

# ***RISC-V architecture and language***

These transparencies are based on those provided with the following book:  
David Harris and Sarah Harris, “**Digital Design and Computer Architecture, RISC-V Edition**”,  
1st Edition, 2021, Elsevier

# Assembly Language

- **Instructions:** commands in a computer's language
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1s and 0s)
- **RISC-V architecture:**
  - Developed by Krste Asanovic, David Patterson and their colleagues at UC Berkeley in 2010, starting from the initial design principles of the Reduced Instruction Set Computer (RISC) architecture proposed by D. Patterson and John Hennessy (Stanford University)
  - First widely accepted open-source computer architecture
- **RISC-V Foundation:**
  - The RISC-V Foundation ([www.riscv.org](http://www.riscv.org)) was founded in 2015 to build an open, collaborative community of software and hardware innovators based on the RISC-V ISA.
  - The Foundation, a non-profit corporation controlled by its members, directed the development to drive the initial adoption of the RISC-V ISA
  - In March 2020, the RISC-V International Association was incorporated in Switzerland.

# Assembly Language

Once you've learned one CPU architecture, it's easy to learn others !

# Architecture Design Principles

Underlying design principles, as articulated by Hennessy and Patterson:

- 1. Simplicity favors regularity**
- 2. Make the common case fast**
- 3. Smaller is faster**
- 4. Good design demands good compromises**

# Instructions: Addition

## C Code

```
a = b + c;
```

## RISC-V assembly code

```
add a, b, c
```

- **add:** mnemonic indicates operation to perform
- **b, c:** source operands (on which the operation is performed)
- **a:** destination operand (to which the result is written)

# Instructions: Subtraction

- Similar to addition - only mnemonic changes

## C Code

```
a = b - c;
```

## RISC-V assembly code

```
sub a, b, c
```

- **sub:** mnemonic
- **b, c:** source operands
- **a:** destination operand

# Design Principle 1

## **Simplicity favors regularity**

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Easier to encode and handle in hardware

# Multiple Instructions

- More complex code is handled by multiple RISC-V instructions.

## C Code

```
a = b + c - d;
```

## RISC-V assembly code

```
add t, b, c    # t = b + c  
sub a, t, d    # a = t - d
```

- What follows the ‘#’ is a **comment**



# Design Principle 2

## Make the common case fast

- RISC-V includes only simple, commonly used instructions
- Hardware to decode and execute instructions can be simple, small, and fast
- More complex operations (that are less common) are performed using multiple simple instructions
- RISC-V is a *reduced instruction set computer (RISC)*, with a small number of simple instructions
- Other architectures, such as Intel's x86, are *complex instruction set computers (CISC)*

# Operands

- Operand location: physical location in computer
  - Registers
  - Memory
  - Constants (also called *immediates*)

# Operands: Registers

- RISC-V has 32 32-bit registers
- Registers are faster than memory
- RISC-V is called “32-bit architecture” because it operates on 32-bit data

# Design Principle 3

## **Smaller is Faster**

- RISC-V includes only a small number of registers

# RISC-V Register Set

Name	Register Number	Usage
<b>zero</b>	x0	Constant value 0
<b>ra</b>	x1	Return address
<b>sp</b>	x2	Stack pointer
<b>gp</b>	x3	Global pointer
<b>tp</b>	x4	Thread pointer
<b>t0-2</b>	x5-7	Temporaries
<b>s0/fp</b>	x8	Saved register / Frame pointer
<b>s1</b>	x9	Saved register
<b>a0-1</b>	x10-11	Function arguments / return values
<b>a2-7</b>	x12-17	Function arguments
<b>s2-11</b>	x18-27	Saved registers
<b>t3-6</b>	x28-31	Temporaries

# Operands: Registers

- **Registers:**
  - Can use either name (i.e., `ra`, `zero`) or `x0`, `x1`, etc.
  - Using name is preferred
- Registers used for **specific purposes**:
  - `zero` always holds the **constant value 0**.
  - the ***saved registers***, `s0–s11` are used to hold variables
  - the ***temporary registers***, `t0–t6` are used to hold intermediate values during a larger computation
  - Discuss others later

# Instructions with Registers

- Revisit add instruction

## C Code

```
a = b + c
```

## RISC-V assembly code

```
# t0 = a, t1 = b, t2 = c
```

```
add t0, t1, t2
```

# Instructions with Constants

- `addi` instruction

## C Code

```
a = b + 6;
```

## RISC-V assembly code

```
# t0 = a, t1 = b  
addi t0, t1, 6
```

One operand corresponds to an immediate,  
instead of a register



# Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables kept in registers

# Reading Byte-Addressable Memory

- The address of a memory word is always a multiple of 4. For example,
  - the address of memory word #2 is  $2 \times 4 = 8$
  - the address of memory word #10 is  $10 \times 4 = 40$  (0x28)
- **RISC-V is byte-addressed**

Address	Data	
:	:	:
0000000C	4 0 F 3 0 7 8 8	Word 3
00000008	0 1 E E 2 8 4 2	Word 2
00000004	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

# Reading Memory

- Memory read called *load*
- **Mnemonic:** *load word* ( $lw$ )
- **Format:**  
 $lw\ t1,\ 4(s0)$   
 $lw\ destination,\ offset(base)$
- **Address calculation:**
  - add *base address* ( $s0$ ) to the *offset* (4)
  - $address = (s0 + 4)$
- **Result:**
  - $t1$  holds the data value at address  $(s0 + 4)$

**Any register** may be used as base address

# Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 8 into `s3`
- `s3` holds the value `0x1EE2842` after load

## RISC-V assembly code

```
lw s3, 8(zero) # read word at address 8 into s3
```

Byte Address				Word Address	Data	Word Number
⋮				⋮	⋮	⋮
13	12	11	10	00000010	C D 1 9 A 6 5 B	Word 4
F	E	D	C	0000000C	4 0 F 3 0 7 8 8	Word 3
B	A	9	8	00000008	0 1 E E 2 8 4 2	Word 2
7	6	5	4	00000004	F 2 F 1 A C 0 7	Word 1
3	2	1	0	00000000	A B C D E F 7 8	Word 0
MSB				width = 4 bytes		
LSB						

# Load halfword and byte

- **Signed:**

- Load halfword: `lh`
- Load byte: `lb`
- Sign-extends to create 32-bit value to load into register

- **Unsigned:**

- Load halfword unsigned: `lhu`
- Load byte: `lbu`
- Zero-extends to create 32-bit value

# Writing Memory

- Memory write are called *store*
- **Mnemonic:** *store word* (SW)

# Writing Memory

- **Example:** store the value held in  $t7$  into memory address  $0x10$  (16)
  - if  $t7$  holds the value  $0xAABBCCDD$ , then after the  $sw$  completes, word 4 (at address  $0x10$ ) in memory will contain that value

## RISC-V assembly code

```
sw t7, 0x10(zero) # write t7 into address 16
```

Offset can be written in **decimal** (default) or **hexadecimal**

Byte Address				Word Address	Data	Word Number
⋮				⋮	⋮	⋮
13	12	11	10	<b>00000010</b>	<b>A A B B C C D D</b>	<b>Word 4</b>
F	E	D	C	0000000C	4 0 F 3 0 7 8 8	<b>Word 3</b>
B	A	9	8	00000008	0 1 E E 2 8 4 2	<b>Word 2</b>
7	6	5	4	00000004	F 2 F 1 A C 0 7	<b>Word 1</b>
3	2	1	0	00000000	A B C D E F 7 8	<b>Word 0</b>
MSB		LSB			width = 4 bytes	

# Byte-Addressable Memory

- Each data byte has unique address
- Load/store words or single bytes: load byte (lb) and store byte (sb)
- 32-bit word = 4 bytes, so word address increments by 4

Address	Data	
⋮	⋮	⋮
0000000C	4 0 F 3 0 7 8 8	Word 3
00000008	0 1 E E 2 8 4 2	Word 2
00000004	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

width = 4 bytes

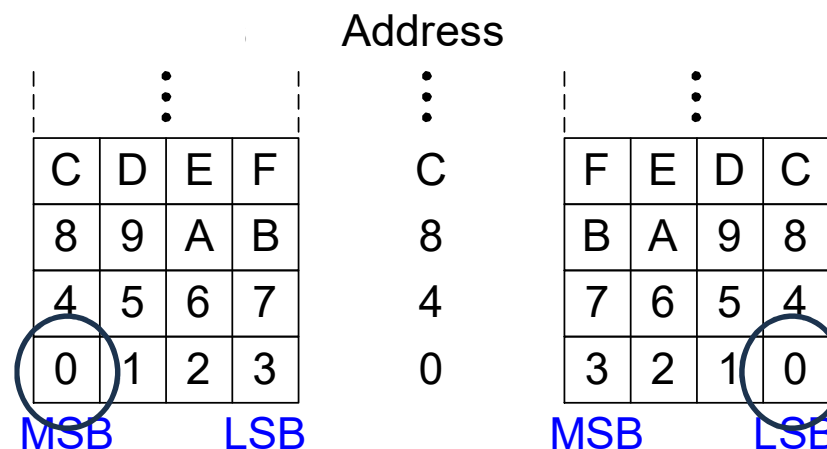


# Big-Endian & Little-Endian Memory

- How to number bytes within a word?
- **Little-endian:** byte numbers start at the little (least significant) end
- **Big-endian:** byte numbers start at the big (most significant) end
- **Word address** is the **same** for big- or little-endian

Big-Endian

Little-Endian



# Big-Endian & Little-Endian Memory

- Jonathan Swift's *Gulliver's Travels*: the Little-Endians broke their eggs on the little end of the egg and the Big-Endians broke their eggs on the big end
- It doesn't really matter which addressing type used – except when the two systems need to share data!

## Big-Endian

⋮			
C	D	E	F
8	9	A	B
4	5	6	7
0	1	2	3
MSB		LSB	

## Little-Endian

Address

⋮			
F	E	D	C
B	A	9	8
7	6	5	4
3	2	1	0
MSB		LSB	

# Big-Endian & Little-Endian Example

- Suppose `t0` initially contains `0x23456789`
- After following code runs on big-endian system, what value is `s0`?

```
sw t0, 0(zero)
lb s0, 1(zero)
```

# Big-Endian & Little-Endian Example

- Suppose `t0` initially contains `0x23456789`
- After following code runs on big-endian system, what value is `s0`?

```
sw t0, 0(zero)
lb s0, 1(zero)
```

- Big-endian: `0x00000045`
- Little-endian: `0x00000067`

## Big-Endian

Byte Address	0	1	2	3
Data Value	23	45	67	89
	MSB		LSB	

## Little-Endian

Address	3	2	1	0	Byte Address
0	23	45	67	89	Data Value
	MSB		LSB		

# Operands: Constants/Immediates

- `lw` and `sw` use constants or *immediates*
- *immediately* available from instruction
- 12-bit two's complement number
- `addi`: add immediate
- Subtract immediate (`subi`) necessary?

## C Code

```
a = a + 4;  
b = a - 12;  
c = -372;
```

## RISC-V assembly code

```
# s0 = a, s1 = b, s2 = c  
addi s0, s0, 4  
addi s1, s0, -12  
addi s2, zero, -372
```

# Generating 32-bit constants

- Use load upper immediate (`lui`) and `addi`
- `lui`: puts an immediate in the upper 20 bits of destination register and 0's in lower 12 bits

## C Code

```
int a = 0xFEDC8765;
```

## RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC8
addi s0, s0, 0x765
```

`addi` **sign-extends** its 12-bit immediate

# Generating 32-bit constants

- If **bit 11** of 32-bit constant is **1**, increment upper 20 bits by **1** in `lui`

## C Code

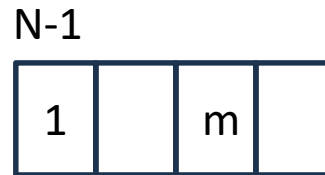
```
int a = 0xFEDC8EAB;
```

**Note:** -341 = 0xEAB

## RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC9      # s0 = 0xFEDC9000
addi s0, s0, -341     # s0 = 0xFEDC9000 + 0xFFFFFEAB
                        #      = 0xFEDC8EAB
```

# Demonstration



- Let consider 2 identical numbers with the same number of bits starting with 1 (MSB):
  - A: a signed number:  $-2^{n-1} + m$
  - B: an unsigned number:  $2^{n-1} + m$
  
- Which is the difference between them (A-B)?
  - $-2^{n-1} + m - (2^{n-1} + m) = -2 * 2^{n-1} = -2^n$



## Demonstration (II)

- The original value (unsigned) on 12 bits became signed and its value is **decremented** by  $2^{12}$
- To correct this alteration the upper 20 bits are **incremented** by **1** in `lui`

# Pseudo-instructions

- **Pseudoinstructions** are not actual RISC-V instructions but they are often more convenient for the programmer
- Assembler converts them to real RISC-V instructions (one or more)

# Pseudo-instruction: `li`

- `li`: Load immediate
- `li t0, 4`  
translates to
- `addi t0, zero, 4`
- but what should  
`li $t0, 0x000001800`  
`li $t0, -5`  
translate to?

## Load immediate (II)

- So

```
li    t0, 0x00001800
```

- translates to

```
lui   t0, 0x00002    #load upper 20 bits  
                        # + 1
```

```
addi  t0, t0, 0x800   #load lower 12 bits
```

# Assembler Directives

- Assembler directives guide the assembler in allocating and initializing global variables, defining constants, and differentiating between code and data
- Directives are features of the assembler tool, not of the processor ISA!

