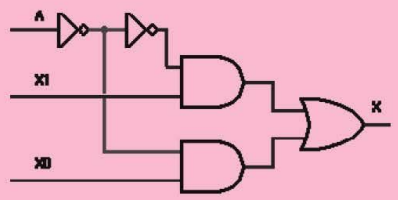
RISC-V

CS 3410: Computer System Organization and Programming

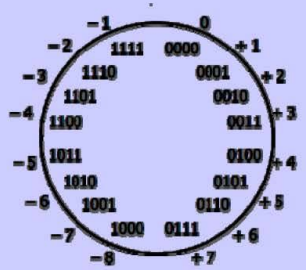
Spring 2025



to:

from:

if I were a mux I'd select you

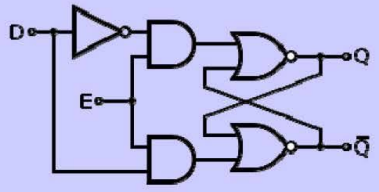


valentine, we two's complement each other perfectly <3

to:

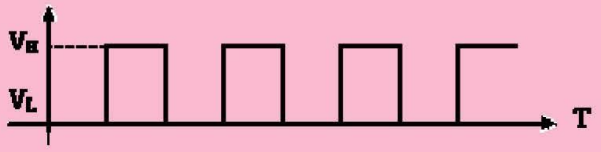
from:

you make my heart flip-flop <3



to:

from:

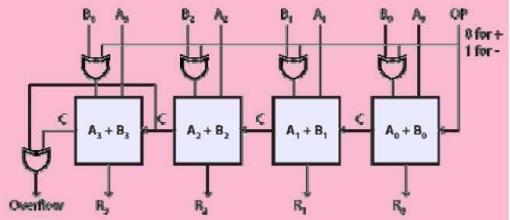


I'm falling (edge) in love with you

to: from:

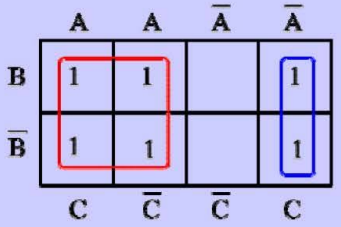
to:

from:



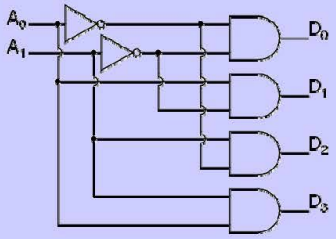
I'm overflowing with love for you

you hold the karnaugh map to my heart



to:

from:



it's not hard to decode my feelings for you <3

to: from:

1	0	0	1	0	1	0	1

are u the leftmost bit? because u are the most significant to me <3

to: from:

THIS IS WHAT LEARNING LOGIC GATES FEELS LIKE

Logistics

- **A3: Huffman Encoding** was due last night; accepted late using slip days until Saturday (2/15)
- **Lab 4: GDB** to be completed in-class today & tomorrow
- No homework over February break
  - **No class on Tuesday (2/18)!!**
- **Prelim 1** next Thursday (2/20) @ 7:30pm in STL185
  - We will have a lecture next Thursday (2/20)
  - **A4: CPUSIM** will be released Thursday (2/20)

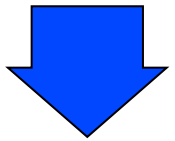
Roadmap

- Machine Code & ISAs
- RISC-V Overview
  - Instruction Encoding
  - Arithmetic Instructions
  - Logical Instructions
  - Immediate Instructions
  - Assembly Programming

Levels of Languages

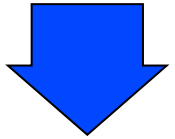
C

```
int x = 10;
x = 2 * x + 15;
```



RISC-V assembly language

```
addi x5, x0, 10
muli x5, x5, 2
addi x5, x5, 15
```



RISC-V machine code

```
00000000101000000000001010010011
00000000001000101001001010011111
00000000111100101000001010010011
```

High Level Language

- C, Java, Python, Rust, …
- Loops, control flow, variables

Assembly Language

- No symbols (except labels)
- One operation per statement
- “human readable machine language”

Machine Code

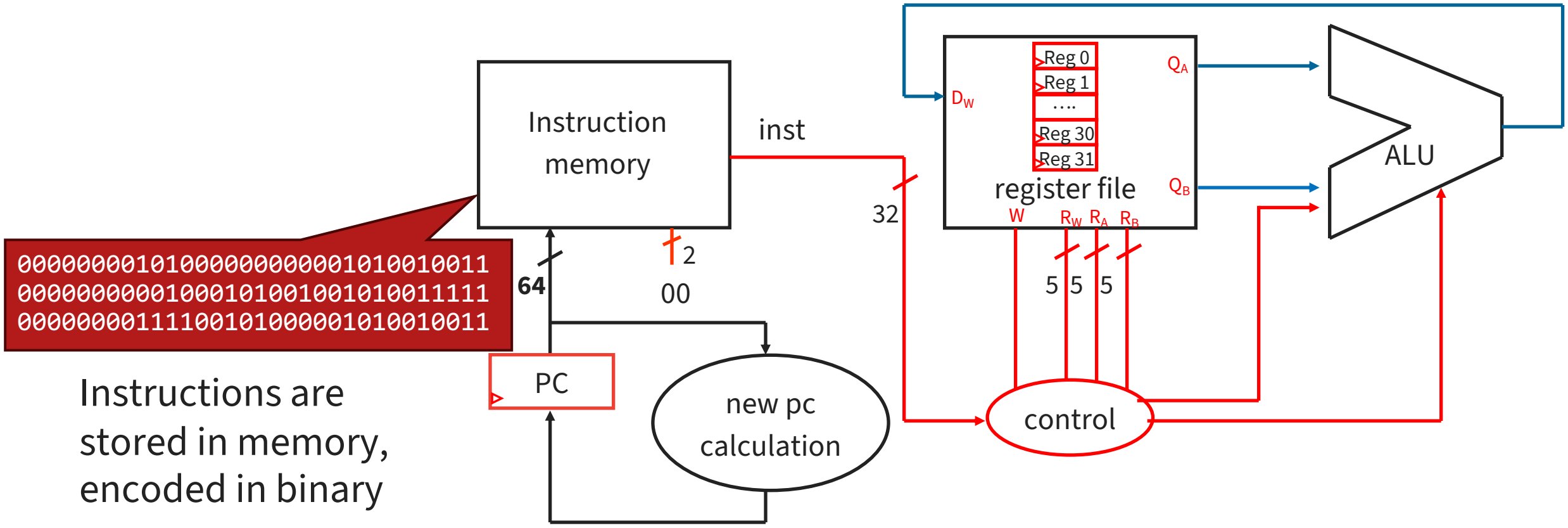
- Binary-encoded assembly
- Labels become addresses
- **The language of the CPU**

Instruction Set Architecture

ALU, Control, Register File, …

Machine Implementation (Microarchitecture)

Instruction Processing



Instructions are stored in memory, encoded in binary

RISC-V

- An *instruction set architecture (ISA)*
  - A *language* for machine code
- Design Principles
  - Simplicity favors regularity
    - 32-bit instructions
  - General purpose
  - Open source!

