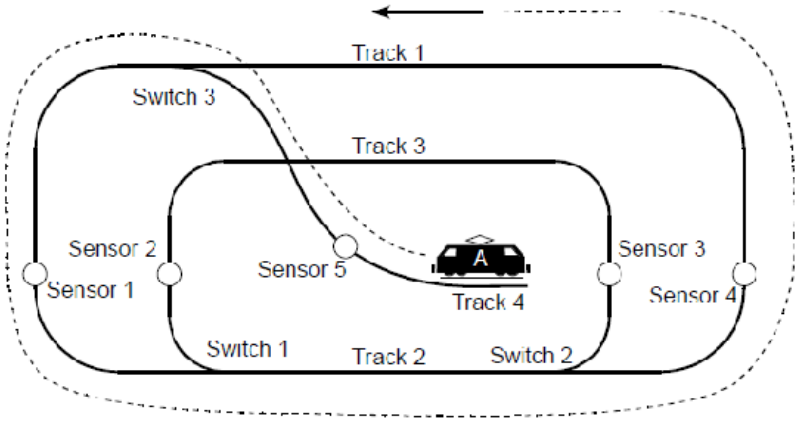



**Figure 8.15** Return Path of Train A.

3. Use the single pulse FPGAcore functions on each raw sensor input to produce state machine sensor inputs that go High for only one clock cycle per passage of a train. Rework the state machine design with this assumption and repeat problem 1 or 2.
4. Develop another pattern of train movement and design a state machine to implement it.
5. Implement a real train setup using DCC model trains. Debug your control module using the video simulation module first, to avoid any real train crashes that may damage the trains. Typically laboratory space is limited, so keep in mind that the smaller gauge model trains will require less space for the track layout.

Name of Candidate: _____

# LAB 8:

## MEMORY BLOCKS:

## Register Files (RF)

DURATION: 2 HOURS

1. Construct a Register File as shown in Figure 8.25 using HDL code.
2. Simulate your design using HDL based Test Bench (Verify with sample waveform shown in Figure 8.28)
3. Implement your RF on the FPGA.

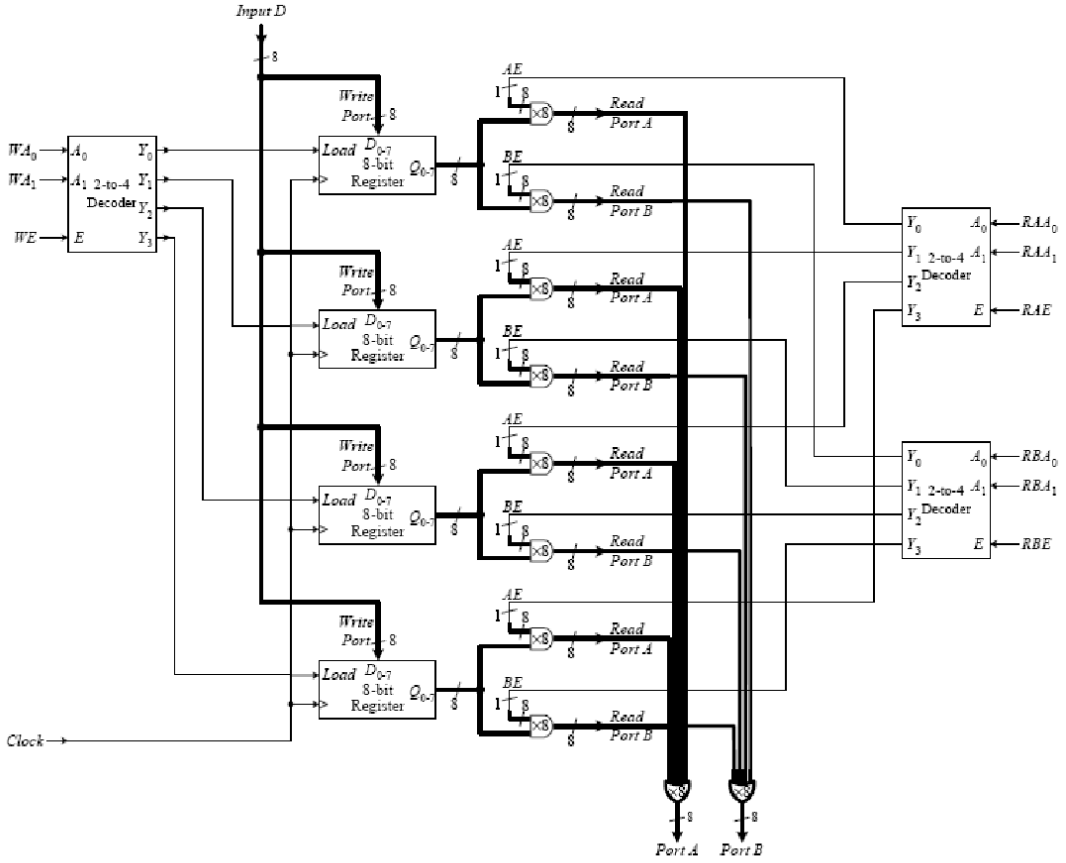



**Figure 8.25** A 4 × 8 register file circuit with one write port and two read ports.

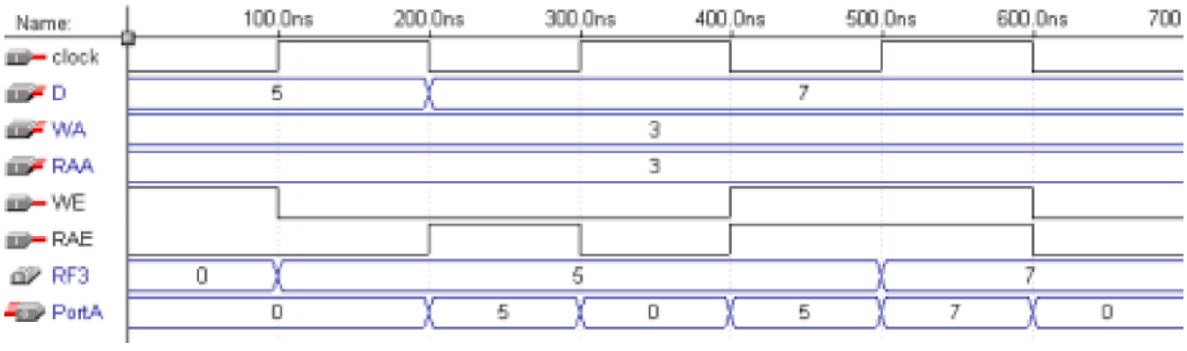



**Figure 8.28** Sample simulation trace for the 4 × 8 register file.

Name of Candidate: _____

# LAB 8B:

## 4 BY 4 RAM CHIP

DURATION: 2 HOURS

1. Design a 4 by 4 RAM chip as shown in Figure 8.31 & 8.32 using HDL based entry.
2. Simulate your design using HDL based Test Bench.
3. Implement on the FPGA.

