RISC-V Instruction Overview II

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

pollev.com/cis2400

- How are you? Any Questions?

Logistics

- None I think
- Yes there will be a concept check this week

Lecture Outline

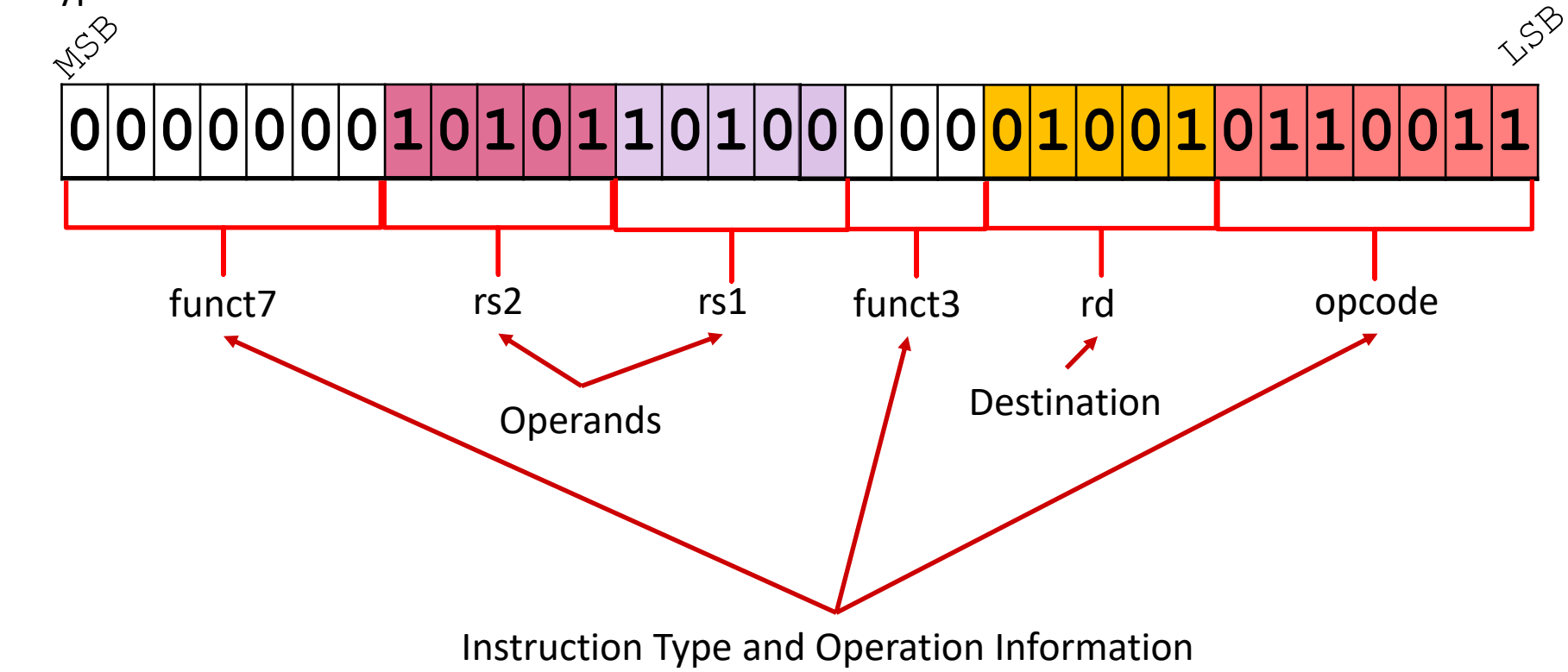
- Machine Code
  - RISC-V Encoding
  - In Memory
- The Program Counter
  - AUIPC
  - JALR
- Conditional Statements
  - Branching
- C in RISC-V
  - Caller/Callee Owned Registers
  - Stack Pointer
  - Full Set Up

RISC-V: 6-Types

- Last Lecture
  - Register-Type (R-Type)
    - Operands: Registers
    - Destination: Register
    - `e.g. add, mul, div, or, and`
  - Immediate-Type (I-Type)
    - Operands: Register, Immediate (12 bits)
    - Destination: Register
    - `e.g. addi, lw, jalr, andi, ori`

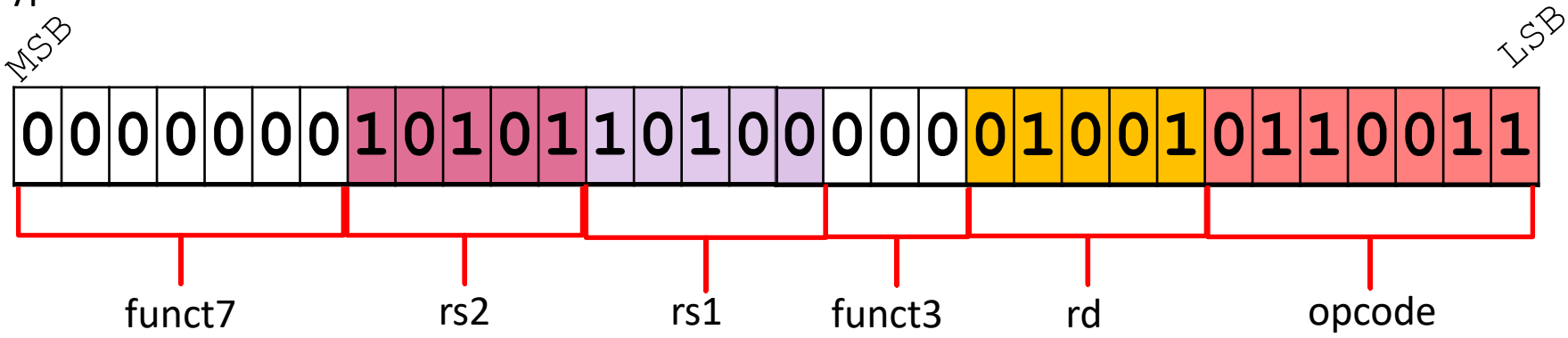
Side-by-Side

R-Type

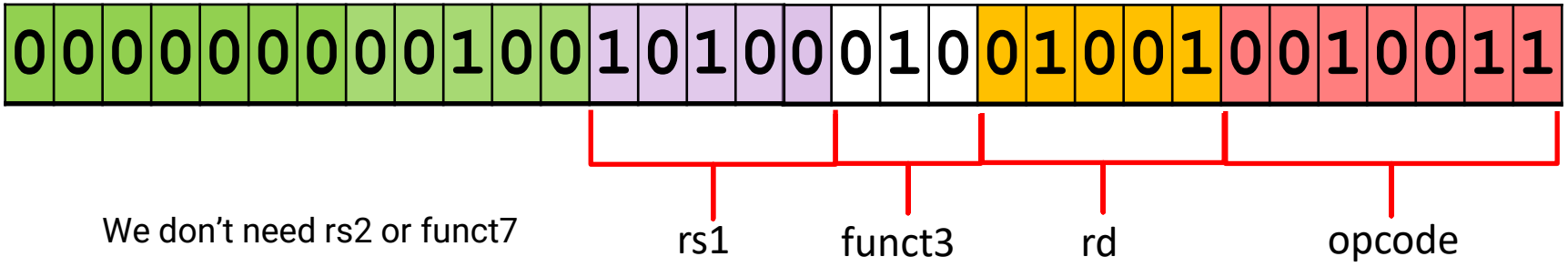


Side-by-Side

R-Type



I-Type

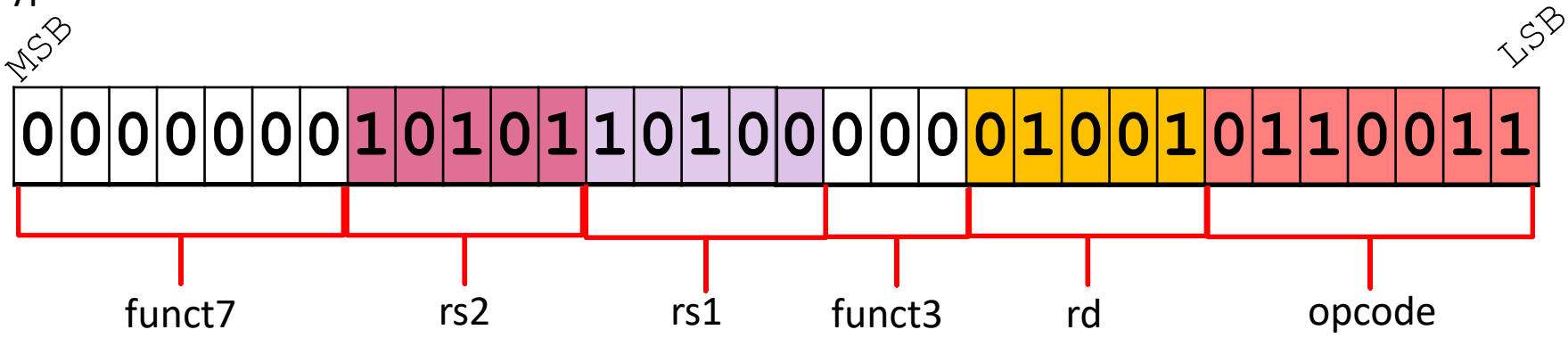


Can use remaining bits for the Immediate!

