**MIPS R-Type Instruction Processor for add, addu, sub, and subu**

**CS 34200, Computer Organization**

**Taught by Prof. Izidor Gertner**

**By David Zhang**

**Table of Contents**

[**Abstract 1**](#_9rfeqwjgsxbs)

[**Introduction 1**](#_ic2fiasq4jtj)

[**Background and Literature Review 1**](#_lfqm8p3z8cq6)

[**Components 2**](#_s9epvbb1bamf)

[**Instruction Register 3**](#_mgkdlr6rhtrd)

[**Instruction Decoder 3**](#_x3s3urfzlofr)

[**32 32-Bit Register Files 4**](#_dk5ywavsvwn5)

[**32-bit adder/subtractor 5**](#_tk8wb55u3n1x)

[**1-bit adder/subtractor 6**](#_o848taqq6nj0)

[**Full Adder 6**](#_nbc8kvxkcswm)

[**Top Level Processor 7**](#_rfqulrmdhwwr)

[**Methodology/discussion 7**](#_dwyzde6fykjr)

[**Testing/Waveforms 8**](#_noqbsnyyu26)

[**Conclusion 10**](#_cco7ghyg294o)

[**Appendices 11**](#_q28ziv9om38a)

[**Appendix A; IR code 11**](#_bn88e76hwn8i)

[**Appendix B; Instruction decoder code 11**](#_yeul9nw6kwm9)

[**Appendix C; Register File code 13**](#_qkjudyfnq7z2)

[**Appendix D; Register package code 14**](#_cfjdkbaz8fbp)

[**Appendix E; Adder/subtractor code 15**](#_cwlh5nj38vs7)

[**Appendix F; One-bit adder/subtractor code 17**](#_8v2vrxhabau7)

[**Appendix G; Full adder code 18**](#_yrtdw169p6k1)

[**Appendix H; Adder/subtractor package 18**](#_hbub0bjh7kgq)

[**Appendix I; Top level code 19**](#_rmsl0p576o26)

[**Appendix J; Processor Testbench**](#_whxb4e2nheex) **21**

# **Abstract**

This report presents the design and implementation of a MIPS processor that supports R-type arithmetic instructions, specifically add, addu, sub, and subu. The processor architecture consists of 32 general-purpose registers, each 32 bits wide, enabling efficient computation and data manipulation. The project focuses on designing a streamlined datapath and control unit to execute these instructions correctly while adhering to MIPS r-type instruction conventions. By simulating and testing the processor, we validate its correctness and performance. The results demonstrate the successful execution of the target instructions, paving the way for potential extensions such as additional instruction support.

# **Introduction**

For this project, I have designed and simulated a MIPS processor exclusively for the instructions add, addu, sub, and subu in Quartus Prime and Modelsim using VHDL. I would later delve into the specific design implementation chosen in the project and shortcomings/improvements that the implementation has.

# **Background and Literature Review**

MIPS (Microprocessor without Interlocked Pipeline Stages) is a widely studied RISC (Reduced Instruction Set Computing) architecture known for its simplicity and efficiency. Developed in the 1980s, MIPS architecture has played a significant role in academic and industrial applications, providing a foundational model for learning computer architecture and processor design.

The MIPS instruction set consists of three main categories: R-type (register-based), I-type (immediate-based), and J-type (jump-based) instructions. R-type instructions, which this project focuses on, involve operations that require three registers: two source registers and one destination register. These include arithmetic operations such as add (addition with overflow detection), addu (unsigned addition without overflow detection), sub (subtraction with overflow detection), and subu (unsigned subtraction without overflow detection). These instructions are fundamental to arithmetic computations and play an important role in processor functionality.

Several hardware description languages (HDLs), such as Verilog and VHDL, have been used to implement MIPS processors in simulation environments. The choice of implementation strategy impacts factors such as execution speed, hardware complexity, and power consumption. This project adopts a structured approach to designing a MIPS processor that effectively supports R-type arithmetic instructions, emphasizing correctness and efficiency in execution.

