

9/2/26

COMP 311:

Computer Organization!

Lecture 5: Branch + Jump Instructions



The University
of North Carolina
at Chapel Hill

<https://tinyurl.com/comp311-fa26>

Plan for Class Today



- 3:35-4:10: Quiz!
- 4:10-4:15: BREAK
- 4:20-4:35: Finish tracing problem from last time!
- 4:35-4:50: The Program Counter, Data and Control Flow

Guiding Question 😊

How does the machine choose which instruction to execute next?



UNIT 1: PROGRAMMING THE MACHINE

Accessing Memory

Memory



- The register file only contains 32, 32-bit registers
 - There be other registers in the hardware, i.e., pipeline registers, but the programmer doesn't have access to these!
- We can't store all of our data in the register file. So where does it all go?
- ***In the main memory***

RISC-V Memory Model



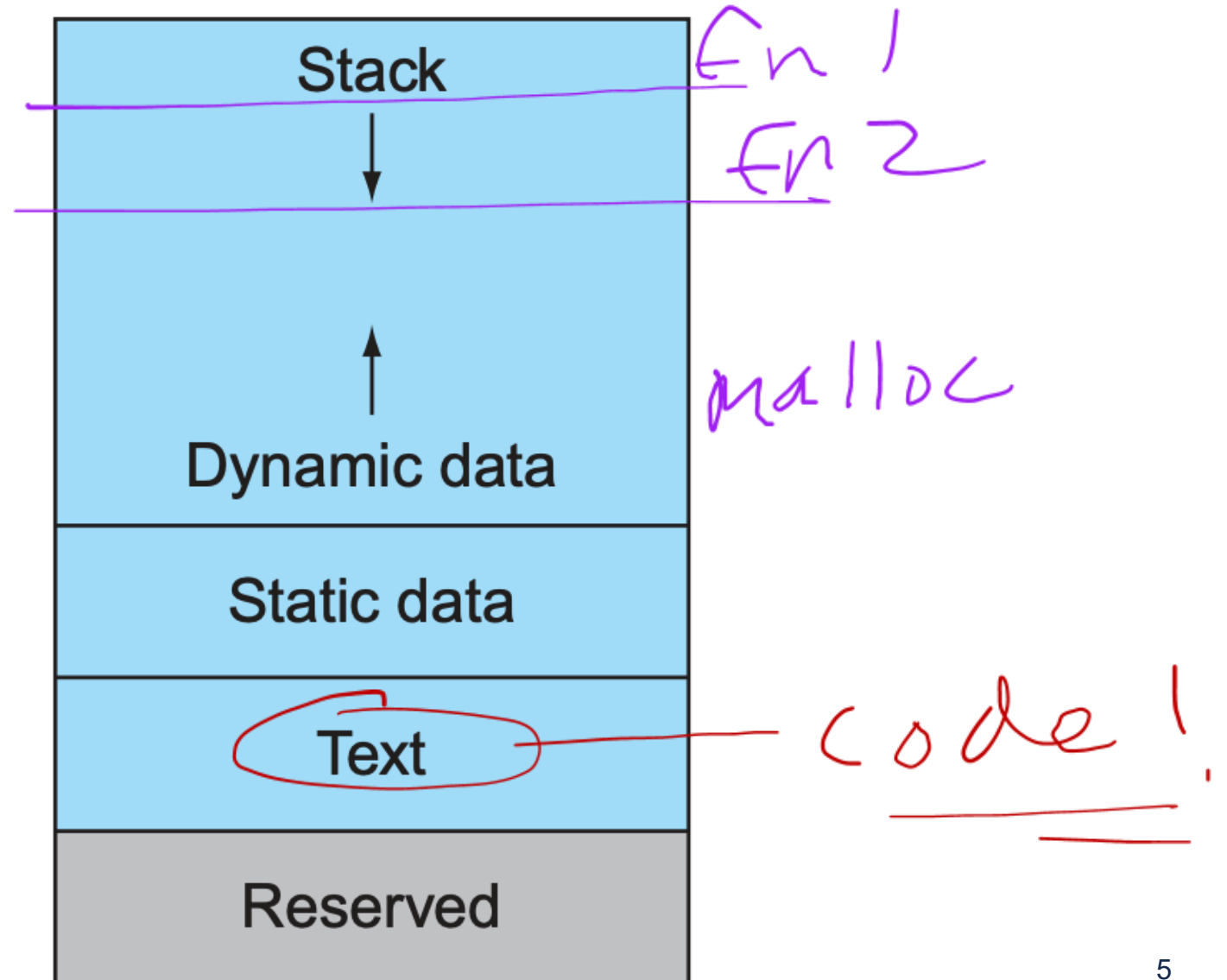
Used to manage function calls and local variables