# Assembler Directives: a subset

Directive	Description
<b>.section</b> name	section named name
<b>.text</b>	text section
<b>.data</b>	global Data Section
<b>.rodata</b>	read-only Data Section
<b>.bss</b>	global data section initialized to 0
<b>.globl</b> symbol_name	emit symbol_name to symbol table (scope GLOBAL)
<b>.local</b> symbol_name	emit symbol_name to symbol table (scope LOCAL)
<b>.string</b> "str"	store string "str" in memory
<b>.asciz</b> "str"	store string "str" in memory followed by a 0 byte
<b>.equ</b> name, constant	define symbol name with value constant
<b>.align</b> N	align next N data/instruction on 2 <sup>N</sup> -byte boundary
<b>.byte</b> b1, b2, ..., bN	store N 8-bit values in successive memory words
<b>.half</b> h1, h2, ..., hN	store N 16-bit values in successive memory words
<b>.word</b> w1, w2, ..., wN	store N 32-bit values in successive memory words
<b>.dword</b> d1, d2, ..., dN	store N 64-bit values in successive memory words
<b>.space</b> N	reserve N bytes to store variables
<b>.end</b>	end of assembly code

# Logical Instructions

- **and, or, xor**
  - and: useful for **masking** bits
    - Masking all but the least significant *byte* of a value:  
 $0xF234012F \text{ AND } 0x000000FF = 0x0000002F$
  - or: useful for **combining** bit fields
    - Combine  $0xF2340000$  with  $0x000012BC$ :  
 $0xF2340000 \text{ OR } 0x000012BC = 0xF23412BC$

# Logical Instructions Example 1

Source Registers

s1	0100 0110	1010 0001	1111 0001	1011 0111
s2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly Code

Result

and s3, s1, s2	s3	0100 0110	1010 0001	0000 0000	0000 0000
or s4, s1, s2	s4	1111 1111	1111 1111	1111 0001	1011 0111
xor s5, s1, s2	s5	1011 1001	0101 1110	1111 0001	1011 0111



# Logical Instructions

- **andi, ori, xori**
  - 12-bit immediate is *sign-extended*

# Logical operation NOT


- **xori** useful for **inverting** bits:
  - $A \text{ XOR } -1 = \text{NOT } A$  (remember that  $-1 = 0xFFFFFFFF$ )

```
xori s0, t0, 0xFFF
```

```
xori s0, t0, -1
```

# Logical Instructions Example 2

## Source Values

t3	0011	1010	0111	0101	0000	1101	0110	1111
imm	1111	1111	1111	1111	1111	1010	0011	0100
								

## Assembly Code

```
andi s5, t3, -1484
ori  s6, t3, -1484
xori s7, t3, -1484
```

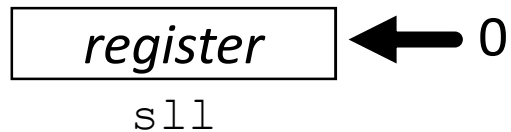
## Result

s5	0011	1010	0111	0101	0000	1000	0010	0100
s6	1111	1111	1111	1111	1111	1111	0111	1111
s7	1100	0101	1000	1010	1111	0111	0101	1011

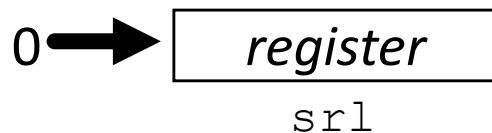
-1484 = **0xA34** in 12-bit 2's complement representation.

# Logical Shift Instructions

- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by  $i$  bits multiplies by  $2^i$

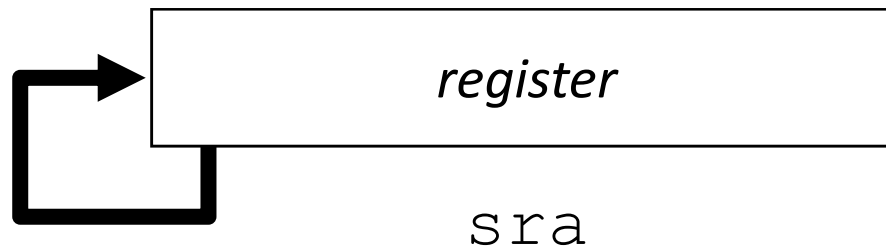


- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by  $i$  bits divides by  $2^i$  (unsigned only)



# Arithmetic Shift Instructions

- Right shifts can be either
  - logical: (0's shift into the most significant bits)
  - arithmetic: the sign bit shifts into the most significant bits
- `sra` by  $i$  bits divides by  $2^i$  (unsigned and signed)



# Shift Instructions

**Shift amount** is in (lower 5 bits of) a register

- `sll`: shift left logical
  - **Example:** `sll t0, t1, t2 # t0 = t1 << t2`
- `srl`: shift right logical
  - **Example:** `srl t0, t1, t2 # t0 = t1 >> t2`
- `sra`: shift right arithmetic
  - **Example:** `sra t0, t1, t2 # t0 = t1 >>> t2`

# Immediate Shift Instructions

**Shift amount** is an immediate between 0 to 31

- **slli**: shift left logical immediate
  - **Example:** `slli t0, t1, 23 # t0 = t1 << 23`
- **srli**: shift right logical immediate
  - **Example:** `srli t0, t1, 18 # t0 = t1 >> 18`
- **srai**: shift right arithmetic immediate
  - **Example:** `srai t0, t1, 5 # t0 = t1 >>> 5`

# Multiplication

- $32 \times 32$  multiplication, 64-bit result

**mul s0, s1, s2**

s0 = lower 32 bits of result

**mulh s0, s1, s2**

s0 = upper 32 bits of result, treats operands as signed

**mulhu s0, s1, s2**

s0 = upper 32 bits of result, treats operands as unsigned

**mulhsu s0, s1, s2**

s0 = upper 32 bits of result, treats s1 as signed, s2 as unsigned



# Multiplication

32 × 32 multiplication → 64 bit result

**mul s3, s1, s2**

s3 = lower 32 bits of result

**mulh s4, s1, s2**

s4 = upper 32 bits of result, treats operands as signed

{s4, s3} = s1 × s2

**Example:** s1 = 0x40000000 =  $2^{30}$ ; s2 = 0x80000000 =  $-2^{31}$

s1 × s2 =  $-2^{61}$  = 0xE0000000 00000000

s4 = 0xE0000000; s3 = 0x00000000

# Multiplication

- **Signed:** `mulh`
- **Unsigned:** `mulhu`, `mulhsu`
  - `mulhu`: treat both operands as unsigned
  - `mulhsu`: treat first operand as signed, second as unsigned
  - lower 32 bits are identical whether signed/unsigned; use `mul`

Example: `s1 = 0x80000000`; `s2 = 0xC0000000`

<code>mulh s4, s1, s2</code>	<code>mulhu s4, s1, s2</code>	<code>mulhsu s4, s1, s2</code>
<code>mul s3, s1, s2</code>	<code>mul s3, s1, s2</code>	<code>mul s3, s1, s2</code>

$s1 = -2^{31}$ ;  $s2 = -2^{30}$

$s1 \times s2 = 2^{61}$

`s4 = 0x20000000`

`s3 = 0x00000000`

$s1 = 2^{31}$ ;  $s2 = 3 \times 2^{30}$

$s1 \times s2 = 3 \times 2^{61}$

`s4 = 0x60000000`

`s3 = 0x00000000`

$s1 = -2^{31}$ ;  $s2 = 3 \times 2^{30}$

$s1 \times s2 = -3 \times 2^{61}$

`s4 = 0xA0000000`

`s3 = 0x00000000`

# Division

32-bit division → 32-bit quotient & remainder

- `div s3, s1, s2` #  $s3 = s1 / s2$
- `rem s4, s1, s2` #  $s4 = s1 \% s2$

Example:  $s1 = 0x00000011 = 17$ ;  $s2 = 0x00000003 = 3$

$$s1 / s2 = 5$$

$$s1 \% s2 = 2$$

$$s3 = 0x00000005; s4 = 0x00000002$$

# Division

- **Signed:** `div, rem`
- **Unsigned:** `divu, remu`

# Programming

- High-level languages:
  - e.g., C, Java, Python
  - Written at higher level of abstraction
- Common high-level software constructs:
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

# Branching

- Execute instructions out of sequence
- Types of branches:
  - **Conditional**
    - branch if equal (`beq`)
    - branch if not equal (`bne`)
    - branch if less than (`blt`)
    - branch if greater than or equal (`bge`)
  - **Unconditional**
    - jump and link (`j al`)
    - jump and link register (`j al r`)

# Branching: conditional jumps

- Syntax

B<condition> <reg1>, <reg2>, <target\_label>

**beq** <reg1>, <reg2>, <target\_label>

**bne** <reg1>, <reg2>, <target\_label>

**blt** <reg1>, <reg2>, <target\_label>

**bge** <reg1>, <reg2>, <target\_label>

**bltu** <reg1>, <reg2>, <target\_label>

**bgeu** <reg1>, <reg2>, <target\_label>

- The instruction

- reads the registers
- evaluates the condition
- if the condition is verified, jumps to the target label; otherwise, it continues with the next instruction

# Branching: conditional jumps

**bgt**, **bgtu**, **ble**, and **bleu** can be implemented by reversing the operands to **blt**, **bltu**, **bge**, and **bgeu** respectively

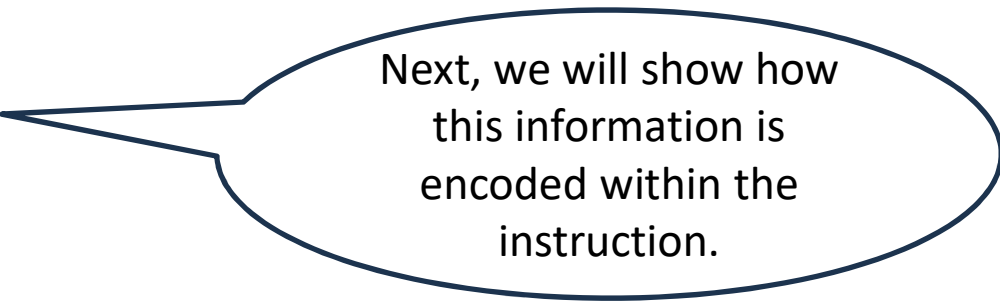
**A > B equivalent to B < A    => blt/bltu**

**A <= B equivalent to B >= A   => bge/bgeu**



# Branch calculation

- If the instruction does not take the branch (i.e., the condition is not verified):
  - $PC = PC + 4$
- If the instruction takes the branch (i.e., the condition is verified):
  - $PC = \text{<target\_label>}$



Next, we will show how this information is encoded within the instruction.

# Conditional Branching (beq)

## Branch if equal

```
addi s0, zero, 4      # s0 = 0 + 4 = 4  
addi s1, zero, 1      # s1 = 0 + 1 = 1  
slli s1, s1, 2         # s1 = 1 << 2 = 4
```

```
beq s0, s1, target    # s0 = s1? YES, branch is taken  
addi s1, s1, 1         # not executed  
sub  s1, s1, s0        # not executed
```

```
target:                # label  
  add s1, s1, s0       # s1 = 4 + 4 = 8
```

# Conditional Branching (bne)

## Branch if NOT equal

```
addi    s0, zero, 4      #  $s0 = 0 + 4 = 4$ 
addi    s1, zero, 1      #  $s1 = 0 + 1 = 1$ 
slli    s1, s1, 2        #  $s1 = 1 \ll 2 = 4$ 
```

```
bne      s0, s1, target  #  $s0 \neq s1$ ? NO, branch not taken
```

```
addi     s1, s1, 1       # executed:  $s1 = 4 + 1 = 5$ 
```

```
sub      s1, s1, s0      # executed:  $s1 = 5 - 4 = 1$ 
```

target:

```
add      s1, s1, s0      #  $s1 = 1 + 4 = 5$ 
```

# Conditional Branching (blt)

## Branch if LESS than

```
addi    s0, zero, 3      # s0 = 0 + 3 = 3
addi    s1, zero, 1      # s1 = 0 + 1 = 1
slli    s1, s1, 2        # s1 = 1 << 2 = 4
```

```
blt      s0, s1, target  # s0 < s1? YES, branch is taken
```

```
addi    s1, s1, 1        # not executed
sub      s1, s1, s0       # not executed
```

target:

```
add      s1, s1, s0      # s1 = 4 + 3 = 7
```

# Conditional Branching (bge)

## Branch if GREATER than or EQUAL

```
addi    s0, zero, 3           # s0 = 0 + 4 = 3
addi    s1, zero, 1           # s1 = 0 + 1 = 1
slli    s1, s1, 2             # s1 = 1 << 2 = 4
```

```
bge     s0, s1, target        # s0 >= s1? NO, branch not taken
addi    s1, s1, 1             # executed: s1 = 4 + 1 = 5
sub     s1, s1, s0             # executed: s1 = 5 - 3 = 2
```

```
target:
    add  s1, s1, s0           # s1 = 2 + 4 = 6
```

# Unconditional Jump

- Syntax:

`j <target_label>`

It jumps to `<target_label>` unconditionally.

- **This is not an instruction**, but it is a pseudo-instruction
- We will see later how it is converted

# High-Level Code Constructs

- `if` statements
- `if/else` statements
- `while` loops
- `for` loops

# If statement

## C Code

```
if (i == j)
    f = g + h;

f = f - i;
```

## RISC-V assembly code

```
# s0 = f, s1 = g, s2 = h
# s3 = i, s4 = j
        bne s3, s4, L1
        add s0, s1, s2

L1: sub s0, s0, s3
```

Assembly tests opposite case ( $i \neq j$ ) of high-level code ( $i == j$ )



# If/Else statement

## C Code

```
if (i == j)
    f = g + h;
else
    f = f - i;
```

## RISC-V assembly code

```
# s0 = f, s1 = g, s2 = h
# s3 = i, s4 = j
        bne s3, s4, L1
        add s0, s1, s2
        j    done
L1:     sub s0, s0, s3
done:  ...
```

# If/Else statement (II)

## C Code

```
if (i != j)
    f ++;
else
    f --;
```

## RISC-V assembly code

```
# s0 = f
# s3 = i, s4 = j

        beq s3, s4, L1
        addi s0, s0, 1
        j    done
L1:     addi s0, s0, -1
done:
```

# While loops

## C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x    = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

## RISC-V assembly code

```
# s0 = pow, s1 = x

        addi    s0, zero, 1
        add     s1, zero, zero
        addi    t0, zero, 128
while:   beq     s0, t0, done
        slli    s0, s0, 1
        addi    s1, s1, 1
        j       while
done:
```

Assembly tests for the opposite case (`pow == 128`) of the C code (`pow != 128`).

