SEMESTER PROJECT

COMPUTER ARCHITECTURE AND LOGIC DESIGN

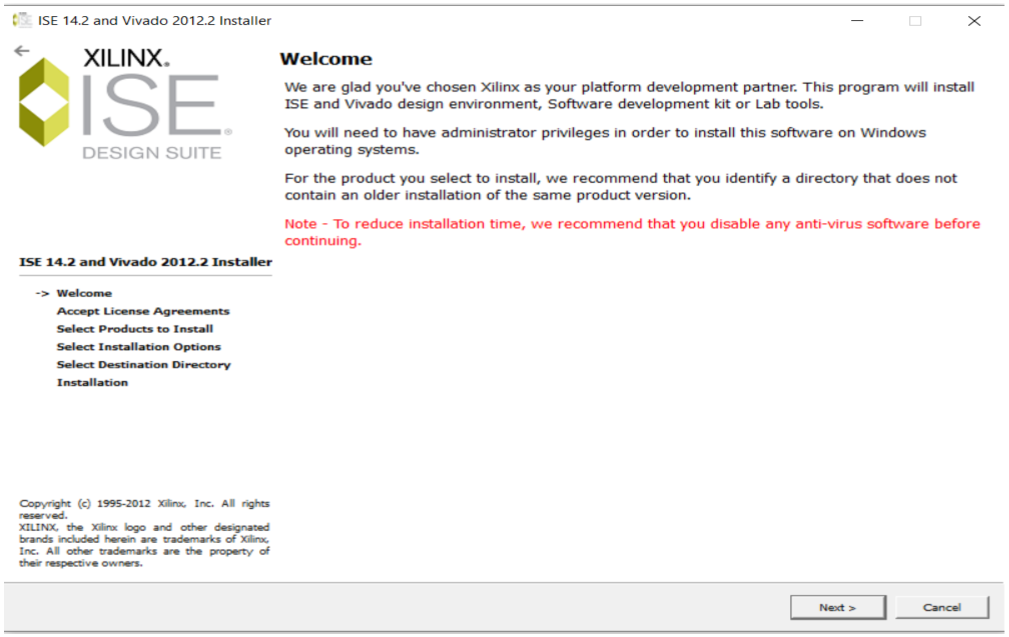
SUBMITTED TO:

- **DR. IRUM MATLOOB**

SUBMITTED BY:

- **AREEJ INTISHAD (2022-BSE-046)**
- **EMAN ZAI (2022-BSE-049)**
- **SYEDA FARWA BATOOL (2022-BSE-071)**

INSTALLATION :



ISE 14.2 and Vivado 2012.2 Installer

Welcome

We are glad you've chosen Xilinx as your platform development partner. This program will install ISE and Vivado design environment, Software development kit or Lab tools.

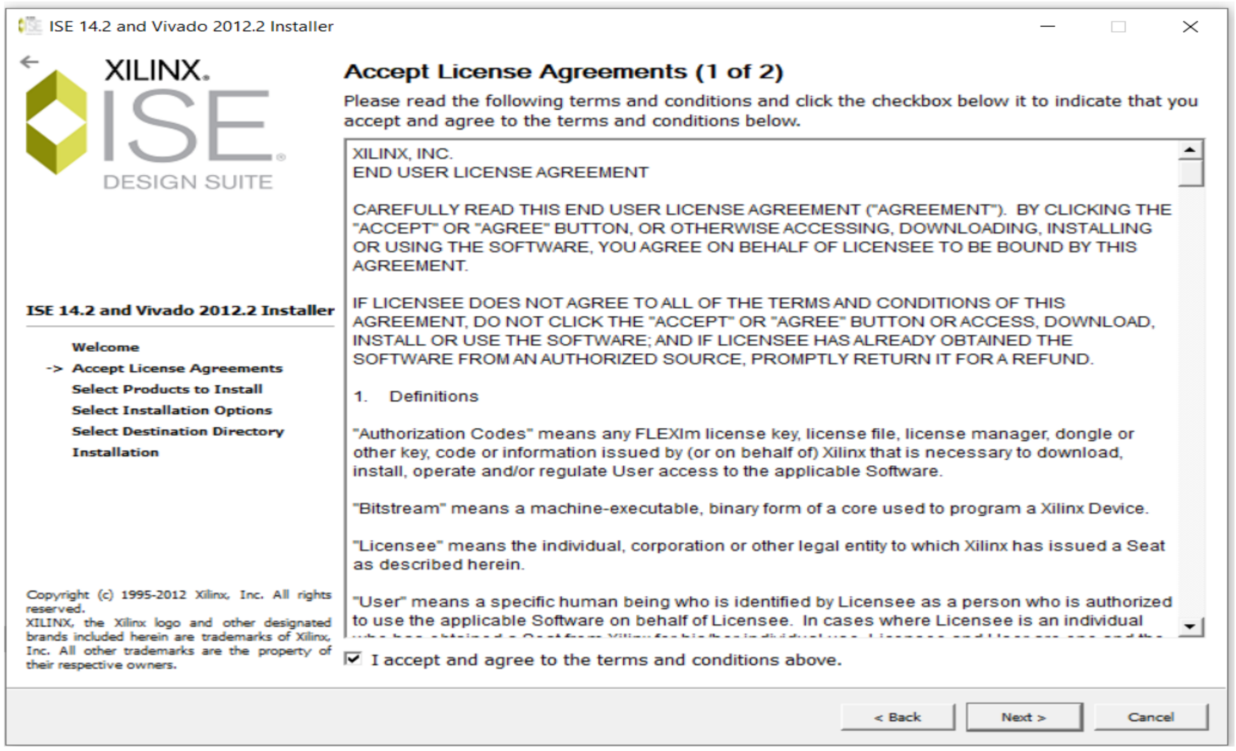
You will need to have administrator privileges in order to install this software on Windows operating systems.

For the product you select to install, we recommend that you identify a directory that does not contain an older installation of the same product version.

Note - To reduce installation time, we recommend that you disable any anti-virus software before continuing.

ISE 14.2 and Vivado 2012.2 Installer

-> Welcome
Accept License Agreements
Select Products to Install
Select Installation Options
Select Destination Directory
Installation



ISE 14.2 and Vivado 2012.2 Installer

Accept License Agreements (1 of 2)

Please read the following terms and conditions and click the checkbox below it to indicate that you accept and agree to the terms and conditions below.

XILINX, INC.
END USER LICENSE AGREEMENT

CAREFULLY READ THIS END USER LICENSE AGREEMENT ("AGREEMENT"). BY CLICKING THE "ACCEPT" OR "AGREE" BUTTON, OR OTHERWISE ACCESSING, DOWNLOADING, INSTALLING OR USING THE SOFTWARE, YOU AGREE ON BEHALF OF LICENSEE TO BE BOUND BY THIS AGREEMENT.

IF LICENSEE DOES NOT AGREE TO ALL OF THE TERMS AND CONDITIONS OF THIS AGREEMENT, DO NOT CLICK THE "ACCEPT" OR "AGREE" BUTTON OR ACCESS, DOWNLOAD, INSTALL OR USE THE SOFTWARE; AND IF LICENSEE HAS ALREADY OBTAINED THE SOFTWARE FROM AN AUTHORIZED SOURCE, PROMPTLY RETURN IT FOR A REFUND.

1. Definitions

"Authorization Codes" means any FLEXlm license key, license file, license manager, dongle or other key, code or information issued by (or on behalf of) Xilinx that is necessary to download, install, operate and/or regulate User access to the applicable Software.

"Bitstream" means a machine-executable, binary form of a core used to program a Xilinx Device.

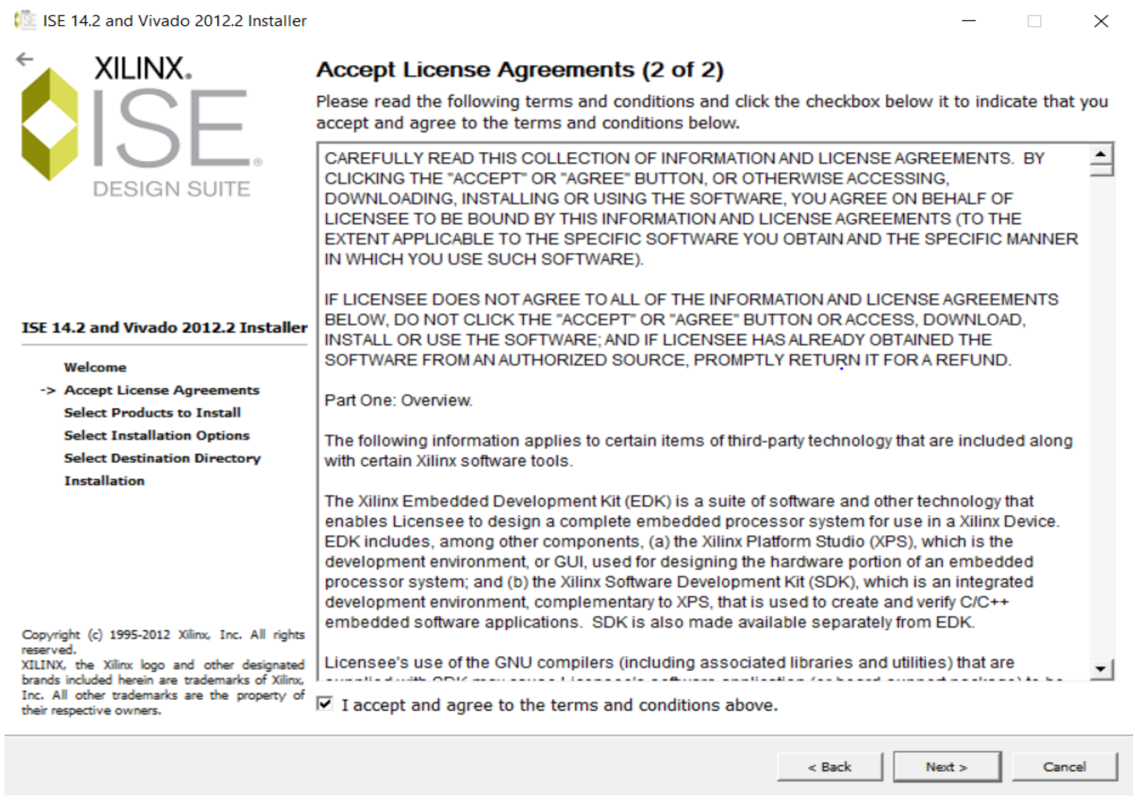
"Licensee" means the individual, corporation or other legal entity to which Xilinx has issued a Seat as described herein.

"User" means a specific human being who is identified by Licensee as a person who is authorized to use the applicable Software on behalf of Licensee. In cases where Licensee is an individual

☑ I accept and agree to the terms and conditions above.

ISE 14.2 and Vivado 2012.2 Installer

Welcome
-> Accept License Agreements
Select Products to Install
Select Installation Options
Select Destination Directory
Installation


ISE 14.2 and Vivado 2012.2 Installer

ISE 14.2 and Vivado 2012.2 Installer

- Welcome
- -> Accept License Agreements
- Select Products to Install
- Select Installation Options
- Select Destination Directory
- Installation

Accept License Agreements (2 of 2)

Please read the following terms and conditions and click the checkbox below it to indicate that you accept and agree to the terms and conditions below.

CAREFULLY READ THIS COLLECTION OF INFORMATION AND LICENSE AGREEMENTS. BY CLICKING THE "ACCEPT" OR "AGREE" BUTTON, OR OTHERWISE ACCESSING, DOWNLOADING, INSTALLING OR USING THE SOFTWARE, YOU AGREE ON BEHALF OF LICENSEE TO BE BOUND BY THIS INFORMATION AND LICENSE AGREEMENTS (TO THE EXTENT APPLICABLE TO THE SPECIFIC SOFTWARE YOU OBTAIN AND THE SPECIFIC MANNER IN WHICH YOU USE SUCH SOFTWARE).

IF LICENSEE DOES NOT AGREE TO ALL OF THE INFORMATION AND LICENSE AGREEMENTS BELOW, DO NOT CLICK THE "ACCEPT" OR "AGREE" BUTTON OR ACCESS, DOWNLOAD, INSTALL OR USE THE SOFTWARE; AND IF LICENSEE HAS ALREADY OBTAINED THE SOFTWARE FROM AN AUTHORIZED SOURCE, PROMPTLY RETURN IT FOR A REFUND.

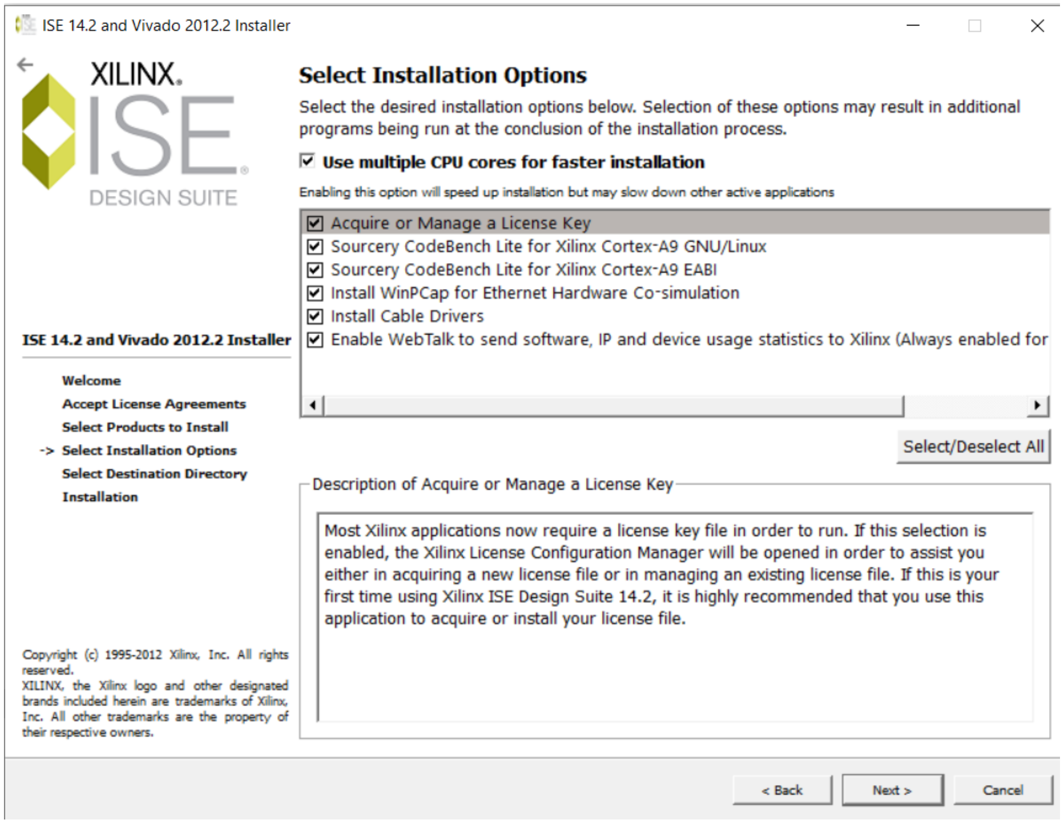
Part One: Overview.

The following information applies to certain items of third-party technology that are included along with certain Xilinx software tools.

The Xilinx Embedded Development Kit (EDK) is a suite of software and other technology that enables Licensee to design a complete embedded processor system for use in a Xilinx Device. EDK includes, among other components, (a) the Xilinx Platform Studio (XPS), which is the development environment, or GUI, used for designing the hardware portion of an embedded processor system; and (b) the Xilinx Software Development Kit (SDK), which is an integrated development environment, complementary to XPS, that is used to create and verify C/C++ embedded software applications. SDK is also made available separately from EDK.

Licensee's use of the GNU compilers (including associated libraries and utilities) that are

☑ I accept and agree to the terms and conditions above.


ISE 14.2 and Vivado 2012.2 Installer

ISE 14.2 and Vivado 2012.2 Installer

- Welcome
- Accept License Agreements
- Select Products to Install
- -> Select Installation Options
- Select Destination Directory
- Installation

Select Installation Options

Select the desired installation options below. Selection of these options may result in additional programs being run at the conclusion of the installation process.