Used for dynamic memory allocation

Variables allocated at compile-time

Programs

Reserved for the OS





Memory vs. Register File

- So what's the point of the register file?
- It is MUCH, much..! faster to access the register file
- We'll store the values that we are currently working with in the register file

Memory

1. large
2. takes longer to access
3. not part of the CPU

Register File

1. small
2. quick access
3. part of the CPU

Additional Simulator Notes: Symbol Table



UNC miniRISC-V Architecture Simulator V 1.0

Symbol Table

```
reset:  lui    sp,0xc0000    # initilize stack pointer
        addi   sp,sp,0xff0
        jal    ra,main      # call main
*halt:  j      halt
```

#####

```
main:   addi    x31,x0,10
        addi    x5,x0,0
        lw      x6,values(x6)
        add     x7,x7,x6
        sw      x7,sum(x0)
        addi    x5,x5,1
        blt     x5,x31,loop
*halt:  jal     x0,halt

sum:    .word    0
values: .word    1,3,5,7,9,11,13,15,17,19
```

Symbol

reset
halt
main
loop
sum
values

Location

0x00000000 (0)
0x00000034 (52)
0x00000010 (16)
0x00000018 (24)
0x00000038 (56)
0x0000003C (60)

Assemble Reset Step Multistep 10 Run Memory Dump 0x00000000 Advanced

Let's us know the locations in memory of different labels, data structures, references

Additional Simulator Notes: Memory Dump



Memory Dump

UNC miniRISC-V Architecture Simulator V 1.0

```
reset: lui      sp,0xc0000    # initilize stack pointer
      addi     sp,sp,0xff0
      jal      ra,main       # call main
*halt: j        halt

#####

main: addi     x31,x0,10
      addi     x5,x0,0
loop: slli     x6,x5,2
      lw       x6,values(x6)
      lw       x7,sum(x0)
      add      x7,x7,x6
      sw       x7,sum(x0)
      addi     x5,x5,1
      blt      x5,x31,loop
*halt: jal      x0,halt

sum:   .word   0
values: .word   1,3,5,7,9,11,13,15,17,19
```

Assemble Reset Step Multistep 10 Run Memory Dump 0x0000003C Advanced

Memory Dump: 0x0000003C = 60

Address	Contents	
0x0000003C	0x00000001	1
0x00000040	0x00000003	3
0x00000044	0x00000005	5
0x00000048	0x00000007	7
0x0000004C	0x00000009	9
0x00000050	0x0000000B	11
0x00000054	0x0000000D	13
0x00000058	0x0000000F	15
0x0000005C	0x00000011	17
0x00000060	0x00000013	19
0x00000064	0x00000000	0
0x00000068	0x00000000	0
0x0000006C	0x00000000	0
0x00000070	0x00000000	0
0x00000074	0x00000000	0
0x00000078	0x00000000	0

Read contents (in hex) of specific memory locations (potentially after running)

Load and Stores in action

Array access in C → in RISC-V:
compute addr, load value, then
use it!



An example of how loads and stores are used to access arrays. What C code produced this?

Hints:

- sizeof(int) in C is 4 bytes.
- What does slli x6, x5, 2 do to the value in x5?
Why might that be useful before a load?
- Instructions that help us with the loop pattern?
- Which instructions access memory?
- What does the add x7, x7, x6 suggest is happening repeatedly?
- Use the simulator! Use the docs!

Assembly:

```
                addi    x31, x0, 10
                addi    x5, x0, 0
loop:           slli    x6, x5, 2
                lw      x6, values(x6)
                lw      x7, sum(x0)
                add     x7, x7, x6
                sw      x7, sum(x0)
                addi    x5, x5, 1
                blt     x5, x31, loop
*halt:         jal     x0, halt
sum:           .word   0
values:        .word   1, 3, 5, 7, 9, 11, 13, 15, 17, 19
```



Load Address

- How do you get the address of the array into a register?
 - with an instruction called load address (la)
- `la rd, Label`
 - places the address of ***Label*** in rd
- This is actually a ***pseudoinstruction***

A pseudoinstruction is a “fake” assembly instruction that doesn’t exist in HW, but assembler expands into one or more real RISC-V instructions!

