GENISUP

INTERNSHIP REPORT

Digital System Implementation Using Verilog

OCTOBER 7, 2021

SHUBHI AGRAWAL

ACKNOWLEDGEMENT

It is a genuine pleasure to express our deep sense of thanks and gratitude to my mentor, and guides **Sh. Sheetal Swaroop Buroda**, Digital Design Engineer, NXP Semiconductors, Bangalore. His dedication and keen interest above all their overwhelming attitude to help students had been solely and mainly responsible for completing my work. Their timely advice, meticulous scrutiny, scholarly advice, and scientific approach have helped me to a very great extent to complete this internship.

I would like to express my thanks and deep regards to **Sh. Jwalant Kumar Mishra**, who made this internship possible during these difficult times. Their supportive attitude, job guidance in VLSI industry and continuous encouragement have always motivated us.

We would also like to thank our Head of Department (Electrical), **Prof. A.K. Saxena** for his assistance and providing us with the opportunity to work with Bangalore Team.

Table of Contents

Abstract

Design of digital system is a very important part of computer science and electronics, and it increased importance in recent years. Nowadays, digital systems are not only a hardware but also microprocessors, dedicated hardware and software for them. Successful design and manufacture of future digital systems is depended upon the availability of a suitable design language. To design the digital system we required precise, concise language which facilitates the specification of complex digital systems. The goal of this internship is to make theoretical knowledge applicable to practical complexities. In this report, I have mentioned all the topics which I have learned throughout this internship. In this report you'll learnt about various complex digital systems along with AMBA APB Protocol and RISC-V Processor.

VLSI Design Flow – An Overview

The journey of designing an ASIC (application specific integrated circuit) is long and involves a number of major steps – moving from a concept to specification to tape-outs. Although the end product is typically quite small (measured in nanometers), this long journey is interesting and filled with many engineering challenges.

VLSI design flow is not exactly a push-button process. To succeed in the VLSI design flow process, one must have a robust and silicon-proven flow, a good understanding of the chip specifications and constraints, and absolute mastery over the required EDA tools (and their reports).

VLSI System Design

When your VLSI specifications are completed and approved by the different parties, it's time to start thinking about the architectural design. In VLSI system design phase, the entire chip functionality is broken down into small pieces with a clear understanding of the block implementation. In this phase the working environment is documentation.

Register Transfer Level (RTL)

For digital VLSIs or for digital blocks within a mixed-signal chip, this phase includes the detailed logic implementation of the entire VLSI. This is where the detailed system specifications are converted into VHDL/Verilog language. In addition to the digital implementation, functional verification is performed to ensure the RTL design is done according to the specifications.

Synthesis

In this phase, the hardware description (RTL) is converted to a gate-level netlist. This is done using synthesis tool that takes a standard cell library, constraints and the RTL code and produces a gate-level netlist.
The results of the Synthesis tools can vary much from each other as it is running different implementations to provide best gate level netlist that meets the constraints (power,

speed, size). To verify whether the synthesis tool has correctly generated the gate-level netlist a verification should be done.

Layout

During this stage, the gate-level netlist is converted to a complete physical geometric representation. The first step is floor planning which is a process of placing the various blocks and the I/O pads across the chip area based on the design constraints.
Then placement of physical elements within each block and integration of analog blocks or external IP cores is performed. When all the elements are placed, a global and detailed routing is running to connect all the elements together.
Also, after this phase, a complete simulation/verification is required to ensure the layout phase is properly done.

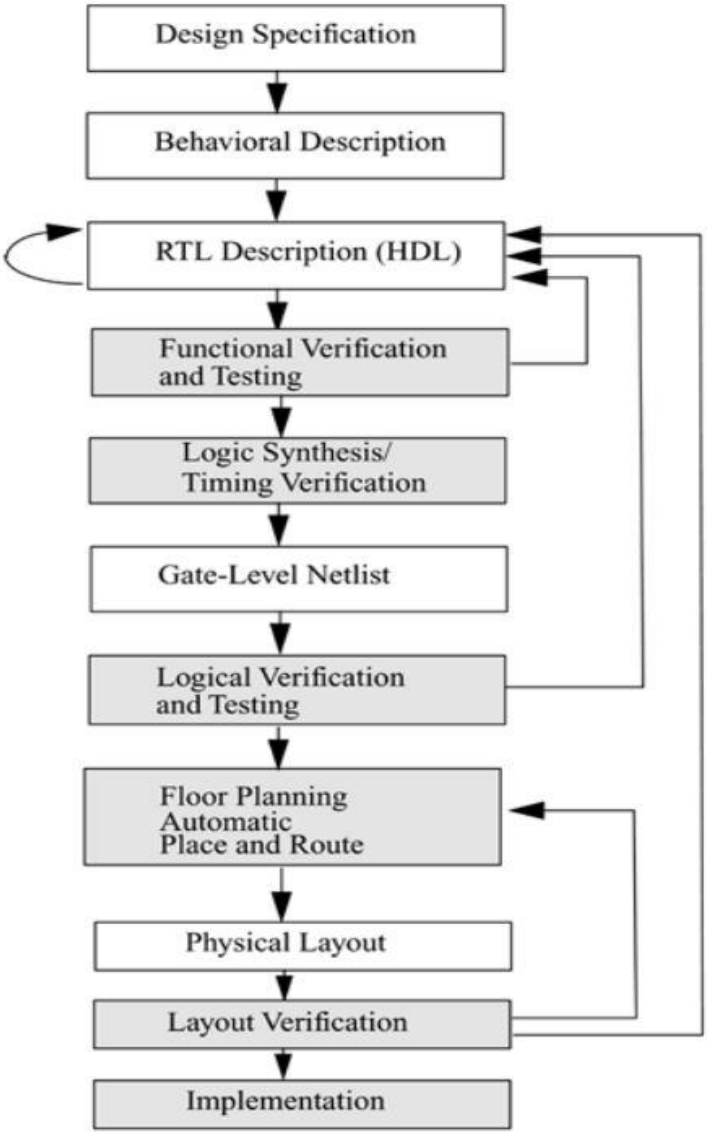


Fig. 1 ASIC Flow

Verilog

Verilog, standardized as **IEEE 1364,** is Hardware description language (HDL) used to describe a digital system like a network switch or a microprocessor or a memory or a flip−flop. It means, by using a HDL we can describe any digital hardware at any level. It is most commonly used in the design and verification of digital circuits at the register transfer level of abstraction.

Verilog supports a design at many levels of abstraction. The major three are −

- Behavioral level
- Register-transfer level
- Gate level

Behavioral level

This level describes a system by concurrent algorithms (Behavioral). Every algorithm is sequential, which means it consists of a set of instructions that are executed one by one. Functions, tasks and blocks are the main elements. There is no regard to the structural realization of the design.

Register−Transfer Level

Any code that is synthesizable is called RTL code. Designs using the Register−Transfer Level specify the characteristics of a circuit using operations and the transfer of data between the registers.

Gate Level

Gate level modelling may not be a right idea for logic design. It uses standard cells to define a design. We generated Gate level code using tools like synthesis tools and his netlist is used for gate level simulation and for backend.

Simulation Tool

Icarus Verilog (Used Version 11.0)

An open-source *Icarus Verilog* is a Verilog simulation and synthesis tool. It operates as a compiler, compiling source code written in Verilog (IEEE-1364) into some target format. For batch simulation, the compiler can generate an intermediate form called *vvp assembly*. This intermediate form is executed by the ``vvp'' command. For synthesis, the compiler generates netlists in the desired format.

The main porting target is Linux, although it works well on many similar operating systems. Various people have contributed precompiled binaries of stable releases for a variety of targets. These releases are ported by volunteers, so what binaries are available depends on who takes the time to do the packaging.

ModelSim–Intel FPGA Starter Edition 10.5b

ModelSim is a multi-language environment by Mentor Graphics, for simulation of hardware description languages such as VHDL, Verilog and SystemC, and includes a built-in C debugger. ModelSim can be used independently, or in conjunction with Intel Quartus Prime, PSIM, Xilinx ISE or Xilinx Vivado.
ModelSim required licence to work with advanced function. I used Modelsim-Intel FPGA Starter Edition 10.5b in this internship program.

Implementation of Simple Designs.

WEEK 1 MODULES

- D-FlipFlop with Edge Detector Circuit using gate-level modeling.
- Lemmings (2-D) using Behavioral Level modelling.
- User defined FSM using Behavioral Level modelling.

1. D-Flipflop: A D (or Delay) Flip flop is a digital electronics circuits used to delay the change of state of its output signal until the next rising edge of a clock input signal occurs. In this below design there is an edge detector circuit along with a d-latch which is behaving as a D-flipflop. It is implemented in gate level model.

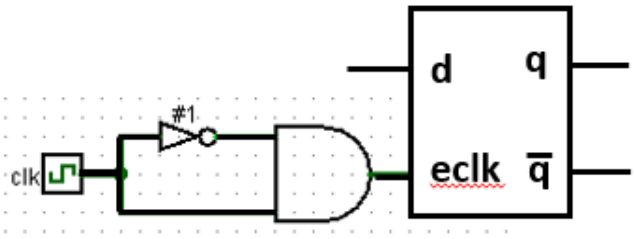


Fig. 1.1 D-FlipFlop with Edge Detector Circuit

D	CLK	Q(t+1)	Comments
1	↑	1	Set
0	↑	0	Reset

↑ = clock transition LOW to HIGH

Fig. 1.2 D-Flipflop Truth table.

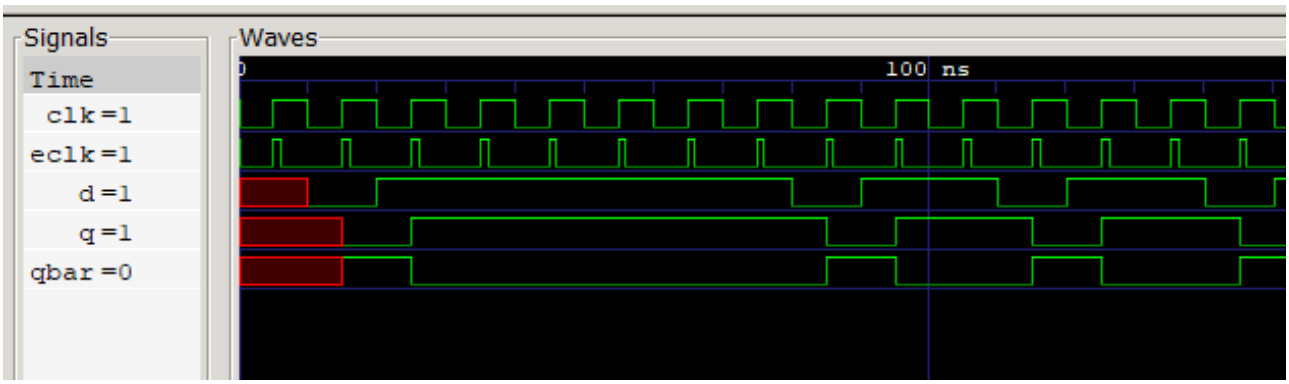


Fig. 1.3 D-FlipFlop with Edge Detector Circuit Simulation Waveform.

Lemmings (2-D)

In the Lemmings' 2D world, Lemmings can be in one of two states: walking left or walking right. It will switch directions if it hits an obstacle. In particular, if a Lemming is bumped on the left, it will walk right. If it's bumped on the right, it will walk left. If it's bumped on both sides at the same time, it will still switch directions. It is model using Simple FSM logic.

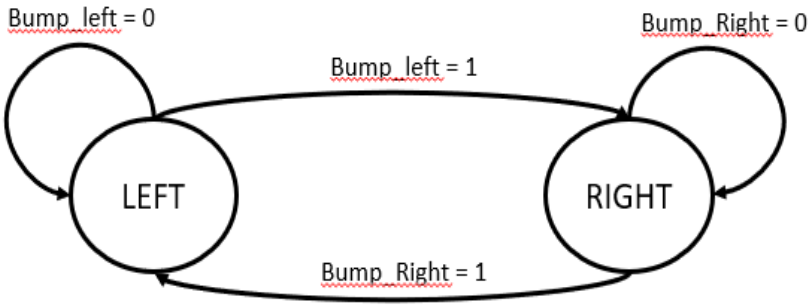


Fig. 1.4 State Diagram of Lemmings (2-D).

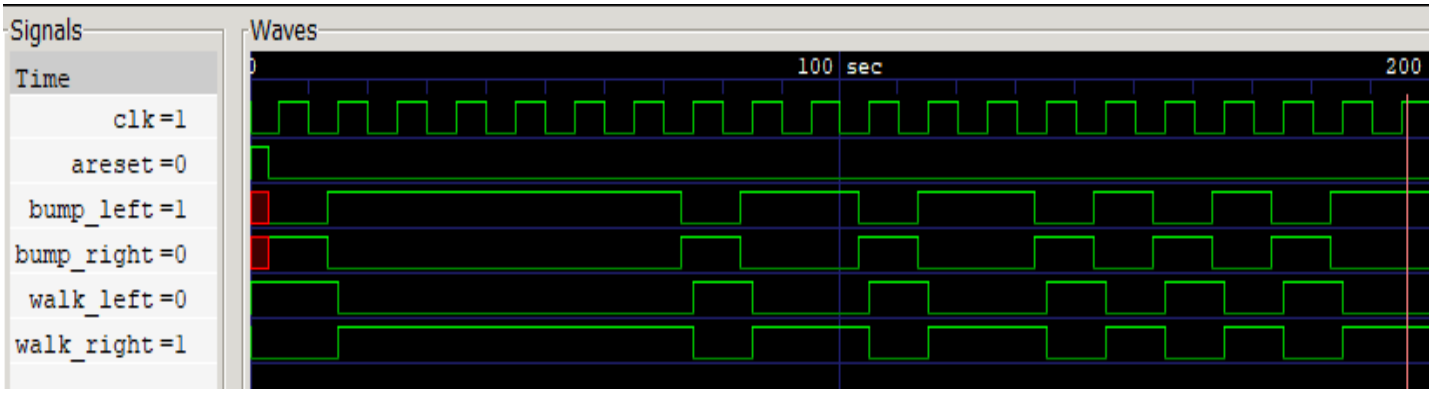


Fig. 1.5 Lemming(2-D) simulation Waveform.

User Defined FSM

This is a special type of FSM designed using shift register. In this FSM, user will give the sequence that has to be detected as an input and it will detect for the same sequence. Below is a simulation result of it as:

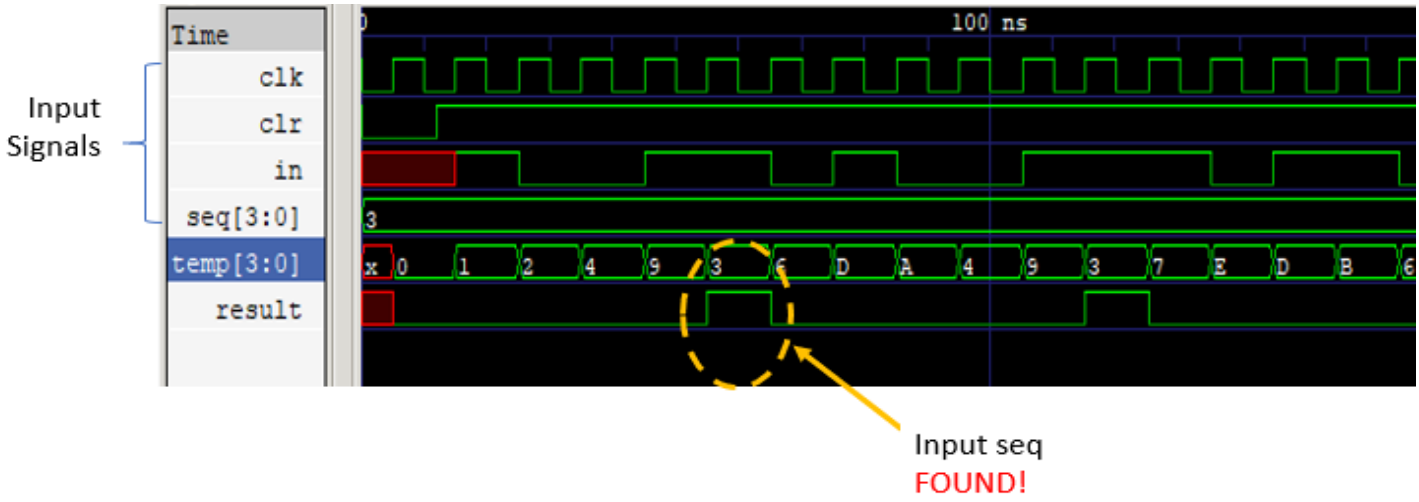


Fig. 1.6 User-defined FSM simulation Waveform.

Textbook Exercise and Implement Logic Function using ONLY NAND gate.

Steps Involve in Implementing Logic Functions Using Only NAND Gate

- Create a truth table.
- Perform Karnaugh map minimization.
- Write the corresponding Boolean Expression.
- Taking a circuit described using AND, OR and NOT gates.
- Replace the logic gates with the corresponding circuit using NAND gate.

