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

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

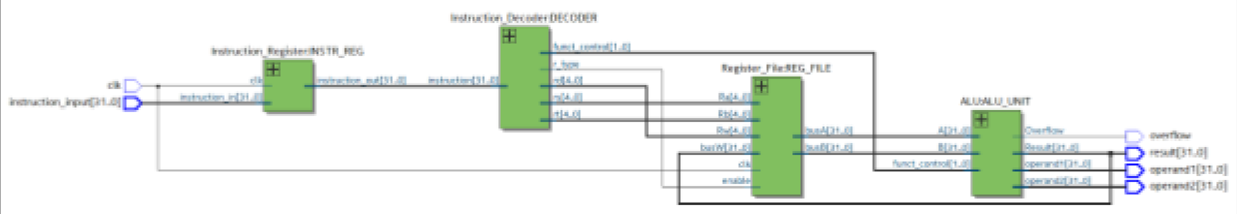


Fig.1: RTL view of add/sub processor

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.

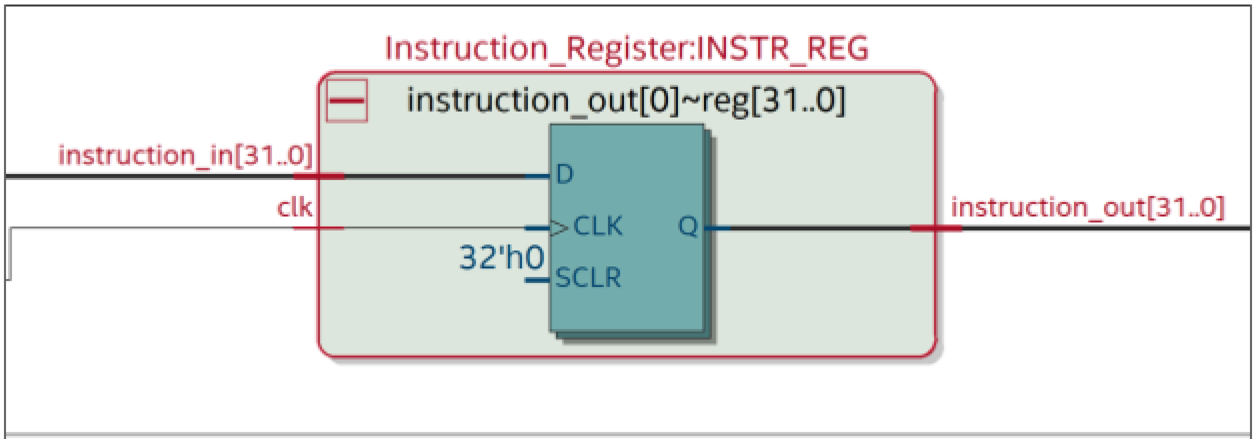


Fig 2. Enhanced view of the instruction register from fig.1

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.

