The RISC-V Instruction Set Manual

Volume I: Base User-Level ISA

Version 1.0

Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanović
CS Division, EECS Department, University of California, Berkeley
{waterman|yunsup|pattrsn|krste}@eecs.berkeley.edu

May 13, 2011

1 Introduction

RISC-V is a new instruction set architecture (ISA) designed to support computer architecture research and education. Our goals in defining RISC-V include:

- Provide a *realistic* but *open* ISA that captures important details of commercial general-purpose ISA designs and that is suitable for direct hardware implementation.
- Provide a small but complete base ISA that avoids "over-architecting" for a particular microarchitecture style (e.g., microcoded, in-order, decoupled, out-of-order) or implementation technology (e.g., full-custom, ASIC, FPGA), but which allows efficient implementation in any of these.
- Support both 32-bit and 64-bit address space variants for applications, operating system kernels, and hardware implementations.
- Support highly-parallel multicore or manycore implementations, including heterogeneous multiprocessors.
- Support an efficient dense instruction encoding with variable-length instructions, improving performance and reducing energy and code size.
- Support the revised 2008 IEEE 754 floating-point standard.
- Be fully virtualizable.
- Be simple to subset for educational purposes and to reduce complexity of bringing up new implementations.
- Support experimentation with user-level ISA extensions and specialized variants.
- Support independent experimentation with new supervisor-level ISA designs.

This manual is structured into two volumes. This volume covers the base user-level ISA design and provides examples of possible ISA extensions. The second volume provides examples of supervisor-level ISA design. This manual represents only a snapshot of the RISC-V ISA, which is still under active development; some aspects of the instruction set may change in future revisions.

Commentary on our design decisions is formatted as in this paragraph, and can be skipped if the reader is only interested in the specification itself. The name RISC-V was chosen to represent the fifth major RISC ISA design from UC Berkeley (RISC-I, RISC-II, SOAR, and SPUR were the first four). We also pun on the use of the Roman numeral "V" to signify "variations" and "vectors", as support for a range of architecture research, including various data-parallel accelerators, is an explicit goal of the ISA design.

Our intent is to provide a long-lived open ISA with significant infrastructure support, including documentation, compiler tool chains, operating system ports, reference SAME simulators, cycle-accurate FAME-7 FPGA simulators, high-performance FPGA computers, efficient ASIC *implementations of various target platform designs, configurable processor generators, architecture test suites, and teaching materials. Initial versions of all of these have been developed or are under active development. This material is to be made available under open licenses (either modified BSD or GPL/LGPL).*

2 Base User-Level ISA

This section defines the standard base user-level ISA, which has two variants, RV32 and RV64, providing 32-bit or 64-bit user-level address spaces respectively. Hardware implementations and operating systems might provide only one or both of RV32 and RV64 for user programs. The ISA may be subset by a hardware implementation, but opcode traps and software emulation must then be used to implement functionality not provided by hardware. The base ISA may be extended with new instructions, but the base instructions cannot be redefined. Several standard extensions have been defined and are described in subsequent sections.

Although 64-bit address spaces are a requirement for larger systems, we believe 32-bit address spaces will remain adequate for many embedded and client devices for decades to come and will be desirable to lower memory traffic and energy consumption. In addition, 32-bit address spaces are sufficient for educational purposes.

2.1 Base Programmers' Model

Figure 1 shows the base user-visible state in a RISC-V CPU. There are 31 general-purpose registers `x1`–`x31`, which hold fixed-point values. Register `x0` is hardwired to the constant 0. For RV64, the `x` registers are 64 bits wide, and for RV32, they are 32 bits wide. This document uses the term XPRLEN to refer to the current width of an `x` register in bits (either 32 or 64). Additionally, there are 32 64-bit registers `f0`–`f31`, which hold single- or double-precision floating-point values.

There are also two special user-visible registers defined in the architecture. The program counter `pc` holds the address of the current instruction. The floating-point status register `fsr` contains the operating mode and exception status of the floating-point unit.

We considered a unified register file for both fixed-point and floating-point values as this simplifies software register allocation and calling conventions, and reduces total user state. However, a split organization increases the total number of registers accessible with a given instruction width, simplfies provision of enough regfile ports for wide superscalar issue, supports decoupled floating-point unit architectures, and simplifies use of internal floating-point encoding techniques. Compiler support and calling conventions for split register file architectures are well understood, and using dirty bits on floating-point register file state can reduce context-switch overhead.

The number of available architectural registers can have large impacts on performance and energy consumption. For the base ISA, we chose a conventional size of 32 integer plus 32 floating-point registers based on the behavior of standard compilers on existing code. Register usage tends to be dominated by a few frequently accessed registers, and regfile implementations can be optimized to reduce access energy for the frequently accessed registers. The optional compressed 16-bit instruction format mostly only accesses 8 registers, while instruction-set extensions could support a much larger register space (either flat or hierarchical) if desired.

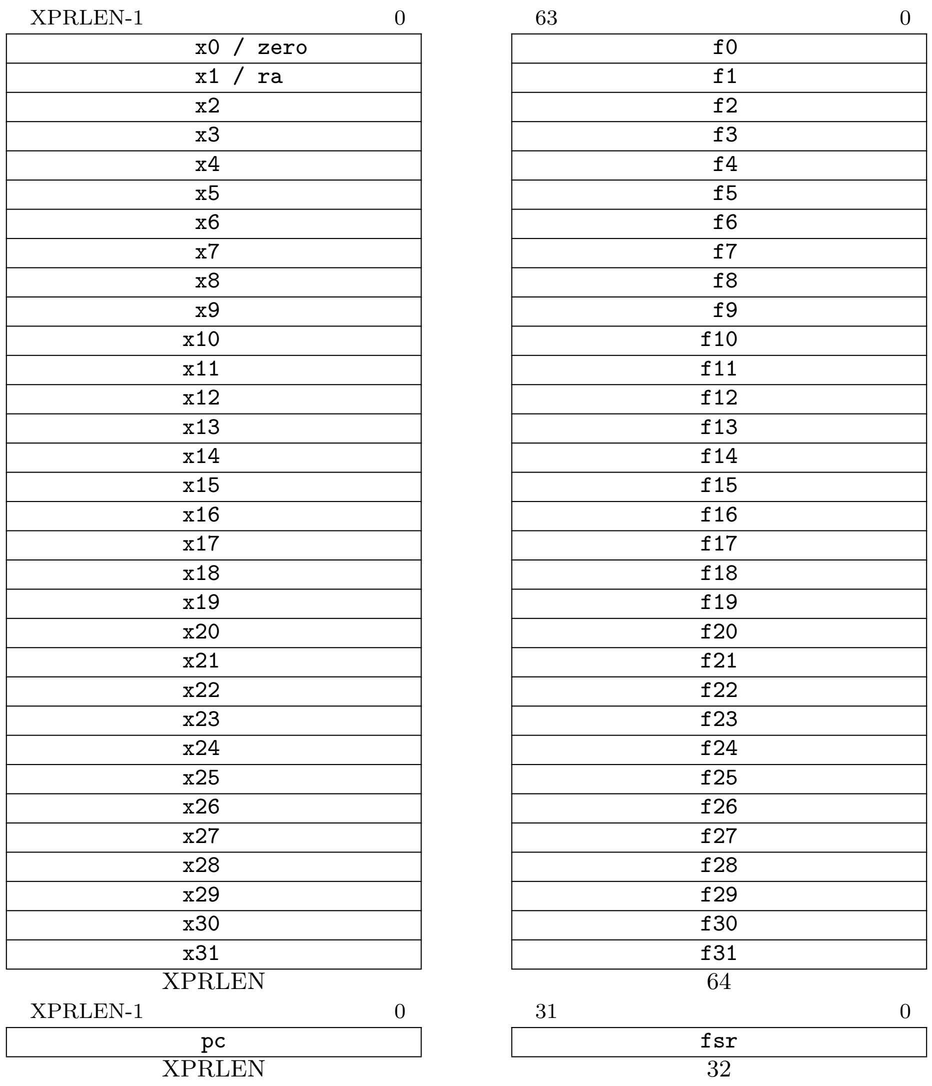

Figure 1: RISC-V base user-level programmer state.

2.2 Instruction Length Encoding

The base RISC-V ISA has fixed-length 32-bit instructions that must be naturally aligned on 32-bit boundaries. However, the RISC-V encoding scheme is designed to support ISA extensions with variable-length instructions, where each instruction can be any number of 16-bit instruction *parcels* in length and parcels are naturally aligned on 16-bit boundaries. A standard compressed ISA extension described in the following section reduces code size by providing compressed 16-bit instructions and relaxes the alignment constraints to allow all instructions (16 bit and 32 bit) to be aligned on any 16-bit boundary to improve code density.

Figure 2 illustrates the RISC-V instruction length encoding convention. All the 32-bit instructions in the base ISA have their lowest two bits set to `11`. The compressed 16-bit instruction-set extensions have their lowest two bits equal to `00`, `01`, or `10`. Instruction-set extensions encoded with more than 32 bits have additional low-order bits set to `1`.

