

**Department of Electronic and Telecommunication  
Engineering  
University of Moratuwa**

**EN3021 – Digital System Design**



**Non-pipelined Single Stage (Cycle) CPU Design  
Report**

Name: Kavinda W.M.C.

Index Number: 200301D

**16<sup>th</sup> of October 2023**

## Table of Contents

1. Introduction .....	3
2. Implementation.....	3
2.1. Program Counter (PC).....	4
2.2. Instruction Memory.....	4
2.3. Register File .....	4
2.4. ALU (Arithmetic Logic Unit) .....	4
2.5. Sign Extender (Imm_Gen) .....	4
2.6. Data Memory.....	5
2.7. Muxes.....	5
2.8. Branch Prediction.....	5
2.9. Control Unit.....	5
2.10. Data Flow .....	5
3. Microprogramming Approach .....	6
4. Implemented Instructions .....	8
4.1. R-Type Instructions .....	8
4.2. I-Type Computational Instructions.....	9
4.3. I-Type Load Instructions .....	10
4.4. S-Type Instructions .....	11
4.5. SB-Type Instructions.....	12
4.6. MUL Instruction.....	13
5. Resource Utilization .....	14
5.1. Slice Logic.....	14
5.1.1. Summary of Registers by Type.....	14
5.2. Memory .....	14
5.3. DSP.....	15
5.4. IO and GT Specific .....	15
5.5. Clocking .....	15
5.6. Specific Feature.....	15
5.7. Primitives .....	16
6. Conclusion.....	16

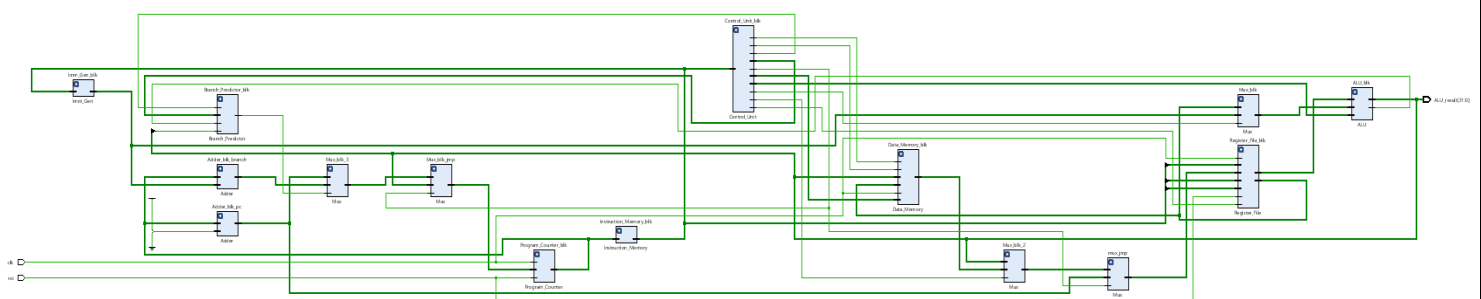
# 1. Introduction

This project focused on the design and construction of a 32-bit non-pipelined RISC-V processor on an FPGA platform, with a primary emphasis on understanding computer architecture and instruction execution within the RV32I instruction set. It aimed to create practical, hands-on experience in processor design and operation. The central objectives included the development of a fully functional RISC-V processor capable of executing essential instruction types, such as R, I, S, and SB instructions. In addition, the project involved the incorporation of a new instruction, MUL, to expand the processor's capabilities. Rigorous testing was carried out through RTL simulations and FPGA-based real-world testing to ensure accurate execution and to verify the reliability and high performance of the implemented processor.

The architecture of the RISC-V processor adopted a 3-bus structure to efficiently manage data flow. Modifications were made to the Arithmetic Logic Unit (ALU) and its controller to accommodate the new MUL instruction. This architectural approach aimed to optimize instruction execution time and ensure precise computation. The successful implementation of the MUL instruction showcased the processor's adaptability and efficiency in handling a broader range of computational tasks. The project's comprehensive testing strategy affirmed the processor's practical functionality, further emphasizing its importance in the realm of computer architecture.

## 2. Implementation

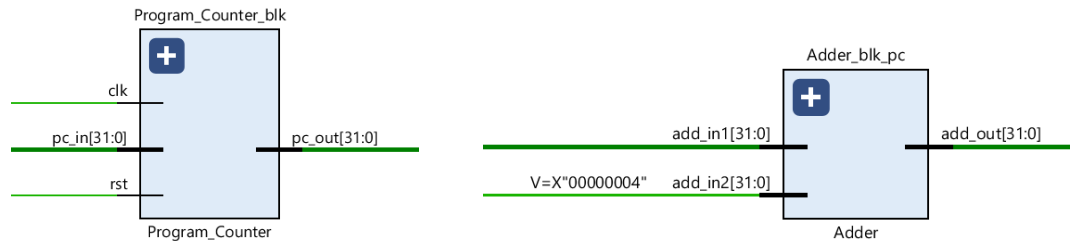
The implemented Datapath is a critical component of a processor architecture designed to execute a set of instructions in a systematic manner. It serves as the core of the processor, orchestrating the flow of data and control signals to execute instructions. Below is an overview of the key components and how they interact within the Datapath.



