RISC-V Instruction Encoding - 1

Caiwen Ding
Department of Computer Science and Engineering
University of Connecticut

CSE3666: Introduction to Computer Architecture

Outline

- RISC-V Instruction encoding
  - Instruction format
  - Encoding of instructions like ADD, ADDI, and Load/store
  - Decoding

Design Principle 4*:* Good design demands good compromises

Keep formats as similar as possible

Reading: Section 2.5 and the beginning of Section 2.10.

References: Reference card in the book.

Representing instructions with bits

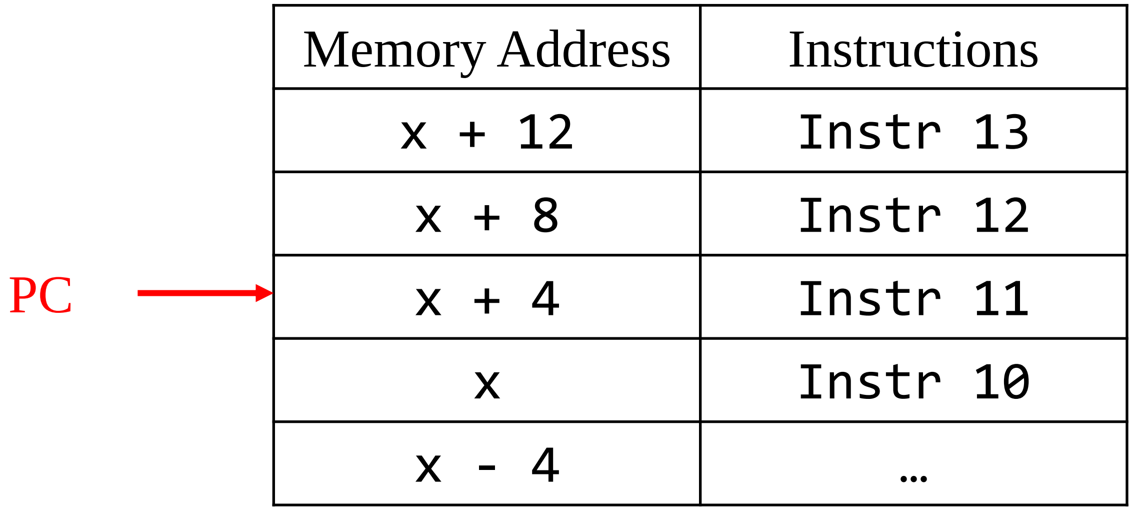
- We use bits to represent numbers, characters, etc.
- We also use bits to represent instructions

Design questions:

- How many bits should we use to encode instructions?
- Are we using the same number of bits to encode all instructions?
  - Do all instructions have the same length?

RISC-V instruction words

- RISC-V base ISA are encoded as 32-bit instruction words
  - Encoded instructions are also called machine (language) code
- Both instructions and data are stored in memory
- Program Counter (PC) points to the current instruction
  - Incremented by 4 in normal flow for sequential execution



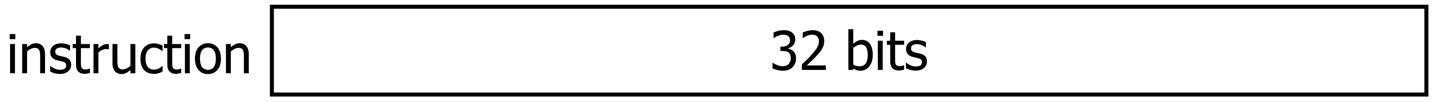
Memory Address	Instructions
`x + 12`	`Instr 13`
`x + 8`	`Instr 12`
`x + 4`	`Instr 11`
`x`	`Instr 10`
`x - 4`	…

C extension allows compressed instructions of 16 bits, but it is not a stand-alone ISA.
Machine language uses binary representation of instructions.
Instructions in machine language are called machine code.

