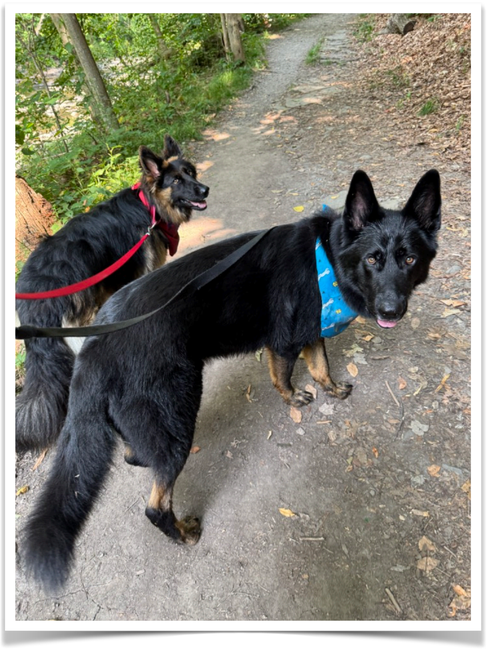


CS3410: Computer Systems and Organization

LEC12: Review + RISC-V Calling Convention (Wednesday)

Professor Giulia Guidi
Monday, October 6, 2025

Credits: Bala, Bracy, Garcia, Guidi, Kao, Sampson, Sirer, Weatherspoon

CC BY NC SA

Plan for Today

- Review: Control Flow, Logical Instruction (self study), and Memory
- RISC-V Calling Convention (Wednesday)

CC BY NC SA

Review of RISC-V control flow

CC BY NC SA

Computer Decisions Making

I.e., based on computation, do something different

- In programming languages, *if*-statement

RISC-V *if*-statement instruction:

```
beq rs1, rs2, .label
```

It marks the **address of an instruction**

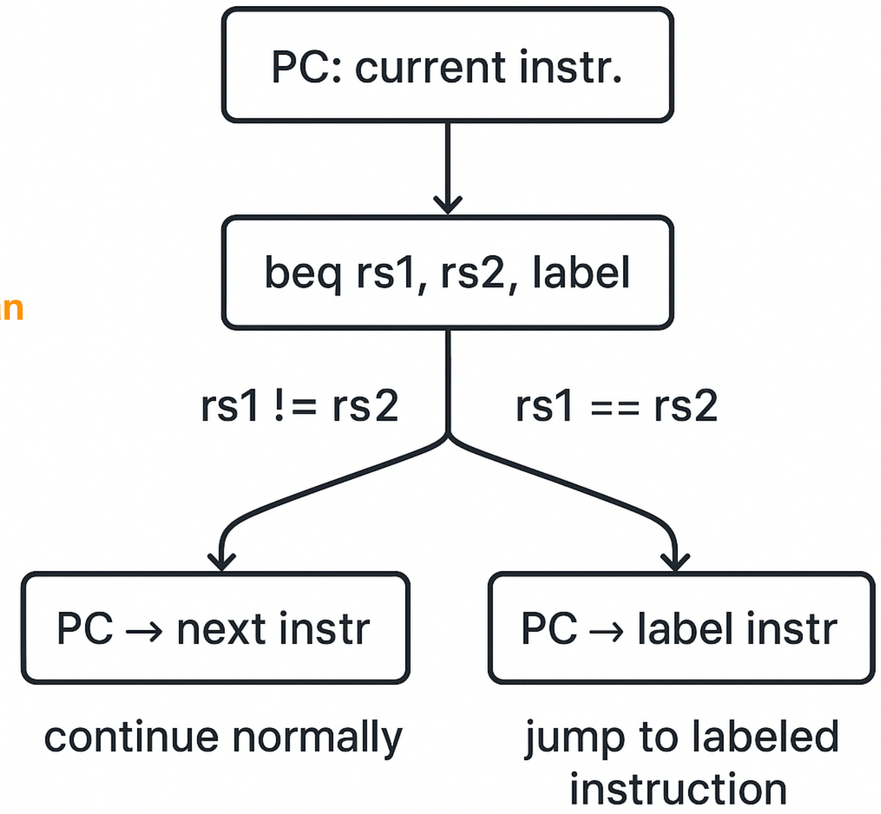
"branch if equal"

