RISC-V Assembly

RISC-V ISA available at:
https://llp.mit.edu/6190/_static/FA26/material/resources/references/riscv_isa_reference.pdf

Let's Sum an Array

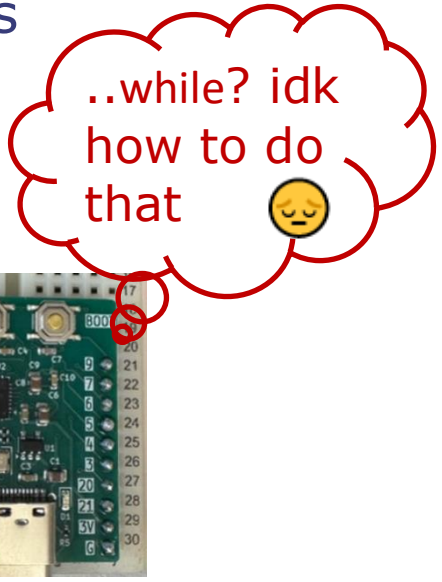
Assume we have an array arr with 10 integers starting at memory address 0x700.

C Program

```
int i = 0;
int arr_sum = 0;
int arr_len = 10;
while (i < arr_len) {
  int val = *(arr + i);
  arr_sum += val;
  i++;
}
```

This is pretty readable (now that we know C!)

- Explicit data types
- While loop
- Pointer arithmetic to access array elements



However, computer hardware does not directly understand or implement this.

"General Purpose" Processor

- A general-purpose processor has a list of *instructions* that it can execute using its hardware.
  - High-level languages (Python, Java, C) are translated into a language that uses just these instructions.
  - This language is called *assembly language*
- Assembly language then gets translated into binary that the processor has hardware to decode.
  - This is called *machine code*
  - This gets stored in memory for the processor to fetch as it goes through the program.
  - More on this translation later!

Takeaway: Assembly language tells us the exact steps that a processor carries out when running through a program.

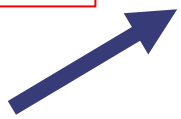
Let's Sum an Array

Assume we have an array arr with 10 integers starting at memory address 0x700.

C Program

```
int i = 0;
int arr_sum = 0;
while (i < 10) {
  int val = *(arr + i);
  arr_sum += val;
  i++;
}
```

this would translate into something like



Assembly Language

```
  addi x5, x0, 0
  addi x6, x0, 0
  addi x7, x0, 40
loop:
  bge x5, x7, end
  lw x8, 0x700(x5)
  add x6, x6, x8
  addi x5, x5, 4
  jal x0, loop
end:
```

Let's Sum an Array

Assume we have an array arr with 10 integers starting at memory address 0x700.

Assembly Language

```
   addi x5, x0, 0
   addi x6, x0, 0
   addi x7, x0, 40
loop:
   bge x5, x7, end
   lw x8, 0x700(x5)
   add x6, x6, x8
   addi x5, x5, 4
   jal x0, loop
end:
```

Machine Language (Code)

```
0x00000293
0x00000313
0x02800393
0x0072da63
0x7002a403
0x00830333
0x00428293
0xff1ff06f
```



Direct translation (same level of detail)!

Assembly is for the programmer, machine code is for the processor.

High Level vs Assembly Language

High Level Language

1. Complex arithmetic and logical operations
2. Complex data types and data structures
3. Complex control structures - conditional statements, loops and functions
4. Not suitable for direct implementation in hardware

Assembly Language

1. Primitive arithmetic and logical operations
2. Primitive data structures – bits/bytes
3. Control flow instructions
4. Designed to be directly implementable in hardware

tedious programming!

Assembly Language Program

- An assembly language program is a sequence of instructions which execute in a sequential order unless a *control flow* instruction is executed

- Each instruction specifies an operation directly supported by the processor hardware.
  - Examples of supported operations:
    - Arithmetic and logical operations
    - Comparisons
    - Jumps
    - Memory accesses

Instruction Set Architecture (ISA)

- ISA: The contract between software and hardware
  - Functional definition of operations and storage locations
  - Precise description of how software can invoke and access them
- Many different Instruction Set Architectures out there
- Two main approaches:
  - **CISC** (Complex Instruction Set Computer):
    - Instruction set is ***larger*** with more options and operation variants
  - **RISC** (Reduced Instruction Set Computer):
    - Instruction set is ***smaller*** with only bare minimum operations

CISC Philosophy

- Fundamental instructions are more complex
- It's ok to have lots of different instructions to handle different variants and needs – larger instruction set.
- Programs will tend to have fewer instructions.

RISC Philosophy

- Fundamental instructions are more basic
  - but generally take less time for the processor to execute
- Do not have lots of different instructions to handle different variants and needs – smaller instruction set.
- You may need to use more instructions than a CISC processor for a given program, but…
  - The hope is you can do those simpler instructions *faster* than you can do your fewer instructions in a CISC processor!

An Example

- We'd like to add two numbers from memory locations 0x7E8 and 0x7EC and store the resulting sum back in 0x7EC.
- In a hypothetical **CISC** architecture you might be able to do this in one instruction:

ADDMR 0x7EC, 0x7E8

"add contents stored at memory address 0x7E8 to contents stored at 0x7EC and put the resulting sum into 0x7EC"

- With a **RISC** architecture, this might look like:

```
addi x10, x0, 0x7E8  # x10 <- 0x7E8 + 0 = 0x7E8
lw x11, 0(x10)       # x11 <- Mem[x10]
lw x12, 4(x10)       # x12 <- Mem[x10 + 4]
add x11, x11, x12    # x11 <- x11 + x12
sw x11, 4(x10)       # Mem[x10 + 4] <- x11
```

Who is What?

- Intel: CISC (x86 architecture is their creation)
- ARM: RISC (ARM stands for **A**dvanced **R**ISC **M**achines)
- Apple: used to be CISC when x86…now with M1…ARM so RISC
- …
- 6.190: RISC…and not just any RISC…

RISC-V! 

RISC-V ISA

- A newer (2010), open, free ISA from Berkeley
- Several variants
  - RV32, RV64, RV128: Different data widths
  - 'I': Base Integer instructions
  - 'M': Multiply and Divide
  - 'C': Compressed instructions
  - 'F' and 'D': Single- and Double-precision floating point
  - And many other extensions!!
- We will study the RV32I processor, which is the base integer 32-bit variant.
- Our ESP32-C3 microcontroller (lab) is RV32IMC
  - **I**nteger, Int **M**ult/Divide, **C**ompressed Instructions
- In 6.191, you will build your own RV32I processor!

Registers vs. Memory

- Both store information

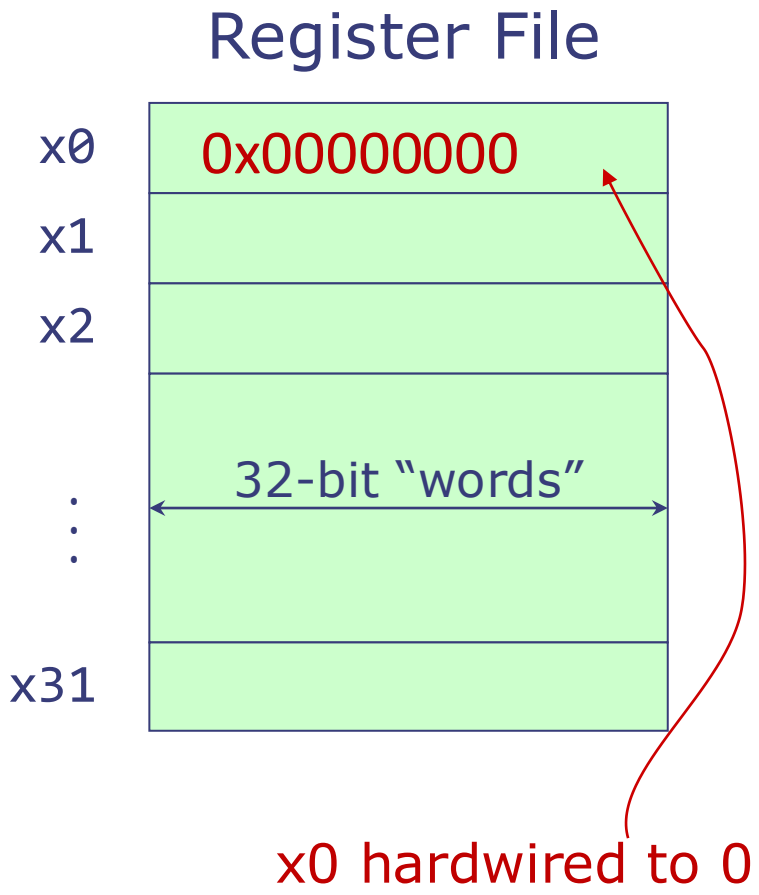
Registers	Memory
• Expensive • Physically large • Quick to access • Actually in the computer itself	• Cheap • Very dense/small • Slower to access • Further away/separate from the computer.

