# *RISCV HW3 (Bonus)*

# RISC-V Compressed ISA Extension

Speaker: Ya-Cheng Ko

Advisor: 吳安宇教授

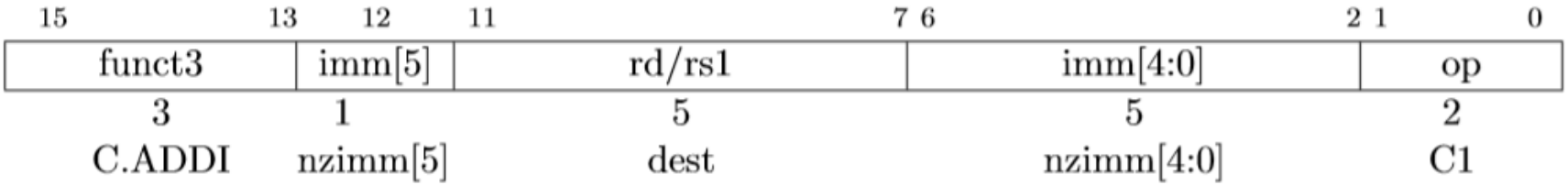
Date:2020/04/16

# Outline

- Introduction
  - What is RVC?
  - Why do we need RVC? (Advantages)
- A Closer Look of C-instructions
  - Register-based load / store instructions
  - Control transfer instructions
  - Integer computational instructions
- Implementation Issues of C extension
  - Instruction decoding
  - PC increment
  - Address alignment issues (NOT in HW3, but in Final Project)

# What is RVC?

- A set of 16-bit instructions
- A standard extension designed under the constraint that each RVC instruction can be expanded into a single 32-bit instruction in either the base ISA or the F and D standard extension
  - One-to-one mapping
  - E.g. $C.ADDI$ expands to $addi\ rd, rd, nzimm[5:0]$




- Goal – efficiently reduce the code size.
  - Principle 1: the most common 32-bit instructions ⟶ 16-bit versions
  - Principle 2: avoid ~~duplicated~~ or ~~meaningless~~ encodings

# Compression Schemes

- How is compression achieved?
  - 1. Small immediate
  - 2. Specify the base register (i.e. x0, x1, or x2)
  - 3. Source and destination are the same register
  - 4. Use x8~x15 only
  - E.g. C.JAL (scheme 1, 2)

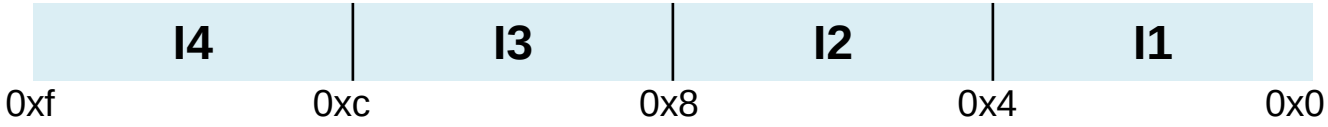
| | Immediate (offset) | | rd | Func / OP code |
|---|---|---|---|---|
| C.JAL | offset[11:1] | 11 bits | x1 0 bits | 5 bits |
| JAL | offset[20:1] | 20 bits | 5 bits | 7 bits |

  - E.g. C.ANDI (scheme 1, 3, 4)

| | immediate | rs1 | rd | Func / OP code |
|---|---|---|---|---|
| C.ANDI | 6 bits | 3 bits | | 7 bits |
| ANDI | 12 bits | 5 bits | 5 bits | 10 bits |

# Advantages – I: Performance

Case 1 4-word block / RV32I



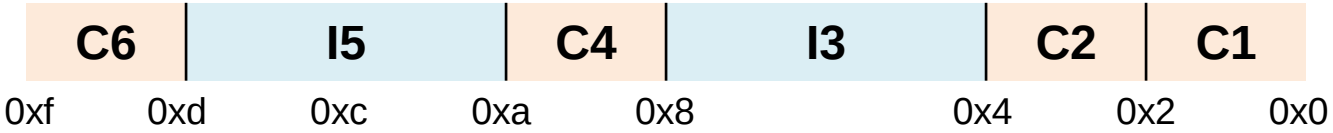

Case 2 6-word block / RV32I

- Reduced miss rate
- Increased miss penalty (larger block)


Case 3 4-word block / RV32IC

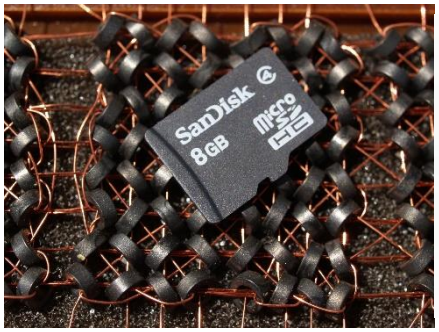
- Lower power (reduced # of access)
- Higher throughput 减少bubbling

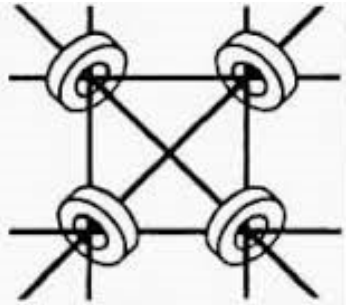
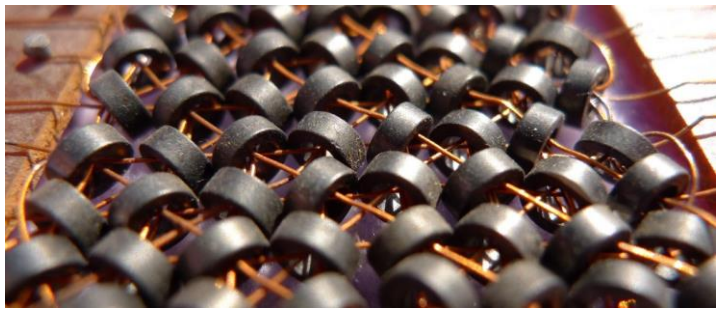
reduces miss rate but doesn't increase miss penalty