Discussion

- What information do you want to keep in the instruction word?

```
ADD         rd, rs1, rs2
ADDI        rd, rs1, immd
LW          rd, offset(rs1)
SW          rs2, offset(rs1)
```



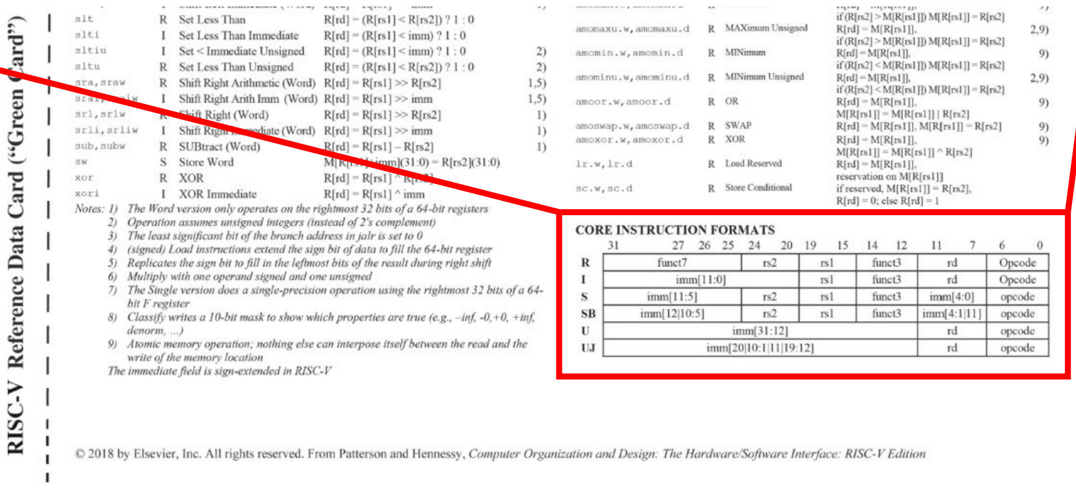
Instruction format

- The layout of bits in instruction words is instruction format
  - How do we use 32-bit bits to specify operation code (opcode), registers, immediate, offset, etc? How many bits for each?
- RISC-V has six instruction formats (while MIPS has 3)
  - R, I, S, SB, U, and UJ
- Instructions we have learned so far
  - Using registers only
  - Having an immediate as the second operand
  - Load and store
  - LUI
  - Branch, to be discussed next week

Binary compatibility allows compiled programs to work on different computers

RISC-V Core Instruction Format

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	Opcode
I	imm[11:0]		rs1	funct3	rd	Opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode



RISC-V Reference Data Card (“Green Card”)

slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra,sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai,sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl,srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli,srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

Notes: 1) *The Word version only operates on the rightmost 32 bits of a 64-bit registers*
2) *Operation assumes unsigned integers (instead of 2's complement)*
3) *The least significant bit of the branch address in jalr is set to 0*
4) *(signed) Load instructions extend the sign bit of data to fill the 64-bit register*
5) *Replicates the sign bit to fill in the leftmost bits of the result during right shift*
6) *Multiply with one operand signed and one unsigned*
7) *The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register*
8) *Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0,+0, +inf, denorm, ...)*
9) *Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location*

The immediate field is sign-extended in RISC-V

amomaxu.w,amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w,amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w,amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w,amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w,amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w,amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w,lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	Opcode
I	imm[11:0]		rs1	funct3	rd	Opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

The (green) card in the textbook

imm is the 32-bit immediate processor uses for computation.
It is not the immediate written in instruction.

RISC-V R-type Instructions

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- For instructions that have 3 registers as operands
- Fields in R-type
  - opcode: 7-bit operation code
  - rd: destination register number
  - rs1: first source register number
  - rs2: second source register number
  - funct3: additional function code
  - funct7: even more function code

