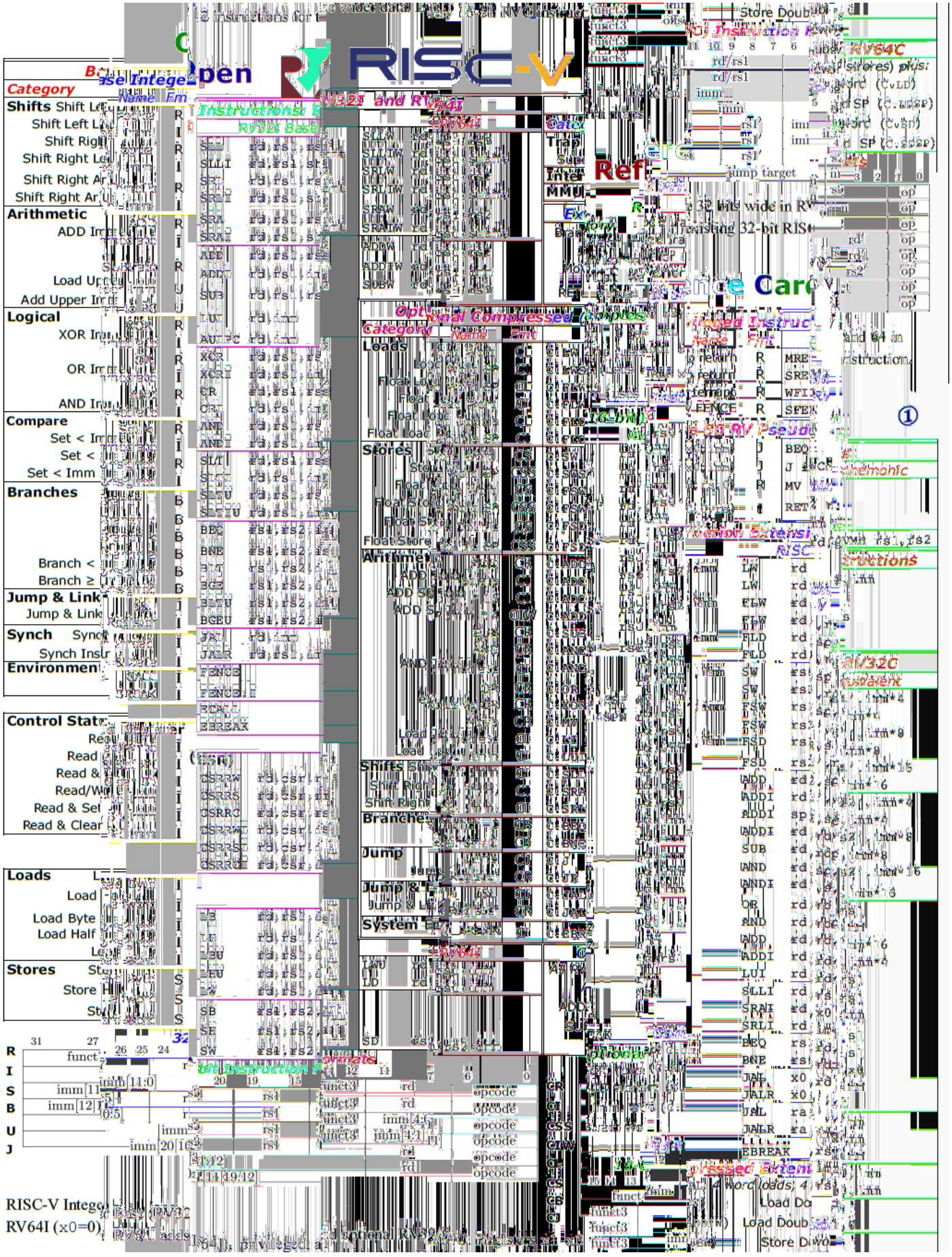
2018

RISC-V 手册

一 开 令 南

DAVID PATTERSON, ANDREW WATERMAN

勾凌 、 、刘志刚
包云岗


RISC-V
Reference Card
RV32I and RV64I
Instruction Formats
Optional Compressed
Pseudo
RV32C
RV64C
RV64I (x0=0)

Open RISC-V Reference Card

Name	Fmt	RV32V/R64V
	R	VLDX rd,rs1,rs2
	S	VST rd,rs1,imm
	R	VSTS rd,rs1,rs2
	R	VSTX rd,rs1,rs2
	R	AMOSWAP rd,rs1,rs2
	R	AMOADD rd,rs1,rs2
	R	AMOXOR rd,rs1,rs2
	R	AMOAND rd,rs1,rs2
	R	AMOOR rd,rs1,rs2
	R	AMOMIN rd,rs1,rs2
	R	AMOMAX rd,rs1,rs2
	R	VPEQ rd,rs1,rs2
	R	VPNE rd,rs1,rs2
	R	VPLT rd,rs1,rs2
	R	VPGE rd,rs1,rs2
	R	VPAND rd,rs1,rs2
	R	VPANDN rd,rs1,rs2
	R	VPOR rd,rs1,rs2
	R	VPXOR rd,rs1,rs2
	R	VPNOT rd,rs1
	R	VPSWAP rd,rs1
	R	VMOV rd,rs1
	R	VCVT rd,rs1
	R	VADD rd,rs1,rs2
	R	VSUB rd,rs1,rs2
	R	VMUL rd,rs1,rs2
	R	VDIV rd,rs1,rs2
	R	VSQRT rd,rs1,rs2
	R	VFMADD rd,rs1,rs2,rs3
	R	VFMSUB rd,rs1,rs2,rs3
	R	VFNMSUB rd,rs1,rs2,rs3
	R	VFNMADD rd,rs1,rs2,rs3
	R	VSGNJ rd,rs1,rs2
	R	VSGNJN rd,rs1,rs2
	R	VSGNJX rd,rs1,rs2
	R	VMIN rd,rs1,rs2
	R	VMAX rd,rs1,rs2
	R	VXOR rd,rs1,rs2
	R	VOR rd,rs1,rs2
	R	VAND rd,rs1,rs2
	R	VCLASS rd,rs1
	R	VSETDCFG rd,rs1
	R	VEXTRACT rd,rs1,rs2
	R	VMERGE rd,rs1,rs2
	R	VSELECT rd,rs1,rs2

32- and 64-bit data (RV32F, ... (F|D) adds registers f0-f31, ... registers v0-v31, vector ... F 6, RVD 6, and RVV 0.

②

Vector Extension: RVV

Name	Fmt	RV32V/R64V
SET Vector Len.	R	SETVL rd,rs1
MULtiply High	R	VMULH rd,rs1,rs2
REMainder	R	VREM rd,rs1,rs2
Shift Left Log.	R	VSLL rd,rs1,rs2
Shift Right Log.	R	VSRL rd,rs1,rs2
Shift R. Arith.	R	VSRA rd,rs1,rs2
LoaD	I	VLD rd,rs1,imm

、

、、

RISC-V

RISC-V

C. Gordan Bell Digital PDP-11
VAX-11

RISC-V
RISC-V
RISC-V
Ivan Sutherland

RISC-V
Michael B. Taylor

RISC-V
RISC-V
RISC-V

John Mashey MIPS

RISC-V ISA

ISA ISA
RISC-V
Megan Wachs SiFive

David Patterson

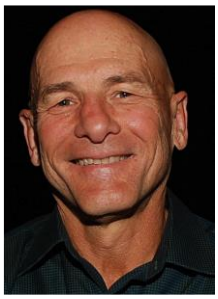
David

Lucy

Andrew Waterman

Elizabeth

John



David Patterson 40 2016 Google distinguished engineer Google RISC-V Computer Science Division Computing Research Association ACM Association for Computing Machinery 20 80 RISC Reduced Instruction Set Computer RISC RISC Five” Andrew Waterman RISC-V RISC RAID Redundant Arrays of Inexpensive Disks NOW Networks of Workstations 7 35

ACM IEEE AAAS Karstrom ACM Mulligan IEEE IEEE Text and Academic Authors Association Texty

50

Andrew Waterman SiFive SiFive RISC-V RISC-V

RISC-V ISA RISC-V Andrew RISC-V Rocket Chisel Linux GNU C C RISC-V RISC-V RVC

RISC-V 2011

(ISA)

RISC-V

See MIPS Run 500

RISC-V

100

2.1 4.1 5.1 6.1 7.1 8.1 9.1 9.2 9.3 9.4

- 2.3 3.3 3.4 4.2 5.2 5.3 6.2 7.6 7.5 7.7 9.5 10.1
- A RISC-V [1] RISC-V 50
-

www.riscvbook.com

2017 5 8 11 RISC-V Patterson

Patterson 1 2 3 4 5 6 7 8 9 11 Waterman 10 A Waterman Armando Fox LaTeX 2017 800 2017 2017 11 28 30 RISC RISC-V

Armando Fox LaTeX

Krste Asanovi´c, Nikhil Athreya, C. Gordon Bell, Stuart Hoad, David Kanter, John Mashey, Ivan Sutherland, Ted Speers, Michael Taylor, Megan Wachs,….

David Patterson Andrew Waterman
2017 9 1

[1] RV32V A RV32V

David Patterson Andrew Waterman
The RISC-V Reader
David Patterson Krste
Asanovic
RISC-V
2015 2
“ ”
RISC-V RISC-V Patterson
RISC-V
RISC-V ——RISC-V Linux
RISC-V
Instruction Sets Want to be Free ——
RISC-V RISC-V

Linux Linux Python LLVM
GCC MySQL Apache Hadoop
150
BAT
3-5
RISC-V Linux RISC-V
RISC-V
IP SoC
RISC-V

RISC-V

2018 11 3

✧
✧
✧
✧ RISC-V

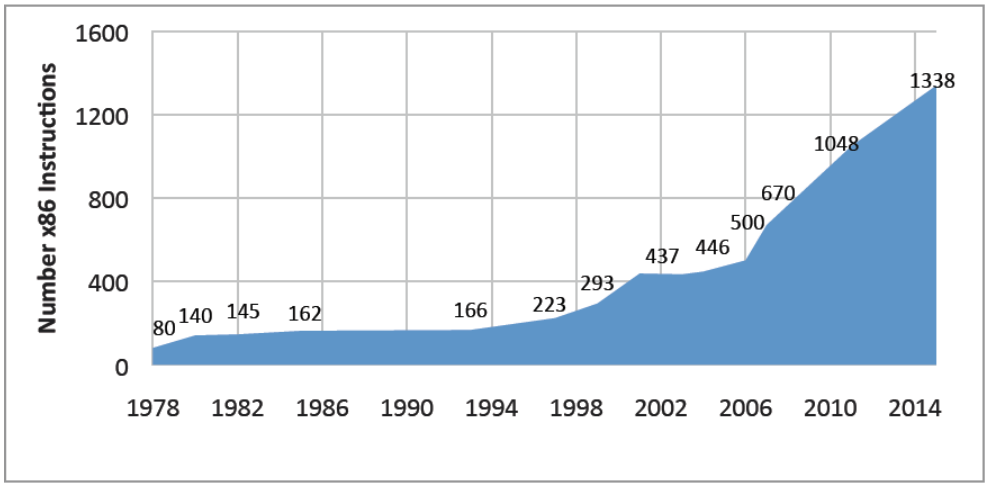
RISC-V

(Leonardo da Vinci)

RISC-V RISC five ISA

-
-
- FPGA
  ASIC
- ;
  ;
-
-
  AMD Am29000 Digital Alpha Digital VAX
  Hewlett Packard PA-RISC Intel i860 Intel i960 Motorola 88000 Zilog
  Z8000

RISC-V




```
The AL register is the default source and destination.
If the low 4-bits of AL register are > 9,
   or the auxiliary carry flag AF = 1,
 Then
    Add 6 to low 4-bits of AL and discard overflow
    Increment the high byte of AL
    Carry flag CF = 1
    Auxiliary carry flag AF = 1
 Else
    CF = AF = 0
Upper 4-bits of AL = 0
```

aaa

aas aam aad

Zilog

8086

8086 32

64 80186,80286 i386 i486 Pentium

Stephen P. Morse, 8086 [Morse 2017]

ISA ISA

ISA 1.2 ISA
80x86 1978

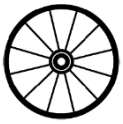
x86-32 32 x86
1.3 x86 ASCII Adjust after
Addition aaa

RISC-V
ISA *RV32I* ISA
RV32I
RISC-V
RISC-V

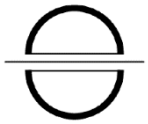
RV32IMFD RV32M RV32F
RV32D RV32I
RISC-V

RISC-V RISC-V

ISA



RISC-V ISA ISA



RISC-V

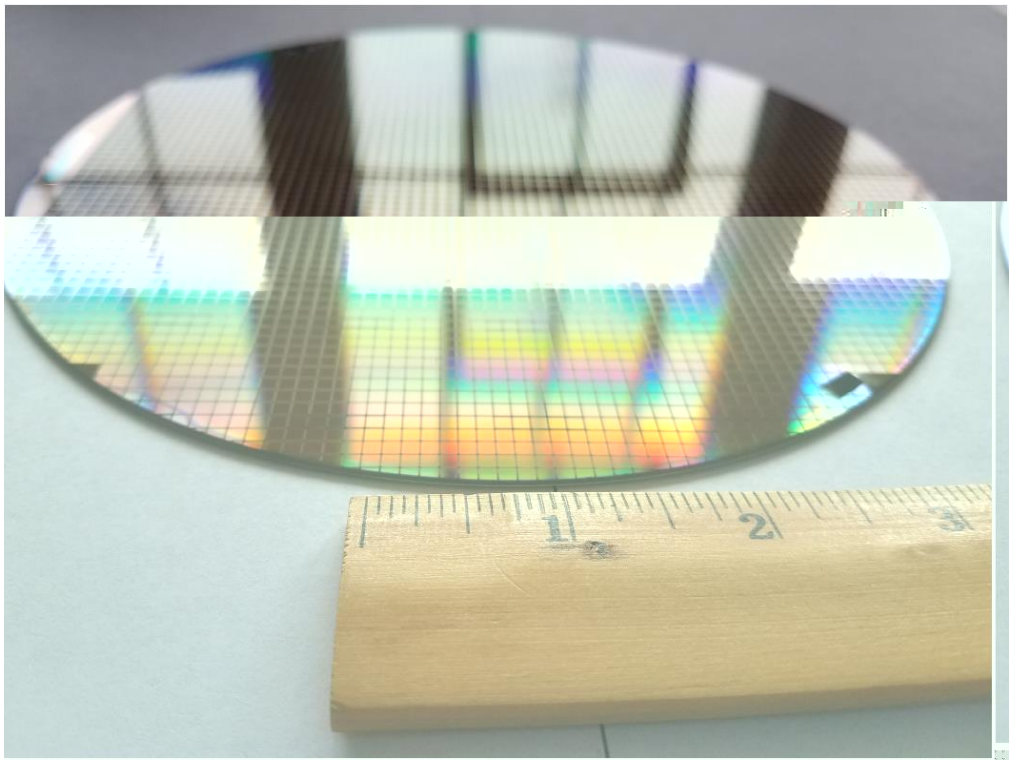
-
-
-
-
-
-
- / / ABC

ISA

RISC-V

1.4 RISC-V

$$cost \approx f(die\ area^2)$$



ISA ISA

RISC-V ISA ARM-32 ISA

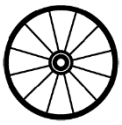
16KiB RISC-V Rocket TSMC40GPLUS ARM-32 Cortex A5 RISC-V 0.27mm^2 ARM-32 0.53mm^2 ARM-32 Cortex A5 RISC-V Rocket 4 2^2 10% 1.2 1.1^2

ISA ISA

ISA

ARM-32 ISA

ldmiaeq SP!, {R4-R7, PC}



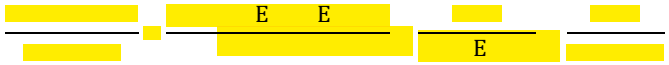
LoaD Multiple, Increment-Address, on EQual 5 6

EQ PC

x86-32 enter

x86-32

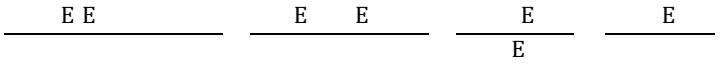
```
push ebp        #
mov  ebp, esp   #
```



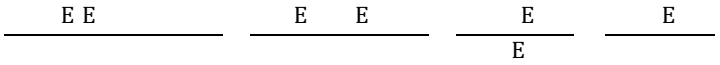
ISA ISA

CPI

CoreMark [Gal-On, Levy 2012] 100000 ARM-32 Cortex-A9



RISC-V BOOM



ARM RISC-V

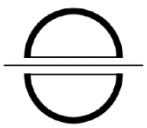
ISA

RISC-V 10%

30% ISA

20 60

ISA



MIPS-32 ISA

MIPS-32

MIPS-32 MIPS-32

ISA 1.2 MIPS-

32 29 2.10

ARM-32 ISA

Load Multiple

Load Multiple

Moore’s law

ISA

20 70 80

ARM-32

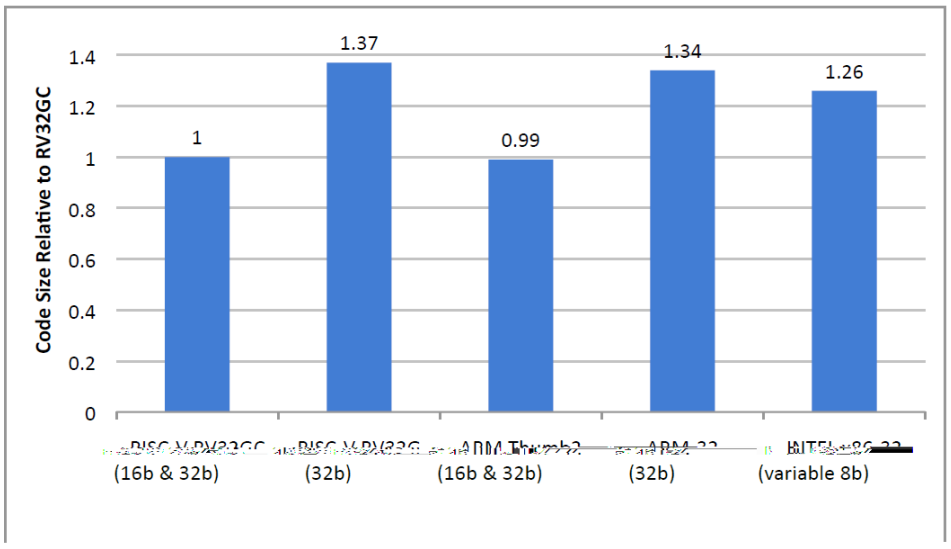
32 ISA 16

16 ISA Thumb 16 32

ISA Thumb-2

1

16 32 0



(

) ARM Thumb Thumb-2 ISA

DRAM

SRAM ISA

x86-32 ISA 1 15 x86
ISA ARM-32 RISC-V 32
16 32
ISA Thumb-2 RV32C RISC-V 1.5
32 ARM-32 RISC-V x86-32 6% 9%
x86-32 16 32 Thumb-2 RV32C 26%
ISA RV32C Thumb-2
20 70 x86 ISA
1.2 x86-32
x86

```
lock
add dword ptr ds
[esi+ecx*4+0x1234
5678] 0xefcdab89
```

```
inc eax
```

ARM-32 16 x86-32 8 ISA RISC-V 32

RISC-V
ARM-32 x86-32
ARM-32 RISC-V x86-32

ISA PIC 3.5
PC PIC
ISA PC x86-32 MIPS-32 PC

ARM-32 MIPS-32 x86-32

ISA 32 ARM ISA
1986 2005 ARMv7 ARM-32
ARMv7 ISA MIPS 32 MIPS I “MIPS32”
MIPS-32 ISA 16 1978
8086 80386 ISA 1985 32 x86-32 IA-
32 x86 ISA 32 ISA

RISC-V
RISC-V [Patterson Hennessy 2017]
RV32I RISC-V RISC-V
RISC-V

RISC-V RISC-V

ISA

RISC-V RIV32I

RV32G G

- RV32M
- RV32F RV32D
- RV32A

3 4 RISC-V“ ” RISC-V RV32G RV64G

RV32 / 64V

RV32C RISC-V

16 32 RV32G

RV32C 16 RV32C

32 RV32G 400 RISC-V

RV32V *SIMD* ARM-

32 MIPS-32 x86-32 ISA

1.2 x86-32 SIMD RV32V

ISA

RV32V SIMD ISA RISC-V

RV64G RISC-V 64

RISC-V RV32G

32 64

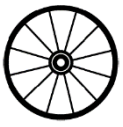
RISC-V

RISC-V

A

RISC-V ISA 50

RISC-V



[ ]

[ ] [ ]

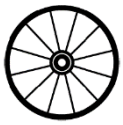
von Neumann 1947

RISC-V ISA ISA

RISC-V

70 ISA

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



1.6 RISC-V
ARM-32 x86-32 8
5 ARM-32 x86-32
ARM-32 x86-32 RISC-V
ARM-32 —,x86-32 — — RISC-V ISA

ISA 2011
RISC-V

ISA
RV32I

RISC-V ISA
ISA
RISC-V

ARM Ltd. ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition, 2014. URL http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/

A. Baumann. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 132–137. ACM, 2017.

C. Celio, D. Patterson, and K. Asanovic. The Berkeley Out-of-Order Machine (BOOM): an industry-competitive, synthesizable, parameterized RISC-V processor. *Tech. Rep. UCB/EECS-2015–167, EECS Department, University of California, Berkeley*, 2015.

S. Gal-On and M. Levy. Exploring CoreMark-a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012.

Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference*. September 2016.

S. P. Morse. The Intel 8086 chip and the future of microprocessor design. *IEEE Computer*, 50(4): 8–9, 2017.

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

S. Rodgers and R. Uhlig. X86: Approaching 40 and still going strong, 2017.

J. L. von Neumann, A.W. Burks, and H. H. Goldstine. Preliminary discussion of the logical design of an electronic computing instrument. *Report to the U.S. Army Ordnance Department*, 1947.

A. Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10*. May 2017a. URL https://riscv.org/specifications/privileged-isa/.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017b. URL https://riscv.org/specifications/.

RV32I RISC-V

Frances Elizabeth “Fran” Allen

——Frances Elizabeth “Fran” Allen, 1981

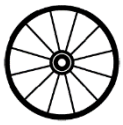
2.1 RV32I

RV32I {}

– –

RV32I slt slti sltu sltiu:

set less than { – immediate } { – unsigned }



2.2 - R

load I store S

B U J 2.3

2.2 2.1 RV32I

RISC-V ISA

32 ARM-

32 x86-32

RISC-V

x86-32

ISA

RISC-V

ARM-32 MIPS-32

x 0xfffffff0 RISC-V

andi

MIPS-32

addiu

and MIPS

ARM-32

bic rx

immediate

.

move

RV32I

Integer Computation

- add { _ / immediate }
- subtract
- { and / or / exclusive or } { _ / immediate }
- { shift left logical / shift right arithmetic / shift right logical } { _ / immediate }
- load upper immediate
- add upper immediate to pc
- set less than { _ / immediate } { _ / unsigned }

Control transfer

- branch { equal / not equal }
- branch { greater than or equal / less than } { _ / unsigned }
- jump and link { _ / register }

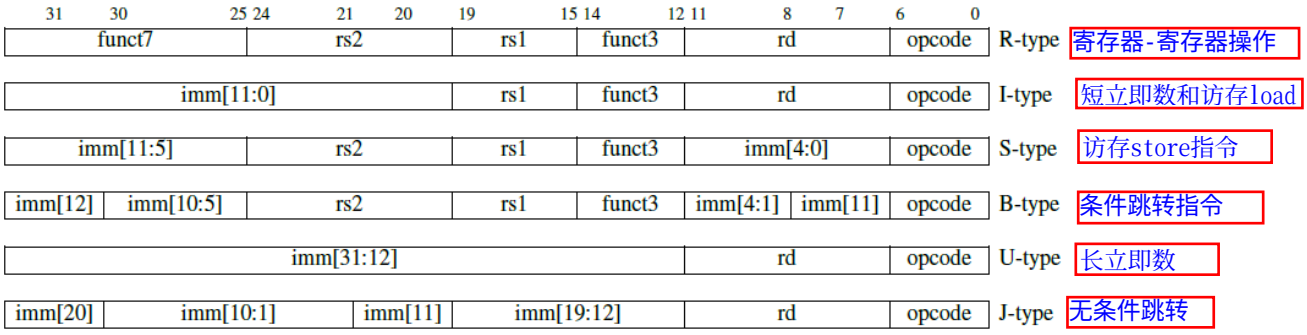
Loads and Stores

- { load / store } { byte / halfword / word }
- load { byte / halfword } unsigned

Miscellaneous instructions

- fence loads & stores
- fence.instruction & data
- environment { break / call }
- control status register { read & clear bit / read & set bit / read & write } { _ / immediate }

and or xor andi ori xori



31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred	succ	00000	000	00000	0001111	I fence
0000 0000	0000	00000	001	00000	0001111	I fence.i
000000000000		00000	00	00000	1110011	I ecall
000000000000		00000	000	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I cssrrsi
csr		zimm	111	rd	1110011	I csrrci

B J
B S 1 J
U 12 ,RISC-V

0 RV32I ,
1

ISA RV32I 32
RV32I
, B
J 1 2
RISC-V

RISC-V
RV32M RV32F
RISC-V

RISC-V
RISC-V

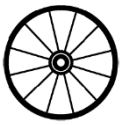
ARM-32 12
12 8
12 4 2 12
ARM-32

4 RV32I RISC-V ABI
ABI
RV32I 31 0 x0
ARM-32 16 x86-32 8
0 RISC-V ISA
3 36 3 ARM-32 x86-32
RV32I

PC ARM-32 16
PC ISA
10%-20% ARM-32
PC

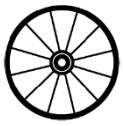
90

A RISC-V
ISA
RISC-V RISC-V ISA
5 add, sub and, or, xor 2.1
5 sll, srl, sra ISA 32 32
10 RV32I ARM-32



sub set less than
RV32I
1
0 slt sltu
slti sltiu
RV32I

slt
2.1
lui 20 20 32
2 32 RV32I 32 PC
auipc PC
auipc 20 jalr 12
32 PC auipc 12
32 PC



RISC-V
ARM-32
RV32I

RV32I RV32M 4
ARM-32 x86-32 RISC-V
MIPS-32

RV32I
RV32I 2.6

31 0

x0 / zero	Hardwired zero
x1 / ra	Return address
x2 / sp	Stack pointer
x3 / gp	Global pointer
x4 / tp	Thread pointer
x5 / t0	Temporary
x6 / t1	Temporary
x7 / t2	Temporary
x8 / s0 / fp	Saved register, frame pointer
x9 / s1	Saved register
x10 / a0	Function argument, return value
x11 / a1	Function argument, return value
x12 / a2	Function argument
x13 / a3	Function argument
x14 / a4	Function argument
x15 / a5	Function argument
x16 / a6	Function argument
x17 / a7	Function argument
x18 / s2	Saved register
x19 / s3	Saved register
x20 / s4	Saved register
x21 / s5	Saved register
x22 / s6	Saved register
x23 / s7	Saved register
x24 / s8	Saved register
x25 / s9	Saved register
x26 / s10	Saved register
x27 / s11	Saved register
x28 / t3	Temporary
x29 / t4	Temporary
x30 / t5	Temporary
x31 / t6	Temporary

32

31 0

pc

32

“ ”

RISC-V rotate RV32B
11

xor

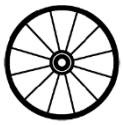
x1 x2

E

```
xor x1,x1,x2 # x1’ == x1^x2, x2’ == x2
xor x2,x1,x2 # x1’ == x1^x2, x2’ == x1’^x2 == x1^x2^x2 == x1
xor x1,x1,x2 # x1” == x1’^x2’ == x1^x2^x1 == x1^x1^x2 == x2, x2’ == x1
```

RISC-V

32 lw sw 2.1 RV32I
lb lbu lh lhu sb sh
32
32
32
12



x86-32 [Irvine 2014]
RV32I ARM-32 x86-32 ARM-32
RV32I RISC-
V x86 0 x86
x86-32 RISC-V 31
2.4 push pop
ISA MIPS-32 RISC-V *delayed load*
MIPS-32 load
load load
RISC-V

ARM-32 MIPS-32
RISC-V
ISA

MIPS-32 Load Word Left Load WordRight
lwl lwr

RISC-V x86-32 Apple iOS Android Windows for ARM

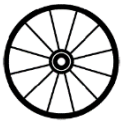
RV32I

beq bne bge blt bltu RV32I bgeu bltu

$x < y$ $y > x$ $x \geq y$ $y \leq x$

RISC-V 12 2 PC PC 3

RISC-V MIPS-32 Oracle SPARC ARM-32 x86-32



x86-32 loop loope loopz loopne loopnz

RV32I sltu

```
add a0,a2,a4 #      32    : a0 = a2 + a4
sltu a2,a0,a2 #    (a2+a4) < a2      a2' = 1,      a2' = 0
add a5,a3,a5 #      32    : a5 = a3 + a5
add a1,a2,a5 #        32
```

PC

PC auipc U 0 x86-32 PC PC ;

PC PC PC

1 store 2 load 2

RfC

2.1 jal PC + 4
ra 2.4
x0 ra x0
jal 20 2 PC

jalr
ra x0
Switch case jalr x0

RV32I x86-32
Intel Itanium Oracle SPARC Cadence Tensilica

2.1 (csrrc csrrs csrrw csrrci csrrsi csrrwi)
64 , 32
,
RISC-V ecall
ebreak
fence I / O
fence.i
fence.i RISC-V

10 RISC-V

RISC-V I / O x86-32 in ins insb insw out out outsb I/O RISC-V x86-32 rep movs coms scas lods 16

RISC-V ARM-32 MIPS-32 x86-32 2.5 C 2.6

2.7 1 ISA

RV32I RISC-V

ISA RV32I RISC-I [Patterson 2017]

RISC-V [Chen & Patterson 2016]

- 32
- 32
- 31 32 0
- /
- /
- +
- PC
- 32

RISC-V ISA

Santayana -

RISC-I RISC-V

Lindy [Lin 2017]

RISC

IBM 360 44 Digital Equipment microVAX RV32I

RV32I RV32G

RV32I RISC-V ISA

Tensilica Xtensa 80

Tensilica

Xtensa RV32I ISA 64

Lindy effect, 2017. URL https://en.wikipedia.org/wiki/Lindy_effect

T. Chen and D. A. Patterson. RISC genealogy. Technical Report UCB/EECS-2016-6, EECS Department, University of California, Berkeley, Jan 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-6.html.

