Minimal Instruction Set Processor for Basic Arithmetic and Logic

Operations

Team: CompArch Kids

Team members

* Sameeth (23114017)
* Sonika (23114019)
* Raviteja (23114040)
* Koushik (23114063)
* Aasish Varma (23114069)

**Overview of the Project**

This project involves the design and implementation of a 32-bit pipelined processor, capable of performing a set of core operations, including arithmetic, logic, and control flow. Our processor is built with a streamlined, yet versatile instruction set that supports operations on both immediate values and register-stored values. Below is a breakdown of the main features and capabilities of our processor design:

1. **Arithmetic and Logical Operations**: The processor performs essential arithmetic operations (e.g., addition, subtraction, multiplication) and logical operations (e.g., AND, OR). These operations can be executed both on values stored in registers and on immediate constants directly provided in the instruction. The immediate operand feature enables the processor to perform operations on constant values without needing an additional memory fetch, which improves efficiency for certain tasks.
2. **Instruction Set Architecture**: We utilized a compact instruction format with a 3-bit opcode field to specify the operation, a 1-bit immediate flag to indicate the type of operand, and 5-bit register addressing. This structure ensures efficient decoding and execution of instructions while providing sufficient flexibility for future expansion. The instruction format enables the processor to handle a comprehensive set of commands with a reduced instruction width, conserving space and simplifying the hardware.
3. **Memory Interaction**: Loading data into registers for computation and storing results back into registers when needed. These capabilities make the processor versatile enough to support a variety of tasks, from basic arithmetic to more complex program execution.
4. **Further Expandability of the Design**: With the current setup, the processor is modularly designed, allowing for future enhancements such as adding more complex instructions or additional control mechanisms like conditional branch by incorporating flags.

**What have we done for implementing our project**

**Objective**:

* The goal was to design a simple 32-bit pipelined processor in Verilog that can execute fundamental ALU operations, covering basic arithmetic (e.g., addition, subtraction) and logic (e.g., AND, OR) functions.
* This processor would serve as an educational tool to understand processor architecture and core digital logic concepts in a streamlined manner.

**Code Development**:

* We wrote the Verilog code for the processor's core components, focusing on creating a flexible ALU capable of performing essential operations.
* Additionally, a Verilog test bench was developed to simulate how the processor handles inputs and to verify the accuracy of the implemented ALU instructions.
* Each operation was carefully coded to meet the 32-bit requirement, ensuring all inputs and outputs adhere to this word length.

**Simulation and Verification on EDA Playground:**

* To test our Verilog code, we used EDA Playground, an online platform that supports Verilog simulations and provides bit-level output visualization to observe signal behavior.
* Simulations were conducted to verify that each ALU instruction (e.g., ADD, SUB, AND, OR) behaved as expected in response to specific input patterns.
* We reviewed the output values as binary bits, rather than waveforms, to confirm the correctness of each instruction's result.
* This bit-level output verification confirmed that our processor handled data accurately and produced expected results for each ALU operation.

**Hardware Visualization in Quartus Prime Lite**:

* We installed Quartus Prime Lite to synthesize the Verilog code, transforming the logical design into a hardware model.
* In Quartus, we synthesized the processor code, generating a hardware representation of the processor's internal components.
* This synthesis process allowed us to view the internal layout of the ALU, register files, control units, data paths, and the connections between these components.
* By examining this hardware layout, we gained a clearer understanding of the processor's data flow and signal routing, crucial for understanding how a real processor implements similar functions.

**Outcome**:

* By using EDA Playground, we confirmed that the ALU and processor logic were implemented correctly at the code level, ensuring accurate operation for each instruction.
* Quartus Prime Lite provided us with an in-depth view of how the processor’s hardware would be organized, reinforcing our understanding of digital processor architecture and the structure of an ALU within a processor.
* This combined approach of simulation and synthesis offered both functional verification and hardware insight, making it a comprehensive study of processor design.