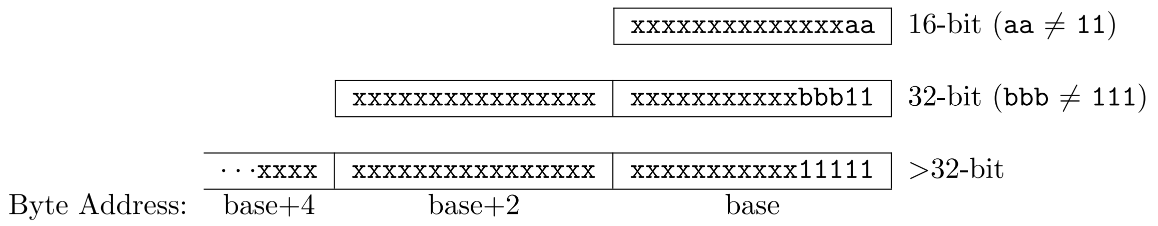

Figure 2: RISC-V instruction length encoding.

RISC-V can be implemented with either big-endian or little-endian memory systems. Instructions are stored in memory with each 16-bit parcel stored in a memory halfword according to the implementation's natural endianess. Parcels comprising one instruction are stored at increasing halfword addresses, with the lowest addressed parcel holding the lowest numbered bits in the instruction specification, i.e., instructions are always stored in a little-endian sequence of parcels regardless of the memory system endianess. The code sequence in Figure 3 will store a 32-bit instruction to memory correctly regardless of memory system endianess.

```
// Store 32-bit instruction in x2 register to location pointed to by x3.
sh   x2, 0(x3)    // Store low bits of instruction in first parcel.
srli x2, x2, 16   // Move high bits down to low bits, overwriting x2.
sh   x2, 2(x3)    // Store high bits in second parcel.
```

Figure 3: Recommended code sequence to store 32-bit instruction from register to memory. Operates correctly on both big- and little-endian memory systems and avoids misaligned accesses when used with variable-length instruction-set extensions.

Given the code size and energy savings of a compressed format, we wanted to build in support for a compressed format to the base ISA rather than adding this as an afterthought, but to allow simpler implementations we didn't want to make the compressed format mandatory. We also wanted to optionally allow longer instructions to support experimentation and instruction-set extensions. Although our encoding convention reduces opcode space for the base 32-bit ISA, 32-bit RISC ISAs are generally very loosely encoded, and our scheme simplifies hardware for variable-length instructions, which support a much larger potential instruction encoding space.

A base implementation need only hold the most-significant 30 bits in instruction caches (a 6.25% saving). On instruction cache refills, any instructions encountered with either low bit clear should be recoded into trap instructions before storing in the cache to preserve illegal instruction trap behavior.

We have to fix the order in which parcels are stored in memory, independent of memory system endianess, to ensure that the length-encoding bits always appear first in halfword address order. This allows the length of a variable-length instruction to be quickly determined by an instruction fetch unit by examining only the first few bits of the first 16-bit instruction parcel. The parcel ordering could have been fixed to be either big-endian (most-significant parcel first) or little-endian (least-significant parcel first). We chose to fix the parcel order to be little-endian, as little-endian systems are currently dominant commercially (all x86 systems; iOS, Android, and Windows for ARM). Once we had decided to fix on a little-endian instruction parcel ordering, this naturally led to placing the length-encoding bits in the LSB positions of the instruction format to avoid breaking up opcode fields.

2.3 Base Instruction Formats

In the base ISA, there are six basic instruction formats as shown in Table 1. These are a fixed 32 bits in length, and must be aligned on a four-byte boundary in memory. An instruction address misaligned exception is generated if the PC is not four-byte aligned on an instruction fetch.

31 27	26 22	21 17	16 12	11 10	9 7	6 0	
rd	rs1	rs2	funct10			opcode	R-type
rd	rs1	rs2	rs3	funct5		opcode	R4-type
rd	rs1	imm[11:7]	imm[6:0]		funct3	opcode	I-type
imm[11:7]	rs1	rs2	imm[6:0]		funct3	opcode	B-type
rd	LUI immediate[19:0]					opcode	L-type
jump offset [24:0]						opcode	J-type

Table 1: RISC-V base instruction formats.

R-Type

31 27	26 22	21 17	16 7	6 0
rd	rs1	rs2	funct10	opcode
5	5	5	10	7

R-type instructions specify two source registers (*rs1* and *rs2*) and a destination register (*rd*). The *funct10* field is an additional opcode field.

R4-Type

31 27	26 22	21 17	16 12	11 7	6 0
rd	rs1	rs2	rs3	funct5	opcode
5	5	5	5	5	7

R4-type instructions specify three source registers (*rs1*, *rs2*, and *rs3*) and a destination register (*rd*). The *funct5* field is a second opcode field. This format is only used by the floating-point fused multiply-add instructions.

I-Type

31 27	26 22	21 17	16 10	9 7	6 0
rd	rs1	imm[11:7]	imm[6:0]	funct3	opcode
5	5	5	7	3	7

I-type instructions specify one source register (*rs1*) and a destination register (*rd*). The second source operand is a sign-extended 12-bit immediate, encoded contiguously in bits 21–10. The *funct3* field is a second opcode field.

B-Type

31 27	26 22	21 17	16 10	9 7	6 0
imm[11:7]	rs1	rs2	imm[6:0]	funct3	opcode
5	5	5	7	3	7

B-type instructions specify two source registers (*rs1* and *rs2*) and a third source operand encoded as a sign-extended 12-bit immediate. The immediate is encoded as the concatenation of the upper 5 bits in bits 31–27, and a lower 7 bits in bits 16–10. The *funct3* field is a second opcode field.

L-Type

31 27	26 7	6 0
rd	LUI immediate[19:0]	opcode
5	20	7

L-type instructions specify a destination register (*rd*) and a 20-bit immediate value. `lui` is the only instruction of this format.

J-Type

31 7	6 0
Jump offset[24:0]	opcode
25	7

J-type instructions encode a 25-bit jump target address as a PC-relative offset. The 25-bit immediate value is shifted left one bit and added to the current PC to form the target address.

Decoding register specifiers is usually on the critical paths in implementations, and so the instruction format was chosen to keep all register specifiers at the same position in all formats at the expense of having to move low immediate bits across some formats (a property shared with SPUR aka. RISC-IV). We also took the opportunity to pack all opcode-related fields (opcode + functX) together at the low end of the word.

In practice, most immediates are small or require all 32 bits (or all 64 bits). We chose an asymmetric immediate split (12 bits in regular instructions plus a special load upper immediate instruction with 20 bits) to increase the opcode space available for regular instructions. In addition, the ISA only has sign-extended immediates. We did not observe a benefit to using zero-extension for some immediates and wanted to keep the ISA as simple as possible.

Major Opcode Map

Table 2 shows a map of the major opcodes for the base ISA.

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32)
00	LOAD	LOAD-FP			OP-IMM		OP-IMM-32	🟥
01	STORE	STORE-FP	AMO	MISC-MEM	OP	LUI	OP-32	🟥
10	MADD	MSUB	NMSUB	NMADD	OP-FP			🟥
11	BRANCH	J	JALR	JAL		SYSTEM		🟥

Table 2: RISC-V base opcode map, inst[1:0]=11

2.4 Load and Store Instructions

RISC-V provides a byte-addressed user memory address space and is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers. The memory system can be either big-endian or little-endian depending on the implementation. Byte addresses are 64 bits wide for RV64, and 32 bits wide for RV32.

31 27	26 22	21 17	16 10	9 7	6 0
rd	rs1	imm[11:7]	imm[6:0]	funct3	opcode
5	5	5	7	3	7
dest	base	offset[11:0]		width	LOAD
dest	base	offset[11:0]		width	LOAD-FP

31 27	26 22	21 17	16 10	9 7	6 0
imm[11:7]	rs1	rs2	imm[6:0]	funct3	opcode
5	5	5	7	3	7
offset[11:7]	base	src	offset[6:0]	width	STORE
offset[11:7]	base	src	offset[6:0]	width	STORE-FP

Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format, and stores are B-type. The effective byte address is obtained by adding register *rs1* to the sign-extended immediate. Loads write to register *rd* a value in memory. Stores write to memory the value in register *rs2*.

The LD instruction loads a 64-bit value from memory into register *rd* for RV64. LD is illegal for RV32. The LW instruction loads a 32-bit value from memory for RV32, and sign-extends this to 64 bits before storing it in register *rd* for RV64. The LWU instruction, on the other hand, zero-extends the 32-bit value from memory for RV64, but is illegal for RV32. LH and LHU are defined analogously for 16-bit values, as are LB and LBU for 8-bit values. The SD, SW, SH, and SB instructions store 64-bit, 32-bit, 16-bit, and 8-bit values in register *rd* to memory, with SD only being valid for RV64.

The FLD instruction loads a 64-bit double-precision floating-point value from memory into floating-point register *rd*, and the FLW instruction loads a 32-bit single-precision floating-point value. FSD and FSW store double- and single-precision values, respectively, from floating-point registers to memory.

For best performance, the effective address for all loads and stores should be naturally aligned for each data type (i.e., on an eight-byte boundary for 64-bit accesses, a four-byte boundary for 32-bit accesses, and a two-byte boundary for 16-bit accesses). The base ISA supports misaligned accesses, but these might run extremely slowly depending on the implementation. Furthermore, naturally aligned loads and stores are guaranteed to execute atomically, whereas misaligned loads and stores might not, and hence require additional synchronization to ensure atomicity.

