ENGR-304-L

Software Lab 03

Agenda

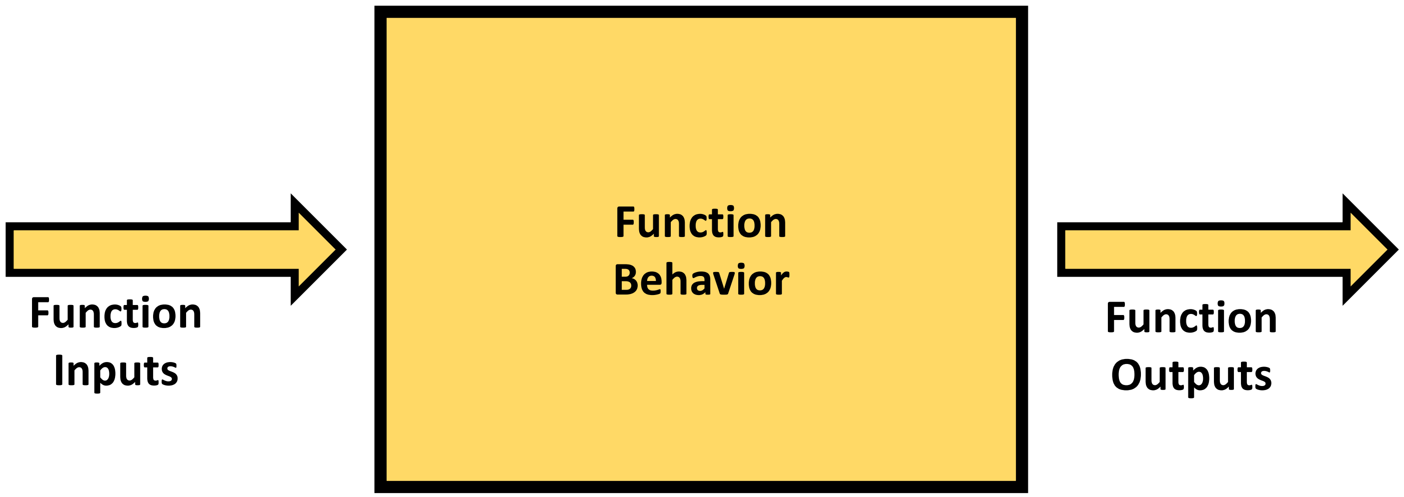
1. Attendance
2. Recap
3. Functions & Recursion
4. Functions in Assembly
5. The Stack
6. Getting Started

Recap

- Assembly programming language is closely related to machine code
- NIOS CPU executes machine code while running program
- Program results can be found in registers or memory
- Compiler optimization can be a useful tool

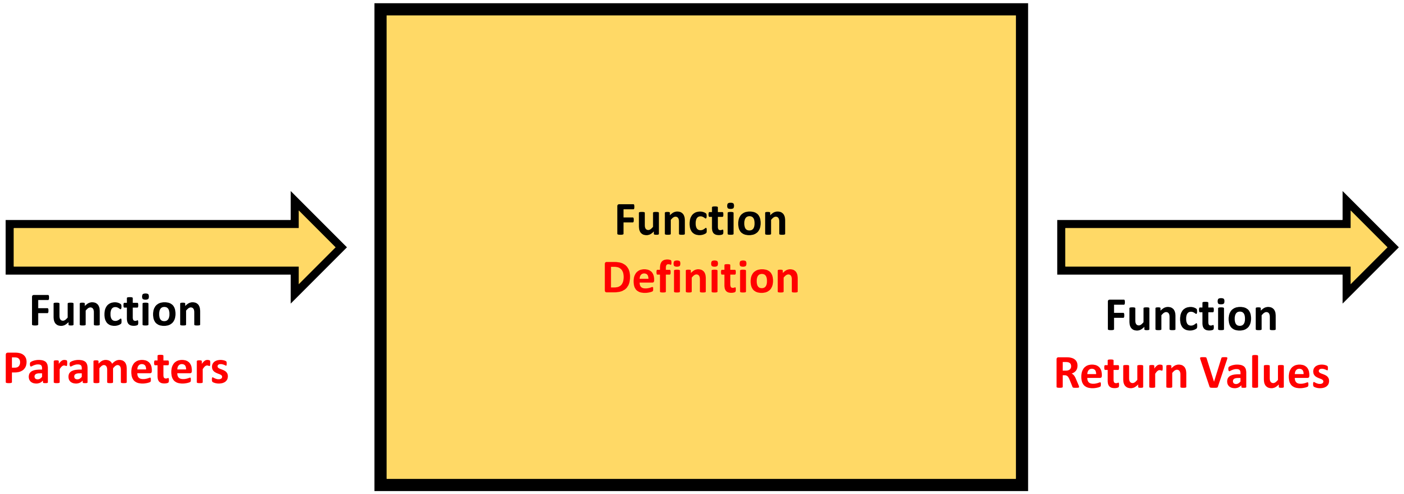
Functions

- Functions defined by behavior, inputs, & outputs



Functions

- Functions defined by behavior, inputs, & outputs
- Often referred to as definition, parameters, return values



Recursion

- Recursive problems can be divided into sub-problems
- The recursive solution is identical for the original problem and the sub-problems
- There must be at least 1 base-case defining a “smallest” sub-problem with a non-recursive solution

