Department of Electronic Engineering
N.E.D. University of Engineering & Technology

PRACTICAL WORK BOOK

For the course

EMBEDDED ELECTRONICS (EL-421)
For B.E (EL)

Instructors name: _____

Student Name: _____

Roll No.: _____ **Batch:** _____

Semester : _____ **Year:** _____

Department: _____

LABORATORY WORK BOOK

FOR THE COURSE

EL-421 Embedded Electronics

Prepared by:

Ms. Hafsa Amanullah

Reviewed by:

Electronic Engineering Department

Approved by:

Board of Studies

Electronic Engineering Department

Contents

Lab Experiment 01

Objective: To *setup* Raspberry Pi with Raspbian OS and to *introduce* python programming.

Introduction

Raspberry Pi is a credit card size single board computer or a programmable PC. It is developed in U.K. by Raspberry Pi Foundation in 2009. The concept was initiated by Eben Upton who works at Broadcom. It is a very low cost board, and a great tool for learning computer programming & concepts of Embedded Linux, etc. It supports all age groups. Supports and runs free and open source Linux OS. It consumes less than 5W of power. Supports full HD video output (1080p), multiple USB ports, etc.

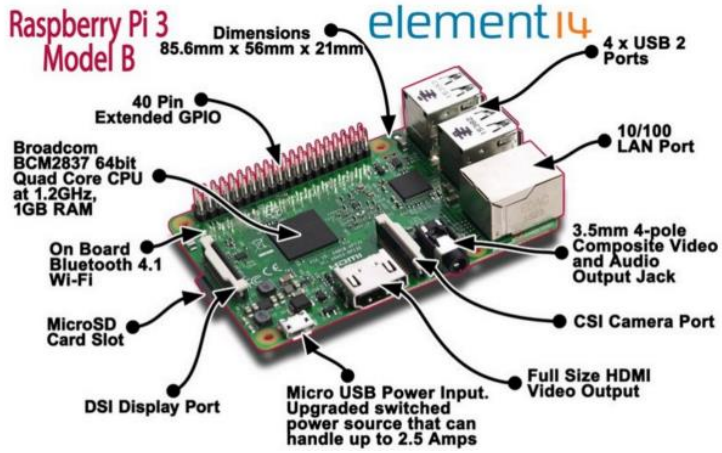


Figure 1-Raspberry Pi 3

i. Technical Specs:

Processer: Quad core 1.2GHz Broadcom BCM2837 SoC (System on Chip)

Memory (RAM): 1GB

On Board Storage: Micro SD port

I/O Lines: 40 Pin GPIO connector

Lab Equipment(s)

Raspberry Pi Board, Power Adapter, SD card, SD card reader, HDMI – VGA converter, Monitor, Mouse, Keyboard

Procedure

i. Setting up the Raspberry Pi:

1. Plug in a monitor (via HDMI) and a keyboard and mouse (via USB)
2. Get an operating system. Raspberry Pi needs an operating system. The image file of the operating system must be present in the micro SD card of Raspberry Pi.

ii. Installing Operating System:

1. Use New-Out-Of-Box Software (NOOBS) [www.raspberrypi.org/downloads]
2. Format the micro SD card with FAT file system.
3. Extract the files from NOOBS and put it on micro SD card.
4. NOOBS will install an operating system on micro SD of Raspberry Pi.
5. From the window shown in the Figure 2, choose Raspbian (a Linux Distribution), the default option.

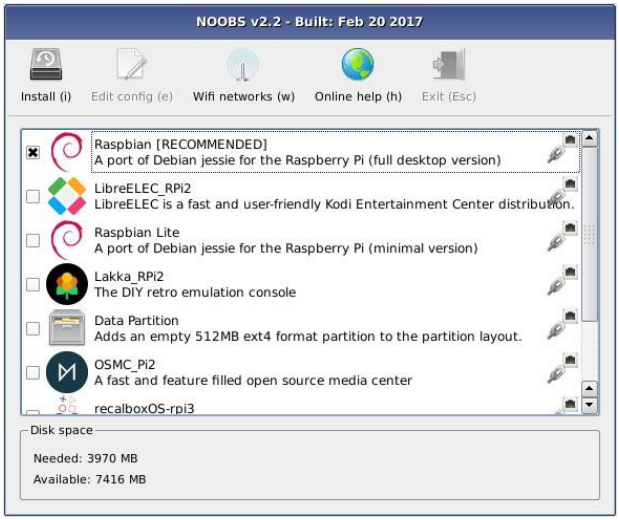


Figure 2-Operating system selection

iii. Configuration of Raspberry Pi (Raspi-Config):

Raspi-Config is a tool which lets you to setup various step/boot options for the Raspberry Pi.

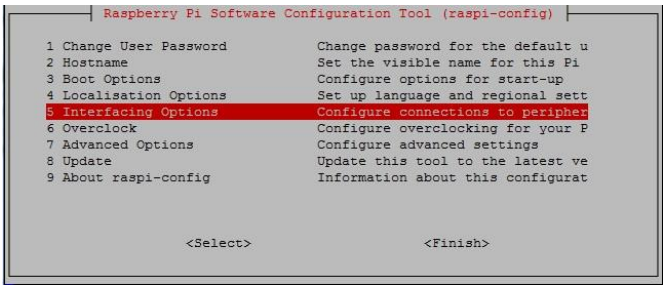


Figure 3

Python Programming language:

Python is a high-level language and it is very easy to use. It is slower as compared to C/C++ as this language is interpreted rather than compiled. There are two versions on this language i.e.; Python 2 and Python 3. Both the versions are valid.

There are two possible environments of python programming:

- Integrated Development Environment (IDE) [default].
- Text editor and interpreter, separately.

There are two ways to execute the python code:

- **Interactive:** executes lines typed interactively in a Python console.
- **Batch:** execute an entire python program.

i. Python Expressions

Algebraic Expressions

Python shell can evaluate algebraic expressions (+, -, *, /). Many algebraic functions are available like abs (), min (), max (). Some examples are:

```
>>> 2 + 2          >>> 2 * (3 + 2)   >>> 8 – 5         >>> 7 / 2
>>> 4              >>> 10            >>> 3             >>> 3.5
```

Boolean Expressions

Evaluate to True or False. It involves comparison operators like <, >, ==, !=, <= and >=. Some examples are:

```
>>> 2 < 4          >>> 2 != 3        >>> 1 >= 3        >>> 2 == 4
>>> True           >>> True          >>> False         >>> False
```

Boolean Operators

It includes Boolean operators like and, or, not. It also evaluates to True or False.

```
>>> 4 == 5 and 3 < 4    >>> True and True      >>> False or False
False                   True                  False
```

Variables Assignments

Variables types are not declared. Interpreter determines type by usage.

```
>>> x == 3              >>> 4 * x             >>> y = 4 * x
>>> x                                         >>> y
3                       12                    12
```

ii. Strings

A sequence of characters enclosed in quotes “Hello, world”. It can be assigned to a variable. It can also be manipulated using string operators and functions.

```
>>> ‘Hello world’       >>> s = ‘still’        >>> t = ‘life’
‘Hello world’
```

String operators

Operator	Definition
x in s	x is a substring of s
x notin s	x is not a substring of s

s + t	Concatenation of s and t
s * n, n* s	Concatenation of n copies of s
s[i]	Character at index i of s
len (s)	(function) length of string

Indexing operators

Index of an item in a sequence is its position in the sequence. Indexing operator is [ ], takes an index as argument. Indices start at 0. It can be used to identify characters in a string.

```
>>> s = 'Apple'

>>> s[0]        >>> s[1]        >>> s[4]
  'A'             'p'             'e'
```

iii. Functions

A sequence of instructions associated with a function name is called function. Function definition starts with **def.** followed by function name, open/close parenthesis and colon.

Function Definition	Function Call
`>>> def test():` `  print ('A test function')`	`>>> test()` `A test function`

Function parameters/arguments:

A function can take arguments which are values bound to variables inside the function. Arguments are listed between parentheses in the function call.

```
>>> def circle_area (rad):
        print (3.14 * rad * rad)
>>> circle_area (2)
12.56
```

Function Return Values:

Function can return values with the **return** instruction. A function call is substituted for its return value in an expression.

```
>>> def circle_area (rad):
       return (3.14 * rad * rad)
>>> total = 3 + circle_area (2)
>>> print(total)
15.56
```

iv. Lists:

A comma-separated sequence of items enclosed in square brackets is called list. Items can be numbers, strings, other lists, etc.

```
>>> pets = ['ant', 'bat', 'cat', 'dog']
>>> lst = [0, 1, 'two', [4, 'five']]
>>> nums = [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

List operators and functions

Operator	Definition
x in lst	x is an item of list named lst
x notin lst	x is not an item of list named lst
lst + lstB	Concatenation of lst and lstB
lst * n, n* lst	Concatenation of n copies of lst
lst [i]	Item at index i of lst
len (lst)	Number of items in lst
min (lst)	Minimum item in lst
max (lst)	Maximum item in lst
sum (lst)	Sum of item in lst

List methods

Operator	Definition
lst.append (item)	Adds item to the end of lst
lst.count (item)	Returns the number of times item occurs in lst
lst.index (item)	Returns index of (first occurance of) item in lst
lst.pop ()	Removes and returns the last item in lst
lst.remove (item)	Removes (the first occurrence of) item from lst
lst.reverse (item)	Reverses the order of items in lst
lst.sort (item)	Sorts the items of lst in increasing order

append ( ), remove ( ), reverse ( ) and sort ( ) do not return values.

v. *Control Flow*

Control flow instructions are the statements that change the order in which lines of code are executed.

If statement

Template	Example

`if <condition>:`
`  <indented code block>`
`<non_indented statement>`
	`if temp > 80:`
	`  print ('It is hot!')`
	`print ('Good Bye!')`

if-else statement:

Template	Example

`if <condition>:`
`  <indented code 1>`
`else:`
`  <indented code 2>`
`<non_indented statement>`
	`if temp > 80:`
	`  print ('It is hot!')`
	`else:`
	`  print ('not hot!')`
	`print ('Goodbye')`

For loop:

Executes a block of code for every element in a sequence. Variable is bound to a sequence element on each pass.

```
>>> name = 'Ali'
>>> for char in name:
        print (char)
A

l

i
```

While loop:

Execute intended block of code while condition is True.

```
>>> i = 0
>>> while i < 3:
        print (i)
        i = i + 1
0
1
2
```

Task

1. Write a python code to print Fibonacci series up to 6 terms for even numbered roll no. and 7 terms for odd numbered roll no.
2. Write a python code that takes string as input and displays the number of vowels present in the string.

NED University of Engineering & Technology
Department of **Electronic** Engineering
Course Code and Title: EL - 421 Embedded Electronics
Laboratory Session No. _____ Date: _____

Psychomotor Domain Assessment Rubric-Level P3					
Skill Sets	Extent of Achievement				
	0	1	2	3	4
Equipment Identification Sensory skill to ***identify*** equipment and/or its component for a lab work.	Not able to identify the equipment	-	-	-	Able to identify equipment as well as its components
Equipment Use Sensory skills to ***demonstrate*** the use of the equipment for the lab work.	Doesn't demonstrate the use of equipment.	Slightly demonstrates the use of equipment.	Somewhat demonstrates the use of equipment.	Moderately demo nstrates the use of equipment.	Fully demonstrates the use of equipment.
Procedural Skills ***Displays*** skills to act upon sequence of steps in lab work.	Not able to either learn or perform lab work procedure.	Able to slightly understand lab work procedure and perform lab work.	Able to somewhat understand lab work procedure and perform lab work.	Able to moderately understand lab work procedure and perform lab work.	Able to fully understand lab work procedure and perform lab work.
Response Ability to ***imitate*** the lab work on his/her own.	Not able to imitate the lab work.	Able to slightly imitate the lab work.	Able to somewhat imitate the lab work.	Able to moderately imitate the lab work.	Able to fully imitate the lab work.
Observation's Use ***Displays*** skills to use the observations from lab work for experimental verifications and illustrations.	Not able to use the observations from lab work for experimental verifications and illustrations.	Slightly able to use the observations from lab work for experimental verifications and illustrations.	Somewhat able to use the observations from lab work for experimental verifications and illustrations.	Moderately able to use the observations from lab work for experimental verifications and illustrations.	Fully able to use the observations from lab work for experimental verifications and illustrations.
Safety Adherence Adherence to ***safety*** procedures.	Doesn't adhere to safety procedures.	Slightly adheres to safety procedures.	Somewhat adheres to safety procedures.	Moderately adheres to safety procedures.	Fully adheres to safety procedures
Equipment Handling ***Equipment care*** during the use.	Doesn't handle equipment with required care.	Rarely handles equipment with required care.	Occasionally handles equipment with required care.	Often handles equipment with required care.	Handles equipment with required care.
Group Work ***Contributes*** in a group based lab work.	Doesn't participate and contribute.	Slightly participates and contributes.	Somewhat participates and contributes.	Moderately participates and contributes.	Fully participates and contributes.

Weighted CLO (Psychomotor Score)	
Remarks	
Instructor's Signature with Date:	

Lab Experiment 02

Objective: To *practice* interfacing digital I/Os using General Purpose Input / Output (GPIO) pins of Raspberry Pi.

Introduction

Digital I/O

One powerful feature of the Raspberry Pi is the row of GPIO (general purpose input/output) pins along the top edge of the board. These pins are a physical interface between the Pi and the outside world. At the simplest level, you can think of them as switches that you can turn on or off (input) or that the Pi can turn on or off (output). Of the 40 pins, 26 are GPIO pins and the others are power or ground pins (plus two ID EEPROM pins which you should not play with unless you know your stuff!)

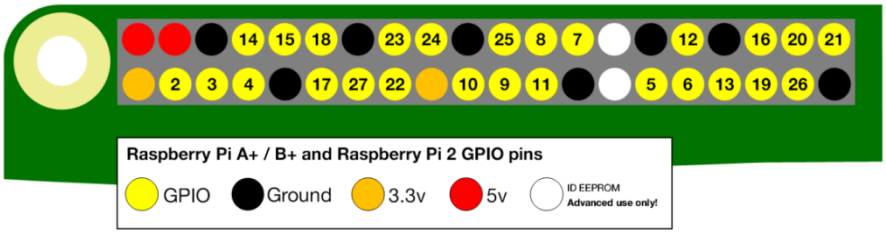


Figure 4-GPIO layout

General Purpose/Multi-Function:

- Yellow colored pins can be used a general purpose I/O.
- Some pins are multi-function.

UART Pins:

- Pins 8 and 10 can be used for UART (serial) communication
- Tx for transmission.
- Rx for reception.

I2C Pins:

- Pins 3 and 5 can be used for I2C Communication.
- SDA for data.
- SCL for clock.

SPI Communication:

- MOSI, MISO, SCLK.
- 2 pins for chip enable, CE0 and CE1.

i. GPIO Access:

To access the GPIO pins, GPIO library is used.