Misaligned accesses are occasionally required when porting legacy code, and are essential for good performance on many applications when using any form of packed SIMD extension. Our rationale for supporting misaligned accesses via the regular load and store instructions is to simplify *the addition of misaligned hardware support. One option would have been to disallow misaligned accesses in the base ISA and then provide some separate ISA support for misaligned accesses, either special instructions to help software handle misaligned accesses or a new hardware addressing mode for misaligned accesses. Special instructions are difficult to use, complicate the ISA, and often add new processor state (e.g., SPARC VIS align address offset register) or complicate access to existing processor state (e.g., MIPS LWL/LWR partial register writes). In addition, for loop-oriented packed SIMD code, the extra overhead when operands are misaligned motivates software to provide multiple forms of loop depending on operand alignment, which complicates code generation and adds to startup overhead. New misaligned hardware addressing modes take considerable space in the instruction encoding or require very simplified addressing modes (e.g., register indirect only).*

We do not mandate atomicity for misaligned accesses so simple implementations can just use a machine trap and software handler to handle misaligned accesses. If hardware misaligned support is provided, software can exploit this by simply using regular load and store instructions. Hardware can automatically optimize accesses depending on whether runtime addresses are aligned.

Atomic Memory Operation Instructions

31 27	26 22	21 17	16 10	9 7	6 0
rd	rs1	rs2	funct7	funct3	opcode
5	5	5	7	3	7
dest	addr	src	operation	width	AMO

The atomic memory operation (AMO) instructions perform read-modify-write operations for multiprocessor synchronization and are encoded with an R-type instruction format. These AMO instructions atomically load a data value from the address in *rs1*, place the value into register *rd*, apply a binary operator to the loaded value and the value in *rs2*, then store the result back to the address in *rs1*. AMOs can either operate on 32-bit or 64-bit words in memory. For RV64, 32-bit AMOs always sign-extend the value placed in *rd*. The address held in *rs1* must be naturally aligned to the size of the operand (i.e., eight-byte aligned for 64-bit words and four-byte aligned for 32-bit words). If the address is not naturally aligned, a misaligned address trap will be generated.

The operations supported are integer add, logical AND, logical OR, swap, and signed and unsigned integer maximum and minimum.

Even uniprocessor systems need atomic instructions to support operating systems. We selected fetch-and-op style synchronization primitives for the base ISA as they guarantee forward progress unlike compare-and-swap (CAS) or load-linked/store-conditional (LLSC) constructs, and scale better to highly parallel systems. CAS or LLSC could help in the implementation of lock-free data structures, but CAS suffers from the ABA problem and would require a new integer instruction format to support three source operands (address, compare value, swap value) as well as a different memory system message format. LLSC can avoid the ABA problem but is more susceptible to livelock, and implementations usually impose strict constraints or prohibit access to other memory locations while a reservation is held.

In general, a multi-word atomic primitive is desirable but there is still considerable debate about what form this should take. Our current thoughts are to include a small limited-capacity transactional memory buffer along the lines of the original transactional memory proposals.

A simple microarchitecture can implement AMOs by locking a private cache line for the duration. More complex implementations might also implement AMOs at memory controllers, and can optimize away fetching the original value when the destination is `x0`.

2.5 Integer Computational Instructions

Integer computational instructions are either encoded as register-immediate operations using the I-type format or as register-register operations using the R-type format. The destination is register *rd* for both register-immediate and register-register instructions. No integer computational instructions cause arithmetic traps.

Most integer instructions operate on XPRLEN bits of values held in the fixed-point register file. Additional instruction variants are provided to manipulate 32-bit values in RV64. These are indicated with a 'W' suffix to the opcode; they ignore the upper 32 bits of their inputs and always produce 32-bit signed values, i.e. bits XPRLEN-1 through 31 are equal. These instructions cause an illegal instruction trap in RV32.

Integer Register-Immediate Instructions

31 27	26 22	21 17	16 10	9 7	6 0
rd	rs1	imm[11:7]	imm[6:0]	funct3	opcode
5	5	5	7	3	7
dest	src	immediate[11:0]		ADDI/SLTI[U]	OP-IMM
dest	src	immediate[11:0]		ANDI/ORI/XORI	OP-IMM
dest	src	immediate[11:0]		ADDIW	OP-IMM-32

ADDI and ADDIW add the sign-extended 12-bit immediate to register *rs1*. ADDIW is an RV64-only instruction that produces the proper sign-extension of a 32-bit result. Note, ADDIW *rd, rs1, 0* writes the sign-extension of the lower 32 bits of register *rs1* into register *rd*.

SLTI (set less than immediate) places the value 1 in register *rd* if register *rs1* is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to *rd*. SLTIU is similar but compares the values as unsigned numbers.

ANDI, ORI, XORI are logical operations that perform bit-wise AND, OR, and XOR on register *rs1* and the sign-extended 12-bit immediate and place the result in *rd*. Note, XORI *rd, rs1, -1* performs a logical inversion (NOT) of register *rs1*.

31 27	26 22	21 16	15	14 10	9 7	6 0
rd	rs1	imm[11:6]	imm[5]	imm[4:0]	funct3	opcode
5	5	6	1	5	3	7
dest	src	SRA/SRL	shamt[5]	shamt[4:0]	SRxI	OP-IMM
dest	src	SRA/SRL	0	shamt[4:0]	SRxIW	OP-IMM-32
dest	src	0	shamt[5]	shamt[4:0]	SLLI	OP-IMM
dest	src	0	0	shamt[4:0]	SLLIW	OP-IMM-32

Shifts by a constant are also encoded as a specialization of the I-type format. The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 6 bits of the immediate field for RV64, and in the lower 5 bits for RV32. The shift type is encoded in the upper bits of the immediate field.

SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits). In RV32, SLLI, SRLI, and SRAI generate an illegal instruction trap if $imm[5] \neq 0$.

SLLIW, SRLIW, and SRAIW are RV64-only instructions that are analogously defined but operate on 32-bit values and produce signed 32-bit results. SLLIW, SRLIW, and SRAIW generate an illegal instruction trap if $imm[5] \neq 0$.

31 27	26 7	6 0
rd	immediate[19:0]	opcode
5	20	7
dest	20-bit upper immediate	LUI

LUI (load upper immediate) is used to build 32-bit constants. LUI shifts the 20-bit immediate left 12 bits, filling in the vacated bits with zeros, then places the result in register *rd*. For RV64, the 32-bit result is sign-extended to 64 bits.

Integer Register-Register Operations

RISC-V defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct* field selects the type of operation.

31 27	26 22	21 17	16 7	6 0
rd	rs1	rs2	funct10	opcode
5	5	5	10	7
dest	src1	src2	ADD/SUB/SLT/SLTU	OP
dest	src1	src2	AND/OR/XOR	OP
dest	src1	src2	SLL/SRL/SRA	OP
dest	src1	src2	ADDW/SUBW	OP-32
dest	src1	src2	SLLW/SRLW/SRAW	OP-32

ADD and SUB perform addition and subtraction respectively. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. AND, OR, and XOR perform bitwise logical operations.

ADDW and SUBW are RV64-only instructions that are defined analogously to ADD and SUB but operate on 32-bit values and produce signed 32-bit results.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in register *rs2*. In RV64, only the low 6 bits of *rs2* are considered for the shift amount. Similarly for RV32, only the low 5 bits of *rs2* are considered.

SLLW, SRLW, and SRAW are RV64-only instructions that are analogously defined but operate on 32-bit values and produce signed 32-bit results. The shift amount is given by *rs2[4:0]*.

31 27	26 22	21 17	16 7	6 0
rd	rs1	rs2	funct10	opcode
5	5	5	10	7
dest	src1	src2	MUL/MULH[[S]U]	OP
dest	dividend	divisor	DIV[U]/REM[U]	OP
dest	src1	src2	MUL[U]W	OP-32
dest	dividend	divisor	DIV[U]W/REM[U]W	OP-32

MUL performs an XPRLEN-bit$\times$XPRLEN-bit multiplication and places the lower XPRLEN bits in the destination register. MULH, MULHU, and MULHSU perform the same multiplication but return the upper XPRLEN bits of the full 2$\times$XPRLEN-bit product, for signed$\times$signed, unsigned$\times$unsigned, and signed$\times$unsigned multiplication respectively. If both the high and low bits of the same product are required, then the recommended code sequence is: MULH[[S]U] *rdh, rs1, rs2*; MUL *rdl, rs1, rs2* (source register specifiers must be in same order and *rdh* cannot be the same as *rs1* or *rs2*). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies.

MULW is an RV64-only instruction that multiplies the lower 32 bits of the source registers, placing the sign-extension of the lower 32 bits of the result into the destination register. MUL can be used to obtain the upper 32 bits of the 64-bit product, but signed arguments must be proper 32-bit signed values, whereas unsigned arguments must have their upper 32 bits clear.

DIV and DIVU perform signed and unsigned integer division of XPRLEN bits by XPRLEN bits. REM and REMU provide the remainder of the corresponding division operation. If both the quotient and remainder are required from the same division, the recommended code sequence is: DIV[U] *rdq, rs1, rs2*; REM[U] *rdr, rs1, rs2* (*rdq* cannot be the same as *rs1* or *rs2*). Microarchitectures can then fuse these into a single divide operation instead of performing two separate divides.

