Department of Electronic & Telecommunication Engineering
University of Moratuwa

EN3021 - Digital System Design

Single Cycle RISC-V Microprocessor

H.M.S.D. Bandara - 200064C

10 th of oct 2023

Introduction

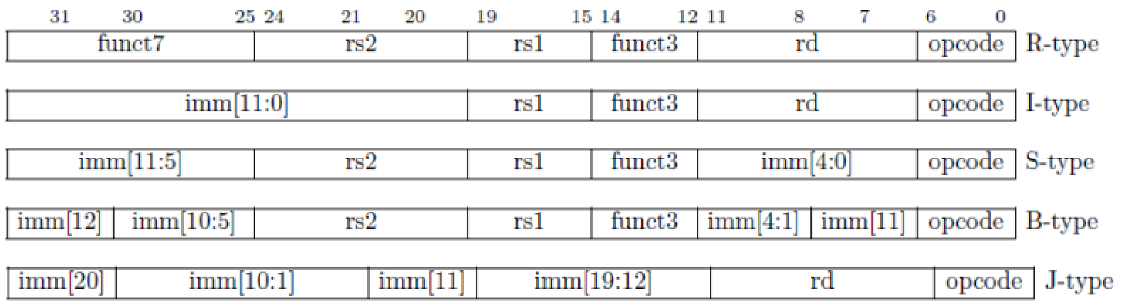
A RISC-V 32I single-cycle CPU is a processor that employs the Reduced Instruction Set Computing (RISC) architecture, utilizing a 32-bit instruction set and executing each instruction within a single clock cycle. This design offers a swift and efficient method for handling instructions. It is based on the open-source RISC-V ISA, which is gaining popularity in embedded systems, microcontrollers, and various applications. The "32I" in its name denotes the 32-bit instruction width, striking a balance between code density and processing speed. RISC-V processors are recognized for their straightforward architecture, effective implementation, and low power consumption, rendering them suitable for a broad spectrum of devices and use cases.

The processor

An FPGA, or Field Programmable Gate Array, is a remarkably adaptable resource used in the realm of digital electronics. It comprises a grid pattern of millions of logic integrated circuits (ICs), and users have the capability to customize the interconnections between these ICs to suit their specific requirements. In the project at hand, the FPGA board in use incorporates a chip manufactured by Terasic Inc., along with supplementary peripheral components. To establish the desired connections between the ICs on the FPGA, instructions need to be fed into the system from a computer.

- ### Instruction Set Architecture

  Our proposed design accommodates all RISC-V instructions across the various instruction classes, with each instruction being standardized at a length of 32 bits.


31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

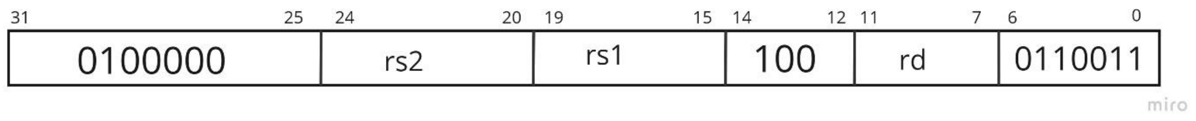
Type	Inst	Name	Description
	add	ADD	rd = rs1 + rs2
	sub	SUB	rd = rs1 - rs2
	xor	XOR	rd = rs1 ˆ rs2
	or	OR	rd = rs1 | rs2
R - Type	and	AND	rd = rs1 & rs2
	sll	Shift Left Logical	rd = rs1 << rs2
	srl	Shift Right Logical	rd = rs1 >> rs2
	sra	Shift Right Arith*	rd = rs1 >> rs2
	slt	Set Less Than	rd = (rs1 < rs2)?1:0
	sltu	Set Less Than (U)	rd = (rs1 < rs2)?1:0

	addi	ADD Immediate	rd = rs1 + imm
	xori	XOR Immediate	rd = rs1 ˆ imm
	ori	OR Immediate	rd = rs1 | imm
	andi	AND Immediate	rd = rs1 & imm
	slli	Shift Left Logical Imm	rd = rs1 << imm[0:4]
	srli	Shift Right Logical Imm	rd = rs1 >> imm[0:4]
	srai	Shift Right Arith Imm	rd = rs1 >> imm[0:4]
I Type	slti	Set Less Than Imm	rd = (rs1 < imm)?1:0
	sltiu	Set Less Than Imm (U)	rd = (rs1 < imm)?1:0
	lb	Load Byte	rd = M[rs1+imm][0:7]
	lh	Load Half	rd = M[rs1+imm][0:15]
	lw	Load Word	rd = M[rs1+imm][0:31]
	lbu	Load Byte (U)	rd = M[rs1+imm][0:7]
	lhu	Load Half (U)	rd = M[rs1+imm][0:15]
	sb	Store Byte	M[rs1+imm][0:7] = rs2[0:7]
S - Type	sh	Store Half	M[rs1+imm][0:15] = rs2[0:15]
	sw	Store Word	M[rs1+imm][0:31] = rs2[0:31]
	beq	Branch ==	if(rs1 == rs2) PC += imm
	bne	Branch !=	if(rs1 != rs2) PC += imm
	blt	Branch <	if(rs1 < rs2) PC += imm
B - Type	bge	Branch ≥	if(rs1 >= rs2) PC += imm
	bltu	Branch < (U)	if(rs1 < rs2) PC += imm
	bgeu	Branch ≥ (U)	if(rs1 >= rs2) PC += imm
	jal	Jump And Link	rd = PC+4; PC += imm
J - Type	jalr	Jump And Link Reg	rd = PC+4; PC = rs1 + imm

- ## Additional instructions

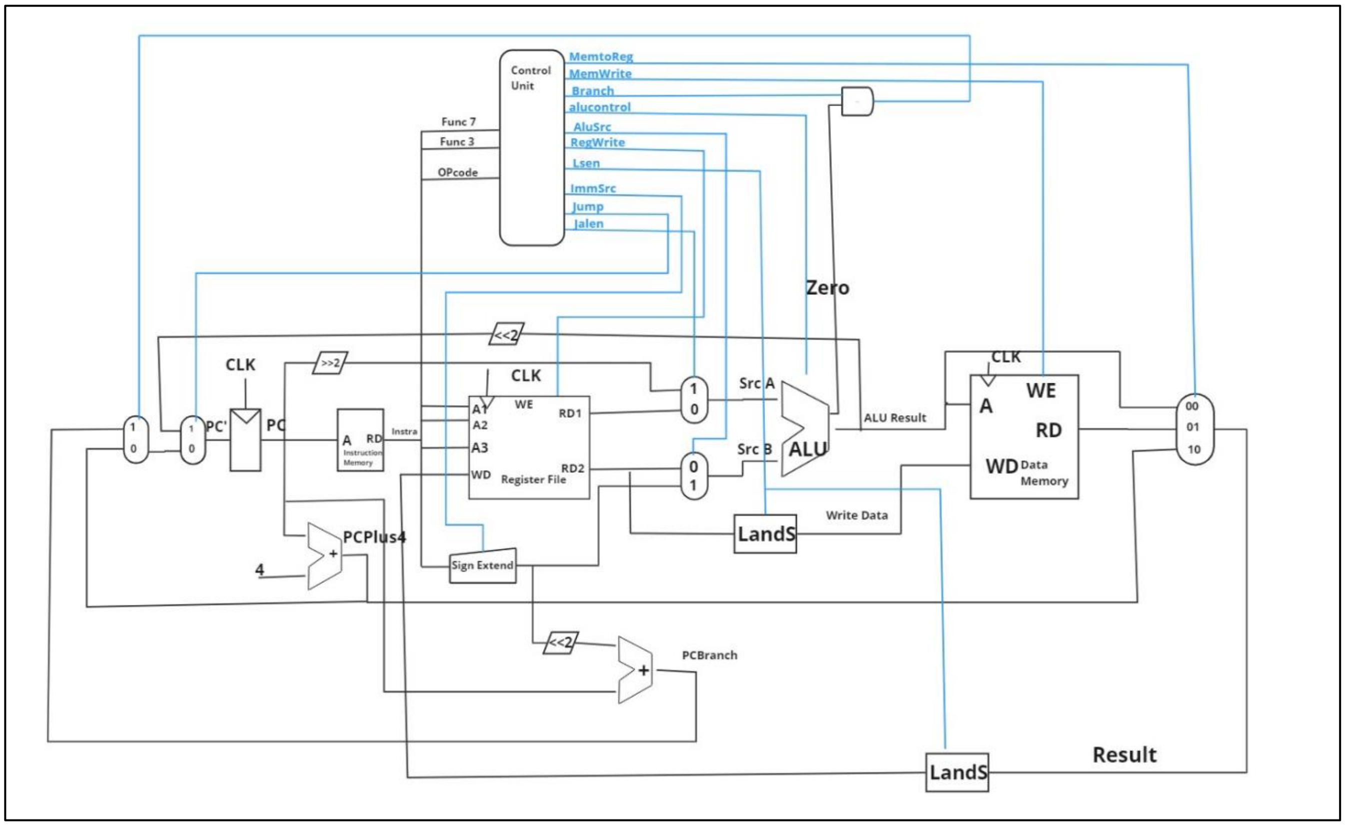
In the context of our single-cycle RISC-V processor project, we've taken a significant step by introducing an additional instruction: the Unsigned Multiplication instruction. This instruction operates similarly to R-type instructions and is designed to perform multiplication operations on 16-bit operands. By expanding the processor's instruction set to include this specific operation, we've enhanced its functionality, catering to applications that require efficient and precise unsigned multiplication. This report outlines the design, implementation, and performance of this new instruction within the processor, showcasing how it contributes to the processor's versatility and its potential to address a broader range of computational tasks.