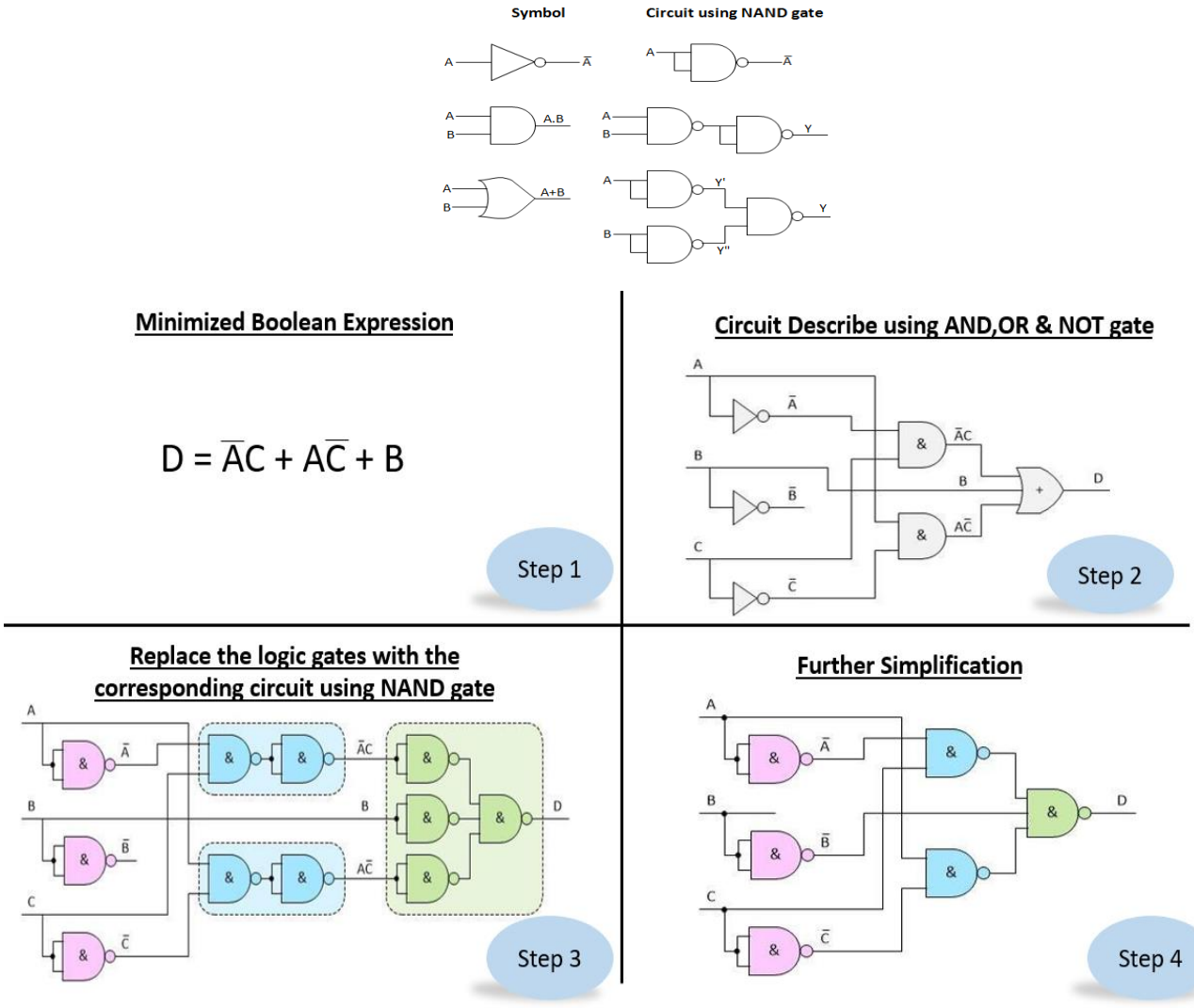
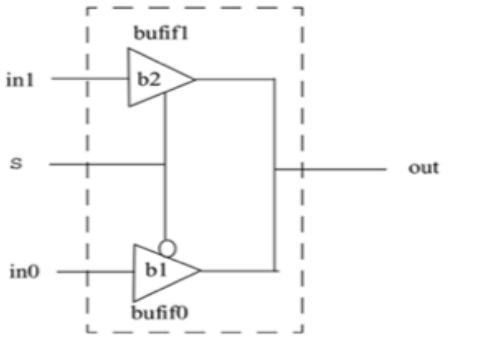


Fig. 2.2 Pictorial Representation of implementing logic Function using ONLY NAND gate.

2 to 1 MUX

A mux (or multiplexer) to pass one of the inputs from the output port. It is designed using tri-state buffer with rise time, fall time and turnoff delay.



The delay specification for gates b1 and b2 are as follows:

	Min	Typ	Max
Rise	1	2	3
Fall	3	4	5
Turnoff	5	6	7

Fig. 2.3 Circuit diagram of 2-1 mux with given delays.

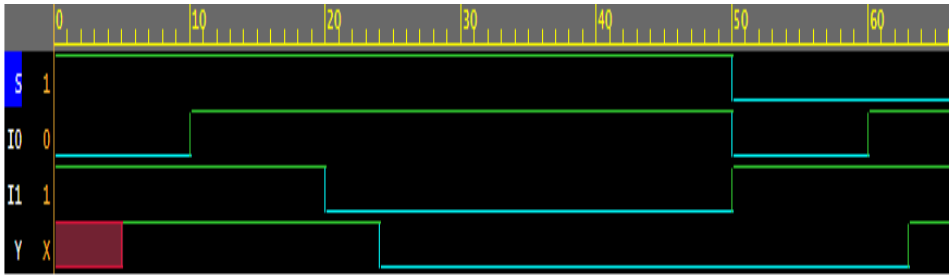


Fig. 2.4 Simulation Waveform of MUX.

DATAFLOW MODELLING

WEEK 3 MODULES

- Simulation results of various Shift Registers.
- Booth Multiplier algorithm and its simulation result.

1. Serial In Serial Out (or SISO)

It is a type of shift register which contain the series of flipflop connected together in such a way that the output of last flipflop is connected with the input of next flipflop. It contains 1 bit input connected to 1st flipflop and 1 bit output taking from last flipflop.

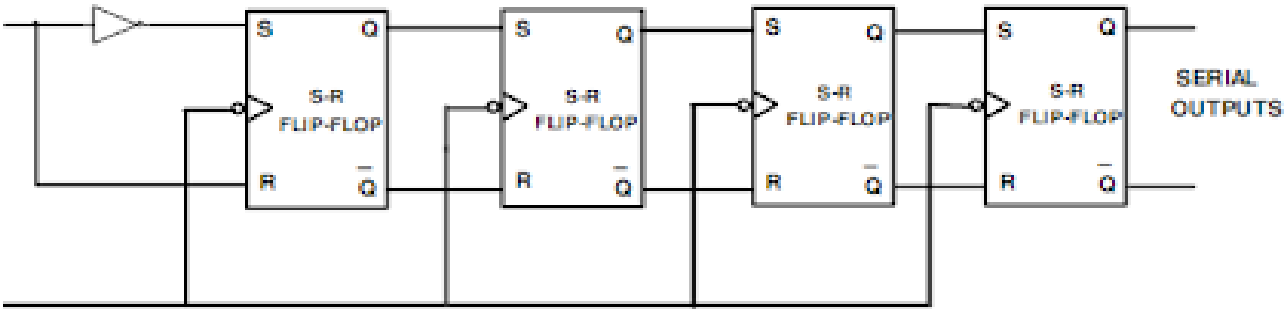


Fig. 3.1 SISO circuit diagram.

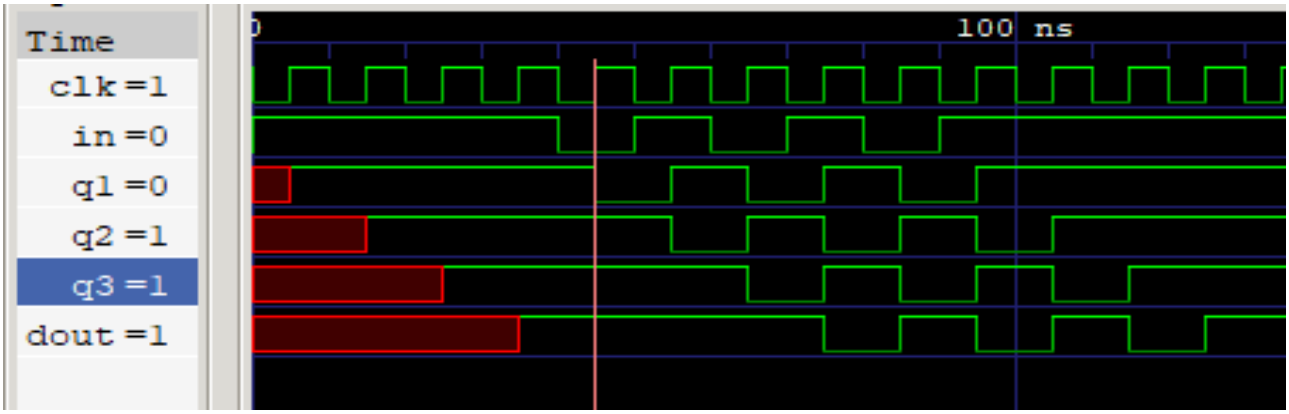


Fig. 3.2 SISO simulation waveform.

2. Serial In Parallel Out (or SIPO)

It is a type of shift register which contain the series of flipflop connected together in such a way that the output of last flipflop is connected with the input of next flipflop. It contains 1 bit input connected to 1st flipflop and 4 bits output taking from all flipflop outputs.

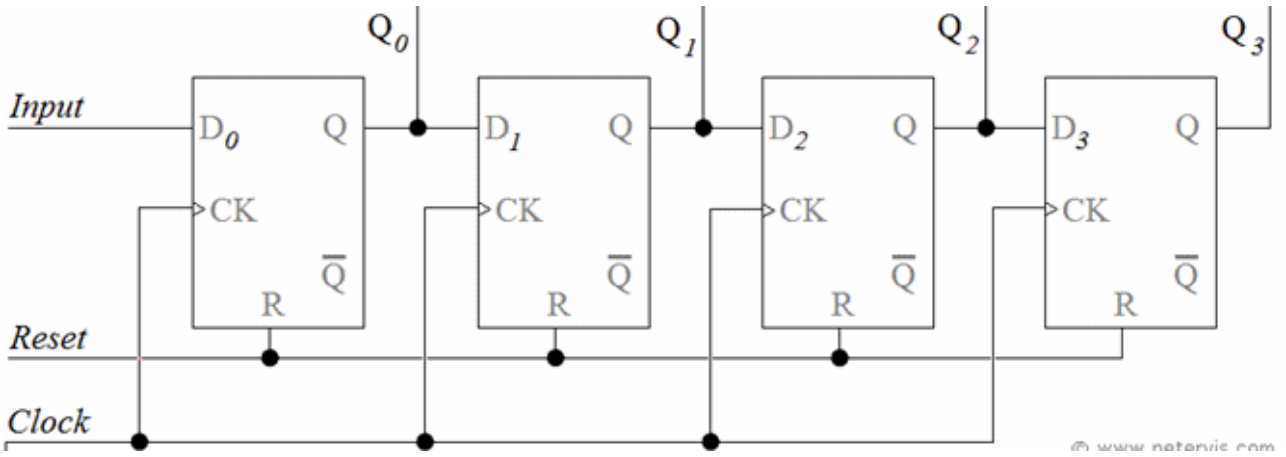


Fig. 3.3 SIPO circuit diagram.

Fig. 3.4 SIPO simulation waveform.

3. Parallel In Serial Out (or PISO)

It is a type of shift register which contain the series of flipflop connected together in such a way that the output of last flipflop is connected with the input of next flipflop. It contains 4 bits input connected to all flipflop and 1 bit output taking from last flipflop output.

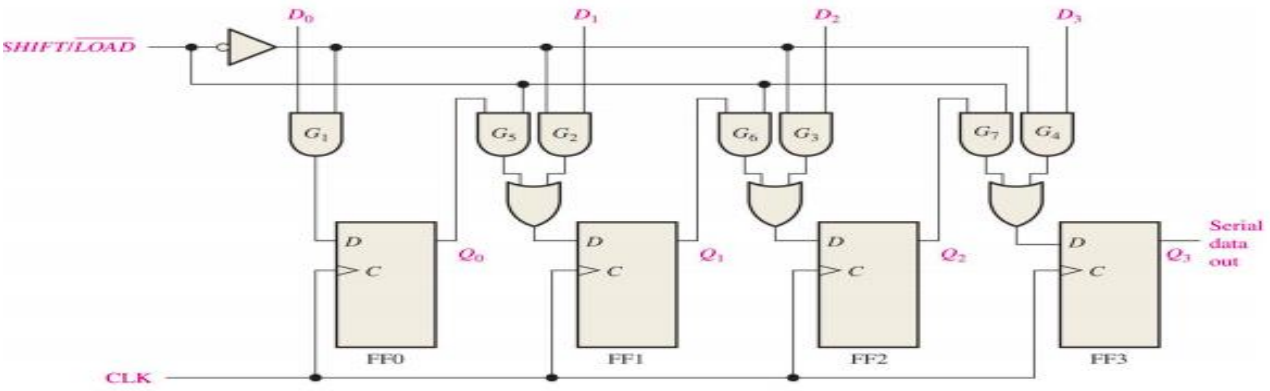


Fig. 3.5PISO circuit diagram.

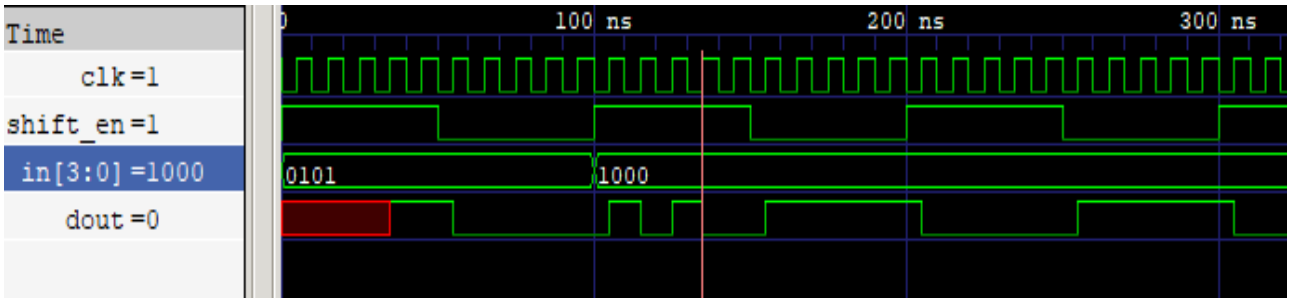


Fig. 3.6 PISO simulation waveform.

4. Parallel In Parallel Out (or PIPO)

It is a type of shift register which contain the series of flipflop connected together in such a way that the output of last flipflop is connected with the input of next flipflop. It contains 4 bits input connected to all flipflop and 4 bits output taking from all flipflop outputs.

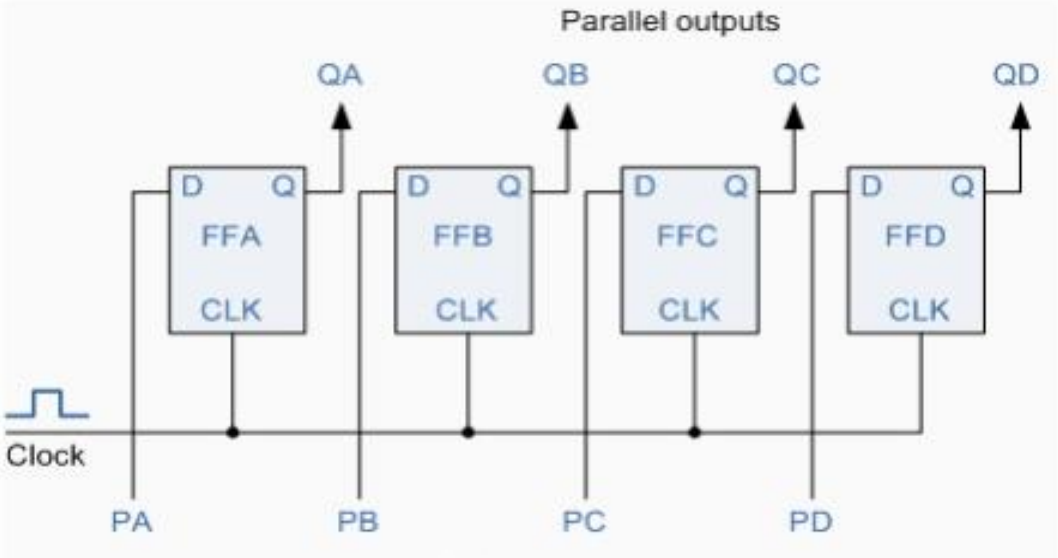


Fig. 3.7 PIPO circuit diagram.

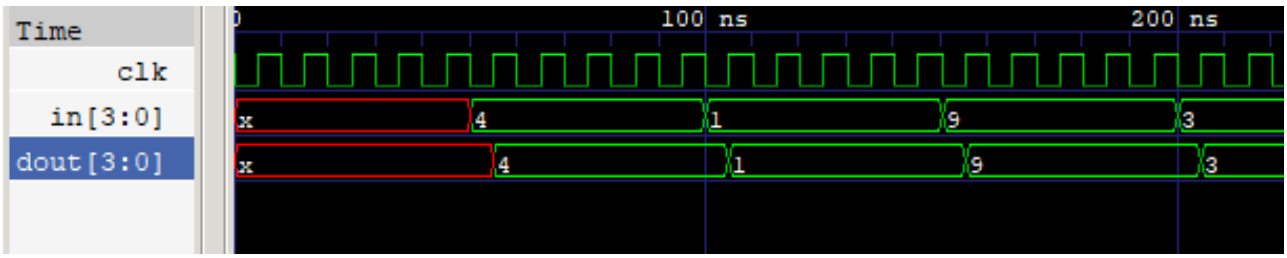


Fig. 3.8 PIPO simulation waveform.

Booth Multiplier (Without Using Clock Signal)

The booth algorithm is a multiplication algorithm that allows us to multiply the two signed binary integers in 2's complement, respectively. It is also used to speed up the performance of the multiplication process. It is very efficient too. It works on the string bits 0's in the multiplier that requires no additional bit only shift the right-most string bits and a string of 1's in a multiplier bit weight 2^k to weight 2^m that can be considered as $\mathbf{2^{k+1} - 2^m}$.

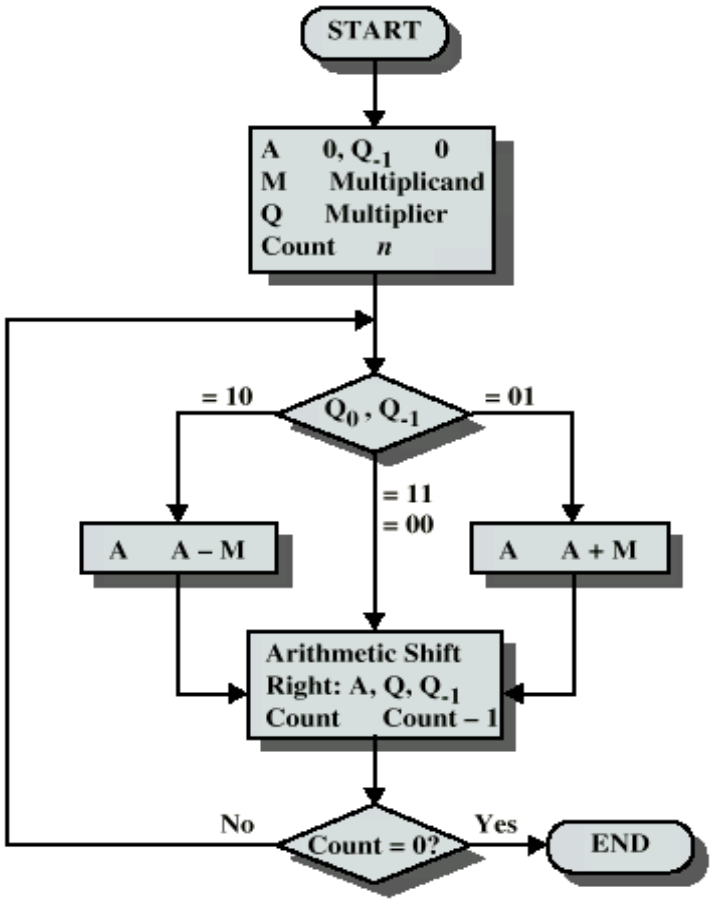


Fig. 3.7 Booth algorithm flowchart.