Loads the address of the label into *Reg[rd]*. It is typically implemented using the two instruction sequence:

```
auipc rd, label
addi rd, rd, label
```

Load Address



- How do you get the address of the array into a register?
 - with an instruction called load address (la)
- la rd Label
 - places the address of ***Label*** in rd
- This is actually a ***pseudoinstruction***

→ AUIPC is an instruction we will talk about more in a bit... it adds an immediate to the PC. Let's review what the PC is / related to program execution.

A pseudoinstruction is a “fake” assembly instruction that doesn't exist in HW, but assembler expands into one or more real RISC-V instructions!

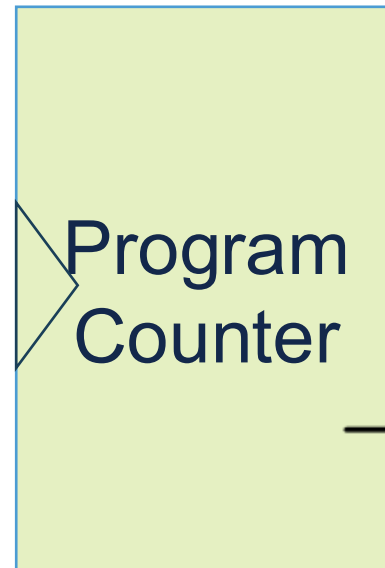
Loads the address of the label into *Reg[rd]*. It is typically implemented using the two instruction sequence:

```
auipc rd, label
addi rd, rd, label
```

Program Counter (PC)



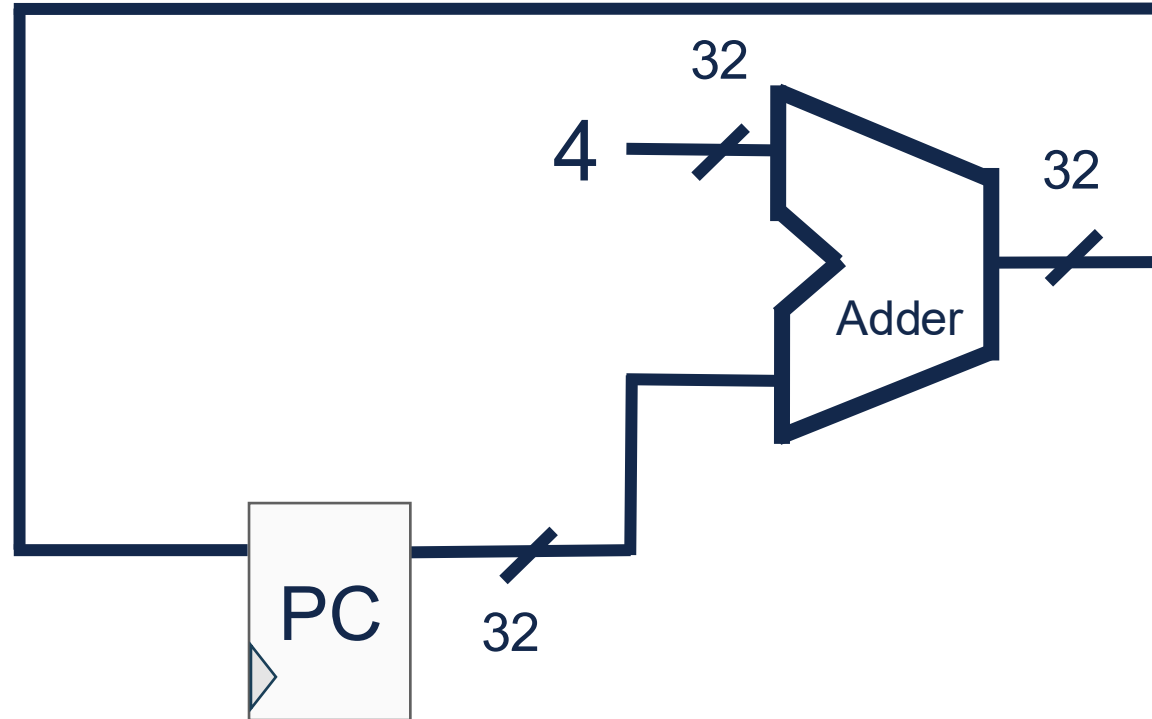
- The Program Counter (PC) is a 32-bit register
- The PC is **not inside of the register file!!!**
- The datapath executes one instruction per clock cycle
- On every clock cycle of processor execution, PC is incremented by 4 to fetch the next instruction