# For loops

```
for (initialization; condition; loop operation)  
    statement
```

- **initialization**: executes before the loop begins
- **condition**: is tested at the beginning of each iteration
- **loop operation**: executes at the end of each iteration
- **statement**: executes each time the condition is met

# For loops

## C Code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

## RISC-V assembly code

```
# s0 = i, s1 = sum
        add  s1, zero, zero
        add  s0, zero, zero
        addi t0, zero, 10
for:     beq  s0, t0, done
        add  s1, s1, s0
        addi s0, s0, 1
        j    for
done:
```

# Magnitude comparison

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - `if (rs < rt) rd = 1; else rd = 0;`
- Used in combination with **beq**, **bne**
  - `slt t0, s1, s2`      `# if (s1 < s2)`
  - `bne t0, zero, L`      `# branch to L`

# Less Than Comparison

## C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
    sum = sum + i;
}
```

## RISC-V assembly code

```
# s0 = i, s1 = sum
    addi s1, zero, 0
    addi s0, zero, 1
    addi t0, zero, 101

loop:  slt   t2, s0, t0
       beq  t2, zero, done
       add  s1, s1, s0
       slli s0, s0, 1
       j    loop
```

done:

**t2 = 1 if i < 101**

# Less Than Comparison: Version 2

## C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
    sum = sum + i;
}
```

## RISC-V assembly code

```
# s0 = i, s1 = sum
    addi s1, zero, 0
    addi s0, zero, 1
    addi t0, zero, 101

loop: bge s0, t0, done
    add  s1, s1, s0
    slli s0, s0, 1
    j    loop

done:
```



# If-then-else: <

## C Code

```
int a, b, c;
```

```
if (a < b)
```

```
{
```

```
    c = 5;
```

```
}
```

```
else
```

```
{
```

```
    c = 8;
```

```
}
```

## RISC-V assembly code

```
# t1 = a, t2 = b, t3 = c
```

```
main:
```

```
    addi      t1, zero, 6
```

```
    addi      t2, zero, 5
```

```
    slt t4, t1, t2      # if (a < b)
```

```
    beq t4, zero, L1    # else
```

```
    addi t3, zero, 5    # then
```

```
    j      L2
```

```
L1: addi t3, zero, 8    # else
```

```
L2:
```

# If-then-else: <

# Version 2

## C Code

```
int a, b, c;
```

```
if (a < b)
```

```
{
```

```
    c = 5;
```

```
}
```

```
else
```

```
{
```

```
    c = 8;
```

```
}
```

## RISC-V assembly code

```
# t1 = a, t2 = b, t3 = c
```

```
main:
```

```
    addi      t1, zero, 6
```

```
    addi      t2, zero, 5
```

```
    bge t1, t2, L1      # if (a < b)
```

```
    # else
```

```
    addi t3, zero, 5    # then
```

```
    j      L2
```

```
L1: addi t3, zero, 8    # else
```

```
L2:
```

# If-then-else: >

## C Code

```
int a, b, c;
```

```
if (a > b)
```

```
{
```

```
    c = 5;
```

```
}
```

```
else
```

```
{
```

```
    c = 8;
```

```
}
```

## RISC-V assembly code

```
# t1 = a, t2 = b, t3 = c
```

```
main:
```

```
    addi      t1, $0, 6
```

```
    addi      t2, $0, 5
```

```
    slt t4, t2, t1
```

```
    beq t4, zero, L1
```

```
    addi t3, zero, 5
```

```
    j      L2
```

```
L1: addi t3, zero, 8
```

```
L2:
```

```
# if (b < a)
```

```
# else
```

```
# then
```

```
# else
```

# If-then-else: $\geq$

## C Code

```
int a, b, c;

if (a  $\geq$  b)
{
    c = 5;
}
else /* a < b */
{
    c = 8;
}
```

## RISC-V assembly code

```
# t1 = a, t2 = b, t3 = c
main:
    addi      t1, zero, 6
    addi      t2, zero, 5

    slt t4, t1, t2      # if (a < b)
    beq t4, zero, L1

    addi t3, zero, 8     # (a < b)
    j     L2

L1: addi t3, zero, 5     # (a  $\geq$  b)

L2:
```

# If-then-else: >=

## Version 2

### C Code

```
int a, b, c;

if (a >= b)
{
    c = 5;
}
else /* a < b */
{
    c = 8;
}
```

### RISC-V assembly code

```
# t1 = a, t2 = b, t3 = c
main:
    addi    t1, zero, 6
    addi    t2, zero, 5

    blt     t1, t2, L1      # if (a < b)

    addi    t3, zero, 5     # (a >= b)
    j       L2

L1: addi    t3, zero, 8     # (a < b)
    j       L2

L2:
```

# If-then-else: <=

## C Code

```
int a, b, c;

if (a <= b)
{
    c = 5;
}
else /* a > b */
{
    c = 8;
}
```

## RISC-V assembly code

```
# $t1 = a, $t2 = b, $t3 = c
main:
    addi    $t1, $0, 6
    addi    $t2, $0, 5

    slt     $t4, $t2, $t1 # if (b < a)
    beq     $t4, $0, L1

    addi    $t3, $0, 8    # (b < a)
    j       L2

L1: addi    $t3, $0, 5    # (a <= b)

L2:
```

# Branches

- **Signed:**            `blt, bge`
- **Unsigned:**       `bltu, bgeu`

**Examples:** `s1 = 0x80000000; s2 = 0x40000000`

`blt s1, s2`

`s1 = -231; s2 = 230`

taken

`bltu s1, s2`

`s1 = 231; s2 = 230`

not taken

# Set Less Than

- **Signed:** `slt, slti`
- **Unsigned:** `sltu, sltiu`

**Note:** RISC-V always **sign-extends** the immediate, even for `sltiu`

**Examples:** `s1 = 0x80000000; s2 = 0x40000000`

`slt t0, s1, s2`

`s1 = -231; s2 = 230`

`t0 = 1`

`slti t2, s1, -1 # -1 = 0xFFF`

`s1 = -231; imm = 0xFFFFFFFF = -1`

`t2 = 1`

`sltu t1, s1, s2`

`s1 = 231; s2 = 230`

`t1 = 0`

`sltiu t3, s1, -1 # -1 = 0xFFF`

`s1 = 231; imm = 0xFFFFFFFF = 232 - 1`

`t3 = 1`



# Detecting Overflow

- RISC-V does not provide unsigned addition or instructions for overflow detection because it can be done with existing instructions:

- **Example: Detecting unsigned overflow:**

```
add  t0, t1, t2
bltu t0, t1, overflow
```

- **Example: Detecting signed overflow:**

```
add  t0, t1, t2
slt  t3, t2, zero      # t3=1 if t2 neg.
slt  t4, t0, t1        # t4=1 if result < t1
bne  t3, t4, overflow  # overflow if:
                        # t2 neg & result>=t1 or
                        # t2 pos & result<t1
```

# Arrays

- Access large amounts of similar data
- **Index**: access each element
- **Size**: number of elements

# Arrays

- 5-element array
- **Base address** = 0x123B4780 (address of first element, `array[0]`)
- First step in accessing an array: load base address into a register

Address	Data
123B4790	<code>array[4]</code>
123B478C	<code>array[3]</code>
123B4788	<code>array[2]</code>
123B4784	<code>array[1]</code>
123B4780	<code>array[0]</code>

Main Memory

# Accessing Arrays

## // C Code

```
int array[5];  
array[0] = array[0] * 2;  
array[1] = array[1] * 2;
```

## # RISC-V assembly code

# s0 = array base address

```
lui    s0, 0x123B4           # 0x1234 in upper 20 bits of s0  
addi   s0, s0, 0x780         # s0 = 0x123B4780
```

```
lw      t1, 0(s0)            # t1 = array[0]  
slli    t1, t1, 1            # t1 = t1 * 2  
sw      t1, 0(s0)            # array[0] = t1
```

```
lw      t1, 4(s0)            # t1 = array[1]  
sll     t1, t1, 1            # t1 = t1 * 2  
sw      t1, 4(s0)            # array[1] = t1
```

# Arrays Using For Loops

**// C Code**

```
int array[1000];  
int i;  
  
for (i=0; i < 1000; i = i + 1)  
    array[i] = array[i] * 8;
```

**# RISC-V assembly code**

```
# s0 = array base address, s1 = i
```

# Arrays Using For Loops

## # RISC-V assembly code

```
# s0 = array base address, s1 = i
# initialization code
lui    s0, 0x23B8F          # s0 = 0x23B8F000
ori    s0, s0, 0x400        # s0 = 0x23B8F400
addi   s1, zero, 0          # i = 0
addi   t2, zero, 1000       # t2 = 1000

loop:
    bge  s1, t2, done        # if s1 >= t2 then done
    slli t0, s1, 2           # t0 = i * 4 (byte offset)
    add  t0, t0, s0          # address of array[i]
    lw   t1, 0(t0)           # t1 = array[i]
    slli t1, t1, 3           # t1 = array[i] * 8
    sw   t1, 0(t0)           # array[i] = array[i] * 8
    addi s1, s1, 1           # i = i + 1
    j    loop                # repeat

done:
```

# Bi-dimensional array

// C Code

```
int matrix[2][4];
```

<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>
<b>E</b>	<b>F</b>	<b>G</b>	<b>H</b>

- The elements of a matrix are stored in memory in order, starting from the first row (*stored by rows*)

`matrix[0][0]`

**A**

`matrix[0][1]`

**B**

`matrix[0][2]`

**C**

`matrix[0][3]`

**D**

`matrix[1][0]`

**E**

`matrix[1][1]`

**F**

`matrix[1][2]`

**G**

`matrix[1][3]`

**H**

# Item Address

// C Code

```
int matrix[2][4];
```

- Considering:
  - $N$  = number of columns
  - $M$  = number of rows
  - Base Address = address of `matrix[0][0]`
- Address of `matrix[i][j]` =  $\text{Base Address} + i * N * 4 + j * 4$



# Access to a row in a matrix

## C Code

```
int matrix[5][4];
int i;

for (i=0; i < 4; i++)
    matrix[3][i]++;
```

## RISC-V assembly code

```
.data
matrix:
    .word 8, 10, 11, 1
    .word 7, 5, 9, 2
    .word 6, 11, 0, 3
    .word 4, 3, 8, 4
    .word 0, 1, 2, 5
.text
    .globl main
main:
    la t0, matrix
    addi t1, zero, 3          # row number 3
    addi t2, zero, 4          # number of columns
    mul  t3, t2, t1           # number of words
                                # in the first 3 rows
    slli t3, t3, 2            # number of bytes
                                # in the first 3 rows
    add  t5, t0, t3           # t5: initial address
                                # of row with index 3
    add  t4, zero, zero       # t4 = 0 (i)
Lab1:
    lw   t6, (t5)             # matrix[3][i]
    addi t6, t6, 1            # ++
    sw   t6, (t5)
    addi t5, t5, 4            # update address
    addi t4, t4, 1
    bne  t4, t2, lab1
```

# Access to a column in a matrix

## C Code

```
int matrix[5][4];
int i;

for (i=0; i < 5; i++)
    matrix[i][2]++;
```

## RISC-V assembly code

```
.data
matrix:
    .word 8, 10, 11, 1
    .word 7, 5, 9, 2
    .word 6, 11, 0, 3
    .word 4, 3, 8, 4
    .word 0, 1, 2, 5
.text
    .globl main
main:
    la      t0, matrix
    addi    t1, zero, 4           # number of columns
    slli    t1, t1, 2             # bytes occupied by 1 row

    addi    t3, zero, 2           # column number 2
    slli    t3, t3, 2             # number of bytes
    add     t5, t0, t3            # t5: initial address
                                    # of matrix[0][2]

    addi    t2, zero, 5           # number of rows
    add     t4, zero, zero        # t4 = 0 (i)

Lab1:
    lw      t6, (t5)              # matrix[i][2]
    addi    t6, t6, 1             # ++
    sw      t6, (t5)
    add     t6, t5, t1            # update address
    addi    t4, t4, 1
    bne     t4, t2, lab1
```

# ASCII Code

- *American Standard Code for Information Interchange*
- Each text character has unique byte value
  - For example, S = 0x53, a = 0x61, A = 0x41
  - Lower-case and upper-case differ by 0x20 (32)

# Cast of Characters

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
<b>20</b>	space	<b>30</b>	0	<b>40</b>	@	<b>50</b>	P	<b>60</b>	`	<b>70</b>	p
<b>21</b>	!	<b>31</b>	1	<b>41</b>	A	<b>51</b>	Q	<b>61</b>	a	<b>71</b>	q
<b>22</b>	“	<b>32</b>	2	<b>42</b>	B	<b>52</b>	R	<b>62</b>	b	<b>72</b>	r
<b>23</b>	#	<b>33</b>	3	<b>43</b>	C	<b>53</b>	S	<b>63</b>	c	<b>73</b>	s
<b>24</b>	\$	<b>34</b>	4	<b>44</b>	D	<b>54</b>	T	<b>64</b>	d	<b>74</b>	t
<b>25</b>	%	<b>35</b>	5	<b>45</b>	E	<b>55</b>	U	<b>65</b>	e	<b>75</b>	u
<b>26</b>	&	<b>36</b>	6	<b>46</b>	F	<b>56</b>	V	<b>66</b>	f	<b>76</b>	v
<b>27</b>	'	<b>37</b>	7	<b>47</b>	G	<b>57</b>	W	<b>67</b>	g	<b>77</b>	w
<b>28</b>	(	<b>38</b>	8	<b>48</b>	H	<b>58</b>	X	<b>68</b>	h	<b>78</b>	x
<b>29</b>	)	<b>39</b>	9	<b>49</b>	I	<b>59</b>	Y	<b>69</b>	i	<b>79</b>	y
<b>2A</b>	*	<b>3A</b>	:	<b>4A</b>	J	<b>5A</b>	Z	<b>6A</b>	j	<b>7A</b>	z
<b>2B</b>	+	<b>3B</b>	;	<b>4B</b>	K	<b>5B</b>	[	<b>6B</b>	k	<b>7B</b>	{
<b>2C</b>	,	<b>3C</b>	<	<b>4C</b>	L	<b>5C</b>	\	<b>6C</b>	l	<b>7C</b>	
<b>2D</b>	-	<b>3D</b>	=	<b>4D</b>	M	<b>5D</b>	]	<b>6D</b>	m	<b>7D</b>	}
<b>2E</b>	.	<b>3E</b>	>	<b>4E</b>	N	<b>5E</b>	^	<b>6E</b>	n	<b>7E</b>	~
<b>2F</b>	/	<b>3F</b>	?	<b>4F</b>	O	<b>5F</b>	_	<b>6F</b>	o		

# Example with strings

## // C Code

```
char str[80] = "CAT";
int len = 0;

// compute length of string
while (str[len]) len++;
```

## # RISC-V assembly code

```
# s0 = array base address, s1 = len

        addi s1, zero, 0           # len = 0
while:   add t0, s0, s1             # address of str[len]
        lb t1, 0(t0)              # load str[len]
        beq t1, zero, done         # are we at the end of the string?
        addi s1, s1, 1            # len++
        j while                   # repeat while loop
done:
```

# Function Calls

- **Caller:** calling function (in this case, `main`)
- **Callee:** called function (in this case, `sum`)

## C Code

```
void main()
{
    int y;
    y = sum(42, 7);
    ...
}

int sum(int a, int b)
{
    return (a + b);
}
```

# Function Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee
- **Callee:**
  - **reads** the parameters
  - **performs** the function
  - **returns** result to caller
  - **returns** to point of call
  - **must not overwrite** registers or memory needed by caller

# RISC-V Function Conventions

- **Call Function:** jump and link (`j a1`)
- **Return from function:** jump register (`j r`)
- **Arguments:** `a0` – `a7`
- **Return value:** `a0`



