Assembler Functions

COMP375 Computer Architecture and Organization

Goals

• Understand how function calls are implemented.

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

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

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

Push Example

```assembly
// preserve edx and ecx so previous code is not disrupted
push edx
push ecx
// do something using edx and ecx
pop ecx
pop edx
```

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.
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.

Example Function Call

- Consider the function
  ```c
  void thefunc( float &b, int a ){
      int r = a;
  }
  ```
- that is called by the main program
  ```c
  int x = 5;
  float y = 7.0;
  float *w = &y;
  thefunc( w, x );
  ```

Stack for Call

- push x
Stack for Call
- push x
- push w

Stack Use by Function
- push x
- push w
- call thefunc
- increment stack

Stack for Call
- push x
- push w
- call thefunc
- increment stack
- get R1,12[sp] // param a
Stack for Return
• push x
• push w
• call thefunc
• increment stack
• get R1,12[sp] // param a
• decrement stack

Stack for Return
• push x
• push w
• call thefunc
• increment stack
• get R1,12[sp] // param a
• decrement stack
• return

Cleanup Stack
• push x
• push w
• call thefunc
• increment stack
• get R1,12[sp] // 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.
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.

```
020 sub1 whatever
  ... SP → 504
030   ret          Stack pointer  504
  ...
100  call sub1     Program Counter 102
102  something
```

```
sub1( x, y )
020 sub1 mov    eax, 4[esp]
  ... SP → 50C
030   ret         50C
  ...
100  mov    eax, y
104  push    eax
106  mov    eax, x
10A  push    eax
10C  call    sub1 Stack pointer  50C
110  add    esp, 8 Program Counter 100
```

```
sub1( x, y )
020 sub1 mov    eax, 4[esp]
  ... SP → 50C
030   ret         50C
  ...
100  mov    eax, y
104  push    eax
106  mov    eax, x
10A  push    eax
10C  call    sub1 Stack pointer  50C
110  add    esp, 8 Program Counter 100
```
### sub1( x, y )

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>020</td>
<td>sub1</td>
<td>50C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov eax, 4[esp]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>030</td>
<td>ret</td>
<td>508</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>SP →</td>
<td>504</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>mov eax, y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>push eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>mov eax, x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10A</td>
<td>push eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10C</td>
<td>call sub1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>add esp, 8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Stack pointer: 508

### Program Counter: 106

---

### sub1( x, y )

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>020</td>
<td>sub1</td>
<td>50C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov eax, 4[esp]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>030</td>
<td>ret</td>
<td>508</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>SP →</td>
<td>504</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>mov eax, y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>push eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>mov eax, x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10A</td>
<td>push eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10C</td>
<td>call sub1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>add esp, 8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Stack pointer: 504

### Program Counter: 10C

---

### sub1( x, y )

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>020</td>
<td>sub1</td>
<td>50C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov eax, 4[esp]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>030</td>
<td>ret</td>
<td>508</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>SP →</td>
<td>504</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>mov eax, y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>push eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>mov eax, x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10A</td>
<td>push eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10C</td>
<td>call sub1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>add esp, 8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Stack pointer: 504

### Program Counter: 10C

---

### sub1( x, y )

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>020</td>
<td>sub1</td>
<td>50C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov eax, 4[esp]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>030</td>
<td>ret</td>
<td>508</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>SP →</td>
<td>504</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>mov eax, y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>push eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>mov eax, x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10A</td>
<td>push eax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10C</td>
<td>call sub1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>add esp, 8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Stack pointer: 500

### Program Counter: 20
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.
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 );
}
```

```
020 sub1   mov  eax,-4[esp]
           ...
030       ret
           ...
100       mov  eax, x
104       push eax
106       call sub1
10A       add esp,4
```

```assembly
Stack pointer 510
Program Counter 100
```
Assembler Function Calls

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

...                              510
030   ret              SP → 50C  17
...                              508
100   mov   eax, x      504
104   push   eax       500
106   call   sub1
10A   add   esp,4

Stack pointer 50C
Program Counter 106
```

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

...                              510
030   ret              SP → 50C  17
...                              508
100   mov   eax, x      504
104   push   eax       500
106   call   sub1
10A   add   esp,4

Stack pointer 508
Program Counter 020
```

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

...                              510
030   ret              SP → 50C  17
...                              508
100   mov   eax, x      504
104   push   eax       500
106   call   sub1
10A   add   esp,4

Stack pointer 50C
Program Counter 10A
```

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

...                              510
030   ret              SP → 50C  17
...                              508
100   mov   eax, x      504
104   push   eax       500
106   call   sub1
10A   add   esp,4

Stack pointer 508
Program Counter 106
```

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

...                              510
030   ret              SP → 50C  17
...                              508
100   mov   eax, x      504
104   push   eax       500
106   call   sub1
10A   add   esp,4

Stack pointer 50C
Program Counter 10A
```
### sub1(x)

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>020</td>
<td><code>sub1</code> mov eax,-4[esp]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>030</td>
<td>ret</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>mov eax, x</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>push eax</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>call sub1</td>
<td></td>
</tr>
<tr>
<td>10A</td>
<td>add esp,4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>510</td>
</tr>
<tr>
<td>PC</td>
<td>10C</td>
</tr>
</tbody>
</table>