```
import RPi.GPIO as GPIO
```

Pin Numbering Modes:

There are two ways to refer the GPIO pins:

1. The number of the pins in their order on the board (referring to Figure 4, lower rail of pins being odd numbered likewise upper rail being even numbered).

```
GPIO.setmode (GPIO.BOARD)
```

2. The Broadcom SoC (System on Chip) number.

```
GPIO.setmode (GPIO.BCM)
```

Pin Direction and Assignment:

```
GPIO.setup (13, GPIO.OUT) // Set the pin direction

GPIO.output (13, True) // Assign value to output pin
```

Reading Input:

```
GPIO.setup (13, GPIO.IN) // Set the pin direction to an input

value = GPIO.input(13) // Read value on input pin.
```

Analog output

Pulse Width Modulation (PWM) is the most common technique in digital systems to provide analog signals. In digital systems, we cannot directly apply the analog signal. In this technique, the width of the waveform is varied by changing the value of duration of the high signal.

In Raspberry Pi 3, pin 12 and 24 (board numbering) generates PWM signals.

PWM Initialization:

```
pwm_obj = GPIO.PWM (12, 50)
pwm_obj.start (100)
```

PWM Control:

```
pwm_obj.ChangeDutyCycle (50)
```

Lab Equipment(s)

Raspberry Pi Board, Power Adapter, SD card, SD card reader, HDMI – VGA converter, Monitor, Mouse, Keyboard

Task(s)

1. Connect the circuit as shown in the schematic. Write the code given below using Python IDLE 3, run the code and observe the output.

```
import RPi.GPIO as GPIO
import time

GPIO.setmode GPIO.BOARD)
GPIO.setup (13, GPIO.OUT)

While True:
GPIO.output (13, True)
time.sleep (1)
GPIO.output (13, False)
time.sleep (1)
```

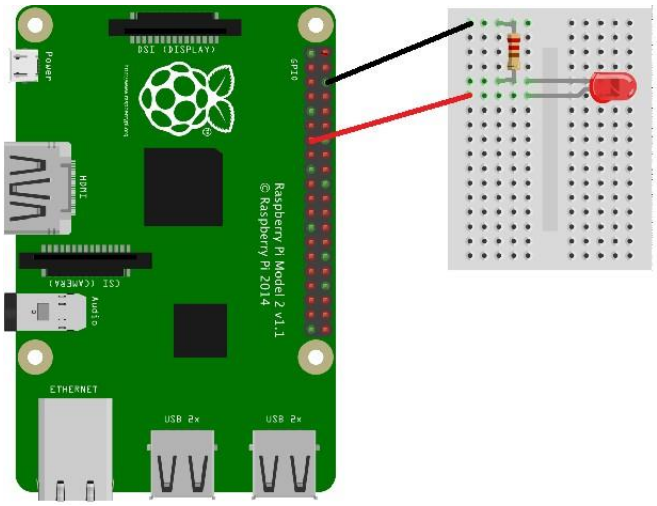


Figure 5-Raspberry Pi connections

2. Design a traffic signal by employing red, yellow, and green LEDs and turning them on and off in the appropriate sequence.
3. Connect the LED to pin 12 of RPi (Board Numbering). Write the code with the help of pseudo code given below to control the brightness of LED using PWM Function.

```
// import GPIO Library
// import time Library

// set the GPIO mode (board/BCM)
// set the pin mode as (input/output)

// Initialize the PWM
while True:
      for i in range (100):
            pwm.ChangeDutyCycle (i)
            time.sleep (0.1)
      for i in range (100, 0, -1):
            pwm.ChangeDutyCycle (i)
            time.sleep (0.1)
```

NED University of Engineering & Technology
Department of **Electronic** Engineering
Course Code and Title: EL - 421 Embedded Electronics
Laboratory Session No. _____ Date: _____

Psychomotor Domain Assessment Rubric-Level P3					
Skill Sets	Extent of Achievement				
	0	1	2	3	4
Equipment Identification Sensory skill to ***identify*** equipment and/or its component for a lab work.	Not able to identify the equipment	-	-	-	Able to identify equipment as well as its components
Equipment Use Sensory skills to ***demonstrate*** the use of the equipment for the lab work.	Doesn't demonstrate the use of equipment.	Slightly demonstrates the use of equipment.	Somewhat demonstrates the use of equipment.	Moderately demo nstrates the use of equipment.	Fully demonstrates the use of equipment.
Procedural Skills ***Displays*** skills to act upon sequence of steps in lab work.	Not able to either learn or perform lab work procedure.	Able to slightly understand lab work procedure and perform lab work.	Able to somewhat understand lab work procedure and perform lab work.	Able to moderately understand lab work procedure and perform lab work.	Able to fully understand lab work procedure and perform lab work.
Response Ability to ***imitate*** the lab work on his/her own.	Not able to imitate the lab work.	Able to slightly imitate the lab work.	Able to somewhat imitate the lab work.	Able to moderately imitate the lab work.	Able to fully imitate the lab work.
Observation's Use ***Displays*** skills to use the observations from lab work for experimental verifications and illustrations.	Not able to use the observations from lab work for experimental verifications and illustrations.	Slightly able to use the observations from lab work for experimental verifications and illustrations.	Somewhat able to use the observations from lab work for experimental verifications and illustrations.	Moderately able to use the observations from lab work for experimental verifications and illustrations.	Fully able to use the observations from lab work for experimental verifications and illustrations.
Safety Adherence Adherence to ***safety*** procedures.	Doesn't adhere to safety procedures.	Slightly adheres to safety procedures.	Somewhat adheres to safety procedures.	Moderately adheres to safety procedures.	Fully adheres to safety procedures
Equipment Handling ***Equipment care*** during the use.	Doesn't handle equipment with required care.	Rarely handles equipment with required care.	Occasionally handles equipment with required care.	Often handles equipment with required care.	Handles equipment with required care.
Group Work ***Contributes*** in a group based lab work.	Doesn't participate and contribute.	Slightly participates and contributes.	Somewhat participates and contributes.	Moderately participates and contributes.	Fully participates and contributes.

Weighted CLO (Psychomotor Score)	
Remarks	
Instructor's Signature with Date:	

Lab Experiment 03

Objective: To *build* traffic light system with Finite State Machine (FSM) using Raspberry Pi.

Introduction

An FSM could take one of forms illustrated in Figure 6. An FSM could have multiple inputs, multiple outputs, and state represented by multiple bits, (k bits in diagram). It also receives a clock signal and a possible reset signal. There are two blocks of combinational logic: one that determines next state, based on current state and inputs; and another that determines the output. Every FSM also has a memory element that stores the state. On each clock edge, the FSM transitions from current state to next state. In a Moore machine, the output is a function of state only. In a Mealy machine, output is a function of state and inputs.

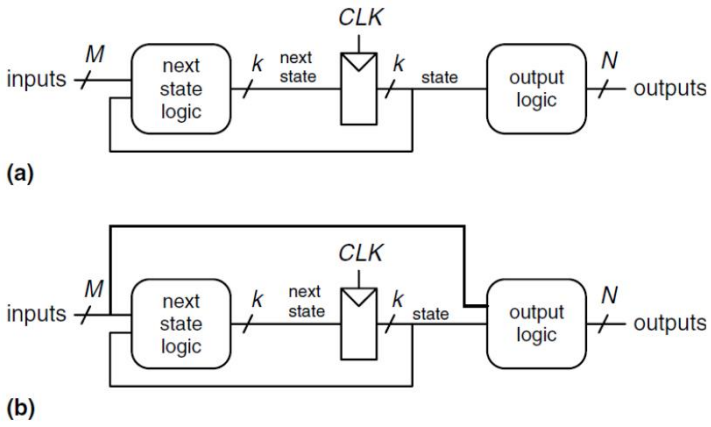


Figure 6-(a) Moore Machine (b) Mealy Machine

Steps to design FSM

1. Identify the inputs and outputs.
2. Sketch a state transition diagram.
3. For a Moore Machine:
    a. Write a state transition table.
    b. Write an output table.

    For a Mealy Machine:
    a. Write a combined state transition and output table
4. Select state encoding and convert the above tables to truth tables.
5. Write Boolean equations for next state and output logic based on truth tables.
6. Draw schematic outlining connections between gates based on Boolean equations. These schematics are next state logic and output logic blocks in Figure 6(a).
7. For implementation, we need memory devices that store current state till the next clock edge. This is the middle block, accepting CLK signal, in Figure 6(a).

Lab Equipment(s)

Raspberry Pi Board, Power Adapter, SD card, SD card reader, HDMI – VGA converter, Monitor, Mouse, Keyboard

Task

Design a traffic light system for the intersection shown in Figure 7. Specifications for the design are that traffic should flow either on Academic Ave. or Bravado Blvd. at a time, i.e. when LA is green then LB should be red and vice versa. A green light first turns yellow for 5 seconds before it turns red. A red light directly switches to green. Additionally, there are two traffic sensors, TA and TB, on each of the roads respectively. The sensors TA and TB are HIGH if there is traffic on corresponding road, and LOW if it's empty. If a light is green it should remain in that state till all the traffic passes. The system should also provide a reset setting by which system is put in a known state, specifically green on Academic; red on Bravado.

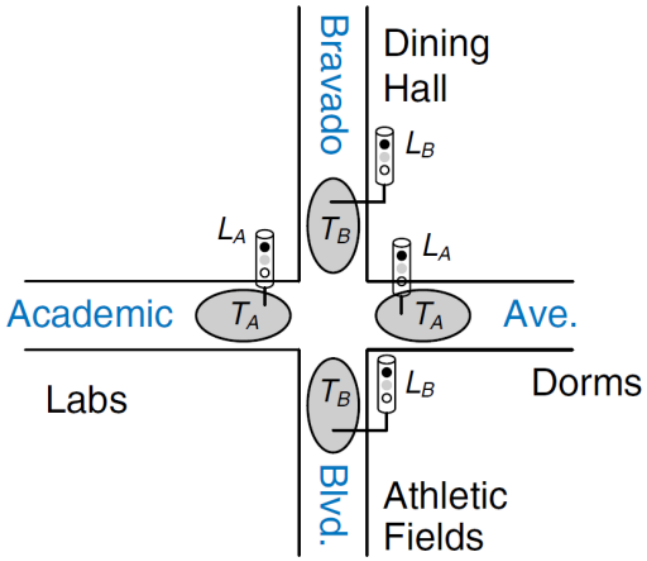


Figure 7-Intersection scenario of traffic light

Reference:

Harris, D., & Harris, S. (2010). Digital design and computer architecture. Morgan Kaufmann.

NED University of Engineering & Technology
Department of **Electronic** Engineering
Course Code and Title: EL - 421 Embedded Electronics
Laboratory Session No. _____ Date: _____

Psychomotor Domain Assessment Rubric-Level P3					
Skill Sets	Extent of Achievement				
	0	1	2	3	4
Equipment Identification Sensory skill to ***identify*** equipment and/or its component for a lab work.	Not able to identify the equipment	-	-	-	Able to identify equipment as well as its components
Equipment Use Sensory skills to ***demonstrate*** the use of the equipment for the lab work.	Doesn't demonstrate the use of equipment.	Slightly demonstrates the use of equipment.	Somewhat demonstrates the use of equipment.	Moderately demo nstrates the use of equipment.	Fully demonstrates the use of equipment.
Procedural Skills ***Displays*** skills to act upon sequence of steps in lab work.	Not able to either learn or perform lab work procedure.	Able to slightly understand lab work procedure and perform lab work.	Able to somewhat understand lab work procedure and perform lab work.	Able to moderately understand lab work procedure and perform lab work.	Able to fully understand lab work procedure and perform lab work.
Response Ability to ***imitate*** the lab work on his/her own.	Not able to imitate the lab work.	Able to slightly imitate the lab work.	Able to somewhat imitate the lab work.	Able to moderately imitate the lab work.	Able to fully imitate the lab work.
Observation's Use ***Displays*** skills to use the observations from lab work for experimental verifications and illustrations.	Not able to use the observations from lab work for experimental verifications and illustrations.	Slightly able to use the observations from lab work for experimental verifications and illustrations.	Somewhat able to use the observations from lab work for experimental verifications and illustrations.	Moderately able to use the observations from lab work for experimental verifications and illustrations.	Fully able to use the observations from lab work for experimental verifications and illustrations.
Safety Adherence Adherence to ***safety*** procedures.	Doesn't adhere to safety procedures.	Slightly adheres to safety procedures.	Somewhat adheres to safety procedures.	Moderately adheres to safety procedures.	Fully adheres to safety procedures
Equipment Handling ***Equipment care*** during the use.	Doesn't handle equipment with required care.	Rarely handles equipment with required care.	Occasionally handles equipment with required care.	Often handles equipment with required care.	Handles equipment with required care.
Group Work ***Contributes*** in a group based lab work.	Doesn't participate and contribute.	Slightly participates and contributes.	Somewhat participates and contributes.	Moderately participates and contributes.	Fully participates and contributes.

Weighted CLO (Psychomotor Score)	
Remarks	
Instructor's Signature with Date:	

Lab Experiment 04

Objective: To *establish* SPI communication between Raspberry Pi and Arduino.

Introduction

SPI is a common communication protocol used by many different devices. For example, SD card modules, RFID card reader modules, and 2.4 GHz wireless transmitter/receivers all use SPI to communicate with microcontrollers.

One unique benefit of SPI is the fact that data can be transferred without interruption. Any number of bits can be sent or received in a continuous stream. With I2C and UART, data is sent in packets, limited to a specific number of bits. Start and stop conditions define the beginning and end of each packet, so the data is interrupted during transmission.

Devices communicating via SPI are in a master-slave relationship. The master is the controlling device (usually a microcontroller), while the slave (usually a sensor, display, or memory chip) takes instruction from the master.

The simplest configuration of SPI is a single master, single slave system, but one master can control more than one slave (more on this below).

MOSI (Master Output/Slave Input) – Line for the master to send data to the slave.

MISO (Master Input/Slave Output) – Line for the slave to send data to the master.

SCLK (Clock) – Line for the clock signal.

SS/CS (Slave Select/Chip Select) – Line for the master to select which slave to send data to.

Lab Equipment(s)

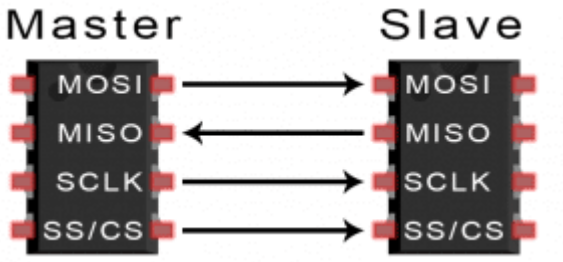
Raspberry Pi Board, Power Adapter, SD card, SD card reader, HDMI – VGA converter, Monitor, Mouse, Keyboard, Arduino Uno, Desktop PC with Arduino IDE.

Hardware setup for SPI Communication

Make following connections:

- GND (first)
- MISO (master) to MISO (slave)
- MOSI (master) to MOSI (slave)
- SCLK (master) to SCLK (slave)

Make sure to connect MISO to MISO and MOSI to MOSI, not MISO to MOSI. The software side will handle that depending on which device is set as a slave or master.



Note that for SPI, you normally have another wire connected to CS (Chip Select), or SS (Slave Select). This is useful to choose which slave you are talking to. Here, as we have only one Arduino slave, no need for this wire, the communication will still work.

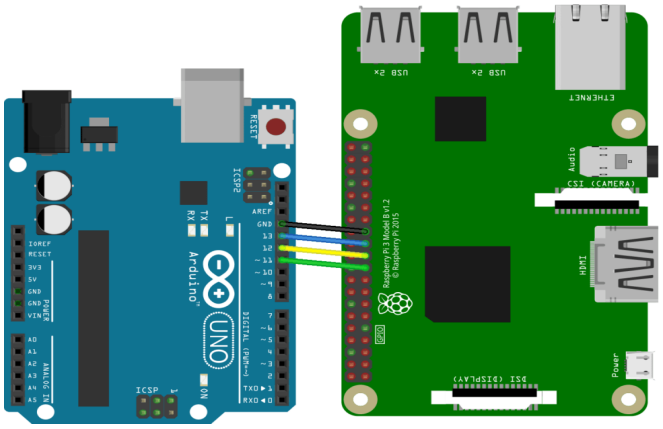


Figure 8-SPI Connections

Software setup for SPI Communication

If you haven't used SPI on your Raspberry Pi yet, it probably means that the SPI communication is not activated. To activate it, search for the **`/boot/config.txt`** file.

Open this file (with sudo), find the line `#dtparam=spi=on`, and remove the leading '#' to uncomment it.

After that, reboot your Pi, and SPI will be activated as long as you don't comment the SPI line again in the config file.

Task(s): SPI Communication using Arduino (as slave) and Raspberry Pi (as master).

Raspberry Pi SPI master Program	Basic Arduino SPI slave program

1. #include <iostream>
2. #include <wiringPiSPI.h>
3.
4. #define SPI_CHANNEL 0
5. #define SPI_CLOCK_SPEED 1000000
6.
7. int main(int argc, char **argv)
8. {
9. int fd = wiringPiSPISetupMode(SPI_CHANNEL, SPI_CLOCK_SPEED, 0);
10. if (fd == -1) {
11. std::cout << "Failed to init SPI communication.\n";
12. return -1;
13. }
14. std::cout << "SPI communication successfully setup.\n";
	1. #include <SPI.h>
	2.
	3. void setup() {
	4. // have to send on master in, *slave out*
	5. pinMode(MISO, OUTPUT);
	6.
	7. // turn on SPI in slave mode
	8. SPCR |= _BV(SPE);
	9.
	10. // turn on interrupts
	11. SPI.attachInterrupt();
	12. }
	13.
	14. // SPI interrupt routine