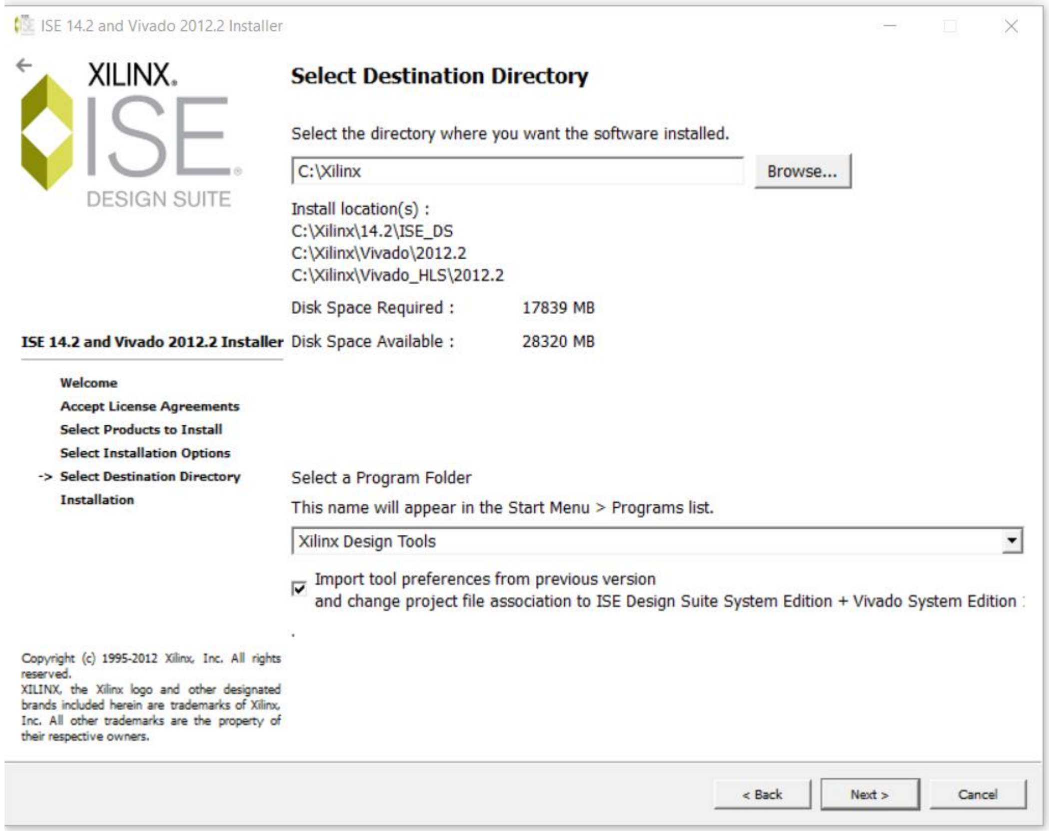
☑ **Use multiple CPU cores for faster installation**

Enabling this option will speed up installation but may slow down other active applications

- ☑ Acquire or Manage a License Key
- ☑ Sourcery CodeBench Lite for Xilinx Cortex-A9 GNU/Linux
- ☑ Sourcery CodeBench Lite for Xilinx Cortex-A9 EABI
- ☑ Install WinPCap for Ethernet Hardware Co-simulation
- ☑ Install Cable Drivers
- ☑ Enable WebTalk to send software, IP and device usage statistics to Xilinx (Always enabled for

Description of Acquire or Manage a License Key

Most Xilinx applications now require a license key file in order to run. If this selection is enabled, the Xilinx License Configuration Manager will be opened in order to assist you either in acquiring a new license file or in managing an existing license file. If this is your first time using Xilinx ISE Design Suite 14.2, it is highly recommended that you use this application to acquire or install your license file.


ISE 14.2 and Vivado 2012.2 Installer

Select Destination Directory

Select the directory where you want the software installed.

C:\Xilinx [Browse...]

Install location(s) :
C:\Xilinx\14.2\ISE_DS
C:\Xilinx\Vivado\2012.2
C:\Xilinx\Vivado_HLS\2012.2

Disk Space Required : 17839 MB

Disk Space Available : 28320 MB

ISE 14.2 and Vivado 2012.2 Installer

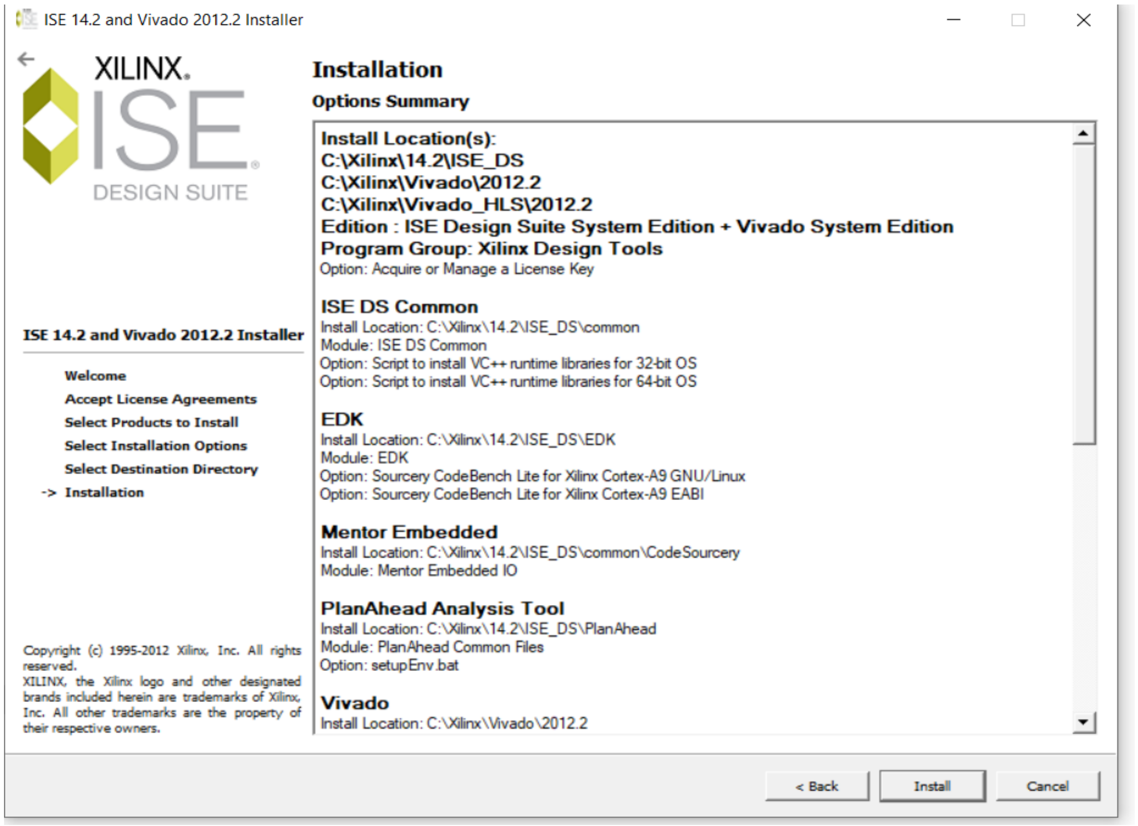
- Welcome
- Accept License Agreements
- Select Products to Install
- Select Installation Options
- -> Select Destination Directory
- Installation

Select a Program Folder

This name will appear in the Start Menu > Programs list.

Xilinx Design Tools

[✓] Import tool preferences from previous version and change project file association to ISE Design Suite System Edition + Vivado System Edition :


ISE 14.2 and Vivado 2012.2 Installer

Installation

Options Summary

Install Location(s):
C:\Xilinx\14.2\ISE_DS
C:\Xilinx\Vivado\2012.2
C:\Xilinx\Vivado_HLS\2012.2
Edition : ISE Design Suite System Edition + Vivado System Edition
Program Group: Xilinx Design Tools
Option: Acquire or Manage a License Key

ISE DS Common
Install Location: C:\Xilinx\14.2\ISE_DS\common
Module: ISE DS Common
Option: Script to install VC++ runtime libraries for 32-bit OS
Option: Script to install VC++ runtime libraries for 64-bit OS

EDK
Install Location: C:\Xilinx\14.2\ISE_DS\EDK
Module: EDK
Option: Sourcery CodeBench Lite for Xilinx Cortex-A9 GNU/Linux
Option: Sourcery CodeBench Lite for Xilinx Cortex-A9 EABI

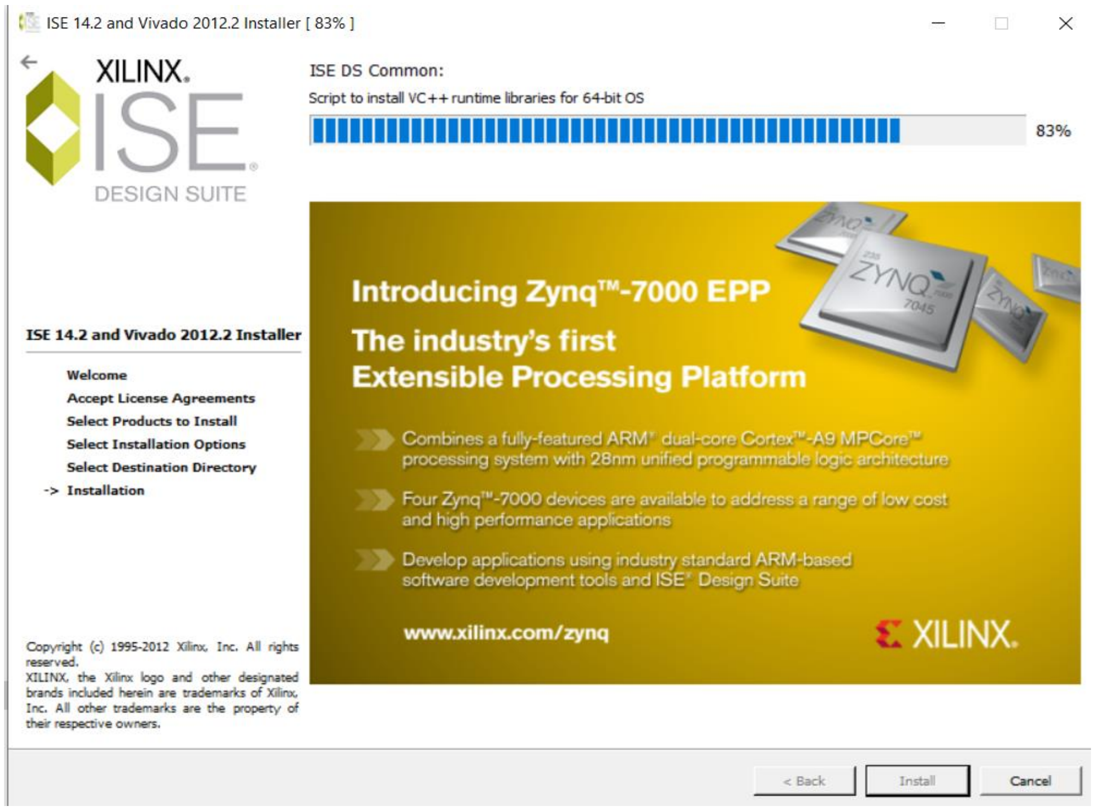
Mentor Embedded
Install Location: C:\Xilinx\14.2\ISE_DS\common\CodeSourcery
Module: Mentor Embedded IO

PlanAhead Analysis Tool
Install Location: C:\Xilinx\14.2\ISE_DS\PlanAhead
Module: PlanAhead Common Files
Option: setupEnv.bat

Vivado
Install Location: C:\Xilinx\Vivado\2012.2

ISE 14.2 and Vivado 2012.2 Installer

- Welcome
- Accept License Agreements
- Select Products to Install
- Select Installation Options
- Select Destination Directory
- -> Installation


ISE 14.2 and Vivado 2012.2 Installer [83%]
XILINX. ISE DESIGN SUITE
ISE 14.2 and Vivado 2012.2 Installer
Welcome
Accept License Agreements
Select Products to Install
Select Installation Options
Select Destination Directory
-> Installation
Introducing Zynq™-7000 EPP
The industry's first Extensible Processing Platform
www.xilinx.com/zynq
< Back
Install
Cancel

ISE DS Common:

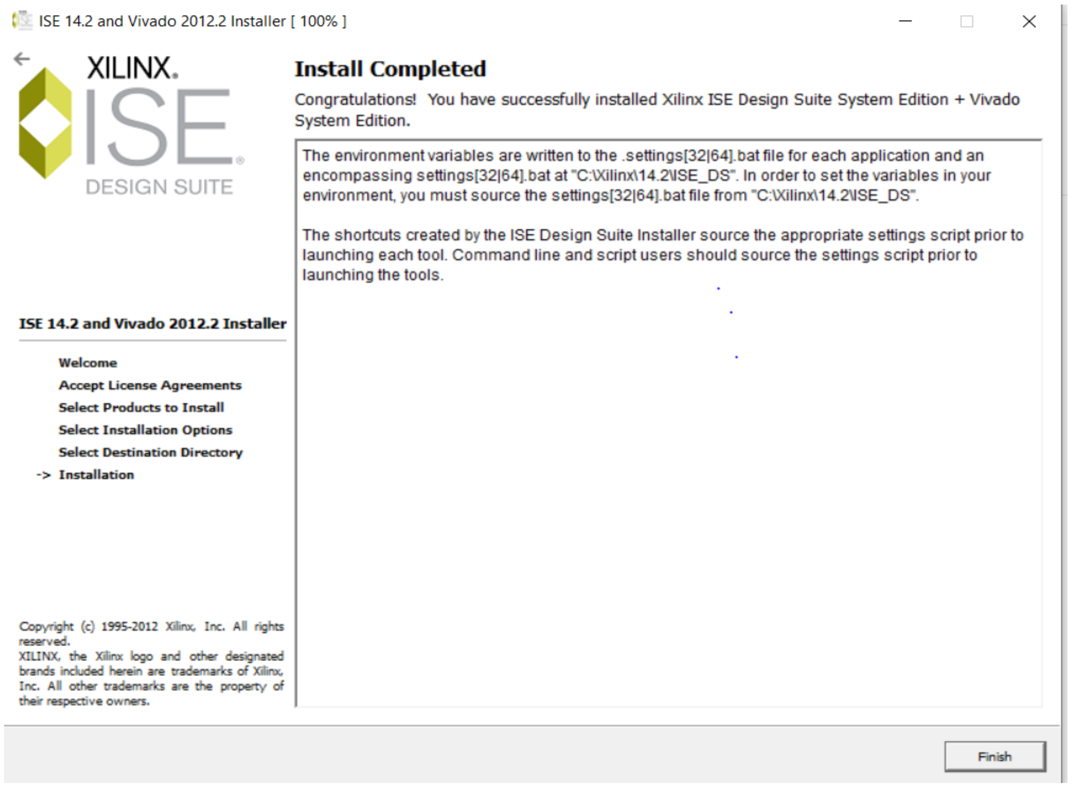
Script to install VC++ runtime libraries for 64-bit OS

83%

Combines a fully-featured ARM® dual-core Cortex™-A9 MPCore™ processing system with 28nm unified programmable logic architecture

Four Zynq™-7000 devices are available to address a range of low cost and high performance applications

Develop applications using industry standard ARM-based software development tools and ISE® Design Suite

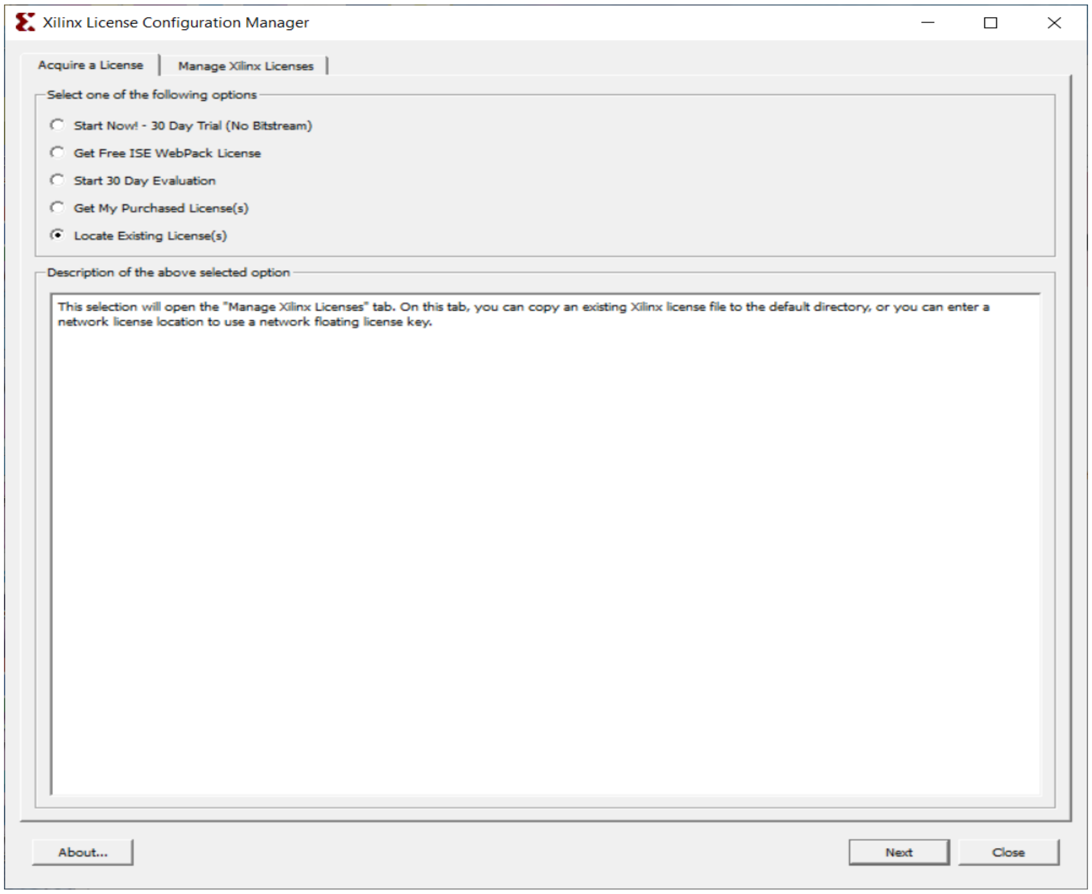

ISE 14.2 and Vivado 2012.2 Installer [100%]
XILINX. ISE DESIGN SUITE
ISE 14.2 and Vivado 2012.2 Installer
Welcome
Accept License Agreements
Select Products to Install
Select Installation Options
Select Destination Directory
-> Installation
Finish

Install Completed

Congratulations! You have successfully installed Xilinx ISE Design Suite System Edition + Vivado System Edition.

The environment variables are written to the .settings[32|64].bat file for each application and an encompassing settings[32|64].bat at "C:\Xilinx\14.2\ISE_DS". In order to set the variables in your environment, you must source the settings[32|64].bat file from "C:\Xilinx\14.2\ISE_DS".

The shortcuts created by the ISE Design Suite Installer source the appropriate settings script prior to launching each tool. Command line and script users should source the settings script prior to launching the tools.



