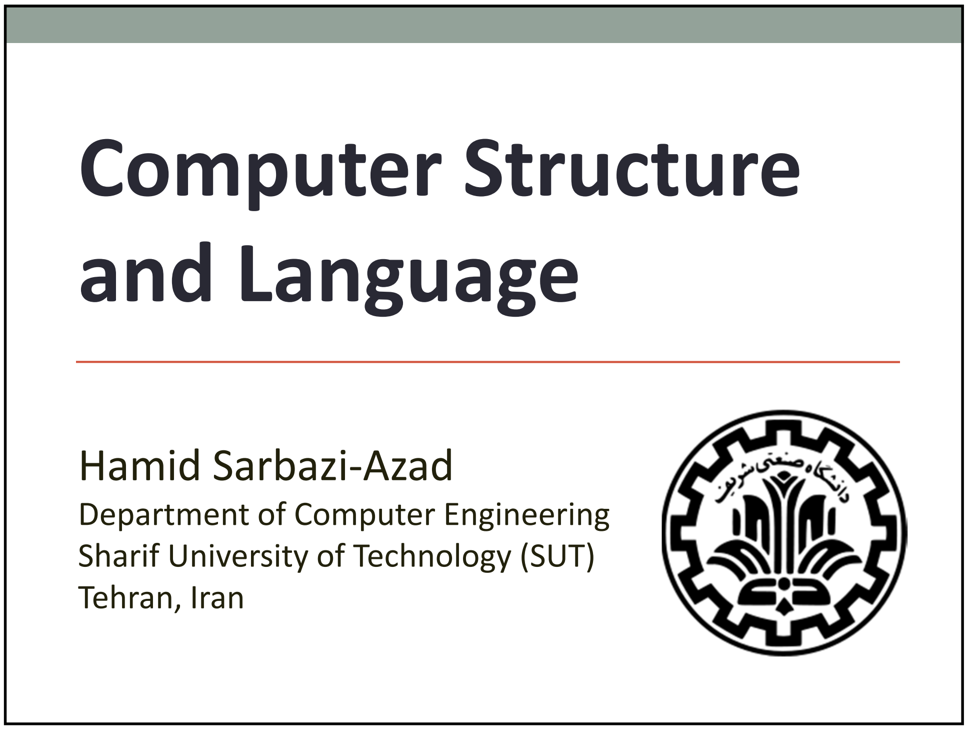
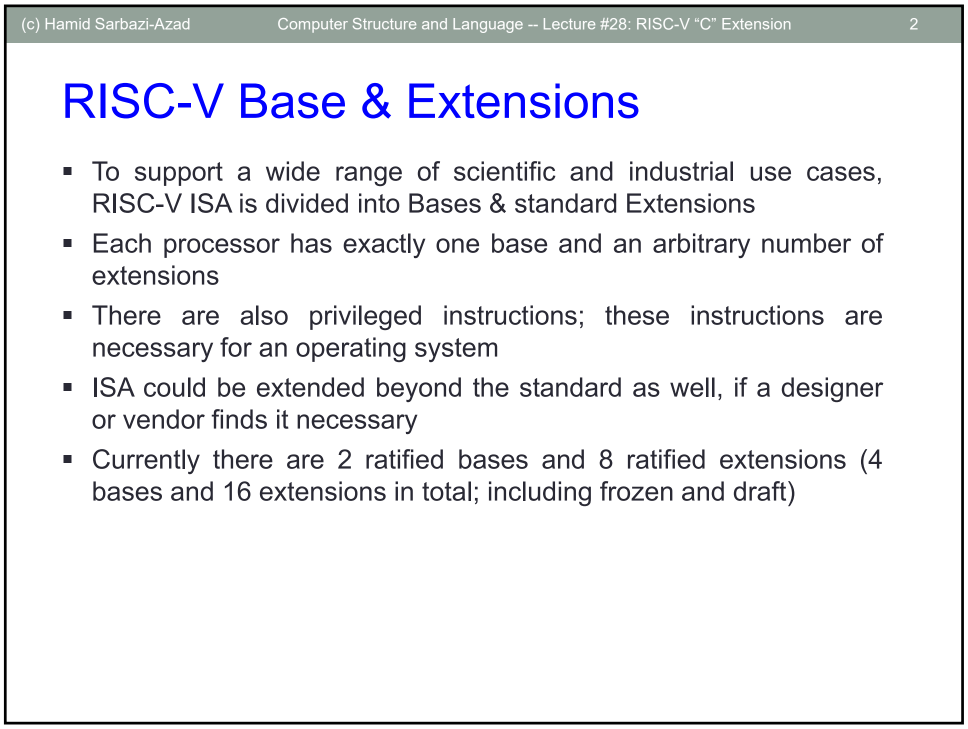


Computer Structure and Language

Hamid Sarbazi-Azad
Department of Computer Engineering
Sharif University of Technology (SUT)
Tehran, Iran

1



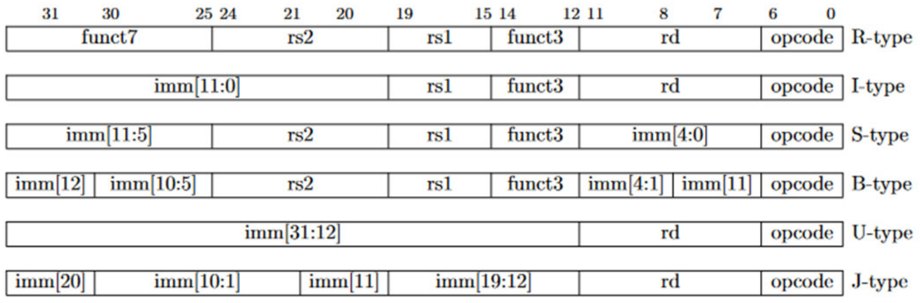
RISC-V Base & Extensions

- To support a wide range of scientific and industrial use cases, RISC-V ISA is divided into Bases & standard Extensions
- Each processor has exactly one base and an arbitrary number of extensions
- There are also privileged instructions; these instructions are necessary for an operating system
- ISA could be extended beyond the standard as well, if a designer or vendor finds it necessary
- Currently there are 2 ratified bases and 8 ratified extensions (4 bases and 16 extensions in total; including frozen and draft)

2

RV32I

- Has 6 instruction formats



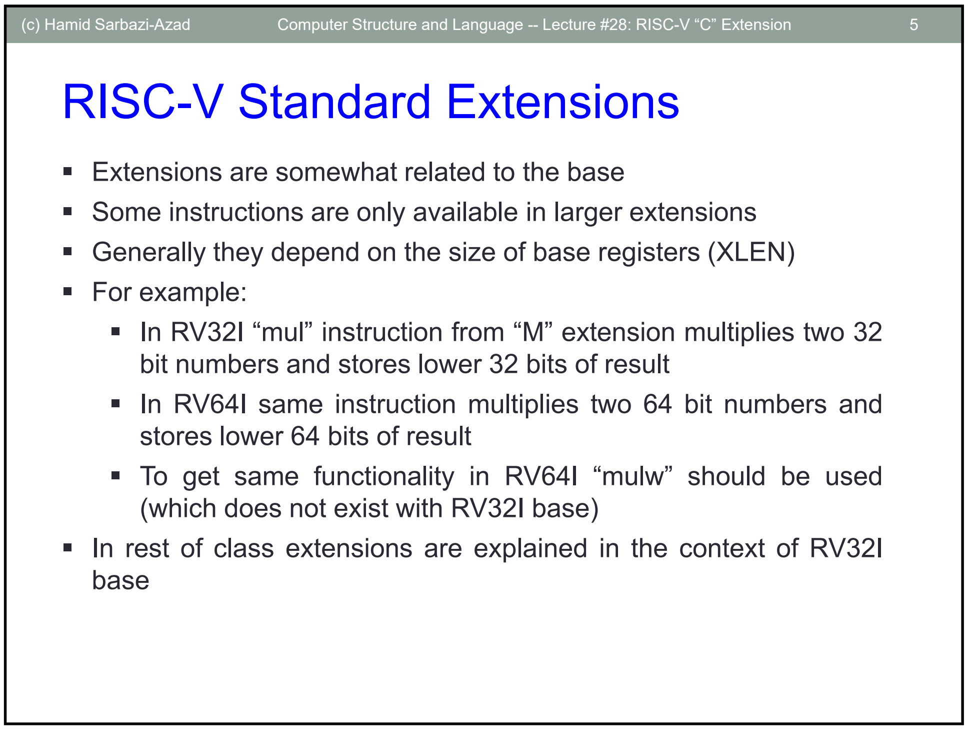
31	30 … 25	24 … 21	20	19 … 15	14 … 12	11 … 8	7	6 … 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]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

3

RISC-V Standard Extensions

- There are 8 ratified standard extensions (as of Dec. 2023)
- Each extension provides a specific functionality that is missing from base and other extensions
- Not all extensions are as useful as others
- Most useful extensions are grouped into “G” or “GC” subset
  - “M” for integer multiplication and division
  - “F” & “D” for single and double precision floating point operations
  - “A” for atomic operations
  - “C” for compressed instructions

