pollev.com/kakiryan

COMP311: *COMPUTER ORGANIZATION!*

Lecture 23: Branches, Jumps!

tinyurl.com/comp311-fa25

RISC-V Resources

- I posted a "reading". Please do this if you haven't already!
- Chapter 2 of the RISC-V reader/"Mona Lisa Book"
  - *I will give you p. 15-16 of this book on the next quiz and on the final as a reference sheet (on the next slides, too)*
- Play around with the simulator!!
  - *The best way to learn is by doing in this setting. (Think of other times you have learned a programming language ☺)*
- If you want even more details, you can read Andrew Waterman's PhD dissertation
  - *https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf*

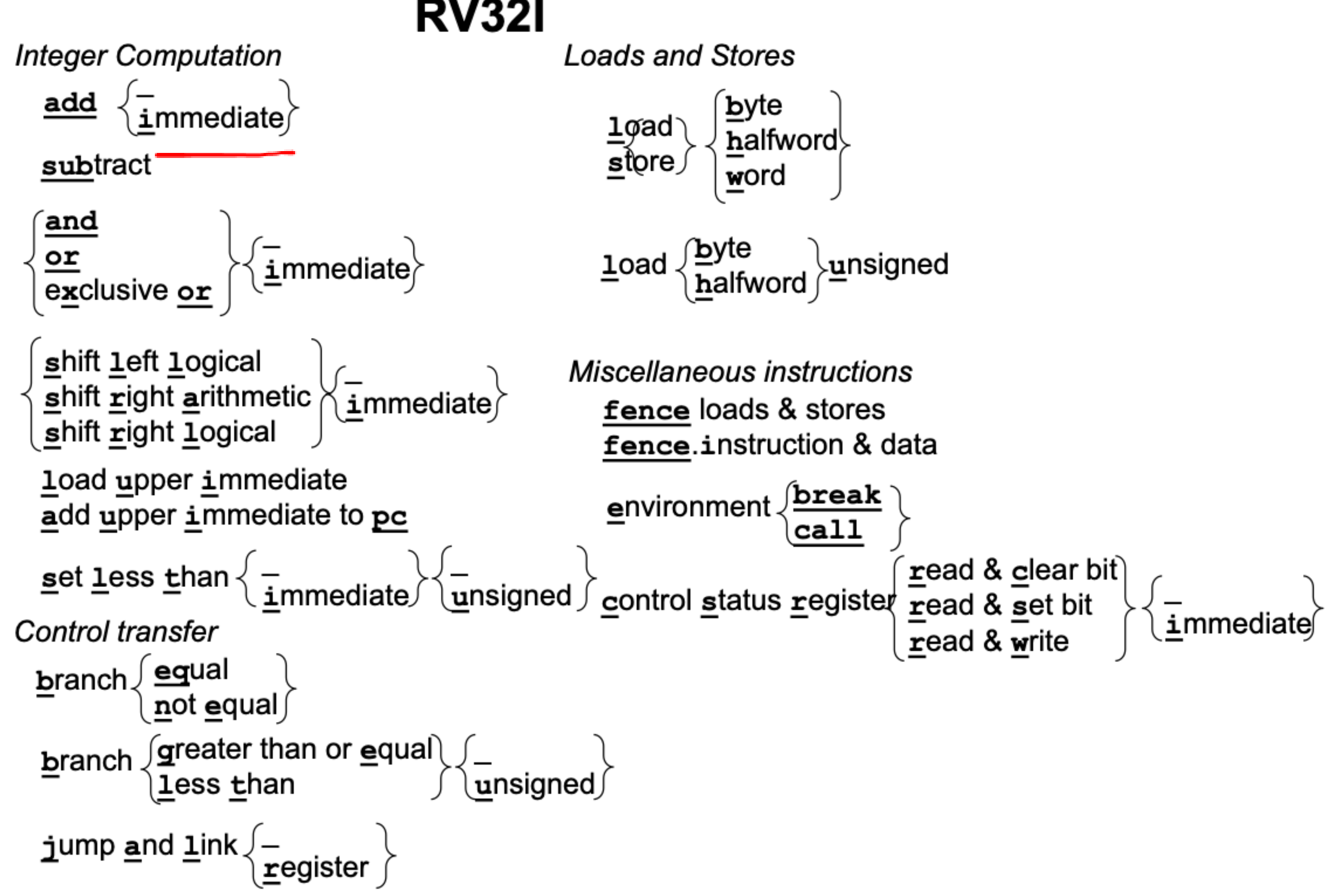


Figure 2.1: Diagram of the RV32I instructions. The underlined letters are concatenated from left to right to form RV32I instructions. The curly bracket notation { } means each vertical item in the set is a different variation of the instruction. The underscore _ within a set means that one option is simply the instruction name so far without a letter from this set. For example, the notation near the upper left-hand corner represents the following six instructions: `and`, `or`, `xor`, `andi`, `ori`, `xori`.

