Design and Implementation of Five Stages Pipelined RISC Processor on FPGA

Dania Alamen (022160629)

Amani BenYousuf (022150310)

A report submitted in part fulfilment of the degree of

Bachelor in Computer Engineering

Supervisor: Dr. Mohamed Eljhani

Department of Computer Engineering

University of Tripoli

November 15, 2024

Declaration

This Pipelined Processor project report has been prepared on the basis of our own work for the award of the degree of Bachelor in Computer Engineering under the guidance of Dr. Mohamed Eljhani.

Word Count: approximately 15,000 words

Student Names:

Date of Submission:

Signature:

Acknowledgement

I would like to thank my project supervisor, Dr. Mohamed Eljhani for providing nice ideas to work upon and to his valuable guidance and support for doing this project. I sincerely want to thank my Dr. for his time to review my project report and their suggestion

Abstract

The main purpose of this project is to design, verify and implement a 5 stages pipelined processor that used as embedded system, which is a subset of a 16-bit RISC (Reduced Instruction Set Computer) CPU (Central Processing Unit) architecture. The basic modules of this processor are programmed and simulated using Verilog HDL (Hardware Description Language), and implemented on Cyclone IV FPGA (Field-Programmable Gate Arrays) board.

The Processor used as an embedded system in many applications like mobile computing, automobiles, industrial process controls, home appliances, office automation, and security etc, can be done by using processing element. Two commonly used processor architectures are CISC (Complex Instruction Set Computer) and RISC processor. The RISC processor uses large number of simple instructions and performs complex operations by using pipeline concept with improved performance.

Content

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Definition
RISC	Reduced Instruction Set Computer
CPU	Central Processing Unit
HDL	Hardware Description Language
FPGA	Field-Programmable Gate Arrays
CISC	Complex Instruction Set Computer
SPARC	Scalable Processor Architecture
MIPS	Million Instructions Per Second
ARM	Advanced RISC Machine
DEC	Department Equipment Corporation
PDP	Programmed Data Processor
ISA	Instruction Set Architectures
TTL	Transistor-Transistor Logic
OPCODE	Operation Code
PC	Program Counter
IF	Instruction Fetch
ID	Instruction Decode

Chapter 1: **Introduction**

Among all kinds of CPUs in use, the RISC CPU has the majority market share. It is most commonly used in embedded systems, which are in almost every consumer product on the market. RISC CPUs are basic and offer low-power consumption and small size. They are sometimes referred to as load-store processor because of the basic mechanics upon which it operates. The idea of the RISC CPU is to reduce the complexity of the system and increase the speed. Any complex operation can be split into smaller chunks that can be calculated simultaneously in most cases. Other important features of the RISC CPU include uniform instruction coding, which allows faster coding. A good example is that the opcode (operation code) is always in the same bit position in each instruction, which is always one word long. Another advantage is a homogeneous register set, which allows any register to be used in any context and simplifies compiler design. Lastly, complex addressing modes are replaced by sequences of simple arithmetic instructions. The convenience of the RISC processor is a direct explanation for why it dominates the CPU market.

Figure 1. Appearance of CPU

According to the characteristics of the executed instructions, the CPU is divided into RISC and CISC two. Table 1 compares the characteristics of RISC and CISC, and gives their representative products.

Table 1. Comparison of RISC and CISC

	Command function	Number of instructions	hardware	high speed	# of instructions performing same processing	Representative products
RISC	simple	Few	Simple	Suitable for	many	IBM Power, Sun MicroSystems SPARC, MIPS, ARM, etc.
CISC	complex	many	complex	Not suitable	few	Intel i386, 1BM System/360, DEC PDP, etc.

The instruction function of the RISC class CPU is simple and there are few types. Correspondingly, the instruction functions of CISC class CPUs are complex. The advantage of RISC instruction simplification is that the internal structure of the CPU can be simplified, which is suitable for high-speed operation. But when doing the same operation, since each instruction is functionally simple, it needs to use more instruction count than CISC. Although the internal structure of CISC is complex and not suitable for high-speed operation, the number of instructions for the same processing is less than that of RISC.

The biggest feature of the RISC architecture is that it only uses load and store instructions to access memory. This architecture is called Load/Store Architecture. The advantage of this is that the instruction set and pipeline design can be simplified. In this architecture, operation instructions can only operate on the data in the register.

MIPS stands for (Million Instructions Per Second) is a family of RISC ISA (Instruction Set Architectures). The MIPS processor, designed in 1984 by researchers at Stanford University, is a RISC processor. Compared with their CISC counterparts (such as the Intel Pentium processors), RISC processors typically support fewer and much simpler instructions.

The premise is, however, that a RISC processor can be made much faster than a CISC processor because of its simpler design. These days, it is generally accepted that RISC processors are more efficient than CISC processors; and even the only popular CISC processor that is still around (Intel Pentium) internally translates the CISC instructions into RISC instructions before they are executed.

RISC processors typically have “load-store architecture”. This means there are two instructions for accessing memory: A load instruction to load data from memory and a store instruction to write data to memory. It also means that none of the other instructions can access memory directly.

The instruction pipelining is a technique that is used to execute multiple instructions. The advantage of this technique is that it allows a faster throughput. In pipelining, the instruction execution is usually divided into stages. The number of stages varies depending on implementation. In our case, we have a five-stage pipelined structure. The instruction is split into five different steps which can be executed in parallel and the instructions can be processed concurrently, i.e., starting one instruction before finishing the previous one.

The main aim of the project is to design a 16-bit pipelined processor. Verilog HDL is used as the hardware description language for writing the modules. The length of instruction and registers is 8 bits long. The modules are simulated and the final results of the simulation are analyzed. The designed processor runs fixed point integer arithmetic and logical instructions, branch instructions, and integer load/store instructions.

1.1 Requirements

Apart from developing only a conceptual model, we propose implementing our design practically to develop a prototype. Following this, we state the following requirements for a successful project implementation:

- **Software Requirements:**
    - ModelSim HDL Simulator
    - Quartus II

- **Language Proficiency:**
    - Verilog

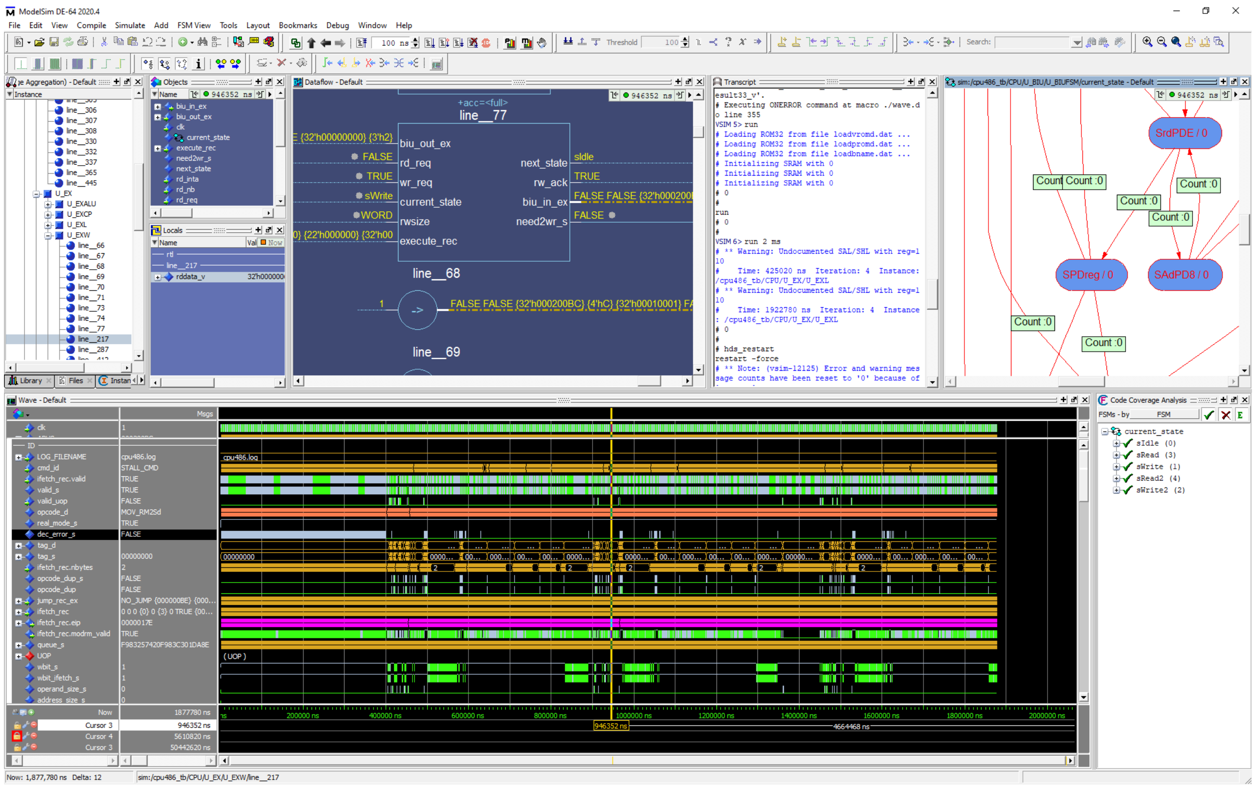
- **Hardware environment:**

- DE2i-150 FPGA Development Board module (Cyclone IV EP4CGX150DF31) device

1.2 ModelSim HDL Simulator

Modelsim is a program created by Mentor Graphics used for simulating behavioral, RTL, and gate-level code, including VHDL and Verilog gate libraries, with timing provided by the standard delay format (SDF). It is the most widely use simulation program in business and education.

Simulation is a critical step in designing FPGAs and ASICs. Simulation allows the designer to stimulate his or her design and see how the code that they wrote reacts to the stimulus. A great simulation will exercise all possible states of the design to ensure that all input scenarios will be handled appropriately.



```
esult33_v'.
# Executing ONERROR command at macro ./wave.d
o line 355
VSIM 5> run
# Loading ROM32 from file loadvromd.dat ...
# Loading ROM32 from file loadpromd.dat ...
# Loading ROM32 from file loadbname.dat ...
# Initializing SRAM with 0
# Initializing SRAM with 0
# Initializing SRAM with 0
# 0
#
run
# 0
#
VSIM 6> run 2 ms
# ** Warning: Undocumented SAL/SHL with reg=1
10
#    Time: 425020 ns  Iteration: 4  Instance:
/cpu486_tb/CPU/U_EX/U_EXL
# ** Warning: Undocumented SAL/SHL with reg=1
10
#    Time: 1922780 ns  Iteration: 4  Instance
: /cpu486_tb/CPU/U_EX/U_EXL
# 0
#
# hds_restart
restart -force
# ** Note: (vsim-12125) Error and warning mes
sage counts have been reset to '0' because of
```

Figure 2. ModelSim Simulator

1.3 Altera Quartus II

Altera Quartus is programmable logic device design software produced by Altera Quartus enables analysis and synthesis of HDL designs, which enables the developer to compile their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer. Quartus includes an implementation of VHDL and Verilog for hardware description, visual editing of logic circuits, and vector waveform simulation.

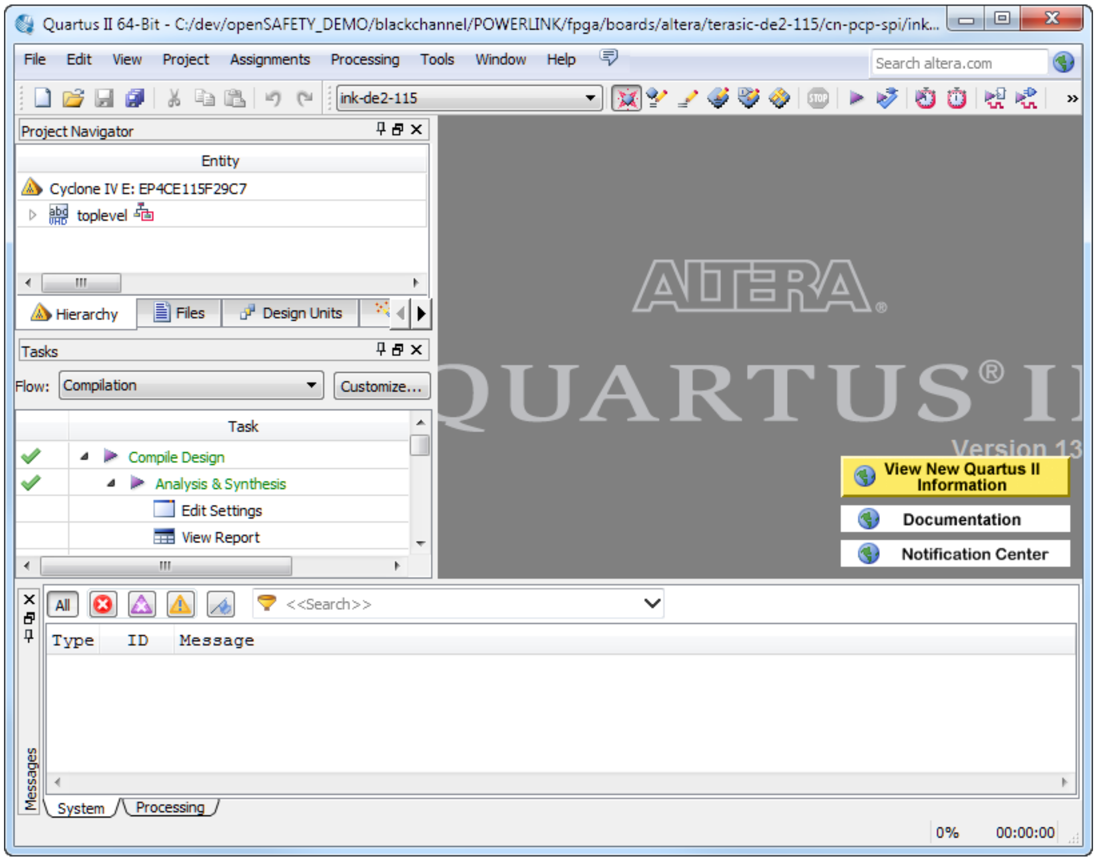


Figure 3. The main Altera Quartus II display

1.4 Verilog HDL Language

Verilog HDL is one of the most popular HDL languages developed by Gateway Design Automation as a proprietary language for logic simulation in 1984. The language became an IEEE standard in 1995 (IEEE STD 1364) and was updated in 2001 and 2005.

