Computer Systems Design

Lesson 4

Introduction to RISC-V architecture

Alexander Antonov, Assoc. Prof., ITMO University

Hangzhou, 2025

Outline the lesson

- History and motivation for development of RISC-V architecture
- Specifications
- ISA structure
- Registers view
- Instruction formats
- Base ISA, extensions

RISC-V: open RISC CPU architecture

Proposed at University of California, Berkeley (2010)

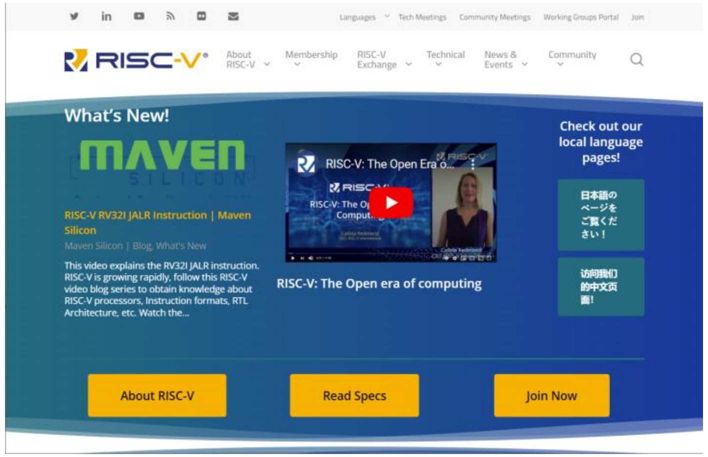
Offered multiple virtues:

1) Royalty free both for academic and industrial use
2) High-quality, clean-slate, accumulating previous experience of HW and SW design

Published under permissive BSD license

Now run by RISC-V Foundation (Switzerland)

Official web site: https://riscv.org/



RISC-V RV32I JALR Instruction | Maven Silicon

Maven Silicon | Blog, What's New

This video explains the RV32I JALR instruction. RISC-V is growing rapidly, follow this RISC-V video blog series to obtain knowledge about RISC-V processors, Instruction formats, RTL Architecture, etc. Watch the...

RISC-V: The Open era of computing

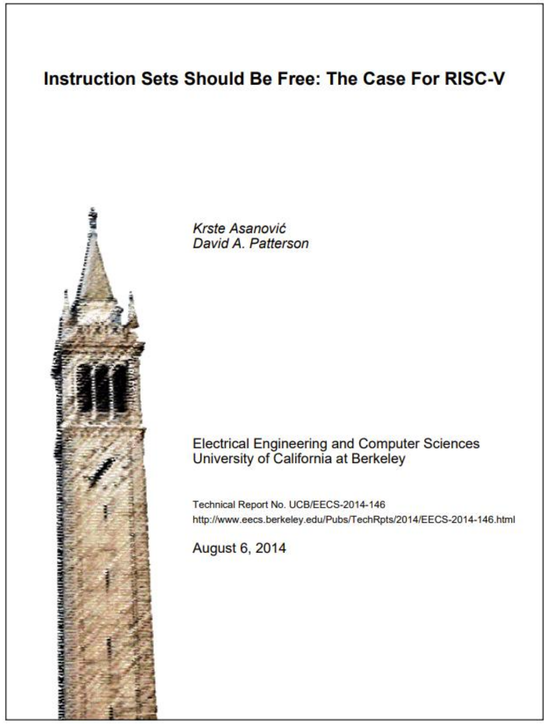
Gaining big interest both from academia and industry in recent years

RISC-V: motivation for development

Technical Report No. UCB/EECS-2014-146:
https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

Contains:

1) Justification for new ISA development
2) Base ISA



Instruction Sets Should Be Free: The Case For RISC-V

Krste Asanović
David A. Patterson

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-146
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

August 6, 2014

RISC-V: specifications

Available at: https://riscv.org/technical/specifications/

Structure of specifications:

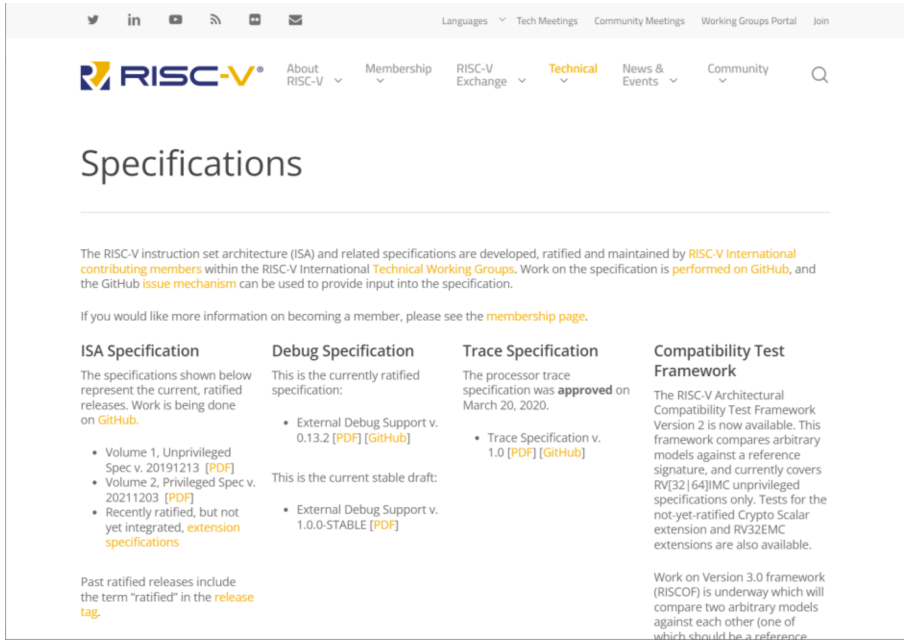
1) ISA spec, Vol. 1, unprivileged spec
   generic, computation-related ISA part

2) ISA spec, Vol. 2, privileged spec
   interaction between apps, OS, hypervisors, etc.

3) Debug Specification
   interaction with debug infrastructure

4) Trace Specification
   interaction with trace infrastructure

5) Compatibility Test Framework
   CPU tests proving compatibility with ISA



Specifications

The RISC-V instruction set architecture (ISA) and related specifications are developed, ratified and maintained by RISC-V International contributing members within the RISC-V International Technical Working Groups. Work on the specification is performed on GitHub, and the GitHub issue mechanism can be used to provide input into the specification.

If you would like more information on becoming a member, please see the membership page.

ISA Specification

The specifications shown below represent the current, ratified releases. Work is being done on GitHub.

- Volume 1, Unprivileged Spec v. 20191213 [PDF]
- Volume 2, Privileged Spec v. 20211203 [PDF]
- Recently ratified, but not yet integrated, extension specifications

Past ratified releases include the term "ratified" in the release tag.

Debug Specification

This is the currently ratified specification:

- External Debug Support v. 0.13.2 [PDF] [GitHub]

This is the current stable draft:

- External Debug Support v. 1.0.0-STABLE [PDF]

Trace Specification

The processor trace specification was **approved** on March 20, 2020.

