Why RISC-V Processor

1. Introduction to RISC-V

- RISC-V (pronounced "Risk-Five") is an open standard instruction set architecture (ISA) based on the principles of Reduced Instruction Set Computing (RISC).
- Developed at the University of California, Berkeley, RISC-V is designed to be simple, modular, and extensible.

2. Key Features of RISC-V

- **Open and Free:** RISC-V is an open ISA, meaning it is not restricted by patents or licenses. Developers can use it for free.
- **Simplicity and Modularity:** It has a small base ISA, with optional standard extensions for added functionality (e.g., floating-point operations, vector processing).
- **Scalability:** Suitable for a wide range of applications—from embedded systems to supercomputers.
- **Ecosystem Support:** Supported by a growing community, including tools, simulators, compilers, and hardware implementations.

3. Advantages Over Proprietary ISAs

- **Cost-Effectiveness:** Eliminates licensing fees and reduces dependency on a single vendor.
- **Innovation-Friendly:** Encourages experimentation and research due to its openness.
- **Standardization:** Promotes consistency in hardware and software development.

4. Why Now?

- Growing demand for customized processors in AI/ML, IoT, and cloud computing.
- Challenges in proprietary systems, such as restricted control and licensing costs.

5. Examples of RISC-V in Use

- Processors: SiFive processors, C-DAC VEGA processors.
- Applications: IoT devices, research projects, and educational purposes.

RISC-V Processor Overview

1. What is a Processor?

- A processor, or CPU, is the brain of a computer. It executes instructions and performs calculations necessary to run software and control hardware.
- The ISA defines the instructions the processor can execute and the way it interacts with memory and peripherals.

2. RISC-V Processor Overview

- RISC-V processors are implementations of the RISC-V ISA. Multiple designs exist, each tailored to specific applications.
- RISC-V processors are typically categorized into:
  - **Embedded:** Optimized for low-power, resource-constrained systems.
  - **Application:** Designed for general-purpose computing.
  - **High-Performance:** Targeting computationally intensive tasks like AI/ML.

3. Components of a RISC-V Processor

- **Core:** The part of the processor that executes instructions.
  - Includes Arithmetic Logic Unit (ALU), Control Unit, and Registers.
- **Memory Hierarchy:**
  - Cache (L1, L2)
  - Main memory
- **Peripherals:**
  - GPIO, UART, SPI, I2C, etc., for hardware interaction.

4. Standard RISC-V Profiles

- **RV32I:** Base ISA for 32-bit systems.
- **RV64I:** Base ISA for 64-bit systems.
- **Extensions:** Modular additions to enhance functionality.
  - M: Integer multiplication and division.
  - A: Atomic instructions.
  - F: Single-precision floating-point.

- D: Double-precision floating-point.
- V: Vector instructions for parallel processing.
- G: General-Purpose (I + M + A + F + D)
- C: Compressed instructions (16-bit instructions)

5. Pipeline Architecture

- RISC-V processors often use a pipeline to improve performance.
- Example stages:
  - Fetch: Retrieve the instruction from memory.
  - Decode: Interpret the instruction.
  - Execute: Perform the operation.
  - Memory Access: Read/write data to memory.
  - Write-back: Store results back to registers.

6. Customizability

- One of RISC-V's strengths is the ability to implement custom instructions for specific applications.

ARM vs. RISC-V

1. Introduction

- ARM and RISC-V are two popular ISAs in the modern processor landscape.
- While ARM is proprietary and widely used, RISC-V is open and gaining traction for its flexibility.

