CSE 420
Computer Architecture I

Brief Review
Computer Organization & Assembly Language

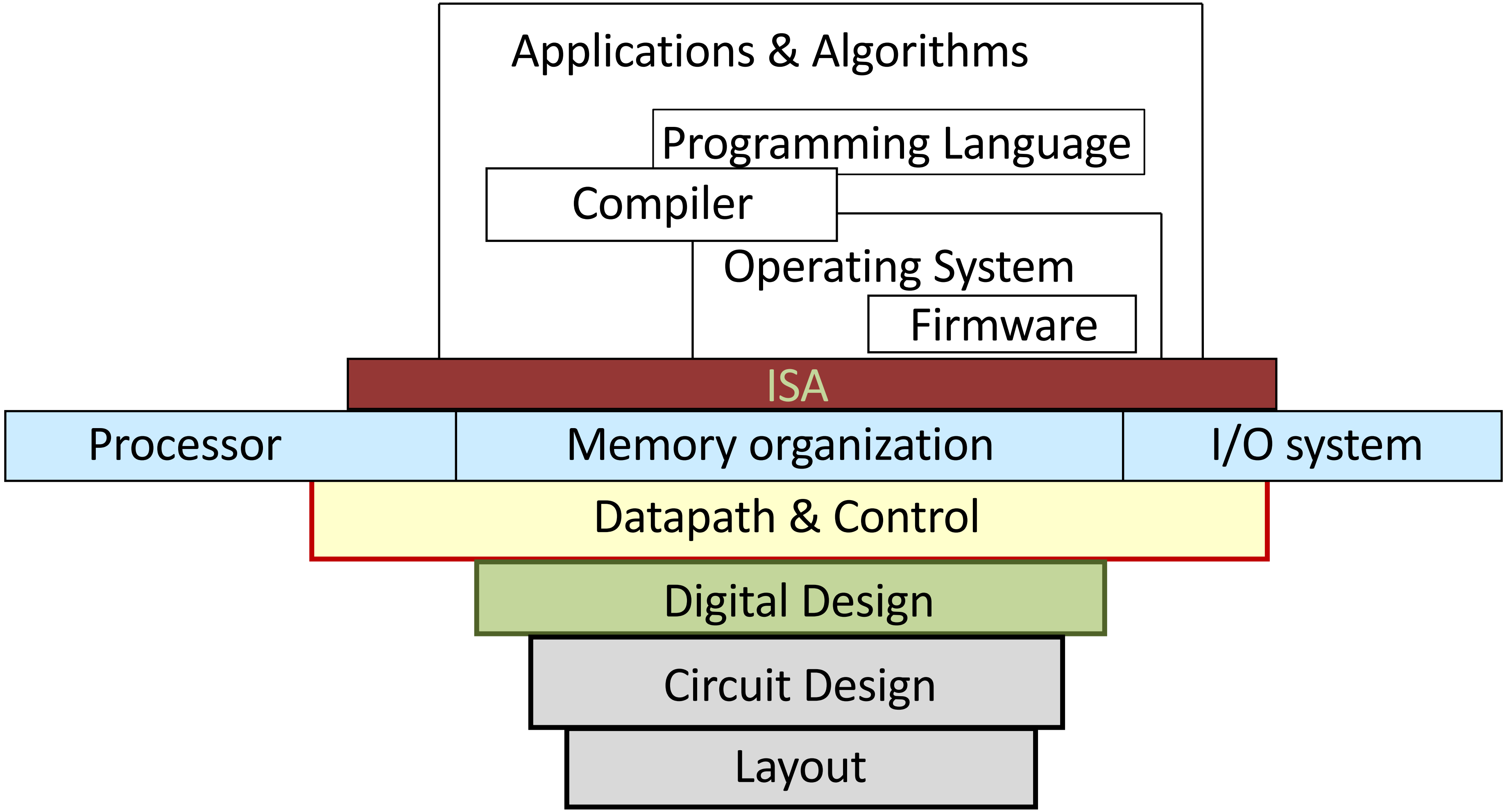
Prof. Michel A. Kinsy

Software Mechanics for Bridging

- The Art of Abstraction

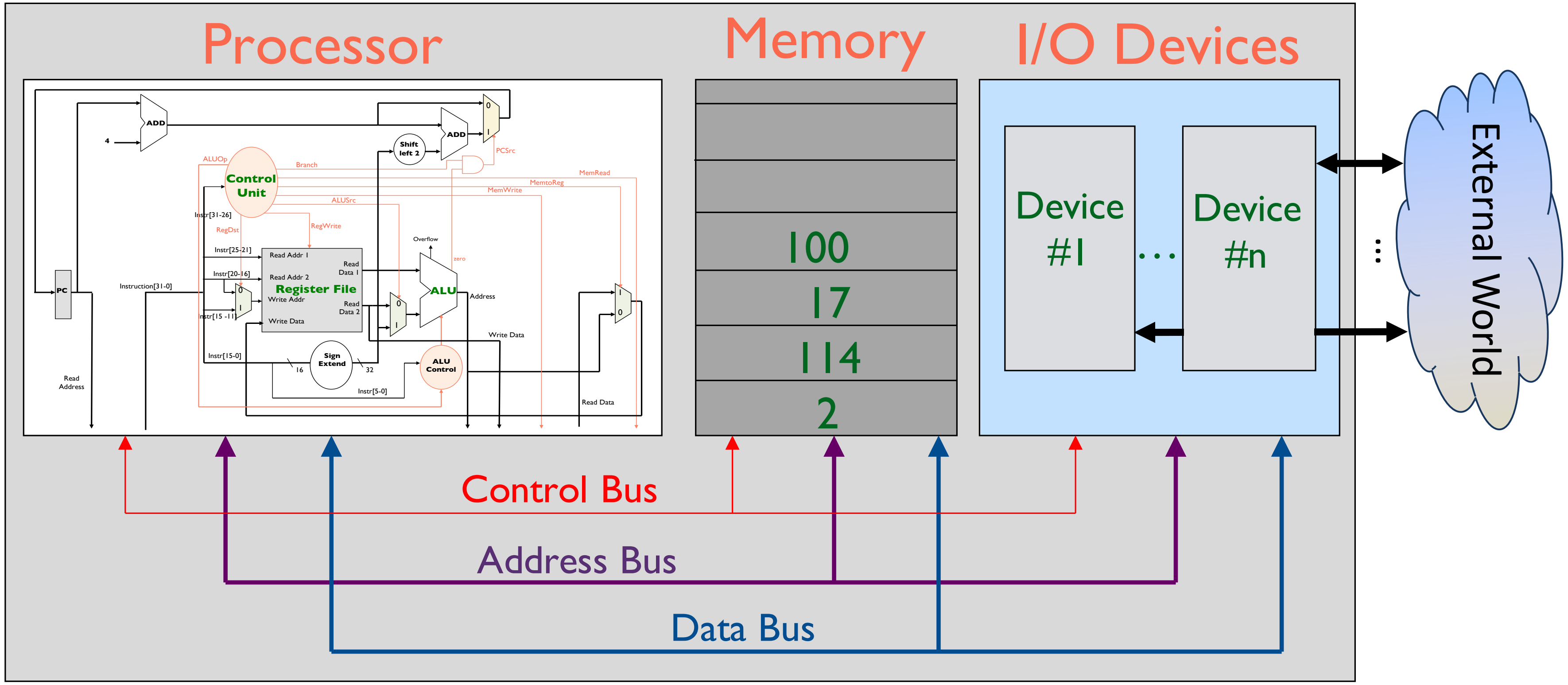
Application
Algorithm
Programming Language
Operating System/Virtual Machine
Instruction Set Architecture (ISA)
Microarchitecture
Register-Transfer Level (RTL)
Circuits
Devices
Physics

