Stack Organization

John Winans

1 Overview

A stack is an area of memory that is used to store data using what is called a LIFO (Last In First Out) data structure. Data is inserted into a stack by using a push operation and removed by using a pop operation.

2 Ascending/Descending (Direction of Growth)

During the initialization of an application, a section of memory is reserved for the program’s call stack (or simply the program’s stack). The starting address of the stack is loaded into a 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 grow (or “be filled with data”) from the highest address to the lowest or from the lowest address to the highest.\(^1\)

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 a descending 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 by the number of bytes pushed and when data elements are popped out of an ascending stack, it is decremented by the number of bytes popped.

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 first 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 first adjusted to reflect its new location and then the data is stored in the stack at the address in the stack pointer.

In other words, this answers the question “is the stack pointer pointing at a location where the last value that has been pushed into the stack or does it point at a location where next pushed value would go?”

\(^1\)Note that the direction of a stack’s growth has nothing to do with endianness.
4 Conclusion

This allows for four different types of stack organizations:

- Full Descending
- Empty Descending
- Full Ascending
- Empty Ascending

Most microprocessors use a Full, Descending stack starting at the highest memory address.

5 Examples

This section presents an example of push and pop operations for each of the four types of stack implementation.

The reader should observe that each type of push instruction varies only in that the SP register is adjusted either before (full) or after (empty) data is stored into the stack memory and such adjustment is either done by adding (ascending) or subtracting (descending) the number of bytes representing the size of the value being pushed.

The reader should also observe that each type of pop instruction varies only in that the SP register is adjusted either before (empty) or after (full) data is read from the stack memory and such adjustment is performed by either adding (descending) or subtracting (ascending) the number of bytes representing the size of the value being popped.

5.1 Descending Stacks

This section provides a detailed discussion of push and pop operations on descending stacks.

5.1.1 Full Descending Push

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
000000f0 78 31 b8 c2 a7 d0 de 27 47 24 5b b3 e0 e9 3b e2
00000100 91 de b5 9e e1 90 d4 b5 06 d7 66 ad 79 dc 17 fb
```

and the SP (stack pointer) register is currently 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=0x00000fc
mem32(SP) ← 0x01021a04  // store the 32-bit value into memory
```

Resulting in changing the contents of the four bytes at address 0x00000fc to this:

```
000000e0 c9 b9 6e 41 b2 85 af 8d c8 71 dd 31 12 2b 2c f9
000000f0 78 31 b8 c2 a7 d0 de 27 47 24 5b b3 01 02 1a 04
00000100 91 de b5 9e e1 90 d4 b5 06 d7 66 ad 79 dc 17 fb
```
... and SP=0x000000fc.

Note that if this same example were run on a little-endian machine, the only difference would be that the four bytes starting at memory at address 0x000000fc would have been set to 04 1a 02 01 (rather than 01 02 1a 04) like this:

```
000000f0 78 31 b8 c2 a7 d0 de 27 47 24 5b b3 04 1a 02 01
```

The SP register will have started the same as before (at 0x00000100) and finished the same as before as well (at 0x000000fc.)

### 5.1.2 Full Descending Pop

If the SP register contains the value 0x000000fc and the memory contains:

```
000000e0 c9 b9 6e 41 b2 85 af 8d c8 71 dd 31 12 2b 2c f9
000000f0 78 31 b8 c2 a7 d0 de 27 47 24 5b b3 01 02 1a 04
00000100 91 de b5 9e e1 90 d4 b5 06 d7 66 79 dc 17 fb
```

... then if we pop three 32-bit values out of the stack into registers A, B, and C:

\[
\begin{align*}
A & \leftarrow \text{mem32}(\text{SP}) \quad \text{// load from memory address in SP register} \\
\text{SP} & \leftarrow \text{SP} + 4 \quad \text{// post-increment using operand size SP=0x00000100} \\
B & \leftarrow \text{mem32}(\text{SP}) \quad \text{// load from memory address in SP register} \\
\text{SP} & \leftarrow \text{SP} + 4 \quad \text{// post-increment using operand size SP=0x00000104} \\
C & \leftarrow \text{mem32}(\text{SP}) \quad \text{// load from memory address in SP register} \\
\text{SP} & \leftarrow \text{SP} + 4 \quad \text{// post-increment using operand size SP=0x00000108}
\end{align*}
\]

... then the SP register will contain 0x00000108, \(A=0x01021a04\), \(B=0x91de5b9e\), \(C=0xe190d4b5\) and the contents of the memory will not have been changed by the pop operations.