### Schematic

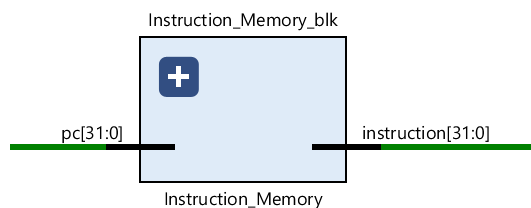
## 2.1. Program Counter (PC)

The program counter is responsible for keeping track of the memory address of the next instruction to be fetched. It is updated by adding 4 for sequential instruction execution or by branching to a new address for control flow instructions.



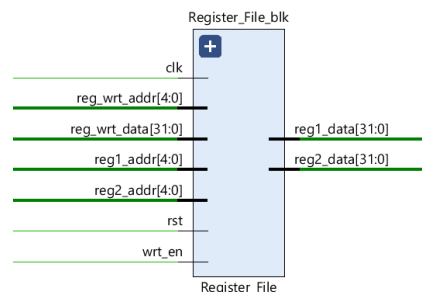
## 2.2. Instruction Memory

The instruction memory module is responsible for fetching instructions based on the current program counter value. It provides instructions to the processor for decoding and execution.



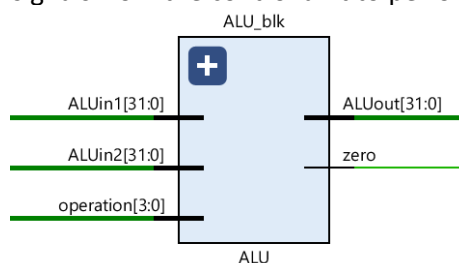
## 2.3. Register File

The register file consists of a set of registers used for storing data. The datapath reads data from these registers based on the instruction's register addresses and can also write data back to them if required.



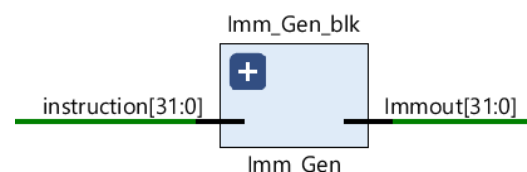
## 2.4. ALU (Arithmetic Logic Unit)

The ALU performs arithmetic and logic operations on data. It takes inputs from the register file, immediate values from instruction, and control signals from the control unit to perform operations such as addition, subtraction, logical operations, etc.



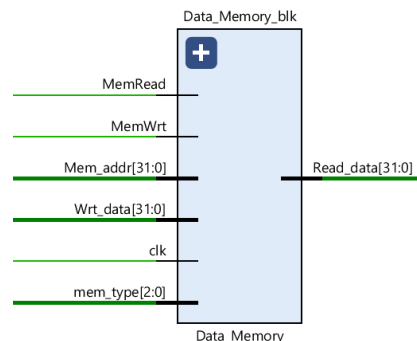
## 2.5. Sign Extender (Imm\_Gen)

The sign extender is responsible for generating properly signed immediate values for instructions. It extends the immediate value based on the instruction type, ensuring correct data processing in ALU operations.



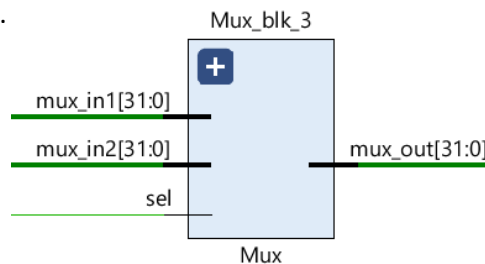
## 2.6. Data Memory

The data memory module is used for read and write operations to data memory. It supports various memory access types like byte (lb/lbu), half-word (lh/lhu), and word (lw) reads, and byte (sb), half-word (sh), and word (sw) writes.



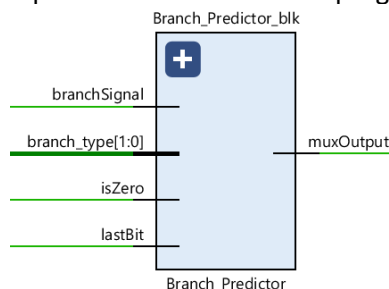
## 2.7. Muxes

Multiplexers are used to select data sources and control paths within the datapath. They allow for flexibility in routing data to different components.



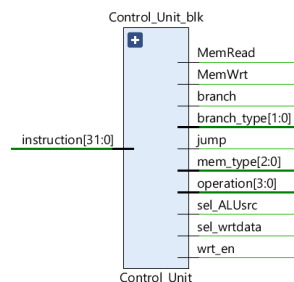
## 2.8. Branch Prediction

The branch predictor helps improve the instruction fetching process by predicting whether a branch instruction will be taken or not. This prediction influences the program counter.



## 2.9. Control Unit

The control unit interprets the instruction fetched from memory and generates control signals. These control signals are used to configure and control the various Datapath components, ensuring the correct execution of the instruction.



