FULLY FUNCTIONAL SINGLE CYCLE RV32I PROCESSOR

An Implementation Guide & Documentation

Hazem Taha

Ph.D. Student, McMaster University

Supervised by: Prof. Ameer Abdelhadi

Project Date: 10/02/2023

Contents

1. Introduction

This documentation aims to provide a comprehensive overview of the design, architecture, and functionalities of our RV32I-based processor. It has been developed with a clear emphasis on modularity and clarity, ensuring that users can understand, modify, and extend the design as needed.

1.1. Background

RISC-V (pronounced "risk-five") is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. Unlike proprietary ISAs, RISC-V is free and open, allowing for versatile implementations without licensing fees.

The RV32I, standing for "RISC-V 32-bit Integer", is the base integer instruction set of the RISC-V architecture. It provides a foundational layer for building more complex RISC-V cores and extensions. Delving deeper into its technical aspects:

- **Registers:**
  The RV32I features 32 general-purpose registers, each 32 bits wide. These registers are labeled **x0** through **x31**. Notably, **x0** is hardwired to zero and cannot be altered by any instruction as you will find in the Verilog implementation. The rest of the registers (**x1** to **x31**) can be used freely in operations.
- **Instruction Types:**
  RV32I specifies several instruction formats:
  - **R-type:** Used for arithmetic and logic operations that take two source register operands and produce a result. Example instructions include **ADD**, **SUB**, and **AND**.
  - **I-type:** Primarily for operations that involve a single source register operand and an immediate value. Examples include **ADDI** (add immediate) and **LW** (load word).
  - **S-type:** Used for store operations.
  - **B-type:** Used for branch instructions.
  - **U-type:** Used for operations like **LUI** (load upper immediate).
  - **J-type:** Used for jump and link operations.
- **Branching & Control:**
  RV32I offers a diverse set of branching instructions, enabling conditional and unconditional jumps, supporting both relative and absolute addresses.

- **Arithmetic & Logical Operations:**
  The core ISA provides a rich set of arithmetic (add, subtract, etc.) and logical (AND, OR, XOR, etc.) operations. Both immediate values and register contents can serve as operands, depending on the instruction.

This processor described in this documentation is an embodiment of the RV32I standard, tailored to serve specific application needs. Its modular design ensures that each component effectively performs its role, making the system both efficient and extensible.

Note: The complete Instruction Set, Instruction Set Encodings and Immediate Encodings are referenced at the end pages from J. Winans, "RISC-V Assembly Language Programming."

1.2. Purpose of this Implementation

As there is a lack of complete and stable implementation of Single Cycle RISC-V 32I processor, the purpose of this processor implementation is to provide a clean, modular, and comprehensible Verilog design of the RV32I core. Whether you're a student trying to understand the inner workings of processors, a hobbyist aiming to implement this on FPGA, or an enthusiast in the field of computer architecture, this design serves as a practical reference point.

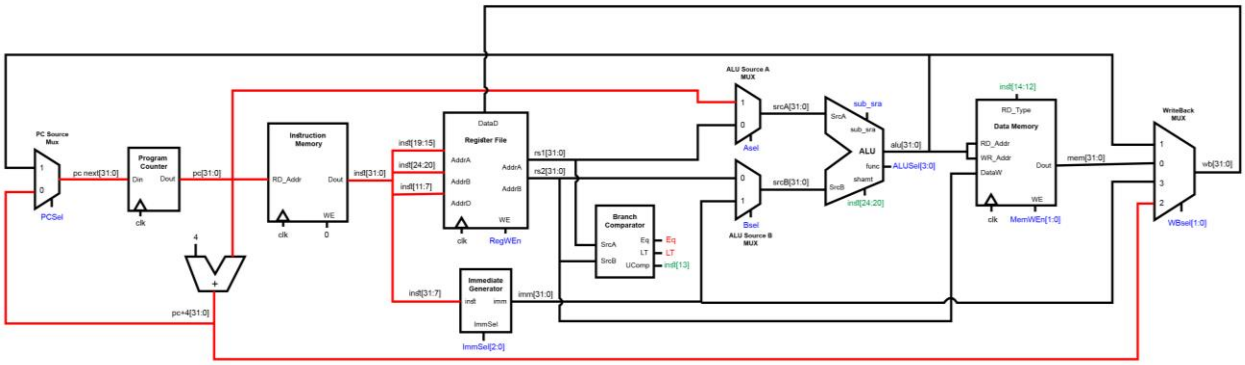
1.3. Key Features

- **Modular Design**: Each component of the processor, from the ALU to the Control Unit, is encapsulated as a separate module. This modularity allows for easy understanding, modification, and testing of individual parts.
- **Complete RV32I Support**: The core provides comprehensive support for the base integer instruction set, ensuring compatibility with RV32I programs.
- **Control Unit Driven**: The control unit directs data flow and operations, making modifications or extensions straightforward.
- **Clock and Reset Integration**: The design incorporates clock and reset logic, ensuring synchronous operations and safe initialization.

