What is RISC-V?

Learn RISC-V

Prerequisites

This lecture is best suited to learners with the following:

- Familiarity with digital logic design principles: Boolean algebra, logic gates, and circuit design.
- Understanding of the fundamental concepts of computer architecture and organization; (processors, memory, and I/Os).
- Basic knowledge of assembly language.

Common acronyms

- RISC-V: Reduced Instruction Set Computer Five
- ISA: Instruction Set Architecture
- RV32: 32-bit RISC-V implementation
- RV64: 64-bit RISC-V implementation
- RV128: 128-bit RISC-V implementation
- M - Multiplication
- A - Atomics
- F - Floating point (32-bit)
- D - Floating point (64-bit)
- Q - Floating point (128-bit)
- C - Compressed instructions (16-bit)
- J - Interpreted or JIT-compiled languages support

Outline

- What is RISC-V?
- What is an Instruction Set Architecture?
- Why use RISC-V?
- RISC-V base and extensions
- RISC-V Registers
- 32-bit RISC-V instruction formats
- Classes of instructions
- RISC-V Implementations

What is RISC-V?

RISC-V is an open standard instruction set architecture (ISA) based on reduced instruction set computer principles.

The "V" in RISC-V signifies that it is the fifth major version of the RISC architecture developed at the University of California, Berkeley, where the project originated.

RISC-V history

The RISC-V ISA was developed at the University of California, Berkeley. The project began in May 2010, intending to create an open, royalty-free ISA that anyone could use to build processors.

The first RISC-V processors were fabricated in 2011, and the first commercial RISC-V chips were released in 2016.

What is an instruction set architecture?

An Instruction Set Architecture is the set of specifications and interface between the hardware components (processor) and the software components (compiler, operating system, etc.) of a computer system.

It defines the set of instructions that a processor can execute, along with the encoding, semantics, and behavior of those instructions.

What is an instruction set architecture?

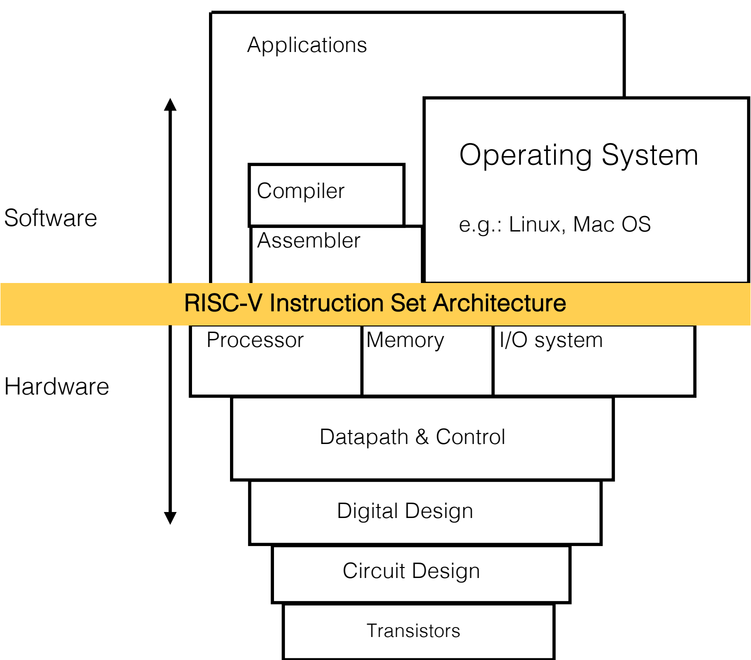
The instruction set architecture includes the following features:

- **Instruction formats**: The structure and encoding of instructions that the processor can execute.
- **Data types**: The types of data that the instructions can process, e.g., integers, floating-point numbers, and vectors.
- **Memory addressing modes**: The mechanisms for accessing memory, e.g., direct addressing, indirect addressing, and indexed addressing.
- **Register set**: The available registers in the processor and their purposes, e.g., general-purpose registers, special-purpose registers, and control registers.
- **Control flow**: The instructions and mechanisms for controlling the flow of program execution, including branches, jumps, and subroutine calls.

Why use RISC-V?

- Open-source: RISC-V specifications are freely available to the public
- License-Free: RISC-V is not encumbered by licensing fees or intellectual property restrictions.
- Extensible: RISC-V has a standard base ISA with multiple standard extensions.
- Flexible: RISC-V is modular for design space exploration with custom instructions.
- Stable: The base standard extensions are frozen.
- OS ready: Linux support is available.
- Growth: Significant industry support from both established technology companies and startups.