# 

# **Components**

The add/sub processor is split up into multiple components:

* The Instruction Register
* The instruction Decoder
* The 32 32-bit register files
* 32-bit adder/subtractor (ALU.vhd)
  + 1-bit adder/subtrator
    - Full-adder
* Top-level processor

## 

## **Instruction Register**

A single 32-bit register which reads a 32-bit vector from input on a rising clock edge and sends it out to the instruction decoder.



## **Instruction Decoder**

A custom decoder that takes the 32-bit vector from the Instruction register and parses the input into sections according to the MIPS r-type instruction format. Specific signals pertaining to the operation will then be passed to the register files to send the correct data to be processed by the adder and subtractor.



## 

## **32 32-Bit Register Files**

A 3 port register file that takes addresses (Ra and Rb) of two source registers to be sent to the processor and the address (Rw) of a destination register to store the result. This was constructed using an array of 32-bit vectors that uses a built-in decoder to transform the binary address to an integer corresponding to the desired element in the register array.

## 

## **32-bit adder/subtractor**

This component takes in control signals from the instruction decoder and determines the operation to perform on the two numbers. It then uses its structure, which is comprised of 32 1-bit adders/subtractors, which are custom-made from a regular full adder to have bit inverting capabilities to perform subtraction and carryout overflow detection for signed operations.

### 

### **1-bit adder/subtractor**

This is a modified full adder that takes in an extra value for bit inversion in cases of subtraction.



#### **Full Adder**

The lowest level component that is used to calculate the result of bit addition with carryout and carryin values.



## **Top Level Processor**

Although not a physical component that can be observed through a block, the top-level processor facilitates the movement of data between all the main components.

# **Methodology/discussion**

In implementing my processor in VHDL, some design choices were made to make the final result more efficient and shorter. A regular register is usually comprised of X amount of flip-flops, depending on the size of the information that is to be stored in the register. My implementation involves creating a single 32-bit register by using a vector and then creating a register array of 32 to create a 32 32-bit register file: (refer to Appendix C for full code)



Another implementation that differs from the classic style is the use of loops to generate large circuits without the need to initiate each separate component manually. This is usually discouraged since loops are not well implemented in actual hardware and can be resource-wasteful and inefficient. This would lead to longer path times and bottleneck the overall speed of the program. (refer to Appendix E for full code)



# **Testing/Waveforms**

To test the program's capabilities, I have created a testbench that will run cases of no overflow and overflow for each operation. (refer to Appendix J for full code)

STIM\_PROC: process

begin

-- initialization

wait for CLK\_PERIOD\*2;

-- Case 1: add $9, $1, $2 (1 + 2 = 3, NO OVERFLOW)

-- 000000 00001 00010 01001 00000 100000

instruction\_input <= "00000000001000100100100000100000";

wait for CLK\_PERIOD\*2;

-- Case 2: add $10, $4, $1 (MAX\_INT + 1, OVERFLOW)

-- 000000 00100 00001 01010 00000 100000

-- This will overflow because 0x7FFFFFFF + 1 = 0x80000000 (positive + positive = negative)

instruction\_input <= "00000000100000010101000000100000";

wait for CLK\_PERIOD\*2;

-- Case 3: addu $11, $1, $2 (1 + 2 = 3, NO OVERFLOW IN UNSIGNED)

-- 000000 00001 00010 01011 00000 100001

instruction\_input <= "00000000001000100101100000100001";

wait for CLK\_PERIOD\*2;

-- Case 4: addu $12, $3, $1 (MAX\_INT + 1, would OVERFLOW in unsigned operations)

-- 000000 00011 00001 01100 00000 100001

-- Note: addu does not detect overflow, result will be 0x00000000

instruction\_input <= "00000000011000010110000000100001";

wait for CLK\_PERIOD\*2;

-- Case 5: sub $13, $8, $6 (10 - 5 = 5, NO OVERFLOW)

-- 000000 00100 00110 01101 00000 100010

instruction\_input <= "00000000100001100110100000100010";

wait for CLK\_PERIOD\*2;

-- Case 6: sub $14, $5, $1 (MIN\_INT - 1, OVERFLOW)