Working on the Booth Algorithm

1. Set theMultiplicand andMultiplier binary bits as M and Q, respectively. And A and Q_{n-1} registers value to 0.
2. Count represents the number of Multiplier bits (Q), and it is a sequence counter that is continuously decremented till equal to the number of bits (n) or reached to 0.
3. A Qn represents the last bit of the Q, and the Q_{n-1} shows the incremented bit of Qn by 1.
4. On each cycle of the booth algorithm, Q_n and Q_{n-1} bits will be checked on the following parameters as follows:
    i. When two bits Q_n and Q_{n-1} are 00 or 11, we simply perform the arithmetic shift right operation (ashr) to the partial product A. And the bits of Qn and Q_{n-1} is incremented by 1 bit.
    ii. If the bits of Q_n and Q_{n-1} is shows to 01, the multiplicand bits (M) will be added to the A (Accumulator register). After that, we perform the right shift operation to the A and Q bits by 1.
    iii. If the bits of Q_n and Q_{n-1} is shows to 10, the multiplicand bits (M) will be subtracted from the A (Accumulator register). After that, we perform the right shift operation to the A and Q bits by 1.
5. The operation continuously works till we reached n - 1 bit in the booth algorithm.
6. Results of the Multiplication binary bits will be stored in the A and Q registers.

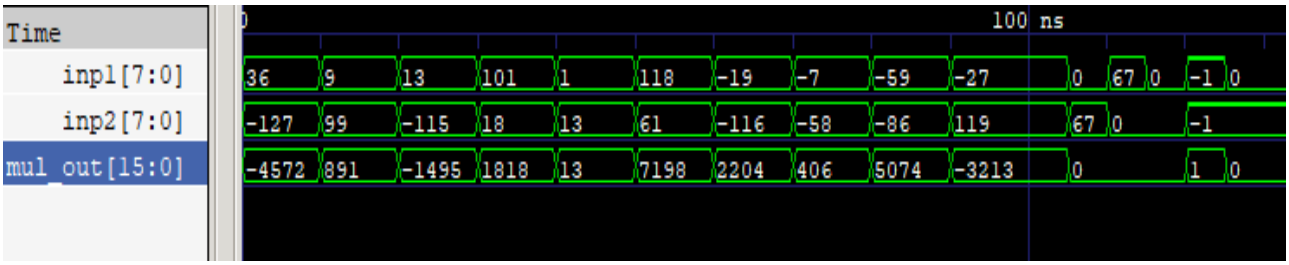


Fig. 3.8 Booth algorithm simulation waveform.

Behavioural Modelling

- Blocking & Non-Blocking Assignment.
- case, casex &casez statement.
- RAM Based 001 FSM.
- RoboAnt.
- 32 bits ALU.

1. Blocking & Non-Blocking Assignment

A ***blocking*** statement will not block the execution of statement that are in parallel block, means it will execute sequentially. A blocking statement is a ***one step*** process
i.e., a) evaluate the RHS of the expression and update the LHS without any delay.
Nonblocking assignment allow scheduling of assignment that are executed in sequential block. Nonblocking is a ***two-step*** process
i.e., a) Evaluate the RHS expression at the beginning of time step and
b) Update the LHS at the end of time step.

Initial Value of a = 1, b = 0;

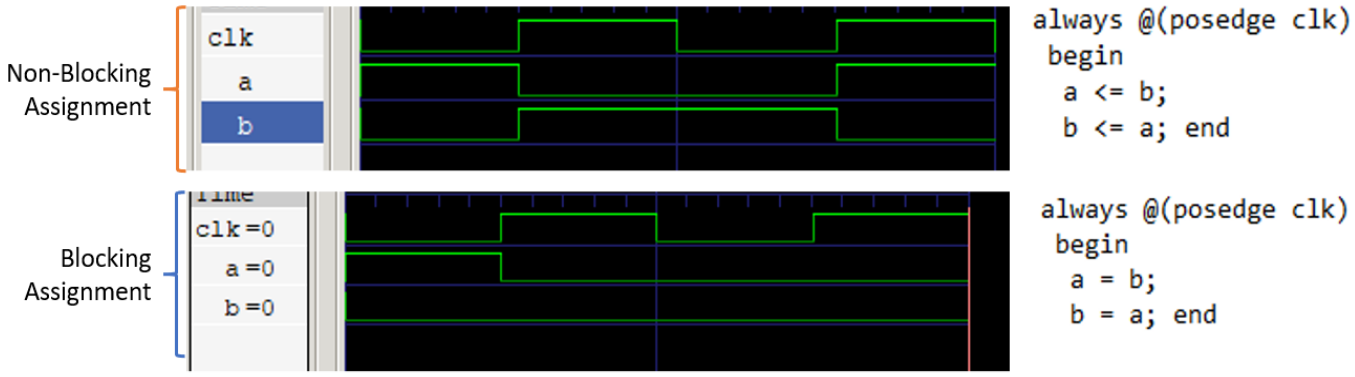


Fig. 4.1 Simulation waveform of blocking and non-blocking assignments.

2. case, casex & casez statement

The basic differences are as follows (referring to pre-synthesis simulation outputs):

- **case** statement considers x or z as it is. So, a case expression containing x or z will only match a case item containing x or z at the corresponding bit positions. If no case item matches, then default item is executed.
- **casez** statement considers z as don't care.
- **casex** statement considers both x and z as don't care.

3. 32-bit ALU

ALU (Arithmetic Logical Unit) is a unit by which we can perform arithmetic, logical, relational, shift and many more operations. It is a 32-bit ALU it means it contain 2 32-bits inputs and a 32-bits output port. It contains4-bit opcode input port which is use to choose the type of operation required to be done. This ALU consist of 5 flags, carry flag, sign flag, zero flag, overflow flag and parity flag.

Opcode	Operation
4'd0	Addition
4'd1	Subtraction
4'd2	Multiplication
4'd3	Bitwise AND
4'd4	Bitwise NAND
4'd5	Bitwise OR
4'd6	Bitwise NOT
4'd7	Bitwise NOR
4'd8	Left Shift by 1
4'd9	Right Shift by 1
4'd10	Arithmetic Right Shift
4'd11	Arithmetic Left Shift
4'd12	Decrement
4'd13	Increment
4'd14	Greater than
4'd15	Equal to

Fig. 4.2 ALU operation table.

4. RAM-Based 001 FSM

This is a special type of FSM in which we can assume RAM address as a present state and RAM addressed word as next state. It is used to reduce the complicated combination circuit from the design. But in this design, I used RAM addressed word bit as a present state data of the FSM. Below diagram is showing the 001-sequence detector FSM.

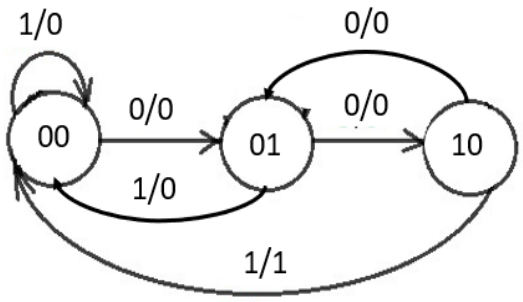


Fig. 4.3 FSM for 001 sequence.

Simulation Result

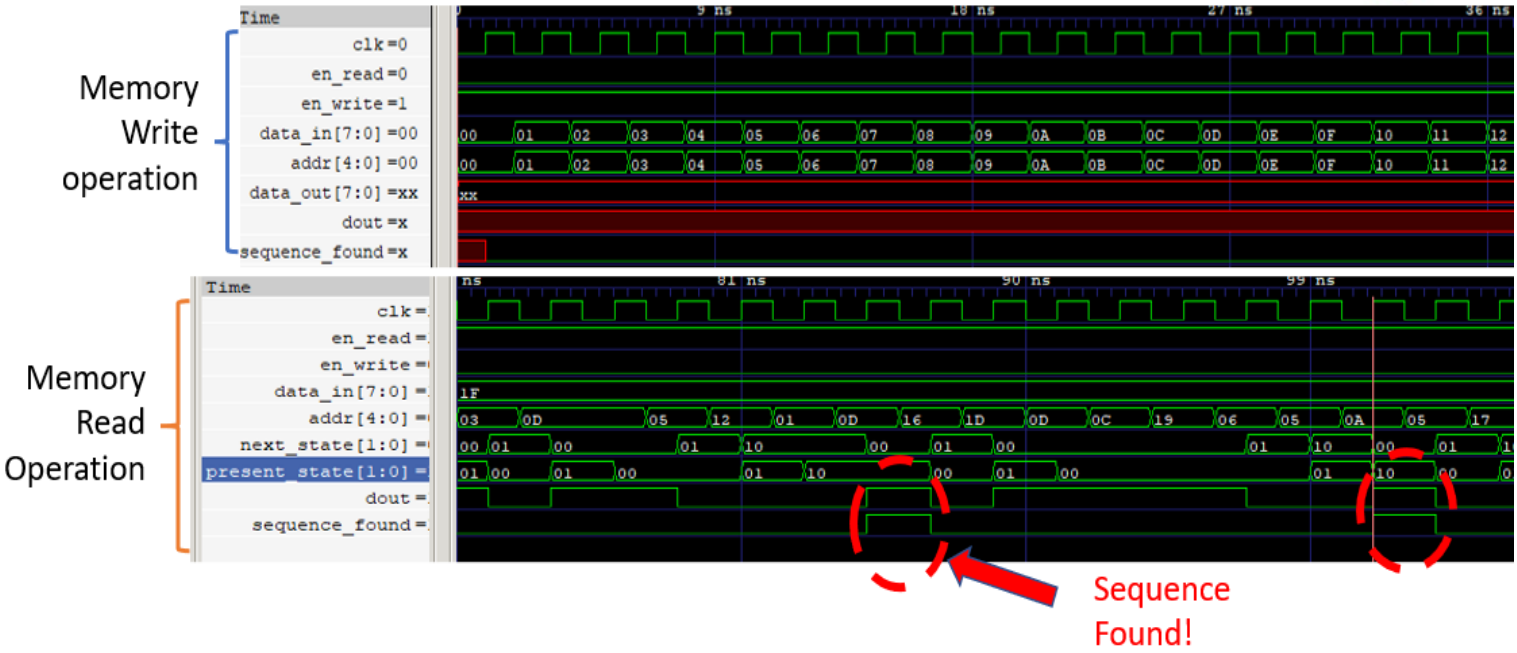


Fig. 4.4 RAM-based FSM (001) simulation Waveform.

5. RoboAnt FSM Implementation

A RoboAnt implementation to walk in the maze wall automatically. The inputs to the FSM come from the ant's two antennae, labeled L and R. An antenna input is 1 if the antenna is touching something, otherwise its 0. The outputs of the FSM control the ant's motion. We can make it step forward by setting the F output to 1, and turn left or right by asserting the TL or TR outputs respectively. If the ant tries to both turn and step forward, the turn happens first. Note: That the ant can turn when its antenna is touching something, but it can't move forward.

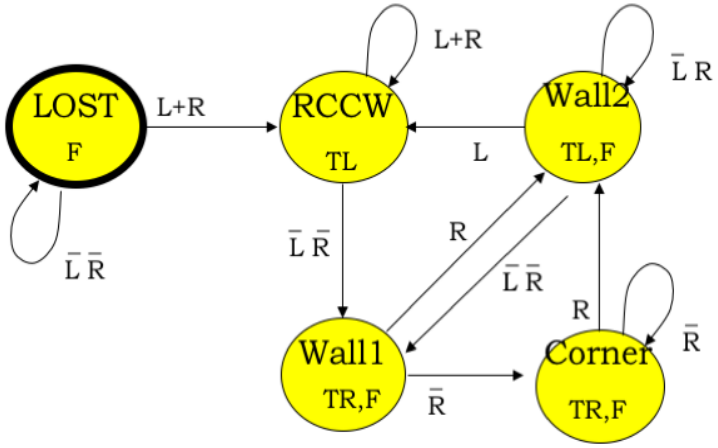


Fig. 4.5 RoboAnt FSM

Let's assume that WALL1 and CORNER are equivalent and ask if they transition to equivalent states for each applicable combination of input values. For these two states, all the transitions depend only on the value of the R input, so we just have to check two cases. If R is 0, both states transition to CORNER. If R is 1, both states transition to WALL2.

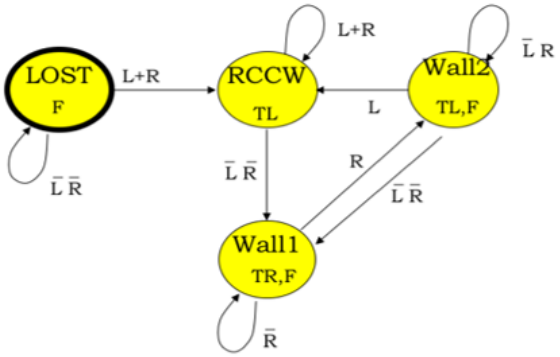


Fig. 4.6 Reduced RoboAnt FSM

RoboAnt Simulation Result

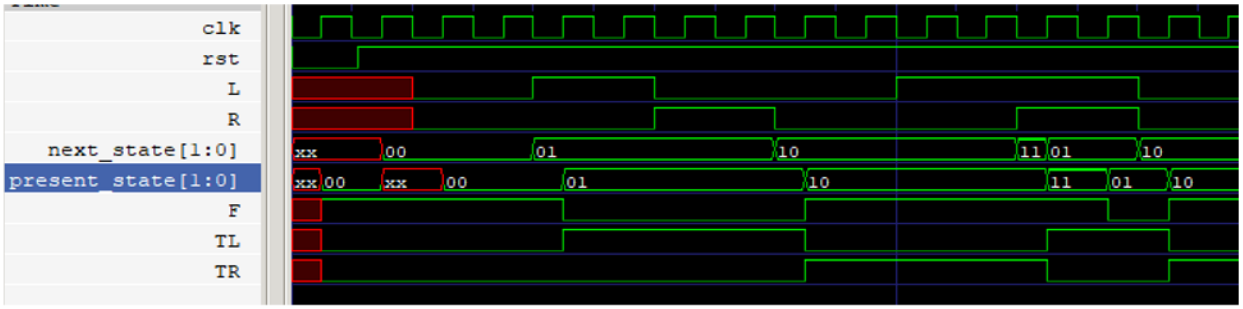


Fig. 4.5 RoboAnt FSM diagram with simulation Waveform.

Implementation of Complex circuits

- Double-Edge detector register.
- Large states FSM.
  - Vending Machine.
  - Tennis Scoreboard.
- Pipelining implementation of Multiplier.

1. Double-Edge detector register

It is a double edge detector circuit which is designed using XOR of ‘en’ and ‘! en’ signal. When the ‘en’ signal changed, the ‘out’ value goes to the don’t care state until the arrival of active clock edge.

Simulation Result Using Modelsim

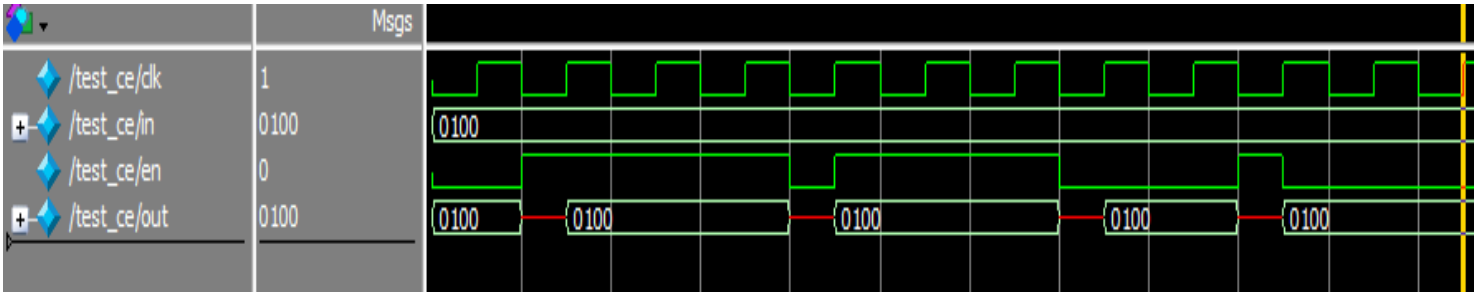


Fig.5.1 Simulation waveform of double-edge detection register.

2. Vending Machine

Vending Machines are used to dispense various products like Coffee, Snacks, and Cold Drink etc. when money is inserted into it. It is an FSM based Vending Machine design.

Operation of Vending Machine

I. When the user puts in money, money counter tells the control unit, the amount of money inserted in the Vending Machine.
II. When the user presses the button to purchase the item that he wants, the control unit allows the functional unit to dispense the product if correct amount is inserted.
III. If there is any change, machine will return it to the user.
IV. The machine will display message “Item Not Available” when the products are not available inside the machine.

PIN Description

- CLK– Vending Machine is synchronous in active clock edge.
- RST – When RST = 0, it remains in IDLE state else not.
- Item – There are two items in this designed machine (named as tea, coffee).
- coin1 – coin1 worth assume to be $5.
- coin2– coin2 worth assume to be $10.
- deliver_tea – If tea delivered then it goes high for one clock period.
- deliver_coffee–If coffee delivered then it goes high for one clock period.
- tea_available – This tells the amount of tea available.
- coffee_available–This tells the amount of coffee available.

Vending Machine FSM

When Item=1, it means customer ordered tea else coffee. If ordered item is available then FSM proceed to next state else remain in IDLE state.