2. Key Comparisons

Aspect	ARM	RISC-V
Ownership	Proprietary, owned by ARM Holdings	Open standard, managed by the RISC-V Foundation
Licensing	License required for design and use	Free to use, no licensing fees

Aspect	ARM	RISC-V
Architecture	Complex but mature	Simpler and modular
Ecosystem	Established, with widespread adoption	Growing rapidly with community support
Customizability	Limited; must adhere to ARM standards	Highly customizable; allows custom instructions
Applications	Mobile devices, IoT, servers	IoT, research, AI/ML, education, high-performance computing
Instruction Set	Rich, with advanced features	Minimal, modular base with extensions
Tools and Support	Well-supported (e.g., ARM GCC, DS-5, Keil)	Strong support growing (e.g., GCC, LLVM, QEMU, Spike)

3. Technical Comparisons

- **Instruction Length:**
  - ARM uses fixed-length (32-bit) instructions in ARM mode; Thumb mode allows variable-length.
  - RISC-V uses fixed-length 32-bit instructions in the base ISA, with support for compressed 16-bit instructions (`C` extension).
- **Register Set:**
  - ARM: Registers vary across versions; modern ARM (AArch64) has 31 general-purpose registers.
  - RISC-V: Consistent register set with 32 general-purpose registers.
- **Interrupt Handling:**
  - ARM: Sophisticated interrupt handling; uses multiple modes for exceptions.
  - RISC-V: Simplified interrupt model with a flexible CSR-based approach.
- **Pipeline Complexity:**
  - ARM: Generally more complex, with deeper pipelines in higher-performance cores.
  - RISC-V: Simplified pipeline designs with flexibility for customization.

4. Philosophical Differences

- ARM focuses on licensing optimized processor designs and tools.
- RISC-V prioritizes openness, academic research, and fostering innovation without barriers.

Si-Five U74 Pipeline Stages vs Cortex-A Pipeline

1. **SiFive U74 (RISC-V):**
    - **Pipeline Depth:** 8 stages.
    - **Stages:**
        1. **Fetch (IF):** Instruction fetch from memory.
        2. **Decode (ID):** Decode the instruction and read registers.
        3. **Execute (EX):** Perform ALU operations.
        4. **Memory (MEM):** Access memory for load/store instructions.
        5. **Write-back (WB):** Write results to the register file.
    - Focuses on simplicity and power efficiency, with basic branch prediction.
2. **ARM Cortex-A:**
    - **Pipeline Depth:** Varies; Cortex-A72 has up to 15 stages.
    - **Stages:**
        1. **Fetch:** Instruction fetch.
        2. **Decode:** Multi-level decode for complex instructions.
        3. **Issue:** Dispatch to multiple execution units.
        4. **Execute:** Includes integer, floating-point, and vector operations.
        5. **Memory Access:** Access data memory.
        6. **Write-back:** Store results back to registers.
    - Advanced features like out-of-order execution, deeper branch prediction, and SIMD processing increase complexity and performance.

Comparison:

- The U74 pipeline is simpler, with fewer stages and in-order execution, making it ideal for low-power applications.
- Cortex-A pipelines are deeper and more complex, optimized for higher throughput and latency hiding.

Modes in RISC-V

1. Introduction to Processor Modes

- Processor modes determine the privilege level at which code runs.
- Privilege levels control access to resources, system instructions, and memory regions.
- RISC-V defines a simple and effective privilege model.

2. RISC-V Privilege Levels

- RISC-V specifies three privilege levels:
    1. **User Mode (U):**
        - Least privileged.
        - For running application-level code.
        - No direct access to hardware or system-critical instructions.
    2. **Supervisor Mode (S):**
        - Medium privilege.
        - Used by operating systems to manage resources.
        - Can execute privileged instructions like context switches.
    3. **Machine Mode (M):**
        - Highest privilege level.
        - Handles critical tasks like hardware initialization, interrupt handling, and booting.
        - Mandatory in all RISC-V implementations.

3. Additional Modes

- Some implementations may add custom privilege levels (e.g., Hypervisor Mode `H` for virtualization).

4. Key Features of Each Mode

- **Control and Status Registers (CSRs):**
    - Each mode has its own set of CSRs to manage privileges, exceptions, and interrupts.
- **Physical Memory Protection (PMP):**
    - Used in `M` mode to control memory access for lower privilege levels.