-- 000000 00101 00001 01110 00000 100010

-- This will overflow because 0x80000000 - 1 = 0x7FFFFFFF (negative - positive = positive)

instruction\_input <= "00000000101000010111000000100010";

wait for CLK\_PERIOD\*2;

-- Case 7: subu $15, $8, $6 (10 - 5 = 5, NO OVERFLOW IN UNSIGNED)

-- 000000 00100 00110 01111 00000 100010

-- Note: subu does not detect overflow, but this operation wouldn't overflow anyway

instruction\_input <= "00000000100001100111100000100010";

wait for CLK\_PERIOD\*2;

-- Case 8: subu $16, $7, $1 (MIN\_INT - 1, would OVERFLOW in signed operations)

-- 000000 00111 00001 10000 00000 100011

-- Note: subu does not detect overflow, result will be 0x7FFFFFFF

instruction\_input <= "00000000100001010111000000100011";

wait for CLK\_PERIOD\*2;

wait;

end process;



Referring to the cases laid out in the testbench code on the prior page, we can match them to the waveforms to see that the processor is working as intended. Cases 2 and 6 are the only operations that threw an overflow, which is expected since they are add and sub MIPS instructions. We can conclude that the processor works as intended.

# **Conclusion**

In this project, we successfully designed and implemented a MIPS processor that supports fundamental R-type arithmetic instructions, including add, addu, sub, and subu. The processor was developed with a focus on maintaining MIPS architecture conventions while ensuring accurate instruction execution. Through simulation and testing, we verified the functionality and correctness of the implemented design.

# **Appendices**

## **Appendix A; IR code**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity Instruction\_Register is

Port (

clk : in STD\_LOGIC;

instruction\_in : in STD\_LOGIC\_VECTOR(31 downto 0);

instruction\_out : out STD\_LOGIC\_VECTOR(31 downto 0)

);

end Instruction\_Register;

architecture Behavioral of Instruction\_Register is

begin

-- Store the instruction during execution time

process(clk)

begin

if rising\_edge(clk) then

instruction\_out <= instruction\_in;

end if;

end process;

end Behavioral;

## **Appendix B; Instruction decoder code**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity Instruction\_Decoder is

Port (

instruction : in STD\_LOGIC\_VECTOR(31 downto 0);

opcode : out STD\_LOGIC\_VECTOR(5 downto 0);

rs : out STD\_LOGIC\_VECTOR(4 downto 0);

rt : out STD\_LOGIC\_VECTOR(4 downto 0);

rd : out STD\_LOGIC\_VECTOR(4 downto 0);

shamt : out STD\_LOGIC\_VECTOR(4 downto 0);

funct\_control: out STD\_LOGIC\_VECTOR(1 downto 0); -- Prepare signal to be sent to ALU

r\_type : out STD\_LOGIC -- for r-type confirmation

);

end Instruction\_Decoder;

architecture Behavioral of Instruction\_Decoder is

signal opcode\_internal : STD\_LOGIC\_VECTOR(5 downto 0);

signal funct\_internal : STD\_LOGIC\_VECTOR(5 downto 0);

begin

opcode\_internal <= instruction(31 downto 26); -- should be all 0 for r-tpye

rs <= instruction(25 downto 21);

rt <= instruction(20 downto 16);

rd <= instruction(15 downto 11);

shamt <= instruction(10 downto 6); -- don't think we are using this currently

funct\_internal <= instruction(5 downto 0); -- code for operation

opcode <= opcode\_internal;

process(opcode\_internal, funct\_internal)

begin

-- Default values

r\_type <= '0';

funct\_control <= "00";

-- For R-type instructions (opcode = 000000)

if opcode\_internal = "000000" then

r\_type <= '1';

case funct\_internal is

-- if I do ever add to this, just add more funct codes and increase function\_control if needed

when "100000" => funct\_control <= "00"; -- add

when "100010" => funct\_control <= "01"; -- sub

when "100001" => funct\_control <= "10"; -- addu (same ALU operation as add)

when "100011" => funct\_control <= "11"; -- subu (same ALU operation as sub)

when others => funct\_control <= "00"; -- Default to add

end case;

end if;

end process;

end Behavioral;