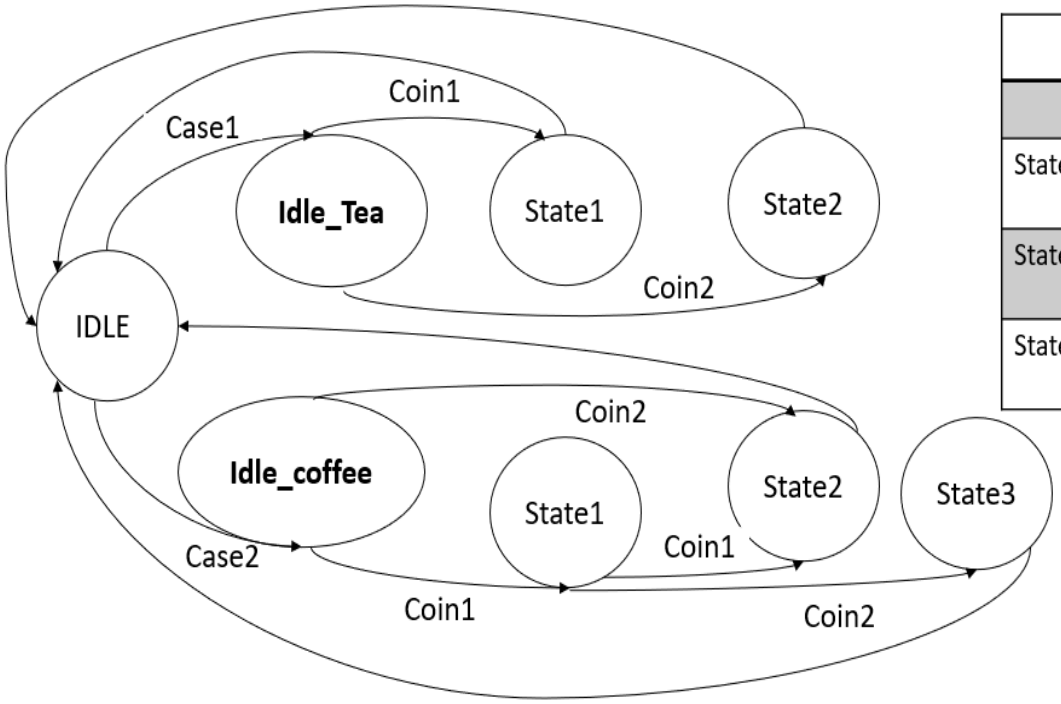
Cases of VM

- When case1 satisfied then next state will be IDLE_TEA state. When coin 1 or coin2 will be inserted then next state either is STATE1 or STATE2 respectively. If coin1 was inserted then tea will be delivered with no change amount. If coin2 was inserted then tea will be delivered with change of coin 1. If coins were not inserted then it remains in IDLE_TEA state.

- When case2 satisfied then next state will be IDLE_COFFEE state. When coin1/coin2 is inserted then the next state will be the STATE1/STATE2. If coin1 was inserted in successively then coffee will be delivered with no change amount. If coin2 was inserted then coffee will be delivered with no change. If first coin1 was inserted then coin2 then coffee will be delivered with change of coin1. If coins were not inserted then it remains in IDLE_COFFEE state.

  Note: We can insert one coin at a time.

Case1 : Item&tea_available
Case2 : ~Item&coffee_available



Output		
	Idle_Tea	Idle_Coffee
State 1	Tea_Deliver = 1, Change = 0	Coffee_Deliver = 0, Change = 0
State 2	Tea_Deliver = 1, Change = 1	Coffee_Deliver = 1, Change = 0
State 3	-	Coffee_Deliver = 1, Change = 1

Fig. 5.2 Vending Machine FSM with Truth Table.

Simulation Result

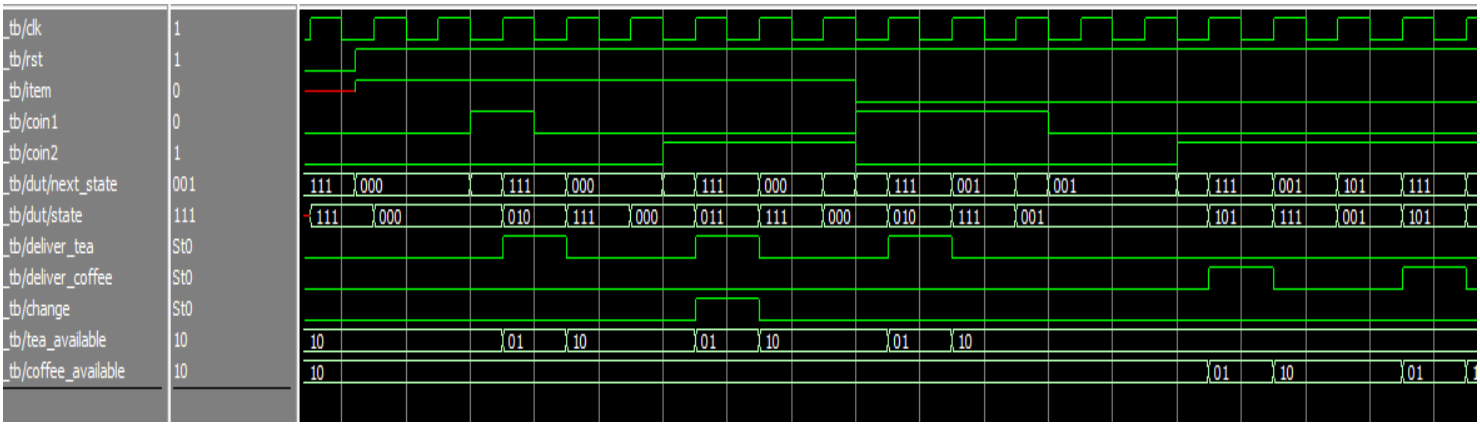


Fig.5.3 Vending Machine simulation waveform.

3. Tennis Scoreboard

Tennis is a very popular sport that can be played individually against a single opponent (singles) or between two teams of two players each (doubles). Each player uses a tennis racket that is strung with cord to strike a hollow rubber ball covered with felt over or around a net and into the opponent's court. It has certain rules to calculate the scores of both the playing team and evaluate a winner.

Tennis rules with FSM description

To win, you must score at least 4 points and win by at least 2. In practice, we used labels like "love" (0 points), "15" (1 point), "30" (2 points), "40" (3 points), "deuce" (3 or more points each, and the players are tied), "all" (players are tied) instead of simply tracking points as numbers.

The game begins in the left-most (unlabeled) state, and then each time either player 1 (red) or player 2 (blue) scores. In each state, player 1's score is first followed by player 2's. For example, "40 30" means player 1 has scored 3 points and player 2 has scored 2 and "15 all" means both players have scored once. "adv2" means player 2 is ahead by 1 point and will win if she/he scores again.
There are only 20 states, and there are cycles which means a tennis game can in fact go on indefinitely, if the players pass back and forth through the "deuce" state.

Once enough points (4 or more) have been scored by either player, their absolute scores no longer matter. All that matters is the relative score: whether player 1 is ahead by 1, equal, or behind by 1. For example, we don't care if the score is 197 to 196 or 6 to 5: they are the same thing.
The FSM track the absolute scores to ensure that at least 4 points were scored by the winner. The original FSM has 20-states, the crossover between these two

phases was what would have been "40 40" (each player scored 3 points). But in the minimal machine, the crossover became "30 30" (each player scored 2 points), which is safe since each player must still "win by 2" so if player 1 scores 2 points from "30 30", that means player 1 scored 4 points overall.

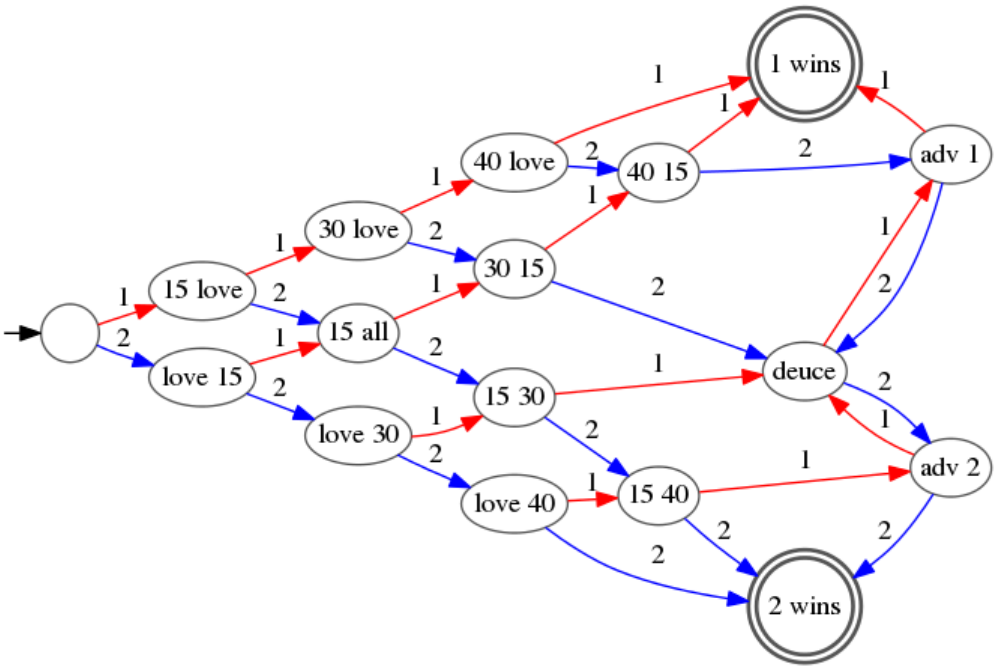


Fig.5.4 Tennis scoreboard reduced FSM.

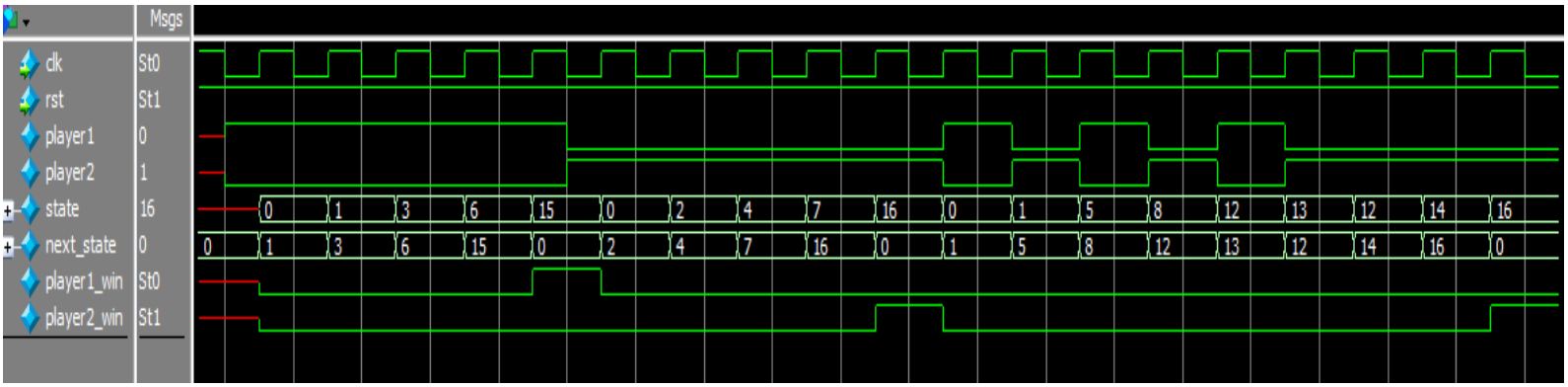


Fig.5.5 Tennis scoreboard simulation waveform.

Pipelining Concept

Pipelining is a process of arrangement of hardware elements of the CPU such that its overall performance is increased. Simultaneous execution of more than one instruction takes place in a pipelined processor.
Let us see a real-life example that works on the concept of pipelined operation. Suppose the work we want to do is laundry.

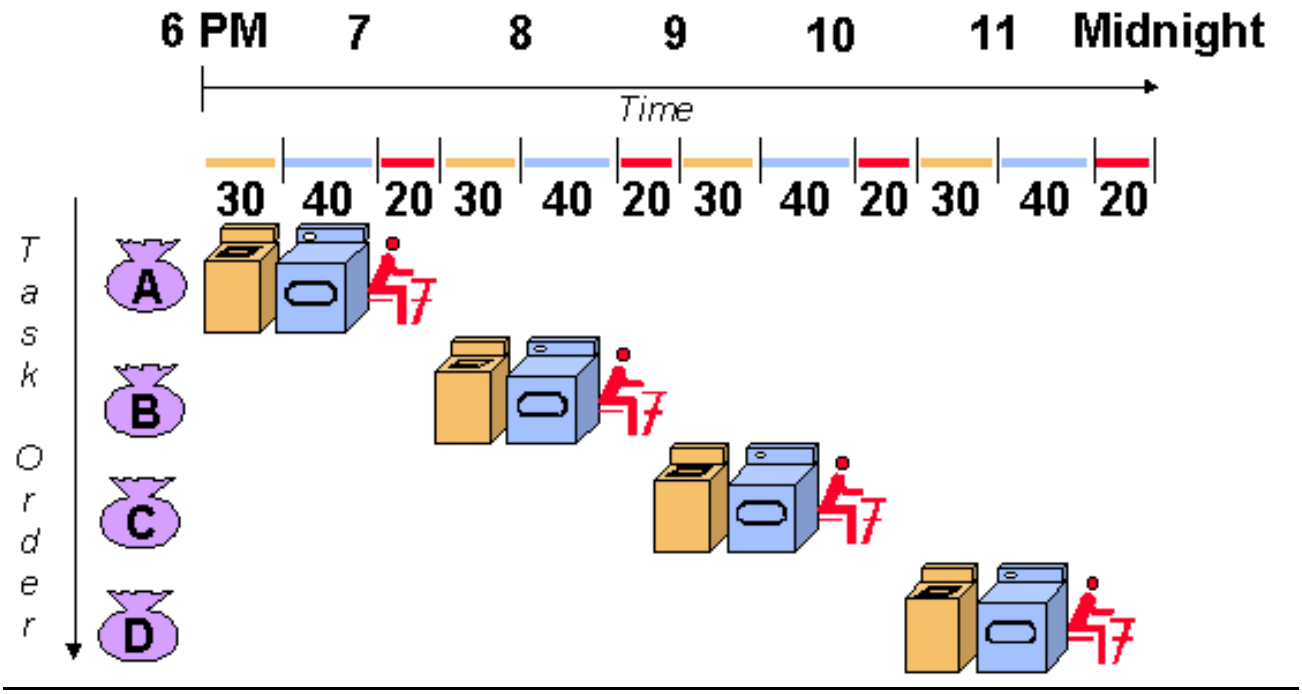


Fig.5.6 Comparison of concept of pipelining with non-pipelining.

There are three main steps in this process:

- (**W**) Wash the clothes in the washing machine (takes 30 minutes)
- (**D**) Dry the clothes in the dryer (takes 40 minutes)
- (**F**) Fold and store the clothes (takes 20 minutes)

We have a pile of clothes to wash, so not all clothes will fit into the washing machine. Thus, we will have to divide the work into several loads. One way to wash all the clothes is sequentially:

W-D-F-W-D-F-W-D-F-W-D-F-W-D-F.....

Suppose that the washer can accommodate 20 items of clothing in one load, and that we have 200 items total to be washed. So, we will have to do 10 loads, and if each one will have a latency of 90 minutes. Thus, it will take 900 minutes to wash all the clothes. This works out to an average of 0.22 items of clothing per minute, or 13 items per hour.
Observe that, while the dryer is drying clothes, the washer is idle. Similarly, while we are folding the laundry, both the dryer and washer are idle. Right after we wash the first load, we place it in the dryer and can immediately start another load. When the dryer is finished, we can immediately take out the clothes, place them on the folding table, and then put the (now finished) load from the washer into the dryer. The process looks like this:

```
W-W-W-W-W-W
  D-D-D-D-D-D
    F-F-F-F-F-F
```

The washing, drying, and folding proceed in parallel. The longest phase is drying at 40 minutes, and we will have to dry 10 loads, so the drying time will be 400 minutes. We have to add the latency for the first wash (30 minutes) and the last fold (20 minutes), during which no drying is occurring, which accounts for another 50 minutes. So, the total time is 450 minutes, or an average of 0.44 items of clothing per minute, or 26.67 items per hour. This rate represents a speedup of almost a factor of two over the previous case.

Note that we haven't actually speed up the laundering of a single item it still takes the same amount of time for a single shirt to enter and leave the system.
This system is a three-stage pipeline. Pipelining is a very old trick to increase performance for tasks that can be divided into independent stages. It's known in manufacturing as using an assembly line.

Simple Multiplier using non-pipeline and pipelining concept

From the below waveforms of Multiplier using pipelining and non-pipelining concept we found as we applied inputs, we get the output (assuming circuit delay to be zero). Whereas in the pipelining structure, we get the output of first couple of input after the delay of 4 clock cycle. This delay is taken by internal registers of the circuit to store the result.

Multiplier using non- Pipelining concept

Multiplier using Pipelining concept

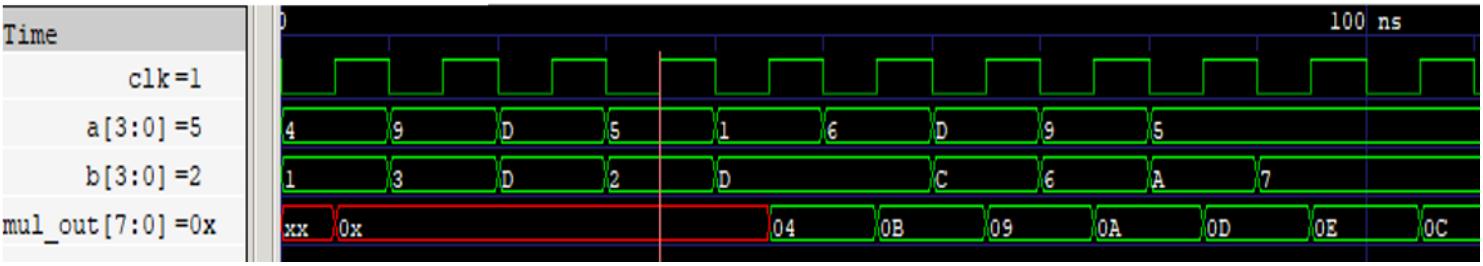


Fig.5.6 Simulation waveform of pipelining and non-pipelining multiplier.

Project-1
APB PROTOCOL

Advanced Peripheral Bus (APB) is the part of Advanced Microcontroller Bus Architecture (AMBA) family protocols. It is a low-cost interface and it is optimized for minimal power consumption and reduced interface complexity. Unlike AHB, it is a non-Pipelined protocol, used to connect low-bandwidth peripherals. Mostly, used to connect the external peripheral to the SOC (Like Timer, I/O peripherals). In APB, every transfer takes at least two clock cycles (SETUP Cycle and ACCESS Cycle) to complete. It can also interface with AHB and ASB protocols using the bridges in between.