Why learn Assembly Programming?

- You get to understand the language that the computer *actually* speaks
- Relevant for exceptional cases:
  1. Performance-sensitive applications (e.g., FFmpeg)
  2. Operating systems
  3. Security-sensitive applications (e.g., to avoid timing attacks)
  4. Advanced diagnostics (e.g., compiler bugs!)

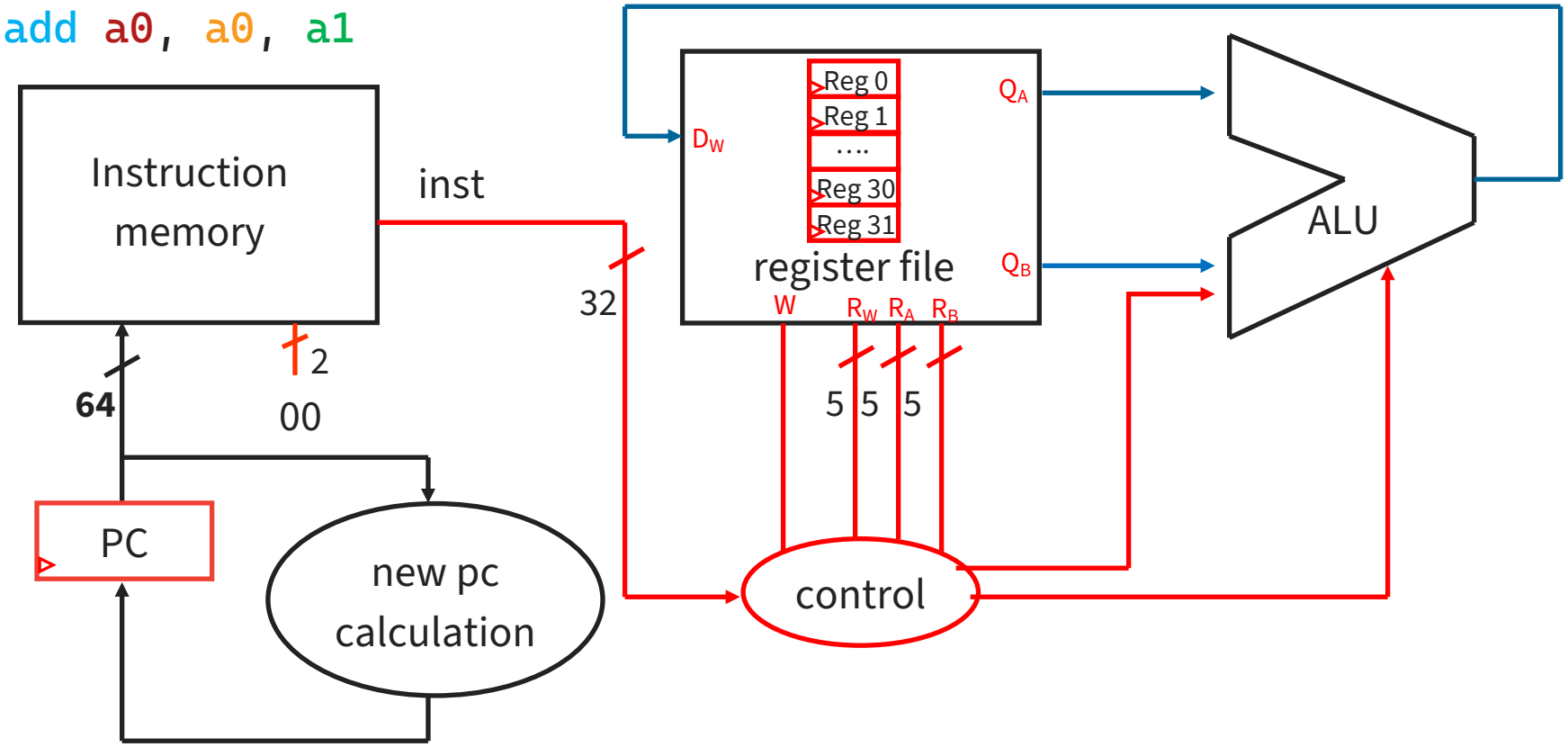
Have you programmed in assembly before?