# Advantages – II: Lightweight

- Motivation: Few decades ago, the size of memory is limited



- New tech, new demand reduce memory space
  – embedded systems, IOT, and wearable tech


*Compression is an indispensable member that boosts the new-generation trend*

# C Instructions – A Closer Look

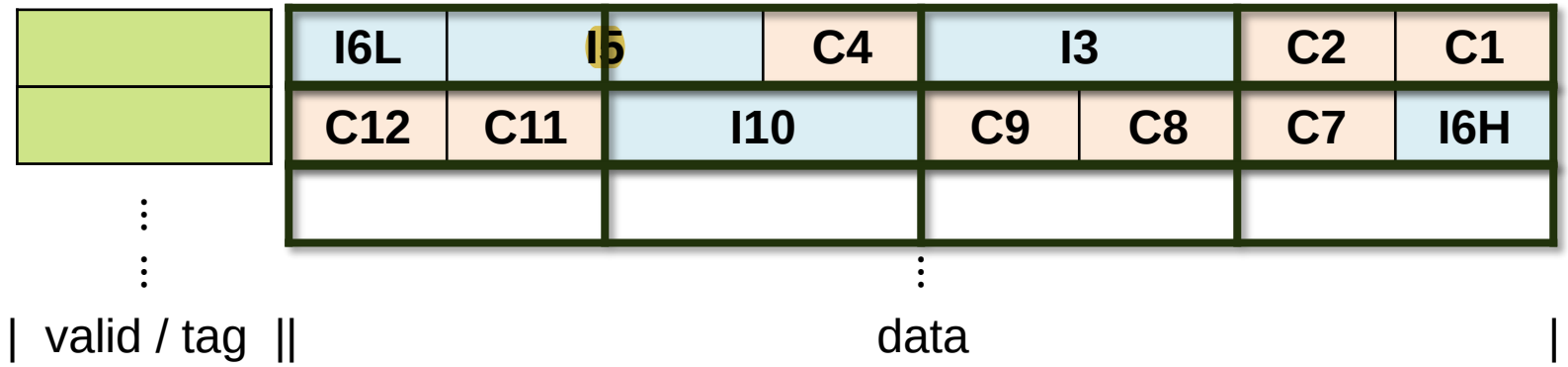
| Register-based Data Transfer instr. |
| --- |
| C.LW |
| C.SW |

| Control Transfer instr. |
| --- |
| C.J |
| C.JAL |
| C.JR |
| C.JALR |
| C.BEQZ |
| C.BNEZ |

| Integer Computational instr. |
| --- |
| C.ADDI |
| C.SLLI |
| C.SRLI |
| C.SRAI |
| C.ANDI |
| C.MV |
| C.ADD |
| C.NOP |

# Overview of Compressed ISA - I

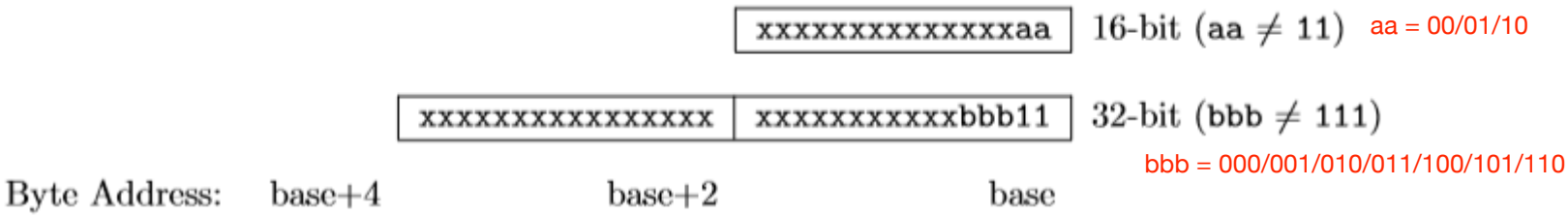
- Instructions in cache
  - To improve code density, all instructions can be aligned on any 16-bit boundary 允許不規則排列
  - A write-through cache may look like:




  - A result is that in machines that supports C extension, target addresses of jump / branch instructions are relaxed to 16-bit boundary.

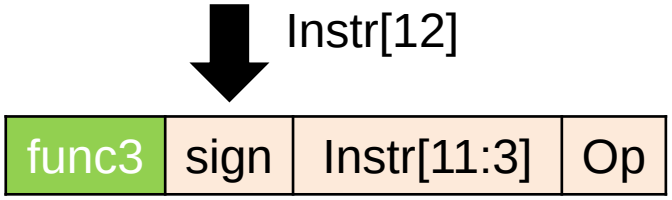
# Overview of Compressed ISA - II

- How to recognize a compressed instruction?
  - RISC-V encodes the instruction length with the least significant bits

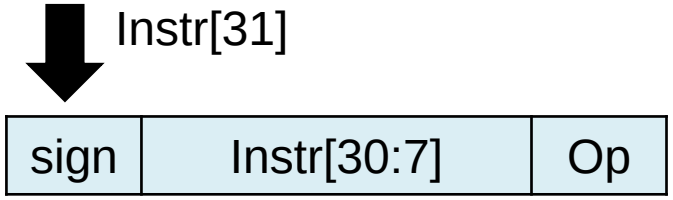



  - Known as soon as the first half-word of an instruction is fetched!