- Ex: Fibonacci

Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)   // n > 1

Fibonacci(n) = n   // 0 <= n <= 1

Recursion

Problem is solved by sub-problems

Fib(n) = Fib(n-1) + Fib(n-2)   // n > 1

Fib(n) = n   // 0 <= n <= 1

Recursion

Fib(n) = Fib(n-1) + Fib(n-2) // n > 1

Fib(n) = n // 0 <= n <= 1

Problem is solved by sub-problems

There are 2 base-cases

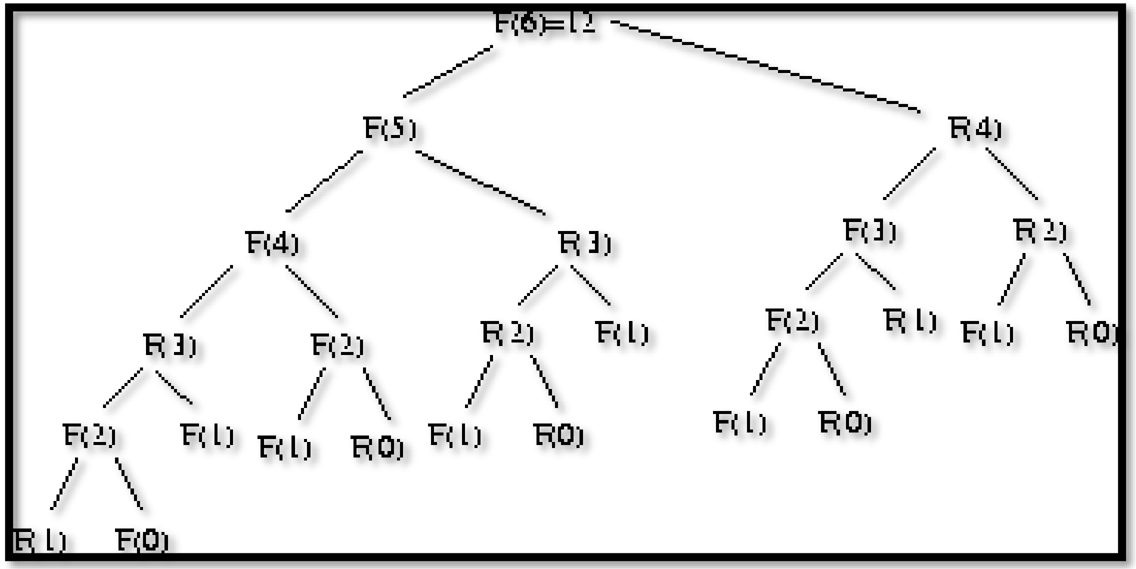
Recursion

Problem is solved by sub-problems

There are 2 base-cases

Fib(n) = Fib(n-1) + Fib(n-2)   // n > 1

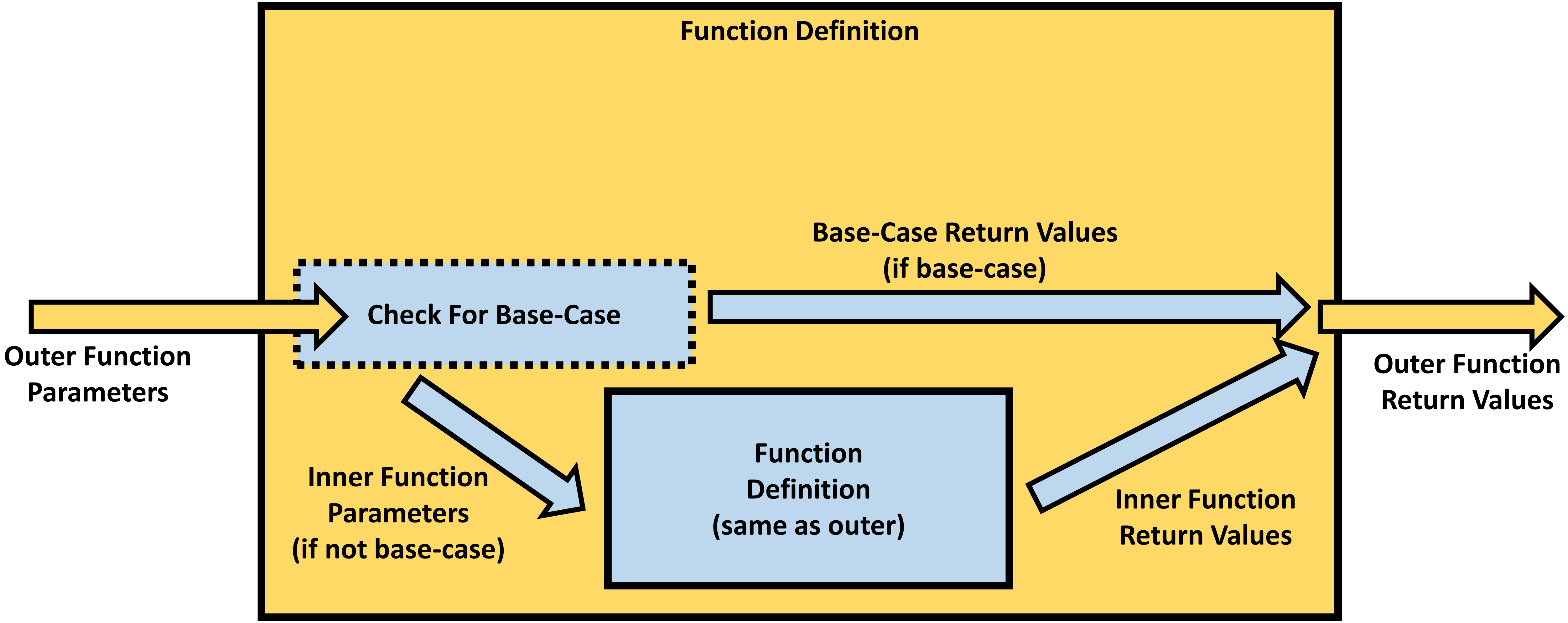
Fib(n) = n   // 0 <= n <= 1



https://www8.cs.umu.se/kurser/TDBA77/VT06/algorithms/BOOK/BOOK25/IMG313.GIF

Recursive Functions

- Recursive functions call themselves as sub-problems



Recursive Functions

- C Function Declarations

```c
// Function to determine the fibonacci value for the received "n"
unsigned int fibonacci( unsigned int n );
```

Parameter list

Return type

Function name

First parameter type

First parameter name

Recursive Functions

- C Function Definitions

```c
// Function to determine the fibonacci value for the received "n"
unsigned int fibonacci( unsigned int n )
{
    unsigned int value = 0;
    
    // for n==0, the fibonacci value is always 0
    // for n==1, the fibonacci value is always 1
    if ( n <= 1 )
    {
        value = n;
    }
    // for n>1, the fibonacci value is the sum of the previous 2 fibonacci values
    else if ( n > 1 )
    {
        value = ( fibonacci(n - 1) + fibonacci(n - 2) );
    }
    
    return value;
}
```

Local variables

Recursive base-case check

Return statement

Recursive function calls

Assembly Functions

- Global label for function name
- “call”
- “ret”

```
/* The ".text" assemlber directive indicates the beginning of the code section of the program */
.text

/*TEMPLATE: replace FunctionTemplate with the name of the function below */
/* The ".global FunctionTemplate" assembler directive exports the */
/* "FunctionTemplate" label as an external symbol, so that C-Code can call it */
.global FunctionTemplate
FunctionTemplate: /* Starting location of the function */

/************************FunctionTemplate********************************/
/*TEMPLATE: use comments to outline the function psuedo-code here */
/* Pseudo-code for this function is as follows 
 *    
 */
/*TEMPLATE: rename the function labels as desired below */
/*TEMPLATE: write the assembly function below */
FUNCTION_INIT:
    /*TEMPLATE: include any register pushes here */
    /*TEMPLATE: include any initialization steps */

FUNCTION_EXEC:
    /*TEMPLATE: include main execution of the function here */

FUNCTION_END:
    /*TEMPLATE: include any register pops here */

    /* Note that there should only be one "ret" instruction in your function */
    ret  /* instruction to return to the calling function */

.end  /* the assembler throws away all text after this line */
```

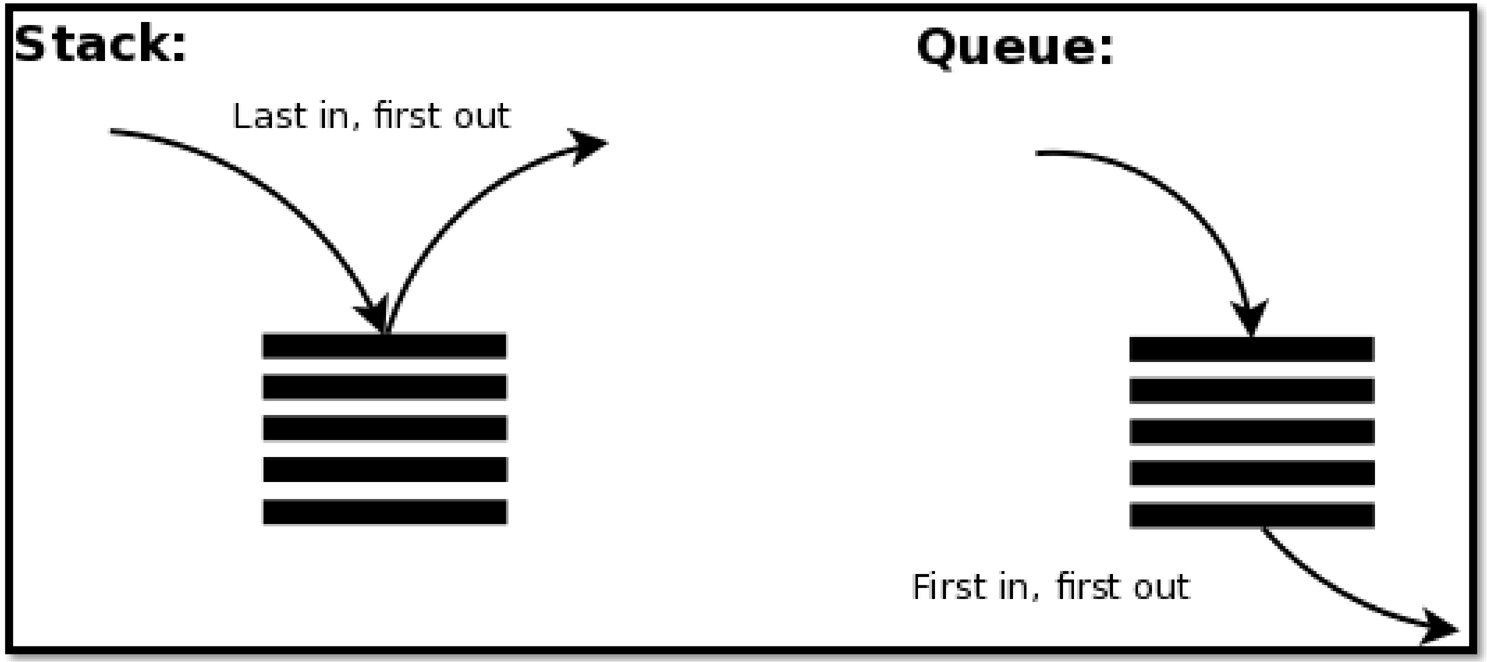
NIOS Function Registers

- Standard register usage "agreement"
- R2, R3 – return value
- R4, R5, R6, R7 – parameters
- RA (R31) – return address
- Callee-saved vs Caller-saved

Register	Name	Used by Compiler	Callee Saved (1)	Normal Usage
r0	zero	x		0x00000000
r1	at			Assembler Temporary
r2		x		Return Value (Least-significant 32 bits)
r3		x		Return Value (Most-significant 32 bits)
r4		x		Register Arguments (First 32 bits)
r5		x		Register Arguments (Second 32 bits)
r6		x		Register Arguments (Third 32 bits)
r7		x		Register Arguments (Fourth 32 bits)
r8		x		Caller-Saved General-Purpose Registers
r9		x		Caller-Saved General-Purpose Registers
r10		x		Caller-Saved General-Purpose Registers
r11		x		Caller-Saved General-Purpose Registers
r12		x		Caller-Saved General-Purpose Registers
r13		x		Caller-Saved General-Purpose Registers
r14		x		Caller-Saved General-Purpose Registers
r15		x		Caller-Saved General-Purpose Registers
r16		x	x	Callee-Saved General-Purpose Registers
r17		x	x	Callee-Saved General-Purpose Registers
r18		x	x	Callee-Saved General-Purpose Registers
r19		x	x	Callee-Saved General-Purpose Registers
r20		x	x	Callee-Saved General-Purpose Registers
r21		x	x	Callee-Saved General-Purpose Registers
r22		x	x	Callee-Saved General-Purpose Registers
r23		x	x	Callee-Saved General-Purpose Registers
r24	et			Exception Temporary
r25	bt			Break Temporary
r26	gp	x		Global Pointer
r27	sp	x		Stack Pointer
r28	fp	x		Frame Pointer (2)
r29	ea			Exception Return Address
r30	ba			Break Return Address
r31	ra	x		Return Address

