



COMPUTER ORGANIZATION AND DESIGN

The Hardware/Software Interface



Computer Architecture

Lectures 4-5

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
 - But with many aspects in common
- Early computers had very simple instruction sets
 - Simplified implementation
- Many modern computers also have simple instruction sets

The RISC-V Instruction Set

- Used as the example throughout the book
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation
(riscv.org)
- Typical of many modern ISAs
 - See RISC-V Reference Data tear-out card
- Similar ISAs have a large share of embedded core market
 - Applications in consumer electronics, network/storage equipment, cameras, printers, ...

RISC-V Assembly language summary (1)

| 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 doubleword | ld x5, 40(x6) | x5 = Memory[x6 + 40] | Doubleword from memory to register |
| | Store doubleword | sd x5, 40(x6) | Memory[x6 + 40] = x5 | Doubleword from register to memory |
| | 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 | lrd 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 |

RISC-V Assembly language summary (2)

| | | | | |
|----------------------|--------------------------------------|-------------------------------|-------------------------------------|--|
| Shift | Shift left logical | <code>sll x5, x6, x7</code> | $x5 = x6 \ll x7$ | Shift left by register |
| | Shift right logical | <code>srl x5, x6, x7</code> | $x5 = x6 \gg x7$ | Shift right by register |
| | Shift right arithmetic | <code>sra x5, x6, x7</code> | $x5 = x6 \gg x7$ | Arithmetic shift right by register |
| | Shift left logical immediate | <code>slli x5, x6, 3</code> | $x5 = x6 \ll 3$ | Shift left by immediate |
| | Shift right logical immediate | <code>srli x5, x6, 3</code> | $x5 = x6 \gg 3$ | Shift right by immediate |
| | Shift right arithmetic immediate | <code>srai x5, x6, 3</code> | $x5 = x6 \gg 3$ | Arithmetic shift right by immediate |
| Conditional branch | Branch if equal | <code>beq x5, x6, 100</code> | if ($x5 == x6$) go to PC+100 | PC-relative branch if registers equal |
| | Branch if not equal | <code>bne x5, x6, 100</code> | if ($x5 != x6$) go to PC+100 | PC-relative branch if registers not equal |
| | Branch if less than | <code>blt x5, x6, 100</code> | if ($x5 < x6$) go to PC+100 | PC-relative branch if registers less |
| | Branch if greater or equal | <code>bge x5, x6, 100</code> | if ($x5 \geq x6$) go to PC+100 | PC-relative branch if registers greater or equal |
| | Branch if less, unsigned | <code>bltu x5, x6, 100</code> | if ($x5 < x6$) go to PC+100 | PC-relative branch if registers less, unsigned |
| | Branch if greater or equal, unsigned | <code>bgeu x5, x6, 100</code> | if ($x5 \geq x6$) go to PC+100 | PC-relative branch if registers greater or equal, unsigned |
| Unconditional branch | Jump and link | <code>jal x1, 100</code> | $x1 = \text{PC}+4$; go to PC+100 | PC-relative procedure call |
| | Jump and link register | <code>jalr x1, 100(x5)</code> | $x1 = \text{PC}+4$; go to $x5+100$ | Procedure return; indirect call |



Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination

```
add a, b, c // a gets b + c
```
- All arithmetic operations have this form
- *Design Principle 1: Simplicity favours regularity*
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

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

- Compiled RISC-V code:

```
add t0, g, h    // temp t0 = g + h
add t1, i, j    // temp t1 = i + j
add f, t0, t1   // f = t0 - t1
```

Register Operands

- Arithmetic instructions use register operands
- RISC-V has a 32×64 -bit register file
 - Use for frequently accessed data
 - 64-bit data is called a “doubleword”
 - 32 x 64-bit general purpose registers x0 to x30
 - 32-bit data is called a “word”
- *Design Principle 2: Smaller is faster*
 - c.f. main memory: millions of locations

RISC-V Registers

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Register Operand Example

- C code:

$f = (g + h) - (i + j);$

- f, ..., j in x19, x20, ..., x23

- Compiled RISC-V code:

add x5, x20, x21

add x6, x22, x23

sub x19, x5, x6

Memory Operands

- Main memory used for composite data
 - Arrays, structures, dynamic data
- To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory
- Memory is byte addressed
 - Each address identifies an 8-bit byte
- RISC-V is Little Endian
 - Least-significant byte at least address of a word
 - *c.f.* Big Endian: most-significant byte at least address
- RISC-V does not require words to be aligned in memory
 - Unlike some other ISAs

Memory Operand Example

- C code:

$A[12] = h + A[8];$

- h in $x21$, base address of A in $x22$

- Compiled RISC-V code:

- Index 8 requires offset of 64
 - 8 bytes per doubleword

ld x9, 64(x22)

add x9, x21, x9

sd x9, 96(x22)

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

```
addi x22, x22, 4
```

- Make the common case fast
 - Small constants are common
 - Immediate operand avoids a load instruction

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to $+2^n - 1$
- Example
 - 0000 0000 ... 0000 1011₂
= 0 + ... + 1×2³ + 0×2² + 1×2¹ + 1×2⁰
= 0 + ... + 8 + 0 + 2 + 1 = 11₁₀
- Using 64 bits: 0 to +18,446,774,073,709,551,615

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
 - $1111\ 1111\ \dots\ 1111\ 1100_2$
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$
- Using 64 bits: $-9,223,372,036,854,775,808$
to $9,223,372,036,854,775,807$

2s-Complement Signed Integers

- Bit 63 is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111

Signed Negation

- Complement and add 1
 - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111\dots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
 - $+2 = 0000\ 0000\dots0010_{\text{two}}$
 - $-2 = 1111\ 1111\dots1101_{\text{two}} + 1$
 $= 1111\ 1111\dots1110_{\text{two}}$

Sign Extension

- Representing a number using more bits
 - Preserve the numeric value
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110
- In RISC-V instruction set
 - 1b: sign-extend loaded byte
 - 1bu: zero-extend loaded byte

Representing Instructions

- Instructions are encoded in binary
 - Called machine code
- RISC-V instructions
 - Encoded as 32-bit instruction words
 - Small number of formats encoding operation code (opcode), register numbers, ...
 - Regularity!

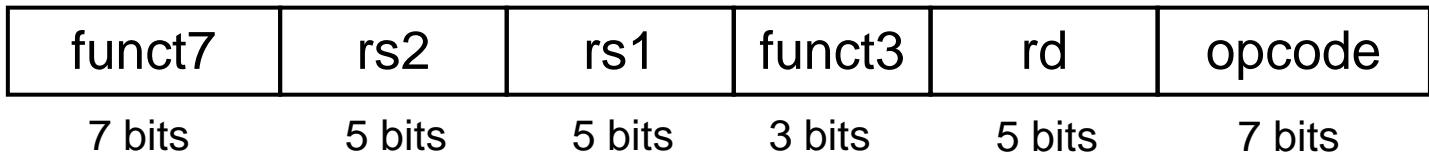
Hexadecimal

- Base 16
 - Compact representation of bit strings
 - 4 bits per hex digit

| | | | | | | | |
|---|------|---|------|---|------|---|------|
| 0 | 0000 | 4 | 0100 | 8 | 1000 | c | 1100 |
| 1 | 0001 | 5 | 0101 | 9 | 1001 | d | 1101 |
| 2 | 0010 | 6 | 0110 | a | 1010 | e | 1110 |
| 3 | 0011 | 7 | 0111 | b | 1011 | f | 1111 |

- Example: eca8 6420
 - 1110 1100 1010 1000 0110 0100 0010 0000

RISC-V R-format Instructions



Instruction fields

- opcode: operation code
- rd: destination register number
- funct3: 3-bit function code (additional opcode)
- rs1: the first source register number
- rs2: the second source register number
- funct7: 7-bit function code (additional opcode)

