Center of Excellence: Supercomputing for AI & Big-Data

RISC-V Processor Hardware and Instruction Set Architecture

by: Tassadaq Hussain

Director Centre for AI and BigData

Professor Department of Electrical Engineering

Namal University Mianwali

Collaborations:

Barcelona Supercomputing Center, Spain

European Network on High Performance and Embedded Architecture and Compilation

Pakistan Supercomputing Center

Topics

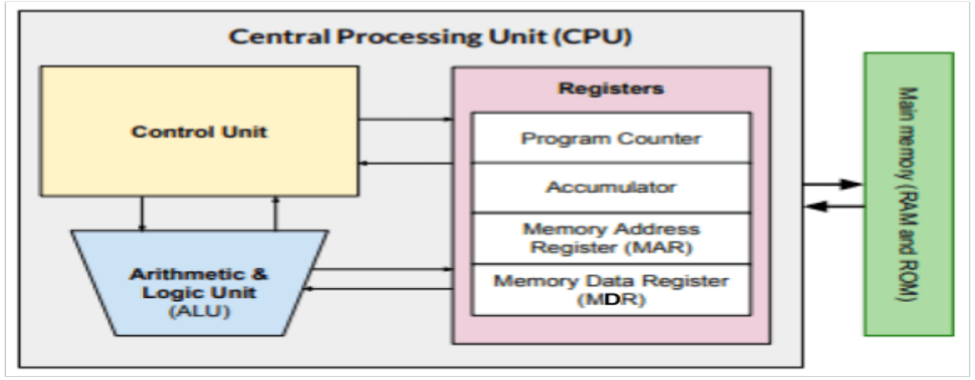
1. **Basic Processor Architecture**
2. Different Types of Processor Architectures
3. RISC-V Processor Architecture
4. RISC-V Instruction Set Architecture
5. Programming RISC-V using assembly language

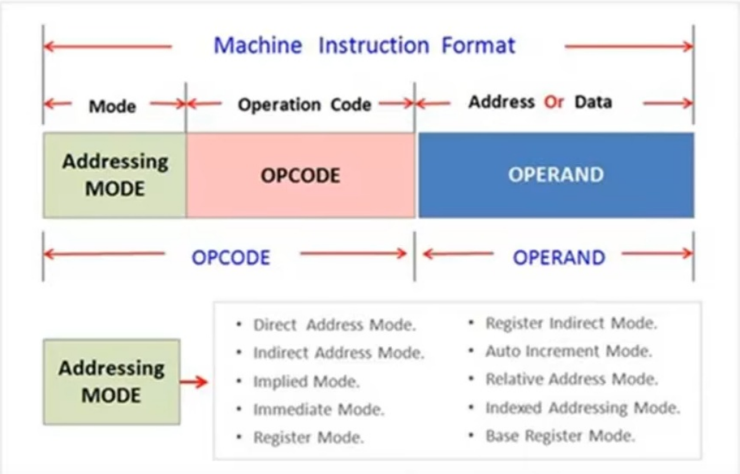
Basic Processor Architecture

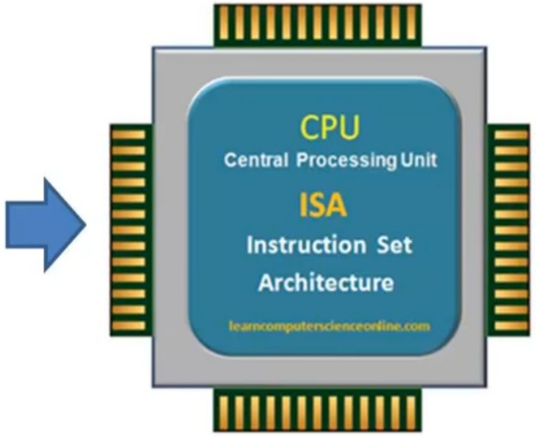
Processor Architecture refers to the design and organization of a processor's central processing unit (CPU).

Components of Processor:

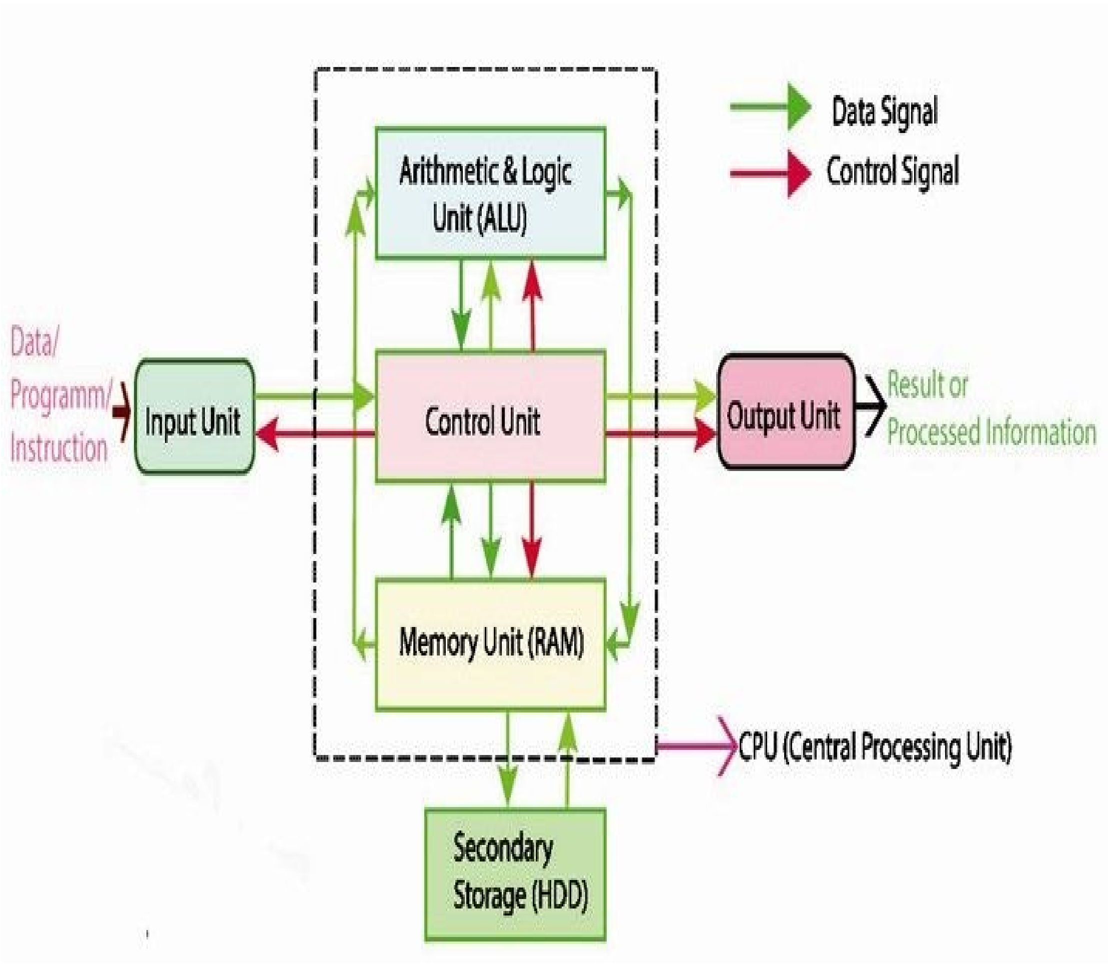
- **Arithmetic and Logic Unit:** Performs mathematical calculations.
- **Control Unit:** Control the overall processing of the processor.
- **Decoders Unit:** Convert coded instructions into signals that can control other components.
- **Registers:** Hold data, instructions, and addresses temporarily during processing.
- **Buses:** Electrical pathways that transmit data and signals between components. Types include the data bus, address bus, and control bus.







- **Clock:** Generates timing signals to synchronize the operations of the CPU components. The clock speed determines how many instructions per second the CPU can execute.
- **Instruction Set Architecture (ISA):** Defines the set of instructions the CPU can execute
- **Cache:** Stores frequently accessed data and instructions to speed up processing.
- **Memory Management Unit (MMU):** Handles the translation of virtual addresses to physical addresses. Manages memory protection and caching.
- **Input/Output (I/O)** Interfaces: Allow the CPU to communicate with peripheral devices. Include ports and controllers for devices such as keyboards, mice, and storage.
- **Power Control Unit:**



Arithmetic Logic Unit ALU:

In computing, an arithmetic logic unit (ALU) is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers.

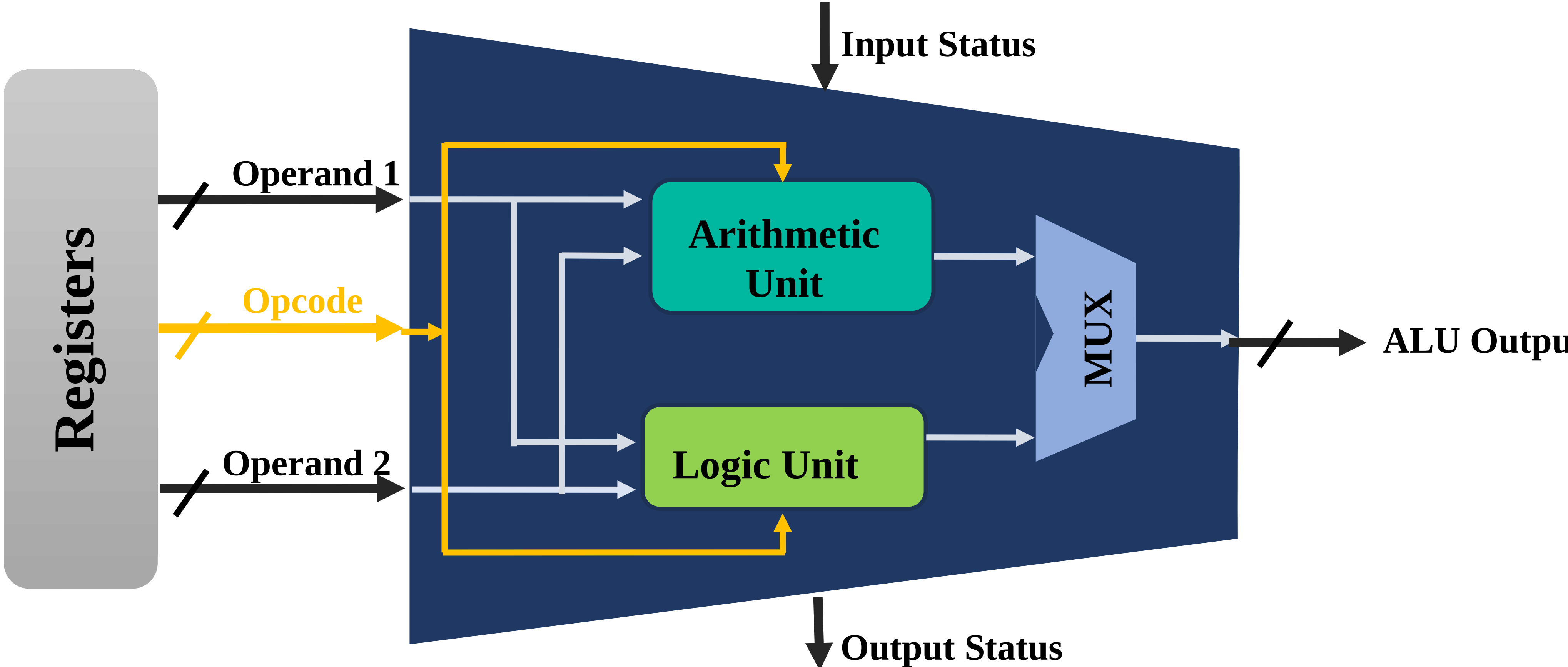
It is a fundamental building block of many types of computing circuits, including the central processing unit (CPU) of computers, FPUs, and graphics processing units (GPUs).

