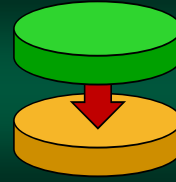




Part 7

The Stack

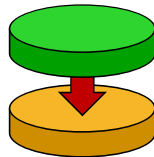


Stacks

Piles of... Data

Stack

- A stack is an abstract data structure that stores objects
- Based on the concept of a stack of items – like a stack of dishes
- Data can only be added to or removed from the top of the stack



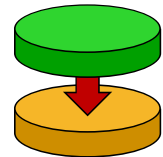
11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

3

Stack

- This gives a **First-In-Last-Out** logic (aka FILO)
- Same concept is also called **Last-In-First-Out** (LIFO)



11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

4

Examples of Stacks

- Page-visited "back button" history in a web browser
- Undo sequence in a text editor
- Deck of cards in Windows Solitaire



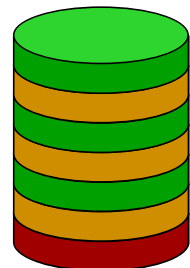
11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

5

Stack Operation: Push

- A value is added to the stack
- It is placed on the top location
- Rest of the items are "covered"



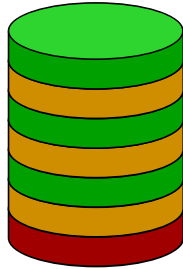
11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

6

Stack Operation: Pop

- Removes an item from the stack
- Last item added is removed
- 2nd item becomes the top



11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

7

Implementing in Memory

- On a processor, the stack stores integers
 - size of the integer the bit-size of the system
 - 64-bit system → 64-bit integer
 - these are called **words**
- A fixed location pointer (S0) defines the bottom of the stack
- A **stack pointer** (SP) gives the location of the **top** of the stack

11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

8

Approaches

- Growing upwards
 - Bottom Pointer (S0) points to the **lowest** address in the stack buffer
 - stack grows towards **higher** addresses
- Grow downwards
 - Bottom Pointer (S0) points to the **highest** address in the stack buffer
 - stack grows towards **lower** addresses

11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

9

Size of the Stack

- As an abstract data structure...
 - stacks are assumed to be infinitely deep
 - so, an arbitrary amount of data can stored
- However...
 - stacks are implemented using memory buffers
 - which are finite in size
- If the amount of data **exceeds** the allocated space, a **stack overflow** error occurs

11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

10

Subroutine Call Basics



Organizing Your Program

Subroutine Call Basics

- Subroutines are important part of virtually all computer languages
- Essential to designing **any** assembly program



11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

12

Why Use Subroutines

1. Allow commonly used functions to be written once and used whenever needed
2. Provide abstraction, making it easier for multiple programmers to collaborate on a program



11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

13

Subroutine Call

- Processors are designed to support subroutines and, as a result, contain special instructions and hardware
- The stack
 - used to save the current state of the system
 - also used to pass data between subroutines
 - ... but, we will cover these later

11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

14

When you call a subroutine...

1. Processor pushes the program counter (PC) – an address – on the stack
2. PC is changed to the address of the subroutine
3. Subroutine executes and ends with a "return" instruction
4. Processor then pops and restores the original PC
5. Execution continues after the initial call

11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

15

Nesting Calls

- Subroutines can call other subroutines
- For example:
 - Main program calls Subroutine A
 - Subroutine A calls Subroutine B
- Each time another subroutine is called...
 - a new PC is put on the stack
 - additional calls will put their PCs on the stack
 - top of the stack is always to the calling routine

11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

16

Passing Parameters

- Useful subroutines...
 - need you to be able to pass data into them
 - and be able to read data from it
- One of the easiest ways to accomplish this is by using the processors registers
- *Incredibly efficient!*



11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

17

Passing Through Registers

- However... processor might not have enough registers for each parameter
- *Other forms of passing data have to be used*



11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

18



x64 Subroutines

Organizing Your Programs ... with Intel

Instruction: Call

- The *Call Instruction* is used to transfer control to a subroutine
- Other processors call it different names such as JSR (Jump Subroutine)
- The stack is used to save the current PC

11/12/2017

Sacramento State - Cook - CSIS 35 - Fall 2017

20

Instruction: Call

Usually a label
(which is an address)

CALL *address*

11/12/2017

Sacramento State - Cook - CSIS 35 - Fall 2017

21

Instruction: Return

- The Return Instruction is used mark the end of subroutine
- When the instruction is executed...
 - the old program counter is read from the system stack
 - the current program counter is updated – restoring execution after the initial call

11/12/2017

Sacramento State - Cook - CSIS 35 - Fall 2017

22

Instruction: Return

- Do not forget this!
- If you do...
 - execution will simply continue, in memory, until a return instruction is encountered
 - often is can run past the end of your program
 - ...and run data!