- Trace Specification v. 1.0 [PDF] [GitHub]

Compatibility Test Framework

The RISC-V Architectural Compatibility Test Framework Version 2 is now available. This framework compares arbitrary models against a reference signature, and currently covers RV[32|64]IMC unprivileged specifications only. Tests for the not-yet-ratified Crypto Scalar extension and RV32EMC extensions are also available.

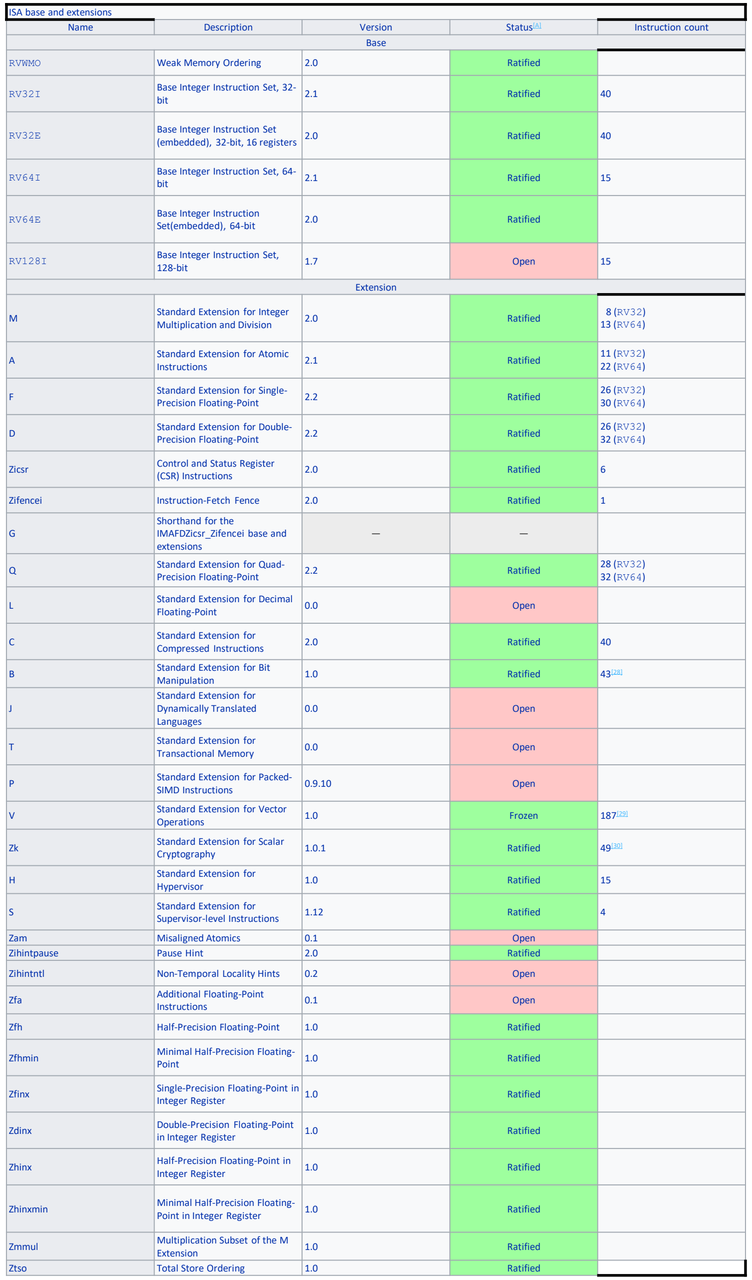
Work on Version 3.0 framework (RISCOF) is underway which will compare two arbitrary models against each other (one of which should be a reference

Used as ISA study example in our course

RISC-V: ISA structure

Designed to be modular, consists of ***base (obligatory)*** part and ***optional extensions***

- Opcode space supports ***custom extensions***
- Key elements frozen, is an active work in progress



ISA base and extensions

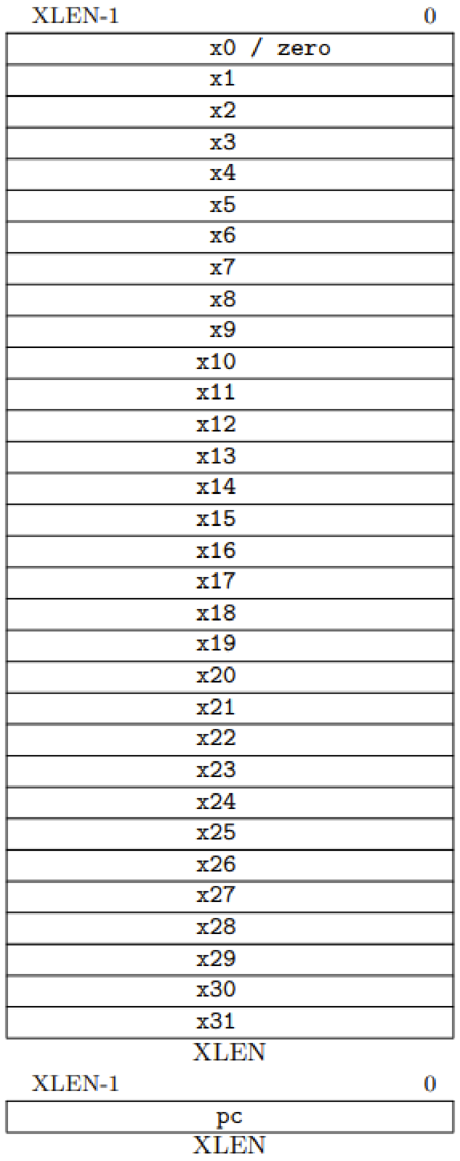
Name	Description	Version	Status[A]	Instruction count
Base				
RVWMO	Weak Memory Ordering	2.0	Ratified	
RV32I	Base Integer Instruction Set, 32-bit	2.1	Ratified	40
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers	2.0	Ratified	40
RV64I	Base Integer Instruction Set, 64-bit	2.1	Ratified	15
RV64E	Base Integer Instruction Set(embedded), 64-bit	2.0	Ratified	
RV128I	Base Integer Instruction Set, 128-bit	1.7	Open	15
Extension				
M	Standard Extension for Integer Multiplication and Division	2.0	Ratified	8 (RV32) 13 (RV64)
A	Standard Extension for Atomic Instructions	2.1	Ratified	11 (RV32) 22 (RV64)
F	Standard Extension for Single-Precision Floating-Point	2.2	Ratified	26 (RV32) 30 (RV64)
D	Standard Extension for Double-Precision Floating-Point	2.2	Ratified	26 (RV32) 32 (RV64)
Zicsr	Control and Status Register (CSR) Instructions	2.0	Ratified	6
Zifencei	Instruction-Fetch Fence	2.0	Ratified	1
G	Shorthand for the IMAFDZicsr_Zifencei base and extensions	—	—	
Q	Standard Extension for Quad-Precision Floating-Point	2.2	Ratified	28 (RV32) 32 (RV64)
L	Standard Extension for Decimal Floating-Point	0.0	Open	
C	Standard Extension for Compressed Instructions	2.0	Ratified	40
B	Standard Extension for Bit Manipulation	1.0	Ratified	43[28]
J	Standard Extension for Dynamically Translated Languages	0.0	Open	
T	Standard Extension for Transactional Memory	0.0	Open	
P	Standard Extension for Packed-SIMD Instructions	0.9.10	Open	
V	Standard Extension for Vector Operations	1.0	Frozen	187[29]
Zk	Standard Extension for Scalar Cryptography	1.0.1	Ratified	49[30]
H	Standard Extension for Hypervisor	1.0	Ratified	15
S	Standard Extension for Supervisor-level Instructions	1.12	Ratified	4
Zam	Misaligned Atomics	0.1	Open	
Zihintpause	Pause Hint	2.0	Ratified	
Zihintntl	Non-Temporal Locality Hints	0.2	Open	
Zfa	Additional Floating-Point Instructions	0.1	Open	
Zfh	Half-Precision Floating-Point	1.0	Ratified	
Zfhmin	Minimal Half-Precision Floating-Point	1.0	Ratified	
Zfinx	Single-Precision Floating-Point in Integer Register	1.0	Ratified	
Zdinx	Double-Precision Floating-Point in Integer Register	1.0	Ratified	
Zhinx	Half-Precision Floating-Point in Integer Register	1.0	Ratified	
Zhinxmin	Minimal Half-Precision Floating-Point in Integer Register	1.0	Ratified	
Zmmul	Multiplication Subset of the M Extension	1.0	Ratified	
Ztso	Total Store Ordering	1.0	Ratified	