# Call Function

## Caller

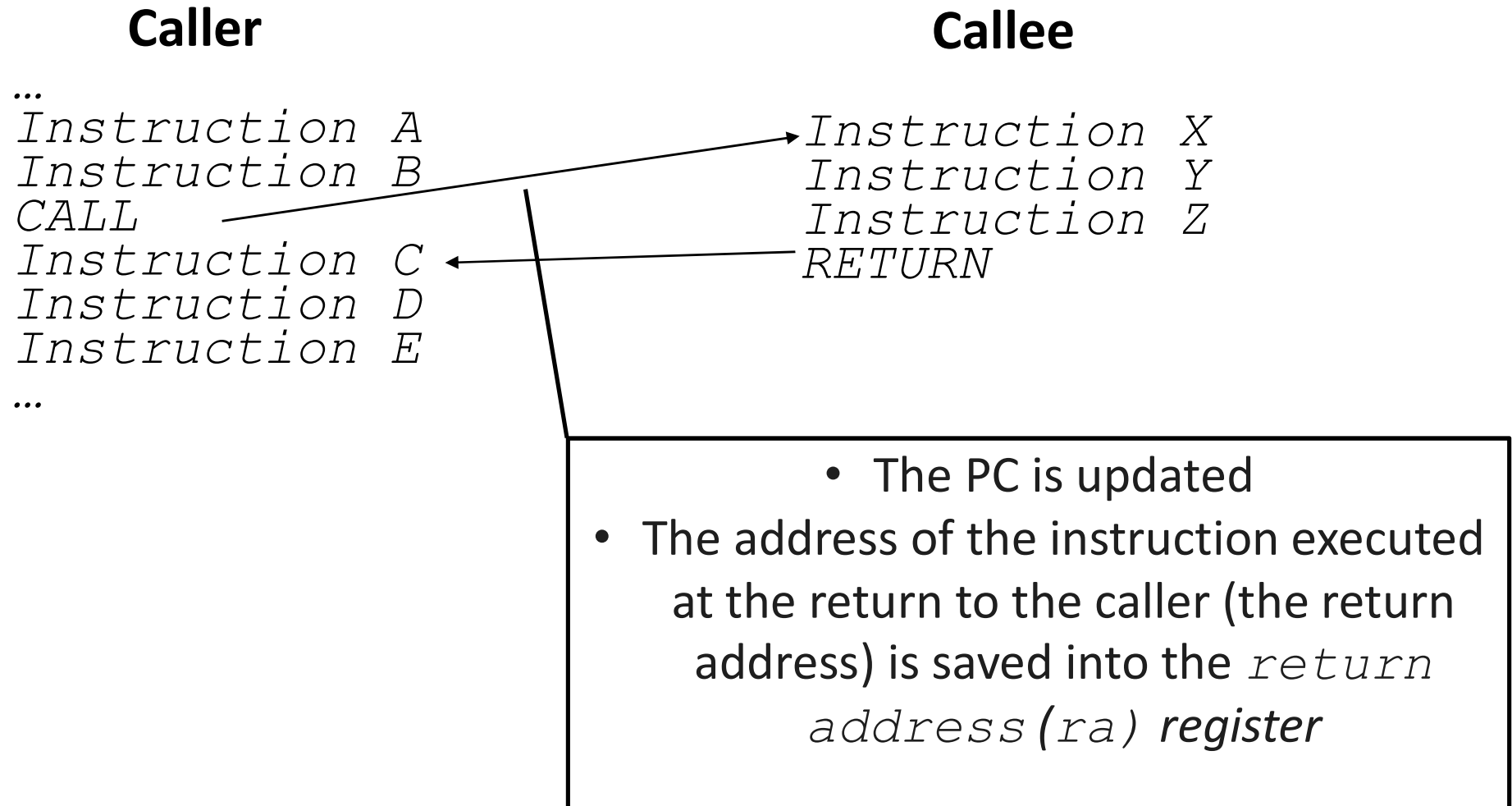
...  
*Instruction A*  
*Instruction B*  
*CALL*  
*Instruction C*  
*Instruction D*  
*Instruction E*  
...

## Callee

*Instruction X*  
*Instruction Y*  
*Instruction Z*  
*RETURN*

- The control flow of execution passes from the caller to the callee
- At the end of the called function the control returns to the caller

# Call Function



# Call Function

## Caller

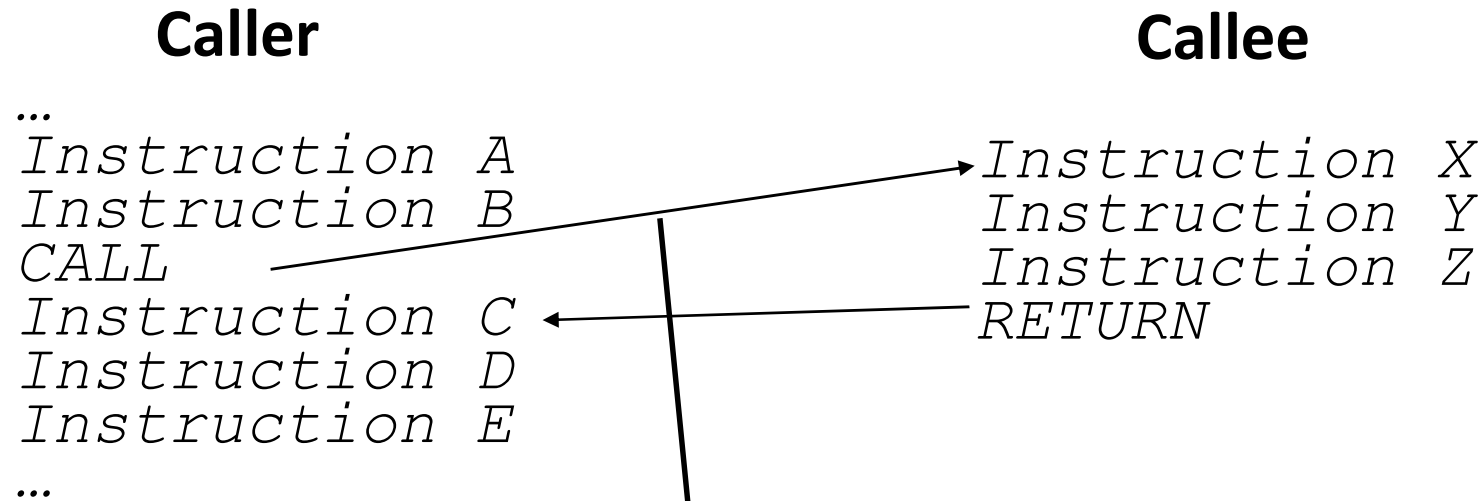
...  
Instruction A  
Instruction B  
CALL  
Instruction C  
Instruction D  
Instruction E  
...

## Callee

Instruction X  
Instruction Y  
Instruction Z  
RETURN

- The contents of *return address (ra) register* is restored into the Program Counter

# Call Function



- The call is executed with the pseudo-instruction `jal` (jump and link)

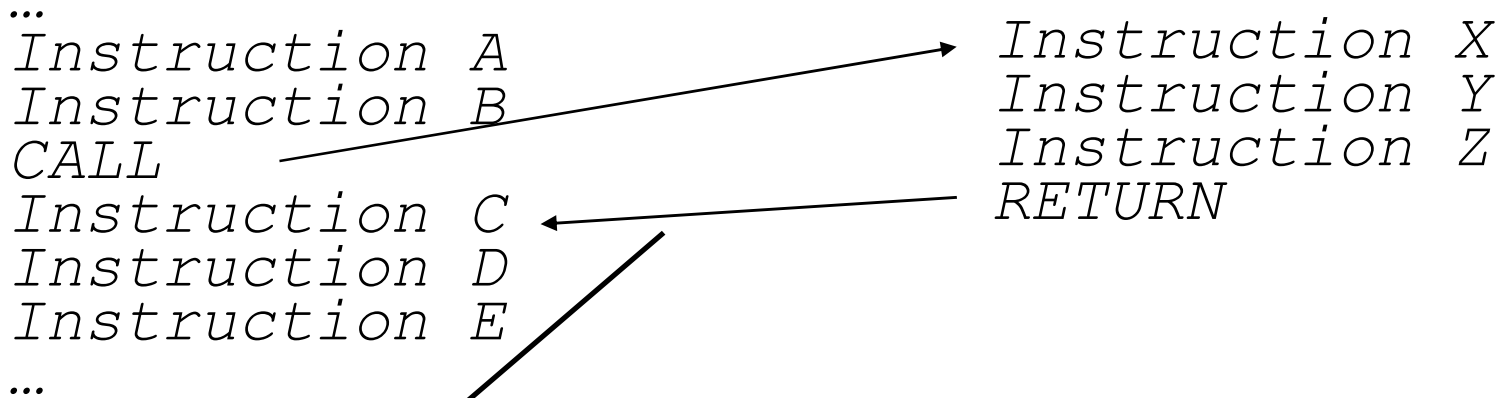
# Call Function

## Caller

...  
*Instruction A*  
*Instruction B*  
*CALL*  
*Instruction C*  
*Instruction D*  
*Instruction E*  
...

## Callee

*Instruction X*  
*Instruction Y*  
*Instruction Z*  
*RETURN*

- 
- The *return* is executed with the pseudo-instruction `j r` (jump to return address register)

# Function Calls

## C Code

```
int main() {  
    simple();  
    a = b + c;  
}
```

```
void simple() {  
    return;  
}
```

## RISC-V assembly code

```
0x00000300 main: jal simple  
0x00000304          add s0, s1, s2  
...
```

```
0x0000051c simple: jr ra
```

**jal simple:**

ra = PC + 4 (0x00000304)

jumps to simple label (PC = 0x0000051c)

**jr ra:**

PC = ra (0x00000304)

# Input Arguments & Return Value

## C Code

```
int main()
{
    int y;
    ...
    y = diffofsums(2, 3, 4, 5); // 4 arguments
    ...
}

int diffofsums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result; // return value
}
```

# Input Arguments & Return Value

## RISC-V assembly code

```
# s7 = y
main:
. . .
addi a0, zero, 2 # argument 0 = 2
addi a1, zero, 3 # argument 1 = 3
addi a2, zero, 4 # argument 2 = 4
addi a3, zero, 5 # argument 3 = 5
jal  diffofsums # call function
add  s7, a0, zero # y = returned value
. . .
# s3 = result
diffofsums:
add  t0, a0, a1 # t0 = f + g
add  t1, a2, a3 # t1 = h + i
sub  s3, t0, t1 # result = (f + g) - (h + i)
add  a0, s3, zero # put return value in a0
jr   ra # return to caller
```



# Input Arguments & Return Value

## RISC-V assembly code

```
# s3 = result
```

```
diffofsums:
```

```
    add  t0, a0, a1    # t0 = f + g
    add  t1, a2, a3    # t1 = h + i
    sub  s3, t0, t1    # result = (f + g) - (h + i)
    add  a0, s3, zero  # put return value in a0
    jr   ra            # return to caller
```

- `diffofsums` overwrote 3 registers: `t0`, `t1`, `s3`
- `diffofsums` can use *stack* to temporarily store registers

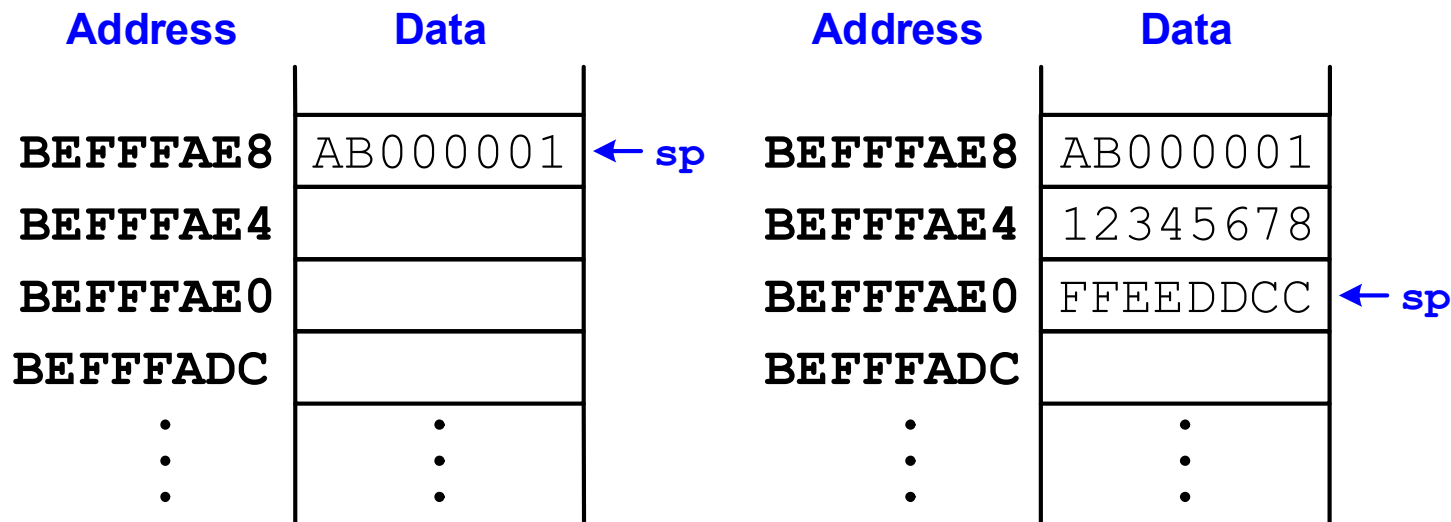
# The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- ***Expands (or push)***: uses more memory when more space needed
- ***Contracts (or pop)***: uses less memory when the space is no longer needed



# The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer:  $sp$  points to top of the stack



# How Functions use the Stack

- Called functions must have no unintended side effects
- But `diffofsums` overwrites 3 registers: `t0`, `t1`, `s3`

## # RISC-V assembly

```
# s3 = result
```

```
diffofsums:
```

```
add  t0, a0, a1    # t0 = f + g
```

```
add  t1, a2, a3    # t1 = h + i
```

```
sub  s3, t0, t1    # result = (f + g) - (h + i)
```

```
add  a0, s3, zero  # put return value in a0
```

```
jr   ra           # return to caller
```

# Stack frame

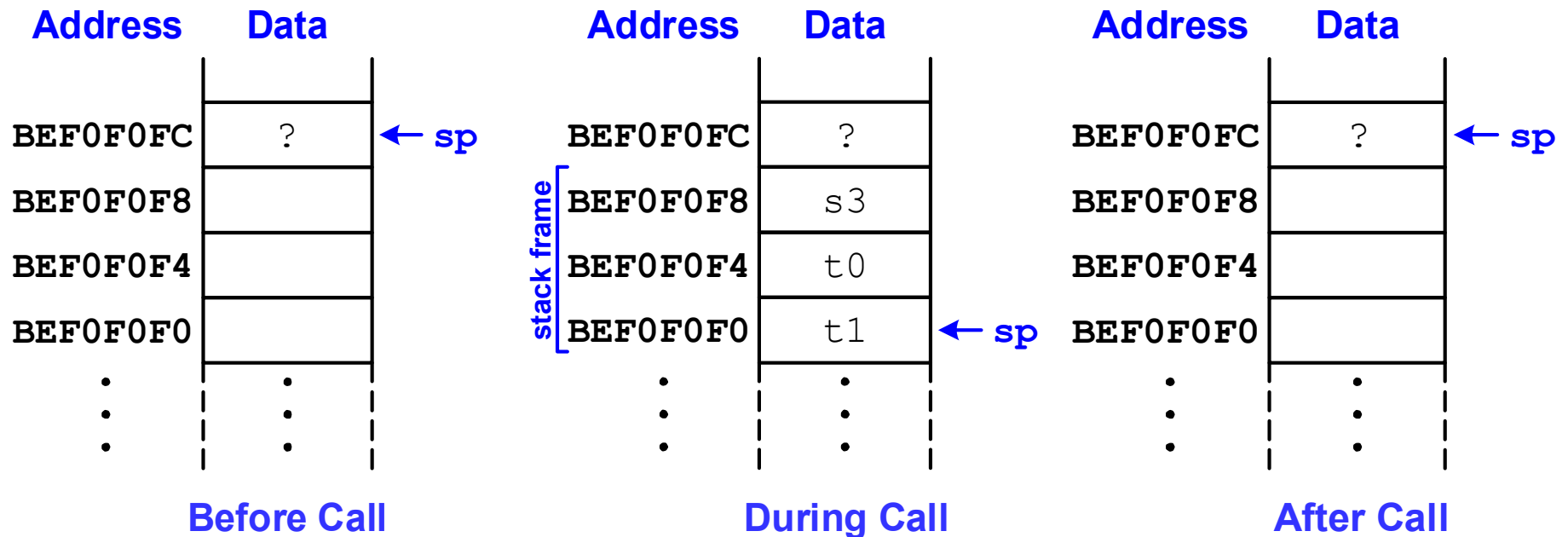
- The stack space that a function allocates for itself is called its *stack frame*
- Each function should access only its own stack frame, not the frames belonging to other functions.