32-bit →
ISA

PC →

Program Counter (PC)

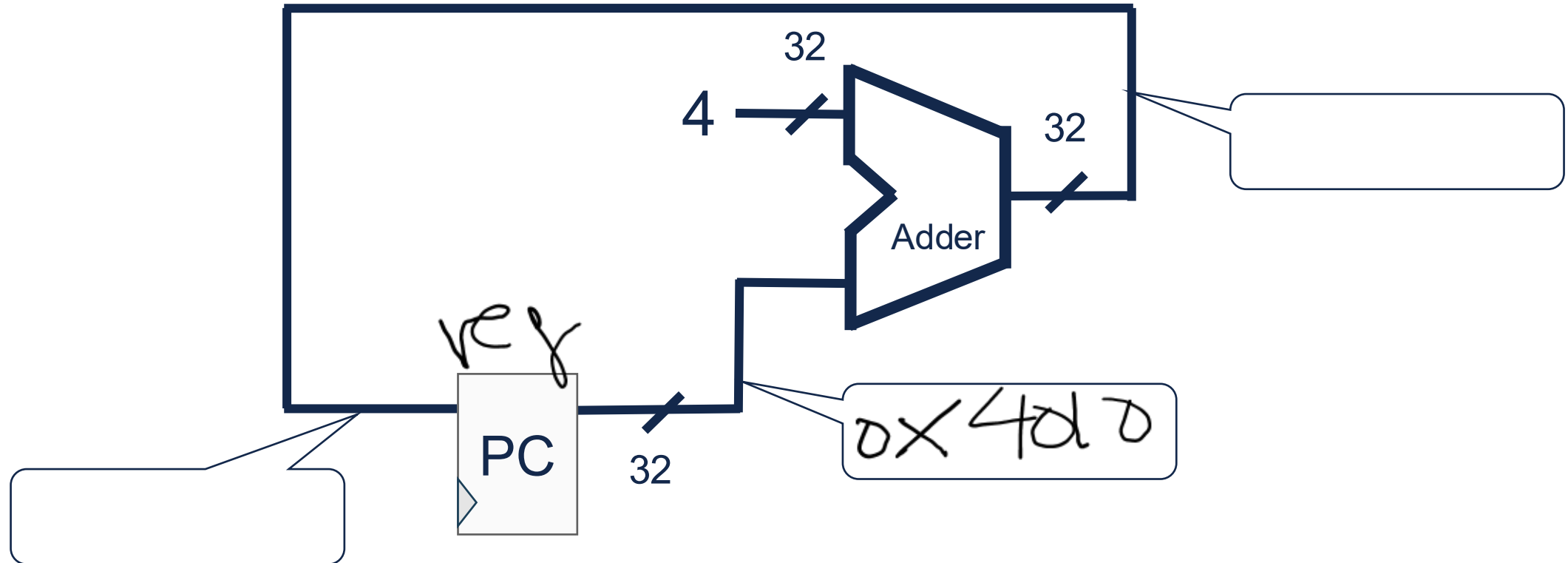


Increment the PC by 4 on every step of CPU execution

Program Counter (PC)