2. Architecture Overview and Control Flow

2.1. Instruction Fetch

The RV32I Processor starts with the **Program_Counter** module which keeps track of the address of the instruction that needs to be fetched and executed next. In every clock cycle, the Program Counter can either increment by 4 (to fetch the next instruction in sequence) or be updated with a new value (in the case of jumps or branches). The **Instruction_Memory** module retrieves the instruction stored at the address currently held in the Program Counter. This instruction is then passed on to be decoded in the subsequent module.

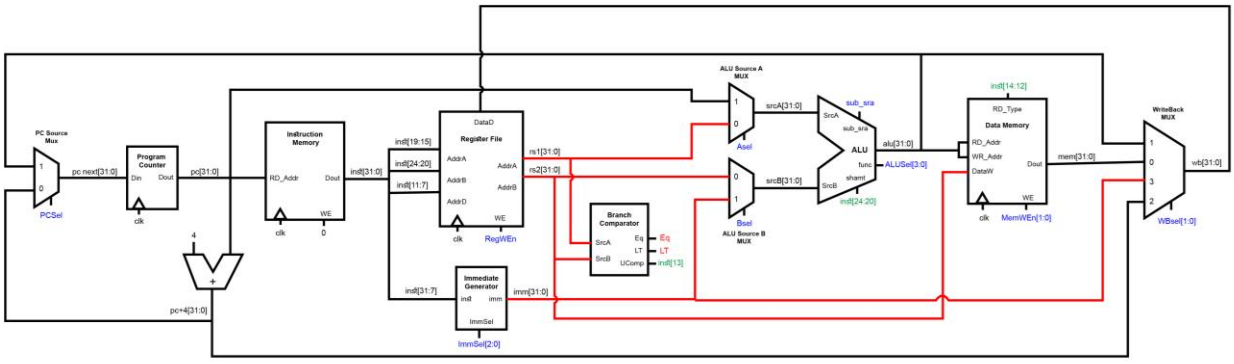


2.2. Instruction Decode & Register Fetch

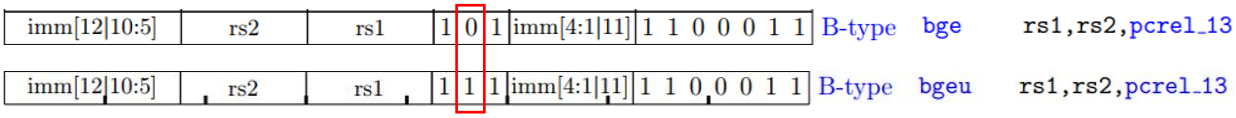
Once the instruction is fetched, it goes to the **Register_File** where two primary tasks occur:

1. Decode the instruction to determine its type and operation.
2. Fetch the register values if the instruction requires them.
3. Generate the correct immediate value using the **Immediate Generator** based on the instruction type as different instructions have different internal representation for the immediate.
4. Generate **Eq** (equal) and **LT** (less than) signals using the **Branch Comparator** to help the control unit decide if a branch should be taken (in case of a branch instruction).

The addresses of the required registers and the immediate (if available) are extracted from specific bits of the instruction, and their corresponding values are fetched for execution.

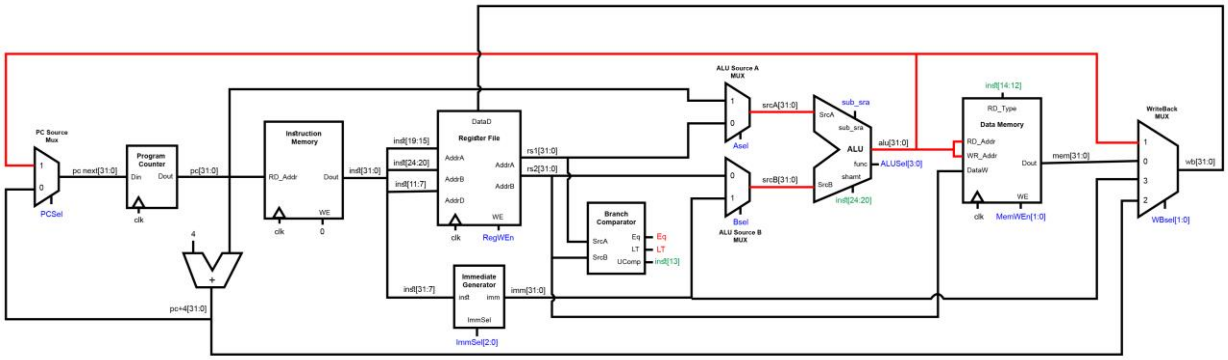


Note that there is a **Ucomp** signal that is fed to the comparator directly from the instruction bit (13). This signal is to decide whether the register comparison is signed or unsigned and hence implement operations like (blt, bltu) and (bge, bgeu).



