Programming (assembly) and Modeling (Verilog HDL) RISC-V architecture with ChatGPT assistance

Content

0. Introduction..1
Notation:..1
0.1.1 RISC-V Architecture Overview..1
0.1.2 Key Features of RISC-V Architecture..1
0.1.3 RISC-V ISA Components..2
0.1.4 RISC-V Ecosystem..3
0.1.5 Advantages of RISC-V..3
0.2.1. Fine-Grained Control of Hardware..4
0.2.2. Performance Optimization..4
0.3.3 Small Code Size..4
0.2.4 Embedded Systems and Real-Time Applications..4
0.2.5 Educational Value..4
0.3.1 Arithmetic Instructions..5
0.3.2 Logical Instructions..5
0.3.3 Shift Instructions..5
0.3.4 Comparison Instructions..5
0.3.5 Memory Access Instructions..5
0.3.6 Control Transfer Instructions..6
0.3.7 System Instructions..6
0.3.8 No-Operation Instruction..6
0.3.9 Example Program: Sum of Two Numbers..7
0.3.10 Summary..7
To do:..7
0.3.11 Implementation..7
1. Writing simple programs with core instruction set..8
Explanation:..8
Program Explanation:..9
Testing the Program:..9
Explanation:..17
Using external functions..18
Explanation:..22
2. Modeling with Verilog HDL..24
Key Features of Verilog HDL..24
Example Verilog Code..24
Simple 2-to-1 Multiplexer (MUX):..24

  Testbench Example (for the above MUX)........................25
Applications of Verilog HDL........................25
Advantages of Verilog HDL........................25
Step 1: Install Required Tools........................27
  2. Install GTKWave:........................27
Step 2: Create a Verilog Testbench........................27
  Example: Simple Verilog Module and Testbench........................27
Step 3: Compile and Simulate with Icarus Verilog........................28
Summary........................28
ALU Model in Verilog HDL........................29
Explanation........................29
Example Simulation........................30
Explanation of the Testbench........................30
Conclusion........................31
Explanation of the Model........................32
Testbench for the RAM Model........................32
Explanation of the Testbench........................33
Expected Output........................33
Summary :........................34
Steps to Create a Verilog Model:........................35
Basic RISC-V I Instruction Set........................35
  1. ALU Module........................35
2. Instruction Memory (ROM)........................35
3. Register File........................36
4. Control Unit........................36
5. CPU Module........................37
  6. Testbench........................38
Simulation........................38
Summary........................38
Key Modules:........................39
1. DataPath module........................39
2. ControlPath module........................40
3. RegisterFile module........................41
4. ALU module........................41
5. ImmGen module........................41
6. Instruction Memory module........................41
7. Data Memory module........................42
8. Top level - Processor module........................42
Summary:........................42

Programming (assembly) and Modeling (Verilog HDL) RISC-V architecture with ChatGPT assistance

0. Introduction

In these laboratories we start the work by the presentation of our **RISC-V software/hardware platform**.

The software includes Linux (Ubuntu) OS , **C/assembly compiler-assembler** and necessary modeling – simulation tolls based on **Verilog HDL**.

All presentation except the graphic part is assisted by **ChatGPT** that generates the introductory text and the initial assembly and Verilog codes.

These codes are **often with errors.** They are be **corrected** and **modified and finally adapted** to our tools and boards to provide final results.

We communicate with **ChatGPT** via **request messages** concerning the general, programming and modeling aspects.

Each section starts with a ***Message*** sent to **CharGPT**.

Then we **correct modify** and **adapt the assembly language** and **Verilog HDL codes** to run them on our board. The board is based on **SpacemiT** RISC-V (RV64) SoC; The SoC (K1) implements **G** (**IAMFD**), and **C,V,B** ISA extensions.

Notation:

The messages (prompts) sent to ChatGTP are in ***italics bold.***

Below is an example message followed by the **ChatGPT answer.** Read carefully this introduction**.**

0.1 Message :

What is RISC-V architecture ?

0.1.1 RISC-V Architecture Overview

RISC-V is an open, royalty-free **instruction set architecture (ISA)** based on the **reduced instruction set computer (RISC)** principles. It is designed to be simple, modular, and extensible, making it suitable for a wide range of applications, from embedded systems to high-performance computing. RISC-V is unique because it is open-source, allowing anyone to implement and modify it without licensing fees, unlike proprietary ISAs such as ARM or x86.

0.1.2 Key Features of RISC-V Architecture

1. **Open and Royalty-Free**
    - RISC-V is developed under an open standard by the **RISC-V International** organization, meaning that it can be freely used, extended, and implemented by anyone without paying licensing fees or royalties.
2. **Modular and Extensible**
    - RISC-V’s architecture is highly modular, consisting of a **base ISA** and multiple optional **extensions**. The base ISA provides fundamental instructions, while extensions add functionality like floating-point arithmetic, atomic operations, vector processing, and more.

  - Custom extensions can be created for specific use cases, allowing organizations to optimize RISC-V for specialized hardware or applications.

3. **Simplicity and Efficiency (RISC Principles)**
    - The architecture is based on the RISC design philosophy, emphasizing simplicity, with a small, highly optimized set of instructions.
    - A fixed-length, easy-to-decode instruction format simplifies hardware design, leading to more power-efficient and higher-performance processors.
4. **Scalability**
    - RISC-V can scale across a wide range of computing needs, from small microcontrollers (MCUs) to high-end servers and supercomputers.
    - It has variants for 32-bit (RV32), 64-bit (RV64), and even 128-bit (RV128) address spaces, which allow RISC-V to cover different performance and memory capacity requirements.
5. **Multiple Operating Modes**
    - RISC-V supports multiple privilege levels and operating modes:
        - **User mode (U-mode)**: For executing user applications.
        - **Supervisor mode (S-mode)**: For running operating system kernels.
        - **Machine mode (M-mode)**: The highest privilege level, used for managing hardware resources and bootstrapping the system.
        - These modes provide security and isolation, similar to other modern architectures.

0.1.3 RISC-V ISA Components

1. **Base ISA**
    - The base ISA defines the core set of instructions that must be implemented on every RISC-V processor. It comes in two primary forms:
        - **RV32I**: A 32-bit integer instruction set.
        - **RV64I**: A 64-bit integer instruction set.
    - These base instructions include operations for arithmetic, memory access (loads/stores), control flow (jumps/branches), and more.
    - Example instructions in the base ISA include:
        - `ADD`: Add two registers.
        - `SUB`: Subtract two registers.
        - `LW`: Load a word from memory.
        - `SW`: Store a word to memory.
        - `BEQ`: Branch if two registers are equal.
2. **Optional Extensions**
    - **M (Multiplication and Division)**: Adds support for integer multiplication and division operations.
    - **A (Atomic)**: Provides atomic instructions for multi-threaded synchronization.
    - **F (Single-Precision Floating Point)**: Adds support for 32-bit floating-point arithmetic.
    - **D (Double-Precision Floating Point)**: Adds support for 64-bit floating-point arithmetic.
    - **C (Compressed)**: A 16-bit compressed instruction set that reduces code size, useful for memory-constrained environments.
    - **V (Vector)**: Adds support for vector operations, enabling efficient parallel data processing, especially useful in applications like machine learning.
    - **B (Bit Manipulation)**: Provides instructions for manipulating individual bits in registers, useful in cryptographic algorithms and digital signal processing.
3. **Privilege Levels**
    - **User Mode (U)**: Lowest privilege level, where applications run.

- **Supervisor Mode (S)**: Operating system level for managing resources and security.
- **Machine Mode (M)**: The most privileged mode, handling tasks like initialization and hardware control.

0.1.4 RISC-V Ecosystem

1. **Toolchain and Software Support**
    - RISC-V is supported by a growing ecosystem of development tools, including:
        - **GNU GCC Toolchain**: Open-source compiler support for building RISC-V programs.
        - **LLVM/Clang**: Another widely-used compiler infrastructure with RISC-V support.
        - **QEMU**: Open-source processor emulator supporting RISC-V.
        - **Operating Systems**: Linux, FreeBSD, and other open-source operating systems are ported to RISC-V, providing a robust platform for development.
2. **Hardware Implementations**
    - **SiFive** and other companies have developed commercial RISC-V processors and development boards.
    - The architecture is widely used in embedded systems, microcontrollers, and custom SoC (System on Chip) designs.
3. **Custom Extensions**
    - One of the unique aspects of RISC-V is the ability to define custom instruction extensions, allowing users to implement hardware features specific to their application domains, such as cryptography, AI, or signal processing.

0.1.5 Advantages of RISC-V

1. **Open and Flexible**
    - RISC-V's open-source nature and modular design make it ideal for academic research, startups, and companies seeking a customizable ISA.
2. **Scalability and Versatility**
    - It can scale from low-power embedded devices to high-performance computing, covering a broad range of industries and applications.
3. **Simpler Design**
    - The simplicity of the RISC-V instruction set leads to easier hardware implementation, reduced power consumption, and lower manufacturing costs compared to more complex ISAs like x86.
4. **Customizability**
    - With the ability to create custom extensions, RISC-V allows specialized processing requirements to be met without compromising compatibility with the base architecture.

A we have mentioned we start the study of RISC-V architectures with **assembly level programming**. We are going to ask what are the assembly level programming ChatGPT with the following message.

0.2 Message :

What are the advantages to program RISC-V with assembly language ?

Programming RISC-V with assembly language offers several key advantages, especially in scenarios where control, optimization, and hardware awareness are critical. Below are some of the primary benefits:

0.2.1. Fine-Grained Control of Hardware

- **Direct Access to CPU Features**: RISC-V assembly allows developers to directly interact with processor instructions, registers, memory, and I/O devices. This level of control is essential for hardware-level tasks such as interrupt handling, device drivers, or manipulating specific hardware peripherals.
- **Custom Instruction Set Extensions**: RISC-V allows for user-defined custom instructions, so writing in assembly helps exploit these extensions effectively when needed for specialized tasks.

0.2.2. Performance Optimization

- **Manual Optimization**: Assembly language gives developers the ability to optimize their code for speed, size, or power efficiency by manually tuning instructions, avoiding unnecessary overhead, and making decisions about which operations are faster for a given processor.
- **Instruction-Level Parallelism**: Developers can control how instructions are scheduled, potentially reducing instruction stalls, pipeline hazards, and maximizing the use of the CPU’s pipelines.
- **Efficient Use of Memory**: Assembly allows developers to minimize memory usage, a critical factor for embedded systems or resource-constrained environments like micro-controllers.

0.3.3 Small Code Size

- **Minimal Overhead**: Writing in assembly produces minimal overhead since high-level language constructs like loops, conditionals, and function calls are replaced with direct machine instructions. This is particularly useful in systems with limited memory (e.g., embedded systems).
- **Precise Control of Memory Layout**: In assembly, the programmer has direct control over how data and code are laid out in memory, allowing for optimized and compact memory usage.

0.2.4 Embedded Systems and Real-Time Applications

- **Low-Level Access**: Assembly language is often used in embedded and real-time systems where low-level control is essential, such as controlling specific peripherals, real-time performance tuning, and interrupt handling.
- **Deterministic Execution**: In real-time systems, knowing the exact execution time of instructions is important. Assembly provides a clear understanding of how long each instruction will take, ensuring real-time constraints are met.

0.2.5 Educational Value

- **Understanding Architecture**: Writing in assembly helps **developers gain a deep understanding of the underlying RISC-V architecture**, including how **memory** is accessed, how **instructions are executed**, and how **control flow** is managed.
- **Learning How Compilers Work**: By learning to write in assembly, programmers also develop insights into what compilers do behind the scenes, allowing for better high-level code optimization and debugging.

....

Now when we have the above explanation let us ask about the basic instruction set dealing only with integer data values RV32I or RV64I.

0.3 Message:

Write RISC-V basic assembly instruction set (I)

The RISC-V base integer instruction set, often referred to as the **"I" (Integer)** instruction set, includes a small but complete set of instructions necessary for general-purpose computing. This set is part of the **RV32I** and **RV64I** instruction sets, with RV32I being a 32-bit variant and RV64I being a 64-bit variant. These instructions include basic arithmetic, logical, control, memory access, and system instructions.

Here's an overview of the basic instructions in the RISC-V "I" instruction set, categorized by their purpose.

0.3.1 Arithmetic Instructions

These instructions perform integer arithmetic operations.

- **`add rd, rs1, rs2`** — Add two registers (`rd = rs1 + rs2`).
- **`addi rd, rs1, imm`** — Add immediate (`rd = rs1 + imm`).
- **`sub rd, rs1, rs2`** — Subtract (`rd = rs1 - rs2`).
- **`lui rd, imm`** — Load upper immediate (`rd = imm << 12`).
- **`auipc rd, imm`** — Add upper immediate to PC (`rd = PC + (imm << 12)`).