- **Trap Handling:**
  - Lower privilege levels can transfer control to higher levels through traps (e.g., exceptions, interrupts).

5. Mode Transitions

- Mode transitions occur when:
  - **System Calls:** Application code in `U` mode requests OS services (trap to `S` mode).
  - **Interrupts/Exceptions:** Hardware triggers a transfer to `M` or `S` mode.
  - **Return from Trap:** Resumes execution at the previous privilege level.

RISC-V Register Set

1. Classification of Registers in RISC-V

RISC-V has a clean and simple register set, categorized as follows:

Type	Name	Count	Purpose
General-Purpose (GPR)	`x0` to `x31`	32	Used for arithmetic, logical, and data operations.
Control and Status	`mstatus`, `mtvec`, etc.	Varies	Control traps, interrupts, and privilege levels.
Program Counter	`pc`	1	Points to the current instruction being executed.
Floating-Point	`f0` to `f31` (optional)	32	Used for floating-point operations (if supported).

2. General-Purpose Registers (GPRs)

Register	Name	Description
`x0`	Zero Register	Always holds the constant `0`. Writing to it has no effect.
`x1`	Return Address	Used for storing return addresses (`ra`).
`x2`	Stack Pointer	Points to the top of the stack (`sp`).

Register	Name	Description
x3	Global Pointer	Global Pointer (gp).
x4	Thread Pointer	Thread Pointer (tp).
x5-x7	Temporary	Caller-saved temporary registers (t0 to t2).
x8	Frame Pointer	Frame pointer (fp) or saved register (s0).
x9	Saved Register	Saved register (s1).
x10-x17	Argument/Return	Used to pass arguments and return values (a0 to a7).
x18-x27	Saved Registers	Callee-saved registers (s2 to s11).
x28-x31	Temporary	Caller-saved temporary registers (t3 to t6).

3. Special Registers

1. **Program Counter (pc)**
    - Holds the memory address of the next instruction to execute.
    - Automatically updated after each instruction.
2. **Control and Status Registers (CSRs)**
    - mstatus: Machine status register. Tracks privilege level and processor state.
    - mtvec: Machine trap vector register. Specifies the base address of the trap handler.
    - mie / mip: Machine interrupt enable/pending registers.
    - We will cover CSRs in depth later.

4. Register Conventions

- **Caller-Saved Registers**: t0 to t6, a0 to a7.
- **Callee-Saved Registers**: s0 to s11.
- Always follow the RISC-V **calling convention** when writing functions.

RISC-V Calling Convention

The calling convention in RISC-V specifies:

1. **How functions receive parameters.**
2. **Where return values are stored.**
3. **How the call stack is used.**
4. **Which registers must be saved and restored by a function.**

Key Points:

1. **Register Classification:**
   - **Argument Registers**: `a0-a7` are used to pass arguments to functions (up to 8 arguments).
   - **Return Registers**: `a0` and `a1` are used to store return values (if more than one value is returned).
   - **Temporary Registers**: `t0-t6` can be used freely by a function but are not preserved across function calls.
   - **Saved Registers**: `s0-s11` must be preserved across function calls. If a function modifies them, it must save and restore their values.
   - **Stack Pointer**: `sp` points to the current top of the stack.
   - **Frame Pointer**: `s0` (or `fp`) is used as a frame pointer for certain calling conventions.
2. **Stack Usage:**
   - The stack grows downward in memory.
   - The caller is responsible for allocating space for arguments beyond `a7` (if more than 8 are needed).
   - Local variables are stored in the stack frame.
3. **Calling a Function:**
   - The caller saves any registers that need to be preserved.
   - The callee sets up a stack frame, typically storing `ra` (return address) and `s0`.
4. **Returning from a Function:**
   - The callee restores saved registers.
   - The callee sets the return values in `a0` and `a1`.

- Control is transferred back to the caller using `ret`.