## 2.10. Data Flow

Data flows through the datapath according to the fetched instruction and the control signals generated by the control unit. The control unit's signals dictate which components are active and how data is processed.

### 3. Microprogramming Approach

The processor follows a microprogramming approach. This approach involves using a Control Unit to decode each instruction and generate control signals for the various components of the processor. The Control Unit takes into consideration the instruction's opcode and other relevant fields to determine how the instruction should be executed.

TYPE	Instruction	Opcode	Func3	Func7	ALU operation	wrt en	jump	sel ALUsrc	Mem Read	Mem Wrt	sel wrtdata	mem type	branch	branch type
R	ADD	0110011	000	0000000	0000	1	0	0	0	0	0	000	0	00
	SUB	0110011	000	0100000	0001	1	0	0	0	0	0	000	0	00
	SLL	0110011	001	0000000	0101	1	0	0	0	0	0	000	0	00
	SLT	0110011	010	0000000	1000	1	0	0	0	0	0	000	0	00
	SLTU	0110011	011	0000000	1001	1	0	0	0	0	0	000	0	00
	XOR	0110011	100	0000000	0100	1	0	0	0	0	0	000	0	00
	SRL	0110011	101	0000000	0110	1	0	0	0	0	0	000	0	00
	SRA	0110011	101	0100000	0111	1	0	0	0	0	0	000	0	00
	OR	0110011	110	0000000	0011	1	0	0	0	0	0	000	0	00
	AND	0110011	111	0000000	0010	1	0	0	0	0	0	000	0	00
	MUL	0110011	001	0100000	1010	1	0	0	0	0	0	000	0	00
I	ADDI	0010011	000	0000000	0000	1	0	1	0	0	0	000	0	00
	SLTI	0010011	010	0000000	1000	1	0	1	0	0	0	000	0	00
	SLTIU	0010011	011	0000000	1001	1	0	1	0	0	0	000	0	00
	XORI	0010011	100	0000000	0100	1	0	1	0	0	0	000	0	00
	ORI	0010011	110	0000000	0011	1	0	1	0	0	0	000	0	00
	ANDI	0010011	111	0000000	0010	1	0	1	0	0	0	000	0	00
	SLLI	0010011	001	0000000	0101	1	0	1	0	0	0	000	0	00
	SRLI	0010011	101	0000000	0110	1	0	1	0	0	0	000	0	00
	SRAI	0010011	101	0100000	0111	1	0	1	0	0	0	000	0	00
I / LOAD	LB	0000011	000	0000000	0000	1	0	1	1	0	1	000	0	00
	LH	0000011	001	0000000	0000	1	0	1	1	0	1	001	0	00
	LW	0000011	010	0000000	0000	1	0	1	1	0	1	010	0	00
	LBU	0000011	100	0000000	0000	1	0	1	1	0	1	100	0	00
	LHU	0000011	101	0000000	0000	1	0	1	1	0	1	101	0	00
S	SB	0100011	000	0000000	0000	0	0	1	0	1	0	000	0	00
	SH	0100011	001	0000000	0000	0	0	1	0	1	0	001	0	00
	SW	0100011	010	0000000	0000	0	0	1	0	1	0	010	0	00
SB	BEQ	1100001	000	0000000	0001	0	0	0	0	0	0	000	1	00
	BNE	1100001	001	0000000	0001	0	0	0	0	0	0	000	1	01
	BLT	1100001	100	0000000	1000	0	0	0	0	0	0	000	1	10
	BGE	1100001	101	0000000	1000	0	0	0	0	0	0	000	1	11
	BLTU	1100001	110	0000000	1001	0	0	0	0	0	0	000	1	10
	BGEU	1100001	111	0000000	1001	0	0	0	0	0	0	000	1	11
JALR	JALR	1100111	000	0000000	0000	1	1	1	0	0	0	000	0	00

The table provides details about different types of instructions implemented in the processor. The "TYPE" column categorizes instructions into "R" (R-Type), "I" (I-Type), "I/LOAD" (I-Type load), "S" (S-Type), and "SB" (SB-Type). Each instruction is associated with specific opcode, func3, func7, ALU operation, wrt\_en (write enable), jump, sel\_ALUsrc (select ALU source), MemRead (memory read), MemWrt (memory write), sel\_wrtdata (select write data), mem\_type (memory type), branch, and branch\_type.

- **Opcode:** The opcode field for these instructions.
- **Func3:** The func3 field, which differentiates between the load variations.
- **Func7:** The func7 field for distinguishing different loads.
- **ALU Operation:** Describes the ALU operation corresponding to each load operation.
- **wrt\_en:** Indicates whether writing to the register file is enabled (1 for enabled, 0 for disabled).
- **jump:** Specifies jump conditions (0 for not jumping, 1 for jumping).
- **sel\_ALUsrc:** Selects the ALU source (0 for register data, 1 for immediate value).
- **MemRead:** Indicates memory read operation (1 for read, 0 for no read).
- **MemWrt:** Indicates memory write operation (1 for write, 0 for no write).
- **sel\_wrtdata:** Selects the source for writing data (1 for memory data, 0 for ALU result).
- **mem\_type:** Describes the type of memory operation (e.g., LB, LH, LW, LBU, LHU).
- **branch:** Indicates whether this instruction can result in a branch (0 for no branch, 1 for branch).
- **branch\_type:** Specifies the branch type (00 for no branch, 01 for branch on not equal, 10 for branch on less than, 11 for branch on less than unsigned).