Example Instruction:



Type	Inst	Name	Description
R - Type	mul	Multiplication (U)	rd = rs1 * rs2

- **Data Path**



- **Modules and Components**

Program Counter

The program counter (PC) is a vital register within a computer's central processing unit (CPU), storing the memory address of the currently executing instruction. It is designed to automatically increment after the execution of each instruction, ensuring that the CPU proceeds to the subsequent instruction located at the next memory address.

Control Unit

In a RISC-V 32I processor, the control unit plays a pivotal role by decoding instructions retrieved from the instruction memory and generating control signals to orchestrate the processor's internal operations. It's responsible for ascertaining the type of instruction, and it manages a broad spectrum of instruction categories, including those related to registers, immediate values, branching, as well as load and store operations.

Instructions in the RISC-V 32I architecture adhere to a fixed-length and binary encoding format. The control unit diligently deciphers each instruction and transmits specific control signals that govern the flow of data and instructions within the processor. These control signals act as directives, harmonizing the activities of various components within the processor, such as the Arithmetic Logic Unit (ALU), the register file, and the memory unit.

The control unit generates these control signals solely based on the instruction's opcode.

1. **ALUOp** 2-bit signal allows for the subsequent 'ALU controller' module to determine the ALU operation.

2. **MemtoReg** signal is used to control the multiplexer that selects between the data from the (cache) memory and the ALU result for storing in the register during a register write.

3. **MemWrite** signal indicates whether the processor should write data to memory.

4. **Branch** signal specifies whether the processor should take a branch instruction, i.e., change the program flow to a different location in the code as specified in the branch instruction.

5. **ALUSrc** signal determines the source of the data (2nd register or immediate value) that should be given to the ALU B-port

6. **RegWrite** signal enables writing to the register file.

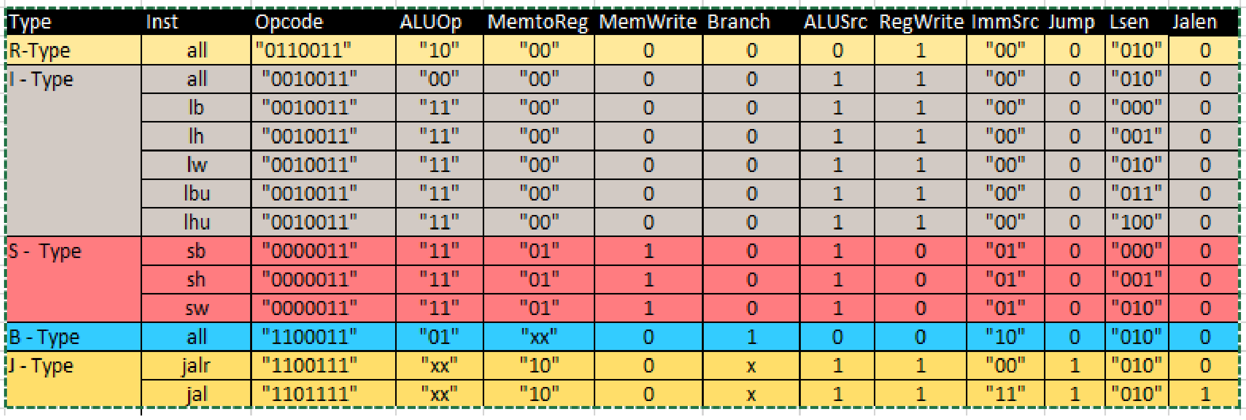
7. **ImmSrc** Immediate values in instructions are set based on the enable signal (Immsrc)

8. **Jump** signal is specifically for the I-Type JALR instruction. It facilitates the storing of PC+4 value in the destination register by selecting it and to increment the PC to RS+IMM

9. **LSen** used to enable and control data transfer operations for load (lb, lh, lw, lbu, lhu) and store (sb, sh, sw) instructions.

10. **Jalen** The jalen signal is used to enable and control the execution of the 'jal' instruction, allowing for jump and link operations in the processor.

