

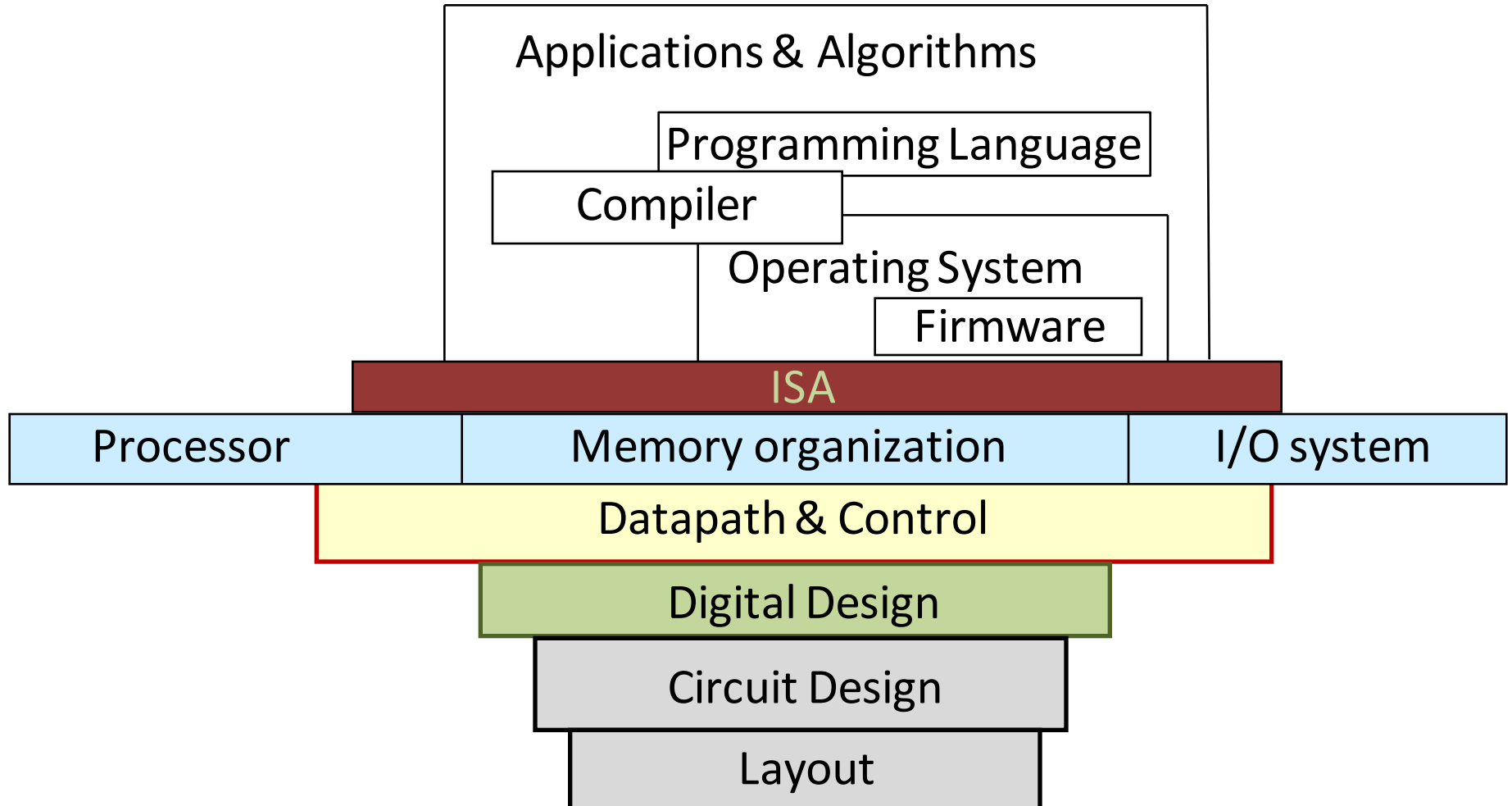
Introduction to Cybersecurity

A Software/Hardware Approach

C Programming & Computer Organization

Prof. Michel A. Kinsy

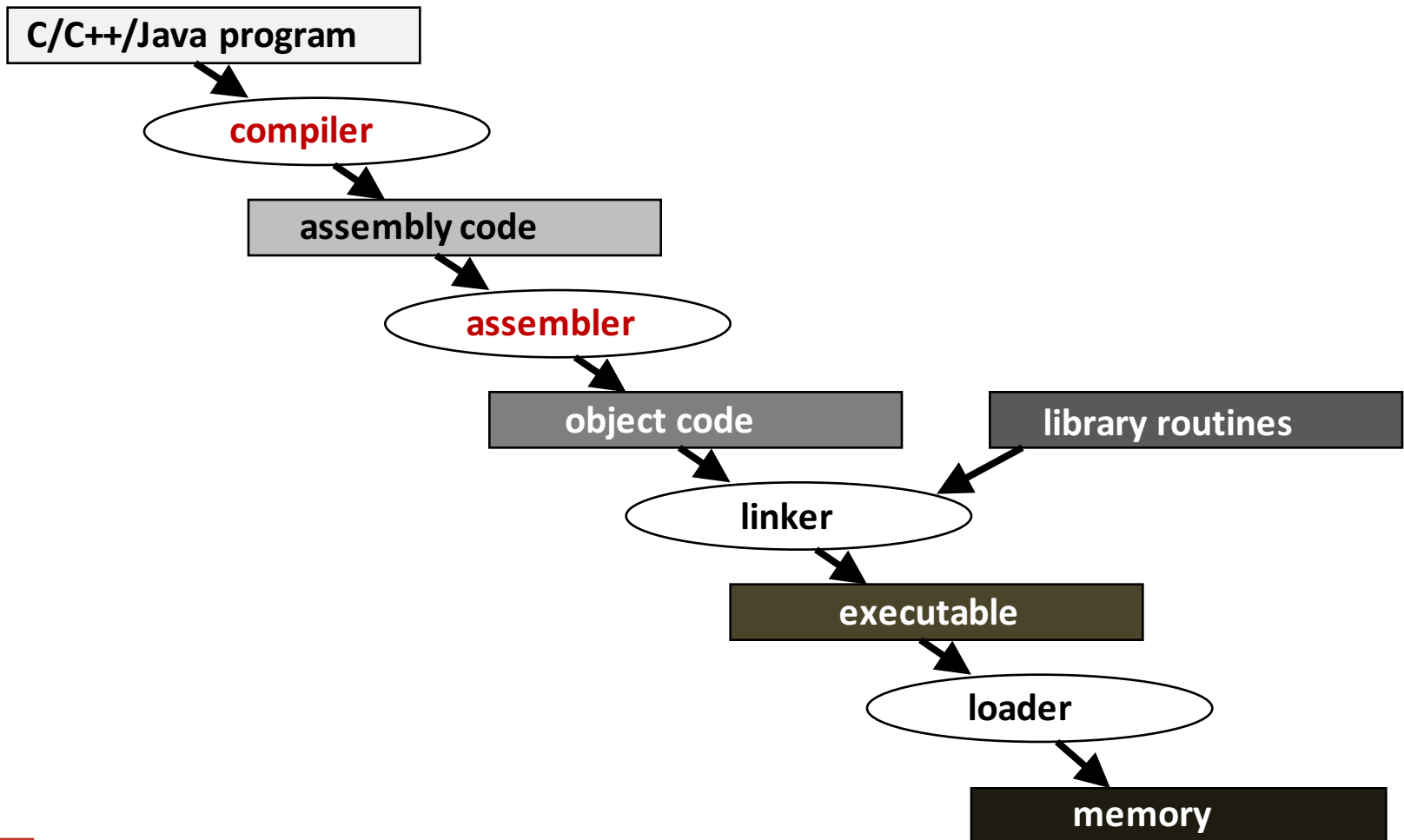
