**Lecture #2a - CPU registers - Program memory - Runtime Stack**

UPLOAD

**CPU - Central Processing Unit -** Comprised of electronic circuitry that execute program

instructions dealing with such things as arithmetic, logic and input / output.

**- x86 processor**, a family of 32-bit processors made by Intel and run on Microsoft PCs.

**Registers** - Registers are high-speed storage locations inside the CPU.

- **x86 CPU includes the following registers**

**The** Intel **x86** **processor** uses complex **instruction** set computer (CISC) architecture, which means there is a modest number of special-purpose registers instead of large quantities of general-purpose registers.

Extended Base

; Pointer

Extended

Stack Pointer

* Image shows the registers of an x86 CPU**8 General-purpose registers**
* **6 Segment registers**
* **1 Processor status flags register**

(EFLAGS) (32-bit register)

* **1 Instruction Pointer**

(EIP) (32-bit register)

**General-purpose registers** - The 8 general purpose registers are used for arithmetic and data.

* **EAX** – **EBX** – **ECX - EDX - ESI - EDI**
* **ESP – Extended Stack Pointer** register addresses data on the stack.
  + The ESP is not used for math or data transfer.
  + A "stack pointer" register tracks the top of the stack; it is adjusted each time a value is "pushed" onto the stack
  + By default, the stack grows downward in memory, so newer values are placed at

lower memory addresses.

* **EBP - Extended Base Pointer** register- (Also called **Frame Pointer**)
  + Used by high-level languages (C++) to reference function parameters and local variables on

the runtime stack.

* + The EBP is not used for ordinary math or data transfer instructions

**EIP** – **Instruction pointer** - Holds the memory address of the next instruction to be fetched.

* Certain machine instructions manipulate the EIP, causing the program to branch to a

new location. (C++ if statements).

**Segment registers**

* **CS** – Code segment register
  + Image shows the 16-bit segment registers and the Eflags and EIP registers of an x86 CPUHolds a pointer to program instructions (code)
* **DS** – Data segment register
  + Holds pointers to variables (data)
  + The data segment contains initialized global variables and local static variables
  + The size of this segment is determined by the size of the values in the program's source code, and does not change at [run time](https://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase)).

**Runtime Stack** (call stack)

- Each process’s memory has a **stack segment (ss)** functioning as a runtime stack.

* The **stack segment** contains the call stack,typically located in the higher parts of memory.
  + A "stack pointer" register (ESP) tracks the top of the stack.
  + The ESP is adjusted each time a value is "pushed" on or popped off the stack.
  + The set of values pushed for one function call is termed a "stack frame" or activation record.
  + When a function is called, **these items are pushed on the stack:**
    - Return address (ra) (Always)
    - Space for a return value (if not a void function)
    - Actual parameters (arguments)
    - Local variables (variables declare in the function)
  + The stack segment traditionally adjoined the heap segment and they grew towards each other; when the stack pointer met the heap pointer, free memory was exhausted. With large address spaces and virtual memory techniques they tend to be placed more freely, but they still typically grow in a converging direction. On the standard PC x86 architecture the stack grows toward address zero, meaning that more recent items, deeper in the call chain, are at numerically lower addresses and closer to the heap. On some other architectures it grows the opposite direction.

**Program Memory** (conceptual view)

* **Process** - A program running in memory is called process.
  + Each process has its own code segment, data segment, bss, heap and stack.
* **CPU segment registers** hold addresses of where the segments are located in memory.
  + **DS register** holds the memory address of the data segment in memory.
  + **CS register** holds the memory address of the code segment in memory.
  + **SS register** holds the memory address of the stack segment.
    - **ESP** (Stack Pointer) holds the memory address of the top of the stack.

A screenshot of a computer

Description automatically generated

***Runtime Stack***

***(Stack Segment - SS)***

**The stack segment** contains

int value; // uninitialized global variable

int main()

{

int number; // local variable goes on the stack

static int val; // uninitialized static variable

**BSS** - Block Starting Symbol.

The BSS holds uninitialized data, both variables and constants

**The data segment** contains initialized static variables,

***Data Segment***

( global variables and static local variables).

**The code segment** contains program code. As a memory region,

***Code Segment***

the code segment is be placed below the heap or stack in order

to prevent heap and stack overflows.