The Stack

- Stacks & Queues
- Ex: buffet dishes & amusement park lines

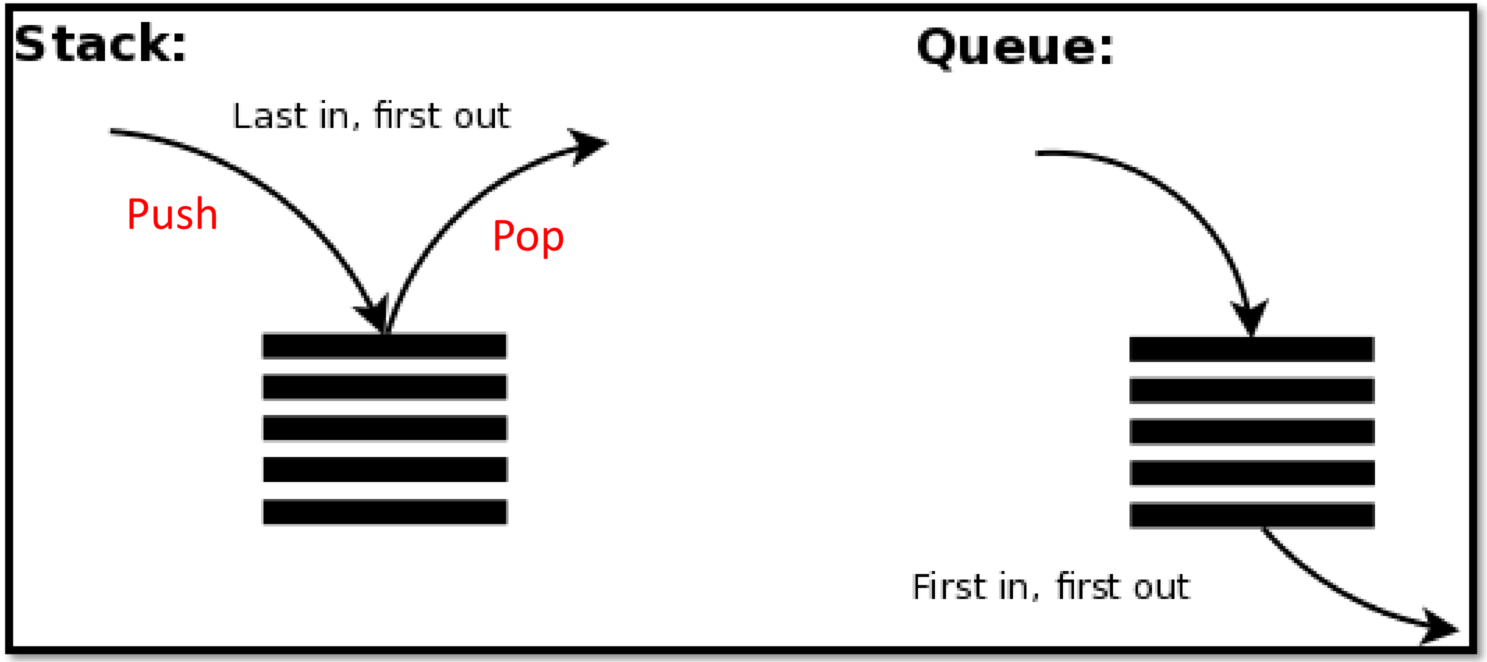


*https://cdn-images-1.medium.com/max/523/1*asgh4mx9ez5Ji8PmBayS8g.png*

The Stack

- Stacks & Queues
- Ex: buffet dishes & amusement park lines

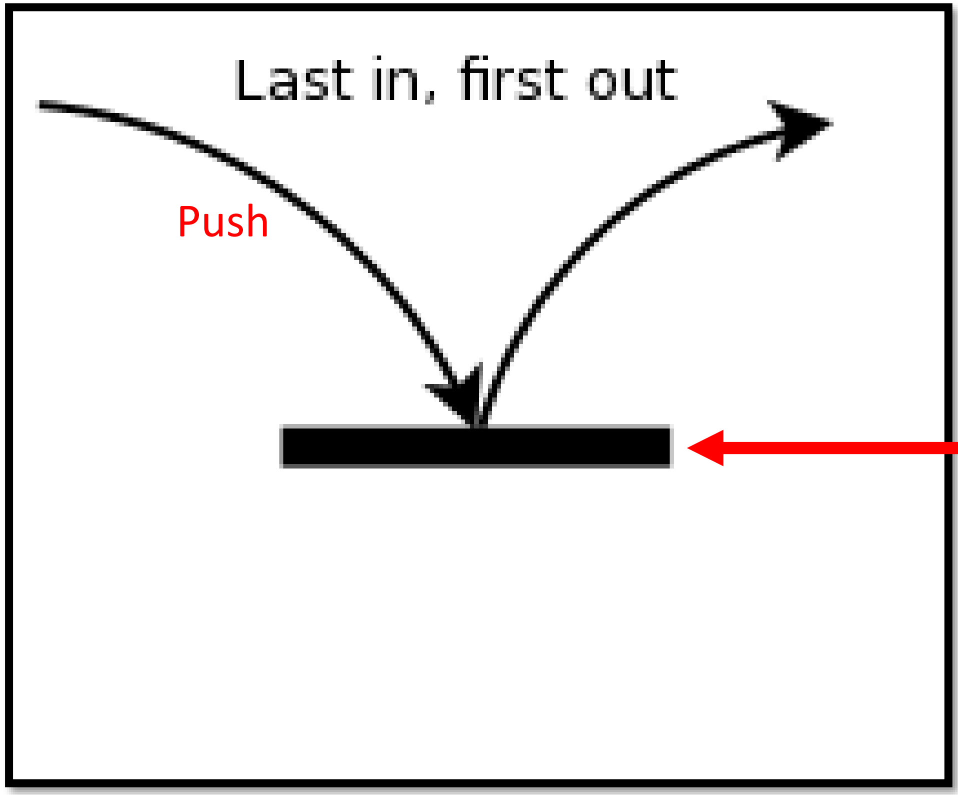
- Stack Push & Pop



*https://cdn-images-1.medium.com/max/523/1*asgh4mx9ez5Ji8PmBayS8g.png*

The Stack

- Stack is useful for nested function calls (ex: recursion)

