**FPGA-Based RISC-V Instruction Set Development**

Submitted by

**Yang Zihao**

3036120117

*M.Sc.(Eng.)*

A dissertation submitted for the degree of

Master of Science in Engineering

At the

Department of Electrical and Electronic Engineering

The University of Hong Kong

In

July 2023

# DECLARATION

I am Yang Zihao, and I hereby declare that this thesis represents my own work, except where due acknowledgement is made, and that it has not been previously included in a thesis, dissertation, or report submitted to this University or any other institution for a degree, diploma or other qualifications.

Signed

[DECLARATION 2](#_Toc171863143)

**ABSTRACT**

The flexibility and open-source nature of the RISC-V instruction set architecture (ISA) have made it an attractive choice for FPGA-based development, enabling customized and optimized hardware solutions. This paper presents an in-depth exploration of the design and implementation of a RISC-V based processor on an FPGA platform. We begin by discussing the key features of RISC-V that make it particularly suitable for FPGA development, such as its modularity and extensibility. The design process is described, including the development of a RISC-V compliant core and the integration of peripheral interfaces tailored for specific application requirements.

I detail the hardware description language (HDL) coding, synthesis, and place-and-route processes, and provide insights into the optimization techniques used to balance performance, resource utilization, and power consumption. Additionally, the verification and validation methodologies are outlined, employing both simulation tools and on-chip testing to ensure functional correctness and reliability.

All in all, I need to develop hardware-software digital systems from high level functional specifications and prototype them on to FPGA hardware using a standard hardware description language and software programming language.

**Keywords**: Embedded Digital System Design, Embedded Processor Programming, Verilog, Data path and Control Path design, Hardware-Software Co-design

**Chapter 1: Introduction**

**1.1Background**

The RISC-V (pronounced as risk-five) architecture is an open-source instruction set architecture (ISA) that has gained significant attention in recent years due to its flexibility, modularity, and extensibility. This means, unlike proprietary architectures, you get access to the blueprints and can customize it as you see fit. As an open-source ISA, RISC-V allows for a wide range of customization options, enabling developers to create processors tailored to specific applications and use cases. This has led to its adoption in various industries, from embedded systems and IoT devices to high-performance computing and artificial intelligence. With RISC-V, the benefits are: cost-effective custom processors, innovative applications, and robust security implementations. The technology is considered as the future of processing, customizable in your hands.

**1.1.1 History and Evolution of RISC-V**

The key aspects of RISC-V architecture include its design principles, instruction set, register file, memory model, privilege levels, and implementations. We will also discuss the RISC-V ecosystem and community, as well as its applications in different sectors.

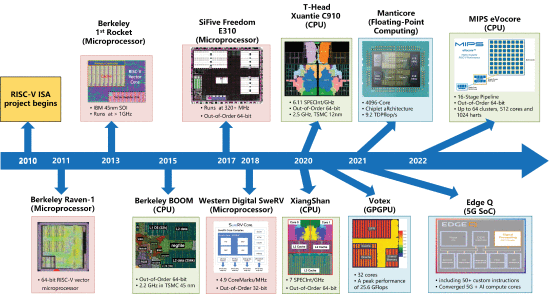


Figure 1: RISC-V Processor Evolution [1]

Proprietary ISAs were tightly controlled by specific companies, limiting access to their architecture and imposing licensing fees. This lack of openness hindered innovation, discouraged competition, and made it challenging for smaller companies or academic institutions to experiment and develop custom processors. This led to the rise of RISC-V.

Before RISC-V, there were several RISC (Reduced Instruction Set Computer) processors in the market. Notable examples include MIPS, SPARC, and PowerPC. These architectures were considered efficient and had their applications, but they often came with licensing costs and restricted access to their inner workings.

The origins of RISC-V can be traced back to the University of California, Berkeley, where it was initially developed as a research project in 2010. The project aimed to create a new, open-source ISA that would address the limitations of existing proprietary ISAs and provide a foundation for future processor designs. The RISC-V project was led by computer scientists Krste Asanović, Yunsup Lee, and Andrew Waterman, who were inspired by the success of open-source software and sought to bring similar benefits to the hardware domain.

The first version of the RISC-V ISA, known as the "RV32I" base integer instruction set, was released in 2011. This initial release focused on simplicity and efficiency, adhering to the principles of reduced instruction set computing (RISC). Over the years, the RISC-V ISA has evolved through several iterations, with the addition of new extensions and features to enhance its capabilities and address a broader range of applications.

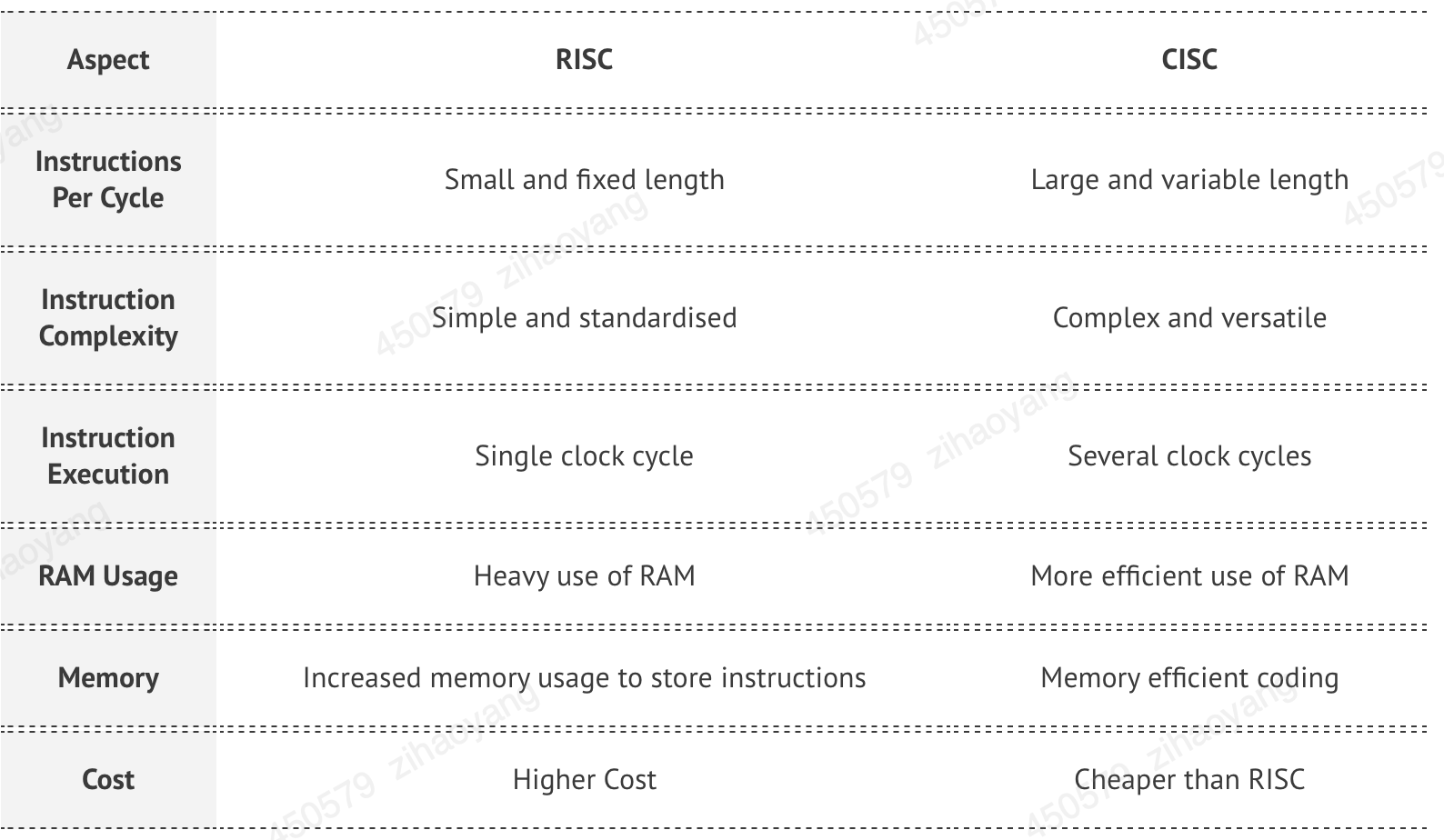
In 2015, the RISC-V Foundation was established to promote the adoption and standardisation of the RISC-V ISA. The foundation brought together industry leaders, academic institutions, and individual contributors to collaborate on the development and dissemination of RISC-V technology. Since its inception, the RISC-V Foundation has grown to include over 200 member organizations, and the RISC-V ISA has been adopted by numerous companies for various applications, from microcontrollers and embedded systems to high-performance computing and data centre processors.

The evolution of RISC-V has been driven by several factors, including the need for greater customization and flexibility in processor design, the desire to reduce the reliance on proprietary ISAs, and the growing demand for energy-efficient and cost-effective computing solutions. By providing an open, modular, and extensible ISA, RISC-V has enabled a new era of innovation in processor design and has the potential to reshape the landscape of the semiconductor industry.

At the core of the RISC-V architecture is the concept of reduced instruction set computing (RISC). RISC is a processor design philosophy that emphasizes simplicity and efficiency by using a small set of simple and general-purpose instructions. This contrasts with complex instruction set computing (CISC), which employs a larger set of more complex instructions that can perform multiple operations in a single instruction.

**1.1.2 RISC vs CISC**

RISC architectures prioritize simplicity and execute one instruction per clock cycle, resulting in streamlined designs and efficient decoding. CISC architectures, on the other hand, employ complex instructions capable of performing multiple actions but may require several clock cycles for execution. Both the CPUs aim to enhance CPU performance.



The RISC approach has several advantages over CISC:

1. **Simplifies Hardware Implementation**: It simplifies the hardware implementation of the processor, as fewer instructions need to be decoded and executed. This can lead to faster execution times and lower power consumption.
2. **Higher Instruction Level Parallelism**: RISC processors typically have a higher instruction-level parallelism, allowing them to execute multiple instructions simultaneously, which can further improve performance.
3. **Simplicity**: The simplicity of the RISC instruction set makes it easier to develop compilers and other software tools that can generate efficient code for the processor.

RISC-V adheres to the RISC philosophy by providing a minimal set of instructions that can be combined to perform complex operations. This simplicity allows for a more streamlined processor design, resulting in improved performance and energy efficiency. Additionally, the RISC-V instruction set is designed to be easily extensible, enabling the addition of new instructions and features as needed to address specific application requirements. This combination of simplicity and extensibility makes RISC-V a versatile and powerful foundation for processor design.

**1.1.3 Base Integer Instruction Set**

The base integer instruction set, also known as the "RV32I" or "RV64I" instruction set, depending on the address space size, provides the core functionality required for general-purpose computing. It includes instructions for arithmetic, logical, and control operations, as well as memory access and manipulation. The base integer instruction set is designed to be minimal and efficient, adhering to the principles of reduced instruction set computing (RISC).

RISC-V instructions are encoded using a fixed-length 32-bit format, which simplifies decoding and execution. The instruction formats are categorized into six types: R, I, S, B, U, and J. Each format serves a specific purpose and has a unique encoding structure:

* **R-type instructions**: Used for register-to-register operations, such as arithmetic and logical operations. They include three register operands: two source registers and one destination register. *Eg:- add (Add 2 registers and store results in another)*
* **I-type instructions**: Used for immediate operations, such as arithmetic and logical operations with an immediate value. They include two register operands and a 12-bit immediate value. *Eg:- li (Load immediate value)*
* **S-type instructions**: Used for store operations, which store data from a register to memory. They include two register operands and a 12-bit immediate value for the memory address offset. *Eg:- sw (store the value in register)*
* **B-type instructions**: Used for conditional branch operations, which transfer control to a different instruction based on a condition. They include two register operands and a 12-bit immediate value for the branch target address. *Eg:- beq (compare and label)*
* **J-type instructions**: Used for unconditional jump operations, which transfer control to a different instruction unconditionally. They include one register operand and a 20-bit immediate value for the jump target address. *Eg:- J (jump)*

Instruction set architectures (ISA), are standards for how the hardware of a processor should function and interact with the ISA’s own assembly language. ISAs have all the information necessary to create a processor that will run machine code correctly and consistently according to the standard [2]. This includes information about instructions, registers, memory access, arithmetic, data buses and so on. While every function of the processor is defined in the standard, how hardware and circuits are implemented is left up to the designer. Most ISAs will typically have many extensions and variations to suit the requirements of different designs. This could include extensions with support for multiplication, floating point numbers, or different data, address and instruction widths, or ISAs targeted for embedded systems, personal computers, super computers, etc.

Commercially successful ISAs have typically been proprietary and required licenses. With the specific designs and implementations being hidden behind patents and non- disclosure agreements. The new RISC-V ISA [3] is promising for the hardware industry with it being the first open-source modern ISA. RISC-V is a combination of work by the University of California and companies such as AMD, Google, Microsoft, IBM and many more. Since the finalisation of the first variants of RISC-V, many open-source tools and designs have been created and published [4].

RISC-V is not only exciting because it is open source, but also an ISA that can compete with the other restricted commercial ISAs by Intel, AMD and ARM. While most academic designs are optimised for learning purposes, RISC- V is intended for modern practical use.

The open-source aspect of RISC-V made it particularly well suited for this project as we could use it to create an open- source CPU without having to use one of the "toy" or academic ISAs, and instead use something more realistic and practical. While there already exist opensource implementations of RISC-V processors, none is simple and intuitive enough for a beginner to follow.

In this paper, I have designed a light weight, open-source implementation of a RISC-V processor using modern hardware design techniques and implemented the design onto a Field Programmable Gate Array (FPGA), and to test it. We wanted to create a RISC-V processor that is simple enough to learn from and lightweight enough to be implemented on even small FPGAs.

**1.2 Literature Review**

**1.2.1 Soft-Core Implementation vs Hard-Core Implementation**

RISC-V implementations refer to the various ways the RISC-V ISA can be realized in hardware. These implementations can range from soft-core designs that run on programmable logic devices, such as FPGAs, to hard-core designs that are integrated into custom silicon chips. Each type of implementation has its own advantages and disadvantages, depending on factors such as performance, power consumption, and development cost.

### Soft-Core Implementations

Soft-core RISC-V implementations are designs that can be synthesized and run on programmable logic devices, such as field-programmable gate arrays (FPGAs) or complex programmable logic devices (CPLDs). These implementations are typically written in hardware description languages (HDLs), such as Verilog or VHDL, and can be customized and configured to meet specific design requirements.

The benefits of soft-core RISC-V implementations include:

* **Flexibility**: Soft-core implementations can be easily modified and reconfigured to meet specific design requirements, making them ideal for prototyping and experimentation. This flexibility allows designers to optimize their processor design for performance, power consumption, or other design goals.
* **Lower Development Cost**: Developing a soft-core RISC-V implementation on an FPGA or CPLD can be more cost-effective than creating a custom silicon chip, especially for small-scale projects or proof-of-concept designs. This lower development cost can make RISC-V more accessible to a wider range of developers and organizations.
* **Faster time-to-market**: Soft-core implementations can be developed, tested, and deployed more quickly than hard-core implementations, as they do not require the lengthy fabrication process associated with custom silicon chips. This faster time-to-market can be a significant advantage for companies looking to bring new products to market quickly.

However, there are also some drawbacks to using soft-core RISC-V implementations:

**Lower Performance**: Soft-core implementations running on FPGAs or CPLDs typically have lower performance compared to hard-core implementations on custom silicon chips. This is due to factors such as the overhead of programmable logic and the limitations of FPGA or CPLD technology.

**Higher Power Consumption**: Soft-core implementations on FPGAs or CPLDs can consume more power than hard-core implementations, as programmable logic devices are generally less power-efficient than custom silicon chips.

Despite these drawbacks, soft-core RISC-V implementations can be an attractive option for many applications, particularly in the early stages of development or for projects with limited budgets and resources.

### Hard-Core Implementations

Hard-core RISC-V implementations are designs that are integrated directly into custom silicon chips, such as application-specific integrated circuits (ASICs) or system-on-chip (SoC) devices. These implementations are typically developed using a combination of custom logic and standard cell libraries, which provide pre-designed building blocks for creating complex digital circuits.

The advantages of hard-core RISC-V implementations include:

* **Higher Performance**: Hard-core implementations can offer higher performance compared to soft-core implementations, as they are designed and optimized specifically for the target silicon technology. This can result in faster clock speeds, lower latency, and improved instruction-level parallelism.
* **Lower Power Consumption**: Hard-core implementations can be more power-efficient than soft-core implementations, as they can take advantage of custom silicon optimizations and power management techniques. This can be particularly important for battery-powered devices or energy-constrained applications.
* **Higher Integration**: Hard-core RISC-V implementations can be integrated directly into a larger SoC design, which can include other components such as memory, peripherals, and accelerators. This high level of integration can lead to improved system performance, lower power consumption, and reduced board complexity.

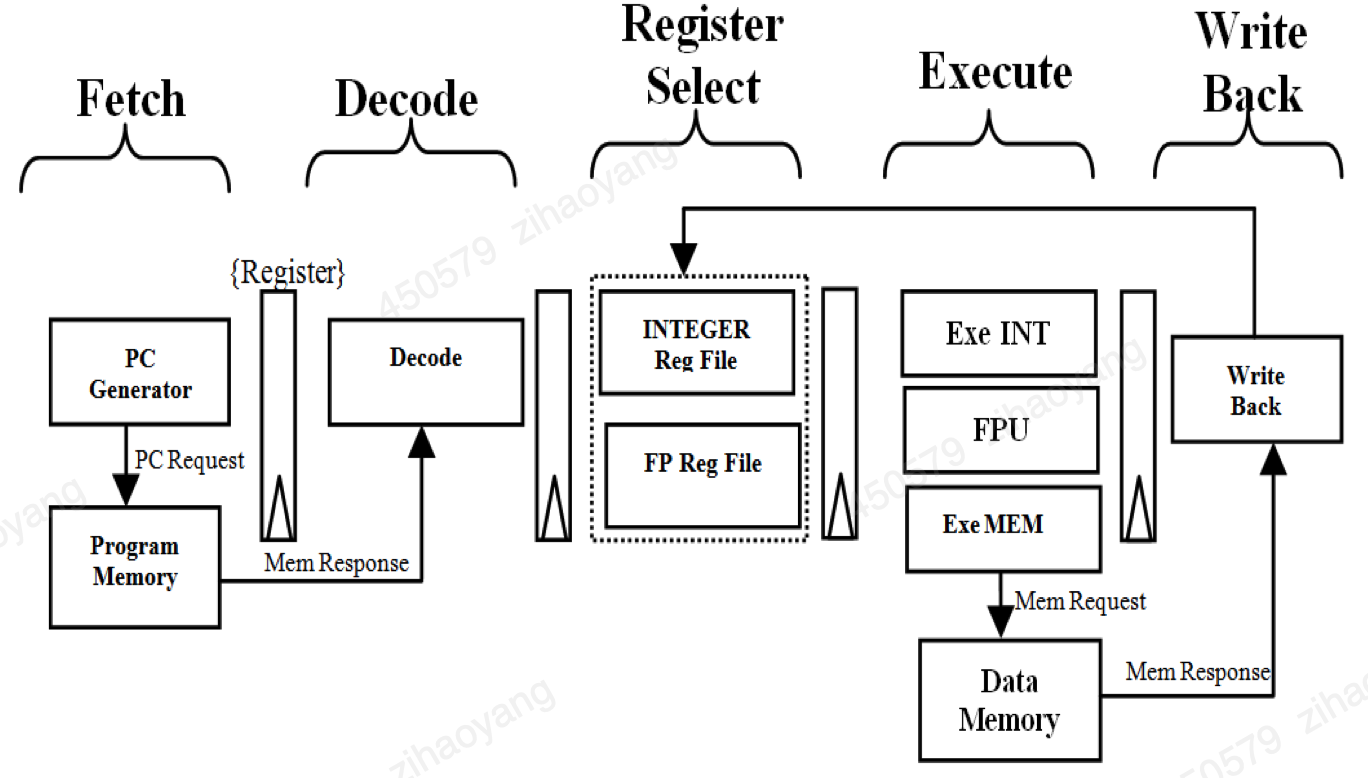
However, there are also some drawbacks to using hard-core RISC-V implementations:

* **Higher Development Cost**: Developing a custom silicon chip with a hard-core RISC-V implementation can be more expensive than using a soft-core implementation on an FPGA or CPLD, particularly for small-scale projects or proof-of-concept designs. The cost of chip fabrication, testing, and packaging can be significant, especially for cutting-edge silicon technologies.
* **Longer Development Time**: The process of designing, fabricating, and testing a custom silicon chip can be time-consuming, which can result in a longer time-to-market compared to soft-core implementations. This can be a disadvantage for companies looking to bring new products to market quickly.