Functions of ALU:

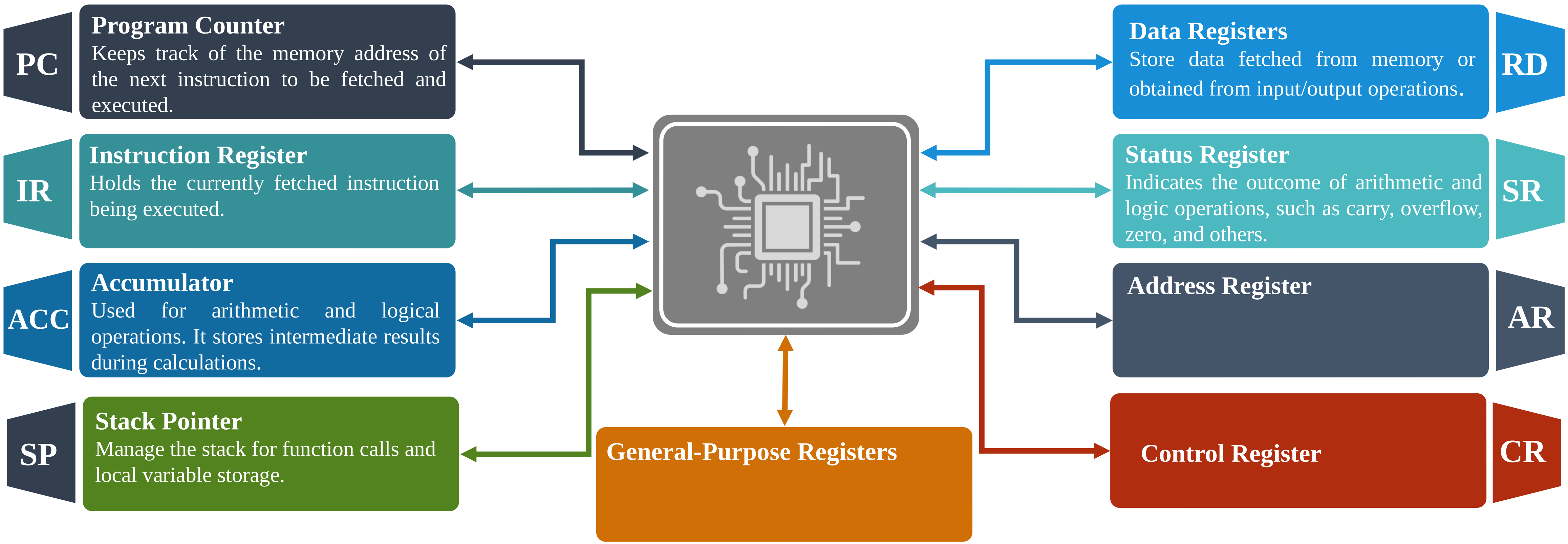
Basic Operations	Basic Instructions
Arithmetic operations	Addition, Subtraction, Multiplication, division
Logical operations	Logical Sum(OR), Logical Product(AND), Logical negation (NOT)
Comparison	Comparison Instruction (size compare)
Branch	Branch instructions to alter the instruction sequence based on conditions

Registers

- Registers are a type of computer memory built directly into the processor that is used to store and manipulate data during the execution of instructions.
- A register may hold an instruction, a storage address, or any kind of data (such as a bit sequence or individual characters).



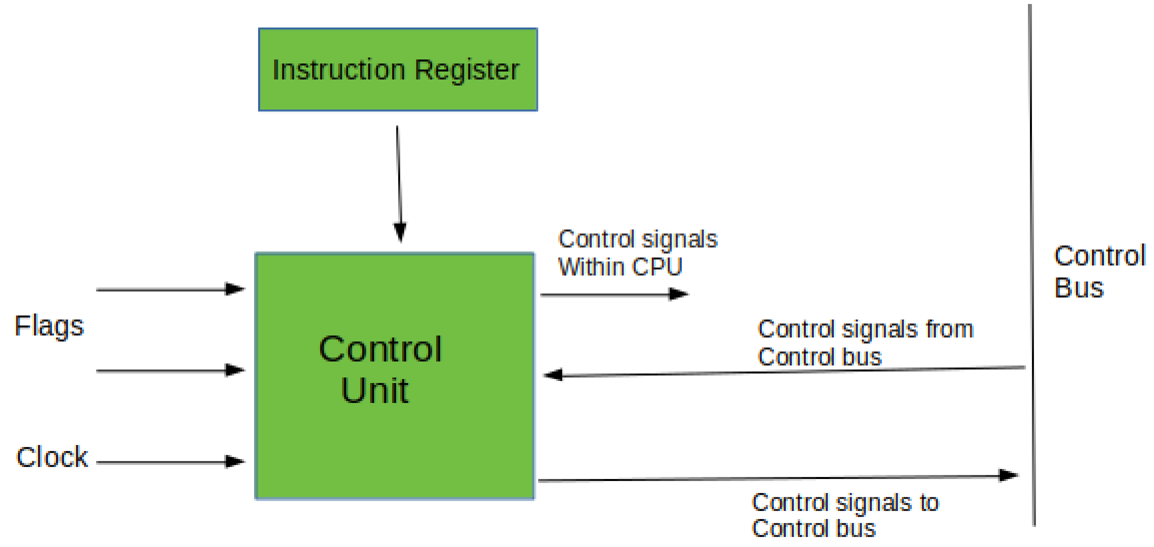
Registers in Processor Architecture



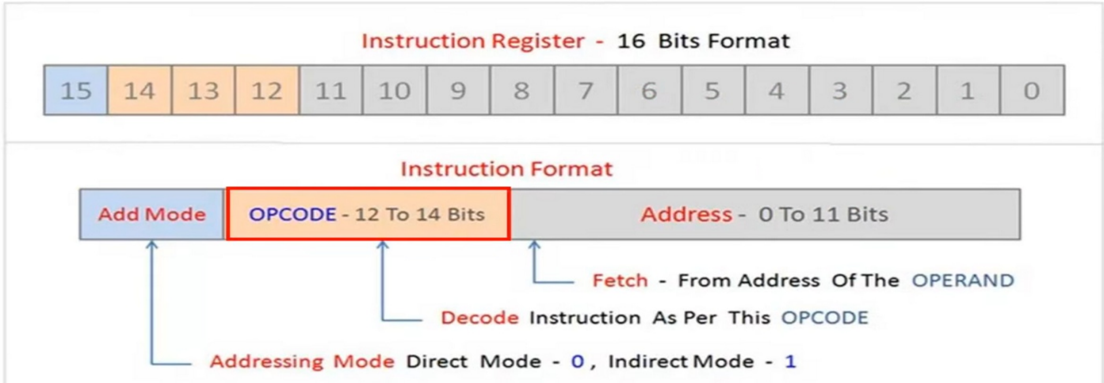
Control Unit:

The control unit controls all the operations of the processor. It retrieves, decodes and executes the code instructions one-by-one in the order they are stored in the main memory.

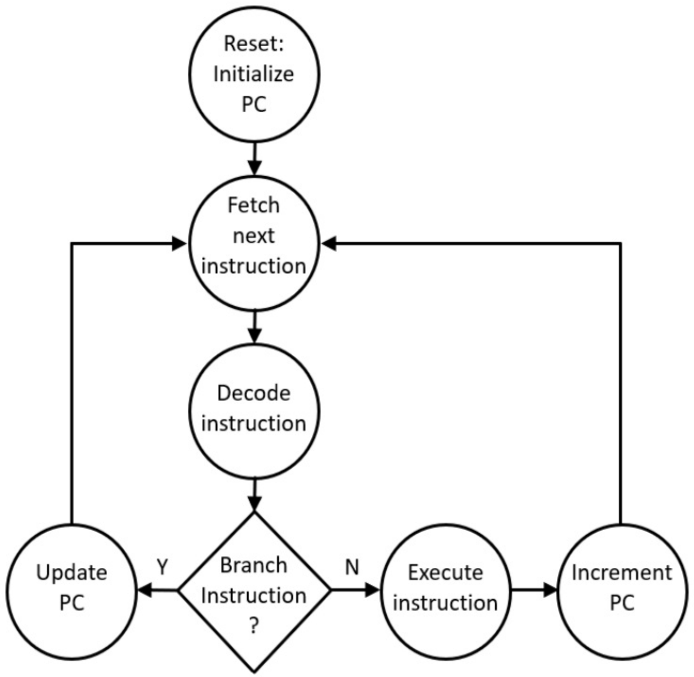
It instructs the arithmetic logic unit, memory, input/output devices how to respond to the instructions of the program.

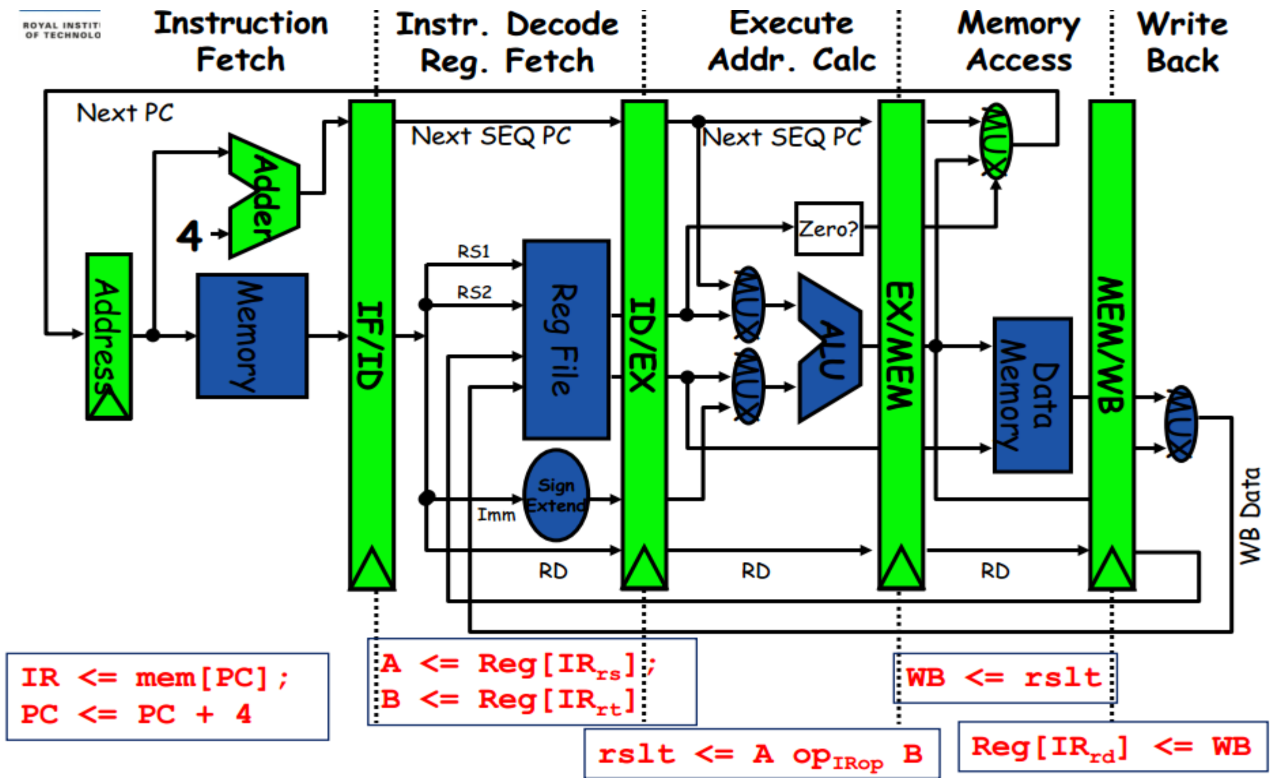


Block Diagram of the Control Unit



Stages: Execution Clock Cycles

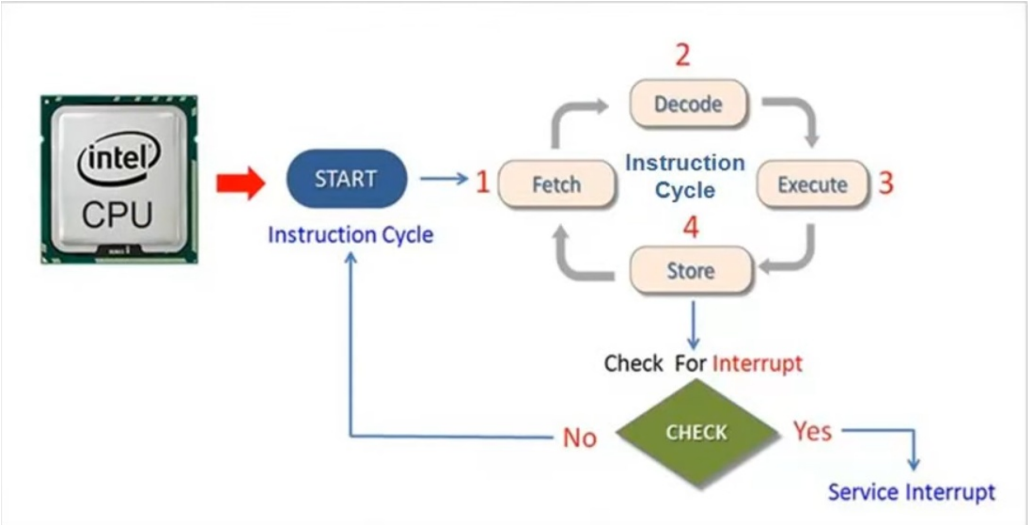


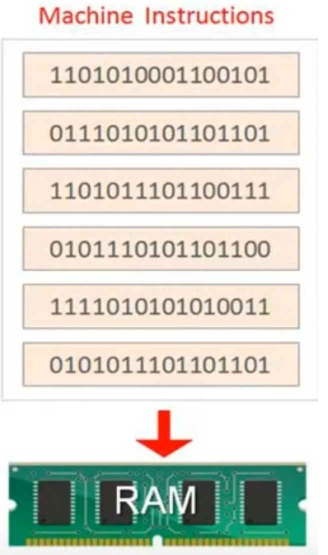


Instruction Set Architecture (ISA)

An **Instruction Set Architecture (ISA)** is part of the abstract model of a computer that defines how the CPU is controlled by the software.