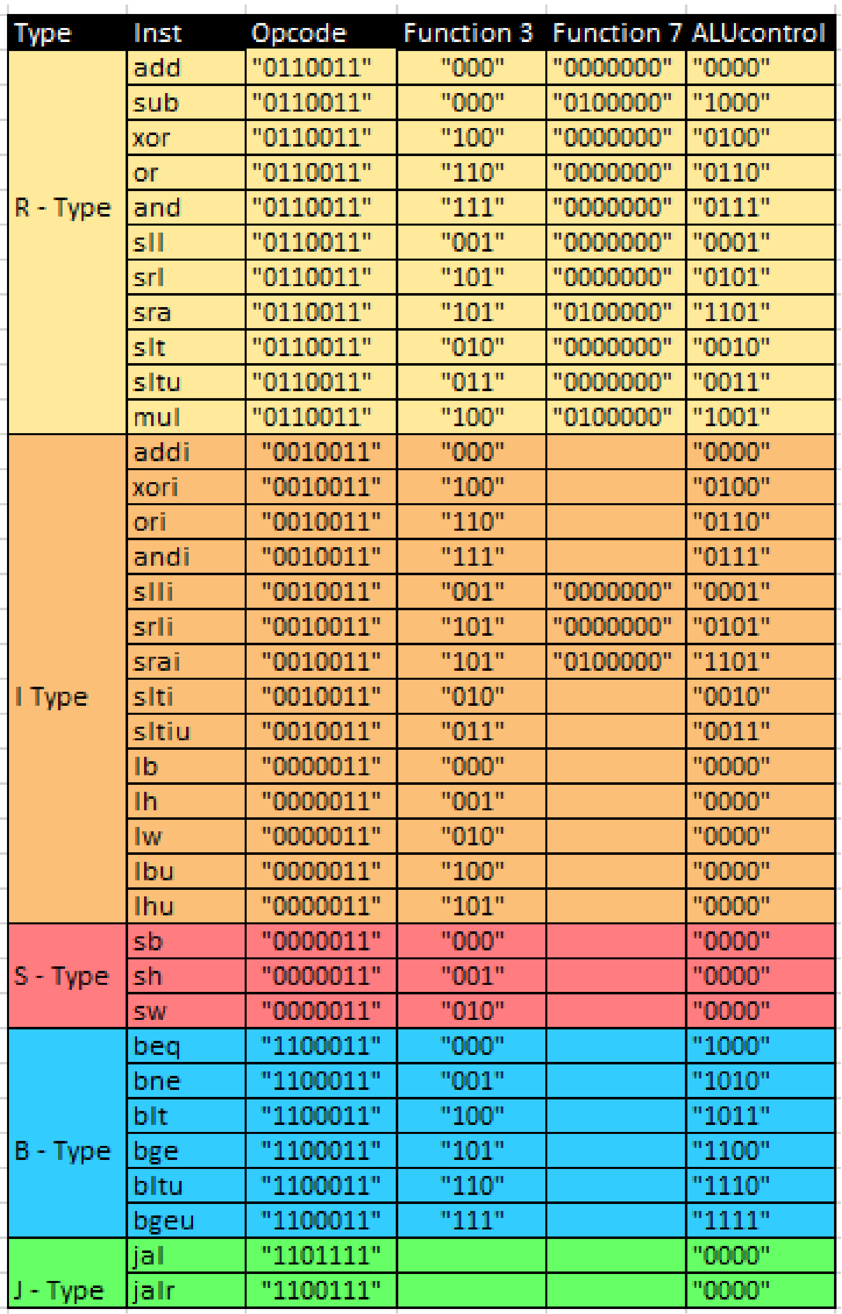
The derivation of the control logic



Type	Inst	Opcode	ALUOp	MemtoReg	MemWrite	Branch	ALUSrc	RegWrite	ImmSrc	Jump	Lsen	Jalen
R-Type	all	"0110011"	"10"	"00"	0	0	0	1	"00"	0	"010"	0
I - Type	all	"0010011"	"00"	"00"	0	0	1	1	"00"	0	"010"	0
	lb	"0010011"	"11"	"00"	0	0	1	1	"00"	0	"000"	0
	lh	"0010011"	"11"	"00"	0	0	1	1	"00"	0	"001"	0
	lw	"0010011"	"11"	"00"	0	0	1	1	"00"	0	"010"	0
	lbu	"0010011"	"11"	"00"	0	0	1	1	"00"	0	"011"	0
	lhu	"0010011"	"11"	"00"	0	0	1	1	"00"	0	"100"	0
S - Type	sb	"0000011"	"11"	"01"	1	0	1	0	"01"	0	"000"	0
	sh	"0000011"	"11"	"01"	1	0	1	0	"01"	0	"001"	0
	sw	"0000011"	"11"	"01"	1	0	1	0	"01"	0	"010"	0
B - Type	all	"1100011"	"01"	"xx"	0	1	0	0	"10"	0	"010"	0
J - Type	jalr	"1100111"	"xx"	"10"	0	x	1	1	"00"	1	"010"	0
	jal	"1101111"	"xx"	"10"	0	x	1	1	"11"	1	"010"	1

ALU Controller

In our processor, the ALU controller generates a 4-bit ALU control signal that dictates the operation to be performed by the Arithmetic Logic Unit (ALU). This signal plays a pivotal role in executing instructions, ensuring proper data manipulation, and facilitating computation within the processor's architecture. The ALU control signal is determined based on the opcode or specific instruction being processed, directing the ALU to perform operations such as addition, subtraction, bitwise logical operations, comparisons, and more, depending on the immediate instruction requirements.



Type	Inst	Opcode	Function 3	Function 7	ALUcontrol
	add	"0110011"	"000"	"0000000"	"0000"
	sub	"0110011"	"000"	"0100000"	"1000"
	xor	"0110011"	"100"	"0000000"	"0100"
	or	"0110011"	"110"	"0000000"	"0110"
R - Type	and	"0110011"	"111"	"0000000"	"0111"
	sll	"0110011"	"001"	"0000000"	"0001"
	srl	"0110011"	"101"	"0000000"	"0101"
	sra	"0110011"	"101"	"0100000"	"1101"
	slt	"0110011"	"010"	"0000000"	"0010"
	sltu	"0110011"	"011"	"0000000"	"0011"
	mul	"0110011"	"100"	"0100000"	"1001"
	addi	"0010011"	"000"		"0000"
	xori	"0010011"	"100"		"0100"
	ori	"0010011"	"110"		"0110"
	andi	"0010011"	"111"		"0111"
	slli	"0010011"	"001"	"0000000"	"0001"
	srli	"0010011"	"101"	"0000000"	"0101"
	srai	"0010011"	"101"	"0100000"	"1101"
I Type	slti	"0010011"	"010"		"0010"
	sltiu	"0010011"	"011"		"0011"
	lb	"0000011"	"000"		"0000"
	lh	"0000011"	"001"		"0000"
	lw	"0000011"	"010"		"0000"
	lbu	"0000011"	"100"		"0000"
	lhu	"0000011"	"101"		"0000"
	sb	"0000011"	"000"		"0000"
S - Type	sh	"0000011"	"001"		"0000"
	sw	"0000011"	"010"		"0000"
	beq	"1100011"	"000"		"1000"
	bne	"1100011"	"001"		"1010"
	blt	"1100011"	"100"		"1011"
B - Type	bge	"1100011"	"101"		"1100"
	bltu	"1100011"	"110"		"1110"
	bgeu	"1100011"	"111"		"1111"
	jal	"1101111"			"0000"
J - Type	jalr	"1100111"			"0000"

ALU

The ALU in a RISC-V 32I processor is a versatile unit responsible for a wide array of operations, encompassing addition, subtraction, multiplication, division, bit shifting, and various bitwise logical operations such as AND, OR, and XOR. Additionally, it handles comparison operations like checking for equality, inequality, greater than, less than, and unsigned comparisons.

In the processor's operation, the ALU's actions are determined by the ALU Controller, which obtains its instructions from the control unit and the instruction decoder. This controller instructs the ALU on which operation to perform. The ALU then takes its input data from the register file and conducts the specified operation. The resulting output is then written back into the register file or stored in memory, depending on the specific task at hand.

Instruction Memory

The instruction memory in our processor is responsible for storing the program instructions to be executed, separate from the data memory where processing data is stored. The processor fetches instructions from the instruction memory, and in the control unit, these instructions are decoded, and control signals are generated to execute them. Instructions are represented in fixed-length binary format, simplifying the decoding and execution process. Typically, instruction memory is implemented using read-only memory (ROM) or flash memory, ensuring that it remains unalterable during execution. In our design, the instruction memory has a size of 32x1024, providing the capacity to store a substantial number of instructions for program execution.

Data Memory

In our processor, the data memory serves as the repository for variables and data structures employed by the program. Data is fetched from the data memory when the program necessitates operations, and the results are subsequently stored back in the data memory upon completion. Typically, the data memory is structured as an extensive array of memory cells. In our design, the data memory is configured with a size of 32x1024 bits and operates on a byte-addressable basis, ensuring ample storage capacity and granularity for handling data manipulation tasks.

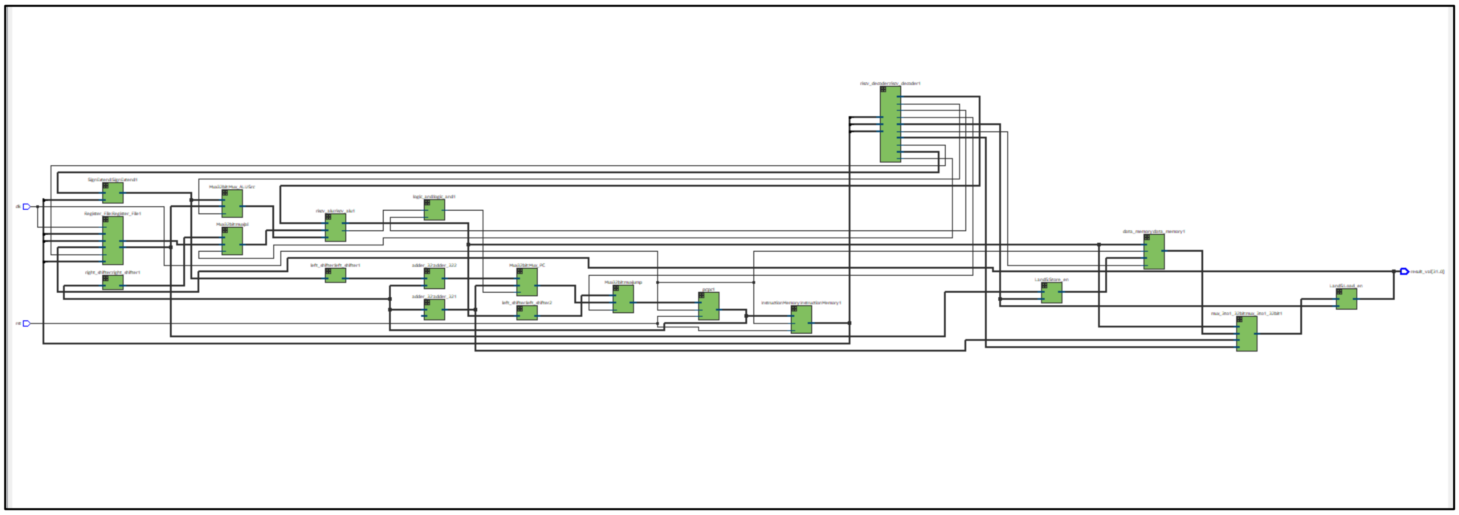
Sign Extend

The sign extender is a critical component in instruction processing that ensures the correct handling of immediate values in instructions. It extends the sign bit of a shorter immediate value to match the required length, ensuring consistent and accurate data representation. This operation is particularly important for signed integer values, as it maintains their signedness during instruction execution.

The sign extender is especially vital in RISC-V architecture, as it helps avoid unintended sign-related issues that could lead to incorrect results in arithmetic or logical operations. By preserving the sign bit, the sign extender enables the processor to correctly interpret immediate values in both positive and negative contexts, contributing to the precision and reliability of the instruction execution process.