**Flow Chart of our 32-bit Pipelined Processor**

**Pipelined Processor**

Ripple Carry Adder

Booths Multiplier

Logical Operations

Register File

ALU

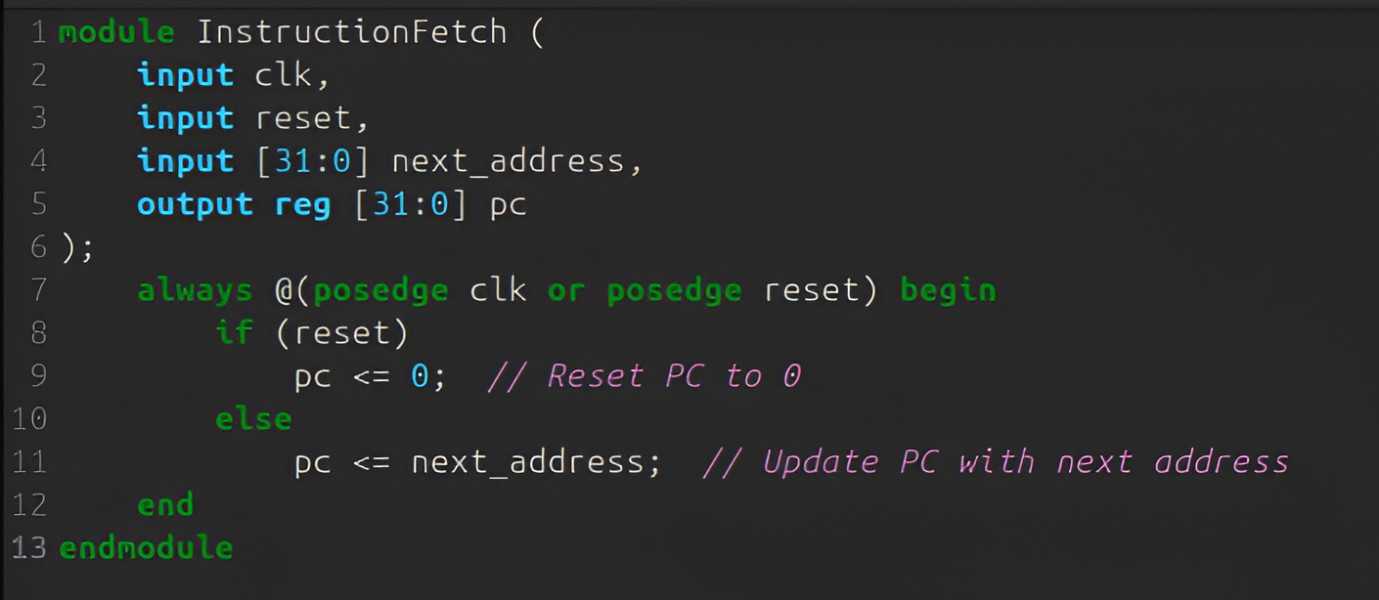
Memory Unit

Instruction fetch Unit

Arithmetic Operations

**Detailed Code Explanation with Hardware Implementation**

1) Instruction Fetch Unit :



For viewing the Hardware Implementation click on the below link:

<https://drive.google.com/file/d/1KRilvjs1yf2XoFvs9E4zKHG45GZ8pqxW/view?usp=drivesdk>

**Code Explanation for Instruction Fetch Unit:**

1. **Inputs and Outputs**:
   * **Inputs**:
     + clk: The clock signal to synchronize the fetching process.
     + reset: A signal to reset the pc\_unit to zero.
     + next\_address: The address of the next instruction to be fetched.
   * **Output**:
     + pc: The 32-bit value which holds the address of the current instruction.
2. **Always Block Logic**:
   * This always block triggers on the positive edge of the clock (clk) or when the reset signal is asserted (reset).
   * **Reset Condition**:
     + If the reset signal (reset) is high, the (pc) is set to 0. This initializes or resets the pc, ensuring the processor starts execution from address 0.
   * **Normal Operation**:
     + When the reset signal is not active, the (pc) is updated with the value of the next\_address. This ensures that the PC always points to the address of the next instruction to be fetched.

2) Memory Unit :

A computer screen shot of a program

Description automatically generated

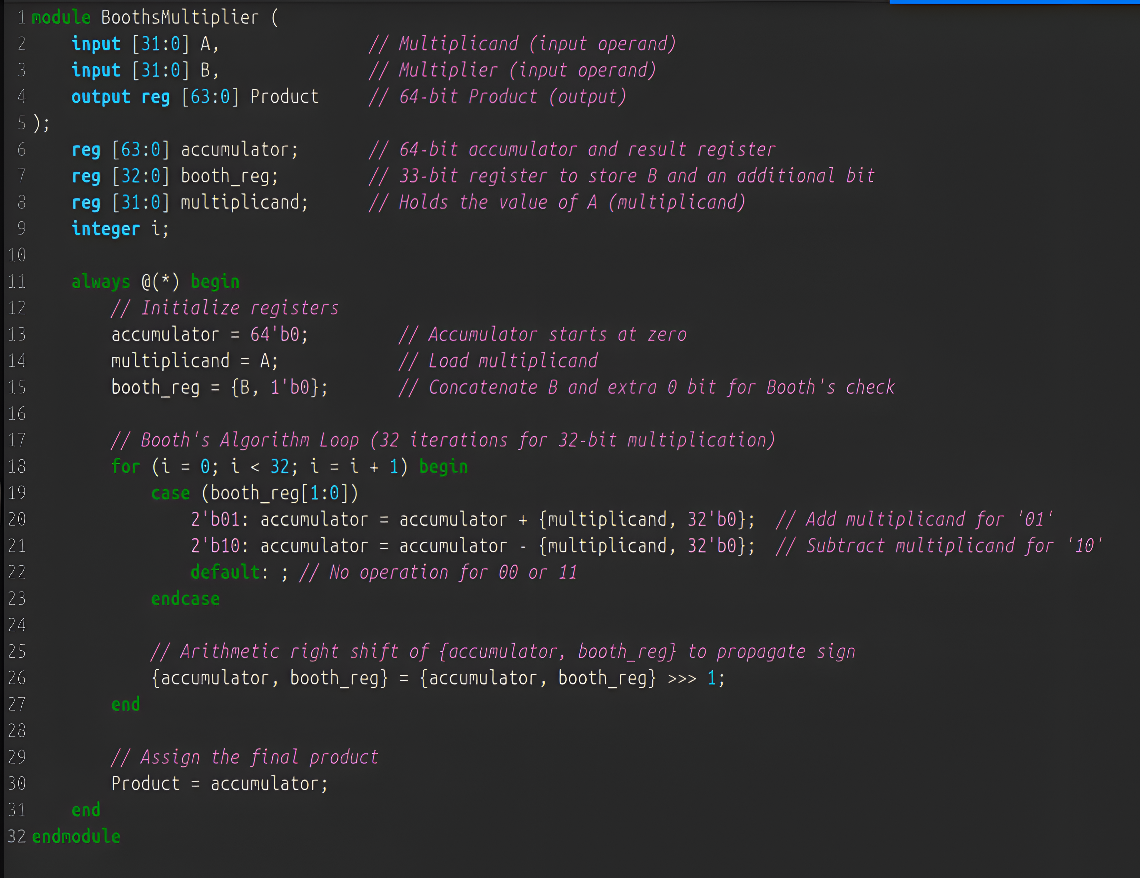
For viewing the Hardware Implementation click on the below link:

<https://drive.google.com/file/d/1KUVmGRvplMS6VeyAivlyEOoRhpJgZaaL/view?usp=drivesdk>

**Code Explanation for Memory Unit:**

1. **Inputs and Outputs**:
   * **Inputs**:
     + pc: The Program Counter (PC) that holds the address of the current instruction to be fetched from memory.
   * **Output**:
     + instruction: The 32-bit instruction fetched from memory corresponding to the address in pc.
2. **Memory Declaration**:
   * This declares a 256-entry memory array where each entry is 32-bits wide. The memory holds the instructions for the processor, where each instruction is 32 bits long.
3. **Initialization Block**:
   * The initial block initializes the memory with predefined instruction values. These instructions are represented as 32-bit values where:
     + The first 3 bits represent the opcode (instruction type).
     + The next bit (after the opcode) is for the condition (whether it's an immediate or register-based operation).
     + The following 5 bits represent the register numbers or immediate values used by the instruction.
     + The last 13-18 bits are for immediate values or unused space in the instruction format.
   * Sample instructions such as ADD, SUB, AND, MOV and other ALU operations are loaded into specific memory locations (e.g., memory[0], memory[1], etc.).
4. **Instruction Fetching**:
   * The always block triggers whenever there is a change in the pc (Program Counter).
   * The instruction corresponding to the current pc is fetched from memory. Since instructions are word-aligned (4 bytes per instruction), the pc is right-shifted by 2 (pc >> 2) to match the correct memory index for the instruction.
5. **Word-Aligned Addressing**:
   * The address (pc >> 2) ensures that memory accesses are word-aligned, as each instruction is 32-bits (4 bytes). Shifting the PC by 2 divides the address by 4, which aligns with the 32-bit word size of the memory

3) ALU:

* Booths Multiplier:
* For viewing the Hardware Implementation click on the below link:

<https://drive.google.com/file/d/1K_2lq9AF5A4RFkihTcSuRDy1tSRvE-GD/view?usp=drivesdk>

**Code Explanation for Booth's Multiplier**

1. **Module Declaration**:
   * module BoothsMultiplier (input [31:0] A, input [31:0] B, output reg [63:0] Product);
   * This module named BoothsMultiplier performs signed multiplication of two 32-bit operands using Booth's algorithm, and it outputs a 64-bit product.
   * Inputs:
     + A: The 32-bit multiplicand.
     + B: The 32-bit multiplier.
   * Output:
     + Product: A 64-bit register to store the final product.
2. **Register Declarations**:
   * reg [63:0] accumulator;
     + A 64-bit register used to accumulate intermediate products during the multiplication process.
   * reg [32:0] booth\_reg;
     + A 33-bit register that holds the multiplier B along with an extra bit (1'b0) for Booth's algorithm processing.
   * reg [31:0] multiplicand;
     + Holds the value of A for use in the calculations.
   * integer i;
     + Integer used as an index for the for loop.
3. **always Block**:
   * The always block is triggered continuously as it is combinational (@(\*)).
   * **Purpose**: Initializes registers and performs Booth’s multiplication algorithm.
4. **Initialization**:
   * accumulator = 64'b0;
     + The accumulator starts at zero, representing an initial product of zero.
   * multiplicand = A;
     + The multiplicand register is loaded with the value of A.
   * booth\_reg = {B, 1'b0};
     + booth\_reg is initialized by concatenating B with an additional 0 bit at the least significant position, enabling Booth’s algorithm to process the last bit.
5. **Booth's Algorithm Loop**:
   * **Loop Structure**:
     + The loop iterates 32 times, corresponding to the 32-bit length of the operands.
   * **Case Statement**:
     + case (booth\_reg[1:0]): Checks the last two bits of booth\_reg (i.e., booth\_reg[1:0]) to determine the required action.
       - 2'b01: Adds multiplicand shifted left by 32 bits to accumulator if the last two bits are 01, as per Booth's encoding.

