

Tiny RISC-V Instruction Set Architecture

Christopher Batten

The Tiny RISC-V ISA is a subset of the 32-bit RISC-V ISA suitable for teaching. More specifically, the Tiny RISC-V ISA is a subset of the RV32IM ISA. The Tiny RISC-V ISA is divided into two versions: **TinyRV1** includes just eight instructions and is suitable for lecture notes, problem sets, and exams, while **TinyRV2** includes 34 instructions and is suitable for running simple C programs. This document provides a compact description of the Tiny RISC-V ISA, but it should be read in combination with the full RISC-V ISA manuals, which can be found on Canvas. The full RISC-V ISA manuals include a wealth of useful information about why the architecture designers made specific design decisions.

Table of Contents

1	Architectural State	2
2	Tiny RISC-V Instruction Overview	4
3	Tiny RISC-V Instruction and Immediate Encoding	5
3.1	Tiny RISC-V Instruction Formats	5
3.2	Tiny RISC-V Immediate Formats	5
4	Tiny RISC-V Instruction Details	6
4.1	Control/Status Register Instructions	6
4.2	Register-Register Arithmetic Instructions	7
4.3	Register-Immediate Arithmetic Instructions	11
4.4	Memory Instructions	14
4.5	Unconditional Jump Instructions	15
4.6	Conditional Branch Instructions	16
5	Tiny RISC-V Privileged ISA	19
6	Tiny RISC-V Pseudo-Instructions	21

1 Architectural State

Data Formats

Tiny RISC-V only supports 4B signed and unsigned integer values. There are no byte nor half-word values and no floating-point.

General Purpose Registers

There are 31 general-purpose registers x1-x31 (called x registers), which hold integer values. Register x0 is hardwired to the constant zero. Each register is 32 bits wide. Tiny RISC-V uses the same calling convention and symbolic register names as RISC-V:

Register	ABI Name	Description	Register	ABI Name	Description
x0	zero	The constant value 0	x16	a6	Function Arguments (caller saved)
x1	ra	Return Address (caller saved)	x17	a7	
x2	sp	Stack Pointer (callee saved)	x18	s2	Saved Registers (callee saved)
x3	gp	Global Pointer	x19	s3	
x4	tp	Thread Pointer	x20	s4	
x5	t0	Temporary Register (caller saved)	x21	s5	
x6	t1				
x7	t2				
x8	s0/fp	Saved Register/Frame Pointer (callee saved)	x22	s6	
x9	s1	Saved Register (callee saved)	x23	s7	
x10	a0	Function Arguments and/or Return values (caller saved)	x24	s8	
x11	a1				
x12	a2	Function Arguments (callee saved)	x25	s9	
x13	a3		x26	s10	
x14	a4		x27	s11	
x15	a5		x28	t3	Temporary Register (caller saved)
		x29	t4		
		x30	t5		
		x31	t6		

Memory

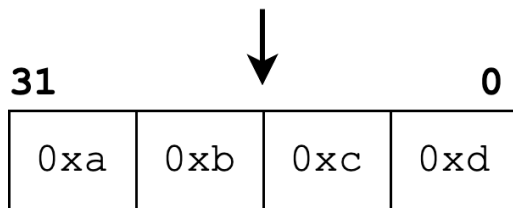
Tiny RISC-V only supports a 1MB virtual memory address space from `0x00000000` to `0x000fffff`. The result of memory accesses to addresses larger than `0x000fffff` are undefined.

A key feature of any ISA is identifying the endianness of the memory system. Endianness specifies if we load a word in memory, what order should those bytes appear in the destination register. Assume the letter `0x0a` is at byte address `0x0`, the letter `0x0b` is at byte address `0x1`, the letter `0x0c` is at byte address `0x2`, and the letter `0x0d` is at byte address `0x3`. If we load a four-byte word from address `0x0`, there are two options: the destination register can either hold `0x0a0b0c0d` (big endian) or `0x0d0c0b0a` (little endian). There is no significant benefit of one system over the other. Tiny RISC-V uses a little endian memory system.

Address	Value
<code>0x3</code>	<code>0xd</code>
<code>0x2</code>	<code>0xc</code>
<code>0x1</code>	<code>0xb</code>
<code>0x0</code>	<code>0xa</code>

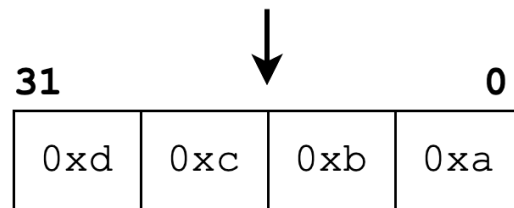
Big Endian

Load from `0x0`



Little Endian

Load from `0x0`



2 Tiny RISC-V Instruction Overview

Here is a brief list of the instructions which make up both versions of the Tiny RISC-V ISA, and then some discussion about the differences between the two versions.

TinyRV1

TinyRV1 contains a very small subset of the TinyRV2 ISA suitable for illustrating how small assembly sequences execute on various microarchitectures in lecture, problem sets, and exams.

- **Register-Register Arithmetic Instructions:** [ADD](#), [MUL](#)
- **Register-Immediate Arithmetic Instructions:** [ADDI](#)
- **Memory Instructions:** [LW](#), [SW](#)
- **Unconditional Jump Instructions:** [JAL](#), [JR](#)
- **Conditional Branch Instructions:** [BNE](#)

TinyRV2

TinyRV2 is suitable for executing simple C programs that do not use system calls.

