Computer Architecture

Ngo Lam Trung, Pham Ngoc Hung, Hoang Van Hiep
Department of Computer Engineering
School of Information and Communication Technology (SoICT)
Hanoi University of Science and Technology
E-mail: [trungnl, hungpn, hiephv]@soict.hust.edu.vn

Chapter 3: Instruction Set Architecture

(Language of the Computer)

[with materials from *COD, RISC-V 2nd Edition*, Patterson & Hennessy 2021,
M.J. Irwin's presentation, PSU 2008,
The RISC-V Instruction Set Manual, Volume I, ver. 2.2]

Content

- Introduction
- RISC-V Instruction Set Architecture
  - Operands
  - Instruction set (basic RV32I variant)
  - RISC-V instruction formats
  - Other RISC-V instructions
- Basic programming structures
  - Branch and loop
  - Procedure call
  - Array and string

What is RISC-V and its advantages (over ARM, x86)?

- Developed at UC Berkeley as open ISA (2010).

- Typical of many modern ISAs, which have a large share of embedded market.
  - RISC CPUs: Pioneers of Modern Computer Architecture Receive ACM A.M. Turing Award

- Now managed by the RISC-V Foundation/RISC-V International (https://riscv.org/, since 2015).

- "**RISC-V combines a modular technical approach with an open, royalty-free ISA — meaning that anyone, anywhere can benefit from the IP contributed and produced by RISC-V.**" - RISC-V International.

- "**RISC-V does not take a political position on behalf of any geography.**" - RISC-V International.

Computer language: hardware operation

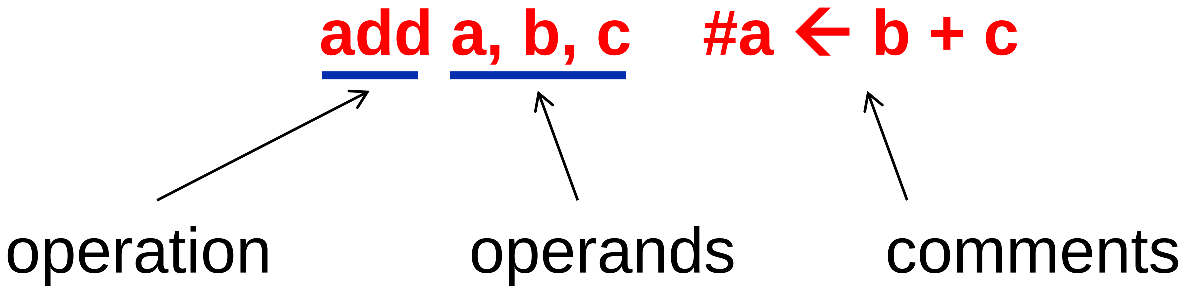
- Want to command the computer?

  ➔ You need to speak its language!!!

- Example: RISC-V assembly instruction

  add a, b, c   #a ← b + c

- Operation performed
  - add b and c,
  - then store result into a



add a, b, c   #a ← b + c

operation

operands

comments

Hardware operation

- What does the following code do?

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

- Equivalent C code

f = (g + h) – (i + j)

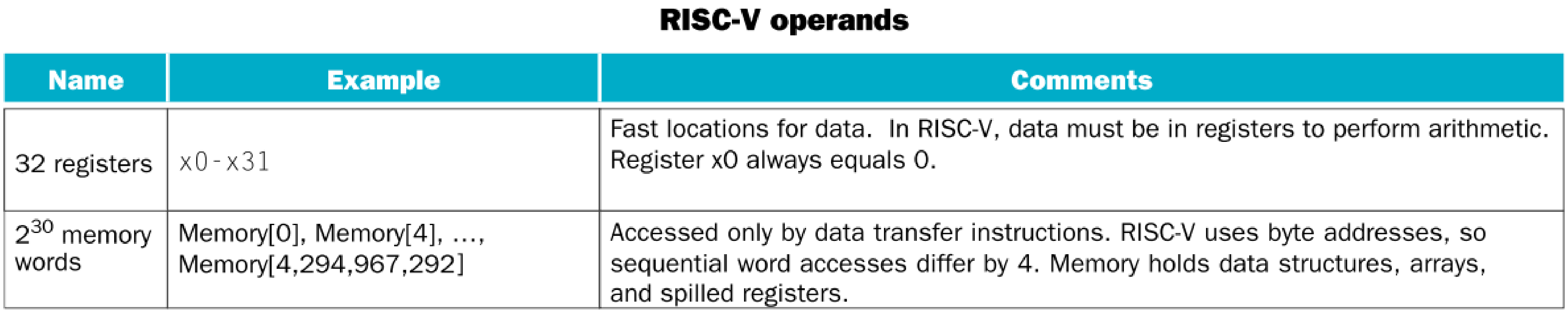
➔ ***Why not making 4- or 5-input instructions?***

Instruction format significantly influences hardware design.

➔ ***DP1: Simplicity favors regularity!***

Operands

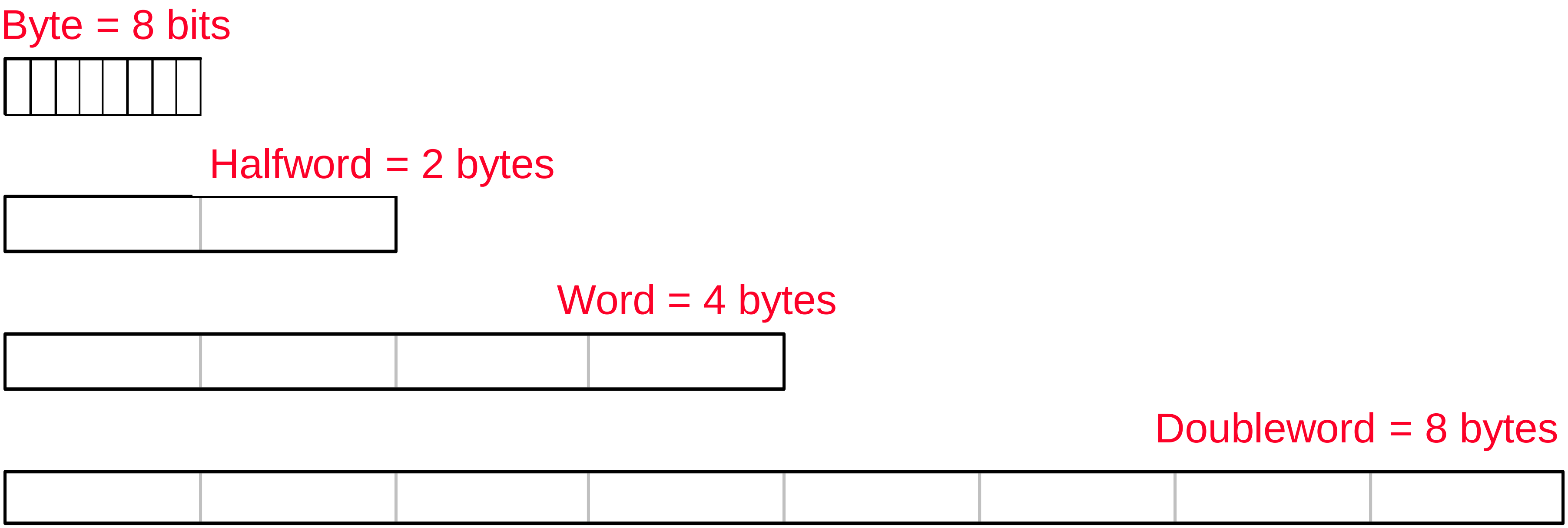
- Object of operation
  - Source operand: provides input data
  - Destination operand: stores the result of operation
- RISC-V operands
  - Registers
  - Memory
  - Constant/Immediate



RISC-V operands

Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4,294,967,292]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.

Data types in RISC-V



Byte = 8 bits

Halfword = 2 bytes

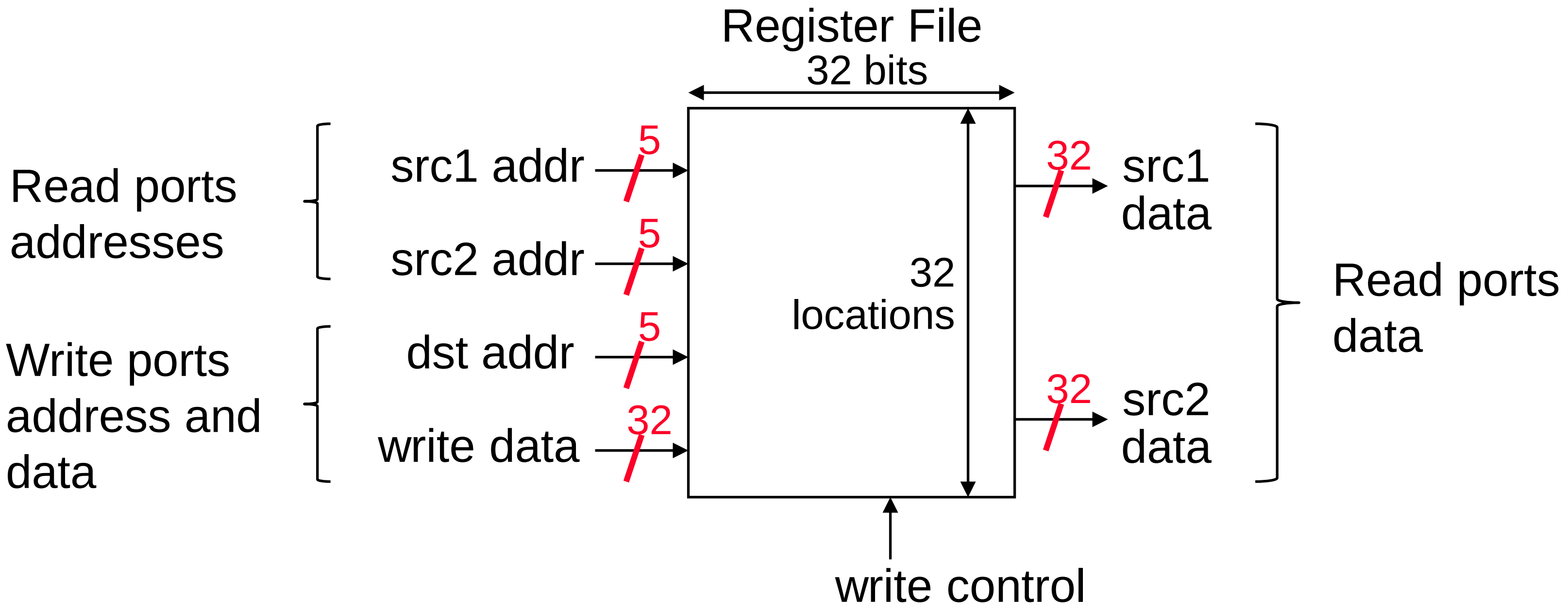
Word = 4 bytes

Doubleword = 8 bytes

RV32 registers hold 32-bit (4-byte) words. Other common data sizes include byte, halfword, and doubleword.

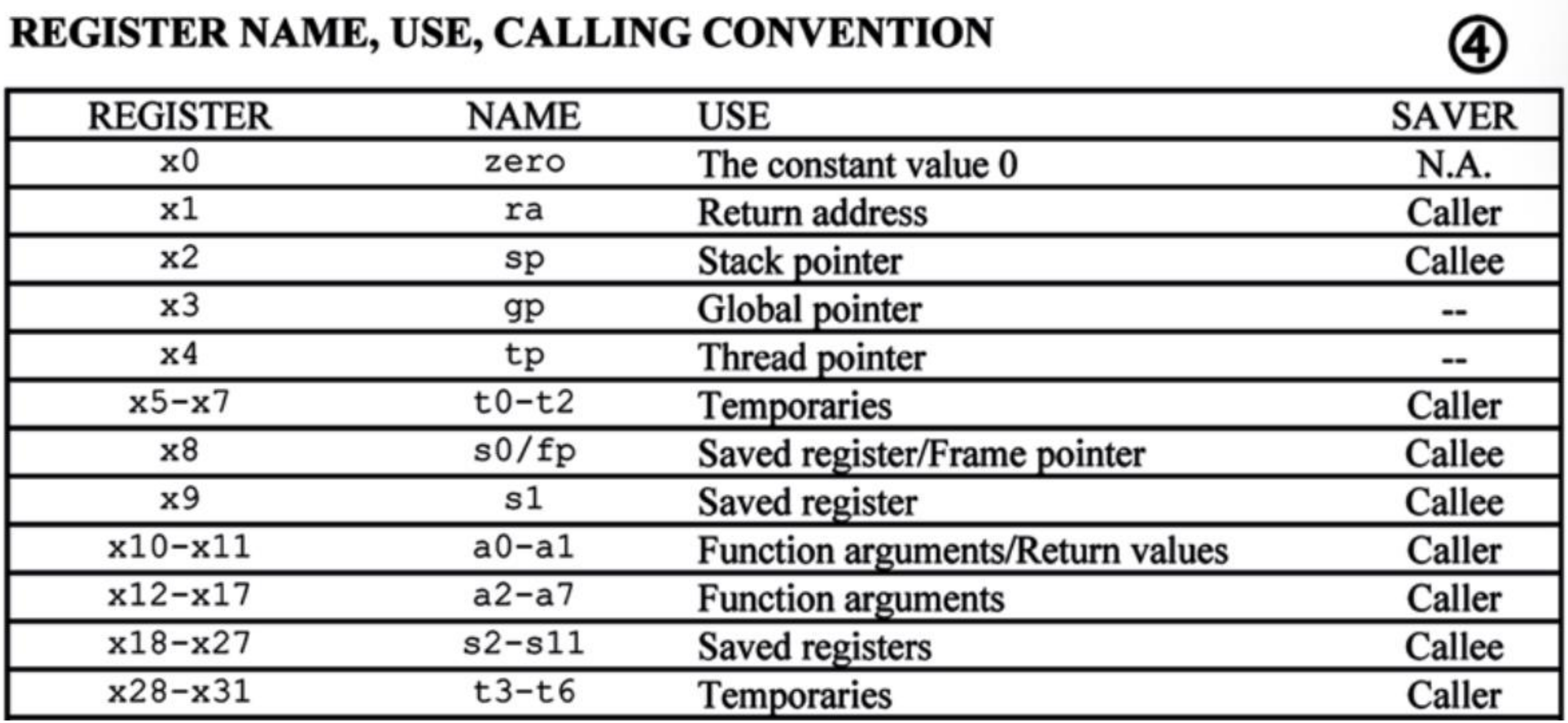
Register operand: RISC-V Register File

- Special memory inside CPU, called register file
- 32 slots, each slot is called a register (RV32I)
- Each register holds 32 bits of data
- Each register has a unique 5-bit address



RISC-V Register Convention

- RISC-V: load/store machine
- Data processing done on registers inside CPU



REGISTER NAME, USE, CALLING CONVENTION ④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

RISC-V integer registers

Register operand: RISC-V Register File

- ❑ Register file: “work place” right inside CPU.
- ❑ Larger register file should be better, more flexibility for CPU operation.
- ❑ Moore’s law: doubled number of transistor every 18 mo.
- ❑ Why only 32 registers, not more?