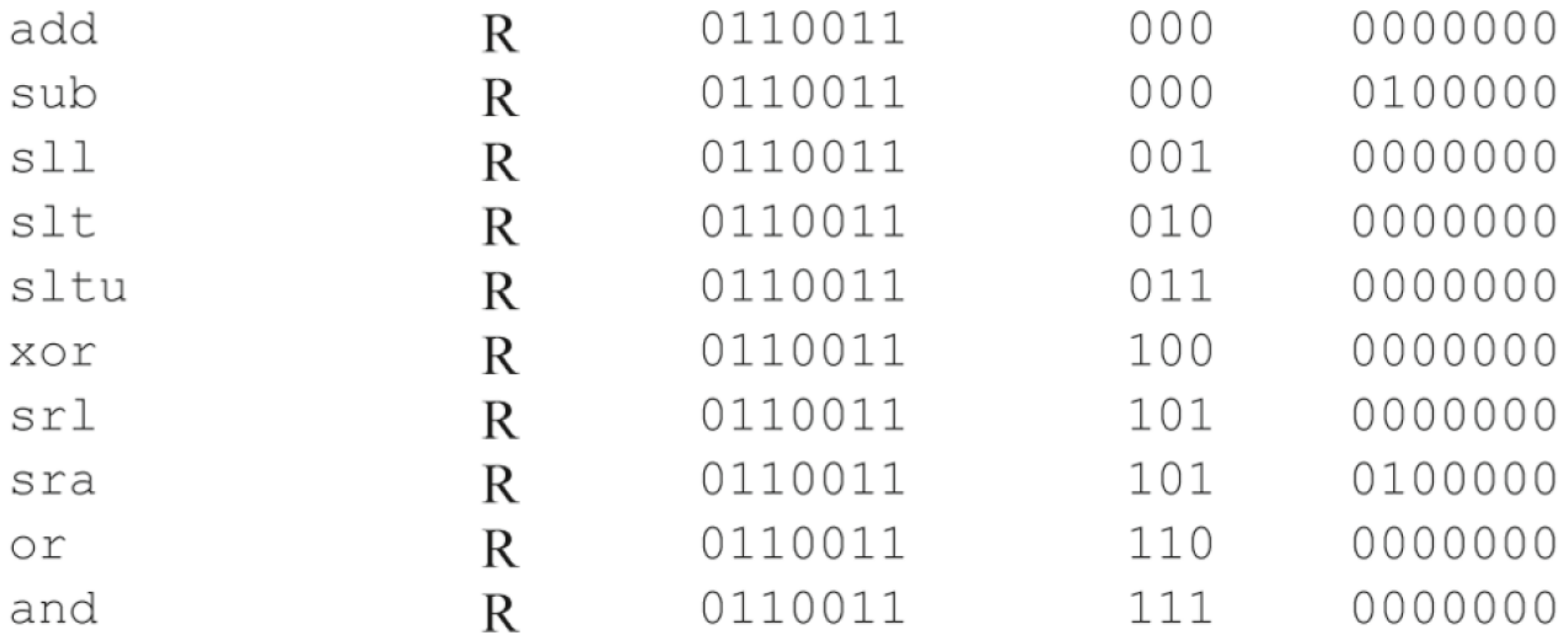
R-format Example: ADD

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

```
add x1, x2, x3
```


Opcode and funct codes

- From the green card, which also has hexadecimal representation



	type	opcode	funct3	funct7
add	R	0110011	000	0000000
sub	R	0110011	000	0100000
sll	R	0110011	001	0000000
slt	R	0110011	010	0000000
sltu	R	0110011	011	0000000
xor	R	0110011	100	0000000
srl	R	0110011	101	0000000
sra	R	0110011	101	0100000
or	R	0110011	110	0000000
and	R	0110011	111	0000000

The right most two bits in opcode are always 11
for 32-bit instructions

R-type Example: ADD

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

```
add x1, x2, x3
```

0	3	2	0	1	0x33

000 0000	00011	00010	000	00001	011 0011

0x 003100B3

What if we change ADD to SUB? How many bits are changed?