2.3. ALU Operations

The Arithmetic Logic Unit (**ALU**) module is responsible for performing arithmetic and logical operations based on the decoded instruction. Inputs to the ALU are determined by multiplexers **ALU_SourceA_Mux** and **ALU_SourceB_Mux**, which decide whether a



source operand is a register value, an immediate value, or some other data, like the current PC value. These are controlled by two signals from the control unit, namely **Asel** and **Bsel.**

The **ALU** functionality is determined by the **func** input which takes 4-bits **ALUSel** signal from the control unit according to the below table. In addition to that, some special operations like the (add/sub) and (srl/sra) make use of the same internal component (adder, bit shifter) to save area and cost. The correct function of the component is then decided based on the **sub_sra** control signal from the control unit. If you compared the decoding of the instructions I mentioned, you will find that the only difference between the add and subtract is in bit (30). Same applies to the (srl) and (sra). This can be exploited in the control unit by assigning this bit as the **sub_sra** signals to the ALU during such instructions.

ALU Decoder

Func	Operation
0000	Use 32-Bit Adder (using **sub_sra** signal)
0001	XOR
0010	OR
0011	AND
0100	SLI (using the **shamt** value)
0101	SL
0110	SR (using **sub_sra** signal)
0111	SRI (using the **shamt** value and **sub_sra** signal)
1000	SLT
1001	SLTU

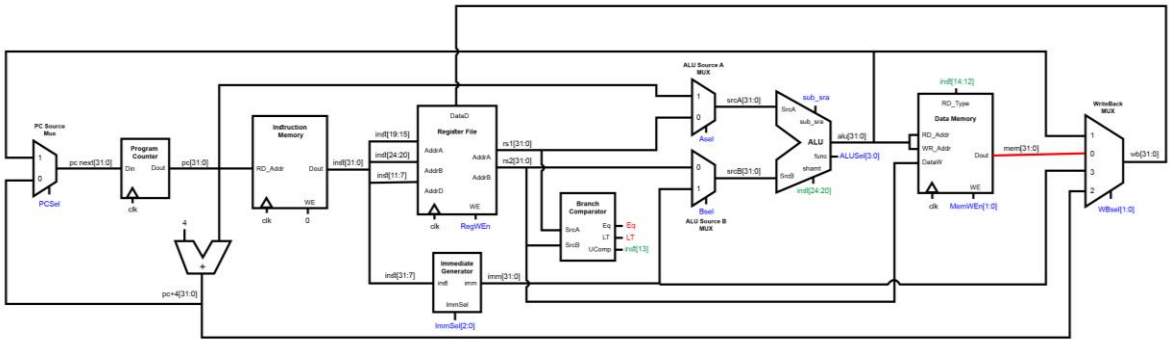
2.4. Memory Operations

The **Data_Memory** module handles data memory operations. If the instruction is a load or store, this module is activated to either read data from or write data to the specified memory address. The developed memory is **aligned**, meaning that full words can only be accesses from addresses that are multiple of 4, and half-words are accessed from addresses that are multiple of 2 as shown in the following table:

Address	Access Size		
	Byte (8bits)	2 Bytes (16bits)	4 Bytes (32bits)
0x0	aligned	aligned	aligned
0x1	aligned	unaligned	unaligned
0x2	aligned	aligned	unaligned
0x3	aligned	unaligned	unaligned
0x4	aligned	aligned	aligned

The memory takes the **WE** (write enable) control signal as 2 bits as it has 4 different configurations. Read, write 1-byte, write half-word, and write full-word.

To be able to perform the different load operations of signed and unsigned data like (lb, lbu, lh, lhu, lw), the **Data_Memory** uses the **funct3** to ensure data is correctly fetched, properly extended, and andy unaligned access is prevented. This is because each of the different load instructions have a unique funct3 bits that can be used as an identifier.