DIVW and DIVUW are RV64-only instructions that divide the lower 32 bits *rs1* by the lower 32 bits of *rs2*, treating them as signed and unsigned integers respectively, placing the 32-bit quotient in *rd*. REMW and REMUW are RV64-only instructions that provide the corresponding signed and unsigned remainder operations respectively.

The quotient of division by 0 has all bits set, i.e. $2^{XPRLEN} - 1$ for unsigned division or -1 for signed division. The remainder of division by 0 equals the dividend. Signed division overflow occurs only when the most-negative integer, $-(2^{XPRLEN-1})$, is divided by -1. The quotient of signed division overflow is equal to the dividend, and the remainder is 0.

2.6 Control Transfer Instructions

RISC-V provides two types of control transfer instructions: unconditional jumps and conditional branches. Control transfer instructions in RISC-V do *not* have architecturally visible delay slots.

Unconditional Jumps

Absolute jumps (J) and jump and link (JAL) instructions use the J-type format. The 25-bit jump target offset is sign-extended and shifted left one bit to form a byte offset, then added to the `pc` to form the jump target address. Jumps can therefore target a ±32 MB range. JAL stores the address of the instruction following the jump (`pc`+4) into register `x1`.

31 7	6 0
Jump offset[24:0]	opcode
25	7
target offset	J/JAL

The indirect jump instruction JALR (jump and link register) uses the I-type encoding. It has three variants that are functionally identical but provide hints to the implementation: JALR.C is used to call subroutines; JALR.R is used to return from subroutines; and JALR.J is used for indirect jumps. The target address is obtained by sign-extending the 12-bit immediate then adding it to the address contained in register *rs1*. The address of the instruction following the jump (`pc`+4) is written to register *rd*. Register `x0` can be used as the destination if the result is not required.

The JALR major opcode is also used to encode the RDNPC instruction, which writes the address of the following instruction (`pc`+4) to register *rd* without changing control flow.

31 27	26 22	21 17	16 10	9 7	6 0
rd	rs1	imm[11:7]	imm[6:0]	funct3	opcode
5	5	5	7	3	7
dest	base	offset[11:7]	offset[6:0]	C/R/J	JALR
dest	0	0	0	RDNPC	JALR

The unconditional jump instructions all use PC-relative addressing to help support position-independent code. The JALR instruction was defined to enable a two-instruction sequence to jump anywhere in a 32-bit address range. A LUI instruction can first load rs1 *with the upper 20 bits of a target address, then JALR can add in the lower bits.*

Note that the JALR instruction does not shift the 12-bit immediate by one bit, unlike the conditional branch instructions. This is to allow the same linker relocation format to be used for JALR as for global loads and stores. For implementations with dedicated branch target address adders, this is only a minor inconvenience, as some of the immediate field is already in a different position than for conditional branches. For implementations that use the execute-stage adders to perform jump target arithmetic, this reuses the same datapath required for load address calculations.

The JALR hints are used to guide an implementation's instruction-fetch predictors, indicating whether JALR instructions should push (C), pop (R), or not touch (J/RDNPC) a return-address stack.

Conditional Branches

All branch instructions use the B-type encoding. The 12-bit immediate is sign-extended, shifted left one bit, then added to the current `pc` to give the target address.

31 27	26 22	21 17	16 10	9 7	6 0
imm[11:7]	rs1	rs2	imm[6:0]	funct3	opcode
5	5	5	7	3	7
offset[11:7]	src1	src2	offset[6:0]	BEQ/BNE	BRANCH
offset[11:7]	src1	src2	offset[6:0]	BLT[U]	BRANCH
offset[11:7]	src1	src2	offset[6:0]	BGE[U]	BRANCH

Branch instructions compare two registers. BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively. BLT and BLTU take the branch if *rs1* is less than *rs2*, using signed and unsigned comparison respectively. BGE and BGEU take the branch if *rs1* is greater than or equal to *rs2*, using signed and unsigned comparison respectively. Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

Software should be optimized such that the sequential code path is the most common path, with less-frequently-taken code paths placed out of line. Software should also assume that backward branches will be predicted taken and forward branches as not-taken, at least the first time they are encountered. Dynamic predictors should quickly learn any predictable branch behavior.

The conditional branches were designed to include arithmetic comparison operations between two registers, rather than use condition codes (x86, ARM, SPARC, PowerPC), or to only compare one register against zero (Alpha, MIPS), or two registers only for equality (MIPS). This design was motivated by the observation that a combined compare-and-branch instruction fits into a regular pipeline, avoids additional condition code state or use of a temporary register, and reduces static code size and dynamic instruction fetch traffic. Another point is that comparisons against zero require non-trivial circuit delay (especially after the move to static logic in advanced processes) and so are almost as expensive as arithmetic magnitude compares. Another advantage of a fused compare-and-branch instruction is that branches are observed earlier in the front-end instruction stream, and so can be predicted earlier. There is perhaps an advantage to a design with condition codes in the case where multiple branches can be taken based on the same condition codes, but we believe this case to be relatively rare.

We considered but did not include static branch hints in the instruction encoding. These can reduce the pressure on dynamic predictors, but require more instruction encoding space and software profiling for best results.

We considered but did not include conditional moves or predicated instructions, which can effectively replace unpredictable short forward branches. Conditional move and predicated instructions cause complications in out-of-order microarchitectures, due to the need to copy the original value of the destination architectural register into the renamed destination physical register if the predicate is false, adding an implicit third source operand. Predicates also add additional user state and require additional instruction encoding space.

2.7 Floating-Point Instructions

The base RISC-V ISA provides both single- and double-precision floating-point computational instructions compliant with the IEEE 754-2008 floating-point arithmetic standard. Most floating-point instructions operate on values in the 32-entry floating-point register file. Floating-point load and store instructions transfer floating-point values between registers and memory, as described in Section 2.4. Instructions to transfer values to and from the fixed-point register file are also provided.

Floating-Point Status Register

32 8	7 5	4	3	2	1	0
0	Rounding Mode	Accrued Exceptions				
		NV	DZ	OF	UF	NX
24	3	1	1	1	1	1

Figure 4: Floating-point status register.

The `fsr` register is a 32-bit read/write register that selects the dynamic rounding mode for floating-point arithmetic operations and holds the accrued exception flags. The `fsr` is read and written with the MFFSR and MTFSR floating-point instructions, described below.

Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in the Rounding Mode field and encoded as shown in Table 3. If the Rounding Mode field is set to an invalid value (101–111), any subsequent attempt to execute a floating-point operation with a dynamic rounding mode will cause an illegal instruction trap. Some instructions are never affected by rounding mode, and should have their *rm* field set to RNE (000).

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round toward Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101–111		*Invalid.*

Table 3: Rounding Mode field encoding.

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software, as shown in Table 4.

NaN Generation and Propagation

If a floating-point operation on non-NaN inputs is invalid, e.g. $\sqrt{-1.0}$, the result is the canonical NaN: the sign bit is 0, and the fraction and exponent have all bits set. As the MSB of the significand (aka. the quiet bit) is set, the canonical NaN is quiet.

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

Table 4: Current and accrued exception flag encoding.

With the exception of the FMIN and FMAX operations, if a floating-point operation has at least one signaling NaN input, the first such input (*rs1*, *rs2*, or *rs3*, in that order) is returned with its quiet bit set. Otherwise, if a floating-point operation has at least one quiet NaN input, the first such input is returned.

For FMIN and FMAX, if at least one input is a signaling NaN, the first such input is returned with its quiet bit set. If both inputs are quiet NaNs, the first input is returned. If just one input is a quiet NaN, the non-NaN input is returned.

If a NaN value is converted to a larger floating-point type, the significand of the input becomes the MSBs of the significand of the output; the LSBs are cleared. If a NaN value is converted to a smaller floating-point type, the LSBs of the significand are discarded. In both cases, the quiet bit of the output is set, even for signaling NaN inputs.

Floating-Point Computational Instructions

Floating-point arithmetic instructions with one or two source operands use the R-type format with the OP-FP major opcode. FADD.*fmt*, FSUB.*fmt*, FMUL.*fmt*, and FDIV.*fmt* perform floating-point addition, subtraction, multiplication, and division, respectively, between *rs1* and *rs2*, writing the result to *rd*. FMIN.*fmt* and FMAX.*fmt* write, respectively, the smaller or larger of *rs1* and *rs2* to *rd*. FSQRT.*fmt* computes the square root of *rs1* and writes the result to *rd*. The *fmt* field encodes the datatype of the operands and destination: S for single-precision or D for double-precision.

All floating-point operations that perform rounding can select the rounding mode statically using the *rm* field with the same encoding as shown in Table 3. A value of 111 in the instruction's *rm* field selects the dynamic rounding mode held in the `fsr`. Any attempt to execute a floating-point operation that performs rounding with an invalid value for *rm*, or with dynamic rounding and an invalid value for *rm* in the `fsr`, will cause an illegal instruction trap.