Never I've dabbled I've written a function or two I'm an elite assembly h4x0r

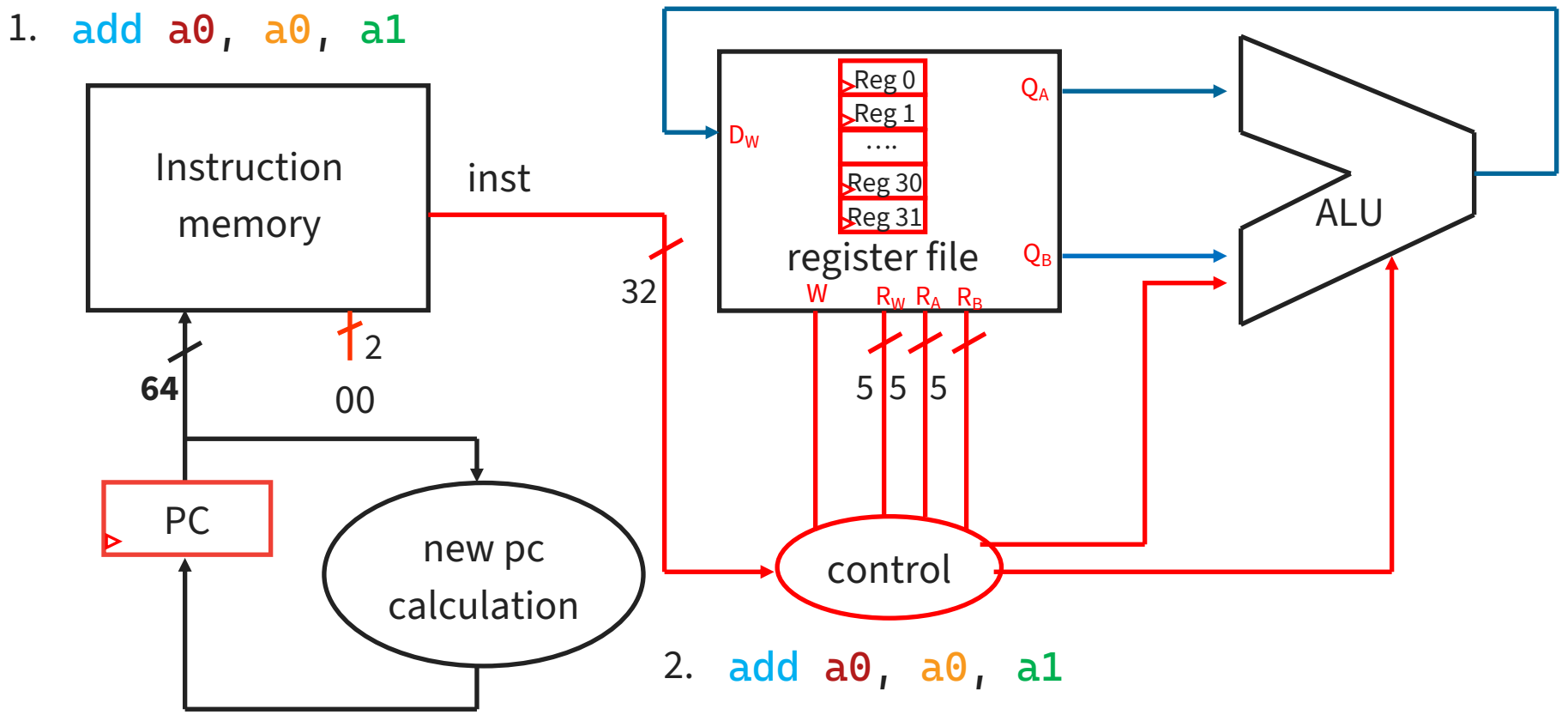
Demo

Executing `add a0, a0, a1`

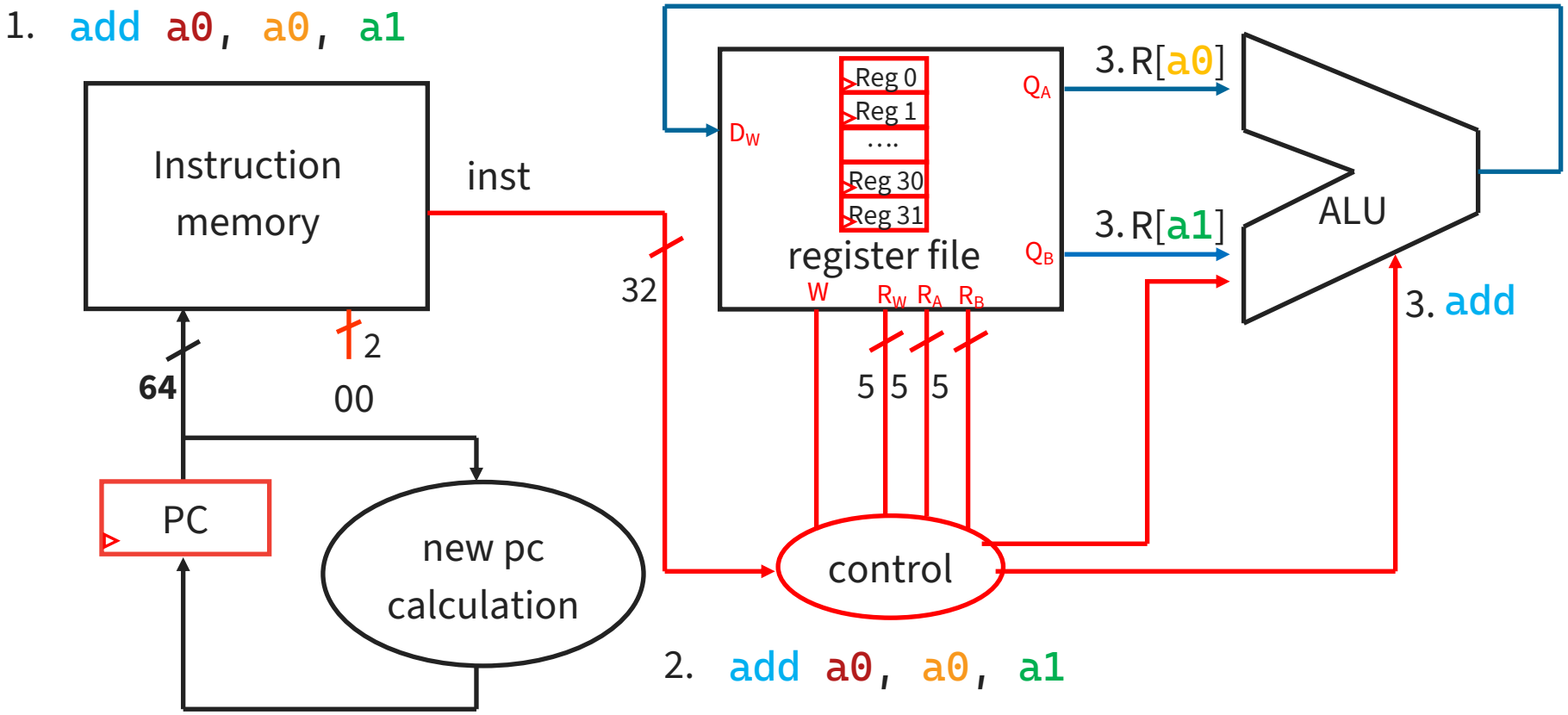
1. `add a0, a0, a1`



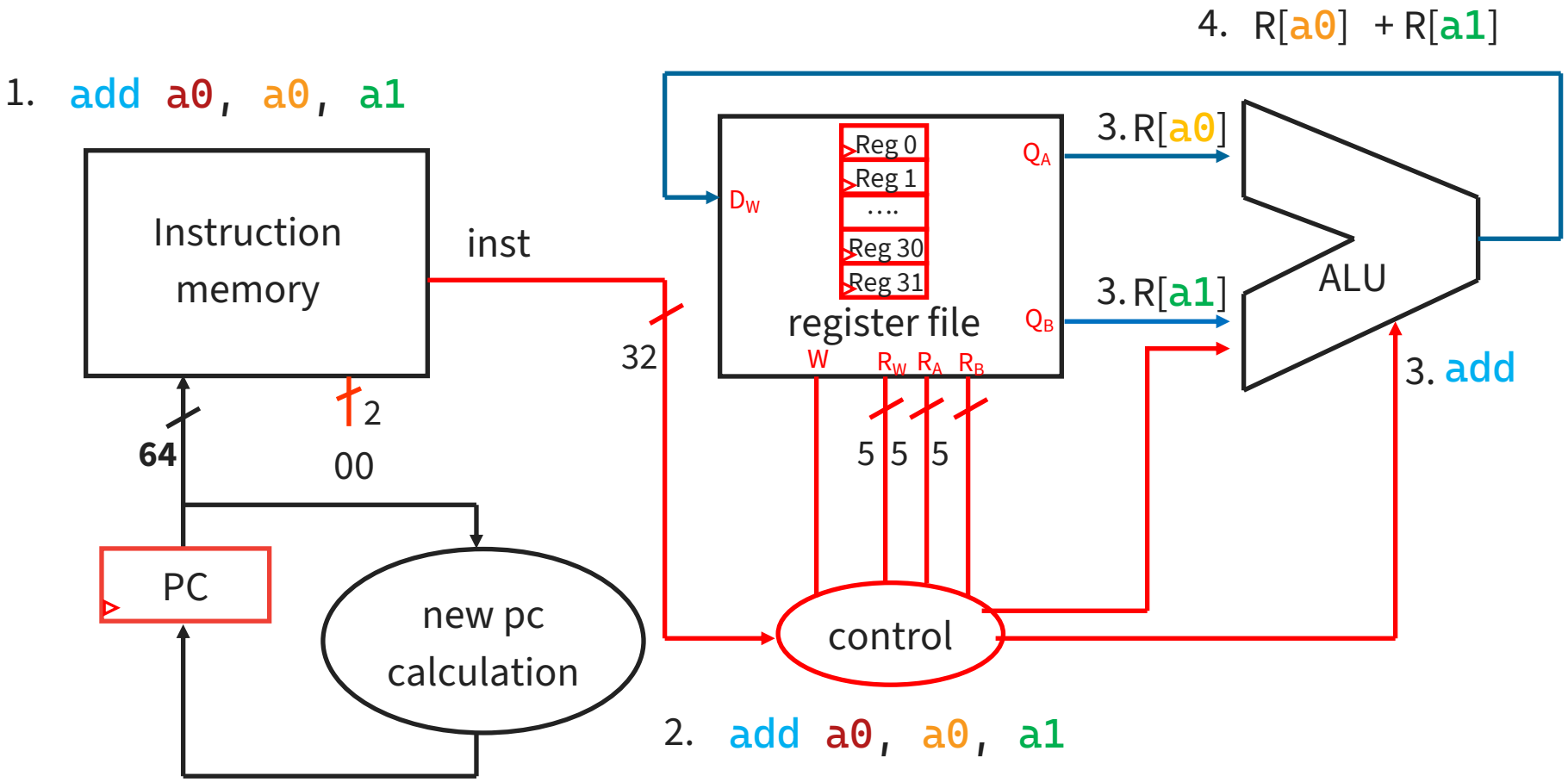
Executing `add a0, a0, a1`



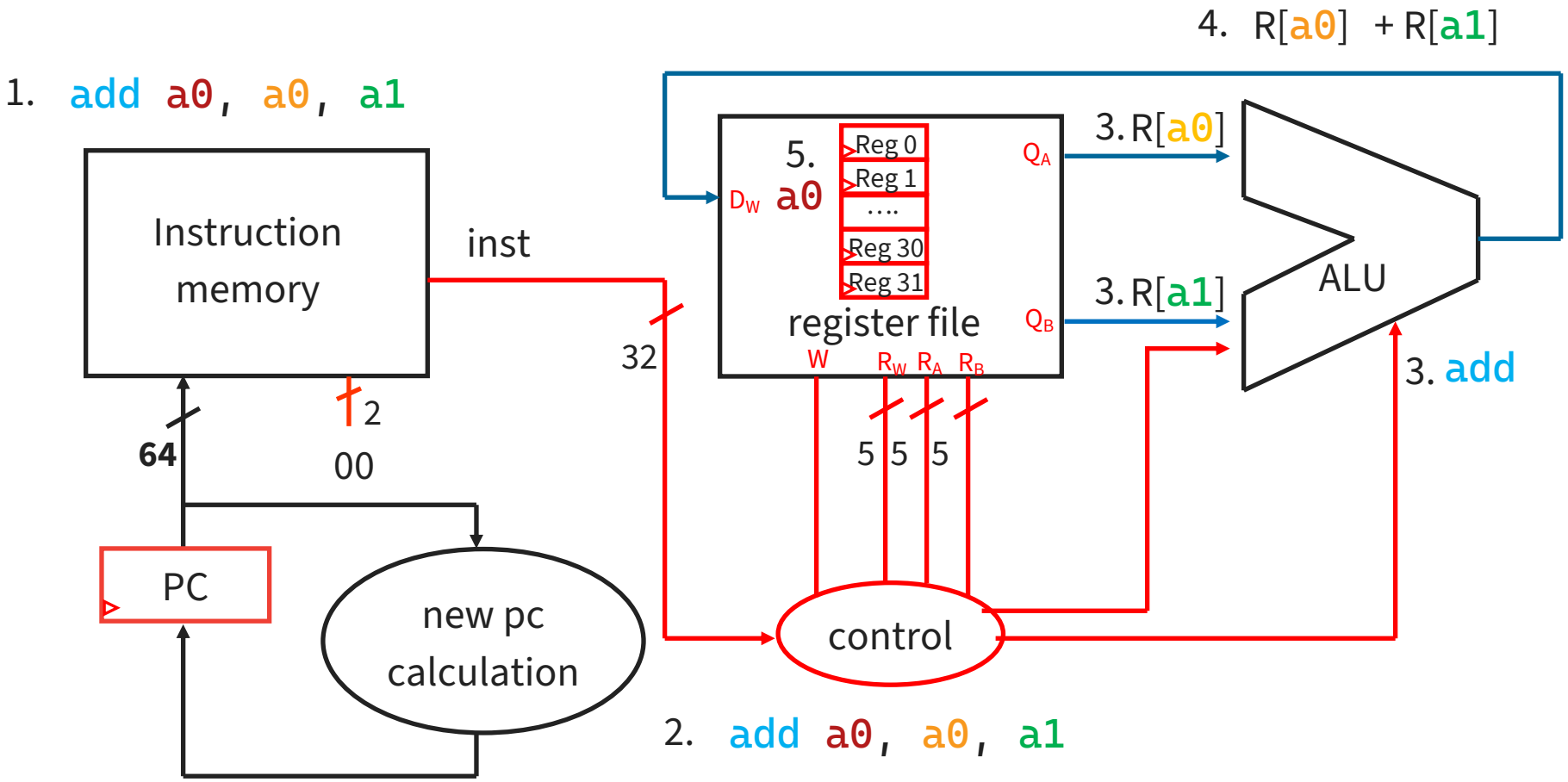
Executing `add a0, a0, a1`



Executing `add a0, a0, a1`



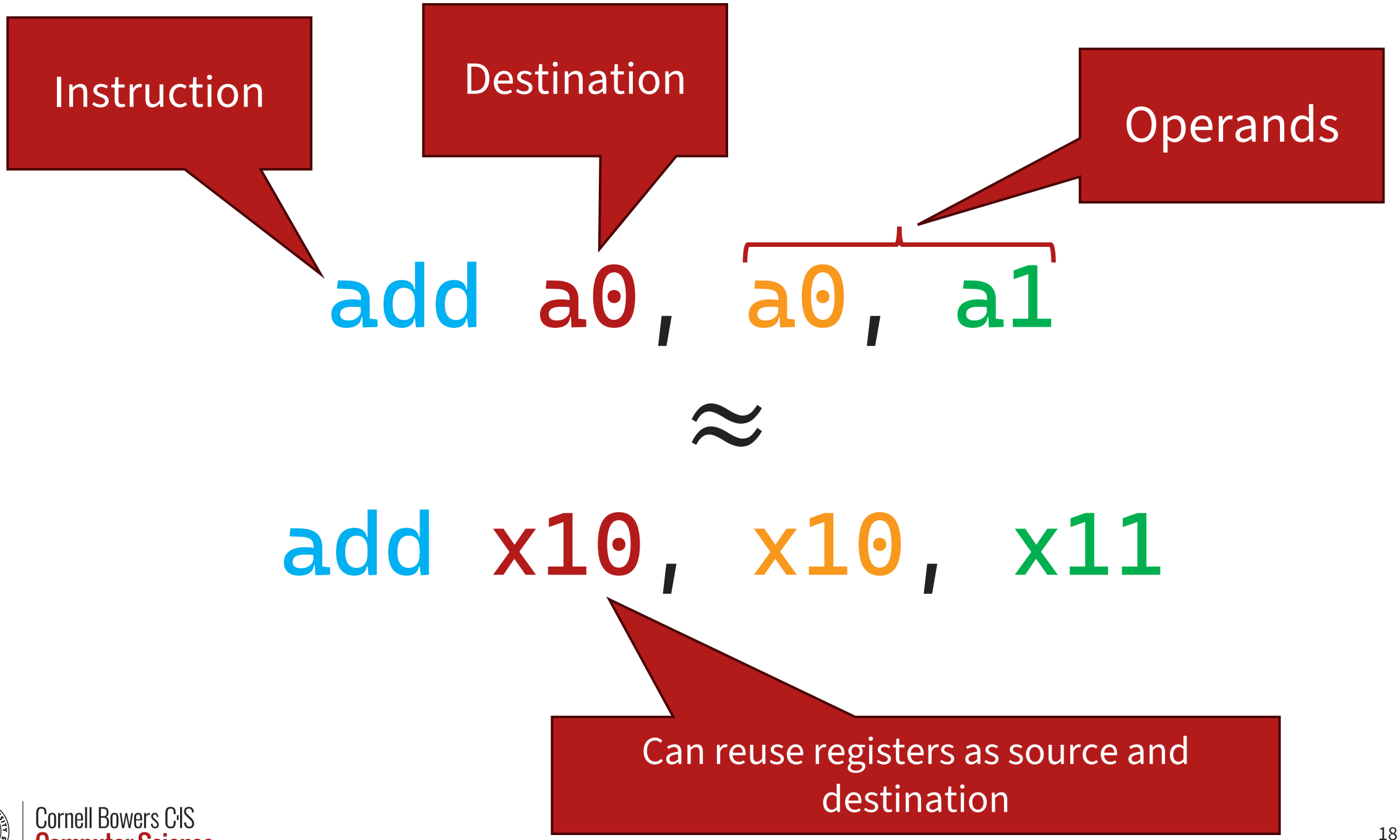
Executing `add a0, a0, a1`



Registers

- RISC-V has 32 registers
  - Each stores 64-bit integers
- Different registers are used for different purposes
  - `x0` is also known as `zero`
  - `x10` through `x17` are `a0` through `a7`
  - `x5,x6,x7,x28-x31` are `t0-t6`
  - `x8,x9,x18-x27` are `s0-s11`

Register name	Symbolic name	Description