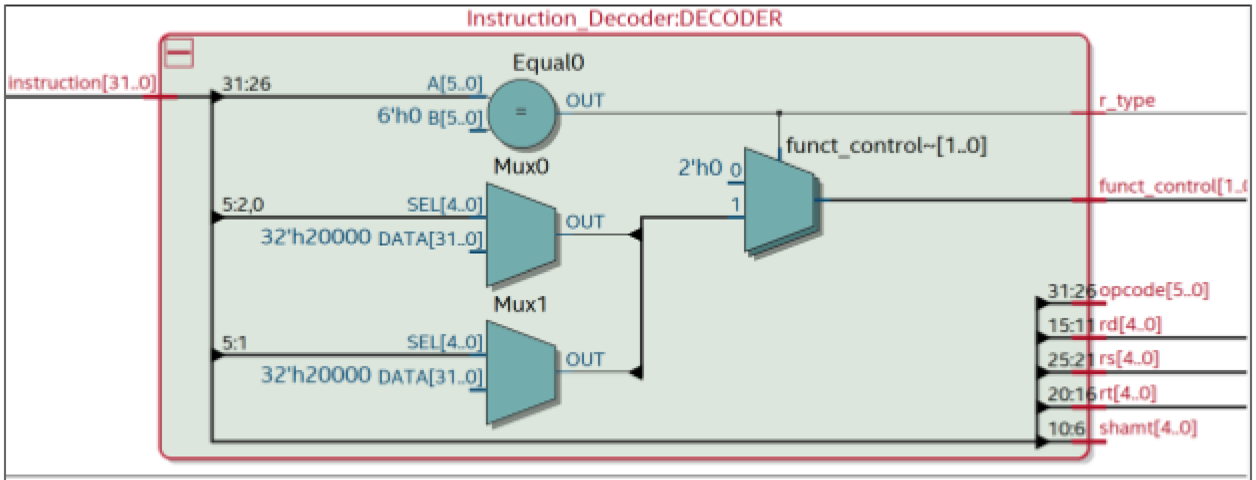


Fig 3. Enhanced view of the instruction decoder from fig.1

32 32-Bit Register Files

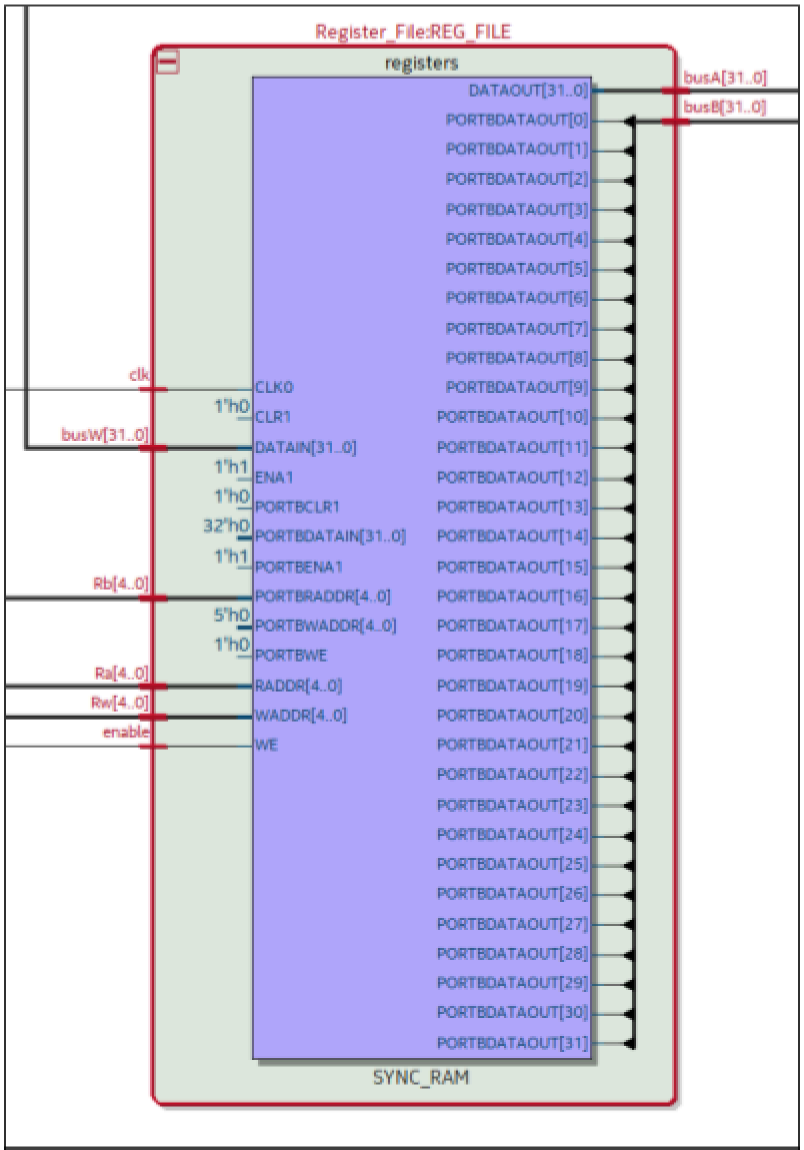


Fig 4. Enhanced view of 32 32-bit register file from fig1.

