# RISC-V Instruction Overview

## Introduction to Computer Systems, Fall 2022

**Instructors:** Joel Ramirez Travis McGaha

**Head TAs:** Adam Gorka Daniel Gearhardt
Ash Fujiyama Emily Shen

**TAs:**

Ahmed Abdellah
Angie Cao
August Fu
Caroline Begg
Cathy Cao
Claire Lu
Eric Sungwon Lee
Ethan Weisberg
Garrett O'Malley Kirsch
Hassan Rizwan
Iain Li
Jerry Wang
Juan Lopez
Keith Mathe
Maya Huizar
Meghana Vasireddy
Perrie Quek
Sidharth Roy
Sydnie-Shea Cohen
Vivi Li
Yousef AlRabiah

# Poll Everywhere

**pollev.com/cis2400**

- How are you? Any Questions?

# Logistics

- Yay the midterm is over!
    - Maybe was a bit longer than planned -- Duly noted.
    - Aiming to have grades out by Friday but no promises!!!!!!!
- The Last Written HW is due this Friday
    - Already know about the extension ***hard*** deadline
- Mid Semester Survey Posted
    - Due this Saturday!
- Recitation; there will be two this week
    - Recitation 1: Wednesday 4 pm DRL 3C6
    - Recitation 2: Wednesday 7:30 pm Towne 100.
    - Most Attendance wins for rest of semester

# Lecture Outline

- What is an ISA
  - Why RISC-V?
  - X86 & ARM
- RISC-V
  - R- and I-Type Instructions
  - Memory Operations
- Machine Code
  - RISC-V Encoding
  - In Memory
- The Program Counter
  - AUIPC
  - JALR

# What is an ISA?

- Instruction Set Architecture
  - Interaction with Hardware must occur in a specific language
    - These are ***ISA***'s
    - X86, Arm, and RISC-V are just a couple
  - ISA are composed of instructions that perform ***'basic'*** operations
    - Arithmetic
      - Add, Sub, Div, Mult,…
    - Memory
      - Load, Store,…
  - Complex vs Reduced
    - Complex means instructions can do more than one thing
    - Reduced means instructions only do one thing at a time
      - Requires more instructions to do the same thing as complex but more simple

# Why Choose RISC-V®

- Berkeley Developed RISC-V in 2010
- Unlike other Academic ISA’s, made to ***be used***
- Wanted to create a license free ISA
  - Royalty Free and Open Source!
  - Heavily supported!

# x86

Intel 8086

- x86 began in 1978
  - **CISC (Complex Instruction Set Computer)**
  - HUGE EMPHASIS ON BACKWARDS COMPATIBLE
    - 16 bit user-applications should work on the 32 bit ISA
    - 32 bit user-applications should work on the 64 bit ISA
  - Processors break down instructions into smaller pieces before executing…
  - A lot of baggage from eternal support of all previous versions.
  - You can reverse engineer x86 and try to implement it on your own
    - More than likely will be sued because everything is patented; including how registers are set up, how memory is loaded, etc.
  - Intel and AMD own most/all of the IP of x86.

# arm

- Idea started in 1981 by Acorn Computers

Acorn

  - Wanted to compete with Intel and Apple
  - Inspired by Berkeley's RISC lectures and publications
  - When they saw Highschoolers design chip layouts they thought
    - “yeah, we could do this too”.
    - “and make it even better?!”
  - ARM became a joint venture with Apple in 1990.

# arm

- Big in the Smart-device space
  - Made it the most ***used*** ISA in the world
  - Apple uses ARM now!
- Why use ARM?
  - Anyone can use it! As long as you pay a license fee ☺
  - Can use the ISA itself and design your own chip (Apple)
  - Or, you can use ARMs ISA and Designs (Samsung)

Apple A4, Armv7, 2010
Apple “Designed”, Made by Samsung

Apple M1, Armv8.5-A, 2020
Apple ***Designed***

# Lecture Outline

- What is an ISA
  - Why RISC-V?
  - X86 & ARM
- RISC-V
  - R- and I-Type Instructions
  - Memory Operations
- Machine Code
  - RISC-V Encoding
  - In Memory
- The Program Counter
  - AUIPC
  - JALR

# RISC-V Instruction: Register-Type

```
instruction destination, source1, source2    RISC-V
```

- instruction
    - what operation we'll do
- destination
    - Where the result will be stored
    - It is a register
- source1 and source2
    - what the arguments/operands of the instruction are
    - Both registers

***There are 6 type of instruction formats, this slide is the R-type***

# RISC-V Instruction Immediate-Type

```
instruction destination, source1, immediate
```

RISC-V

```
instruction destination, immediate(source1)
```

RISC-V

we'll see the semantic difference between these two uses as we go on

- instruction
  - what operation we'll do
- destination
  - Where the result will be stored
  - Register
- Source1 & Immediate
  - what the arguments/operands of the instruction are
  - Source1 is a register
  - Immediate is a constant fixed value

***There are 6 type of instruction formats, this slide is the I-type***

# Your First RISC-V Instructions

C code

```
b = (a + 5)
a =  a + b;
```

RISC-V

```
addi b, a, 5
add  a, a, b
```

## addi rd, rs1, imm12

- Used to add together ***rs1*** and an ***imm12 (12 bit immediate)***
- Value is stored in ***rd***
- ***I-Type Format (Immediate Format)***

## add rd, rs1, rs2

- Used to add together ***two registers, rs1 and rs2***
- Value is stored in **rd**
- ***R-Type Format (Register Format)***

In general, each r-type has a corresponding i-type except for some instructions (e.g. there is NO subi)

**C doesn't translate into assembly this way; this is just a comparison for learning**

# RISC-V and her many Registers

- We don’t operate on ***variables***, we operate on registers.
  - We have 32 of these: (x0, x1, x2 … x31)
  - Each Register is 32 bits!
  - Some registers are saved for specific purposes
    - E.g. (***x0 is the zero register***, x2 is the Stack Pointer Register,..)
    - We’ll get more into this on Thursday

We’ll assume a and b are assigned to temporary registers x5, x6 respectively

C code

```
b =  (a + 5)
a =   a + b;
```

RISC-V

```
addi x6, x5, 5
add  x5, x5, x6
```

**Now, you’re seeing two legitimate RISC-V instructions**

# Poll Everywhere

**pollev.com/cis2400**

- Variables **a**, **b**, **c**, **d**, and **e** are assigned to registers x5, x6, x7, x10, x11 respectively
- How many RISC-V instructions are necessary to implement the following C code

```
e = (a + b + c + d);
```

C code

A) 1

B) 2

C) 3

D) 4

E) I don’t know

# Poll Everywhere

pollev.com/cis2400

- Variables **a**, **b**, **c**, **d**, and **e** are assigned to registers x5, x6, x7, x10, x11 respectively
- How many RISC-V instructions are necessary to implement the following C code

```
e = (a + b + c + d);
```

C code

A) 1

B) 2

C) 3

D) 4

E) I don’t know

```
add  x5, x5, x6    \\a = a + b
add  x7, x5, x7    \\c = a + c
add  x11, x10, x7 \\e = d + c
```

RISC-V

# Poll Everywhere

**pollev.com/cis2400**

- Variables **a**, **b**, **c**, **d**, and **e** are assigned to registers x5, x6, x7, x10, x11 respectively
- How many RISC-V instructions are necessary to implement the following C code

```
d = (a + b + c + d);
e = d;
```

C code

A) 1

B) 2

C) 3

D) 4

E) I don’t know

# Poll Everywhere

**pollev.com/cis2400**

- Variables **a**, **b**, **c**, **d**, and **e** are assigned to registers x5, x6, x7, x10, x11 respectively
- How many RISC-V instructions are necessary to implement the following C code

C code

```
d = (a + b + c + d);
e = d;
```

A) 1

B) 2

C) 3

D) 4

E) I don't know

RISC-V

```
add  x5, x5, x6    \\a = a + b
add  x7, x5, x7    \\c = a + c
add  x11, x10, x7 \\e = d + c
```

We can use the exact same assembly!

We could even use x11 to be both **d** and **e**

# RISC-V: No Copy/Move Instruction?

What if you wanted to make it follow the code ***more closely?***

C code

```
d = (a + b + c + d);
e = d; //this is explicit in the assembly
```

RISC-V

```
add  x5, x5, x6   \\a = a + b
add  x7, x5, x7   \\c = a + c
add  x10, x10, x7 \\d = d + c
addi x11, x10, 0  \\e = d + 0
```

There is no 'copy' or 'move' instructions; we use other instructions to our advantage!

`addi rd, rs1, 0` is move/copy!

**Although we only removed one instruction, this demonstrates a core principle of *optimization*:

reducing the number of instructions to improve efficiency**

# Lecture Outline

- What is an ISA
  - Why RISC-V?
  - X86 & ARM
- RISC-V
  - R- and I-Type Instructions
  - Memory Operations
- Machine Code
  - RISC-V Encoding
  - In Memory
- The Program Counter
  - AUIPC
  - JALR

# RISC-V Memory Operations

- We’re limited to 32 registers (and some are *always* in use)
  - So we need to store variables and other stuff in memory
- All values stored in memory (Stack, Heap, Etc.) must be put on register before used in operations
  - WE CAN NOT USE A VALUE IN MEMORY IN AN OPERATION
  - WE CAN NOT USE A VALUE IN MEMORY IN AN OPERATION
  - WE CAN NOT USE A VALUE IN MEMORY IN AN OPERATION
  - WE CAN NOT USE A VALUE IN MEMORY IN AN OPERATION
  - WE CAN NOT USE A VALUE IN MEMORY IN AN OPERATION
  - WE CAN NOT USE A VALUE IN MEMORY IN AN OPERATION

# RISC-V Memory Operations

- Values must be read/loaded from memory into registers before use.
- The instruction `add rd, rs1, rs2` only adds the values already in the registers.
- If `rs2` holds the address of an integer, the value must be loaded into a register before performing the addition.