Verilog HDL language is a case-sensitive language. Used for describing a digital system like a network switch or a microprocessor or a memory or a flip−flop. It means, that by using an HDL we can describe any digital hardware at any level. Designs, which are described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.

Verilog HDL is commonly used to write text models that describe a logic circuit. Such a model is processed by a synthesis program, only if it is part of the logic design using simulation models to represent the logic circuits that interface to the design. This collection of simulation models is commonly called a testbench.

Verilog HDL language can be used for RTL (register transfer level) circuit design with a high degree of abstraction. RTL is a design model that describes circuit actions according to signal flow between registers and circuit logic. One can design hardware in a Verilog HDL IDE (for FPGA implementation such as Xilinx ISE, or Altera Quartus) to produce the RTL schematic of the desired circuit. After that, the generated schematic can be verified using simulation software which shows the waveforms of inputs and outputs of the circuit after generating the appropriate testbench. To generate an appropriate testbench for a particular circuit or Verilog HDL code, the inputs have to be defined correctly. For example, for clock input, a loop process or an iterative statement is required.

The key advantage of Verilog HDL when used for systems design is that it allows the behaviour of the required system to be described (modelled) and verified (simulated) before synthesis tools translate the design into real hardware (gates and wires).

Another benefit is that Verilog HDL allows the description of a concurrent system (many parts, each with its sub-behaviour, working together at the same time). Verilog HDL is a Dataflow language, unlike procedural computing languages such as BASIC, C, and assembly code, which all run sequentially, one instruction at a time.

1.5 DE2i-150 FPGA Board

FPGA stands for field-programmable gate array. At its core, an FPGA is an array of interconnected digital subcircuits that implement common functions while also offering very high levels of flexibility. Where the user can program what the logic gate does (be it a NAND or NOR or some form of SUM-of-PRODUCT implementation) or an adder, you as a user, can "program" the chip to perform that logic function. Now we can add another layer of user programmability – you can program how these logic gates are connected! In that way, we have a general programmable logic chip. Unlike the microprocessor where the program is just the instruction to fix digital hardware, here you can program the hardware itself!

The first FPGA was introduced by Xilinx in 1985. It has arrays of logic blocks that are programmable. It is surrounded by PROGRAMMABLE ROUTING RESOURCES, which allows the user to define the interconnections between the logic blocks. It also has lots of very flexible input and output circuits that programmable for TTL (Transistor-Transistor Logic), CMOS (Complementary Metal Oxide Semiconductor) , and other interface standards.

Nowadays, there are two major players in the FPGA domain: Xilinx and Altera (now part of Intel). These two companies dominate 90% of the FPGA market with roughly equal shares.

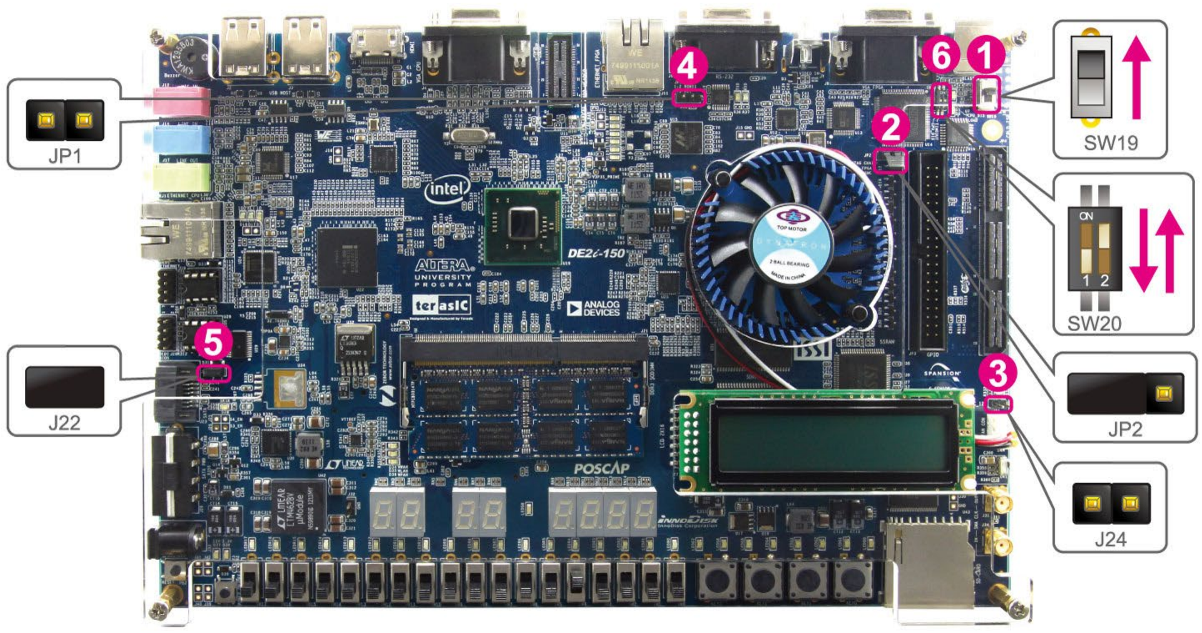


Figure 4. The DE2i-150 board (top view)

At this point when a Verilog HDL model completed here, we translate it into "gates and wires" that are mapped onto the programmable logic device such as a DE2i-150 FPGA, then it is the actual hardware being configured, rather than the Verilog HDL code being "executed" as if on some form of the processor chip.

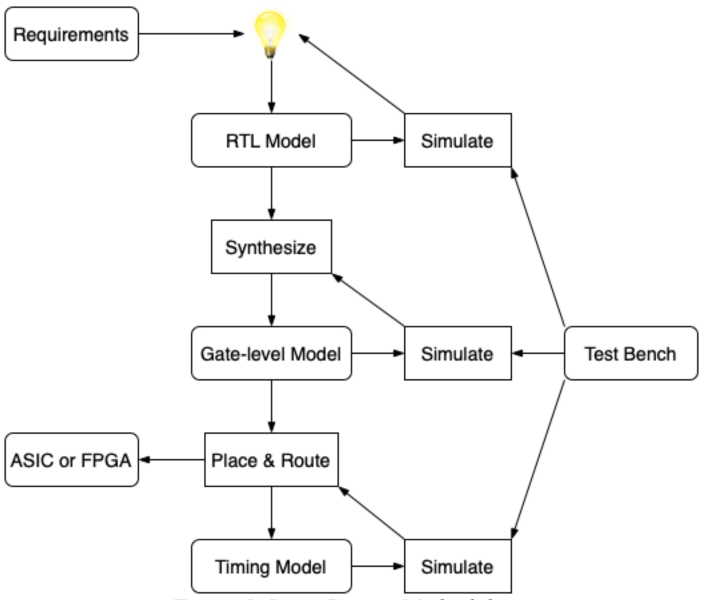


Figure 5. Basic Design Methodology.

Chapter 2: **Design Methodology**

A Pipelined Processor Design is referred to as a Pipelined CPU. The CPU is a device that executes various software instructions and data processing. In designing a CPU, we must first define its instruction set and how many instructions do we want? What are the instructions? What opcode do we assign to each of the instructions? How many bits do we use to encode an instruction? Once we have decided on the instruction set, we can proceed to design an architecture that can execute all the instructions in the instruction set. In this step we are creating a custom structure, so we need to answer questions such as How many registers do we need? Do we use a single register file separate register? How the different units are connected? Finally, we can design the control unit. Just like the dedicated processor, the control unit asserts the control signals to the processor. This finite state machine cycles through two main states: 1) idle; 2) execute. The control unit performs these states by sending the appropriate control signals to the processing. And the processor consists of general-purpose registers and various special-purpose registers such as PC (Program counter).

The five-stage design of the processor is similar to the MIPS architecture, it is designed according to the pipeline structure. As long as the CPU obtains data from the memory, the execution of each MIPS instruction is divided into five flowing stages, and each stage takes up a fixed time, usually only one processor clock cycle.

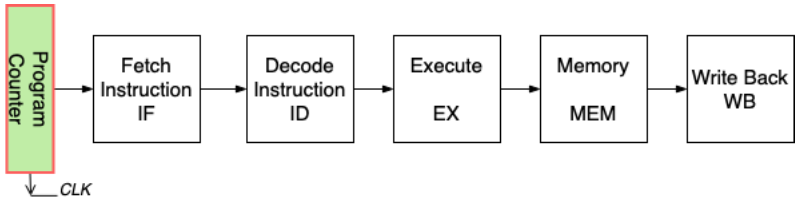


Figure 6. Single-Cycle CPU logic architecture

When the processor is designed, the execution phase of the processor is divided into the following five stages:

1. IF: Instruction Fetch. Get the next instruction from the instruction memory
2. ID: Instruction Decode (Read Register). Translate instructions, identify opcodes and operands, and read data from the register heap to the ALU input register.
3. EX (ALU): Execute, (arithmetic/logical operation). Complete arithmetic or logical operations in a clock cycle. Note that floating-point arithmetic and integer multiplication and division operations cannot be completed in a single clock cycle which we didn't use in our design.
4. MEM: Memory Access, memory data read or write. At this stage, instructions can read/write memory variables from data memory. On average, about three-quarters of instruction do not perform any operations at this stage. This stage is assigned to each instruction to ensure that no two instructions will access the data cache at the same time.
5. WB: Write Back. After the operation is completed, write the calculation results from the ALU output register back to the general register.

2.1 Datapath

The Datapath is responsible for the manipulation of data. It includes ALU, registers, memory elements for the temporary storage of data, buses, and multiplexers for the transfer of data between the different components in the Datapath. External data can be entered into the Datapath through the data input lines. Results from the computation are provided through the Datapath output lines.

For the Datapath to function correctly, appropriate control signals must be asserted at the right time. Control signals are needed for all the select and control lines for all the components used in the Datapath. This includes all the select lines for multiplexers, ALU, and other functional units having multiple operations, all the read/write enable signals for registers and register files, address lines for register files, and enable signals for tri-state buffers. The operation of the Datapath is determined by which control signals are asserted and at what time. In a processor, these control signals are generated by the control unit. In return, the Datapath needs to supply status signals back to the control unit for it to operate correctly. These status signals provide input information for the control unit to determine what operation to perform next. Since the Datapath performs all the functional operations of a processor, and the processor is for solving problems, therefore the Datapath must be able to perform all the operations required to solve the given problem. For example, if the problem requires the addition of two numbers, the Datapath, therefore, must contain an Adder. If the problem requires the storage of three temporary variables, the Datapath must have three registers.

However, even with these requirements, there are still many options as to what is implemented in the Datapath. Registers can be separate register units or combined in a register file. Furthermore, two temporary variables can share the same register if they are not needed at the same time. Datapath design is also referred to as the **register-transfer level (RTL)** design. In the register-transfer level design, we look at how data is transferred from one register to another or back to the same register. If the same data is written back to a register without any modification, then nothing has been accomplished. So before writing the data to a register, the data passes through one or more functional units and gets modified. The time from the reading of the data to the modifying of the data by functional units and finally to the writing of the data back to a register must all happen within one clock cycle.

The Top-view of the design should look as follows, in which select_out selects the output signal is related to the CPU board-level test by using 7-segment light.