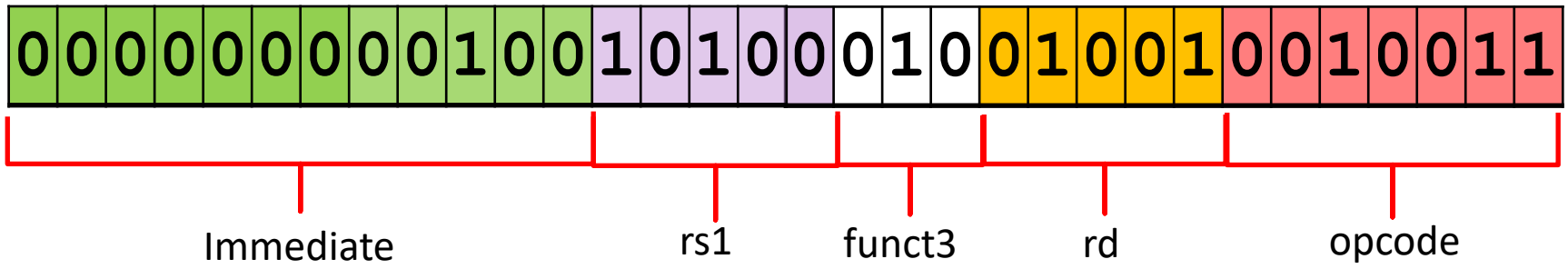
imm12 == 12 bits for immediate

Side-by-Side

R-Type



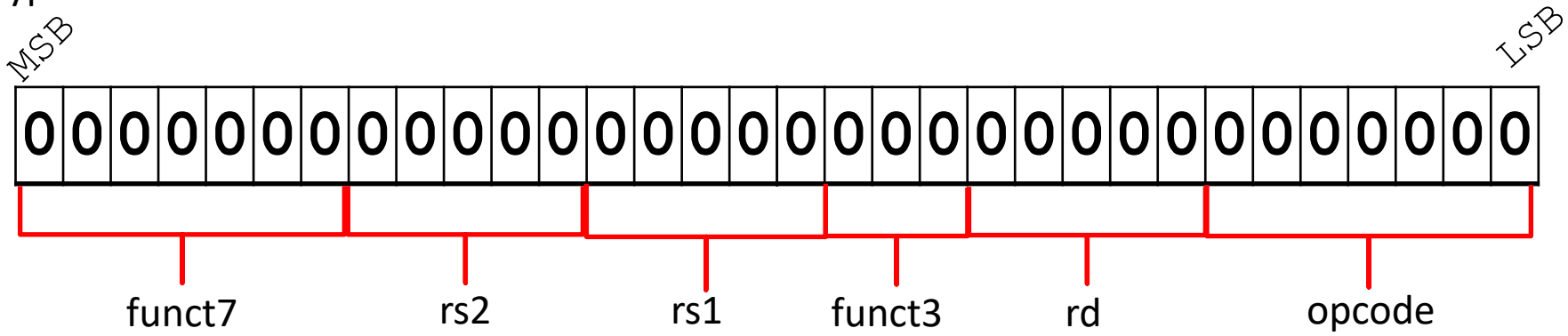
I-Type



RISC Principles: Keep arguments where they are,
use the remaining space meaningfully to simplify the whole system

Machine Code: The Store Type

S-Type

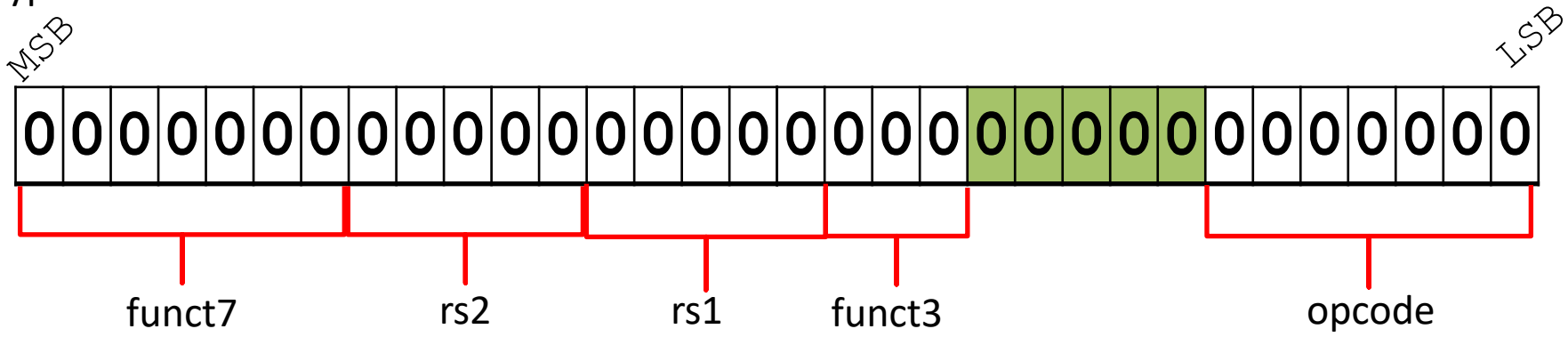


When `storing`…no need for destination register to save the result of the operation

Remove rd…free's up 5 bits

Machine Code: The Store Type

S-Type



When `storing`…no need for destination register to save the result of the operation

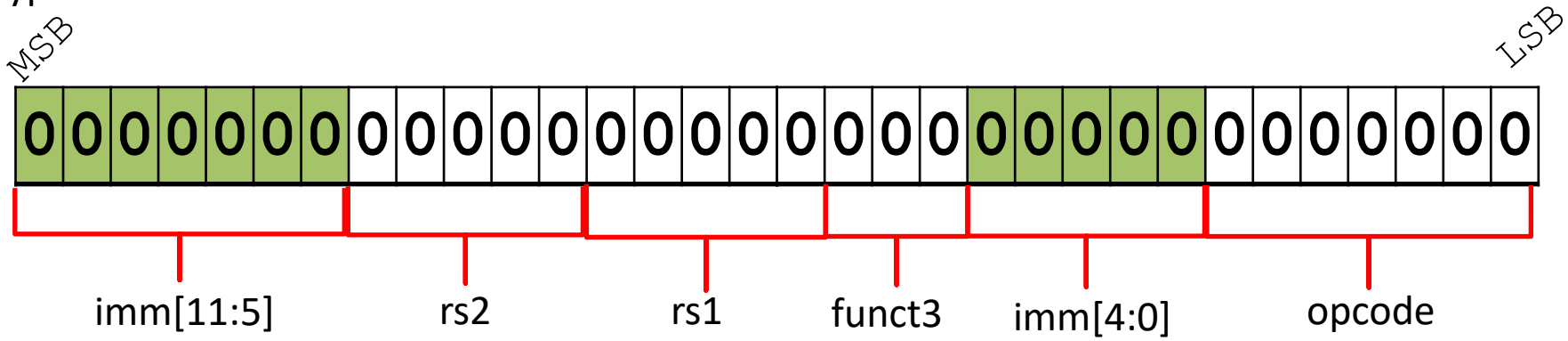
Remove rd…free's up 5 bits

Opcode and funct3: enough bits to encode the instruction and any specific stores (`w`,`h`,`b`)

So, let's remove funct7…free's up 7 bits…

Machine Code: The Store Type

S-Type



When `storing`…no need for destination register to save the result of the operation

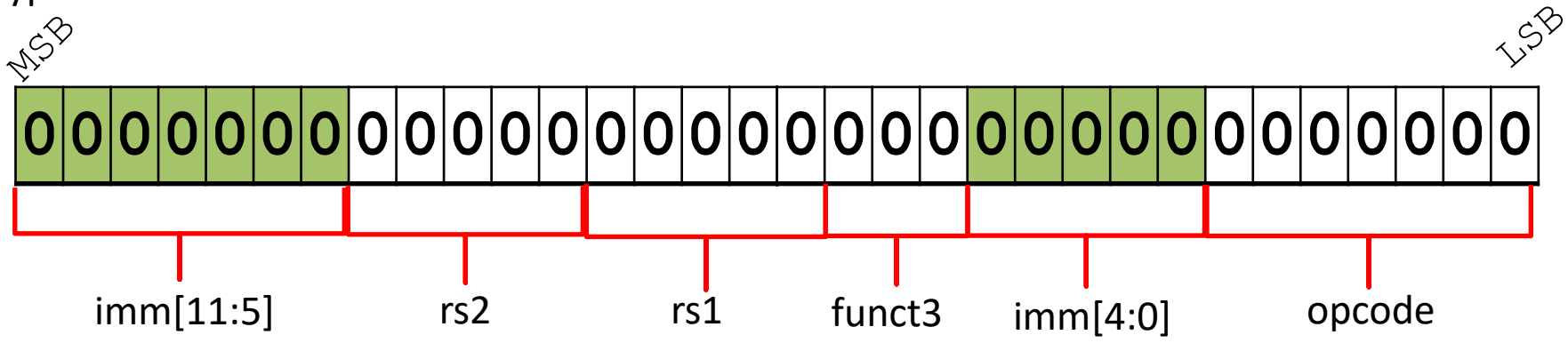
Remove rd…free's up 5 bits

Opcode and funct3: enough bits to encode the instruction and any specific stores (`w`, `h`, `b`)

So, let's remove funct7…free's up 7 bits…

Machine Code: The Store Type

S-Type



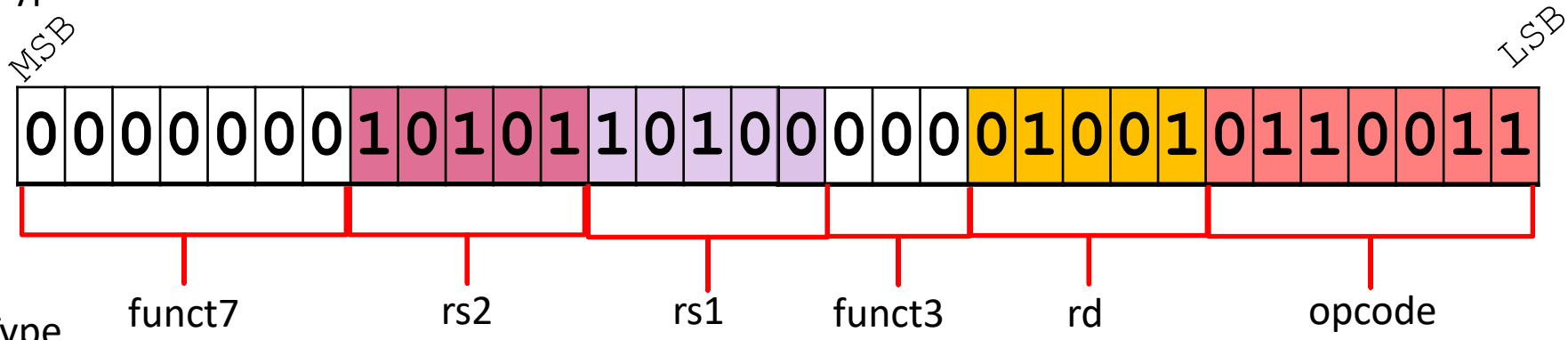

```
sw rs2, imm12(rs1) //store word
```

The 12 bit immediate is split between two locations, where rd and funct7 were….