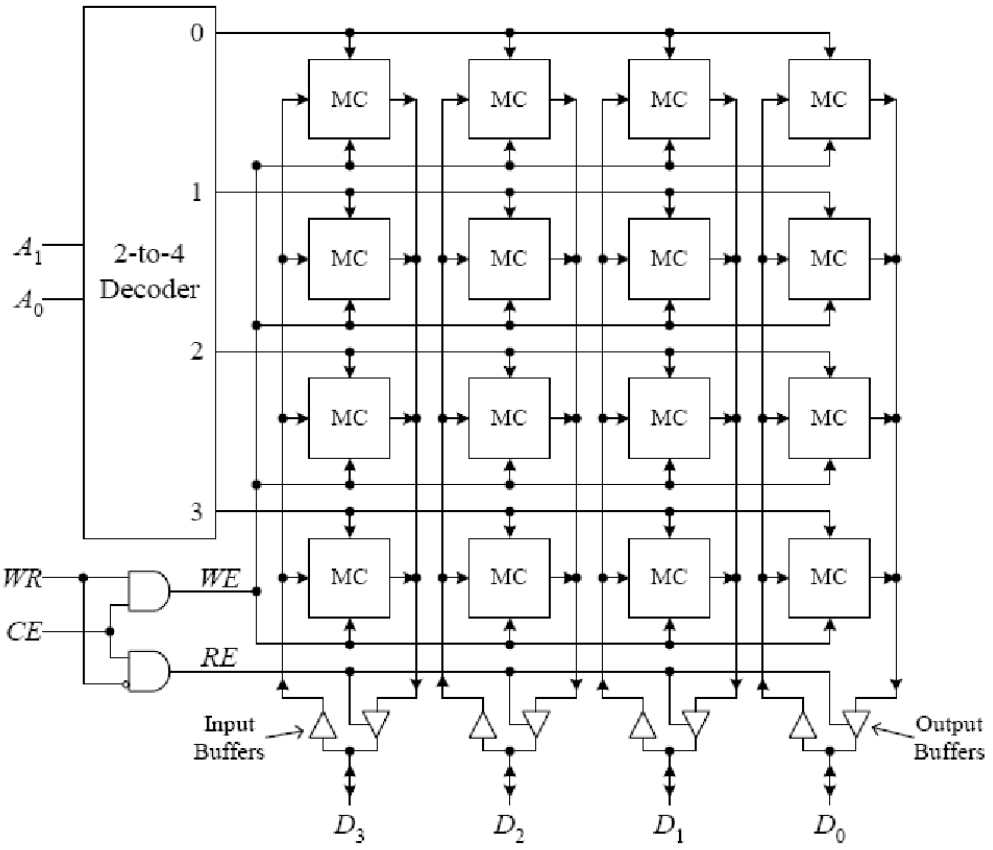



**Figure 8.32** A 4 × 4 RAM chip circuit.

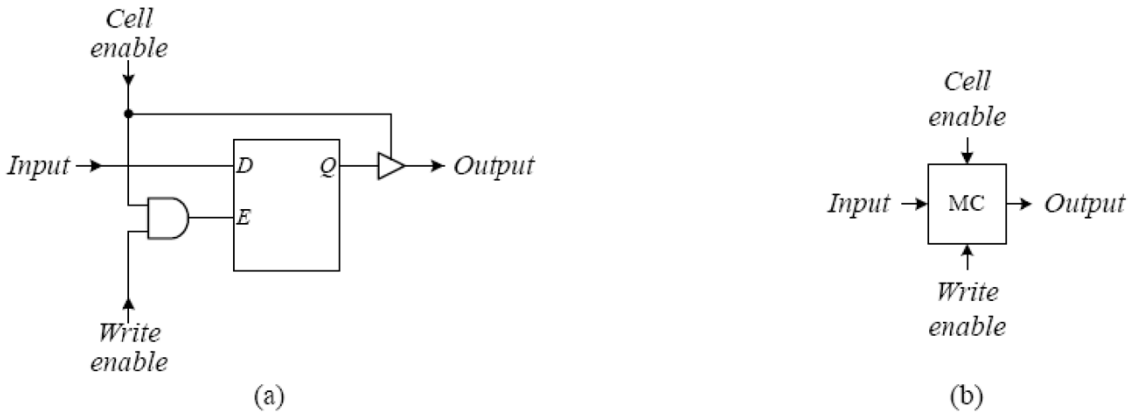



**Figure 8.31** Memory cell: (a) circuit; (b) logic symbol.

Name of Candidate: _____

# LAB 9/10/11:

## ASSIGMENT II:
## DEDICATED MICROPROCESSOR/ GENERAL PURPOSE PROCESSOR

Design, test, simulate, analyze and implement with creativity on your FPGA a dedicated microprocessor of choice complete with an integrated datapath and control unit.

GROUP WORK:-

1. Design and Test your HDL based ***datapath*** using a testbench in ModelSIM.
2. Implement your ***datapath*** design in the FPGA.
3. Design and Test your HDL based ***control unit*** using a testbench in ModelSIM.
4. Implement your ***control unit*** design in the FPGA.
5. Integrate and Test your HDL based dedicated ***microprocessor*** using a testbench in ModelSIM.
6. Implement your ***microprocessor*** design in the FPGA.
7. NO written report is required. Report will be in the form of an oral presentation.
8. Perform a presentation on your GROUP based work using POWER POINT slides in the following sequence/ format:-
    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 (FLOW CHART)
    g. RESULTS AND DISCUSSION (RESULT AND OUTCOME OF YOUR CASE STUDY AND HARDWARE IMPLEMENTATION)
    h. CONCLUSION (WRAP UP/ IMPROVEMENT/ FUTURE TRENDS)

*Your presentation is 15 minutes with 5/10 minutes of Q and A.
*You SHOULD NOT EXCEED 15 minutes.
*You MUST present as a group on Week 14.