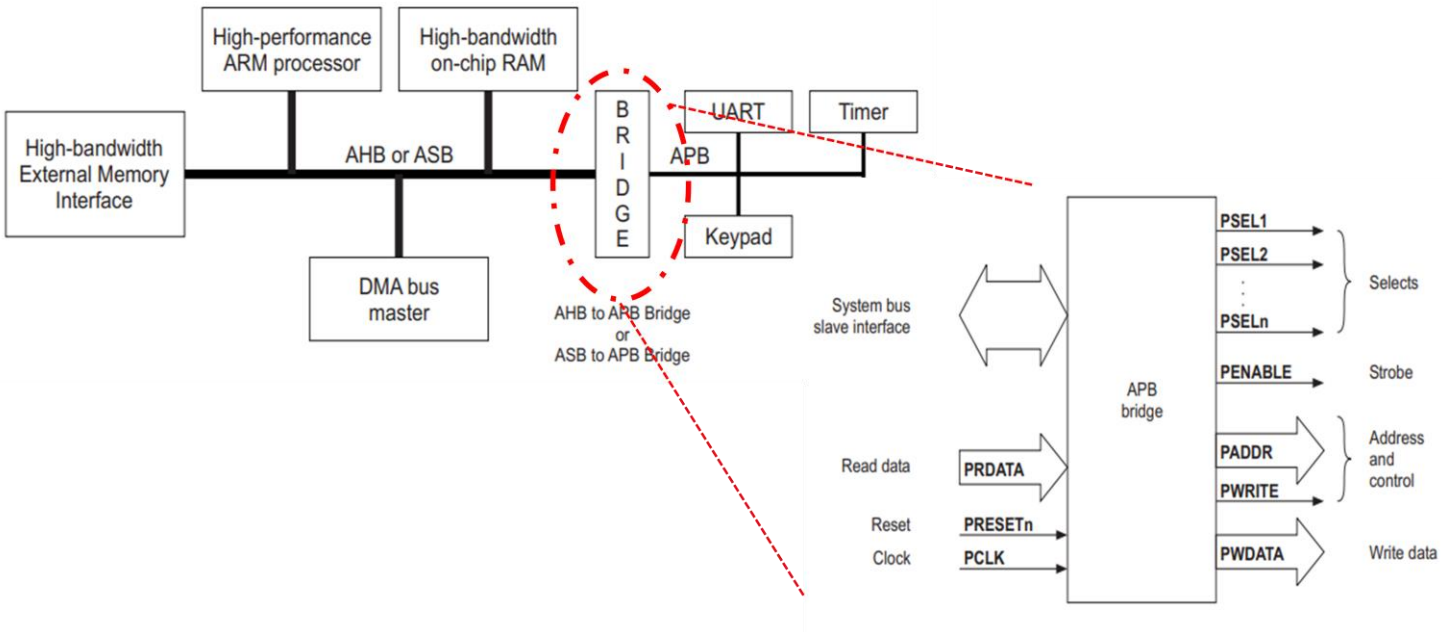


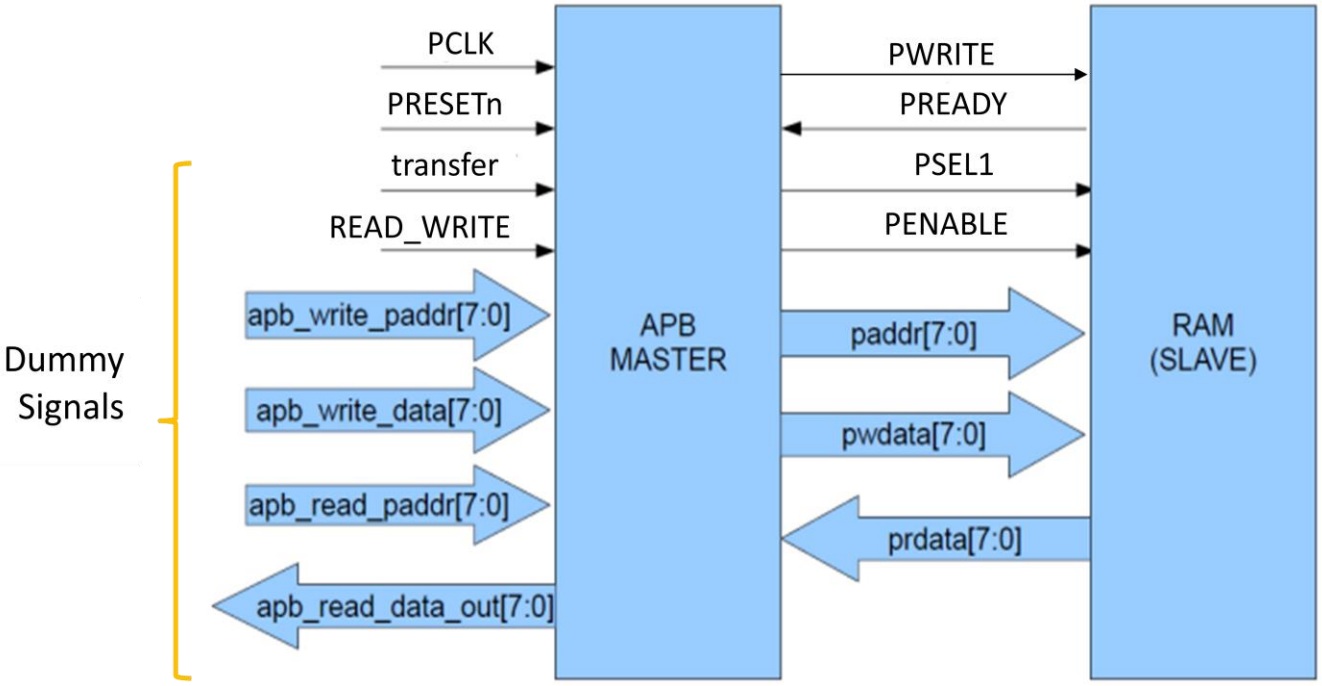
Fig. 6.1 AMBA Bus Architecture.

APB Specification

1. Parallel bus operation.
2. All the data will be captured at rising edge clock.
3. Signal priority: 1. PRESET (active low) 2. PSEL (active high) 3. PENABLE (active high) 4. PREADY (active high) 5. PWRITE
4. Data and address width 32 bit.
5. PWRITE=1 indicates write PWDATA to slave. PWRITE=0 indicates read PRDATA from slave.
6. Before PENABLE is asserted, address and data must be stable and stored. Any change in data or address will result in an error.

7. Start of data transmission is indicated when PENABLE changes from low to high. End of transmission is indicated by PREADY changes from high to low.
8. PSEL and PENABLE can never be asserted at a same clock edge.
9. Completion of data transmission can be seen after one clock.

Fig. 6.2 Bridge Interface with slave



Operation Of APB

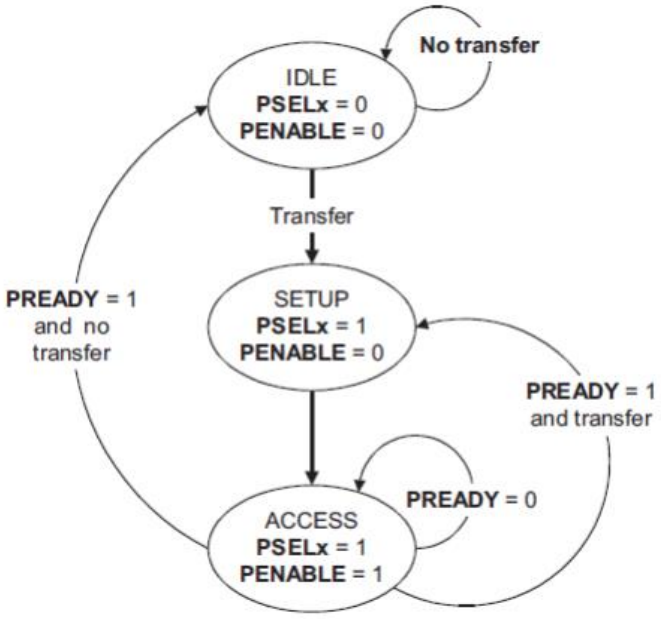


Fig. 6.3 APB Operation states.

- IDLE: The APB is in idle state when reset is changed from 0 to 1. In Idle state PSEL and PENABLE are asserted to 0. As long as Transfer signal is 0 APB remains in IDLE. If Transfer is 1 state changes to SETUP.
- SETUP: Once in setup state APB checks if PREADY is 0 and Transfer is 1, if the condition is true then PSEL is asserted to 1 and PADDR and PDATA are stored, these have to be constant till ACCESS. If transfer is made 0 at any time state goes to IDLE.
- ACCESS: In ACCESS state APB checks if PREADY and transfer to be 1, if true PENABLE is asserted. Once asserted APB checks if address and data are unchanged from setup state, if unchanged transfer takes place. If address or data is changed the it goes back to SETUP state.

PIN Description:

SIGNAL	SOURCE	Description	WIDTH(Bit)
Transfer	System Bus	APB enable signal. If high APB is activated else APB is disabled	1
PCLK	Clock Source	All APB functionality occurs at rising edge.	1
PRESETn	System Bus	An active low signal.	1
PADDR	APB bridge	The APB address bus can be up to 32 bits.	8
PSEL1	APB bridge	There is a PSEL for each slave. It's an active high signal.	1
PENABLE	APB bridge	It indicates the 2nd cycle of a data transfer. It's an active high signal.	1
PWRITE	APB bridge	Indicates the data transfer direction. PWRITE=1 indicates APB write access(Master to slave) PWRITE=0 indicates APB read access(Slave to master)	1
PREADY	Slave Interface	This is an input from Slave. It is used to enter access state.	1
PSLVERR	Slave Interface	This indicates a transfer failure by the slave.	1
PRDATA	Slave Interface	Read Data. The selected slave drives this bus during read operation	8
PWDATA	Slave Interface	Write data. This bus is driven by the peripheral bus bridge unit during write cycles when PWRITE is high.	8

Simulation result of Bridge/APB Master

Memory Write Operation

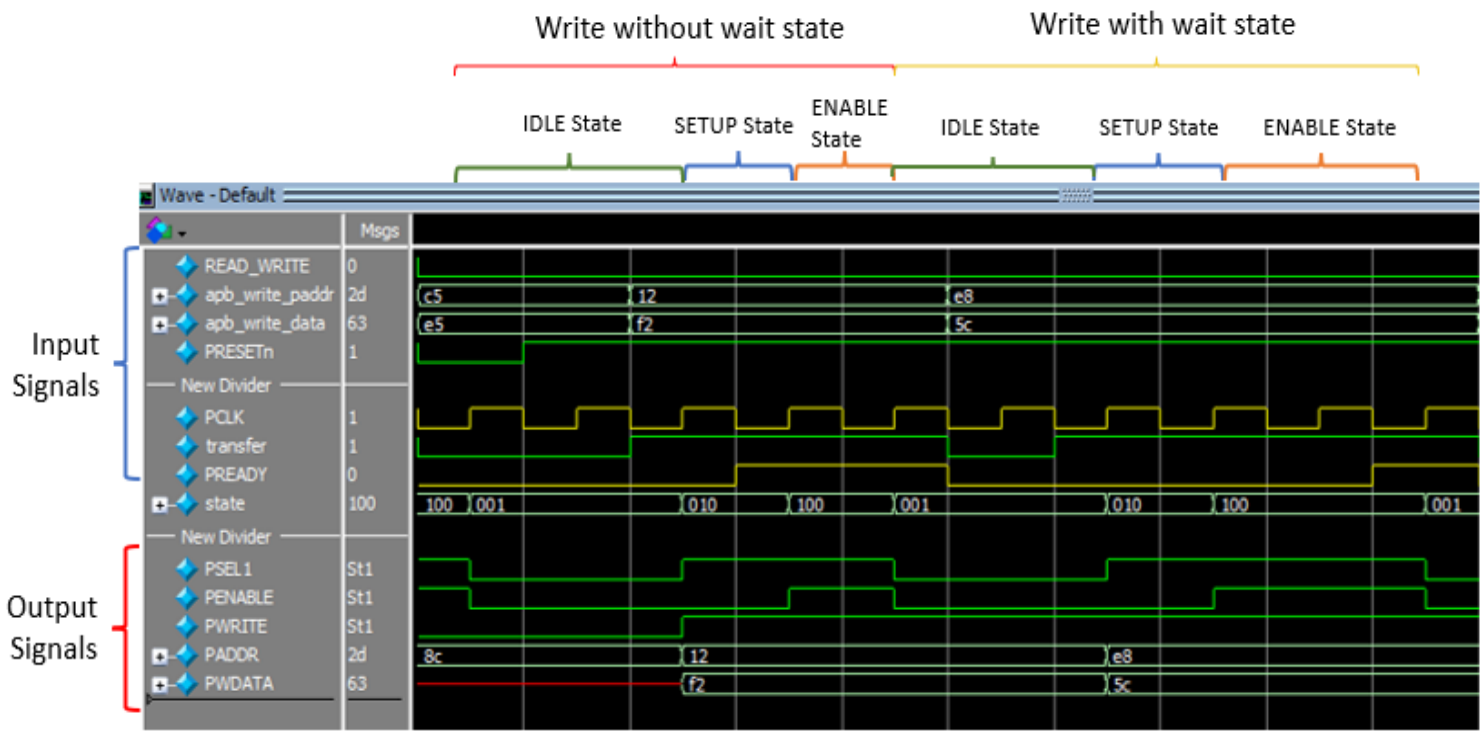


Fig. 6.4 Simulation Result of Memory Write Operation.

- At T1 time period the Address and Data will be available in the bus. The bus checks the status of PWRITE and PSEL. For write operation both need to be high.
- After checking the PWRITE and PSEL signal the bus checks the status of PENABLE and PREADY. Between T1 and T2 the bus stores the Address and Data. If PENABLE and PREADY are high then the write transfer can take place in the next clock T3.
- All the control signals and Address and Data must be constant throughout, till T3

Memory Read Operation

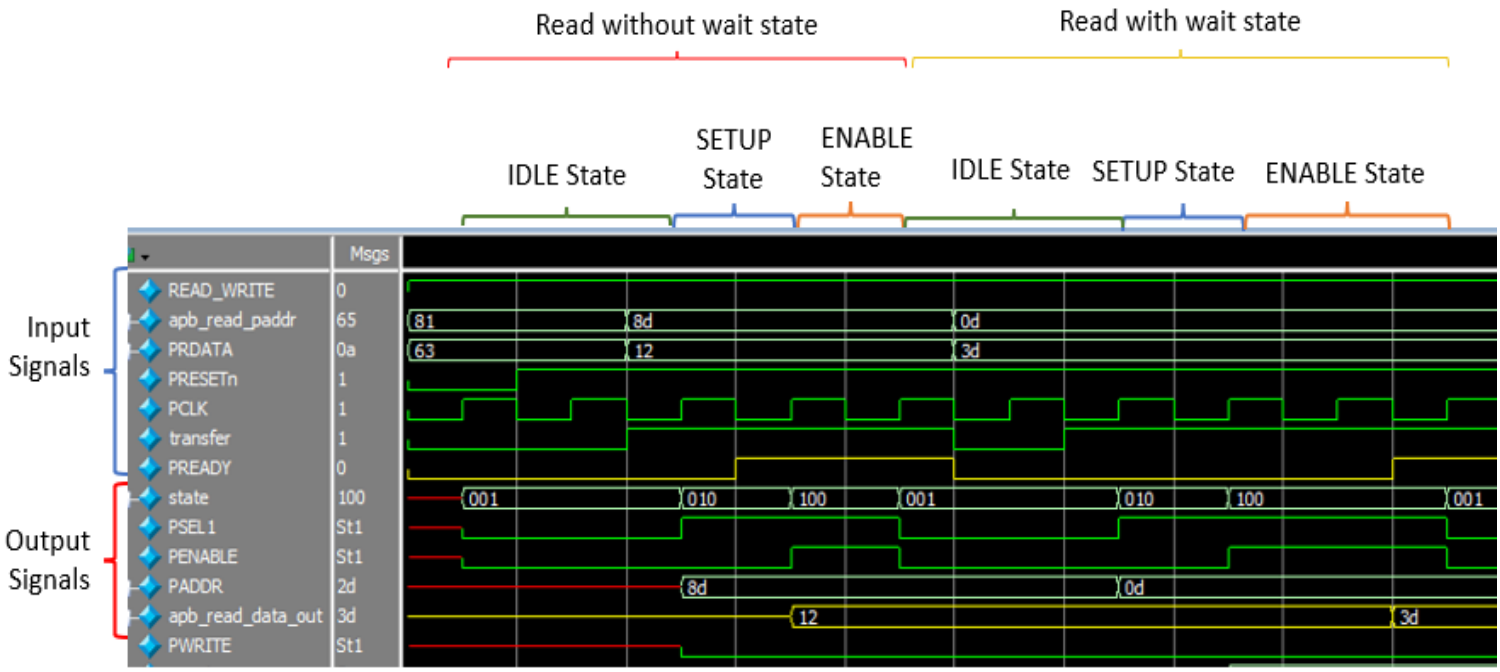
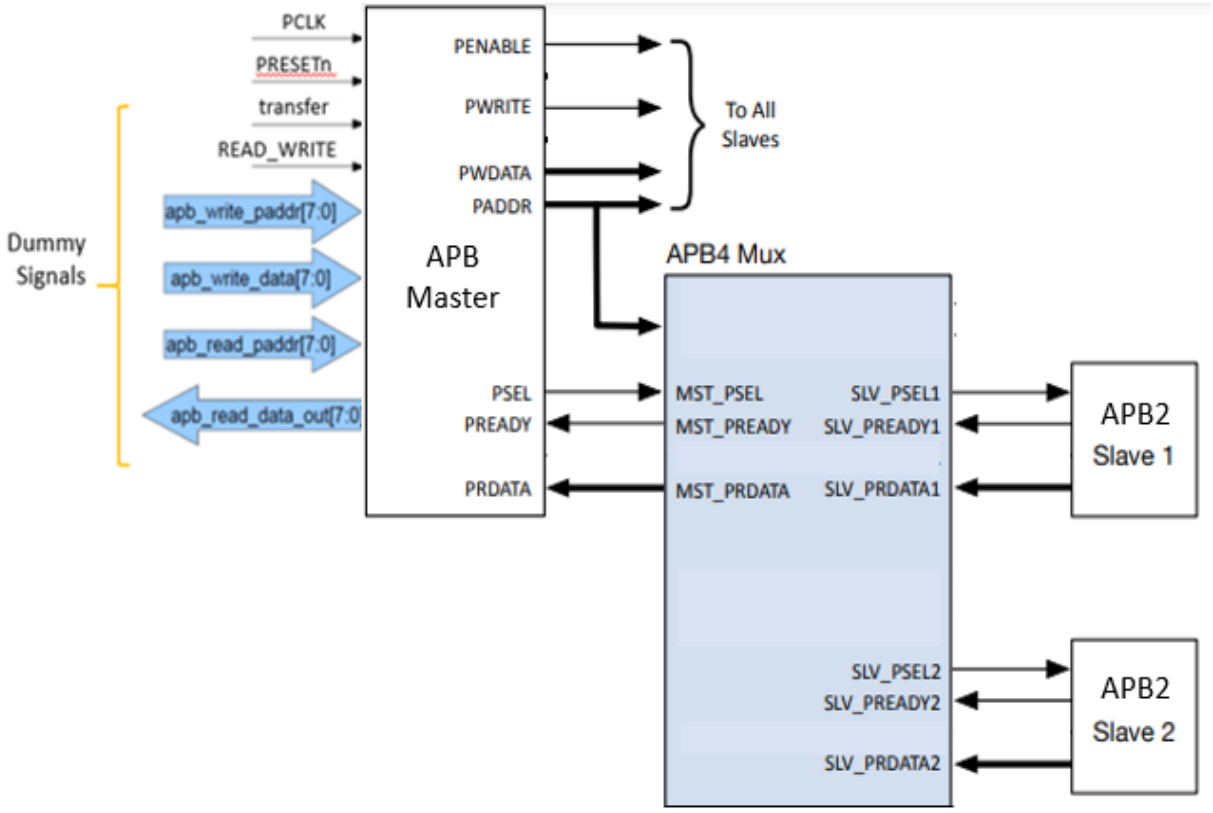


Fig. 6.5 Simulation Result of Memory Read Operation.