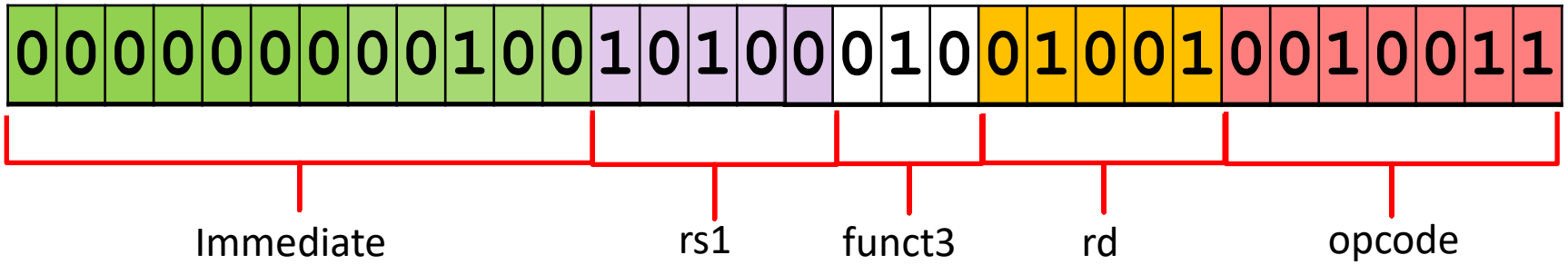
This is what RISC-V is all about.

All Types

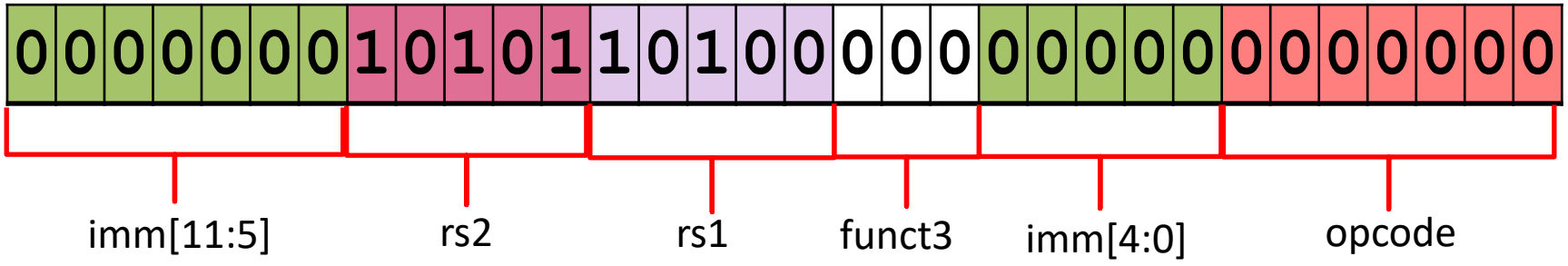
R-Type



I-Type

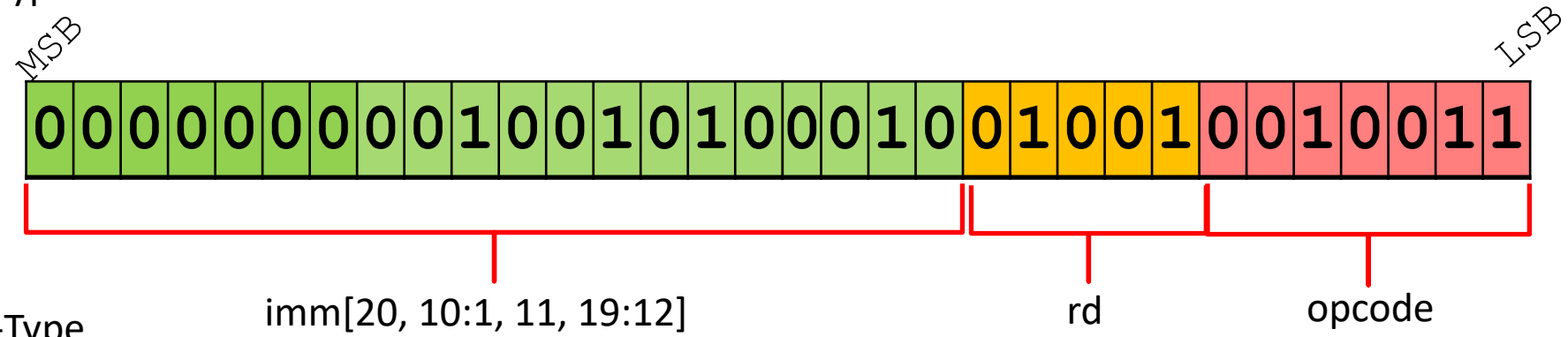


S-Type

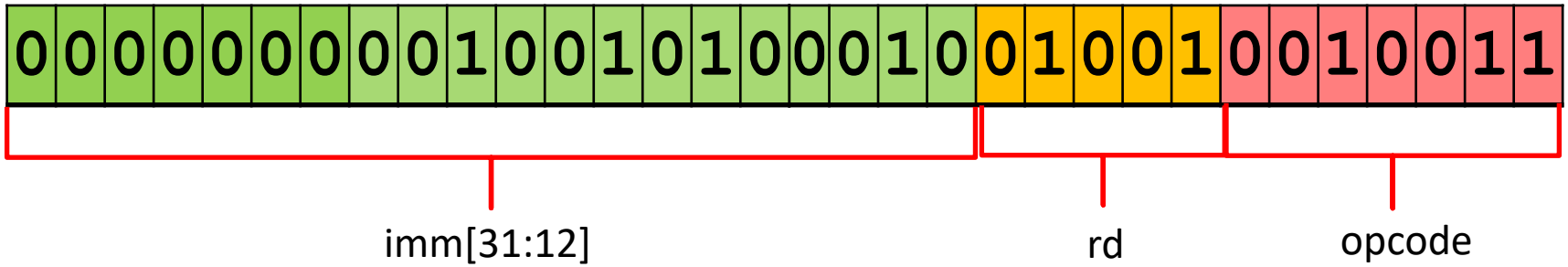


All Types

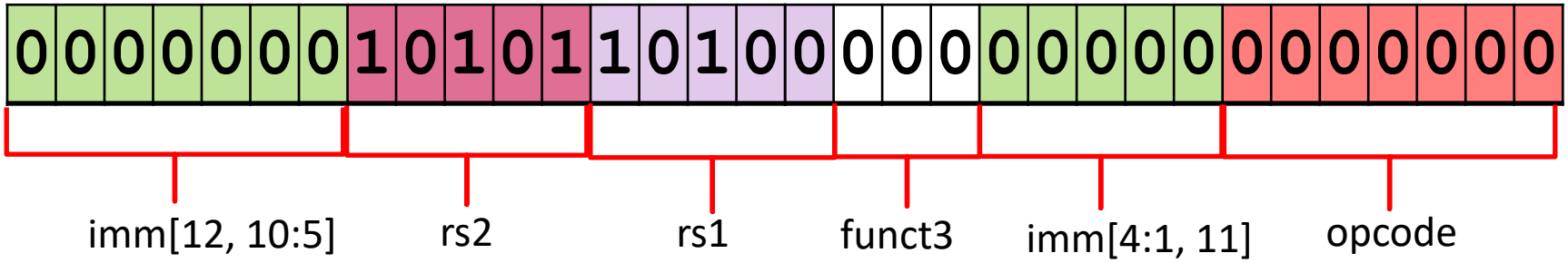
J-Type



U-Type



B-Type



Lecture Outline

- Machine Code
  - RISC-V Encoding
  - In Memory
- The Program Counter
  - AUIPC
  - JALR
- Conditional Statements
  - Branching
- C in RISC-V
  - Caller/Callee Owned Registers
  - Stack Pointer
  - Full Set Up

Code in Memory

- Instructions are 32 bits
- Instructions are stored in memory and accessed sequentially
  - To know what to execute next, we just access the next instruction!

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

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

Just like C, RISC-V is byte addressable

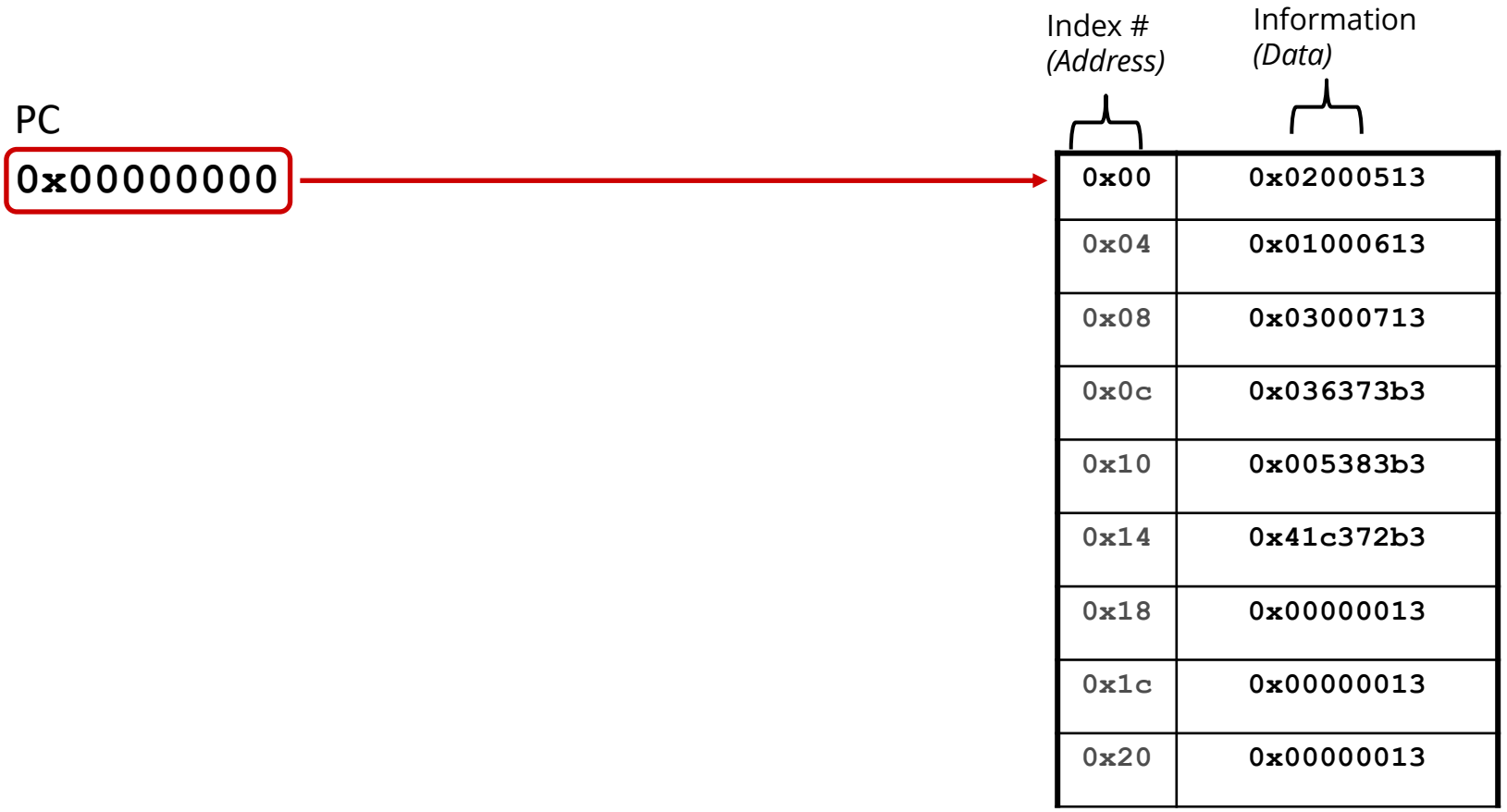
So, the next instruction is '4' bytes away.

