COMP311: *COMPUTER ORGANIZATION!*

Lecture 19: RISC-V Programming with Procedures, Odds & Ends

tinyurl.com/comp311-sp26

The RISC-V was designed so every instruction keeps the same few fields (opcode, rs1, rs2, rd) in the same positions, and the immediates have to bend around that fixed layout.

FUN IMMEDIATE ENCODING DETAILS ☺

The immediates are complicated/confusing in some cases b/c RISC-V prioritizes a consistent instruction layout over simple immediate encoding! (design tradeoff…)

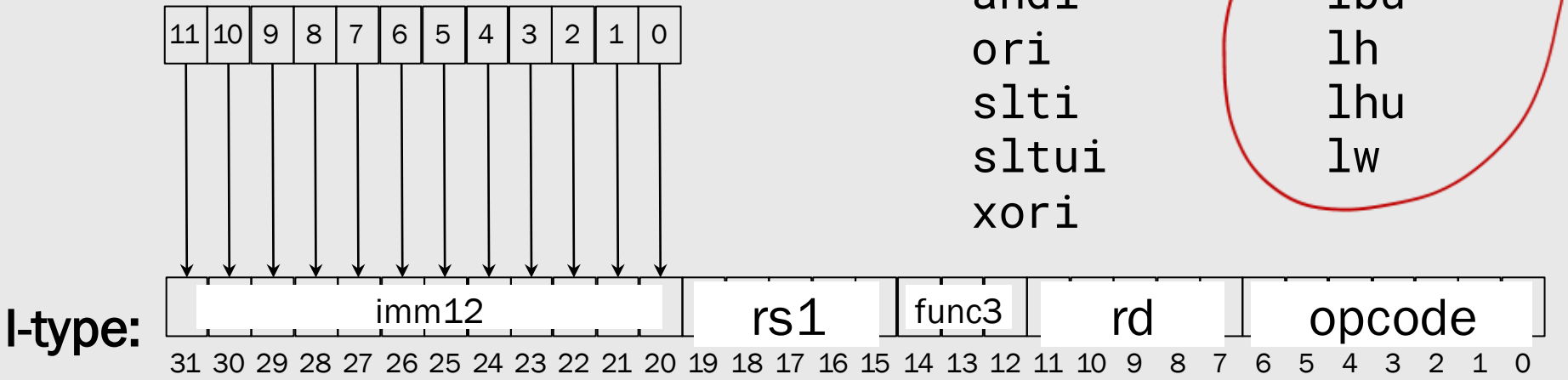
Immediate Encodings

Many RISC-V instructions encode a constant value as part of the instruction. Unlike other ISAs, the encoding of constants is not uniform. The I-type instructions encode their immediate operand as follows:

Examples:

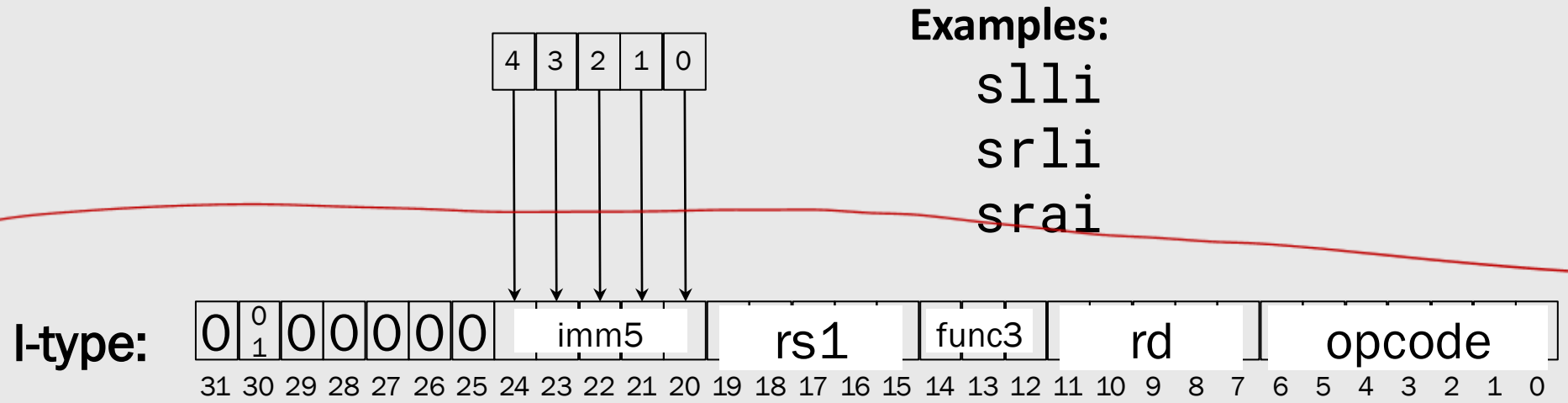
```
addi
andi
ori
slti
sltui
xori
```

```
lb
lbu
lh
lhu
lw
```



Immediate Encodings

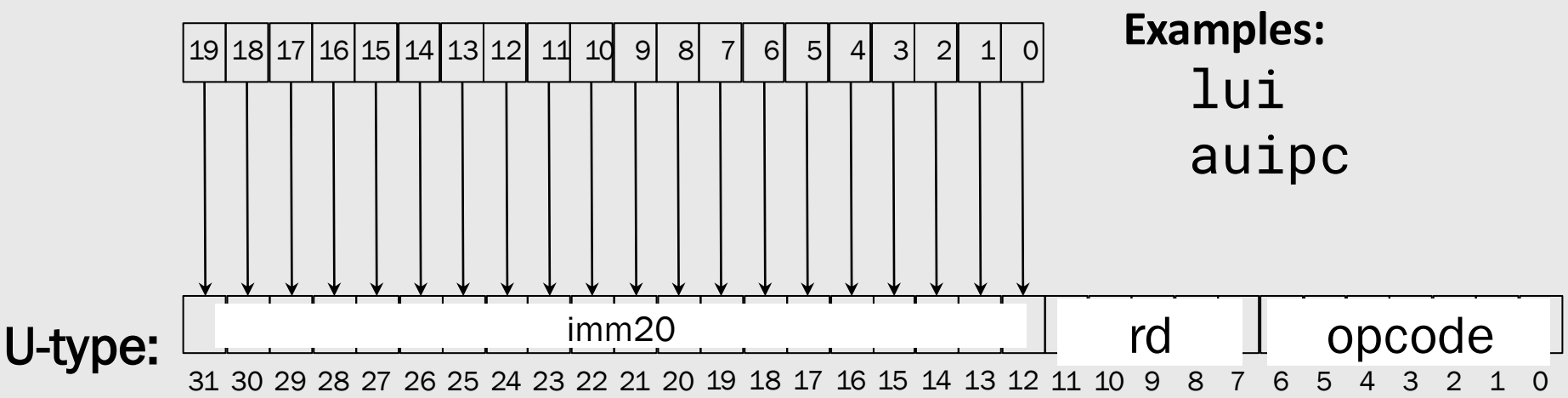
Many RISC-V instructions encode a constant value as part of the instruction. Unlike other ISAs, the encoding of constants is not uniform. The I-type instructions encode their immediate operand as follows:



But there are exceptions!

LUI's Immediate Value

The immediate-field encoding of `lui` and `auipc` also seems straightforward, but, the immediate values might not be what you expect due to the sign-extension of its companion addi instruction.



Examples:

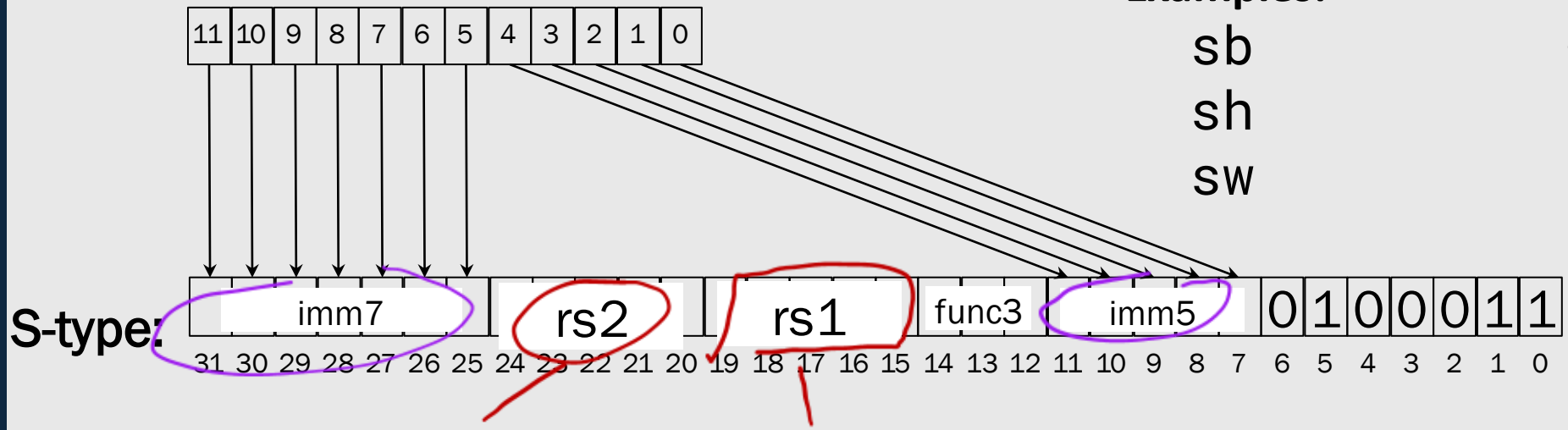
```
lui
auipc
```

if bit 11 of the large constant is set, then you add 1 to the LUI/AUIPC immediate value.

sw x5, (12)(x10)

Immediate Offsets for Stores

RISC-V store instructions include an offset, but it is not encoded in an obvious way. Note that there is no need for a rd register in a store, but both rs1 and rs2 are needed if the encodings are used consistently.