Verilog Implementation

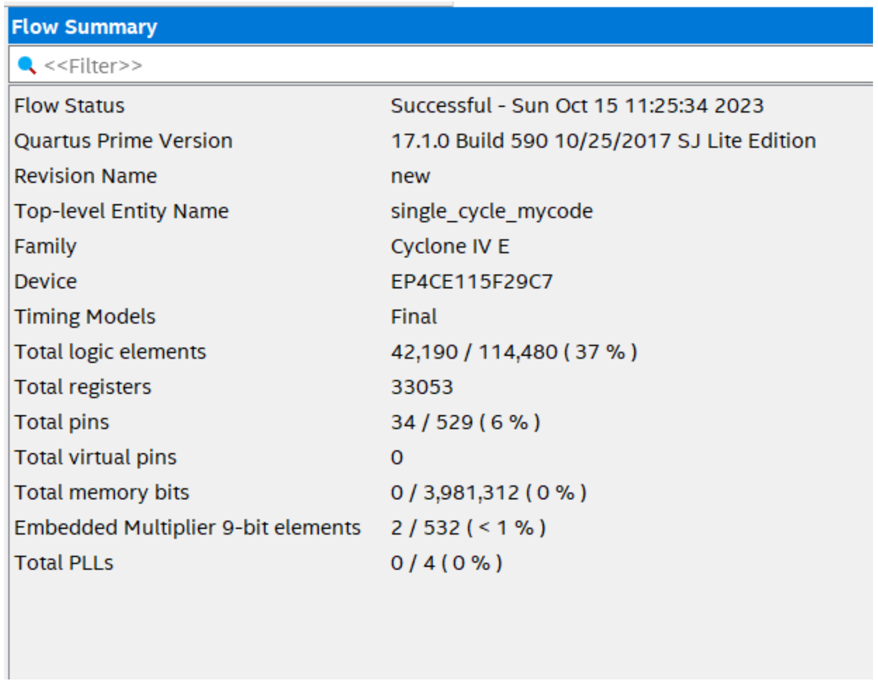
RTL view of the complete processor



Implement on Quartus Prime Lite

This processor's development was accomplished using Quartus Prime software and the Verilog Hardware Description Language (HDL). To ensure the robustness of the design, each module was compiled and rigorously tested. The top-level processor design was subsequently compiled and subjected to simulation using the integrated ModelSim software within Quartus Prime, where every instruction was thoroughly tested for functionality and correctness. The data paths within the processor were meticulously scrutinized and validated through the Netlist Viewer (RTL Viewer) feature, ensuring that the processor's internal connections and data flow were accurate and efficient.

The flow summary is as follows:

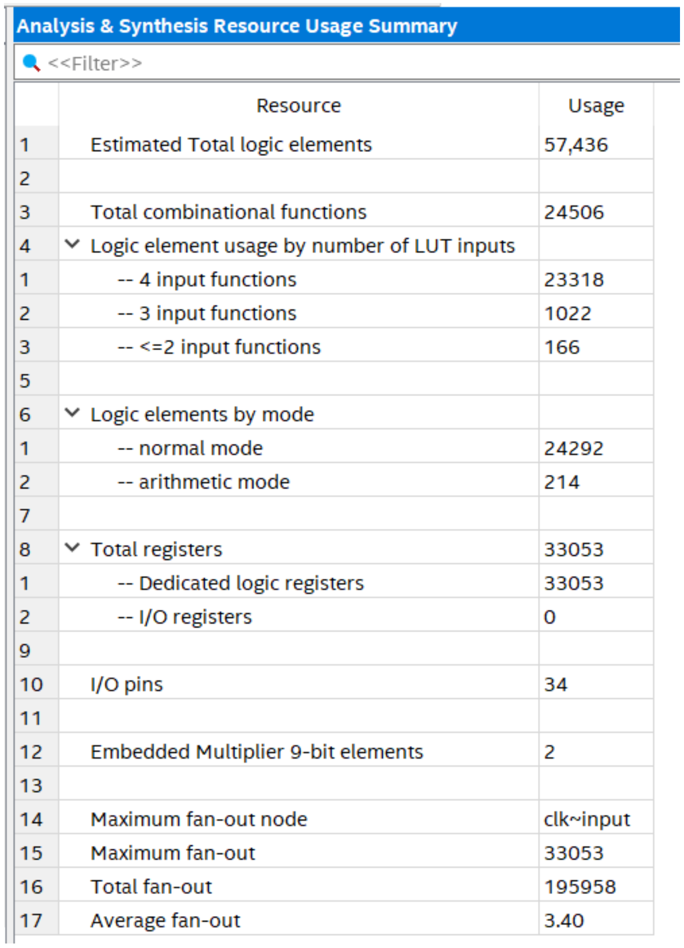


Flow Summary

<<Filter>>

Flow Status	Successful - Sun Oct 15 11:25:34 2023
Quartus Prime Version	17.1.0 Build 590 10/25/2017 SJ Lite Edition
Revision Name	new
Top-level Entity Name	single_cycle_mycode
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	42,190 / 114,480 (37 %)
Total registers	33053
Total pins	34 / 529 (6 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	2 / 532 (< 1 %)
Total PLLs	0 / 4 (0 %)

The resource usage is as follows:

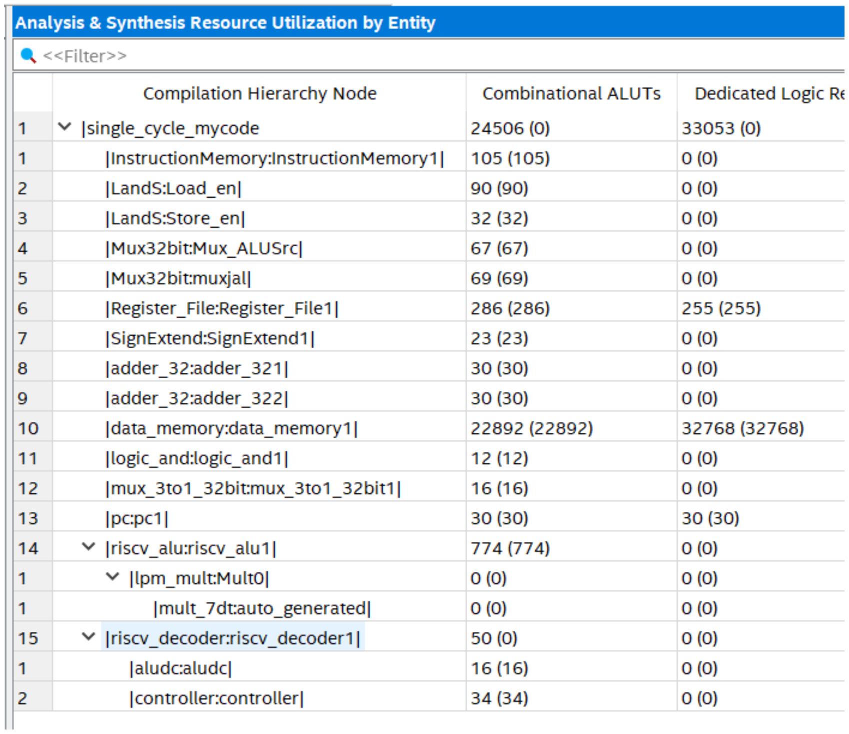


Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Estimated Total logic elements	57,436
2		
3	Total combinational functions	24506
4	Logic element usage by number of LUT inputs	
1	-- 4 input functions	23318
2	-- 3 input functions	1022
3	-- <=2 input functions	166
5		
6	Logic elements by mode	
1	-- normal mode	24292
2	-- arithmetic mode	214
7		
8	Total registers	33053
1	-- Dedicated logic registers	33053
2	-- I/O registers	0
9		
10	I/O pins	34
11		
12	Embedded Multiplier 9-bit elements	2
13		
14	Maximum fan-out node	clk~input
15	Maximum fan-out	33053
16	Total fan-out	195958
17	Average fan-out	3.40

Resource utilization summary:



Analysis & Synthesis Resource Utilization by Entity

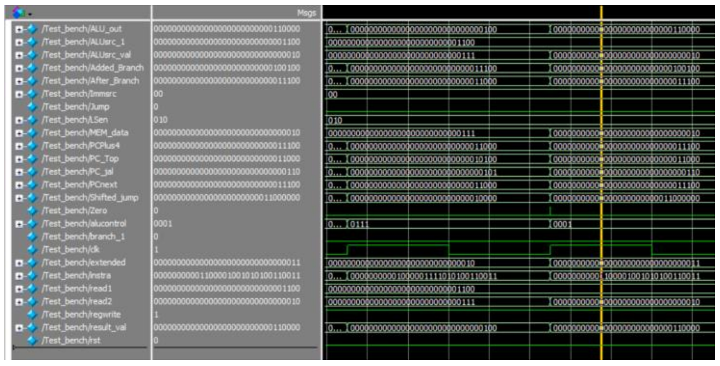
<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Re
1	|single_cycle_mycode	24506 (0)	33053 (0)
1	|InstructionMemory:InstructionMemory1|	105 (105)	0 (0)
2	|LandS:Load_en|	90 (90)	0 (0)
3	|LandS:Store_en|	32 (32)	0 (0)
4	|Mux32bit:Mux_ALUSrc|	67 (67)	0 (0)
5	|Mux32bit:muxjal|	69 (69)	0 (0)
6	|Register_File:Register_File1|	286 (286)	255 (255)
7	|SignExtend:SignExtend1|	23 (23)	0 (0)
8	|adder_32:adder_321|	30 (30)	0 (0)
9	|adder_32:adder_322|	30 (30)	0 (0)
10	|data_memory:data_memory1|	22892 (22892)	32768 (32768)
11	|logic_and:logic_and1|	12 (12)	0 (0)
12	|mux_3to1_32bit:mux_3to1_32bit1|	16 (16)	0 (0)
13	|pc:pc1|	30 (30)	30 (30)
14	|riscv_alu:riscv_alu1|	774 (774)	0 (0)
1	|lpm_mult:Mult0|	0 (0)	0 (0)
1	|mult_7dt:auto_generated|	0 (0)	0 (0)
15	|riscv_decoder:riscv_decoder1|	50 (0)	0 (0)
1	|aludc:aludc|	16 (16)	0 (0)
2	|controller:controller|	34 (34)	0 (0)

Testing and Simulation

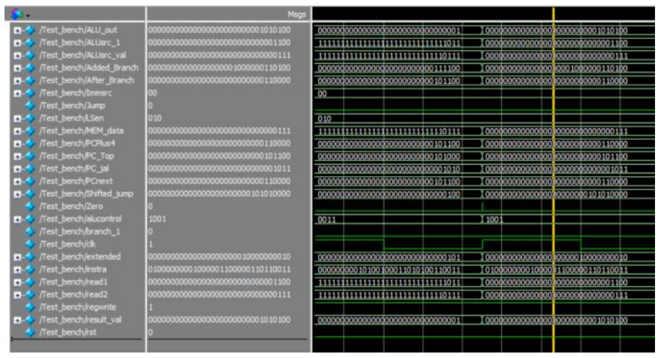
Simulating a RISC-V single-cycle processor is a crucial step in its development and validation. To accomplish this, we utilized the integrated ModelSim software within Quartus Prime. ModelSim is a powerful simulation tool that allows us to thoroughly test and verify the functionality of our processor design. Through this simulation process, we can evaluate the processor's performance, identify and resolve potential issues, and ensure that it adheres to the RISC-V architecture and our specific design requirements. This step is instrumental in confirming that the processor operates as intended and paves the way for further refinement and optimization.

Result Waves from the Modelsim:



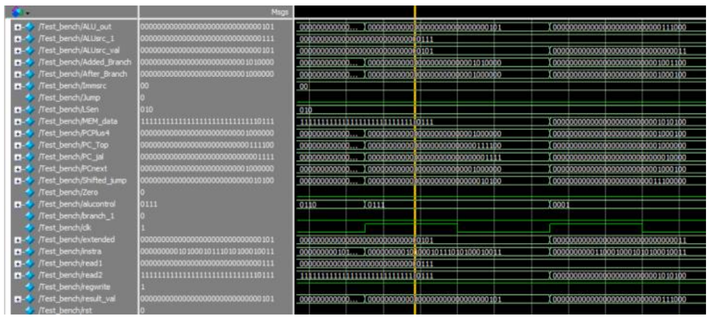
```
/Test_bench/ALU_out         00000000000000000000000000110000
/Test_bench/ALUsrc_1        00000000000000000000000000001100
/Test_bench/ALUsrc_val      00000000000000000000000000000010
/Test_bench/Added_Branch    00000000000000000000000000100100
/Test_bench/After_Branch    00000000000000000000000000011100
/Test_bench/Immsrc          00
/Test_bench/Jump            0
/Test_bench/LSen            010
/Test_bench/MEM_data        00000000000000000000000000000010
/Test_bench/PCPlus4         00000000000000000000000000011100
/Test_bench/PC_Top          00000000000000000000000000011000
/Test_bench/PC_jal          00000000000000000000000000000110
/Test_bench/PCnext          00000000000000000000000000011100
/Test_bench/Shifted_jump    00000000000000000000000011000000
/Test_bench/Zero            0
/Test_bench/alucontrol      0001
/Test_bench/branch_1        0
/Test_bench/clk             1
/Test_bench/extended        00000000000000000000000000000011
/Test_bench/instra          0000000001100001001010100110011
/Test_bench/read1           00000000000000000000000000001100
/Test_bench/read2           00000000000000000000000000000010
/Test_bench/regwrite        1
/Test_bench/result_val      00000000000000000000000000110000
/Test_bench/rst             0
```

R - Type



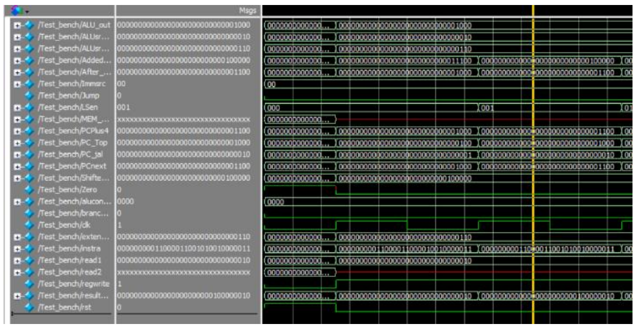
```
/Test_bench/ALU_out         00000000000000000000000001010100
/Test_bench/ALUsrc_1        00000000000000000000000000001100
/Test_bench/ALUsrc_val      00000000000000000000000000000111
/Test_bench/Added_Branch    00000000000000000001000000110100
/Test_bench/After_Branch    00000000000000000000000000110000
/Test_bench/Immsrc          00
/Test_bench/Jump            0
/Test_bench/LSen            010
/Test_bench/MEM_data        00000000000000000000000000000111
/Test_bench/PCPlus4         00000000000000000000000000110000
/Test_bench/PC_Top          00000000000000000000000000101100
/Test_bench/PC_jal          00000000000000000000000000001011
/Test_bench/PCnext          00000000000000000000000000110000
/Test_bench/Shifted_jump    00000000000000000000000101010000
/Test_bench/Zero            0
/Test_bench/alucontrol      1001
/Test_bench/branch_1        0
/Test_bench/clk             1
/Test_bench/extended        00000000000000000000100000000010
/Test_bench/instra          010000000010000011000000110110011
/Test_bench/read1           00000000000000000000000000001100
/Test_bench/read2           00000000000000000000000000000111
/Test_bench/regwrite        1
/Test_bench/result_val      00000000000000000000000001010100
/Test_bench/rst             0
```

Multiplication



```
/Test_bench/ALU_out         00000000000000000000000000000101
/Test_bench/ALUsrc_1        00000000000000000000000000000111
/Test_bench/ALUsrc_val      00000000000000000000000000000101
/Test_bench/Added_Branch    00000000000000000000000001010000
/Test_bench/After_Branch    00000000000000000000000001000000
/Test_bench/Immsrc          00
/Test_bench/Jump            0
/Test_bench/LSen            010
/Test_bench/MEM_data        11111111111111111111111111110111
/Test_bench/PCPlus4         00000000000000000000000001000000
/Test_bench/PC_Top          00000000000000000000000000111100
/Test_bench/PC_jal          00000000000000000000000000001111
/Test_bench/PCnext          00000000000000000000000001000000
/Test_bench/Shifted_jump    00000000000000000000000000010100
/Test_bench/Zero            0
/Test_bench/alucontrol      0111
/Test_bench/branch_1        0
/Test_bench/clk             1
/Test_bench/extended        00000000000000000000000000000101
/Test_bench/instra          00000000010100010111010100010011
/Test_bench/read1           00000000000000000000000000000111
/Test_bench/read2           11111111111111111111111111110111
/Test_bench/regwrite        1
/Test_bench/result_val      00000000000000000000000000000101
/Test_bench/rst             0
```

