Experiment 5

Stack in ARM

Stack is a Last-In-First-Out (LIFO) data structure implemented using contiguous area of memory block. In real life stack is analogous to the pile of dinner plates found at cafeterias. To remove a plate from the pile, we take the plate on the top which is the same plate most recently added (inserted) to the pile by the dishwasher. To get the last plate from the pile, we must remove all the plates on top of it. Similarly, undo operation in different softwares and the back button in web browsers are also examples of stack. Only two operations are associated with a stack: insertion and deletion. If we view the stack as a linear array of elements, stack insertion and deletion operations are restricted to one end of the array. Thus, the only element that is directly accessible is the element at the top-of-stack (TOS). In the stack terminology, insert and delete operations are referred to as push and pop operations, respectively. Stack is used to store local variables, for passing additional arguments to subroutines when insufficient argument registers are available, for supporting nested routine calls, and for handling processor interrupts.

Stack Pointer (SP)

Stack is referenced by a single pointer called Stack Pointer (SP). When data is inserted or retrieved from the stack, this pointer is automatically adjusted to ensure that multiple insertions/retrievals do not erase or the old stacked data. In Cortex-M4 architecture, stack pointer is a register that points to the 32-bit data on the top of the stack. The Cortex-M4 contains two stack pointers (R13). They are banked so that only one is visible at a time. These two stack pointers are as follows:

1. *Main Stack Pointer (MSP)*: used when handling interrupts and optionally used during regular program execution.

2. *Process Stack Pointer (PSP)*: Used by user application code.

The lowest 2 bits of the stack pointers are always 0, which means they are always word aligned. Figure 5.1 represents the banked stack pointers.

Banked Stack Pointer for ARM Cortex-M4 Processor

Types of Stack

Stacks are highly flexible in the ARM architecture because their implementation is completely dependent on software. Since it is left to the software to implement a stack, different implementation choices result different types of stacks. There are two types of stack depending on how it grows.

1. Ascending Stack: When items are pushed on to the stack, the stack pointer is increasing.i.e., the stack grows towards higher address.

2. Descending Stack: When items are pushed on to the stack, the stack pointer is decreasing.i.e., the stack is growing towards lower address.

There are two types of stack depending on what stack pointer points to.

1. Empty Stack: Stack pointer points to the location in which the next item will be stored. A push will store the value, and increment the stack pointer.

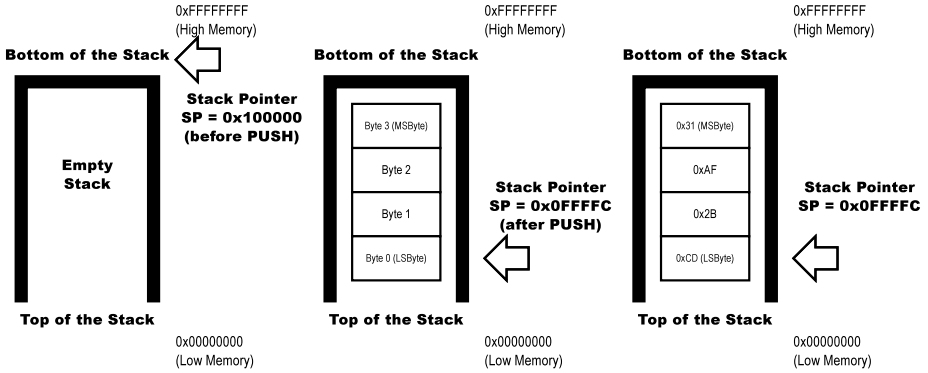
2. Full Stack: Stack pointer points to the location in which the last item was stored. A push will increment the stack pointer and store the value.

The above description suggests that there are four variations of a stack, representing all the combinations of ascending and descending full and empty stacks. The ARM Cortex-M processors implement the full-descending type of stack. Full descending stack adopts the convention where the stack starts from a high memory address (known as the *Bottom of the Stack*) and grows towards a lower memory address (where the *Top of the Stack* is located). The stack pointer will always point to the topmost item, or if there were no items in the stack, the Bottom of the Stack address. The Bottom of the stack address location is initialized at the beginning of the program and is not used to store any data.