W. Hohl and C. Hinds. *ARM Assembly Language: Fundamentals and Techniques*. CRC Press, 2016.

K. R. Irvine. *Assembly language for x86 processors*. Prentice Hall, 2014.

D. Patterson. How closes is RISC-V to RISC-I?, 2017.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/

	ARM-32 (1986)	MIPS-32 (1986)	x86-32 (1978)	RV32I (2011)
			8 16	8 16 (RV32M)
	push/pop /		(enter/leave) (push/pop)	x0 0
	PC			load PC
	PC	Load HI LO	(AX,CX,DX, DI,SI)	Load
	32bit (Thumb-2 ISA)	32bit (microMIPS ISA)		32 +16 RV32C
/ /	15		15	31 PC

```
# RV32I (19 instructions, 76 bytes, or 52 bytes with RVC)
# a1 is n, a3 points to a[0], a4 is i, a5 is j, a6 is x
   0: 00450693  addi  a3,a0,4    # a3 is pointer to a[i]
   4: 00100713  addi  a4,x0,1    # i = 1
Outer Loop:
   8: 00b76463  bltu  a4,a1,10   # if i < n, jump to Continue Outer loop
Exit Outer Loop:
   c: 00008067  jalr  x0,x1,0    # return from function
Continue Outer Loop:
  10: 0006a803  lw    a6,0(a3)  # x = a[i]
  14: 00068613  addi  a2,a3,0   # a2 is pointer to a[j]
  18: 00070793  addi  a5,a4,0   # j = i
Inner Loop:
  1c: ffc62883  lw    a7,-4(a2) # a7 = a[j-1]
nner Loop
  20: 0118ba63  bge   a6,a7,34  # if a[j-1] <= a[i], jump to Exit I
  24: 01162023  sw    a7,0(a2)  # a[j] = a[j-1]
  28: fff78793  addi  a5,a5,-1  # j--
  2c: ffc60613  addi  a2,a2,-4  # decrement a2 to point to a[j]
  30: fe0796e3  bne   a5,x0,1c  # if j != 0, jump to Inner Loop
Exit Inner Loop:
  34: 00279793  slli  a5,a5,0x2 # multiply a5 by 4
  38: 00f507b3  add   a5,a0,a5  # a5 is now byte address oi a[j]
  3c: 0107a023  sw    a6,0(a5)  # a[j] = x
  40: 00170713  addi  a4,a4,1   # i++
  44: 00468693  addi  a3,a3,4   # increment a3 to point to a[i]
  48: fc1ff06f  jal   x0,8      # jump to Outer Loop
```

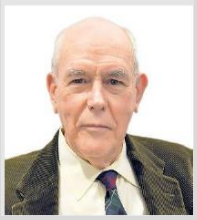
```
# ARM-32 (19 instructions, 76 bytes; or 18 insns/46 bytes with Thumb-2)
# r0 points to a[0], r1 is n, r2 is j, r3 is i, r4 is x
  0: e3a03001 mov  r3, #1              # i = 1
  4: e1530001 cmp  r3, r1              # i vs. n (unnecessary?)
  8: e1a0c000 mov  ip, r0              # ip = a[0]
  c: 212fff1e bxcs lr                  # don't let return address change ISAs
 10: e92d4030 push {r4, r5, lr}        # save r4, r5, return address
Outer Loop:
 14: e5bc4004 ldr  r4, [ip, #4]!       # x = a[i] ; increment ip
 18: e1a02003 mov  r2, r3              # j = i
 1c: e1a0e00c mov  lr, ip              # lr = a[0] (using lr as scratch reg)
Inner Loop:
 20: e51e5004 ldr  r5, [lr, #-4]       # r5 = a[j-1]
 24: e1550004 cmp  r5, r4              # compare a[j-1] vs. x
 28: da000002 ble  38                  # if a[j-1]<=a[i], jump to Exit Inner Loop
 2c: e2522001 subs r2, r2, #1          # j--
 30: e40e5004 str  r5, [lr], #-4       # a[j] = a[j-1]
 34: 1afffff9 bne  20                  # if j != 0, jump to Inner Loop
Exit Inner Loop:
 38: e2833001 add  r3, r3, #1          # i++
 3c: e1530001 cmp  r3, r1              # i vs. n
 40: e7804102 str  r4, [r0, r2, lsl #2] # a[j] = x
 44: 3afffff2 bcc  14                  # if i < n, jump to Outer Loop
 48: e8bd8030 pop  {r4, r5, pc}        # restore r4, r5, and return address
```

```
# x86-32 (20 instructions, 45 bytes)
# eax is j, ecx is x, edx is i
# pointer to a[0] is in memory at address esp+0xc, n is in memory at esp+0x10
  0: 56             push esi                 # save esi on stack (esi needed below)
  1: 53             push ebx                 # save ebx on stack (ebx needed below)
  2: ba 01 00 00 00 mov  edx,0x1             # i = 1
  7: 8b 4c 24 0c    mov  ecx,[esp+0xc]       # ecx is pointer to a[0]
Outer Loop:
  b: 3b 54 24 10    cmp  edx,[esp+0x10]      # compare i vs. n
  f: 73 19          jae  2a <Exit Loop>      # if i >= n, jump to Exit Outer Loop
 11: 8b 1c 91       mov  ebx,[ecx+edx*4]     # x = a[i]
 14: 89 d0          mov  eax,edx             # j = i
Inner Loop:
 16: 8b 74 81 fc    mov  esi,[ecx+eax*4-0x4] # esi = a[j-1]
 1a: 39 de          cmp  esi,ebx             # compare a[j-1] vs. x
 1c: 7e 06          jle  24 <Exit Loop>      # if a[j-1]<=a[i],jump Exit Inner Loop
 1e: 89 34 81       mov  [ecx+eax*4],esi     # a[j] = a[j-1]
 21: 48             dec  eax                 # j--
 22: 75 f2          jne  16 <Inner Loop>     # if j != 0, jump to Inner Loop
Exit Inner Loop:
 24: 89 1c 81       mov  [ecx+eax*4],ebx     # a[j] = x
 27: 42             inc  edx                 # i++
 28: eb e1          jmp  b <Outer Loop>      # jump to Outer Loop
Exit Outer Loop:
 2a: 5b             pop  ebx                 # restore old value of ebx from stack
 2b: 5e             pop  esi                 # restore old value of esi from stack
 2c: c3             ret                      # return from function
```

RISC-V

Ivan Sutherland
1938-
1962
Sketchpad

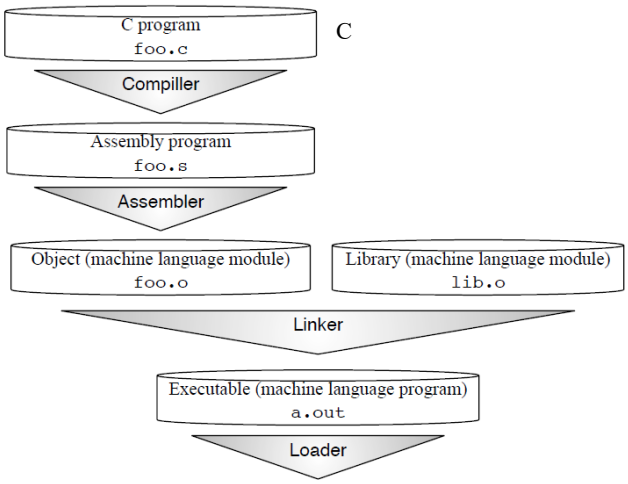
Sketchpad



Ivan Sutherland

3.1 C

RISC-V



C

.C, .ASM, .OBJ, .LIB .EXE

6 [Patterson and Hennessy 2017]

1.
2. RV32I jal
3.
4.
5.

6. ret

RISC-V

spilling

3.2 RISC-V

ABI

Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero 0	—
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	No /
x6–7	t1–2	Temporaries	No
x8	s0/fp	Saved register/frame pointer	Yes /
x9	s1	Saved register	Yes
x10–11	a0–1	Function arguments/return values	No /
x12–17	a2–7	Function arguments	No
x18–27	s2–11	Saved registers	Yes
x28–31	t3–6	Temporaries	No
f0–7	ft0–7	FP temporaries	No
f8–9	fs0–1	FP saved registers	Yes
f10–11	fa0–1	FP arguments/return values /	No
f12–17	fa2–7	FP arguments	No
f18–27	fs2–11	FP saved registers	Yes
f28–31	ft8–11	FP temporaries	No

ABI RV32I

```
entry_label:
  addi sp,sp,-framesize    # Allocate space for stack frame                  sp
                           # by adjusting stack pointer (sp register)
  sw   ra,framesize-4(sp)  # Save return address (ra register)               ra
  # save other registers to stack if needed
  ... # body of the function
```

Pseudoinstruction	Base Instruction(s)	Meaning
`nop`	`addi x0, x0, 0`	No operation
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`j offset`	`jal x0, offset`	Jump
`jr rs`	`jalr x0, rs, 0`	Jump register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Read instructions-retired counter
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Read cycle counter
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Read real-time clock
`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	Set bits in CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Clear bits in CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Set bits in CSR, immediate
`csrci csr, imm`	`csrrci x0, csr, imm`	Clear bits in CSR, immediate
`frcsr rd`	`csrrs rd, fcsr, x0`	Read FP control/status register
`fscsr rs`	`csrrw x0, fcsr, rs`	Write FP control/status register
`frrm rd`	`csrrs rd, frm, x0`	Read FP rounding mode
`fsrm rs`	`csrrw x0, frm, rs`	Write FP rounding mode
`frflags rd`	`csrrs rd, fflags, x0`	Read FP exception flags
`fsflags rs`	`csrrw x0, fflags, rs`	Write FP exception flags

Pseudoinstruction	Base Instruction(s)	Meaning
`lla rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load local address
`la rd, symbol`	*PIC*: `auipc rd, GOT[symbol][31:12]` `l{w|d} rd, rd, GOT[symbol][11:0]` *Non-PIC*: Same as `lla rd, symbol`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`	Floating-point load global
ore global	`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point st
	`li rd, immediate`	*Myriad sequences*	Load immediate
	`mv rd, rs`	`addi rd, rs, 0`	Copy register
nt	`not rd, rs`	`xori rd, rs, -1`	One's compleme
	`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend wor
	`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero 0
ision register	`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-prec
absolute value	`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision
negate	`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision
sion register	`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-pre
absolute value	`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision
negate	`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision
	`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
	`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if <
qz rd	`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if > , uns
qz rd	`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if < , uns
	`jal offset`	`jal x1, offset`	Jump and link
gister	`jalr rs`	`jalr x1, rs, 0`	Jump and link reg
routine	`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away sub
mory and I/O	`fence`	`fence iorw, iorw`	Fence on all mem I/O
status register	`fscsr rd, rs`	`csrrw rd, fcsr, rs`	Swap FP control/ FP /
g mode	`fsrm rd, rs`	`csrrw rd, frm, rs`	Swap FP roundin FP
on flags	`fsflags rd, rs`	`csrrw rd, fflags, rs`	Swap FP exceptic FP

```
#include <stdio.h>
int main()
{
   printf("Hello, %s\n", "world");
   return 0;
}
```

hello.c 。

```
  .text                     # Directive: enter text section
  .align 2                  # Directive: align code to 2^2 bytes          2^2
  .globl main               # Directive: declare global symbol main                main
main:                       # label for start of main  main
  addi sp,sp,-16            # allocate stack frame
  sw   ra,12(sp)            # save return address
  lui  a0,%hi(string1)      # compute address of          string1
  addi a0,a0,%lo(string1)   #   string1
  lui  a1,%hi(string2)      # compute address of          string2
  addi a1,a1,%lo(string2)   #   string2
  call printf               # call function printf        printf
  lw   ra,12(sp)            # restore return address
  addi sp,sp,16             # deallocate stack frame
  li   a0,0                 # load return value 0
  ret                       # return
  .section .rodata          # Directive: enter read-only data section
  .balign 4                 # Directive: align data section to 4 bytes       4
string1:                    # label for first string
  .string "Hello, %s!\n"    # Directive: null-terminated string
string2:                    # label for second string
  .string "world"           # Directive: null-terminated string
```

hello.s

```
00000000 <main>:
 0: ff010113  addi  sp,sp,-16
 4: 00112623  sw    ra,12(sp)
 8: 00000537  lui   a0,0x0
 c: 00050513  mv    a0,a0
10: 000005b7  lui   a1,0x0
14: 00058593  mv    a1,a1
18: 00000097  auipc ra,0x0
1c: 000080e7  jalr  ra
20: 00c12083  lw    ra,12(sp)
24: 01010113  addi  sp,sp,16
28: 00000513  li    a0,0
2c: 00008067  ret
```

hello.o

jump and

link link editor 3.1

Unix .o a.out MS-

DOS .OBJ .LIB .EXE

3.10 RISC-V

PIC PIC

2 RV32I PC-relative branch
PIC

3.6 string1 string2
main printf 32 32
RV32I 3.6 lui
addi auipc jalr 3.8 3.7
a.out

```
000101b0 <main>:
   101b0: ff010113 addi sp,sp,-16
   101b4: 00112623 sw   ra,12(sp)
   101b8: 00021537 lui  a0,0x21
   101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
   101c0: 000215b7 lui  a1,0x21
   101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
   101c8: 288000ef jal  ra,10450 <printf>
   101cc: 00c12083 lw   ra,12(sp)
   101d0: 01010113 addi sp,sp,16
   101d4: 00000513 li   a0,0
   101d8: 00008067 ret
```

a.out

RISC-V ABI F D RV32 ABI
ilp32 ilp32f ilp32d ilp32 C int long pointer
32 lip32
ilp32f ilp32d

ISA F D
5 RV32I GCC -march=rv32i ilp32 ABI GCC
-mabi=lib32
RV32IFD ilp32 ilp32f ilp32d
ABI ABI ISA

---

linker relaxation
jump and link 20

gp ±2KiB RISC-V
lui auipc tp ±2KiB

Directive	Description
`.text`	Subsequent items are stored in the text section (machine code).
`.data`	Subsequent items are stored in the data section (global variables).
`.bss`	Subsequent items are stored in the bss section (global variables initialized to 0).
`.section .foo`	Subsequent items are stored in the section named `.foo`.
`.align n`	Align the next datum on a 2^n-byte boundary. For example, `.align 2` aligns the next value on a word boundary.
`.balign n`	Align the next datum on a n-byte boundary. For example, `.balign 4` aligns the next value on a word boundary.
`.globl sym`	Declare that label sym is global and may be referenced from other files.
`.string "str"`	Store the string str in memory and null-terminate it.
`.byte b1,..., bn`	Store the n 8-bit quantities in successive bytes of memory.
`.half w1,...,wn`	Store the n 16-bit quantities in successive memory halfwords.
`.word w1,...,wn`	Store the n 32-bit quantities in successive memory words.
`.dword w1,...,wn`	Store the n 64-bit quantities in successive memory doublewords.
`.float f1,..., fn`	Store the n single-precision floating-point numbers in successive memory words.
`.double d1,..., dn`	Store the n double-precision floating-point numbers in successive memory doublewords.
`.option rvc`	Compress subsequent instructions (see Chapter 7).
`.option norvc`	Don't compress subsequent instructions.
`.option relax`	Allow linker relaxations for subsequent instructions.
`.option norelax`	Don't allow linker relaxations for subsequent instructions.
`.option pic`	Subsequent instructions are position-independent code.
`.option nopic`	Subsequent instructions are position-dependent code.
`.option push`	Push the current setting of all `.options` to a stack, so that a subsequent `.option pop` will restore their value.
`.option pop`	Pop the option stack, restoring all `.options` to their setting at the time of the last `.option push`.

bss 0

.foo

2n .align 2

n .balign 4

sym

str

n 8

n 16

n 32

n 64

n

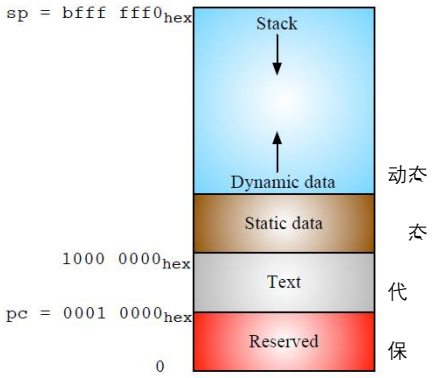
n

7

linker relaxation

.option

.option



sp

static linking

bug bug

dynamic linking
fast linking

stub function

3.8

a.out

Kelly Johnson KISS 1960

RISC-V ISA 60 RISC-V
0
lui auipc

/

RISC-V

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

TIS Committee. Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2. *TIS Committee*, 1995.

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

William of Occam 1320

RV32M RV32I 4.1 RV32M
4.2

srl

srli
t2,a1,4
divu t2,a1,a2

RV32M divide(div) divide unsigned(divu)
RV32M
remainder(rem) remainder unsigned(remu)

RV32M

multiply

multiply **h**igh { –, **u**nsigned, **s**igned **u**nsigned }

{ **div**ide, **rem**ainder } { –, **u**nsigned }

31 25	24 20	19 15	14 12	11 7	6 0	
0000001	rs2	rs1	000	rd	0110011	R mul
0000001	rs2	rs1	001	rd	0110011	R mulh
0000001	rs2	rs1	010	rd	0110011	R mulhu
0000001	rs2	rs1	011	rd	0110011	R mulhu
0000001	rs2	rs1	100	rd	0110011	R div
0000001	rs2	rs1	101	rd	0110011	R divu
0000001	rs2	rs1	110	rd	0110011	R rem
0000001	rs2	rs1	111	rd	0110011	R remu

32
64 64 RISC-V
32 64 32 mul 32
mulh mulhu
mulhsu 64 32
RV32M
64
2
32
4.3 3

sll
slli
t2,a1,4
mul t2,a1,a2

```
# Compute unsigned division of a0 by 3 using multiplication.
   0: aaaab2b7    lui   t0,0xaaaab  # t0 = 0xaaaaaaab
   4: aab28293    addi  t0,t0,-1365 #    = ~ 2^32 / 1.5
   8: 025535b3    mulhu a1,a0,t0    # a1 = ~ (a0 / 1.5)
   c: 0015d593    srli  a1,a1,0x1   # a1 = (a0 / 3)
```

ARM-32 ARM
20 2005 ARM MIPS-32
HI LO MIPS

HI LO

mulhsu
mulhsu

15%

beqz RV32I
trap

mulh mulhu
mulhu mul mulh
mul mul mulh 0 mul
mulh ffffffff mul

C. Gordan Bell

RISC-V RISC-V
RV32M RISC-V RV32M

T. Granlund and P. L. Montgomery. Division by invariant integers using multiplication. *ACM SIGPLAN Notices*, volume 29, pages 61–72. ACM, 1994.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL: https://riscv.org/specifications/

RV32F RV32D

Antoine de Saint Exup'ery, L'Avion, 1940

RV32F RV32D

32 64

5.1 RV32F RV32D 5.2 RV32F
5.3 RV32D ISA RISC-V IEEE 754-2008
[IEEE 2008]

RV32F RV32D 32 f x
RISC-V
2 RISC-V
f
x f 5.4 RV32D RV32F RISC-V
ABI
RV32F RV32D f 32
RV32I x0 f0 0 31 f

IEEE 754-2008
RNE
fcsr 5.5 fcsr

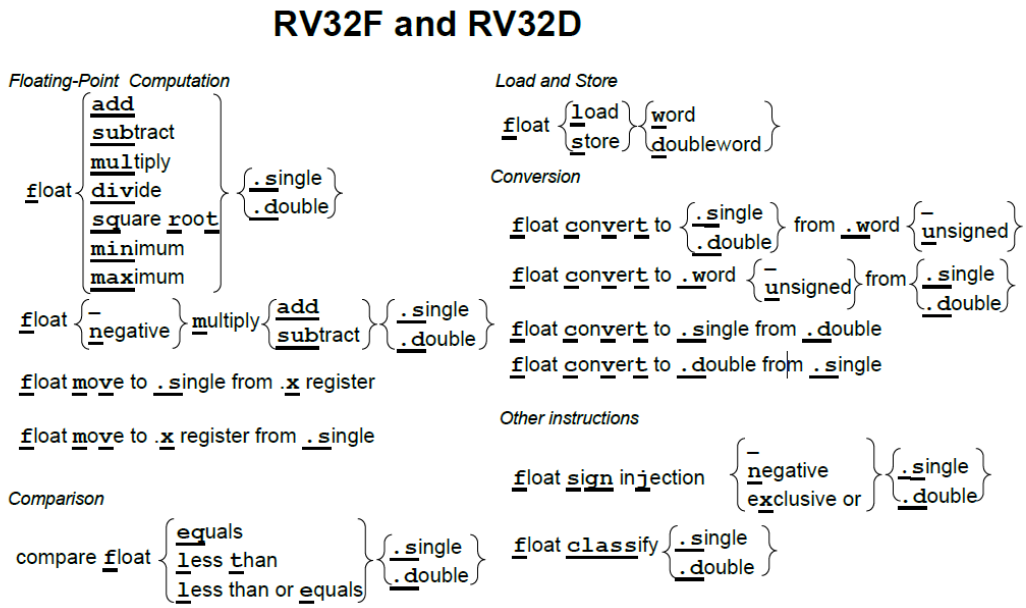
ARM-32 MIPS-32 32 16
x86-32
80

MIPS
John Mashey
16
MIPS ISA

RV32FD

fcsr
fadd.s fto ft1 ft2 rtz fcsr 5.5

32 64 80 x86-32
8 64 RV32FD MIPS-32 ARM-
32 x86-32



RV32F RV32D RISC-V flw fld fsw
fsd lw sw
fadd.s fadd.d fsub.s fsub.d fmul.s fmul.d fdiv.s fdiv.d RV32F RV32D
fsqrt.s fsqrt.d fmin.s fmin.d
fmax.s fmax.d

RV32F
RF32D

RISC-V fmadd.s fmadd.d
fmsub.s fmsub.d
fnmadd.s fnmadd.d fnmsub.s fnmsub.d
-

4 R4 5.2 5.3
R4 R
RV32F RV32D
1 0 feq.s feq.d flt.s flt.d fle.s fle.d

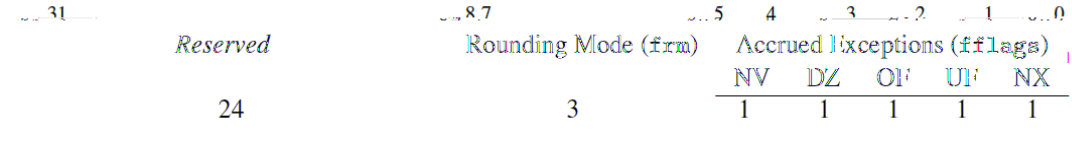
f1 <f2 Exit
flt x5 f1 f2 f1 < f2 x5 = 1; x5 = 0
bne x5 x0 Exit x5 = 0 Exit

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	010	rd	0000111	I flw
imm[11:5]		rs2	rs1	010	imm[4:0]	0100111	S fsw
rs3	00	rs2	rs1	rm	rd	1000011	R4
rs3	00	rs2	rs1	rm	rd	1000111	R4
rs3	00	rs2	rs1	rm	rd	1001011	R4
rs3	00	rs2	rs1	rm	rd	1001111	R4
0000000		rs2	rs1	rm	rd	1010011	R fadd.s
0000100		rs2	rs1	rm	rd	1010011	R fsub.s
0001000		rs2	rs1	rm	rd	1010011	R fmul.s
0001100		rs2	rs1	rm	rd	1010011	R fdiv.s
0001100		00000	rs1	rm	rd	1010011	R fsqrt.s
0010000		rs2	rs1	000	rd	1010011	R fsgnj.s
0010000		rs2	rs1	001	rd	1010011	R fsgnjn.s
0010000		rs2	rs1	010	rd	1010011	R fsgnjx.s
0010100		rs2	rs1	000	rd	1010011	R fmin.s
0010100		rs2	rs1	001	rd	1010011	R fmax.s
1100000		00000	rs1	rm	rd	1010011	R fcvt.w.s
1100000		00001	rs1	rm	rd	1010011	R
1110000		00000	rs1	000	rd	1010011	R fmv.x.w
1010000		rs2	rs1	010	rd	1010011	R feq.s
1010000		rs2	rs1	001	rd	1010011	R flt.s
1010000		rs2	rs1	000	rd	1010011	R fle.s
1110000		00000	rs1	001	rd	1010011	R fclass.s
1101000		00000	rs1	rm	rd	1010011	R fcvt.s.w
1101000		00001	rs1	rm	rd	1010011	R
1111000		00000	rs1	000	rd	1010011	R fmv.w.x

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	011	rd	0000111	I fld
imm[11:5]		rs2	rs1	011	imm[4:0]	0100111	S fsd
rs3	01	rs2	rs1	rm	rd	1000011	R4
rs3	01	rs2	rs1	rm	rd	1000111	R4
rs3	01	rs2	rs1	rm	rd	1001011	R4
rs3	01	rs2	rs1	rm	rd	1001111	R4
0000001		rs2	rs1	rm	rd	1010011	R fadd.d
0000101		rs2	rs1	rm	rd	1010011	R fsub.d
0001001		rs2	rs1	rm	rd	1010011	R fmul.d
0001101		rs2	rs1	rm	rd	1010011	R fdiv.d
0001101		00000	rs1	rm	rd	1010011	R fsqrt.d
0010001		rs2	rs1	000	rd	1010011	R fsgnj.d
0010001		rs2	rs1	001	rd	1010011	R fsgnjn.d
0010001		rs2	rs1	010	rd	1010011	R fsgnjx.d
0010101		rs2	rs1	000	rd	1010011	R fmin.d
0010101		rs2	rs1	001	rd	1010011	R fmax.d
0100000		00001	rs1	rm	rd	1010011	R fcvt.s.d
0100001		00000	rs1	rm	rd	1010011	R fcvt.d.s
1010001		Rs2	rs1	010	rd	1010011	R feq.d
1010001		rs2	rs1	001	rd	1010011	R flt.d
1010001		rs2	rs1	000	rd	1010011	R fle.d
1110001		00000	rs1	001	rd	1010011	R fclass.d
1100001		00000	rs1	rm	rd	1010011	R fcvt.w.d
1100001		00001	rs1	rm	rd	1010011	R
1101001		00000	rs1	rm	rd	1010011	R fmv.d.w
1101001		00001	rs1	rm	rd	1010011	R

fcvt.s.d fcvt.d.s fmv.x.w fmv.w.x

63 32	31 0	
	f0 / ft0	FP Temporary
	f1 / ft1	FP Temporary
	f2 / ft2	FP Temporary
	f3 / ft3	FP Temporary
	f4 / ft4	FP Temporary
	f5 / ft5	FP Temporary
	f6 / ft6	FP Temporary
	f7 / ft7	FP Temporary
	f8 / fs0	FP Saved register
	f9 / fs1	FP Saved register
	f10 / fa0	FP Function argument, return value
	f11 / fa1	FP Function argument, return value
	f12 / fa2	FP Function argument
	f13 / fa3	FP Function argument
	f14 / fa4	FP Function argument
	f15 / fa5	FP Function argument
	f16 / fa6	FP Function argument
	f17 / fa7	FP Function argument
	f18 / fs2	FP Saved register
	f19 / fs3	FP Saved register
	f20 / fs4	FP Saved register
	f21 / fs5	FP Saved register
	f22 / fs6	FP Saved register
	f23 / fs7	FP Saved register
	f24 / fs8	FP Saved register
	f25 / fs9	FP Saved register
	f26 / fs10	FP Saved register
	f27 / fs11	FP Saved register
	f28 / ft8	FP Temporary
	f29 / ft9	FP Temporary
	f30 / ft10	FP Temporary
	f31 / ft11	FP Temporary
32	32	



frm

To	From			
	32b signed integer (w)	32b unsigned integer (wu)	32b floating point (s)	64b floating point (
.d	32b signed integer (w)	–	–	fcvt.w.s	fcvt.w
ı.d	32b unsigned integer (wu)	–	–	fcvt.wu.s	fcvt.wu
.d	32b floating point (s)	fcvt.s.w	fcvt.s.wu	–	fcvt.s
	64b floating point (d)	fcvt.d.w	fcvt.d.wu	fcvt.d.s	–

RV32F RV32D 32 32 32 64
5.6
10
RV32F f fmv.x.w x
fmv.w.x

RV32F RV32D
IEEE 754

1. fsgnj.s fsgnj.d rs2
2. fsgnjn.s fsgnjn.d rs2
3. fsgnjx.s fsgnjx.d rs1 rs2

37 3.4

1.

fmv.s rd rs fsgnj.s rd rs rs
fmv.d rd rs sgnj.d rd rs rs

```
void daxpy(size_t n, double a, const double x[], double y[])
{
  for (size_t i = 0; i < n; i++) {
    y[i] = a*x[i] + y[i];
  }
}
```

ISA	ARM-32	ARM Thumb-2	MIPS-32	microMIPS	x86-32	RV32FD	RV32FD+RV32C
Instructions	10	10	12	12	16	11	11
Per Loop	6	6	7	7	6	7	7
Bytes	40	28	48	32	50	44	28

2.