Name of Candidate: _____

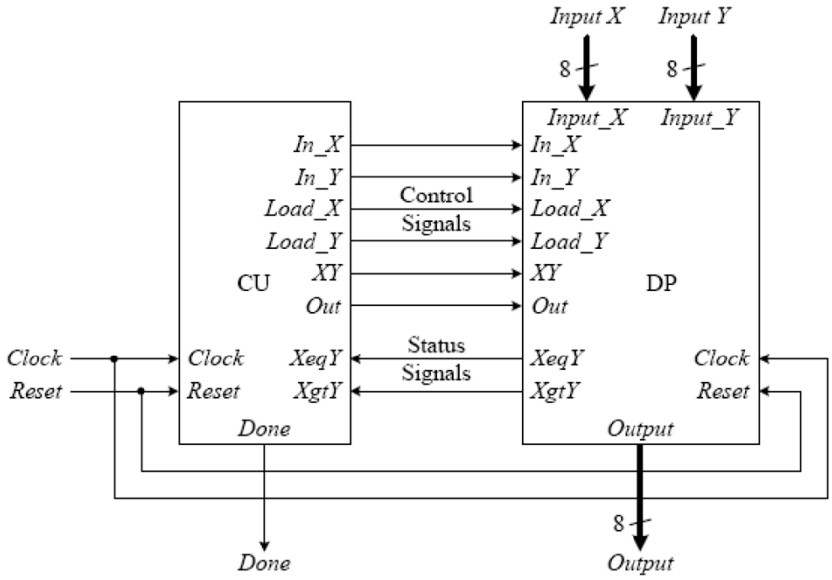
# LAB 9:

## A Dedicated Microprocessor
## GCD @ GREAT COMMON DIVISOR

Design, test, simulate and implement on your FPGA a dedicated microprocessor complete with a dedicated datapath & control unit integrated.

```
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
```

Algorithm for solving the GCD problem




Microprocessor for solving GCD problem

**GROUP WORK**

1. Design & Test your ***Datapath***
2. Design & Test your ***Control Unit***
3. Integrate Control Unit & Datapath with appropriate ***Control*** & ***Status Signals***.
4. Implement design on the FPGA & Shown simulation trace.

**Due date: Week 13**

Name of Candidate: _____

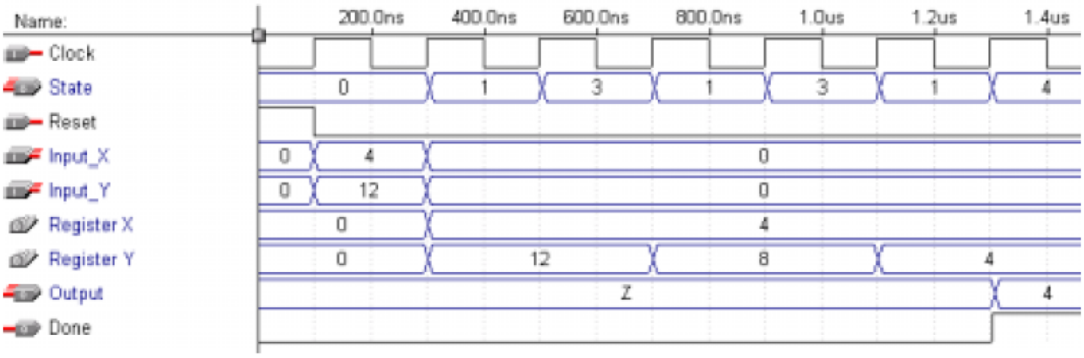



**Figure 11.7**. Sample simulation for the GCD problem for the two input numbers 4 and 12. The GCD of these two numbers is 4.

Name of Candidate: _____

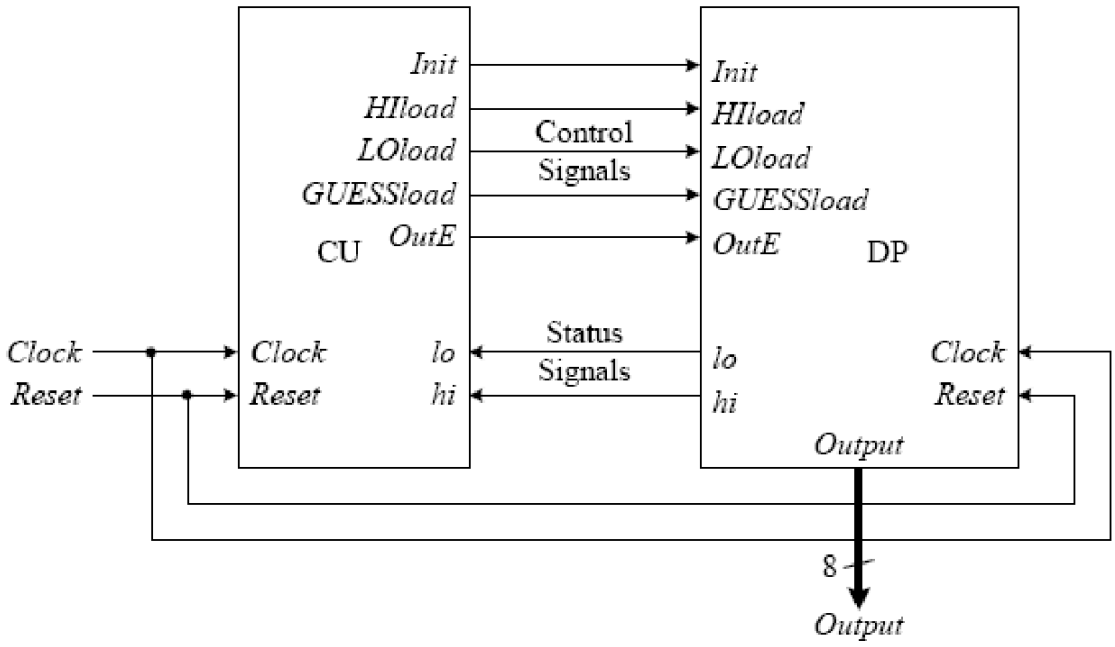
# LAB10:

## A Dedicated Microprocessor
## HIGH LOW GUESSING GAME

Design, test, simulate and implement on your FPGA a dedicated microprocessor complete with a dedicated datapath & control unit integrated.

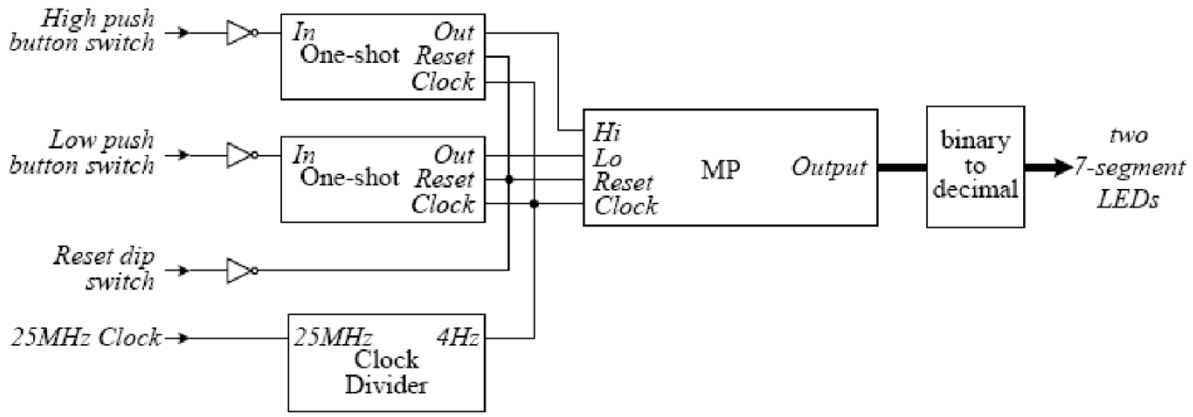
```
1     Low = 0                        // initialize Low
2     High = 100                     // initialize High
3     repeat {
4        Guess = (Low + High) / 2  // calculate guess using binary search
5        output Guess
6        if (lo_button = '1' and hi_button = '0')    // low button pressed
7           Low = Guess
8        else if (lo_button = '0' and hi_button = '1') // hi button pressed
9           High = Guess
10       end if
11    } until (lo_button = '1' and hi_button = '1')  // repeat until both
                                                     // buttons are pressed
12    while (lo_button = '0' and hi_button = '0')
         // blink correct guess
13       output Guess
14       turn off display
15    end while
```

