16-bit RISC ISA Design and CPU Implementation

CPE 531 Final Technical Report

Nabin Kumar Singh

December 2025

Contents

1 Introduction

This project presents the design and implementation of a custom 16-bit RISC instruction set architecture and a working single-cycle processor that executes that ISA. The design follows a simple and uniform structure so that the hardware can be implemented clearly in SystemVerilog and verified through simulation. The overall goal is to create a small but complete processor that supports the required C programming constructs while keeping the hardware minimal.

1.1 Design Philosophy

The ISA is designed around the same ideas emphasized in class when studying the MIPS architecture, but scaled down to a 16-bit environment. The basic structure is similar: fixed-length instructions, register-to-register ALU operations, a load–store model for memory access, and a straightforward branching mechanism. Register fields, opcode widths, and immediate lengths are reduced to fit a single 16-bit instruction word, but the style and behavior remain close to what was taught in lecture.

The project follows common RISC principles:

- simple, fixed-size instructions that are easy to decode,
- operations that execute in a single clock cycle,
- a small general-purpose register file for fast operand access,
- separation of memory and arithmetic operations (load–store model),
- minimal but sufficient instruction set to keep hardware small.

1.2 Project Scope

All instructions are encoded in a single 16-bit word, and the processor operates on 16-bit two's-complement integers. The register file contains eight registers (R0–R7), where R0 is hardwired to zero and R7 is reserved as a link register for subroutine calls. The memory system uses a linear 1 Kiword address space and is word-addressable. A single-cycle datapath is used so every instruction completes in one clock cycle, except `HALT`, which stops the PC.

The remainder of this report describes the ISA, the register set, the different instruction formats, justification for each instruction, HDL implementation of the datapath and control logic, and simulation results that verify correct execution of several test programs.

2 Architectural Overview

This processor is built as a simple 16-bit single-cycle CPU. Every instruction is fetched, decoded, executed, and written back within one clock cycle. The design contains the core

components needed for a basic RISC processor: a program counter, instruction memory, a small register file, a combinational ALU, data memory, and a control unit that decodes each instruction.

2.1 System Specifications

- **Word size:** 16 bits (two's-complement signed integers)
- **Instruction size:** fixed 16-bit instruction word
- **Registers:** eight 16-bit registers (R0–R7), with R0 fixed to zero and R7 used as a link register
- **Memory:** 1 Kiword linear memory space, word-addressable
- **Instruction types:** R-type, I-type, LI-type (LHI/LLI), and J-type
- **Execution model:** single-cycle datapath
- **Control flow support:** conditional branches (BEQ, BNE, BLT, BGE), CALL, RET, and HALT

The CPU uses a direct and predictable control path, making it easier to implement and debug. Immediate decoding, ALU control, register selection, and PC update logic are fully combinational and depend only on the opcode and funct fields of the current instruction.

3 Memory Architecture

The processor uses a simple and linear 16-bit memory system. The machine addresses 1024 words (1 Kiword), where each address refers to a 16-bit word. There is no byte addressing in this design. Two memories are used in the hardware implementation: an instruction memory (ROM) and a data memory (RAM). Both memories are word-addressed, and the program counter supplies a 16-bit address whose lower 10 bits select one of the 1024 words. The data memory uses a simple synchronous interface. Both reads and writes occur on the rising edge of the clock. A load instruction places the effective address on the memory port, and the corresponding word is returned on the next clock edge. This timing model keeps the hardware simple and matches the behavior of the synthesizable SystemVerilog implementation.

Only the `LD` and `ST` instructions access memory. All other instructions work entirely inside the register file. Load and store addresses are computed as

$$\text{effective address} = Rs + \text{sign-extend(imm6)},$$

where the offset is interpreted in word units.

3.1 Register Roles

The processor contains eight 16-bit general-purpose registers (R0–R7). These are used to hold intermediate values, function parameters, return values, loop indices, and memory addresses during program execution.

- **R0**: Hardwired to zero. Reads always return 0 and writes are ignored.
- **R1–R6**: General-purpose registers used for arithmetic, temporary values, pointer arithmetic, array indexing, and comparison operands.
- **R7**: Link register (LR). Used by `CALL` to store the return address (PC+1) and by `RET` to return control to the caller.

This register organization matches the needs of C programs and simplifies the hardware design by using a fixed link register instead of a stack-based return address.

3.2 Linear Addressing Model

The memory system uses a linear addressing model: the data memory is a contiguous array of 1024 words, and each 16-bit address maps directly to a specific word without segmentation or paging. The lower 10 bits of the ALU result select the word address, ensuring simple and predictable memory access.

4 Instruction Encoding

4.1 Summary of Formats

All instructions in the ISA are encoded as a single 16-bit word. Four instruction formats are used: R-type, I-type, LI-type (LHI/LLI), and J-type. The formats are kept simple and uniform so decoding can be performed directly from the opcode and field positions without additional microcoding.

4.2 Field Layout Tables

Bits	15–12	11–9	8–6	5–3	2–0
Field	opcode	rs	rt	rd	funct3
Width	4	3	3	3	3

Table 1: R-type instruction encoding.

Bits	15–12	11–9	8–6	5–0
Field	opcode	rs	rt	imm6
Width	4	3	3	6

Table 2: I-type instruction encoding.

Bits	15–12	11–9	8–1	0
Field	opcode	rt	imm8	unused
Width	4	3	8	1

Table 3: LI-type instruction encoding (LHI and LLI).

Bits	15–12	11–0
Field	opcode	target
Width	4	12

Table 4: J-type instruction encoding (CALL, RET, HALT).

4.3 Encoding Examples

This subsection presents several representative examples that show how assembly instructions translate into their 16-bit machine encoding. These examples illustrate the use of each instruction format: R-type, I-type, LI-type, and J-type.

Example 1: R-type Instruction ADD R3, R1, R2

Opcode (0000)	R-type ALU operation
rs = 001	R1
rt = 010	R2
rd = 011	R3
funct3 = 000	ADD

Machine code (binary):

0000 001 010 011 000

Machine code (hex):

$0x0298$

Example 2: I-type Instruction ADDI R4, R1, 5

Opcode (0100)	ADDI
rs = 001	R1
rt = 100	R4
imm6 = 000101	5 (sign-extended)

Machine code (binary):

0100 001 100 000101

Machine code (hex):

$0x4305$

Example 3: LI-type Instruction LHI R2, 0x3A

Opcode (0010)	LHI
rt = 010	R2
imm8 = 00111010	0x3A
unused = 0	fixed

Machine code (binary):

0010 010 00111010 0

Machine code (hex):

$0x2474$

Example 4: J-type Instruction CALL 0x120

Opcode (1101)	CALL
target = 000100100000	0x120

Machine code (binary):

1101 000100100000

Machine code (hex):

$0xD120$

5 Instruction Semantics and Rationale

This section describes the behavior of every instruction supported by the 16-bit ISA. Each subsection groups instructions by function (ALU, immediate, memory, branches, and control flow). For each instruction, the mnemonic, operand format, opcode, and behavioral meaning are summarized. The purpose of each instruction class is also explained to justify its inclusion in the ISA.

5.1 ALU Instructions (R-type)

The ALU instructions perform register–to–register operations. All R-type instructions share the same 4-bit opcode, while the `funct3` field selects the specific ALU operation. This reduces opcode usage while preserving a complete arithmetic and logical instruction set.

Opcode	funct3	Mnemonic	Meaning
0000	000	ADD	$Rd \leftarrow Rs + Rt$ (signed addition)
0000	001	SUB	$Rd \leftarrow Rs - Rt$ (signed subtraction)
ALU_L: logical/shift operations			
0001	000	AND	bitwise AND
0001	001	OR	bitwise OR
0001	010	XOR	bitwise XOR
0001	011	NOT	bitwise inversion: $Rd \leftarrow \neg Rs$
0001	100	SLL	logical left shift: $Rd \leftarrow Rs \ll (Rt \ \& \ 4'b1111)$
0001	101	SRL	logical right shift: $Rd \leftarrow Rs \gg (Rt \ \& \ 4'b1111)$

Table 5: R-type ALU operations.

Design Rationale. Using a single opcode for each ALU class (arithmetic and logical/shift) keeps the ISA within the 16-opcode limit while still providing a complete set of useful operations. Shifts use the lower four bits of `Rt` to select a shift amount in the range 0–15, matching the 16-bit datapath width and simplifying hardware. The use of `funct3` matches designs commonly taught in lecture, while being compact enough for a 16-bit ISA.

Example: SLL Operation.

$$\text{SLL R1, R2, R3} \quad \Rightarrow \quad R1 \leftarrow R2 \ll (R3 \ \& \ 4'b1111)$$

If $R2 = 0x0003$ and $R3 = 0x0004$, then the shift amount is 4, yielding:

$$R1 = 0x0003 \ll 4 = 0x0030.$$

5.2 Immediate and load-immediate instructions

Immediate instructions support arithmetic on small constants and simple bitwise manipulation. These instructions follow the I-type format where the destination register is `Rt`.

Opcode	Mnemonic	Meaning
0100	ADDI	$Rt \leftarrow Rs + \text{signext}(imm6)$
0101	ANDI	$Rt \leftarrow Rs$ & $\text{zeroext}(imm6)$
0110	ORI	$Rt \leftarrow Rs \mid \text{zeroext}(imm6)$
0010	LHI	$Rt \leftarrow (imm8 \ll 8)$
0011	LLI	$Rt \leftarrow (Rt[15:8] \cdot 256) \mid imm8$

Table 6: Immediate and load-immediate instructions.

Design Rationale. ADDI enables simple arithmetic and array indexing using small offsets. ANDI and ORI allow efficient bit masking and flag operations without requiring a constant in memory. The pair LHI/LLI provides a lightweight mechanism for constructing full 16-bit constants in two instructions, which is critical in a 16-bit ISA with limited immediate size.

5.3 Memory Instructions

The ISA uses a pure load–store architecture: only `LD` and `ST` access data memory. Effective addresses are computed as:

$$EA = Rs + \text{signext}(imm6).$$

Opcode	Mnemonic	Meaning
0111	LD	$Rt \leftarrow MEM[Rs + \text{signext}(imm6)]$
1000	ST	$MEM[Rs + \text{signext}(imm6)] \leftarrow Rt$

Table 7: Load and store instructions.