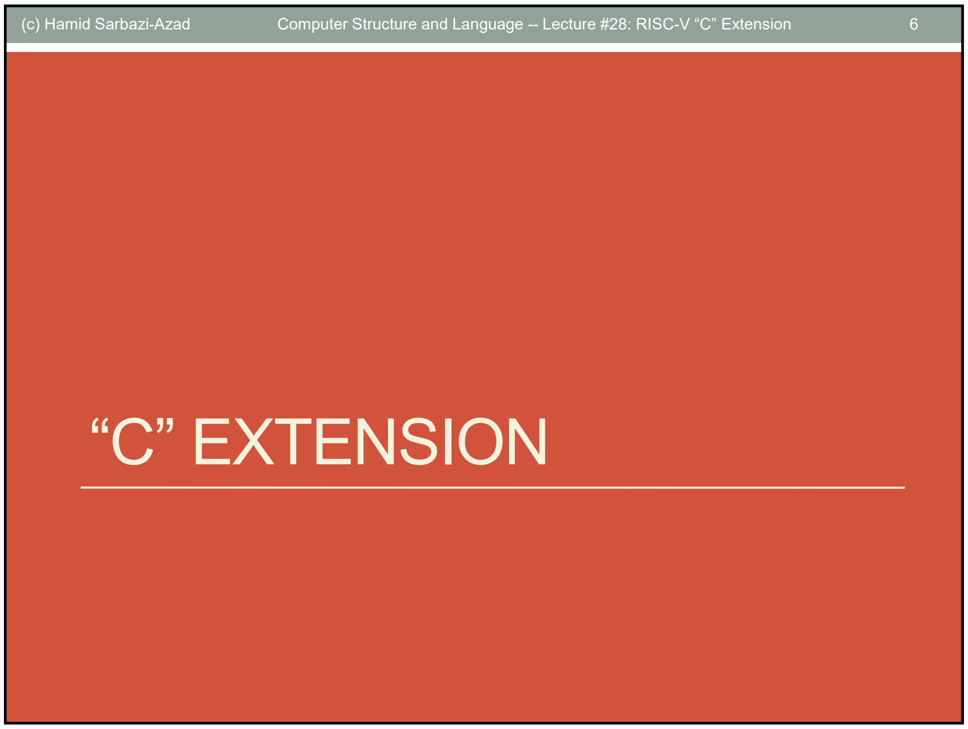
4



RISC-V Standard Extensions

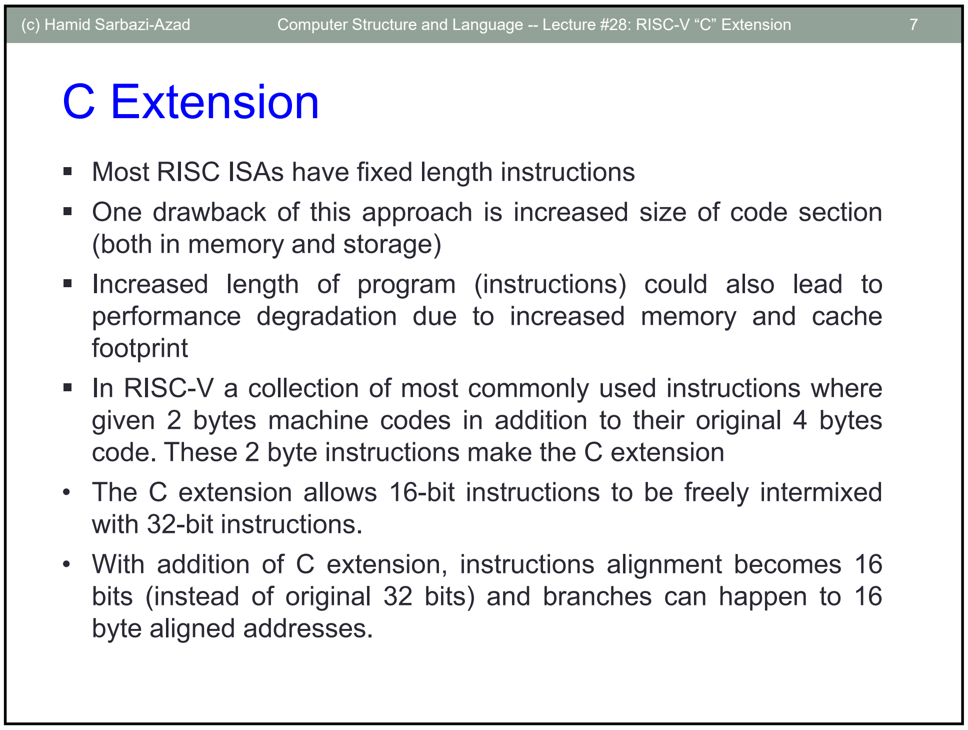
- Extensions are somewhat related to the base
- Some instructions are only available in larger extensions
- Generally they depend on the size of base registers (XLEN)
- For example:
  - In RV32I "mul" instruction from "M" extension multiplies two 32 bit numbers and stores lower 32 bits of result
  - In RV64I same instruction multiplies two 64 bit numbers and stores lower 64 bits of result
  - To get same functionality in RV64I "mulw" should be used (which does not exist with RV32I base)
- In rest of class extensions are explained in the context of RV32I base

5



"C" EXTENSION

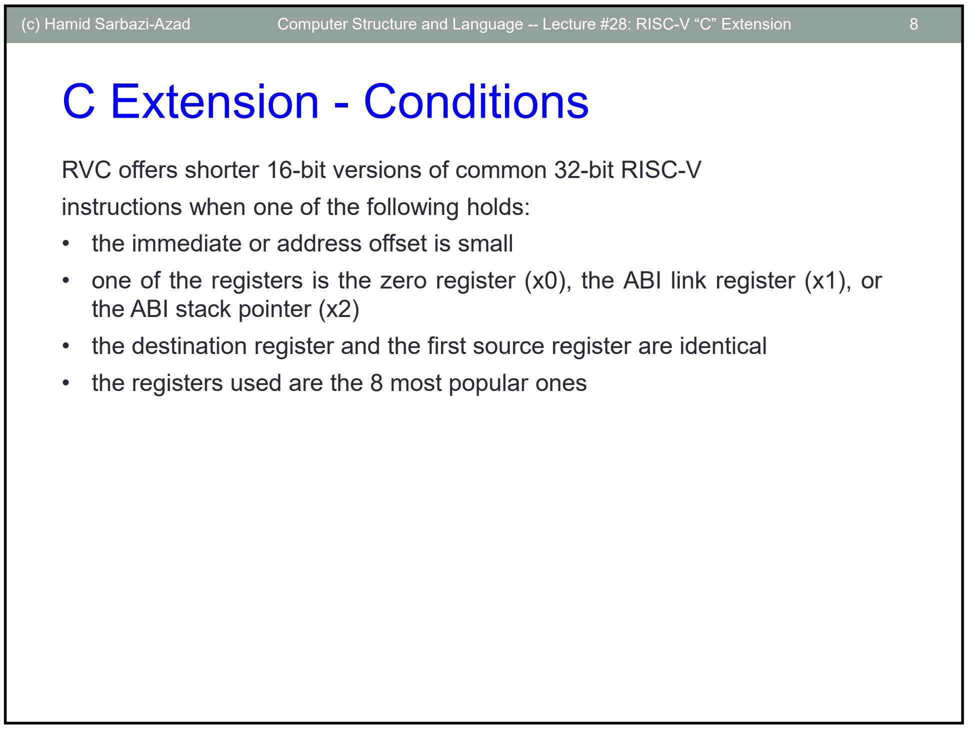
6



C Extension

- Most RISC ISAs have fixed length instructions
- One drawback of this approach is increased size of code section (both in memory and storage)
- Increased length of program (instructions) could also lead to performance degradation due to increased memory and cache footprint
- In RISC-V a collection of most commonly used instructions where given 2 bytes machine codes in addition to their original 4 bytes code. These 2 byte instructions make the C extension
- The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions.
- With addition of C extension, instructions alignment becomes 16 bits (instead of original 32 bits) and branches can happen to 16 byte aligned addresses.

7



C Extension - Conditions

RVC offers shorter 16-bit versions of common 32-bit RISC-V instructions when one of the following holds:

- the immediate or address offset is small
- one of the registers is the zero register (x0), the ABI link register (x1), or the ABI stack pointer (x2)
- the destination register and the first source register are identical
- the registers used are the 8 most popular ones

8

C Extension

- Typically, 50%–60% of the RISC-V instructions in a program can be replaced with RVC instructions, resulting in a 25%–30% code-size reduction.
- RVC was designed under the constraint that each RVC instruction expands into a single 32-bit instruction in either the base ISA (RV32I/E, RV64I, or RV128I) or the F and D standard extensions where present.
  - Potentially reduces hardware design complexity
  - Allows assembler and linker to perform compression in absence of compression aware compiler; however compression aware compiler can still offer better compression.

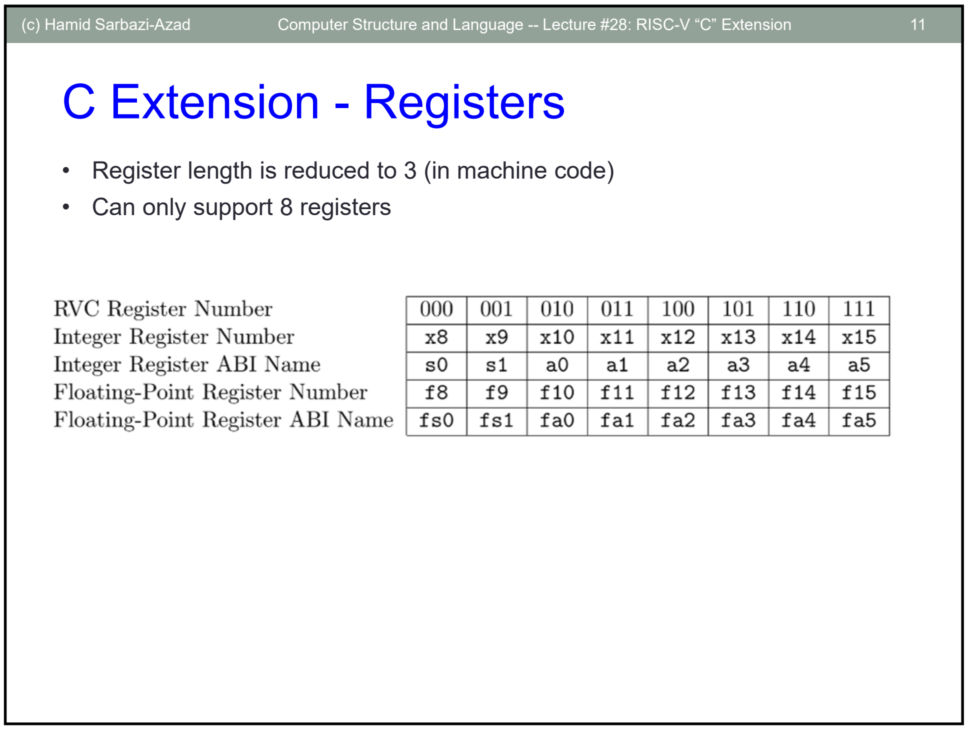
9

C Extension – Instruction formats

- Instruction come in 9 formats

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

10

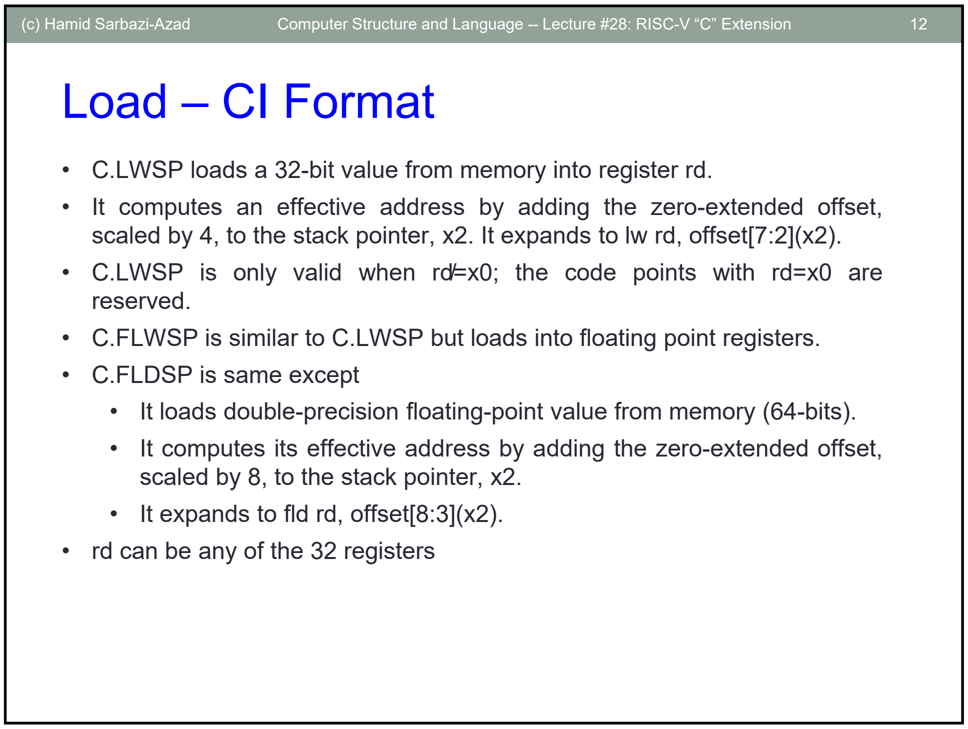


C Extension - Registers

- Register length is reduced to 3 (in machine code)
- Can only support 8 registers