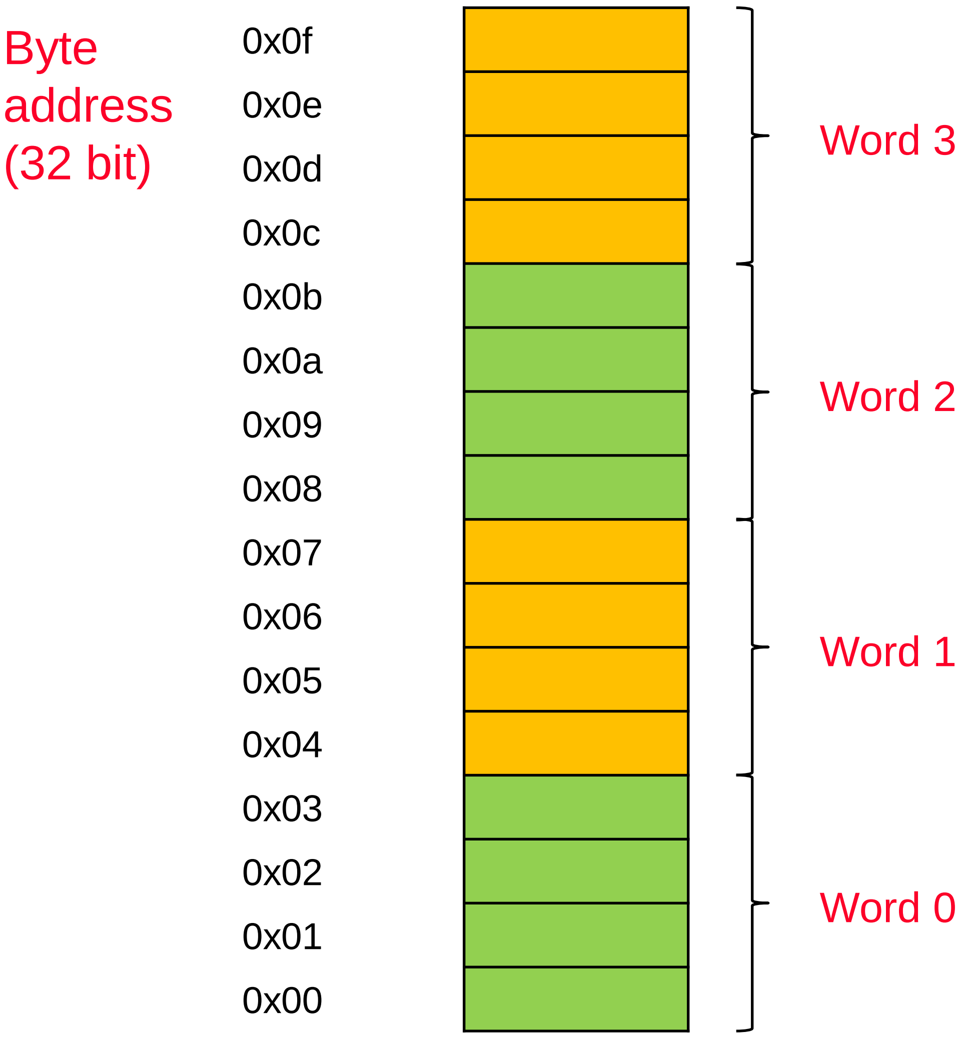
➔ ***DP2: Smaller is faster!***

Effective use of register file is critical!

Memory operand

- Memory operands are stored in main memory
  - Large size
  - Outsize CPU →Slower than register file (100 to 500 times)
- High level language programs use memory operands
  - Variables
  - Array and string
  - Composite data structures
- Operations with memory operands
  - Units of byte/half word/word/double word
  - Load data from memory to register
  - Store data from register to memory

RISC-V memory organization



Byte address (32 bit)

0x0f
0x0e
0x0d
0x0c
0x0b
0x0a
0x09
0x08
0x07
0x06
0x05
0x04
0x03
0x02
0x01
0x00

Word 3

Word 2

Word 1

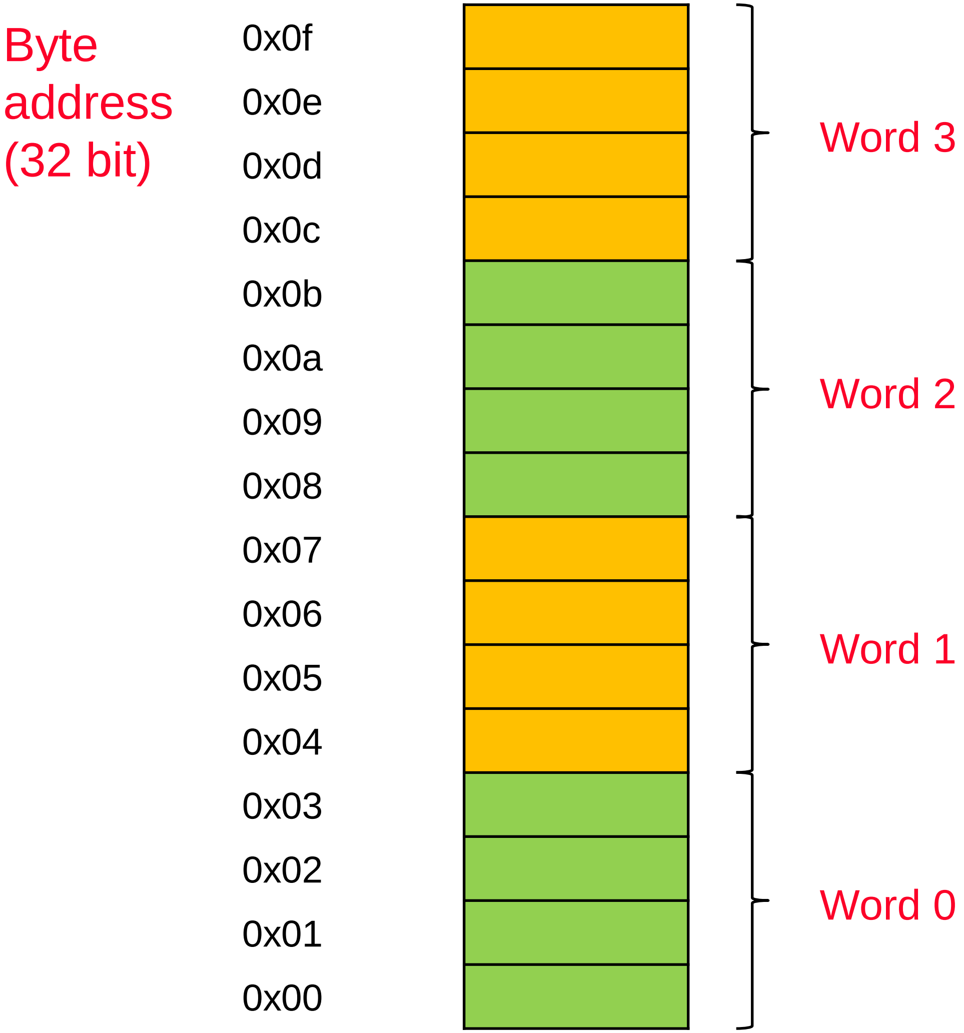
Word 0

- Byte addressable
- Words are accessed via byte address
- Only accessible via load/store instructions

Data alignment
word address = 4 * word number

RISC-V does not require data alignment, but it is **strongly recommended.**
➔ handled by compiler

RISC-V memory organization



Is this optimized to declare a struct in C like this?

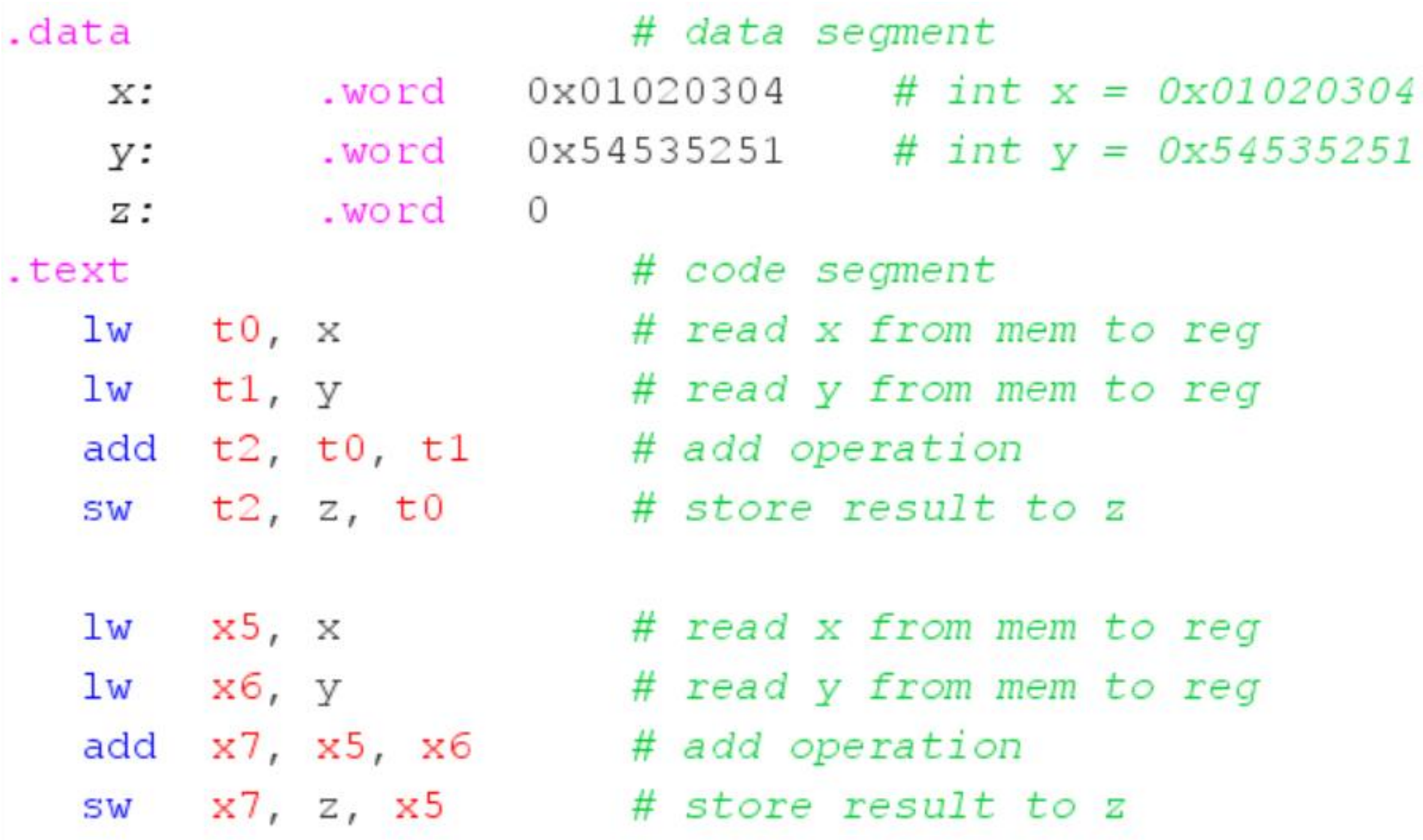
```
Struct data
{
        char x;
        short y;
        int z;
}
```

Aligned Data

- Primitive data type requires ***K*** bytes
- Address must be multiple of ***K***

Example: z = x + y

- x, y, z are allocated in mem, but must transfer to reg before adding
- Note: currently focus on instruction set first, assembly programming will be presented later



```
.data                          # data segment
    x:      .word   0x01020304     # int x = 0x01020304
    y:      .word   0x54535251     # int y = 0x54535251
    z:      .word   0
.text                          # code segment
   lw    t0, x                 # read x from mem to reg
   lw    t1, y                 # read y from mem to reg
   add   t2, t0, t1            # add operation
   sw    t2, z, t0             # store result to z

   lw    x5, x                 # read x from mem to reg
   lw    x6, y                 # read y from mem to reg
   add   x7, x5, x6            # add operation
   sw    x7, z, x5             # store result to z
```

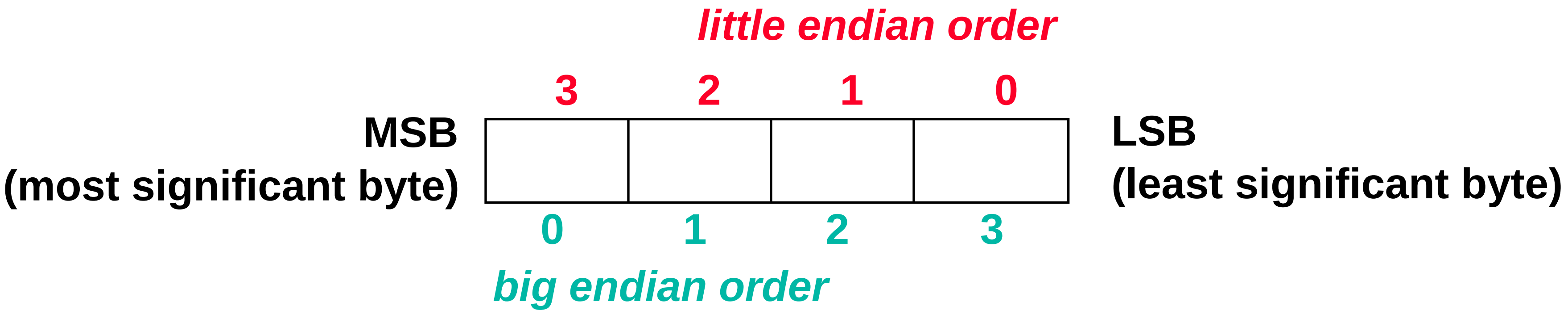
Byte Order

- Big Endian: word address points to MSB

  IBM 360/370, Motorola 68k, Sparc, HP PA

- Little Endian: word address points to LSB

  Intel 80x86, DEC, MIPS, RISC-V



little endian order

3 2 1 0

MSB
(most significant byte)

LSB
(least significant byte)

0 1 2 3

big endian order

Example

- Consider a word in RISC-V memory consists of 4 byte with hexa value as below
- What is the word’s value?

address	value
X	68
X+1	1B
X+2	5D
X+3	FA

- RISC-V is little-endian: address of LSB is X

➔ word’s value: FA5D1B68

Immediate operand

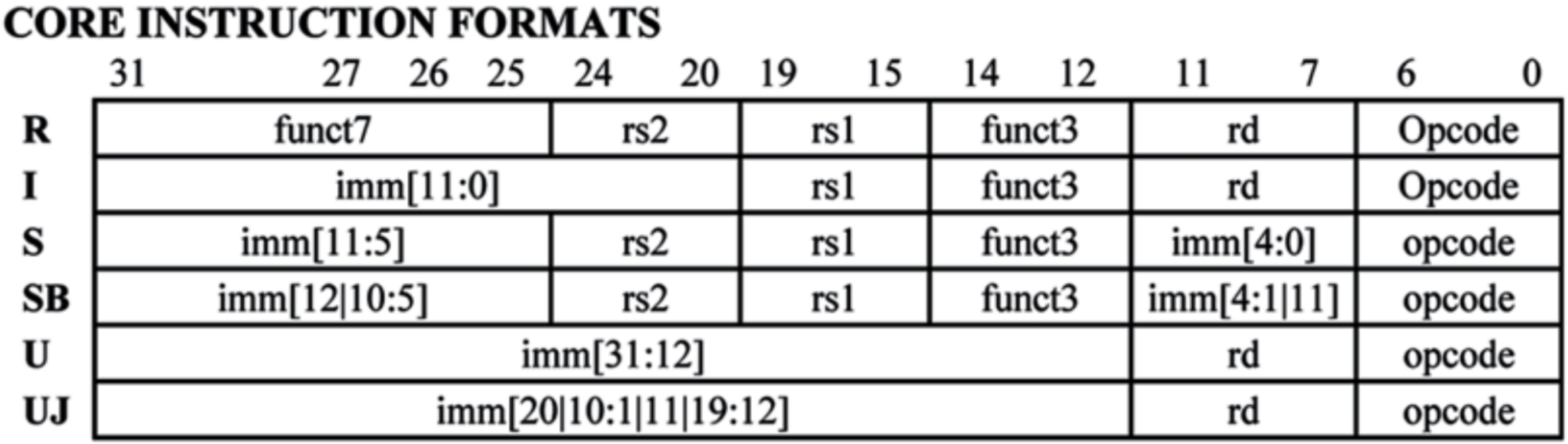
- Immediate value specified by a constant number
- Examples:

Assignment:	int x = 2024;
Const in expression:	x = y + 10;
Branching:	if.. else.., goto,…

- Does not need to be stored in register file or memory
  - Value stored right in instruction → faster
  - Fixed value specified at design time
  - Cannot change value at run time
- What is the most-used constant?
  - 0 value is stored in the special register: zero (x0)
  - Make common cases fast!