Despite these challenges, hard-core RISC-V implementations can offer significant benefits in terms of performance, power consumption, and integration, making them an attractive option for many applications, particularly in high-performance or power-constrained systems.

**1.2.2 RISC-V Instruction Set Microprocessor Architecture**

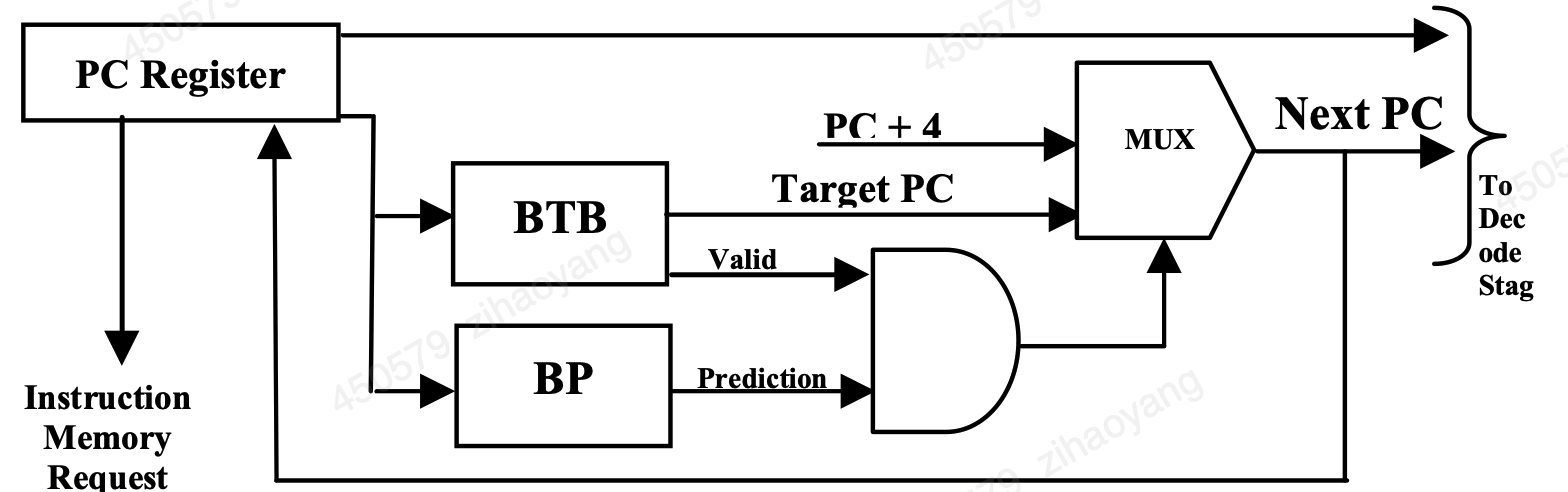
RISC-V instruction set processor adopts typical RISC micro-architecture with in-order fetch, out-of order execution and in-order completion. Proposed architecture comprises of 5 pipeline stages viz. Fetch, Decode, Register Select (RS), Execute (EXE) and Write Back (WB). Fig2 shows the architecture of 5-stage pipelined processor. In fetch stage, instruction is fetched from instruction memory from location indicated by Program Counter (PC). Instruction memory sends instruction to decode stage for instruction information extraction. The extracted information is forwarded to register select stage. From register select stage, the pipeline is split into three concurrent pipeline units, as integer instructions pass through integer execution pipeline, memory instructions pass through the memory pipeline and floating-point instructions pass through the FP execution pipeline unit. Write back stage receives the responses from all of these three concurrent execution stages and based on the scheduling of the instruction, WB stage allows commit the instructions in-order.

 **Figure 2: Top level architecture of RISC-V Processor**

A. Fetch Stage

Fig 3 shows the micro architecture of fetch unit. It computes the current PC and predicts the next PC value. Fetch unit has a 32 bit register to hold the current PC value, a 32-bit adder to calculate the next PC by adding 4 to the current PC. Next PC is predicted either as PC+4 or by using a sophisticated branch prediction scheme [4] [5].

Branch prediction scheme consists of Branch Target Buffer (BTB) and Branch Predication (BP) unit. BTB unit outputs a Boolean valid signal along with target PC address. BP unit provides Boolean information which contains whether the PC value is present or not. A multiplexer selects either PC+4 or predicted PC as next PC value based on the valid and prediction signals. The PC and next PC are given to the decoder stage for further processing. In this paper, the branch prediction is ignored since there are few branch instructions in this project.

****

**Figure 3: Micro Architecture of Fetch Unit**

B. Decode Stage

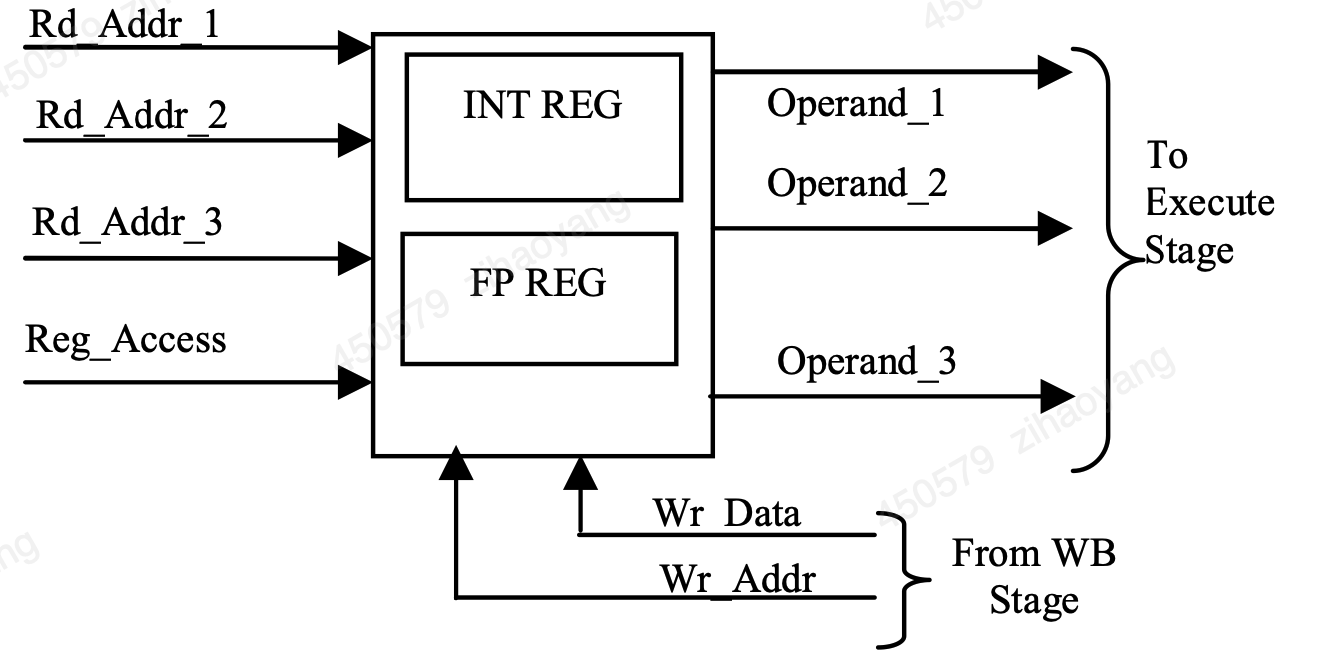
Instruction decoder receives the instructions from program memory with corresponding PC and predicted PC from fetch stage. Based on the lower 7-bit (Op-code) of instruction, instructions are classified as per Table 1. Subgroup information is decoded from 3-bit field (function field). Address of sources and destination registers are of 5-bit field each and their position is fixed irrespective of type of instruction. If an immediate type instruction has occurred, the immediate data is sign-extended to 32-bit for further processing. In this paper, the op-code is simplified to an 8 bits op-code because there are not many types of instructions and 32 bits are not required which would waste FPGA resource.

Decoder generates control signals for further processing of the instruction. Control signals include, Register Access, Register update, and pipeline information (the pipeline on which the instruction is to be scheduled). Register Access can be integer register file access (INT-ACCESS) or floating-point register access (FP-ACCESS) or NO-ACCESS.

C. Register Select Stage

RS stage accepts decoded information of instructions from decoder stage and selects the operand from either integer or floating-point register file. RS stage contains an integer (32- No’s of 32-bit wide) and floating-point register file (32-No’s of 64-bit wide). Register file is implemented as a Random Access Memory (RAM), which has a latency of one clock cycle with three read ports and one write port.

Fig4 shows the register file organization for RISC-V processor. Register file unit accepts three source addresses (Rd\_Addr\_1, Rd\_Addr\_2, Rd\_Addr\_3} and a control signal (Reg\_Access) which specifies the access of Register files (Integer Access or Floating-point Access). Register file output is 64-bit which holds the data from either integer or floating-point register file and has one 64-bit write port for write back.

****

**Figure 4: Register File Organization**

Based on the pipeline information from decode stage, operands after register select are passed either to integer ALU / memory address computation pipeline or FP execution pipeline. An instruction scheduler FIFO which is part of integer / memory / FPU execute stage is used to schedule the instructions in the various pipelines to commit the instructions in-order.

D. Execute Stage

Execute stage consists of three concurrent units for Integer arithmetic and logic operations etc., operand memory address computation and floating-point computation. Integer execution performs arithmetic (Addition, subtraction, multiplication and division) and logical (AND, OR, XOR and shift) operations. Also, Integer arithmetic unit calculates the target address for unconditional or conditional jump and branch instructions. Integer execution unit executes the system related instruction such as SCALL, SBREAK instructions for supervisor level access of the instruction. An efficient data forwarding scheme is used to forward output of execution units to the input of the execution unit. Detailed micro-architecture for data forwarding scheme is explained in section VI.

Unit for Memory related instructions calculate the target data memory address for load and store operations. RISC-V ISA supports load or store operations on a byte, half-word and word data to and from data memory. Micro-architecture of an out-of-order FP execution unit is explained in section V.

E. Write Back Stage

Write Back (WB) stage commits the instruction from the pipeline and updates the register file with the results from execute units. WB reads the instruction from top of the scheduled instruction FIFO and based on the pipeline information, it reads either from integer or memory or floating-point concurrent units.

**1.2.3 Logical and Arithmetic Unit**

It is a combinational circuit that operates arithmetically and bitwise on integer binary integers. It is a crucial part of many different computing circuits. The arithmetic operations are addition and subtraction. Relational operations, which test conditions like less than, not equal to, greater than equal, greater than, less than and equal to, are used to compare data. Rotate right, rotate left, right shift and left shift are all examples of shifting procedures. The logical operations XOR, AND & OR are a few examples.

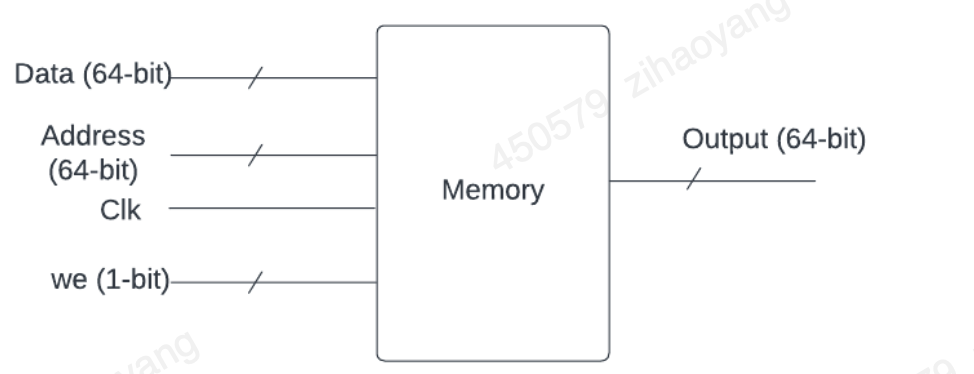


Figure 5: ALU diagram

The ALU performs eight different operations on its inputs and output the result. It can perform addition, subtraction, multiplication, division, logical AND, logical OR Logic Left shift and Logic right shift. Which operation is performed is determined by the control signal coming from the control unit, which in turn is determined by the instruction type. Naturally, the ALU is used by the **add**, **sub**, **and**, **or** instructions. However, also the **write, read** and **BREQ** instructions also require a calculation to be performed (such as the calculation of memory addresses). The ALU also has a “zero flag” output, which is a single-bit output that goes high when the result of an operation is zero. This is used during the **BREQ** instruction execution to decide whether to branch or not.

The data memory is an asynchronous read, synchronous write memory used for storing data generated by the program or data flashed during power up. It has a single address input shared during read and write operations. There is a write data input and a read data output. There are also control signals for enabling write and enabling read separately. The write data input is provided by the contents of one of the read register outputs of the register file. The read data output goes to the register file’s write data input. The address input is provided by the ALU output since the address for reading and writing needs to be calculated by adding a register from an instruction’s source register field to the offset field. Additionally, a small region of the data memory is read only and flashable on start-up which is useful for a programmer since the small instruction set doesn't provide any immediate instructions. This can be flashed with the same method as instruction memory by using a “.mem” file.

|  |  |
| --- | --- |
| **Instruction** | **Description** |
| Write | Write value from register to memory |
| Read | Read value from memory to register |
| Add A, B, Out | Addition |
| Sub A, B, Out | Subtraction |
| Multi A, B, Out | Multiplication |
| Divide A, B, Out | Division |
| Left Shift | Bit Operation left shift |
| Right Shift | Bit Operation right shift |
| And | Bitwise And operation |
| Or | Bitwise Or operation |
| BREQ | Branch to address ADDR i.e. load program counter with ADDR if register A’s content is equal to Register B’s |
| BGTQ | Branch to address ADDR i.e. load program counter with ADDR if register A’s content is greater than Register B’s |
| BLTQ | Branch to address ADDR i.e. load program counter with ADDR if register A’s content is less than Register B’s |
| Jump | Branch to address ADDR i.e. load program counter with ADDR |

Table 1

**Write** and **Read** are two memory related instructions. **Write** takes a value from a given memory address and puts it into a register. While **Read** does the opposite and takes a double from a register and stores it at an address in memory.

**Add** and **sub** are two arithmetic instructions. **Add** takes the contents of two source registers, adds them together using signed addition, and stores the results in a destination register. **Sub** subtracts the contents of the two source registers instead. If the result of an arithmetic operation overflows (i.e., the result cannot fit in a double integer), then the result will wrap around back through the negative numbers. As all arithmetic is signed, the negative numbers are represented in two’s complement.

In digital electronics, such as a computer, a binary multiplier is an electrical circuit that multiplies two binary integers. Due to its low power consumption, the Baugh Wooley signed multiplier was found to be the ideal option for our micro-architecture. Baugh-Wooley multiplication is one of the most cost-effective methods for dealing with sign bits. This method was created to create regular multipliers that are suitable for 2’s complement numbers. The Baugh Wooley signed multiplier circuit has a delay of 5.97 ns. If this circuit is used directly at the IE stage, the critical path for the entire system lengthens. As a result, the multiplier’s hardware circuit is split into two stages. Partially computed products are calculated in the first stage. They are then separated into six parts and added together. The adder’s output is then buffered for the subsequent step. The outcome is obtained in the second stage by again summing the six buffered results. There are 29 Fas and 1 HA (Half Adder) on the first stage’s important route. There are 1HA (half adder) and 29 Fas on the first stage’s important route (Full Adders). The first stage’s delay is kept lower than the second stage’s during the entirety of the partial product segmentation. This is because the multiplier’s input data could originate from the register bank or from different pipeline stages. There will therefore be a multiplexer delay at the multiplier’s inputs [7].



Figure 6: Multiplier block diagram

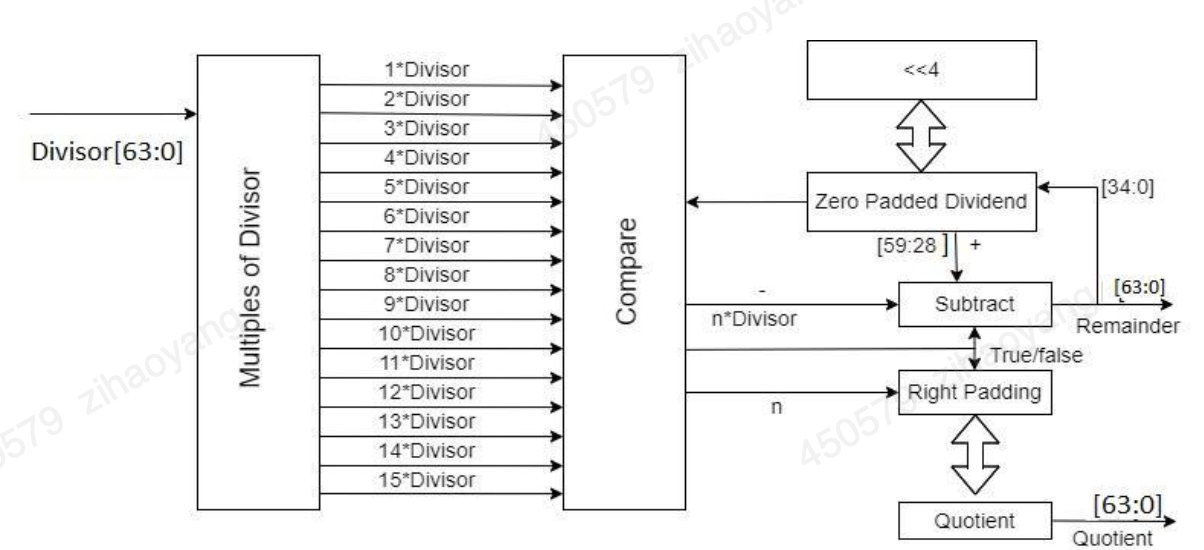
The divider unit for the suggested micro-architecture is chosen to be Radix-16. The results of the initial constant multiplication of the divisor are saved in registers. It takes a lot of multipliers, each of which is expensive in terms of size, time, and power, to calculate these constant multiplications. As a result, shifters and adders are used to accomplish these multiplications in hardware circuits. For multiples of 2, 4, and 8, the divisor is shifted to the left, and the other multiples are found by averaging the results. The updating of the quotient and remainder is the next step in the division process. At each clock cycle, the 36-bits of the dividend that are the most important are compared to its multiples that have been previously calculated.