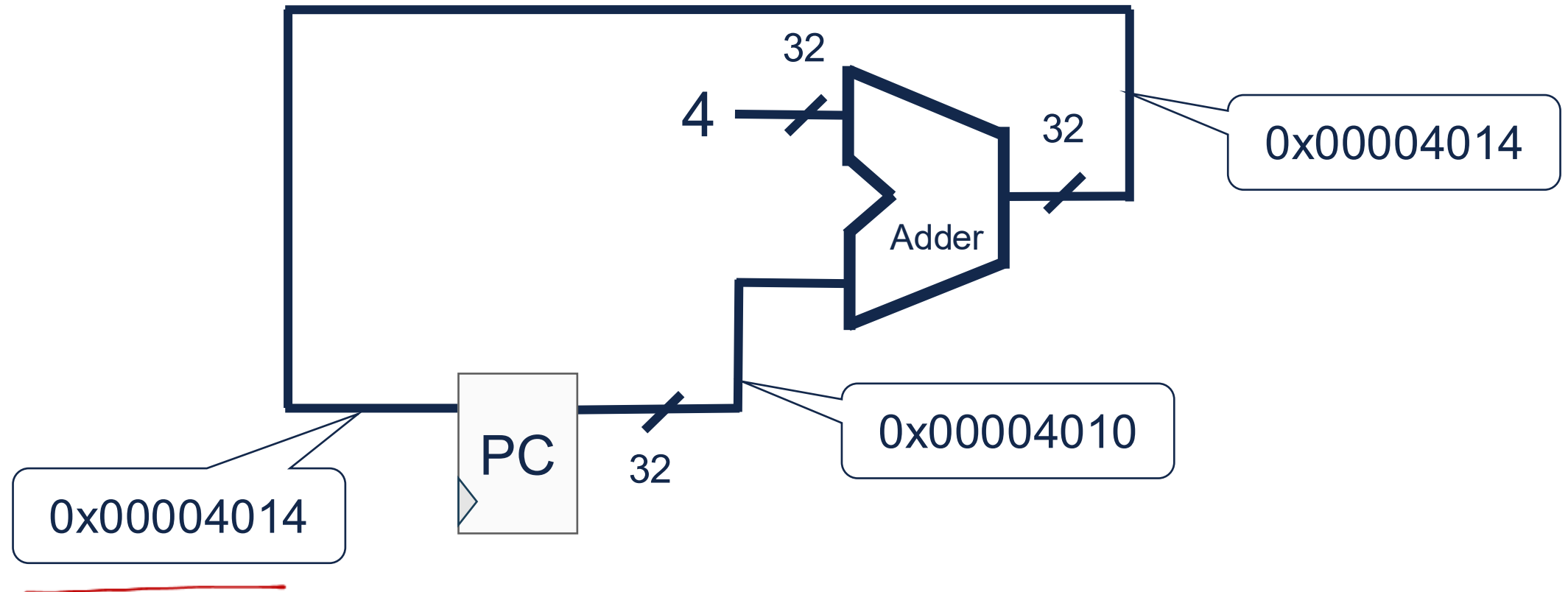
Example: Current PC = 0x00004010



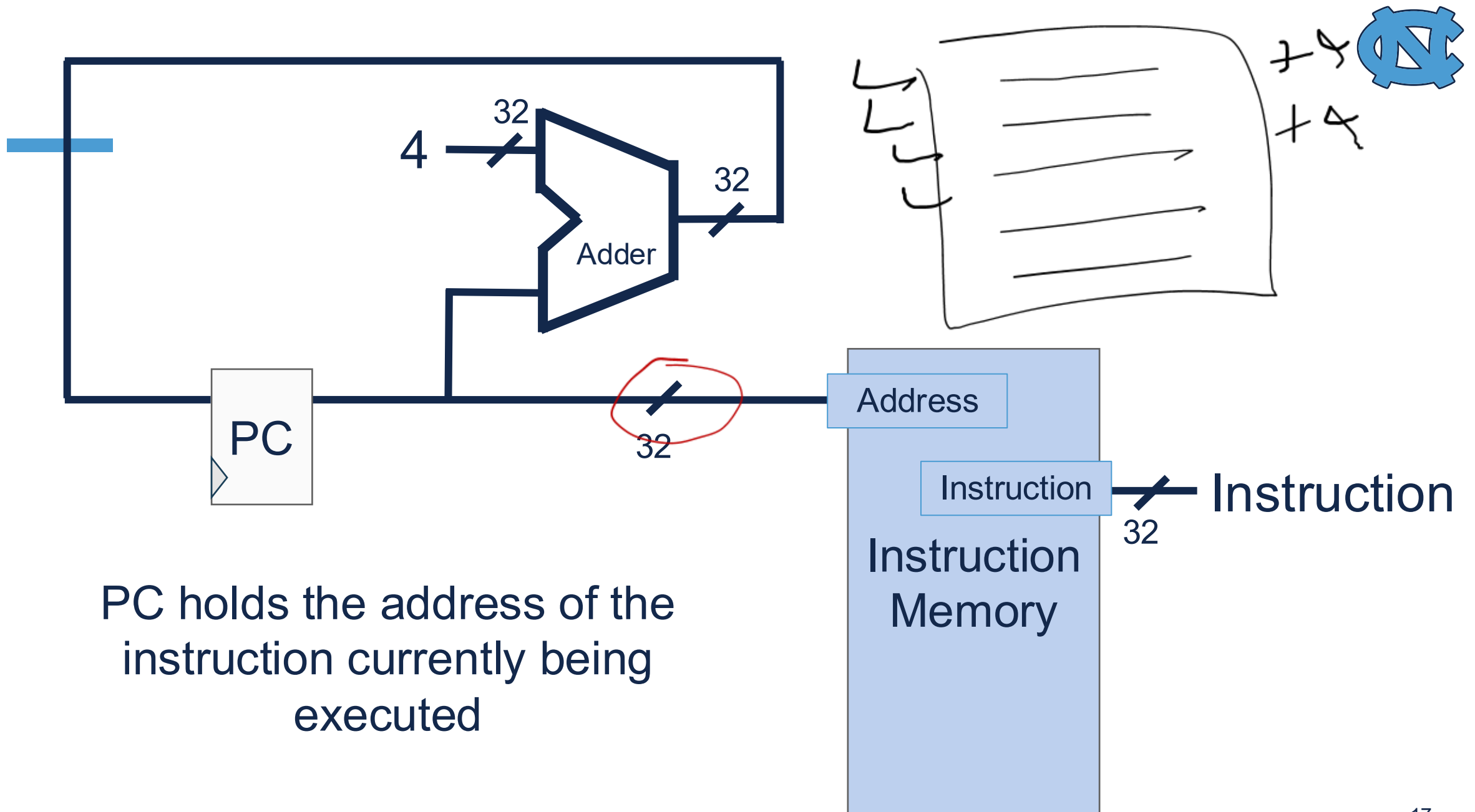
Program Counter (PC)