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

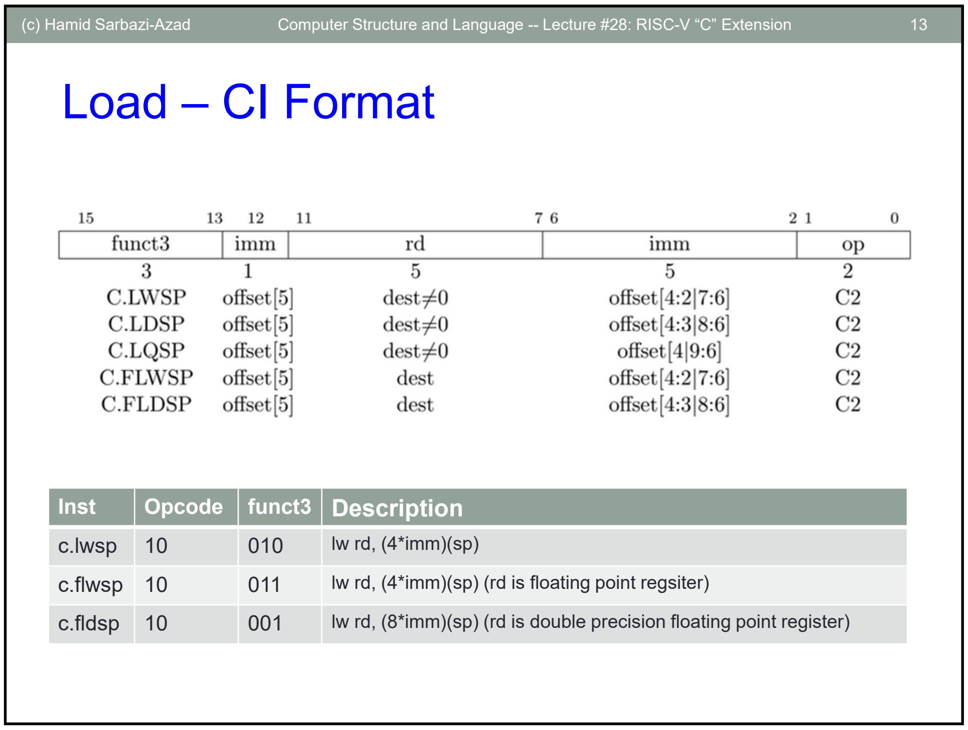
11



Load – CI Format

- C.LWSP loads a 32-bit value from memory into register rd.
- It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2. It expands to lw rd, offset[7:2](x2).
- C.LWSP is only valid when rd≠x0; the code points with rd=x0 are reserved.
- C.FLWSP is similar to C.LWSP but loads into floating point registers.
- C.FLDSP is same except
  - It loads double-precision floating-point value from memory (64-bits).
  - It computes its effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, x2.
  - It expands to fld rd, offset[8:3](x2).
- rd can be any of the 32 registers

12

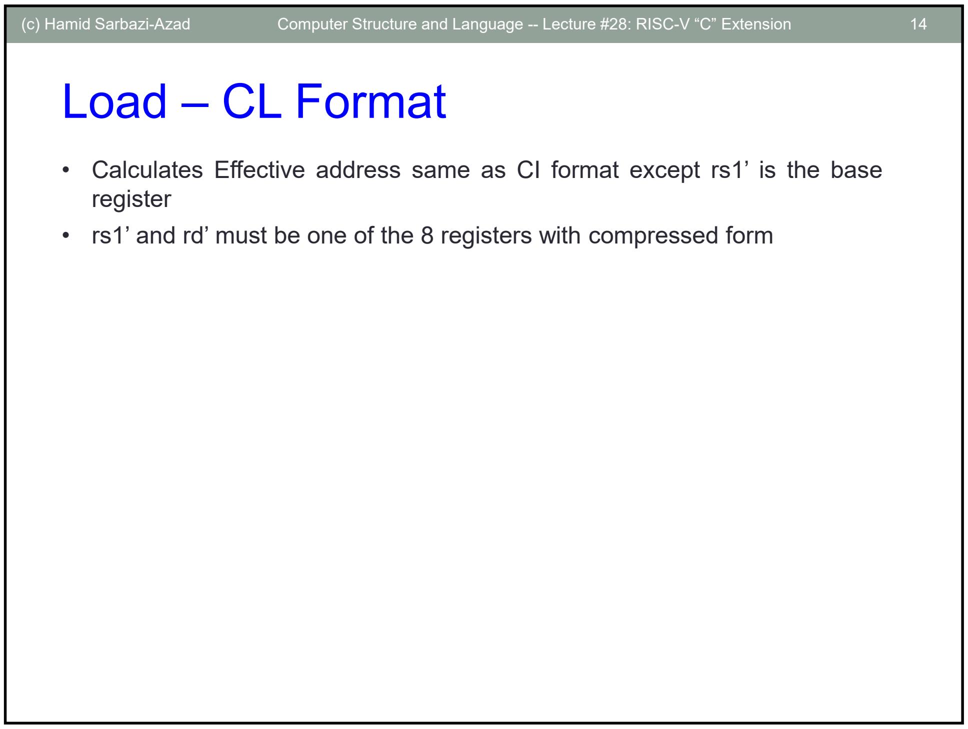


Load – CI Format

15 13	12	11 7	6 2	1 0
funct3	imm	rd	imm	op
3	1	5	5	2
C.LWSP	offset[5]	dest≠0	offset[4:2|7:6]	C2
C.LDSP	offset[5]	dest≠0	offset[4:3|8:6]	C2
C.LQSP	offset[5]	dest≠0	offset[4|9:6]	C2
C.FLWSP	offset[5]	dest	offset[4:2|7:6]	C2
C.FLDSP	offset[5]	dest	offset[4:3|8:6]	C2

Inst	Opcode	funct3	Description
c.lwsp	10	010	lw rd, (4*imm)(sp)
c.flwsp	10	011	lw rd, (4*imm)(sp) (rd is floating point regsiter)
c.fldsp	10	001	lw rd, (8*imm)(sp) (rd is double precision floating point register)

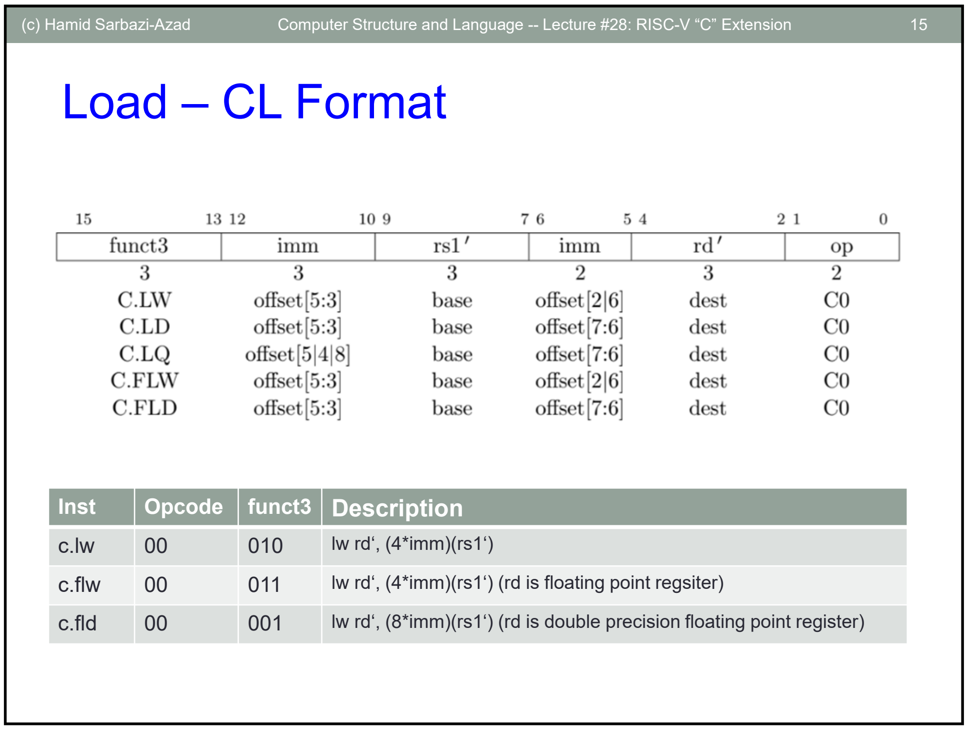
13



Load – CL Format

- Calculates Effective address same as CI format except rs1’ is the base register
- rs1’ and rd’ must be one of the 8 registers with compressed form

14

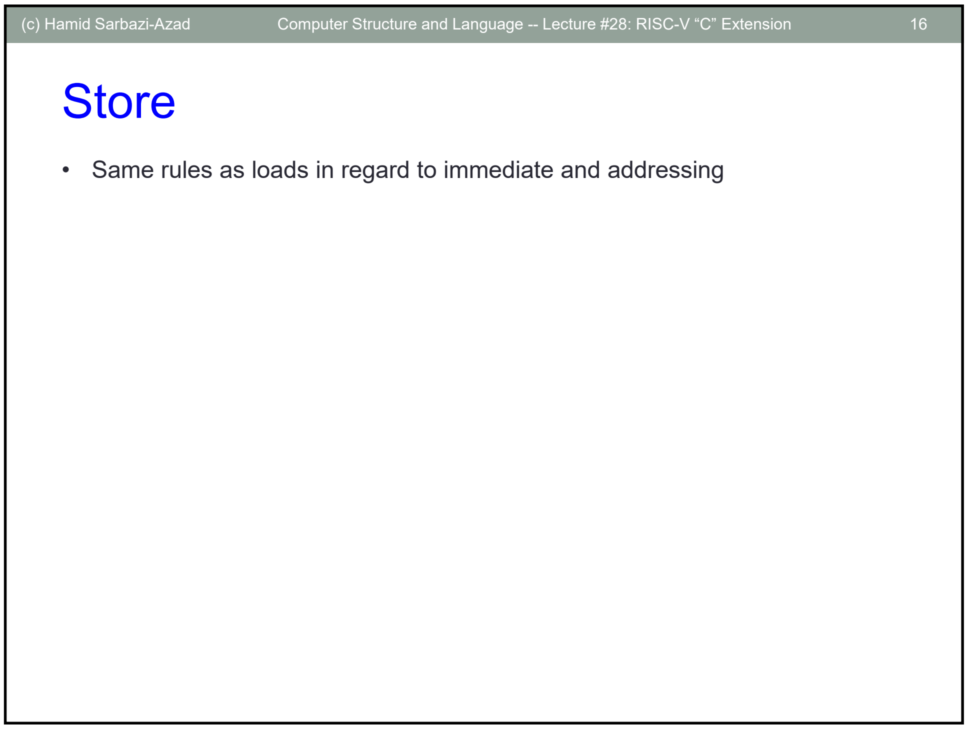


Load – CL Format

15 13	12 10	9 7	6 5	4 2	1 0
funct3	imm	rs1′	imm	rd′	op
3	3	3	2	3	2
C.LW	offset[5:3]	base	offset[2|6]	dest	C0
C.LD	offset[5:3]	base	offset[7:6]	dest	C0
C.LQ	offset[5|4|8]	base	offset[7:6]	dest	C0
C.FLW	offset[5:3]	base	offset[2|6]	dest	C0
C.FLD	offset[5:3]	base	offset[7:6]	dest	C0

