Project on Design of Branching-Free Pipelined RISC-V based Processor

GitHub Contributor: Sai Sathvik G B

About RISC-V CPU Architecture:

Computer Architecture introduced a simple approach in how instructions are made. Reduced Instruction Set Computer (RISC) is a CPU design philosophy that emphasizes simplicity and efficiency by using a smaller set of instructions and was pioneered in the 1980s, with notable contributions from IBM and UC Berkeley. RISC uses limited set of simple fixed-length instructions, often executed in single-digit order clock cycles. Memory operations are limited to load and store instructions, while arithmetic and logic operations work only on processor registers. RISC processors heavily utilise pipelining to execute multiple instructions simultaneously, enhancing throughput. Designs often rely on compilers to generate efficient machine code, shifting complexity from hardware to software.

RISC-V compared with other CPU architectures and GPUs:

The simplified instruction set enables faster execution, easier pipelining, low-power consumption makes it ideal for mobile and embedded systems. RISC architectures are easily scalable and support high-performance computing, thereby simplifying hardware, reducing cost and development time. But a problem is RISC requires more sophisticated compilers to optimise code, which is why simple instructions often result in longer programs. ARM (Advanced RISC Machine) is a specific implementation of RISC principles, optimised for low-power and high-efficiency, but RISC follows general-purpose design principles, that's why it's inferred that ARM is a subset of RISC. ARM extends its functionality with fixed-length yet adaptable instructions for specific tasks. RISC is used in academic and startup purposes whereas ARM dominates mobile markets.

Complex Instruction Set Computer (CISC) is another CPU design philosophy which is more complicated. The difference is that RISC has less no. of simplified yet fixed-length instructions, whereas CISC uses large no. of complex variable-length instructions. Due to this, RISC instructions typically execute in single-digit order clock cycles, while CISC instructions vary in execution times. For achieving this, RISC relies on registers for computations, whereas CISC can directly manipulate memory/ use registers/ accumulate into one entity. These qualities make RISC processors simpler, cheaper to design than CISC as well as popular. CPUs with RISC architecture are designed for general-purpose computing, while GPUs don't have such popular architectures and are specialized for parallel processing tasks, so it’s hard to compare them due to their application differences. GPUs are optimised for Single Instruction Multiple Data works and excel in massively parallel operations, due to which their efficiency varies. RISC processors are commonly used in embedded systems, mobile devices and servers, & GPUs are prominent in AI-ML, gaming & scientific computing.

About Pipelining:

Pipelining is fundamental design principle in modern RISC architecture-processors, that enhances instruction throughput by overlapping the execution of multiple instructions. The concept is analogous to an assembly line in industrial manufacturing, where different stages of a task are performed simultaneously. Pipelining is particularly effective due to the simplicity and uniformity of their instruction set, making it easier to divide instruction execution into discrete stages. In a non-pipelined processor, each instruction goes through all phases of execution sequentially, and pipelining addresses this by splitting the execution into stages operating concurrently. By overlapping instruction stages, a RISC processor can complete one instruction per clock cycle, ensuring that all hardware resources are utilised efficiently. The fixed-length, uniform instructions of RISC simplify pipeline design.

Pipelining minimises delays in executing consecutive instructions, enhancing performance. But while pipelining improves performance, it also introduces certain hazards particularly when instructions in the pipeline interfere with one another, leading to stalls or incorrect results. There are three primary types of pipeline hazards, namely Structural (when two or more instructions simultaneously compete for the same resource); Control (arise from branching, when pipeline must decide next instruction); and Data (when an instruction depends on the result of an instruction still in the pipeline) respectively. Data hazards are further categorised into Read After Write (RAW), Write After Write (WAW) and Write After Read (WAR) respectively. Some techniques to mitigate them are forwarding, stalling, branch prediction etc. Increasing the number of pipeline stages can lead to higher throughput but also increases the complexity of hazard management and may lead to increased power consumption. To add to, RISC's reliance on load/store operations can result memory delays.