Figure 6: Divider block diagram

**and** and **or** are the two logical instructions in the instruction set. **and** takes the contents of two source registers, performs a bitwise AND operation, and stores the results in a destination register. **or** does the same but performs a bitwise OR operation on the source registers instead.

A shifter is a digital circuit that can change the order of bits in a data word. It has the ability to shift N-bit data in a single cycle. Data can be shifted left or right using a Shifter. A Shifter can perform arithmetic shifting, logical shifting, and rotation functions in general. Bits are multiplied or divided by powers of two by shifters. A shifter, as the name implies, shifts a binary integer a predetermined number of places to the right or left. An N-bit shifter can be made using N N:1 multiplexers. Depending on the value of the log2N-bit select lines, the input is shifted by 0 to N 1 bits. A shifter is used to move the bits in the partial product of the multiplication and division result.

Figure 7: Shift diagram

**BREQ** is the only control instruction (a conditional instruction). BREQ stands for *“branch if equal”*. It takes the contents of two source registers and verifies if they are equal by checking if their difference by subtraction is zero. If they are equal, then the program will jump to a new instruction, based on a signed offset provided by the **BREQ** instruction. Normally the program counter register will increment through each instruction one at a time with each new clock cycle. However, when **BREQ** is used, the program counter can be incremented or decremented by a specified amount, jumping the program forwards or backwards by some number of instructions.

**BGTQ** is the only control instruction (a conditional instruction). BGTQ stands for *“branch if A is greater than B”*. It takes the contents of two source registers and verifies if register A is greater than register B by checking if their difference by subtraction is over zero. If register A is greater than B, then the program will jump to a new instruction, based on a signed offset provided by the **BGTQ** instruction. Normally the program counter register will increment through each instruction one at a time with each new clock cycle. However, when **BGTQ** is used, the program counter can be incremented or decremented by a specified amount, jumping the program forwards or backwards by some number of instructions.

**BLTQ** is the only control instruction (a conditional instruction). BLTQ stands for *“branch if register A is less than register B”*. It takes the contents of two source registers and verifies if register A is little than register B by checking if their difference by subtraction is less than zero. If register A is smaller than register B, then the program will jump to a new instruction, based on a signed offset provided by the **BLTQ** instruction. Normally the program counter register will increment through each instruction one at a time with each new clock cycle. However, when **BLTQ** is used, the program counter can be incremented or decremented by a specified amount, jumping the program forwards or backwards by some number of instructions.

**2. Digital system Design**

**2.1 Microprocessor-based System for Remote Car Control**

The following figure gives a block diagram of the overall FPGA-based system for remote car control. At its heart is a microprocessor which will be the master of an 8-bit bus to which various peripherals (slaves) will be connected.

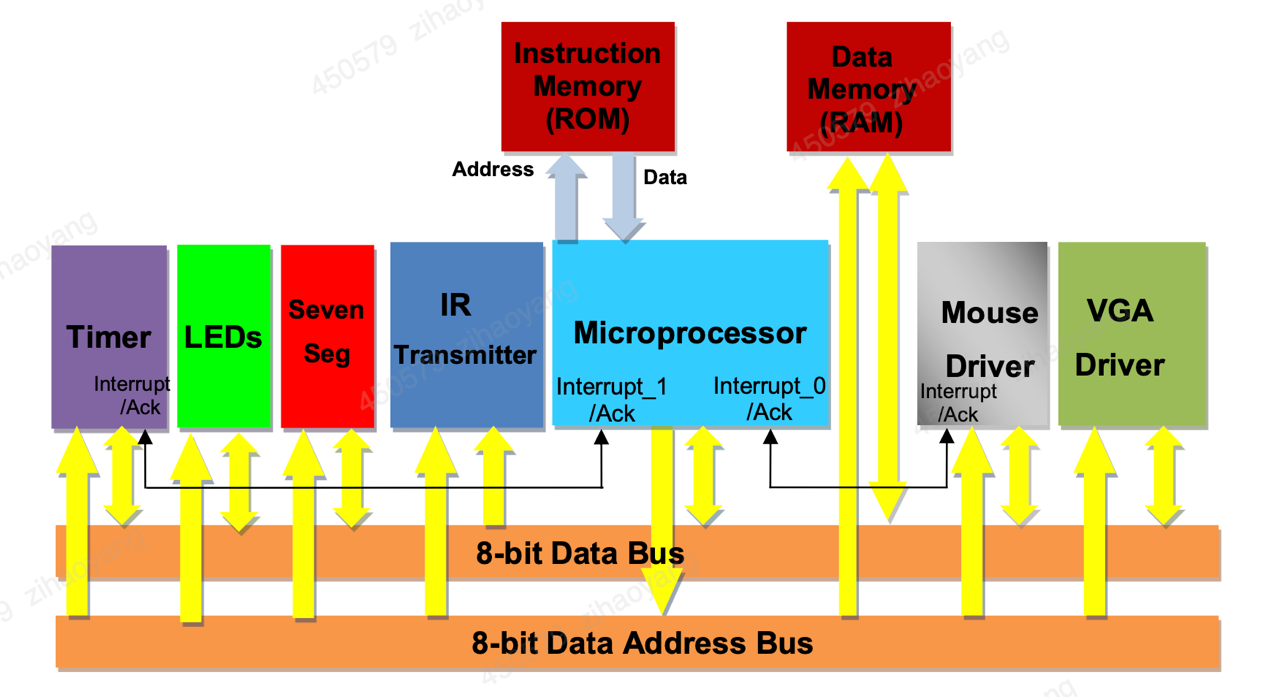
****

Figure 8: Microprocessor-based system block diagram

The following presents the memory mapping of the peripherals on the microprocessor’s data address bus.

|  |  |  |
| --- | --- | --- |
| **Peripheral** | **Base Address** | **High Address** |
| IR Transmitter | 0x90 | 0x90 |
| Mouse | 0xA0 | 0xA2 |
| VGA Display | 0xB0 | 0xB2 |
| LEDs | 0xC0 | 0xC0 |
| SevenSeg | 0xD0 | 0xD1 |
| Timer | 0xF0 | 0xF3 |
| Data Memeory | 0x00 | 0x7F |

**Table 2**

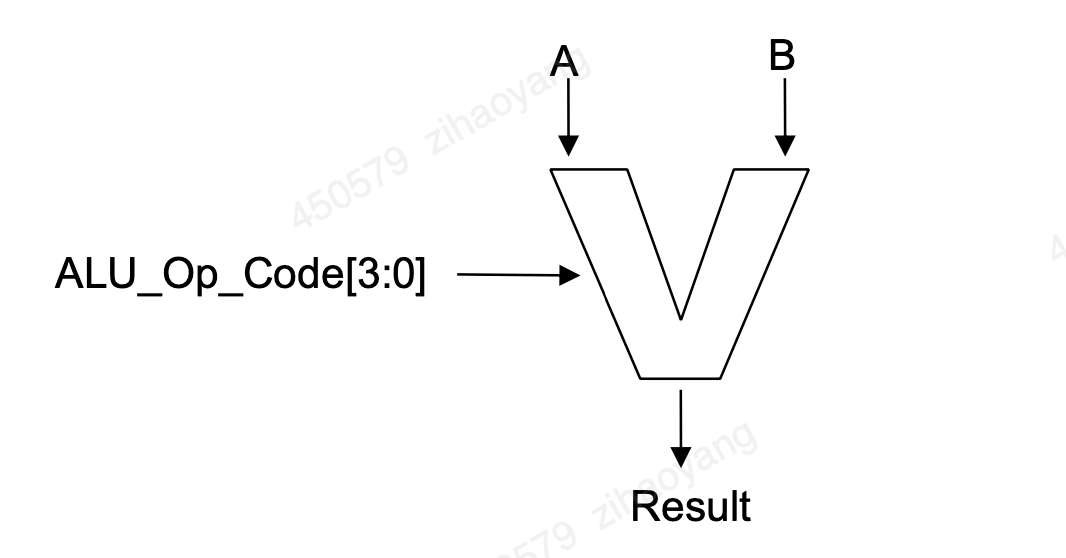
Note that the proposed microprocessor has a Harvard architecture which means that the instruction memory and data memory are distinct with separate address and data busses for each of them. Parallel accesses to both instructions and program data can thus be made, resulting in higher performance compared to Von-Neumann processors where instructions and data are stored on the same memory.

In this paper architecture, the data memory consists of 128 bytes and is mapped to the microprocessor’s address space as follow:

The microprocessor I design is an 8-bit processor with a load-store architecture whereby two internal registers (A and B) are used to hold operands and operation results. Central to the processor is an Arithmetic and Logic Unit (ALU) which can perform the following operations:

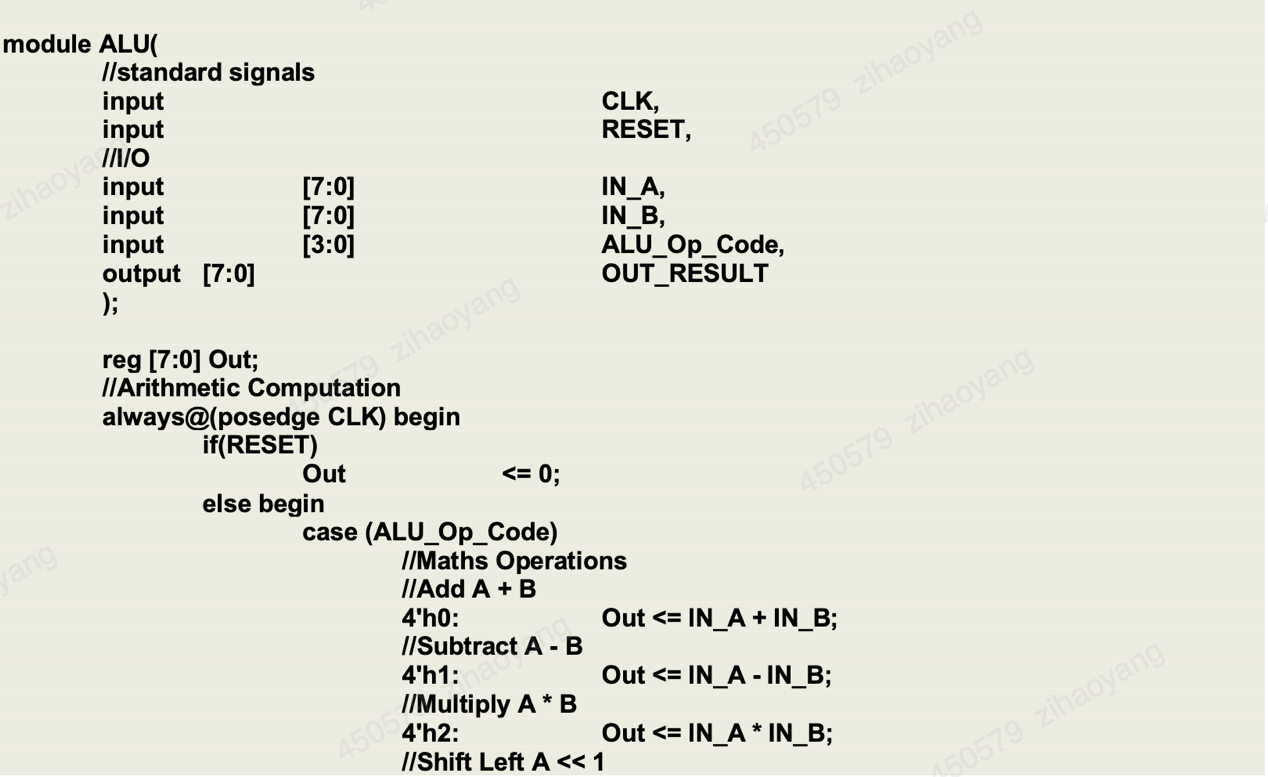
* -  Add two operands A, B
* -  Subtract two operands A, B
* -  Multiply two operands A, B
* -  Left shift operand A (by one bit)
* -  Right shift operand A (by one bit)
* -  Increment operand A
* -  Increment operand B
* -  Decrement operand A
* -  Decrement operand B
* -  Check if two operands A, B are equal (output 0x01 if that’s the case, otherwise 0x00)
* -  Check if A>B (output 0x01 if that’s the case, otherwise 0x00)
* -  Check if A<B (output 0x01 if that’s the case, otherwise 0x00)

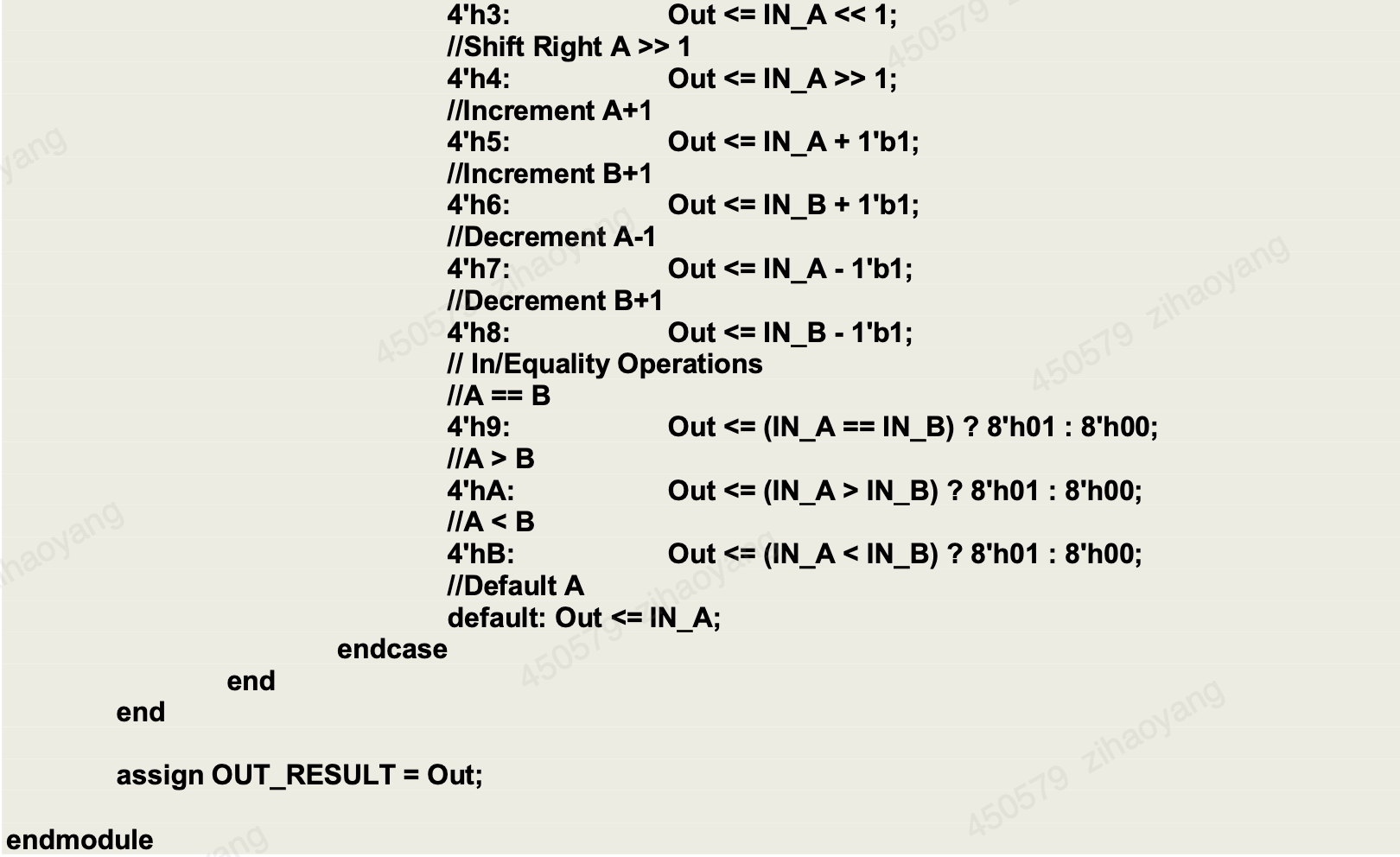
An operation code of four bits dictates which of these operations is performed (see Figure 14 below).



**Figure 9:**

Verilog code to implement the above ALU is given to you below.

****

****

The microprocessor has 6 types of instructions overall:

1. Read from memory to Register A or B
2. Write to memory from Register A or B
3. Do an ALU operation and save the result in register A or B
4. Jump operations: *conditional jump* depending on whether register A’s content

is equal, greater than or less than register B’s content, or *unconditional jump* to

a particular address.

1. Function call and return
2. Dereference Register A or B

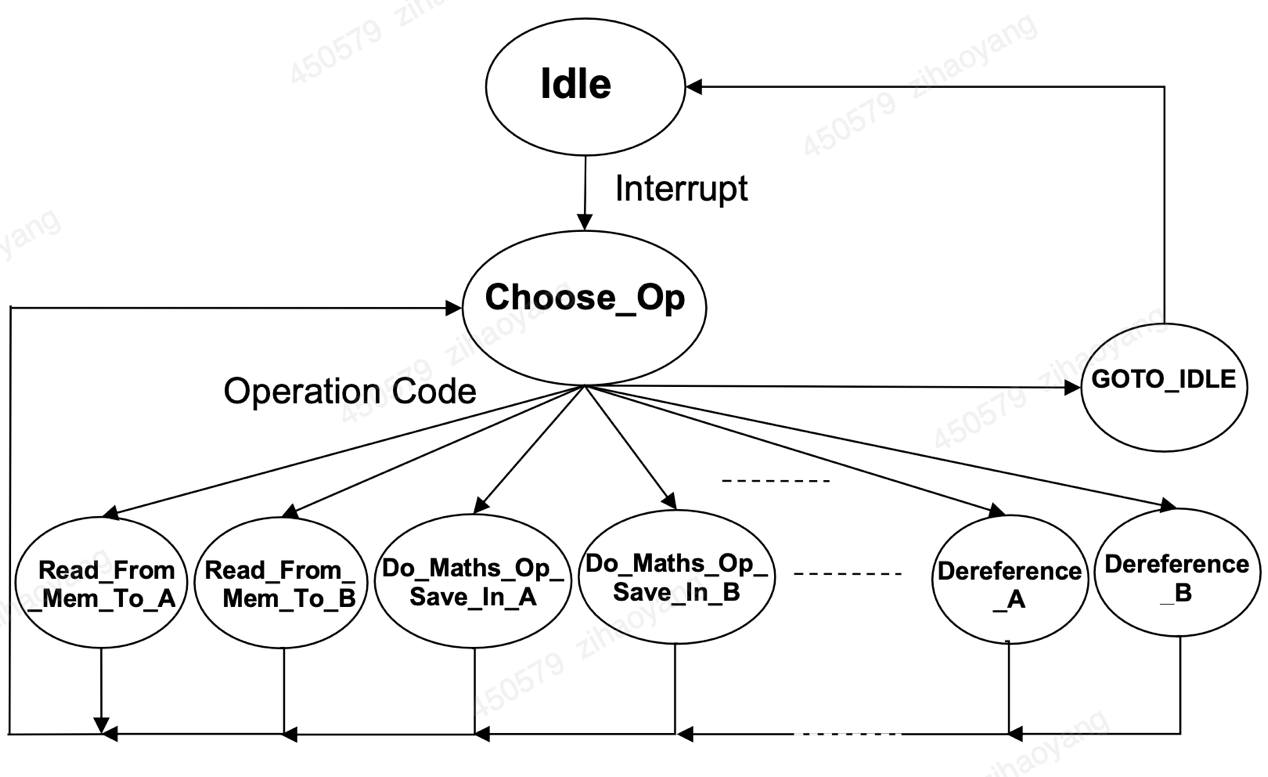
Different instructions have different word lengths, but they are always issued in consecutive bytes. The following table presents the instruction set architecture in more detail.