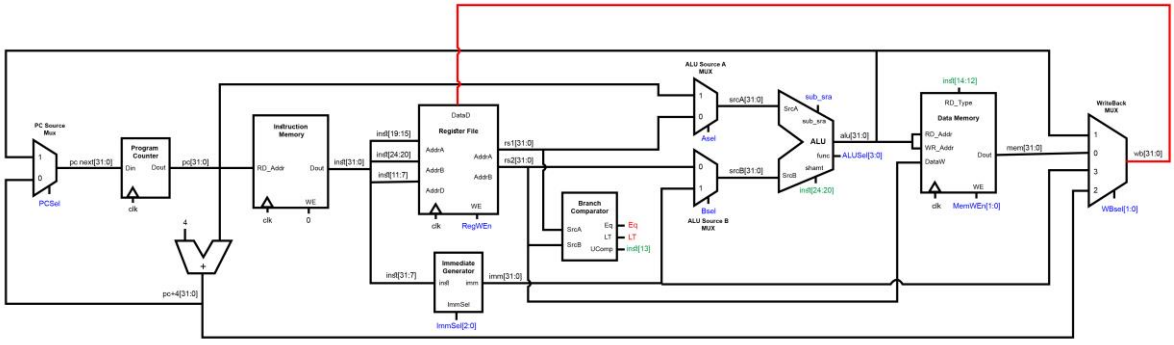


2.5. Write Back

The final stage is the **Write_Back** process where the result of the instruction (be it from the ALU, data extender, immediate generator or PC+4) is written back to the register file. The source of this write-back data is determined by the two-bit **Write_Back_Mux** signal from the control unit.

Different ways of write-back are for the following:

- The output of the data extender is written back in case of load instructions.
- The output of the ALU is written back in case of (auipc, some R-type and I-type) instructions.
- The output of the immediate generator is written back in case of (lui) instruction.
- PC+4 is written back in case of jump instructions (jal, jalr).



3. RV32I Reference Cards

All pages below are copied form J. Winans, “RISC-V Assembly Language Programming.”

Usage Template		Type	Description	Detailed Description
add	rd, rs1, rs2	R	Add	rd ← rs1 + rs2, pc ← pc+4
addi	rd, rs1, imm	I	Add Immediate	rd ← rs1 + imm_i, pc ← pc+4
and	rd, rs1, rs2	R	And	rd ← rs1 ∧ rs2, pc ← pc+4
andi	rd, rs1, imm	I	And Immediate	rd ← rs1 ∧ imm_i, pc ← pc+4
auipc	rd, imm	U	Add Upper Immediate to PC	rd ← pc + imm_u, pc ← pc+4
beq	rs1, rs2, pcrel_13	B	Branch Equal	pc ← pc + ((rs1==rs2) ? imm_b : 4)
bge	rs1, rs2, pcrel_13	B	Branch Greater or Equal	pc ← pc + ((rs1>=rs2) ? imm_b : 4)
bgeu	rs1, rs2, pcrel_13	B	Branch Greater or Equal Unsigned	pc ← pc + ((rs1>=rs2) ? imm_b : 4)
blt	rs1, rs2, pcrel_13	B	Branch Less Than	pc ← pc + ((rs1<rs2) ? imm_b : 4)
bltu	rs1, rs2, pcrel_13	B	Branch Less Than Unsigned	pc ← pc + ((rs1<rs2) ? imm_b : 4)
bne	rs1, rs2, pcrel_13	B	Branch Not Equal	pc ← pc + ((rs1!=rs2) ? imm_b : 4)
csrrw	rd, csr, rs1	I	Atomic Read/Write	rd ← csr, csr ← rs1, pc ← pc+4
csrrs	rd, csr, rs1	I	Atomic Read and Set	rd ← csr, csr ← csr ∨ rs1, pc ← pc+4
csrrc	rd, csr, rs1	I	Atomic Read and Clear	rd ← csr, csr ← csr ∧ ∼rs1, pc ← pc+4
csrrwi	rd, csr, zimm	I	Atomic Read/Write Immediate	rd ← csr, csr ← zimm, pc ← pc+4
csrrsi	rd, csr, zimm	I	Atomic Read and Set Immediate	rd ← csr, csr ← csr ∨ zimm, pc ← pc+4
csrrci	rd, csr, zimm	I	Atomic Read and Clear Immediate	rd ← csr, csr ← csr ∧ ∼zimm, pc ← pc+4
ecall		I	Environment Call	Transfer Control to Debugger
ebreak		I	Environment Break	Transfer Control to Operating System
jal	rd, pcrel_21	J	Jump And Link	rd ← pc+4, pc ← pc+imm_j
jalr	rd, imm(rs1)	I	Jump And Link Register	rd ← pc+4, pc ← (rs1+imm_i) & ∼1
lb	rd, imm(rs1)	I	Load Byte	rd ← sx(m8(rs1+imm_i)), pc ← pc+4
lbu	rd, imm(rs1)	I	Load Byte Unsigned	rd ← zx(m8(rs1+imm_i)), pc ← pc+4
lh	rd, imm(rs1)	I	Load Halfword	rd ← sx(m16(rs1+imm_i)), pc ← pc+4
lhu	rd, imm(rs1)	I	Load Halfword Unsigned	rd ← zx(m16(rs1+imm_i)), pc ← pc+4
lui	rd, imm	U	Load Upper Immediate	rd ← imm_u, pc ← pc+4
lw	rd, imm(rs1)	I	Load Word	rd ← sx(m32(rs1+imm_i)), pc ← pc+4
or	rd, rs1, rs2	R	Or	rd ← rs1 ∨ rs2, pc ← pc+4
ori	rd, rs1, imm	I	Or Immediate	rd ← rs1 ∨ imm_i, pc ← pc+4
sb	rs2, imm(rs1)	S	Store Byte	m8(rs1+imm_s) ← rs2[7:0], pc ← pc+4
sh	rs2, imm(rs1)	S	Store Halfword	m16(rs1+imm_s) ← rs2[15:0], pc ← pc+4
sll	rd, rs1, rs2	R	Shift Left Logical	rd ← rs1 << (rs2%XLEN), pc ← pc+4
slli	rd, rs1, shamt	I	Shift Left Logical Immediate	rd ← rs1 << shamt_i, pc ← pc+4
slt	rd, rs1, rs2	R	Set Less Than	rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4
slti	rd, rs1, imm	I	Set Less Than Immediate	rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4
sltiu	rd, rs1, imm	I	Set Less Than Immediate Unsigned	rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4
sltu	rd, rs1, rs2	R	Set Less Than Unsigned	rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4
sra	rd, rs1, rs2	R	Shift Right Arithmetic	rd ← rs1 >> (rs2%XLEN), pc ← pc+4
srai	rd, rs1, shamt	I	Shift Right Arithmetic Immediate	rd ← rs1 >> shamt_i, pc ← pc+4
srl	rd, rs1, rs2	R	Shift Right Logical	rd ← rs1 >> (rs2%XLEN), pc ← pc+4
srli	rd, rs1, shamt	I	Shift Right Logical Immediate	rd ← rs1 >> shamt_i, pc ← pc+4
sub	rd, rs1, rs2	R	Subtract	rd ← rs1 - rs2, pc ← pc+4
sw	rs2, imm(rs1)	S	Store Word	m32(rs1+imm_s) ← rs2[31:0], pc ← pc+4
xor	rd, rs1, rs2	R	Exclusive Or	rd ← rs1 ⊕ rs2, pc ← pc+4
xori	rd, rs1, imm	I	Exclusive Or Immediate	rd ← rs1 ⊕ imm_i, pc ← pc+4