31 27	26 22	21 17	16 12	11 9	8 7	6 0
rd	rs1	rs2	funct	rm	fmt	opcode
5	5	5	5	3	2	7
dest	src1	src2	FADD/FSUB	RM	S/D	OP-FP
dest	src1	src2	FMUL/FDIV	RM	S/D	OP-FP
dest	src1	src2	FMIN/FMAX	000	S/D	OP-FP
dest	src	0	FSQRT	RM	S/D	OP-FP

Floating-point fused multiply-add instructions are encoded as R4-type instructions and multiply the values in *rs1* and *rs2*, optionally negate the result, then add or subtract the value in *rs3* to or from that result. FMADD.*fmt* computes *rs1*×*rs2*+*rs3*; FMSUB.*fmt* computes *rs1*×*rs2*-*rs3*; FNMSUB.*fmt* computes -*(rs1*×*rs2*-*rs3)*; and FNMADD.*fmt* computes -*(rs1*×*rs2*+*rs3)*.

31 27	26 22	21 17	16 12	11 9	8 7	6 0
rd	rs1	rs2	rs3	rm	fmt	opcode
5	5	5	5	3	2	7
dest	src1	src2	src3	RM	S/D	F[N]MADD/F[N]MSUB

The 2-bit floating-point format field *fmt* is encoded as shown in Table 5.

fmt field	Mnemonic	Meaning
00	S	32-bit single-precision
01	D	64-bit double-precision
10	-	*reserved*
11	-	*reserved*

Table 5: Format field encoding.

Floating-Point Conversion and Move Instructions

Floating-point-to-integer and integer-to-floating-point conversion instructions are encoded in the OP-FP major opcode space. The *fmt* field encodes the datatype of the lone floating-point operand. FCVT.W.*fmt* or FCVT.L.*fmt* converts a floating-point number in floating-point register *rs1* to a signed 32-bit or 64-bit integer, respectively, in fixed-point register *rd*. FCVT.*fmt*.W or FCVT.*fmt*.L converts a 32-bit or 64-bit signed integer, respectively, in fixed-point register *rs1* into a floating-point number in floating-point register *rd*. FCVT.WU.*fmt*, FCVT.LU.*fmt*, FCVT.*fmt*.WU, and FCVT.*fmt*.LU variants convert to or from unsigned integer values. FCVT.L[U].*fmt* and FCVT.*fmt*.L[U] are illegal in RV32.

All floating-point to integer and integer to floating-point conversion instructions round according to the *rm* field. Note FCVT.D.W[U] always produces an exact result and is unaffected by rounding mode. A floating-point register can be initialized to floating-point positive zero using FCVT.*fmt*.W *rd*, `x0`, which will never raise any exceptions.

31 27	26 22	21 17	16 12	11 9	8 7	6 0
rd	rs1	rs2	funct	rm	fmt	opcode
5	5	5	5	3	2	7
dest	src	0	FCVT.W[U].*fmt*	RM	S/D	OP-FP
dest	src	0	FCVT.*fmt*.W[U]	RM	S/D	OP-FP
dest	src	0	FCVT.L[U].*fmt*	RM	S/D	OP-FP
dest	src	0	FCVT.*fmt*.L[U]	RM	S/D	OP-FP

The double-precision to single-precision and single-precision to double-precision conversion instructions, FCVT.S.D and FCVT.D.S, are encoded in the OP-FP major opcode space and both the source and destination are floating-point registers. The *fmt* field encodes the datatype of the result. FCVT.S.D rounds according to the RM field; FCVT.D.S will never round.

31 27	26 22	21 17	16 12	11 9	8 7	6 0
rd	rs1	rs2	funct	rm	fmt	opcode
5	5	5	5	3	2	7
dest	src	0	FCVT.S.D	RM	S	OP-FP
dest	src	0	FCVT.D.S	RM	D	OP-FP

Floating-point to floating-point sign-injection instructions, FSGNJ.*fmt*, FSGNJN.*fmt*, and FSGNJX.*fmt*, produce a result that takes all bits except the sign bit from *rs1*. For FSGNJ, the result's sign bit is *rs2*'s sign bit; for FSGNJN, the result's sign bit is the opposite of *rs2*'s sign bit; and for FSGNJX, the sign bit is the XOR of the sign bits of *rs1* and *rs2*. Sign-injection instructions do not set floating-point exception flags. Note, FSGNJ *rx, ry, ry* moves *ry* to *rx*; FSGNJN *rx, ry, ry* moves the the negation of *ry* to *rx*; and FSGNJX *rx, ry, ry* moves the absolute value of *ry* to *rx*.

31 27	26 22	21 17	16 12	11 9	8 7	6 0
rd	rs1	rs2	funct	rm	fmt	opcode
5	5	5	5	3	2	7
dest	src1	src2	FSGNJ[N]	000	S/D	OP-FP
dest	src1	src2	FSGNJX	000	S/D	OP-FP

Instructions are provided to move bit patterns between the floating-point and fixed-point registers. MFTX.S moves the single-precision value in floating-point register *rs2* represented in IEEE 754-2008 encoding to the lower 32 bits of fixed-point register *rd*. For RV64, the higher 32 bits of the destination register are filled with copies of the floating-point number's sign bit. MXTF.S moves the single-precision value encoded in IEEE 754-2008 standard encoding from the lower 32 bits of fixed-point register *rs1* to the floating-point register *rd*. MFTX.D and MXTF.D are defined analogously for double-precision values in RV64, but are illegal in RV32. RV32 can use stores and loads to transfer double-precision values between fixed-point and floating-point registers.

31 27	26 22	21 17	16 12	11 9	8 7	6 0
rd	rs1	rs2	funct	rm	fmt	opcode
5	5	5	5	3	2	7
dest	0	src	MFTX.*fmt*	000	S/D	OP-FP
dest	src	0	MXTF.*fmt*	000	S/D	OP-FP

The Floating-point Status Register `fsr` can be read and written with the MFFSR and MTFSR instructions. MFFSR copies `fsr` into fixed-point register rd. MTFSR writes `fsr` with the value in fixed-point register *rs1*, and also copies the original value of `fsr` into fixed-point register rd.

31 27	26 22	21 17	16 12	11 9	8 7	6 0
rd	rs1	rs2	funct	rm	fmt	opcode
5	5	5	5	3	2	7
dest	0	0	MFFSR	000	S	OP-FP
dest	src	0	MTFSR	000	S	OP-FP

Floating-Point Compare Instructions

Floating-point compare instructions perform the specified comparison (equal, less than, or less than or equal) between floating-point registers *rs1* and *rs2* and record the boolean result in fixed-point register *rd*.

31 27	26 22	21 17	16 12	11 9	8 7	6 0
rd	rs1	rs2	funct	rm	fmt	opcode
5	5	5	5	3	2	7
dest	src1	src2	FEQ/FLT/FLE.*fmt*	000	S/D	OP-FP

The base floating-point ISA was defined so as to allow implementations to employ an internal recoding of the floating-point format in registers to simplify handling of subnormal values and possibly to reduce functional unit latency. To this end, the base ISA avoids representing integer values in the floating-point registers by defining conversion and comparison operations that read and write the fixed-point register file directly. This also removes many of the common cases where explicit moves between integer and floating-point registers are required, reducing instruction count and critical paths for common mixed-format code sequences.

We require implementations to return the standard-mandated default values in the case of exceptional conditions, without any further intervention on the part of user-level software (unlike *the Alpha ISA floating-point trap barriers). We believe full hardware handling of exceptional cases will become more common, and so wish to avoid complicating the user-level ISA to optimize other approaches.*

As allowed by the standard, we do not support traps on floating-point exceptions in the base ISA, but instead require explicit checks of the flags in software. We are contemplating addition of a branch controlled directly by the contents of the floating-point accrued exception flags to support fast user-level exception handling.

The desire to support IEEE 754-2008 requires the addition of the three-source-operands fused multiply-add instructions, and the fifth rounding mode.

The C99 language standard mandates the provision of a dynamic rounding mode register.

The MTFSR instruction was defined to both read and write the floating-point status register to allow rapid save and restore of floating-point context. The operation MTFSR x0, rd *will save the accrued exception flags and rounding mode in fixed-point register* rd*, then clear the flags.*

2.8 Memory Model

In the base RISC-V ISA, each hardware thread observes its own memory operations as if they executed sequentially in program order. RISC-V has a relaxed memory model between different hardware threads, requiring an explicit FENCE instruction to guarantee any specific ordering between memory operations from different threads.

31 27	26 22	21 10	9 7	6 0
rd	rs1	imm[11:0]	funct3	opcode
5	5	12	3	7
-	-	-	FENCE	MISC-MEM

The FENCE instruction is used to order loads and stores as viewed by other hardware threads. Informally, no other hardware thread can observe any memory operations (LOAD/STORE/AMO) following a FENCE before any memory operations preceding the FENCE.

We chose a relaxed memory model to allow high performance from simple machine implementations. The base ISA provides only a global FENCE operation, but sufficient encoding space is reserved to allow finer-grain FENCE instructions in optional extensions. A base implementation should ignore the higher-order bits in a FENCE instruction and simply execute a conservative global fence to provide forwards compatibility with finer-grain fences.

31 27	26 22	21 10	9 7	6 0
rd	rs1	imm[11:0]	funct3	opcode
5	5	12	3	7
-	-	-	FENCE.I	MISC-MEM

The FENCE.I instruction is used to synchronize the instruction and data streams. RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches until a FENCE.I instruction is executed. A FENCE.I instruction only ensures that a subsequent instruction fetch on the same hardware thread will see any previous data stores. FENCE.I does *not* ensure that other hardware threads' instruction fetches will observe the local thread's stores in a multiprocessor system.