Algorithm for solving the high-low guessing game




Microprocessor for solving high-low guessing game

Name of Candidate: _____




Suggested Interface for the High-Low guessing game microprocessor

**GROUP WORK:**

1. Design & Test your ***Datapath***
2. Design & Test your ***Control Unit***
3. Integrate Control Unit & Datapath with appropriate ***Control*** & ***Status Signals***.
4. Implement design on the FPGA

**DUE DATE: Week 13**

Name of Candidate: _____

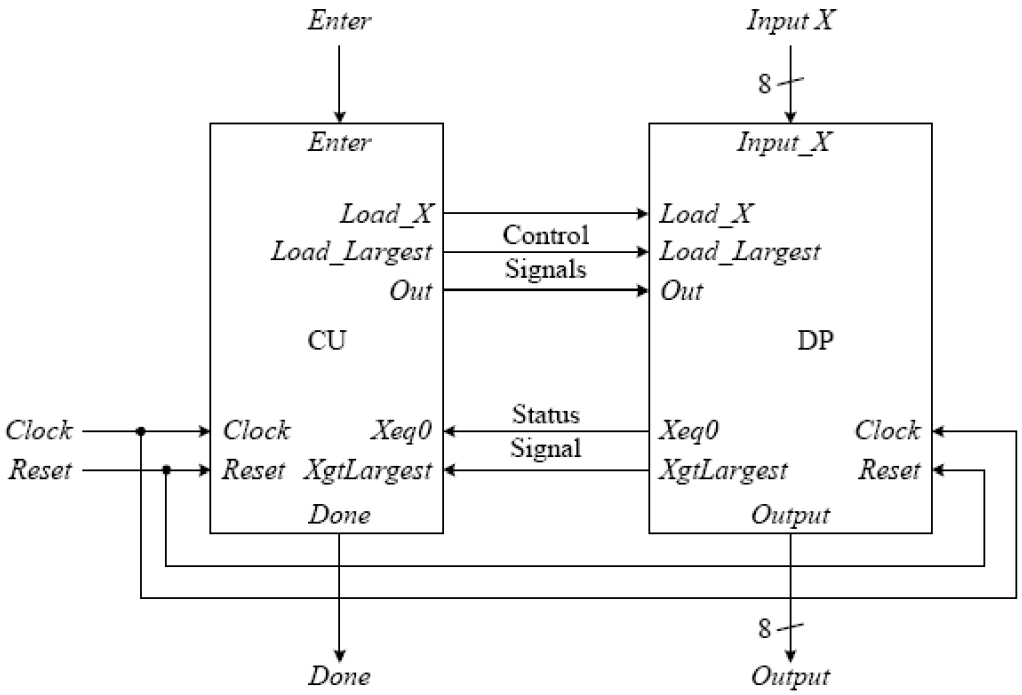
# LAB 11:

## A Dedicated Microprocessor
## FINDING THE LARGEST NUMBER

Design, test, simulate and implement on your FPGA a dedicated microprocessor complete with a dedicated datapath & control unit integrated.

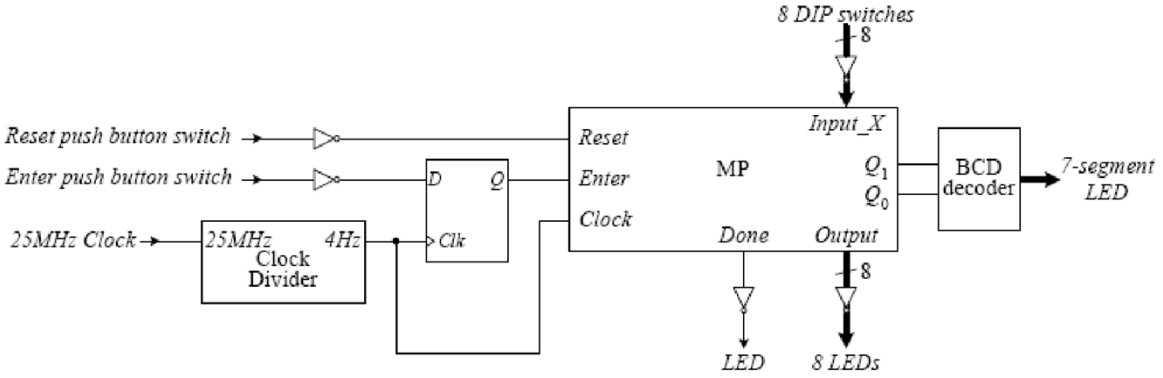
```
1     Largest = 0              // for storing the current largest number
2     input X                  // enter first number
3     while (X ≠ 0){
4        if (X > Largest) then     // if new number greater?
5           Largest = X            // yes, remember new largest number
6        end if
7        output Largest
8        input X                   // get next number
9     }
```

Algorithm for solving finding largest number problem




Microprocessor for solving finding largest number problem

Name of Candidate: _____




Suggested Interface for the finding largest number microprocessor

**GROUP WORK**

1. Design & Test your ***Datapath***
2. Design & Test your ***Control Unit***
3. Integrate Control Unit & Datapath with appropriate ***Control*** & ***Status Signals***.
4. Implement design on the FPGA & Show simulation trace

**Due date: WEEK 13**

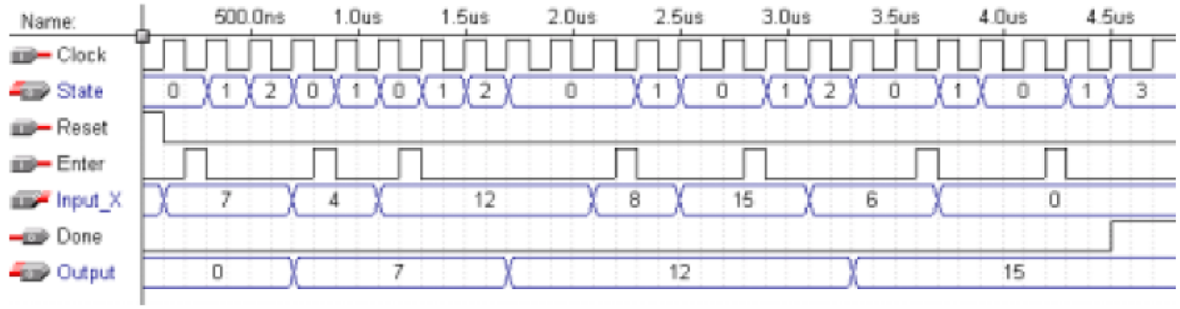



**Figure 11.24**. Sample simulation trace for the finding largest number problem. The last output number 15 when *Done* is asserted is the largest of the six numbers entered 7, 4, 12, 8, 15, and 6.