- Fixed position for sign-extending bit



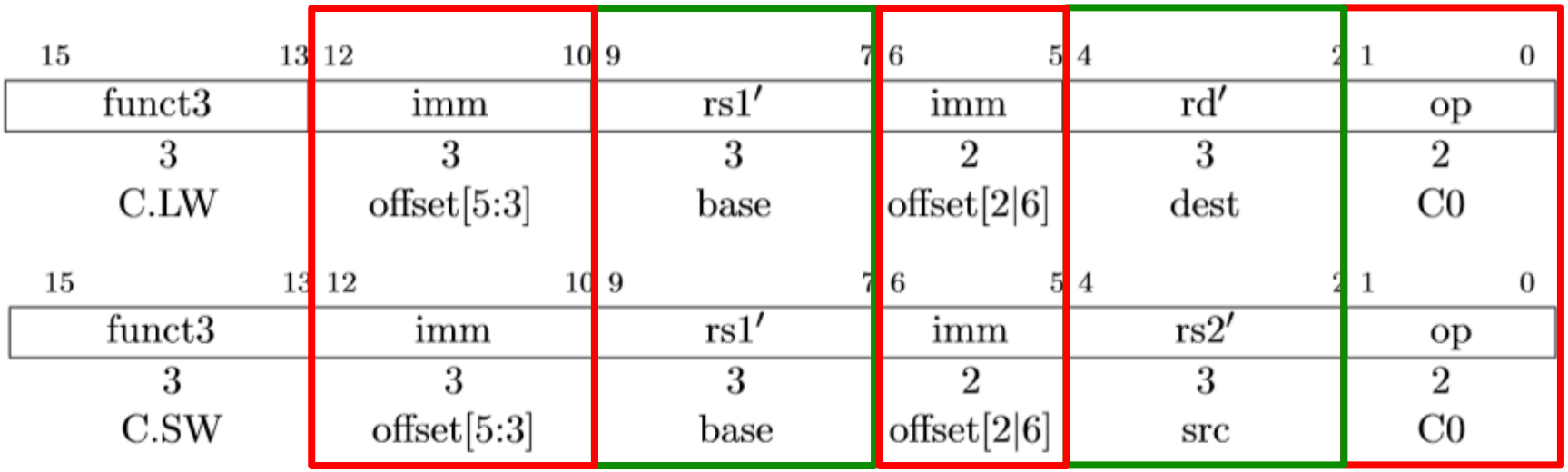

Recall RV32I:



# Register-based Load / Store – I

- In RVC, there are two register-based compressed data transfer instructions for RV32I: C.LW and C.SW

C0: 00
C1: 01
C2: 10




**offset [6:2]** word offset 而非 byte offset

**x8 ~ x15** 有prime

**2'b00**

- Observations
  1. offset[6] is not the ~~first bit~~ in the immediate field (not like in RV32I)
  2. The least significant bit specified is offset[2] rather than ~~offset[0]~~

# Register-based Load / Store – II

- The notable differences of C.LW / C.SW from LW / SW lie in the way their immediates (byte offsets) are obtained.
  - 1. Zero-extend the immediate

| | Immediate | Reason |
|---|---|---|
| C.LW / C.SW | Zero-extended | To increase the reach 正offset比負offset常見 |
| LW / SW | Sign-extended | Offset can be either positive or negative |

  - Positive and negative offset examples:

[Array elements access]

```
int data[1024];
for (int i = 0; i < 1024; ++i)
    data[i] = i;
```

[Reverse iteration through an array]

```
int* p = data + 1023;
while (*p > *(p-1)) --p;
```

negative offset

**Extremely common program behavior!**

# Register-based Load / Store – III

- The notable differences of C.LW / C.SW from LW / SW lie in the way their immediates (byte offsets) are obtained.
  - 2. Pad the immediate with 2'b00 at LSB position

| | Immediate | Reason |
|---|---|---|
| C.LW / C.SW | Scaled by a factor of four | Word access only |
| LW / SW | Translated faithfully | Base ISA support LB / SB |

- Byte access granularity

| | 1 bit | 9 bits | 1 bit | 1 bit |
|---|---|---|---|---|
| LW | offset[13] | offset [10:2] | offset[12] | offset[11] |
| LH | offset[12] | | offset[1] | offset[11] |
| LB | offset[11] | | offset[1] | offset[0] |

If … else if … else

vs.

less hardware complexity

| | 12 bits |
|---|---|
| LW | offset[11:0] |
| LH | offset[11:0] |
| LB | offset[11:0] |

{ 20{imm[11]}, imm[11:0] }

# Register-based Load / Store – IV

- Summary –

  How C.LW / C.SW are different from LW / SW in terms of immediates?

  - 1. Zero-extend the immediate

| | Immediate | Reason |
|---|---|---|
| C.LW / C.SW | Zero-extended | To increase the reach |
| LW / SW | Sign-extended | Offset can be either positive or negative |

  - 2. Pad the immediate with 2'b00 at LSB position

| | Immediate | Reason |
|---|---|---|
| C.LW / C.SW | Scaled by a factor of four | Word access only |
| LW / SW | Translated faithfully | Base ISA support byte access |

# Control Transfer Instructions – I

- Unconditional jump / conditional branch instructions
- C.J, C.JAL, C.JR, C.JALR, C.BEQZ, C.BNEZ
- The offsets of all six instructions are in multiple of 2 bytes (16 bits)
  - offset [ ∗ :1]