R-format Example

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

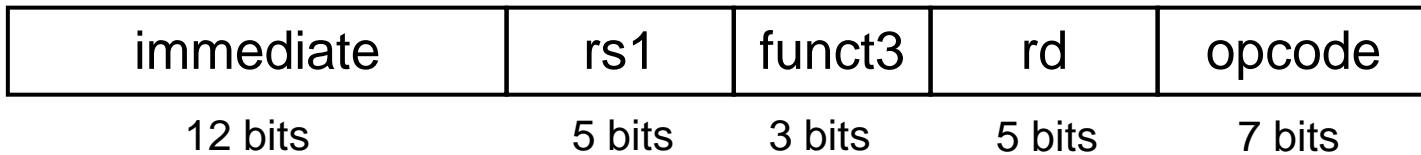
add x9, x20, x21

| | | | | | |
|---|----|----|---|---|----|
| 0 | 21 | 20 | 0 | 9 | 51 |
|---|----|----|---|---|----|

| | | | | | |
|---------|-------|-------|-----|-------|---------|
| 0000000 | 10101 | 10100 | 000 | 01001 | 0110011 |
|---------|-------|-------|-----|-------|---------|

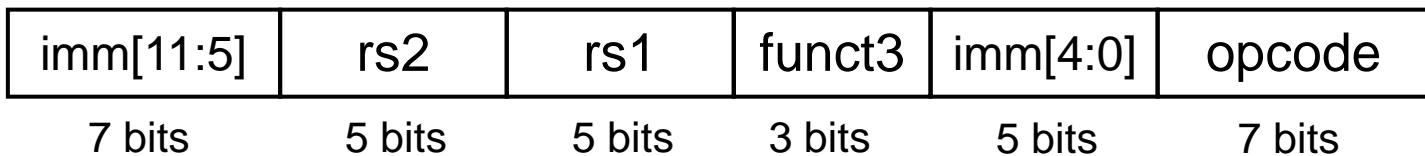
0000 0001 0101 1010 0000 0100 1011 0011_{two} =
015A04B3₁₆

RISC-V I-format Instructions



- Immediate arithmetic and load instructions
 - rs1: source or base address register number
 - immediate: constant operand, or offset added to base address
 - 2s-complement, sign extended
- *Design Principle 3: Good design demands good compromises*
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible

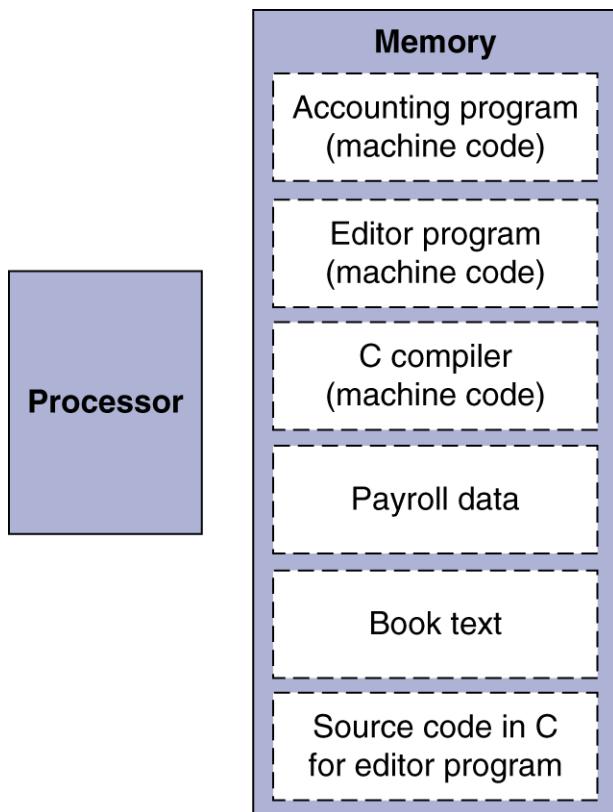
RISC-V S-format Instructions



- Different immediate format for store instructions
 - rs1: base address register number
 - rs2: source operand register number
 - immediate: offset added to base address
 - Split so that rs1 and rs2 fields always in the same place

Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
 - e.g., compilers, linkers, ...
- Binary compatibility allows compiled programs to work on different computers
 - Standardized ISAs

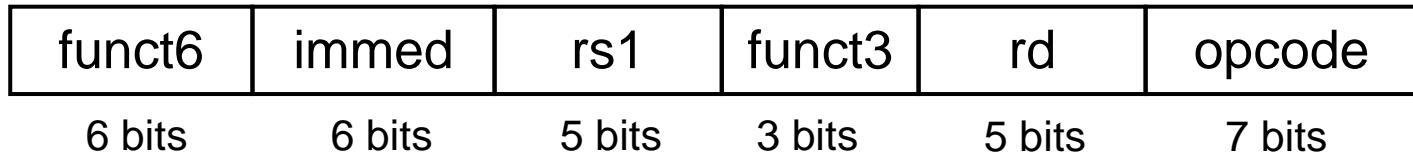
Logical Operations

- Instructions for bitwise manipulation

| Operation | C | Java | RISC-V |
|----------------|----|------|------------|
| Shift left | << | << | slli |
| Shift right | >> | >>> | srl |
| Bit-by-bit AND | & | & | and, andi |
| Bit-by-bit OR | | | or, ori |
| Bit-by-bit XOR | ^ | ^ | xor, xorri |
| Bit-by-bit NOT | ~ | ~ | |

- Useful for extracting and inserting groups of bits in a word

Shift Operations



- immed: how many positions to shift
- Shift left logical
 - Shift left and fill with 0 bits
 - $s11i$ by i bits multiplies by 2^i
- Shift right logical
 - Shift right and fill with 0 bits
 - $sr1i$ by i bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
 - Select some bits, clear others to 0

and x_9, x_{10}, x_{11}

x_{10} 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

x_{11} 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

x_9 00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

OR Operations

- Useful to include bits in a word
 - Set some bits to 1, leave others unchanged

or $x9, x10, x11$

$x10$ 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

$x11$ 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

$x9$ 00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

XOR Operations

- Differencing operation
 - Set some bits to 1, leave others unchanged

```
xor x9,x10,x12 // NOT operation
```

| | |
|-----|---|
| x10 | 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000 |
| x12 | 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 |
| x9 | 11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111 |

Conditional Operations

- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- **beq rs1, rs2, L1**
 - if ($rs1 == rs2$) branch to instruction labeled L1
- **bne rs1, rs2, L1**
 - if ($rs1 != rs2$) branch to instruction labeled L1

Compiling If Statements

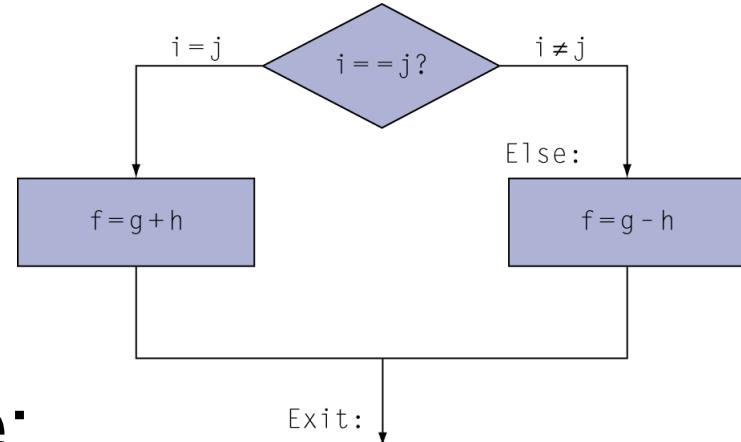
- C code:

```
if (i==j) f = g+h;  
else f = g-h;
```

- f, g, ... in x19, x20, ...