32 integer registers		
x0	zero	Always zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate return address
x6–7	t1–2	Temporaries
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments / return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries
32 floating-point extension registers		
f0–7	ft0–7	Floating-point temporaries
f8–9	fs0–1	Floating-point saved registers
f10–11	fa0–1	Floating-point arguments/return values
f12–17	fa2–7	Floating-point arguments
f18–27	fs2–11	Floating-point saved registers
f28–31	ft8–11	Floating-point temporaries


Instruction
Destination
Operands
add a0, a0, a1
≈
add x10, x10, x11
Can reuse registers as source and destination

Instruction Types

Arithmetic

- add, subtract, shift left, shift right, multiply, divide

Memory

- load value from memory to a register
- store value to memory from a register

Control flow

- conditional jumps (branches)
- jump and link (subroutine call)

Many other instructions are possible

- vector add/sub/mul/div, string operations
- manipulate coprocessor
- I/O

RISC-V Instruction Types

Arithmetic/Logical

- **R-type:** result and two source registers, shift amount
- **I-type:** result and source register, shift amount in 12-bit immediate with sign/zero extension
- **U-type:** result register, 20-bit immediate with sign/zero extension

Memory Access

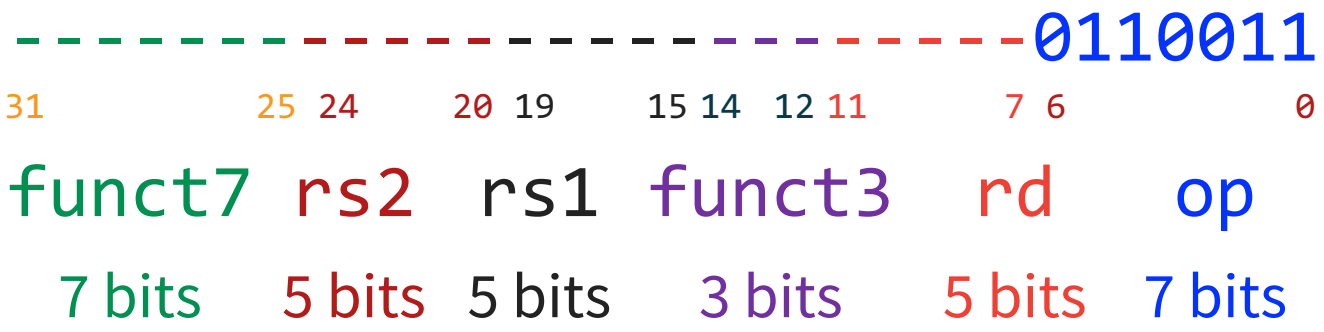
- **I-type** for loads and **S-type** for stores
- load/store between registers and memory
- word, half-word and byte operations