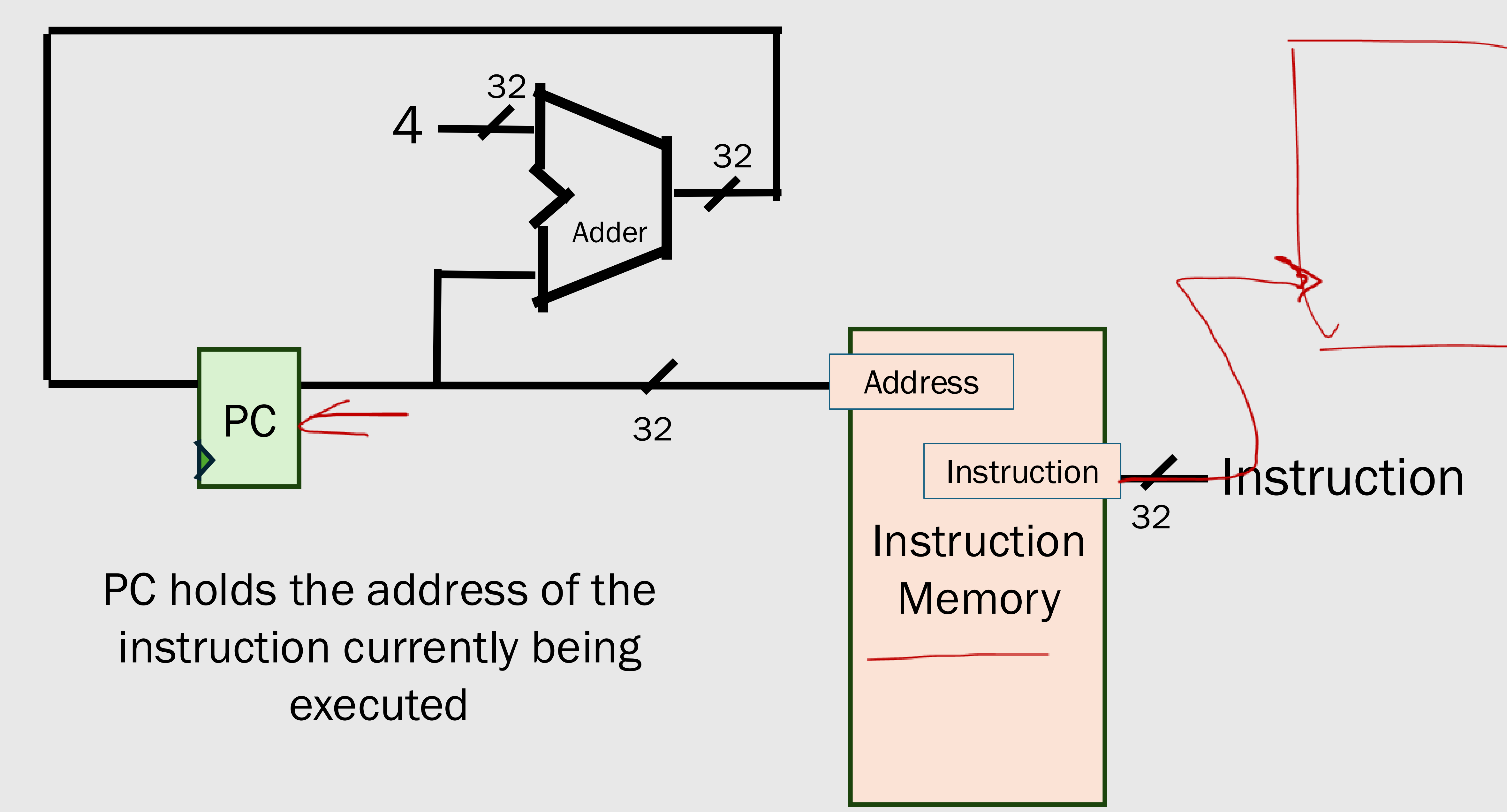
31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred	succ	00000	000	00000	0001111	I fence
0000 0000	0000	00000	001	00000	0001111	I fence.i
000000000000		00000	000	00000	1110011	I ecall
000000000001		00000	000	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I csrrsi
csr		zimm	111	rd	1110011	I csrrci

Figure 2.3: RV32I opcode map has instruction layout, opcodes, format type, and names. (Table 19.2 of [Waterman and Asanović 2017] is the basis of this figure.)