Design Rationale. Restricting memory access to LD and ST simplifies the datapath and keeps the ISA aligned with standard RISC practice. Word-addressing avoids byte-enable wiring and matches the 1 Kiword memory design used in hardware.

5.4 Branch Instructions

Branches compare two registers and compute a PC-relative offset. All branch targets lie within $[-32, +31]$ instructions because of the 6-bit signed immediate.

Opcode	Mnemonic	Meaning
1001	BEQ	if $(Rs == Rt)$: $PC \leftarrow PC + 1 + imm6$
1010	BNE	if $(Rs \neq Rt)$: $PC \leftarrow PC + 1 + imm6$
1011	BLT	if $(Rs < Rt)$ (signed): $PC \leftarrow PC + 1 + imm6$
1100	BGE	if $(Rs \geq Rt)$ (signed): $PC \leftarrow PC + 1 + imm6$

Table 8: Conditional branch instructions.

Design Rationale. These four branches directly support all relational operators used in C (==, !=, <, >=), enabling efficient compilation of if–else, while, and for statements without requiring complex comparison hardware.

5.5 Control-Flow Instructions: CALL, RET, HALT

Jump and subroutine support use a 12-bit absolute target. Register R7 acts as the link register (LR) for procedure calls.

Opcode	Mnemonic	Meaning
1101	CALL	$R7 \leftarrow PC + 1; \quad PC \leftarrow target$
1110	RET	$PC \leftarrow R7$
1111	HALT	Stop execution (PC disabled)

Table 9: Control-flow and program termination instructions.

Design Rationale. CALL and RET provide minimal but complete subroutine support, enabling function calls in compiled C code. HALT is included to simplify simulation and to mark program completion explicitly.

5.6 Why Each Instruction Exists

Every instruction in the ISA was chosen to support a minimal but complete set of operations needed for C programs while keeping the hardware simple enough for a single-cycle implementation. The following list summarizes the purpose of each instruction class.

ALU Instructions (ADD, SUB, AND, OR, XOR, NOT, SLL, SRL) These operations form the core of integer arithmetic and bit manipulation. Addition and subtraction are required for expression evaluation, array indexing, pointer arithmetic, and function argument updates. Logical operations support masking, clearing, and combining bit fields. The shift operations are needed for multiplying or dividing by powers of two and for efficient bit manipulation. All of these fit naturally into an R-type format and re-use the same ALU hardware.

Immediate Instructions (ADDI, ANDI, ORI) C programs frequently operate on small constants, index offsets, and loop increments. An immediate add instruction avoids loading constants from memory and is essential for implementing loops and conditionals. ANDI and ORI allow bit masking and flag manipulation without requiring extra registers.

Load and Store Instructions (LD, ST) A load–store architecture keeps the ALU simple and restricts memory access to two dedicated instructions. LD reads a 16-bit value from memory, and ST writes a register value back. These instructions are necessary for accessing arrays, passing by reference, and maintaining local variables in memory.

Branch Instructions (BEQ, BNE, BLT, BGE) Control flow in C relies heavily on comparisons. Equality and inequality tests are used in conditionals, loops, and function return checks. Signed relational branches such as `BLT` and `BGE` allow implementation of the full set of C comparisons without additional ALU instructions. All branches compute their target as `PC + 1 + offset`, enabling compact loop structures.

LI-Type Instructions (LHI, LLI) Many programs require loading full 16-bit constants. Since the instruction word can only hold an 8-bit literal, the two-part LHI/LLI mechanism allows the upper and lower bytes of a constant to be written independently. This avoids adding a new instruction format while keeping the hardware trivial.

CALL and RET These instructions provide subroutine support using a fixed link register (R7). `CALL` stores `PC+1` in R7 and jumps to the target address, while `RET` restores the PC from R7. This simple calling convention avoids stack hardware and is sufficient for C function calls with arguments passed in registers.

HALT The HALT instruction cleanly stops the processor by disabling PC updates. It is necessary for simulation control and marks the end of a program.

Overall, each instruction was included because it directly enables essential C constructs (arithmetic expressions, branching, loops, and functions) while keeping the ISA compact and the hardware easy to implement in a single cycle.

6 C Construct Support

This section illustrates how common C language constructs can be translated into the 16-bit ISA. The goal is to show that the instruction set is sufficient to implement arithmetic expressions, comparisons, control flow, loops, and function calls using straightforward sequences of instructions.

In the examples below, variables are assumed to be stored in registers (`R1`, `R2`, `R3`, ...), and memory locations are used only when needed to model C variables or arrays.

6.1 Arithmetic Expressions

Basic integer expressions in C map directly onto the ALU and immediate instructions of the ISA.

Example 1: Simple Addition

C code:

```
x = a + b;
```

Assume:

- `a` is in register R1,
- `b` is in register R2,
- `x` will be written to register R3.

Assembly:

```
ADD R3, R1, R2    ; R3 = a + b
```

This uses the R-type `ADD` instruction and performs a single-cycle register-to-register addition.

Example 2: Addition with a Small Constant

C code:

```
x = a + 5;
```

Assume:

- `a` is in R1,
- `x` will be stored in R2.

Assembly:

```
ADDI R2, R1, 5    ; R2 = a + 5
```

Here `ADDI` uses the 6-bit signed immediate to add a small constant without loading it from memory.

Example 3: Bitwise Masking

C code:

```
x = a & 0x000F;
```

Assume:

- `a` is in R1,
- `x` is written to R2.

Assembly:

```
ANDI R2, R1, 0x0F    ; R2 = a & 0x000F
```

The `ANDI` instruction zero-extends the 6-bit immediate and applies a bit mask directly to the register value.

Example 4: Shift Expression

C code:

```
x = a << 3;
```

Assume:

- `a` is in R1,
- `x` is written to R2.

Assembly:

```
ADDI R3, R0, 3        ; R3 = 3 (shift amount)
SLL  R2, R1, R3       ; R2 = a << 3
```

The shift amount is placed in a register and masked internally by the ALU, so any value from 0 to 15 can be used as a valid shift distance.

6.2 Relational Operators

Comparison operators C such as `==`,! =, `<`, and¿ `=` are implemented using the branch instructions `BEQ`, `BNE`, `BLT` and `BGE`. These branches compare two registers and update the program counter relative to the current instruction when the condition is satisfied.

All "unconditional jumps" in the following examples are implemented using:

```
BEQ R0, R0, offset
```

since register `R0` is hardwired to zero, making the branch always taken.

Registers used in examples:

- `R1`: value of `a`
- `R2`: value of `b`
- `R3`: result `x`

Example 1: Equality Comparison

C code:

```
if (a == b) {
    x = 1;
} else {
    x = 0;
}
```

Assembly:

```
BEQ   R1, R2, 2        ; branch taken  → jump to eq_true
ADDI  R3, R0, 0        ; branch not taken → x = 0
BEQ   R0, R0, 2        ; unconditional jump → skip true block
eq_true:
ADDI  R3, R0, 1        ; x = 1
eq_end:
```

Example 2: Less-Than Comparison

C code:

```
if (a < b) {
    x = 1;
} else {
    x = 0;
}
```

Assembly:

```
BLT   R1, R2, 2        ; branch taken → jump to lt_true
ADDI  R3, R0, 0        ; branch not taken → x = 0
BEQ   R0, R0, 2        ; unconditional jump
lt_true:
ADDI  R3, R0, 1        ; x = 1
lt_end:
```

Example 3: Comparison as an Integer Expression

C code:

```
x = (a >= b);
```

Assembly:

```
ADDI  R3, R0, 0        ; default: x = 0
BGE   R1, R2, 2        ; branch taken → set x = 1
BEQ   R0, R0, 2        ; unconditional jump
ge_true:
ADDI  R3, R0, 1        ; x = 1
ge_end:
```

Additional Relational Operators

To complete the set of C relational operators, the remaining cases can be implemented using existing branch instructions and simple operand reversal.

Not Equal (!=) The `BNE` instruction directly supports the `!=` operator.
C:

```
if (a != b) { ... }
```

Assembly:

```
BNE  R1, R2, offset    ; branch if a != b
```

Greater Than: The ISA does not include a dedicated "greater than" instruction, but the condition `a > b` is equivalent to `b < a`. Therefore, the `BLT` instruction can be used simply by swapping operand order, just like in the classical MIPS.
C:

```
if (a > b) { ... }
```

Assembly:

```
BLT  R2, R1, offset    ; branch if b < a  → a > b
```

Less Than or Equal (¡=) The condition `a <= b` is logically equivalent to `a < b` OR `a == b`. Using the `BGE` instruction with operands reversed provides a compact encoding:
C:

```
if (a <= b) { ... }
```

Assembly:

```
BGE  R2, R1, offset    ; branch if b >= a  → a <= b
```

These patterns demonstrate how the ISA supports all relational operators required by compiled C code.

6.3 If–Else Constructs

C `if--else` statements rely on evaluating a relational expression and then executing one of two blocks depending on whether the branch condition is satisfied. In this ISA, conditional behavior is implemented with the branch instructions (`BEQ`, `BNE`, `BLT`, `BGE`) followed by an unconditional jump using `BEQ R0, R0, offset`.

