EE3043 Computer Architecture

Semester 241

Design of a Single Cycle RISC-V Processor

T.A: CAO XUAN HẢI

Student: Nguyễn Thanh Tùng – 2213874

Lê Tấn Dũng – 2210575

Lưu Lê Anh Khôi – 2211683

# Introduction

This project focuses on designing a single-cycle RV32I RISC-V processor. The design includes essential components like the Arithmetic Logic Unit (ALU) and Load-Store Unit (LSU), with integrated memory-mapped I/O. Following RV32I specifications, the processor will be verified through testbenches to ensure accurate functionality and performance.

# Design Strategy

## Overview

Figure 1: Single cycle overview

## ALU

The ALU (arithmatic logic unit) is responsible for executing various opera tions such as addition, subtraction, bitwise XOR, bitwise AND, bitwise OR, set less than, set less than unsigned, shift left logical, shift right logical, and shift right arithmetic.

In this implementation, the ALU is typically 32 bits data width. It takes input value from processor’s registers (from register file, immediate generation or program counter). The output value is selected base on alu operation, then store back to a register in register file. The ALU executes instructions of different types, including R-type and I-type instructions:

R-type: the input value always take from register file (rs1 and rs2) then store back the output value into rd.

I-type: the input value is take from register file, immediate generation or program counter and also store back the output value into rd.

To implement lui operation, the ALU need to add a new alu operation OPB that ALU output data equal to operand b data.

The ALU design requirement is “do not use subtraction (−), comparison (<, >), orshifting(≪, ≫, and ≫).”.

To perform subtraction (-) operation, calculate a - b by adding a to the two’s complement of b (a + ~b + 1).

To compare two numbers:

If operands a and b are unsigned, examine the overflow of the result from a - b: if overflow[a - b] = 1, then a < b; otherwise, a >= b.

If operands a and b are signed, check the MSB of a - b as above when a and b have the same sign. If a and b have opposite signs, then simply examine the sign bits of the operands.

To perform shift (<<, >>, and >>>), we create 5 stage mux which each stage result is a shift operation of operand\_a with each low bit of operand\_b. There is a example of SLL operation present in Figure 2. The operation SRL and SRA is design similar to the operation SLL.

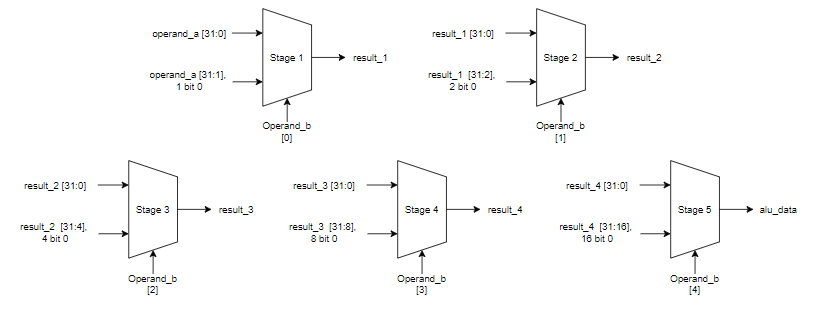


Figure 2: SLL operation

Table 1: The ALU Operation

|  |  |  |  |
| --- | --- | --- | --- |
| **alu\_op** | **Opcode** | **Description (R type)** | **Description (I type)** |
| ADD (0)  SUB (1)  SLT (2)  SLTU (3)  XOR (4)  OR (5)  AND (6)  SLL (7)  SRL (8)  SRA (9)  OPB (10) | 0000  0001  0010  0011  0100  0101  0110  0111  1000  1001  1010 | rd = rs1 + rs2  rd = rs1 − rs2  rd = (rs1 < rs2) ? 1 : 0  rd = (rs1 < rs2) ? 1 : 0  rd = rs1 ^ rs2  rd = rs1 | rs2  rd = rs1 & rs2  rd = rs1 << rs2[4:0]  rd = rs1 >> rs2[4:0]  rd = rs1 >>> rs2[4:0]  rd = rs2 | rd = rs1 + imm  n/a  rd = (rs1 < imm) ? 1 : 0  rd = (rs1 < imm) ? 1 : 0  rd = rs1 ^ imm  rd = rs1 | imm  rd = rs1 & imm  rd = rs1 << imm[4:0]  rd = rs1 >> imm[4:0]  rd = rs1 >>> imm[4:0]  rd = rs2 |

### Specification

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Width** | **Direction** | **Description** |
| i\_operand\_a  i\_operand\_b  i\_alu\_op  o\_alu\_data | 32  32  4  32 | Input  Input  Input  Output | First operand for ALU operations.  Second operand for ALU operations.  The operation to be performed.  Result of ALU operation |

## BRU

The BRU (branch comparison unit) compare two registers rs1 and rs2 with two function: the bit compara tor and the sign converter.

The input of this unit include of rs1 data, rs2 data from regfile and br\_un from control unit. The function is selected base on i\_br\_un signal (sign went active).

The BRU design requirement is “do not use subtraction (−), comparison (<, >).

If the operands a and b are unsigned integers, compare their MSB after performing the operation rs1\_data + ~rs2\_data + 1. If the overflow bit of operation is 1, then a < b (br\_less active); otherwise, a >= b (br\_less negative).

If the operands rs1\_data and rs2\_data are signed integers:

If both and b are either both positive or both negative, just use the MSB of operation rs1\_data + ~rs2\_data + 1 as result for br\_less.

If rs1\_data and rs2\_data have opposite signs, just use rs1\_data’s MSB as result of br\_less.

The br\_equal active when the result and overflow bit of a + (~b) + 1 equal to zero.

### Specification

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Width** | **Direction** | **Description** |
| i\_rs1\_data | 32 | Input | Data from the first register |
| i\_rs2\_data | 32 | Input | Data from the second register |
| i\_br\_un | 1 | Input | Comparison mode (1 if signed, 0 if unsigned). |
| o\_br\_less | 1 | Output | Output is 1 if *𝑟𝑠*1 *< 𝑟𝑠*2. |
| o\_br\_equal | 1 | Output | Output is 1 if *𝑟𝑠*1 = *𝑟𝑠*2. |

# Regfile

The register file is a critical component in a RISC-V processor, designed to store temporary values that are needed for the execution of instructions. Here’s an analysis of its operation, features, and considerations based on the provided specifications:

1. *Purpose and Functionality:*

The register file serves as a bank of 32 registers, each 32 bits wide, that stores operands and results of instructions.

It supports three simultaneous access operations: two read operations and one write operation. This means two registers can be read while data is written to a third register within the same clock cycle.