Another View of the Abstraction

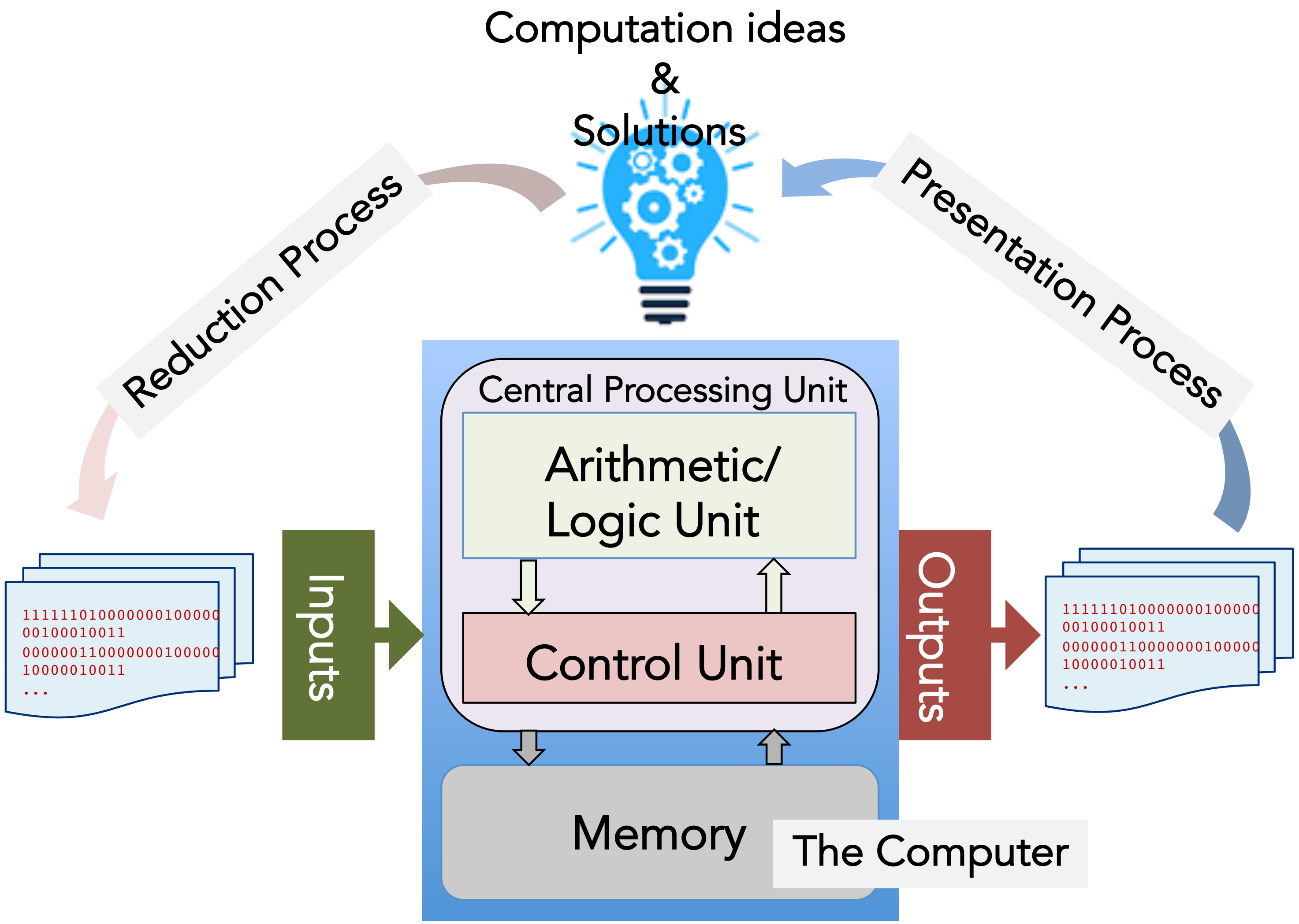


Computer Organization

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



Computing Process



Hardware Prospective

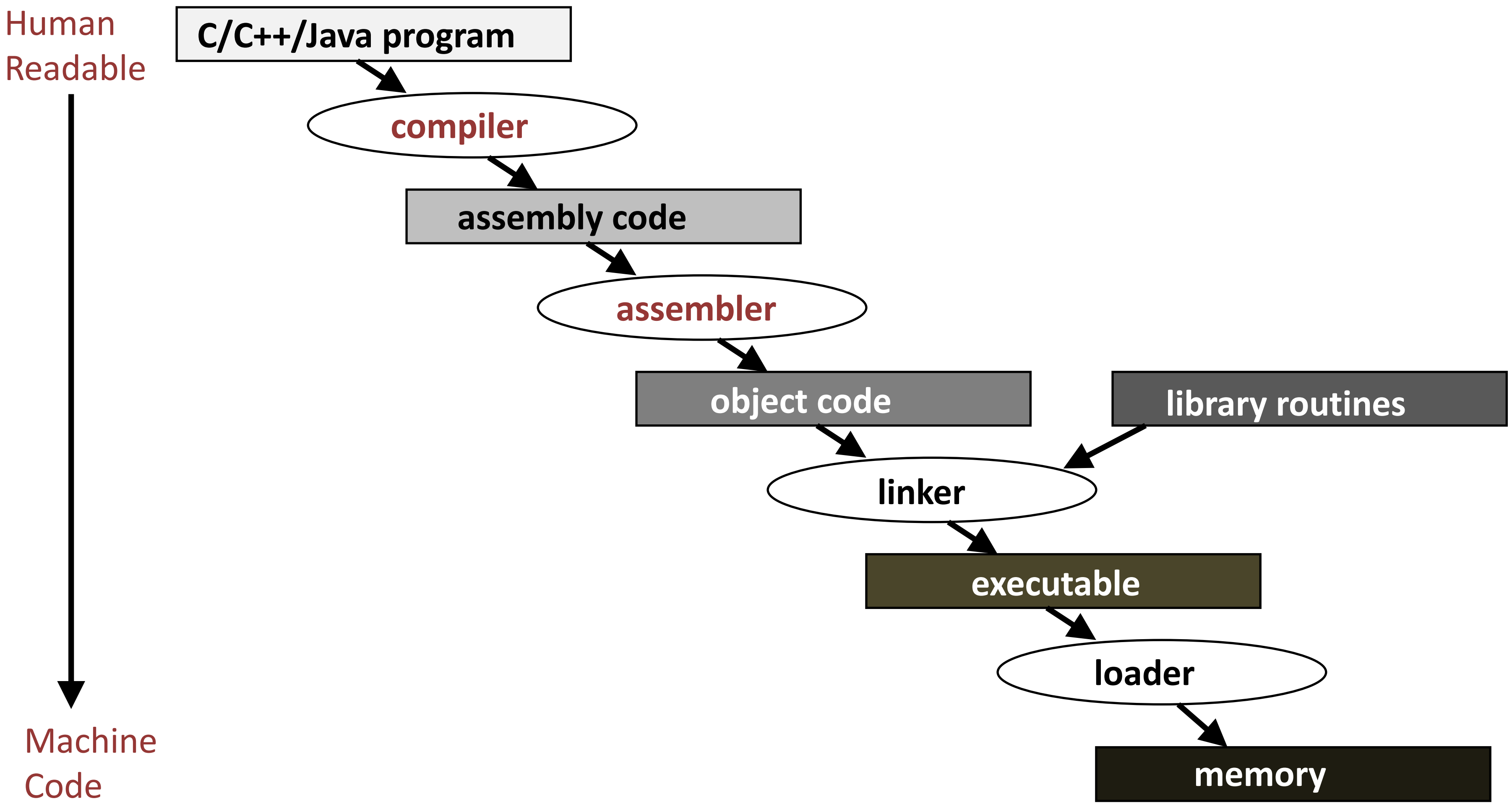
```
1111 1110 0000 0001 0000 0001 0001 0011  fe010113  // 0000019c addi  sp,sp,-32
0000 0000 0001 0001 0010 1110 0010 0011  00112e23  // 000001a0 sw    ra,28(sp)
0000 0000 1000 0001 0010 1100 0010 0011  00812c23  // 000001a4 sw    s0,24(sp)
0000 0010 0000 0001 0000 0100 0001 0011  02010413  // 000001a8 addi  s0,sp,32
0000 0000 1000 0000 0000 0111 1001 0011  00800793  // 000001ac addi  a5,zero,8
1111 1110 1111 0100 0010 0110 0010 0011  fef42623  // 000001b0 sw    a5,-20(s0)
1111 1110 1100 0100 0010 0111 1000 0011  fec42783  // 000001b4 lw    a5,-20(s0)
0000 0000 0000 0111 1000 0101 0001 0011  00078513  // 000001b8 addi  a0,a5,0
0000 0000 0000 0000 0000 0000 1001 0111  00000097  // 000001bc auipc ra,0x0
1111 0110 0100 0000 1000 0000 1110 0111  f64080e7  // 000001c0 jalr  ra,-156(ra)
```

