DESIGN AND IMPLEMENTATION OF SOFT PROCESSOR ON FPGA

A PROJECT REPORT

Submitted by

ADITHYA S	**(312212105005)**
DARISI VENKATA ABHILASH	**(312212105020)**
GADEPALLI SAI KRISHNA DILEEP	**(312212105027)**
GOWTHAM CPS	**(312212105032)**

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

in

ELECTRICAL AND ELECTRONICS ENGINEERING

SSN COLLEGE OF ENGINEERING

ANNA UNIVERSITY::CHENNAI 600 025

APRIL 2016

ANNA UNIVERSITY:CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report "**DESIGN AND IMPLEMENTATION OF SOFT PROCESSOR ON FPGA**" is the bonafide work of **"ADITHYA.S (312212105005), DARISI VENKATA ABHILASH (312212105020), GADEPALLI SAI KRISHNA DILEEP (312212105027) and GOWTHAM CPS (312212105032)"** who carried out the project work under my supervision.

SIGNATURE	**SIGNATURE**
Dr. V. KAMARAJ	Dr. M. SENTHIL KUMARAN
HEAD OF THE DEPARTMENT	**SUPERVISOR**
PROFESSOR	ASSOCIATE PROFESSOR
Department of Electrical and	Department of Electrical and
Electronics Engineering	Electronics Engineering
SSN College of Engineering	SSN College of Engineering
Kalavakkam, Chennai-603110	Kalavakkam, Chennai-603110

VIVA-VOCE EXAMINATION

The viva-voce examination for the project work, **“DESIGN AND IMPLEMENTATION OF SOFT PROCESSOR ON FPGA”** submitted by **ADITHYA.S (312212105005), DARISI VENKATA ABHILASH (312212105020), GADEPALLI SAI KRISHNA DILEEP (312212105027),** and **GOWTHAM CPS (312212105032)** held on ………………….

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ABSTRACT

A soft processor is a microprocessor core that can be wholly implemented using logic synthesis. It can be implemented through different semiconductor devices containing programmable logic like FPGA including both high-end and commodity variations. A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing. FPGAs contain an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together", like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR.

When designing an embedded system in a FPGA, we will most likely need some form of “controller” in our system. This controller can be a simple microcontroller or a fully-fledged microprocessor running the Linux operating system. One solution is to use an off-the-shelf (OTS) microprocessor mounted on the board and connecting to the FPGA. In fact, this still appears to be the most commonly-used solution. There are times, however, where an OTS processor-based approach will not meet our requirements. An example would be an application that requires peripheral functionality that is not available in a discrete solution, or where board real estate is limited. Another option is to embed a “hard”

processor core on the chip. A hard processor core has dedicated silicon area on the FPGA. Unfortunately, a hard processor core does not provide the ability to adjust it to better meet the needs of the application, nor does it allow for the flexibility of adding a processor to an existing FPGA design or adding an additional processor to provide more processing capabilities. A soft-core processor solution is one that is implemented entirely in the logic primitives of an FPGA. Thus, it is the solution that gives maximum flexibility. The main advantages of a soft processor include a higher level of portability, affordability, easier to implement and easier to modify.

ACKNOWLEDGEMENT

We would like to thank our college management for their support and for providing us with the equipment required for the successful completion of the project.

We convey our gratitude to **Dr.S.Salivahanan**, Principal, SSN College of Engineering for providing the opportunity to complete the project. We would like to express our sincere thanks to **Dr.V.Kamaraj**, Professor and Head, Department of Electrical and Electronics Engineering for his persistent encouragement and support to carry out this project.

We express our deepest thanks to our **Dr.M.Senthil Kumaran**, Associate Professor, Department of Electrical and Electronics Engineering for guiding us with care and attention. He has taken pain to go through the project and make necessary corrections when needed. Without him, this project would never have taken shape.

We are indebted to all the faculty members and assistants of the Electrical and Electronics Engineering Department for their invaluable assistance.

ADITHYA S

DARISI VENKATA ABHILASH

GADEPALLI SAI KRISHNA DILEEP

GOWTHAM CPS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABBREVIATION	EXPANSION
ALU	Arithmetic and Logic Unit
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
PS2	Personal System
RAM	Random Access Memory
ROM	Read only Memory
VGA	Virtual Graphic Accelerator

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

When designing an embedded solution, the designer will have product level requirements that mandate the processing of various inputs to yield predictable outputs. The requirements will vary according to the application. Therefore, we need a processing platform that is flexible and can be configured accordingly. Thus, it is necessary to develop a soft processor for such situations.

The soft processor provides the flexibility through the configurable nature of the FPGA. This will allow the designer to change the design according to changing requirements. It also prevents the designer to be confined to a specific set of peripherals that may no longer fit the application as requirements change or new features are desired as peripherals can be added or removed with ease. The designer has the flexibility to create a custom system that contains only the functionality needed which will eliminate the burden of handling unwanted functionality. Also, soft-core processors and their accompanying toolsets can make the task of implementing multiple processor cores that interface with a common set of peripherals much more feasible and appealing to designers. Also, there are not any additional costs for adding a soft-core processor in an FPGA as long as there is enough space in the FPGA for the implementation.

1.2 LITERATURE REVIEW

In the year 2006, J.G Tong presented a paper on “Soft core processors for Embedded Systems”. This paper presented a survey of soft processors that are used in embedded systems. In addition, several real world examples of embedded systems that employ soft processors are summarized.
In the year 2007, Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose presented a paper on “Exploration and Customization of FPGA-Based Soft Processors”. They presented an exploration of the micro-architectural tradeoffs for soft processors and a set of customization techniques that capitalizes on these tradeoffs to improve the efficiency of the soft processor for specific applications.

In the year 2005, Noam Nisan, Shimon Schocken published the book titled” The Elements of Computing Systems”. The book specifies a simple but powerful hardware and software systems are tightly interrelated through a hidden web of abstractions, interfaces, and contract-based implementations. It also demonstrates the ability to construct, through recursive ascent and human reasoning, fantastically complex and useful systems from nothing more than a few primitive building blocks.

In the year 2008, Pong P Chu, published the book “FPGA Prototyping by VHDL examples”. The book gives an insight on practical implementation of different protocols like VGA, PS2 required to interface the keyboard and screen with the processor.

1.3 OBJECTIVES

- To design a soft core processor.
- To implement it on a Field programmable Gate Array.
- To increase ease of customization by providing open source architecture.

1.4 CHAPTER ORGRANIZATION

The project report is structured into 7 chapters.

Chapter 1 gives a summary of what is to follow and gives a bird's eye view of the report. The chapters that follow start with an introduction to the specific subject under review. The report finally has a conclusion that reiterates the results.

Chapter 2 gives a brief introduction to the FPGA and Xilinx software. It also specifies the method to convert a circuit into VHDL code using Xilinx software with help of system generator block.

Chapter 3 gives general description and implementation of basic components like Logic gates, combinational circuits, sequential circuits and memory components.

Chapter 4 presents the Architecture of the soft processor, focuses on design and implementation of the Arithmetic and Logic Unit and details the design and implementation of Central Processing Unit.

Chapter 5 presents the need for assembler, its design and implementation in a high level programming language, presents the design and implementation of Virtual Machine and compiler. It also describes the Keyboard and Screen drivers required to interface between the devices and processor.

Chapter 6 presents the hardware implementation of the processor.

Chapter 7 gives the conclusion and the future scope of the project.

CHAPTER-2

INSIGHT ON FPGA AND XILINX

2.1 INTRODUCTION

The FPGA's are semiconductor devices that are based around a matrix of configurable logic blocks connected via programmable interconnects. They can be reprogrammed and reconfigured according to the requirements of the application. Hence, they are addressed as "field programmable". The logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGA's, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

The most common architecture of a FPGA consists of an array of logic blocks, I/O pads, and routing channels. An application circuit must be mapped into FPGA with adequate resources.

Due to its high density of LUTs it supports parallel execution of considerable amount of bit level operations. With an FPGA, calculations that would normally consume large amounts of CPU time are accelerated.

The Basic Diagram of FPGA is given in the Fig.2.1.

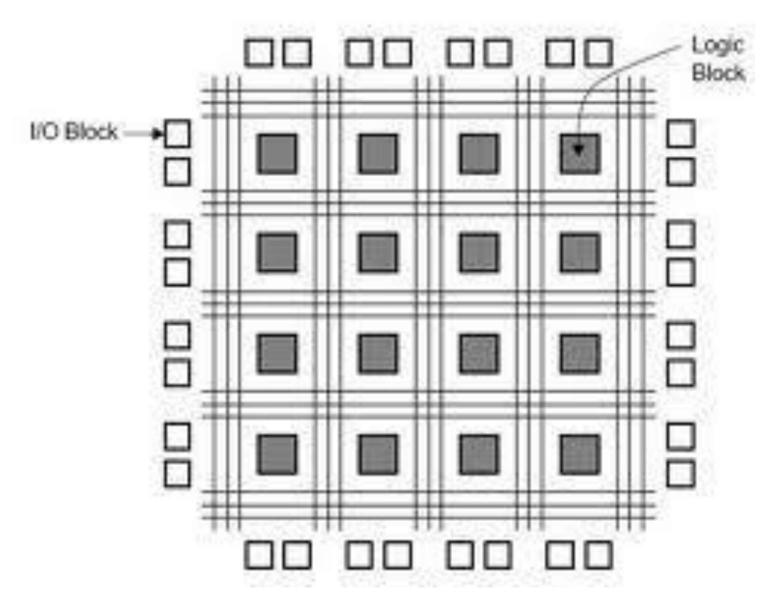


Fig.2.1. Basic Block Diagram of FPGA

2.2 INTERFACING OF FPGA USING MATLAB

Using System Generator, one can automatically generate VHDL code for FPGAs from Xilinx Blockset models. It is easy to create models of algorithms for implementation in FPGAs using high-level components from Xilinx-specific blocksets. Xilinx libraries include blocks for communication, control logic, signal processing, mathematics, and memory. After creation of Xilinx based model, VHDL based code is extracted from MATLAB system generator block. This VHDL code is then converted into bit file using ISE Design software and it is written into FPGA memory.

2.3 SYSTEM GENERATOR

System Generator is a design tool box from Xilinx that enables the use of the Mathworks model-based design environment Simulink for FPGA design. System Generator supports a black box block that allows RTL to be imported into Simulink and co-simulated with either ModelSim or Xilinx ISE Simulator. It includes a set of complex DSP building blocks such as forward error correction blocks, FFTs, filters and memories. Over 90 DSP building blocks are provided in the Xilinx DSP blockset for Simulink. These blocks include the common building blocks such as adders, multipliers and registers .

CHAPTER 3

BASIC COMPONENTS

The basic components like logic gates, adders, incrementers, and flip-flops are designed and implemented for using them in implementation of logic circuits of greater complexity in future. The components are built upon the basic NAND gate which is considered as the most basic gate.

3.1 LOGIC GATES

3.1.1 AND GATE

- Its output can be obtained in a way by negating the output of a NAND gate.

Table 3.1 Truth Table of AND Gate

A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

- It can be derived from the NAND gate as below:

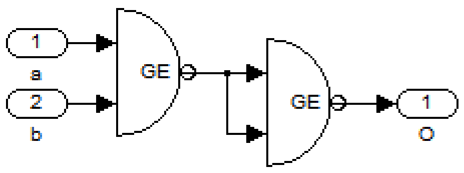


Fig.3.1 Implementation of AND Gate

- Once we develop a single AND gate, we can use it to develop AND gate to AND two sixteen bit words.

3.1.2 OR GATE

- To implement this particular gate, we need to know De' Morgan's Law in hand.
- According to this law, if A and B are two inputs, then
- (A.B)'=A'+B' ; (A+B)'=A'.B'
- Using the above law, we can derive Or gate as below: (A'.B')'=(A+B)''=A+B

Table 3.2 Truth Table of OR Gate

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1

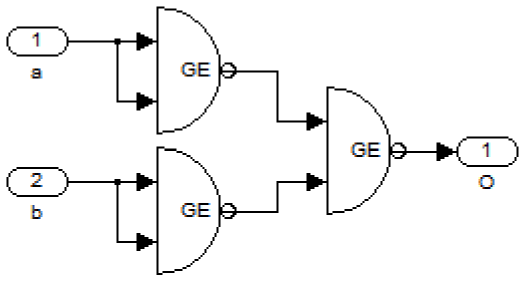


Fig.3.2 Implementation of OR Gate

3.1.3 EX-OR GATE

- It can be easily understood as a dislike operator i.e.. that it gives an output of 0 for like inputs and an output of 1 for dislike inputs.

Table 3.3 Truth Table of EX-OR Gate

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	0

- The output expression can be expressed in terms of sum of products.
- Its output expression is Or(And((a)’,b),And(a,(b)’))

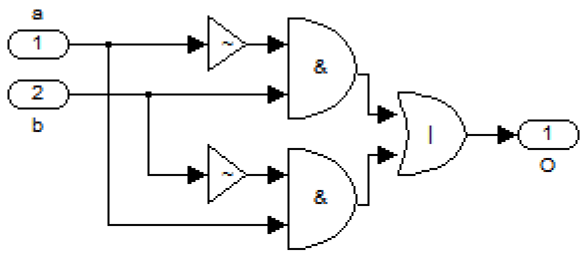


Fig.3.3 Implementation of EX-OR Gate

3.2 COMBINATIONAL LOGIC CIRCUITS

3.2.1 MULTIPLEXER

- This is a digital logic chip used to fan in different input channels into a single output channel.
- This is mainly used in communication techniques for broadcasting purposes. It is used mostly on the transmitter side of the communication end.
- This is also called as Many-to-one gate.
- The logic behind the operation is that a particular input is sent to output channel based on the value of the select line.
- The basic Multiplexer will have two inputs, a select line and an output.
- The first input is sent to output channel if the select line is 0 else the second input is sent to output channel.

Table 3.4 Truth Table of Multiplexer

A	B	SEL	OUT
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

- The output expression for the output can be obtained from the truth table by taking terms with one in their outputs.
- Thus , OUT = A'.B.SEL+A.B'.SEL'+A.B.SEL'+A.B.SEL
- Thus, it can be implemented using AND, NOT, OR gates by connecting them in the way as given by the output expression.
- The Multiplexer can be viewed as below:

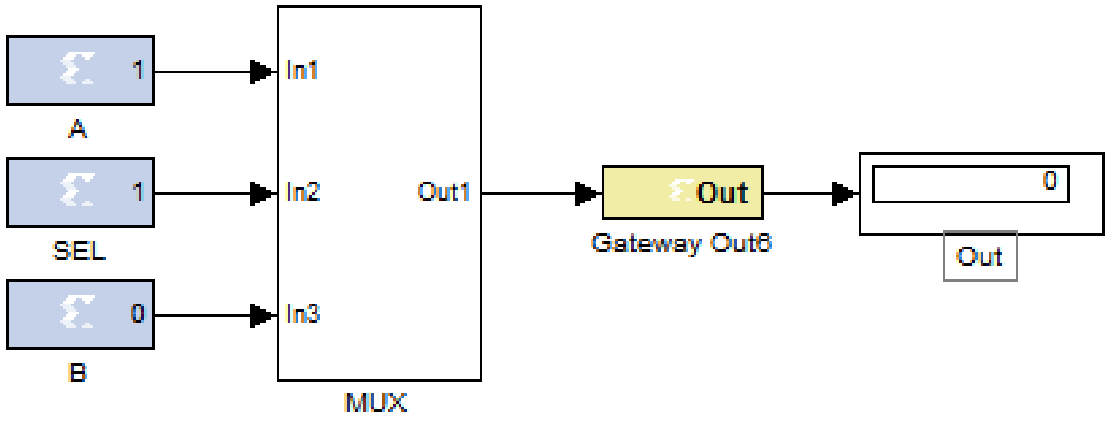


Fig.3.4 Implementation of Multiplexer

- Once we develop the basic 2:1 multiplexer, we can use it develop a Multiplexer which can handle 16 bit words. This multiplexer can further be used to develop 4:1 Multiplexer.

3.2.2 DE-MULTIPLEXER

- This is a digital logic chip used to fan out a single input to different destinations.
- This is mainly used in communication techniques for broadcasting purposes. It is used mostly on the receiver side of the communication end.
- This is also called as One-to-Many gate
- The logic behind the operation is that the input is sent to a particular output channel based on the value of the select line.
- The basic De-multiplexer will have one input, a select line and two outputs.

- The input is sent to first output channel if the select line is 0 else it is sent to second output channel.

Table 3.5 Truth Table of De-Multiplexer

IN	SEL	A	B
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

- The output expression for each of the two outputs can be obtained from the truth table by taking terms with one in their outputs.
- Thus, A=IN.SEL' and B=IN.SEL
- Thus, it can be implemented using AND, NOT, OR gates by connecting them in the way as given by the truth table.
- Once we develop the basic 1:2 De-Multiplexer, we can use it to develop 1:4 and 1:8 De-Multiplexers.

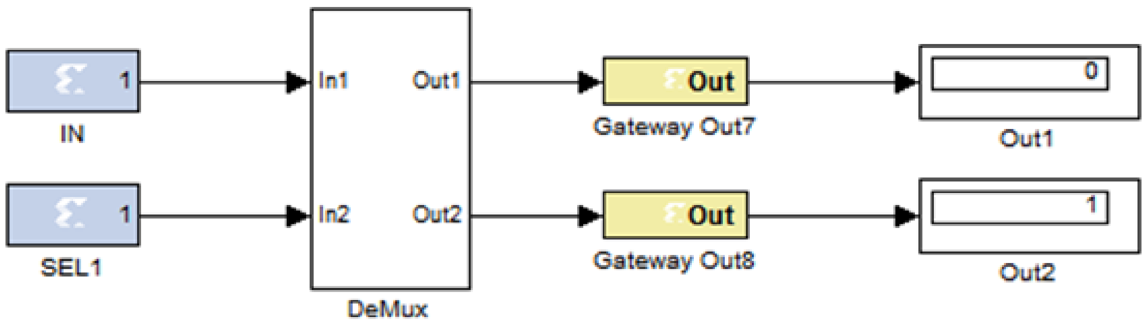


Fig.3.5 Implementation of De-Multiplexer

3.2.3 HALF ADDER

- This is a digital logic circuit used to add two binary bits and give their sum and carry if any.
- Carry occurs due to overflow of bits. For example, when we add 1 and 1,the sum is two i.e 10 in binary. Therefore, sum is 0 and carry is 1.