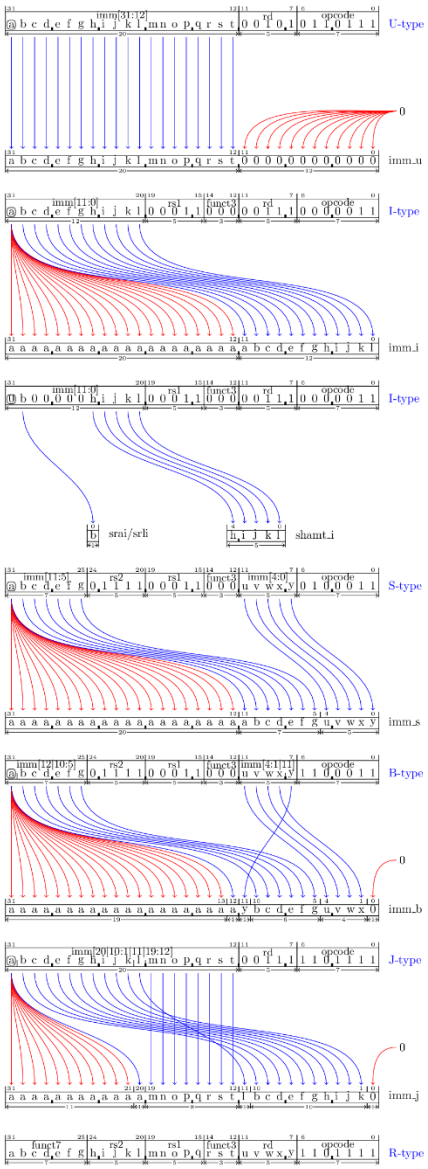
RV32I Base Instruction Set Encoding [1, p. 104]