accumulator = accumulator + {multiplicand, 32'b0};

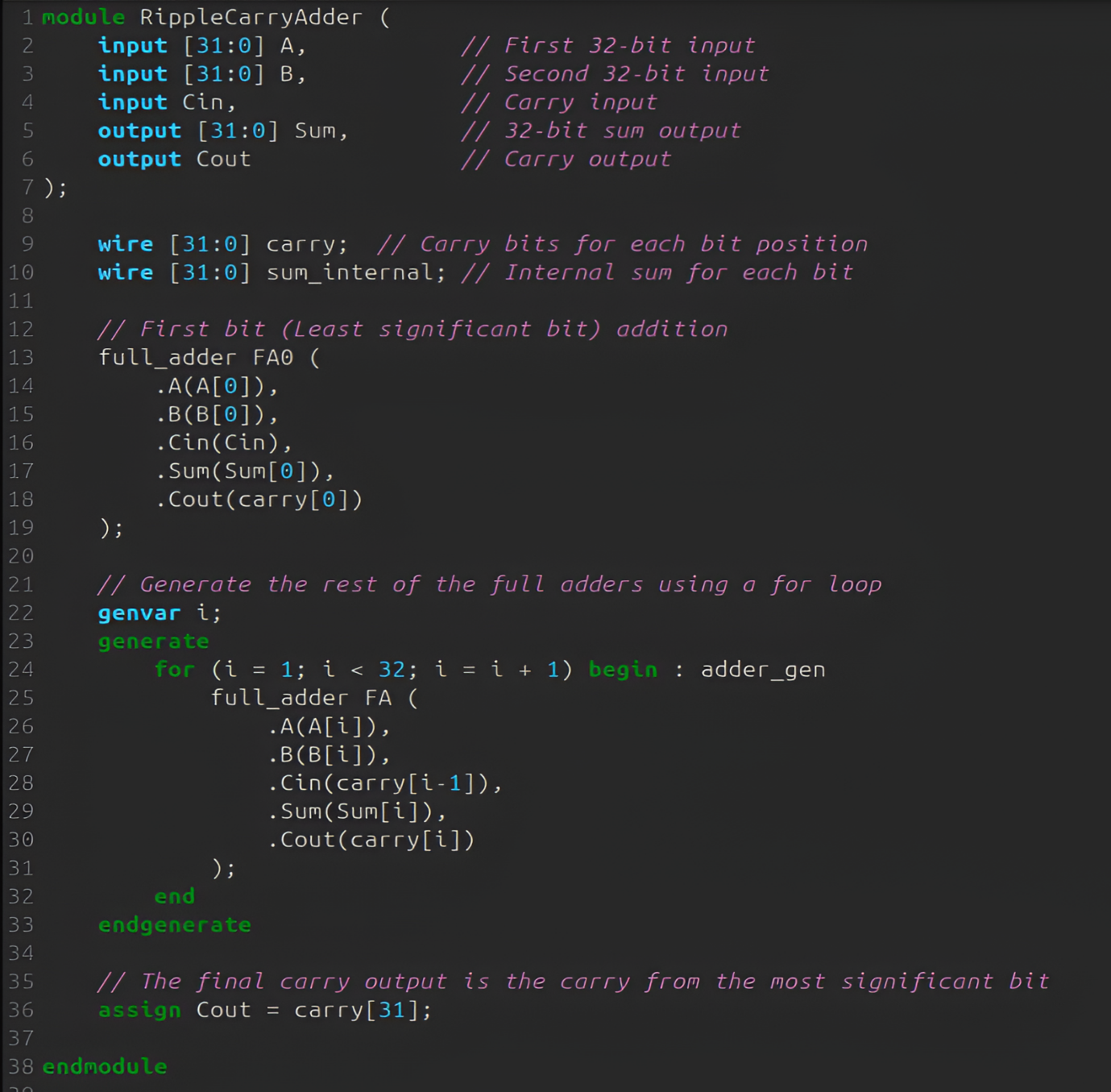
* + - * 2'b10: Subtracts multiplicand shifted left by 32 bits from accumulator if the last two bits are 10.

accumulator = accumulator - {multiplicand, 32'b0};

* + - * default: For 00 or 11, no operation is performed, as no addition or subtraction is needed in these cases.