fneg.s rd rs fsgnjn.s rd rs rs
fneg.d rd rs fsgnjn.d rd rs rs

3. 0 0= 0 1 1= 0

fabs.s rd rs fsgnjx.s rd rs rs
fabs.d rd rs sgnjx.d rd rs rs

classify fclass.s fclass.d
10
10

1 0

x[rd]	
0	f [*rs1*]
1	f [*rs1*]
2	f [*rs1*]
3	f [*rs1*] -0
4	f [*rs1*] +0
5	f [*rs1*]
6	f [*rs1*]
7	f [*rs1*] +
8	f [*rs1*] (signaling)NaN
9	f [*rs1*] (quiet)NaN

DAXPY ISA 5.7

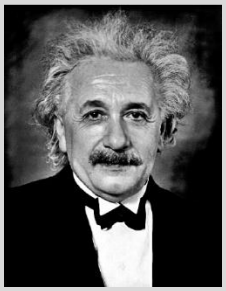
```
# RV32FD (7 insns in loop; 11 insns/44 bytes total; 28 bytes RVC)
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], fa0 is a
   0: 02050463 beqz    a0,28               # if n == 0, jump to Exit
   4: 00251513 slli    a0,a0,0x2           # a0 = n*8
ment)    8: 00a60533 add     a0,a2,a0         # a0 = address of x[n] (last ele
       Loop:
   c: 0005b787 fld     fa5,0(a1)           # fa5 = x[]
  10: 00063707 fld     fa4,0(a2)           # fa4 = y[]
  14: 00860613 addi    a2,a2,8             # a2++ (increment pointer to y)
  18: 00858593 addi    a1,a1,8             # a1++ (increment pointer to x)
  1c: 72a7f7c3 fmadd.d fa5,fa5,fa0,fa4 # fa5 = a*x[i] + y[i]
  20: fef63c27 fsd     fa5,-8(a2)          # y[i] = a*x[i] + y[i]
  24: fea614e3 bne     a2,a0,c             # if i != n, jump to Loop
      Exit:
  28: 00008067         ret                 # return
```

```
# ARM-32 (6 insns in loop; 10 insns/40 bytes total; 28 bytes Thumb-2)
# r0 is n, d0 is a, r1 is pointer to x[0], r2 is pointer to y[0]
   0: e3500000 cmp     r0, #0              # compare n to 0
   4: 0a000006 beq     24 <daxpy+0x24>     # if n == 0, jump to Exit
zero, 0x30(sp)\n\tbne aa0, 0xc, 0x1\n\t" "=m" (fa5) : [k] "r" (&a)
      Loop:
o x    c: ecb16b02 vldmia   r1!, {d6}           # d6 = x[i], increment pointer t
  10: ed927b00 vldr    d7, [r2]            # d7 = y[i]
  14: ee067b00 vmla.f64 d7, d6, d0         # d7 = a*x[i] + y[i]
r to y  18: eca27b02 vstmia   r2!, {d7}           # y[i] = a*x[i] + y[i], incr. pt
  1c: e1520000 cmp     r2, r0              # i vs. n
  20: 1afffff9 bne     c <daxpy+0xc>       # if i != n, jump to Loop
      Exit:
  24: e12fff1e bx      lr                  # return
```

```
# MIPS-32 (7 insns in loop; 12 insns/48 bytes total; 32 bytes microMIPS)
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], f12 is a
  0: 10800009 beqz    a0,28 <daxpy+0x28>  # if n == 0, jump to Exit
  4: 000420c0 sll     a0,a0,0x3           # a0 = n*8 (filled branch delay slot)
  8: 00c42021 addu    a0,a2,a0            # a0 = address of x[n] (last element)
Loop:
  c: 24c60008 addiu   a2,a2,8             # a2++ (increment pointer to y)
 10: d4a00000 ldc1    $f0,0(a1)          # f0 = x[i]
 14: 24a50008 addiu   a1,a1,8            # a1++ (increment pointer to x)
 18: d4e20000 ldc1    $f2,0(a2)          # f2 = y[i]
 1c: 4c406021 madd.d  $f0,$f2,$f12,$f0   # f0 = a*x[i] + y[i]
 20: 14c4fffa bne     a2,a0,c <daxpy+0xc> # if i != n, jump to Loop
 24: f4c0fff8 sdc1    $f0,-8(a2)         # y[i] = a*x[i] + y[i] (filled delay slot)
Exit:
 28: 03e00008 jr      ra                 # return
 2c: 00000000 nop                        # (unfilled branch delay slot)
```

```
# x86-32 (6 insns in loop; 16 insns/50 bytes total)
# eax is i, n is in memory at esp+0x8, a is in memory at esp+0xc
# pointer to x[0] is in memory at esp+0x14
# pointer to y[0] is in memory at esp+0x18
  0: 53                push    ebx                 # save ebx
  1: 8b 4c 24 08       mov     ecx,[esp+0x8]       # ecx has copy of n
  5: c5 fb 10 4c 24 0c vmovsd  xmm1,[esp+0xc]      # xmm1 has a copy of a
  b: 8b 5c 24 14       mov     ebx,[esp+0x14]      # ebx points to x[0]
  f: 8b 54 24 18       mov     edx,[esp+0x18]      # edx points to y[0]
 13: 85 c9             test    ecx,ecx             # compare n to 0
 15: 74 19             je      30 <daxpy+0x30>     # if n==0, jump to Exit
 17: 31 c0             xor     eax,eax             # i = 0 (since x^x==0)
Loop:
 19: c5 fb 10 04 c3    vmovsd  xmm0,[ebx+eax*8]    # xmm0 = x[i]
 1e: c4 e2 f1 a9 04 c2 vfmadd213sd xmm0,xmm1,[edx+eax*8] # xmm0 = a*x[i] + y[i]
 24: c5 fb 11 04 c2    vmovsd  xmm0,xmm1,[edx+eax*8] # y[i] = a*x[i] + y[i]
       2c: 39 c1             cmp     ecx,eax             # compare i vs n
Loop   2e: 75 e9             jne     19 <daxpy+0x19>     # if i!=n, jump to
      Exit:
       30: 5b                pop     ebx                 # restore ebx
       31: c3                ret                         # return
```

test ecx ecx xor eax,eax



Albert Einstein 1933

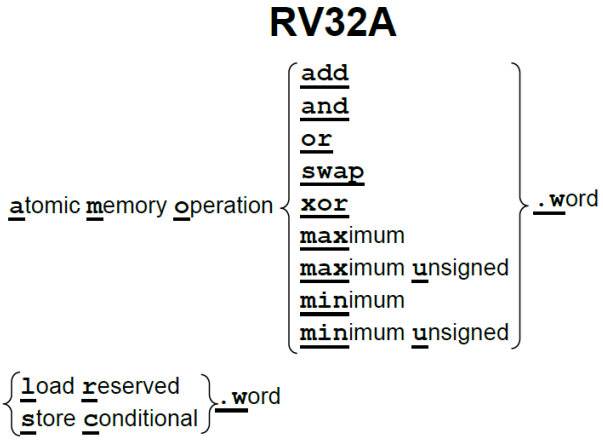
ISA RV32A

https://en.wikipedia.org/wiki/Synchronization
https://zh.wikipedia.org/wiki/%E5%90%8C%E6%AD%A5 RISC-V 2.1
[Patterson and Hennessy 2017]

RV32A

- AMO
- / load reserved / store conditional

6.1 RV32A 6.2



31 25	24 20	19 15	14 12	11 7	6 0	
00010 aq rl	00000	rs1	010	rd	0101111	R lr.w
00011 aq rl	rs2	rs1	010	rd	0101111	R sc.w
00001 aq rl	rs2	rs1	010	rd	0101111	R amoswap.w
00000 aq rl	rs2	rs1	010	rd	0101111	R amoadd.w
00100 aq rl	rs2	rs1	010	rd	0101111	R amoxor.w
01100 aq rl	rs2	rs1	010	rd	0101111	R amoand.w
01000 aq rl	rs2	rs1	010	rd	0101111	R amoor.w
10000 aq rl	rs2	rs1	010	rd	0101111	R amomin.w
10100 aq rl	rs2	rs1	010	rd	0101111	R amomax.w
11000 aq rl	rs2	rs1	010	rd	0101111	R amominu.w
11100 aq rl	rs2	rs1	010	rd	0101111	R amomaxu.w

AMO

AMO LR/SC

cache

0

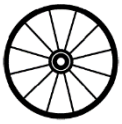
0

RV32A

- compare-and-swap

[Herlihy 1991]

ISA 3



1

RV32FD multiply-add

6.3

lr/sc 实 内存字 M[a0] -交 作

```
# Compare-and-swap (CAS) memory word M[a0] using lr/sc.
# Expected old value in a1; desired new value in a2.
   0: 100526af    lr.w  a3,(a0)     # Load old value
   4: 06b69e63    bne   a3,a1,80    # Old value equals a1?
   8: 18c526af    sc.w  a3,a2,(a0)  # Swap in new value if so
   c: fe069ae3    bnez  a3,0        # Retry if store failed
      ... code following successful CAS goes here ...
  80:                               # Unsuccessful CAS.
```

```
#加      值
#        值与 a1   否
#    则存入     值
#如  存入失          尝
     -交     功之后   代
#    -交   不   功
```

```
# Critical section guarded by test-and-set spinlock using an AMO.
   0: 00100293    li           t0,1        # Initialize lock value
   4: 0c55232f    amoswap.w.aq t1,t0,(a0)  # Attempt to acquire lock
   8: fe031ee3    bnez         t1,4        # Retry if unsuccessful
      ... critical section goes here ...
  20: 0a05202f    amoswap.w.rl x0,x0,(a0)  # Release lock.
```

```
#初始化
#尝     取
#如  失          尝
  临   区代
#
```

AMO /

AMO I/O

I/O 6.3

RISC-V relaxed memory consistency model

6.2 RV32A aq

rl aq AMO

rl

[Adve and Gharachorloo 1996]

MIPS-32 MIPS ISA

/

RV32A RISC-V

RV32A

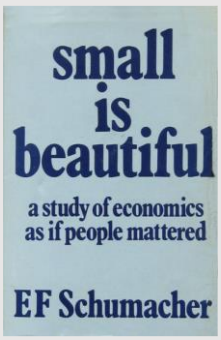
S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 1991.

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

A. Waterman and K. Asanovi´c, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

E. F. Schumacher, 1973



ISA

ARM MIPS

ARM ARM Thumb Thumb 2 MIPS

MIPS16 microMIPS ISA

RV32C 32 RISC-V

16

RV32C

ISA 7.1 RV32C

RISC-V RVC

16

ao-a5 so-s1 sp ra

RV32C

load store

RV32C

Integer Computation

c.add { – / immediate }

c.add immediate * 16 to stack pointer

c.add immediate * 4 to stack pointer nondestructive

c.subtract

c. { shift left logical / shift right arithmetic / shift right logical } immediate

c.and { – / immediate }

c.or

c.move

c.exclusive or

c.load { – / upper } immediate

Loads and Stores

c. { – / float } { load / store } word { – / using stack pointer }

c.float { load / store } doubleword { – / using stack pointer }

Control transfer

c.branch { equal / not equal } to zero

c.jump { – / and link }

c.jump { – / and link } register

Other instructions

c.environment break

Benchmark	ISA	ARM Thumb-2	microMIPS	x86-32	RV32I+RVC
Insertion-Sort	Instructions	18	24	20	19
	Bytes	46	56	45	52
DAXPY	Instructions	10	12	16	11
	Bytes	28	32	50	28

7.3 7.4 DAXPY RV32C RV32C

RV32C 32 A 16
RV32C 32 RISC-V
7.3 4 32 RV32I

addi a4,x0,1 # i = 1
16 RV32C
c.li a4,1 # addi a4,x0,1 i = 1
RV32C load
c.li 7.3 4 c.li 2
7.3 10
add a2,x0,a3 # a2 a[j]
16 RV32C
c.mv a2,a3 # add a2,x0,a3 a2 a[j]
RV32C move 16
RV32C
16 32 7.6 7.8
RV32C 32
RISC-V 8000 400
5% 100,000 cache

RV32C
9 1.5 Thumb-2 RV32C Load and Store
Multiple
RV32G RV32C RV32G
Thumb-2 ARM-32 ISA
ISA ISA ARM-32
Thumb-2 RV32GC ISA RISC-V

RV32C

macrofusion RISC-V
16 RV32C 32 RV32I

7.2 ISA DAXPY
19 RV32I 12 RV32C
DAXPY

9 1.5
RV32G RV32GC 37%
RV32C

RV32C
RV32 RV32IC Thumb-2 ISA 16
ARMv7 Thumb-2 Compare and Branch on
Zero ARMv7 Reverse Subtarct with Carry microMIPS
MIPS32 microMIPS 2 MIPS32
4 RISC-V 2

Blaise Pascal, 1656

Niklaus Wirth

RV32C RISC-V

RISC-V

RISC-V RV32C

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

```
# RV32C (19 instructions, 52 bytes)
# a1 is n, a3 points to a[0], a4 is i, a5 is j, a6 is x
  0: 00450693 addi   a3,a0,4   # a3 is pointer to a[i]
  4: 4705     c.li   a4,1      # (expands to addi a4,x0,1) i = 1
Outer Loop:
  6: 00b76363 bltu   a4,a1,c   # if i < n, jump to Continue Outer loop
  a: 8082     c.ret            # (expands to jalr x0,ra,0) return from function
Continue Outer Loop:
  c: 0006a803 lw     a6,0(a3)  # x = a[i]
 10: 8636     c.mv   a2,a3     # (expands to add a2,x0,a3) a2 is pointer to a[j]
 12: 87ba     c.mv   a5,a4     # (expands to add a5,x0,a4) j = i
InnerLoop:
 14: ffc62883 lw     a7,-4(a2) # a7 = a[j-1]
 18: 01185763 ble    a7,a6,26  # if a[j-1] <= a[i], jump to Exit InnerLoop
 1c: 01162023 sw     a7,0(a2)  # a[j] = a[j-1]
 20: 17fd     c.addi a5,-1     # (expands to addi a5,a5,-1) j--
to a[j]  22: 1671     c.addi a2,-4     # (expands to addi a2,a2,-4)decr a2 to point
nerLoop  24: fbe5     c.bnez a5,14     # (expands to bne a5,x0,14)if j!=0,jump to In
         Exit InnerLoop:
4        26: 078a     c.slli a5,0x2    # (expands to slli a5,a5,0x2) multiply a5 by
of a[j]  28: 97aa     c.add  a5,a0     # (expands to add a5,a5,a0)a5 = byte address
         2a: 0107a023 sw     a6,0(a5)  # a[j] = x
         2e: 0705     c.addi a4,1      # (expands to addi a4,a4,1) i++
to a[i]  30: 0691     c.addi a3,4      # (expands to addi a3,a3,4) incr a3 to point
         32: bfd1     c.j    6         # (expands to jal x0,6) jump to Outer Loop
```

c.

```
# RV32DC (11 instructions, 28 bytes)
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], fa0 is a
  0: cd09     c.beqz a0,1a        # (expands to beq a0,x0,1a) if n==0, jump to Exit
  2: 050e     c.slli a0,a0,0x3    # (expands to slli a0,a0,0x3) a0 = n*8
  4: 9532     c.add a0,a2         # (expands to add a0,a0,a2) a0 = address of x[n]
Loop:
  6: 2218     c.fld fa4,0(a2)     # (expands to fld fa4,0(a2) ) fa5 = x[]
  8: 219c     c.fld fa5,0(a1)     # (expands to fld fa5,0(a1) ) fa4 = y[]
  a: 0621     c.addi a2,8         # (expands to addi a2,a2,8) a2++ (incr. ptr to y)
  c: 05a1     c.addi a1,8         # (expands to addi a1,a1,8) a1++ (incr. ptr to x)
  e: 72a7f7c3 fmadd.d fa5,fa5,fa0,fa4 # fa5 = a*x[i] + y[i]
 12: fef63c27 fsd fa5,-8(a2)      # y[i] = a*x[i] + y[i]
6: fea618e3 bne a2,a0,6          # if i != n, jump to Loop
it:                                                              Ex
a: 8082     ret                 # (expands to jalr x0,ra,0) return from function      1
```

c.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
000			nzimm[5]	0					nzimm[4:0]					01		CI c.nop
000			nzimm[5]	rs1/rd≠0					nzimm[4:0]					01		CI c.addi
001			imm[11|4|9:8|10|6|7|3:1|5]											01		CJ c.jal
010			imm[5]	rd≠0					imm[4:0]					01		CI c.li
011			nzimm[9]	2					nzimm[4|6|8:7|5]					01		CI c.addi16sp
011			nzimm[17]	rd≠{0, 2}					nzimm[16:12]					01		CI c.lui
100			nzuimm[5]	00		rs1′/rd′			nzuimm[4:0]					01		CI c.srli
100			nzuimm[5]	01		rs1′/rd′			nzuimm[4:0]					01		CI c.srai
100			imm[5]	10		rs1′/rd′			imm[4:0]					01		CI c.a
100			0	11		rs1′/rd′			00		rs2′			01		CR c.
100			0	11		rs1′/rd′			01		rs2′			01		CR c.
100			0	11		rs1′/rd′			10		rs2′			01		CR c.
100			0	11		rs1′/rd′			11		rs2′			01		CR c.
101			imm[11|4|9:8|10|6|7|3:1|5]											01		CJ c.
110			imm[8|4:3]			rs1′			imm[7:6|2:1|5]					01		CB c.
111			imm[8|4:3]			rs1′			imm[7:6|2:1|5]					01		CB c.

a0-a5 s0-s1 sp ra

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
000			0								0			00		CIW *Illegal instruction*
000			nzuimm[5:4|9:6|2|3]								rd′			00		CIW c.addi4spn
001			uimm[5:3]			rs1′			uimm[7:6]		rd′			00		CL c.fld
010			uimm[5:3]			rs1′			uimm[2|6]		rd′			00		CL c.lw
011			uimm[5:3]			rs1′			uimm[2|6]		rd′			00		CL c.flw
101			uimm[5:3]			rs1′			uimm[7:6]		rs2′			00		CL c.fsd
110			uimm[5:3]			rs1′			uimm[2|6]		rs2′			00		CL c.sw
111			uimm[5:3]			rs1′			uimm[2|6]		rs2′			00		CL c.fsw

a0-a5 s0-s1 sp ra

15 14 13	12	11 10 9 8 7	6 5 4 3 2	1 0	
000	nzuimm[5]	rs1/rd≠0	nzuimm[4:0]	10	CI c.slli
000	0	rs1/rd≠0	0	10	CI c.slli64
001	uimm[5]	rd	uimm[4:3|8:6]	10	CSS c.fldsp
010	uimm[5]	rd≠0	uimm[4:2|7:6]	10	CSS c.lwsp
011	uimm[5]	rd	uimm[4:2|7:6]	10	CSS c.flwsp
100	0	rs1≠0	0	10	CJ c.jr
100	0	rd≠0	rs2≠0	10	CR c.mv
100	1	0	0	10	CI c.ebreak
100	1	rs1≠0	0	10	CJ c.jalr
100	1	rs1/rd≠0	rs2≠0	10	CR c.add
101	uimm[5:3|8:6]		rs2	10	CSS c.fsdsp
110	uimm[5:2|7:6]		rs2	10	CSS c.swsp
111	uimm[5:2|7:6]		rs2	10	CSS c.fswsp

Format	Meaning	15 14 13	12 11 10	9 8 7	6 5 4 3 2	1 0
CR	Register	funct4		rd/rs1	rs2	op
Immediate		funct3	imm	rd/rs1	imm	op
Stack-relative Store		funct3	imm		rs2	op
Wide Immediate		funct3	imm		rd′	op
Load		funct3	imm	rs1′ imm	rd′	op
Store		funct3	imm	rs1′ imm	rs2′	op
Branch		funct3	offset	rs1′	offset	op
Jump		funct3	jump target			op

CI			
CSS			
CIW			
CL			
CS			
CB			
CJ			

rd',rs1' rs2' a0-a5 s0-s1 sp ra

Seymour Cray

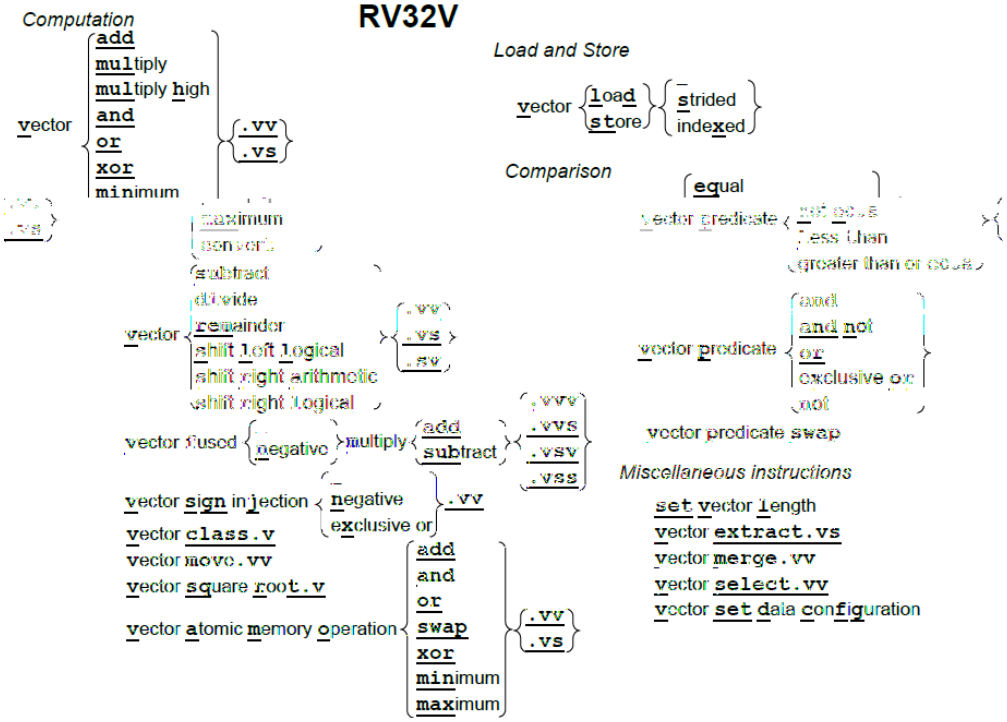
8 16
(SIMD Single Instruction Multiple Data)
SIMD 64 8 16 32
SIMD
load store
64 SIMD
SIMD ISA
SIMD SIMD
SIMD SIMD ISA

1997
MMX
SIMD
1999
SIMD SSE,
Streaming SIMD
Extensions 2010
AVX

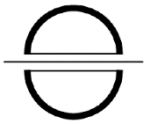
MMX

(https
//www.youtube.com/wat
ch?v=paU16B-bZEA)

RISC-V SIMD



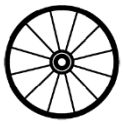
SIMD



SIMD

SIMD

SIMD ISA



[Hennessy and Patterson 2011] 4 G RV32V ISA

8.1 RV32V RV32V

8.1

RV32I RV32M RV32F RV32D RV32A

.vv

.vs x f

v DAXPY 55 5.7

- rs1

.sv

.sv

fused -

.vvv .vvs .vsv .vss

8.1

RV32V 32 v

4096 32

16 64 32 32 64 16 128 8

ISA

mvl

vl

mvl mvl vl 8

vpi
RV32V

8.1 8.9 SIMD

64
1024
512 512/8=64 mvl 64 mvl

vsetdcfg 8.2 RV32V
RV64V RV32V
F32 RV32FV F64
RV32FDV RV32V 16 F16 RV32V RV32F
F16 F32

Type	Floating Point		Signed Integer		Unsigned Integer	
Width	Name	vetype	Name	vetype	Name	vetype
8 bits	–	–	X8	10 100	X8U	11 100
16 bits	F16	01 101	X16	10 101	X16U	11 101
32 bits	F32	01 110	X32	10 110	X32U	11 110
64 bits	F64	01 111	X64	10 111	X64U	11 111

vetype

MMX SIMD

RV32V
SIMD

RV32V

load store
7

load

store

Load Store
Load vld
vl store
vld
a0 1024 v0 X32 vld v0, 0(a0) 1024
1028 1032 1036 vl

vlds vsts vlds vsts
vld vst vld vst
vld vst
vlds vsts

ao 1024 a1 64 vlds vo,
ao, a1
vl

vldx vstx

ao 1024 v1
16 48 80 160 vldx vo, ao, v1

load
(*c gt*)
store
(*ec gt*)

Load Store

RV32G 64
64

SIMD
4 64 8 32
16 16 32 8
SIMD ISA
RV32V ISA

SIMD SIMD
SIMD RV32V

if
8.1
1 0

Gather scatter

i 1 i 0
RV32V 8 vpi vpand vpandn vpor vpxor vpnot
RV32V vpo vp1
1 RV32V vpswap
vpo vp1

v3

```
vplt.vs    vpo,v3,xo    #    v3 < o          1
add.vv,vpo  vo,v1,v2    #    vo        1          v1+v2
```

vpo 1 0 vo
v1 v2 vo

vsetdcfg
setvl vl mvl

mvl
setvl
RV32V
vselect

```
# vindices    o   mvl-1          vsrc
vselect vedst, vsrc, vindices
```

v2 8 0 4 2 vselect vo, v1, v2 v1 8
vo 0 v1 0 v0 1 v1 4
vo 2 v1 2 vo 3
vmerge

0 vsrc1 1 vsrc2

vpo i vdest i vsrc1 i o

vsrc2 i 1

vmerge,vpo vdest, vsrc1, vsrc2

vpo 1 0 0 1 v1 1 2 3 4 v2

10 20 30 40 vmerge,vpo vo, v1, v2 vo 10

2 3 40

start vsrc

vextract vdest, vsrc, start

vl 64 a0 32 vextract vo,v1,ao v1

32 vo

vextract

vextract

1

```
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], fa0 is a
  0:  li t0, 2<<25
  4:  vsetdcfg t0             # enable 2 64b Fl.Pt. registers
loop:
  8:  setvl  t0, a0           # vl = t0 = min(mvl, n)
  c:  vld    v0, a1           # load vector x
 10:  slli   t1, t0, 3        # t1 = vl * 8 (in bytes)
 14:  vld    v1, a2           # load vector y
 18:  add    a1, a1, t1       # increment C pointer to x by vl*8
 1c:  vfmadd v1, v0, fa0, v1  # v1 += v0 * fa0 (y = a * x + y)
 20:  sub    a0, a0, t0       # n -= vl (t0)
[illegible]
 28:  add    a2, a2, t1       # increment C pointer to y by vl*8
 2c:  bnez   a0, loop         # repeat if n != 0
 30:  ret                     # return
```

RISC-V V
RISC-V

SIMD

V
RISC
ISA

8.3 RV32V DAXPY 55 5.7

RV32V DAXPY

x y 8

vcfgd

F64 8.2
vo v1
RV32V 1024
8
mvl 64
setvl mvl n
vl to n
64 mvl vl n mvl
x y n setvl
to vl 10
c vld load a1 x
x x vl vo slli
8 x y
14 vld vl load v1
add x
1c vfmadd x vo vl a
fo y v1 vl vl y v1
20 sub
n ao vl vst vl
y 28 add y
n ao 0 ret

setvl

vl
n
n

10
n 64
19 13 SIMD

NEON SIMD

DAXPY

SIMD DAXPY
SIMD 8 8

83 8.5 DAXPY MIPS SIMD MSA
MSA 128 MSA SIMD
RV32V MSA n
n - MSA
8.5 3c 4c n
0 10
18 splati.d a
SIMD w2 SIMD
SIMD
1c ld.d y load SIMD wo
y x load SIMD w1

8.5
vl
SIMD
RV32V

n = 0
RV32V n = 0
nops

28 x 2c

y y
34 SIMD store y
n n

MIPS MSA DAXPY 7 6
4 1 0.5

ISA	MIPS-32 MSA	x86-32 AVX2	RV32FDV
Instructions (static)	22	29	13
Bytes (static)	88	92	52
Instructions per Main Loop	7	6	10
Results per Main Loop	2	4	64
Instructions (dynamic, n=1000)	3511	1517	163

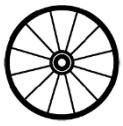
84 8.6 Intel SIMD
SSE 128 SIMD xmm AVX
256 SIMD ymm
0 25 load 256 ymm a
n 4 SSE
AVX 8.6
DAXPY 27 AVX vmovapd x 4
load ymm0 2c AVX vfmadd213pd a ymm2 x ymm0
4 4 y ecx+edx*8 4
ymm0 32 AVX vmovapd 4 y

MIPS MSA 3e 57 n 4
SSE
x86-32 AVX2 DAXPY 6 12 8
2 1

Illiac IV SIMD
64 64 FPU Illiac IV 100
10 1000MFLOPS
15MFLOPS 1966 800 1972
3100 256 FPU 64 1965
1976 Cray-1
[Falk 1976]

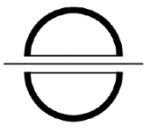
Jim Smith 1994

8.4 RV32IFDV MIPS-32 MSA x86-32 AVX2 DAXPY
SIMD MIPS-32
MSA x86-32 AVX2 SIMD
SIMD n SIMD

8.3 RV32V

SIMD RV32V n
n 0 RV32V RV32V
vl=0
SIMD SIMD
RV32V 10 20 SIMD 2 4
RV32V 64

8.4 29 5.8 DAXPY
SIMD
2 4 SIMD RV32V
1.2 1.4 1/43
SIMD
MIPS-32 x86-32
ISA SIMD
SIMD SIMD ISA
MIPS-32 x86-32
ISA vfmadd213pd

RV32V

RV32V mvl
1024 4096
256 8.3

SIMD ISA ISA
RV32V ISA
SIMD ISA ISA
RV32V ARM-32 MIPS-32 x86-32 SIMD

- - RISC-V

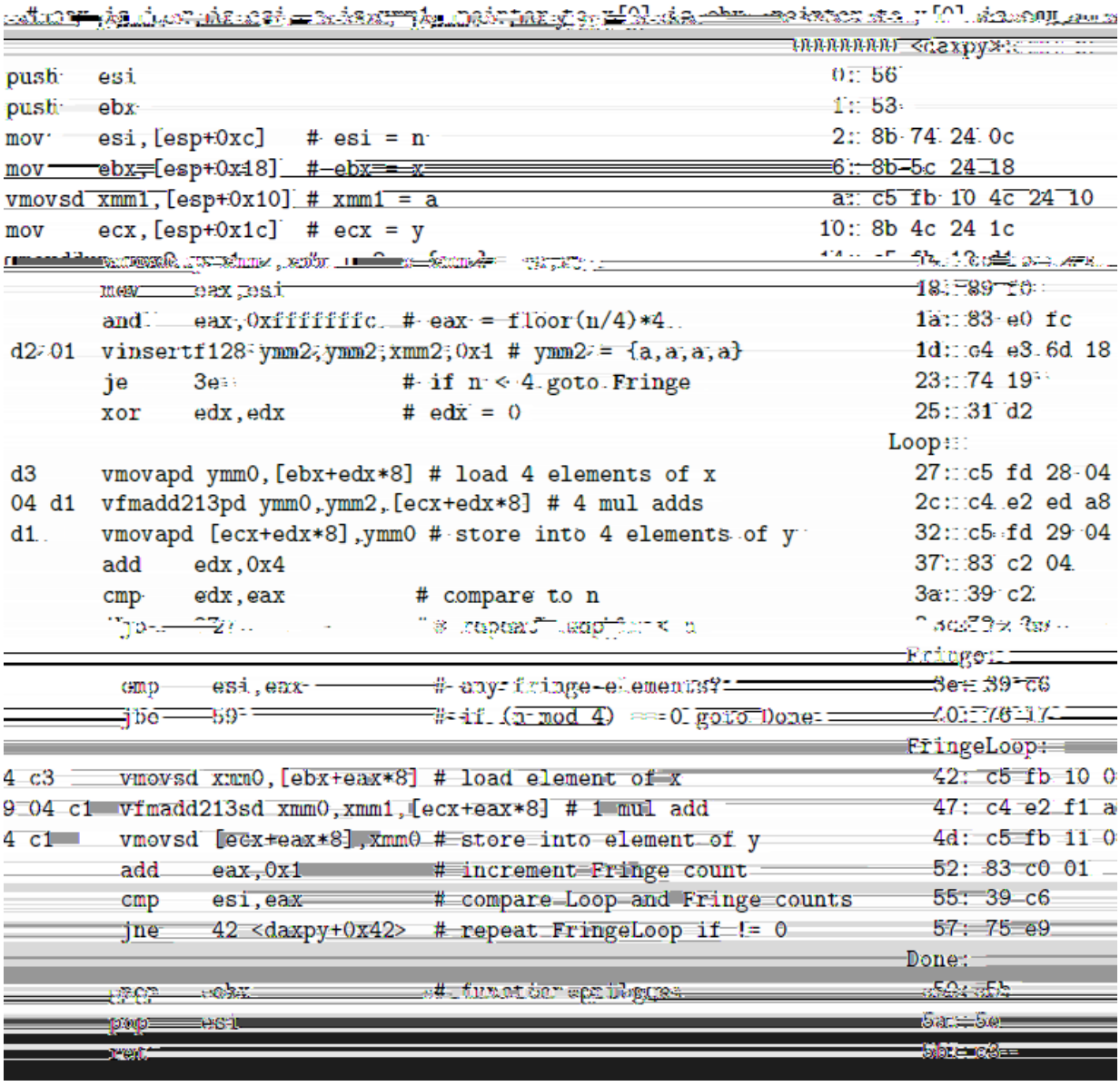
H. Falk. What went wrong V: Reaching for a gigaflop: The fate of the famed Illiac IV was shaped by both research brilliance and real-world disasters. *IEEE spectrum*, 13(10):65–70, 1976.

