

9/9/26

COMP 311:

Computer Organization!

Lecture 6: Control Flow in Assembly Programs



The University
of North Carolina
at Chapel Hill

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

Logistics Update



- First lab will be released by Friday!
- Systems Reading Group Officially Starting
 - 9/23, FB141 at 5:30pm



Plan for Class Today

- 3:35-4:00: Review Branches, Jumps
- 4:00-4:15: the ABI, Stack Review
- 4:15-4:20: BREAK
- 4:20-4:50: Calling Convention, Assembly Programming with the Stack

Guiding Question ☺

How can one assembly program call another?

Quick Review: The Program Counter

RISC-V Memory Model



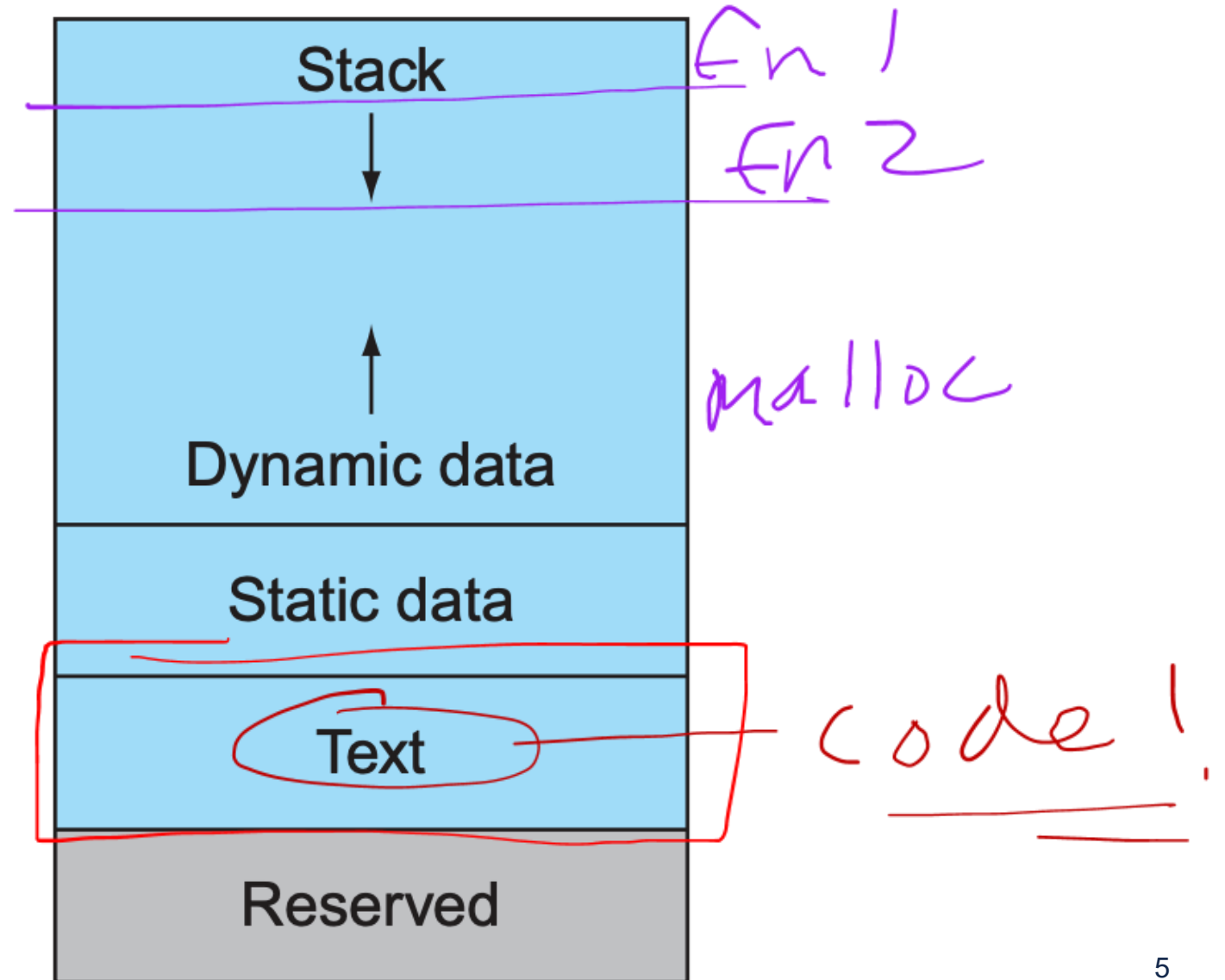
Used to manage function calls and local variables