1. **Arithmetic Right Shift**:
   * {accumulator, booth\_reg} = {accumulator, booth\_reg} >>> 1;
   * This statement performs an arithmetic right shift on the combined {accumulator, booth\_reg} to propagate the sign bit.
   * **Purpose**: The right shift simulates shifting the accumulator and booth\_reg down by one bit, preparing for the next bit in Booth’s algorithm.
2. **Assigning the Final Product**:
   * Product = accumulator;
   * After all iterations of the loop, the accumulated result in accumulator is assigned to Product, which represents the final 64-bit product of the multiplication.

* Ripple Carry Adder:



For viewing the Hardware Implementation click on the below link:

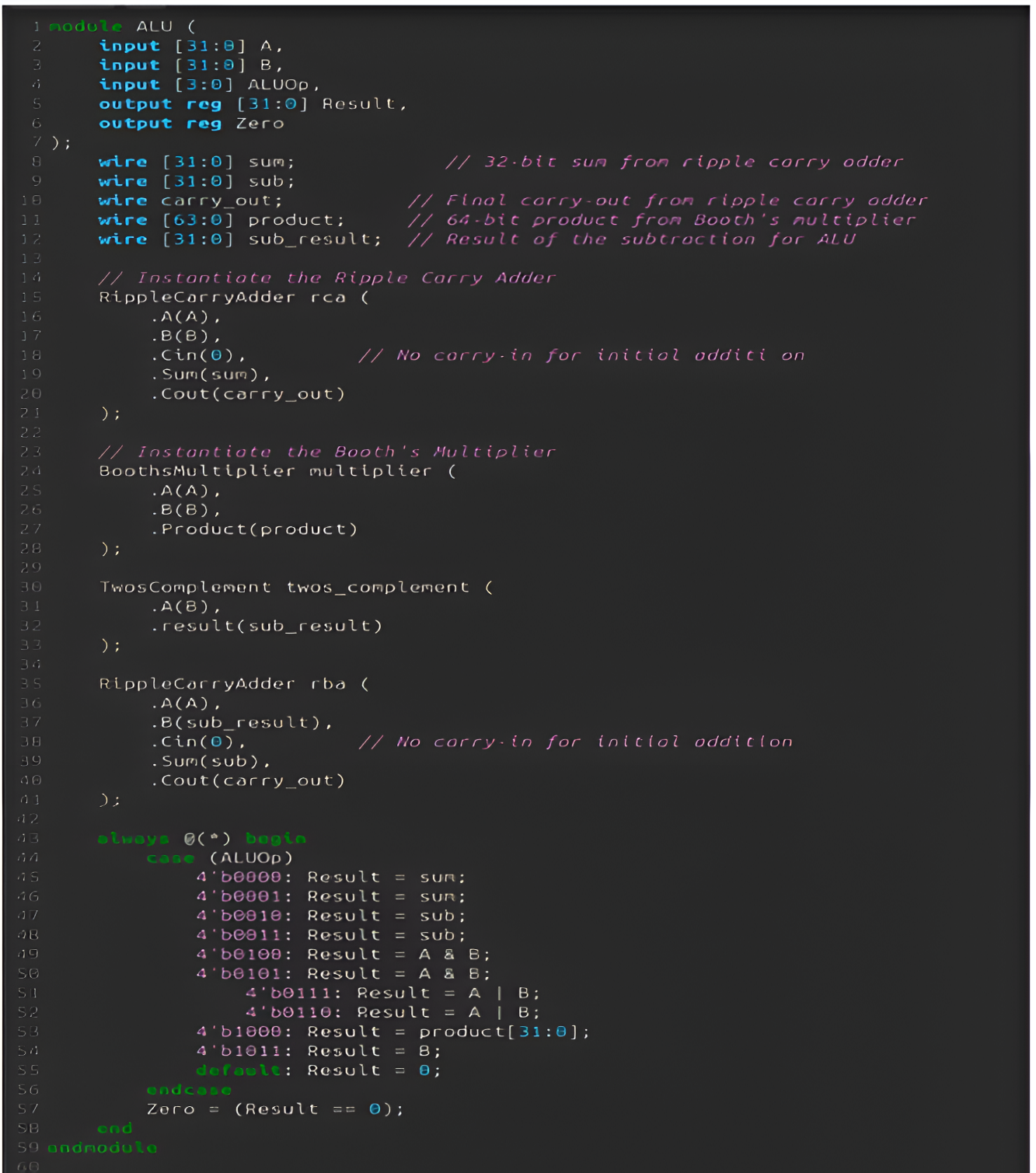
<https://drive.google.com/file/d/1KQHde0ycJbM-HIwc_9-qFISRZvkCUYo1/view?usp=drivesdk>

