# Processor Design: From Specification to Gate-Level Netlist Realization

*Project report submitted in partial fulfillment of the requirement for the degree of*

Bachelor of Technology

Submitted by

Aditi Sharma (2010110034)

And

Devyam Seal (2010110745)

Under Supervision of

Dr. Venkatnarayan Hariharan
Department of Electrical Engineering

**Department of Electrical Engineering**
**School of Engineering**
**Shiv Nadar Institution of Eminence**

(April 2024)

# Candidate Declaration

We hereby declare that the thesis entitled "Processor Design: From Specification to Gate-Level Netlist Realization" submitted for the B. Tech. degree program. This thesis has been written in our own words. We have adequately cited and referenced the original sources.

| | |
|---|---|
| Aditi Sharma | Devyam Seal |
| (2010110034) | (2010110745) |

# Certificate

It is certified that the work contained in the project report titled "Processor Design: From Specification to Gate-Level Netlist Realization," by "Aditi Sharma" has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Venkatnarayan Hariharan
Dept. of Electrical Engineering
School of Engineering
Shiv Nadar Institution of Eminence
Date: 29/04/2024

# Certificate

It is certified that the work contained in the project report titled “Processor Design; From Specification to Gate-Level Netlist Realization,” by “Devyam Seal” has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Dr. Venkatnarayan Hariharan
Dept. of Electrical Engineering
School of Engineering
Shiv Nadar Institution of Eminence
Date: 29/04/2024

# Abstract

Microprocessors act as the central elements of modern computing systems and understanding their design principles is crucial for the progression of computer architecture. This project revolves around designing an 8-bit microprocessor, utilizing a gradual approach that addresses the different phases of microprocessor designing.

Starting with an examination of the architectural specifications, the project addresses the operational prerequisites and abilities that the 8-bit microprocessor should possess. These prerequisites establish the groundwork for subsequent design choices, steering the project towards an adept microprocessor framework.

The implementation stage stands as the focal point of this project, involving the transformation of the conceptual design into a digital circuit. After finalizing the ISA specification, this phase involves RTL (Register-Transfer Level) design, where the microprocessor's components are described using Verilog. Addressing timing challenges that arise, the project relies on rigorous testing and simulation techniques to ensure accurate functioning. Furthermore, the project delves into hardware intricacies by ensuring that the code is synthesizable and can be burned on an FPGA board. This helps evaluate the impact on timing, temperature, and power, providing insights into practical implications and trade-offs.

The project provides an enlightening perspective into the realm of microprocessor designing and its hands-on realization, delivering a deep understanding of the steps involved in designing the core of computing systems.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Microprocessors are the brains of a wide range of modern computing devices, including laptops, smartphones etc. Microprocessor designs have evolved to a stage today where they have complex features like pipelining, branch-prediction, out-of-order execution, multi-core, hyperthreading, data and instruction level parallelism, etc. Learning all these at an undergraduate level is an arduous task, and it is also a fact that mere theoretical learning without hands-on implementation gives one a false sense of comfort. Thus, it is noteworthy if a hands-on implementation, even of a simple design at that, is attempted. This project builds on our previously designed microprocessor and maps it out on a Nexys A7 FPGA board. This implementation enabled the analysis of the performance of the microprocessor, including both timing and power consumption reports. To start with, simple existing microprocessors were studied, such as the 8085 and RNBIP. The 8085 is an 8-bit microprocessor designed by Intel with 6 general-purpose registers which can also be used as 3 distinct register pairs. The overall design of the microprocessor represents a substantial upgrade from the 8085 microprocessor, owed to its RISC architecture, increased memory capacity, and reduced hardware module requirements for instruction execution. Starting with a short description of the microprocessor, the work dives into specifications of the FPGA board used and the methods used to map the input and output of the microprocessor to the board and the VGA display. The implementation stage stands as the focal point of this work. It involves using switches as data for address and opcode values and buttons to enter them into the memory. Another button is used to start the clock for the microprocessor.

## 1.1 BASIC ARCHITECTURE:

A microprocessor is a combination of various components integrated to make one functional circuit. The choice of the blocks along with their specifications need to be tailored to the specific instruction set. We have intricately chosen our components along with the sizes and connections between those components for efficient execution of our specific instruction set.

## 1.2 IMPORTANT FUNCTIONAL BLOCKS:

Few of the essential functional blocks are covered here. Along with these, there are a few more fundamental blocks which can be seen in the block diagram.

1. **Control Unit:**

   The Control Unit is the head of the processor responsible for sending all the control signals necessary for the smooth functioning of the circuit. It enables only the required data paths for any given instruction and blocks off the rest. For this purpose, the Instruction register sends the 8-bit opcode stored in it directly to the Control Unit.

2. **Arithmetic and Logical Unit (ALU):**

   The ALU is responsible for handling all instructions that require performing various arithmetic and logical operations. It operates on at most two variables for any given instruction. The values for these variables come from either the Operand Register or the Accumulator or the Register Array.

3. **Data and Address Buses:**

   Data and Address Buses help transport data from one block to another. The size of a microprocessor essentially depends on the size of the data bus. For our microprocessor, we are using an 8-bit Data Bus and a 16-bit address bus.

4. **Register Array:**

   Registers are necessary to store data upon which various operations can be performed. The more the number of registers the more data we can utilize at any point. We have eight 8-bit general purpose registers (excluding the accumulator) namely, B, C, D, E, F, G, H, I. All of these can also be used in pairs to store 16-bit memory address locations.

5. **Memory:**

   Since we're using a 16-bit address bus, we can use up to 65,356 memory locations, i.e., we have a 64kB memory. This has been divided into three parts, namely, General Purpose Memory (30kB), Program Memory(30kB) and a 4kB Stack memory with 16-bit blocks.

6. **Flag Register:**

   The flag register contains some of the most fundamental but crucial flags, namely, carry, zero, sign and parity flags.

# Chapter 2
# Literature Survey

## Basics of Architecture and Design:

We have referred to a standard book from the beginning of the project to build our basics and clear up any of our doubts. ‘Computer Organization and Design RISC V Edition’ by David A. Patterson and John L. Hennessy has been our guide throughout the project.

## Learning Verilog:

We attended a training program at IIIT Allahabad where we learnt the basics of Verilog. We learnt about the Behavioral, Structural and Data Flow models. We also did a few FSM based simple projects. We ended it by designing an ALU which motivated us to choose this project. Along with this, we also decided to attend an FPGA based course this semester where we are learning how to implement our Verilog code on an FPGA board.

## Previously used Microprocessors and their differences:

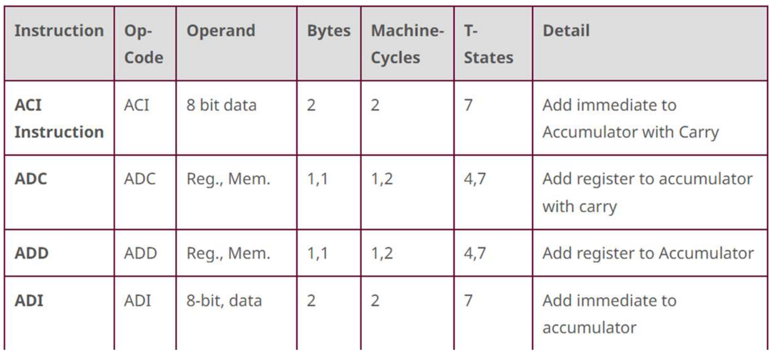
We have worked with two 8-bit microprocessors for our previous courses, namely, 8085 Microprocessor and RNBIP. We’ve taken inspiration from both while designing our microprocessor. We have extensively studied the architecture and functioning of these microprocessors and analyzed their differences while deciding what to incorporate or change in our own. We took inspiration from 8085 for using register pairs to store addresses but we also made it possible for all registers to work as pairs to make it more versatile. RNBIP does not have any 2 register instructions. We designed our opcode in a way that makes it possible for two register instructions to be incorporated. We have incorporated some of what we have learnt from these microprocessors but also made it our own. One of the key differences would be that our processor leans a lot towards being RISC which is the current industry standard.

## 2.1 RISC VS CISC ARCHITECTURE

CISC and RISC are two types of Instruction Set architectures. Comparatively, RISC is known to have a smaller number of instructions which can be combined to perform more complex operations. On the other hand, CISC contains complex and more in-depth instructions. CISC focuses on a smaller number of instructions needed to perform a specific task while RISC focuses more on the efficiency of the processor. RISC ensures that each instruction gets executed in approximately 1 clock cycle each which saves a lot more time as compared to its CISC contemporaries. After research on current industry standards, we also found that CISC computers in today's world have become almost obsolete and most modern-day processors run on RISC. We also plan on implementing a pipeline structure in the next semester which is only possible in RISC architectures which is why we decided to go with a RISC based microprocessor.

## 2.2 CLOCK CYCLES OF 8085 MICROPROCESSOR

Every instruction cycle (Instruction Fetch, Operand Fetch, Execution Cycle) takes more than one clock cycle to complete. 8085 uses a maximum of 7 clock cycles (or T states) to finish executing an instruction. This helped us understand that separating the cycles is more important than just reducing the number of cycles. Our microprocessor takes a maximum of 6 clock cycles or T states to execute any instruction. The added states allow a buffer time for data to update accurately and on time for the next clock.



| Instruction | Op-Code | Operand | Bytes | Machine-Cycles | T-States | Detail |
|---|---|---|---|---|---|---|
| **ACI Instruction** | ACI | 8 bit data | 2 | 2 | 7 | Add immediate to Accumulator with Carry |
| **ADC** | ADC | Reg., Mem. | 1,1 | 1,2 | 4,7 | Add register to accumulator with carry |
| **ADD** | ADD | Reg., Mem. | 1,1 | 1,2 | 4,7 | Add register to Accumulator |
| **ADI** | ADI | 8-bit, data | 2 | 2 | 7 | Add immediate to accumulator |

**Table 2.1 Instructions and T-States of 8085 Microprocessor**

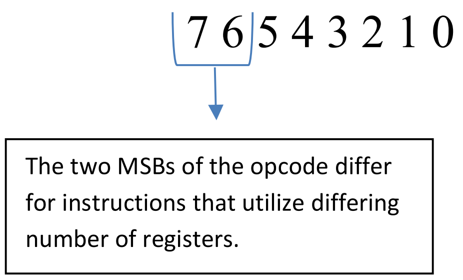
# Chapter 3

# Work Done

## 3.1 INSTRUCTION SET ARCHITECTURE

### 3.1.1 Opcode Format

Firstly, the opcode format was described. The main motive behind this was for the microprocessor to be able to execute a few important two register instructions. For a total of 8 registers, the opcode format was designed in such a way that zero, single and two register instructions could be easily distinguished. The final opcode format is as follows:



The two MSBs of the opcode differ for instructions that utilize differing number of registers.

**A. Zero Register Instructions:**

- **[7 6] – 00** - The first two bits are 00.
- **[5 4 3 2 1 0]** – The rest of the bits are different for different instructions but might have a pattern associated with different subsections of the instruction set.

**B. One Register Instructions:**

- **[7 6] – 01/10 –** The first two bits are either 01 or 10.
- **[5 4 3] – instruction** – The next three bits differ for different instructions.
- **[2 1 0] - <rn> -** The last three bits differ for different registers based on the table below:

| **Register Name** | **Address** |
|---|---|
| B | 000 |
| C | 001 |
| D | 010 |
| E | 011 |
| F | 100 |
| G | 101 |
| H | 110 |
| I | 111 |

**Table 3.1 Register Addresses**

## C. Two Register Instructions:

- **[7 6] – 11 –** The first two bits are 11.
- **[5]** – differs for different instructions.
- **[4 3 2 1 0] - <rp> -** differs for different register pairs based on the table below:

| **Register Pair** | **Associated Bits** |
|---|---|
| BC | 00000 |
| BD | 00001 |
| BE | 00010 |
| BF | 00011 |
| BG | 00100 |
| BH | 00101 |
| BI | 00110 |
| CD | 00111 |
| CE | 01000 |
| CF | 01001 |
| CG | 01010 |
| CH | 01011 |
| CI | 01100 |
| DE | 01101 |
| DF | 01110 |
| DG | 01111 |
| DH | 10000 |
| DI | 10001 |
| EF | 10010 |
| EG | 10011 |
| EH | 10100 |
| EI | 10101 |
| FG | 10110 |
| FH | 10111 |
| FI | 11000 |
| GH | 11001 |
| GI | 11010 |
| HI | 11011 |

**Table 3.2 Register Pairs**

- For 8 registers, we would have ${}^{8}C_{2} = 28$ possible ordered pairs. We would require at least 5 bits to store these.

### 3.1.2 Instruction Set

An initial set of instructions was finalized with the appropriate sub sections. Instructions that could be executed by using one or more of the already existing essential instructions were eliminated resulting in a significantly reduced instruction set. The opcodes were modified and the order of the instructions in the set was changed with the intention of making the code more concise.