Real Machine Code

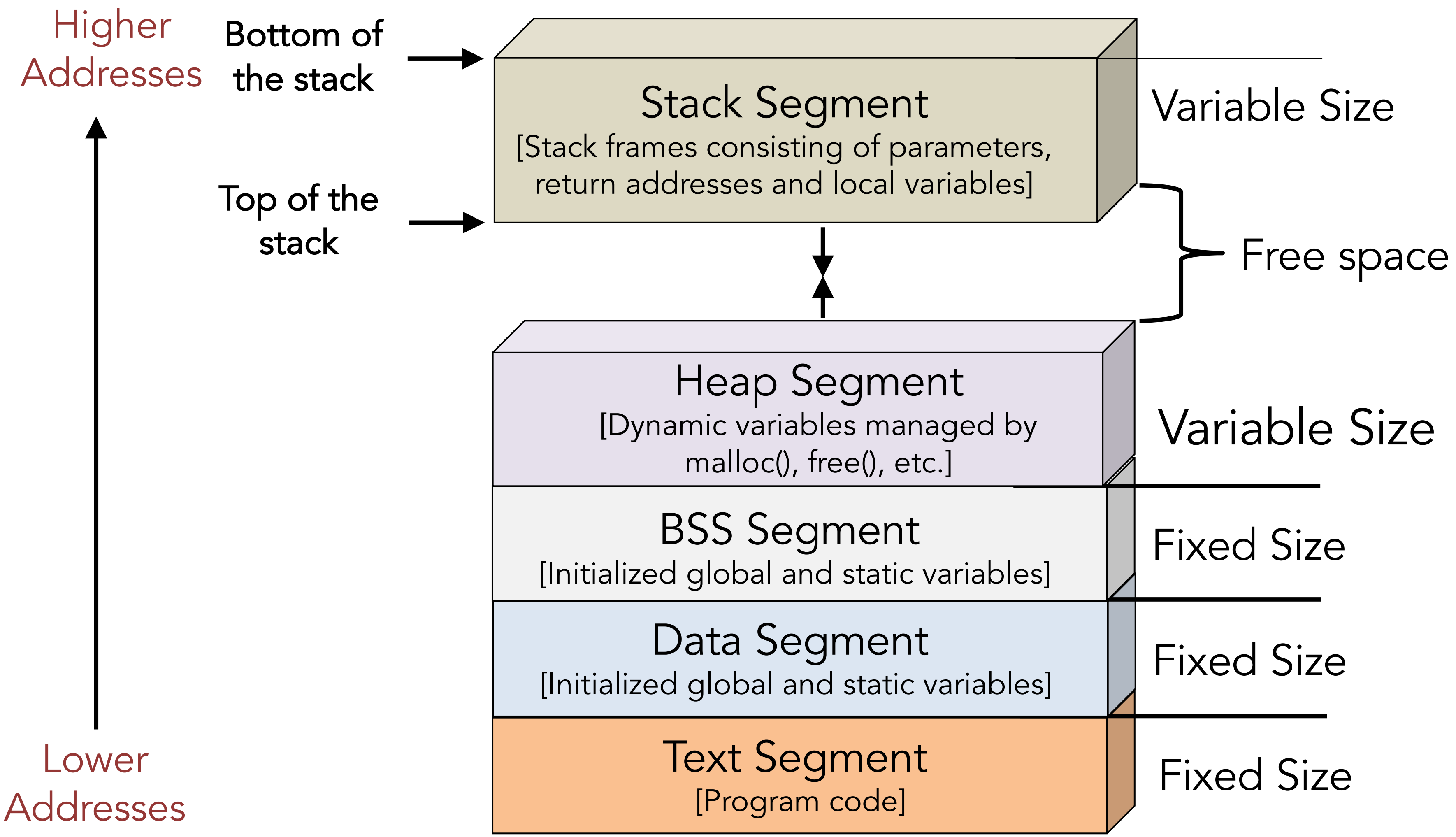
Addresses

Bridging/Compiling Process

- High-Level Language



Program memory management



Application Side

- Higher-level languages
  - Allow the programmer to think in a more natural language and for their intended use
  - Improve programmer productivity Improve program maintainability
  - Allow programs to be independent of the computer on which they are developed
    - Compilers and assemblers can translate high-level language programs to the binary instructions of any machine