Note that if this same example were run on a little-endian machine starting with the same register and memory contents as a big-endian example above then the three pop instructions would have resulted in \(\text{SP}=0x00000108, A=0x041a0201, B=0x9e5bde91, C=0xb5d490e1\) and, again, the contents of the memory will not have been changed by the pop operations.

### 5.1.3 Empty Descending Push

Note that the difference between a Full Descending and an Empty Descending stack is simply the order in which the operations are performed in the RTL (pseudocode) that describes when the SP register is decremented and when the values are stored into the stack.

If an Empty 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
000000f0 78 31 b8 c2 a7 d0 de 27 47 24 5b b3 e0 e9 3b e2
00000100 91 de b5 9e e1 90 d4 b5 06 d7 66 ad 79 dc 17 fb
```

and the SP (stack pointer) register is set to 0x000000fc then a push of the 32-bit value 0x01021a04 onto the stack would be performed like this:

```
\text{mem32}(\text{SP}) \leftarrow 0x01021a04 \quad \text{// store the 32-bit value into memory} \\
\text{SP} \leftarrow \text{SP} - 4 \quad \text{// post-decrement using operand size SP=0x000000fc}
```

Resulting in changing the contents of the four bytes at address 0x00000100 to this:
Note that if this same example were run on a little-endian machine, the only difference would be that the four bytes starting at memory at address 0x00000100 would have been set to 04 1a 02 01 (rather than 01 02 1a 04) like this:

00000100 04 1a 02 01 e1 90 d4 b5 06 d7 66 ad 79 dc 17 fb

The SP register will have started the same as on a big-endian machine (at 0x00000100) and finished the same (at 0x000000fc.)

5.1.4 Empty Descending Pop

As seen in the Empty Descending Push example, the only difference between a Full Descending and an Empty Descending stack is the order in which the operations are performed in the RTL (pseudocode).

If the SP register contains the value 0x000000fc and the memory contains:

000000e0 c9 b9 6e 41 b2 85 af 8d c8 71 dd 31 12 2b 2c f9
000000f0 78 31 b8 c2 a7 d0 de 27 47 24 5b b3 e0 e9 3b e2
00000100 01 02 1a 04 e1 90 d4 b5 06 d7 66 ad 79 dc 17 fb

... then if we pop three 32-bit values out of the stack into registers A, B, and C:

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

... then the SP register will contain 0x00000108, A=0x91deb59e, B=0xe190d4b5, C=0x06d766ad and the contents of the memory will not have been changed by the pop operations.

Note that if this same example were run on a little-endian machine starting with the same register and memory contents as a big-endian example above then the three pop instructions would have resulted in SP=0x00000108, A=0x9eb5de91, B=0xb5d490e1, C=0xad66d706 and, again, the contents of the memory will not have been changed by the pop operations.

5.2 Ascending Stacks

This section provides a detailed discussion of push and pop operations on ascending stacks.

5.2.1 Full Ascending Push

The only difference between a Full Ascending push and a Full Descending push is that the stack pointer is incremented rather than decremented in the RTL (pseudocode).

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

\[
\begin{align*}
\text{SP} & \leftarrow \text{SP} + 4 & \quad \text{\textit{// pre-increment} using operand size SP=0x00000104} \\
\text{mem32(\text{SP})} & \leftarrow 0x01021a04 & \quad \text{\textit{// store the 32-bit value into memory}}
\end{align*}
\]

Resulting in changing the contents of the four bytes at address 0x00000104 to this:

000000e0  c9 b9 6e 41 b2 85 af 8d  c8 71 dd 31 12 2b 2c f9
000000f0  78 31 b8 c2 a7 d0 de 27  47 24 5b b3 e0 e9 3b e2
00000100  91 de b5 9e e1 90 d4 b5  06 d7 66 ad 79 dc 17 fb

... and SP=0x00000104.

Note that if this same example were run on a little-endian machine, the only difference would be
that the four bytes starting at memory at address 0x00000104 would have been set to 04 1a 02 01
(rather than 01 02 1a 04) like this:

00000100  91 de b5 9e 04 1a 02 01  06 d7 66 ad 79 dc 17 fb

The SP register will have started the same as on the big-endian machine (at 0x00000100) and finished
the same as well (at 0x00000104.).

5.2.2 Full Ascending Pop

The only difference between a Full Ascending pop and a Full Descending pop is that the stack pointer is
decremented rather than incremented in the RTL (pseudocode).