The final instruction set:

| S.No | INSTRUCTION | OPERATION | OPCODE | Control Bits (Execute) |
|---|---|---|---|---|
| | | **Zero register Instructions** | | |
| **A) Flags and Machine Control Instructions** | | | | |
| 1. | CLR | [<rn>](n = 0, 1, …7) ← 0 (all registers cleared) | 00000000 | $WA_{RN}$ |
| 2. | CLF | fl ← 0 | 00000011 | $CL_{FR}$ |
| **B) Data Transfer Instructions** | | | | |
| 3. | LDI <od> | [A] ← <od> | 00000100 | $R_{OR}$, $W_{A}$ |
| 4. | STI | [<od>] ← [A] | 00000101 | $R_{A}$, $W_{OR}$ |
| **C) Logical Instructions** | | | | |
| 5. | RTL <od> | Rotate [A] left by <od> bits | 00000110 | $W_{A}$ |
| 6. | RTR <od> | Rotate [A] Right by <od> bits | 00000111 | $W_{A}$ |
| 7. | CPI <od> | Compare [A] with <od> | **00001**000 | $W_{A}$ |
| 8. | ANI <od> | [A]← [A] & <od> | **00001**001 | $W_{A}$ |
| 9. | ORI <od> | [A]← [A] \| <od> | **00001**010 | $W_{A}$ |
| 10. | XRI <od> | [A]← [A] ^ <od> | **00001**011 | $W_{A}$ |
| 11. | CMA | [A] ← ~[A] | **00001**100 | $W_{A}$ |
| **D) Arithmetic Instructions** | | | | |
| 12. | ADI <od> | [A]←[A] + <od> with carry | **00001**101 | $W_{A}$ |
| 13. | SBI <od> | [A]←[A]-<od> with borrow | **00001**110 | $W_{A}$ |
| **E) Branch Control Instructions** | | | | |
| 14. | JMP <ad> | PC ← <ad> | 00010000 | $I2_{PC}$, $I_{SP}$, $R_{PC}$, $WR_{S}$, $RD_{AB}$, $W_{PC}$ |
| 15. | JNC <ad> | If Carry fl = 0, PC ← <ad> | 00010001 | $I2_{PC}$, $I_{SP}$, $R_{PC}$, $WR_{S}$, $RD_{AB}$, $W_{PC}$ |
| 16. | JNZ <ad> | If zero fl = 0, PC ← <ad> | 00010010 | $I2_{PC}$, $I_{SP}$, $R_{PC}$, $WR_{S}$, $RD_{AB}$, $W_{PC}$ |
| 17. | JNS <ad> | If sign fl = 0, PC ← <ad> | 00010011 | $I2_{PC}$, $I_{SP}$, $R_{PC}$, $WR_{S}$, $RD_{AB}$, $W_{PC}$ |
| 18. | JC <ad> | If Carry fl= 1, PC ← <ad> | 00010100 | $I2_{PC}$, $I_{SP}$, $R_{PC}$, $WR_{S}$, $RD_{AB}$, $W_{PC}$ |
| 19. | JZ <ad> | If zero fl = 1, PC ← <ad> | 00010101 | $I2_{PC}$, $I_{SP}$, $R_{PC}$, $WR_{S}$, $RD_{AB}$, $W_{PC}$ |
| 20. | JS <ad> | If sign fl = 1, PC ← <ad> | 00010110 | $I2_{PC}$, $I_{SP}$, $R_{PC}$, $WR_{S}$, $RD_{AB}$, $W_{PC}$ |
| 21. | RET | PC ← <stack> | 00011000 | $RD_{S}$, $W_{PC}$, $D_{SP}$ |

| One Register Instructions | | | | |
|---|---|---|---|---|
| **A) Logical Instructions** | | | | |
| 22. | CPR <rn> | if [A] < [<rn>]: carry flag is set.<br>if [A] = [<rn>]: Sign flag is set if [A] > [<rn>]: carry and Sign flags are reset | **01**000<rn> | $R_{RN}$ |
| 23. | AND <rn> | [<rn>] ← [<rn>] & [A] | **01**001<rn> | $R_{RN}$ , $W_{RN}$ |
| 24. | OR <rn> | [<rn>] ← [<rn>] \| [A] | **01**010<rn> | $R_{RN}$ , $W_{RN}$ |
| 25. | XOR <rn> | [<rn>] ← [<rn>] ^ [A] | **01**011<rn> | $R_{RN}$ , $W_{RN}$ |
| 26. | CMR <rn> | [<rn>] ← ~[<rn>] | **01**100<rn> | $R_{RN}$, $W_{RN}$ |
| **B) Arithmetic Instructions** | | | | |
| 27. | ADIR <rn> <od> | [<rn>] ← [<rn>] + <od> | **01**101<rn> | $R_{RN}$ , $W_{RN}$ |
| 28. | SBIR <rn> <od> | [<rn>] ← [<rn>] - <od> | **01**110<rn> | $R_{RN}$ , $W_{RN}$ |
| 29. | ADD <rn> | [<rn>] ← [<rn>] + [A] | **01**111<rn> | $R_{RN}$ , $W_{RN}$ |
| 30. | SUB <rn> | [<rn>] ← [<rn>] - [A] | 10011<rn> | $R_{RN}$ , $W_{RN}$ |
| **C) Data Transfer Instructions** | | | | |
| 31. | MVI <rn> <od> | [<rn>] ← <od> | 10000<rn> | $R_{OR}$ , $W_{RN}$ |
| 32. | MOVD <rn> | [<rn>] ← [A] | 10001<rn> | $R_{A}$, $W_{RN}$ |
| 33. | MOVS <rn> | [A] ←[<rn>] | 10010<rn> | $R_{RN}$ , $W_{A}$ |
| **Two Register Instructions** | | | | |
| 34. | STAR <rnp> | [[<rnp>]] ← [A] | 110<rnp> | $R_{RP}$ , $A_{AR}$ , $R_{A}$, WR |
| 35. | LDAR <rnp> | [A] ← [[<rnp>]] | 111<rnp> | $R_{RP}$ , $A_{AR}$, RD, $W_{A}$ |

**Table 3.3 Instruction Set**

- The register pairs are used for indirect addressing and accessing the general-purpose memory.
- The blocks and the data paths are decided such that the number of clock cycles required for the execution of every instruction are reduced. As a result, no instruction requires more than two clock cycles for either fetch or execution.
- The control signals are decided based on the different paths to different blocks as well as the instructions themselves.

### 3.1.3 Block Diagram

The block diagram consists of an 8-bit Data Bus and a 16-bit Address Bus along with different functional blocks. The Stack Pointer, the Address Register and the Program Counter all need to be 16-bit each to accommodate the 16-bit addresses. The size of the address bus gives us a 64kB on-board memory which has been divided into three parts: General Purpose Memory (30kB), Program Memory (30kB) and Stack Memory (4kB), which consists of 16-bit blocks. Along with the 8 registers, there is an Accumulator which has a direct connection with the ALU to reduce the number of clock cycles required for Accumulator-based arithmetic and logical instructions.

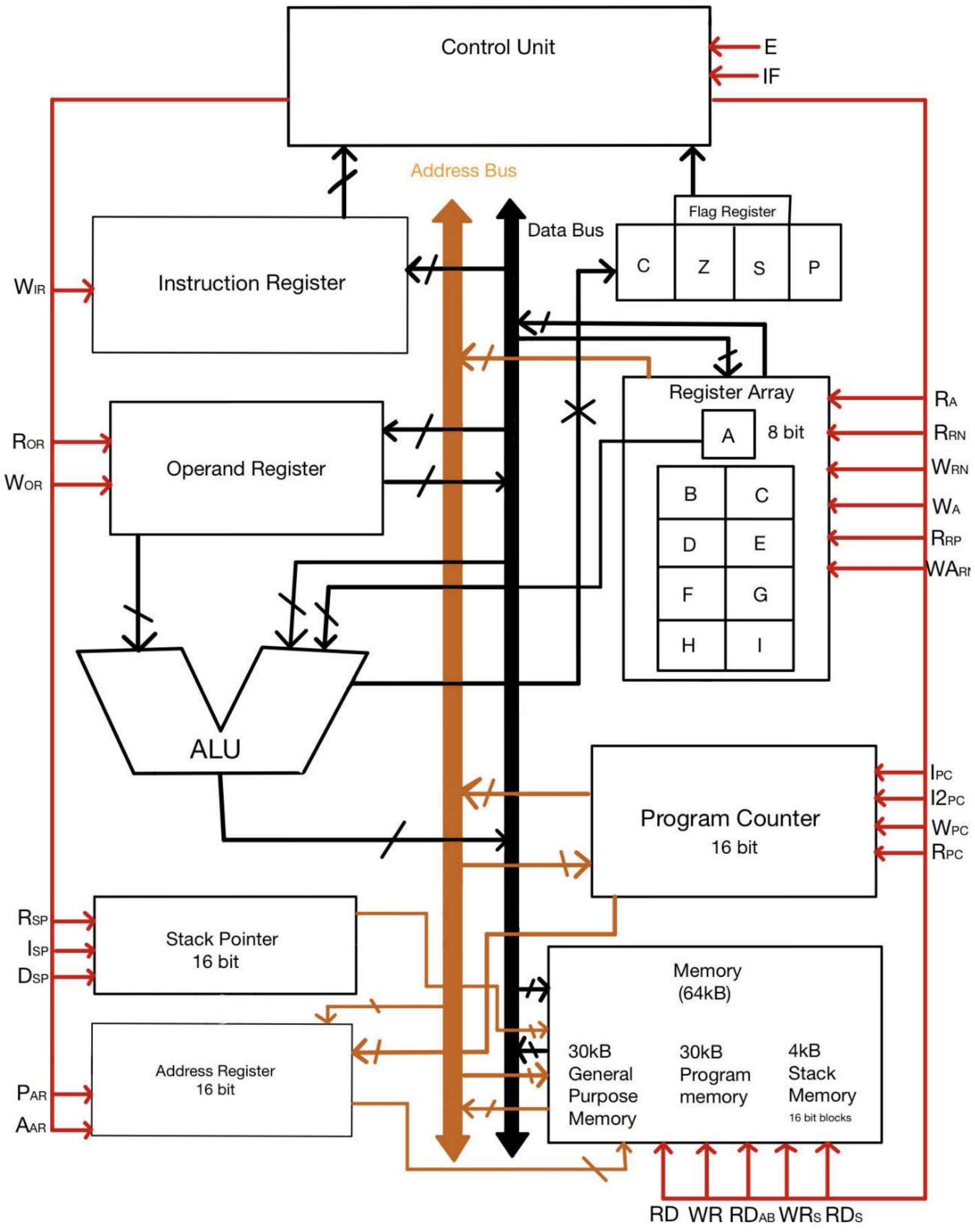


**Fig 3.1 Block Diagram**

- The black lines are Data Buses, and the orange lines are Address Buses.
- The red lines signify the different control signals and their respective blocks.

### 3.1.4 Control Signals

There are different control signals for different functional blocks based on the connections as well as the instructions themselves. Control Signals like $RD_{AB}$, $I2_{PC}$, $R_{RP}$ and $WA_{RN}$ are based on the instructions and not just the connections.

The table below lists all the control signals along with their actions and the blocks they are associated with.

| Control Bits | Action | Module |
|---|---|---|
| RD | Selected memory location is read onto the data bus | Memory |
| WR | Data on Data Bus is written into the memory location | |
| $RD_{AB}$ | Data from two consecutive address locations is concatenated and read onto the address bus (for jumps) | |
| $WR_S$ | Contents of address bus are written into stack memory | |
| $RD_S$ | Contents of stack memory are read onto the address bus | |
| $R_{OR}$ | Contents of Operand Register are read onto the Data Bus | Operand Register |
| $W_{OR}$ | Data on Data Bus is written into the Operand Register | |
| $W_{IR}$ | Data on Data Bus is written into the Instruction Register | Instruction Register |
| $I_{PC}$ | Contents of Program Counter are incremented | Program Counter |
| $I2_{PC}$ | Program counter is incremented by 2 | |
| $W_{PC}$ | Contents of Address Bus are written into the Program Counter | |
| $R_{PC}$ | Contents of Program Counter are read onto the Address Bus | |
| $PC_{AR}$ | Address is read from Program Counter | Address Register |
| $AB_{AR}$ | Address is read from Address Bus | |
| $R_A$ | Data from Accumulator is read onto the Data Bus | Register Array |
| $R_{RN}$ | Data from the specified register is read onto the Data Bus | |

| | | |
|---|---|---|
| $W_{RN}$ | Data from Data Bus is written onto the specified register | |
| $R_{RP}$ | Data from Register Pair is concatenated and read onto the Address Bus | |
| $W_{A}$ | Data from Data Bus is written onto the Accumulator | |
| $WA_{RN}$ | Data from Address Bus is written into all registers | |
| $R_{SP}$ | Contents of Stack Pointer are read onto the Address Bus | Stack Pointer |
| $I_{SP}$ | Stack Pointer is incremented | |
| $D_{SP}$ | Stack Pointer is decremented | |
| E[1] = 1 | Execute Cycle initiated | Control Unit |
| IF[1] = 1 | Instruction Fetch initiated | |
| IF[0] = 1 | Operand Fetch | |
| E[0] = 1 | Write access required | |
| $CL_{FR}$ | Clear Flag Register | Flag Register |

**Table 3.4 Control Signals**

- The control bits required for instruction fetch, operand fetch and the execution cycles of different instructions can be inferred from the above table.
- The control bits required for the execution cycles were added to the instruction set.