Inst	Opcode	funct3	Description
c.lw	00	010	lw rd', (4*imm)(rs1')
c.flw	00	011	lw rd', (4*imm)(rs1') (rd is floating point regsiter)
c.fld	00	001	lw rd', (8*imm)(rs1') (rd is double precision floating point register)

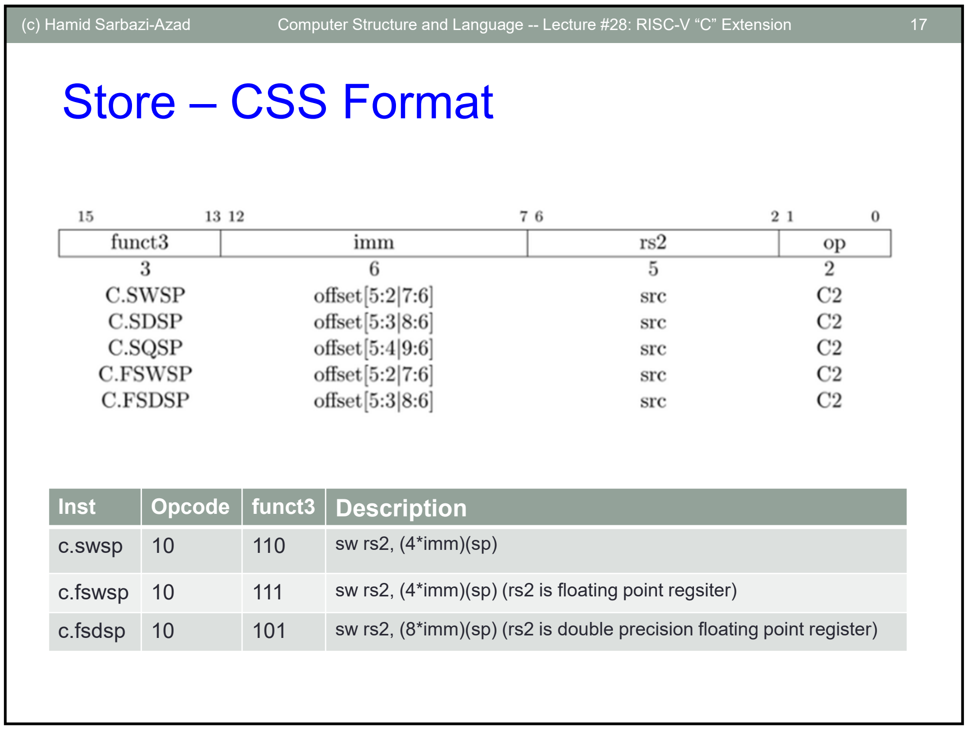
15



Store

- Same rules as loads in regard to immediate and addressing

16

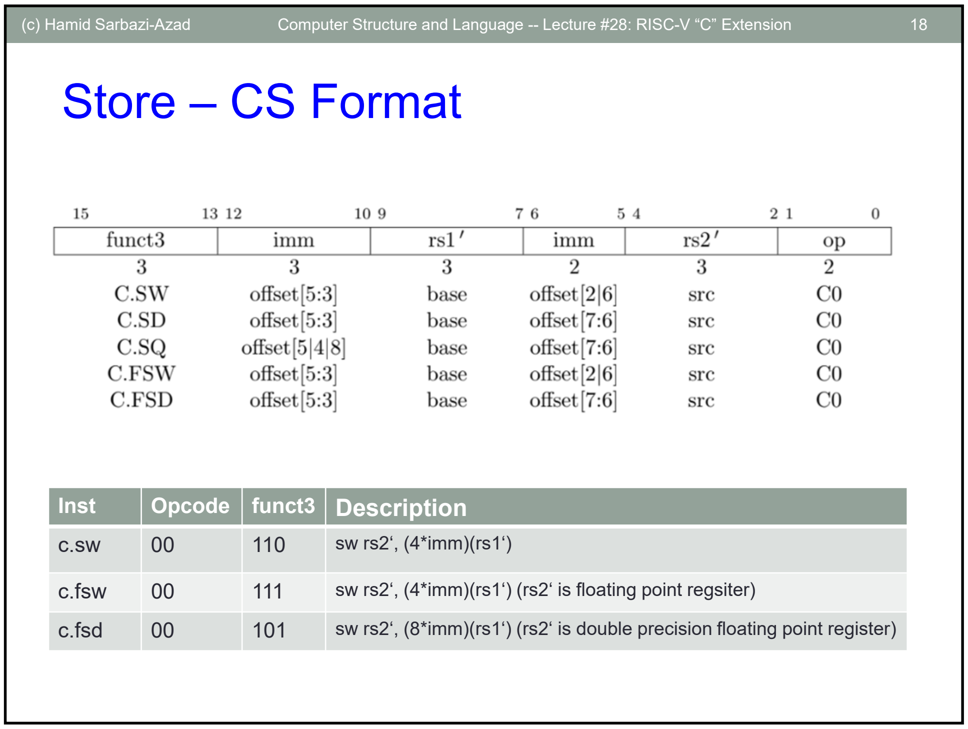


Store – CSS Format

15–13	12–7	6–2	1–0
funct3	imm	rs2	op
3	6	5	2
C.SWSP	offset[5:2|7:6]	src	C2
C.SDSP	offset[5:3|8:6]	src	C2
C.SQSP	offset[5:4|9:6]	src	C2
C.FSWSP	offset[5:2|7:6]	src	C2
C.FSDSP	offset[5:3|8:6]	src	C2

Inst	Opcode	funct3	Description
c.swsp	10	110	sw rs2, (4*imm)(sp)
c.fswsp	10	111	sw rs2, (4*imm)(sp) (rs2 is floating point regsiter)
c.fsdsp	10	101	sw rs2, (8*imm)(sp) (rs2 is double precision floating point register)

17

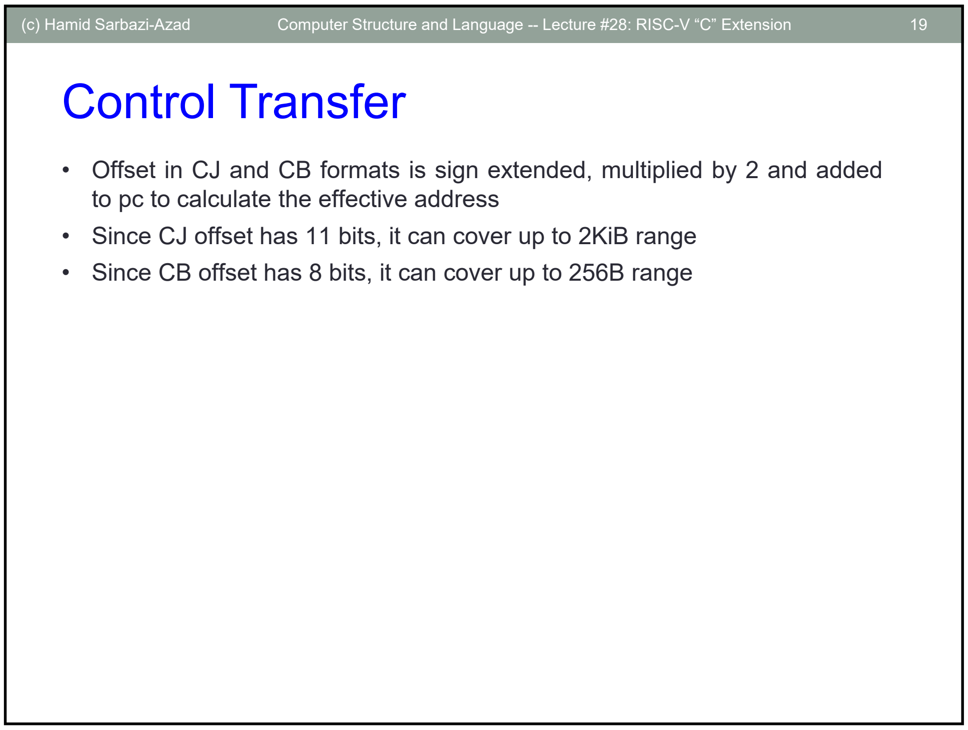


Store – CS Format

15–13	12–10	9–7	6–5	4–2	1–0
funct3	imm	rs1′	imm	rs2′	op
3	3	3	2	3	2
C.SW	offset[5:3]	base	offset[2|6]	src	C0
C.SD	offset[5:3]	base	offset[7:6]	src	C0
C.SQ	offset[5|4|8]	base	offset[7:6]	src	C0
C.FSW	offset[5:3]	base	offset[2|6]	src	C0
C.FSD	offset[5:3]	base	offset[7:6]	src	C0

Inst	Opcode	funct3	Description
c.sw	00	110	sw rs2‘, (4*imm)(rs1‘)
c.fsw	00	111	sw rs2‘, (4*imm)(rs1‘) (rs2‘ is floating point regsiter)
c.fsd	00	101	sw rs2‘, (8*imm)(rs1‘) (rs2‘ is double precision floating point register)