Table 3.6 Truth Table of Half-Adder

A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

- The output expressions for sum and carry can be obtained by expressing the terms in the output with one as sum of products.
- Thus, SUM = A'.B+A.B' and CARRY = A.B
- Thus, a half adder can be implemented using an XOR gate for SUM and an AND gate for CARRY.

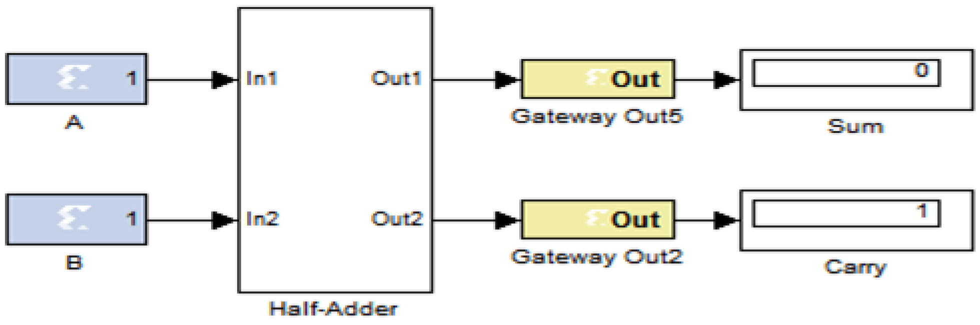


Fig.3.6 Implementation of Half Adder

3.2.4 FULL ADDER

- This is a digital logic circuit used to add three bits and give out their sum and carry if any.

Table 3.7 Truth Table of Full Adder

A	B	C	SUM	CARRY
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- This is implemented by either implementing the SUM and CARRY as sum of products of those terms with one in their outputs. But a better method is to represent the sum and carry terms in K-Map to get simplified expression.
- Thus by using a K-map, we get SUM = A XOR B XOR C and CARRY = A.B+ B.C+ C.A

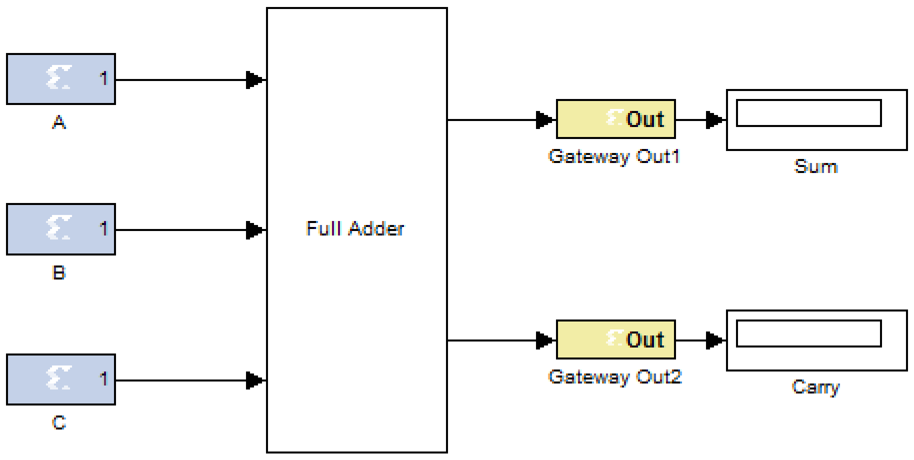


Fig.3.7 Implementation of Full Adder

3.3 SEQUENTIAL LOGIC CIRCUITS

3.3.1 D-FLIP FLOP

- The D type flip-flop has one data input 'D' and a clock input. The circuit edge triggers on the clock input. The flip-flop also has two outputs Q and Q' (where Q' is the reverse of Q).

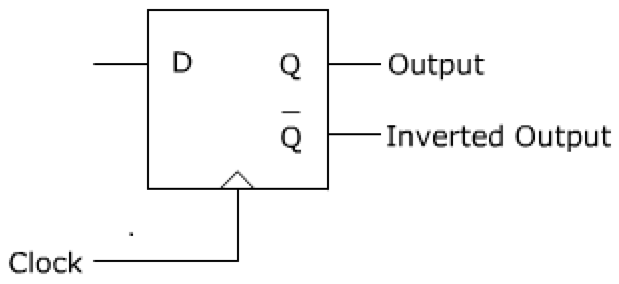


Fig.3.8 Symbol of D-Flip Flop

- The operation of the D type flip-flop is as follows: Any input appearing (present state) at the input D, will be produced at the output Q in time T+1 (next state). e.g. if in the present state we have D = 0 and Q = 1, the next state will be D = anything and Q = 0.
- Knowing the above, we can now generate the state change table

Input	Circuit Action
D	$Q_{(\text{Time } t+1)}$
0	0
1	1

- The operation of the D type flip flop delays any input by exactly one clock cycle (given an instantaneous response time i.e. a perfect flip-flop). Cascading several D type flip-flops together can produce delaying circuits. The possible applications could be for matching time delays in digital television systems.

- The Xilinx blockset provides an in-built D-dlip flop which was used to implement the other components.

3.4 MEMORY COMPONENTS

- The memory of our processor is divided into instruction and data memory.
- The instruction memory is implemented using a ROM-Read Only Memory which is a non-volatile memory and doesn't require continuous power supply to contain the data stored in it.
- The data memory is implemented using a RAM-Random Access Memory which is a volatile memory and once the power supply is removed, the data is erased.
- On the whole, we would implement memory as a package of three lower level chips namely a RAM16k for data memory, RAM8k for screen memory map and a single 16 bit register for keyboard memory map.
- The screen is considered to be a matrix of 400 columns and 300 rows where each element of the matrix represents a pixel. Each pixel is given a unique memory location in the memory map where each row of the screen is represented as a collection of 25 rows. Now, to turn on a particular pixel, we just have to set that bit to one in the memory map. In this fashion we can display alphabets, draw figures by appropriately programming the memory dedicated for the screen.
- The implementation of the keyboard is much simpler. Each of the keys on the keyboard has a unique scan code. Therefore, when a key is pressed, its scan code gets stored in the dedicated memory register in binary format. Thus, by constantly polling this memory register we can know the key pressed and send appropriate signals to the screen logic to display the

corresponding letter. The below picture was taken from book written by Noam Nisan, Shimon Schocken (2005)

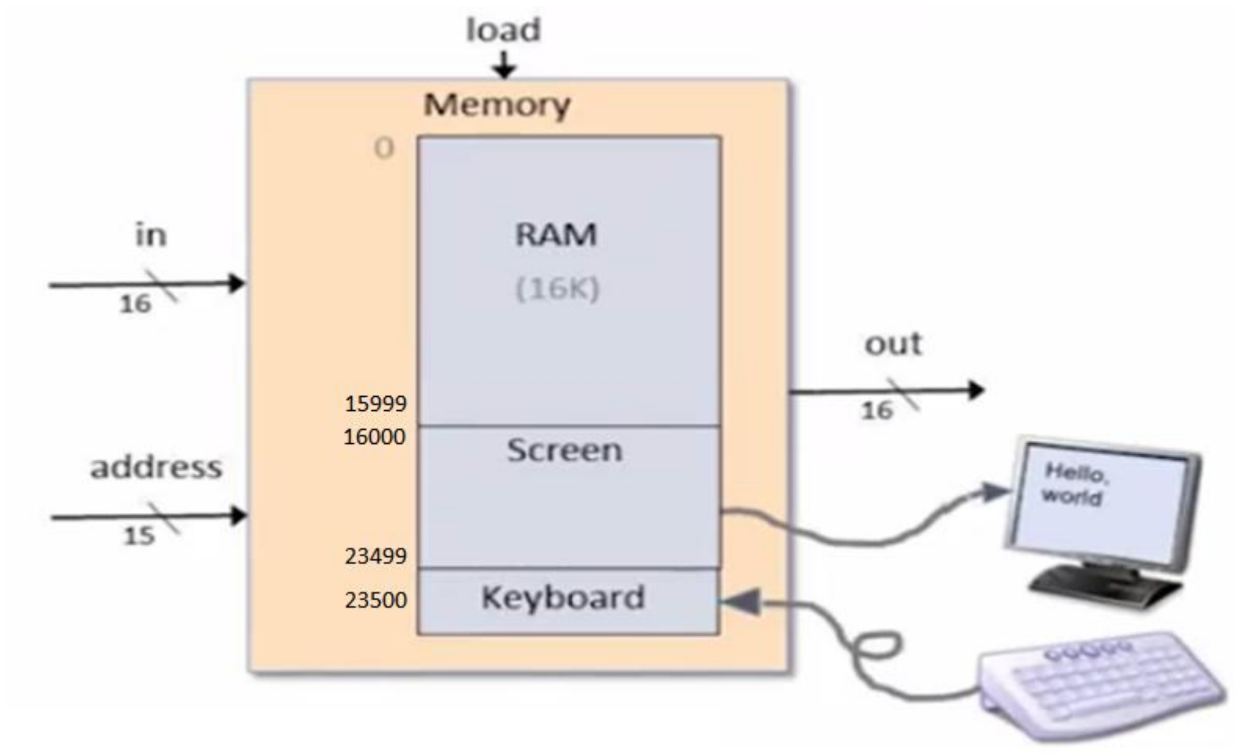


Fig.3.9 Memory Organization of Soft Processor

3.4.1 SINGLE BIT REGISTER

- It is a storage device that can store or remember data over time.
- The main component in the register is the D-flip flop.
- The output of the register can either be the input or the output of the previous instant.
- We cannot tell the D-flip flop when to take the input and when to take the output from previous instant.
- Therefore, we go with the below setup to solve this ambiguity:

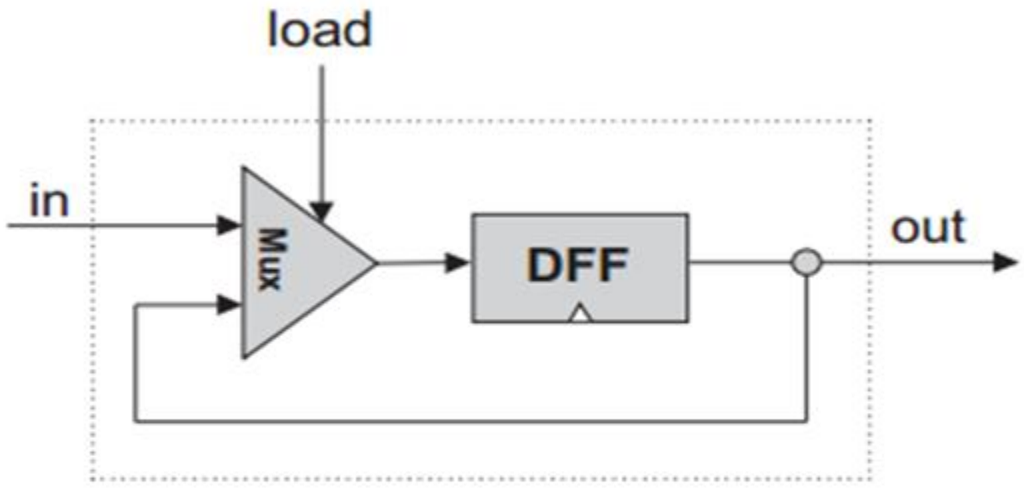


Fig.3.10 Implementation of Single Bit Register

- By the help of load bit and multiplexer, we can solve the above ambiguity.
- When the load bit is set, the D-flip flop takes its input from in-wire and when load bit is zero, it takes the input from the out-wire. Thus, the load bit acts as a select line to the multiplexer for choosing between the two options.
- We should remember that the D-flip flop is clocked.
- Once we have a single bit register, we can extend it to obtain a 16 bit register.
- This can be obtained by using 16 single bit registers which can store 16 bits and hence make up a single 16 bit register.

3.4.2 RAM-RANDOM ACCESS MEMORY

- It is a volatile memory used for temporary storage of data. It requires continuous power supply to maintain the data. Once the supply is cut off, the data gets erased.
- A basic RAM device has three inputs: a data input, an address input and a load bit.
- It can be implemented using the single bit registers.

- Our aim is to develop a 16k RAM. But it cannot be built directly. We have to design them in a hierarchical fashion building upon each of the RAM chip.
- Hence, we first make a basic RAM8 chip which will have eight 16 bit registers.
- But we have to choose one of the eight registers to store the data.
- This can be done using a de-multiplexer which will take load bit as the input and address as the select lines to choose between the registers and give eight outputs.
- Each of these outputs is given as load bits to each of the registers which also take the input of the RAM as their input and give output with respect to the load bit.
- A multiplexer is again to be used to collect all the outputs of the registers and output the contents to register only selected with help of the address.
- To read the data from a particular register, we should just specify the address and the output is emitted by inherent RAM logic.
- To write the data into a particular register, we should specify the address and enable the load bit by which the input is written into the desired register.
- Once we build a RAM8 chip, we can use eight such chips to develop a RAM64 chip.
- We can extend the above concept to build RAM512, RAM4k, RAM16k chips in a similar fashion.

3.4.3 PROGRAM COUNTER-PC

- A Program counter is type of counter used to store the address of the next instruction that is to be executed by the CPU.
- Thus, it is an integral part of the CPU.

- It works by below conditions:
- if (reset[t] == 1) out[t+1] = 0
- else if (load[t] == 1) out[t+1] = in[t]
- else if (inc[t] == 1) out[t+1] = out[t] + 1 (integer addition)
- else out[t+1] = out[t]
- Therefore, to implement this, we need a register to satisfy second condition and a 16 bit incrementer to satisfy third condition.
- Also to choose between the four condition's outputs, we would need a multiplexer with reset and load bits as select lines and last four inputs as zero with first four inputs being the outputs from each condition.

- With the above knowledge, the program counter was implemented as below in Fig.3.11

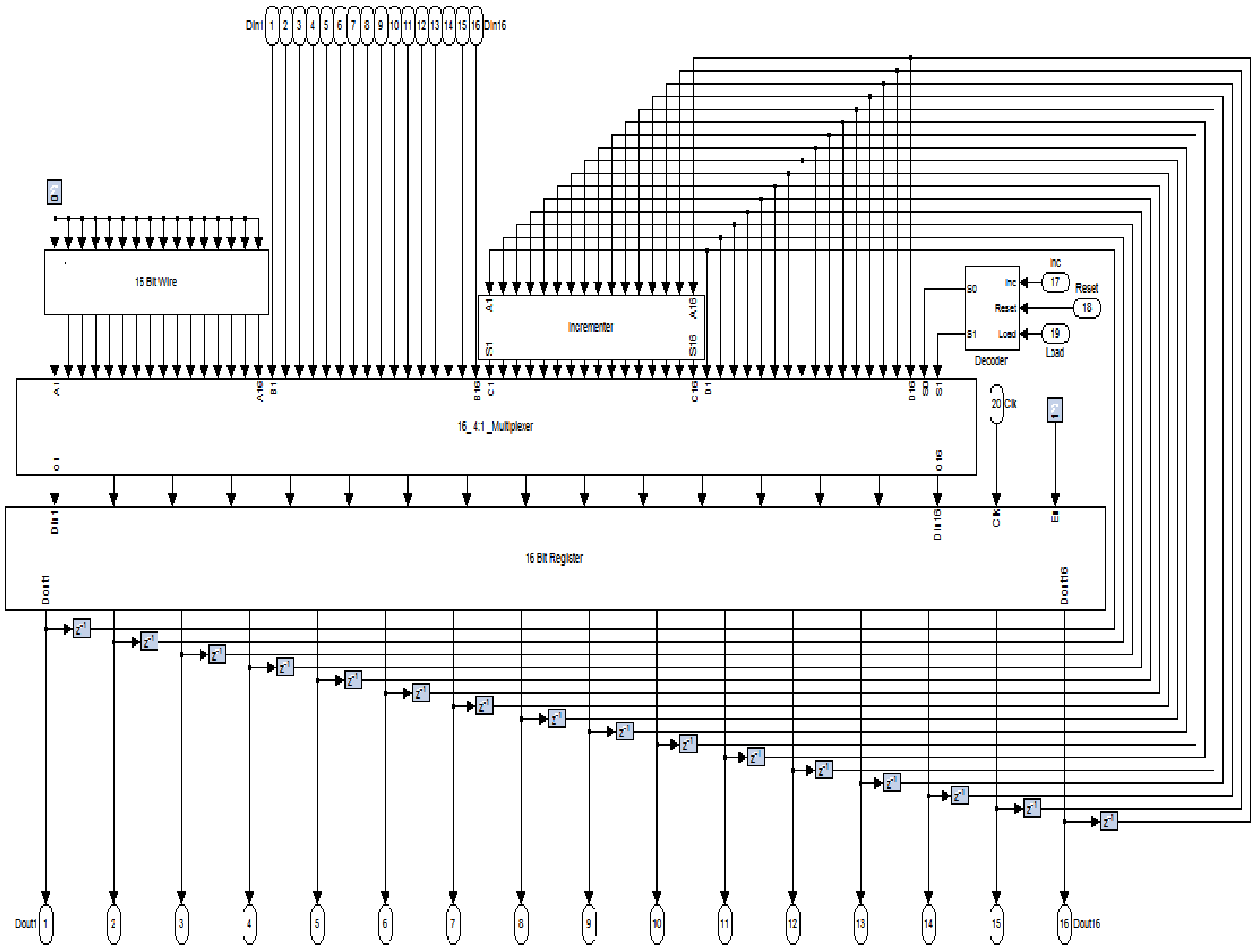


Fig.3.11 Implementation of Program Counter

CHAPTER 4

ARCHITECTURE OF PROCESSOR, ALU AND CPU

4.1 ARCHITECTURE OF SOFT PROCESSOR

- We will implement our soft processor using the well-known Von-Neumann architecture.
- The Von Neumann architecture is a practical architecture and the conceptual blueprint of almost all the computers.
- The Von Neumann architecture is based on a central processing unit (CPU), interacting with a memory device, receiving data from some input device, and sending data to some output device.
- The computer’s memory stores not only the data that the computer manipulates, but also the very instructions that tell the processor what to do.
- The memory allocated to data and instruction is unique and independent of each other so that there will not be any problem while processing instructions that need to access data and program memory at the same time.
- High-level programs manipulate abstract artifacts like variables, arrays, and objects. When translated into machine language, these data abstractions become series of binary numbers, stored in the computer’s data memory. Once an individual word has been selected from the data memory by specifying its address, it can be either read or written to. In the former case, we retrieve the word’s value. In the latter case, we store a new value into the selected location, erasing the old value.
- When translated into machine language, each high-level command becomes a series of binary words, representing machine language instructions. These instructions are stored in the computer’s instruction memory. In each step of the computer’s operation, the CPU fetches (i.e., reads) a word from the

instruction memory, decodes it, executes the specified instruction, and figures out which instruction to execute next. Thus, changing the contents of the instruction memory has the effect of completely changing the computer's operation.