Name of Candidate: _____

# APPENDIX

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

## APPENDIX A: SAMPLE TEST BENCH CODE

Name of Candidate: _____

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

Name of Candidate: _____

# 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:**


- Design Project
- Creating Waveforms for Simulation
- Simulation
- Making Changes and Resimulating
- Concluding Remarks




## 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.

Name of Candidate: _____



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

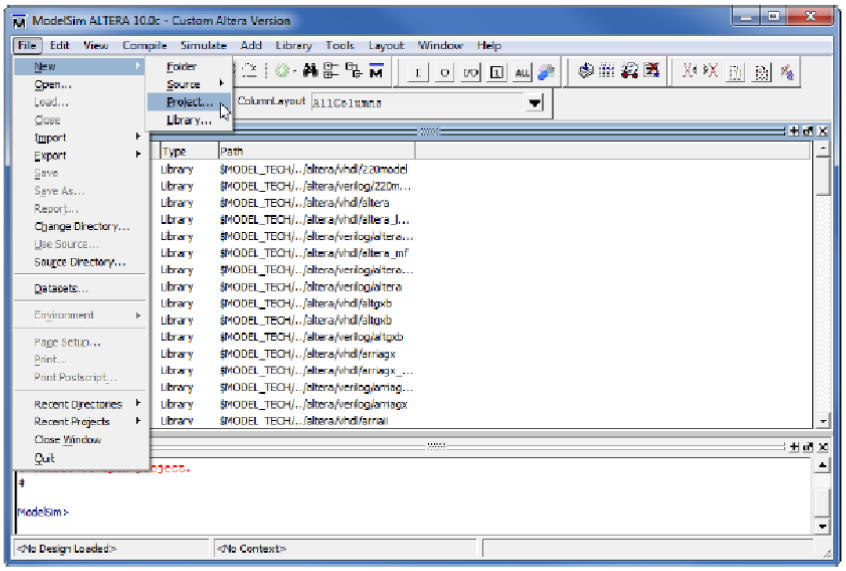



Figure 1. The ModelSim window.

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.

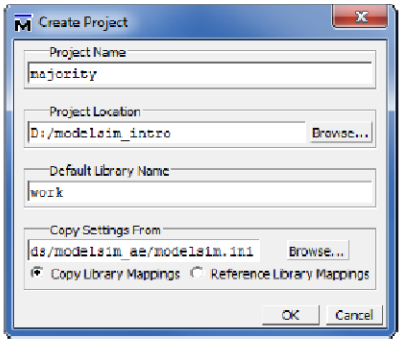



Figure 2. Created Project window.

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.

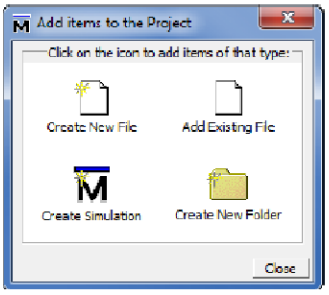



Figure 3. Add Items window.

Name of Candidate: _____

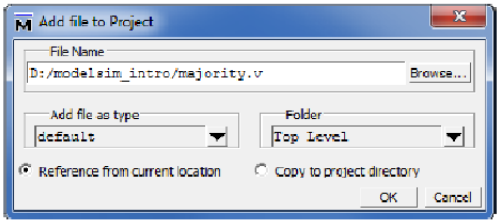



Figure 4. Add Items window.

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.

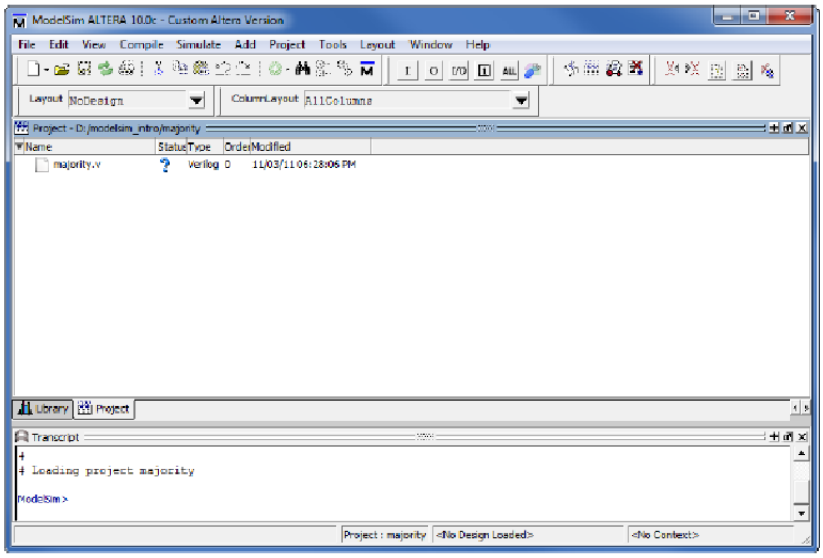



Figure 5. Updated ModelSim window.

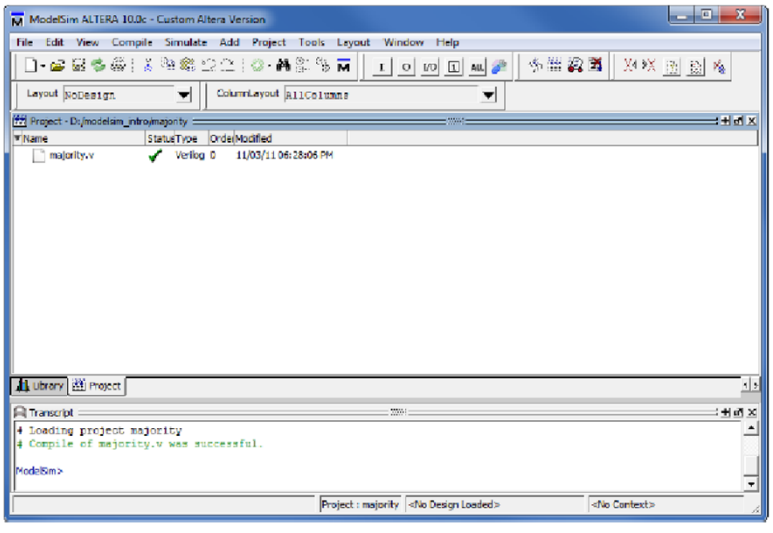



Figure 6. ModelSim window after compilation.