0.3.2 Logical Instructions

These instructions perform **bitwise logical operations**.

- **`and rd, rs1, rs2`** — Bitwise AND (`rd = rs1 & rs2`).
- **`andi rd, rs1, imm`** — Bitwise AND with immediate (`rd = rs1 & imm`).
- **`or rd, rs1, rs2`** — Bitwise OR (`rd = rs1 | rs2`).
- **`ori rd, rs1, imm`** — Bitwise OR with immediate (`rd = rs1 | imm`).
- **`xor rd, rs1, rs2`** — Bitwise XOR (`rd = rs1 ^ rs2`).
- **`xori rd, rs1, imm`** — Bitwise XOR with immediate (`rd = rs1 ^ imm`).

0.3.3 Shift Instructions

These instructions perform **left or right shifts**.

- **`sll rd, rs1, rs2`** — Shift left logical (`rd = rs1 << rs2`).
- **`slli rd, rs1, imm`** — Shift left logical immediate (`rd = rs1 << imm`).
- **`srl rd, rs1, rs2`** — Shift right logical (`rd = rs1 >> rs2`).
- **`srli rd, rs1, imm`** — Shift right logical immediate (`rd = rs1 >> imm`).
- **`sra rd, rs1, rs2`** — Shift right arithmetic (`rd = rs1 >> rs2`).
- **`srai rd, rs1, imm`** — Shift right arithmetic immediate (`rd = rs1 >> imm`).

0.3.4 Comparison Instructions

These **instructions compare values** in registers and set the **destination register to 1** if the comparison is **`true`**, **otherwise set it to 0**.

- **`slt rd, rs1, rs2`** — Set if less than (`rd = (rs1 < rs2)`).
- **`slti rd, rs1, imm`** — Set if less than immediate (`rd = (rs1 < imm)`).
- **`sltu rd, rs1, rs2`** — Set if less than (unsigned) (`rd = (rs1 < rs2)` unsigned).
- **`sltiu rd, rs1, imm`** — Set if less than immediate (unsigned) (`rd = (rs1 < imm)` unsigned).

0.3.5 Memory Access Instructions

These instructions **load** data from **memory into registers** or **store** data from registers into memory.

- **`lw rd, imm(rs1)`** — Load word (`rd = Mem[rs1 + imm]`).
- **`lh rd, imm(rs1)`** — Load halfword.
- **`lb rd, imm(rs1)`** — Load byte.
- **`lbu rd, imm(rs1)`** — Load byte unsigned.

- **`lhu rd, imm(rs1)`** — Load halfword unsigned.
- **`sw rs2, imm(rs1)`** — Store word (`Mem[rs1 + imm] = rs2`).
- **`sh rs2, imm(rs1)`** — Store halfword.
- **`sb rs2, imm(rs1)`** — Store byte.

0.3.6 Control Transfer Instructions

These instructions control the flow of execution, including conditional branches and unconditional jumps.

- **`beq rs1, rs2, offset`** — Branch if equal.
- **`bne rs1, rs2, offset`** — Branch if not equal.
- **`blt rs1, rs2, offset`** — Branch if less than (signed).
- **`bge rs1, rs2, offset`** — Branch if greater than or equal (signed).
- **`bltu rs1, rs2, offset`** — Branch if less than (unsigned).
- **`bgeu rs1, rs2, offset`** — Branch if greater than or equal (unsigned).
- **`jal rd, offset`** — Jump and link (used for function calls).
- **`jalr rd, offset(rs1)`** — Jump and link register.

0.3.7 System Instructions

These instructions provide system-level control, including **traps and environment calls** (for example, for operating system services).

- **`ecall`** — Environment call (used to invoke system services, e.g., syscalls).
- **`ebreak`** — Environment break (used for debugging or breakpoints).

0.3.8 No-Operation Instruction

This instruction does nothing and is often used for padding.

- **`nop`** — No operation (`addi x0, x0, 0` is commonly used as `nop`).

0.3.9 Example Program: Sum of Two Numbers

Here is a simple RISC-V assembly program that adds two numbers and stores the result in a register.

```
    .text
    .globl _start

_start:
    # Load two numbers into registers
    li a0, 10         # Load immediate value 10 into register a0
    li a1, 20         # Load immediate value 20 into register a1
    # Perform addition
    add a2, a0, a1    # a2 = a0 + a1 (10 + 20 = 30)
    # Exit the program using ecall
    li a7, 93         # Syscall number for exit
    ecall             # Make system call
```

This program loads two numbers, `10` and `20`, into the **`a0`** and **`a1`** registers, adds them, and stores the result in **`a2`**. It then uses **`ecall`** to exit.

0.3.10 Summary

The **RISC-V "I" instruction set** is the **core instruction set** for 32-bit and 64-bit RISC-V architectures. It provides a complete set of operations for basic integer arithmetic, logic, control, and memory access. This minimal set allows for building more complex systems while keeping the instruction set simple and efficient.

To do:

Read the above section

Analyze and try to understand the “first” program presented above.

0.3.11 Implementation

Now let us implement this “first” program on our board

```
    .text
    .globl _start

_start:
    # Load two numbers into registers
    li a0, 10         # Load immediate value 10 into register a0
    li a1, 20         # Load immediate value 20 into register a1
    # Perform addition
    add a2, a0, a1    # a2 = a0 + a1 (10 + 20 = 30)
    # Exit the program using ecall
    li a7, 93         # Syscall number for exit
    ecall             # Make system call
```

Modification:

To run this program on our board we have to replace **`_start`** with **`main`**. And save it as **`simple.s`** .

Now we can generate machine code with:

```
bako@k1:~/Programs/labgen$ gcc simple.s –o simple
```

and run it:

```
bako@k1:~/Programs/labgen$ ./simple
```

We have no printed output, the program just exits.

1. Writing simple programs with core instruction set

In this section we are going to program some simple arithmetical functions to show essential **arithmetic** and flow **control instructions**.

Message :

Write RISC-V assembly program to multiply the given value by 10.

Here is a RISC-V assembly program to multiply a given value in register **`x10`** (also called `a0`, which is commonly used for function arguments) by 10. The result will be stored back in **`x10`**.

We will use a combination of shifts and additions since multiplication by 10 can be optimized as: **x×10=(x×8)+(x×2)**

Where:

- x×8 can be achieved with a left shift by 3 bits (`x << 3`)
- x×2 can be achieved with a left shift by 1 bit (`x << 1`)

RISC-V Assembly Code:

```
slli x7, x5, 3 # x7 = x5 << 3 (Multiply x5 by 8)
slli x8, x5, 1 # x8 = x5 << 1 (Multiply x5 by 2)
add x6, x7, x8 # x6 = x7 + x8 (x6 = 8*x5 + 2*x5, i.e., x5 * 10)
```

Explanation:

- **`slli t0,a0,3`**: Shifts the value in `a0` left by 3 positions (i.e., multiplies by 8) and stores the result in **`t0`**.
- **`slli t1,a0,1`**: Shifts the value in `a0` left by 1 position (i.e., multiplies by 2) and stores the result in **`t1`**.
- **`add a0,t0,t1`**: Adds the values in `t0` and `t1` (i.e., **`a0*8+a0*2`**), storing the result back into `a0`.

This efficiently multiplies the input value by 10 using **only shifts and additions**.

Message:

Write RISC-V complete assembly program to multiply the given value by 10.

Here's a complete RISC-V assembly program to multiply a given value by 10. The program assumes the **input value is provided via the `a0`** register (which is typically used for function arguments) and will return the result in the same register.

```
    .data
    # Optional section for declaring constants or other data (not used here)

    .text
    .globl _start         # Entry point for the program

_start:
    # Assume input value is already in a0 (for example, you can load a value manually for testing)
    # a0 = input value (this could be set by a caller function or hardcoded below)

    # Multiply the value in a0 by 10
    slli t0, a0, 3        # t0 = a0 * 8 (shift left a0 by 3 bits)
    slli t1, a0, 1        # t1 = a0 * 2 (shift left a0 by 1 bit)
    add  a0, t0, t1       # a0 = t0 + t1 (a0 = a0 * 8 + a0 * 2)
    # End the program (using the RISC-V exit system call)
    li   a7, 93           # Load the system call number for exit (93)
    ecall                # Make the system call to exit

# The result of multiplying the input value by 10 will be stored in a0 when the program ends.
```

Program Explanation:

1. **Initialization**:
    - The program begins with `_start`, which is the entry point.
    - It assumes that the input value to be multiplied by 10 is already stored in the `a0` register. For testing purposes, you can modify this value in a simulator or a calling program.
2. **Multiplication by 10**:
    - The value in `a0` is multiplied by 8 (`a0<<3`) and stored in `t0`.
    - The value in `a0` is multiplied by 2 (`a0<<1`) and stored in `t1`.
    - The values in `t0` and `t1` are added together and stored back into `a0`, which now holds the result of `a0*10`.
3. **Exit**:
    - A system call (`ecall`) is used to terminate the program.
    - The system call number for exiting a RISC-V program is loaded into `a7` (`exit` is system call `93`).
    - The program then exits, returning the multiplied value in the `a0` register.

Testing the Program:

- You can test this program by providing an initial value in `a0` and observing the result in `a0` after the program runs.

For example, if you load the value **5** into `a0`, after the program executes, `a0` will contain `50` (i.e., `5 * 10`).

Modification:

To run this program on our board we have to replace `_start` with `main`.

Execution:

```
bako@k1:~/Programs/labgen$ gcc simple.s –o  simple
bako@k1:~/Programs/labgen$ ./ simple
```

Remark: no output values are printed on the terminal

Message:

Write assembly program for RISC-V RV64 64-bit to read one variable with `scanf` function provided by the operating system and outputs it with `printf` function.

Here is an example of a RISC-V RV64 assembly program that reads an integer from the user using the **`scanf`** function provided by the operating system and then outputs it using the **`printf`** function. This program assumes you are working with the standard C library (**`libc`**), which provides **`scanf`** and **`printf`**.

```
    .data
    fmt_scanf:  .asciz "%ld"        # Format string for scanf (reading a long integer)
    fmt_printf: .asciz "You entered: %ld\n"  # Format string for printf (printing a long integer)
    number:     .quad 0             # Reserve space for a 64-bit integer
    .text
    .globl _start

_start:
    # Step 1: Read an integer from the user using scanf
    # Load the address of the format string ("%ld") into a0 (first argument for scanf)
    la a0, fmt_scanf           # First argument for scanf (the format string)
    # Load the address of the variable 'number' into a1 (second argument for scanf)
    la a1, number              # Second argument for scanf (address of the variable)
    # Call scanf
    call scanf                 # Use the provided scanf function to read input
    # Step 2: Print the entered integer using printf
    # Load the address of the printf format string ("You entered: %ld\n") into a0
    la a0, fmt_printf          # First argument for printf (format string)
    # Load the value of 'number' into a1 (second argument for printf)
    ld a1, number              # Load the integer value from memory (64-bit integer)
    # Call printf
    call printf                # Use the provided printf function to print output
    # Step 3: Exit the program
    li a7, 93                  # syscall number for exit (in RV64)
    li a0, 0                   # exit code 0 (success)
    ecall                      # Exit the program
```

Explanation:

1. **Data Section (`.data`):**
    - **`fmt_scanf`**: This is the format string for **`scanf`**, specifying that we want to read a 64-bit integer (**`"%ld"`**).
    - **`fmt_printf`**: This is the format string for `printf`, specifying that we want to print a 64-bit integer with a message (**`"You entered: %ld\n"`**).
    - **`number`**: This reserves 8 bytes of memory (64 bits) to store the integer input from the user.
2. **Text Section (`.text`):**
    - **Step 1: `scanf` Call:**
        - Load the format string for **`scanf`** into register **`a0`** (first argument).
        - Load the address of the variable `number` (where the input will be stored) into register **`a1`** (second argument).
        - Call the **`scanf`** function using the **`call scanf`** instruction. This reads a 64-bit integer from the user and stores it in the memory location pointed to by `number`.
    - **Step 2: `printf` Call:**
        - Load the format string for **`printf`** into register **`a0`** (first argument).
        - Load the value of the `number` variable from memory into register **`a1`** (second argument).
        - Call the **`printf`** function using the **`call printf`** instruction to print the message and the integer value.
    - **Step 3: Exit the Program:**
        - After printing the integer, the program exits using the **`exit`** syscall (syscall number **`93`**).

Modification:

To run this program on our board we have to replace `_start` with `main`.