Examples:

sb

sh

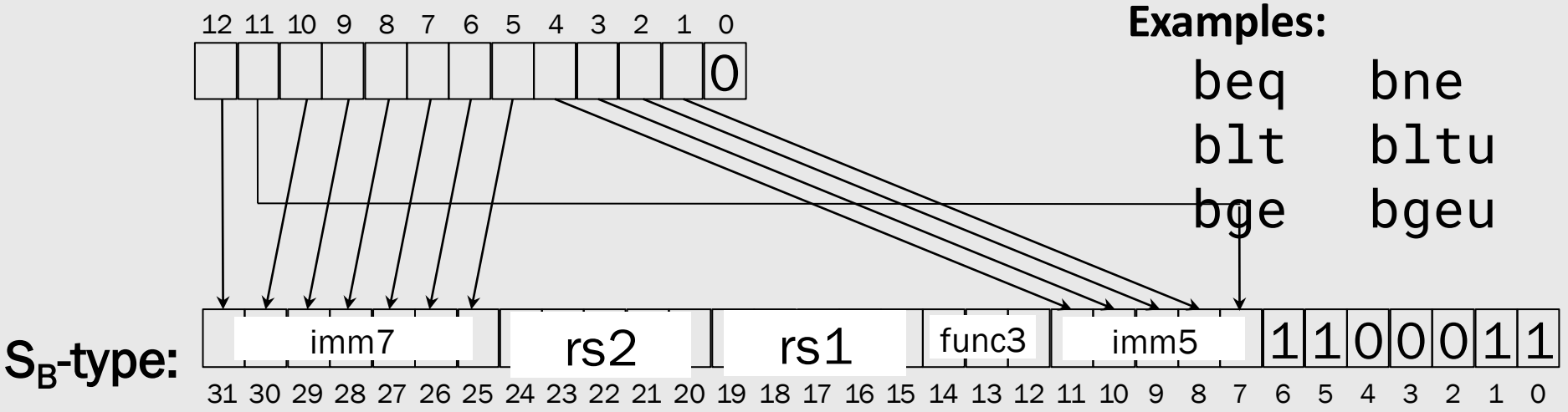
sw

The assembler will take care of filling in the immediate fields, of the instruction. It has some impact on the H/W design

data to store base addr

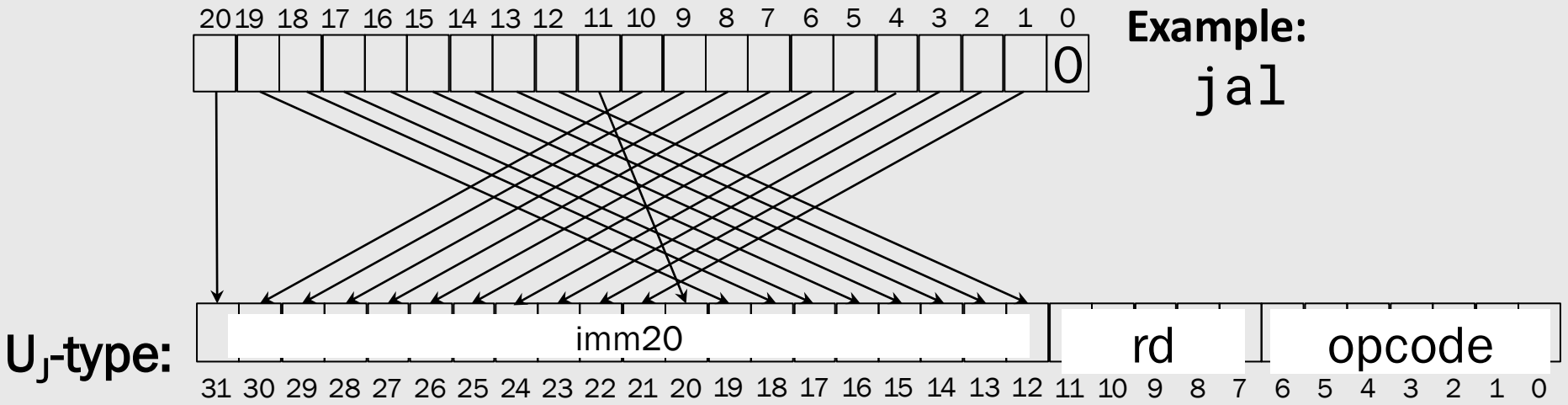
Immediate Offsets for Branches

The offsets of RISC-V branch instructions use the same format as a stores, but the ***offset encoding*** is non-obvious. Again, there is no need for a rd register; only rs1 and rs2 are used. Branch offsets can only be even. In fact, all instructions must be aligned to half-word boundaries.



Crazy Jump offsets

The offsets of RISC-V jump instructions use the same U-format as `lui/auipc`, but the ***offset encoding*** is even stranger. Jump offsets, like branch offsets can only be even.



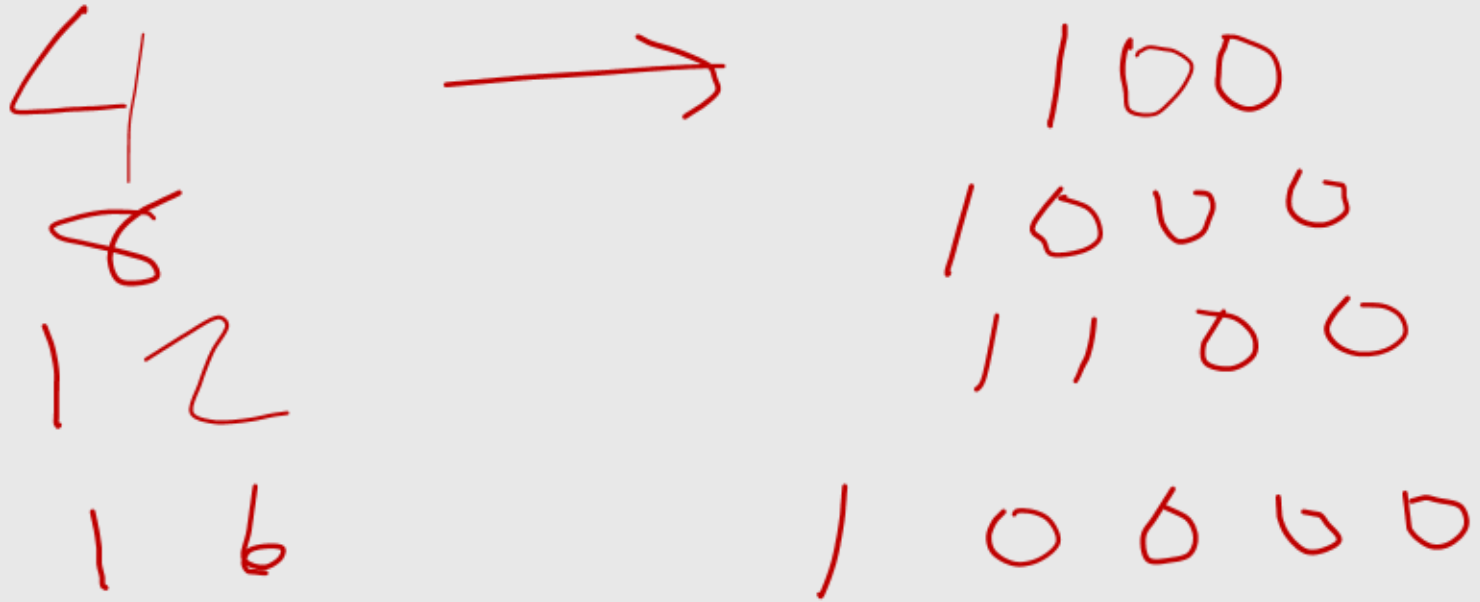
SOME REMAINING REMAINING RISC-V DETAILS…

Branch Encoding

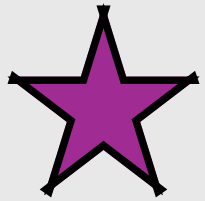
- Target instruction
  - *The instruction the program will go to if the branch is taken*
- Byte offset to target instruction
  - *The distance, in bytes, between the current branch instruction and the instruction it would go to if the branch is taken*

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.