J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/.

```
# a0 is n, a2 is pointer to x[0], a3 is pointer to y[0], $w13 is a
00000000 <daxpy>:
  0: 2405fffe  li       a1,-2
  4: 00852824  and      a1,a0,a1        # a1 = floor(n/2)*2 (mask bit 0)
  8: 000540c0  sll      t0,a1,0x3       # t0 = byte address of a1
  c: 00e81821  addu     v1,a3,t0        # v1 = &y[a1]
 10: 10e30009  beq      a3,v1,38        # if y==&y[a1] goto Fringe (t0==0 so n is 0 | 1)
 14: 00c01025  move     v0,a2           # (delay slot) v0 = &x[0]
 18: 78786899  splati.d $w2,$w13[0]     # w2 = fill SIMD register with copies of a
Loop:
 1c: 78003823  ld.d     $w0,0(a3)        # w0 = 2 elements of y
 20: 24e70010  addiu    a3,a3,16         # increment C pointer to y by 2 Fl.Pt. numbers
 24: 78001063  ld.d     $w1,0(v0)        # w1 = 2 elements of x
 28: 24420010  addiu    v0,v0,16         # increment C pointer to x by 2 Fl.Pt. numbers
 2c: 7922081b  fmadd.d  $w0,$w1,$w2      # w0 = w0 + w1 * w2
 30: 1467fffa  bne      v1,a3,1c         # if (end of y != ptr to y) go to Loop
 34: 7bfe3827  st.d     $w0,-16(a3)      # (delay slot) store 2 elts of y
Fringe:
 38: 10a40005  beq      a1,a0,50         # if (n is even) goto Done
 3c: 00c83021  addu     a2,a2,t0         # (delay slot) a2 = &x[n-1]
 40: d4610000  ldc1     $f1,0(v1)        # f1 = y[n-1]
 44: d4c00000  ldc1     $f0,0(a2)        # f0 = x[n-1]
 48: 4c206b61  madd.d   $f13,$f1,$f13,$f0  # f13 = f1 + f0 * f13 (muladd if n is odd)
 4c: f46d0000  sdc1     $f13,0(v1)       # y[n-1] = f13 (store odd result)
Done:
 50: 03e00008  jr       ra               # return
 54: 00000000  nop                       # (delay slot)
```

vl setvl



vmovsd xmm1

vmovddup xmm1

vinsertf128 xmm1 ymm2

vmovsd vfmadd213sd vmovsd

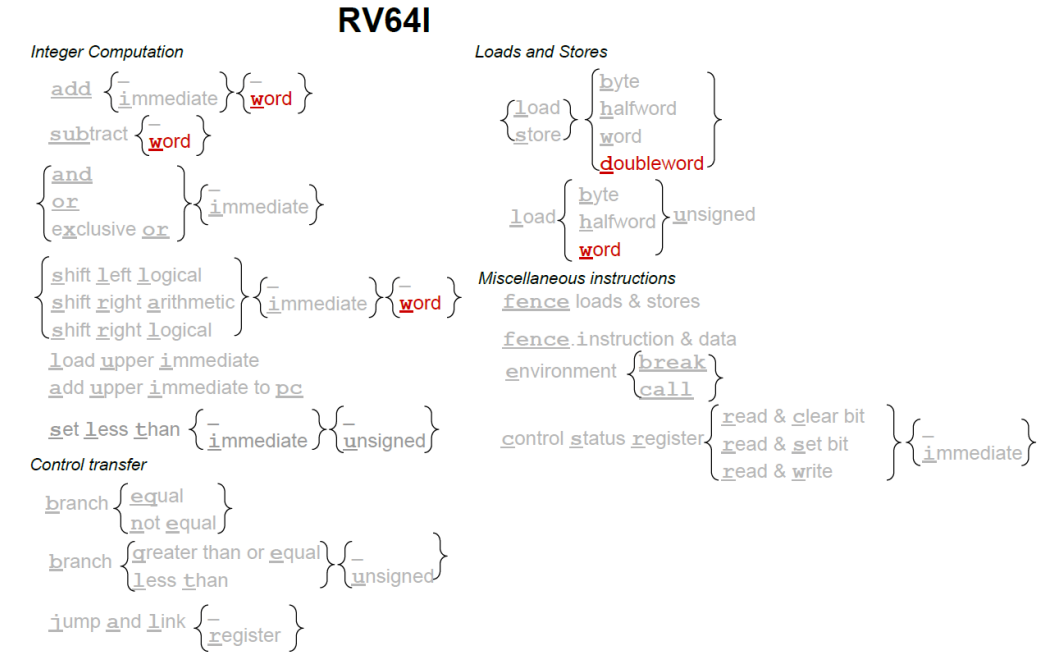
vl setvl

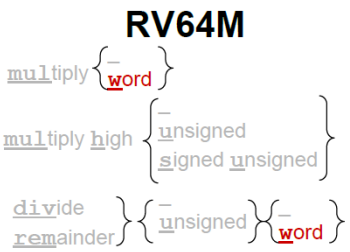
RV64 64

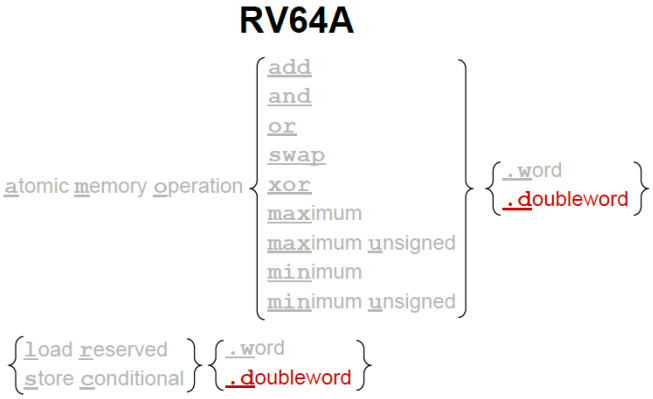
C. Gordon Bell, 1976

9.1 9.4 RV32G 64 RV64G
64 ISA ISA 32 (word)
(doubleword) (long) PC 64
RV64I sub 64 RV32I 32 RV64 RV32
; RV32
9.8 RV64I 2 27 2.8 RV32I

4 4 8 8 9.5 9.1
9.4 RV64GC
RV64I 64 64 32
RV64I RV32I
64 RV64I addw addiw subw
32 RV64I
sllw slliw srlw srliw sraw sraiw 32 64
64 RV64 ld sd
RV32I RV64I
lwu
RV64 mulw divw divuw
remw remuw RV64A 11


RV64I
Integer Computation
add { _ immediate } { _ word }
subtract { _ word }
{ and / or / exclusive or } { _ immediate }
{ shift left logical / shift right arithmetic / shift right logical } { _ immediate } { _ word }
load upper immediate
add upper immediate to pc
set less than { _ immediate } { _ unsigned }
Control transfer
branch { equal / not equal }
branch { greater than or equal / less than } { _ unsigned }
jump and link { _ register }
Loads and Stores
{ load / store } { byte / halfword / word / doubleword }
load { byte / halfword / word } unsigned
Miscellaneous instructions
fence loads & stores
fence instruction & data
environment { break / call }
control status register { read & clear bit / read & set bit / read & write } { _ immediate }


RV64M
multiply { _ word }
multiply high { _ unsigned / signed unsigned }
divide / remainder { _ unsigned } { _ word }


RV64A
atomic memory operation { add / and / or / swap / xor / maximum / maximum unsigned / minimum / minimum unsigned } { .word / .doubleword }
{ load reserved / store conditional } { .word / .doubleword }

RV64F and RV64D

Floating-Point Computation

float { add | subtract | multiply | divide | square root | minimum | maximum } { .single | .double }

float { – | negative } multiply { add | subtract } { .single | .double }

float move to { .single | .double } from .x register

float move to .x register from { .single | .double }

Comparison

compare float { equals | less than | less than or equals } { .single | .double }

Load and Store

float { load | store } { word | doubleword }

Conversion

float convert to { .single | .double } from { .word | .long } { – | unsigned }

float convert to { .word | .long } { – | unsigned } from { .single | .double }

float convert to .single from .double

float convert to .double from .single

Other instructions

float sign injection { – | negative | exclusive or } { .single | .double }

float classify { .single | .double }

RV64C

Integer Computation

c.add { – | immediate } { – | word }

c.add immediate * 16 to stack pointer

c.add immediate * 4 to stack pointer nondestructive

c.subtract { – | word }

c. { shift left logical | shift right arithmetic | shift right logical } immediate

c.and { – | immediate }

c.or

c.move

c.exclusive or

c.load { – | upper } immediate

Loads and Stores

c. { – | ~~float~~ } { load | store } { ~~word~~ | doubleword } { – | using stack pointer }

c.float { load | store } doubleword { – | using stack pointer }

Control transfer

c.branch { equal | not equal } to zero

c.jump { – | ~~and link~~ }

c.jump { – | and link } register

Other instructions

c.environment break

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	110	rd	0000011	I lwu
imm[11:0]			rs1	011	rd	0000011	I ld
imm[11:5]		rs2	rs1	011	imm[4:0]	0100011	S sd
0000000		shamt	rs1	001	rd	0010011	I slli
0000000		shamt	rs1	101	rd	0010011	I srli
0100000		shamt	rs1	101	rd	0010011	I srai
imm[11:0]			rs1	000	rd	0011011	I addiw
0000000		shamt	rs1	001	rd	0011011	I slliw
0000000		shamt	rs1	101	rd	0011011	I srliw
0100000		shamt	rs1	101	rd	0011011	I sraiw
0000000		rs2	rs1	000	rd	0111011	R addw
0100000		rs2	rs1	000	rd	0111011	R subw
0000000		rs2	rs1	001	rd	0111011	R sllw
0000000		rs2	rs1	101	rd	0111011	R srlw
0100000		rs2	rs1	101	rd	0111011	R sraw

0000001	rs2	rs1	000	rd	0111011	R mulw
0000001	rs2	rs1	100	rd	0111011	R divw
0000001	rs2	rs1	101	rd	0111011	R divuw
0000001	rs2	rs1	110	rd	0111011	R remw
0000001	rs2	rs1	111	rd	0111011	R remuw

00010	aq	rl	00000	rs1	011	rd	0101111	R lr.d
00011	aq	rl	rs2	rs1	011	rd	0101111	R sc.d
00001	aq	rl	rs2	rs1	011	rd	0101111	R amoswap..d
00000	aq	rl	rs2	rs1	011	rd	0101111	R amoadd.d
00100	aq	rl	rs2	rs1	011	rd	0101111	R amoxor.d
01100	aq	rl	rs2	rs1	011	rd	0101111	R amoand.d
01000	aq	rl	rs2	rs1	011	rd	0101111	R amoor.d
10000	aq	rl	rs2	rs1	011	rd	0101111	R amomin.d
10100	aq	rl	rs2	rs1	011	rd	0101111	R amomax.d
11000	aq	rl	rs2	rs1	011	rd	0101111	R amominu.d
11100	aq	rl	rs2	rs1	011	rd	0101111	R amomaxu.d

1100000	00010	rs1	rm	rd	1010011	R fcvt.l.s
1100000	00011	rs1	rm	rd	1010011	R fcvt.lu.s
1101000	00010	rs1	rm	rd	1010011	R fcvt.s.l
1101000	00011	rs1	rm	rd	1010011	R fcvt.s.lu

1100001	00010	rs1	rm	rd	1010011	R fcvt.l.d
1100001	00011	rs1	rm	rd	1010011	R fcvt.lu.d
1110001	00000	rs1	000	rd	1010011	R fmv.x.d
1101001	00010	rs1	rm	rd	1010011	R fcvt.d.l
1101001	00011	rs1	rm	rd	1010011	R fcvt.d.lu
1111001	00000	rs1	000	rd	1010011	R fmv.d.x

RV64F RV64D
fcvt.l.s fcvt.l.d fcvt.lu.s fcvt.lu.d fcvt.s.l fcvt.s.lu fcvt.d.l fcvt.d.lu.
x 64 RV64D
fmv.x.w fmv.w.x.
RV64 RV32 RV64C
RV32C 64 RV64C
c.jal c.lw c.sw c.lwsp c.swsp
c.flw c.fsw c.flwsp c.fswsp RV64C
c.addw c.addiw c.subw c.ld c.sd c.ldsp c.sdsp

RV64 ABI lp64 lp64f lp64d
lp64 C 64 ; 32 RV32 3
f d

RV64V
RV32V 75 8.2
X64 X64U RV64V RV32V

Gordon Bell
32
64 [Mashey 2009]
MIPS 1991 32 64
64 MIPS-32 MIPS-64 d” daddu
dsll 9.10 MIPS-32 MIPS-64
MIPS-64 MIPS-32
x86-32 64
x86-64

- 8 16 r8-r15 ;
- SIMD 8 16 xmm8-xmm15 PC
- PC

x86-32

x86-32 2 30 2.11 x86-64 9.11
x86-64 64 ISA
x86-32 20
15 64 32 45

46 x86-64

ISA	ARM-64	MIPS-64	x86-64	RV64I	RV64I+RV64C
Instructions	16	24	15	19	19
Bytes	64	96	46	76	52

x86-32 x86-64
ARM x86-64 ISA
64 ISA ISA
ARM-32 ISA

- 15 31;
- PC;
- r31 ;
- ARM-32 ARM-64 ;
- ARM-64 ARM-32
- ARM-64 ARM-32

ARM-32 ARM-64

32 ARM-64 Thumb-2 ISA Thumb-2 32

RISC-V ARM ISA ISA ARM-32
1000 ARM-64 3185 [ARM 2015]
ARM-64
9.9 ARM-64 RV64I x86-64
ARM-32 31
PC ARM-64
9.6 ISA 9.8 9.11 RV64I
ARM-64 MIPS-64 x86-64 2
RV32I RV64I
MIPS-64 nop
RV64I
RV64I ARM-64 x86-64
RV64I
RV64I + RV64C

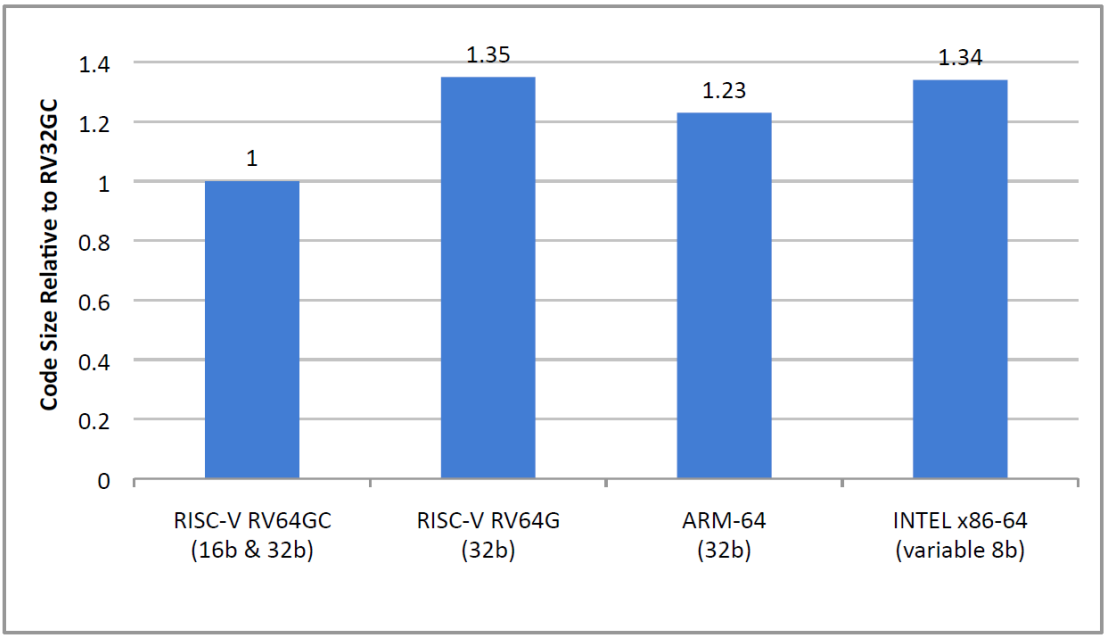
x86-64 ISA 64

Itanium ISA x86-32 x86-32

Itanium AMD AMD64 64 x86-32 Itanium

AMD64 ISA x86-32 64 x86-64 [Kerner & Padgett 2007]

ARM-64 MIPS-64 x86-64
ARM-64 ARMv8 MIPS-64 MIPS-IV x86-64 AMD64 x86-64



9.7 RV64 ARM-64 x86-64 1 9 1.5 RV32GC RV64GC ; RV64GC 1 RV32I RV64I ARM-64 ARM-32 8 64 Thumb-2 32 ARM-64 ARMThumb-2 25 x86-64 x86-32 7 RV64GC ARM-64 RV64GC 23 x86-64 RV64GC 34 RV64

Seymour Cray 1976

ARM-32

Thumb-2 32 MIPS-64 x86-64
ISA x86-64 ISA MIPS-
64 ARM-64 ISA

RISC-V 32 64 ISA
RISC-V 32 64 ; RV64I
ISA RV32I
RV32GCV RV64GCV 64
32 RV64I RV64C
64 ISA
64 RISC-V 64 ISA
20

MIPS
Imagination Technologies
2012 1
MIPS ISA
;

RV128
RV128 RISC-V 128 ISA
2^{64}
DRAM
[2014] 128 RISC-V
128 ISA RV128G [Waterman and Asanovic 2017] 9.1 9.4
RV32 RV64 128
RV128 128 Q quadword
64 D double word

I. ARM. Armv8-a architecture reference manual. 2015.

M. Kerner and N. Padgett. A history of modern 64-bit computing. Technical report, CS Department, University of Washington, Feb 2007. URL http://courses.cs.washington.edu/courses/csep590/06au/projects/history-64-bit.pdf.

J. Mashey. The long road to 64 bits. *Communications of the ACM*, 52(1):45–53, 2009.

A. Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL https://riscv.org/specifications/

J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 457–468. IEEE, 2014.

```
# RV64I (19 instructions, 76 bytes, or 52 bytes with RV64C)
# a1 is n, a3 points to a[0], a4 is i, a5 is j, a6 is x
   0: 00850693  addi  a3,a0,8    # (8 vs 4) a3 is pointer to a[i]
   4: 00100713  li    a4,1       # i = 1
Outer Loop:
   8: 00b76463  bltu  a4,a1,10   # if i < n, jump to Continue Outer loop
Exit Outer Loop:
   c: 00008067  ret              # return from function
Continue Outer Loop:
  10: 0006b803  ld    a6,0(a3)   # (ld vs lw) x = a[i]
  14: 00068613  mv    a2,a3      # a2 is pointer to a[j]
  18: 00070793  mv    a5,a4      # j = i
Inner Loop:
  1c: ff863883  ld    a7,-8(a2)  # (ld vs lw, 8 vs 4) a7 = a[j-1]
  20: 01185a63  ble   a7,a6,34   # if a[j-1] <= a[i], jump to Exit Inner Loop
  24: 01163023  sd    a7,0(a2)   # (sd vs sw) a[j] = a[j-1]
  28: fff78793  addi  a5,a5,-1   # j--
  2c: ff860613  addi  a2,a2,-8   # (8 vs 4) decrement a2 to point to a[j]
  30: fe0796e3  bnez  a5,1c      # if j != 0, jump to Inner Loop
Exit Inner Loop:
  34: 00379793  slli  a5,a5,0x3  # (8 vs 4) multiply a5 by 8
  38: 00f507b3  add   a5,a0,a5   # a5 is now byte address oi a[j]
  3c: 0107b023  sd    a6,0(a5)   # (sd vs sw) a[j] = x
  40: 00170713  addi  a4,a4,1    # i++
  44: 00868693  addi  a3,a3,8    # increment a3 to point to a[i]
  48: fc1ff06f  j     8          # jump to Outer Loop  # continue outer loop
```

```
# ARM-64 (16 instructions, 64 bytes)
# x0 points to a[0], x1 is n, x2 is j, x3 is i, x4 is x
  0: d2800023  mov  x3, #0x1            # i = 1
Outer Loop:
  4: eb01007f  cmp  x3, x1              # compare i vs n
  8: 54000043  b.cc 10                  # if i < n, jump to Continue Outer loop
Exit Outer Loop:
  c: d65f03c0  ret                      # return from function
Continue Outer Loop:
 10: f8637804  ldr  x4, [x0, x3, lsl #3] # (x4 ca r4) vs x = a[i]
 14: aa0303e2  mov  x2, x3              # (x2 vs r2) j = i
Inner Loop:
 18: 8b020c05  add  x5, x0, x2, lsl #3   # x5 is pointer to a[j]
 1c: f85f80a5  ldur x5, [x5, #-8]       # x5 = a[j]
 20: eb0400bf  cmp  x5, x4              # compare a[j-1] vs. x
 24: 5400008d  b.le 34                  # if a[j-1]<=a[i], jump to Exit Inner Loop

 28: f8227805  str  x5, [x0, x2, lsl #3] # a[j] = a[j-1]
 2c: f1000442  subs x2, x2, #0x1        # j--
 30: 54ffff41  b.ne 18                  # if j != 0, jump to Inner Loop
Exit Inner Loop:
 34: f8227804  str  x4, [x0, x2, lsl #3] # a[j] = x
 38: 91000463  add  x3, x3, #0x1        # i++
 3c: 17fffff2  b    4                   # jump to Outer Loop
```

```
# MIPS-64 (24 instructions, 96 bytes)
# a1 is n, a3 is pointer to a[0], v0 is j, v1 is i, t0 is x
  0: 64860008 daddiu a2,a0,8   # (daddiu vs addiu, 8 vs 4) a2 is pointer to a[i]
  4: 24030001 li     v1,1      # i = 1
Outer Loop:
  8: 0065102b sltu   v0,v1,a1  # set on i < n
  c: 14400003 bnez   v0,1c     # if i < n, jump to Continue Outer Loop
 10: 00c03825 move   a3,a2     # a3 is pointer to a[j] (slot filled)
 14: 03e00008 jr     ra        # return from function
 18: 00000000 nop              # branch delay slot unfilled
Continue Outer Loop:
 1c: dcc80000 ld     a4,0(a2)  # (ld vs lw) x = a[i]
 20: 00601025 move   v0,v1     # j = i
Inner Loop:
 24: dce9fff8 ld     a5,-8(a3) # (ld vs lw, 8 vs. 4, a5 vs t1) a5 = a[j-1]
 28: 0109502a slt    a6,a4,a5  # (no load delay slot) set a[i] < a[j-1]
 2c: 11400005 beqz   a6,44     # if a[j-1] <= a[i], jump to Exit Inner Loop
 30: 00000000 nop              # branch delay slot unfilled
 34: 6442ffff daddiu v0,v0,-1  # (daddiu vs addiu) j--
 38: fce90000 sd     a5,0(a3)  # (sd vs sw, a5 vs t1) a[j] = a[j-1]
 3c: 1440fff9 bnez   v0,24     # if j != 0, jump to Inner Loop (next slot filled)
 40: 64e7fff8 daddiu a3,a3,-8  # (daddiu vs addiu, 8 vs 4) decr a2 pointer to a[j]
Exit Inner Loop:
 44: 000210f8 dsll   v0,v0,0x3 # (dsll vs sll)
 48: 0082102d daddu  v0,a0,v0  # (daddu vs addu) v0 now byte address oi a[j]
 4c: fc480000 sd     a4,0(v0)  # (sd vs sw) a[j] = x
 50: 64630001 daddiu v1,v1,1   # (daddiu vs addiu) i++
 54: 1000ffec b      8         # jump to Outer Loop (next delay slot filled)
 58: 64c60008 daddiu a2,a2,8   # (daddiu vs addiu, 8 vs 4) incr a2 pointer to a[i]
 5c: 00000000 nop              # Unncessary(?)
```

```
# x86-64 (15.xx-new-std-range-for-loops) -O3 -std=c++20
        # rax is j, rcx is x, rdx is i, rsi is n, rdi is pointer to a[0]
          0: ba 01 00 00 00 mov edx,0x1
        Outer Loop:
          5: 48 39 f2       cmp rdx,rsi             # compare i vs. n
Loop      8: 73 23          jae 2d <Exit Loop>      # if i >= n, jump to Exit Outer
          a: 48 8b 0c d7    mov rcx,[rdi+rdx*8]     # x = a[i]
          e: 48 89 d0       mov rax,rdx             # j = i
        Inner Loop:
         11: 4c 8b 44 c7 f8 mov r8,[rdi+rax*8-0x8] # r8 = a[j-1]
         16: 49 39 c8       cmp r8,rcx              # compare a[j-1] vs. x
nnerLoop 19: 7e 09          jle 24 <Exit Loop>      # if a[j-1]<=a[i],jump to Exit I
         1b: 4c 89 04 c7    mov [rdi+rax*8],r8      # a[j] = a[j-1]
         1f: 48 ff c8       dec rax                 # j--
         22: 75 ed          jne 11 <Inner Loop>     # if j != 0, jump to Inner Loop
        Exit InnerLoop:
         24: 48 89 0c c7    mov [rdi+rax*8],rcx     # a[j] = x
         28: 48 ff c2       inc rdx                 # i++
         2b: eb d8          jmp 5 <Outer Loop>      # jump to Outer Loop
        Exit Outer Loop:
         2d: c3             ret                     # return from function
```

RV32/64

Edsger W. Dijkstra

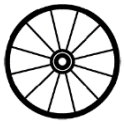
RISC-V

machine mode Linux FreeBSD Windows
supervisor mode

I/O

runtime

RISC-V



10.1 RISC-V 10.2
CSR

RV32 RV64
XLEN
RV32 XLEN 32 RV64 XLEN 64

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
0001000		00010	00000	000	00000	1110011	R sret
0011000		00010	00000	000	00000	1110011	R mret
0001000		00101	00000	000	00000	1110011	R wfi
0001001		rs2	rs1	000	00000	1110011	R sfence.vma

RV32/64 Privileged Instructions

machine-mode
supervisor-mode
} trap return

supervisor-mode fence virtual memory address

wait for interrupt

M M-mode RISC-V *hart* hardware thread
M hart I/O
RISC-V
RISC-V M
RISC-V

RISC-V
10.3
M

- ROM
- ebreak
- ecall
-
- 0x12

amoadd.w

Interrupt / Exception mcause[XLEN-1]	Exception Code mcause[XLEN-2:0]	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

mcause

hart
(hardware thread)
harts
hart

C
JAL
JALR
C
4 2

load store 10.3
load store

load store

load store load store

load store RISC-
V load store load store

hart hart
hart mtimecmp
mtime

RISC-V

CSR

- mtvec Machine Trap Vector
- mepc Machine Exception PC
- mcause Machine Exception Cause
- mie Machine Interrupt Enable
- mip Machine Interrupt Pending
- mtval Machine Trap Value trap
  0
- mscratch Machine Scratch
- mstatus Machine Status
  10.4