- There are three types of registers we would implement in our processor. They are A-register or Address register to store addresses, D-register or Data register to store data temporarily and M-register or Memory register too for storing data. The below picture was taken from the book written by Noam Nisan, Shimon Schocken (2005).

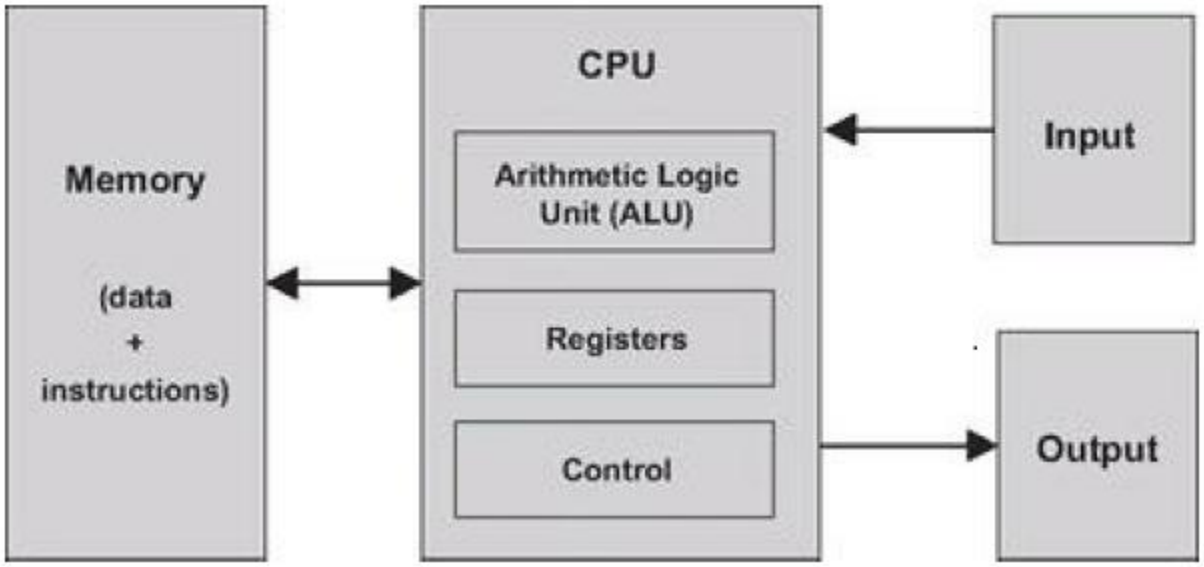


Fig.4.1 Architecture of Soft Processor

- There are three types of information that flow between these components. They are data, instructions and address.
- This flow of information is represented by the following diagram:

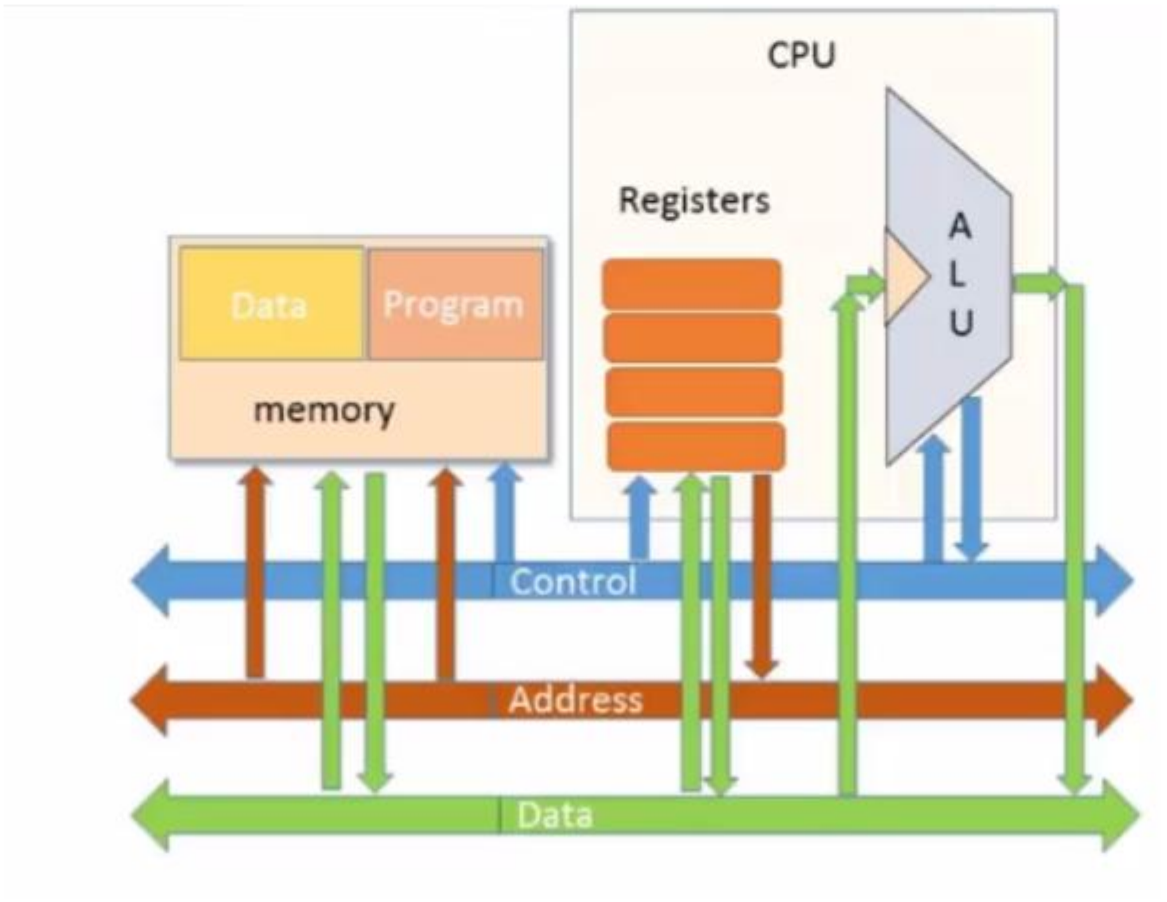


Fig.4.2 Information Flow

- We can see from the above diagram that the data flows from data memory and registers to ALU and vice-versa. Therefore, it is a bi-directional flow of data.
- Similarly, the address flows from registers to memory.
- The control instructions flow from ALU to registers and memory and vice versa.

4.2 DESIGN OF ALU--ARITHMETIC AND LOGIC UNIT

- It is the key part of the central processing unit.
- It can operate certain defined mathematical and logical operations on the 16 bit inputs x, y.
- Therefore, the output, out=f(x, y), where, out is a 16 bit output and f is the operation to be operated on the inputs.
- The operation to be performed is decided by the control bits supplied by the user.
- Actually, these control bits are decoded from the instruction given to the CPU. The below picture was taken from the book written by Noam Nisan, Shimon Schocken (2005).

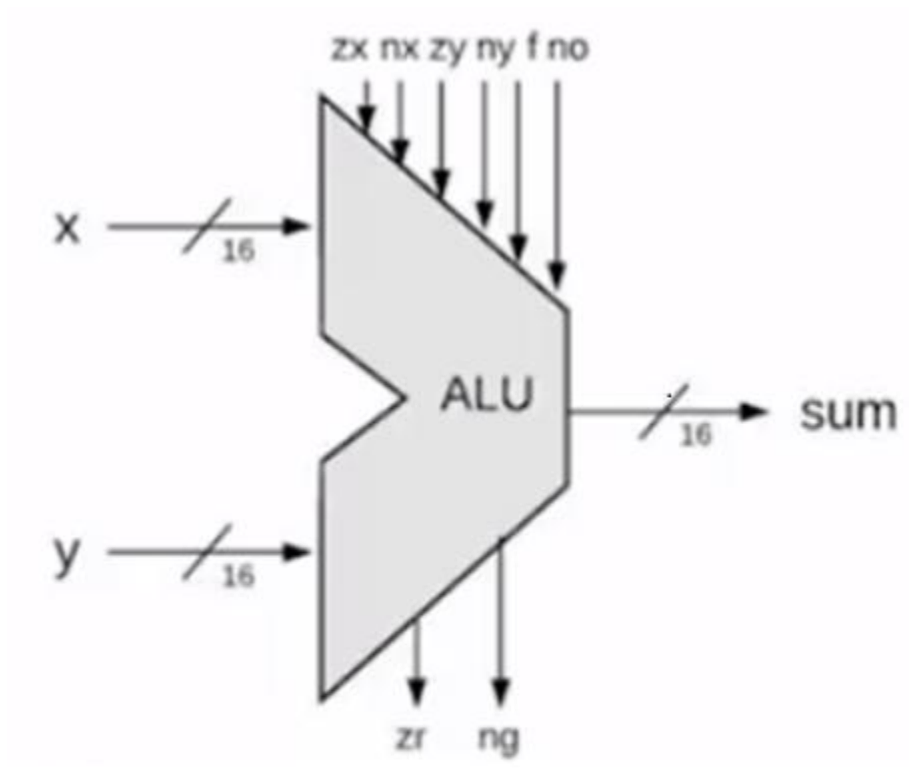
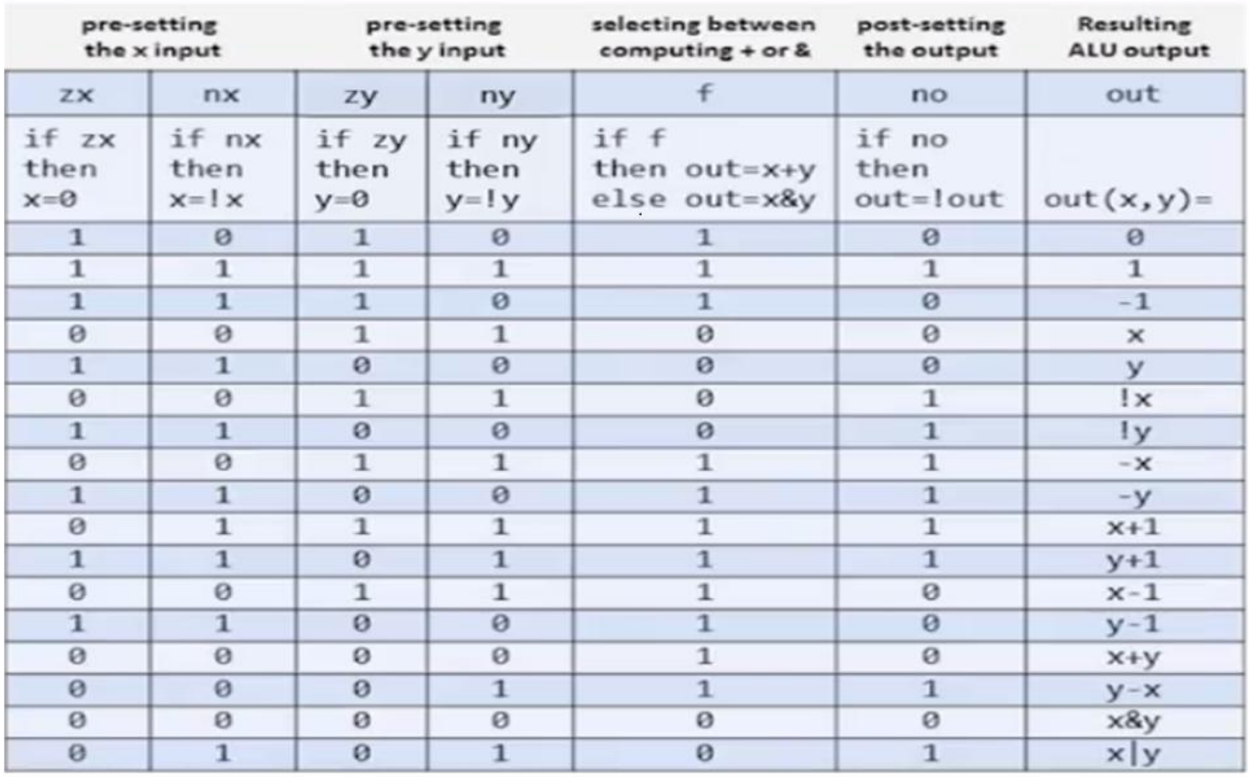


Fig.4.3 Basic Block Diagram of ALU

- The meaning of each of the inputs is as follows:
- If zx bit is set then all the bits of the x input are set to zero and if not, then there will not be any impact on the x input because of zx bit.
- If nx bit is set then all the bits of the x input are negated and if not,then there will not be any impact on the x input because of nx bit.
- Similarly,it is the same with zy and ny bits too.
- If f bit is set, then x+y is computed, else, x&y is computed.
- If no bit is set,then the output is negated, else,there will not be any impact
- because of nx bit.

- Therefore, considering the definitions and basic operations to be performed, we can come with a truth table as below:

Table 4.1 Truth Table of ALU



pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	nx	zy	ny	f	no	out
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

- It is to be noted that each bit definition will be implemented on the output from the operation of the input by the previous bit.
- We opt for an ALU with ability to perform basic functions in order to ensure simplicity and better economy but at the same time equally powerful.
- The other operations can be done with the existing functions. They have not been specifically taken up in order to reduce hardware complexity.
- To understand the operation ,let us take up an example as below:

- Let x=1100 and y=1011 and let us compute x!
- As we want to compute x!, we should not change its original value.
- Therefore, zx=0, nx=0.
- To get the negated x input, we can take an AND product with -1(all one’s) and negate the output.
- Therefore, to get -1, we make zy=1 and ny =1 to get -1 in the y input.
- Now, making f=0 will give us x&y i.e.. 1100
- Now, making no=1 will give us !(x&y) i.e.. 0011 which is actually !x

4.3 IMPLEMENTATION OF ALU

The Implementation is as follows:

```
if (zx == 1) set x = 0        // 16-bit constant

if (nx == 1) set x = !x       // bitwise not

 if (zy == 1) set y = 0        // 16-bit constant

 if (ny == 1) set y = !y       // bitwise not

 if (f == 1)  set out = x + y  // integer 2's complement addition

 if (f == 0)  set out = x & y  // bitwise and

 if (no == 1) set out = !out   // bitwise not

 if (out == 0) set zr = 1

 if (out < 0) set ng = 1
```

To implement the first four conditions, we use two 4:1 multiplexers to give suitable outputs depending upon the control inputs acting as select lines of the multiplexer.

To implement the fifth and sixth conditions, we have to obtain the AND product and the Adder outputs, for which, we implement the AND gate which can handle 16 bit inputs and 16 bit adder to get their sum. The output are chosen using a multiplexer capable of handling 16 bits with the ‘f’ bit as the select line.

The seventh condition can be implemented by obtaining the negated output using the NOT gate capable of handling 16 bits and a multiplexer capable of handling 16 bits with ‘no’ bit as the select line to choose between the original and negated outputs.

The MSB of the output after processing of the first six conditions is assigned to be the ‘ng’ bit. If it set, it means that the output is negative. The ‘zr’ bit denotes if the output is zero or non-zero. This is checked in the way that 16 bit output is split into two 8 bit arrays and each of them are logically ORed. The outputs of the two OR gates are negated using a NOT gate and its output is assigned to the ‘zr’ bit.