Analysis and Execution:

```
bako@k1:~/Programs/labgen$ gcc scanfprintf.s -o scanfprintf
bako@k1:~/Programs/labgen$ ./scanfprintf
45
Result: 45
bako@k1:~/Programs/labgen$
```

Message :

Write RISC-V complet assembly program for a power function `pow(a,b)` *.*

Here's a complete RISC-V assembly program that calculates the power function **pow(a,b)=a^b**, where the **base a and exponent b** are provided as **arguments**. The result will be stored in the `a0` register.

We'll use a simple iterative method (**Exponentiation by Squaring**) to efficiently compute the power function. This algorithm is useful as it reduces the number of multiplications to approximately **log(b)** instead of the straightforward approach that would take **b** multiplications.

```
    .data
    # No data section needed for this program

    .text
    .globl _start          # Entry point for the program

# Entry point of the program
.section  .text
_start:
    # Load the base 'a' and exponent 'b' (you can change these for testing)
    li a0, 2               # Load base 'a' = 2 into a0
    li a1, 5               # Load exponent 'b' = 5 into a1
    # Call the power function: pow(a0, a1)
    jal ra, power          # Jump and link to the 'power' function
    # End the program (use ecall to exit)
    li   a7, 93            # Load exit system call number (93)
    ecall                  # Make the system call to exit

# Function: power(a, b) -> a^b
# Arguments:
#   a0 = base (a)
#   a1 = exponent (b)
# Result:
#   a0 = result (a^b)
power:
    li t0, 1               # Initialize t0 = 1 (result accumulator)

power_loop:
    beqz a1, end_power     # If exponent b (a1) is 0, exit loop (base case)
    andi t1, a1, 1         # t1 = a1 & 1 (check if b is odd)
    beqz t1, skip_mult     # If b is even, skip multiplication
    mul t0, t0, a0         # If b is odd, result *= base (t0 = t0 * a0)

skip_mult:
    mul a0, a0, a0         # base *= base (a0 = a0 * a0)
    srli a1, a1, 1         # b >>= 1 (right shift exponent by 1)
    j power_loop           # Repeat the loop

end_power:
    mv a0, t0              # Move the result (t0) back to a0
    ret                    # Return to the caller
```

Explanation:

1. **Main Program**:
    - The program starts by initializing two values: `a0` with the **base a** (e.g., 2) and `a1` with the **exponent b** (e.g., 5). You can change these values to test different inputs.
    - It then calls the `power` function using the `jal` (**Jump and Link**) instruction.
2. **Power Function (`power`)**:
    - **Initial Setup**:
        - The result accumulator `t0` is initialized to 1.
    - **Main Loop** (`power_loop`):
        - The program checks if the exponent (`a1`) is zero using `beqz`. If so, it exits the loop, since any number raised to the power of 0 is 1.
        - It then checks if the current exponent is **odd** (`a1&1`) using the `andi` instruction.
        - If the exponent is **odd**, the **current result** (`t0`) is **multiplied** by the **base** (`a0`).
        - The **base is squared** (`a0=a0*a0`), and the **exponent is halved** (`a1>>=1`).
    - **Loop Exit**:
        - Once the exponent reaches zero, the program exits the loop, and the final result is stored in `a0` before returning to the caller.
3. **Exit**:
    - After the `power` function finishes, the program exits using the `ecall` system call with the number `93` to indicate the exit system call in RISC-V.
    -

Analysis, Modifications, Assembly, and Execution:

To run this program on our board we have to replace `_start` with `main`

Add the output message with the calculated value via (before exit call). In `.data` section add:

```
    .data
result_mes:
    .asciz  "Result (a^b): %d\n"


..
    mv  a1,a0
    la  a0,result_mes
    call printf
```

Assembly the program and run it:

```
bako@k1:~/Programs/labgen$ gcc power.s -o power
bako@k1:~/Programs/labgen$ ./power
Result (a^b): 32
```

```
main:
```

Message :

Write RISC-V complete assembly program for a power function `pow(a,b)`. Read the arguments from terminal and print the result in the terminal. Use system `scanf` and `printf` functions.

To implement the power function `pow(a,b)` in RISC-V assembly with reading arguments from the terminal and printing the result using `scanf` and `printf` from the C library, we need to utilize the ABI (Application Binary Interface) conventions for interacting with the C standard library.

Steps:

1. **Read input** using `scanf`.
2. **Compute the power** using an iterative approach or exponentiation by **squaring**.
3. **Print the result** using `printf`.

```
    .data
prompt_a:
    .asciz "Enter base (a): "   # Prompt for base
prompt_b:
    .asciz "Enter exponent (b): " # Prompt for exponent
result_msg:
    .asciz "Result (a^b): %d\n"  # Format string for result
input_format:
    .asciz "%d"                  # Input format for scanf

    .text
    .globl main                   # Define global entry point for C runtime

main:
    # Prompt for base 'a'
    la   a0, prompt_a             # Load address of prompt_a
    call printf                   # Call printf to print the prompt for base
    # Read base 'a'
    la   a0, input_format          # Load address of input format string ("%d")
    la   a1, base                 # Load address of base variable
    call scanf                    # Call scanf to read base into memory
    # Prompt for exponent 'b'
    la   a0, prompt_b             # Load address of prompt_b
    call printf                   # Call printf to print the prompt for exponent
    # Read exponent 'b'
    la   a0, input_format          # Load address of input format string ("%d")
    la   a1, exponent             # Load address of exponent variable
    call scanf                    # Call scanf to read exponent into memory
    # Load base and exponent into registers
    lw   a0, base                 # Load base value from memory into a0
    lw   a1, exponent             # Load exponent value from memory into a1
    # Call the power function pow(a0, a1)
    jal  ra, power                # Jump to power function
    # Print the result
    la   a0, result_msg           # Load address of result message
    mv   a1, a0                   # Move result (in a0) to a1 for printf
    call printf                   # Call printf to print the result
    # Exit the program
    li   a7, 93                   # Load exit syscall number
    ecall                         # Make the syscall to exit

# Function: power(a, b) -> a^b
# Arguments:
#   a0 = base (a)
#   a1 = exponent (b)
# Result:
#   a0 = result (a^b)
power:
    li   t0, 1                    # Initialize t0 = 1 (result accumulator)

power_loop:
    beqz a1, end_power            # If exponent (b) is 0, exit loop (base case)
    andi t1, a1, 1                # t1 = a1 & 1 (check if b is odd)
    beqz t1, skip_mult            # If b is even, skip multiplication
    mul t0, t0, a0                # If b is odd, result *= base (t0 = t0 * a0)
```

```
skip_mult:
    mul a0, a0, a0              # base *= base (a0 = a0 * a0)
    srli a1, a1, 1              # b >>= 1 (right shift exponent by 1)
    j power_loop                # Repeat the loop

end_power:
    mv a0, t0                   # Move result (t0) back to a0
    ret                         # Return to caller

    .bss
    .align 4
base:
    .space 4                    # Reserve space for the base
exponent:
    .space 4                    # Reserve space for the exponent
```

Explanation:

1. **Data Section** (`.data`):
    - Prompts for the base and exponent are stored as strings (`prompt_a`, `prompt_b`).
    - The result message format is stored (`result_msg`), including a format specifier for `printf` to display the result.
    - The format for `scanf` to read integers (`input_format`, set to `"%d"`).
2. **Text Section** (`.text`):
    - **Main program** (`main`):
        - Prompts the user for the base (a) and exponent (b) using `printf`.
        - Reads the input values using `scanf` and stores them in memory.
        - The base is stored in a memory location labeled `base`, and the exponent is stored in `exponent`.
        - The power function is then called (`power`), which calculates ab.
        - The result is printed using `printf` with the `result_msg` format string.
3. **Power Function (`power`)**:
    - Implements the iterative **exponentiation by squaring** algorithm.
    - The result is stored in `t0`, initialized to `1`.
    - If the exponent is odd, the result is multiplied by the base.
    - The base is squared, and the exponent is halved.
    - This continues until the exponent is zero.
4. **BSS Section** (`.bss`):
    - Reserves space in memory for the base and exponent inputs using `.space` directive.

Analysis, Modifications, Assembly, and Execution:

To run this program on our board we have to replace **`_start`** with **`main`**.

```
.text
.globl main          # Entry point for the program
```

Note that the order (sequence) of the following instructions *is not correct !*

```
    # Print the result
    la   a0, result_msg         # Load address of result message
    mv   a1, a0                 # Move result (in a0) to a1 for printf
    call printf                 # Call printf to print the result
```

It should be:

```
    # Print the result
    mv   a1, a0                  # Move result (in a0) to a1 for printf
    la   a0, result_msg          # Load address of result message
    call printf                  # Call printf to print the result
```

```
bako@k1:~/Programs/labgen$ gcc powersys.s -o powersys
bako@k1:~/Programs/labgen$ ./powersys
Enter base (a): 4
Enter exponent (b): 4
Result (a^b): 256
bako@k1:~/Programs/labgen$
```

Message:

Write assembly program for RISC-V RV64 with factorial function and number input with `scanf` function. Output the result with `printf` function.

Here is a RISC-V RV64 assembly program that combines reading an integer using `scanf`, calculating its factorial recursively, and outputting the result using `printf`.

```
    .data
fmt_scanf:  .asciz "%ld"              # Format string for scanf (reading a 64-bit integer)
fmt_printf: .asciz "Factorial: %ld\n"    # Format string for printf (printing a 64-bit integer)
number:     .quad 0                   # Variable to store the input number
result:     .quad 0                   # Variable to store the factorial result

    .text
    .globl _start

_start:
    # Step 1: Read an integer from the user using scanf

    # Load the address of the format string ("%ld") into a0
    la a0, fmt_scanf                  # First argument to scanf (format string)

    # Load the address of the variable 'number' into a1
    la a1, number                     # Second argument to scanf (address of variable)

    # Call scanf to read user input
    call scanf                        # Read an integer from stdin

    # Step 2: Compute the factorial of the number

    # Load the value of 'number' into a0 (for factorial calculation)
    ld a0, number                     # Load the input number into a0

    # Call the factorial function
    jal ra, factorial                 # Calculate factorial(a0) -> result in a0

    # Store the result in memory (at 'result')
    la t0, result                     # Load the address of the result variable
    sd a0, 0(t0)                      # Store the result in 'result'

    # Step 3: Print the result using printf

    # Load the format string for printf ("Factorial: %ld\n") into a0
    la a0, fmt_printf                 # First argument to printf (format string)

    # Load the result from memory into a1 (second argument for printf)
    ld a1, result                     # Load the factorial result into a1

    # Call printf to print the result
    call printf                       # Print the result

    # Step 4: Exit the program
    li a7, 93                         # syscall number for exit
    li a0, 0                          # exit code 0 (success)
    ecall                             # Exit the program

# Function: factorial
# Input: a0 (the number n)
# Output: a0 (the factorial of n)
factorial:
    # Base case: if a0 == 0 or a0 == 1, return 1
    li t0, 1                          # Load 1 into t0 for comparison
    beq a0, t0, base_case             # If a0 == 1, jump to base_case
    li t1, 0                          # Load 0 into t1
    beq a0, t1, base_case             # If a0 == 0, jump to base_case
    # Recursive case: n * factorial(n-1)
    addi sp, sp, -16                  # Allocate space on the stack
    sd ra, 8(sp)                      # Save return address (ra) to the stack

    addi a0, a0, -1                   # Decrement a0 by 1 (n = n - 1)
    jal ra, factorial                     # Call factorial(n-1)

    ld ra, 8(sp)                          # Restore return address (ra)
    addi sp, sp, 16                       # Deallocate space from the stack
    addi a0, a0, 1                        # a0 = a0 + 1 (restore original n)
    mul a0, a0, t0                        # a0 = n * factorial(n-1)
    ret                                   # Return to the caller

base_case:
    li a0, 1                              # Return 1 for base case
    ret                                   # Return to the caller
```

Explanation:

1. **`Data Section (.data):`**
    - `fmt_scanf`: Format string for `scanf` to read a 64-bit integer (`%ld`).
    - `fmt_printf`: Format string for `printf` to output a 64-bit integer (`Factorial: %ld\n`).
    - `number`: A 64-bit memory location to store the input number from the user.
    - `result`: A 64-bit memory location to store the computed factorial.
2. **Text Section (`.text`):**
    - **Step 1:** Read an integer from the user using `scanf`. The input is stored in the `number` variable.
    - **Step 2:** Load the number into register `a0` and call the `factorial` function, which computes the factorial of the number and returns it in `a0`.
    - **Step 3:** Store the result in the `result` variable, and print it using `printf`.
    - **Step 4:** Exit the program using the `exit` system call (`li a7, 93`).
3. **Factorial Function (`factorial`):**
    - The function is recursive. It uses the stack to store the return address for each recursive call.
    - **Base Case:** If `n == 0` or `n == 1`, the function returns 1.
    - **Recursive Case:** For `n > 1`, the function computes `n * factorial(n - 1)` by recursively calling itself with `n - 1` and multiplying the result by n.