SD	0	TSR	TW	TVM	MXR	SUM	MPRV	
1	XLEN-24	1	1	1	1	1	1	

16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
XS	FS	MPP	0	SPP	MPIE	0	SPIE	UPIE	MIE	0	SIE	UIE
2	2	2	2	1	1	1	1	1	1	1	1	1

mstatus

M mstatus.MIE 1
. mie mie

10.3 mie[7] M
mip

Encoding	Name	Abbreviation
00	User	U
01	Supervisor	S
11	Machine	M

mstatus.MIE = 1 mie[7] = 1 mip[7] = 1
hart

- PC mepc PC mtvec mepc
- mcause 10.3 mtval
- mstatus MIE MIE
  MPIE
- mstatus MPP
  M 10.5 MPP M

RISC-V
mcause
mtval [o]
1 ;
x
PC mtval-1 +
4x
mtval

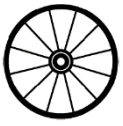
mscratch
a0 mscratch

mscratch a0
mret M
mret PC mepc mstatus MPIE
MIE mstatus MPP

10.6 RISC-V

RISC-V

mepc
mcause mtval mstatus



mret M wfi Wait For
Interrupt wfi

r n r nu RISC-V

nop
wfi

```
# save registers
csrrw a0, mscratch, a0  # save a0; set a0 = &temp storage
sw a1, 0(a0)            # save a1
sw a2, 4(a0)            # save a2
sw a3, 8(a0)            # save a3
sw a4, 12(a0)           # save a4

# decode interrupt cause
csrr a1, mcause         # read exception cause
bgez a1, exception      # branch if not an interrupt
andi a1, a1, 0x3f       # isolate interrupt cause
li a2, 7                # a2 = timer interrupt cause
bne a1, a2, otherInt    # branch if not a timer interrupt

# handle timer interrupt by incrementing time comparator
la a1, mtimecmp         # a1 = &time comparator
lw a2, 0(a1)            # load lower 32 bits of comparator
lw a3, 4(a1)            # load upper 32 bits of comparator
addi a4, a2, 1000       # increment lower bits by 1000 cycles
sltu a2, a4, a2         # generate carry-out
add a3, a3, a2          # increment upper bits
sw a3, 4(a1)            # store upper 32 bits
sw a4, 0(a1)            # store lower 32 bits

# restore registers and return
lw a4, 12(a0)           # restore a4
lw a3, 4(a0)            # restore a3
lw a2, 4(a0)            # restore a2
lw a1, 0(a0)            # restore a1
csrrw a0, mscratch, a0  # restore a0; mscratch = &temp storage
mret                    # return from handler
```

mstatus

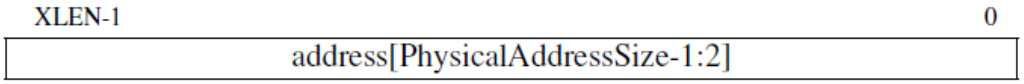
mie[7] mtvec mscratch

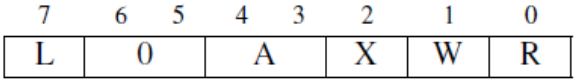
a0

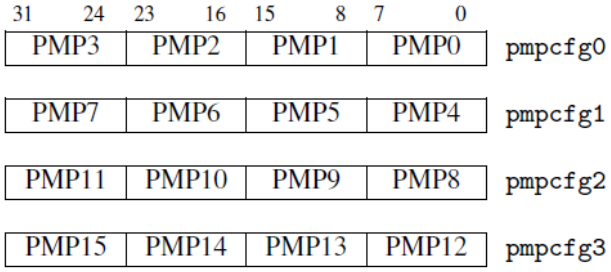
mscratch a1 a4 mcause mcause

mcause

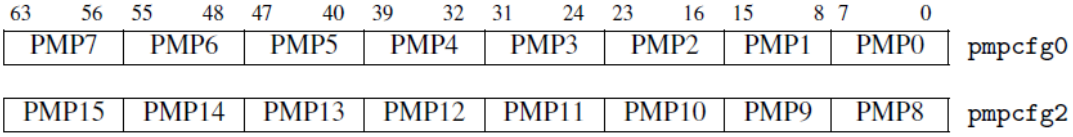
a0 a4 mscratch mret







RV32



RV64

pmpcfg

---

wfi

mstatus.MIE = 1 wfi

wfi

wfi

mip

M

RISC-V

mret

mstatus

U M

CSR U M

mstatus.MPP U 10.5 0 mret

M U U M

M U

PMP Physical Memory Protection M

U PMP 8 16

U

load store PMP

PMP i PMP i+1 PMP i+1

10.7 PMP CSR

pmpaddro pmpaddrN N+1 PMP

PMP CSR 10.8

PMP R W X load store fetch

A 0 PMP 1 PMP

[Waterman and Asanovic 2017]

PMP

PMP

PMP

RISC-V

S

Unix Linux FreeBSD Windows S U

M U S M CSR

PMP S S

S

M S

M

S

M Unix S

M S

RISC-V

S M

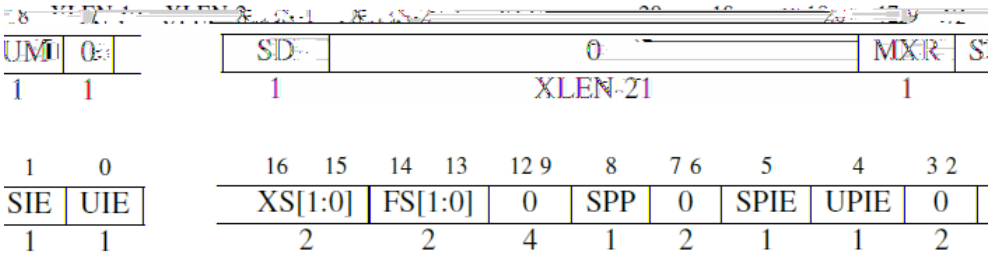
mideleg Machine Interrupt Delegation CSR S

mip mie mideleg 10.3

mideleg[5] S S S

M

S S sie Supervisor Interrupt



sstatus sstatus mstatus

mstatus

Enable sip Supervisor Interrupt Pending CSR
S mie mip M
sie sip mideleg

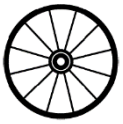
M medeleg CSR S
medeleg 10.3
medeleg[15] store page fault store S

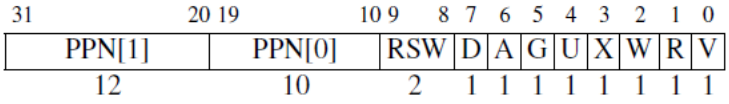
M M S
M S U
S CSR sepc stvec scause sscratch stval sstatus
10.2 M CSR 10.9 sstatus
sret mret S CSR
M CSR
S M hart
S S M
CSR

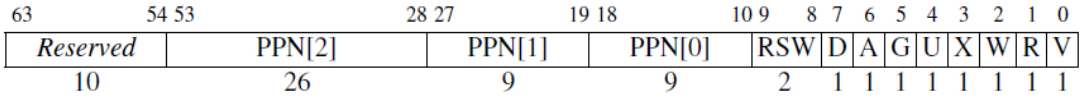
- PC sepc PC stvec
- scause 10.3 stval
- sstatus CSR SIE SIE SPIE
- sstatus SPP S

S
load store
PC

S
ecall M
(Supervisor Binary Interface)







IBM 360 67
4 KiB

Atlas
3 KiB
6

page fault exception

RISC-V SvX , X
RV32 Sv32 4GiB 2^{10} 4 MiB
2^{10} 4 KiB Sv32
2^{10}
4 KiB

10.10 Sv32 page-table entry PTE

- V V = 1 V = 0
- R W X 0
- U U = 0 U S
  U = 1 U S
- G

A
D

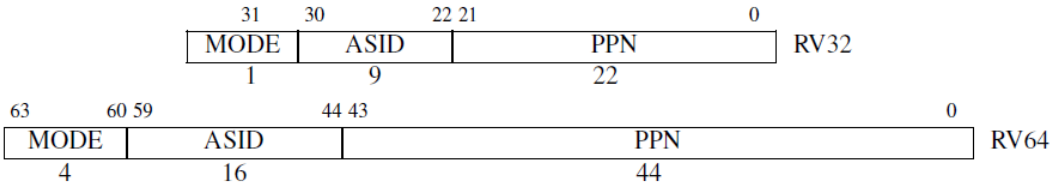
A OS
D

- A A
- D D
- RSW
- PPN
  PPN PPN
  10.10 PPN

RV64
Sv48 Sv39
2^9

RV64 Sv39 Sv39 Sv32
4 KiB 8
2^9 Sv39 512
GiB 2^9 1 GiB 2^9
Sv39 2 MiB Sv32 2^9 4 KiB

10.11 Sv39 Sv32 PPN
44 56 2^{26} GiB



satp

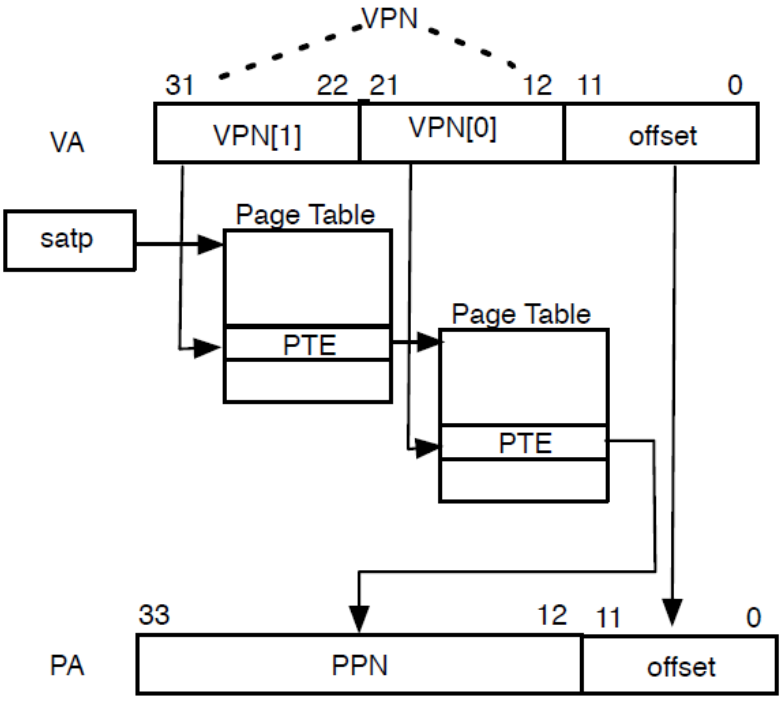
RV32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing.

RV64		
Value	Name	Description
0	Bare	No translation or protection.
8	Sv39	Page-based 39-bit virtual addressing.
9	Sv48	Page-based 48-bit virtual addressing.

satp

Sv39 RV64 35 Sv39
63-39 38 0000_0000_0000_0000$_{hex}$-
0000_003f_ffff_ffff$_{hex}$ ffff_ffc0_0000_0000$_{hex}$-ffff_ffff_ffff_ffff$_{hex}$
2^{25} 64
99.999997% 25
512 GiB
25

satp Supervisor Address Translation and Protection
S 10.12 satp MODE
10.13 ASID Address Space Identifier
PPN
4 KiB M S
satp S
satp



satp S U

10.14

1. satp.PPN VA[31:22]
   xf yu S F
2. PTE VA[21:12]
   S F
3. PPN 12
   f S F

Sv39 Sv32

PTE 10.19

RISC-V load

store

TLB Translation Lookaside Buffer

S

sfence.vma

rs1 rs2

ASID x0

XLEN-1	12	11	10	9	8	7	6	5	4	3	2	1	0
WIRI		MEIP	WIRI	SEIP	UEIP	MTIP	WIRI	STIP	UTIP	MSIP	WIRI	SSIP	USIP
WPRI		MEIE	WPRI	SEIE	UEIE	MTIE	WPRI	STIE	UTIE	MSIE	WPRI	SSIE	USIE
XLEN-12		1	1	1	1	1	1	1	1	1	1	1	1

mip

mie mip

XLEN-1	10	9	8	7	6	5	4	3	2	1	0
WIRI		SEIP	UEIP	WIRI		STIP	UTIP	WIRI		SSIP	USIP
WPRI		SEIE	UEIE	WPRI		STIE	UTIE	WPRI		SSIE	USIE
XLEN-10		1	1	2		1	1	2		1	1

sip

XLEN-1	2	1	0
BASE[XLEN-1:2]		MODE	
XLEN-2		2	

mtvec stvec

XLEN-1	XLEN-2	0
Interrupt	Exception Code	
1	XLEN-1	

cause mcause scause

---

sfence.vma hart hart hart
hart hart sfence.vma
TLB

Fred Brooks,Jr.,1986

Brooks IBM System/360
1964

RISC-V

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10*. May 2017. URL https://riscv.org/specifications/privileged-isa/.

http://parlab.eecs.berkeley.edu

1. Let a be `satp`.$ppn \times$ PAGESIZE, and let $i =$ LEVELS $- 1$.
2. Let pte be the value of the PTE at address $a + va.vpn[i] \times$ PTESIZE.
3. If $pte.v = 0$, or if $pte.r = 0$ and $pte.w = 1$, stop and raise a page-fault exception.
4. Otherwise, the PTE is valid. If $pte.r = 1$ or $pte.x = 1$, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let $i = i - 1$. If $i < 0$, stop and raise a page-fault exception. Otherwise, let $a = pte.ppn \times$ PAGESIZE and go to step 2.
5. A leaf PTE has been found. Determine if the requested memory access is allowed by the $pte.r$, $pte.w$, $pte.x$, and $pte.u$ bits, given the current privilege mode and the value of the SUM and MXR fields of the `mstatus` register. If not, stop and raise a page-fault exception.
6. If $i > 0$ and $pa.ppn[i - 1 : 0] \neq 0$, this is a misaligned superpage; stop and raise a page-fault exception.
7. If $pte.a = 0$, or if the memory access is a store and $pte.d = 0$, then either:
   - Raise a page-fault exception, or:
   - Set $pte.a$ to 1 and, if the memory access is a store, also set $pte.d$ to 1.
8. The translation is successful. The translated physical address is given as follows:
   - *pa.pgoff* $=$ *va.pgoff*.
   - If $i > 0$, then this is a superpage translation and $pa.ppn[i - 1 : 0] = va.vpn[i - 1 : 0]$.
   - $pa.ppn[\text{LEVELS} - 1 : i] = pte.ppn[\text{LEVELS} - 1 : i]$.

RISC-V

Alan Perlis

Alan Perlis, 1982

RISC-V 8

B insert, extract, and test bit fields
rotations funnel shifts bit and byte permutations
0 0 count leading and trailing zeros count bits set

16 RV32E
0-15 16-31 3.2

H

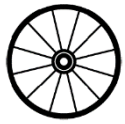
Java Javascript
ISA J Just-In-Time

L IEEE 754-2008
0.1 RV32L

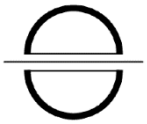
N

M U 10

Unix Unix



garbage collection barriers integer overflow floating-point traps



P

8 RVV

Q IEEE 754-2008 128

RV64IFD

Henry David Thoreau 19 1854

RISC-V

FPGA RISC-V

ISA x86-32

[1] 1.2

RISC-V RISC-V

ISA

A RISC-V

Coco Chanel 1923

RV32/64I RVM RVA RVF RVD RVC RVV

RISC-V [Waterman and Asanovic 2017]

amoadd.w adoand.d amoadn.w amomax.d amomax.w amomaxu.d amomaxu.w amomin.d amomin.w amominu.d

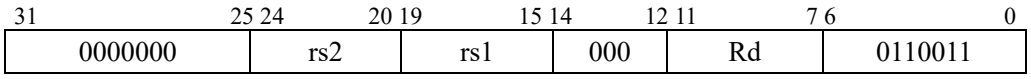
add rd, rs1, rs2

x[rd] = x[rs1] + x[rs2]

(Add). R-type, RV32I and RV64I.

x[*rs2*] x[*rs1*] x[*rd*]

rd, rs2; rd, rs2



---

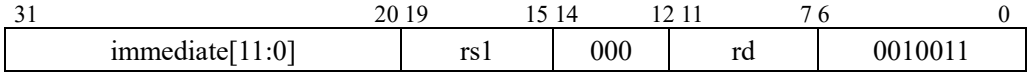
addi rd, rs1, immediate

x[rd] = x[rs1] + sext(immediate)

(Add Immediate). I-type, RV32I and RV64I.

x[*rs1*] x[*rd*]

rd, imm; rd, imm; imm; rd, imm



---

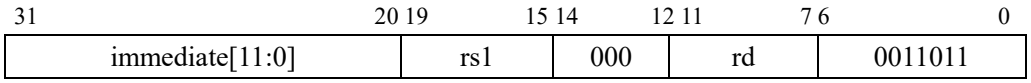
addiw rd, rs1, immediate

x[rd] = sext((x[rs1] + sext(immediate))[31:0])

(Add Word Immediate). I-type, RV64I.

x[*rs1*] 32 x[*rd*]

rd, imm



---

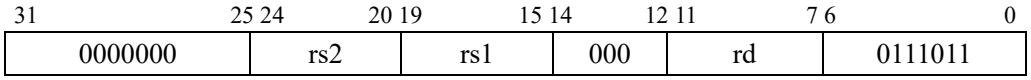
addw rd, rs1, rs2

x[rd] = sext((x[rs1] + x[rs2])[31:0])

(Add Word). R-type, RV64I.

x[*rs2*] x[*rs1*] 32 x[*rd*]

rd, rs2



---

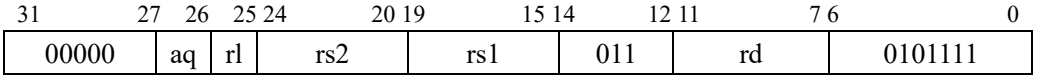
amoadd.d rd, rs2, (rs1)

x[rd] = AMO64(M[x[rs1]] + x[rs2])

(Atomic Memory Operation: Add Doubleword). R-type, RV64A.

x[*rs1*] *t* *t*+x[*rs2*]

x[*rd*] *t*



---

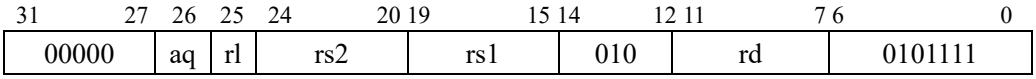
amoadd.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] + x[rs2])

(Atomic Memory Operation: Add Word). R-type, RV32A and RV64A.

x[*rs1*] *t* *t*+x[*rs2*] x[*rd*]

t



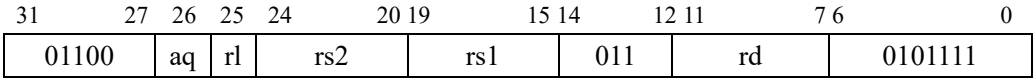
amoand.d rd, rs2, (rs1)

x[rd] = AMO64(M[x[rs1]] & x[rs2])

(Atomic Memory Operation: AND Doubleword). R-type, RV64A.

x[*rs1*] *t* t x[*rs2*]

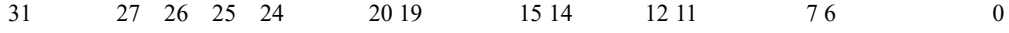
x[*rd*] *t*

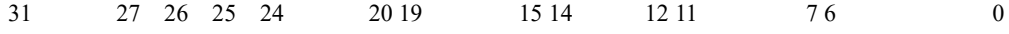


amoand.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] & x[rs2])

(Atomic Memory Operation: AND Word)

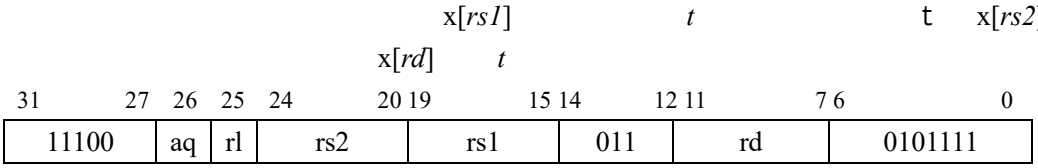




amomaxu.d rd, rs2, (rs1)

x[rd] = AMO64(M[x[rs1]] MAXU x[rs2])

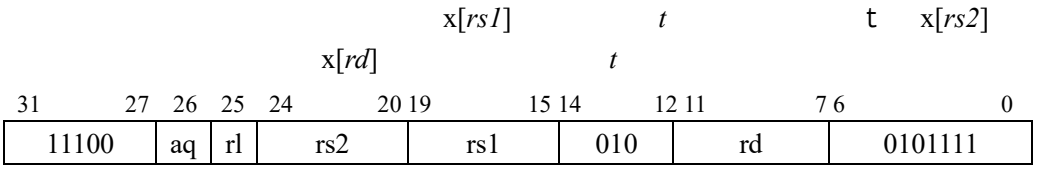
(Atomic Memory Operation: Maximum Doubleword, Unsigned). R-type, RV64A.



amomaxu.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] MAXU x[rs2])

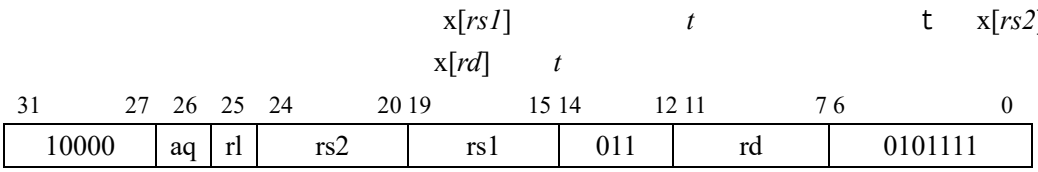
(Atomic Memory Operation: Maximum Word, Unsigned). R-type, RV32A and RV64A.



amomin.d rd, rs2, (rs1)

x[rd] = AMO64(M[x[rs1]] MIN x[rs2])

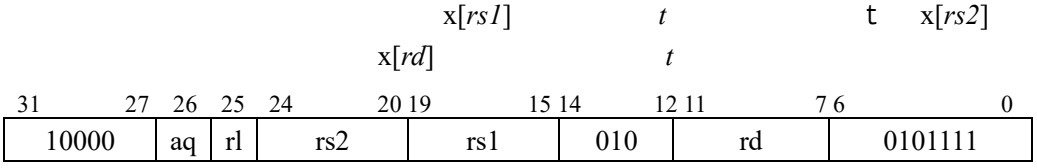
(Atomic Memory Operation: Minimum Doubleword). R-type, RV64A.



amomin.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] MIN x[rs2])

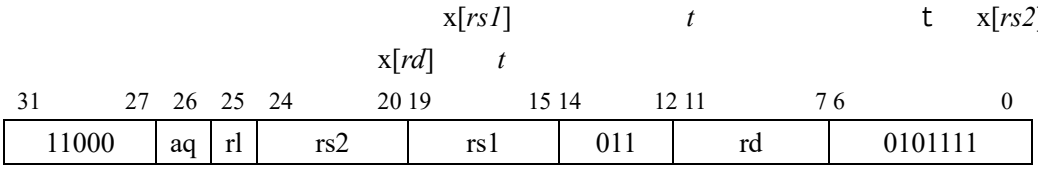
(Atomic Memory Operation: Minimum Word). R-type, RV32A and RV64A.



amominu.d rd, rs2,(rs1)

x[rd] = AMO64(M[x[rs1]] MINU x[rs2])

(Atomic Memory Operation: Minimum Doubleword, Unsigned). R-type, RV64A.



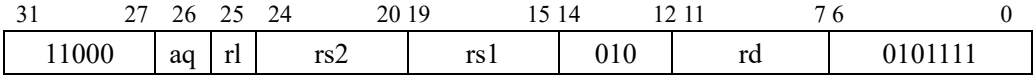
amominu.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] MINU x[rs2])

(Atomic Memory Operation: Minimum Word, Unsigned). R-type, RV32A and RV64A.

x[*rs1*] *t* t x[*rs2*]

x[*rd*] *t*



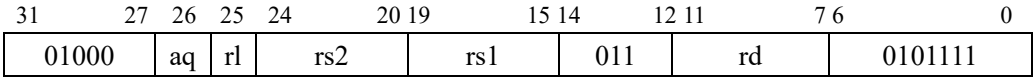
amoor.d rd, rs2, (rs1)

x[rd] = AMO64(M[x[rs1]] | x[rs2])

(Atomic Memory Operation: OR Doubleword). R-type, RV64A.

x[*rs1*] *t* t x[*rs2*]

x[*rd*] *t*

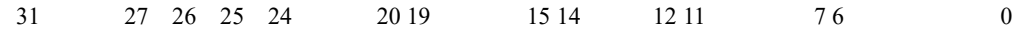


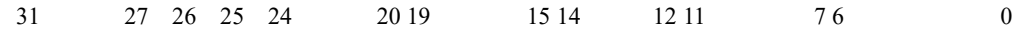
amoor.w rd, rs2, (rs1)

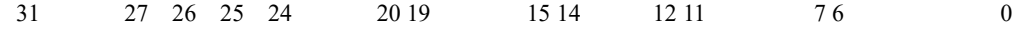
x[rd] = AMO32(M[x[rs1]] | x[rs2])

(Atomic Memory Operation: OR Word). R-type, RV32A and RV64A.

.







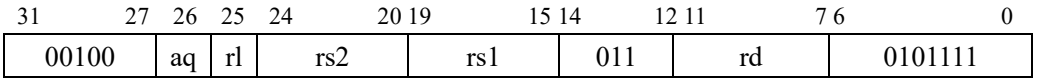
amoxor.d rd, rs2, (rs1)

x[rd] = AMO64(M[x[rs1]] ^ x[rs2])

(Atomic Memory Operation: XOR Doubleword). R-type, RV64A.

x[*rs1*] *t* t x[*rs2*]

x[*rd*] *t*



---

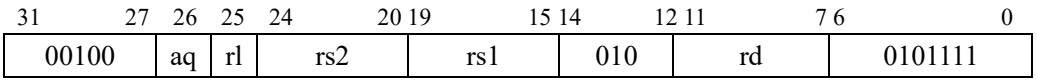
amoxor.w rd, rs2, (rs1)

x[rd] = AMO32(M[x[rs1]] ^ x[rs2])

(Atomic Memory Operation: XOR Word). R-type, RV32A and RV64A.

x[*rs1*] *t* t x[*rs2*]

x[*rd*] *t*



---

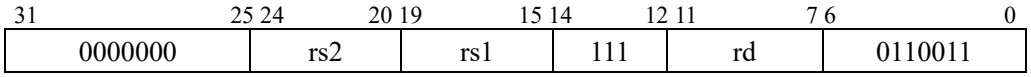
and rd, rs1, rs2

x[rd] = x[rs1] & x[rs2]

(And). R-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] x[*rd*]

rd, rs2



---

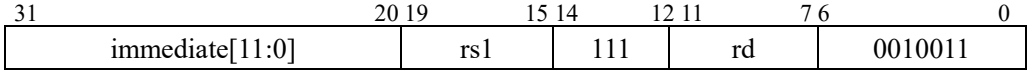
andi rd, rs1, immediate

x[rd] = x[rs1] & sext(immediate)

(And Immediate). I-type, RV32I and RV64I.

x[*rs1*] x[*rd*]

rd, imm



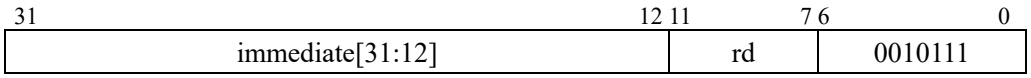
---

auipc rd, immediate

x[rd] = pc + sext(immediate[31:12] << 12)

PC *(Add Upper Immediate to PC)*. U-type, RV32I and RV64I.

20 12 *pc* x[*rd*]



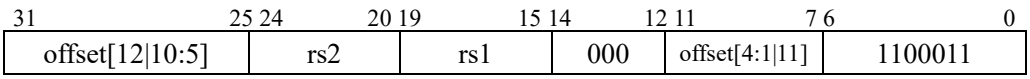
---

beq rs1, rs2, offset if (rs1 == rs2) pc += sext(offset)

(Branch if Equal). B-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] *pc* *offset*

rs1, offset



---

beqz rs1, offset if (rs1 == 0) pc += sext(offset)

(Branch if Equal to Zero). (Pesudoinstruction), RV32I and RV64I.

rs1, x0, offset.

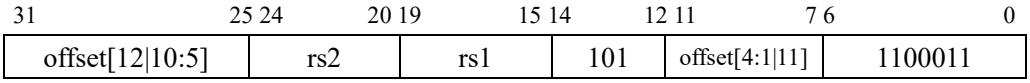
---

bge rs1, rs2, offset if (rs1 $_s$ rs2) pc += sext(offset)

(Branch if Greater Than or Equal). B-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] *pc*

offset



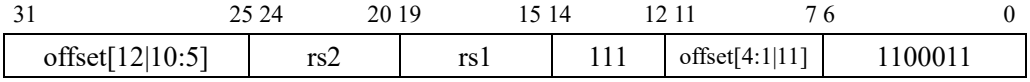
---

bgeu rs1, rs2, offset if (rs1 $_u$ rs2) pc += sext(offset)

(Branch if Greater Than or Equal, Unsigned). B-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] *pc*

offset



---

bgez rs1, offset if (rs1 $_s$ 0) pc += sext(offset)

(Branch if Greater Than or Equal to Zero). (Pesudoinstruction), RV32I and RV64I.

rs1, x0, offset.

---

bgt rs1, rs2, offset if (rs1 $>_s$ rs2) pc += sext(offset)

(Branch if Greater Than). (Pesudoinstruction), RV32I and RV64I.

rs2, rs1, offset.