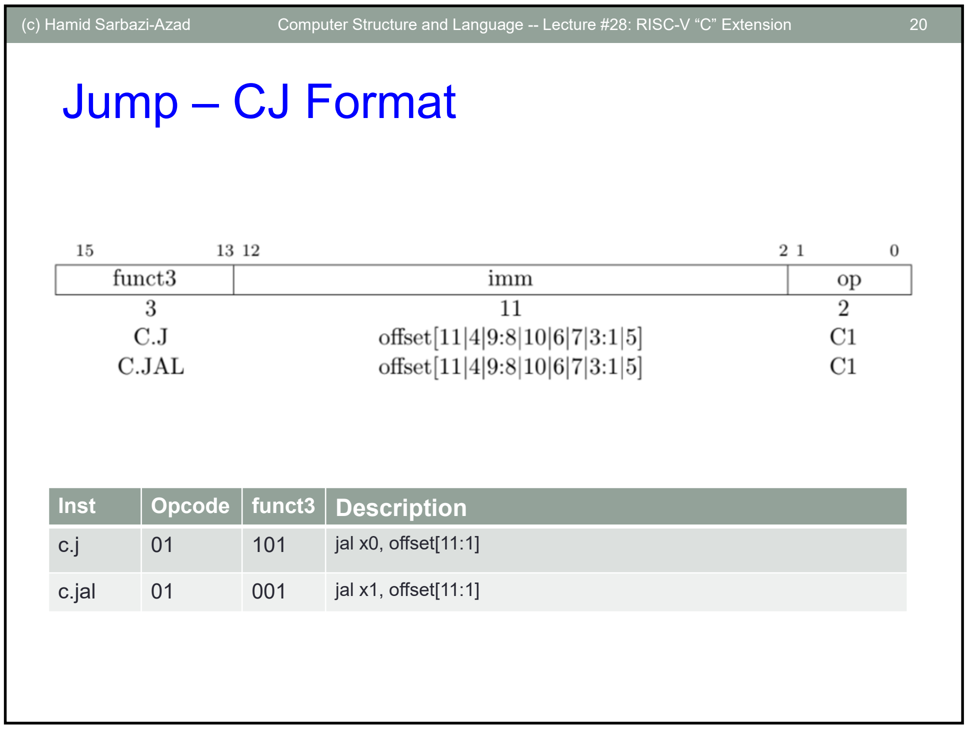
18



Control Transfer

- Offset in CJ and CB formats is sign extended, multiplied by 2 and added to pc to calculate the effective address
- Since CJ offset has 11 bits, it can cover up to 2KiB range
- Since CB offset has 8 bits, it can cover up to 256B range

19

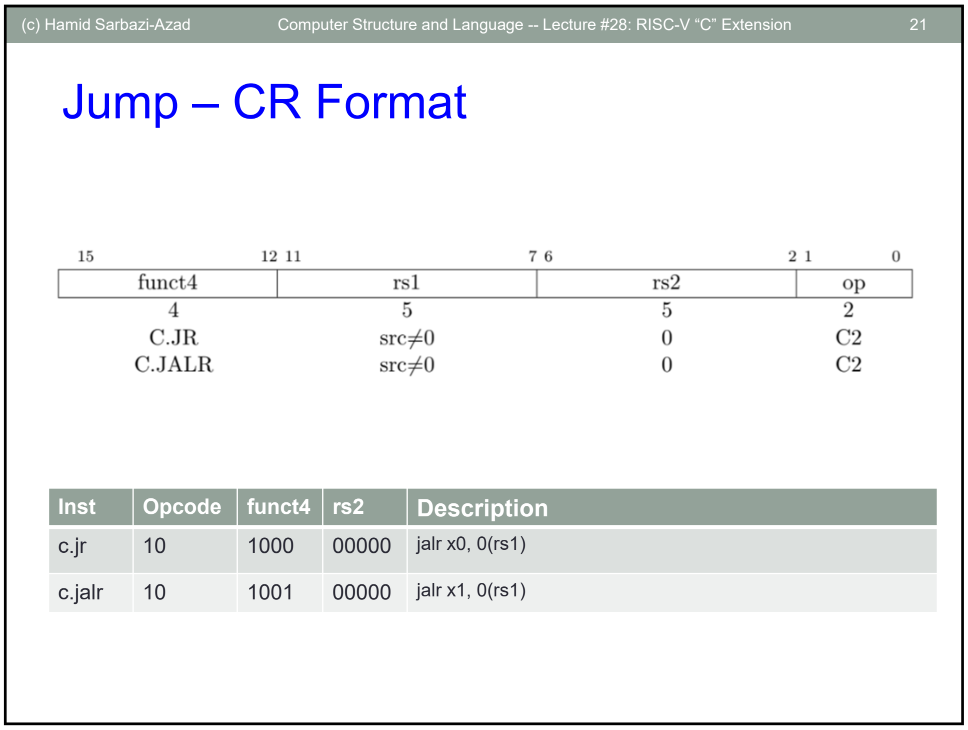


Jump – CJ Format

15 13	12 2	1 0
funct3	imm	op
3	11	2
C.J	offset[11|4|9:8|10|6|7|3:1|5]	C1
C.JAL	offset[11|4|9:8|10|6|7|3:1|5]	C1

Inst	Opcode	funct3	Description
c.j	01	101	jal x0, offset[11:1]
c.jal	01	001	jal x1, offset[11:1]

20

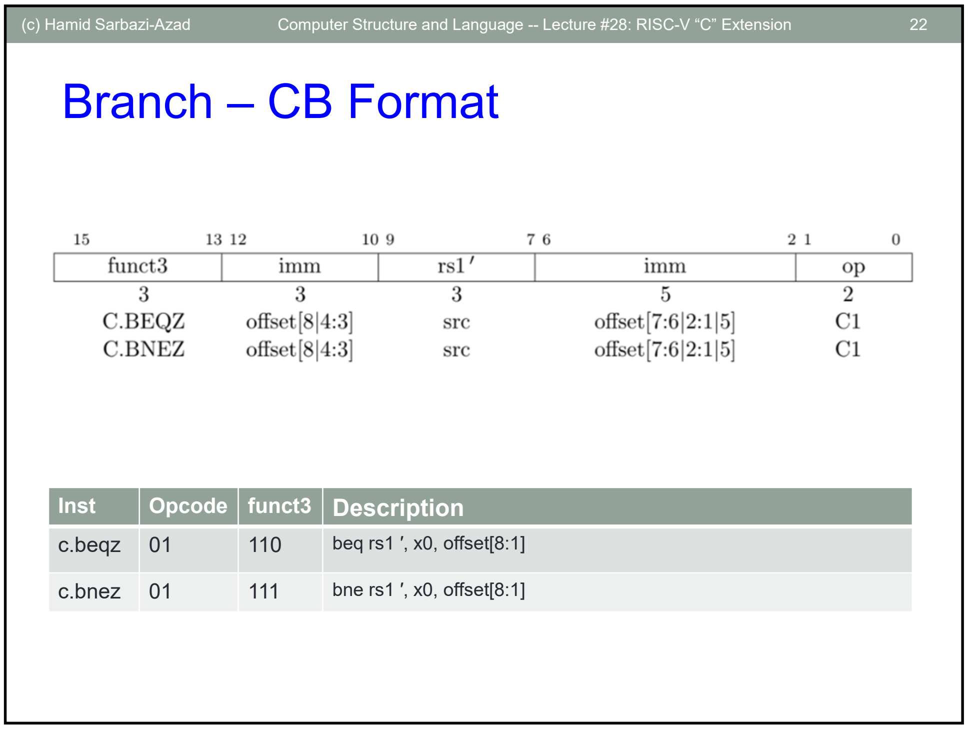


Jump – CR Format

15 – 12	11 – 7	6 – 2	1 – 0
funct4	rs1	rs2	op
4	5	5	2
C.JR	src≠0	0	C2
C.JALR	src≠0	0	C2

Inst	Opcode	funct4	rs2	Description
c.jr	10	1000	00000	jalr x0, 0(rs1)
c.jalr	10	1001	00000	jalr x1, 0(rs1)

21

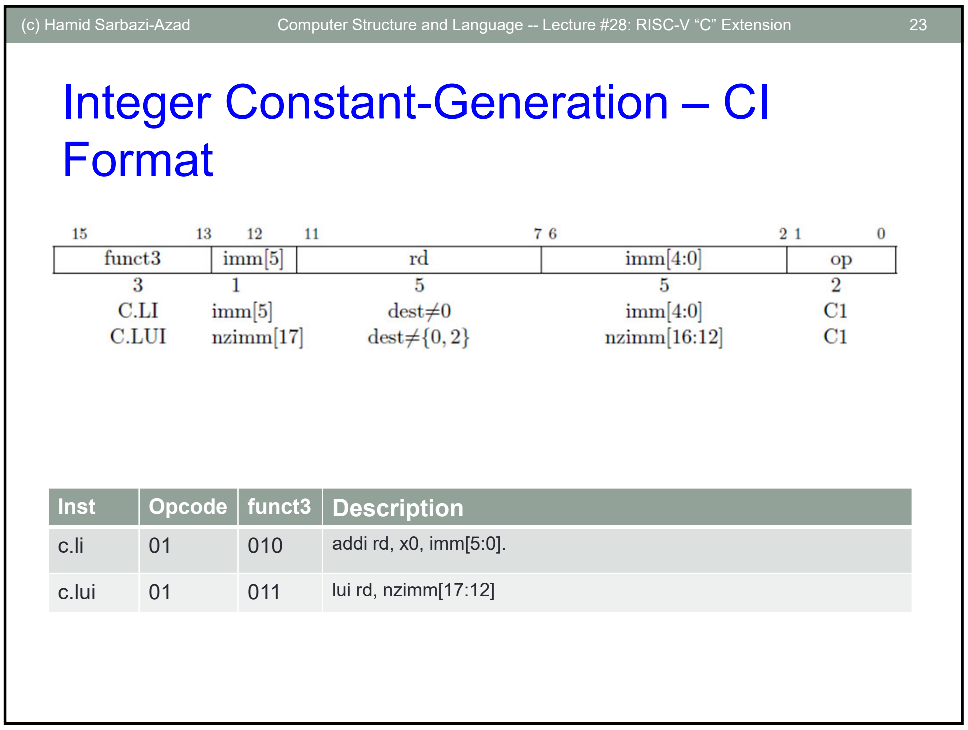


Branch – CB Format

15 – 13	12 – 10	9 – 7	6 – 2	1 – 0
funct3	imm	rs1′	imm	op
3	3	3	5	2
C.BEQZ	offset[8|4:3]	src	offset[7:6|2:1|5]	C1
C.BNEZ	offset[8|4:3]	src	offset[7:6|2:1|5]	C1

Inst	Opcode	funct3	Description
c.beqz	01	110	beq rs1 ′, x0, offset[8:1]
c.bnez	01	111	bne rs1 ′, x0, offset[8:1]