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Function** | **Instruction Structure** |
| A <- [Mem] | Read value from memory address Mem and store in register A |  |
| B <- [Mem] | Read value from memory address Mem and store in register B |  |
| [Mem] <- A | Write value of register A to memory address Mem |  |
| [Mem] <- B | Write value of address Mem register B to memory |  |
| A <- ALU\_OP(A,B) | Do Math ALU operation of type ALU\_Op\_Code on register value(s) and store result in register A |  |
| B <- ALU\_OP(A,B) | Do Math ALU operation of type ALU\_Op\_Code on register value(s) and store result in register B |  |
| BREQ ADDR | Branch to address ADDR i.e. load program counter with ADDR if register A’s content is equal to Register B’s |  |
| BGTQ ADDR | Branch to address ADDR i.e. load program counter with ADDR if register A’s content is greater than Register B’s |  |
| BLTQ ADDR | Branch to address ADDR i.e. load program counter with ADDR if register A’s content is less than Register B’s |  |
| GOTO ADDR | Branch to address ADDR i.e. load program counter with ADDR |  |
| GOTO\_IDLE | Go to Idle State and wait for Interrupts |  |
| FUNCTION\_CALL ADDR | Branch to memory address ADDR. Save the next program address to execute from after returning from the function (program context) |  |
| RETURN | Returns from a function call i.e. loads program context to the program counter for next instruction execution |  |
| Dereference A | Read memory address given by the value of register A and set the result as the new register A value A <- [A] |  |
| Dereference B | Read memory address given by the value of register B and set the result as the new register B value B <- [B] |  |

**Table 3**

Such a small instruction set with direct hardware support for its implementation is referred to as a RISC architecture (RISC = Reduced Instruction Set Computing), as opposed to CISC architecture (CISC = Complex Instruction Set Computing) whereby the instruction set is large and complex with further microcode translation needed for complex instruction execution. RISC architectures are dominant nowadays as they are simpler and faster in hardware.

The proposed microprocessor’s behaviour is essentially a state machine, with one sequential pipeline of states for each operation as shown in the figure 11.

****

**Figure 11: Microprocessor’s state machine**

**2.2 IR transmitter**

The BASYS 3 board provides four 12-pin peripheral module connectors. Each is organised in two rows of 6-pins, with each row providing a power pin (Vdd @3.3V), a ground pin (GND), and four unique FPGA signals (see BASYS 3 reference manual, page 17). (Note that one of the connectors is designed for JXADC. Use any of the other three for this lab). Several 6-pin (or 12-pin) module boards that can attach to this connector are available from Digilent (see www.digilentinc.com for more information). For the purpose of this lab, we have developed our own 6-pin IR Transmitter module, which can be attached to any of the three peripheral module connectors. The following gives the circuit diagram of the IR Transmitter module.

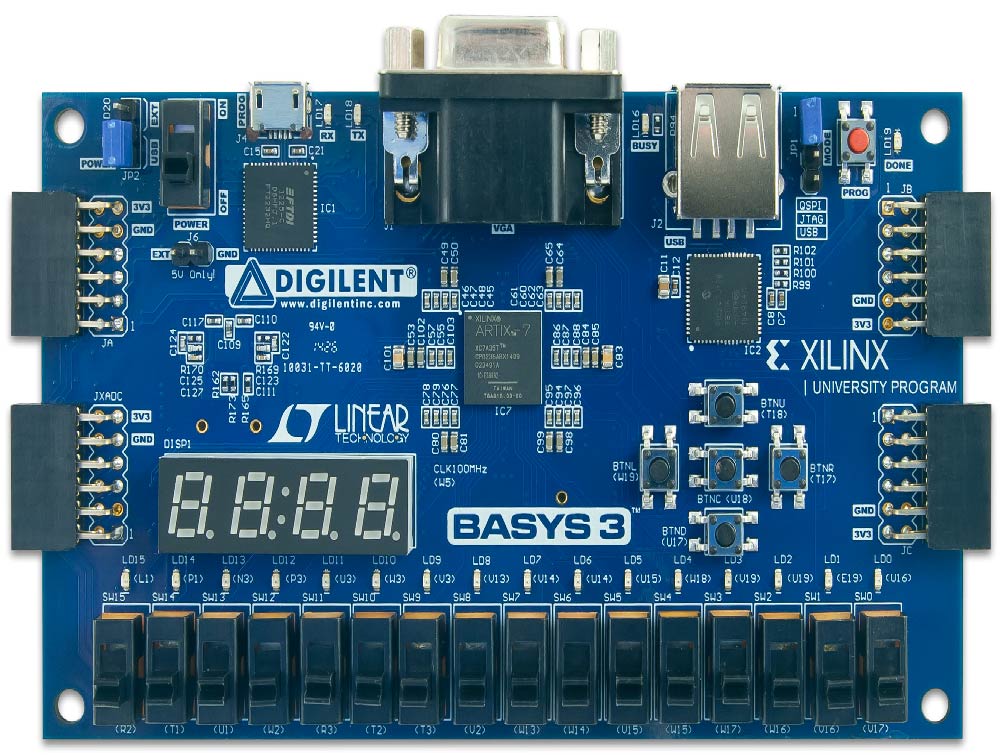
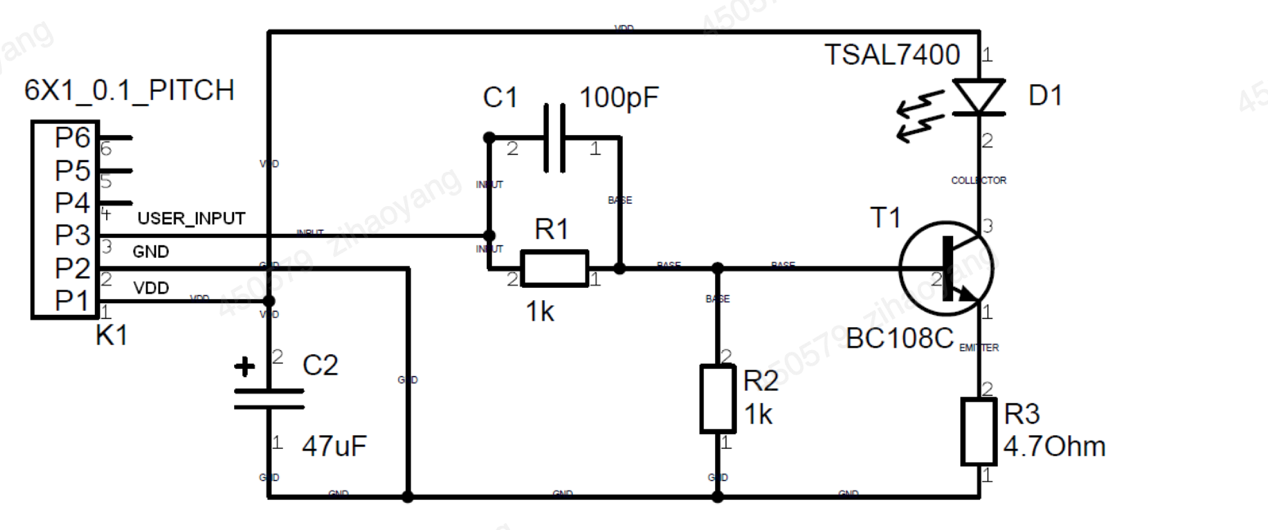


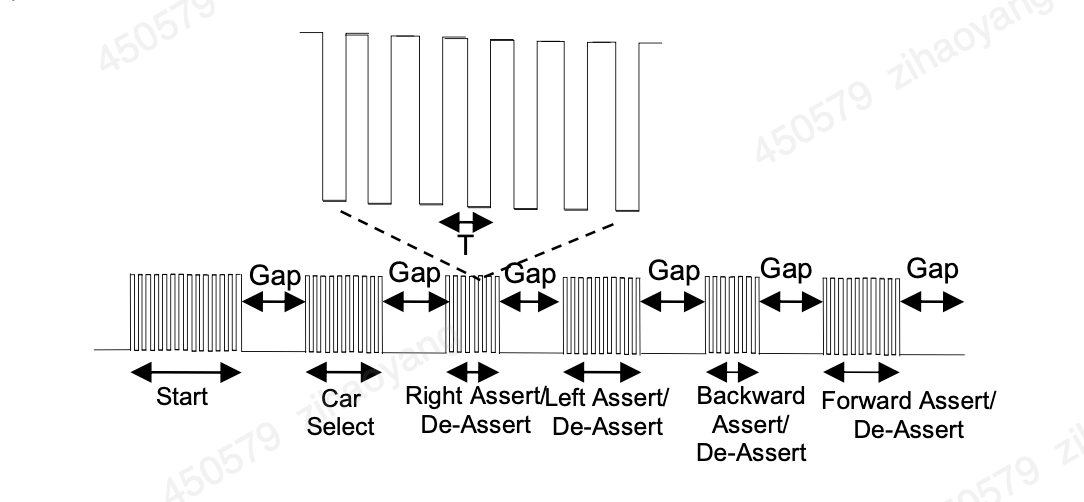
Figure 12: Xilinx Basys3

IR Transmitter module is to generate the proper pulse codes to control the cars remotely using the Artix-7 FPGA chip with the above IR Transmitter peripheral module. The following will present the pulse codes needed to control car. In general, however, the pulse code for any car consists of the following generic packet, generated at 10Hz frequency.

****

**Figure 12: IR transmitter circuit**

The following describes the packet details for each car type. These might vary from one car batch to another, so it is always wise to check the packet details using a logic analyser.

****

**Figure 13: Car Pulse Code**

Yellow-coded Cars:  
In the case of yellow-coded cars, the pulse frequency F = 1/T (see figure above) is equal to 40KHz. The following gives the number of pulses in each packet region as labelled in Figure 8:

* + Start = 88
  + Gap=40
  + CarSelect = 22
  + Right: If the right direction is to be asserted i.e. the car is to move to the right, the number of pulses here is set to 44, otherwise it is 22
  + Left: If the left direction is to be asserted i.e. the car is to move to the left, the number of pulses here is set to 44, otherwise it is 22
  + Backward: If the Backward direction is to be asserted i.e. the car is to move backwards, the number of pulses here is set to 44, otherwise it is 22
  + Forward: If the Forward direction is to be asserted i.e. the car is to move forwards, the number of pulses here is set to 44, otherwise it is 22

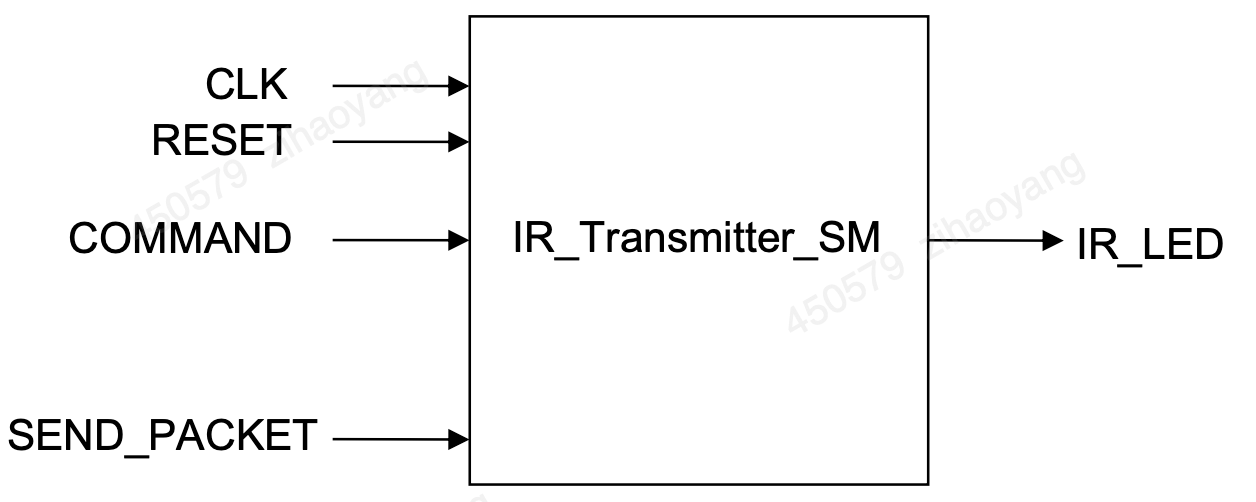
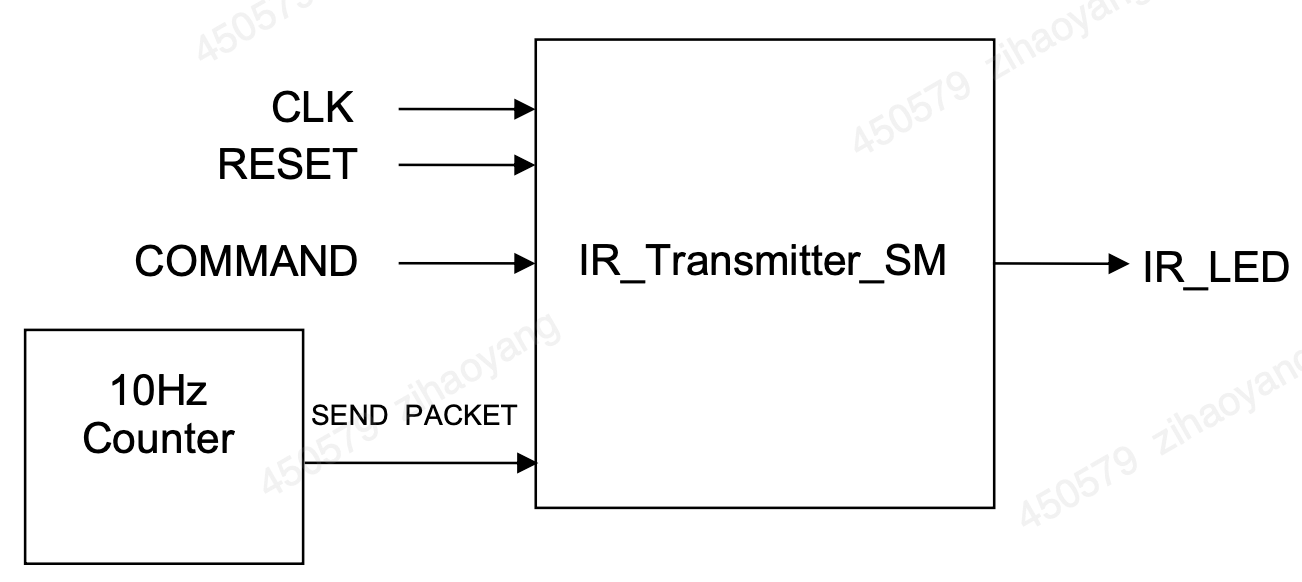


Figure 14: IR Transmitter State Machine

The state machine is illustrated in Figure 14 above where COMMAND consists of four bits: bit 0 to assert or de-assert the right direction, bit 1 to assert or de-assert the left direction, bit 2 to assert or de-assert the backward direction, and bit 3 to assert or de-assert the forward direction. Note that the state machine can be enabled by setting SEND\_PACKET to ‘1’ for one clock cycle, which will result in the generation of one single packet of those shown Figure 13. This has to be done ten times per second for the car to respond to the command in question. A 10Hz counter is thus needed as illustrated in Figure 15 below.

****

**Figure 15: IR transmitter Interface**

The Xilinx Basys 3 board frequency is 100MHz. Therefore, 10Hz Counter module is to generate 10Hz clock signal from Xilinx Basys 3 board 100Mhz frequency. So TenHz\_cnt module is a frequency divider is a module that reduces the frequency of a signal. There are different methods to generate different duty cycles’ reduced frequency clock. Duty cycle is defined as the percentage of one period in which a signal is active (high) compared to its inactive (low) period. It's a measure of how long the signal stays in the active state during a single cycle and is crucial for a variety of applications in FPGA design. Here's a detailed look at the importance of duty cycle in FPGA:

### 1. Clock Signal Characteristics

#### **Clock Management**

* **Precise Timing Signals:** Proper duty cycle is crucial for generating precise clock signals that maintain a consistent and predictable timing reference. A 50% duty cycle (where the signal is high and low for equal periods) is often ideal for clock signals because it provides equal setup and hold times for timing-sensitive operations.
* **Jitter Reduction:** Maintaining an accurate duty cycle helps reduce clock jitter, which is variations in the clock signal's timing. Jitter can lead to timing errors in synchronous operations, so a stable duty cycle is important for reliable performance.

### 2. Data Integrity and Signal Quality

#### **Data Transmission**

* **Transmission Reliability:** For data transmitted in serial protocols (like UART, SPI), maintaining a consistent duty cycle ensures that bits are correctly distinguished as high or low. Any deviation in the duty cycle might lead to incorrect data interpretation.
* **Noise Immunity:** A uniform duty cycle improves the signal-to-noise ratio, which is particularly important in high-speed data transfer applications. This can help reduce errors caused by electromagnetic interference (EMI).

### 3. Pulse Width Modulation (PWM)

#### **Control and Signal Modulation**

* **PWM Control:** Duty cycle is a key parameter in Pulse Width Modulation (PWM) used for controlling power to devices like motors, LEDs, and other actuators. The effective power delivered to the device is directly proportional to the duty cycle.
  + **Motor Speed Control:** In motor control, adjusting the duty cycle of the PWM signal can vary the motor speed.
  + **Light Intensity Control:** For LEDs, the duty cycle can control brightness levels.
* **Signal Generation:** FPGAs frequently use PWM signals to generate analog signals. By adjusting the duty cycle, a wide range of analog outputs can be synthesized.

### 4. Power Management

#### **Efficient Power Utilization**

* **Dynamic Power Control:** Adjusting the duty cycle of control signals can help manage power consumption efficiently. Reducing the duty cycle reduces average power usage, which is fundamental in battery-powered and energy-sensitive applications.
* **Thermal Management:** Proper duty cycle management can lead to reduced heat generation, thereby addressing thermal constraints in high-performance FPGA systems.

### 5. Synchronization and Timing

#### **Multiphase Clock Generation**

* **Phase Alignment:** In applications where multiple clock phases are required, such as in data converters or certain DSP architectures, maintaining a consistent duty cycle across phases ensures proper synchronization.
* **Timing Closure:** Consistent duty cycles are essential for timing closure in complex FPGA designs, ensuring that all parts of a circuit meet their timing requirements.