Instruction set

- Instruction: binary string represent opcode + operands
- RISC-V (RV32 variant) base instructions are 32 bits long.
  - Must be word-aligned in memory.
- 6 instruction formats



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

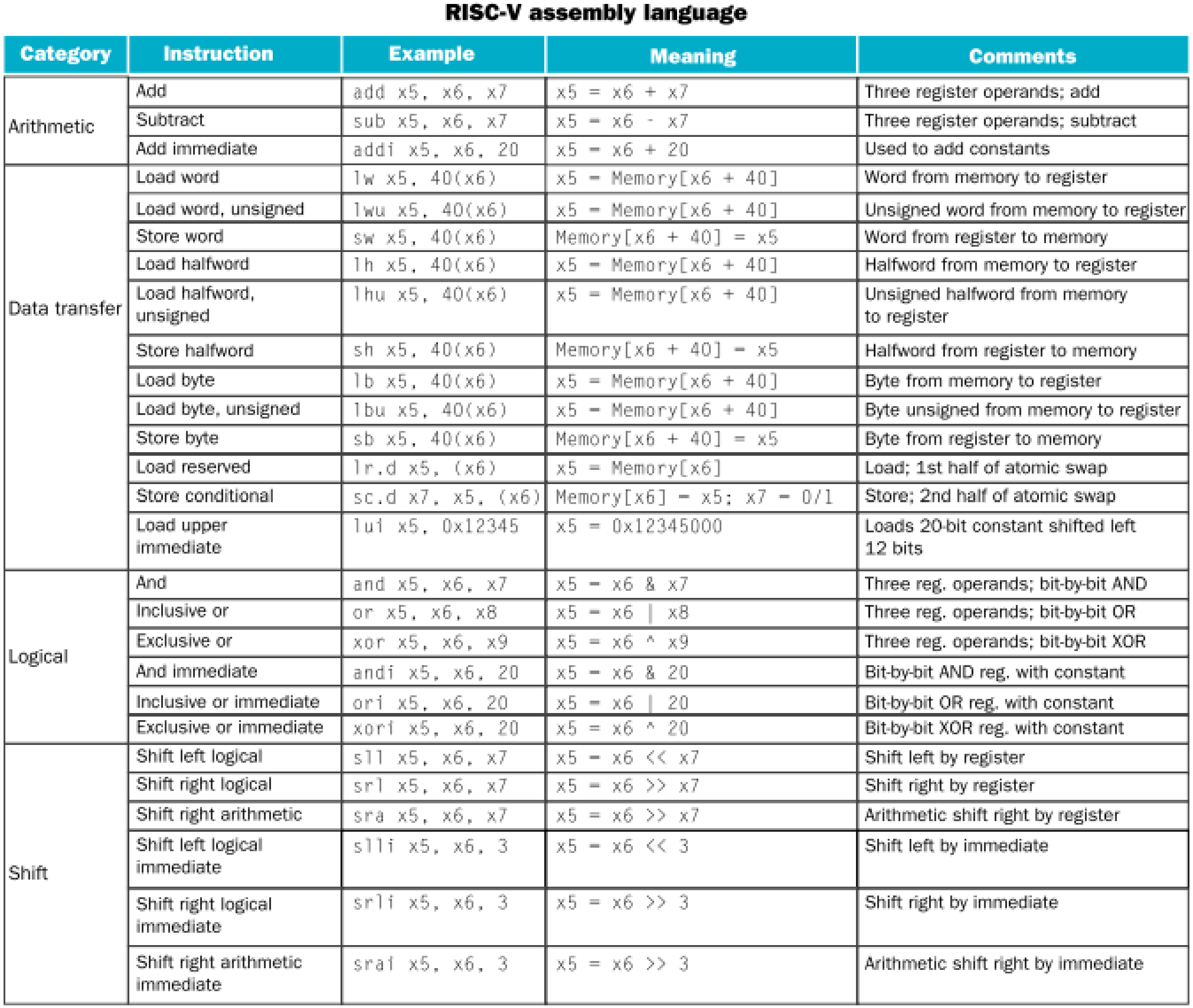
➔ ***Why not only one format? Or 20 formats?***

➔ ***DP3: Good design demands good compromises!***

Instruction categories

- Arithmetic: addition, subtraction,…
- Data transfer: transfer data between registers, memory, and immediate
- Logical and bitwise: and, or, xor, shift left/right…
- Branch: conditional and unconditional

Overview of RISC-V instruction set

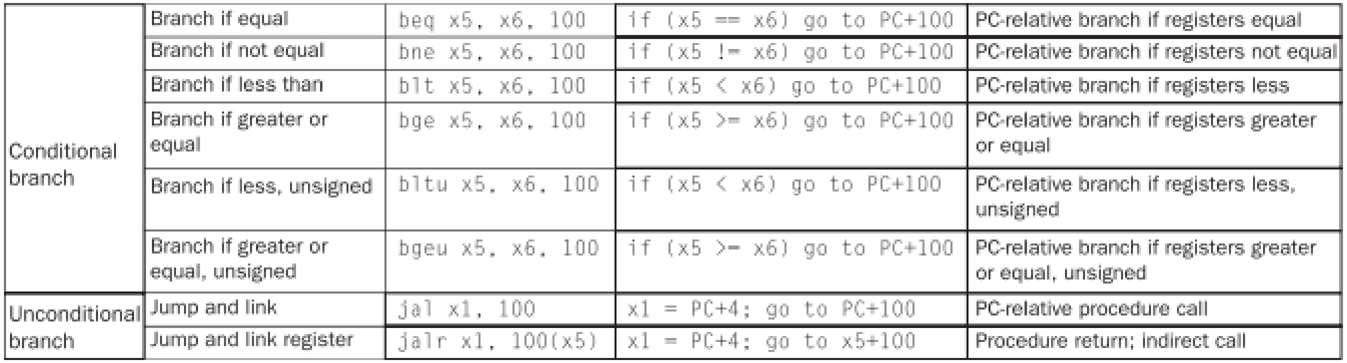


RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
Data transfer	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	`x5 = Memory[x6]`	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	`Memory[x6] = x5; x7 = 0/1`	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate

Fig. 2.1

Overview of RISC-V instruction set



Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

FIGURE 2.1 **(Continued).**

RISC-V Instruction set: Arithmetic operations

- RISC-V arithmetic statement

```
add  rd, rs1, rs2    #rd ← rs1 + rs2
sub  rd, rs1, rs2    #rd ← rs1 - rs2
addi rd, rs1, imm    #rd ← rs1 + imm
```

rs1	5-bits	register file address of the 1st source operand
rs2	5-bits	register file address of the 2nd source operand
rd	5-bits	register file address of the result's destination

*Why there is no **subi** instruction?*

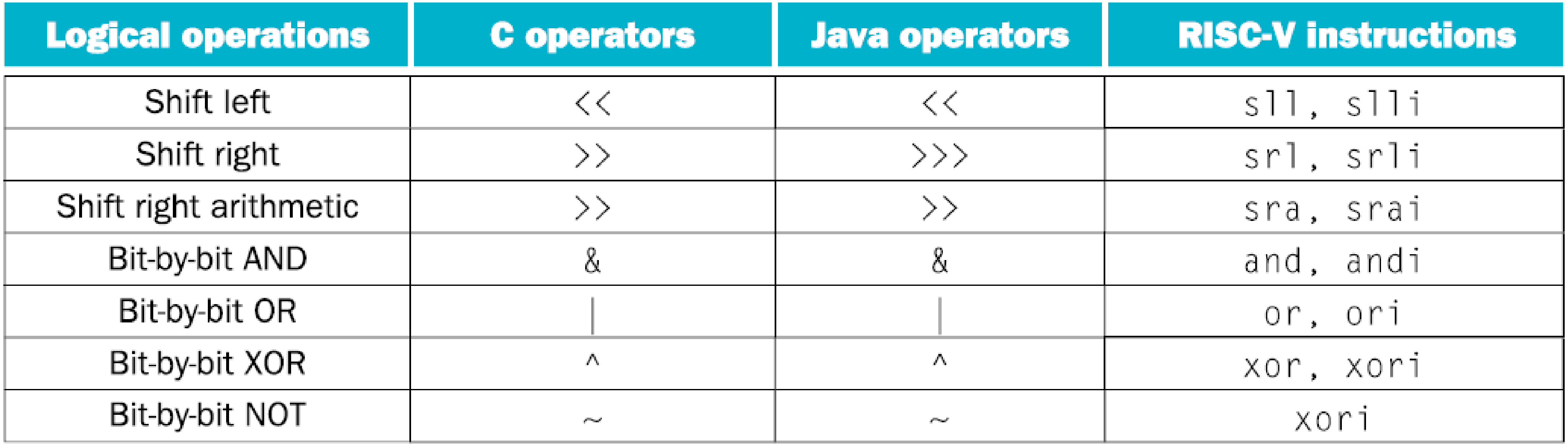
Example

- Currently s1 = 6
- What is value of s1 after executing the following instruction

```
addi    s2, s1, 3
addi    s1, s1, -2
sub     s1, s2, s1
```

RISC-V Instruction set: Logical operations

- Bitwise operations



Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	xori

RISC-V Instruction set: Logical operations

- Basic logic operations

```
and  rd, rs1, rs2    #rd ← rs & rs2
andi rd, rs1, imm    #rd ← rs & imm
or   rd, rs1, rs2    #rd ← rs | rs2
ori  rd, rs1, imm    #rd ← rs | imm
xor  rd, rs1, rs2    #rd ← rs ^ rs2
xor  rd, rs1, imm    #rd ← rs ^ imm
```

- Example s1 = 8 = 0000 1000, s2 = 14 = 0000 1110

```
and  s3, s1, s2
or   s4, s1, s2
```

RISC-V Instruction set: Shift operations

- Logical shift and arithmetic shift: move all the bits left or right

```
sll  rd, rs1, rs2    #rd ← rs1 << rs2

srl  rd, rs1, rs2    #rd ← rs1 >> rs2

sra  rd, rs1, rs2    #rd ← rs1 >> rs2
                     (keep sign bit)

slli rd, rs1, imm    #rd ← rs1 << imm

srli rd, rs1, imm    #rd ← rs1 >> imm

srai rd, rs1, imm    #rd ← rs1 >> imm
                     (keep sign bit)
```

RISC-V Instruction set: Memory access instructions

- RISC-V has two basic data transfer instructions for accessing memory

```
lw rd, imm(rs1)   #load word from memory

sw rs2, imm(rs1)   #store word to memory
```

- The data is loaded into (lw) or stored from (sw) a register in the register file
- The memory address is formed by adding the contents of the base address register to the offset value
- Offset can be negative
- Data alignment is strongly recommended
- Why not the instruction is just like this: lw rd, imm?

RISC-V Instruction set: Load Instruction

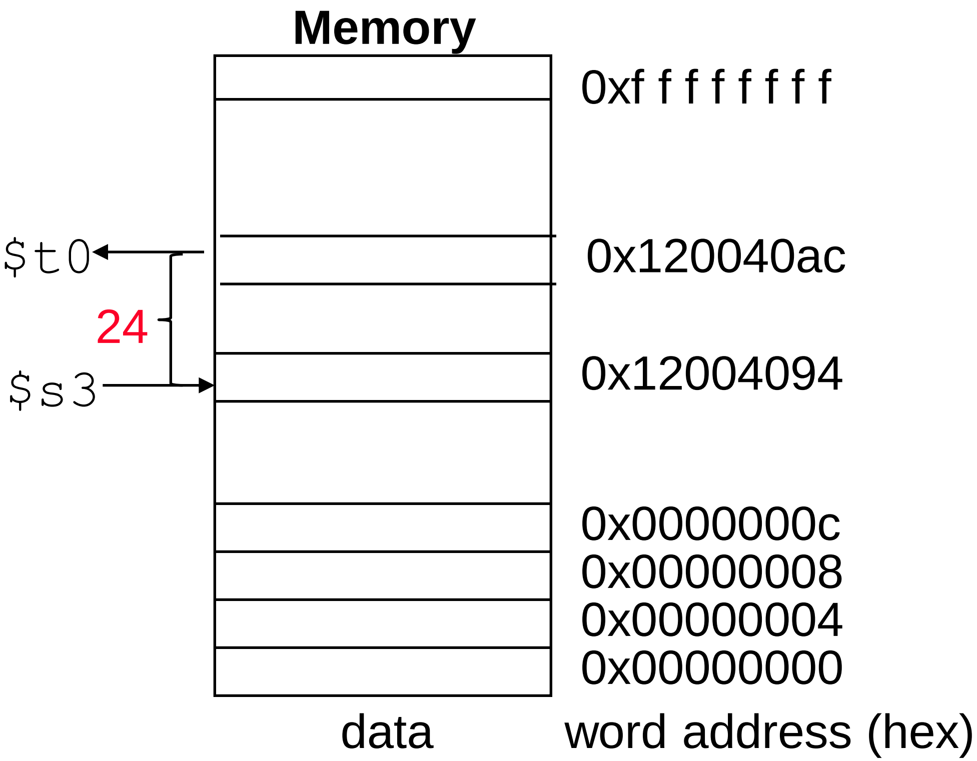
- Load/Store Instruction Format:

```
lw t0, 24(s3)       #t0← mem at 24+s3
```

Which memory word will be loaded to t0?

24_{10} + \$s3 =

```
  . 0001 1000 (24)
+ . 1001 0100 (94)
  . 1010 1100 (ac)
= 0x1200 40ac
```



RISC-V Instruction set: Load Instruction

- Given the integer array A stored in memory, with base address stored in x13.

```
int A[100]; //x13 holds address of A[0]
```

- What is equivalent C code of this?

```
lw x10, 12(x13)

addi x12, x10, 10

sw x12, 40(x13)
```

RISC-V control flow instructions

- RISC-V conditional branch instructions:

```
bne rs1, rs2, Dest #go to Dest if rs1≠rs2
beq rs1, rs2, Dest #go to Dest if rs1=rs2
bge rs1, rs2, Dest #go to Dest if rs1>=rs2
blt rs1, rs2, Dest #go to Dest if rs1<rs2

bgeu/bltu: unsigned comparison
```

Ex:

```
    if (i==j)
          h = i + j;

          bne s0, s1, Exit
          add s3, s0, s1
Exit :    ...
```

Example

```
start:
                addi    s0, zero, 2    #load value for s0
                addi    s1, zero, 2
                addi    s3, zero, 0
                beq     s0, s1, Exit
                add     s3, s2, s1
        Exit:   add     s2, s3, s1
.end start
```

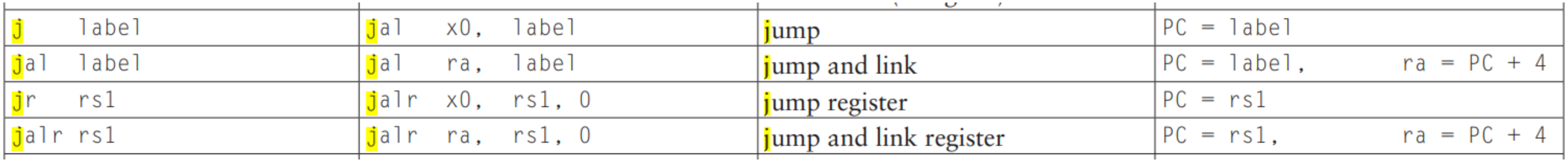
What is final value of s2?