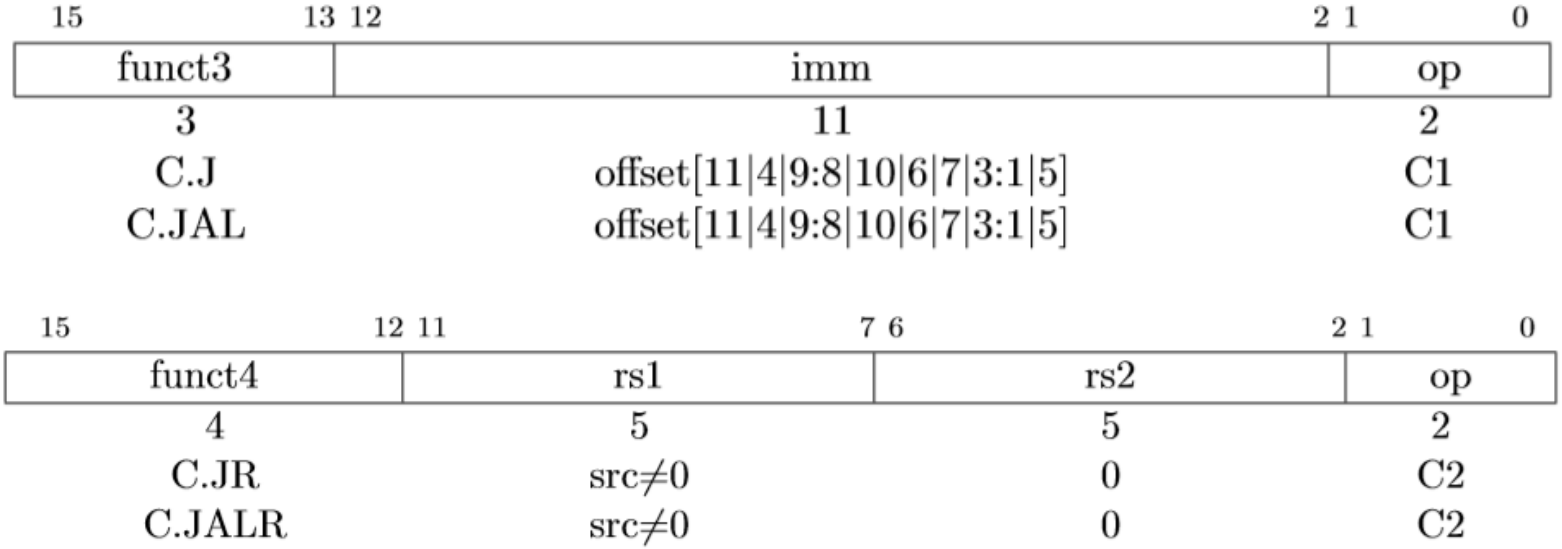
- Branch instructions

| 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 |

| RVC instr. | Equivalent |
|---|---|
| C.BEQZ | beq rs1′, x0, offset[8:1] |
| C.BNEZ | bne rs1′, x0, offset[8:1] |

# Control Transfer Instructions – II

- Direct / Indirect jump instructions



| 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 | |

| 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 | |

| RVC instr. | Equivalent | RVC instr. | Equivalent |
|---|---|---|---|
| C.J | jal x0, offset[11:1] | C.JR | jalr x0, rs1, 0 |
| C.JAL | jal **x1**, offset[11:1] | C.JALR | jalr **x1**, rs1, 0 |

# Control Transfer Instructions – III

- Q: Why do we need four types of J-type compressed instruction but only using JAL and JALR in RV32I?
  - J and JR are frequently used in computer programs and commonly defined in other ISAs. In RV32I, they can be replaced with JAL and JALR perfectly. However, according to the compression Scheme 2 (p.4), C.JAL and C.JALR have the fixed link register x1. Hence, we need the additional C.J and C.JR for our purpose.

| Pseudo Instruction | RV32I | RVC |
|---|---|---|
| j offset | jal x0, offset | C.J offset |
| jal offset | jal x1, offset | C.JAL offset |
| jr rs | jalr x0, rs, 0 | C.JR rs |
| jalr rs | jalr x1, rs, 0 | C.JALR rs |

# Control Transfer Instructions – IV

- Q: Why C.JR / C.JALR has a 5'b00000 in its field?
  - C.JR and C.JALR do not need a ~~second source register~~.
    The non-zero encoding values for rs2 are reserved for C.MV and C.ADD.

- Q: Why the source register of C.JR / C.JALR can't be x0?
  - RES / C.EBREAK

| 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 |

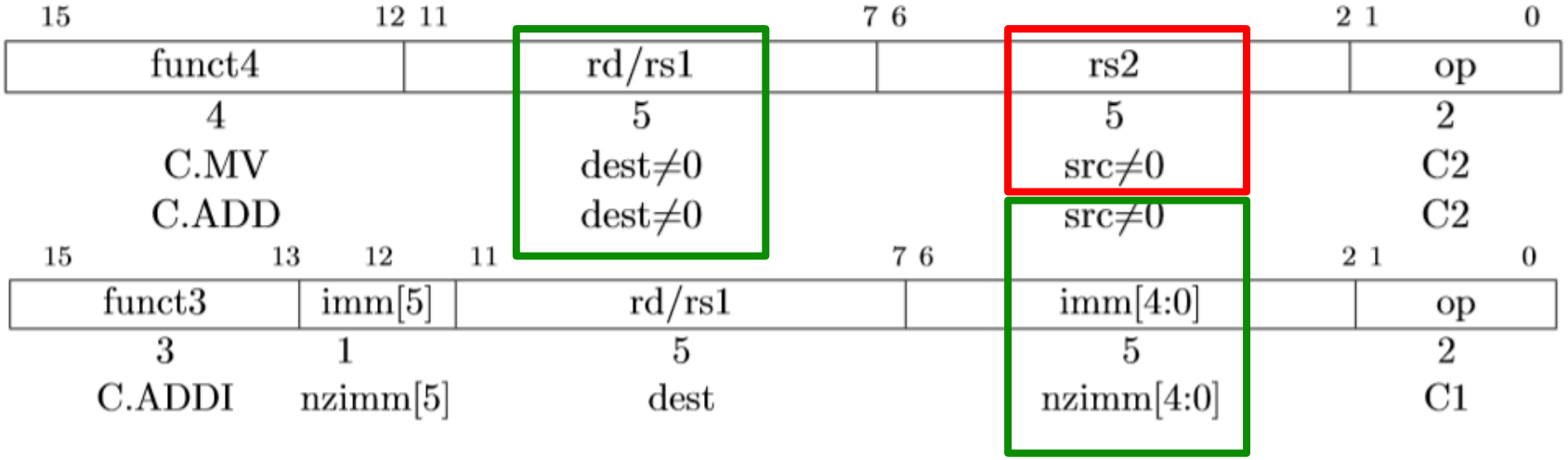
| | | | | | |
|---|---|---|---|---|---|
| 100 | 0 | rs1≠0 | 0 | 10 | C.JR *(RES, rs1=0)* |
| 100 | 0 | rd≠0 | rs2≠0 | 10 | C.MV *(HINT, rd=0)* |
| 100 | 1 | 0 | 0 | 10 | C.EBREAK |
| 100 | 1 | rs1≠0 | 0 | 10 | C.JALR |
| 100 | 1 | rs1/rd≠0 | rs2≠0 | 10 | C.ADD *(HINT, rd=0)* |