- **The ISA acts as an interface between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done.**
- The ISA defines the supported data types, the registers, how the hardware manages main memory, key features (such as virtual memory), which instructions a microprocessor can execute, and the input/output model of multiple ISA implementations.
- Provides:
  - **Programmability**
  - **Flexibility**
  - **Reusablility**
  - **Adaptability**
  - **Accessibility**





31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Instruction Set Format

A form of representation of an instruction composed of fields of binary numbers.”

Fields of instruction:

There are several fields of the instruction that serve a specific role in the format. Some common are fields are given below:

1. **Opcode:**
   - Specifies the operation to be performed (e.g., add, subtract, load, store).
   - Determines what action the CPU should take.
2. **Operand:**
   - The data or the addresses of the data on which the operation is to be performed.
   - Can include immediate values, register addresses, or memory addresses.

3. Addressing Modes:

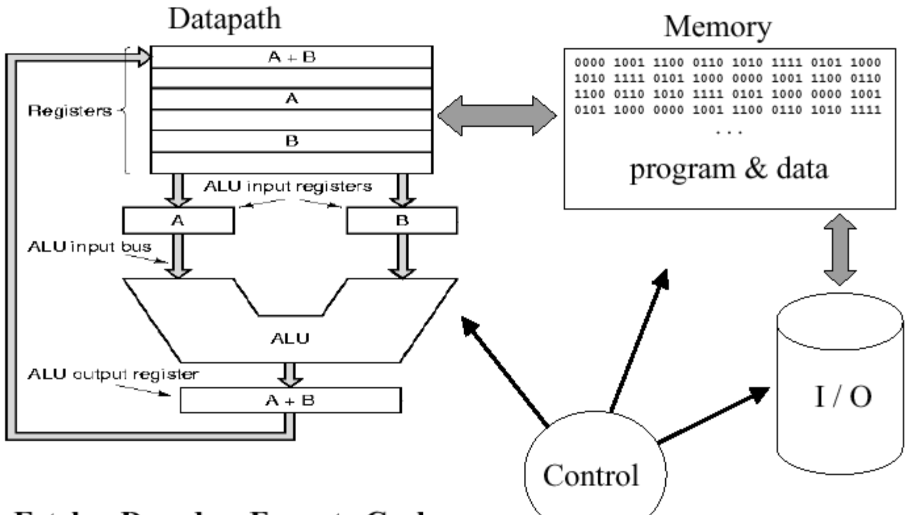
Processor uses different Addressing modes Common modes include:

immediate, direct, indirect, register, and indexed addressing.

4. Registers:

Specifies which CPU registers are to be used in the operation.

Could include source and destination registers.



Fetch – Decode – Execute Cycle

Instruction Types

A computer's instructions can be any length and have any number of addresses.

- The arrangement of a computer's registers determines the different address fields in the instruction format.
- The instruction can be classified as three, two, and one address instruction or zero address instruction, depending on the number of address fields.

Based on these differences the instructions are classified as

1) **Three Address Instruction**
2) **Two Address Instruction**
3) **One Address Instruction**
4) **Zero Address Instruction**

Three Address Instruction:

Three-address instruction is a format of machine instruction. It has one opcode and three address fields.

One address field is used for destination and two address fields for source.

OPCODE	DESTINATION	SOURCE 1	SOURCE 2

Example:

ADD	R1, A, B	R1 = M[A] + M[B]
ADD	R2, C, D	R2 = M[C] + M[D]
MUL	X, R1, R2	M[X] = R1 * R2

Two Address Instruction:

Two-address instruction is a format of machine instruction. It has one opcode and two address fields which may be memory locations or registers..

One address field is used for destination and one address field for source.

For example, a two-address instruction might add the contents of two registers together and store the result in one of the registers.

OPCODE	DESTINATION	SOURCE

Example :

MOV	R1, A	R1 = M[A]
ADD	R1, B	R1 = R1 + M[B]

One Address Instruction:

These instructions specify one operand or address, which typically refers to a memory location or register.

The instruction operates on the contents of that operand, and the result may be stored in the same or a different location.

For example, a one-address instruction might load the contents of a memory location into a register.

OPCODE	DESTINATION

Example:

STORE	T	M[T] = AC
LOAD	C	AC = M[C]

Zero Address Instruction:

These instructions do not specify any operands or addresses. Instead, they operate on data stored in registers or memory locations implicitly defined by the instruction.

For example, a zero-address instruction might simply add the contents of two registers together without specifying the register names.

Types of Instructions and Addressing Modes

Implied Mode

Example: CLC ; Clear the carry flag, no operands needed

Immediate Mode

Example: ADDI x1, x2, 10 ; Add immediate value 10 to register x2 and store result in x1

Register Mode

Example: MOV r0, r1 ; Move the contents of register r1 to register r0

Register Indirect Mode

Example: LW $t0, 0($t1) ; Load the word at the address in $t1 into $t0

Autodecrement Mode

Example: MOV -(R1), R0 ; Decrement R1 and then move the value at the new address in R1 to R0

Autoincrement Mode

Example: MOV (R1)+, R0 ; Move the value at the address in R1 to R0, then increment R1

Direct Address Mode

Example: LDA $4000 ; Load the accumulator with the value at memory address $4000

Indirect Address Mode

Example: JMP ($1234) ; Jump to the address stored at memory location $1234

Indexed Addressing Mode

Example: MOV AX, [BX+SI] ; Move the value at address (BX + SI) into AX

Important Parameters of a Processor

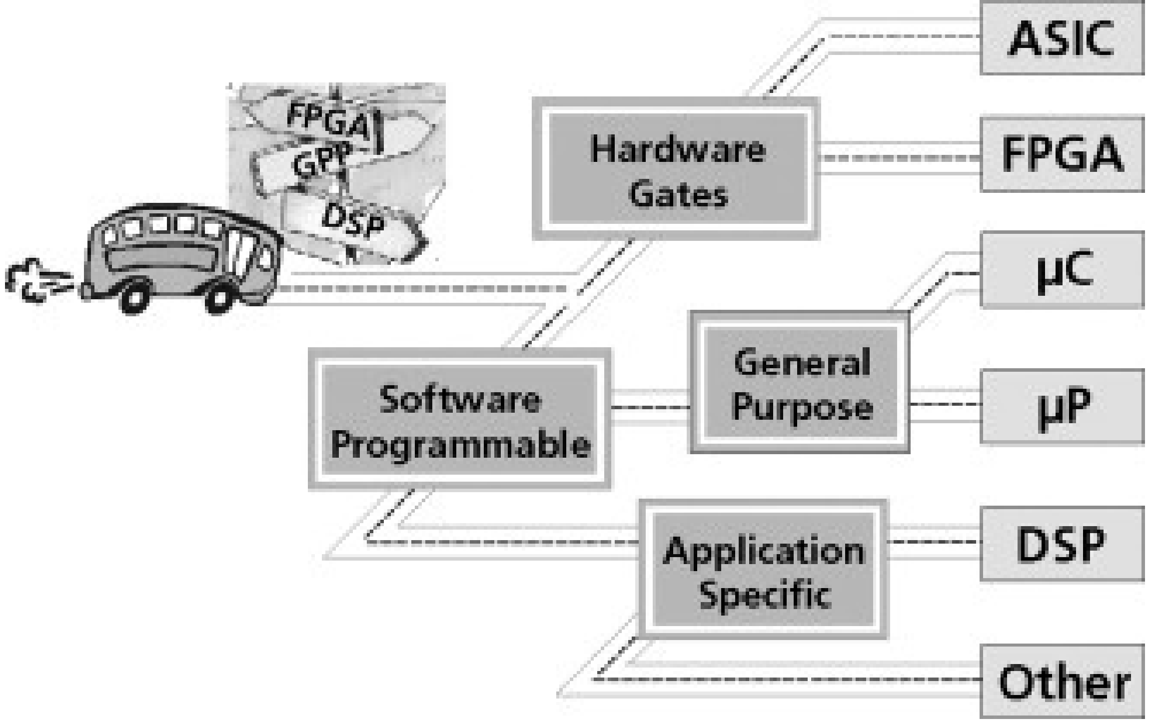
- Clock
- Data Bus
- Instruction Bus
- Instructions Per Cycles
- Pipeline Stage

Topics

1. Basic Processor Architecture
2. **Different Types of Processor Architectures**
3. RISC-V Processor Architecture
4. RISC-V Instruction Set Architecture
5. Programming RISC-V using assembly language

Processors Types

- General Purpose Processor
- Digital Signal Processor
- Vector Processor
- Application specific Processor



Flynn Taxonomy

- The matrix below defines the 4 possible classifications according to Flynn

S I S D Single Instruction, Single Data	S I M D Single Instruction, Multiple Data
M I S D Multiple Instruction, Single Data	M I M D Multiple Instruction, Multiple Data

Types of Processor ISA

Reduced Instruction Set Computing (RISC) vs Complex Instruction Set Computing (CISC)

Aspect	RISC	CISC
Instructions Per Cycle	Small and fixed length	Large and variable length
Instruction Complexity	Simple and standardised	Complex and versatile
Instruction Execution	Single clock cycle	Several clock cycles
RAM Usage	Heavy use of RAM	More efficient use of RAM
Memory	Increased memory usage to store instructions	Memory efficient coding
Cost	Cheaper than CISC	Higher

RISC vs CISC

The RISC approach has several advantages over CISC:

- **Simplifies Hardware Implementation:** It simplifies the hardware implementation of the processor, as fewer instructions need to be decoded and executed. This can lead to faster execution times and lower power consumption.
- **Higher Instruction Level Parallelism:** RISC processors typically have a higher instruction-level parallelism, allowing them to execute multiple instructions simultaneously, which can further improve performance.
- **Simplicity:** The simplicity of the RISC instruction set makes it easier to develop compilers and other software tools that can generate efficient code for the processor.

RISC vs **CISC**

RISC is a processor design philosophy that emphasizes simplicity and efficiency by using a small set of simple and general-purpose instructions.

- The ***complex instruction set computing*** (CISC), employs a larger set of more complex instructions that can perform multiple operations in a single instruction.
- RISC architectures prioritize simplicity and execute one instruction per clock cycle, resulting in streamlined designs and efficient decoding.
- CISC architectures, on the other hand, employ complex instructions capable of performing multiple actions but may require several clock cycles for execution. Both the CPUs aim to enhance CPU performance.