→ If `rs1` == `rs2`, then go to the instruction at `label`

```
bne rs1, rs2, label
```

"branch if **not** equal"

→ If `rs1` != `rs2`, then go to the instruction at `label`



CC BY NC SA

Types of Branches

- **Branch:** change of control flow
- **Conditional branch:** change control flow depending on outcome of comparison

`beq, bne`	
`blt, bge`	"branch if **less than**", "branch if **greater or equal than**"
`bltu, bgeu`	"branch if **less than**" unsigned, "branch if **greater than or equal to**" unsigned

→ `beq, bne, blt, etc.` do **not** update any register in the register file

→ Branches only affect **program counter** (PC)

CC BY NC SA

Types of Branches

- **Branch:** change of control flow
- **Conditional branch:** change control flow depending on outcome of comparison

`beq, bne`	
`blt, bge`	"branch if **less than**", "branch if **greater or equal than**"
`bltu, bgeu`	"branch if **less than**" unsigned, "branch if **greater than or equal to**" unsigned

- **Unconditional branch:** always branch

```
jump (j)      j label == beq x0, x0, label
```

Review lecture 11 by Prof. Guidi and notes on the course website

CC BY NC SA

Example *if* Statement

Let us assume the translations below, **compile** *if* block

```
f → x10, g → x11, h → x12, i → x13, j → x14
```

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

```
bne x13, x14, Exit
add x10, x11, x12
Exit: % terminate
```


CC BY NC SA

Example *if-else* Statement

Let us assume the translations below, **compile** *if-else* block

```
f → x10, g → x11, h → x12, i → x13, j → x14
```

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

```
bne x13, x14, Else
add x10, x11, x12
j Exit
Else: sub x10, x11, x12
Exit: % terminate
```


CC BY NC SA

Loops in C and Assembly

3 types of **loops**:

- `while`
- `do … while`
- `for`

Each can be rewritten as either of the other two!

```c
int main() {
    int i = 0;
    while (i < 10) {
        printf("%d\n", i);
        i++;
    }
    return 0;
}
```

```c
int main() {
    for (int i = 0; i < 10; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

```c
int main() {
    int i = 0;
    do {
        printf("%d\n", i);
        i++;
    } while (i < 10);
    return 0;
}
```


CC BY NC SA

Loops in C and Assembly

```
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

Review lecture 11 by Prof. Guidi and notes on the course website

CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

By using x9, x8 is unchanged—It's useful if you need to reference the start of the array A again later

```
add  x9,  x8, x0   # x9  = &A[0]
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

By using x9, x8 is unchanged—It's useful if you need to reference the start of the array A again later

```
add  x9,  x8, x0   # x9  = &A[0]
add  x10, x0, x0   # sum = 0
```

Review lecture 11 by Prof. Guidi and notes on the course website

CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

By using x9, x8 is unchanged—It's useful if you need to reference the start of the array A again later

```
add  x9,  x8, x0   # x9  = &A[0]
add  x10, x0, x0   # sum = 0
add  x11, x0, x0   # i   = 0
```

Review lecture 11 by Prof. Guidi and notes on the course website

CC BY NC SA

Loops in C and Assembly

By using x9, x8 is unchanged—It's useful if you need to reference the start of the array A again later

```
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
add  x9,  x8, x0    # x9  = &A[0]
add  x10, x0, x0    # sum = 0
add  x11, x0, x0    # i   = 0
addi x13, x0, 20    # x13 = 20
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

By using x9, x8 is unchanged—It's useful if you need to reference the start of the array A again later

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

By using x9, x8 is unchanged—It's useful if you need to reference the start of the array A again later

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
    bge  x11, x13, Done
```

You can only compare branches using registers—register to register, **not** e.g., branch less than immediate

CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

By using x9, x8 is unchanged—It's useful if you need to reference the start of the array A again later

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
    bge  x11, x13, Done
```

Ok, why `bge` and not `blt`?