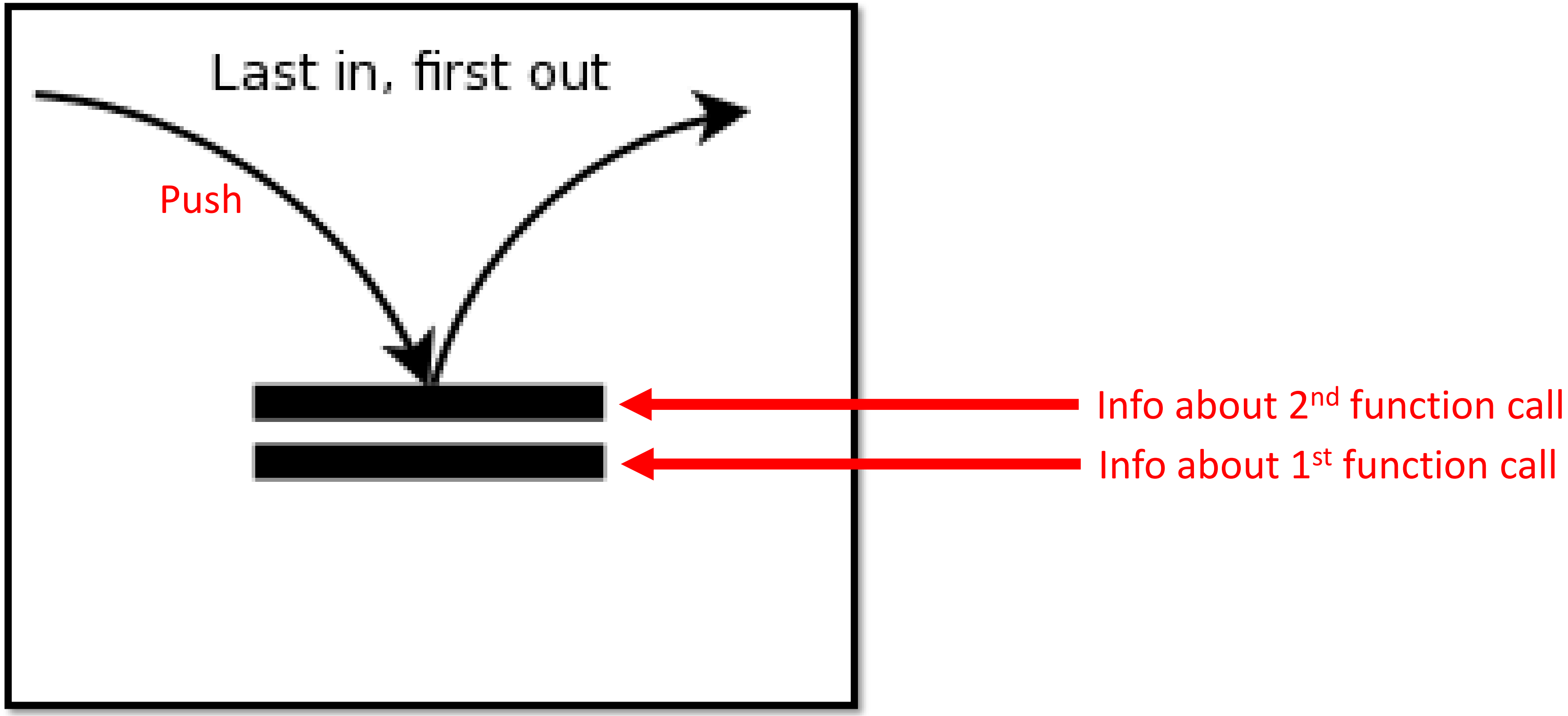


Info about 1st function call

https://cdn-images-1.medium.com/max/523/1*asgh4mx9ez5Ji8PmBayS8g.png

The Stack

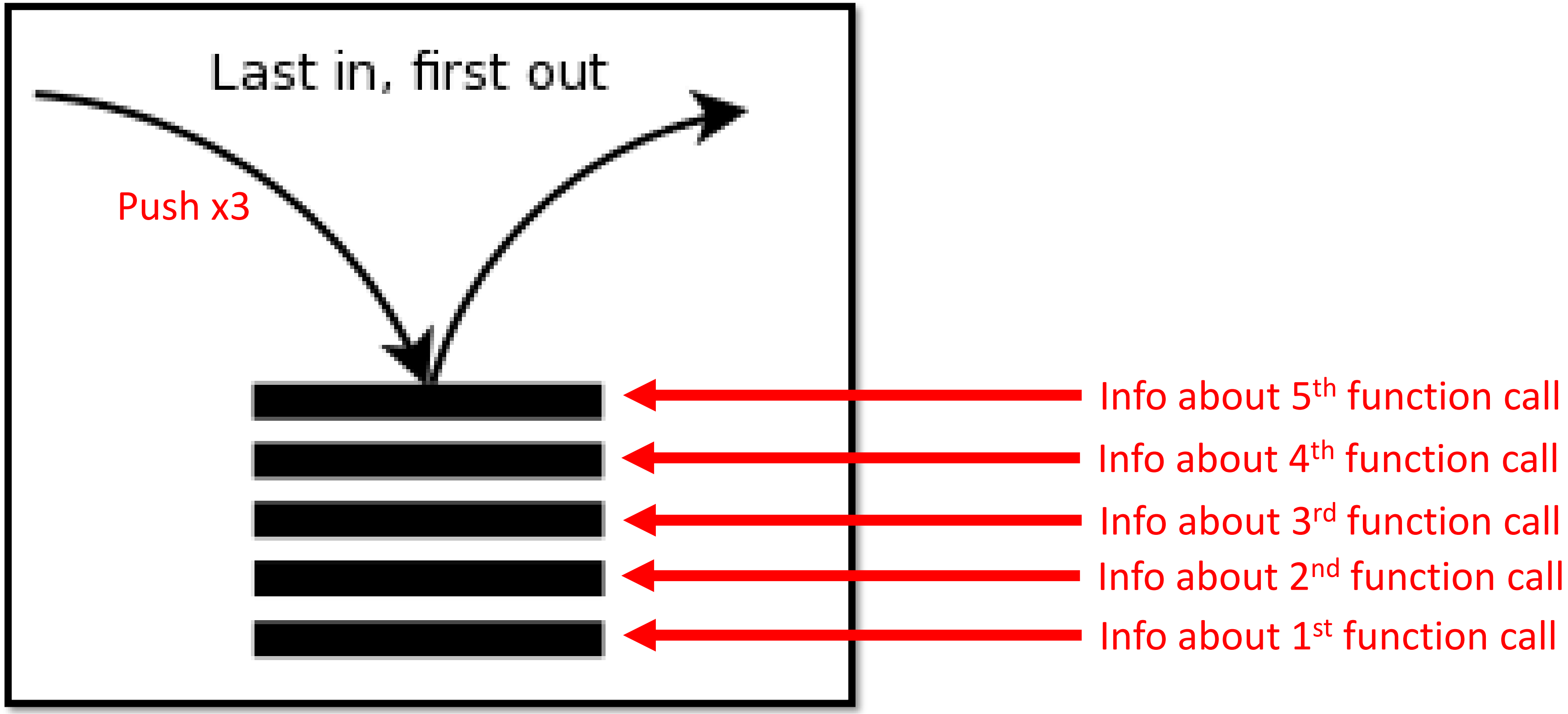
- Stack is useful for nested function calls (ex: recursion)



*https://cdn-images-1.medium.com/max/523/1*asgh4mx9ez5Ji8PmBayS8g.png*

The Stack

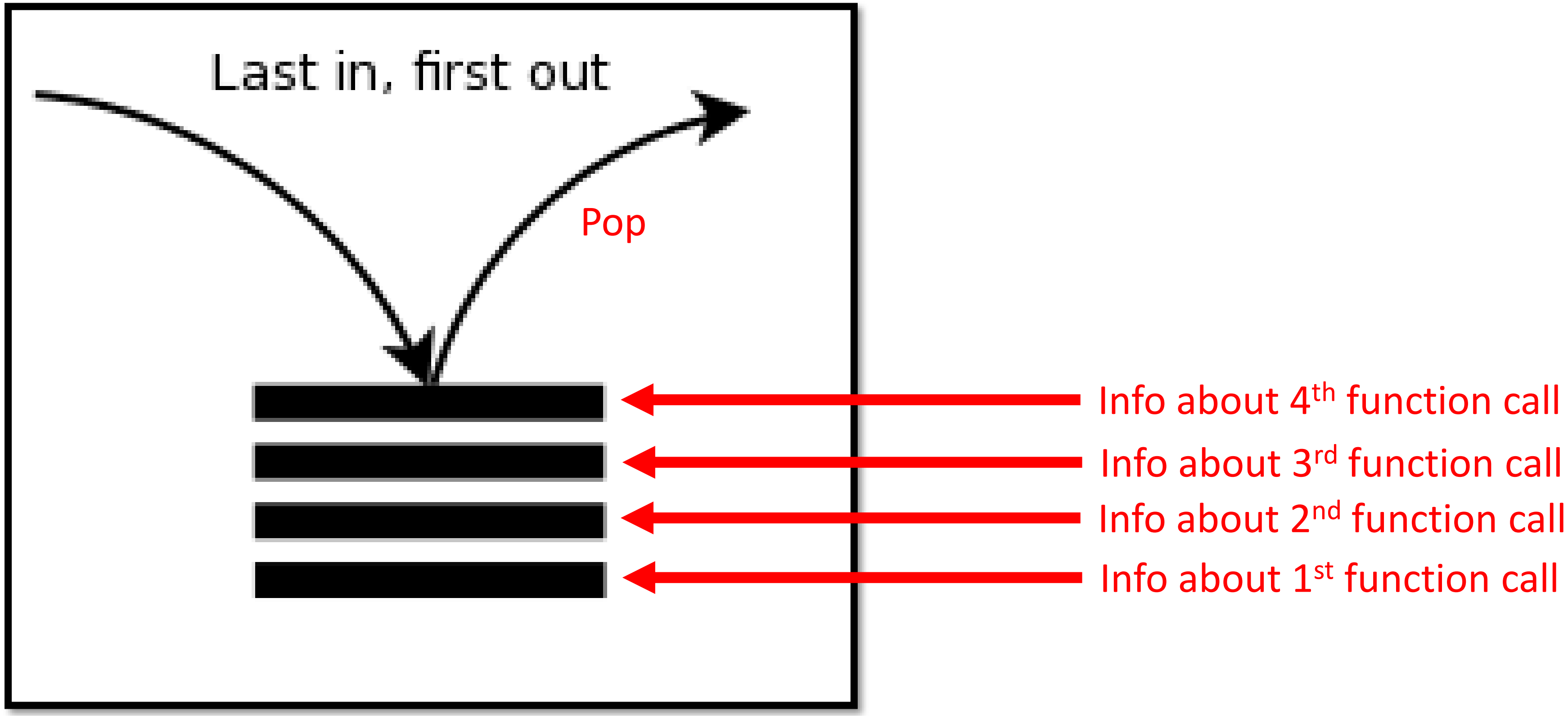
- Stack is useful for nested function calls (ex: recursion)



*https://cdn-images-1.medium.com/max/523/1*asgh4mx9ez5Ji8PmBayS8g.png*

The Stack

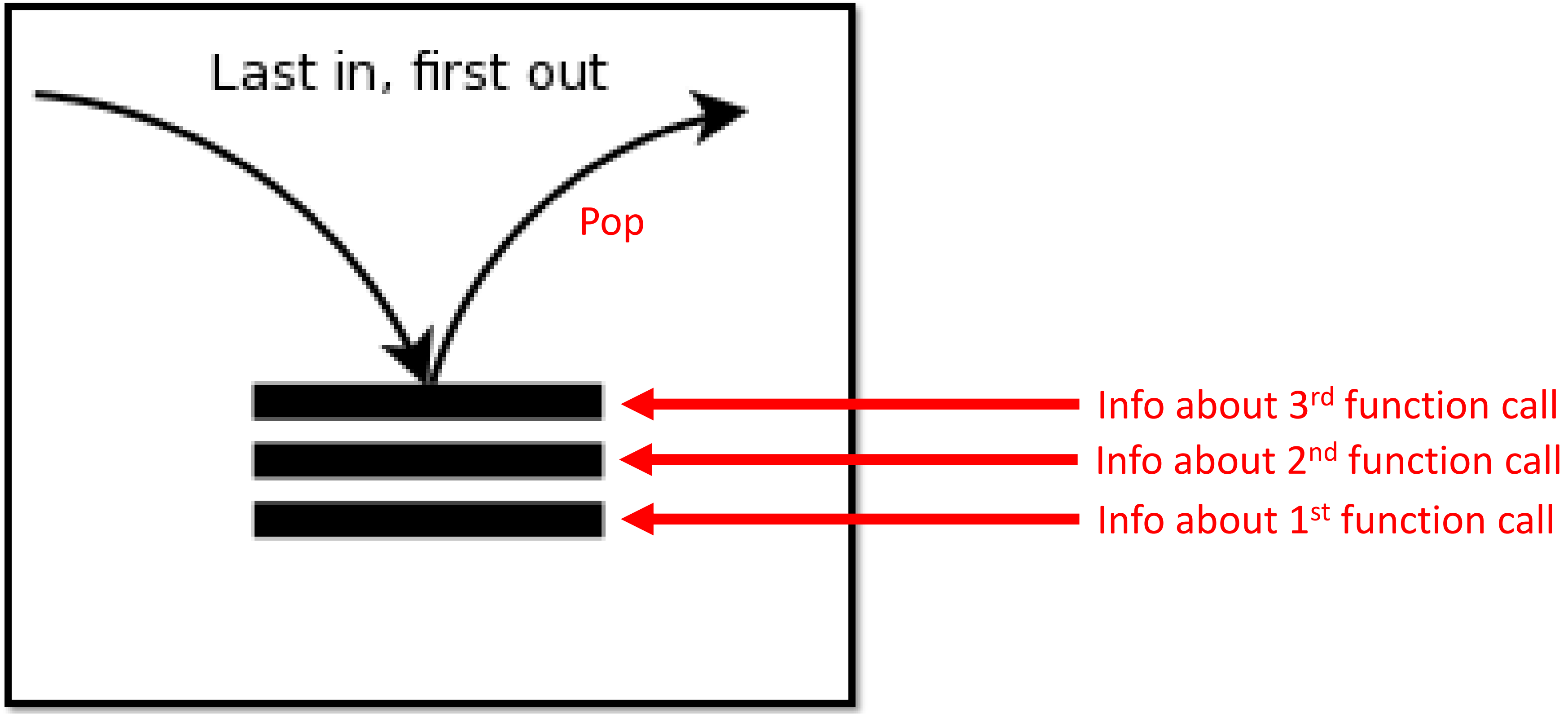
- Stack is useful for nested function calls (ex: recursion)