Unconditional branch

- Unconditional branch instruction or jump instruction:

```
j   Dest          #go to Dest
```

- Note: this is a pseudo-instruction implemented with the jal instruction, and auipc instruction if necessary



j label	jal x0, label	jump	PC = label
jal label	jal ra, label	jump and link	PC = label, ra = PC + 4
jr rs1	jalr x0, rs1, 0	jump register	PC = rs1
jalr rs1	jalr ra, rs1, 0	jump and link register	PC = rs1, ra = PC + 4

Comparison instruction

- Set flag based on condition: `slt`
- Set on less than instruction:

```
slt $t0, $s0, $s1     # if $s0 < $s1     then
                      # $t0 = 1          else
                      # $t0 = 0
```

- Alternate versions of `slt`

```
slti $t0, $s0, 25     # if $s0 < 25 then $t0=1 ...
sltu $t0, $s0, $s1    # if $s0 < $s1 then $t0=1 ...
sltiu $t0, $s0, 25    # if $s0 < 25 then $t0=1 ...
```

- Can be combined with bne/beq for conditional branches

Example

- Write assembly code to do the following

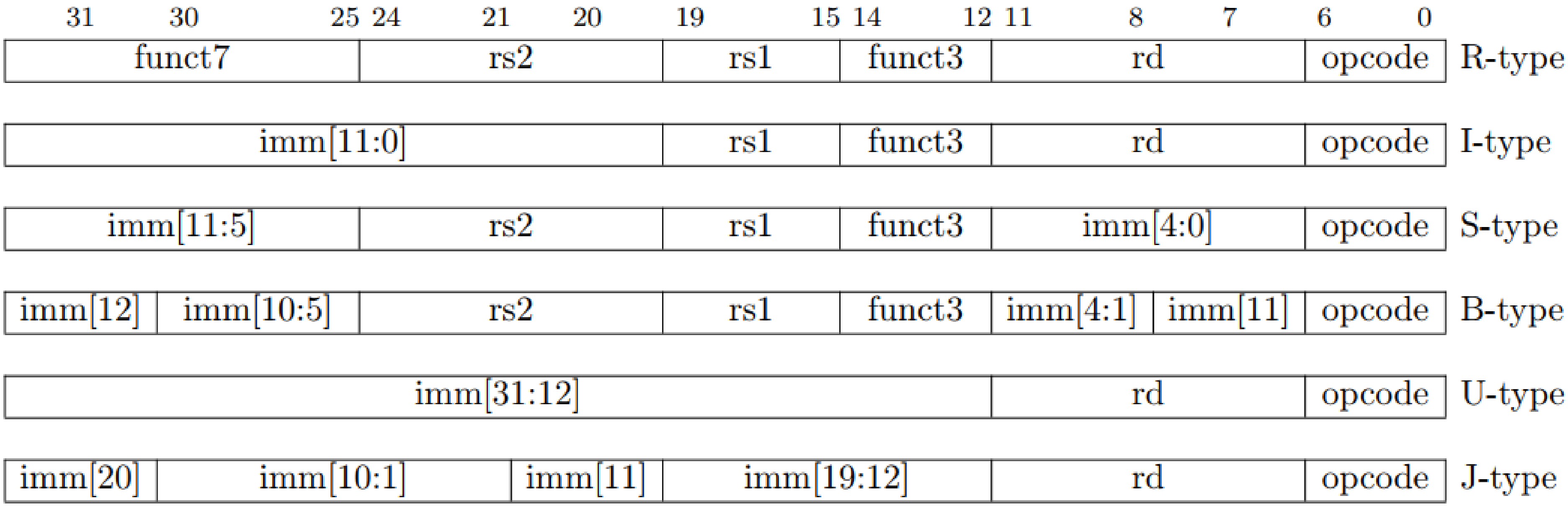
```
if (i<5)
    X = 3;
else
    X = 10;
```

Solution

```
      slti t0,s1,5           # i<5? (inverse condition)
      beq  t0,zero,else      # if i>=5 goto else part
      addi t1,zero,3         # X = 3
      j    endif             # skip the else part
else: addi t1,zero,10        # X = 10
endif:...
```

Representation of RISC-V instruction

- All RISC-V instructions are 32 bits wide

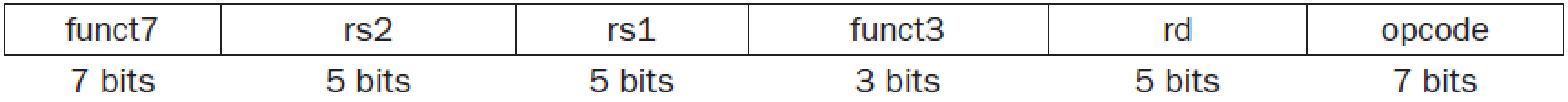


31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

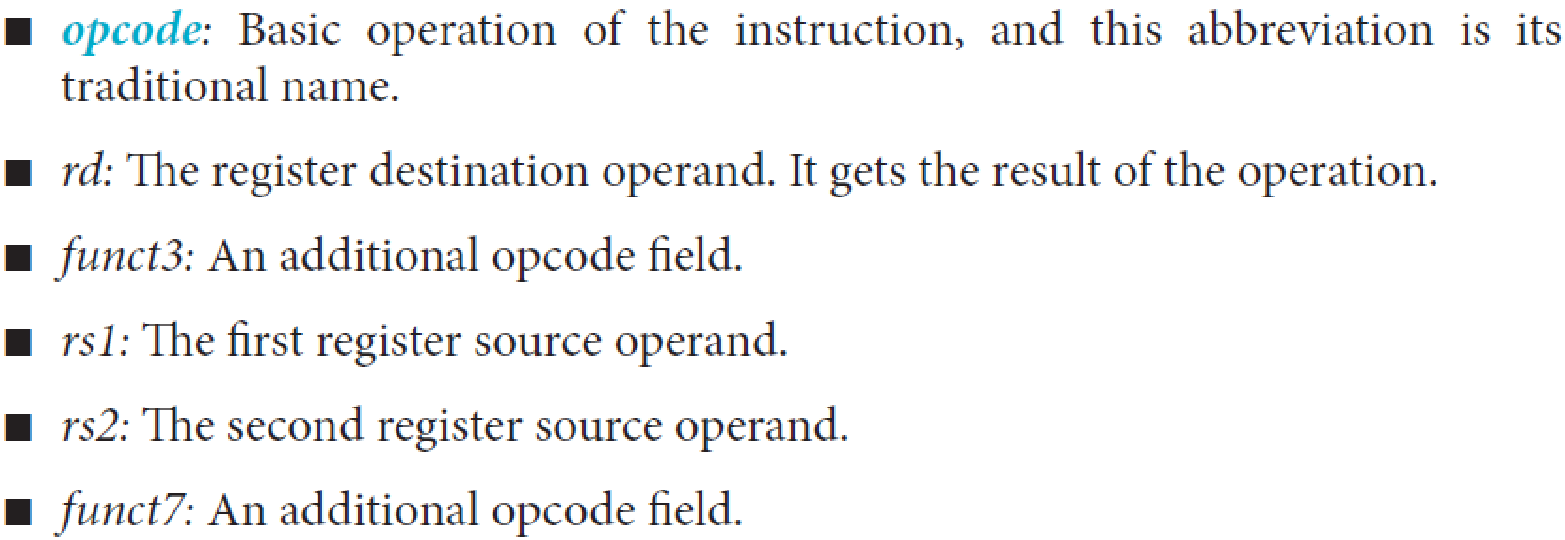
The RISC-V Instruction Set Manual, Volume I: User-Level ISA

R-format instruction: all operands are registers

- All fields are encoded by mnemonic names

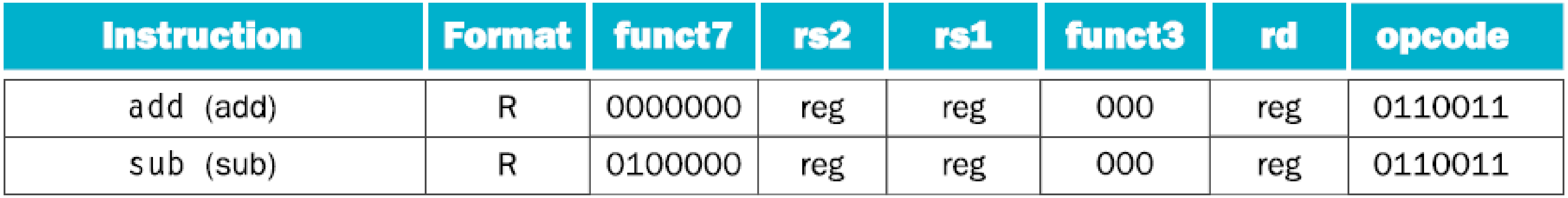


funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits



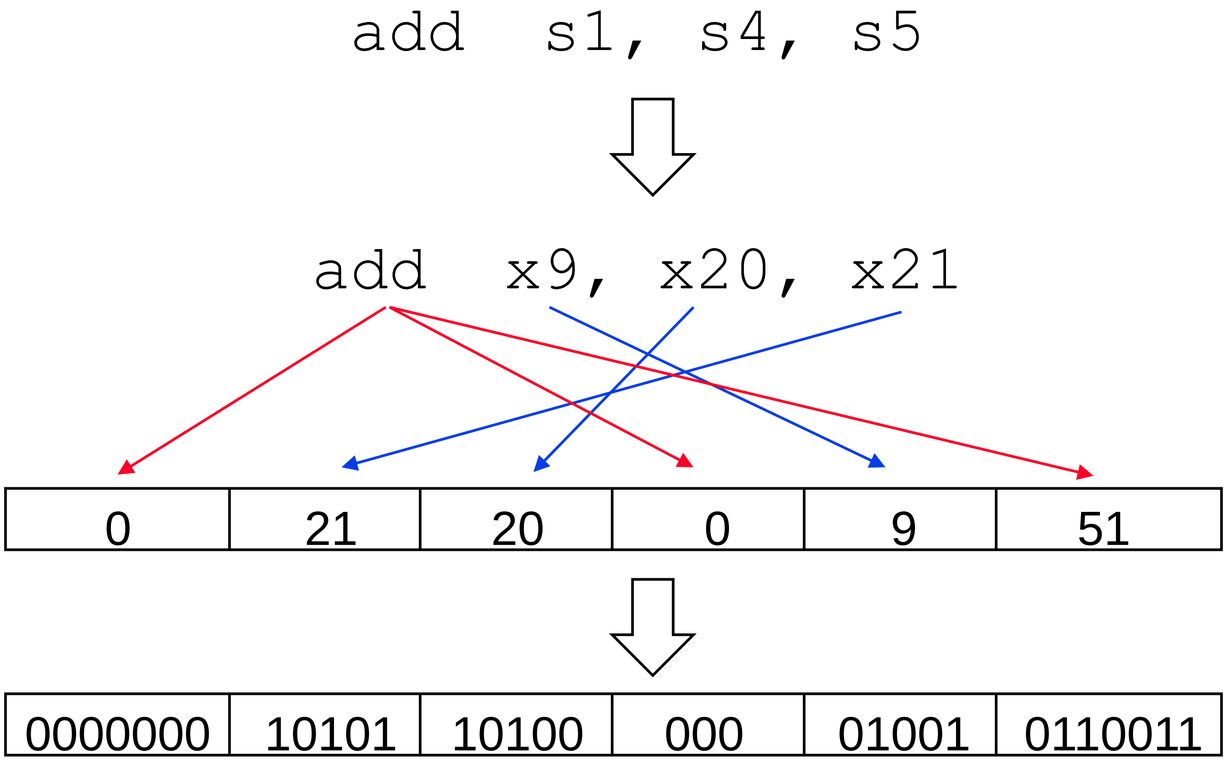
- *opcode*: Basic operation of the instruction, and this abbreviation is its traditional name.
- *rd:* The register destination operand. It gets the result of the operation.
- *funct3:* An additional opcode field.
- *rs1:* The first register source operand.
- *rs2:* The second register source operand.
- *funct7:* An additional opcode field.

- Examples



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

Example of R-format instruction



```
add  s1, s4, s5
```

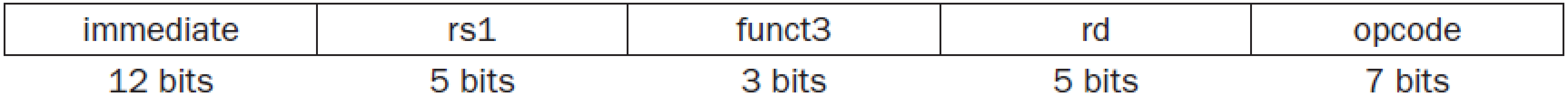
```
add  x9, x20, x21
```

0	21	20	0	9	51

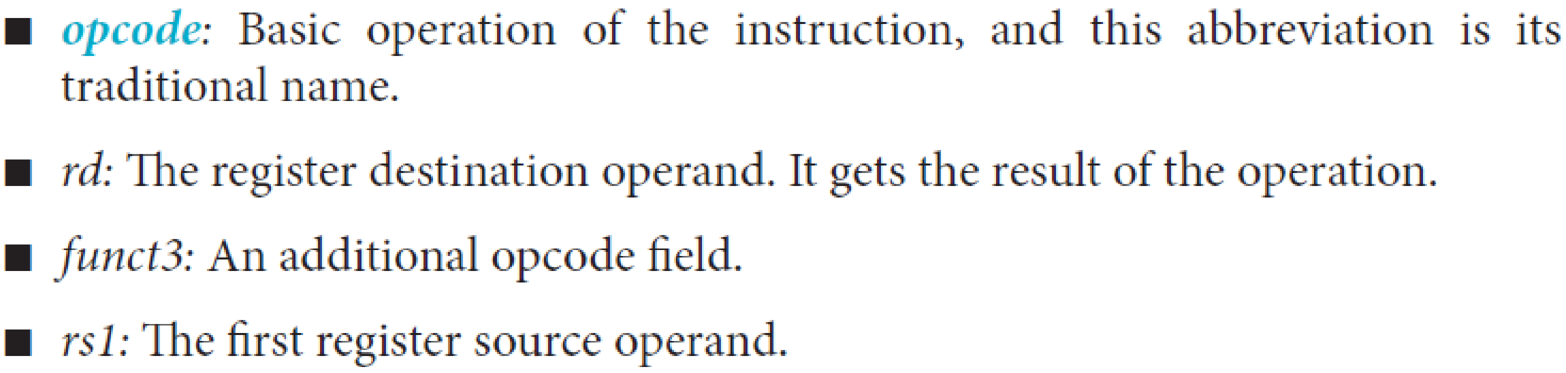
0000000	10101	10100	000	01001	0110011

I-format instruction: 2 registers + 1 immediate

- Combines the funct7 and rs2 for 12-bit immediate

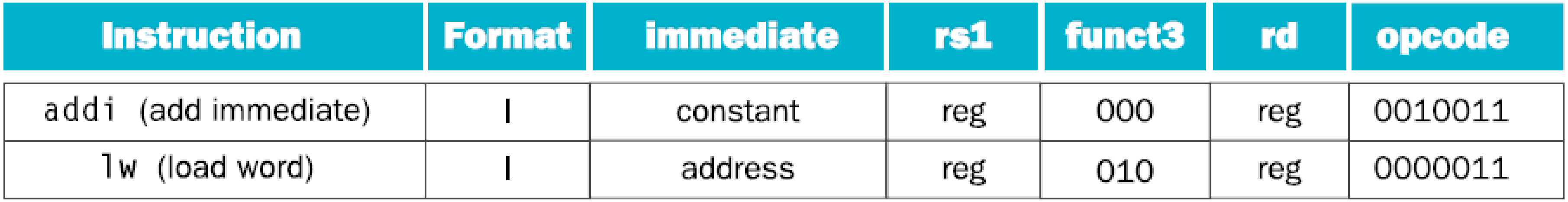


immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits



- *opcode*: Basic operation of the instruction, and this abbreviation is its traditional name.
- *rd*: The register destination operand. It gets the result of the operation.
- *funct3*: An additional opcode field.
- *rs1*: The first register source operand.