# Integer Computational Instructions – I

- C.ADDI, C.MV, C.ADD

meaningless to write x0

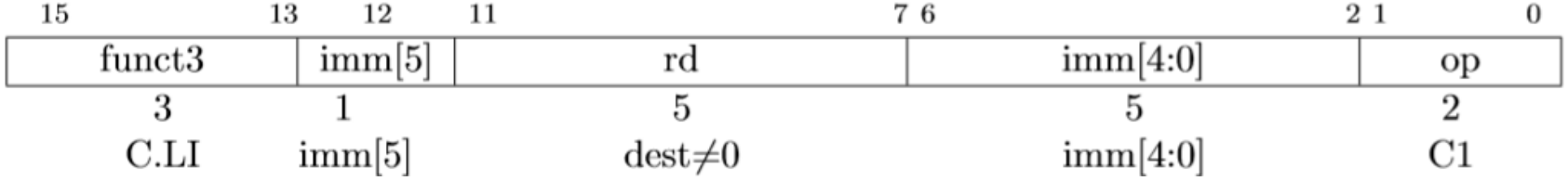
How to clear registers?




adding zero here is also meaningless (rd = rs1)

= C.MV

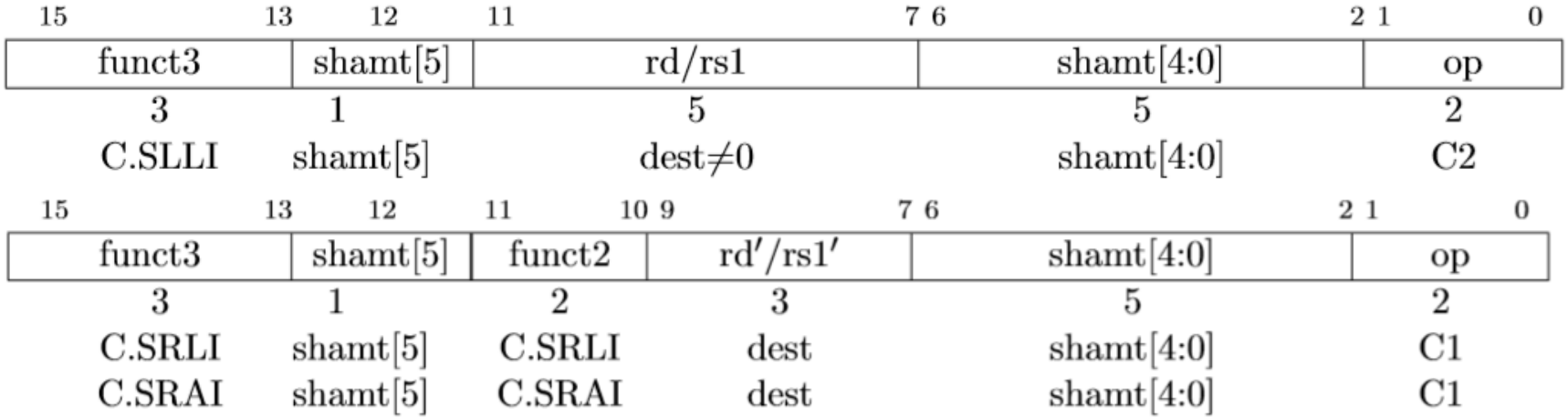
- C.LI (load immediate) is the formal instruction to clear registers.




- C.ANDI / C.SUB / C.XOR can zero the register value (x8~x15 only).