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

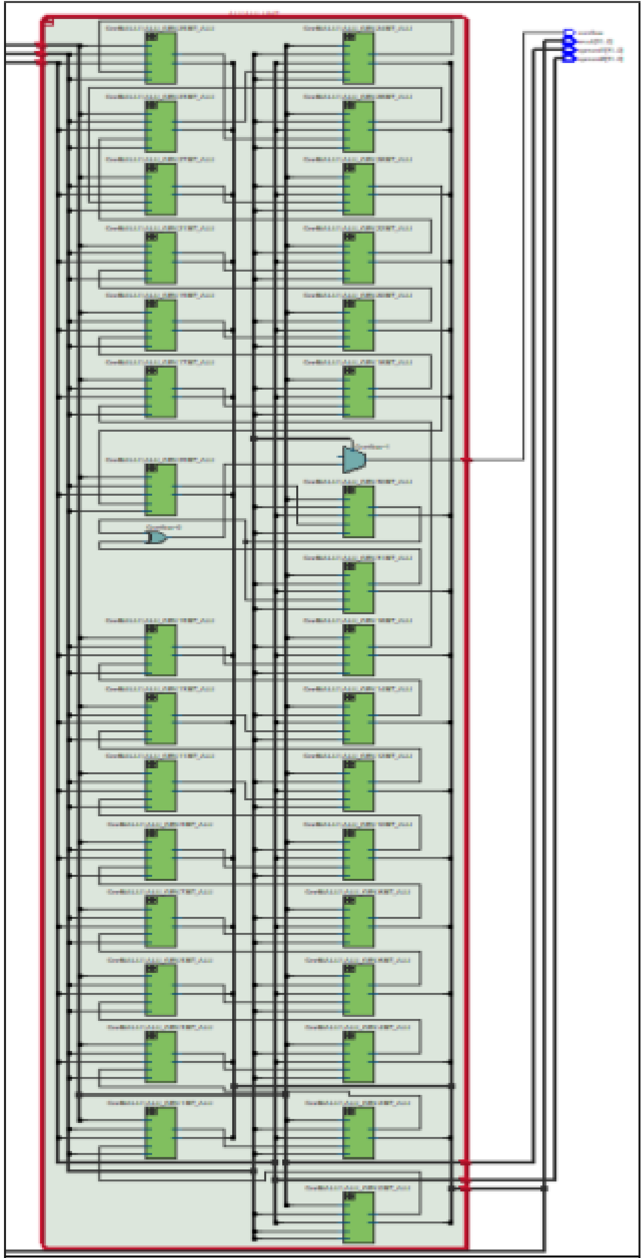


Fig 5. Expanded view of adder/subtractor unit

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.