- Examples



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

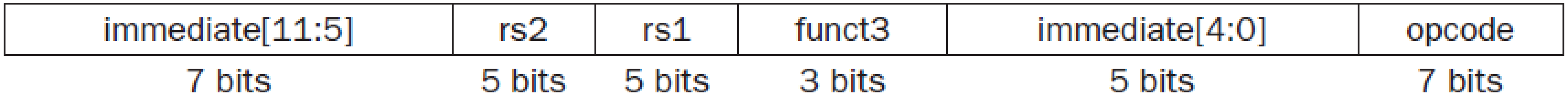
Example

- Find machine codes of the following instructions

```
lw     t0, 0(s1)    # initialize maximum to A[0]
addi   t1, zero, 0 # initialize index i to 0
addi   t1, t1, 1    # increment index i by 1
```

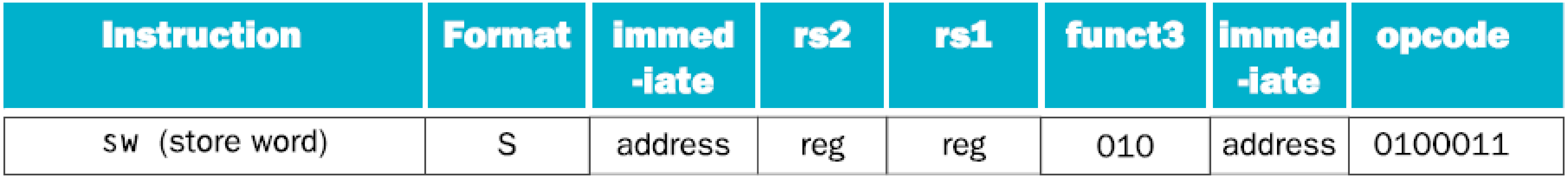
S-format instruction: 2 registers + 1 immediate

- Combines the funct7 and rd for 12-bit immediate
- Used for the store instructions, which does not require rd



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

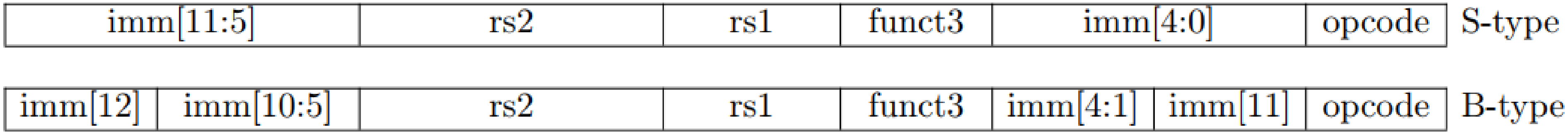
- Examples



Instruction	Format	immed -iate	rs2	rs1	funct3	immed -iate	opcode
sw (store word)	S	address	reg	reg	010	address	0100011

B-format instruction: 2 registers + 1 immediate

- Combines the funct7 and rd for 13-bit immediate
  - Lsb = 0 (imm[12:1] for half-word instruction address, more on this later).
  - Keep the same bit position as S-format
  - Msb always in bit 31 of instruction word (simplified sign-extension, also more on this later)
- As a result: position of 13 bits immediate are mixed
- Used for conditional jump instructions

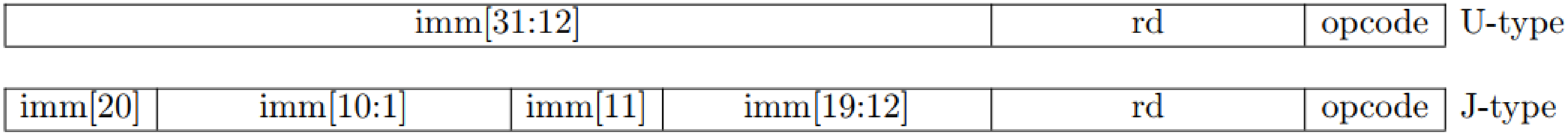


imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

U- and J-format instruction: 1 register + 1 immediate

- Combines the funct7, funct3, rs1 and rs2 for 20-bit immediate



imm[31:12]	rd	opcode	U-type

imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode	J-type

- U-format: for load/add 20 bit upper-immediate to register

```
lui rd, upimm         # rd ← {upimm,000}

auipc rd, upimm       # rd ← PC + {upimm,000}
```

- J-format: for jump and link

```
jal rd, label         # PC ← PC+addr, rd ← PC+4

                      addr = SignExt{imm,0}
```

- Pseudo-instruction j label ➔ jal x0, label

Working with wide immediates and addresses

- Many operations need 32-bit immediates
  - Loading 32-bit immediates to registers
  - Loading addresses to registers
- However, instructions are only 32 bit-long
  - Not sufficient to store 32-bit immediates in one instruction
  - →combine 2 instructions to support wide immediates
- Example: load the value 0x3D0100 into s0

```
lui s0, 0x003D0           #s0 ← 0x003D0000
addi s0, s0, 0x0100       #s0 ← 0x003D0100
```

- *Pseudo-instructions*: combination of real instructions, for convenience
  - li, la

Working with wide immediates and addresses

- Special case: long jump
- Conditional branches: blt, bne,..
  - B-format, with 12 bit immediates
  - Limited to 4 KB → limit branching distance
  - Solution: change to jal

```
beq x10, x0, L1 #limit 4KB
```

⇨

```
     bne x10, x0, L2
     jal x0, L1 #limit 1MB
L2:
```

- Unconditional jump (jal)
  - Distance is limited to 1MB
  - Solution: use jalr, combine with auipc if necessary

```
jalr rd, rs1, imm # PC = rs1 + SignExt(imm), rd = PC+4
```

Exercise

- How branch instruction is executed?
- ➔ PC-relative addressing mode

```
      slti t0, s1, 5
      bne  t0, zero, else
      addi t1, zero, 3
      j    endif
else: addi t1, zero, 10
endif:...
```

→ **How can CPU jump from here to the “else” label?**

Example

The simple switch

```
switch(test) {
   case 0:
         a=a+1; break;
   case 1:
         a=a-1; break;
   case 2:
         b=2*b; break;
   default:
}
```

Assuming that: `test,a,b` are stored in `$s1,$s2,$s3`

Solution

```
          beq    s1,t0,case_0
          beq    s1,t1,case_1
          beq    s1,t2,case_2
          j      default
case_0:
          addi   s2,s2,1        #a=a+1
          j      continue
case_1:
          sub    s2,s2,t1       #a=a-1
          j      continue
case_2:
          add    s3,s3,s3       #b=2*b
          j      continue
default:
continue:
```

Example

- Write assembly code correspond to the following C code

```
for (i = 0; i < n; i++)
    sum = sum + A[i];
```

```
loop:

    addi  s1,s1,1       #i=i+step

    add   t1,s1,s1      #t1=2*s1

    add   t1,t1,t1      #t1=4*s1

    add   t1,t1,a0      #t1 <- address of A[i]

    lw    t0,0(t1)      #load value of A[i] in t0

    add   s0,s0,t0      #sum = sum+A[i]

    bne   s1,a1,loop    #if i != n, goto loop
```

Example

The simple while loop: `while (A[i]==k) i=i+1;`

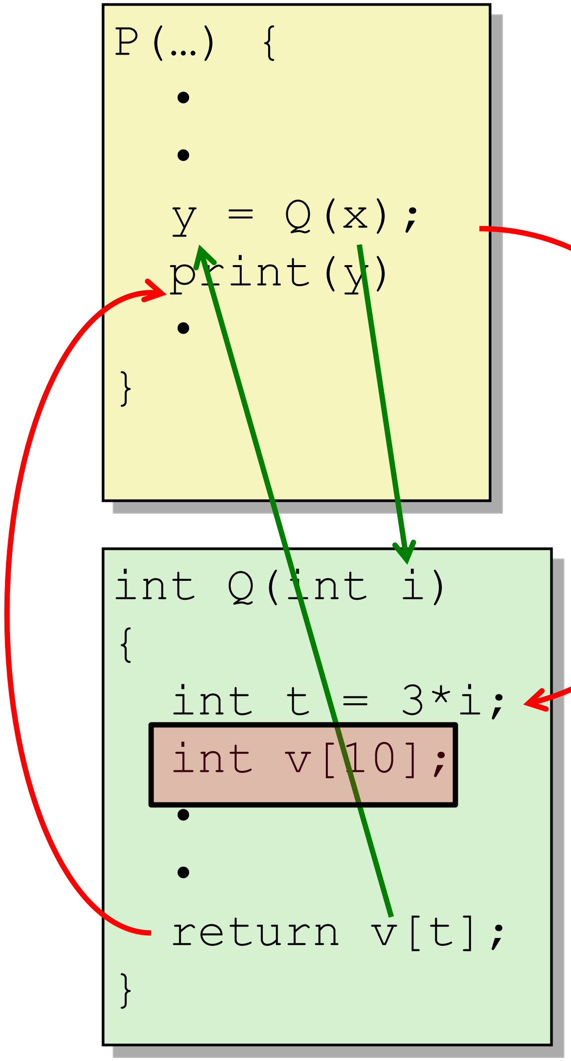
Assuming that: `i, k, A` are stored in `x22,x24,x25`

Solution

```
Loop:
      slli x10, x22, 2    #i*4
      add x10, x10, x25   #A[i] address
      lw x9, 0(x10)       #A[i] value
      bne x9, x24, Exit   #break if != k
      addi x22, x22, 1    #next element
      beq x0, x0, Loop
Exit: …
```

Procedures

- Stack structure
- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Register saving conventions
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return

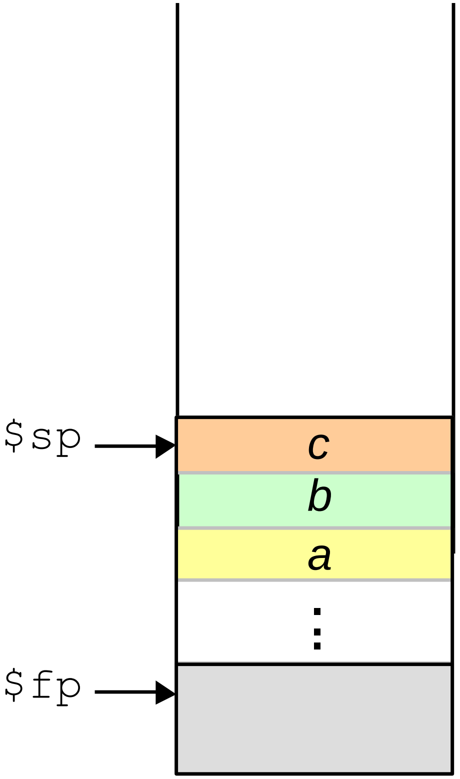


```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
int Q(int i)
{
   int t = 3*i;
   int v[10];
   •
   •
   return v[t];
}
```

Stack structure

- A region of memory operating on a Last In First Out (LIFO) principle
- The bottom of stack is at the highest location
- sp: point to the top of the stack



Stack structure

- To push data into stack
  - `addi sp, sp, -4`
  - `sw t0, 0(sp)`
- To pop data from the stack
  - `lw t0, 0(sp)`
  - `addi sp, sp, 4`
  - Note that: the data is still there in the stack, but we are not going to work with it anymore.

Passing control flow

- **Procedure call**: using RISC-V procedure call instruction

```
    jal  rd, ProcAddress      #jump and link
```

  - Saves the return address (PC+4) in destination register rd (usually in `ra` or `x1`)
  - Jump to the ProcAddress

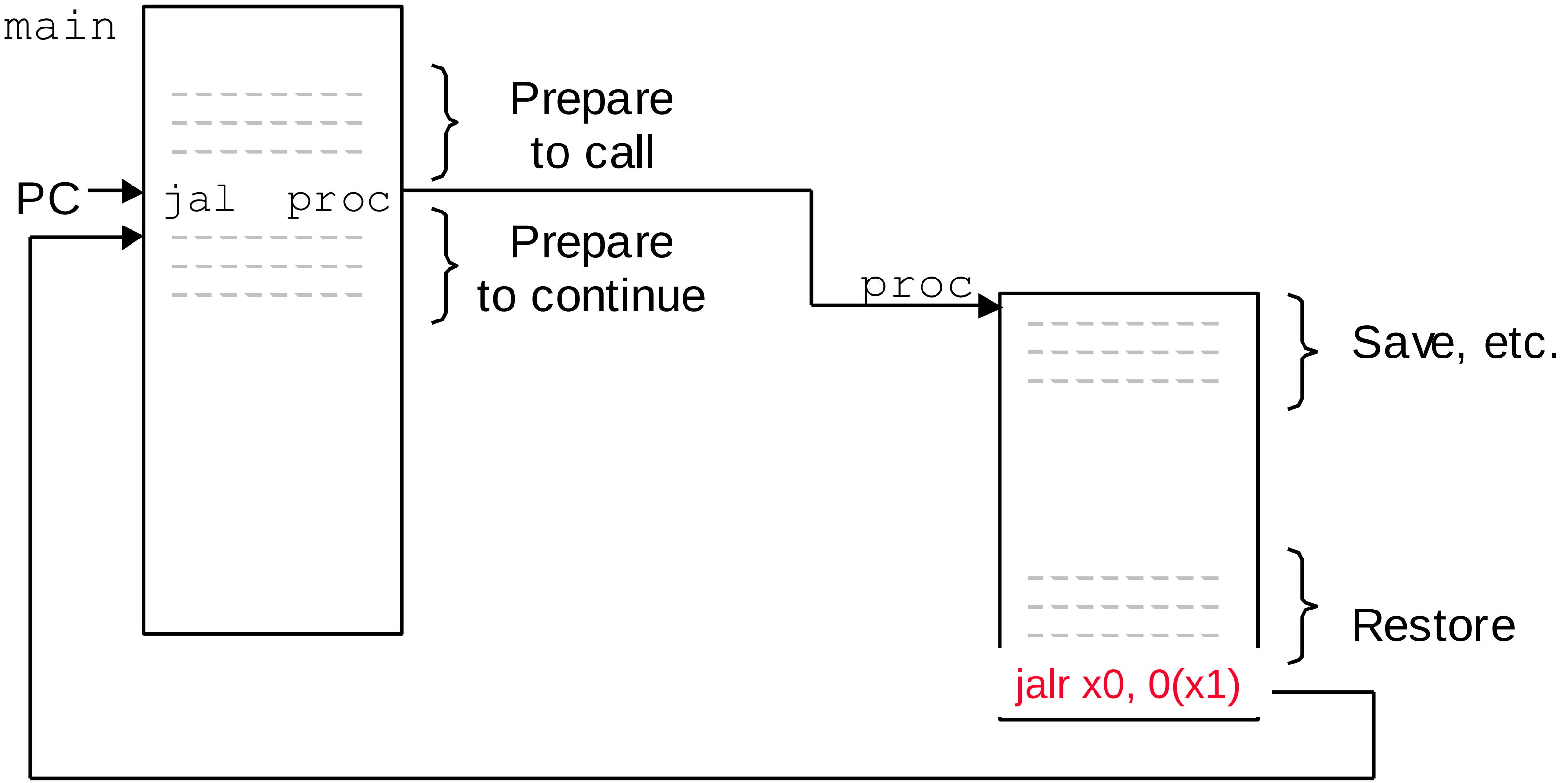
- **Return address**:
  - Address of the next instruction right after call