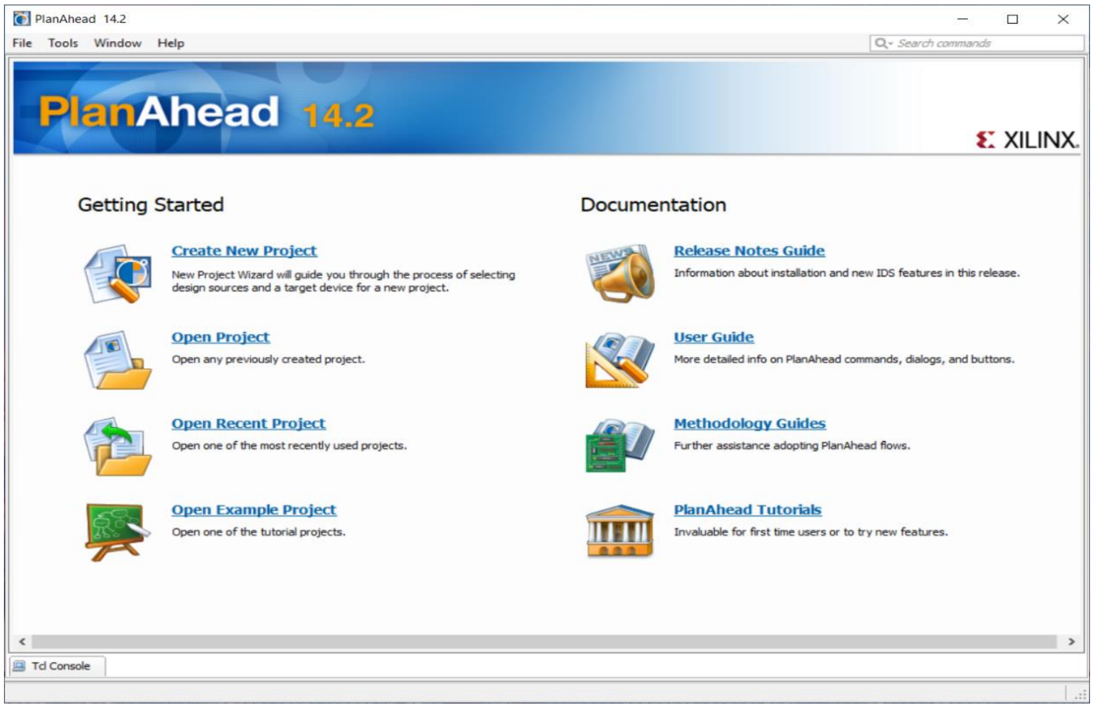
Xilinx License Configuration Manager

Select one of the following options

- Start Now! - 30 Day Trial (No Bitstream)
- Get Free ISE WebPack License
- Start 30 Day Evaluation
- Get My Purchased License(s)
- [✓ Locate Existing License(s)]

Description of the above selected option

This selection will open the "Manage Xilinx Licenses" tab. On this tab, you can copy an existing Xilinx license file to the default directory, or you can enter a network license location to use a network floating license key.



PlanAhead 14.2

Getting Started

Create New Project
New Project Wizard will guide you through the process of selecting design sources and a target device for a new project.

Open Project
Open any previously created project.

Open Recent Project
Open one of the most recently used projects.

Open Example Project
Open one of the tutorial projects.

Documentation

Release Notes Guide
Information about installation and new IDS features in this release.

User Guide
More detailed info on PlanAhead commands, dialogs, and buttons.

Methodology Guides
Further assistance adopting PlanAhead flows.

PlanAhead Tutorials
Invaluable for first time users or to try new features.

CONTENTS:

ARITHMETIC AND LOGIC UNIT

Arithmetic **A**nd **L**ogic **U**nit is a part of the central processing unit in which **all arithmetic and logical operations are being performed** on the **operands** in the **instruction memory.**

SCHEMATIC DIAGRAM:

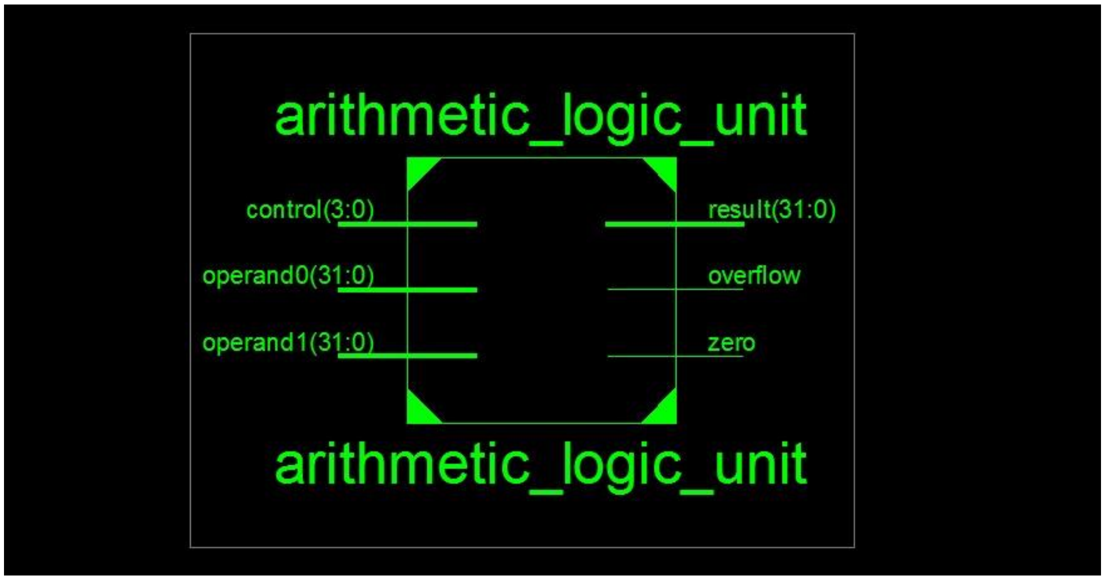


Figure 1.1

CODE EXPLANATION:

The **main module** contains **control, two operands and a result with overflow and zero flag**.

Flag overflow is generally needed for unsigned numbers. When some operation is being performed on unsigned numbers, they are positive but their sum could negative, here is where overflow occurs.

A **flag** generally indicates that a certain condition has been met. It is generally declared with data type **bool.**

1. The **input port** contains **control, operand 0 and operand 1**. These 2 operands and control is given by **user.**
2. **Control is of 4 bits** which will determine which operation will be performed on the instruction to get the result bit, operand 0 and operand 1 is of 32 bits individually. **3. Output** contains **result, zero flag** and **overflow.**

The **bidirectional port** is dependent on an input or an output. It is beneficial when the microprocessor receives information and sends it to an input/output device which is not used here. Then the design begins. In signal declaration local parameters are used. The signal generally verifies whether the input provided or the output value is in the given range or not.

3 registers are being used for **result, overflow and for zero**. The signal declaration wire is used for internal connections of input and output signals. In this program wire is not being initialized.

Then the **combinational logic** starts along with the signals.

There are different cases used for different arithmetic operations which have been discussed below.

The **operand0**, **operand1** and **control** will be taken from the user. Overflow will be generally initialized with 0 values and if the program contains any overflow it will be shown in output.

2 operands are given as input by right clicking the operand 0 and 1 by selecting the **force value** and then by adding the addresses which are given in Verilog MIPS file.

Control is given in the same way.

CASE SCENARIOS

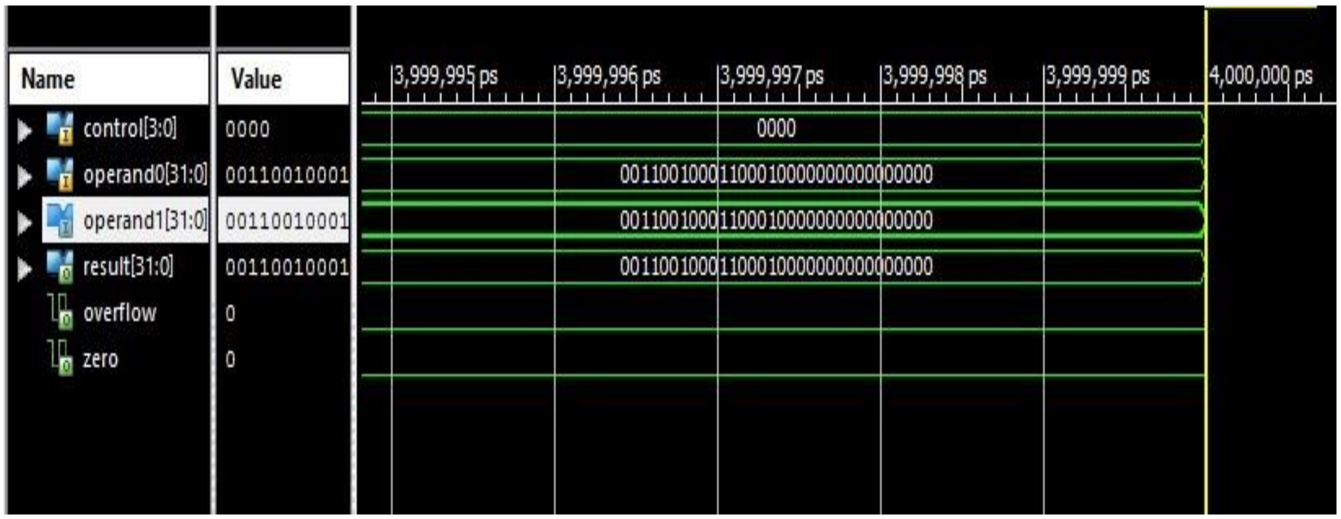
AND CASE:

Initially, **Control** is given by the user as **0000.**

Along with it, **2 operands** are provided by the user in hexadecimal form as input.

The software will convert these operands in **binary form** and after that will perform operation on them i.e. it will **compare each bit of two operands,** if the bits of both operands is 1 than result will be 1 else result will be zero (Implementing the AND logic) .

The **result** of AND will be displayed in output portion and if there will be no overflow then **overflow and flag zero will remain zero.**

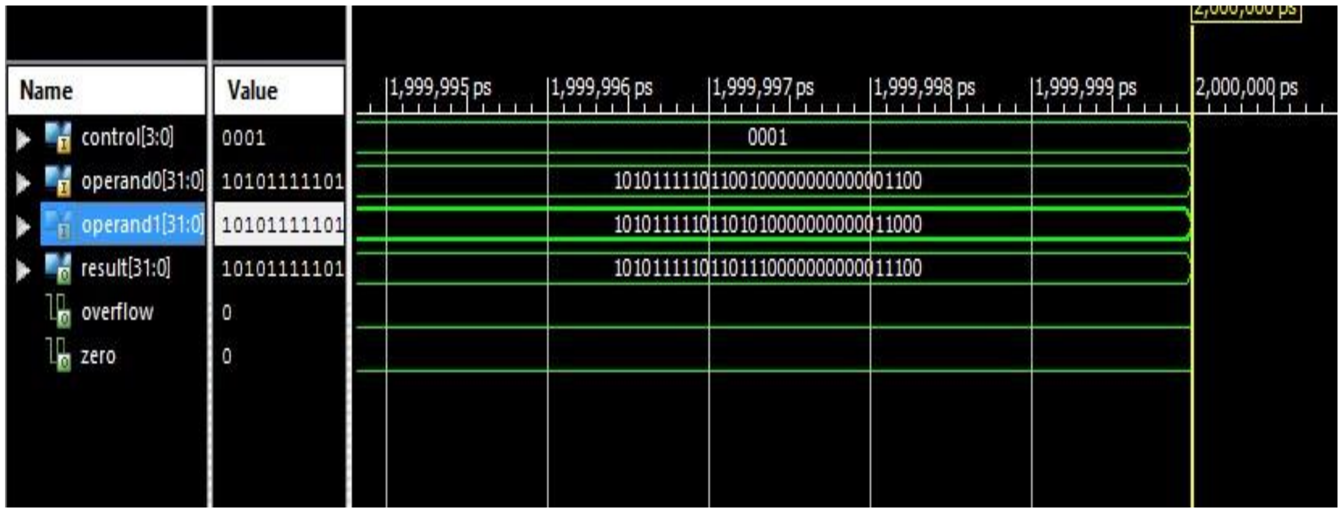


```
Name              Value          3,999,995 ps  3,999,996 ps  3,999,997 ps  3,999,998 ps  3,999,999 ps  4,000,000 ps
control[3:0]      0000           0000
operand0[31:0]    00110010001    00110010001100010000000000000000
operand1[31:0]    00110010001    00110010001100010000000000000000
result[31:0]      00110010001    00110010001100010000000000000000
overflow          0
zero              0
```

Figure 1.2

OR CASE:

When the user enters **0001** as control along with 2 operands then OR operation will be performed i.e**. it will compare each bit of two operands, if the bits of both operands is 0 than result will be 0 else result will be 1.**

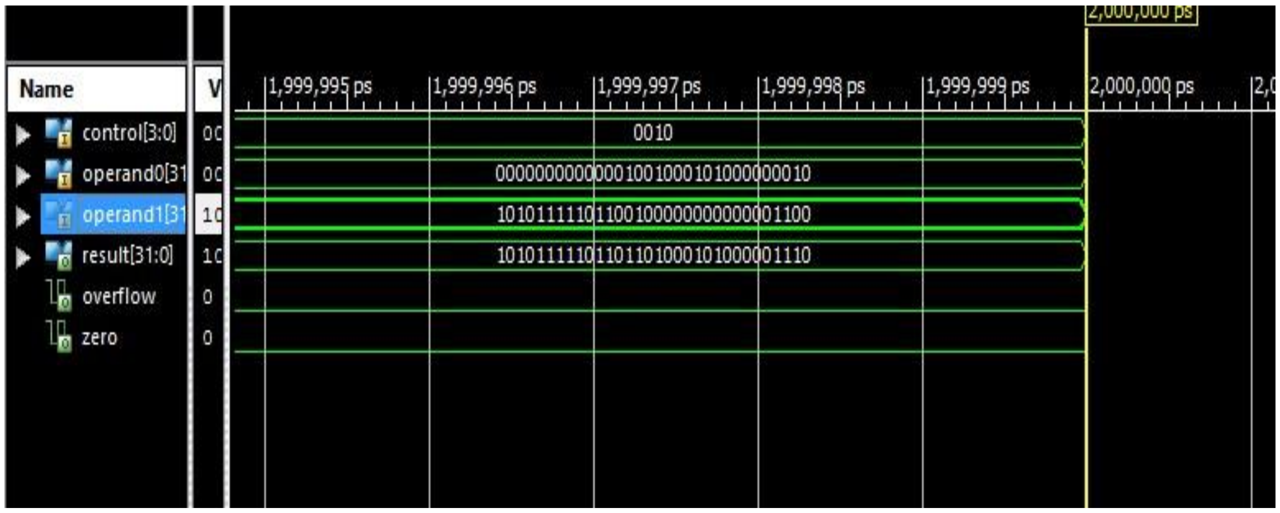


```
Name              Value          1,999,995 ps  1,999,996 ps  1,999,997 ps  1,999,998 ps  1,999,999 ps  2,000,000 ps
control[3:0]      0001           0001
operand0[31:0]    10101111101    10101111101100100000000000001100
operand1[31:0]    10101111101    10101111101101010000000000011000
result[31:0]      10101111101    10101111101101110000000000011100
overflow          0
zero              0
```

Figure 1.3

ADD CASE:

When the user enters **0010** as control then **ADD** operation will be performed on the operands. It will add the both operands. If the bit of both operands is **0** then the result bit will be **0, otherwise** result bit will be **1** but if the bit of both operands are 1, then the result bit will become 1 and 1 that will be taken as a **carry** for the next bit. This will form a new number that is the result of addition of the two operands.

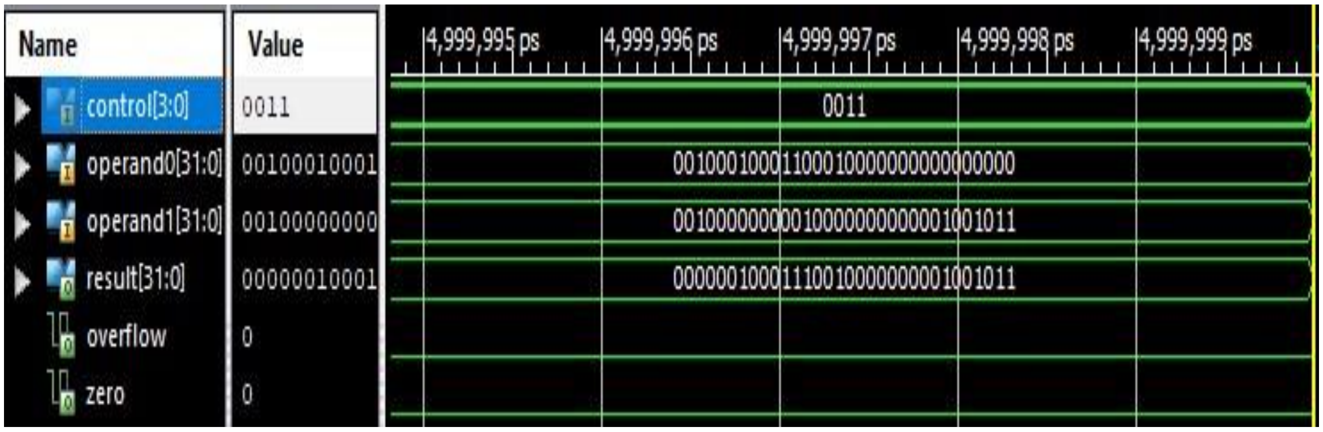


```
2,000,000 ps
Name            V   |1,999,995 ps |1,999,996 ps |1,999,997 ps |1,999,998 ps |1,999,999 ps |2,000,000 ps |2,0
control[3:0]    00  0010
operand0[31     00  00000000000001001000101000000010
operand1[31     10  10101111101100100000000000001100
result[31:0]    10  10101111101101101000101000001110
overflow        0
zero            0
```

Figure 1.4

XOR CASE:

If the control bit is **0011,** XOR operation will be performed on the operands. It will compare each bit of the 2 operands. If the bits of both operands are **opposite** then the result bit will be **1** and if they are **same** then resultant bit will be **0.**



```
Name            Value        |4,999,995 ps |4,999,996 ps |4,999,997 ps |4,999,998 ps |4,999,999 ps
control[3:0]    0011         0011
operand0[31:0]  00100010001  00100010001100010000000000000000
operand1[31:0]  00100000000  00100000000010000000000001001011
result[31:0]    0000010001   00000010001110010000000001001011
overflow        0
zero            0
```

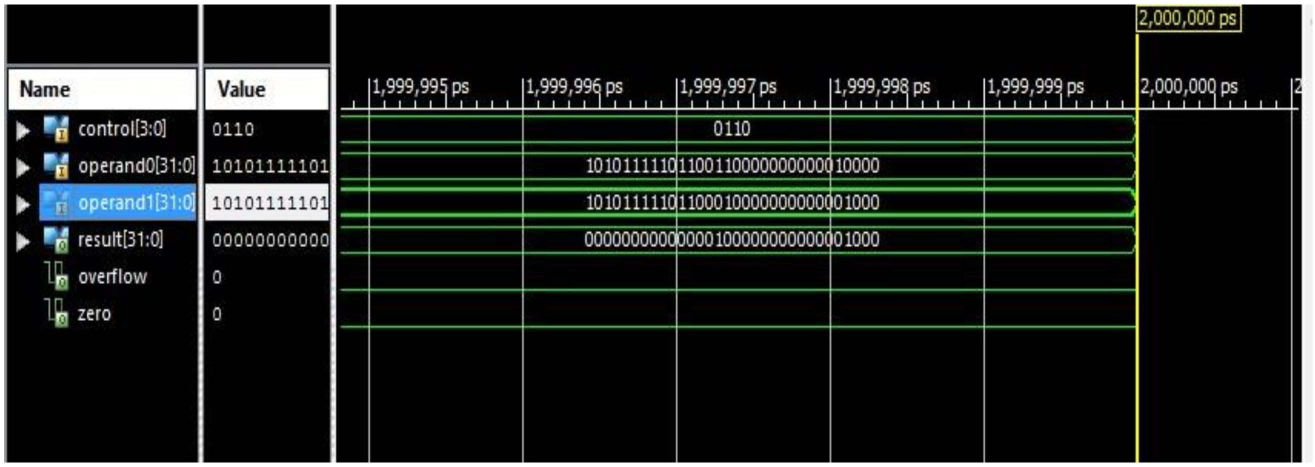
Figure 1.5

NOR CASE:

If the control bit is **001,** NOR operation will be performed on the operands. It will compare each bit of the 2 operands. If the bits of both operands are 0 then the result bit will be 1 otherwise result bit will be 0.