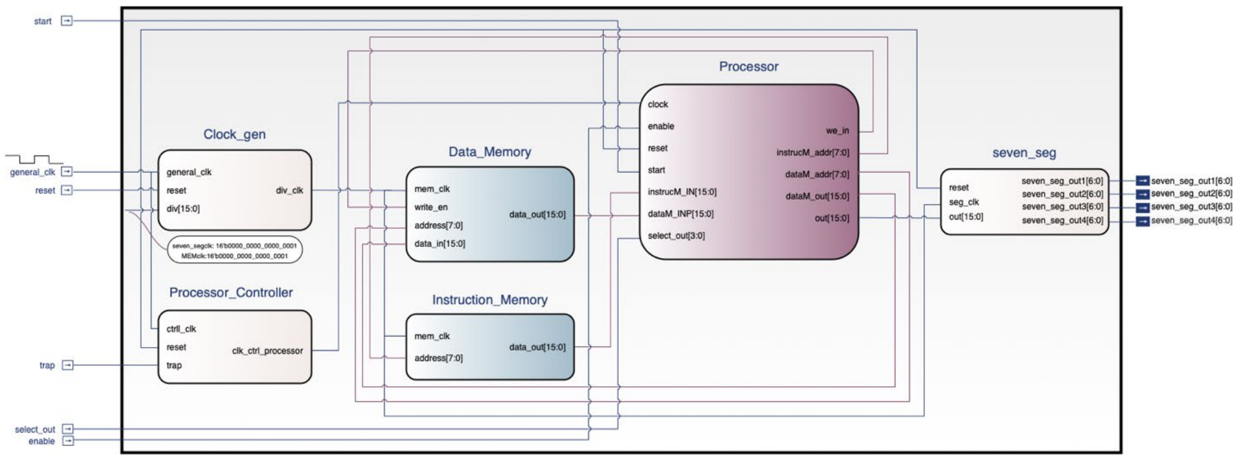


Figure 7. CPU top view

According to the Top view, the entire Processor is divided into 6 modules:

1. **Processor** main module: perform arithmetic processing with the instructions obtained from the instruction memory, and write or read data from the data memory. This is the core part.
2. **Instruction Memory** instruction module: Provide instructions for the Processor module, so that the Processor can find the memory address of the instruction through the value in the instruction register, thereby reading the instruction.
3. **Data Memory** data module: Processor reads data or writes data.
4. Clock Generator: is an electronic oscillator that produces a clock signal for use in synchronizing a circuit's operation.
5. **Processor Controller**: It is used to control the processor execution by the clock
6. **7-Segment Interface**: a seven-segment display is a form of an electronic display device for displaying decimal numerals that are an alternative to the more complex dot matrix displays.

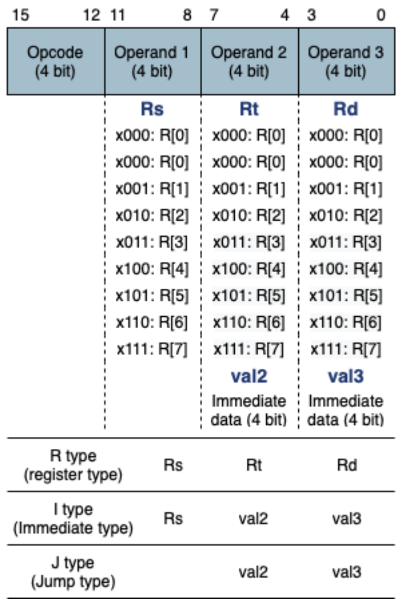
2.2 The Instruction Set Architecture

The instruction that we used is in three-address format, and the opcode length is 4 bits. According to the different operands, the instruction can be divided into three types, namely the register type (R type), the immediate type (I type), and the Jump type (J type).

There 8 general-purpose registers that are 16-bit long. These registers are used by the fixed point integer instructions. The general-purpose registers are selected by the 3-bit address in the register field in the instruction. Each of the general-purpose registers is used to store the result of the operations performed by the instruction. All the data manipulation is done in the registers which are internal to the processor.

The number of instructions implemented determines the number of bits required to encode all the instructions. All instructions are encoded using 4-bit except for instructions that have a memory address as one of its operands, in which case a second byte for the address is needed. The encoding scheme uses the first four bits as the opcode. Depending on the opcode, the last twelve bits are interpreted differently as follows.

Table 2. The Instruction set Format



15 12	11 8	7 4	3 0
Opcode (4 bit)	Operand 1 (4 bit)	Operand 2 (4 bit)	Operand 3 (4 bit)
	Rs	**Rt**	**Rd**
	x000: R[0]	x000: R[0]	x000: R[0]
	x000: R[0]	x000: R[0]	x000: R[0]
	x001: R[1]	x001: R[1]	x001: R[1]
	x010: R[2]	x010: R[2]	x010: R[2]
	x011: R[3]	x011: R[3]	x011: R[3]
	x100: R[4]	x100: R[4]	x100: R[4]
	x101: R[5]	x101: R[5]	x101: R[5]
	x110: R[6]	x110: R[6]	x110: R[6]
	x111: R[7]	x111: R[7]	x111: R[7]
		val2	**val3**
		Immediate data (4 bit)	Immediate data (4 bit)
R type (register type)	Rs	Rt	Rd
I type (Immediate type)	Rs	val2	val3
J type (Jump type)		val2	val3

The instructions that our processer can execute and the corresponding encoding are 8 instructions defined in Table 2. and the remaining 8 unused opcodes (0000, 0100, 0101, 0111, 1001, 1011, 1100, 1110) can be supplemented by other operations, such as add-immediate ADDI, sub-immediate SUBI, and OR operation. The Instruction column shows the syntax and mnemonic to use for the instruction when writing a program in Verilog language. The Encoding column shows the binary encoding for the instructions and the Operation column shows the binary encoding for the instructions and the Operation column shows the actual operation of the instruction. The instructions are separated into four categories:

1) Arithmetic and logical instructions for performing arithmetic and logic;
2) Data movement instructions (Load/Store Instructions) for transferring data between the accumulator, the general registers, and memory ;
3) Jump instruction for changing the instruction execution sequence; and
4) Miscellaneous instruction.

Table 3. 16-bit Instruction set encoding

Instruction	Operand1	Operand2	Operand3	Opcode	Operation
Arithmetic and logical instructions					
ADD	Rs	Rt	Rd	#1000	$Rs \leftarrow Rt + Rd$
SUB	Rs	Rt	Rd	#1010	$Rs \leftarrow Rt - Rd$
AND	Rs	Rt	Rd	#1101	$Rs \leftarrow Rt\ and\ Rd$
XOR	Rs	Rt	Rd	#1111	$Rs \leftarrow Rt\ xor\ Rd$
Data movement instructions					
LOAD	Rs	Rt	val3	#0010	$R[Rs] \leftarrow m[Rt + val3]$
STORE	Rs	Rt	val3	#0011	$m[Rt + val3] \leftarrow Rs$
Jump instruction					
JUMP		val2	val3	#0110	$jump\ to\ \{val2, val3\}$

Miscellaneous					
HALT				#0001	$halt$

There are two data movement instructions, one jump instruction, four arithmetic and logic instructions, and a miscellaneous instruction.

2.2.1 Two Operand Instructions

The Arithmetic instructions perform addition and subtraction. These instructions use general-purpose registers as their source and destination operands. Arithmetic instructions support both signed and unsigned operations. This carry is stored in the carry bit in a buffer register. If the carry flag buffer (CF buff) in the instruction is 1, then the Carry Flag of the condition register is updated. If the result of the arithmetic operation is zero, Zero Flag is set to 1. For the signed operation, the Negative Flag is set to 1 when the MSB is set to 1.

The Logical instructions are used to perform the logical operations such as logical AND and logical XOR. These instructions perform on the 16-bit operands.

If the instruction requires two operands, it allows the use of the accumulator (register A) for one operand. And the second operand is register B, then the last four bits in the encoding specify the register file number. An example of this is the LOAD (load accumulator from the register) instruction where it loads the accumulator with the content of the register file number specified in the last three bits of the encoding. Another example is the ADD instruction where it adds the contents of the accumulator with the content of the specified register file and put the result in the accumulator.

2.2.2 Instructions Using a Memory Address

For instructions that have a memory address as one of its operands, an additional 8 bits are needed to access the 265 bytes of memory space. These 8 bits (xxxx_xxxx) are specified in the 8 least significant bits of the second byte of the instruction. An example is the Load/Store instruction. The address of the memory location where the data is to be loaded/stored, where is specified in the second byte.

2.2.3 Jump Instruction

For jump instructions, the last four bits of the encoding also serve to differentiate between absolute and relative jumps. If the last eight bits are zeros, then it is an absolute jump, otherwise, they represent a sign and magnitude format relative displacement from the current location as specified in the PC. For example, the two-byte encoding 0110 0000 0000 0100 specifies an absolute unconditional jump to memory location 4. The first four bits (0110) specify the unconditional jump

The jump instruction transfers the program sequence to the memory address given in the operand based on the specified flag. Jump instructions are of two types: Unconditional Jump Instructions, and Conditional Jump Instructions. Unconditional Jump Instructions: Transfers the program sequence to the described memory address.

2.3 Five-stage pipeline

In addition to the instruction set, the most important thing in designing the CPU is the following block-level circuit diagram of the CPU. The code implementation of the five-stage pipeline was dependent on this diagram.

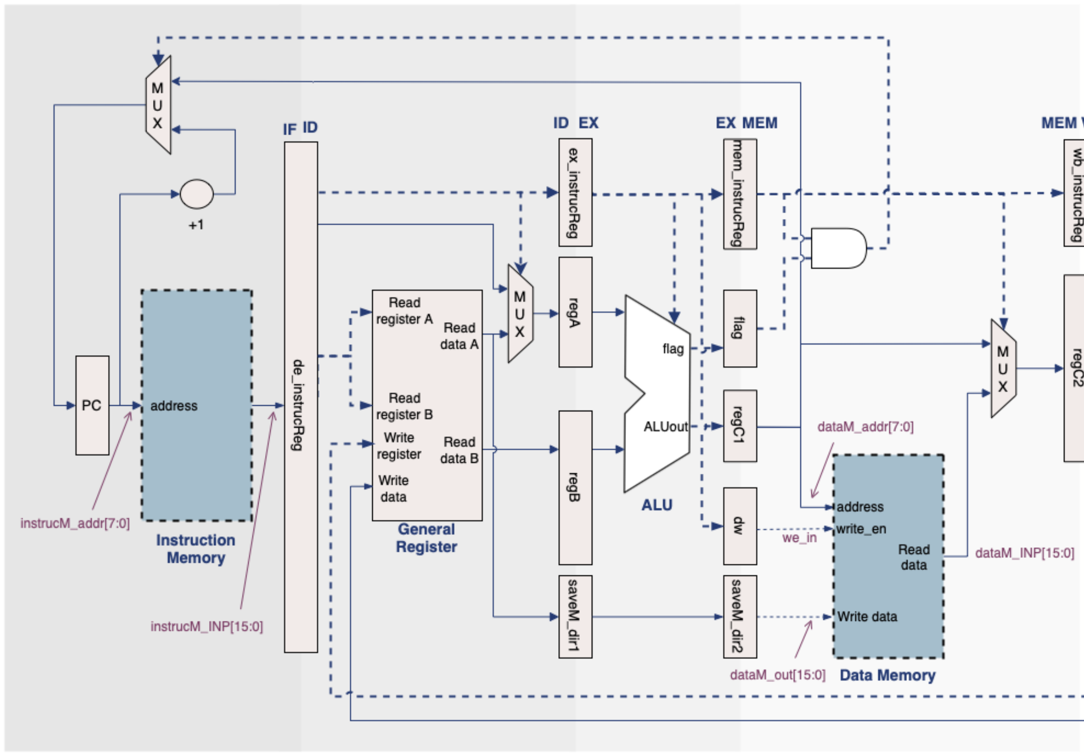


Figure 8. CPU block-level circuit diagram

The circuit diagram is not complicated. The CPU is nothing more than a combination of combinational logic circuits and sequential circuits. All the rectangles in the diagram mark the registers inside the CPU. The entire circuit diagram shows the flow of instructions and data inside the CPU. At each rising edge of the clock, the data of the register of the upper-stage pipeline will flow to the register of the next-stage pipeline through the intermediate combinational logic circuit. Therefore, a machine instruction is executed after 5 clock cycles. A brief description of the execution process of instruction is to first fetch an instruction from the memory according to the value of the PC, decode the instruction to extract two operands for operation, and decide whether to access the data memory and how to access it according to the instruction function and operation result, and finally according to the instruction. The function determines whether to perform a write-back operation, that is, to modify the value of the register.

The following will explain the CPU control and the behavior of each stage of the five-stage pipeline.

2.3.1 CPU control

The CPU control is naturally based on the finite state machine. There are only two main states: idle and execute. In the idle state, the CPU can enter the execute state only if enable and start are enabled at the same time.

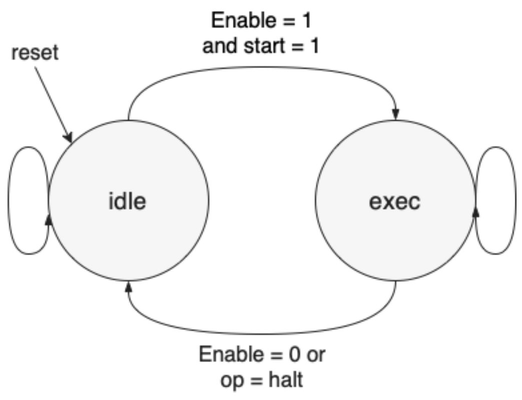


Figure 9. CPU control state diagram

2.3.2 Instruction Fetch

The Instruction Fetch (IF) stage is timing logic, each rising edge the CPU should read the instruction to be executed from the instruction memory, according to the value of the PC which holds the address of the instruction to be executed, and set the value of the PC in the next cycle (the instruction is read by outputting the value of the PC register to the instruction memory, and the memory returns the instruction in the address corresponding to the value which can be executed sequentially or jump to a specific address).in jump instruction, The transfer instruction directly jumps and handles the case that the pc value is delayed by one cycle, and the other case directly reads the next instruction.

