Very SIMPLE ADD/SUB Processor

Demo Simulation Report

Shein Htike

CSC 342, Spring 2025

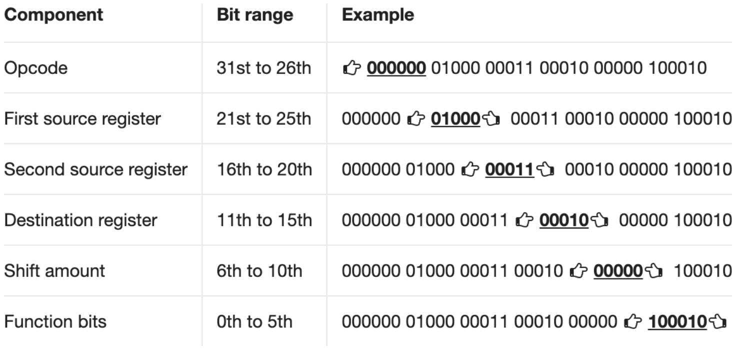
March 12th, 2025

Project Overview

- **Project Title:** Very SIMPLE ADD/SUB Processor
- **Course:** CSC 342, Spring 2025
- **Instructor:** Professor Izidor Gertner
- **Goal:** Designing and simulating a simple processor which can execute 32-bit MIPS R-type add/subtract instructions.
- **Components Used:**
  - 3 Port Register File (register_file_3port.vhd)
  - Full Adder (full_adder.vhd)
  - N-bit Adder (adder_nbit.vhd)
  - Adder/Subtracter (adder_subtracter.vhd)
  - Add/Sub Processor (add_sub_processor.vhd)

Instruction Format

The instruction format is summarized in this image below.



Component	Bit range	Example
Opcode	31st to 26th	☞ **000000** 01000 00011 00010 00000 100010
First source register	21st to 25th	000000 ☞ **01000**☜ 00011 00010 00000 100010
Second source register	16th to 20th	000000 01000 ☞ **00011**☜ 00010 00000 100010
Destination register	11th to 15th	000000 01000 00011 ☞ **00010**☜ 00000 100010
Shift amount	6th to 10th	000000 01000 00011 00010 ☞ **00000**☜ 100010
Function bits	0th to 5th	000000 01000 00011 00010 00000 ☞ **100010**☜

Figure: Instruction Format Diagram

Instruction Format Cont.

In our case, we had four opcodes to implement.

- **add (100000):** Adds two operands.
- **sub (100010):** Subtracts the second operand from the first.
- **addu (100001):** Performs unsigned addition, ignoring overflow.
- **subu (100011):** Performs unsigned subtraction, ignoring overflow.

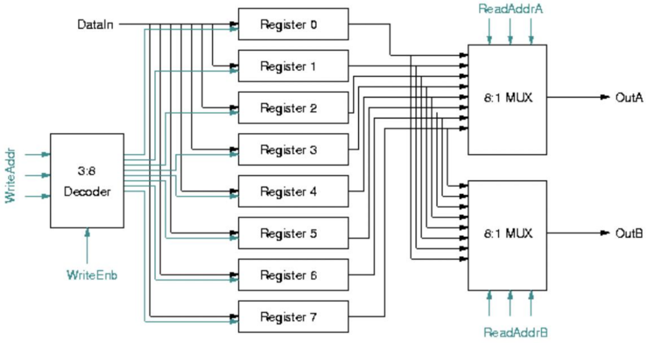
Observations:

- The fifth bit determines if we are adding (0) or subtracting (1).
- The sixth bit determines if we are detecting overflow (0) or ignoring it (1).

This will be useful when we implement these instructions.

Component 1: 3 Port Register File

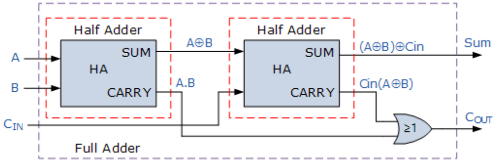
The code for the three-port register was taken from a laboratory assignment. It is essentially a flip-flop array with one decoder and two multiplexers. The design we were given had 8 registers, but I modified it to have 32 registers instead.



Component 2: Full Adder

The code for the full adder was given to us in the laboratory class.

```
ENTITY full_adder IS
    PORT (
        A : IN STD_LOGIC;
        B : IN STD_LOGIC;
        CarryIn : IN STD_LOGIC;
        Sum : OUT STD_LOGIC;
        CarryOut : OUT STD_LOGIC);
END full_adder;
ARCHITECTURE dataflow OF full_adder IS
BEGIN
    Sum <= A XOR B XOR CarryIn;
    CarryOut <= (A AND B) OR ((A XOR B) AND CarryIn);
END dataflow;
```



Component 3: N-Bit Adder

In order to create an N-bit adder, I had to create an array of N full adders.
For each adder, a signal was needed to contain the carry out signal.
This carry out signal is connected to the carry in of the next adder in the line.
The carry out signal of the final adder is connected to the N-bit adder's carry out.