# RISC-V Quick Vocab

## Vocabulary: ***Word***

- A ‘word’ in RISC-V is 32 Bits
- Size of Registers is 32 Bits.
- So each register can hold a word of memory

# RISC-V Memory: Load

`lw dest, imm12(sr1) //load word`

- dest
  - destination register
- `sr1`:
  - address in memory (called the base register)
- imm12:
  - 12 bit immediate value that is the offset from `sr1`

this loads a word amount of memory (32 bits)
from memory location **`sr1 + imm12`**

```
lw  x5, 40(x6)  //x5 = *(x6 + 40)
```
RISC-V

***There are 6 type of instruction formats, this instruction uses the I-type***

# RISC-V Load Word

```
int x = arr[1];
```
C code

*arr is an array of ints*

Assumptions: x will be stored in x5, array's address is in x6

```
lw  x5, 4(x6)
```
RISC-V

Reminder: Pointer arithmetic works in increments based on the size of the type the pointer points to.

This is the same thing as:

```
int x = *(arr + 1);
```
C code

If arr is an array of integers, then arr + 1 moves the pointer by the size of one integer (4 bytes, in this case).

# RISC-V Load Half Word

`short x = arr[1];` C code

*arr is an array of shorts*

Assumptions: x will be stored in x5, array's address is in x6

`lh  x5, 2(x6)` RISC-V

This is the same thing as: `short x = *(arr + 1);` C code

What about unsigned 2-byte types? *They get a special instruction.*

`unsigned short x = arr[1];` C code

`lhu  x5, 2(x6)` RISC-V

Why? 🤔

load half-word unsigned

# RISC-V Load Half Word

`short x = arr[1];` C code

arr is an array of shorts

Assumptions: x will be stored in x5, array's address is in x6

`lh x5, 2(x6)` RISC-V

This is the same thing as: `short x = *(arr + 1);` C code

What about unsigned 2-byte types? *They get a special instruction.*

`unsigned short x = arr[1];` C code

`lhu x5, 2(x6)` RISC-V

load half-word unsigned

Why?

`lh` *sign extends* to the entire register width
lhu *zero extends* to the entire register width

There is a load byte/load byte unsigned too!

# RISC-V Store

`sw rs2, imm12(rs1) //store word`

- `rs2`
  - source register that contains what we'll store
- rs1:
  - source register that contains base address in memory
- imm12:
  - 12 bit immediate value that is the offset from rs1

```
sw x5, 40(x6) //*(x6 + 40) = x5
```

RISC-V

We also have `store half-word(sh)`, and `store byte(sb)`

**`NO UNSIGNED VERSIONS`**

***There are 6 type of instruction formats, this instruction uses the S-type***

# Practice: Translate this to RISC-V

C code

```
int x = arr[1] + arr[2];
int y = x + x;
arr[3] = y;
unsigned int z = arr[0];
```

| | |
|---|---|
| x5 | address of arr |
| x6 | `int x` |
| x7 | `int y` |
| x10 | `int z` |

RISC-V

```
............
```

`x28, x29, x30, x31` are available for any temp values

# Practice: Translate this to RISC-V

C code

```
int x = arr[1] + arr[2];
int y = x + x;
arr[3] = y;
unsigned int z = arr[0];
```

RISC-V

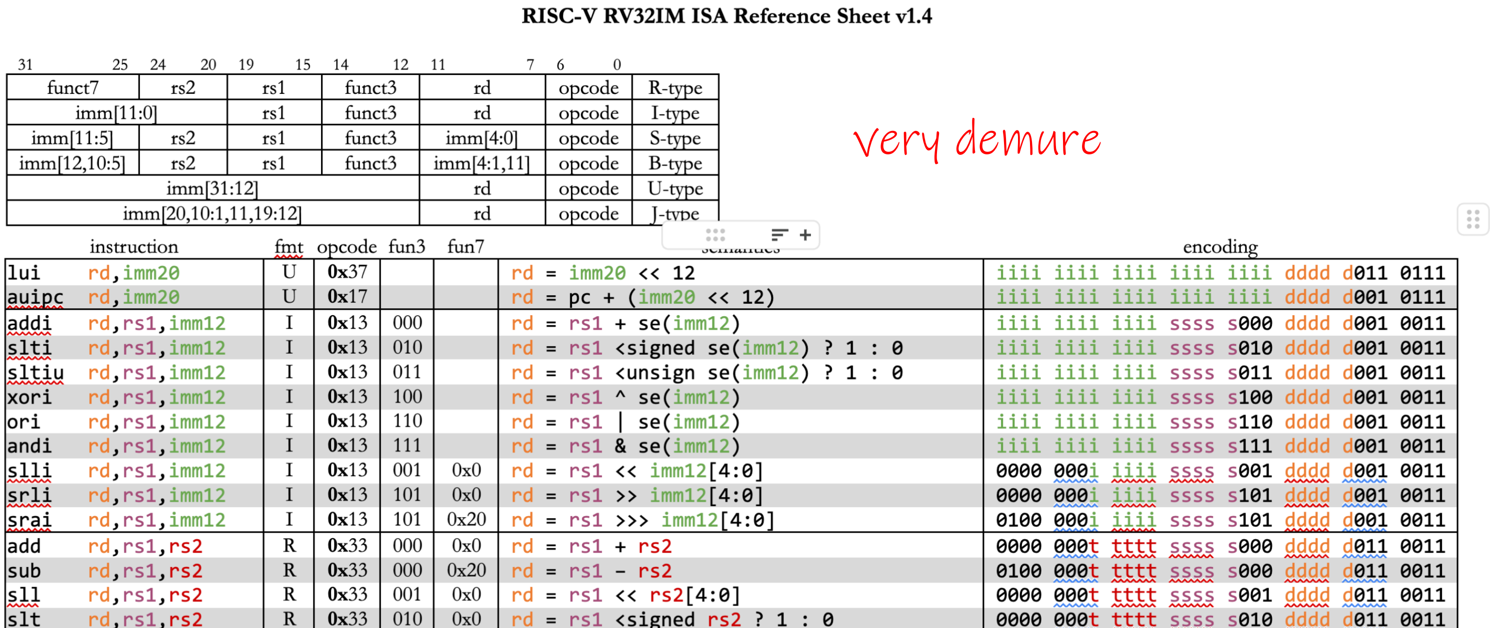
```
lw x28, 4(x5) //load arr[1]
lw x29, 8(x5) //load arr[2]
add x6, x28, x29 //x = ...
add x7, x6, x6   //y = ...
sw x7, 12(x5) // *(x5 + 12)= x7
lw x10, 0(x5)
```

| | |
|---|---|
| `x5` | address of arr |
| `x6` | `int x` |
| `x7` | `int y` |
| `x10` | `int z` |

`x28, x29, x30, x31` are available for any temp values

# RISC-V “Cheat Sheet”

- Contains every RISC-V instruction, its behavior, and other information we will discuss later
  - On the website under “references”
  - HIGHLY recommend you print a copy
  - Will be provided on exams if needed



**RISC-V RV32IM ISA Reference Sheet v1.4**

| 31 25 | 24 20 | 19 15 | 14 12 | 11 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[12,10:5] | rs2 | rs1 | funct3 | imm[4:1,11] | opcode | B-type |
| imm[31:12] | | | | rd | opcode | U-type |
| imm[20,10:1,11,19:12] | | | | rd | opcode | J-type |

very demure

| instruction | | fmt | opcode | fun3 | fun7 | semantics | encoding |
|---|---|---|---|---|---|---|---|
| lui | rd,imm20 | U | 0x37 | | | rd = imm20 << 12 | iiii iiii iiii iiii iiii dddd d011 0111 |
| auipc | rd,imm20 | U | 0x17 | | | rd = pc + (imm20 << 12) | iiii iiii iiii iiii iiii dddd d001 0111 |
| addi | rd,rs1,imm12 | I | 0x13 | 000 | | rd = rs1 + se(imm12) | iiii iiii iiii ssss s000 dddd d001 0011 |
| slti | rd,rs1,imm12 | I | 0x13 | 010 | | rd = rs1 <signed se(imm12) ? 1 : 0 | iiii iiii iiii ssss s010 dddd d001 0011 |
| sltiu | rd,rs1,imm12 | I | 0x13 | 011 | | rd = rs1 <unsign se(imm12) ? 1 : 0 | iiii iiii iiii ssss s011 dddd d001 0011 |
| xori | rd,rs1,imm12 | I | 0x13 | 100 | | rd = rs1 ^ se(imm12) | iiii iiii iiii ssss s100 dddd d001 0011 |
| ori | rd,rs1,imm12 | I | 0x13 | 110 | | rd = rs1 \| se(imm12) | iiii iiii iiii ssss s110 dddd d001 0011 |
| andi | rd,rs1,imm12 | I | 0x13 | 111 | | rd = rs1 & se(imm12) | iiii iiii iiii ssss s111 dddd d001 0011 |
| slli | rd,rs1,imm12 | I | 0x13 | 001 | 0x0 | rd = rs1 << imm12[4:0] | 0000 000i iiii ssss s001 dddd d001 0011 |
| srli | rd,rs1,imm12 | I | 0x13 | 101 | 0x0 | rd = rs1 >> imm12[4:0] | 0000 000i iiii ssss s101 dddd d001 0011 |
| srai | rd,rs1,imm12 | I | 0x13 | 101 | 0x20 | rd = rs1 >>> imm12[4:0] | 0100 000i iiii ssss s101 dddd d001 0011 |
| add | rd,rs1,rs2 | R | 0x33 | 000 | 0x0 | rd = rs1 + rs2 | 0000 000t tttt ssss s000 dddd d011 0011 |
| sub | rd,rs1,rs2 | R | 0x33 | 000 | 0x20 | rd = rs1 - rs2 | 0100 000t tttt ssss s000 dddd d011 0011 |
| sll | rd,rs1,rs2 | R | 0x33 | 001 | 0x0 | rd = rs1 << rs2[4:0] | 0000 000t tttt ssss s001 dddd d011 0011 |
| slt | rd,rs1,rs2 | R | 0x33 | 010 | 0x0 | rd = rs1 <signed rs2 ? 1 : 0 | 0000 000t tttt ssss s010 dddd d011 0011 |

# All Arithmetic Instructions in RISC-V

## Register Type

| | |
|---|---|
| add rd,rs1,rs2 | rd = rs1 + rs2 |
| sub rd,rs1,rs2 | rd = rs1 – rs2 |
| sll rd,rs1,rs2 | rd = rs1 << rs2[4:0] |
| slt rd,rs1,rs2 | rd = rs1 < signed rs2 ? 1 : 0 |
| sltu rd,rs1,rs2 | rd = rs1 < unsign rs2 ? 1 : 0 |
| xor rd,rs1,rs2 | rd = rs1 ^ rs2 |
| srl rd,rs1,rs2 | rd = rs1 >> rs2[4:0] |
| sra rd,rs1,rs2 | rd = rs1 >>> rs2[4:0] |
| or rd,rs1,rs2 | rd = rs1 \| rs2 |
| and rd,rs1,rs2 | rd = rs1 & rs2 |

Note: remember how sub didn't have an immediate version?