Register 0 is hardwired to always read as zero, which is typical in RISC architectures and useful for various operations like clearing registers or conditionally zeroing out data.

1. *Key Signals:*

Clock (i\_clk): Synchronizes all register operations, ensuring that reads and writes occur at specific clock edges.

Reset (i\_rst): Initializes all registers, typically setting them to zero during power-on or reset events.

Read Addresses (i\_rs1\_addr, i\_rs2\_addr): Specify which registers are to be read. These addresses are 5 bits wide, allowing access to 32 different registers.

Write Address (i\_rd\_addr): Specifies which register to write data to. This also uses a 5-bit address width.

Write Enable (i\_rd\_wren): Controls whether data is written to the register specified by i\_rd\_addr. If this signal is inactive, no data will be written, regardless of the address or data values.

1. *Operation and Control Logic:*

Read Operations: The register file outputs the values stored in the registers specified by i\_rs1\_addr and i\_rs2\_addr to o\_rs1\_data and o\_rs2\_data, respectively. These outputs are generated based on the provided addresses and are independent of the write operations, meaning reads can happen concurrently with writes.

* + 1. ***Specification***

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Width** | **Direction** | **Description** |
| i\_clk | 1 | Input | Global clock. |
| i\_rst | 1 | Input | Global active reset. |
| i\_rs1\_addr | 5 | Input | Address of the first source register. |
| i\_rs2\_addr | 5 | Input | Address of the second source register. |
| o\_rs1\_data | 32 | Output | Data from the first source register. |
| o\_rs2\_data | 32 | Output | Data from the second source register. |
| i\_rd\_addr | 5 | Input | Address of the destination register |
| i\_rd\_data | 32 | Input | Data to write to the destination register. |
| i\_rd\_wren | 1 | Input | Write enable for the destination register |

* 1. **I/O System and Memory:**

In this part, we implement a Load-Store Unit (LSU) which functions as an I/O System and Memory. According to Havard structure, we separate I/O system and Memory two part: Instruction Memory and Load Store Unit.

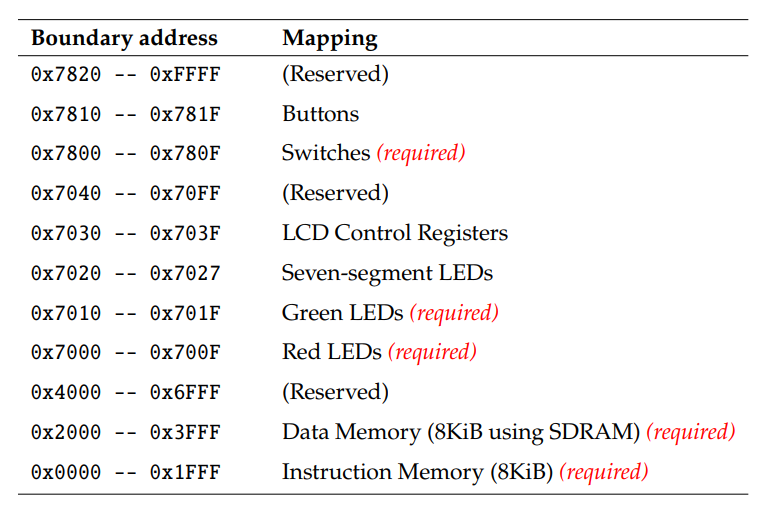


Figure 3: LSU Mapping

Instruction Memory is implemented as 2048 x 8 bit registers (8 KiB). An instruction has 32 bit in lengh, so we need four registers for an instruction. That means the distance between two instructions is four registers (or four addresss).

The LSU includes 5 main components: Decoder, Input\_Buffer, Output\_Buffer and Data\_Memory that present in Figure 4.

To prevent misaligned, LSU’s address need to clear two LSB for SW, LW operation or clear LSB for SH, SHU, LH.

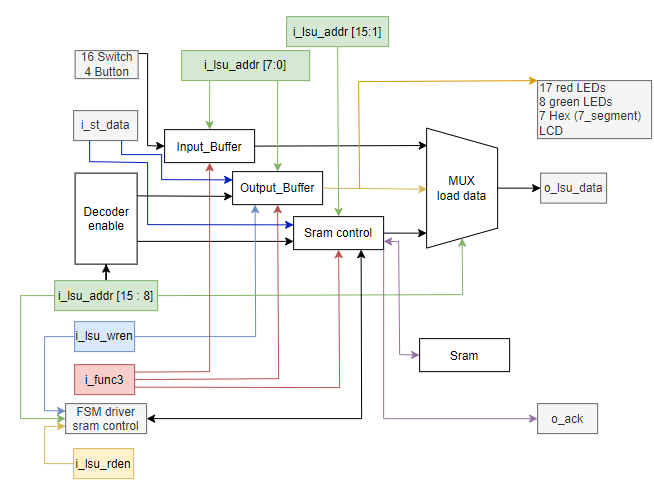


Figure 4: LSU Diagram.

LSU’s address is range 0x2000 to 0x781F, which represented in 16 bits, but i\_lsu\_data represented in 32 bits (4 byte). We only use 2 byte low of LSU address and don’t care the other.

DECODER: base on i\_lsu\_adder [15 : 8] (second byte) to select the store region (Output\_buffer and Data\_Memory).

MUX: base on i\_lsu\_adder [15 : 8] (second byte) to select the load region (Input\_buffer, Output\_buffer and Data\_Memory).

\*I/O System

The proccessor interacts with environment through Input\_Buffer and Output\_Buffer.

Input\_Buffer and Output\_Buffer is implemented as 8-bit register array (similar to Instruction Memory).

Input\_Buffer: contains value of Switch and Button Signal. When we interact with Switch and Button that reflected in the region of Input\_Buffer (the address of register). It’s address is in range 0x7800 to 0x781F. The Input\_Buffer is implemented as 32 x 8 bit-registers (32 byte). Because we use the second byte to select region of LSU, we just use the first byte as address for interact with Input\_Buffer. We can’t store data in Input\_Buffer, only load the data from Input\_Buffer.

Output\_Buffer: is connected to the red LEDs, green LEDs, 7-segment displays and LCD1602. It’s address is in range 0x7000 to 0x703F. The Output\_Buffer is implemented as 64 x 8 bit-registers (64 byte). Similar to Input\_Buffer, we just use the first byte as address for interact with Out\_Buffer. We can load or store data in Output\_Buffer.

\*Data Memory

The Data Memory is used to store data and load it out when needed. It’s address is in range 0x2000 to 0x3FFF. In this implementation, we use sram IS61WV25616 replace for Data\_Memory.