## 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.

Name of Candidate:

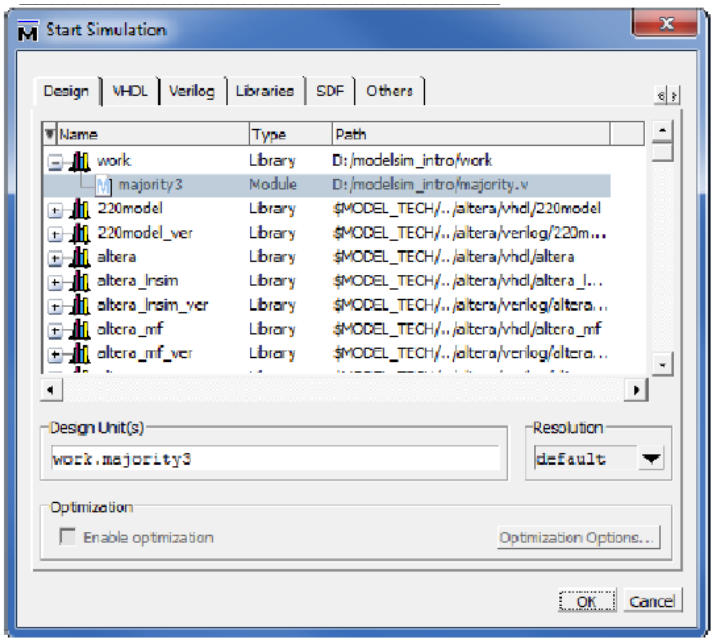



Figure 7. Start Simulation window.

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.

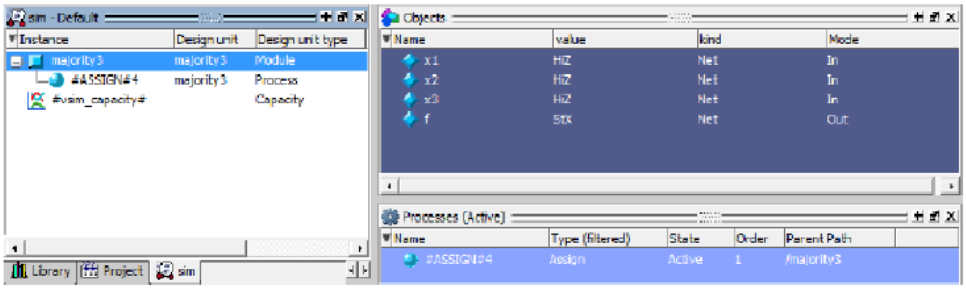



Figure 8. Signals in the Objects window.

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

Name of Candidate:

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.

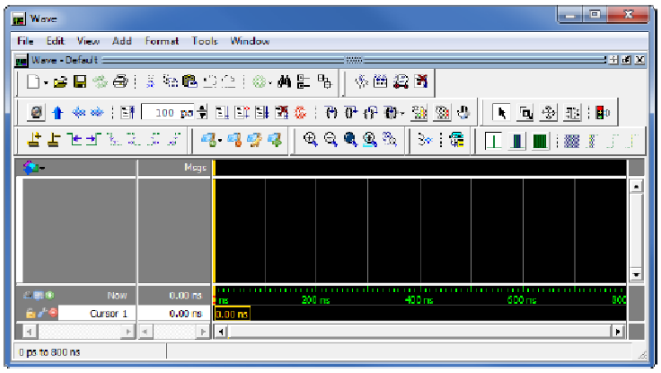



Figure 9. The Wave window.

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.

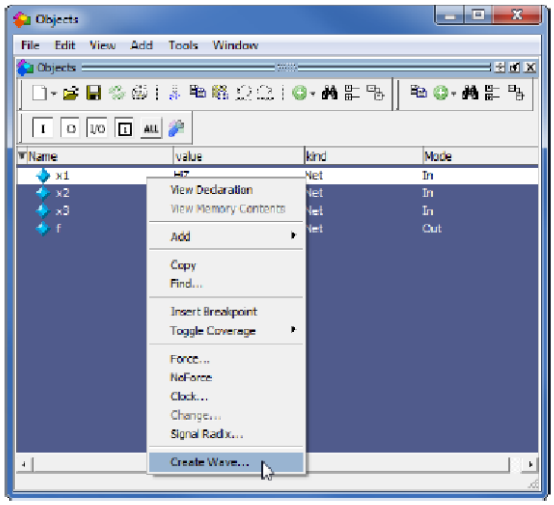



Figure 10. Selecting a signal in the Objects window.

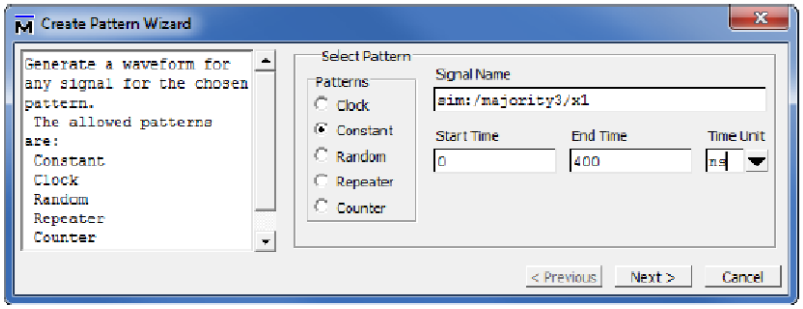



Figure 11. Specifying the type and duration of a signal.

Name of Candidate: _____

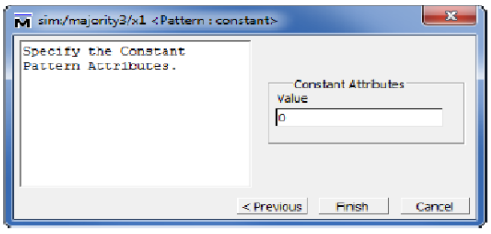



Figure 12. Specifying the value of a signal.