Question (from textbook)

- What RISC-V instruction does this represent? Choose from one of the four options below?
  - The numbers are in decimal

funct7	rs2	rs1	funct3	rd	opcode
32	9	10	0	11	51

```
A. sub  x9, x10, x11
B. add  x11, x9, x10
C. sub  x11, x10, x9
D. sub  x11, x9, x10
```

	Funct7
ADD	0b 000 0000
SUB	0b 010 0000

RISC-V R-type Instructions

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- We can use the format to encode any instructions like

```
instr_name      rd, rs1, rs2
```

- How about instructions like addi, slli?

```
addi      rd, rs1, imm
```

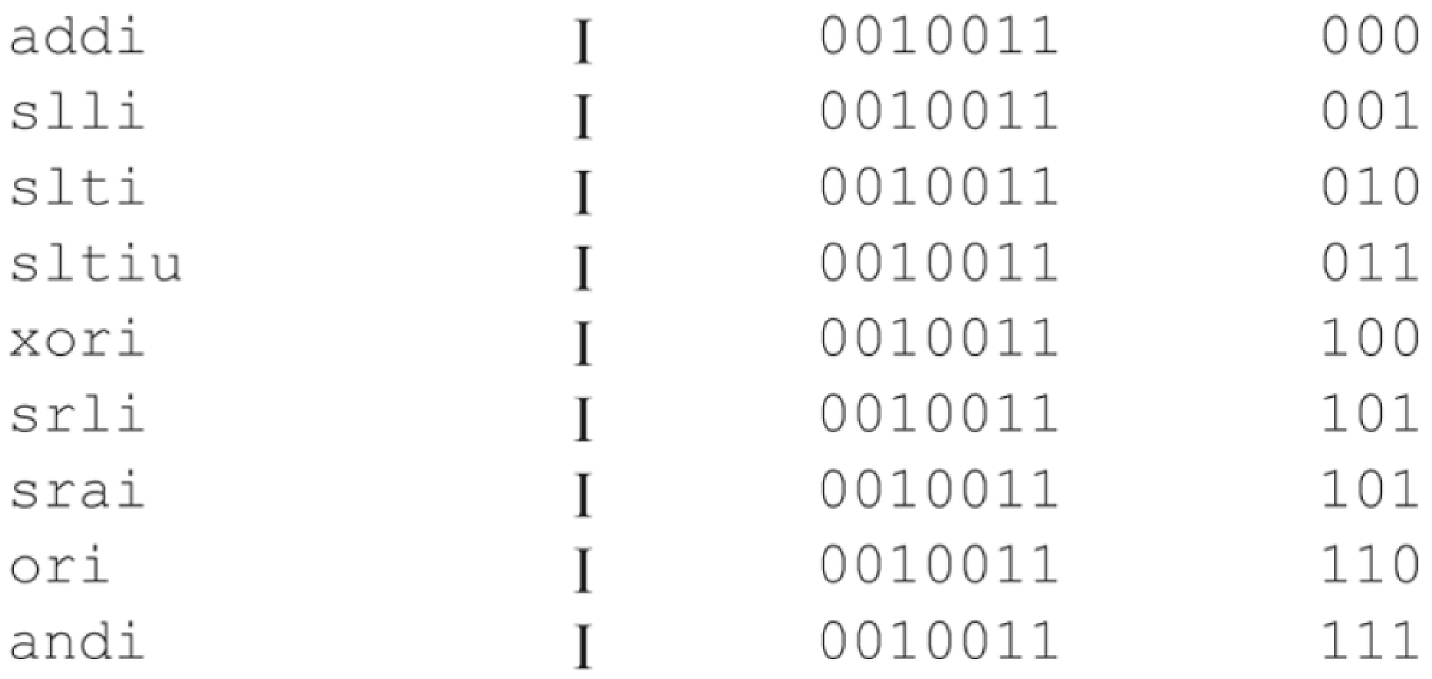
RISC-V I-type Instructions

imm[11:0]	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

- Fields in I-type
  - opcode: operation code
  - rd: destination register number
  - rs1: first source register number
  - funct3: additional function code
  - imm: lower 12 bits of the immediate, in the place of funct7 and rs2

- Since only 12 bits are kept in machine code,
  the immediate must be in $[-2^{11}, +2^{11} - 1]$ or $[-2048, 2047]$

Example of I-type opcode and funct3 code



	type	opcode	funct3
addi	I	0010011	000
slli	I	0010011	001
slti	I	0010011	010
sltiu	I	0010011	011
xori	I	0010011	100
srli	I	0010011	101
srai	I	0010011	101
ori	I	0010011	110
andi	I	0010011	111

In slli, srli, and srai, only lower 5 bits of the immediate are used for shift amount.
5 bits are enough for 32-bit registers!