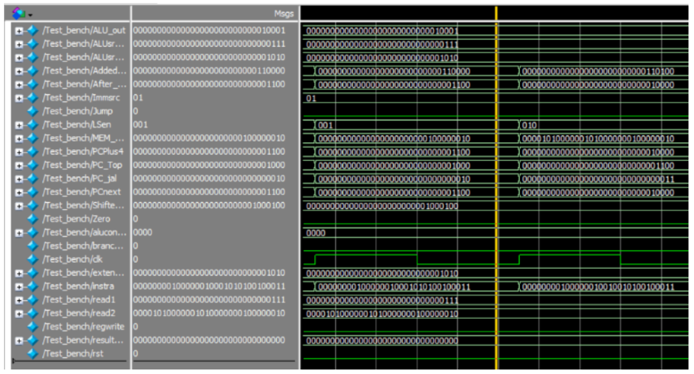
I-type



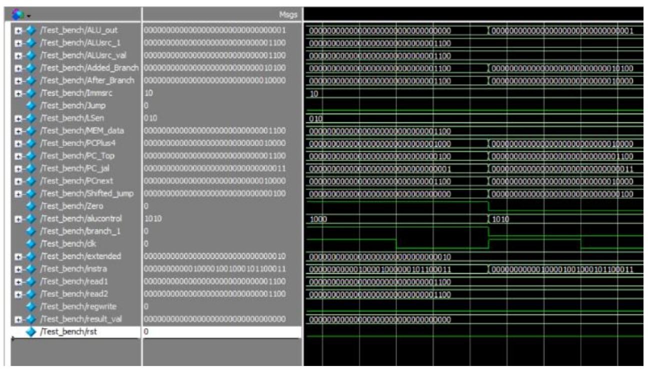
```
/Test_bench/ALU_out         00000000000000000000000000001000
/Test_bench/ALUsr...        00000000000000000000000000000010
/Test_bench/ALUsr...        00000000000000000000000000000110
/Test_bench/Added...        00000000000000000000000000100000
/Test_bench/After_...       00000000000000000000000000001100
/Test_bench/Immsrc          00
/Test_bench/Jump            0
/Test_bench/LSen            001
/Test_bench/MEM_...         xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/Test_bench/PCPlus4         00000000000000000000000000001100
/Test_bench/PC_Top          00000000000000000000000000001000
/Test_bench/PC_jal          00000000000000000000000000000010
/Test_bench/PCnext          00000000000000000000000000001100
/Test_bench/Shifte...       00000000000000000000000000100000
/Test_bench/Zero            0
/Test_bench/alucon...       0000
/Test_bench/branc...        0
/Test_bench/clk             1
/Test_bench/exten...        00000000000000000000000000000110
/Test_bench/instra          00000000011000011001010010000011
/Test_bench/read1           00000000000000000000000000000010
/Test_bench/read2           xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/Test_bench/regwrite        1
/Test_bench/result...       00000000000000000000000100000010
/Test_bench/rst             0
```

I-type

Load Type



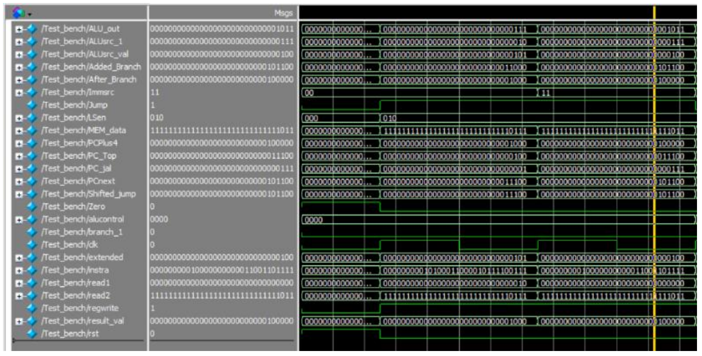
```
/Test_bench/ALU_out         00000000000000000000000000010001
/Test_bench/ALUsr...        00000000000000000000000000000111
/Test_bench/ALUsr...        00000000000000000000000000001010
/Test_bench/Added...        00000000000000000000000000110000
/Test_bench/After_...       00000000000000000000000000001100
/Test_bench/Immsrc          01
/Test_bench/Jump            0
/Test_bench/LSen            001
/Test_bench/MEM_...         00000000000000000000000100000010
/Test_bench/PCPlus4         00000000000000000000000000001100
/Test_bench/PC_Top          00000000000000000000000000001000
/Test_bench/PC_jal          00000000000000000000000000000010
/Test_bench/PCnext          00000000000000000000000000001100
/Test_bench/Shifte...       00000000000000000000000001000100
/Test_bench/Zero            0
/Test_bench/alucon...       0000
/Test_bench/branc...        0
/Test_bench/clk             0
/Test_bench/exten...        00000000000000000000000000001010
/Test_bench/instra          00000000100000010001010100100011
/Test_bench/read1           00000000000000000000000000000111
/Test_bench/read2           00001010000001010000000100000010
/Test_bench/regwrite        0
/Test_bench/result...       00000000000000000000000000000000
/Test_bench/rst             0
```



```
/Test_bench/ALU_out         00000000000000000000000000000001
/Test_bench/ALUsrc_1        00000000000000000000000000001100
/Test_bench/ALUsrc_val      00000000000000000000000000001100
/Test_bench/Added_Branch    00000000000000000000000000010100
/Test_bench/After_Branch    00000000000000000000000000010000
/Test_bench/Immsrc          10
/Test_bench/Jump            0
/Test_bench/LSen            010
/Test_bench/MEM_data        00000000000000000000000000001100
/Test_bench/PCPlus4         00000000000000000000000000010000
/Test_bench/PC_Top          00000000000000000000000000001100
/Test_bench/PC_jal          00000000000000000000000000000011
/Test_bench/PCnext          00000000000000000000000000010000
/Test_bench/Shifted_jump    00000000000000000000000000000100
/Test_bench/Zero            0
/Test_bench/alucontrol      1010
/Test_bench/branch_1        0
/Test_bench/clk             0
/Test_bench/extended        00000000000000000000000000000010
/Test_bench/instra          00000000000100001001000101100011
/Test_bench/read1           00000000000000000000000000001100
/Test_bench/read2           00000000000000000000000000001100
/Test_bench/regwrite        0
/Test_bench/result_val      00000000000000000000000000000000
/Test_bench/rst             0
```

Store Type

Branch Type



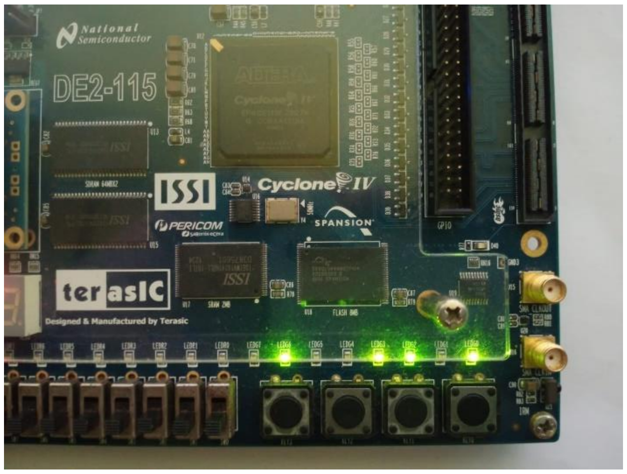
```
Msgs
/Test_bench/ALU_out          00000000000000000000000000001011
/Test_bench/ALUsrc_1         00000000000000000000000000000111
/Test_bench/ALUsrc_val       00000000000000000000000000000100
/Test_bench/Added_Branch     00000000000000000000000000101100
/Test_bench/After_Branch     00000000000000000000000000100000
/Test_bench/Immsrc           11
/Test_bench/Jump             1
/Test_bench/LSen             010
/Test_bench/MEM_data         11111111111111111111111111111011
/Test_bench/PCPlus4          00000000000000000000000000100000
/Test_bench/PC_Top           00000000000000000000000000011100
/Test_bench/PC_jal           00000000000000000000000000000111
/Test_bench/PCnext           00000000000000000000000000101100
/Test_bench/Shifted_jump     00000000000000000000000000101100
/Test_bench/Zero             0
/Test_bench/alucontrol       0000
/Test_bench/branch_1         0
/Test_bench/clk              0
/Test_bench/extended         00000000000000000000000000000100
/Test_bench/instra           00000000100000000000011001101111
/Test_bench/read1            00000000000000000000000000000000
/Test_bench/read2            11111111111111111111111111111011
/Test_bench/regwrite         1
/Test_bench/result_val       00000000000000000000000000100000
/Test_bench/rst              0
```