Application Side

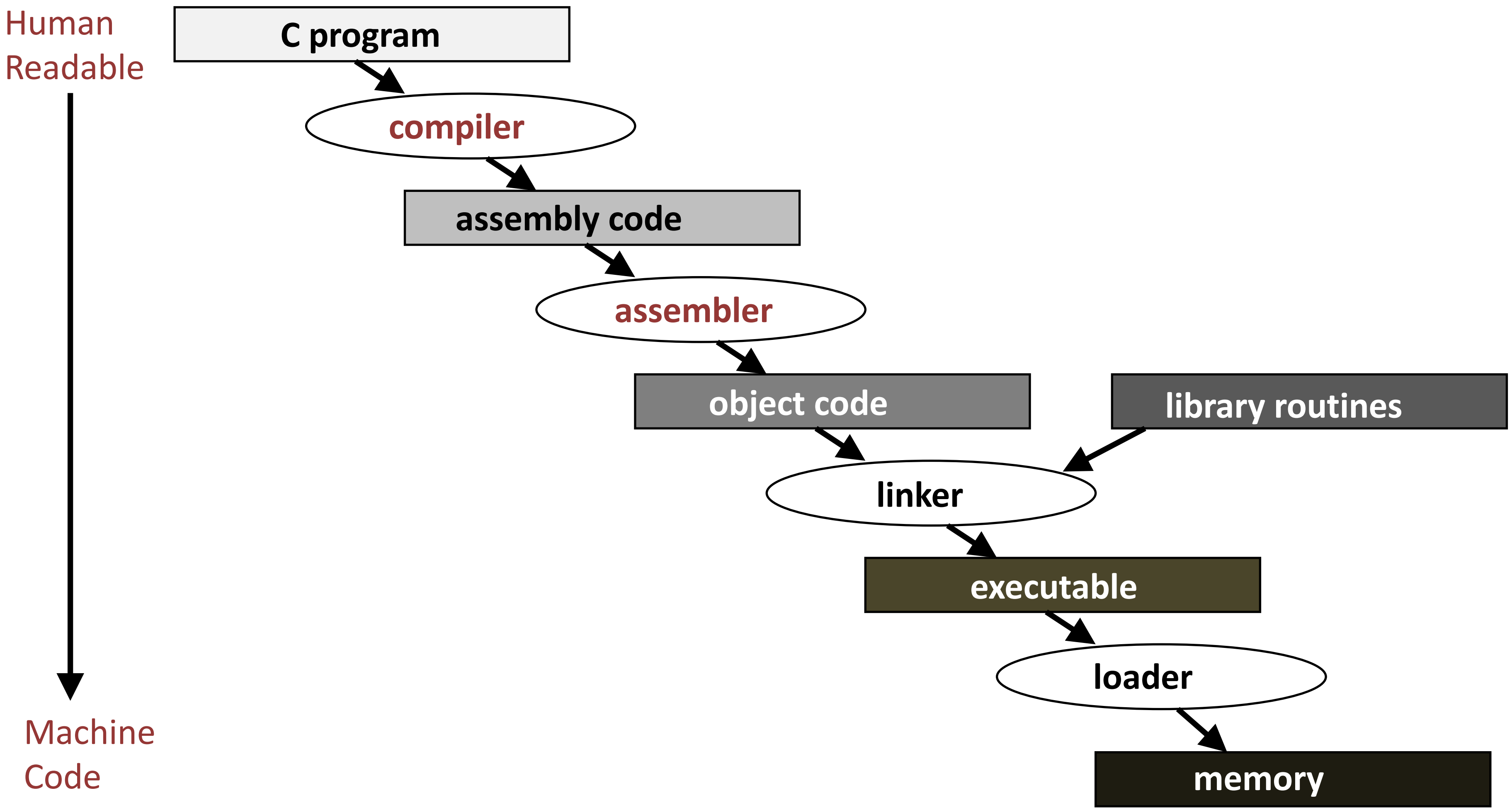
- Higher-level languages
  - Allow the programmer to think in a more natural language and for their intended use
  - Improve programmer productivity Improve program maintainability
  - Allow programs to be independent of the computer on which they are developed
  - Emergence of optimizing compilers that produce very efficient assembly code
  - As a result, very little programming is done today at the assembler level

System Software Side

- System software
  - Operating system – supervising program that interfaces the user's program with the hardware (e.g., Linux, MacOS, Windows)
    - Handles basic input and output operations
    - Allocates storage and memory
    - Provides for protected sharing among multiple applications
  - Compiler – translate programs written in a high-level language (e.g., C, Java) into instructions that the hardware can execute

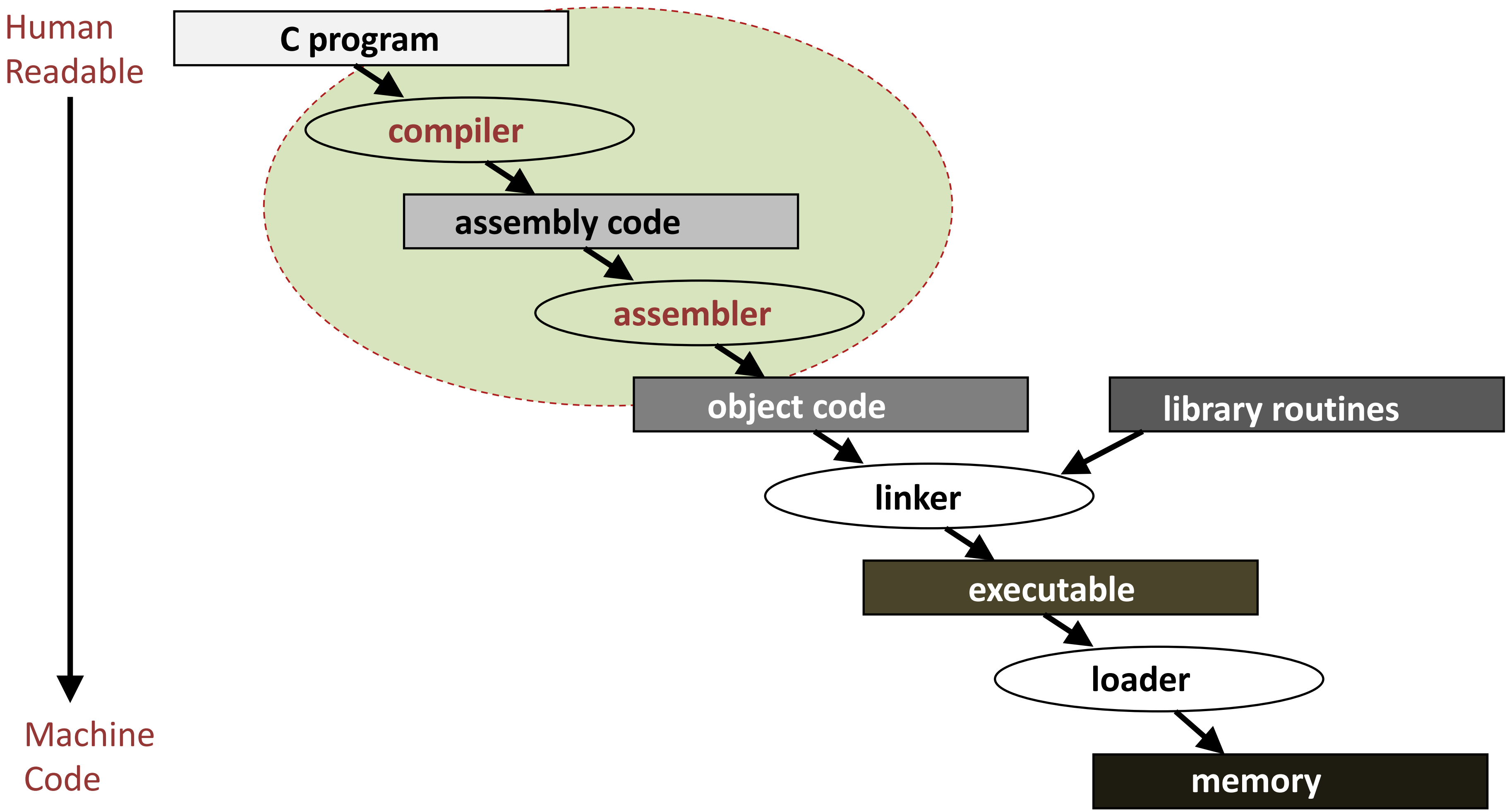
Application Compiling Process

- C Language



Application Compiling Process

- C Language



Why is assembly level view?