Base ISA: register space (RV32I)

Register bank consists of 32x ***32-bit general-purpose registers*** (XLEN=32) + program counter

Only x0 register is special – always zero (writes don’t take effect)

(ISA spec, Vol. 1, unprivileged spec, p. 14)



XLEN-1 0
x0 / zero
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31
XLEN

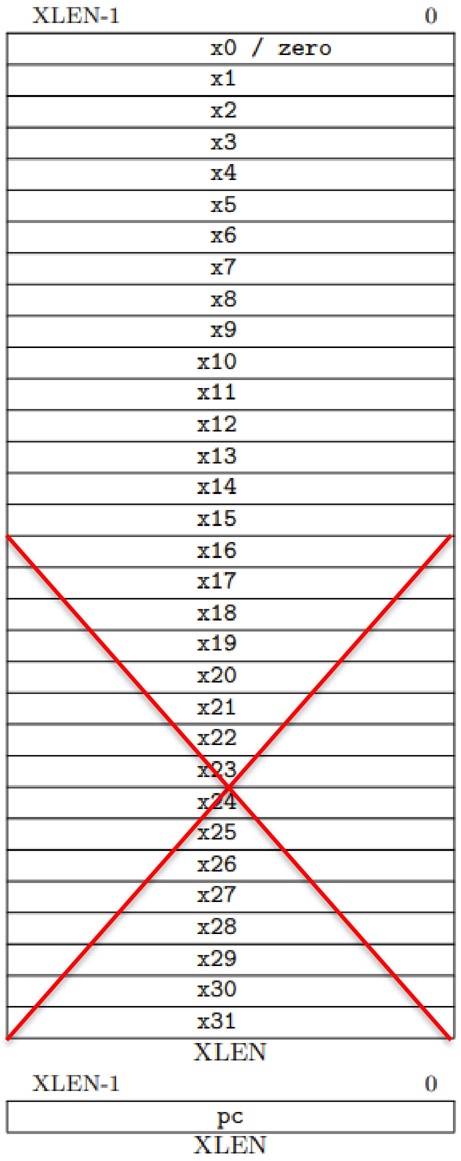
XLEN-1 0
pc
XLEN

Base ISA: register space (RV32E)

Instead of 32x, 16x 32-bit registers are provided

Targeted for embedded applications

(ISA spec, Vol. 1, unprivileged spec, p. 33)



XLEN-1 0

x0 / zero

x1

x2

x3

x4

x5

x6

x7

x8

x9

x10

x11

x12

x13

x14

x15

x16

x17

x18

x19

x20

x21

x22

x23

x24

x25

x26

x27

x28

x29

x30

x31

XLEN

XLEN-1 0

pc

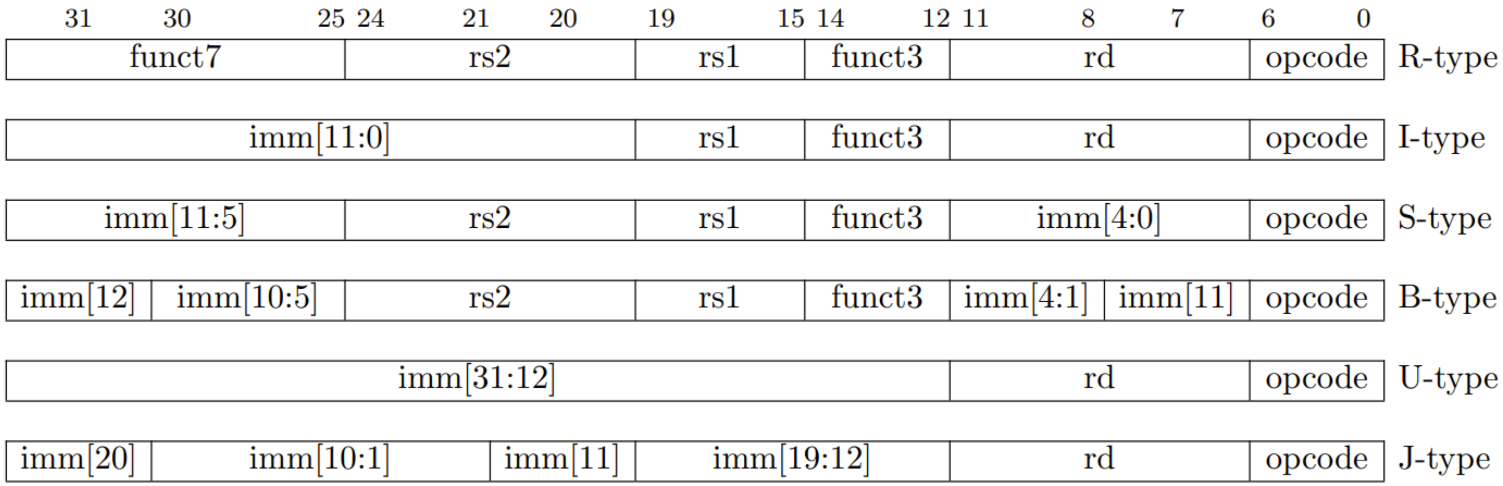
XLEN

Base ISA: instruction formats

Base instructions are ***32-bit data words*** assembled according to the following formats (ISA spec, Vol. 1, unprivileged spec, p. 16):

Priorities:

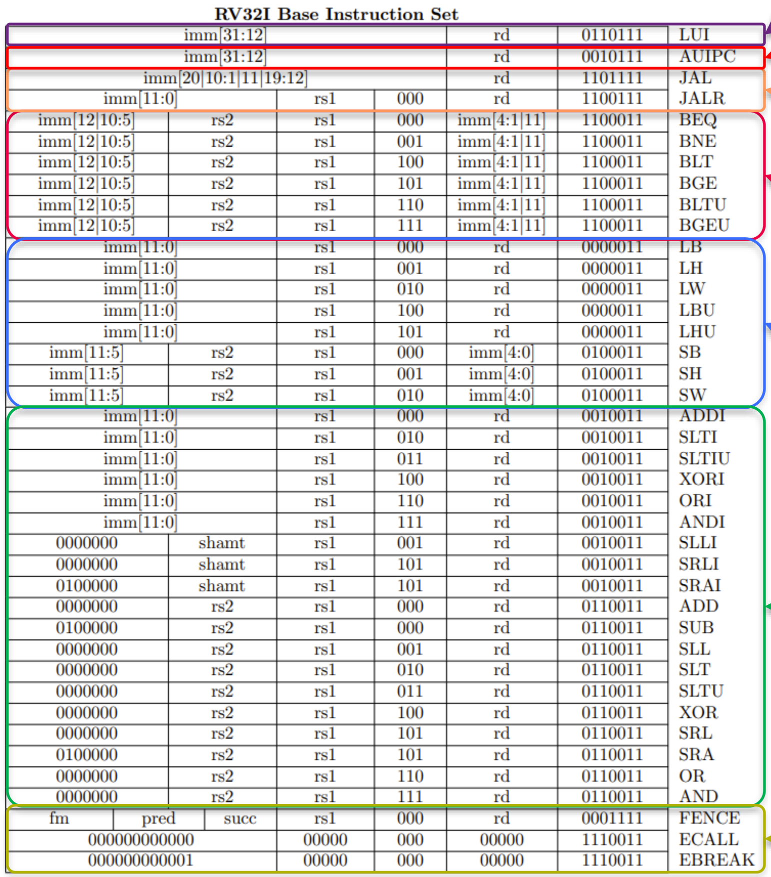
- Unification of instructions sizes and formats (preserving high code density)
- Decreasing diversity of possible locations of instruction fields for different types of instructions



<table>
<tr><td>31</td><td>30 25</td><td>24 21</td><td>20</td><td>19 15</td><td>14 12</td><td>11 8</td><td>7</td><td>6 0</td><td></td></tr>
<tr><td colspan="2">funct7</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td colspan="2">rd</td><td>opcode</td><td>R-type</td></tr>
<tr><td colspan="4">imm[11:0]</td><td>rs1</td><td>funct3</td><td colspan="2">rd</td><td>opcode</td><td>I-type</td></tr>
<tr><td colspan="2">imm[11:5]</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td colspan="2">imm[4:0]</td><td>opcode</td><td>S-type</td></tr>
<tr><td>imm[12]</td><td>imm[10:5]</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td>imm[4:1]</td><td>imm[11]</td><td>opcode</td><td>B-type</td></tr>
<tr><td colspan="6">imm[31:12]</td><td colspan="2">rd</td><td>opcode</td><td>U-type</td></tr>
<tr><td>imm[20]</td><td colspan="2">imm[10:1]</td><td>imm[11]</td><td colspan="2">imm[19:12]</td><td colspan="2">rd</td><td>opcode</td><td>J-type</td></tr>
</table>

ITMO

Base RV32I ISA



RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred	succ	rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Write constant to register (high bits)

Writing PC+offset to register
(for pc-relative address construction)

Branches with memorizing return points
(for procedure calls)

Conditional branches

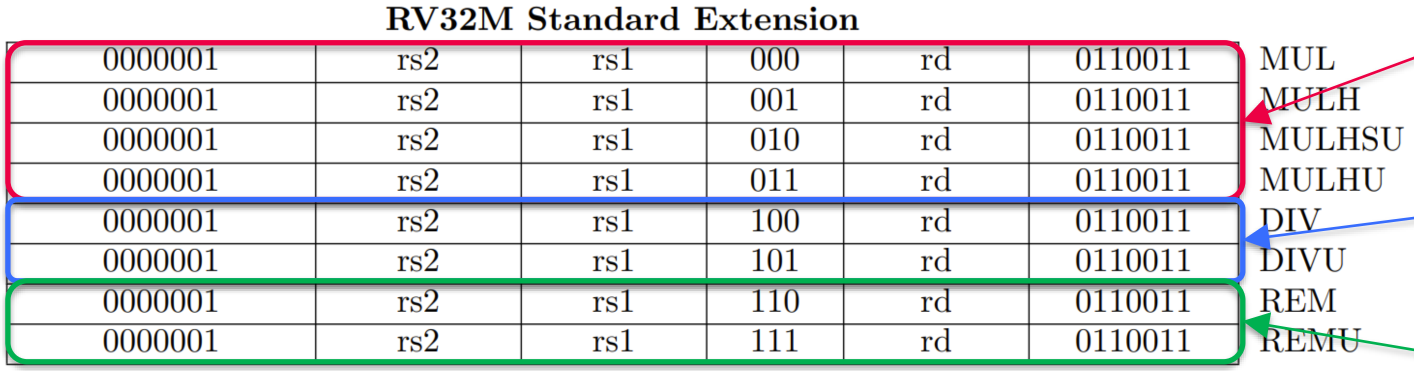
Load/store instructions
(transfers between registers and
main memory)

Arithmetic and logical operations

Special-purpose instructions

ISA spec, Vol. 1, unprivileged spec, p. 130

M extension



RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Multiplication

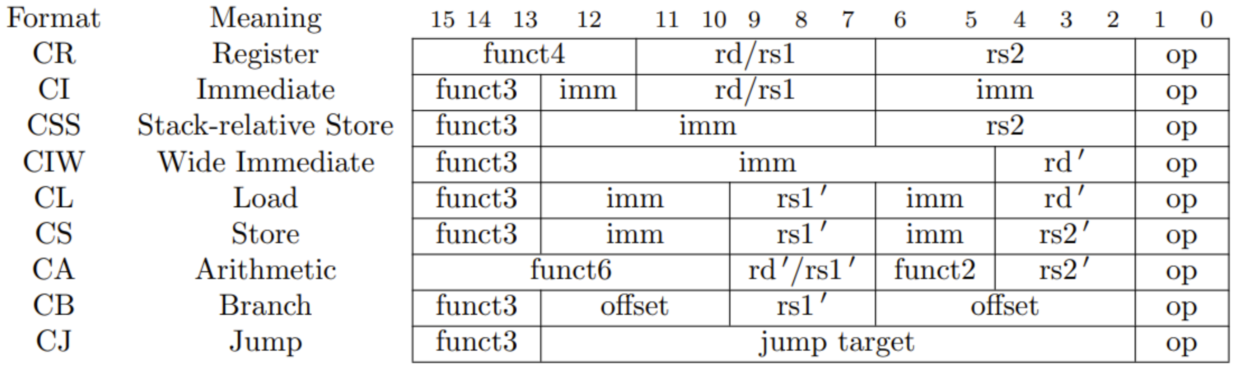
Division

Remainder of the division

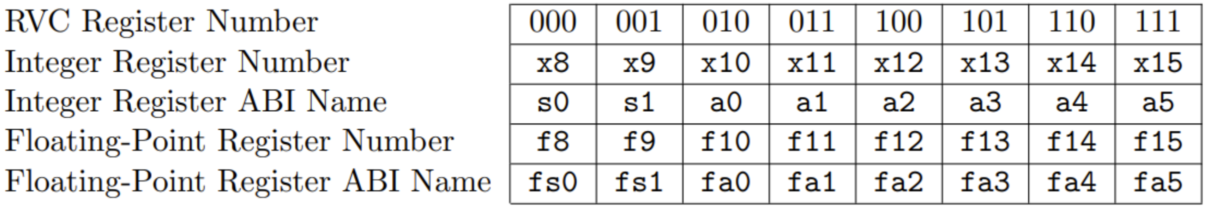
ISA spec, Vol. 1, unprivileged spec, p. 131