Overview of a computer system



RISC-V base and extensions

RISC-V has a modular design consisting of base variants and optional extensions.

- The base ISA specifies instructions, encoding, control flow, registers, memory, and addressing.
- RISC-V extensions provide a way to add specific features and capabilities to RISC-V processors without having to modify the base instruction set. e.g.:
  - Compressed instructions
  - Vector operations
  - Security extensions
  - Cryptographic extensions
  - etc.

RISC-V base and common extensions

Name	Description	Version	Status	Instruction count
Base				
RVWMO	Weak Memory Ordering	2.0	Ratified	
RV32I	Base Integer Instruction Set, 32-bit	2.1	Ratified	40
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers	2.0	Ratified	40
RV64I	Base Integer Instruction Set, 64-bit	2.1	Ratified	15
RV64E	Base Integer Instruction Set (embedded), 64-bit	2.0	Ratified	
RV128I	Base Integer Instruction Set, 128-bit	1.7	Open	15
Extension				
M	Standard Extension for Integer Multiplication and Division	2.0	Ratified	8 (RV32) 13 (RV64)
A	Standard Extension for Atomic Instructions	2.1	Ratified	11 (RV32) 22 (RV64)
F	Standard Extension for Single-Precision Floating-Point	2.2	Ratified	26 (RV32) 30 (RV64)
D	Standard Extension for Double-Precision Floating-Point	2.2	Ratified	26 (RV32) 32 (RV64)
More extensions …				

Tip

RISC-V defines an exact order that must be used to define the RISC-V ISA subset.

RV [32, 64, 128] I, M, A, F, D, G, Q, L, C, B, J, T, P, V, N

e.g.

- RV32IMAFDQC is legal
- RV32IMAFDCQ is not legal

RISC-V Registers

32 integer registers

- x0, x1, …, x31

32 floating point registers

- f0, f1, …, f31

1 Program Counter register

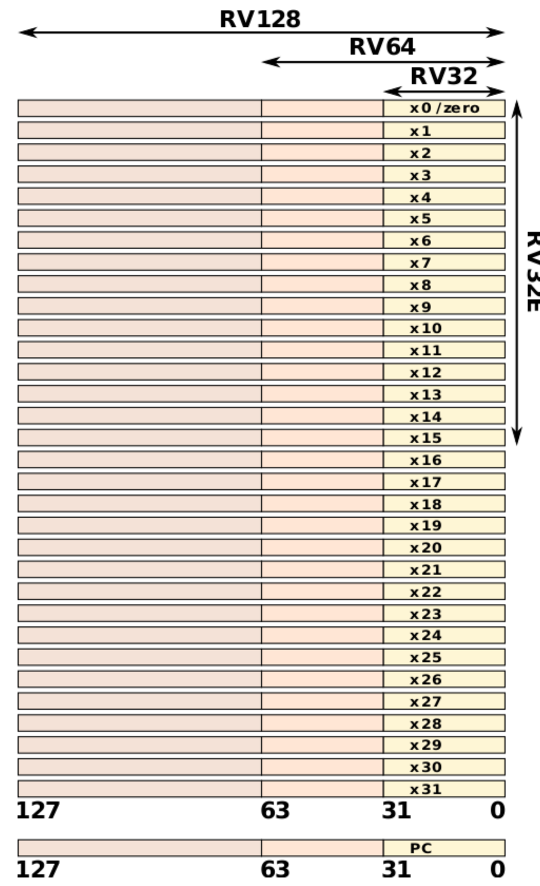
- 32-bit

Control Status Registers

- User-mode
- Machine-mode
- Custom

RISC-V Registers

Register name	ABI 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
32 floating-point extension registers			
f0–7	ft0–7	Floating-point temporaries	Caller
f8–9	fs0–1	Floating-point saved registers	Callee
f10–11	fa0–1	Floating-point arguments/return values	Caller
f12–17	fa2–7	Floating-point arguments	Caller
f18–27	fs2–11	Floating-point saved registers	Callee
f28–31	ft8–11	Floating-point temporaries	Caller



Tip

The ABI (Application Binary Interface), describes how registers are used by programs.

- Registers that start with the letter t (temporary) can be used for any purpose.
- Registers starting with the letter a (argument) are used for arguments passed to a function.
- Registers starting with the letter s (saved) are registers that are preserved across function calls, (except sp).