- To become familiar with the process of compiling a program/application (e.g., C) onto a computer system
- To know what assemblers are and what compilers do
- To understand the computer hardware view of the program/application

Why is assembly level view?

- To become familiar with the process of compiling a program/application (e.g., C) onto a computer system
- To, then, fully realize why computers are built the way they are
  - In turn, you will gain new insights into how to write better and more efficient code
  - And explore new opportunities in the field of embedded system programming

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

```
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
```

Hardware Prospective

```
1111 1110 0000 0001 0000 0001 0001 0011  fe010113  // 0000019c addi  sp,sp,-32
0000 0000 0001 0001 0010 1110 0010 0011  00112e23  // 000001a0 sw    ra,28(sp)
0000 0000 1000 0001 0010 1100 0010 0011  00812c23  // 000001a4 sw    s0,24(sp)
0000 0010 0000 0001 0000 0100 0001 0011  02010413  // 000001a8 addi  s0,sp,32
0000 0000 1000 0000 0000 0111 1001 0011  00800793  // 000001ac addi  a5,zero,8
1111 1110 1111 0100 0010 0110 0010 0011  fef42623  // 000001b0 sw    a5,-20(s0)
1111 1110 1100 0100 0010 0111 1000 0011  fec42783  // 000001b4 lw    a5,-20(s0)
0000 0000 0000 0111 1000 0101 0001 0011  00078513  // 000001b8 addi  a0,a5,0
0000 0000 0000 0000 0000 0000 1001 0111  00000097  // 000001bc auipc ra,0x0
1111 0110 0100 0000 1000 0000 1110 0111  f64080e7  // 000001c0 jalr  ra,-156(ra)
```

Real Machine Code

Addresses

Assembly Code

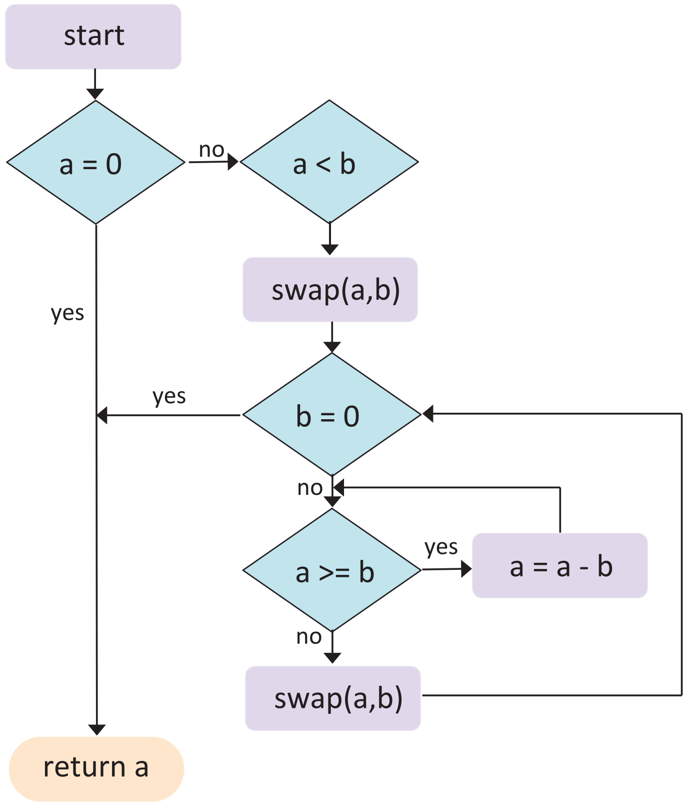
- Three types of statements in assembly language
  - Typically, one statement per a line
  1. Executable assembly instructions
     - Operations to be performed by the processor
  2. Pseudo-Instructions and Macros
     - Translated by the assembler into real assembly instructions
     - Simplify the programmer task
  3. Assembler Directives
     - Provide information to the assembler while translating a program
     - Used to define segments, allocate memory variables, etc.

Computer Organization Overview

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

Assembly Code

- There are 3 main types of assembly instructions
  - Arithmetic
    - add, sub, mul, sll, srl, and, or, etc.
  - Load/store
    - lw,sw,lb,sb
  - Conditional – branches
    - beq, bne, j, jra



Assembly Code

- There are 3 main types of assembly instructions
  - Arithmetic
    - add, sub, mul, sll, srl, and, or, etc.
  - Load/store
    - lw,sw,lb,sb
  - Conditional – branches
    - beq, bne, j, jra

```
.L5:
  lw a4,-36(s0)
  lw a5,-40(s0)
  blt a4,a5,.L4
  lw a4,-36(s0)
  lw a5,-40(s0)
  sub a5,a4,a5
  sw a5,-36(s0)
  j .L5
.L4:
  lw a5,-36(s0)
```

Assembly Code

- There are 3 main types of assembly instructions
  - Arithmetic
    - add, sub, mul, sll, srl, and, or, etc.
  - Load/store
    - lw,sw,lb,sb
  - Conditional – branches
    - beq, bne, j, jra
- Assembly language instructions have the format:
  - [label:] mnemonic [operands] [#comment]

```
.L2
beqz x1, done        # if(x1 == 0) goto done
```

Assembly Code

- There are 3 main types of assembly instructions
  - Arithmetic
    - add, sub, mul, sll, srl, and, or, etc.
  - Load/store
    - lw,sw,lb,sb
  - Conditional – branches
    - beq, bne, j, jra
- Assembly language instructions have the format:
  - [label:] mnemonic [operands] [#comment]

```
main:
   addi sp,sp,-32
   sd ra,24(sp)
```

Assembly Code

- There are 3 main types of assembly instructions
- Assembly language instructions have the format:
  - [label:] mnemonic [operands] [#comment]
- Label: (optional)
  - Marks the address of a memory location
  - Typically appear in data and text segments

```
int array [] = {2, 4, 5, 0, 1, 7};
int main(void) {
  int x,y,z;
  x = array[0];
  y = array[1];
  z = array[2];
  ...
```

Assembly Code

- There are 3 main types of assembly instructions
- Assembly language instructions have the format:
  - [label:] mnemonic [operands] [#comment]
- Label: (optional)
  - Marks the address of a memory location
  - Typically appear in data and text segments

```
int array [] = {2, 4, 5, 0, 1, 7};
int main(void) {
  int x,y,z;
  x = array[0];
  y = array[1];
  z = array[2];
  ...
```

```
array:
  .word 2
  .word 4
  .word 5
  .word 0
  .word 1
  .word 7
```

```
main:
addi sp,sp,-48
sw ra,44(sp)
sw s0,40(sp)
addi s0,sp,48
lui a5,%hi(array)
lw x5,%lo(array)(a5)
lw x6,4(a5)
lw x7,8(a5)
```

Assembly Code

- .DATA directive
- .TEXT directive
- .GLOBL directive
  - Declares a symbol as global

```
int array [] = {2, 4, 5, 0, 1, 7};
char name [9];
int main(void) {
  int x,y,z;
  x = array[0];
  y = array[1];
  z = array[2];
  ...
```

```
    .globl main
    .type main, @function
main:
    addi sp,sp,-48
    sw ra,44(sp)
    ...
```