C extension

Shorter (16-bit) instructions that can replace ~50% traditional instructions
(resulting in 25–30% total code size reduction)



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	
CI	Immediate	funct3			imm	rd/rs1					imm					op	
CSS	Stack-relative Store	funct3			imm						rs2					op	
CIW	Wide Immediate	funct3			imm								rd′			op	
CL	Load	funct3			imm			rs1′			imm		rd′			op	
CS	Store	funct3			imm			rs1′			imm		rs2′			op	
CA	Arithmetic	funct6						rd′/rs1′			funct2		rs2′			op	
CB	Branch	funct3			offset			rs1′			offset					op	
CJ	Jump	funct3			jump target											op	

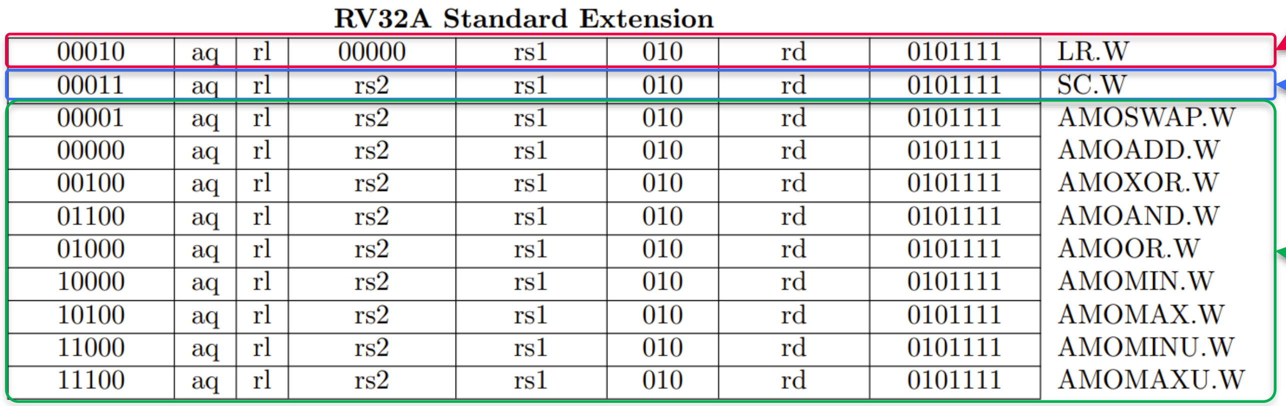


RVC Register Number	000	001	010	011	100	101	110	111
Integer Register Number	x8	x9	x10	x11	x12	x13	x14	x15
Integer Register ABI Name	s0	s1	a0	a1	a2	a3	a4	a5
Floating-Point Register Number	f8	f9	f10	f11	f12	f13	f14	f15
Floating-Point Register ABI Name	fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

ISA spec, Vol. 1, unprivileged spec, p. 100

A extension

Atomic read/modify/write operation to ***avoid race conditions*** between multiple threads/core



RV32A Standard Extension

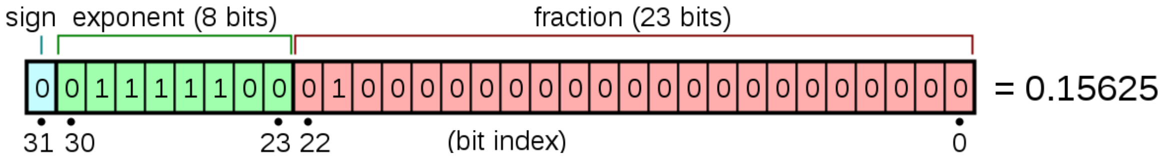
00010	aq	rl	00000	rs1	010	rd	0101111	LR.W
00011	aq	rl	rs2	rs1	010	rd	0101111	SC.W
00001	aq	rl	rs2	rs1	010	rd	0101111	AMOSWAP.W
00000	aq	rl	rs2	rs1	010	rd	0101111	AMOADD.W
00100	aq	rl	rs2	rs1	010	rd	0101111	AMOXOR.W
01100	aq	rl	rs2	rs1	010	rd	0101111	AMOAND.W
01000	aq	rl	rs2	rs1	010	rd	0101111	AMOOR.W
10000	aq	rl	rs2	rs1	010	rd	0101111	AMOMIN.W
10100	aq	rl	rs2	rs1	010	rd	0101111	AMOMAX.W
11000	aq	rl	rs2	rs1	010	rd	0101111	AMOMINU.W
11100	aq	rl	rs2	rs1	010	rd	0101111	AMOMAXU.W