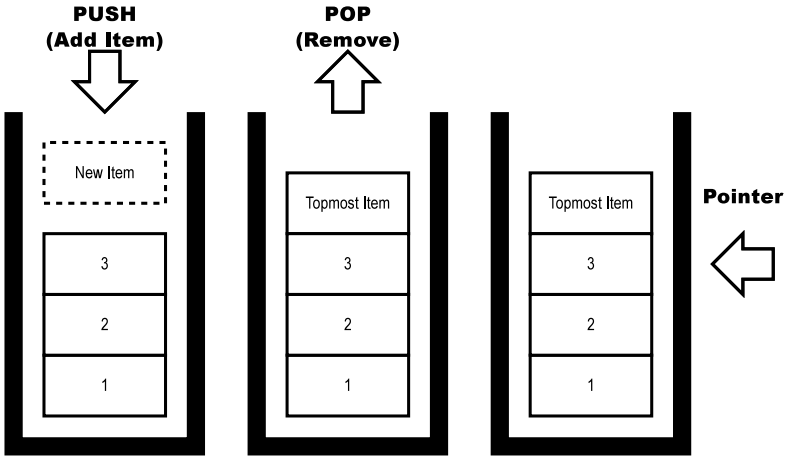
Full Descending stack is illustrated in Figure 5.2. Bottom of the stack was initialized to address 0x100000. Items are stored on the stack according to the endian-ness of the processor. For example, a 32-bit value stored into the stack would observe the processor endian-ness by storing the MSByte and LSByte appropriately based on the address location that the item uses. There is a 32-bit word item previously PUSHed onto the stack during program execution, located at address 0x0FFFFC (LSByte) to 0x0FFFFF (MSByte), following the little-endian processor format.

Adding, Removing and Accessing Items in a Stack

When data is added to the stack, it is said to be pushed onto the stack. When data is removed from the stack, it is said to be popped off the stack. These operations are done on the top of the stack. Figure 5.3 shows these operations.



Full Descending Stack for Little Endian Processor



Adding, Removing and Accessing Items from Stack

PUSH and POP Instructions

In a typical PUSH operation, the contents of one or more registers will be placed onto the stack. The memory address location where the first item is to be stored will be held in the stack pointer (SP) register. As the stack pointer points to the first empty location in the stack when data is pushed onto the stack, its value is decremented so that it points to the next free location. Thus the stack grows downward in memory. For example, PUSH R0 instruction pushes the contents of R0 register on the stack. It first decrements the SP by 4 and then stores the contents of R0 into the memory location pointed to by SP.

Example 5.1

```
PUSH {R0}           ; Push R0 onto the stack
PUSH {R0, R4–R7}    ; Push R0, R4, R5, R6, R7 onto the stack
PUSH {R2, LR}       ; Push R2 and link register onto the stack
```

In order to retrieve data from the stack we use POP instruction. Upon execution, first data is

moved to the register specified in the instruction and then stack pointer incremented by 4. The content is popped in the reverse order For example, POP R0 instruction retrieves the contents at the top of the stack and moves them to register R0, and then increments SP by 4.

Example 5.2

```
POP {R0}              ; Pop R0 from the stack
POP {R1, R4-R7}       ; Pop R1, R4, R5, R6, R7 from the stack
POP {R3, R5, PC}      ; Pop R3, R4 and PC from the stack, then branch to the
     new PC
```

As already mentioned, each PUSH/POP instruction transfers 4 bytes of data due to which stack pointer is decremented/incremented by 4 at a time or multiple of 4 if more than one register is pushed or popped. Registers are pushed on the stack in numerical order, with the lowest numbered register at the lowest address. On contrary, data is retrieved from the stack in reverse order, with data at highest address in highest numbered register.

Accessing Stack Using Multiple Load and Store Instructions

When executing thumb instructions in thumb mode, you can use the push and pop instructions which do not give you the freedom to use any register, it only uses R13 (SP) and you cannot save all the registers only a specific subset of them. In THUMB mode, PUSH instruction can access only Lo registers and the LR and POP instruction can only include Lo registers and PC. So, we can use the equivalent multiple load and store instructions LDMIA and STMDB which are more generic and all the registers can be accessed by them. Following example shows the use of these instructions.

Example 5.3

```
...
STMDB SP!, {R4-R7}        ; Push R4, R5, R6 and R7 onto the stack
...temporarily use R4,R5,R6,R7 for something else...
LDMIA R13!,{R4-R7}        ; Pop R4, R5, R6 and R7 from the stack
...
```

1. STM is store multiple you can save more than one register at a time, up to all of them in one instruction.
2. DB means decrement before, this is a downward moving stack from high addresses to lower addresses.
3. You can use SP or R13 here to indicate the stack pointer. This particular instruction is not limited to stack operations, can be used for other things.

4. The ! means update the SP register with the new address after it completes, here again STM can be used for non-stack operations so you might not want to change the base address register, leave the ! off in that case.

5. Then in the brackets list the registers you want to save, comma separated.

LDMIA is the reverse, LDM means load multiple. IA means increment after and the rest is the same as STM. Since stack manipulation maps naturally to the LDM/STM instructions, the ARM syntax has been given alternative names to make it easy to refer to the correct stack operation. In this case STMFD/LDMFD implements the full descending stack and is equivalent to STMDB/LDMIA.