---

bgtu rs1, rs2, offset if (rs1 $>_u$ rs2) pc += sext(offset)

(Branch if Greater Than, Unsigned). (Pesudoinstruction), RV32I and RV64I.

rs2, rs1, offset.

---

bgtz rs1, offset — if (rs2 >$_s$ 0) pc += sext(offset)

(Branch if Greater Than Zero). (Pesudoinstruction), RV32I and RV64I.

x0, rs2, offset.

---

ble rs1, rs2, offset — if (rs1 $_s$ rs2) pc += sext(offset)

(Branch if Less Than or Equal). (Pesudoinstruction), RV32I and RV64I.

rs2, rs1, offset.

---

bleu rs1, rs2, offset — if (rs1 $_u$ rs2) pc += sext(offset)

(Branch if Less Than or Equal, Unsigned). (Pesudoinstruction), RV32I and RV64I.

rs2, rs1, offset.

---

blez rs2, offset — if (rs2 $_s$ 0) pc += sext(offset)

(Branch if Less Than or Equal to Zero). (Pesudoinstruction), RV32I and RV64I.

x0, rs2, offset.

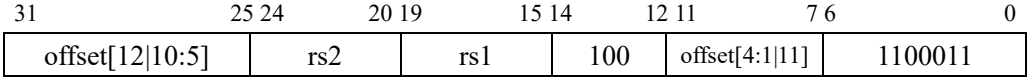
---

blt rs1, rs2, offset — if (rs1 <$_s$ rs2) pc += sext(offset)

(Branch if Less Than). B-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] *pc*

offset



---

bltz rs2, offset — if (rs1 <$_s$ 0) pc += sext(offset)

(Branch if Less Than Zero). (Pesudoinstruction), RV32I and RV64I.

rs1, x0, offset.

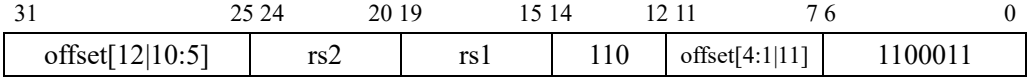
---

bltu rs1, rs2, offset — if (rs1 <$_u$ rs2) pc += sext(offset)

(Branch if Less Than, Unsigned). B-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] *pc*

offset



---

bne rs1, rs2, offset

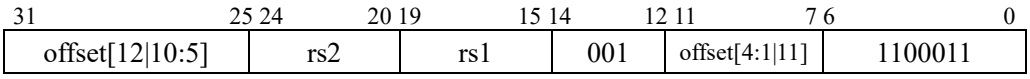
if (rs1 rs2) pc += sext(offset)

(Branch if Not Equal). B-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] *pc*

offset

rs1, offset



bnez rs1, offset

if (rs1 o) pc += sext(offset)

(Branch if Not Equal to Zero). (Pesudoinstruction), RV32I and RV64I.

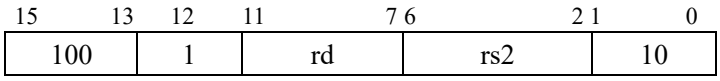
rs1, x0, offset.

c.add rd, rs2

x[rd] = x[rd] + x[rs2]

(Add). RV32IC and RV64IC.

rd, rd, rs2. rd=x0 rs2=x0

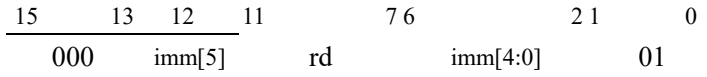


c.addi rd, imm

x[rd] = x[rd] + sext(imm)

(Add Immediate). RV32IC and RV64IC.

rd, rd, imm.

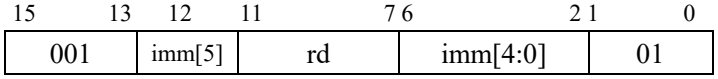


c.addiw rd, imm

x[rd] = sext((x[rd] + sext(imm))[31:0])

(Add Word Immediate). RV64IC.

rd, rd, imm. rd=x0



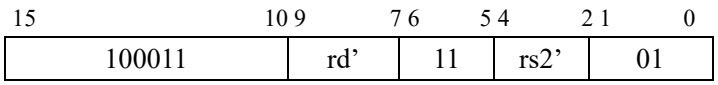
---

c.and rd', rs2'

x[8+rd'] = x[8+rd'] & x[8+rs2']

(AND). RV32IC and RV64IC.

rd, rd, rs2, rd=8+rd', rs2=8+rs2'.



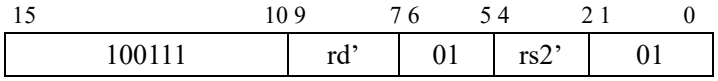
---

c.addw rd', rs2'

x[8+rd'] = sext((x[8+rd'] + x[8+rs2'])[31:0])

(Add Word). RV64IC.

rd, rd, rs2, rd=8+rd', rs2=8+rs2'.



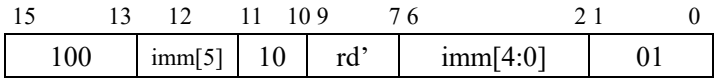
---

c.andi rd', imm

x[8+rd'] = x[8+rd'] & sext(imm)

(AND Immediate). RV32IC and RV64IC.

rd, rd, imm, rd=8+rd'.



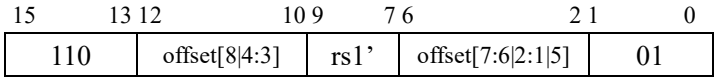
---

c.beqz rs1', offset

if (x[8+rs1'] == 0) pc += sext(offset)

(Branch if Equal to Zero). RV32IC and RV64IC.

rs1, x0, offset, rs1=8+rs1'.



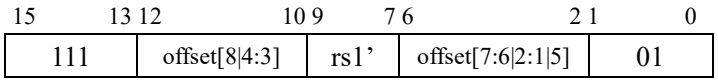
---

c.bnez rs1', offset

if (x[8+rs1'] 0) pc += sext(offset)

(Branch if Not Equal to Zero). RV32IC and RV64IC.

rs1, x0, offset, rs1=8+rs1'.



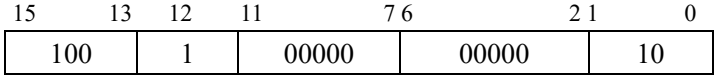
---

c.ebreak

RaiseException(Breakpoint)

(Environment Breakpoint). RV32IC and RV64IC.

.



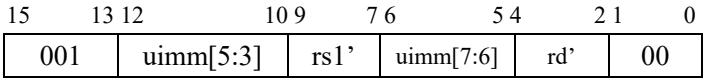
---

c.fld rd', uimm(rs1')

f[8+rd'] = M[x[8+rs1'] + uimm][63:0]

(Floating-point Load Doubleword). RV32DC and RV64DC.

rd, uimm(rs1), rd=8+rd', rs1=8+rs1'.



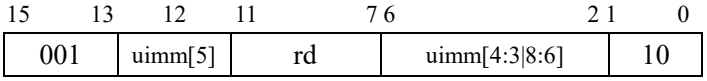
---

c.fldsp rd, uimm(x2)

f[rd] = M[x[2] + uimm][63:0]

(Floating-point Load Doubleword, Stack-Pointer Relative). RV32DC and RV64DC.

rd, uimm(x2).



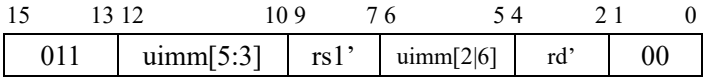
---

c.flw rd', uimm(rs1')

f[8+rd'] = M[x[8+rs1'] + uimm][31:0]

(Floating-point Load Word). RV32FC.

rd, uimm(rs1), rd=8+rd', rs1=8+rs1'.



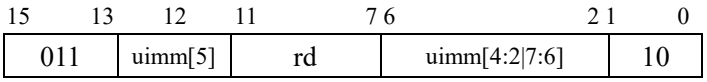
---

c.flwsp rd, uimm(x2)

f[rd] = M[x[2] + uimm][31:0]

(Floating-point Load Word, Stack-Pointer Relative). RV32FC.

rd, uimm(x2).



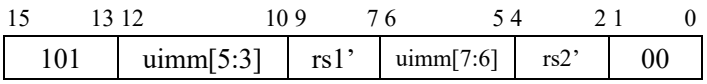
---

c.fsd rs2', uimm(rs1')

M[x[8+rs1'] + uimm][63:0] = f[8+rs2']

(Floating-point Store Doubleword). RV32DC and RV64DC.

rs2, uimm(rs1), rs2=8+rs2', rs1=8+rs1'.



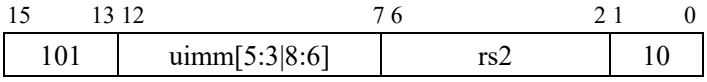
---

c.fsdsp rs2, uimm(x2)

M[x[2] + uimm][63:0] = f[rs2]

(Floating-point Store Doubleword, Stack-Pointer Relative). RV32DC and RV64DC.

rs2, uimm(x2).



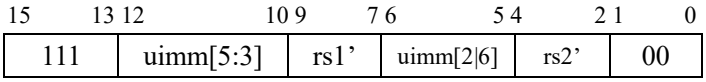
---

c.fsw rs2', uimm(rs1')

M[x[8+rs1'] + uimm][31:0] = f[8+rs2']

(Floating-point Store Word). RV32FC.

rs2, uimm(rs1), rs2=8+rs2', rs1=8+rs1'.



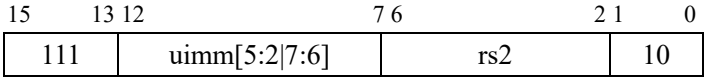
---

c.fswsp rs2, uimm(x2)

M[x[2] + uimm][31:0] = f[rs2]

(Floating-point Store Word, Stack-Pointer Relative). RV32FC.

rs2, uimm(x2).



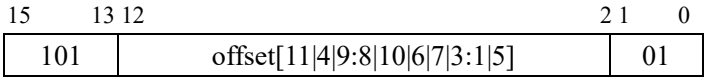
---

c.j offset

pc += sext(offset)

(Jump). RV32IC and RV64IC.

x0, offset.



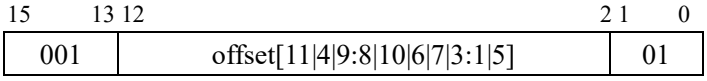
---

c.jal offset

x[1] = pc+2; pc += sext(offset)

(Jump and Link). RV32IC.

x1, offset.



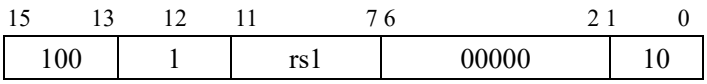
---

c.jalr rs1

t = pc+2; pc = x[rs1]; x[1] = t

(Jump and Link Register). RV32IC and RV64IC.

x1, 0(rs1). rs1=x0



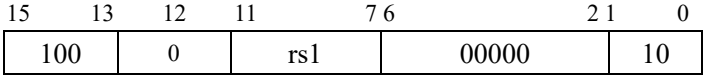
---

c.jr rs1

pc = x[rs1]

(Jump Register). RV32IC and RV64IC.

x0, 0(rs1). rs1=x0



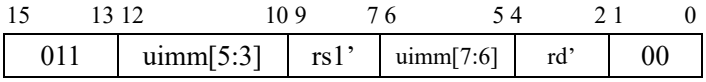
---

c.ld rd', uimm(rs1')

x[8+rd'] = M[x[8+rs1'] + uimm][63:0]

(Load Doubleword). RV64IC.

rd, uimm(rs1), rd=8+rd', rs1=8+rs1'.



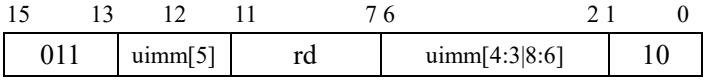
---

c.ldsp rd, uimm(x2)

x[rd] = M[x[2] + uimm][63:0]

(Load Doubleword, Stack-Pointer Relative). RV64IC.

rd, uimm(x2). rd=x0



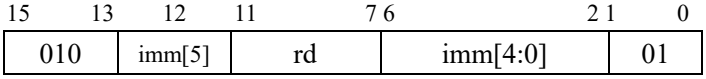
---

c.li rd, imm

x[rd] = sext(imm)

(Load Immediate). RV32IC and RV64IC.

rd, x0, imm.



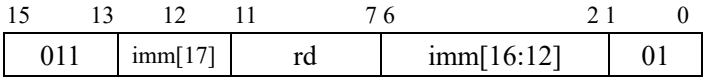
---

c.lui rd, imm

x[rd] = sext(imm[17:12] << 12)

(Load Upper Immediate). RV32IC and RV64IC.

rd, imm. rd=x2 imm=0



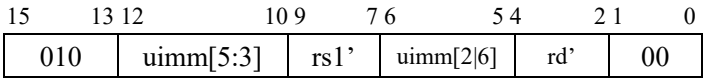
---

c.lw rd', uimm(rs1')

x[8+rd'] = sext(M[x[8+rs1'] + uimm][31:0])

(Load Word). RV32IC and RV64IC.

rd, uimm(rs1), rd=8+rd', rs1=8+rs1'.



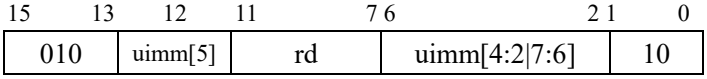
---

c.lwsp rd, uimm(x2)

x[rd] = sext(M[x[2] + uimm][31:0])

(Load Word, Stack-Pointer Relative). RV32IC and RV64IC.

rd, uimm(x2). rd=x0



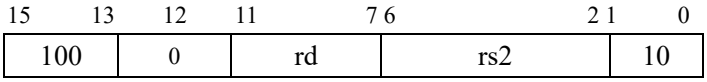
---

c.mv rd, rs2

x[rd] = x[rs2]

(Move). RV32IC and RV64IC.

rd, x0, rs2. rs2=x0



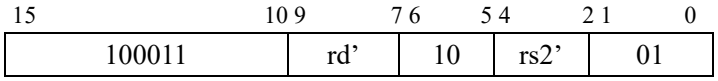
---

c.or rd', rs2'

x[8+rd'] = x[8+rd'] | x[8+rs2']

(OR). RV32IC and RV64IC.

rd, rd, rs2, rd=8+rd', rs2=8+rs2'.



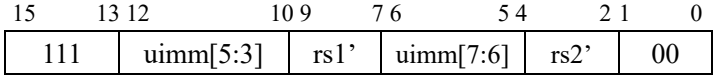
---

c.sd rs2', uimm(rs1')

M[x[8+rs1'] + uimm][63:0] = x[8+rs2']

(Store Doubleword). RV64IC.

rs2, uimm(rs1), rs2=8+rs2', rs1=8+rs1'.



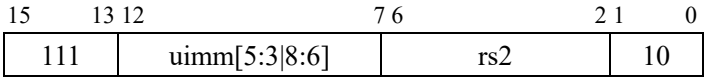
---

c.sdsp rs2, uimm(x2)

M[x[2] + uimm][63:0] = x[rs2]

(Store Doubleword, Stack-Pointer Relative). RV64IC.

rs2, uimm(x2).



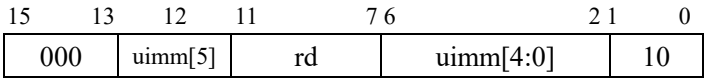
---

c.slli rd, uimm

x[rd] = x[rd] << uimm

(Shift Left Logical Immediate). RV32IC and RV64IC.

rd, rd, uimm.



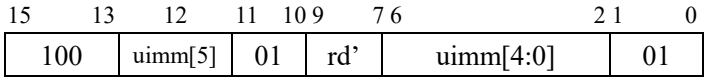
---

c.srai rd', uimm

x[8+rd'] = x[8+rd'] >>$_s$ uimm

(Shift Right Arithmetic Immediate). RV32IC and RV64IC.

rd, rd, uimm, rd=8+rd'.



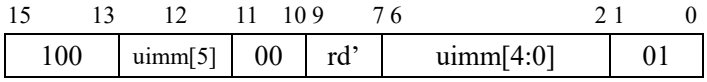
---

c.srli rd', uimm

x[8+rd'] = x[8+rd'] >>$_u$ uimm

(Shift Right Logical Immediate). RV32IC and RV64IC.

rd, rd, uimm, rd=8+rd'.



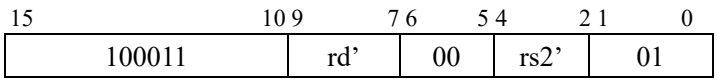
---

c.sub rd', rs2'

x[8+rd'] = x[8+rd'] - x[8+rs2']

(Subtract). RV32IC and RV64IC.

rd, rd, rs2. rd=8+rd', rs2=8+rs2'. .



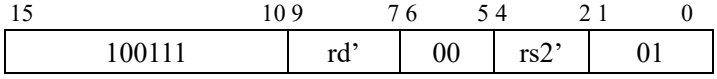
---

c.subw rd', rs2'

x[8+rd'] = sext((x[8+rd'] - x[8+rs2'])[31:0])

(Subtract Word). RV64IC.

rd, rd, rs2. rd=8+rd', rs2=8+rs2'. .



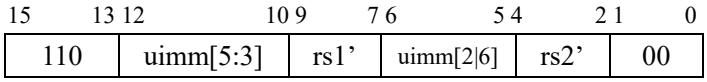
---

c.sw rs2', uimm(rs1')

M[x[8+rs1'] + uimm][31:0] = x[8+rs2']

(Store Word). RV32IC and RV64IC.

rs2, uimm(rs1), rs2=8+rs2', rs1=8+rs1'.



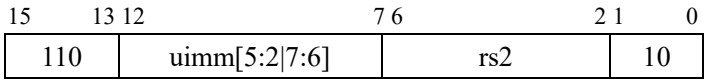
---

c.swsp rs2, uimm(x2)

M[x[2] + uimm][31:0] = x[rs2]

(Store Word, Stack-Pointer Relative). RV32IC and RV64IC.

rs2, uimm(x2).

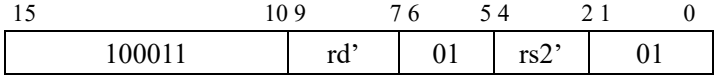


---

c.xor rd', rs2' x[8+rd'] = x[8+rd'] ^ x[8+rs2']

(Exclusive-OR). RV32IC and RV64IC.

rd, rd, rs2, rd=8+rd', rs2=8+rs2'.



---

call rd, symbol x[rd] = pc+8; pc = &symbol

(Call). (Pesudoinstruction), RV32I and RV64I.

pc+8 x[*rd*] *pc* *symbol* rd, offestHi, rd, offsetLo(rd). *rd* x1.

---

csrr rd, csr x[rd] = CSRs[csr]

(Control and Status Register Read). (Pesudoinstruction), RV32I and RV64I.

csr x[*rd*] rd, csr, x0.

---

csrc csr, rs1 CSRs[csr] &= ~x[rs1]

(Control and Status Register Clear). (Pesudoinstruction), RV32I and RV64I.

x[*rs1*] 1 *csr* x0, csr, rs1.

---

csrci csr, zimm[4:0] CSRs[csr] &= ~zimm

(Control and Status Register Clear Immediate). (Pesudoinstruction), RV32I and RV64I.

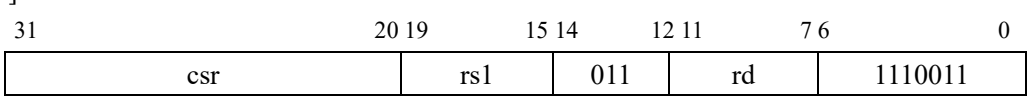
1 *csr* x0, csr, zimm.

---

csrrc rd, csr, rs1 t = CSRs[csr]; CSRs[csr] = t &~x[rs1]; x[rd] = t

(Control and Status Register Read and Clear). I-type, RV32I and RV64I.

csr *t* *t* x[*rs1*] *csr* *t* x[*rd*]

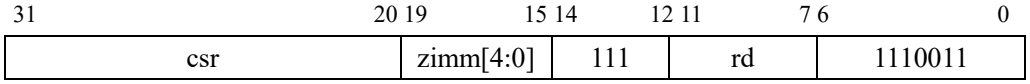


---

csrrci rd, csr, zimm[4:0] t = CSRs[csr]; CSRs[csr] = t &~zimm; x[rd] = t

(Control and Status Register Read and Clear Immediate). I-type, RV32I and RV64I.

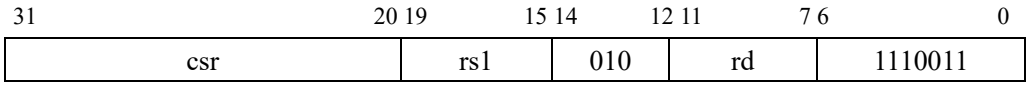
csr *t* x[*rd*] *csr* *t* 5 *zimm*



csrrs rd, csr, rs1 t = CSRs[csr]; CSRs[csr] = t | x[rs1]; x[rd] = t

(Control and Status Register Read and Set). I-type, RV32I and RV64I.

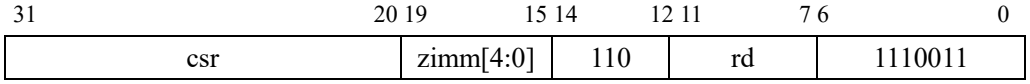
csr *t* *t* x[*rs1*] *csr* *t* x[*rd*]

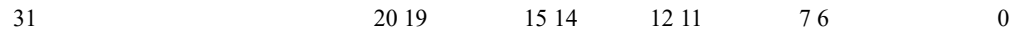


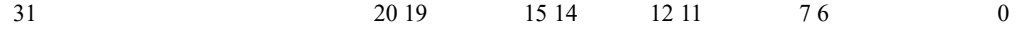
csrrci rd, csr, zimm[4:0] t = CSRs[csr]; CSRs[csr] = t | zimm; x[rd] = t

(Control and Status Register Read and Set Immediate). I-type, RV32I and RV64I.

csr *t* x[*rd*] *csr* *t* 5 *zimm*







csrc csr, rs1 CSRs[csr] |= x[rs1]

(Control and Status Register Set). (Pesudoinstruction), RV32I and RV64I.

x[*rs1*] 1 *csr* x0, csr, rs1.

---

csrci csr, zimm[4:0] CSRs[csr] |= zimm

(Control and Status Register Set Immediate). (Pesudoinstruction), RV32I and RV64I.

1 *csr*

x0, csr, zimm.

---

csrw csr, rs1 CSRs[csr] = x[rs1]

(Control and Status Register Set). (Pesudoinstruction), RV32I and RV64I.

x[*rs1*] 1 *csr* x0, csr, rs1.

---

csrwi csr, zimm[4:0] CSRs[csr] = zimm

(Control and Status Register Write Immediate). (Pesudoinstruction), RV32I and RV64I.

csr x0, csr, zimm.

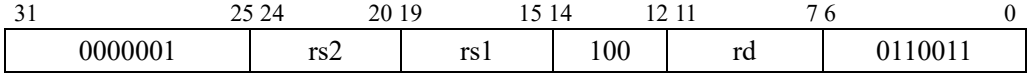
---

div rd, rs1, rs2 x[rd] = x[rs1] $_s$ x[rs2]

(Divide). R-type, RV32M and RV64M.

x[*rs1*] x[*rs2*]

x[*rd*]



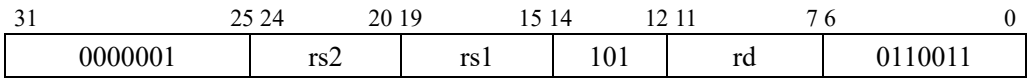
---

divu rd, rs1, rs2 x[rd] = x[rs1] $_u$ x[rs2]

(Divide, Unsigned). R-type, RV32M and RV64M.

x[*rs1*] x[*rs2*]

x[*rd*]



---

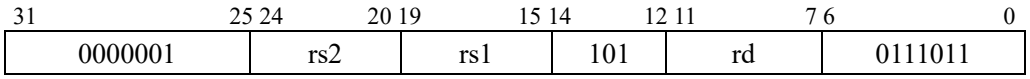
divuw rd, rs1, rs2

x[rd] = sext(x[rs1][31:0] $_u$ x[rs2][31:0])

(Divide Word, Unsigned). R-type, RV64M.

x[*rs1*] 32 x[*rs2*] 32

32 x[*rd*]



---

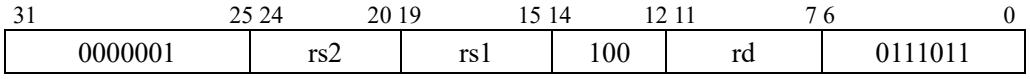
divw rd, rs1, rs2

x[rd] = sext(x[rs1][31:0] $_s$ x[rs2][31:0])

(Divide Word). R-type, RV64M.

x[*rs1*] 32 x[*rs2*] 32

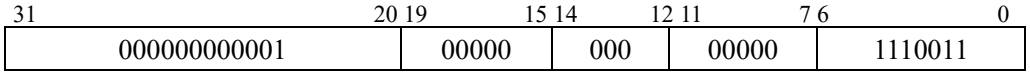
32 x[*rd*]



Ebreak

RaiseException(Breakpoint)

(Environment Breakpoint). I-type, RV32I and RV64I.

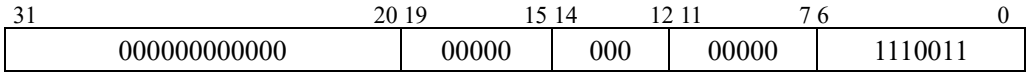


---

ecall

RaiseException(EnvironmentCall)

(Environment Call). I-type, RV32I and RV64I.



---

fabs.d rd, rs1

f[rd] = |f[rs1]|

(Floating-point Absolute Value). (Pesudoinstruction), RV32D and RV64D.

f[*rs1*] f[*rd*]

rd, rs1, rs1.

---

fabs.s rd, rs1

f[rd] = |f[rs1]|

(Floating-point Absolute Value). (Pesudoinstruction), RV32F and RV64F.

f[*rs1*] f[*rd*]

rd, rs1, rs1.

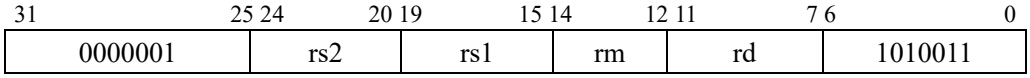
---

fadd.d rd, rs1, rs2

f [rd] = f [rs1] + f [rs2]

(Floating-point Add, Double-Precision). R-type, RV32D and RV64D.

f[*rs1*] f[*rs2*] f[*rd*]

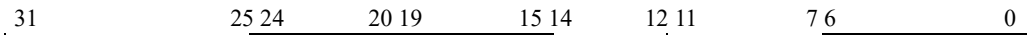


fadd.s rd, rs1, rs2

f [rd] = f [rs1] + f [rs2]

(Floating-point Add, Single-Precision). R-type, RV32F and RV64F.

f[*rs1*] f[*rs2*] f[*rd*]

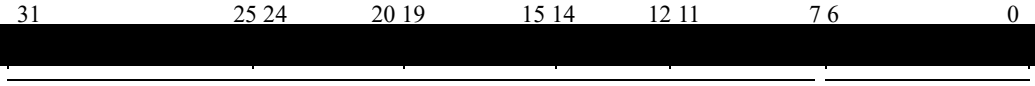


fclass.d

x[rd] = classify$_d$(f[rs1])

(-point Classify, Double-Precision) R-type, RV RV64D.