We use the sram\_control module which is provided (2 cycle for write and 5 cycle for read).

According to the datasheet, this sram is 2 bytes data width corresponding to an address. We need to adjust the LSU’s address that fit to sram’s address. Solution is shift right LSU’s address one bit, we have sram’s address. Because one LSU’s address can be stored one byte data, but one sram’s address can be stored two byte data. So the LSU’s address is “double” than the sram’s address. An example is presented in Figure 5.

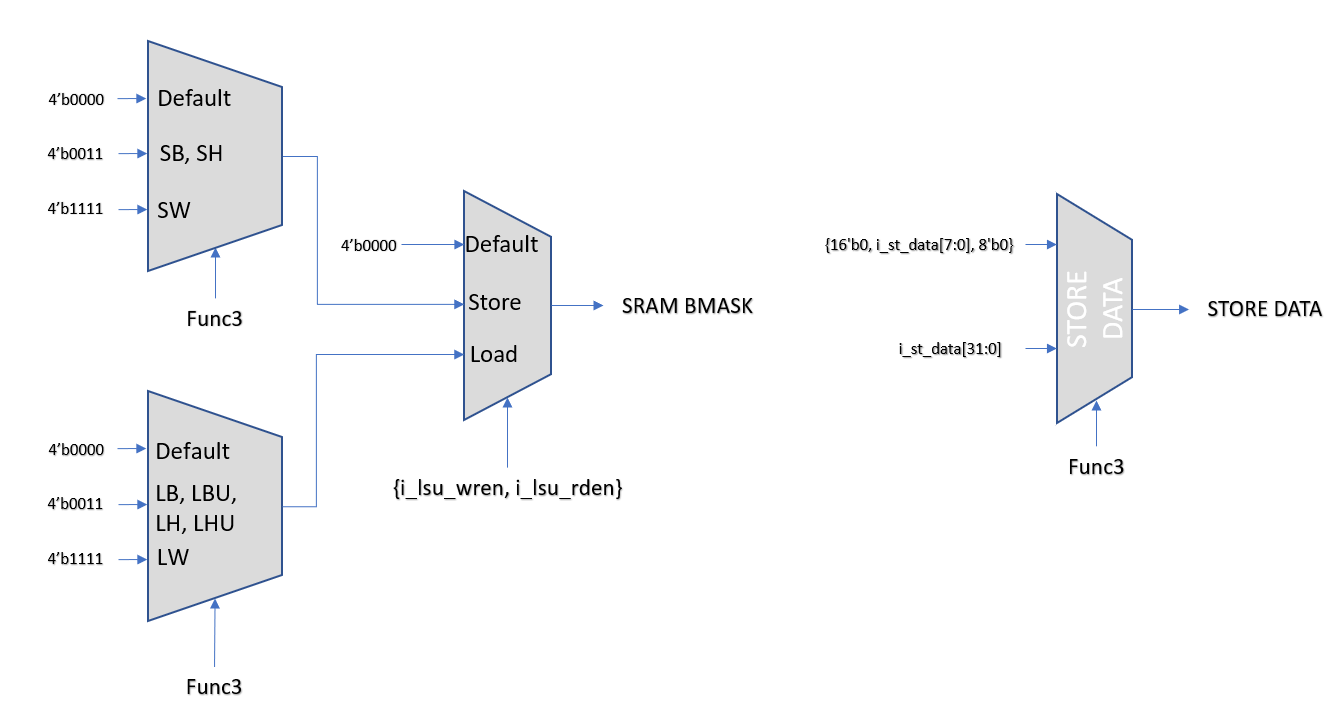


Figure 5: Example of relation between LSU’s address and Sram address

Sram has 18 bits address width, we just use 15 bit low, so 3 bit high set to zero.

Because the sram cause 2 or 5 cycle for store or load and the design using i\_lsu\_wren for control both store and load. In some sistuation, the result of ALU in range of x2000 to 0x3FFF (Sram’s address region) and this cause something can’t control because i\_lsu\_wren negative is read operation. So the Sram need one more signal (i\_lsu\_rden) for more stable control. When both i\_lsu\_wren and i\_lsu\_rden negative, the sram do nothing.

To driver this module, we design a combination logic for driver BMASK and store data by i\_lsu\_wren, i\_lsu\_rden and func\_3 (sw, sb, sh, lw, lb, lbu, lh, lhu) that present in Figure 6 and a ASM for driver WREN and RDEN that is present in Figure 7.

Figure 6: Combinational logic of bit mask and store data control

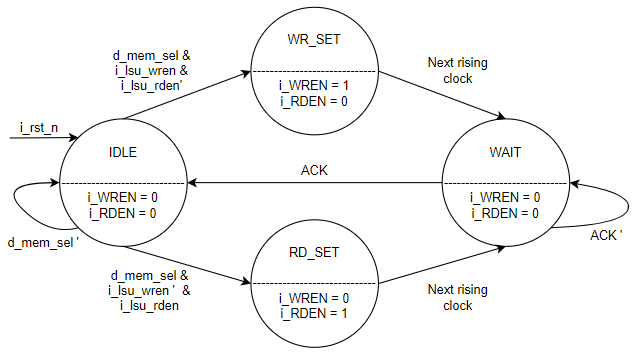


Figure 7: FSM Driver Sram Control.

Because the operation delay 2 cycle for write and 5 cycle for read, we need to hold the PC value, wait for the operation completely. Solution is design a combinational logic that can stop PC update next value after each cycle which present in Figure 8.

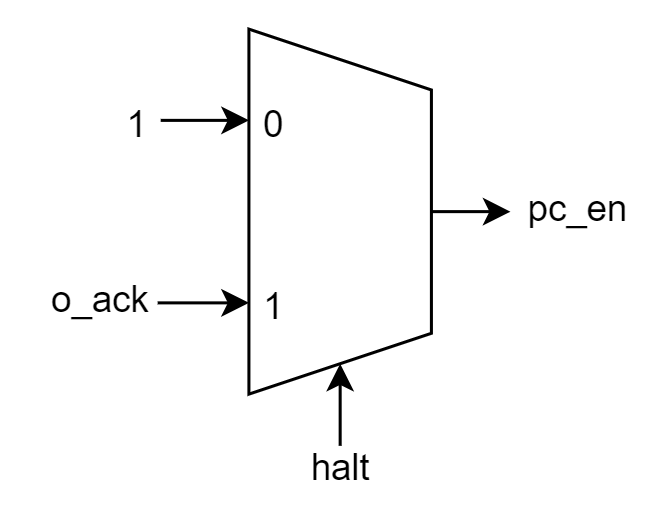


Figure 8: Combinational logic of control program counter

The halt signal is active when meet these condition: Opcode is of R-format layout (Store) or I-format load and i\_lsu\_adder in Data\_Memory’s address range (0x2000 to 0x3FFF).