The general pattern is:

```
if (condition) {
    block_true
} else {
    block_false
}
```

Compilation pattern:

```
BRANCH condition, offset_to_true      ; branch taken → jump to true block
block_false                           ; executed when branch not taken
BEQ R0, R0, offset_to_end             ; unconditional jump to end
true_block:
block_true
end_if:
```

Example 1: Simple If–Else (Equality)

C code:

```
if (a == b) {
    x = 1;
} else {
    x = 0;
}
```

Assembly:

```
BEQ   R1, R2, 2        ; branch taken  → jump to if_true
ADDI  R3, R0, 0        ; branch not taken → x = 0
BEQ   R0, R0, 2        ; unconditional jump → skip true block
if_true:
ADDI  R3, R0, 1        ; x = 1
if_end:
```

Example 2: Less-Than Condition

C code:

```
if (a < b) {
    y = 4;
} else {
    y = 7;
}
```

Assembly:

```
BLT   R1, R2, 2        ; branch taken → jump to lt_true
ADDI  R4, R0, 7        ; y = 7  (false block)
BEQ   R0, R0, 2        ; unconditional jump
lt_true:
ADDI  R4, R0, 4        ; y = 4  (true block)
lt_end:
```

Example 3: Nested If–Else

C code:

```
if (a < b) {
    if (c == d) {
        z = 1;
    } else {
        z = 2;
    }
} else {
    z = 3;
}
```

Assembly:

```
BLT   R1, R2, 4        ; branch taken → enter first true block
ADDI  R5, R0, 3        ; z = 3  (outer false block)
BEQ   R0, R0, 6        ; unconditional jump → end_if

; inner if
inner_if:
BEQ   R3, R4, 2        ; c == d → jump to inner_true
ADDI  R5, R0, 2        ; z = 2  (inner false)
BEQ   R0, R0, 2        ; unconditional → skip inner true
```

```
inner_true:
ADDI  R5, R0, 1        ; z = 1
inner_end:
; continue outer true block
if_end:
```

These examples demonstrate how nested and multi-block C control structures map cleanly onto branch and offset-based control flow in the 16-bit ISA.

6.4 While Loops

A `while` loop repeatedly executes a block of code as long as a relational condition remains true. The ISA implements this using a branch at the top of the loop and an unconditional jump at the end of the loop body.

C code:

```
while (a < b) {
    a = a + 1;
}
```

Assume:

- `a` in R1
- `b` in R2

Assembly:

```
loop_start:
BLT   R1, R2, 2        ; branch taken → enter loop body
BEQ   R0, R0, 4        ; branch not taken → exit loop
ADDI  R1, R1, 1        ; a = a + 1
BEQ   R0, R0, -4       ; unconditional jump → loop_start
loop_end:
```

This structure mirrors the standard compiler lowering pattern: evaluate the condition → execute body → jump back → repeat until false.

6.5 For Loops

A `for` loop combines initialization, a loop condition, and an update operation. In this ISA, the loop is implemented using a branch at the top of the loop and an unconditional branch at the bottom.

C code:

```
for (i = 0; i < n; i++) {
    sum = sum + i;
}
```

Assume:

- i in R1
- n in R2
- sum in R3

Assembly:

```
ADDI  R1, R0, 0        ; i = 0
for_start:
BLT   R1, R2, 2        ; branch taken → enter loop body
BEQ   R0, R0, 5        ; exit loop
ADD   R3, R3, R1       ; sum += i
ADDI  R1, R1, 1        ; i++
BEQ   R0, R0, -5       ; repeat loop
for_end:
```

6.6 Functions and Parameter Passing

The ISA supports subroutines through the `CALL` and `RET` instructions. Register R7 is used as the link register (LR). Arguments are passed in general- purpose registers, and return values are placed in a designated register.

Call by Value

C code:

```
int add(int a, int b) { return a + b;}
x = add(u, v);
```

Assume:

- a in R1, b in R2
- return value in R3

Function:

```
add:
ADD   R3, R1, R2       ; result = a + b
RET
```

Caller:

```
CALL  add              ; R7 = PC+1, jump to function
; R3 now holds x
```

Call by Reference

A pointer or array element is passed by supplying its address in a register. The callee uses LD and ST to access memory indirectly.

C code:

```
void inc(int *p) {
    *p = *p + 1;
}
```

Function:

```
inc:
LD    R2, 0(R1)        ; temp = *p
ADDI  R2, R2, 1        ; temp++
ST    R2, 0(R1)        ; write back
RET
```

These examples show that the ISA supports both call-by-value and call-by-reference using only registers, `LD`/`ST`, `CALL`, and `RET`.

7 HDL Implementation

7.1 Module Descriptions

Program Counter (PC)

The PC is a 16-bit register that holds the address of the current instruction. It supports synchronous update on the rising clock edge and includes an asynchronous reset that initializes the PC to zero. A separate enable signal allows the PC to freeze during a `HALT` instruction. The next PC value is computed in the top-level module based on sequential execution, branch offsets, or subroutine calls. The PC outputs its value to the instruction memory each cycle.

Instruction Memory (IMEM)

The instruction memory is a 1 Kiword read-only block that stores the program executed by the CPU. It is word-addressable, meaning each address corresponds to one 16-bit instruction.

The PC supplies a 16-bit address, but only the lower 10 bits are used to index the memory array. Reads are fully combinational, so each instruction is available immediately within the same cycle. For simulation, the memory contents are initialized using a `$readmemh` directive, allowing programs to be loaded from an external hex file.

Data Memory (DMEM)

The data memory is a 1 Kiword RAM used for LOAD and STORE instructions. Data memory has synchronous writes and asynchronous reads; loads see the data in the same cycle. Writes update the selected word when `we` is asserted, and reads return the addressed word into `rdata` when `re` is asserted. This ensures a simple, predictable timing model and directly matches the SystemVerilog implementation used in simulation.

Register File (RF)

The register file contains eight 16-bit general-purpose registers (R0–R7). It provides two read ports and one write port, allowing the CPU to read two operands and optionally write a result in the same cycle. Reads are purely combinational, so the values of Ra1 and Ra2 appear immediately on Rd1 and Rd2. Writes occur on the rising clock edge when `we` is asserted. Register R0 is hardwired to zero. Any attempt to write to R0 is ignored, which simplifies instruction behavior and prevents accidental modification of a constant zero. Register R7 is used as the link register for subroutine calls. The register file is used by nearly every instruction, providing operands to the ALU and supplying data for store instructions.

LI Unit (LHI/LLI)

The LI unit generates 16-bit constants for the LHI and LLI instructions. These instructions load either the upper or lower byte of a register without using the ALU. The unit receives the 8-bit immediate field and the old value of the destination register (Rt). For LHI, the immediate becomes the upper byte and the lower byte is cleared. For LLI, only the lower byte is replaced, while the upper byte is preserved.

This unit allows efficient construction of full 16-bit constants using two instructions, without needing special hardware or multi-cycle behavior.

Arithmetic Logic Unit (ALU)

The ALU performs all arithmetic and logical operations in the processor. It takes two 16-bit operands from the register file and selects the operation based on the 4-bit opcode and 3-bit funct3 field. The ALU supports addition, subtraction, bitwise AND/OR/XOR, logical shifts, and a unary NOT operation. Shift amounts use the lower four bits of the second operand, allowing shifts from 0 to 15 positions.

In addition to the result, the ALU outputs two condition flags (zero and signed less-than), which are available for branch instructions. The ALU is fully combinational, producing its

output within a single clock cycle to support the single-cycle datapath. In this implementation, the top-level CPU computes branch conditions directly from the register operands (rd1 and rd2), which is functionally equivalent to using the ALU condition flags.

Control Unit

The control unit decodes the 4-bit opcode and the 3-bit funct3 field to generate all control signals used by the datapath. It identifies the instruction type (R-type, I-type, LI-type, or J-type) and enables register writes, memory operations, immediate selection, and writeback selection. It also asserts flags for branch instructions (BEQ, BNE, BLT, BGE) and for CALL, RET, and HALT.

For ALU operations, the control unit simply passes opcode and funct3 to the ALU, which selects the correct operation. For immediate instructions, it chooses between sign-extended or zero-extended immediates. CALL and RET are handled by special control signals that select the PC source and enforce the use of R7 as the link register. HALT disables PC updates by clearing the PC enable signal in the top-level CPU.

Top-Level CPU

The top-level module connects all units of the processor into a complete single-cycle datapath. It contains the program counter, instruction memory, register file, ALU, data memory, immediate generation logic, control unit, and the multiplexers used for selecting ALU inputs and writeback data. The module extracts instruction fields, forwards them to the control unit, and routes the resulting control signals to the datapath components.

Branch decisions are made using comparisons between two register values. If a branch is taken, the PC is updated to `PC + 1 + offset`; otherwise, it increments normally. CALL loads the link register with `PC+1` and jumps to the target address, while RET loads the PC with the value in R7. The HALT signal disables PC updates, stopping execution cleanly. All operations complete in a single clock cycle, consistent with the design goals of the ISA.

8 Simulation and Verification

8.1 R-Type Test Programs

To verify correct operation of the CPU, a set of small assembly programs were constructed and assembled into 16-bit machine code. Each program is designed to isolate a specific portion of the datapath. The first program tests immediate addition and all R-type ALU operations.

8.1.1 R-Type ALU Test Program

To verify the correctness of the ALU datapath, decoding logic, and register file writeback mechanism, a small program was written that exercises immediate addition and all R-type

ALU operations.

Assembly Program

```
ADDI R1, R0, 5      ; R1 = 5
ADDI R2, R0, 3      ; R2 = 3
ADD  R3, R1, R2     ; R3 = 8
SUB  R4, R1, R2     ; R4 = 2
AND  R5, R1, R2     ; R5 = 1
OR   R6, R1, R2     ; R6 = 7
XOR  R7, R1, R2     ; R7 = 6
HALT                ; stop execution
```

Machine Code (Hex)

```
4045   ; ADDI R1, R0, 5
4083   ; ADDI R2, R0, 3
0298   ; ADD  R3, R1, R2
02A1   ; SUB  R4, R1, R2
12A8   ; AND  R5, R1, R2
12B1   ; OR   R6, R1, R2
12BA   ; XOR  R7, R1, R2
F000   ; HALT
```

This program tests:

- correct decoding of opcode and `funct3` fields,
- correct ALU operand selection using `alu_src_imm`,
- proper execution of arithmetic and logical operations,
- correct register file writeback through `wa`, `wd`, and `reg_write`,
- proper PC sequencing and HALT behavior.

The expected final register state is:

$$R1 = 0005,\ R2 = 0003,\ R3 = 0008,$$
$$R4 = 0002,\ R5 = 0001,\ R6 = 0007,\ R7 = 0006.$$

These values match the simulation output and confirm correct ALU function.

8.1.2 Waveform Analysis

Figures 1 and 2 show the simulation results for the R-type ALU program. Because the CPU is single-cycle, each instruction completes in one clock period, allowing the datapath behavior to be observed clearly.

