RISC V

WHAT IS RISC V ?

- RISC-V (pronounced "risk-five") is an open-source instruction set architecture (ISA) for designing computer processors
- RISC-V is open-source, meaning that its design is publicly available, and anyone can implement it without needing to pay licensing fees. This openness has led to a growing ecosystem of RISC-V hardware and software development tools.

WHY RISC V

- Open Source
- Simplicity
- Modularity
- Customization

ARM VS RISC V

- 1. Licensing and Openness:

- ARM: ARM is a proprietary ISA developed by ARM Holdings , Licensing ARM's architecture involves fees and agreements with ARM Ltd.

- RISC-V: RISC-V is an open-source ISA with publicly available specifications. It can be used without licensing fees, making it more accessible for customization and development.

2. Ecosystem:

ARM: ARM has a well-established ecosystem with a wide range of processors, development tools, and third-party support. It's widely used in mobile devices, embedded systems, and data centers.

RISC-V: RISC-V's ecosystem is growing rapidly but is still evolving. It has gained traction in academia, research, startups, and certain industry segments but may not have the same breadth of support as ARM.

3. Customization:

ARM: ARM processors are often customized by ARM's licensees, allowing for some flexibility in design. However, customization options can be limited compared to RISC-V.

RISC-V: RISC-V is highly customizable. Users can create custom instructions and extensions to tailor processors for specific applications, potentially optimizing performance and power efficiency.

4. Architecture Variants:

ARM: ARM offers various architecture variants, including ARM Cortex-A (application processors), Cortex-R (real-time processors), and Cortex-M (microcontrollers), each tailored to different application domains.

RISC-V: RISC-V offers scalability through standard profiles and optional extensions, accommodating a wide range of applications, from microcontrollers to high-performance computing.

5. Licensing Costs:

ARM: Licensing ARM's technology typically involves costs, which can vary depending on the specific agreement and design requirements.

RISC-V: RISC-V is cost-effective for organizations since it is open source and does not involve licensing fees.

In summary, ARM and RISC-V are two different ISA options with distinct advantages and trade-offs. ARM is known for its industry dominance, established ecosystem, and specialized architecture variants, while RISC-V offers openness, customization, and cost-effectiveness. The choice between ARM and RISC-V depends on specific application requirements, customization needs, licensing considerations, and the availability of ecosystem support.

MODES IN RISC V

- In the context of the RISC-V instruction set architecture (ISA), there are several modes that define the privilege levels and execution environments of a processor. These modes are designed to provide different levels of privilege and control to the software running on the processor.
- The primary privilege modes in RISC-V are:

1. Machine Mode (M-mode): Machine mode is the most privileged mode in RISC-V. It is typically used by the machine's firmware and operating system kernel to control and manage the entire system. In this mode, the processor has access to all resources and instructions, and it can execute any instruction.
2. Supervisor Mode (S-mode): Supervisor mode is a privilege level below machine mode. It is intended for the supervisor software, such as the operating system kernel, that manages user programs and hardware resources. In S-mode, certain instructions and resources are restricted, providing a level of isolation and protection between the supervisor and user programs

   (Provides support for operating systems like linux,windows,etc)

3.User Mode (U-mode): User mode is the least privileged mode in RISC-V. It is used for running regular user applications. In U-mode, many instructions and resources that could potentially disrupt the system are restricted, providing isolation and security.(Isolate application processes from each other and from trusted code running in M-mode)

- **M** : Simple Embedded Systems
- **M,U** : Embedded systems with memory protection
- **M,S,U** : Operating Systems.

- It's worth noting that additional privilege modes, such as hypervisor mode&Debug mode might be implemented by specific processors or extensions to the RISC-V architecture to support virtualization or other specialized use cases. However, the three modes mentioned above (M-mode, S-mode, and U-mode) are the core privilege levels defined by the RISC-V ISA.

RISC V REGISTER SET:

Registers (x0 to x31):

x0: Hardwired to the constant value 0. It is often used as the "zero" register.

x1 to x31: General-purpose integer registers. These registers are used for arithmetic and data manipulation operations.

1. Program Counter (pc):

The program counter is used to keep track of the address of the current instruction being executed. It points to the next instruction to be fetched and executed.

2. Stack Pointer (sp):

The stack pointer is used to manage the stack in memory. It points to the top of the stack, and it is often used for managing function calls and local variables.

3. Link Register (ra):

The link register is used to store the return address when a function call is made. It is used to return control to the calling function after a function call is complete.

4. Global Pointer (gp):
The global pointer is used to access global data or variables

5. Thread Pointer (tp):
The thread pointer is used in multi-threaded environments to point to thread-local storage.

7. Saved Registers :(s0-s11)

These are callee-saved registers. They are used to save the values of general-purpose registers that need to be preserved across function calls. The specific registers that are callee-saved can vary depending on the calling convention.