31	30 25	24 21	20	19 15	14 12	11 8	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]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

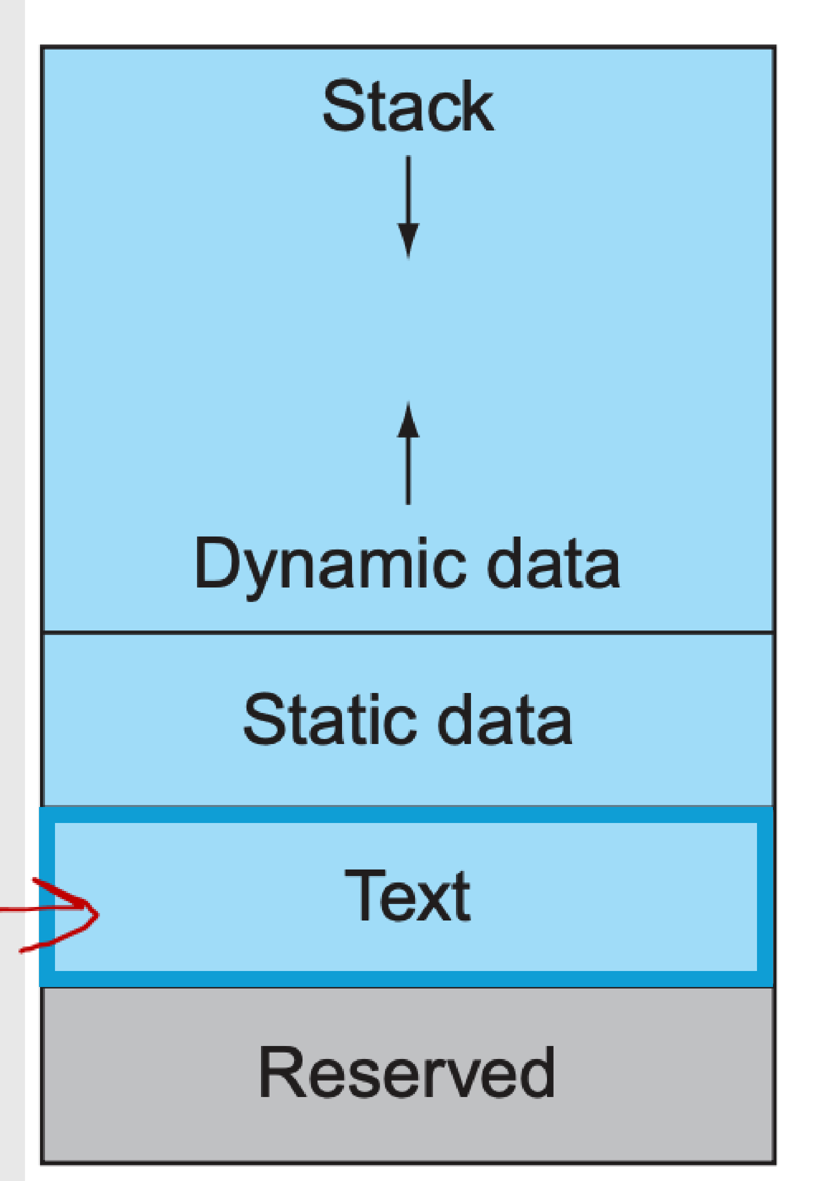
Figure 2.2: RISC-V instruction formats. We label each immediate subfield with the bit position (imm[x]) in the immediate value being produced, rather than the bit position in the instruction's immediate field as is usually done. Chapter 10 explains how the control status register instructions use the I-type format slightly differently. (Figure 2.2 of Waterman and Asanović 2017 is the basis of this figure).

LET’S DO SOME RISC-V REVIEW NOW!