The Computing Stack



Bridging/Compiling Process

- High-Level Language

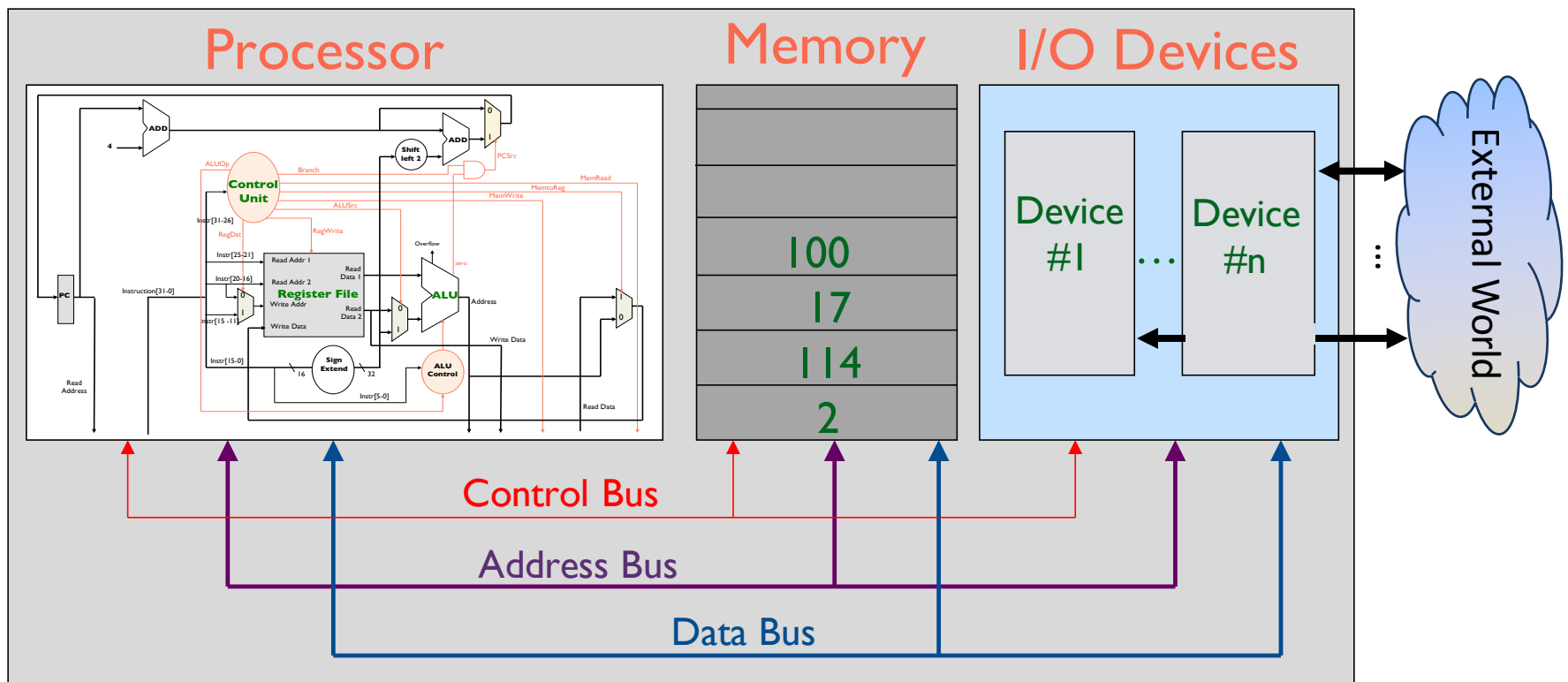
Human
Readable



Machine
Code

The Overall Organization!