The FENCE.I instruction was designed to support a wide variety of implementations. A simple implementation can flush the local instruction cache and the instruction pipeline when the FENCE.I is decoded. A more complex implementation might snoop the instruction (data) cache on every data (instruction) cache miss, or use an inclusive unified private L2 cache to invalidate lines from the primary instruction cache when they are being written by a local store instruction. If instruction and data caches are kept coherent in this way, then only the pipeline needs to be flushed at a FENCE.I.

Extensions might define finer-grain FENCE.I instructions targeting specific instruction addresses, so a base implementation should ignore the higher-order bits in a FENCE.I instruction and simply execute a conservative local FENCE.I to provide forwards compatibility.

To make a store to instruction memory visible to all hardware threads, the writing thread has to issue a global FENCE before requesting that all remote hardware threads execute a FENCE.I.

We considered but did not include a "store instruction" instruction (as in MAJC). JIT compilers may generate a large trace of instructions before a single FENCE.I, and amortize any instruction cache snooping/invalidation overhead.

2.9 System Instructions

SYSTEM instructions are used to access system functionality that might require privileged access and are encoded as an R-type instruction.

The SYSTEM instructions are defined to allow simpler implementations to always trap to a single software exception handler. More sophisticated implementations might execute more of each system instruction in hardware.

SYSCALL and BREAK

31 27	26 22	21 17	16 7	6 0
rd	rs1	rs2	funct10	opcode
5	5	5	10	7
0	0	0	SYSCALL	SYSTEM
0	0	0	BREAK	SYSTEM

The SYSCALL instruction is used to make a request to an operating system environment. The ABI for the operating system will define how parameters for the OS request are passed, but usually these will be in defined locations in the fixed-point register file.

The BREAK instruction is used by debuggers to cause control to be transferred back to the debugging environment.

Timers and Counters

31 27	26 22	21 17	16 7	6 0
rd	rs1	rs2	funct10	opcode
5	5	5	10	7
dest	0	0	RDCYCLE	SYSTEM
dest	0	0	RDTIME	SYSTEM
dest	0	0	RDINSTRET	SYSTEM

The RDCYCLE instruction writes fixed-point register *dest* with a count of the number of clock cycles executed by the processor on which the hardware thread is running from an arbitrary start time in the past. In RV32, this returns a 32-bit unsigned integer value that will wrap around when the count value overflows (modulo arithmetic). In RV64, this will return a 64-bit unsigned integer value, which will never overflow. The rate at which the cycle counter advances will depend on the implementation and operating environment. The software environment should provide a means to determine the current rate (cycles/second) at which the cycle counter is incrementing.

The RDTIME instruction writes fixed-point register *dest* with an integer value corresponding to the wall-clock real time that has passed from an arbitrary start time in the past. In RV32, this returns a 32-bit unsigned integer value that will wrap around when the time value overflows (modulo arithmetic). In RV64, this will return a 64-bit unsigned integer value, which should never overflow. The software environment should provide a means of determining the period of the real-time counter (seconds/tick). The period must be constant and should be no greater than 100 ns (at least 10 MHz rate). For RV32, the real-time clock period should be no shorter than 10 ns to allow periods of up to 4 seconds to be measured simply. The real-time clocks of all hardware threads in a single user application should be synchronized to within one tick of the real-time clock. The environment should provide a means to determine the accuracy of the clock.

The RDINSTRET instruction writes fixed-point register *dest* with the number of instructions retired by this hardware thread from some arbitrary start point in the past. In RV32, this returns an unsigned 32-bit integer value that will wrap around when the count overflows. In RV64, this returns an unsigned 64-bit integer value that will never overflow.

We mandate these basic counters be provided in all implementations as they are essential for basic performance analysis, adaptive and dynamic optimization, and to allow an application to work with real-time streams. Additional counters should be provided to help diagnose performance problems and these should be made accessible from user-level application code with low overhead.

In some applications, it is important to be able to read multiple counters at the same instant in time. When run under a multitasking environment, a user thread can suffer a context switch while attempting to read the counters. One solution is for the user thread to read the real-time counter before and after reading the other counters to determine if a context switch occured in the middle of the sequence, in which case the reads can be retried. We considered adding output latches to allow a user thread to snapshot the counter values atomically, but this would increase the size of the user context especially for implementations with a richer set of counters.

3 Compressed Instruction Set Extension

The RISC-V compressed instruction set extension reduces static and dynamic code size by adding short 16-bit instruction encodings for common integer operations. The compressed instruction encodings can be added to both RV64 and RV32, forming RVC64 and RVC32 respectively.

The RVC64 and RVC32 ISAs allow 16-bit instructions to be freely intermixed with the 32-bit base instructions, with the latter now able to start on any 16-bit boundary. All of the 16-bit instructions can be expanded into one or more of the base RISC-V instructions.

The RVC ISAs are still under development, but we expect a 25–30% reduction in static and dynamic code size.

31 27	26 22	21 17	16 15 14 12	11 10	9 8 7	6 0	
jump target						opcode	J-type
rd	LUI-immediate					opcode	LUI-type
rd	rs1	imm[11:7]	imm[6:0]		funct3	opcode	I-type
imm[11:7]	rs1	rs2	imm[6:0]		funct3	opcode	B-type
rd	rs1	rs2	funct10			opcode	R-type
rd	rs1	rs2	rs3	funct5		opcode	R4-type

Unimplemented Instruction

Control Transfer Instructions

imm25					1100111	J imm25
imm25					1101111	JAL imm25
imm12hi	rs1	rs2	imm12lo	000	1100011	BEQ rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	001	1100011	BNE rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	100	1100011	BLT rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	101	1100011	BGE rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	110	1100011	BLTU rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	111	1100011	BGEU rs1,rs2,imm12
rd	rs1	imm12		000	1101011	JALR.C rd,rs1,imm12
rd	rs1	imm12		001	1101011	JALR.R rd,rs1,imm12
rd	rs1	imm12		010	1101011	JALR.J rd,rs1,imm12
rd	00000	000000000000		100	1101011	RDNPC rd

Memory Instructions

rd	rs1	imm12		000	0000011	LB rd,rs1,imm12
rd	rs1	imm12		001	0000011	LH rd,rs1,imm12
rd	rs1	imm12		010	0000011	LW rd,rs1,imm12
rd	rs1	imm12		011	0000011	LD rd,rs1,imm12
rd	rs1	imm12		100	0000011	LBU rd,rs1,imm12
rd	rs1	imm12		101	0000011	LHU rd,rs1,imm12
rd	rs1	imm12		110	0000011	LWU rd,rs1,imm12
imm12hi	rs1	rs2	imm12lo	000	0100011	SB rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	001	0100011	SH rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	010	0100011	SW rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	011	0100011	SD rs1,rs2,imm12

Atomic Memory Instructions

rd	rs1	rs2	0000000	010	0101011	AMOADD.W rd,rs1,rs2
rd	rs1	rs2	0000001	010	0101011	AMOSWAP.W rd,rs1,rs2
rd	rs1	rs2	0000010	010	0101011	AMOAND.W rd,rs1,rs2
rd	rs1	rs2	0000011	010	0101011	AMOOR.W rd,rs1,rs2
rd	rs1	rs2	0000100	010	0101011	AMOMIN.W rd,rs1,rs2
rd	rs1	rs2	0000101	010	0101011	AMOMAX.W rd,rs1,rs2
rd	rs1	rs2	0000110	010	0101011	AMOMINU.W rd,rs1,rs2
rd	rs1	rs2	0000111	010	0101011	AMOMAXU.W rd,rs1,rs2
rd	rs1	rs2	0000000	011	0101011	AMOADD.D rd,rs1,rs2
rd	rs1	rs2	0000001	011	0101011	AMOSWAP.D rd,rs1,rs2
rd	rs1	rs2	0000010	011	0101011	AMOAND.D rd,rs1,rs2
rd	rs1	rs2	0000011	011	0101011	AMOOR.D rd,rs1,rs2
rd	rs1	rs2	0000100	011	0101011	AMOMIN.D rd,rs1,rs2
rd	rs1	rs2	0000101	011	0101011	AMOMAX.D rd,rs1,rs2
rd	rs1	rs2	0000110	011	0101011	AMOMINU.D rd,rs1,rs2
rd	rs1	rs2	0000111	011	0101011	AMOMAXU.D rd,rs1,rs2

31 27	26 22	21 17	16 15 14 12	11 10	9 8 7	6 0	
jump target						opcode	J-type
rd	LUI-immediate					opcode	LUI-type
rd	rs1	imm[11:7]	imm[6:0]		funct3	opcode	I-type
imm[11:7]	rs1	rs2	imm[6:0]		funct3	opcode	B-type
rd	rs1	rs2	funct10			opcode	R-type
rd	rs1	rs2	rs3	funct5		opcode	R4-type