With the above knowledge, the ALU was implemented as below in Fig.4.4

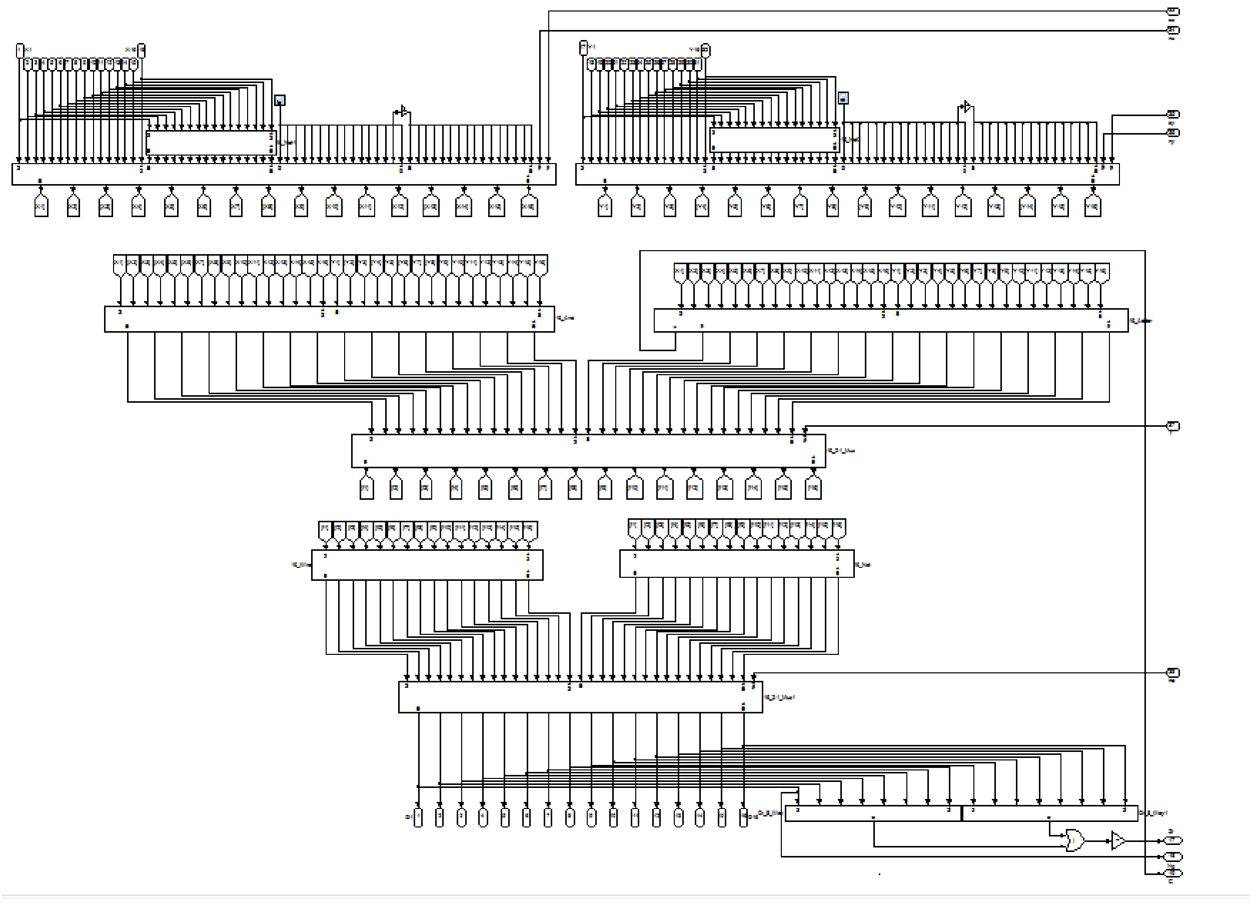


Fig.4.4 Implementation of ALU

4.4 DESIGN OF CPU- CENTRAL PROCESSING UNIT-CPU

- The CPU—the centerpiece of the computer's architecture—is in charge of executing the instructions of the currently loaded program. These instructions tell the CPU to carry out various calculations, to read and write values from and into the memory, and to conditionally jump to execute other instructions in the program. The CPU executes these tasks using three main hardware elements: an Arithmetic-Logic Unit (ALU), a set of registers, and a control unit.
- Before we go on with the architecture and operation of CPU it is required that we have to note a few points about our hack machine language.
- It's based on two 16 bit command types namely A-command i.e.. address command and C-command i.e.. compute command.
- These commands are differentiated by the CPU using the MSB bit which is the op-code. If it is 1, then, the instruction is a C-command and when it is zero, the instruction is an A-command.
- The A-instruction will cause the CPU to load the fifteen bit word which represents the address into the A register. The C-instruction is of this format 111accccccdddjjj, where the first bit is the op-code, a and c bits instruct the ALU to compute a particular function and d bits instruct where to store the ALU output and j bits specify an optional conditional jump.
- The CPU is connected to two separate memory modules: an instruction memory, from which it fetches instructions for execution, and a data memory, from which it can read, and into which it can write, data values.
- Therefore, with above information, we can go into the architecture and operation of CPU.
- The CPU black box can be viewed as:

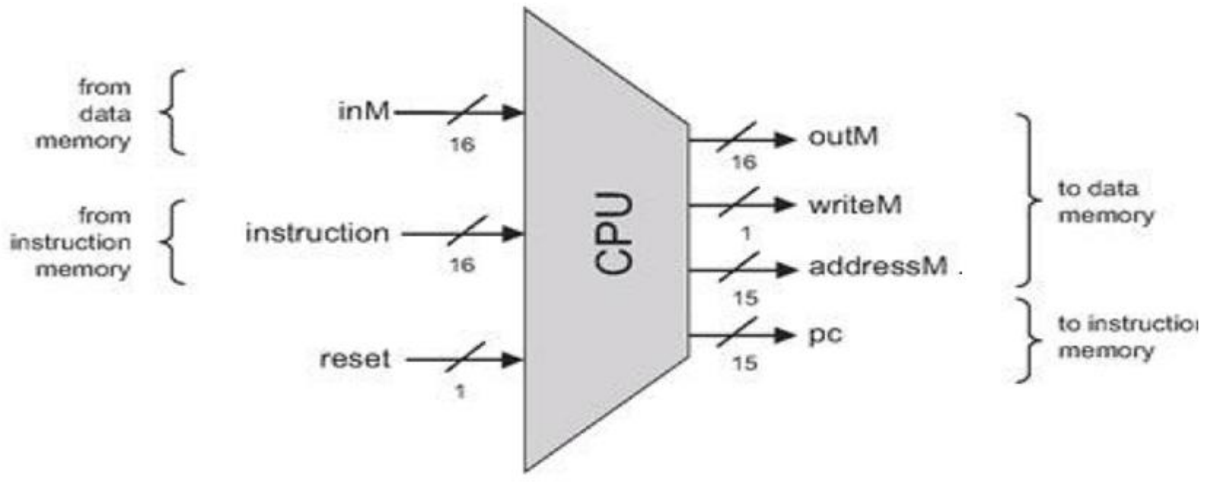


Fig.4.5 Black Box of CPU

- The above picture was taken from the book written by Noam Nisan, Shimon Schocken (2005).
- The inM is a 16 bit word which is obtained from the data memory.
- The instruction is a 16 bit word which is obtained from the instruction memory.
- The reset bit specifies if the program counter is to be set to point to start of the program or point to the next location.
- The outM is a 16 bit word which represents the output of the ALU. It has to be written in the memory. Therefore, it requires an address where it can be written. This is specified by addressM 15 bit word. The WriteM is single bit output which when enabled will allow the data in outM word to be written at the address specified by the addressM.
- The CPU implementation objective is to create a logic gate architecture capable of executing a given instruction and fetching the next instruction to be executed. Naturally, the CPU will include an ALU capable of executing instructions, a set of registers, and some control logic designed to fetch and decode instructions.

- One of the most efficient and simple ways of connecting the ALU, registers and the program counter is as below:

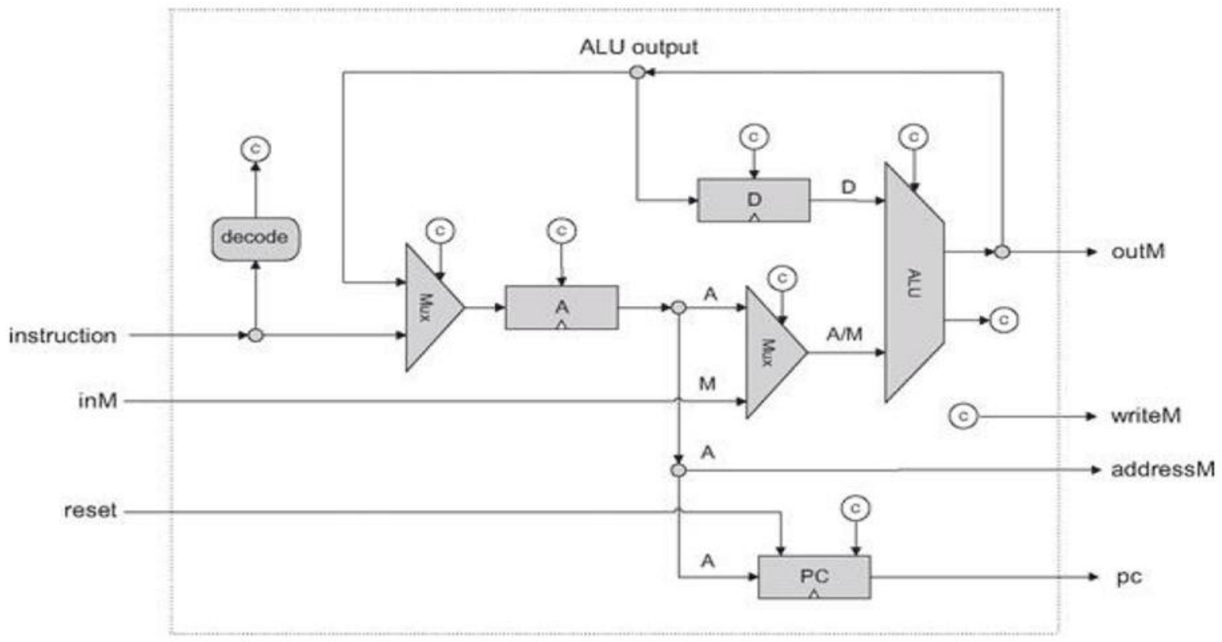


Fig.4.6 Design of CPU

- The above picture was taken from the book written by Noam Nisan, Shimon Schocken (2005).
- The lines represent the flow of data and instructions between the components.

4.5 IMPLEMENTATION OF CPU

- The control logic of the CPU can be divided into three tasks, namely Instruction decoding, Instruction fetching and Instruction execution.
- **Instruction Decoding:** The 16-bit word located in the CPU's instruction input can represent either an A-instruction or a C-instruction. In order to figure out what this 16-bit word means, it can be broken into the fields "i xx a cccccc ddd jjj". The i-bit codes the instruction type, which is 0 for an A instruction and 1 for a C-instruction. In case of a C-instruction, the a-bit and the c-bits code the comp part, the d-bits code the dest part, and the j-bits code the jump part of the instruction. In case of an A instruction, the 15 bits other than the i-bit should be interpreted as a 15-bit constant.
- **Instruction Execution:** The various fields of the instruction (i-, a-, c-, d-, and j-bits) are routed simultaneously to various parts of the architecture, where they cause different chips to do what they are supposed to do in order to execute either the A-instruction or the C-instruction, as mandated by the machine language specification. In particular, the a-bit determines whether the ALU will operate on the A register input or on the Memory input, the c-bits determine which function will the ALU compute, and the d-bits enable various locations to accept the ALU result. The i-bit is used as the select line in the first multiplexer to choose between the instruction and ALU output. The a-bit is used as the select line for the second multiplexer to choose between A-register and memory register. The c-bits are used as control bits of ALU. The j-bits are used in program counter logic to determine the address of next instruction. The addressM word is taken from A-register and writeM is enabled when the CPU is processing a c instruction.

- **Next Instruction Fetching:** As a side effect of executing the current instruction, the CPU also determines the address of the next instruction and emits it via its pc output. This task is done by the program counter—an internal part of the CPU whose output is fed directly to the CPU's pc output. Therefore, we have to come up with a hardware implementation of following logic: It jump(t) then PC(t) = A(t-1) else PC(t) = PC(t-1)+1. To effect the desired jump control logic, we start by connecting the output of the A register to the input of the PC. The only remaining question is when to enable the PC to accept this value, namely, when does a jump need to occur. This is a function of two signals: (a) the j-bits of the current instruction, specifying on which condition we are supposed to jump, and (b) the ALU output status bits, indicating whether the condition is satisfied. If we have a jump, the PC should be loaded with A's output. Otherwise, the PC should increment by 1. Additionally, if we want the computer to restart the program's execution, all we have to do is reset the program counter to 0. That's why the proposed CPU implementation feeds the CPU's reset input directly into the reset pin of the PC

- With the above knowledge, the CPU was implemented as below in Fig4.7.

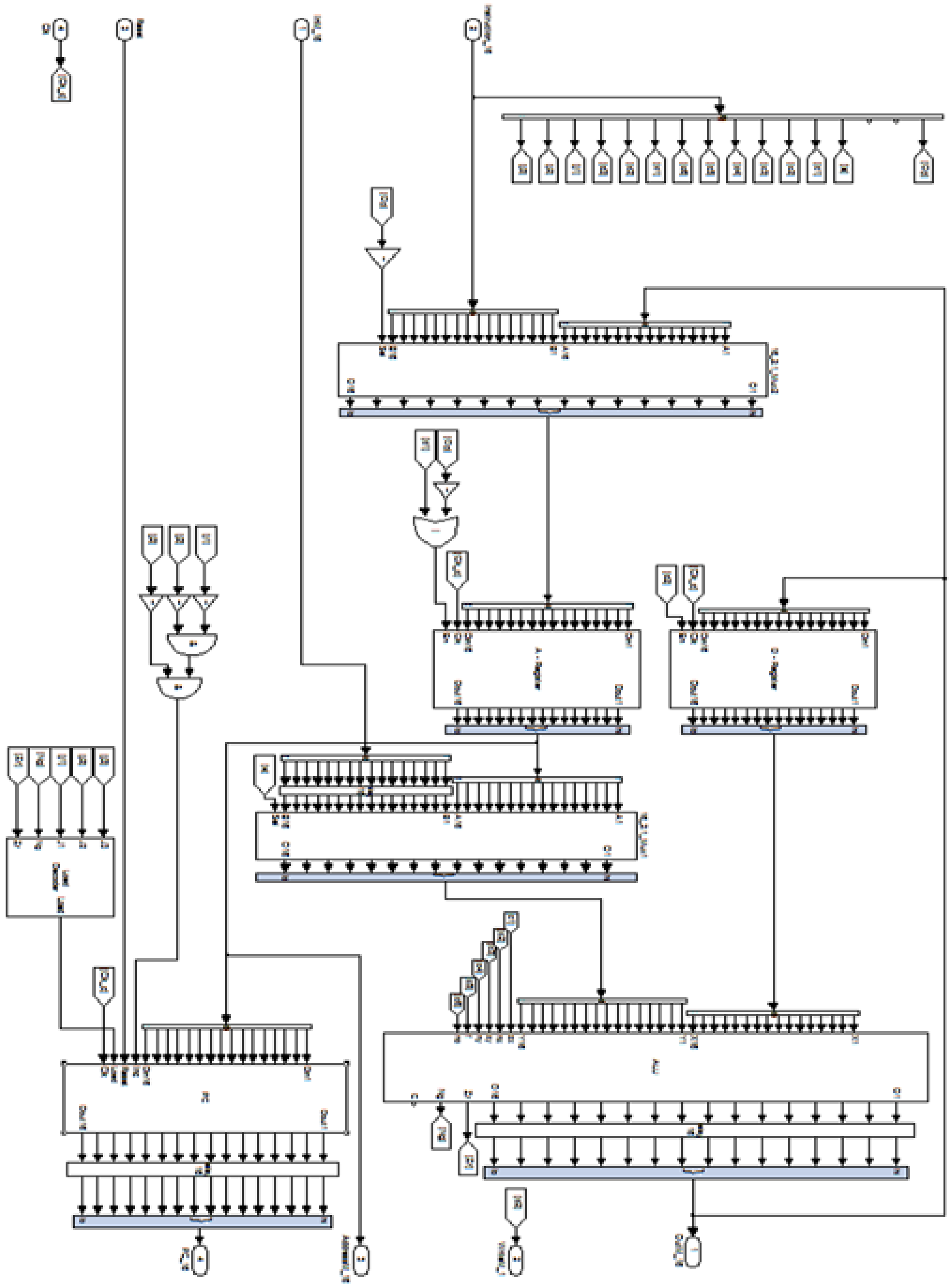


Fig.4.7 Implementation of CPU in MATLAB

CHAPTER 5

SUPPORT MODULES AND DRIVERS

5.1 ASSEMBLER

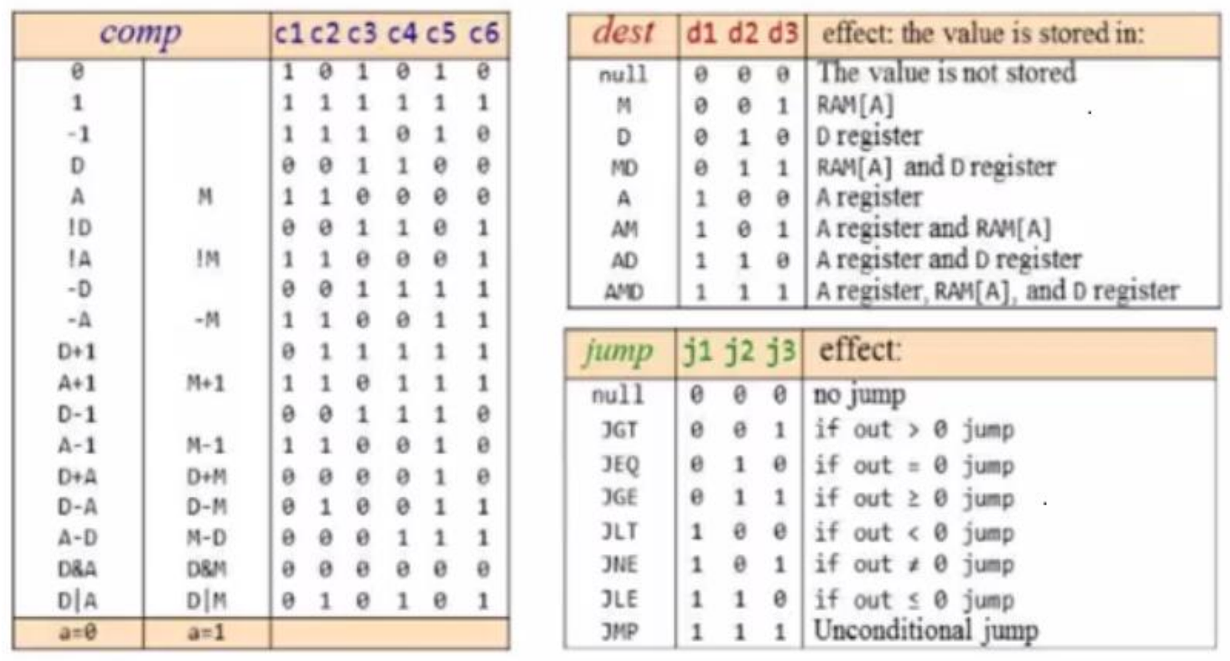
- Assembler may refer to a computer program which translates assembly language to an object file or machine language format.
- It is basically a program written in a high level language to convert the assembly language commands into machine language command in binary format which can only be understood by the processor.
- It can be implemented by knowing the corresponding translations. These translations are as below:
- Once the translations are known, we just have to map the commands appropriately to get the corresponding binary translations.
- We implemented the program as combination of three runs.
- The first run is to remove all the blank spaces and comments in the file where the assembly language program is written. The blank spaces are removed in a way that when the file is read in a sequential order, commands are written in a new array and if blank spaces occur, they are not written into it. This is done in a same way for the comments too. Hence, the new array of strings will have the assembly language program without any blank spaces and comments.
- The second run is to map the labels and variables into the corresponding memory locations. It is to be noted that the variables are mapped from 16th memory location as the first fifteen memory locations are dedicated to registers, stack pointer, this, that pointers. When a new variable occurs in the program, its value is stored in the sixteenth location in the memory and when the next variable occurs, its valued is stored in the next location and so

on. Similarly, when a label occurs its line number is stored in a memory location and when its reference occurs the next time, the label is replaced by the line number which will move the control to that location. The labels are recognized by the program as tags between parenthesis and variables are recognized as tags next to '@' character. Thus, in this way, both the labels and variables are mapped into appropriate memory locations. Therefore, after the second run, the array of strings will have an assembly program without comments, blank spaces and with labels, variables suitably mapped and referenced.