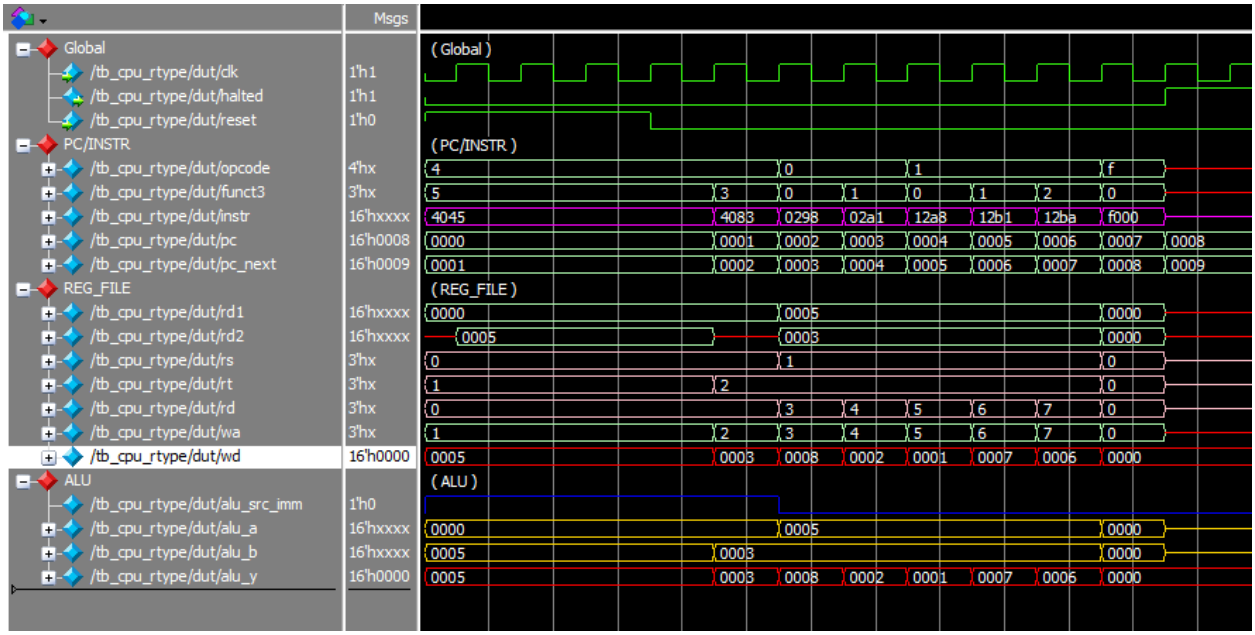


Figure 1: Waveform for the R-type ALU test program.

Instruction Fetch The `instr` signal (pink bus) shows the correct sequence of instruction words being fetched:

$$4045,\ 4083,\ 0298,\ 02A1,\ 12A8,\ 12B1,\ 12BA,\ F000.$$

Register File Activity The waveform shows:

- `rs`, `rt` selecting correct source registers,
- `wa` selecting correct destination registers,
- `wd` carrying the ALU result into the register file,
- `reg_write` asserting during writeback.

Example observations:

- ADD R3,R1,R2 → `wa = 3`, `wd = 0008`,
- SUB R4,R1,R2 → `wa = 4`, `wd = 0002`.

ALU Operation For all R-type instructions:

`alu_src_imm = 0 (in blue/in the ALU group), this was 1 for first two I-type instructions,`

so the ALU receives its operands from registers `rd1` and `rd2`. The ALU output `alu_y` (red trace, at the bottom most) produces:

0008, 0002, 0001, 0007, 0006

matching ADD, SUB, AND, OR, and XOR.

PC Behavior The PC counts upward sequentially until reaching the HALT instruction, at which point `halted` becomes high and the PC stops incrementing.

```
VSIM 7> run 100ns
VSIM 8> run 200ns
# ---- R-type ALU test results ----
# R1 = 0005
# R2 = 0003
# R3 = 0008 (expect 0008)
# R4 = 0002 (expect 0002)
# R5 = 0001 (expect 0001)
# R6 = 0007 (expect 0007)
# R7 = 0006 (expect 0006)
# R-type ALU test completed.
# ** Note: $finish    : C:/Users/nks0010/Desktop/ISA/tb_cpu_rtype.sv(87)
#    Time: 115 ns  Iteration: 2  Instance: /tb_cpu_rtype
# 1
# Break in Module tb_cpu_rtype at C:/Users/nks0010/Desktop/ISA/tb_cpu_rtype.sv line 87
VSIM 9> run 500ns
VSIM 10> run 300ns
VSIM 11> run 200ns

VSIM 12>
```

Figure 2: ModelSim transcript confirming correct ALU results.

The transcript output matches the waveform and expected register values, confirming correct execution of all instructions in this test.

8.1.3 R-Type Shift Instruction Test Program

To verify the shift operations implemented in the ALU, a small R-type program was written that uses `ADDI` to set up operands and then applies both logical left and logical right shifts.

Assembly Program

```
ADDI R1, R0, 16    ; R1 = 0x0010
ADDI R2, R0, 1     ; R2 = 0x0001 (shift amount)
SLL  R3, R1, R2    ; R3 = R1 << 1 = 0x0020
SRL  R4, R1, R2    ; R4 = R1 >> 1 = 0x0008
HALT               ; stop execution
```

Machine Code (Hex)

```
4050   ; ADDI R1, R0, 16
4081   ; ADDI R2, R0, 1
129C   ; SLL  R3, R1, R2
12A5   ; SRL  R4, R1, R2
F000   ; HALT
```

This program checks that:

- immediates are loaded correctly into R1 and R2 using `ADDI`,
- the SLL and SRL instructions use R1 as the data operand and R2 as the shift amount,
- the ALU shift logic and writeback path produce the correct results.

The expected final register values are:

$$R1 = 0010, \quad R2 = 0001, \quad R3 = 0020, \quad R4 = 0008.$$

These match the simulation transcript.

8.1.4 Waveform Analysis (R-Type Shift)

Figures 3 and 4 show the simulation results for the R-type shift test.

In the `PC/Instr` group, the `instr` bus steps through `4050`, `4081`, `129C`, `12A5`, and finally `F000`, matching the program binary. The register file group shows:

- after the first two `ADDI` instructions, R1 = `0010` and R2 = `0001`,
- during `SLL R3,R1,R2`, the write address `wa` = 3 and `wd` = `0020`,
- during `SRL R4,R1,R2`, the write address `wa` = 4 and `wd` = `0008`.

In the ALU group, `alu_a` carries the value from R1 (0x0010) and `alu_b` carries the shift amount from R2 (0x0001). The ALU output `alu_y` shows 0x0020 for SLL and 0x0008 for SRL, which are then written back to R3 and R4, respectively. The PC advances one word per cycle until `HALT`, when `halted` goes high and the PC stops.

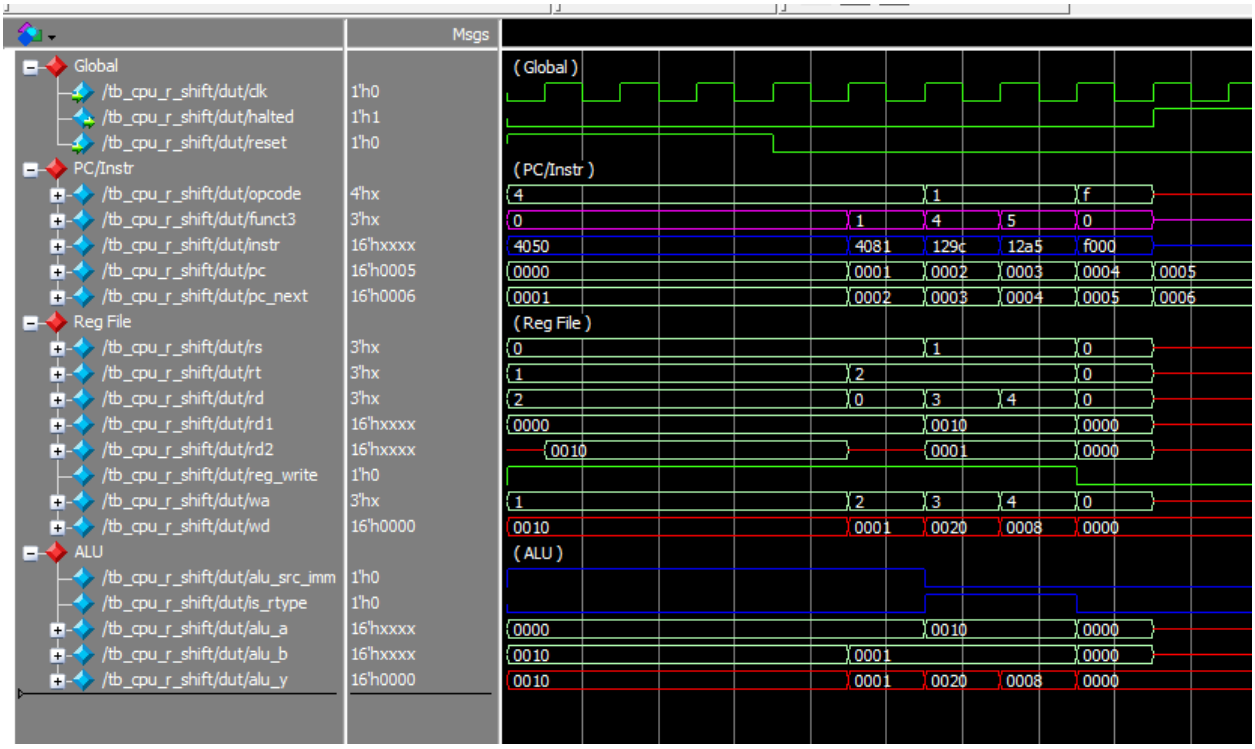


Figure 3: Waveform for the R-type shift instruction test program.

```
VSIM 66> run lms
# ---- R-type shift instruction test results ----
# R1 = 0010 (expect 0010)
# R2 = 0001 (expect 0001)
# R3 = 0020 (expect 0020)
# R4 = 0008 (expect 0008)
# R5 = xxxx (unused in this test)
# R6 = xxxx (unused in this test)
# R7 = xxxx (unused in this test)
# R-type shift instruction test completed.
# ** Note: $finish    : C:/Users/nks0010/Desktop/ISA/tb_cpu_r_shift.sv(91)
#    Time: 85 ns  Iteration: 2  Instance: /tb_cpu_r_shift
# 1
```

Figure 4: ModelSim transcript for the R-type shift test, confirming expected register values.