Current Address + 4;

Index # *(Address)*	Information *(Data)*
0	0x02000513
4	0x01000613
8	0x03000713
12	0x036373b3
16	0x005383b3
20	0x41c372b3
24	0x00000013
28	0x00000013
32	0x00000013

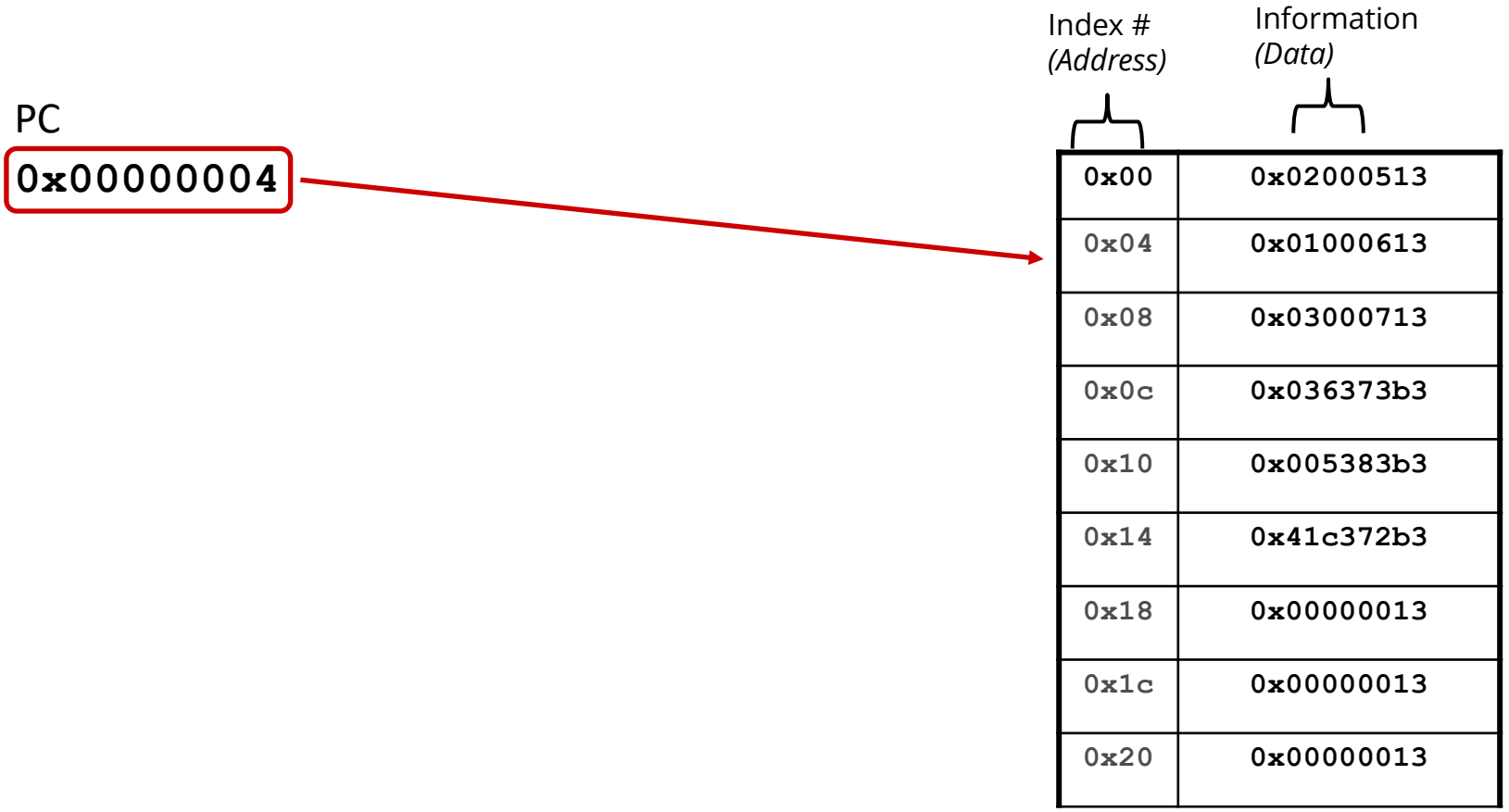
Program Counter

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



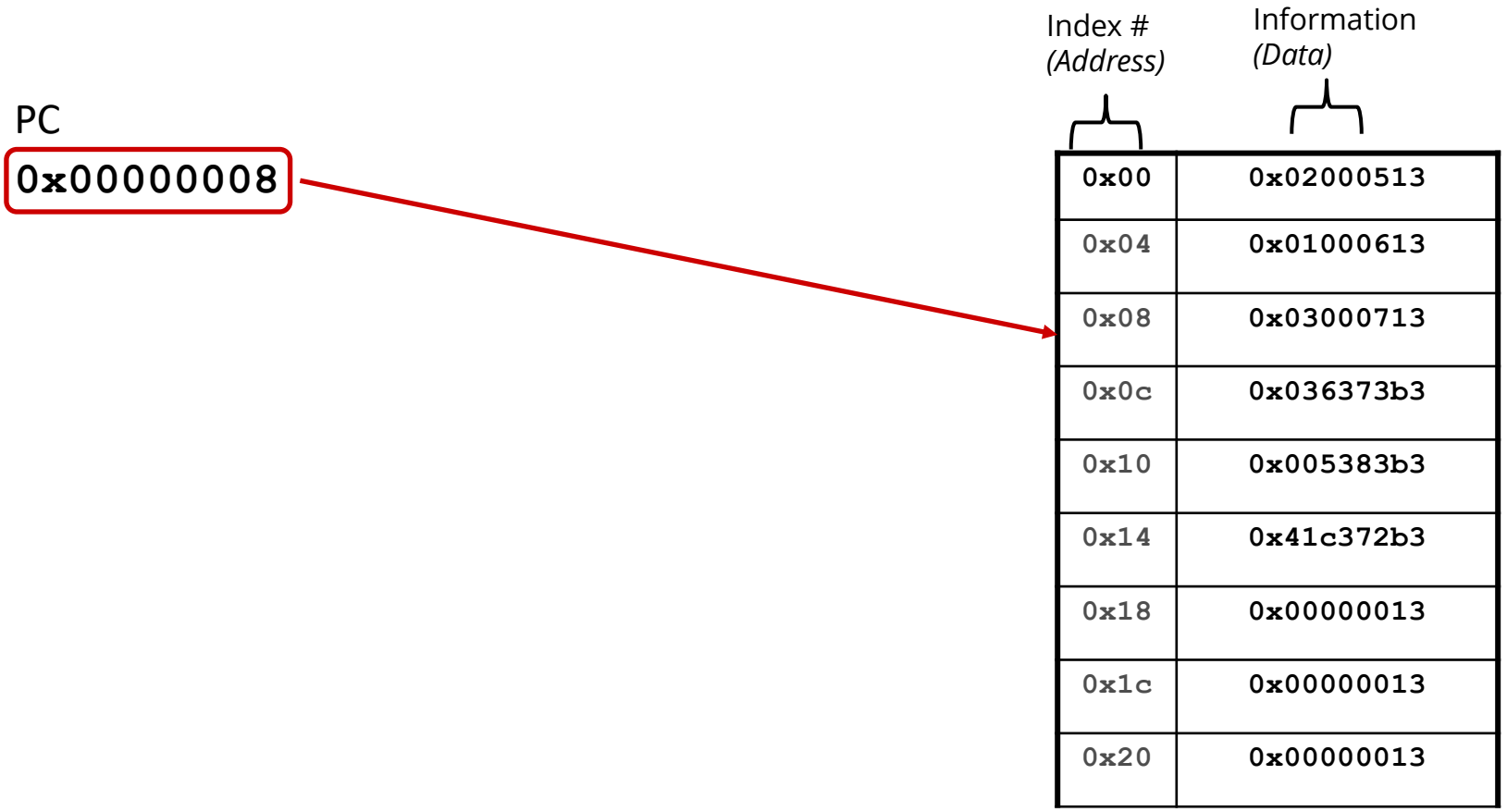
Program Counter

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



Program Counter

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

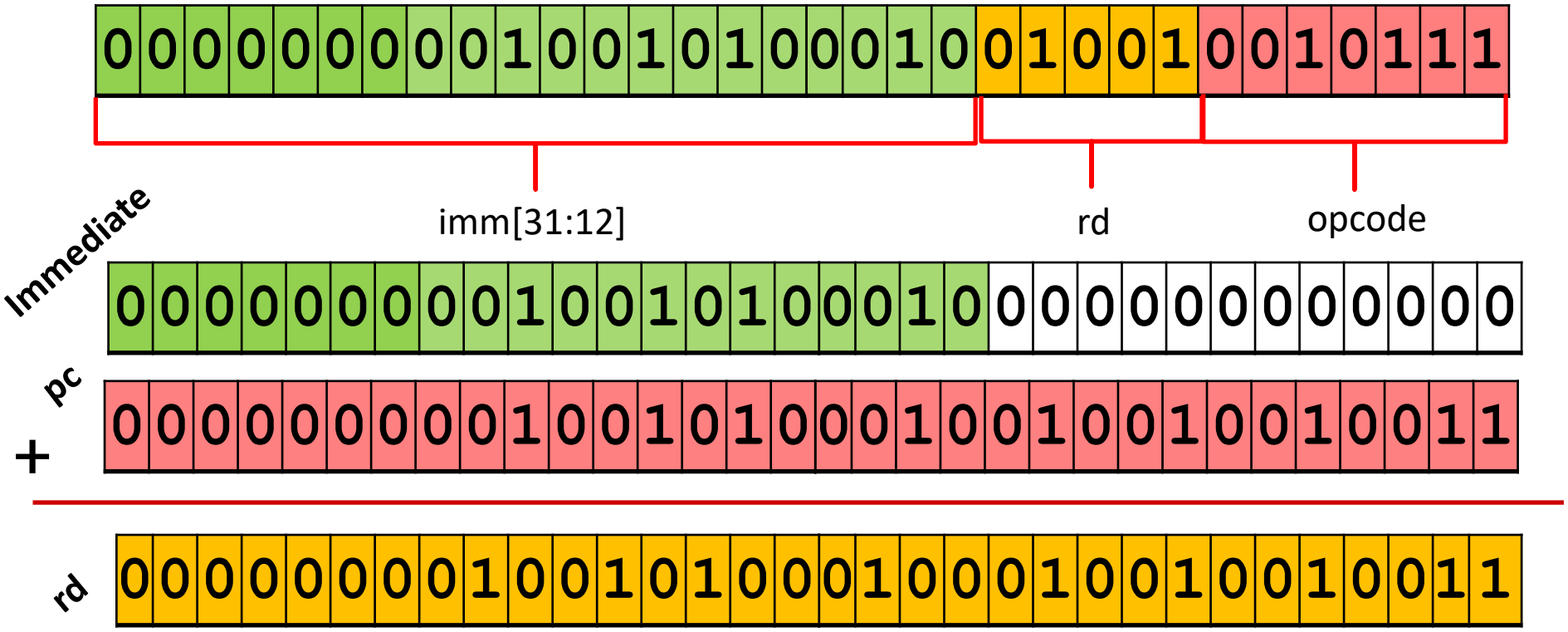


Lecture Outline

- Machine Code
  - RISC-V Encoding
  - In Memory
- The Program Counter
  - AUIPC
  - JALR
- Conditional Statements
  - Branching
- C in RISC-V
  - General Stack Setup
  - Register Usage
  - Full Set Up

Accessing the Program Counter

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