Examples: integer registers usage

`lb t0, 8(sp)`	Load a byte from the memory address (sp + 8) into register t0.
`sb t0, 8(sp)`	Store a byte from register t0 into memory address (sp + 8).
`add a0, t0, t1`	Add the value in t0 to the value in t1 and store the sum in a0.
`addi a0, t0, -10`	Add the value in t0 to the value -10 and store the sum in a0.
`sub a0, t0, t1`	Subtract the value in t1 from value in t0 and store the difference in a0.
`mul a0, t0, t1`	Multiply the value in t0 with the value in t1 and store the product in a0.
`div a1, s3, t3`	Divide the value in s3 (numerator) by the value in t3 (denominator) and store the quotient in a1.
`rem a1, s3, t3`	Divide the value in s3 (numerator) by the value in t3 (denominator) and store the remainder in a1.
`and a3, t3, s3`	Perform the logical AND of t3 and s3 and store the result in a3.
`or a3, t3, s3`	Perform the logical OR of t3 and s3 and store the result in a3.
`xor a3, t3, s3`	Perform the logical XOR of t3 and s3 and store the result in a3.

Examples: floating point registers usage

`flw ft0, 0(sp)`	Load in register ft0 the value at memory + 0
`flw ft1, 4(sp)`	Load in register ft1 the value at memory + 4
`fadd.s  ft2, ft0, ft1`	Add the value in ft0 to the value ft1 and store the sum in ft2. Note: the fadd.s instruction to tell the RISC-V processor to add two single-precision values (ft0 and ft1) and store it as a single precision value into ft2.

Tip

RISC-V is a reduced instruction set with a limited set of simple and optimized instructions.

- There are no instructions to save and restore multiple registers.
- Many instructions that can be completed by using another instruction are left off.

  e.g., the neg a0, a1 (two's complement) instruction does not exist. However, this is equivalent to sub a0, zero, a1. In other words, 0 - a1 is the same as -a1.

32-bit RISC-V instruction formats

<table>
<tr><th rowspan="2">Format</th><th colspan="32">Bit</th></tr>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>Register</td><td colspan="7">funct7</td><td colspan="5">rs2</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
<tr><td>Immediate</td><td colspan="12">imm[11:0]</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
<tr><td>Upper immediate</td><td colspan="20">imm[31:12]</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
<tr><td>Store</td><td colspan="7">imm[11:5]</td><td colspan="5">rs2</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="5">imm[4:0]</td><td colspan="7">opcode</td></tr>
<tr><td>Branch</td><td>[12]</td><td colspan="6">imm[10:5]</td><td colspan="5">rs2</td><td colspan="5">rs1</td><td colspan="3">funct3</td><td colspan="4">imm[4:1]</td><td>[11]</td><td colspan="7">opcode</td></tr>
<tr><td>Jump</td><td>[20]</td><td colspan="10">imm[10:1]</td><td>[11]</td><td colspan="8">imm[19:12]</td><td colspan="5">rd</td><td colspan="7">opcode</td></tr>
</table>

Classes of instructions: arithmetic instructions

Arithmetic instructions perform mathematical computations such as addition, subtraction, multiplication, and division.

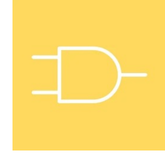
The RISC-V ISA provides a set of arithmetic instructions that operate on integer data types.

Examples of arithmetic instructions

Instruction	Type	Example	Description
Add	R	add rd, rs1, rs2	R[rd] = R[rs1] + R[rs2]
Subtract	R	sub rd, rs1, rs2	R[rd] = R[rs1] – R[rs2]
Add immediate	I	addi rd, rs1, imm12	R[rd] = R[rs1] + SignExt(imm12)
Set less than	R	slt rd, rs1, rs2	R[rd] = (R[rs1] < R[rs2])? 1 : 0
Set less than immediate	I	slti rd, rs1, imm12	R[rd] = (R[rs1] < SignExt(imm12))? 1 : 0
Set less than unsigned	R	sltu rd, rs1, rs2	R[rd] = (R[rs1] <u R[rs2])? 1 : 0
Set less than immediate unsigned	I	sltiu rd, rs1, imm12	R[rd] = (R[rs1] <u SignExt(imm12))? 1 : 0
Load upper immediate	U	lui rd, imm20	R[rd] = SignExt(imm20 << 12)
Add upper immediate to PC	U	auipc rd, imm20	R[rd] = PC + SignExt(imm20 << 12)

Classes of instructions: logical instructions

Logical instructions manipulate the individual bits of binary data and often involve logical AND, OR, XOR, and bit-shifting operations.