https://cdn-images-1.medium.com/max/523/1*asgh4mx9ez5Ji8PmBayS8g.png

The Stack

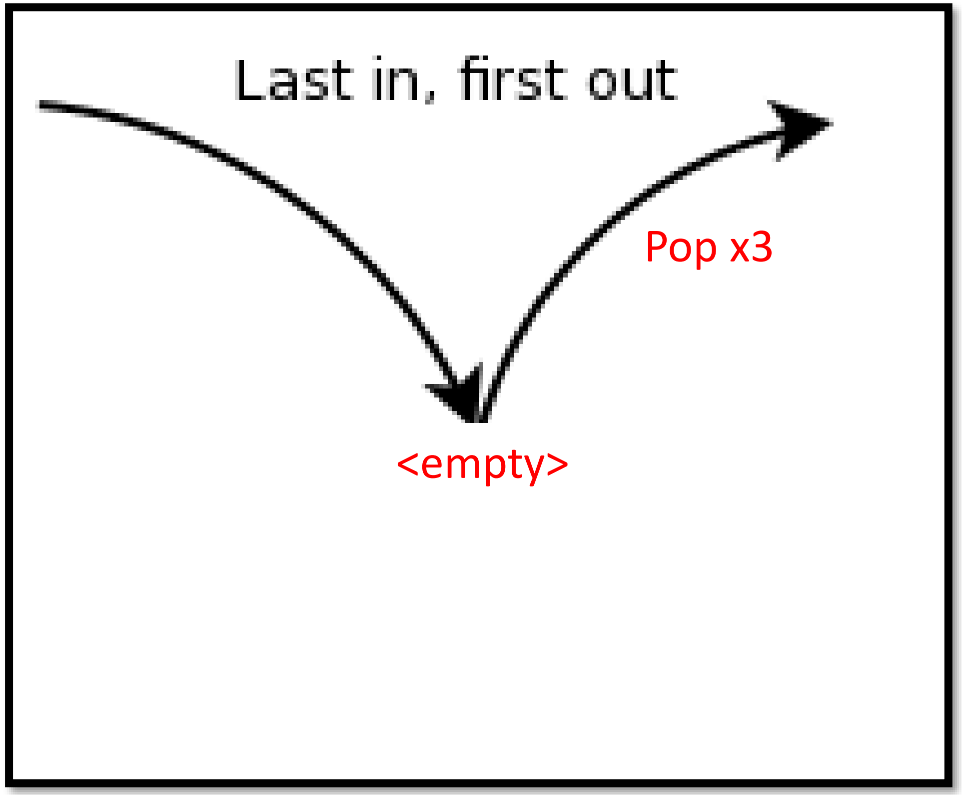
- Stack is useful for nested function calls (ex: recursion)



*https://cdn-images-1.medium.com/max/523/1*asgh4mx9ez5Ji8PmBayS8g.png*

The Stack

- Stack is useful for nested function calls (ex: recursion)



*https://cdn-images-1.medium.com/max/523/1*asgh4mx9ez5Ji8PmBayS8g.png*

The Stack

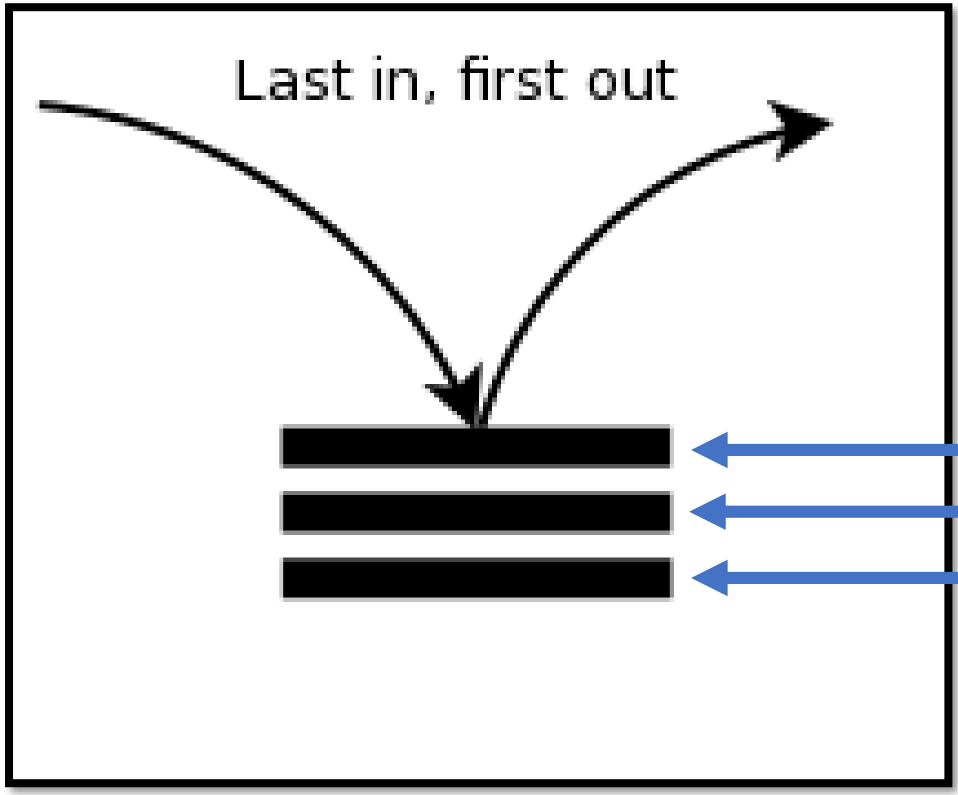
- “Info about Xth function call” → preserve state

```
// Function to determine the fibonacci value for the received "n"
unsigned int fibonacci( unsigned int n )
{
    unsigned int value = 0;
    
    // for n==0, the fibonacci value is always 0
    // for n==1, the fibonacci value is always 1
    if ( n <= 1 )
    {
        value = n;
    }
    // for n>1, the fibonacci value is the sum of the previous 2 fibonacci values
    else if ( n > 1 )
    {
        value = ( fibonacci(n - 1) + fibonacci(n - 2) );
    }
    
    return value;
}
```

Even though recursive calls use the same n parameter and return statement, the former call should not have its n changed by the sub-call. We need to preserve state.

The Stack

- “Info about Xth function call” → preserve state



State of registers as used by 3rd function call

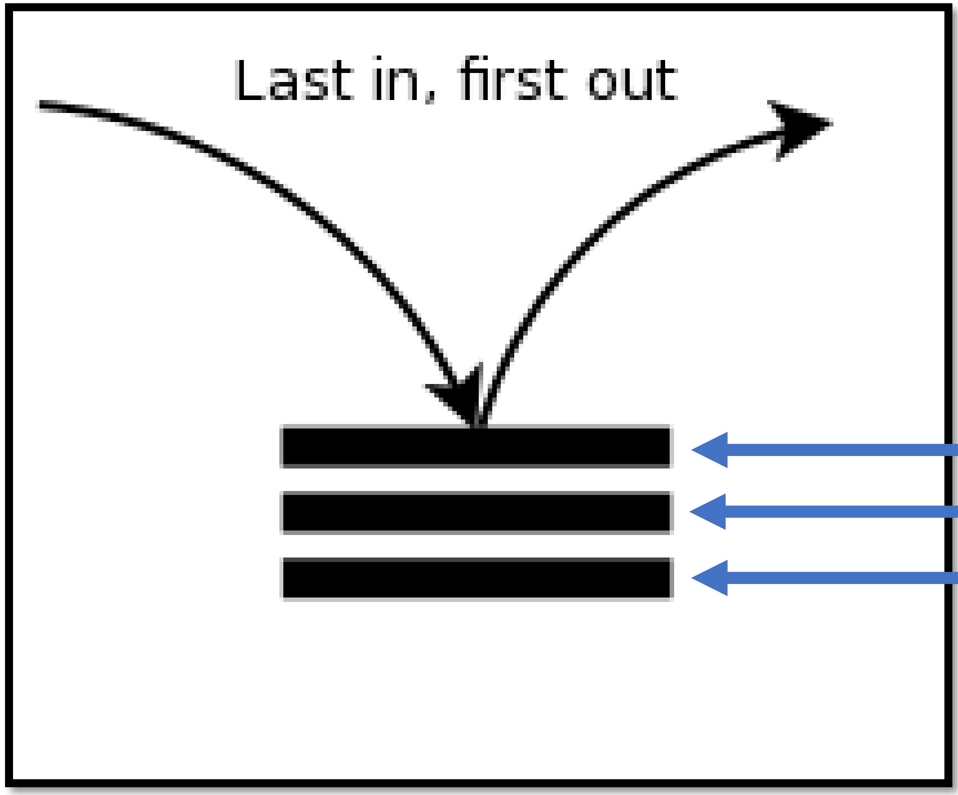
State of registers as used by 2nd function call

