Stack Organization

John Winans

1 Overview

A *stack* is an area of memory that is used to store data. Data is inserted into a stack using a *push* operation. Data is removed from a stack using a *pop* operation.

2 Direction of Growth

During the initialization of an application, a section of memory is reserved for the *program stack.* The starting address of the stack is loaded into the *stack pointer* register that is then used to keep track of the location of the top of the stack.

The section of memory reserved for a stack can be filled from the highest address to the lowest or from the lowest to the highest.

When a stack is filled from the highest address to the lowest, the stack is referred to as a descending stack. When data elements are pushed onto a descending stack, the stack pointer is decremented by the number of bytes pushed. When data elements are popped out of the stack, the stack pointer is incremented by the number of bytes popped.

When a stack is filled from the lowest address to the highest, the stack is referred to as an ascending stack. When data elements are pushed into an ascending stack, the stack pointer is incremented and when data elements are popped, it is decremented.

3 Full vs Empty

A stack can also be classified as being *full* or *empty.*

In an Empty stack, the stack pointer points to the next free (empty) location on the stack. When pushing data into an *Empty* stack, the data is stored in the stack at the address in the stack pointer and then the stack pointer is adjusted to reflect its new location.

In a Full stack, the stack pointer points to the topmost item in the stack (the last item pushed into it.) When pushing data into a *Full* stack, the stack pointer is adjusted to reflect its new location and then the data is stored in the stack at the address in the stack pointer.

4 Conclusion

This allows for four different types of stack organizations:

- Full Descending
- Full Ascending

- Empty Descending
- Empty Ascending

5 An Example

If a *Full Descending* stack is employed on a *big-endian* machine with the following memory contents:

```
000000e0  c9 b9 6e 41 b2 85 af 8d  c8 71 dd 31 12 2b 2c f9  |..nA.....q.1.+,.|
000000f0  78 31 b8 c2 a7 d0 de 27  47 24 5b b3 e0 e9 3b e2  |x1.....'G$[...;.|
00000100  91 de b5 9e e1 90 d4 b5  06 d7 66 ad 79 dc 17 fb  |..........f.y...|
```

and the SP (stack pointer) register is set to `0x00000100` then a *push* of the 32-bit value `0x01021a04` onto the stack would be performed like this:

```
SP ← SP - 4                // pre-decrement using operand size SP=0x000000fc
mem32(SP) ← 0x01021a04     // store the 32-bit value into memory
```

Resulting in changing the contents of the four bytes at address `0x000000fc` to this:

```
000000e0  c9 b9 6e 41 b2 85 af 8d  c8 71 dd 31 12 2b 2c f9  |..nA.....q.1.+,.|
000000f0  78 31 b8 c2 a7 d0 de 27  47 24 5b b3 01 02 1a 04  |x1.....'G$[.....|
00000100  91 de b5 9e e1 90 d4 b5  06 d7 66 ad 79 dc 17 fb  |..........f.y...|
```

... and leaving the value `0x000000fc` in the SP register.

If we then *pop* 3, 32-bit values out of the stack and into registers A, B, and C:

```
A ← mem32(SP)    // load from memory address in SP register
SP ← SP + 4      // post-increment using operand size SP=0x00000100
B ← mem32(SP)    // load from memory address in SP register
SP ← SP + 4      // post-increment using operand size SP=0x00000104
C ← mem32(SP)    // load from memory address in SP register
SP ← SP + 4      // post-increment using operand size SP=0x00000108
```

... then the SP register will contain `0x00000108`, A=`0x01021a04`, B=`0x91deb59e`, C=`0xe190d4b5` and the contents of the memory will not have been changed by the push operations.