31 25	24 20	19 15	14 12	11 7	6 0			
imm[31:12]				rd	0 1 1 0 1 1 1	U-type	lui	rd,imm
imm[31:12]				rd	0 0 1 0 1 1 1	U-type	auipc	rd,imm
imm[20|10:1|11|19:12]				rd	1 1 0 1 1 1 1	J-type	jal	rd,pcrel_21
imm[11:0]		rs1	0 0 0	rd	1 1 0 0 1 1 1	I-type	jalr	rd,imm(rs1)
imm[12|10:5]	rs2	rs1	0 0 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	beq	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	0 0 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bne	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 0 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	blt	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 0 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bge	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 1 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bltu	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 1 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bgeu	rs1,rs2,pcrel_13
imm[11:0]		rs1	0 0 0	rd	0 0 0 0 0 1 1	I-type	lb	rd,imm(rs1)
imm[11:0]		rs1	0 0 1	rd	0 0 0 0 0 1 1	I-type	lh	rd,imm(rs1)
imm[11:0]		rs1	0 1 0	rd	0 0 0 0 0 1 1	I-type	lw	rd,imm(rs1)
imm[11:0]		rs1	1 0 0	rd	0 0 0 0 0 1 1	I-type	lbu	rd,imm(rs1)
imm[11:0]		rs1	1 0 1	rd	0 0 0 0 0 1 1	I-type	lhu	rd,imm(rs1)
imm[11:5]	rs2	rs1	0 0 0	imm[4:0]	0 1 0 0 0 1 1	S-type	sb	rs2,imm(rs1)
imm[11:5]	rs2	rs1	0 0 1	imm[4:0]	0 1 0 0 0 1 1	S-type	sh	rs2,imm(rs1)
imm[11:5]	rs2	rs1	0 1 0	imm[4:0]	0 1 0 0 0 1 1	S-type	sw	rs2,imm(rs1)
imm[11:0]		rs1	0 0 0	rd	0 0 1 0 0 1 1	I-type	addi	rd,rs1,imm
imm[11:0]		rs1	0 1 0	rd	0 0 1 0 0 1 1	I-type	slti	rd,rs1,imm
imm[11:0]		rs1	0 1 1	rd	0 0 1 0 0 1 1	I-type	sltiu	rd,rs1,imm
imm[11:0]		rs1	1 0 0	rd	0 0 1 0 0 1 1	I-type	xori	rd,rs1,imm
imm[11:0]		rs1	1 1 0	rd	0 0 1 0 0 1 1	I-type	ori	rd,rs1,imm
imm[11:0]		rs1	1 1 1	rd	0 0 1 0 0 1 1	I-type	andi	rd,rs1,imm
0 0 0 0 0 0 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1	I-type	slli	rd,rs1,shamt
0 0 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	I-type	srli	rd,rs1,shamt
0 1 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	I-type	srai	rd,rs1,shamt
0 0 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1	R-type	add	rd,rs1,rs2
0 1 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1	R-type	sub	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1	R-type	sll	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 1 0	rd	0 1 1 0 0 1 1	R-type	slt	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 1 1	rd	0 1 1 0 0 1 1	R-type	sltu	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1	R-type	xor	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	R-type	srl	rd,rs1,rs2
0 1 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	R-type	sra	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1	R-type	or	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1	R-type	and	rd,rs1,rs2
0 0 0 0 0 0 0 0 0 0 0 0		0 0 0 0 0	0 0 0	0 0 0 0 0	1 1 1 0 0 1 1	I-type	ecall	
0 0 0 0 0 0 0 0 0 0 0 1		0 0 0 0 0	0 0 0	0 0 0 0 0	1 1 1 0 0 1 1	I-type	ebreak	
csr[11:0]		rs1	0 0 1	rd	1 1 1 0 0 1 1	I-type	csrrw	rd,csr,rs1
csr[11:0]		rs1	0 1 0	rd	1 1 1 0 0 1 1	I-type	csrrs	rd,csr,rs1
csr[11:0]		rs1	0 1 1	rd	1 1 1 0 0 1 1	I-type	csrrc	rd,csr,rs1
csr[11:0]		zimm[4:0]	1 0 1	rd	1 1 1 0 0 1 1	I-type	csrrwi	rd,csr,zimm
csr[11:0]		zimm[4:0]	1 1 0	rd	1 1 1 0 0 1 1	I-type	csrrsi	rd,csr,zimm
csr[11:0]		zimm[4:0]	1 1 1	rd	1 1 1 0 0 1 1	I-type	csrrci	rd,csr,zimm