22

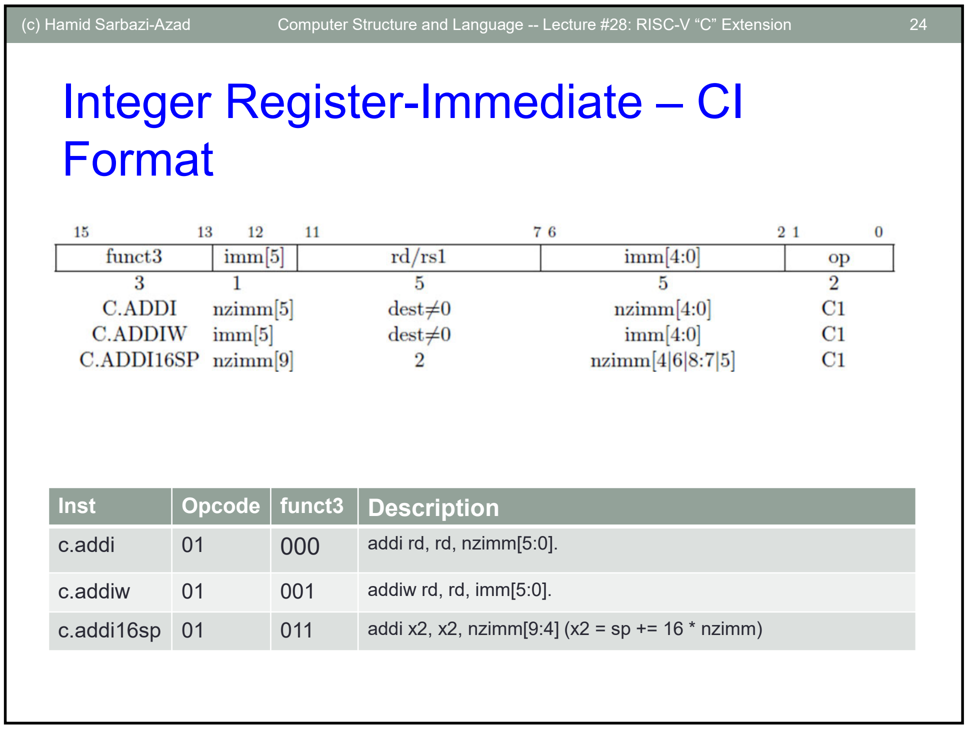


Integer Constant-Generation – CI Format

15–13	12	11–7	6–2	1–0
funct3	imm[5]	rd	imm[4:0]	op
3	1	5	5	2
C.LI	imm[5]	dest≠0	imm[4:0]	C1
C.LUI	nzimm[17]	dest≠{0, 2}	nzimm[16:12]	C1

Inst	Opcode	funct3	Description
c.li	01	010	addi rd, x0, imm[5:0].
c.lui	01	011	lui rd, nzimm[17:12]

23

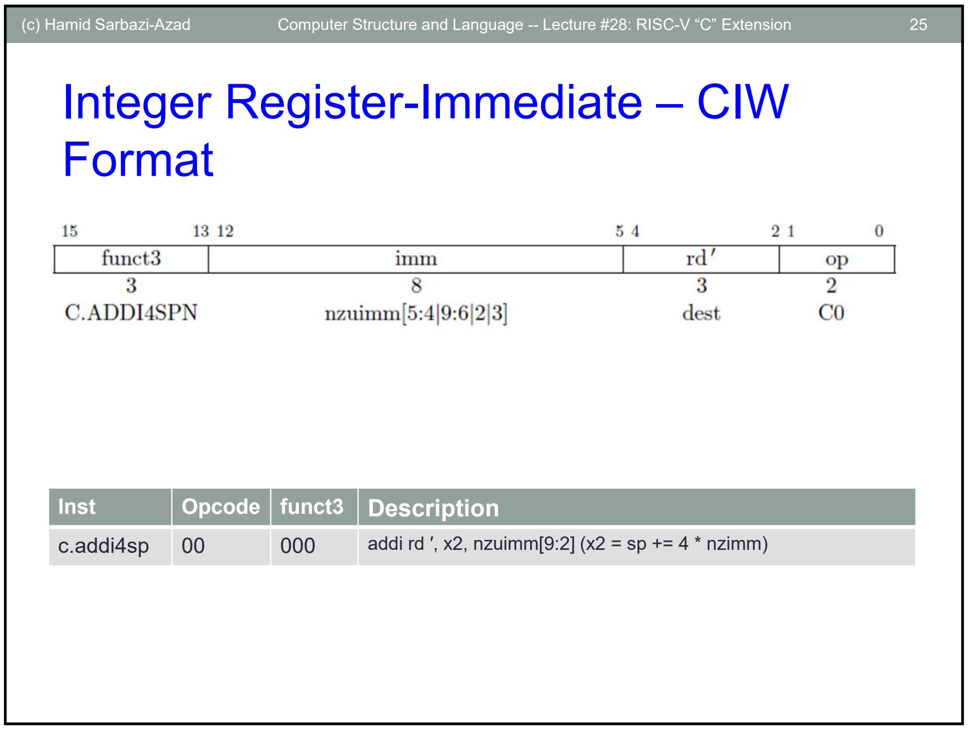


Integer Register-Immediate – CI Format

15–13	12	11–7	6–2	1–0
funct3	imm[5]	rd/rs1	imm[4:0]	op
3	1	5	5	2
C.ADDI	nzimm[5]	dest≠0	nzimm[4:0]	C1
C.ADDIW	imm[5]	dest≠0	imm[4:0]	C1
C.ADDI16SP	nzimm[9]	2	nzimm[4|6|8:7|5]	C1

Inst	Opcode	funct3	Description
c.addi	01	000	addi rd, rd, nzimm[5:0].
c.addiw	01	001	addiw rd, rd, imm[5:0].
c.addi16sp	01	011	addi x2, x2, nzimm[9:4] (x2 = sp += 16 * nzimm)

24

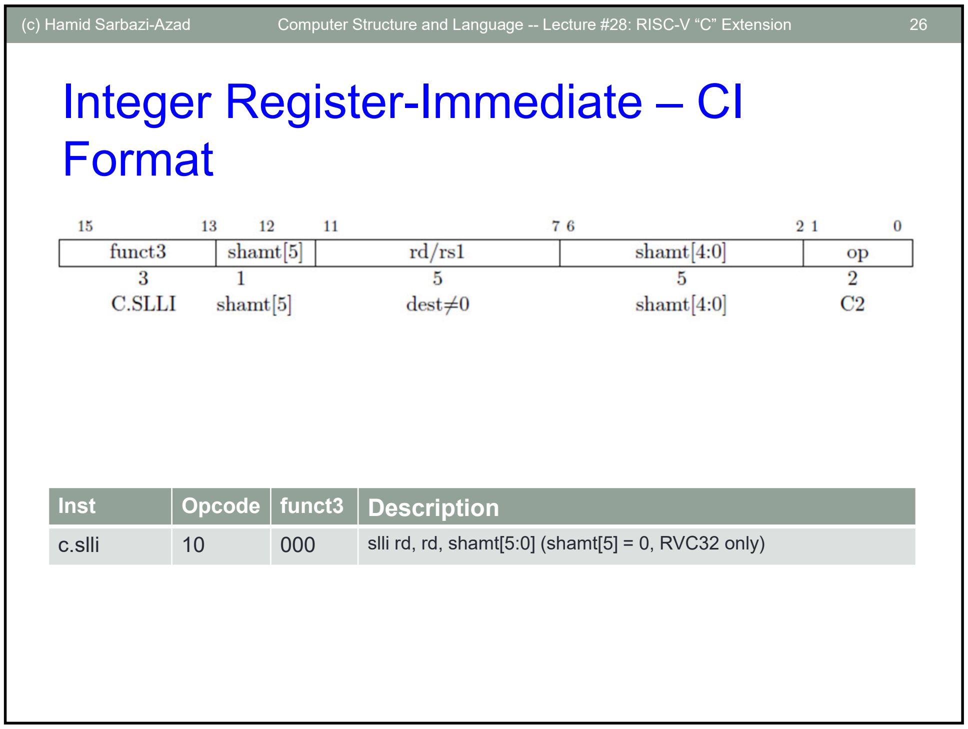


Integer Register-Immediate – CIW Format

15 13	12 5	4 2	1 0
funct3	imm	rd′	op
3	8	3	2
C.ADDI4SPN	nzuimm[5:4|9:6|2|3]	dest	C0

Inst	Opcode	funct3	Description
c.addi4sp	00	000	addi rd ′, x2, nzuimm[9:2] (x2 = sp += 4 * nzimm)

25

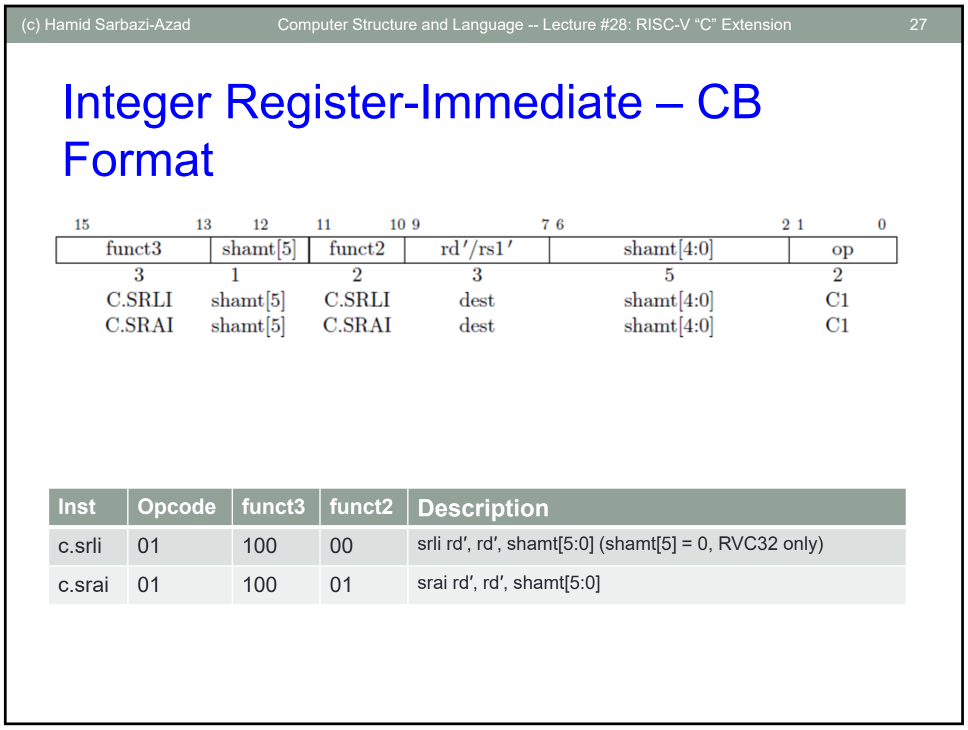


Integer Register-Immediate – CI Format

15 13	12	11 7	6 2	1 0
funct3	shamt[5]	rd/rs1	shamt[4:0]	op
3	1	5	5	2
C.SLLI	shamt[5]	dest≠0	shamt[4:0]	C2

Inst	Opcode	funct3	Description
c.slli	10	000	slli rd, rd, shamt[5:0] (shamt[5] = 0, RVC32 only)