```
15.
16.    unsigned char buf[2] = { 23, 0 };
17.    wiringPiSPIDataRW(SPI_CHANNEL, buf, 2);
18.
19.    std::cout << "Data returned: " << +buf[1] << "\n";
20.    return 0;
21.    }
```

```
15.    ISR (SPI_STC_vect)
16.    {
17.    byte c = SPDR;
18.
19.    SPDR = c+10;
20.    } // end of interrupt service routine (ISR) for SPI
21.
22.    void loop() { }
```

- First the Raspberry Pi sends the value 23 to the Arduino, and receives a byte. No need to really care about this received byte here.
- Upon reception of the first byte, the Arduino will trigger the SPI interrupt, add 10, and set the new value, 33, to the SPI shift register, so it's ready for the next transfer.
- Then, the Raspberry Pi sends the second value from the buffer, and receives the value 33. The received value will be printed which is the second byte of the buffer.

NED University of Engineering & Technology
Department of **Electronic** Engineering
Course Code and Title: EL - 421 Embedded Electronics
Laboratory Session No. _____ Date: _____

Psychomotor Domain Assessment Rubric-Level P3					
Skill Sets	Extent of Achievement				
	0	1	2	3	4
Equipment Identification Sensory skill to ***identify*** equipment and/or its component for a lab work.	Not able to identify the equipment	-	-	-	Able to identify equipment as well as its components
Equipment Use Sensory skills to ***demonstrate*** the use of the equipment for the lab work.	Doesn't demonstrate the use of equipment.	Slightly demonstrates the use of equipment.	Somewhat demonstrates the use of equipment.	Moderately demo nstrates the use of equipment.	Fully demonstrates the use of equipment.
Procedural Skills ***Displays*** skills to act upon sequence of steps in lab work.	Not able to either learn or perform lab work procedure.	Able to slightly understand lab work procedure and perform lab work.	Able to somewhat understand lab work procedure and perform lab work.	Able to moderately understand lab work procedure and perform lab work.	Able to fully understand lab work procedure and perform lab work.
Response Ability to ***imitate*** the lab work on his/her own.	Not able to imitate the lab work.	Able to slightly imitate the lab work.	Able to somewhat imitate the lab work.	Able to moderately imitate the lab work.	Able to fully imitate the lab work.
Observation's Use ***Displays*** skills to use the observations from lab work for experimental verifications and illustrations.	Not able to use the observations from lab work for experimental verifications and illustrations.	Slightly able to use the observations from lab work for experimental verifications and illustrations.	Somewhat able to use the observations from lab work for experimental verifications and illustrations.	Moderately able to use the observations from lab work for experimental verifications and illustrations.	Fully able to use the observations from lab work for experimental verifications and illustrations.
Safety Adherence Adherence to ***safety*** procedures.	Doesn't adhere to safety procedures.	Slightly adheres to safety procedures.	Somewhat adheres to safety procedures.	Moderately adheres to safety procedures.	Fully adheres to safety procedures
Equipment Handling ***Equipment care*** during the use.	Doesn't handle equipment with required care.	Rarely handles equipment with required care.	Occasionally handles equipment with required care.	Often handles equipment with required care.	Handles equipment with required care.
Group Work ***Contributes*** in a group based lab work.	Doesn't participate and contribute.	Slightly participates and contributes.	Somewhat participates and contributes.	Moderately participates and contributes.	Fully participates and contributes.

Weighted CLO (Psychomotor Score)	
Remarks	
Instructor's Signature with Date:	

Lab Experiment 05

Objective: To *introduce* Verilog HDL for digital design and functional verification.

Introduction

Three basic kinds of devices available in today's digital era are:

1. Memory devices – store random information as the contents of a spreadsheet or database.
2. Microprocessors – execute various software instructions to perform a wide variety of tasks.
3. Logic devices – provide specific functions, including device-to-device interfacing, data communication, signal processing, data display, timing and controlling operations, and almost every other function a system must perform.

i. Categories of Logic Devices

Logic devices can be classified into two broad categories – fixed and programmable. The circuits in fixed logic devices are permanent and can't be reprogrammed once after manufacturing, e.g. Application Specific ICs (ASICs). With fixed logic devices, the time required to go from design, to prototypes, to a final manufacturing run can take from several months to more than a year, depending on the complexity of the device. If the device does not work properly, or if the requirements change, a new design must be developed.

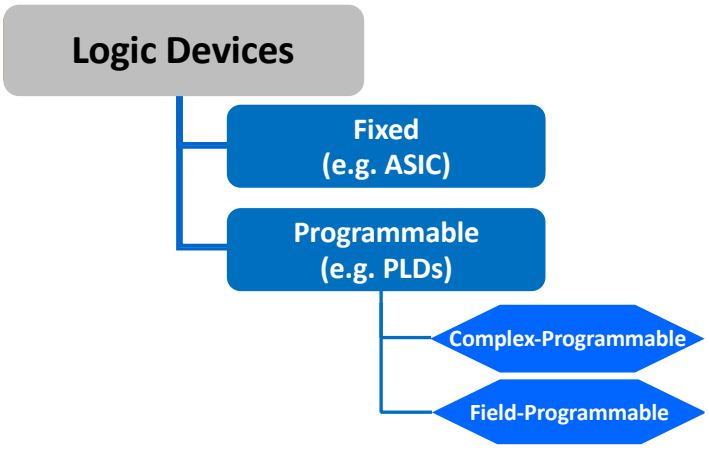


Figure 9-Categories of logic devices

Whereas, programmable logic devices offer customers a wide range of logic capacity and functionality that can be reconfigured whenever required. For PLDs, designers use inexpensive software tools to quickly develop, simulate, and test their designs. Then, a design can be quickly programmed into a device, and immediately tested in a live circuit. The PLD that is used for this prototyping is the exact same PLD that will be used in the chip for the final product such as iphones, USB flash drives etc. Major advantage of using PLDs is

that designer can change the circuitry, during the design phase, as often as it requires until meets the required functionality.

The two major types of programmable logic devices are field programmable gate arrays (FPGAs) and complex programmable logic devices (CPLDs). FPGAs are fine-grained devices and offer the highest amount of logic density (up to 100,000s of logic blocks). They are RAM based and needs to be configured at each power-up. FPGAs also have special routing resources to implement arithmetic functions efficiently. CPLDs, on the other hand, are coarse-grained devices and offer much smaller amounts of logic - up to about 10,000 gates. They are EEPROM based and are active at power-up. Like FPGAs, CPLDs do not have such special routing resources. In these labs, we will be using an FPGA device to create our own processor.

ii. Evolution of Digital Designing using HDLs

For a long time, programming languages such as FORTRAN, Pascal, and C were being used to describe computer programs that were sequential in nature. Similarly, in the digital design field, designers felt the need for a standard language to describe digital circuits. Thus, Hardware Description Languages (HDLs) came into existence. HDLs allow the designers to model the concurrency of processes found in hardware elements. Hardware description languages such as Verilog HDL and VHDL (Very High Speed Integrated Circuit (VHSIC) HDL) became popular. Verilog HDL originated in 1983 at Gateway Design Automation. Later, VHDL was developed under contract from Defense Advanced Research Project Agency (DARPA). Both Verilog and VHDL simulators to simulate large digital circuits quickly gained acceptance from designers.

Even though HDLs were popular for logic verification, designers had to manually translate the HDL-based design into a schematic circuit with interconnections between gates. The advent of logic synthesis in the late 1980s changed the design methodology radically. Digital circuits could be described at a Register Transfer Level (RTL) by use of an HDL. Thus, the designer had to specify how the data flows between registers and how the design processes the data. The details of gates and the interconnections to implement the circuit were automatically extracted by logic synthesis tools from the RTL description.

Thus, logic synthesis pushed the HDLs into the forefront of digital design. Designers no longer had to manually place gates to build digital circuits. They could describe circuits at an abstract level in terms of functionality and dataflow by designing those circuits in HDLs. Logic synthesis tools would implement the specified functionality in terms of gates and gate interconnections.

HDLs also began to be used for system-level design. HDLs were used for simulation of system boards, interconnect buses, FPGAs (Field Programmable Gate Arrays), and PALs (Programmable Array Logic). A common approach is to design each IC chip, using an HDL, and then verify system functionality via simulation.

iii. Selection of HDL

There are two major HDLs – Verilog and VHDL. One should consider the following points while choosing HDL.

- "Ease of Learning" which relates to how easy it is to learn the language without prior experience with HDLs.
- "Ease of Use" means once the first-time user has learned the language, how easy will it be to use the language for their specific design requirements.
- "Adaptability" is another important factor i.e., how the HDL can integrate into the current design environment and the existing design philosophy.

Introduction to Verilog HDL

This section briefly describes the syntax of Verilog HDL and covers some basic lexical elements to write a Verilog HDL code.

i. Lexical Elements

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language. Verilog HDL is a case-sensitive language. All keywords are in lowercase.
Here we will only discuss some of the lexical conventions that are new to Verilog. For a detailed reference, Verilog Language Reference manual or any other reference on Verilog may be consulted.

Identifier – An *identifier* gives a unique name to an object, such as `counter`, `seven_segment`, `e12`, etc. It is composed of letters, digits, the underscore character (_), and the dollar sign (`$`). `$` is usually used with a system task or function.

The first character of an identifier must be a letter or underscore.

Verilog is a *case-sensitive language.* Thus, `data-bus`, `Data-bus`, and `DATA_BUS` refer to three different objects. To avoid confusion, we should refrain from using the case to create different identifiers.

Keywords – *Keywords* are predefined identifiers that are used to describe language constructs. For example, **`module`, `wire`, `not`,** etc.

White space – White space, which includes the space, tab, and newline characters, is used to separate identifiers and can be used freely in the Verilog code. We can use proper white spaces to format the code and make it more readable.

Comments – *A comment* is just for documentation purposes and will be ignored by a compiler. Verilog has two forms of comments. A one-line comment starts with //, as:

```
// This is a comment
```

A multiple-line comment is encapsulated between /* and */, as shown below:

```
/* This is comment line 1.
   This is comment line 2.
   This is comment line 3.
*/
```

ii. Number representation

There are two types of number specification in Verilog: sized and unsized.

Sized numbers

Sized numbers are represented as

```
[sign][size] ‘[base format] [number]
```

[sign] is written in the case of signed numbers. [size] is written only in decimal and specifies the number of bits in the number. Legal base formats are decimal (‘d or ‘D), hexadecimal (‘h or ‘H), binary (‘b or ‘B) and octal (‘o or ‘O). The number is specified as consecutive digits from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. Only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification.

```
4’b1111 // This is a 4-bit binary number
12’habc // This is a 12-bit hexadecimal number
16’d255 // This is a 16-bit decimal number
```

Unsized numbers

Numbers that are specified without a [base format] specification are decimal numbers by default. Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine specific (must be at least 32).