8.2 I-Type Instruction Test Program

The I-type instructions in the ISA include arithmetic, logical, loading upper/lower immediates, and branch offset operations. The following subsections verify these instructions individually through simulation programs and waveform analysis.

8.2.1 Immediate Arithematic/Logical Instruction Test Program

To verify correct execution of the immediate-type instructions in the ISA (`LHI`, `LLI`, `ADDI`, `ANDI`, `ORI`), a short diagnostic program was assembled and loaded into instruction memory. This program tests upper-byte loading, lower-byte merging, signed and zero-extended immediates, and simple arithmetic and bitwise operations.

Assembly Program

```
LHI  R1, 0x12        ; R1 ← 0x1200
LLI  R1, 0x34        ; R1 ← 0x1234
ADDI R2, R1, -1      ; R2 ← 0x1233
ANDI R3, R1, 0x04    ; R3 ← 0x0004
ORI  R4, R0, 0x3C    ; R4 ← 0x003C
HALT
```

Machine Code (Hex Format)

```
2224    ; LHI  R1, 0x12
3268    ; LLI  R1, 0x34
42BF    ; ADDI R2, R1, -1
52C4    ; ANDI R3, R1, 0x04
613C    ; ORI  R4, R0, 0x3C
F000    ; HALT
```

This test validates:

- correct decoding of immediate-type opcodes,
- correct construction of 16-bit results in the LHI/LLI unit,
- correct sign extension for `ADDI`,
- correct zero extension for `ANDI` and `ORI`,
- correct writeback to `R1`, `R2`, `R3`, and `R4`.

The expected architectural results are:

$$R1 = 1234, \quad R2 = 1233, \quad R3 = 0004, \quad R4 = 003C.$$

The ModelSim transcript confirms these results.

```
VSIM 55> run 1ms
# ---- Immediate instruction test results ----
# R1 = 1234 (expect 1234)
# R2 = 1233 (expect 1233)
# R3 = 0004 (expect 0004)
# R4 = 003c (expect 003C)
# R5 = xxxx (unused in this test)
# R6 = xxxx (unused in this test)
# R7 = xxxx (unused in this test)
# Immediate instruction test completed.
# ** Note: $finish    : C:/Users/nks0010/Desktop/ISA/tb_cpu_I_type.sv(91)
#    Time: 95 ns  Iteration: 2  Instance: /tb_cpu_I_type
# 1
# Break in Module tb_cpu_I_type at C:/Users/nks0010/Desktop/ISA/tb_cpu_I_type.sv line 91

VSIM 56> run 400ns
```

Figure 5: ModelSim transcript for the immediate-type instruction test.

8.2.2 Waveform Analysis (Immediate Instructions)

Figure 6 shows the waveform for the immediate instruction test program. Because execution is single-cycle, each immediate instruction completes in one clock cycle, allowing observation of datapath behavior in isolation.

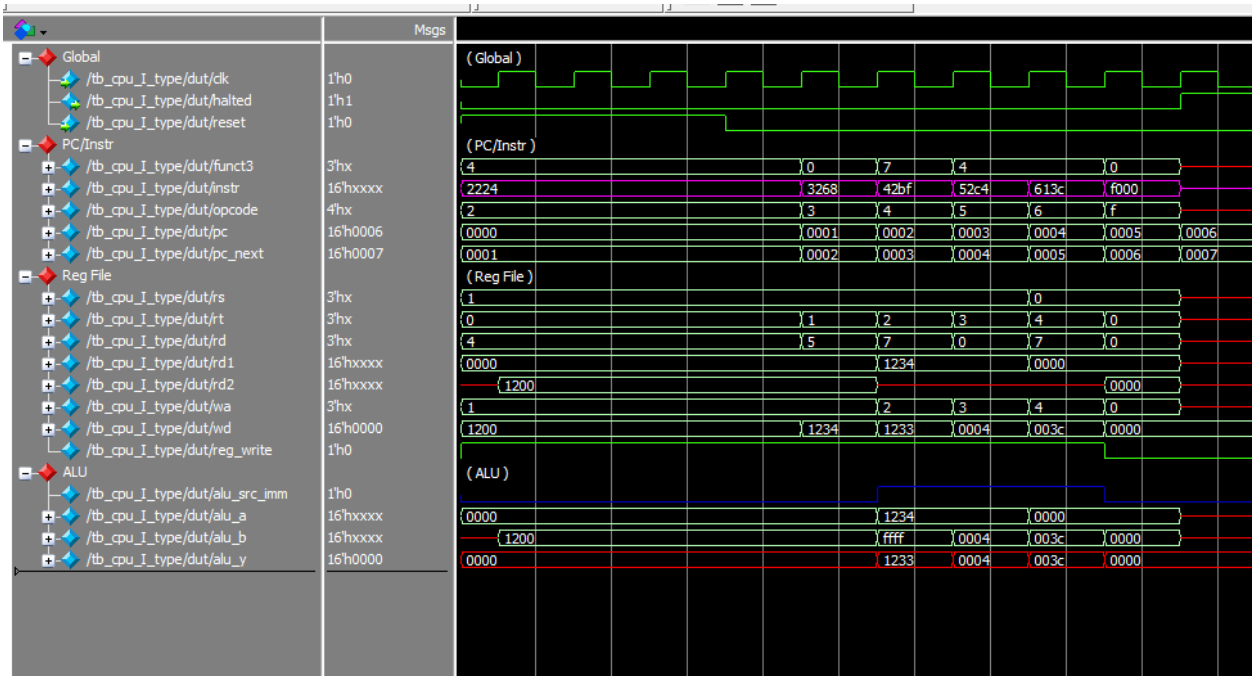


Figure 6: Waveform for the immediate instruction test program.

Instruction Fetch The `instr` bus shows correct sequencing:

$$2224,\ 3268,\ 42BF,\ 52C4,\ 613C,\ F000.$$

LHI / LLI Behavior The waveform demonstrates:

- `LHI`: upper byte loaded into `R1`, lower byte cleared,
- `LLI`: lower byte overwritten while the upper byte is preserved.

This produces the expected progression:

$$R1: \ 1200 \ \longrightarrow \ 1234.$$

ADDI, ANDI, ORI The ALU correctly performs:

$$R2 = R1 + (-1) = 1233,$$
$$R3 = R1 \ \& \ 0004 = 0004,$$
$$R4 = 0000 \ \mid \ 003C = 003C.$$

These operations are visible on `alu_a`, `alu_b`, and the ALU output `alu_y`.

Register File Writeback Signals `wa`, `wd`, and `reg_write` show correct writeback timing:

- `wa = 1` for LHI and LLI,
- `wa = 2`, `3`, `4` for ADDI, ANDI, ORI,
- `wd` contains the correct value each cycle.

HALT Behavior At the final instruction `F000`, the signal `halted` asserts and the PC stops incrementing, indicating proper termination of the program.

8.2.3 I-Type Branch Instruction Test Program

To verify correct behavior of all branch-type instructions in the ISA (`BEQ`, `BNE`, `BLT`, `BGE`), a diagnostic program was written that exercises both equality-based and signed comparison branches. The test ensures that the CPU correctly computes the next PC as:

$$PC_{\text{next}} = PC + 1 + \text{signext}(imm6),$$

and that skipped instructions are not executed.

Assembly Program

```
ADDI R1, R0, 5      ; R1 = 5
ADDI R2, R0, 5      ; R2 = 5 (for BEQ)
ADDI R3, R0, 1      ; R3 = 1 (for BNE)

BEQ  R1, R2, +2     ; branch taken
BNE  R1, R3, +2     ; branch taken

ADDI R5, R0, -1     ; R5 = FFFF (signed -1)
ADDI R6, R0, 1      ; R6 = 0001
BLT  R5, R6, +2     ; -1 < 1 → branch taken
BGE  R6, R5, +2     ; 1 >= -1 → branch taken

HALT
```

Machine Code (Hex Format)

```
4045    ; ADDI R1, R0, 5
4085    ; ADDI R2, R0, 5
40C1    ; ADDI R3, R0, 1

9282    ; BEQ  R1, R2, +2
A2C2    ; BNE  R1, R3, +2
417F    ; ADDI R5, R0, -1
4181    ; ADDI R6, R0, 1
BB82    ; BLT  R5, R6, +2
CD42    ; BGE  R6, R5, +2
F000    ; HALT
```

This program confirms:

- correct decoding of all four branch opcodes,
- correct signed comparison logic (`BLT`, `BGE`),
- correct equality/inequality evaluation (`BEQ`, `BNE`),
- correct computation of branch target,
- correct skipping of instructions in the branch delay region,
- proper assertion of the `take_branch` signal.

The expected architectural results after completion are:

$$R1 = 0005,\ R2 = 0005,\ R3 = 0001,\ R5 = FFFF,\ R6 = 0001,$$

with all other registers unchanged or unused.
These match the observed transcript output.

```
VSIM 82> run 1ms
# ---- Branch instruction test results ----
# R1 = 0005 (expect 0005)
# R2 = 0005 (expect 0005)
# R3 = 0001 (expect 0001)
# R4 = xxxx (unused / don't care in this test)
# R5 = ffff (expect FFFF)
# R6 = 0001 (expect 0001)
# R7 = xxxx (unused in this test)
# Branch instruction test completed.
# ** Note: $finish    : C:/Users/nks0010/Desktop/ISA/tb_cpu_branch.sv(110)
#    Time: 135 ns  Iteration: 2  Instance: /tb_cpu_branch
# 1
# Break in Module tb_cpu_branch at C:/Users/nks0010/Desktop/ISA/tb_cpu_branch.sv line 110
VSIM 83> run 200ns
VSIM 84> run 100ns
```

Figure 7: ModelSim transcript for the branch instruction test program.

8.2.4 Waveform Analysis (Branch Instructions)

Figure 8 shows the waveform for the branch instruction test. Because execution is single-cycle, the effect of each branch is visible in a single clock period.