- The modern computer system has three major functional hardware units: CPU (Processing Engine), Main Memory (Storage) and Input/Output (I/O) Units



What does any language need to do?

Language Perspective

1. Declare and initialize variables
2. Access variables
3. Control flow of execution
4. Use data structures
5. Execute statements

Potential Attack Vectors

Language Prospective

```
// Fibonacci
int fib(int n) {
    if (n <= 1) {
        return n;
    }
    else {
        return fib(n-1)+fib(n-2);
    }
}

int main (void) {
    int number = 4;
    int result = 0;

    result = fib(number);

    return result;
}
```

Greatest Common Divisor Example

- Simple task
 - Albert would like to compute the greatest common divisor (GCD) of two numbers
- How can this be done?
 - Albert, you should know how to do this from your introductory to discrete mathematics course!
- The Euclidean iterative approach
 - $r_{k-2} = q_k r_{k-1} + r_k$
 - where r_k is strictly less than that of r_{k-1}
- Using the modulo operation
 - $r_k = r_{k-2} \bmod r_{k-1}$

Greatest Common Divisor Example

- The GCD translation from the mathematical form
 - $r_k = r_{k-2} \bmod r_{k-1}$
- In to the computer algorithmic form
 - But what if we do not have a modulo/division operation in our computer system?
 - The alternative

```

//Euclidean algorithm
function gcd(a:int, b:int):int
    var tmp:int
    if(a < b)
        tmp = a
        a = b
        b = tmp
    //Find the gcd
    while(b != 0)
        while (a >= b)
            a = a - b
        tmp = a
        a = b
        b = tmp
    return a
    
```


Greatest Common Divisor Example

- The GCD translation from the mathematical form
 - $r_k = r_{k-2} \bmod r_{k-1}$
- In to the computer algorithmic form
 - The alternative
 - It could even be made prettier

```

//Euclidean algorithm
function gcd(a:int, b:int):int
    var tmp:int
    if(a < b)
        tmp = a
        a = b
        b = tmp
    //Find the gcd
    while(b != 0)
        while (a >= b)
            a = a - b
        tmp = a
        a = b
        b = tmp
    return a
    
```

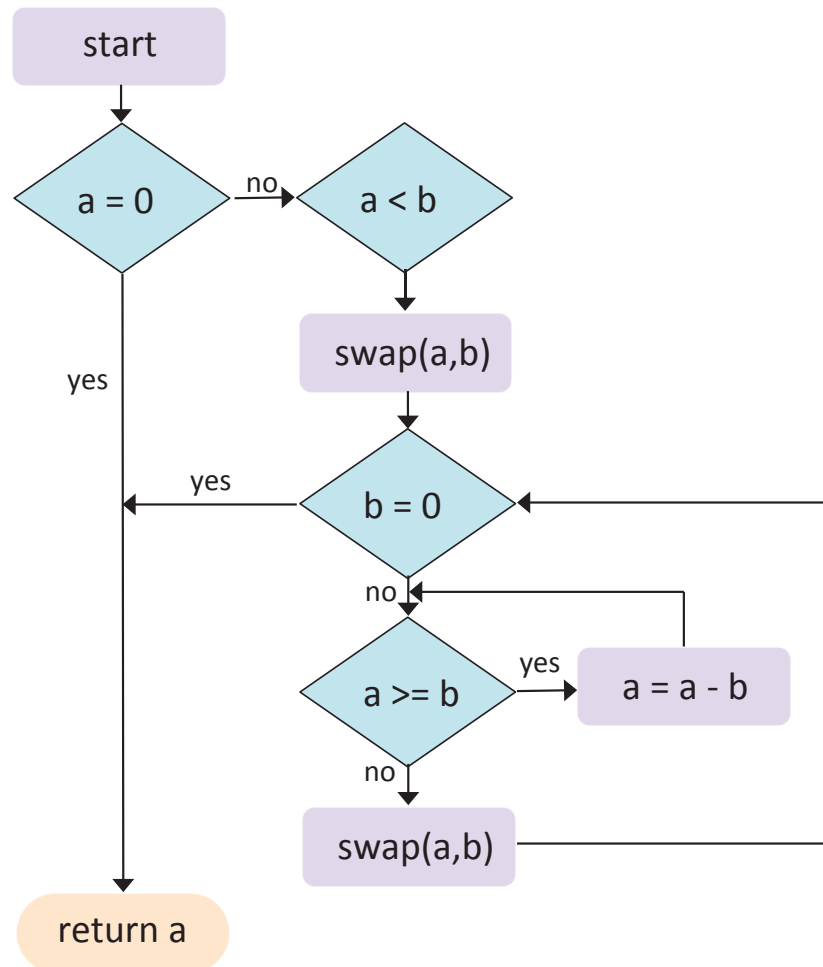
```

//Euclidean algorithm
function gcd(a:int, b:int):int
    var tmp:int
    if(a < b)
        swap(a,b)
    //Find the gcd
    while(b != 0)
        while (a >= b)
            a = a - b
        swap(a,b)
    return a
    
```

