Assembler Functions

COMP375 Computer Architecture and Organization
“Make everything as simple as possible, but not simpler.”

Albert Einstein
Assembler Assignment

• The second assembler programming assignment has been posted on Blackboard
• You are required to write four short program segments in assembler
• Upload your .cpp files to Blackboard by midnight on Wednesday, September 11, 2019
In Class Quiz

There will be a quiz

Today
Goals

• Understand how function calls are implemented in machine language
Stacks

- Many programming languages use stacks to pass parameters
- Many computer architectures have stack instructions to help implement these programming languages
- Most architectures have stack pointer register. The stack pointer always points to the top item on the stack.
Program Memory Organization

- Heap
- Stack
- Global data
- Program instructions
Program Memory Organization

- Heap
- Stack
- Global data
- Program instructions

Intel method
Pushing and Popping

• A **PUSH** copies the value of a register onto the top of the stack
  – Decrement the stack pointer
  – Store the register at the address pointed to by the stack pointer

• A **POP** remove the value on the top of stack and put it in a register
  – Load the value pointed to by the stack pointer into the register
  – Increment the stack pointer
Temporary Location to Save Stuff

• The Intel architecture only has four registers to do arithmetic
• Sometimes you need to use a register and they all have something valuable in them
• You can temporarily save the value by pushing it on the stack
• After using the register, you can restore it to its original contents by popping from the stack
Push Example

// preserve edx and ecx so previous code is not disrupted

push edx
push ecx

// do something using edx and ecx

pop ecx
pop edx

Note the order
Intel PUSHA and POPA

• The **PUSHA** instruction pushes the contents of all eight of the registers on the stack. (*push all*)
• The **POPA** instruction pops seven values from the stack and places the values in the registers (in reverse order of PUSHA)
• These instructions provide a quick way to save the state upon function entry
SAVE THE REGISTERS

• By convention, functions and methods should return to the calling program with the same values in the registers
• Methods usually save the register values on the stack at the start of the method and restore them just before returning
• Some OS have rules on where to save the registers while others allow the programmer to save them anywhere
Function Call Hardware

• All computers have machine language instructions to support function calls
• The level of hardware support varies with modern computers providing more support
Intel Call instruction

- The **CALL** instruction basically pushes the program counter on the stack and branches to a new location.
- There are many versions of the Intel **CALL** instruction to support different addressing modes and changes in privileges.
Intel RET instruction

• The **RET** or return instruction pops a value from the stack and places it in the program counter register.
• Since the program counter contains the address of the next instruction to execute, this has the effect of branching back to the calling program.
Steps to perform a function call

• Compute any equations used in the parameters, such as 
  \[ x = \text{func}(a + b); \]

• Push the parameter values on the stack

• Execute a call instruction to push the return address on the stack and start execution at the first address of the function
Upon function entry

- Store the contents of the registers
- Increase the stack pointer to reserve memory for the local variable
- Start executing the function code
Upon function exit

- Reduce the stack by the size of the local variable
- Pop the register values
- Execute the return instruction to pop the address from the stack into the program counter
A call instruction does

A. A push and a jump
B. A pop and a jump
C. A push and a pop
D. A hop, skip and jump
Example Function Call

• Consider the function

```java
void thefunc(Widget b, int a ){
    int r = a;
}
```

• that is called by the main program

```java
int x = 5;
Widget y = new Widget();
thefunc( y, x );
```

• The Widget y is passed by reference. The int x is passed by value.
Stack for Call

- push x

```
5 (value of x)
```

...
Stack for Call

• push x
• push address of y

<table>
<thead>
<tr>
<th>5 (value of x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>address of Widget y</td>
</tr>
</tbody>
</table>
Stack for Call

- push x
- push address of y
- call thefunc
Stack Use by Function

- push x
- push address of y
- call thefunc
- increment stack

<table>
<thead>
<tr>
<th></th>
<th>5 (value of x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of Widget y</td>
<td></td>
</tr>
<tr>
<td>Return address</td>
<td></td>
</tr>
<tr>
<td>Local variable r</td>
<td></td>
</tr>
</tbody>
</table>