I-type Example: ADDI

imm[11:0]	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

```
addi x1, x2, 32
```


I-type Example: ADDI

imm[11:0]	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

```
addi x1, x2, 32
```

0000 0010 0000	00010	000	00001	0010011

When executing the instruction, processor builds a 32-bit immediate imm

	imm[31:12] sign	imm[11:0]
imm	0000 0000 0000 0000 0000	0000 0010 0000

I-type Example: SRLI vs SRAI

imm[11:0]	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

```
srli x1, x2, 16
```

0000 0001 0000	00010	101	00001	0010011

Same opcode and funct3
Bit 30 is different!

```
srai x1, x2, 16
```

0100 0001 0000	00010	101	00001	0010011

srli	I	0010011	101	0000000
srai	I	0010011	101	0100000

funct7

What format would you use for load instructions?

- Load instructions

```
    lw  rd, offset(rs1)
```

A. R-format
B. I-format

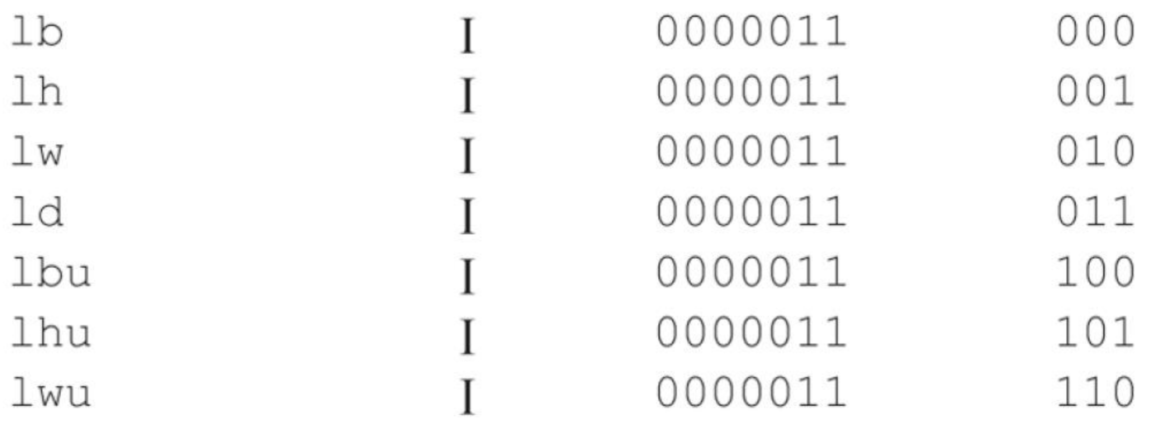
R:	funct7	rs2	rs1	funct3	rd	opcode
I:	imm [11:0]		rs1	funct3	rd	opcode

Loads are I-type

imm[11:0]	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

```
lw  x1, 32(x2)
```

0000 0010 0000	00010	010	00001	000 0011



lb	I	0000011	000
lh	I	0000011	001
lw	I	0000011	010
ld	I	0000011	011
lbu	I	0000011	100
lhu	I	0000011	101
lwu	I	0000011	110

Do you see the pattern in funct3?

How about store?

How about store instructions?

- Store instructions have rs2, but not rd

```
sw  rs2,offset(rs1)
```

R:	funct7	rs2	rs1	funct3	rd	opcode
I:	imm [11:0]		rs1	funct3	rd	opcode

RISC-V S-type Instructions

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Fields in S-type
  - opcode: operation code
  - rs1: first source register number
  - rs2: second source register number
  - imm[11:5] and imm[4:0]:

    The lower 12 bits of the offset/immediate are stored into two fields.