## 3.2 RTL CODE USING VERILOG

### 3.2.1. Structure

In order to make the simulation as realistic as possible, we wanted to be able to load the program into memory from the testbench. Along with that, we have also added a hardware reset which can be set from the testbench. To be able to execute this plan, the memory needed to be made separate from the rest of the modules so that it can be directly loaded from the testbench. The following is a diagram illustrating this plan:

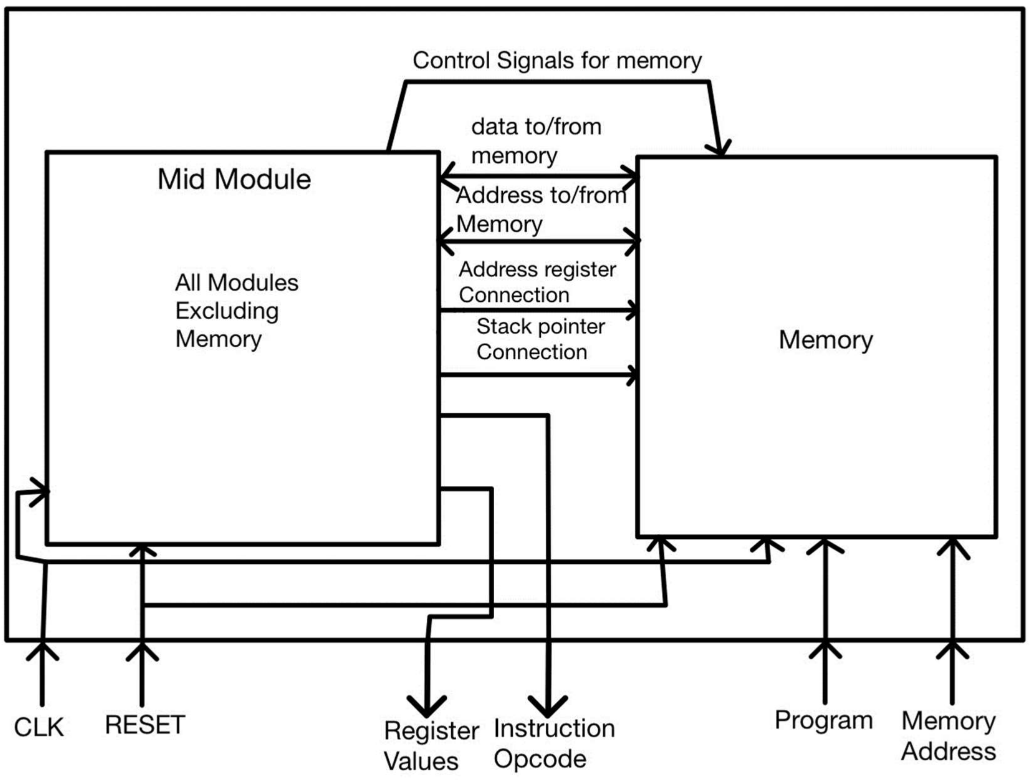


**Fig 3.2 Top Module**

'Mid Module' contains all the modules except for the memory. The control signals from the control unit and the data from the buses need to have access to the memory. We have made this possible by making a 'Top Module' which encompasses both the Mid Module and the memory.

## 3.2.2. Data Flow

Each instruction is executed in 6 clock cycles at most. Determining the specific control bits that need to be set for different stages of every instruction accurately is extremely crucial for it to be executed properly. The six different stages are **Instruction Fetch, Instruction loading, Decision cycle, Operand Fetch, 1st Execution cycle and 2nd Execution cycle.**

**Instruction Fetch:** The program to be executed is stored in the memory with each opcode taking up one memory location followed by an operand, an address location, or the next opcode. The program counter, initially set to 0, gets incremented. This value goes to the address register which in turn points to the first memory location.

**Instruction Loading:** The opcode stored in the memory location gets sent to the instruction register from where it goes to the Control Unit to be decoded.

**Decision Cycle:** If, after decoding, the instruction requires an Operand Fetch, the Control Unit sets the 'fch' control signal, signifying an operand fetch. It also sets the 'ex' control bit if the instruction needs two execution cycles.

**Operand Fetch:** If the instruction requires an Operand Fetch, the program counter is incremented again and the operand kept in the next memory location addressed by the new value in the address register, goes to the Operand Register. Jump instructions utilize this clock cycle for a different purpose.

**Execution Cycles:** Based on the modules used, an instruction can be executed in a maximum of two clock cycles. This helps in preventing contention of the data and address buses. The control signals for the execution of different groups of instructions are unique.

Example:

For any jump instruction, the control signals sent during each stage are as follows:

Instruction Fetch - $I_{PC}$, $P_{AR}$, RD, $W_{IR}$

Operand Fetch - $I2_{PC}$, $I_{SP}$

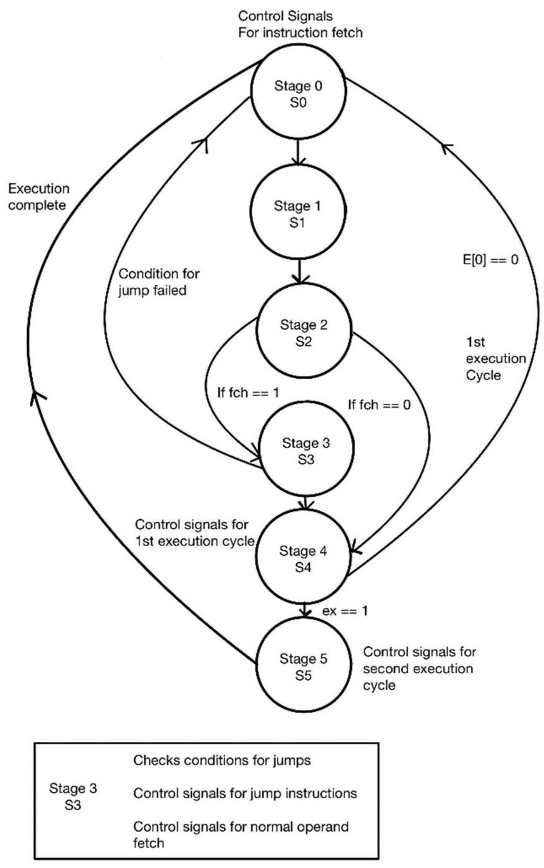
1st Execution Cycle - $R_{PC}$, $WR_S$

2nd Execution Cycle - $RD_{AB}$, $W_{PC}$

After the opcode is fetched and decoded by the Control Unit during the Instruction Fetch, the special control signal $I2_{PC}$ is set which increments the Program Counter by 2 as well as $I_{SP}$ which increments the stack pointer. In the first execution cycle, this value is stored in the stack pointer. After which, $RD_{AB}$ control signal helps read the next two memory locations which are then concatenated to form a 16-bit memory location which is then written onto the Program counter.

## 3.2.3. Control Unit

The Control Unit sends control signals to all the functional blocks. It ensures that the functioning of all blocks is synchronized with the clock. Some instructions require two clock cycles for fetch, and some require two for execution - the Control Unit ensures smooth functioning of each operation and prevents contention when the buses are being used. The addition of two clock cycles made our control unit logic very structured and seamless. The RTL code for the control unit utilizes an FSM model. The following state diagram describes the functioning of the updated FSM model:



Control Signals
For instruction fetch
Stage 0
S0
Stage 1
S1
Stage 2
S2
Stage 3
S3
Stage 4
S4
Stage 5
S5
Execution complete
Condition for jump failed
E[0] == 0
1st execution Cycle
If fch == 1
If fch == 0
Control signals for 1st execution cycle
ex == 1
Control signals for second execution cycle
Stage 3
S3
Checks conditions for jumps
Control signals for jump instructions
Control signals for normal operand fetch

**Fig 3.3 State Diagram**

## 3.3 FPGA Implementation

### 3.3.1 NEXYS A7 Specifications

The FPGA board being used in the implementation is a Digilent Nexys A7 board. This board comes with the following specifications [9]:

- four-digit seven-segment displays, USB HID for mice, keyboards, and memory sticks, 16 switches
- 15,850 logic slices, 4,860Kbits block RAM, 128MiB DDR2 memory
- 6 clock tiles (with PLL), 240 DSP slices
- Powered from USB or any 7V to 15V source
- microSD card connector

### 3.3.2 VGA Implementation

The output of the implementation was displayed on a VGA monitor. For this, an ASCII ROM module was designed using the on-board BRAM. The ASCII ROM module was designed such that, when called, it will enable specific pixels on the screen thereby creating the specific character. This ASCII ROM along with the VGA monitor scanning is controlled by the VGA Controller module. This module runs a constant scan of the pixels on the screen at a frequency of 100MHz and calls ASCII characters needed to display the outputs of the processor. The code is used for the VGA controller and the ASCII ROM has been taken with due permission.

### 3.3.3 Input Mapping

For the inputs, switches and buttons present on the board are used. The 16 switches that are available are used to input Address and Data. All 16 are used to input the Address values and 8 of these are used to input Data values for these addresses. The differentiation between Address and Data input is done using buttons; 4 buttons are used for the functioning of the processor. Two of these buttons are used to input Data and Address values, one is used to emulate a clock cycle and the one is used as a System reset and the last one is used as a VGA reset. The components used are shown below:

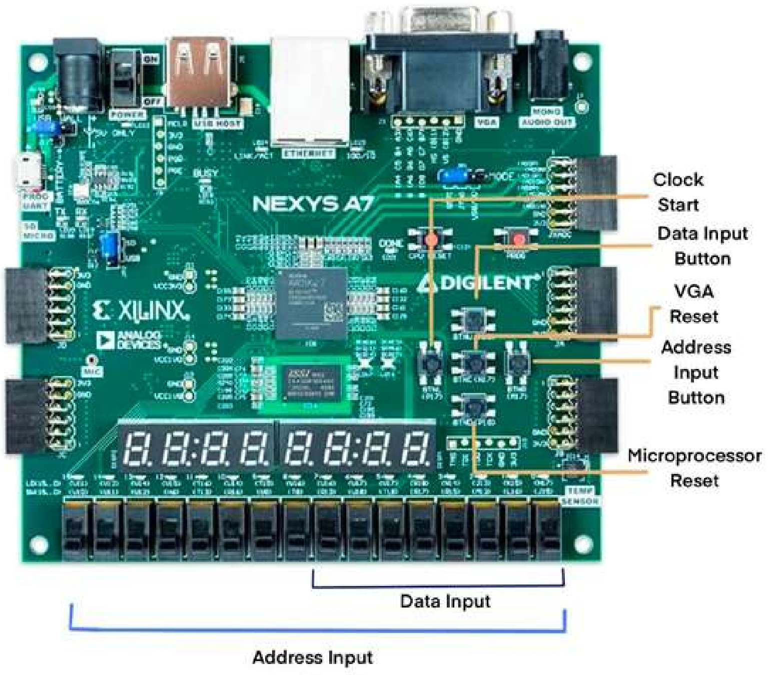


**Fig 3.4 NEXYS A7 Board**

## 3.4 RESULTS

### 3.4.1 MULTIPLICATION

Our next step in this project was to make sure that all groups of instructions can be executed without any issues. We simulated a few small programs to test different categories of instructions in our instruction set. The algorithm for our multiplication program is as follows:

| ALGORITHM | ASSEMBLY CODE |
|---|---|
| Rb <- First Number | MVI Rb 0x3 |
| A <- Second Number | LDI 0x3 |
| Rc <- Rc + A | ADD Rc |
| Rb <- Rb -1 | SBIR Rb 0x1 |
| Jump when not zero | JNZ 0x04 |

The assembly code was written as 20 lines of machine language code in our testbench and simulated. The following are the results:

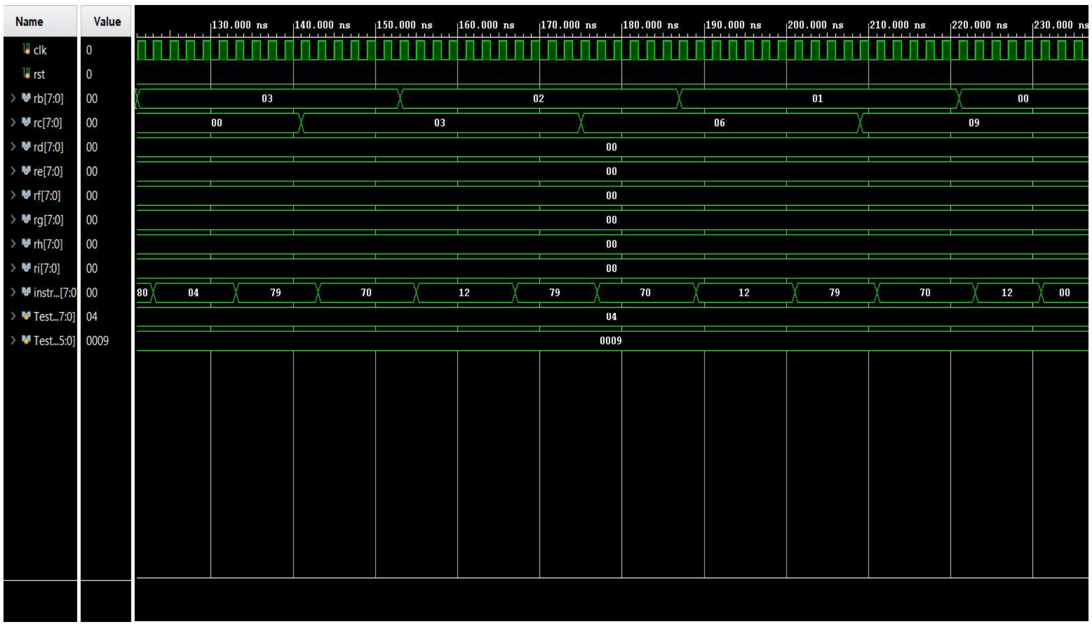


**Fig 3.5 Simulation for Multiplication**

### 3.4.2 DIVISION

While writing the algorithm for division, we realized the need for a subtract operation and decided to add it to the instruction set. The following is the algorithm for a simple division program. It is to be noted that the program has certain limitations such as loading the smaller number into the accumulator and the numbers having to be perfectly divisible due to the lack of a remainder. These limitations can be easily overcome by a more complex algorithm which we plan on executing during the course of this project.

| ALGORITHM | ASSEMBLY CODE |
|---|---|
| Rc <- First Number | MVI Rc 0x9 |
| A <- Second Number | LDI 0x3 |
| Rb <- Rb + 1 | ADIR Rb 0x1 |
| Rc <- Rc - A | SUB Rc |
| Jump when not zero | JNZ 0x04 |

The following are the results:

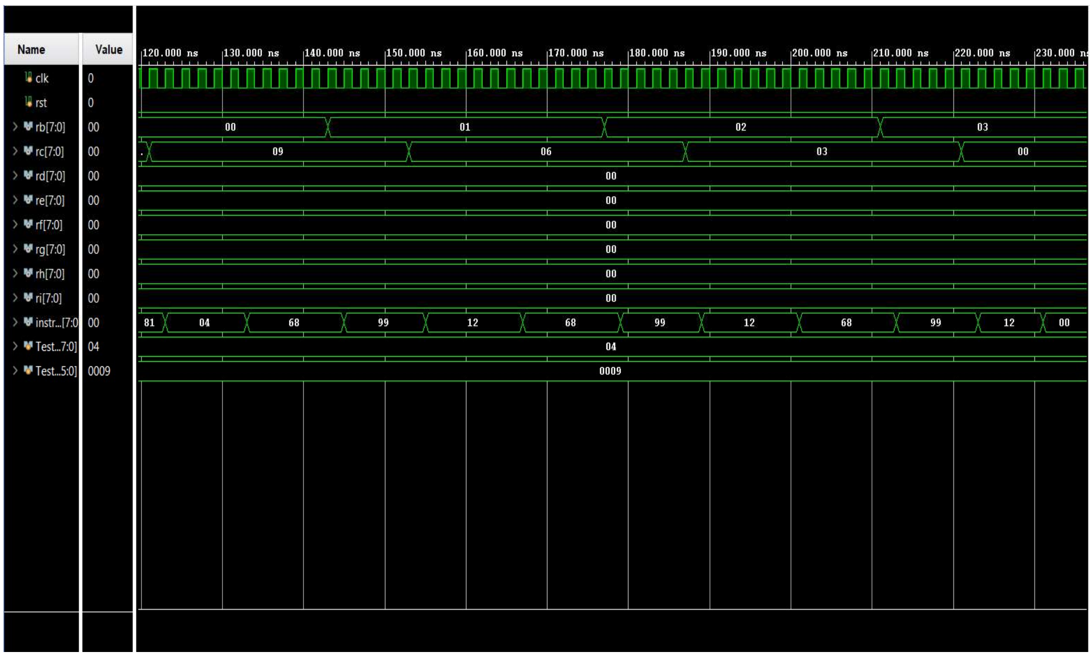
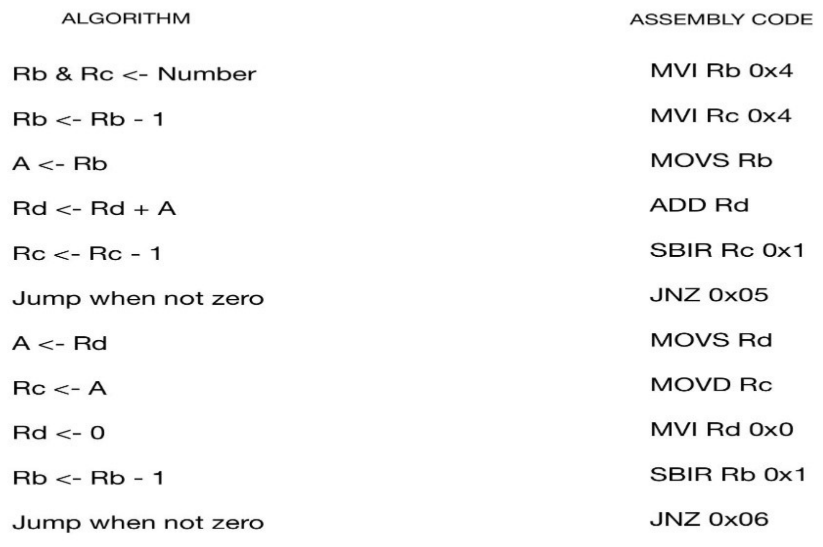


**Fig 3.6 Division Simulation**

### 3.4.3 FACTORIAL

The last algorithm we used to test the processor was a Factorial algorithm. The algorithm is as follows:



| ALGORITHM | ASSEMBLY CODE |
|---|---|
| Rb & Rc <- Number | MVI Rb 0x4 |
| Rb <- Rb - 1 | MVI Rc 0x4 |
| A <- Rb | MOVS Rb |
| Rd <- Rd + A | ADD Rd |
| Rc <- Rc - 1 | SBIR Rc 0x1 |
| Jump when not zero | JNZ 0x05 |
| A <- Rd | MOVS Rd |
| Rc <- A | MOVD Rc |
| Rd <- 0 | MVI Rd 0x0 |
| Rb <- Rb - 1 | SBIR Rb 0x1 |
| Jump when not zero | JNZ 0x06 |

The following are the results:

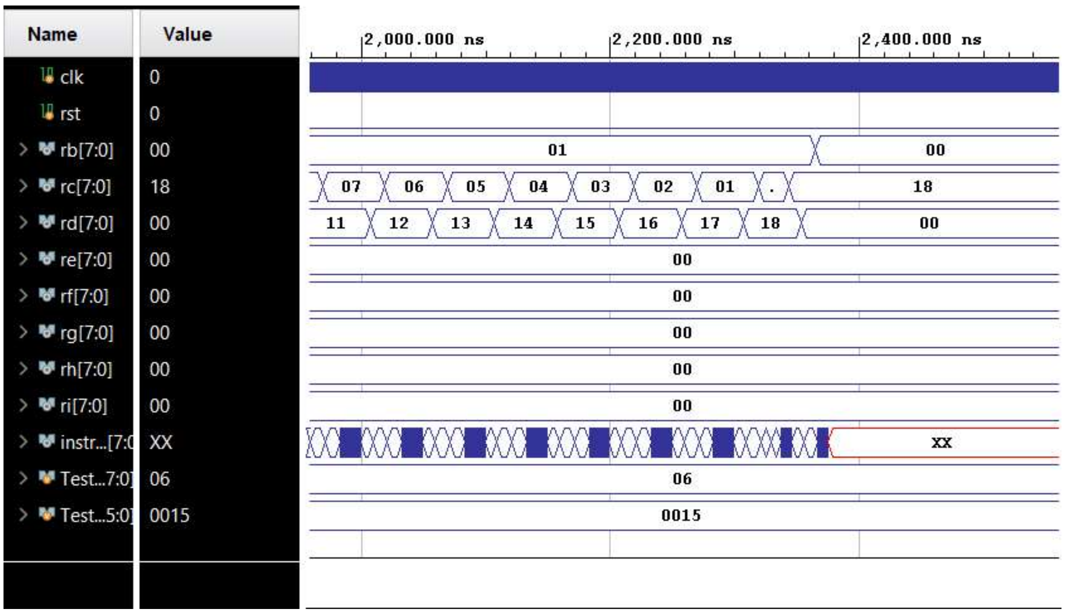


Fig 3.7 Factorial Simulation

### 3.4.4 VGA Monitor Display

The multiplication algorithm was used as a test for the display. The results for the display are shown below:

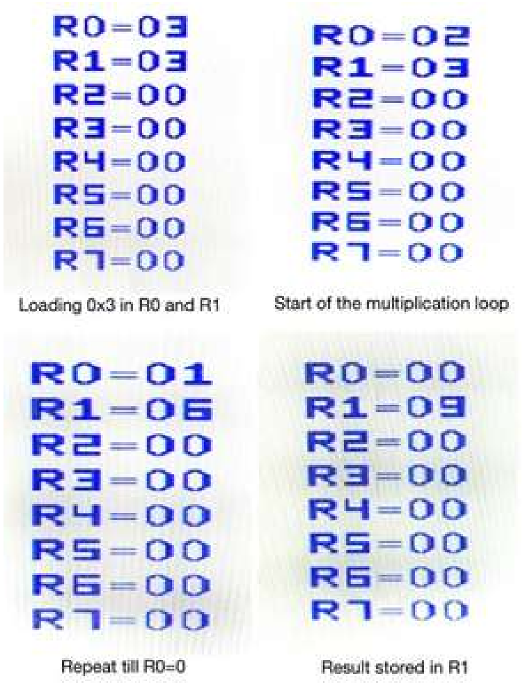
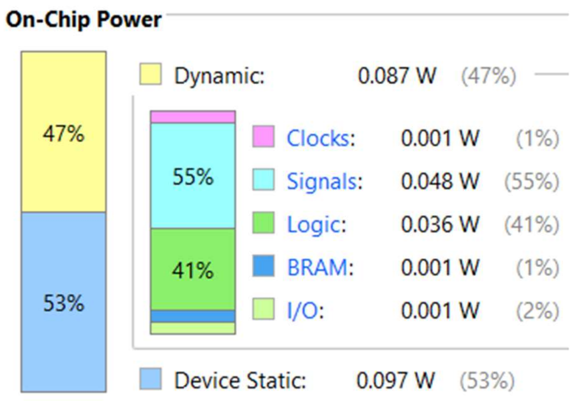


Fig 3.8 VGA Display

## 3.4.5 Timing and Power

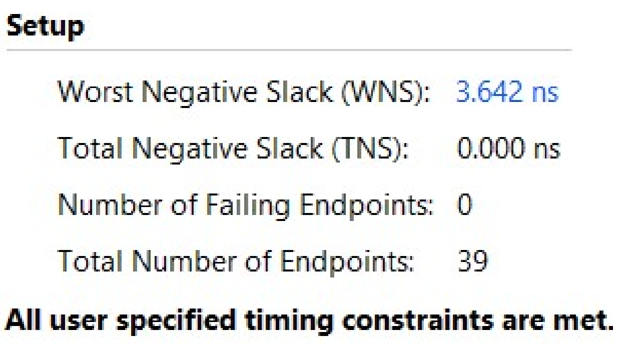
After burning the RTL code onto the FPGA board, the timing and power consumption reports were obtained. A margin of 3.642ns was kept on setup time to account for process variations. The maximum clock frequency for the microprocessor to function correctly was found to be 100MHz, which can be increased further by reducing the margin. The total on-chip power was found to be 0.184W which meets the power criteria for the Nexys A7 FPGA board. The analysis reports are given in Fig. 3.9 to 3.12.