Used for dynamic memory allocation

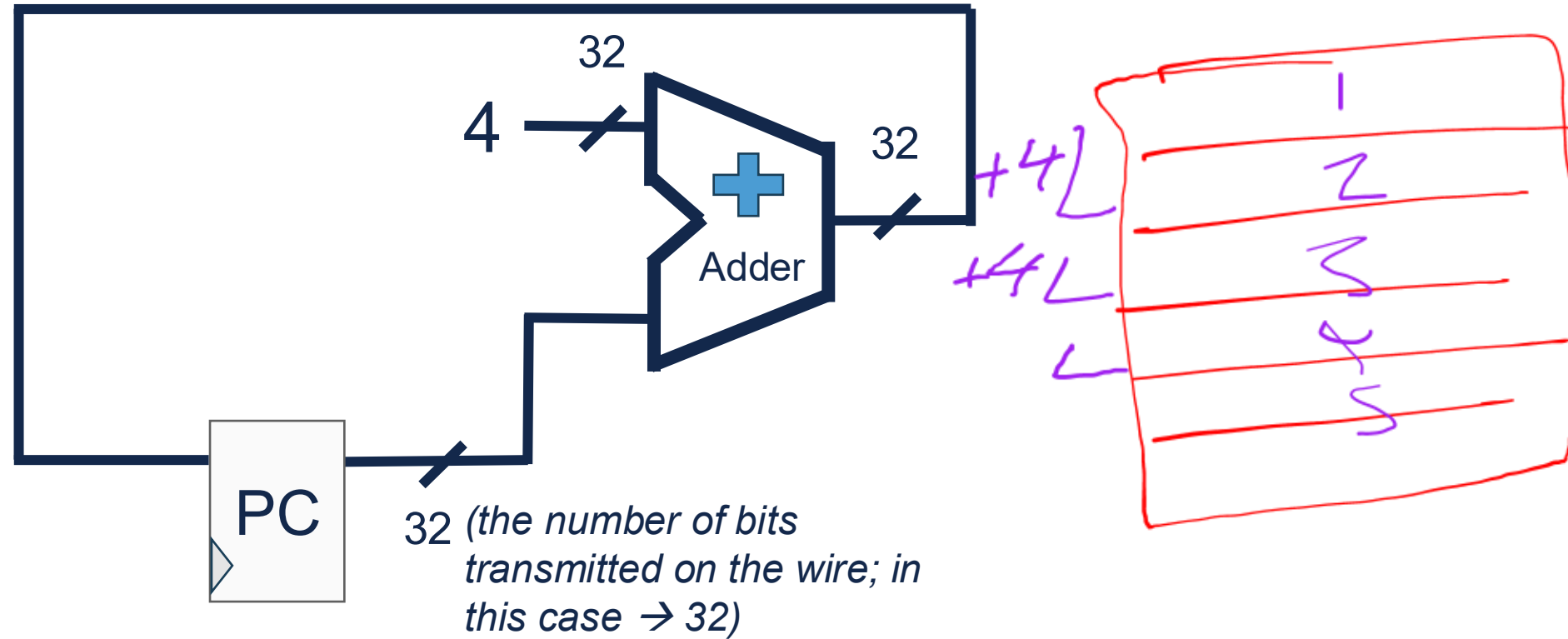
Variables allocated at compile-time

Programs

Reserved for the OS



Program Counter (PC)