This microprogramming approach allows the processor to execute a wide range of instructions by generating appropriate control signals for the components based on the instruction's opcode and other fields.

## 4. Implemented Instructions
























### 4.1. R-Type Instructions

These instructions perform operations on registers. Examples include ADD, SUB, SLL (shift left logical), SLT (set less than), SLTU (set less than unsigned), XOR, SRL (shift right logical), SRA (shift right arithmetic), OR, and AND.

```
logic [31:0] Inst_mem [2** (inst_addr_width)-1:0];

assign Inst_mem[0]   = 32'h00100093; //      addi r1,r0, 1      ALUResult = h1 = r1
assign Inst_mem[1]   = 32'h00200113; //      addi r2,r0, 2      ALUResult = h2 = r2
assign Inst_mem[2]   = 32'h00308193; //      addi r3,r1, 3      ALUResult = h4 = r3
assign Inst_mem[3]   = 32'h00408213; //      addi r4,r1, 4      ALUResult = h5 = r4
assign Inst_mem[4]   = 32'h00510293; //      addi r5,r2, 5      ALUResult = h7 = r5
assign Inst_mem[5]   = 32'h00610313; //      addi r6,r2, 6      ALUResult = h8 = r6
assign Inst_mem[6]   = 32'h00718393; //      addi r7,r3, 7      ALUResult = hB = r7

//R type
assign Inst_mem[7]   = 32'h00208433; //      add  r8,r1,r2      ALUResult = h3 = r8
assign Inst_mem[8]   = 32'h404404b3; //      sub  r9,r8,r4      ALUResult = hfffffffe = -2 = r9
assign Inst_mem[9]   = 32'h00517533; //      and  r10 = r2 & r5  ALUResult = h2 = r10
assign Inst_mem[10]  = 32'h0041e5b3; //      or   r11 = r3 | r4  ALUResult = h5 = r11
assign Inst_mem[11]  = 32'h0072c633; //      xor  r12, r5, r7    ALUResult = hc = r12
assign Inst_mem[12]  = 32'h002356b3; //      srl  r13, r6, r2    ALUResult = h2 = r13
assign Inst_mem[13]  = 32'h00311733; //      sll  r14, r2, r3    ALUResult = h20 = r14
assign Inst_mem[14]  = 32'h4034d7b3; //      sra  r15, r9, r3    ALUResult = hfffffffe = r15
assign Inst_mem[15]  = 32'h00a8a833; //      slt  r16, r17,r10   ALUResult = h1 = r16
assign Inst_mem[16]  = 32'h00a8b8b3; //      sltu r17, r9,r10   ALUResult = h1 = r17
```

Name	Value	Data Type
▼  register[31:0][3...	00000000...	Array
>  [22][31:0]	00000000	Array
>  [21][31:0]	00000000	Array
>  [20][31:0]	00000000	Array
>  [19][31:0]	00000000	Array
>  [18][31:0]	00000000	Array
>  [17][31:0]	00000001	Array
>  [16][31:0]	00000001	Array
>  [15][31:0]	fffffffe	Array
>  [14][31:0]	00000020	Array
>  [13][31:0]	00000002	Array
>  [12][31:0]	0000000c	Array
>  [11][31:0]	00000005	Array
>  [10][31:0]	00000002	Array
>  [9][31:0]	fffffffe	Array
>  [8][31:0]	00000003	Array
>  [7][31:0]	0000000b	Array
>  [6][31:0]	00000008	Array
>  [5][31:0]	00000007	Array
>  [4][31:0]	00000005	Array
>  [3][31:0]	00000004	Array
>  [2][31:0]	00000002	Array
>  [1][31:0]	00000001	Array
>  [0][31:0]	00000000	Array