Example: Current PC = 0x00004010



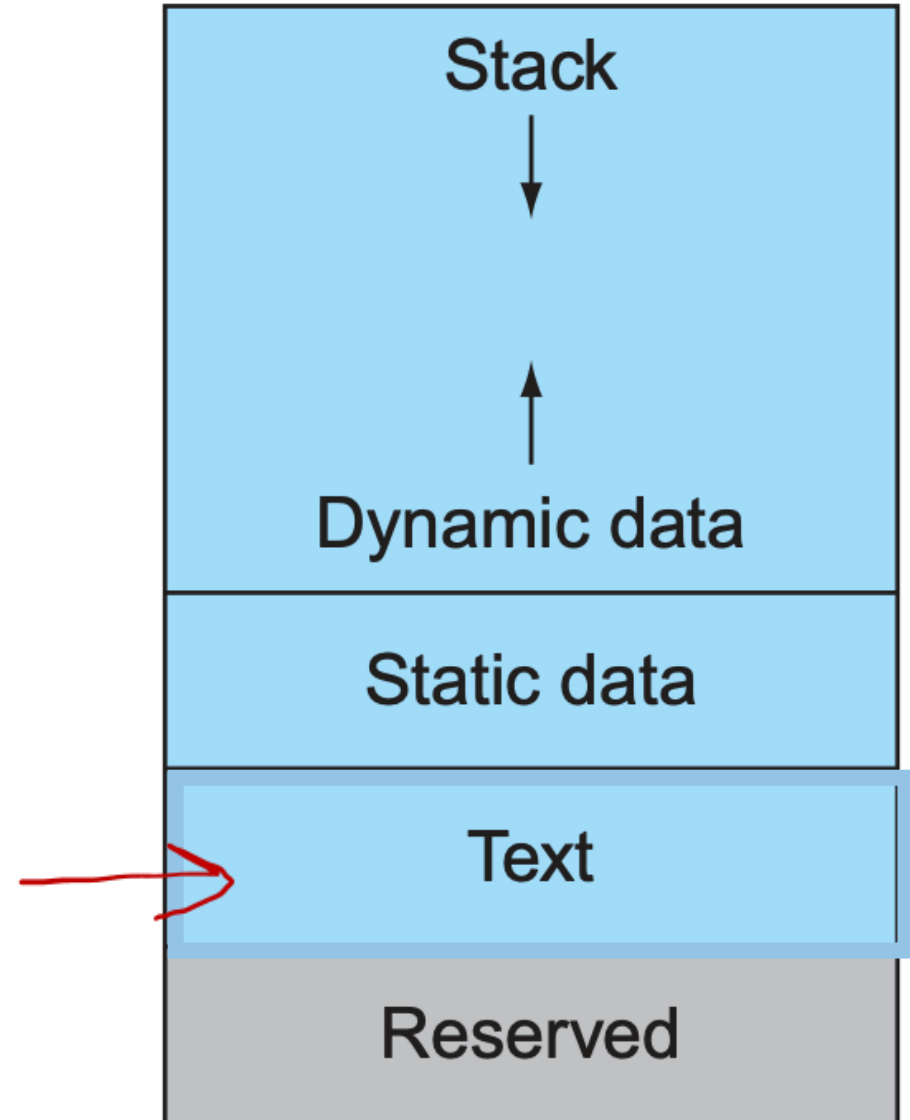
With the next step of execution, PC will become 0x00004014