```
23456      // This is a 32-bit decimal number by default
‘hc3       // This is a 32-bit hexadecimal number
‘o21       // This is a 32-bit octal number
```

X or Z values

Verilog has two symbols for unknown and high impedance values. These values are very important for modeling real circuits. An unknown value is denoted by **x**. A high impedance value is denoted by **z**.

```
12’h13x    // This is a 12-bit hex number; 4 least significant
           // bits unknown
6’hx       // This is a 6-bit hex number
32’bz      // This is a 32-bit high impedance number
```

An **x** or a **z** sets four bits for a number in the hexadecimal base, three bits for a number in the octal base, and one bit for a number in the binary base. If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z. This makes it easy to assign x or z to whole vector. If the most significant digit is 1, then it is also zero extended.

Negative numbers

Putting a minus sign before the size for a constant number can specify negative numbers. Size constants are always positive. It is illegal to have a minus sign between <base format> and <number>.

```
-6'd3       //8-bit negative number stored as 2's complement of 3
4'd-2       // illegal specification
```

iii. Data Types

Four basic values are used in most data types:

0: for "logic 0, or a false condition
1: for "logic 1", or a true condition
z: High impedence, floating state
x: for an unknown value

The **z** value corresponds to the output of a tri-state buffer. The **x** value is usually used in modeling and simulation, representing a value that is not 0, I, or z, such as an uninitialized input or output conflict.

Verilog has two main groups of data types: `net` and `register`

Nets

Nets represent connections between hardware elements. Just as in real circuits, nets have values continuously driven onto them by the outputs of devices that they are connected to.

Nets are declared primarily with the keyword **wire**. Nets are one-bit values by default unless they are declared explicitly as vectors.

```
wire pO, pl;                // two 1-bit signals
wire [7:0] datal, data2;    // 8-bit data
wire [31:0] addr;           // 32- bit address
wire [0:7] revers-data;     //ascending index should be avoided
```

While the index range can be either descending (as in [7:0]) or ascending (as in [0:7]), the former is preferred since the leftmost position (i.e., 7) corresponds to the MSB of a binary number.

It is possible to address bits or parts of a vector. For example, `data1[3]` refers to the bit 3 of a wire `data1` declared above. `addr[2:0]` three least significant bits of a vector `addr`

The term **wire** and net are often used interchangeably. The default value of a net is **z**. Nets get the output value of their drivers. If a net has no driver, it gets the value **z**.

Registers

Registers represent storage data elements. Registers retain value until another value is placed onto them. Do not confuse the term registers in Verilog with hardware registers built from edge triggered flip-flops in real circuits. In Verilog, the term register merely means a variable that can hold a value. Unlike a net, a register does not need a driver. Verilog registers do not need a clock as hardware registers do. Values of registers can be changed anytime in a simulation by assigning a new value to the register.

Register data types are commonly declared by the key word `reg`. The default value for a `reg` data type is `x`. An example of how registers are declared and used is shown below:

```
reg reset;   // declare a variable that can hold its value
```

iv. Modules

Verilog provides the concept of a module. A module is the basic building block in Verilog. A module can be an element or a collection of lower-level design blocks. A module provides the necessary functionality to the higher-level blocks through its port interface (inputs and outputs) but hides the internal implementation. This allows the designer to modify module internals without affecting the rest of the design.

In Verilog, a module is declared by the keyword **`module`**. A corresponding keyword **`endmodule`** must appear at the end of the module definition. Each module must have a *module_name*, which is the identifier for the module, and a *module_terminal_list*, which describes the input and output terminals of the module.

```
module     <module_name> (<module_terminal_list>);
. . .
<module internals>
. . .
endmodule
```

v. Ports

Ports provide the interface by which modules can communicate with its environment. For example, the *input/output* pins of an IC chip are its ports. The environment can interact with the module only through its ports. The internals of the module are not visible to the environment. This provides a very powerful flexibility to the designer. The internals of the module can be changed without affecting the environment as long as the interface is not modified. Ports are also referred to as *terminals.*

A module definition contains as optional list of ports. If the module does not exchange any signals with the environment, there are no ports in the list.

Port Declaration

All ports in the list of ports must be declared in the module. Ports can be declared as follows

Verilog Keyword	Type of Port
input	Input port
output	Output port
Inout	Bi-directional port

Each port in the port list is defined as input, output, or inout, based on the direction of the port signal. Thus, for the example of the `fulladd4`, the port declarations will be as shown below:

```
module fulladd4 (sum, c_out, a, b, c_in);

// Begin port declarations section
output [3:0] sum; output c_out;

input [3:0] a, b;
input c_in;
// End port declarations section
…
<module internals>
…
endmodule
```

Note that all port declarations are implicitly declared as **wire** in Verilog. Thus, if port is intended to be a **`wire`**, it is sufficient to declare it as **`output`**, **`input`**, or **`inout`**. Input or inout ports as normally declared as **`wire`**. However, if output ports hold their value, they must be declared as **`reg`**. For example, consider a module for a D-type flip-flop:

```
module DFF (q, d, clk, reset);
output reg q; // Output port q holds value; therefore it
                // is declared as reg
input d, clk, resetl
…
…
endmodule
```

Ports of the type **`input`** cannot be declared as **`reg`**, because **`reg`** variables store values and **`input`** ports should not store values but simply reflect the changes in the external signals they are connected to.

vi. Instances

A module provides a template from which you can create actual objects. When a module is invoked, Verilog creates a unique object from the template. Each object has its own name, variables, parameters and I/O interfaces. The process of creating objects from a module template is called *instantiation*, and the objects are called *instances*. In the example below, which shows the module for a 4- bit ripple carry counter, four instances from the T-flipflop template are created. The internals of the T_FF module are not shown.

```
// Define the top-level module called ripple carry counter.   //
It instantiates 4 T-flipflops.

module ripple_carry_counter (q, clk, reset);

output [3:0] q;
input clk, reset;

// Four instances of the module T_FF are created.  Each has a  //
unique name. Each instance is passed a set of signals.
// Notice that each instance is a copy of the module T_FF

T_FF tff0 (.q(q[0]), .c(clk), .reset(reset));
T_FF tff1 (.q(q[1]), .c(q[0]), .reset(reset));
T_FF tff2 (.q(q[2]), .c(q[1]), .reset(reset));
T_FF tff3 (.q(q[3]), .c(q[2]), .reset(reset));

endmodule
```

The port name of a calling module (the module being instantiated) comes first. In above example, the port `c` of a `T_FF tff0` module is connected to the port `clk` of a `ripple_carry_counter`.

Hands-on Verilog HDL

In this section, we will develop an example hardware of a 4-bit ripple carry counter using the concepts learnt in previous section. The top-level diagram of a 4-bit ripple carry counter is shown in the Figure 10. It is designed with four negative edge-triggered T-flip flops.

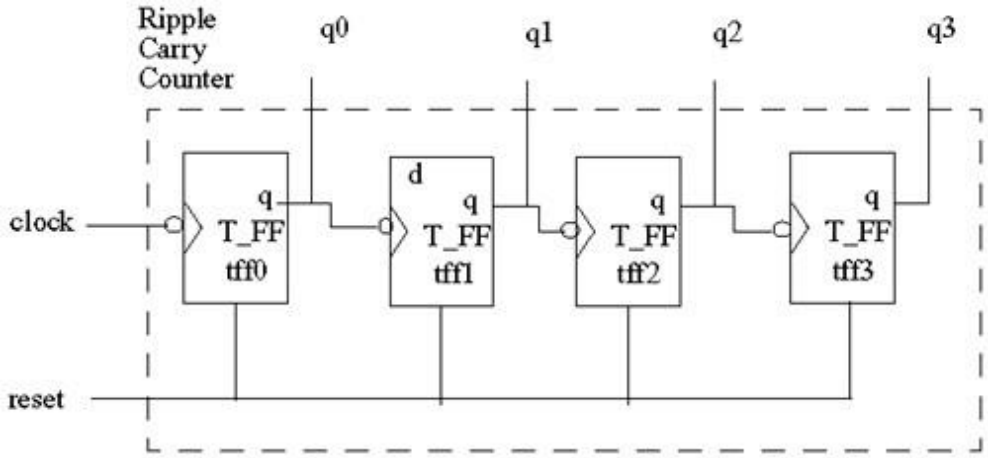


Figure 10-Top-level diagram of a ripple carry counter

Each of these above T-flipflops can be made from a negative edge-triggered D-flipflop and an inverter (assuming that a q' output is not available). The implementation of T_FF using D_FF and an inverter is shown in Figure 11.

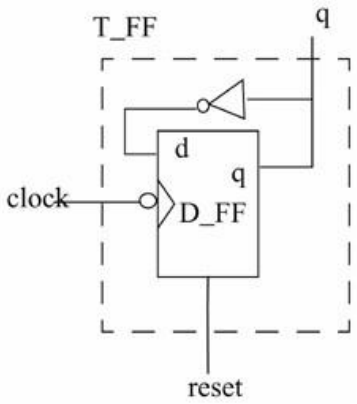


Figure 11-Implementation of a T_FF using D_FF and inverter

i. Ripple Carry Counter

Using the top-down design approach, we will first write the Verilog description of a top-level module, which is a 4-bit ripple carry counter. The Verilog code of ripple carry counter is shown in Figure 12.

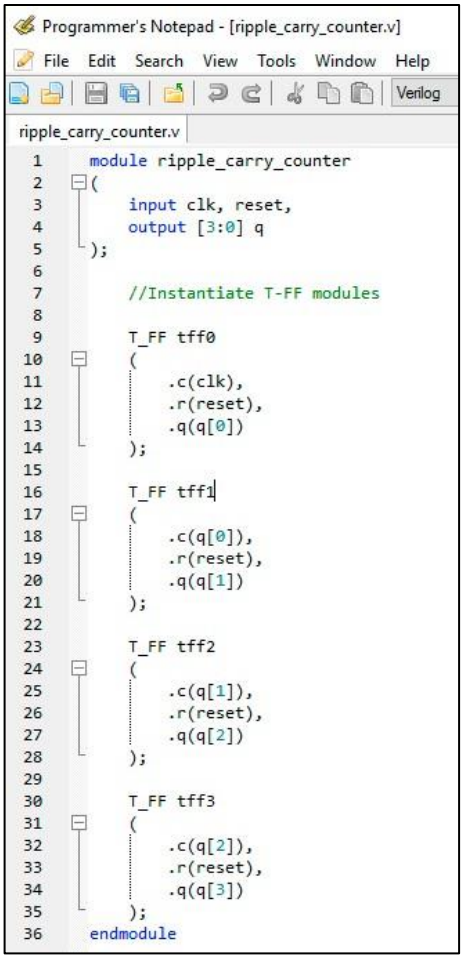


Figure 12-Verilog description of a top-level module named ripple_carry_counter

The module name should match the file name as shown in the figure above

In `ripple_carry_counter` module, shown above, four instances of the module `T_FF` (`tff0` to `tff3`) are used. Since the port name of a module being instantiated comes first, therefore it is clear from the above instantiation that the `T_FF` module has three port names `c`, `r`, and `q`.

The syntax of a not gate in verilog is: not <*instance_name*> (output, input) since there could be multiple not gates in the same module, therefore it is mandatory to define the different instance name for each instantiation.

ii. T-flipflop

We now have to define the internals of the module `T_FF`. The Verilog description of a `T_FF` module is shown in Figure 13. This module contains an instance of a `D_FF` module, having four ports. The `T_FF` module also contains a NOT gate which connects the output of a `D_FF` i.e. `q`, with its input i.e. `d`. This implementation is already shown in Figure 11.

```
1   module T_FF
2   (
3       input c, r,
4       output q
5   );
6       wire d;
7
8       //Instantiate D_FF
9       D_FF dff0
10      (
11          .c(c),
12          .r(r),
13          .q(q),
14          .d(d)
15      );
16
17      not n1(d,q);
18  endmodule
```

Figure 13-Verilog description of T_FF module

iii. D-flipflop

We are now left with designing the leaf module which is D-flipflop. The Verilog description of a `D_FF` module is shown in Figure 14. This module contains an always block which is activated either at the `posedge` (positive edge) of `r` (reset signal) or at the `negedge` (negative edge) of `c` (clock signal). When a reset signal is triggered, the output of `D_FF` resets to `0`. When reset is disserted, the output `q` retains the value of input `d` on each negative edge of a clock signal.

```
1   module D_FF
2   (
3   input c, r,
4   input d,
5   output reg q
6   );
7
8       always @ (posedge r or negedge c)
9           begin
10              if (r)
11                  q <= 1'b0;
12              else
13                  q <= d;
14          end
15
16  endmodule
```

Figure 14-Verilog implementation of D_FF

Note that the data type of the output port `q` is defined as `reg` only in `D_FF` module and not in any other top-level modules. It is because the `D_FF` is the main source of generating the output `q`, therefore it should be defined as `reg`. The generated output signal `q` then simply propagates to the top-level modules, therefore the data type of the output ports of these modules are simply `net`.

The design of 4-bit ripple carry counter module is complete.

Functional Verification

Once a design block is completed, it must be tested. The functionality of a design block can be tested by applying stimulus and checking results. We call such a block the *stimulus* block. It is good practice to keep the stimulus and design block separate. The stimulus block can be written in Verilog. A separate language is not required to describe stimulus. The stimulus block is also commonly called a *test bench*. Different test benches can be used to thoroughly test the design block.

i. Creating Testbench/Stimulus

The stimulus block instantiates the design block and directly drives the signals in the design block. In Figure 15, the stimulus block becomes the top-level block. It manipulates the signals `clk` and `reset`, and checks and displays output signal `q`.

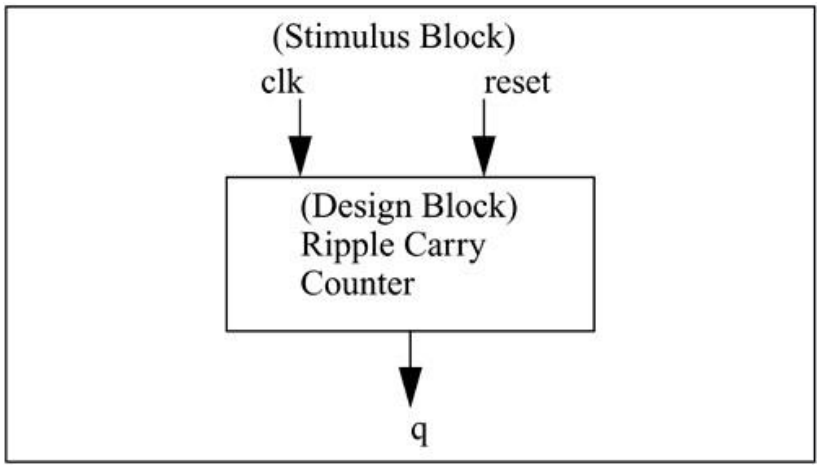


Figure 15-Stimulus block instantiates design block

In stimulus block approach (Figure 15), stimulus block acts like an environment to provide inputs to the design block and observe its outputs. Therefore, the stimulus block do not require any I/O ports interface.

We must now write the description of a stimulus block to check if the design module `ripple_carry_counter` is functioning correctly. In this case, we must control the signals `clk` and `reset` so that the regular function of the ripple carry counter and the reset mechanism are both tested.

The Verilog module of the testbench for testing `ripple_carry_counter` design is shown in Figure 16. It is clearly shown that the module `tb` in Figure 16 doesn't have any I/O ports. We only have internal connections declared as a wire/reg. Now the Design Under Verification (DUV) is instantiated and its ports are connected with the stimulus signals which, in our case, are `clk`, `reset` and `q`. Then the clock signal (`clk`) is initialized and generated with a time period of 10 units i.e. it toggles between 1 and 0 after each 5 units, so the time period is 10 units. The `initial` block (line 16) is used to assign the values at the beginning of simulation. This block is not synthesizable, and therefore only used for simulation purpose. The `always` block (line 19) enforce the clock signal to toggle continuously each after 5 time units.