On-Chip Power

| | | | |
|---|---|---|---|
| 🟨 | Dynamic: | 0.087 W | (47%) |
| 🟪 | Clocks: | 0.001 W | (1%) |
| 🟦 | Signals: | 0.048 W | (55%) |
| 🟩 | Logic: | 0.036 W | (41%) |
| 🟦 | BRAM: | 0.001 W | (1%) |
| 🟩 | I/O: | 0.001 W | (2%) |
| 🟦 | Device Static: | 0.097 W | (53%) |

Fig 3.9 Power Consumption Report

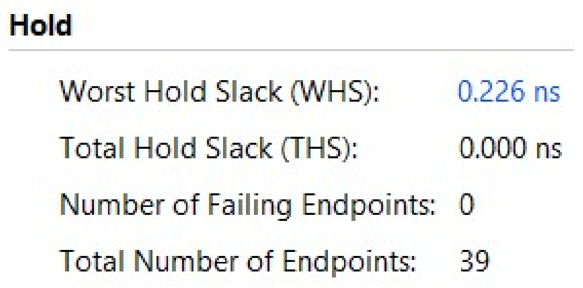


**Setup**

| | |
|---|---|
| Worst Negative Slack (WNS): | 3.642 ns |
| Total Negative Slack (TNS): | 0.000 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 39 |

**All user specified timing constraints are met.**

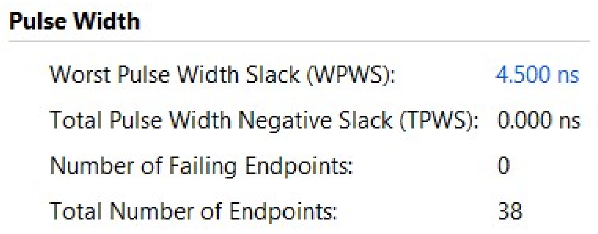
Fig 3.10 Setup Timing



**Hold**

| | |
|---|---|
| Worst Hold Slack (WHS): | 0.226 ns |
| Total Hold Slack (THS): | 0.000 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 39 |

**Fig 3.11 Hold Timing**



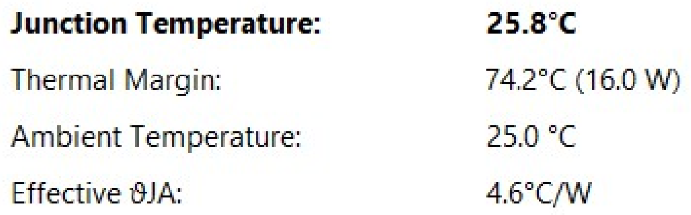
**Pulse Width**

| | |
|---|---|
| Worst Pulse Width Slack (WPWS): | 4.500 ns |
| Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 38 |

**Fig. 3.12 Pulse Width Timing**

### 3.4.6 Temperature

The junction temperature was found to be 25.8°C with a thermal margin of 74.2°C. Fig. 3.13 shows this result.



| | |
|---|---|
| **Junction Temperature:** | **25.8°C** |
| Thermal Margin: | 74.2°C (16.0 W) |
| Ambient Temperature: | 25.0 °C |
| Effective ϑJA: | 4.6°C/W |

**Fig 3.13 Temperature Report**

# Chapter 4
# Future Work

- Our project has a lot of scope for enhancements. One prospect would be to enhance efficiency through the implementation of a pipeline structure. A pipeline architecture facilitates parallel processing of instructions, thereby optimizing the overall performance of the microprocessor. We have already divided the process into six stages. This lays the groundwork for the pipeline. In the months, we plan on allowing these stages to work concurrently allowing the processor to initiate the next instruction without waiting for the current one to complete. This would be a very crucial step towards creating a high-performance microprocessor.

- Lastly, a compiler can be designed to work with this microprocessor. It would significantly improve the accessibility of this system.

- This project is meant to help the student population understand the working and design of a microprocessor. Therefore, integration of this project in coursework would be a natural next step.

# BIBLIOGRAPHY

[1] A. Sharma, D. Seal, and V. Hariharan, "CPU 101: Design of a Simple Microprocessor from First Principles", 2024 IEEE India Council International Subsections Conference (INDISCON), Chandigarh, India [Submitted]

[2] A. Sharma, D. Seal, and V. Hariharan, "CPU 101: Design and FPGA Implementation of a Simple Microprocessor", 2024 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)

[3] Kant, Krishna. Microprocessors and Microcontrollers: Architecture, Programming and System Design 8085, 8086, 8051, 8096. PHI Learning Pvt. Ltd., 2007.

[4] Hennessy, John L., and David A. Patterson. Computer architecture: a quantitative approach. Elsevier, 2011.

[5] Smruti R. Sarangi. Basic Computer Architecture. White Falcon Publishing, 2021.

[6] Flake, Peter, Phil Moorby, Steve Golson, Arturo Salz, and Simon Davidmann. 2020. "Verilog HDL and Its Ancestors and Descendants." Proceedings of the ACM on Programming Languages 4 (HOPL). Association for Computing Machinery. doi:10.1145/3386337.

[7] Available: https://github.com/digital-design-snu/RNBIP

[8] Available: https://gnusim8085.srid.ca/

[9] Available: https://www.rfwireless-world.com/Tutorials/8085-instruction-set.html

[10] Available : https://github.com/CasperWhite13/AD8Microprocessor

# APPENDIX

## CODES

### CONTROL UNIT

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
/*
* Copyright (c) 2023, Shiv Nadar University, Delhi NCR, India. All Rights
* Reserved. Permission to use, copy, modify and distribute this software for
* educational, research, and not-for-profit purposes, without fee and without a
* signed license agreement, is hereby granted, provided that this paragraph and
* the following two paragraphs appear in all copies, modifications, and
* distributions.
*
* IN NO EVENT SHALL SHIV NADAR UNIVERSITY BE LIABLE TO ANY PARTY FOR DIRECT,
* INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST
* PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE.
*
* SHIV NADAR UNIVERSITY SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT
* NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
* PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS PROVIDED "AS IS". SHIV
* NADAR UNIVERSITY HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,
* ENHANCEMENTS, OR MODIFICATIONS.
* Revision History:
*           Date                        By                                Change Notes
* ---------------------- --------------------- ------------------------------------------
*  8th October 2023        Aditi Sharma            #control bits changed
*
*  13th October 2023       Aditi Sharma            Changed logic for 'stage'
*
*  3rd November 2023       Aditi Sharma            Attempt to move the complexity of
*                                                  register array to control unit
*
*  7th November 2023       Aditi Sharma            Control signals added for remaining
*                          Devyam Seal             instructions
*
*  8th November 2023       Aditi Sharma            'Case' statements changed to 'casex'
*
*  20th February 2024      Aditi Sharma            Operand fetch and second execution
*                                                  cycle decisions were made asynchronous
*/
//////////////////////////////////////////////////////////////////////////////////


module ControlUnit(rd, wr, rd_ab, wr_s, rd_s,r_or, w_or,
                   w_ir,i_pc, i2_pc, w_pc, r_pc,p_ar, a_ar,r_a, r_rn, w_rn,
                   r_rp, w_a, wa_rn, r_sp, i_sp, d_sp, cl_fr, E, IF, clk, opcode, rst, flags,
stage);


function [23:0] setBits; // Creating a function to set control bits
    input integer a, b, c, d;
        begin
            setBits = 24'b0;

            setBits[a] = 1'b1;
            setBits[b] = 1'b1;
            setBits[c] = 1'b1;
            setBits[d] = 1'b1;
        end
```

```
endfunction


input clk;
input [3:0] flags;
input [7:0] opcode;
output rd, wr, rd_ab, wr_s, rd_s; // memory
output r_or, w_or; // operand register
output w_ir; // instruction register
output i_pc, i2_pc, w_pc, r_pc; // program counter
output p_ar, a_ar; //address register
output r_a, w_a, wa_rn;//register array
output r_rn, w_rn;
output r_rp;
output r_sp, i_sp, d_sp;//stack pointer
output reg [1:0] E, IF;//control unit
output cl_fr;
input rst;
wire fch, ex;

output reg [2:0] stage;
reg [23:0] cb;
//reg [3:0] flagscu;

parameter [2:0] s0=3'b000; // IF state
parameter [2:0] s05=3'b001; // Operand Fetch
parameter [2:0] s052=3'b010; // State delay
parameter [2:0] s1=3'b011; // State delay
parameter [2:0] s2=3'b100; // Execute State 1
parameter [2:0] s3=3'b101; // Execute State 2



  assign {rd, wr, rd_ab, wr_s, rd_s,r_or, w_or,w_ir,i_pc, i2_pc, w_pc,
     r_pc,p_ar, a_ar,r_a, r_rn, w_rn, r_rp, w_a, wa_rn, r_sp, i_sp,
     d_sp, cl_fr} = cb; // Assigning values to signals based on 'cb'




initial
begin
    stage = 2'b00;
    cb <= 24'b000000000000000000000000;
end

// value of fch changes based on whether the instruction requires operand fetch
assign fch = ( (opcode == 8'b00000100) || (opcode == 8'b00000110) || (opcode == 8'b00000111) ||
               (opcode == 8'b00001000) || (opcode == 8'b00001001) || (opcode == 8'b00001010) ||
               (opcode == 8'b00001011) || (opcode == 8'b00001101) || (opcode == 8'b00001110) ||
               (opcode[7:3] == 5'b01101) || (opcode[7:3] == 5'b01110) || (opcode == 8'b00010000)
||
               (opcode == 8'b00010001) ||  (opcode ==8'b00010010) ||  (opcode == 8'b00010011) ||
               (opcode == 8'b00010100) || (opcode == 8'b00010101) ||  (opcode == 8'b00010110) ||
               (opcode[7:3] == 5'b10000))? (1'b1) : 1'b0 ;

// value of ex changes based on whether the instruction requires a second execution cycle
assign ex = ( (opcode[7:3] == 5'b00010) || (opcode == 8'b00011000) || (opcode[7:3] == 5'b01001)
||
              (opcode[7:3] == 5'b01010) || (opcode[7:3] == 5'b01011) || (opcode[7:5] == 3'b011)
||
              (opcode[7:6] == 2'b11) || (opcode[7:3] == 5'b10011))? 1'b1 : 1'b0 ;



// can be put in the other 'always' block
always @(posedge clk)
  begin
    if(rst == 1'b1)
    begin
```

```
        cb <= 0;
        E <= 0;
        IF <= 0;
    end
  end


always @(posedge clk)
begin
    $monitor("%t, stage = %b" , $time, stage);
    $monitor("%t, opcode = %b" , $time, opcode);
    $monitor("%t, IF[0] = %b" , $time, fch);
    $monitor("%t, cb = %b " ,$time, cb);
    $monitor("%t, icontrol_pc = %b " ,$time, i_pc);

    // main logic of the control unit
     case (stage)
     s0:

        begin


          IF[1] <= 1'b1;
          E[1] <= 1'b0;
           // cb = 24'b100000011000100000000000;
           cb <= setBits(11,15,16,23);

           stage <= s05;
        end

     s05: // extra clock cycle for fch & ex to change with IR
         begin
            stage <= s052;
            cb <= 0;
         end

     s052: // clock cycle for decisions
          begin
                 if(fch == 1)
                     stage <= s1;
                 else
                     stage <= s2;
          end


        s1:
        begin
           if (opcode[7:3] == 5'b00010) //checking if conditions for jump are satisfied. If no,
I2PC and move on to next instruction
           begin
                if (((opcode[2:0] == 3'b001) && (flags[3] == 1'b1)) || ((opcode[2:0] == 3'b010)
&& (flags[2] == 1'b1)) || ((opcode[2:0] == 3'b011) && (flags[1] == 1'b1)) ||
                    ((opcode[2:0] == 3'b100) && (flags[3] == 1'b0)) || ((opcode[2:0] == 3'b101)
&& (flags[2] == 1'b0)) || ((opcode[2:0] == 3'b110) && (flags[3] == 1'b0)) )
                begin
                    //cb = 24'b000000000100100000000000;
                    cb <= setBits(11,14,14,14);
                    stage <= 2'b00;
                end
                else
                begin
                    //cb = 24'b000000000100100000000100;
                      cb <= setBits(2,12,14,14); // if condition satisfied, ISP with I2PC
                      stage <= s2;
                end
          end

          else
          begin
             //cb = 24'b100000101000100000000000; // Normal operand fetch,
             cb <= setBits(11,15,17,23);
             stage <= s2;
          end
        end
     s2:
        begin