- At T1 time period the Address will be available in the bus. The the bus checks the status of PWRITE and PSEL. For read operation PSEL needs to be high and PWRITE needs to be low. The Data that appears will not be considered.
- After checking the PWRITE and PSEL signal the bus checks the status of PENABLE and PREADY. Between T1 and T2 the bus stores the Address and again Data is not considered. If PENABLE and PREADY are high the slave must provide before the start of T3.
- All the control signals and Address and Data must be constant throughout, till T3.

APB Interface with Two Slaves

In this Address is 33 bits long, where MSB of address if use as a select bit of MUX to select a particular slave.



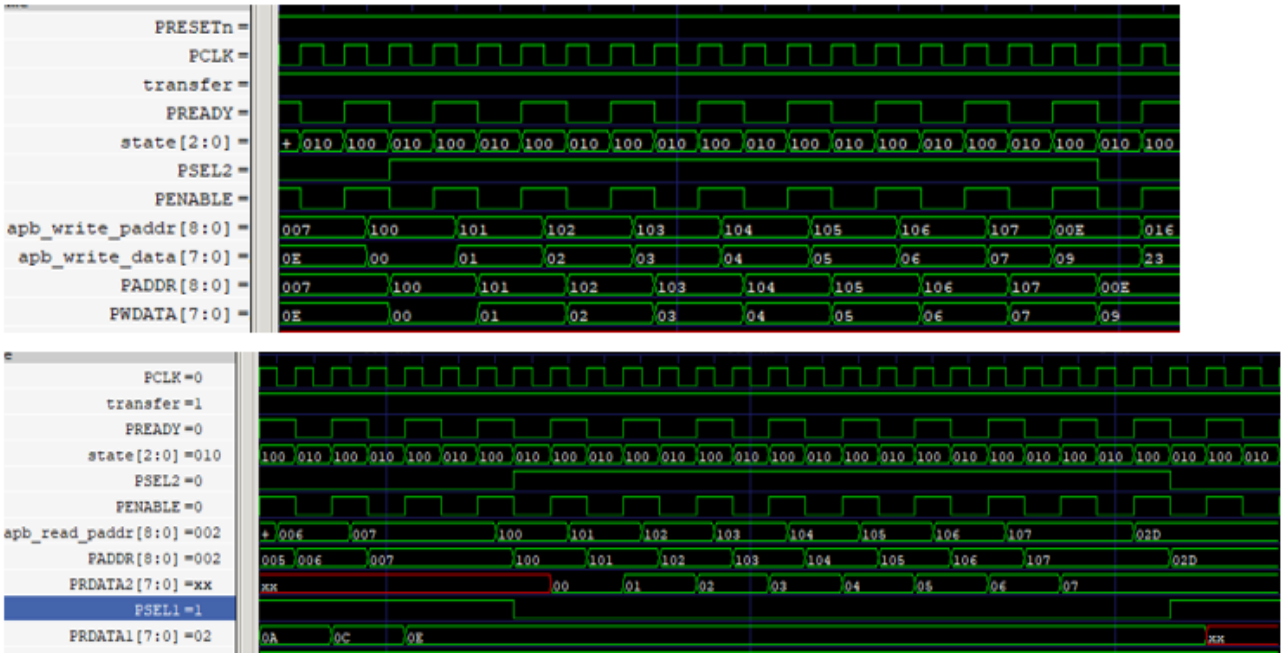
Simulation Result

Fig. 6.7 APB Read-Write Slave-1

Write Operation

Read Operation

Fig. 6.8 APB Read-Write Slave-2



Write Operation

Read Operation

Project 2
RISC-V Processor

Introduction

The Brain of most electronic equipment is a microprocessor. To understand the entire computing and functioning in microprocessors, it is important to understand the basics of microprocessor design.

Microprocessors are instruction set processors (ISPs). An ISP executes instructions from a predefined instruction set. All the programs that run on a microprocessor are encoded in that instruction set. An ISA serves as an interface between software and hardware, or between programs and processors. In terms of processor design methodology, an ISA is the specification of a design while a microprocessor or ISP is the implementation of a design. As with all forms of engineering design, microprocessor design is inherently a creative process that involves subtle tradeoffs and requires good intuition and clever insights.

Processor Design addresses the design of different types of embedded, firmware-programmable computation engines. Because the design and customization of embedded processors has become a mainstream task in the development of complex SoCs (Systems-on-Chip), ASIC. Processor Design provides insight into several different flavors of processor architectures and their design, software tool generation, implementation, and verification.

The mode of operation of any processor is the execution of lists of instructions. Instructions typically include those to compute or manipulate data values using registers, change or retrieve values in read/write memory, perform relational tests between data values and to control program flow.

The 32-bit Berkeley RISC I and RISC II architecture and the first chips were mostly designed by a series of students as part of a four-quarter sequence of graduate courses. This design became the basis of the commercial SPARC processor design.

RISC-V is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. Unlike most other ISA designs, the RISC-V ISA is provided under open-source licenses that do not require fees to use. RISC-V ISA is the main interface in a computer because it lies between the hardware and the software. Software dependency of RISC-V makes it more usable according to the need of user.

Statement of Problem

*"**Implementation** of a RISC-V instruction set based multi- bank architecture with **non-interlocked pipeline** and **scalable instruction slots**."*

Problem Statement in Detail

RISC-V is a Load-Store architecture, in which the compute block cannot directly fetch data from the data memory. Instead, it does so through the Register File.

Multi- bank architecture means that the data memory has been divided into multiple banks to increase the memory bandwidth thereby making the memory access faster.

Non- interlocked pipeline means that the hardware does not have mechanism to counter pipeline violations. It is up to the programmer to schedule the instructions in such a way that each subsequent instruction is independent of its previous instructions.

Scalable Instruction Slots means that some extra slots have been left unoccupied in the instructions to keep scope open for future expansion of the instruction set.

Traditional RISC-V Details

RISC-V (pronounced "risk-five") is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. Unlike most other ISA designs, the RISC-V ISA is provided under open-source licenses that do not require fees to use.

Figure 1 RISC-V Logo

Several companies are offering or have announced RISC-V hardware, open-source operating systems with RISC-V support are available and the instruction set is supported in several popular software tool chains. The major goals of designing this ISA are listed below-:

- A completely open ISA that is freely available to academia and industry.
- Areal ISA suitable for direct native hardware implementation, not just simulation or binary translation.
- An ISA that avoids “over-architecting” for a particular micro architecture style (e.g., micro coded, in-order, decoupled, out-of-order) or implementation technology (e.g., full-custom, ASIC, FPGA), but which allows efficient implementation in any of these.
- An ISA separated into a small base integer ISA, usable by itself as a base for customized accelerators or for educational purposes, and optional standard extensions, to support general purpose software development.
- Support for the revised 2008 IEEE-754 floating-point standard.
- An ISA supporting extensive user-level ISA extensions and specialized variants.
- Both 32-bit and 64-bit address space variants for applications, operating system kernels, and

• Optional variable-length instructions to both expand available instruction encoding space and to support an optional dense instruction encoding for improved performance, static code size, and energy efficiency.

• An ISA that simplifies experiments with new privileged architecture designs.

Figure 2 RISC-V Instruction Pipeliing

Harvard Architecture

The Harvard architecture is a computer architecture with separate storage and signal pathways for instructions and data. It contrasts with the von Neumann architecture, where program instructions and data share the same memory and pathways.

Harvard Architecture consists of Arithmetic Logic Unit, Data Memory, Input/output, Instruction memory, and the Control Unit. It has separate memory for data and instructions. In that way, both instruction and data can be fetched at the same time, thus making it comfortable to the users. In Harvard Architecture, Instructions are generally used in Read-only memory and, Data are used in Read-Write Memory. Harvard Architecture is used with CPU mostly, but it is used with main memory at times as it is a little complex and on the expensive side. Generally, the size of memory for both instructions and data are different in the case of Harvard Architecture.

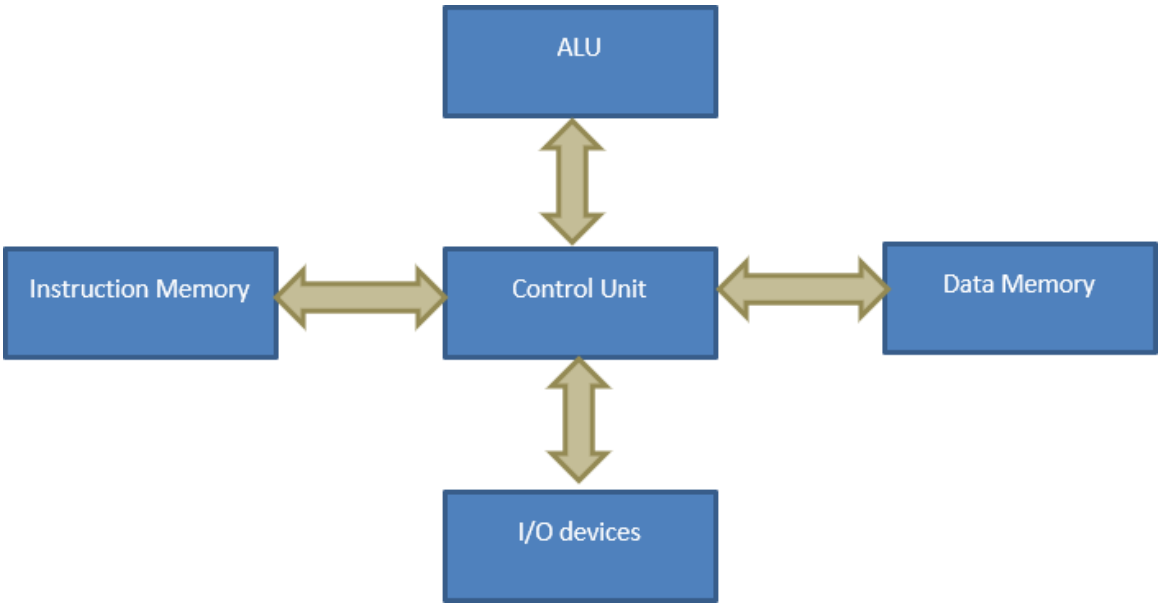


Figure 3 Harvard Architecture

Advantages offered by Harvard Architecture

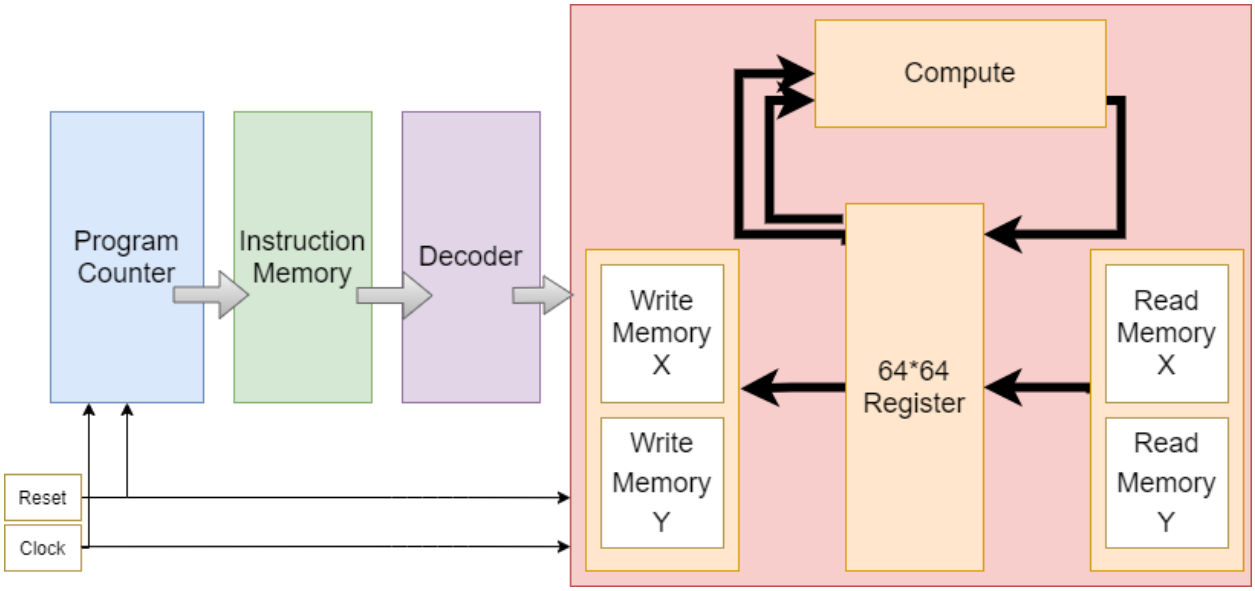
- Since data and instructions are stored in separate buses there are very few chances of corruption.
- Data that uses Read-Only mode and instructions which uses Read-Write mode are operated in the same way. They can also be accessed similarly.
- They generally offer high performance as data and buses are kept in separate memory and travel on different buses.
- Parallel access to data and instructions can be maintained.
- Scheduling would no longer be required as there are separate buses for data and instructions.
- Programmers can design the memory unit according to their requirements.

Limitations

- The un-occupied data memory cannot be used by instructions and the free instruction memory cannot be used by data. Memory dedicated to each unit has to be balanced carefully.
- The program cannot be written by the machine on its own as in Von Neumann Architecture.
- Control Unit takes more time to develop and is on the expensive side.
- There are 2 buses on the architecture. Which in the way means that the motherboard would be more complex, which in turn means that there would be two RAMs and thus tends to have a very complex cache design. That is the reason for it being used mostly inside the CPU and not outside of it.
- Production of computer with 2 buses takes more time to get manufactured and is again on the expensive side like the control unit.
- It has more pins on its IC's. Therefore, it is very difficult to implemented.
- It is not used widely, so the development of it would be on the backward edge.

Load Store Architecture

The L/S architecture describes many of the RISC (reduced instruction set computer) microprocessors (IBM RS/6000, MIPS R-series, Hewlett-Packard's PA-RISC, Sun Micro system SPARC, Motorola M88000). All values must be loaded into registers before an execution can take place. An ALU ADD instruction must have both operands and result specified as registers (three addresses). Thus, an ADD with one operand in memory is not allowed. The purpose of the RISC architecture is to establish regularity of execution and ease of decoding to improve overall performance. RISC architects have tried to reduce the amount of complexity in the instruction set itself and regularize the instruction format to simplify decoding of the instruction. A simpler instruction set with straight forward timing is more readily implemented.


Program Counter
Instruction Memory
Decoder
Compute
Write Memory X
Write Memory Y
64*64 Register
Read Memory X
Read Memory Y
Reset
Clock

Stage wise Description

.

We have followed a bottom to top approach to design the processor. Until now the processor consists of the following layers.

1. **Stage 1: Computational Unit with Register File**

   Computational blocks were added to create a combinational Compute (Computational Unit) block. The 32-bit Compute features a 32-bit Arithmetic, Logical, Bitwise and Shift block. The Computational Unit takes in an opcode and two data inputs and gives results according to the specified opcode.

   Register File is the closest memory to the processor. It is usually like a multi-ported SRAM, made using Flip-Flops. This provides fastest data access to the processor. But as there are multiple buses and high speed of transfer, these are very small in size (64 Registers). In RISC V the Register file is the intermediary between the Computational Unit and the data memory. All data written in or fetched from the memory has to come from or be stored in the Register file. So, at this stage, the design starts to look like a processor with compute happening on data fetched from registers and the result being stored back again.

2. **Stage 2: Compute with Data Memory**

   After the Compute unit has been defined and executed, the next step is to increase the memory capacity of the processor by including data memory. The data memory has been included as two double ported SRAM banks, to increase memory bandwidth and also provide parallel access to two locations at different memory. The data memory and register file circuit is independent of the compute circuit as mentioned above. The total address size for the memory is 32 bits which gives 4 GB of data memory. The details of the Data Memory are explained later on.

3. **Stage 3: Byte Addressable Memory**

   After increasing the memory capacity inside the processor next step is to make the memory byte addressable. Byte Addressable memory refers to architectures where data can be accessed and addressed in units that are narrowed than the bus. In this memory, based on this data storage i.e., Bytewise storage, memory chip configuration is named as Byte Addressable Memory. For e.g.: 128K X 8 chip has 32-bit Address and cell size is 8 bits (1 byte) which means in this chip, data is stored byte by byte.

4. **Stage 4: Instruction Memory:**

   This stage completes the main requirement for the Harvard Architecture. An independent Instruction Memory has been included in the design, which can provide one 32-bit instructions and data at the same time according to the word it is fed with. Therefore, the instruction memory has a 32-bit wide. The Address and data come from an instruction address generator which is a fetch counter for non-jump operations. The Memory output is fed to the Instruction decoder which generates the necessary control signals for the processor to work.

5. **Stage 5: Jumps:**

   This stage completes the necessity for running a c program with the help of processor. For any processor, jumps are very important to run a basic program. Jumps can be of various types and used according to the need. The two broad categories of jumps are Conditional and Unconditional jumps

   Conditional jumps are special type of jumps which depend on conditions of flag registers. The flags are used to determine the status of the result of Computational Unit operation such even, odd, positive, negative etc. If the condition stated by the instruction is true only then the jump takes place.

   Unconditional jumps are jumps which always takes place whenever such jump instruction is fed to instruction memory.

Types of Memories in the Architecture

Memory organization and Hierarchy are a very important part of a processor as this directly affects the computational capacity of the processor. The Trade-off between cost, speed and size must be carefully fine-tuned according to the application while designing the Memory Architecture. The design features five memory banks, one for instruction memory and four for data memory and a Register File. They are described as follows:

1. Register File

Register file is the closest memory to the processor. It is of small size, typically 32 or 64 registers of 64-bits or 32-bits. It can have multiple ports and is typically made up of flip- flops.

The Register file module that has been created in our design consists of three read ports (A, B and C) and two write ports (A and B). Having multiple ports is advantageous for implementing a parallel read-write operation in memory. Our design has so far 64, 32-bit registers.

As our design is based on load store architecture means the data memories and Computational Unit does not interact directly with each other for any kind of operation, there should be a mediator in between them and that mediator is register file which not only provides register for data storing but also interact with data memory and compute unit directly to perform various instructions.

Every compute instruction cannot directly write into the data memory, the instruction can only write into register file and then only data can be moved from register file to data memory. Similarly, the compute unit cannot take input directly from data memory, it can take input only from regfile.