```
1   module tb
2   (
3
4   );
5       reg clk;
6       reg reset;
7       wire [3:0] q;
8
9       ripple_carry_counter r1
10      (
11          .clk(clk),
12          .reset(reset),
13          .q(q)
14      );
15
16      initial
17      clk = 1'b0;
18
19      always
20      #5 clk = ~clk;
21
22      initial
23      begin
24          reset = 1'b1;
25          #15 reset = 1'b0;
26          #180 reset = 1'b1;
27      end
28
29      initial
30      $monitor("Time = ", $time, "---> Output = %d", q);
31  endmodule
```

Figure 16-Stimulus/testbench for testing ripple_carry_counter module.

Similarly, the `reset` signal is defined to be high initially; deasserted after 15 time units, and then asserted again after the total of 195 time units.
`$monitor` block is used to print the results on the transcript window of Modelsim®.

Once the testbench/stimulus is generated, we are now ready to perform simulation in Modelsim®. Lab instructor will demonstrate about Quartus and Modelsim environment.

Task

1. Launch Quartus Prime Lite on your desktop PC. Create New Project and test the 4-bit ripple carry counter. Attach printout of output waveform.

Reference

Pong P. Chu Xilinx Spartan 3 Version. FPGA Prototyping with Verilog examples, Wiley

NED University of Engineering & Technology
Department of **Electronic** Engineering
Course Code and Title: EL - 421 Embedded Electronics
Laboratory Session No. _____ Date: _____

Software Use Rubric					
Criterion	**Level of Attainment**				
	Below Average (1)	**Average (2)**	**Good (3)**	**Very Good (4)**	**Excellent (5)**
Identification of software menu (syntax, components, commands, tools, layout etc.).	Can't identify software menus.	Rarely identifies software menus.	Occasionally identifies software menus.	Able to identify software menus.	Perfectly able to identify software menus.
Skills to use software (schematic, syntax, commands, tools, layout) efficiently.	Can't use software efficiently.	Rarely uses software efficiently.	Occasionally uses software efficiently.	Often uses software efficiently.	Efficiently uses software (syntax, commands, tools, layout)
Adherence to safety procedures and handling of equipment (computing unit, peripheral devices, and other equipment in lab).	Doesn't handle equipment with required care and safety.	Rarely handles equipment with required care and safety.	Occasionally handles equipment with required care and safety.	Often handles equipment with required care and safety.	Handles equipment with required care and safety.
Ability to troubleshoot software errors (detection and debugging).	Not able to troubleshoot the errors	Rarely able to troubleshoot the errors	Occasionally able to troubleshoot the errors	Often able to troubleshoot the errors	Fully able to troubleshoot the errors
Analysis and interpretation of results/outputs.	Not able to analyze and interpret results/outputs.	Rarely able to perform the analysis and interpretation.	Occasionally able to perform the analysis and interpretation.	Often able to perform the analysis and interpretation.	Perfectly able to perform the analysis and interpretation.

Weighted CLO (Score)	
Remarks	
Instructor's Signature with Date:	

Lab Experiment 06

Objective: To *implement* RISC-V basic modules such as Multiplexer, ALU, and Immediate data generator.

Introduction

From now on, you will mostly be developing modules and simulating the results at your own. In this lab you are required to develop a multiplexer, an ALU, and an immediate data extractor.

To complete this lab, you may need to use *if/else structure*, *case structure*, *concatenation operator* or *assignment* operator in Verilog.

i. Case Structure

We already have seen the syntax and usage of if-else structure in Lab01 (D_FF implementation). The if-else structure can also be nested similar to traditional programming languages. However, the nested if-else-if can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the case statement.

The syntax of Case conditional structure, in Verilog, is shown below. The keywords `case`, `endcase`, and `default` are used in the case statement.

```
case (expression)
alternative1: statement1;
alternative2: statement2;
alternative3: statement3; ...
    default: default_statement;
 endcase
```

Each of statement1, statement2, default_statement can be a single statement or a block of multiple statements. A block of multiple statements must be grouped by keywords `begin` and `end`. The expression is compared to the alternatives in the order they are written. For the first alternative that matches, the corresponding statement or block is executed. If none of the alternatives matches, the default_statement is executed.

Placing of multiple default statements in one case statement is not allowed.

The default_statement is optional.

Task: Multiplexer

Develop a 2x1 multiplexer in which the two inputs are `a` and `b`, and each of them are 32-bits wide, as shown in Figure 17.

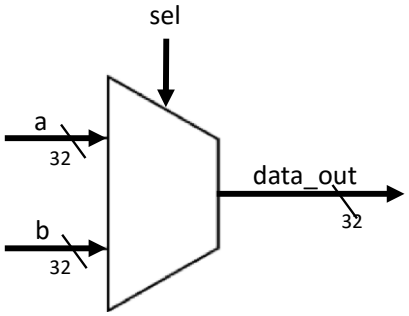


Figure 17-2x1 multiplexer

Write a testbench to simulate its behavior in ModelSim.

Arithmetic and Logical Unit (ALU)

The arithmetic logic unit (ALU) is the brain of the computer, the device that performs the arithmetic operations like addition and subtraction or logical operations like AND and OR.

Table 1-ALU operation

ALU Operation [2:0]	Function
000	Add
001	Subtract
101	Set less than
011	Or
010	And

Task

You are required to develop a behavioral model of 32-bit ALU just by declaring `a` and `b` 32-bits wide and declare the corresponding operations using a single multiplexer (refer Table 1). You also need to add an additional output named ZERO in your 32-bit ALU, as shown in Figure 18. The ZERO output should be set to 1 if the `Result` is 0, else set it to 0.

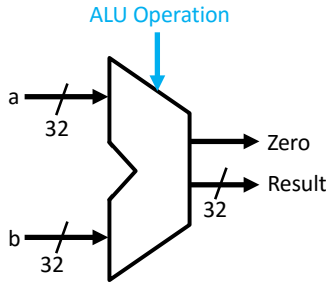


Figure 18-32-bit ALU with ZERO output

The behavioral model of ALU can be implemented using if-else or case structures.

Immediate Data Generator

Develop a module which takes the 32-bit input `instruction` and extracts the 12-bit immediate data field depending on the type of instruction. Then sign-extend these 12-bits to 32-bits output `imm_data`, as shown in Figure 19.

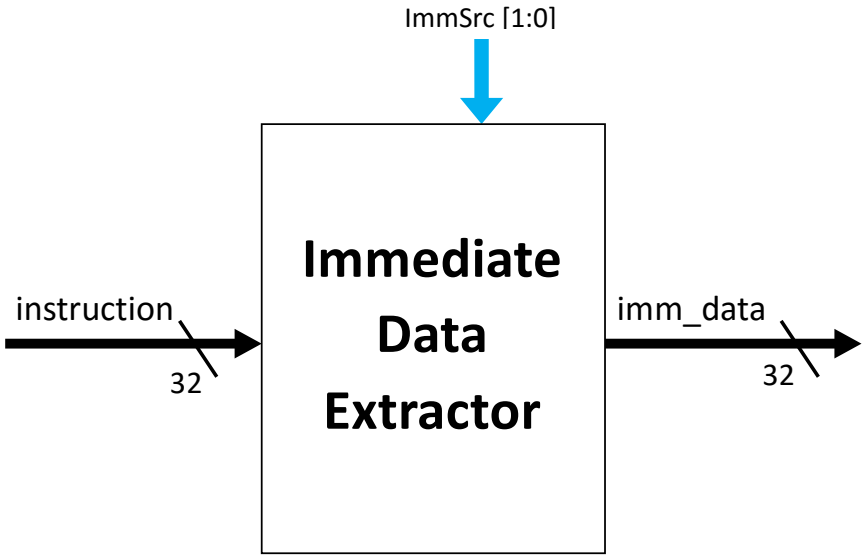


Figure 19-Immediate_data_generator module

The immediate generation logic must choose between sign-extending a 12-bit field in instruction bits 31:20 for load instructions, bits 31:25 and 11:7 for store instructions, or bits 31, 7, 30:25, and 11:8 for the conditional branch as shown in Table 2.

Table 2-ImmSrc encoding

ImmSrc	ImmExt	Type
00	{{20{instruction[31]}},instruction[31:20]}	I
01	{{20{instruction[31]}},instruction[31:25],instruction[11:7]}	S
10	{{20{instruction[31]}},instruction[7],instruction[30:25],instruction[11:8]}	B

Write a testbench for this module and verify its functionality.

NED University of Engineering & Technology
Department of **Electronic Engineering**
Course Code and Title: EL - 421 Embedded Electronics
Laboratory Session No. _____ Date: _____

Software Use Rubric					
Criterion	**Level of Attainment**				
	Below Average (1)	**Average (2)**	**Good (3)**	**Very Good (4)**	**Excellent (5)**
Identification of software menu (syntax, components, commands, tools, layout etc.).	Can't identify software menus.	Rarely identifies software menus.	Occasionally identifies software menus.	Able to identify software menus.	Perfectly able to identify software menus.
Skills to use software (schematic, syntax, commands, tools, layout) efficiently.	Can't use software efficiently.	Rarely uses software efficiently.	Occasionally uses software efficiently.	Often uses software efficiently.	Efficiently uses software (syntax, commands, tools, layout)
Adherence to safety procedures and handling of equipment (computing unit, peripheral devices, and other equipment in lab).	Doesn't handle equipment with required care and safety.	Rarely handles equipment with required care and safety.	Occasionally handles equipment with required care and safety.	Often handles equipment with required care and safety.	Handles equipment with required care and safety.
Ability to troubleshoot software errors (detection and debugging).	Not able to troubleshoot the errors	Rarely able to troubleshoot the errors	Occasionally able to troubleshoot the errors	Often able to troubleshoot the errors	Fully able to troubleshoot the errors
Analysis and interpretation of results/outputs.	Not able to analyze and interpret results/outputs.	Rarely able to perform the analysis and interpretation.	Occasionally able to perform the analysis and interpretation.	Often able to perform the analysis and interpretation.	Perfectly able to perform the analysis and interpretation.

Weighted CLO (Score)	
Remarks	
Instructor's Signature with Date:	

Lab Experiment 07

Objective: To *implement* Register file for 32-bit RISC-V processor.

Register File

Register files are necessary for computer memory. For a computer to be able to function, it needs to have some form of memory. There are two different forms of memory devices: external memory devices and local memory devices. External memory devices are items such as RAM, ROM, disk drives, etc. Registers provide local memory, which allows for temporary storage of data that is about to be processed. For this lab, registers will be used to save and store numbers.

Task

Create a module named, *`registerFile`*, which should have input address ports for reading register 1, register 2, and for writing data in a destination register. Furthermore, it should have a 32-bits data input port, and two 32-bits data output ports to read the values from two different registers. Finally, it should have a `clk`, and a `RegWrite` signal which controls when the data should be written to a register. You need to define 32 32-bit internal registers.

The top-level diagram of this module is shown in the figure below.

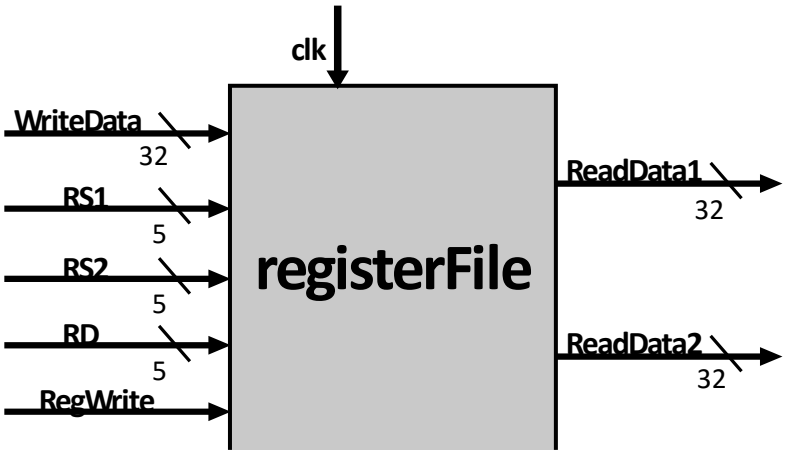


Figure 20-Register File

As an example, the following command creates an `Array` of type `reg` containing 10 elements, each of which are 5 bits wide.