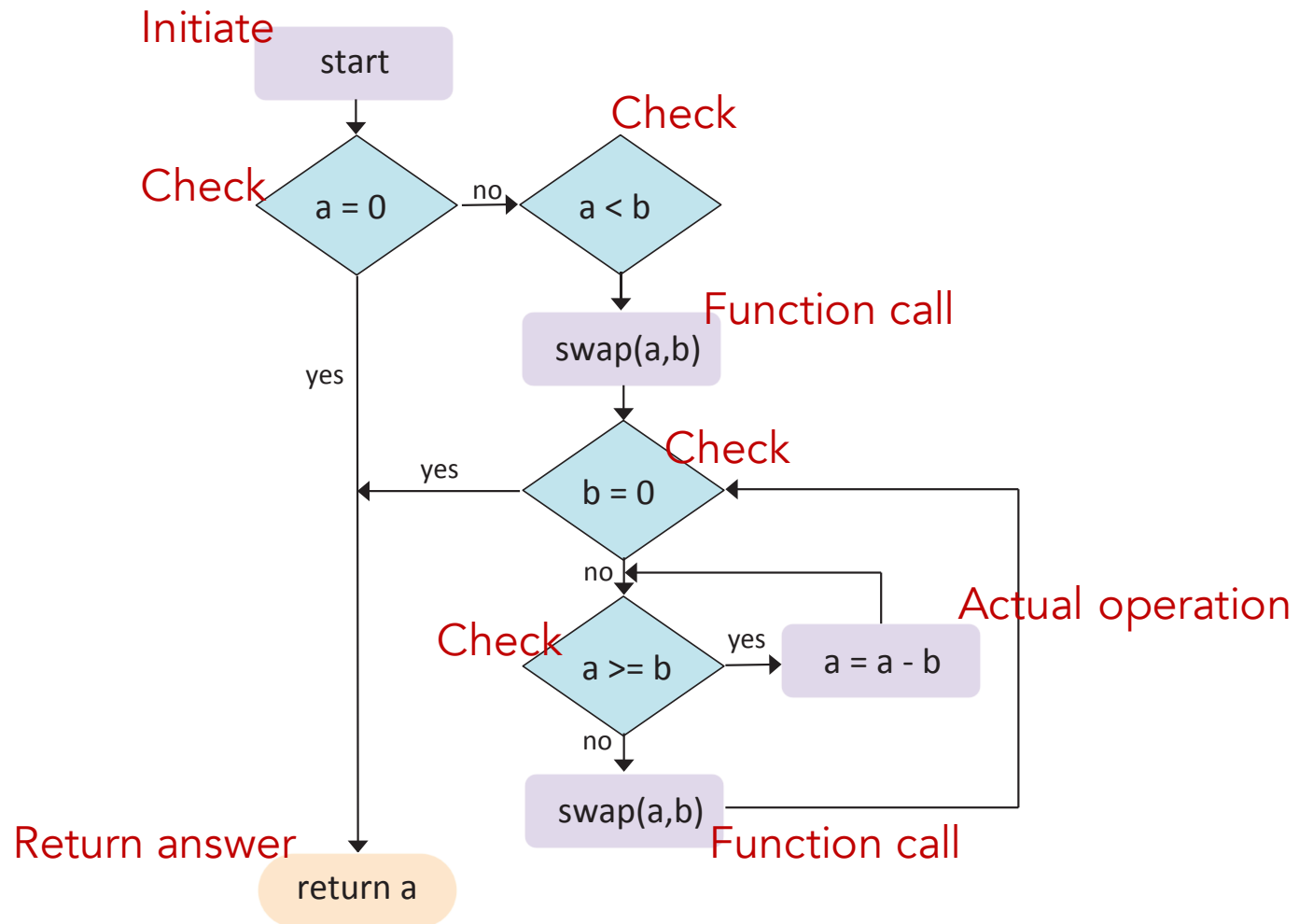
Greatest Common Divisor Example

```

//Euclidean algorithm
function gcd(a:int, b:int):int
var tmp:int
if(a < b)
    swap(a,b)
//Find the gcd
while(b != 0)
    while (a >= b)
        a = a - b
    swap(a,b)
return a
    
```