## **Appendix C; Register File code**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity Register\_File is

Port (

clk : in STD\_LOGIC;

enable : in STD\_LOGIC;

Ra : in STD\_LOGIC\_VECTOR(4 downto 0); -- Address to be read and sent (corresponding to busA)

Rb : in STD\_LOGIC\_VECTOR(4 downto 0); -- Address to be read and sent (corres. to bus B)

Rw : in STD\_LOGIC\_VECTOR(4 downto 0); -- Address to write new value or value from ALU

busW : in STD\_LOGIC\_VECTOR(31 downto 0); -- Data to be written, sent from ALU (result)

busA : out STD\_LOGIC\_VECTOR(31 downto 0); -- Data to be sent to ALU

busB : out STD\_LOGIC\_VECTOR(31 downto 0) -- Data to be sent to ALU

);

end Register\_File;

architecture Behavioral of Register\_File is

type register\_array is array(0 to 31) of STD\_LOGIC\_VECTOR(31 downto 0);

signal registers : register\_array := (

1 => x"00000001", -- $1 = 1 (small positive value)

2 => x"00000002", -- $2 = 2 (small positive value)

3 => x"FFFFFFFF", -- $3 = -1 (in 2's complement) or max unsigned value

4 => x"7FFFFFFF", -- $4 = 2,147,483,647 (MAX\_INT - largest positive 32-bit signed int)

5 => x"80000000", -- $5 = -2,147,483,648 (MIN\_INT - smallest negative 32-bit signed int)

6 => x"00000005", -- $6 = 5 (small positive value)

7 => x"00000000", -- $7 = 0 (smallest unsinged value)

8 => x"0000000A", -- $8 = 10 (small positive value)

others => (others => '0')

);

begin

process(clk)

begin

if rising\_edge(clk) then

if enable = '1' then

registers(to\_integer(unsigned(Rw))) <= busW;

end if;

end if;

end process;

busA <= registers(to\_integer(unsigned(Ra)));

busB <= registers(to\_integer(unsigned(Rb)));

end architecture Behavioral;

## **Appendix D; Register package code**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

package Zhang\_David\_registers\_pkg is

component Instruction\_Register is

Port (

clk : in STD\_LOGIC;

instruction\_in : in STD\_LOGIC\_VECTOR(31 downto 0);

instruction\_out : out STD\_LOGIC\_VECTOR(31 downto 0)

);

end component;

component Instruction\_Decoder is

Port (

instruction : in STD\_LOGIC\_VECTOR(31 downto 0);

opcode : out STD\_LOGIC\_VECTOR(5 downto 0);

rs : out STD\_LOGIC\_VECTOR(4 downto 0);

rt : out STD\_LOGIC\_VECTOR(4 downto 0);

rd : out STD\_LOGIC\_VECTOR(4 downto 0);

shamt : out STD\_LOGIC\_VECTOR(4 downto 0);

funct\_control: out STD\_LOGIC\_VECTOR(1 downto 0);

r\_type : out STD\_LOGIC

);

end component;

component Register\_File is

Port (

clk : in STD\_LOGIC;

Ra : in STD\_LOGIC\_VECTOR(4 downto 0);

Rb : in STD\_LOGIC\_VECTOR(4 downto 0);

Rw : in STD\_LOGIC\_VECTOR(4 downto 0);

busW : in STD\_LOGIC\_VECTOR(31 downto 0);

enable : in STD\_LOGIC;

busA : out STD\_LOGIC\_VECTOR(31 downto 0);

busB : out STD\_LOGIC\_VECTOR(31 downto 0)

);

end component;

end package;

## **Appendix E; Adder/subtractor code**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity ALU is

Port (

A : in STD\_LOGIC\_VECTOR(31 downto 0); -- First operand (busA)

B : in STD\_LOGIC\_VECTOR(31 downto 0); -- Second operand (busB)

funct\_control : in STD\_LOGIC\_VECTOR(1 downto 0); --

Result : out STD\_LOGIC\_VECTOR(31 downto 0);