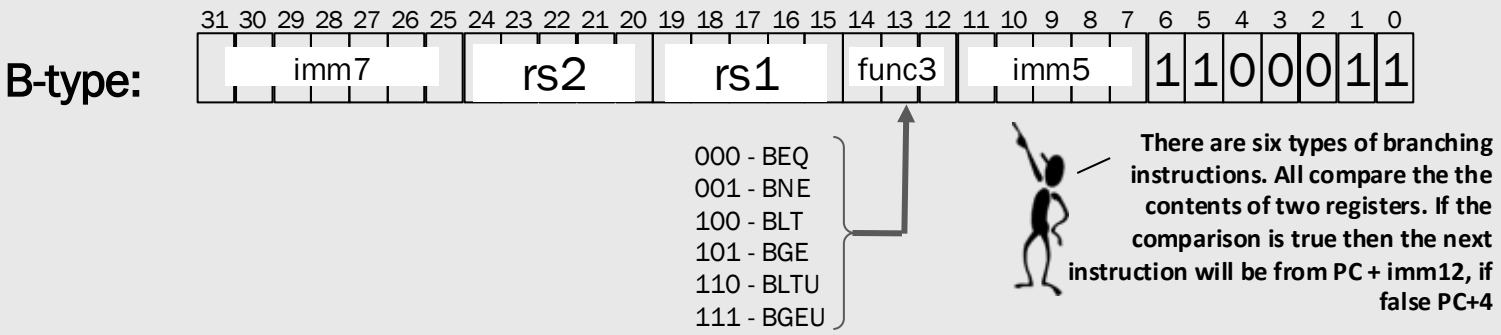
Fill in the blanks



- 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 = 0b00000*
      - *4 = 0b00100*
      - *8 = 0b01000*
      - *12 = 0b01100*

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

```
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

Above is the encoding for:

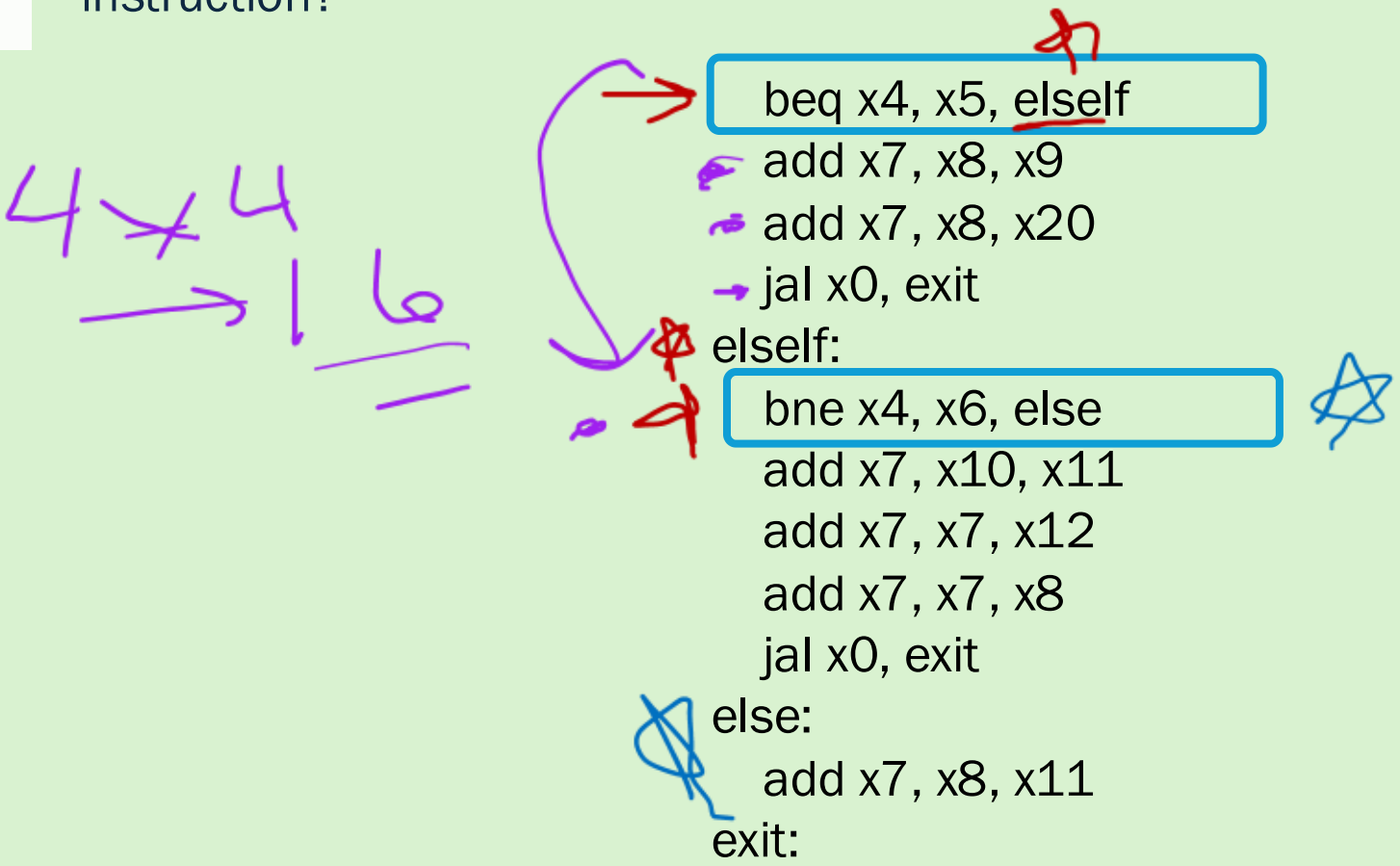
`beq x0, x0`

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 Immediate

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



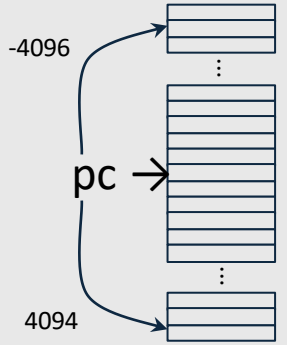
PC Relative offsets

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

```
        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

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…

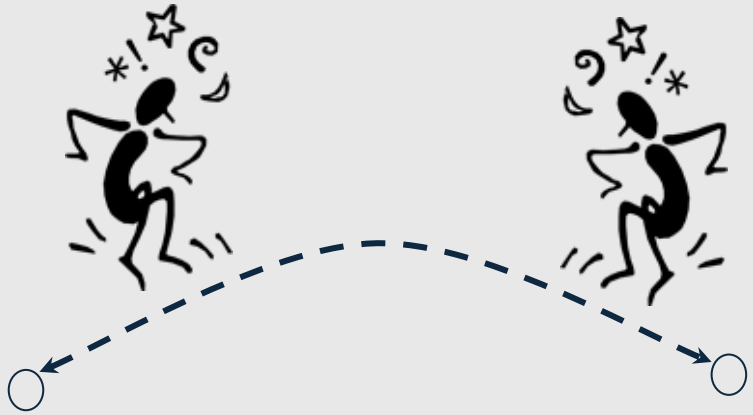
They *<u>**LINK**</u>*! Or record where to come to come back to!

1. JAL (jump and link)
2. JALR (jump and link register)

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

Why store PC + 4?

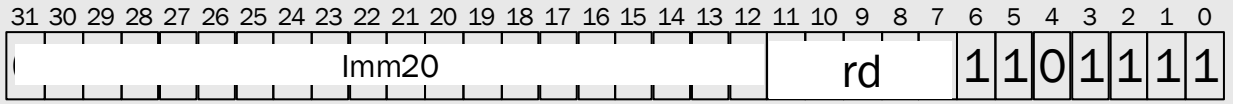
These long-distance "jump" instructions also save the address of the following instruction in a register specified by **Rd**. Often, this register will be x0, and therefore is ignored. But in other cases it is useful to get back to where we jumped from.



Jumping long distances

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

U-type:



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:

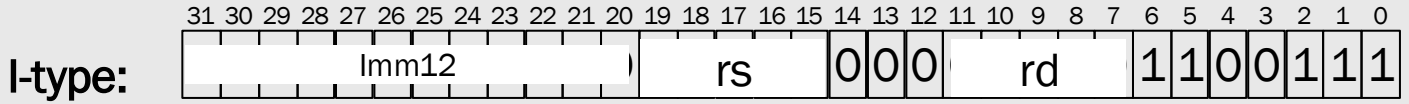
Reg[d] ← PC + 4

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.

Jumping long distances

JALR: jump anywhere by computing addr in register first.



JALR: jump and link register

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

Description:

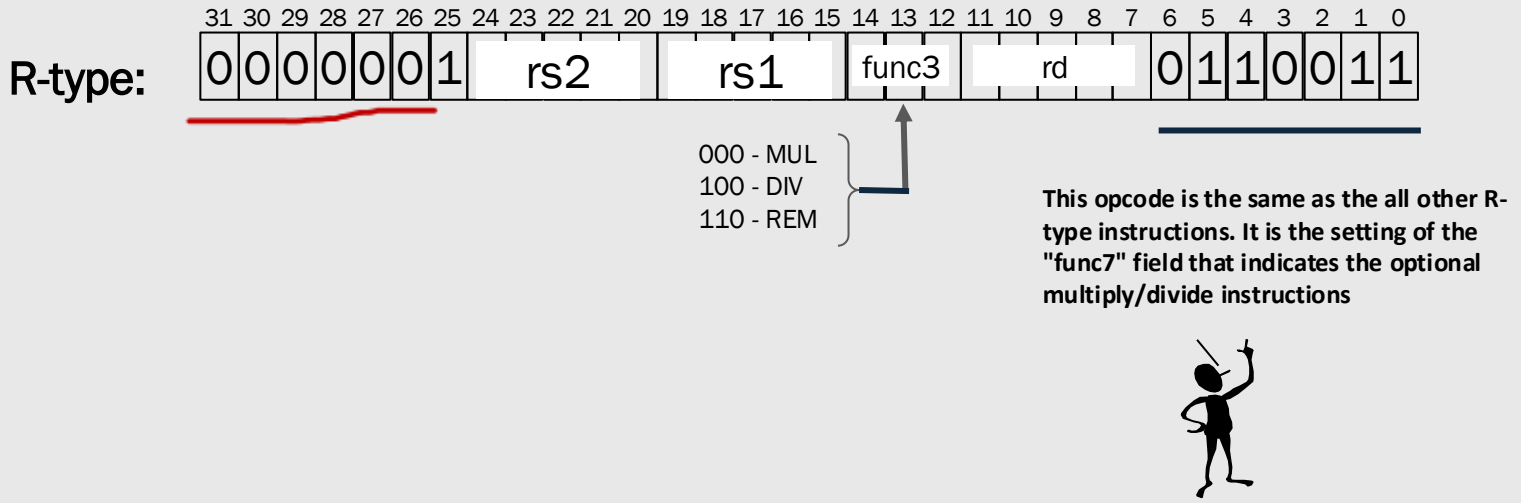
Reg[d] ← PC + 4
PC ← Reg[s] + sign_extended(imm12)

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

Use JAL when the target is known at assembly time (PC-relative),
and JALR when the target is computed at runtime (register-based).