If that test fails, you never enter the for loop—branches usually point to the **exit**, not the body

CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

By using x9, x8 is unchanged—It's useful if you need to reference the start of the array A again later

```
  add  x9,  x8, x0    # x9  = &A[0]
  add  x10, x0, x0    # sum = 0
  add  x11, x0, x0    # i   = 0
  addi x13, x0, 20    # x13 = 20
Loop:
  bge  x11, x13, Done
  lw   x12, 0(x9)     # x12  = A[i]
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

By using x9, x8 is unchanged—It's useful if you need to reference the start of the array A again later

```
  add  x9,  x8, x0   # x9  = &A[0]
  add  x10, x0, x0   # sum = 0
  add  x11, x0, x0   # i   = 0
  addi x13, x0, 20   # x13 = 20
Loop:
  bge  x11, x13, Done
  lw   x12, 0(x9)    # x12  = A[i]
  add  x10, x10, x12 # sum += x12
```


CC BY NC SA

Loops in C and Assembly

By using x9, x8 is unchanged—It's useful if you need to reference the start of the array A again later

```
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
    bge  x11, x13, Done
    lw   x12, 0(x9)     # x12  = A[i]
    add  x10, x10, x12  # sum += x12
    addi x9,  x9,  4    # &A[i+1]
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

By using x9, x8 is unchanged—It's useful if you need to reference the start of the array A again later

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
    bge  x11, x13, Done
    lw   x12, 0(x9)     # x12  = A[i]
    add  x10, x10, x12  # sum += x12
    addi x9,  x9,  4    # &A[i+1]
    addi x11, x11, 1    # i++
```


CC BY NC SA

Loops in C and Assembly

By using x9, x8 is unchanged—It's useful if you need to reference the start of the array A again later

```
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

```
  add  x9,  x8, x0    # x9  = &A[0]
  add  x10, x0, x0    # sum = 0
  add  x11, x0, x0    # i   = 0
  addi x13, x0, 20    # x13 = 20
Loop:
  bge  x11, x13, Done
  lw   x12, 0(x9)     # x12  = A[i]
  add  x10, x10, x12  # sum += x12
  addi x9,  x9,  4    # &A[i+1]
  addi x11, x11, 1    # i++
  j Loop
```


CC BY NC SA

Loops in C and Assembly

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

E: Can you write this using `blt` instead of `bge`? If so, how many conditional jumps and how many unconditional jumps? Can you rewrite it to decrease `i` from `20` to `0`?

By using x9, x8 is unchanged—It's useful if you need to reference the start of the array A again later

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
    bge  x11, x13, Done
    lw   x12, 0(x9)     # x12  = A[i]
    add  x10, x10, x12  # sum += x12
    addi x9,  x9,  4    # &A[i+1]
    addi x11, x11, 1    # i++
    j Loop
Done:
```


CC BY NC SA

`blt` Fall-Through

```c
int A[20];
// fill A with data
int sum = 0;
for (int i = 0; i < 20; i++)
    sum += A[i];
```

This **fall-through** approach works and uses only one conditional branch in this example, but if the loop bound is zero—or, more generally, if the code might never take the branch—it still executes the loop body once. This is inefficient or even incorrect for large loops or performance-critical code.

```
  add  x9,  x8, x0    # x9  = &A[0]
  add  x10, x0, x0    # sum = 0
  add  x11, x0, x0    # i   = 0
  addi x13, x0, 20    # x13 = 20
Loop:
  lw   x12, 0(x9)     # x12  = A[i]
  add  x10, x10, x12  # sum += x12
  addi x9,  x9,  4    # &A[i+1]
  addi x11, x11, 1    # i++
  blt  x11, x13, Loop# if (i < 20)
goto Loop
Done:
```


CC BY NC SA

Poll

Change the code to sum only the even-indexed elements of the array A:

```
    add  x9,  x8, x0    # x9  = &A[0]
    add  x10, x0, x0    # sum = 0
    add  x11, x0, x0    # i   = 0
    addi x13, x0, 20    # x13 = 20
Loop:
    bge  x11, x13, Done
    lw   x12, 0(x9)     # x12  = A[i]
    add  x10, x10, x12  # sum += x12
    addi x9,  x9,  4    # &A[i+1]
    addi x11, x11, 1    # i++
    j Loop
Done:
```


CC BY NC SA

Ok, review (self study): logical instruction

Review lecture 9 by Dr. Laeufer and notes on logical instructions

CC BY NC SA

RISC-V Logical Instruction

- Useful to operate on fields of bits within a word
  - E.g., characters within a word (8 bits)
- Operations to pack/unpack bits into words
- Called logical operations

Does not have **`not`**

Logical Op	C Op	Java Op	RISC-V Instruction
Bit-by-bit AND	&	&	`and`
Bit-by-bit OR	|	|	`or`
Bit-by-bit XOR	^	^	`xor`
Left logical shift	<<	<<	`sll`
Right logical shift	>>	>>	`srl`

Review lecture 9 by Dr. Laeufer and notes on logical instructions

CC BY NC SA

RISC-V Logical Instruction

- Two variants
  - Register: `and  x5, x6, x7 # x5 = x6 & x7`
  - Immediate: `andi x5, x6, 3  # x5 = x6 &  3`

`and rd, rs1, rs2` meaning `rd = rs1 & rs2`

$rs1 = 1101_2$

$rs2 = 1011_2$

Bit position	rs1	rs2	rs1 & rs2	rs1 | rs2	rs1 ^ rs2
3 (MSB)	1	1	1	1	0
2	1	0	0	0	1
1	0	1	0	0	1
0 (LSB)	1	1	1	1	0
Result (rd)			**1001_2**	**1111_2**	**0110_2**

Review lecture 9 by Dr. Laeufer and notes on logical instructions

CC BY NC SA

RISC-V Logical Instruction

- Two variants
  - Register: `and  x5, x6, x7 # x5 = x6 & x7`