- **Register-Register Arithmetic Instructions:** [ADD](#), [SUB](#), [AND](#), [OR](#), [XOR](#), [SLT](#), [SLTU](#), [SRA](#), [SRL](#), [SLL](#), [MUL](#)
- **Register-Immediate Arithmetic Instructions:** [ADDI](#), [ANDI](#), [ORI](#), [XORI](#), [SLTI](#), [SLTIU](#), [SRAI](#), [SRLI](#), [SLLI](#), [LUI](#), [AUIPC](#)
- **Memory Instructions:** [LW](#), [SW](#)
- **Unconditional Jump Instructions:** [JAL](#), [JALR](#)
- **Conditional Branch Instructions:** [BEQ](#), [BNE](#), [BLT](#), [BGE](#), [BLTU](#), [BGEU](#)
- **Control/Status Register Instructions:** [CSRR](#), [CSRW](#) (`proc2mngr`, `mngr2proc`, `stats_en`, `coreid`, `numcores`)

Differences

TinyRV1 includes the JR instruction, but technically this is not a real instruction but is instead a pseudo-instruction for the following usage of JALR:

```
jalr x0, rs1, 0
```

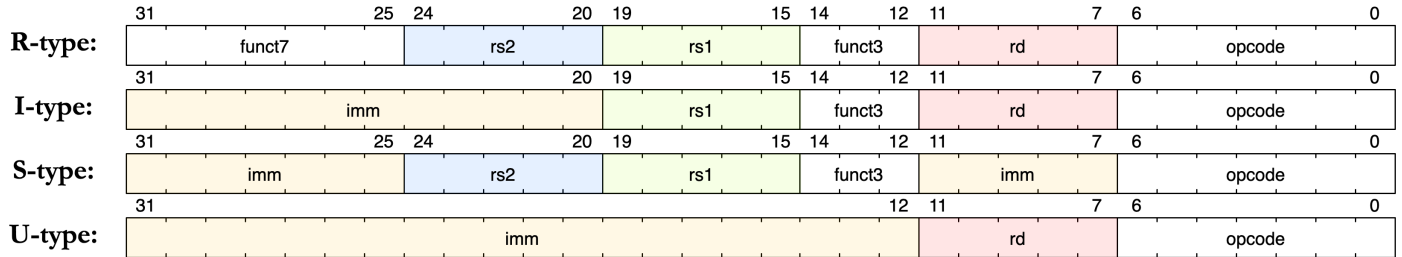
The JALR instruction is a bit complicated, and we really only need the JR functionality to explain function calls in TinyRV1. So TinyRV1 only includes the JR pseudo-instruction, while TinyRV2 includes the full JALR instruction.

CSRR and CSRW are also pseudo-instructions in the full RV32IM ISA for specific usage of the CSRRW and CSRWS instructions. The full CSRRW and CSRWS instructions are rather complicated and we don't actually need any functionality beyond what CSRR and CSRW provide. TinyRV2 only includes the CSRR and CSRW pseudo-instructions.

3 Tiny RISC-V Instruction and Immediate Encoding

The Tiny RISC-V ISA uses the same instruction encoding as RISC-V. There are four instruction types and five immediate encodings. Each instruction has a specific instruction type, and if that instruction includes an immediate, then it will also have an immediate type.

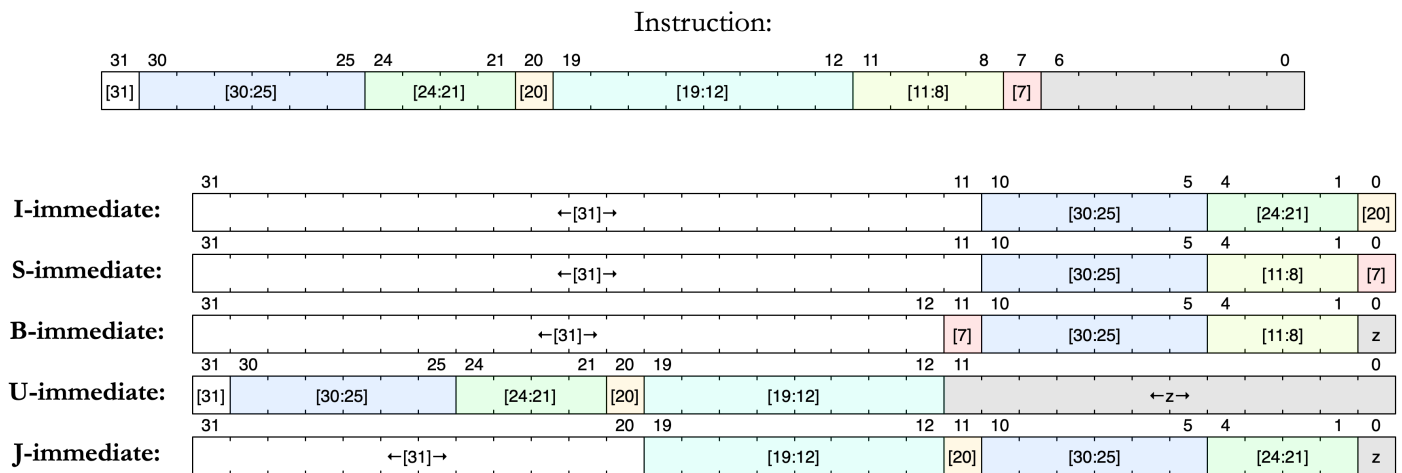
3.1 Tiny RISC-V Instruction Formats



3.2 Tiny RISC-V Immediate Formats

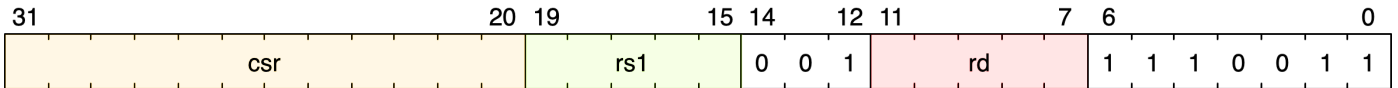
RISC-V has an asymmetric immediate encoding, which means that the immediates are formed by concatenating different bits in an asymmetric order based on the specific immediate formats. Note that in RISC-V, all immediates are always sign extended, and the sign-bit for the immediate is always in bit 31 of the instruction.