"Load-reserved" – load and reserve exclusive rights to the word

"Store-conditional" – store is having rights

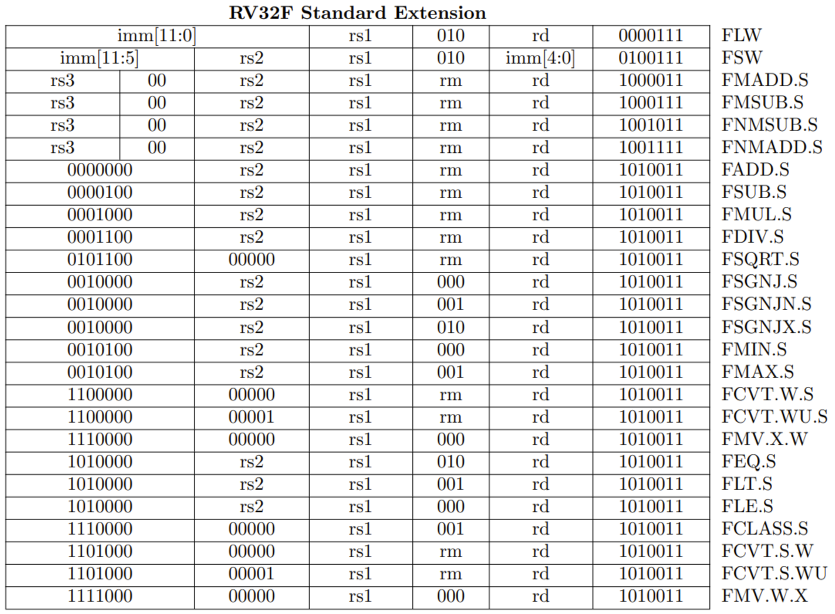
Atomic read/modify/write operations

ISA spec, Vol. 1, unprivileged spec, p. 132



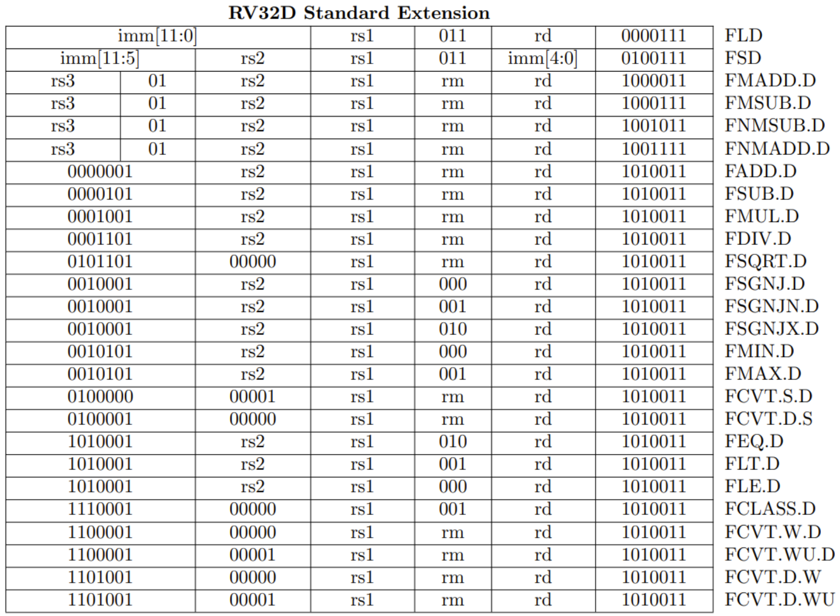
F and D extensions

- FLx/FSx – load/store to floating-point registers
- FADD, FSUB, FMUL, FMIN, FMAX Arithmetic functions, input and output are float
- FSGNJ, FSGNJN, FSGNJX Sign-injection instructions, input and output are float
- FEQ, FLT, FLE, FCLASS Comparison operations, input is float, output is integer
- FCVT.W.S, FCVT.S.W, FCVT.WU.S, FCVT.S.WU Transfer operations from float to integer and vice versa.
- FMADD, FMSUB, FNMSUB, FNMADD Floating-point fused multiply-add instructions, input and output are float



RV32F Standard Extension

imm[11:0]			rs1	010	rd	0000111	FLW
imm[11:5]		rs2	rs1	010	imm[4:0]	0100111	FSW
rs3	00	rs2	rs1	rm	rd	1000011	FMADD.S
rs3	00	rs2	rs1	rm	rd	1000111	FMSUB.S
rs3	00	rs2	rs1	rm	rd	1001011	FNMSUB.S
rs3	00	rs2	rs1	rm	rd	1001111	FNMADD.S
0000000		rs2	rs1	rm	rd	1010011	FADD.S
0000100		rs2	rs1	rm	rd	1010011	FSUB.S
0001000		rs2	rs1	rm	rd	1010011	FMUL.S
0001100		rs2	rs1	rm	rd	1010011	FDIV.S
0101100		00000	rs1	rm	rd	1010011	FSQRT.S
0010000		rs2	rs1	000	rd	1010011	FSGNJ.S
0010000		rs2	rs1	001	rd	1010011	FSGNJN.S
0010000		rs2	rs1	010	rd	1010011	FSGNJX.S
0010100		rs2	rs1	000	rd	1010011	FMIN.S
0010100		rs2	rs1	001	rd	1010011	FMAX.S
1100000		00000	rs1	rm	rd	1010011	FCVT.W.S
1100000		00001	rs1	rm	rd	1010011	FCVT.WU.S
1110000		00000	rs1	000	rd	1010011	FMV.X.W
1010000		rs2	rs1	010	rd	1010011	FEQ.S
1010000		rs2	rs1	001	rd	1010011	FLT.S
1010000		rs2	rs1	000	rd	1010011	FLE.S
1110000		00000	rs1	001	rd	1010011	FCLASS.S
1101000		00000	rs1	rm	rd	1010011	FCVT.S.W
1101000		00001	rs1	rm	rd	1010011	FCVT.S.WU
1111000		00000	rs1	000	rd	1010011	FMV.W.X