[Diagram showing stack usage]
Stack for Call

- push x
- push address of y
- call thefunc
- increment stack
- mov eax,12[esp] // param a
Stack for Return

- push x
- push address of y
- call thefunc
- increment stack
- mov eax, 12[esp]  // param a
- decrement stack
Stack for Return

- push x
- push address of y
- call thefunc
- increment stack
- mov eax,12[esp] // param a
- decrement stack
- return
Cleanup Stack

- push x
- push address of y
- call thefunc
- increment stack
- mov eax,12[esp] // param a
- decrement stack
- return
- decrement stack by 2
Stack Overflow Attack

• A common security attack is to cause a program to overflow the stack
• If the program stores a value into ar[4], it will right in the data past ar, the return address
• Instructions might be loaded in the rest of the stack
Stack Protection

• Good programs should check all parameters to ensure values are within range
• Some processors prohibit instructions from being fetched from the stack
Which memory segments can be made read-only?

A. Program Instruction
B. Global Data
C. Stack
D. Heap
020 sub1 whatever
...
030 ret
...
100 call sub1
102 something

<table>
<thead>
<tr>
<th>Stack pointer</th>
<th>504</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter</td>
<td>100</td>
</tr>
</tbody>
</table>
020 sub1 *whatever*

...  

030 ret

...  

100 call sub1

102 *something*
020 sub1 whatever

... 508

030 ret 504

... SP → 500 102

100 call sub1

102 something

<table>
<thead>
<tr>
<th>Stack pointer</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter</td>
<td>022</td>
</tr>
</tbody>
</table>
020 sub1 \textit{whatever}  
...  
030 ret  
...  
100 call sub1  
102 \textit{something}  

\begin{tabular}{|c|c|}
\hline
Stack pointer & 504 \\
Program Counter & 102 \\
\hline
\end{tabular}
Passing Parameters

• Pass by value parameters can be pushed on the stack before calling the function
• Parameters are usually pushed in a right to left order. The left most parameter is then on top.
• The function can access them using an offset from the stack pointer
• The stack must be popped or incremented upon return from the function
sub1( x, y )

020 sub1 mov eax, 4[esp]
  ...
030 ret
  ...
100 mov eax, y  
104 push eax
106 mov eax, x
10A push eax
10C call sub1
110 add esp, 8

SP → 50C
  508
  504
  500

Stack pointer 50C
Program Counter 100

x 17
y 43
sub1( x, y )

020 sub1  mov  eax,4[esp]
      ...  
030  ret  
      ...  
100  mov  eax, y
104  push  eax
106  mov  eax, x
10A  push  eax
10C  call  sub1
110  add  esp,8

<table>
<thead>
<tr>
<th>Stack pointer</th>
<th>50C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter</td>
<td>104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>43</td>
</tr>
</tbody>
</table>
sub1( x, y )

020 sub1 mov eax,4[esp]
... 50C
030 ret SP → 508
... 504
100 mov eax, y 500
104 push eax
106 mov eax, x
10A push eax
10C call sub1
110 add esp,8

Stack pointer 508
Program Counter 106

x 17
y 43
sub1( x, y )

020 sub1 mov eax,4[esp]

... 50C

030 ret 508 43

... SP →

100 mov eax, y 504

104 push eax 500

106 mov eax, x 50C

10A push eax 508

10C call sub1 50A

110 add esp,8 Program Counter 10A
sub1( x, y )

020 sub1 mov eax,4[esp]
   ...
030 ret
   ...
100 mov eax, y
104 push eax
106 mov eax, x
10A push eax
10C call sub1
110 add esp,8

SP →

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>17</td>
</tr>
</tbody>
</table>

Stack pointer 504
Program Counter 10C
sub1( x, y )

020 sub1  mov  eax,4[esp]
            ...

030    ret
            ...