- **Procedure return**: procedure return with

```
    jalr x0, 0(x1)
```

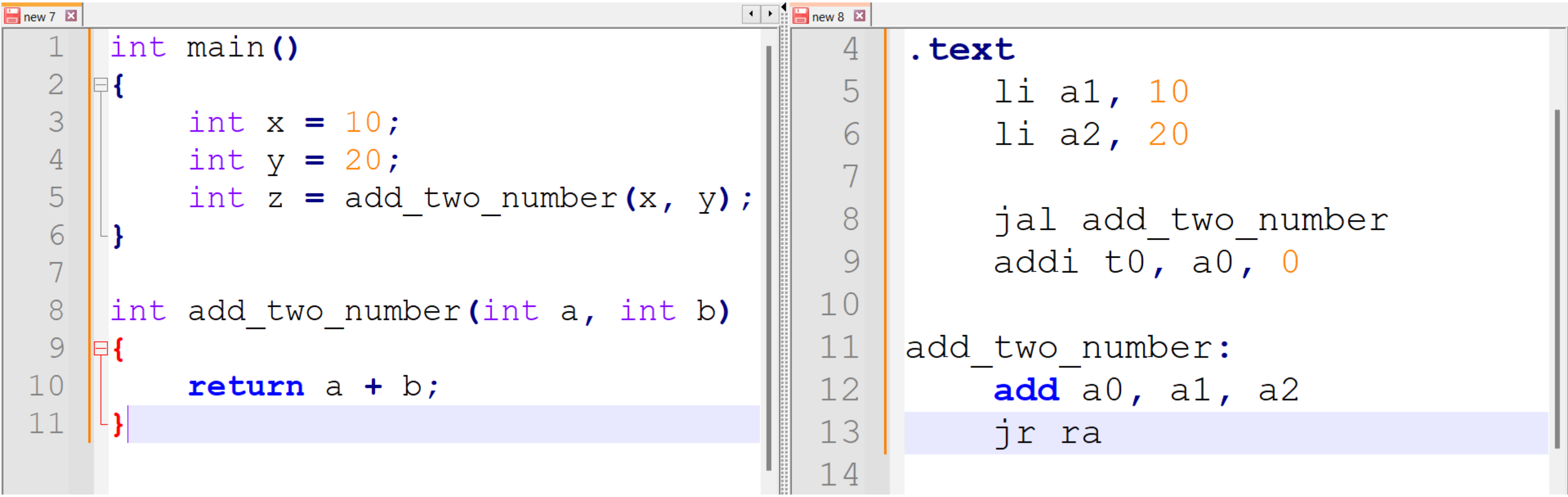
  - Update the value of PC = ra
  - Jump to the address of PC (the next instruction right after procedure call)

Passing control



Passing control

- Demonstrate on the Rars simulator



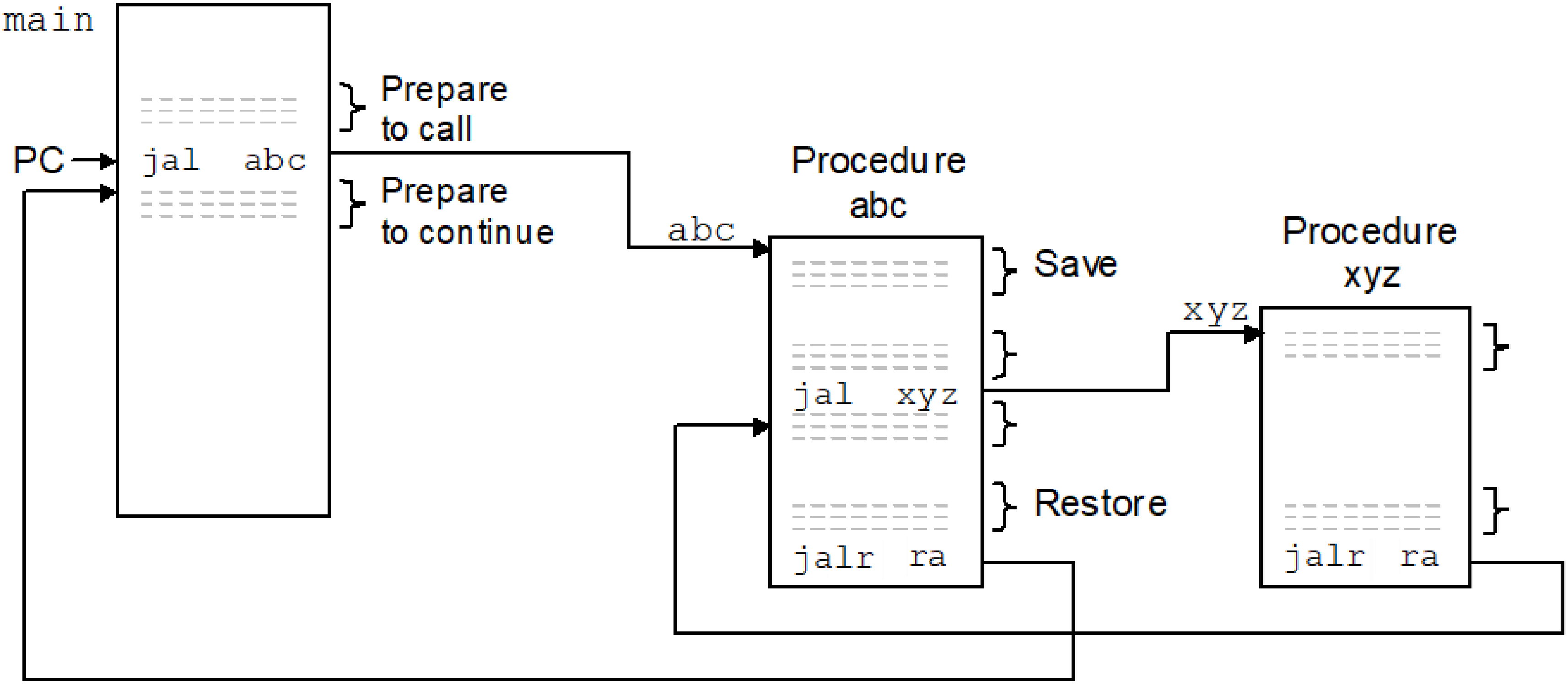
```c
int main()
{
    int x = 10;
    int y = 20;
    int z = add_two_number(x, y);
}

int add_two_number(int a, int b)
{
    return a + b;
}
```

```asm
.text
    li a1, 10
    li a2, 20

    jal add_two_number
    addi t0, a0, 0

add_two_number:
    add a0, a1, a2
    jr ra

```

- Take care the value of the pc and ra register!

Procedure call and nested procedure call

Question: how can the CPU resume the main program execution?



Example of nested procedure call

Passing data

- Use registers
  - Input arguments:
    - a0-a7
    - 8 parameters (arguments) maximum
  - Return value:
    - a0

- What if we want to pass more than 8 arguments → use the stack:
  - Caller pushes arguments into stack before calling the callee
  - Callee get arguments from the stack
  - (Optional) Callee saves the return value to the stack
  - Question: who clean the stack, caller or callee?

Register saving convention

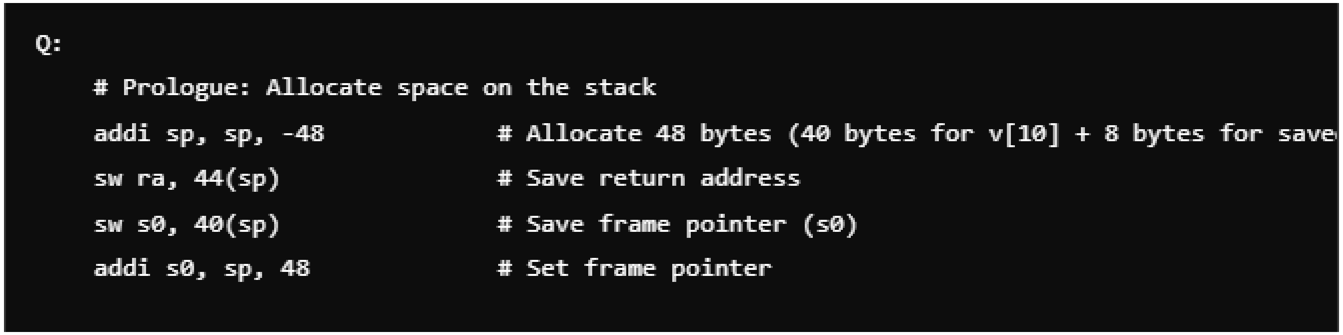
- Registers to be saved by the **caller**
  - ra, t0-t6, a0-a7
- Registers must be saved by the **callee**
  - sp, s0-s11
- Note: save the registers just in case you need to modify them.
- Question: where to save the above registers?

Memory management

- Question: where to locate the local variables: t and v?
  - Use registers: # of registers is limited
  - Use Stack

```
P(…) {
   •
   •
   y = Q(x);
   print(y)
   •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

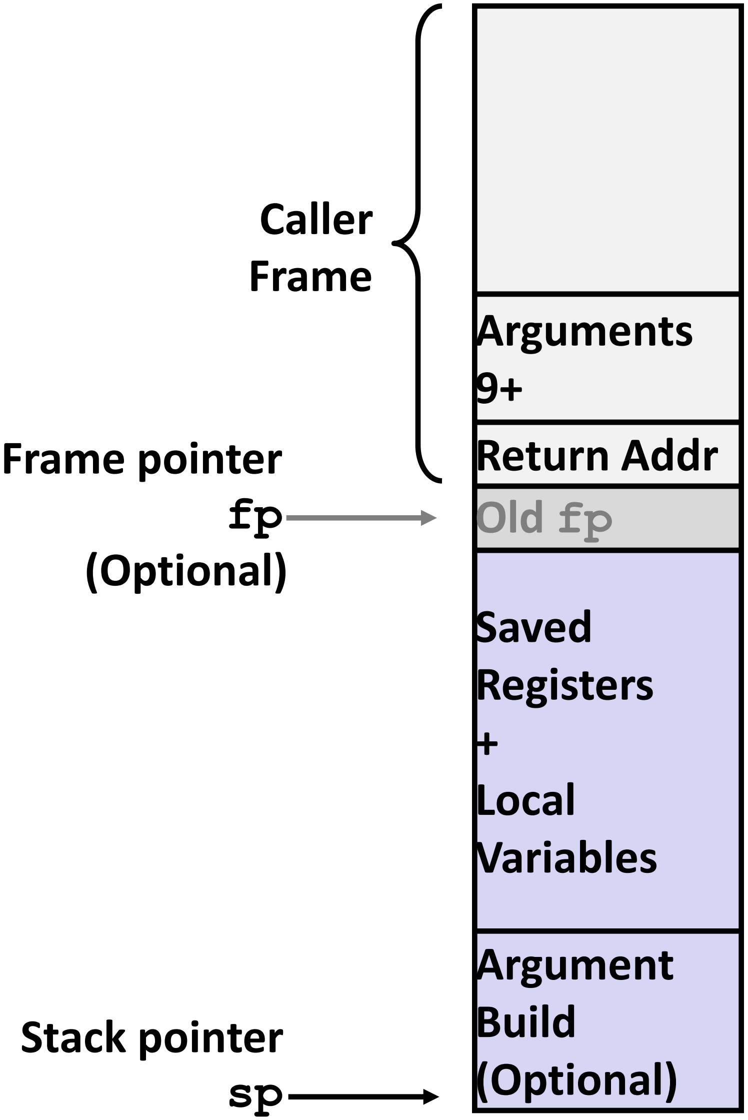


```
Q:
    # Prologue: Allocate space on the stack
    addi sp, sp, -48            # Allocate 48 bytes (40 bytes for v[10] + 8 bytes for save
    sw ra, 44(sp)               # Save return address
    sw s0, 40(sp)               # Save frame pointer (s0)
    addi s0, sp, 48             # Set frame pointer
```

Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)

- Caller Stack Frame
  - Return address
    - Pushed by `jal` instruction
  - Arguments for this call


Caller Frame
Arguments 9+
Return Addr
Frame pointer fp (Optional)
Old fp
Saved Registers + Local Variables
Argument Build (Optional)
Stack pointer sp

Six Steps in the Execution of a Procedure

1. Main routine (caller) places parameters in a place where the procedure (callee) can access them
   - `a0 – a7 (x10 - x17)`: 8 argument registers
2. Caller transfers control to the callee (`jal`)
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where the caller can access it
   - `a0 - a1`: two value registers for result values
6. Callee returns control to the caller (`jalr`)
   - `ra (x1)`: one return address register to return to caller

Procedure that does not call another proc.

- C code:

```
int leaf_example (int g, h, i, j)
{

    int f;
    f = (g + h) - (i + j);
    return f;
}
```

  - g, h, i, j stored in a0, a1, a2, a3
  - f in s0 (need to be saved)
  - t0 and t1 used for temporary data
  - Preserve all s0, t0, t1 for safety
  - Result in a0

Sample code

```
leaf_example:
      addi  sp, sp, -12     # room for 3 items
      sw    t1, 8(sp)       # save t1
      sw    t0, 4(sp)       # save t0
      sw    s0, 0(sp)       # save s0
      add   t0, a0, a1      # t0 = g+h
      add   t1, a2, a3      # t1 = i+j
      sub   s0, t0, t1      # s0 = (g+h)-(i+j)
      add   a0, s0, zero    # return value in a0
      lw    s0, 0(sp)       # restore s0
      lw    t0, 4(sp)       # restore t0
      lw    t1, 8(sp)       # restore t1
      addi  sp, sp, 12      # deallocate
      jalr  zero, 0(ra)     # return to caller
```

Exercise: write code to utilize the procedure above

Stack usage

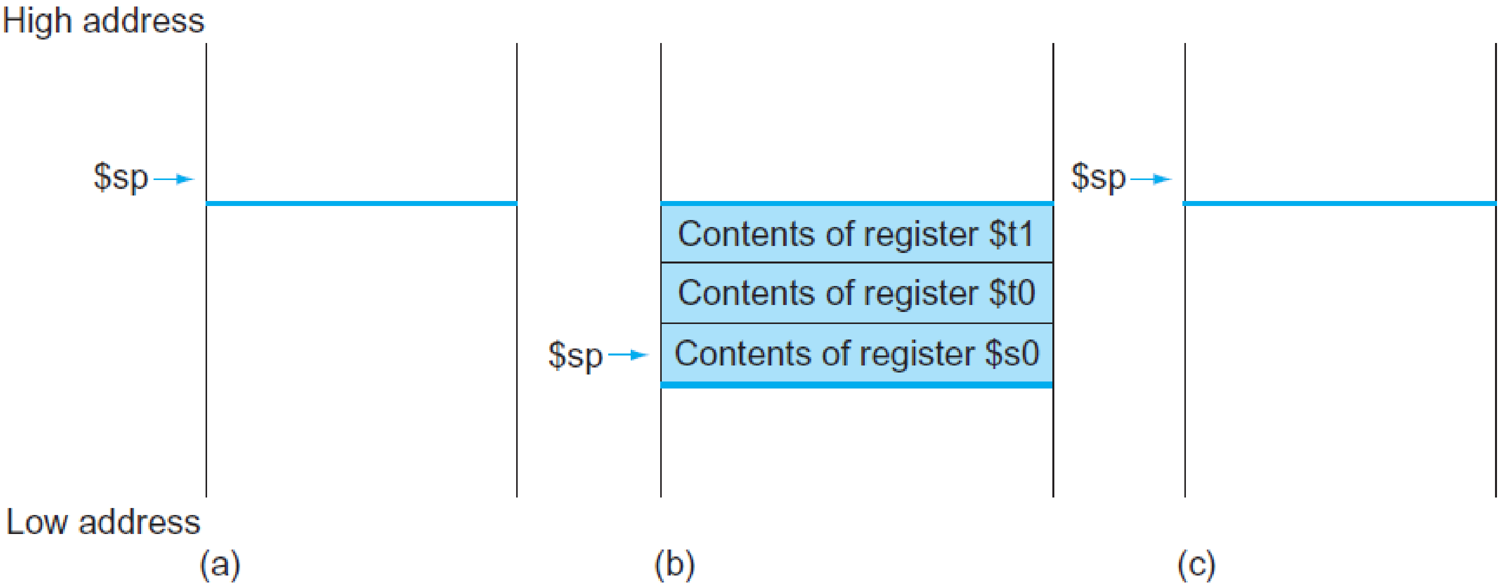


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Procedure with nested proc.

- C code:

```
int fact (int n)
{
  if (n < 1) return (1);
  else return n * fact(n - 1);
}
```

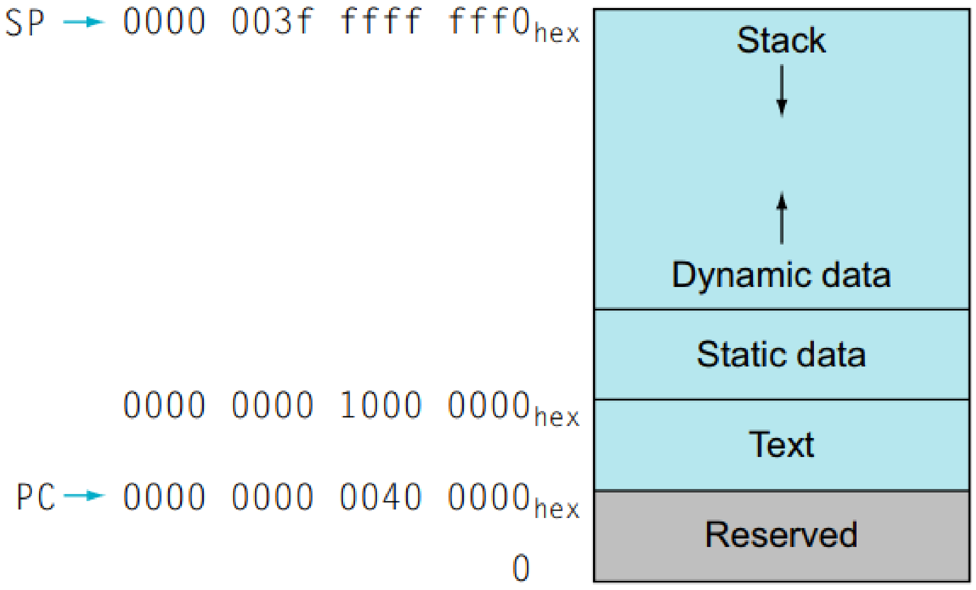
  - n in $a0
  - Result in $a0

Sample code

```
fact:
     addi   sp, sp, -8        #2 items in stack
     sw     ra, 4(sp)         #save return address
     sw     a0, 0(sp)         #and current n
     addi   t0, a0, -1        #n-1 > 0
     bge    t0, zero, L1      #continue
     addi   a0, zero, 1       #the base case, return 1
     addi   sp, sp, 8         #deallocate 2 words
     jr     ra                #and return
L1:  addi   a0, a0, -1        #otherwise reduce n
     jal    fact              #then call fact again
     add    a1, a0, zero      #restore n
     lw     a0, 0(sp)         #and return address
     lw     ra, 4(sp)         #shrink stack
     addi   sp, sp, 8         #value for normal case
     mul    a0, a0, a1        #multiply with n
     jalr   x0, 0(ra)         #and return
```

RISC-V memory configuration

- Program text: stores machine code of program, declared with *.text*
- Static data: data segment, declared with *.data*
- Heap: for dynamic allocation
- Stack: for local variable and dynamic allocation (LIFO)



Accessing characters and string

- String is accessed as array of characters
- Accessing 1-byte characters

```
lb rd, imm(rs1)       #load byte with sign-extension
lbu rd, imm(rs1)      #load byte with zero-extension
sb rs2, 0(rs1)        #store LSB to memory
```

Accessing characters and string

- Accessing 2-byte characters

```
lh rd, imm(rs1)       #load half with sign-extension
lhu rd, imm(rs1)      #load half with zero-extension
sh rs2, 0(rs1)        #store 2 LSB to memory
```

- Example: string copy

```
void strcpy (char x[], char y[])
{
    int i = 0;
    while ((x[i] = y[i]) != ‘\0’)
        i += 1;
}
```

Accessing characters and string

```
#x and y are in a0 and a1, i in s0
strcpy:
    addi sp,sp,-4         # adjust stack for 1 more item
    sw s0, 0(sp)          # save s0
    add s0, zero, zero    # i = 0
L1: add t1, s0, a1        # address of y[i] in t1
    lbu t2, 0(t1)         # t2 = y[i]
    add t3, s0, a0        # address of x[i] in t3
    sb  t2, 0(t3)         # x[i] = y[i]
    beq t2, zero,L2       # if y[i] == 0, go to L2
    addi s0, s0, 1        # i = i + 1
    beq x0, x0, L1        # go to L1
L2: lw s0, 0(sp)          # y[i] == 0: end of string.
                          # Restore old $s0
    addi sp,sp,4          # pop 1 word off stack
    jalr ra               # return
```

Interchange sort function

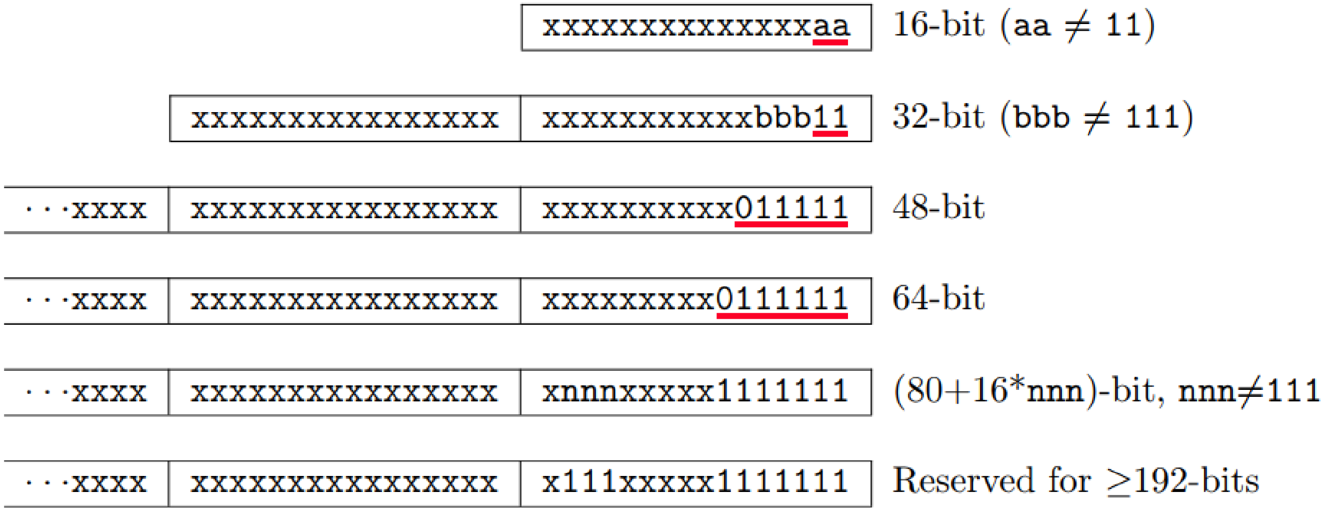
```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1)
    {
        for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j-=1)
        {
            swap(v,j);
        }
    }
}
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

RISC-V Instruction Set Extensions

- The RV31I Instruction Set (that we have learnt so far)
  - Instruction word: 32 bits
  - Only work on integers
  - Supports arithmetic, logic and shift, data transfer, branches
- How about:
  - Other instruction word length?
  - Data other than integers?
  - Additional operations: multiplication, division…?
- Instruction Set Extensions
  - Additional operations and data types
  - Additional formats and customed formats
  - → RISC-V scalable ISA

RISC-V Standard Extensions

- 32-bit instruction extensions
  - “M”: Integer Multiplication and Division Instructions
  - “A”: Atomic (Memory) Instructions
  - “F”: Single-Precision Floating-Point Instructions
  - “D”: Double-Precision Floating-Point Instructions
- 16-bit: “C”: Compressed Instructions

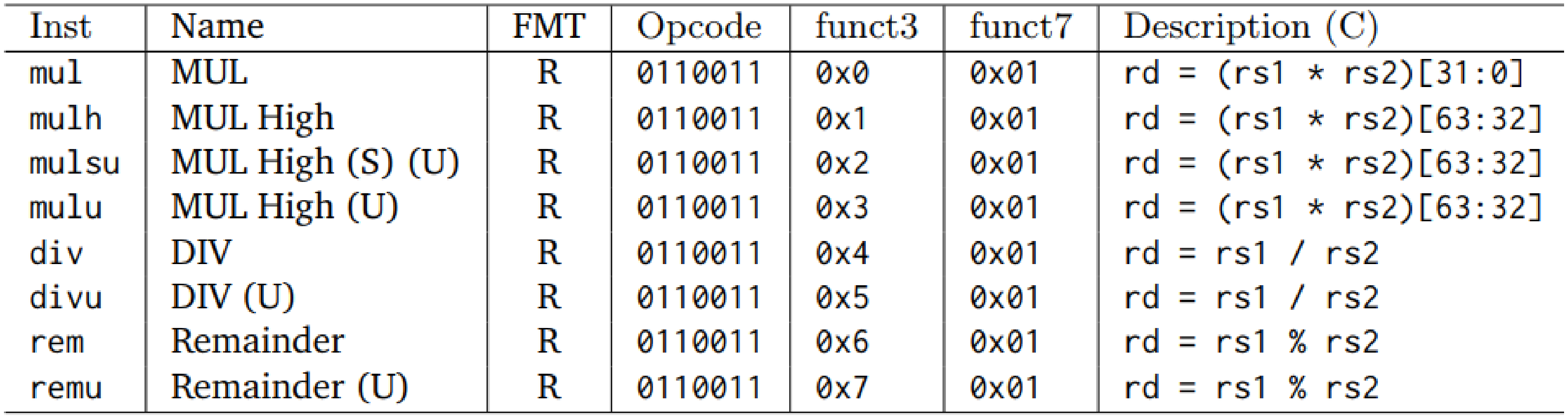


		xxxxxxxxxxxxxxaa	16-bit (aa $\neq$ 11)
	xxxxxxxxxxxxxxxx	xxxxxxxxxxxbbb11	32-bit (bbb $\neq$ 111)
···xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxxx011111	48-bit
···xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxx0111111	64-bit
···xxxx	xxxxxxxxxxxxxxxx	xnnnxxxxx1111111	(80+16*nnn)-bit, nnn$\neq$111
···xxxx	xxxxxxxxxxxxxxxx	x111xxxxx1111111	Reserved for $\geq$192-bits

RISC-V instruction length encoding

RV32M: Integer Multiplication and Division Extension

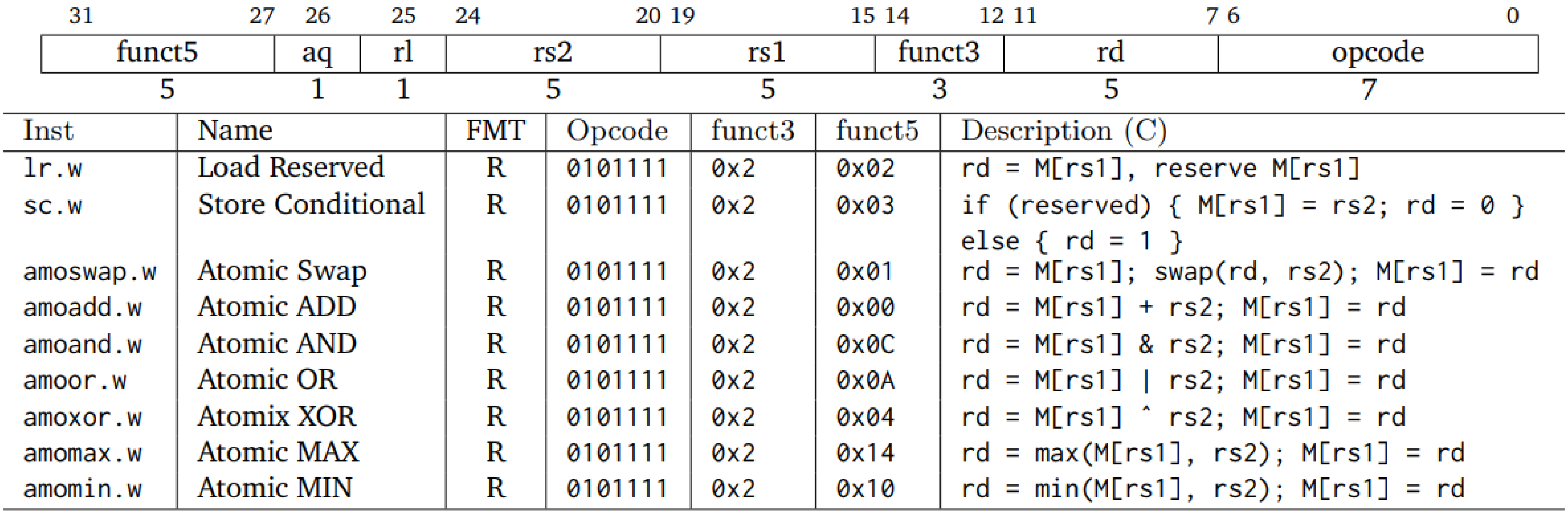
- Support integer multiplication, division (div and rem) operations.
- All are R-format.



Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
mul	MUL	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	MUL High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
mulsu	MUL High (S) (U)	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulu	MUL High (U)	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]
div	DIV	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	DIV (U)	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder (U)	R	0110011	0x7	0x01	rd = rs1 % rs2

RV32A: Atomic Extension

- Support synchronized “atomic” memory access.
  - Load + data op + Store become atomic.
  - Similar to semaphore/mutex in multithread software.
  - Basically R-format, with aq (acquire) and rl (release) bits.



31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7

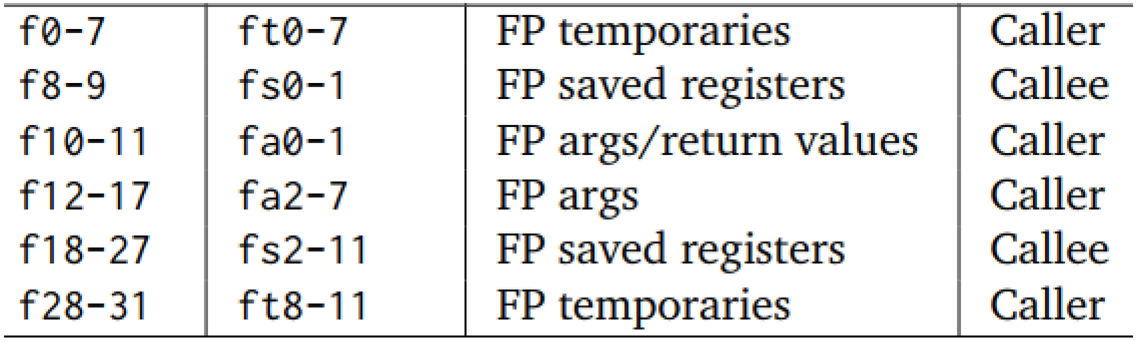
Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
lr.w	Load Reserved	R	0101111	0x2	0x02	rd = M[rs1], reserve M[rs1]

sc.w	Store Conditional	R	0101111	0x2	0x03	if (reserved) { M[rs1] = rs2; rd = 0 }
						else { rd = 1 }

amoswap.w	Atomic Swap	R	0101111	0x2	0x01	rd = M[rs1]; swap(rd, rs2); M[rs1] = rd
amoadd.w	Atomic ADD	R	0101111	0x2	0x00	rd = M[rs1] + rs2; M[rs1] = rd
amoand.w	Atomic AND	R	0101111	0x2	0x0C	rd = M[rs1] & rs2; M[rs1] = rd
amoor.w	Atomic OR	R	0101111	0x2	0x0A	rd = M[rs1] | rs2; M[rs1] = rd
amoxor.w	Atomix XOR	R	0101111	0x2	0x04	rd = M[rs1] ^ rs2; M[rs1] = rd
amomax.w	Atomic MAX	R	0101111	0x2	0x14	rd = max(M[rs1], rs2); M[rs1] = rd
amomin.w	Atomic MIN	R	0101111	0x2	0x10	rd = min(M[rs1], rs2); M[rs1] = rd

RV32F / D Floating-Point Extensions

- Support floating point operations.
  - Additional floating point register file for new data type.
  - Additional instructions to work with the new register file.
  - Additional load/store instructions.
- Data representation and computation are compliant with the IEEE 754-2008 standard (chapter 4).
  - “F”: 32-bit single precision floating point numbers (float in C).
  - “D”: 64-bit double precision floating point numbers (double in C).