Example (Assembly Code):

```
# Function: int add(int x, int y)
# Arguments in a0 (x) and a1 (y), result in a0
add:
    add a0, a0, a1  # Add x and y, store result in a0
    ret             # Return to caller

# Main program
main:
    li a0, 5        # Load immediate 5 into a0
    li a1, 10       # Load immediate 10 into a1
    call add        # Call the add function
    # Result is now in a0 (15)
    # Program continues...
```

Instruction Formats and Types in RISC-V

RISC-V instructions are designed to be simple and efficient. Each instruction is 32 bits wide in the base ISA, though extensions like compressed instructions (C-extension) allow for 16-bit instructions.

1. Instruction Formats

RISC-V uses several formats to encode different types of instructions. Each format divides the 32-bit instruction into fields for specific purposes like opcode, register addresses, and immediate values.

- **R-Type (Register):** Used for operations involving two source registers and one destination register.

```
| opcode (7) | rd (5) | funct3 (3) | rs1 (5) | rs2 (5) | funct7 (7) |
```

- Example: `add x3, x1, x2` (x3 = x1 + x2)

- **I-Type (Immediate):** Used for operations with one source register and an immediate value.

```
| opcode (7) | rd (5) | funct3 (3) | rs1 (5) | imm[11:0] (12) |
```

  - Example: `addi x3, x1, 10` (x3 = x1 + 10)

- **S-Type (Store):** Used for store instructions.

```
| opcode (7) | imm[4:0] (5) | funct3 (3) | rs1 (5) | rs2 (5) | imm[11:5] (7) |
```

  - Example: `sw x2, 8(x1)` (Store x2 at address x1 + 8)

- **B-Type (Branch):** Used for conditional branch instructions.

```
| opcode (7) | imm[11] (1) | imm[4:1] (4) | funct3 (3) | rs1 (5) | rs2 (5) | imm[10:5] (6) | imm[12] (1) |
```

  - Example: `beq x1, x2, label` (Branch if x1 == x2)

- **U-Type (Upper Immediate):** Used for loading 20-bit upper immediate.

```
| opcode (7) | rd (5) | imm[31:12] (20) |
```

  - Example: `lui x1, 0x12345` (Load 0x12345000 into x1)

- **J-Type (Jump):** Used for jump instructions.

```
| opcode (7) | rd (5) | imm[20] (1) | imm[10:1] (10) | imm[11] (1) | imm[19:12] (8) |
```

- Example: `jal x1, label` (Jump and link)

2. Instruction Types

RISC-V instructions can be categorized by their functionality:

- **Arithmetic and Logical Instructions:**
  - Examples: `add`, `sub`, `and`, `or`, `sll`, `sra`
- **Memory Access Instructions:**
  - Examples: `lw`, `sw`, `lb`, `lh`
- **Control Transfer Instructions:**
  - Examples: `jal`, `jalr`, `beq`, `bne`
- **Immediate Instructions:**
  - Examples: `addi`, `andi`, `ori`
- **System Instructions:**
  - Examples: `ecall`, `ebreak`

Practical Example:

```
# Example to demonstrate R-Type, I-Type, and S-Type

main:
    li x1, 10          # Load immediate 10 into x1 (I-Type)
```

```
    li x2, 20         # Load immediate 20 into x2 (I-Type)
    add x3, x1, x2    # Add x1 and x2, store result in x3 (R-Type)
    sw x3, 0(x0)      # Store x3 value at memory address 0 (S-Type)
    ret
```

Build Process in RISC-V

The build process involves converting high-level language programs (like C or assembly) into machine-executable binaries. For RISC-V, the typical build process includes the following steps:

1. Steps in the Build Process

1. **Source Code Creation:**
   - Write your code in a high-level language (C/C++) or assembly language.
2. **Preprocessing:**
   - The preprocessor expands macros, includes headers, and resolves conditional compilation directives.
   - Command: `riscv64-linux-gnu-gcc -E source.c -o source.i`
3. **Compilation:**
   - The compiler converts the preprocessed code into assembly language.
   - Command: `riscv64-linux-gnu-gcc -S source.i -o source.s`
4. **Assembly:**
   - The assembler translates the assembly code into object code (machine language).
   - Command: `riscv64-linux-gnu-as source.s -o source.o`
5. **Linking:**
   - The linker combines object files and resolves references to create an executable.
   - Command: `riscv64-linux-gnu-ld source.o -o executable`