26

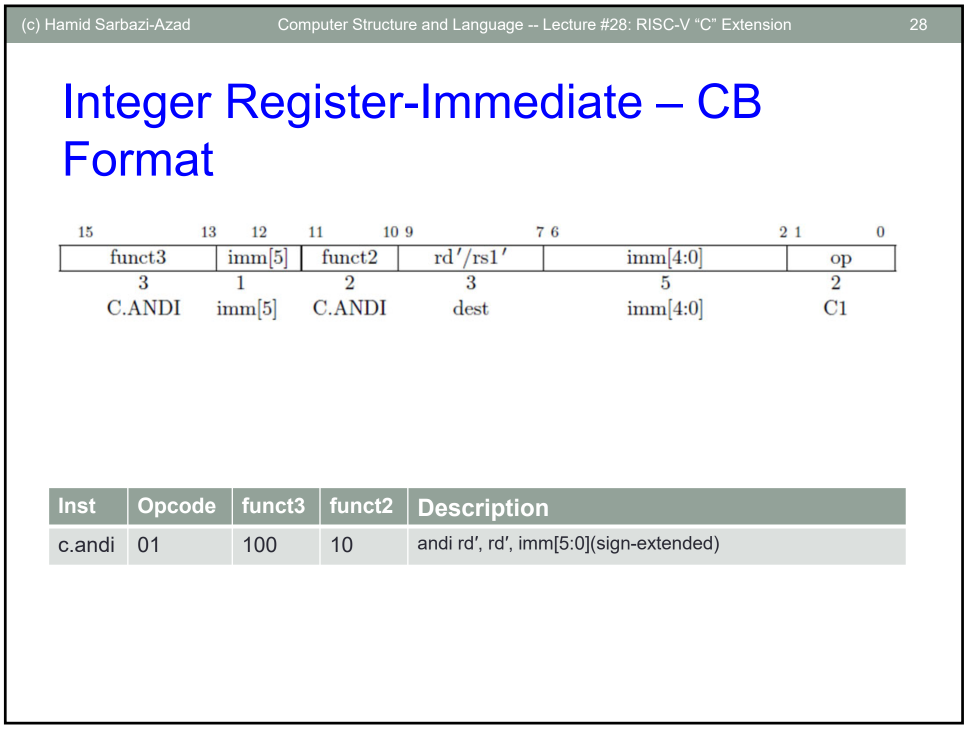


Integer Register-Immediate – CB Format

15 13	12	11 10	9 7	6 2	1 0
funct3	shamt[5]	funct2	rd′/rs1′	shamt[4:0]	op
3	1	2	3	5	2
C.SRLI	shamt[5]	C.SRLI	dest	shamt[4:0]	C1
C.SRAI	shamt[5]	C.SRAI	dest	shamt[4:0]	C1

Inst	Opcode	funct3	funct2	Description
c.srli	01	100	00	srli rd′, rd′, shamt[5:0] (shamt[5] = 0, RVC32 only)
c.srai	01	100	01	srai rd′, rd′, shamt[5:0]

27

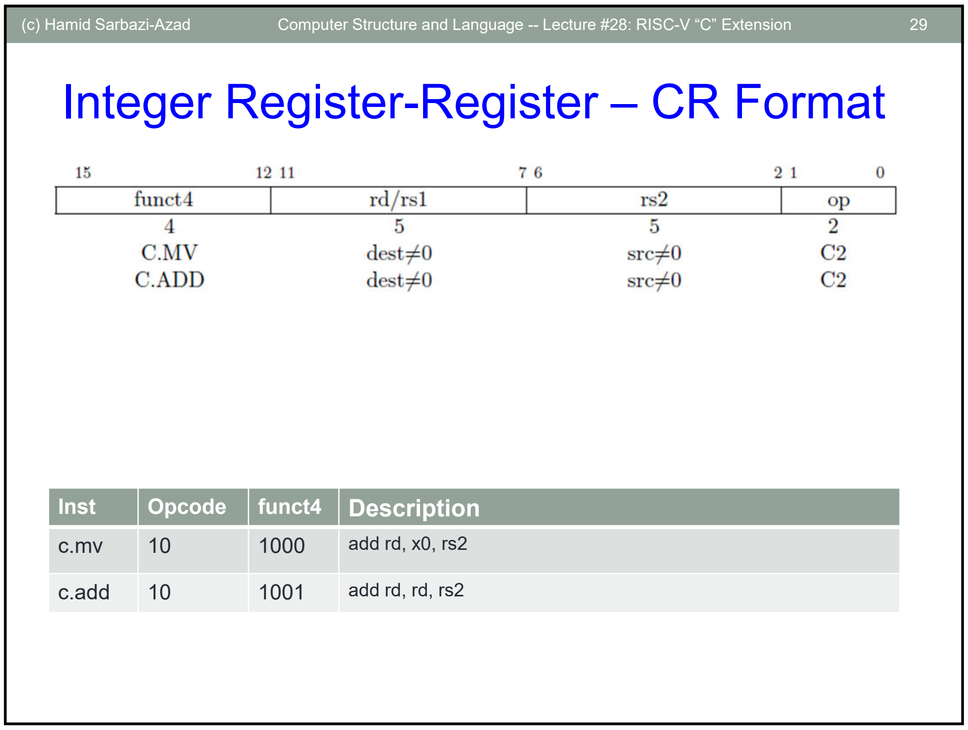


Integer Register-Immediate – CB Format

15 13	12	11 10	9 7	6 2	1 0
funct3	imm[5]	funct2	rd′/rs1′	imm[4:0]	op
3	1	2	3	5	2
C.ANDI	imm[5]	C.ANDI	dest	imm[4:0]	C1

Inst	Opcode	funct3	funct2	Description
c.andi	01	100	10	andi rd′, rd′, imm[5:0](sign-extended)

28

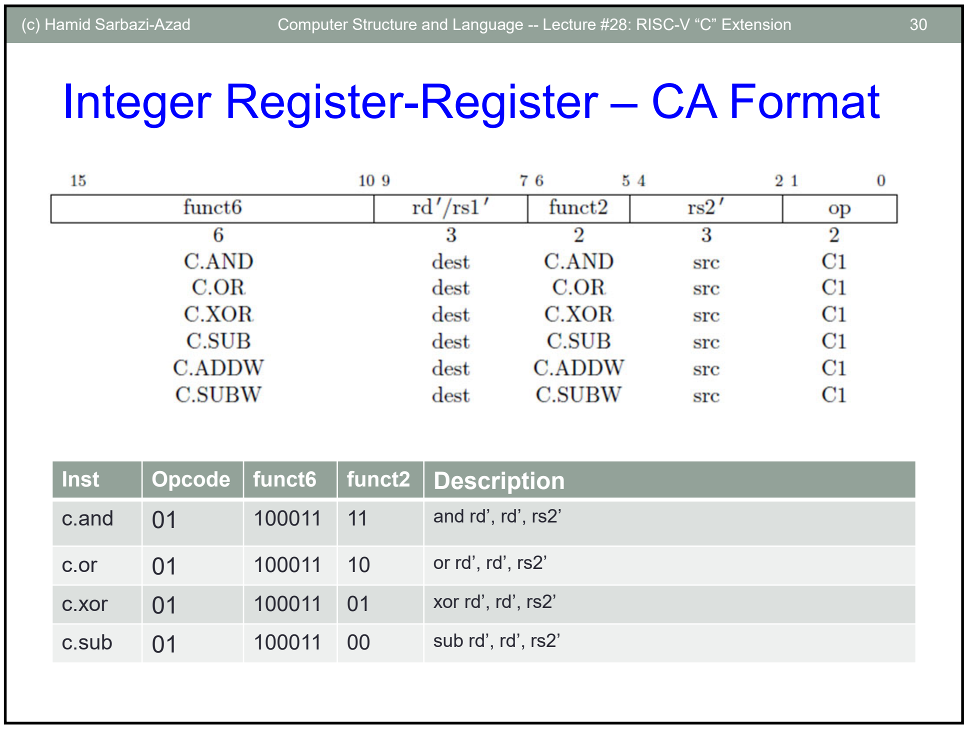


Integer Register-Register – CR Format

15 12	11 7	6 2	1 0
funct4	rd/rs1	rs2	op
4	5	5	2
C.MV	dest$\neq$0	src$\neq$0	C2
C.ADD	dest$\neq$0	src$\neq$0	C2

Inst	Opcode	funct4	Description
c.mv	10	1000	add rd, x0, rs2
c.add	10	1001	add rd, rd, rs2

29

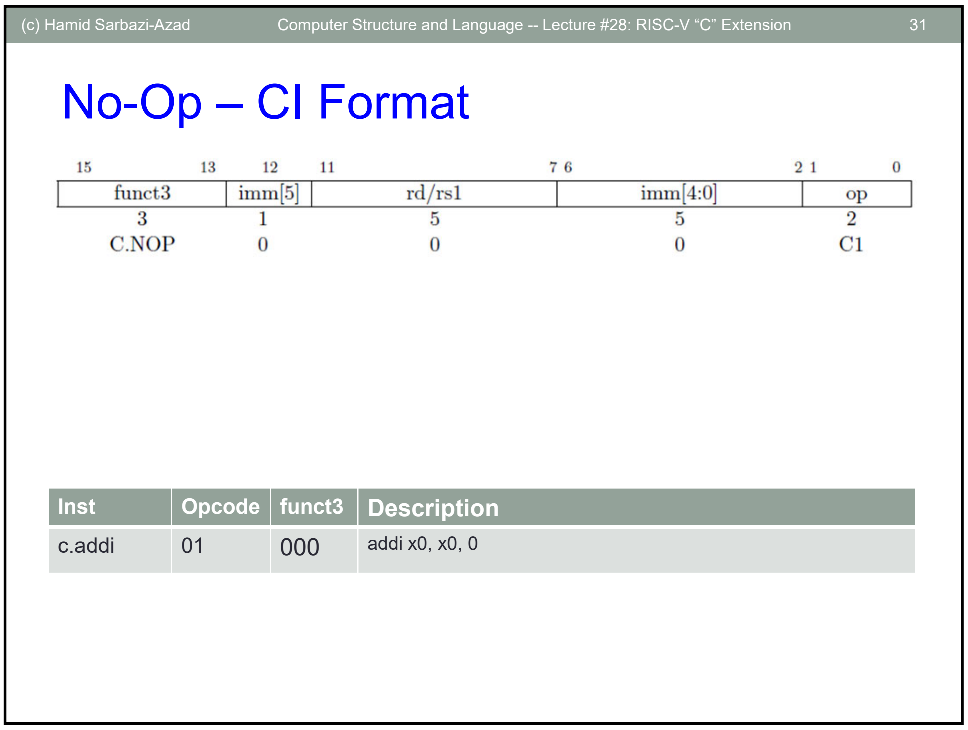


Integer Register-Register – CA Format

15 10	9 7	6 5	4 2	1 0
funct6	rd′/rs1′	funct2	rs2′	op
6	3	2	3	2
C.AND	dest	C.AND	src	C1
C.OR	dest	C.OR	src	C1
C.XOR	dest	C.XOR	src	C1
C.SUB	dest	C.SUB	src	C1
C.ADDW	dest	C.ADDW	src	C1
C.SUBW	dest	C.SUBW	src	C1

Inst	Opcode	funct6	funct2	Description
c.and	01	100011	11	and rd’, rd’, rs2’
c.or	01	100011	10	or rd’, rd’, rs2’
c.xor	01	100011	01	xor rd’, rd’, rs2’
c.sub	01	100011	00	sub rd’, rd’, rs2’