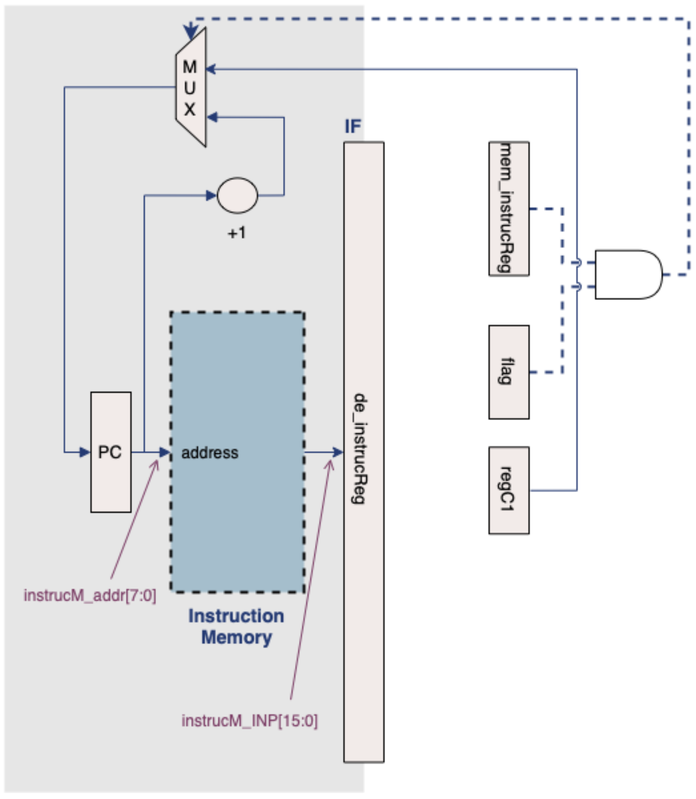


Figure 10. IF stage

Because reading memory is a function implemented by the memory module, the CPU only needs to give the instruction address instrucM_addr to get the corresponding instruction instrucM_INP.

2.3.3 ID

In the ID stage, the CPU needs to decode or extract the corresponding operand from the instruction according to the function of the instruction (i.e., the opcode). The operand may come from general-purpose registers R [0]-R [7], or it may be an immediate value. There are many kinds of instructions, including instructions for performing various operations, instructions for controlling the next command, instructions for reading and writing memory, and instructions for controlling the CPU. In addition, if the instruction is a STORE instruction, also prepare the data to be stored in memory. These instructions are decoded by a module in the CPU called an instruction decoder.

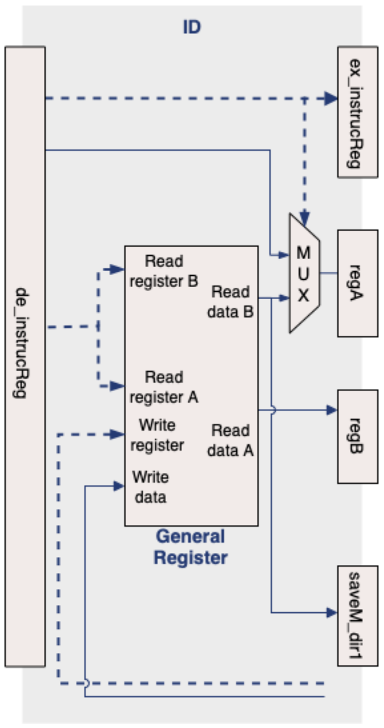


Figure 11. ID stage

2.3.4 EX

The EX-stage, the CPU executes and processes the ALU operation determined by the decoder and the flag register will setting. In addition, the CPU can read and process data from internal storage registers or external memory, and then write the results back to the register C1 or memory, if it is the STORE instruction, the memory write enable signal dw and the data saveM_dir2 to be written to the memory should also be given.

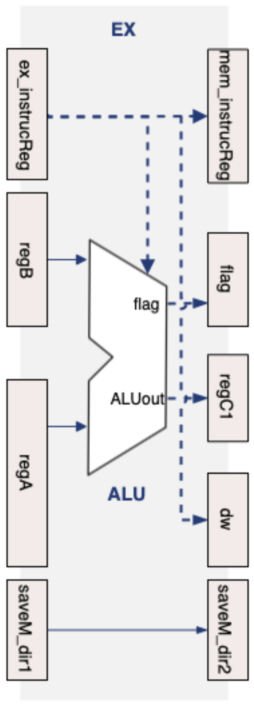
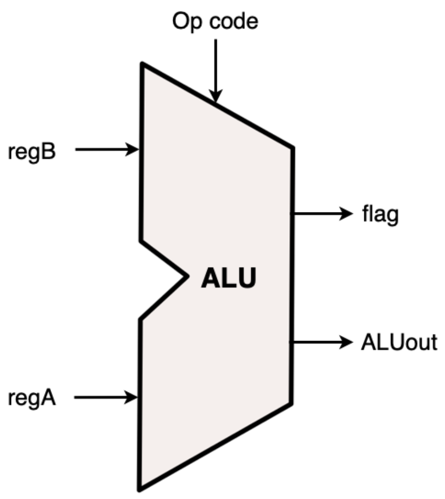


Figure 12. EX stage

When reading an instruction, the CPU outputs the value of the PC register to memory, and then retrieves the corresponding instruction from memory. The fetched instruction is stored in the instruction register, and instruction decoding is to decode the instruction stored in the instruction register to determine the operation to be processed. In most cases, while determining the operation to be processed, the CPU reads the data to be used by the operation from general-purpose registers. When the instruction is executed, the operand value is fetched from the general-purpose register, processed by the arithmetic unit, and the result is written back. The result of an operation performed by the CPU can be written back to a general-purpose register or to memory. The CPU can also read data from memory and return it as a result.

ALU

In computing, an Arithmetic Logic Unit (ALU) is a combinational digital circuit that performs arithmetic an bitwise operations on integer binary numbers.



Signal name [size in bits]	Description
regB[16]	Input data “regB”, from Reg [Rs] in most cases
regA[16]	Input data “regA”, from Reg [Rt] or the immediate in most cases.
ALUout[16]	Output data
Op code [4]	Generated by the control unit when the instruction currently in the ALU was in Decode

Figure 13. ALU interface

2.3.5 MEM

In the MEM stage, it is necessary to decide whether to access the memory and how to access it according to the instruction function and the operation result of the previous stage (as the memory address during memory operation). It is only valid for instructions that require memory operations such as LOAD and STORE and, in the rest, pass reg_C to reg_C1. The cf carry flag is updated at this stage.

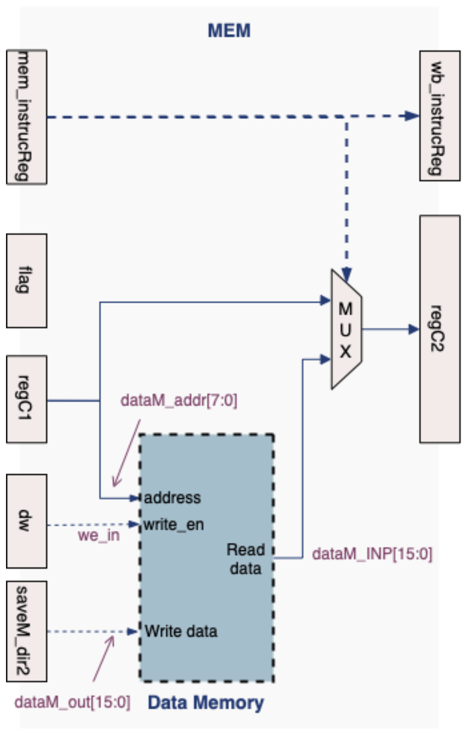


Figure 14. MEM stage

2.3.6 WB

The Write Back stage also decides whether to modify the value of the register and how to modify it according to the function of the instruction and the result of the previous stage. It is only valid for instructions that need to modify the value of the register. Write back writes the calculation result to the value of the instruction. Except for the jump instruction and the load and store instructions, the first operand (register) is written back. and just keep the register unchanged in other cases.

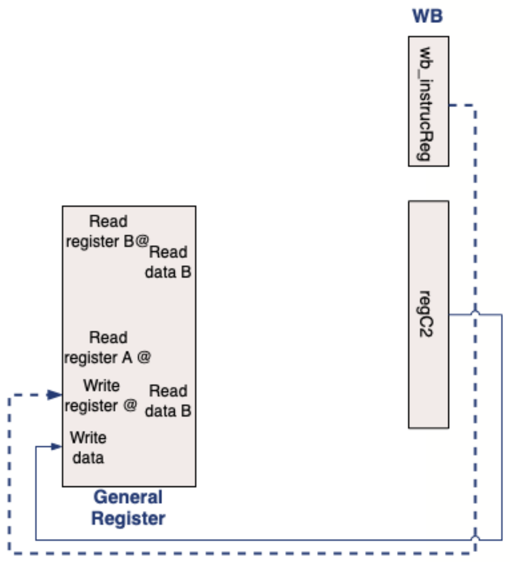


Figure 15. WB stage

2.4 Memory

Memory is the storage used to store runtime instructions (programs) and data. In order to distinguish it from the long-term storage of data and programs in a computer, memory is sometimes called main memory.

Memory uses the concept of addresses to manage stored data. Addresses represent where data is stored, just like where data exist in. Each data unit has an address. In most cases, the data unit is one byte (8 bits) long. This method is called byte addressing. Figure 16 illustrates the relationship between memory and addresses.

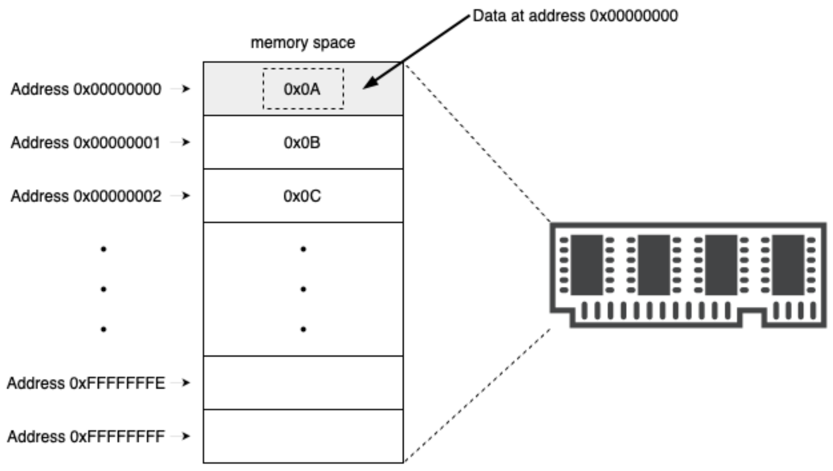


Figure 16. Memory and Address

The characteristic of memory is that the faster the speed, the higher the cost. Therefore, a hybrid architecture of various memory combinations such as "high-speed small-capacity", "medium-speed medium-capacity" and "low-speed large-capacity" is usually used. This construction is called a memory hierarchy. Figures 17 are examples of memory hierarchies.

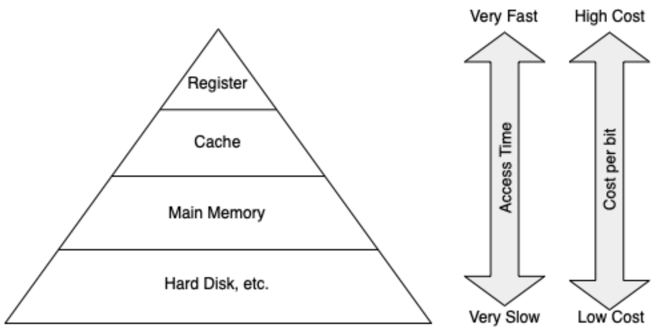


Figure 17. Memory hierarchy

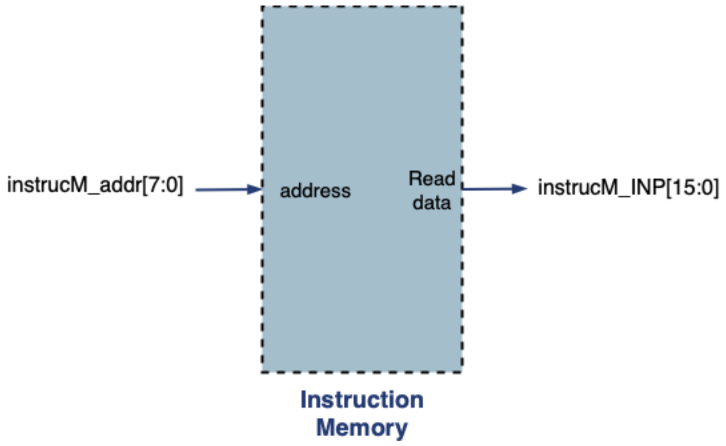
At the storage level, the fastest are the registers in the CPU. The CPU is much faster than the memory, and the direct access to the memory by the CPU is less efficient. To improve the memory access speed, a high-speed small-capacity memory called cache is added between the CPU and the memory.