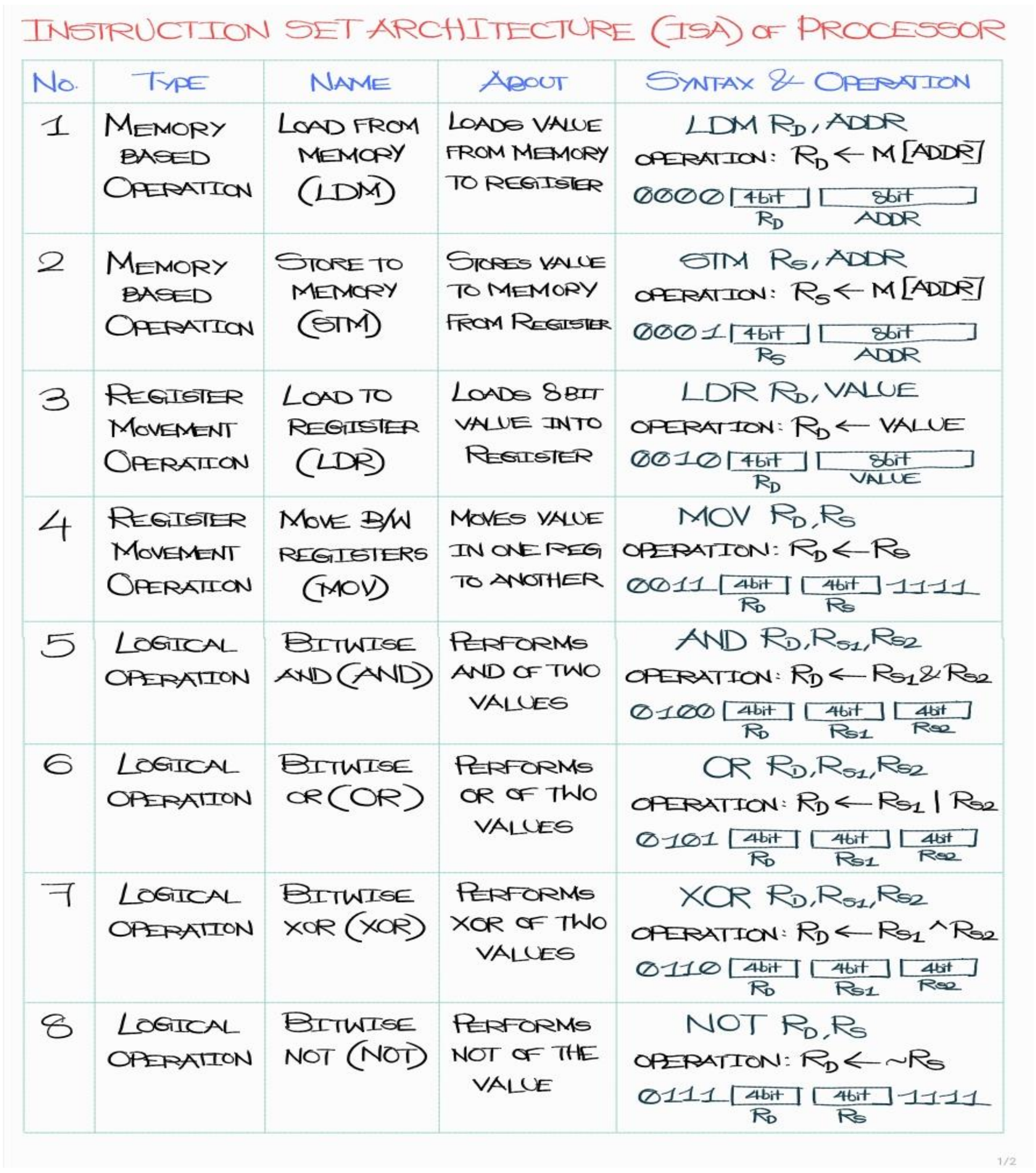
Latency and Throughput are two important timing aspects of general-purpose processors; latency is the time taken for an instruction to be completed i.e., traverse the entire processor, while throughput is the no. of instructions fully executed per unit time. For pipelined processors, latency is slight high but throughput is higher than non-pipelined ones.

About the Assembler:

Assembler is a key component in translating assembly language code into machine code for processor execution and involves the processor's architecture and ISA. Writing in binary or hexadecimal is impractical for humans, so assembly language simplifies this process and assembler is the mediator between low-level language and processor language. Since each processor has unique ISA, it ensures correct encoding for the processor, thereby giving low-level control of the hardware and useful in performance-critical applications. The assembler also checks for syntax errors, undefined symbols or invalid instructions and only allows correct code to enter the processor. The phases of assembler involve: Lexical Analysis (breaking down the assembly code into meaningful units or operands comprising addresses, immediates and conditions); Syntax Analysis (checking if the tokens confirm to the ISA's rules; errors here are invalid instructions and wrong operand types); Instruction Encoding (each low-level instruction is translated into its binary code based on the encoding scheme); and Output Generation (producing an output file in bin/hex/mif format) respectively.

<u>About the Processor designed in this project:</u>

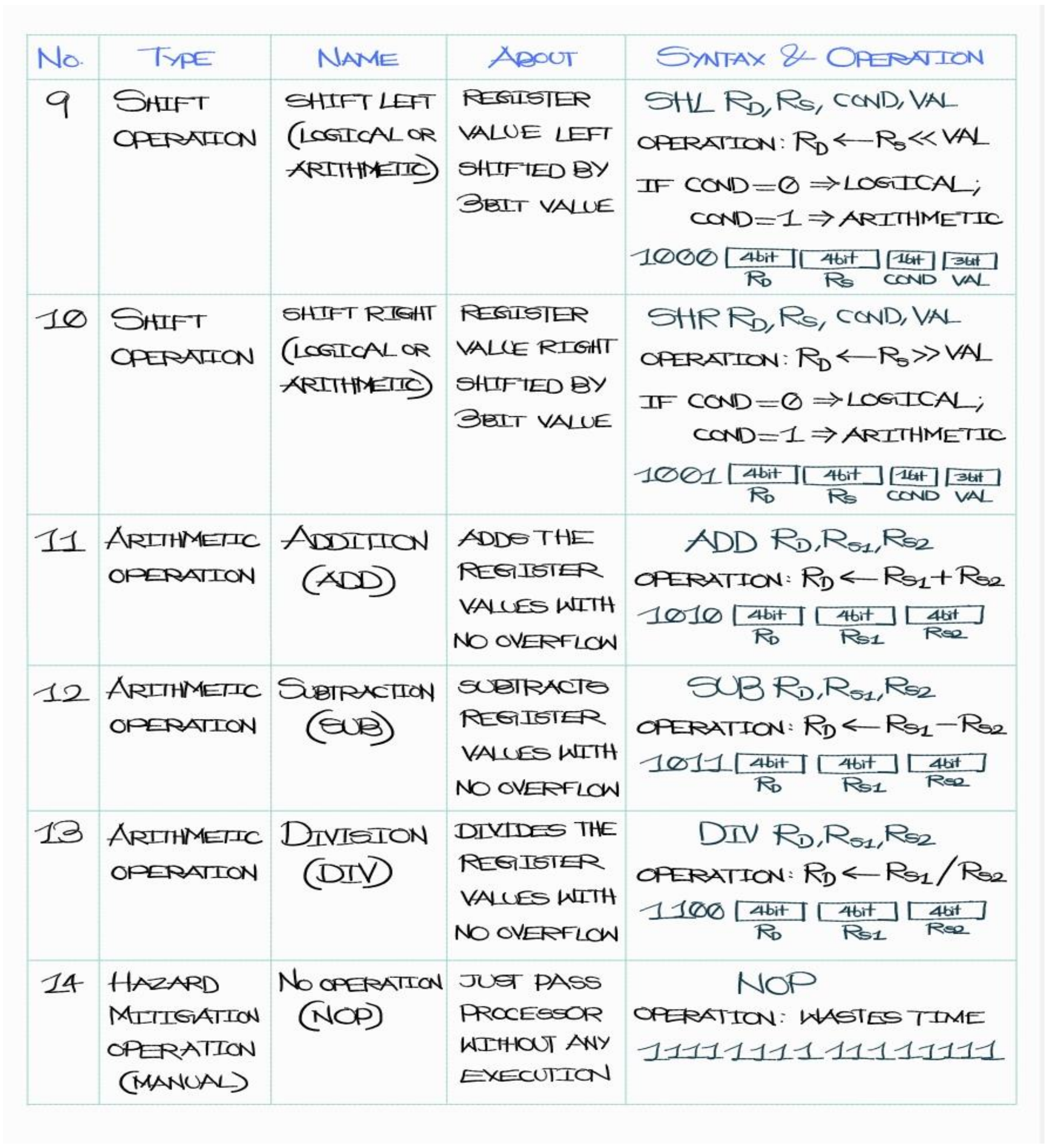
The processor designed has total of 14 Arithmetic, Logical, Shift and Data movement Instructions, with instruction and data memories each of size 0.5kB (8-bit address and 16-bit word) [Harvard Architecture memory]; general purpose register file of size 32 Bytes (4-bit address and 16-bit word) and an Assembler written in Python for converting the Assembly-code into memory initialisation file (mif) format output. The general format of this processor's instruction is 4-bit opcode, followed by register addresses, memory address or conditions for easier ALU operand extraction. Arithmetic and Logical instructions are only-register based, Shift instructions involve arithmetic and logical conditions along with the amount to be shifted as an immediate; Move and Not instructions doesn't care the last 4 bits and just involves the two 4-bit register addresses; finally the manual hazard instruction NOP just passes the processor without disturbing other stages execution thereby making the processor to read the data for next instructions safely. The table below describes all the instructions in the ISA.