The following diagram illustrates how to create a 32-bit immediate from each of the five immediate formats. The fields are labeled with $[n]$ to indicate that bit(s) n from the instruction bits are used to construct the field. $\leftarrow [n] \rightarrow$ is used to indicate repeating bit n of the instruction to fill that field, and z is used to indicate bit(s) which is always set to zero.



CSRW

- **Summary:** Move value in GPR to control/status register
- **Assembly:** `csrw csr, rs1`
- **Semantics:** $CSR[csr] = R[rs1]$
- **Format:** I-type, I-immediate

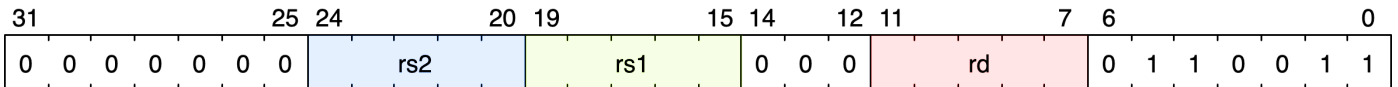


The control/status register write instruction is used to read a GPR and write the result to a CSR. The CSRs supported in TinyRV2 are listed in Section 5. Note that in RISC-V CSRW is really a pseudo-instruction for a specific usage of CSRRW, but in TinyRV2 we only support the subset of CSRRW captured by CSRW. See Section 6 for more details about pseudo-instructions.

4.2 Register-Register Arithmetic Instructions

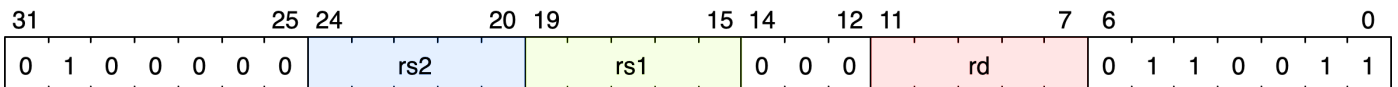
ADD

- **Summary:** Addition with 3 GPRs, no overflow exception
- **Assembly:** `add rd, rs1, rs2`
- **Semantics:** $R[rd] = R[rs1] + R[rs2]$
- **Format:** R-type



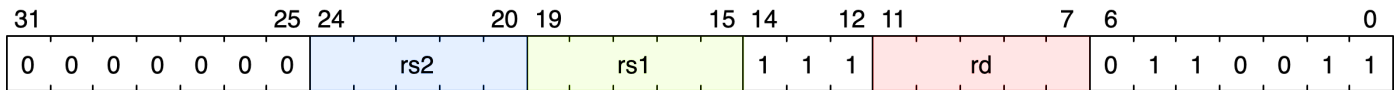
SUB

- **Summary:** Subtraction with 3 GPRs, no overflow exception
- **Assembly:** `sub rd, rs1, rs2`
- **Semantics:** $R[rd] = R[rs1] - R[rs2]$
- **Format:** R-type



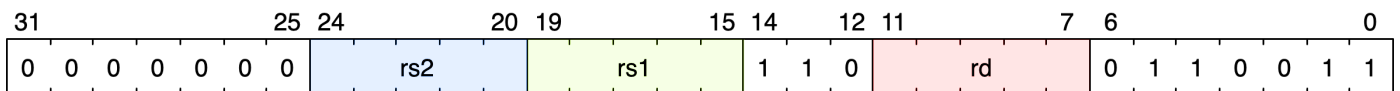
AND

- **Summary:** Bitwise logical AND with 3 GPRs
- **Assembly:** `and rd, rs1, rs2`
- **Semantics:** $R[rd] = R[rs1] \& R[rs2]$
- **Format:** R-type



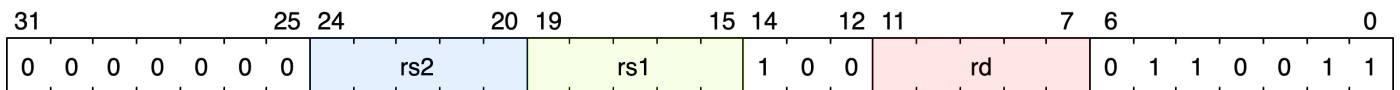
OR

- **Summary:** Bitwise logical OR with 3 GPRs
- **Assembly:** `or rd, rs1, rs2`
- **Semantics:** $R[rd] = R[rs1] | R[rs2]$
- **Format:** R-type