A cache can temporarily buffer data read from memory. When the CPU accesses the memory, if the required data has been saved in the cache, it can directly read from the cache to improve the access efficiency. According to the different capacities and speeds, the cache is also divided into multiple levels, usually the first-level cache, the second-level cache, and other levels.

Memory is a reg array, with a combined logic circuit for reading memory and a timing circuit for writing memory. The initialization of memory can be initialized by a reset signal, or directly in the initial of the test bench file when software simulation.

2.4.1 Instruction Memory

The Instruction Memory stores all the prefetch instructions. It's a Combination logic, according to the address read, output an instruction. It does need one read port, which will have 16-bit width (fetch one instruction at a time). The instruction Memory can model as having arbitrarily fast asynchronous reads.



Signal name [size in bits]	Description
Address [16]	Current instruction to fetch
Read Data [16]	Current instruction fetched

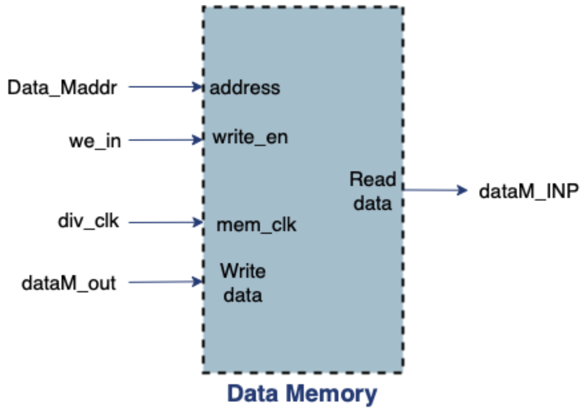
Figure 18. Instruction Memory interface

2.4.2 Data Memory

You need three ports for writing data into memory, reading data from memory, and specifying memory addresses. The data input and output need 16 bits to match the register size. There are two other ports, one is to enable or disable write operation and the other is clock signal.

Reading Data is a combinational logic, which directly reads the output of an instruction according to the address.

Writing to Data is sequential logic, and may be written once per cycle. To write according to the we_in signal, the clock frequency of the Memory needs to be faster than the CPU clock.



Signal name [size in bits]	Description
dataM_INP[16]	Data in Mem[address]. Output is still in the M stage if read asynchronously.
dataM_addr[8]	Address to perform the next read/write.
dataM_out[16]	Data to write to Mem[address].
we_in	Set high if the unit should perform a write on the next clock edge.
Div_clk	Clock signal to synchronize writes.

Figure 19. Data Memory interface

Normally, the reset signal should push the memory back to a known state, with the specific implementation defining exactly how. It doesn't matter if the design process resets synchronously or asynchronously. For simplicity, you may design under the assumption that reads are performed arbitrarily quickly without a clock, and writes are performed arbitrarily quickly after a clock edge.

Design a unit-testing testbench for your memory unit. At a minimum, it should perform some reads to verify the initial state after a reset, some write to arbitrary addresses, and some later reads to verify correctly written data.

2.5 Clock Generator

For our system clock, we use the built-in 50MHz clock that is available on the development board. In order to see some intermediate actions by the CPU, we used a clock divider to slow down the clock. One control word is executed in one clock cycle. In one clock cycle, first data is read from a register, then it passes through functional units and gets modified, and finally, it is written back to a register. Performing both a read and write from/to the same register in the same control word, i.e., same clock cycle does not create any signal conflict because the reading occurs immediately in the current clock cycle and is getting the original value that is in the register. The writing occurs at the beginning of the next clock cycle after the reading.

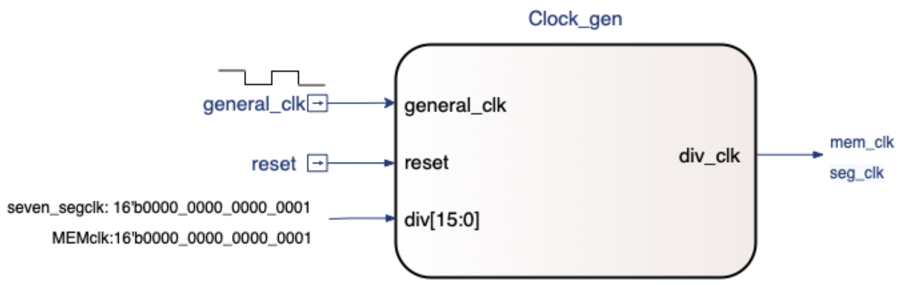


Figure 20. Clock Generation

2.6 Processor controller

The processor controller is used to control the running time of our processor. It consists of a clock, reset and trap as input the trap is work as the clock of the processor controller, and the output of the model is clk_ctrrl_processor to give a specific synchronization to the system.

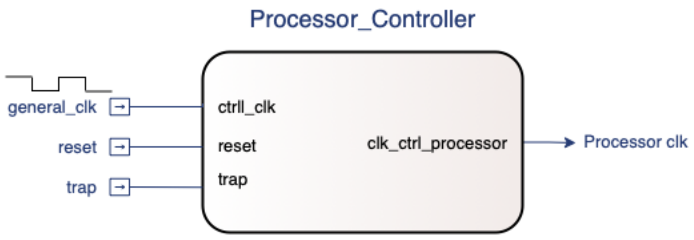


Figure 21. Processor controller

2.7 Seven-Segment Interface

In our design, we will interface 4-Digit 7-Segment to display the output according to the condition case. Each segment in a display is identified by an index from 0 to 6 by applying a low logic level to a segment that will light it up and applying a high logic level turns it off.

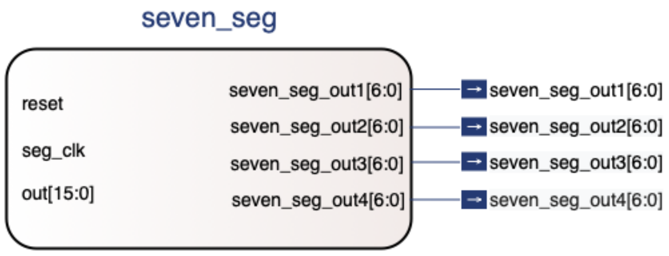


Figure 22. Seven Segment Interface

Figure 23. The 7-segment Display

Chapter 3: **Hazards Conflict**

In pipeline processing, due to the dependencies of various stages and the competition of hardware resources, the operation cannot be performed at the same time. The cause of pipeline failure is called a hazard. Hazard is divided into three types: structural hazard, data hazard, and control hazard.

3.1 Structural hazard

Due to the competition for hardware resources, the operation cannot be performed at the same time. Because the memory design adopts the separation of instruction memory and data memory, there will be no structural hazard.

3.2 Data hazard

The hazard caused by the data required for the execution of the instruction is not yet prepared. When the instructions to be executed rely on the unprocessed data, the instructions will not be executed immediately, causing a data hazard.

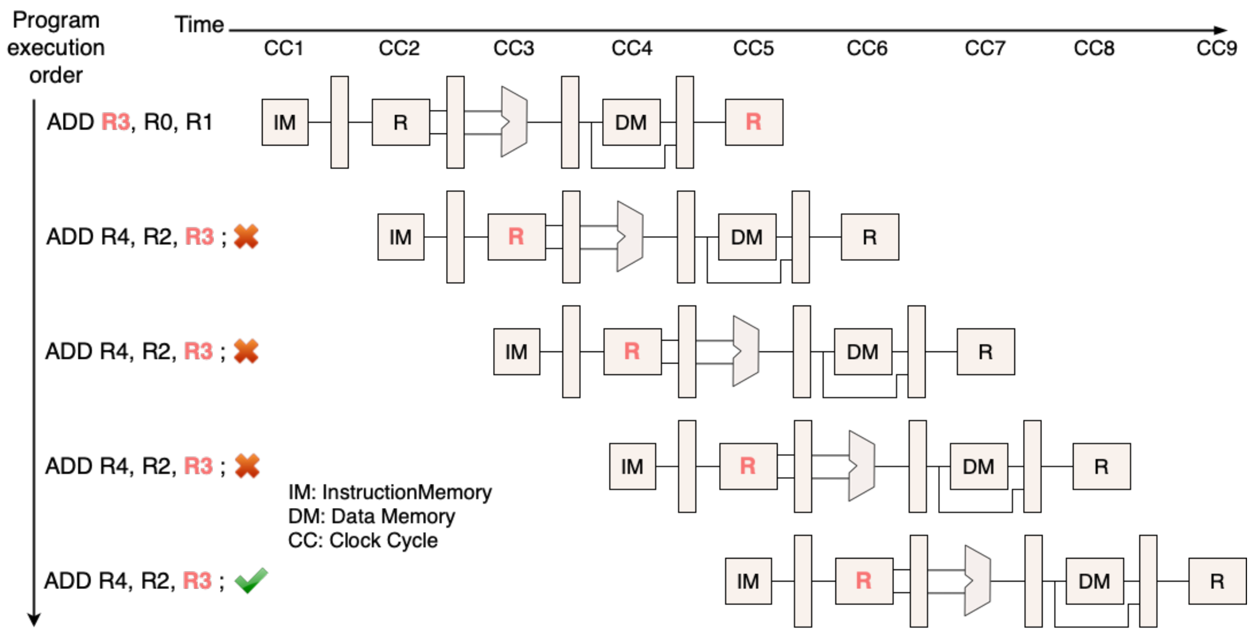


Figure 24. Data Hazard

Data risk can be simply avoided by inserting NOP (No Operation), but this greatly reduces the performance of the assembly line. Another method is through data forwarding. Although the data is written back in the WB stage, the actual operating results have been determined at the EX-stage and can be passed to the next instruction.

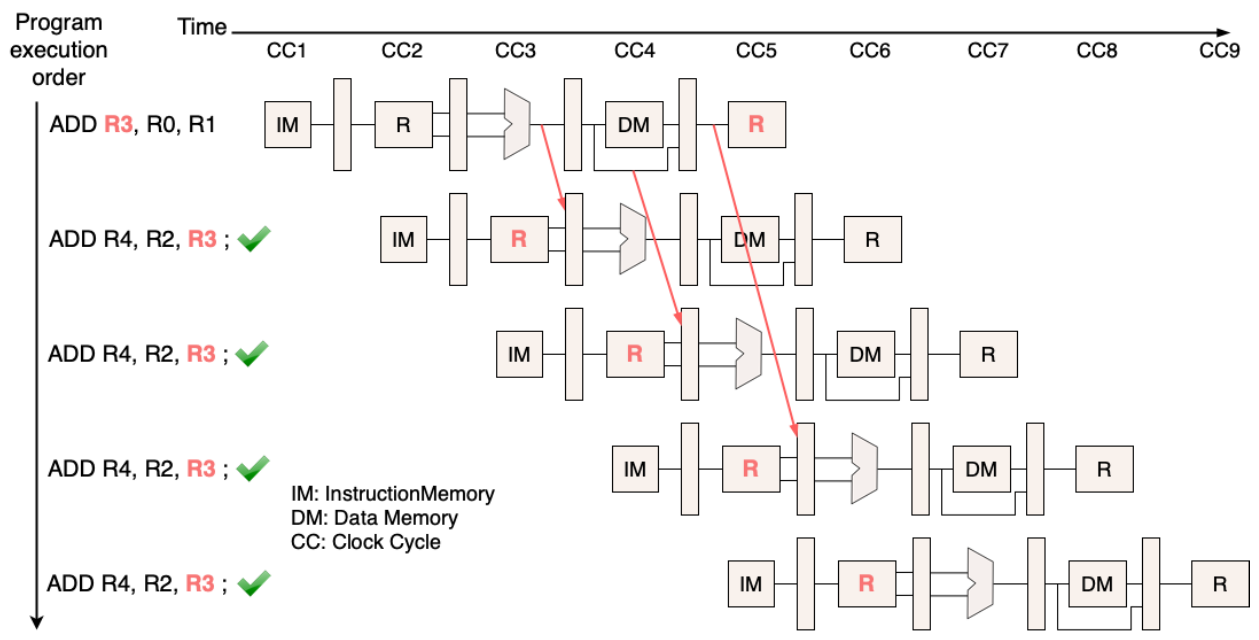


Figure 25. Data Forwarding

An exception to data forwarding is the load instruction. because the load instruction to extraocular data from memory is only completed in the memory stage, and the next instruction has also come to the ex-stage, which does not comply with data forwarding. The way to solve the LOAD adventure is to pause the mechanism, block the pipeline for one cycle, and continue to execute the instructions after LOAD in the next cycle.