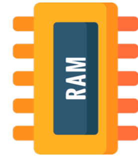


Examples: logical instructions

Instruction	Type	Example	Description
AND	R	and rd, rs1, rs2	R[rd] = R[rs1] & R[rs2]
OR	R	or rd, rs1, rs2	R[rd] = R[rs1] | R[rs2]
XOR	R	xor rd, rs1, rs2	R[rd] = R[rs1] ^ R[rs2]
AND immediate	I	andi rd, rs1, imm12	R[rd] = R[rs1] & SignExt(imm12)
OR immediate	I	ori rd, rs1, imm12	R[rd] = R[rs1] | SignExt(imm12)
XOR immediate	I	xori rd, rs1, imm12	R[rd] = R[rs1] ^ SignExt(imm12)
Shift left logical	R	sll rd, rs1, rs2	R[rd] = R[rs1] << R[rs2]
Shift right logical	R	srl rd, rs1, rs2	R[rd] = R[rs1] >> R[rs2] (logical)
Shift right arithmetic	R	sra rd, rs1, rs2	R[rd] = R[rs1] >> R[rs2] (arithmetic)
Shift left logical immediate	I	slli rd, rs1, shamt	R[rd] = R[rs1] << shamt
Shift right logical imm.	I	srli rd, rs1, shamt	R[rd] = R[rs1] >> shamt (logical)
Shift right arithmetic immediate	I	srai rd, rs1, shamt	R[rd] = R[rs1] >> shamt (arithmetic)

Classes of instructions: data transfer instructions

Data transfer instructions manipulate the individual bits of binary data and often involve logical AND, OR, XOR, and bit-shifting operations.



Examples: data transfer instructions

Instruction	Type	Example	Description
Load doubleword	I	ld rd, imm12(rs1)	R[rd] = Mem8[R[rs1] + SignExt(imm12)]
Load word	I	lw rd, imm12(rs1)	R[rd] = SignExt(Mem4[R[rs1] + SignExt(imm12)])
Load halfword	I	lh rd, imm12(rs1)	R[rd] = SignExt(Mem2[R[rs1] + SignExt(imm12)])
Load byte	I	lb rd, imm12(rs1)	R[rd] = SignExt(Mem1[R[rs1] + SignExt(imm12)])
Load word unsigned	I	lwu rd, imm12(rs1)	R[rd] = ZeroExt(Mem4[R[rs1] + SignExt(imm12)])
Load halfword unsigned	I	lhu rd, imm12(rs1)	R[rd] = ZeroExt(Mem2[R[rs1] + SignExt(imm12)])
Load byte unsigned	I	lbu rd, imm12(rs1)	R[rd] = ZeroExt(Mem1[R[rs1] + SignExt(imm12)])
Store doubleword	S	sd rs2, imm12(rs1)	Mem8[R[rs1] + SignExt(imm12)] = R[rs2]
Store word	S	sw rs2, imm12(rs1)	Mem4[R[rs1] + SignExt(imm12)] = R[rs2](31:0)
Store halfword	S	sh rs2, imm12(rs1)	Mem2[R[rs1] + SignExt(imm12)] = R[rs2](15:0)
Store byte	S	sb rs2, imm12(rs1)	Mem1[R[rs1] + SignExt(imm12)] = R[rs2](7:0)

Classes of instructions: control flow instructions

Control flow instructions allow the program to alter the flow of execution by branching to different sections of code or by calling and returning from subroutines.

Control flow instructions are essential for implementing loops, conditionals, function calls, and program sequencing.