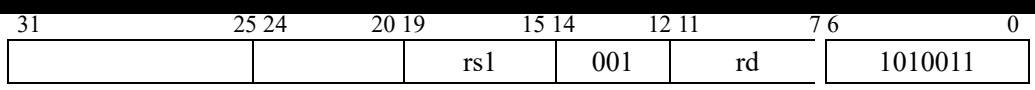
f *1*] x[*rd*] x[*rd*]



fclass. rd, rs1, rs2

x[rd] = classify$_s$(f[rs1])

(-point Classify, Single-Precision) R-type, RV32F and RV64F



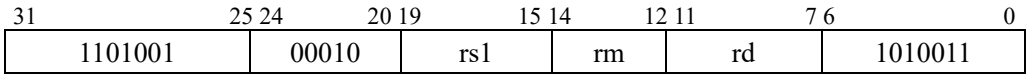
32D and

fcvt.d.l rd, rs1, rs2

f[rd] = $f64_{s64}$(x[rs1])

(Floating-point Convert to Double from Long). R-type, RV64D.

x[*rs1*] 64 f[*rd*]



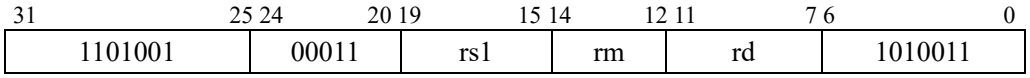
---

fcvt.d.lu rd, rs1, rs2

f[rd] = $f64_{u64}$(x[rs1])

(Floating-point Convert to Double from Unsigned Long). R-type, RV64D.

x[*rs1*] 64 f[*rd*]



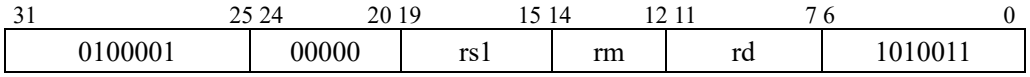
---

fcvt.d.s rd, rs1, rs2

f[rd] = $f64_{f32}$(f[rs1])

(Floating-point Convert to Double from Single). R-type, RV32D and RV64D.

f[*rs1*] f[*rd*]



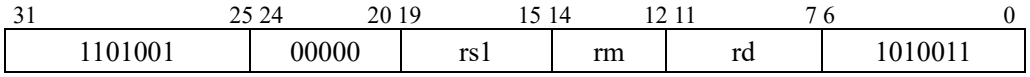
---

fcvt.d.w rd, rs1, rs2

f[rd] = $f64_{s32}$(x[rs1])

(Floating-point Convert to Double from Word). R-type, RV32D and RV64D.

x[*rs1*] 32 f[*rd*]



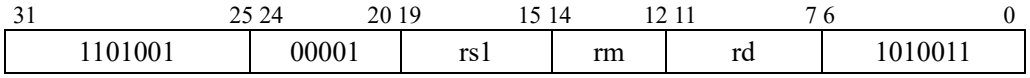
---

fcvt.d.wu rd, rs1, rs2

f[rd] = $f64_{u32}$(x[rs1])

(Floating-point Convert to Double from Unsigned Word). R-type, RV32D and RV64D.

x[*rs1*] 32 f[*rd*]



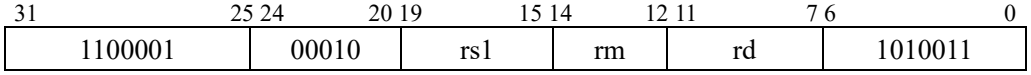
---

fcvt.l.d rd, rs1, rs2

x[rd] = s64$_{f64}$(f[rs1])

(Floating-point Convert to Long from Double). R-type, RV64D.

f[*rs1*] 64 x[*rd*]



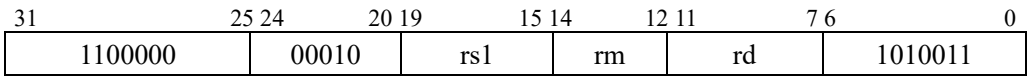
---

fcvt.l.s rd, rs1, rs2

x[rd] = s64$_{f32}$(f[rs1])

(Floating-point Convert to Long from Single). R-type, RV64F.

f[*rs1*] 64 x[*rd*]



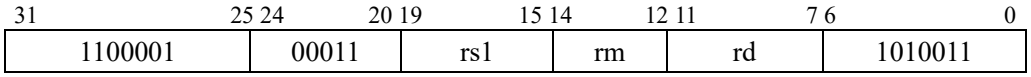
---

fcvt.lu.d rd, rs1, rs2

x[rd] = u64$_{f64}$(f[rs1])

(Floating-point Convert to Unsigned Long from Double). R-type, RV64D.

f[*rs1*] 64 x[*rd*]



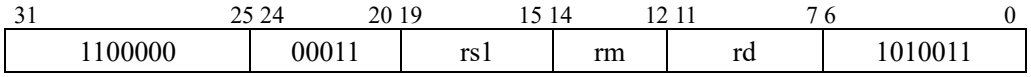
---

fcvt.lu.s rd, rs1, rs2

x[rd] = u64$_{f32}$(f[rs1])

(Floating-point Convert to Unsigned Long from Single). R-type, RV64F.

f[*rs1*] 64 x[*rd*]



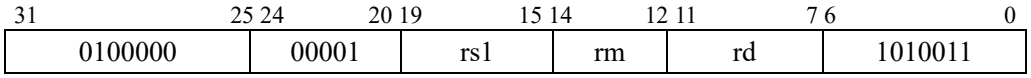
---

fcvt.s.d rd, rs1, rs2

f[rd] = f32$_{f64}$(f[rs1])

(Floating-point Convert to Single from Double). R-type, RV32D and RV64D.

f[*rs1*] f[*rd*]



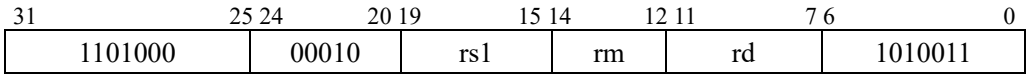
---

fcvt.s.l rd, rs1, rs2

f[rd] = f32$_{s64}$(x[rs1])

(Floating-point Convert to Single from Long). R-type, RV64F.

x[*rs1*] 64 f[*rd*]



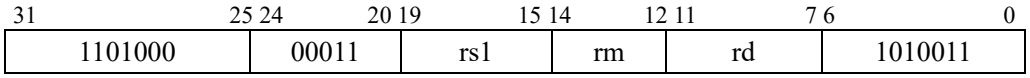
---

fcvt.s.lu rd, rs1, rs2

f[rd] = f32$_{u64}$(x[rs1])

(Floating-point Convert to Single from Unsigned Long). R-type, RV64F.

x[*rs1*] 64 f[*rd*]



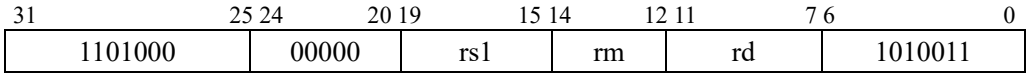
---

fcvt.s.w rd, rs1, rs2

f[rd] = f32$_{s32}$(x[rs1])

(Floating-point Convert to Single from Word). R-type, RV32F and RV64F.

x[*rs1*] 32 f[*rd*]



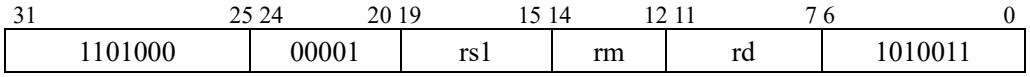
---

fcvt.s.wu rd, rs1, rs2

f[rd] = f32$_{u32}$(x[rs1])

(Floating-point Convert to Single from Unsigned Word). R-type, RV32F and RV64F.

x[*rs1*] 32 f[*rd*]



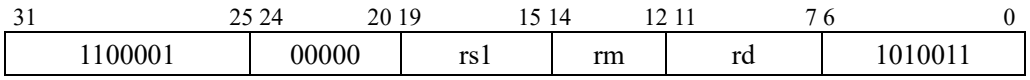
---

fcvt.w.d rd, rs1, rs2

x[rd] = sext(s32$_{f64}$(f[rs1]))

(Floating-point Convert to Word from Double). R-type, RV32D and RV64D.

f[*rs1*] 32 x[*rd*]



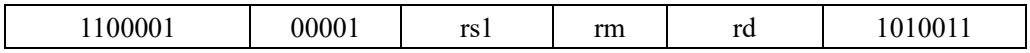
---

fcvt.wu.d rd, rs1, rs2

x[rd] = sext($u32_{f64}$(f[rs1]))

(Floating-point Convert to Unsigned Word from Double). R-type, RV32D and RV64D.

f[*rs1*] 32 x[*rd*]



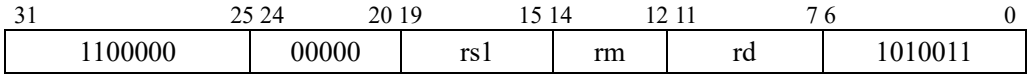
---

fcvt.w.s rd, rs1, rs2

x[rd] = sext($s32_{f32}$(f[rs1]))

(Floating-point Convert to Word from Single). R-type, RV32F and RV64F.

f[*rs1*] 32 x[*rd*]



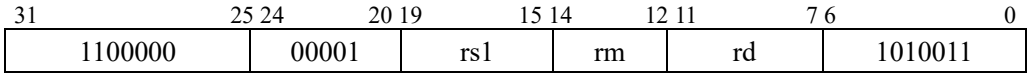
---

fcvt.wu.s rd, rs1, rs2

x[rd] = sext($u32_{f32}$(f[rs1]))

(Floating-point Convert to Unsigned Word from Single). R-type, RV32F and RV64F.

f[*rs1*] 32 x[*rd*]



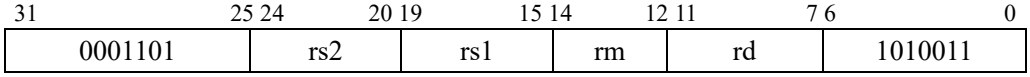
---

fdiv.d rd, rs1, rs2

f[rd] = f[rs1] f[rs2]

(Floating-point Divide, Double-Precision). R-type, RV32D and RV64D.

f[*rs1*] f[*rs2*] f[*rd*]



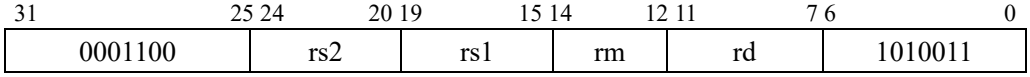
---

fdiv.s rd, rs1, rs2

f[rd] = f[rs1] f[rs2]

(Floating-point Divide, Single-Precision). R-type, RV32F and RV64F.

f[*rs1*] f[*rs2*] f[*rd*]



---

fence pred, succ

Fence(pred, succ)

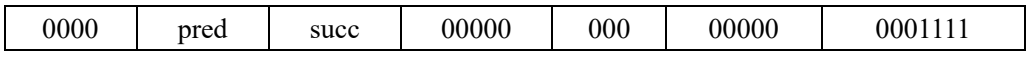
I/O*(Fence Memory and I/O)*. I-type, RV32I and RV64I.

I/O

I/O 3,2,1 0

r, rw *pred* = 0010 *succ* = 0011

iorw, iorw

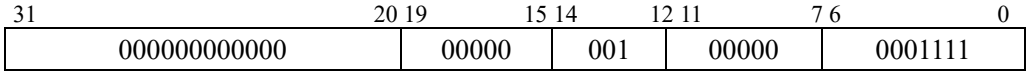


---

fence.i

Fence(Store, Fetch)

(Fence Instruction Stream). I-type, RV32I and RV64I.



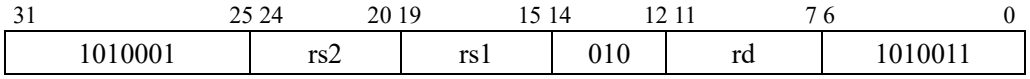
---

feq.d rd, rs1, rs2

x[rd] = f[rs1] == f[rs2]

(Floating-point Equals, Double-Precision). R-type, RV32D and RV64D.

f[*rs1*] f[*rs2*] x[*rd*] 1 0



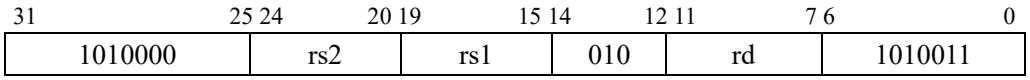
---

feq.s rd, rs1, rs2

x[rd] = f[rs1] == f[rs2]

(Floating-point Equals, Single-Precision). R-type, RV32F and RV64F.

f[*rs1*] f[*rs2*] x[*rd*] 1 0



---

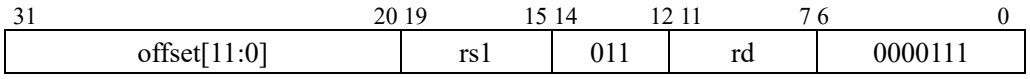
fld rd, offset(rs1)

f[rd] = M[x[rs1] + sext(offset)][63:0]

(Floating-point Load Doubleword). I-type, RV32D and RV64D.

x[*rs1*] + *sign-extend*(*offset*) f[*rd*]

rd, offset; rd, offset(rs1)



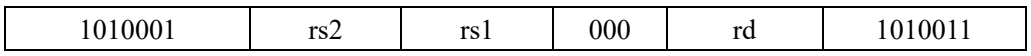
---

fle.d rd, rs1, rs2

x[rd] = f[rs1] f[rs2]

(Floating-point Less Than or Equal, Double-Precision). R-type, RV32D and RV64D.

f[*rs1*] f[*rs2*] x[*rd*] 1 0



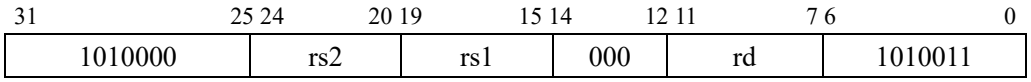
---

fle.s rd, rs1, rs2

x[rd] = f[rs1] f[rs2]

(Floating-point Less Than or Equal, Single-Precision). R-type, RV32F and RV64F.

f[*rs1*] f[*rs2*] x[*rd*] 1 0



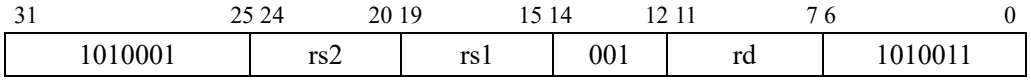
---

fle.d rd, rs1, rs2

x[rd] = f[rs1] < f[rs2]

(Floating-point Less Than, Double-Precision). R-type, RV32D and RV64D.

f[*rs1*] f[*rs2*] x[*rd*] 1 0



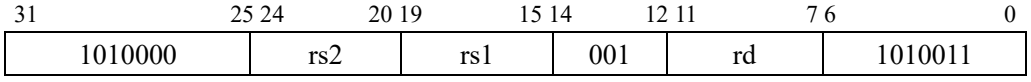
---

fle.s rd, rs1, rs2

x[rd] = f[rs1] < f[rs2]

(Floating-point Less Than, Single-Precision). R-type, RV32F and RV64F.

f[*rs1*] f[*rs2*] x[*rd*] 1 0



---

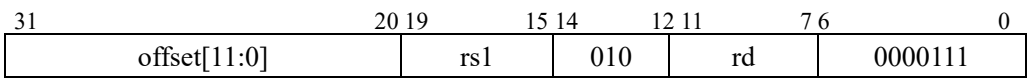
flw rd, offset(rs1)

f[rd] = M[x[rs1] + sext(offset)][31:0]

(Floating-point Load Word). I-type, RV32F and RV64F.

x[*rs1*] + *sign-extend*(*offset*) f[*rd*]

rd, offset; rd, offset(rs1)



---

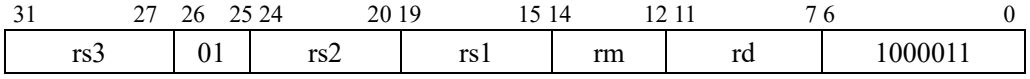
fmadd.d rd, rs1, rs2, rs3

f[rd] = f[rs1] f[rs2]+f[rs3]

(Floating-point Fused Multiply-Add, Double-Precision). R4-type, RV32D and RV64D.

f[*rs1*] f[*rs2*] f[*rs3*]

f[*rd*]



---

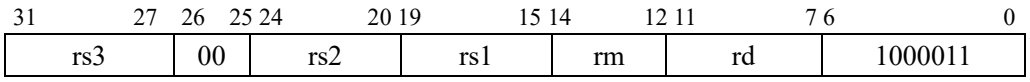
fmadd.s rd, rs1, rs2, rs3

f[rd] = f[rs1] f[rs2]+f[rs3]

(Floating-point Fused Multiply-Add, Single-Precision). R4-type, RV32F and RV64F.

f[*rs1*] f[*rs2*] f[*rs3*]

f[*rd*]



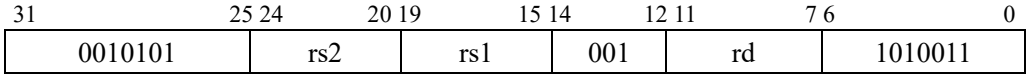
---

fmax.d rd, rs1, rs2

f[rd] = max(f[rs1], f[rs2])

(Floating-point Maximum, Double-Precision). R-type, RV32D and RV64D.

f[*rs1*] f[*rs2*] f[*rd*]



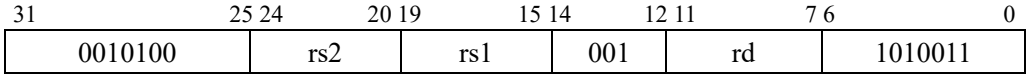
---

fmax.s rd, rs1, rs2

f[rd] = max(f[rs1], f[rs2])

(Floating-point Maximum, Single-Precision). R-type, RV32F and RV64F.

f[*rs1*] f[*rs2*] f[*rd*]



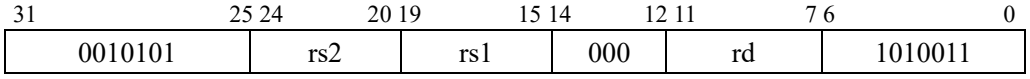
---

fmin.d rd, rs1, rs2

f[rd] = min(f[rs1], f[rs2])

(Floating-point Minimum, Double-Precision). R-type, RV32D and RV64D.

f[*rs1*] f[*rs2*] f[*rd*]



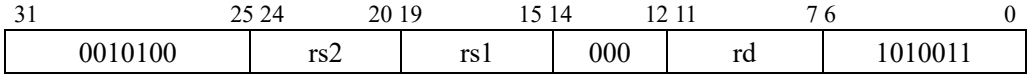
---

fmin.s rd, rs1, rs2

f[rd] = min(f[rs1], f[rs2])

(Floating-point Minimum, Single-Precision). R-type, RV32F and RV64F.

f[*rs1*] f[*rs2*] f[*rd*]



---

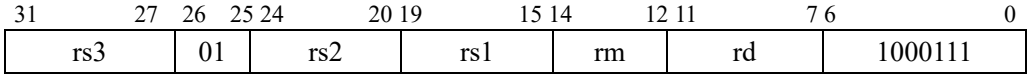
fmsub.d rd, rs1, rs2, rs3

f[rd] = f[rs1] f[rs2]-f[rs3]

(Floating-point Fused Multiply-Subtract, Double-Precision). R4-type, RV32D and RV64D.

f[*rs1*] f[*rs2*] f[*rs3*]

f[*rd*]



---

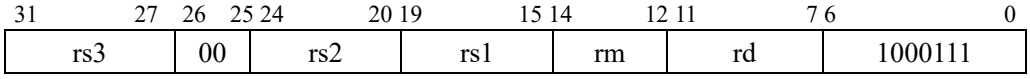
fmsub.s rd, rs1, rs2, rs3

f[rd] = f[rs1] f[rs2]-f[rs3]

(Floating-point Fused Multiply-Subtarct, Single-Precision). R4-type, RV32F and RV64F.

f[*rs1*] f[*rs2*] f[*rs3*]

f[*rd*]



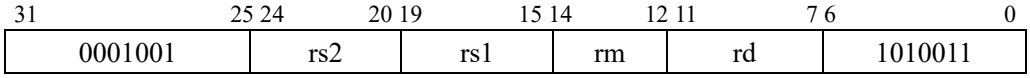
---

fmul.d rd, rs1, rs2

f[rd] = f[rs1] f[rs2]

(Floating-point Multiply, Double-Precision). R-type, RV32D and RV64D.

f[*rs1*] f[*rs2*] f[*rd*]



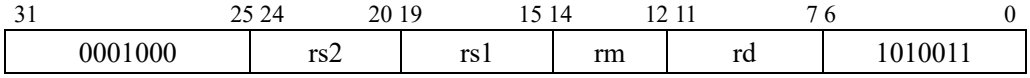
---

fmul.s rd, rs1, rs2

f[rd] = f[rs1] f[rs2]

(Floating-point Multiply, Single-Precision). R-type, RV32F and RV64F.

f[*rs1*] f[*rs2*] f[*rd*]



---

fmv.d rd, rs1

f[rd] = f[rs1]

(Floating-point Move). (Pesudoinstruction), RV32D and RV64D.

f[*rs1*] f[*rd*] rd, rs1, rs1.

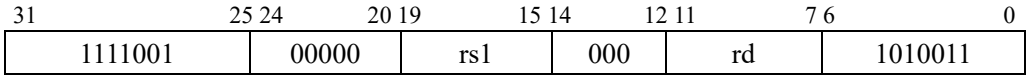
---

fmv.d.x rd, rs1, rs2

f[rd] = x[rs1][63:0]

(Floating-point Move Doubleword from Integer). R-type, RV64D.

x[*rs1*] f[*rd*]



---

fmv.s rd, rs1

f[rd] = f[rs1]

(Floating-point Move). (Pesudoinstruction), RV32F and RV64F.

f[*rs1*] f[*rd*] rd, rs1, rs1.

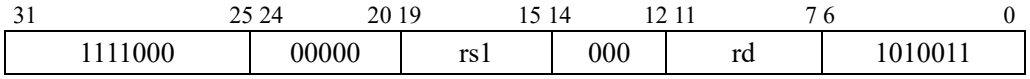
---

fmv.d.x rd, rs1, rs2

f[rd] = x[rs1][31:0]

(Floating-point Move Word from Integer). R-type, RV32F and RV64F.

x[*rs1*] f[*rd*]



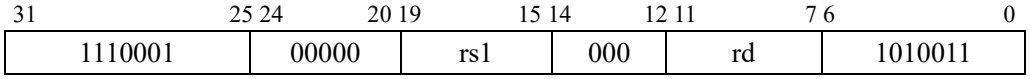
---

fmv.x.d rd, rs1, rs2

x[rd] = f[rs1][63:0]

(Floating-point Move Doubleword to Integer). R-type, RV64D.

f[*rs1*] x[*rd*]



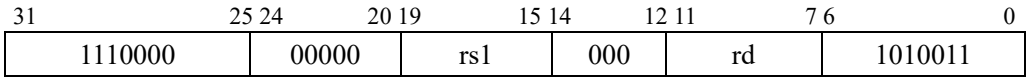
---

fmv.x.w rd, rs1, rs2

x[rd] = sext(f[rs1][31:0])

(Floating-point Move Word to Integer). R-type, RV32F and RV64F.

f[*rs1*] x[*rd*] RV64F



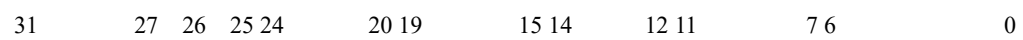
---

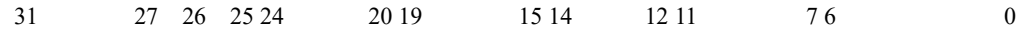
fneg.d rd, rs1

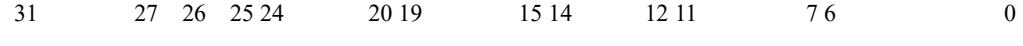
f[rd] = -f[rs1]

(Floating-point Negate). (Pesudoinstruction), RV32D and RV64D.

f[*rs1*] f[*rd*] rd, rs1, rs1.







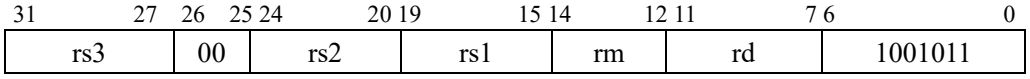
fnmsub.s rd, rs1, rs2, rs3

f[rd] = -f[rs1] f[rs2]+f[rs3]

(Floating-point Fused Negative Multiply-Subtract, Single-Precision). R4-type, RV32F and RV64F.

f[*rs1*] f[*rs2*]

f[*rs3*] f[*rd*]



---

frcsr rd

x[rd] = CSRs[fcsr]

(Floating-point Read Control and Status Register). (Pseudoinstruction), RV32F and RV64F.

x[*rd*] rd, fcsr, x0.

---

frflags rd

x[rd] = CSRs[fflags]

(Floating-point Read Exception Flags). (Pseudoinstruction), RV32F and RV64F.

x[*rd*] rd, fflags, x0.

---

frrm rd

x[rd] = CSRs[frm]

(Floating-point Read Rounding Mode). (Pseudoinstruction), RV32F and RV64F.

x[*rd*] rd, frm, x0.

---

fscsr rd, rs1

t = CSRs[fcsr]; CSRs[fcsr] = x[rs1]; x[rd] = t

(Floating-point Swap Control and Status Register). (Pseudoinstruction), RV32F and RV64F.

x[*rs1*] x[*rd*]

rd, fcsr, rs1 *rd* x0

---

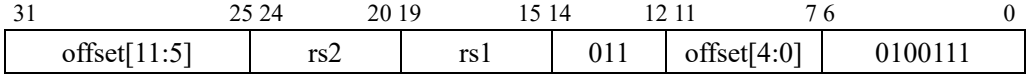
fsd rs2, offset(rs1)

M[x[rs1] + sext(offset)] = f[rs2][63:0]

(Floating-point Store Doubleword). S-type, RV32D and RV64D.

f[*rs2*] x[*rs1*] + *sign-extend*(*offset*)

rs2, offset; rs2, offset(rs1)



---

fsflags rd, rs1

t = CSRs[fflags]; CSRs[fflags] = x[rs1]; x[rd] = t

(Floating-point Swap Exception Flags). (Pseudoinstruction), RV32F and RV64F.

x[*rs1*] x[*rd*]

rd, fflags, rs1 *rd* x0

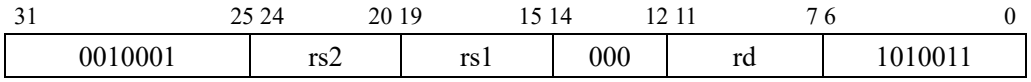
---

fsgnj.d rd, rs1, rs2

f[rd] = {f[rs2][63], f[rs1][62:0]}

(Floating-point Sign Inject, Double-Precision). R-type, RV32D and RV64D.

f[*rs1*] f[*rs2*]
f[*rd*]



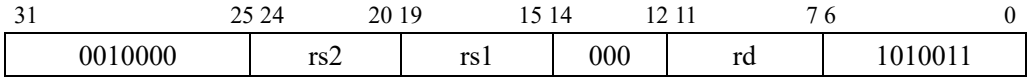
---

fsgnj.s rd, rs1, rs2

f[rd] = {f[rs2][31], f[rs1][30:0]}

(Floating-point Sign Inject, Single-Precision). R-type, RV32F and RV64F.

f[*rs1*] f[*rs2*]
f[*rd*]



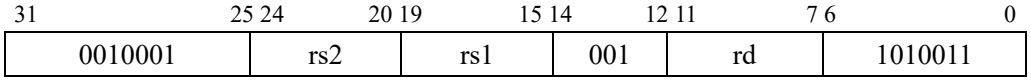
---

fsgnjn.d rd, rs1, rs2

f[rd] = {~f[rs2][63], f[rs1][62:0]}

(Floating-point Sign Inject-Negate, Double-Precision). R-type, RV32D and RV64D.

f[*rs1*] f[*rs2*]
f[*rd*]



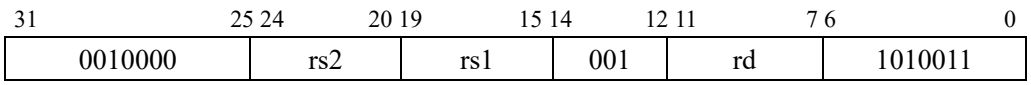
---

fsgnjn.s rd, rs1, rs2

f[rd] = {~f[rs2][31], f[rs1][30:0]}

(Floating-point Sign Inject-Negate, Single-Precision). R-type, RV32F and RV64F.

f[*rs1*] f[*rs2*]
f[*rd*]



---

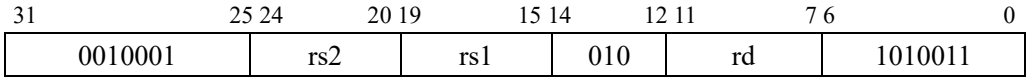
fsgnjx.d rd, rs1, rs2

f[rd] = {f[rs1][63] ^ f[rs2][63], f[rs1][62:0]}

(Floating-point Sign Inject-XOR, Double-Precision). R-type, RV32D and RV64D.

f[*rs1*] f[*rs1*] f[*rs2*]

f[*rd*]



---

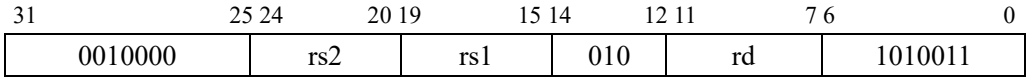
fsgnjx.s rd, rs1, rs2

f[rd] = {f[rs1][31] ^ f[rs2][31], f[rs1][30:0]}

(Floating-point Sign Inject-XOR, Single-Precision). R-type, RV32F and RV64F.

f[*rs1*] f[*rs1*] f[*rs2*]

f[*rd*]



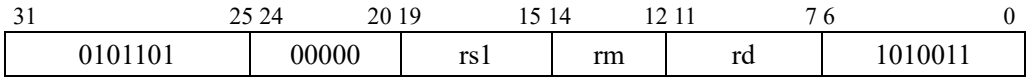
---

fsqrt.d rd, rs1, rs2

f[rd] = $\overline{\text{E vx}}$

(Floating-point Square Root, Double-Precision). R-type, RV32D and RV64D.

f[*rs1*] f[*rd*]



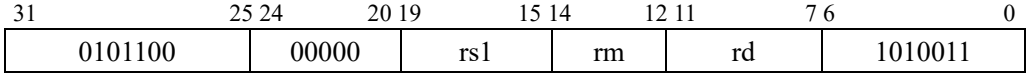
---

fsqrt.s rd, rs1, rs2

f[rd] = $\overline{\text{E vx}}$

(Floating-point Square Root, Single-Precision). R-type, RV32F and RV64F.

f[*rs1*] f[*rd*]



---

fsrm rd, rs1

t = CSRs[frm]; CSRs[frm] = x[rs1]; x[rd] = t

(Floating-point Swap Rounding Mode). (Pseudoinstruction), RV32F and RV64F.

x[*rs1*] x[*rd*]

rd, frm, rs1 *rd* x0

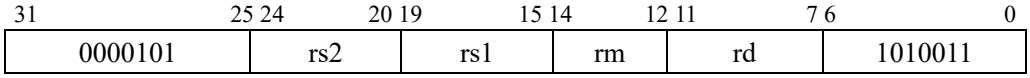
---

fsub.d rd, rs1, rs2

f[rd] = f[rs1] - f[rs2]

(Floating-point Subtract, Double-Precision). R-type, RV32D and RV64D.

f[*rs1*] f[*rs2*] f[*rd*]



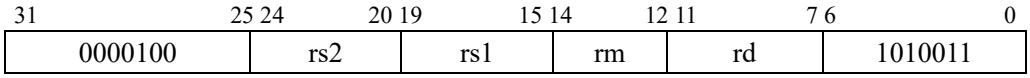
---

fsub.s rd, rs1, rs2

f[rd] = f[rs1] - f[rs2]

(Floating-point Subtract, Single-Precision). R-type, RV32F and RV64F.

f[*rs1*] f[*rs2*] f[*rd*]



---

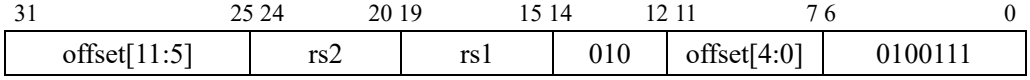
fsw rs2, offset(rs1)

M[x[rs1] + sext(offset)] = f[rs2][31:0]

(Floating-point Store Word). S-type, RV32F and RV64F.

f[*rs2*] x[*rs1*] + *sign-extend*(*offset*)

rs2, offset; rs2, offset(rs1)



---

j offset

pc += sext(offset)

(Jump). (Pseudoinstruction), RV32I and RV64I.

pc *offset* x0, offset.