## 4.2. I-Type Computational Instructions

These instructions operate on immediate values. Examples include ADDI (add immediate), SLTI (set less than immediate), SLTIU (set less than immediate unsigned), XORI (XOR immediate), ORI (OR immediate), ANDI (AND immediate), SLLI, SRLI, and SRAI.

```
logic [31:0] Inst_mem [2**(inst_addr_width)-1:0];

assign Inst_mem[0]   = 32'h00100093; //      addi r1,r0, 1      ALUResult = h1 = r1
assign Inst_mem[1]   = 32'h00200113; //      addi r2,r0, 2      ALUResult = h2 = r2
assign Inst_mem[2]   = 32'h00308193; //      addi r3,r1, 3      ALUResult = h4 = r3
assign Inst_mem[3]   = 32'h00408213; //      addi r4,r1, 4      ALUResult = h5 = r4
assign Inst_mem[4]   = 32'h00510293; //      addi r5,r2, 5      ALUResult = h7 = r5
assign Inst_mem[5]   = 32'h00610313; //      addi r6,r2, 6      ALUResult = h8 = r6

//I type
assign Inst_mem[6]   = 32'h00718393; //      addi r7,r3, 7      ALUResult = hB = r7
assign Inst_mem[7]   = 32'h01614413; //      xori r8, r2, 16h    ALUResult = h14 = r8
assign Inst_mem[8]   = 32'h02e2e493; //      ori  r9, r5, 2eh   ALUResult = h2f = r9
assign Inst_mem[9]   = 32'h06f37513; //      andi r10, r6, 6fh  ALUResult = h8 = r10
assign Inst_mem[10]  = 32'h00349593; //      slli r11, r9, 3h   ALUResult = h178 = r11
assign Inst_mem[11]  = 32'h00335613; //      srli r12, r6, 3h   ALUResult = h1 = r12
assign Inst_mem[12]  = 32'h4026d693; //      srai r13, r13, 2h  ALUResult = h0 = r13
assign Inst_mem[13]  = 32'h0028a713; //      slti r14, r9, 2    ALUResult = h1 = r14
assign Inst_mem[14]  = 32'h0028b793; //      sltiu r15, r9, 2   ALUResult = h1 = r15
```

Name	Value	Data Type
▼  register[31:0][3...]	0000000...	Array
>  [22][31:0]	00000000	Array
>  [21][31:0]	00000000	Array
>  [20][31:0]	00000000	Array
>  [19][31:0]	00000000	Array
>  [18][31:0]	00000000	Array
>  [17][31:0]	00000000	Array
>  [16][31:0]	00000000	Array
>  [15][31:0]	00000001	Array
>  [14][31:0]	00000001	Array
>  [13][31:0]	00000000	Array
>  [12][31:0]	00000001	Array
>  [11][31:0]	00000178	Array
>  [10][31:0]	00000008	Array
>  [9][31:0]	0000002f	Array
>  [8][31:0]	00000014	Array
>  [7][31:0]	0000000b	Array
>  [6][31:0]	00000008	Array
>  [5][31:0]	00000007	Array
>  [4][31:0]	00000005	Array
>  [3][31:0]	00000004	Array
>  [2][31:0]	00000002	Array
>  [1][31:0]	00000001	Array
>  [0][31:0]	00000000	Array











### 4.3. I-Type Load Instructions

I-Type load instructions are used to load data from memory into a register. These instructions operate on immediate values and require a memory address to access data from memory. The loaded data is then stored in a specified register. The most common I-type load instructions include LW (Load Word), LB (Load Byte), LBU (Load Byte Unsigned), LH (Load Halfword), and LHU (Load Halfword Unsigned).

```
logic [31:0] Inst_mem [2** (inst_addr_width)-1:0];

assign Inst_mem[0] = 32'h00100093; //      addi r1,r0, 1      ALUResult = h1 = r1
assign Inst_mem[1] = 32'h00200113; //      addi r2,r0, 2      ALUResult = h2 = r2
assign Inst_mem[2] = 32'h00308193; //      addi r3,r1, 3      ALUResult = h4 = r3
assign Inst_mem[3] = 32'h00408213; //      addi r4,r1, 4      ALUResult = h5 = r4
assign Inst_mem[4] = 32'h00510293; //      addi r5,r2, 5      ALUResult = h7 = r5
assign Inst_mem[5] = 32'h00610313; //      addi r6,r2, 6      ALUResult = h8 = r6
assign Inst_mem[6] = 32'h00208433; //      add  r8,r1,r2      ALUResult = h3 = r8
assign Inst_mem[7] = 32'h404404b3; //      sub  r9,r8,r4      ALUResult = hffffffe = -2 = r9
assign Inst_mem[8] = 32'h00902a23; //      sw   20(r0)<- r9

//I type load
assign Inst_mem[9] = 32'h01402503; //      lw   r10<-20(r0)    ALUResult = fffffffe = r10
assign Inst_mem[10] = 32'h01400583; //      lb   r11<-20(r0)    ALUResult = fffffffe = r11
assign Inst_mem[11] = 32'h01401603; //      lh   r12<-20(r0)    ALUResult = fffffffe = r12
assign Inst_mem[12] = 32'h01404683; //      lbu  r13<-20(r0)    ALUResult = 000000fe = r13
assign Inst_mem[13] = 32'h01405703; //      lhu  r14<-20(r0)    ALUResult = 0000fffe = r14
```

Name	Value	Data Type
▼  register[31:0][3...	00000000...	Array
>  [15][31:0]	00000000	Array
>  [14][31:0]	0000fffe	Array
>  [13][31:0]	000000fe	Array
>  [12][31:0]	fffffffe	Array
>  [11][31:0]	fffffffe	Array
>  [10][31:0]	fffffffe	Array
>  [9][31:0]	fffffffe	Array
>  [8][31:0]	00000003	Array
>  [7][31:0]	00000000	Array
>  [6][31:0]	00000008	Array
>  [5][31:0]	00000007	Array
>  [4][31:0]	00000005	Array
>  [3][31:0]	00000004	Array
>  [2][31:0]	00000002	Array
>  [1][31:0]	00000001	Array
>  [0][31:0]	00000000	Array