Single-purpose processors

Digital circuit designed to execute exactly one program

a.k.a. coprocessor, accelerator or peripheral

Features

Contains only the components needed to execute a single program

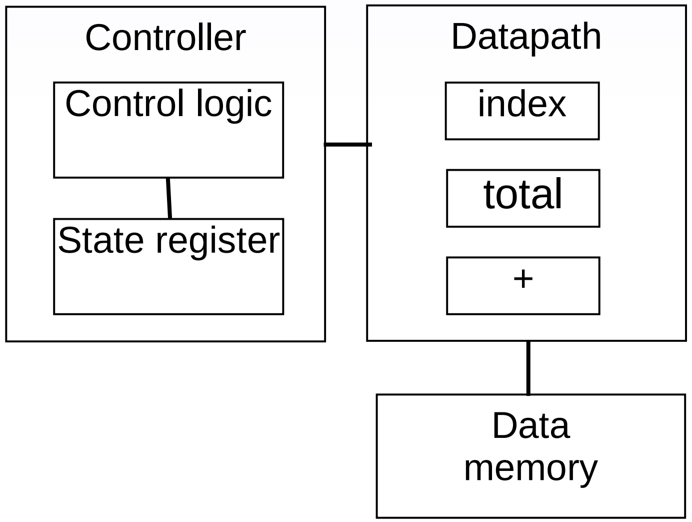
No program memory

Benefits

Fast

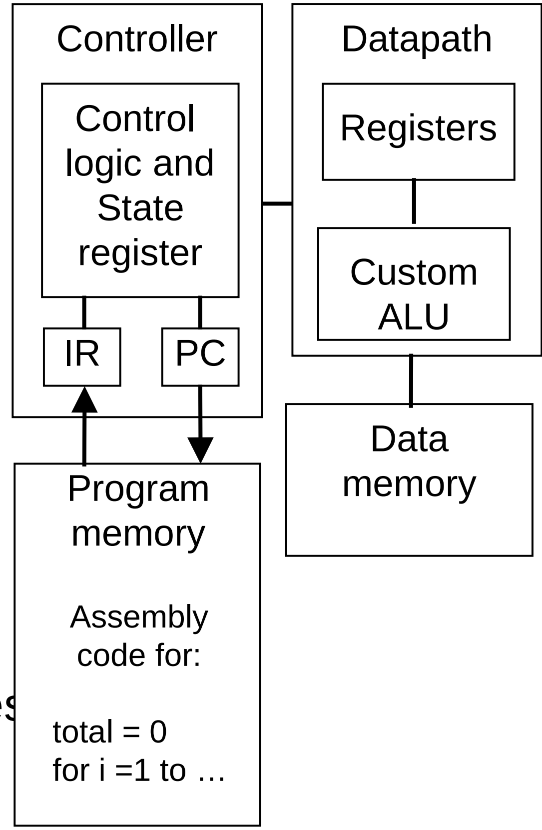
Low power

Small size



Embedded System Processor Architecture

- **Reduced Instruction Set Computing (RISC):**
  - Common architectures: ARM, RISC-V.
  - Simple, efficient instruction set optimized for low power and high performance.
- **System on Chip (SoC):**
  - Frequently used in embedded systems.
  - Integrates CPU, memory, peripherals, and other components on a single chip.
- **Microcontroller Units (MCUs):**
  - Often used in simpler embedded applications.
  - Includes integrated peripherals like ADCs, DACs, timers, and communication interfaces
- **Real-Time Capabilities:**
  - Designed for deterministic performance and real-time operating system (RTOS) support.
- **Low Power Consumption:**
  - Architectures and components optimized for minimal power usage.
- **Integrated Analog and Digital Peripherals:**
  - Features like GPIOs, serial communication interfaces, and specialized hardware accelerators.



Digital Signal Processor

Specialized Instruction Set:

Optimized for mathematical operations like multiply-accumulate (MAC).

Single-cycle multiply and MAC instructions.

Harvard Architecture:

Separate program and data memories to allow simultaneous access and increase throughput.

Specialized Data Path:

Multiple data buses and address buses.

Dedicated hardware for specific tasks such as FFT (Fast Fourier Transform) and filters.

High-Performance ALUs:

Multiple arithmetic logic units (ALUs) to perform parallel operations.

Support for fixed-point and floating-point arithmetic.

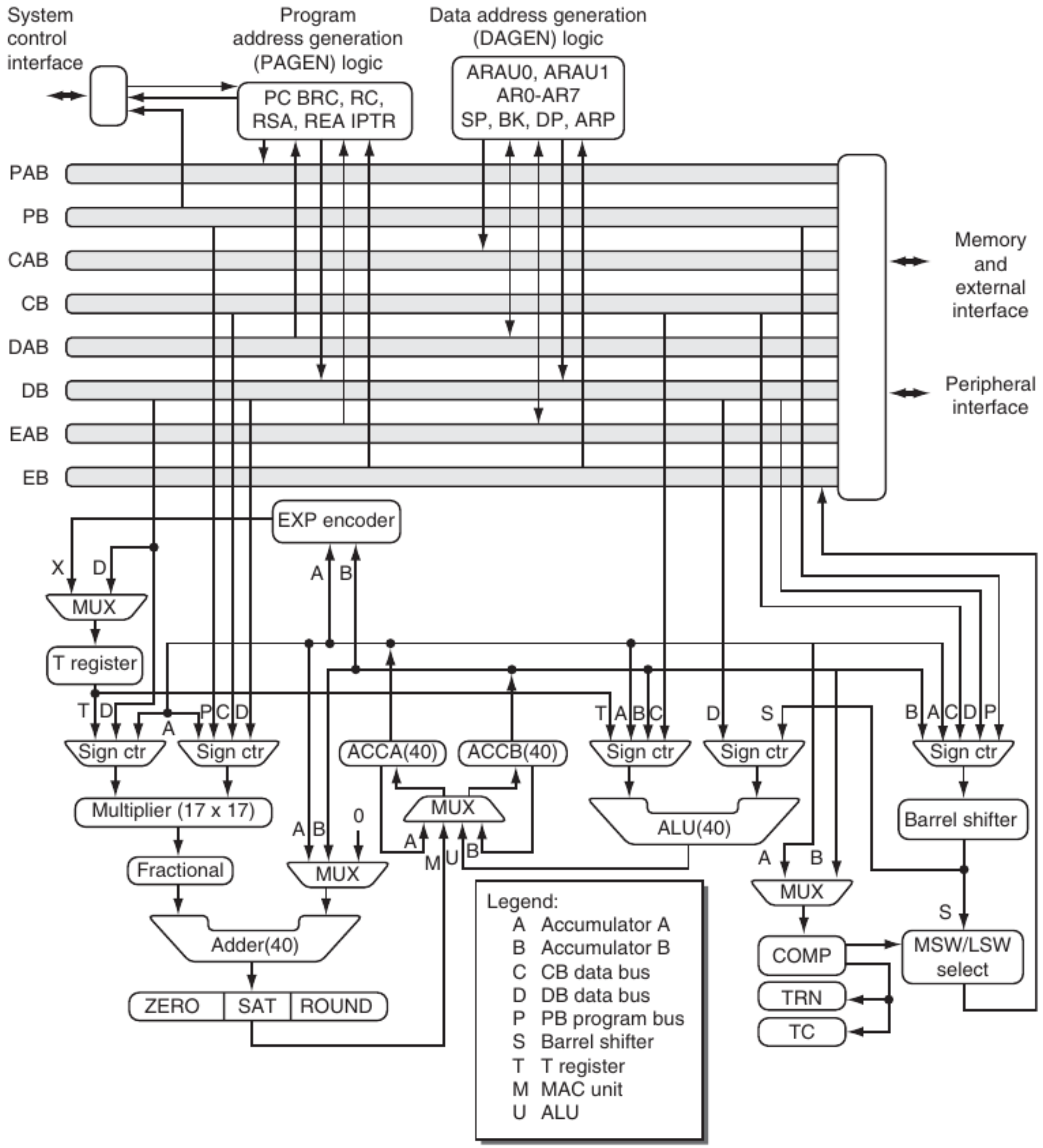
Circular and Bit-Reversed Addressing:

Efficiently manage circular buffers and data structures used in signal processing.

Low-Latency Memory Access:

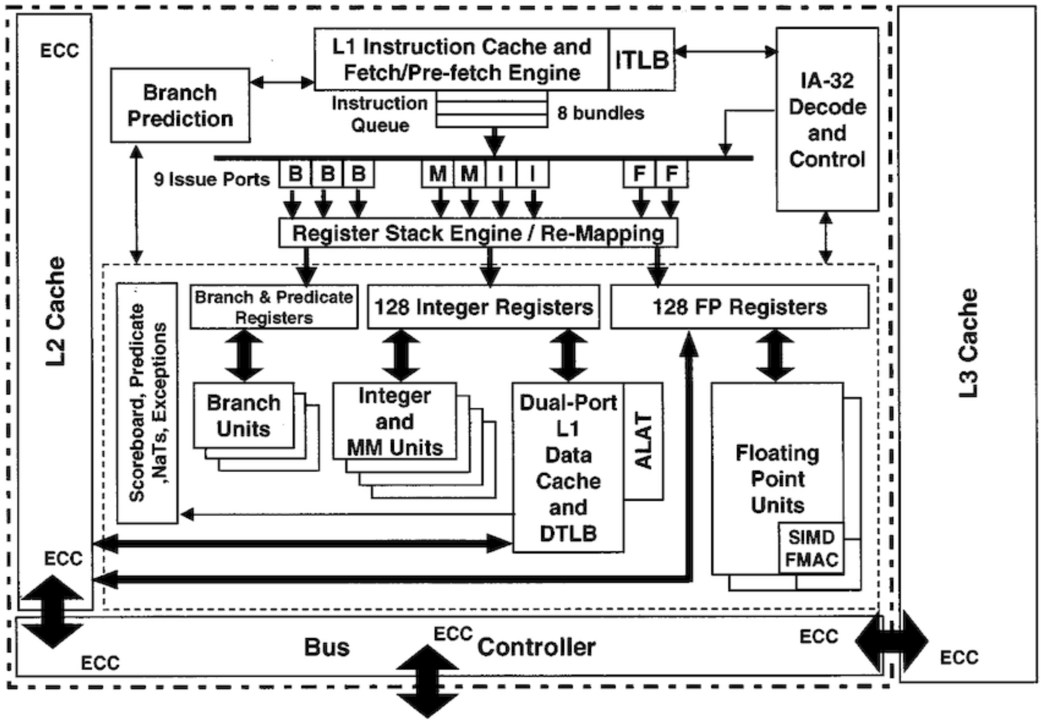
On-chip RAM with very low access latency.

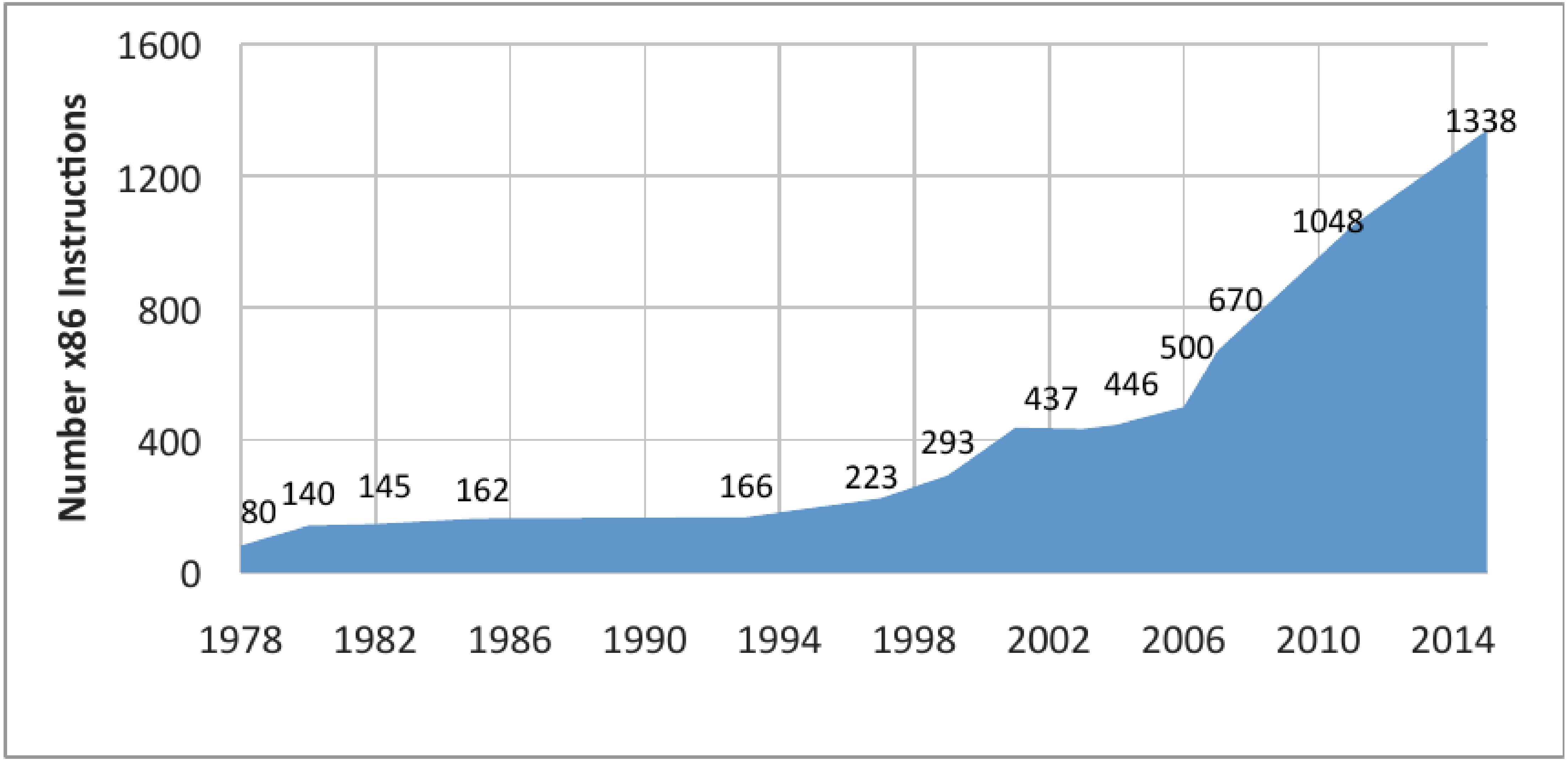
Multi-level cache hierarchy optimized for predictable access patterns.