Modifications:

Analysis, Modifications, Assembly, and Execution:

Modify the following section – it is not correct – it does not multiply **n*(n-1) – `t0`** equals **1** : we need this: **`a0= a0*a1.`** **Imagine** the evolution of the **stack.** Draw an example **for `a0=4.`**

It is:

```
    ld ra, 8(sp)                          # Restore return address (ra)
    addi sp, sp, 16                       # Deallocate space from the stack
    addi a0, a0, 1                        # a0 = a0 + 1 (restore original n)
    mul a0, a0, t0                        # a0 = n * factorial(n-1)
    ret                                   # Return to the caller
```

Should be:

```
    ld a1, 0(sp)            # Restore the original n value into a0 (after recursive call)
    ld ra, 8(sp)            # Restore the return address (ra)
    addi sp, sp, 16         # Deallocate stack space (pop)
    mul a0, a1, a0          # a0 = n * factorial(n - 1)
    ret                     # Return from the function
```

Compile and execute:

```
bako@k1:~/Programs/labgen$ gcc factorial.s -o factorial
bako@k1:~/Programs/labgen$ ./factorial
4
Result factorial: 24
```

Using external functions

The following is the same main section code with the additional call to external function to print one integer value. The function is prepared in the separate file **`print_integer.s`**

```
main:
    # Read base 'a'
    la   a0, input_format          # Load address of input format string ("%d")
    la   a1, number                 # Load address of base variable
    call scanf                    # Call scanf to read base into memory

    # Load the number into register a0
    # la a1, number              # Load address of number into a1
    ld a0, number               # Load the 64-bit value of number into a0 (this is the argument to
factorial)

    # Call factorial function
    jal ra, factorial          # Call factorial function, return value in a0

    # Print the result
    # mv   a1, a0                  # Move result (in a0) to a1 for printf
    #la   a0, result_msg           # Load address of result message
    #call printf                   # Call printf to print the result
    jal print_integer

    # Exit (end of program)
    li a7, 93                  # ecall for program exit (in RV64)
    ecall
```

The following is **`print_integer.s`** file with the corresponding function.

```
    .section .data
format_string:
    .string "%d\n"          # Format string to print an integer with newline
fmt_printf: .asciz "You entered: %ld\n"  # Format string for printf (printing a long integer)

    .section .text
    .globl print_integer

print_integer:
    # a0: The integer to print

    # Save registers that will be modified
    addi sp, sp, -16        # Allocate space on stack
    sd ra, 8(sp)            # Save return address
    sd t0, 0(sp)            # Save temporary register t0
    # Load the address of the format string
    mv a1, a0               # Move the integer (a0) into a1 (the printf argument)
    la a0, format_string    # Load the address of the format string into a1
    # System call for printf
    call   printf
    # Restore registers
    ld ra, 8(sp)            # Restore return address
    ld t0, 0(sp)            # Restore temporary register t0
    addi sp, sp, 16         # Deallocate stack space

    ret                     # Return from the function
```

The assembly command line:

```
$ gcc print_integer.s factorialext.s -o factorialext
```

Message:

Write assembly program for RISC-V RV64G to calculate the surface of the circle from its radius value.

Below is a simple RISC-V assembly program (RV64G) that calculates the surface area of a circle from a given radius. The formula for the surface area of a circle is:

Area=π×r2

In this example, we'll assume the value of π as 3.14159265359 and calculate the area using the formula. The result will be stored in **floating-point registers** (since RISC-V RV64G supports floating-point operations).

```
    .data
radius:  .float 5.0                 # Radius of the circle, adjust this value as needed
pi:      .float 3.1415927           # Approximation of pi in single precision
result:  .float 0.0                 # Variable to store the result

    .text
    .global _start

_start:
    # Load the radius and pi into floating-point registers
    flw     ft0, radius            # Load radius (r) into ft0 (single-precision)
    flw     ft1, pi                # Load pi into ft1 (single-precision)
    # Calculate r^2 (square of the radius)
    fmul.s  ft2, ft0, ft0          # ft2 = r * r (single-precision)
    # Calculate area = pi * r^2
    fmul.s  ft3, ft1, ft2          # ft3 = pi * (r * r) (single-precision)
    # Store the result back to memory
    fsw     ft3, result            # Store the result in 'result' (single-precision)
    # Exit the program (using system call for exit)
    li      a7, 93                 # System call number for exit in RISC-V (64-bit)
    ecall                           # Exit the program
```

Explanation:

1. **Data Section**:
    - **`radius`** contains the value of the radius of the circle (e.g., 5.0 in this example).
    - **`pi`** contains the approximation of π (3.14159265359).
    - **`result`** is where the computed area will be stored.
2. **Text Section**:
    - **`fld`** instructions load floating-point numbers (64-bit **`double`**) into floating-point registers.
    - **`fmul.d`** is used to multiply floating-point numbers (**`double`**).
    - The area is calculated using **π×r2** and the result is stored in the **`result`** memory location.
    - **`ecall`** is used to exit the program.

Modifications

Analyze, assembly and test the program. Add **`printf`** function.

Let us assembly the generated program:

```
bako@k1:~/Programs/labgen$ gcc circlegen.s -o circlegen
circlegen.s: Assembler messages:
circlegen.s:11: Error: illegal operands `flw ft0,radius'
circlegen.s:12: Error: illegal operands `flw ft1,pi'
circlegen.s:21: Error: illegal operands `fsw ft3,result'
bako@k1:~/Programs/labgen$
```

The flw and fsw instructions are not cerrect !

They may **modified** as follows to introduce address base in a4 and a5 registers before actually loading the values to the floating point registers.

```
    lla     a4 , radius
    flw     ft0, 0(a4)            # Load radius (r) into ft0 (single-precision)
    lla     a5, pi
    flw     ft1, 0(a5)                # Load pi into ft1 (single-precision)
```

and the same operation to store the floating point value:

```
    # Store the result back to memory
    lla     a4, result
    fsw     ft3, 0(a4)            # Store the result in 'result' (single-precision)
```

The introduction of **`printf`** to printout the result requires the **conversion and extension** of the floating point data:

```
fmt_printf: .asciz "Surface: %f\n"   # in .data section


    fcvt.d.s     ft3, ft3
    fmv.x.d      a1, ft3
    la           a0, fmt_printf
    call         printf
```

The following is the modified and completed code:

```
    .data
radius:  .float 10.0                # Radius of the circle, adjust this value as needed
pi:      .float 3.1415927          # Approximation of pi in single precision
result:  .float 0.0                # Variable to store the result
fmt_printf: .asciz "Surface: %f\n"

    .text
    .global main

main:
    # Load the radius and pi into floating-point registers
    lla     a4 , radius
    flw     ft0, 0(a4)            # Load radius (r) into ft0 (single-precision)
    lla     a5, pi
    flw     ft1, 0(a5)                # Load pi into ft1 (single-precision)

    # Calculate r^2 (square of the radius)
    fmul.s  ft2, ft0, ft0          # ft2 = r * r (single-precision)

    # Calculate area = pi * r^2
    fmul.s  ft3, ft1, ft2          # ft3 = pi * (r * r) (single-precision)

    # Store the result back to memory
    lla     a4, result
    fsw     ft3, 0(a4)            # Store the result in 'result' (single-precision)

    fcvt.d.s     ft3, ft3
    fmv.x.d      a1, ft3
    la      a0, fmt_printf
    call    printf

    # Exit the program (using system call for exit)
    li      a7, 93                 # System call number for exit in RISC-V (64-bit)
    ecall                           # Exit the program
```

Message:

What is the function of `fcvt.d.s ft3,ft3` RISC-V instruction

The instruction **`fcvt.d.s ft3,ft3`** in RISC-V performs a **floating-point conversion** from a **single-precision (32-bit)** floating-point value to a **double-precision (64-bit)** floating-point value.

Breakdown of the instruction:

- **`fcvt.d.s`**: This is the RISC-V floating-point conversion instruction that converts a value from **single-precision (32-bit)** floating-point (denoted by **`.s`**) to **double-precision (64-bit)** floating-point (denoted by **`.d`**).
- **`ft3`**: This is both the source and destination floating-point register. In this case, the source register contains a single-precision value, and the destination will contain the converted double-precision value.

Message:

What is the function of `fmv.x.d a1,ft3` RISC-V instruction.

The **`fmv.x.d a1,ft3`** instruction in RISC-V performs a **floating-point to integer move**. Specifically, it moves the contents of the **64-bit floating-point register `ft3`** into the **integer register `a1` without modifying the bit pattern**. This is a raw bitwise move, not a conversion between floating-point and integer values.

Breakdown of the instruction:

- **`fmv.x.d`**: This instruction moves the raw bit pattern from a **double-precision (64-bit)** floating-point register to a general-purpose integer register.
- **`a1`**: The destination general-purpose register (an integer register).
- **`ft3`**: The source floating-point register (double-precision floating-point register).

Message:

Write assembly program for RISC-V RV64 with vector extension V to add two vectors with 8 elements each using vector instructions. Each vector element is 32-bit integer.

Here's an assembly program for RISC-V RV64 with Vector Extension (RVV) to add two vectors with 8 elements each, where each element is a 32-bit integer. The program will load the vectors into vector registers, perform the addition using vector instructions, and store the result in memory.

```
    .section .data
vector_a:
    .word 1, 2, 3, 4, 5, 6, 7, 8       # First input vector with 8 elements (32-bit integers)

vector_b:
    .word 8, 7, 6, 5, 4, 3, 2, 1       # Second input vector with 8 elements (32-bit integers)

result_vector:
    .space 32                          # Space to store the result (8 elements * 4 bytes)

    .section .text
    .globl _start
_start:
    # Set up the vector length to 8 elements (each 32-bit wide)
    li t0, 8                            # Set vl (vector length) to 8 elements
    vsetvli t0, t0, e32, m1             # Set vector length (VL) to 8 elements, 32-bit wide

    # Load vector_a into vector register v1
    la t1, vector_a                     # Load address of vector_a into t1
    vle32.v v1,0(t1)                    # Load 8 elements (32-bit each) from vector_a into v1

    # Load vector_b into vector register v2
    la t2, vector_b                     # Load address of vector_b into t2
    vle32.v v2,0(t2)                    # Load 8 elements (32-bit each) from vector_b into v2

    # Perform vector addition: v3 = v1 + v2
    vadd.vv v3, v1, v2                  # Add vectors in v1 and v2, store result in v3

    # Store the result vector from v3 into memory
    la t3, result_vector                # Load address of result_vector into t3
    vse32.v v3,0(t3)                    # Store the result (32-bit elements) from v3 into memory

    # Exit the program
    li a0, 0                            # Exit code 0
    li a7, 93                           # Syscall number for exit
    ecall                               # Make the system call
```

Explanation:

1. **Data Section**:
    - `vector_a` and `vector_b` contain two input vectors, each with 8 elements of 32-bit integers.
    - `result_vector` is reserved to store the result, with 32 bytes of space (8 elements × 4 bytes each).
2. **Main Program**:
    - **Vector Length Setup**:
        - The `vsetvli` instruction sets the vector length (`vl`) to 8 elements, where each element is 32 bits wide (`e32`). `m1` indicates single-width elements.
    - **Vector Load**:
        - The `vle32.v` instruction loads 8 elements (32-bit integers) from `vector_a` and `vector_b` into vector registers `v1` and `v2`.
    - **Vector Addition**:

        - The `vadd.vv` instruction performs element-wise addition of the two vectors stored in `v1` and `v2`, and stores the result in `v3`.
    - **Vector Store**:
        - The `vse32.v` instruction stores the result from `v3` (which contains the 8 element-wise sums) into the memory location pointed to by `result_vector`.
3. **Program Exit**:
    - After the addition is complete, the program exits using the `exit` system call with exit code `0`.