State of registers as used by 1st function call

*https://cdn-images-1.medium.com/max/523/1*asgh4mx9ez5Ji8PmBayS8g.png*

The Stack

- "Info about Xth function call" → preserve state



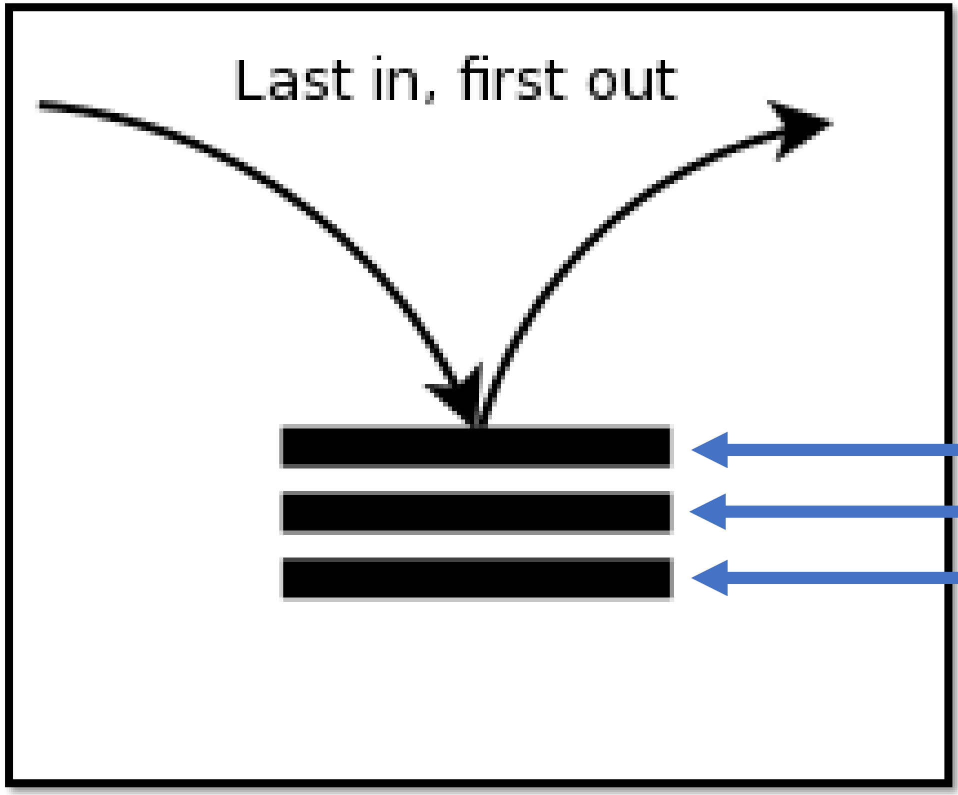
Since the stack is in memory,
Push is done by "stw" = add to the top of the stack
Pop is done by "ldw" = remove from the top of the stack

State of registers as used by 3rd function call
State of registers as used by 2nd function call
State of registers as used by 1st function call

https://cdn-images-1.medium.com/max/523/1*asgh4mx9ez5Ji8PmBayS8g.png

The Stack

- “Info about Xth function call” → preserve state



Since the stack is in memory,
Push is done by “stw” = add to the top of the stack
Pop is done by “ldw” = remove from the top of the stack

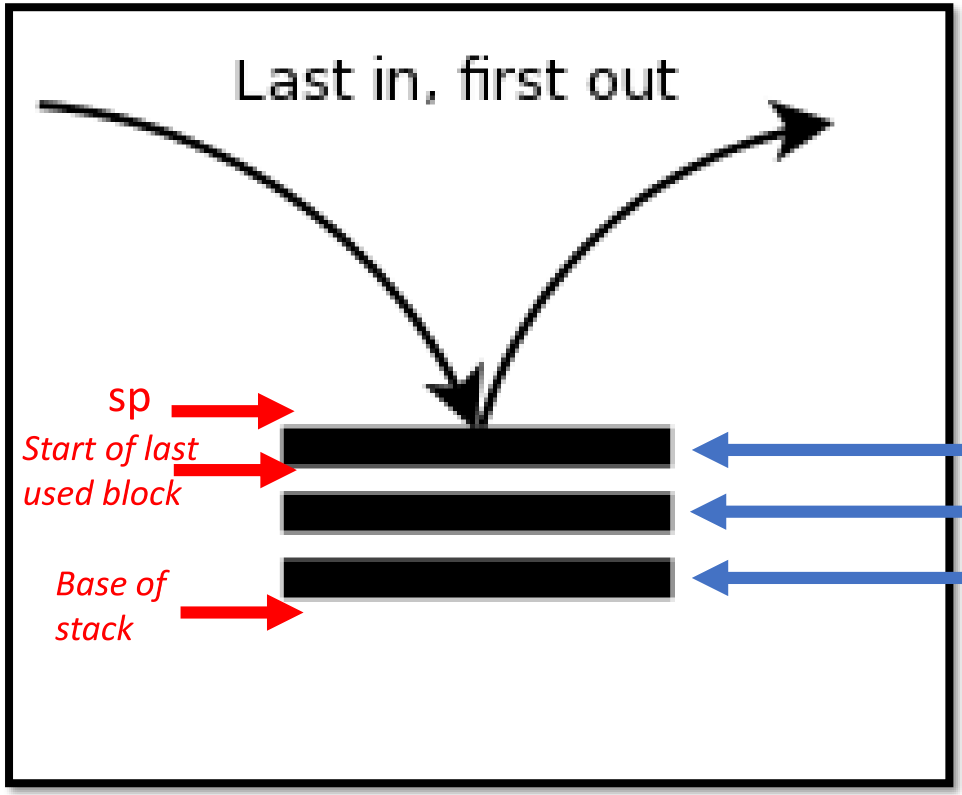
R# as used by 2^{nd} unction call
R# as used by 1^{st} function call
R# as used by 1^{st} function call

The registers all share the same stack, may be several for each function call *(arbitrarily showing 2 for 1^{st} and 1 for 2^{nd})*

*https://cdn-images-1.medium.com/max/523/1*asgh4mx9ez5Ji8PmBayS8g.png*

The Stack

- “Info about Xth function call” → preserve state



Since the stack is in memory,
Push is done by “stw” = add to the top of the stack
Pop is done by “ldw” = remove from the top of the stack

R27 (sp) is the stack pointer, it points to the top of the stack
(1 byte beyond the last used memory address)

State of a register on stack
State of a register on stack
State of a register on stack

*https://cdn-images-1.medium.com/max/523/1*asgh4mx9ez5Ji8PmBayS8g.png*

The Stack

```
                                fibonacci:
0x0000432c    defffd04          addi    sp, sp, -0xc
0x00004330    dfc00215          stw     ra, 8(sp)
0x00004334    dc400115          stw     r17, 4(sp)
0x00004338    dc000015          stw     r16, 0(sp)
0x0000433c    [redacted]
0x00004340    [redacted]
0x00004344    [redacted]
0x00004348    [redacted]
0x0000434c    [redacted]
0x00004350    [redacted]
0x00004354    000432c0          call    0x000010cb (0x0000432c: fibonacci)
0x00004358    [redacted]
0x0000435c    [redacted]
0x00004360    000432c0          call    0x000010cb (0x0000432c: fibonacci)
0x00004364    [redacted]
0x00004368    dfc00217          ldw     ra, 8(sp)
0x0000436c    dc400117          ldw     r17, 4(sp)
0x00004370    dc000017          ldw     r16, 0(sp)
0x00004374    dec00304          addi    sp, sp, 0xc
0x00004378    f800283a          ret
```

Pushing onto stack, note sp decremented because stack memory goes from end to start (stack & heap work towards each other)

These calls continue to use and change the stack, but each pops it back to the state it was in originally before returning so the caller doesn't notice the changes (state preserved)

Popping from stack, note sp has same value in the end as it did originally

Note: comments removed for slides, always use good comments in assembly, portions of function are redacted