```
reg [4:0] Array [9:0]
```

Where `Array` is the name of a variable. In your case, it should be `Registers`.

Design Requirements

- Initialize `Registers` with random values.

Use `initial` block to initialize the `Registers` with any random value.

To verify correct functioning of your design using test bench, make sure that different values are initialized in different `Registers`.

- The operation of writing data into a `Registers` should always be done when there is a positive edge of `clk` and `RegWrite` signal is asserted (or set i.e. HIGH).
- Reading a data from the register file should be made independent of `clk` signal. Reading should rather be sensitive to the change in inputs `RS1`, `RS2`, or Registers.

Create a test bench to verify the correct functioning of designed module in ModelSim.

NED University of Engineering & Technology
Department of **Electronic** Engineering
Course Code and Title: EL - 421 Embedded Electronics
Laboratory Session No. _____ Date: _____

Software Use Rubric					
Criterion	**Level of Attainment**				
	Below Average (1)	**Average (2)**	**Good (3)**	**Very Good (4)**	**Excellent (5)**
Identification of software menu (syntax, components, commands, tools, layout etc.).	Can't identify software menus.	Rarely identifies software menus.	Occasionally identifies software menus.	Able to identify software menus.	Perfectly able to identify software menus.
Skills to use software (schematic, syntax, commands, tools, layout) efficiently.	Can't use software efficiently.	Rarely uses software efficiently.	Occasionally uses software efficiently.	Often uses software efficiently.	Efficiently uses software (syntax, commands, tools, layout)
Adherence to safety procedures and handling of equipment (computing unit, peripheral devices, and other equipment in lab).	Doesn't handle equipment with required care and safety.	Rarely handles equipment with required care and safety.	Occasionally handles equipment with required care and safety.	Often handles equipment with required care and safety.	Handles equipment with required care and safety.
Ability to troubleshoot software errors (detection and debugging).	Not able to troubleshoot the errors	Rarely able to troubleshoot the errors	Occasionally able to troubleshoot the errors	Often able to troubleshoot the errors	Fully able to troubleshoot the errors
Analysis and interpretation of results/outputs.	Not able to analyze and interpret results/outputs.	Rarely able to perform the analysis and interpretation.	Occasionally able to perform the analysis and interpretation.	Often able to perform the analysis and interpretation.	Perfectly able to perform the analysis and interpretation.

Weighted CLO (Score)	
Remarks	
Instructor's Signature with Date:	

Lab Experiment 08

Objective: To *implement* instruction and data memory Verilog modules for 32-bit RISC-V processor.

Instruction Memory

Instruction and Data memories are essential components of a processor. We will develop modules for the Instruction and Data memory separately. The instruction memory is used to store instructions. During the processor execution, it is required to only read the instruction from the instruction memory and not to write any data or instruction in it. Hence, we could say that the instruction memory is the read-only memory. However, the data memory is used to both read and write data.

Recall that the size of each instruction is 32-bits. Each location in instruction and data memory is 32-bits (4 bytes) wide.

In this lab, we will design an instruction memory with each location 8-bits wide. Hence, four memory locations will be required to store a 32-bit instruction in an instruction memory, as shown in Figure 21.

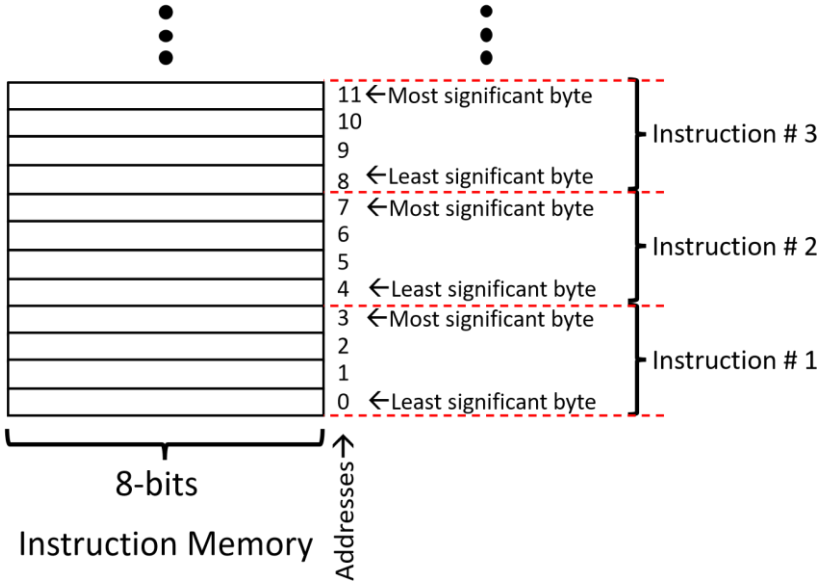


Figure 21-8xN bytes instruction memory, where N specify the number of n memory locations each of which are 8-bits (or 1-byte) wide

Task

Design a module named, `Instruction_Memory`, having a 32-bit input, `Inst_Address`, and a 32-bit output, `Instruction`. Now, declare a 1x16 bytes instruction memory using *.mif* file and initialized its contents as shown in Figure 22-Memory contentsFigure 22. The behavior of this module should be designed as such that whenever an `Inst_Address` field is changed, the 32-bit instruction corresponding to the `Inst_Address` should appear at the 32-bit output port, `Instruction`.

15	11111110
14	01000010
13	00001010
12	11100011
11	00000000
10	01100010
9	11100010
8	00110011
7	00000000
6	01100100
5	10100100
4	00100011
3	11111111
2	11000100
1	10100011
0	00000011

Figure 22-Memory contents

For example, if the `Inst_Address` is 0, the consecutive 4 bytes (byte no. 0 to 3) should appear at the 32-bit `Instruction` output. Similarly, when the value of `Inst_Address` is 8, the corresponding next four bytes (8 to 11) should be placed at the 32-bit output port, `Instruction`.

While placing four bytes on the 32-bit `Instruction` port, make sure that the bytes are in correct order. That is, the least-significant byte should be the right most byte and that the most-significant byte should be the left most byte in the 32bit `Instruction` output.

Write a testbench and verify the functionality of this module.

Data memory

The top-level diagram of a Data Memory is shown in Figure 23. Write a module named, Data_Memory, with inputs and outputs as shown in Figure 23. The signals shown in blue color are the control signals. We will design the control unit in upcoming labs.
The data at the input port, Write_Data, should only be written at the positive edge of clk signal and when `MemWrite` signal is asserted (i.e. High). The data can be read from memory at any instant whenever the `Mem_Addr` value changes.
Initialized the memory with random data, write a testbench and verify its functionality.

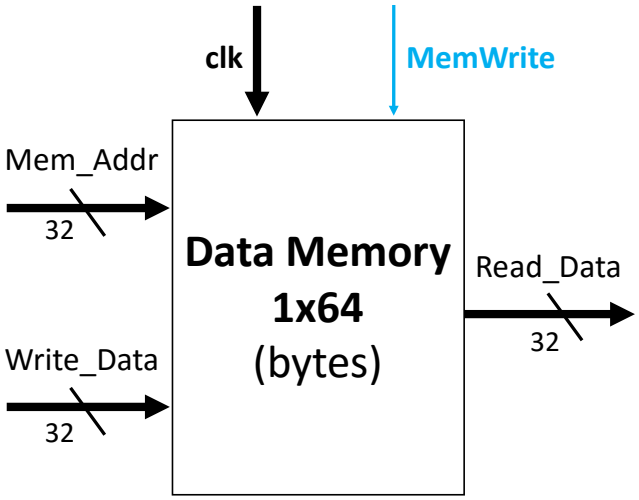


Figure 23-IOs of Data Memory module. There are 64 memory locations, each of which is 1 byte wide.

NED University of Engineering & Technology
Department of **Electronic** Engineering
Course Code and Title: EL - 421 Embedded Electronics
Laboratory Session No. _____ Date: _____

Software Use Rubric					
Criterion	**Level of Attainment**				
	Below Average (1)	**Average (2)**	**Good (3)**	**Very Good (4)**	**Excellent (5)**
Identification of software menu (syntax, components, commands, tools, layout etc.).	Can't identify software menus.	Rarely identifies software menus.	Occasionally identifies software menus.	Able to identify software menus.	Perfectly able to identify software menus.
Skills to use software (schematic, syntax, commands, tools, layout) efficiently.	Can't use software efficiently.	Rarely uses software efficiently.	Occasionally uses software efficiently.	Often uses software efficiently.	Efficiently uses software (syntax, commands, tools, layout)
Adherence to safety procedures and handling of equipment (computing unit, peripheral devices, and other equipment in lab).	Doesn't handle equipment with required care and safety.	Rarely handles equipment with required care and safety.	Occasionally handles equipment with required care and safety.	Often handles equipment with required care and safety.	Handles equipment with required care and safety.
Ability to troubleshoot software errors (detection and debugging).	Not able to troubleshoot the errors	Rarely able to troubleshoot the errors	Occasionally able to troubleshoot the errors	Often able to troubleshoot the errors	Fully able to troubleshoot the errors
Analysis and interpretation of results/outputs.	Not able to analyze and interpret results/outputs.	Rarely able to perform the analysis and interpretation.	Occasionally able to perform the analysis and interpretation.	Often able to perform the analysis and interpretation.	Perfectly able to perform the analysis and interpretation.

Weighted CLO (Score)	
Remarks	
Instructor's Signature with Date:	

Lab Experiment 09

Objective: To *implement* Verilog module for instruction fetch data path.

Instruction Fetch Data Path

A reasonable way to start a data path design is to examine the major components required to execute each class of RISC-V instructions. First, an instruction has to be fetched, which requires the components shown in Figure 24.

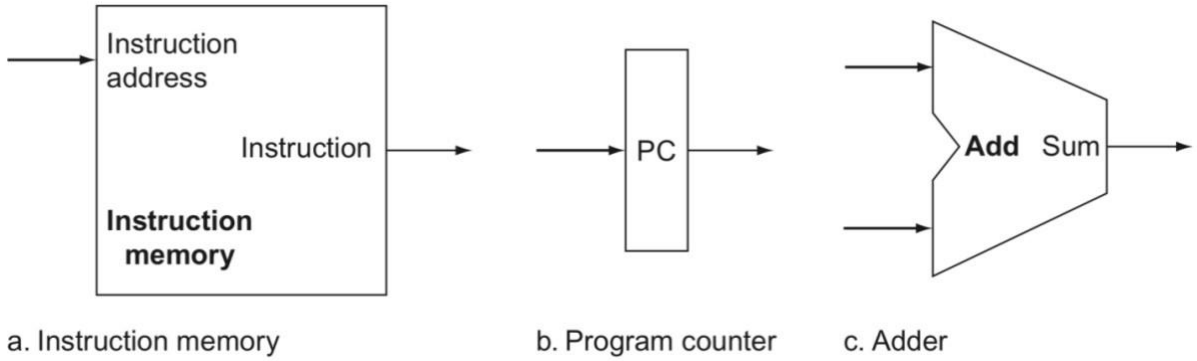


Figure 24-Required components for fetching a processor instruction

As shown in Figure 24, two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.

Implementation

We have already developed an instruction memory in Lab07. Now, we will start off with developing the separate modules of a program counter (PC) and a two-input adder.

Task (s)

1. Write a module, named `Program_Counter`, which takes three inputs – `clk`, a 32-bit input, `PC_In` and `reset`; and a 32-bit output, `PC_Out`. Initialize `PC_Out` to 0 if reset signal is high, else reflect the value of `PC_In` to `PC_Out`, at the positive edge of clock.
2. Adder takes two 32-bits inputs, `a` and `b`; add them, and reflect the results at the 32-bit output port, `out`.
3. Now connect the above two modules and an `Instruction_Memory` (developed in Lab07) to construct an instruction fetch datapath as shown below. Name the module

`Instruction_Fetch`; instantiate all three modules and make necessary connections as shown in Figure 25.

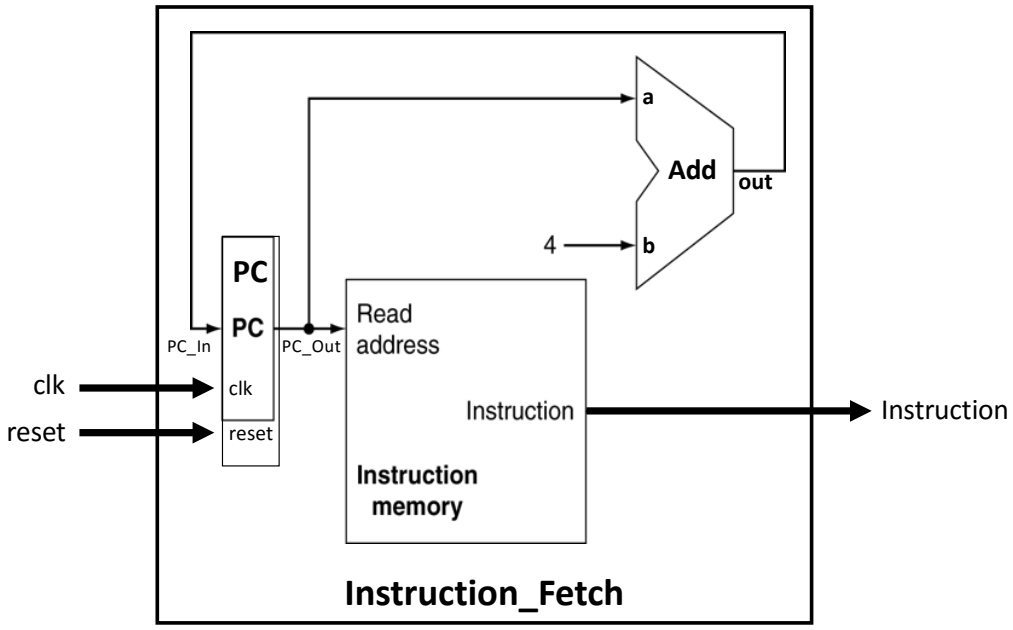


Figure 25-Instruction fetch datapath

Write a testbench and verify the functionality of instruction fetch datapath.

NED University of Engineering & Technology
Department of **Electronic** Engineering
Course Code and Title: EL - 421 Embedded Electronics
Laboratory Session No. _____ Date: _____

Software Use Rubric					
Criterion	**Level of Attainment**				
	Below Average (1)	**Average (2)**	**Good (3)**	**Very Good (4)**	**Excellent (5)**
Identification of software menu (syntax, components, commands, tools, layout etc.).	Can't identify software menus.	Rarely identifies software menus.	Occasionally identifies software menus.	Able to identify software menus.	Perfectly able to identify software menus.
Skills to use software (schematic, syntax, commands, tools, layout) efficiently.	Can't use software efficiently.	Rarely uses software efficiently.	Occasionally uses software efficiently.	Often uses software efficiently.	Efficiently uses software (syntax, commands, tools, layout)
Adherence to safety procedures and handling of equipment (computing unit, peripheral devices, and other equipment in lab).	Doesn't handle equipment with required care and safety.	Rarely handles equipment with required care and safety.	Occasionally handles equipment with required care and safety.	Often handles equipment with required care and safety.	Handles equipment with required care and safety.
Ability to troubleshoot software errors (detection and debugging).	Not able to troubleshoot the errors	Rarely able to troubleshoot the errors	Occasionally able to troubleshoot the errors	Often able to troubleshoot the errors	Fully able to troubleshoot the errors
Analysis and interpretation of results/outputs.	Not able to analyze and interpret results/outputs.	Rarely able to perform the analysis and interpretation.	Occasionally able to perform the analysis and interpretation.	Often able to perform the analysis and interpretation.	Perfectly able to perform the analysis and interpretation.

Weighted CLO (Score)	
Remarks	
Instructor's Signature with Date:	

Lab Experiment 10

Objective: To *implement* Verilog module for Control unit of RISC-V processor.

Introduction

The last module we are left with is the control unit, which we have to design before we proceed further to integrate the already-designed processor components. In this lab, we will develop a module for generating control signals for specific instructions. These control signals are used to control the data flow and enabling or disabling the modules which are not needed for specific instructions.

Control Unit

Besides control unit, we also have to develop ALU Control unit to set the control signals of ALU.

Task(s)

1. As shown in Figure 26, write a module, named`Control_Unit`, which takes a 7-bit wide input, named `Opcode`, and generate 7 output signals. Out of these seven outputs, one is `ALUOp` which is 2-bits wide, and the remaining six are 1-bit wide, which are `Branch`, `MemRead`, `MemtoReg`, `MemWrite`, `ALUSrc`, and `RegWrite`.

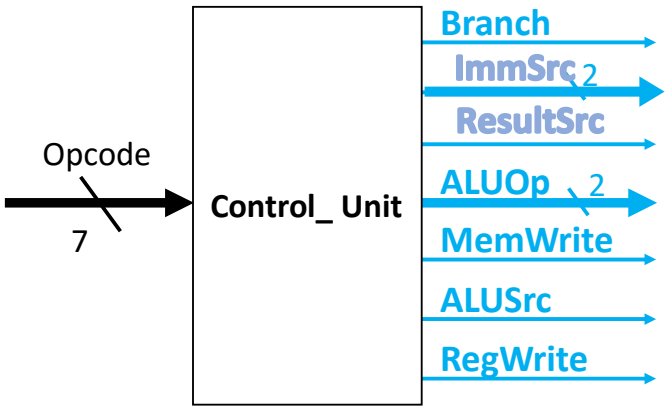


Figure 26-I/O diagram of Control Unit

The behavior of Control_Unit module should be designed according to the Table 3.

Table 3-This table shows how the control signals are set based on the input values of Opcode

Instruction Type	Opcode	ALUSrc	ResultSrc	ImmSrc [1:0]	RegWrite	MemWrite	Branch	ALUOp [1:0]
R-Type	0110011	0	0	XX	1	0	0	10
I-Type (lw)	0000011	1	1	00	1	0	0	00

I-Type (sw)	0100011	1	X	01	0	1	0	00
SB-Type (beq)	1100011	0	X	10	0	0	1	01
R-Type (addi)	0010011	1	0	00	1	0	0	10

2. Write a module, named `ALU_Control`, which takes single bit input $funct7_5$, a 2-bit input, named `ALUOp`, and a 3-bit input, named $Funct3_{2:0}$, and produces a 3-bit output, named `ALUControl`, as shown in Figure 27.

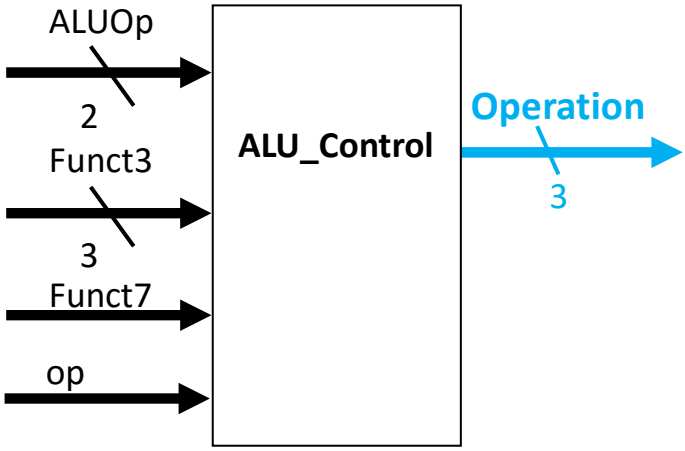


Figure 27-I/O diagram of ALU_Control module

The values of the output, `ALUControl`, should be set based on the input signals, `ALUOp`, `Funct3` and `funct7`, as shown in Table 4.

Table 4-ALU_Control truth table

Instruction Type	ALUOp [1:0]	[op_5, $funct7_5$]	Funct3	ALUControl
I/S-Type (ld, sd)	00	X	xxx	000 (add)
SB-Type (Beq)	01	X	xxx	001 (subtract)
R-Type	10	00,01,10	000	000 (add)
		11	000	001 (subtract)
		X	010	101 (set less than)
		X	110	011 (or)
		X	111	010 (and)

NED University of Engineering & Technology
Department of **Electronic** Engineering
Course Code and Title: EL - 421 Embedded Electronics
Laboratory Session No. _____ Date: _____

Software Use Rubric					
Criterion	**Level of Attainment**				
	Below Average (1)	**Average (2)**	**Good (3)**	**Very Good (4)**	**Excellent (5)**
Identification of software menu (syntax, components, commands, tools, layout etc.).	Can't identify software menus.	Rarely identifies software menus.	Occasionally identifies software menus.	Able to identify software menus.	Perfectly able to identify software menus.
Skills to use software (schematic, syntax, commands, tools, layout) efficiently.	Can't use software efficiently.	Rarely uses software efficiently.	Occasionally uses software efficiently.	Often uses software efficiently.	Efficiently uses software (syntax, commands, tools, layout)
Adherence to safety procedures and handling of equipment (computing unit, peripheral devices, and other equipment in lab).	Doesn't handle equipment with required care and safety.	Rarely handles equipment with required care and safety.	Occasionally handles equipment with required care and safety.	Often handles equipment with required care and safety.	Handles equipment with required care and safety.
Ability to troubleshoot software errors (detection and debugging).	Not able to troubleshoot the errors	Rarely able to troubleshoot the errors	Occasionally able to troubleshoot the errors	Often able to troubleshoot the errors	Fully able to troubleshoot the errors
Analysis and interpretation of results/outputs.	Not able to analyze and interpret results/outputs.	Rarely able to perform the analysis and interpretation.	Occasionally able to perform the analysis and interpretation.	Often able to perform the analysis and interpretation.	Perfectly able to perform the analysis and interpretation.

Weighted CLO (Score)	
Remarks	
Instructor's Signature with Date:	

Lab Experiment 11

Objective: To *implement* a single-cycle RISC processor by integrating previously designed Verilog modules.

Introduction

We have developed all the necessary modules of a single cycle processor. We can now combine all the pieces to make a simple data path for the core RISC-V architecture to run specific set of instructions, which include R-Type, I-Type and Branch-Type instructions.

Task(s)

In this lab, we will use the modules developed in the previous labs and integrate them together to construct a single-cycle data path processor. Copy the following modules in a new folder, named Lab11

1. Mux.v from /Lab05 folder
2. ImmGen.v, ALU.v from /Lab06 folder.
3. RegisterFile.v from /Lab07 folder.
4. Data_Memory.v and Instruction_Memory.v from /Lab08 folder.
5. Program_Counter.v and Adder.v from /Lab09 folder.
6. ALU_Control.v and Control_Unit.v from /Lab10 folder.

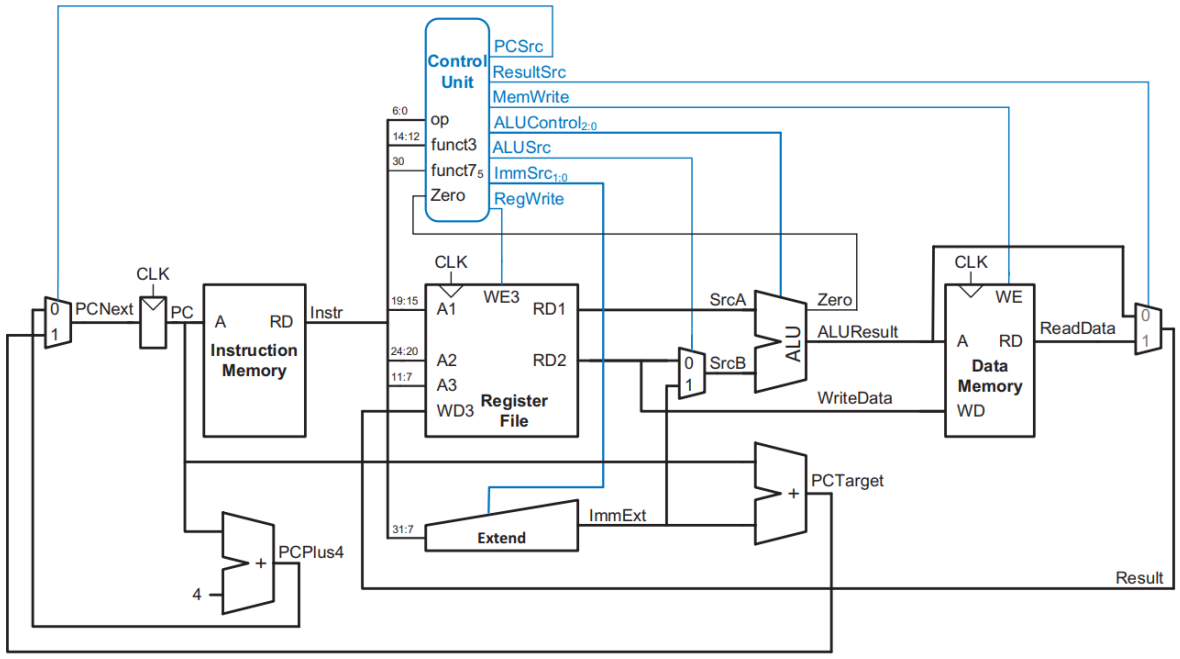


Figure 28-Single Cycle Processor

Now integrate all the above modules in a top module named, `RISC_V_Processor`, according to the processor diagram shown in Figure 28. The inputs to the top-level module are `clk` and `reset`, and there is no output.

Load instruction memory with the contents shown below

	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$		rs1	f3	rd	op	
L7:	lw x6, -4(x9)	I	111111111100		01001	010	00110	0000011	FFC4A303
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423
			funct7	rs2	rs1	f3	rd	op	
	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

1. Write a testbench. Initialize the simulation by first resetting the module under test for, say, 10ns. Toggle the clock signal at every 5ns.
2. Calculate the expected results of decoded instructions using the contents of registers and memory array initialized in modules. Compare them with the simulation results and briefly describe.

Since the top module doesn't have any particular output, so you can verify results by observing the following signals:

Instruction Address
Instruction
WriteData
ReadData1, ReadData2
Data_Memory Read Data

NED University of Engineering & Technology
Department of **Electronic** Engineering
Course Code and Title: EL - 421 Embedded Electronics
Laboratory Session No. _____ Date: _____

Software Use Rubric					
Criterion	**Level of Attainment**				
	Below Average (1)	**Average (2)**	**Good (3)**	**Very Good (4)**	**Excellent (5)**
Identification of software menu (syntax, components, commands, tools, layout etc.).	Can't identify software menus.	Rarely identifies software menus.	Occasionally identifies software menus.	Able to identify software menus.	Perfectly able to identify software menus.
Skills to use software (schematic, syntax, commands, tools, layout) efficiently.	Can't use software efficiently.	Rarely uses software efficiently.	Occasionally uses software efficiently.	Often uses software efficiently.	Efficiently uses software (syntax, commands, tools, layout)
Adherence to safety procedures and handling of equipment (computing unit, peripheral devices, and other equipment in lab).	Doesn't handle equipment with required care and safety.	Rarely handles equipment with required care and safety.	Occasionally handles equipment with required care and safety.	Often handles equipment with required care and safety.	Handles equipment with required care and safety.
Ability to troubleshoot software errors (detection and debugging).	Not able to troubleshoot the errors	Rarely able to troubleshoot the errors	Occasionally able to troubleshoot the errors	Often able to troubleshoot the errors	Fully able to troubleshoot the errors
Analysis and interpretation of results/outputs.	Not able to analyze and interpret results/outputs.	Rarely able to perform the analysis and interpretation.	Occasionally able to perform the analysis and interpretation.	Often able to perform the analysis and interpretation.	Perfectly able to perform the analysis and interpretation.

Weighted CLO (Score)	
Remarks	
Instructor's Signature with Date:	

Lab Experiment 12

Objective: To *implement* Verilog module for SPI communication between FPGA and a peripheral.

Introduction

Serial Peripheral Interface (SPI) is a simple synchronous serial protocol that is easy to use and relatively fast. The physical interface consists of three pins: serial clock (SCK), serial data out (SDO), and serial data in (SDI). SPI connects a controller device to a peripheral device, as shown in Figure 29(a). The controller produces the clock. It initiates communication by sending clock pulses on SCK. The controller sends data from its SDO pin to the peripheral's SDI pin one bit per cycle, starting with the most significant bit. The peripheral may simultaneously respond with its SDO pin back to the controller's SDI pin. Figure 29(b) shows the SPI waveforms for an 8-bit data transmission. Bits change on the falling edge of SCK and are stable to sample on the rising edge. The SPI interface may also send an active-low chip enable to alert the receiver that data is coming.

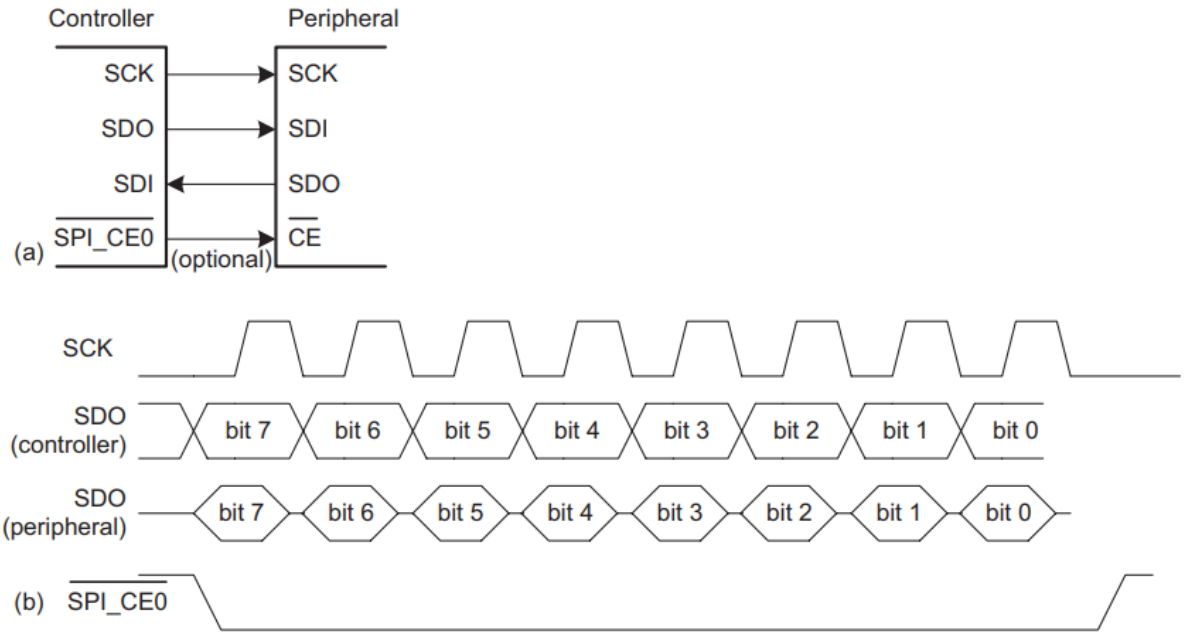


Figure 29-SPI configuration: (a) SPI controller-peripheral connection diagram, (b) Example SPI data signals

Task

HDL Example code below gives the Verilog code for an SPI peripheral that can both send and receive data (i.e., an SPI transceiver), and Figure 30 shows its block diagram and timing with CPHA = CPOL = 0. The main component is still a shift register, shown on the right of Figure 30. The shift register parallel loads the byte to send (d[7:0]) into the shift register and then shifts out this data on sdo while it shifts in data transmitted from the controller (t[7:0]) on sdi. A counter, cnt, keeps track of how many bits have been sent/received. When sck is idle, cnt = 0 and the most significant bit of d (d[7]) sits on the sdo wire. One subtlety is that sdo can only change on

the falling clock edge, so the sdo output (which is the most significant bit of the shift register, q[7], is delayed by half a clock cycle by the negative-edge triggered qdelayed register on the bottom left of Figure 30.