### 6. Interfacing with External Components

#### **Compatibility with Peripherals**

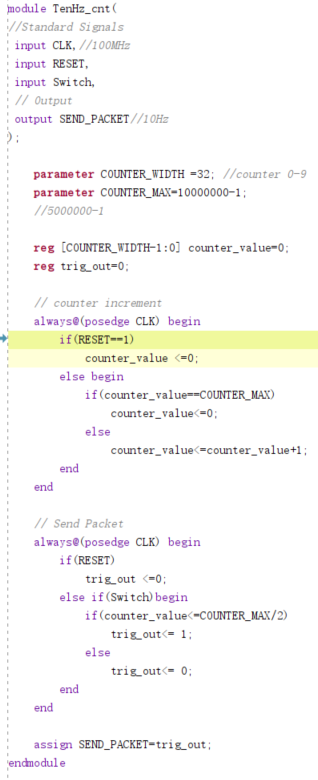
* **External Device Communication:** Many external devices have specific duty cycle requirements for their input signals. Ensuring the FPGA-generated signals meet these requirements is crucial for proper interfacing.
* **Sensor Readings:** Some sensors and ADCs rely on specific duty cycles for accurate data sampling and clocking.

### 7. Debugging and Diagnostics

#### **Signal Analysis**

* **Waveform Integrity:** When debugging FPGA designs, analyzing the duty cycle of various signals can provide insights into performance issues or timing errors.
* **Test Patterns:** Controlled duty cycles can be used in test patterns to validate the correct operation of peripherals and interfaces [8].

Here is the specific Verilog code to realize:

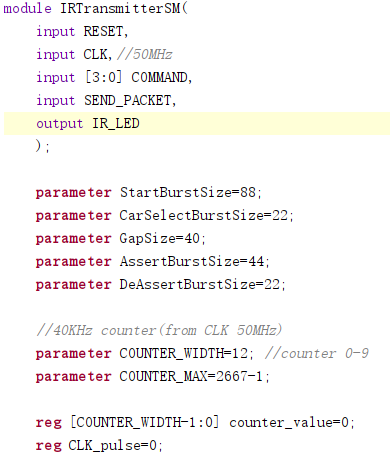


Because I set out to 1 only when the counter\_value is less than half of COUNTER\_MAX, otherwise it is 0, this ensures a duty cycle of 50%

Here is the simulation outcome:



It is very obvious that Clk\_out\_10Hz is divided by 10000000 times from 100MHz clk.

****

Here is the definition of parameters of car operation stages.

**2.3 VGA Display**

In order to display video onto a monitor screen, we will make use of the VGA port available on the BASYS 3 board. The latter uses 14 signals (carried by 14 FPGA pins, see Figure on page 11 of the BASYS 3 Reference Manual) with 12-bit colour and two synchronisation signals (HS – Horizontal Sync, and VS – Vertical Sync). HS and VS allow the monitor to align the incoming colour signals such that individual pixels on the screen are coloured correctly. Colours are derived by a combination of the three primary colours: red, blue and green. Four bits are used for each colour (given 16 possible levels of each colour).

However, it is possible to use only 8 of the 12-bit colours, especially for compatibility with an 8 bit bus processor architecture. In this configuration, three bits are used for Red and Green (giving 8 possible levels of each colour) and two bits are used for Blue (giving four possible levels only, as the human eye is less sensitive to blue than it is to red and green). You could either chose to implement an 8-bit colour (connecting only the 3 LSBs of Red and Green, and 2 LSBs of Blue), or implement the full 12 bits colour but you will have to send the colour information twice on an 8-bit bus. In this description, we target the 8-bit configuration.

In this paper, I develop an FPGA-based VGA driver which generates the necessary VGA signals (8 bit colour information and 2 synchronisation signals) to a VGA monitor using a frame buffer that holds frame pixel values. The following will demostrate suggested design of the VGA driver.

VGA provides a resolution of 640 x 480 pixels (horizontal x vertical). A VGA driver could be built from two modules as shown in the following figure:

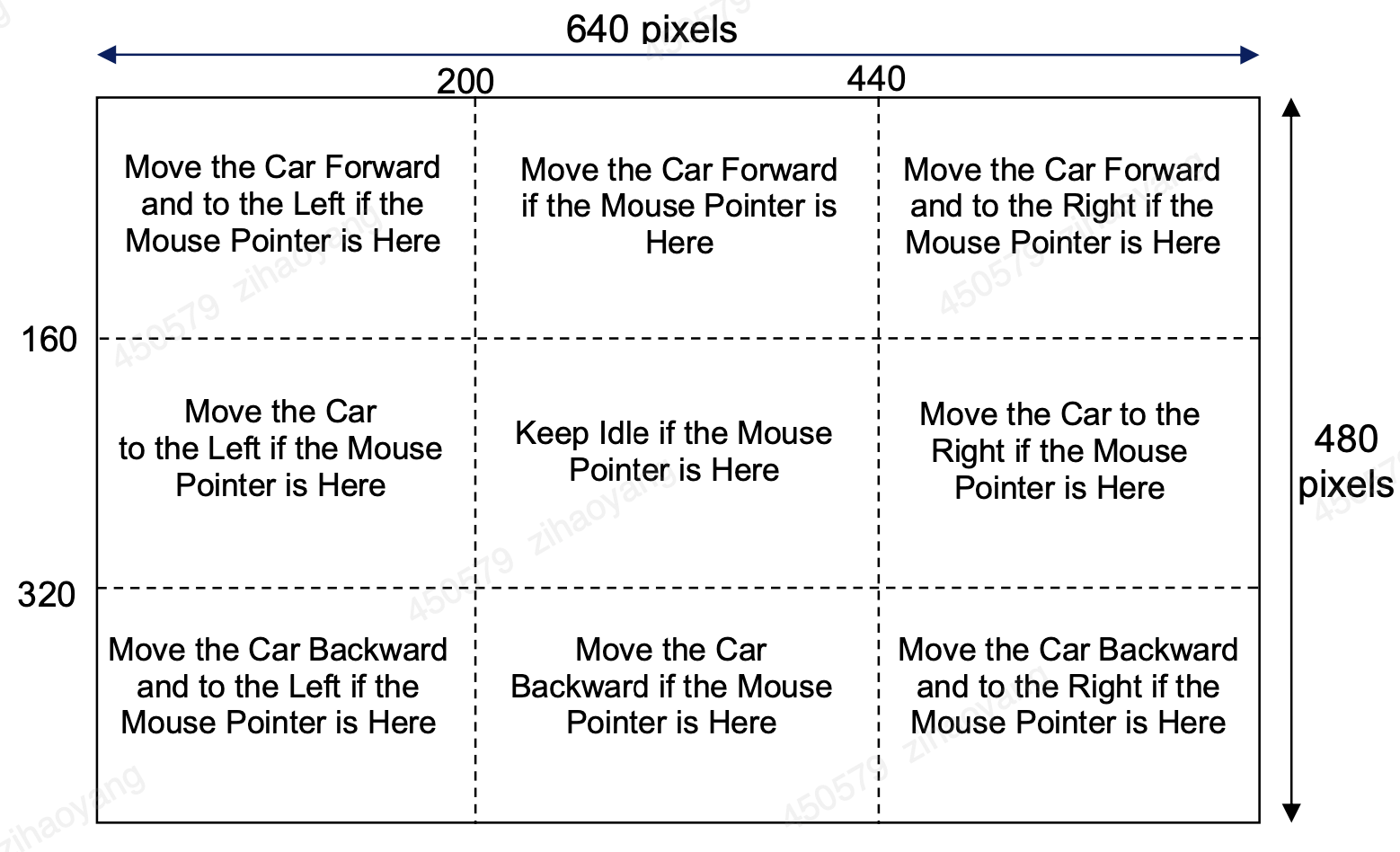


Figure 16: Car movement (command) depending on mouse pointer position

* - “Frame\_Buffer”: A dual ported memory module which can be written to by an outside module e.g. a microprocessor, and read from by the “VGA\_Sig\_Gen” module (see Figure 11 below) for VGA display
* - “VGA\_Sig\_Gen”: VGA signal generator module, which reads consecutive pixel colours in raster pattern, from the “Frame\_Buffer” module, and outputs the VGA signals through the proper FPGA pins

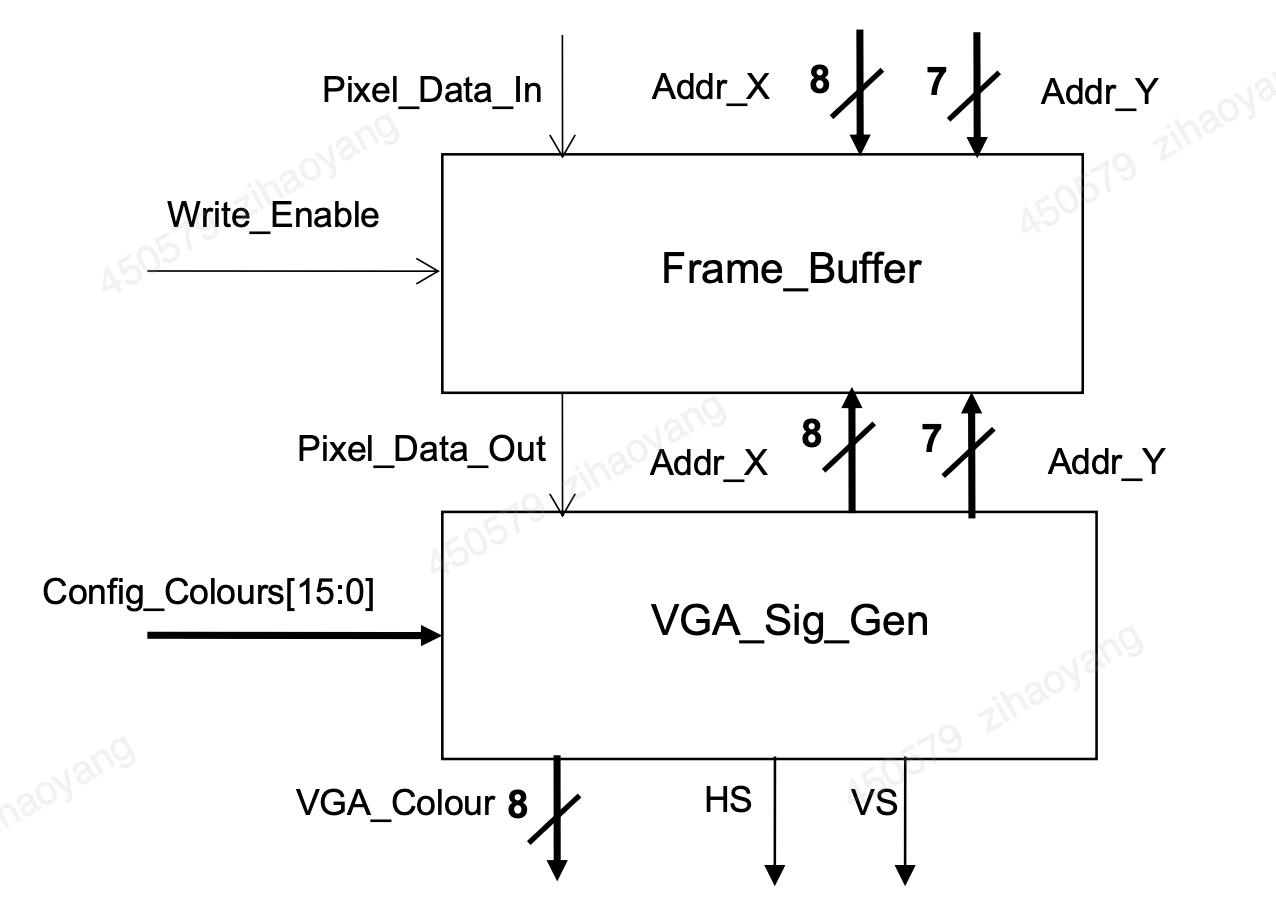
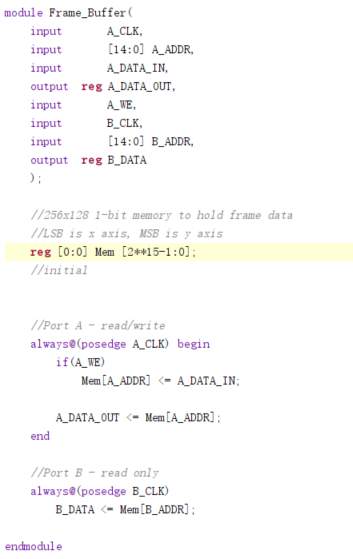
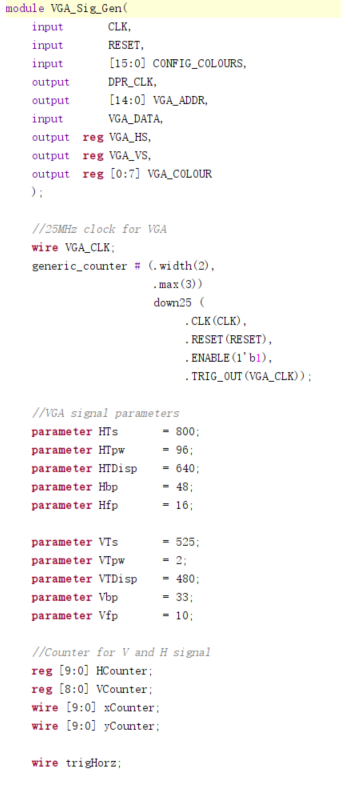


Figure 17: VGA Interface

Unfortunately, the FPGA chip on the BASYS 3 board does not have enough memory resources to store a full VGA frame with 8-bit pixels (i.e. 640 x 480 x 8 bits memory). In fact, few FPGA chips on the market can hold complete frames on-chip. Instead, off-chip memory (e.g. SDRAM) is often used to implement the frame buffer. The BASYS 3 board, however, does not have such external memory resource, which means that FPGA logic must be used to hold frame buffer data. To solve this problem, we can use a reduced colour resolution as well as a reduced pixel resolution to minimise memory storage requirements. In our suggested design, we will reduce the screen resolution by 4 in each direction i.e. each 4x4 pixel region on the screen would be represented by one single data element on the frame buffer, hence the above wordlength of the X and Y addresses in Figure 16 (i.e. 8 and 7 bits respectively instead of 10 and 9 bits for the 640 x 480 VGA resolution). We also reduce the colour resolution from 256 possible colours to just two colours (foreground and background) hence 1 bit data is sufficient for the colour information. A dual ported 256 x 128 1-bit memory should then be large enough to hold the whole frame buffer, which could be easily stored on the Artix-7 chip. Note that in the proposed code below, the foreground and background colours can be set by an external module e.g. a microprocessor, through input Config\_Colour[15:0] (see Figure 16 above) which represents 2 x 8 bits of data for two possible 8-bit colours.

The following shows possible Verilog code for the Frame\_Buffer module:

  
The following shows Verilog code fragments for the VGA\_Sig\_Gen module:



Since the VGA display needs 25MHz, I need to design a quarter frequency in this module called generic\_counter.

Note that the VGA standard was designed for CRT monitors, and hence it is important to be aware of the timing of the original system, even if we are using a digital display. It should also be noted that in addition to the HS and VS timing definitions within the reference manual, the 8-bit colour signal needs to be set to 8'h00 (zero) whilst not during the display time.

During the development of this project, I complete your code based on the above, synthesis and verify your code with simple test benches, generate an FPGA bitstream and test on the BASYS 3 board. Test the above driver block by packaging it within a top-level module that generates frame buffer and colour configuration data, and outputs the VGA signals to the proper FPGA pins. Display the following chequered image on the VGA display, changing colours every one second.

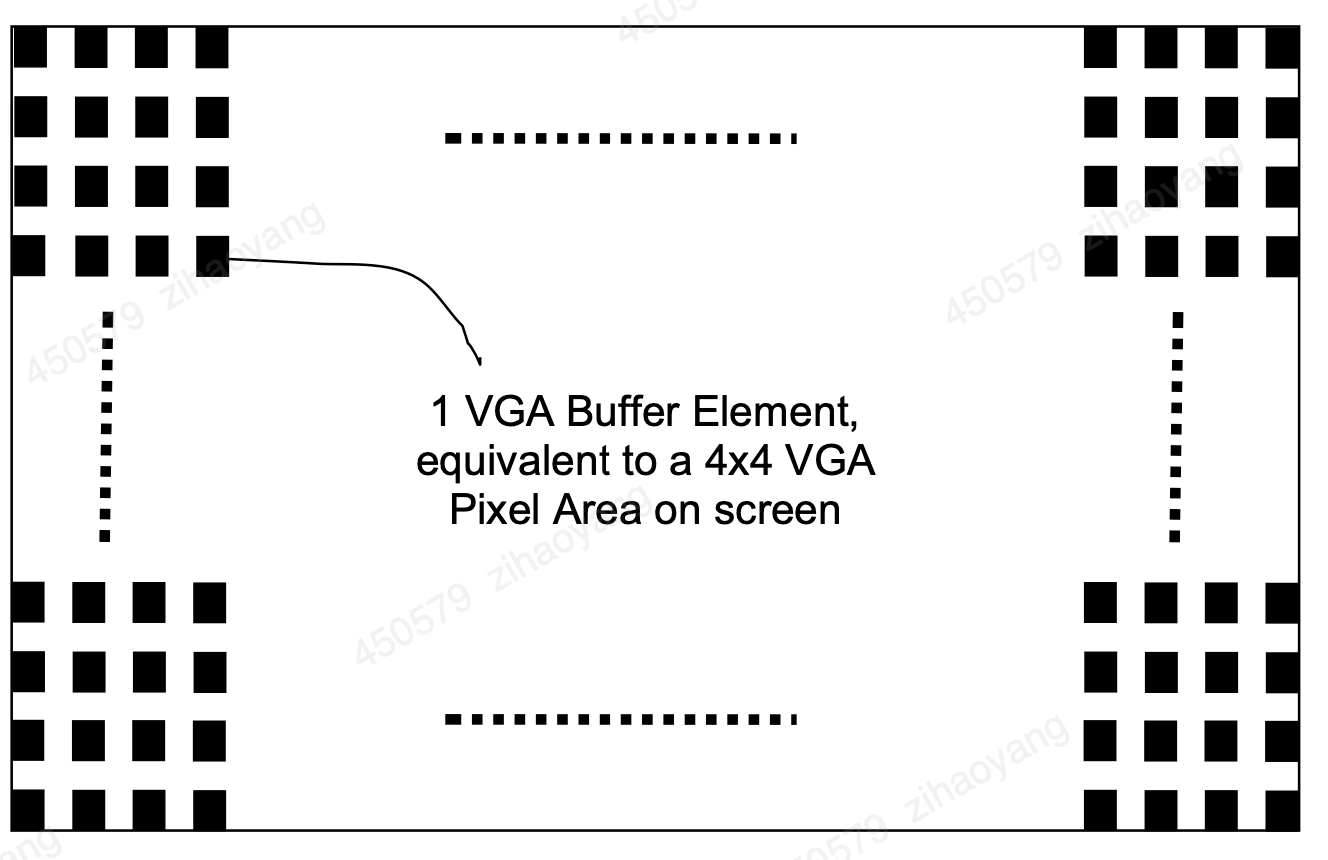


Figure 18: Chequered image to be displayed on the VGA screen

**2.4 PS/2 and Mouse Driver**

The USB connector on the BASYS 3 board can accommodate a USB mouse. Internally, the signals are converted to PS/2-like signals via a microcontroller as discussed on page 7 and 8 of the BASYS 3 reference manual. Hence, all we need do is implement a driver for a PS/2 mouse as the USB connector could be seen as just a wrapper which has already been implemented.

The PIC24 drives several signals into the FPGA – two are used to implement a standard PS/2 interface for communication with a mouse or keyboard (see figure 19 below).