Corrected and completed version.

```
   .data
vector1:  .word 1, 2, 3, 4, 5, 6, 7, 8          # First 8-word vector
vector2:  .word 1, 2, 3, 4, 5, 6, 7, 8          # Second 8-word vector
result:   .space 32                                                   # Space for the result (8 x 4
bytes)
result_msg:
    .string   "%d\n"  # Format string for result
format_string:
    .string "%d\n"          # Format string to print an integer with newline

    .text
    .global main
    .globl print_integer

print_integer_in:
    # a0: The integer to print
    # Save registers that will be modified
    addi sp, sp, -16        # Allocate space on stack
    sd ra, 8(sp)            # Save return address
    sd t0, 0(sp)            # Save temporary register t0
    # Load the address of the format string
    mv a1, a0               # Move the integer (a0) into a1 (the printf argument)
    la a0, format_string    # Load the address of the format string into a1
    # System call for printf
    call   printf
    # Restore registers
    ld ra, 8(sp)            # Restore return address
    ld t0, 0(sp)            # Restore temporary register t0
    addi sp, sp, 16         # Deallocate stack space
    ret                     # Return from the function

main:
    # Set up vector length register(VLEN = 8 elements, each 32 bits)
    li t0, 8                      # vector length is 8
    vsetvli t0,t0,e32,m1          # vector length for 32-bit elements, one operation per element
    # Load the vectors into vector registers
    la t1, vector1                   # Load address of vector1 into t1
    vle32.v v0,0(t1)                   # Load 8-word vector1 into vector register v0
    la t2, vector2                   # Load address of vector2 into t2
    vle32.v v1,0(t2)                   # Load 8-word vector2 into vector register v1
    # Perform vector addition
    vadd.vv v2, v0, v1               # v2 = v0 + v1 (element-wise vector addition)
    # Store the result back to memory
    la t3, result                    # Load address of result into t3
    vse32.v v2,0(t3)                   # Store the result from vector register v2 into memory
    # Print the result
     lw   a0, 28(t3)                   # Move result (in a0) - here: last element in the vector
     jal  print_integer_in             # call the prepared print function

    # Exit the program (using system call for exit)
    li a0,0
    li a7, 93                        # System call number for exit in RISC-V (64-bit)
    ecall                            # Exit the program
```

2. Modeling with Verilog HDL

Message:

What is Verilog HDL

Verilog HDL (Hardware Description Language) is a high-level language used to model, design, and simulate digital circuits. It allows engineers to describe the structure, behavior, and functionality of electronic systems, including microprocessors, memory, and custom logic, at various levels of abstraction (from gate-level to system-level). Verilog is widely used in the design and verification of digital hardware, including field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs).

Key Features of Verilog HDL

1. **Hardware Modeling**:
   - Verilog allows you to describe circuits at various abstraction levels:
     - **Gate level**: Describes the digital circuit using basic logic gates (AND, OR, etc.).
     - **Register-transfer level (RTL)**: Describes how data is transferred between registers and how operations are performed on data.
     - **Behavioral level**: Focuses on the behavior of the circuit, often resembling high-level programming.
2. **Concurrency**:
   - Unlike traditional programming languages that execute instructions sequentially, Verilog models the **concurrent** nature of hardware, where multiple parts of the circuit can operate simultaneously.
3. **Simulation and Synthesis**:
   - **Simulation**: Verilog can be used to simulate how a digital circuit behaves before physically implementing it. You can create **testbenches** to test the functionality of your design.
   - **Synthesis**: Verilog descriptions can be synthesized into gate-level representations and eventually into hardware, such as FPGAs or ASICs.
4. **Modules**:
   - A Verilog design is composed of **modules**, which are basic building blocks. Each module can represent a component or an entire system. Modules have inputs, outputs, and internal logic that define the behavior of the system.
5. **Structural and Behavioral Descriptions**:
   - **Structural modeling**: Describes the hardware design by connecting components (e.g., gates or submodules) together.
   - **Behavioral modeling**: Describes the functionality of the system without worrying about how it is connected at a low level. It focuses on "what" the circuit does rather than "how" it's done.

Example Verilog Code

Simple 2-to-1 Multiplexer (MUX):

```
module mux2to1 (
    input wire a,      // Input A
    input wire b,      // Input B
    input wire sel,    // Select line
    output wire y      // Output Y
);
    assign y = (sel) ? b : a;   // If sel = 1, choose B; otherwise, choose A
endmodule
```

In this example:

- A simple **2-to-1 multiplexer** is described in Verilog.
- `sel` is the select signal that controls which input (a or b) is passed to the output y.

Testbench Example (for the above MUX)

```
module tb_mux2to1;
    reg a, b, sel;
    wire y;

    mux2to1 uut (
        .a(a),
        .b(b),
        .sel(sel),
        .y(y)
    );

    initial begin
        // Test case 1: sel = 0, a = 0, b = 1
        a = 0; b = 1; sel = 0;
        #10; // Wait 10 time units
        $display("Test 1: y = %b", y);  // Should print y = 0

        // Test case 2: sel = 1, a = 0, b = 1
        sel = 1;
        #10;
        $display("Test 2: y = %b", y);  // Should print y = 1

        $finish; // End simulation
    end
endmodule
```

In this **testbench**:

- The multiplexer is instantiated and tested with different input combinations.
- The **`$display`** statement prints the output, allowing verification of the behavior in a simulation environment.

Applications of Verilog HDL

1. **FPGA Programming**:
    - FPGAs are reconfigurable hardware devices. Verilog is often used to program FPGAs by describing the desired logic and then synthesizing it into a hardware configuration.
2. **ASIC Design**:
    - In ASIC development, Verilog is used to design and verify digital circuits that will be manufactured as custom chips. It helps ensure that the design behaves correctly before it is fabricated.
3. **Digital Circuit Design and Simulation**:
    - Verilog allows engineers to model complex digital systems such as microprocessors, communication protocols, and data processing units, and simulate them for functional correctness.
4. **Prototyping and Testing**:
    - Verilog can be used to test hardware designs in simulation (using testbenches) to verify correctness before actual implementation. This is critical in reducing hardware design errors.

Advantages of Verilog HDL

1. **Industry Standard**:
    - Verilog is one of the most widely used HDLs in industry and academia, with extensive tool support for simulation, synthesis, and hardware testing.
2. **Ease of Use**:
    - Verilog syntax is similar to C, making it easier for software engineers to learn and transition to hardware design. It supports both structural (low-level) and behavioral (high-level) descriptions of hardware.
3. **Simulation and Debugging**:

- Verilog is well-suited for simulating digital designs before they are synthesized into hardware. It allows thorough testing, ensuring that the hardware will function as intended.

4. **Modular Design**:
    - Verilog supports hierarchical and modular design, allowing large systems to be broken down into smaller, manageable components. Each module can be independently designed and tested.

Message:

How o simulate Verilog HDL models with iverilog and show the simulation rtesults with GTKWave.

To simulate Verilog HDL models with **Icarus Verilog (iverilog)** and visualize the results with **GTKWave**, follow the step-by-step guide below:

Step 1: Install Required Tools

1. Install Icarus Verilog:

Icarus Verilog (iverilog) is a Verilog **simulation** and **synthesis** tool.

- On **Linux (Ubuntu)**, you can install it using:

```
sudo apt-get install iverilog
```

2. Install GTKWave:

GTKWave is a **waveform viewer** to visualize simulation results.

```
sudo apt-get install gtkwave
```

Step 2: Create a Verilog Testbench

You'll need two files:

1. **Verilog Module** (the design you want to simulate).
2. **Testbench** to drive the simulation.

Example: Simple Verilog Module and Testbench

File: `simple_module.v`

```
// simple_module.v
module simple_module(input a, input b, output y);
    assign y = a & b;  // AND gate
endmodule
```

File: `simple_tb.v` (**Testbench**)

```
// simple_tb.v
`timescale 1ns/1ps
module simple_tb;
    // Inputs
    reg a;
    reg b;
    // Output
    wire y;
    // Instantiate the design under test (DUT)
    simple_module dut (.a(a), .b(b), .y(y));
    // Testbench process
    initial begin
        // Initialize Inputs
        a = 0;
        b = 0;
        // Simulation sequence
        #10 a = 1;
        #10 b = 1;
        #10 a = 0;
        #10 b = 0;
        #10 a = 1; b = 1;
        #10;

        // End the simulation
        $finish;
    end
    // Dumping the waveform for GTKWave
    initial begin
        $dumpfile("simple_tb.vcd");  // Create the VCD file
        $dumpvars(0, simple_tb);     // Dump variables
    end
endmodule
```

Step 3: Compile and Simulate with Icarus Verilog

Once you have your Verilog module and **testbench** ready, follow these steps to simulate:

1. **Compile the Verilog files**:

```
iverilog -o simple_tb simple_tb.v simple_module.v
```

This will compile the Verilog files and create an output binary named `simple_tb`.

- **Run the simulation**:

```
vvp simple_tb
```

This command **runs** the compiled Verilog simulation model using `vvp`, the Icarus Verilog simulation runtime. If the **testbench** has **waveform dumping enabled** (`$dumpfile` and `$dumpvars`), this will create a `simple_tb.vcd` file.

In GTKWave:

- Once GTKWave opens, you should see the hierarchy of your Verilog module and signals in the **SST (Signal Selection Tree)** panel on the left.
- Select the signals (e.g., `a`, `b`, and `y`) and **drag them into the waveform panel** to visualize their behavior over time.
- The waveform display will show how the inputs (`a` and `b`) and output (`y`) change over time as per the simulation.

```
bako@k1:~/Design/labgen$ vi simple_module.v
bako@k1:~/Design/labgen$ vi simple_tb.v
bako@k1:~/Design/labgen$ iverilog -o simple_tb simple_tb.v simple_module.v
bako@k1:~/Design/labgen$
bako@k1:~/Design/labgen$ vvp simple_tb
VCD info: dumpfile simple_tb.vcd opened for output.
simple_tb.v:25: $finish called at 60000 (1ps)
bako@k1:~/Design/labgen$ ls -l
total 16
-rw-rw-r-- 1 bako bako  115  9月  7 15:02 simple_module.v
-rwxr-xr-x 1 bako bako 2165  9月  7 15:03 simple_tb
-rw-rw-r-- 1 bako bako  710  9月  7 15:03 simple_tb.v
-rw-rw-r-- 1 bako bako  464  9月  7 15:04 simple_tb.vcd
```

Summary

1. **Write Verilog Module** and **Testbench**.
2. **Compile the code** using `iverilog`:

```
iverilog -o <output_name> <testbench.v> <module.v>
```

Run the simulation using `vvp`:

```
vvp <output_name>
```

Open the waveform in `GTKWave`:

```
gtkwave <waveform.vcd>
```

With these steps, you can easily simulate your Verilog HDL designs and visualize the results using GTKWave.

Message:

Write in Verilog HDL simple ALU model with 32-bit words.

Here's a simple 32-bit **ALU (Arithmetic Logic Unit)** model written in Verilog HDL. This ALU can perform basic arithmetic and logic operations on 32-bit inputs, such as addition, subtraction, AND, OR, XOR, and shift operations.

ALU Model in Verilog HDL

```
module ALU(
    input [31:0] A,         // First 32-bit operand
    input [31:0] B,         // Second 32-bit operand
    input [3:0] ALUOp,      // 4-bit control signal to choose operation
    output reg [31:0] Result, // 32-bit output result
    output Zero             // Zero flag to indicate if result is zero
);

    always @(*) begin
        case (ALUOp)
            4'b0000: Result = A + B;         // Addition
            4'b0001: Result = A - B;         // Subtraction
            4'b0010: Result = A & B;         // Bitwise AND
            4'b0011: Result = A | B;         // Bitwise OR
            4'b0100: Result = A ^ B;         // Bitwise XOR
            4'b0101: Result = A << B[4:0];   // Logical shift left
            4'b0110: Result = A >> B[4:0];   // Logical shift right
            4'b0111: Result = $signed(A) >>> B[4:0]; // Arithmetic shift right
            default: Result = 32'b0;         // Default: Zero result
        endcase
    end

    // Zero flag: Set if result is zero
    assign Zero = (Result == 32'b0) ? 1'b1 : 1'b0;

endmodule
```

Explanation

- **Inputs:**
    - `A` and `B`: The two 32-bit input operands.
    - `ALUOp`: A 4-bit control signal to select the operation. This control signal determines which operation the ALU will perform.
- **Outputs:**
    - `Result`: The 32-bit output result of the ALU operation.
    - `Zero`: A flag indicating if the result of the operation is zero (`1` if `Result` is zero, otherwise `0`).
- **Operations:**
    - **Addition (`ALUOp = 4'b0000`)**: Adds the two operands.
    - **Subtraction (`ALUOp = 4'b0001`)**: Subtracts the second operand from the first.
    - **Bitwise AND (`ALUOp = 4'b0010`)**: Performs a bitwise AND between the two operands.
    - **Bitwise OR (`ALUOp = 4'b0011`)**: Performs a bitwise OR between the two operands.
    - **Bitwise XOR (`ALUOp = 4'b0100`)**: Performs a bitwise XOR between the two operands.
    - **Logical Shift Left (`ALUOp = 4'b0101`)**: Shifts the first operand left by the number of positions specified in the lower 5 bits of the second operand.
    - **Logical Shift Right (`ALUOp = 4'b0110`)**: Shifts the first operand right logically by the number of positions specified in the lower 5 bits of the second operand.
    - **Arithmetic Shift Right (`ALUOp = 4'b0111`)**: Shifts the first operand right arithmetically (preserving the sign bit).