**Code Explanation for Ripple-Carry Adder:**

1. **Module Declaration**:
   * **Inputs**:
     + A and B: 32-bit input operands.
     + Cin: The carry-in input for the least significant bit (LSB).
   * **Outputs**:
     + Sum: 32-bit result of adding A and B along with the carry-in.
     + Cout: Carry-out, which is the carry from the most significant bit (MSB).
2. **Wire Declarations**:
   * carry: Holds the carry bits for each bit position (one bit per position from 0 to 31).
   * sum\_internal: Stores the internal sum for each bit, though it’s not directly used in the code (likely a leftover for future enhancements).
3. **First Bit (Least Significant Bit) Addition**:
   * The least significant bit (bit 0) of the operands A and B are added together with the initial carry Cin using the full\_adder module.
   * The sum is stored in Sum[0] and the carry-out is stored in carry[0].
4. **Generate the Rest of the Full Adders**:
   * **genvar i**: A loop variable used to generate multiple instances of the full\_adder module for each bit of the operands.
   * **generate block**: This block generates 31 additional full adders (from bit 1 to bit 31).
     + For each bit i, the corresponding inputs A[i], B[i], and carry[i-1] (carry from the previous bit) are fed into the full adder.
     + The sum for the current bit is assigned to Sum[i], and the carry-out is assigned to carry[i].
5. **Final Carry Output**:
   * The final carry-out, Cout, is assigned the value of carry[31], which is the carry generated by the most significant bit (MSB).

**Full Adder Explanation**

1. **Module Declaration**:
   * **Inputs**:
     + A and B: Single-bit inputs to the full adder.
     + Cin: Carry input (from the previous less significant bit).
   * **Outputs**:
     + Sum: The sum of A, B, and Cin.
     + Cout: Carry output (which will be sent to the next more significant bit).
2. **Sum Calculation**:
   * The Sum is computed as the XOR of A, B, and Cin. XOR is used because it adds the bits and generates a sum without considering the carry directly.
     + A ^ B: Adds the two bits and produces a sum without carry.
     + XOR with Cin: Adds the carry bit and produces the final sum.
3. **Carry Calculation**:
   * The Cout (carry-out) is determined by:
     + (A & B): If both A and B are 1, there will be a carry-out.
     + (Cin & (A ^ B)): If Cin is 1 and the sum of A and B is 1 (but not both), there will be a carry-out.

**Now, we combine both these to implement ALU.**

For viewing the Hardware Implementation click on the below link:  
<https://drive.google.com/file/d/1KTRwyA5nl53HDBJSCO6SwPLd5na3XyKD/view?usp=drivesdk>

**Code Explanation for ALU:**

1. **Inputs and Outputs**:

* **Inputs**:
  + A and B: 32-bit operands to the ALU.
  + ALUOp: A 4-bit control signal that determines which operation the ALU will perform (e.g., addition, subtraction, logical AND, etc.).
* **Outputs**:
  + Result: The 32-bit result of the ALU operation.
  + Zero: A flag that indicates whether the result is zero.

1. **Wire Declarations**:

* These wires are used to hold intermediate results for the ALU operations:
  + sum: The result of adding A and B.
  + sub: The result of subtracting B from A.
  + carry\_out: The carry-out from the ripple carry adder (for use in addition/subtraction).
  + product: The result from the Booth's multiplier (used for multiplication).
  + sub\_result: The two's complement of B, which is used for subtraction.

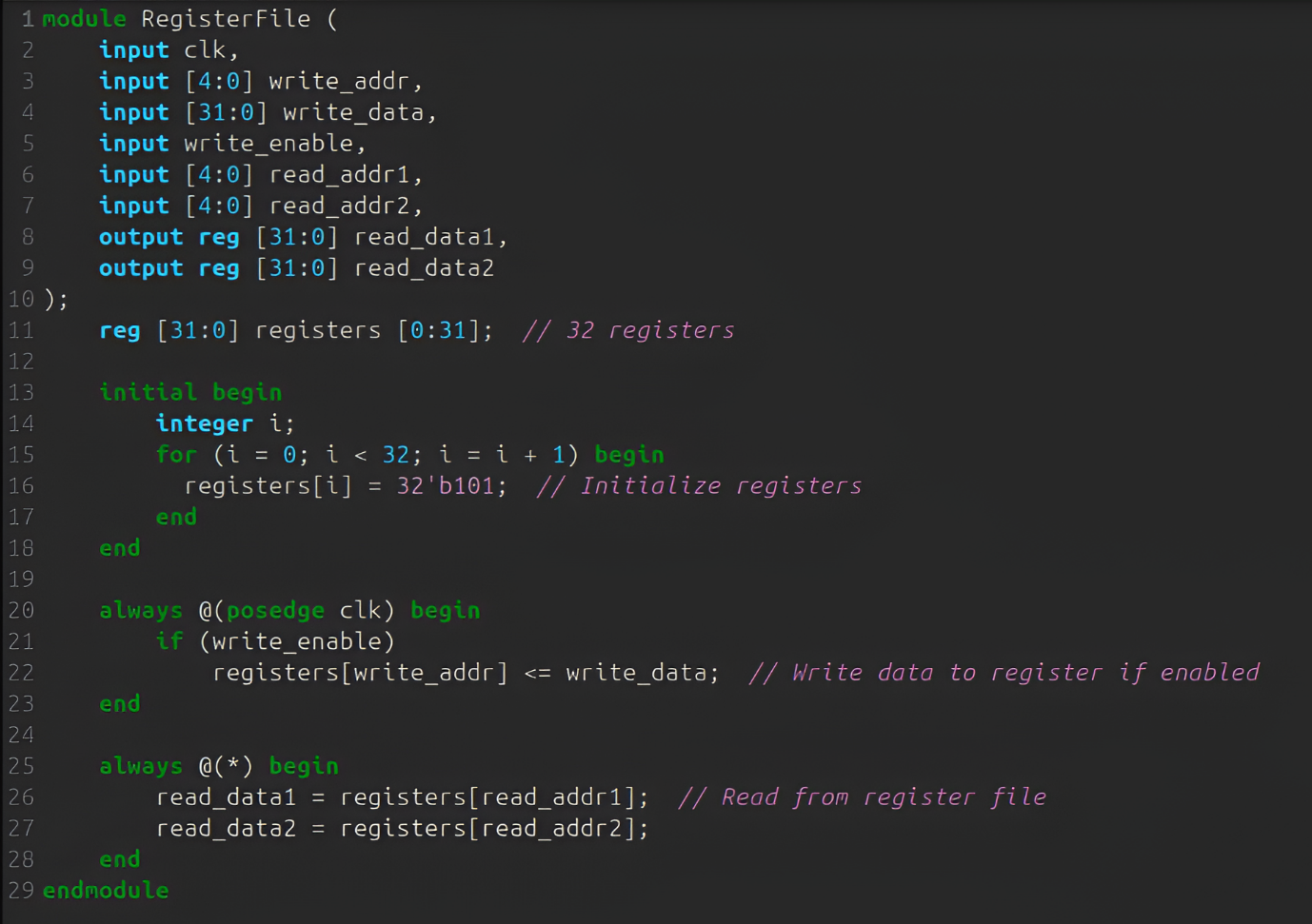
1. **Instantiating Submodules**:

* **Ripple Carry Adder**: This adder computes the sum of A and B when the ALUOp requires an addition. This module explanation is done above in detail.
* **Booth's Multiplier**: This multiplier performs multiplication between A and B. This module explanation is done above in detail.
* **Two's Complement for Subtraction**: This module calculates the two's complement of B, which is used to perform the subtraction (A - B).
* **Second Ripple Carry Adder**: This adder computes the subtraction result by adding A and the two's complement of B (i.e., A + (-B)).

1. **always Block: ALU Operation Selector**:

* **ALU Operations**:
  + Based on the value of ALUOp, the ALU will perform different operations:
    - **Addition**: When ALUOp is 0000 or 0001, the ALU performs addition (the sum of A and B).
    - **Subtraction**: When ALUOp is 0010 or 0011, the ALU performs subtraction (A - B).
    - **AND**: When ALUOp is 0100 or 0101, the ALU performs a bitwise AND on A and B.
    - **OR**: When ALUOp is 0110 or 0111, the ALU performs a bitwise OR on A and B.
    - **Multiplication**: When ALUOp is 1000, the ALU performs multiplication and returns the lower 32 bits of the 64-bit product.
    - **Move**: When ALUOp is 1011, the ALU moves the value of B to the result.
    - **Default**: For any other ALUOp value, the result is set to zero.
* **Zero Flag**: The Zero flag is set to 1 if the result is zero, indicating that the operation resulted in zero.

4) Register File :



For viewing the Hardware Implementation click on the below link:

<https://drive.google.com/file/d/1KLqkMLCjL4tc5odl0fpRUfwXHIKfK3v5/view?usp=drivesdk>

**Code Explanation for Register File:**

1. **Inputs and Outputs**:
   * **Inputs**:
     + clk: The clock signal, used to trigger updates to the register values.
     + write\_addr: A 5-bit address that specifies which register to write data to.
     + write\_data: A 32-bit data input that will be written into the specified register.
     + write\_enable: A signal that determines whether the write operation is enabled or not.
     + read\_addr1, read\_addr2: The 5-bit addresses for the registers from which data is to be read.
   * **Outputs**:
     + read\_data1, read\_data2: The 32-bit values read from the registers specified by read\_addr1 and read\_addr2.
2. **Register Array Declaration**:
   * A 32-entry register array is declared, where each entry (register) is 32 bits wide. This array is used to hold the values of the 32 registers.
3. **Initialization Block**:
   * The initial block runs once at the start of simulation, initializing all 32 registers to the binary value 101 (which is equivalent to 5 in decimal). This is done through a loop that iterates over all 32 registers.
4. **Write Operation (Triggered by clk)**:
   * The always @(posedge clk) block triggers on the rising edge of the clock (clk).
   * If write\_enable is high, the data (write\_data) is written to the register at the address write\_addr. The <= operator ensures that the write is non-blocking, updating the register on the next clock cycle.
5. **Read Operation (Combinational Logic)**:
   * The always @(\*) block triggers on any change to the inputs. It reads the data from the registers specified by read\_addr1 and read\_addr2 and assigns the values to read\_data1 and read\_data2.
   * This is combinational logic, meaning the register file is always "read" as soon as the addresses change, without waiting for the clock signal.