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 s*000 *dddd d*011 0011**

Description:

Reg[d] ← Reg[s] * 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 s*100 *dddd d*011 0011**

Description:

Reg[d] ← Reg[s] / 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 s*110 *dddd d*011 0011**
Description:

Reg[d] ← Reg[s] % 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.

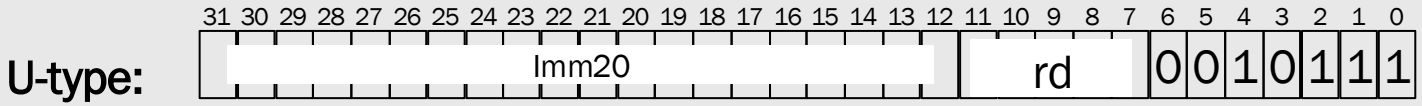
Example: `rem x7,x5,x6 # Encoded as: 0x0262E3B3`

Another instruction: AUIPC

PC-relative branch + jump instructions allow for relocatable code. However, what about relocatable “data”?

Ex: you might want to place a data structure in a code section and be able to relocate it without the need for keeping track of the absolute addresses.

RISCV fixes this problem with a PC-relative **lui-like** instruction called **auipc**




```
auipc x10,next  # encoded as 0x00000517
```

AUIPC details

This instruction gives you a PC-relative base address. Once you have it in a register (rd), you can use a load instruction with it to read memory

AUIPC: Add Upper Immediate to PC

Syntax: **auipc *rd,imm20***
Encoding: ***iiii iiii iiii iiii iiii dddd d*011 0011**
Description:

Reg[d] ← PC + sign_extend(imm20 << 12)
Adds the sign-extended 20-bit immediate value, left-shifted by 12 bits, to the program counter, and writes the result to Reg[d].

Example: **`auipc x31,1024`** **`# Encoded as: 0x00400F97`**

Also, the combination of an **auipc** instruction and a **jalr** can transfer control (jump) to any memory location. Both branch and jump instructions have limited ranges.

Load/Store offsets are only 12 bits, we can't encode large addresses directly. AUIC shifts a 20 bit immediate by 12 to get the upper 20 bits of an address near our target.

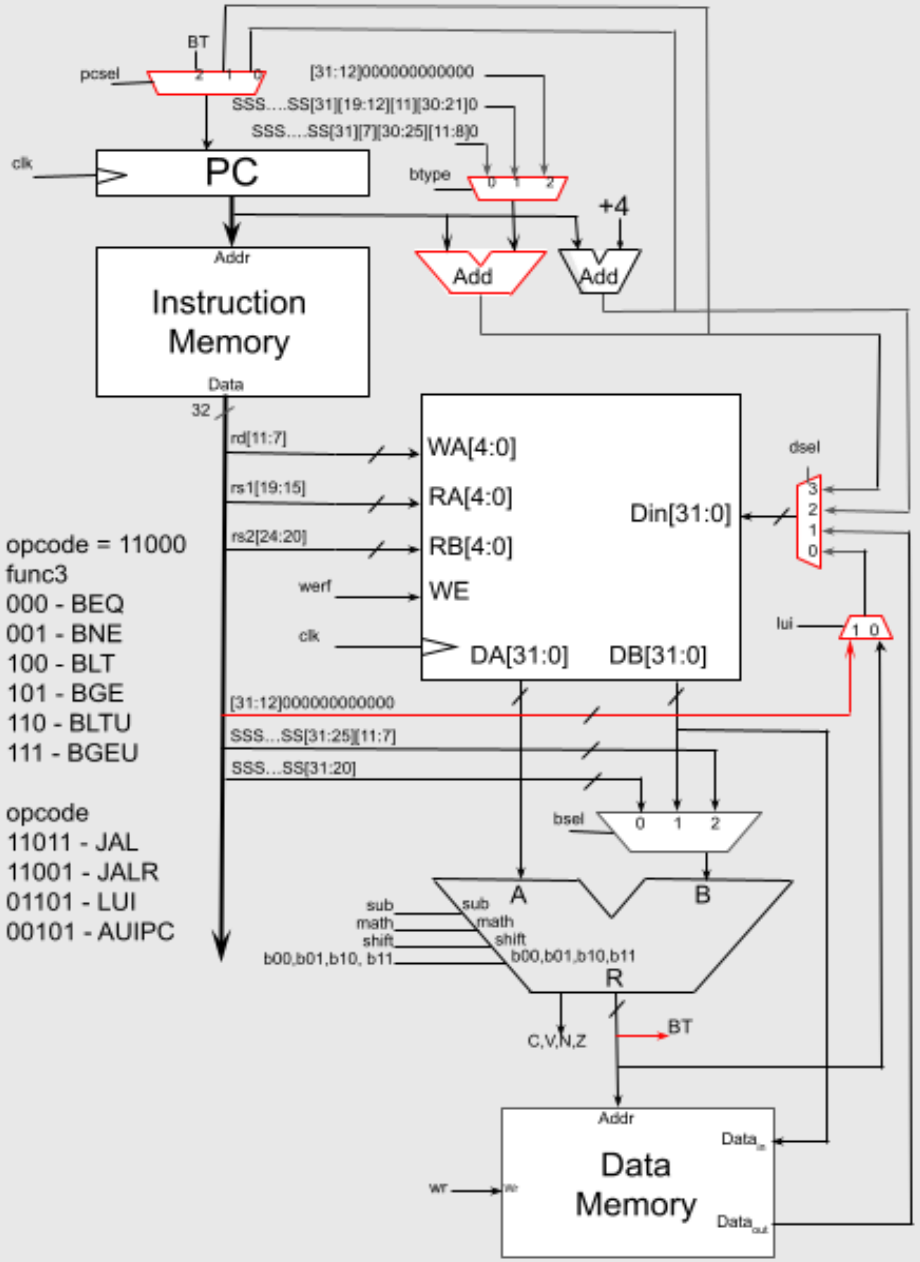
AUIPC + load offset = full 32 bit address computation

Branch, Jump, lui and auipc

Branches add a 12-bit “sign-extended” immediate value (multiplied by 2) to the current PC if taken.

Jumps always add a 20-bit sign-extended immediate value and support saving PC+4.

We also need to be able to compute the PC-relative offsets used by the AUIPC instruction, and large immediates of LUI and store them into the register file.



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);
        …
```

- Functions (procedures that return values)

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

```
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);
```

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"

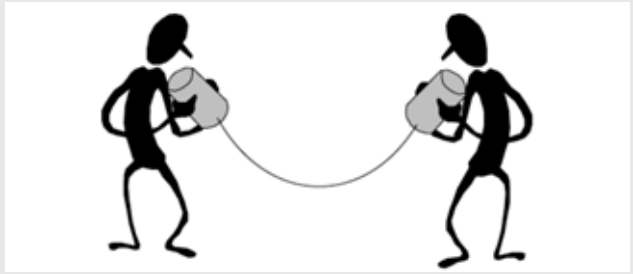
That "fee( )" seems odd to me?
And, foo( )'s a little square.

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:

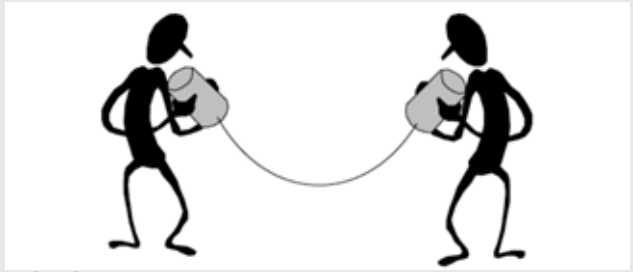
- can be called from anywhere by a caller, and, when finished, they return back to where they were called from
- can have their only local variables
- clean up behind themselves, they avoid creating unintended side-effects
- can call themselves to implement
  Recursive methods/functions



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.***

Basics of procedure calling:



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!

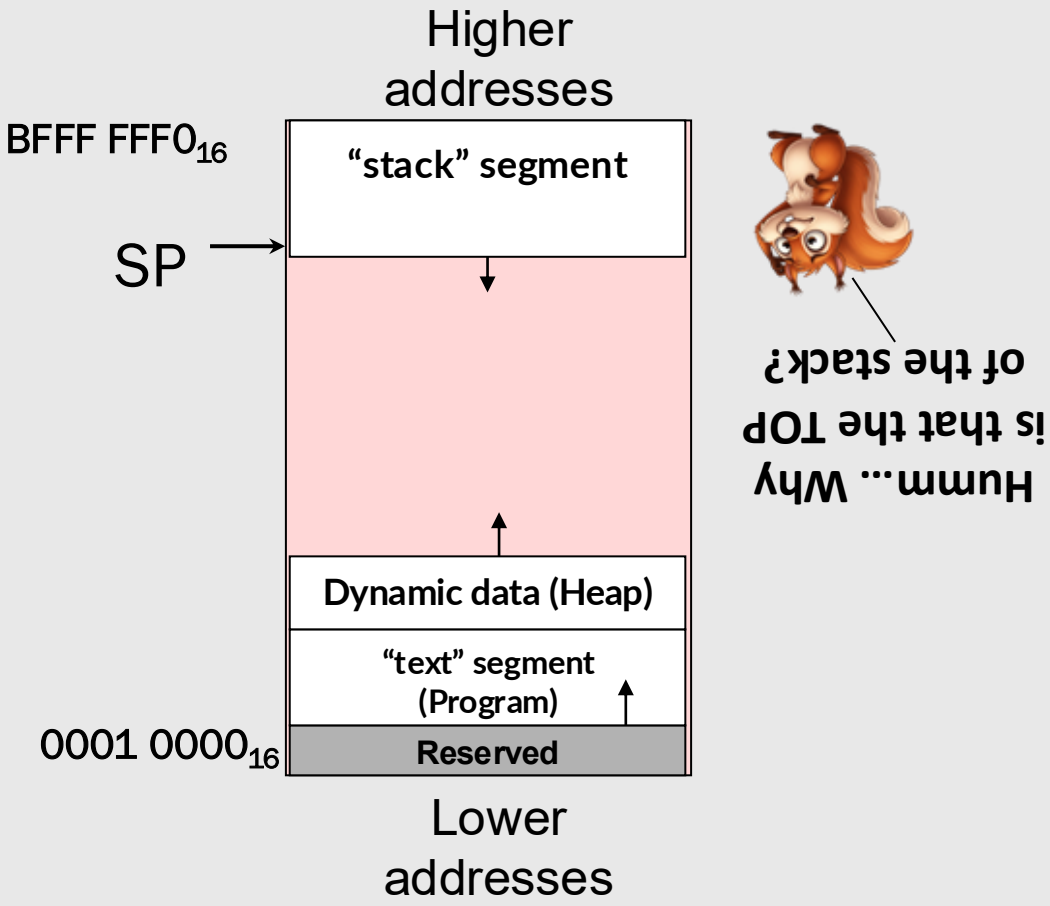
Separately written fns can safely share CPU without corrupting each other's data

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.**



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);
}
```