Overflow : out STD\_LOGIC;

operand1: out STD\_LOGIC\_VECTOR(31 downto 0);

operand2: out STD\_LOGIC\_VECTOR(31 downto 0)

);

end ALU;

architecture Structural of ALU is

-- Component declaration

component OneBitALU

Port (

A : in STD\_LOGIC;

B : in STD\_LOGIC;

Cin : in STD\_LOGIC;

BInvert : in STD\_LOGIC;

Operation : in STD\_LOGIC;

Sum : out STD\_LOGIC;

Cout : out STD\_LOGIC

);

end component;

-- Internal signals

signal carry : STD\_LOGIC\_VECTOR(32 downto 0); -- 33 bits for carry chain

signal result\_internal : STD\_LOGIC\_VECTOR(31 downto 0);

signal b\_invert : STD\_LOGIC; -- 0 for add, 1 for subtract

signal operation\_type : STD\_LOGIC; -- 00: signed, 01: unsigned

signal carry\_operand1 : STD\_LOGIC\_VECTOR(31 downto 0);

signal carry\_operand2 : STD\_LOGIC\_VECTOR(31 downto 0);

begin

with funct\_control select

b\_invert <= '1' when "01", -- SUB (signed sub)

'1' when "11", -- SUBU (unsigned sub)

'0' when others; -- since 0 is add in this case

with funct\_control select

operation\_type <= '0' when "00", -- ADD (signed)

'0' when "01", -- SUB (signed)

'1' when others; -- For overflow control

-- needed for two-s complement since invert and then add 1

carry(0) <= b\_invert;

-- Generate 32 one-bit ALUs to create the 32-bit ALU

ALU\_GEN: for i in 0 to 31 generate

BIT\_ALU: OneBitALU

port map (

A => A(i),

B => B(i),

Cin => carry(i),

BInvert => b\_invert,

Operation => operation\_type,

Sum => result\_internal(i),

Cout => carry(i+1)

);

end generate;

carry\_operand1 <= A;

carry\_operand2 <= B;

-- Connect result

Result <= result\_internal;

operand1 <= carry\_operand1;

operand2 <= carry\_operand2;

-- Overflow detection logic

-- For signed operations: overflow occurs when carry-in to MSB != carry-out from MSB

-- For unsigned operations: overflow is simply the final carry-out

Overflow <= (carry(31) xor carry(32)) when operation\_type = '0' else -- Signed

'0'; -- unsigned does not throw any overflow

end Structural;

## **Appendix F; One-bit adder/subtractor code**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity OneBitALU is

Port (

A : in STD\_LOGIC; -- First input bit

B : in STD\_LOGIC; -- Second input bit

Cin : in STD\_LOGIC; -- Carry in

BInvert : in STD\_LOGIC; -- Invert B (0: add, 1: subtract)

Operation : in STD\_LOGIC; -- 0: signed, 1: unsigned

Sum : out STD\_LOGIC; -- Sum/Difference output

Cout : out STD\_LOGIC -- Carry out

);

end OneBitALU;

architecture Structural of OneBitALU is

component FullAdder

Port (

A : in STD\_LOGIC;

B : in STD\_LOGIC;

Cin : in STD\_LOGIC;

Sum : out STD\_LOGIC;

Cout : out STD\_LOGIC

);

end component;

signal b\_input : STD\_LOGIC; -- B or not B based on BInvert

begin

-- B input logic - invert for subtraction

b\_input <= B xor BInvert;

FA: FullAdder

port map (

A => A,

B => b\_input,

Cin => Cin,

Sum => Sum,

Cout => Cout

);

end Structural;

## **Appendix G; Full adder code**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity FullAdder is

Port (

A : in STD\_LOGIC; -- First input bit

B : in STD\_LOGIC; -- Second input bit

Cin : in STD\_LOGIC; -- Carry in

Sum : out STD\_LOGIC; -- Sum output

Cout : out STD\_LOGIC -- Carry out

);

end FullAdder;

architecture Behavioral of FullAdder is

begin

-- Full adder logic

Sum <= A xor B xor Cin;