          E[1] <= 1'b1;
          IF[1] <= 1'b0;

            casex(opcode)
               // 8'b11xxxxxx : cb <= {15'b0, r_rn, w_rn, r_rp, 6'b0}  ;
                8'b11XXXXXX : cb <= setBits(6,10,10,10);
                //cb <= 24'b000000000000010001000000; //both two register functions
                8'b10010XXX : cb <= setBits(5,8,8,8); //**Shift 5 to next stage
                //cb <= 24'b000000000000000100100000;
                8'b10001XXX : cb <= setBits(7,9,9,9);
                //cb <= 24'b000000000000001010000000;
                8'b10000XXX : cb <= setBits(7,18,18,18);
                //cb <= 24'b000001000000000010000000;
                8'b01000XXX : cb <= setBits(8,8,8,8);
                //cb <= 24'b000000000000000100000000;
                8'b01001XXX : cb <= setBits(8,8,8,8);
                //cb <= 24'b000000000000000100000000;
                8'b01010XXX : cb <= setBits(8,8,8,8);
                //cb <= 24'b000000000000000100000000;
                8'b01011XXX : cb <= setBits(8,8,8,8);
                //cb <= 24'b000000000000000100000000;
                8'b01100XXX : cb <= setBits(8,8,8,8);
                //cb <= 24'b000000000000000100000000;
                8'b01101XXX : cb <= setBits(8,8,8,8);
                //cb <= 24'b000000000000000100000000;
                8'b01110XXX : cb <= setBits(8,8,8,8);
                //cb <= 24'b000000000000000100000000;
                8'b01111XXX : cb <= setBits(8,8,8,8);
                //cb <= 24'b000000000000000100000000;
                8'b00010XXX : cb <= setBits(20,12,12,12);
                //cb <= 24'b000100000001000000000000; // all jump instructions no return
                8'b00011000 : cb <= setBits(13,19,19,19);
                //cb <= 24'b000010000001000000000000;
                8'b00001XXX : cb <= setBits(5,5,5,5);
                //cb <= 24'b000000000000000000100000;
                8'b00000111 : cb <= setBits(5,5,5,5);
                //cb <= 24'b000000000000000000100000;
                8'b00000110 : cb <= setBits(5,5,5,5);
                //cb <= 24'b000000000000000000100000;
                8'b00000101 : cb <= setBits(9,17,17,17);
                //cb <= 24'b000000100000001000000000;
                8'b00000100 : cb <= setBits(5,18,18,18);
                //cb <= 24'b000001000000000000100000;
                8'b00000011 : cb <= setBits(0,0,0,0);
                //cb <= 24'b000000000000000000000001;
                8'b00000000 : cb <= setBits(4,4,4,4);
                //cb <= 24'b000000000000000000010000;
                8'b10011XXX : cb <= setBits(8,8,8,8);
           endcase

           if(ex == 1'b1)
           begin
               stage <= s3;
           end

           else
           begin
               stage <= 2'b00;
           end

        end

     s3:
        begin
```

```
            casex (opcode)

                8'b00010XXX : cb <= setBits(13,21,21,21);
                //cb <= 24'b001000000001000000000000000;
                8'b00011000 : cb <= setBits(1,1,1,1);
                //cb <= 24'b000000000000000000000010;
                8'b01001XXX : cb <= setBits(7,7,7,7);
                //cb <= 24'b000000000000000010000000;
                8'b01010XXX : cb <= setBits(7,7,7,7);
                //cb <= 24'b000000000000000010000000;
                8'b01011XXX : cb <= setBits(7,7,7,7);
                //cb <= 24'b000000000000000010000000;
                8'b011XXXXX : cb <= setBits(7,7,7,7);
                //cb <= 24'b000000000000000010000000;
                8'b110XXXXX : cb <= setBits(22,9,9,9);
                //cb <= 24'b010000000000001000000000;
                8'b111XXXXX : cb <= setBits(23,5,5,5);
                //cb <= 24'b100000000000000000100000;
                8'b10011XXX : cb <= setBits(7,7,7,7);

            endcase

            stage <= 2'b00;


        end

      endcase

   end

endmodule
```

## ALU

```
module ALU(
    input clk,
    //input rst,
    output [7:0] Out,
    output [3:0] Flag,
    input [7:0] RN,
    input [7:0] OD,
    input [7:0] AC,
    input [7:0] opcode,
    input [2:0] stage
);

reg [7:0] A, B;
reg [3:0] inst;



parameter [3:0] CP = 4'b0000;
parameter [3:0] AND = 4'b0001;
parameter [3:0] OR = 4'b0010;
parameter [3:0] XOR = 4'b0011;
parameter [3:0] CM = 4'b0100;
parameter [3:0] ADD = 4'b0101;
parameter [3:0] SUB = 4'b0110;
parameter [3:0] ADDR = 4'b0111;
parameter [3:0] SUBR = 4'b1001;

reg Carry, Zero, Parity, Sign;
wire carry2;
wire [7:0] outFF;

initial
begin
```

```
    Carry <= 1'b0;
    Zero <= 1'b0;
    Parity <= 1'b0;
    Sign <= 1'b0;
    //Out <= 0;
end

assign outFF = (stage == 3'b000)? Out : outFF;

always @(posedge clk)
begin

    $monitor("%t, ALU RN in = is %b" , $time, RN);
    $monitor("%t, ALU OD in = is %b" , $time, OD);
    $monitor("%t, ALU out = is %b" , $time, Out);
    $monitor("%t, carry = is %b" , $time, carry2);
    $monitor("%t, inst = is %b" , $time, inst);
    $monitor("%t, A = is %b" , $time, A);
    $monitor("%t, B = is %b" , $time, B);
    $monitor("%t, ALU outFF = is %b" , $time, outFF);


    Sign <= 1'b0;

    if(opcode[7:3] == 5'b00001)
        begin
            inst <= opcode[2:0];
        end

    else if(opcode[7:6] == 2'b01)
        begin
            inst <= opcode[5:3];
        end
    else if(opcode[7:3] == 5'b10011)
        begin
            inst <= opcode[7:4];
        end

    else
        begin
            inst <= 3'bzzz;
        end


     if (opcode[7:3] == 5'b00001)
        begin
            A <= AC;
            B <= OD;
        end

     else if (opcode[7:5] == 3'b010)
        begin
            A <= RN;
            B <= AC;
        end

     else if ((opcode[7:3] == 5'b01101) || (opcode[7:3] == 5'b01110) )
        begin
            A <= RN;
            B <= OD;
        end

     else if ((opcode[7:3] == 5'b01111) || (opcode[7:3] == 5'b10011))
        begin
            A <= RN;
            B <= AC;
        end

     else
        begin
            A <= 8'b0;
            B <= 8'b0;
        end


     case (inst)

        CP:
            begin
                if(B<A)
                    begin
                        Carry <= 1'b1;
                        Sign <= 1'b0;
                    end
                else if(A==B)
                    begin
                        Sign <= 1'b1;
                        Carry <= 1'b0;
                    end
                else
                    begin
                        Sign <= 1'b0;
                        Carry <= 1'b0;
                    end
            end
    endcase

    Parity <= ^outFF;
    Zero <= ~(|outFF);

end

 assign {carry2,Out}=  (inst == CM) ? {1'b0,~A} : (
                       (inst == AND)? {1'b0,A & B} : (
                       (inst == OR) ? {1'b0,A | B} : (
                       (inst == XOR)? {1'b0,A ^ B} : (
                       ((inst == ADD) || (inst == ADDR))?  (A + B) : (
                       ((inst == SUB) || (inst == SUBR))?  (A - B) : (
                       (opcode[7:3] == 5'b01111)? {1'b0,RN + AC} : (
                       (opcode == 8'b00000110)? {1'b0,AC << OD} : (
                       (opcode == 8'b00000111)? {1'b0,AC >> OD} : {1'b0,Out} ))))))));


assign Flag = {(Carry||carry2), Zero, Sign, Parity};

endmodule
```

## OPERAND REGISTER

```
module OperandRegister( input clk,
                        input rst,
                        input r_or, w_or,
                        input [7:0] data,
                        output [7:0] out_or,
                        output [7:0] alu_out);

reg [7:0] opr;
wire [7:0] opreg;

assign opreg = (w_or)? data : opr;

assign alu_out = opreg;

assign out_or = opreg;

initial
begin
    opr = 8'bz;
end

always @(posedge clk)
```

```
begin
    $monitor("%t, OR = %b" , $time, opreg);


    if(rst == 1'b1)
        begin
            opr = 0;
        end


    opr <= opreg;

end


endmodule
```

## ADDRESS REGISTER

```
module AddressRegister( input clk,
                        input rst,
                        input [15:0] PC_in,
                        input [15:0] AB_in,
                        input P_ar, A_ar,
                        output [15:0] AdReg_out);

reg [15:0] ADR;
initial
begin
ADR <= 16'b0;
end


assign  AdReg_out = (P_ar)? PC_in :(
                 (A_ar)? AB_in  : ADR );

always @(posedge clk)
begin

    if(rst == 1'b1)
    begin
        ADR <= 0;
    end

    $monitor("%t, AR = %b" , $time, AdReg_out);
    ADR <= AdReg_out;
```

## FLAG RESGISTER

```
module FlagRegister(input [3:0] ALU_in,
                    input rst,
                    input CL_f,
                    output [3:0] cu_out);

reg [3:0] FlagArray;

initial                         // Initialising
begin
    FlagArray[0] <= 1'b0;
    FlagArray[1] <= 1'b0;
    FlagArray[2] <= 1'b0;
    FlagArray[3] <= 1'b0;
end
```

```
assign cu_out = FlagArray;

always @(*)
begin

    if(rst == 1'b1)
    begin
        FlagArray <= 0;
    end

    if  (CL_f == 1'b1)  // Control signal to clear flags
        begin
            FlagArray[0] <= 1'b0;
            FlagArray[1] <= 1'b0;
            FlagArray[2] <= 1'b0;
            FlagArray[3] <= 1'b0;
        end

    FlagArray[0] <= ALU_in[0];  // Getting flag data from ALU
    FlagArray[1] <= ALU_in[1];
    FlagArray[2] <= ALU_in[2];
    FlagArray[3] <= ALU_in[3];

end
endmodule
```

## INSTRUCTION REGISTER

```
module InstructionRegister(
    input rst,
    input clk,
    input w_ir,
    input [7:0] data,
    output [7:0] out_ir);

reg [7:0] ir;
//reg skip;

assign out_ir = ir;


initial
begin
    ir <= 8'bz;
end

always @(posedge clk)
begin

    if(rst == 1'b1)
    begin
        ir <= 8'b0;
    end

end


always @(posedge clk)
begin

    if(w_ir == 1'b1)
    begin
        ir <= data;
    end

    $monitor("%t, IR = %b", $time, ir);
end

endmodule
```

## PROGRAM COUNTER

```
module ProgramCounter(input rst,
                      input clk,
                      input i_pc, i2_pc, w_pc, r_pc,
                      input [15:0] address,
                      output [15:0] out,
                      output [15:0] ar_out);

reg [15:0] pc;

initial
begin
    pc <= 16'b0;
end


assign ar_out = pc;
assign out = pc;

always @(posedge clk)
begin

    $monitor("%t, PC = %b", $time, pc);
    $monitor("%t, IncPC = %b", $time, i_pc);

    if(rst == 1'b1)
    begin
        pc <= 0;
    end

    if(i_pc == 1'b1)
    begin
        pc <= pc + 1'b1;
    end

    if(i2_pc == 1'b1)
    begin
        pc <= pc + 2;
    end

    if(w_pc == 1'b1)
    begin
        pc <= address;
    end

end

endmodule
```

## STACK POINTER

```
module StackPointer(input clk,
                    input rst,
                    input I_sp,
                    input D_sp,
                    output [15:0] Mem_out);

reg SP = 0; // Initialising

assign Mem_out = SP; // output is asynchronous

always @(posedge clk)
begin

    if(rst == 1'b1)
    begin
        SP <= 0;
    end

    if (I_sp == 1'b1) // Increment control signal
    begin
        SP <= SP+1;
    end

    else if(D_sp == 1'b1) // Decrement control signal
    begin
        SP <= SP-1;
    end

 //   Mem_out = SP;  // Sending Stack Pointer value to memory
end
endmodule
```

## REGISTER ARRAY

```
module RegisterArray(   input rst,
                        input [7:0] data_in,
                        input r_a,r_rn,w_rn,wa_rn,w_a,r_rp,
                        input [7:0] opcode,
                        input clk,
                        //output reg [7:0] data_out, ac_out,
                        output [7:0] data_out,
                        output [15:0] addr_out,
                        output [7:0] ac_ALU,
                        output [7:0] rb,
                        output [7:0] rc,
                        output [7:0] rd,
                        output [7:0] re,
                        output [7:0] rf,
                        output [7:0] rg,
                        output [7:0] rh,
                        output [7:0] ri


                        );

reg [7:0] register [7:0];
reg [7:0] ac;
reg [7:0] rn_out;
//reg [7:0] a_out;
reg [15:0] pair;
reg [2:0] reg_sel;
reg [4:0] pair_sel;



initial
    begin
        ac <= 0;
        reg_sel <= 3'bz;
        pair_sel <= 5'bz;
        register[0]<= 0;
        register[1]<= 0;
        register[2]<= 0;
        register[3]<= 0;
        register[4]<= 0;
        register[5]<= 0;
        register[6]<= 0;
        register[7]<= 0;
    end

assign ac_ALU = ac;
assign addr_out = pair;
assign data_out = r_a? ac : register[reg_sel];
```