https://github.com/johnwinans/rvalp

Instruction		Description	Operation	Type	funct7 31–25	rs2 24–20	rs1 19–15	funct3 14–12	rd 11–7	opcode 6–0
lui	rd,imm	Load Upper Immediate	rd ← imm_u, pc ← pc+4	U	imm[31:12]				rd	0 1 1 0 1 1 1
auipc	rd,imm	Add Upper Immediate to PC	rd ← pc + imm_u, pc ← pc+4	U	imm[31:12]				rd	0 0 1 0 1 1 1
jal	rd,pcrel_21	Jump And Link	rd ← pc+4, pc ← pc+imm_j	J	imm[20|10:1|11|19:12]				rd	1 1 0 1 1 1 1
jalr	rd,imm(rs1)	Jump And Link Register	rd ← pc+4, pc ← (rs1+imm_i) ∧ ∼1	I	imm[11:0]		rs1	0 0 0	rd	1 1 0 0 1 1 1
beq	rs1,rs2,pcrel_13	Branch Equal	pc ← pc + ((rs1==rs2) ? imm_b : 4)	B	imm[12|10:5]	rs2	rs1	0 0 0	imm[4:1|11]	1 1 0 0 0 1 1
bne	rs1,rs2,pcrel_13	Branch Not Equal	pc ← pc + ((rs1!=rs2) ? imm_b : 4)	B	imm[12|10:5]	rs2	rs1	0 0 1	imm[4:1|11]	1 1 0 0 0 1 1
blt	rs1,rs2,pcrel_13	Branch Less Than	pc ← pc + ((rs1<rs2) ? imm_b : 4)	B	imm[12|10:5]	rs2	rs1	1 0 0	imm[4:1|11]	1 1 0 0 0 1 1
bge	rs1,rs2,pcrel_13	Branch Greater or Equal	pc ← pc + ((rs1>=rs2) ? imm_b : 4)	B	imm[12|10:5]	rs2	rs1	1 0 1	imm[4:1|11]	1 1 0 0 0 1 1
bltu	rs1,rs2,pcrel_13	Branch Less Than Unsigned	pc ← pc + ((rs1<rs2) ? imm_b : 4)	B	imm[12|10:5]	rs2	rs1	1 1 0	imm[4:1|11]	1 1 0 0 0 1 1
bgeu	rs1,rs2,pcrel_13	Branch Greater or Equal Unsigned	pc ← pc + ((rs1>=rs2) ? imm_b : 4)	B	imm[12|10:5]	rs2	rs1	1 1 1	imm[4:1|11]	1 1 0 0 0 1 1
lb	rd,imm(rs1)	Load Byte	rd ← sx(m8(rs1+imm_i)), pc ← pc+4	I	imm[11:0]		rs1	0 0 0	rd	0 0 0 0 0 1 1
lh	rd,imm(rs1)	Load Halfword	rd ← sx(m16(rs1+imm_i)), pc ← pc+4	I	imm[11:0]		rs1	0 0 1	rd	0 0 0 0 0 1 1
lw	rd,imm(rs1)	Load Word	rd ← sx(m32(rs1+imm_i)), pc ← pc+4	I	imm[11:0]		rs1	0 1 0	rd	0 0 0 0 0 1 1
lbu	rd,imm(rs1)	Load Byte Unsigned	rd ← zx(m8(rs1+imm_i)), pc ← pc+4	I	imm[11:0]		rs1	1 0 0	rd	0 0 0 0 0 1 1
lhu	rd,imm(rs1)	Load Halfword Unsigned	rd ← zx(m16(rs1+imm_i)), pc ← pc+4	I	imm[11:0]		rs1	1 0 1	rd	0 0 0 0 0 1 1
sb	rs2,imm(rs1)	Store Byte	m8(rs1+imm_s) ← rs2[7:0], pc ← pc+4	S	imm[11:5]	rs2	rs1	0 0 0	imm[4:0]	0 1 0 0 0 1 1
sh	rs2,imm(rs1)	Store Halfword	m16(rs1+imm_s) ← rs2[15:0], pc ← pc+4	S	imm[11:5]	rs2	rs1	0 0 1	imm[4:0]	0 1 0 0 0 1 1
sw	rs2,imm(rs1)	Store Word	m32(rs1+imm_s) ← rs2[31:0], pc ← pc+4	S	imm[11:5]	rs2	rs1	0 1 0	imm[4:0]	0 1 0 0 0 1 1
addi	rd,rs1,imm	Add Immediate	rd ← rs1 + imm_i, pc ← pc+4	I	imm[11:0]		rs1	0 0 0	rd	0 0 1 0 0 1 1
slti	rd,rs1,imm	Set Less Than Immediate	rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4	I	imm[11:0]		rs1	0 1 0	rd	0 0 1 0 0 1 1
sltiu	rd,rs1,imm	Set Less Than Immediate Unsigned	rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4	I	imm[11:0]		rs1	0 1 1	rd	0 0 1 0 0 1 1
xori	rd,rs1,imm	Exclusive Or Immediate	rd ← rs1 ⊕ imm_i, pc ← pc+4	I	imm[11:0]		rs1	1 0 0	rd	0 0 1 0 0 1 1
ori	rd,rs1,imm	Or Immediate	rd ← rs1 ∨ imm_i, pc ← pc+4	I	imm[11:0]		rs1	1 1 0	rd	0 0 1 0 0 1 1