11/12/2017

Sacramento State - Cook - CSIS 35 - Fall 2017

23

Instruction: Return

No arguments!

RET

11/12/2017

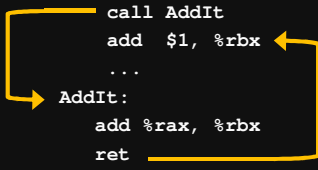
Sacramento State - Cook - CSIS 35 - Fall 2017

24

Subroutine Example

```

_start:
    mov $4, %rax
    mov $12, %rbx
    call AddIt
    add $1, %rbx
    ...
AddIt:
    add %rax, %rbx
    ret
    
```



11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

25



Saving Registers & Lost Data

Avoiding horrible side-effects

Saving Registers & Lost Data

- Each subroutine will use the registers as it needs
- So, when a sub is called, *it may modify the caller's registers*
- Some processors have few registers – so its *very* likely
- This can lead to hard-to-fix bugs if caution is not used – e.g. loop counter gets changed



11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

27

Two Solutions

- Caller saves values
 - caller saves all their registers to memory before making the subroutine call
 - after, it restores the values before continuing
 - not recursion friendly – it pushes all of them!
- Subroutine saves the values
 - push registers (it will change) onto the stack
 - before it returns, it pops (and restores) the old values off the stack

11/12/2017


Sacramento State - Cook - CSc 35 - Fall 2017

28

Saving Registers... How nice! :-)

```

DoSomething:
    push %rax
    push %rbx
    push %rcx
    ...
    pop %rcx
    pop %rbx
    pop %rax
    ret
    
```



11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

29

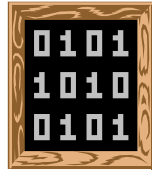


Stack Frames

The Jenga of data!

Subroutine Challenges

- Need to pass data...
 - as input from the calling program
 - as outputted results
 - must be done for *any* number of inputs – *even if it exceeds the available number of registers*
- Need local variables
 - space needs to be allocated
 - cannot overwrite caller data



11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

31

Subroutine Challenges

- Subroutines may be called from anywhere
 - used in different locations within a program
 - often compiled *separately* – outside the caller
 - generally impossible to determine which registers may be safely used by a subroutine

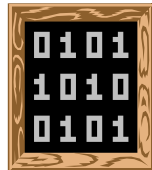
11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

32

Stack Frames

- So, many compilers use *Stack Frames*
- Rather than pass all parameters through registers, the system *stack is used*
- The stack is also used to store local variables



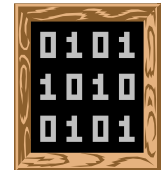
11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

33

Stack Frames

- This approach is used by most compilers given most processors are *very limited in their registers*
- Easy concept, but very difficult to program (*at first*)



11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

34

Stack Frame Contents

- Contains all the information needed by subroutine
- Includes:
 - calling program's return address
 - input parameters to the subroutine
 - the subroutine's local variables
 - space to backup the caller's register file

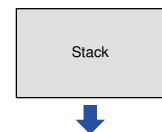
11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

35

Nesting is Possible

- Because the stack is LIFO (last in first out), subroutines can call subroutines
- This approach allows recursion and all the features found in high-level programming languages



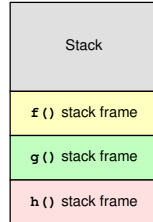
11/12/2017

Sacramento State - Cook - CS5c 35 - Fall 2017

36

Nesting is Possible

- For example, subroutine **f()** calls **g()** which then calls **h()**
- Since the stack is used, the stack frames follow the LIFO behavior