  - Immediate: `andi x5, x6, 3  # x5 = x6 &  3`

- Used for “masks”
  - `andi` with `0000 00FF`$_{hex}$ isolates the **least** significant byte
  - `andi` with `FF00 0000`$_{hex}$ isolates the **most** significant byte

Review lecture 9 by Dr. Laeufer and notes on logical instructions

CC BY NC SA

RISC-V no NOT

- There's no logical **NOT** in RISC-V
  - Use `xor` with 11111111_{two}

In a circuit, think of **y** as "Do you want to flip the value?"—**0** means "I don't want to flip the value" while **1** means "I want to flip the value"

In a circuit, think of **x** as the input signal

XOR is a conditional inverter

x	y	XOR(x, y)
0	0	0
0	1	1
1	0	1
1	1	0

CC BY NC SA

Ok, let’s review memory allocation and stack

CC BY NC SA

If I call a function in C, where do its variables go?

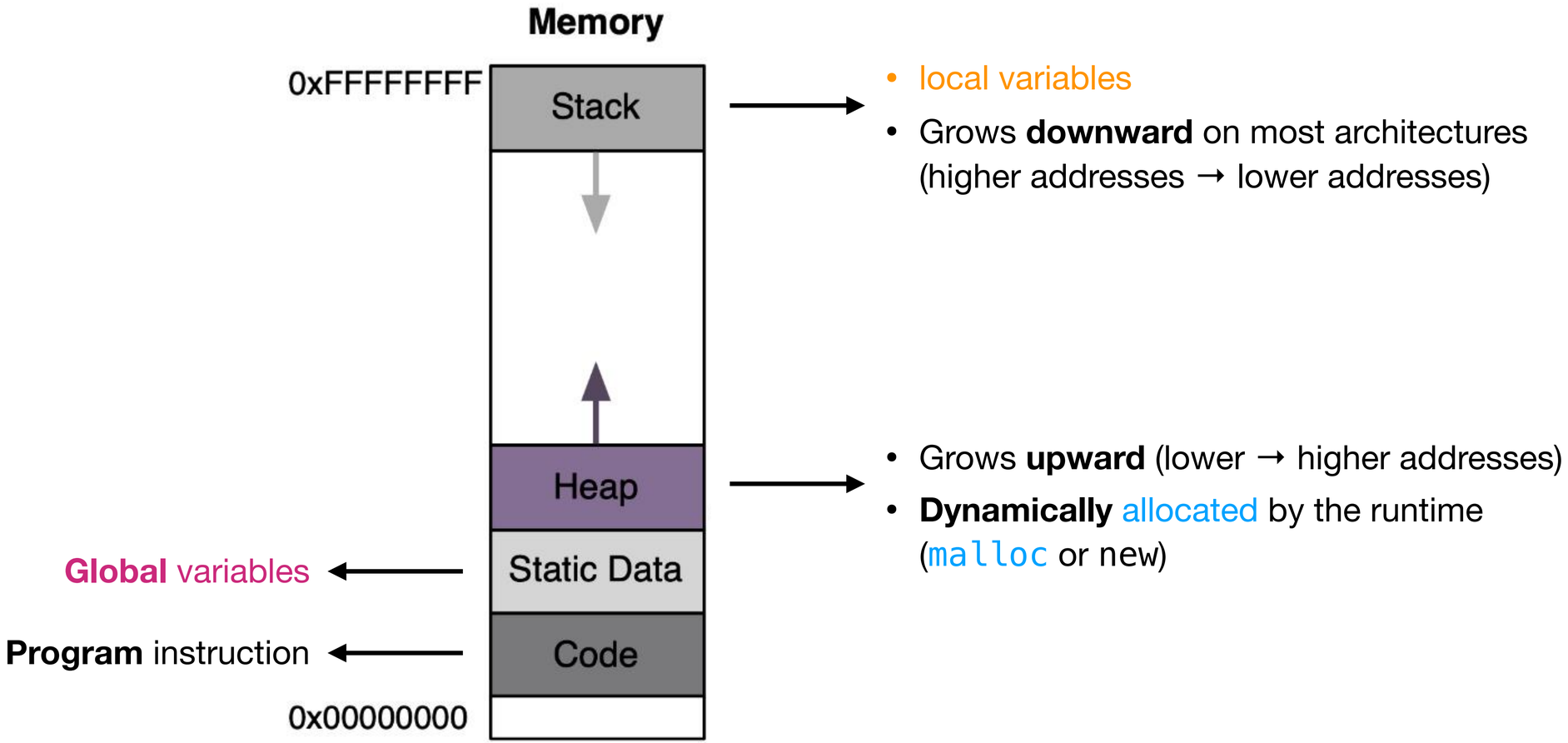
The table summarizes **where** different types of **variables** live in memory, **how long they exist**, and **when** they are automatically or manually released

The storage duration	Declared in?	The memory area	The lifetime	Freed when?
static	global or static	Data	entire program	program exit
automatic	local variables	Stack	function entry → exit	return
allocated	via `malloc`	Heap	until `free`	explicit free

CC BY NC SA

If I call a function in C, where do its variables go?

The three regions — stack, heap, static — correspond directly to the three storage durations



CC BY NC SA

Stack and Heap

In which memory area do the variables highlighted in **bold** live?

Code A:

```
#include <stdio.h>
void f(int x) {
    int y = 42;
    printf("f: x=%d, y=%d\n", x, y);
}
int main(void) {
    int a = 10;
    f(a);
    f(20);
}
```

Code B:

```
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    int *p = malloc(sizeof(int));
    *p = 5;
    printf("%d\n", *p);
    free(p);
}
```


CC BY NC SA

Stack and Heap

In which memory area do the variables highlighted in **bold** live?

Code A:

```
#include <stdio.h>

void f(int x) {
    int y = 42;    // stack
    printf("f: x=%d, y=%d\n", x, y);
}

int main(void) {
    int a = 10;    // stack
    f(a);
    f(20);
}
```

Code B:

```
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    int *p = malloc(sizeof(int));
    *p = 5;
    printf("%d\n", *p);
    free(p);
}
```

// ***p** pointee is on the heap

// **p** pointer is on the stack

CC BY NC SA

Stack and Heap

In which memory area do the variables highlighted in **bold** live?

Code A:

```
#include <stdio.h>
void f(int x) {
    int y = 42;    // stack
    printf("f: x=%d, y=%d\n", x, y);
}
int main(void) {
    int a = 10;    // stack
    f(a);
    f(20);
}
```

Code C:

```
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    int x = 5;
    int *p = &x;
    printf("%d\n", *p);
    free(p);
}
```

// ***p** pointee is on the stack

// **p** pointer is on the stack

CC BY NC SA

Poll

Q: spot the bug!

Code C:

```
int *make_it(void) {
    int val = 7;
    return &val;
}
```

PollEv.com /gguidi

Or send **gguidi** to **22333**

CC BY NC SA

On to stack [not in the prelim]

PS: this is needed for **A7** + notes on the course website

CC BY NC SA

Stack

A stack is like a vertical stack of boxes: You add (**push**) boxes on top and take (**pop**) boxes from the top

Code A:

```
#include <stdio.h>

void f(int x) {
    int y = 42;   // stack
    printf("f: x=%d, y=%d\n", x, y);
}

int main(void) {
    int a = 10;   // stack
    f(a);
    f(20);
}
```


CC BY NC SA

Stack

Let us draw **stack frames** after each call:

Code A:

```c
#include <stdio.h>

void f(int x) {
    int y = 42;   // stack
    printf("f: x=%d, y=%d\n", x, y);
}

int main(void) {
    int a = 10;   // stack
    f(a);
    f(20);
}
```

Before `f()`:

CC BY NC SA

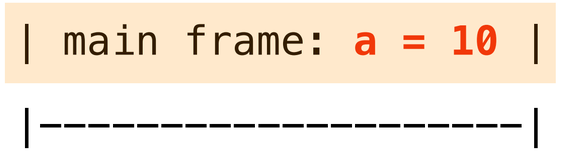
Stack

Let us draw **stack frames** after each call:

Code A:

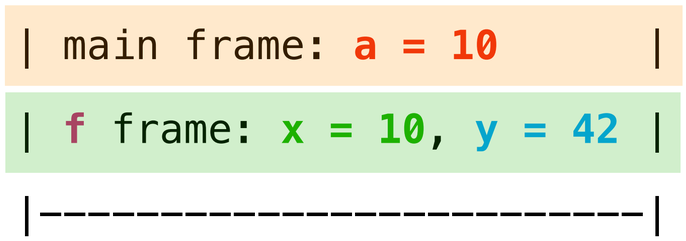
```c
#include <stdio.h>

void f(int x) {
    int y = 42;    // stack
    printf("f: x=%d, y=%d\n", x, y);
}

int main(void) {
    int a = 10;    // stack
    f(a);
    f(20);
}
```

Before `f()`:



This frame sits at e.g., **0x1000 + 12**

Once inside `f(10)`:



This frame sits at e.g., **0x1000 + 12**

CPU **pushes** a new **stack frame** onto the stack

CC BY NC SA

Stack

Let us draw **stack frames** after each call:

Code A:

```
#include <stdio.h>
void f(int x) {
    int y = 42;    // stack
    printf("f: x=%d, y=%d\n", x, y);
}
int main(void) {
    int a = 10;    // stack
    f(a);
    f(20);
}
```

Before `f()`:

```
| main frame: a = 10 |
|--------------------|
```

This frame sits at e.g., **0x1000 + 12**

Once inside `f(10)`:

```
| main frame: a = 10         |
| f frame: x = 10, y = 42    |
|----------------------------|
```

This frame sits at e.g., **0x1000 + 12**

This frame sits at e.g., **0x1000 + 8**

CC BY NC SA

Ex: Stack

```
int main() {
    uint8_t a = 0;   // 1 byte
    uint8_t b = 1;   // 1 byte
    uint8_t *p = &a; // pointer (4 bytes on 32-bit or 8 bytes on 64-bit)
}
```

1. Push a onto the stack

```
address    | variable | value
---------------------------
0x… + 12   | a        | 0
           |          |
           |          |
```

PS: This is the same example Dr. Laeufer used in Lecture 5

CC BY NC SA

Ex: Stack

```
int main() {
    uint8_t a = 0;   // 1 byte
    uint8_t b = 1;   // 1 byte
    uint8_t *p = &a; // pointer (4 bytes on 32-bit or 8 bytes on 64-bit)
}
```

1. Push b onto the stack