Cout <= (A and B) or (Cin and (A xor B));

end Behavioral;

## **Appendix H; Adder/subtractor package**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

package Zhang\_David\_add\_sub\_ALU\_pkg is

component ALU is

Port (

A : in STD\_LOGIC\_VECTOR(31 downto 0);

B : in STD\_LOGIC\_VECTOR(31 downto 0);

funct\_control : in STD\_LOGIC\_VECTOR(1 downto 0);

Result : out STD\_LOGIC\_VECTOR(31 downto 0);

Overflow : out STD\_LOGIC;

operand1: out STD\_LOGIC\_VECTOR(31 downto 0);

operand2: out STD\_LOGIC\_VECTOR(31 downto 0)

);

end component;

end package;

## **Appendix I; Top level code**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

use work.Zhang\_David\_add\_sub\_ALU\_pkg.all;

use work.Zhang\_David\_registers\_pkg.all;

entity Zhang\_Add\_Sub\_Processor is

Port (

clk : in STD\_LOGIC;

instruction\_input : in STD\_LOGIC\_VECTOR(31 downto 0);

result : out STD\_LOGIC\_VECTOR(31 downto 0);

overflow : out STD\_LOGIC;

operand1 : out STD\_LOGIC\_VECTOR(31 downto 0);

operand2 : out STD\_LOGIC\_VECTOR(31 downto 0)

);

end Zhang\_Add\_Sub\_Processor;

architecture Structural of Zhang\_Add\_Sub\_Processor is

signal instruction : STD\_LOGIC\_VECTOR(31 downto 0);

signal opcode : STD\_LOGIC\_VECTOR(5 downto 0);

signal rs, rt, rd : STD\_LOGIC\_VECTOR(4 downto 0); -- rs is first register, rt is second, rd destination

signal shamt : STD\_LOGIC\_VECTOR(4 downto 0); -- not used

signal busA, busB, busW: STD\_LOGIC\_VECTOR(31 downto 0);

signal funct\_control : STD\_LOGIC\_VECTOR(1 downto 0);

signal r\_type : STD\_LOGIC;

signal overflow\_sig : STD\_LOGIC;

signal operand1\_sig : STD\_LOGIC\_VECTOR(31 downto 0);

signal operand2\_sig : STD\_LOGIC\_VECTOR(31 downto 0);

begin

-- Instruction Register instantiation

INSTR\_REG: Instruction\_Register

port map (

clk => clk,

instruction\_in => instruction\_input,

instruction\_out => instruction

);

-- Instruction Decoder instantiation

DECODER: Instruction\_Decoder

port map (

instruction => instruction,

opcode => opcode,

rs => rs,

rt => rt,

rd => rd,

shamt => shamt,

funct\_control => funct\_control,

r\_type => r\_type

);

-- Register File instantiation

REG\_FILE: Register\_File

port map (

clk => clk,

Ra => rs,

Rb => rt,

Rw => rd,

busW => busW,

enable => r\_type,

busA => busA,

busB => busB

);

-- ALU instantiation

ALU\_UNIT: ALU

port map (

A => busA,

B => busB,

funct\_control => funct\_control,

result => busW,

overflow => overflow\_sig, -- maybe just put overflow here

operand1 => operand1\_sig,

operand2 => operand2\_sig

);

-- Output the result

operand1 <= operand1\_sig;

operand2 <= operand2\_sig;

result <= busW;

overflow <= overflow\_sig;

end Structural;

## **Appendix J; Processor Testbench**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity Add\_Sub\_Processor\_TB is

end Add\_Sub\_Processor\_TB;

architecture Behavioral of Add\_Sub\_Processor\_TB is

component Zhang\_Add\_Sub\_Processor is

Port (

clk : in STD\_LOGIC;

instruction\_input : in STD\_LOGIC\_VECTOR(31 downto 0);

result : out STD\_LOGIC\_VECTOR(31 downto 0);

overflow : out STD\_LOGIC;

operand1 : out STD\_LOGIC\_VECTOR(31 downto 0);

operand2 : out STD\_LOGIC\_VECTOR(31 downto 0)