- **Zero Flag**: The **`Zero`** output flag is set to `1` if the result is zero, and `0` otherwise. This flag is typically used in conditional branching to detect zero results.

Example Simulation

You can simulate this **ALU** with a simple **testbench** to see the results for various operations.

```
module ALU_tb; // Testbench module

    reg [31:0] A, B;
    reg [3:0] ALUOp;
    wire [31:0] Result;
    wire Zero;

    // Instantiate the ALU
    ALU uut (
        .A(A),
        .B(B),
        .ALUOp(ALUOp),
        .Result(Result),
        .Zero(Zero)
    );

    initial begin
        // Test addition
        A = 32'h00000005; B = 32'h00000003; ALUOp = 4'b0000;
        #10;
        $display("Addition: A = %h, B = %h, Result = %h, Zero = %b", A, B, Result, Zero);
        // Test subtraction
        A = 32'h00000005; B = 32'h00000005; ALUOp = 4'b0001;
        #10;
        $display("Subtraction: A = %h, B = %h, Result = %h, Zero = %b", A, B, Result, Zero);
        // Test AND
        A = 32'h0000000F; B = 32'h000000F0; ALUOp = 4'b0010;
        #10;
        $display("AND: A = %h, B = %h, Result = %h, Zero = %b", A, B, Result, Zero);
        // Test OR
        A = 32'h0000000F; B = 32'h000000F0; ALUOp = 4'b0011;
        #10;
        $display("OR: A = %h, B = %h, Result = %h, Zero = %b", A, B, Result, Zero);
        // Test XOR
        A = 32'h0000000F; B = 32'h000000F0; ALUOp = 4'b0100;
        #10;
        $display("XOR: A = %h, B = %h, Result = %h, Zero = %b", A, B, Result, Zero);
        // Test shift left
        A = 32'h00000001; B = 32'h00000004; ALUOp = 4'b0101;
        #10;
        $display("Shift Left: A = %h, B = %h, Result = %h, Zero = %b", A, B, Result, Zero);
        // Test shift right
        A = 32'h00000010; B = 32'h00000002; ALUOp = 4'b0110;
        #10;
        $display("Shift Right: A = %h, B = %h, Result = %h, Zero = %b", A, B, Result, Zero);

        // Test arithmetic shift right
        A = 32'h80000000; B = 32'h00000002; ALUOp = 4'b0111;
        #10;
        $display("Arithmetic Shift Right: A = %h, B = %h, Result = %h, Zero = %b", A, B, Result,
Zero);

        $finish;
    end

endmodule
```

Explanation of the Testbench

- The **testbench** drives different values to the inputs of the ALU and selects various operations using the **`ALUOp`** signal.
- The **result** and the **zero flag** are displayed for each test case using **`$display()`**.

- The testbench checks basic operations such as addition, subtraction, logical operations (AND, OR, XOR), and shift operations.

Conclusion

This simple Verilog HDL model simulates a 32-bit ALU that performs a variety of arithmetic and logical operations. It is commonly used as a key building block in RISC-V processors and other digital systems. The model can be further extended to support more complex operations as needed.

Compilation (`iverilog`) and execution (`vvp`):

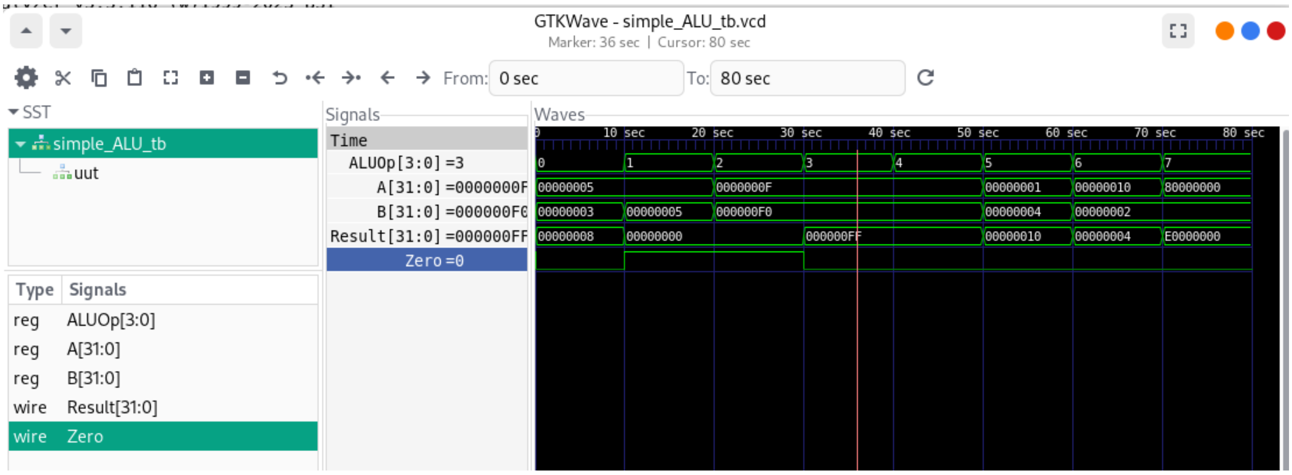
```
bako@k1:~/Design/labgen$ iverilog simple_ALU_tb.v simple_ALU.v -o simple_ALU

bako@k1:~/Design/labgen$ ls
simple_ALU  simple_ALU_tb.v  simple_ALU.v  simple_module.v  simple_tb  simple_tb.v  simple_tb.vcd

bako@k1:~/Design/labgen$ vvp simple_ALU

VCD info: dumpfile simple_ALU_tb.vcd opened for output.
Addition: A = 00000005, B = 00000003, Result = 00000008, Zero = 0
Subtraction: A = 00000005, B = 00000005, Result = 00000000, Zero = 1
AND: A = 0000000f, B = 000000f0, Result = 00000000, Zero = 1
OR: A = 0000000f, B = 000000f0, Result = 000000ff, Zero = 0
XOR: A = 0000000f, B = 000000f0, Result = 000000ff, Zero = 0
Shift Left: A = 00000001, B = 00000004, Result = 00000010, Zero = 0
Shift Right: A = 00000010, B = 00000002, Result = 00000004, Zero = 0
Arithmetic Shift Right: A = 80000000, B = 00000002, Result = e0000000, Zero = 0
simple_ALU_tb.v:52: $finish called at 80 (1s)

bako@k1:~/Design/labgen$gtkwave simple_ALU_tb.vcd
```



Message:

Write in Verilog HDL simple RAM memory model with 32-bit words.

Here's a simple **RAM memory model** written in **Verilog HDL**. This RAM model uses 32-bit words and allows you to perform read and write operations.

```
module RAM (
    input clk,                    // Clock signal
    input we,                     // Write enable (1 for write, 0 for read)
    input [7:0] addr,             // 8-bit address (for 256 words)
    input [31:0] data_in,         // 32-bit input data for writing
    output reg [31:0] data_out    // 32-bit output data for reading
);

    // Declare the RAM memory (256 words of 32-bit data)
    reg [31:0] memory [255:0];

    // Read/Write logic
    always @(posedge clk) begin
        if (we) begin
            // Write operation: If write enable is high, store data_in at addr
            memory[addr] <= data_in;
        end else begin
            // Read operation: If write enable is low, output data from addr
            data_out <= memory[addr];
        end
    end

endmodule
```

Explanation of the Model

- **Inputs:**
    - **`clk`**: Clock signal used to synchronize the memory operations.
    - `we`: Write enable signal. When **`we`** is **high** (`1`), the memory performs a **write operation**. When **`we`** is **low** (`0`), the memory performs a **read operation**.
    - **`addr`**: The 8-bit address input, allowing the RAM to access up to 256 memory locations.
    - **`data_in:`** The 32-bit input data, used when writing data to memory.
- **Output:**
    - **`data_out`**: The 32-bit output data, used when reading data from memory.
- **Memory Array:**
    - The memory array `memory` is a **256-word** RAM, where each word is **32 bits wide**. This is modeled using **`reg[31:0] memory[255:0]`**.
- **Behavior:**
    - On the **positive edge of the clock**, the RAM performs either a write or a read operation depending on the value of the `we` signal.
    - If `we` is high, the **`data_in`** is written to the memory location specified by **`addr`**.
    - If `we` is low, the **`data_out`** will hold the value stored in the memory location specified by **`addr`**.

Testbench for the RAM Model

Below is a simple testbench that simulates read and write operations on the RAM model.

```
module RAM_tb;

    reg clk;                // Clock signal
    reg we;                 // Write enable signal
    reg [7:0] addr;         // 8-bit address for memory
    reg [31:0] data_in;     // 32-bit input data for write
    wire [31:0] data_out;   // 32-bit output data for read
    // Instantiate the RAM module
    RAM uut (
        .clk(clk),
        .we(we),
        .addr(addr),
        .data_in(data_in),
        .data_out(data_out)
    );
    // Clock generator: Toggle clk every 5 time units
    always #5 clk = ~clk;
    initial begin
        // Initialize signals
        clk = 0;
        we = 0;
        addr = 0;
        data_in = 0;
        // Write data to address 0
        #10;
        we = 1;
        addr = 8'h00;       // Address 0
        data_in = 32'hDEADBEEF; // Data to write
        #10;
        // Write data to address 1
        we = 1;
        addr = 8'h01;       // Address 1
        data_in = 32'hCAFEBABE; // Data to write
        #10;
        // Read data from address 0
        we = 0;
        addr = 8'h00;       // Address 0
        #10;
        $display("Read Address 0: Data = %h", data_out);
        // Read data from address 1
        we = 0;
        addr = 8'h01;       // Address 1
        #10;
        $display("Read Address 1: Data = %h", data_out);
        // Finish simulation
        $finish;
    end

endmodule
```

Explanation of the Testbench

- The **testbench** initializes the clock (`clk`) and sets it to toggle every 5 time units.
- It performs the following operations:
    1. Writes `32'hDEADBEEF` to address `0x00`.
    2. Writes `32'hCAFEBABE` to address `0x01`.
    3. Reads from address `0x00` and prints the value to the console.
    4. Reads from address `0x01` and prints the value to the console.
- After these operations, the simulation stops.

Expected Output

The **testbench** will print the following output after simulating the read operations:

```
Read Address 0: Data = DEADBEEF
Read Address 1: Data = CAFEBABE
```

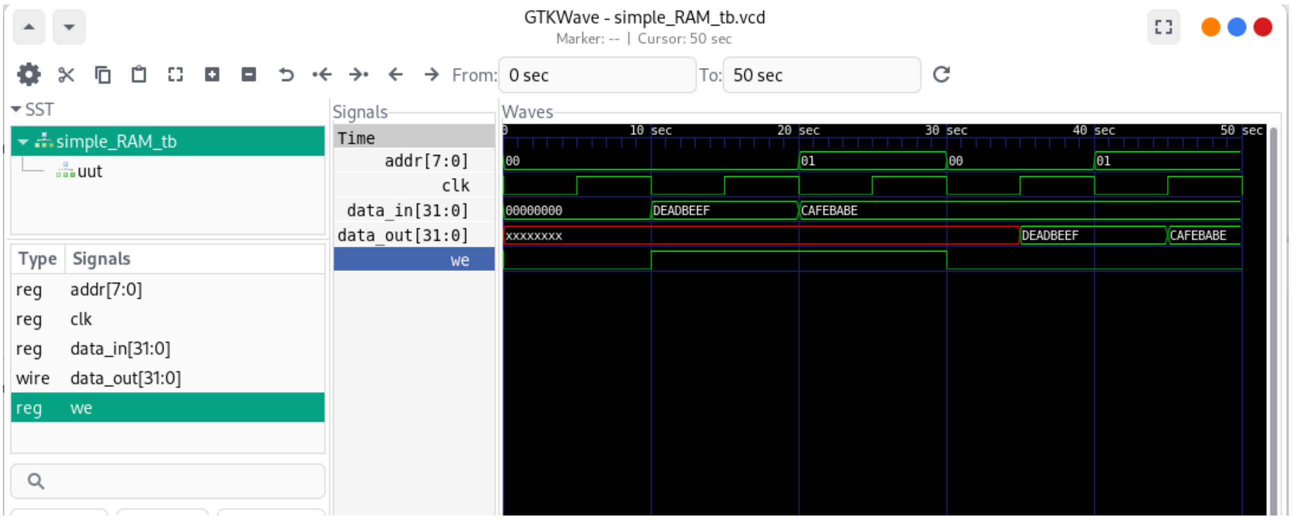
Compilation and execution:

```
bako@k1:~/Design/labgen$ iverilog simple_RAM.v simple_RAM_tb.v –o simple_RAM
bako@k1:~/Design/labgen$ vvp simple_RAM
VCD info: dumpfile simple_RAM_tb.vcd opened for output.
Read Address 0: Data = deadbeef
Read Address 1: Data = cafebabe
simple_RAM_tb.v:54: $finish called at 50 (1s)

bako@k1:~/Design/labgen$ ls –l
```

```
-rwxr-xr-x 1 bako bako 3665  9月  7 16:19 simple_RAM
-rw-rw-r-- 1 bako bako 1575  9月  7 16:19 simple_RAM_tb.v
-rw-rw-r-- 1 bako bako  981  9月  7 16:19 simple_RAM_tb.vcd
-rw-rw-r-- 1 bako bako  805  9月  7 16:15 simple_RAM.v

bako@k1:~/Design/labgen$gtkwave simple_RAM_tb.vcd
```



From: 0 sec

To: 50 sec

Type	Signals
reg	addr[7:0]
reg	clk
reg	data_in[31:0]
wire	data_out[31:0]
reg	we

Summary :

This **32-bit RAM model** allows for read and write operations with a 32-bit data width and an 8-bit address space, allowing for up to 256 memory locations.