Integer Compute Instructions

rd	rs1	imm12		000	0010011	ADDI rd,rs1,imm12
rd	rs1	000000	shamt	001	0010011	SLLI rd,rs1,shamt
rd	rs1	imm12		010	0010011	SLTI rd,rs1,imm12
rd	rs1	imm12		011	0010011	SLTIU rd,rs1,imm12
rd	rs1	imm12		100	0010011	XORI rd,rs1,imm12
rd	rs1	000000	shamt	101	0010011	SRLI rd,rs1,shamt
rd	rs1	000001	shamt	101	0010011	SRAI rd,rs1,shamt
rd	rs1	imm12		110	0010011	ORI rd,rs1,imm12
rd	rs1	imm12		111	0010011	ANDI rd,rs1,imm12
rd	rs1	rs2	0000000	000	0110011	ADD rd,rs1,rs2
rd	rs1	rs2	1000000	000	0110011	SUB rd,rs1,rs2
rd	rs1	rs2	0000000	001	0110011	SLL rd,rs1,rs2
rd	rs1	rs2	0000000	010	0110011	SLT rd,rs1,rs2
rd	rs1	rs2	0000000	011	0110011	SLTU rd,rs1,rs2
rd	rs1	rs2	0000000	100	0110011	XOR rd,rs1,rs2
rd	rs1	rs2	0000000	101	0110011	SRL rd,rs1,rs2
rd	rs1	rs2	1000000	101	0110011	SRA rd,rs1,rs2
rd	rs1	rs2	0000000	110	0110011	OR rd,rs1,rs2
rd	rs1	rs2	0000000	111	0110011	AND rd,rs1,rs2
rd	rs1	rs2	0000001	000	0110011	MUL rd,rs1,rs2
rd	rs1	rs2	0000001	001	0110011	MULH rd,rs1,rs2
rd	rs1	rs2	0000001	010	0110011	MULHSU rd,rs1,rs2
rd	rs1	rs2	0000001	011	0110011	MULHU rd,rs1,rs2
rd	rs1	rs2	0000001	100	0110011	DIV rd,rs1,rs2
rd	rs1	rs2	0000001	101	0110011	DIVU rd,rs1,rs2
rd	rs1	rs2	0000001	110	0110011	REM rd,rs1,rs2
rd	rs1	rs2	0000001	111	0110011	REMU rd,rs1,rs2
rd	imm20				0110111	LUI rd,imm20

32-bit Integer Compute Instructions

rd	rs1	imm12		000	0011011	ADDIW rd,rs1,imm12
rd	rs1	0000000	shamtw	001	0011011	SLLIW rd,rs1,shamtw
rd	rs1	0000000	shamtw	101	0011011	SRLIW rd,rs1,shamtw
rd	rs1	0000010	shamtw	101	0011011	SRAIW rd,rs1,shamtw
rd	rs1	rs2	0000000	000	0111011	ADDW rd,rs1,rs2
rd	rs1	rs2	1000000	000	0111011	SUBW rd,rs1,rs2
rd	rs1	rs2	0000000	001	0111011	SLLW rd,rs1,rs2
rd	rs1	rs2	0000000	101	0111011	SRLW rd,rs1,rs2
rd	rs1	rs2	1000000	101	0111011	SRAW rd,rs1,rs2
rd	rs1	rs2	0000001	000	0111011	MULW rd,rs1,rs2
rd	rs1	rs2	0000001	100	0111011	DIVW rd,rs1,rs2
rd	rs1	rs2	0000001	101	0111011	DIVUW rd,rs1,rs2
rd	rs1	rs2	0000001	110	0111011	REMW rd,rs1,rs2
rd	rs1	rs2	0000001	111	0111011	REMUW rd,rs1,rs2

31 27	26 22	21 17	16 15	14 12	11 10	9	8 7	6 0	
jump target								opcode	J-type
rd	LUI-immediate							opcode	LUI-type
rd	rs1	imm[11:7]	imm[6:0]			funct3		opcode	I-type
imm[11:7]	rs1	rs2	imm[6:0]			funct3		opcode	B-type
rd	rs1	rs2	funct10					opcode	R-type
rd	rs1	rs2	rs3		funct5			opcode	R4-type

Floating-Point Memory Instructions

rd	rs1	imm12		010	0000111	FLW rd,rs1,imm12
rd	rs1	imm12		011	0000111	FLD rd,rs1,imm12
imm12hi	rs1	rs2	imm12lo	010	0100111	FSW rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	011	0100111	FSD rs1,rs2,imm12

Floating-Point Compute Instructions

rd	rs1	rs2	00000	rm	00	1010011	FADD.S rd,rs1,rs2[,rm]
rd	rs1	rs2	00001	rm	00	1010011	FSUB.S rd,rs1,rs2[,rm]
rd	rs1	rs2	00010	rm	00	1010011	FMUL.S rd,rs1,rs2[,rm]
rd	rs1	rs2	00011	rm	00	1010011	FDIV.S rd,rs1,rs2[,rm]
rd	rs1	00000	00100	rm	00	1010011	FSQRT.S rd,rs1[,rm]
rd	rs1	rs2	11000	000	00	1010011	FMIN.S rd,rs1,rs2
rd	rs1	rs2	11001	000	00	1010011	FMAX.S rd,rs1,rs2
rd	rs1	rs2	00000	rm	01	1010011	FADD.D rd,rs1,rs2[,rm]
rd	rs1	rs2	00001	rm	01	1010011	FSUB.D rd,rs1,rs2[,rm]
rd	rs1	rs2	00010	rm	01	1010011	FMUL.D rd,rs1,rs2[,rm]
rd	rs1	rs2	00011	rm	01	1010011	FDIV.D rd,rs1,rs2[,rm]
rd	rs1	00000	00100	rm	01	1010011	FSQRT.D rd,rs1[,rm]
rd	rs1	rs2	11000	000	01	1010011	FMIN.D rd,rs1,rs2
rd	rs1	rs2	11001	000	01	1010011	FMAX.D rd,rs1,rs2
rd	rs1	rs2	rs3	rm	00	1000011	FMADD.S rd,rs1,rs2,rs3[,rm]
rd	rs1	rs2	rs3	rm	00	1000111	FMSUB.S rd,rs1,rs2,rs3[,rm]
rd	rs1	rs2	rs3	rm	00	1001011	FNMSUB.S rd,rs1,rs2,rs3[,rm]
rd	rs1	rs2	rs3	rm	00	1001111	FNMADD.S rd,rs1,rs2,rs3[,rm]
rd	rs1	rs2	rs3	rm	01	1000011	FMADD.D rd,rs1,rs2,rs3[,rm]
rd	rs1	rs2	rs3	rm	01	1000111	FMSUB.D rd,rs1,rs2,rs3[,rm]
rd	rs1	rs2	rs3	rm	01	1001011	FNMSUB.D rd,rs1,rs2,rs3[,rm]
rd	rs1	rs2	rs3	rm	01	1001111	FNMADD.D rd,rs1,rs2,rs3[,rm

31 27	26 22	21 17	16 15 14 12	11 10	9	8 7	6 0	
jump target							opcode	J-type
rd	LUI-immediate						opcode	LUI-type
rd	rs1	imm[11:7]	imm[6:0]		funct3		opcode	I-type
imm[11:7]	rs1	rs2	imm[6:0]		funct3		opcode	B-type
rd	rs1	rs2	funct10				opcode	R-type
rd	rs1	rs2	rs3	funct5			opcode	R4-type

Floating-Point Move & Conversion Instructions

rd	rs1	rs2	00101	000	00	1010011	FSGNJ.S rd,rs1,rs2
rd	rs1	rs2	00110	000	00	1010011	FSGNJN.S rd,rs1,rs2
rd	rs1	rs2	00111	000	00	1010011	FSGNJX.S rd,rs1,rs2
rd	rs1	rs2	00101	000	01	1010011	FSGNJ.D rd,rs1,rs2
rd	rs1	rs2	00110	000	01	1010011	FSGNJN.D rd,rs1,rs2
rd	rs1	rs2	00111	000	01	1010011	FSGNJX.D rd,rs1,rs2
rd	rs1	00000	10001	rm	00	1010011	FCVT.S.D rd,rs1[,rm]
rd	rs1	00000	10000	rm	01	1010011	FCVT.D.S rd,rs1[,rm]

Integer to Floating-Point Move & Conversion Instructions

rd	rs1	00000	01100	rm	00	1010011	FCVT.S.L rd,rs1[,rm]
rd	rs1	00000	01101	rm	00	1010011	FCVT.S.LU rd,rs1[,rm]
rd	rs1	00000	01110	rm	00	1010011	FCVT.S.W rd,rs1[,rm]
rd	rs1	00000	01111	rm	00	1010011	FCVT.S.WU rd,rs1[,rm]
rd	rs1	00000	01100	rm	01	1010011	FCVT.D.L rd,rs1[,rm]
rd	rs1	00000	01101	rm	01	1010011	FCVT.D.LU rd,rs1[,rm]
rd	rs1	00000	01110	rm	01	1010011	FCVT.D.W rd,rs1[,rm]
rd	rs1	00000	01111	rm	01	1010011	FCVT.D.WU rd,rs1[,rm]
rd	rs1	00000	11110	000	00	1010011	MXTF.S rd,rs1
rd	rs1	00000	11110	000	01	1010011	MXTF.D rd,rs1
rd	rs1	00000	11111	000	00	1010011	MTFSR rd,rs1