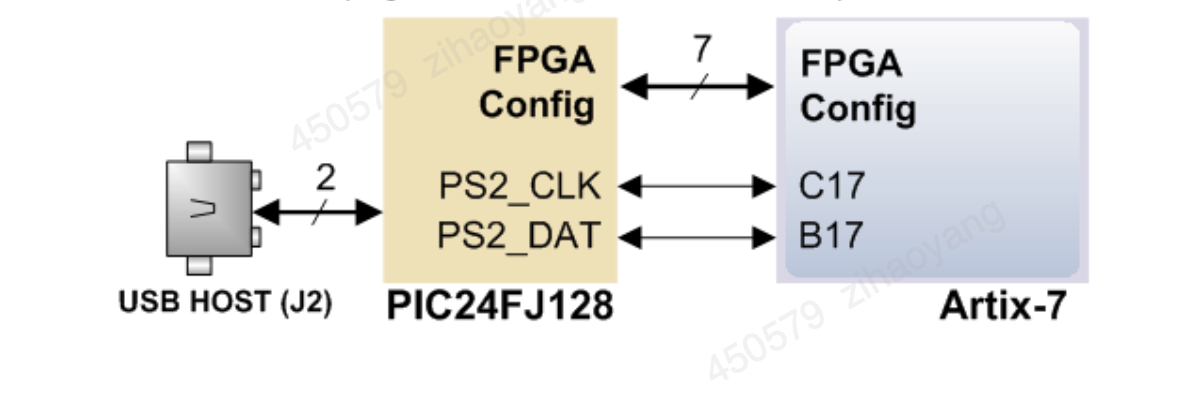


Figure 19: USB Host signals to PS/2 signal conversion

PS/2 devices use a two-wire serial bus (clock and data) to communicate with a host device

Communication is bidirectional and performed in packets of 11-bit words, with each word containing a start, stop and odd parity bit. The following describes the PS/2 mouse protocol in detail (NB. the PS/2 keyboard protocol can be found in the BASYS 3 user manual if you are interested in developing a keyboard interface but this is not required in this lab).

The mouse device only outputs a clock and data signal when it is moved. Otherwise, the clock and data lines remain at logic high (i.e. ‘1’). Open-Collector drivers are usually used to drive the two-wire bus between the mouse and host.

The device can send data to the host only when both data and clock lines are high. Because the host is the bus master, the device must check whether the host is sending data before driving the bus. The clock line is used for this purpose as a “clear to send” signal; if the host pulls the clock line low, the device must not send any data until the clock is released.

Communication is performed in 11-bit words, where each word consists of a ‘0’ start bit, followed by 8 bits of data (LSB first), followed by an odd parity bit (i.e. a bit that is set to ‘1’ if the number of 1’s in the 8 bits of data is even, and ‘0’ otherwise), and terminated with a ‘1’ stop bit. The odd-parity bit is used for error detection.

Data sent from a PS/2 device to a host is read on the falling edge of the clock signal, whereas data sent from a host to a PS/2 device is read on the rising edge of the clock signal. The following figure shows PS/2 signal timing for a device to host communication. Note the timing requirements which must be strictly adhered to. The clock frequency, for instance, must lie between 10 and 16.7 KHz.

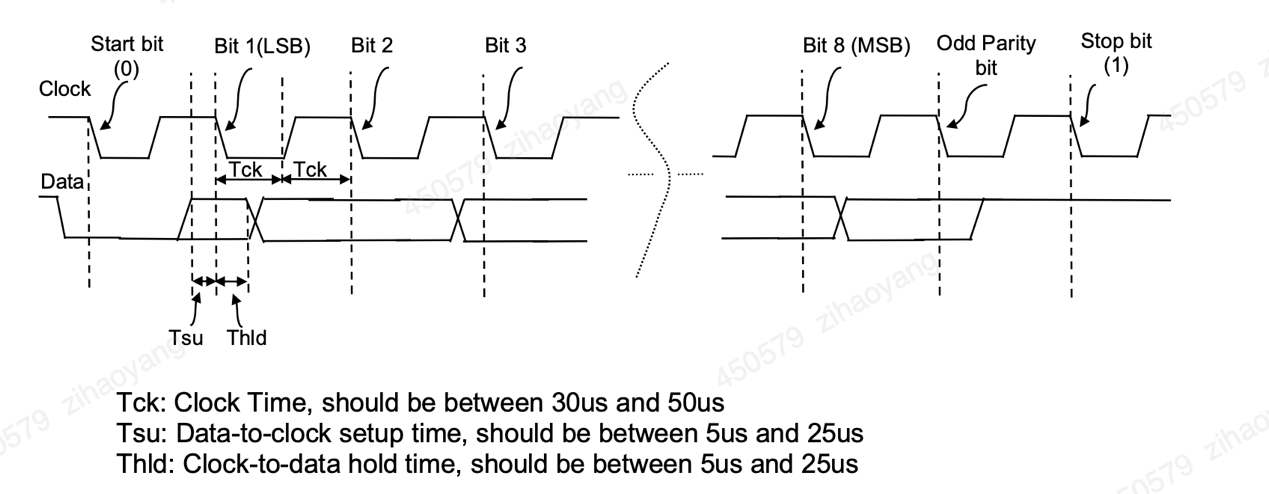


Figure 20: PS/2 Device to Host Signal Timing

The following figure shows PS/2 signal timing for a host to device communication. The host brings the clock line low first, for at least 100μs. It then brings the data line low and releases the clock line. The host then waits for the PS/2 device to bring the clock line low. After that, it sets or resets the data line with the first data bit, and waits for the device to bring clock line high. It then waits for the device to bring the clock line low before it sets/rests the data line with the second data bit. This process is repeated until all eight data bits are sent as well as the odd-parity bit. Next, the host releases the data line, and waits for the device to bring the data line low, and then the clock line low. Finally, the host waits for the device to release data and clock lines.

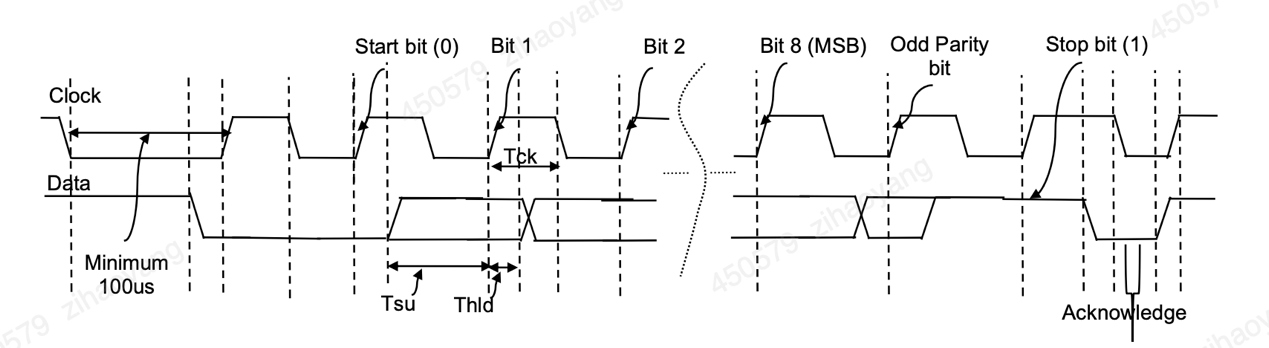


Figure 21: Host to PS/2 Device Signal Timing

Now that we have seen the low level PS/2 protocol, let us look at the high level host-mouse communication. At power-up, a typical host-mouse communication consists of the following steps:

1)  The host sends a Reset Command (consisting of byte “FF”) to the mouse,

2)  The mouse responds with an Acknowledgement byte “FA”,

3)  The mouse then goes through a self-test process and sends “AA” when this is passed. Then a mouse ID byte “00” is sent to the host, after which the host knows that the mouse is functioning well and ready to transmit data,

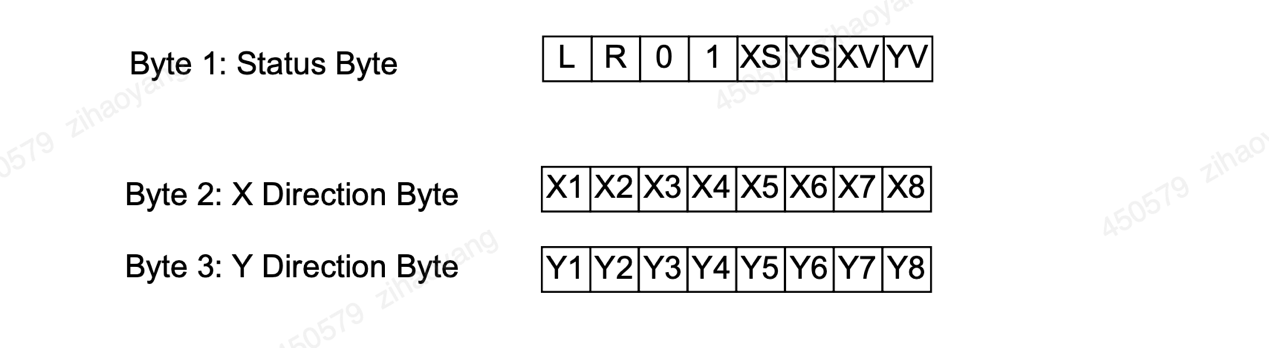
4)  The host sends byte “F4” to instruct the mouse to “Start Transmitting” its position information,

5)  The mouse acknowledges the “Start Transmitting” command by sending byte “FA” back to the host,

6)  After this, the mouse starts transmitting its position information in the form of 3 bytes at a sample rate that can be set by the host (the default is 100Hz)

however that in the Basys3 FPGA board, probably due to the USB to PS/2 conversion, F4 instead of FA is returned, and parity test fails. Hence in state 8 of MasterStateSM module, the acknowledgement code have been changed to F4, and parity check skipped.

Thus, each data transmission from the mouse to the host after initialisation consists of 33 bits, where bits 1 (first bit), 12, and 23 are ‘0’ start bits; bits 10, 21, and 32 are Odd-Parity bits; and bits 11, 22, and 33 are ‘1’ stop bits. The three-byte data fields contain status and movement data as shown in the figure below.



The mouse reports a relative coordinate system whereby a move to the right generates a positive number in the X Direction Byte field, and a move to the left generates a negative number in this field. Similarly, a move upwards generates a positive number in the Y Direction Byte field, and a move downwards generates a negative number. Note that the X and Y Direction Bytes represent the magnitude of the rate of mouse movement, the larger the number the faster the mouse is moving. Bits XS and YS in the Status Byte are the sign bits, whereby a ‘1’ indicates a negative number, whereas XV and YV bits are movement overflow indicators, whereby a ‘1’ means overflow has occurred. The L and R fields in the Status Byte indicate that the left and right button have been pressed, respectively (‘1’ indicates the button has been pressed).

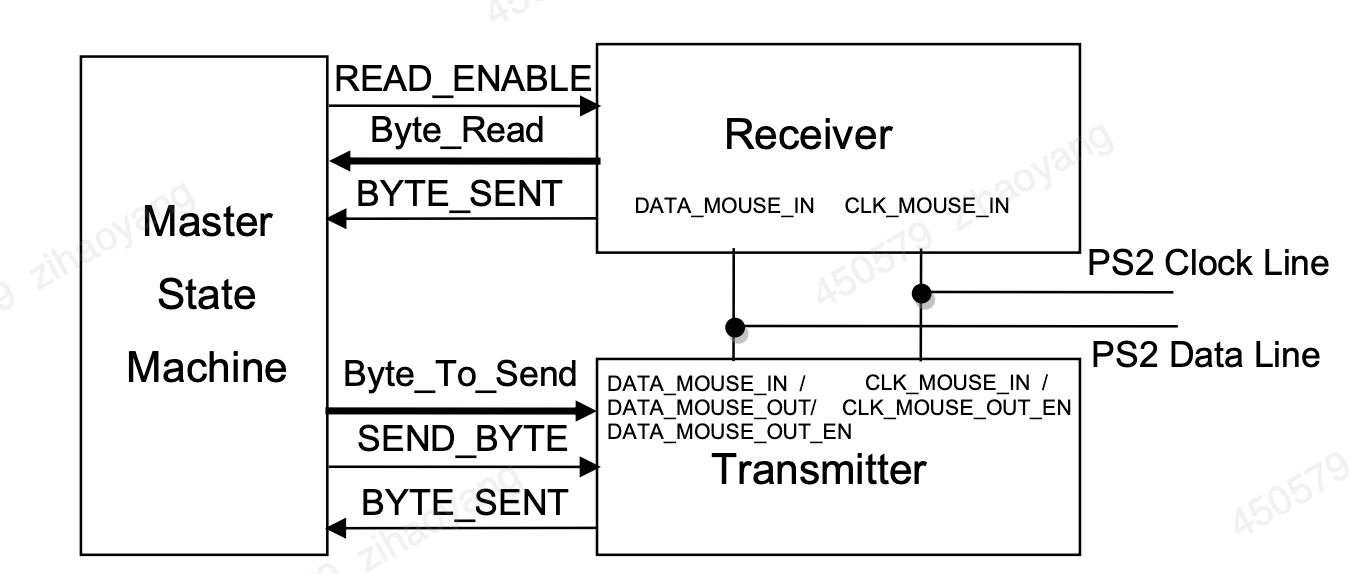


Figure 22: Mouse Interface: Simplified Block Diagram

**2.5 7-segments LED**

The 7-segment display, also written as “seven segment display”, consists of seven LEDs (hence its name) arranged in a rectangular fashion as shown. Each of the seven LEDs is called a segment because when illuminated the segment forms part of a numerical digit (both Decimal and Hex) to be displayed.

An additional 8th LED is sometimes used within the same package thus allowing the indication of a decimal point, (DP) when two or more 7-segment displays are connected together to display numbers greater than ten.

Each one of the seven LEDs in the display is given a positional segment with one of its connection pins being brought straight out of the rectangular plastic package. These individually LED pins are labelled from a through to g representing each individual LED. The other LED pins are connected together and wired to form a common pin.

So by forward biasing the appropriate pins of the LED segments in a particular order, some segments will be light and others will be dark allowing the desired character pattern of the number to be generated on the display. This then allows us to display each of the ten decimal digits 0 through to 9 on the same 7-segment display.

The displays common pin is generally used to identify which type of 7-segment display it is. As each LED has two connecting pins, one called the “Anode” and the other called the “Cathode”, there are therefore two types of LED 7-segment display called: **Common Cathode** (CC) and **Common Anode** (CA).

The difference between the two displays, as their name suggests, is that the common cathode has all the cathodes of the 7-segments connected directly together and the common anode has all the anodes of the 7-segments connected together and is illuminated as follows

Seven anodes of the seven segments in a single LED are connected together to one common anode node while its cathodes are separate as shown in the following figure. DP-segment is to illuminate the dot so we omit the DP-segment for now since it is not contributing to the number value of the seven-segment display.

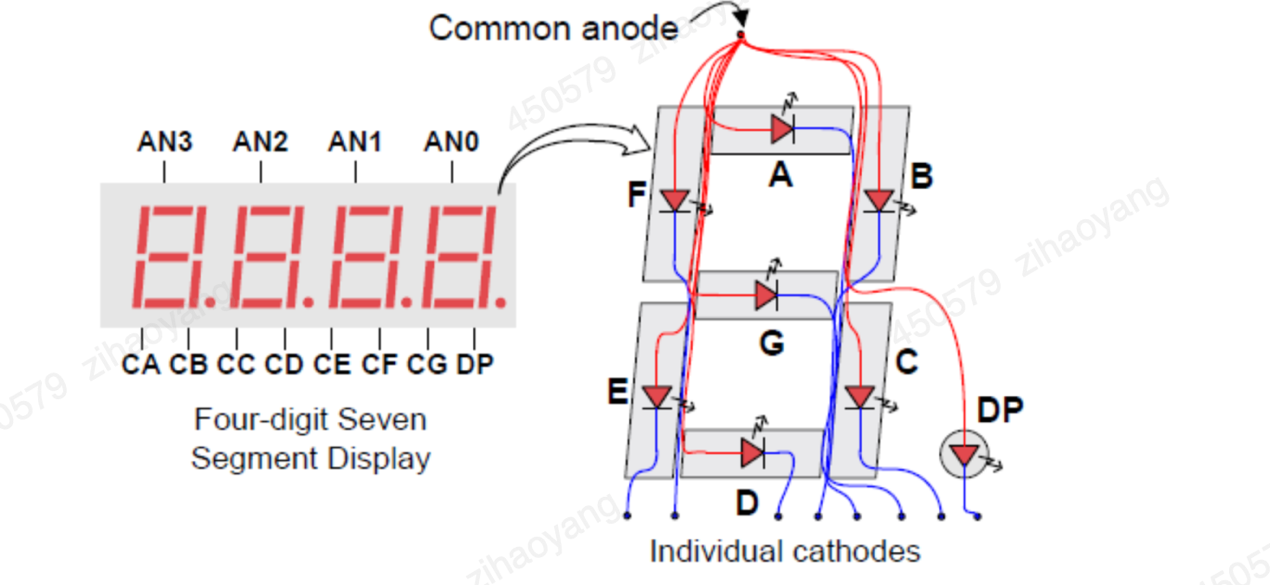


Figure 23: 7-segment LED structure

7-segment Display Truth Table

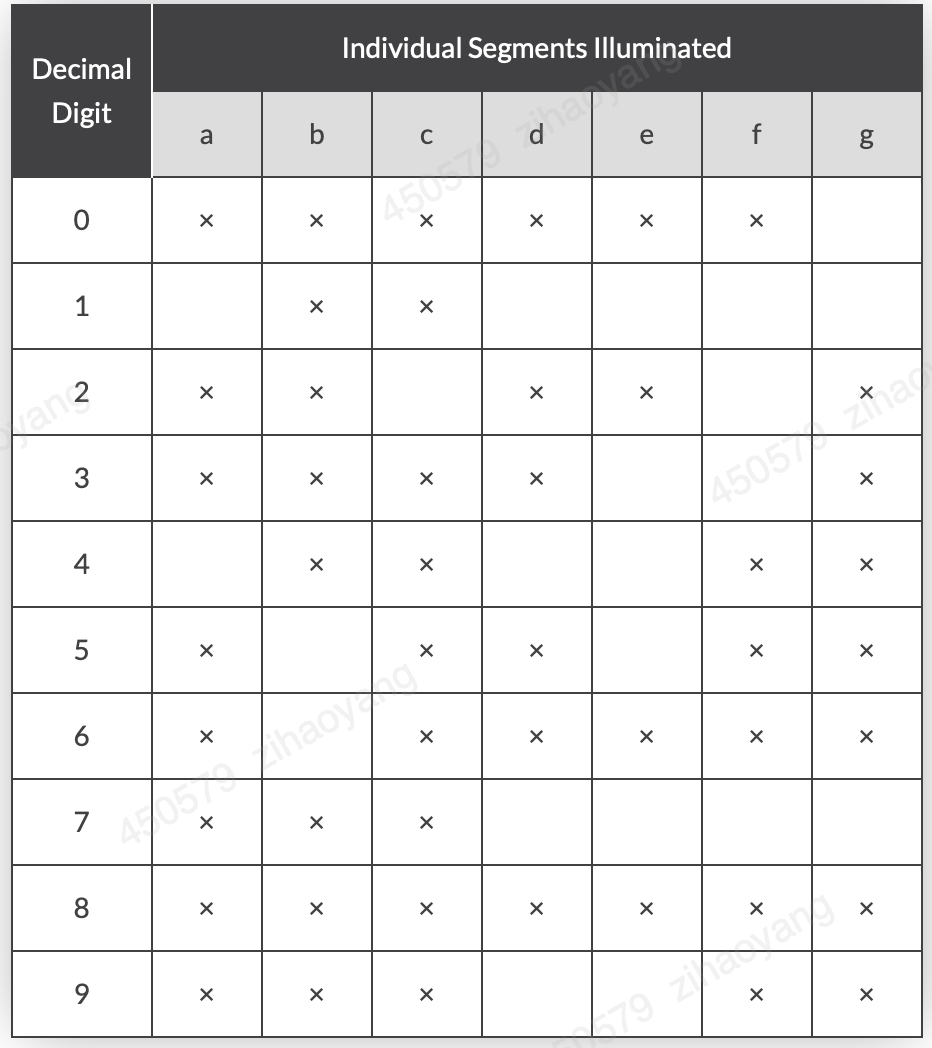


Table 4

Although a 7-segment display can be thought of as a single display, it is still seven individual LEDs within a single package and as such these LEDs need protection from over current. LEDs produce light only when it is forward biased with the amount of light emitted being proportional to the forward current.