General-Purpose Processor (GPP) Architecture

- **Complex Instruction Set Computing (CISC):**
  - Common architecture: x86.
  - Rich instruction set with complex instructions.
  - Often integrates many features directly in hardware.
- **Multi-Core and Hyper-Threading:**
  - Multiple cores for parallel processing.
  - Hyper-threading for improved performance through parallel execution within each core.
- **Large Cache Hierarchy:**
  - Multiple levels of cache (L1, L2, L3) to reduce latency and increase speed.
  - Advanced Branch Prediction and Speculative Execution:
  - Techniques to predict instruction paths and execute ahead to improve performance.
- **Integrated Memory Management Unit (MMU):**
  - Manages virtual memory, enabling sophisticated operating system features.
- **High-Speed Interconnects:**
  - Fast communication between CPU, memory, and peripherals.
- **Graphics Processing Unit (GPU) Integration:**
  - Some GPPs include integrated GPUs for handling graphics processing tasks.





Number x86 Instructions
1600
1200
800
400
0
80
140
145
162
166
223
293
437
446
500
670
1048
1338
1978
1982
1986
1990
1994
1998
2002
2006
2010
2014

Topics

1. Basic Processor Architecture
2. Different Types of Processor Architectures
3. **RISC-V Processor Architecture**
4. RISC-V Instruction Set Architecture
5. Programming RISC-V using assembly language

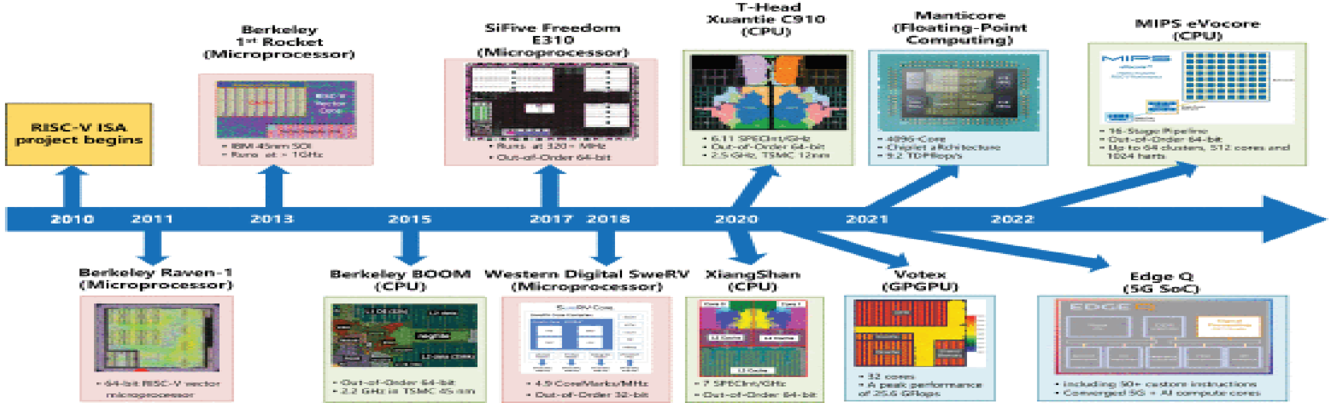
RISC-V Processor Architecture

The RISC-V (pronounced as risk-five) architecture is an open-source instruction set architecture (ISA) implementation of reduced instruction set computing RISC.

RISC-V is open-hardware architecture, its open source allows anyone to utilize the ISA.

History of RISC-V

- Prof. Krste Asanović and graduate students Yunsup Lee and Andrew Waterman started the RISC-V instruction set in May 2010 as part of the [Parallel Computing Laboratory]() (Par Lab) at UC Berkeley, of which Prof. David Patterson was Director.
- No patents were filed related to RISC-V in any of these projects, as the RISC-V ISA itself does not represent any new technology.
- RISC processor implementations—including some based on other open ISA standards— are widely available from various vendors worldwide.



Processor Architecture

Base Instruction Set

RV32I	Base Integer Instruction Set, 32-bit
RV32E	Base Integer Instruction Set (embedded), 32-bit
RV64I	Base Integer Instruction Set, 64-bit

Extension:

Name	Description
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point
Zicsr	Control and Status Register (CSR) Instructions
Zifencei	Instruction-Fetch Fence
G	Shorthand for the IMAFDZicsr_Zifencei base and extensions
C	Standard Extension for Compressed Instructions

Base and Extension of RISC-V

- Four base integer ISAs
  - RV32E, RV32I, RV64I, RV128I
  - RV32E is 16-register subset of RV32I
  - Only <50 hardware instructions needed for base
- Standard extensions
  - M: Integer multiply/divide
  - A: Atomic memory operations (AMOs + LR/SC)
  - F: Single-precision floating-point
  - D: Double-precision floating-point
  - G = IMAFD, "General-purpose" ISA
  - Q: Quad-precision floating-point
- All the above are a fairly standard RISC encoding in a fixed 32-bit instruction format
- Above user-level ISA components frozen in 2014
  - Supported forever after

RISCV: Registers and Mapping

RISC-V uses a memory-mapped I/O architecture, which means that input and output operations, memory access, and peripheral access are all performed using the same load and store instructions.

This unified approach simplifies the instruction set and enhances the flexibility and efficiency of the architecture. There are two basic types of instructions:

- Instructions that either load memory into registers or store data from registers into memory
- Instructions that perform arithmetical or logical operations between two registers

Why RISC-V

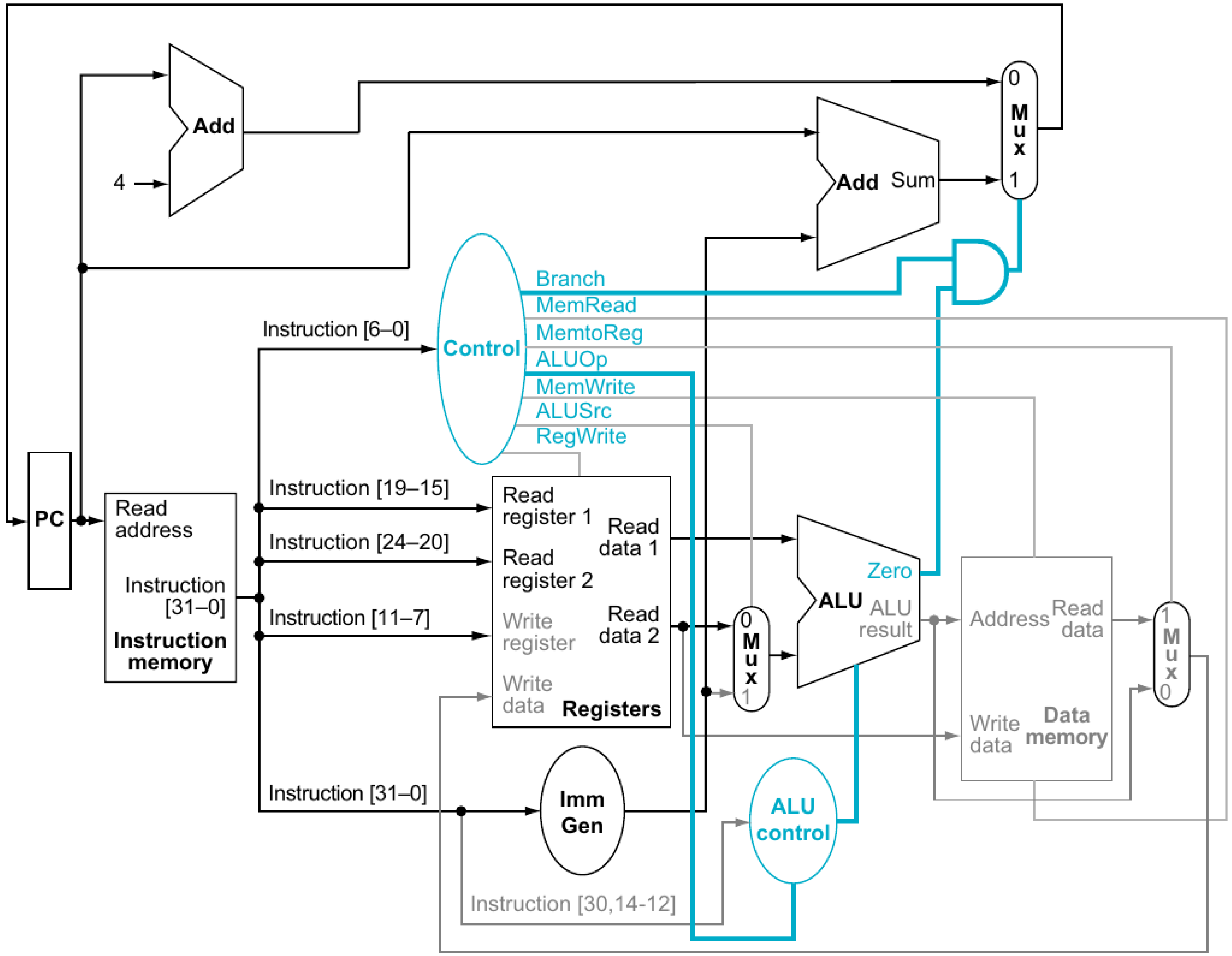
Open Hardware: Allowing anyone to design, implement, and customize processors without restrictions, fostering innovation and collaboration within the community.
Royalty-Free: There are no licensing fees, reducing costs for developers and manufacturers.
Security: Rigorous security analysis and the implementation of custom security features, enhancing trustworthiness.

ISA	Pages	Words	Hours to read	Weeks to read
RISC-V	236	76,702	6	0.2
ARM-32	2736	895,032	79	1.9
x86-32	2198	2,186,259	182	4.5

	Mistakes of the Past			Lessons learned
	ARM-32 (1986)	MIPS-32 (1986)	x86-32 (1978)	RV32I (2011)
Cost	Integer multiply mandatory	Integer multiply and divide mandatory	8-bit and 16-bit operations. Integer multiply and divide mandatory	No 8-bit and 16-bit operations. Integer multiply and divide optional (RV32M)
Simplicity	No zero register. Conditional instruction execution. Complex data address modes. Stack instructions (push/pop). Shift-option for arithmetic/logic instructions	Zero- and sign-extended immediates. Some arithmetic instructions can cause overflow traps	No zero register. Complex procedure call/return instructions (enter/leave). Stack instructions (push/pop). Complex data address modes. Loop instructions	Register x0 dedicated to 0. Immediates only sign-extended. One data addressing mode. No conditional execution. No complex call/return or stack instructions. No traps for arithmetic overflow. Separate shift instructions
Performance	Condition codes for branches. Source and destination registers vary in instruction format. Load multiple. Computed immediates. PC a general purpose register	Source and destination registers vary in instruction format.	Condition codes for branches. At most 2 registers per instruction	Compare and branch instructions (no condition codes). 3 registers per instruction. No load multiple. Source and destination registers fixed in instruction format. Constant immediates. PC not a general purpose register
Isolate architecture from implementation	Exposes the pipeline length when writing the PC as a general purpose register	Delayed branch. Delayed load. HI and LO registers just for multiply and divide	Registers not general purpose (AX, CX, DX, DI, SI have unique uses)	No delayed branch. No delayed load. General purpose registers
Room for growth	Limited available opcode space	Limited available opcode space		Generous available opcode space
Program size	Only 32-bit instructions (+Thumb-2 as separate ISA)	Only 32-bit instructions (+microMIPS as separate ISA)	Byte-variable instructions, but poor choices	32-bit instructions + 16-bit RV32C extension
Ease of programming / compiling / linking	Only 15 registers. Aligned data in memory. Irregular data address modes. Inconsistent performance counters	Aligned data in memory. Inconsistent performance counters	Only 8 registers. No PC-relative data addressing. Inconsistent performance counters	31 registers. Data can be unaligned. PC-relative data addressing. Symmetric data address mode. Performance counters defined in architecture

Types of RISC-V Processor Architectures

- RISC-V provides a detailed, open Instruction Set Architecture (ISA), which serves as a blueprint for designing processors architecture.
- Single-Cycle Architecture:
- Multi-Cycle Architecture:
- **Pipelined Architecture:**
- Superscalar Architecture:
- Out-of-Order Execution:
- Very Long Instruction Word (VLIW) Architecture:
- Vector Processing Architecture:
- Custom Instruction Set Extensions:



Add
4
Add
Sum
0
Mux
1
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [6–0]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [19–15]
Instruction [24–20]
Instruction [11–7]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
Instruction [31–0]
Imm Gen
ALU control
Instruction [30,14-12]

Defining/Designing RISC-V Processor Architecture

- Fetch: Retrieve the instruction from memory.
- Decode: Interpret the instruction and prepare operands.
- Execute: Perform the computation or operation (ALU operations, branches).
- Memory: Access memory for load/store operations.
- Writeback: Write the result to the register file or memory.