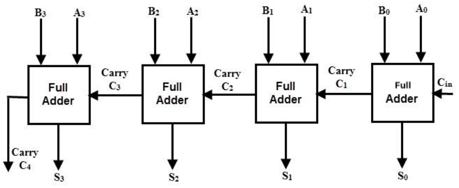


Figure: Example of a 4 Bit Adder

Component 4: Adder Subtracter

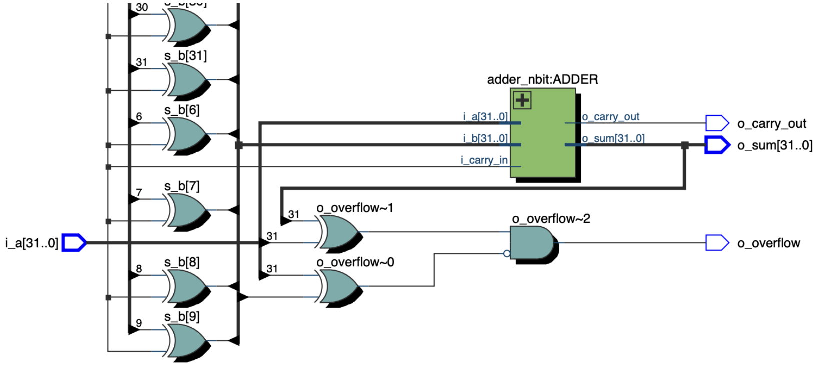


Figure: Adder Subtracter Component

Component 4: Adder Subtracter

Subtraction Logic

We learned in class that an n-bit adder can be used as a subtractor if we flip the bits of the second operand and set carry in to 1.
To do this, I took the bits of the second operand and XOR'ed them with the subtraction signal. This causes the bits of the second operand to flip whenever the subtraction signal is 1. I also connected the subtraction signal into the carry-in input of my n-bit adder.

$$\begin{aligned} \texttt{adder\_b} &= \texttt{input\_b} \oplus \texttt{add\_sub}, \\ \texttt{adder\_cin} &= \texttt{add\_sub} \end{aligned}$$

Component 4: Adder Subtracter

Subtraction Logic
For the ADD and SUB instructions, there needs to be overflow detection. I detected overflows by comparing the signs of the inputs to the outputs.

$$\texttt{overflow} = (\texttt{sign\_a} \odot \texttt{sign\_b}) \wedge (\texttt{sign\_a} \oplus \texttt{sign\_output})$$

An overflow occurs when the sign bits of A and B match, and the sign of the output does not match the signs of A and B.

Component 5: Add Sub Processor

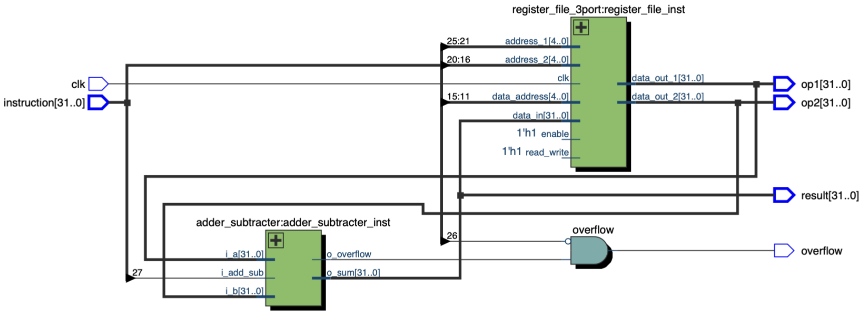
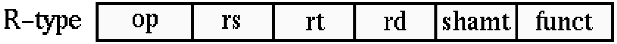


Figure: Add Sub Processor

Component 5: Add Sub Processor

This component divides instruction and connects its bits to our components. *shamt* and *funct* are ignored.



Arithmetic instruction format

- **Opcode**:
  - The 5th bit is connected to `i_add_sub` on the adder subtracter unit.
  - The 6th bit gets XOR'ed with our overflow output from the adder subtracter and the result is connected to the overflow signal of this component.
- **Operands**:
  - `rs`: Connected to address_1 of our register file.
  - `rt`: Connected to address_2 of our register file.
  - `rd`: Connected to data_address of our register file.

Conclusion

This project was very interesting and gave me an understanding of how processors really work.
I realized that the processors powering our devices are really nothing more than collections of logic gates such as adders and flip flops, like the components we designed.

All code for this project will be available at:
https://github.com/SheinH/CSC342-Add-Sub-Processor