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

Changing the Program Counter

`jalr rd, imm12(rs1)`

- "Jump and Link Register"

```
//`link` part
rd = pc+4;
```

Why is it useful to have pc + 4?

So that we can *'return'* back to where we jumped from

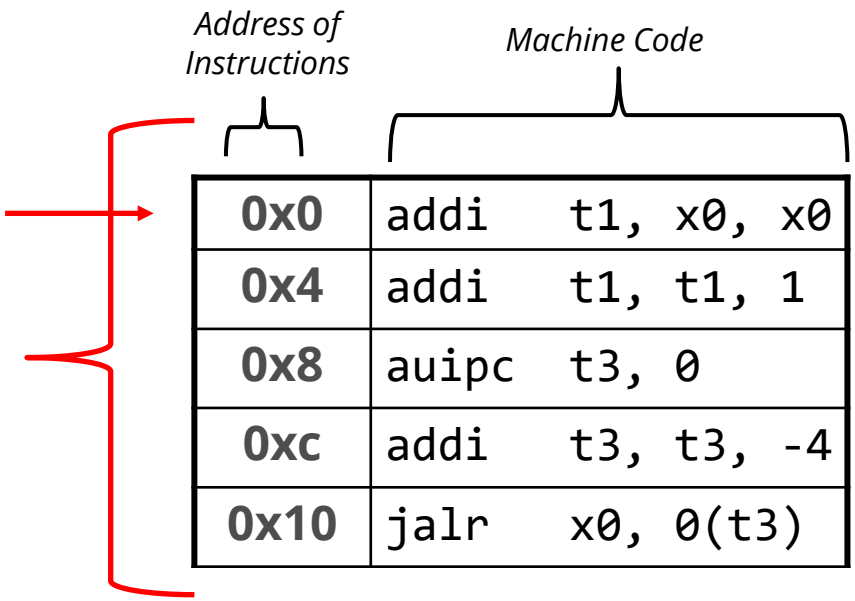
```
//`jump` part
pc = (rs1+sign_extend(imm12)) & ~0x1
```

What does this do?

"Clearing the least-significant bit when calculating the JALR target address both simplifies the hardware slightly"

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

Our First Infinite Loop RISC-V Program

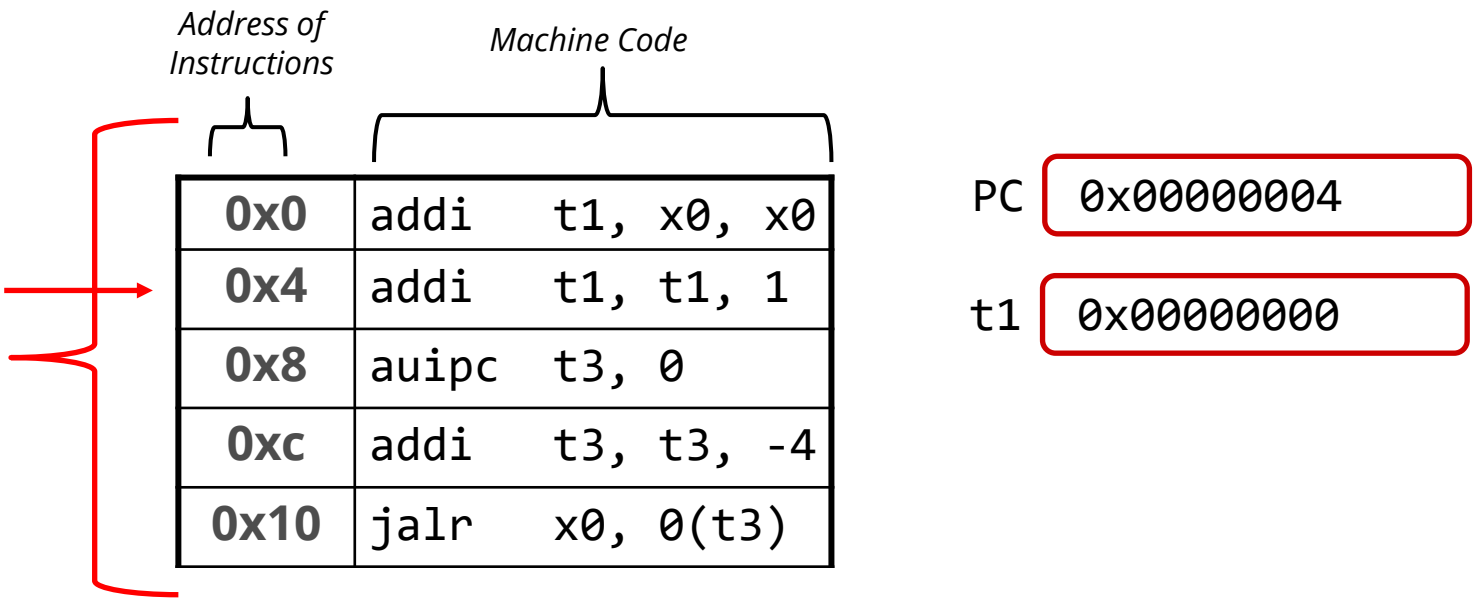


PC 0x00000000

t1 ??????????

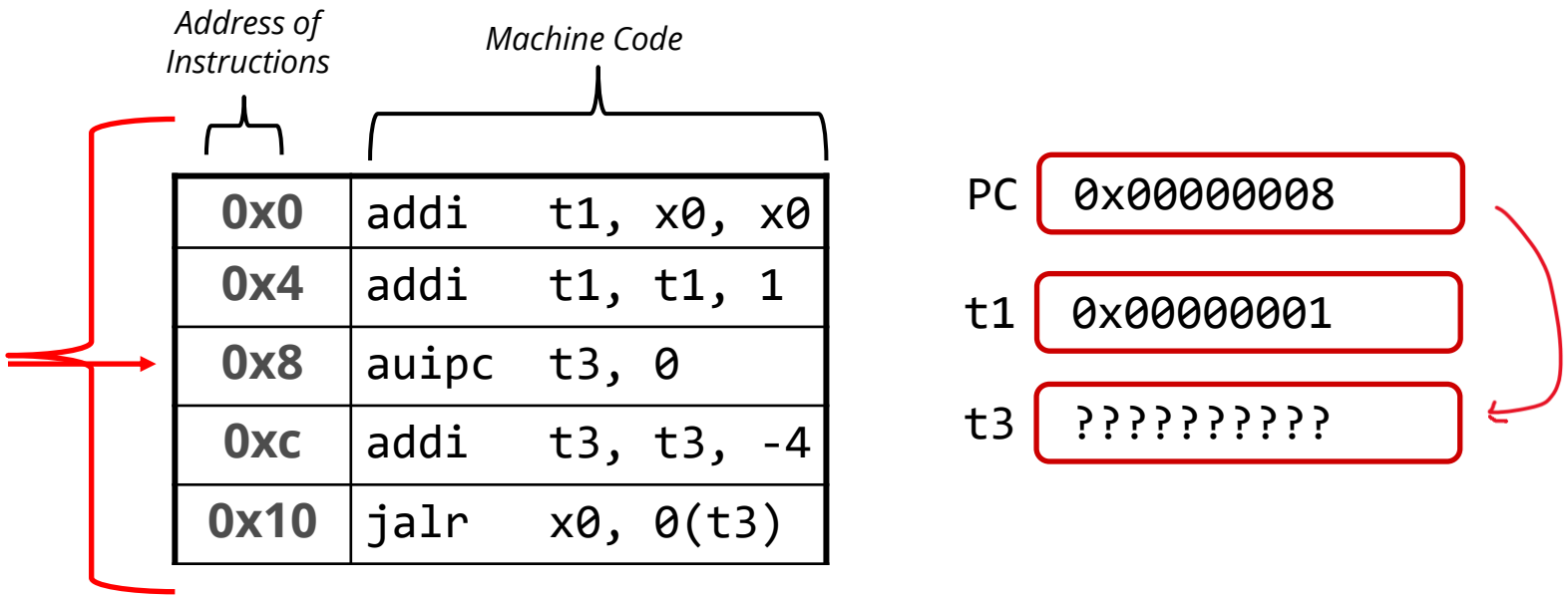
Disclaimer: the real intricacies of the PC are hidden, purely education example