8. CSR Registers (csrs):
Control and Status Register (CSR) registers are a set of special-purpose registers used for controlling various aspects of the processor's behavior, including privilege modes, interrupts, and other system-related functions. The number and functionality of CSR registers can vary depending on the RISC-V variant.

9.Argument Register:
In RISC-V, 8 argument registers, namely, x10 to x17 are used to pass arguments in a subroutine.
Before a subroutine call is made, the arguments to the subroutine are copied to the argument registers

10.Temporary Register:
As the name suggests, the temporary registers are used to hold intermediate values during instruction execution. There are seven temporary registers (t0 − t6) in RISC-V.

11. Additional Registers and Extensions:

RISC-V processors may include additional registers or extensions based on their specific design and requirements. For example, processors with vector extensions may include vector registers for SIMD (Single Instruction, Multiple Data) operations.

Register Name	ABI Name	Description
x0	zero	Hard-Wired Zero
x1	ra	Return Address
x2	sp	Stack Pointer
x3	gp	Global Pointer
x4	tp	Thread Pointer
x5	t0	Temporary/Alternate Link Register
x6-7	t1-t2	Temporary Register
x8	s0/fp	Saved Register (Frame Pointer)
x9	s1	Saved Register
x10-11	a0-a1	Function Argument/Return Value Registers
x12-17	a2-a7	Function Argument Registers
x18-27	s2-s11	Saved Registers
x28-31	t3-t6	Temporary Registers

Control and status Registers:

- In the RISC-V architecture, Control and Status Registers (CSRs) play a vital role in controlling and monitoring the processor's operation
- These CSRs are used by the processor and privileged software to manage the execution environment, handle exceptions and interrupts, and configure the behavior of the CPU.
- Here are some key aspects and examples of control and status registers in RISC-V:

1. CSR Addressing: CSRs are identified by their numeric CSR address. The CSR address is a 12-bit value that is used to access a specific CSR. To read or write to a CSR, you use special instructions like csrrw, csrrs, and csrrc.
2. Privilege Levels: RISC-V defines several privilege levels, including User (U), Supervisor (S), Hypervisor (H), and Machine (M). Different CSRs may be accessible at different privilege levels. For example, M-mode CSRs can typically be accessed by all privilege levels, whereas S-mode CSRs are only accessible by S and M modes.

3. M-Mode CSRs: Machine mode CSRs are used for controlling the whole machine. They include:

mstatus: Controls processor status and control bits.
mcause: Indicates the cause of the most recent exception.
mtvec: Holds the base address of the trap-handler.
mepc: Holds the program counter at the time of an exception.
mie, mip: Control and indicate machine-level interrupts.

4. S-Mode CSRs: Supervisor mode CSRs are used when running in supervisor mode. They include:

- sstatus: Similar to mstatus but for supervisor mode.
- scause: Similar to mcause but for supervisor mode.
- stvec: Similar to mtvec but for supervisor mode.
- sepc: Similar to mepc but for supervisor mode.
- sie, sip: Similar to mie and mip but for supervisor mode.

5. U-Mode CSRs: User mode CSRs are used when running in user mode and are more limited in functionality compared to the supervisor and machine mode CSRs. Examples include ustatus and utvec.

some example CSR-related assembly instructions:

csrrw x1, mstatus, x2: Read the mstatus CSR into x1 and write the value of x2 into it.

csrrs x1, mie, x2: Read the mie CSR into x1 and set the bits specified in x2.

csrrc x1, mip, x2: Read the mip CSR into x1 and clear the bits specified in x2.

- Custom and Platform-Specific CSRs: Some RISC-V implementations may define custom CSRs for platform-specific features. These are implementation-defined and can vary between different RISC-V processors.

Risc V ISA

- What is ISA?
- It is a crucial aspect of computer architecture and defines the set of instructions that a computer's central processing unit (CPU) can execute.
- The ISA acts as an interface between the hardware and software of a computer system, allowing software programs to communicate with and control the hardware components, especially the CPU.
- **Instruction Sets :**

1. **RV32I**
2. **RV32M**
3. **RV32C**
4. **RV32F/D**
5. **RV32E**

- There are 6 Instruction formats
  - R-type
  - I-type
  - S-type
  - B-type
  - U-type
  - J-type

Instruction Encoding:

Format	Bit																													
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- **opcode (7 bits):** Partially specifies which of the 6 types of *instruction formats*.
- **funct7, and funct3 (10 bits):** These two fields, further than the *opcode* field, specify the operation to be performed.
- **rs1 (5 bits):** Specifies, by index, the register containing first operand (i.e., source register).
- **rs2 (5 bits):** Specifies the second operand register.
- **rd (5 bits):** Specifies the destination register to which the computation result will be directed.

RV32I:

1. Register operations

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends

2. Immediate operations:

addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends

3.load and store

lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	

4.Branches:

beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends

5.Jumps and load immidiate

jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	