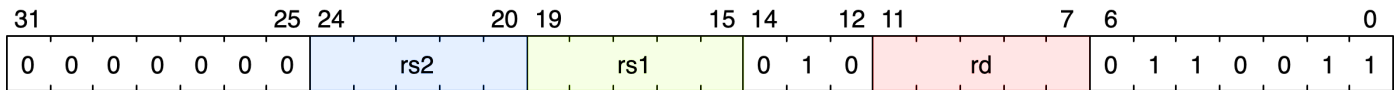
XOR

- **Summary:** Bitwise logical XOR with 3 GPRs
- **Assembly:** `xor rd, rs1, rs2`
- **Semantics:** $R[rd] = R[rs1] \hat{\ } R[rs2]$
- **Format:** R-type



SLT

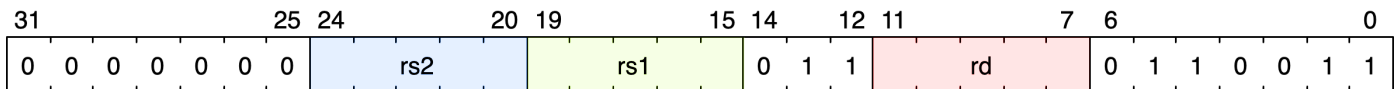
- **Summary:** Record result of signed less-than comparison with 2 GPRs
- **Assembly:** `slt rd, rs1, rs2`
- **Semantics:** $R[rd] = (R[rs1] <_s R[rs2])$
- **Format:** R-type



This instruction uses a *signed* comparison.

SLTU

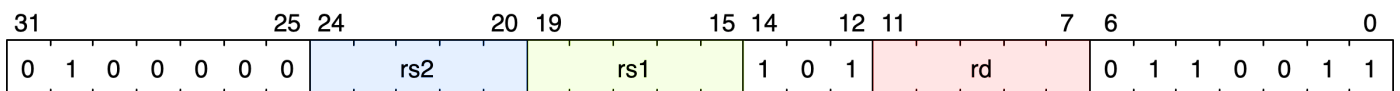
- **Summary:** Record result of unsigned less-than comparison with 2 GPRs
- **Assembly:** `sltu rd, rs1, rs2`
- **Semantics:** $R[rd] = (R[rs1] <_u R[rs2])$
- **Format:** R-type



This instruction uses an *unsigned* comparison.

SRA

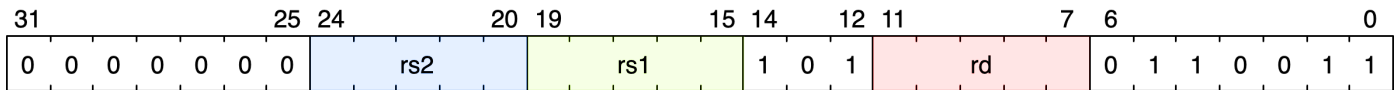
- **Summary:** Shift right arithmetic by register value (sign-extend)
- **Assembly:** `sra rd, rs1, rs2`
- **Semantics:** $R[rd] = R[rs1] \ggg R[rs2][4:0]$
- **Format:** R-type



Note that the hardware should ensure that the sign-bit of $R[rs1]$ is extended to the right as it does the right shift. The hardware *must* only use the bottom five bits of $R[rs2]$ when performing the shift.

SRL

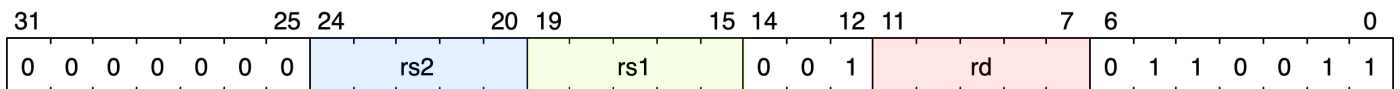
- **Summary:** Shift right logical by register value (append zeroes)
- **Assembly:** `srl rd, rs1, rs2`
- **Semantics:** $R[rd] = R[rs1] \gg R[rs2][4:0]$
- **Format:** R-type



Note that the hardware should append zeros to the left as it does the right shift. The hardware *must* only use the bottom five bits of $R[rs2]$ when performing the shift.

SLL

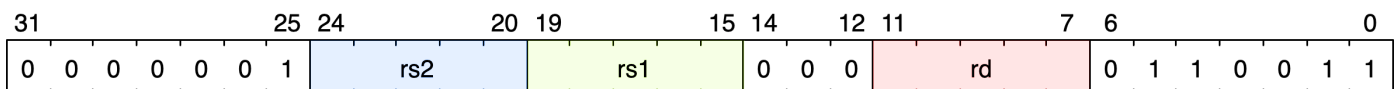
- **Summary:** Shift left logical by register value (append zeroes)
- **Assembly:** `sll rd, rs1, rs2`
- **Semantics:** $R[rd] = R[rs1] \ll R[rs2][4:0]$
- **Format:** R-type



Note that the hardware should append zeros to the right as it does the left shift. The hardware *must* only use the bottom five bits of $R[rs2]$ when performing the shift.