Our First Infinite Loop RISC-V Program



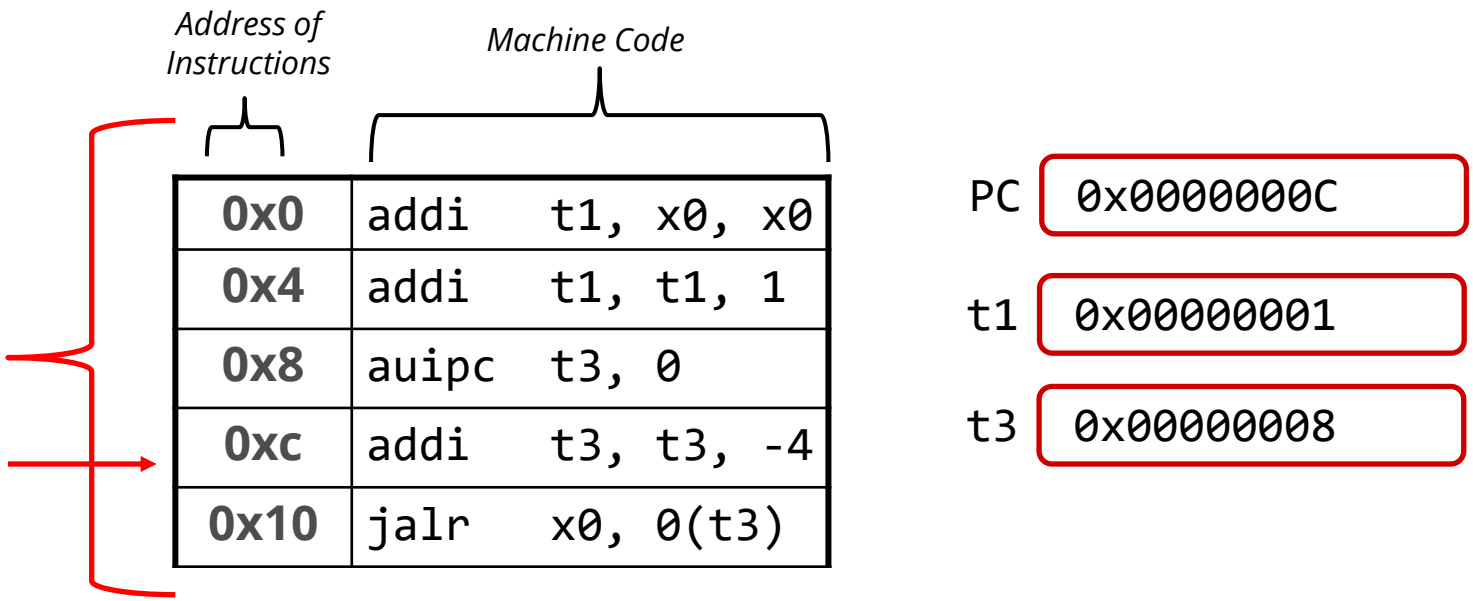
Disclaimer: the real intricacies of the PC are hidden, purely education example

Our First Infinite Loop RISC-V Program



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

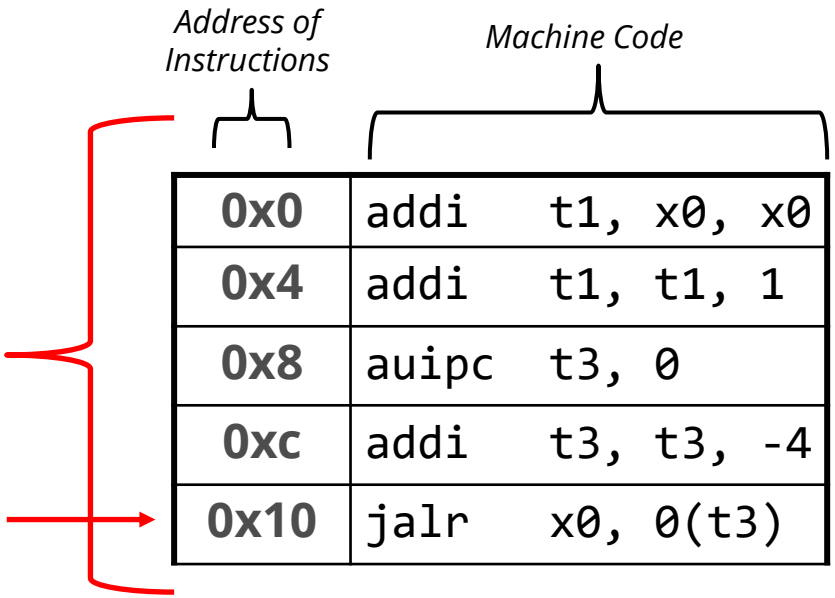
Our First Infinite Loop RISC-V Program



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

Poll Everywhere

pollev.com/cis2400



PC 0x00000010

t1 0x00000001

t3 0x00000004

After this instruction is finished; what value is stored in PC?

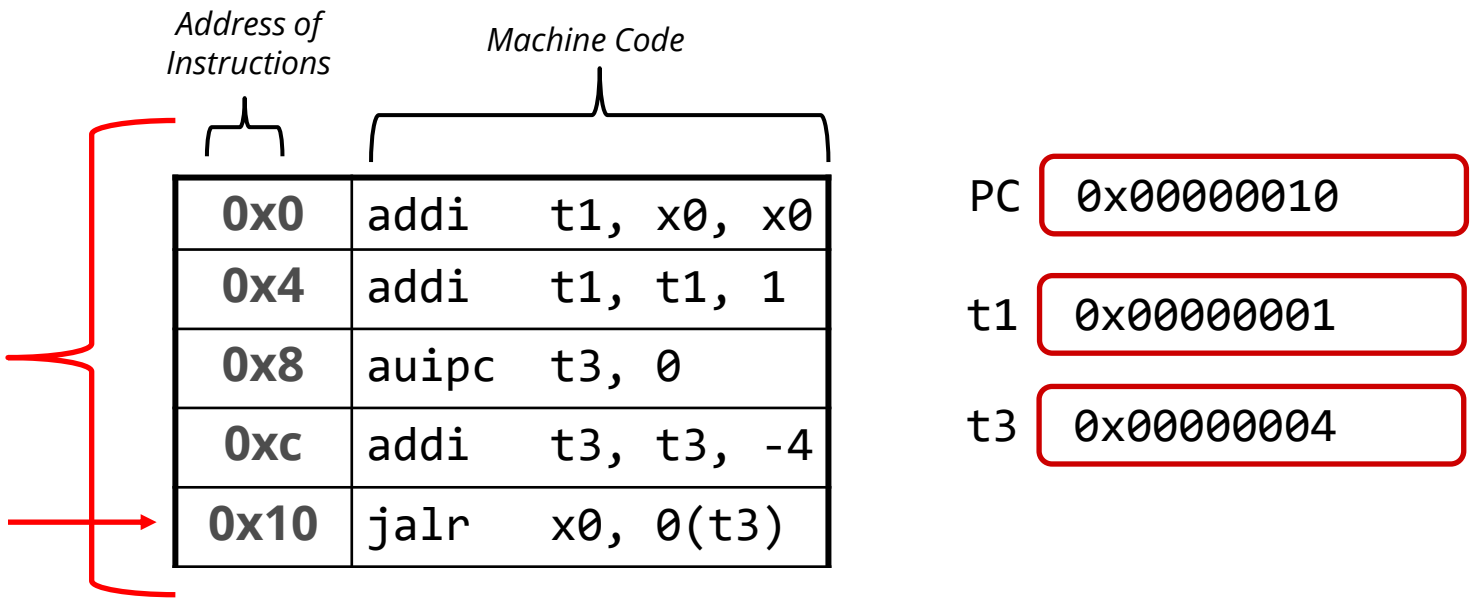
A) 0x00000010

B) 0x0000000c

C) 0x00000008

D) 0x00000004

Our First Infinite Loop RISC-V Program



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

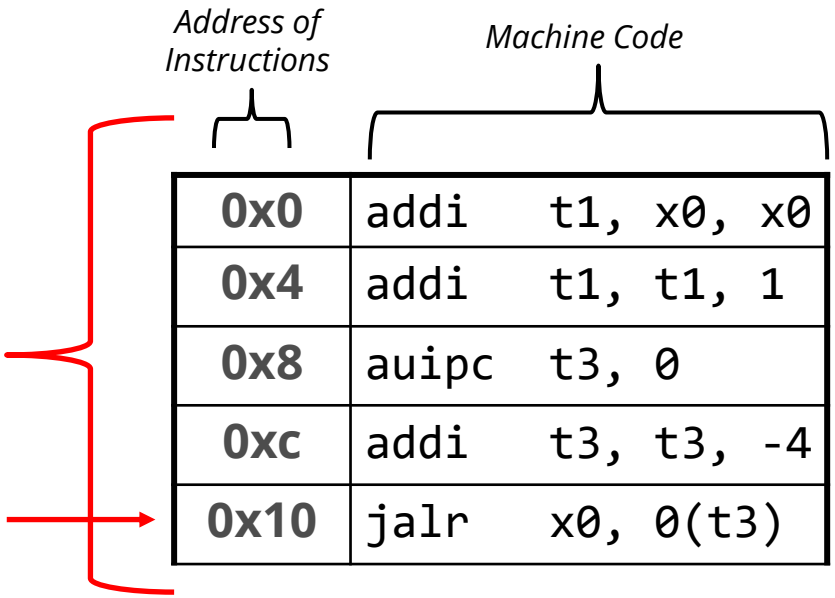
x0 never changes, writing to it does nothing (because it's always zero)

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

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

Poll Everywhere

pollev.com/cis2400



PC 0x00000010

t1 0x00000001

t3 0x00000004

After this instruction is finished; what value is stored in PC?