- Conclusion for most computers – *compromise*:
  - Have a small set of registers (maybe twenty or thirty) that you use as much as possible as temporary variables
  - Use memory as rarely as possible!

RISC-V Processor Storage

Registers

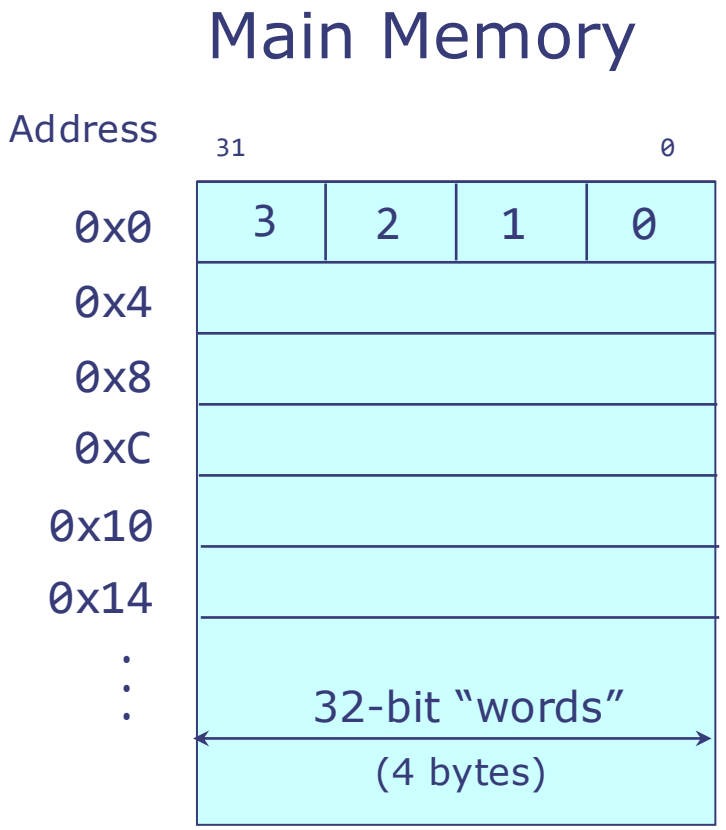
- 32 general-purpose registers
  - x0 – x31
- Each register is 32 bits wide
- x0 = 0



RISC-V Processor Storage

Memory

- Each memory location is 32 bits (1 word) wide
  - Memory is byte (8 bits) addressable
  - Addresses of adjacent words are 4 apart.
- Addresses are 32 bits
  - Can address 2^{32} bytes or 2^{30} words.



RISC-V Instructions

Instruction	Syntax	Description	Execution
LUI	`lui rd, luiConstant`	Load Upper Immediate	`reg[rd] <= luiConstant « 12`
JAL	`jal rd, label`	Jump and Link	`reg[rd] <= pc + 4` `pc <= label`
JALR	`jalr rd, offset(rs1)`	Jump and Link Register	`reg[rd] <= pc + 4` `pc <= {(reg[rs1] + offset)[31:1], 1'b0}`
BEQ	`beq rs1, rs2, label`	Branch if =	`pc <= (reg[rs1] == reg[rs2]) ? label: pc + 4`
BNE	`bne rs1, rs2, label`	Branch if ≠	`pc <= (reg[rs1] != reg[rs2]) ? label: pc + 4`
BLT	`blt rs1, rs2, label`	Branch if < (Signed)	`pc <= (reg[rs1] <`$_s$` reg[rs2]) ? label: pc + 4`
BGE	`bge rs1, rs2, label`	Branch if ≥ (Signed)	`pc <= (reg[rs1] >=`$_s$` reg[rs2]) ? label: pc + 4`
BLTU	`bltu rs1, rs2, label`	Branch if < (Unsigned)	`pc <= (reg[rs1] <`$_u$` reg[rs2]) ? label: pc + 4`
BGEU	`bgeu rs1, rs2, label`	Branch if ≥ (Unsigned)	`pc <= (reg[rs1] >=`$_u$` reg[rs2]) ? label: pc + 4`
LW	`lw rd, offset(rs1)`	Load Word	`reg[rd] <= mem[reg[rs1] + offset]`
SW	`sw rs2, offset(rs1)`	Store Word	`mem[reg[rs1] + offset] <= reg[rs2]`
ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <`$_s$` constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <`$_u$` constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, shamt`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « shamt`
SRLI	`srli rd, rs1, shamt`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »`$_u$` shamt`
SRAI	`srai rd, rs1, shamt`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »`$_s$` shamt`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2][4:0]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <`$_s$` reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <`$_u$` reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »`$_u$` reg[rs2][4:0]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »`$_s$` reg[rs2][4:0]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

RISC-V ISA: Instructions

- Three types of operations:
  - **Computational:** Perform arithmetic and logical operations on registers
  - **Loads and stores:** Move data between registers and main memory
  - **Control Flow:** Change the execution order of instructions

RISC-V Instructions

Instruction	Syntax	Description	Execution
LUI	`lui rd, luiConstant`	Load Upper Immediate	`reg[rd] <= luiConstant « 12`
JAL	`jal rd, label`	Jump and Link	`reg[rd] <= pc + 4` `pc <= label`
JALR	`jalr rd, offset(rs1)`	Jump and Link Register	`reg[rd] <= pc + 4` `pc <= {(reg[rs1] + offset)[31:1], 1'b0}`
BEQ	`beq rs1, rs2, label`	Branch if =	`pc <= (reg[rs1] == reg[rs2]) ? label: pc + 4`
BNE	`bne rs1, rs2, label`	Branch if ≠	`pc <= (reg[rs1] != reg[rs2]) ? label: pc + 4`
BLT	`blt rs1, rs2, label`	Branch if < (Signed)	`pc <= (reg[rs1] <$_s$ reg[rs2]) ? label: pc + 4`
BGE	`bge rs1, rs2, label`	Branch if ≥ (Signed)	`pc <= (reg[rs1] >=$_s$ reg[rs2]) ? label: pc + 4`
BLTU	`bltu rs1, rs2, label`	Branch if < (Unsigned)	`pc <= (reg[rs1] <$_u$ reg[rs2]) ? label: pc + 4`
BGEU	`bgeu rs1, rs2, label`	Branch if ≥ (Unsigned)	`pc <= (reg[rs1] >=$_u$ reg[rs2]) ? label: pc + 4`
LW	`lw rd, offset(rs1)`	Load Word	`reg[rd] <= mem[reg[rs1] + offset]`
SW	`sw rs2, offset(rs1)`	Store Word	`mem[reg[rs1] + offset] <= reg[rs2]`
ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <$_s$ constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, shamt`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « shamt`
SRLI	`srli rd, rs1, shamt`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »$_u$ shamt`
SRAI	`srai rd, rs1, shamt`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »$_s$ shamt`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2][4:0]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <$_s$ reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »$_u$ reg[rs2][4:0]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »$_s$ reg[rs2][4:0]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

RISC-V ISA: Instructions

- Three types of operations:
  - **Computational:** Perform arithmetic and logical operations on registers
  - **Loads and stores:** Move data between registers and main memory
  - **Control Flow:** Change the execution order of instructions
    - To support conditional statements and loops
    - To support function calls (next week)

Computational Instructions

- Four main classes of operations
  - Arithmetic
  - Comparison
  - Bitwise Logical
  - Shift
- Two formats:
  - Register-Register instructions
  - Register-Immediate instructions

Format	Arithmetic	Comparisons	Logical	Shifts
Register-Register	add, sub	slt, sltu	and, or, xor	sll, srl, sra
Register-Immediate	addi	slti, sltiu	andi, ori, xori	slli, srli, srai

Register-Register Instructions

Format: oper rd, rs1, rs2

- 2 source operand registers (rs1, rs2)
- 1 destination register for result (rd)

Operations (oper):

Arithmetic	Comparisons	Logical	Shifts
add, sub	slt, sltu	and, or, xor	sll, srl, sra

- add x3, x1, x2 # x3 ← x1 + x2
- slt x3, x1, x2 # If x1 < x2 then x3 = 1 else x3 = 0
- and x3, x1, x2 # x3 ← x1 & x2
- sll x3, x1, x2 # x3 ← x1 << x2

Register-Immediate Instructions

Format: oper rd, rs1, constant

- 1 source operand comes from register (rs1)
- 1 source operand comes from constant (constant)
- 1 destination register (rd)

12 bits (5 for shifts)

Operations (oper):

Arithmetic	Comparisons	Logical	Shifts
addi	slti, sltiu	andi, ori, xori	slli, srli, srai

- addi x3, x1, 4 # x3 ← x1 + 4
- slti x3, x1, 4 # If x1 < 4 then x3 = 1 else x3 = 0
- andi x3, x1, 4 # x3 ← x1 & 4
- slli x3, x1, 4 # x3 ← x1 << 4

- No subi, instead use addi with negative constant.
  - addi x3, x1, -4 # x3 ← x1 - 4

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - add x3, x1, x2 # x3 = x1 + x2

Base 10	Base 2
	111
5	00101
+ 3	+ 00011
8	*01000*

All Values are Binary

- Suppose: x1 = 00101; x2 = 00011
  - sll x3, x1, x2 # shift x1 left by x2 bits

00101
01010
10100
01000

Note fixed width!