MUL

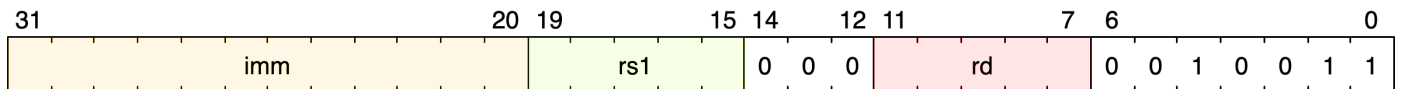
- **Summary:** Signed multiplication with 3 GPRs, no overflow exception
- **Assembly:** `mul rd, rs1, rs2`
- **Semantics:** $R[rd] = R[rs1] * R[rs2]$
- **Format:** R-type



4.3 Register-Immediate Arithmetic Instructions

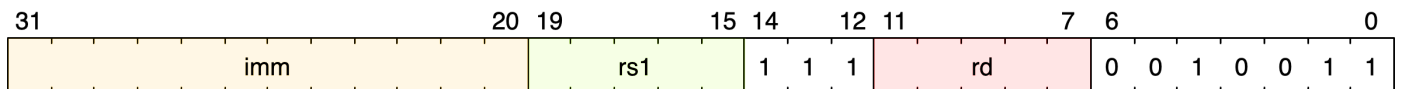
ADDI

- **Summary:** Add constant, no overflow exception
- **Assembly:** `addi rd, rs1, imm`
- **Semantics:** $R[rd] = R[rs1] + \text{sext}(imm)$
- **Format:** I-type, I-immediate



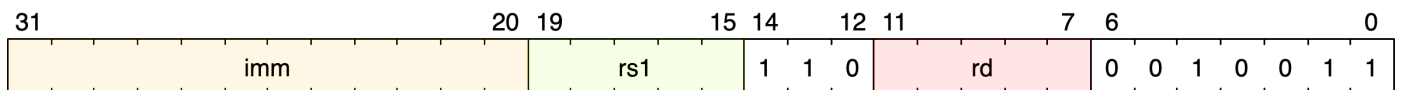
ANDI

- **Summary:** Bitwise logical AND with constant
- **Assembly:** `andi rd, rs1, imm`
- **Semantics:** $R[rd] = R[rs1] \& \text{sext}(imm)$
- **Format:** I-type, I-immediate



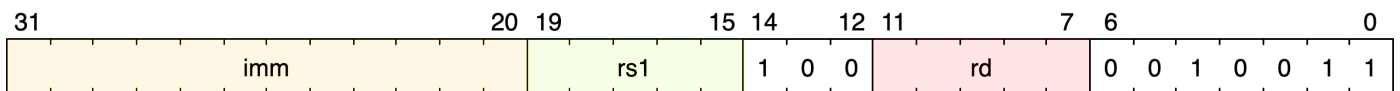
ORI

- **Summary:** Bitwise logical OR with constant
- **Assembly:** `ori rd, rs1, imm`
- **Semantics:** $R[rd] = R[rs1] | \text{sext}(imm)$
- **Format:** I-type, I-immediate



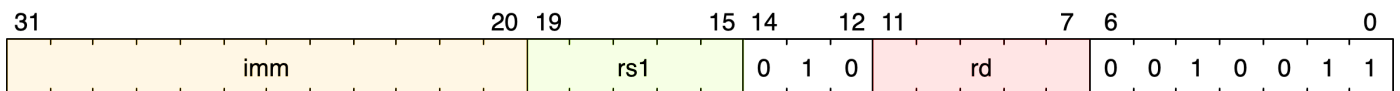
XORI

- **Summary:** Bitwise logical XOR with constant
- **Assembly:** `xori rd, rs1, imm`
- **Semantics:** $R[rd] = R[rs1] \hat{\ } sext(imm)$
- **Format:** I-type, I-immediate



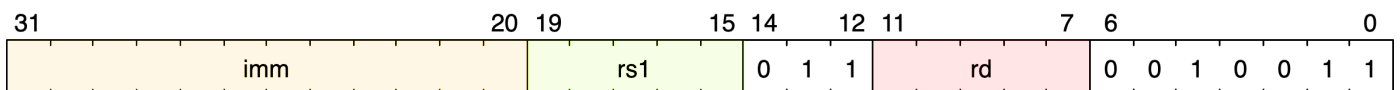
SLTI

- **Summary:** Set GPR if source GPR is less than constant, signed comparison
- **Assembly:** `slti rd, rs1, imm`
- **Semantics:** $R[rd] = (R[rs1] <_s sext(imm))$
- **Format:** I-type, I-immediate



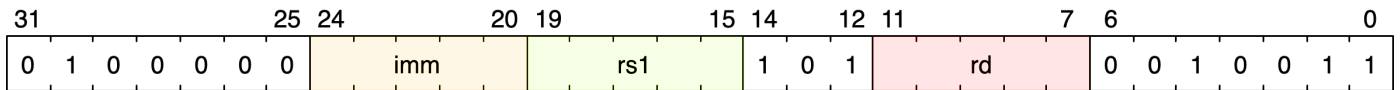
SLTIU

- **Summary:** Set GPR if source GPR is less than constant, unsigned comparison
- **Assembly:** `sltiu rd, rs1, imm`
- **Semantics:** $R[rd] = (R[rs1] <_u sext(imm))$
- **Format:** I-type, I-immediate



SRAI

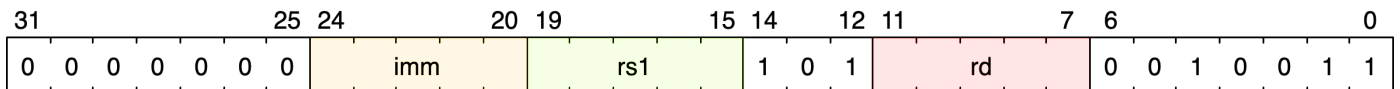
- **Summary:** Shift right arithmetic by constant (sign-extend)
- **Assembly:** `srai rd, rs1, imm`
- **Semantics:** $R[rd] = R[rs1] \ggg imm$
- **Format:** R-type, I-immediate (variant)



Note that the hardware should ensure that the sign-bit of $R[rs1]$ is extended to the right as it does the right shift.