5 Stages of Processor Arch

- **Fetch Unit**

Function: Retrieves instructions from memory.
PC Usage: The PC holds the address of the next instruction to be fetched. After fetching an instruction, the PC is typically incremented to point to the next instruction address.
Example: If the starting address of the first instruction is 0x8000000, the Fetch Unit will fetch the instruction from address 0x8000000 initially.

- **Decode Unit**

Function: Interprets the fetched instruction to determine its operation and operands.
Memory Access: Decodes memory addresses and identifies whether they are for RAM, ROM, or I/O devices. It also decodes which registers are involved.
ALU: Determines the type of ALU operation required (e.g., addition, subtraction) and prepares operands for execution.
Example: Decodes an instruction to add two registers and prepare the operands for the ALU.

Execute Unit

Function: Performs the arithmetic or logical operations as specified by the instruction.

ALU: Executes ALU operations (e.g., addition, subtraction) using the operands provided by the Decode Unit.

Memory Access: Computes effective addresses for load/store operations.

Example: Executes an addition operation on two registers or calculates the address for a load instruction.

Memory Unit

Function: Accesses memory or I/O based on the address computed in the Execute stage.

Memory Access: Performs read/write operations to RAM or memory-mapped I/O devices based on the effective address.

Example: Reads data from address 0x00002000 in RAM or writes data to a memory-mapped I/O device at 0x20000000.

Write Back Unit

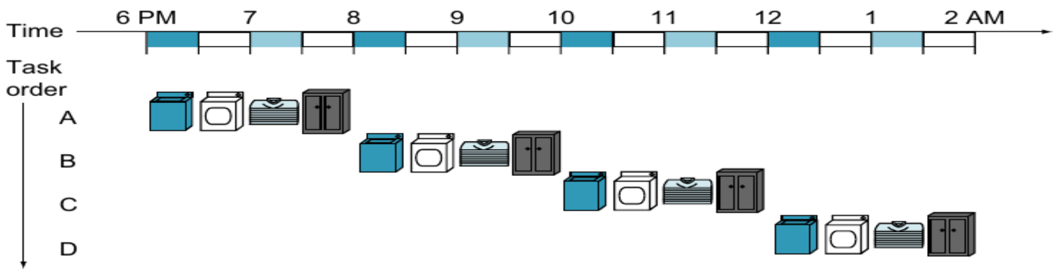
Function: Writes the result of computations or memory accesses back to the register file or memory.

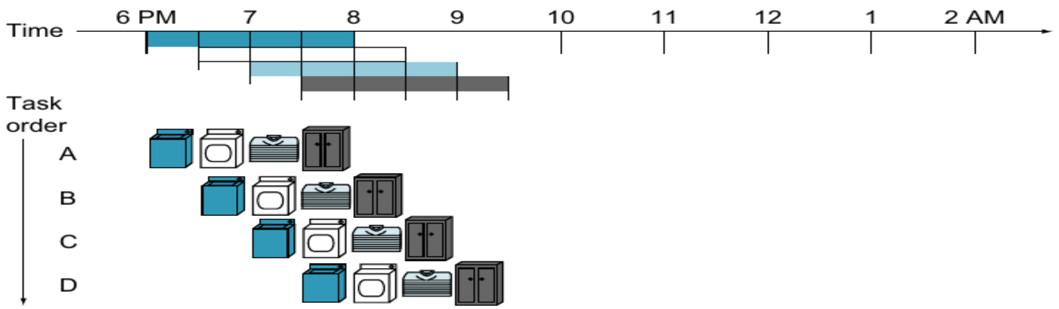
Memory Access: Updates the register file with results from the Memory Unit or ALU operations.

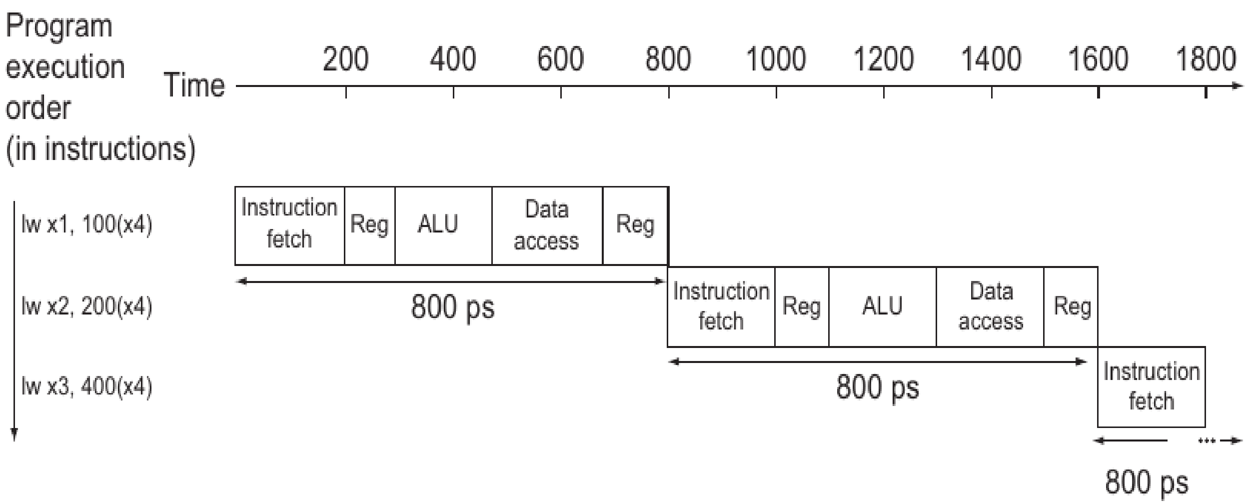
Example: Writes the result of an addition operation back to a register or stores data retrieved from memory to a register.

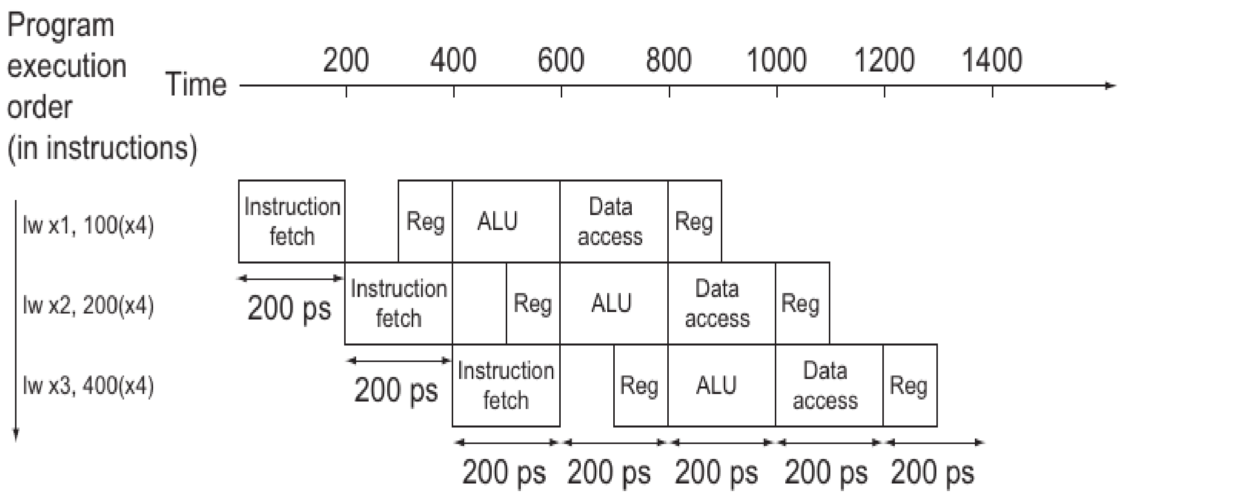
Pipe-lined VS Single Cycle Processor Architecture

1. Fetch instruction from memory.
2. Read registers and decode the instruction.
3. Execute the operation or calculate an address.
4. Access an operand in data memory (if necessary).
5. Write the result into a register (if necessary).

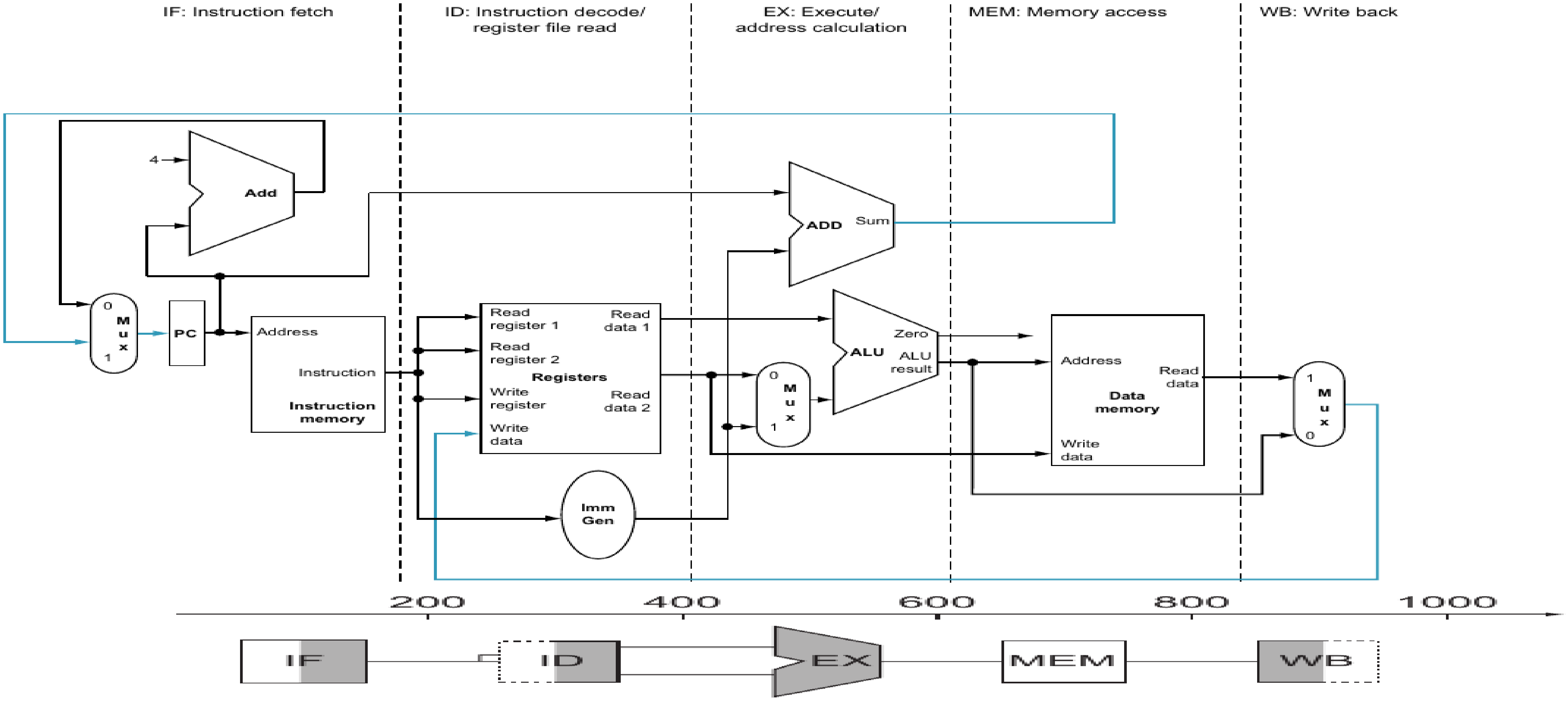








5 Stage Pipelined Processor Architecture



Topics

1. Basic Processor Architecture
2. Different Types of Processor Architectures
3. RISC-V Processor Architecture
4. **RISC-V Instruction Set Architecture**
5. Programming RISC-V using assembly language

RISCV Instructions Set

- RISC-V (Reduced Instruction Set Computing V) is an open standard instruction set architecture (ISA) that is designed to be scalable and extensible. The number of instructions in RISC-V can vary based on the specific subset or extensions of the ISA being used. Here's a breakdown of the primary RISC-V instruction sets and their respective instruction counts:
- Base ISA:
  - RV32I (32-bit Integer): The base integer instruction set for 32-bit processors includes approximately 47 instructions.
  - RV64I (64-bit Integer): The base integer instruction set for 64-bit processors extends RV32I and includes a few additional instructions specific to 64 bit operations.