Right Shifts: Logical vs. Arithmetic

- Suppose: x1 = **0**0101
  x2 = 00010

```
srl x3, x1, x2
```

00101
*0*0010
*00*001

```
sra x3, x1, x2
```

00101
*0*0010
*00*001

- Suppose: x1 = **1**0101
  x2 = 00010

```
srl x3, x1, x2
```

10101
*0*1010
*00*101

```
sra x3, x1, x2
```

10101
*1*1010
*11*101

Logical right shift: Shift in 0s.
Arithmetic right shift: Shift in value of most significant bit.

Logical Operations

- Suppose: x1 = 00101
  x2 = 00110

and x3, x1, x2

```
  00101
& 00110
-------
  00100
```

or x3, x1, x2

```
  00101
| 00110
-------
  00111
```

xor x3, x1, x2

```
  00101
^ 00110
-------
  00011
```

Putting Values into Registers

- What if we wanted to put 0x123 into x2?
- No problem!

```
addi x2, x0, 0x123 # x2 = 0x123 + 0
                   # x2 = 0x00000123
```



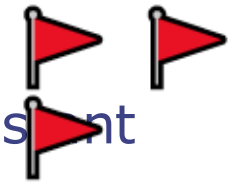
Can use the fact that x0 is always equal to 0!!

Putting Values into Registers

- What if we wanted to put 0x12345678 into x2?
- No problem!

```
addi x2, x0, 0x12345678 # x2 = 0x12345678 + 0
                        # x2 = 0x12345678
```

- Actually, there is a problem:
  - addi can only encode a 12 bit constant
  - 0x12345678 is more than 12 bits
  - The above code would not compile.



So… how would we do this?

LUI: Load Upper Immediate

Format: lui rd, luiConstant

- Puts 20-bit constant value (luiConstant) in the upper 20 bits of a register (rd)
- Appends 12 zeroes to the low end of the register
- Supports putting constants that are larger than 12 bits into a register

- What if we wanted to put 0x12345678 into x2?

```
lui x2, 0x12345      # put 0x12345 into upper 20 bits of x2
                     # x2 = 0x12345000
addi x2, x2, 0x678   # x2 = 0x12345000 + 0x678
                     # x2 = 0x12345678
```

Compound Computation

- Let's do a = ((b+3) >> c) - 1

1. Break up complex expression into basic computations.
   - Our instructions can only specify two source operands and one destination operand
2. Computational instructions can only access registers, not memory.
   - Assume a, b, c are in registers x1, x2, and x3 respectively. Use x4 for temp0, and x5 for temp1.

```
temp0 = b + 3;          addi x4, x2, 3
temp1 = temp0 >> c;     sra x5, x4, x3
a = temp1 - 1;          addi x1, x5, -1
```

Computational Instructions

Instruction	Syntax	Description	Execution
LUI	`lui rd, luiConstant`	Load Upper Immediate	`reg[rd] <= luiConstant « 12`
ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <$_s$ constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, shamt`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « shamt`
SRLI	`srli rd, rs1, shamt`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »$_u$ shamt`
SRAI	`srai rd, rs1, shamt`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »$_s$ shamt`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2][4:0]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <$_s$ reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »$_u$ reg[rs2][4:0]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »$_s$ reg[rs2][4:0]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

Let's Sum an Array

Assume we have an array arr with 10 integers starting at memory address 0x700.

C Program

```
int i = 0;
int arr_sum = 0;
int arr_len = 10;
while (i < arr_len) {
  int val = *(arr + i);
  arr_sum += val;
  i++;
}
```

Assembly

```
addi x5, x0, 0
addi x6, x0, 0
addi x7, x0, 10

add x6, x6, x8
addi x5, x5, 1
```

*What about int val = *(arr + i);??*



RISC-V ISA: Instructions

- Three types of operations:
  - **Computational:** Perform arithmetic and logical operations on registers
  - **Loads and stores:** Move data between registers and main memory
  - **Control Flow:** Change the execution order of instructions
    - To support conditional statements and loops
    - To support function calls (next week)

Accessing Words in Memory

- In 6.190, all RISC-V memory accesses will involve entire 32-bit words.
  - We will read an entire 32-bit word from memory
  - We will store an entire 32-bit word into memory
- However, memory is *byte (8-bit)* addressable.
  - Addresses of adjacent words are 4 apart.



Memory addresses used in load and store instructions must be multiples of 4.

RISC-V Load Instruction

Format: lw rd, offset(rs1)

- Read the 32-bit value stored at the memory address rs1 + offset
  - Base address (rs1) always stored in a register.
  - Offset (offset) is a 12-bit constant.
  - base + offset sum must be a multiple of 4!
- Put that 32-bit value in a register (rd)

```
lw x1, 0x4(x0)      # x1 ← Mem[x0 + 0x4]

lw x2, 0x8(x0)      # x2 ← Mem[x0 + 0x8]
```

RISC-V Store Instruction

Format: sw rs2, offset(rs1)

- Take 32-bit value in register (rs2) and store it at the memory address rs1 + offset.
  - Base address (rs1) always stored in a register.
  - Offset (offset) is a 12-bit constant.
  - base + offset sum must be a multiple of 4!

```
sw x3, 0x10(x0)          # Mem[x0 + 0x10] ← x3
```

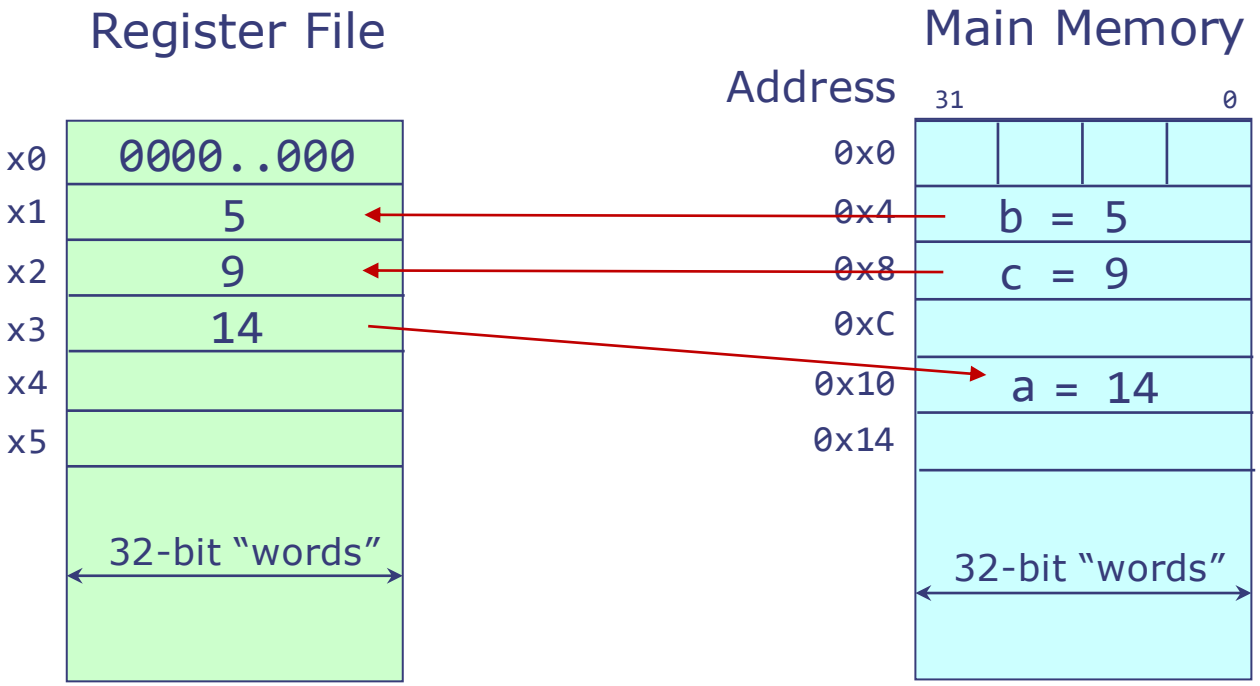
Performing Computations on Values in Memory

a = b + c

Assume:

- a is at address 0x10
- b is at address 0x4
- c is at address 0x8

```
lw x1, 0x4(x0)
lw x2, 0x8(x0)
add x3, x1, x2
sw x3, 0x10(x0)
```



Load and Store (Memory Access) Instructions

Instruction	Syntax	Description	Execution
LW	`lw rd, offset(rs1)`	Load Word	`reg[rd] <= mem[reg[rs1] + offset]`
SW	`sw rs2, offset(rs1)`	Store Word	`mem[reg[rs1] + offset] <= reg[rs2]`

LW and SW are the only instructions that interact with memory!!!