**Calling a function and passing arguments to the stack by value.**

* First push argument values on the stack before execution jumps tothe function code.

Ex: int addNumbers(int val1, int val2); 🡨 C++ function prototype

int sum = addNumbers(val1, val2); 🡨 C++ function call

Ex: Assembly language (MASM)

.data

int val1 = 5; val1 DWORD 5 **Stack before the call**

A screenshot of a computer

Description automatically generatedint val1 = 6; val2 DWORD 6

.code val2

push val2 val1

push val1

call AddNumbers

**Calling a function and passing by Reference**

* First push offsets of argument on the stack before execution jumps tothe function code.

Ex: void addNumbers (int& val1, int& val2); 🡨 C++ function prototype

addNums(val1, val2); 🡨 C++ function call

Ex: Assembly language (MASM)

.data **Stack before the call**

A screenshot of a computer

Description automatically generated val1 DWORD 5

val2 DWORD 6

.code

push OFFSET val2

push OFFSET val1

call AddNumbers

**Stack after the procedure call** (whether passed by value or reference).

A screenshot of a computer

Description automatically generated

**EBP** - **Base pointer register (Frame pointer**)

**-** C++ and other high-level languages access stack parameters using offsets from

the EBP register.

- **Offset** - The offset is the number of bytes from a point of origin.

**EBP** is called the **base pointer** or **frame pointer** because it holds the base address of

the stack frame.

* The **EBP** can be explicitly used in a program.
  + When pushed on the stack, it provides a way to access parameters on the stack.
  + After the EBP has been pushed on the stack, it is assigned the address in

the ESP (Stack Pointer)

* The value in the EBP does not change during the function execution.

Ex: Assembly language (MASM)

The values on the stack (6 and 5) can be accessed by using an offset from the EBP.

.code

push val2

push val1

call AddNumbers ; Call the AddNumbers procedure (function)

. . .

. . .

AddNumbers PROC

**push ebp ;** Push the stack pointer

**mov ebp,esp ;** EBP = ESP

mov eax,[ebp+12]

A screenshot of a computer

Description automatically generated add eax,[ebp+8]

**pop ebp**

ret

AddNumbers ENDP

Note: The EBP must be popped

before RET.

**Passing an Array by Reference**

* High-level languages (C++, Java, etc.) always pass arrays to functions by reference.
  + The address of the array is pushed on the stack.
  + A function can get the address from the stack and use it to access the array.
* The calling program passes the address of the array, along with a size of the array.
* The EBP holds the address of the stack frame and therefore, provides a way to access

The parameters on the stack.

Ex: void getNums(int numbers[]); 🡨 C++ function prototype

int numbers[50]; 🡨 Declare an array

getNums(numbers); 🡨 C++ function call

Ex: Assembly language (MASM)

.data

count = 50

numbers DWORD 50 DUP(?) ; 🡨 Declare an array

 .code

; Push the offset of the array

push OFFSET numbers

push COUNT

call GetNums 🡨 Function call

. . .

. . .

GetNums PROC

push ebp

mov ebp,esp ; base of stack frame

mov esi,[ebp+12] ; address of the numbers array

mov ecx,[ebp+8] ; ECX = 3

. . .

. . .

pop ebp

ret

GetNums ENDP

**Lecture #2a - Runtime Stack** - NOTE: USE THE Stack Diagram (Lecture 2b) with this lecture

When a function is called, an **activation record** (also called a **stack frame**) is created on the

runtime stack (part of RAM)

A stack frame consists of the following 4 items.

Note: Every stack frame must have a return address, but the other 3 items may, or may not, exist.

1. **Arguments** (if the function has any parameters)

2. **Space for a return value** (if it is not a void function)

3. **return address** - (always) - The memory address of the next instruction after the

function is done.