Lab SW03 Tips

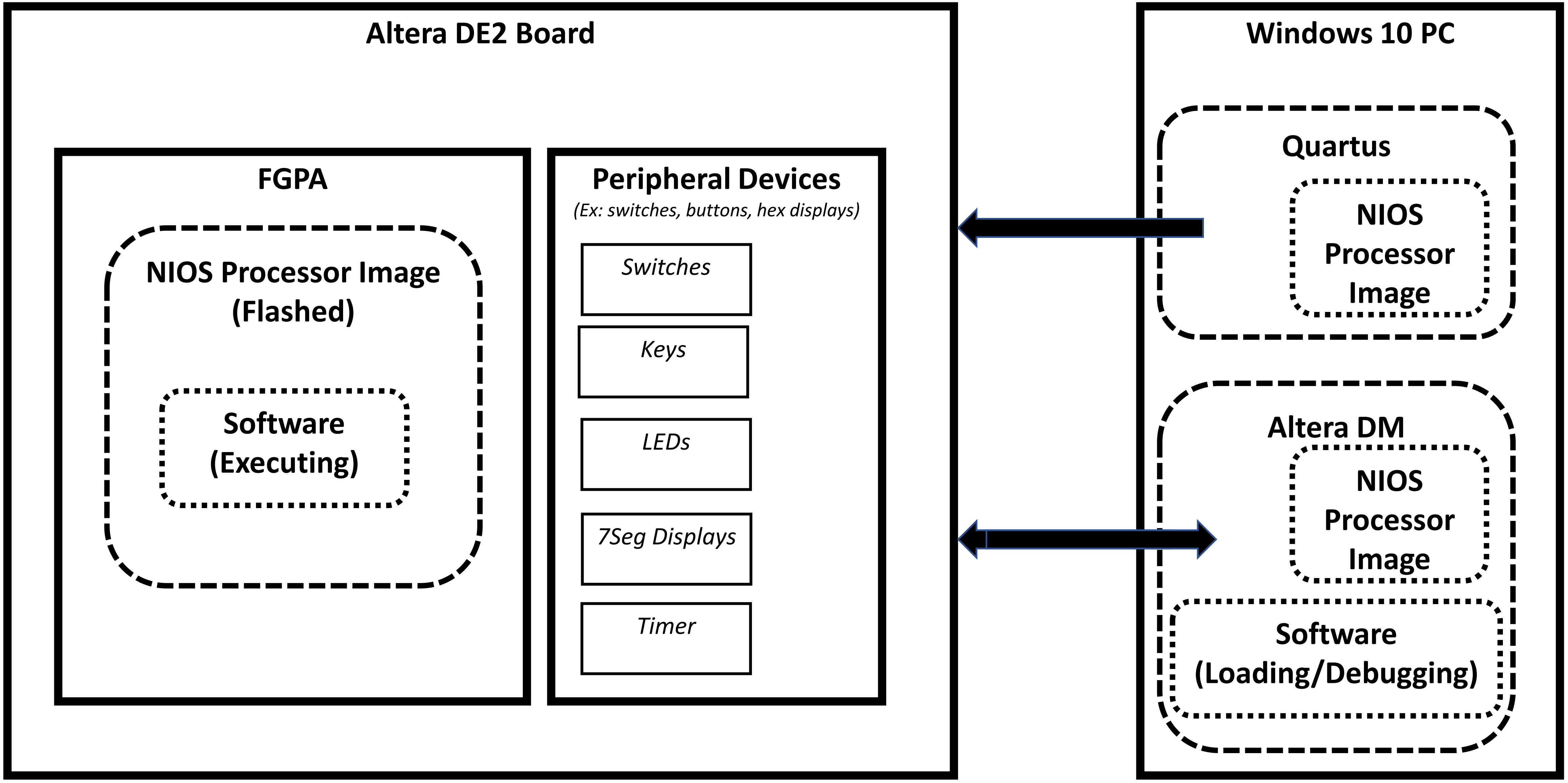
- Use only the n=0 recursive base case for Factorial program
- Use optimization level –O1
- Refer to the NIOS summary reference “registers table”
- Refer to “NIOS Processor Stack” slide deck for details on assembly functions & stack usage

Getting Started

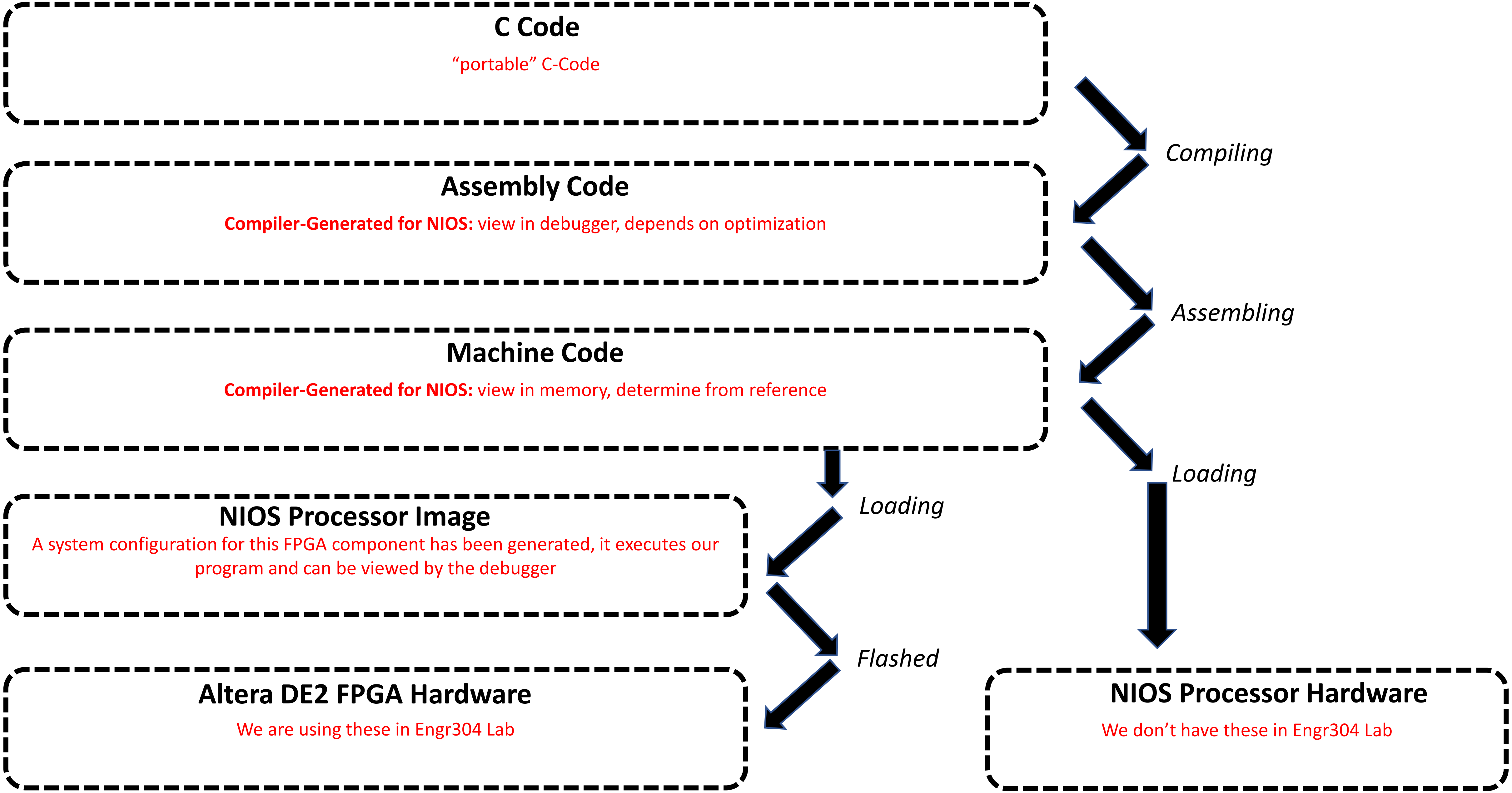
1. Setup project directory for SW03 on the **H: drive**
2. Download SW03 files from Moodle or S: drive
3. Also locate the reference files from SW01 & SW02
4. Start following directions in the lab assignment document

Reference Diagrams

- The remaining slides explain certain file types and compilation processes that will apply throughout the semester



Altera DE2 Board
FGPA
NIOS Processor Image
(Flashed)
Software
(Executing)
Peripheral Devices
(Ex: switches, buttons, hex displays)
Switches
Keys
LEDs
7Seg Displays
Timer
Windows 10 PC
Quartus
NIOS
Processor
Image
Altera DM
NIOS
Processor
Image
Software
(Loading/Debugging)



C Code
“portable” C-Code
Compiling
Assembly Code
Compiler-Generated for NIOS: view in debugger, depends on optimization
Assembling
Machine Code
Compiler-Generated for NIOS: view in memory, determine from reference
Loading
NIOS Processor Image
A system configuration for this FPGA component has been generated, it executes our program and can be viewed by the debugger
Flashed
Altera DE2 FPGA Hardware
We are using these in Engr304 Lab
Loading
NIOS Processor Hardware
We don’t have these in Engr304 Lab

C Code

C Code is fairly platform-agnostic and can be run on a variety of difference processors, each with their own assembly instruction sets. C Code is compiled into assembly code which is beneficial because the combination of C Code and a compiler has better maintainability, portability, readability, and shorter development times than raw assembly code.

Compiling

Assembly Code

Assembly code is fairly platform-specific and may be shared by families of processors. Assembly code is almost at the level of machine code, but is more readable and maintainable than raw 0's and 1's. It is assembled into machine code.

Assembling

Machine Code

Machine code is the raw 0's and 1's that can be interpreted by the processor in order to execute programs. It is rarely used or modified directly since changes to source code occur at the C Code or Assembly Code levels instead. However, it may be captured as an artifact of the build process for a particular program.

Loading

NIOS Processor Image

Machine code can be loaded onto an image of the processor for which it was assembled and executed therein. In this case, the processor image is the same as a physical processor (for the most part).