Floating-Point to Integer Move & Conversion Instructions

rd	rs1	00000	01000	rm	00	1010011	FCVT.L.S rd,rs1[,rm]
rd	rs1	00000	01001	rm	00	1010011	FCVT.LU.S rd,rs1[,rm]
rd	rs1	00000	01010	rm	00	1010011	FCVT.W.S rd,rs1[,rm]
rd	rs1	00000	01011	rm	00	1010011	FCVT.WU.S rd,rs1[,rm]
rd	rs1	00000	01000	rm	01	1010011	FCVT.L.D rd,rs1[,rm]
rd	rs1	00000	01001	rm	01	1010011	FCVT.LU.D rd,rs1[,rm]
rd	rs1	00000	01010	rm	01	1010011	FCVT.W.D rd,rs1[,rm]
rd	rs1	00000	01011	rm	01	1010011	FCVT.WU.D rd,rs1[,rm]
rd	00000	rs2	11100	000	00	1010011	MFTX.S rd,rs2
rd	00000	rs2	11100	000	01	1010011	MFTX.D rd,rs2
rd	00000	00000	11101	000	00	1010011	MFFSR rd

31 27	26 22	21 17	16 15 14 12	11 10 9	8 7	6 0	
jump target						opcode	J-type
rd	LUI-immediate					opcode	LUI-type
rd	rs1	imm[11:7]	imm[6:0]		funct3	opcode	I-type
imm[11:7]	rs1	rs2	imm[6:0]		funct3	opcode	B-type
rd	rs1	rs2	funct10			opcode	R-type
rd	rs1	rs2	rs3	funct5		opcode	R4-type

Floating-Point Compare Instructions

rd	rs1	rs2	10101	000	00	1010011	FEQ.S rd,rs1,rs2
rd	rs1	rs2	10110	000	00	1010011	FLT.S rd,rs1,rs2
rd	rs1	rs2	10111	000	00	1010011	FLE.S rd,rs1,rs2
rd	rs1	rs2	10101	000	01	1010011	FEQ.D rd,rs1,rs2
rd	rs1	rs2	10110	000	01	1010011	FLT.D rd,rs1,rs2
rd	rs1	rs2	10111	000	01	1010011	FLE.D rd,rs1,rs2

Miscellaneous Memory Instructions

rd	rs1	imm12	001	0101111	FENCE.I rd,rs1,imm12
rd	rs1	imm12	010	0101111	FENCE rd,rs1,imm12

System Instructions

00000	00000	00000	0000000	000	1110111	SYSCALL
00000	00000	00000	0000000	001	1110111	BREAK
rd	00000	00000	0000000	100	1110111	RDCYCLE rd
rd	00000	00000	0000001	100	1110111	RDTIME rd
rd	00000	00000	0000010	100	1110111	RDINSTRET rd

Table 6: Instruction listing for RISC-V

4 Floating-Point Extensions

This section describes the optional 128-bit binary floating-point instructions, and how we would intend to support the decimal floating-point arithmetic defined in the IEEE 754-2008 standard.

4.1 Quad-Precision Binary Floating-Point Extension

The 128-bit or quad-precision binary floating-point extensions are built upon the base floating-point instructions, and are only available as an extension to RV[C]64. The floating-point registers are now extended to hold either a single, double, or quad-precision floating-point value.

A new supported format is added to the format field of most instructions, as shown in Table 7.

fmt field	Mnemonic	Meaning
00	S	32-bit single-precision
01	D	64-bit double-precision
10	-	*reserved*
11	Q	128-bit quad-precisionn

Table 7: Format field encoding.

The following instructions support the quad-precision format: FADD/FSUB, FMUL/FDIV, FSQRT, F[N]MADD/F[N]MSUB, FCVT.*fmt*.W[U], FCVT.W[U].*fmt*, FCVT.*fmt*.L[U], FCVT.L[U].*fmt*, FS-GNJ[N], FSGNX, and FEQ/FLT/FLE.

New floating-point to floating-point conversion instructions FCVT.S.Q, FCVT.Q.S, FCVT.D.Q, FCVT.Q.D are added.

MFTX.Q and MXTF.Q instructions are not provided, so quad-precision bit patterns must be moved to the integer registers via memory.

New 128-bit variants of LOAD-FP and STORE-FP instructions are added, encoded with a new value for the width field.

4.2 Decimal Floating-Point Extension

The existing floating-point registers can be used to hold 64-bit and 128-bit decimal floating-point values, with the existing floating-point load and store instructions used to move values to and from memory.

Due to the large opcode space required by the fused multiply-add instructions, the decimal floating-point instruction extension requires a 48-bit instruction encoding.

5 Packed-SIMD Extensions

In this section, we outline how a packed-SIMD extension could be added to RISC-V (any of RV[C]64 or RV[C]32).

A packed-SIMD extension will have some fixed register width of 64 bits, 128 bits, or larger. These registers will be overlaid on the RISC-V floating-point registers. Each register can be treated as $N\times$64-bit, $2N\times$32-bit, $4N\times$16-bit, or $8N\times$8-bit packed variables, where N is 1 for the base 64-bit floating-point ISA but can be extended up to $N = 64$ (4096-bit registers). Packed-SIMD instructions operate on these packed values in FP registers.

The existing floating-point load and store instructions can be used to load and store various-sized words from memory. The base ISA supports 32-bit and 64-bit loads and stores, but the LOAD-FP and STORE-FP instruction encodings allow up to 8 different widths to be encoded (32, 64, 128, 256, 512, 1024, 2048, 4096). When used with packed-SIMD operations, it is desirable to support non-naturally aligned loads and stores in hardware.

Simple packed-SIMD extensions might fit in unused 32-bit instruction opcodes, but more extensive packed-SIMD extensions will likely require a 48-bit instruction encoding.

It is natural to use the floating-point registers for packed-SIMD values rather than the integer registers (PA-RISC and Alpha packed-SIMD extensions) as this frees the integer registers for control and address values and leads naturally to a decoupled integer/floating-point unit hardware design. The floating-point load and store instruction encodings also have space to handle wider packed-SIMD registers.

Reusing the floating-point registers for packed-SIMD values does make it more difficult to use a recoded internal format for floating-point values.

6 History and Acknowledgements

The RISC-V ISA and instruction set manual builds up several earlier projects. Several aspects of the supervisor-level machine and the overall format of the manual date back to the T0 (Torrent-0) vector microprocessor project at UC Berkeley and ICSI, begun in 1992. T0 was a vector processor based on the MIPS-II ISA, with Krste Asanović as main architect and RTL designer, and Brian Kingsbury and Bertrand Irrisou as principal VLSI implementers. David Johnson at ICSI was a major contributor to the T0 ISA design, particularly supervisor mode, and to the manual text. John Hauser also provided considerable feedback on the T0 ISA design.

The Scale (Software-Controlled Architecture for Low Energy) project at MIT, begun in 2000, built upon the T0 project infrastructure, refined the supervisor-level interface, and moved away from the MIPS scalar ISA by dropping the branch delay slot. Ronny Krashinsky and Christopher Batten were the principal architects of the Scale Vector-Thread processor at MIT, while Mark Hampton ported the GCC-based compiler infrastructure and tools for Scale.

A lightly edited version of the T0 MIPS scalar processor specification (MIPS-6371) was used in teaching a new version of the MIT 6.371 Introduction to VLSI Systems class in the Fall 2002 semester, with Chris Terman and Krste Asanović as lecturers. Chris Terman contributed most of the lab material for the class (there was no TA!). The 6.371 class evolved into the trial 6.884 Complex Digital Design class at MIT, taught by Arvind and Krste Asanović in Spring 2005, which became a regular Spring class 6.375. A reduced version of the Scale MIPS-based scalar ISA, named SMIPS, was used in 6.884/6.375. Christopher Batten was the TA for the early offerings of these classes and developed a considerable amount of documentation and lab material based around the SMIPS ISA. This same SMIPS lab material was adapted and enhanced by TA Yunsup Lee for the UC Berkeley Fall 2009 CS250 VLSI Systems Design class taught by John Wawrzynek, Krste Asanović, and John Lazzaro.

The Maven (Malleable Array of Vector-thread ENgines) project was a second-generation vector-thread architecture, whose design was led by Christopher Batten while an Exchange Scholar at UC Berkeley starting in summer 2007. Hidetaka Aoki, a visiting industrial fellow from Hitachi gave considerable feedback on the early Maven ISA and microarchitecture design. The Maven infrastructure was based on the Scale infrastructure but the Maven ISA moved further away from the MIPS ISA variant defined in Scale, with a unified floating-point and integer register file. Maven was designed to support experimentation with alternative data-parallel accelerators. Yunsup Lee was the main implementor of the various Maven vector units, while Rimas Avizienis was the main implementor of the various Maven scalar units. Christopher Batten and Yunsup Lee ported GCC to work with the new Maven ISA. Christopher Celio provided the initial definition of a traditional vector instruction set ("Flood") variant of Maven.

Based on experience with all these previous projects, the RISC-V ISA definition was begun in Summer 2010. An initial version of the RISC-V 32-bit instruction subset was used in the UC Berkeley Fall 2010 CS250 VLSI Systems Design class. RISC-V is a clean break from the earlier MIPS-inspired designs. John Hauser contributed to the floating-point ISA definition.