andi	rd,rs1,imm	And Immediate	rd ← rs1 ∧ imm_i, pc ← pc+4	I	imm[11:0]		rs1	1 1 1	rd	0 0 1 0 0 1 1
slli	rd,rs1,shamt	Shift Left Logical Immediate	rd ← rs1 << shamt_i, pc ← pc+4	I	0 0 0 0 0 0 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1
srli	rd,rs1,shamt	Shift Right Logical Immediate	rd ← rs1 >> shamt_i, pc ← pc+4	I	0 0 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1
srai	rd,rs1,shamt	Shift Right Arithmetic Immediate	rd ← rs1 >> shamt_i, pc ← pc+4	I	0 1 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1
add	rd,rs1,rs2	Add	rd ← rs1 + rs2, pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1
sub	rd,rs1,rs2	Subtract	rd ← rs1 - rs2, pc ← pc+4	R	0 1 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1
sll	rd,rs1,rs2	Shift Left Logical	rd ← rs1 << (rs2%XLEN), pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1
slt	rd,rs1,rs2	Set Less Than	rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	0 1 0	rd	0 1 1 0 0 1 1
sltu	rd,rs1,rs2	Set Less Than Unsigned	rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	0 1 1	rd	0 1 1 0 0 1 1
xor	rd,rs1,rs2	Exclusive Or	rd ← rs1 ⊕ rs2, pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1
srl	rd,rs1,rs2	Shift Right Logical	rd ← rs1 >> (rs2%XLEN), pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1
sra	rd,rs1,rs2	Shift Right Arithmetic	rd ← rs1 >> (rs2%XLEN), pc ← pc+4	R	0 1 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1
or	rd,rs1,rs2	Or	rd ← rs1 ∨ rs2, pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1
and	rd,rs1,rs2	And	rd ← rs1 ∧ rs2, pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1
ecall		Trap to Debugger		I	0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0	0 0 0	0 0 0 0 0	1 1 1 0 0 1 1	
ebreak		Trap to Operating System		I	0 0 0 0 0 0 0 0 0 0 0 1	0 0 0 0 0	0 0 0	0 0 0 0 0	1 1 1 0 0 1 1	
csrrw	rd,csr,rs1	Atomic Read/Write	rd ← csr, csr ← rs1, pc ← pc+4	I	csr[11:0]		rs1	0 0 1	rd	1 1 1 0 0 1 1
csrrs	rd,csr,rs1	Atomic Read and Set	rd ← csr, csr ← csr ∨ rs1, pc ← pc+4	I	csr[11:0]		rs1	0 1 0	rd	1 1 1 0 0 1 1
csrrc	rd,csr,rs1	Atomic Read and Clear	rd ← csr, csr ← csr ∧ ∼rs1, pc ← pc+4	I	csr[11:0]		rs1	0 1 1	rd	1 1 1 0 0 1 1
csrrwi	rd,csr,zimm	Atomic Read/Write Immediate	rd ← csr, csr ← zimm, pc ← pc+4	I	csr[11:0]		zimm[4:0]	1 0 1	rd	1 1 1 0 0 1 1
csrrsi	rd,csr,zimm	Atomic Read and Set Immediate	rd ← csr, csr ← csr ∨ zimm, pc ← pc+4	I	csr[11:0]		zimm[4:0]	1 1 0	rd	1 1 1 0 0 1 1
csrrci	rd,csr,zimm	Atomic Read and Clear Immediate	rd ← csr, csr ← csr ∧ ∼zimm, pc ← pc+4	I	csr[11:0]		zimm[4:0]	1 1 1	rd	1 1 1 0 0 1 1

RV32I Reference Card


imm[31:12]
rd
opcode
U-type
0
imm_u
imm[11:0]
rs1
funct3
I-type
imm_i
shamt
srai/srli
shamt_i
imm[11:5]
rs2
S-type
imm_s
imm[12|10:5]
imm[4:1|11]
B-type
imm_b
imm[20|10:1|11|19:12]
J-type
imm_j
funct7
R-type