This means then that an LEDs light intensity increases in an approximately linear manner with an increasing current. So this forward current must be controlled and limited to a safe value by an external resistor to prevent damage to the LED segments.

The forward voltage drop across a red LED segment is very low at about 2-to-2.2 volts, (blue and white LEDs can be as high as 3.6 volts) so to illuminate correctly, the LED segments should be connected to a voltage source in excess of this forward voltage value with a series resistance used to limit the forward current to a desirable value.

Typically for a standard red coloured 7-segment display, each LED segment can draw about 15 mA to illuminated correctly, so on a 5 volt digital logic circuit, the value of the current limiting resistor would be about 200Ω (5v – 2v)/15mA, or 220Ω to the nearest higher preferred value.

So to understand how the segments of the display are connected to a 220Ω current limiting resistor consider the circuit below.

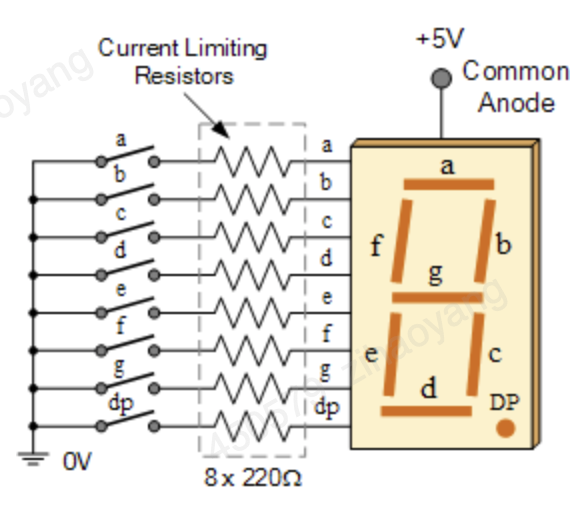
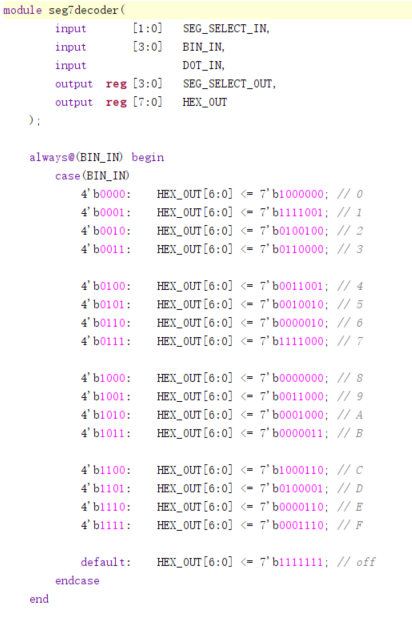


Figure 24: 7-segment circuit

In this example, the segments of a common anode display are illuminated using the switches. If switch a is closed, current will flow through the “a” segment of the LED to the current limiting resistor connected to pin a and to 0 volts, making the circuit. Then only segment a will be illuminated. So a LOW condition (switch to ground) is required to activate the LED segments on this common anode display.

But suppose we want the decimal number “4” to illuminate on the display. Then switches b, c, f and g would be closed to light the corresponding LED segments. Likewise for a decimal number “7”, switches a, b, c would be closed. But illuminating 7-segment displays using individual switches is not very practical.

Here is the 7-segment decoder code:



**2.6 Interrupt**

As we know, a computer CPU is a deeply orderly entity, following the instructions of the program one by one and doing what it is told in a precise and predictable fashion. An interrupt disturbs this order. Coming maybe when least expected, its function is to alert the CPU in no uncertain terms that some significant external event has happened, to stop it from what it is doing and force it (at the greatest speed possible) to respond to what has happened. Originally interrupts were applied to allow emergency external events, such as power failure, the system overheating or major failure of a subsystem to get the attention of the CPU. But the concept of interrupts was recognized as being very powerful. As time went on, more and more subsystems gained the power to generate interrupts. This forced increasing complexity in interrupt structures and a need to recognize that not all interrupts were equal.

To work successfully with interrupts, we need to understand both the hardware interrupt structure and the programming techniques needed to program successfully with them. An introduction to these now follows.

Different microcontrollers have rather different interrupt structures. Inevitably they have more than one interrupt source, usually with some internally generated and others external. A generic structure, which illustrates the main hardware principles, is shown in Figure 6.1. On the left we see one of several sources, ‘Interrupt X’. If an interrupt occurs, it sets an S–R bistable. The occurrence of the interrupt, even if it is only momentary, is thus recorded. The output of the bistable, the latched version of the interrupt, is called the ‘interrupt flag’. This is then gated with an enable signal, ‘Interrupt X Enable’. If this is high, then the interrupt signal progresses to an OR gate. If it is low, the interrupt signal gets no further. If enabled, it is ORed with other enabled interrupt inputs of the microcontrollers. The OR gate output will go high if any interrupt input is high. There is then a further gating of the OR gate output, this time with a ‘Global Interrupt Enable’. Only if that value is high can any interrupt signal reach the CPU. When the CPU has responded to an interrupt, it is necessary to clear the interrupt flag. In some processors this is done automatically by the CPU, in others it must be done within the program [9].

In our system architecture presented in Figure 8 above, interrupts come from two possible sources: 1) the mouse, when it is moved or clicked, and 2) the timer, which outputs an interrupt signal every one second. In general, an interrupt is an asynchronous signal from hardware indicating the need for attention, or a synchronous event in software indicating the need for a change in execution. Multiple hardware interrupts can be serviced through an interrupt controller which acts as another bus peripheral whose task is to service interrupts before they are sent to the microprocessor. This includes dealing with priorities for instance. In this lab, interrupts will be served on a “first-come first-served” basis. If two interrupts arrive at exactly the same time, the mouse interrupt will be dealt with first.

When an interrupt request is received by the microprocessor, the current program execution is suspended after the execution of the current instruction. The context information is then saved so that execution can return to the current program after interrupt servicing. In the case of our microprocessor, the context consists in the address of the next instruction to be executed by the interrupted process. After context saving, execution is transferred to an interrupt handler to service the interrupt. The start (or base) address of the interrupt handler’s service routine is usually predefined, for each interrupt line, in a particular memory address. In our case, the memory address where the base address of the mouse interrupt handler’s service routine is stored is “FF”, whereas that of the timer is “FE”. This means that whenever a mouse interrupt is received, for instance, the microprocessor fetches the base address of the mouse interrupt handler’s service routine to be executed from address “FF”. In other words, the content of address “FF” will be the address of the first instruction to be executed by the interrupt handler’s service routine.

The following gives you a possible Verilog code to implement the Timer peripheral. Code for the seven-segment display peripheral has already been given to you as part of the mouse interface module (you would need to wrap your existing peripheral codes with bus interfaces though), while the LEDs peripheral is straightforward to implement.

**2.7 Rom**

ROM (Read-Only Memory) holds significant importance in RISC-V based microprocessor designs. Given the flexibility, openness, and scalability of the RISC-V architecture, ROM plays a crucial role in multiple aspects of the microprocessor's operation [10]. Here’s a detailed outline of the importance of ROM in a RISC-V based microprocessor:

### 1. Bootstrapping the Processor

#### **Bootloader Storage**

* **System Initialization:** In a RISC-V microprocessor, the ROM typically stores the bootloader, which is the initial program that executes when the system powers up. This bootloader is responsible for initializing hardware components and preparing the system for operation.
* **Secure Boot:** ROM can contain a secure bootloader that verifies the integrity of the firmware before it is loaded, ensuring that the system boots securely.

### 2. Storing Firmware and Software

#### **Embedded Firmware**

* **Firmware Storage:** ROM is used to store the firmware required for the microprocessor’s operation. This includes basic input/output system (BIOS), operating systems, and system-level utilities.
* **Immutable Storage:** Since ROM is immutable, it ensures that the fundamental software required to operate the microprocessor remains unchanged and reliable across reboots.

### 3. Efficient Storage of Constants and Lookup Tables

#### **Fixed Data Storage**

* **Lookup Tables:** ROM efficiently stores fixed data such as lookup tables, mathematical constants, and predefined data arrays. These are frequently used in RISC-V implementations for tasks such as floating-point operations and complex mathematical computations.
* **Microcode Storage:** For certain RISC-V implementations that use microcoded instructions, ROM can store the microcode that defines the behavior of complex instructions.

### 4. Performance Optimization

#### **Speed and Efficiency**

* **Fast Access Times:** ROM provides fast access to stored data and instructions, which is crucial for the performance of the microprocessor. The speed at which the RISC-V processor can fetch instructions from ROM directly impacts its overall efficiency.
* **Instruction Fetching:** In the initial stages of booting, the processor fetches instructions directly from ROM, ensuring quick and reliable execution of critical startup routines.

### 5. Supporting RISC-V’s Modularity and Extensibility

#### **Custom Extensions**

* **Custom Instructions:** RISC-V’s open and extensible nature allows for the addition of custom instructions. ROM can store the microcode or control logic associated with these custom instructions, enabling quick access and execution.
* **Debug and Diagnostics:** ROM can store diagnostic routines and debug support code, facilitating easy debugging and validation during development and post-deployment support.

### 6. Interfacing and Integration

#### **Peripheral Initialization**

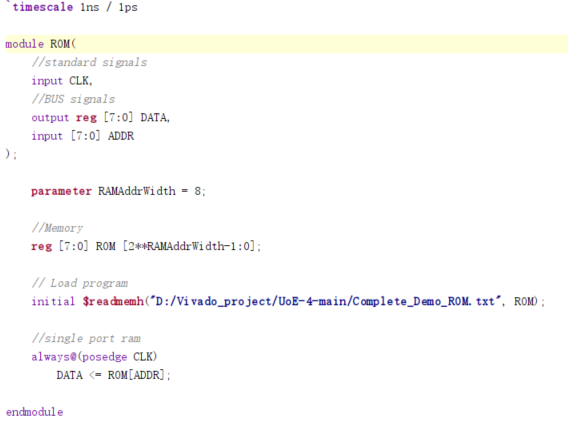
* **Peripheral Drivers:** ROM can store drivers and initialization code for various peripherals and hardware components attached to the microprocessor. Ensuring these drivers reside in ROM guarantees their availability and correct operation at system start.
* **Configuration Data:** ROM can include configuration parameters for various components, making it easier to achieve consistent and reliable system startup configurations.

### 7. Power Management

#### **Efficient Operation**

* **Embedded ROM in Low-Power Modes:** In power-sensitive applications, data and code stored in ROM can be accessed with less power compared to writeable memory types. This helps in maintaining efficient operation during low-power modes.
* **Power-On Initialization:** During power-on, the microprocessor can quickly initialize and transition to a working state by executing the initialization code stored in ROM.

Here is the Verilog code of how ROM could be defined and used in a RISC-V based microprocessor:



In conclusion, ROM is a vital component in microprocessor design within FPGAs, providing essential functions such as boot and initialization storage, firmware and software storage, security, performance optimization, and efficient integration with FPGA logic. Its immutability and reliability make it indispensable for secure and consistent system behavior, while its ability to store crucial code and data contributes significantly to the overall design and operational efficiency of FPGA-based microprocessors.

**2.8 RAM**

Random Access Memory (RAM) is a fundamental component in the design of RISC-V based microprocessors. The importance of RAM in such designs can be understood from multiple perspectives, including system performance, functionality, flexibility, and development process. Below are the details outlining the importance of RAM in RISC-V microprocessor design:

### 1. **Execution of Programs**

#### **Instruction Fetching and Execution**

* **Program Storage**: RAM stores the executable code of applications and the operating system. By holding active instructions, it enables the microprocessor to fetch, decode, and execute instructions at high speed.
* **Instruction Cache (ICache)**: RAM is often used to implement instruction caches, enhancing the efficiency and performance by reducing the latency associated with fetching instructions from slower storage like ROM or external memory.

#### **Data Storage and Access**

* **Data Cache (DCache)**: RAM is also used for data caches, allowing rapid access to frequently used data, which is crucial for high-performance computing and real-time applications.
* **Temporary Data Storage**: RAM stores temporary data and variables used during program execution, facilitating quick read/write operations necessary for efficient computation.

### 2. **System Performance**

#### **Latency and Bandwidth**

* **High-speed Access**: RAM provides low-latency, high-bandwidth access to data. This is crucial for maintaining the performance of the microprocessor as it ensures data and instructions are readily available without significant delays.
* **Parallelism**: Modern RAM designs support multiple parallel accesses, allowing the RISC-V processor to perform multiple reads/writes simultaneously, further boosting performance.

### 3. **Flexibility and Scalability**

#### **Dynamic Operation**

* **Dynamic Memory Allocation**: RAM enables dynamic memory allocation, allowing programs to request and release memory as needed during execution. This is essential for the efficient management of memory resources, particularly in complex applications and operating systems.
* **Multi-Tasking Support**: In multi-tasking environments, RAM provides the necessary support for context switching by storing the state of each task or process. This allows the RISC-V microprocessor to efficiently manage and switch between multiple tasks.

#### **Expandable Memory**

* **Scalability**: Systems can be designed with expandable RAM, allowing for increased memory capacity as needed. This is particularly important in applications that require handling large datasets or running memory-intensive applications.
* **Modular Design**: RISC-V systems can be designed with different RAM modules for different purposes, such as separate instruction and data caches or additional RAM for specific peripherals, providing modularity and flexibility in design.

### 4. **Development and Prototyping**

#### **Testing and Debugging**

* **Real-time Prototyping**: RAM allows for real-time testing and debugging of applications and firmware. Developers can load and modify code in RAM during debugging sessions to quickly test changes and iterate on the design.
* **Simulation Support**: During the design and simulation of a RISC-V microprocessor, RAM provides a realistic environment for testing programs and verifying processor behavior under different scenarios.

### 5. **System Memory Hierarchy**

#### **Memory Management**

* **Hierarchical Memory**: RAM is a crucial part of the memory hierarchy, often positioned between the high-speed cache and slower storage like disk drives. This hierarchical approach ensures efficient data handling and access speeds appropriate to different levels of data usage.
* **Memory Maps**: RAM is used to create memory maps, allowing the microprocessor to access different memory regions efficiently. This includes mapping peripheral devices, stack, heap, and other memory areas required for structured data management.

### 6. **User Experience and Application Performance**

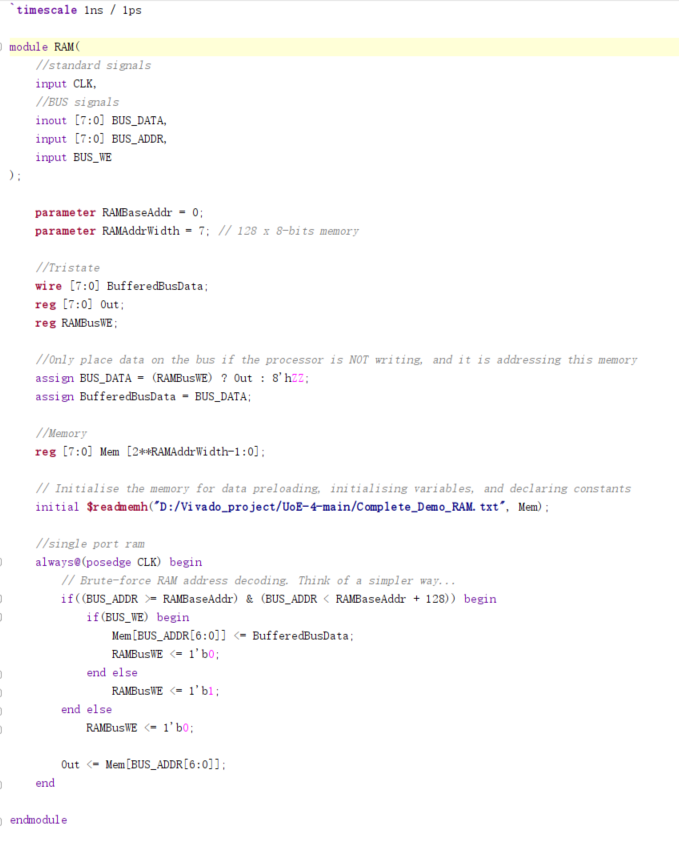
#### **Responsive Systems**

* **Interactive Applications**: For interactive and GUI-based applications, RAM ensures that data required for user interactions is quickly accessible, providing a smooth and responsive user experience.
* **Multimedia Processing**: RAM is critical for multimedia applications such as video and audio processing, where large amounts of data need to be processed and accessed in real-time.

### Practical Example and Application

Here's the designed Verilog code illustrating RAM usage in a RISC-V microprocessor design with a focus on main memory for storing instructions and data.

**RAM Module Definition in Verilog**



**3.Simulation and test result**

**3.1 Assembly code and machine code**

An assembly language is a type of low-level programming language that is intended to communicate directly with a computer’s hardware. Unlike machine language, which consists of binary and hexadecimal characters, assembly languages are designed to be readable by humans.

Low-level programming languages such as assembly language are a necessary bridge between the underlying hardware of a computer and the higher-level programming languages—such as Python or JavaScript—in which modern software programs are written. [12].

Fundamentally, the most basic instructions executed by a computer are binary codes, consisting of ones and zeros. Those codes are directly translated into the “on” and “off” states of the electricity moving through the computer’s physical circuits. In essence, these simple codes form the basis of “machine language,” the most fundamental variety of programming language.

Of course, no human can construct modern software programs by explicitly programming ones and zeros. Instead, human programmers must rely on various layers of abstraction that can allow themselves to articulate their commands in a format that is more intuitive to humans.

Specifically, modern programmers issue commands in so-called “high-level languages,” which utilize intuitive syntax such as whole English words and sentences, as well as logical operators such as “and,” “or,” and “else” that are familiar to everyday usage.

Ultimately, however, these high-level commands need to be translated into machine language. Rather than doing so manually, programmers rely on assembly languages whose purpose is to automatically translate between these high-level and low-level languages. The first assembly languages were developed in the 1940s, and though modern programmers and modern natural language processors spend very little time dealing with assembly languages, they nevertheless remain essential to the overall functioning of a computer.