32
4
32
Adder
PC
32
Address
Instruction
Instruction
32
Instruction
Memory
PC holds the address of the
instruction currently being
executed

Storing Code in Memory

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



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

###########################################

x:       .word   144          # int x = 144;

main:    addi    sp,sp,-16    # main() {
         sw      ra,12(sp)
         lw      a0,x
         jal     ra,sqrt      #     return sqrt(x);
         lw      ra,12(sp)
         addi    sp,sp,16
         jalr    zero,(ra)    # }

sqrt:    addi    t0,a0,0      # sqrt(x) {
         addi    t1,zero,1    #     int odd = 1;
         addi    a0,zero,0    #     int result = 0;
         blt     t0,t1,return #     while (x >= odd) {
loop:    sub     t0,t0,t1     #         x -= odd;
         addi    t1,t1,2      #         odd += 2;
         addi    a0,a0,1      #         result += 1;
         bge     t0,t1,loop   #     }
return:  jalr    zero,(ra)    #     return result;
                              # }
```

ically implemented using the

Memory Example

- If the address of A is 0x00003000, what is the value of registers 14-18 after the program is run?

```
A: .word      5 90 100 40 11

la    x14, A
lw    x15, 0(x14)    # x15 = A[0]
addi  x16, x14, 8    # x16 = &A[2]
lw    x17, 4(x16)    # x17 = A[3]
sw    x17, 4(x14)    # A[1] = x17
lw    x18, -4(x16)   # x18 = A[1]
```

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.

C:

```c
int sum = 0;
int values[10] = {1,3,5,7,9,11,
                  13,15,17,19};

int i;

for (i = 0; i < 10; i++)
    sum += value[i];
```

Assembly:

```
        addi   x31,x0,10
        addi   x5,x0,0          # x5 is i
loop:   slli   x6,x5,2
        lw     x6,values(x6)    # value[i]
        lw     x7,sum(x0)       # x5 is sum
        add    x7,x7,x6         # sum += value[i];
        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
```

Straight-Line Code

At this point all of our standard R-type, I-type, and S-type instructions are implemented.

The resulting machine executes only straight in-line code

Can't get much interesting done this way.

No function calls, for-loops, while-loops, or if-then-else logic

BRANCHING, JUMPING & (OTHER MISC. INSTRUCTIONS..)

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

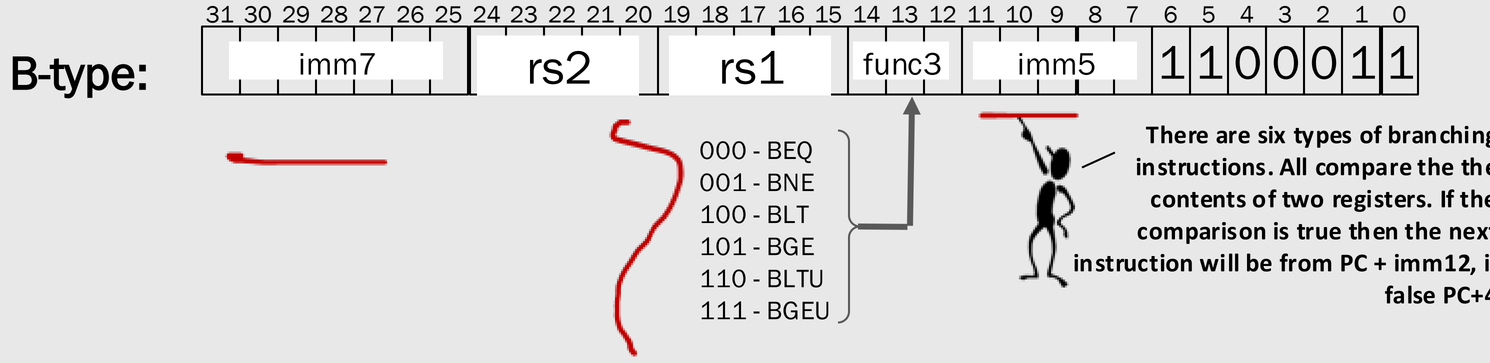
- The byte offset will always be a multiple of ____4____, which means that the last ____2____ bits of the byte offset will always be 0.