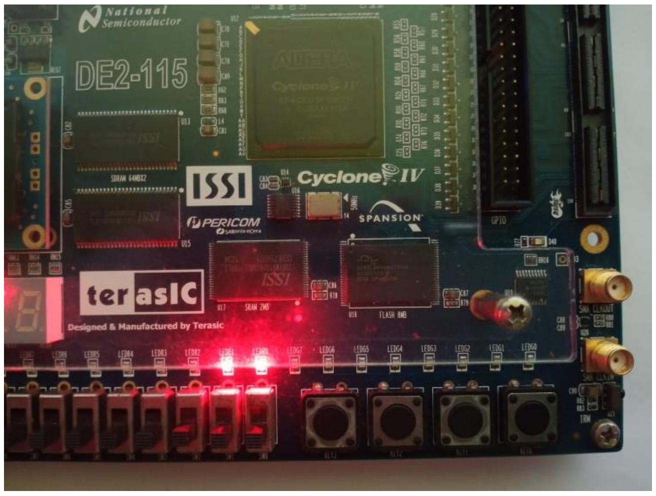
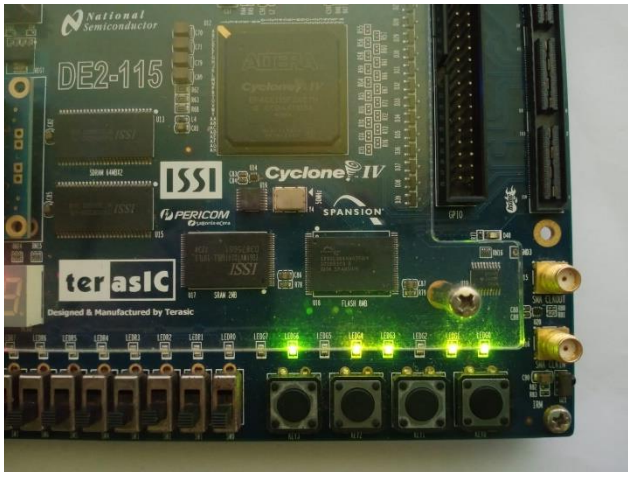
Jump Type

Demonstration

In our RISC-V single-cycle processor implementation, we've harnessed the versatility of the Cyclone IV EP4CE115F29C7 FPGA board to create a practical and interactive platform. This FPGA board provides us with an array of input options, including push buttons and switches, which serve as the means for providing input to our processor. Additionally, we've employed the board's resources such as LEDG and LEDR indicators to visually convey the results of our processor's operations, making the execution of instructions and their outcomes more tangible. This integration of hardware and software exemplifies the real-world applications and utility of our RISC-V processor within the context of the FPGA board, where it can be utilized for diverse computational tasks and demonstrations.

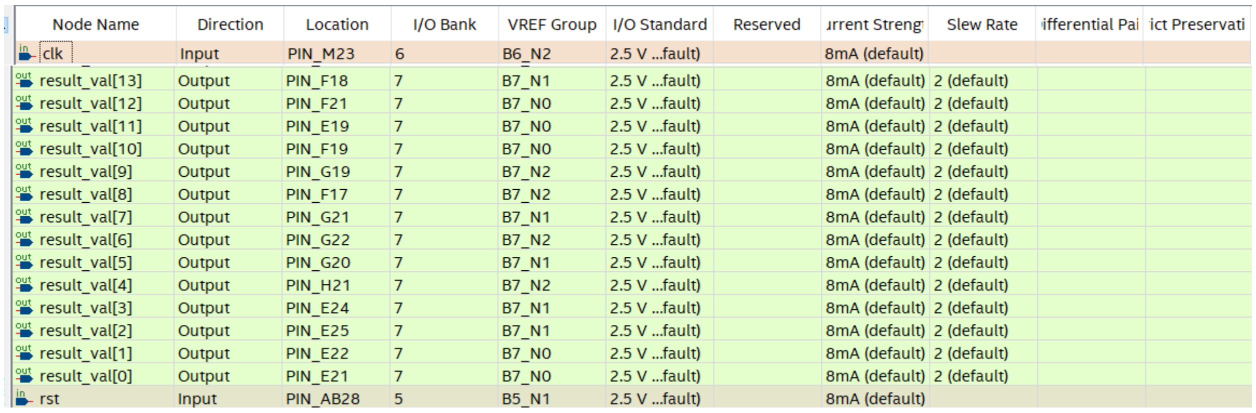
Example for implementing some instructions:



Pin planner for the FPGA Board

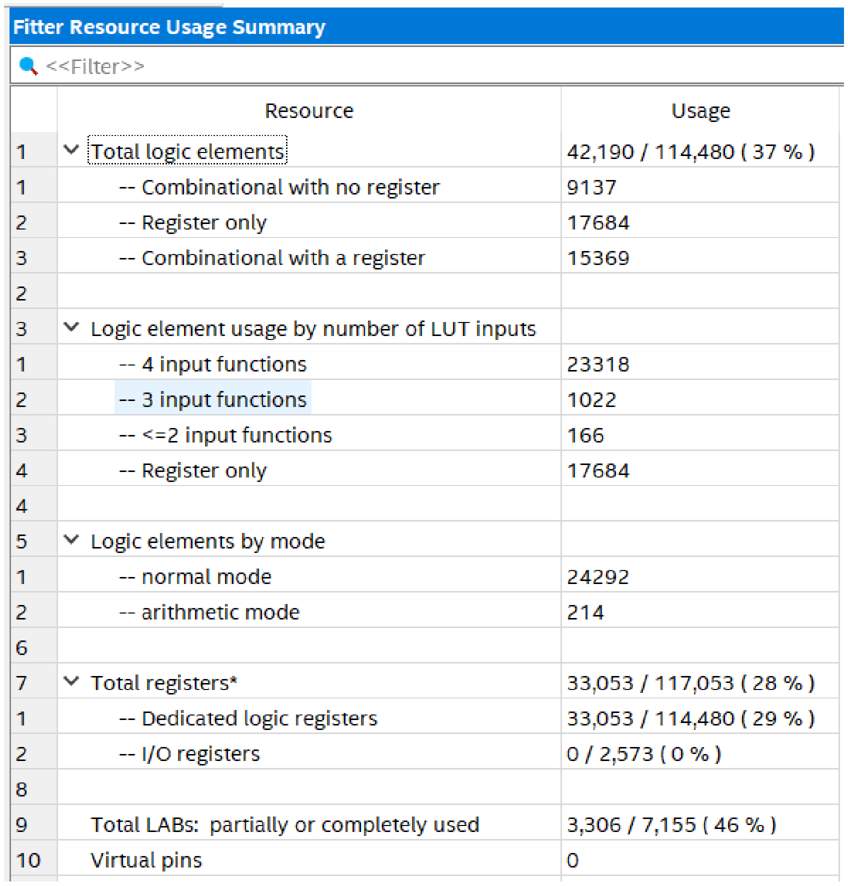
Push buttons: PIN_M23
Switch: PIN_AB288
LEDG [8:0]: LED indicators
LEDR [17:0]: LED indicators



Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard	Reserved	urrent Streng	Slew Rate	ifferential Pai	ict Preservati
clk	Input	PIN_M23	6	B6_N2	2.5 V ...fault)		8mA (default)			
result_val[13]	Output	PIN_F18	7	B7_N1	2.5 V ...fault)		8mA (default)	2 (default)		
result_val[12]	Output	PIN_F21	7	B7_N0	2.5 V ...fault)		8mA (default)	2 (default)		
result_val[11]	Output	PIN_E19	7	B7_N0	2.5 V ...fault)		8mA (default)	2 (default)		
result_val[10]	Output	PIN_F19	7	B7_N0	2.5 V ...fault)		8mA (default)	2 (default)		
result_val[9]	Output	PIN_G19	7	B7_N2	2.5 V ...fault)		8mA (default)	2 (default)		
result_val[8]	Output	PIN_F17	7	B7_N2	2.5 V ...fault)		8mA (default)	2 (default)		
result_val[7]	Output	PIN_G21	7	B7_N1	2.5 V ...fault)		8mA (default)	2 (default)		
result_val[6]	Output	PIN_G22	7	B7_N2	2.5 V ...fault)		8mA (default)	2 (default)		
result_val[5]	Output	PIN_G20	7	B7_N1	2.5 V ...fault)		8mA (default)	2 (default)		
result_val[4]	Output	PIN_H21	7	B7_N2	2.5 V ...fault)		8mA (default)	2 (default)		
result_val[3]	Output	PIN_E24	7	B7_N1	2.5 V ...fault)		8mA (default)	2 (default)		
result_val[2]	Output	PIN_E25	7	B7_N1	2.5 V ...fault)		8mA (default)	2 (default)		
result_val[1]	Output	PIN_E22	7	B7_N0	2.5 V ...fault)		8mA (default)	2 (default)		
result_val[0]	Output	PIN_E21	7	B7_N0	2.5 V ...fault)		8mA (default)	2 (default)		
rst	Input	PIN_AB28	5	B5_N1	2.5 V ...fault)		8mA (default)			