The ack signal is feelback signal of sram control module. When it’s active means the operation completely and allow PC update next value.

***2.5.1 Specification***

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Width** | **Direction** | **Description** |
| i\_clk  i\_rst\_n  i\_lsu\_addr  i\_st\_data  i\_lsu\_wren  i\_func\_3  o\_ld\_data  o\_io\_ledr  o\_io\_ledg  o\_io\_hex0…7  o\_io\_lcd  i\_io\_sw  i\_io\_btn  o\_ack | 1  1  32  32  1  3  32  32  32  7  32  32  4  1 | input  input  input  input  input  input  output  output  output  output  output  input  input  output | Globa clock, active on the rising edge.  Globa low active reset.  Address for data read or write.  Data to be store.  Write enable signal (1 if writting).  Select function sw, sb, sh, lw, lb, lbu, lh, lhu.  Data read from memory.  Output for red LEDs.  Output for green LEDs.  Output for 7-segment displays.  Output for the LCD register.  Input for switch.  Input for button.  Signal to pc\_halt\_by\_sram |

* 1. **Control Unit:**

Based on the RISCV ISA instruction set table, the team will design the control unit, based on opcode (inst[6:2]), funtion 3 (inst[14:12]), function 7 (inst[30]), to decide the control signals according to the table following statistics:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| INSTRUCTION | TYPE | OPCODE [6:2] | funct7 [30] | funct3 [14:12] | pc\_sel | rd\_wren | br\_un | br\_less | br\_equal | opa\_sel | opb\_sel | alu\_op | mem\_rd | mem\_wren | wb\_sel | data\_out\_addj |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ADD | R | 01100 | 0 | 000 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | ADD | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SUB | R | 01100 | 1 | 000 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | SUB | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SLL | R | 01100 | 0 | 001 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | SLL | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SLT | R | 01100 | 0 | 010 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | SLT | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SLTU | R | 01100 | 0 | 011 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | SLTU | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| XOR | R | 01100 | 0 | 100 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | XOR | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SRL | R | 01100 | 0 | 101 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | SRL | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SRA | R | 01100 | 1 | 101 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | SRA | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| OR | R | 01100 | 0 | 110 | PC +4 | allow | x | x | x | rs1\_data | rs2\_data | OR | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| AND | R | 01100 | 0 | 111 | PC +4 | allow | x | x | x | rs1\_data | rs2\_data | AND | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ADDI | I | 00100 | x | 000 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | ADD | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SLLI | I | 00100 | x | 001 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | SLL | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SLTI | I | 00100 | x | 010 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | SLT | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SLTIU | I | 00100 | x | 011 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | SLTU | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| XORI | I | 00100 | x | 100 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | XOR | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SRLI | I | 00100 | x | 101 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | SRL | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SRAI | I | 00100 | 1 | 101 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | SRA | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ORI | I | 00100 | x | 110 | PC + 4 | allow | x | x | x | rs1\_data | rs2\_data | OR | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ANDI | I | 00100 | x | 111 | PC +4 | allow | x | x | x | rs1\_data | rs2\_data | AND | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| LB | I | 00000 | x | 000 | PC + 4 | allow | x | x | x | rs1\_data | imm\_data | ADD | allow | unallow | LSU\_data | {24{data[7]},data[7:0]} |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| LH | I | 00000 | x | 001 | PC + 4 | allow | x | x | x | rs1\_data | imm\_data | ADD | allow | unallow | LSU\_data | {16{data[15]},data[15:0]} |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| LW | I | 00000 | x | 010 | PC + 4 | allow | x | x | x | rs1\_data | imm\_data | ADD | allow | unallow | LSU\_data | data[31:0] |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| LBU | I | 00000 | x | 100 | PC + 4 | allow | x | x | x | rs1\_data | imm\_data | ADD | allow | unallow | LSU\_data | {24'b0,data[7:0]} |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| LHU | I | 00000 | x | 101 | PC + 4 | allow | x | x | x | rs1\_data | imm\_data | ADD | allow | unallow | LSU\_data | {16'b0,data[15:0]} |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SB | S | 01000 | x | 000 | PC + 4 | unallow | x | x | x | rs1\_data | imm\_data | ADD | unallow | allow | x | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SH | S | 01000 | x | 001 | PC + 4 | unallow | x | x | x | rs1\_data | imm\_data | ADD | unallow | allow | x | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| SW | S | 01000 | x | 010 | PC + 4 | unallow | x | x | x | rs1\_data | imm\_data | ADD | unallow | allow | x | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| BEQ | B | 11000 | x | 000 | br\_equal | unallow | sign | x | 1 | PC | imm\_data | ADD | unallow | unallow | x | x |
|  | B | 11000 | x | 000 | br\_equal | unallow | sign | x | 0 | PC | imm\_data | ADD | unallow | unallow | x | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| BNE | B | 11000 | x | 001 | !br\_equal | unallow | sign | x | 0 | PC | imm\_data | ADD | unallow | unallow | x | x |
|  | B | 11000 | x | 001 | !br\_equal | unallow | sign | x | 1 | PC | imm\_data | ADD | unallow | unallow | x | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| BLT | B | 11000 | x | 100 | br\_less | unallow | sign | 1 | 0 | PC | imm\_data | ADD | unallow | unallow | x | x |
|  | B | 11000 | x | 100 | br\_less | unallow | sign | 0 | x | PC | imm\_data | ADD | unallow | unallow | x | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| BGE | B | 11000 | x | 101 | !br\_less | unallow | sign | 0 | x | PC | imm\_data | ADD | unallow | unallow | x | x |
|  | B | 11000 | x | 101 | !br\_less | unallow | sign | 1 | 0 | PC | imm\_data | ADD | unallow | unallow | x | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| BLTU | B | 11000 | x | 110 | br\_less | unallow | unsign | 1 | 0 | PC | imm\_data | ADD | unallow | unallow | x | x |
|  | B | 11000 | x | 110 | br\_less | unallow | unsign | 0 | x | PC | imm\_data | ADD | unallow | unallow | x | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| BGEU | B | 11000 | x | 111 | !br\_less | unallow | unsign | 0 | x | PC | imm\_data | ADD | unallow | unallow | x | x |
|  | B | 11000 | x | 111 | !br\_less | unallow | unsign | 1 | 0 | PC | imm\_data | ADD | unallow | unallow | x | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| JAL | J-U | 11011 | x | x | PC + imm (alu) | allow | x | x | x | PC | imm\_data | ADD | unallow | unallow | PC + 4 | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| JALR | J-I | 11001 | x | x | PC + imm (alu) | allow | x | x | x | rs1\_data | imm\_data | ADD | unallow | unallow | PC + 4 | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| AUIPC | U | 00101 | x | x | PC + 4 | allow | x | x | x | rs1\_data | imm\_data | ADD | unallow | unallow | alu\_data | x |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| LUI | U | 01101 | x | x | PC + 4 | allow | x | x | x | x | imm\_data | operand b | unallow | unallow | alu\_data | x |
|  | | | | | | | | | | | | | | | | |