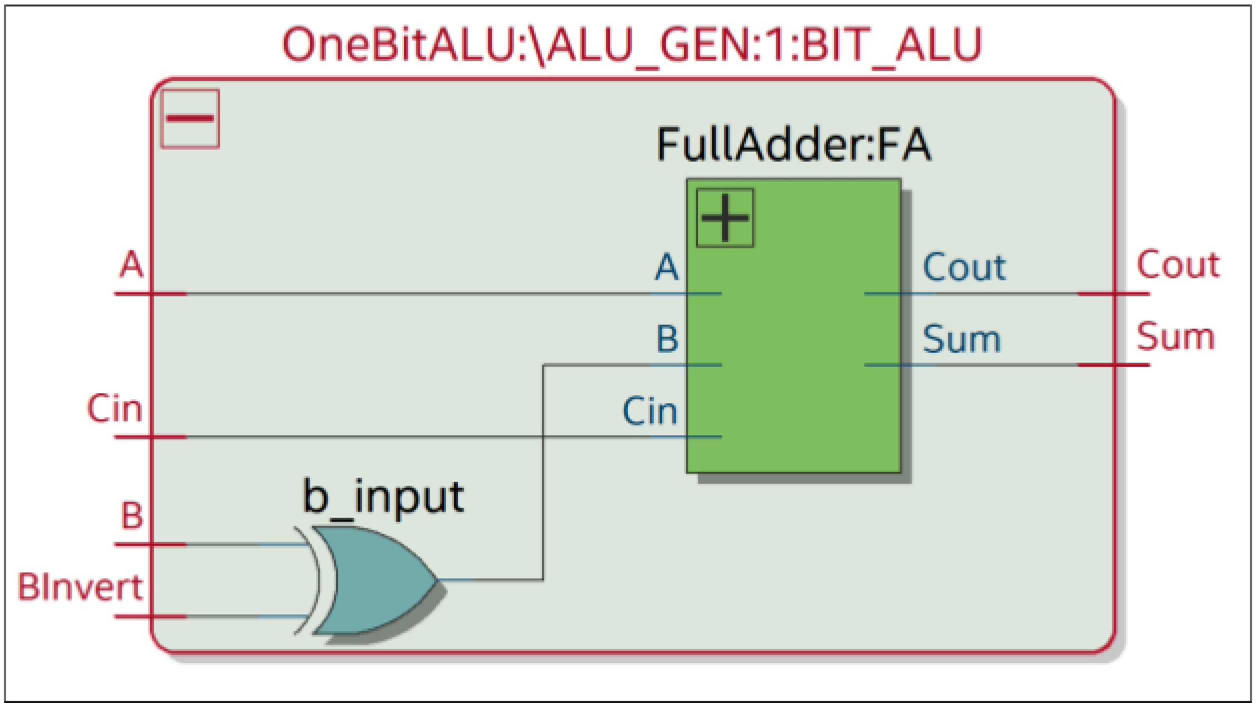


Fig 6. Expanded view of 1-bit adder/subtractor

Full Adder

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

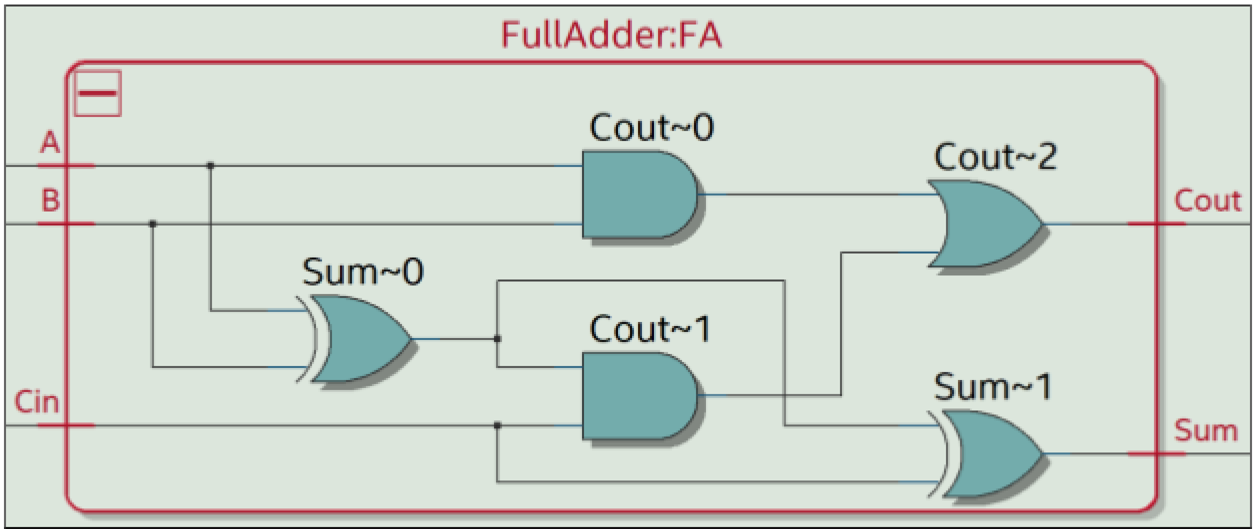


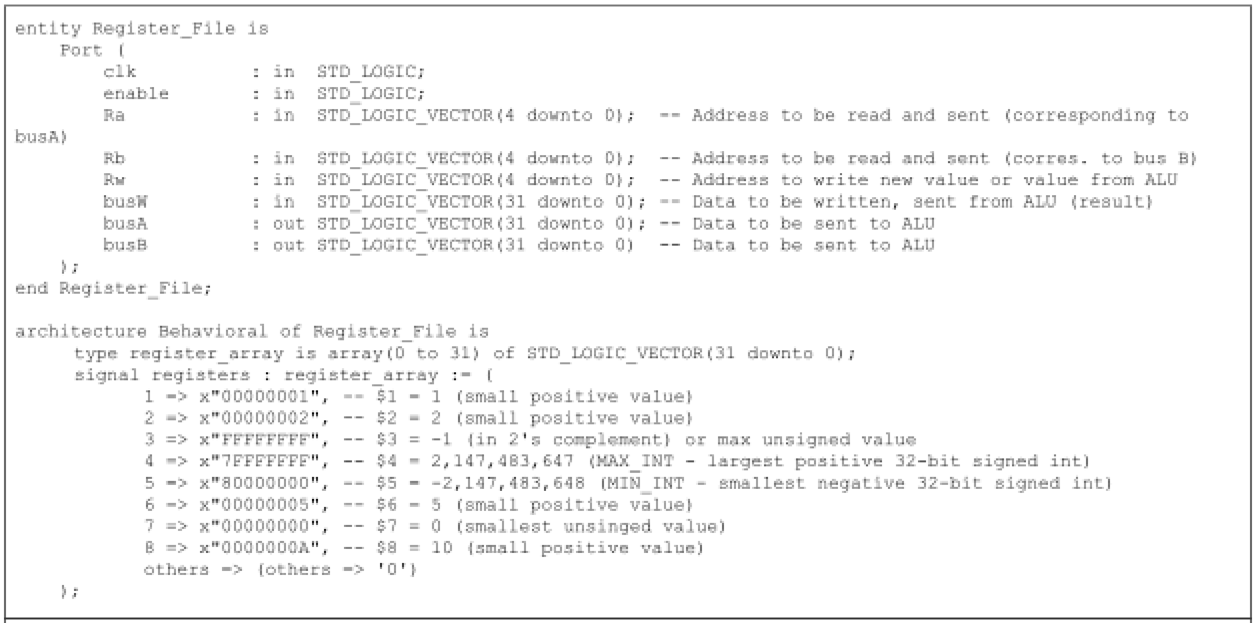
Fig 7. Expanded view of Full adder inside of a 1-bit adder/subtractor