Examples: control flow instructions

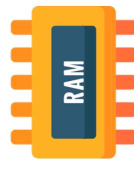
Instruction	Type	Example	Description
Branch equal	SB	beq rs1, rs2, imm12	if (R[rs1] == R[rs2]) pc = pc + SignExt(imm12 << 1)
Branch not equal	SB	bne rs1, rs2, imm12	if (R[rs1] != R[rs2]) pc = pc + SignExt(imm12 << 1)
Branch greater than or equal	SB	bge rs1, rs2, imm12	if (R[rs1] >= R[rs2]) pc = pc + SignExt(imm12 << 1)
Branch greater than or equal unsigned	SB	bgeu rs1, rs2, imm12	if (R[rs1] >=u R[rs2]) pc = pc + SignExt(imm12 << 1)
Branch less than	SB	blt rs1, rs2, imm12	if (R[rs1] < R[rs2]) pc = pc + SignExt(imm12 << 1)
Branch less than unsigned	SB	bltu rs1, rs2, imm12	if (R[rs1] <u R[rs2]) pc = pc + SignExt(imm12 << 1)
Jump and link	UJ	jal rd, imm20	R[rd] = PC + 4 PC = PC + SignExt(imm20 << 1)
Jump and link register	I	jalr rd, imm12(rs1)	R[rd] = PC + 4 PC = (R[rs1] + SignExt(imm12)) & (~1)

Assembler pseudo-instructions

Assembler pseudo-instructions are instructions used in assembly language programming that are not directly translated into machine code instructions.

They provide directives to the assembler to perform certain tasks during the assembly process.

Pseudo-instructions are specific to the assembler and are not part of the actual instruction set architecture.



Assembler pseudo-instructions

<table>
<tr><th>Pseudo-instruction</th><th>Base instruction(s)</th><th>Description</th></tr>
<tr><td>li rd, imm</td><td>addi rd, x0, imm</td><td>Load immediate</td></tr>
<tr><td>la rd, symbol</td><td>auipc rd, D[31:12]+D[11] addi rd, rd, D[11:0]</td><td>Load absolute address where D = symbol – pc</td></tr>
<tr><td>mv rd, rs</td><td>addi rd, rs, 0</td><td>Copy register</td></tr>
<tr><td>not rd, rs</td><td>xori rd, rs, -1</td><td>One’s complement</td></tr>
<tr><td>neg rd, rs</td><td>sub rd, x0, rs</td><td>Two’s complement</td></tr>
<tr><td>bgt{u} rs, rt, offset</td><td>blt{u} rt, rs, offset</td><td>Branch if > (u: unsigned)</td></tr>
<tr><td>ble{u} rs, rt, offset</td><td>bge{u} rt, rs, offset</td><td>Branch if ≥ (u: unsigned)</td></tr>
<tr><td>b{eq|ne}z rs, offset</td><td>b{eq|ne} rs, x0, offset</td><td>Branch if { = |≠ }</td></tr>
<tr><td>b{ge|lt}z rs, offset</td><td>b{ge|lt} rs, x0, offset</td><td>Branch if { ≥ | < }</td></tr>
<tr><td>b{le|gt}z rs, offset</td><td>b{ge|lt} x0, rs, offset</td><td>Branch if { ≤ | > }</td></tr>
<tr><td>j offset</td><td>jal x0, offset</td><td>Unconditional jump</td></tr>
<tr><td>call offset</td><td>jal ra, offset</td><td>Call subroutine (near)</td></tr>
<tr><td>ret</td><td>jalr x0, 0(ra)</td><td>Return from subroutine</td></tr>
<tr><td>nop</td><td>addi x0, x0, 0</td><td>No operation</td></tr>
</table>

RISC-V Implementations

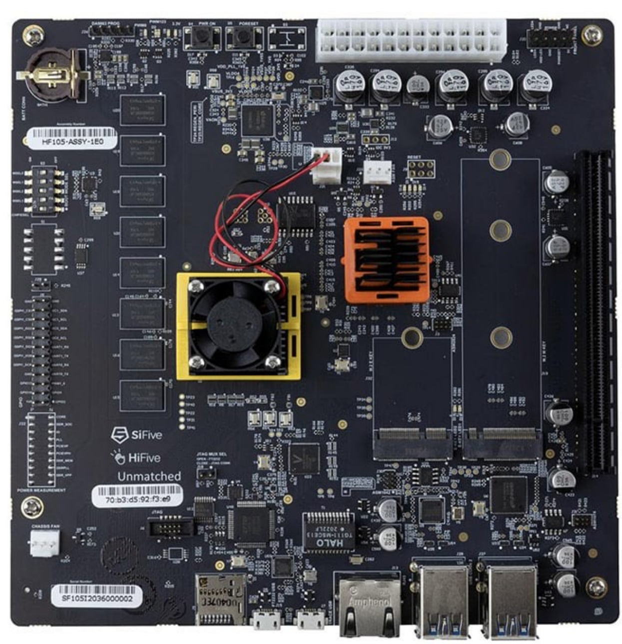
Several companies manufacture RISC-V cores in their microcontrollers, microprocessors, and SoCs.