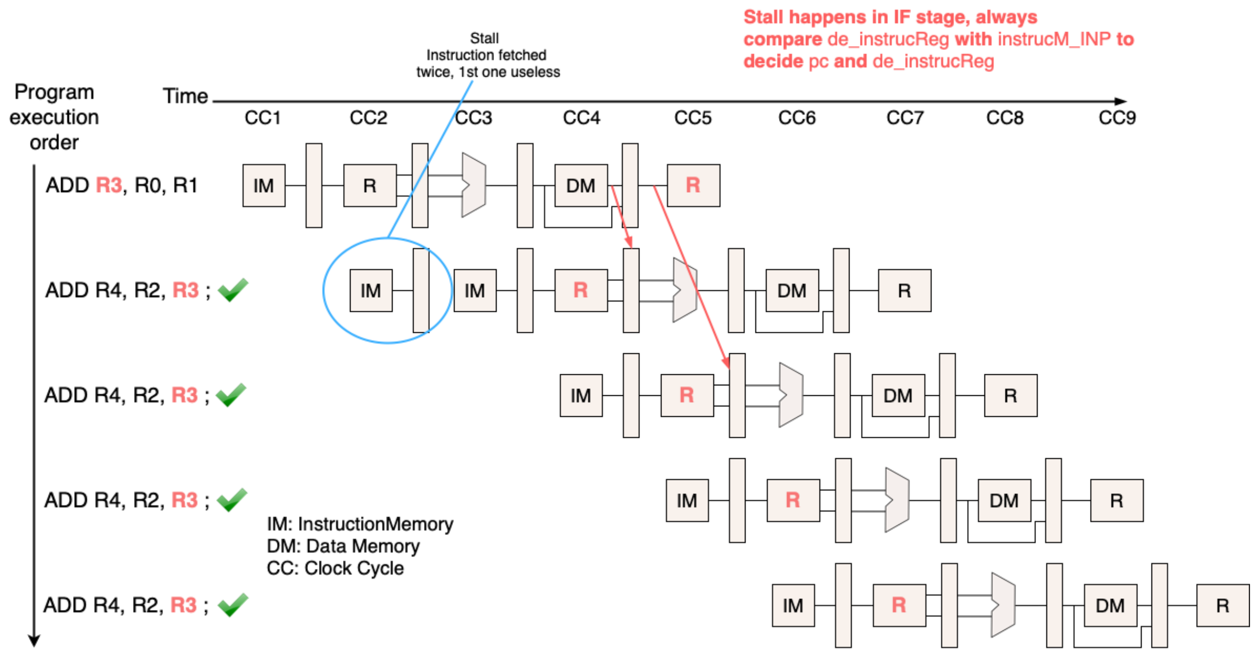


Figure 26. LOAD Hazard Delay Mechanism

LOAD hazard is to detect in the IF stage. If there is currently a LOAD instruction that has entered the ID stage, and the next instruction (that is, the instruction in the IF stage) conflicts with it, it will block the pipeline for a period. The specific method is to Keep the PC value unchanged and insert the NOP instruction at the IF stage.

3.3 Control hazard

It is impossible to determine the hazard caused by the next command. When executing a branch instruction that may change the next instruction, the next instruction cannot be executed until the execution result of this instruction is determined, thus causing control hazards. Control hazard can also be avoided by inserting 3 NOP instructions after the branch instruction, but a more efficient method is to adopt static branch prediction, that is, assuming that the branch instruction is not transferred, the CPU continues to execute the instructions followed by the branch instruction. When the branch instruction is executed to the MEM stage, the result is determined. If the branch needs to be transferred, the empty pipeline at the IF, ID, and EX stages, re-read the instructions at the destination address of the transfer and start to execute sequentially.

A jump flag register can be set up at the MEM stage, and the flag register is valid only if the branch instruction is confirmed to be a jump. If the flag register is valid, it will empty the pipeline IF, ID, and EX stages, to avoid modifying the values of zero, negative, carry flags, 8 general registers, and data memory in the EX, MEM, and WB stages. The specific method is to make an ID, EX, and MEM set as NOP, data write is not enabled, and it is also necessary to ensure that the PC is set as the jump destination address.

Chapter 4: **Simulation Result**

In the test, by inverting the clock every 5 ns, the CPU reverses from the trap every 20 ns, that is, the clock period is 20ns, and the memory clock is set to three clocks, that is, 15 ns, reversed once, through the output of the universal register used and the used Variables in data memory observe the process results of the entire CPU and test whether the CPU works correctly. And observe the change process of the simulation process to test whether the CPU works properly.

4.1 Software simulation

Instruction Memory:

```
initial begin

      RAM[0]  <= {`LOAD, 1'b0, `R1, 1'b0, `R0, 4'b0010}; //load 2 in regA
      RAM[1]  <= {`LOAD, 1'b0, `R2, 1'b0, `R0, 4'b0011}; //load 3 in regA
      RAM[2]  <= {`ADD, 1'b0, `R3, 1'b0, `R1, 1'b0, `R2}; //regC= regB + regA
      RAM[3]  <= {`XOR, 1'b0, `R3, 1'b0, `R1, 1'b0, `R2}; //regC= regB ^ regA
      RAM[4]  <= {`LOAD, 1'b0, `R1, 1'b0, `R0, 4'b0001}; //load 1 in regA
      RAM[5]  <= {`ADD, 1'b0, `R3, 1'b0, `R1, 1'b0, `R2}; //regC= regB + regA
      RAM[6]  <= {`STORE, 1'b0, `R3, 1'b0, `R0, 4'b0010}; //store 2 in regA
      RAM[7]  <= {`ADD, 1'b0, `R6, 1'b0, `R1, 1'b0, `R4}; //regC= regB + regA
      RAM[8]  <= {`AND, 1'b0, `R3, 1'b0, `R2, 1'b0, `R1}; //regC= regB & regA
      RAM[9]  <= {`STORE, 1'b0, `R4, 1'b0, `R0, 4'b0000}; //store 0 in regA
      RAM[10] <= {`SUB, 1'b0, `R3, 1'b0, `R1, 1'b0, `R2}; //regC= regB - regA
      RAM[11] <= {`XOR, 1'b0, `R3, 1'b0, `R1, 1'b0, `R2}; //regC= regA=B ^ regA
      RAM[12] <= {`JUMP,12'b0000_0000_1111};             //jump to addres 15
      RAM[13] <= {`ADD, 1'b0, `R3, 1'b0, `R1, 1'b0, `R2}; // we jumped it
      RAM[14] <= {`LOAD, 1'b0, `R6, 1'b0, `R0, 4'b0000}; // we jumped it
      RAM[15] <= {`LOAD, 1'b0, `R7, 1'b0, `R0, 4'b0001}; // we jumped it
      RAM[16] <= {`SUB, 1'b0, `R3, 1'b0, `R1, 1'b0, `R2}; //regC= regB - regA
      RAM[17] <= {`AND, 1'b0, `R3, 1'b0, `R1, 1'b0, `R2}; //regC= regB & regA
      RAM[18] <= {`HALT, 12'b0000_0000_0000}; //stop
end
```

The following results are shown in hexadecimal.

Instruction 0:	Load the value of 0002 to regA.
Instruction 1:	Load the value of 0003 to regA.
Instruction 2:	Add values from R1 and R2, and the results are saved in R3. R3 should be 0000.
Instruction 3:	XOR values from R1 and R2 and the results are saved in R3. R3 should be FFF8.
Instruction 4:	Load the value of 0001 to regA.
Instruction 5:	Add values from R1 and R2, and the results are saved in R3. R3 should be DECE.
Instruction 6:	Save the value of 0002 to the data memory.
Instruction 7:	Add values from R1 and R2, and the results are saved in R3. R3 should be 0002.
Instruction 8:	AND values from R1 and R2, and the results are saved in R3. R3 should be ABC0.
Instruction 9:	Save the value of 0000 to the data memory.
Instruction 10:	Subtract from R2 and R1, and the result is saved in R3, and R3 should be ABC9.
Instruction 11:	XOR values from R1 and R2 and the results are saved in R3. R3 should be 5439.

Instruction 12:	Jump instruction, PC jumps to instruction 15.
Instruction 13:	This instruction will not be executed.
Instruction 14:	This instruction will not be executed.
Instruction 15:	The instruction is not executed.
Instruction 16:	Subtract from R2 and R1, and the result is saved in R3, and R3 should be 0001.
Instruction 17:	AND values from R1 and R2, and the results are saved in R3. R3 should be ABC9.
Instruction 18:	Termination Detective.

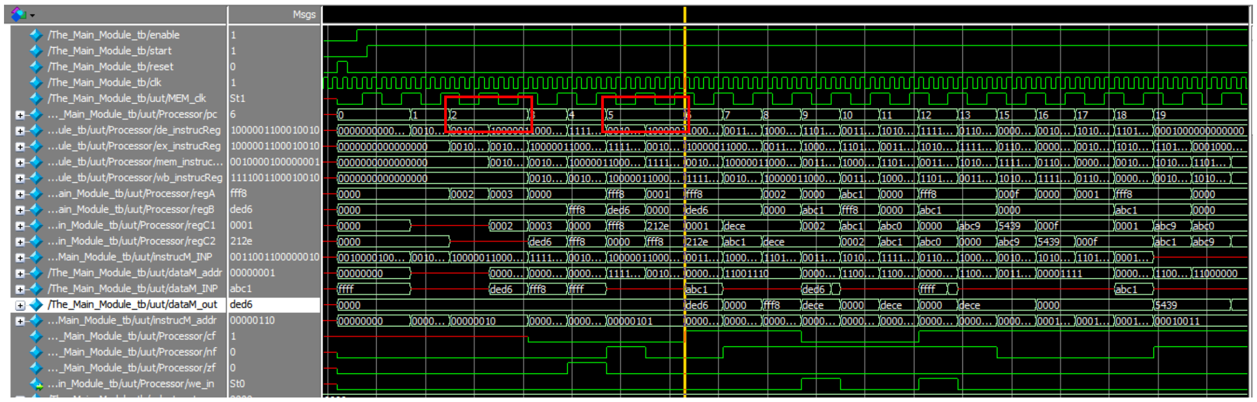
```
#  pc : de_instrucReg :: ex_instrucReg :: mem_instrucReg :: wb_instrucReg :regB :regA :regC1:da:dd :w:regC2: R1 : R2 : R3 :dm0: dm1: dm2:cf:zf:nf
#   x :xxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxx :xxxx :xxxx :xx:xxxx:x:xxxx:xxxx:xxxx:xxxx:ffff:abc1:ded6:x:x:x
#   0 :0000000000000000:0000000000000000:0000000000000000:0000000000000000:0000 :0000 :0000 :00:0000:0:0000:0000:0000:0000:ffff:abc1:ded6:x:0:0
#   1 :0010000100000010:0000000000000000:0000000000000000:0000000000000000:0000 :0000 :xxxx :xx:0000:0:0000:0000:0000:0000:ffff:abc1:ded6:x:0:0
#   2 :0010001000000011:0010000100000010:0000000000000000:0000000000000000:0000 :0002 :xxxx :xx:0000:0:xxxx:0000:0000:0000:ffff:abc1:ded6:x:0:0
#   2 :1000001100010010:0010001000000011:0010000100000010:0000000000000000:0000 :0003 :0002 :02:0000:0:xxxx:0000:0000:0000:ffff:abc1:ded6:x:0:0
#   3 :1000001100010010:1000001100010010:0010001000000011:0010000100000010:0000 :0000 :0003 :03:0000:0:ded6:0000:0000:0000:ffff:abc1:ded6:0:0:0
#   4 :1111001100010010:1000001100010010:1000001100010010:0010001000000011:fff8 :0000 :0000 :00:0000:0:fff8:ded6:0000:0000:ffff:abc1:ded6:0:1:0
#   5 :0010000100000001:1111001100010010:1000001100010010:1000001100010010:ded6 :fff8 :fff8 :f8:0000:0:0000:ded6:fff8:0000:ffff:abc1:ded6:0:0:1
#   5 :1000001100010010:0010000100000001:1111001100010010:1000001100010010:0000 :0001 :212e :2e:0000:0:fff8:ded6:fff8:0000:ffff:abc1:ded6:0:0:0
#   6 :1000001100010010:1000001100010010:0010000100000001:1111001100010010:ded6 :fff8 :0001 :01:ded6:0:212e:ded6:fff8:fff8:ffff:abc1:ded6:1:0:0
#   7 :0011001100000010:1000001100010010:1000001100010010:0010000100000001:ded6 :fff8 :dece :ce:0000:0:abc1:ded6:fff8:212e:ffff:abc1:ded6:1:0:1
#   8 :1000011000010100:0011001100000010:1000001100010010:1000001100010010:0000 :0002 :dece :ce:fff8:0:dece:abc1:fff8:212e:ffff:abc1:ded6:1:0:1
#   9 :1101001100100001:1000011000010100:0011001100000010:1000001100010010:abc1 :0000 :0002 :02:dece:1:dece:abc1:fff8:dece:ffff:abc1:ded6:0:0:1
#   9 :1101001100100001:1000011000010100:0011001100000010:1000001100010010:abc1 :0000 :0002 :02:dece:1:dece:abc1:fff8:dece:ffff:abc1:dece:0:0:1
#  10 :0011010000000000:1101001100100001:1000011000010100:0011001100000010:fff8 :abc1 :abc1 :c1:0000:0:0002:abc1:fff8:dece:ffff:abc1:dece:0:0:1
#  11 :1010001100010010:0011010000000000:1101001100100001:1000011000010100:0000 :0000 :abc0 :c0:dece:0:abc1:abc1:fff8:dece:ffff:abc1:dece:0:0:1
#  12 :1111001100010010:1010001100010010:0011010000000000:1101001100100001:abc1 :fff8 :0000 :00:0000:1:abc0:abc1:fff8:dece:ffff:abc1:dece:1:0:1
#  12 :1111001100010010:1010001100010010:0011010000000000:1101001100100001:abc1 :fff8 :0000 :00:0000:1:abc0:abc1:fff8:dece:0000:abc1:dece:1:0:1
#  13 :0110000000001111:1111001100010010:1010001100010010:0011010000000000:abc1 :fff8 :abc9 :c9:dece:0:0000:abc1:fff8:abc0:0000:abc1:dece:1:0:1
#  15 :0000000000000000:0110000000001111:1111001100010010:1010001100010010:0000 :000f :5439 :39:dece:0:abc9:abc1:fff8:abc0:0000:abc1:dece:1:0:0
#  16 :0010011100000001:0000000000000000:0110000000001111:1111001100010010:0000 :0000 :000f :0f:0000:0:5439:abc1:fff8:abc9:0000:abc1:dece:1:0:0
#  17 :1010001100010010:0010011100000001:0000000000000000:0110000000001111:0000 :0001 :000f :0f:0000:0:000f:abc1:fff8:5439:0000:abc1:dece:1:0:0
#  18 :1101001100010010:1010001100010010:0010011100000001:0000000000000000:abc1 :fff8 :0001 :01:0000:0:000f:abc1:fff8:5439:0000:abc1:dece:1:0:0
#  19 :0001000000000000:1101001100010010:1010001100010010:0010011100000001:abc1 :fff8 :abc9 :c9:5439:0:abc1:abc1:fff8:5439:0000:abc1:dece:1:0:1
#  19 :0001000000000000:0001000000000000:1101001100010010:1010001100010010:0000 :0000 :abc0 :c0:5439:0:abc9:abc1:fff8:5439:0000:abc1:dece:1:0:1
#  19 :0001000000000000:0001000000000000:0001000000000000:1101001100010010:0000 :0000 :abc0 :c0:0000:0:abc0:abc1:fff8:abc9:0000:abc1:dece:1:0:1
#  19 :0001000000000000:0001000000000000:0001000000000000:0001000000000000:0000 :0000 :abc0 :c0:0000:0:abc0:abc1:fff8:abc0:0000:abc1:dece:1:0:1
```