    The higher 7 bits, bits 11 to 5, are in funct7

    The lower 5 bits, bits 4 to 0, are in rd

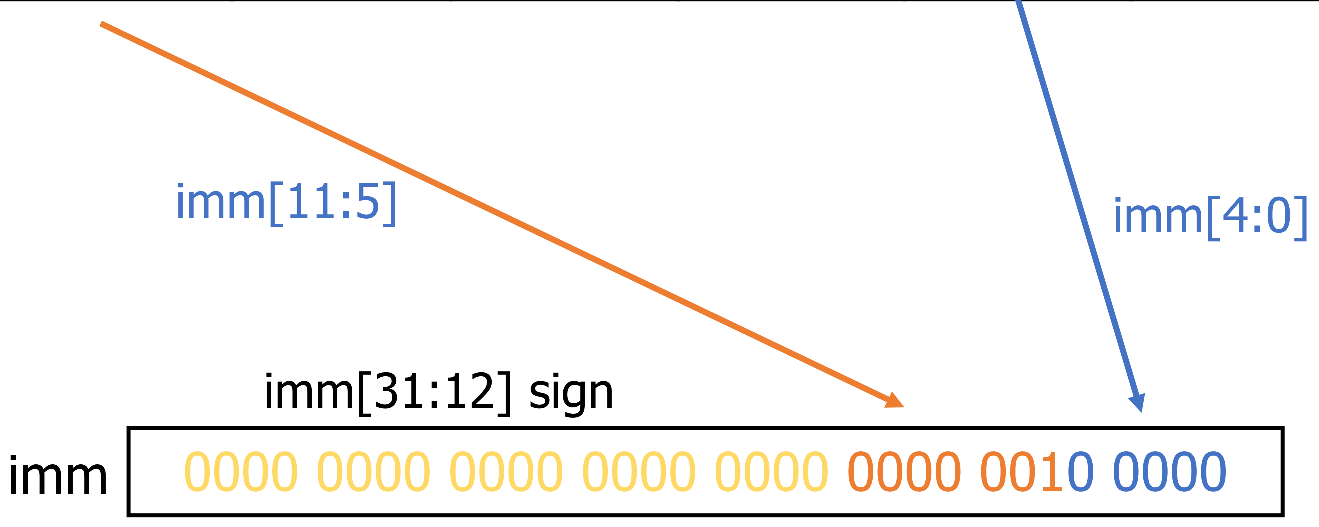
Stores are S-type

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