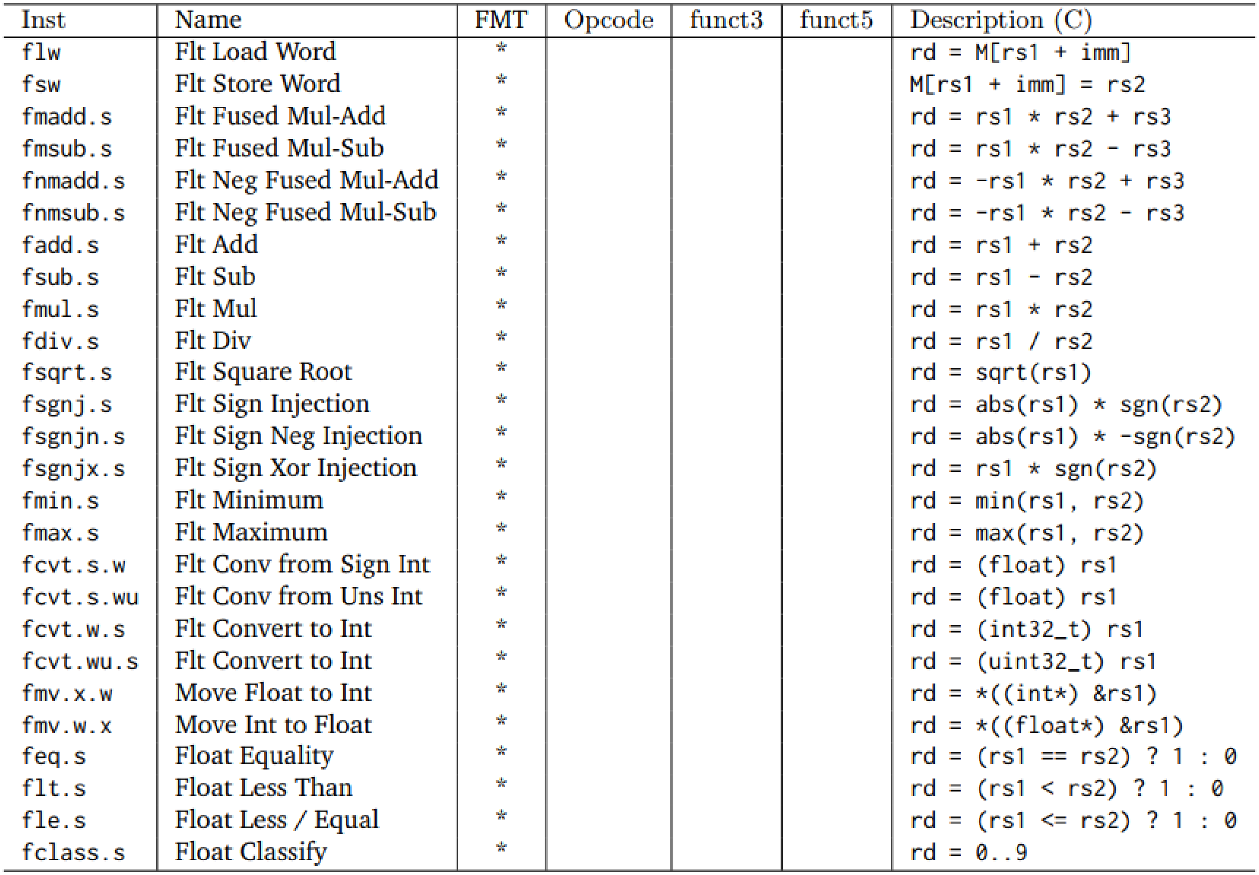


`f0-7`	`ft0-7`	FP temporaries	Caller
`f8-9`	`fs0-1`	FP saved registers	Callee
`f10-11`	`fa0-1`	FP args/return values	Caller
`f12-17`	`fa2-7`	FP args	Caller
`f18-27`	`fs2-11`	FP saved registers	Callee
`f28-31`	`ft8-11`	FP temporaries	Caller

RISC-V floating point register file

RV32F / D Floating-Point Extensions

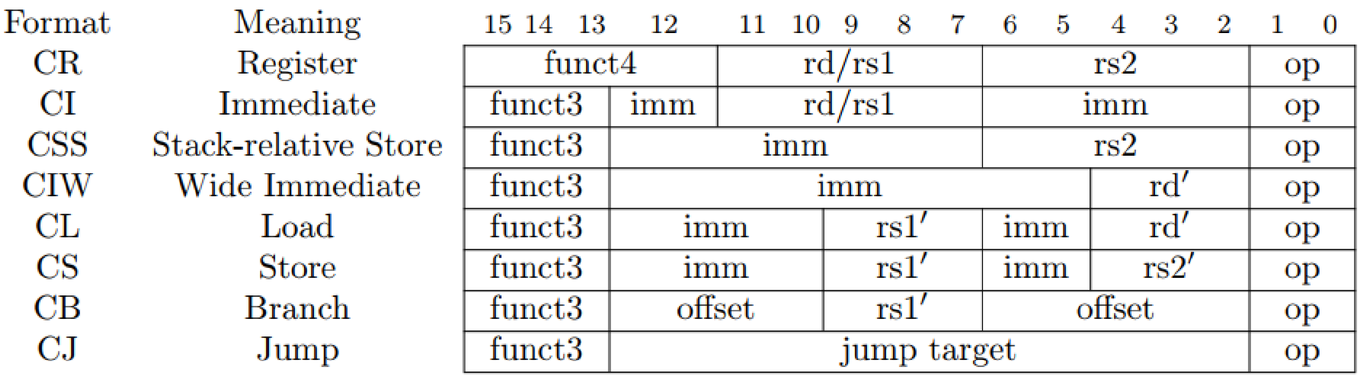
- Single precisision instructions
- .s for single, .d for double



Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
flw	Flt Load Word	*				rd = M[rs1 + imm]
fsw	Flt Store Word	*				M[rs1 + imm] = rs2
fmadd.s	Flt Fused Mul-Add	*				rd = rs1 * rs2 + rs3
fmsub.s	Flt Fused Mul-Sub	*				rd = rs1 * rs2 - rs3
fnmadd.s	Flt Neg Fused Mul-Add	*				rd = -rs1 * rs2 + rs3
fnmsub.s	Flt Neg Fused Mul-Sub	*				rd = -rs1 * rs2 - rs3
fadd.s	Flt Add	*				rd = rs1 + rs2
fsub.s	Flt Sub	*				rd = rs1 - rs2
fmul.s	Flt Mul	*				rd = rs1 * rs2
fdiv.s	Flt Div	*				rd = rs1 / rs2
fsqrt.s	Flt Square Root	*				rd = sqrt(rs1)
fsgnj.s	Flt Sign Injection	*				rd = abs(rs1) * sgn(rs2)
fsgnjn.s	Flt Sign Neg Injection	*				rd = abs(rs1) * -sgn(rs2)
fsgnjx.s	Flt Sign Xor Injection	*				rd = rs1 * sgn(rs2)
fmin.s	Flt Minimum	*				rd = min(rs1, rs2)
fmax.s	Flt Maximum	*				rd = max(rs1, rs2)
fcvt.s.w	Flt Conv from Sign Int	*				rd = (float) rs1
fcvt.s.wu	Flt Conv from Uns Int	*				rd = (float) rs1
fcvt.w.s	Flt Convert to Int	*				rd = (int32_t) rs1
fcvt.wu.s	Flt Convert to Int	*				rd = (uint32_t) rs1
fmv.x.w	Move Float to Int	*				rd = *((int*) &rs1)
fmv.w.x	Move Int to Float	*				rd = *((float*) &rs1)
feq.s	Float Equality	*				rd = (rs1 == rs2) ? 1 : 0
flt.s	Float Less Than	*				rd = (rs1 < rs2) ? 1 : 0
fle.s	Float Less / Equal	*				rd = (rs1 <= rs2) ? 1 : 0
fclass.s	Float Classify	*				rd = 0..9

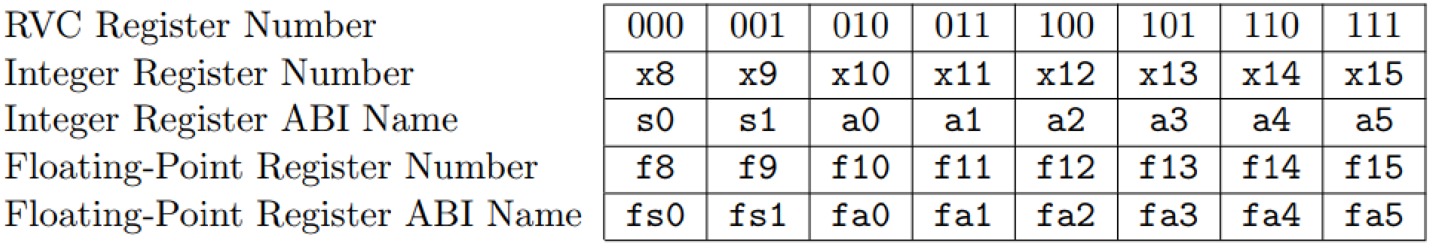
RVC Compressed Extension

- 16-bit length instructions
  - Double code density compared to 32-bit instructions.
  - Limited to most frequently-used instructions/operands.
  - Overall 25% - 30% 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
CB	Branch	funct3	offset		rs1′	offset		op
CJ	Jump	funct3	jump target					op

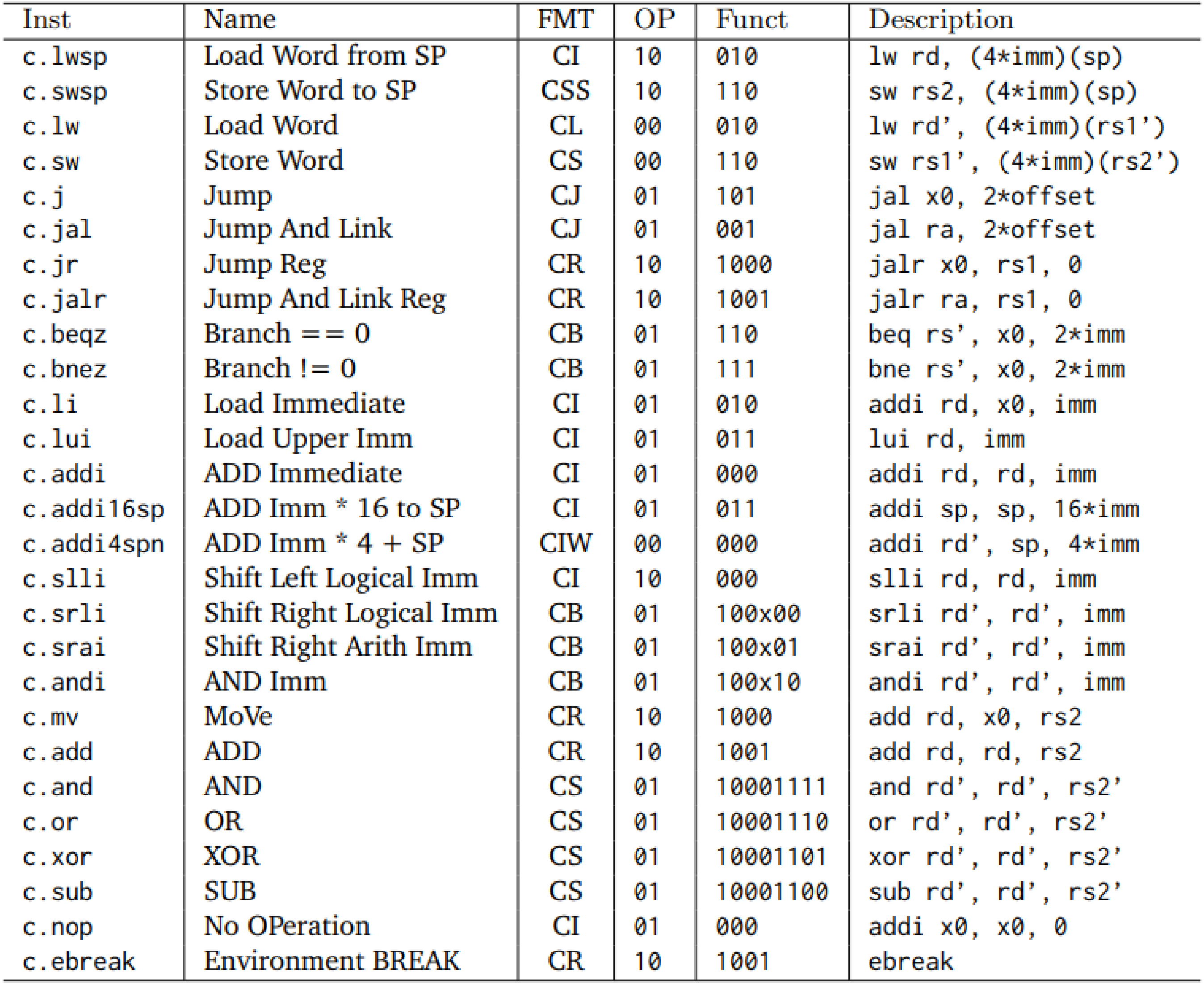
RVC Instruction Formats



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

RVC Registers for CIW, CL, CS, CB instructions

RVC Compressed Instructions



Inst	Name	FMT	OP	Funct	Description
c.lwsp	Load Word from SP	CI	10	010	lw rd, (4*imm)(sp)
c.swsp	Store Word to SP	CSS	10	110	sw rs2, (4*imm)(sp)
c.lw	Load Word	CL	00	010	lw rd', (4*imm)(rs1')
c.sw	Store Word	CS	00	110	sw rs1', (4*imm)(rs2')
c.j	Jump	CJ	01	101	jal x0, 2*offset
c.jal	Jump And Link	CJ	01	001	jal ra, 2*offset
c.jr	Jump Reg	CR	10	1000	jalr x0, rs1, 0
c.jalr	Jump And Link Reg	CR	10	1001	jalr ra, rs1, 0
c.beqz	Branch == 0	CB	01	110	beq rs', x0, 2*imm
c.bnez	Branch != 0	CB	01	111	bne rs', x0, 2*imm
c.li	Load Immediate	CI	01	010	addi rd, x0, imm
c.lui	Load Upper Imm	CI	01	011	lui rd, imm
c.addi	ADD Immediate	CI	01	000	addi rd, rd, imm
c.addi16sp	ADD Imm * 16 to SP	CI	01	011	addi sp, sp, 16*imm
c.addi4spn	ADD Imm * 4 + SP	CIW	00	000	addi rd', sp, 4*imm
c.slli	Shift Left Logical Imm	CI	10	000	slli rd, rd, imm
c.srli	Shift Right Logical Imm	CB	01	100x00	srli rd', rd', imm
c.srai	Shift Right Arith Imm	CB	01	100x01	srai rd', rd', imm
c.andi	AND Imm	CB	01	100x10	andi rd', rd', imm
c.mv	MoVe	CR	10	1000	add rd, x0, rs2
c.add	ADD	CR	10	1001	add rd, rd, rs2
c.and	AND	CS	01	10001111	and rd', rd', rs2'
c.or	OR	CS	01	10001110	or rd', rd', rs2'
c.xor	XOR	CS	01	10001101	xor rd', rd', rs2'
c.sub	SUB	CS	01	10001100	sub rd', rd', rs2'
c.nop	No OPeration	CI	01	000	addi x0, x0, 0
c.ebreak	Environment BREAK	CR	10	1001	ebreak

Further reading

- MIPS instruction set
- ARM instruction set
- x86 instruction set

The end