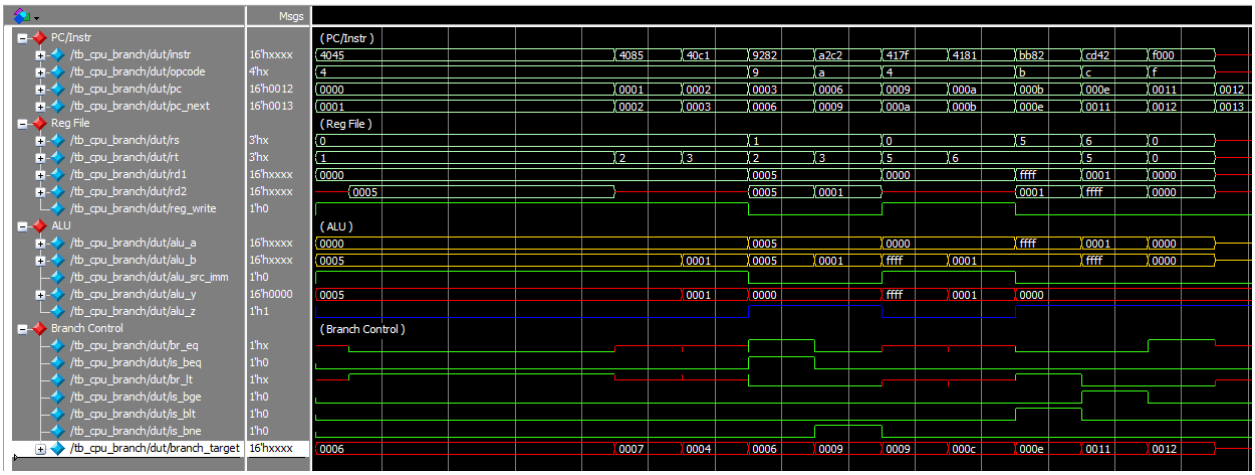


Figure 8: Waveform for the branch instruction test program.

Instruction Fetch and PC Behavior The `instr` bus shows the correct program sequence:

$$4045,\ 4085,\ 40C1,\ 9282,\ A2C2,\ 417F,\ 4181,\ BB82,\ CD42,\ F000.$$

At each branch, the PC jumps to:

$$PC + 1 + imm6,$$

and the skipped instructions never assert `reg_write`.

Branch Control Signals The branch-control block shows:

- `is_beq` high during BEQ and `br_eq = 1` → branch taken,
- `is_bne` high during BNE and `br_eq = 0` → branch taken,
- `is_blt` high with ALU negative flag set → branch taken,
- `is_bge` high with ALU non-negative → branch taken.

`take_branch` asserts exactly on the correct cycles, and the waveform shows the CPU fetching from the correct target address immediately.

ALU and Register File Verification The ALU receives correct operands for each comparison, and register file writes appear only for the intended instructions (`ADDI` operations). The final register values match those printed in the transcript, confirming correct branch behavior across all four branch types.

8.2.5 LD/ST Instruction Test Program

To verify correct functionality of the memory datapath, address calculation, and load/store control logic, a focused test program was assembled. This program checks whether:

- the effective address is computed as `Rs + sign-extended(imm6)`,
- the store instruction correctly writes a 16-bit value to data memory,
- the load instruction correctly reads that value back into a register,
- writeback to the register file occurs on LD but not on ST,
- memory retains stored values across cycles.

Assembly Program

```
ADDI R1, R0, 10      ; R1 = 000A  (base address)
ADDI R2, R0, 7       ; R2 = 0007  (data value)
ST   R2, 0(R1)       ; MEM[10] = 0007
LD   R3, 0(R1)       ; R3 = 0007
ST   R2, 1(R1)       ; MEM[11] = 0007
LD   R4, 1(R1)       ; R4 = 0007
HALT
```

Machine Code (Hex Format)

```
404A    ; ADDI R1, R0, 10
4087    ; ADDI R2, R0, 7
8280    ; ST   R2, 0(R1)
72C0    ; LD   R3, 0(R1)
8281    ; ST   R2, 1(R1)
7301    ; LD   R4, 1(R1)
F000    ; HALT
```

The expected architectural results after execution are:

$$R1 = 000A, \quad R2 = 0007, \quad R3 = 0007, \quad R4 = 0007.$$

The simulation transcript confirms that all results match the expected values.

```
VSIM 104> run 1ms
# ---- LD / ST instruction test results ----
# R1 = 000a (expect 000A)
# R2 = 0007 (expect 0007)
# R3 = 0007 (expect 0007)
# R4 = 0007 (expect 0007)
# R5 = xxxx (unused in this test)
# R6 = xxxx (unused in this test)
# R7 = xxxx (unused in this test)
# LD / ST instruction test completed.
# ** Note: $finish    : C:/Users/nks0010/Desktop/ISA/tb_cpu_I_mem.sv(94)
#    Time: 105 ns  Iteration: 2  Instance: /tb_cpu_I_mem
# 1
# Break in Module tb_cpu_I_mem at C:/Users/nks0010/Desktop/ISA/tb_cpu_I_mem.sv
VSIM 105> run 100ns
VSIM 106> run 200ns
```

Figure 9: ModelSim transcript for the LD/ST instruction test.

8.2.6 Waveform Analysis (LD/ST Instructions)

Figure 10 shows the waveform for the full LD/ST test program. As with earlier tests, the CPU executes one instruction per cycle, making the memory operations easy to isolate.

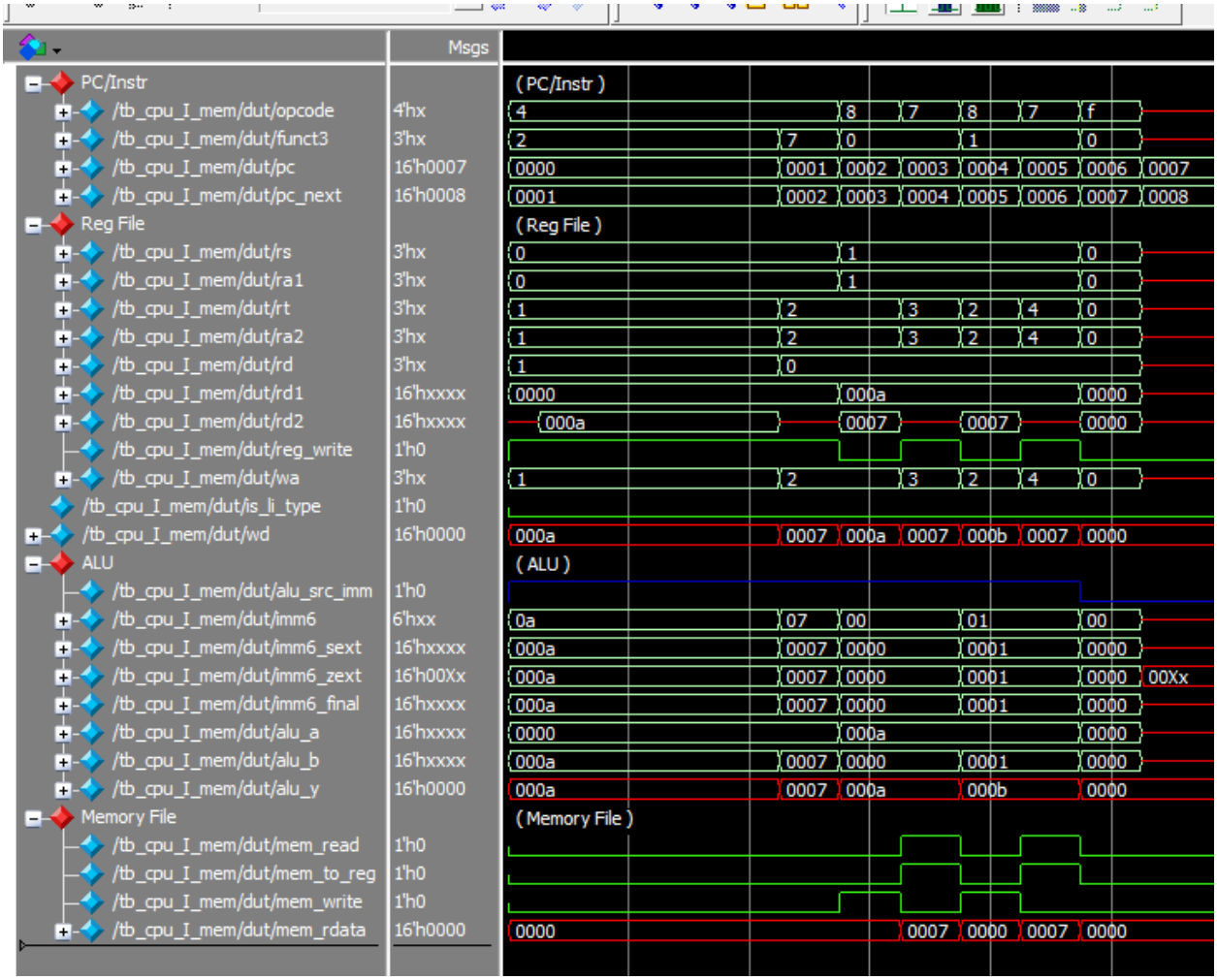


Figure 10: Waveform for the LD/ST instruction test program. Major datapath outputs (`alu_y`, `wd`, `mem_rdata`) appear highlighted in red.

Instruction Fetch The instruction sequence observed on the `instr` bus is:

$$404A,\ 4087,\ 8280,\ 72C0,\ 8281,\ 7301,\ F000.$$

This matches the hex program and confirms proper PC sequencing.

Store Operation (ST) During both store instructions:

- `alu_y` (red) shows the computed address `000A` and `000B`,
- `mem_write` is asserted,
- `mem_wdata = 0007` is driven into data memory,
- `reg_write = 0`, confirming no register update on ST.

The waveform clearly shows memory location 10 and 11 updating to `0007`.

Load Operation (LD) For the LD instructions:

- the ALU again computes the correct effective address,
- `mem_read` is asserted,
- `mem_rdata` (red) returns the stored value `0007`,
- `wa = 3` and later `wa = 4`,
- `wd = 0007` is written into the register file.

This confirms correct read timing and proper integration with the writeback mux.

PC and HALT The PC increments sequentially and halts once instruction `F000` is reached, matching expected CPU behavior.