SUB CASE:

Control is given by the user as **0110** and subtraction is performed on 2 operands.

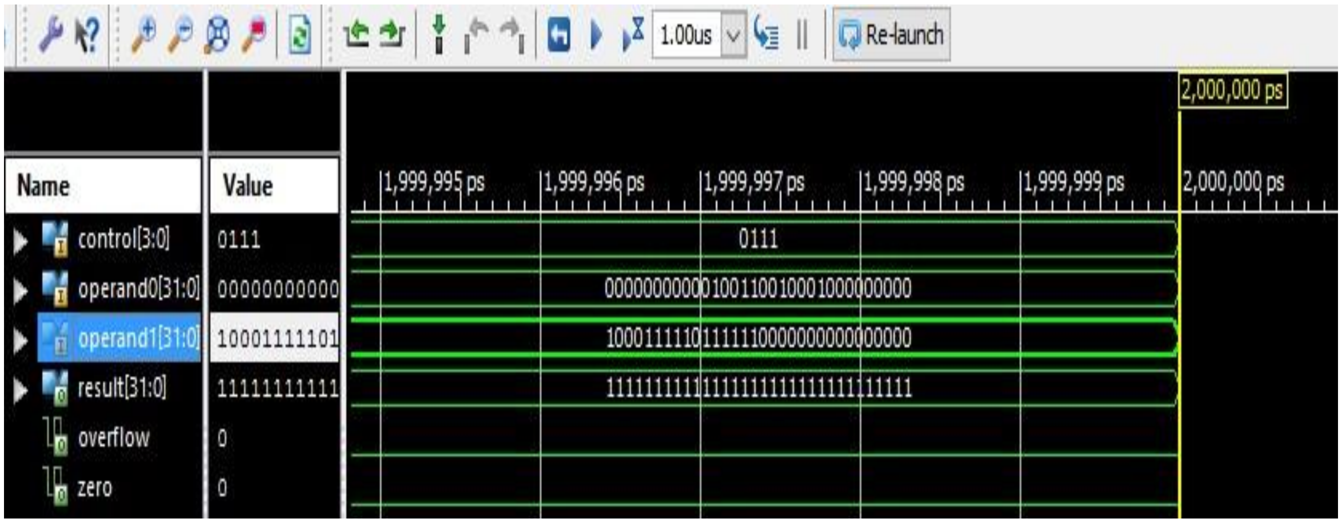


```
Name            Value         2,000,000 ps
control[3:0]    0110          0110
operand0[31:0]  10101111101   10101111101100110000000000010000
operand1[31:0]  10101111101   10101111101100010000000000001000
result[31:0]    00000000000   00000000000000010000000000001000
overflow        0
zero            0
```

Figure 1.6

SET ON LESS THAN CASE:

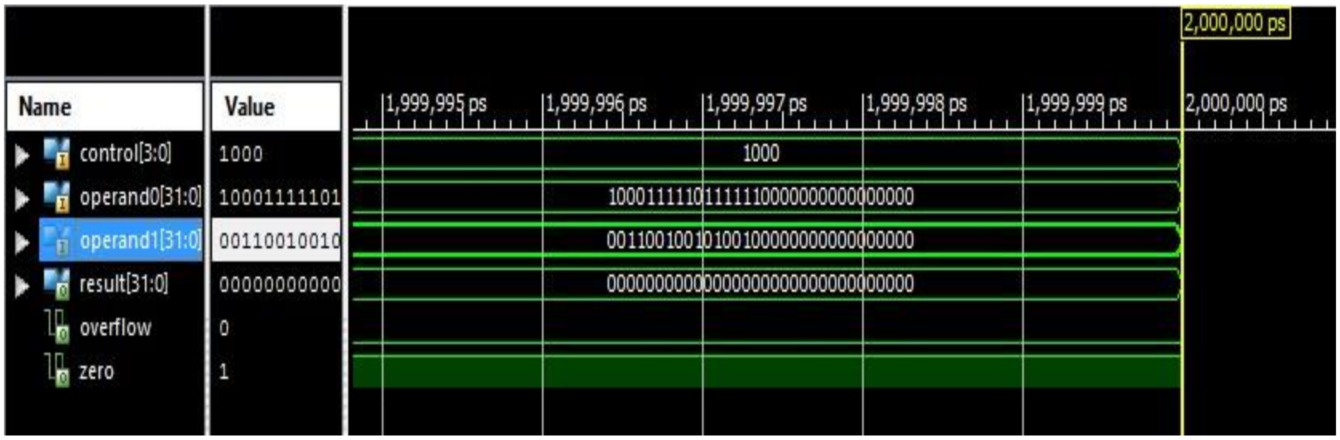
Control is given by the user as **0111.**



```
1.00us   Re-launch
Name            Value         2,000,000 ps
control[3:0]    0111          0111
operand0[31:0]  00000000000   00000000000100110010001000000000
operand1[31:0]  10001111101   10001111101111110000000000000000
result[31:0]    11111111111   11111111111111111111111111111111
overflow        0
zero            0
```

SHIFT LEFT CASE:

When the user enters **1000** as control then shift left operation will be performed.

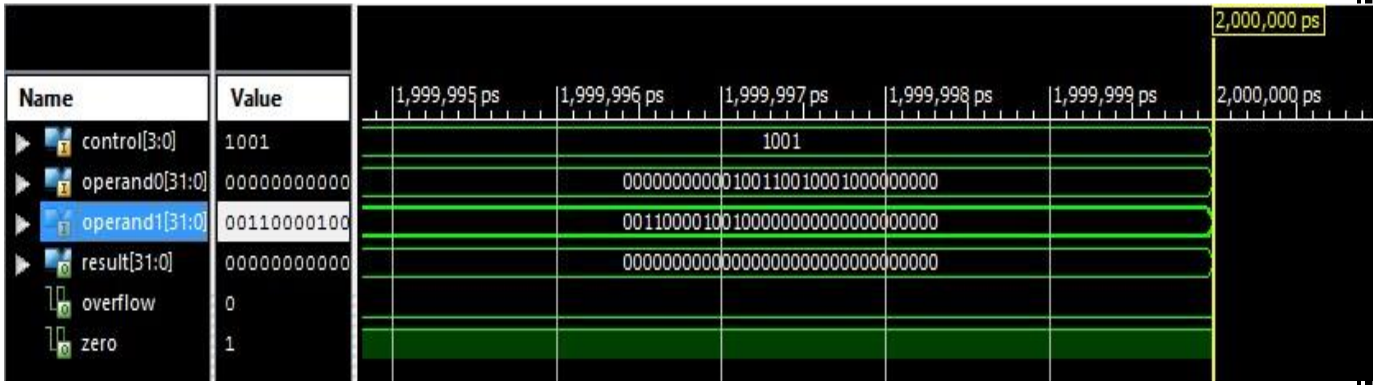


```
Name            Value         2,000,000 ps
control[3:0]    1000          1000
operand0[31:0]  10001111101   10001111101111110000000000000000
operand1[31:0]  00110010010   00110010010100100000000000000000
result[31:0]    00000000000   00000000000000000000000000000000
overflow        0
zero            1
```

Figure 1.7

SHIFT RIGHT CASE:

When the user enters **1001** as control then shift right operation will be performed.

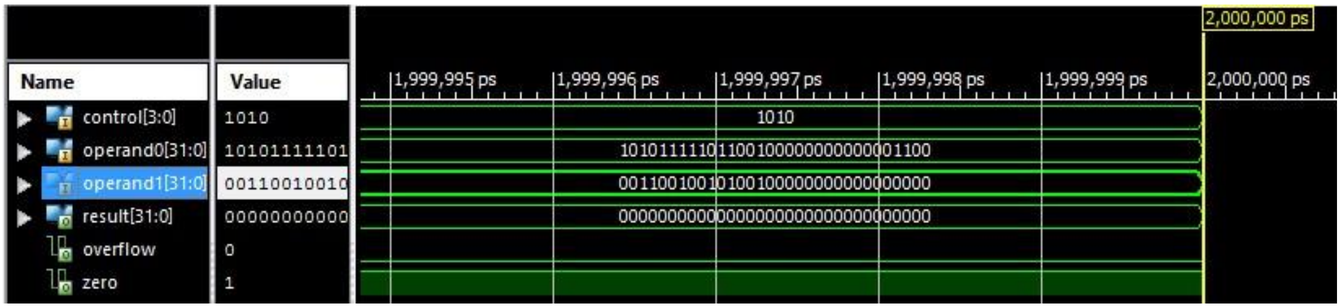


```
Name              Value         2,000,000 ps
                                |1,999,995 ps  |1,999,996 ps  |1,999,997 ps  |1,999,998 ps  |1,999,999 ps  |2,000,000 ps
control[3:0]      1001          1001
operand0[31:0]    00000000000   00000000000100110010001000000000
operand1[31:0]    00110000100   00110000100100000000000000000000
result[31:0]      00000000000   00000000000000000000000000000000
overflow          0
zero              1
```

Figure 1.8

SHIFT RIGHT ARITHMETIC CASE:

When the user enters 1010 as control then shift right arithmetic operation will be performed.

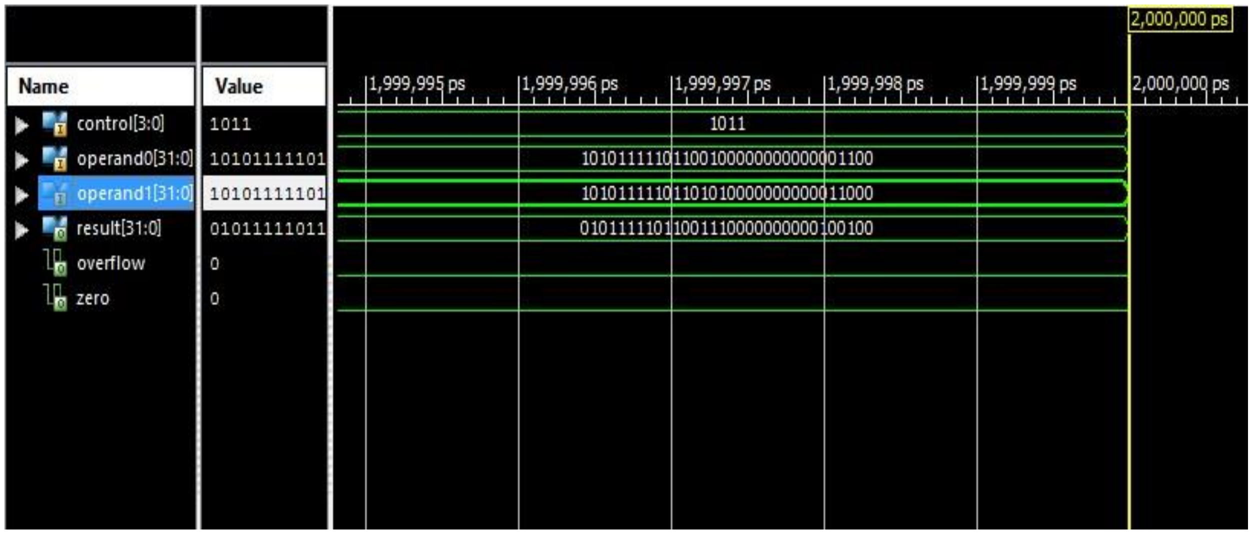


```
Name              Value         2,000,000 ps
                                |1,999,995 ps  |1,999,996 ps  |1,999,997 ps  |1,999,998 ps  |1,999,999 ps  |2,000,000 ps
control[3:0]      1010          1010
operand0[31:0]    10101111101   10101111101100100000000000001100
operand1[31:0]    00110010010   00110010010100100000000000000000
result[31:0]      00000000000   00000000000000000000000000000000
overflow          0
zero              1
```

Figure 1.9

SIGNED ADD CASE:

When the user enters **1011** as control then signed add operation will be performed.

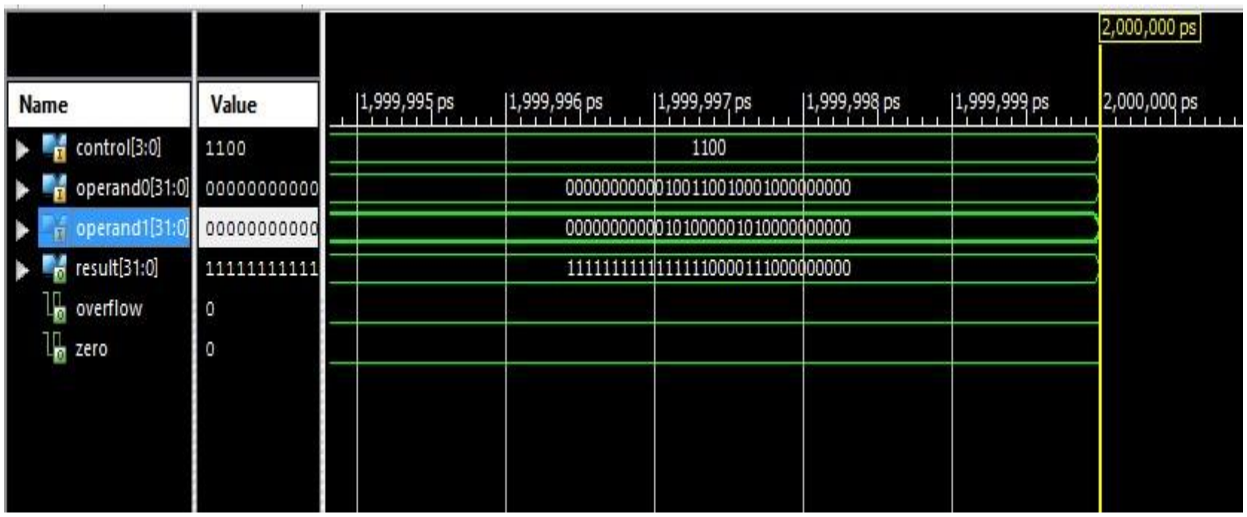


```
Name              Value         2,000,000 ps
                                |1,999,995 ps  |1,999,996 ps  |1,999,997 ps  |1,999,998 ps  |1,999,999 ps  |2,000,000 ps
control[3:0]      1011          1011
operand0[31:0]    10101111101   10101111101100100000000000001100
operand1[31:0]    10101111101   10101111101101010000000000011000
result[31:0]      01011111011   01011111011001110000000000100100
overflow          0
zero              0
```

Figure 1.10

SIGNED SUB CASE:

When the user enters **1100** as control then signed sub operation will be performed.



```
control[3:0]      1100          1100
operand0[31:0]    00000000000   00000000000100110010001000000000
operand1[31:0]    00000000000   00000000000101000001010000000000
result[31:0]      11111111111   11111111111111110000111000000000
overflow          0
zero              0
```

Figure 1.11

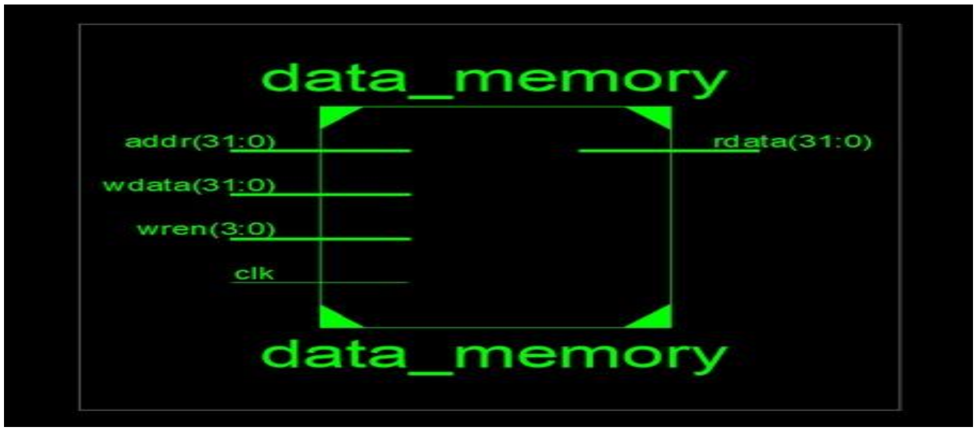
DATA MEMORY

Data memory module store frames which are managed by **DATA MEMORY MANAGEMENT UNITS.**

A frame is a logical unit of transfer between main memory and secondary storage device.
Data memory in RAM is used for storing and keeping data required for proper operation of the program.