Storing Code in Memory



- Code is stored in the text segment
- To store code in the text segment, we use the `.text` directive





UNIT 1: PROGRAMMING THE MACHINE

Branches and Jumps



What instructions are left?

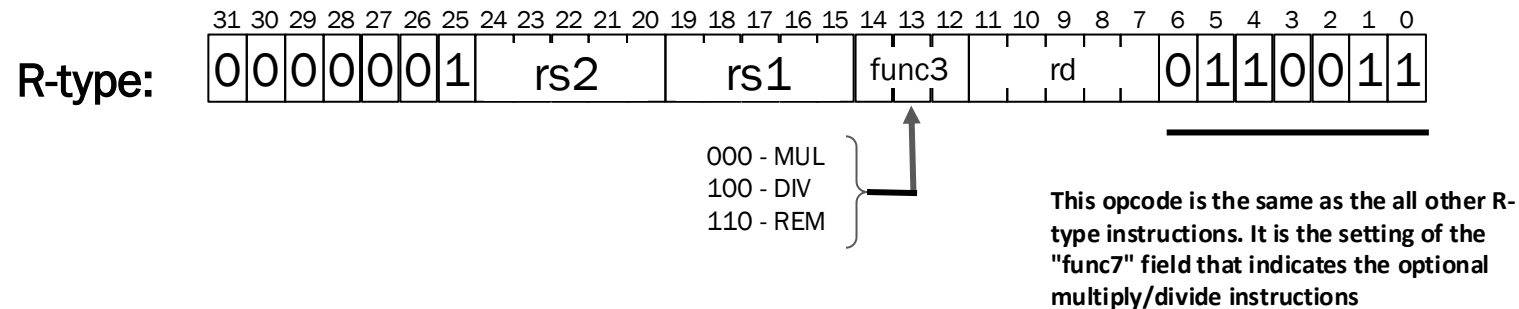
- So far, we have studied instructions that let us...
 - Do basic math!
 - Logical operations
 - Bitwise
 - Read memory
 - Write memory

Missing Math instructions



The RISC-V ISA includes integer multiplication and division as an extension to the RV32I minimal ISA called RV32M.

This is because multiply and divide require significant additional H/W. These instructions can always be emulated, and cost is a consideration for embedded systems. Our miniRISCV simulator includes a subset of RV32M, the subset necessary to implement C.



Multiply



MUL: multiply registers

Syntax: **mul *rd,rs,rt***

Encoding: **0000 001*t* *tttt* *ssss* *s000* *dddd* *d011* 0011**

Description:

$\text{Reg}[d] \leftarrow \text{Reg}[s] * \text{Reg}[t]$

Multiply the contents of *Reg[s]* and *Reg[t]*, and place the lower 32-bits of the product in *Reg[d]*. **Overflows are ignored.** MUL is binary and semantically compliant with the RV32M mul instruction.

Example: **mul x5, x6, x7**

Encoded as: 0x027302B3

Divide



DIV: divide registers

Syntax: **div *rd,rs,rt***

Encoding: **0000 001*t* *tttt* *ssss* *s100* *dddd* *d011* 0011**

Description:

$\text{Reg}[d] \leftarrow \text{Reg}[s] / \text{Reg}[t]$

Divide the contents of *Reg[s]* by *Reg[t]*, and place the quotient in *Reg[d]*. **A divisor of 0 does not generate an overflow**, and the destination register *rd* is set to all ones. DIV is binary and semantically compliant with the RV32M div instruction.