Control flow

- **S-type:** conditional branches: pc-relative addresses
- **U-type:** jump-and-link
- **I-type:** jump-and-link register

The Manual

R-Type (1): Arithmetic and Logic



funct7	funct3	mnemonic	description
0000000	000	ADD rd, rs1, rs2	R[rd] = R[rs1] + R[rs2]
0100000	000	SUB rd, rs1, rs2	R[rd] = R[rs1] – R[rs2]
0000000	110	OR rd, rs1, rs2	R[rd] = R[rs1] | R[rs2]
0000000	100	XOR rd, rs1, rs2	R[rd] = R[rs1] ⊕ R[rs2]

R-Type (1): Arithmetic and Logic

```
0000000 00110 01000 100 01000 110011
31      25 24 20 19  15 14 12 11  7 6   0
funct7  rs2   rs1   funct3 rd    op
7 bits  5 bits 5 bits 3 bits 5 bits 7 bits
```

funct7	funct3	mnemonic	description
0000000	000	ADD rd, rs1, rs2	R[rd] = R[rs1] + R[rs2]
0100000	000	SUB rd, rs1, rs2	R[rd] = R[rs1] – R[rs2]
0000000	110	OR rd, rs1, rs2	R[rd] = R[rs1] | R[rs2]
0000000	100	XOR rd, rs1, rs2	R[rd] = R[rs1] ⊕ R[rs2]

```
example:   x4 = x8 ⊕ x6    #XOR x4, x8, x6
                            rd, rs1,rs2
```

Aside: Truncation

- Suppose we want to convert an 8-bit value into a 4-bit value

`0000 0111` = 7 = `0111`

`0000 1111` = 15 ≠ `1111` (-1)

Aside: Zero-Extension

- Suppose we want to convert a 4-bit number into an 8-bit number

```
1 = 0001 = 0000 0001
```

Aside: Sign-Extension

- Suppose we want to convert a 4-bit *negative* number into an 8-bit number

```
-1 = 1111
```

Remember negative numbers are encoded using **Two's Complement**!

Aside: Sign-Extension

- Suppose we want to convert a 4-bit *negative* number into an 8-bit number

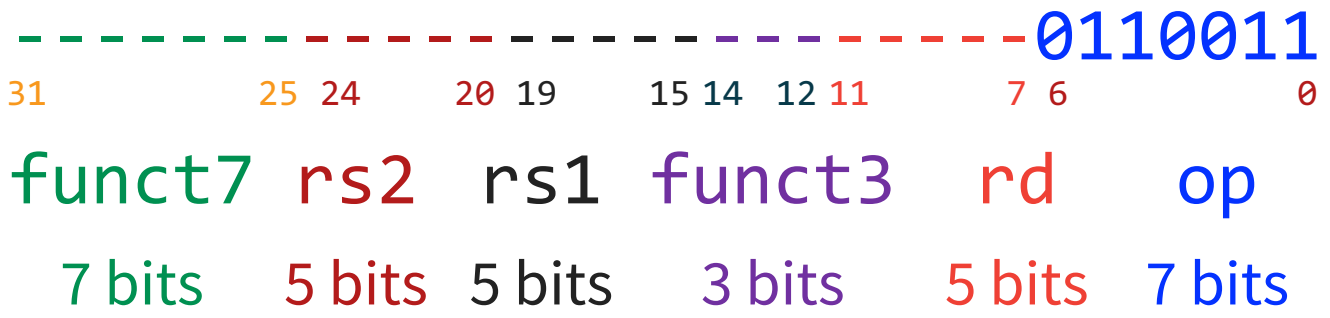
```
-1 = 1111 = 1111 1111
```

The MSB bit (i.e., the sign-bit!) is copied!

Aside: Truncation & Extension

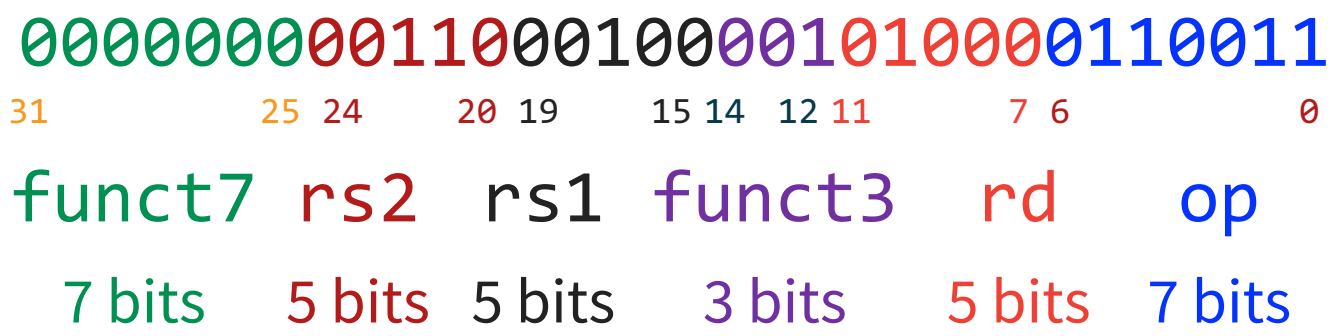
- **Truncation** *decreases* the size of a value
- **Extension** *increases* the size of a value
  - **Zero-extension** fills upper bits with 0
    - Used to extend *unsigned* numbers
  - **Sign-extension** fills upper bits with *copies of the most-significant bit*
    - Used to extend *signed* numbers

R-Type (2): Shift Instructions



funct7	funct3	mnemonic	description
0000000	001	SLL rd, rs1, rs2	R[rd] = R[rs1] << R[rs2]
0000000	101	SRL rd, rs1, rs2	R[rd] = R[rs1] >>> R[rs2] (zero ext.)
0100000	101	SRA rd, rs1, rs2	R[rd] = R[rt] >>> R[rs2] (sign ext.)