Assembly Code

- .DATA directive
- .TEXT directive
- .GLOBL directive
- .BSS directive
  - The BSS contains variables that are initialized to zero or are explicitly initialized in code

```
int array [] = {2, 4, 5, 0, 1, 7};
char name [9];
int main(void) {
  int x,y,z;
  x = array[0];
  y = array[1];
  z = array[2];
  ...
```

```
  .globl name
  .bss
  .align 2
  .type name, @object
  .size name, 9
name:
  .zero 9
  .text
  .align 1
```

Assembly Code

- .DATA directive
  - Defines the data segment of a program containing data
  - The program's variables should be defined under this directive
- .TEXT directive
  - Defines the code segment of a program containing instructions
- .GLOBL directive
  - Declares a symbol as global
- .BSS directive
  - The BSS contains variables that are initialized to zero or are explicitly initialized in code

Assembly Code

```
 .LC0:
   .string "Enter positive
    integers a and b: "
   .align 2
.LC1:
   .string "%d %d"
   .align 2
.LC2:
   .string "GCD = %d"
   .text
   .align 1
   .globl main
   .type main, @function
main:
  addi sp,sp,-48
  sw ra,44(sp)
  ...
```

Directive	Arguments	Description
.2byte		6-bit comma separated words (unaligned)
.4byte		32-bit comma separated words (unaligned)
.half		16-bit comma separated words (naturally aligned)
.word		32-bit comma separated words (naturally aligned)
.asciz	"string"	emit string (alias for .string)
.string	"string"	emit string
.macro	name arg1 [, argn]	begin macro definition \argname to substitute
.type	symbol, @function	accepted for source compatibility
…	…	…

Assembly Languages

- Assemblers:
  - Convert mnemonic operation codes to their machine language equivalents
  - Convert symbolic operands to their equivalent machine addresses
  - Build the machine instructions in the proper format
  - Convert the data constants to internal machine representations
  - Write the object program and the assembly listing

System Calls

- Programs do input/output through system calls
- To obtain services from the operating system
- Using the syscall system services
- Issue the syscall instruction

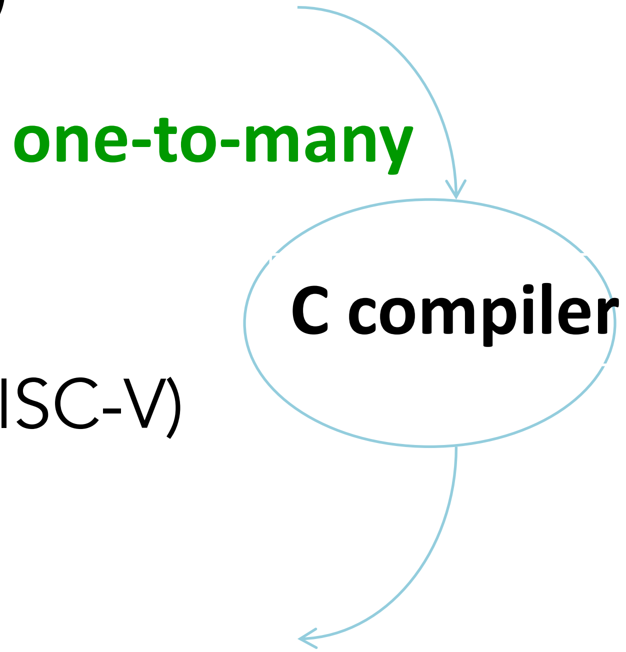
```
addi a0,a5,%lo(.LC0)
call printf
...
call scanf
lw a5,-36(s0)
...
```

- Retrieve return values, if any, from result registers

Application Compiling Process

- High-level language program (in C)

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

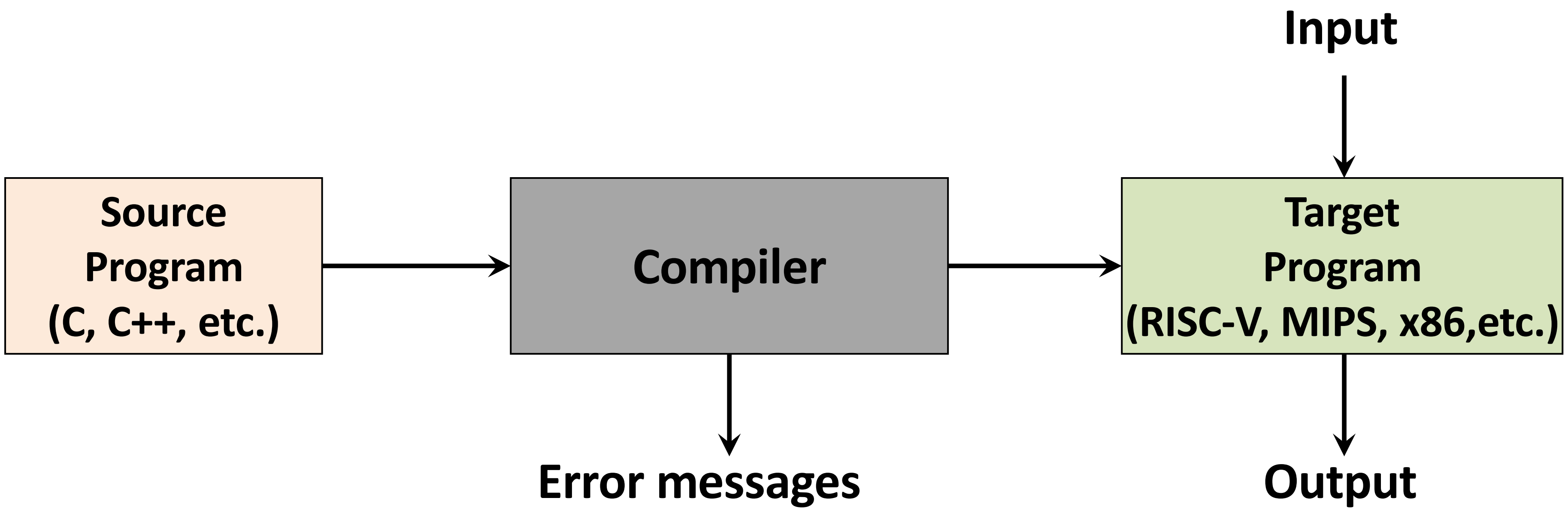


- Assembly language program (for RISC-V)

```
swap:
  addi sp,sp,-48
  ...
  mv a5,a1
  ...
  ld s0,40(sp)
  addi sp,sp,48
  jr ra
```

Application Compiling Process

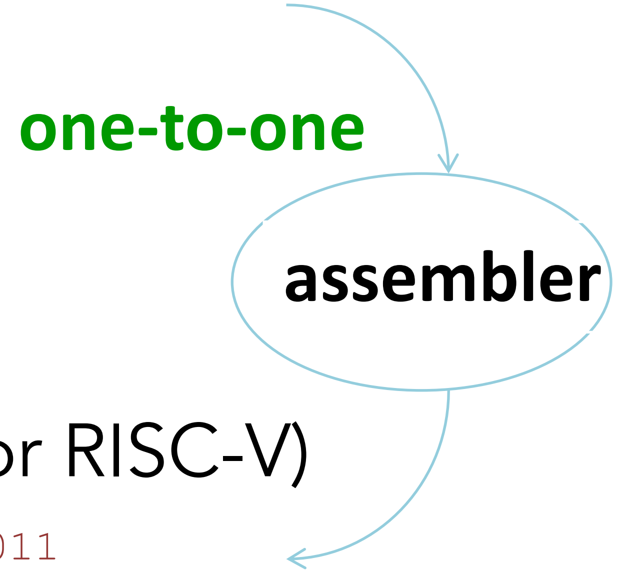
- A compiler is a software program that translates a human-oriented high-level programming language code into computer-oriented machine language