```
always @(posedge clk)
begin

    if(rst == 1'b1)
    begin

        ac <= 0;
        register[0]<= 0;
        register[1]<= 0;
        register[2]<= 0;
        register[3]<= 0;
        register[4]<= 0;
        register[5]<= 0;
        register[6]<= 0;
        register[7]<= 0;

    end

    if (opcode[7:6] == 2'b11)
    begin
        pair_sel <= opcode[4:0];
    end

    else if(opcode[7:6] == 2'b10 || opcode[7:6] == 2'b01)
    begin
        reg_sel <= opcode[2:0];
    end

    case(pair_sel)
        00000: pair <= {register[0], register[1]};
        00001: pair <= {register[0], register[2]};
        00010: pair <= {register[0], register[3]};
        00011: pair <= {register[0], register[4]};
        00100: pair <= {register[0], register[5]};
        00101: pair <= {register[0], register[6]};
        00110: pair <= {register[0], register[7]};
        00111: pair <= {register[1], register[2]};
        01000: pair <= {register[1], register[3]};
        01001: pair <= {register[1], register[4]};
        01010: pair <= {register[1], register[5]};
        01011: pair <= {register[1], register[6]};
        01100: pair <= {register[1], register[7]};
        01101: pair <= {register[2], register[3]};
        01110: pair <= {register[2], register[4]};
        01111: pair <= {register[2], register[5]};
        10000: pair <= {register[2], register[6]};
        10001: pair <= {register[2], register[7]};
        10010: pair <= {register[3], register[4]};
        10011: pair <= {register[3], register[5]};
        10100: pair <= {register[3], register[6]};
        10101: pair <= {register[3], register[7]};
        10110: pair <= {register[4], register[5]};
        10111: pair <= {register[4], register[6]};
        11000: pair <= {register[4], register[7]};
        11001: pair <= {register[5], register[6]};
        11010: pair <= {register[5], register[7]};
        11011: pair <= {register[6], register[7]};
    endcase

end



always @(posedge clk)
begin

    $monitor("%t, accumulator = %b" , $time, ac);
    if(wa_rn)
    begin
        ac <= 0;
        register[0]<= 0;
        register[1]<= 0;
        register[2]<= 0;
        register[3]<= 0;
        register[4]<= 0;
        register[5]<= 0;
        register[6]<= 0;
        register[7]<= 0;
    end

    if (w_a)
    begin
        ac <= data_in;
    end
    if (w_rn)
    begin
        register[reg_sel] <= data_in;
    end
end

assign rb = register[0];
assign rc = register[1];
assign rd = register[2];
assign re = register[3];
assign rf = register[4];
assign rg = register[5];
assign rh = register[6];
assign ri = register[7];

endmodule
```

## MEMORY

```
module Memory(
             output [7:0] out_data,
             output [15:0] out_address,
             input [7:0] data,
             input [15:0] address,
             input rd, wr, rd_ab, wr_s, rd_s,
             input [15:0] areg,
             input [15:0] sp,
             input [15:0] TestAd,
             input [7:0] TestDat,
             input clk
             );

reg [7:0] mem[59999:0];
reg [15:0] stack_mem[3999:0];
integer i;


initial
begin
    for(i = 0 ; i<= 59999; i = i+1)
    begin
      mem[i] <= 8'b0;
    end

    for( i = 0; i<=3999; i = i+1)
    begin
      stack_mem[i] <= 16'b0;
    end

 //   out_data = 8'b0;
 //   out_address = 16'b0;
end
//** making all outputs asynchronous
assign out_data = mem[areg];
assign out_address = rd_ab? {mem[areg+1], mem[areg + 2]} : stack_mem[sp];
```

```
always @(*)
begin

    mem[TestAd] = TestDat;
    $monitor("at time, %t, m0 = %b", $time, mem[0]);
    $monitor("at time, %t, m1 = %b", $time, mem[16'b0000000000000001]);
    $monitor("at time, %t, m2 = %b", $time, mem[16'b0000000000000010]);
    $monitor("at time, %t, m3 = %b", $time, mem[16'b0000000000000011]);
    $monitor("at time, %t, m4 = %b", $time, mem[16'b0000000000000100]);
    $monitor("at time, %t, m1jump = %b", $time, mem[16'b0000000010000000]);
    $monitor("at time, %t, jumpmem = %b", $time, mem[16'b1000000000000000]);
    $monitor("at time, %t, STACK = %b", $time, stack_mem[1]);



    if(wr == 1)
    begin
        mem[areg] <= data ;
    end


    if(wr_s == 1)
    begin
        stack_mem[sp] <= address;
    end


end

// add program to memory for execution.


Endmodule
```

## MID MODULE

```
module MidModule(from_memdata, from_memad, clk, rst, rbout, rcout, rdout,
                 reout, rfout, rgout, rhout, riout, instruction, to_memdata,
                 to_memad, RD, wr, rd_ab, wr_s, rd_s, areg, sp); //control signals & I/Os
                                                                  // for memory block

input clk, rst;
input [7:0] from_memdata;
input [15:0] from_memad;
output [7:0] to_memdata;
output [15:0] to_memad;
output  RD, wr, rd_ab, wr_s, rd_s;
output [15:0] areg;
output [15:0] sp;
//output reg [7:0] rb, rc, rd ,re, rf, rg, rh, ri;

output [7:0] rbout, rcout, rdout ,reout, rfout, rgout, rhout, riout;


//wire [7:0] Rb, Rc, Rd ,Re, Rf, Rg, Rh, Ri;

output [7:0] instruction;

//control signals
wire r_or, w_or,w_ir,i_pc,i2_pc, w_pc, r_pc,p_ar, a_ar,r_a,
     r_rn, w_rn, r_rp,w_a, wa_rn, r_sp, i_sp, d_sp, cl_fr, E, IF;

// busses
wire [7:0] databus;
wire [15:0] addressbus;

// data and address bus connections
```

```
wire [7:0] from_memdata, r_out, or_out, alu_out;
wire [15:0] rp_out, pc_out, from_memad;

//wire connections
wire [3:0] flags;
wire [7:0] or_ALU;
wire [7:0] ac_ALU;
wire [7:0] ir_out;
wire [15:0] pc_ar;
wire [3:0] flag_cu;
wire [7:0] opcode;
wire [2:0] stage;


assign instruction = ir_out;

assign    opcode = ir_out;

assign    databus = (RD)? from_memdata :(
              (r_rn)? r_out     :(
              (r_a)?  r_out     :( //data_out for register array can be AC or RN
              (r_or)? or_out     : alu_out)));

assign    addressbus = (r_rp)? rp_out :(
                 (r_pc)? pc_out :(
                 (rd_ab | rd_s)? from_memad : 16'b0 ));

assign    to_memdata = databus;
assign    to_memad = addressbus;


always @(*)
begin
    $monitor("At time, %t, databus = %b" , $time, databus);
    $monitor("At time, %t, addressbus = %b" , $time, addressbus);
end

ControlUnit control(RD, wr, rd_ab, wr_s, rd_s,r_or, w_or,w_ir,i_pc,
                    i2_pc, w_pc, r_pc,p_ar, a_ar,r_a, r_rn, w_rn, r_rp,
                    w_a, wa_rn, r_sp, i_sp, d_sp, cl_fr, E, IF, clk, opcode, rst, flags, stage);

ALU Alu(
         .clk(clk),
         .Out(alu_out),
         .Flag(flags),
         .RN(databus),
         .OD(or_ALU),
         .AC(ac_ALU),
         .opcode(opcode),
         .stage(stage)
        );

OperandRegister  OpR( .clk(clk),
                      .rst(rst),
                      .r_or(r_or),
                      .w_or(w_or),
                      .data(databus),
                      .out_or(or_out),
                      .alu_out(or_ALU)
                     );
AddressRegister AR( .clk(clk),
                    .rst(rst),
                    .PC_in(pc_ar),
                    .AB_in(addressbus),
                    .P_ar(p_ar),
                    .A_ar(a_ar),
                    .AdReg_out(areg)

                 );

FlagRegister FR( .rst(rst),
                .ALU_in(flags),
                .CL_f(cl_fr),
                .cu_out(flag_cu)

               );

InstructionRegister IR( .rst(rst),
                        .clk(clk),
                        .w_ir(w_ir),
                        .data(databus),
                        .out_ir(ir_out)

                       );

ProgramCounter PC( .rst(rst),
                   .clk(clk),
                   .i_pc(i_pc),
                   .i2_pc(i2_pc),
                   .w_pc(w_pc),
                   .r_pc(r_pc),
                   .address(addressbus),
                   .out(pc_out),
                   .ar_out(pc_ar)

                  );

StackPointer SP( .clk(clk),
                 .rst(rst),
                 .I_sp(i_sp),
                 .D_sp(d_Sp),
                 .Mem_out(sp)

              );

RegisterArray RA( .rst(rst),
                  .data_in(databus),
                  .r_a(r_a),
                  .r_rn(r_rn),
                  .w_rn(w_rn),
                  .wa_rn(wa_rn),
                  .w_a(w_a),
                  .r_rp(r_rp),
                  .opcode(opcode),
                  .clk(clk),
                  .data_out(r_out),
                  .addr_out(rp_out),
                  .ac_ALU(ac_ALU),

                  .rb(rbout),
                  .rc(rcout),
                  .rd(rdout),
                  .re(reout),
                  .rf(rfout),
                  .rg(rgout),
                  .rh(rhout),
                  .ri(riout)
```

## TOP MODULE

```
module Top_Module(clk, rst, TestDat, TestAd, rbf, rcf, rdf ,ref, rff, rgf, rhf, rif,
instructionout );

input clk, rst;
input  [7:0] TestDat;
input  [15:0] TestAd;
output [7:0] rbf, rcf, rdf ,ref, rff, rgf, rhf, rif, instructionout;

wire [7:0] to_memdata, from_memdata;
wire [15:0] to_memad, from_memad;
wire RD;
wire wr, rd_ab, wr_s, rd_s;
wire [15:0] areg;
wire [15:0] sp;

MidModule  MM(
               .rbout(rbf),
               .rcout(rcf),
               .rdout(rdf),
               .reout(ref),
               .rfout(rff),
               .rgout(rgf),
               .rhout(rhf),
               .riout(rif),
               .instruction(instructionout),
               .to_memdata(to_memdata),
               .to_memad(to_memad),
               .RD(RD),
               .wr(wr),
               .rd_ab(rd_ab),
               .wr_s(wr_s),
               .rd_s(rd_s),
               .areg(areg),
               .sp(sp),
               .from_memdata(from_memdata),
               .from_memad(from_memad),
               .clk(clk),
               .rst(rst)
                );

Memory Mem(.out_data(from_memdata),
           .out_address(from_memad),
           .data(to_memdata),
           .address(to_memad),
           .rd(RD),
           .wr(wr),
           .rd_ab(rd_ab),
           .wr_s(wr_s),
           .rd_s(rd_s),
           .areg(areg),
           .sp(sp),
           .TestAd(TestAd),
           .TestDat(TestDat),
           .clk(clk)

         );


Endmodule
```

## ASCII TEST

```
module ascii_test(
    input clkvga, clk,
    input [15:0] enter,
    input clk2, clk3, rst,
    input video_on,
    input [9:0] x, y,
    output reg [11:0] rgb
    );


    // signal declarations
    wire [10:0] rom_addr;           // 11-bit text ROM address
    wire [6:0] ascii_char;          // 7-bit ASCII character code
    wire [3:0] char_row;            // 4-bit row of ASCII character
    wire [2:0] bit_addr;            // column number of ROM data
    wire [7:0] rom_data;            // 8-bit row data from text ROM
    wire ascii_bit, ascii_bit_on;     // ROM bit and status signal
    wire [7:0] instructionout, R1,R2,R3,R4,R5,R6,R7,R8;

    wire [15:0] TestAd;
    wire [7:0] TestDat;

    reg [15:0] testadreg;
    reg [7:0] testdatreg;

   always @(*)
   begin
        if(clk3 == 1'b1)
        begin
            testadreg <= enter;
        end
        else if(clk2 == 1'b1)
        begin
            testdatreg <= enter[7:0];
        end
   end

   assign TestAd = testadreg;
   assign TestDat = testdatreg;

    Top_Module topm(.clk(clk), .rst(rst), .TestDat(TestDat), .TestAd(TestAd), .rbf(R1), .rcf(R2),
.rdf(R3) ,.ref(R4), .rff(R5), .rgf(R6), .rhf(R7), .rif(R8), .instructionout(instructionout) );

    // instantiate ASCII ROM
    ascii_rom rom(.clkvga(clkvga), .addr(rom_addr), .data(rom_data));

    // ASCII ROM interface
    assign rom_addr = {ascii_char, char_row};   // ROM address is ascii code + row
    assign ascii_bit = rom_data[~bit_addr];     // reverse bit order