## 4.4. S-Type Instructions

These instructions store values in memory. Examples include SB (store byte), SH (store half-word), and SW (store word).

```
logic [31:0] Inst_mem [2** (inst_addr_width)-1:0];

assign Inst_mem[0]   = 32'h00100093; //      addi r1,r0, 1      ALUResult = h1 = r1
assign Inst_mem[1]   = 32'h00200113; //      addi r2,r0, 2      ALUResult = h2 = r2
assign Inst_mem[2]   = 32'h00308193; //      addi r3,r1, 3      ALUResult = h4 = r3
assign Inst_mem[3]   = 32'h00408213; //      addi r4,r1, 4      ALUResult = h5 = r4
assign Inst_mem[4]   = 32'h00510293; //      addi r5,r2, 5      ALUResult = h7 = r5
assign Inst_mem[5]   = 32'h00610313; //      addi r6,r2, 6      ALUResult = h8 = r6
assign Inst_mem[6]   = 32'h00208433; //      add  r8,r1,r2      ALUResult = h3 = r8
assign Inst_mem[7]   = 32'h404404b3; //      sub  r9,r8,r4      ALUResult = hffffffe = -2 = r9

//S type
assign Inst_mem[8]   = 32'h00902a23; //      sw  r9 ->20(r0)      ALUResult = h14
assign Inst_mem[9]   = 32'h00908a23; //      sb  r9 ->20(r1)      ALUResult = h15
assign Inst_mem[10]  = 32'h00911a23; //      sh  r9 ->20(r2)      ALUResult = h16
```

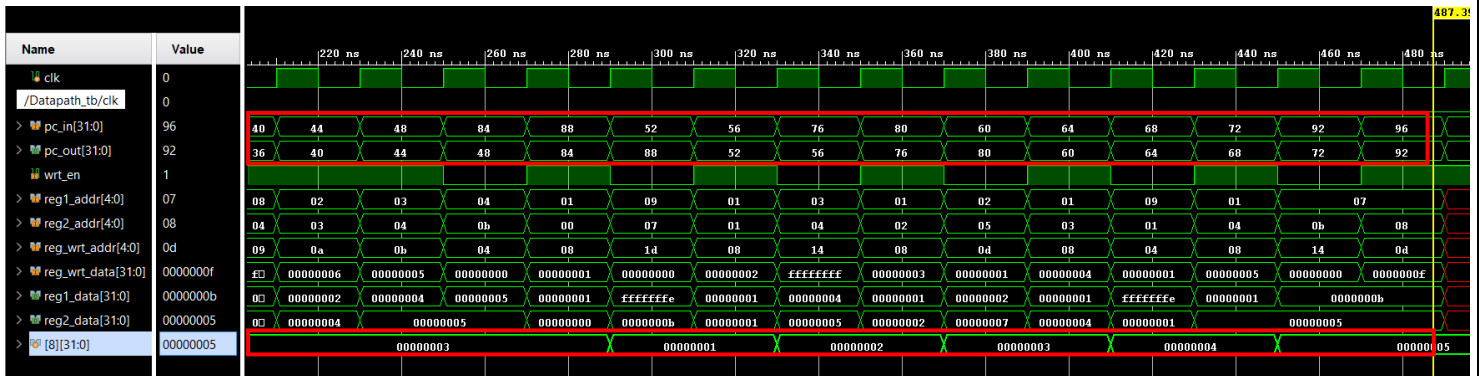
Name	Value	Data Type
▼  memory[511:0]...	XXXXXX...	Array
>  [30][31:0]	XXXXXX...	Array
>  [29][31:0]	XXXXXX...	Array
>  [28][31:0]	XXXXXX...	Array
>  [27][31:0]	XXXXXX...	Array
>  [26][31:0]	XXXXXX...	Array
>  [25][31:0]	XXXXXX...	Array
>  [24][31:0]	XXXXXX...	Array
>  [23][31:0]	XXXXXX...	Array
>  [22][31:0]	0000fffe	Array
>  [21][31:0]	000000fe	Array
>  [20][31:0]	fffffffe	Array
>  [19][31:0]	XXXXXX...	Array
>  [18][31:0]	XXXXXX...	Array