---

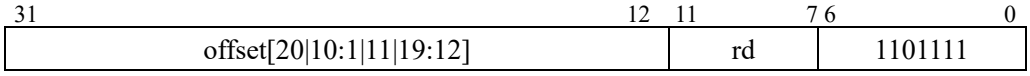
jal rd, offset

x[rd] = pc+4; pc += sext(offset)

(Jump and Link). J-type, RV32I and RV64I.

(*pc+4*) *pc* *offset* *rd* x1

offset; offset



---

jalr rd, offset(rs1)

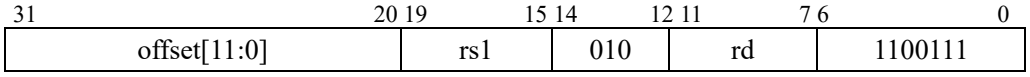
t =pc+4; pc=(x[rs1]+sext(offset))&~1; x[rd]=t

(Jump and Link Register). I-type, RV32I and RV64I.

pc x[*rs1*] + *sign-extend*(*offset*) 0 *pc+4*

f[*rd*] *rd* x1

rs1; rs1



---

jr rs1

pc = x[rs1]

(Jump Register). (Pseudoinstruction), RV32I and RV64I.

pc x[*rs1*] x0, 0(rs1)

---

la rd, symbol

x[rd] = &symbol

(Load Address). (Pseudoinstruction), RV32I and RV64I.

symbol x[*rd*]

(Global Offset Table) RV32I rd, offsetHi rd

offsetLo(rd); RV64I rd offsetHi rd, offsetLo(rd) offset

rd, offsetHi rd, rd, offsetLo

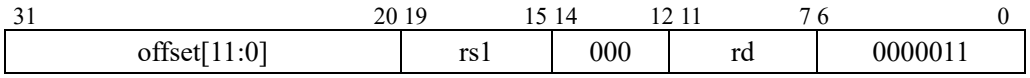
---

lb rd, offset(rs1)

x[rd] = sext(M[x[rs1] + sext(offset)][7:0])

(Load Byte). I-type, RV32I and RV64I.

x[*rs1*] + *sign-extend*(*offset*) x[*rd*]



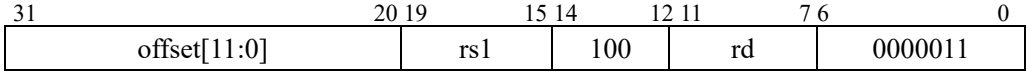
---

lbu rd, offset(rs1)

x[rd] = M[x[rs1] + sext(offset)][7:0]

(Load Byte, Unsigned). I-type, RV32I and RV64I.

x[*rs1*] + *sign-extend*(*offset*) x[*rd*]



---

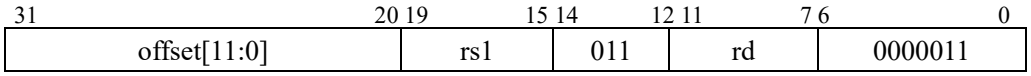
ld rd, offset(rs1)

x[rd] = M[x[rs1] + sext(offset)][63:0]

(Load Doubleword). I-type, RV32I and RV64I.

x[*rs1*] + *sign-extend*(*offset*) x[*rd*]

rd, offset; rd, offset(rs1)



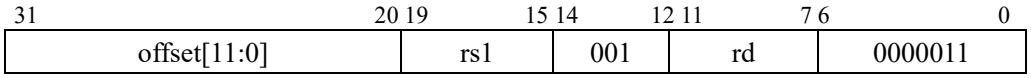
---

lh rd, offset(rs1)

x[rd] = sext(M[x[rs1] + sext(offset)][15:0])

(Load Halfword). I-type, RV32I and RV64I.

x[*rs1*] + *sign-extend*(*offset*) x[*rd*]



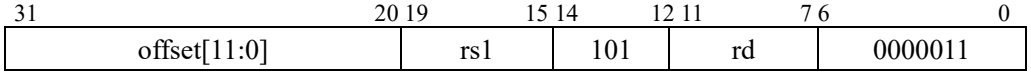
---

lhu rd, offset(rs1)

x[rd] = M[x[rs1] + sext(offset)][15:0]

(Load Halfword, Unsigned). I-type, RV32I and RV64I.

x[*rs1*] + *sign-extend*(*offset*) x[*rd*]



---

li rd, immediate

x[rd] = immediate

(Load Immediate). (Pseudoinstruction), RV32I and RV64I.

x[*rd*] RV32I /

RV64I

---

lla rd, symbol

x[rd] = &symbol

(Load Local Address). (Pseudoinstruction), RV32I and RV64I.

symbol x[*rd*] rd, offsetHi rd, rd, offsetLo

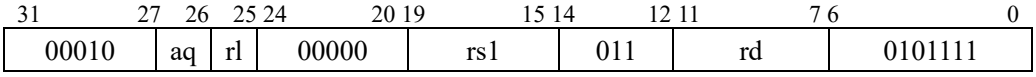
---

lr.d rd, (rs1)

x[rd] = LoadReserved64(M[x[rs1]])

(Load-Reserved Doubleword). R-type, RV64A.

x[*rs1*] x[*rd*]



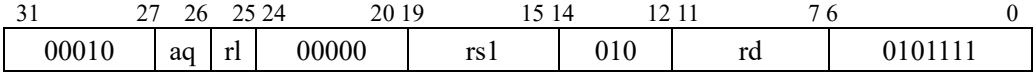
---

lr.w rd, (rs1)

x[rd] = LoadReserved32(M[x[rs1]])

(Load-Reserved Word). R-type, RV32A and RV64A.

x[*rs1*] x[*rd*]



---

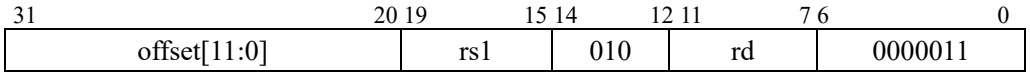
lw rd, offset(rs1)

x[rd] = sext(M[x[rs1] + sext(offset)][31:0])

(Load Word). I-type, RV32I and RV64I.

x[*rs1*] + *sign-extend*(*offset*) x[*rd*] RV64I

rd, offset; rd, offset(rs1)



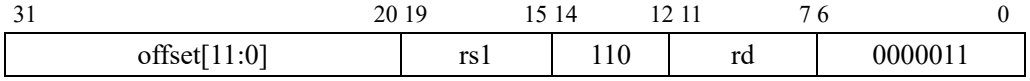
---

lwu rd, offset(rs1)

x[rd] = M[x[rs1] + sext(offset)][31:0]

(Load Word, Unsigned). I-type, RV64I.

x[*rs1*] + *sign-extend*(*offset*) x[*rd*]



---

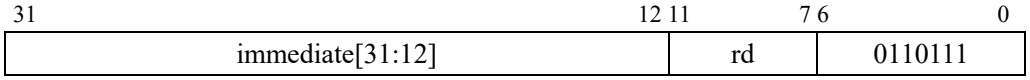
lui rd, immediate

x[rd] = sext(immediate[31:12] << 12)

(Load Upper Immediate). U-type, RV32I and RV64I.

20 *immediate* 12 12 x[*rd*]

rd, imm



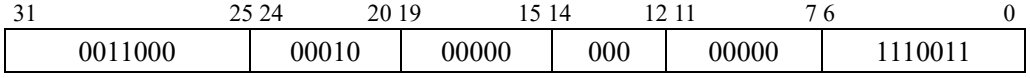
---

mret

ExceptionReturn(Machine)

(Machine-mode Exception Return). R-type, RV32I and RV64I

pc CSRs[mepc],
CSRs[mstatus].MPP, CSRs[mstatus].MIE CSRs[mstatus].MPIE,
CSRs[mstatus].MPIE 1; CSR [mstatus].MPP 0

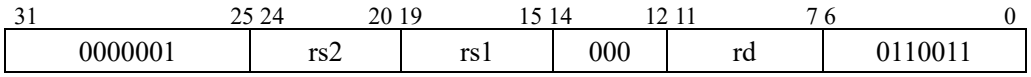


---

mul rd, rs1, rs2 wi vx vx

(Multiply). R-type, RV32M and RV64M.

x[*rs2*] x[*rs1*] x[*rd*]

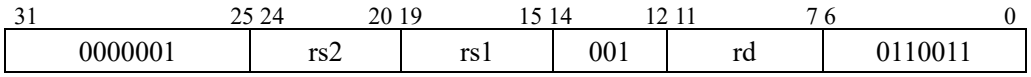


---

mulh rd, rs1, rs2 wi vx E vx S

(Multiply High). R-type, RV32M and RV64M.

x[*rs2*] x[*rs1*] 2 x[*rd*]

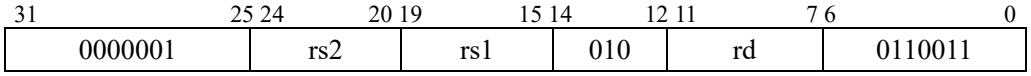


---

mulhsu rd, rs1, rs2 wi vx E vx S

(Multiply High Signed-Unsigned). R-type, RV32M and RV64M.

x[*rs2*] x[*rs1*] x[*rs1*] 2 x[*rs2*]
x[*rd*]

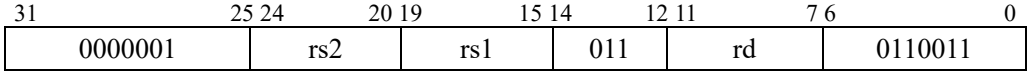


---

mulhu rd, rs1, rs2 wi vx E vx S

(Multiply High Unsigned). R-type, RV32M and RV64M.

x[*rs2*] x[*rs1*] x[*rs1*] x[*rs2*] x[*rd*]

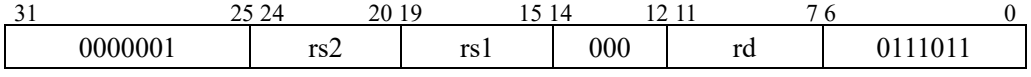


---

mulw rd, rs1, rs2 wi x y vx vx

(Multiply Word). R-type, RV64M only.

x[*rs2*] x[*rs1*] 32 x[*rd*]



---

mv rd, rs1 wi vx

(Move). (Pseudoinstruction), RV32I and RV64I.

x[*rs1*] x[*rd*] rd, rs1, 0

---

neg rd, rs2 vw vx

(Negate). (Pseudoinstruction), RV32I and RV64I.

x[*rs2*] x[*rd*] rd, x0, rs2

---

negw rd, rs2 vw x y vx

(Negate Word). (Pseudoinstruction), RV64I only.

x[*rs2*] 2 32 x[*rd*]

rd, x0, rs2

---

nop

(No operation). (Pseudoinstruction), RV32I and RV64I.

pc x0, x0, 0

---

not rd, rs1 vw vx

(NOT). (Pseudoinstruction), RV32I and RV64I.

x[*rs1*] 1 x[*rd*] rd, rs1, -1

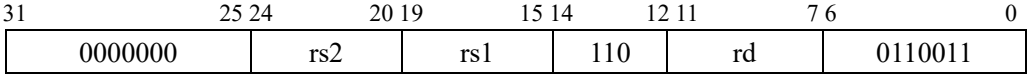
---

or rd, rs1, rs2 vw vx EE

(OR). R-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] x[*rd*]

rd, rs2



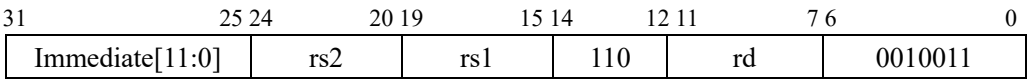
---

ori rd, rs1, immediate vw vx EEx y rr r i rfy

(OR Immediate). R-type, RV32I and RV64I.

x[*rs1*] *immediate* x[*rd*]

rd, rs2



---

rdcycle rd vw X X h h

(Read Cycle Counter). (Pseudoinstruction), RV32I and RV64I.

x[*rd*] rd, cycle, x0

---

rdcycleh rd

vir X x h h m

(Read Cycle Counte High). (Pseudoinstruction), RV32I only.

32 x[*rd*] rd, cycleh, x0

---

rdinstret rd

vir X x rs xywy

(Read Instruction-Retired Counter). (Pseudoinstruction), RV32I and RV64I.

x[*rd*] rd, instret, x0

---

rdinstreth rd

vir X x rs xywym

(Read Instruction-Retired Counter High). (Pseudoinstruction), RV32I only.

32 x[*rd*] rd, instreth, x0

---

rdtime rd

vir X x yrr

(Read Time). (Pseudoinstruction), RV32I and RV64I.

x[*rd*] rd, time, x0

---

rdtimeh rd

vir X x yrr m

(Read Time High). (Pseudoinstruction), RV32I only.

32 x[*rd*] rd, timeh, x0

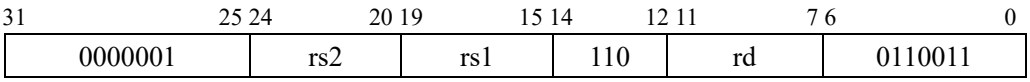
---

rem rd, rs1, rs2

vir vx E E vx

(Remainder). R-type, RV32M and RV64M.

x[*rs1*] x[*rs2*] 0 2 x[*rd*]



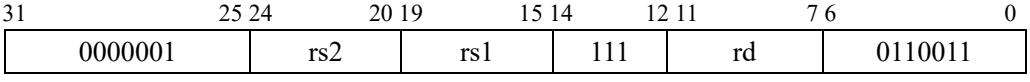
---

remu rd, rs1, rs2

vir vx E E vx

(Remainder, Unsigned). R-type, RV32M and RV64M.

x[*rs1*] x[*rs2*] 0 x[*rd*]



---

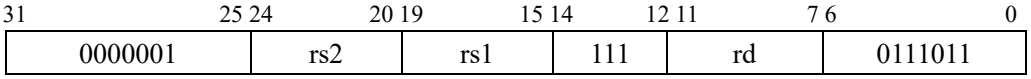
remuw rd, rs1, rs2

EEE wv x y wx E E wx

(Remainder Word, Unsigned). R-type, RV64M only.

x[*rs1*] 32 x[*rs2*] 32 0

x[*rd*]



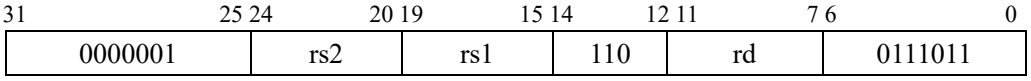
remw rd, rs1, rs2

EE wv x y wx E E wx

(Remainder Word). R-type, RV64M only.

x[*rs1*] 32 x[*rs2*] 32 0 2

x[*rd*]



ret

Euh

(Return). (Pseudoinstruction), RV32I and RV64I.

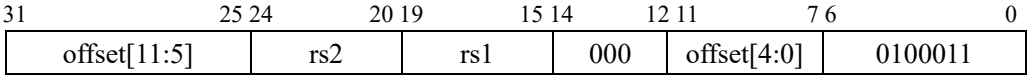
x0, 0(x1)

sb rs2, offset(rs1)

R wx x y t x y wx E

(Store Byte). S-type, RV32I and RV64I.

x[*rs2*] x[*rs1*]+*sign-extend(offset)*



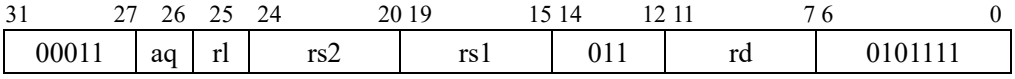
sc.d rd, rs2, (rs1)

wv Xytw tsinytsf R wx wx

(Store-Conditional Doubleword). R-type, RV64A only.

x[*rs1*] x[*rs2*] 8

x[*rd*] 0 0

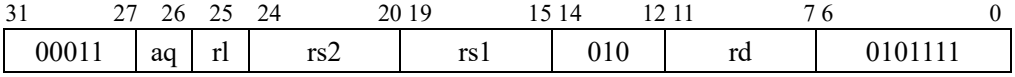


sc.w rd, rs2, (rs1) wi Xytw tsinytsf R vwx vwx

(Store-Conditional Word). R-type, RV32A and RV64A.

x[*rs1*] x[*rs2*] 4

x[*rd*] 0 0



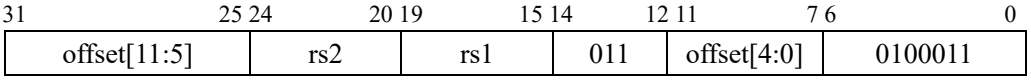
---

sd rs2, offset(rs1) R vwx x y t x y vwx

(Store Doubleword). S-type, RV64I only.

x[*rs2*] 8 x[*rs1*]+*sign-extend(offset)*

rs2, offset; rs2, offset(rs1)



---

seqz rd, rs1 wi vwx

(Set if Equal to Zero). (Pseudoinstruction), RV32I and RV64I.

x[*rs1*] 0 x[*rd*] 1 0 rd, rs1, 1

---

sext.w rd, rs1 wi x y vwx

(Sign-extend Word). (Pseudoinstruction), RV64I only.

x[*rs1*] 32 x[*rd*] rd, rs1, 0

---

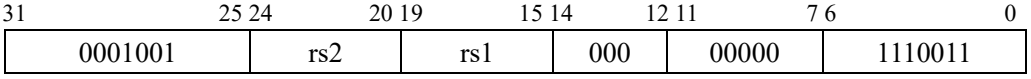
sfence.vma rs1, rs2 sh Xytw Fiiwxx wfsxfyrts

(Fence Virtual Memory). R-type, RV32I and RV64I

rs2=0

x[*rs2*] *rs1*=0

x[*rs1*]



---

sgtz rd, rs2 wi vwx

(Set if Greater Than Zero). (Pseudoinstruction), RV32I and RV64I.

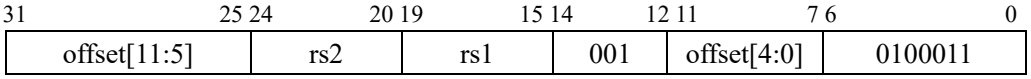
x[*rs2*] 0 x[*rd*] 1 0 rd, x0, rs2

---

sh rs2, offset(rs1) R vwx x y t x y vwx

(Store Halfword). S-type, RV32I and RV64I.

x[*rs2*] 2 x[*rs1*]+*sign-extend(offset)*



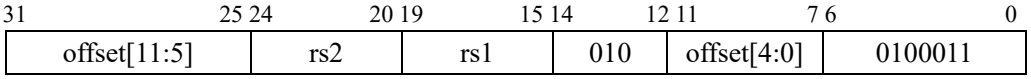
---

sw rs2, offset(rs1) R vwx x y t x y vwx

(Store Word). S-type, RV32I and RV64I.

x[*rs2*] 4 x[*rs1*]+*sign-extend(offset)*

rs2, offset; rs2, offset(rs1)



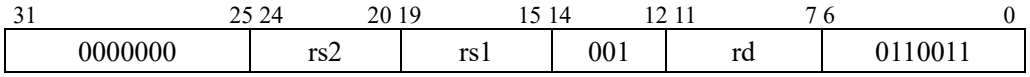
---

sll rd, rs1, rs2 vw vwx vwx

(Shift Left Logical). R-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] 0 x[*rd*] x[*rs2*] 5

RV64I 6



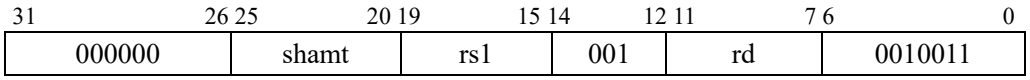
---

slli rd, rs1, shamt vw vwx xmfr y

(Shift Left Logical Immediate). I-type, RV32I and RV64I.

x[*rs1*] *shamt* 0 x[*rd*] RV32I *shamt*[5]=0

rd, shamt



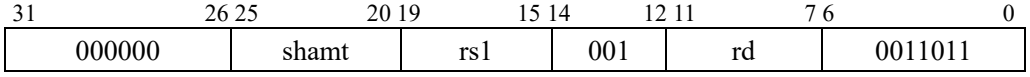
---

slliw rd, rs1, shamt vw x y vwx xmfr y

(Shift Left Logical Word Immediate). I-type, RV64I only.

x[*rs1*] *shamt* 0 32

x[*rd*] *shamt*[5]=0



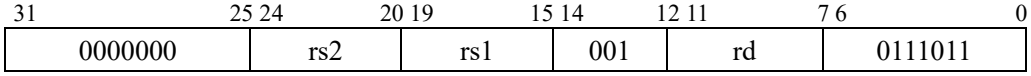
---

sllw rd, rs1, rs2 w x y vx vx

(Shift Left Logical Word). R-type, RV64I only.

x[*rs1*] 32 x[*rs2*] 0

x[*rd*] x[*rs2*] 5

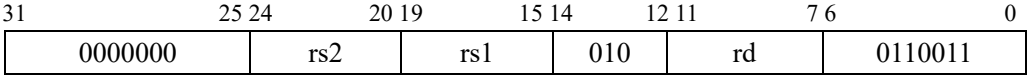


---

slt rd, rs1, rs2 E w vx vx

(Set if Less Than). R-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] x[*rs1*] x[*rd*] 1 0

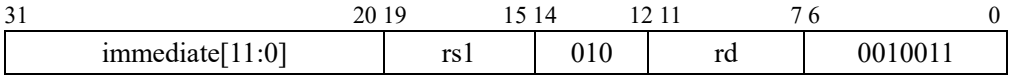


---

slti rd, rs1, immediate w vx x y m r r i rf y

(Set if Less Than Immediate). I-type, RV32I and RV64I.

x[*rs1*] *immediate* x[*rs1*] x[*rd*] 1 0



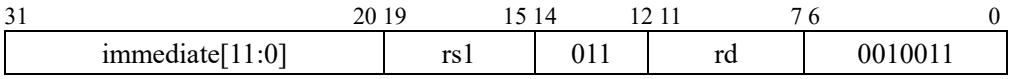
---

sltiu rd, rs1, immediate w vx x y m r r i rf y

(Set if Less Than Immediate, Unsigned). I-type, RV32I and RV64I.

x[*rs1*] *immediate* x[*rs1*] x[*rd*]

1 0

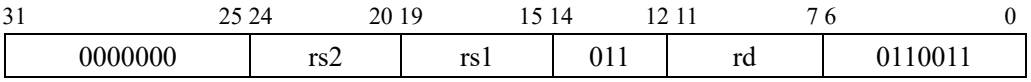


---

sltu rd, rs1, rs2 w vx vx

(Set if Less Than, Unsigned). R-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] x[*rs1*] x[*rd*] 1 0



---

sltz rd, rs1 vir vsx

(Set if Less Than to Zero). (Pseudoinstruction), RV32I and RV64I.

x[*rs1*] 0 x[*rd*] 1 0 rd, rs1, x0

---

snez rd, rs2 vir vsx

(Set if Not Equal to Zero). (Pseudoinstruction), RV32I and RV64I.

x[*rs1*] 0 x[*rd*] 1 0 rd, x0, rs2

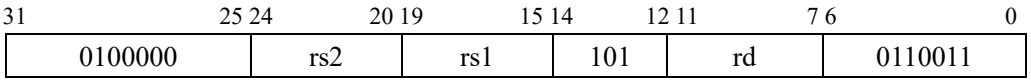
---

sra rd, rs1, rs2 vir vsx vsx

(Shift Right Arithmetic). R-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] x[*rs1*] x[*rd*] x[*rs2*] 5

RV64I 6



---

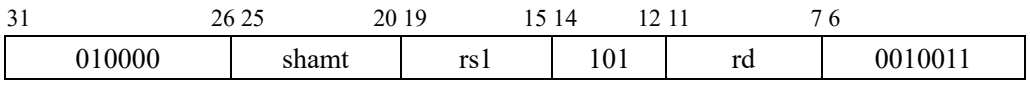
srai rd, rs1, shamt vir vsx xmfr y

(Shift Right Arithmetic Immediate). I-type, RV32I and RV64I.

x[*rs1*] *shamt* x[*rs1*] x[*rd*] RV32I

shamt[5]=0

rd, shamt



---

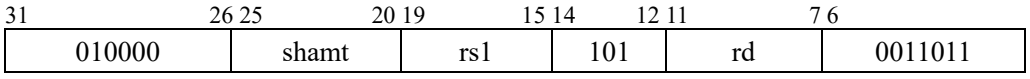
sraiw rd, rs1, shamt vir x y vsx xmfr y

(Shift Right Arithmetic Word Immediate). I-type, RV64I only.

x[*rs1*] 32 *shamt* x[*rs1*][31]

x[*rd*] *shamt*[5]=0

rd, shamt



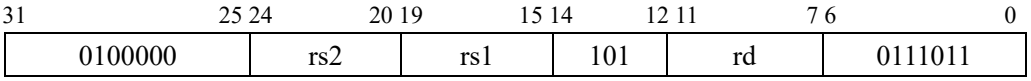
---

sraw rd, rs1, rs2 vw x y vwx vwx E

(Shift Right Arithmetic Word). R-type, RV64I only.

x[*rs1*] 32 x[*rs2*] x[*rs1*][31]

x[*rd*] x[*rs2*] 5



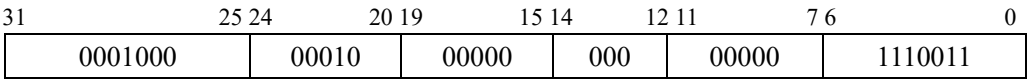
---

sret h uyrts y vs X u wrxtw

(Supervisor-mode Exception Return). R-type, RV32I and RV64I

pc CSRs[spec] CSRs[sstatus].SPP

CSRs[sstatus].SIE CSRs[sstatus].SPIE CSRs[sstatus].SPIE 1 CSRs[sstatus].spp 0



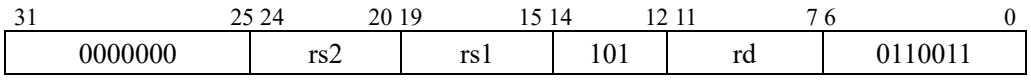
---

srl rd, rs1, rs2 vw vwx vwx

(Shift Right Logical). R-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] 0 x[*rd*] x[*rs2*] 5

RV64I 6



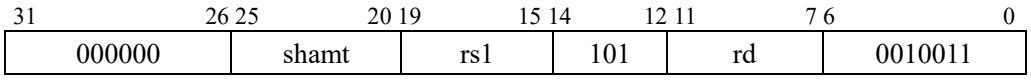
---

srli rd, rs1, shamt vw vwx xmfr y

(Shift Right Logical Immediate). I-type, RV32I and RV64I.

x[*rs1*] *shamt* 0 x[*rd*] RV32I *shamt*[5]=0

rd, shamt



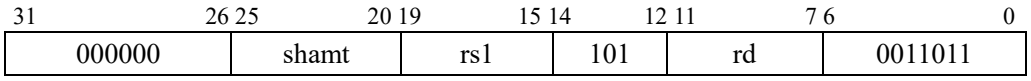
---

srliw rd, rs1, shamt vw x y vwx xmfr y

(Shift Right Logical Word Immediate). I-type, RV64I only.

x[*rs1*] *shamt* 0 32

x[*rd*] *shamt*[5]=0



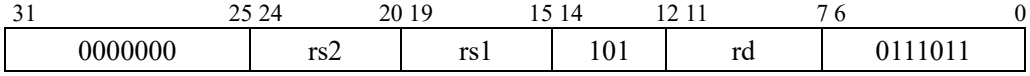
---

srlw rd, rs1, rs2 vw x y vwx vwx

(Shift Right Logical Word). R-type, RV64I only.

x[*rs1*] 32 x[*rs2*] 0

x[*rd*] x[*rs2*] 5



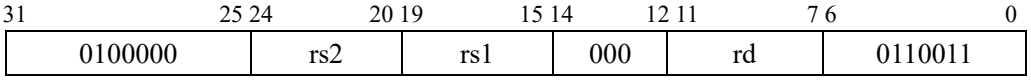
---

sub rd, rs1, rs2 vw vwx vwx

(Substract). R-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] x[*rd*]

rd, rs2



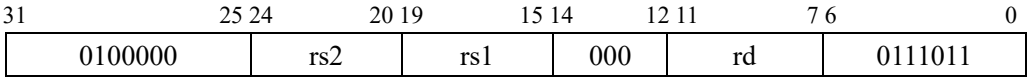
---

subw rd, rs1, rs2 vw x y vwx vwx

(Substract Word). R-type, RV64I only.

x[*rs1*] x[*rs2*] 32 x[*rd*]

rd, rs2



---

tail symbol uh x r gt Hhtgg vE

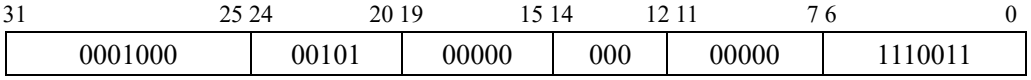
(Tail call). (Pseudoinstuction), RV32I and RV64I.

pc *symbol* x[6] x6, offsetHi x0, offsetLo(x6)

---

wfi mm EstNsy wwvuy sinsl Hi

(Wait for Interrupt). R-type, RV32I and RV64I



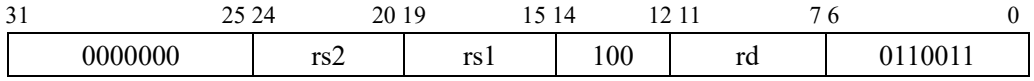
---

xor rd, rs1, rs2 vw vwx EE EE vwx

(Exclusive-OR). R-type, RV32I and RV64I.

x[*rs1*] x[*rs2*] x[*rd*]

rd, rs2



---

xori rd, rs1, immediate vw vwx EE EEx y rr r i rfy

(Exclusive-OR Immediate). I-type, RV32I and RV64I.

x[*rs1*] *immediate* x[*rd*]

rd, rs2