* + 1. ***Specification***

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Width** | **Direction** | **Description** |
| i\_instruction  i\_br\_less  i\_br\_equal  o\_pc\_sel  o\_rd\_wren  o\_insn\_vld  o\_br\_un  o\_opa\_sel  o\_opb\_sel  o\_mem\_wren  o\_mem\_rd\_en  o\_alu\_op  o\_wb\_sel | 32  1  1  1  1  1  1  1  1  1  1  4  2 | Input  Input  Input  Output  Output  Output  Output  Output  Output  Output  Output  Output  Output | Data from Instruction Memory.  Result of equal comparison from BRC  Result of less comparison from BRC  Signal control program counter  Signal control write enable for regfile  Signal anounce instruction valid or not  Signal control comparison sign or unsign for BRC  Signal select input data for the ALU (rs1 or pc)  Signal select input data for the ALU (rs2 or imm)  Enable writing for data memory  Enable read only for sram  Select the operation for ALU  Select the write back date for regfile |

* 1. **Immediate Generator**

The implementation of the immediate generator is a critical component in the design of a RISC-V CPU, responsible for arranging the bits or bytes of instructions to form the appropriate immediate values. In the RISC-V format, there are various types ofimmediates, including I, B, S, J, and U formats, which require the use of multiplexers to direct the wiring order of the bus. The figure illustrates a design that guides the bus of immediates for these different formats.

The I-format serves as the starting point, requiring minimal sign extension and acting as a reference for subsequent stages. The S-stage, identified by a MUX, transforms the bus into an S-type immediate, distinct from the I-type. Similar approaches are applied to B-type masking over S-type and J-type masking over J-type, with special note taken for the commonality of the last bits in J and B types, which are handled separately to reduce the need for additional MUXes.

The U-type is entirely different from the others, and the chosen mechanism is applied to determine whether it is a U-type or part of the multi-stage ISBJ formats. The selection of the immediate bus path is governed by the imm sel port, which is protocol decoded in a one-hot format with 5 bits, ensuring precise and effective control of immediate generation.

The instruction 32 bit in Single Cycle RISCV CPU is composed of different fields that encode the operation, the operands and the destination of the result. One of these fields is the bit that, which indicates whether the instruction is an immediate or a register instruction. The table below shows some summary of how the bit that affects the generation of immediate values from the instruction 32 bit in Single Cycle RISCV CPU.

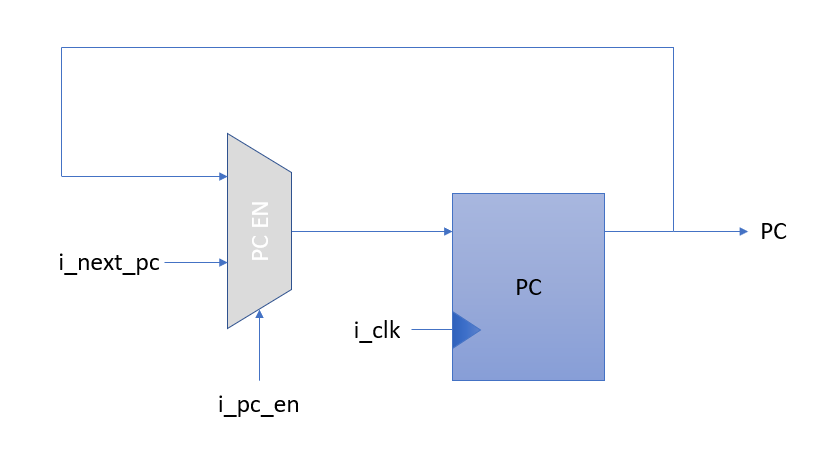
|  |  |
| --- | --- |
| **Instruction Type** | **Bit Extraction** |
| I-Type | {{21{i\_inst[31]}}, i\_inst[30:20]} |
| S-Type | {{21{i\_inst[31]}}, i\_inst[30:25], i\_inst[11:7]} |
| B-Type | {{20{i\_inst[31]}}, i\_inst[7], i\_inst[30:25], i\_inst[11:8],1'b0} |
| U-Type | {{12{i\_inst[31]}}, i\_inst[19:12], i\_inst[20], i\_inst[30:21],1'b0} |
| J-Type | {i\_inst[31:12], 12'b0} |

* + 1. ***Specification***

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Width** | **Direction** | **Description** |
| i\_instruction  o\_imm | 32  32 | Input  Output | Data from Instruction Memory.  Output data of ImmGen |

* 1. **Program counter**

Program counter degisn as a sequential system. At the rising of clock, PC will update new PC value (PC + 4 or PC + immediate). Because the sram cause 2 or 5 cycle delay for store or load, we add a enable signal for halt the PC. When the enable signal negactive, PC hold it’s value and can’t update new PC value until active.

****Figure 8: Program Counter

* + 1. ***Specification***

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Width** | **Direction** | **Description** |
| i\_next\_pc  i\_pc\_en  o\_pc | 32  1  32 | Input  Input  Output | New PC value  Signal hold PC value  Output data of ImmGen |

# Verification Strategy

* 1. **ALU**

To verify the function of ALU, randomly the value of operand a and operand b, then compare the expected result with the output value of ALU corresponding to each operands case. Total sample around 500 testcase.

=== Starting Random Tests ===

OP\_SLT 2: 37 < 24 ? 0 (ALU output: 0)- PASS

OP\_SUB 1: 17- 4294967291 = 22 (ALU output: 22)- PASS

OP\_AND 6: fffffff8 & 13 = 10 (ALU output: 10)- PASS

OP\_AND 6: fffffff6 & ffffffe1 = ffffffe0 (ALU output: ffffffe0)- PASS

OP\_SLL 7: ffffffc5 << 23 = e2800000 (ALU output: e2800000) PASS

OP\_ADD 0: 4294967294 + 6 = 4 (ALU output: 4)- PASS

OP\_OUTPUT\_B 10: Operand B = 55 (ALU output: 55)- PASS

OP\_AND 6: c & 20 = 0 (ALU output: 0)- PASS

…..