That's because addi does sign extension! So we can add negative values...

## Immediate Type

| | |
|---|---|
| addi rd,rs1,imm12 | rd = rs1 + se(imm12) |
| slti rd,rs1,imm12 | rd = rs1 <signed se(imm12) ? 1 : 0 |
| sltiu rd,rs1,imm12 | rd = rs1 <unsign se(imm12) ? 1 : 0 |
| xori rd,rs1,imm12 | rd = rs1 ^ se(imm12) |
| ori rd,rs1,imm12 | rd = rs1 \| se(imm12) |
| andi rd,rs1,imm12 | rd = rs1 & se(imm12) |
| slli rd,rs1,imm12 | rd = rs1 << imm12[4:0] |
| srli rd,rs1,imm12 | rd = rs1 >> imm12[4:0] |
| srai rd,rs1,imm12 | rd = rs1 >>> imm12[4:0] |

As we talk about how these instructions are turned into machine code It will make more sense!

# Lecture Outline

- What is an ISA
  - Why RISC-V?
  - X86 & ARM
- RISC-V
  - R- and I-Type Instructions
  - Memory Operations
- Machine Code
  - RISC-V Encoding
  - In Memory
- The Program Counter
  - AUIPC
  - JALR

# Instruction Encodings

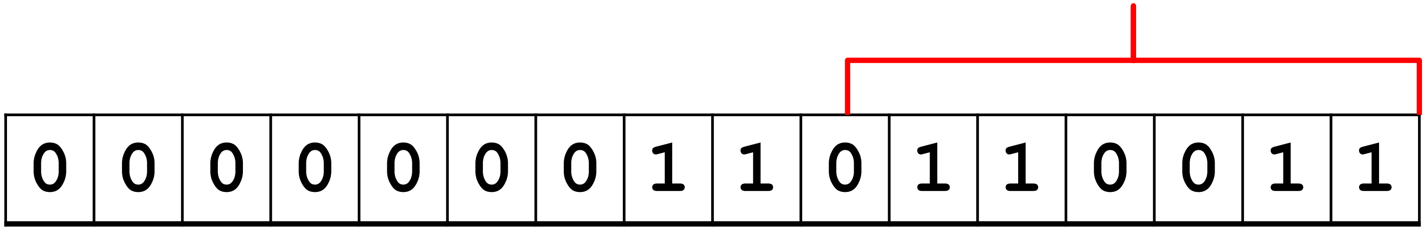
- Instructions are stored in memory over the lifetime of the program
- All Instructions are the 'same length', 32 bits
- These 32 bits can be read to:
  - Identify the instruction
  - Identify the registers used in that instruction
  - Identify any integer constants used in that instruction

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

R-Type instructions use this encoding format! The others are also very similar!

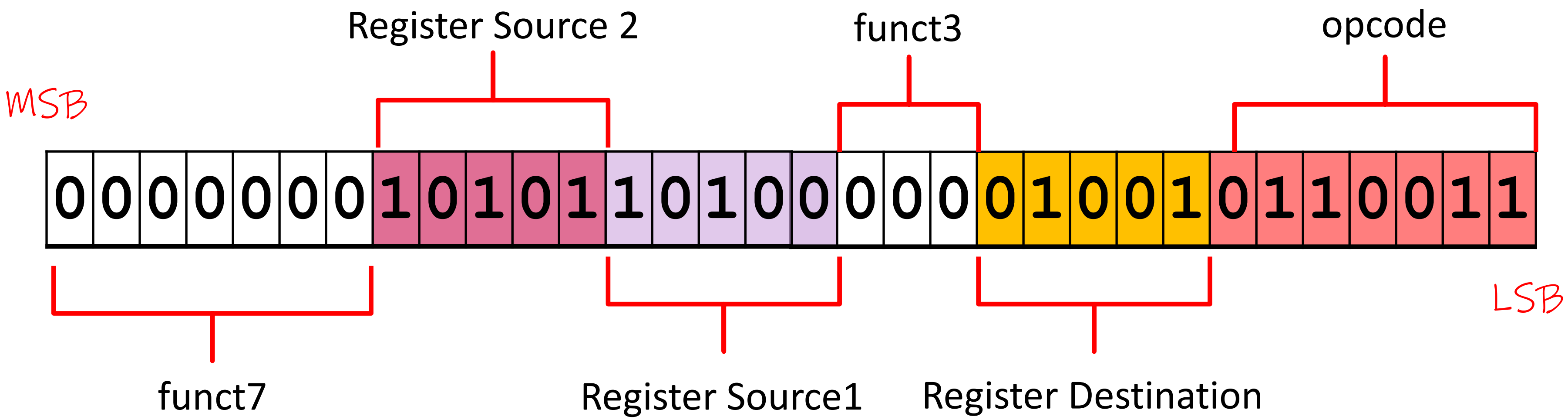
# Encoding Example: Op-codes

- Many instructions are grouped into categories.
  - Arithmetic Instructions, Logical Instructions, Shift instructions…
- This group can be identified by the first 7-bits of the instructions called the **op-code**
  - The op-code denotes the format of the instruction and operation
  - Further information is stored in ***funct3 and funct7***
- Example:
- The op-code is the first seven bits “0110011” which is 51



| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |

# Encoding Example: Op-code, Funct3, Funct7



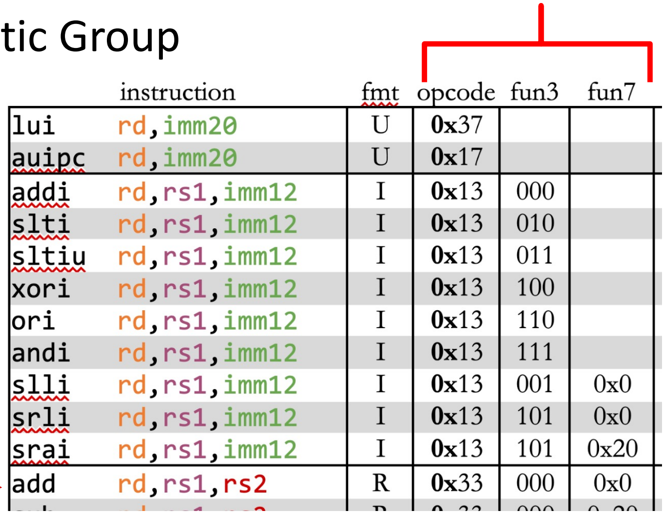
opcode

- 0110011
- Represents Register-Type Instruction and Arithmetic Group

funct3 and funct7

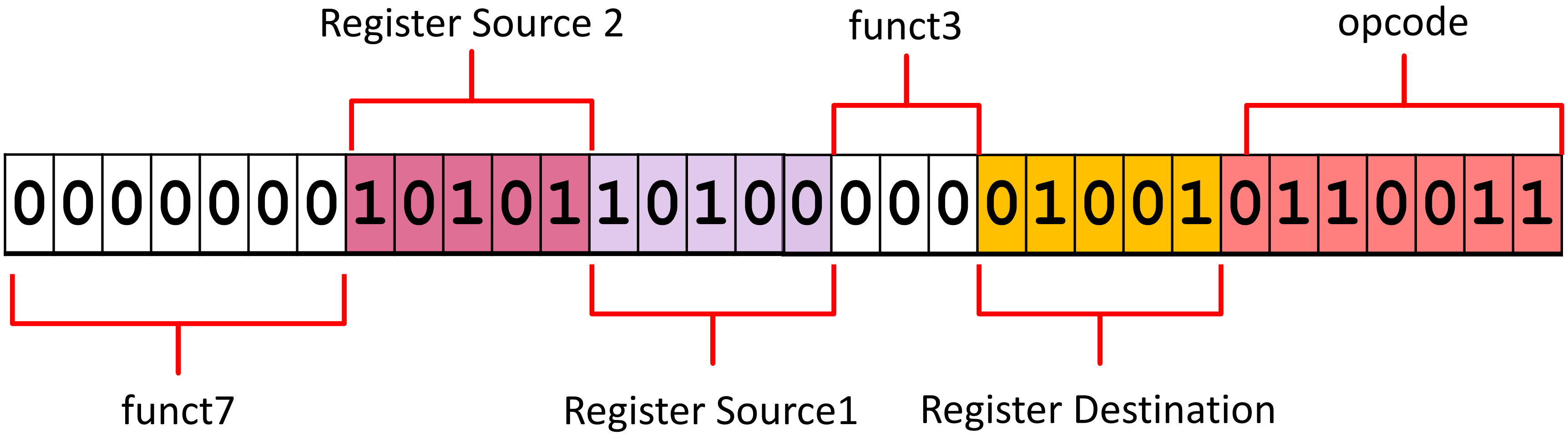
- all zeros

this is the *add instruction*



| instruction | | fmt | opcode | fun3 | fun7 |
|---|---|---|---|---|---|
| lui | rd,imm20 | U | 0x37 | | |
| auipc | rd,imm20 | U | 0x17 | | |
| addi | rd,rs1,imm12 | I | 0x13 | 000 | |
| slti | rd,rs1,imm12 | I | 0x13 | 010 | |
| sltiu | rd,rs1,imm12 | I | 0x13 | 011 | |
| xori | rd,rs1,imm12 | I | 0x13 | 100 | |
| ori | rd,rs1,imm12 | I | 0x13 | 110 | |
| andi | rd,rs1,imm12 | I | 0x13 | 111 | |
| slli | rd,rs1,imm12 | I | 0x13 | 001 | 0x0 |
| srli | rd,rs1,imm12 | I | 0x13 | 101 | 0x0 |
| srai | rd,rs1,imm12 | I | 0x13 | 101 | 0x20 |
| add | rd,rs1,rs2 | R | 0x33 | 000 | 0x0 |