INSTRUCTION SET ARCHITECTURE (ISA) OF PROCESSOR

No.	TYPE	NAME	ABOUT	SYNTAX & OPERATION
1	MEMORY BASED OPERATION	LOAD FROM MEMORY (LDM)	LOADS VALUE FROM MEMORY TO REGISTER	LDM R_D, ADDR OPERATION: $R_D \leftarrow M[ADDR]$ $0000\ \underset{R_D}{\boxed{4bit}}\ \underset{ADDR}{\boxed{8bit}}$
2	MEMORY BASED OPERATION	STORE TO MEMORY (STM)	STORES VALUE TO MEMORY FROM REGISTER	STM R_S, ADDR OPERATION: $R_S \leftarrow M[ADDR]$ $0001\ \underset{R_S}{\boxed{4bit}}\ \underset{ADDR}{\boxed{8bit}}$
3	REGISTER MOVEMENT OPERATION	LOAD TO REGISTER (LDR)	LOADS 8BIT VALUE INTO REGISTER	LDR R_D, VALUE OPERATION: $R_D \leftarrow VALUE$ $0010\ \underset{R_D}{\boxed{4bit}}\ \underset{VALUE}{\boxed{8bit}}$
4	REGISTER MOVEMENT OPERATION	MOVE B/W REGISTERS (MOV)	MOVES VALUE IN ONE REG TO ANOTHER	MOV R_D, R_S OPERATION: $R_D \leftarrow R_S$ $0011\ \underset{R_D}{\boxed{4bit}}\ \underset{R_S}{\boxed{4bit}}\ 1111$
5	LOGICAL OPERATION	BITWISE AND (AND)	PERFORMS AND OF TWO VALUES	AND R_D, R_{S1}, R_{S2} OPERATION: $R_D \leftarrow R_{S1} \& R_{S2}$ $0100\ \underset{R_D}{\boxed{4bit}}\ \underset{R_{S1}}{\boxed{4bit}}\ \underset{R_{S2}}{\boxed{4bit}}$
6	LOGICAL OPERATION	BITWISE OR (OR)	PERFORMS OR OF TWO VALUES	OR R_D, R_{S1}, R_{S2} OPERATION: $R_D \leftarrow R_{S1} \mid R_{S2}$ $0101\ \underset{R_D}{\boxed{4bit}}\ \underset{R_{S1}}{\boxed{4bit}}\ \underset{R_{S2}}{\boxed{4bit}}$
7	LOGICAL OPERATION	BITWISE XOR (XOR)	PERFORMS XOR OF TWO VALUES	XOR R_D, R_{S1}, R_{S2} OPERATION: $R_D \leftarrow R_{S1} \wedge R_{S2}$ $0110\ \underset{R_D}{\boxed{4bit}}\ \underset{R_{S1}}{\boxed{4bit}}\ \underset{R_{S2}}{\boxed{4bit}}$
8	LOGICAL OPERATION	BITWISE NOT (NOT)	PERFORMS NOT OF THE VALUE	NOT R_D, R_S OPERATION: $R_D \leftarrow \sim R_S$ $0111\ \underset{R_D}{\boxed{4bit}}\ \underset{R_S}{\boxed{4bit}}\ 1111$