Let's Sum an Array

Assume we have an array arr with 10 integers starting at memory address 0x700.

C Program

```
int i = 0;
int arr_sum = 0;
int arr_len = 10;
while (i < arr_len) {
  int val = *(arr + i);
  arr_sum += val;
  i++;
}
```

Assembly

```
addi x5, x0, 0
addi x6, x0, 0
addi x7, x0, 10

lw x8, 0x700(x5)
add x6, x6, x8
addi x5, x5, 1
```

Is there a problem here?

Let's Sum an Array

Assume we have an array arr with 10 integers starting at memory address 0x700.

C Program

```
int i = 0;
int arr_sum = 0;
int arr_len = 10;
while (i < arr_len) {
  int val = *(arr + i);
  arr_sum += val;
  i++;
}
```

Assembly

```
addi x5, x0, 0
addi x6, x0, 0
addi x7, x0, 40

lw x8, 0x700(x5)
add x6, x6, x8
addi x5, x5, 4
```

What about the while loop?



RISC-V ISA: Instructions

- Three types of operations:
  - **Computational:** Perform arithmetic and logical operations on registers
  - **Loads and stores:** Move data between registers and main memory
  - **Control Flow:** Change the execution order of instructions
    - To support conditional statements and loops
    - To support function calls (next week)

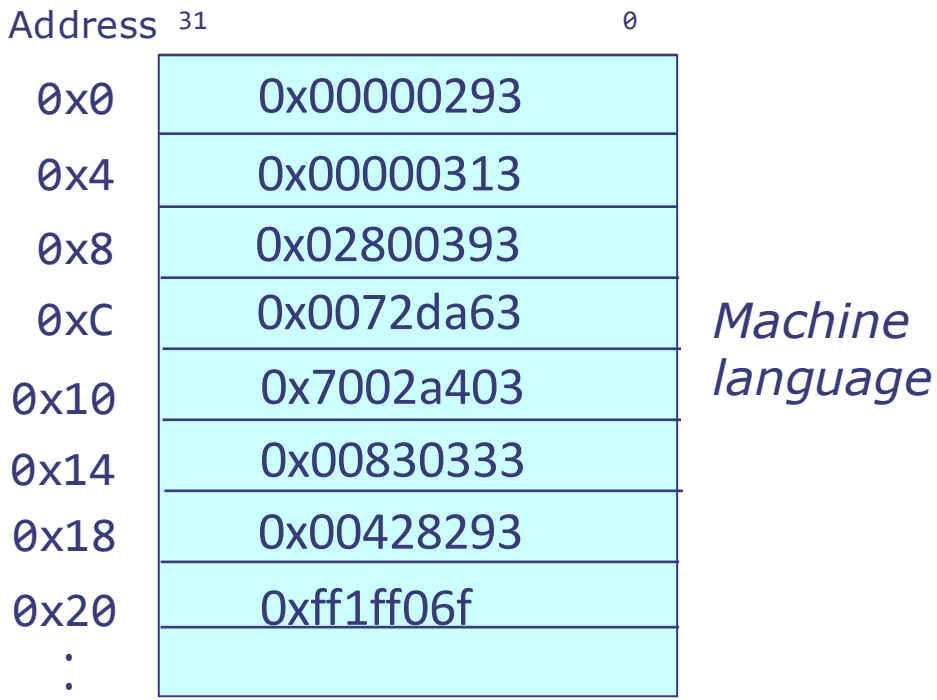
Instructions are Stored in Memory

- Assembly language gets translated into machine language, which is then stored in memory for the processor to read as it executes the program.
  - In RV32I, all instructions are 32 bits. More on this in a few.

Assembly Language

```
   addi x5, x0, 0
   addi x6, x0, 0
   addi x7, x0, 40
loop:
   bge x5, x7, end
   lw x8, 0x700(x5)
   add x6, x6, x8
   addi x5, x5, 4
   jal x0, loop
end:
```

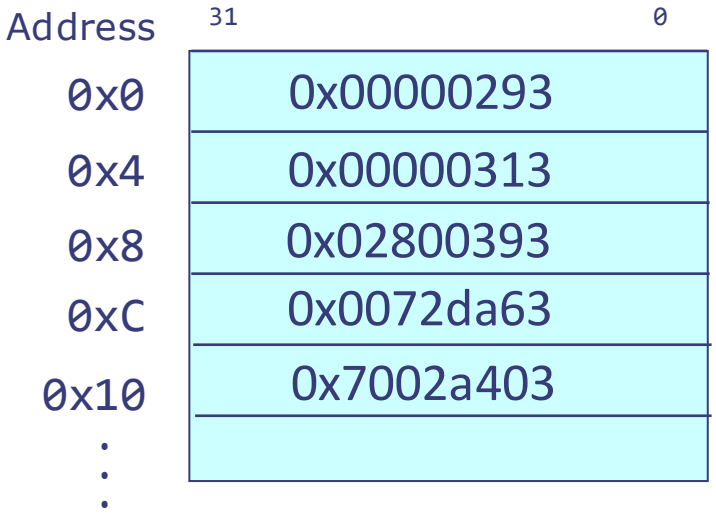
Instruction Memory



How does the computer know where it is in the program?

- A RISC-V processor has a special register called the *program counter (pc)*.
  - Holds the memory address of the current instruction.
  - Processor uses this address to fetch the instruction from memory.
    - Then it decodes and executes it.

Instruction Memory Ex: pc = 0x8



- Processor fetches 32 bits stored at address 0x8
- Processor interprets those 32 bits as an instruction.
- Processor executes instruction.
  - pc is updated

Control Flow Instructions

- An assembly program will always continue to the next instruction (pc = pc + 4) unless otherwise told by a control flow instruction (pc = pc + offset).
  - The offset is the distance between the memory address of the current instruction and the memory address of the instruction the program is jumping to.

- Two main types of control flow instructions
  - Conditional branch: branch (jump) only if a condition is met
  - Unconditional jump: always jump

Allow us to create conditional (if/else) statements and loops (while, for).