SCHEMATIC DIAGRAM:

Figure 2.1



CODE EXPLANATION:

- The **main module** contains **clock, read data, write data, addr and write valid**.
- **Wren** shows **memory write** or by using which signal it would be written.
- **Address** is used to **handle** the **address transferred** to the memory module.
- **Four inputs** will be taken from the user **clk, addr, wdata and wren**.
- **Clock enable signal** enables the memory only when the memory is being used and for the rest of the time it shuts it down which reduces the overall memory consumption power.
- So, the **user** will **decide** when the clock should be **ON** and when it should be **OFF.**
- The **wdata** contains the **data** that is to be **written in the memory**.
- Address contains the address of the data that is to be written and wren specifies the signals.
- The input wdata and address will be of 32 bits and wren will be of 4 bits.
- The **output** contains **rdata** of **32bits** which represents the data to be read. There are no design or bidirectional ports.
- Signal declaration contains one register of **8 bits** which will contain memory lanes and **memory lane is capable of transferring 65536 bits of data**. **4 lanes** are available for the data. The **lane** will be **selected** according to the **value given by user**. No wire is initialized in the program.
- **In combinational logic** all of the 4 lanes are assigned to rdata. Which means any of the lane containing data for transformation will be taken by rdata. These lanes transfer the data of address given by user which is of 18 bits. No sequential logic is for the program.

  When the clock enables the signal the program will check the value of wren.
- When **0000** is given as an input this means that **no data is read.**
- When **0001** is given as an input this reads **last 8 bits of wdata** [31:0].
- When **0011** is given as an input this reads last **16 bits of wdata** [31:0].
- When **0111 is** given as an input this reads last **24 bits of wdata** [31:0].
- When **1111** is given as an input this reads all **32 bits of wdata** [31:0].

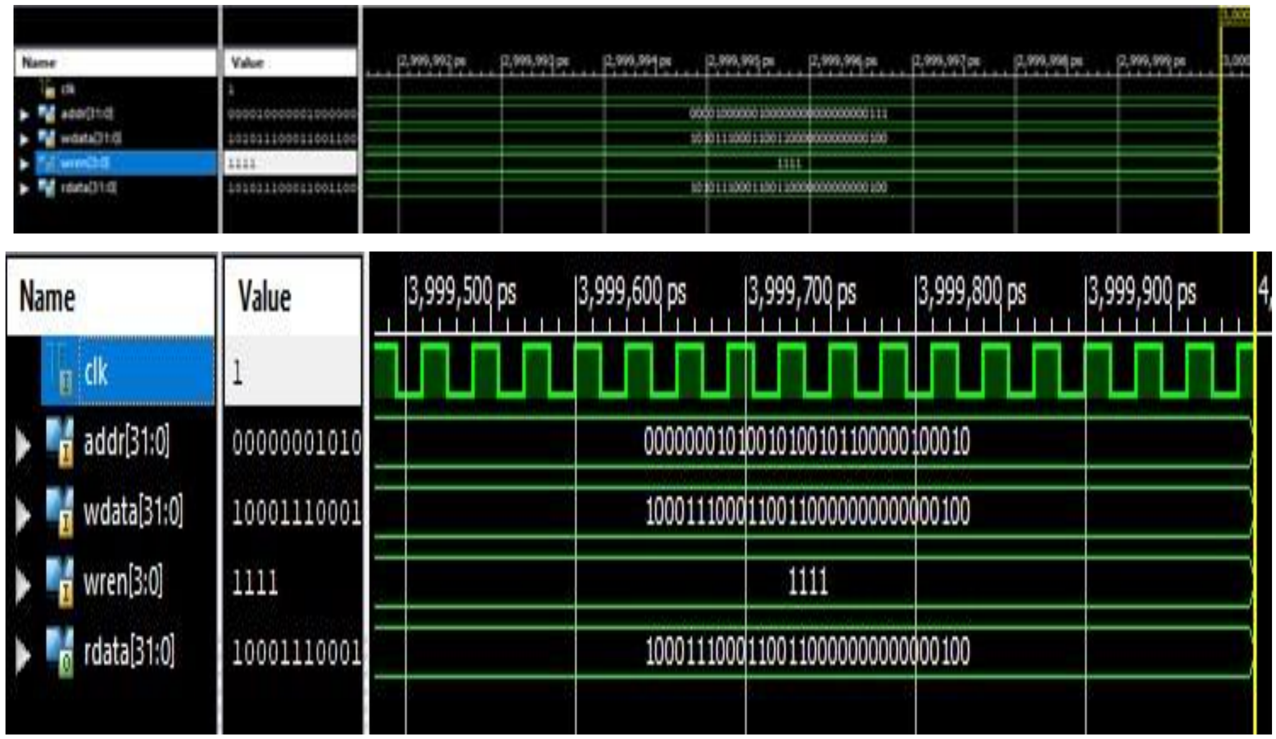


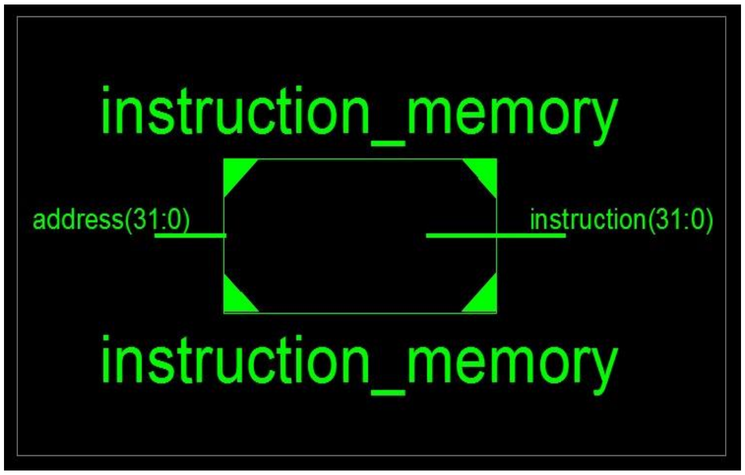
Figure 2.2

INSTRUCTION MEMORY

The instruction memory stores all the pre-fetched instructions. It has five major components PC unit, PC Decoder, Input buffer, instruction memory and output buffer.

SCHEMATIC DIAGRAM:

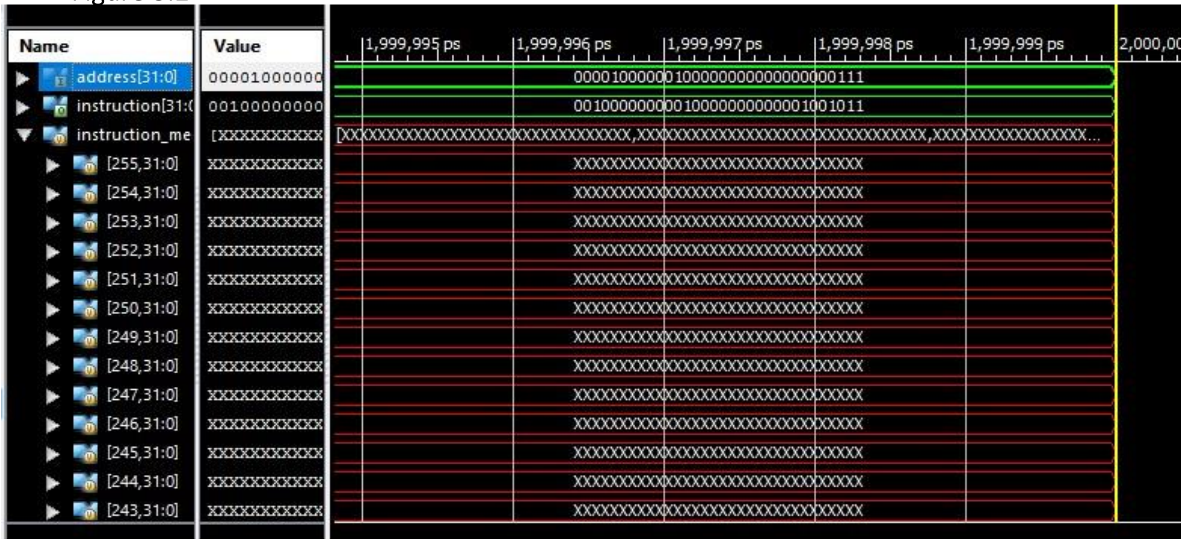
Figure 3.1



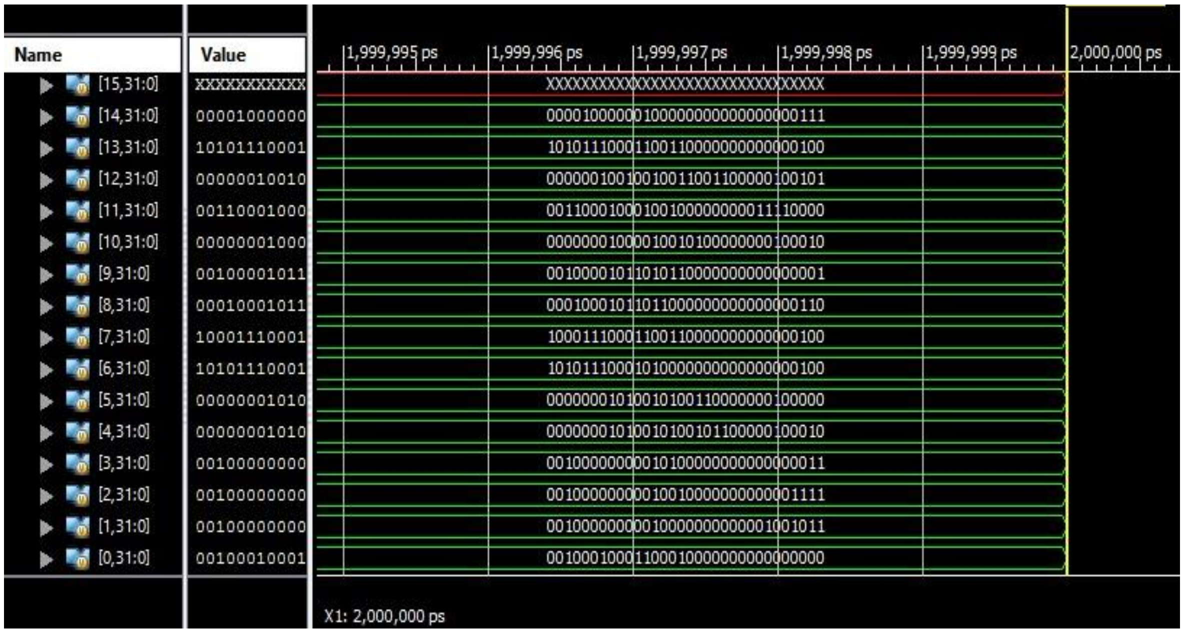
CODE EXPLANATION:

- The **main module** contains **instruction and address.**
- The **input port** contains the **address** having **32 bits**. The output port contains the instruction of 32 bits. It means the address is entered by the user and the instruction on that address will be fetched.
- There is **no design or bi-directional ports** used here. Signal contains register of 32 bits having instructions from 0-255.
- The **combinational logic** begins with the function which reads a file “program.mips” from memory and this file contains the addresses. It will assign instructions one by one on the given address.

Figure 3.2



```
Name                 Value          |1,999,995 ps |1,999,996 ps |1,999,997 ps |1,999,998 ps |1,999,999 ps |2,000,00
address[31:0]        00001000000    00001000000100000000000000000111
instruction[31:(     00100000000    00100000000100000000000001001011
instruction_me       [XXXXXXXXXXX   [XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX,XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX,XXXXXXXXXXXXXXXXXX...
  [255,31:0]         XXXXXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  [254,31:0]         XXXXXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  [253,31:0]         XXXXXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  [252,31:0]         XXXXXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  [251,31:0]         XXXXXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  [250,31:0]         XXXXXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  [249,31:0]         XXXXXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  [248,31:0]         XXXXXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  [247,31:0]         XXXXXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  [246,31:0]         XXXXXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  [245,31:0]         XXXXXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  [244,31:0]         XXXXXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  [243,31:0]         XXXXXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```



```
Name         Value          |1,999,995 ps |1,999,996 ps |1,999,997 ps |1,999,998 ps |1,999,999 ps |2,000,000 ps
  [15,31:0]  XXXXXXXXXXX    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  [14,31:0]  00001000000    00001000000100000000000000000111
  [13,31:0]  10101110001    10101110001100110000000000000100
  [12,31:0]  00000010010    00000010010010011001100000100101
  [11,31:0]  00110001000    00110001000100100000000011110000
  [10,31:0]  00000001000    00000001000010010100000000100010
  [9,31:0]   00100001011    00100001011010110000000000000001
  [8,31:0]   00010001011    00010001011011000000000000000110
  [7,31:0]   10001110001    10001110001100110000000000000100
  [6,31:0]   10101110001    10101110001010000000000000000100
  [5,31:0]   00000001010    00000001010010100110000000100000
  [4,31:0]   00000001010    00000001010010100101100000100010
  [3,31:0]   00100000000    00100000000010100000000000000011
  [2,31:0]   00100000000    00100000000100100000000000001111
  [1,31:0]   00100000000    00100000000100000000000001001011
  [0,31:0]   00100010001    00100010001100010000000000000000