## 4.5. SB-Type Instructions

These branch instructions compare values and change the program counter accordingly. Examples include BEQ (branch equal), BNE (branch not equal), BLT (branch less than), BGE (branch greater than or equal), BLTU (branch less than immediate unsigned), and BGEU (branch greater than or equal immediate unsigned).

```
assign Inst_mem[0] = 32'h00007033; // and r0,r0,r0          ALUResult = h0 = r0
assign Inst_mem[1] = 32'h00100093; // addi r1,r0, 1         ALUResult = h1 = r1
assign Inst_mem[2] = 32'h00200113; // addi r2,r0, 2         ALUResult = h2 = r2
assign Inst_mem[3] = 32'h00308193; // addi r3,r1, 3         ALUResult = h4 = r3
assign Inst_mem[4] = 32'h00408213; // addi r4,r1, 4         ALUResult = h5 = r4
assign Inst_mem[5] = 32'h00510293; // addi r5,r2, 5         ALUResult = h7 = r5
assign Inst_mem[6] = 32'h00610313; // addi r6,r2, 6         ALUResult = h8 = r6
assign Inst_mem[7] = 32'h00718393; // addi r7,r3, 7         ALUResult = hB = r7
assign Inst_mem[8] = 32'h00208433; // add r8,r1,r2          ALUResult = h3 = r8
assign Inst_mem[9] = 32'h404404b3; // sub r9,r8,r4          ALUResult = hfffffffe = -2 = r9
assign Inst_mem[10] = 32'h00310533; // add r10,r2,r3         ALUResult = h6 = r10
assign Inst_mem[11] = 32'h0041e5b3; // or r11 = r3 | r4      ALUResult = h5 = r11

//SB type
assign Inst_mem[12] = 32'h02b20263; // beq r4,r11,36         ALUResult = 00000000    branch taken to inst_mem[21]
assign Inst_mem[13] = 32'h00108413; // addi r8,r1,1          ALUResult = h2 = r8
assign Inst_mem[14] = 32'h00419a63; // bne r3,r4,20          ALUResult = ffffffff    branch taken to inst_mem[19]
assign Inst_mem[15] = 32'h00308413; // addi r8,r1,3          ALUResult = h4 = r8
assign Inst_mem[16] = 32'h0014c263; // blt r9,r1,4           ALUResult = 00000001    branch taken to inst_mem[17]
assign Inst_mem[17] = 32'h00408413; // addi r8,r1,4          ALUResult = h5 = r8
assign Inst_mem[18] = 32'h00b3da63; // bge r7,r11,20         ALUResult = 00000001    branch taken to inst_mem[23]
assign Inst_mem[19] = 32'h00508413; // addi r8,r1,5          ALUResult = h6 = r8
assign Inst_mem[20] = 32'hfe5166e3; // bltu r2, r5, -24      ALUResult = 00000001    branch taken to inst_mem[15]
assign Inst_mem[21] = 32'h00008413; // add r8,r1,0           ALUResult = 1 = r8
assign Inst_mem[22] = 32'hfc74fee3; // bgeu r9,r7,-36        ALUResult = 00000001    branch taken to inst_mem[13]
assign Inst_mem[23] = 32'h0083e6b3; // or r13 = r7 | r8      ALUResult = hf = r13
```



Name	Value	Data Type
register[31:0][31:0]	00000000	Array
> [15][31:0]	00000000	Array
> [14][31:0]	00000000	Array
> [13][31:0]	0000000f	Array
> [12][31:0]	00000000	Array
> [11][31:0]	00000005	Array
> [10][31:0]	00000006	Array
> [9][31:0]	fffffffe	Array
> [8][31:0]	00000005	Array
> [7][31:0]	0000000b	Array
> [6][31:0]	00000008	Array
> [5][31:0]	00000007	Array
> [4][31:0]	00000005	Array
> [3][31:0]	00000004	Array
> [2][31:0]	00000002	Array
> [1][31:0]	00000001	Array
> [0][31:0]	00000000	Array

## 4.6. MUL Instruction

The MUL instruction is a custom instruction designed to perform multiplication of two unsigned values. The MUL instruction is implemented using the ALU component. It takes two source operands, multiplies them, and stores the result in a destination register.

```

assign Inst_mem[0] = 32'h0007033; // and r0,r0,r0          ALUResult = h0 = r0
assign Inst_mem[1] = 32'h00100093; // addi r1,r0, 1        ALUResult = h1 = r1
assign Inst_mem[2] = 32'h00200113; // addi r2,r0, 2        ALUResult = h2 = r2
assign Inst_mem[3] = 32'h00308193; // addi r3,r1, 3        ALUResult = h4 = r3
assign Inst_mem[4] = 32'h00408213; // addi r4,r1, 4        ALUResult = h5 = r4
assign Inst_mem[5] = 32'h00510293; // addi r5,r2, 5        ALUResult = h7 = r5
assign Inst_mem[6] = 32'h00610313; // addi r6,r2, 6        ALUResult = h8 = r6
assign Inst_mem[7] = 32'h00718393; // addi r7,r3, 7        ALUResult = hB = r7
assign Inst_mem[8] = 32'h00208433; // add r8,r1,r2        ALUResult = h3 = r8
assign Inst_mem[9] = 32'h404404b3; // sub r9,r8,r4        ALUResult = hffffffe = -2 = r9

//MUL
assign Inst_mem[10] = 32'h40419533; // mul r10,r3,r4       ALUResult = h14 = r10