11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

37

How it Starts Up

- Caller
 - pushes the subroutine's arguments onto the stack
 - caller calls the subroutine
- Subroutine then...
 - uses the stack to backup registers
 - and "carve" out local variables

11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

38

How it Finishes

- Subroutine...
 - restores the original register values
 - removes the local variables from the stack
 - calls the processor "return" instruction
- Caller, then...
 - removes its arguments from the stack
 - handles the result – which can be passed either in a register or on the stack

11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

39

Calling Convention

- Different programming systems may arrange the stack frame differently
- A *calling convention* is used by a programming system (e.g. a language) to define *how* data will be passed
- In particular, it defines the structure of the stack frame and how data is returned

11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

40

Calling convention

- For example:
 - Is the first argument pushed first? Or last?
 - Is the result in a register? Or the stack?
- If all subroutines follow the same format
 - caller can use the same format for each
 - subroutines can also be created separately and linked together

11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

41

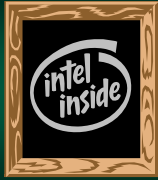
Stack Frame Size Varies

- The number of input arguments and local variables varies from subroutine to subroutine
- The arrangement of data within the stack frame also varies from compiler to compiler
- Stack frames is a *concept* – and it is used with various differences

11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

42

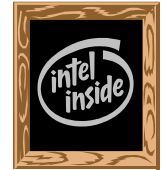


Stack Frames on the x64

Pretty much how its done on all processors

Stack Frames on the x64

- Stack frames on the x64 are accomplished pretty much the same way as other processors
- How it is done in real life is not simple – and is one of the hardest concepts to understand



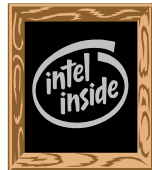
11/12/2017

Sacramento State - Cook - CSIS 35 - Fall 2017

44

Stack Frames on the x64

- On the x64, we will use the Base Pointer (RBP) to access elements in the stack frame
- This is a pointer register
- We will use it as an "anchor" in our stack frame



11/12/2017

Sacramento State - Cook - CSIS 35 - Fall 2017

45

Stack Frames on the x64

- As we build the stack frame, we will set RBP to fixed address in the stack frame
- Our parameters and local variables will be accessed by looking at memory *relative* to the RBP
- So, we will look *x* many bytes above and below the "anchor"

11/12/2017

Sacramento State - Cook - CSIS 35 - Fall 2017

46

Stack on the x64

- The stack base on the x64 is stored in high memory and grows downwards towards 0
- So, as the size of the stack increases, the stack pointer (RSP) will decrease in value



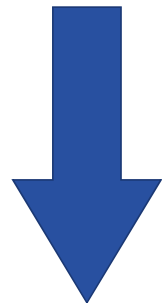
11/12/2017

Sacramento State - Cook - CSIS 35 - Fall 2017

47

Stack on the x64

- On a 64-bit system, it will decrease by increments of 8 bytes
- So each of our values (local variables and parameters) will be offsets of 8



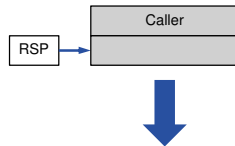
11/12/2017

Sacramento State - Cook - CSIS 35 - Fall 2017

48

Structure of a Stack Frame

- In this example, the stack is growing downward
- The stack pointer (RSP) is always on the bottom of the stack frame



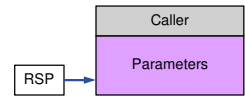
11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

49

1. Push Parameters

- Caller starts by pushing each of the parameters onto the stack
- The order in which the parameters are pushed is defined in the *calling convention*



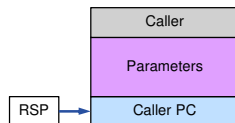
11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

50

2. Call the subroutine

- The caller then uses the *Call Instruction* to pass control to the subroutine
- The processor pushes the PC (program counter) on the stack
- Subroutine now runs



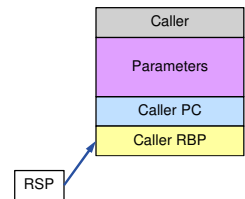
11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

51

3. Save the Old Base Pointer

- We need to set the Base Pointer (RBP)
- So, the old version needs to be saved (so it can be restored)
- Old value is pushed