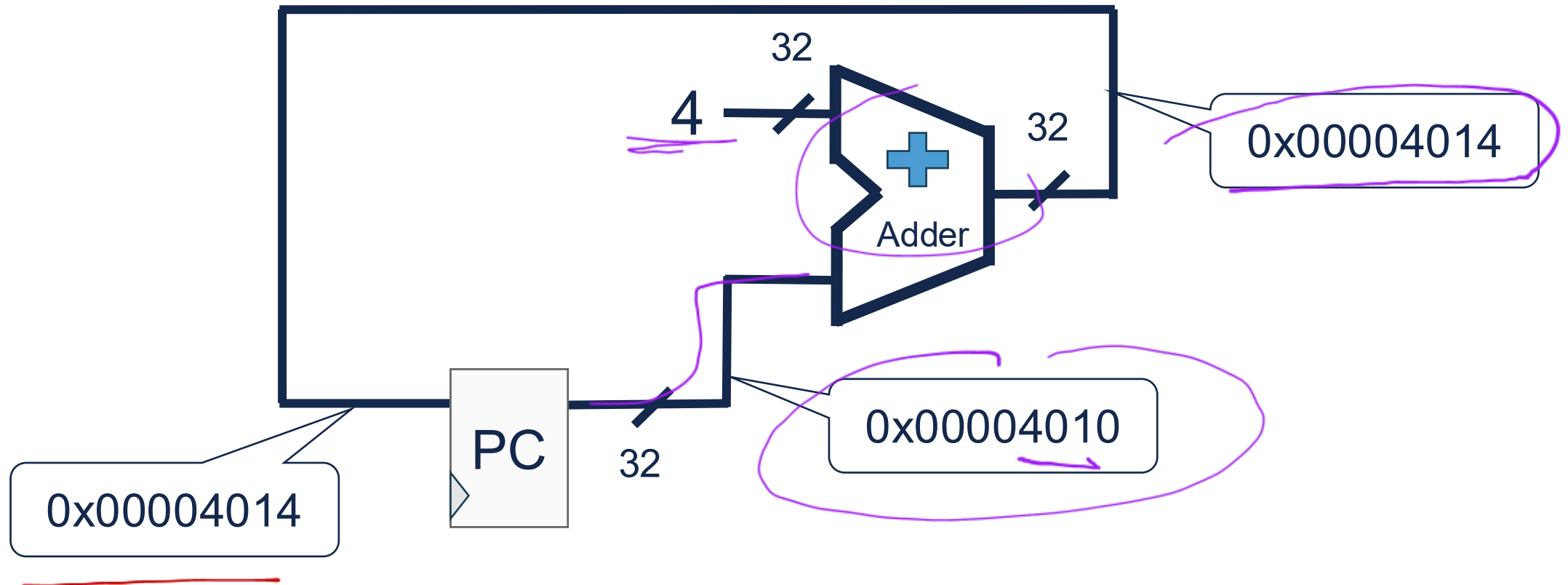


Increment the PC by 4 on every step of CPU execution

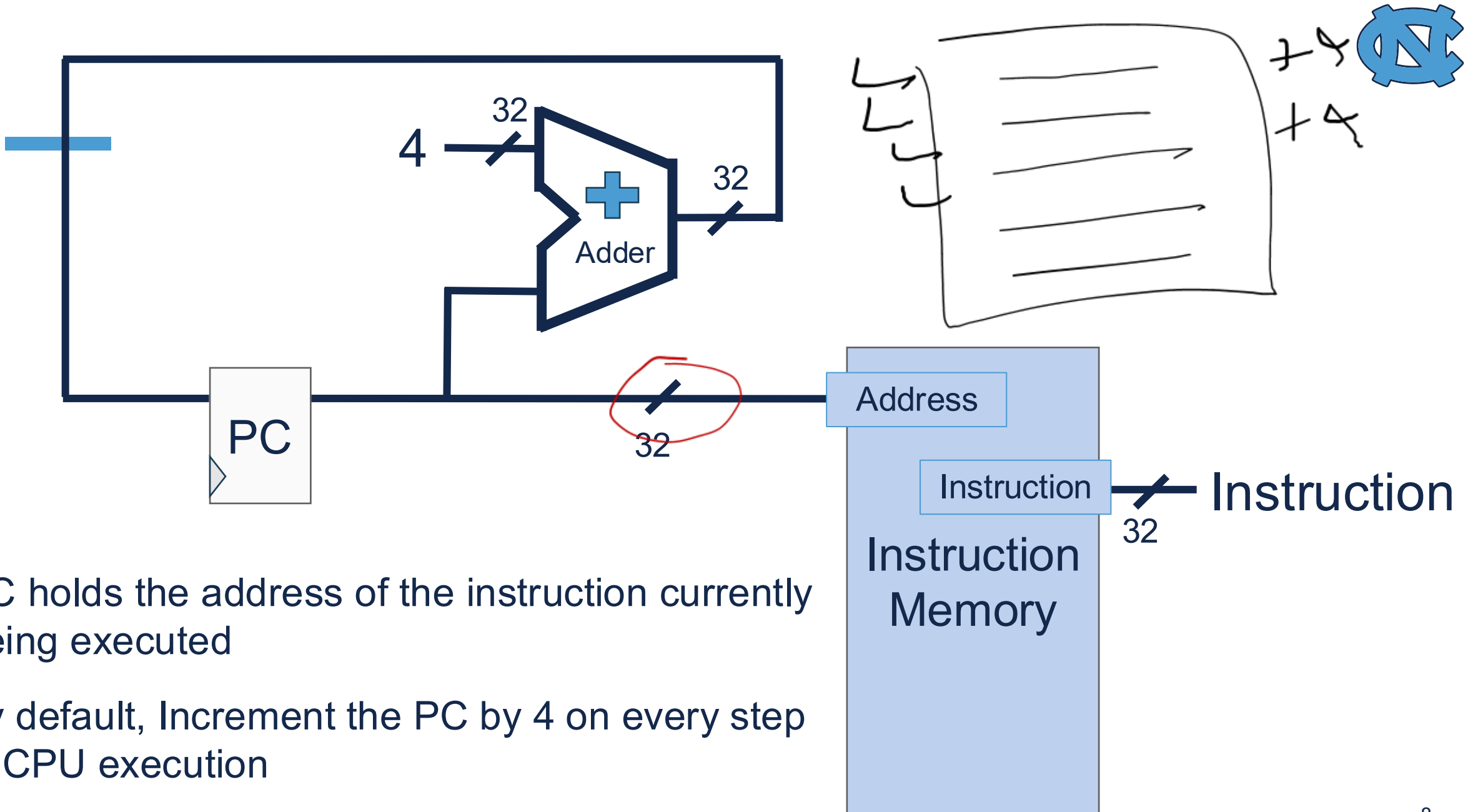
Program Counter (PC)



Example: Current PC = 0x00004010



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





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



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



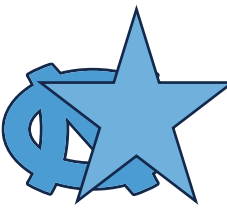
- **Target instruction**
 - The instruction the program will go to if the branch is taken
- **Byte offset** to target instruction
 - The distance, in bytes, between the current branch instruction and the instruction it would go to if the branch is taken

Fill in the blanks



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

Fill in the blanks



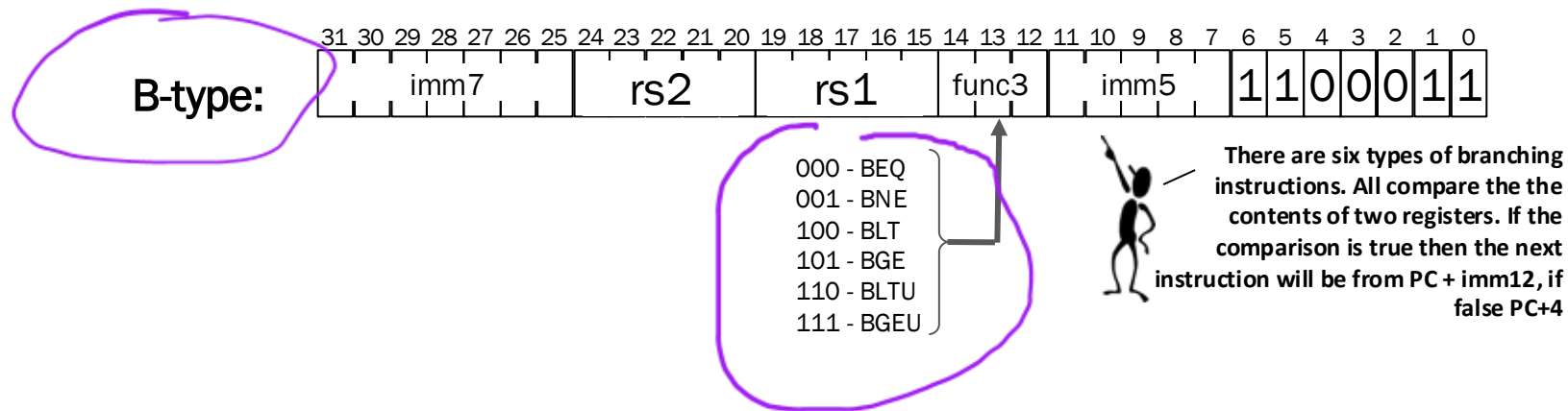
- The byte offset will always be a multiple of 4, which means that the last 2 bits of the byte offset will always be 0.
 - The byte offset will always be a multiple of 4 because the size of each instruction is 4 bytes
 - The last two bits will always be 0 because every multiple of 4 ends in two zeros
 - For example:
 - 0 = 0b000000
 - 4 = 0b00100
 - 8 = 0b01000
 - 12 = 0b01100

Branching Instructions: Changing the PC



Branches are one type of special instruction for changing the PC.

Branches are to a nearby place within +/- 512 instructions away.



Generating the Branch Immediate



1. Compute the byte offset relative to $PC + 4$ (the next instruction)
2. Remove the bottom two bits (these will always be 0)

Branch Examples



```
bne x10,x0,else
```



If the contents of x10 is not equal to 0 then branch to the nearby address with the label "else"

```
blt x10,x0,neg
```



If the contents of x10 is less than 0 then branch to the nearby address with the label "neg"

```
loop: beq x0,x0,loop
```



An infinite loop.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1



beq x0,x0,'s encoding.

Byte offset calculations



distance

- For each of the branch instructions below, what is the **byte offset** of the target instruction?