Assembly languages were originally designed with a one-to-one correspondence between mnemonics and machine language instructions, as shown in this example.1 Translating from mnemonics to machine language became the job of a systems program known as an assembler. Assemblers were eventually augmented with elaborate “macro expansion” facilities to permit programmers to define parameterized abbreviations for common sequences of instructions. The correspondence between assembly language and machine language remained obvious and explicit, however. Programming continued to be a machine-centered enterprise: each different kind of computer had to be programmed in its own assembly language, and programmers thought in terms of the instructions that the machine would actually execute.

One of the most common operations computers perform is addition. Figure 25 shows code for adding variables b and c and writing the result to a. The code is shown on the left in a high-level language (using the syntax of C, C++, and Java) and then rewritten on the right in RISC-V assembly language. Note that statements in a C program end with a semicolon.

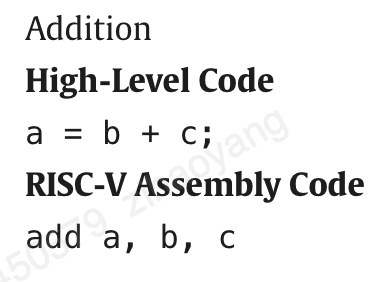


Figure 25: Addition codes

The first part of the assembly instruction, add, is called the mnemonic and indicates what operation to perform. The operation is performed on b and c, the source operands, and the result is written to a, the destination operand.

The RISC-V instruction set makes the common case fast by including only simple, commonly used instructions. The number of instructions is kept small so that the hardware required to decode the instruction and its operands can be simple, small, and fast. More elaborate operations that are less common are performed using sequences of multiple simple instructions. Thus, RISC-V is a reduced instruction set computer (RISC) architecture. Architectures with many complex instructions, such as Intel’s x86 architecture, are complex instruction set computers (CISC). For example, x86 defines a “string move” instruction that copies a string (a series of characters) from one part of memory to another. Such an operation requires many, possibly even hundreds, of simple instructions in a RISC machine. However, the cost of implementing complex instructions in a CISC architecture is added hardware and overhead that slows down the simple instructions. In the Preface, we describe several simulators and tools for compiling and simulating C and RISC-V assembly code [13].

Assembly language can usually be executed faster than high level languages. It's relatively easy to insert of delete components of assembly language code, and assembly language usually requires fewer instructions to complete a task when compared against other types of languages.

Assembly languages are also often used by programmers wanting greater control over their computers as assembly languages allow you to directly manipulate your hardware. Because of its speed and importance, some programs are specifically written using assembly language as the code can usually remain smaller.

Assembly languages do tend to have several drawbacks. Long programs written using assembly language usually require heavier computing power and can not be run on small computers. Some may find the syntax of assembly language more difficult to remember, and it may take some longer to code using assembly language as it is more complex. In addition, assembly language is usually not portable between different makes of different computers; similar to how employee benefits are lost when switching companies, languages cannot be seamlessly translated to different computers.

Pros:

Execution may be simpler compared to other languages

Execution is usually faster compared to other languages

Allows for direct control over hardware

Code may remain smaller compared to other languages

Cons:

Programming may be more challenging to pick up compared to high-level languages

Syntax of assembly languages is difficult

Not portable between machines

3.2 Machine code

Machine code, also known as machine language, is the elemental language of computers. It is read by the computer's central processing unit (CPU), is composed of digital binary numbers and looks like a very long sequence of zeros and ones. Ultimately, the source code of every human-readable programming language must be translated to machine language by a compiler or an interpreter, because binary code is the only language that computer hardware can understand [14].

Each CPU has its own specific machine language. The processor reads and handles instructions, which tell the CPU to perform a simple task. Instructions are comprised of a certain number of bits. If instructions for a particular processor are 8 bits, for example, the first 4 bits part (the opcode) tells the computer what to do and the second 4 bits (the operand) tells the computer what data to use.

Depending upon the processor, a computer's instruction sets may all be the same length, or they may vary, depending upon the specific instruction. The architecture of the particular processor determines how instructions are patterned. The execution of instructions is controlled by firmware or the CPU's internal wiring.

Human programmers rarely, if ever, deal directly with machine code anymore. If developers are debugging a program at a low level, they might use a printout that shows the program in its machine code form. The printout, which is called a dump, is very difficult and to work with a tool called a dump. Utility programs used to create dumps will often represent four bits by a single hexadecimal to make the machine code easier to read and contain other information about the computer's operation, such as the address of the instruction that was being executed at the time the dump was initiated.

Advantages of Machine Language

Some advantages of machine language are listed below:

1. Machine languages are faster in execution because they are in binary form.
2. Machine language does not need to be translated , because it is already present in simple binary form.
3. The CPU directly executes the machine language.
4. The evolution of the computer system and operating system over the time period is due to machine language.
5. Machine languages are used in developing a high-grade computer system.

Disadvantages of Machine Language

Some disadvantages of machine language are listed below:

1. Machine language is complex to understand and memorize.
2. Writing codes in machine language is time-consuming.
3. It is very difficult to resolve bugs and errors present in the codes and programs.
4. Codes written in machine languages are more prone to error.
5. Machine languages are not easy to modify.
6. Machine language are platform Independent.

Low-Level but Readable: Although it is low-level, assembly language is more readable and writable compared to binary language. One-to-One Correspondence: Each assembly language instruction typically corresponds to a single machine code instruction. Uses Mnemonics: Instructions are represented using mnemonics (e.g., MOV for move, ADD for addition) and symbolic addresses. Architecture-Specific: Like binary language, assembly language is also specific to a particular CPU architecture.

Assembler: An assembler is used to translate assembly language code into machine code.

### Machine and Assembly Comparison and Relationship

1. **Level of Abstraction**:
   * **Binary Language**: Directly corresponds to the machine-level instructions that the CPU executes.
   * **Assembly Language**: Provides a slightly higher abstraction level with more human-readable mnemonics and labels.
2. **Ease of Use**:
   * **Binary Language**: Extremely difficult to read, write, and debug.
   * **Assembly Language**: Easier to understand and write than binary language but still requires knowledge of the hardware.
3. **Translation**:
   * **Binary Language**: No translation needed as it's executed directly by the CPU.
   * **Assembly Language**: Requires an assembler to translate assembly instructions into machine code.
4. **Portability**:
   * Both binary and assembly languages are not portable across different CPU architectures. Code written for one type of processor typically will not run on another without modification.

Practical Use Cases

When to Use Binary Language: Rarely written by humans directly due to its complexity.

Used primarily in development and debugging tools, or in certain high-performance computing situations where you may manually manipulate binary code (e.g., embedded systems programming).

When to Use Assembly Language:

Embedded Systems: Where low-level hardware manipulation and optimization are required.

Performance Optimization: For performance-critical portions of code that require fine-tuned optimization.

Device Drivers: To interface directly with hardware.

Reverse Engineering: For analyzing and understanding compiled binary programs.

Example in RISC-V Assembly Language

Assembly language and binary language (machine code) are both low-level programming languages that are closely related to the architecture of a computer's hardware. Both play crucial roles in the development and functioning of software systems, but they operate at different levels of abstraction.

In this project, I choose to use machine code to realize remote car control because it is hard to write and implement a compiler due to the FPGA resource limitation.

|  |  |  |
| --- | --- | --- |
| Address (Hex) | Value (Hex) | Comment |
| //VGA |  |  |
| @0x00 | 0x00 | //output of Hcounter |
| @0x01 | 0x00 | //output of Vcounter |
| @0x02 | 0xA0 | //max of Hcounter 160 |
| @0x03 | 0x78 | //max of Vcounter 120 |
| @0x04 | 0x00 | //constant zero |
| @0x05 | 0x01 | //pixel data |
| @0x06 | 0x00 | //H index |
| @0x07 | 0x00 | //V index |
| @0x08 | 0x00 | //H base addr 1 0 |
| @0x09 | 0x35 | //H base addr 2 53 |
| @0x0A | 0x6A | //H base addr 3 106 |
| @0x0B | 0x00 | //V base addr 1 0 |
| @0x0C | 0x28 | //V base addr 2 40 |
| @0x0D | 0x50 | //V base addr 3 80 |
| @0x0E | 0x01 | //constant one |
| //IR transmitter |  |  |
| @0x0F | 0x09 | //left mouse button |
| @0x10 | 0x0A | //1010 Forward left |
| @0x11 | 0x08 | //1000 Forward |
| @0x12 | 0x09 | //1001 Forward right |
| @0x13 | 0x02 | //0010 Left |
| @0x14 | 0x00 | //0000 Idle |
| @0x15 | 0x01 | //0001 Right |
| @0x16 | 0x06 | //0110 Backward left |
| @0x17 | 0x04 | //0100 Backward |
| @0x18 | 0x05 | //0101 Backward right |
| @0x19 | 0x00 | //H |
| @0x1A | 0x00 | //V |
| @0x1B | 0x00 | //0 |
| @0x1C | 0x01 | //1 |
| @0x1D | 0x02 | //2 |
| @0x1E | 0x10 | //16 |
| @0x1F | 0x13 | //19 |

Table 4

RAM is volatile memory, which means that the data stored in it is lost when the power is turned off. RAM is typically used to store the operating system, application programs, and data that the computer is currently using. Therefore, in this project, I save various parameters like car direction navigation control command (such as Forward, Backward, Left, Right and so on), VGA screen display address and some specific constants.

ROM is non-volatile memory, which means that the data stored in it is retained even when the power is turned off. ROM is typically used to store the computer’s BIOS (basic input/output system), which contains the instructions for booting the computer, as well as firmware for other hardware devices. Here is the C code of execution of control car system

while(1):

{

if (X < 53){

if(Y<40)

command = 1010;

else if (40<Y<80)

command = 1000;

else

command = 1001;

}

else if (53<X<106){

if(Y<40)

command = 0010;

else if (40<Y<80)

command = 0000;

else

command = 0001;

}

else{

if(Y<40)

command = 0110;

else if (40<Y<80)

command = 0100;

else

command = 0101;

}

}

And it is necessary to translate it into machine code since there isn’t complier due to limitation of FPGA resource.

|  |  |  |
| --- | --- | --- |
| Address | Data | Comment |
| //checking X position |  |  |
| @0x9A | 0x00 | //Read mouse X to A |
| @0x9B | 0xA1 | //Address of mouse X |
| @0x9C | 0x01 | //Read H 53 to B |
| @0x9D | 0x09 | //address of H 53 |
| @0x9E | 0xB6 | //If mouse X less than 53 |
| @0x9F | 0xAA | //Go to set H to 0 |
|  |  |  |
| @0xA0 | 0x01 | //Else read H 106 to B |
| @0xA1 | 0x0A | //Address of H 106 |
| @0xA2 | 0xB6 | //If mouse X less than 106 |
| @0xA3 | 0xB0 | //Go to set H to 1 |
|  |  |  |
| @0xA4 | 0x00 | //Else read 2 to A |
| @0xA5 | 0x1D | //Address of 2 |
| @0xA6 | 0x02 | //Write 2 to H |
| @0xA7 | 0x19 | //H=2 |
| @0xA8 | 0x07 | //Go to addr |
| @0xA9 | 0xB4 | //Address of check V |
|  |  |  |
| @0xAA | 0x00 | //Read 0 to A |
| @0xAB | 0x1B | //Address of 0 |
| @0xAC | 0x02 | //Write 0 to H |
| @0xAD | 0x19 | //H=0 |
| @0xAE | 0x07 | //Go to addr |
| @0xAF | 0xB4 | //Address of check V |
|  |  |  |
|  |  |  |
| @0xB0 | 0x00 | //Read 1 to A |
| @0xB1 | 0x1C | //Address of 1 |
| @0xB2 | 0x02 | //Write 1 to H |
| @0xB3 | 0x19 | //H=1 |
|  |  |  |
| //checking Y position |  |  |
| @0xB4 | 0x00 | //Read mouse Y to A |
| @0xB5 | 0xA2 | //Address of mouse Y |
| @0xB6 | 0x01 | //Read V 40 to B |
| @0xB7 | 0x0C | //Address of V40 |
| @0xB8 | 0xB6 | //If mouse Y less than 40 |
| @0xB9 | 0xC4 | //Go to set V to 16 |
|  |  |  |
| @0xBA | 0x01 | //Else read V 80 to B |
| @0xBB | 0x0D | //Address of V 80 |
| @0xBC | 0xB6 | //If mouse Y less than 80 |
| @0xBD | 0xCA | //Go to set V to 19 |
|  |  |  |
| @0xBE | 0x00 | //Else read 22 |
| @0xBF | 0x20 | //Address of 22 |
| @0xC0 | 0x02 | //Write 22 to V |
| @0xC1 | 0x1A | //address of V, V=22 |
| @0xC2 | 0x07 | //Go to addr |
| @0xC3 | 0xCE | //Address of send command |
|  |  |  |
| @0xC4 | 0x00 | //Read 16 |
| @0xC5 | 0x1E | //Address of 16 |
| @0xC6 | 0x02 | //Write 16 to V |
| @0xC7 | 0x1A | //Address of V, V=16 |
| @0xC8 | 0x07 | //Go to addr |
| @0xC9 | 0xCE | //Address of send command |
|  |  |  |
| @0xCA | 0x00 | //Read 19 |
| @0xCB | 0x1F | //Address of 19 |
| @0xCC | 0x02 | //Write 19 to V |
| @0xCD | 0x1A | //Address of V, V=19 |
|  |  |  |
| //read mouse status |  |  |
| @0xCE | 0x00 | //read mouse status to A |
| @0xCF | 0xA0 | //address of mouse status |
| @0xD0 | 0x01 | //read 9 to B, means left clicked on mouse |
| @0xD1 | 0x0F | //address of 9 |
| @0xD2 | 0x96 | //if mouse status==9 |
| @0xD3 | 0xD8 | //read command |
|  |  |  |
| @0xD4 | 0x00 | //else read 0 to A |
| @0xD5 | 0x1B | //address of 0 |
| @0xD6 | 0x07 | //go to address of write command |
| @0xD7 | 0xDE | //address of write command |
|  |  |  |
| @0xD8 | 0x00 | //Read H to A |
| @0xD9 | 0x19 | //Address of H |
| @0xDA | 0x01 | //Read V to B |
| @0xDB | 0x1A | //Address of V |
| @0xDC | 0x04 | //A=H+V |
| @0xDD | 0x0B | //Dereference A |
|  |  |  |
|  |  |  |
| @0xDE | 0x02 | //Write A to IR |
| @0xDF | 0x90 | //address of IR |
|  |  |  |

Table 7

3.3 Demo and Resource Cost

The resource utilization is shown below

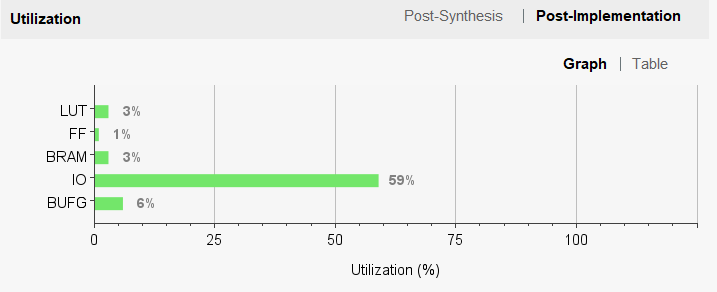


Figure : FPGA resource utilization

**4, Conclusion**

The implementation of a RISC-V based microprocessor on FPGA represents a significant advancement in the field of custom processor design and reconfigurable computing. This project has demonstrated the feasibility and advantages of leveraging the open-source RISC-V instruction set architecture in conjunction with the flexible and powerful capabilities of FPGA technology. Throughout this work, we have successfully designed, developed, and tested a RISC-V compliant microprocessor, showcasing the inherent benefits of this approach.

The modularity and extensibility of the RISC-V ISA allowed us to tailor the microprocessor to specific application requirements, providing a robust platform for innovation in hardware design. By utilizing HDL for detailed hardware description, synthesis tools for efficient implementation, and comprehensive verification methodologies, we ensured that the microprocessor meets stringent performance and reliability standards.

Furthermore, the adaptability of RISC-V on FPGAs facilitated rapid prototyping and iterative development, making it an ideal solution for both academic research and industrial applications. The extensive software support, including toolchain enhancements and optimized compilers, further streamlined the development process, allowing us to harness the full potential of the microprocessor.

Our benchmarking and application-specific testing highlighted notable performance gains and the ability to customize the architecture for specialized tasks, thereby validating the efficacy of our design choices. This work not only reaffirms the promise of RISC-V as a versatile and efficient ISA but also underscores the transformative potential of FPGA-based development in creating next-generation microprocessors.

In conclusion, the successful implementation of a RISC-V based microprocessor on FPGA has opened new avenues for customizable, high-performance computing solutions. This project serves as a blueprint for future endeavors, providing valuable insights and methodologies that can be leveraged by researchers and practitioners to advance the state of the art in reconfigurable computing and processor design.

**Reference**

[1] M. H. Banna Saif, N. U. Sadad and M. N. Islam Mondal, "FPGA Implementation of Educational RISC- V Processor Suitable for Embedded Applications," 2023 International Conference on Electrical, Computer and Communication Engineering (ECCE), Chittagong, Bangladesh, 2023, pp. 1-5,

[2] David A. Patterson and John L. Hennessy *“Computer Organization and Design RISC-V Edition: The Hardware Software Interface.”* Page 22 Morgan Kaufmann Publishers Inc., 2017, San Francisco, CA, USA.

[3] Andrew Waterman, Krste Asanovi, and Five Inc. *“The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.”* CS Division, EECS Department, University of California, Berkeley 2017

[4] RISC-V *"RISC-V GNU Compiler Toolchain"* Github 2021 https://github.com/riscv/riscv-gnu-toolchain/

[5] ShakthiProcessorseriesIITMadras2015

[6] T-Y Yeh and Y.N. Patt, “Two-Level Adaptive Branch Prediction”, Technical Report CSE-TR-117-91, Computer Science & Engineering Division, Department of EECS, University of Michigan, (Nov. 1991).

[7] C. R. Baugh and B. A. Wooley, “A two’s complement parallel array multiplication algorithm,” IEEE Trans. Comput., vol. C-22, no. 12, pp. 1045–1047,

Dec. 1973.

[8] Wang, Yi-Ming & Wang, Jinn-Shyan. (2004). An all-digital 50% duty-cycle corrector. Proceedings - IEEE International Symposium on Circuits and Systems. 2. II - 925. 10.1109/ISCAS.2004.1329424.

[9] Tim Wilmshurst, in Designing Embedded Systems with PIC Microcontrollers (Second Edition), 2010

[10] R. . -T. R. de J, A. Ordaz-Moreno, J. A. Vite-frias and A. Garcia-Perez, "8-bit CISC Microprocessor Core for Teaching Applications in the Digital Systems Laboratory," 2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006), San Luis Potosi, Mexico, 2006, pp. 1-5, doi: 10.1109/RECONF.2006.307782.

[11] S.Navalgund, Siddalingesh & Tonse, Prakash. (2013). Design, Development and Implementation of ALU, RAM and ROM for 8051 Microcontroller on FPGA using VHDL. International Journal of Computer Applications. 80. 8-14. 10.5120/13823-1280.

[12] Randall Hyde. 2003. The Art of Assembly Language. No Starch Press, USA.

[13] Scott, M. L. (2006). Programming language pragmatics Michael L. Scott. Morgan Kaufmann pub.

[14] Mansfield, R. (1983). Machine language for beginners: Machine language programming for basic language programmers. Compute! Books.

Scott, M. L. (2006). Programming language pragmatics Michael L. Scott. Morgan Kaufmann pub.