100    mov  eax, y
104    push eax
106    mov  eax, x
10A    push eax
10C    call  sub1
110    add   esp,8

Stack pointer  500
Program Counter 020

| x  | 17 |
| y  | 43 |

SP → 500

50C
508
504
sub1(x, y)

020 sub1 mov eax, 4[esp]

... 50C

030 ret 508

... 504

100 mov eax, y 17

104 push eax 43

106 mov eax, x

10A push eax

10C call sub1

110 add esp, 8

Stack pointer 500

Program Counter 024
sub1(x, y)

020 sub1 mov eax, 4[esp]
030 ret
100 mov eax, y
104 push eax
106 mov eax, x
10A push eax
10C call sub1
110 add esp, 8

Stack pointer 504
Program Counter 110
sub1( x, y )

020 sub1 mov  eax,4[esp]  
    ...  
    SP → 50C
    50C  508  43
    508  504  17
    504  500 110

030 ret
    ...

100 mov  eax, y

104 push eax

106 mov  eax, x

10A push eax

10C call sub1

110 add  esp,8

Stack pointer  50C
Program Counter 112
Returning Function Results

• Simple return types *(i.e. int, char, address, etc.)* are returned in the eax register

• For more complex data types *(i.e. objects, arrays)* the return value is in memory and the eax register contains the address of the returned value
Which is *NOT* true

The Stack pointer

A. is the ESP register
B. is kept in RAM
C. can be added or subtracted
D. changed by pop and push
Local Variables

- Local variables (sometimes called automatic variable) are those allocated within a function
- Local variable are allocated on the stack
- When a function returns, the stack space is available for other functions
Example

```c
void sub1( int x ) {
    int  a;
    int  b;
    ...
    return;
}

int main( ) {
    int x = 17;
    sub1( x );
}
```
```assembly
sub1(x)

020 sub1 mov eax, -4[esp]

... SP → 510

030 ret 50C

... 508

100 mov eax, x 504

104 push eax 500

106 call sub1

10A add esp, 4
```

Stack pointer | 510
---|---
Program Counter | 100
sub1( x )

020 sub1 mov eax,-4[esp]
     ...

030  ret
     ...

100 mov eax, x

104 push eax

106 call sub1

10A add esp,4

<table>
<thead>
<tr>
<th>Stack pointer</th>
<th>510</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter</td>
<td>104</td>
</tr>
</tbody>
</table>
```
020 sub1  mov  eax,-4[esp]
  ...
030  ret
  ...
100  mov  eax, x
104  push eax
106  call sub1
10A  add  esp,4
```
```assembly
020 sub1  mov  eax,-4[esp]
       ...                     510
030   ret
       ...                     50C  17
100   mov  eax, x
104   push eax               508
106   call  sub1
10A   add  esp,4
```

```
<table>
<thead>
<tr>
<th>Stack pointer</th>
<th>508</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter</td>
<td>020</td>
</tr>
</tbody>
</table>
```
020 sub1 mov eax,-4[esp]

... 510

030 ret 50C 17

... SP → 508 10A

100 mov eax, x 504 a

104 push eax 500 b

106 call sub1

10A add esp,4

Stack pointer 508

Program Counter 023

sub1( x )
sub1( x )

020 sub1 mov eax, -4[esp]

... 510

030 ret 50C

... 17

100 mov eax, x 508

104 push eax 504

106 call sub1 a

10A add esp, 4 500

b

Stack pointer | 50C
Program Counter | 10A
020 sub1  mov  eax,-4[esp]  
         ...                     
030   ret                      
         ...                     
100   mov  eax, x              
104   push eax                 
106   call  sub1               
10A   add  esp,4               

sub1( x )
Moore’s Law

- Increased density of components on chip
- Gordon Moore – co-founder of Intel
- **Number of transistors on a chip doubles every 18 months**
  - Cost of a chip has remained almost unchanged
  - Higher packing density means shorter electrical paths, giving higher performance
  - Reduced power and cooling requirements
  - Fewer interconnections increases reliability