```

Name	Value	Data Type
▼  register[31:0][3...]	0000000...	Array
>  [15][31:0]	00000000	Array
>  [14][31:0]	00000000	Array
>  [13][31:0]	00000000	Array
>  [12][31:0]	00000000	Array
>  [11][31:0]	00000000	Array
>  [10][31:0]	00000014	Array
>  [9][31:0]	fffffffe	Array
>  [8][31:0]	00000003	Array
>  [7][31:0]	0000000b	Array
>  [6][31:0]	00000008	Array
>  [5][31:0]	00000007	Array
>  [4][31:0]	00000005	Array
>  [3][31:0]	00000004	Array
>  [2][31:0]	00000002	Array
>  [1][31:0]	00000001	Array
>  [0][31:0]	00000000	Array

## 5. Resource Utilization

```
| Tool Version : Vivado v.2018.3 (win64) Build 2405991 Thu Dec 6 23:38:27 MST 2018
| Date        : Mon Oct 16 22:18:12 2023
| Host       : LAPTOP-5KE0II8J running 64-bit major release (build 9200)
| Command    : report_utilization -file Datapath_utilization_synth.rpt -pb Datapath_utilization_synth.pb
| Design     : Datapath
| Device     : 7z010clg400-1
| Design State : Synthesized
```

### 5.1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	1282	0	17600	7.28
LUT as Logic	1026	0	17600	5.83
LUT as Memory	256	0	6000	4.27
LUT as Distributed RAM	256	0		
LUT as Shift Register	0	0		
Slice Registers	589	0	35200	1.67
Register as Flip Flop	512	0	35200	1.45
Register as Latch	77	0	35200	0.22
F7 Muxes	287	0	8800	3.26
F8 Muxes	127	0	4400	2.89

Note: The Final LUT count, after physical optimizations and full implementation, is typically lower. Run `opt_design` after synthesis, if not already completed, for a more realistic count.

#### 5.1.1. Summary of Registers by Type

Total	Clock Enable	Synchronous	Asynchronous
0	-	-	-
0	-	-	Set
0	-	-	Reset
0	-	Set	-
0	-	Reset	-
0	Yes	-	-
0	Yes	-	Set
109	Yes	-	Reset
0	Yes	Set	-
480	Yes	Reset	-

### 5.2. Memory

Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	0	0	60	0.00
RAMB36/FIFO*	0	0	60	0.00
RAMB18	0	0	120	0.00

### 5.3. DSP

Site Type	Used	Fixed	Available	Util%
DSPs	3	0	80	3.75
DSP48E1 only	3			

### 5.4. IO and GT Specific

Site Type	Used	Fixed	Available	Util%
Bonded IOB	34	0	100	34.00
Bonded IPADs	0	0	2	0.00
Bonded IOPADs	0	0	130	0.00
PHY_CONTROL	0	0	2	0.00
PHASER_REF	0	0	2	0.00
OUT_FIFO	0	0	8	0.00
IN_FIFO	0	0	8	0.00
IDELAYCTRL	0	0	2	0.00
IBUFDS	0	0	96	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	8	0.00
PHASER_IN/PHASER_IN_PHY	0	0	8	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	100	0.00
ILOGIC	0	0	100	0.00
OLOGIC	0	0	100	0.00

### 5.5. Clocking

Site Type	Used	Fixed	Available	Util%
BUFGCTRL	2	0	32	6.25
BUFIO	0	0	8	0.00
MMCME2_ADV	0	0	2	0.00
PLLE2_ADV	0	0	2	0.00
BUFMRCE	0	0	4	0.00
BUFHCE	0	0	48	0.00
BUFR	0	0	8	0.00

### 5.6. Specific Feature

Site Type	Used	Fixed	Available	Util%
BSCANE2	0	0	4	0.00
CAPTUREE2	0	0	1	0.00
DNA_PORT	0	0	1	0.00
EFUSE_USR	0	0	1	0.00
FRAME_ECCE2	0	0	1	0.00
ICAPE2	0	0	2	0.00
STARTUPE2	0	0	1	0.00
XADC	0	0	1	0.00

## 5.7. Primitives

Ref Name	Used	Functional Category
FDRE	480	Flop & Latch
LUT6	475	LUT
MUXF7	287	MuxFx
LUT5	265	LUT
RAMS64E	256	Distributed Memory
LUT2	155	LUT
MUXF8	127	MuxFx
LUT4	117	LUT
LUT3	113	LUT
LDCE	77	Flop & Latch
CARRY4	39	CarryLogic
OBUF	32	IO
FDCE	32	Flop & Latch
LUT1	3	LUT
DSP48E1	3	Block Arithmetic
IBUF	2	IO
BUFG	2	Clock

- No black boxes used.
- No instantiated netlists used.

## 6. Conclusion

This report provides a comprehensive overview of the design and implementation of the 32-bit non-pipelined RISC-V processor using Microprogramming with 3 bus structure. The processor architecture is meticulously structured, featuring microprogramming approach that enables it to decode instructions and generate control signals based on opcode and other relevant fields. The implementation encompasses a range of instruction types, including R-Type, I-Type, S-Type, SB-Type, and additional instructions such as MUL. The processor's design leverages various modules to execute these instructions efficiently. The successful synthesis of this processor, as indicated by the resource utilization report, is a key milestone in ensuring its functionality on the target FPGA device. Overall, this report offers a detailed insight into the design, implementation, and resource utilization of the processor, highlighting its potential for various computing applications.