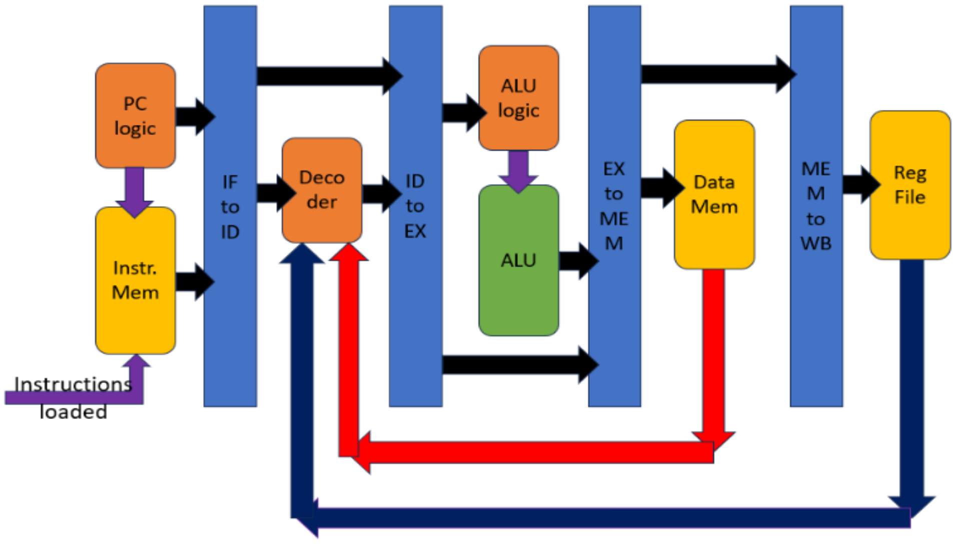
1/2



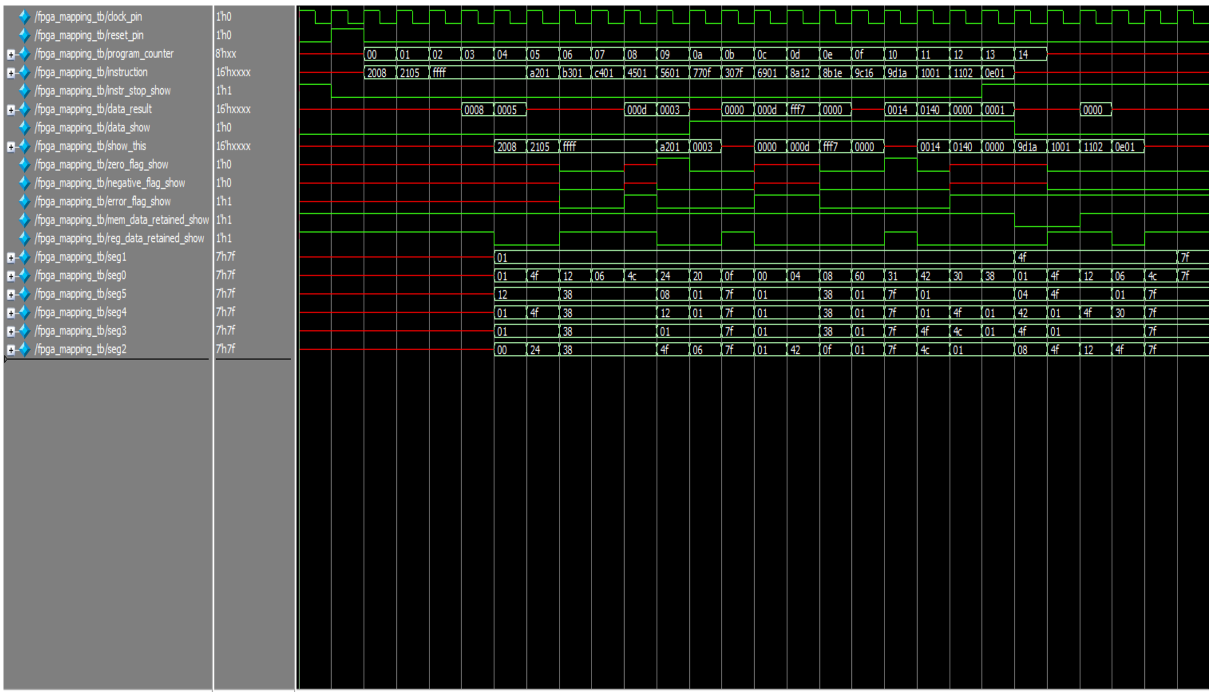
No.	TYPE	NAME	ABOUT	SYNTAX & OPERATION
9	SHIFT OPERATION	SHIFT LEFT (LOGICAL OR ARITHMETIC)	REGISTER VALUE LEFT SHIFTED BY 3BIT VALUE	SHL R_D, R_S, COND, VAL OPERATION: $R_D \leftarrow R_S \ll VAL$ IF COND = 0 ⇒ LOGICAL; COND = 1 ⇒ ARITHMETIC 1000 [4bit R_D] [4bit R_S] [1bit COND] [3bit VAL]
10	SHIFT OPERATION	SHIFT RIGHT (LOGICAL OR ARITHMETIC)	REGISTER VALUE RIGHT SHIFTED BY 3BIT VALUE	SHR R_D, R_S, COND, VAL OPERATION: $R_D \leftarrow R_S \gg VAL$ IF COND = 0 ⇒ LOGICAL; COND = 1 ⇒ ARITHMETIC 1001 [4bit R_D] [4bit R_S] [1bit COND] [3bit VAL]
11	ARITHMETIC OPERATION	ADDITION (ADD)	ADDS THE REGISTER VALUES WITH NO OVERFLOW	ADD R_D, R_{S1}, R_{S2} OPERATION: $R_D \leftarrow R_{S1} + R_{S2}$ 1010 [4bit R_D] [4bit R_{S1}] [4bit R_{S2}]
12	ARITHMETIC OPERATION	SUBTRACTION (SUB)	SUBTRACTS REGISTER VALUES WITH NO OVERFLOW	SUB R_D, R_{S1}, R_{S2} OPERATION: $R_D \leftarrow R_{S1} - R_{S2}$ 1011 [4bit R_D] [4bit R_{S1}] [4bit R_{S2}]
13	ARITHMETIC OPERATION	DIVISION (DIV)	DIVIDES THE REGISTER VALUES WITH NO OVERFLOW	DIV R_D, R_{S1}, R_{S2} OPERATION: $R_D \leftarrow R_{S1} / R_{S2}$ 1100 [4bit R_D] [4bit R_{S1}] [4bit R_{S2}]
14	HAZARD MITIGATION OPERATION (MANUAL)	NO OPERATION (NOP)	JUST PASS PROCESSOR WITHOUT ANY EXECUTION	NOP OPERATION: WASTES TIME 11111111 11111111