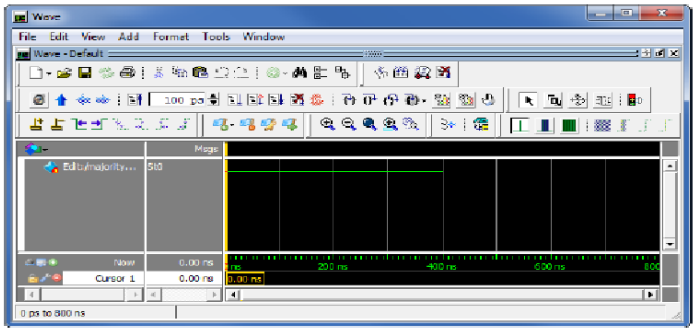



Figure 13. The updated Wave window.

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.

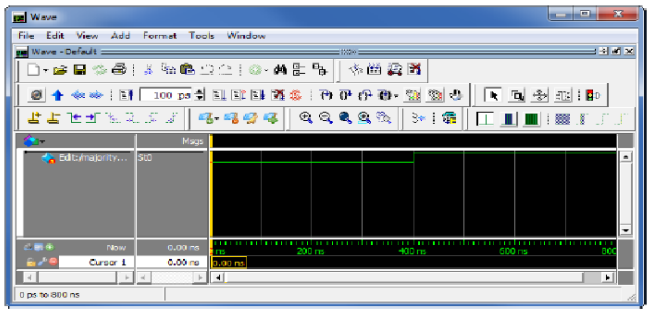



Figure 14. The completed waveform for $x_1$ input.

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.

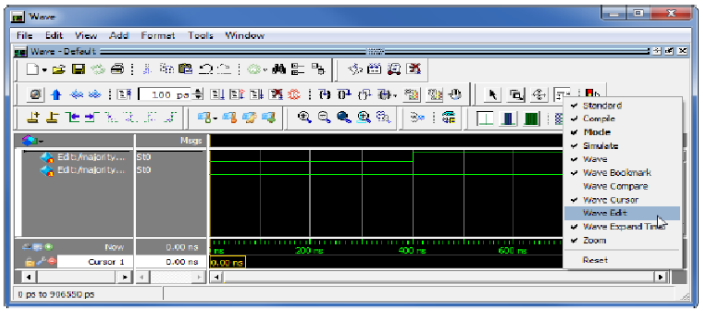



Figure 15. Selecting the Wave Edit mode.

Name of Candidate: _____

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.

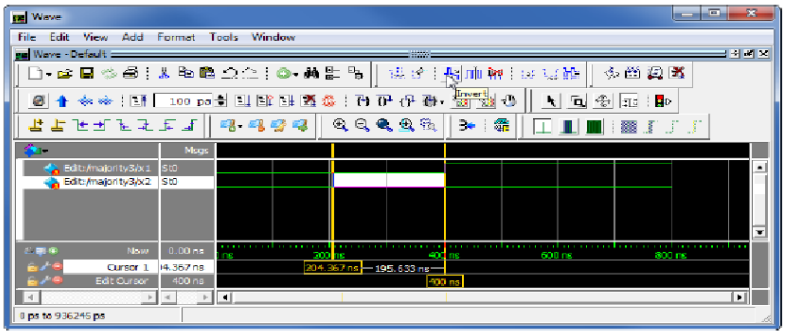



Figure 16. Editing the waveform.

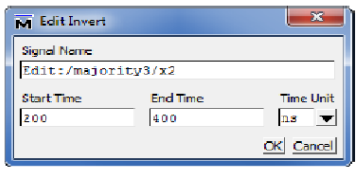



Figure 17. Specifying the exact time interval.

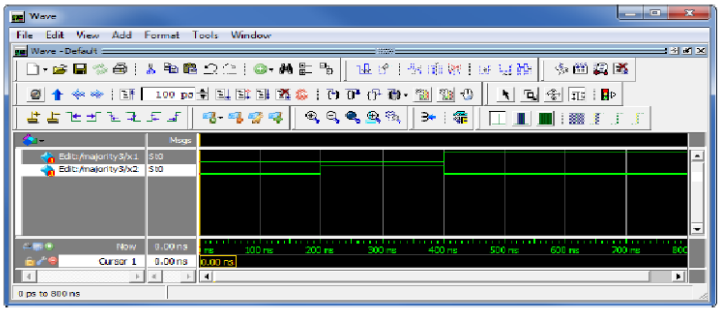



Figure 18. The modified waveform.

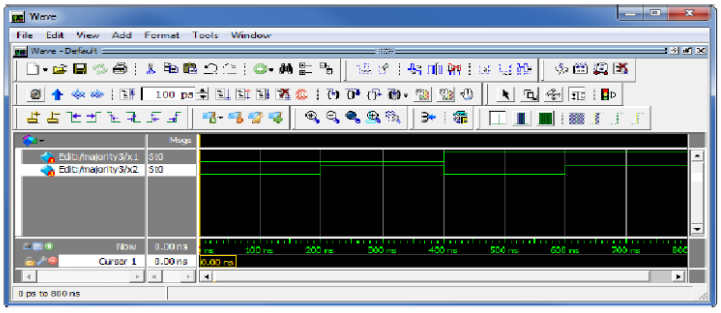



Figure 19. Completed waveforms for $x_1$ and $x_2$.

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.

Name of Candidate: _____

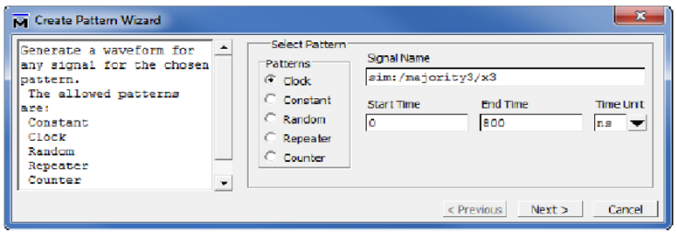



Figure 20. Selecting a signal of clock type.

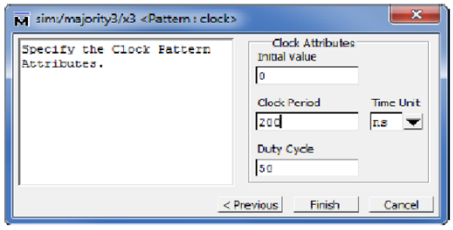



Figure 21. Defining the characteristics of a clock signal.

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.

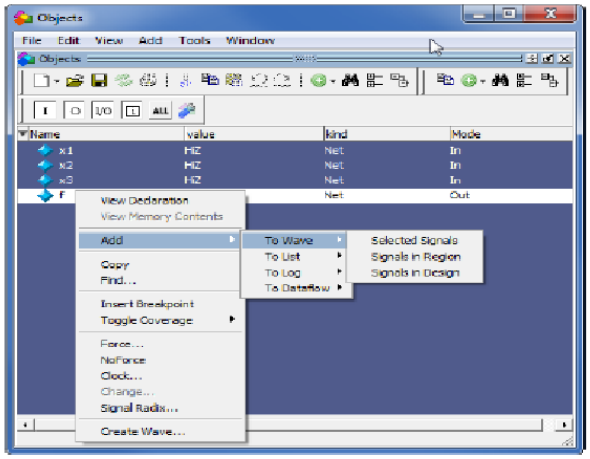



Figure 22. Adding a signal to the Wave window.