OP\_SRA 9: 6 >>> 16 = 0 (ALU output: 0)- PASS

OP\_AND 6: ffffffe6 & 41 = 40 (ALU output: 40)- PASS

OP\_OUTPUT\_B 10: Operand B = 4294967239 (ALU output: 4294967239)- PASS

OP\_SLTU 3: 25 < 12 ? 0 (ALU output: 0)- PASS

Test Summary:

Total test cases: 500

Passed: 500

Failed: 0

Pass rate: 100.00%

## 

## BRC

To verify the function of BRC, design two test bench for sign and unsign. Randomly the value of rs1 data and rs2 data, then compare the expected result with the the output value of BRC. Total sample around 500 testcase.

=== Starting Random Tests ===

Random Signed Test PASSED - a: 303379748, b: -1064739199, unsigned: 0

Random Signed Test PASSED - a: -2071669239, b: -1309649309, unsigned: 0

Random Signed Test PASSED - a: 112818957, b: 1189058957, unsigned: 0

Random Signed Test PASSED - a: -1295874971, b: -1992863214, unsigned: 0

Random Signed Test PASSED - a: 777537884, b: -561108803, unsigned: 0

….

Random Unsigned Test PASSED - a: -359791147, b: 220526106, unsigned: 1

Random Unsigned Test PASSED - a: 1559232697, b: -1685626569, unsigned: 1

Random Unsigned Test PASSED - a: -1084580994, b: 1837990107, unsigned: 1

Random Unsigned Test PASSED - a: 1740879567, b: -1125307527, unsigned: 1

Random Unsigned Test PASSED - a: 2099832058, b: 817688161, unsigned: 1

Random Unsigned Test PASSED - a: -1953232617, b: 1358264481, unsigned: 1

=== Test Summary ===

Total Tests: 500

Passed: 500

Failed: 0

Pass rate: 100.00%

# 

# Regfile

To verify the function of Register File, design a test bench that can be check value of all register equal to zero after reset, check the value of register 0 always equal to zero and check all register after receive a value feedback (depend on rd\_en).

=== Starting Tests ===

PASS: All registers reset to 0

PASS: Register[1] = a5a5a5a5 Expected = a5a5a5a5

PASS: Register[2] = 5a5a5a5a Expected = 5a5a5a5a

PASS: Register[3] = 12345678 Expected = 12345678

……

PASS: Register[31] = faceb00c Expected = faceb00c

PASS: Register[0] = 00000000 Expected = 00000000

PASS: Register x0 = 0 (unchanged)

=== Test Summary ===

Total Tests: 50

Passed: 50

Failed: 0

Pass rate: 100.00%

* 1. **I/O system and memory**

To verify the operation of the Load-Store Unit (LSU), a driver will be created to generate random input stimuli as follows:

Randomized Address Selection: Randomly assign values to dut -> i\_lsu\_addr within the LSU's addressable ranges (0x2000–0x781F). This will include generating addresses for the Input\_Buffer (0x7800–0x781F), Output\_Buffer (0x7000–0x703F), and Data\_Memory (0x2000–0x3FFF) (replace for sram because we can’t verification it). For each address range, 32-bit random data (dut -> i\_st\_data) will be generated to verify both read and write operations across LSU regions.

Randomized Write Enable and Reset Signals: Randomly set the dut -> i\_lsu\_wren signal to 1 or 0 to toggle between read and write operations with an equal probability (50%) for each. Toggle the i\_rst\_n signal periodically to ensure that the LSU can properly reset and resume normal operation, as described in the test setup.

Output Assertions:

Use assertions to verify that the LSU outputs match expected behaviors based on address decoding and data operations. For example: Check that data written to Output\_Buffer or Data\_Memory can be read back correctly. Assert that o\_ld\_data (for memory read operations) and o\_io\_\* signals (for I/O operations) are consistent with the expected results, ensuring correct LSU functionality.For Input\_Buffer, assert that only reads occur and data is not modified by any write operations.

This verification setup ensures comprehensive testing of the LSU’s operation, covering a wide range of normal and edge-case scenarios to validate functionality under various input conditions.

=== Starting Tests ===

PASS: Write Test - Addr: 00007000, Data: 12345678, LEDR: 12345678

PASS: Write Test - Addr: 00007002, Data: 0000abcd, LEDR: abcd5678

PASS: Write Test - Addr: 00007001, Data: 000000ef, LEDR: abcdef78

PASS: Read Test - Addr: 00007800, Control: 010, Data: 89abcdef

PASS: Read Test - Addr: 00007802, Control: 001, Data: ffff89ab

PASS: Read Test - Addr: 00007803, Control: 000, Data: ffffff89

….

PASS: Read Test - Addr: 00007803, Control: 100, Data: 00000089

PASS: Read Test - Addr: 00007802, Control: 101, Data: 000089ab

PASS: Write Test - Addr: 00007010, Data: faceb00c, LEDR: abcdef78

PASS: Write Test - Addr: 00007020, Data: 0000005a, LEDR: abcdef78

PASS: Read Test - Addr: 00007020, Control: 000, Data: 0000005a

=== Test Summary ===

Total Tests: 500

Passed: 500

Failed: 0

Pass rate: 100.00%

* 1. **Control Unit**

To verify the functionality of the Control Unit, we will use a driver to generate random input stimuli as described below:

Simulation Count: Run MAX\_SIM = 2000 simulations to thoroughly test the Control Unit across a broad spectrum of cases, including both typical and edge-case scenarios.

Random Operation Mode Selection:

Randomly set the unsigned/signed mode to verify that the Control Unit can correctly handle both signed and unsigned operations. This is essential for testing the Control Unit's ability to perform comparisons and branching accurately, regardless of operand sign.

Random Operand Generation:

Generate random 32-bit values for the two operands, ensuring diverse input values. This includes testing with both small and large numbers to cover a full range of potential inputs. This approach allows us to examine how the Control Unit handles various operand values in different operations.

Output Assertions:

Use assertions to check that the output signals (such as the comparison results for less-than or equal) behave as expected based on the operation mode and operand values. This helps verify that the Control Unit is performing correct branching and comparison operations:

For unsigned operations, confirm that the output signals accurately reflect the comparison between the operands.

For signed operations, ensure that the outputs align with expected behavior for signed values.