# Storing Register Values on the Stack

```
# s3 = result
diffofsums:
    addi sp, sp, -12           # make space on stack to
                                # store three registers
    sw    s3, 8(sp)           # save s3 on stack
    sw    t0, 4(sp)           # save t0 on stack
    sw    t1, 0(sp)           # save t1 on stack

    add    t0, a0, a1          # t0 = f + g
    add    t1, a2, a3          # t1 = h + i
    sub    s3, t0, t1          # result = (f + g) - (h + i)
    add    a0, s3, zero        # put return value in a0
    lw    s3, 8(sp)           # restore s3 from stack
    lw    t0, 4(sp)           # restore t0 from stack
    lw    t1, 0(sp)           # restore t1 from stack
    addi sp, sp, 12           # deallocate stack space
    jr     ra                  # return to caller
```

# The stack during diffofsums Call



# Registers

<b>Preserved</b> <i>Callee-Saved</i>	<b>Nonpreserved</b> <i>Caller-Saved</i>
<b>s0-s11</b>	<b>t0-t6</b>
<b>sp</b>	<b>a0-a7</b>
<b>ra</b>	
stack above <b>sp</b>	stack below <b>sp</b>



# Storing Saved Registers on the Stack

```
# s3 = result
diffofsums:
    addi sp, sp, -4           # make space on stack to
                                # store one register
    sw    s3, 0(sp)          # save s3 on stack
    add    t0, a0, a1         # t0 = f + g
    add    t1, a2, a3         # t1 = h + i
    sub    s3, t0, t1         # result = (f + g) - (h + i)
    add    a0, s3, zero       # put return value in a0
    lw    s3, 0(sp)          # restore s3 from stack
    addi sp, sp, 4           # deallocate stack space
    jr     ra                 # return to caller
```

# Optimized diffofsums

```
# a0 = result
diffofsums:
    add    t0, a0, a1    # t0 = f + g
    add    t1, a2, a3    # t1 = h + i
    sub    a0, t0, t1    # result = (f + g) - (h + i)
    jr     ra            # return to caller
```

# Leaf and nonleaf functions

- A *leaf* function does not call other functions
- A *nonleaf* function calls other functions

# Nonleaf function

- A nonleaf function has to save non-preserved registers on the stack before it calls another function, and then restores those registers afterward
- The caller saves any non-preserved register (`t0-t6` and `a0-a7`) that are needed after the call
- The callee saves any of the preserved registers (`s0-s11` and `ra`) that intends to modify.

# Leaf function

- A leaf function may not save non-preserved registers (in particular  $\text{t}0 - \text{t}6$ ).

# Non-Leaf Function Calls

## Non-leaf function:

a function that calls another function

func1:

```
addi sp, sp, -4    # make space on stack
sw    ra, 0(sp)    # save ra on stack
jal   func2
...
lw    ra, 0(sp)    # restore ra from stack
addi sp, sp, 4     # deallocate stack space
jr    ra           # return to caller
```

Must preserve **ra** before function call.

# Non-leaf vs. leaf functions

# **f1 (non-leaf function)** uses s4-s5 and needs a0-a1 after call to f2

f1:

```
addi sp, sp, -20    # make space on stack for 5 words
sw    a0, 16(sp)
sw    a1, 12(sp)
sw    ra, 8(sp)      # save ra on stack
sw    s4, 4(sp)
sw    s5, 0(sp)
jal   f2
...
lw    ra, 8(sp)      # restore ra (and other regs) from stack
...
addi sp, sp, 20     # deallocate stack space
jr    ra            # return to caller
```

# **f2 (leaf function)** only uses s4 and calls no functions

f2:

```
addi sp, sp, -4     # make space on stack for 1 word
sw    s4, 0(sp)
...
lw    s4, 0(sp)
addi sp, sp, 4       # deallocate stack space
jr    ra            # return to caller
```

# Stack during Function Calls

Address	Data
BEF7FF0C	?
BEF7FF08	
BEF7FF04	
BEF7FF00	
BEF7FEFC	
BEF7FEF8	
BEF7FEF4	
⋮	⋮

Before Calls

Address	Data
BEF7FF0C	?
BEF7FF08	a0
BEF7FF04	a1
BEF7FF00	ra
BEF7FEFC	s4
BEF7FEF8	s5
BEF7FEF4	
⋮	⋮

After Call to f1

Address	Data
BEF7FF0C	?
BEF7FF08	a0
BEF7FF04	a1
BEF7FF00	ra
BEF7FEFC	s4
BEF7FEF8	s5
BEF7FEF4	s4
⋮	⋮

After Call to f2



# Function Call Summary

- **Caller**

- Save any needed registers (`ra`, maybe `t0–t6/a0–a7`)
- Put arguments in `a0–a7`
- Call function: `jal callee`
- Look for result in `a0`
- Restore any saved registers

- **Callee**

- Save registers that might be disturbed (`s0–s11`)
- Perform function
- Put result in `a0`
- Restore registers
- Return: `jr ra`

## Example: C strlen()

- Illustrates use of assembly instructions for the C strlen() function

```
int strlen(const char *str) {  
    int i;  
    for (i = 0; str[i] != '\0'; i++);  
    return i;  
}
```

- \*str in a0, i in t0, return value in a0

# Example: C strlen()

```
.section .text
.global strlen
strlen:          # a0 = const char *str
li      t0, 0    # i = 0
start:          # Start of for loop
add     t1, t0, a0 # Add the byte offset for str[i]
lb      t1, 0(t1) # Dereference str[i]
beqz    t1, end   # if str[i] == 0, break for loop
addi    t0, t0, 1 # Add 1 to our iterator
j       start     # Jump back to condition (1 backwards)
end:      # End of for loop
mv      a0, t0    # Move t0 into a0 to return
ret      # Return back via the return address register
```

# Example: C sort

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- v in a0, k in a1, temp in t0

# The procedure swap

swap: sll t1, a1, 2	# t1 = k * 4
add t1, a0, t1	# t1 = v+(k*4)
	# (address of v[k])
lw t0, 0(t1)	# t0 (temp) = v[k]
lw t2, 4(t1)	# t2 = v[k+1]
sw t2, 0(t1)	# v[k] = t2 (v[k+1])
sw t0, 4(t1)	# v[k+1] = t0 (temp)
jr ra	# return to calling routine

# The sort procedure

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1;
             j >= 0 && v[j] > v[j + 1];
             j -= 1) {
            swap(v, j);
        }
    }
}
```

- $v$  in  $a0$ ,  $n$  in  $a1$ ,  $i$  in  $s0$ ,  $j$  in  $s1$

# The procedure body

	mv s2, a0	# save a0 into s2	Move
	mv s3, a1	# save a1 into s3	params
	mv s0, zero	# i = 0	
for1tst:	slt t0, s0, s3	# t0 = 0 if s0 ≥ s3 (i ≥ n)	Outer loop
	beq t0, zero, exit1	# go to exit1 if s0 ≥ s3 (i ≥ n)	
	addi s1, s0, -1	# j = i - 1	
for2tst:	slti t0, s1, 0	# t0 = 1 if s1 < 0 (j < 0)	
	bne t0, zero, exit2	# go to exit2 if s1 < 0 (j < 0)	
	sll t1, s1, 2	# t1 = j * 4	
	add t2, s2, t1	# t2 = v + (j * 4)	Inner loop
	lw t3, 0(t2)	# t3 = v[j]	
	lw t4, 4(t2)	# t4 = v[j + 1]	
	slt t0, t4, t3	# t0 = 0 if t4 ≥ t3	
	beq t0, zero, exit2	# go to exit2 if t4 ≥ t3	
	mv a0, s2	# 1st param of swap is v (old a0)	Pass
	mv a1, s1	# 2nd param of swap is j	params
	jal swap	# call swap procedure	& call
	addi s1, s1, -1	# j -= 1	
	j for2tst	# jump to test of inner loop	Inner loop
exit2:	addi s0, s0, 1	# i += 1	
	j for1tst	# jump to test of outer loop	Outer loop

# The full procedure

sort:	addi sp, sp, -20	# make room on stack for 5 registers
	sw ra, 16(sp)	# save ra on stack
	sw s3, 12(sp)	# save s3 on stack
	sw s2, 8(sp)	# save s2 on stack
	sw s1, 4(sp)	# save s1 on stack
	sw s0, 0(sp)	# save s0 on stack
	...	# procedure body
	...	
exit1:	lw s0, 0(sp)	# restore s0 from stack
	lw s1, 4(sp)	# restore s1 from stack
	lw s2, 8(sp)	# restore s2 from stack
	lw s3, 12(sp)	# restore s3 from stack
	lw ra, 16(sp)	# restore ra from stack
	addi sp, sp, 20	# restore stack pointer
	jr ra	# return to calling routine



# Recursive Function Call

## High-level code

```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n * factorial(n-1));  
}
```

# Recursive Function Call

## High-Level Code

```
int factorial(int n) {  
  
    if (n <= 1)  
        return 1;  
  
    else  
        return (n*factorial(n-1));  
}
```

**Pass 1.** Treat as if calling another function. Ignore stack.

**Pass 2.** Save overwritten registers (needed after function call) on the stack before call.

## RISC-V Assembly

```
factorial:  
    addi sp, sp, -8    # save regs  
    sw    a0, 4(sp)  
    sw    ra, 0(sp)  
    addi t0, zero, 1   # temporary = 1  
    bgt   a0, t0, else # if n>1, go to else  
    addi a0, zero, 1   # otherwise, return 1  
    addi sp, sp, 8     # restore sp  
    jr    ra           # return  
else:  
    addi a0, a0, -1    # n = n - 1  
    jal   factorial    # recursive call  
    lw    t1, 4(sp)    # restore n into t1  
    lw    ra, 0(sp)    # restore ra  
    addi sp, sp, 8     # restore sp  
    mul   a0, t1, a0    # a0=n*factorial(n-1)  
    jr    ra           # return
```

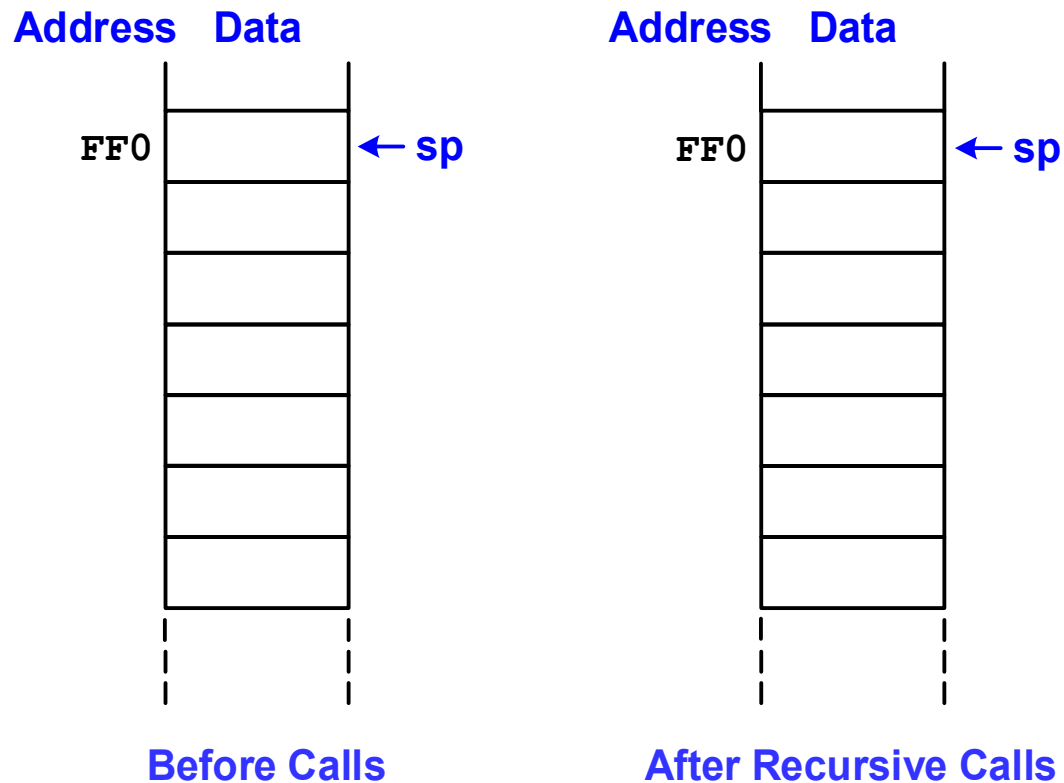
**Note:** n is restored from stack into t1 so it doesn't overwrite return value in a0.

# Stack During Recursive Call

```
0x8500 factorial: addi sp, sp, -8      # save registers
0x8504             sw  a0, 4(sp)
0x8508             sw  ra, 0(sp)
0x850C             addi t0, zero, 1    # temporary = 1
0x8510             bgt  a0, t0, else   # if n > 1, go to else
0x8514             addi a0, zero, 1    # otherwise, return 1
0x8518             addi sp, sp, 8      # restore sp
0x851C             jr   ra             # return
0x8520 else:       addi a0, a0, -1     # n = n - 1
0x8524             jal factorial      # recursive call
0x8528           lw   t1, 4(sp)       # restore n into t1
0x852C             lw   ra, 0(sp)      # restore ra
0x8530             addi sp, sp, 8      # restore sp
0x8534             mul  a0, t1, a0     # a0 = n*factorial(n-1)
0x8538             jr   ra             # return
```

# Stack During Recursive Function

When **factorial(3)** is called:



# Additional arguments

- Functions may have more than 8 input arguments
- The stack is used to store these temporary values
- By RISC-V convention, if a function has more than 8 arguments, the first 8 are passed in the argument registers and the additional arguments are passed on the stack, just above `sp`
- The caller must expand the stack to make room for the additional arguments.

# Example

- res = addem (n1, n2, n3, n4, n5, n6, n7, n8, n9); /\* n1+n2+n3+n4+n5+n6+n7+n8+n9 \*/