- The third run is to convert each of the assembly language command into the binary commands. The translation is divided into two parts, each of it handling the conversion of the A-instructions and C-instructions. The conversion of the A-instruction is very simple. It is noted that the fifteen bits next to the MSB represent an address. Thus, this number in decimal format can be converted into binary format directly. In a similar way, the labels which are referenced to a particular memory location in the previous run can also be converted in to binary format easily. The conversion of C-instructions is not very easy. We have to handle more than one case during conversions. We have C- instructions separated on each side by either,'=' or ';' characters. In each case, we split the assembly command into two parts and map each part into the translation tables to get the corresponding binary words. Thus, after the third run we will obtain the complete translation of the assembly language code. This is written in to a file for further implementations of the program.

- The below table represents the translations between assembly code to binary code.

Table 5.1 Translation Table of Assembler

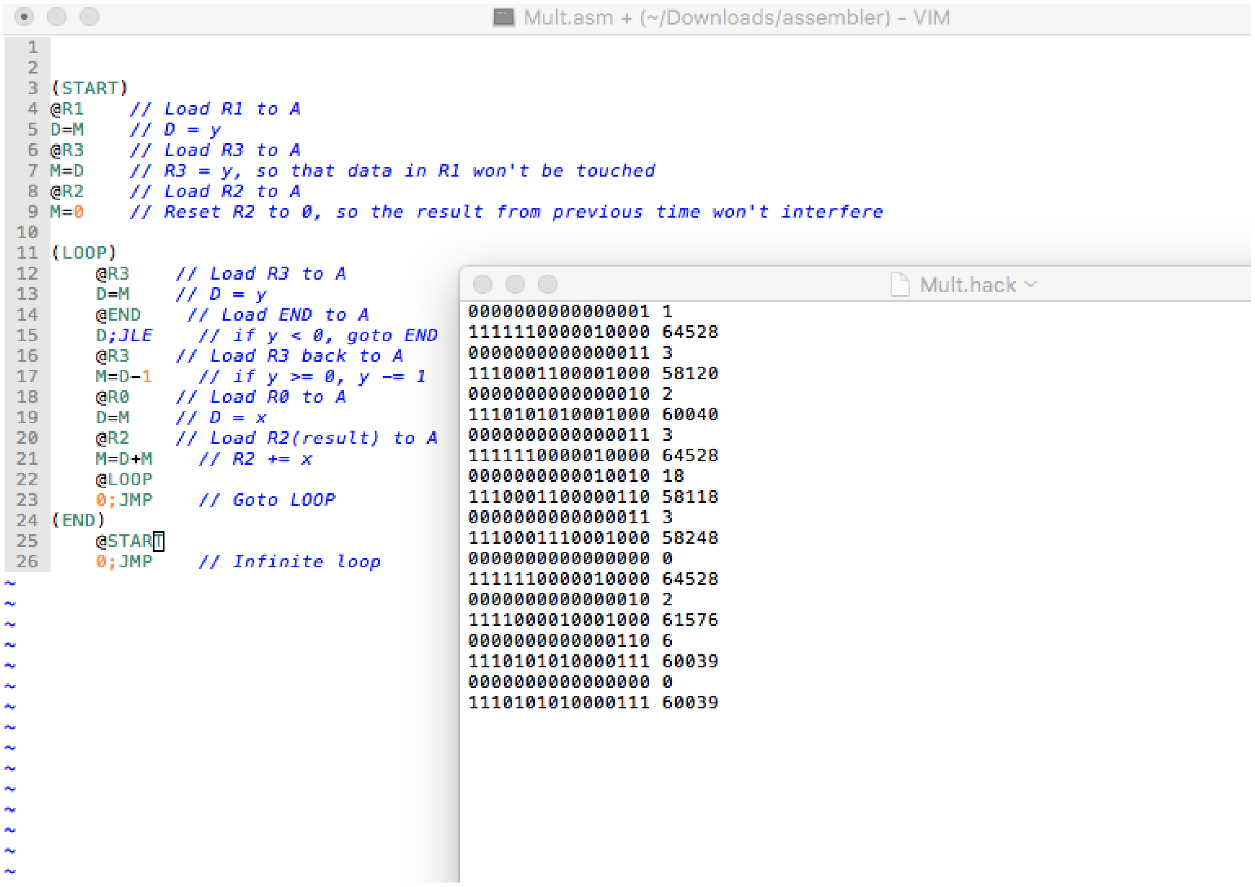


comp		c1	c2	c3	c4	c5	c6
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a=0	a=1						

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Translation of Assembly level Program into Machine level Binary code:



```


(START)
@R1     // Load R1 to A
D=M     // D = y
@R3     // Load R3 to A
M=D     // R3 = y, so that data in R1 won't be touched
@R2     // Load R2 to A
M=0     // Reset R2 to 0, so the result from previous time won't interfere

(LOOP)
    @R3     // Load R3 to A
    D=M     // D = y
    @END     // Load END to A
    D;JLE     // if y < 0, goto END
    @R3     // Load R3 back to A
    M=D-1     // if y >= 0, y -= 1
    @R0     // Load R0 to A
    D=M     // D = x
    @R2     // Load R2(result) to A
    M=D+M     // R2 += x
    @LOOP
    0;JMP     // Goto LOOP
(END)
    @START
    0;JMP     // Infinite loop
```

```
0000000000000001 1
1111110000010000 64528
0000000000000011 3
1110001100001000 58120
0000000000000010 2
1110101010001000 60040
0000000000000011 3
1111110000010000 64528
0000000000010010 18
1110001100000110 58118
0000000000000011 3
1110001110001000 58248
0000000000000000 0
1111110000010000 64528
0000000000000010 2
1111000010001000 61576
0000000000000110 6
1110101010000111 60039
0000000000000000 0
1110101010000111 60039
```

Fig.5.1 Translation of Assembly code to binary code

5.2 VIRTUAL MACHINE

The motive behind developing the VM machine is to make the processor platform independent. Instead of running on a real platform, the intermediate code is designed to run on a Virtual Machine. There are many reasons why this idea makes sense, one of which being code transportability. Since the VM may be implemented with relative ease on multiple target platforms, VM-based software can run on many processors and operating systems without having to modify the original source code. The VM implementations can be realized in several ways, by software interpreters, by special-purpose hardware, or by translating the VM programs into the machine language of the target platform.

Another benefit of the virtual machine approach is modularity. Every improvement in the efficiency of the VM implementation is immediately inherited by all the compilers above it. We would use the stack data structure to implement our VM translator program which will convert VM language instructions into assembly language instructions. In a stack machine model, arithmetic commands pop their operands from the top of the stack and push their results back onto the top of the stack. Other commands transfer data items from the stack's top to designated memory locations, and vice versa. As it turns out, these simple stack operations can be used to implement the evaluation of any arithmetic or logical expression. Further, any program, written in any programming language, can be translated into an equivalent stack machine program.
The virtual machine is stack-based: all operations are done on a stack. In the stack model, arithmetic commands pop their operands from the top of the stack and
push their results back onto the top of the stack. Other commands transfer data items from the stack's top to designated memory locations, and vice versa. As it turns out, these simple stack operations can be used to implement the evaluation of

any arithmetic or logical expression. Further, any program, written in any programming language, can be translated into an equivalent stack machine

It is also function-based: a complete VM program is organized in program units called functions, written in the VM language. Each function has its own stand-alone code and is separately handled. The VM language has a single 16-bit data type that can be used as an integer, a Boolean, or a pointer. The language consists of four types of commands:

Arithmetic and Logic commands perform arithmetic and logical operations on the stack. The VM language features nine stack-oriented arithmetic and logical commands. Seven of these commands are binary: They pop two items off the stack, compute a binary function on them, and push the result back onto the stack. The remaining two commands are unary: they pop a single item off the stack, compute a unary function on it, and push the result back onto the stack.

Memory access commands transfer data between the stack and virtual memory segments. The memory segments are accessed by the commands: push segment index which will push the value of segment[index] into the stack and pop segment index which will pop the stack value and store it in segment[index]. The segment is one of the segment names namely argument, local, static, constant, this, that, pointer, temp which have their unique functions and index is a non-negative integer.

Program flow commands facilitate conditional and unconditional branching operations. These commands include the function, call and return commands.

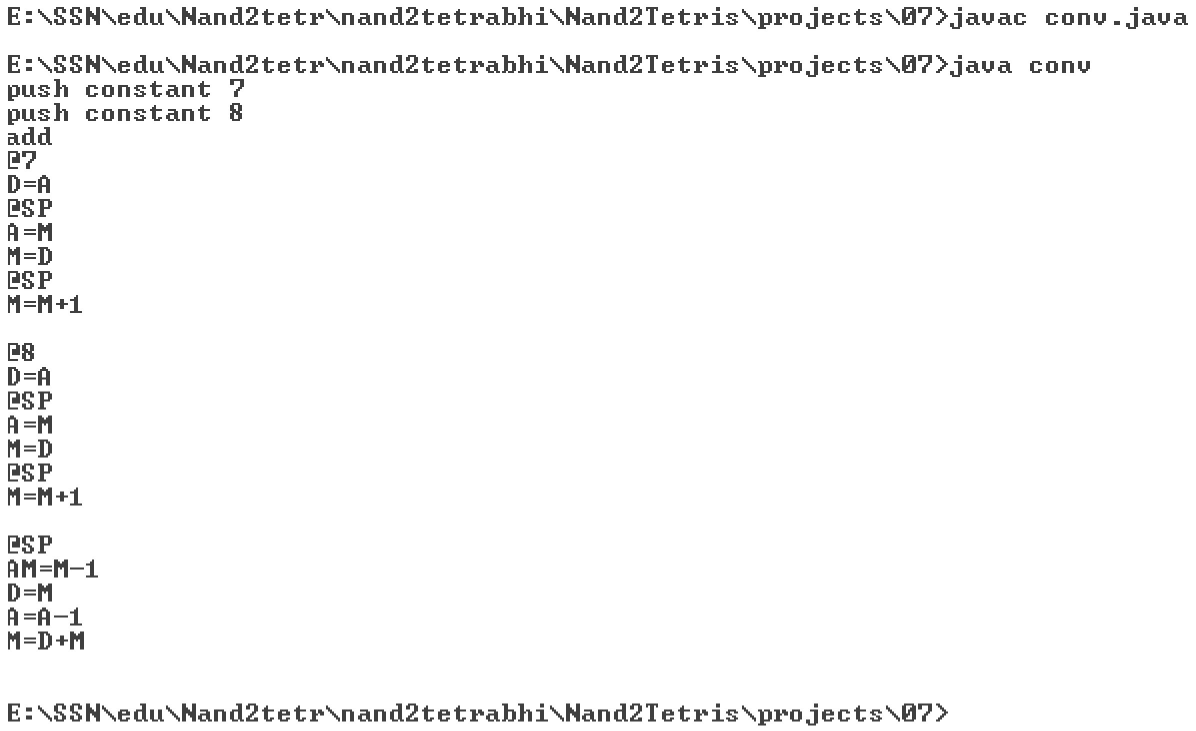
Function calling commands call functions and return from them.

The implementation of virtual machine should be such that it is completely platform independent without actually committing to any one of the hardware platform. These commands are appropriately translated in to assembly level language by the VM translator program. The program was implemented in JAVA programming language. The inputs and output of the program are as below:

INPUT: A VM file with commands written in VM intermediate language.

```
push constant 7
push constant 8
add
```

OUTPUT: Equivalent assembly Language code



```
E:\SSN\edu\Nand2tetr\nand2tetrabhi\Nand2Tetris\projects\07>javac conv.java

E:\SSN\edu\Nand2tetr\nand2tetrabhi\Nand2Tetris\projects\07>java conv
push constant 7
push constant 8
add
@7
D=A
@SP
A=M
M=D
@SP
M=M+1

@8
D=A
@SP
A=M
M=D
@SP
M=M+1

@SP
AM=M-1
D=M
A=A-1
M=D+M


E:\SSN\edu\Nand2tetr\nand2tetrabhi\Nand2Tetris\projects\07>
```

Fig.5.2 Translation of VM code to assembly code

5.3 VGA SCREEN DRIVER

5.3.1 VIDEO CONTROLLER

A video controller generates the synchronization signals and outputs data pixels serially.

A simplified block diagram of a VGA controller is shown in Figure. It contains a synchronization circuit, labeled vga-sync, and a pixel generation circuit.

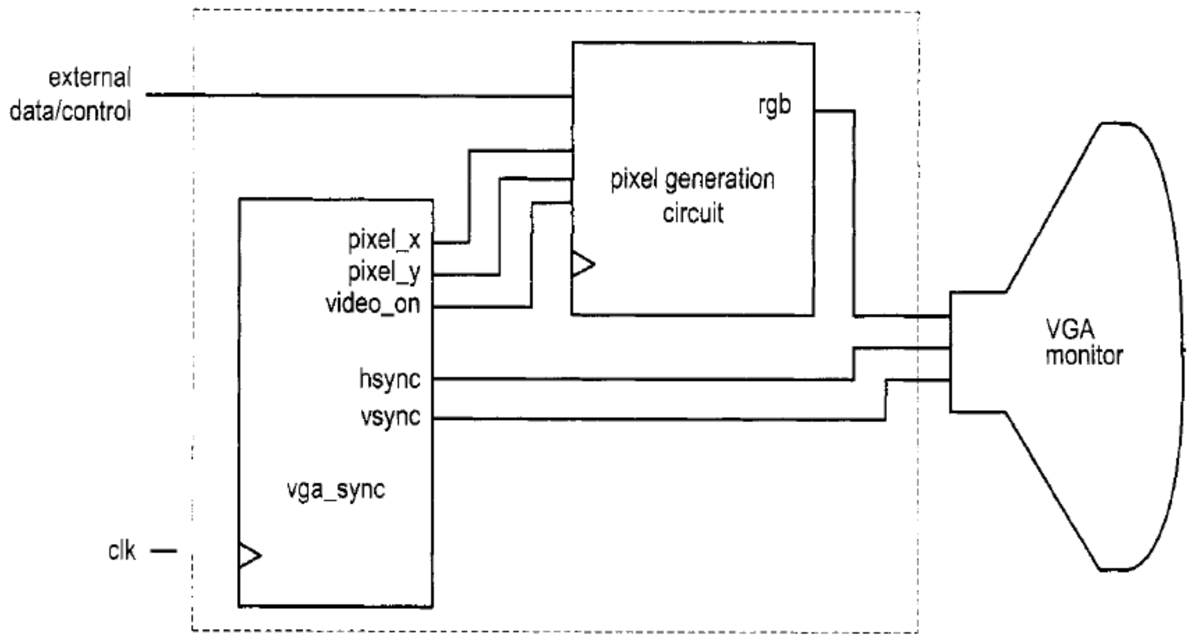


Fig.5.3 Block Diagram of VGA Controller

The above picture was taken from the book written by Pong P Chu (2008).

The vga-sync circuit generates the timing and synchronization signals. The hsync and vsync signals are connected to the VGA port to control the horizontal and vertical scans of the monitor. The two signals are decoded from the internal counters, whose outputs are the pixel-x and pixel-y signals. The pixel-x and pixel-y signals indicate the relative positions of the scans and essentially specify the location of the current pixel. The vga-sync circuit also generates the video-on signal to indicate whether to enable or disable the display.

The pixel generation circuit generates the three video signals, which are collectively referred to as the rgb signal. A color value is obtained according to the current coordinates of the pixel (the pixel-x and pixel-y signals) and the external control and data signals.

5.3.2 VGA SYNCHRONIZATION

The video synchronization circuit generates the hsync signal, which specifies the required time to traverse (scan) a row, and the vsync signal, which specifies the required time to traverse (scan) the entire screen. Subsequent discussions are based on a 640-by-480 VGA screen with a 25-MHz pixel rate, which means that 25M pixels are processed in a second.

Note that this resolution is also known as the VGA mode. Note that the coordinate of the vertical axis increases downward.

5.3.2.1 HORIZONTAL SYNCHRONIZATION

A period of the hsync signal contains 800 pixels and can be divided into four regions:

Display*:* region where the pixels are actually displayed on the screen. The length of this region is 640 pixels.

Retrace: region in which the electron beams return to the left edge. The video signal should be disabled (i.e., black), and the length of this region is 96 pixels.

Right border: region that forms the right border of the display region. It is also known as the front porch (i.e., porch before retrace). The video signal should be disabled, and the length of this region is 16 pixels.

Left border: region that forms the left border of the display region. It is also know as the back porch (i.e., porch after retrace). The video signal should be disabled, and the length of this region is 48 pixels.

Note that the lengths of the right and left borders may vary for different brands of monitors.

The hsync signal can be obtained by a special mod-800 counter and a decoding circuit.

5.3.2.2 VERTICAL SYNCHRONIZATION

During the vertical scan, the electron beams move gradually from top to bottom and then

return to the top. This corresponds to the time required to refresh the entire screen. The format of the vsync signal is similar to that of the hsync signal. The time unit of the movement is represented in terms of horizontal scan lines. A period

of the vsync signal is *525* lines and can be divided into four regions:

Display: region where the horizontal lines are actually displayed on the screen. The length of this region is 480 lines.

Retrace: region that the electron beams return to the top of the screen. The video signal should be disabled, and the length of this region is 2 lines.

Bottom border: region that forms the bottom border of the display region. It is also know as the front porch (i.e., porch before retrace). The video signal should be disabled, and the length of this region is 10 lines.

Top border: region that forms the top border of the display region. It is also know as the back porch (i,e,, porch after retrace). The video signal should be disabled, and the length of this region is 33 lines.