Result Analysis

1. Instructions 0~4, two consecutive LOAD instructions. The value taken by LOAD from Data Memory has not returned to the general register. The ADD and XOR Instructions in the next cycles need to take the corresponding value from the register, so Hazard is generated, but there is no way for Data Forward, so use the Stall method to delay the next instruction for one period. Results R3 got 0000, FFF8.
2. Instructions 5~8, The value taken by LOAD from Data Memory has not returned to the general register. The ADD instruction in the next cycle needs to take the corresponding value from the register, so Hazard is generated, but there is no way for Data Forward. so, use the Stall method to delay the next instruction for one period. Results R3 got DECE.
3. Instruction 9~12, Addition, calculate ABC1 + 0000, and get ABC1 stored in dm1. The SUB instruction subtracts 0000 – 0000 Results reg C got 0000.
4. Instruction 13~15, test jump instructions, and the PC value jump from 13 to 15. Judging from the results.
5. Instruction 16~19, Subtraction, Calculate ABC1 – FFF8 and get ABC9, and the result of AND instruction is ABC0. The Instruction HALT stops the program.

Supplementary explanation:

As a result, two PCs = 2, and two PCs = 5 are caused by the detection mechanism function in ISE (Information Sharing Environment). As long as there is a change in value in output, because the clock frequency of Memory is higher than that of the CPU. Therefore, the change of Memory precedes that in the CPU, and the situation is similar. The actual situation is one PC = 2, and one PCs = 5, of which is the Stall generated by the LOAD instruction, which is correct. Attached is a simulated waveform map:



```
                                                Msgs
/The_Main_Module_tb/enable                      1
/The_Main_Module_tb/start                       1
/The_Main_Module_tb/reset                       0
/The_Main_Module_tb/clk                         1
/The_Main_Module_tb/uut/MEM_clk                 St1
..._Main_Module_tb/uut/Processor/pc             6
...ule_tb/uut/Processor/de_instrucReg           1000001100010010
...ule_tb/uut/Processor/ex_instrucReg           1000001100010010
...ule_tb/uut/Processor/mem_instruc...          0010000100000001
...ule_tb/uut/Processor/wb_instrucReg           1111001100010010
...ain_Module_tb/uut/Processor/regA             fff8
...ain_Module_tb/uut/Processor/regB             ded6
...in_Module_tb/uut/Processor/regC1             0001
...in_Module_tb/uut/Processor/regC2             212e
...Main_Module_tb/uut/instrucM_INP              0011001100000010
/The_Main_Module_tb/uut/dataM_addr              00000001
/The_Main_Module_tb/uut/dataM_INP               abc1
/The_Main_Module_tb/uut/dataM_out               ded6
...Main_Module_tb/uut/instrucM_addr             00000110
..._Main_Module_tb/uut/Processor/cf             1
..._Main_Module_tb/uut/Processor/nf             0
..._Main_Module_tb/uut/Processor/zf             0
...in_Module_tb/uut/Processor/we_in             St0
```

Figure 27. Wave Simulation

4.2 Board-level verification

The 5-stage pipelined Processor functionality is verified by using simulation and FPGA prototyping. The FPGA prototyping can be done after the completion of a valid functional simulatio. The design of 5-stage pipelined processor is targeted for DE2i-150 Cyclone IV for FPGA implementation as shown in Figure 10. The arithemetic operations like addition, subtraction etc is observed using FPGA 7-segment results.

The Pin Assignments where interred as shown in the following tables, in order, and then downloaded to the FPGA board (DE2i-150), and the board begin to work correctly according to our instructions.

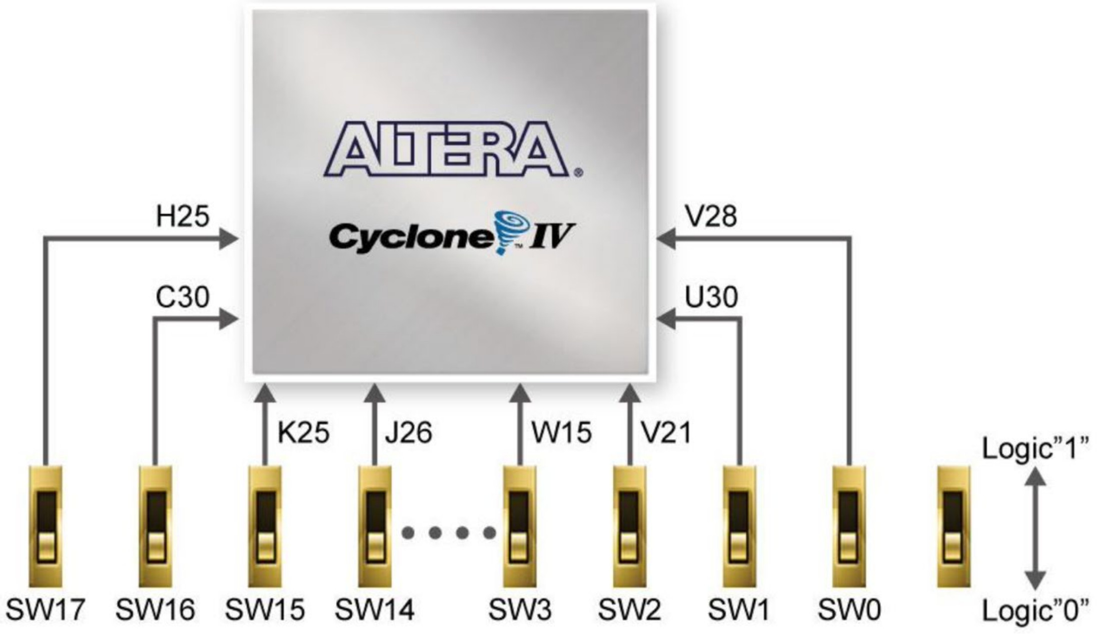


Figure 28. Connections between the slide switches and Cyclone IV GX FPGA

Table 4. Pin Assignments for Slide Switches

Node Name	Direction	Location	I/O Standard
enable	Input	PIN_H25	2.5V
start	Input	PIN_C30	2.5V
select_out[3]	Input	PIN_C2	2.5V
select_out[2]	Input	PIN_V21	2.5V
select_out[1]	Input	PIN_U30	2.5V
select_out[0]	Input	PIN_V28	2.5V

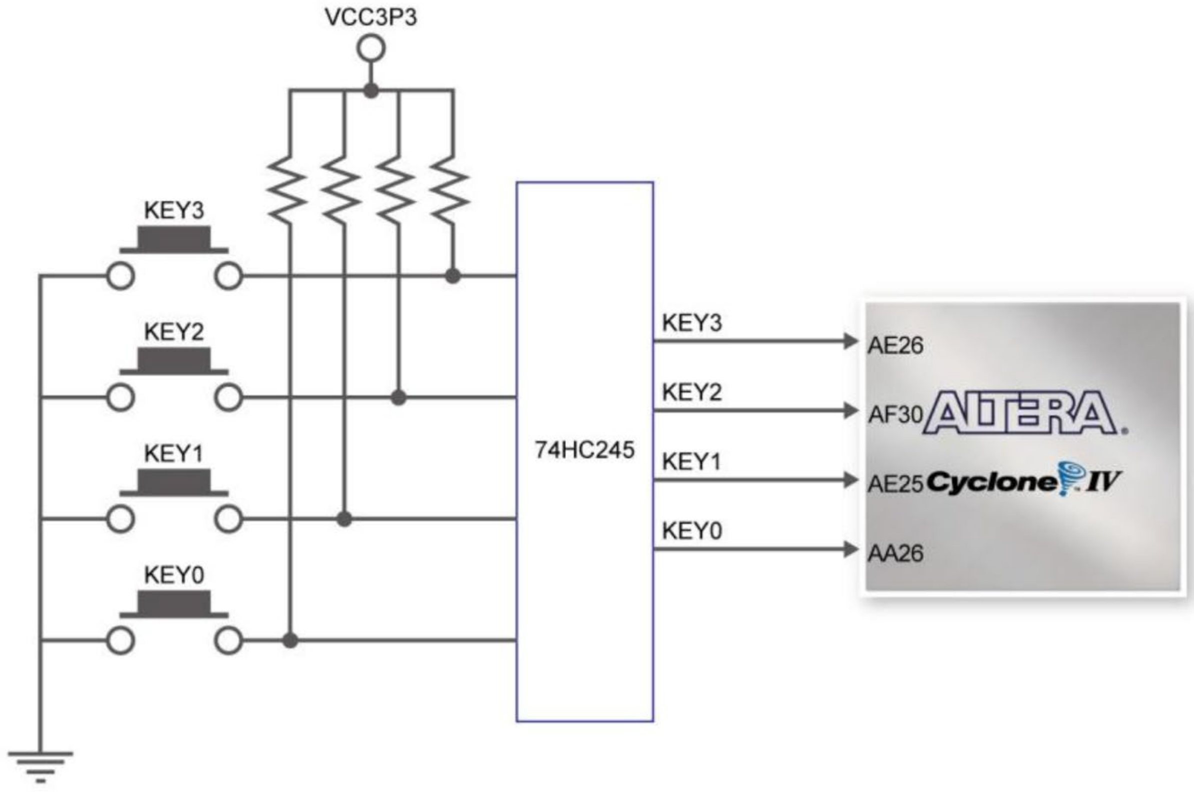


Figure 29. Connections between the push-button and Cyclone IV GX FPGA

Table 5. Pin Assignments for Push-buttons

Node Name	Direction	Location	I/O Standard
reset	Input	PIN_AA26	2.5V
trap	Input	PIN_AE25	2.5V