~~4 x 4~~
→ 16 bytes

beq x4, x5, elseif

add x7, x8, x9

add x7, x8, x20

jal x0, exit

elseif:

bne x4, x6, else

add x7, x10, x11

add x7, x7, x12

add x7, x7, x8

jal x0, exit

else:

~~add x7, x8, x11~~

exit:

~~5 x 4~~
→ 20

Branch Immediate



- For each of the branch instructions below, what is the **byte offset** of the target instruction?

4 instructions = 16 bytes

```
beq x4, x5, elseif  
add x7, x8, x9  
add x7, x8, x20  
jal x0, exit
```

elseif:

```
bne x4, x6, else  
add x7, x10, x11  
add x7, x7, x12  
add x7, x7, x8  
jal x0, exit
```

else:

```
add x7, x8, x11
```

exit:

5 instructions = 20 bytes

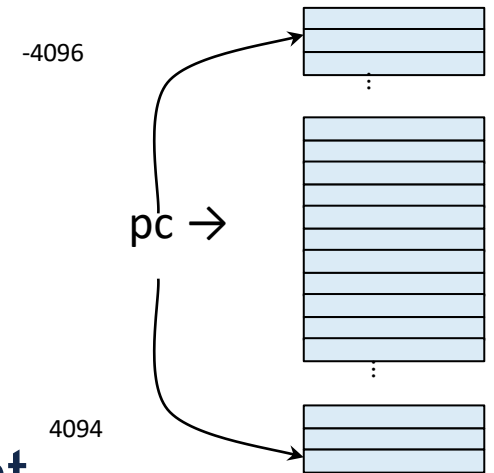
PC Relative offsets

PC relative: make the code relocatable 😊

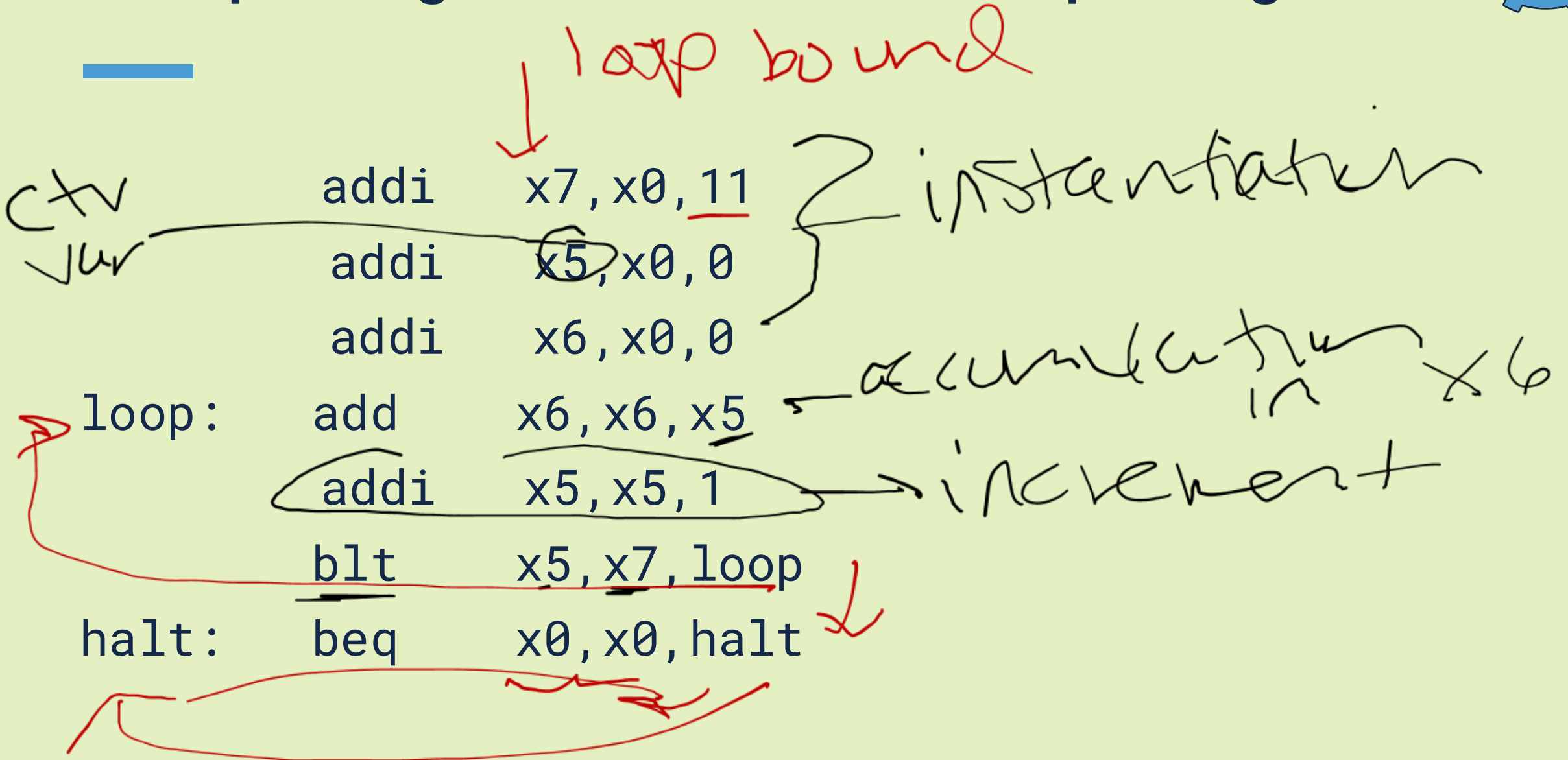
All branch offsets in RISC-V are added to the PC to determine the target address of the next instruction in the case of a valid condition. Earlier ISAs would instead specify the "absolute" target address.

What are the advantages of "relative" vs "absolute"?

- Does not implicitly limit the address space
- Requires fewer instruction bits, supports the most common cases
- Allows for "relocatable" code



A simple Program: What is the corresponding C?