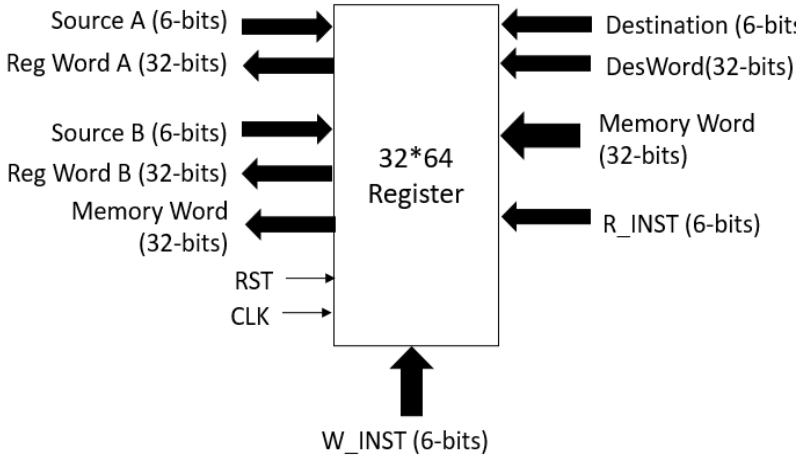


Figure 6 Block Diagram of Register file

2. Data Memory

The data memory that we are using is a single read and single write ported SRAM model. This has a data size of 32-bits and address size is of 32 bits, which seems unusual but has been chosen so to compensate for byte addressing and multiple banks as explained later. This SRAM module has been instantiated four times to make four banks, which increases the capability of memory bandwidth.

We are using multiple single ported banks instead of single multi-ported SRAM, because for same memory size the former requires less area on the chip. For multi-ported SRAM, the area increases quadratically with number of ports more compared to the linear increase with the number of banks.

Memory bandwidth is the rate at which data can be read from or stored into a semiconductor memory by a processor. Memory bandwidth is usually expressed in units of bytes/second. It has been established that by increasing the number of banks and following certain access patterns, we get higher memory bandwidth.

The memory address in the processor is 32-bits, consisting of 2-bit BSEL for byte selection.

The CS used for selecting bank. The BSEL being used for byte selection, implement a concept of byte addressing, which gives the programmer the capability to access 4- bytes, 2-bytes, or 1-byte from the memory according to the application.

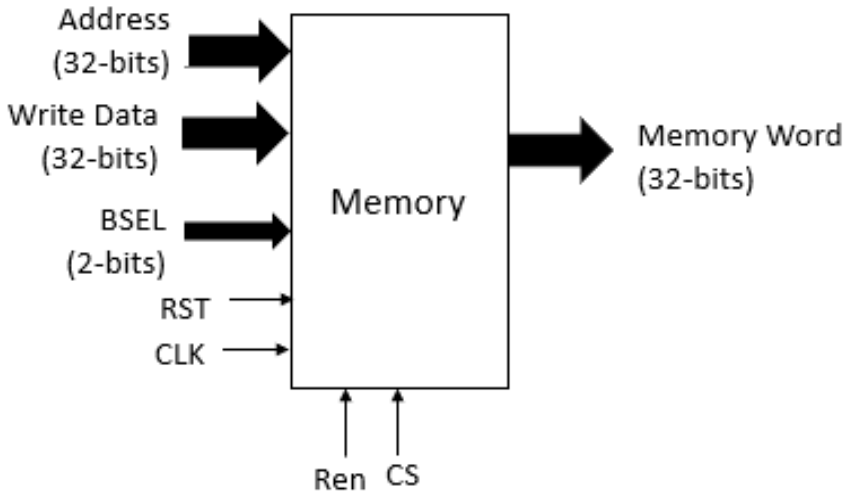


Figure 8 Block Diagram of Data Memory

3. Instruction Memory

Instruction Memory is a defining part of Harvard type architecture. It is generally an SRAM with continuous address being given at its input and it gives the word stored at that address which is fed to the decoder. The address comes from an Instruction address generator, which is normally a fetch counter. The Instruction memory which is implemented has a 32-bit wide word.
The instruction memory which is used in our design works differently in some instructions like jumps.

Conditional jumps

In conditional jumps, the Program counter need to points to the jumped instruction location if the condition of the jump is satisfied, so during the evaluation of the condition the fetch counter needs to be stalled for condition evaluation time.

In **pre modify conditional jump**, the condition of the jump needs to be checked so during that time the fetch counter needs to be stalled for the time of condition checking, thus stalling needs to be done. In condition evaluation time, the two data from register file need to be fetched and then added using Computational Unit, after then the condition will be checked on the result of Computational Unit operation and the fetch counter is loaded with the result of Computational Unit operation

Unconditional Jumps

In **pre modify unconditional jump,** fetch counter needs to be stalled so that the result of Computational Unit operation will be used in fetch counter for jump location.

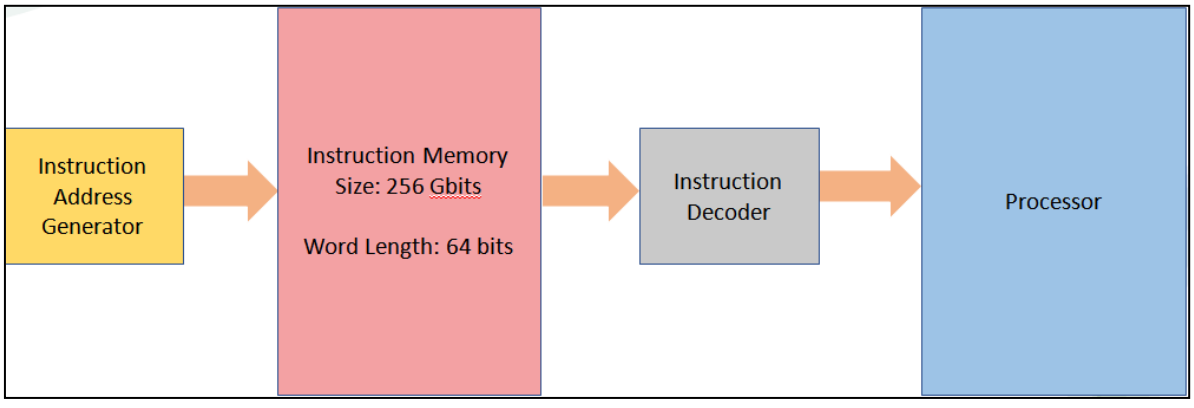


Figure 10 Instruction Memory

Types of Addressing

1. Direct Addressing

In direct addressing mode, address field in the instruction contains the effective address of the operand and no intermediate memory access is required. In our processor the Direct Addressing mode instruction provides the direct memory address (32-bit) and a register-file address (6-bit) to store or provide data. This instruction provides direct memory access so gives the programmer the ability to load data from desired locations from the memory. The main limitation is that it can only address one location per instruction.

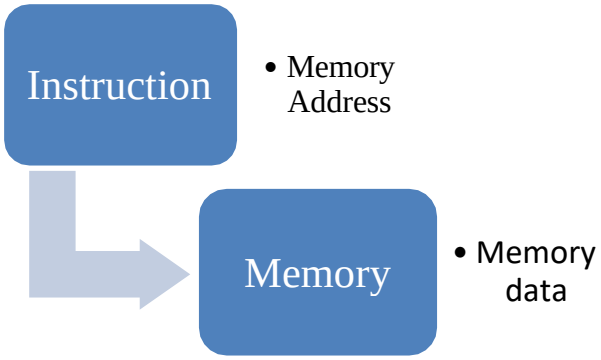


Figure 11 Process to fetch memory data in Direct Addressing Mode

2. Indirect Addressing

In Indirect addressing mode, address field in the instruction contains the memory location or register where effective address of operand is present. It requires two memory accesses for every location that is accessed, one for address and other for data. It is further classified into two categories: Register Indirect, and Memory Indirect.

In our processor we have implemented Register Indirect Addressing Instruction. The instruction provides 4 register addresses, 2 containing addresses and other two for data transfer. This instruction provides the programmer the capability to use the register file as pointer to the memory locations and thus can be used in special methods like matrix multiplication and linked list operations.

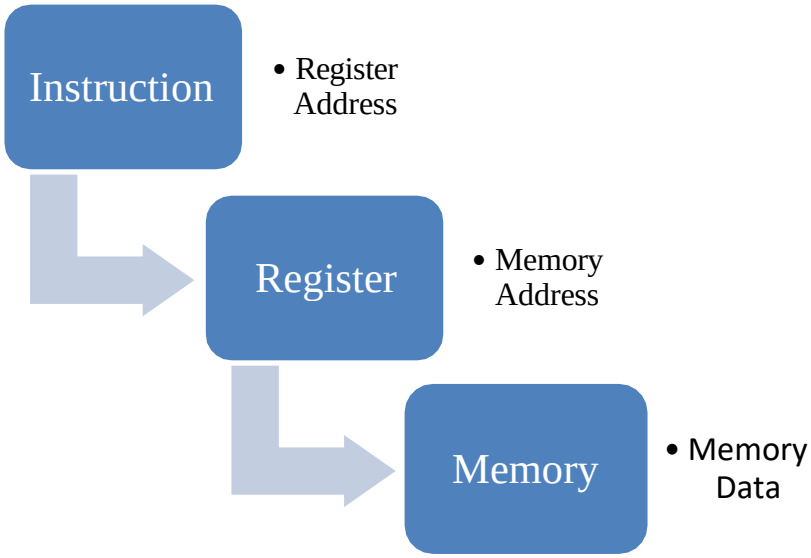


Figure 12 Process to fetch memory data in Indirect Addressing Mode

Computational Unit

Computational Unit is used to perform arithmetic operations (Addition, Subtraction, Multiply, Divide) as well as logical operations (AND, OR, NOT, XOR). It is a fundamental building block of many types of computing circuits, including the central processing unit (CPU) of computers, FPUs, and graphics processing units (GPUs). It is a combinational logic device. It takes input values from register block, in addition to this it is also capable of taking operand values directly from instructions. Result of Computational Unit is stored inside register block. Computational Unit also has status inputs and outputs, which convey information about the current operation between the Computational Unit and external status registers. Computational unit can work in parallel with memory operations.

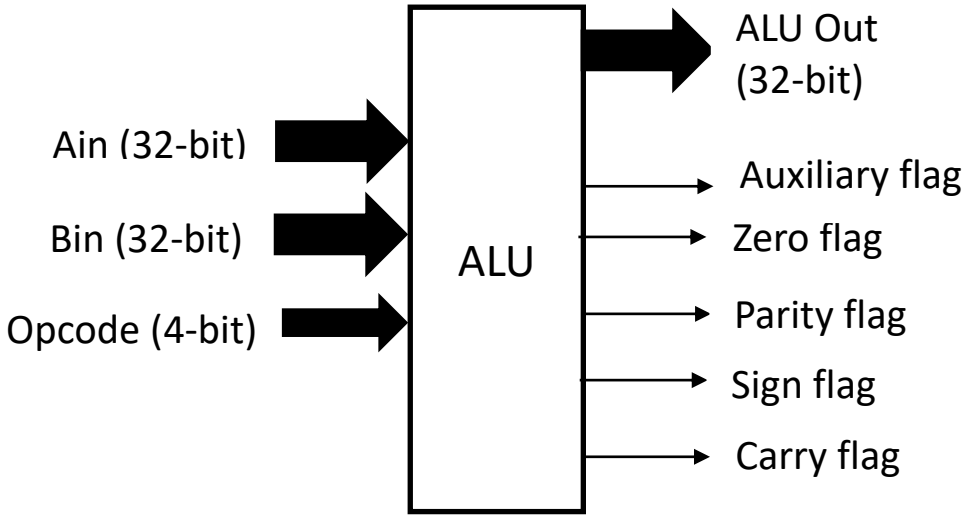


Figure 13 Block Diagram of ALU

1. Data

Computational Unit has three parallel data buses consisting of two input operands (*A* and *B*) and a result output (*Y*). Each data bus is a group of signals which contains 32-bit data width.

2. Opcode

The *Opcode* input is a parallel bus that conveys to the Computational Unit a no operation selection code, which is a numerated value that specifies the desired arithmetic or logic operation to be performed by the Computational Unit. The opcode size (4-bits) determines the maximum number of different operations the Computational Unit can perform.

Opcode	Operation
4’d0	Addition
4’d1	Subtraction
4’d2	Pass Through A
4’d3	Pass Through B
4’d4	Bitwise AND
4’d5	Bitwise OR
4’d6	Bitwise NOT
4’d7	Bitwise XOR
4’d8	Left Shift by B
4’d9	Right Shift by B
4’d10	Arithmetic Right Shift by B
4’d11	Arithmetic Left Shift by B
4’d12	Logical AND
4’d13	Logical OR
4’d14	Logical NAND
4’d15	Logical NOT

Table 2 ALU Operations Opcode

3. Status Flags

In the implemented Computational Unit, we have added 5 status flags as shown in Table. These have been added to provide better computational capability to the programmer.

Status Flag
Zero flag
Parity flag
Carry Flag
Auxiliary carry flag
Overflow flag

Table 3 Status Flag

Control Pin Description

Program Counter	FS – Active high - Conditional jump operation enables. – Active low - Conditional jump operation not get enabled. CON – Active high - Performs conditional jump operation. – Active low – Performs unconditional jump operation. Jump_enable - Active high – Jump operation will be enabled. - Active low – Jump operation will not be enabled.
ALU (32 bit)	Opcode – 4-bits. To choose different ALU operations
Register bank	SRA – 6-bits. Address of source A register SRB – 6-bits. Address of source B register W_inst – 6-bits. Write address R_inst – 6-bits. Read address DR – 6-bits. Destination address
Instruction memory	Addr – It is of 6-bit address, which is obtained from program counter.
Memory block	CS – Selects the memory RE – Active high – Performs read operation - Active low – Performs write operation

Instruction Format

[31:28] B_sel+reg_sel	[27:24] opcode	[23:18] src reg (rs1)	[17:12] src reg (rs2)	[11:6] Dest.reg	[5:2] --	[1:0] function

REG-REG TYPE INSTRUCTION

[31:28] B_sel+reg_sel	[27:24] opcode	[23:18] src reg (rs2)	[17:12] Dest.reg (dr)	[11:2] immediate data	[1:0] function

IMM-REG TYPE INSTRUCTION

[31:28] B_sel+reg_sel	[27:24] opcode	[23:18] src reg (rs1)	[17:12] Dest.reg (dr)	[11:2] immediate data	[1:0] function

REG-IMM TYPE INSTRUCTION

[31:28] B_sel+reg_sel	[27:24] opcode	[11:6] Dest.reg (dr)	[17:10] immediate data	[9:2] immediate data	[1:0] function

REG-REG TYPE INSTRUCTION

[31:28] B_sel+reg_sel	[27] R_W	[26:25] X +Y	[24] IN	[23:18] R_inst	[17:12] W_inst	[11:2] ADDR	[1:0] function

REG-REG TYPE INSTRUCTION

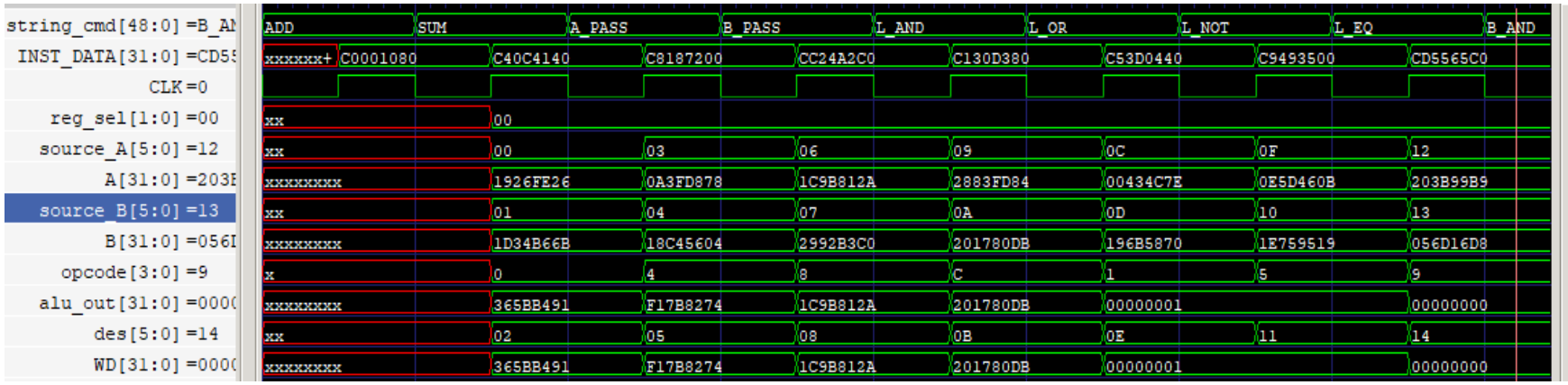
31 28	27 24	23 18		6 5 2 1	0
{BSEL,REG_SEL}	{{X/Y} + R_W + IN}	ADDR		{CON 1 bits + CARRY + PARITY + ZERO}	Funct

JUMP INSTRUCTION

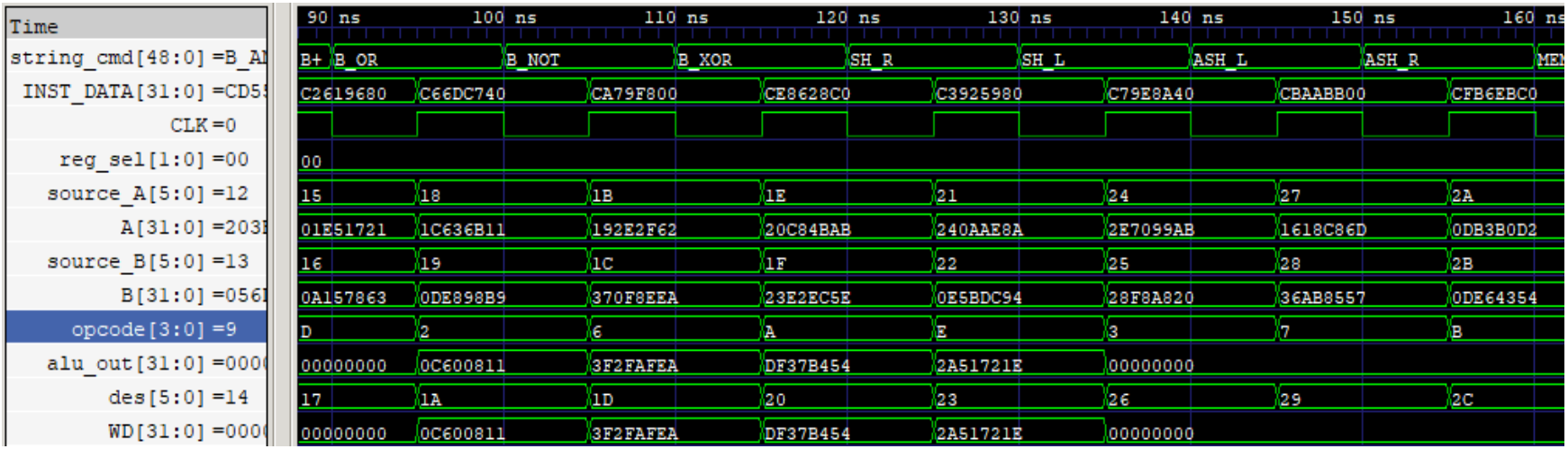
Result Waveform

ALU Operation (Reg – Reg)