# Encoding Example: Op-code, Funct3, Funct7

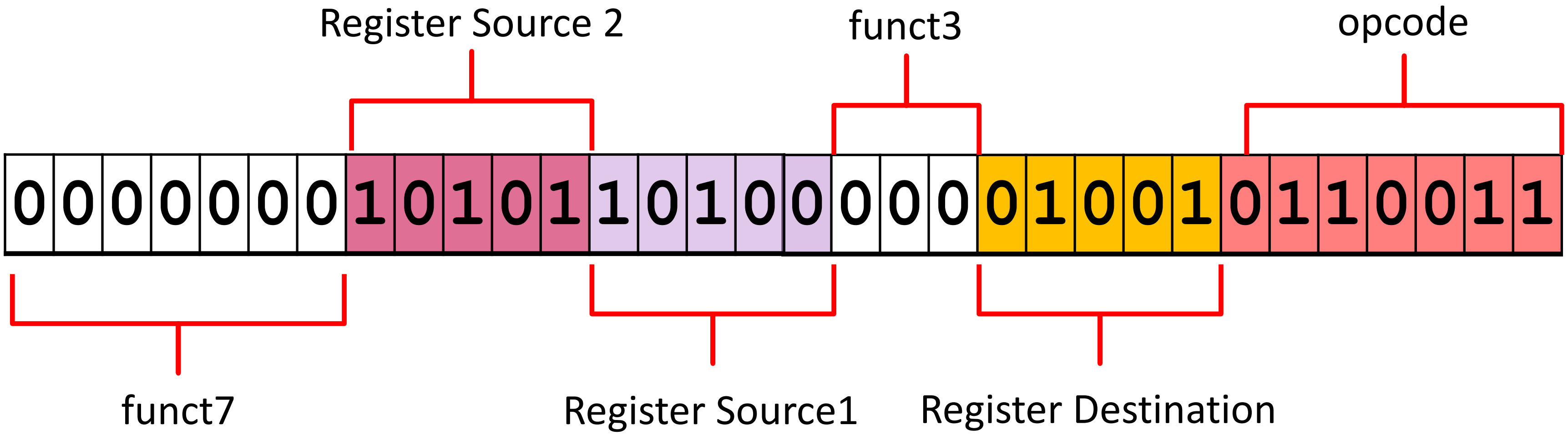


Register Sources and Destination

- Each are 5 bits
- We only have 32 registers, so we only need 5 bits
- 0b00000 - 0b11111 is 32 possible values
- Yes, interpreted as unsigned. No need for ‘negative’ registers…

# Poll Everywhere

**pollev.com/cis2400**



Register Source 2

funct3

opcode

0000000 10101 10100 000 01001 0110011

funct7

Register Source1

Register Destination

**Which registers are rs1 and rs2 respectively?**

A) x19, x20

B) x0, x10

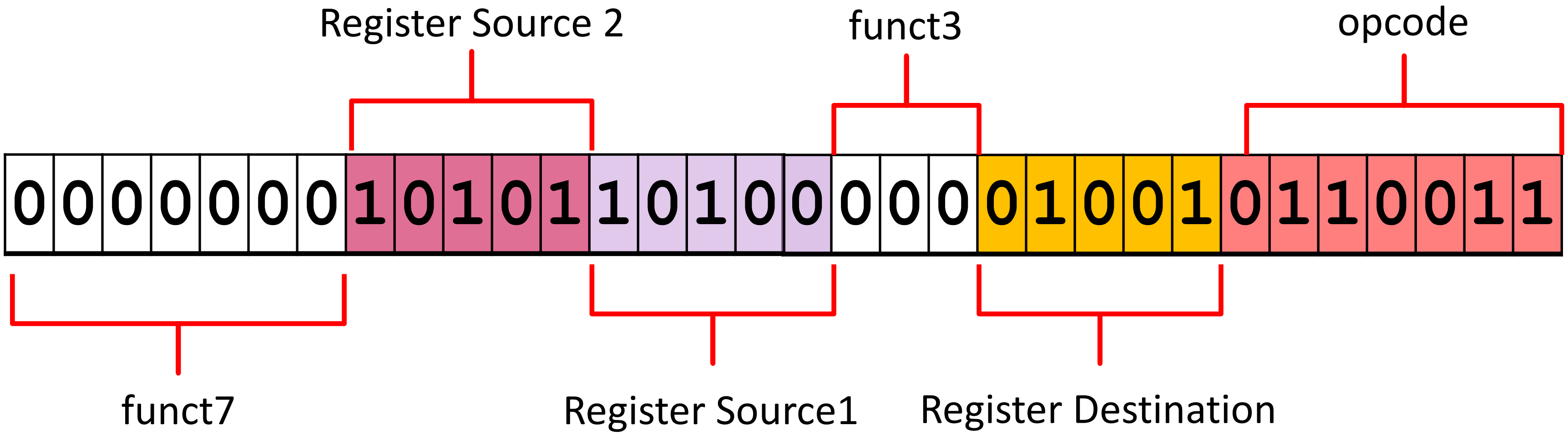
C) x20, x21

D) x21, x20

E) I don’t know

## Poll Everywhere

**pollev.com/cis2400**



**Which registers are rs1 and rs2 respectively?**

A) x19, x20

B) x0, x10

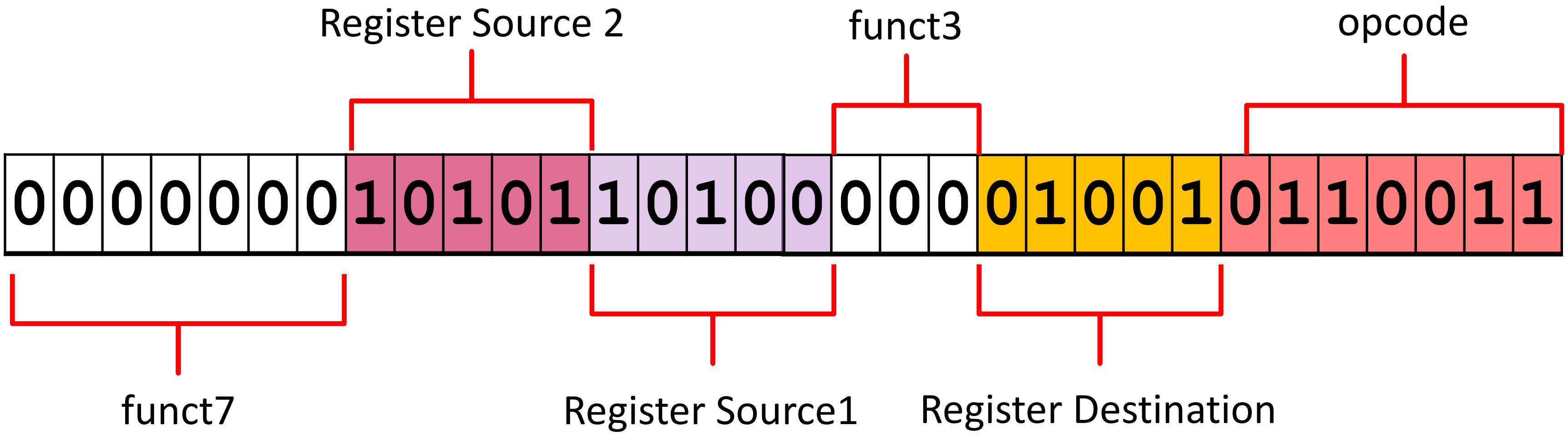
C) x20, x21

D) x21, x20

E) I don't know

```
0b10100 is 16 + 4 = 20
0b10101 is 16 + 4 + 1 = 21
```

# Encoding Example: Op-code, Funct3, Funct7



Register Source 2

funct3

opcode

0000000 10101 10100 000 01001 0110011

funct7

Register Source1

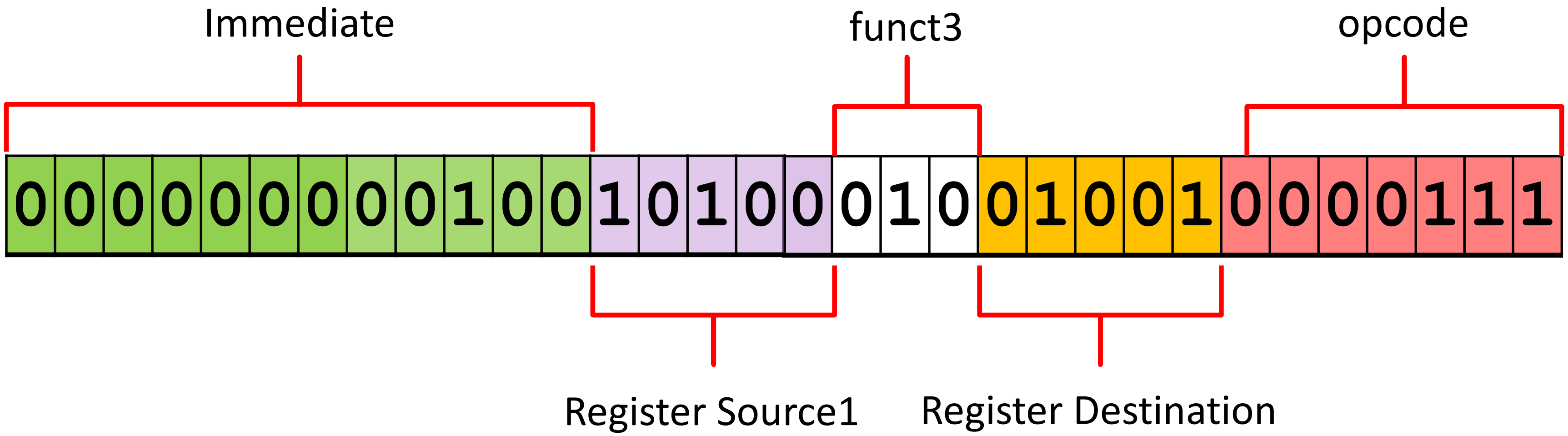
Register Destination

So, this machine code represents:

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

RISC-V

# Encoding Example: Immediate-Type (I-Type)



Immediate funct3 opcode

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

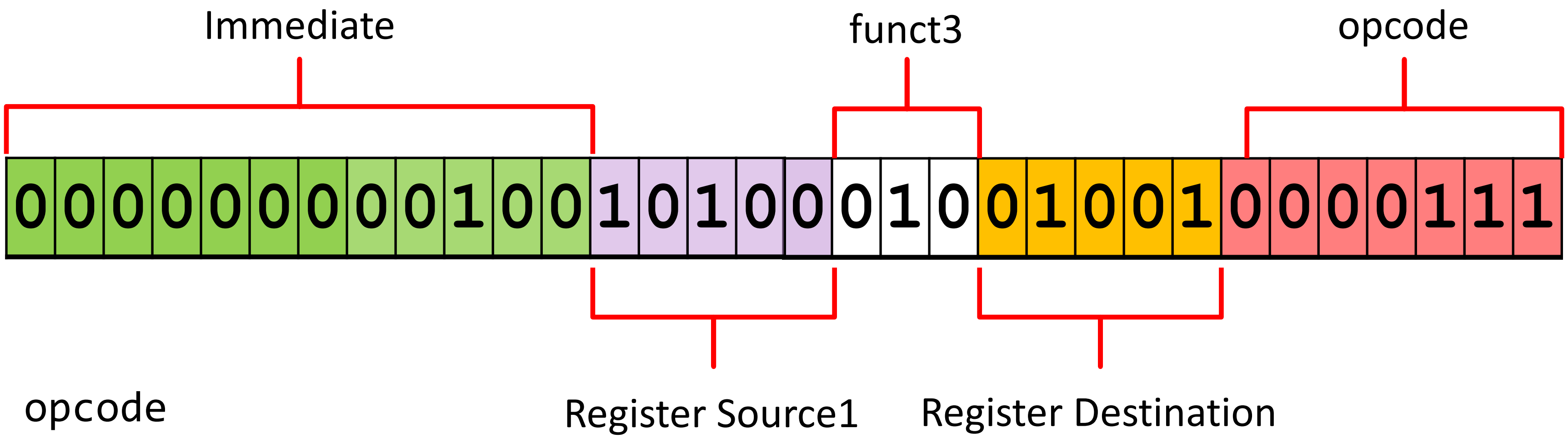
Register Source1 Register Destination

So, this machine code represents:

```
lw   x9, 4(x20)
```

RISC-V

# Decoding Example: Immediate-Type (I-Type)



opcode
- `0x03`
- Represents I-Type Instruction and Load Group

`Funct3`
- `0b010` – word size load

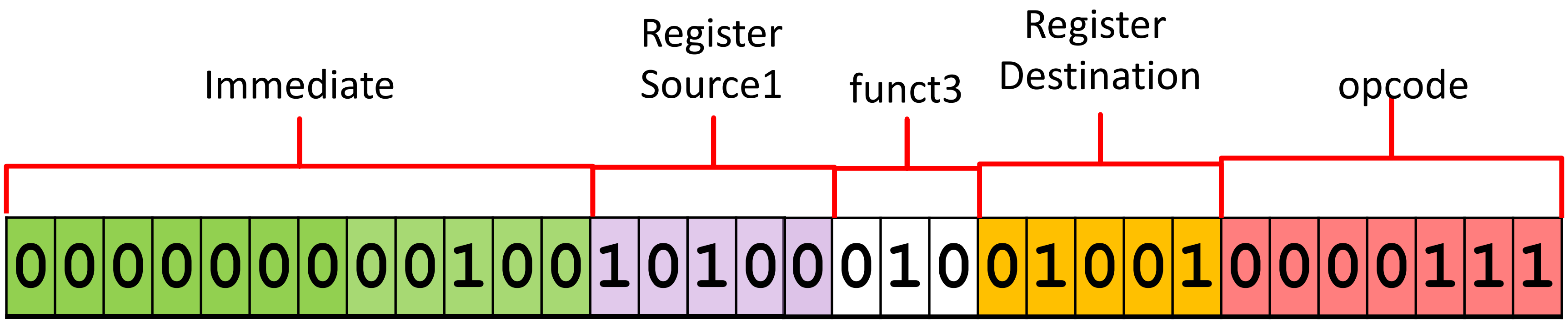
Destination
- `0b01001 (x9)`

`Register Source 1`
- `0b10100 (x20)`

immediate
- Is the number 4

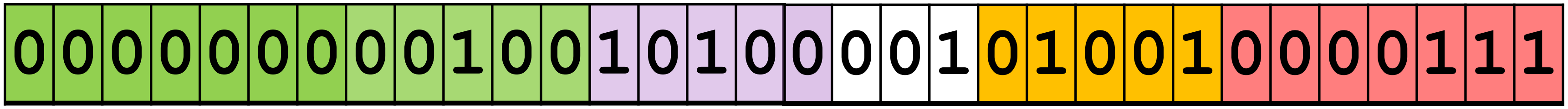
`lw x9, 4(x20)` RISC-V

# Decoding Example: Immediate-Type (I-Type)



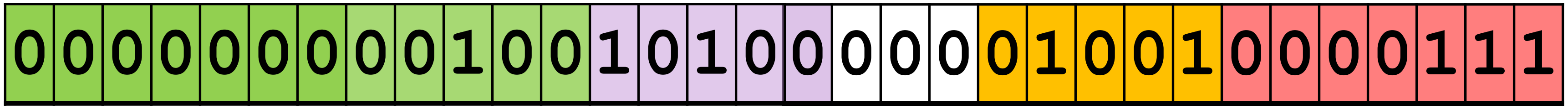
`lw x9, 4(x20)` RISC-V

Load word



`lh x9, 4(x20)` RISC-V

Load Half Word



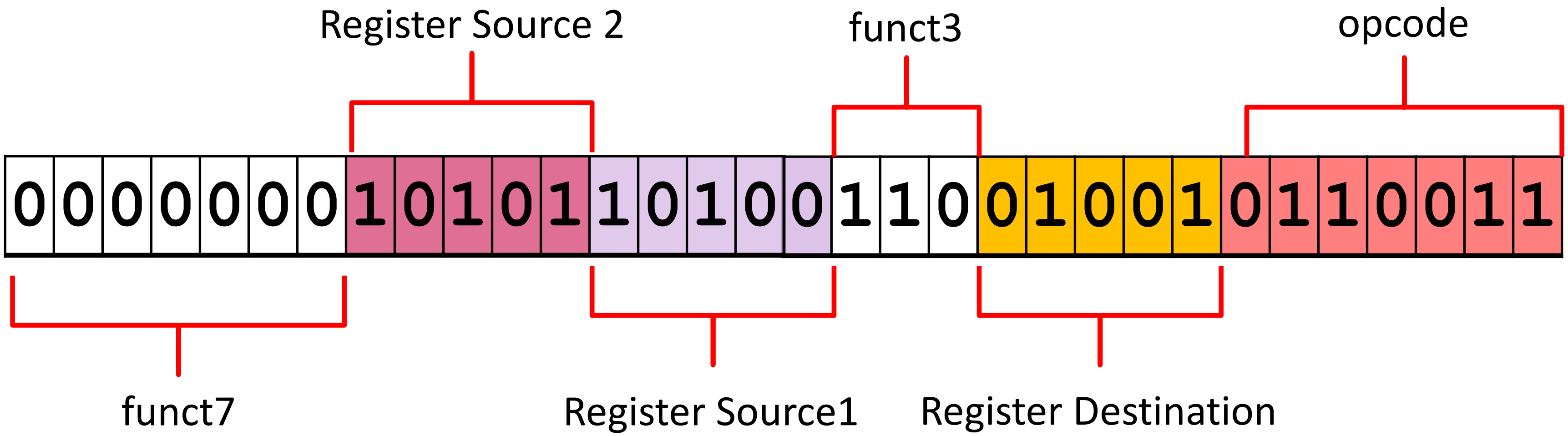
`lb x9, 4(x20)` RISC-V

Load Byte

# Poll Everywhere

**pollev.com/cis2400**

- What instruction does this 32-bit value represent?



| | | | | |
|---|---|---|---|---|---|
| 0000000 | 10101 | 10100 | 110 | 01001 | 0110011 |

**A)** **add**

**B)** **or**

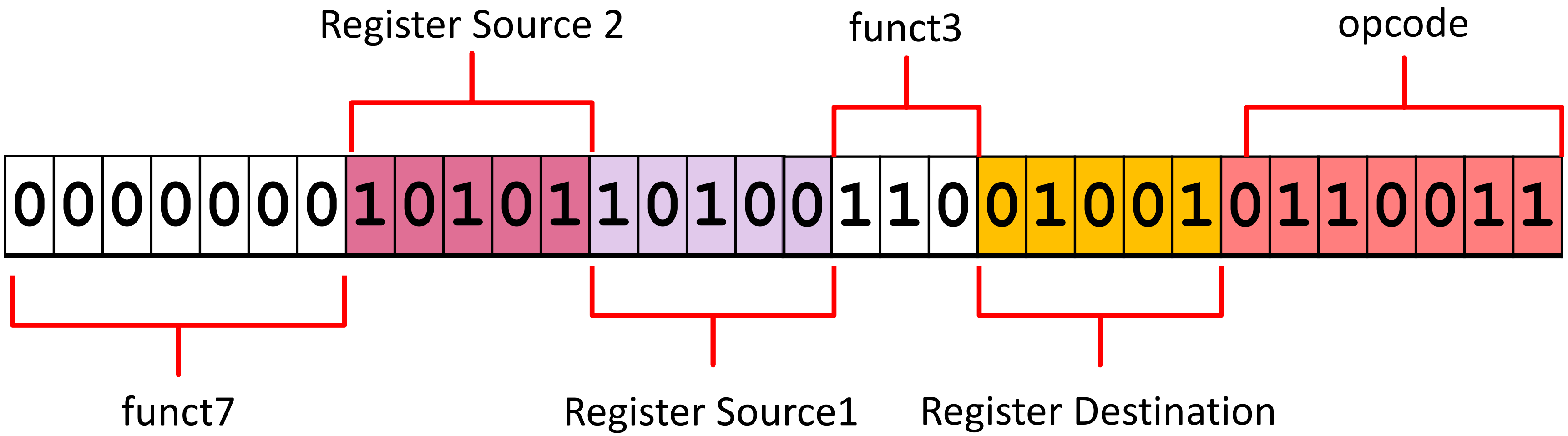
**C)** **ori**

**D)** **sub**

# Poll Everywhere

**pollev.com/cis2400**

- What instruction does this 32-bit value represent?



| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |

A) **add**

[✓ B) **or**]

C) **ori**

D) **sub**

if we just need the instruction, check opcode, funct3, and funct7

We already know that there's no Immediate here! So C) is already wrong

Op Code 0x33 -> R-Type and Arithmetic

Funct3 being 110 tells us it's *or*!