Sequence for Using PUSH and POP Instructions

One important note regarding stack usage is that the number of registers PUSHed into the stack and the number of registers POPped from the stack within a routine must be balanced. For example, if we PUSHed R0, R1, R2 onto the stack, then they must be removed in reverse order of R2, R1, R0 if we intend to return the correct values to their original registers. If this were not done, then stack items would be retrieved to a different register, changing its value to that in another register. Obviously, if we POPped something from the stack before PUSHing an item in, it would corrupt the stack. Fortunately this can be accomplished easily on the ARM architecture by means of the LDM and STM instructions. A single STMFD / LDMFD pair can save and restore all affected registers in the correct sequence. The equivalent PUSH and POP pseudo-instructions are provided as a convenience as well. This is illustrated in the following code, where myfunc_1 is balanced while myfunc_2 and myfunc_3 would results in unexpected behavior or program crashes.

Example 5.4

```
; Balanced stack usage
myfunc_1
     PUSH {R4-R7}     ; Equivalent to STMFD SP!, {R4-R7}
     ...
     POP  {R4-R7}     ; Equivalent to LDMFD SP!, {R4-R7}

; Unbalanced stack usage
myfunc_2
     PUSH {R0}        ; Store one register in the stack
     ...
     POP {R0,R1}      ; Retrieve two registers from the stack

; Unbalanced stack usage
myfunc_3
     PUSH {R1,R2}     ; Store two register in the stack
```

```
        ...
        POP {R2}        ; Retrieve one register from the stack
```

Subroutine and Stack

In assembly language programming one subroutine can call another. In this case, the return address of the second call is also stored in the link register destroying the previous contents. Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. So, if the link register is pushed onto the stack at entry, additional subroutine calls can safely be made without causing the return address to be lost. If you do this, you can also return from a subroutine by popping the PC off the stack at exit, instead of popping LR and then moving that value into the PC. For example:

Example 5.5

```
subroutine  STMFD   SP!, {R5-R7,LR} ; Push work registers and lr
            ; code
            BL      somewhere_else
            ; code
            LDMFD   SP!, {R5-R7,PC} ; Pop work registers and pc
```

Examples

Here we discuss some examples to illustrate the stack usage.

Example 1

```
; Directives
        PRESERVE8
        THUMB

        AREA    MYCODE, CODE, READONLY

        ENTRY
      EXPORT __main

      ALIGN

; Define Procedures

subroutine  PROC        ; Using PROC and ENDP for procedures
      PUSH {R1, LR}     ; Save values in the stack
```

```
     MOV  R1, #8        ; Set initial value for the delay loop

delay
     SUBS R1, R1, #1
     BNE  delay

     POP  {R1, PC}      ; Pop out the saved value from the stack, check
                        ; the value in the R1 and if it is the saved value
     ENDP

;********** user main program *********;

__main


     MOV R0, #0x75
     MOV R3, #5
     PUSH{R0, R3}     ; Notice the stack address is 0x2000003F8
                      ; (Contains R1 then R3)
     MOV R0, #6
     MOV R3, #7
     POP{R0, R3}      ; Should be able to see the value in R0 = #0x75, R3 = #5

Loop

     ADD  R0, R0, #1
     CMP  R0, #0x80
     BNE  Loop

     MOV R1, #9       ; Prepare for function call
     BL subroutine

     MOV R3, #12

STOP

   B STOP
     END
```

Example 2

Execute the following program, understand its working and write the comments for each instruction.

```
    PRESERVE8
    THUMB

    AREA  myDATA, DATA, READWRITE
SQR   SPACE 7
Average DCD 0
```

```
SqrAvg  DCD 0

    ALIGN

    AREA  Program, CODE, READONLY

    ENTRY
    EXPORT   __main

;************ Define Procedures ***********;

; Calculates the square of integer array
SQUARE PROC
    STMDB SP!, {R0-R3, LR}
Loop
    LDRB  R3, [R1], #1
    MUL   R3,R3
    STRB  R3, [R2], #1
    SUBS  R0, R0, #1
    BGT   Loop
    LDMIA SP!, {R0-R3, PC}
    ENDP

; Calculates the average of input array
AVG   PROC
    PUSH  {R0-R4, LR}
    MOV   R4, R0
    EOR   R2,R2
ACC
    LDRB  R3, [R1], #1
    ADD   R2, R3
    SUBS  R0, R0, #1
    BGT   ACC
    UDIV  R5, R2, R4
    POP   {R0-R4, PC}
    ENDP

; Calculates the average of square of input array
AVGSQ PROC
    PUSH {R1, LR}

    BL SQUARE

    LDR R1, =SQR
    BL AVG

    POP {R1, PC}

    ENDP

;*********** User main program ************;
__main
```

```
    LDRB R0, Count
    LDR  R1, =Values
    LDR  R2, =SQR
    BL   SQUARE

    BL AVG

    LDR R3, =Average
    STR R5, [R3]

    BL AVGSQ
    LDR R3, =SqrAvg
    STR R5, [R3]

    B   STOP


Values DCB  7, 6, 5, 3, 9, 2, 1
Count  DCB  7


STOP
      B STOP

      ALIGN

      END
```

Exercise