As in the horizontal scan, the lengths of the top and bottom borders may vary for different brands of monitors.

The vsync signal can be obtained by a special mod-525 counter and a decoding circuit.

Again, we intentionally start counting from the beginning of the display region. This allows us to use the counter output as the vertical (y-axis) coordinate. This output constitutes the pixel-y signal. The vsync signal goes low when the line count is 490 or 491.

As in the horizontal scan, we use the v-video-on signal to indicate whether the current vertical coordinate is in the displayable region. It is asserted only when the line count is smaller than 480.

5.3.3 TIMING CALCULATION OF VGA SYNCHRONIZATION SIGNALS

As mentioned earlier, we assume that the pixel rate is 25 MHz. It is determined by three parameters:

p: The number of pixels in a horizontal scan line . For 640-by-480 resolution, it is

p=(800*pixels)/line

l: The number of lines in a screen. For 640-by-480 resolution, it is

l=(525*lines)/screen

s: The number of screens per second. For flickering free operation, we can set it to

s=(60*screens)/second

The s parameter specifies how fast the screen should be refreshed. For a human eye, the refresh rate must be at least 30 screens per second to make the motion appear to be continuous. To reduce flickering, the monitor usually has a much higher rate, such as the 60 screens per second specification above. The pixel rate can be calculated by the three parameters :

pixel rate= p * l * s

The pixel rate for other resolutions and refresh rates can be calculated in a similar fashion. Clearly, the rate increases as the resolution and refresh rate grow. The above content was referred from the book written by Pong P Chu (2008).

5.3.4 IMPLEMENTATION OF VGA CONTROLLER

The VGA controller was implemented with the above information.

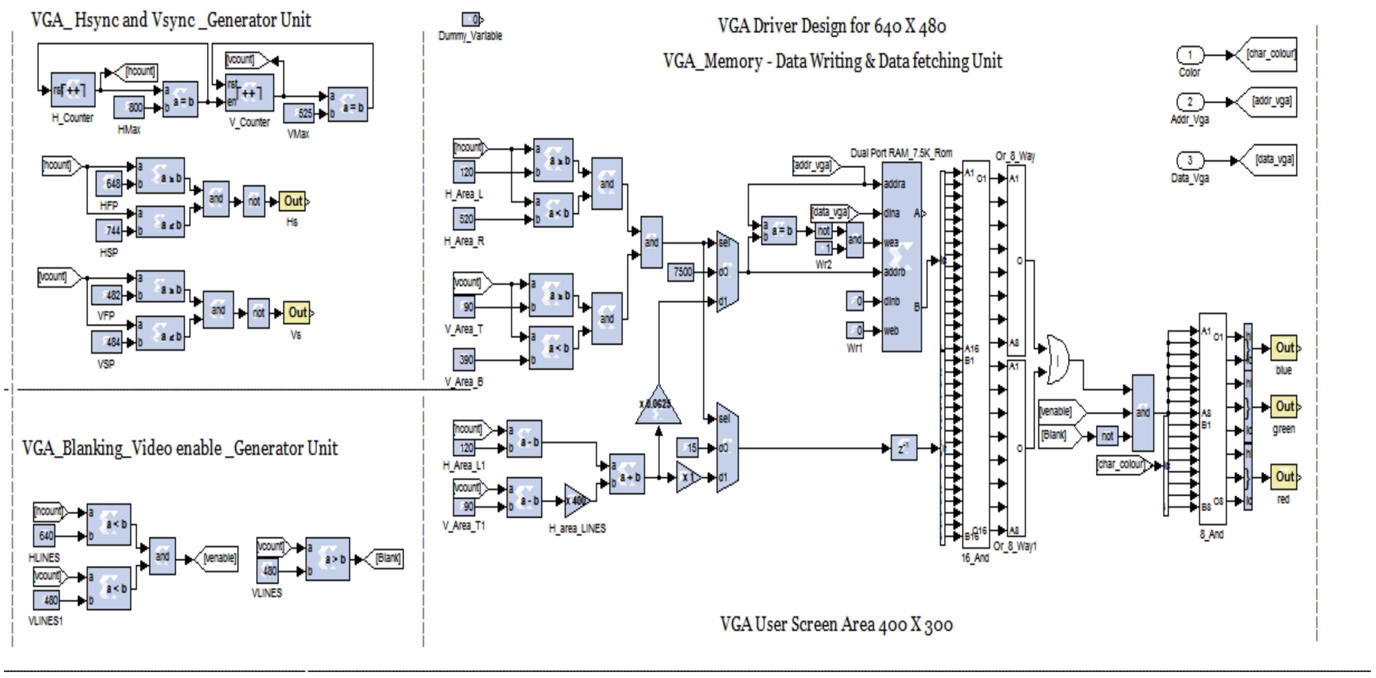


Fig.5.4 Implementation of VGA Driver

5.4 PS2 KEYBOARD

5.4.1 INTRODUCTION

PS2 port was introduced in IBM's Personal Systed2 personnel computers. It is a widely supported interface for a keyboard and mouse to communicate with the host. The PS2 port contains two wires for communication purposes. One wire is for data, which is transmitted in a serial stream. The other wire is for the clock information, which specifies when the data is valid and can be retrieved. The information is transmitted as an 11-bit "packet" that contains a start bit, 8 data bits, an odd parity bit, and a stop bit. The FPGA prototyping board has a PS2 port and acts as a host.

The communication of the PS2 port is bidirectional and the host can send a command to the keyboard or mouse to set certain parameters. For our purposes, the bidirectional communication is hardly required for the PS2 keyboard, and thus our discussion is limited to one direction, from the keyboard to the prototyping board.

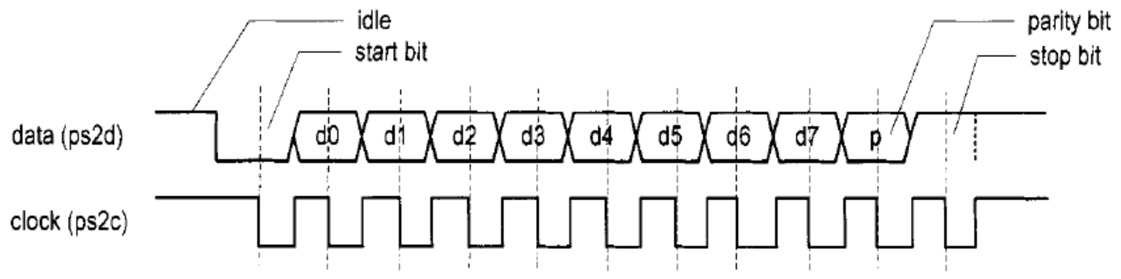


Fig.5.5 Timing Diagram of PS2 port

The above picture was taken from the book written by Pong P Chu (2008).

5.4.2 PHYSICAL INTERFACE OF A PS2 PORT

In addition to data and clock lines, the PS2 port includes connections for power (i.e., Vcc) and ground. The power is supplied by the host. In the original PS2 port, *voltage* is 5 V and the outputs of the data and clock lines are open-collector. However, most current keyboards and mice can work well with 3.3 V. For an older keyboard and mouse, the 5-V supply can be obtained by switching the 52 jumper on the S3 board. The FPGA should still function properly since its I/O pins can tolerate 5-V input.

5.4.3 DEVICE-TO-HOST COMMUNICATION PROTOCOL

A PS2 device and its host communicate via packets. Transmission begins with a start bit, followed by 8 data bits and an odd parity bit, and ends with a stop bit. The falling edge of the ps2c signal indicates that the corresponding bit in the ps2d line is valid and can be retrieved. The clock period of the ps2c signal is between 60 and 100 ps (i.e., 10 kHz to 16.7 kHz), and the ps2d signal is stable at least 5 ps before and after the falling edge of the ps2c signal.

5.4.4 DESIGN AND CODE

The falling-edge of the ps2c signal is used as the reference point to retrieve data. The subsystem includes a falling edge detection circuit, which generates a one-clock-cycle tick at the falling edge of the ps2c signal, and the receiver, which shifts in and assembles the serial bits.

The receiver is initially in the idle state. It includes an additional control signal, rx-en, which is used to enable or disable the receiving operation. The purpose of the signal is to coordinate the bidirectional operation. It can be set to ' 1 ' for the keyboard interface. After the first falling-edge tick and the rx-en signal are asserted, the FSMD shifts in the start bit and moves to the dps state. Since the received data is in fixed format, we shift in the remaining 10 bits in a single state rather than using separate data, parity, and stop states. The FSMD then moves to the load state, in which one extra clock cycle is provided to complete the shifting of the stop bit, and the psrx-done-tick signal is asserted for one clock cycle. The above content was referred from the book written by Pong P Chu (2008).

5.4.5 IMPLEMENTATION OF PS2 KEYBOARD DRIVER

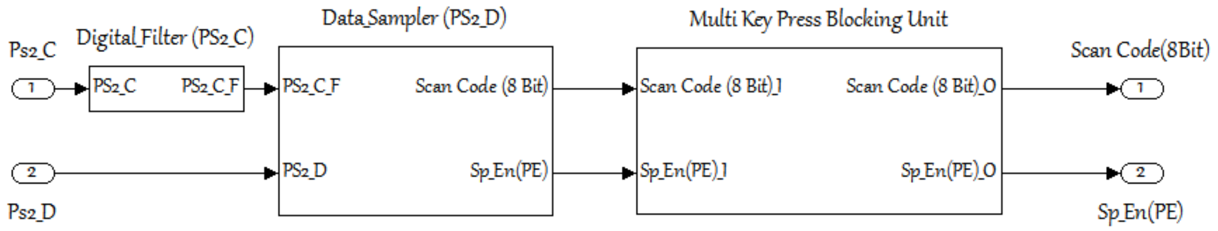


Fig.5.6.1 Implementation of PS2 keyboard Driver

5.4.5.1 DIGITAL FILTER

It is to remove unwanted noise from the required signal containing the scan code. The eight samples of the clock are checked if all samples are 1 or 0. If all the samples are one, then, the 'hi' bit is set and if all the bits are zero, then, the 'lo' bit is set. We know that only either one of the above case is possible at a time. Hence, when all the bits are zero, 'hi' bit is zero and 'lo' bit is one and therefore, 'd1' with zero as input is selected as the output of the multiplexer. Similarly, when all the bits are one, 'hi' bit is one and 'lo' bit is zero and therefore, 'd2' with one as input is selected as the output of multiplexer. This way when the input signal has noise, neither of the conditions exists leading to select 'do' or 'd3' with previous output as the present output of the multiplexer.

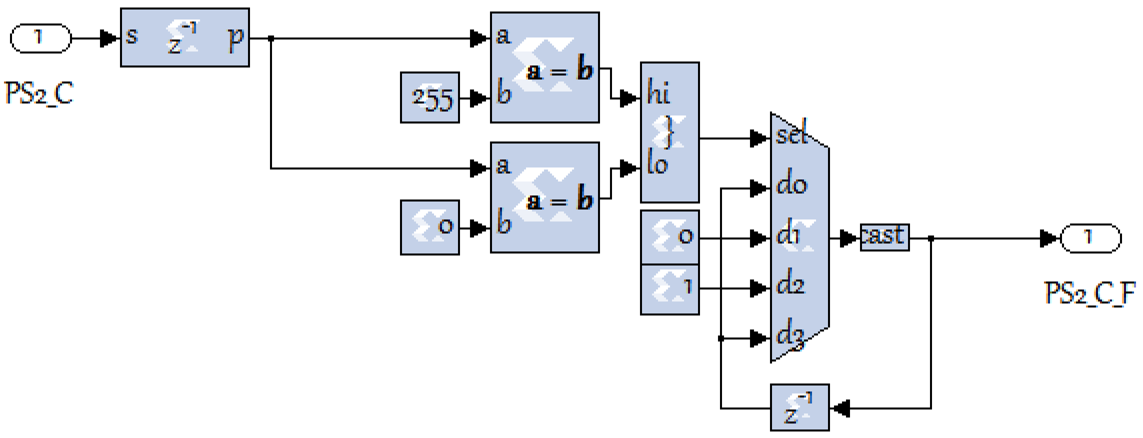


Fig.5.6.2 Implementation of PS2 keyboard Driver

5.4.5.2 DATA SAMPLER

This is primarily to get the scan code from the 11 bit data. The serial to parallel shift register is used to accumulate the 11 bit data. When a negative edge is detected in the clock signal, the data bit counter is incremented and once it reaches eleven, the counter is reset and the scan code is sent on the reception of the next positive edge.

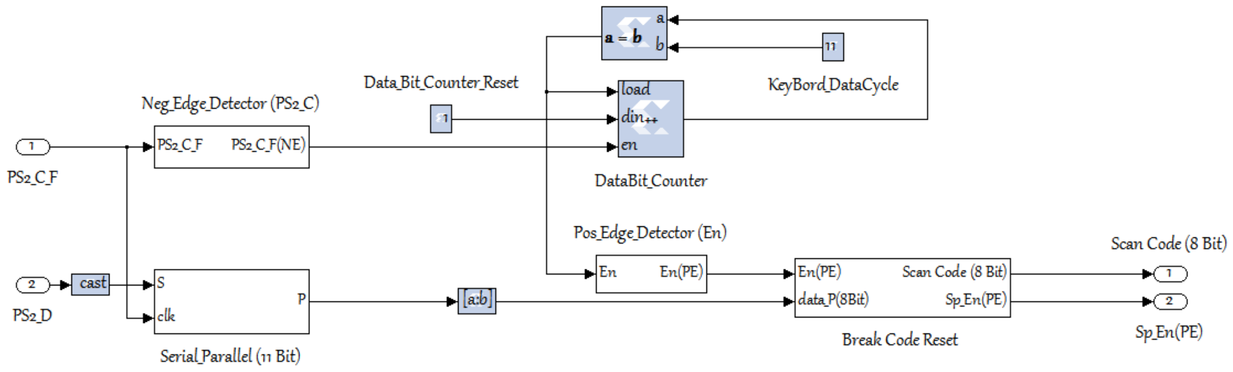


Fig.5.6.3 Implementation of PS2 keyboard Driver

5.4.5.3 MULTI-KEY PRESS BLOCKING UNIT

This is to avoid any ambiguity when two keys are pressed at the same time. When such an event occurs, the below logic ensures that zero gets transmitted as the scan code.

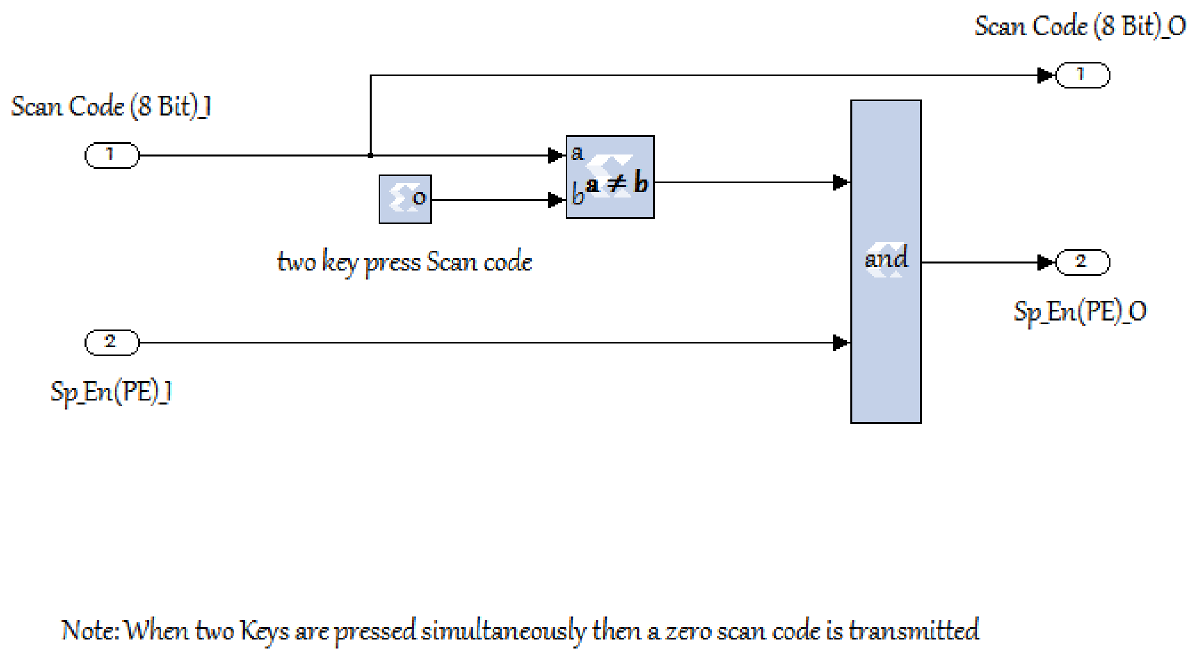


Fig.5.6.4 Implementation of PS2 keyboard Driver

CHAPTER 6

HARDWARE IMPLEMENTATION

The below picture shows the Digilent Nexys 2 FPGA used to implement the soft processor. The soft processor was completely synthesized logically using Xilinx and MATLAB software. The VHDL code generated from the Xilinx software was written on the FPGA to actually implement the soft processor.