A simple Program



```
        addi    x7, x0, 11      # x7 is the loop bound
        addi    x5, x0, 0       # x5 is i
        addi    x6, x0, 0       # x6 is sum
loop:    — add    x6, x6, x5      # sum = sum + i
        — addi   x5, x5, 1       # i++
        — blt    x5, x7, loop
halt:    beq     x0, x0, halt
```

Assembly code for
sum = 0;
for (i = 0; i <= 10; i++)
sum = sum + i;

Jump Instructions



- Sometimes we may want to move to a different location in our code **unconditionally**
- These types of instructions are called ***Jumps!***
- We will study 3 in particular:
 - J (Jump)
 - JAL (Jump and Link)
 - JALR (Jump and Link Register)
- These modify the control flow of program, so they also update the **Program Counter** register.

J < label >

JAL: Jump and Link

What is the technical term for
"location to resume execution from
after a function or method call"?

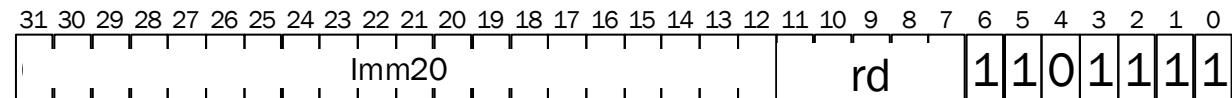


→ return
address

- Move the PC and record where to come back to!
- Link: saves $PC + 4$ in a register so we know where to resume execution from, and is how we can support function call returns.

JAL: jump and link

U-type:



Syntax:

jal rd,imm20

Description:

$Reg[d] \leftarrow PC + 4$

$PC \leftarrow PC + sign_extended(imm20)$

jal ra, label

Write the address of the following instruction, $PC + 4$, into $Reg[d]$, then jump to the instruction that is found by adding the current PC to the signed immediate offset. In practice this is usually specified by a label.

JALR: Jump and Link Register



- Similar to JAL, still writes PC+4 into the destination register
- Allows us to jump anywhere by computing the address in a register first
 - Target addr we jump to comes from base register + smaller signed immediate

JALR: jump and link register

Syntax: `jalr rd,imm12(rs)`

`jalr rd,(rs)`

Description:

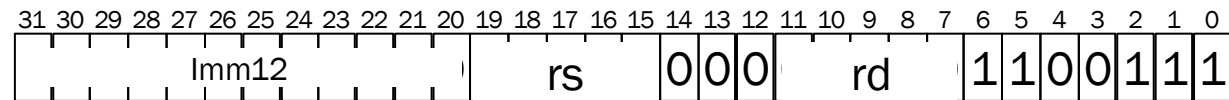
$\text{Reg}[d] \leftarrow \text{PC} + 4$

$\text{PC} \leftarrow \text{Reg}[s] + \text{sign_extended}(\text{imm12})$



jalr can be used to jump to an absolute location by first loading the address into a register

I-type:



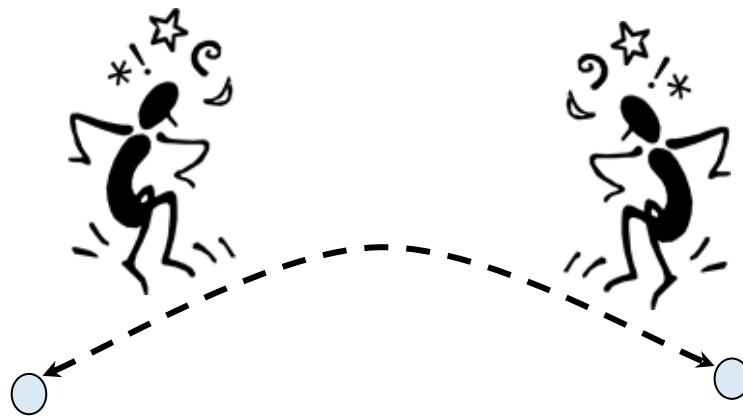
Jump to the instruction given by the contents of $\text{Reg}[s]$, and save the location of the next instruction in $\text{Reg}[s]$.

Why store PC + 4? (AKA why link?)



jal x0, <label>

These long-distance "jump" instructions also save the address of the following instruction in a register specified by **Rd**. Often, this register will be x0, and therefore is ignored. But in other cases it is useful to get back to where we jumped from.



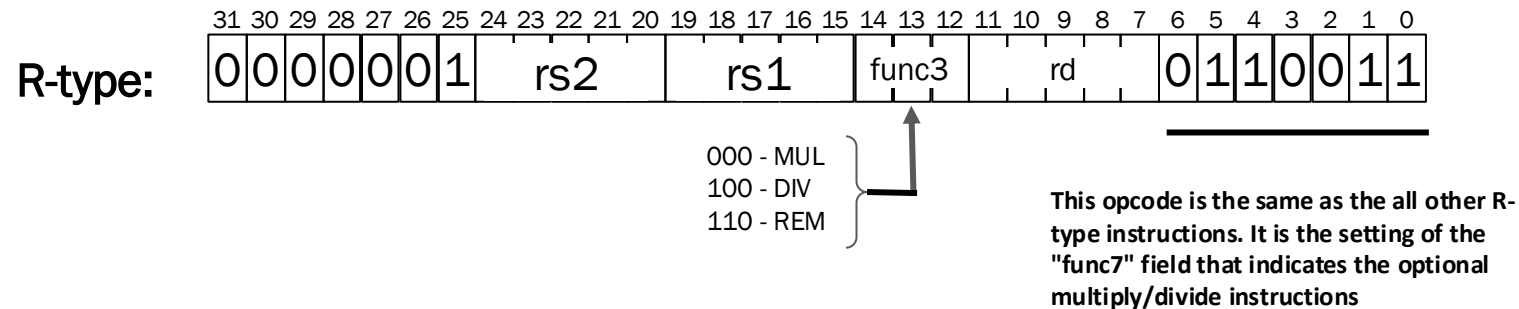
Some Missing Instructions...