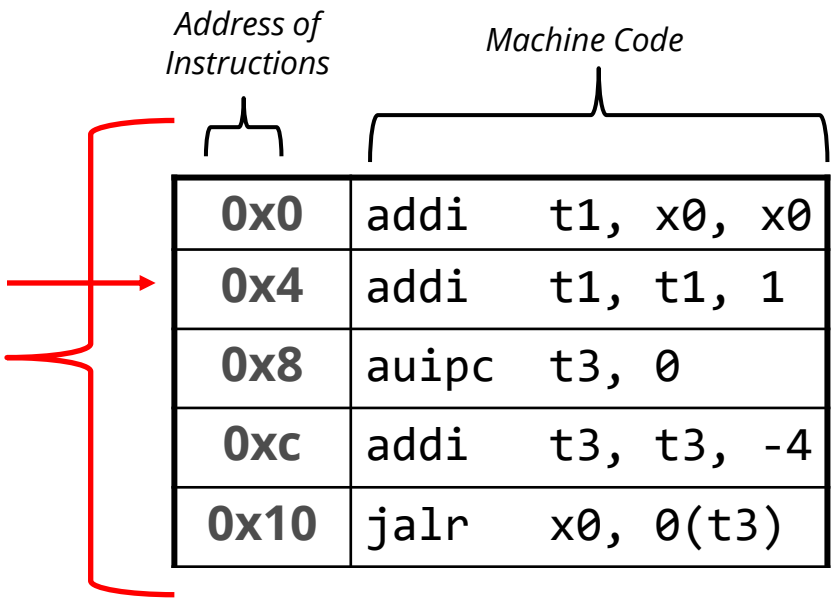
A) 0x00000010

B) 0x0000000c

C) 0x00000008

D) 0x00000004

Our First Infinite Loop RISC-V Program



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

Lecture Outline

- Machine Code
  - RISC-V Encoding
  - In Memory
- The Program Counter
  - AUIPC
  - JALR
- Conditional Statements
  - Branching
- C in RISC-V
  - Caller/Callee Owned Registers
  - Stack Pointer
  - Full Set Up

Labels

- Would be difficult to calculate the offsets for jumps yourself
- Can use Labels to 'jump' to different routines
- Handled by ***assembler***

`jal rd, label`

```
main:
    lw a0, 0(x2)            # Load first number into a0
    lw a1, 4(x2).           # Load second number into a1

    # Call the add_numbers subroutine
    jal ra, add_numbers         # Jump to add_numbers and link

    # Store the result in memory
    sw a0, 0(x2)

add_numbers:
    add a0, a0, a1          # a0 = a0 + a1 (result)
    jalr x0, 0(x1)          # Return to caller using x1
```

Control Flow

Registers

x1	???????????
x2	0x00000ff0
a0	???????????
a1	???????????
pc	main + 0

```
main:
    lw a0, 0(x2)            # Load first number into a0
    lw a1, 4(x2).           # Load second number into a1

    # Call the add_numbers subroutine
    jal x1, add_numbers     # Jump to add_numbers and link

    # Store the result in memory
    sw a0, 0(x2)

add_numbers:
    add a0, a0, a1          # a0 = a0 + a1 (result)
    jalr x0, 0(x1)          # Return to caller using x1
```

This tells you which instruction we are currently executing

No need to write the address…
This is easier to understand

Memory

address	value
0x00000ff4	0x0000000a
0x00000ff0	0x00000001

x2

This is the literal stack

Control Flow

Registers

x1	????????????
x2	0x00000ff0
a0	0x00000001
a1	????????????
pc	main + 4

```
main:
    lw a0, 0(x2)            # Load first number into a0
    lw a1, 4(x2).           # Load second number into a1

    # Call the add_numbers subroutine
    jal x1, add_numbers     # Jump to add_numbers and link

    # Store the result in memory
    sw a0, 0(x2)

add_numbers:
    add a0, a0, a1          # a0 = a0 + a1 (result)
    jalr x0, 0(x1)          # Return to caller using x1
```

Memory

address	value
0x00000ff4	0x0000000a
0x00000ff0	0x00000001

x2

Control Flow

Registers

x1	????????????
x2	0x00000ff0
a0	0x00000001
a1	0x0000000a
pc	main + 8

```
main:
    lw a0, 0(x2)            # Load first number into a0
    lw a1, 4(x2).           # Load second number into a1

    # Call the add_numbers subroutine
    jal x1, add_numbers     # Jump to add_numbers and link

    # Store the result in memory
    sw a0, 0(x2)

add_numbers:
    add a0, a0, a1          # a0 = a0 + a1 (result)
    jalr x0, 0(x1)          # Return to caller using x1
```

Memory

address	value
0x00000ff4	0x0000000a
0x00000ff0	0x00000001

x2

Control Flow

Registers

x1	main + 12
x2	0x00000ff0
a0	0x00000001
a1	0x0000000a
pc	add_numbers + 0

```
main:
    lw a0, 0(x2)            # Load first number into a0
    lw a1, 4(x2).           # Load second number into a1

    # Call the add_numbers subroutine
    jal x1, add_numbers     # Jump to add_numbers and link

    # Store the result in memory
    sw a0, 0(x2)

add_numbers:
    add a0, a0, a1          # a0 = a0 + a1 (result)
    jalr x0, 0(x1)          # Return to caller using x1
```

Memory

address	value
0x00000ff4	0x0000000a
0x00000ff0	0x00000001

x2

Control Flow

Registers

x1	main + 12
x2	0x00000ff0
a0	0x0000000b
a1	0x0000000a
pc	add_numbers + 4

```
main:
    lw a0, 0(x2)            # Load first number into a0
    lw a1, 4(x2).           # Load second number into a1

    # Call the add_numbers subroutine
    jal x1, add_numbers     # Jump to add_numbers and link

    # Store the result in memory
    sw a0, 0(x2)

add_numbers:
    add a0, a0, a1          # a0 = a0 + a1 (result)
    jalr x0, 0(x1)          # Return to caller using x1
```

Memory

address	value
0x00000ff4	0x0000000a
0x00000ff0	0x00000001

x2

Control Flow

Registers

x1	main + 12
x2	0x00000ff0
a0	0x0000000b
a1	0x0000000a
pc	main + 12

```
main:
    lw a0, 0(x2)            # Load first number into a0
    lw a1, 4(x2).           # Load second number into a1

    # Call the add_numbers subroutine
    jal x1, add_numbers     # Jump to add_numbers and link

    # Store the result in memory
    sw a0, 0(x2)

add_numbers:
    add a0, a0, a1          # a0 = a0 + a1 (result)
    jalr x0, 0(x1)          # Return to caller using x1
```

Memory

address	value
0x00000ff4	0x0000000a
0x00000ff0	0x00000001

x2

Control Flow

Registers

x1	main + 12
x2	0x00000ff0
a0	0x0000000b
a1	0x0000000a
pc	main + 16

```
main:
    lw a0, 0(x2)            # Load first number into a0
    lw a1, 4(x2).           # Load second number into a1

    # Call the add_numbers subroutine
    jal x1, add_numbers     # Jump to add_numbers and link

    # Store the result in memory
    sw a0, 0(x2)
    jal x0, exit
add_numbers:
    add a0, a0, a1          # a0 = a0 + a1 (result)
    jalr x0, 0(x1)          # Return to caller using x1
```

x2 has alias “sp”

This is the stack pointer register

Memory

address	value
0x00000ff4	0x0000000a
0x00000ff0	0x0000000b

x2

More Instruction Types

Branching and Conditional Statements

```
beq rs1, rs2, label // if(rs1 == rs2): jump to label
bne rs1, rs2, label // if(rs1 == rs2): jump to label
blt  rs1, rs2, label // if(rs1 < rs2):  jump to label
bltu rs1, rs2, label // if( rs1 (unsigned) < rs2 (unsigned) ):  jump to label
```

Why do we need **`blt`** *and* **`bltu`***?*

We don't compare unsigned and signed numbers in the same way…

Implement this using RISC-V

```
//load int @ i
//save is an array of ints
save[i];
```

x22 `0x00000000` integer i

x25 `0xffffac00` address of save

What is the RISC-V assembly code corresponding to this C code?

We want to load the value, save[i], into register x9.

Poll Everywhere

pollev.com/cis2400

```
//load int @ i
//save is an array of ints
save[i];
```

x22 `0x00000000` integer i

x25 `0xffffac00` address of save

Which sets the correct offset in register x10 to access arr @ i?

a)
```
slli x10, x22, 2
add  x10, x10, x25
```

b)
```
slli x10, x22, 1
add  x10, x10, x25
```

c)
```
slli x10, x22, 3
add  x10, x10, x25
```

d)
```
srli x10, x22, 2
add  x10, x10, x25
```

Poll Everywhere

pollev.com/cis2400

```
//load int @ i
//save is an array of ints
save[i];
```

x22 `0x00000000` integer i

x25 `0xffffac00` address of save

Which sets the correct offset in register x10 to access arr @ i?

a)
```
slli x10, x22, 2
add  x10, x10, x25
```

b)
```
slli x10, x22, 1
add  x10, x10, x25
```
Multiplies by two only

c)
```
slli x10, x22, 3
add  x10, x10, x25
```
left shift by 3, is multiplying by 8

d)
```
srli x10, x22, 2
add  x10, x10, x25
```
Shift right by 2

Poll Everywhere

pollev.com/cis2400

```
//load int @ i
//save is an array of ints
save[i];
```

x22 `0x00000000` integer i

x25 `0xffffac00` address of save

Which sets the correct offset in register x10 to access arr @ i?

a)
```
slli x10, x22, 2
add  x10, x10, x25
```

Implement this using RISC-V

```
//load int @ i
//save is an array of ints
save[i];
```

x22 `0x00000000` integer i

x25 `0xffffac00` address of save

What is the RISC-V assembly code corresponding to this C code?

We want to load the value, save[i], into register x9.

```
slli x10, x22, 2  //shift left logical immediate
add  x10, x10, x25
lw   x9,  0(x10)
```

Implement this using RISC-V

```
while(save[i] == k){
    //loop forever
}
//exit
```

x22 0x00000000

integer i

x24 0x00000010

integer k

x25 0xffffac00

address of save

What is the RISC-V assembly code corresponding to this C code?

```
Loop:
    slli x10, x22, 2  //shift left logical immediate
    add  x10, x10, x25
    lw   x9,  0(x10)
    ...

Exit:
```

Implement this using RISC-V

```
while(save[i] == k){
    //loop forever
}
//exit
```

x22 0x00000000
integer i

x24 0x00000010
integer k

x25 0xffffac00
address of save

What is the RISC-V assembly code corresponding to this C code?

```
Loop:
    slli x10, x22, 2  //shift left logical immediate
    add  x10, x10, x25
    lw   x9,  0(x10)
    bne  x9,  x24, Exit
    beq  x9,  x24, Loop

Exit:
```

Implement this using RISC-V

```
while(save[i] == k){
    i++
}
```

x22 0x00000000 integer i

x25 0xffffac00 address of save

x24 0x00000010 integer k

What is the RISC-V assembly code corresponding to this C code?

```
Loop:
    slli x10, x22, 2  //shift left logical immediate
    add  x10, x10, x25
    lw   x9,  0(x10)
    bne  x9,  x24, Exit
    …
    beq  x9,  x24, Loop

Exit:
```

Implement this using RISC-V

```
while(save[i] == k){
    i++
}
```

x22 0x00000000
integer i

x25 0xffffac00
address of save

x24 0x00000010
integer k

What is the RISC-V assembly code corresponding to this C code?

```
Loop:
    slli x10, x22, 2  //shift left logical immediate
    add  x10, x10, x25
    lw   x9,  0(x10)
    bne  x9,  x24, Exit
    addi x22, x22, 1
    beq  x9,  x24, Loop

Exit:
```