Application Compiling Process

- Assembly language program (for RISC-V)

```
swap:
  addi  sp,sp,-48
  ...
  mv  a5,a1
  ...
  ld  s0,40(sp)
  addi  sp,sp,48
  jr  ra
```

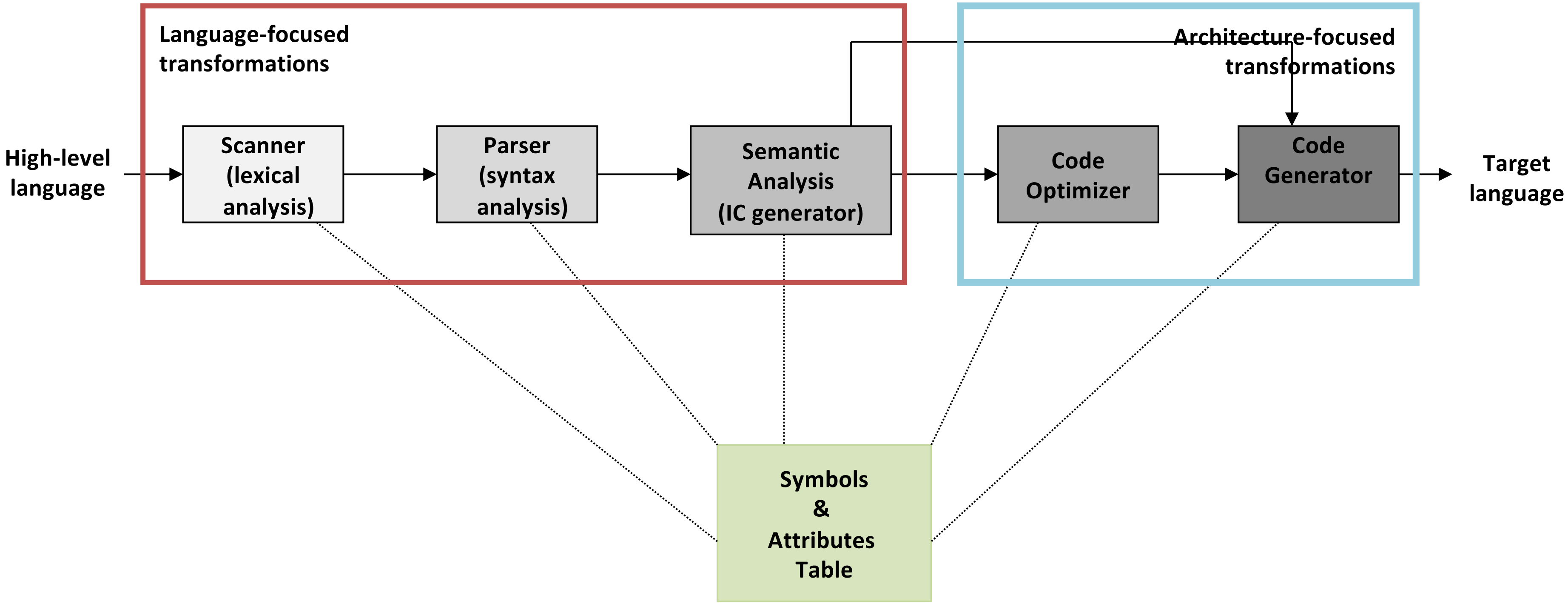


- Machine (object, binary) code (for RISC-V)

```
111111010000   00010   000 00010   0010011
000000110000   00010   000 01000   0010011
...
```