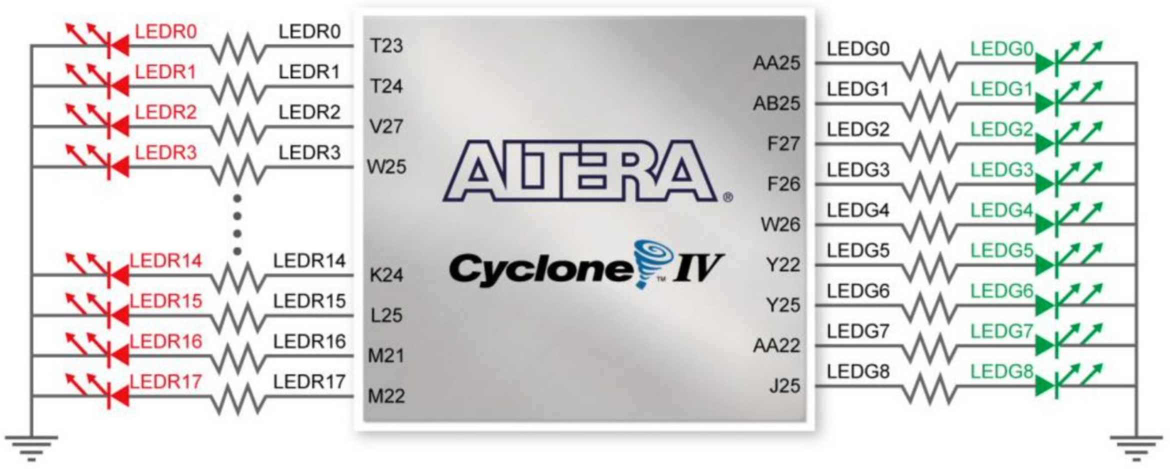


Figure 30. Connections between the LEDs and Cyclone IV GX FPGA

Table 6. Pin Assignments for LEDs

Node Name	Direction	Location	I/O Standard
out[15]	Output	PIN_L25	2.5V
out[14]	Output	PIN_K24	2.5V
out[13]	Output	PIN_M25	2.5V
out[12]	Output	PIN_N21	2.5V
out[11]	Output	PIN_N24	2.5V
out[10]	Output	PIN_P21	2.5V
out[9]	Output	PIN_R24	2.5V
out[8]	Output	PIN_P25	2.5V
out[7]	Output	PIN_T27	2.5V
out[6]	Output	PIN_R24	2.5V
out[5]	Output	PIN_T26	2.5V
out[4]	Output	PIN_T21	2.5V
out[3]	Output	PIN_W25	2.5V
out[2]	Output	PIN_V27	2.5V
out[1]	Output	PIN_T24	2.5V
out[9]	Output	PIN_T23	2.5V

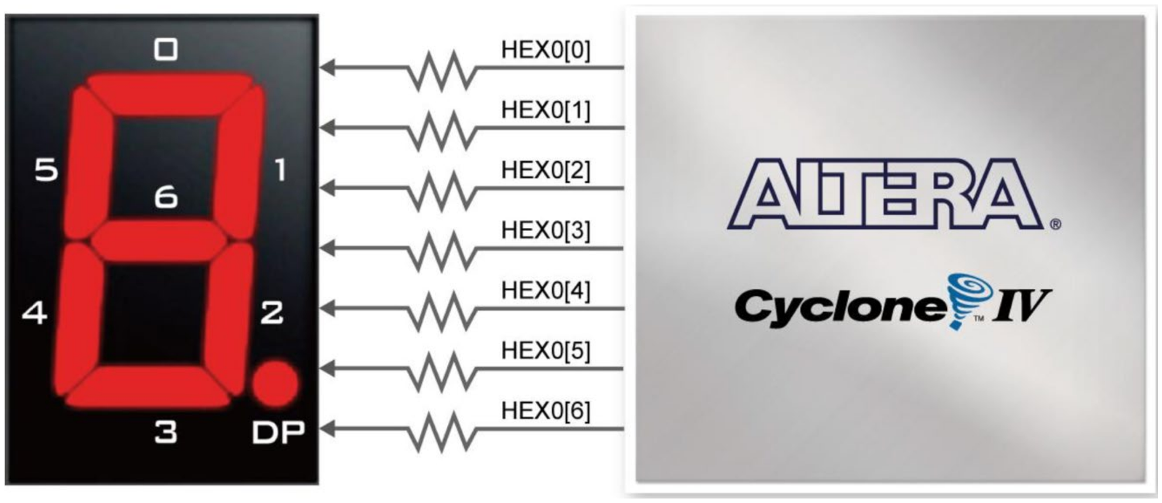


Figure 31. Connections between the 7-segment display HEX0 and Cyclone IV GX FPGA

Table 7. Pin Assignments for 7-segment Displays

Node Name	**Direction**	**Location**	**I/O Standard**
seven_seg_out1[6]	Output	PIN_D4	2.5V
seven_seg_out1[5]	Output	PIN_D5	2.5V
seven_seg_out1[4]	Output	PIN_E3	2.5V
seven_seg_out1[3]	Output	PIN_E4	2.5V
seven_seg_out1[2]	Output	PIN_E6	2.5V
seven_seg_out1[1]	Output	PIN_D7	2.5V
seven_seg_out1[0]	Output	PIN_D10	2.5V
seven_seg_out2[6]	Output	PIN_F10	2.5V
seven_seg_out2[5]	Output	PIN_F4	2.5V
seven_seg_out2[4]	Output	PIN_F6	2.5V
seven_seg_out2[3]	Output	PIN_AG30	2.5V
seven_seg_out2[2]	Output	PIN_F7	2.5V
seven_seg_out2[1]	Output	PIN_G7	2.5V
seven_seg_out2[0]	Output	PIN_G8	2.5V
seven_seg_out3[6]	Output	PIN_G10	2.5V
seven_seg_out3[5]	Output	PIN_J9	2.5V
seven_seg_out3[4]	Output	PIN_G12	2.5V
seven_seg_out3[3]	Output	PIN_F12	2.5V
seven_seg_out3[2]	Output	PIN_G13	2.5V
seven_seg_out3[1]	Output	PIN_B13	2.5V
seven_seg_out3[0]	Output	PIN_G14	2.5V
seven_seg_out4[6]	Output	PIN_F14	2.5V
seven_seg_out4[5]	Output	PIN_D16	2.5V
seven_seg_out4[4]	Output	PIN_F16	2.5V
seven_seg_out4[3]	Output	PIN_F11	2.5V
seven_seg_out4[2]	Output	PIN_G11	2.5V
seven_seg_out4[1]	Output	PIN_E12	2.5V
seven_seg_out4[0]	Output	PIN_E15	2.5V

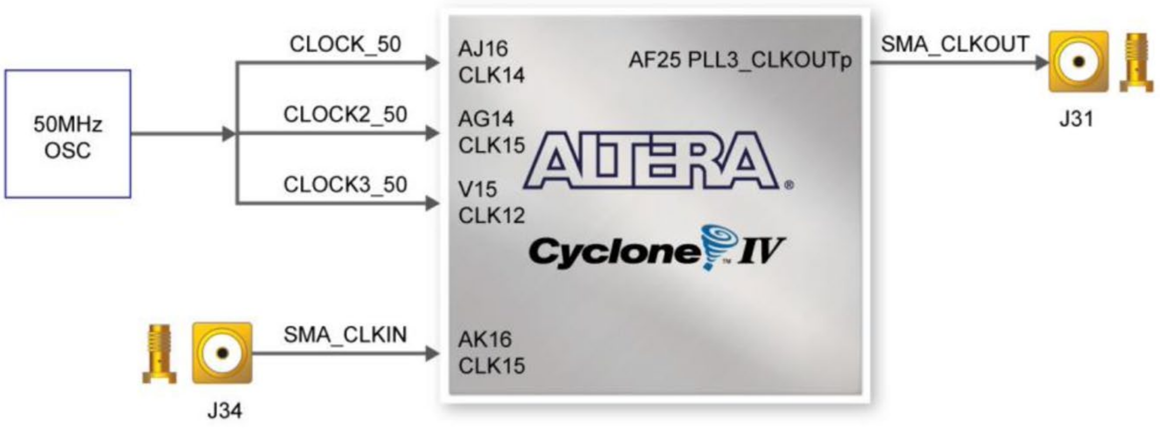


Figure 32. Block diagram of the clock distribution

Table 8. Pin Assignments for Clock Inputs

Node Name	Direction	Location	I/O Standard
clk	Input	PIN_AJ16	2.5V

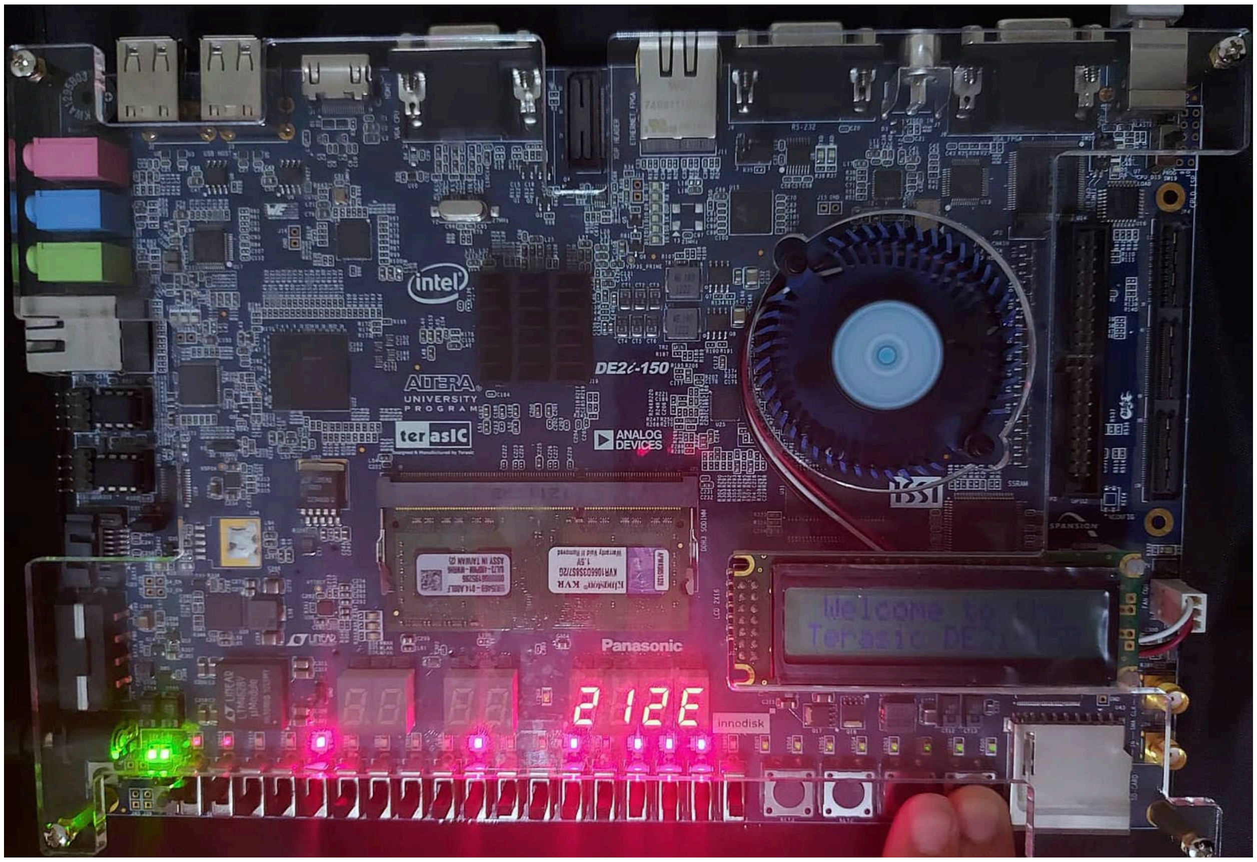


Figure 33. FPGA 7-segment results of 5-stages pipelined processor

Chapter 5: **Conclusion & Future Work**

The aim of this project is to design and simulate a 5-stage Pipelined processor design using Verilog HDL. The pipelined processor performing fixed point integer arithmetic, logical and data movement operation and branch instructions. Each instruction takes five clock cycles for its complete execution. The data hazard is encountered in the pipelined processor and it is eliminated by implementing data forwarding and load-use module. The design is implemented for both single cycle and pipelining concepts. In this, we made our own instruction set and simulated the instructions where the instructions are successfully simulated using Cyclone IV FPGA.

5.1 Achievements:

- Pipeline is implemented in the processor design and fully working.
- All the instructions which are implemented executes successfully.
- The pipelined processor is able to execute fixed point instructions, branch instructions, and load/store instructions.

In the future, the processor can be expanded with implementing more instructions to the instruction set according to our needs, like load/store multiple instructions, load/store byte reverse instructions, system linkage instructions, trap instructions, condition register logical instructions, integer load/store string instructions, branch conditional-count register, branch conditional-link register, TLB Management instructions, processor control instructions, cache management instructions, and synchronization instructions.

References

[1] Osborne, Adam (1980). *An Introduction to microcomputers*. Volume 1: Basic Concepts(2nd ed.). Berkely, California: Osborne-McGraw Hill

[2] https://en.wikipedia.org/wiki/Hazard_(computer_architecture)

[3] Charles Price. MIPS IV Instruction Set[S]. MIPS Technologies, Inc.1995.

[4] Rabaey, J. M., Chandrakasan, A., & Borivoje, N. (2003). Digital Integrated Circuits: A Design Perspective 2nd Edition. Upper Saddle River, NJ: Pearson Education, Inc.

[5] Hennesey, J. L., & Patterson, D. A. (2003). Computer Organization and Design: The hardware/software interface 2nd Edition. San Francisco, CA: Morgan Kaufmann Publishers, Inc.

[6] DE2i-150 FPGA System Manual, ALTERA, 2013.

[7] A. S. Tanenbaum. 2000, „Structured Computer Organization", 4th Edition, Prentice- Hall. Luker, Jarrod D., Prasad, Vinod B.2001,RISC system design in an FPGA", MWSCAS 2001, v2, p532536.

Appendix