# Integer Computational Instructions – II

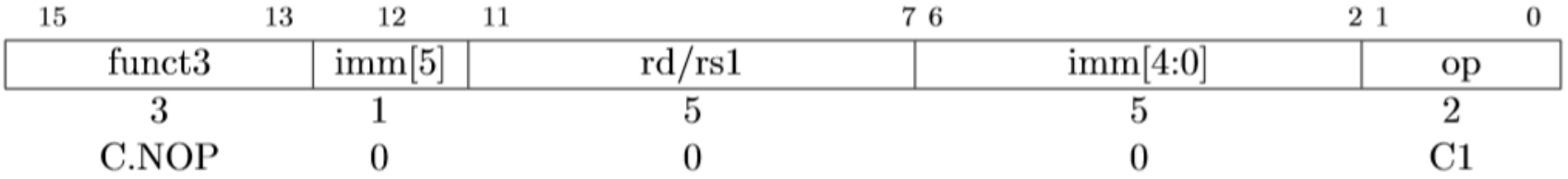
- C.SLLI, C.SRLI, C.SRAI



| 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 |

| 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 |

- C.NOP
  - 16'h0 is not a valid compressed instruction



| 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 |

# Implementation Issues of C Extension based on RV32I core

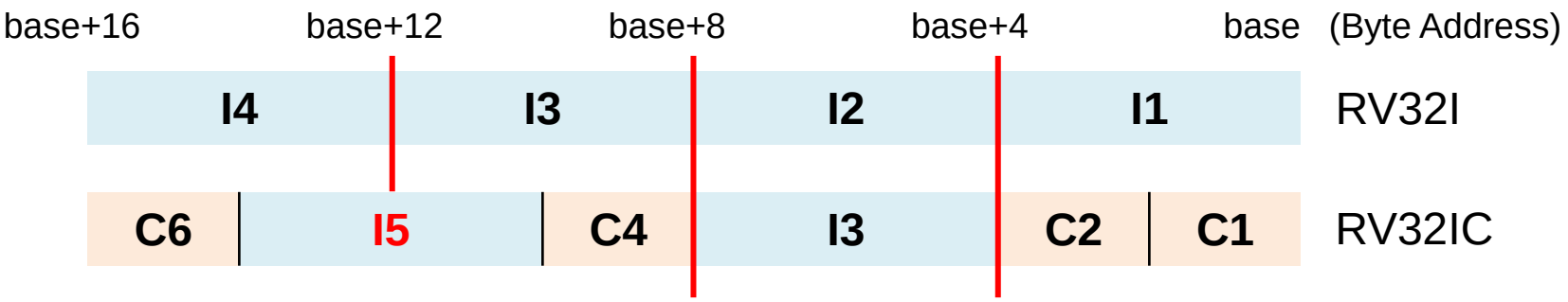
- Instruction decoding
- PC increment
- Address alignment issues
  (Not in HW3, but in Final Project)

# To move on to RV32IC ...

- A compressed instruction differs from RV32I in two ways:
  - Encoding fields
  - Instruction length (16 bits vs. 32 bits or more)

- Modifications (TODOs)

1. Add a dedicated decoder for compressed instruction [Encoding fields]
2. PC (+4 or +2) [Instruction length]
3. ***Address alignment considerations (Not in HW3)*** [Instruction length]



# Extract Information in C instructions (TODO1)

- Approach1: Decoder (ID)
  - Decodes 16-bit instructions into control signals, rs, rd, immediate, etc.
  - Less maintainable and error-prone

- Approach2: Decompressor (IF)
  - This is possible owing to the one-to-one mapping policy
  - Expands 16-bit instructions into their 32-bit counterparts
  - Don't need to ~~decode~~

[Decompression example]

16'b1111_0111_1110_0101 ⟹ C.BNEZ x15 0xF4

32'b1111_1110_0000_0111_1001_0100_1110_0011
(bne x15, x0, 0xFF4)

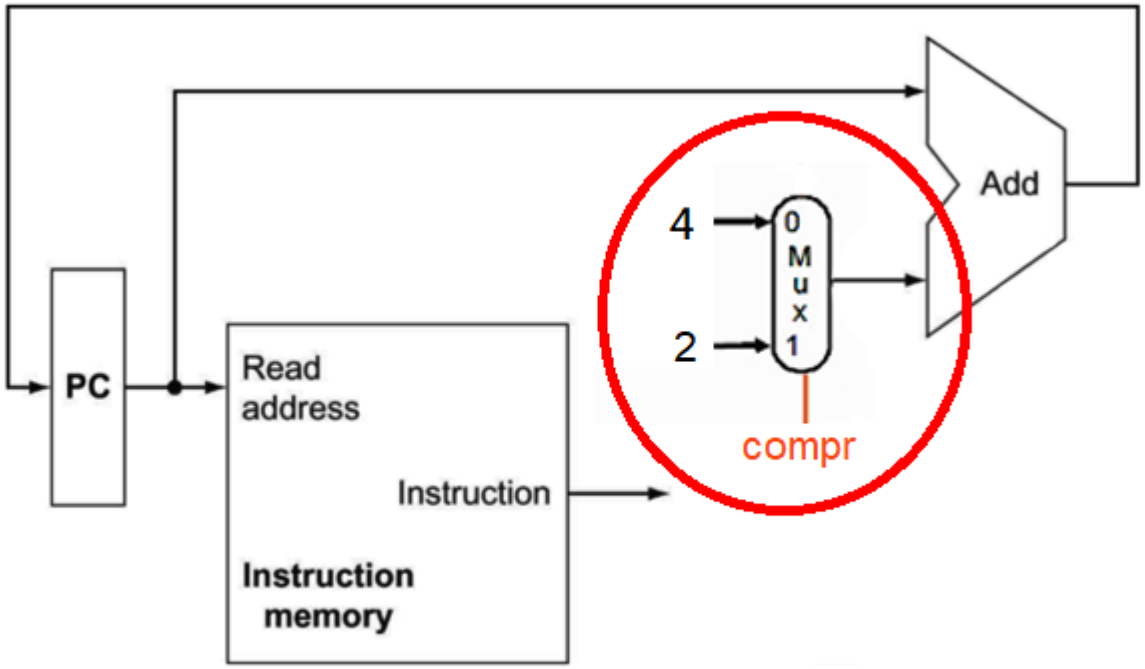
# Extract Information in C instructions (TODO1)