```
.data
n1: .word 1
n2: .word 2
n3: .word 3
n4: .word 4
n5: .word 5
n6: .word 6
n7: .word 7
n8: .word 8
n9: .word 9
sum: .word 0

.text
_start:
la t0, n1
lw a0, 0(t0)
lw a1, 4(t0)
lw a2, 8(t0)
lw a3, 12(t0)
lw a4, 16(t0)
lw a5, 20(t0)
lw a6, 24(t0)
lw a7, 28(t0)
lw t0, 32(t0)

addi sp, sp, -4
sw t0, 0(sp)

jal addem

addi sp, sp, 4

la t0, sum
sw a0, 0(t0)

.addem:
add a0, a0, a1
add a0, a0, a2
add a0, a0, a3
add a0, a0, a4
add a0, a0, a5
add a0, a0, a6
add a0, a0, a7

lw t0, (sp)

add a0, a0, t0

jr ra
.end addem
```

# Local Variables

- Local variables are declared within a function and can be accessed only within that function
- Local variables are stored in t0-t6; if there are too many local variables, they can also be stored in the function's stack frame
- In particular, local arrays are stored in the stack.

# Unconditional Jumps

- RISC-V has two types of unconditional jump **instructions**:
  - `jal` (jump and link)
  - `jalr` (jump and link register).



# Jump and link

- **Mnemonic:** *jump and link* (jal)
- **Format:**
  - `jal rd, imm20:1`
  - **rd** = PC+4; PC = PC + **imm**

# Pseudo-instructions exploiting jal

- **Plain unconditional jump:**

- `j imm => jal x0, imm # PC = PC+imm`

- **Call function:**

- `jal imm => jal ra, imm # ra = PC+4, PC = PC+imm`

# Labels

- Label indicates where to jump
- Represented in jump as immediate offset
  - **imm** = # bytes past jump instruction
  - In example, below, **imm** = (51C-300) = 0x21C
  - `jal simple = jal ra, 0x21C`

## RISC-V assembly code

```
0x00000300 main:  jal  simple          # call
0x00000304          add  s0, s1, s1
...              ...

0x0000051c simple: jr    ra          # return
```

# Jump and link register (jalr)

- **Mnemonic:** *jump and link register* (jalr)
- **Format:**
  - `jalr rd, rs, imm11:0`
  - **rd** = PC+4; PC = **rs** + SignExt(**imm**)

# Jump and link register (jalr)

- The `jalr` instruction allows to jump anywhere in a 32-bit absolute address range

```
li t0, 0x10001234    # loads 32 bits immediate
jalr x0, t0, 0        # jumps to the address store in t0
```

equivalent to

```
lui t0, 0x10001       # loads upper 20 bits
addi t0, t0, 0x234    # adds lower 12 bits to t0
jalr x0, t0, 0        # jumps to the address store in t0
```

## Jump and link register (`jalr`)

- When used with a base `rs=x0`, `jalr` can be used to implement a **function call** to the lowest 2 KB or highest 2 KB address region from anywhere in the address space, which could be used to implement calls to a small runtime library

```
jalr ra, x0, imm
```

# Pseudo-instructions exploiting jalr

- **Jump to register:**

- `jr rs => jalr x0, rs, 0`      # PC = rs

- **return:**

- `ret => jalr x0, ra, 0`      # PC = ra

- `jr ra => jalr x0, ra, 0`      # PC = ra

## Let's go back to jumps

- Considering *unconditional jumps*, the immediate is limited in size (20 bits for `jal`)
  - `j imm => jal x0, imm # PC = PC+imm`
  - Limits how far a program can jump



# Long Jumps

- Special instruction (`auipc`, add upper immediate to PC) helps jumping further

```
auipc rd, imm
```

adds a 20-bit upper immediate to the PC

`auipc` forms a 32-bit offset from the 20-bit immediate, filling in the lowest 12 bits with zeros, adds this offset to the PC, then places the result in register `rd`.

**`rd = PC + imm << 12`**

- Pseudoinstruction:
  - `call imm31:0`
  - Behaves like `jal imm`, but allows a longer jump not representable on the immediate on 12 bits

```
auipc ra, imm31:12
jalr ra, ra, imm11:0
```

## Alternative use of `auipc`

- The Program Counter is not one of the general-purpose registers, so it is not possible to access it directly.
- However, this is possible using **`auipc`** with an immediate of zero.
  - `auipc t0, 0 # copy PC into register t0`

# Pseudo-instruction: `la`

- `la: Load immediate address`
- `t0, label`

- **translates to**

```
auipc t0, ((label-.) + 0x00000800) >> 12
addi t0, t0, ((label-(-4)) & 0x00000fff)
```

## Details:

- `.` represents the address of the current instruction
- the displacement between the label and the current instruction is incremented with `0x00000800` to increment the value of bit 12 if bit 11 is 1. If bit 11 is 0, there isn't any effect due to the following right shifting of 12 bits
- `.-4` considers the address of the previous instruction
- the and operation filters the lower 12 bits.

# Example

```
00010040 la x10,var1
00010048 ...          # note that the la pseudo-instruction expands
                        # into 8 bytes
var1:
00010900 .word 999     # a 32-bit integer constant stored in memory
                        # at address var1
```

The assembler will calculate the value of (var1-. ) by subtracting the address represented by the label var1 from the address of the current instruction resulting in the number of bytes from the current instruction to the target label.

# Example (II)

The expanded example will become:

```
00010040 auipc x10, ((0x00010900 - 0x00010040) + 0x00000800) >> 12
00010044 addi x10,x10, ((0x00010900 - (0x00010044 - 4)) & 0x00000fff)
                        # note the extra -4 here!
```

```
0000 0000 0000 0001 0000 1001 0000 0000 - # 00010900
0000 0000 0000 0001 0000 0000 0100 0000  # 00010040
=====
0000 0000 0000 0000 0000 1000 1100 0000 + # 000008c0
0000 0000 0000 0000 0000 1000 0000 0000  # 00000800
=====
0000 0000 0000 0000 0001 0000 1100 0000  # 000010c0

0000 0000 0000 0000 0000 0000 0000 0001  # 000010c0 << 12
```

Finally, this is the code generated:

```
00010040 auipc x10, 0x0001      #      (0x000008c0 + 0x00000800) >> 12
00010044 addi x10,x10,0x8c0     #      (0x000008c0 & 0x00000fff)
```

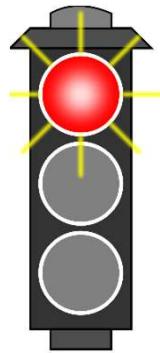
# Example (III)

0000 0000 0000 0001 0000 0000 0100 0000	# 00010040 (PC)
0000 0000 0000 0000 0001 0000 0000 0000	# auipc x10, 0x00001
	# 0x00001 << 12
1111 1111 1111 1111 1111 1000 1100 0000	# addi x10, x10, 0x8c0
	# SignExt(0x8c0)
-----	
0000 0000 0000 0001 0000 1001 0000 0000	# value in x10

# Far jumps

- Conditional branches present a limit into the target address representation

```
if (i == j)
{
    f = g - h;
    . . .
    . . .
    . . .
}
else
    f = g + h;
```



L1:  
L2:

```
bne t0, t1, L1
sub s0, s1, s2
. . .
. . .
. . .
j L2
add s0, s1, s2
```

Let assume that L1 is  
non representable  
due to the far jump

# Far jumps: solution

- The far jump is executed by means of an unconditional jump.

```
if (i == j)
{
    f = g - h;
    . . .
    . . .
    . . .
}
else
    f = g + h;

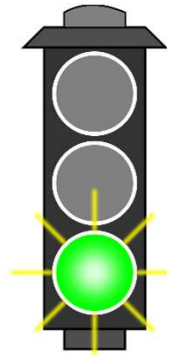
LAB1: sub s0, s1, s2
      . . .
      . . .
      . . .
      j L2
L1:   add s0, s1, s2
L2:
```

```
beq t0, t1, LAB1
```

```
auipc t0, ((L1-. ) + 0x00000800) >> 12
```

```
addi t0, t0, ((L1-(-4)) & 0x00000fff)
```

```
jr t0
```





# Far Calls

- Calls to procedure present a limit into the target address representation (20 bits for the immediate field)
  - `jal my_routine => jal ra, imm20:1`
- The solution:
  - `call my_routine => auipc ra, imm31:12`
  - `jalr ra, ra, imm11:0`

# More RISC-V Pseudoinstructions

Pseudoinstruction	RISC-V Instructions
j label	jal zero, label
jr ra	jalr zero, ra, 0
mv t5, s3	addi t5, s3, 0
not s7, t2	xori s7, t2, -1
nop	addi zero, zero, 0
ret	jalr zero, ra, 0
bgt s1, t3, L3	blt t3, s1, L3
bgez t2, L7	bge t2, zero, L7
call L1	auipc ra, imm <sub>31:12</sub> jalr ra, ra, imm <sub>11:0</sub>

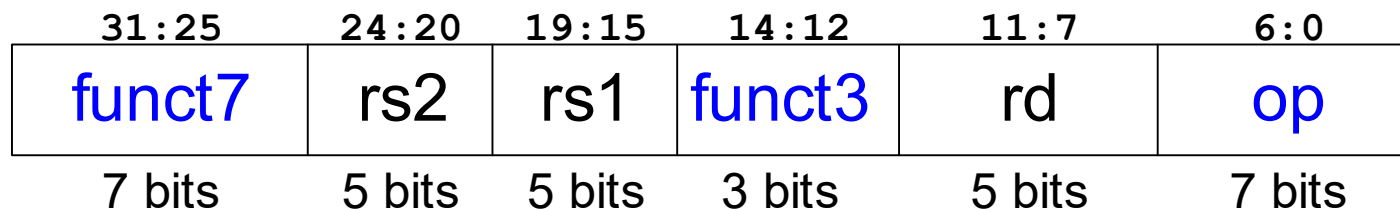
# Machine Language

- Binary representation of instructions
- Computers only understand 1's and 0's
- 32-bit instructions
  - Simplicity favors regularity: 32-bit data & instructions
- **4 Types of Instruction Formats:**
  - R-Type
  - I-Type
  - S/B-Type
  - U/J-Type

# R-Type

- **Register-type**
- 3 register operands:
  - rs1, rs2: source registers
  - rd: destination register
- Other fields:
  - op: the *operation code* or *opcode*
  - funct7, funct3:  
the *function* (7 bits and 3-bits, respectively)  
with opcode, tells CPU what operation to perform

## R-Type



# R-Type examples

## Assembly

## Field Values

## Machine Code

add s2, s3, s4  
add x18, x19, x20  
sub t0, t1, t2  
sub x5, x6, x7

funct7	rs2	rs1	funct3	rd	op
0	20	19	0	18	51
32	7	6	0	5	51
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

funct7	rs2	rs1	funct3	rd	op	
0000,000	10100	1001,1	000	10010	011,0011	(0x01498933)
0100,000	00111	0011,0	000	00101	011,0011	(0x407302B3)
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

## Assembly

## Field Values

## Machine Code

sll s7, t0, s1  
sll x23, x5, x9  
xor s8, s9, s10  
xor x24, x25, x26

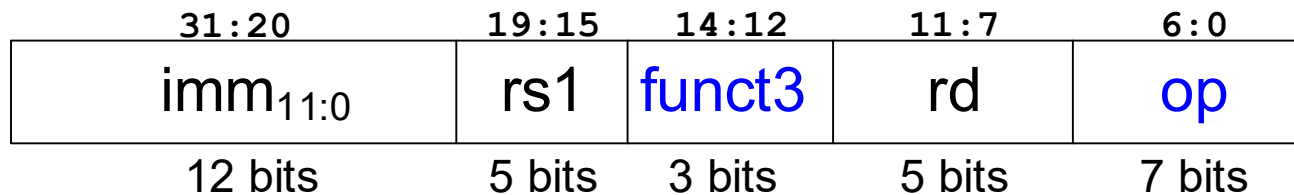
funct7	rs2	rs1	funct3	rd	op
0	9	5	1	23	51
0	26	25	4	24	51

funct7	rs2	rs1	funct3	rd	op	
0000 000	01001	00101	001	10111	011 0011	(0x00929BB3)
0000 000	11010	11001	100	11000	011 0011	(0x01ACCC33)

# I-Type

- *Immediate-type*
- 3 operands:
  - rs1: register source operand
  - rd: register destination operand
  - imm: 12-bit two's complement immediate
- Other fields:
  - op: the opcode
    - Simplicity favors regularity: all instructions have opcode
  - funct3: the function (3-bit function code)
    - with opcode, tells computer what operation to perform

## I-Type



# I-Type examples

## Assembly

## Field Values

## Machine Code