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**

RISC-V Programming Activity



- In groups of 2-3, try completing the programming exercise on the worksheet using the new instructions we've learned.
 - You may use the RISC-V online documentation
 - Leonard's simulator
 - Your peers + the TAs!

```

→ addi t1, x0, 0    # sum = 0
→ addi t2, x0, 0    # i = 0
→ la t3, arr        # t3 = &arr[0]

```

} initialization

loop:

```

slli t4, t2, 2      # i * 4
add t5, t3, t4      # address of arr[i]
lw t6, 0(t5)        # t6 = arr[i]

```

} computing idx
access elts

```

→ beq t6, x0, done  # stop when arr[i] == 0

```

```

andi t7, t6, 1      # t7 = arr[i] & 1
beq t7, x0, next     # skip if even

```

```

add t1, t1, t6       # sum += arr[i]

```

next:

```

addi t2, t2, 1      # i++
j loop

```

```

done:
halt

```

← loop condition

← accumulation of sum

accesses →

if
cancel

8

1	1	0	1
0	0	0	1
0	0	0	1

A CPU's Datapath: Generally 5 Stages to Support



- Instruction Fetch
 - Reading the instruction from memory
- Instruction Decode
 - Splitting up the instruction, generating control signals, and reading the register file
 - Jump address calculation
- Execute
 - ALU Operation
 - Branch address calculation
- Memory
 - Reading/writing data memory
- Writeback
 - Writing the result to the register file

W

Now what if we want programs that include procedures and functions that call each other?

Stack Concepts and the Application Binary Interface

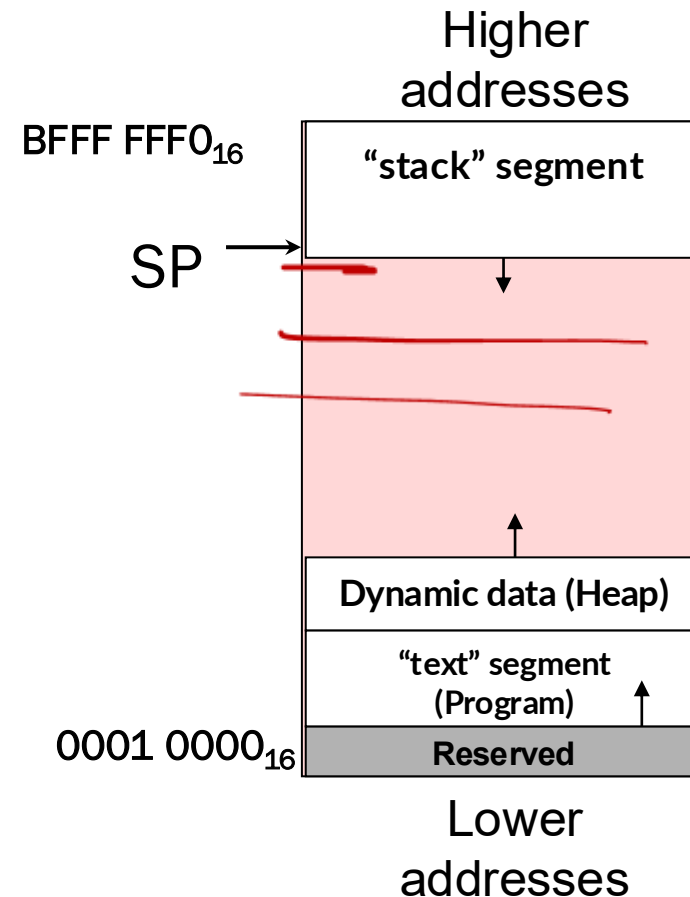
RISC-V Stack Convention



These conventions for allocating variables when calling fns →
part of the ABI

CONVENTIONS:

- Assign a register as the **Stack Pointer** ($sp = x2$).
- Stack grows **DOWN** (towards lower addresses) on *pushes* and *allocates*
- sp points to the last or **TOP** *used* location.
- Stack is placed far away from the program and its data.



Register use conventions

32 regs



By convention, the RISC-V registers are assigned to specific uses and names used in the ABI. These are supported by the assembler, and high-level languages.

We'll use these names increasingly.

Why have such conventions?



x0/zero (always zero)
x1/ra (return address)
x2/sp (stack pointer)
x3/gp (global pointer)
x4/tp (thread pointer)
x5/t0 (temporary)
x6/t1 (temporary)
x7/t2 (temporary)
x8/fp (frame pointer)
x9/s1 (saved)
x10/a0 (argument/return value 1)
x11/a1 (argument/return value 2)
x12/a2 (argument)
x13/a3 (argument)
x14/a4 (argument)
x15/a5 (argument)

x16/a6 (argument)
x17/a7 (argument)
x18/s2 (saved)
x19/s3 (saved)
x20/s4 (saved)
x21/s5 (saved)
x22 (saved)
x23 (saved)
x24 (saved)
x25 (saved)
x26 (saved)
x27 (saved)
x28 (temporary)
x29 (temporary)
x30 (temporary)
x31 (temporary)

Prevents functions from accidentally overwriting each other's values

Allows independently compiled code to co-execute correctly!

Separately written fns can safely share CPU without corrupting each other's data

Basics of Procedure Calling



```
int x = 35;
int y = 55;
int z;

void main() {
    z = gcd(x, y);
}

int gcd(a,b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

Basics of Procedure Calling

Note: ra is our ABI
alias/convention for register x1.
Function calling convention!



```
int x = 35;
int y = 55;
int z;

void main() {
    z = gcd(x, y);
}

int gcd(a,b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

```
main:    lw    a0,x
         lw    a1,y
         jal   ra,gcd
         sw    a0,z
*halt:   j     halt

x:       .word 35
y:       .word 55
z:       .word 0

gcd:     beq   a0,a1,return
         blt   a0,a1,else
         sub   a0,a0,a1
         beq   x0,x0,gcd
else:    sub   a1,a1,a0
         beq   x0,x0,gcd
return:  jalr  zero,(ra)
```

That was a little too easy...



```
int x = 5;
int y;

void main() {
    y = fact(x);
}

int fact(x) {
    if (x <= 1)
        return x;
    else
        return x*fact(x-1);
}
```

Q: What happens when we trace this?

```
main:    lw    a0,x
         jal   ra,fact
         sw    a0,y
*halt:   j     halt

x:       .word  2
y:       .word  0

fact:    ←addi   t0,x0,1
         bge    t0,a0,return
         addi   t0,x0,a0
         addi   a0,a0,-1
         →jal    ra,fact
         mul    a0,a0,t0
return:  jalr   x0,ra
```

That was a little too easy...