- Approach1: Decoder (ID)
  - Decodes 16-bit instructions into control signals, rs, rd, immediate, etc.
  - Less maintainable and error-prone
- Approach2: Decompressor (IF)
  - This is possible owing to the one-to-one mapping policy
  - Expands 16-bit instructions into their 32-bit counterparts
  - Don’t need to decode
  - Abstraction

*"Hardware design can simply expand RVC instructions during decode, simplifying verification and minimizing modification to existing microarchitectures."*

# PC Increment Selection (TODO2)

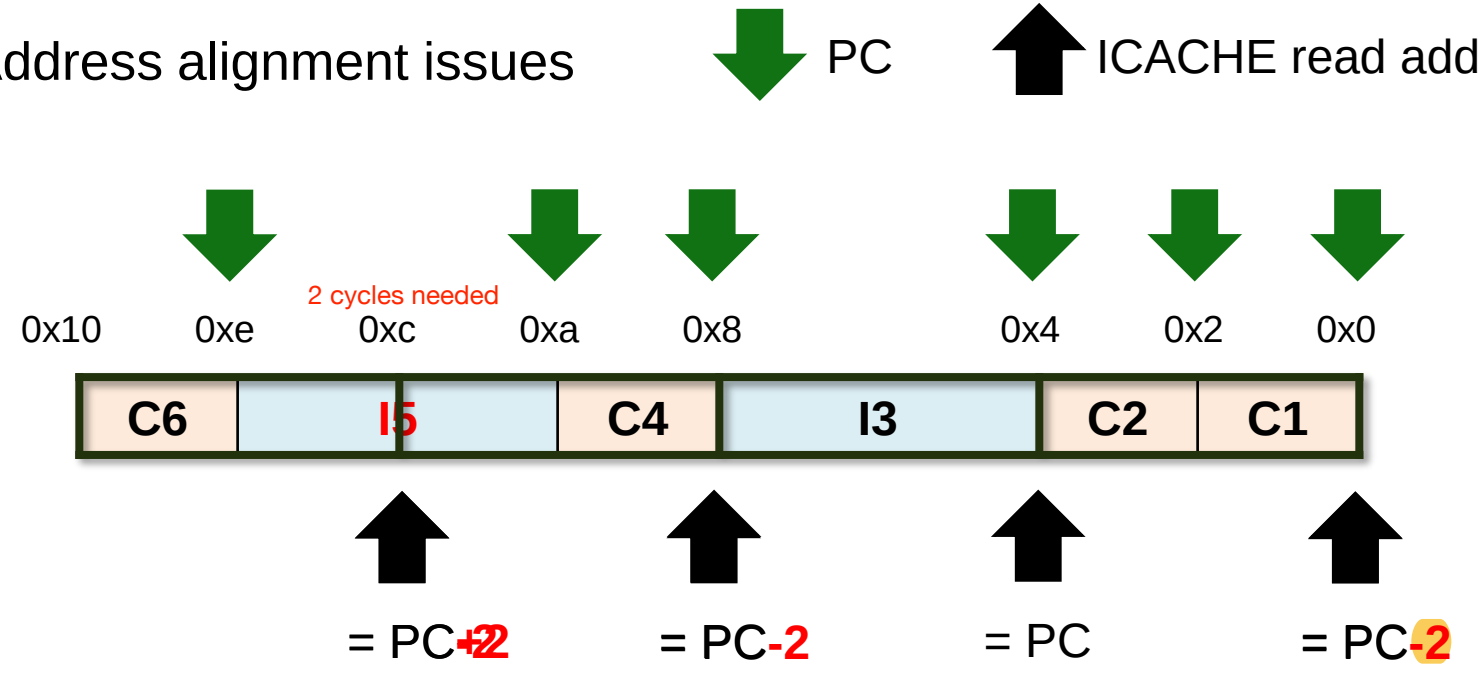
- The next PC value would be PC+2 rather than PC+4 if a compressed instruction is currently fetched.



# Address Alignment (TODO3) NOT in HW3

- Typically, the memory access is at the granularity of the word size (four bytes in our context) to simplify the hardware design.
  - This means the read address should be aligned to a four-byte boundary.

- Address alignment issues



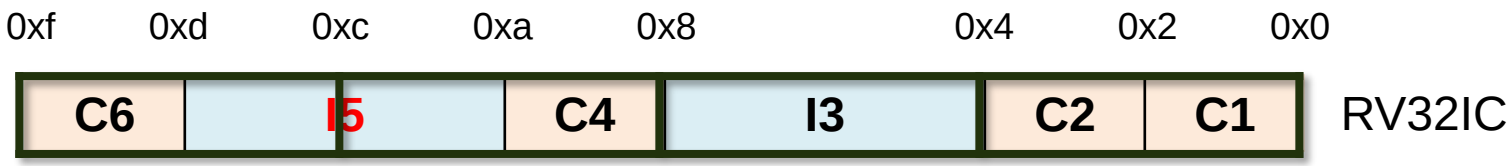
# Address Alignment (TODO3) NOT in HW3

- Typically, the memory access is at the granularity of the word size (four bytes in our context) to simplify the hardware design.
  - This means the read address should be aligned to a four-byte boundary.