Example: **div x7,x5,x6 # Encoded as: 0x0262C3B3**

Remainder



REM: remainder of a register quotient

Syntax: **rem *rd,rs,rt***

Encoding: **0000 001*t* *tttt* *ssss* *s110* *dddd* *d011* 0011**

Description:

$\text{Reg}[d] \leftarrow \text{Reg}[s] \% \text{Reg}[t]$

Divide the contents of *Reg[s]* by *Reg[t]*, and place the remainder in *Reg[d]*. A divisor of 0 does not generate an overflow and the contents of the destination register *rd* is set to the dividend, *Reg[s]*. REM is binary and semantically compliant with the RV32M rem instruction.

Example: **rem x7,x5,x6** # Encoded as: **0x0262E3B3**



What instructions are left?

- So far, we have studied instructions that let us...
 - Do basic math!
 - Logical operations
 - Bitwise
 - Read memory
 - Write memory
- What other operations does a program do?



What instructions are left?

- So far, we have studied instructions that let us...
 - Do basic math!
 - Logical operations
 - Bitwise
 - Read memory
 - Write memory
- What other operations does a program do?
 - Change control flow!
 - I.e. take branches, jump to new locations, call functions

Branch Encoding

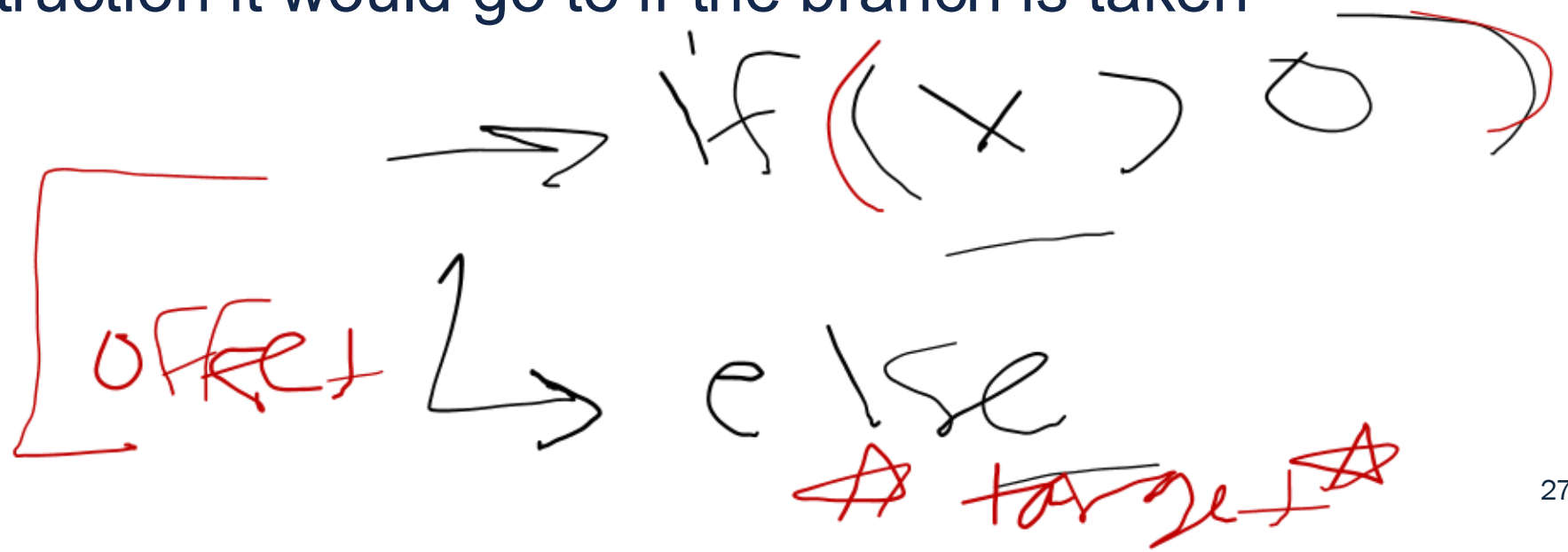


- **Target instruction**

- The instruction the program will go to if the branch is taken

- **Byte offset** to target instruction

- The distance, in bytes, between the current branch instruction and the instruction it would go to if the branch is taken



Fill in the blanks



- The byte offset will always be a multiple of ~~2~~ 4, which means that the last bits of the byte offset will always be 0.