    //assign ascii_char = {y[5:4], x[7:3]};   // 7-bit ascii code
    assign ascii_char = ((x >= 192 && x < 200) && (y >= 208 && y < 336)) ? 7'h52 :
                        ((x >= 201 && x < 208) && (y >= 208 && y < 224)) ? 7'h42 :
                        ((x >= 201 && x < 208) && (y >= 224 && y < 240)) ? 7'h43 :
                        ((x >= 201 && x < 208) && (y >= 240 && y < 256)) ? 7'h44 :
                        ((x >= 201 && x < 208) && (y >= 256 && y < 272)) ? 7'h45 :
                        ((x >= 201 && x < 208) && (y >= 272 && y < 288)) ? 7'h46 :
                        ((x >= 201 && x < 208) && (y >= 288 && y < 304)) ? 7'h47 :
                        ((x >= 201 && x < 208) && (y >= 304 && y < 320)) ? 7'h48 :
                        ((x >= 201 && x < 208) && (y >= 320 && y < 336)) ? 7'h49 :
                        ((x >= 209 && x < 216) && (y >= 208 && y < 336)) ? 7'h3D :
                        ((x >= 217 && x < 224) && (y >= 208 && y < 224)) ? (R1[7:4]<10) ? {3'h3,
R1[7:4]} : {3'h4, (R1[7:4] - 4'h9)} :
                        ((x >= 217 && x < 224) && (y >= 224 && y < 240)) ? (R2[7:4]<10) ? {3'h3,
R2[7:4]} : {3'h4, (R2[7:4] - 4'h9)} :
                        ((x >= 217 && x < 224) && (y >= 240 && y < 256)) ? (R3[7:4]<10) ? {3'h3,
R3[7:4]} : {3'h4, (R3[7:4] - 4'h9)} :
                        ((x >= 217 && x < 224) && (y >= 256 && y < 272)) ? (R4[7:4]<10) ? {3'h3,
R4[7:4]} : {3'h4, (R4[7:4] - 4'h9)} :
                        ((x >= 217 && x < 224) && (y >= 272 && y < 288)) ? (R5[7:4]<10) ? {3'h3,
R5[7:4]} : {3'h4, (R5[7:4] - 4'h9)} :
                        ((x >= 217 && x < 224) && (y >= 288 && y < 304)) ? (R6[7:4]<10) ? {3'h3,
R6[7:4]} : {3'h4, (R6[7:4] - 4'h9)} :
                        ((x >= 217 && x < 224) && (y >= 304 && y < 320)) ? (R7[7:4]<10) ? {3'h3,
R7[7:4]} : {3'h4, (R7[7:4] - 4'h9)} :
                        ((x >= 217 && x < 224) && (y >= 320 && y < 336)) ? (R8[7:4]<10) ? {3'h3,
R8[7:4]} : {3'h4, (R8[7:4] - 4'h9)} :
                        ((x >= 225 && x < 232) && (y >= 208 && y < 224)) ? (R1[3:0]<10) ? {3'h3,
R1[3:0]} : {3'h4, (R1[3:0] - 4'h9)} :
                        ((x >= 225 && x < 232) && (y >= 224 && y < 240)) ? (R2[3:0]<10) ? {3'h3,
R2[3:0]} : {3'h4, (R2[3:0] - 4'h9)} :
                        ((x >= 225 && x < 232) && (y >= 240 && y < 256)) ? (R3[3:0]<10) ? {3'h3,
R3[3:0]} : {3'h4, (R3[3:0] - 4'h9)} :
                        ((x >= 225 && x < 232) && (y >= 256 && y < 272)) ? (R4[3:0]<10) ? {3'h3,
R4[3:0]} : {3'h4, (R4[3:0] - 4'h9)} :
                        ((x >= 225 && x < 232) && (y >= 272 && y < 288)) ? (R5[3:0]<10) ? {3'h3,
R5[3:0]} : {3'h4, (R5[3:0] - 4'h9)} :
```

```
                        ((x >= 225 && x < 232) && (y >= 288 && y < 304)) ? (R6[3:0]<10) ? {3'h3,
R6[3:0]} : {3'h4, (R6[3:0] - 4'h9)} :
                        ((x >= 225 && x < 232) && (y >= 304 && y < 320)) ? (R7[3:0]<10) ? {3'h3,
R7[3:0]} : {3'h4, (R7[3:0] - 4'h9)} :
                        ((x >= 225 && x < 232) && (y >= 320 && y < 336)) ? (R8[3:0]<10) ? {3'h3,
R8[3:0]} : {3'h4, (R8[3:0] - 4'h9)} : 7'b0;

    assign char_row = y[3:0];               // row number of ascii character rom
    assign bit_addr = x[2:0];               // column number of ascii character rom
    // "on" region in center of screen
    //assign ascii_bit_on = ((x >= 192 && x < 448) && (y >= 208 && y < 272)) ? ascii_bit :
1'b0;//if the bit coming from ascii rom
    // is 1, set the pizel on screen coloured
    assign ascii_bit_on = ((x >= 192 && x < 296) && (y >= 208 && y < 336)) ? ascii_bit : 1'b0;

    // rgb multiplexing circuit
    always @*
        if(~video_on)
            rgb = 12'h000;      // blank
        else
            if(ascii_bit_on)
                rgb = 12'h00F;  // blue letters
            else
                rgb = 12'hFFF;  // white background



endmodule
```

## VGA CONTROLLER

```
module vga_controller(
    input clk_100MHz,   // from Basys 3
    input reset,        // system reset
    output video_on,    // ON while pixel counts for x and y and within display area
    output hsync,       // horizontal sync
    output vsync,       // vertical sync
    output p_tick,      // the 25MHz pixel/second rate signal, pixel tick
    output [9:0] x,     // pixel count/position of pixel x, max 0-799
    output [9:0] y      // pixel count/position of pixel y, max 0-524
    );

    // Based on VGA standards found at vesa.org for 640x480 resolution
    // Total horizontal width of screen = 800 pixels, partitioned  into sections
    parameter HD = 640;             // horizontal display area width in pixels
    parameter HF = 48;              // horizontal front porch width in pixels
    parameter HB = 16;              // horizontal back porch width in pixels
    parameter HR = 96;              // horizontal retrace width in pixels
    parameter HMAX = HD+HF+HB+HR-1; // max value of horizontal counter = 799
    // Total vertical length of screen = 525 pixels, partitioned into sections
    parameter VD = 480;             // vertical display area length in pixels
    parameter VF = 10;              // vertical front porch length in pixels
    parameter VB = 33;              // vertical back porch length in pixels
    parameter VR = 2;               // vertical retrace length in pixels
    parameter VMAX = VD+VF+VB+VR-1; // max value of vertical counter = 524

    // *** Generate 25MHz from 100MHz *********************************************************
        reg  [1:0] r_25MHz;
        wire w_25MHz;

        always @(posedge clk_100MHz or posedge reset)
                if(reset)
                  r_25MHz <= 0;
                else
                  r_25MHz <= r_25MHz + 1;

        assign w_25MHz = (r_25MHz == 0) ? 1 : 0; // assert tick 1/4 of the time
    // ****************************************************************************************
    // Counter Registers, two each for buffering to avoid glitches
    reg [9:0] h_count_reg, h_count_next;
    reg [9:0] v_count_reg, v_count_next;

    // Output Buffers
    reg v_sync_reg, h_sync_reg;
    wire v_sync_next, h_sync_next;

    // Register Control
    always @(posedge clk_100MHz or posedge reset)
        if(reset) begin
            v_count_reg <= 0;
            h_count_reg <= 0;
            v_sync_reg  <= 1'b0;
            h_sync_reg  <= 1'b0;
        end
        else begin
            v_count_reg <= v_count_next;
            h_count_reg <= h_count_next;
            v_sync_reg  <= v_sync_next;
            h_sync_reg  <= h_sync_next;
        end

    //Logic for horizontal counter
    always @(posedge w_25MHz or posedge reset)      // pixel tick
        if(reset)
            h_count_next = 0;
        else
            if(h_count_reg == HMAX)                 // end of horizontal scan
                h_count_next = 0;
            else
                h_count_next = h_count_reg + 1;

    // Logic for vertical counter
    always @(posedge w_25MHz or posedge reset)
        if(reset)
            v_count_next = 0;
        else
            if(h_count_reg == HMAX)                 // end of horizontal scan
                if((v_count_reg == VMAX))           // end of vertical scan
                    v_count_next = 0;
                else
                    v_count_next = v_count_reg + 1;

    // h_sync_next asserted within the horizontal retrace area
    assign h_sync_next = (h_count_reg >= (HD+HB) && h_count_reg <= (HD+HB+HR-1));

    // v_sync_next asserted within the vertical retrace area
    assign v_sync_next = (v_count_reg >= (VD+VB) && v_count_reg <= (VD+VB+VR-1));

    // Video ON/OFF - only ON while pixel counts are within the display area
    assign video_on = (h_count_reg < HD) && (v_count_reg < VD); // 0-639 and 0-479 respectively

    // Outputs
    assign hsync  = h_sync_reg;
    assign vsync  = v_sync_reg;
    assign x      = h_count_reg;
    assign y      = v_count_reg;
    assign p_tick = w_25MHz;

endmodule
```

## CONSTRAINT FILE

```
## This file is a general .xdc for the Nexys A7-100T
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal
names in the project

## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clkvga }];
#IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clkvga}];

#set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }];
#IO_L12P_T1_MRCC_35 Sch=clk100mhz
#create_clock -add -name sys_clk_pin -period 20.00 -waveform {0 10} [get_ports {clk}];

set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clk2_IBUF];
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clk3_IBUF];
set_property ALLOW_COMBINATORIAL_LOOPS TRUE [get_nets at/topm/MM/Alu/D[7]];

##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports {enter[0]}];
#IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports {enter[1]}];
#IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 } [get_ports {enter[2]}];
#IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports {enter[3]}];
#IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 } [get_ports {enter[4]}];
#IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports {enter[5]}];
#IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports {enter[6]}];
#IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 } [get_ports {enter[7]}];
#IO_L5N_T0_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 } [get_ports {enter[8] }];
#IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 } [get_ports {enter[9]}]; #IO_25_34
Sch=sw[9]
set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 } [get_ports {enter[10]}];
#IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 } [get_ports {enter[11]}];
#IO_L23P_T3_A03_D19_14 Sch=sw[11]
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports {enter[12]}];
#IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 } [get_ports {enter[13]}];
#IO_L20P_T3_A08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports {enter[14]}];
#IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports {enter[15]}];
#IO_L21P_T3_DQS_14 Sch=sw[15]


##Buttons
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { reset }];
#IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { clk2 }];
#IO_L4N_T0_D05_14 Sch=btnu
set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { clk }];
#IO_L12P_T1_MRCC_14 Sch=btnl
set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { clk3 }];
#IO_L10N_T1_D15_14 Sch=btnr
set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { rst }];
#IO_L9N_T1_DQS_D13_14 Sch=btnd

##VGA Connector
set_property -dict { PACKAGE_PIN A3    IOSTANDARD LVCMOS33 } [get_ports { rgb[11] }];
#IO_L8N_T1_AD14N_35 Sch=vga_r[0]
set_property -dict { PACKAGE_PIN B4    IOSTANDARD LVCMOS33 } [get_ports { rgb[10] }];
#IO_L7N_T1_AD6N_35 Sch=vga_r[1]
set_property -dict { PACKAGE_PIN C5    IOSTANDARD LVCMOS33 } [get_ports { rgb[9] }];
#IO_L1N_T0_AD4N_35 Sch=vga_r[2]
set_property -dict { PACKAGE_PIN A4    IOSTANDARD LVCMOS33 } [get_ports { rgb[8] }];
```

```
#IO_L8P_T1_AD14P_35 Sch=vga_r[3]
set_property -dict { PACKAGE_PIN C6    IOSTANDARD LVCMOS33 } [get_ports { rgb[7] }];
#IO_L1P_T0_AD4P_35 Sch=vga_g[0]
set_property -dict { PACKAGE_PIN A5    IOSTANDARD LVCMOS33 } [get_ports { rgb[6] }];
#IO_L3N_T0_DQS_AD5N_35 Sch=vga_g[1]
set_property -dict { PACKAGE_PIN B6    IOSTANDARD LVCMOS33 } [get_ports { rgb[5] }];
#IO_L2N_T0_AD12N_35 Sch=vga_g[2]
set_property -dict { PACKAGE_PIN A6    IOSTANDARD LVCMOS33 } [get_ports { rgb[4] }];
#IO_L3P_T0_DQS_AD5P_35 Sch=vga_g[3]
set_property -dict { PACKAGE_PIN B7    IOSTANDARD LVCMOS33 } [get_ports { rgb[3] }];
#IO_L2P_T0_AD12P_35 Sch=vga_b[0]
set_property -dict { PACKAGE_PIN C7    IOSTANDARD LVCMOS33 } [get_ports { rgb[2] }];
#IO_L4N_T0_35 Sch=vga_b[1]
set_property -dict { PACKAGE_PIN D7    IOSTANDARD LVCMOS33 } [get_ports { rgb[1] }];
#IO_L6N_T0_VREF_35 Sch=vga_b[2]
set_property -dict { PACKAGE_PIN D8    IOSTANDARD LVCMOS33 } [get_ports { rgb[0] }];
#IO_L4P_T0_35 Sch=vga_b[3]
set_property -dict { PACKAGE_PIN B11   IOSTANDARD LVCMOS33 } [get_ports { hsync }]; #IO_L4P_T0_15
Sch=vga_hs
set_property -dict { PACKAGE_PIN B12   IOSTANDARD LVCMOS33 } [get_ports { vsync }];
#IO_L3N_T0_DQS_AD1N_15 Sch=vga_vs
```