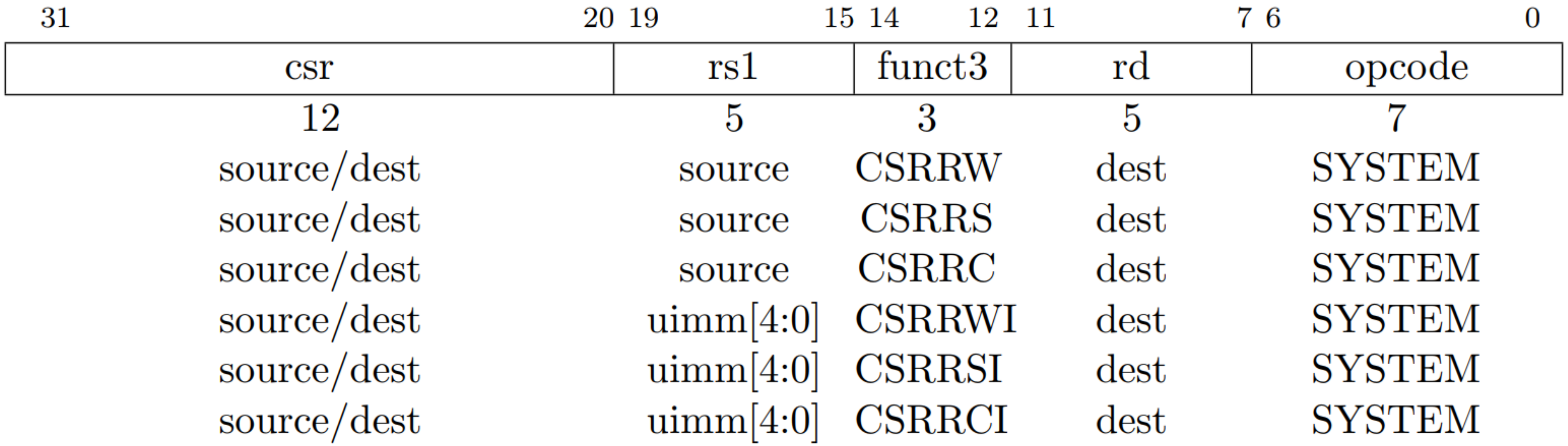


RV32D Standard Extension

imm[11:0]			rs1	011	rd	0000111	FLD
imm[11:5]		rs2	rs1	011	imm[4:0]	0100111	FSD
rs3	01	rs2	rs1	rm	rd	1000011	FMADD.D
rs3	01	rs2	rs1	rm	rd	1000111	FMSUB.D
rs3	01	rs2	rs1	rm	rd	1001011	FNMSUB.D
rs3	01	rs2	rs1	rm	rd	1001111	FNMADD.D
0000001		rs2	rs1	rm	rd	1010011	FADD.D
0000101		rs2	rs1	rm	rd	1010011	FSUB.D
0001001		rs2	rs1	rm	rd	1010011	FMUL.D
0001101		rs2	rs1	rm	rd	1010011	FDIV.D
0101101		00000	rs1	rm	rd	1010011	FSQRT.D
0010001		rs2	rs1	000	rd	1010011	FSGNJ.D
0010001		rs2	rs1	001	rd	1010011	FSGNJN.D
0010001		rs2	rs1	010	rd	1010011	FSGNJX.D
0010101		rs2	rs1	000	rd	1010011	FMIN.D
0010101		rs2	rs1	001	rd	1010011	FMAX.D
0100000		00001	rs1	rm	rd	1010011	FCVT.S.D
0100001		00000	rs1	rm	rd	1010011	FCVT.D.S
1010001		rs2	rs1	010	rd	1010011	FEQ.D
1010001		rs2	rs1	001	rd	1010011	FLT.D
1010001		rs2	rs1	000	rd	1010011	FLE.D
1110001		00000	rs1	001	rd	1010011	FCLASS.D
1100001		00000	rs1	rm	rd	1010011	FCVT.W.D
1100001		00001	rs1	rm	rd	1010011	FCVT.WU.D
1101001		00000	rs1	rm	rd	1010011	FCVT.D.W
1101001		00001	rs1	rm	rd	1010011	FCVT.D.WU

ISA spec, Vol. 1, unprivileged spec, p. 133-134

Zicsr extension



31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

ISA spec, Vol. 1, unprivileged spec, p. 55-58

Binary and mnemonic representation of instructions

Mnemonic of instructions has unique binary representation (and vice versa in most cases)

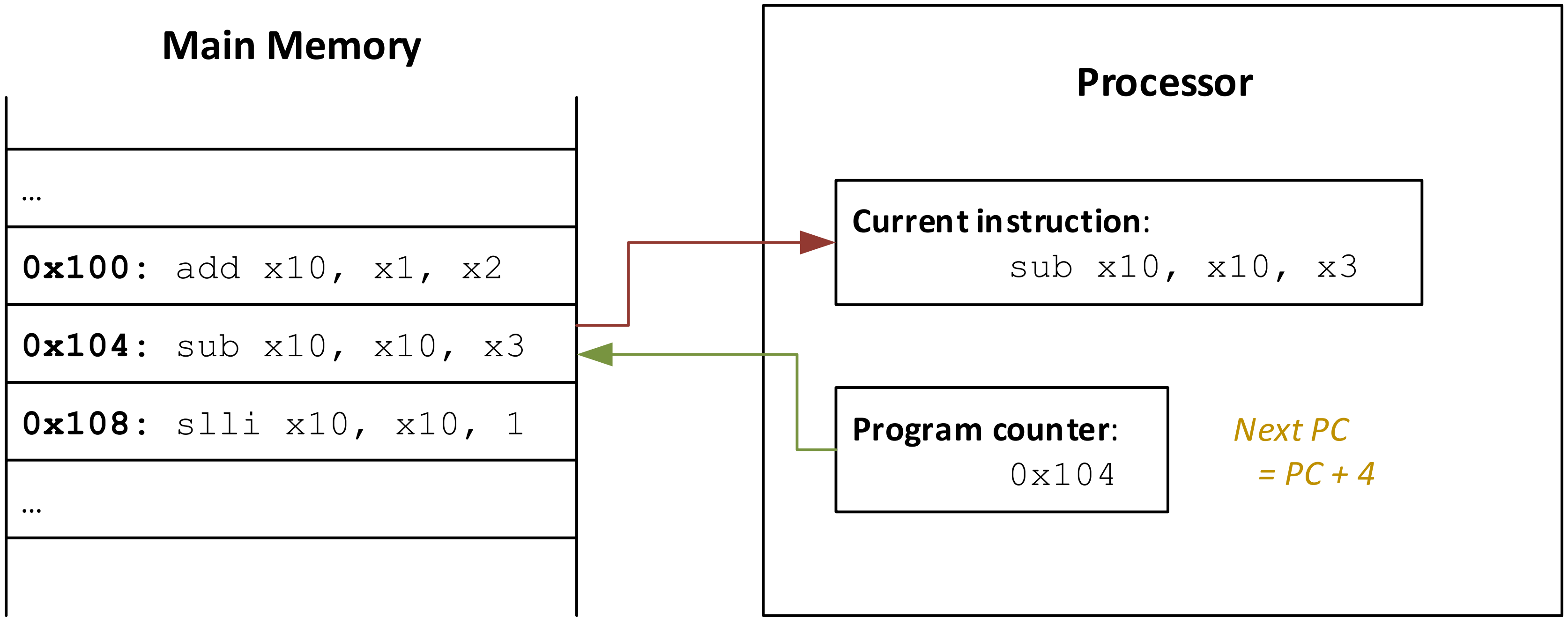
Mnemonic -> binary:

1) Find instruction type and format
2) Fill in the instruction fields

Binary -> mnemonic:

1) Separate instruction opcode
2) Reveal instruction type
3) Separate individual fields

Example: linear code block



Main Memory

...

`0x100: add x10, x1, x2`

`0x104: sub x10, x10, x3`

`0x108: slli x10, x10, 1`

...