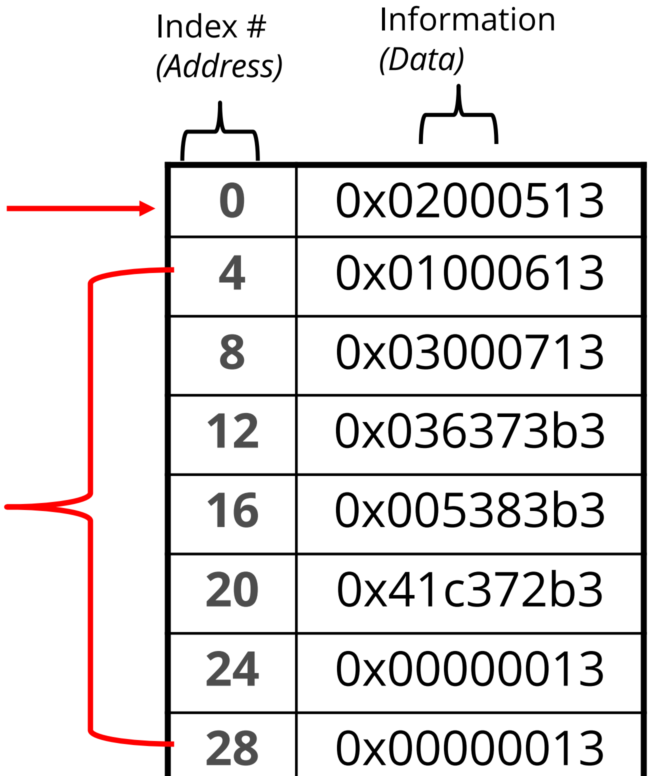
# Lecture Outline

- What is an ISA
  - Why RISC-V?
  - X86 & ARM
- RISC-V
  - R- and I-Type Instructions
  - Memory Operations
- Machine Code
  - RISC-V Encoding
  - In Memory
- The Program Counter
  - AUIPC
  - JALR

# Code in Memory

- An instruction fits in 1 word (32 bits)
- These instructions are stored in memory and accessed sequentially
    - When we trace through the code, we are just accessing the next location in memory

```
addi t0, x0, 32
addi t1, x0, 16
addi t2, x0, 64

div t3, t2, t1
add t3, t3, t0
sub t0, t2, t3
```

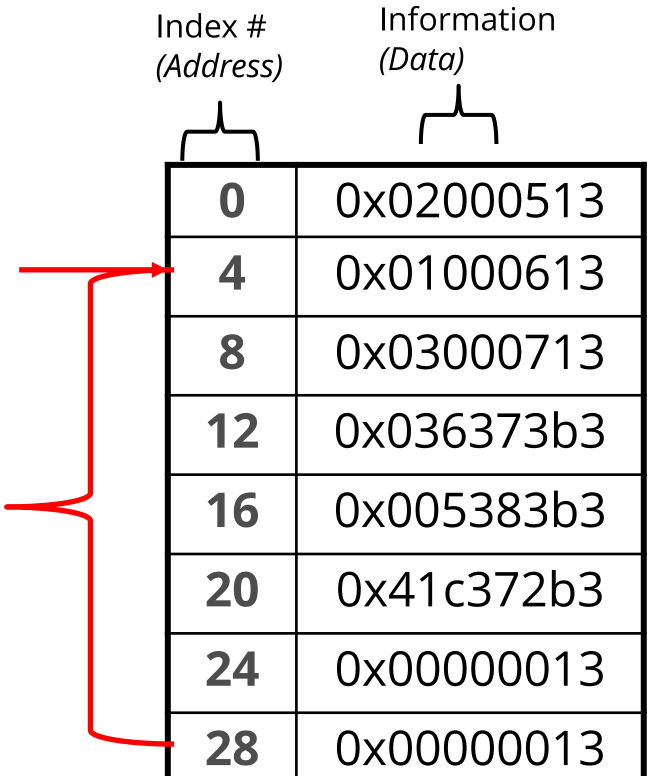


| Index # *(Address)* | Information *(Data)* |
|---|---|
| **0** | 0x02000513 |
| **4** | 0x01000613 |
| **8** | 0x03000713 |
| **12** | 0x036373b3 |
| **16** | 0x005383b3 |
| **20** | 0x41c372b3 |
| **24** | 0x00000013 |
| **28** | 0x00000013 |

# Code in Memory

- An instruction fits in 1 word (32 bits)
- These instructions are stored in memory and accessed sequentially
    - When we trace through the code, we are just accessing the next location in memory

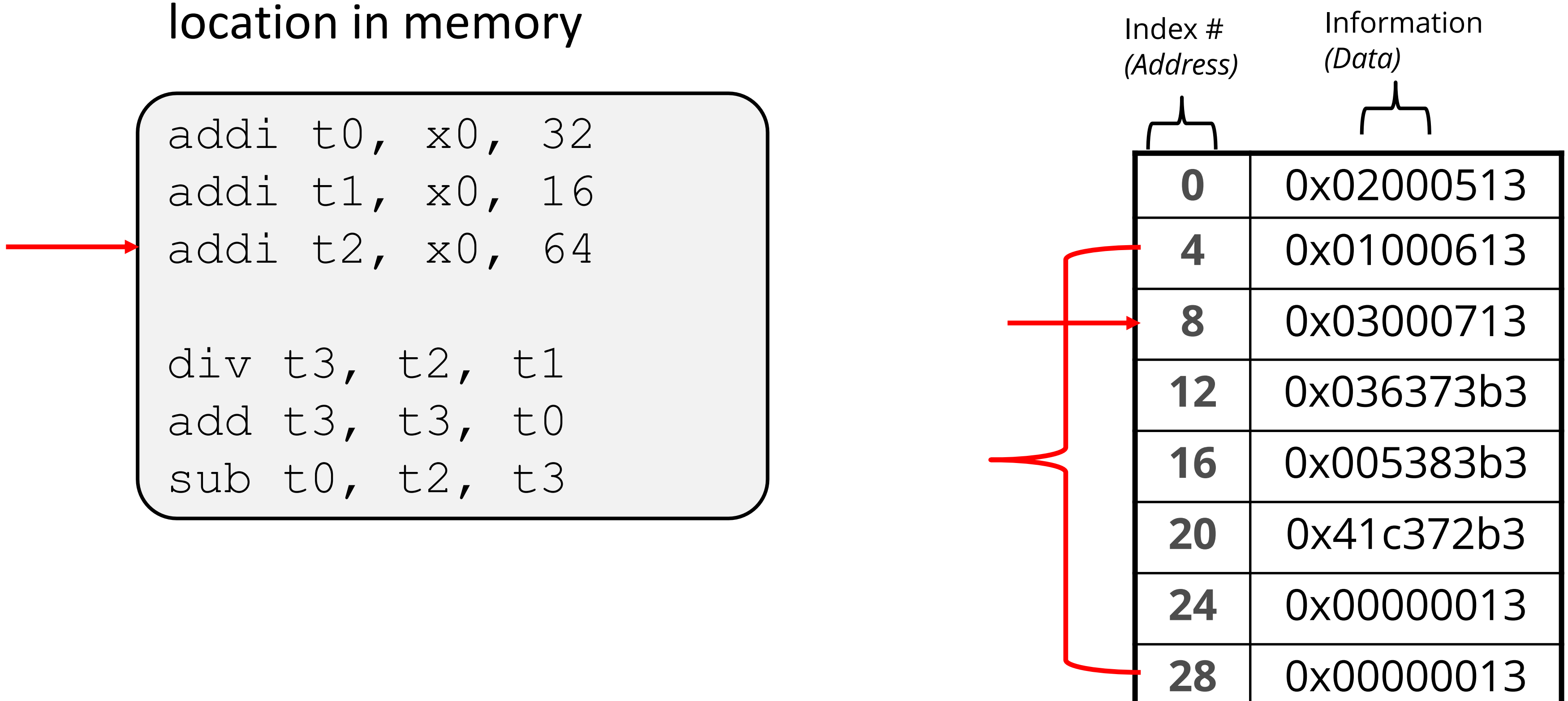
```
addi t0, x0, 32
addi t1, x0, 16
addi t2, x0, 64

div t3, t2, t1
add t3, t3, t0
sub t0, t2, t3
```



| Index # *(Address)* | Information *(Data)* |
|---|---|
| **0** | 0x02000513 |
| **4** | 0x01000613 |
| **8** | 0x03000713 |
| **12** | 0x036373b3 |
| **16** | 0x005383b3 |
| **20** | 0x41c372b3 |
| **24** | 0x00000013 |
| **28** | 0x00000013 |

# Code in Memory

- An instruction fits in 1 word (32 bits)
- These instructions are stored in memory and accessed sequentially
  - When we trace through the code, we are just accessing the next location in memory