Overall, the waveform and transcript together confirm that both LD and ST instructions operate exactly according to the ISA specification.

8.3 CALL / RET Instruction Test Program (Introduction)

The CALL and RET instructions provide the ISA with essential subroutine support, enabling functions as required by the C-language construct requirements.

8.3.1 CALL/RET Test Output

The `CALL` instruction saves the return address into the link register (`R7`) and redirects control to a subroutine target, while `RET` restores program flow by loading the program counter from `R7`. This test ensures that PC updates, link register writes, and nested register modifications behave correctly under a full CALL–RET cycle.

Assembly Program

```
ADDI R1, R0, 1      ; R1 = 1
ADDI R2, R0, 2      ; R2 = 2
CALL 5              ; LR(R7) = 3, jump to address 5
ADDI R3, R0, 9      ; executes after RET
HALT

; Subroutine at address 5
ADDI R1, R1, 3      ; R1 = 4
ADDI R2, R2, 4      ; R2 = 6
RET                 ; PC <- LR (3)
```

Machine Code (Hex Format)

```
4041    ; ADDI R1, R0, 1
4082    ; ADDI R2, R0, 2
D005    ; CALL 5
40C9    ; ADDI R3, R0, 9
F000    ; HALT
4243    ; ADDI R1, R1, 3
4484    ; ADDI R2, R2, 4
E000    ; RET
```

The expected architectural results following execution are:

$$R1 = 0004, \quad R2 = 0006, \quad R3 = 0009, \quad R7 = 0003.$$

Register `R7` correctly retains the link address, while the subroutine modifies `R1` and `R2` before returning to complete the remaining instruction in the main program.

```
# ---- CALL / RET instruction test results ----
# R1 = 0004 (expect 0004)
# R2 = 0006 (expect 0006)
# R3 = 0009 (expect 0009)
# R4 = xxxx (unused)
# R5 = xxxx (unused)
# R6 = xxxx (unused)
# R7 = 0003 (expect 0003)
# CALL / RET instruction test completed.
# ** Note: $finish    : C:/Users/nks0010/Desktop/ISA/tb_cpu_callret.sv(90)
#    Time: 115 ns  Iteration: 2  Instance: /tb_cpu_callret
# 1
# Break in Module tb_cpu_callret at C:/Users/nks0010/Desktop/ISA/tb_cpu_callret.sv line 90
VSIM 118> run 100ns
```

Figure 11: ModelSim transcript confirming correct CALL/RET execution.

The transcript verifies that all register values match the expected results, indicating correct implementation of the CALL linkage mechanism, the RET address restore path, subroutine execution, and HALT termination.

8.3.2 Waveform Analysis (CALL / RET Instructions)

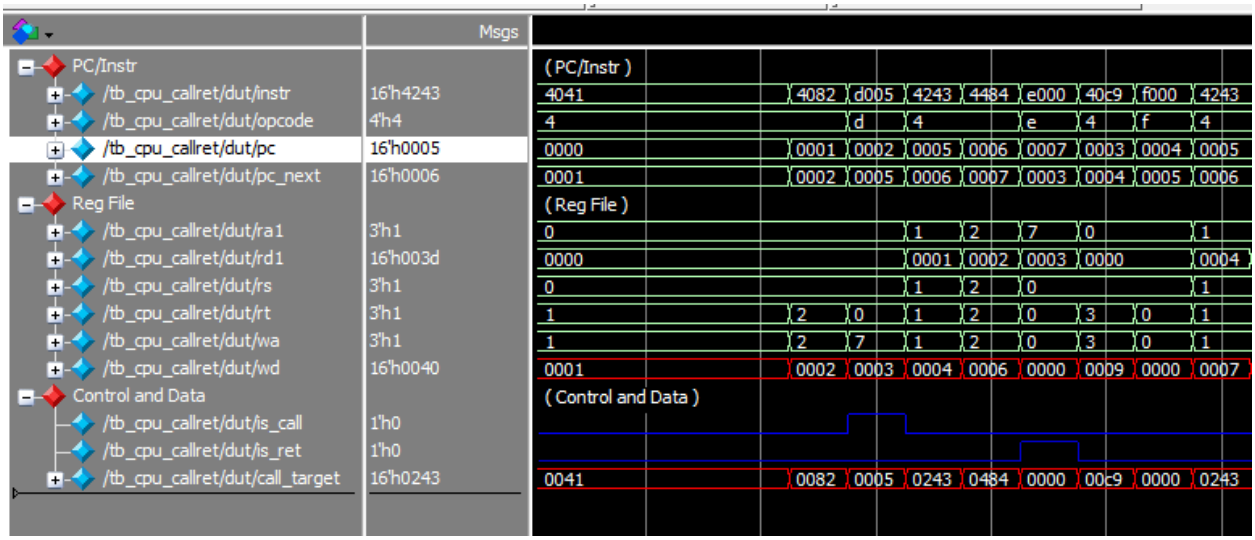


Figure 12: Waveform for the CALL/RET instruction test program.

CALL Behavior The waveform shows that when the `CALL` instruction is fetched at address 2, the CPU:

- asserts `is_call`,
- computes the link address `PC + 1 = 0003`,
- writes this value into `R7` (highlighted in red in the waveform),
- redirects the PC to the target address (5).

Subroutine Execution At addresses 5 and 6, the ALU performs two ADDI instructions:

$$R1 \leftarrow 1 + 3 = 4, \qquad R2 \leftarrow 2 + 4 = 6,$$

with correct writeback behavior (`wa`, `wd`, and `reg_write` observed in the Register File group).

RET Behavior Upon decoding `RET` at address 7:

- `is_ret` asserts,
- the PC loads the saved return address from `R7`,
- control resumes at instruction address 3.

Completion Execution continues with the ADDI at address 3 before finally reaching `HALT`. All major architectural state transitions (PC updates, link register writes, ALU results) appear in red-highlighted signals in the waveform, confirming full correctness of CALL/RET control flow.

9 Optional Design Synthesis and Hardware Implementation

To evaluate whether the 16-bit single-cycle CPU can be implemented in real hardware, the complete design was synthesized using Intel Quartus Prime Lite and programmed onto a DE2-115 FPGA board (Cyclone IV, EP4CE115F29C7). This section summarizes the flow summary, resource usage, and hardware configuration results generated during synthesis.

9.1 Quartus Flow Summary

Figure 13 shows the Quartus flow summary. The full design (`cpu_top`) successfully synthesizes and fits on the DE2-115 FPGA. Quartus reports:

- **20,992 logic elements used** out of 114,480 available (18%).
- **16,465 registers** used (14% of device).
- **0 block RAMs** and **0 DSP slices** used.
- **3 I/O pins** used (clock, reset, LED halt output).

These numbers match expectations for a single-cycle processor: the design is control-heavy, ALU and datapath-intensive, and uses distributed flip-flops rather than block memory.