- The **testbench** demonstrates basic read and write operations on the RAM, showing how to interact with the memory and verify the results.
- You can extend or modify this RAM model for larger or more complex systems by adjusting the address width or adding additional control logic.

Message :

Write Verilog HDL code to model and simulate basic RISC-V (I) instructions set.

Creating a **Verilog HDL model** to simulate the basic **RISC-V I instruction set** involves designing a simplified RISC-V processor. This processor will include key components such as the instruction **memory**, data memory, **ALU**, and **control logic**, and will execute basic RISC-V instructions like **`add, sub, lw, sw, beq,`** and **`jalr`**.

Steps to Create a Verilog Model:

1. **Define the instruction memory (ROM)** to store the RISC-V instructions.
2. **Define registers** to hold data and perform operations.
3. **Create an ALU** to perform arithmetic and logical operations.
4. **Create control logic** to decode instructions and generate control signals.
5. **Simulate the execution of instructions** with a testbench.

Basic RISC-V I Instruction Set

We will handle a subset of instructions from the RV32I set:

- Arithmetic: **`add, sub`**
- Logical: **`and, or, xor`**
- Memory access: **`lw, sw`**
- Control flow: **`beq, jalr`**

1. ALU Module

```
module ALU(
    input [31:0] a, b,
    input [3:0] alu_control,
    output reg [31:0] result,
    output zero
);

    always @(*) begin
        case (alu_control)
            4'b0000: result = a + b;    // ADD
            4'b0001: result = a - b;    // SUB
            4'b0010: result = a & b;    // AND
            4'b0011: result = a | b;    // OR
            4'b0100: result = a ^ b;    // XOR
            default: result = 0;
        endcase
    end

    assign zero = (result == 0) ? 1'b1 : 1'b0;

endmodule
```

2. Instruction Memory (ROM)

```
module InstructionMemory(
    input [31:0] pc,
    output reg [31:0] instruction
);

    reg [31:0] memory [0:255];   // 256 32-bit memory locations
    initial begin
        // Load instructions here (machine code)
        memory[0] = 32'h00000013;  // NOP (addi x0, x0, 0)
        memory[1] = 32'h00100093;  // addi x1, x0, 1  -> x1 = 1
        memory[2] = 32'h00208133;  // add x2, x1, x2  -> x2 = x1 + x2
        memory[3] = 32'h002101b3;  // add x3, x2, x2  -> x3 = x2 + x2
        memory[4] = 32'h00000073;  // ECALL (for terminating simulation)
    end

    always @(*) begin
        instruction = memory[pc[9:2]];  // Fetch the instruction from memory
    end

endmodule
```

3. Register File

```
module RegisterFile(
    input clk,
    input [4:0] read_reg1, read_reg2, write_reg,
    input [31:0] write_data,
    input reg_write,
    output [31:0] read_data1, read_data2
);
    reg [31:0] registers [0:31];  // 32 32-bit registers

    initial begin
        registers[0] = 32'b0;  // x0 is always 0
    end
    // Reading registers
    assign read_data1 = registers[read_reg1];
    assign read_data2 = registers[read_reg2];
    // Writing to a register
    always @(posedge clk) begin
        if (reg_write && write_reg != 0)  // Don't write to x0
            registers[write_reg] <= write_data;
    end

endmodule
```

4. Control Unit

```
module ControlUnit(
    input [6:0] opcode,
    output reg [1:0] alu_op,
    output reg reg_write, alu_src, mem_to_reg, mem_write, branch
);

    always @(*) begin
        case (opcode)
            7'b0110011: begin // R-type (add, sub)
                alu_op = 2'b10;
                reg_write = 1;
                alu_src = 0;
                mem_to_reg = 0;
                mem_write = 0;
                branch = 0;
            end
            7'b0000011: begin // I-type (lw)
                alu_op = 2'b00;
                reg_write = 1;
                alu_src = 1;
                mem_to_reg = 1;
                mem_write = 0;
                branch = 0;
            end
            7'b0100011: begin // S-type (sw)
                alu_op = 2'b00;
                reg_write = 0;
                alu_src = 1;
                mem_to_reg = 0;
                mem_write = 1;
                branch = 0;
            end
            7'b1100011: begin // B-type (beq)
                alu_op = 2'b01;
                reg_write = 0;
                alu_src = 0;
                mem_to_reg = 0;
                mem_write = 0;
                branch = 1;
            end
            default: begin // Default for unsupported instructions
                alu_op = 2'b00;
                reg_write = 0;
                alu_src = 0;
                mem_to_reg = 0;
                mem_write = 0;
                branch = 0;
            end
        endcase
    end
endmodule
```

5. CPU Module

This is the top-level module that integrates all the components together.

```
module CPU(
    input clk,
    input reset
);

    reg [31:0] pc;              // Program counter
    wire [31:0] instruction;    // Current instruction
    wire [31:0] read_data1, read_data2;  // Register read data
    wire [31:0] alu_result;     // ALU result
    wire zero_flag;             // ALU zero flag
    wire reg_write, alu_src, mem_to_reg, mem_write, branch;
    wire [1:0] alu_op;
    // Decode the instruction
    wire [6:0] opcode = instruction[6:0];
    wire [4:0] rs1 = instruction[19:15];
    wire [4:0] rs2 = instruction[24:20];
    wire [4:0] rd = instruction[11:7];
    wire [2:0] funct3 = instruction[14:12];
    wire [6:0] funct7 = instruction[31:25];
    wire [31:0] imm = {{20{instruction[31]}}, instruction[31:20]}; // Sign-extended immediate
    // ALU control
    wire [3:0] alu_control;
    assign alu_control = (alu_op == 2'b10 && funct3 == 3'b000 && funct7 == 7'b0000000) ?
4'b0000 : // ADD
                         (alu_op == 2'b10 && funct3 == 3'b000 && funct7 == 7'b0100000) ?
4'b0001 : // SUB
                         (alu_op == 2'b10 && funct3 == 3'b111) ? 4'b0010 : // AND
                         (alu_op == 2'b10 && funct3 == 3'b110) ? 4'b0011 : // OR
                         (alu_op == 2'b10 && funct3 == 3'b100) ? 4'b0100 : 4'b0000; // XOR
    // PC logic
    always @(posedge clk or posedge reset) begin
        if (reset)   pc <= 0;
        else if (branch && zero_flag)
            pc <= pc + imm;
        else
            pc <= pc + 4;
    end
    // Instantiate modules
    InstructionMemory im(.pc(pc), .instruction(instruction));
    RegisterFile rf(.clk(clk), .read_reg1(rs1), .read_reg2(rs2), .write_reg(rd),
                    .write_data(alu_result), .reg_write(reg_write),
                    .read_data1(read_data1), .read_data2(read_data2));
    ALU alu(.a(read_data1), .b(alu_src ? imm : read_data2), .alu_control(alu_control),
            .result(alu_result), .zero(zero_flag));
    ControlUnit cu(.opcode(opcode), .alu_op(alu_op), .reg_write(reg_write), .alu_src(alu_src),
                   .mem_to_reg(mem_to_reg), .mem_write(mem_write), .branch(branch));
endmodule
```

6. Testbench

To simulate the CPU, we need a testbench that clocks the CPU and checks for expected results.

```
module TestBench;

    reg clk;
    reg reset;
    // Instantiate CPU
    CPU cpu(.clk(clk), .reset(reset));
    // Clock generator
    always #5 clk = ~clk;
    initial begin
        // Initialize signals
        clk = 0;
        reset = 1;
        #10 reset = 0;
        // Run for some clock cycles
        #100 $finish;
    end

endmodule
```

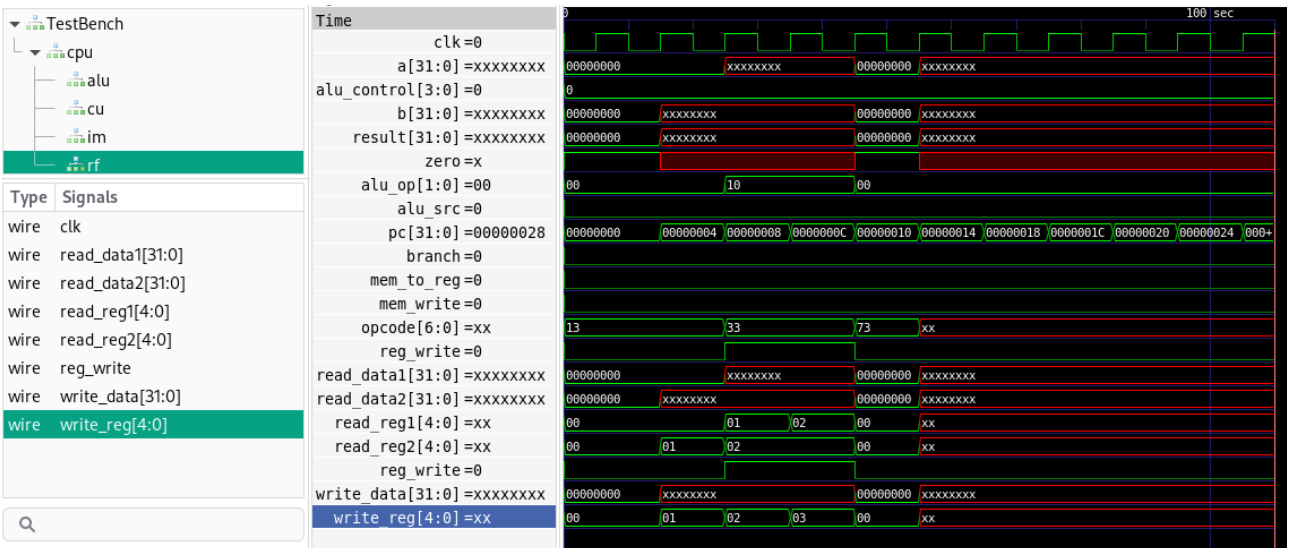
Simulation

- The **testbench** will simulate the processor and its response to a simple set of instructions (e.g., `addi, add, sub`).

Summary

This Verilog HDL model covers the basic RISC-V I instruction set operations and includes an ALU, control unit, register file, and instruction memory. You can modify the instruction memory to load different RISC-V instructions for testing different scenarios.

Compilation, execution



Message:

Write Verilog HDL code to describe and model all RIsc-V I instructions. Provide several modules such as data path module control path module register block module, instruction and data memory modules.

Creating a complete Verilog HDL description of the entire RISC-V "I" instruction set involves several modules to implement the datapath, control logic, register block, and memory modules. Below is a simplified version that models the key components, covering the execution of the most common RISC-V "I" (integer) instructions like **`ADDI, SLTI, ANDI, ORI, XORI, LUI, AUIPC, JAL, JALR, BEQ, BNE, LB, LH, LW, SB, SH,`** and **`SW`**.

Key Modules:

1. **Datapath Module**: Handles arithmetic and logic operations, register file access, immediate generation, and instruction execution.
2. **Control Path Module**: Generates control signals based on the instruction opcode and funct3/funct7 fields.
3. **Register File Module**: Models the register block (32 registers in RISC-V).
4. **Instruction and Data Memory Modules**: For storing instructions and data.
5. **Top-Level Processor Module**: Combines all the above modules.