What happens when trace this assembly code?

```
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…

```
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

change sp ⟹ addi

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
```

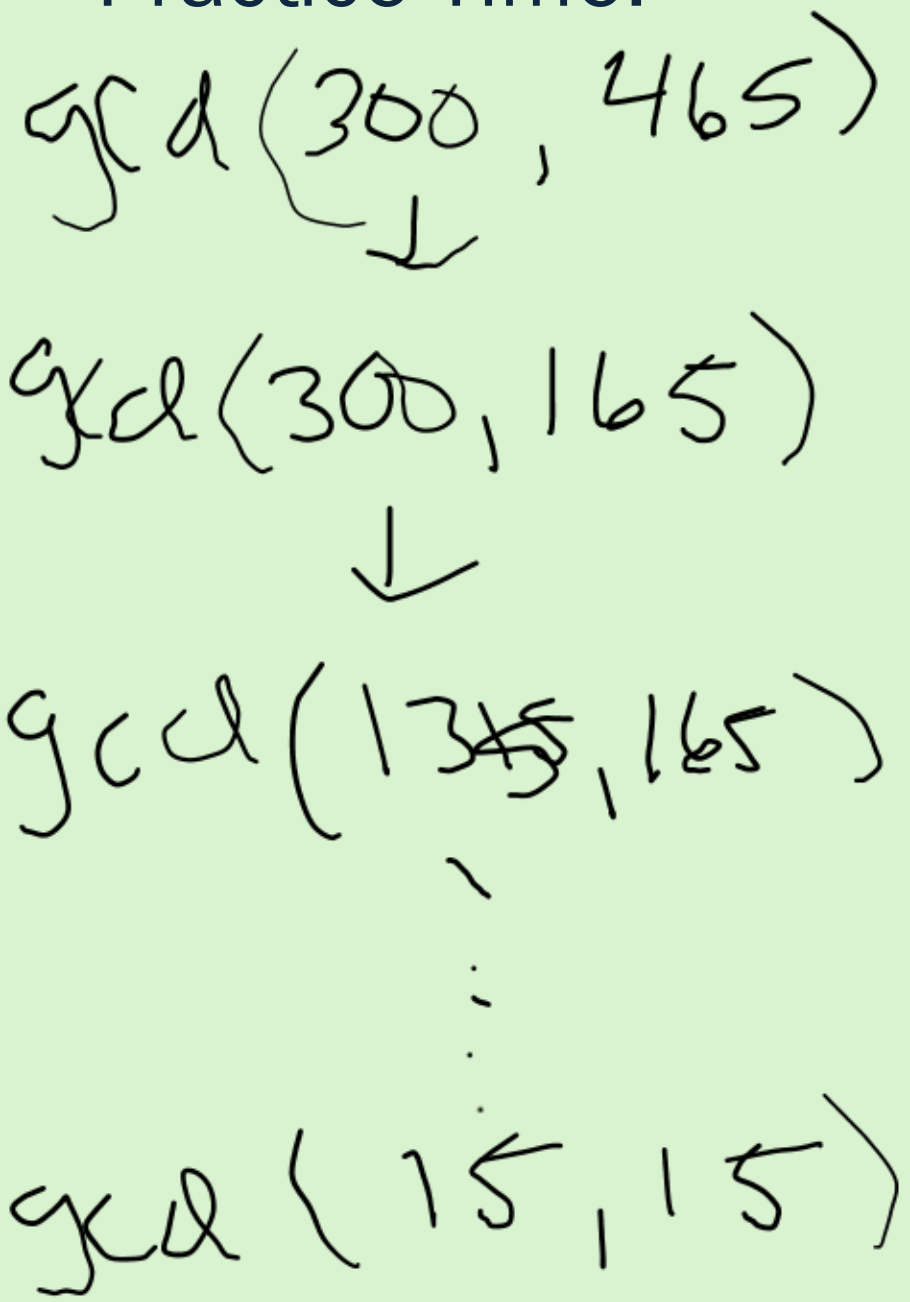
PUSH x: push Reg[x] onto stack
Mem[SP - 4] = Rx
SP = SP - 4

```
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
```

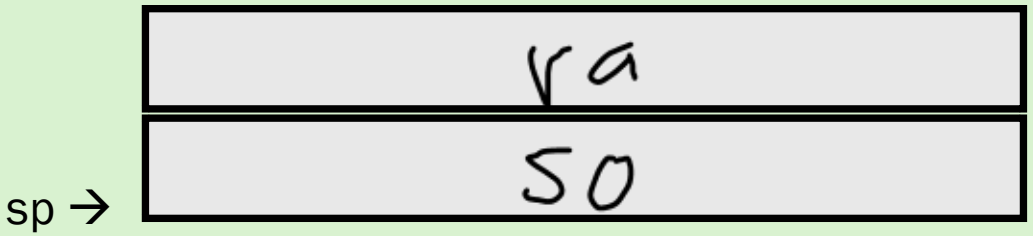
Practice Time!




```
proc1:
    addi sp, sp, -8        # allocate space for ra and s0
    sw   ra, 4(sp)         # save return address
    sw   s0, 0(sp)         # save old frame pointer
    addi s0, sp, 4         # set frame pointer

    addi sp, sp, ______    # allocate space for s1–s4

    sw   s1, ______(sp)    # save s1
    sw   s2, ______(sp)    # save s2
    sw   s3, ______(sp)    # save s3
    sw   s4, ______(sp)    # save s4

    jal  ra, proc2         # call another procedure

    lw   s1, ______(s0)    # restore s1
    lw   s2, ______(s0)    # restore s2
    lw   s3, ______(s0)    # restore s3
    lw   s4, ______(s0)    # restore s4

    addi sp, s0, 4         # restore sp
    lw   ra, 0(s0)         # restore ra
    lw   s0, -4(s0)        # restore old frame pointer
    jalr x0, 0(ra)         # return
```

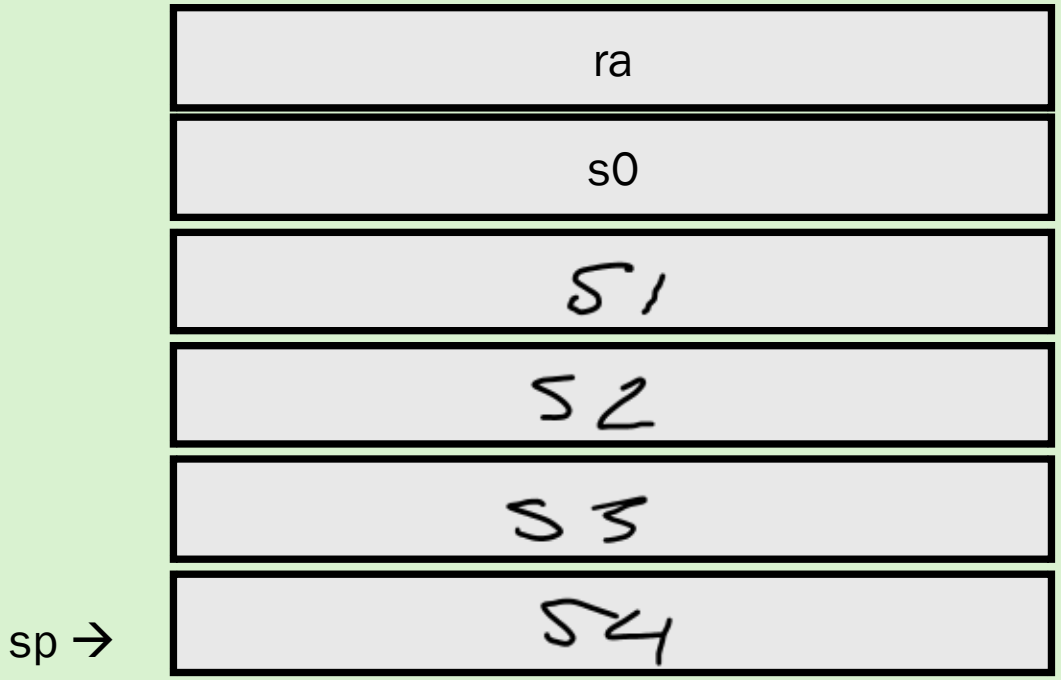
8 calls

Practice Time!