The 5 pipelining stages of this processor are Instruction Fetch (IF) (the processor fetches the instruction from memory using program counter); Instruction Decode (ID) (the fetched instruction is decoded to identify the operation to be performed and reading the necessary registers and/or memory); Execute (EX) (the operation specified by the instruction is performed); Memory Access (MEM) (if the instruction involves memory, this stage handles writing into it whereas reading is done in the earlier stage); and Register Write Back (WB) (the result of the execution is written back to the register file whereas reading is done in the earlier stage) respectively. The picture shows processor flow implemented in this project.

Along with these, a new file is written to map the flags to the LEDs of the FPGA board, along with Data/Instruction & Reset to the switches; the 8-bit Program Counter was directly displayed on two of the six seven-segment displays of the FPGA, the remaining four were used to show the instruction being executed or corresponding data being processed, based on the value of Data/Instruction switch. The following pictures depict the simulation waveform, the pin mapping for the DE10 FPGA and the pins I assigned for implementation with the results.



Processor Flow (Reading is instantaneous within the 2nd stage but not Writing)

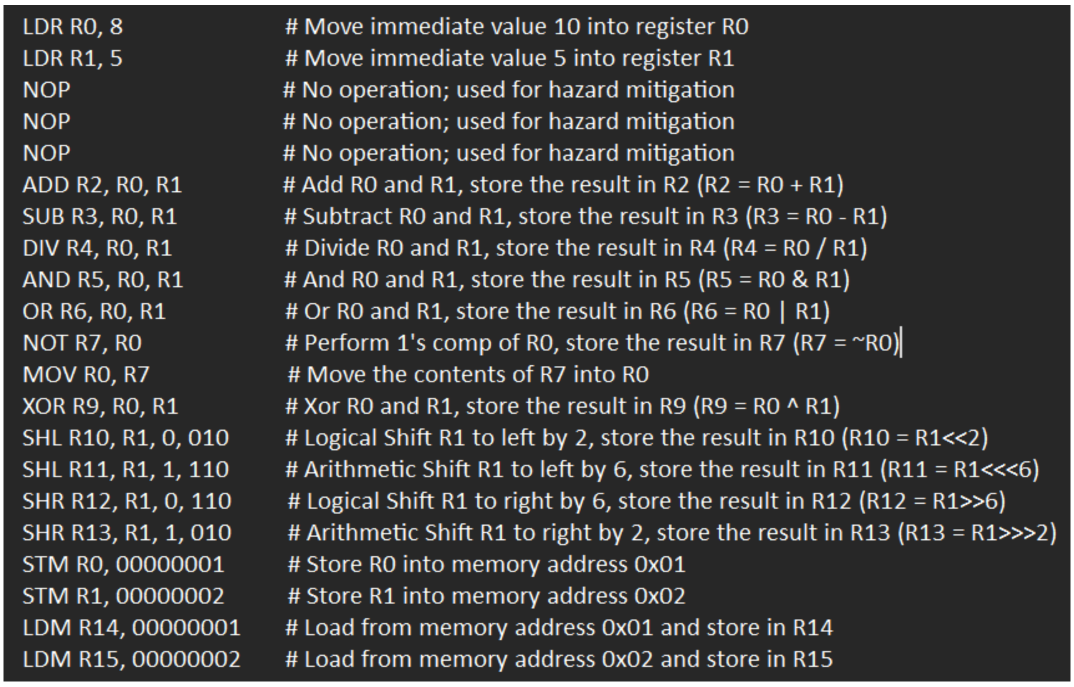

/fpga_mapping_tb/clock_pin 1'h0
/fpga_mapping_tb/reset_pin 1'h0
/fpga_mapping_tb/program_counter 8'hxx
/fpga_mapping_tb/instruction 16'hxxxx
/fpga_mapping_tb/instr_stop_show 1'h1
/fpga_mapping_tb/data_result 16'hxxxx
/fpga_mapping_tb/data_show 1'h0
/fpga_mapping_tb/show_this 16'hxxxx
/fpga_mapping_tb/zero_flag_show 1'h0
/fpga_mapping_tb/negative_flag_show 1'h0
/fpga_mapping_tb/error_flag_show 1'h1
/fpga_mapping_tb/mem_data_retained_show 1'h1
/fpga_mapping_tb/reg_data_retained_show 1'h1
/fpga_mapping_tb/seg1 7'h7f
/fpga_mapping_tb/seg0 7'h7f
/fpga_mapping_tb/seg5 7'h7f
/fpga_mapping_tb/seg4 7'h7f
/fpga_mapping_tb/seg3 7'h7f
/fpga_mapping_tb/seg2 7'h7f

Simulation Waveform as obtained on Questa Sim(redirected from Quartus Prime; if possible, other simulation software can be chosen while creating the project)