1. DataPath module

```
module DataPath(
    input wire clk,
    input wire reset,
    input wire [31:0] instr,      // Instruction from instruction memory
    input wire [31:0] read_data,  // Data read from data memory
    output reg [31:0] pc,        // Program Counter - must be reg
    output wire [31:0] alu_result,
    output wire [31:0] write_data,  // Data to be written to memory
    output wire mem_write          // Memory write signal
);

    wire [6:0] opcode = instr[6:0];
    wire [4:0] rs1 = instr[19:15];
    wire [4:0] rs2 = instr[24:20];
    wire [4:0] rd = instr[11:7];
    wire [2:0] funct3 = instr[14:12];
    wire [6:0] funct7 = instr[31:25];
    wire [31:0] imm;
    wire [31:0] reg_rs1, reg_rs2;
    wire [31:0] alu_src_b;
    wire branch, mem_read, reg_write, alu_src, mem_to_reg;
    wire [2:0] alu_op;
    wire [31:0] pc_next, pc_plus4, alu_out, reg_write_data;
    // Program Counter Logic
    assign pc_plus4 = pc + 4;
    assign pc_next = branch ? alu_result : pc_plus4;
    always @(posedge clk or posedge reset) begin
        if (reset)
            pc <= 0;
        else
            pc <= pc_next;
    end
    // Control Unit
    ControlPath control_unit(
        .opcode(opcode),
        .funct3(funct3),
        .branch(branch),
        .mem_read(mem_read),
        .mem_to_reg(mem_to_reg),
        .alu_op(alu_op),
        .mem_write(mem_write),
        .alu_src(alu_src),
        .reg_write(reg_write)
    );
    // Register File
    RegisterFile reg_file(
        .clk(clk),
        .rs1(rs1),
        .rs2(rs2),
        .rd(rd),
        .reg_write(reg_write),
        .write_data(reg_write_data),
        .read_data1(reg_rs1),
        .read_data2(reg_rs2)
    );
    // ALU
    ALU alu_unit(
        .alu_control(alu_op),
        .input_a(reg_rs1),
        .input_b(alu_src_b),
        .result(alu_result)
    );
    // Immediate Generator
    ImmGen imm_gen(
        .instr(instr),
        .imm_out(imm)
    );
    // ALU Source Mux
    assign alu_src_b = alu_src ? imm : reg_rs2;
    // Data to be written to register file
    assign reg_write_data = mem_to_reg ? read_data : alu_result;
    assign write_data = reg_rs2;

endmodule
```

2. ControlPath module

```
module ControlPath(
    input wire [6:0] opcode,
    input wire [2:0] funct3,
    output wire branch,
    output wire mem_read,
    output wire mem_to_reg,
    output wire [2:0] alu_op,
    output wire mem_write,
    output wire alu_src,
    output wire reg_write
);

    reg [8:0] control_signals;

    always @(*) begin
        case (opcode)
            7'b0010011: control_signals = 9'b001_010_001;  // ADDI, ORI, ANDI (I-type ALU)
            7'b0000011: control_signals = 9'b011_010_001;  // LW (Load)
            7'b0100011: control_signals = 9'b000_001_100;  // SW (Store)
            7'b1100011: control_signals = 9'b100_000_010;  // BEQ, BNE (Branch)
            7'b1101111: control_signals = 9'b101_000_001;  // JAL (Jump)
            default:    control_signals = 9'b000_000_000;  // Default (NOP)
        endcase
    end

    assign branch = control_signals[8];
    assign mem_read = control_signals[7];
    assign mem_to_reg = control_signals[6];
    assign alu_op = control_signals[5:3];
    assign mem_write = control_signals[2];
    assign alu_src = control_signals[1];
    assign reg_write = control_signals[0];

endmodule
```

3. RegisterFile module

```
module RegisterFile(
    input wire clk,
    input wire [4:0] rs1, rs2, rd,
    input wire reg_write,
    input wire [31:0] write_data,
    output wire [31:0] read_data1, read_data2
);

    reg [31:0] regfile [31:0];
    always @(posedge clk) begin
        if (reg_write)
            regfile[rd] <= write_data;
    end
    assign read_data1 = regfile[rs1];
    assign read_data2 = regfile[rs2];
endmodule
```

4. ALU module

```
module ALU(
    input wire [2:0] alu_control,
    input wire [31:0] input_a, input_b,
    output reg [31:0] result
);
    always @(*) begin
        case (alu_control)
            3'b000: result = input_a + input_b;  // ADD, ADDI
            3'b001: result = input_a - input_b;  // SUB
            3'b010: result = input_a & input_b;  // AND
            3'b011: result = input_a | input_b;  // OR
            3'b100: result = (input_a < input_b) ? 1 : 0;  // SLT, SLTI
            default: result = 0;
        endcase
    end
endmodule
```

5. ImmGen module

```
module ImmGen(
    input wire [31:0] instr,
    output reg [31:0] imm_out  // must be reg
);

    wire [6:0] opcode = instr[6:0];
    always @(*) begin
        case (opcode)
            7'b0010011:  imm_out = {{20{instr[31]}}, instr[31:20]};  // I-Type
            7'b1100011:  imm_out = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0};  //
B-Type
            7'b0000011:  imm_out = {{20{instr[31]}}, instr[31:20]};  // Load
            7'b0100011:  imm_out = {{20{instr[31]}}, instr[31:25], instr[11:7]};  // Store
            default:  imm_out = 32'b0;
        endcase
    end
endmodule
```

6. Instruction Memory module

```
module InstructionMemory(
    input wire [31:0] addr,
    output wire [31:0] instr
);
    reg [31:0] memory [0:255];  // Memory to store 256 instructions
    assign instr = memory[addr[31:2]];

endmodule
```

7. Data Memory module

```
module DataMemory(
    input wire clk,
    input wire [31:0] addr,
    input wire [31:0] write_data,
    input wire mem_write,
    output wire [31:0] read_data
);
    reg [31:0] memory [0:255];  // Data memory
    always @(posedge clk) begin
        if (mem_write)
            memory[addr[31:2]] <= write_data;
    end
    assign read_data = memory[addr[31:2]];
endmodule
```

8. Top level - Processor module

```
module Processor(
    input wire clk,
    input wire reset
);
    wire [31:0] instr, read_data, pc, alu_result, write_data;
    wire mem_write;
    // Datapath Module
    Datapath datapath(
        .clk(clk),
        .reset(reset),
        .instr(instr),
        .read_data(read_data),
        .pc(pc),
        .alu_result(alu_result),
        .write_data(write_data),
        .mem_write(mem_write)
    );
    // Instruction Memory
    InstructionMemory instr_mem(
        .addr(pc),
        .instr(instr)
    );
    // Data Memory
    DataMemory data_mem(
        .clk(clk),
        .addr(alu_result),
        .write_data(write_data),
        .mem_write(mem_write),
        .read_data(read_data)
    );
endmodule
```

Summary:

- **Datapath** handles the flow of data and execution of instructions.
- **Control Path** generates control signals based on the instruction opcode.
- **Register File** manages the CPU's 32 registers.
- **ALU** performs arithmetic and logic operations.
- **Immediate Generator** extracts the immediate value for instructions that need it.
- **Instruction and Data Memory** store the instructions and data, respectively.

This Verilog code describes the essential components for modeling the RISC-V "I" instruction set, covering key operations like arithmetic, logical operations, memory access, and branching.

Modifications, Compilation, and Test

Extension of InstructionMemory.v module:

```
module InstructionMemory(
    input wire [31:0] addr,
    output wire [31:0] instr
);
    reg [31:0] memory [0:255];  // Memory to store 256 instructions
    assign instr = memory[addr[31:2]];
    initial begin
            $display("Loading Instruction Memory");
            $readmemh("ram_mem.hex",memory);
    end
endmodule
```

and edition of `ram_mem.hex` file for the program. (to be modified)

```
07b10093
001141b3
07b10093
07b10093
07b10093
07b10093
07b10093
07b10093
07b10093
001141b3
001141b3
001141b3
..
```

Edition of `Processor_tb.v` file to test the `Processor`.

```
module Processor_tb;
    reg clk;
    reg reset;
    // Instantiate CPU
    Processor cpu(.clk(clk), .reset(reset));
    // Clock generator
    always #5 clk = ~clk;
    initial begin
        // Initialize signals
        clk = 0;
        reset = 1;
        #10 reset = 0;
        // Run for some clock cycles
        #200 $finish;
    end
        initial begin
        $dumpfile("Processor_tb.vcd");  // Create the VCD file
        $dumpvars(0, Processor_tb);     // Dump variables
    end

endmodule
```

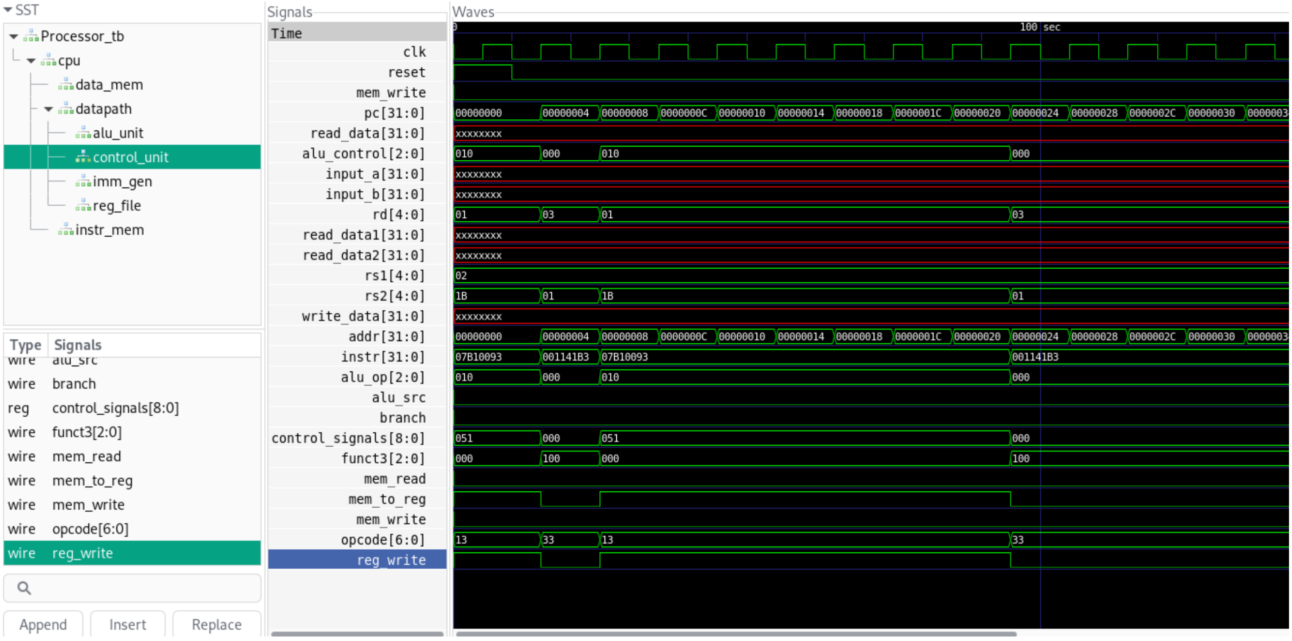
Compilation

```
bako@k1:~/Design/labgen/proc2$ iverilog ControlPath.v RegisterFile.v ALU.v ImmGen.v DataPath.v
InstructionMemory.v DataMemory.v Processor.v Processor_tb.v -o Processor_tb
```

and

Simulation

```
bako@k1:~/Design/labgen/proc2$ vvp Processor_tb
Loading Instruction Memory
WARNING: InstructionMemory.v:9: $readmemh(ram_mem.hex): Not enough words in the file for the
requested range [0:255].
VCD info: dumpfile Processor_tb.vcd opened for output.
Processor_tb.v:14: $finish called at 210 (1s)
```


SST
Processor_tb
cpu
data_mem
datapath
alu_unit
control_unit
imm_gen
reg_file
instr_mem
Type
Signals
wire alu_src
wire branch
reg control_signals[8:0]
wire funct3[2:0]
wire mem_read
wire mem_to_reg
wire mem_write
wire opcode[6:0]
wire reg_write
Append
Insert
Replace
Signals
Time
clk
reset
mem_write
pc[31:0]
read_data[31:0]
alu_control[2:0]
input_a[31:0]
input_b[31:0]
rd[4:0]
read_data1[31:0]
read_data2[31:0]
rs1[4:0]
rs2[4:0]
write_data[31:0]
addr[31:0]
instr[31:0]
alu_op[2:0]
alu_src
branch
control_signals[8:0]
funct3[2:0]
mem_read
mem_to_reg
mem_write
opcode[6:0]
reg_write
Waves
0
100 sec
00000000 00000004 00000008 0000000C 00000010 00000014 00000018 0000001C 00000020 00000024 00000028 0000002C 00000030 0000003
xxxxxxxx
010 000 010 000
xxxxxxxx
xxxxxxxx
01 03 01 03
xxxxxxxx
xxxxxxxx
02
1B 01 1B 01
xxxxxxxx
00000000 00000004 00000008 0000000C 00000010 00000014 00000018 0000001C 00000020 00000024 00000028 0000002C 00000030 0000003
07B10093 001141B3 07B10093 001141B3
010 000 010 000
051 000 051 000
000 100 000 100
13 33 13 33