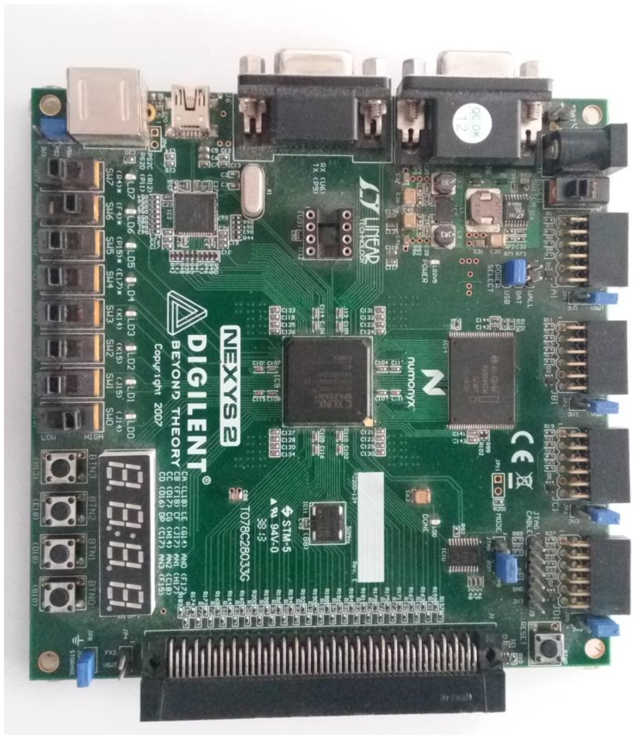


Fig.6.1 Digilent Nexys 2 FPGA

The below picture shows a usual keyboard interfaced with the soft processor using the PS2 keyboard protocol which were referred from the book, Pong P Chu, FPGA Prototyping by VHDL examples. New jersey, John Wiley & Sons Publication,2008

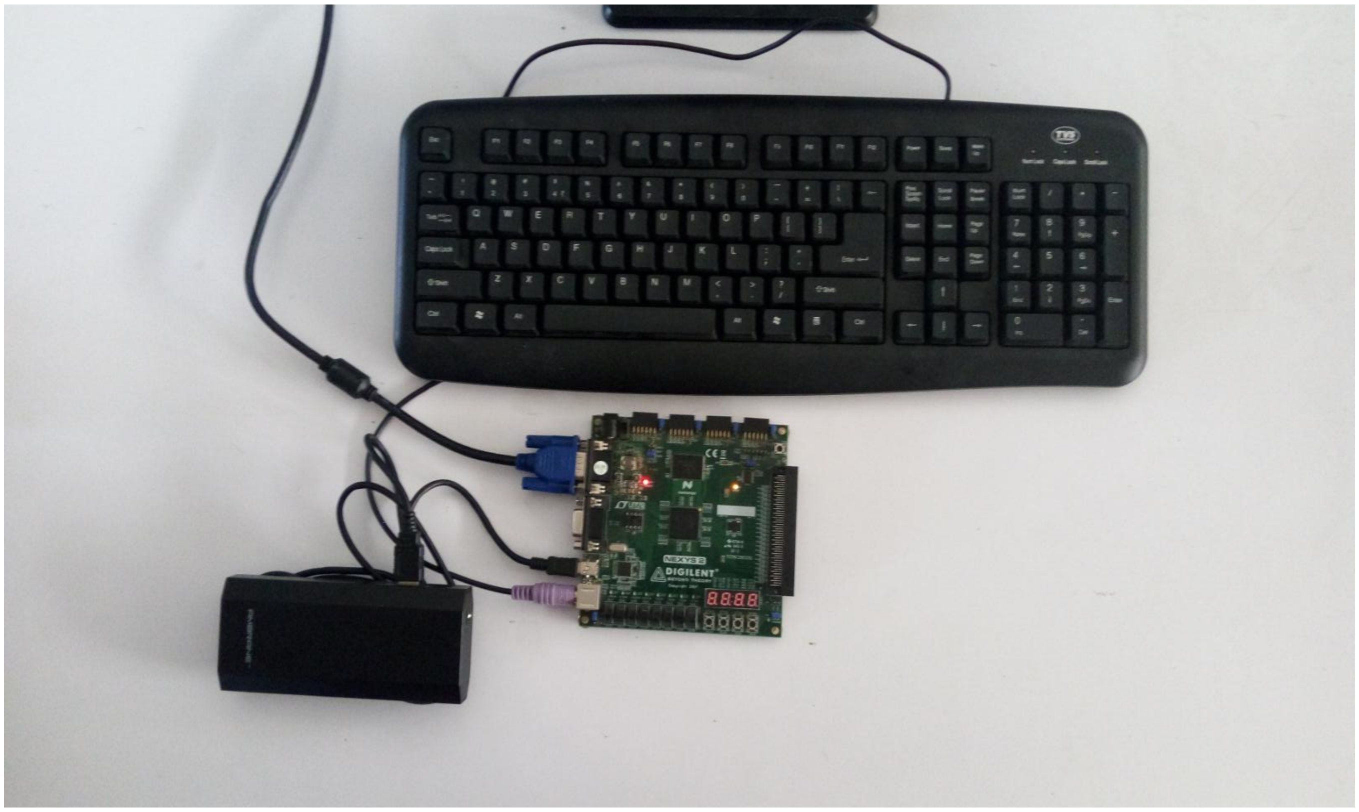
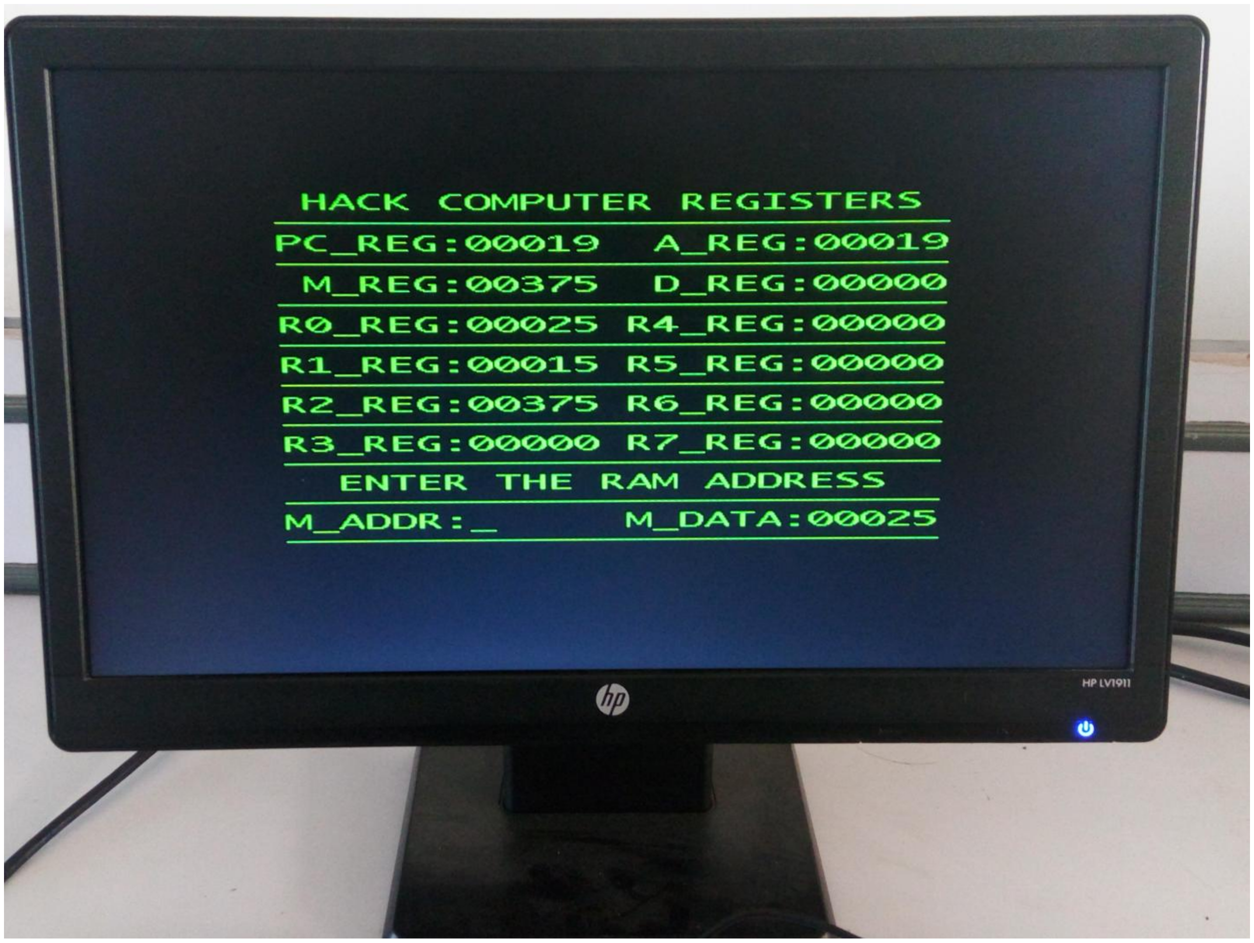


Fig.6.2 Keyboard Interfacing with Soft Processor

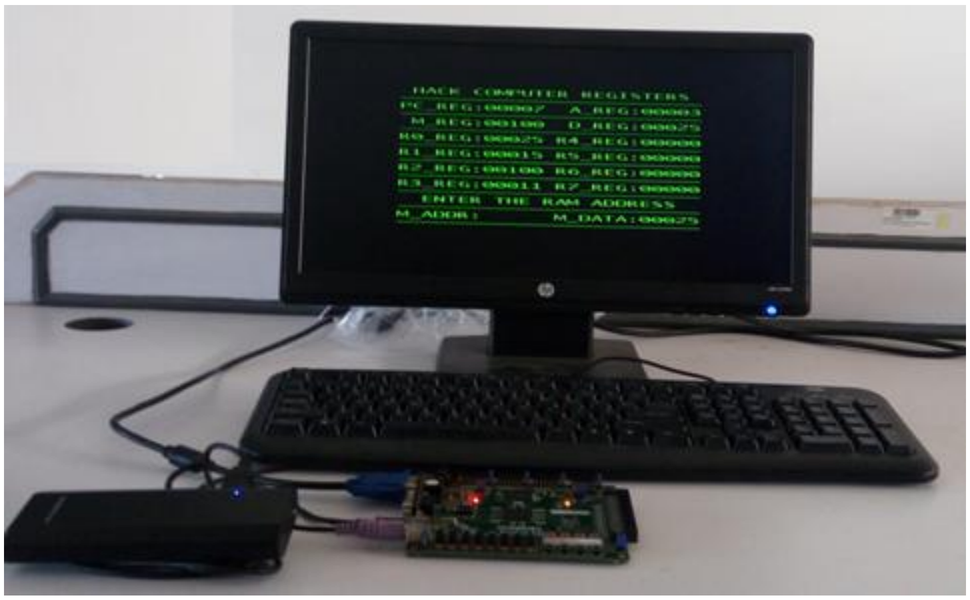
The below picture shows a usual monitor interfaced with the processor. The monitor interfacing was done using the VGA protocol which was referred from the book, Pong P Chu, FPGA Prototyping by VHDL examples. New jersey, John Wiley & Sons Publication, 2008.



```
   HACK  COMPUTER  REGISTERS
PC_REG:00019   A_REG:00019
 M_REG:00375   D_REG:00000
R0_REG:00025 R4_REG:00000
R1_REG:00015 R5_REG:00000
R2_REG:00375 R6_REG:00000
R3_REG:00000 R7_REG:00000
  ENTER  THE  RAM  ADDRESS
M_ADDR:_      M_DATA:00025
```

Fig.6.3 Screen Interfacing with Soft Processor

This picture depicts the entire setup of the soft processor on the FPGA interfaced with the monitor and keyboard.



```
  HACK COMPUTER REGISTERS
PC_REG:00007   A_REG:00003
 M_REG:00100   D_REG:00025
R0_REG:00025 R4_REG:00000
R1_REG:00015 R5_REG:00000
R2_REG:00100 R6_REG:00000
R3_REG:00011 R7_REG:00000
   ENTER THE RAM ADDRESS
M_ADDR:         M_DATA:00025
```

Fig.6.4. Keyboard and Screen interfacing with Soft Processor

The below picture shows an animation graphic developed using the soft processor.

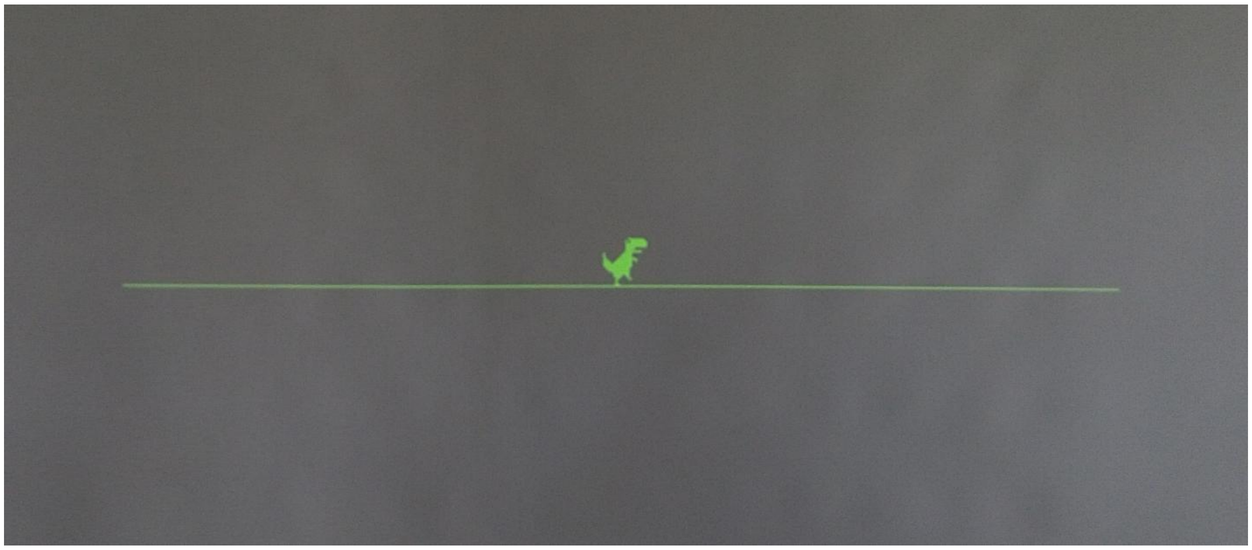


Fig.6.5. Animation graphic developed using soft processor

CHAPTER 7

CONCLUSION AND FUTURE SCOPE

7.1 CONCLUSION

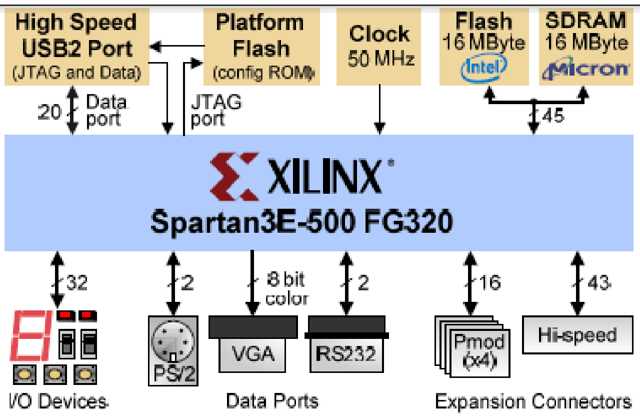
This project provides an insight on the design and implementation of a soft processor. The primary objective is to develop a soft processor that will give a very high degree of flexibility and reconfigurability. The soft processor was purely developed based on logic synthesis. It was implemented on the FPGA. An Assembler was also developed in Python programming language in order to convert assembly language commands into binary commands. The interfacing of the processor with I/O devices like monitor, keyboard was also implemented. A virtual machine too was implemented in order to make the processor independent of the higher level language used for programming the processor.

7.2 FUTURE SCOPE

The project can be extended to run as many as ten soft processors on a single FPGA parallel. We can design and implement compiler for a particular language so that we can directly program our processor in that language. We can also define and develop a set of libraries with certain functions like math, memory etc... to provide a better GUI and make the programming easier. Currently, the program that runs in the soft processor is synthesized along with soft processor. This prevents online modification of the program. This can be overcome by providing serial programming interface to modify the program in the instruction memory of soft processor.

APPENDIX-I

The Nexys2 circuit board is a complete, ready-to-use circuit development platform based on a Xilinx Spartan 3E FPGA. Its on-board high-speed USB2 port, 16Mbytes of RAM and ROM, and several I/O devices and ports make it an ideal platform for digital systems of all kinds, including embedded processor systems based on Xilinx's MicroBlaze. The USB2 port provides board power and a programming interface, so the Nexys2 board can be used with a notebook computer to create a truly portable design station.



The Nexys2 brings leading technologies to a platform that anyone can use to gain digital design experience. It can host countless FPGA-based digital systems, and designs can easily grow beyond the board using any or all of the five expansion connectors. Four 12-pin Peripheral Module (Pmod) connectors can accommodate up to eight low-cost Pmods to add features like motor control, A/D and D/A conversion, audio circuits, and a host of sensor and actuator interfaces. All user-accessible signals on the Nexys2 board are ESD and short-circuit protected, ensuring a long operating life in any environment.

- *500K-gate Xilinx Spartan 3E FPGA*
- *USB2-based FPGA configuration and high-speed data transfers (using the free Adept Suite Software)*
- *USB-powered (batteries and/or wall-plug can also be used)*
- *16MB of Micron PSDRAM &16MB of Intel StrataFlash ROM*
- *Xilinx Platform Flash for nonvolatile FPGA configurations*
- *Efficient switch-mode power supplies (good for battery powered applications)*
- *50MHz oscillator plus socket for second oscillator*
- *60 FPGA I/O's routed to expansion connectors (one high-speed Hirose FX2 connector and four 6-pin headers)*
- *8 LEDs, 4-digit 7-seg display, 4 buttons, 8 slide switches*
- *Ships in a plastic carry case with USB cable*

PS/2 Port

The 6-pin mini-DIN connector can accommodate a PS/2 mouse or keyboard. Most PS/2 devices can operate from a 3.3V supply, but older devices may require a 5VDC supply. A three-pin jumper on the Nexys2 board immediately adjacent to the PS/2 connector selects whether regulated 3.3V or the main input power bus voltage (VU) is supplied to the PS/2 connector. To send 5V to the PS/2 connector, set the PS2 power jumper to Vswt (the main input power bus), and ensure the board is powered from USB or a 5VDC wall-plug supply. To send 3.3V to the connector, set the jumper to 3.3V.