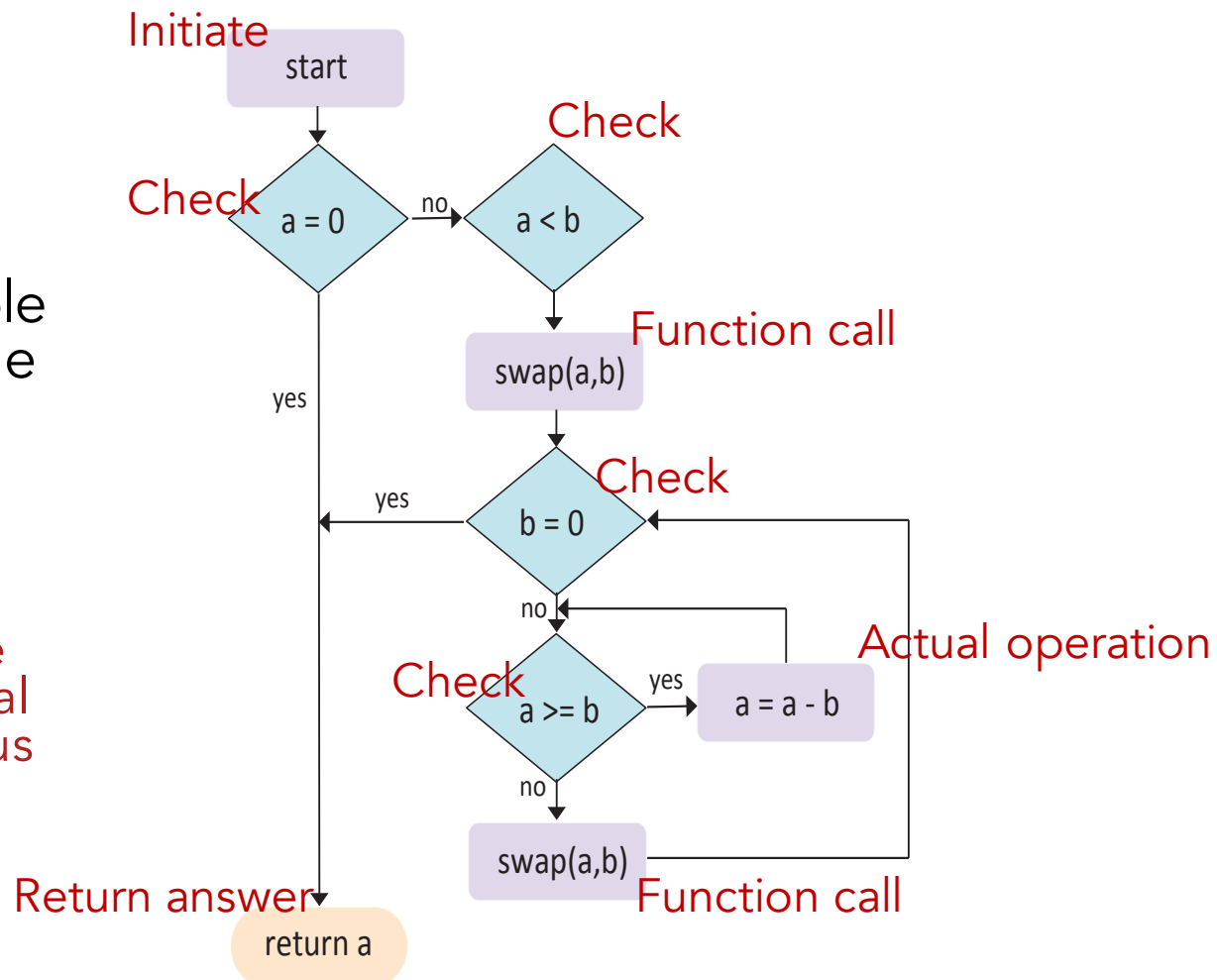
Greatest Common Divisor Example



Greatest Common Divisor Example

- This simple example already reveal some of the key underpinnings of computer organization

- For example the mixture of logical operations versus arithmetic ones



Greatest Common Divisor Example

```

int gcd (int a, int b) {
    int tmp;
    if(a < b) {
        tmp = a;
        a = b;
        b = tmp;
    }
    //Find the gcd
    while(b != 0) {
        while (a >= b) {
            a = a - b;
        }
        tmp = a;
        a = b;
        b = tmp;
    }
    return a;
}
    
```

```

#include <stdio.h>
// GCD function goes here
int main(void) {
    int a, b, answer;
    printf("Enter positive integers a and b: ");
    scanf("%d %d",&a,&b);
    answer = gcd(a, b);
    printf("GCD = %d",answer);
    return 0;
}
    
```

```

void swap (int a, int b) {
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
    
```

```

int gcd2 (int a, int b) {
    if(a < b) {
        swap(a,b);
    }
    //Find the gcd
    while(b != 0) {
        while (a >= b) {
            a = a - b;
        }
        swap(a,b);
    }
    return a;
}
    
```

Greatest Common Divisor Example

```

int gcd (int a, int b) {
    int tmp;
    if(a < b) {
        tmp = a;
        a = b;
        b = tmp;
    }
    //Find the gcd
    while(b != 0) {
        while (a >= b) {
            a = a - b;
        }
        tmp = a;
        a = b;
        b = tmp;
    }
    return a;
}
    
```

```

#include <stdio.h>
// GCD function goes here
int main(void) {
    int a, b, answer;
    printf("Enter positive integers a and b: ");
    scanf("%d %d",&a,&b);
    answer = gcd(a, b);
    printf("GCD = %d",answer);
    return 0;
}
    
```

```

void swap2(int *a, int *b){
    int tmp;
    tmp = *b;
    *b = *a;
    *a = tmp;
}
    
```

```

int gcd2 (int a, int b) {
    if(a < b) {
        swap(&a,&b);
    }
    //Find the gcd
    while(b != 0) {
        while (a >= b) {
            a = a - b;
        }
        swap(&a,&b);
    }
    return a;
}
    
```

Greatest Common Divisor Example

```

int gcd (int a, int b) {
    int tmp;
    if(a < b) {
        tmp = a;
        a = b;
        b = tmp;
    }
    //Find the gcd
    while(b != 0) {
        while (a >= b) {
            a = a - b;
        }
        tmp = a;
        a = b;
        b = tmp;
    }
    return a;
}
    
```

```

#include <stdio.h>
// GCD function goes here
int main(void) {
    int a, b, answer;
    printf("Enter positive integers a and b: ");
    scanf("%d %d",&a,&b);
    answer = gcd(a, b);
    printf("GCD = %d",answer);
    return 0;
}
    
```

```

void swap2(int *a, int *b){
    int tmp;
    tmp = *b;
    *b = *a;
    *a = tmp;
}
    
```

```

int gcd2 (int a, int b) {
    if(a < b) {
        swap(&a,&b);
    }
    //Find the gcd
    while(b != 0) {
        while (a >= b) {
            a = a - b;
        }
        swap(&a,&b);
    }
    return a;
}
    
```

- Later in assembly programming we will observe the call structure runtime behavior

Greatest Common Divisor Example

```

int gcd (int a, int b) {
    int tmp;
    if(a < b) {
        tmp = a;
        a  = b;
        b  = tmp;
    }
    //Find the gcd
    while(b != 0) {
        while (a >= b) {
            a = a - b;
        }
        tmp = a;
        a  = b;
        b  = tmp;
    }
    return a;
}
    
```

- From C to assembly, the translation is straightforward
- We will see more later

```

main:
    sd ra,24(sp)
    ..
    call printf
    addi a4,s0,-28
    ...
    call scanf
    lw a5,-24(s0)
    lw a4,-28(s0)
    mv a1,a4
    mv a0,a5
    call gcd(int, int)
    mv a5,a0
    sw a5,-20(s0)
    ...
    call printf
    ...
    addi sp,sp,32
    jr ra
    
```


What does any language need to do?