30

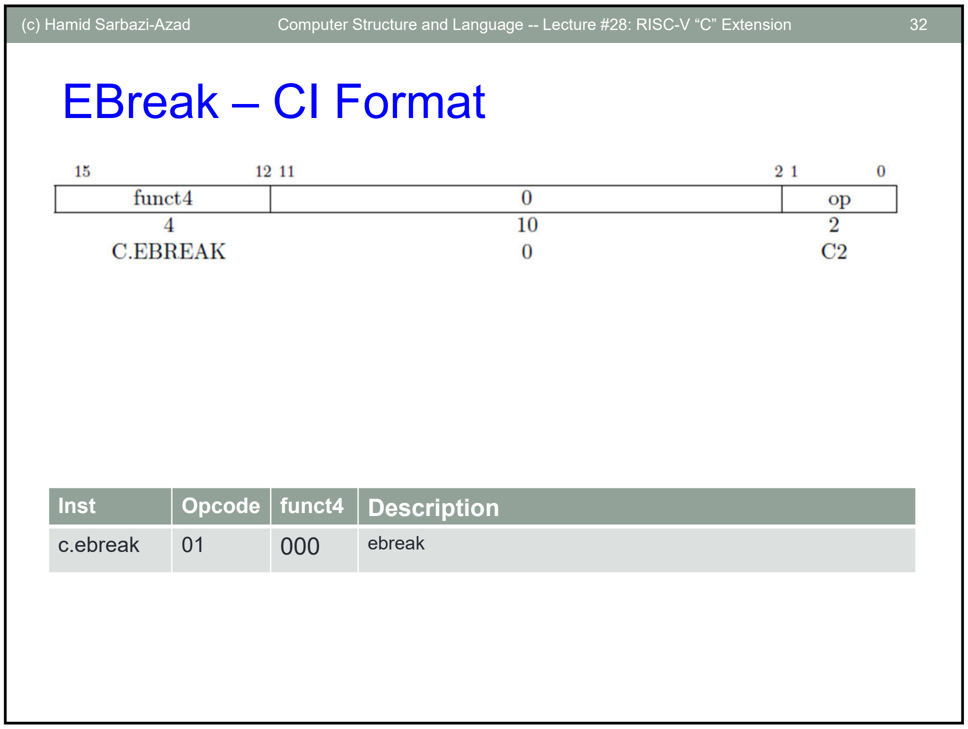


No-Op – CI Format

15 13	12	11 7	6 2	1 0
funct3	imm[5]	rd/rs1	imm[4:0]	op
3	1	5	5	2
C.NOP	0	0	0	C1

Inst	Opcode	funct3	Description
c.addi	01	000	addi x0, x0, 0

31

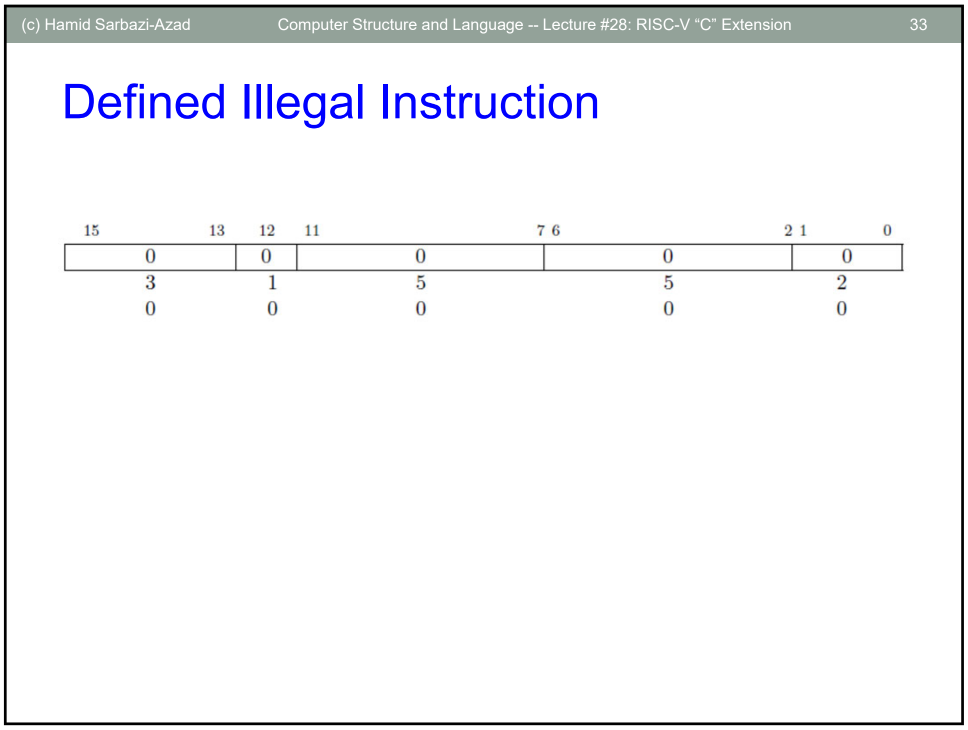


EBreak – CI Format

15 12	11 2	1 0
funct4	0	op
4	10	2
C.EBREAK	0	C2

Inst	Opcode	funct4	Description
c.ebreak	01	000	ebreak

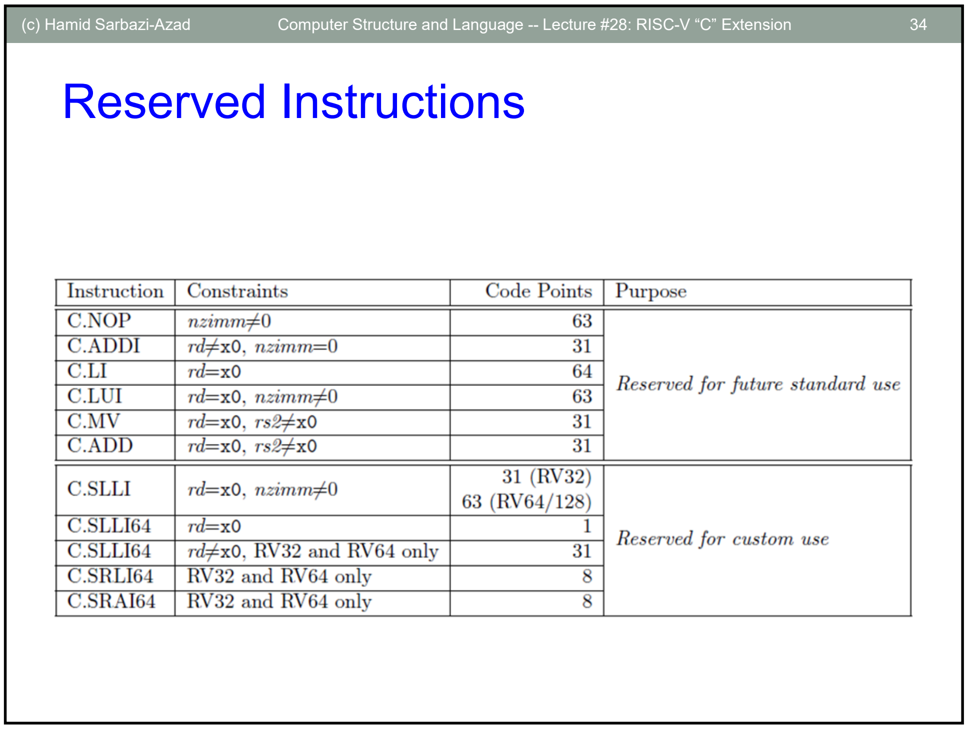
32



Defined Illegal Instruction

15 13	12	11 7	6 2	1 0
0	0	0	0	0
3	1	5	5	2
0	0	0	0	0

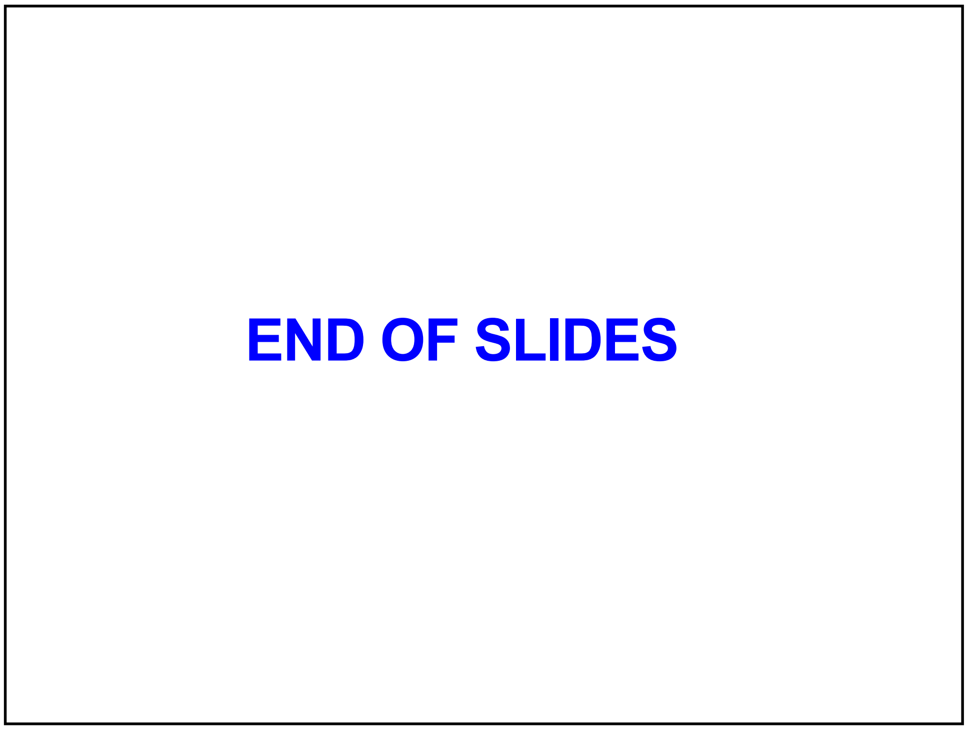
33



Reserved Instructions

Instruction	Constraints	Code Points	Purpose
C.NOP	*nzimm*≠0	63	*Reserved for future standard use*
C.ADDI	*rd*≠x0, *nzimm*=0	31	
C.LI	*rd*=x0	64	
C.LUI	*rd*=x0, *nzimm*≠0	63	
C.MV	*rd*=x0, *rs2*≠x0	31	
C.ADD	*rd*=x0, *rs2*≠x0	31	
C.SLLI	*rd*=x0, *nzimm*≠0	31 (RV32) 63 (RV64/128)	*Reserved for custom use*
C.SLLI64	*rd*=x0	1	
C.SLLI64	*rd*≠x0, RV32 and RV64 only	31	
C.SRLI64	RV32 and RV64 only	8	
C.SRAI64	RV32 and RV64 only	8	

34



END OF SLIDES

35