```
proc1:
   addi sp, sp, -8        # allocate space for ra and s0
   sw   ra, 4(sp)         # save return address
   sw   s0, 0(sp)         # save old frame pointer
   addi s0, sp, 4         # set frame pointer

   addi sp, sp, ______    # allocate space for s1–s4

   sw   s1, ______(sp)    # save s1
   sw   s2, ______(sp)    # save s2
   sw   s3, ______(sp)    # save s3
   sw   s4, ______(sp)    # save s4

   jal  ra, proc2         # call another procedure

   lw   s1, ______(s0)    # restore s1
   lw   s2, ______(s0)    # restore s2
   lw   s3, ______(s0)    # restore s3
   lw   s4, ______(s0)    # restore s4

   addi sp, s0, 4         # restore sp
   lw   ra, 0(s0)         # restore ra
   lw   s0, -4(s0)        # restore old frame pointer
   jalr x0, 0(ra)         # return
```

Practice Time!

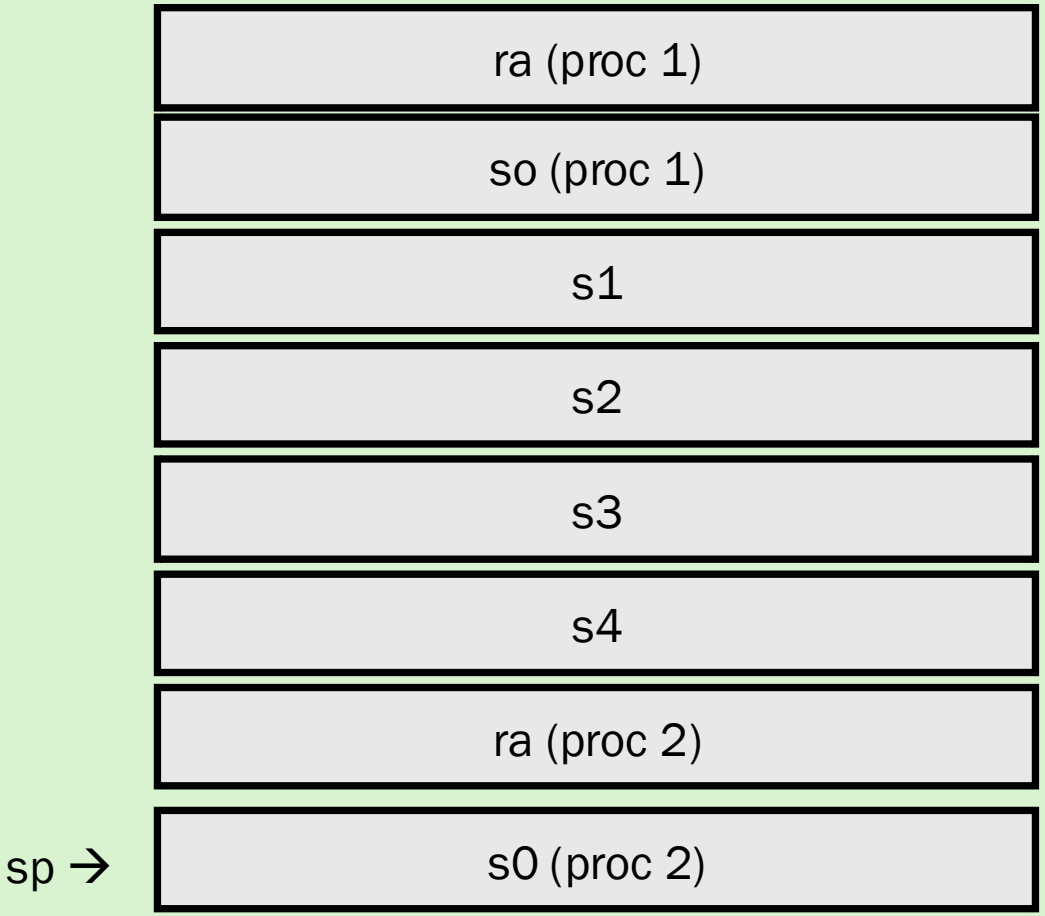



```
proc1:
   addi sp, sp, -8        # allocate space for ra and s0
   sw   ra, 4(sp)         # save return address
   sw   s0, 0(sp)         # save old frame pointer
   addi s0, sp, 4         # set frame pointer

   addi sp, sp, __-16__   # allocate space for s1–s4

   sw   s1, __12__(sp)    # save s1
   sw   s2, __8__(sp)     # save s2
   sw   s3, _____(sp)     # save s3
   sw   s4, _____(sp)     # save s4

   jal  ra, proc2         # call another procedure

   lw   s1, _____(s0)     # restore s1
   lw   s2, _____(s0)     # restore s2
   lw   s3, _____(s0)     # restore s3
   lw   s4, _____(s0)     # restore s4

   addi sp, s0, 4         # restore sp
   lw   ra, 0(s0)         # restore ra
   lw   s0, -4(s0)        # restore old frame pointer
   jalr x0, 0(ra)         # return
```

Practice Time!


```
proc1:
   addi sp, sp, -8        # allocate space for ra and s0
   sw   ra, 4(sp)         # save return address
   sw   s0, 0(sp)         # save old frame pointer
   addi s0, sp, 4         # set frame pointer

   addi sp, sp, ___16__   # allocate space for s1–s4

   sw   s1, ___12___(sp)  # save s1
   sw   s2, ___8___(sp)   # save s2
   sw   s3, ___4___(sp)   # save s3
   sw   s4, ___0___(sp)   # save s4

   jal  ra, proc2         # call another procedure

   lw   s1, _____(s0)     # restore s1
   lw   s2, _____(s0)     # restore s2
   lw   s3, _____(s0)     # restore s3
   lw   s4, _____(s0)     # restore s4

   addi sp, s0, 4         # restore sp
   lw   ra, 0(s0)         # restore ra
   lw   s0, -4(s0)        # restore old frame pointer
   jalr x0, 0(ra)         # return
```

Practice Time!




```
proc1:
   addi sp, sp, -8        # allocate space for ra and s0
   sw   ra, 4(sp)         # save return address
   sw   s0, 0(sp)         # save old frame pointer
   addi s0, sp, 4         # set frame pointer

   addi sp, sp, ___16__   # allocate space for s1–s4

   sw   s1, ___12___(sp)  # save s1
   sw   s2, ___8___(sp)   # save s2
   sw   s3, ___4___(sp)   # save s3
   sw   s4, ___0___(sp)   # save s4

   jal  ra, proc2         # call another procedure

   lw   s1, _____(s0)     # restore s1
   lw   s2, _____(s0)     # restore s2
   lw   s3, _____(s0)     # restore s3
   lw   s4, _____(s0)     # restore s4

   addi sp, s0, 4         # restore sp
   lw   ra, 0(s0)         # restore ra
   lw   s0, -4(s0)        # restore old frame pointer
   jalr x0, 0(ra)         # return
```

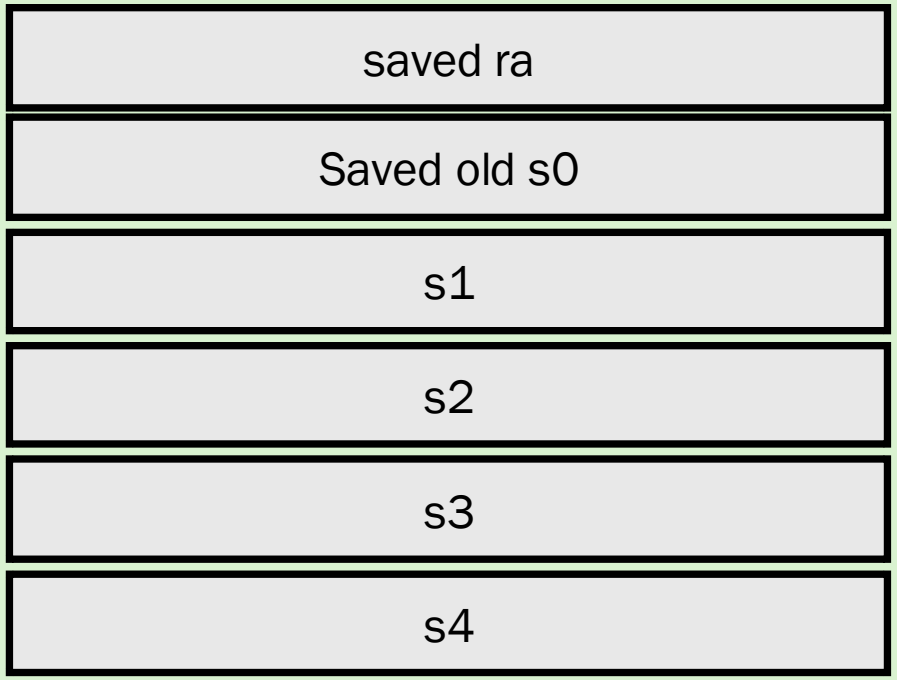
Practice Time!

-4(s0) is reserved for the old frame pointer, so saved registers start at -8(s0).