Generating the Branch Immediate

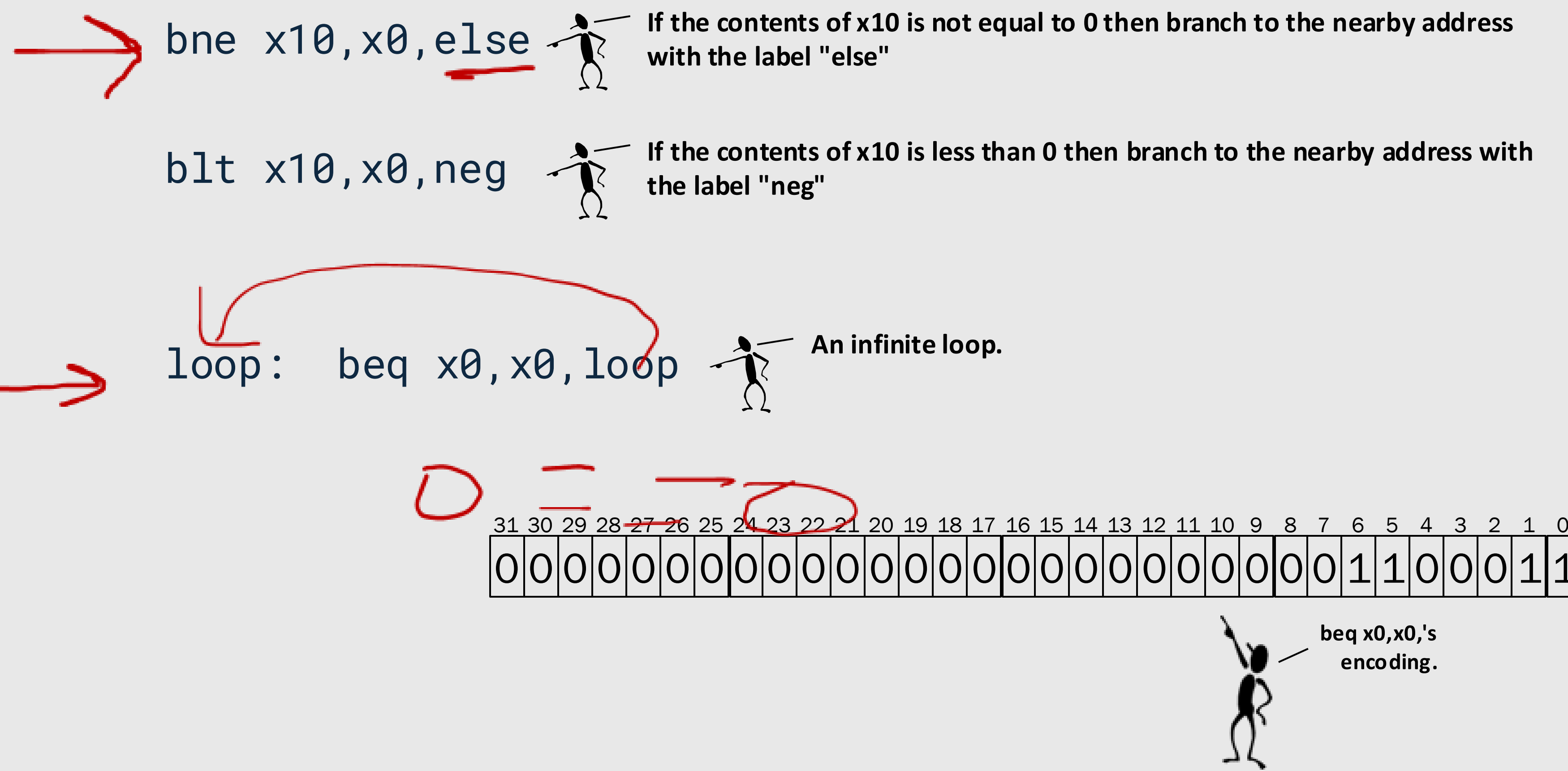
1. Compute the byte offset relative to PC + 4 (the next instruction)
2. Remove the bottom two bits (these will always be 0)

Branching Instructions: Changing the PC

The Program Counter, or PC, is a special register that points to the address of the next instruction to be fetched. There are special instructions for changing the PC. One type is Branching Instructions. Branches are to nearby place within +/- 512 instructions away.

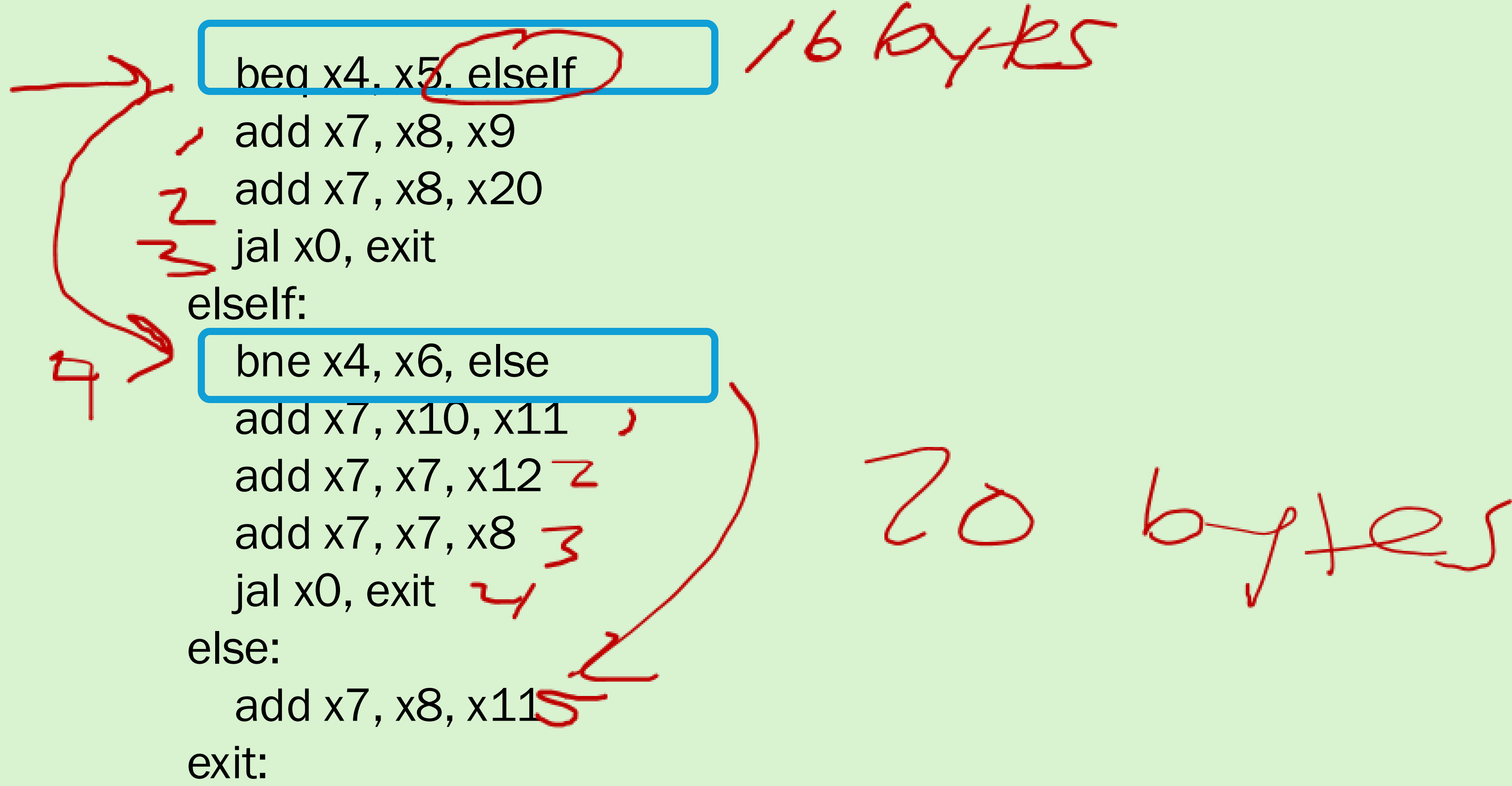


Branch Examples



Branch Immediate

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



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, elself
   add x7, x8, x9
   add x7, x8, x20
   jal x0, exit
elself:
   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

-4096

pc

4094

A simple Program