4. **local variables** – (if any variables are declared within the function)

**// prototypes**

void getAges(int& firstAge, int& secondAge);

**float calcAvg(int firstAge, int secondAge);**

**void displayAvg(float avg);**

int main() - When main is called, a stack frame is pushed on the stack

{

int firstAge = 0; // 21

int secondAge = 0; // 22

float avg = 0;

getAges(firstAge, secondAge); // function call

**avg = calcAvg(firstAge, secondAge); // function call**

**displayAvg(float avg); // function call**

return 0;

}

// -----------------------

void (int& firstAge, int& secondAge)

{

cout << "Enter the age of the first person: ";

cin >> firstAge;

cout << "Enter the age of the second person: ";

cin >> secondAge;

}

// -----------------------

**float calcAvg(int firstAge, int secondAge)**

{

float avgAge;

avgAge = (firstAge + secondAge) / 2.0;

return avgAge;

}

// -----------------------

**void displayAvg(float avg);**

**{**

**cout << "The average age is " << avg << ".\n";**

**}**

// -----------------------

**CPU registers:**

5000

**EIP** (Instruction pointer) 🡪 (stack)

- It holds the memory address of the next instruction to be fetched.

11010111111101001101000011011100

* **IR register** - Instruction Register
  + IR holds the instruction that has been fetched and is currently being executed.

**CPU** Machine Cycle:

1. Fetch an instruction (and increment EIP)
2. Decode the instruction
3. Execute the instruction

**Main memory (RAM)**

Instructions Memory

Addresses little endian architecture

11011100 **5000**

11010000 5001 getAges(firstAge, secondAge); // function call

11110100 5002

11010111 5003

11110100 5004

11010101 5005 **calcAvg(firstAge, secondAge);** // function call

11011100 5006

11110101 5007

11111100 5008

01010101 5009 avg = 21.5;

10000101 5010

10010100 5011

**01010001** 5012

**11010001** 5013 **displayAvg(float avg);** // function call

**10010100** 5014

**00010101** 5015

11010000 5016

01010101 5017 return 0;

11010100 5018 }

01010101 5019

10000100 **5020**

00010101 5021

11010101 5022

10010101 5023

11000101 5024

10010100 5025

Void getAges (int& firstAge, int& secondAge)

{

cout << "Enter the age of the first person: ";

cin >> firstAge; // 21 🡨 (user enters 21)

cout << "Enter the age of the second person: ";

cin >> secondAge; // 22 🡨 (user enters 22)

}

01000001 5026

00010100 5027

01010101 5028

10010101 5029

11010000 5030

11010101 5031

11010101 5032

11010101 5033

01010101 5034

11010100 5035

10010101 5036

00010101 5037

11010100 5038

11010001 5039

01010101 5040

11010101 5041

01010000 5042

11010101 5043

01010100 5044

11010001 5045

**float calcAvg(int firstAge, int secondAge)**

{

float avgAge;

(21 + 23) / 2.0

avgAge = (firstAge + secondAge) / 2.0;

return avgAge; // return 21.5

}

11010101 5046

00010000 5047

11010101 5048

01010101 5049

10010000 5050

10010101 5051

01010101 5052

11010101 5053

01010000 5054

11010101 5055

**01010100** 5056

**11010001** 5057

**11010101** 5058

**00010000** 5059

**void displayAvg(float avg);**

**{**

**cout << "The average age is " << avg << ".\n";**

**}**

**11010101** 5060

**01010101** 5061

**10010000** 5062

**10010101** 5063

**01010101** 5064

**11010101** 5065

**01010000** 5066

**11010101** 5067

01010100 5068

11010001 5069

. . .

. . .

**Review: Scope of a variable**

A local variable in a function is pushed on the runtime stack as part of an activation record (stack frame). It’s popped off the stack when the function is finished. Therefore, a variable’s scope (where it can be used) is only within the function in which is is declared.

**How scope is managed when only using braces**

When using braces to define a block of code the compiler manages the scope the same as it would for an *‘if’ statement*. It does this by pushing a new block (stack frame) on to the stack and proceeds to pop it off when the block of code reaches its end. Scope is created by the way things are stored on the stack.

Upon entry into the block, **a stack frame is pushed on the stack.**

The variable **num2** is allocated on the runtime stack, similar

to a variable declared in a function.

int main()

{

int num1 = 3;

{ // Curly braces define a block

**int num2 = 1;** // A variable declared in a block has local scope to that block

cout << num1 << endl; // 3

cout << num2 << endl; // 1

}

cout << num1 << endl; // 3

cout << **num2** << endl; // **Error! num2 no longer exists.**

return 0;

}