Application Compiling Process

- Detailed compilation process

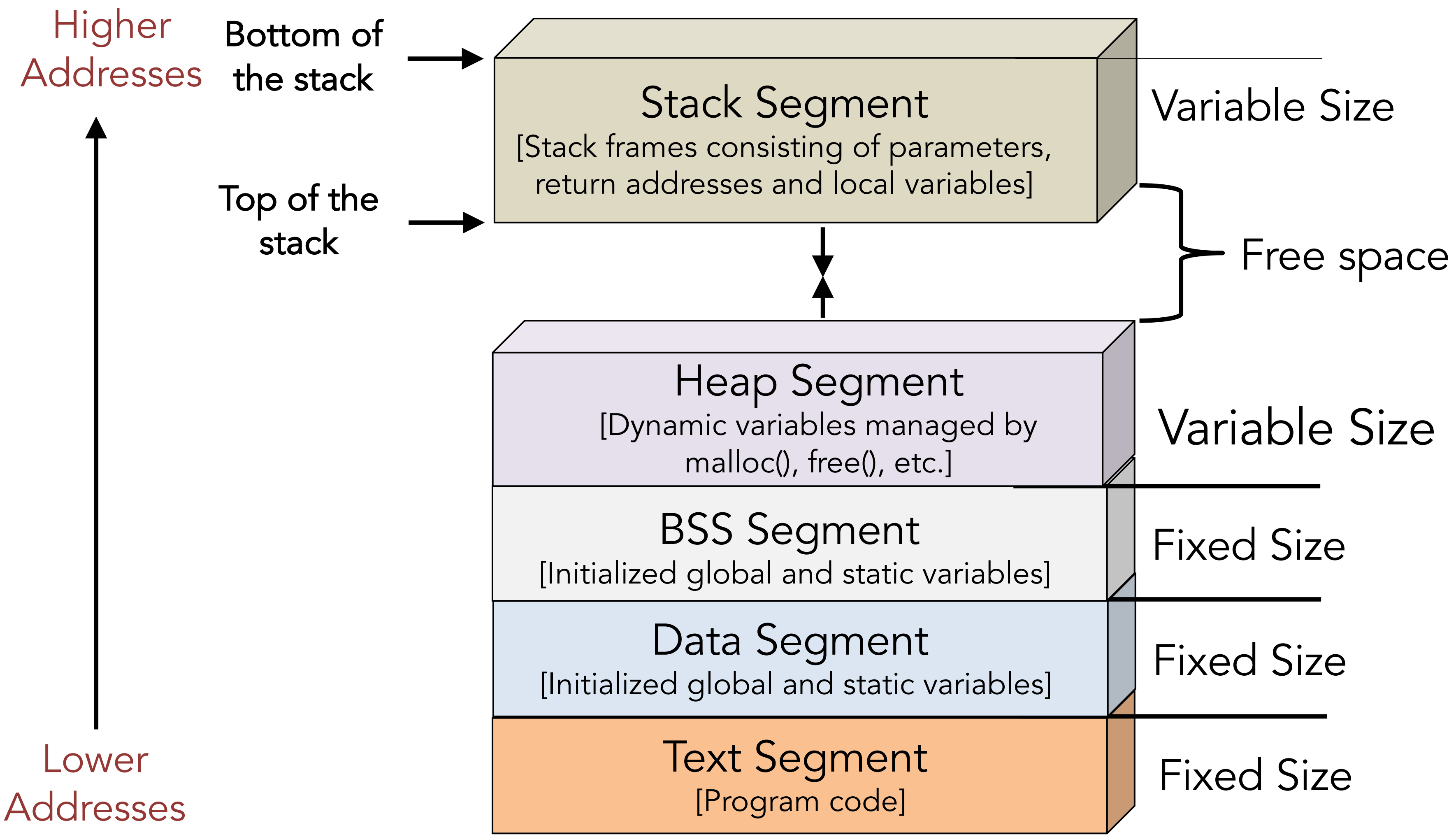


- More on this later when you take a course on compilers

Application Compiling Process

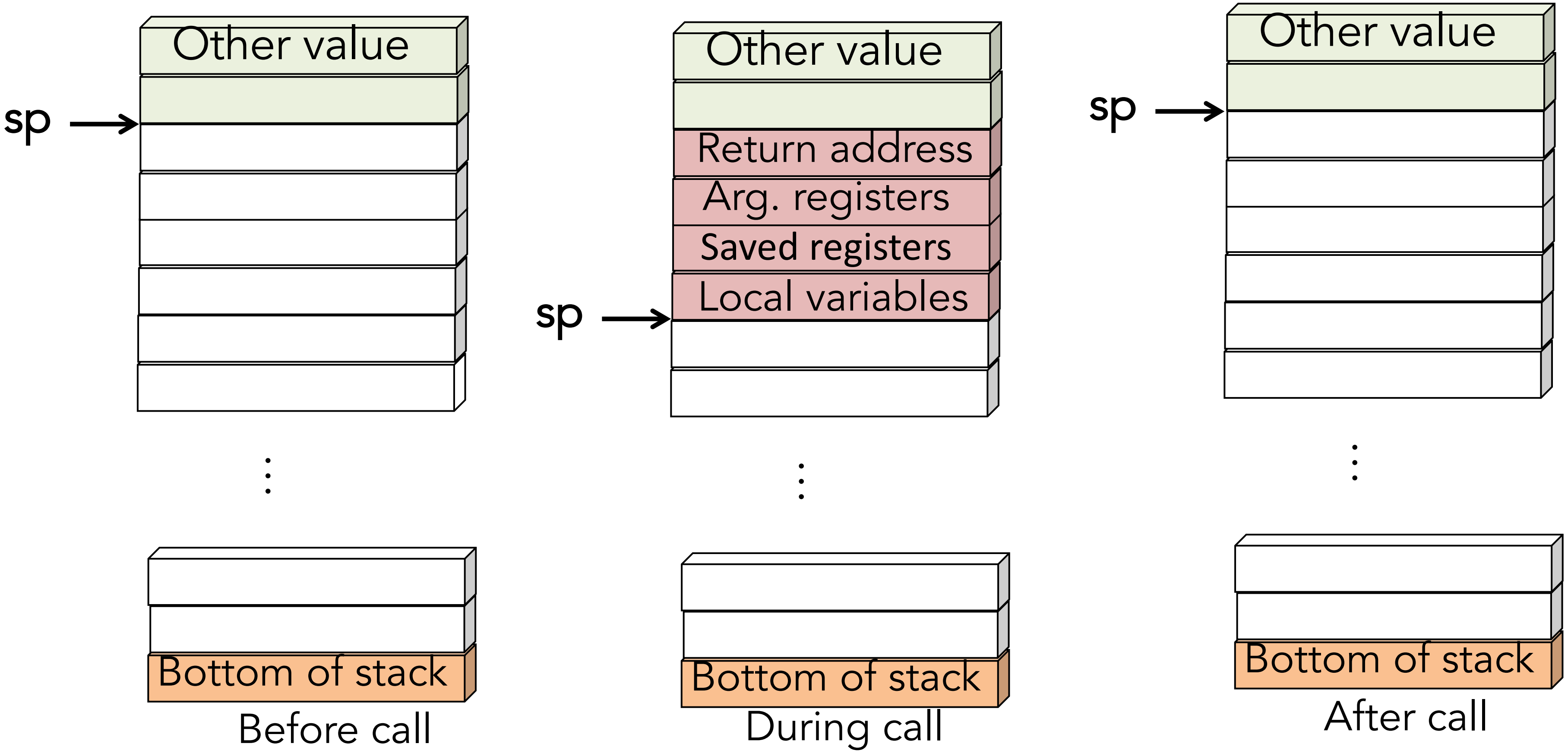
- Symbol Table
  - Identifiers are names of variables, constants, functions, data types, etc.
  - Store information associated with identifiers
  - Information associated with different types of identifiers can be different
  - Information associated with variables are name, type, address, size (for array), etc.

Program memory management



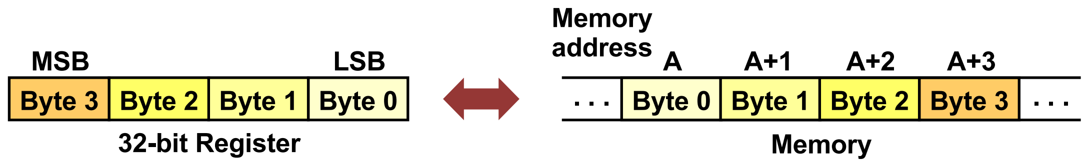
Stack Structure

- Procedure frame or activation record

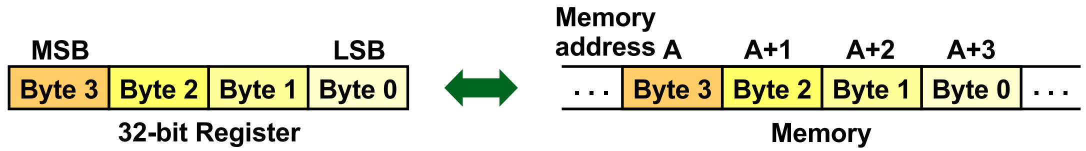


Big Endian – Little Endian

- Processors can order bytes within a word in two ways
  - Little Endian
    - Least significant byte stored at lowest byte address
    - Intel IA-32, Alpha, AMD



  - Big Endian
    - Most significant byte stored at lowest byte address
    - SPARC, PA-RISC, IBM



Big Endian – Little Endian

```
int main(void) {
    int var;          // Integer values
    char *ptr;        // Pointer

    // Assign 'var' and output it in byte order and as a value
     var = 0x12345678;
     ptr = (char *) &var;

     printf("ptr[0] = %02X \n", ptr[0]); // Prints 78
     printf("ptr[1] = %02X \n", ptr[1]); // Prints 56
     printf("ptr[2] = %02X \n", ptr[2]); // Prints 34
     printf("ptr[3] = %02X \n", ptr[3]); // Prints 12

     printf("var = %08X \n", var);        // Prints 12345678
}
```

Big Endian – Little Endian

```
int main(void) {
    int var;         // Integer values
    char *ptr;       // Pointer

    // Assign 'var' and output it in byte order and as a value
     var = 0x12345678;
     ptr = (char *) &var;

     printf("ptr[0] = %02X \n", ptr[0]); // Prints 78
     printf("ptr[1] = %02X \n", ptr[1]); // Prints 56
     printf("ptr[2] = %02X \n", ptr[2]); // Prints 34
     printf("ptr[3] = %02X \n", ptr[3]); // Prints 12

     printf("var = %08X \n", var);          //
}
```

Big Endian	Little Endian
Solaris on SPARC	Windows on Intel
ptr[0] = 12 ptr[1] = 34 ptr[2] = 56 ptr[3] = 78	ptr[0] = 78 ptr[1] = 56 ptr[2] = 34 ptr[3] = 12
var = 12345678	var = 12345678

Concluding Note

- If you feel the need to learn or refresh some of these foundational concepts, you might consider taking CSE 420 first.