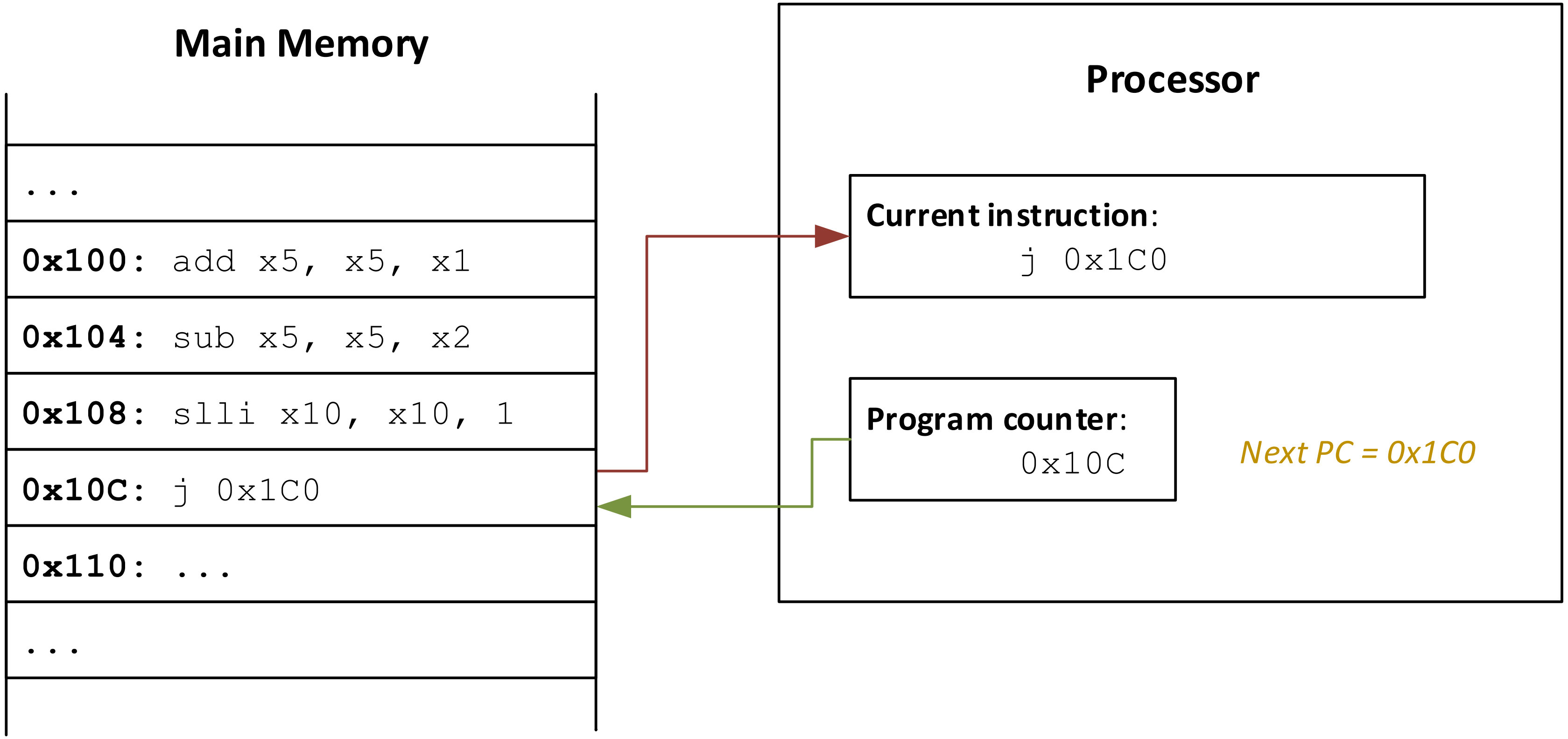
Processor

Current instruction:
`sub x10, x10, x3`

Program counter:
`0x104`

Next PC = PC + 4

Example: unconditional branch (infinite loop)



Main Memory

```
...
0x100: add x5, x5, x1
0x104: sub x5, x5, x2
0x108: slli x10, x10, 1
0x10C: j 0x1C0
0x110: ...
...
```

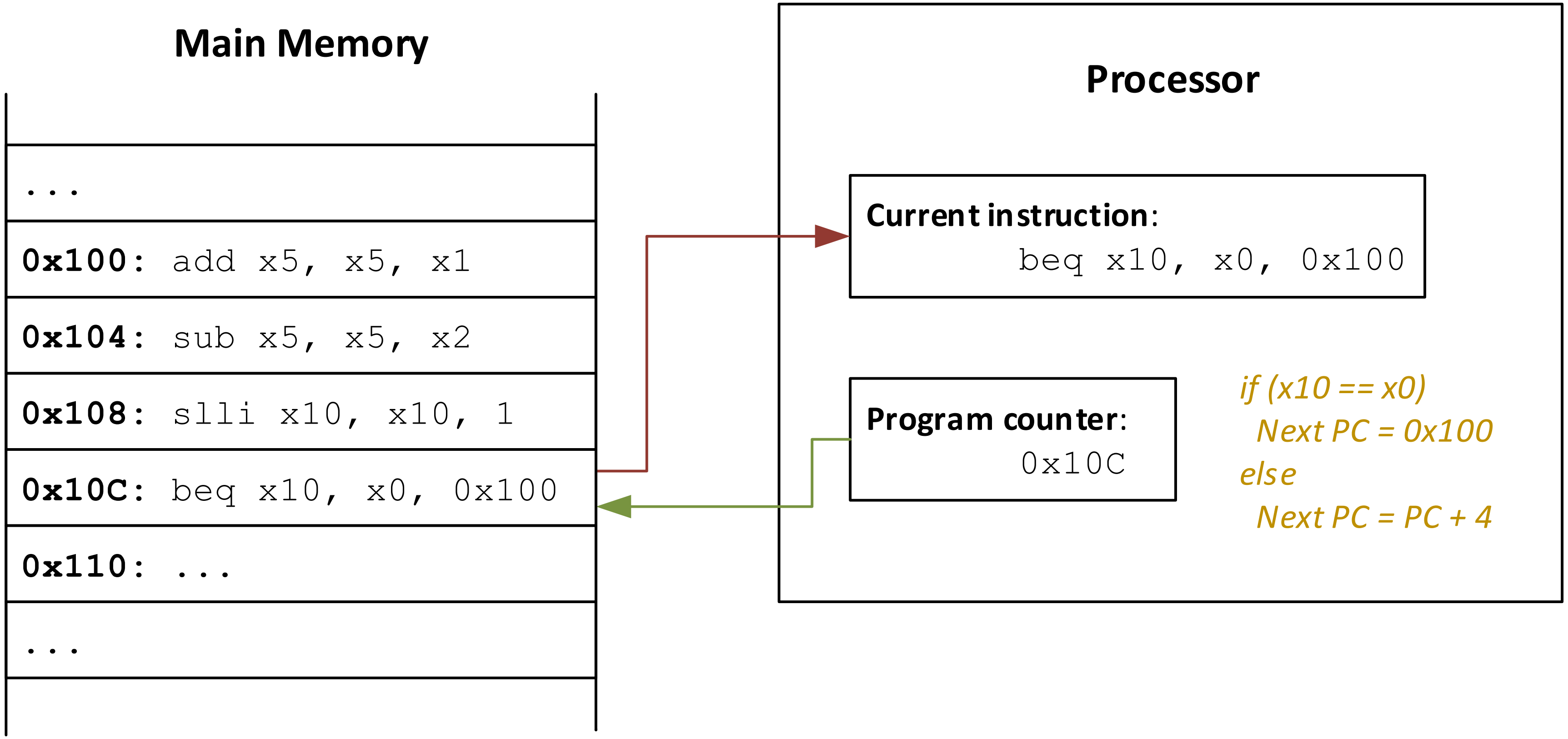
Processor

Current instruction: `j 0x1C0`

Program counter: `0x10C`

Next PC = 0x1C0

Example: conditional branch



Main Memory

```
...
0x100: add x5, x5, x1
0x104: sub x5, x5, x2
0x108: slli x10, x10, 1
0x10C: beq x10, x0, 0x100
0x110: ...
...
```

Processor

Current instruction:
`beq x10, x0, 0x100`

Program counter:
`0x10C`

```
if (x10 == x0)
  Next PC = 0x100
else
  Next PC = PC + 4
```

Thank you for the lesson!

Alexander Antonov, Assoc. Prof., antonov@itmo.ru

Hangzhou, 2025