```
addi s0, s1, 12
addi x8, x9, 12
addi s2, t1, -14
addi x18, x6, -14
lw t2, -6(s3)
lw x7, -6(x19)
lh s1, 27(zero)
lh x9, 27(x0)
lb s4, 0x1F(s4)
lb x20, 0x1F(x20)
```

imm <sub>11:0</sub>	rs1	funct3	rd	op
12	9	0	8	19
-14	6	0	18	19
-6	19	2	7	3
27	0	1	9	3
0x1F	20	0	20	3
12 bits	5 bits	3 bits	5 bits	7 bits

imm <sub>11:0</sub>	rs1	funct3	rd	op	
0000 0000 1100	01001	000	01000	001 0011	(0x00C48413)
1111 1111 0010	00110	000	10010	001 0011	(0xFF230913)
1111 1111 1010	10011	010	00111	000 0011	(0xFFA9A383)
0000 0001 1011	00000	001	01001	000 0011	(0x01B01483)
0000 0001 1111	10100	000	10100	000 0011	(0x01FA0A03)
12 bits	5 bits	3 bits	5 bits	7 bits	

# S/B-Type

- *Store-Type*
- *Branch-Type*
- Differ only in immediate encoding

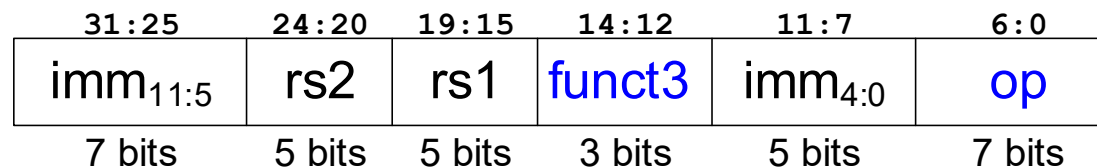
31:25	24:20	19:15	14:12	11:7	6:0	
imm <sub>11:5</sub>	rs2	rs1	funct3	imm <sub>4:0</sub>	op	S-Type
imm <sub>12,10:5</sub>	rs2	rs1	funct3	imm <sub>4:1,11</sub>	op	B-Type
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	



# S-Type

- **Store-Type**
- 3 operands:
  - rs1: base register
  - rs2: value to be stored to memory
  - imm: 12-bit two's complement immediate
- Other fields:
  - op: the opcode
    - Simplicity favors regularity: all instructions have opcode
  - funct3: the function (3-bit function code)
    - with opcode, tells computer what operation to perform

## S-Type



# S-Type examples

## Assembly

## Field Values

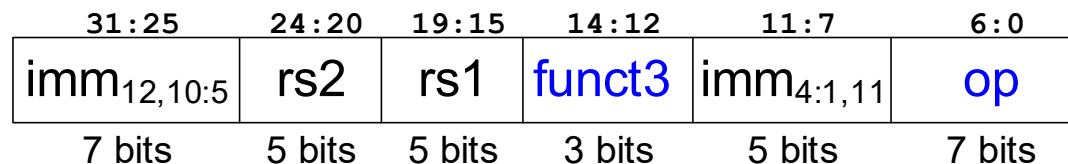
## Machine Code

	imm <sub>11:5</sub>	rs2	rs1	funct3	imm <sub>4:0</sub>	op	imm <sub>11:5</sub>	rs2	rs1	funct3	imm <sub>4:0</sub>	op	
<b>sw</b> t2, -6(s3)	1111 111	7	19	2	11010	35	1111 111	00111	10011	010	11010	010 0011	(0xFE79AD23)
<b>sw</b> x7, -6(x19)													
<b>sh</b> s4, 23(t0)	0000 000	20	5	1	10111	35	0000 000	10100	00101	001	10111	010 0011	(0x01429BA3)
<b>sh</b> x20, 23(x5)													
<b>sb</b> t5, 0x2D(zero)	0000 001	30	0	0	01101	35	0000 001	11110	00000	000	01101	010 0011	(0x03E006A3)
<b>sb</b> x30, 0x2D(x0)													
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

# B-Type

- **Branch-Type** (similar format to S-Type)
- 3 operands:
  - rs1: register source 1
  - rs2: register source 2
  - $\text{imm}_{12:1}$ : 12-bit two's complement immediate – address offset
- Other fields:
  - op: the opcode
    - Simplicity favors regularity: all instructions have opcode
  - funct3: the function (3-bit function code)
    - with opcode, tells computer what operation to perform

## B-Type



# B-Type examples

- The immediate encodes where to branch (relative to the branch instruction)
- Immediate encoding is strange
- **Example:**

# RISC-V Assembly

```
0x70      beq  s0, t5, L1
0x74      add  s1, s2, s3
0x78      sub  s5, s6, s7
0x7C      lw   t0, 0(s1)
0x80 L1: addi s1, s1, -15
```

L1 is 4 instructions (i.e., **16 bytes**) past beq

imm<sub>12:0</sub> = 16    0    0 0 0 0    0 0 0 1    0 0 0 0

bit number    12    11 10 9 8    7 6 5 4    3 2 1 0

**Assembly**

**Field Values**

**Machine Code**

	imm <sub>12,10:5</sub>	rs2	rs1	funct3	imm <sub>4:1,11</sub>	op	imm <sub>12,10:5</sub>	rs2	rs1	funct3	imm <sub>4:1,11</sub>	op	
beq s0, t5, L1	0000 000	30	8	0	1000 0	99	0000 000	11110	01000	000	1000 0	110 0011	(0x01E40863)
beq x8, x30, 16													
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

# U/J-Type

- *Upper-Immediate-Type*
- *Jump-Type*
- Differ only in immediate encoding

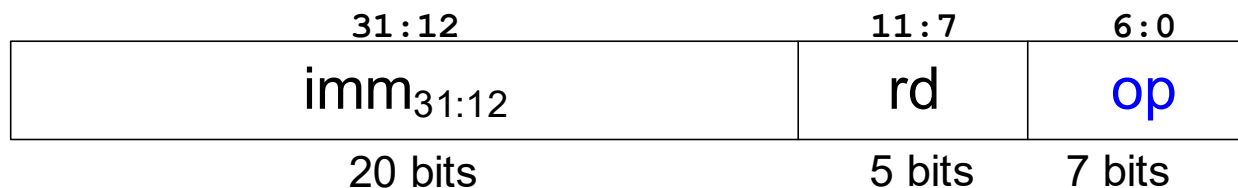
31:12	11:7	6:0
imm <sub>31:12</sub>	rd	op
imm <sub>20,10:1,11,19:12</sub>	rd	op
20 bits	5 bits	7 bits

**U-Type**  
**J-Type**

# U-Type

- ***Upper-immediate-Type***
- Used for `lui` and `auipc`
- 2 operands:
  - `rd`: destination register
  - `imm31:12`: 20 bits shifted to upper 20 bits
- Other fields:
  - `op`: the *operation code* or *opcode* – tells CPU what operation to perform

## U-Type



# U-Type Example

## Assembly

```
lui s5, 0x8CDEF  
lui x21, 0x8CDEF
```

## Field Values

imm <sub>31:12</sub>	rd	op
0x8CDEF	21	55
20 bits	5 bits	7 bits

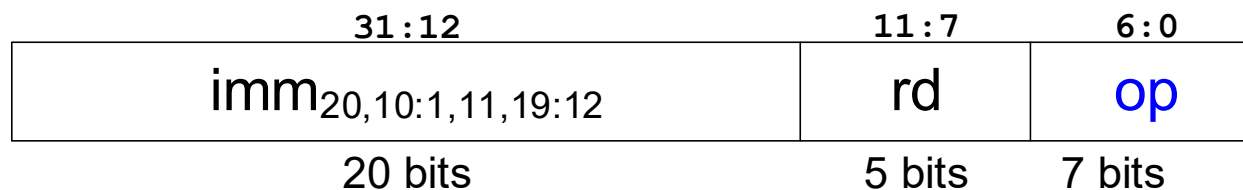
## Machine Code

imm <sub>31:12</sub>	rd	op	
1000 1100 1101 1110 1111	10101	011 0111	(0x8CDEFAB7)
20 bits	5 bits	7 bits	

# J-Type

- **Jump-Type**
- Used for jump-and-link instruction (`jal`)
- 2 operands:
  - `rd`: destination register
  - `imm20,10:1,11,19:12`: 20 bits (20:1) of a 21-bit immediate
- Other fields:
  - `op`: the operation code or opcode – tells computer what operation to perform

## J-Type



- Note: `jalr` is I-type, not J-type, to specify `rs`



# J-Type Example

# Address	RISC-V Assembly
0x0000540C	jal ra, func1
0x00005410	add s1, s2, s3
...	...
0x000ABC04	func1: add s4, s5, s8
...	...

0xABC04 – 0x540C =

**0xA67F8**

func1 is 0xA67F8 bytes past jal

imm = 0xA67F8

0	1	0	1	0	0	1	1	0	0	1	1	1	1	1	1	1	0	0	0	0
20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

bit number

## Assembly

## Field Values

## Machine Code

```
jal ra, func1
jal x1, 0xA67F8
```

imm <sub>20,10:1,11,19:12</sub>	rd	op	imm <sub>20,10:1,11,19:12</sub>	rd	op
0111 1111 1000 1010 0110	1	111	0111 1111 1000 1010 0110	00001	110 1111
20 bits	5 bits	7 bits	20 bits	5 bits	7 bits

(0x7F8A60EF)

# Instruction formats

7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
funct7	rs2	rs1	funct3	rd	op
imm <sub>11:0</sub>		rs1	funct3	rd	op
imm <sub>11:5</sub>	rs2	rs1	funct3	imm <sub>4:0</sub>	op
imm <sub>12,10:5</sub>	rs2	rs1	funct3	imm <sub>4:1,11</sub>	op
imm <sub>31:12</sub>				rd	op
imm <sub>20,10:1,11,19:12</sub>				rd	op
20 bits				5 bits	7 bits

**R-Type**

**I-Type**

**S-Type**

**B-Type**

**U-Type**

**J-Type**

# Design Principle 4

## Good design demands good compromises

- Multiple instruction formats allow flexibility
  - add, sub: use 3 register operands
  - lw, sw, addi: use 2 register operands and a constant
- Number of instruction formats kept small
  - to adhere to design principles 1 and 3 (simplicity favors regularity and smaller is faster).

# Constants / Immediates

## Immediate Bits

imm <sub>11</sub>												imm <sub>11:1</sub>												imm <sub>0</sub>	I, S B U J							
imm <sub>12</sub>												imm <sub>11:1</sub>												0								
imm <sub>31:21</sub>						imm <sub>20:12</sub>						0																				
imm <sub>20</sub>						imm <sub>20:12</sub>						imm <sub>11:1</sub>												0								
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	

# Immediate Encodings

## Instruction Bits

funct7								4	3	2	1	0	rs1					funct3					rd					R I S B U J
11	10	9	8	7	6	5	4	3	2	1	0	rs1					funct3					rd						
11	10	9	8	7	6	5	rs2					rs1					funct3					4	3	2	1	0		
12	10	9	8	7	6	5	rs2					rs1					funct3					4	3	2	1	11		
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	rd								
20	10	9	8	7	6	5	4	3	2	1	11	19	18	17	16	15	14	13	12	rd								
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				

- Immediate bits *mostly* occupy **consistent instruction bits**.
  - Simplifies hardware to build the microprocessor
- **Sign bit** of signed immediate is in the **MSB** of instruction.

# Instruction Fields & Formats

Instruction	op	funct3	Funct7	Type
<b>add</b>	0110011 (51)	000 (0)	0000000 (0)	R-Type
<b>sub</b>	0110011 (51)	000 (0)	0100000 (32)	R-Type
<b>and</b>	0110011 (51)	111 (7)	0000000 (0)	R-Type
<b>or</b>	0110011 (51)	110 (6)	0000000 (0)	R-Type
<b>addi</b>	0010011 (19)	000 (0)	-	I-Type
<b>beq</b>	1100011 (99)	000 (0)	-	B-Type
<b>bne</b>	1100011 (99)	001 (1)	-	B-Type
<b>lw</b>	0000011 (3)	010 (2)	-	I-Type
<b>sw</b>	0100011 (35)	010 (2)	-	S-Type
<b>jal</b>	1101111 (111)	-	-	J-Type
<b>jalr</b>	1100111 (103)	000 (0)	-	I-Type
<b>lui</b>	0110111 (55)	-	-	U-Type

# Interpreting Machine Code

- Write in binary
- Start with **op** (& **funct3**): tells how to parse rest
- Extract fields
- **op**, **funct3**, and **funct7** fields tell operation
- **Ex:** 0x41FE83B3 and 0xFDA58393

Machine Code

Field Values

Assembly

(0x41FE83B3)

funct7

rs2

rs1

funct3

rd

op

0100 000

11111

11101

000

00111

011 0011

7 bits

5 bits

5 bits

3 bits

5 bits

7 bits

funct7

rs2

rs1

funct3

rd

op

32

31

29

0

7

51

7 bits

5 bits

5 bits

3 bits

5 bits

7 bits

sub x7, x29, x31

sub t2, t4, t6

(0xFDA48393)

imm<sub>11:0</sub>

rs1

funct3

rd

op

1111 1101 1010

01001

000

00111

001 0011

12 bits

5 bits

3 bits

5 bits

7 bits

imm<sub>11:0</sub>

rs1

funct3

rd

op

-38

9

0

7

19

12 bits

5 bits

3 bits

5 bits

7 bits

addi x7, x9, -38

addi t2, s1, -38

# Addressing Modes

## How do we address the operands?

- Register Only
- Immediate
- Base Addressing
- PC-Relative.



# Addressing Modes

## Register Only

- Operands found in registers
  - **Example:** `add s0, t2, t3`
  - **Example:** `sub t6, s1, 0`

## Immediate

- 12-bit signed immediate used as an operand
  - **Example:** `addi s4, t5, -73`
  - **Example:** `ori t3, t7, 0xFF`

# Addressing Modes

## Base Addressing

- Loads and Stores
- Address of operand is:

base address + immediate

— **Example:** `lw s4, 72(zero)`

- $\text{address} = 0 + 72$

— **Example:** `sw t2, -25(t1)`

- $\text{address} = t1 - 25$

# Addressing Modes

## PC-Relative Addressing: branches, jal and auipc

### Example:

Address	Instruction
0x354	L1:     addi s1, s1, 1
0x358	sub  t0, t1, s7
...	...
0xEB0	bne  s8, s9, L1

The label is (0xEB0-0x354) = 0xB5C (**2908**) instructions **before** bne

imm<sub>12:0</sub> = -2908    **1**    **0**    **1**    **0**    **0**    **1**    **0**    **1**    **0**    0    1    0    0  
                   bit number    12    11   10   9   8    7   6   5   4    3   2   1   0

Assembly	Field Values						Machine Code						
	imm <sub>12,10:5</sub>	rs2	rs1	funct3	imm <sub>4:1,11</sub>	op	imm <sub>12,10:5</sub>	rs2	rs1	funct3	imm <sub>4:1,11</sub>	op	
beq s8, s9, L1	<b>1100 101</b>	24	25	<b>1</b>	<b>0010 0</b>	<b>99</b>	<b>1100 101</b>	11000	11001	<b>001</b>	<b>0010 0</b>	<b>110 0011</b>	(0xCB8C9263)
(beq x25, x26, L1)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

# Addressing Modes

## PC-Relative Addressing: branches, jal and auipc

### Example:

Address	Instruction
0x354	L1:     addi s1, s1, 1
0x358	sub t0, t1, s7
...	...
0xEB0	bne s8, s9, L1

-2908 corresponds to  
1 0100 1010 0100

The label is  $(0xEB0 - 0x354) = 0xB5C$  (**2908**) instructions **before** bne

imm<sub>12:0</sub> = -2908    **1**    **0**    **1**    **0**    **0**    **1**    **0**    **1**    **0**    0    1    0    0  
                   bit number    12    11   10   9   8    7   6   5   4    3   2   1   0

Assembly	Field Values						Machine Code						
	imm <sub>12,10:5</sub>	rs2	rs1	funct3	imm <sub>4:1,11</sub>	op	imm <sub>12,10:5</sub>	rs2	rs1	funct3	imm <sub>4:1,11</sub>	op	
beq s8, s9, L1	<b>1100 101</b>	24	25	<b>1</b>	<b>0010 0</b>	<b>99</b>	<b>1100 101</b>	11000	11001	<b>001</b>	<b>0010 0</b>	<b>110 0011</b>	(0xCB8C9263)
(beq x25, x26, L1)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

# Exercise

Calculate the immediate field for the `bne` instruction.

<b>0x40</b>	<b>loop:</b>	<code>add</code>	<code>t1, a0, s0</code>
<b>0x44</b>		<code>lb</code>	<code>t1, 0(t1)</code>
<b>0x48</b>		<code>add</code>	<code>t2, a1, s0</code>
<b>0x4C</b>		<code>sb</code>	<code>t1, 0(t2)</code>
<b>0x50</b>		<code>addi</code>	<code>s0, s0, 1</code>
<b>0x54</b>		<code>bne</code>	<code>t1, zero, <b>loop</b></code>
<b>0x58</b>		<code>lw</code>	<code>s0, 0(sp)</code>

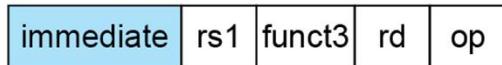
# Solution

$$0x40 - 0x54 = -20$$

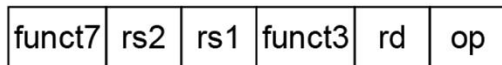
```
bne t1, zero, -20
```

# RISC-V Addressing Modes summary

## 1. Immediate addressing



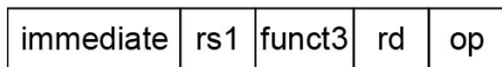
## 2. Register addressing



Registers

Register

## 3. Base addressing



Memory

Register

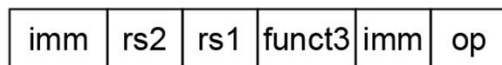
+

Byte Halfword

Word

Doubleword

## 4. PC-relative addressing



Memory

PC

+

Word

# The Power of the Stored Program

- 32-bit instructions & data stored in memory
- To run a new program:
  - No rewiring required
  - Simply store new program in memory
- Program Execution:
  - Processor *fetches* (reads) instructions from memory in sequence
  - Processor performs the specified operation



# The Stored Program

## Assembly Code