Labels

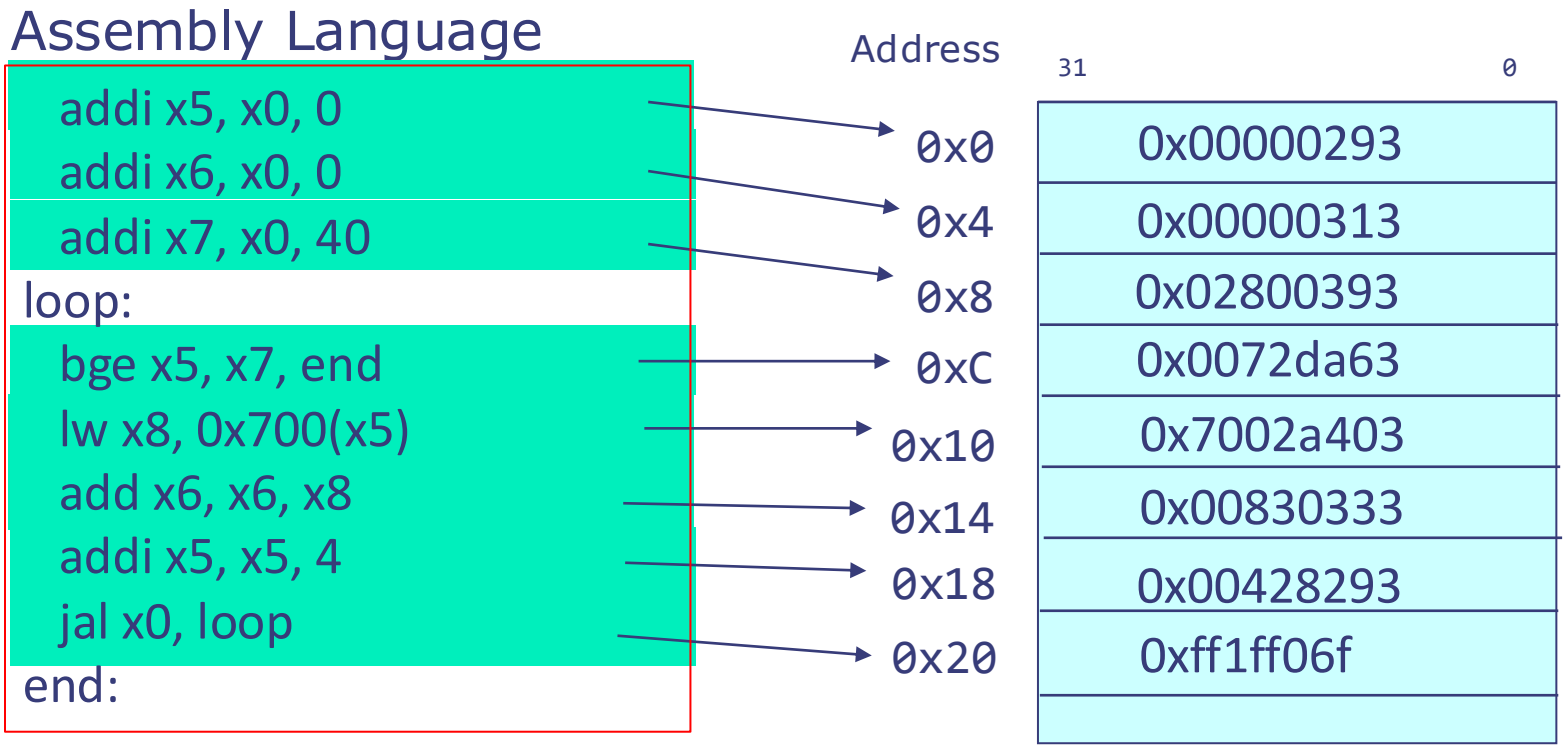
- When writing assembly, you can place "labels" to give convenient names to lines of instructions.
- When you need to branch or jump to another part of your code, you just use the label for that code.

```
some_code: addi x5, x0, 521
  addi x6, x0, 177
some_other_code:
  lui x2, 0x12345
  addi x2, x2, 0x678
```

- `some_code` refers to `addi x5, x0, 521`
- `some_other_code` refers to the instruction immediately below it: `lui x2, 0x12345`

Labels take up no space in memory

- A label has no size (takes up no space in memory).



Conditional Branch Instructions

Format: comp rs1, rs2, label

- First performs comparison to determine if branch is taken or not: rs1 *comp* rs2
- If comparison returns True, then branch is taken (jump to label), else continue executing program in order.
  - Updates program counter (pc) accordingly.

Instruction	beq	bne	blt	bge	bltu	bgeu
comp	==	!=	<	≥	<	≥

- beq x1, x2, label

```
# If x1 == x2, jump to
# label. Otherwise, go to
# next instruction.
```

Conditional Branch Example

C Code:

```
if (a < b) {
   c = a + 1;
} else {
   c = b + 2;
}
```

Assume:

x1 = a
x2 = b
x3 = c

Assembly Code:

```
        bge x1, x2, else
        addi x3, x1, 1
        beq x0, x0, end
else:   addi x3, x2, 2
 end:
```

Unconditional Branch Instructions

jal: Unconditional jump and link

Format: jal rd, label

- Jump to instruction following label
  - program counter (pc) updates to be address of that instruction
- Link (next lecture): stored in register rd

```
jal x3, somewhere        # jump to instruction indicated by
                         # label somewhere, store link in x3
```

Unconditional Branch Instructions

jalr: Unconditional jump via register and link

Format: jalr rd, offset(rs1)

- Jump to instruction specified by rs1 + offset
  - program counter (pc) updates to be address of that instruction
  - rs1 is a register, offset is a constant
- Link (next lecture): stored in register rd
- Can jump to any 32 bit address – supports long jumps
  - Instructions are all 32 bits, so this address must be a multiple of 4
  - jalr x3, 4(x1) # jump to instruction indicated by # x1 + 4, store link in x3

Control Flow Instructions

Instruction	Syntax	Description	Execution
JAL	`jal rd, label`	Jump and Link	`reg[rd] <= pc + 4` `pc <= label`
JALR	`jalr rd, offset(rs1)`	Jump and Link Register	`reg[rd] <= pc + 4` `pc <= {(reg[rs1] + offset)[31:1], 1'b0}`
BEQ	`beq rs1, rs2, label`	Branch if $=$	`pc <= (reg[rs1] == reg[rs2]) ? label: pc + 4`
BNE	`bne rs1, rs2, label`	Branch if $\neq$	`pc <= (reg[rs1] != reg[rs2]) ? label: pc + 4`
BLT	`blt rs1, rs2, label`	Branch if $<$ (Signed)	`pc <= (reg[rs1]` $<_s$ `reg[rs2]) ? label: pc + 4`
BGE	`bge rs1, rs2, label`	Branch if $\geq$ (Signed)	`pc <= (reg[rs1]` $>=_s$ `reg[rs2]) ? label: pc + 4`
BLTU	`bltu rs1, rs2, label`	Branch if $<$ (Unsigned)	`pc <= (reg[rs1]` $<_u$ `reg[rs2]) ? label: pc + 4`
BGEU	`bgeu rs1, rs2, label`	Branch if $\geq$ (Unsigned)	`pc <= (reg[rs1]` $>=_u$ `reg[rs2]) ? label: pc + 4`

Let's Sum an Array

Assume we have an array arr with 10 integers starting at memory address 0x700.

C Program

```
int i = 0;
int arr_sum = 0;
int arr_len = 10;
while (i < arr_len) {
  int val = *(arr + i);
  arr_sum += val;
  i++;
}
```

Assembly

```
  addi x5, x0, 0
  addi x6, x0, 0
  addi x7, x0, 40
loop:
  bge x5, x7, end
  lw x8, 0x700(x5)
  add x6, x6, x8
  addi x5, x5, 4
  jal x0, loop
end:
```

Nice



Instruction Encoding

- Machine language is the binary encoding of assembly instructions.
  - Binary encoding – just like integers, floats, other data types! Everything is bits.
- All RV32I instructions are four bytes (32 bits)

- How do we translate between assembly and instruction encodings?

RISC-V ISA tells us exactly how!

Instruction Encoding

- Six main types of instruction encodings:

MIT 6.190 ISA Reference Card: Instruction Encodings

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

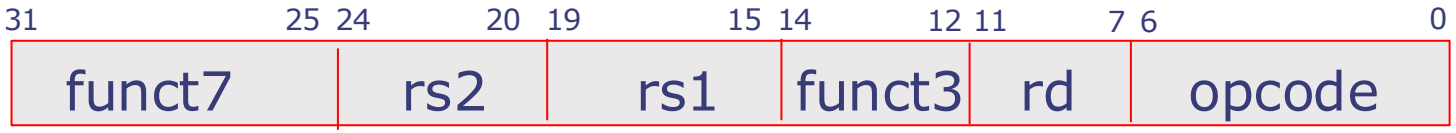
Instruction Encoding

RV32I Base Instruction Set (MIT 6.190 subset)

imm[31:12]				rd	0110111	LUI
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Instruction Encoding

R-type: Register-Register Instruction Format



Some parts are found in the ISA:

- funct7, funct3: encodes the function (add, and, etc.)
- opcode: encodes the instruction type (register-register)

Some parts come from the instruction:

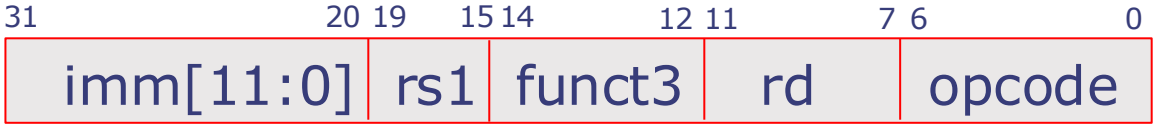
- rs2: source register 2 (5 bits)
- rs1: source register 1 (5 bits)
- rd: destination register (5 bits)

use ISA to determine which register is which!

Why do we use 5 bits for the registers?

Instruction Encoding

I-type: Register-Immediate Instruction Format



Some parts are found in the ISA:

- funct3: encodes the function (addi, andi, etc.)
- opcode: encodes the instruction type (register-immediate)

Some parts come from the instruction:

- imm[11:0]: immediate (12 bits)
- rs1: source register 1 (5 bits)
- rd: destination register (5 bits)

Why do we use 12 bits for the immediate?