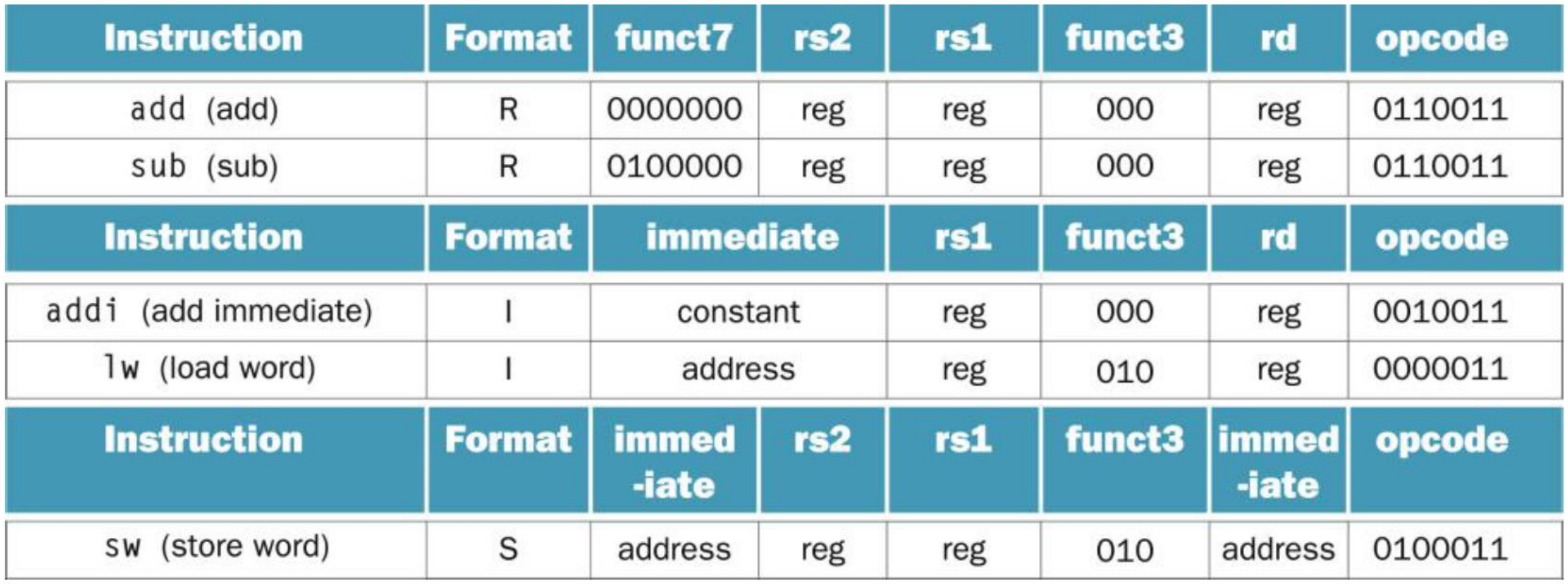
```
sw  x1, 32(x2)
```

The lower 12 bits of the immediate are saved in two fi

0000 00001	00001	00010	010	00000	010 0011



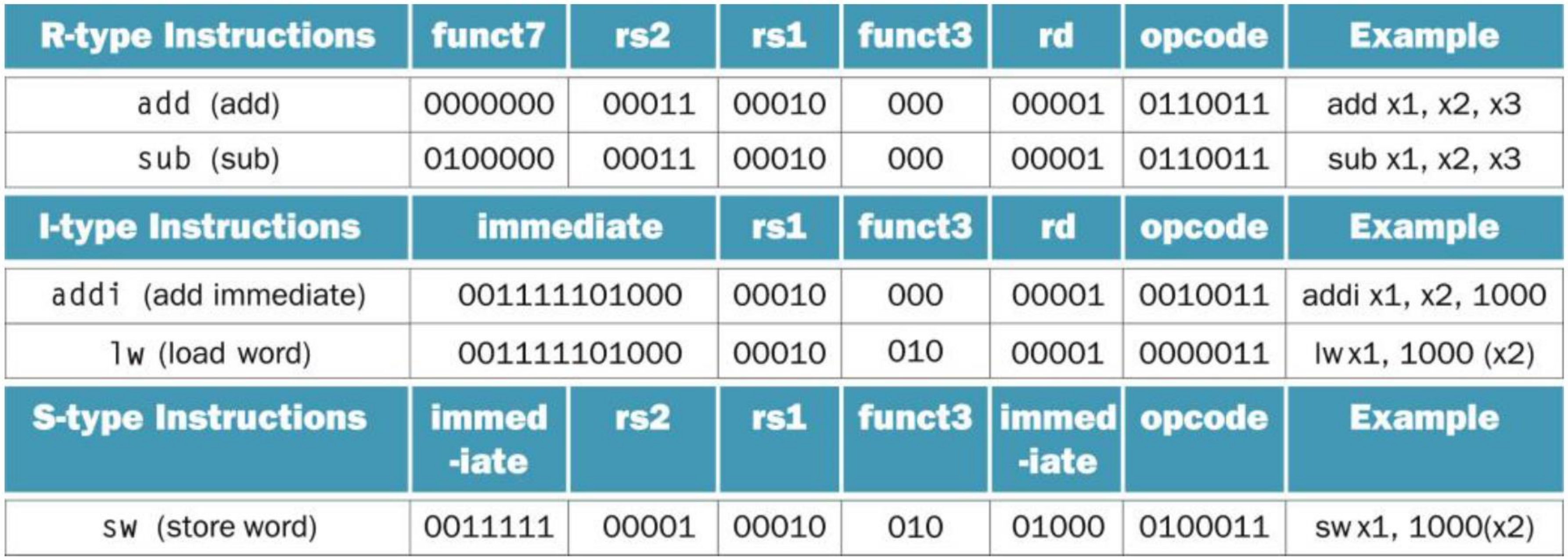
Summary of R-, I-, and S-type instructions



Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
`add` (add)	R	0000000	reg	reg	000	reg	0110011
`sub` (sub)	R	0100000	reg	reg	000	reg	0110011
Instruction	**Format**	**immediate**		**rs1**	**funct3**	**rd**	**opcode**
`addi` (add immediate)	I	constant		reg	000	reg	0010011
`lw` (load word)	I	address		reg	010	reg	0000011
Instruction	**Format**	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**
`sw` (store word)	S	address	reg	reg	010	address	0100011

Fields, other than immediate fields, are located at the same location, for all types
Placement of bits in the immediate is more complicated

Examples of R-, I-, and S-type instructions



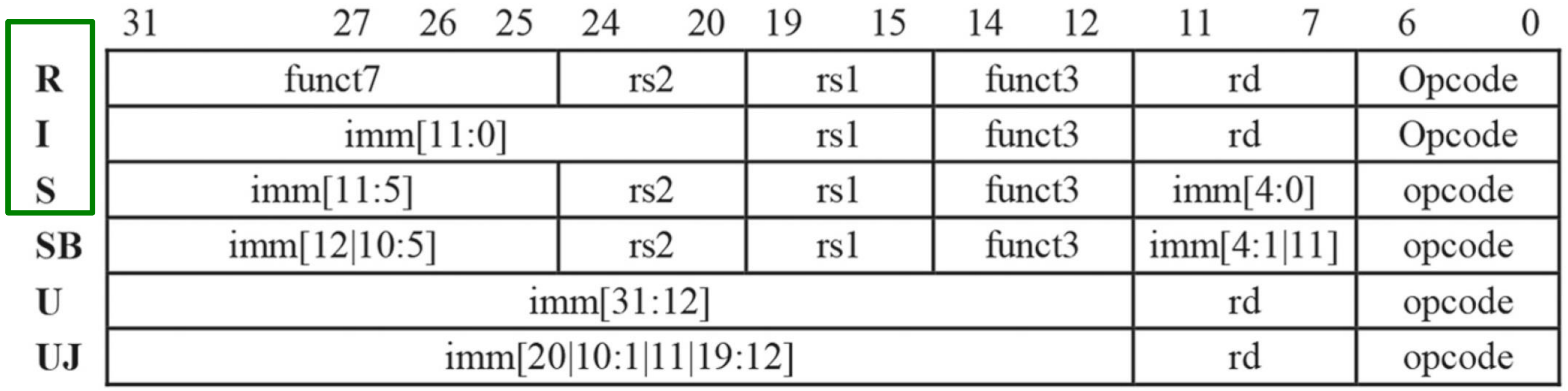
R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (add)	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
sub (sub)	0100000	00011	00010	000	00001	0110011	sub x1, x2, x3

I-type Instructions	immediate	rs1	funct3	rd	opcode	Example
addi (add immediate)	001111101000	00010	000	00001	0010011	addi x1, x2, 1000
lw (load word)	001111101000	00010	010	00001	0000011	lw x1, 1000 (x2)

S-type Instructions	immed -iate	rs2	rs1	funct3	immed -iate	opcode	Example
sw (store word)	0011111	00001	00010	010	01000	0100011	sw x1, 1000(x2)

RISC-V Core Instruction Format

- Now we know four types: R, I, S, and U
- We will discuss SB, and UJ later



	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	Opcode
I	imm[11:0]		rs1	funct3	rd	Opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

Only real instructions can be encoded into machine code.

Exercise

- Pick an instruction and encode it

- Study the machine code generated by RARS

Questions

- RISC-V has three fields for specifying operations: opcode, funct3, and funct7. Why don’t they combine them and have a single opcode field of 17 bits?

- If someone decides to increase the number of registers to 64, how does it affect the encoding of instructions?