```
add  s2, s3, s4
sub  t0, t1, t2
addi s2, t1, -14
lw   t2, -6(s3)
```

## Machine Code

```
0x01498933
0x407302B3
0xFF230913
0xFFA9A383
```

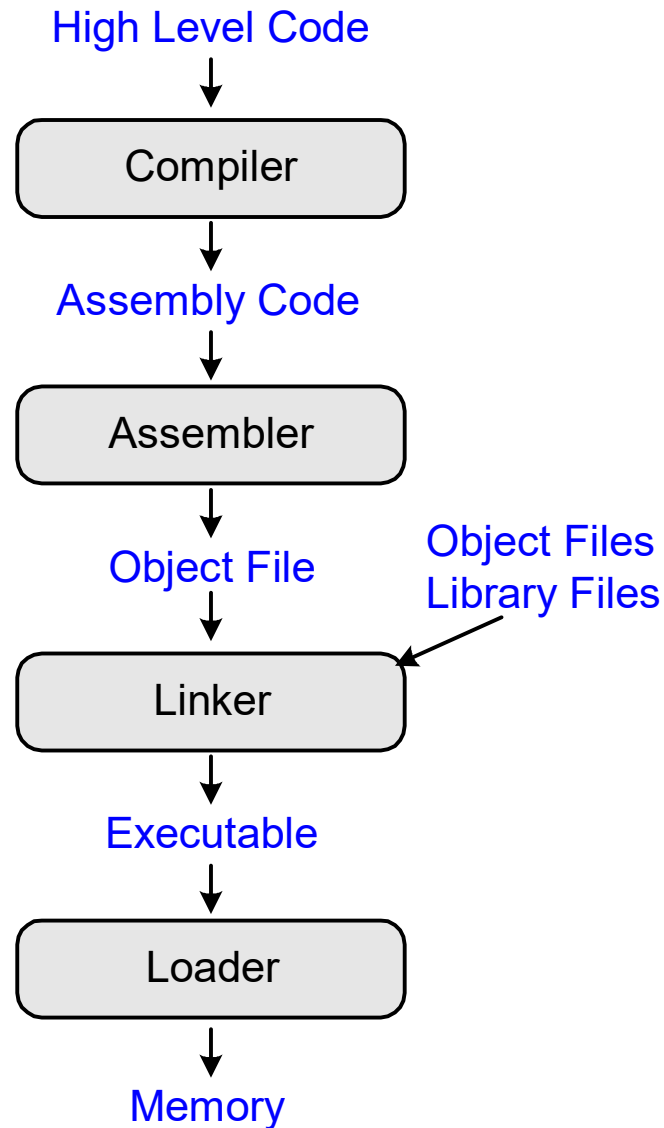
Address	Instructions
⋮	⋮
0000083C	F F A 9 A 3 8 3
00000838	F F 2 3 0 9 1 3
00000834	4 0 7 3 0 2 B 3
00000830	0 1 4 9 8 9 3 3
⋮	⋮

Main Memory

**Program Counter (PC):** keeps track of current instruction

← PC

# How to Compile & Run a Program

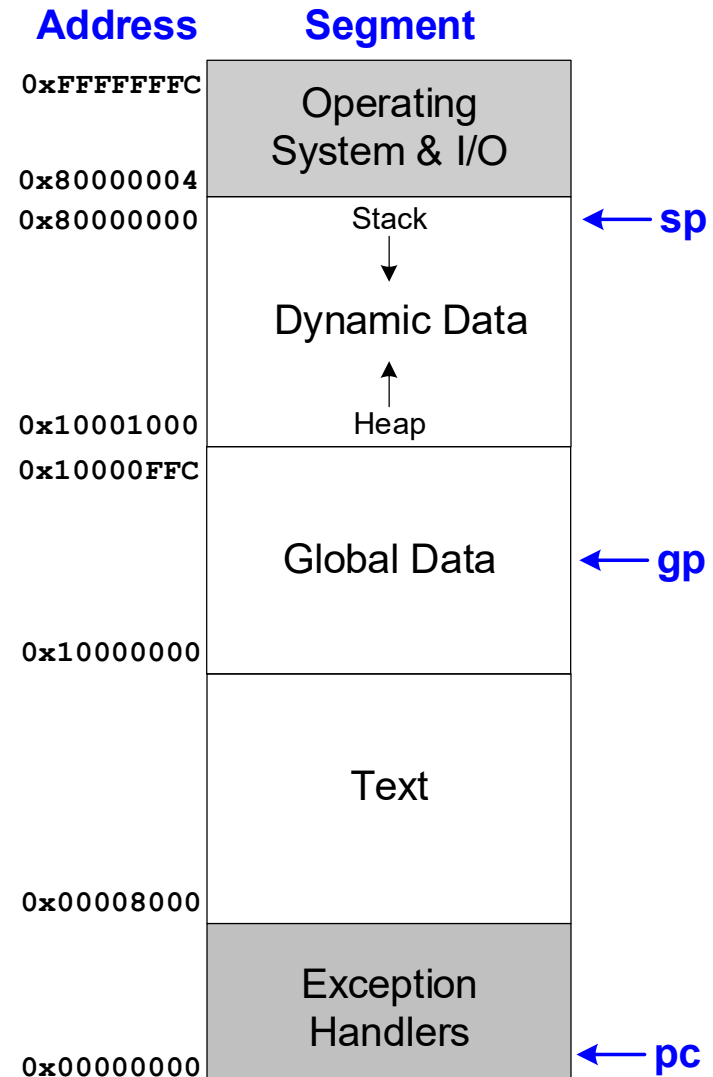


# What is Stored in Memory?

- **Instructions** (also called *text*)
- **Data**
  - **Global/static**: allocated before program begins
  - **Dynamic**: allocated within program
- How **big** is memory?
  - At most  $2^{32} = 4$  gigabytes (4 GB)
  - From address 0x00000000 to 0xFFFFFFFF

# RISC-V Memory Map

- Text: program code
- Global data: global variables
  - e.g., static variables in C, constant arrays and strings
  - x3 (gp, global pointer) initialized to address allowing  $\pm$ offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



# Example Program: C Code

```
int f, g, y; // global variables

int func(int a, int b) {
    if (b < 0)
        return (a + b);
    else
        return(a + func(a, b-1));
}

void main() {
    f = 2;
    g = 3;
    y = func(f,g);

    return;
}
```

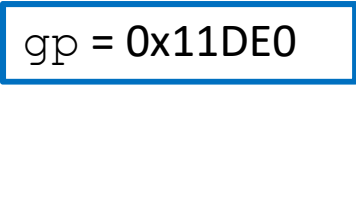
# Example Program: RISC-V Assembly

Address	Machine Code	RISC-V Assembly Code
10144:	ff010113	func: addi sp, sp, -16
10148:	00112623	sw ra, 12(sp)
1014c:	00812423	sw s0, 8(sp)
10150:	00050413	mv s0, a0
10154:	00a58533	add a0, a1, a0
10158:	0005da63	bgez a1, 1016c <func+0x28>
1015c:	00c12083	lw ra, 12(sp)
10160:	00812403	lw s0, 8(sp)
10164:	01010113	addi sp, sp, 16
10168:	00008067	ret
1016c:	fff58593	addi a1, a1, -1
10170:	00040513	mv a0, s0
10174:	fd1ff0ef	jal ra, 10144 <func>
10178:	00850533	add a0, a0, s0
1017c:	fe1ff06f	j 1015c <func+0x18>

Maintain **4-word alignment** of **sp** (for compatibility with RV128I) even though only space for 2 words needed.

# Example Program: RISC-V Assembly

Address	Machine Code	RISC-V Assembly Code
10180:	ff010113	main: addi sp, sp, -16
10184:	00112623	sw ra, 12(sp)
10188:	00200713	li a4, 2
1018c:	c4e1a823	sw a4, -944(gp) # 11a30 <f>
10190:	00300713	li a4, 3
10194:	c4e1aa23	sw a4, -940(gp) # 11a34 <g>
10198:	00300593	li a1, 3
1019c:	00200513	li a0, 2
101a0:	fa5ff0ef	jal ra, 10144 <func>
101a4:	c4a1ac23	sw a0, -936(gp) # 11a38 <y>
101a8:	00c12083	lw ra, 12(sp)
101ac:	01010113	addi sp, sp, 16
101b0:	00008067	ret



Put 2 and 3 in `f` and `g` (and argument registers) and call `func`. Then put result in `y` and return.

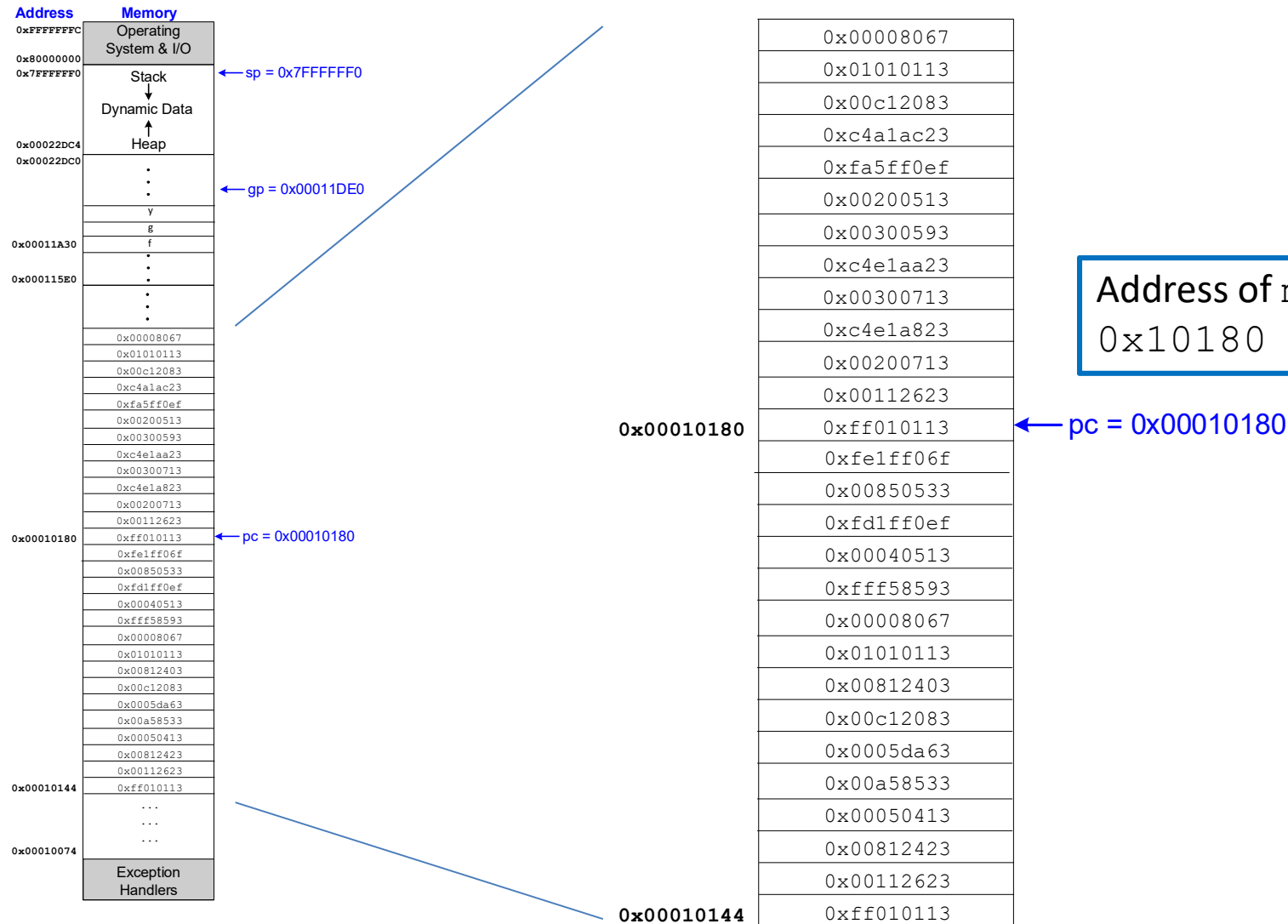
# Example Program: Symbol Table

Address				Size	Symbol Name
00010074	l	d	.text	00000000	.text
000115e0	l	d	.data	00000000	.data
00010144	g	F	.text	0000003c	func
00010180	g	F	.text	00000034	main
00011a30	g	O	.bss	00000004	f
00011a34	g	O	.bss	00000004	g
00011a38	g	O	.bss	00000004	y

- text segment: address 0x10074
- data segment: address 0x115e0
- func function: address 0x10144 (size 0x3c bytes)
- main function: address 0x10180 (size 0x34 bytes)
- f: address 0x11a30 (size 0x4 bytes)
- g: address 0x11a34 (size 0x4 bytes)
- y: address 0x11a38 (size 0x4 bytes)



## Example Program in Memory



# RISC-V Extensions

- Extensions define the set of supported instructions
  - “I” for Integer is the only required extension in a RISC-V implementation and defines 40 basic instructions
- Standard Extensions are defined by the RISC-V Foundation and are optional

Extension	Description
I	Integer
M	Multiplication and Division
A	Atomics
F	Single-Precision Floating Point
D	Double-Precision Floating Point
G	General Purpose = IMAFD
C	16-bit Compressed Instructions

- RISC-V allows for custom, “Non-Standard”, extensions in an implementation (Xext)
- Putting it all together (examples)
  - RV32I: the most basic RISC-V implementation
  - RV32IMAC: Integer + Multiply + Atomic + Compressed
  - RV64GC: 64bit IMAFDC
  - RV64GCXext: IMAFDC + a non-standard extension