);

end component;

constant CLK\_PERIOD : time := 10 ps;

signal clk : STD\_LOGIC := '0';

signal instruction\_input : STD\_LOGIC\_VECTOR(31 downto 0) := (others => '0');

signal result : STD\_LOGIC\_VECTOR(31 downto 0);

signal overflow : STD\_LOGIC;

signal operand1 : STD\_LOGIC\_VECTOR(31 downto 0);

signal operand2 : STD\_LOGIC\_VECTOR(31 downto 0);

-- For visual, I forget the codes

constant FUNCT\_ADD : STD\_LOGIC\_VECTOR(5 downto 0) := "100000"; -- 0x20

constant FUNCT\_ADDU : STD\_LOGIC\_VECTOR(5 downto 0) := "100001"; -- 0x21

constant FUNCT\_SUB : STD\_LOGIC\_VECTOR(5 downto 0) := "100010"; -- 0x22

constant FUNCT\_SUBU : STD\_LOGIC\_VECTOR(5 downto 0) := "100011"; -- 0x23

begin

UUT: Zhang\_Add\_Sub\_Processor port map (

clk => clk,

instruction\_input => instruction\_input,

result => result,

overflow => overflow,

operand1 => operand1,

operand2 => operand2

);

CLK\_CYCLE: process

begin

clk <= '0';

wait for CLK\_PERIOD/2;

clk <= '1';

wait for CLK\_PERIOD/2;

end process;

STIM\_PROC: process

begin

-- initialization

wait for CLK\_PERIOD\*2;

-- Case 1: add $9, $1, $2 (1 + 2 = 3, NO OVERFLOW)

-- 000000 00001 00010 01001 00000 100000

instruction\_input <= "00000000001000100100100000100000";

wait for CLK\_PERIOD\*2;

-- Case 2: add $10, $4, $1 (MAX\_INT + 1, OVERFLOW)

-- 000000 00100 00001 01010 00000 100000

-- This will overflow because 0x7FFFFFFF + 1 = 0x80000000 (positive + positive = negative)

instruction\_input <= "00000000100000010101000000100000";

wait for CLK\_PERIOD\*2;

-- Case 3: addu $11, $1, $2 (1 + 2 = 3, NO OVERFLOW IN UNSIGNED)

-- 000000 00001 00010 01011 00000 100001

instruction\_input <= "00000000001000100101100000100001";

wait for CLK\_PERIOD\*2;

-- Case 4: addu $12, $3, $1 (MAX\_INT + 1, would OVERFLOW in unsigned operations)

-- 000000 00011 00001 01100 00000 100001

-- Note: addu does not detect overflow, result will be 0x00000000

instruction\_input <= "00000000011000010110000000100001";

wait for CLK\_PERIOD\*2;

-- Case 5: sub $13, $8, $6 (10 - 5 = 5, NO OVERFLOW)

-- 000000 00100 00110 01101 00000 100010

instruction\_input <= "00000000100001100110100000100010";

wait for CLK\_PERIOD\*2;

-- Case 6: sub $14, $5, $1 (MIN\_INT - 1, OVERFLOW)

-- 000000 00101 00001 01110 00000 100010

-- This will overflow because 0x80000000 - 1 = 0x7FFFFFFF (negative - positive = positive)

instruction\_input <= "00000000101000010111000000100010";

wait for CLK\_PERIOD\*2;

-- Case 7: subu $15, $8, $6 (10 - 5 = 5, NO OVERFLOW IN UNSIGNED)

-- 000000 00100 00110 01111 00000 100010

-- Note: subu does not detect overflow, but this operation wouldn't overflow anyway

instruction\_input <= "00000000100001100111100000100010";

wait for CLK\_PERIOD\*2;

-- Case 8: subu $16, $7, $1 (MIN\_INT - 1, would OVERFLOW in signed operations)

-- 000000 00111 00001 10000 00000 100011

-- Note: subu does not detect overflow, result will be 0x7FFFFFFF

instruction\_input <= "00000000100001010111000000100011";

wait for CLK\_PERIOD\*2;

wait;

end process;

end Behavioral;