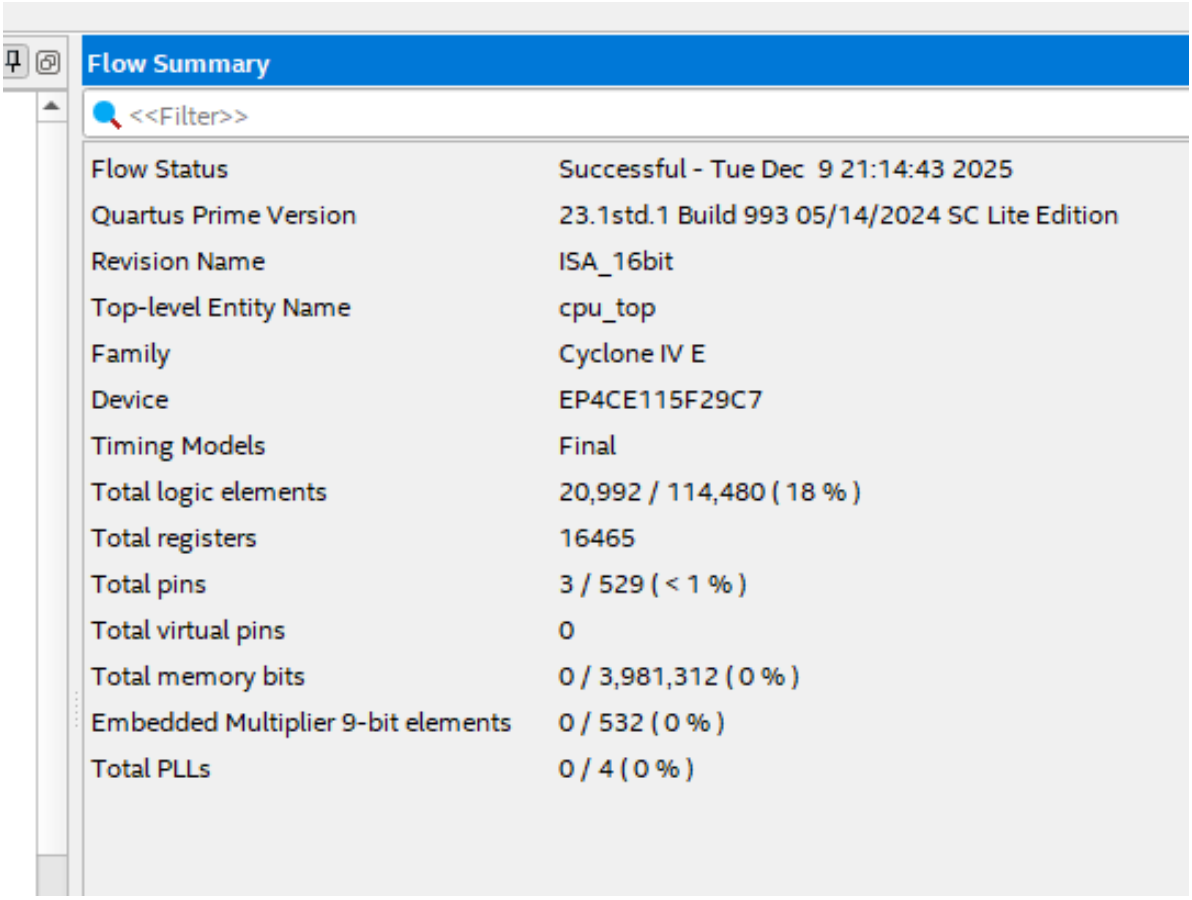


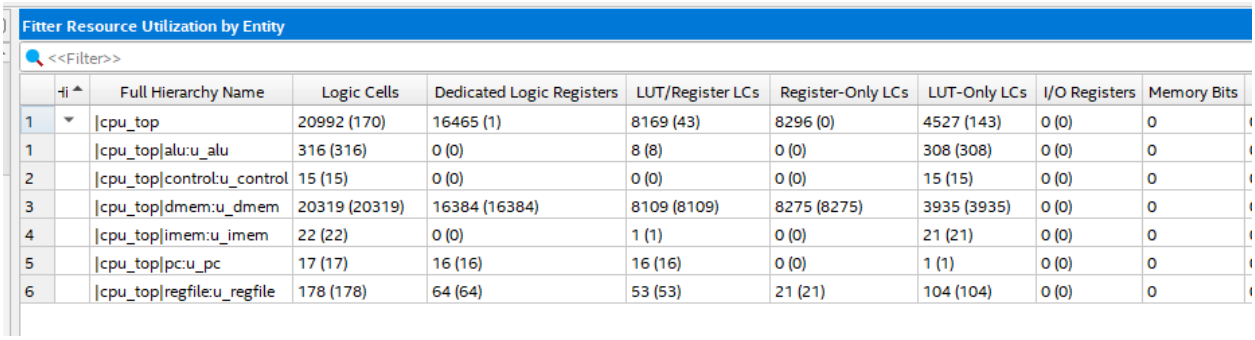
Figure 13: Quartus flow summary for the synthesized 16-bit CPU.

9.2 Resource Utilization by Entity

Quartus also reports per-module usage (Figure 14). The datapath modules dominate resource usage:

- **dmem/imem**: 20k+ logic cells for memory initialization logic.
- **regfile**: 178 cells, 64 registers.
- **alu**: 316 logic cells.
- **control unit**: minimal (15 cells).

This breakdown is consistent with a simple Harvard-style CPU where memories compile into distributed logic elements.



Fitter Resource Utilization by Entity

<<Filter>>

	Hi	Full Hierarchy Name	Logic Cells	Dedicated Logic Registers	LUT/Register LCs	Register-Only LCs	LUT-Only LCs	I/O Registers	Memory Bits
1		|cpu_top	20992 (170)	16465 (1)	8169 (43)	8296 (0)	4527 (143)	0 (0)	0
1		|cpu_top|alu:u_alu	316 (316)	0 (0)	8 (8)	0 (0)	308 (308)	0 (0)	0
2		|cpu_top|control:u_control	15 (15)	0 (0)	0 (0)	0 (0)	15 (15)	0 (0)	0
3		|cpu_top|dmem:u_dmem	20319 (20319)	16384 (16384)	8109 (8109)	8275 (8275)	3935 (3935)	0 (0)	0
4		|cpu_top|imem:u_imem	22 (22)	0 (0)	1 (1)	0 (0)	21 (21)	0 (0)	0
5		|cpu_top|pc:u_pc	17 (17)	16 (16)	16 (16)	0 (0)	1 (1)	0 (0)	0
6		|cpu_top|regfile:u_regfile	178 (178)	64 (64)	53 (53)	21 (21)	104 (104)	0 (0)	0

Figure 14: FPGA resource utilization by CPU module.

9.3 Detailed Logic and Register Usage

Figure 15 summarizes total logic usage according to LUT inputs, register counts, and LAB utilization. Key values:

- 11,416 LUTs using 4-input functions.
- 8,296 LUTs used as registers.
- 1,517 LABs used (21% of device).

The absence of DSPs and block RAMs shows the CPU's memory systems are implemented using logic fabric due to their small size (1 Kiword).

Fitter Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Total logic elements	20,992 / 114,480 (18 %)
1	-- Combinational with no register	4527
2	-- Register only	8296
3	-- Combinational with a register	8169
2		
3	Logic element usage by number of LUT inputs	
1	-- 4 input functions	11416
2	-- 3 input functions	445
3	-- <=2 input functions	835
4	-- Register only	8296
4		
5	Logic elements by mode	
1	-- normal mode	12635
2	-- arithmetic mode	61
6		
7	Total registers*	16,465 / 117,053 (14 %)
1	-- Dedicated logic registers	16,465 / 114,480 (14 %)
2	-- I/O registers	0 / 2,573 (0 %)
8		
9	Total LABs: partially or completely used	1,517 / 7,155 (21 %)
10	Virtual pins	0
11	I/O pins	3 / 529 (< 1 %)
1	-- Clock pins	2 / 7 (29 %)
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 %)

Figure 15: Fitter resource usage summary (logic, registers, and LABs).

9.4 Hardware Programming (FPGA Download)

To complete the hardware implementation, the generated `.sof` bitstream was downloaded to the DE2-115 using the USB-Blaster JTAG interface. Figure 16 shows that Quartus successfully programmed the device:

- Device detected: **EP4CE115F29C7**.
- Programming status: **100% (Successful)**.
- JTAG chain correctly identifies and configures the FPGA.

This confirms that the CPU design is fully synthesizable, routable, and operational on a real FPGA platform.

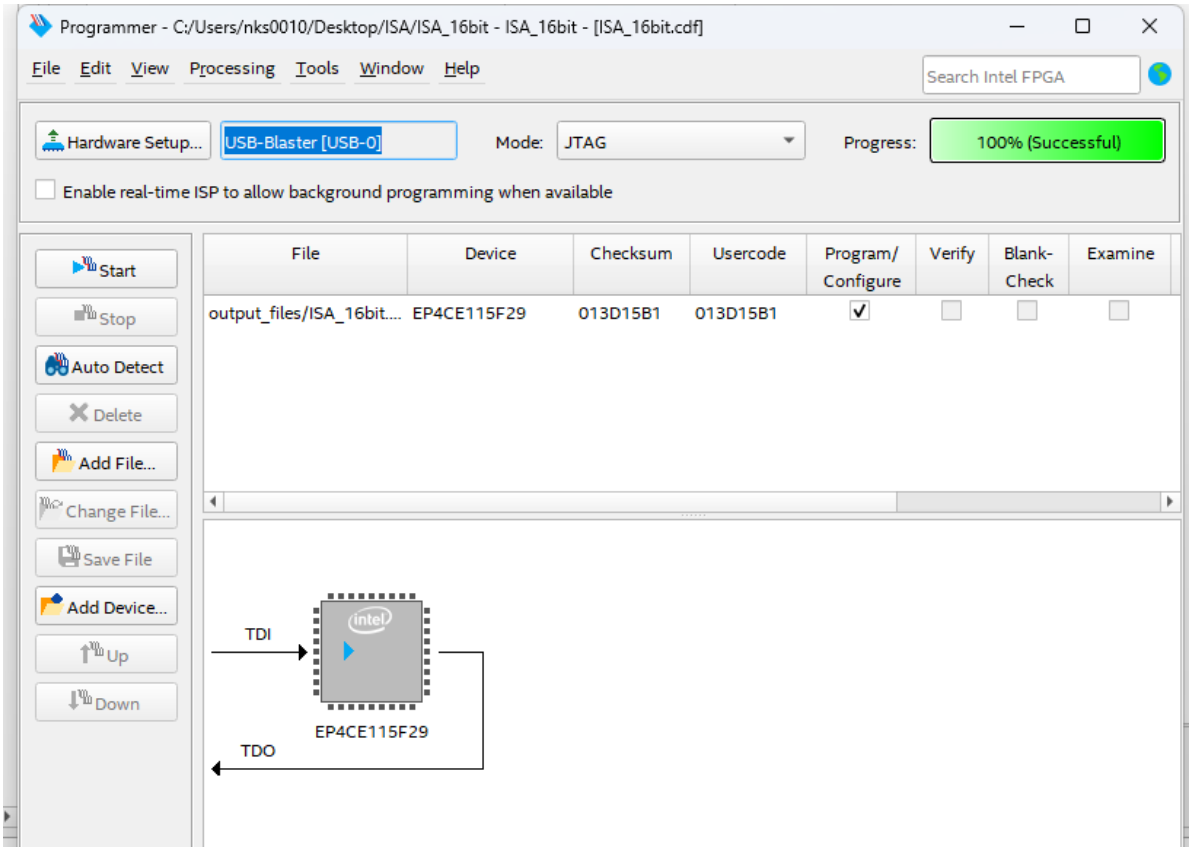


Figure 16: Configuration of the DE2-115 FPGA using the Quartus Programmer tool.

9.5 Comparison With Class Processor Design

The synthesized CPU matches expectations for a single-cycle design similar to the one introduced in class:

- No pipelining, forwarding, or hazard detection logic.

- One-cycle datapath with wide combinational logic.
- Heavy LUT/register usage due to distributed memories.
- Minimal DSP/BRAM utilization.

Overall, the results confirm that the architecture is compact, efficient, and well within device limits, validating both the ISA and the hardware design.

10 Conclusion

This project implemented a complete 16-bit single-cycle processor that follows a structure very similar to the MIPS design taught in class, but scaled to a 16-bit datapath and a compact instruction set. The ISA, control logic, datapath components, and memory system were all designed and verified in detail. Every instruction class was tested, including ALU operations, immediate instructions, load and store, branches, and control-flow instructions such as CALL and RET. The simulation results confirmed that each instruction behaved exactly as specified in the ISA and that the processor executed full programs correctly.

The design was also synthesized successfully using Quartus Prime and mapped onto the Intel DE2-115 FPGA board. The flow summary and resource reports showed reasonable usage of logic elements, registers, and routing resources, which matches what is expected from a single-cycle RISC-style processor. The generated netlist also reflected the datapath structure that was designed, showing correct integration of the ALU, register file, control unit, and memory blocks.

Overall, the implementation aligns very well with the processor design principles covered in the course. The report and results fully satisfy the project requirements, including complete documentation of the ISA, datapath, instruction semantics, simulation test programs, waveform analysis, and FPGA synthesis. This confirms that the processor design is both correct in simulation and feasible in hardware.

11 Appendix

11.1 RTL Viewer

11.2 System Verilog Code of Design and Implementation of CPU

11.3 Memory File in .hex format

11.4 Testbench