- How to fetch C2 (address = 0x2, not 4-byte aligned)?
  - Specify {PC[31:2], 2'b00} as the read address and take the higher 16 bits.
  - PC ≠ read address in this way

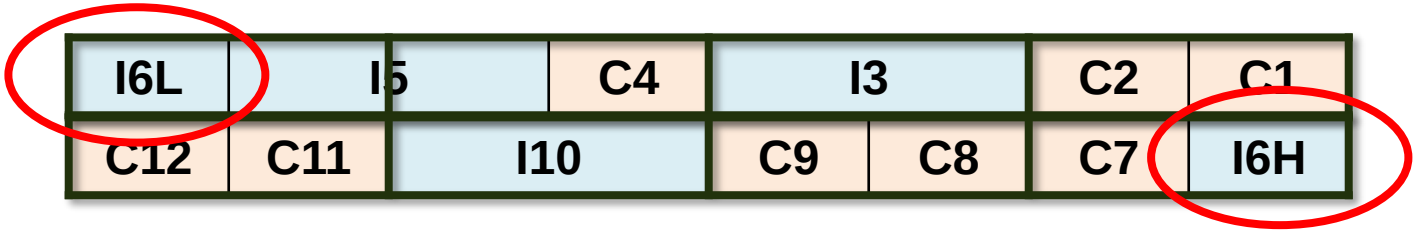
- How about fetching I5 (across byte address bo
  - Need two cycles => bubble
  - Can we do better? (try to optimize by yourself !)

*Buffer?*



# Other Issues NOT in HW3

- Cache miss: an instruction can cross block boundary!
  - What if I5 is a branch back to C1 and then a wrong prediction occurs?
  - Suppress busy cache (clear the memory read command)




- Typically, an improvement for one-cycle instruction fetch results in some other issues. Handle them by yourself!

# The RISC-V Instruction Set Manual

## – A Brief Guide –

## ❖ Detail specification of each instruction

| 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 |

These instructions use the CL format.

C.LW 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 base address in register *rs1′*. It expands to `lw rd′, offset[6:2](rs1′)`.

| 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 |

C.SLLI is a CI-format instruction that performs a logical left shift of the value in register *rd* then writes the result to *rd*. The shift amount is encoded in the *shamt* field, where *shamt[5]* must be zero for RV32C. For RV32C and RV64C, the shift amount must be non-zero. For RV128C, a shift amount of zero is used to encode a shift of 64. C.SLLI expands into `slli rd, rd, shamt[5:0]`, except for RV128C with `shamt=0`, which expands to `slli rd, rd, 64`.

## 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 |

A 16-bit instruction with all bits zero is permanently reserved as an illegal instruction.

---

*We reserve all-zero instructions to be illegal instructions to help trap attempts to execute zero-ed or non-existent portions of the memory space. The all-zero value should not be redefined in any non-standard extension. Similarly, we reserve instructions with all bits set to 1 (corresponding to very long instructions in the RISC-V variable-length encoding scheme) as illegal to capture another common value seen in non-existent memory regions.*

Essence!

## ❖ RVC Opcode map

| inst[15:13] / inst[1:0] | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 | |
|---|---|---|---|---|---|---|---|---|---|
| 00 | ADDI4SPN | FLD<br>FLD<br>LQ | LW | FLW<br>LD<br>LD | *Reserved* | FSD<br>FSD<br>SQ | SW | FSW<br>SD<br>SD | RV32<br>RV64<br>RV128 |
| 01 | ADDI | JAL<br>ADDIW<br>ADDIW | LI | LUI/ADDI16SP | MISC-ALU | J | BEQZ | BNEZ | RV32<br>RV64<br>RV128 |
| 10 | SLLI | FLDSP<br>FLDSP<br>LQ | LWSP | FLWSP<br>LDSP<br>LDSP | J[AL]R/MV/ADD | FSDSP<br>FSDSP<br>SQ | SWSP | FSWSP<br>SDSP<br>SDSP | RV32<br>RV64<br>RV128 |
| 11 | >16b | | | | | | | | |

- Lists of instructions with specification for bit fields

| 15 14 13 | 12 | 11 10 9 8 7 | 6 5 4 3 2 | 1 0 | |
|---|---|---|---|---|---|
| 000 | nzuimm[5] | rs1/rd≠0 | nzuimm[4:0] | 10 | C.SLLI *(HINT, rd=0; RV32 NSE, nzuimm[5]=1)* |
| 000 | 0 | rs1/rd≠0 | 0 | 10 | C.SLLI64 *(RV128; RV32/64 HINT; HINT, rd=0)* |
| 001 | uimm[5] | rd | uimm[4:3\|8:6] | 10 | C.FLDSP *(RV32/64)* |
| 001 | uimm[5] | rd≠0 | uimm[4\|9:6] | 10 | C.LQSP *(RV128; RES, rd=0)* |
| 010 | uimm[5] | rd≠0 | uimm[4:2\|7:6] | 10 | C.LWSP *(RES, rd=0)* |
| 011 | uimm[5] | rd | uimm[4:2\|7:6] | 10 | C.FLWSP *(RV32)* |
| 011 | uimm[5] | rd≠0 | uimm[4:3\|8:6] | 10 | C.LDSP *(RV64/128; RES, rd=0)* |
| 100 | 0 | rs1≠0 | 0 | 10 | C.JR *(RES, rs1=0)* |
| 100 | 0 | rd≠0 | rs2≠0 | 10 | C.MV *(HINT, rd=0)* |
| 100 | 1 | 0 | 0 | 10 | C.EBREAK |
| 100 | 1 | rs1≠0 | 0 | 10 | C.JALR |
| 100 | 1 | rs1/rd≠0 | rs2≠0 | 10 | C.ADD *(HINT, rd=0)* |
| 101 | uimm[5:3\|8:6] | | rs2 | 10 | C.FSDSP *(RV32/64)* |
| 101 | uimm[5:4\|9:6] | | rs2 | 10 | C.SQSP *(RV128)* |
| 110 | uimm[5:2\|7:6] | | rs2 | 10 | C.SWSP |
| 111 | uimm[5:2\|7:6] | | rs2 | 10 | C.FSWSP *(RV32)* |
| 111 | uimm[5:3\|8:6] | | rs2 | 10 | C.SDSP *(RV64/128)* |

Table 12.6: Instruction listing for RVC, Quadrant 2.

# Have a nice trip!