Instruction Extensions

- Standard Extensions:
  - M (Multiply/Divide): Adds multiply and divide instructions.
  - A (Atomic): Adds atomic instructions for synchronization.
  - F (Single-Precision Floating-Point): Adds single-precision floating-point instructions.
  - D (Double-Precision Floating-Point): Adds double-precision floating-point instructions.
  - Q (Quad-Precision Floating-Point): Adds quad-precision floating-point instructions.
  - C (Compressed): Adds compressed instructions to reduce code size.

- Other Extensions:
  - B (Bit-Manipulation): Adds instructions for bit manipulation.
  - V (Vector): Adds vector processing instructions.
  - P (Packed-SIMD): Adds packed SIMD instructions.
  - Z (Various small extensions): These include specific sets of instructions like Zifencei for instruction-fence or Zicsr for control and status registers.

Basic RISCV Processor

- **The 47 standard instructions in RV32I include:**
  - Arithmetic Instructions: ADD, SUB, MUL, etc.
  - Logical Instructions: AND, OR, XOR, etc.
  - Immediate Instructions: ADDI, ORI, XORI, etc.
  - Load Instructions: LB, LH, LW, etc.
  - Store Instructions: SB, SH, SW, etc.
  - Branch Instructions: BEQ, BNE, BLT, etc.
  - Jumps: JAL, JALR
  - System Instructions: ECALL, EBREAK
  - Other Instructions: NOP, AUIPC, LUI, etc.

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred	succ	00000	000	00000	0001111	I fence
0000 0000	0000	00000	001	00000	0001111	I fence.i
000000000000		00000	000	00000	1110011	I ecall
000000000001		00000	000	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I csrrsi
csr		zimm	111	rd	1110011	I csrrci

Types of RISCV ISA

RISC-V Instruction Set:

The RISC-V instruction set is a collection of instructions that define the operations a RISC-V processor can perform.

These instructions are designed to be simple, efficient, and easily extensible, allowing for a high degree of customization and optimization.

Instruction Types:

1. **R-Type (Register Type):** Used for register-register arithmetic and logical operations.
2. **I- Type (Instruction Type):** Used for immediate arithmetic, load instructions, and register-immediate operations.
3. **S-Type (Store Type):** Used for store instructions.
4. **U-Type (Upper Immediate Type) :** Used for upper immediate instructions
5. **B-Type (Branch Type) :** Used for conditional branch instructions.
6. **J-Type (Jump Type) :** Used for jump instructions like JAL.
7. F-Type (Floating-Point) Instructions
8. A-Type (Atomic) Instructions
9. C-Type (Compressed) Instructions

Registers

- Total Registers: 32 general-purpose registers, additional special-purpose and control registers.
  - General Purpose Registers: x0 to x31, with specific roles for some registers.
  - Special Purpose Registers: Includes PC, SP, GP, TP.
    - Program Counter (PC): Holds the address of the current instruction being executed.
    - Instruction Register (IR): Holds the current instruction being executed (in some implementations).
    - Stack Pointer (SP): Points to the top of the stack.
    - Global Pointer (GP): Points to the global data region.
    - Thread Pointer (TP): Points to the thread-local storage.
  - Control and Status Registers: Includes MSR, MEPC, MCAUSE, MSTATUS, MTVEC.
    - Machine Status Register (MSR): Controls machine-level status and configuration.
    - Machine Exception Program Counter (MEPC): Holds the address of the instruction where an exception occurred.
    - Machine Cause Register (MCAUSE): Contains information about the cause of the last exception.
    - Machine Status Register (MSTATUS): Holds the status of the machine, including interrupts and mode.
    - Machine Trap Vector Base Address Register (MTVEC): Base address for the trap vector.
  - Floating-Point Registers: If included, f0 to f31 for floating-point operations.

Register name	Symbolic name	Description	Saved by
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

RISC-V Instruction Format

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/notebooks/RISCV/RISCV_CARD.pdf

RISC-V Instructions and Formats

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 ˆ 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
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ˆ 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

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]	
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
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	

RV32M Multiply Extension

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
mul	MUL	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	MUL High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
mulsu	MUL High (S) (U)	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulu	MUL High (U)	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]
div	DIV	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	DIV (U)	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder (U)	R	0110011	0x7	0x01	rd = rs1 % rs2

RV32A Atomic Extension

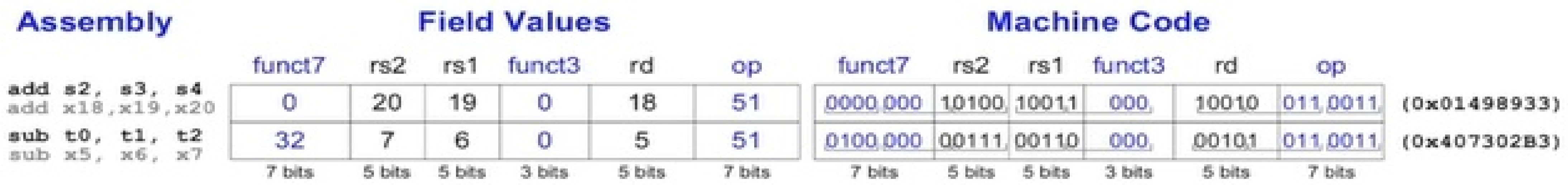
31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7

Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
lr.w	Load Reserved	R	0101111	0x2	0x02	rd = M[rs1], reserve M[rs1]
sc.w	Store Conditional	R	0101111	0x2	0x03	if (reserved) { M[rs1] = rs2; rd = 0 } else { rd = 1 }
amoswap.w	Atomic Swap	R	0101111	0x2	0x01	rd = M[rs1]; swap(rd, rs2); M[rs1] = rd
amoadd.w	Atomic ADD	R	0101111	0x2	0x00	rd = M[rs1] + rs2; M[rs1] = rd
amoand.w	Atomic AND	R	0101111	0x2	0x0C	rd = M[rs1] & rs2; M[rs1] = rd
amoor.w	Atomic OR	R	0101111	0x2	0x0A	rd = M[rs1] | rs2; M[rs1] = rd
amoxor.w	Atomix XOR	R	0101111	0x2	0x04	rd = M[rs1] ^ rs2; M[rs1] = rd
amomax.w	Atomic MAX	R	0101111	0x2	0x14	rd = max(M[rs1], rs2); M[rs1] = rd
amomin.w	Atomic MIN	R	0101111	0x2	0x10	rd = min(M[rs1], rs2); M[rs1] = rd

RISC-V Instruction Format: 1: R-Type

31 25	24 20	19 15	14 12	11 7	6 0
func7	rs2	rs1	func3	rd	opcode

Field	No. O f bits	Function
opcode:	R1, A, B	Basic operation of the instruction, and this abbreviation is its traditional name.
oprand:	R2, C, D	The register destination operand. It gets the result of the operation
funct3:	X, R1, R2	An additional opcode field.
rs1:		The first register source operand.
rs2:		The second register source operand.
func7		An additional opcode field.

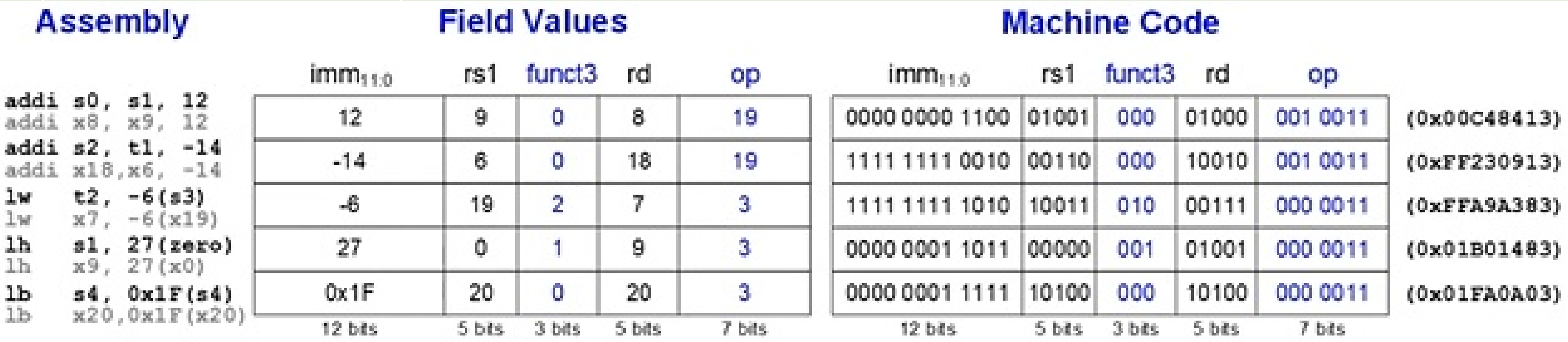


Assembly	Field Values: funct7	rs2	rs1	funct3	rd	op	Machine Code: funct7	rs2	rs1	funct3	rd	op	
add s2, s3, s4 add x18,x19,x20	0	20	19	0	18	51	0000 000	10100	10011	000	10010	011 0011	(0x01498933)
sub t0, t1, t2 sub x5, x6, x7	32	7	6	0	5	51	0100 000	00111	00110	000	00101	011 0011	(0x407302B3)
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Immediate: I-Type

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	func3	rd	opcode

Field	No. O f bits	Function
opcode:	R1, A, B	Basic operation of the instruction, and this abbreviation is its traditional name.
rd:	R2, C, D	The register destination operand. It gets the result of the operation
funct3:	X, R1, R2	An additional opcode field.
rs1:		The first register source operand.
imm		The second register source operand.



Assembly		Field Values: $imm_{11:0}$	rs1	funct3	rd	op	Machine Code: $imm_{11:0}$	rs1	funct3	rd	op	
addi s0, s1, 12	addi x8, x9, 12	12	9	0	8	19	0000 0000 1100	01001	000	01000	001 0011	(0x00C48413)
addi s2, t1, -14	addi x18,x6, -14	-14	6	0	18	19	1111 1111 0010	00110	000	10010	001 0011	(0xFF230913)
lw t2, -6(s3)	lw x7, -6(x19)	-6	19	2	7	3	1111 1111 1010	10011	010	00111	000 0011	(0xFFA9A383)
lh s1, 27(zero)	lh x9, 27(x0)	27	0	1	9	3	0000 0001 1011	00000	001	01001	000 0011	(0x01B01483)
lb s4, 0x1F(s4)	lb x20,0x1F(x20)	0x1F	20	0	20	3	0000 0001 1111	10100	000	10100	000 0011	(0x01FA0A03)
		12 bits	5 bits	3 bits	5 bits	7 bits	12 bits	5 bits	3 bits	5 bits	7 bits	

Store: S-Type

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode

i	No. O f bits	Function
opcode:	R1, A, B	Basic operation of the instruction, and this abbreviation is its traditional name.
rd:	R2, C, D	The register destination operand. It gets the result of the operation
imm[4:0]:	X, R1, R2	An additional opcode field.
rs1:		The first register source operand.
rs2:		The second register source operand.
imm[11:5]		An additional opcode field.

#x1 based address
SW x2, 0(x1) # Memory[x1 + 0] = x2

Branch: B-Type

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode

i	No. O f bits	Function
opcode:	R1, A, B	Basic operation of the instruction, and this abbreviation is its traditional name.
rd:	R2, C, D	The register destination operand. It gets the result of the operation
imm[4:0]:	X, R1, R2	An additional opcode field.
rs1:		The first register source operand.
rs2:		The second register source operand.
imm[11:5]		An additional opcode field.