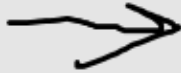


- The recursive call to fact() overwrites the saved value of x in t0.
- To make a bad thing worse, the ra is also overwritten </3

```
main:    lw    a0,x
         jal   ra,fact
         sw    a0,y
*halt:   j     halt

x:        .word    2
y:        .word    0

fact:     addi   t0,x0,1
         bge    t0,a0,return
         addi   t0,x0,a0
         addi   a0,a0,-1
         jal    ra,fact
         mul    a0,a0,t0
         return: jalr   x0,ra
```

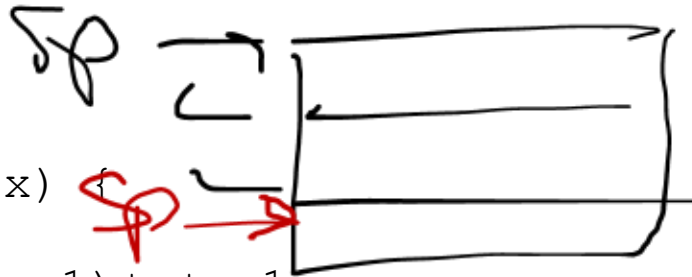


Incorporating A Stack

```
int sqr(int x) {
    if (x > 1)
        x = sqr(x-1)+x+x-1;
    return x;
}
```

```
main()
{
    sqr(10);
}
```

Every call **pushes** a frame (**prologue**) and **pops** it (**epilogue**), so nested/recursive calls don't overwrite each other's saved ra or saved registers



sqr:

```
addi    sp, sp, -8
sw       ra, 4(sp)
sw       s0, 0(sp)
slti     t0, a0, 2
bne      t0, x0, return
add      s0, x0, a0
addi     a0, a0, -1
jal      ra, sqr
add      a0, a0, s0
add      a0, a0, s0
addi     a0, a0, -1
```

function
prologue

return:

```
lw       s0, 0(sp)
lw       ra, 4(sp)
addi    sp, sp, 8
jalr     x0, ra
```

function
epilogue

main:

```
addi     sp, sp, -4
sw       ra, (sp)
addi     a0, x0, 10
jal      ra, sqr
lw       ra, (sp)
addi     sp, sp, 4
jalr     x0, ra
```

2 w over

Stack Management Policies



ALLOCATE k: reserve k WORDS of stack
 $SP = SP - 4 * k$

```
addi  sp,sp,-4*k
```

DEALLOCATE k: release k WORDS of stack
 $SP = SP + 4 * k$

```
addi  sp,sp,4*k
```

PUSH x: push Reg[x] onto stack
 $Mem[SP - 4] = Rx$
 $SP = SP - 4$

```
addi  sp,sp,-4  
sw    rx,(sp)
```

POP x: pop the top of the stack into Reg[x]
 $Rx = Mem[SP]$
 $SP = SP + 4$

```
lw    rx,(sp)  
addi  sp,sp,4
```

Practice Time! 😊



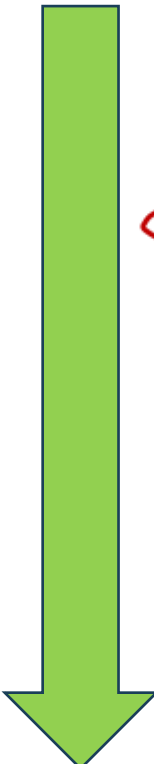
Stack Tracing. Following along w/ WS Code



saved ra: [0] Initial/Base Frame. Nothing is below this on the stack

Each stack entry stores the return address, which is the address of the next instruction after JAL

Stack Tracing. Following along w/ WS Code



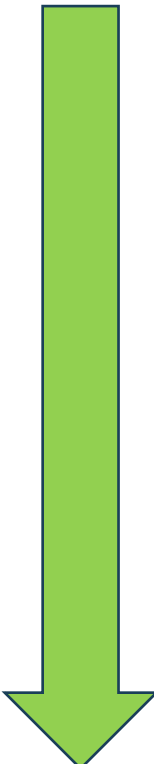
saved ra: [0] Initial/Base Frame. Nothing is below this on the stack

saved ra: [12] Main called from Reset

12
ra: PC + 4

addr 8: jal ra, main

Stack Tracing. Following along w/ WS Code

A large green arrow pointing downwards, indicating the direction of stack growth or the sequence of frames.