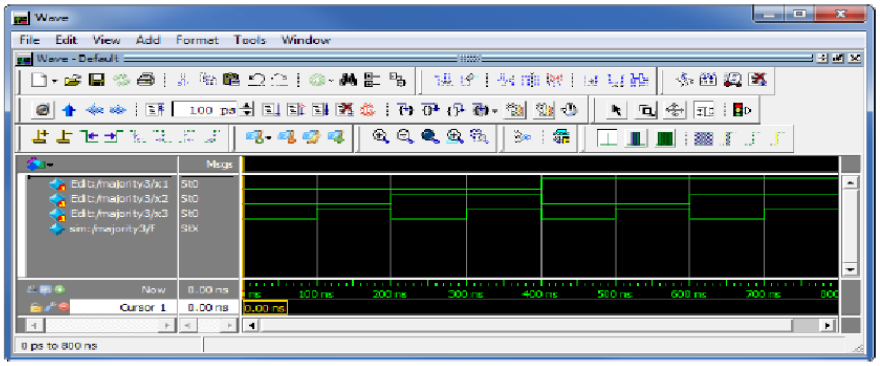



Figure 23. The completed Wave window.

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

Name of Candidate: _____

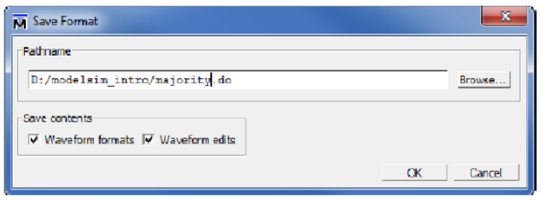



Figure 24. Saving the waveform file.

## 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.

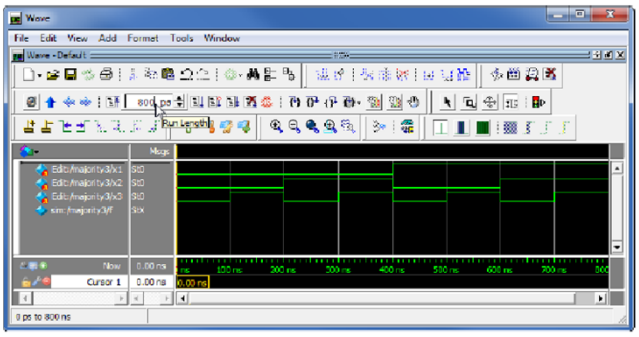



Figure 25. Setting the simulation interval.

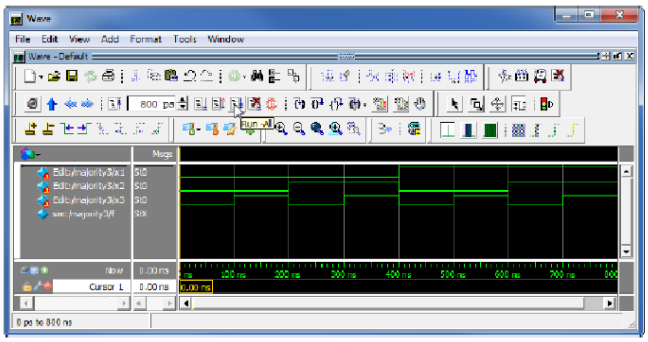



Figure 26. Running the simulation.

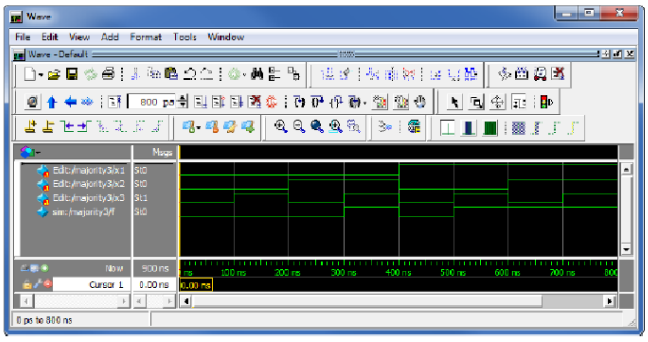



Figure 27. Result of the simulation.

## 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,

Name of Candidate: _____

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

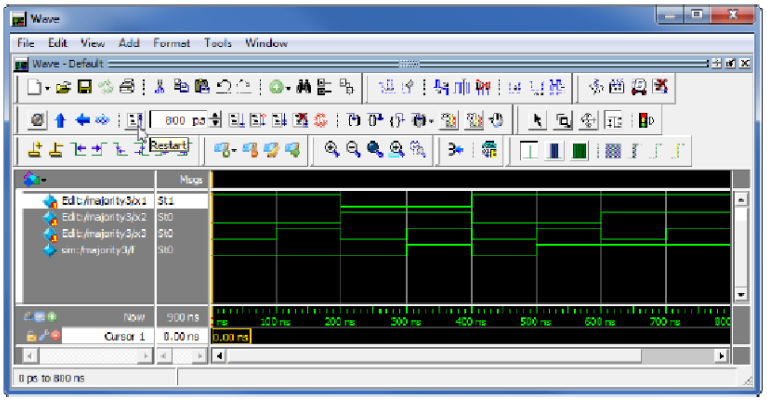



Figure 28. Changed input waveforms.

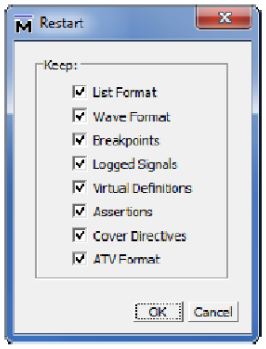



Figure 29. The Restart box.

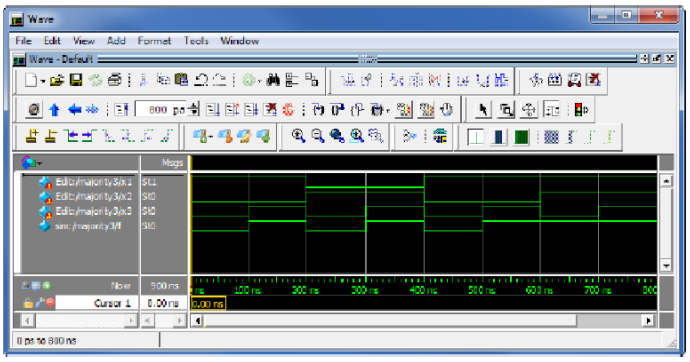



Figure 30. Result of the new simulation.

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

Name of Candidate: _____

# 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**.

Name of Candidate: _____

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;
```