SRLI

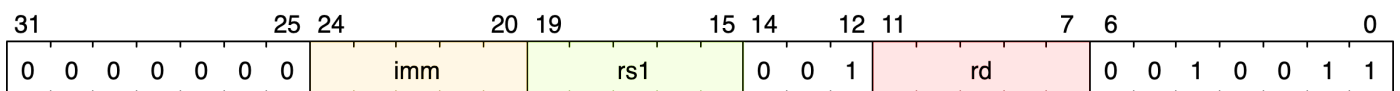
- **Summary:** Shift right logical by constant (append zeroes)
- **Assembly:** `srlr rd, rs1, imm`
- **Semantics:** $R[rd] = R[rs1] \gg imm$
- **Format:** R-type, I-immediate (variant)



Note that the hardware should append zeros to the left as it does the right shift.

SLLI

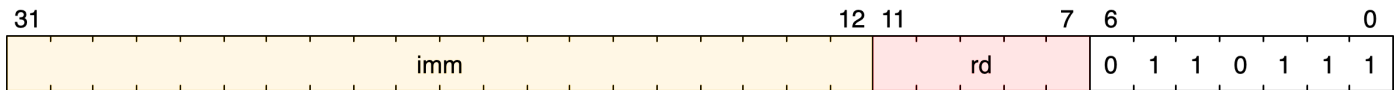
- **Summary:** Shift left logical constant (append zeroes)
- **Assembly:** `slli rd, rs1, imm`
- **Semantics:** $R[rd] = R[rs1] \ll imm$
- **Format:** R-type, I-immediate (variant)



Note that the hardware should append zeros to the right as it does the left shift.

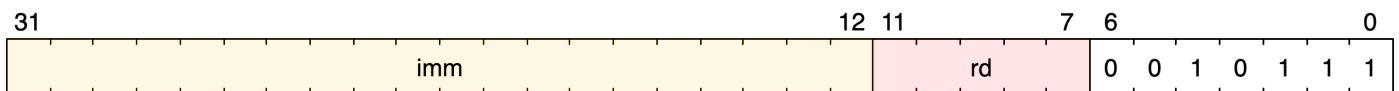
LUI

- **Summary:** Load constant into upper bits of word
- **Assembly:** lui rd, imm
- **Semantics:** $R[rd] = \text{imm} \ll 12$
- **Format:** U-type, U-immediate



AUIPC

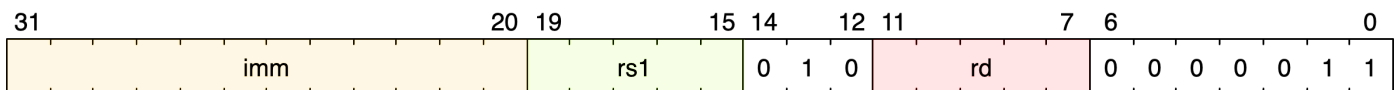
- **Summary:** Load PC + constant into upper bits of word
- **Assembly:** auipc rd, imm
- **Semantics:** $R[rd] = PC + (\text{imm} \ll 12)$
- **Format:** U-type, U-immediate



4.4 Memory Instructions

LW

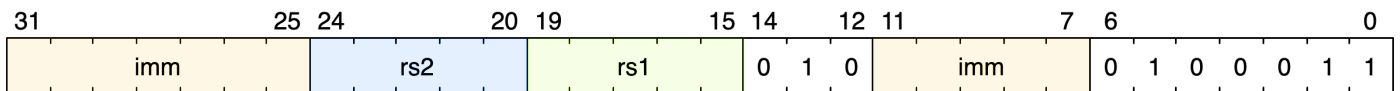
- **Summary:** Load word from memory
- **Assembly:** lw rd, imm(rs1)
- **Semantics:** $R[rd] = M_{4B}[R[rs1] + \text{sext}(\text{imm})]$
- **Format:** I-type, I-immediate



All addresses used with LW instructions must be four-byte aligned. This means the bottom two bits of every effective address (i.e., after the base address is added to the offset) will always be zero.

SW

- **Summary:** Store word into memory
- **Assembly:** `sw rs2, imm(rs1)`
- **Semantics:** $M_{4B}[R[rs1] + sext(imm)] = R[rs2]$
- **Format:** S-type, S-immediate

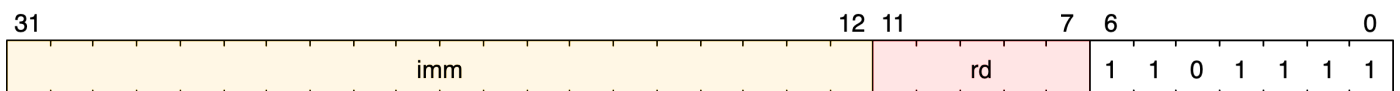


All addresses used with SW instructions must be four-byte aligned. This means the bottom two bits of every effective address (i.e., after the base address is added to the offset) will always be zero. It is undefined to store to an address location which is then fetched as an instruction.