X1: 2,000,000 ps
```

In the above diagrams the instructions are being read from 0-14 and the rest are showing outputs "xxxxxx" and are highlighted as red. It is because the program.mips file contains only 15 instructions and those are read successfully and there are no more instructions present in the file so red output occurs here.

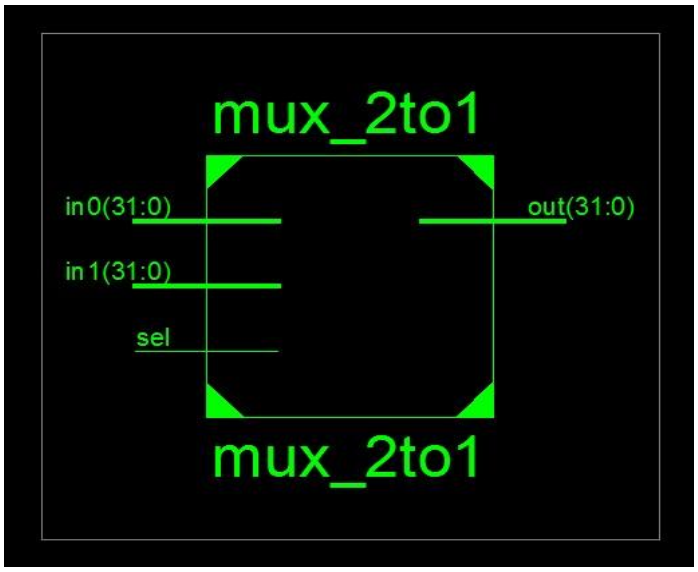
MULTIPLEXER

MUX or multiplexer is a **device** that **allows digital signals** from several sources to be **routed on to a single line of output**.

- It has **several input lines and single output line**.
- It also has data selector lines which will select the input from no. of inputs to produce specific output.

SCHEMATIC DIAGRAM:

Figure 4.1



CODE EXPLANATION:

- The **main module** contains 2 inputs in0 and in1, output out and select-bit sel.
- This **sel** will decide which input is going to be used to produce specific output.
- Then parameter DWIDTH is defined and initialized with 32 which defines the size of the inputs**. Dwidth** [1:0] shows the MSB in input is 1. Then it takes select bit. In output port it

shows the output according to the select bit. In output MSB is 1. No design, signals or bidirectional ports are initialized.

- In **Combinational logic** it assigns the value to out, when select bit is 0 the out will be assigned with in0.

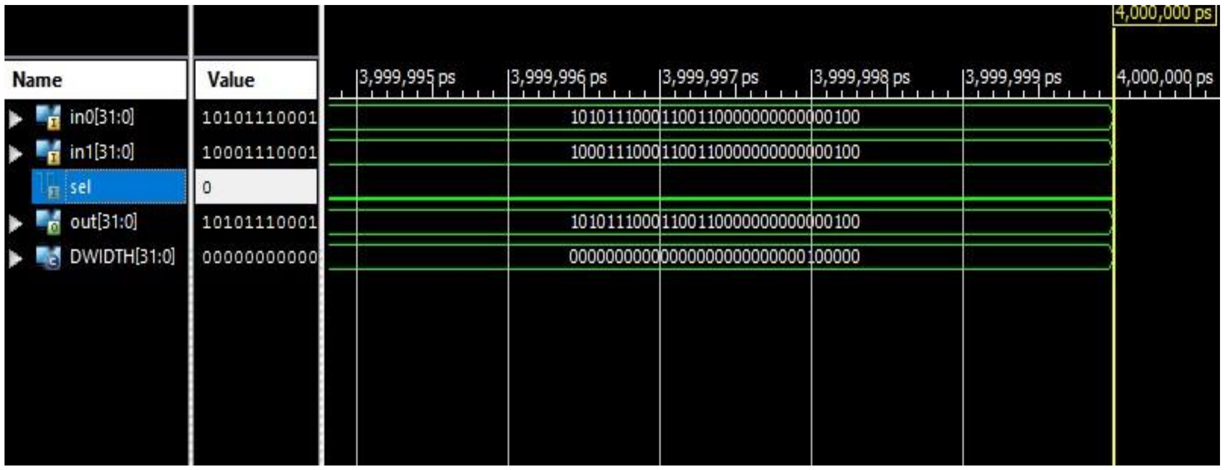


Figure 4.2

When select bit is 1 the out will be assigned with in1.

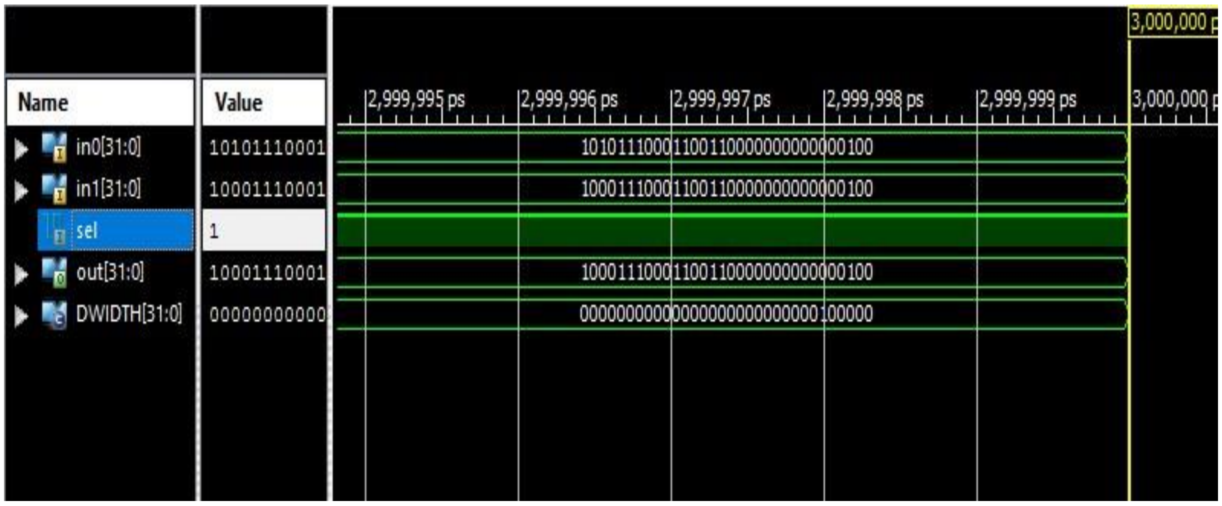


Figure 4.3

SIGN EXTENSION

Sign extension is a **block** that takes in **the input** data and **appends the bits** to it based on the **Most Significant Bit value** to maintain **sign integrity**.

It has nothing to do with endianness. Endianness is related to how the data is stored in memory and varies from one architecture to another.

SCHEMATIC DIAGRAM:

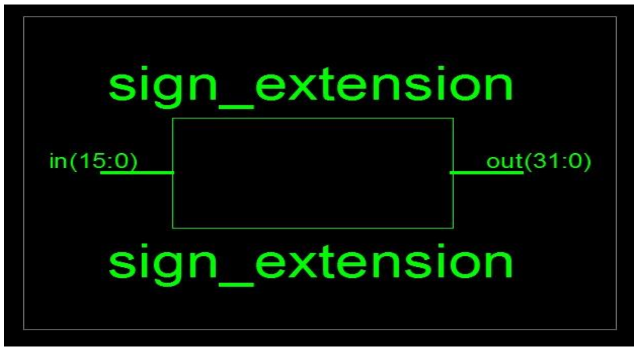
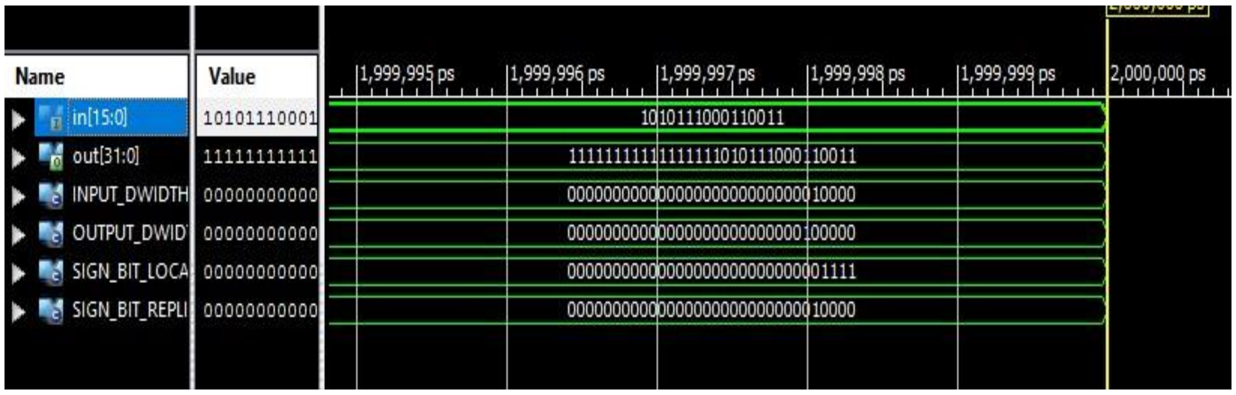


Figure 5.1

CODE EXPLANATION:

- The **main module** contains **input** of **16 bits** and **output** of **32 bits** basically **input is extended to 32 bits.**
- 2 parameters are declared **input_Dwidth and output_Dwidth** it defines the size of input as 16 and size of output as 32 bits.
- After that input is taken from user whose MSB is 1. And output port contains output whose MSB is 1 too.
- No design or bi-directional port is declared here.
- It contains some signal declarations. Sign_Bit_Location is assigned to the input width-1 because the locations start with 0 and 16=0-15 bits and then signal **SIGN_BIT_REPLICATION_COUNT** is assigned the value by subtracting the output width and input width to get the exact number for extension.
- Then **output** will be assigned with the SIGN_BIT_REPLICATION_COUNT containing input location and input width's MSB is 1 which will be displayed.
- Main focus of the program is to take 16 bit instruction and extend it to 32 bit instruction.

Figure 5.2



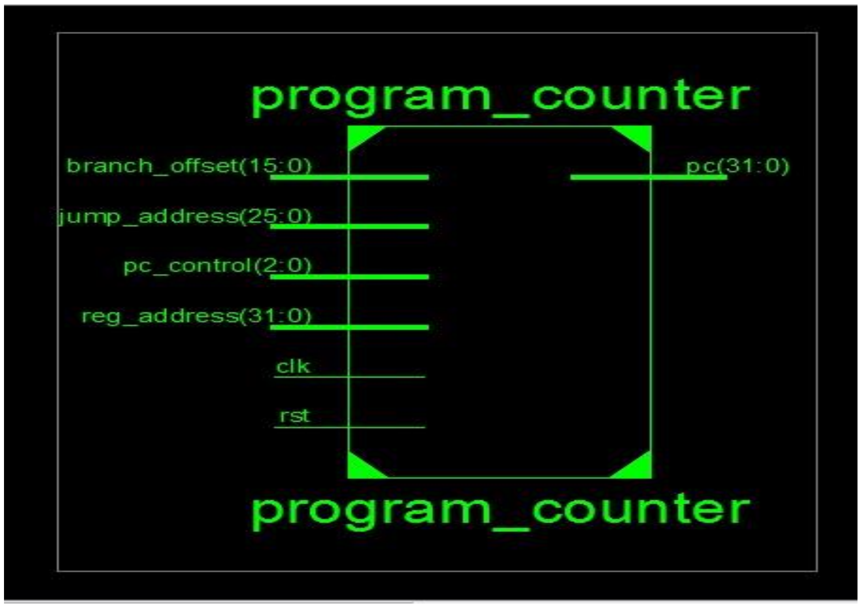
```
Name             Value
in[15:0]         10101110001     1010111000110011
out[31:0]        11111111111     11111111111111111010111000110011
INPUT_DWIDTH     00000000000     00000000000000000000000000010000
OUTPUT_DWID      00000000000     00000000000000000000000000100000
SIGN_BIT_LOCA    00000000000     00000000000000000000000000001111
SIGN_BIT_REPLI   00000000000     00000000000000000000000000010000
```

PROGRAM COUNTER

A program counter(PC) is a **register** in a computer processor that **contains the address (location) of the instruction being executed** at the current time. As each **instruction** gets **fetched**, the program counter **increases its stored value by 4**(in the given case).

SCHEMATIC DIAGRAM:

Figure 6.1



CODE EXPLANATION:

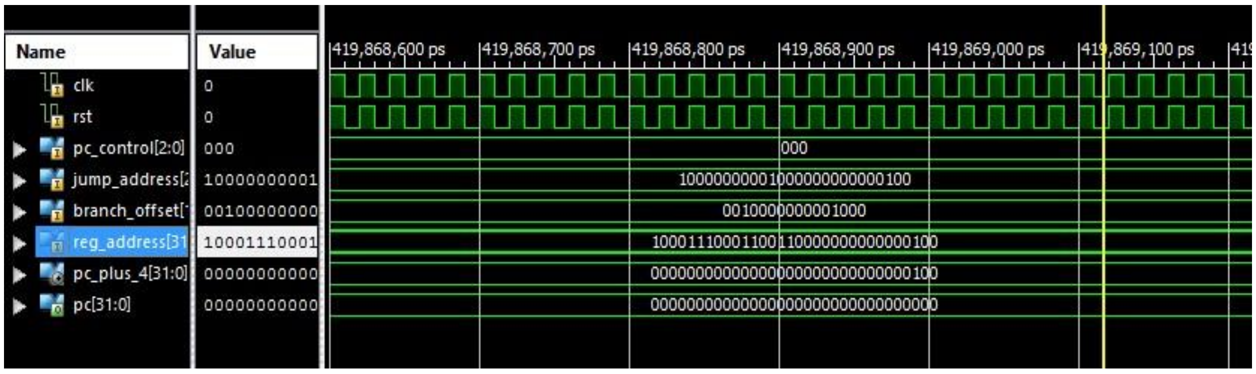
- This **module** contains **clock, reset, program counter, pc control, jump address, branch offset, register address.**
- **Reset** is generally used for **clock to reset** it at some specified position.
- **No parameters** are being used.
- **Input port** contains clock and reset.

- **Jump address of 26 bits** and other bits will be sign extended and appended from program counter. It contains 16 bits branch offset and other bits will be sign extended, shift left too. Register address of 32 bits and pc control of 4 bits.

- **Output port** contains **pc register of 32 bits**. One wire pc plus 4 is declared which will add 4 to the pc value to get the next instruction.
- Then the main logic starts on the rising edge of clock or reset. If reset occurs then pc will load the 32 bit register value to 0.And if there is no reset occurs in the clock then it will check pc control value if value is 000 then function will go on normally and will add 4 to the pc value, if control's value is 001 then jump occurs and it will add the next instruction's address jump's address and shifts it left by 2 to get the 32 bit jump's address.
- If the control's value is **010** then normal register's address will be executed and if the control's value is **011** then branch will occur and to obtain the branch's address the next instruction's address, branch offset, **shift left by 2 will** be added. And for all the other values default value of program counter will be executed. And after that the module ends.

When pc is 000:

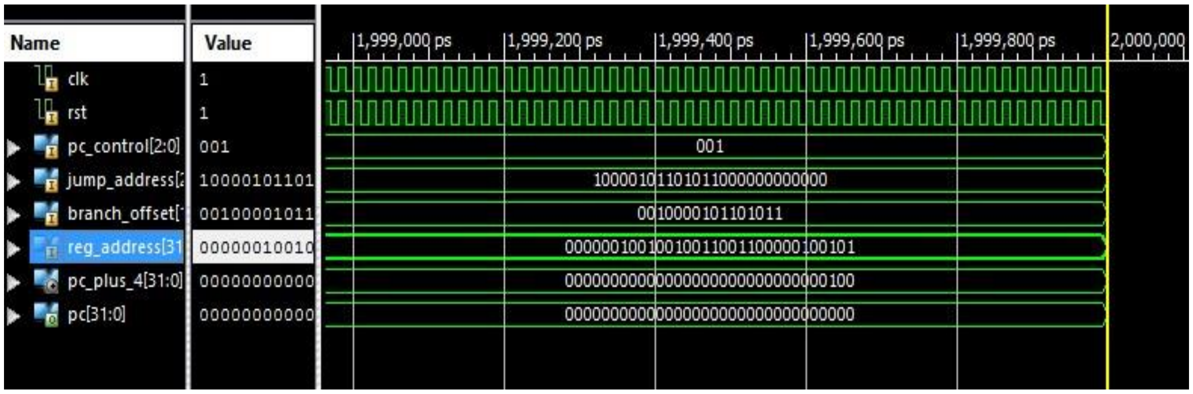
Figure 6.2