```
        addi   x7,x0,11      # x7 is 10 + 1
        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
```

A simple Program

```
        addi   x7,x0,11      # x7 is 10 + 1
        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;
```

Jumping long distances

JAL: move the PC and record where to come back to!

There are two more instructions for jumping long distances. Both are "unconditional", meaning that the branch is always taken, but they have another interesting feature

U-type:

31–12	11–7	6	5	4	3	2	1	0
Imm20	rd	1	1	0	1	1	1	1

JAL: jump and link

Syntax: *jal rd,imm20*

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

Description:

Reg[d] ← PC + 4

PC ← PC + sign_extended(imm20)

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.

The link step (saving PC + 4 in a register) is how we can support fn call returns

Jumping long distances

JALR: jump anywhere by computing addr in register first.

There are two more instructions for jumping long distances. Both are "unconditional", meaning that the branch is always taken, but they have another interesting feature

I-type:

31–20	19–15	14–12	11–7	6–0
Imm12	rs	000	rd	1100111

JALR: jump and link register

Syntax: *jalr rd,imm12(rs)*
jalr rd,(rs)

Description:

$Reg[d] \leftarrow PC + 4$

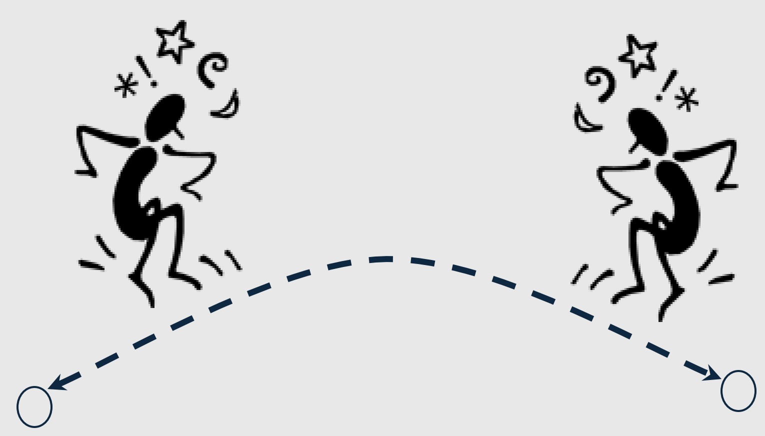
$PC \leftarrow Reg[s] + sign_extended(imm12)$

Jump to the instruction given by the contents of *Reg[s]*, and save the location of the next instruction in *Reg[s]*.

Why store PC + 4?

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.



THE APPLICATION BINARY INTERFACE (ABI) AND PROCEDURE CALLS!

The Beauty and Power of Procedures

- Reusable code fragments (modular design)

```
  clear_screen();
// code to draw a bunch of lines
  clear_screen();
        …
```

- Parameterized procedures (variable behaviors)

```
  line(x1,y1,x2,y2,color);
  line(x2,y2,x3,y3,color);
        …
```

```
for (int i = 0; i < N-1; i++)
    line(x[i],y[i],x[i+1],y[i+1],color);
line(x[i],y[i],x[0],y[0],color);
```

- Functions (procedures that return values)

```
  xMax = max(max(x1,x2),x3);
  yMax = max(max(y1,y2),y3);
```

More Procedure Power

Global vs. Local scope (Name Independence)

```
int x = 9;

int fee(int x) {
   return x+x-1;
}

int foo(int i) {
   int x = 0;
   while (i > 0) {
        x = x + fee(i);
        i = i - 1;
   }
  return x;
}