<u>Explanation of the Simulation Waveform:</u>

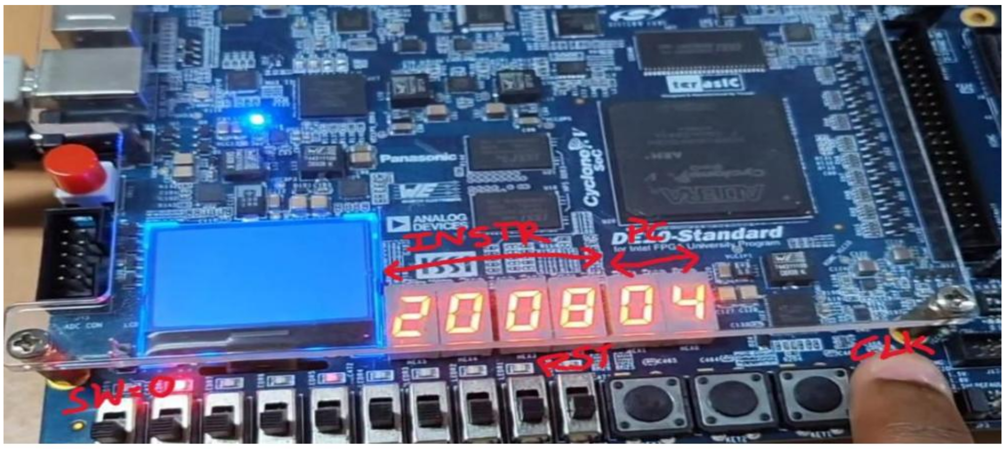
The program_counter into the instruction memory gives out the instruction to be executed, which gives out the data_result; now when data_show is HIGH the data_result is shown instead of the instruction; the error_flag, zero_flag, negative_flag are the results from the ALU after the operation is performed, while instr_stop is that signal from instruction memory which indicates there are no more instructions. The seven segments seg1 & seg0 display the program counter shifted by 4 cycles and the remaining seg5, seg4, seg3, seg2 are to display the instruction (shifted by 4 cycles) or result (unshifted) based on data_show.

Picture of the Assembly code implemented (this becomes the mif file):

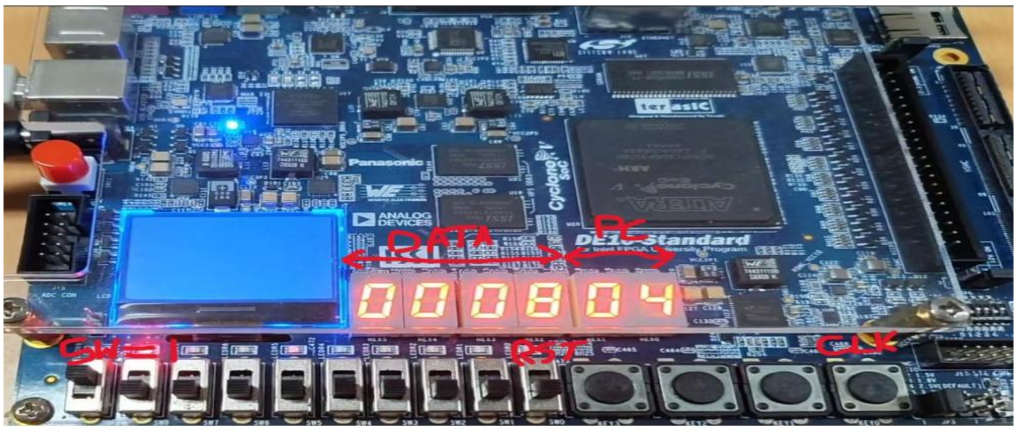


```
LDR R0, 8                # Move immediate value 10 into register R0
LDR R1, 5                # Move immediate value 5 into register R1
NOP                      # No operation; used for hazard mitigation
NOP                      # No operation; used for hazard mitigation
NOP                      # No operation; used for hazard mitigation
ADD R2, R0, R1           # Add R0 and R1, store the result in R2 (R2 = R0 + R1)
SUB R3, R0, R1           # Subtract R0 and R1, store the result in R3 (R3 = R0 - R1)
DIV R4, R0, R1           # Divide R0 and R1, store the result in R4 (R4 = R0 / R1)
AND R5, R0, R1           # And R0 and R1, store the result in R5 (R5 = R0 & R1)
OR R6, R0, R1            # Or R0 and R1, store the result in R6 (R6 = R0 | R1)
NOT R7, R0               # Perform 1's comp of R0, store the result in R7 (R7 = ~R0)
MOV R0, R7               # Move the contents of R7 into R0
XOR R9, R0, R1           # Xor R0 and R1, store the result in R9 (R9 = R0 ^ R1)
SHL R10, R1, 0, 010      # Logical Shift R1 to left by 2, store the result in R10 (R10 = R1<<2)
SHL R11, R1, 1, 110      # Arithmetic Shift R1 to left by 6, store the result in R11 (R11 = R1<<<6)
SHR R12, R1, 0, 110      # Logical Shift R1 to right by 6, store the result in R12 (R12 = R1>>6)
SHR R13, R1, 1, 010      # Arithmetic Shift R1 to right by 2, store the result in R13 (R13 = R1>>>2)
STM R0, 00000001         # Store R0 into memory address 0x01
STM R1, 00000002         # Store R1 into memory address 0x02
LDM R14, 00000001        # Load from memory address 0x01 and store in R14
LDM R15, 00000002        # Load from memory address 0x02 and store in R15
```