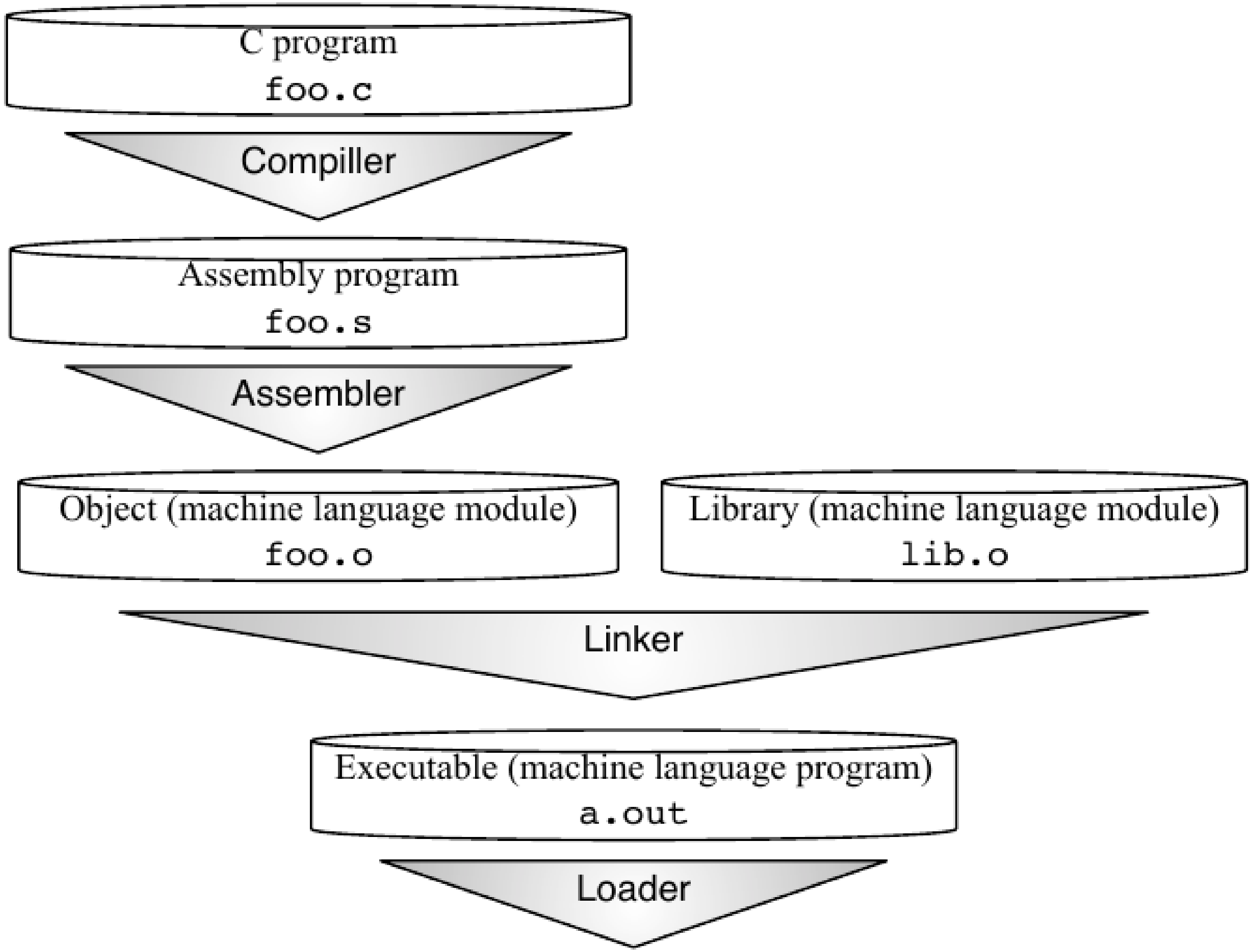
BEQ x1, x2, equal # If x1 == x2, branch to label 'equal'

Jump: J-Type

- Unconditional jump with an optional link to store the return address.
  - opcode (7 bits): Operation code that specifies the jump instruction (e.g., JAL).
  - rd (5 bits): Destination register for the return address.
  - immediate (20 bits): Jump target offset, which is used to calculate the jump address relative to the current program counter (PC).

Topics

1. Basic Processor Architecture
2. Different Types of Processor Architectures
3. RISC-V Processor Architecture
4. RISC-V Instruction Set Architecture
5. **Programming RISC-V using assembly language**



C program
foo.c
Compiller
Assembly program
foo.s
Assembler
Object (machine language module)
foo.o
Library (machine language module)
lib.o
Linker
Executable (machine language program)
a.out
Loader

RISCV GCC Assembler

```
_start:
ld s3 0x001121
Ld rs2 0x0022233
add rd, rs1, rs2
St rd 0x0000001
```

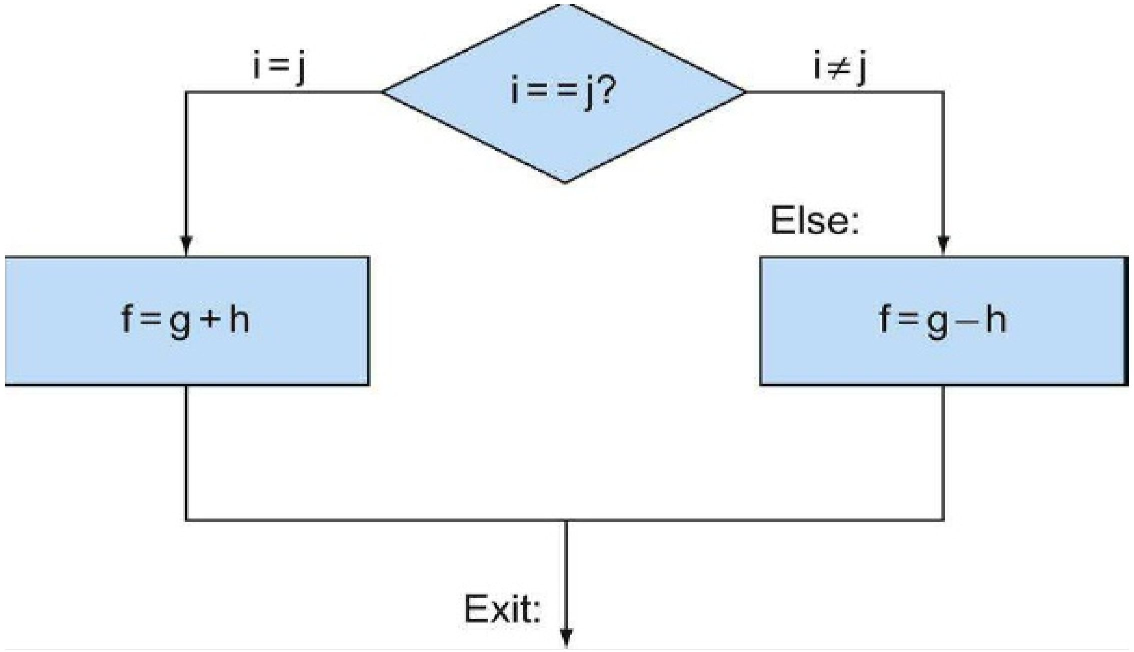
Assembly or C/C++

- Write Efficient Code
- Secure Application
- Multi-Threaded and Complex Program to run multiple devices (OS)
- Real-Time Applications for Real world Problems

Programming RISC-V

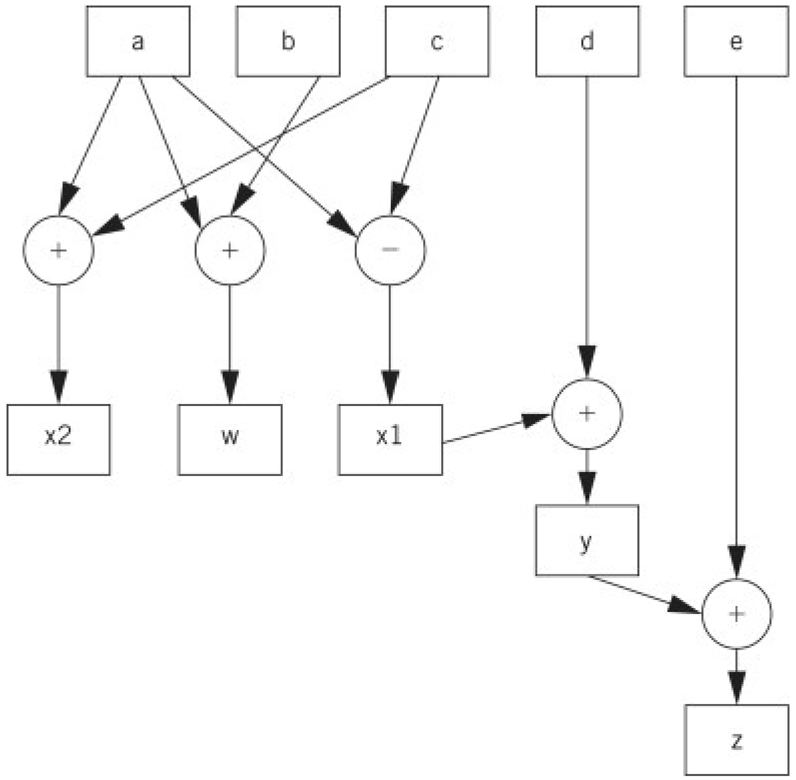
- Problem
- Write it in your own words
- Make Pseudo Code
- Create Control and Data-flow Graph
- Program (C/C++, ASM)
- Debug
- Profile
- Optimize/Fine Tune
- Execute
- Test

Flowchart



Hazards

- Data Hazards: Instructions are waiting for data from other instructions.
- Control Hazards: Changes in instruction flow cause delays.
- Structural Hazards: Limited hardware resources cause delays.



```c
// example.c
int global_var = 10;
int main() {
    int local_var = 5;
    int result = global_var + local_var;
    return result;
}
```

```
riscv32-unknown-elf-gcc example.o -o example
```

- The compiler generates an object file in ELF format. This object file contains machine code, data, and metadata, organized into different sections like .text (code), .data (initialized data), and .bss (uninitialized data).

- Instruction Section: Contains the compiled machine code instructions (text section).
- Data Section: Contains initialized data (data section).
- The linker combines the code and data sections, resolves symbols, and sets up memory addresses.
- The linker script defines how different sections are mapped into the memory of the microcontroller.
- It specifies memory regions and assigns addresses to different sections of the code and data.

```
MEMORY
{
    ROM (rx) : ORIGIN = 0x08000000, LENGTH = 512K
    RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 64K
}

SECTIONS
{
    .text : {
        *(.text)
    } > ROM

    .data : {
        *(.data)
    } > RAM
}
```

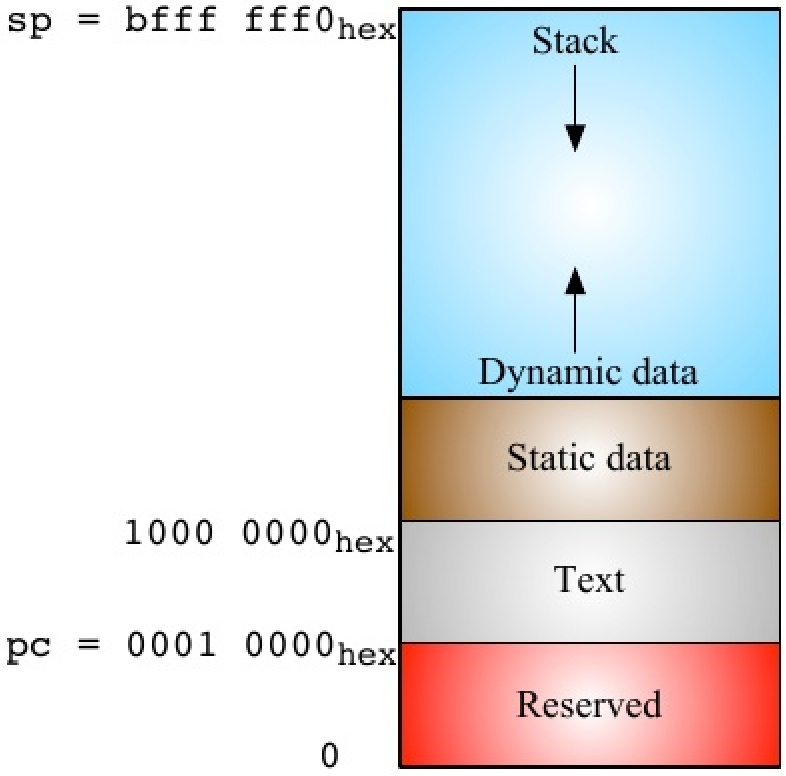
- Next step involves using a programmer or debugger tool to flash the firmware into the RISCV System.
  - Instruction Memory: The code from the .text section is loaded into the system instruction memory.
  - Data Memory: The initialized data from the .data section is loaded into the system data memory.

Linker Script: Program and Data Memory Allocation

The high addresses are the top of the figure and the low addresses are the bottom.

The stack pointer (sp) starts at BFFF FFF0 hex and grows down toward the Static data. The text (program code) starts at 0001 0000hex and includes the statically-linked libraries.

The Static data starts immediately above the text region; in this example, we assume that address is 1000 0000hex . Dynamic data, allocated in C by malloc(), is just above the Static data. Called the heap, it grows upward toward the stack. It includes the dynamically-linked libraries.



Testing and Executing the Code

RIPES

https://ripes.me/

https://github.com/mortbopet/Ripes/releases/download/v2.2.6/Ripes-v2.2.6-win-x86_64.zip

Next:

RISCV Micro Controller

RISCV Simulator and Emulators

RISCV Single Board Computer

Center of Excellence: Supercomputing for AI & Big-Data

RISC-V Processor Hardware and Instruction Set Architecture

by: Tassadaq Hussain

Director Centre for AI and BigData

Professor Department of Electrical Engineering

Namal University Mianwali

Collaborations:

Barcelona Supercomputing Center, Spain

European Network on High Performance and Embedded Architecture and Compilation

Pakistan Supercomputing Center