Lecture Outline

- Machine Code
  - RISC-V Encoding
  - In Memory
- The Program Counter
  - AUIPC
  - JALR
- Conditional Statements
  - Branching
- C in RISC-V
  - Caller/Callee Owned Registers
  - Stack Pointer
  - Full Set Up

Caller and Callee Saved Registers

- Identify which registers need to be saved before modifying them.

```
caller:
   //instructions here
   //I know x8−x9 won't be modified
   jal x1, callee
  //x8−x9 will be the same
```

```
callee:
   // If I use x8 and x9
   // I need to save them
   // and restore before returning
   jalr x0, 0(x1) // go back to caller
```

- Determine which registers are important for the current subroutine calls.

```
caller:
   //the callee expects parameters
   // to be in a0 – a7
   jal x1, callee
```

```
callee:
   // if I have parameters I expect
   // I know they're in a0 – a7
   jalr x0, 0(x1) // go back to caller
```

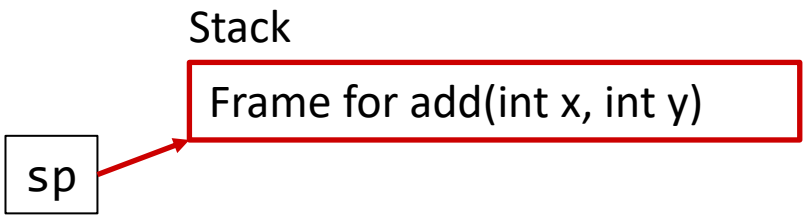
Caller and Callee Saved Registers

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Stack Pointer Register

- Points to the last byte available by the stack
  - When routine needs more memory; just subtract from the Stack Pointer (sp)

```
void add(int x, int y){
        int z = x + y;
        return z;
}
```

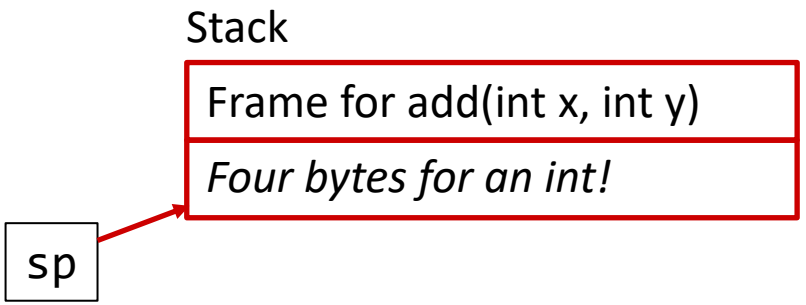


I declare an int, so I need more space…

Stack Pointer Register

- Points to the last byte available by the stack
  - When routine needs more memory; just subtract from the Stack Pointer (sp)

```
void add(int x, int y){
        int z = x + y;
        return z;
}
```



I declare an int, so I need more space…

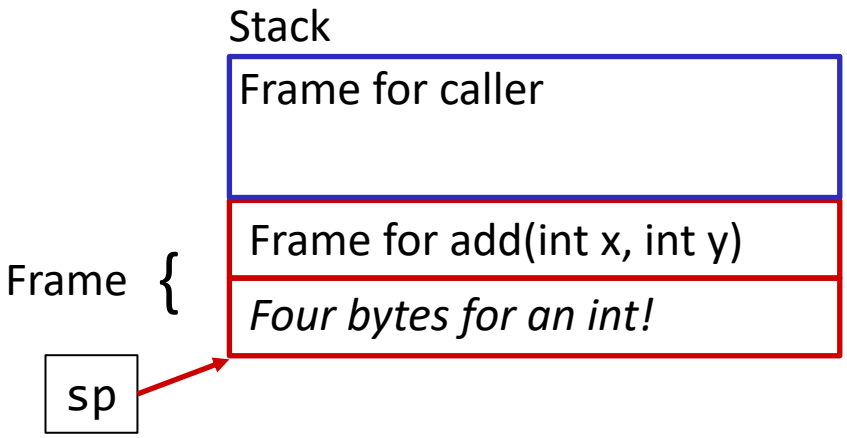
```
addi sp, sp, -4
```

Stack Pointer Register

When routine is done, restore stack pointer

- Done by adding the same amount the SP was subtracted by in setup

```
void add(int x, int y){
        int z = x + y;
        return z;
}
```



If the entire frame is 12 bytes, then we add 12

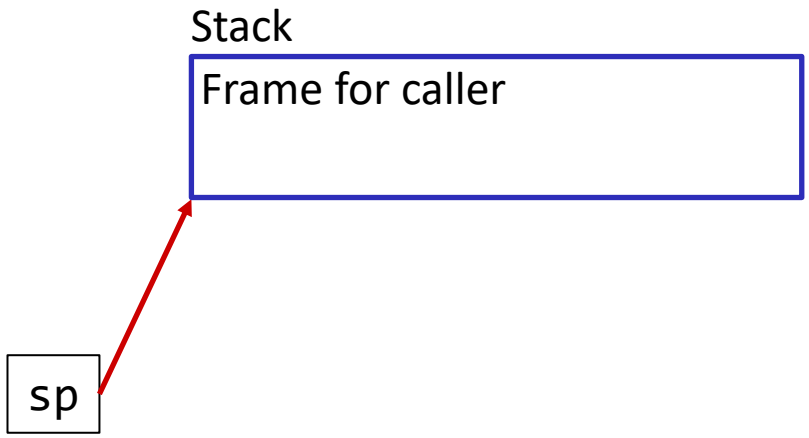
```
addi sp, sp, 12
```

Stack Pointer Register

When routine is done, restore stack pointer

- Done by adding the same amount the SP was subtracted by in setup

```
void add(int x, int y){
        int z = x + y;
        return z;
}
```



If the entire frame is 12 bytes, then we add 12

```
addi sp, sp, 12
jalr x0, 0(x1)
```

And now we return to the caller

Full Routine

```
void add(int x, int y){
        int z = x + y;
        return z;
}
```

```
add(int, int):
    addi sp, sp, -32
    sw ra, 28(sp)
    sw s0, 24(sp)
    addi s0, sp, 32
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    sw a0, -20(s0)
    lw a0, -20(s0)
    lw ra, 28(sp)
    lw s0, 24(sp)
    addi sp, sp, 32
    ret
```

Stack Pointer is being decremented to make more space for registers it needs to save and local variables it will make.

No Optimization Here

Makes space for 32 bytes on its Stack

Saves the following Registers `ra, s0` using the `sp` directly

Saves the following Registers a0, a1 using `s0` directly

Then we load registers with same values?

a0 and a1 are Function Argument Registers
So this is holding ***x*** and ***y*** respectively

Full Routine

```
void add(int x, int y){
        int z = x + y;
        return z;
}
```

No Optimization Here

We finally add the registers here

Now, we save z!

Now, we grab `ra` and `s0` that we saved

Finally, we 'restore' sp

And we return back to caller

```
add(int, int):
    addi sp, sp, -32
    sw ra, 28(sp)
    sw s0, 24(sp)
    addi s0, sp, 32
    sw a0, -12(s0)
    sw a1, -16(s0)
    lw a0, -12(s0)
    lw a1, -16(s0)
    add a0, a0, a1
    sw a0, -20(s0)
    lw a0, -20(s0)
    lw ra, 28(sp)
    lw s0, 24(sp)
    addi sp, sp, 32
    jalr x0, 0(x1) //ret
```

-20 is the offset where z will be

a0 and a1 are Function Argument Registers
So this is holding ***x*** and ***y*** respectively

a0 is also where ***return value is saved***…

Full Routine 'Fully' Optimized

```
void add(int x, int y){
        int z = x + y;
        return z;
}
```

```
add(int, int):
    add a0, a0, a1
    jalr x0, 0(x1)
```

- Registers `a0` and `a1` are used as the first and second arguments for routines.
- Additionally, `a0` is used to store the return value, as the caller expects the return value in `a0`.
- This routine efficiently avoids setting up the stack if the only task is to add the registers together.
- From 15 instructions to 2.

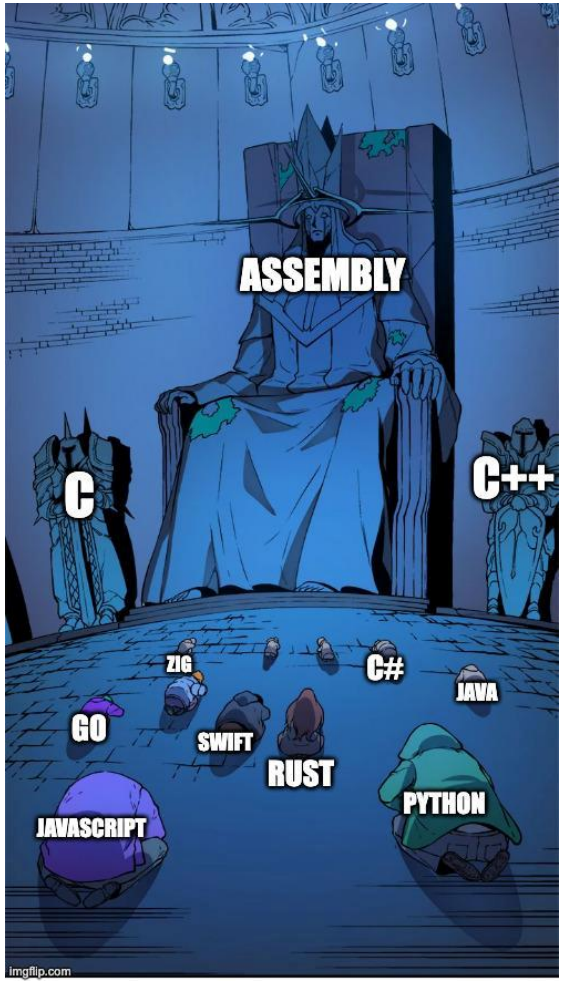
Now, you can read RISC-V no prob…

A Ton Of Stuff Left Out

- How the Stack Frames are Set Up
- Where the return address is saved
- When do we not need the return address
- What about if we want to directly speak to a hardware interface (like a keyboard)
- Ahhhh
- Psuedoinstructions, assembler, yadayadayada
- How to write the assembly from ***scratch***, declaring variables
- In time all these questions will be answered…..

Next Lecture: Travis!

- How is Machine Code ***actually used by hardware?***



Enjoy your weekend!

Here's a meme from facebook