**Now we combine the above 4 parts to make our Simple Processor:**

Since, Code for the entire processor is very long it becomes incompatible to insert the image of the entire code.

For viewing the Hardware Implementation click on the below link:

<https://drive.google.com/file/d/1Xu2PXIA6YeRf_DbceutSfsrgr72nfeNx/view?usp=drivesdk>

**Code Explanation for Pipelined Processor**

**1. Inputs and Outputs**

* Inputs:
  + clk: The clock signal, used to synchronize operations across pipeline stages.
  + reset: A reset signal to initialize the pipeline and processor state.
* Outputs:
  + result: The output of the ALU after processing the instruction.
  + instruction: The current instruction being processed, fetched from memory.

**2. Pipeline Registers**

* Pipeline registers are used to hold data between stages of the pipeline:
  + IF\_ID\_instruction, IF\_ID\_pc: Hold the instruction and program counter (PC) from the instruction fetch (IF) stage.
  + ID\_EX\_reg1, ID\_EX\_reg2, ID\_EX\_immediate, ID\_EX\_pc: Hold register data, immediate value, and PC from the instruction decode (ID) stage.
  + EX\_MEM\_alu\_result, EX\_MEM\_reg2: Hold the ALU result and a register value from the execution (EX) stage.
  + MEM\_WB\_result: Holds the data to be written back to the register file in the write-back (WB) stage.

**3. Instruction Fetch (IF) Stage**

* Purpose: Fetch instructions from memory based on the current program counter (PC).
* Key Components:
  + The pc\_reg register holds the current PC value.
  + next\_pc is calculated as pc + 4 for sequential execution or reset to 0 when reset is active.
  + The Memory module fetches the instruction from memory at the given PC address.
* Pipeline Registers Updated:
  + IF\_ID\_instruction: The fetched instruction.
  + IF\_ID\_pc: The current PC value.

**4. Instruction Decode (ID) Stage**

* Purpose: Decode the fetched instruction and read data from the register file.
* Key Components:
  + Control Signals: Extracted from the instruction fields, such as:
    - alu\_op: Determines the ALU operation.
    - use\_immediate: Indicates if the instruction uses an immediate value.
    - write\_enable: Enables write-back to the register file.
  + Immediate Value Sign-Extension: Converts the immediate value from 18 bits to 32 bits.
  + Register File: Reads data from registers specified in the instruction fields.
* Pipeline Registers Updated:
  + Data read from the register file (ID\_EX\_reg1, ID\_EX\_reg2) and other instruction components are passed to the next stage.

**5. Execution (EX) Stage**

* Purpose: Perform arithmetic or logical operations using the ALU.
* Key Components:
  + ALU Input Selection: The second operand is chosen based on the control signal use\_immediate. It can be a register value (ID\_EX\_reg2) or an immediate value (ID\_EX\_immediate).
  + ALU Operations: The ALU performs operations based on the alu\_op signal.
  + Result Propagation: The ALU result and other necessary data are stored in pipeline registers for the next stage.
* Pipeline Registers Updated:
  + EX\_MEM\_alu\_result: The ALU operation result.
  + EX\_MEM\_reg2: The second operand, often used for memory operations.

**6. Memory Access (MEM) Stage**

* Purpose: Access memory or pass ALU results to the next stage.
* Key Details:
  + In this design, memory operations are not explicitly implemented, so the ALU result is directly forwarded to the write-back stage.
* Pipeline Registers Updated:
  + MEM\_WB\_result: The value to be written back into the register file.

**7. Write-Back (WB) Stage**

* Purpose: Write the result back to the register file if enabled.
* Key Details:
  + The data (MEM\_WB\_result) is written to the register specified by MEM\_WB\_write\_addr, provided MEM\_WB\_write\_enable is active.
* Output: The final result of the ALU operation is provided as output.

**8. Pipeline Flow**

* Instructions progress through the pipeline stages sequentially:
  1. IF: Fetch instruction from memory.
  2. ID: Decode instruction and read registers.
  3. EX: Perform arithmetic or logical operations.
  4. MEM: Access memory (if needed).
  5. WB: Write results back to registers.

This structure allows multiple instructions to be processed simultaneously, with each instruction at a different stage, improving overall throughput.

**9. Code Snippets with Explanation**

1. Instruction Fetch Stage (IF):
   * The PC is updated on every clock cycle unless reset is active.
   * Fetches the instruction at the current PC address.
2. Instruction Decode Stage (ID):
   * Reads data from registers based on the instruction and forwards it to the next stage.
   * Sign-extends the immediate value to 32 bits.
3. Execution Stage (EX):
   * Performs arithmetic or logical operations based on the alu\_op signal.
   * Selects the second ALU operand based on the control signal.
4. Write-Back Stage (WB):
   * Writes the result back to the register file.
   * Outputs the ALU result from the pipeline.