```
Name               Value         |419,868,600 ps |419,868,700 ps |419,868,800 ps |419,868,900 ps |419,869,000 ps |419,869,100 ps
clk                0
rst                0
pc_control[2:0]    000           000
jump_address[2     10000000001   10000000001000000000000100
branch_offset[1    00100000000   0010000000001000
reg_address[31     10001110001   10001110001100110000000000000100
pc_plus_4[31:0]    00000000000   00000000000000000000000000000100
pc[31:0]           00000000000   00000000000000000000000000000000
```

When pc is 001:

Figure

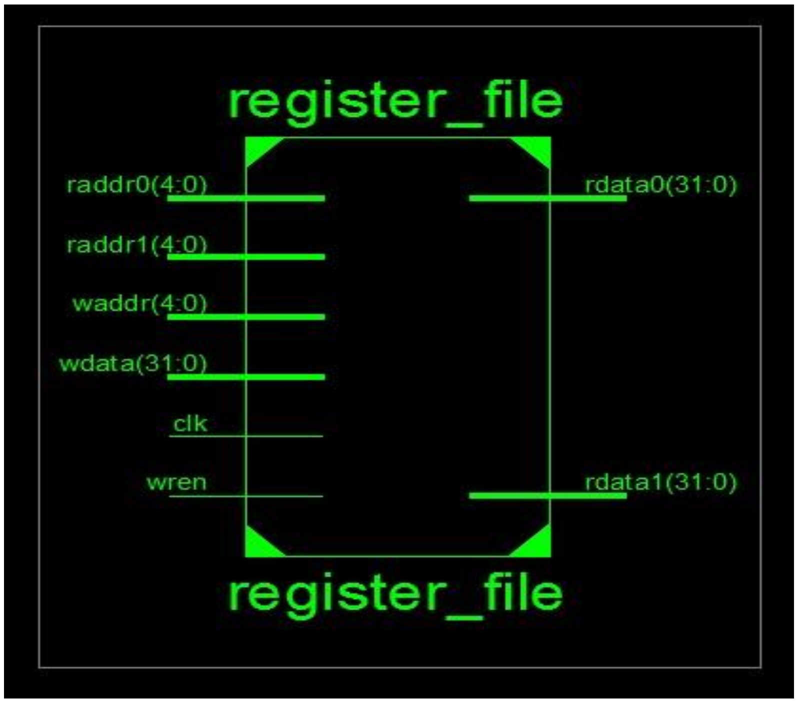


REGISTER FILE

A register file is **an array of processor registers** in a central processing unit (CPU). Register banking is the method of using a single name to **access multiple different physical registers depending on the operating mode.**

SCHEMATIC DIAGRAM:

Figure 7.1



CODE EXPLANATION:

- This **module** contains **clock, 2 data for reading and 2 addresses for reading, 1 data for writing and 1 address for writing data on that address and wren**.
- **No parameters** used in the program.
- **Input ports** contain wren, data of 32 bits for writing on address waddr which is of 5 bits. It contains clock. It has 5 bits of 2 addresses from which the data is going to be read.
- **Output port** contains data to be read of 32 bits, it is rdata0 and rdata1.
- No bidirectional ports or designs is used here.

  A register is declared is single declaration where a register of 32 bits is declared which contains reg-file of 32 bits.

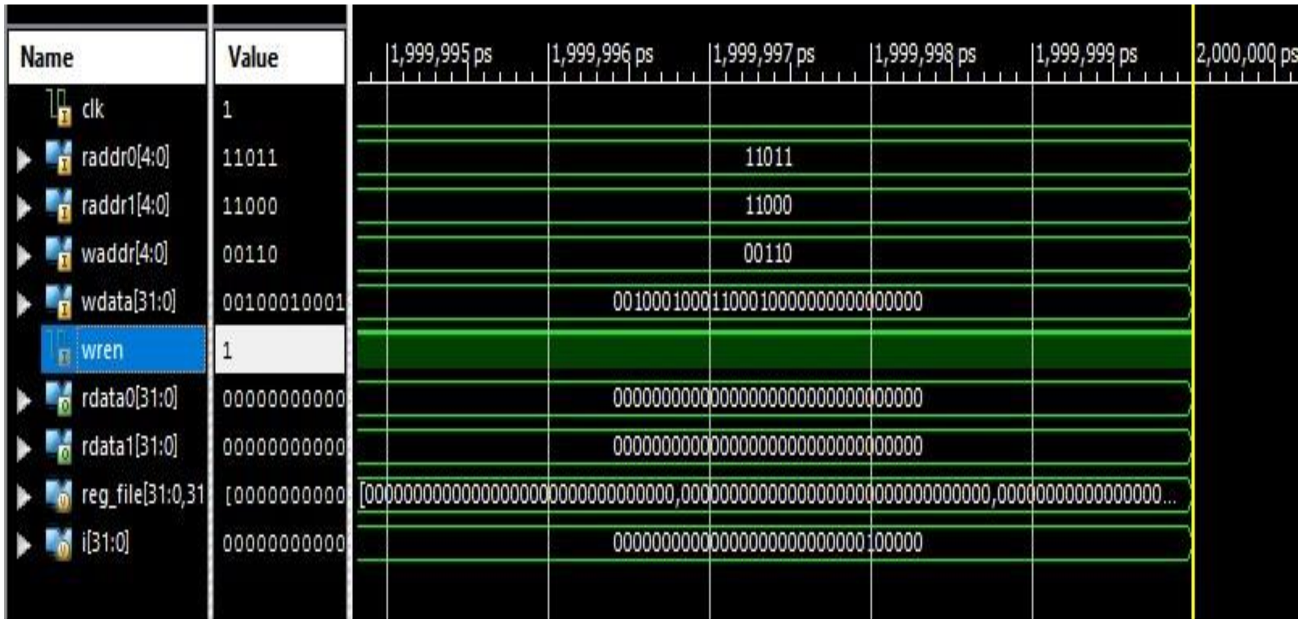
  Then the **combinational logic starts**,

The 1st data to be read (rdata) is assigned with the ref-file containing the address of that rdata and same with the address 2. It assigns 0 to each value in the reg-file, and on run time user will assign addresses to it. Then the program will wait till the fall of clock when clock shows the highest signal then it will start checks the value of wren(data at the address to be written) if condition met then it will display the address on the screen also will write data at that specified location.

This code simply reads the data present on the given addresses and write it at **waddr[31:0].**

Wren in a control signal which **enables to write data.**

Figure 7.2



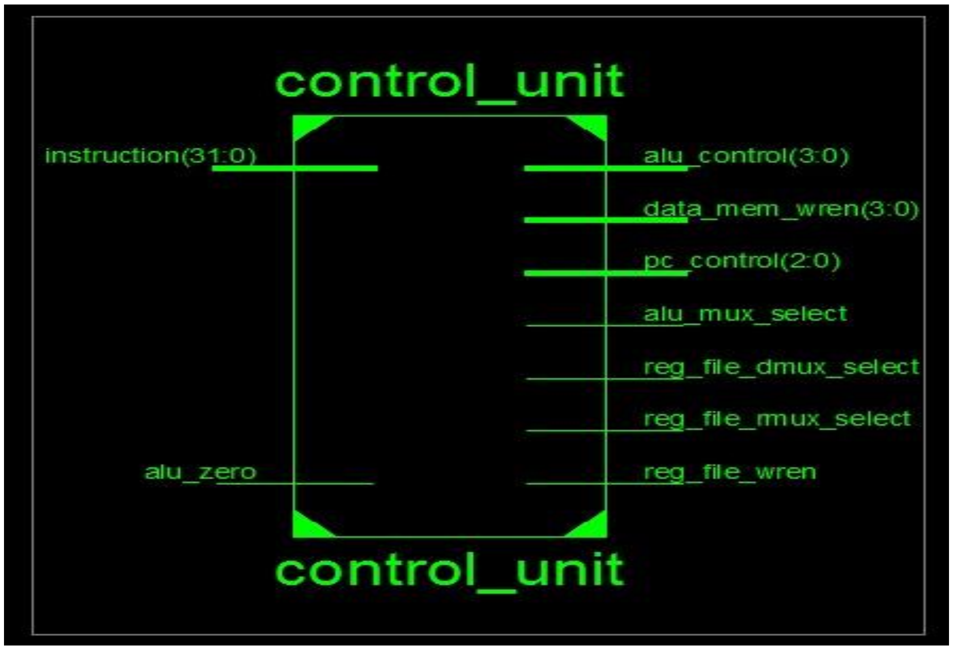
```
clk              1
raddr0[4:0]      11011          11011
raddr1[4:0]      11000          11000
waddr[4:0]       00110          00110
wdata[31:0]      00100010001    00100010001100010000000000000000
wren             1
rdata0[31:0]     00000000000    00000000000000000000000000000000
rdata1[31:0]     00000000000    00000000000000000000000000000000
reg_file[31:0,31 [0000000000    [00000000000000000000000000000000,00000000000000000000000000000000,0000000000000000...
i[31:0]          00000000000    00000000000000000000000000100000
```

CONTROL UNIT

Control Unit is the **part of the computer's** central processing unit (CPU), which **directs the operation of the processor**. It was included as part of the **Von Neumann Architecture** by **John**.

SCHEMATIC DIAGRAM:

Figure 8.1



CODE EXPLANATION:

- This **module** contains instruction, data-memory wren, register file wren(where data is located or to be written), register file dmux select(multiplexer select bit).
- DMUX is data distributer which is generally known as **de-multipluxer.** It **works opposite** to the multiplexer. The demultiplexer takes one single input data line and then switches it to any one of a number of individual output lines one at a time.
- Then it contains register file rmux select (select bit for multipluxer). Rmux means the data has been copied without any changes. The main module also contains alu control, alu zero and pc control.
- The **input port** contains instruction of 32 bits and alu zero.

The **output port** contains data memory wren of 4 bits and register file's wren, Register file's mux and alu mux, alu control of 4 bits and pc control 3 bits.

In **signal declaration** registers are assigned to multiplexers, wren(reg of size 4 bits), alu(reg of size 4 bits) and pc control(reg of size 3 bits). Signal declaration contains internal wires register address of 26 bits, register funct of 6 bits, register immediate of 16 bits, registers rs, rt, rd and shamt of 5 bits, register type of 3 bits and wire op of 6 bits.

- In **combinational logic** opcode of 6 bits from 26 to 31 is set.

The condition always starts when the instruction is given by user. Then it checks the 6 bit opcode(which will be automatically set when that instruction is given) for checking the instruction is **r-type, i-type or j-type**.

For r-type the code is **0000000**. If the instruction is r type then rs and rt will be of 5 bits and these two registers will contain operands on which operations are being performed and their result will be stored in rd whose size is 5 bits, address will be of 26 bits and immediate will be of

16 bits. funct will contain instruction's 6 bits and shamt will contain 5 bits of instruction. If the opcode contains 000010 or 000011 then the instruction is J-type. Otherwise the instruction is Itype. For storing or writing word in data-memory different conditions are being used. If the immediate of 3 bits is equals 0 then it will begin with 0-3 and 4 bits of wren 1111 will be occupied. If the immediate of 3 bits is equals 1 then it will begin with 0-2 and 4 bits of wren 0111 will be occupied. If the immediate of 3 bits is equals 2 then it will begin with 0-1 and 4 bits of wren 0011 will be occupied. If the immediate of 3 bits is equals 3 then it will begin with 0 and 4 bits of wren 0001 will be occupied and default is 1111. Register file dmux file is for load word it will check specific opcode to make the select bit 0 and to load the word otherwise select bit will be one. Register file rmux select will check either the instruction is r type or I type if instruction is r type then mux will be 1 else it will be zero.

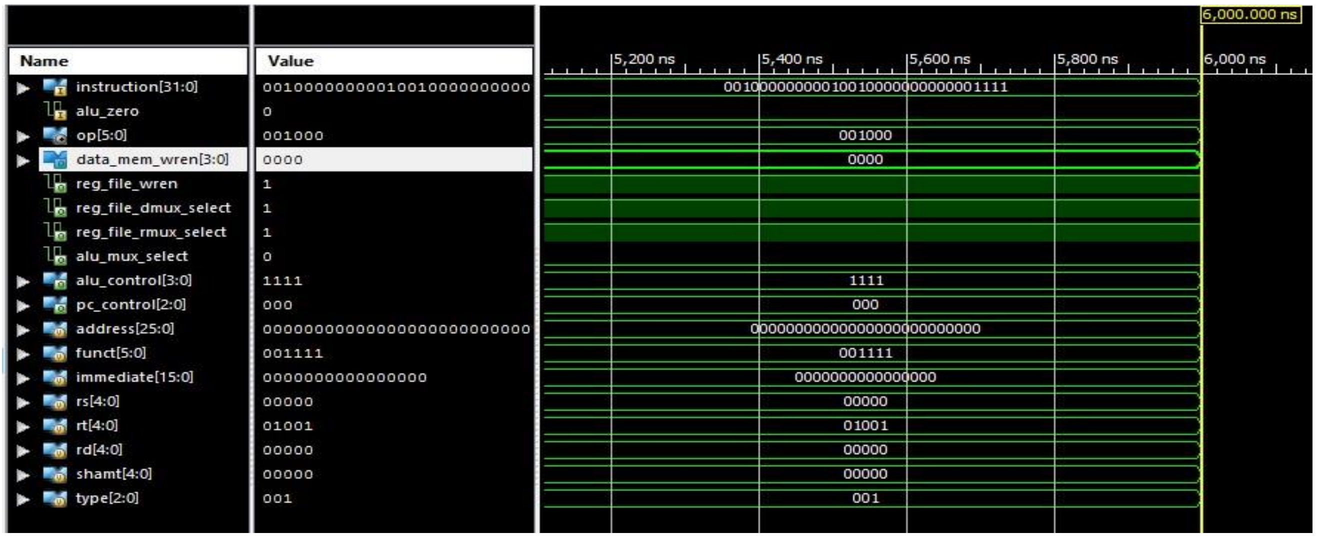
For i-type instructions **ALU mux** will be 1 and of **r or j type** instructions it will be **0.**

- For **checking** which **operation** should be performed on the instruction it will **check ALU control**. Different **conditions** are checked for different operations it will check the opcode and function keys to decide which operation should be performed. The conditions checked for this purpose will be for **r type**. If no conditions met there then obviously the instruction is of j or I type. Then further conditions will be checked either for **jump, branch equal** or **branch not equal**. Then the module ends.

- If **data_mem_wren** is **0000** data cannot be stored in memory otherwise data can be written in the memory. In I-type for load case data_mem_wren is 0000 and for store case it is 1111. For I-type and J-type it will be **0000**.
- **Signal: reg_file_wren:** it determines whether the branch signal is occurred or not. If branch signal is occurred it is 1 otherwise it is 0.
- **Signal: reg_file_dmux_select** is 0 it means variation has been occurred in loaded word and if it is 1 it means there is no variation in loaded word.
- **Signal: reg_file_rmux_**select will be 1 for R-type and 0 for I-type and J-type according to the code.
- **Signal: alu_mux_select** is 1 for I-type and 0 for J-type and J-type.
- **Signal: alu_control** determines which operation is to be applied on the operand(s).
- **PC holds** the **address** of the **next instruction**. In the output below the branch and jump has not occurred so the default part of the code is running that is 000.

R type:

- If the **opcode** is **000000** then the **immediate will be 0000000000000000** bits.
- Address will be assigned 00000000000000000000000000 bits.
- **rs** will be assigned bits for instruction[25:21].
- **rt** will be assigned bits for instruction[20:16].
- **rd** will be for instruction[15:11].
- **shamt** will be assigned bits of instruction**[10:6].**
- **Type** will be **001** and the **funct** bit will be bits of instruction **[5:0].**



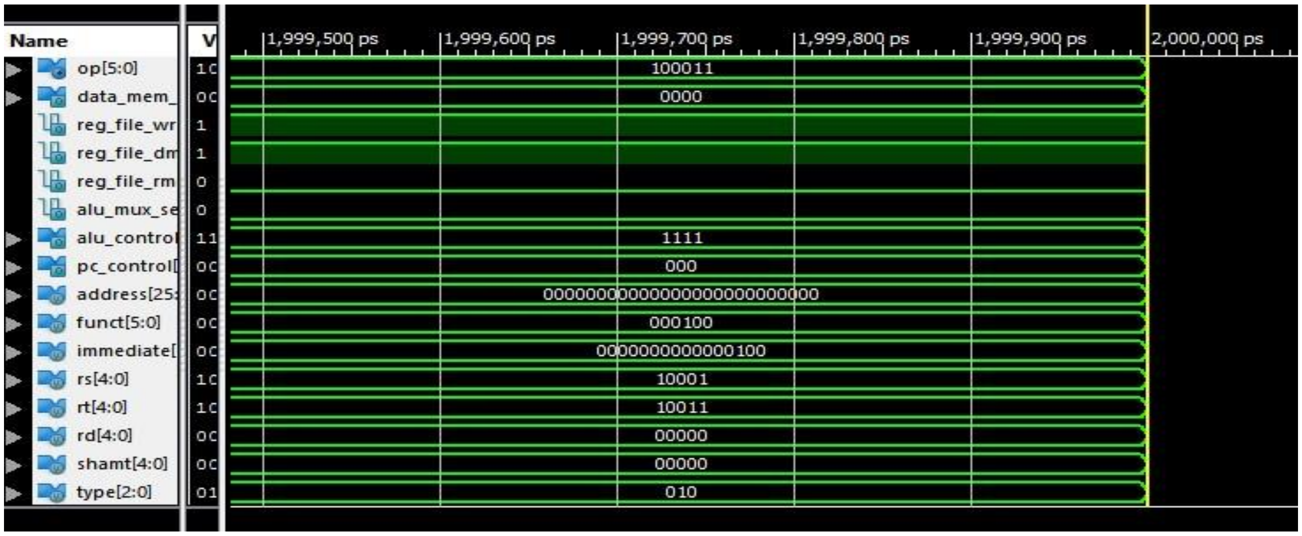
```
Name                      Value                               
instruction[31:0]         00100000000100100000000000          00100000000100100000000000001111
alu_zero                  0
op[5:0]                   001000                              001000
data_mem_wren[3:0]        0000                                0000
reg_file_wren             1
reg_file_dmux_select      1
reg_file_rmux_select      1
alu_mux_select            0
alu_control[3:0]          1111                                1111
pc_control[2:0]           000                                 000
address[25:0]             00000000000000000000000000          00000000000000000000000000
funct[5:0]                001111                              001111
immediate[15:0]           0000000000000000                    0000000000000000
rs[4:0]                   00000                               00000
rt[4:0]                   01001                               01001
rd[4:0]                   00000                               00000
shamt[4:0]                00000                               00000
type[2:0]                 001                                 001
```

Figure 8.2

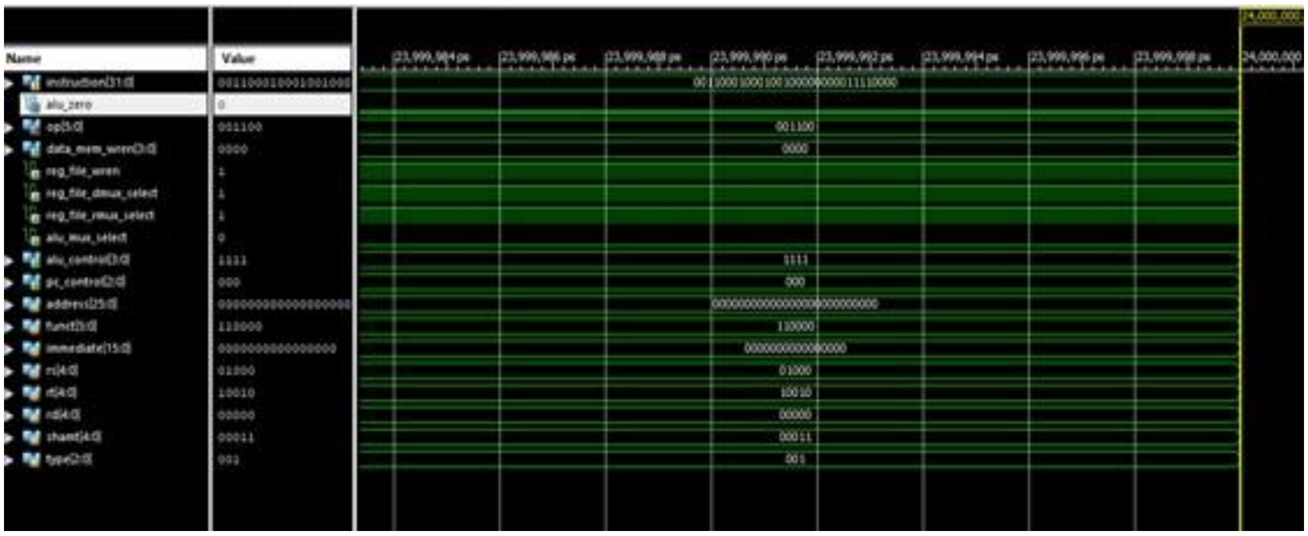
I type:

- If the **opcode is other than 000000, 000010, 000011** then Address will be assigned 00000000000000000000000000 bits and **the immediate** will be assigned instruction [15:0] bits. **rs** will be assigned bits for instruction[**25:21**].
- **rt** will be assigned bits for instruction**[20:16].**
- **rd** will be **00000** because in I type, there is **no need of rd.**
- **shamt** will be assigned bits of instruction[10:6].
- **Type will be 010** and the **funct bit** will be bits of instruction **[5:0].**

Figure 8.3