6. **Execution:**
    - The binary is loaded into the RISC-V processor (real hardware or an emulator like QEMU) for execution.
    - Command: `qemu-riscv64 ./executable`

2. Tools for the Build Process

You have already set up the necessary tools; here's how they fit into the process:

- **Compiler:** `riscv64-linux-gnu-gcc`
- **Assembler:** Part of the GCC toolchain.
- **Linker:** `ld` (GNU Linker).
- **Emulator:** `QEMU`.

3. Example: Compiling and Running a RISC-V Program

C Program:

```c
#include <stdio.h>

int main() {
    int a = 5, b = 10, c;
    c = a + b;
    printf("Sum: %d\n", c);
    return 0;
}
```

Build Steps:

1. **Compile the Code:**

```
riscv64-linux-gnu-gcc -o program program.c
```

2. **Run on QEMU:**

```
qemu-riscv64 ./program
```

Assembly Program:

```
.section .data
msg:    .asciz "Sum: %d\n"

.section .text
.global main

main:
    li a0, 5          # Load immediate 5 into a0
    li a1, 10         # Load immediate 10 into a1
    add a2, a0, a1    # Add a0 and a1, store result in a2
    la a0, msg        # Load address of msg into a0
    mv a1, a2         # Move result to a1 for printf
    call printf       # Call printf function
    ret
```

Build Steps:

1. **Assemble and Link:**

```
riscv64-linux-gnu-gcc -o program program.s
```

2. **Run on QEMU:**

```
qemu-riscv64 ./program
```

Practical Examples of Instructions

Below are examples of most common types of RISC-V instructions:

1. Arithmetic Operations

Arithmetic operations involve basic calculations like addition, subtraction, multiplication, and division.

Example: Adding Two Numbers

```
.section .text
.global main

main:
    li a0, 15        # Load immediate 15 into a0
    li a1, 20        # Load immediate 20 into a1
    add a2, a0, a1   # Add a0 and a1, store result in a2
    ret              # Return
```

2. Logical Operations

Logical operations perform bitwise operations like AND, OR, XOR, and shifts.

Example: Bitwise AND

```
.section .text
.global main
```

```
main:
    li a0, 0xF0      # Load immediate 0xF0 into a0
    li a1, 0x0F      # Load immediate 0x0F into a1
    and a2, a0, a1   # Perform bitwise AND
    ret              # Return
```

3. Memory Operations

These instructions load data from memory or store data to memory.

Example: Load and Store

```
.section .data
value:  .word 42    # Define a word in memory with value 42

.section .text
.global main

main:
    la a0, value     # Load address of value into a0
    lw a1, 0(a0)     # Load word at address a0 into a1
    sw a1, 4(a0)     # Store a1 at address a0 + 4
    ret              # Return
```

4. Branching

Branching instructions control program flow based on conditions.

Example: Check Equality

```
.section .text
.global main

main:
    li a0, 10        # Load immediate 10 into a0
    li a1, 10        # Load immediate 10 into a1
    beq a0, a1, equal # Branch if a0 == a1
    ret              # Return

equal:
    li a2, 1         # Set a2 to 1 (true)
    ret
```

5. Jumping

Jumping is used for unconditional control flow.

Example: Infinite Loop

```
.section .text
.global main

main:
    j main           # Jump to main (infinite loop)
```

6. Function Calls

Using `jal` and `jalr` for calling functions and returning.

Example: Calling a Function

```
.section .text
.global main

main:
    li a0, 5          # Load 5 into a0
    li a1, 7          # Load 7 into a1
    call add          # Call add function
    ret               # Return

add:
    add a0, a0, a1    # Add a0 and a1, store result in a0
    ret               # Return to caller
```