```
proc1:
    addi sp, sp, -8        # allocate space for ra and s0
    sw   ra, 4(sp)         # save return address
    sw   s0, 0(sp)         # save old frame pointer
    addi s0, sp, 4         # set frame pointer

    addi sp, sp, ___16__   # allocate space for s1–s4

    sw   s1, ___12___(sp)  # save s1
    sw   s2, ___8___(sp)   # save s2
    sw   s3, ___4___(sp)   # save s3
    sw   s4, ___0___(sp)   # save s4

    jal  ra, proc2         # call another procedure

    lw   s1, ___-8___(s0)  # restore s1
    lw   s2, ___-12___(s0) # restore s2
    lw   s3, ____-16__(s0) # restore s3
    lw   s4, ___-20___(s0) # restore s4

    addi sp, s0, 4         # restore sp
    lw   ra, 0(s0)         # restore ra
    lw   s0, -4(s0)        # restore old frame pointer
    jalr x0, 0(ra)         # return
```

Practice Time!

sp →



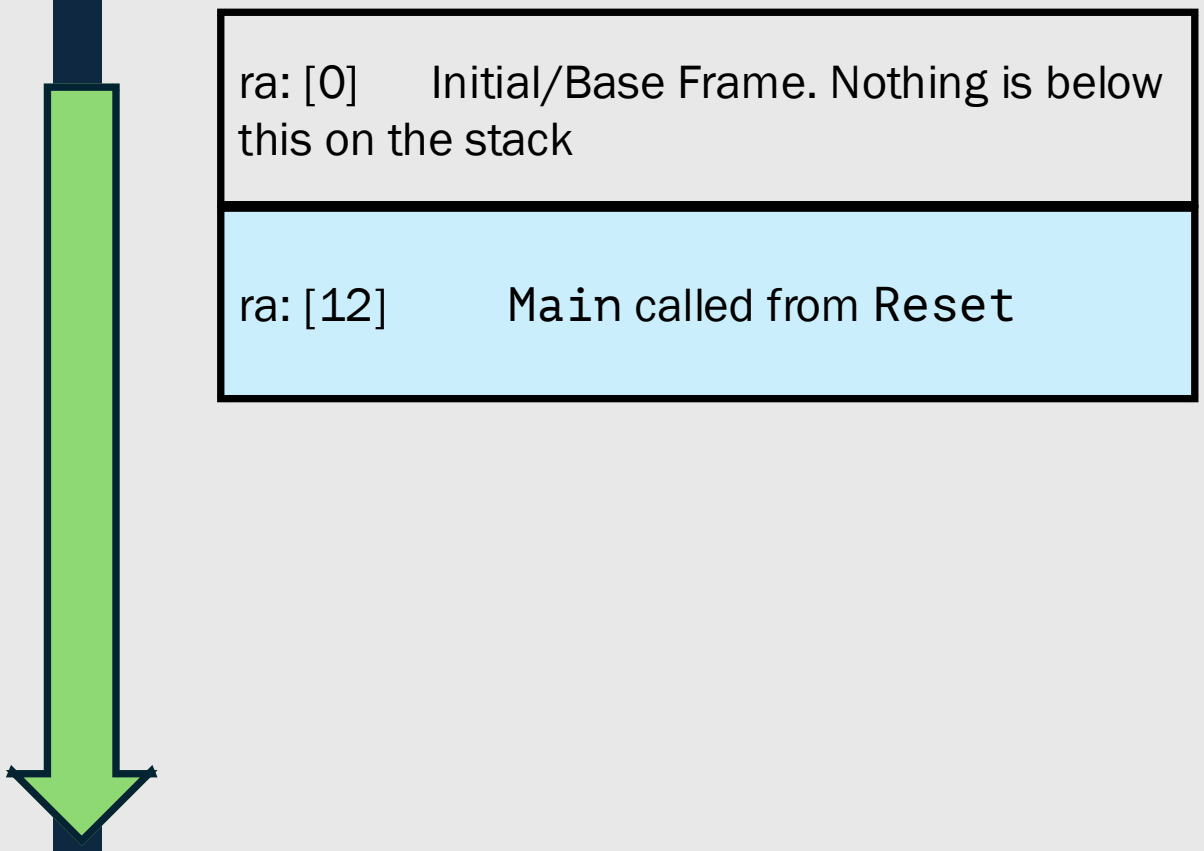

```
proc1:
   addi sp, sp, -8        # allocate space for ra and s0
   sw   ra, 4(sp)         # save return address
   sw   s0, 0(sp)         # save old frame pointer
   addi s0, sp, 4         # set frame pointer

   addi sp, sp, ___16__   # allocate space for s1–s4

   sw   s1, ___12___(sp)  # save s1
   sw   s2, ___8___(sp)   # save s2
   sw   s3, ___4___(sp)   # save s3
   sw   s4, ___0___(sp)   # save s4

   jal  ra, proc2         # call another procedure

   lw   s1, ___-8___(s0)  # restore s1
   lw   s2, ___-12___(s0) # restore s2
   lw   s3, ____-16__(s0) # restore s3
   lw   s4, ___-20___(s0) # restore s4