saved ra: [0] Initial/Base Frame. Nothing is below this on the stack

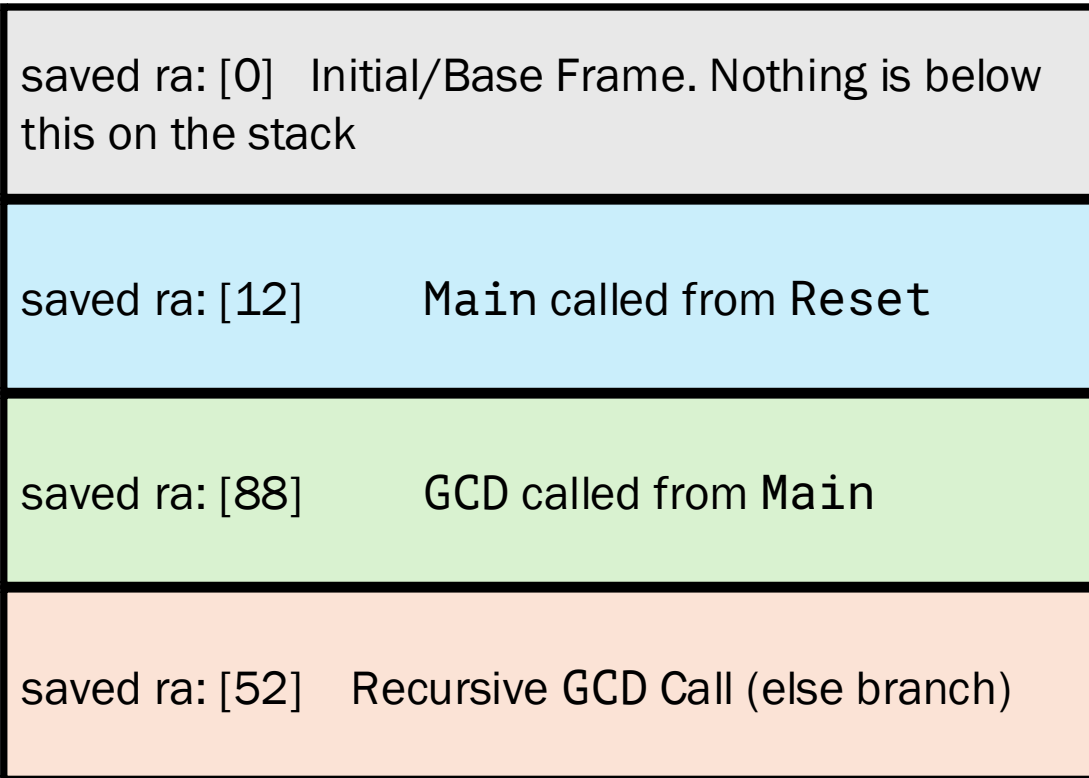
saved ra: [12] Main called from Reset

saved ra: [88] GCD called from Main

addr 8: jal ra, main

addr 84: jal ra, gcd

Stack Tracing. Following along w/ WS Code

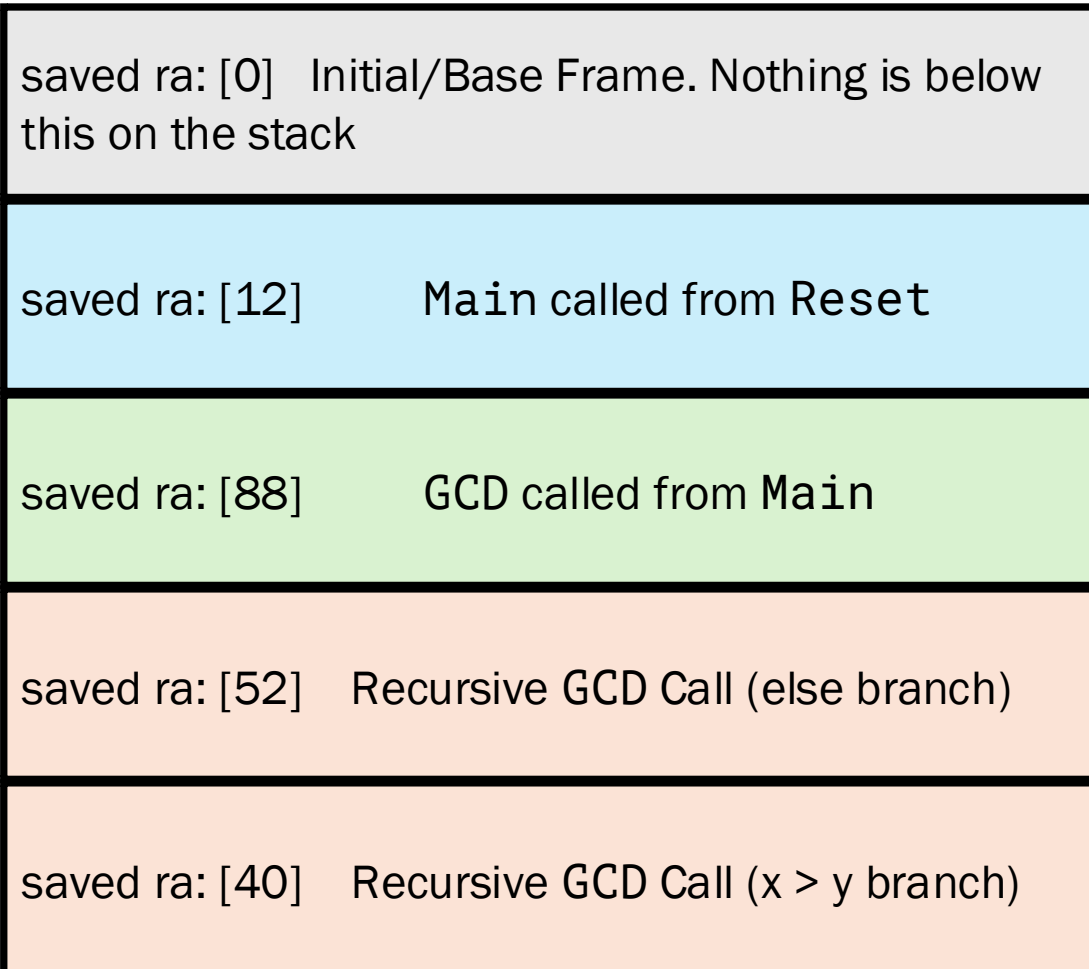


addr 8: jal ra, main

addr 84: jal ra, gcd

addr 48: jal ra, gcd

Stack Tracing. Following along w/ WS Code



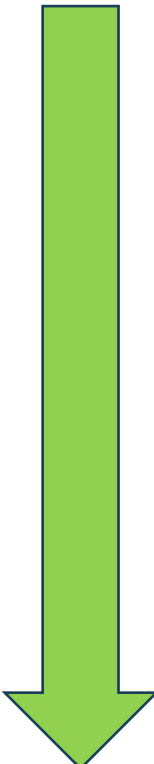
addr 8: jal ra, main

addr 84: jal ra, gcd

addr 48: jal ra, gcd

addr 36: jal ra, gcd

Stack Tracing. Following along w/ WS Code



saved ra: [0] Initial/Base Frame. Nothing is below this on the stack
saved ra: [12] Main called from Reset
saved ra: [88] GCD called from Main
saved ra: [52] Recursive GCD Call (else branch)
saved ra: [40] Recursive GCD Call (x > y branch)

Each stack entry stores the return address, which is the address of the next instruction after JAL

addr 8: jal ra, main

addr 84: jal ra, gcd

addr 48: jal ra, gcd

addr 36: jal ra, gcd

Code interacts with the stack in 3 main ways



- At the beginning of the function, it will “*push* a stack frame onto the call stack” by moving SP downward to make space for its own stack frame.
- During the execution of the function, it will use (positive) offsets from sp or fp to locate each of its local variables.
- At the end of the function, before it returns, it will “*pop* the stack frame off the call stack” by moving sp back up to wherever it used to be, “destroying” its stack frame.