4.5 Unconditional Jump Instructions

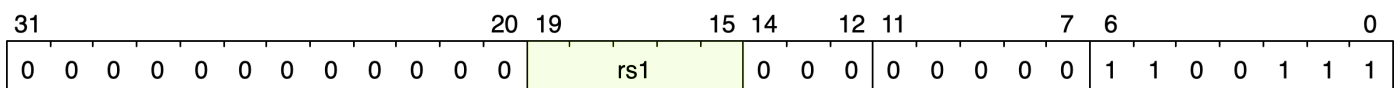
JAL

- **Summary:** Jump to address and place return address in GPR
- **Assembly:** `jal rd, imm`
- **Semantics:** $R[rd] = PC + 4; PC = PC + sext(imm)$
- **Format:** U-type, J-immediate



JR

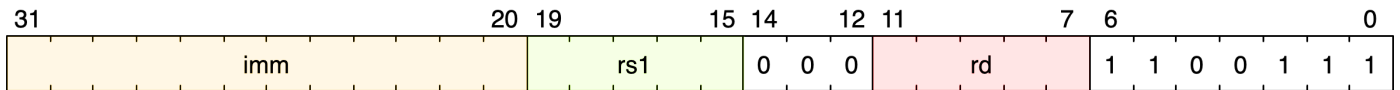
- **Summary:** Jump to address
- **Assembly:** `jr rs1`
- **Semantics:** $PC = R[rs1]$
- **Format:** I-Type



Note that JR is a "real" instruction in TinyRV1, but it is a pseudo-instruction for a specific usage of JALR. We don't really worry about zero-ing out the the least-significant bit to zero in TinyRV1, but this must be done for TinyRV2.

JALR

- **Summary:** Jump to address and place return address in GPR
- **Assembly:** jalr rd, rs1, imm
- **Semantics:** $R[rd] = PC + 4$; $PC = (R[rs1] + sext(imm)) \& 0xffffffffe$
- **Format:** I-Type, I-immediate

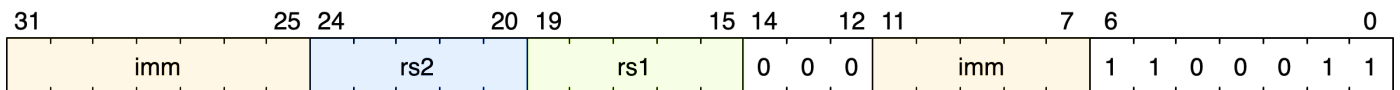


Note that the target address is obtained by adding the 12-bit signed I-immediate to the value in register `rs1`, then setting the least-significant bit of the result to zero. In other words, the JALR instruction ignores the lowest bit of the calculated target address.

4.6 Conditional Branch Instructions

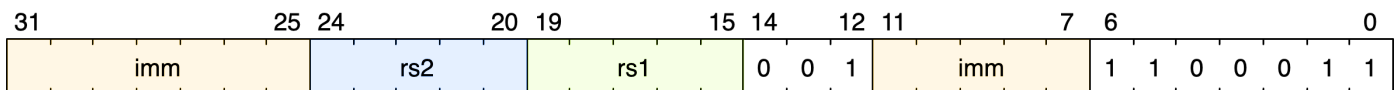
BEQ

- **Summary:** Branch if 2 GPRs are equal
- **Assembly:** beq rs1, rs2, imm
- **Semantics:** $PC = (R[rs1] == R[rs2]) ? PC + sext(imm) : PC + 4$
- **Format:** S-type, B-immediate



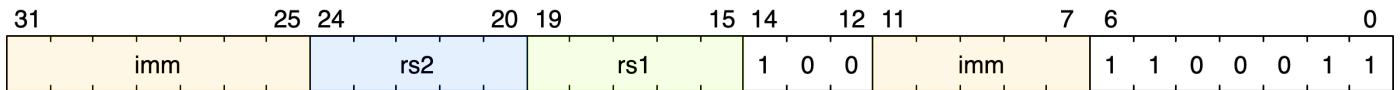
BNE

- **Summary:** Branch if 2 GPRs are not equal
- **Assembly:** bne rs1, rs2, imm
- **Semantics:** $PC = (R[rs1] != R[rs2]) ? PC + sext(imm) : PC + 4$
- **Format:** S-type, B-immediate



BLT

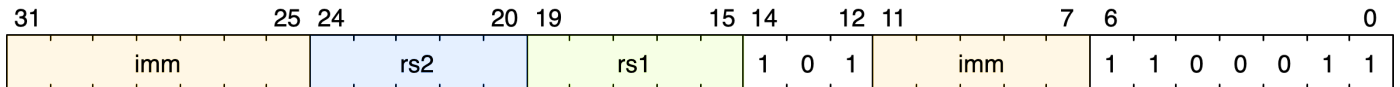
- **Summary:** Branch based on signed comparison of two GPRs
- **Assembly:** `blt rs1, rs2, imm`
- **Semantics:** $PC = (R[rs1] <_s R[rs2]) ? PC + sext(imm) : PC + 4$
- **Format:** S-type, B-immediate



This instruction uses a *signed* comparison.

BGE

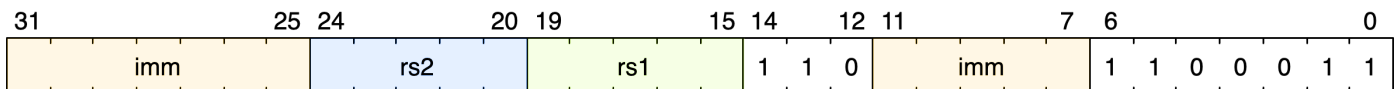
- **Summary:** Branch based on signed comparison of two GPRs
- **Assembly:** `bge rs1, rs2, imm`
- **Semantics:** $PC = (R[rs1] \geq_s R[rs2]) ? PC + sext(imm) : PC + 4$
- **Format:** S-type, B-immediate



This instruction uses a *signed* comparison.