So you're walking down the street…

- …and you come across this in memory:

Address:	Data:
0x10FA0	0b00100000100100000000001010010011
0x10FA4	0b00001011000100000000001100010011
0x10FA8	0b00000000010100110100001010110011

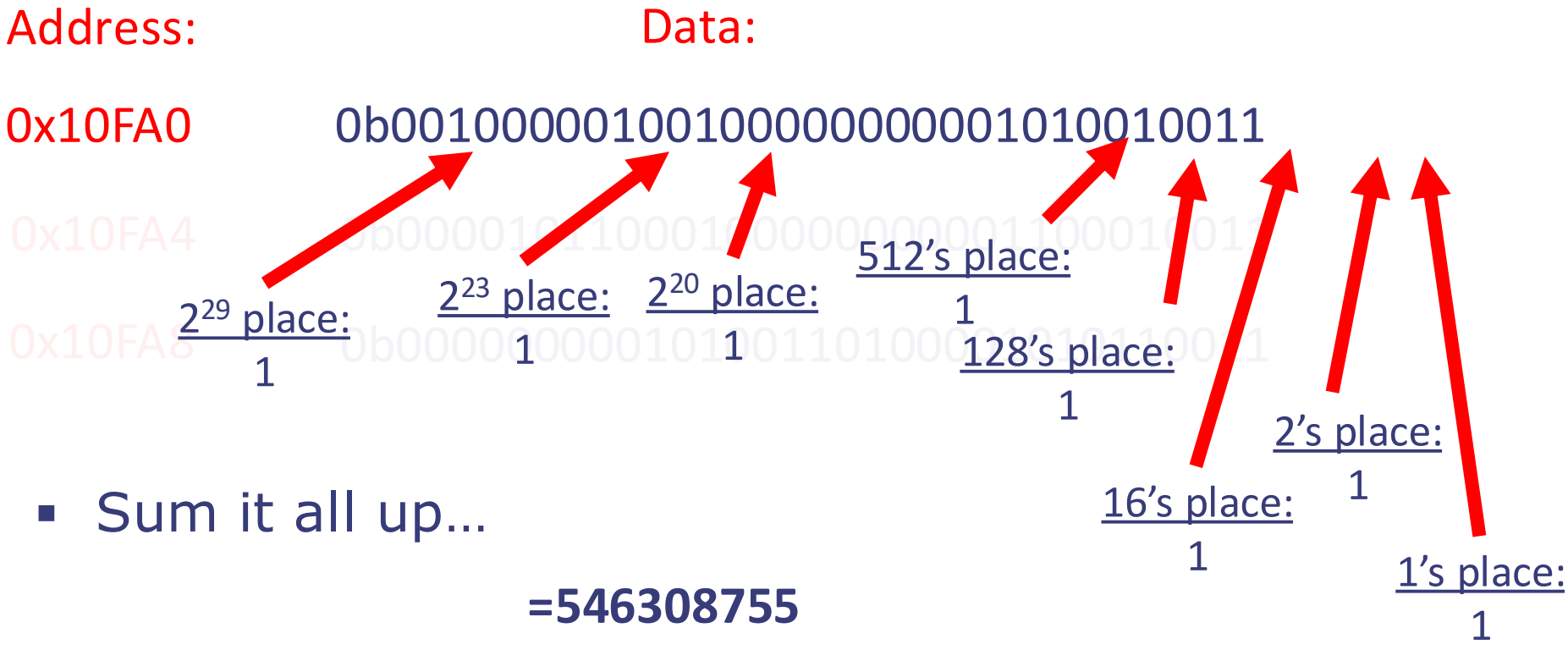
- What does it mean?

  Depends on how we're supposed to interpret it!



If we're in a Data Region of Memory

- Maybe the value at 0x10FA0 is an unsigned int:

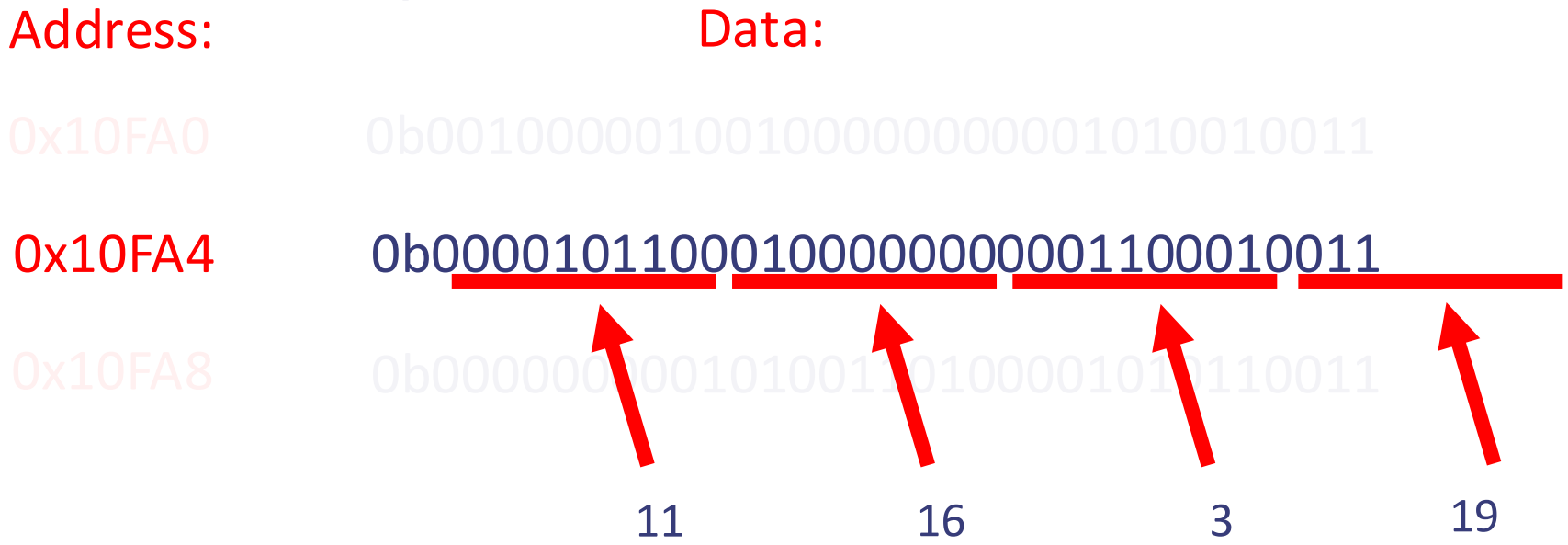


- Sum it all up…

=546308755

If we're in a Data Region of Memory

- Maybe the four bytes at 0x10FA4 are four uint8_ts in an array!!!!



={19,3,16,11};

If we're in a Data Region of Memory

- Maybe the four bytes at 0x10FA8 is a float!

Address:	Data:
0x10FA0	0b00100000100100000000001010010011
0x10FA4	0b00001011000100000000000110010011
0x10FA8	0b00000000010100110100001010110011

https://www.h-schmidt.net/FloatConverter/IEEE754.html

$$= 7.64627335239e \times 10^{-39}$$

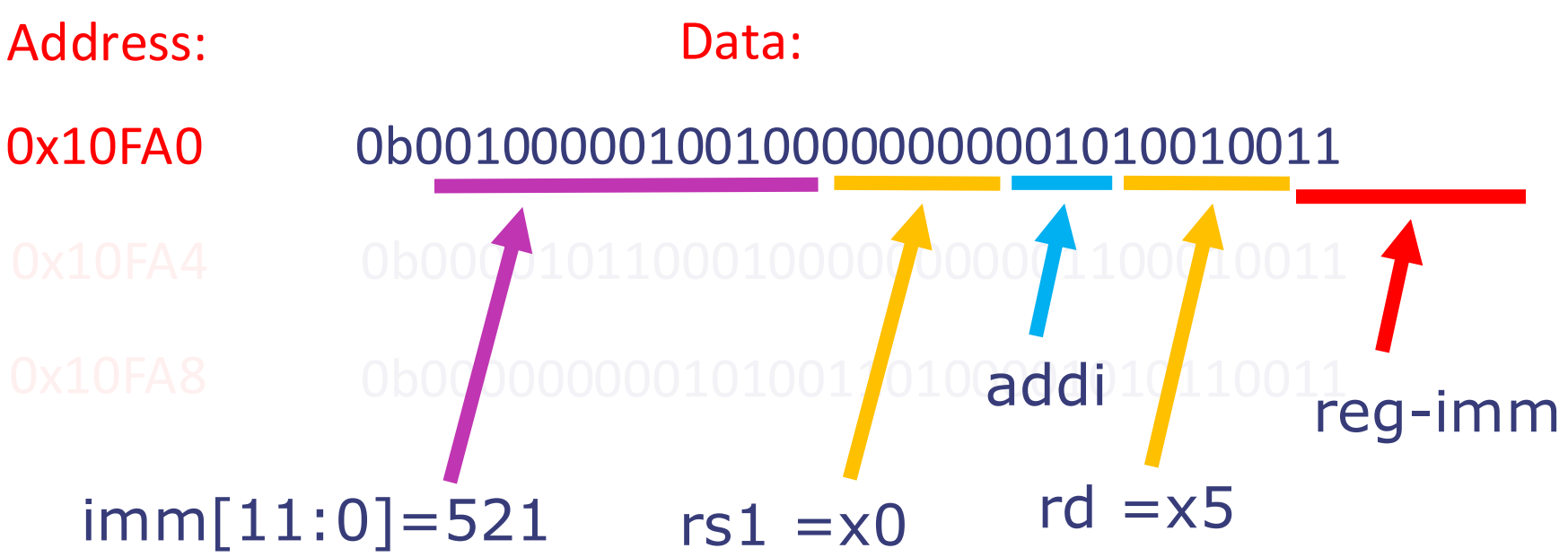
If we're where the program is stored, these are INSTRUCTIONS!