   addi sp, s0, 4         # restore sp
   lw   ra, 0(s0)         # restore ra
   lw   s0, -4(s0)        # restore old frame pointer
   jalr x0, 0(ra)         # return
```

Stack Tracing. Following along w/ WS Code

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

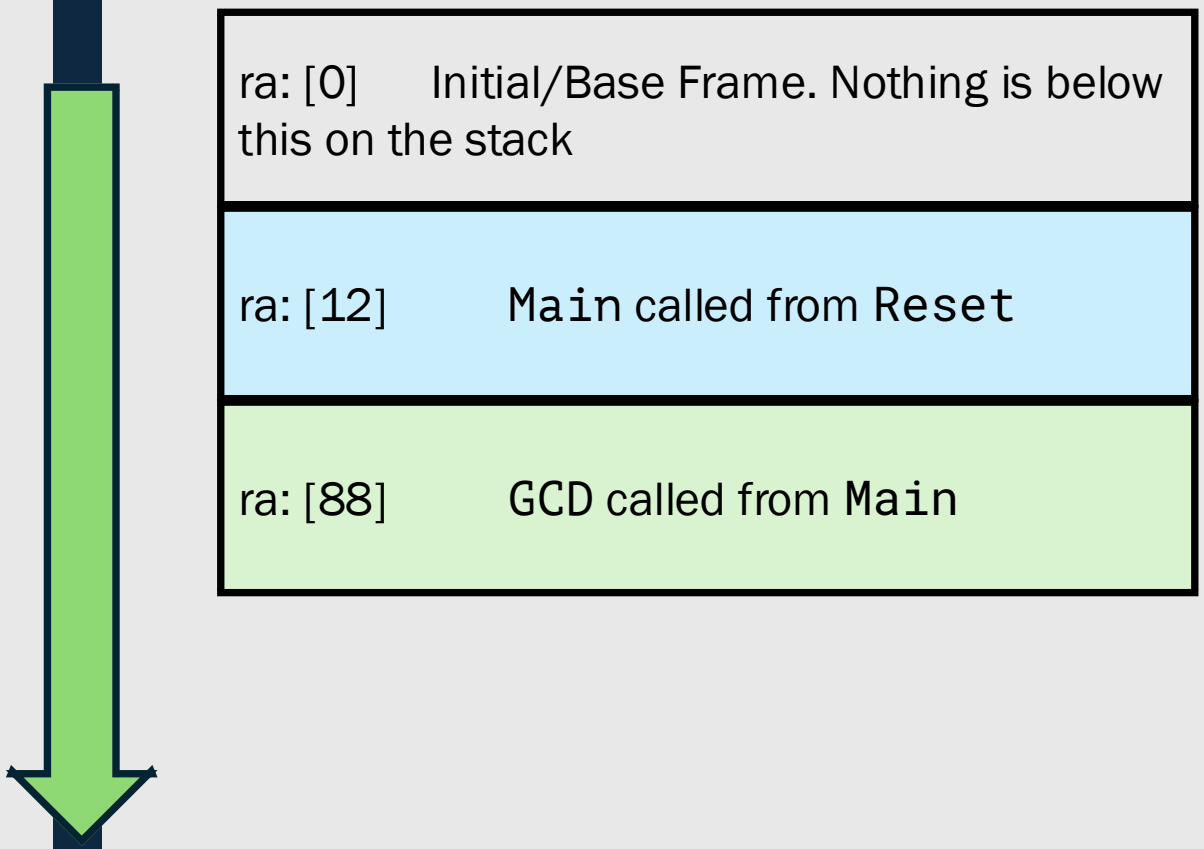
Each stack entry stores the return address, which is the address of the next instruction after JAL

Stack Tracing. Following along w/ WS Code



addr 8: `jal ra, main`

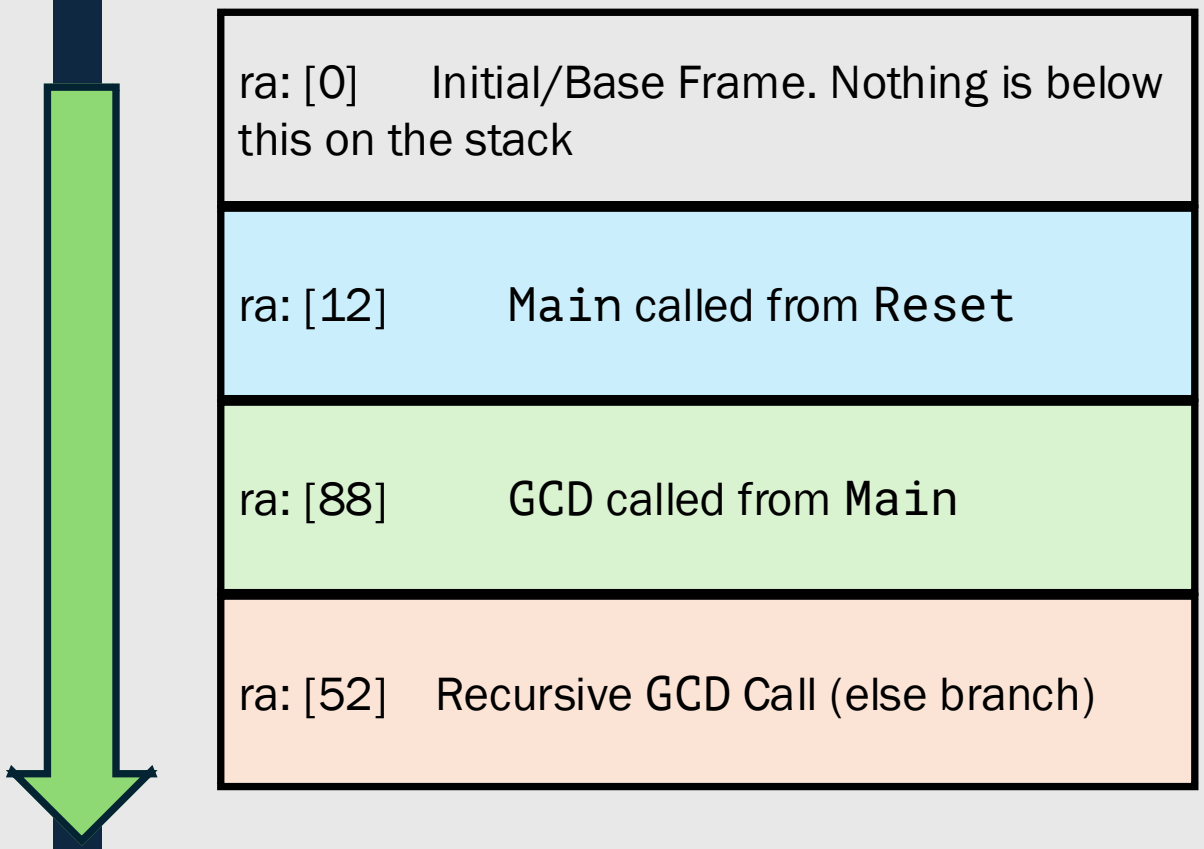
Stack Tracing. Following along w/ WS Code



addr 8: `jal ra, main`

addr 84: `jal ra, gcd`

Stack Tracing. Following along w/ WS Code

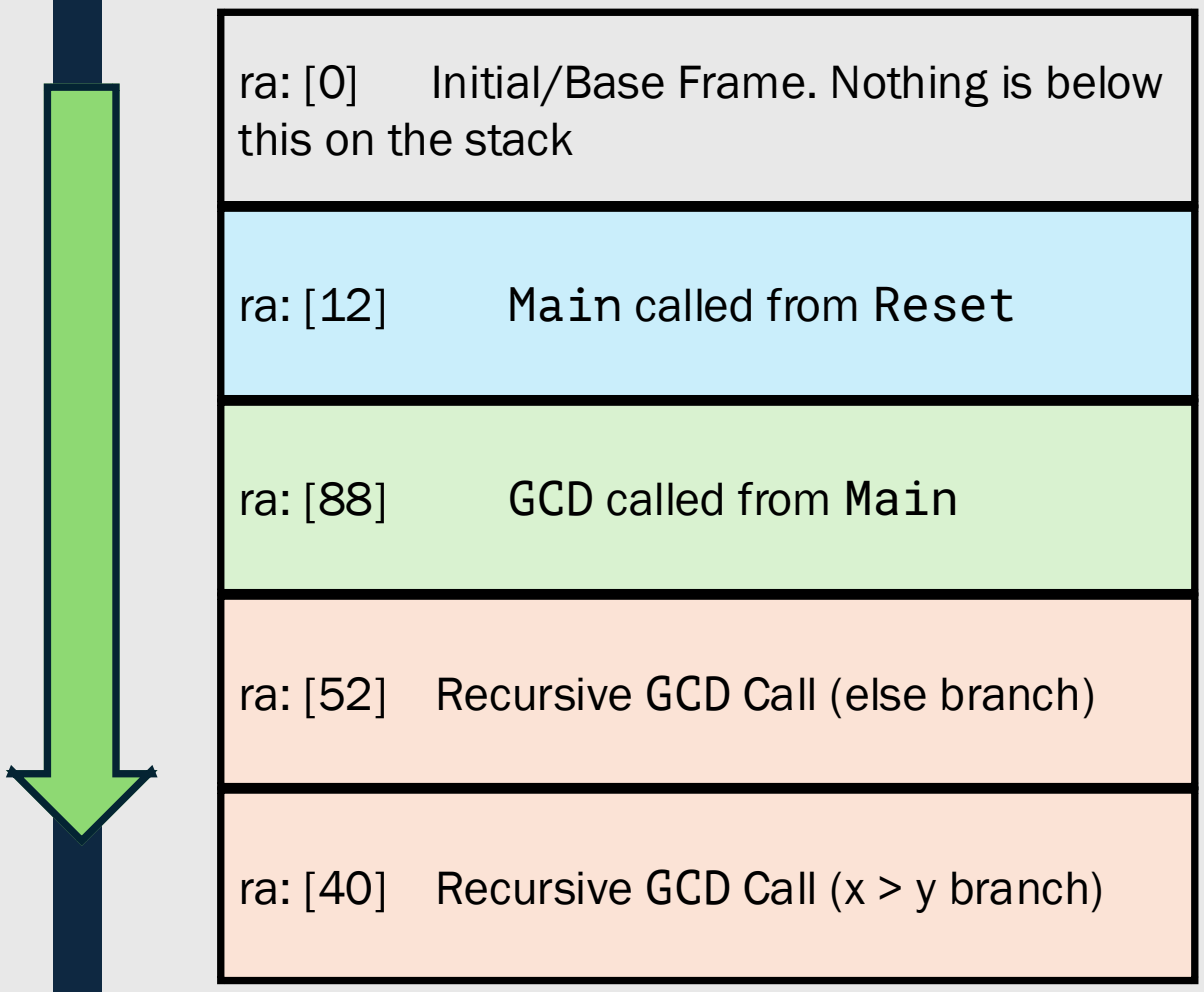


addr 8: `jal ra, main`

addr 84: `jal ra, gcd`

addr 48: `jal ra, gcd`

Stack Tracing. Following along w/ WS Code



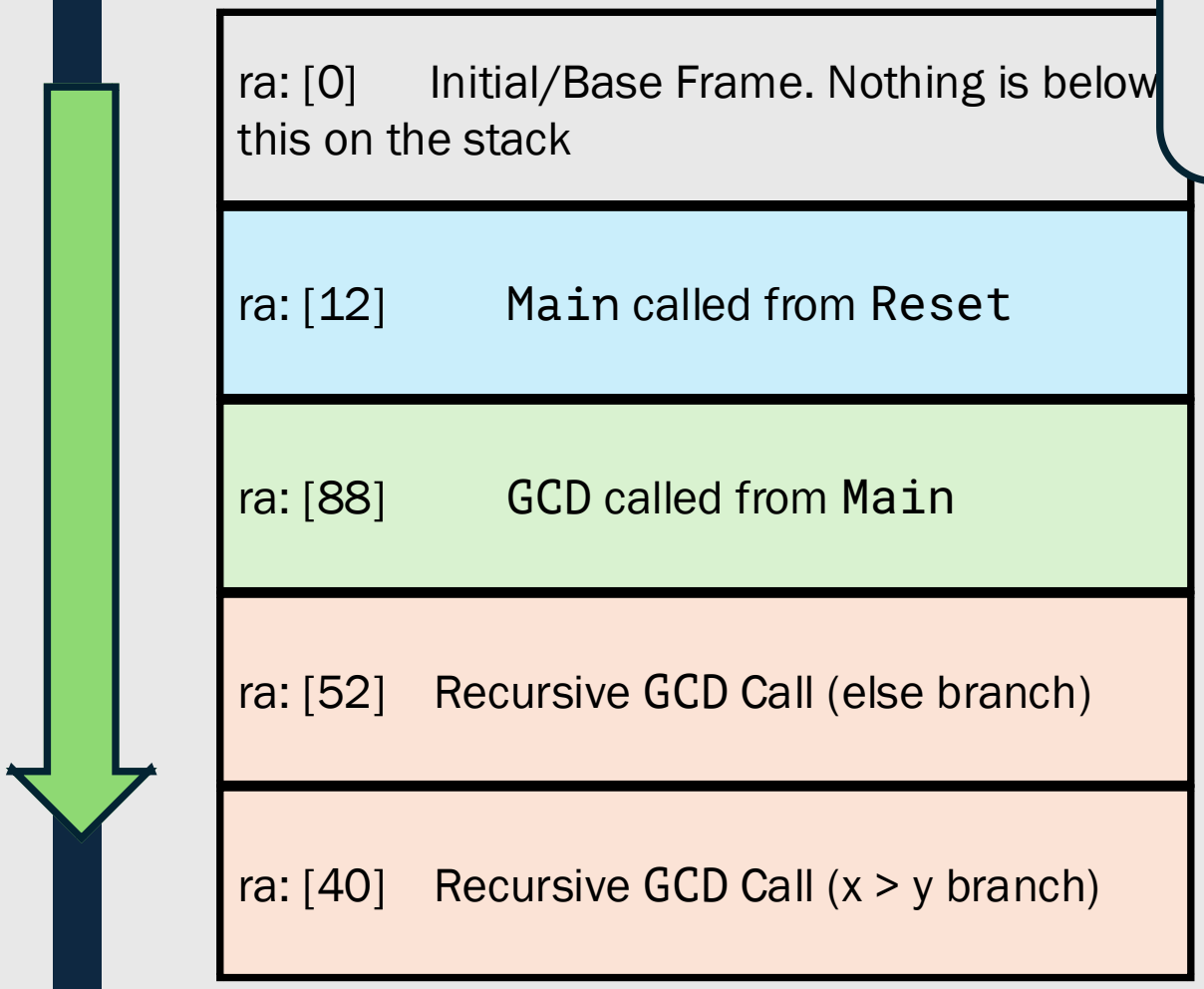
addr 8: `jal ra, main`

addr 84: `jal ra, gcd`

addr 48: `jal ra, gcd`

addr 36: `jal ra, gcd`

Stack Tracing. Following a



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`