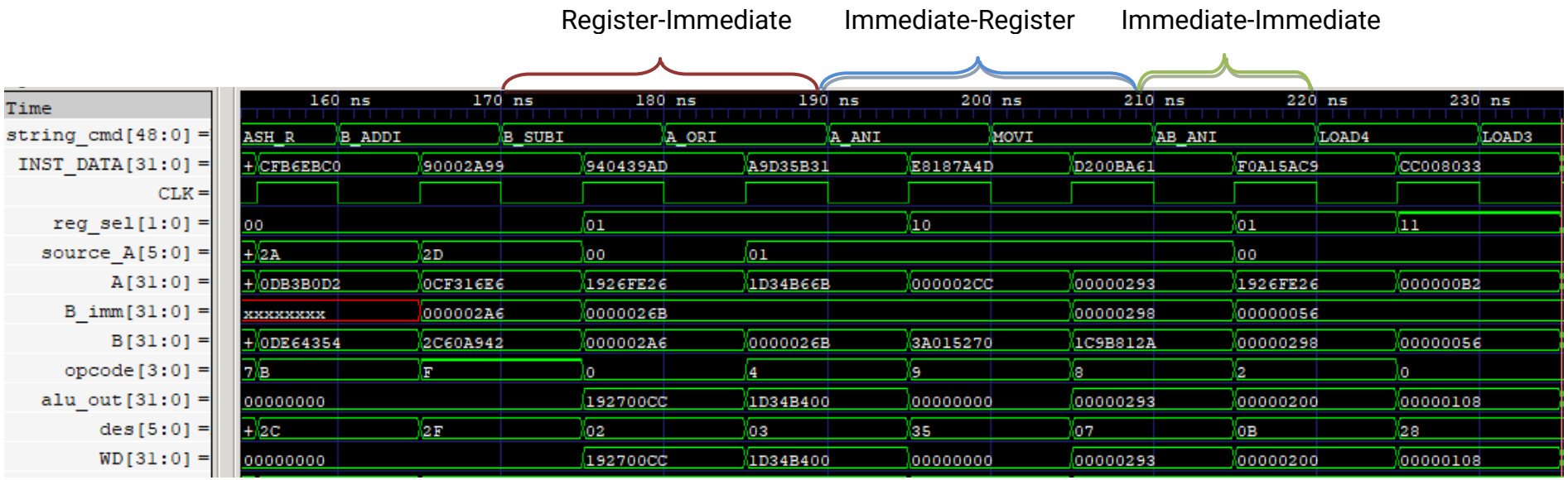
1.



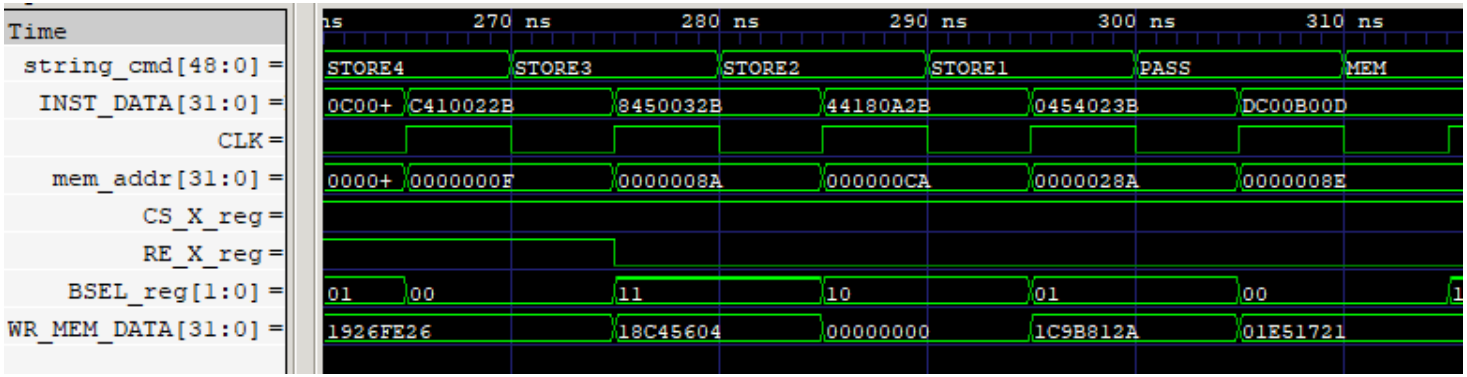
2.



ALU Operation (Reg-IMM/IMM-REG/IMM-IMM)

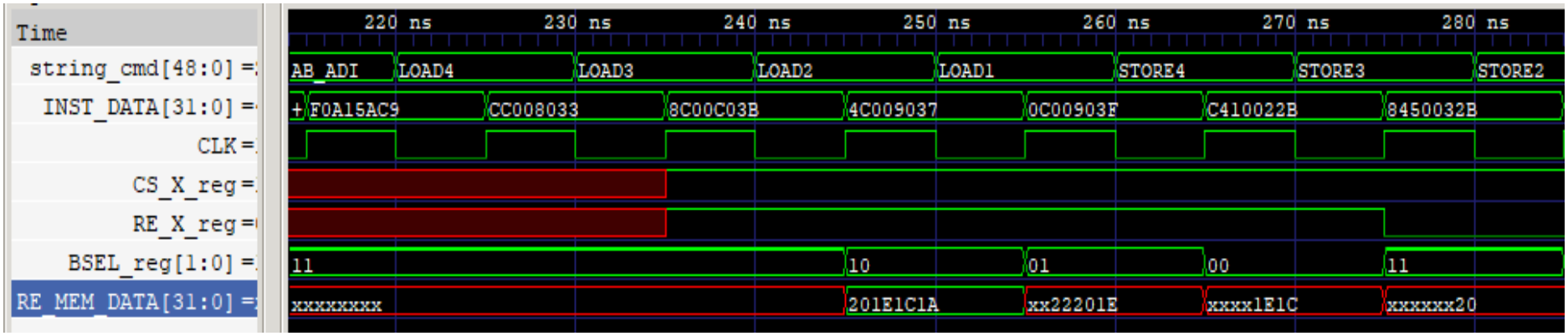


Byte Addressable Write Operation

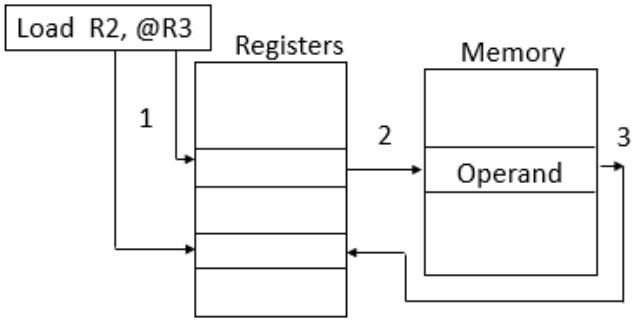


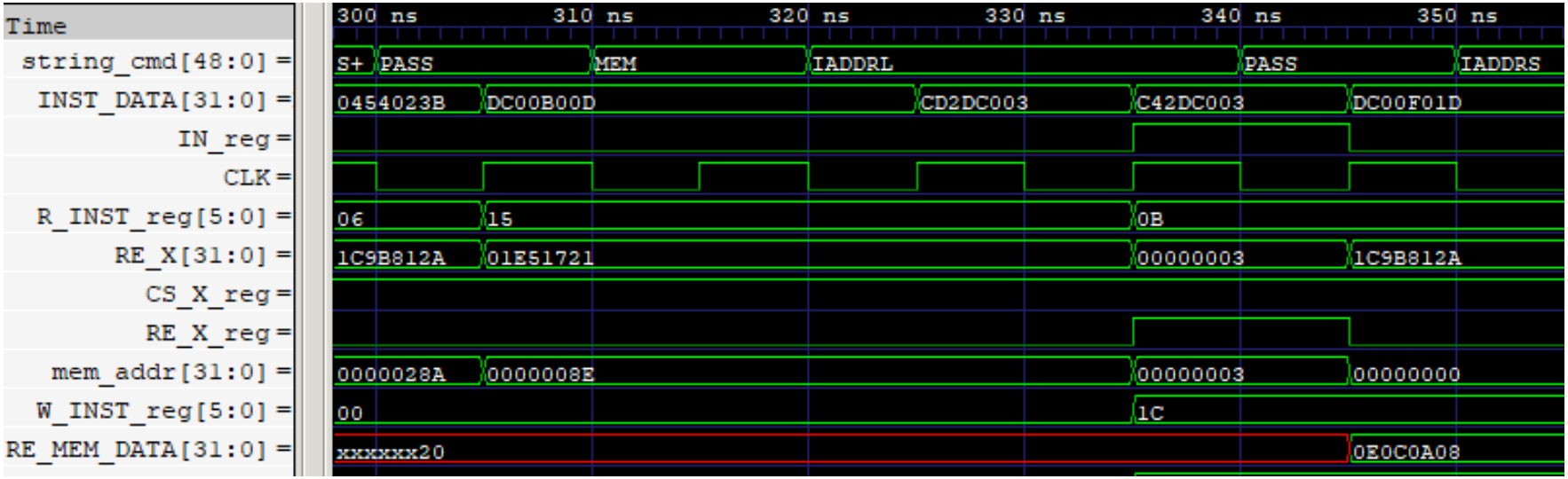
BSEL_reg	Operation
00	1 Byte
01	2 Byte
10	3 Byte
11	4 Byte

Byte Addressable Read Operation

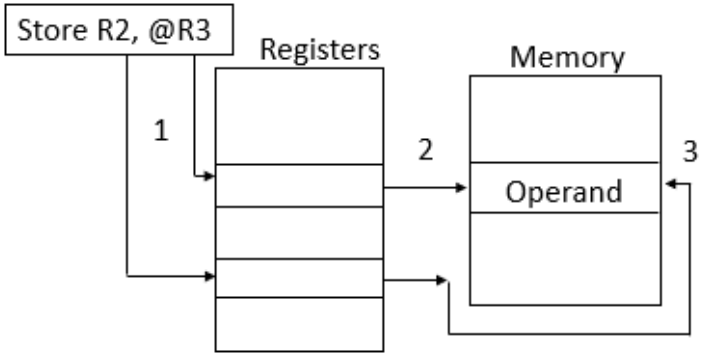


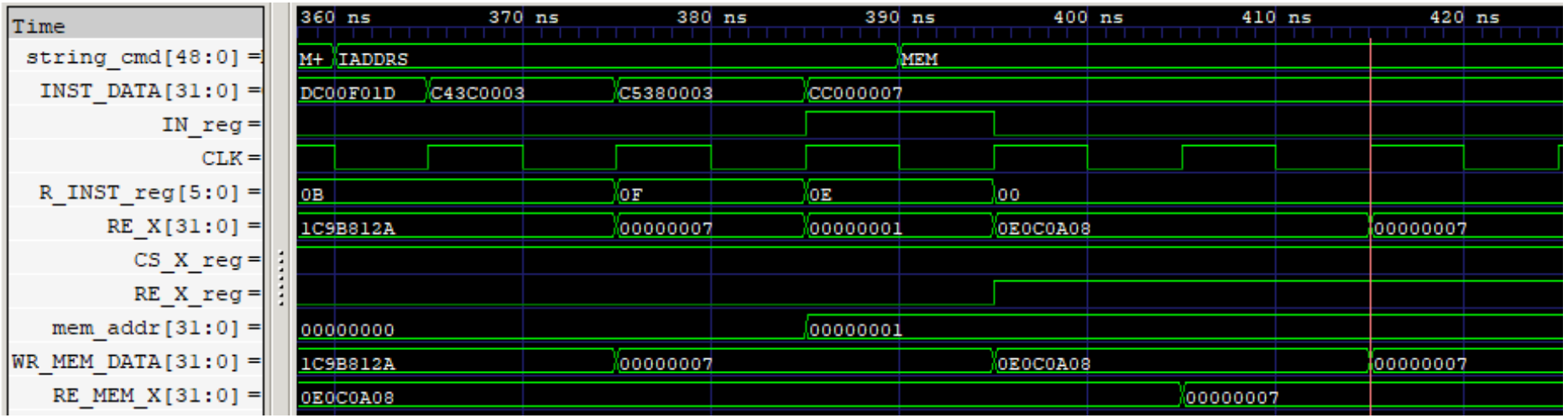
Load Operation
(Indirect Addressing Mode)



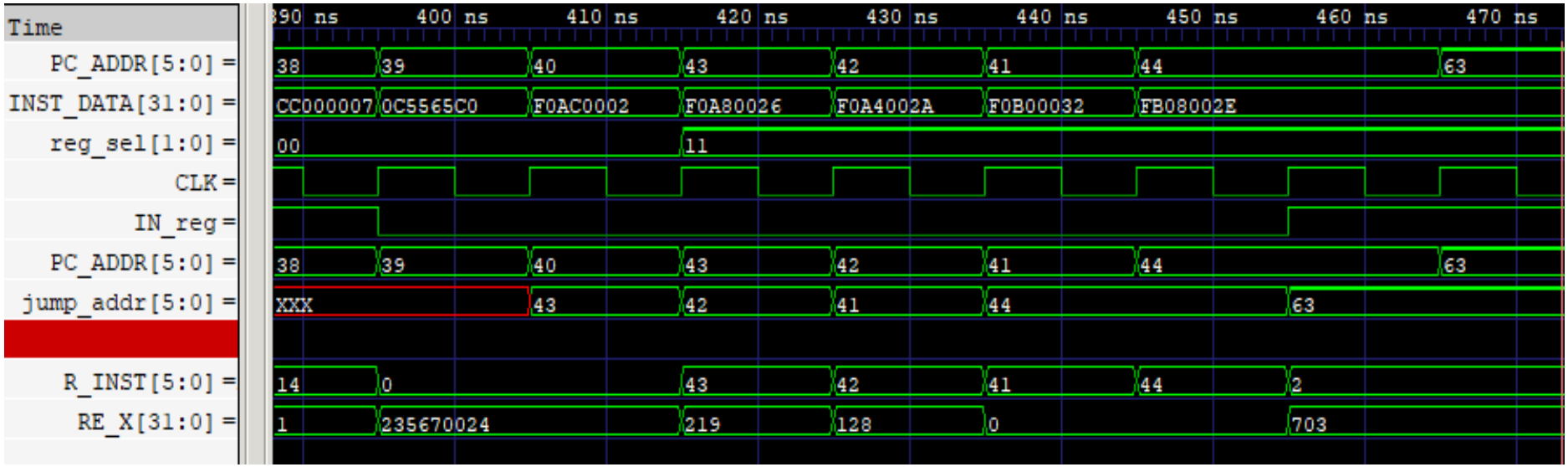


Store Operation
(Indirect Addressing Mode)





Jump Operation

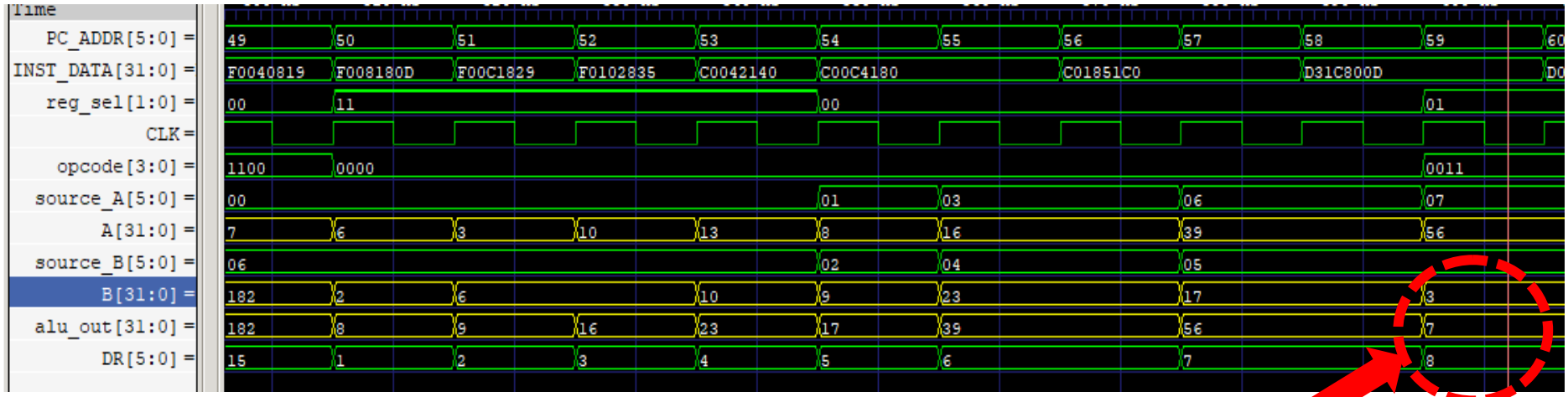


Average of 8 Numbers

1. Assembly Code

```
ADDI  01H. 02H, 06H
ADDI  02H, 03H, 06H
ADDI  03H, 06H, 10H
ADDI  04H, 13H, 10H
ADD   05H, 01H, 02H
ADD   06H, 03H, 04H
ADD   07H, 05H, 06H
SHR   03H, 07H
```

2. Waveform



Conclusion

On a whole, this internship has been an excellent and rewarding experience. I have gained new knowledge and skills. This internship helps me to enhance my understanding in digital systems and Verilog language. I enjoyed working with my group members and implemented basic RISC-V processor. I can conclude that there have been a lot I've learnt from my work during that time. The work experiences I encountered during the internship allowed me to develop skills/languages like Computer Architecture, System Verilog, designing Automatic scripts in Perl, etc.

The internship was also good to find out what my strengths and weaknesses are. This helped me to define what skills and knowledge I have to improve in the coming time. Two main things that I've learned the importance of our time-management skills and self-motivation.
At last, this internship has given me new insights and motivation in the VLSI Domain.

References

Hennessy, D. A. (n.d.). Computer Architecture: A Quantitative Approach.

Malvino, A., & Brown, J. (n.d.). Digital Computer Electronics.

Mano, M. M., & Ciletti, M. D. (n.d.). Digital Design.

Palnitkar, S. (n.d.). Verilog HDL: A Guide to Digital Design and Synthesis.

Patterson, D. A., & Hennessy, J. L. (n.d.). Computer Organization and Design RISC-V Edition: The Hardware Software Interface

Geeks For Geeks: https://www.geeksforgeeks.org