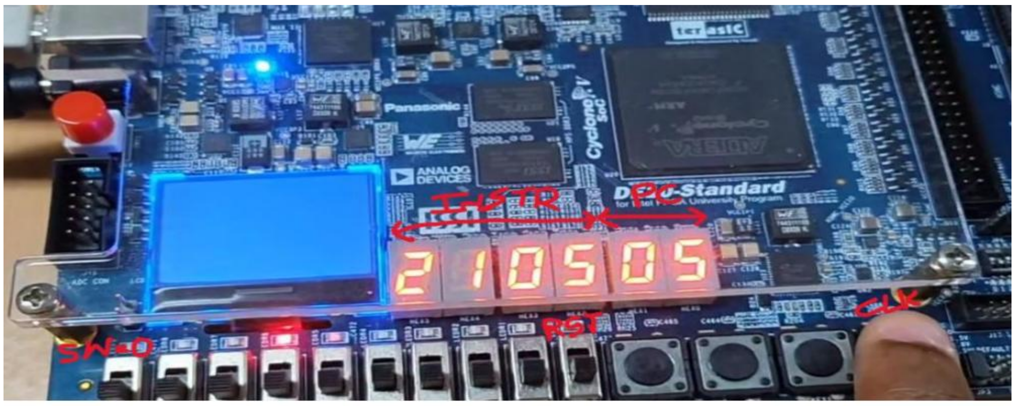
Pictures of Implementation on the DE10 FPGA board (PC unsynced here):