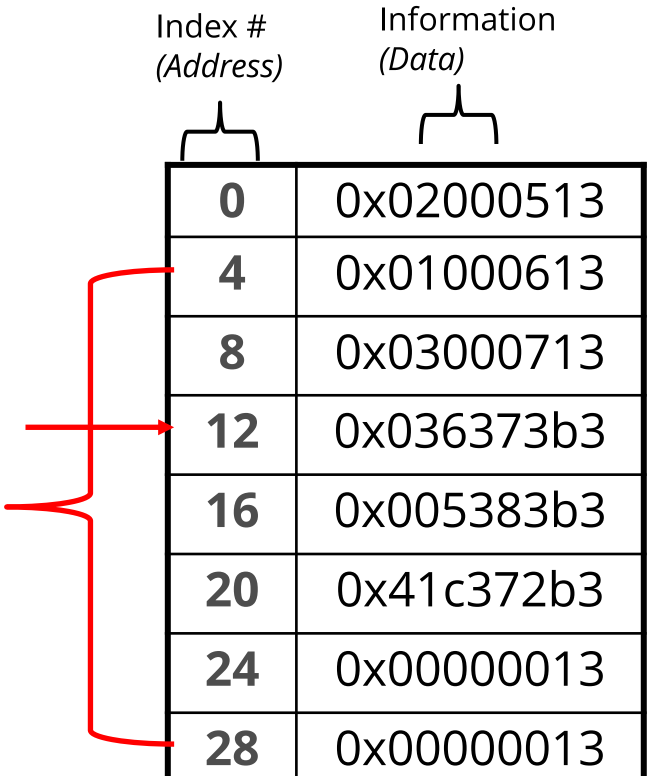
```
addi t0, x0, 32
addi t1, x0, 16
addi t2, x0, 64

div t3, t2, t1
add t3, t3, t0
sub t0, t2, t3
```

| Index # *(Address)* | Information *(Data)* |
|---|---|
| **0** | 0x02000513 |
| **4** | 0x01000613 |
| **8** | 0x03000713 |
| **12** | 0x036373b3 |
| **16** | 0x005383b3 |
| **20** | 0x41c372b3 |
| **24** | 0x00000013 |
| **28** | 0x00000013 |

# Code in Memory

- An instruction fits in 1 word (32 bits)
- These instructions are stored in memory and accessed sequentially
  - When we trace through the code, we are just accessing the next location in memory

```
addi t0, x0, 32
addi t1, x0, 16
addi t2, x0, 64

div t3, t2, t1
add t3, t3, t0
sub t0, t2, t3
```



| Index # *(Address)* | Information *(Data)* |
|---|---|
| **0** | 0x02000513 |
| **4** | 0x01000613 |
| **8** | 0x03000713 |
| **12** | 0x036373b3 |
| **16** | 0x005383b3 |
| **20** | 0x41c372b3 |
| **24** | 0x00000013 |
| **28** | 0x00000013 |

# Lecture Outline

- What is an ISA
  - Why RISC-V?
  - X86 & ARM
- RISC-V
  - R- and I-Type Instructions
  - Memory Operations
- Machine Code
  - RISC-V Encoding
  - In Memory
- The Program Counter
  - AUIPC
  - JALR

# Program Counter

- The **Program Counter** (PC) is a special register that keeps track of the address of the current instruction executing

- Implicitly, every instruction we have covered so far also increments the PC
  - ADD doesn’t just perform addition, but also moves on to the next instruction to execute (the instruction after it) implicitlty
  - Here, you can imagine we’re doing PC + 4, as each instruction is 4 bytes, so the next instruction is 4 bytes away.

# Accessing the Program Counter

`auipc rd, imm20`

- rd
    - register that contains the result
- imm20:
    - 20 bit immediate

`rd = pc + (imm20 << 12)`

```
aupic t0, 0 //t0 will hold pc + 0
```

RISC-V

| reg | alias |
|---|---|
| **x5-x7** | **t0-t2** |

| 31 12 | 11 7 | 6 0 |
|---|---|---|
| imm[31:12] | rd | opcode |
| 20 | 5 | 7 |

***There are 6 type of instruction formats, this instruction uses the U-type***

| reg | alias |
|---|---|
| `x5-x7` | `t0-t2` |

# Changing the Program Counter

## `jalr rd, imm12(rs1)`

- `“Jump and Link Register”`
- `rd`
  - register that contains pc + 4 (i.e. the instruction after ***this one)***
- imm12:
  - 12 bit immediate
- rs1:
  - Base address

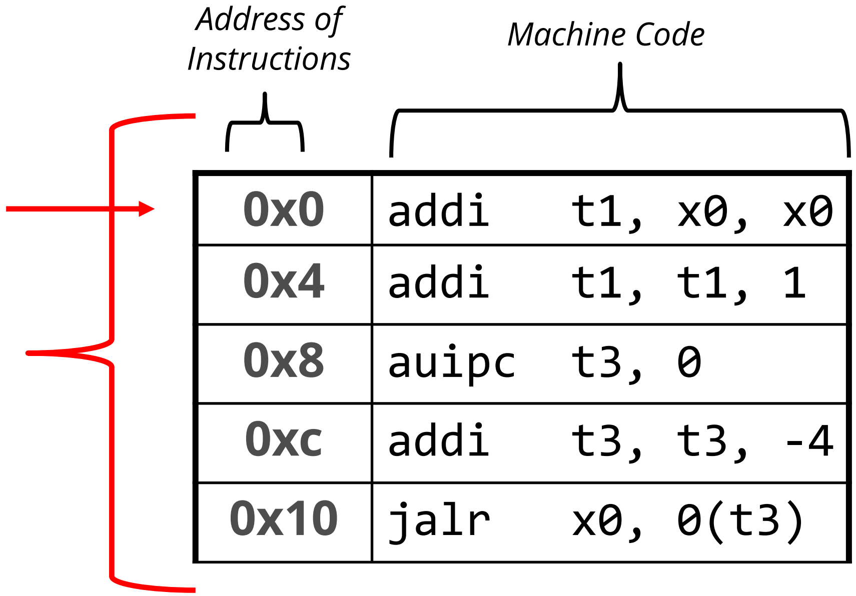
```
//`link` part
rd = pc+4;
//clears the LSB
pc = (rs1+se(imm12)) & ~0x1
```

`jalr x0, 0(t0)` RISC-V

This means the next instruction to be executed will be the one located at the address stored in `t0`.

***There are 6 type of instruction formats, this instruction uses the I-type***

# Our First Infinite Loop RISC-V Program



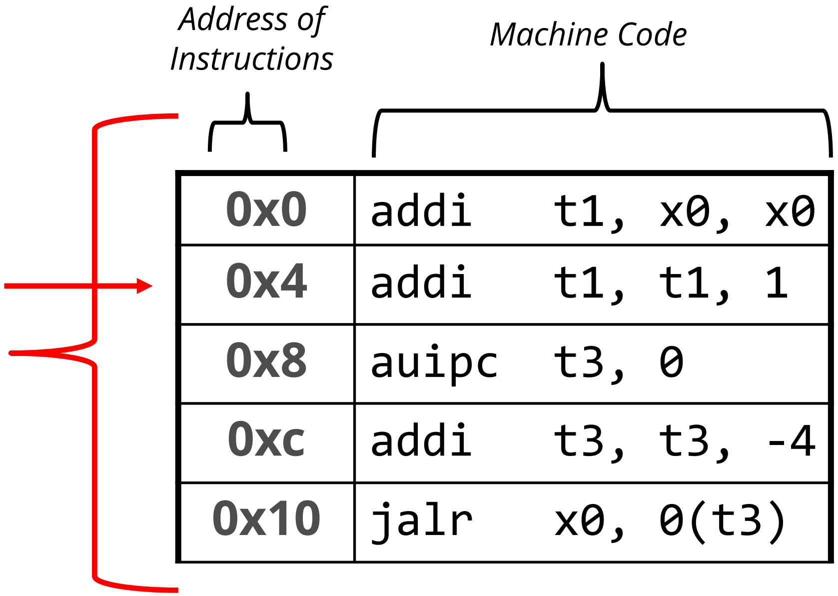
| *Address of Instructions* | *Machine Code* |
|---|---|
| **0x0** | `addi  t1, x0, x0` |
| **0x4** | `addi  t1, t1, 1` |
| **0x8** | `auipc t3, 0` |
| **0xc** | `addi  t3, t3, -4` |
| **0x10** | `jalr  x0, 0(t3)` |