R-Type (2): Shift Instructions

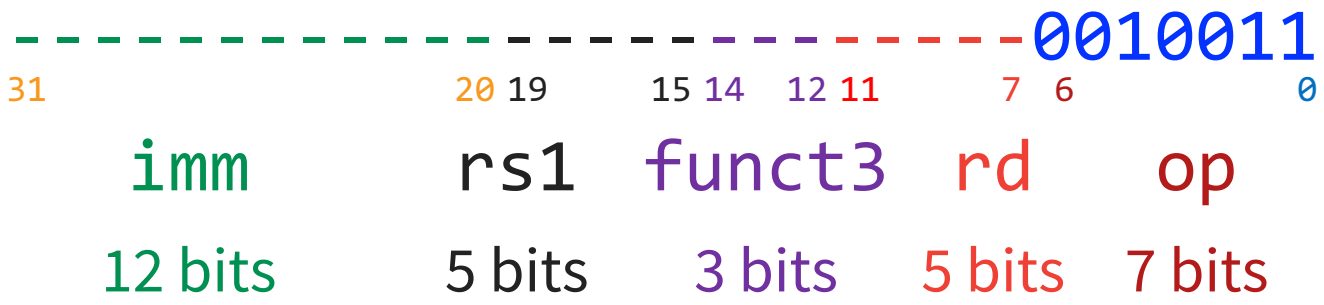


funct7	funct3	mnemonic	description
0000000	001	SLL rd, rs1, rs2	R[rd] = R[rs1] << R[rs2]
0000000	101	SRL rd, rs1, rs2	R[rd] = R[rs1] >>> R[rs2] (zero ext.)
0100000	101	SRA rd, rs1, rs2	R[rd] = R[rt] >>> R[rs2] (sign ext.)

```
example:   x8 = x4 * 2^x6    # SLL  x8, x4, x6
           x8 = x4 << x6             rd, rs1,rs2
```

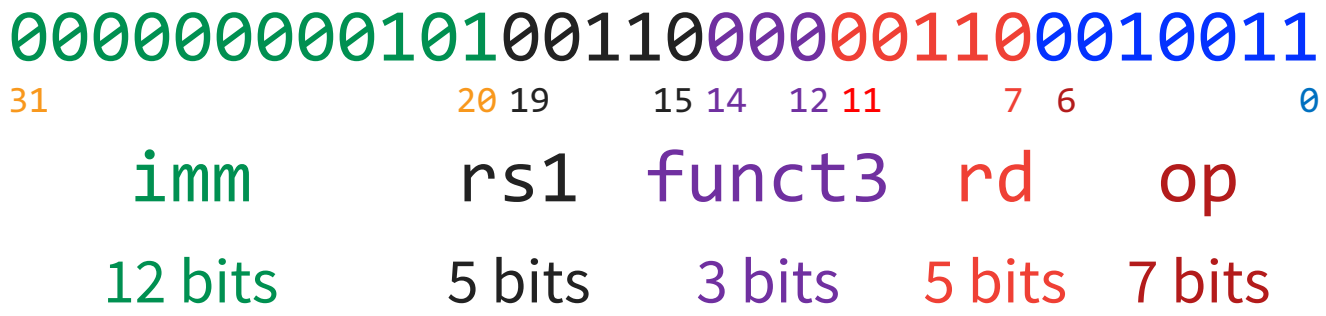
(Want to multiply by 32? → Store 5 in x6.)

I-Type (1): Arithmetic w/ immediates



funct3	mnemonic	description
000	ADDI rd, rs1, imm	R[rd] = R[rs1] + imm
111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

I-Type (1): Arithmetic w/ immediates



funct3	mnemonic	description
000	ADDI rd, rs1, imm	R[rd] = R[rs1] + imm
111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

```
example:    x6 = x6 + 5     # ADDI x6, x6, 5
            x6 += 5                rd, rs1, imm
```

To compile the code y = z + n, assuming n is an integer, y is stored in x1, and z is stored in x2, you can use the ADDI instruction. What is the largest number n for which we can continue to use ADDI?

- 12
- $2^{12-1} - 1 = 2047$
- $2^{12} - 1 = 4095$
- $2^{16} - 1 = 65535$
- $2^{32-1} - 1 \approx 2.1 \text{ billion}$

To compile the code y = z + n, assuming n is an integer, y is stored in x1, and z is stored in x2, you can use the ADDI instruction. What is the largest number n for which we can continue to use ADDI?

12 — 0%

$2^{12-1} - 1 = 2047$ — 0%

$2^{12} - 1 = 4095$ — 0%

$2^{16} - 1 = 65535$ — 0%

$2^{32-1} - 1 \approx 2.1 \text{ billion}$ — 0%

To compile the code y = z + n, assuming n is an integer, y is stored in x1, and z is stored in x2, you can use the ADDI instruction. What is the largest number n for which we can continue to use ADDI?

12 — 0%

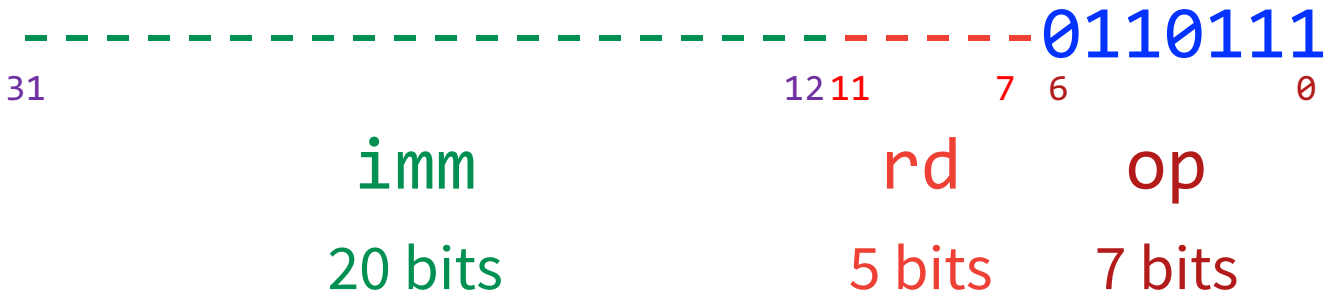
$2^{12-1} - 1 = 2047$ — 0%

$2^{12} - 1 = 4095$ — 0%

$2^{16} - 1 = 65535$ — 0%

$2^{32-1} - 1 \approx 2.1 \text{ billion}$ — 0%

U-Type (1): Load Upper Immediate



mnemonic	description
LUI rd, imm	R[rd] = imm << 12

U-Type (1): Load Upper Immediate

mnemonic	description
LUI rd, imm	R[rd] = imm << 12

```
example:    x5 = 0x5000        # LUI x5, 5
                                   rd, imm
```

Typical Usage Pattern:

```
LUI x5, 0x12345
ADDI x5, x5 0x678
```

```
                1    2    3    4    5
                0001 0010 0011 0100 0101   shift by 12:
0001 0010 0011 0100 0101 0000 0000 0000  + 0x678:
0001 0010 0011 0100 0101 0110 0111 1000
```

What is the binary encoding of srli a0, a0, 1?