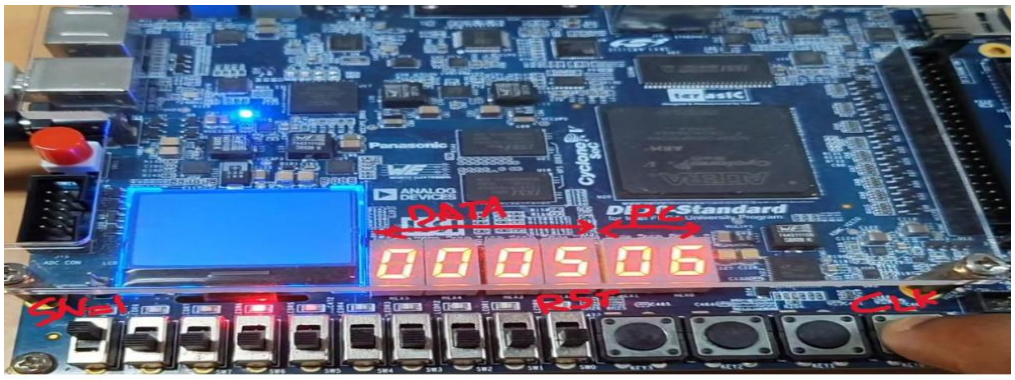
Instruction 2008 shown for Switch = 0



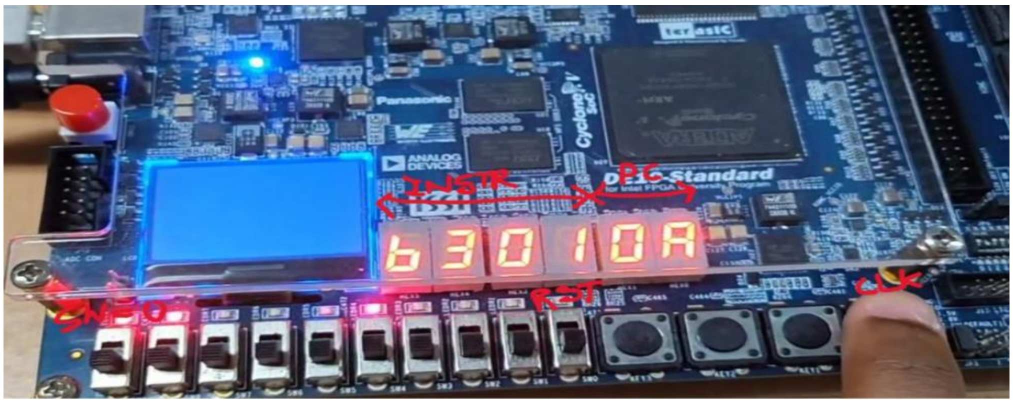
Data 0008 shown for Switch = 1



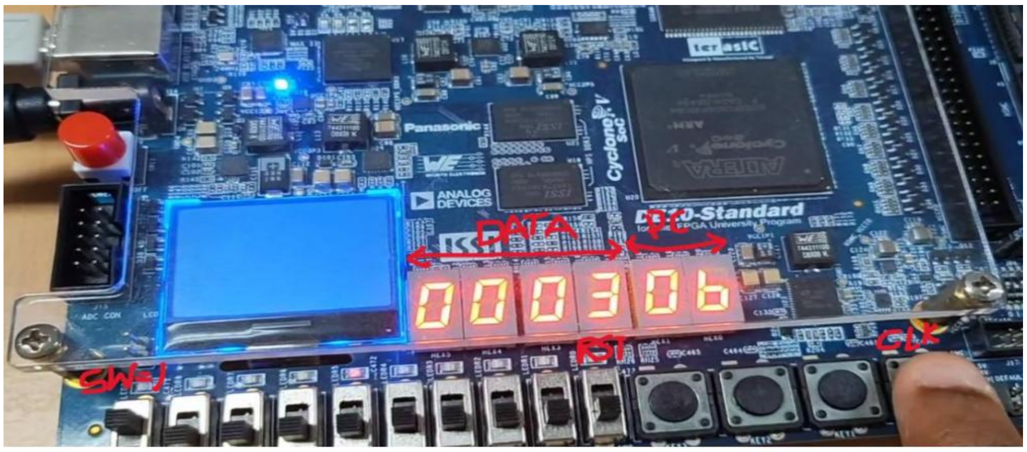
Instruction 2105 shown for Switch = 0



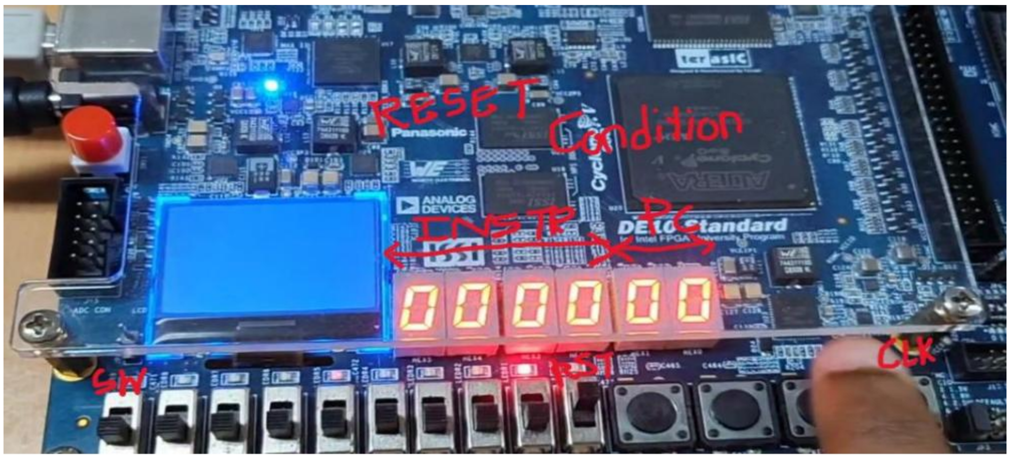
Data 0005 shown for Switch = 1



Instruction B301 shown for Switch = 0



Data 0003 shown for Switch = 1



Initial Condition (Reset = 1) will reset the PC

Pin Assignment for this processor on the DE10 FPGA board:

These are the pin mapping done for implementing this design on the FPGA; the clock is push-button based and isn't built-in due to the fact that clock dividers have to be too large to visualise the working of the processor on the FPGA (dividing the 50MHz clock by 18 will give clock period of approx. 5sec, which itself is too large to synthesise). Additionally the value xxxx is seen as 0 on the seven-segments, this is why the reset starts with 00 as PC.

Type	FPGA Output	Pin	I/O Type on board
Inputs	clock	PIN_AJ4	Pushbutton
	reset	PIN_AB30	Switches
	data_show	PIN_AA30	
1-bit Outputs	error_flag_show	PIN_AD24	LEDs
	zero_flag_show	PIN_AB23	
	negative_flag_show	PIN_AC23	
	mem_data_retained_show	PIN_AG25	
	reg_data_retained_show	PIN_AF25	
	instr_stop_show	PIN_AA24	

Type	FPGA Output	Pin	Assigned values
7-segment[1]	seg1[0]	PIN_AF16	final_program_counter[7:4]
	seg1[1]	PIN_V16	
	seg1[2]	PIN_AE16	
	seg1[3]	PIN_AD17	
	seg1[4]	PIN_AE18	
	seg1[5]	PIN_AE17	
	seg1[6]	PIN_V17	
7-segment[0]	seg0[0]	PIN_W17	final_program_counter[3:0]
	seg0[1]	PIN_V18	
	seg0[2]	PIN_AG17	
	seg0[3]	PIN_AG16	
	seg0[4]	PIN_AH17	
	seg0[5]	PIN_AG18	
	seg0[6]	PIN_AH18	

Type	FPGA Output	Pin	Assigned values
7-segment[5]	seg5[0]	PIN_AF21	display_value[15:12]
	seg5[1]	PIN_AG21	
	seg5[2]	PIN_AF20	
	seg5[3]	PIN_AG20	
	seg5[4]	PIN_AE19	
	seg5[5]	PIN_AF19	
	seg5[6]	PIN_AB21	
7-segment[4]	seg4[0]	PIN_AD21	display_value[11:8]
	seg4[1]	PIN_AG22	
	seg4[2]	PIN_AE22	
	seg4[3]	PIN_AE23	
	seg4[4]	PIN_AG23	
	seg4[5]	PIN_AF23	
	seg4[6]	PIN_AH22	
7-segment[3]	seg3[0]	PIN_Y19	display_value[7:4]
	seg3[1]	PIN_W19	
	seg3[2]	PIN_AD19	
	seg3[3]	PIN_AA20	
	seg3[4]	PIN_AC20	
	seg3[5]	PIN_AA19	
	seg3[6]	PIN_AD20	
7-segment[2]	seg2[0]	PIN_AA21	display_value[3:0]
	seg2[1]	PIN_AB17	
	seg2[2]	PIN_AA18	
	seg2[3]	PIN_Y17	
	seg2[4]	PIN_Y18	
	seg2[5]	PIN_AF18	
	seg2[6]	PIN_W16	

Additional Details:

For implementing the in-built clock on the FPGA to the processor, make sure the clock divider is at least designed for 2sec clock period (close to 17 or 18) to easily visualise what's going on; the below pin mapping can be used in this case.

Table 3-5 Pin Assignment of Clock Inputs

Signal Name	*FPGA Pin No.*	*Description*	*I/O Standard*
CLOCK_50	PIN_AF14	50 MHz clock input	3.3V
CLOCK2_50	PIN_AA16	50 MHz clock input	3.3V
CLOCK3_50	PIN_Y26	50 MHz clock input	3.3V
CLOCK4_50	PIN_K14	50 MHz clock input	3.3V
HPS_CLOCK1_25	PIN_D25	25 MHz clock input	3.3V
HPS_CLOCK2_25	PIN_F25	25 MHz clock input	3.3V

Note for the users:

Please don't change any other files after cloning this repository other than the files: 1_assembly_program.asm & the generated 3_instr_memory_values.mif, for the best results to be seen on the FPGA. You are free to suggest changes (provide the improved results if arrived to and then contact me); the steps to execute the following project are available in the Description file so please go through it as well.