Language Perspective

1. Declare and initialize variables
2. Access variables
3. Control flow of execution
4. Use data structures
5. Execute statements

Hardware Perspective

1. Allocate and initialize memory
2. Access memory
3. Change program counter
4. Perform address computations
5. Transform data

Potential Attack Vectors

Hardware Prospective

```

1111 1110 0000 0001 0000 0001 0001 0011
0000 0000 0001 0001 0010 1110 0010 0011
0000 0000 1000 0001 0010 1100 0010 0011
0000 0010 0000 0001 0000 0100 0001 0011
0000 0000 1000 0000 0000 0111 1001 0011
1111 1110 1111 0100 0010 0110 0010 0011
1111 1110 1100 0100 0010 0111 1000 0011
0000 0000 0000 0111 1000 0101 0001 0011
0000 0000 0000 0000 0000 0000 1001 0111
1111 0110 0100 0000 1000 0000 1110 0111
    
```

Real Machine Code

```

fe010113 // 0000019c addi sp,sp,-32
00112e23 // 000001a0 sw ra,28(sp)
00812c23 // 000001a4 sw s0,24(sp)
02010413 // 000001a8 addi s0,sp,32
00800793 // 000001ac addi a5,zero,8
fef42623 // 000001b0 sw a5,-20(s0)
fec42783 // 000001b4 lw a5,-20(s0)
00078513 // 000001b8 addi a0,a5,0
00000097 // 000001bc auipc ra,0x0
f64080e7 // 000001c0 jalr ra,-156(ra)
    
```

Addresses

Memory Allocation

- There are two types of memory allocation
 - **Static memory allocation:** Memory is allocated at the start of the program, and freed when program exits
 - Done by the compiler automatically (implicitly)
 - Global variables or objects
 - Alive throughout program execution
 - Can be access anywhere in the program
 - Local variables (inside a function)
 - Memory is allocated when the function starts and freed when the routine returns
 - A local variable cannot be accessed from another function

Memory Allocation

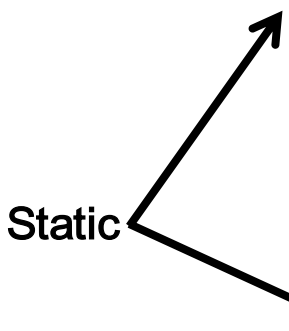
- There are two types of memory allocation
 - Static: Memory is allocated at the start of the program, and freed when program exits

Static

```
#include <stdio.h>
int number1, number2, number3;
int array[4] = {3, 5, 6, 8};

/* declare and define */
int function (int x){
    int number 4, number5;
    ...
}

void main (void){
    ...
}
```



Memory Allocation

- There are two types of memory allocation
 - Dynamic memory allocation deals with objects whose size can be adjusted depending on needs
 - Dynamic – Done explicitly by programmer
 - Programmer explicitly requests the system to allocate memory and return starting address of memory allocated
 - This address can be used by the programmer to access the allocated memory
 - When done using memory, it must be explicitly freed

Memory Allocation

- There are two types of memory allocation
 - Dynamic memory allocation deals with objects whose size can be adjusted depending on needs
 - Dynamic memory allocation in C:
 - `calloc()`
 - `malloc()`
 - `realloc()`
 - Deallocated using the `free()` function

Memory Allocation

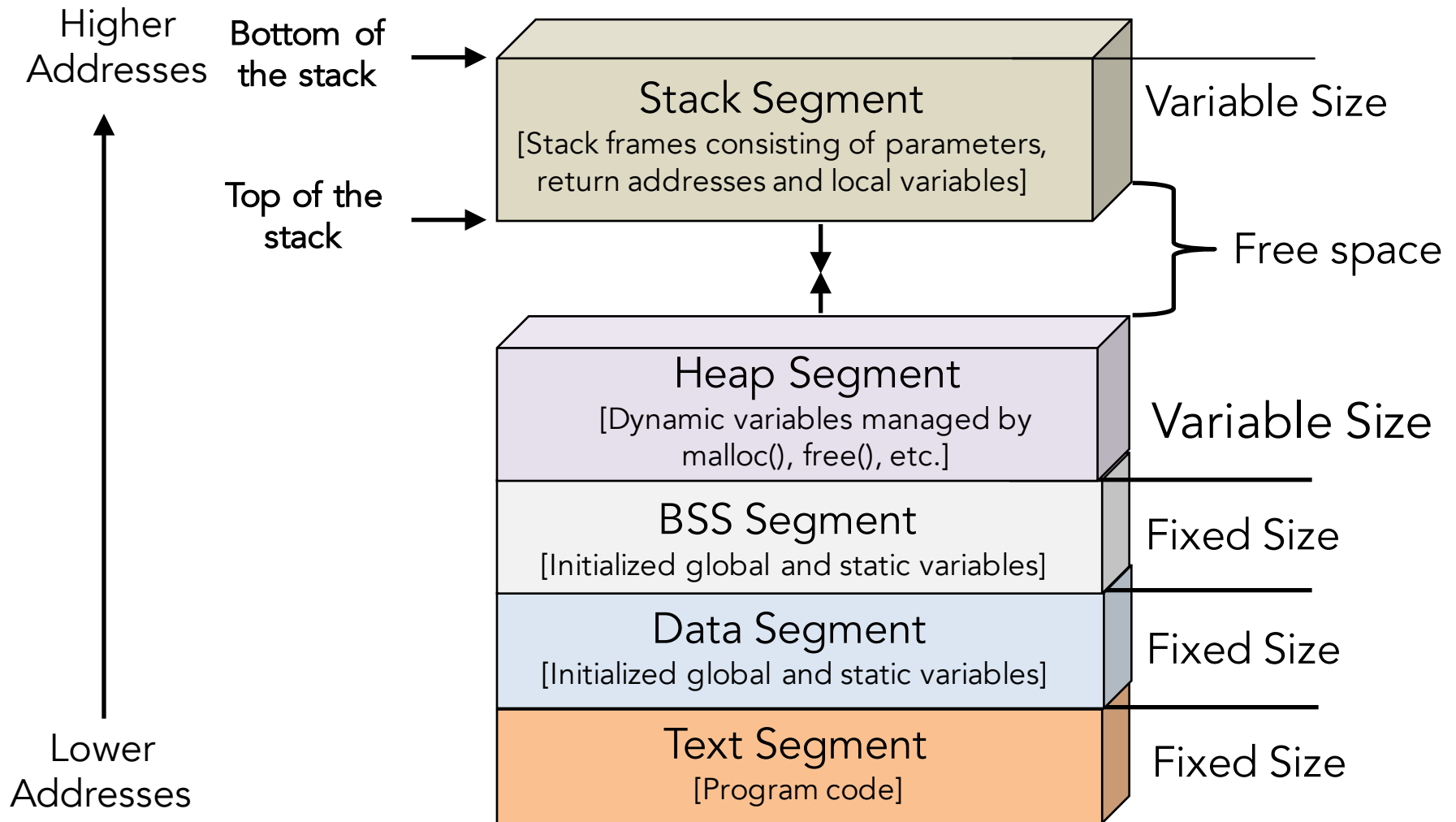
- There are two types of memory allocation
 - Dynamic memory allocation deals with objects whose size can be adjusted depending on needs

```
#include <stdio.h>
void main (void){
    int i = 0; int nelements_wanted = 8;
    int *i_ptr;
    i_ptr = (int*)malloc(sizeof(int)*nelements_wanted);
    if (i_ptr != NULL) {
        i_ptr[i] = 5;
    }
    else {
        /* Couldn't get the memory - recover */
    }
}
```