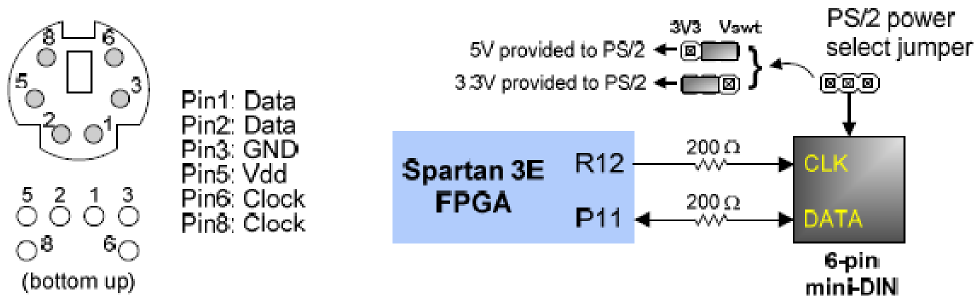
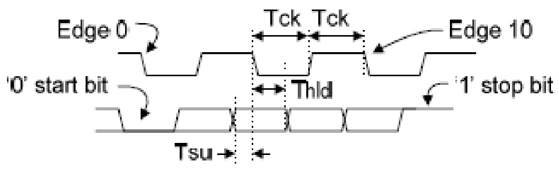


Figure 12: Nexys2 PS/2 circuits

Both the mouse and keyboard use a two-wire serial bus (clock and data) to communicate with a host device. Both use 11-bit words that include a start, stop and odd parity bit, but the data packets are organized differently, and the keyboard interface allows bi-directional data transfers (so the host device can illuminate state LEDs on the keyboard). Bus timings are shown in the figure. The clock and data signals are only driven when data transfers occur, and otherwise they are held in the "idle" state at logic '1'. The timings define signal requirements for mouse-to-host communications and bi-directional keyboard communications. A PS/2 interface circuit can be implemented in the FPGA to create a keyboard or mouse interface.



Symbol	Parameter	Min	Max
T_{CK}	Clock time	30us	50us
T_{SU}	Data-to-clock setup time	5us	25us
T_{HLD}	Clock-to-data hold time	5us	25us

Figure 13: PS/2 signal timings

Keyboard

The keyboard uses open-collector drivers so the keyboard or an attached host device can drive the two-wire bus (if the host device will not send data to the keyboard, then the host can use input-only ports).

PS2-style keyboards use scan codes to communicate key press data. Each key is assigned a code that is sent whenever the key is pressed; if the key is held down, the scan code will be sent repeatedly about once every 100ms. When a key is released, a "F0" key-up code is sent, followed by the scan code of the released key. If a key can be "shifted" to produce a new character (like a capital letter), then a shift character is sent in addition to the scan code, and the host must determine which ASCII character to use. Some keys, called extended keys, send an "E0" ahead of the scan code (and they may send more than one scan code). When an extended key is released, an "E0 F0" key-up code is sent, followed by the scan code. Scan codes for most keys are shown in the figure. A host device can also send data to the keyboard. Below is a short list of some common commands a host might send.

ED Set Num Lock, Caps Lock, and Scroll Lock LEDs. Keyboard returns "FA" after receiving "ED", then host sends a byte to set LED status: Bit 0 sets Scroll Lock; bit 1 sets Num Lock; and Bit 2

EE	Echo (test). Keyboard returns “EE” after receiving “EE”.
F3	Set scan code repeat rate. Keyboard returns “F3” on receiving “FA”, then host sends second byte to set the repeat rate.
FE	Resend. “FE” directs keyboard to re-send most recent scan code.
FF	Reset. Resets the keyboard.

The keyboard can send data to the host only when both the data and clock lines are high (or idle). Since the host is the “bus master”, the keyboard must check to see whether the host is sending data before driving the bus. To facilitate this, the clock line is used as a “clear to send” signal. If the host pulls the clock line low, the keyboard must not send any data until the clock is released. The keyboard sends data to the host in 11-bit words that contain a ‘0’ start bit, followed by 8-bits of scan code (LSB first), followed by an odd parity bit and terminated with a ‘1’ stop bit. The keyboard generates 11 clock transitions (at around 20 - 30KHz) when the data is sent, and data is valid on the falling edge of the clock.

Scan codes for most PS/2 keys are shown in the figure below.

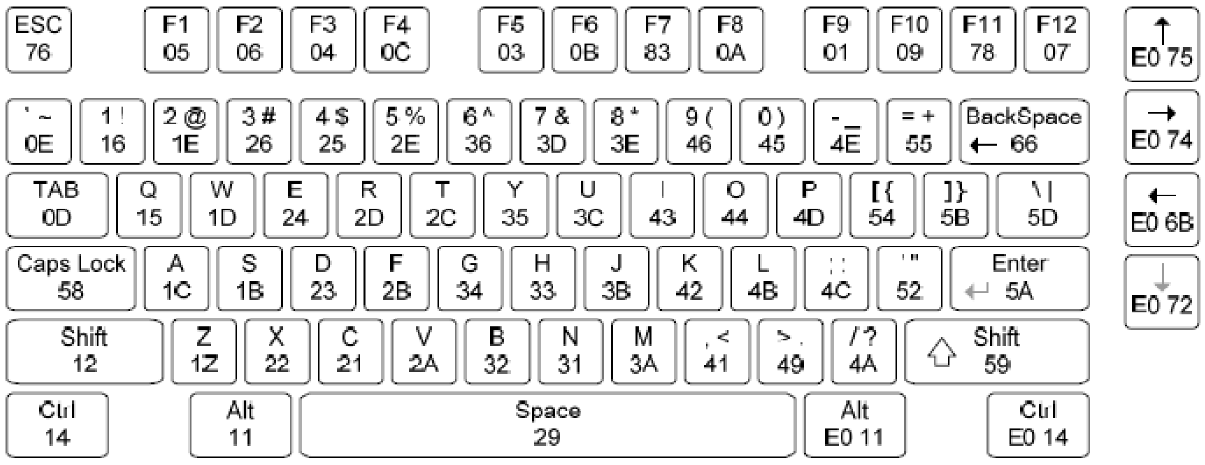


Figure 14: PS/2 keyboard scan codes

VGA Port

The Nexys2 board uses 10 FPGA signals to create a VGA port with 8-bit color and the two standard sync signals (HS – Horizontal Sync, and VS – Vertical Sync). The color signals use resistor-divider circuits that work in conjunction with the 75-ohm termination resistance of the VGA display to create eight signal levels on the red and green VGA signals, and four on blue (the human eye is less sensitive to blue levels). This circuit, shown in figure 13, produces video color signals that proceed in equal increments between 0V (fully off) and 0.7V (fully on). Using this circuit, 256 different colors can be displayed, one for each unique 8-bit pattern. A video controller circuit must be created in the FPGA to drive the sync and color signals with the correct timing in order to produce a working display system.

VGA System Timing

VGA signal timings are specified, published, copyrighted and sold by the VESA organization (www.vesa.org). The following VGA system timing information is provided as an example of how a VGA monitor might be driven in 640 by 480 mode. For more precise information, or for information on other VGA frequencies, refer to documentation available at the VESA website.

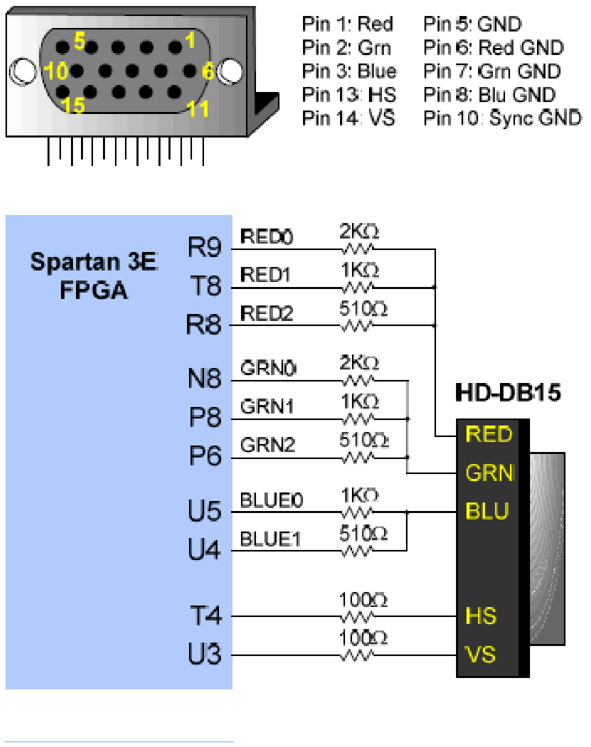
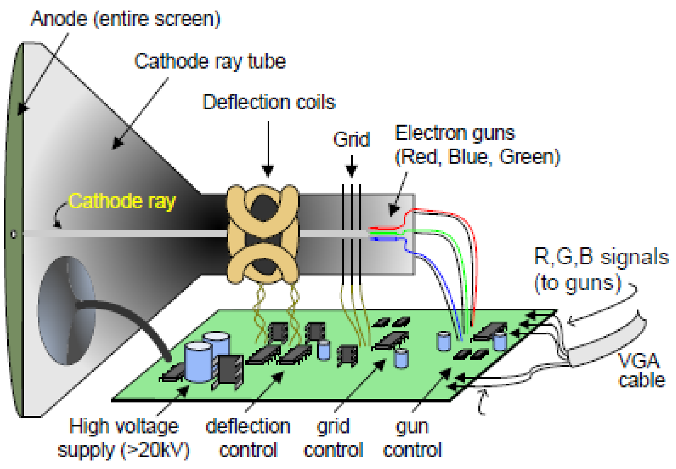


Figure 16: VGA pin definitions and Nexys2 circuit

CRT-based VGA displays use amplitude-modulated moving electron beams (or cathode rays) to display information on a phosphor-coated screen. LCD displays use an array of switches that can impose a voltage across a small amount of liquid crystal, thereby changing light permittivity through the crystal on a pixel-by-pixel basis. Although the following description is limited to CRT displays, LCD displays have evolved to use the same signal timings as CRT displays (so the "signals" discussion below pertains to both CRTs and LCDs). Color CRT displays use three electron beams (one for red, one for blue, and one for green) to energize the phosphor that coats the inner side of the display end of a cathode ray tube (see illustration). Electron beams emanate from "electron guns" which are finely-pointed heated cathodes placed in close proximity to a positively charged annular plate called



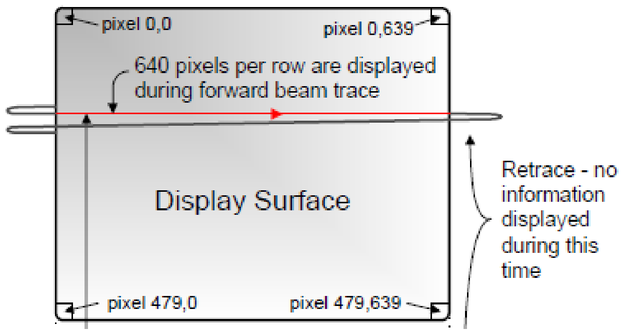
a "grid". The electrostatic force imposed by the grid pulls rays of energized electrons

Figure 17: CRT deflection system

from the cathodes, and those rays are fed by the current that flows into the cathodes. These particle rays are initially accelerated towards the grid, but they soon fall under the influence of the much larger electrostatic force that results from the entire phosphor-coated display surface of the CRT being charged to 20kV (or more). The rays are focused to a fine beam as they pass through the center of the grids, and then they accelerate to impact on the phosphor-coated display surface. The phosphor surface glows brightly at the impact point, and it continues to glow for several hundred microseconds after the beam is removed. The larger the current fed into the cathode, the brighter the phosphor will glow.

Between the grid and the display surface, the beam passes through the neck of the CRT where two coils of wire produce orthogonal electromagnetic fields. Because cathode rays are composed of charged particles (electrons), they can be deflected by these magnetic fields. Current waveforms are passed through the coils to produce magnetic fields that interact with the cathode rays and cause them to transverse the display surface in a "raster" pattern, horizontally from left to right and vertically from top to bottom. As the cathode ray moves over the surface of the display, the current sent to the electron guns can be increased or decreased to change the brightness of the display at the cathode ray impact point.

Information is only displayed when the beam is moving in the "forward" direction (left to right and top to bottom), and not during the time the beam is reset back to the left or top edge of the display. Much of the potential display time is therefore lost in "blanking" periods when the beam is reset and stabilized to begin a new horizontal or vertical display pass. The size of the beams, the frequency at which the beam can be traced across the display, and the frequency at which the electron beam can be modulated determine the display resolution. Modern VGA displays can accommodate different resolutions, and a VGA controller circuit dictates the resolution by producing timing signals to control the raster patterns. The controller must produce synchronizing pulses at 3.3V (or 5V) to set the frequency at which current flows through the deflection coils, and it must ensure that video data is applied to the electron guns at the


pixel 0,0
pixel 0,639
640 pixels per row are displayed during forward beam trace
Display Surface
Retrace - no information displayed during this time
pixel 479,0
pixel 479,639

correct time. Raster video displays define a number of "rows" that corresponds to the number of horizontal passes the cathode makes over the display area, and a number of "columns" that corresponds to an area on each row that is assigned to one "picture element" or pixel. Typical displays use from 240 to 1200 rows and from 320 to 1600 columns. The overall size of a display and the number of rows and columns determines the size of each pixel.

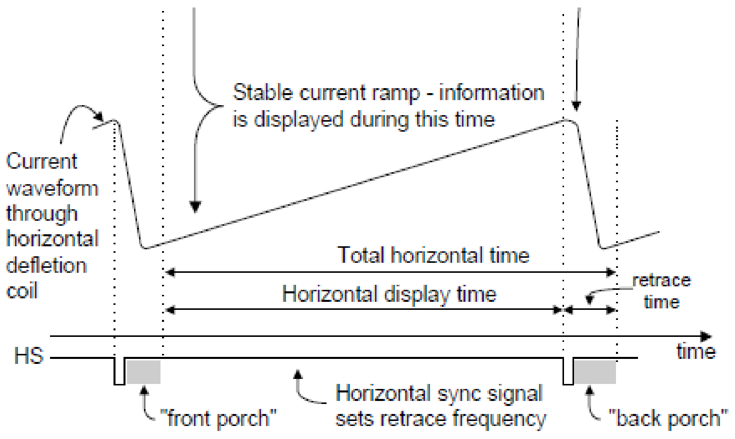
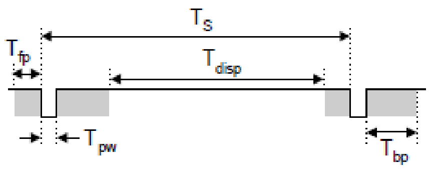

Current waveform through horizontal defletion coil
Stable current ramp - information is displayed during this time
Total horizontal time
Horizontal display time
retrace time
HS
time
"front porch"
Horizontal sync signal sets retrace frequency
"back porch"

Figure 18: VGA system signals

Video data typically comes from a video refresh memory, with one or more bytes assigned to each pixel location (the Nexys2 uses three bits per pixel). The controller must index into video memory as the beams move across the display, and retrieve and apply video data to the display at precisely the time the electron beam is moving across a given pixel.

A VGA controller circuit must generate the HS and VS timings signals and coordinate the delivery of video data based on the pixel clock. The pixel clock defines the time available to display one pixel of information. The VS signal defines the "refresh" frequency of the display, or the frequency at which all information on the display is redrawn. The minimum refresh frequency is a function of the display's phosphor and electron beam intensity, with practical refresh frequencies falling in the 50Hz to 120Hz range. The number of lines to be displayed at a given refresh frequency defines the horizontal "retrace" frequency. For a 640-pixel by 480-row display using a 25MHz pixel clock and 60 +/-1Hz refresh, the signal timings shown in the table at right can be derived. Timings for sync pulse width and front and back porch intervals (porch intervals are the pre- and post-sync pulse times during which information cannot be displayed) are based on observations taken from actual VGA displays.



Symbol	Parameter	Vertical Sync			Horiz. Sync	
		Time	Clocks	Lines	Time	Clks
T_S	Sync pulse	16.7ms	416,800	521	32 us	800
T_{disp}	Display time	15.36ms	384,000	480	25.6 us	640
T_{pw}	Pulse width	64 us	1,600	2	3.84 us	96
T_{fp}	Front porch	320 us	8,000	10	640 ns	16
T_{bp}	Back porch	928 us	23,200	29	1.92 us	48

Figure 19: VGA system timings for 640x480 display

A VGA controller circuit decodes the output of a horizontal-sync counter driven by the pixel clock to generate HS signal timings. This counter can be used to locate any pixel location on a given row. Likewise, the output of a vertical-sync counter that increments with each HS pulse can be used to generate VS signal timings, and this counter can be used to locate any given row. These two continually running counters can be used to form an address into video RAM. No time relationship between the onset of the HS pulse and the onset of the VS pulse is specified, so the designer can arrange the counters to easily form video RAM addresses, or to minimize decoding logic for sync pulse generation.

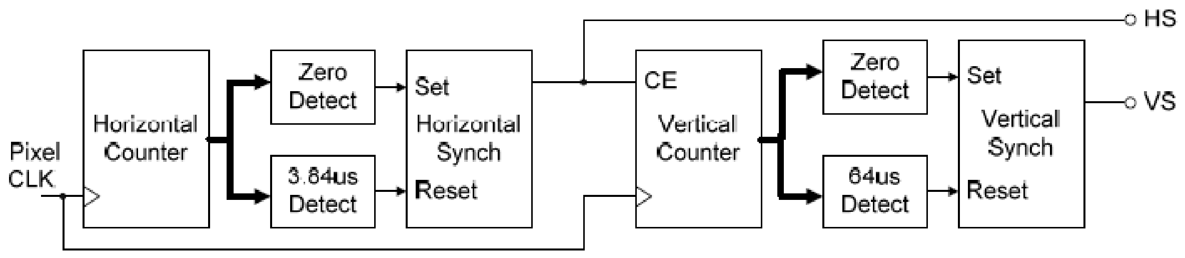


Figure 20: Schematic for a VGA controller circuit

REFERENCES

1. Jason, G. Tong, Ian, D. L. Anderson and Mohammed ,A. S. Khalid(2006),'Soft Core Processors for Embedded Systems', International Conference on Microelectronics.

2. Noam Nisan and Shimon Schocken (2005),' The Elements of Computing Systems', Cambridge, MIT press.

3. Peter Yiannacouras, Gregory Steffan, J. and Jonathan Rose (February 2007),' Exploration and Customization of FPGA-Based Soft Processors', IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 26, No. 2, pp. 266-277.

4. Pong P Chu(2008), 'FPGA Prototyping by VHDL examples', New jersey, John Wiley & Sons Publication.