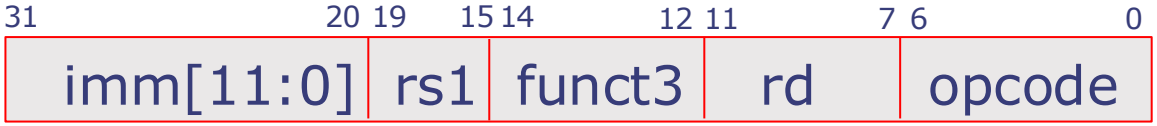
- And we *must* interpret them as instructions

Address:	Data:
0x10FA0	0b00100000100100000000001010010011
0x10FA4	0b00001011000100000000001100010011
0x10FA8	0b00000000010100110100001010110011

If we're where the program is stored, these are INSTRUCTIONS!

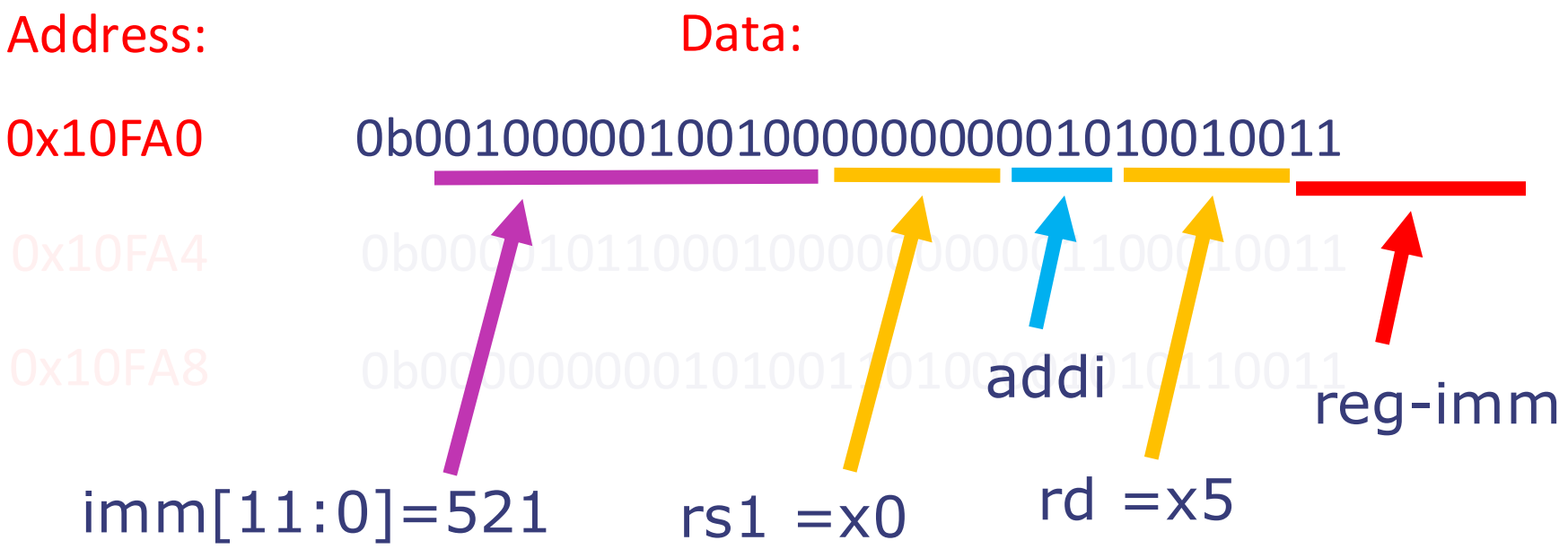
- Instruction at 0x10FA0?





If we're where the program is stored, these are INSTRUCTIONS!

- Instruction at 0x10FA0?



addi x5, x0, 521

x0 always 0 so this is saying "put 521 into x5"

If we're where the program is stored, these are INSTRUCTIONS!

- Instruction at 0x10FA4?

Address: Data:

0x10FA0 0b00100000100100000000001010010011

0x10FA4 0b00001011000100000000001100010011

0x10FA8 0b00000000010100110100001010110011

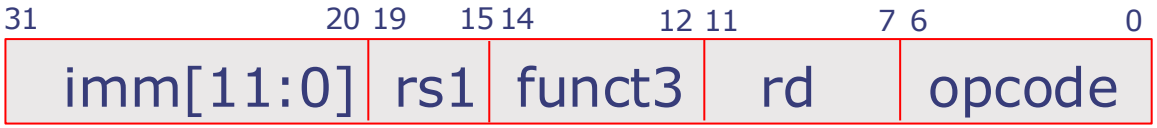
imm[11:0]=177

rs1 =x0

addi

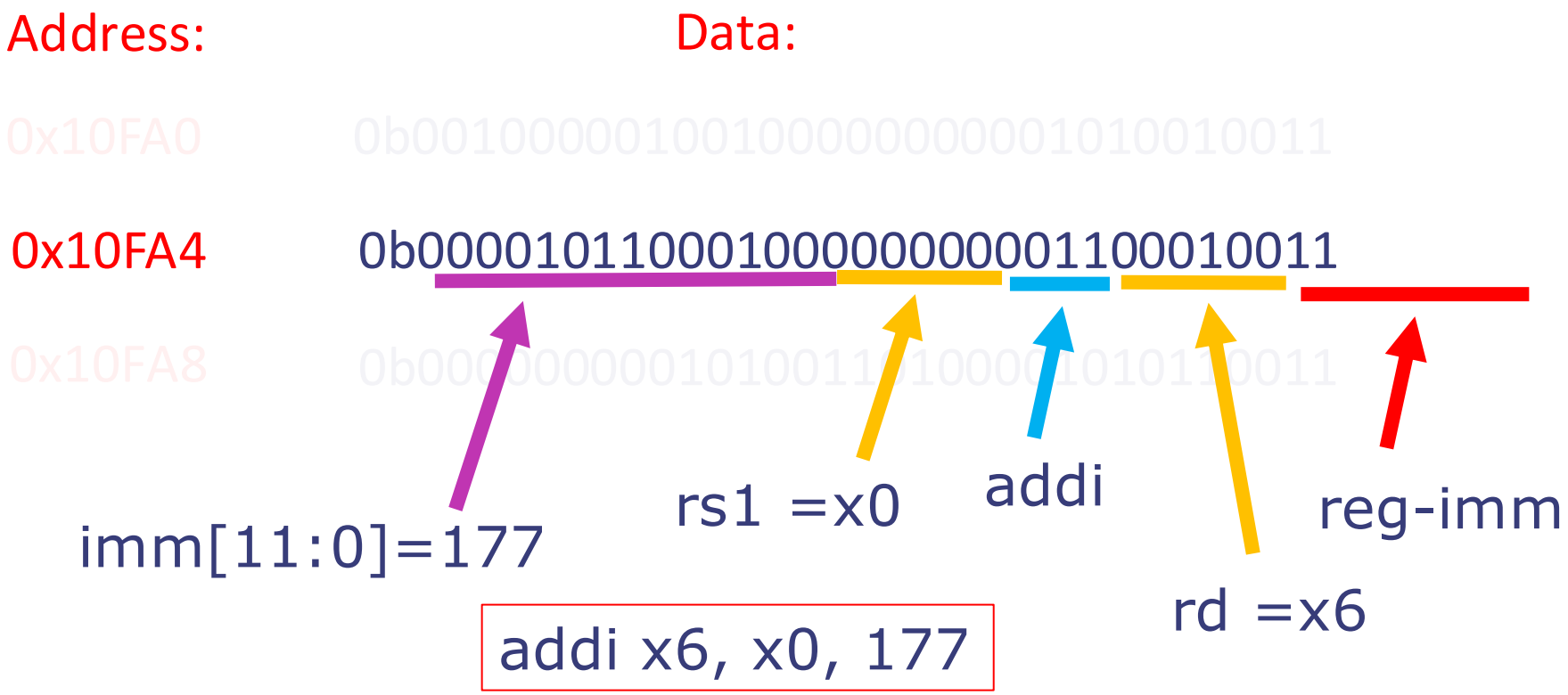
rd =x6

reg-imm



If we're where the program is stored, these are INSTRUCTIONS!