Assertions will cover conditions where the first operand is less than, greater than, or equal to the second operand, confirming that the Control Unit sets the output signals correctly

ADD : PASS SUB : PASS SLL : PASS SLL: PASS

SLTU : PASS XOR : PASS SRL : PASS SRA : PASS

OR : PASS AND : PASS SW : PASS LB : PASS

LH : PASS LW : PASS LBU : PASS LHU : PASS

BEQ : PASS BNE : PASS BLT : PASS BGE : PASS

ADDI : PASS SLTI : PASS SLTIU : PASS XORI : PASS

ORI : PASS ANDI : PASS SLLI : PASS SRAI : PASS

SH : PASS BLTU : PASS BGEU : PASS LUI : PASS

AUIPC : PASS JAL : PASS JALR : PASS SRLI : PASS

SB : PASS

* 1. **Immediate Generator**

To verify the functionality of the Immediate Generator, a driver will be created to generate various types of input stimuli, covering both typical and edge-case scenarios:

Simulation Count: Set the number of simulations to MAX\_SIM = 2000 to ensure thorough testing across multiple cases and immediate types.

Randomized Immediate Type Selection:

Randomly select the type of immediate (e.g., I-type, S-type, B-type, U-type, and J-type) to verify the Immediate Generator’s handling of each type. This tests its ability to correctly interpret and generate immediate values based on the instruction format.

Randomized Instruction Input Generation:

Generate random 32-bit values for the instruction input to simulate various possible instructions. The randomized instructions will contain different bit patterns, allowing comprehensive testing of how the Immediate Generator extracts and extends immediate fields according to each instruction type’s format.

Output Assertions:

Use assertions to validate that the generated immediate values match the expected results based on the selected instruction type. Each type has a unique format for immediate extraction and sign extension, so the assertions will verify:

For I-type, that the 12-bit immediate is correctly sign-extended to 32 bits.

For S-type, that the immediate is accurately constructed by combining bits from two different fields, and then sign-extended.

For B-type, that the immediate is assembled with correct bit shifting and sign extension for branching.

For U-type, that the upper 20 bits are loaded as-is, with the lower bits set to zero.

For J-type, that the immediate is correctly assembled and extended for jump instructions.

Starting tests for imm\_gen module...

<Test: ADD >0: Instruction 002081b3, Calculated Imm = 00000000

<Test: SUB>1: Instruction 402081b3, Calculated Imm = 00000000

<Test: SLL>2: Instruction 002091b3, Calculated Imm = 00000000

<Test: SLT>3: Instruction 0020a1b3, Calculated Imm = 00000000

<Test: SLTU >4: Instruction 0020b1b3, Calculated Imm = 00000000

<Test: XOR>5: Instruction 0020c1b3, Calculated Imm = 00000000

<Test: SRL>6: Instruction 0020d1b3, Calculated Imm = 00000000

…..

<Test: SRA>7: Instruction 4020d1b3, Calculated Imm = 00000000

<Test: OR >8: Instruction 0020e1b3, Calculated Imm = 00000000

<Test: AND>9: Instruction 0020f1b3, Calculated Imm = 00000000

<Test: ADDI >10: Instruction = 12608193, Calculated Imm = 0000012b

<Test: SLTI > 0000012b11: Instruction = 12b0a193, Calculated Imm = 0000012b

<Test: XORI > 0000012b12: Instruction. 12b0c193, Calculated Imm = 0000012b

=== Test Summary ===

Total Tests: 500

Passed: 500

Failed: 0

Pass rate: 100.00%

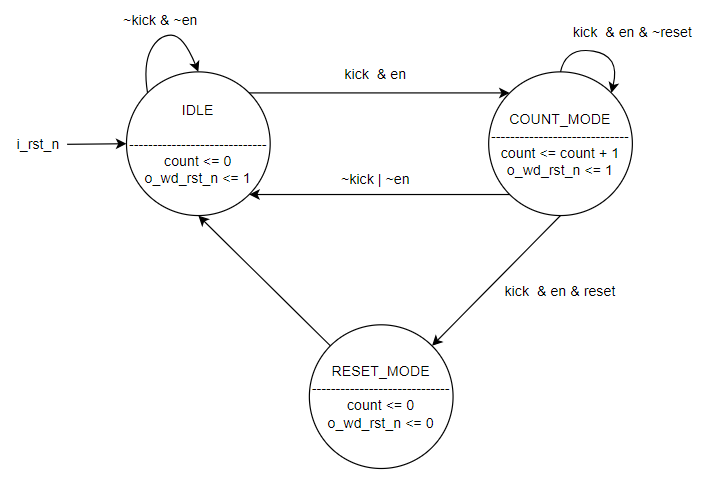
# Alternative Design

Watchdog Timer (WDT) is a circuit or component in embedded systems and computers designed to monitor system activity and detect faults, such as freezes or unresponsiveness. If the system fails to operate correctly within a specified period, the WDT takes action to restore functionality, typically by restarting the system.

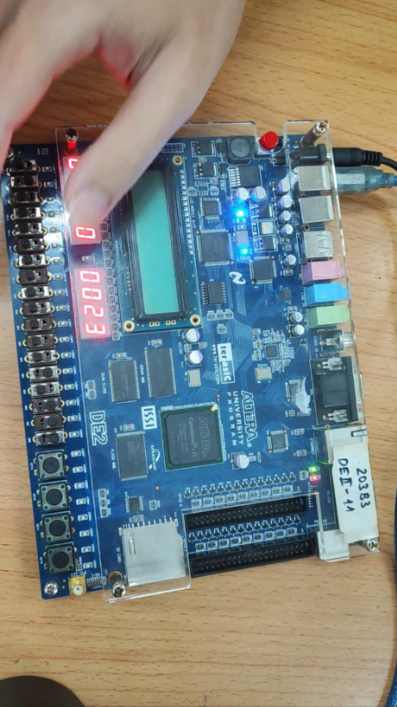
The Watchdog Timer is configured with a specific timeout period. The system must periodically send a signal (often called a "kick" or "feed") to the WDT before the timeout expires. This indicates the system is functioning normally. If the WDT doesn’t receive the signal within the timeout period, it assumes the system has malfunctioned or frozen. When the timeout occurs, the WDT typically triggers action that reset the system and active a signal error (LED, buzzer,…).

In this implementation, WDT is designed as sequential system with four input: i\_en, i\_kick, i\_clk, i\_rst\_n and one output: o\_wdt\_rst\_n. At the rising of clock, i\_kick active when present program counter not equal to next program counter.