- Compiled RISC-V code:

```
bne x22, x23, Else  
add x19, x20, x21  
beq x0,x0,Exit // unconditional  
Else: sub x19, x20, x21  
Exit: ...
```



Assembler calculates addresses

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

- i in x22, k in x24, address of save in x25

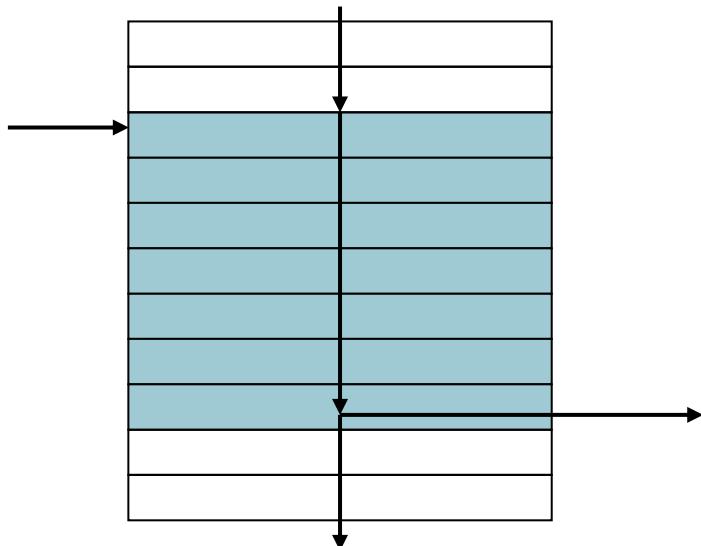
- Compiled RISC-V code:

```
Loop: slli x10, x22, 3  
       add x10, x10, x25  
       ld   x9, 0(x10)  
       bne x9, x24, Exit  
       addi x22, x22, 1  
       beq x0, x0, Loop
```

```
Exit: ...
```

Basic Blocks

- A basic block is a sequence of instructions with
 - No embedded branches (except at end)
 - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- `blt rs1, rs2, L1`
 - if ($rs1 < rs2$) branch to instruction labeled L1
- `bge rs1, rs2, L1`
 - if ($rs1 \geq rs2$) branch to instruction labeled L1
- Example
 - if ($a > b$) $a += 1$;
 - a in x22, b in x23

```
bge x23, x22, Exit      // branch if b >= a
addi x22, x22, 1
```

Exit:

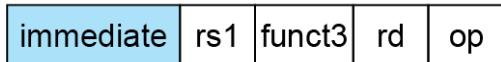


Signed vs. Unsigned

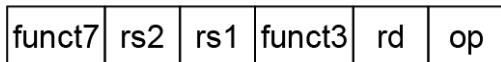
- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu
- Example
 - $x22 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$
 - $x23 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001$
 - $x22 < x23 \text{ // signed}$
 - $-1 < +1$
 - $x22 > x23 \text{ // unsigned}$
 - $+4,294,967,295 > +1$

RISC-V Addressing Summary

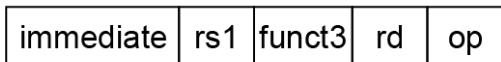
1. Immediate addressing



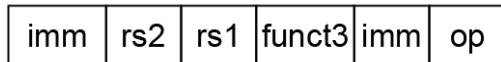
2. Register addressing



3. Base addressing



4. PC-relative addressing



RISC-V Encoding Summary

| | Name (Bit position) | 31:25 | 24:20 | 19:15 | 14:12 | 11:7 | Fields 6:0 |
|-------------|------------------------|-----------------|-------|-------|--------|---------------|---------------|
| (a) R-type | | funct7 | rs2 | rs1 | funct3 | rd | opcode |
| (b) I-type | | immediate[11:0] | | rs1 | funct3 | rd | opcode |
| (c) S-type | | immed[11:5] | rs2 | rs1 | funct3 | immed[4:0] | opcode |
| (d) SB-type | | immed[12,10:5] | rs2 | rs1 | funct3 | immed[4:1,11] | opcode |