```
Example SPI Verilog HDL code
module spi_peripheral(input logic sck, // From controller
input logic sdi, // From controller
output logic sdo, // To controller
input logic reset, // System reset
input logic [7:0] d, // Data to send
output logic [7:0] q); // Data received
logic [2:0] cnt;
logic qdelayed;
// 3-bit counter tracks when full byte is transmitted
always_ff @(negedge sck, posedge reset)
if (reset)
cnt = 0;
else
cnt = cnt + 3'b1;
// Loadable shift register
// Loads d at the start, shifts sdi into bottom on each step
always_ff @(posedge sck)
q < = (cnt = = 0) ? {d[6:0], sdi} : {q[6:0], sdi};
// Align sdo to falling edge of sck
// Load d at the start
always_ff @(negedge sck)
qdelayed = q[7];
assign sdo = (cnt == 0) ? d[7] : qdelayed;
endmodule
```

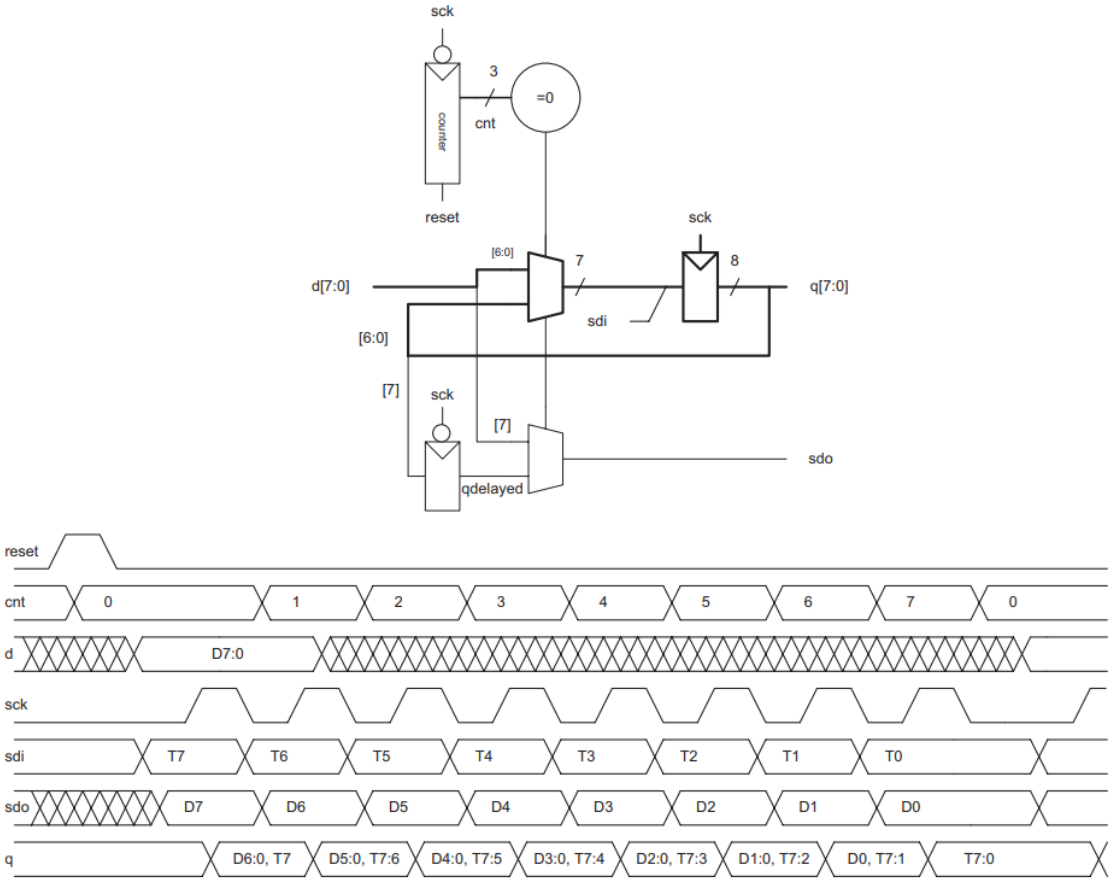


Figure 30- Block and timing diagram for SPI peripheral on FPGA

Reference

Harris, D., & Harris, S. (2010). Digital design and computer architecture. Morgan Kaufmann.

NED University of Engineering & Technology
Department of **Electronic** Engineering
Course Code and Title: EL - 421 Embedded Electronics
Laboratory Session No. _____ Date: _____

Software Use Rubric					
Criterion	**Level of Attainment**				
	Below Average (1)	**Average (2)**	**Good (3)**	**Very Good (4)**	**Excellent (5)**
Identification of software menu (syntax, components, commands, tools, layout etc.).	Can't identify software menus.	Rarely identifies software menus.	Occasionally identifies software menus.	Able to identify software menus.	Perfectly able to identify software menus.
Skills to use software (schematic, syntax, commands, tools, layout) efficiently.	Can't use software efficiently.	Rarely uses software efficiently.	Occasionally uses software efficiently.	Often uses software efficiently.	Efficiently uses software (syntax, commands, tools, layout)
Adherence to safety procedures and handling of equipment (computing unit, peripheral devices, and other equipment in lab).	Doesn't handle equipment with required care and safety.	Rarely handles equipment with required care and safety.	Occasionally handles equipment with required care and safety.	Often handles equipment with required care and safety.	Handles equipment with required care and safety.
Ability to troubleshoot software errors (detection and debugging).	Not able to troubleshoot the errors	Rarely able to troubleshoot the errors	Occasionally able to troubleshoot the errors	Often able to troubleshoot the errors	Fully able to troubleshoot the errors
Analysis and interpretation of results/outputs.	Not able to analyze and interpret results/outputs.	Rarely able to perform the analysis and interpretation.	Occasionally able to perform the analysis and interpretation.	Often able to perform the analysis and interpretation.	Perfectly able to perform the analysis and interpretation.

b.

Weighted CLO (Score)	
Remarks	
Instructor's Signature with Date:	

Lab Experiment 13 (Open-ended Lab)

Objective: To *build* a 5-stage RISC-V pipelined processor capable of executing provided assembly instructions.

Background

The speed of a system is characterized by the latency and throughput of information moving through it. We define a *token* to be a group of inputs that are processed to produce a group of outputs. The term conjures up the notion of placing subway tokens on a circuit diagram and moving them around to visualize data moving through the circuit. The *latency* of a system is the time required for one token to pass through the system from start to end. The *throughput* is the number of tokens that can be produced per unit time. As you might imagine, the throughput can be improved by processing several tokens at the same time. This is called *parallelism*, which comes in two forms: spatial and temporal. With *spatial parallelism*, multiple copies of the hardware are provided so that multiple tasks can be done at the same time. With *temporal parallelism*, a task is broken into stages, like an assembly line. Multiple tasks can be spread across the stages. Although each task must pass through all stages, a different task will be in each stage at any given time, so multiple tasks can overlap. Temporal parallelism is commonly called *pipelining*. Spatial parallelism is sometimes just called *parallelism*.

We design a pipelined processor by subdividing the single-cycle processor into five pipeline stages. Thus, five instructions can execute simultaneously, one in each stage. Because each stage has only one-fifth of the entire logic, the clock frequency is approximately five times faster. So, ideally, the latency of each instruction is unchanged, but the throughput is five times better. Microprocessors execute millions or billions of instructions per second, so throughput is more important than latency. Figure 31 shows 5 stage pipelined RISC-V processor with control signals. Write Verilog implementation of provided RISC-V pipelined processor.

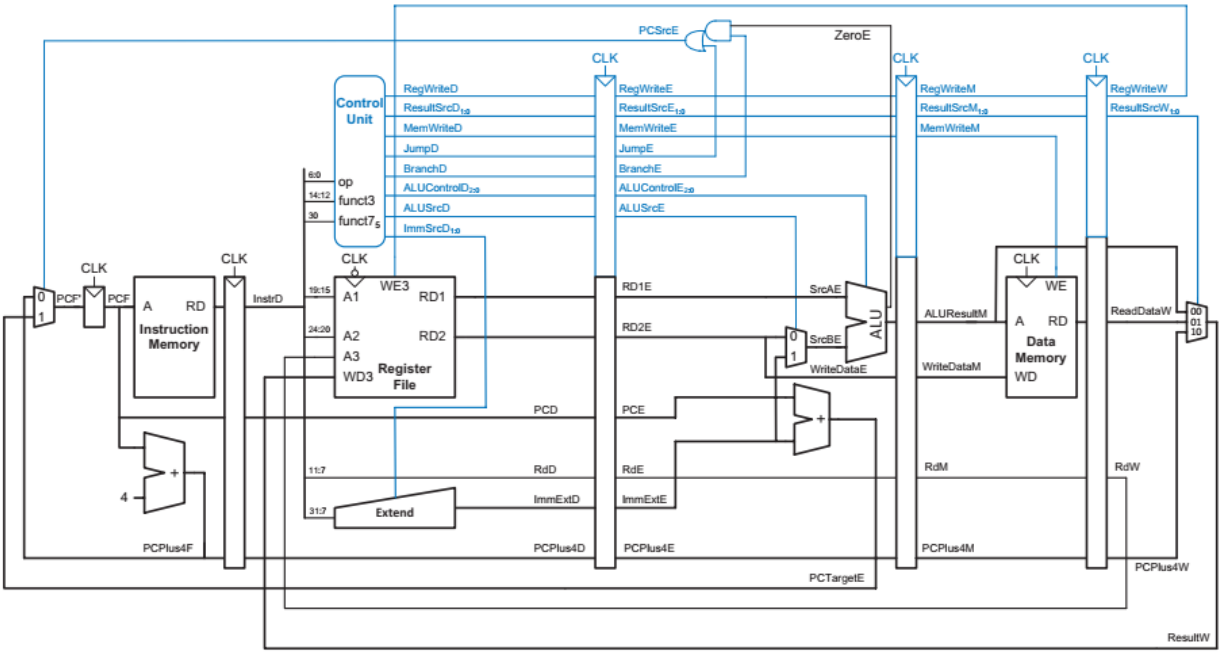


Figure 31-Pipelined processor with Control

Submission Deliverables:

1. You will submit all your codes (the Verilog files for modules as well as test benches) task wise.
2. You will submit a PDF report containing explanations of how you implemented your processor, test cases and results, any difficulties you had and how you overcame them, and any deficiencies in your projects if there are any. Also, add all the codes in appendices.

Assembly Instructions to execute:

```
addi x22,x0,0 # i - loop variable
addi x23,x0,0 # j- loop variable
addi x10,x0,10 # Maximum Loop count

# To load the array
Loop1:
slli x24, x22, 2
sw x22,0x200(x24)
addi x22,x22,1
bne x22,x10,Loop1

addi x22,x0,0 # initializing loop variable i for #Loop2
Loop2:
slli x24, x22, 2 # i*4 for offset of array
add x23,x22,x0   # j=i, first value of j initialized for inner loop
        Loop3:
        slli x25, x23, 2 # j*4 for offset of array
        lw x1,0x200(x24) # a[i]
        lw x2,0x200(x25) # a[j]
bge x1,x2,EndIf       # if a[i]>=a[j], then end the loop
# if a[i]<a[j], then swap
add x5,x1,x0              # temp <- a[j]
        sw x2,0x200(x24)        # a[i] <- a[j]
sw x5,0x200(x25)        # a[j] <- temp
EndIf:
addi x23,x23,1
        bne x23,x10,Loop3     # Repeat Loop3 with next value of j
addi x22,x22,1          # Else end Loop3 and
bne x22,x10,Loop2  # and go to Loop 2 with next value of i
```

Reference

Harris, D., & Harris, S. (2010). Digital design and computer architecture. Morgan Kaufmann.

NED University of Engineering & Technology
Department of **Electronic** Engineering
Course Code and Title: EL - 421 Embedded Electronics
Laboratory Session No. _____ Date: _____

Psychomotor Domain Assessment Rubric-Level P3					
Skill Sets	Extent of Achievement				
	0	1	2	3	4
Pipelined processor implementation	Not implemented.	Incorrect logic/pipeline register modules.	Incomplete implementation /few conditions are not included.	Implemented with few errors.	Correctly implemented.
Reporting the results	Results are not discussed.	Many observations are not reported.	Some significant observations are missed but report covers overall objective of the lab session. Results are briefly discussed.	Few relevant observations are missed. Results are discussed.	All relevant observations are reported and results are thoroughly assessed.
Group Work	Doesn't participate and contribute.	Slightly participates and contributes.	Somewhat participates and contributes.	Moderately participates and contributes.	Fully participates and contributes.

Weighted CLO (Psychomotor Score)	
Remarks	
Instructor's Signature with Date:	