If i\_en active, WDT moniter the i\_kick. If the i\_kick active, WDT increase the counter variable by one. When the counter equal to the set value (ex: 5, 10, ….), WDT send a signal to reset the processor and repeat the loop. We provide more detail of WDT in the report of Milestone 3.

Figure 9: FSM of WDT design

1. **FPGA Implement:**
   1. Stopwatch using seven-segment LEDs as the display

Figure 9: Stopwatch using seven-segment LEDs as the display

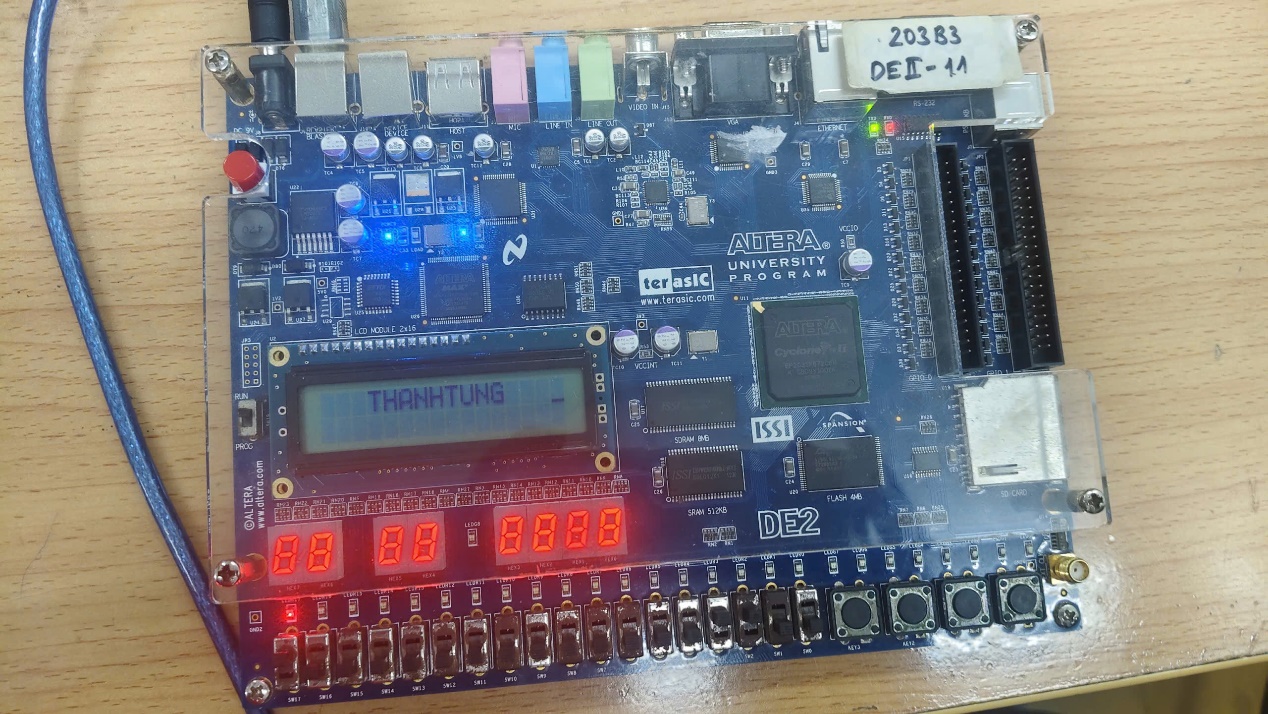
* 1. Display multiline text on the LCD

Figure 10: Text on the LCD

# Evaluation

|  |  |  |
| --- | --- | --- |
| TASK | | EVALUATION |
| Baseline Functionality | Arithmatic Logic Unit | Done |
| Branch Comparison Unit | Done |
| Register File | Done |
| Instruction Memory (8KiB) | Done |
| Input Buffer | Done |
| Output Buffer | Done |
| Sram (8KiB) | Done |
| Control Unit | Done |
| Immediate Generator | Done |
| Alternative Design | Watch Dog Timer |  |
| Verification | Arithmatic Logic Unit | Done |
| Branch Comparison Unit | Done |
| Register File | Done |
| Instruction Memory (8KiB) | Done |
| Input Buffer | Done |
| Output Buffer | Done |
| Sram (using Data Memory 8KiB as an alternative) | Done |
| Control Unit | Done |
| Immediate Generator | Done |
| Hardware Implementation Program | Stopwatch using seven-segment LEDs as the display. | Done |
| Display multiline text on the LCD | Done |

# Result

* 1. **Grand test result:**

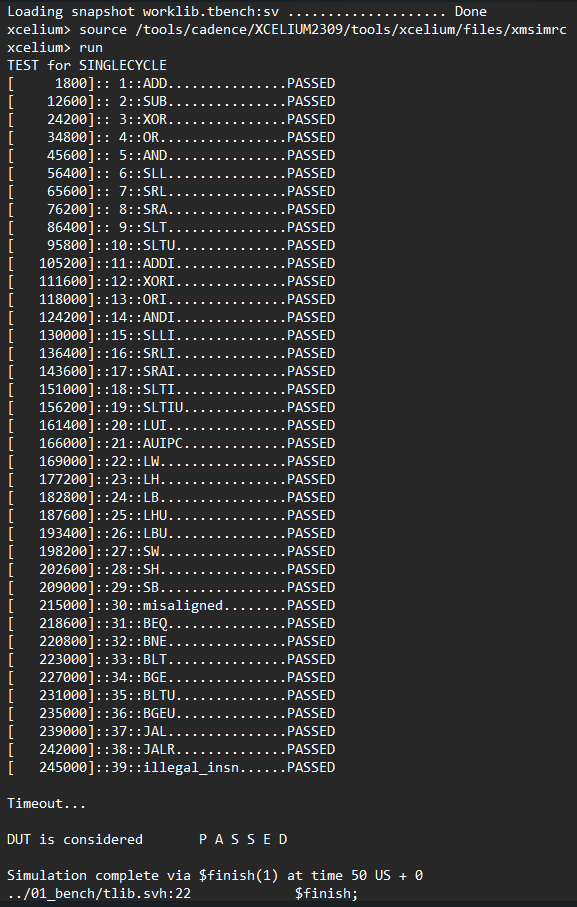


Figure 11: xrun log file

## Quartus

|  |  |  |
| --- | --- | --- |
| Logic Utilization/Total logic elements | 5037/33.216 | 15.16% |

# 

# References.

Patterson, L. .COMPUTER ORGANIZATION AND DESGIN THE HARDWARE/SOFTWARE INTERFACE RISC-V EDITON, Page 243.