PC `0x00000000`

t1 `????????????`

Disclaimer: the real intricacies of the PC are hidden, purely education example

# Our First Infinite Loop RISC-V Program

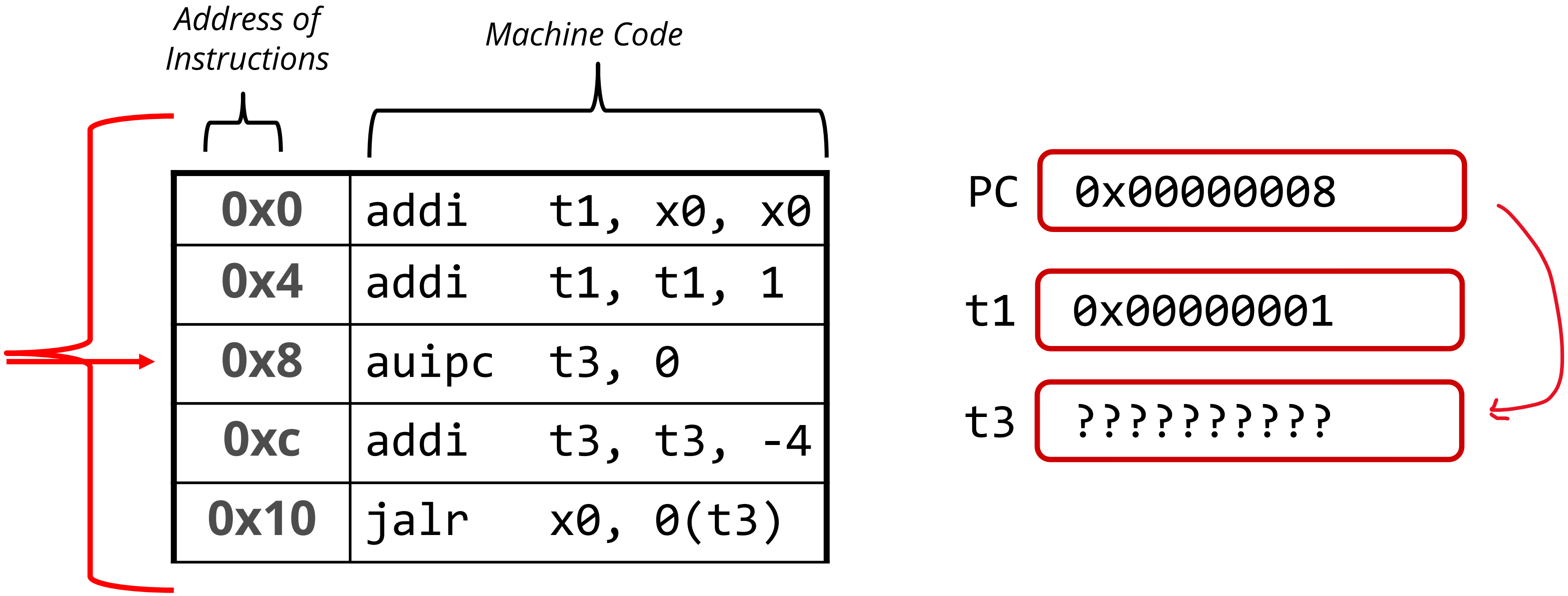


| | |
|---|---|
| **0x0** | `addi t1, x0, x0` |
| **0x4** | `addi t1, t1, 1` |
| **0x8** | `auipc t3, 0` |
| **0xc** | `addi t3, t3, -4` |
| **0x10** | `jalr x0, 0(t3)` |

| | |
|---|---|
| PC | `0x00000004` |
| t1 | `0x00000000` |

Disclaimer: the real intricacies of the PC are hidden, purely education example

# Our First Infinite Loop RISC-V Program



*Address of Instructions*

*Machine Code*

| | |
|---|---|
| **0x0** | `addi   t1, x0, x0` |
| **0x4** | `addi   t1, t1, 1` |
| **0x8** | `auipc  t3, 0` |
| **0xc** | `addi   t3, t3, -4` |
| **0x10** | `jalr   x0, 0(t3)` |

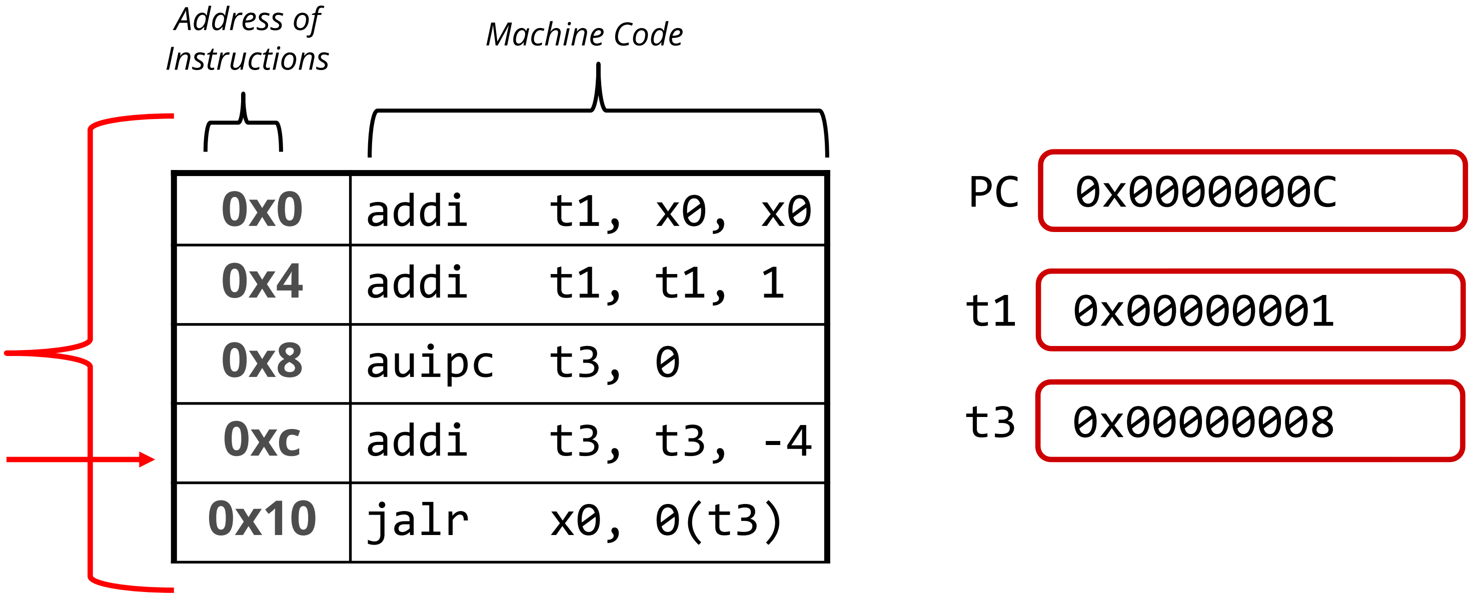
PC `0x00000008`

t1 `0x00000001`

t3 `??????????`

Disclaimer: the real intricacies of the PC are hidden, purely education example

# Our First Infinite Loop RISC-V Program



| | |
|---|---|
| 0x0 | `addi t1, x0, x0` |
| 0x4 | `addi t1, t1, 1` |
| 0x8 | `auipc t3, 0` |
| 0xc | `addi t3, t3, -4` |
| 0x10 | `jalr x0, 0(t3)` |

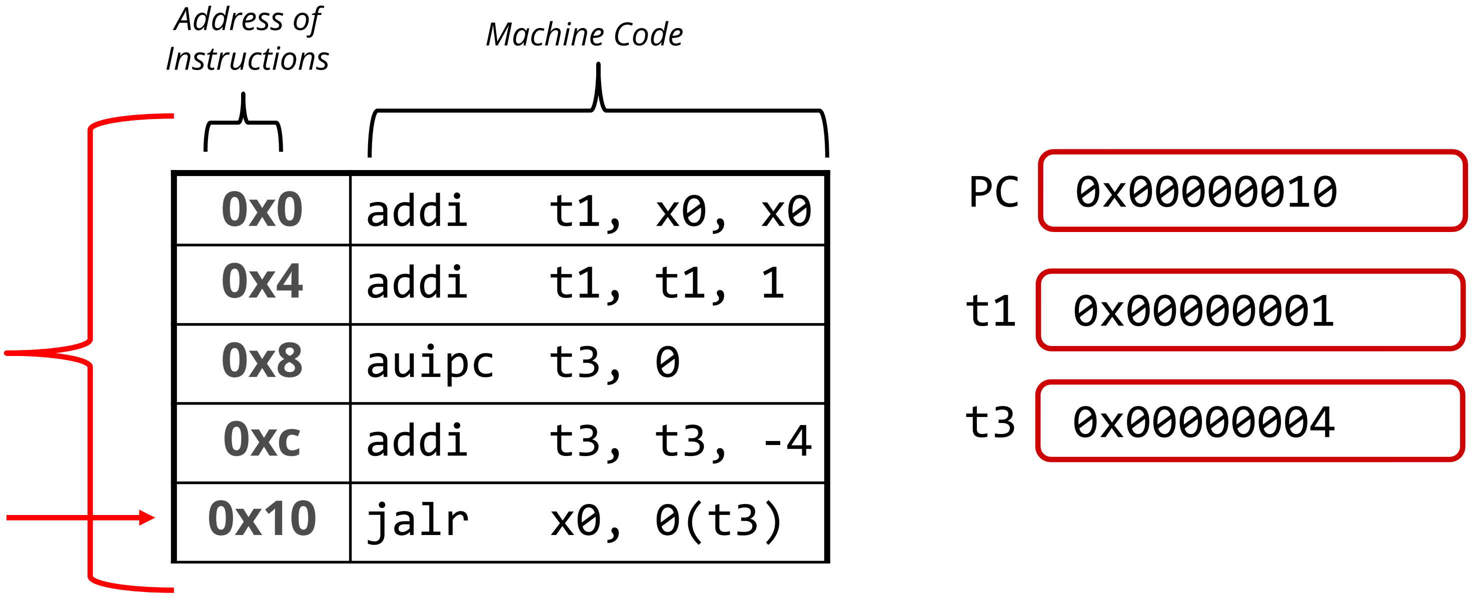
PC `0x0000000C`

t1 `0x00000001`

t3 `0x00000008`

Disclaimer: the real intricacies of the PC are hidden, purely education example

# Our First Infinite Loop RISC-V Program



*Address of Instructions*

*Machine Code*

| | |
|---|---|
| 0x0 | `addi  t1, x0, x0` |
| 0x4 | `addi  t1, t1, 1` |
| 0x8 | `auipc t3, 0` |
| 0xc | `addi  t3, t3, -4` |
| 0x10 | `jalr  x0, 0(t3)` |

| | |
|---|---|
| PC | `0x00000010` |
| t1 | `0x00000001` |
| t3 | `0x00000004` |

This will try to set register x0 to the value 0x00000014
(because that's where the 'next' instruction is after 0x10)

**x0 never changes, writing to it does nothing (because it's always zero)**

**Then, PC is set to 0x4, and we go back up**

Disclaimer: the real intricacies of the PC are hidden, purely education example

# Our First Infinite Loop RISC-V Program

| *Address of Instructions* | *Machine Code* |
|---|---|
| 0x0 | `addi t1, x0, x0` |
| 0x4 | `addi t1, t1, 1` |
| 0x8 | `auipc t3, 0` |
| 0xc | `addi t3, t3, -4` |
| 0x10 | `jalr x0, 0(t3)` |

| | |
|---|---|
| PC | `0x00000004` |
| t1 | `0x00000001` |
| t3 | `0x00000004` |

This loops forever incrementing t1 by 1

……..one must imagine Sisyphus

Disclaimer: the real intricacies of the PC are hidden, purely education example

# Next Lecture

- More RISC-V!