BLTU

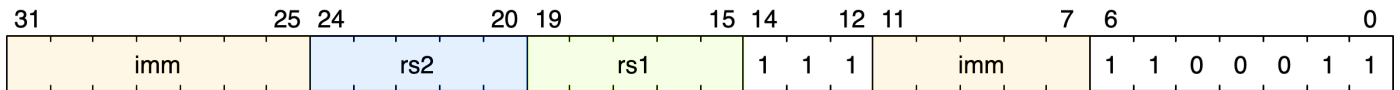
- **Summary:** Branch based on unsigned comparison of two GPRs
- **Assembly:** `bltu rs1, rs2, imm`
- **Semantics:** $PC = (R[rs1] <_u R[rs2]) ? PC + sext(imm) : PC + 4$
- **Format:** S-type, B-immediate



This instruction uses an *unsigned* comparison.

BGEU

- **Summary:** Branch based on unsigned comparison of two GPRs
- **Assembly:** `bgeu rs1, rs2, imm`
- **Semantics:** $PC = (R[rs1] \geq_u R[rs2]) ? PC + sext(imm) : PC + 4$
- **Format:** S-type, B-immediate



This instruction uses an *unsigned* comparison.

5 Tiny RISC-V Privileged ISA

Tiny RISC-V does not support any kind of distinction between user and privileged mode. Using the terminology in the RISC-V Volume 2 (Privileged) ISA manual, Tiny RISC-V only supports *M*-mode (“Machine” privileges, the highest level).

Reset Vector

RISC-V specifies two potential reset vectors: one at a low address, and one at a high address. Tiny RISC-V uses the low address reset vector at 0x00000200. This is where assembly tests should reside, as well as user code in TinyRV2.

Control/Status Registers

Tiny RISC-V includes four non-standard CSRs. `numcores` has the same meaning as the standard CSR `mhartid`, so we make `numcores` a synonym for `mhartid`. Here is the mapping:

CSR Name	Privilege	Read/Write?	CSR Number	Notes
<code>proc2mngnr</code>	M	W	0x7c0	Non-standard
<code>mngnr2proc</code>	M	R	0xfc0	Non-standard
<code>coreid</code>	M	R	0xf14	Synonym of <code>mhartid</code> (hardware thread ID)
<code>numcores</code>	M	R	0xfc1	Non-standard
<code>stats_en</code>	M	RW	0x7c1	Non-standard

These are chosen to conform to the guidelines in Section 2.1 of the RISC-V Volume 2 (Privileged) ISA manual. Here is a description of each of these five CSRs:

- **`proc2mngnr` (0x7c0)**: Used to communicate data from the processor to the manager. This register has register-mapped FIFO-enqueue semantics meaning writing the register essentially enqueues the data on the tail of a FIFO. Writing the register will stall if the FIFO is not ready. Reading the register is undefined.
- **`mngnr2proc` (0xfc0)**: Used to communicate data from the manager to the processor. This register has register-mapped FIFO-dequeue semantics, meaning reading the register essentially dequeues the data from the head of a FIFO. Reading the register will stall if the FIFO has no valid data. Writing the register is undefined.
- **`coreid` (0xf14)**: Used to communicate the core ID in a multi-core system. Writing the register is undefined.
- **`numcores` (0xfc1)**: Used to store the number of cores present in a multi-core system. Writing the register is undefined.
- **`stats_en` (0x7c1)**: Used to enable or disable the statistics tracking feature of the processor (i.e., counting cycles and instructions)

Address Translation

Tiny RISC-V only supports the most basic form of address translation. Every logical address is directly mapped to the corresponding physical address. As mentioned above, Tiny RISC-V only supports a 1MB virtual memory address space from `0x00000000` to `0x000fffff`, and thus Tiny RISC-V only supports a 1MB physical memory address space. In the RISC-V Volume 2 (Privileged) ISA manual, this is called a `Mbare` addressing environment.

6 Tiny RISC-V Pseudo-Instructions

It is very important to understand the relationship between the "real" instructions presented in this manual, the "real" instructions in the official RISC-V ISA manual, and pseudo-instructions. There are four instructions we need to be careful with: NOP, JR, CSRR, CSRW. The following table illustrates which ISAs contain which of these four instructions, and whether or not the instruction is considered a "real" instruction or a "pseudo-instruction":

	TinyRV1	TinyRV2	RISC-V
NOP	pseudo	pseudo	pseudo
JR	real	pseudo	pseudo
CSRR	N/A	real	pseudo
CSRW	N/A	real	pseudo

NOP

NOP is always a pseudo-instruction. It is always equivalent to the following use of the ADDI instruction:

$$\text{nop} \iff \text{addi } x0, x0, 0$$

JR

JR is a "real" instruction in TinyRV1, but it is a pseudo-instruction in TinyRV2 and RISC-V for the following use of the JALR instruction:

$$\text{jr } rs1 \iff \text{jlr } x0, rs1, 0$$

CSRR/CSRW

CSRR and CSRW are real instructions in TinyRV2 but they are pseudo-instructions for the following use of the CSRRS and CSRRW:

$$\begin{aligned} \text{csrr } rd, csr &\iff \text{csrrs } rd, csr, x0 \\ \text{csr } csr, rs1 &\iff \text{csrrw } x0, csr, rs1 \end{aligned}$$

None of this changes the encodings. In TinyRV1, JR is encoded the same way as the corresponding use of the JALR instruction in TinyRV2.