If the SP register contains the value 0x00000100 and the memory contains:

000000e0  c9 b9 6e 41 b2 85 af 8d  c8 71 dd 31 12 2b 2c f9
000000f0  78 31 b8 c2 a7 d0 de 27  47 24 5b b3 e0 e9 3b e2
00000100  91 de b5 9e 01 02 1a 04  06 d7 66 ad 79 dc 17 fb

... then if we pop three 32-bit values out of the stack into registers A, B, and C:

\[
\begin{align*}
\text{A} & \leftarrow \text{mem32(\text{SP})} & \quad \text{\textit{// load from memory address in SP register}} \\
\text{SP} & \leftarrow \text{SP} - 4 & \quad \text{\textit{// post-increment} using operand size SP=0x000000fc} \\
\text{B} & \leftarrow \text{mem32(\text{SP})} & \quad \text{\textit{// load from memory address in SP register}} \\
\text{SP} & \leftarrow \text{SP} - 4 & \quad \text{\textit{// post-increment} using operand size SP=0x000000xf8} \\
\text{C} & \leftarrow \text{mem32(\text{SP})} & \quad \text{\textit{// load from memory address in SP register}} \\
\text{SP} & \leftarrow \text{SP} - 4 & \quad \text{\textit{// post-increment} using operand size SP=0x000000f4}
\end{align*}
\]

... then the SP register will contain 0x000000f4, A=0x91deb5de91, B=0x01021a04, C=0xe47245bb3 and
the contents of the memory will not have been changed by the pop operations.

Note that if this same example were run on a little-endian machine starting with the same register and
memory contents as a big-endian example above then the three pop instructions would have resulted
in SP=0x000000f4, A=0x9eb5de91, B=0x041a0201, C=0xb35b2447 and, again, the contents of the
memory will not have been changed by the pop operations.
5.2.3 Empty Ascending Push

The only difference between a Full Ascending push and an Empty Ascending push is the order in which the operations are performed in the RTL (pseudocode) that describe when the SP register is incremented and when the values are stored into the stack.

If an Empty Ascending 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
000000f0 78 31 b8 c2 a7 d0 de 27 47 24 5b b3 e0 e9 3b e2
00000100 91 de b5 9e e1 90 d4 b5 06 d7 66 ad 79 dc 17 fb

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:

\[ \text{mem32} (\text{SP}) \leftarrow 0x01021a04 \]
\[ \text{SP} \leftarrow \text{SP} + 4 \]

Resulting in changing the contents of the four bytes at address 0x00000104 to this:

00000100 01 02 1a 04 e1 90 d4 b5 06 d7 66 ad 79 dc 17 fb

Note that if this same example were run on a little-endian machine, the only difference would be that the four bytes starting at memory at address 0x00000100 would have been set to 04 1a 02 01 (rather than 01 02 1a 04) like this:

00001000 04 1a 02 01 e1 90 d4 b5 06 d7 66 ad 79 dc 17 fb

The SP register will have started the same as on the big-endian machine (at 0x00000100) and finished the same as well (at 0x00000104.)

5.2.4 Empty Ascending Pop

The only difference between an Empty Ascending pop and a Full Ascending pop is the order in which the operations are performed in the RTL (pseudocode) that describe when the SP register is decremented and when the values are stored into the stack.

If the SP register contains the value 0x000000fc and the memory contains:

000000e0 c9 b9 6e 41 b2 85 af 8d c8 71 dd 31 12 2b 2c f9
000000f0 78 31 b8 c2 a7 d0 de 27 47 24 5b b3 e0 e9 3b e2
00000100 91 de b5 9e e1 90 d4 b5 06 d7 66 ad 79 dc 17 fb

... then if we pop three 32-bit values out of the stack into registers A, B, and C:

\[ \text{SP} \leftarrow \text{SP} - 4 \]
\[ \text{A} \leftarrow \text{mem32} (\text{SP}) \]
\[ \text{SP} \leftarrow \text{SP} - 4 \]
\[ \text{B} \leftarrow \text{mem32} (\text{SP}) \]
\[ \text{SP} \leftarrow \text{SP} - 4 \]
\[ \text{C} \leftarrow \text{mem32} (\text{SP}) \]

Copyright © 2020,2022 John Winans. All Rights Reserved
~/NIU/courses/463/notes/stack/stack.tex
jwinans@niu.edu 2022-01-31 09:26:26 -0600 v2.0-1363-g5b81e212
... then the SP register will contain 0x000000f0, A=0x47245bb3, B=0xa7d0de27, C=0x7831b8c2 and the contents of the memory will not have been changed by the pop operations.

Note that if this same example were run on a little-endian machine starting with the same register and memory contents as a big-endian example above then the three pop instructions would have resulted in SP=0x000000f0, A=0xb35b2447, B=0x27ded0a7, C=0xc2b83178 and, again, the contents of the memory will not have been changed by the pop operations.