0

Nobody has responded yet.

Hang tight! Responses are coming in.

Assembly Programming

Pseudocode

```
a0 = 34
a1 = a0 - 13
a2 = a1 * 2
```

Assembly

How do we put 34 into register a0?

Assembly Programming

Pseudocode

```
a0 = 0 + 34
a1 = a0 - 13
a2 = a1 * 2
```

Assembly

```
addi a0, x0, 34
```

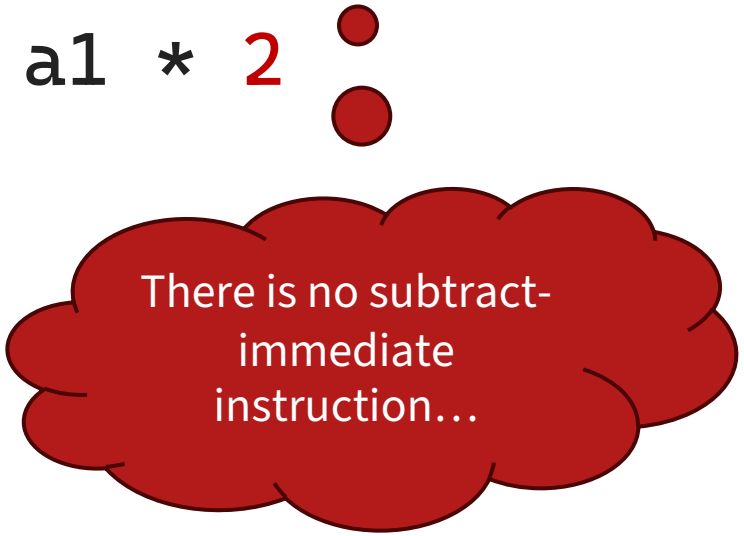
Always zero!

How do we put 34 into register a0?

Assembly Programming

Pseudocode

```
a0 = 0 + 34
a1 = a0 - 13
a2 = a1 * 2
```



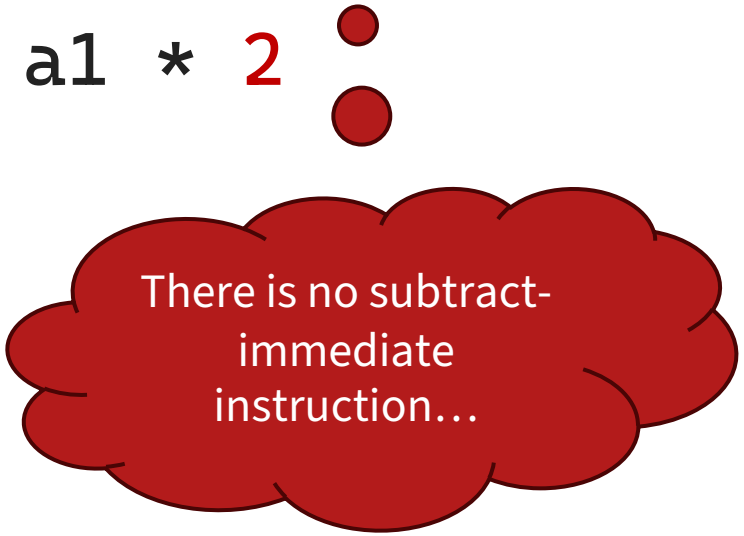
Assembly

```
addi a0, x0, 34
```

Assembly Programming

Pseudocode

```
a0 = 0 + 34
a1 = a0 - 13
a2 = a1 * 2
```



Assembly

```
addi a0, x0, 34
addi a1, a0, -13
```

Assembly Programming

Pseudocode

```
a0 = 0 + 34
a1 = a0 - 13
a2 = a1 * 2
```

Assembly

```
addi a0, x0, 34
addi a1, a0, -13
```

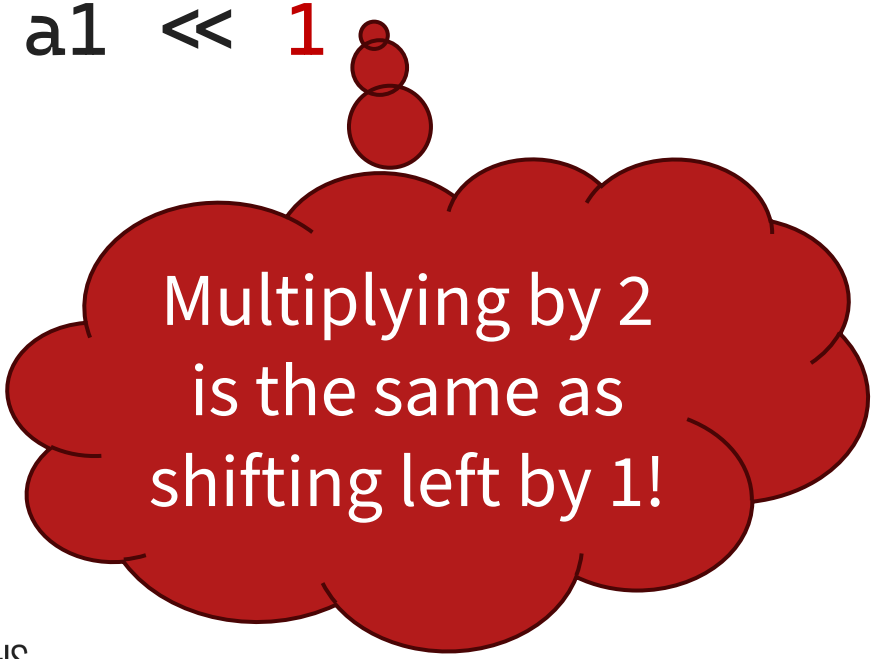
Assembly Programming

Pseudocode

```
a0 = 0 + 34
a1 = a0 - 13
a2 = a1 << 1
```

Assembly

```
addi a0, x0, 34
addi a1, a0, -13
slli a2, a1, 1
```



Takeaways

- Machine code (encoded in binary) is the *language* of the computer
- ISAs provide *meaning* to the machine code
  - RISC-V, Arm, x86
- Assembly instructions tell the processor what to do
  - These instructions have a specific binary encoding

Everything is just bits!