- Instruction at 0x10FA4?



x0 always 0 so this is saying "put 177 into x6"

If we're where the program is stored, these are INSTRUCTIONS!

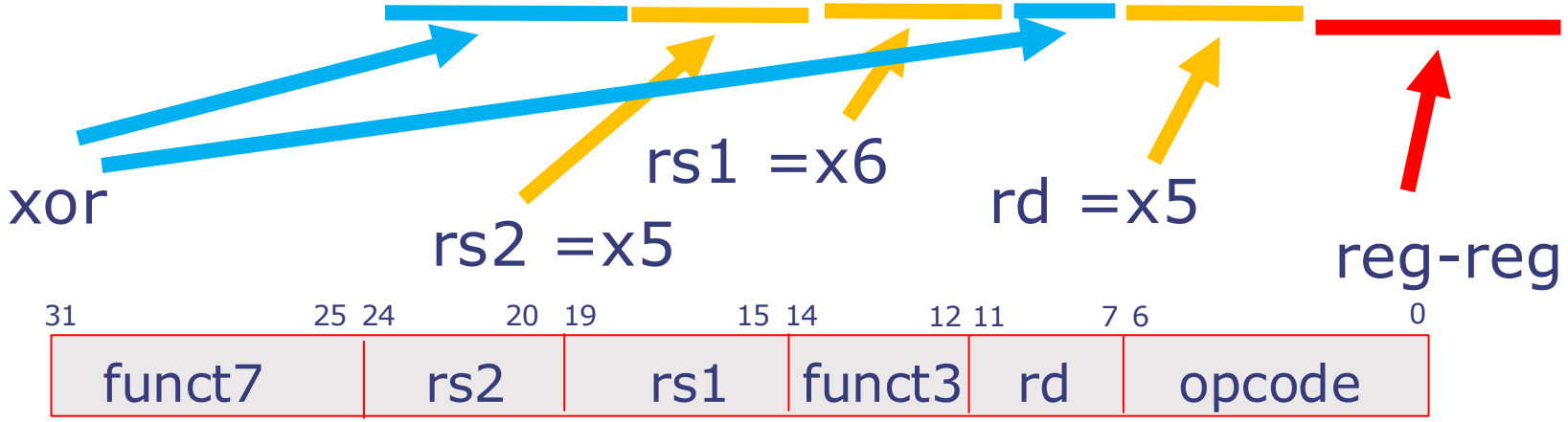
- Instruction at 0x10FA8?

Address: Data:

0x10FA0 0b00100000100100000000001010010011

0x10FA4 0b00001011000100000000000110010011

0x10FA8 0b00000000010100110100001010110011



If we're where the program is stored, these are INSTRUCTIONS!

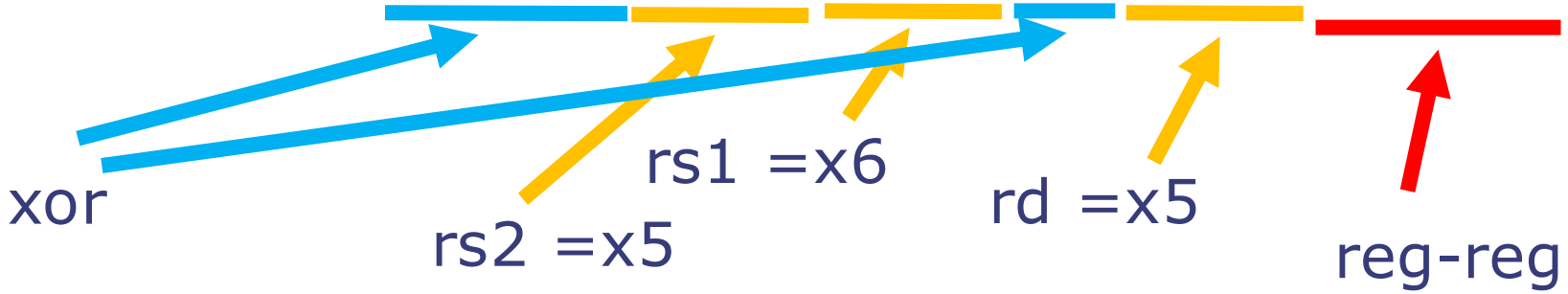
- Instruction at 0x10FA8?

Address: Data:

0x10FA0 0b00100000100100000000001010010011

0x10FA4 0b00001011000100000000000110010011

0x10FA8 0b00000000010100110100001010110011



xor x5, x6, x5

Bitwise xor x6 with x5 and put back into x5

If we're where the program is stored, these are INSTRUCTIONS!

- So this would get interpreted by the computer as:

Address:	Data:
0x10FA0	0b00100000100100000000001010010011
0x10FA4	0b00001011000100000000001100010011
0x10FA8	0b00000000010100110100001010110011

```
addi x5, x0, 521
addi x6, x0, 177
xor x5, x6, x5
```

Data Memory vs. Instruction Memory

- Data will live in one part of the memory.
- The program (instructions) will live in another part of the memory
- Instruction memory is usually write-protected (don’t want to change program during run-time)

Pseudoinstructions

- Not "real"
- While writing assembly, these can save a few keystrokes or lines – makes life a bit easier:

MIT 6.190 ISA Reference Card: Pseudoinstructions

Pseudoinstruction	Description	Execution
`li rd, liConstant`	Load Immediate	`reg[rd] <= liConstant`
`mv rd, rs1`	Move	`reg[rd] <= reg[rs1] + 0`
`not rd, rs1`	Logical Not	`reg[rd] <= reg[rs1] ^ -1`
`neg rd, rs1`	Arithmetic Negation	`reg[rd] <= 0 - reg[rs1]`
`j label`	Jump	`pc <= label`
`jal label` `call label`	Jump and Link (with `ra`)	`reg[ra] <=`
`jr rs1`	Jump Register	
`jalr rs1`	Jump and Link Register	
`ret`	Ret	`eg[ra]`
`bgt rs1, rs2, label`		`pc <= (reg[rs1] >s reg[rs2]) ? label : pc + 4`
`ble rs1, rs2, `		`pc <= (reg[rs1] <=s reg[rs2]) ? label : pc + 4`
`bgtu`	`gned)`	`pc <= (reg[rs1] >s reg[rs2]) ? label : pc + 4`
`bleu`	$\leq$ (Unsigned)	`pc <= (reg[rs1] <=s reg[rs2]) ? label : pc + 4`
`beqz r`	Branch $= 0$	`pc <= (reg[rs1] == 0) ? label : pc + 4`
`bnez rs`	Branch $\neq 0$	`pc <= (reg[rs1] != 0) ? label : pc + 4`
`bltz rs1, label`	Branch < 0 (Signed)	`pc <= (reg[rs1] <s 0) ? label : pc + 4`
`bgez rs1, label`	Branch ≥ 0 (Signed)	`pc <= (reg[rs1] >=s 0) ? label : pc + 4`
`bgtz rs1, label`	Branch > 0 (Signed)	`pc <= (reg[rs1] >s 0) ? label : pc + 4`
`blez rs1, label`	Branch ≤ 0 (Signed)	`pc <= (reg[rs1] <=s 0) ? label : pc + 4`

CAN'T USE THIS WEEK

Registers and their Names

- The registers all have "symbolic names" which more accurately reflect their expected usage and some "conventions" we must adhere to when using them
  - **Instruction encodings (machine code) always use the x-register**

Registers	Symbolic names	Description	Saver
x0	zero	Hardwired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporary registers	Caller
x8-x9	s0-s1	Saved registers	Callee
x10-x11	a0-a1	Function arguments and return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporary registers	Caller

- For week 4, we'll tell you what registers to use

More on this in week 5!

Next lecture:
Implementing Procedures in Assembly