Performance Evaluation

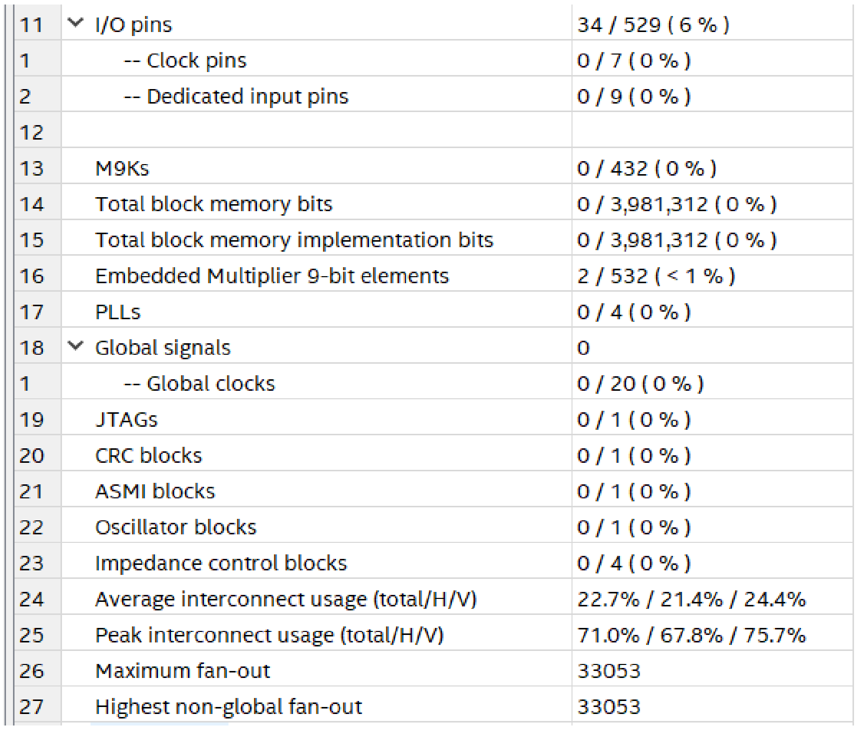
Resource Usage



Fitter Resource Usage Summary

<<Filter>>

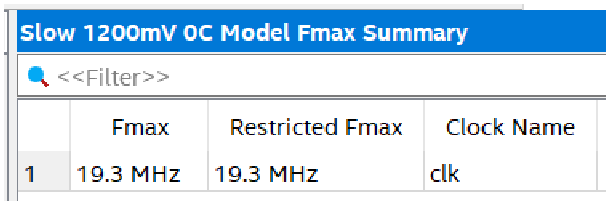
	Resource	Usage
1	Total logic elements	42,190 / 114,480 (37 %)
1	-- Combinational with no register	9137
2	-- Register only	17684
3	-- Combinational with a register	15369
2		
3	Logic element usage by number of LUT inputs	
1	-- 4 input functions	23318
2	-- 3 input functions	1022
3	-- <=2 input functions	166
4	-- Register only	17684
4		
5	Logic elements by mode	
1	-- normal mode	24292
2	-- arithmetic mode	214
6		
7	Total registers*	33,053 / 117,053 (28 %)
1	-- Dedicated logic registers	33,053 / 114,480 (29 %)
2	-- I/O registers	0 / 2,573 (0 %)
8		
9	Total LABs: partially or completely used	3,306 / 7,155 (46 %)
10	Virtual pins	0



11	I/O pins	34 / 529 (6 %)
1	-- Clock pins	0 / 7 (0 %)
2	-- Dedicated input pins	0 / 9 (0 %)
12		
13	M9Ks	0 / 432 (0 %)
14	Total block memory bits	0 / 3,981,312 (0 %)
15	Total block memory implementation bits	0 / 3,981,312 (0 %)
16	Embedded Multiplier 9-bit elements	2 / 532 (< 1 %)
17	PLLs	0 / 4 (0 %)
18	Global signals	0
1	-- Global clocks	0 / 20 (0 %)
19	JTAGs	0 / 1 (0 %)
20	CRC blocks	0 / 1 (0 %)
21	ASMI blocks	0 / 1 (0 %)
22	Oscillator blocks	0 / 1 (0 %)
23	Impedance control blocks	0 / 4 (0 %)
24	Average interconnect usage (total/H/V)	22.7% / 21.4% / 24.4%
25	Peak interconnect usage (total/H/V)	71.0% / 67.8% / 75.7%
26	Maximum fan-out	33053
27	Highest non-global fan-out	33053

Timing Analysis

The maximum clock rates under 0C is as follows



Slow 1200mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name
1	19.3 MHz	19.3 MHz	clk

References

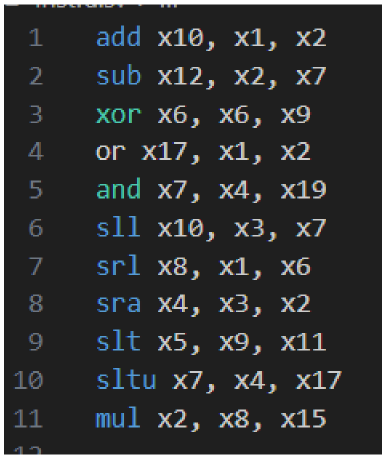
[1] Patterson, D.A. and Hennessy, J.L. (2021) *Computer Organization and Design RISC-V edition: The hardware software interface*. Cambridge, MA: The Morgan Kaufmann. http://home.ustc.edu.cn/~louwenqi/reference_books_tools/Computer%20Organization%20and%20Design%20RISC-V%20edition.pdf

[2] ALTERA DE2_115_User_manual DE2_115_User_manual_2013 (terasic.com.tw)

Appendix

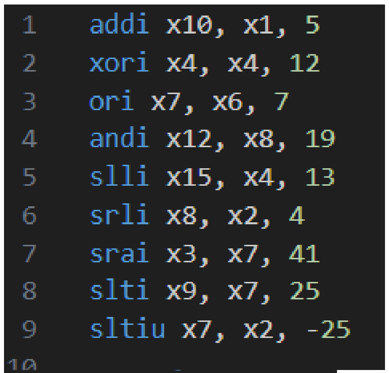
The assembly codes for testing each instruction type

Assembly code used for testing R-Type instructions

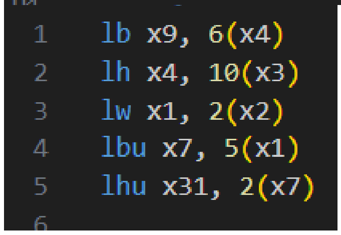


```
add x10, x1, x2
sub x12, x2, x7
xor x6, x6, x9
or x17, x1, x2
and x7, x4, x19
sll x10, x3, x7
srl x8, x1, x6
sra x4, x3, x2
slt x5, x9, x11
sltu x7, x4, x17
mul x2, x8, x15
```

Assembly code used for testing I-Type instructions

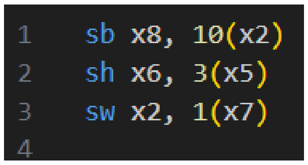


```
addi x10, x1, 5
xori x4, x4, 12
ori x7, x6, 7
andi x12, x8, 19
slli x15, x4, 13
srli x8, x2, 4
srai x3, x7, 41
slti x9, x7, 25
sltiu x7, x2, -25
```



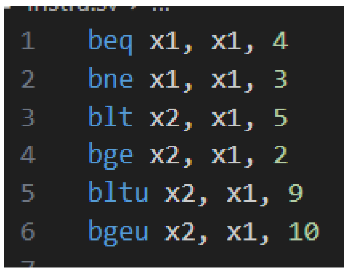
```
lb x9, 6(x4)
lh x4, 10(x3)
lw x1, 2(x2)
lbu x7, 5(x1)
lhu x31, 2(x7)
```

Assembly code used for testing S-Type instructions



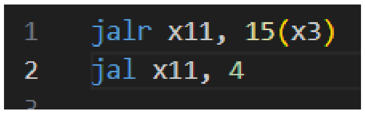
```
sb x8, 10(x2)
sh x6, 3(x5)
sw x2, 1(x7)
```

Assembly code used for testing B-Type instructions



```
beq x1, x1, 4
bne x1, x1, 3
blt x2, x1, 5
bge x2, x1, 2
bltu x2, x1, 9
bgeu x2, x1, 10
```

Assembly code used for testing J-Type instructions



```
jalr x11, 15(x3)
jal x11, 4
```