Flashed

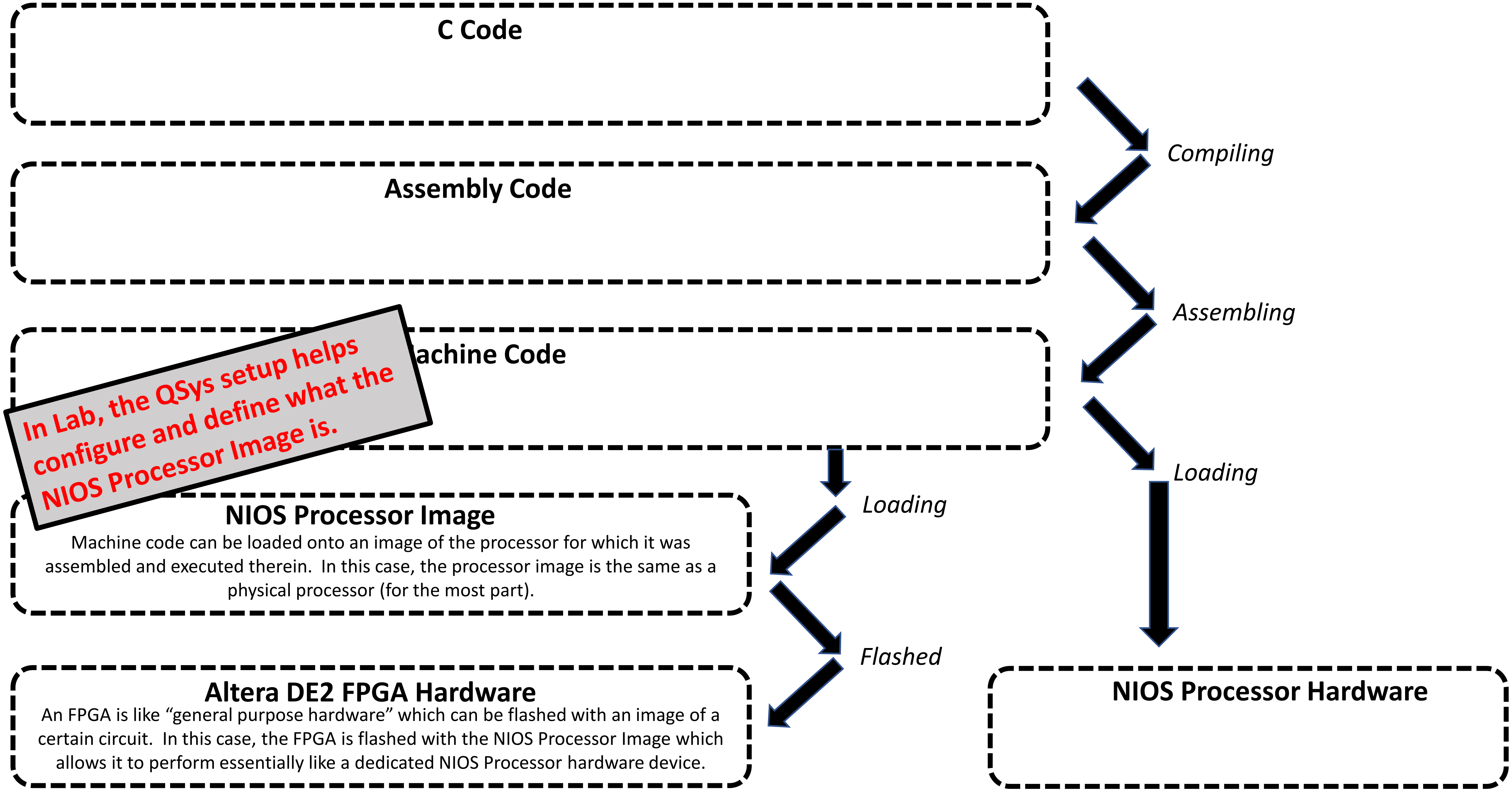
Altera DE2 FPGA Hardware

An FPGA is like "general purpose hardware" which can be flashed with an image of a certain circuit. In this case, the FPGA is flashed with the NIOS Processor Image which allows it to perform essentially like a dedicated NIOS Processor hardware device.

Loading

NIOS Processor Hardware

Machine code can be loaded onto the processor for which it was assembled and "physically" executed therein.


C Code
Compiling
Assembly Code
Assembling
achine Code
Loading
In Lab, the QSys setup helps configure and define what the NIOS Processor Image is.
Loading
NIOS Processor Image
Machine code can be loaded onto an image of the processor for which it was assembled and executed therein. In this case, the processor image is the same as a physical processor (for the most part).
Flashed
Altera DE2 FPGA Hardware
An FPGA is like “general purpose hardware” which can be flashed with an image of a certain circuit. In this case, the FPGA is flashed with the NIOS Processor Image which allows it to perform essentially like a dedicated NIOS Processor hardware device.
NIOS Processor Hardware

C Code

Compiling

Assembly Code

Assembling

Machine Code

Loading

Loading

…mage

Machi… …f the processor for which it was … processor image is the same as a … …r the most part).

In Lab, the Altera DE2 boards are FPGA's (+ some peripherals) that we load the image onto.

Flashed

Altera DE2 FPGA Hardware

An FPGA is like "general purpose hardware" which can be flashed with an image of a certain circuit. In this case, the FPGA is flashed with the NIOS Processor Image which allows it to perform essentially like a dedicated NIOS Processor hardware device.

NIOS Processor Hardware

C Code

Compiling

Assembly Code

Assembling

Machine Code

Loading

NIOS Processor Image

Machine code can be loaded onto an image of the processor for which it was assembled and executed therein. In this case, the processor image is the same as a physical processor (for the most part).

Flashed

Altera DE2 FPGA Hardware

An FPGA is like "general purpose hardware" which can be flashed with an image of a certain circuit. In this case, the FPGA is flashed with the NIOS Processor Image which allows it to perform essentially like a dedicated NIOS Processor hardware device.

Loading

NIOS Processor Hardware

In Lab, part of flashing the FPGA involves compiling a .sof file from a .bdf file. The .bdf holds the NIOS component from QSys and the .sof is specific to the DE2.

In Lab, we write assembly code for the NIOS processor (the .s file).

Code

Compiling

Assembly Code

Assembly code is fairly platform-specific and may be shared by families of processors. Assembly code is almost at the level of machine code, but is more readable and maintainable than raw 0's and 1's. It is assembled into machine code.

Assembling

Machine Code

Machine code is the raw 0's and 1's that can be interpreted by the processor in order to execute programs. It is rarely used or modified directly since changes to source code occur at the C Code or Assembly Code levels instead. However, it may be captured as an artifact of the build process for a particular program.

Loading

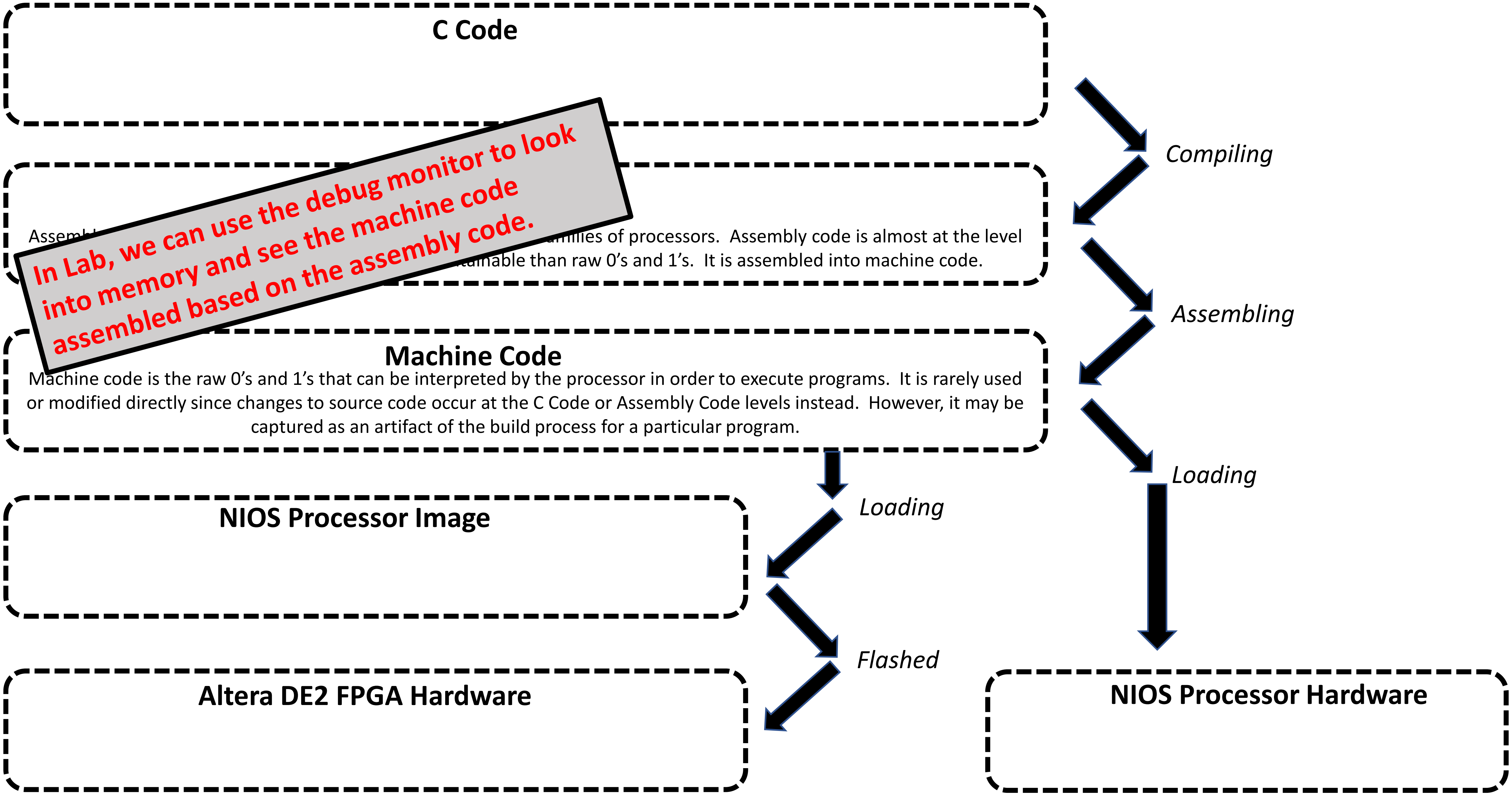
Loading

NIOS Processor Image

Flashed

Altera DE2 FPGA Hardware

NIOS Processor Hardware


C Code
Compiling
In Lab, we can use the debug monitor to look into memory and see the machine code assembled based on the assembly code.
Assemb
amilies of processors. Assembly code is almost at the level
ainable than raw 0's and 1's. It is assembled into machine code.
Assembling
Machine Code
Machine code is the raw 0's and 1's that can be interpreted by the processor in order to execute programs. It is rarely used or modified directly since changes to source code occur at the C Code or Assembly Code levels instead. However, it may be captured as an artifact of the build process for a particular program.
Loading
Loading
NIOS Processor Image
Flashed
Altera DE2 FPGA Hardware
NIOS Processor Hardware