main() {
     fee(foo(x));
}
```

These are different "x"s

This is yet another "x"

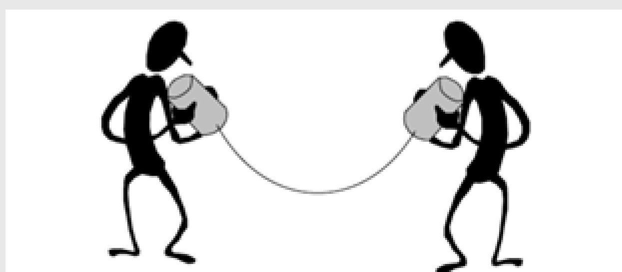
That "fee( )" seems odd to me? And, foo( )'s a little square.

How do we keep track of all these variables?

Functions and procedure Calls

Functions and procedures are essential components of code reuse. The also allow code to be organized into modules. A key components of procedures are they:

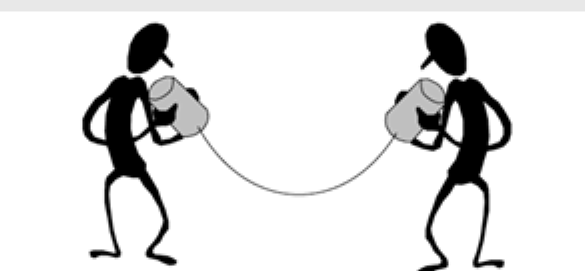
- can be called from anywhere by a caller, and, when finished, they return back to where they were called from
- can have their only local variables
- clean up behind themselves, they avoid creating unintended side-effects
- can call themselves to implement Recursive methods/functions



Supporting procedure Calls

Reusable code also requires agreed upon conventions, such as where a caller's arguments can be found by the callee. These are actually not part of the ISA, ***they are part of a standard called the processor's "Application Binary Interface" or ABI.***

<u>Basics of procedure calling:</u>



1. Put parameters where the called procedure can find them
2. Transfer control to the procedure
3. Acquire the needed storage for procedure variables
4. Perform the expected calculation
5. Put the result where the caller can find them
6. Return control to the point just after where it was called

Register use conventions

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)	x16/a6 (argument)
x1/ra (return address)	x17/a7 (argument)
x2/sp (stack pointer)	x18/s2 (saved)
x3/gp (global pointer)	x19/s3 (saved)
x4/tp (thread pointer)	x20/s4 (saved)
x5/t0 (temporary)	x21/s5 (saved)
x6/t1 (temporary)	x22 (saved)
x7/t2 (temporary)	x23 (saved)
x8/fp (frame pointer)	x24 (saved)
x9/s1 (saved)	x25 (saved)
x10/a0 (argument/return value 1)	x26 (saved)
x11/a1 (argument/return value 2)	x27 (saved)
x12/a2 (argument)	x28 (temporary)
x13/a3 (argument)	x29 (temporary)
x14/a4 (argument)	X30 (temporary)
x15/a5 (argument)	X31 (temporary)

Prevents functions from accidentally overwriting each other's values

Allows independently compiled code to co-execute correctly!

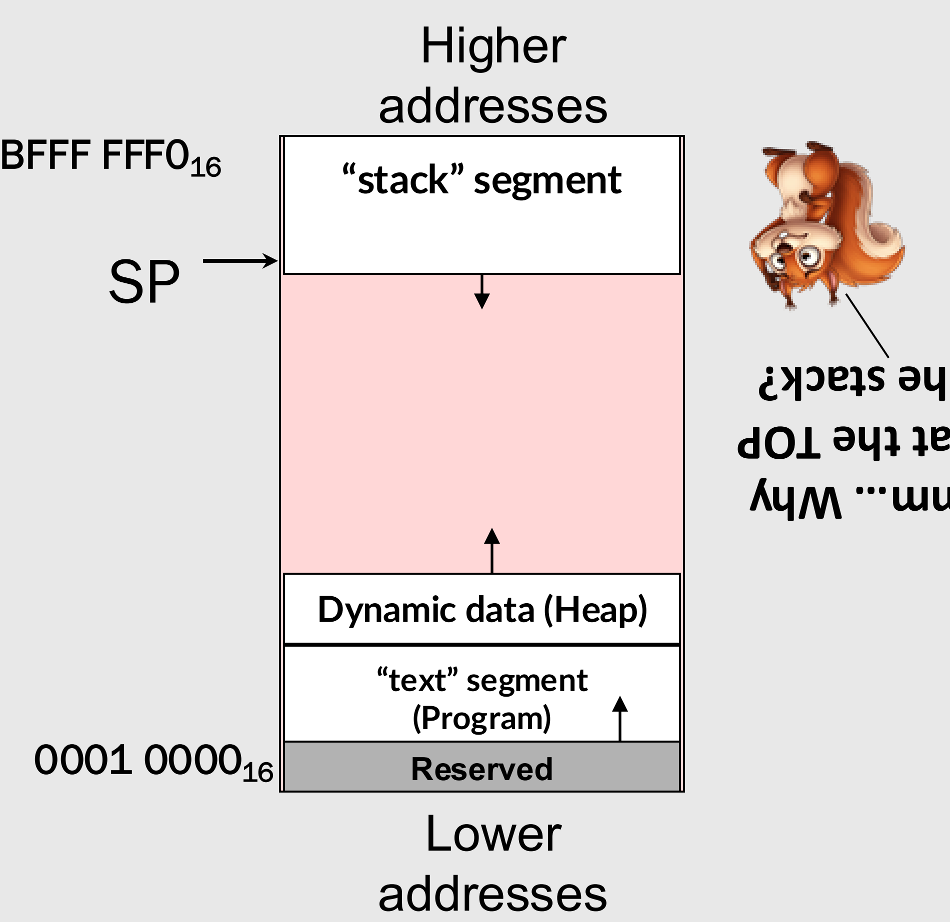
Separately written fns can safely share CPU without corrupting each other's data

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.**



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
```

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

```
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);
}
```

```
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
```

This time, things are really messed up.

The recursive call to fact( ) overwrites the saved value of x in t0.

To make a bad thing worse, the ra is also overwritten.

I knew there was a reason that I avoid recursion.

Incorporating A Stack

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


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

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

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
```

function prologue

function epilogue

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