Memory Allocation

- There are two types of memory allocation
 - Dynamic memory allocation deals with objects whose size can be adjusted depending on needs
 - Remember in C if you allocate some piece of memory, you are responsible as the programmer to free it
 - `x = malloc(n * sizeof(int));`
 - `/* manipulate x */`
 - `free(x);`

Program memory management



Program memory management

- BSS: Block Started by Symbol
 - Developed in the mid-1950s for the IBM 704
 - BSS keyword was later incorporated into FAP (FORTRAN Assembly Program)
 - Used for a part of the data segment containing statically-allocated variables Zero-valued bits when execution begins

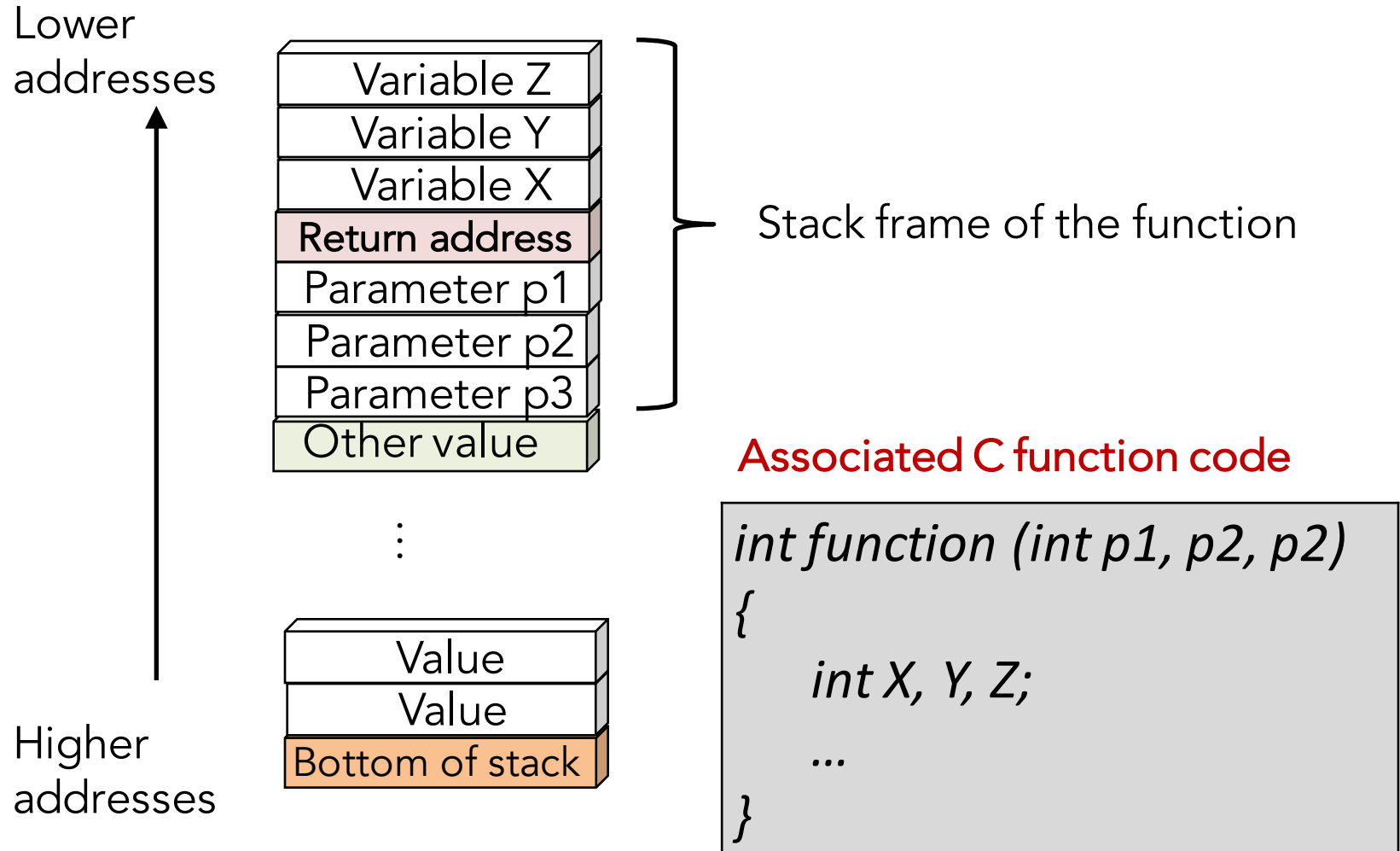
Stack Structure

- Stack and heap are two memory sections in the user mode space
- The stack handles local variables for functions, whose size can be determined at call time
- Some of information saved at function call and restored at function return:
 - Values of callee arguments
 - Register values:
 - Return address (value of PC)
 - Frame pointer (value of FP)

Stack Structure

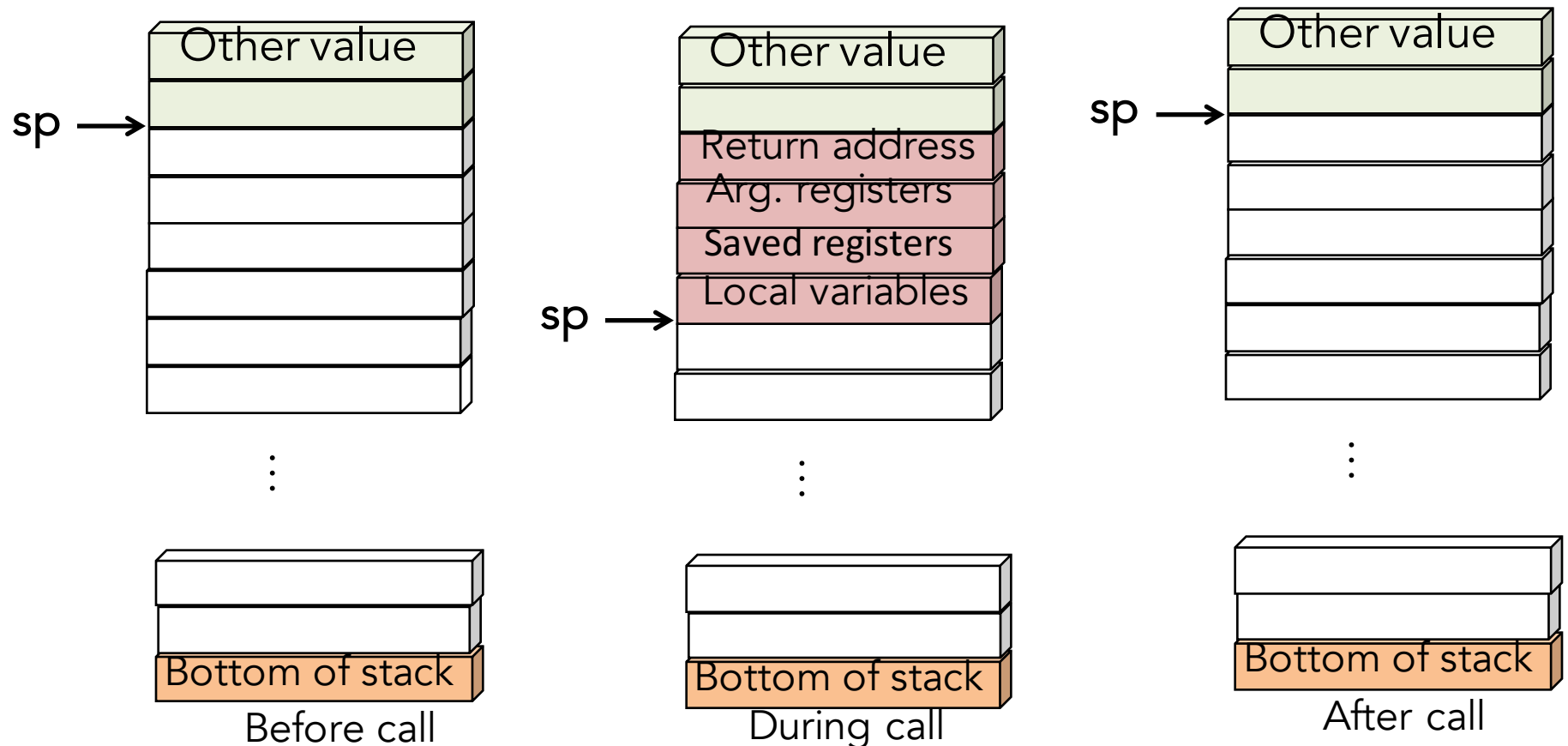
- Stack will be allocated automatically for function call
- It grows downward to the lower address
- It is Last-in First-out (LIFO) mechanism (tally with the assembly language's push and pop instructions)
- Even if the stack grows from higher to lower addresses, the local variables on the stack grow from lower to higher addresses

Stack Structure



Stack Structure

- Procedure frame or activation record



Heap Structure

- The heap is allocated by demand or request using C memory management functions such as malloc(), memset(), realloc() etc.
- It allows data (especially arrays) to take on variable sizes
- It allows locally created variables to live past end of routine
- This is what permits many structures used in Data Structures and Algorithms

Heap Structure

- It is dynamic allocation, grows upward to the higher memory address
- It is possible to allocate memory and “lose” the pointer to that region without freeing it
 - This is called a memory leak
 - A memory leak can cause the heap to become full
- In a multi-threaded environment each thread will have its own completely independent stack but they will share the heap as needed

Next Class

- Application Level Attacks