For example, SiFive manufactures chips from low-end microcontrollers to high-performance SoCs.

Existing RISC-V development boards are designed for software developers, kernel developers, or Linux hackers that are interested in:

- Experimenting with multi-core, Linux-capable RISC-V hardware
- Porting software to the RISC-V platform
- Prototype Embedded/IoT application on RISC-V
- Need to create the next great thing!

RISC-V Implementations: HiFive Unmatched



CPU

SiFive Freedom U740 Quad-core 64-bit

RV64IMAFDC

32KB I-Cache / 32KB D-Cache per core

I/O

1x PCI Express Gen3 x8

4x USB 3.2

2x microUSB

1x JTAG Header

1x 24-pin Peripheral I/O Header

4x GPIO, 2x I2C, 2x QSPI, 2x UART, 1x PWM

Memory and Storage

8 GB DDR4

32 MB Flash

1x microSD card

1x PCI Express Gen 3 x4

Network

1x 10/100/1000 Ethernet

1x M.2 Key E connector for Wi-Fi

RISC-V Implementations: HiFive Pro P550

CPU

Intel-SiFive 64-bit RISC-V core

I/O

USB 3

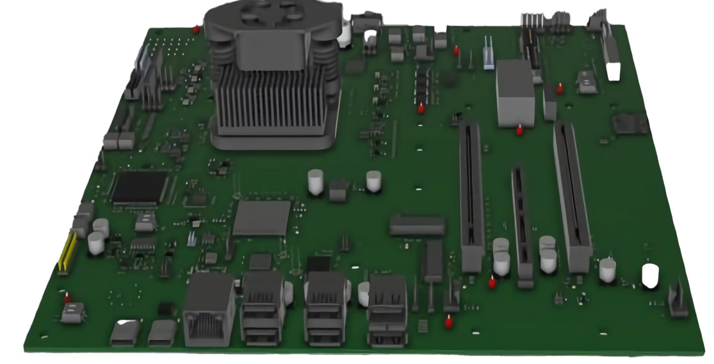
x16 PCIe Gen 3 Expansion Slot

Memory and Storage

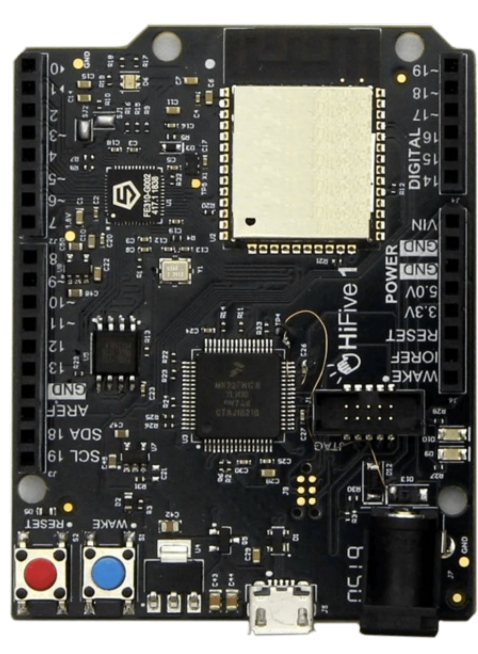
16GB DDR5

Network

Gigabit Ethernet Port



RISC-V Implementations: HiFive1 Rev B



Microcontroller

32-bit RV32IMAC core

16kB L1 Instruction Cache

I/O

JTAG

SPI I2C

UART

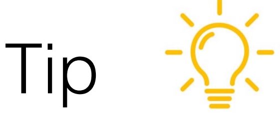
Memory and Storage

16kB Data SRAM

32 Mbit flash

Network

WiFi/BT (off-chip)

Tip 

Check out the RISC-V Green Card.

The green card is a concise reference guide that provides a quick overview of the RISC-V instruction set architecture.

The RISC-V Green Card is a valuable resource for quickly looking up instructions, formats, register usage, and other essential details related to the RISC-V ISA.

https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf

Credits

1. **RISC-V User-level ISA Cheat Sheet**: Jin-Soo Kim, Seoul National University
2. **RISC-V Assembly Language, Computer Systems Architecture:** Stephen Marz, University of Tennessee, Knoxville