```
Name              V
op[5:0]           10     100011
data_mem_         00     0000
reg_file_wr       1
reg_file_dm       1
reg_file_rm       0
alu_mux_se        0
alu_control       11     1111
pc_control[       00     000
address[25:       00     00000000000000000000000000
funct[5:0]        00     000100
immediate[        00     0000000000000100
rs[4:0]           10     10001
rt[4:0]           10     10011
rd[4:0]           00     00000
shamt[4:0]        00     00000
type[2:0]         01     010
```



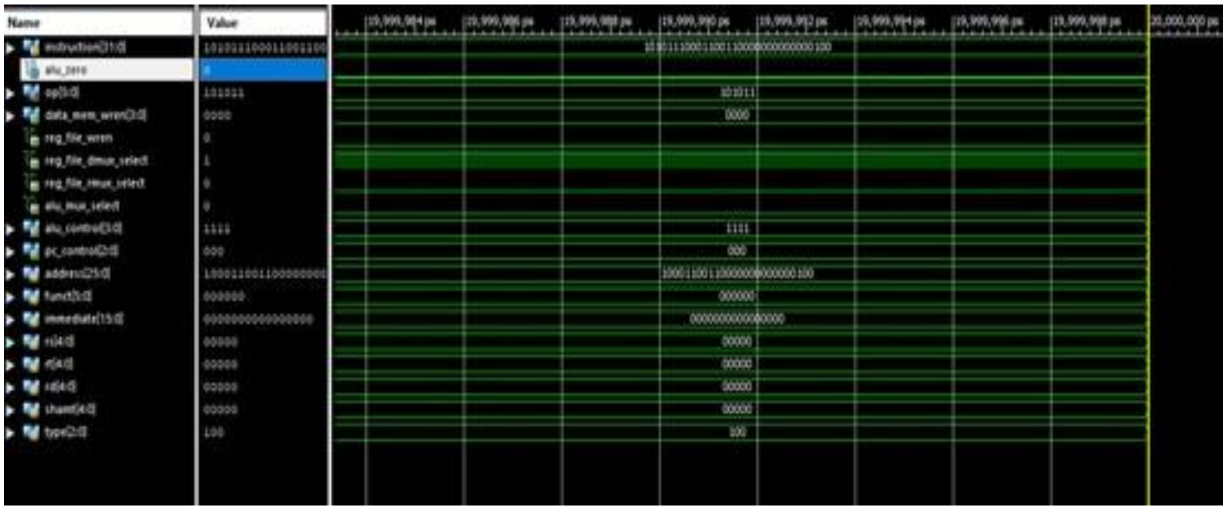
Name	Value
instruction[31:0]	0011000100010010000
alu_zero	0
op[5:0]	001100
data_mem_wren[3:0]	0000
reg_file_wren	1
reg_file_dmux_select	1
reg_file_rmux_select	1
alu_mux_select	0
alu_control[3:0]	1111
pc_control[2:0]	000
address[25:0]	0000000000000000000
funct[5:0]	110000
immediate[15:0]	0000000000000000
rs[4:0]	01000
rt[4:0]	10010
rd[4:0]	00000
shamt[4:0]	00011
type[2:0]	001

Figure 8.4

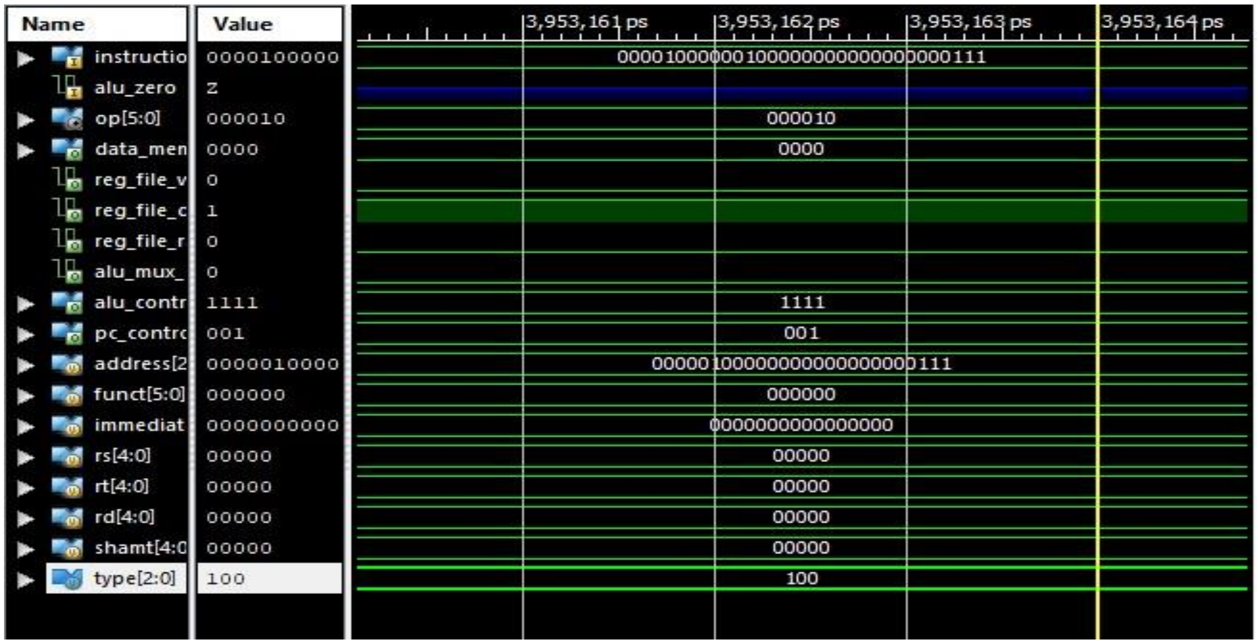
J type:

- If the **opcode is either 000010 or 000011**, then Address will be assigned instruction **[25:0]** bits and the **immediate** will be assigned 0000000000000000 bits.
- **rs** will be assigned 00000 bits.
- **rt** will be assigned 00000 bits.
- **rd** will be 00000.
- **shamt** will be assigned 00000 bits.
- **Type** will be **100** and the **funct bit** will be 000000.

Figure 8.5



Name	Value
instruction[31:0]	1010111000110011000
alu_zero	0
op[5:0]	101011
data_mem_wren[3:0]	0000
reg_file_wren	0
reg_file_dmux_select	1
reg_file_rmux_select	0
alu_mux_select	0
alu_control[3:0]	1111
pc_control[2:0]	000
address[25:0]	1000110011000000000
funct[5:0]	000000
immediate[15:0]	0000000000000000
rs[4:0]	00000
rt[4:0]	00000
rd[4:0]	00000
shamt[4:0]	00000
type[2:0]	100



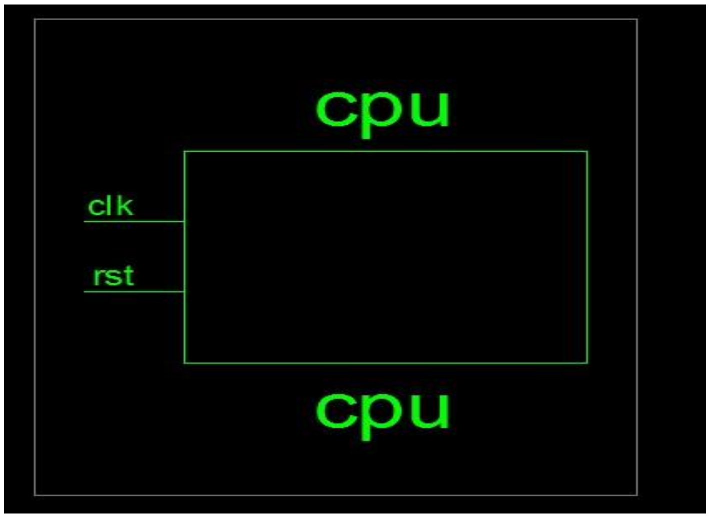
Name	Value
instructio	0000100000
alu_zero	Z
op[5:0]	000010
data_men	0000
reg_file_v	0
reg_file_c	1
reg_file_r	0
alu_mux_	0
alu_contr	1111
pc_contro	001
address[2	0000010000
funct[5:0]	000000
immediat	0000000000
rs[4:0]	00000
rt[4:0]	00000
rd[4:0]	00000
shamt[4:0	00000
type[2:0]	100

CENTRAL PROCESSING UNIT (CPU)

The computer's central processing unit (CPU) is **the portion of a computer that retrieves and executes instructions**. The CPU is essentially the brain of a **CAD** (Computer Aided Design) system. It consists of an arithmetic and logic unit (**ALU**), a control unit (**CU**), and various registers. The CPU is often simply referred to as the **processor.**

SCHEMATIC DIAGRAM:

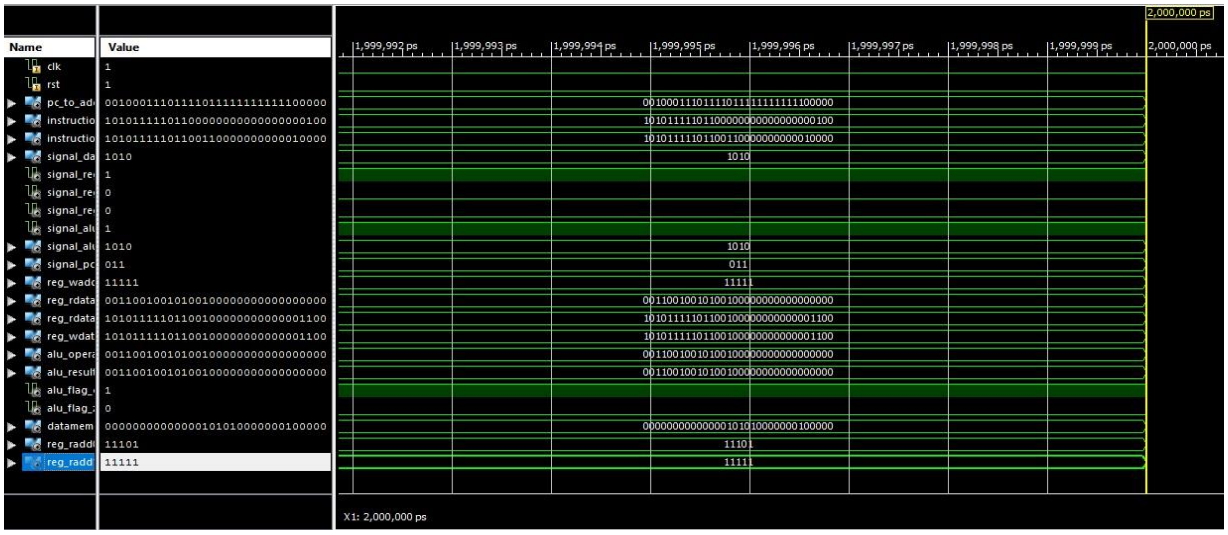
Figure 9.1



CODE EXPLANATION:

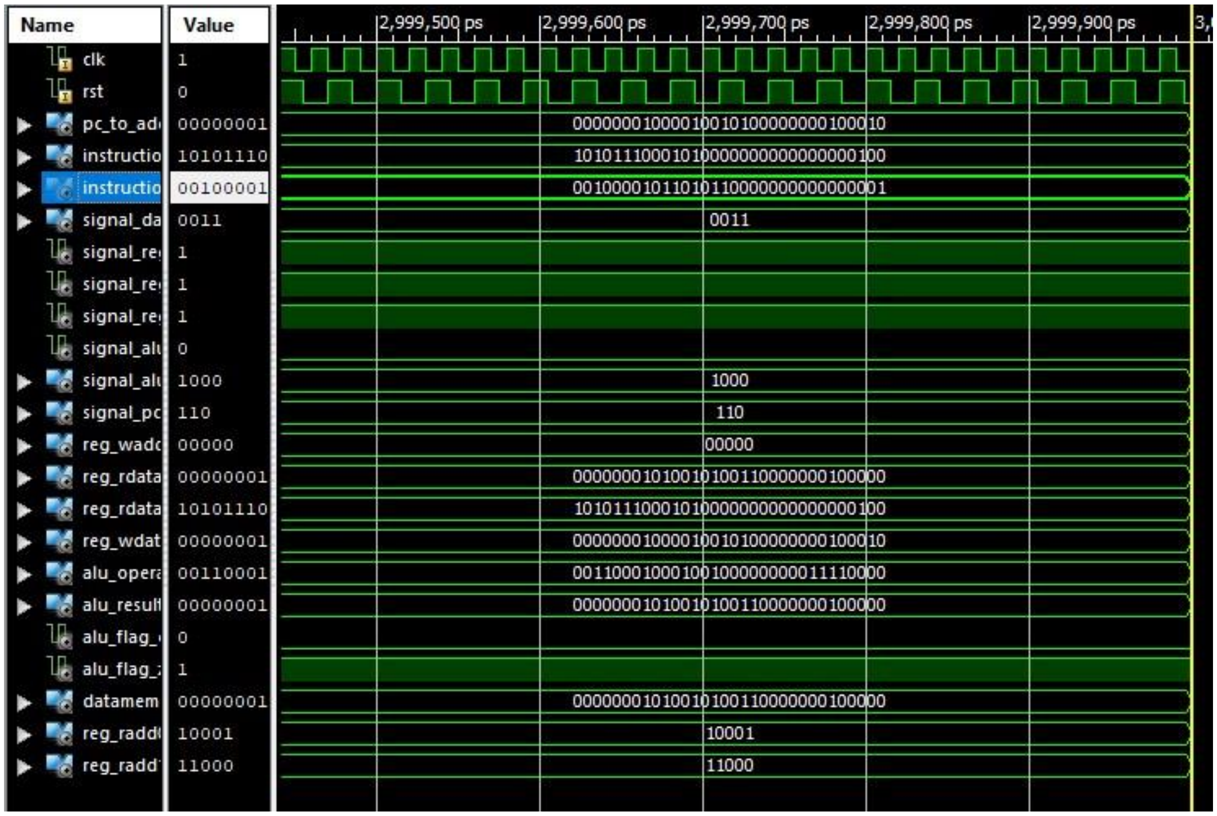
- The code collectively **displays all the functions** performed by the CPU.
- All codes are combined in one **single code.**
- All **modules are collectively included** with their respective **parameters.**
- **Sign extension** takes **a 15 bit instruction** as **input** and **gives 32 bit instruction** as **output.** 2_to_1 mux takes 2 instructions and a select bit as input and gives one instruction as output depending on the select bit.
- **Instruction memory** takes address as its parameter and gives out the instruction present at that memory address.
- **ALU module** takes 4 bits of signal_alu_control as input and performs operation on the operands respectively.
- **Data memory** takes clk as parameter and performs operations using addr, rdata, wdata and wren. No input output ports are used here. Once the clock is enabled processing starts. As in instruction memory address is taken as parameter and more specifically pc_to_address that is of 32 bits. As a result , we will get a 32 bit instruction. Only the operation related to module will be performed on it. Sign extension operation will not be performed on instruction memory module and vice versa. **'.'** is used to connect modules.
- **Physically** it is **done** by **using wires**, that's why we have declared everything as wire whose functionality is same as that of a physical wire. **Wire types can only be read or assigned**. They can **never store a value.** They will always need a continuous assignment statement to be driven. The **program counter module** takes clock as its parameter and operates with 32 bit pc_to_address , 3 bits signal_pc_control, 26 bit jump address, 16 it branch offset and 32 bit reg_data0. All the conditions will be checked as they were in the original prpgramcounter module. If value of pc_control is 000 then function will go on normally and will add 4 to the pc value, if control's value is 001 then jump occurs and it will add the next instruction's address jump's address and shifts it left by 2 to get the 32 bit jump's address. If the control's value is 010 then normal register's address will be executed and if the control's value is 011 then branch will occur and to obtain the branch's address the next instruction's address, branch offset, shift left by 2 will be added. And for all the other values default value of program counter will be executed. And after that the **module ends.**
- In the same manner **all other modules will be operated individually but are collectively written under one module.**

Figure 9.2



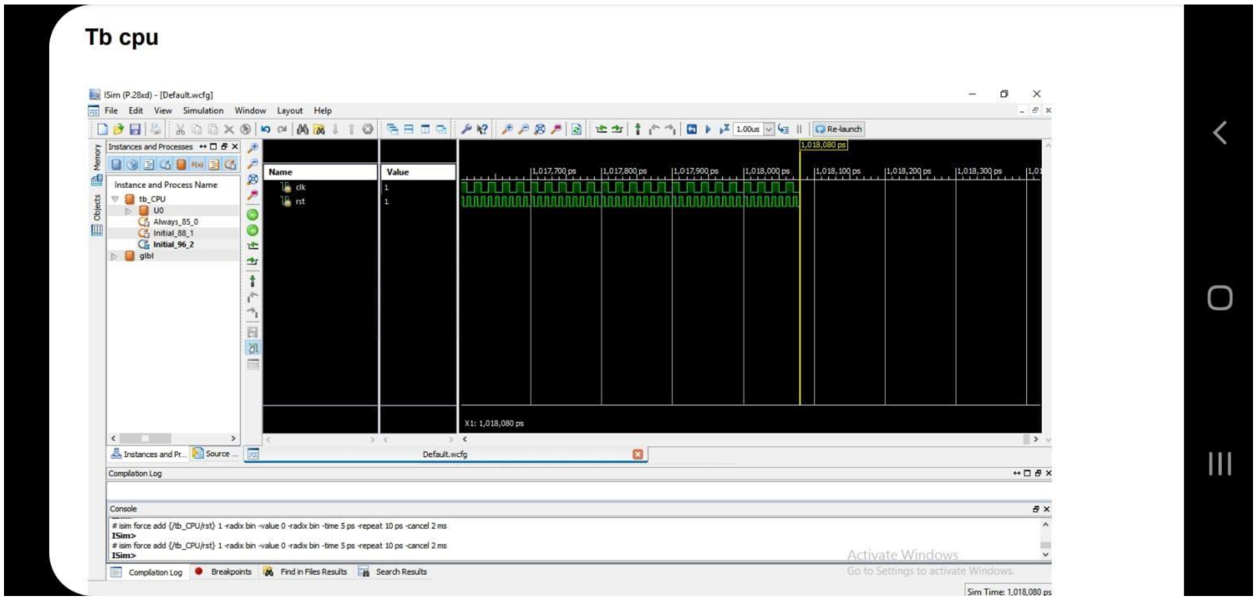
Name	Value
clk	1
rst	1
pc_to_ad	00100011101111011111111111100000
instructio	10101111101100000000000000000100
instructio	10101111101100110000000000010000
signal_da	1010
signal_re	1
signal_re	0
signal_re	0
signal_alu	1
signal_alu	1010
signal_pc	011
reg_wadd	11111
reg_rdata	00110010010100100000000000000000
reg_rdata	10101111101100100000000000001100
reg_wdat	10101111101100100000000000001100
alu_opera	00110010010100100000000000000000
alu_result	00110010010100100000000000000000
alu_flag_	1
alu_flag_	0
datamem	00000000000000101010000000100000
reg_radd	11101
reg_radd	11111

X1: 2,000,000 ps



Name	Value
clk	1
rst	0
pc_to_ad	00000001
instructio	10101110
instructio	00100001
signal_da	0011
signal_re	1
signal_re	1
signal_re	1
signal_alu	0
signal_alu	1000
signal_pc	110
reg_wadd	00000
reg_rdata	00000001
reg_rdata	10101110
reg_wdat	00000001
alu_opera	00110001
alu_result	00000001
alu_flag_	0
alu_flag_	1
datamem	00000001
reg_radd	10001
reg_radd	11000

TB CENTRAL PROCESSING UNIT (CPU)



```
# isim force add {/tb_CPU/rst} 1 -radix bin -value 0 -radix bin -time 5 ps -repeat 10 ps -cancel 2 ms
ISim>
# isim force add {/tb_CPU/rst} 1 -radix bin -value 0 -radix bin -time 5 ps -repeat 10 ps -cancel 2 ms
ISim>
```