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

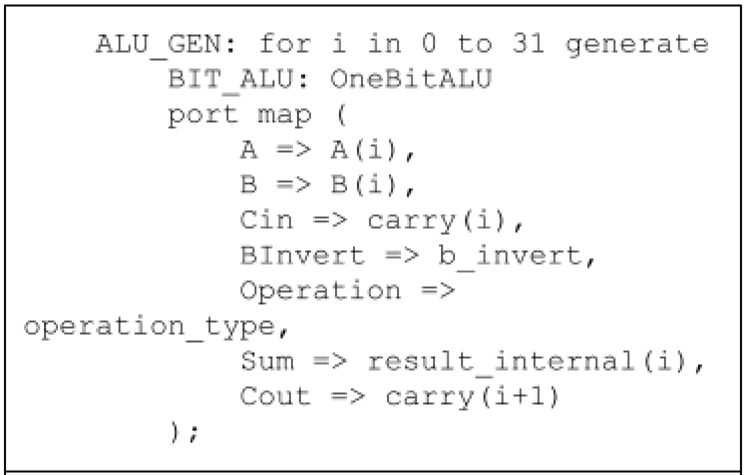
In implementing my processor in VHDL, some design choices were made to make the final result more efficient and shorter. A regular register is usually comprised of X amount of flip-flops, depending on the size of the information that is to be stored in the register. My implementation involves creating 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)



```
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')
    );
```

Fig 8. VHDL Register file implementation

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)



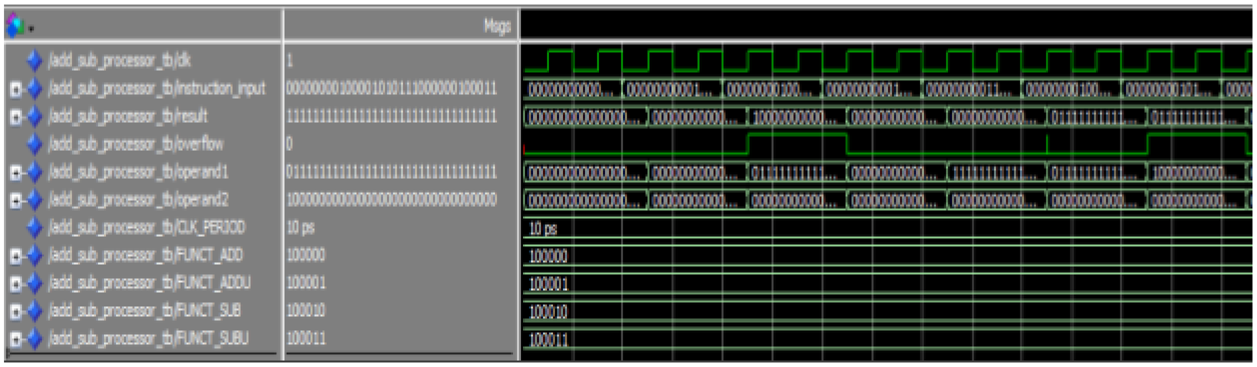
```
    ALU_GEN: for i in 0 to 31 generate
        BIT_ALU: OneBitALU
        port map (
            A => A(i),
            B => B(i),
            Cin => carry(i),
            BInvert => b_invert,
            Operation =>
operation_type,
            Sum => result_internal(i),
            Cout => carry(i+1)
        );
```

Fig 9. Loop used to instantiate 32-bit adder/subtractor

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;
```



```
                                               Msgs
/add_sub_processor_tb/clk                      1
/add_sub_processor_tb/instruction_input        00000000100001010111000000100011
/add_sub_processor_tb/result                   11111111111111111111111111111111
/add_sub_processor_tb/overflow                 0
/add_sub_processor_tb/operand1                 01111111111111111111111111111111
/add_sub_processor_tb/operand2                 10000000000000000000000000000000
/add_sub_processor_tb/CLK_PERIOD               10 ps
/add_sub_processor_tb/FUNCT_ADD                100000
/add_sub_processor_tb/FUNCT_ADDU               100001
/add_sub_processor_tb/FUNCT_SUB                100010
/add_sub_processor_tb/FUNCT_SUBU               100011
```

Fig 10. Waveform output for processor on Modelsim

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;
```