```
address    | variable | value
---------------------------
0x… + 12   | a        | 0
0x… + 8    | b        | 1
           |          |
```

PS: This is the same example Dr. Laeufer used in Lecture 5

CC BY NC SA

Ex: Stack

```
int main() {
    uint8_t a = 0;   // 1 byte
    uint8_t b = 1;   // 1 byte
    uint8_t *p = &a; // pointer (4 bytes on 32-bit or 8 bytes on 64-bit)
}
```

1. Push p onto the stack

```
address   | variable | value
--------------------------
0x… + 12  | a        | 0
0x… + 8   | b        | 1
0x… + 4   | p        | points to `a`
```

PS: This is the same example Dr. Laeufer used in Lecture 5

CC BY NC SA

Ex: Stack

- a and b are 1 byte each, but compilers usually **align** variables to 4 bytes, so each may take 4 bytes

- p is a pointer, so it's 4 bytes on 32-bit or 8 bytes on 64-bit

1. Push p onto the stack

```
address    | variable | value
---------------------------
0x… + 12   | a        | 0
0x… + 8    | b        | 1
0x… + 4    | p        | points to `a`
```

PS: This is the same example Dr. Laeufer used in Lecture 5

CC BY NC SA

On to calling convention [not in the prelim]

PS: this is needed for **A7** + notes on the course website

CC BY NC SA

Purpose of Calling Convention

A calling convention is a set of rules that defines **how functions communicate**:

- E.g., how **arguments** are passed
- E.g., how **return values** are returned
- E.g., how **registers** and the **stack** are managed
- E.g., **who** is responsible for saving/restoring what

The essentially the "**contract**" between the caller (e.g., `main`) and callee, e.g., function `f()`

CC BY NC SA

RISC-V Calling Convention

A calling convention is a set of rules that defines **how functions communicate**:

Register	Use
`x1 → ra`	The **return** address
`x2 → sp`	The **stack** pointer
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved)
`x10–x17 → a0–a7`	The function **arguments** and **return** values (`a0–a1` only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved)

Caller-saved: Caller must save if it wants the value after the call

Callee-saved: Callee must preserve them across the call

CC BY NC SA

RISC-V Calling Convention

A calling convention is a set of rules that defines **how functions communicate**:

Register	Use
`x1 → ra`	The **return** address — If `main` calls a function `add`, **`ra`** stores the address of the instruction following the call to `add` in `main`
`x2 → sp`	The **stack** pointer — **`sp`** stores the address in memory of the top of the **stack**
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved) — The subroutine **can** modify these values
`x10–x17 → a0–a7`	The function **arguments** and **return** values (`a0–a1` only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved)

Caller-saved: Caller must save if it wants the value after the call

Callee-saved: Callee must preserve them across the call

CC BY NC SA

RISC-V Calling Convention

I have **8** registers for the **arguments**. But what happens if my function has **10** arguments?

Register	Use
`x1 → ra`	The **return** address
`x2 → sp`	The **stack** pointer
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved)
`x10–x17 → a0–a7`	The function **arguments** and **return** values (`a0–a1` only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved)

CC BY NC SA

RISC-V Calling Convention

I have **8** registers for the **arguments**. But what happens if my function has **10** arguments?

Register	Use
`x1 → ra`	The **return** address
`x2 → sp`	The **stack** pointer
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved)
`x10–x17 → a0–a7`	The function **arguments** and **return** values (`a0–a1` only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved)

The first 8 integer arguments → `a0–a7`

Remaining arguments → pushed onto the **stack**

CC BY NC SA

RISC-V Calling Convention

A calling convention is a set of rules that defines **how functions communicate**:

Register	Use
`x1 → ra`	The **return** address — If `main` calls a function `add`, **`ra`** stores the address of the instruction following the call to `add` in `main`
`x2 → sp`	The **stack** pointer — **`sp`** stores the address in memory of the top of the **stack**
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved) — The subroutine **can** modify these values
`x10–x17 → a0–a7`	The function **arguments** and **return** values (`a0–a1` only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved) — The subroutine e.g., `add` can touch these values, but it **must** restore them

Caller-saved: Caller must save if it wants the value after the call

Callee-saved: Callee must preserve them across the call

CC BY NC SA