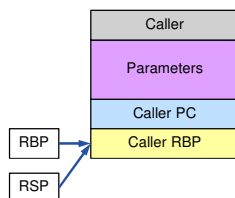
11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

52

4. Set the Base Pointer

- Then, it sets the Base Pointer (RBP) to the current stack pointer (RSP) address
- RBP is an "anchor" that we will use



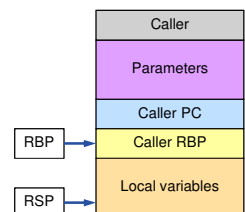
11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

53

5. Add Local Variables

- Subroutine now creates local variables on the stack
- Their initial values can be simply pushed
- *Note:* many compilers use a trick here that we will discuss later



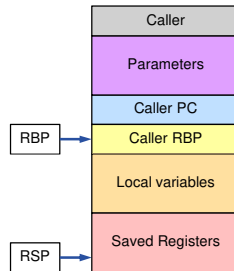
11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

54

6. Backup Registers

- Finally, the subroutine saves all the registers (that will change) on the stack
- It will restore them at the end



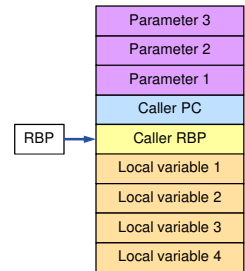
11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

55

Parameters & Local Variables

- Now, RBP is set to an address between the parameters and the local variables
- We can use *offsets* from RBP to access each



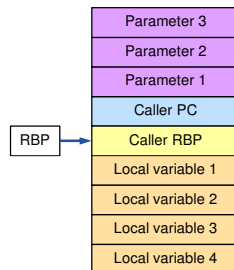
11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

56

Size of Stack Values – 64 bit

- x64 stack grows downward, so...
- Variables will have a negative offset
- Parameters will have a positive offset



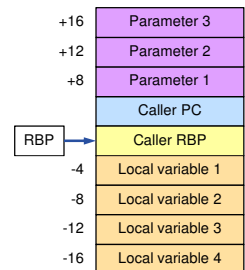
11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

57

Size of Stack Values – 32 bit

- On a 32-bit system, each word is 4 bytes
- So, each value on the stack is 4 bytes
- Offsets increase and decrease by 4



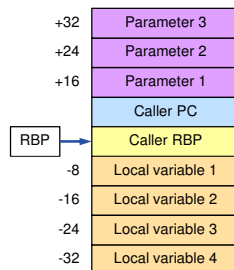
11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

58

Size of Stack Values – 64 bit

- On a 64-bit system, each word is 8 bytes
- So, each value on the stack is 8 bytes
- Offsets increase and decrease by 8

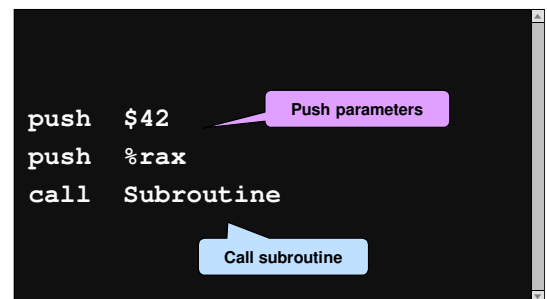


11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

59

Caller Example



11/12/2017

Sacramento State - Cook - CSc 35 - Fall 2017

60

Subroutine: Setup Example

```
push %rbp
mov  %rsp, %rbp
push $1
push $2
push %rax
push %rbx
```

Backup RBP

Set new RBP

Local variables

Backup registers

11/12/2017

Sacramento State - Cook - CS& 35 - Fall 2017

61

Subroutine Data Example

```
mov 16(%rbp), %rax
add 24(%rbp), %rax
mov %rax, -8(%rbp)
```

Offset from pointer

Parameter 1

Parameter 2

Local variable 1

11/12/2017

Sacramento State - Cook - CS& 35 - Fall 2017

62

Subroutine: Ending Example

```
pop %rbx
pop %rax
mov %rbp, %rsp
pop %rbp
ret
```

Restore registers

Set RSP to RBP
Effectively deletes all
local variables

Restore RBP

Only return when the stack is restored

11/12/2017

Sacramento State - Cook - CS& 35 - Fall 2017

63