Parameter Passing (11A)

Young Won Lim
1/9/21

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

ARM System-on-Chip Architecture, 2nd ed, Steve Furber

Introduction to ARM Cortex-M Microcontrollers
– Embedded Systems, Jonathan W. Valvano

Digital Design and Computer Architecture,
D. M. Harris and S. L. Harris

ARM assembler in Raspberry Pi
Roger Ferrer Ibáñez

https://thinkingeek.com/arm-assembler-raspberry-pi/

Local Variables

- dynamic allocation / release allows for resue of RAM
- limited scope of access (making it private) provides for data protection
- only the program that created the local variable can access it
- since an **interrupt** will save registers, the code is reentrant
- since **absolute addressing** is not used, the code is relocatable

- we can use **symbolic names** for the variables making it easier to understand
- the number of variables is only limited by the size of the stack
- because it is more general, it will be easier to add additional variables
-

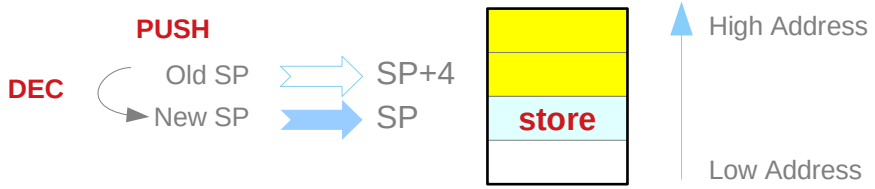
Global Variables

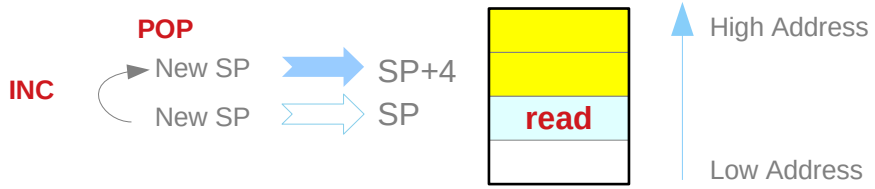
```
void MyFunction (void) {
      static uint32_t count = 0;
      count++;
}


static int32_t myPrivateGlobalVariable;          // accessible by this file only

void static MyPrivateFunction (void) {
}


const int16_t Slope=21;
const uint8_t SinTable[8] = {0, 50, 98, 142, 180, 212, 236, 250};
```

LIFO Stack

- Program segments should have an matching number of pushes and pops
- Stack accesses (push or pop) should not be performed outside the allocation area
- Stack reads and writes should not be performed within the free area
- Stack push should first decrement SP by 4, then store the data
- Stack pop should first read the data, then increment SP by 4

Full Top
Descending Stack





Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

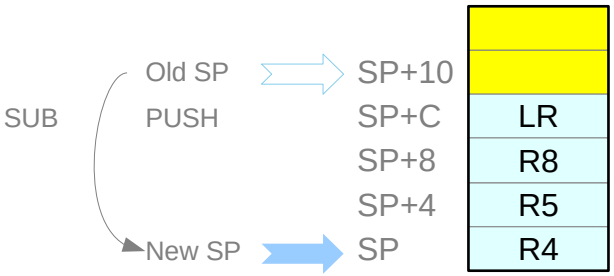
DEC

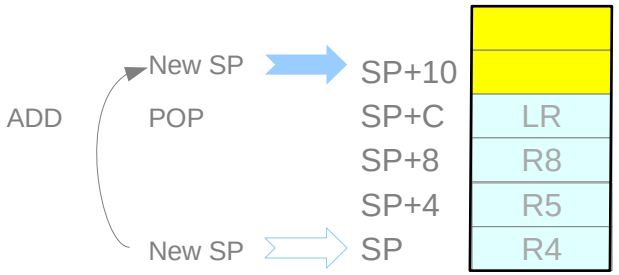
LIFO Stack

```
LDR     R0, [SP, #4]      ; R0 = the next to the top bytes

SUB     R1, SP, #8        ; R1 points to the free area
STR     R0, [R1]          ; Store contents of R0 into free area (*** illegal ***)
LDR     R2, [R1]          ; Read contents of free area into R2 (*** illegal ***)


PUSH    {R0, R1}          ; Store contents of R0, R1 onto the stack
```

Local variables on the stack

```
Func    PUSH    {R4, R5, R8, LR}        ; save registers as needed
        ; 1) allocate local variables
        ; 2) body of the function, access local variables
        ; 3) deallocate local variables
        POP     {R4, R5, R8, PC}
```





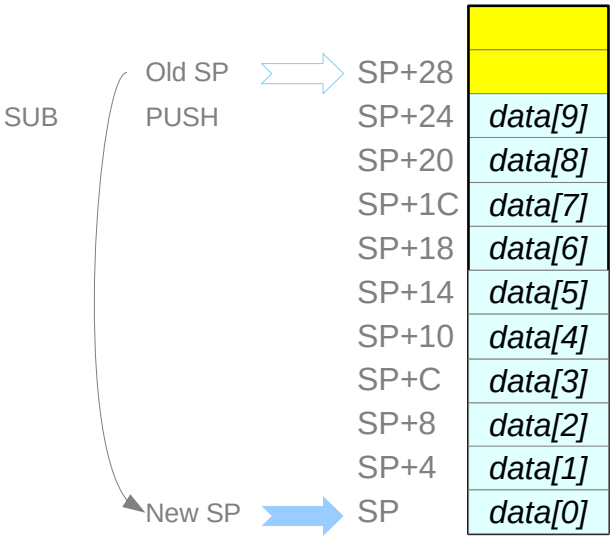
Initializing a local array

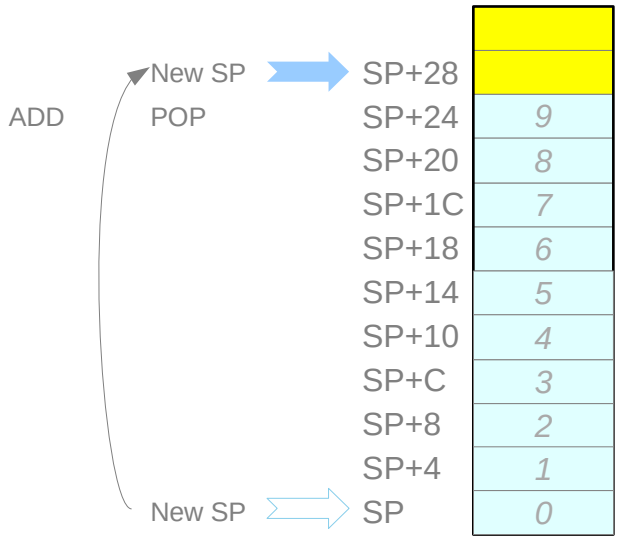
```
void Set(void) {
        uint32_t data[10];
        int i;
        for (i=0; i<10; i++) {
                data[i] = i;
        }
}
```

```
Set     SUB     SP, SP, #40     ; 1) allocate 10 words
        MOVS    R0, #0x00       ; 2) i = 0
        B       test            ; 2)
loop    LSL     R1, R0, #2      ; 2) 4*i
        STR     R0, [SP, R1]    ; 2) access
        ADDS    R0, R0, #1      ; 2) i++
test    CMP     R0, #10         ; 2)
        BLT     loop
        ADD     SP, SP, #40     ; 3) deallocate
        BX      LR
```

Local variables on the stack

```
LSL     R1, R0, #2      ; 2) 4*i
STR     R0, [SP, R1]    ; 2) access
ADDS    R0, R0, #1      ; 2) i++
```





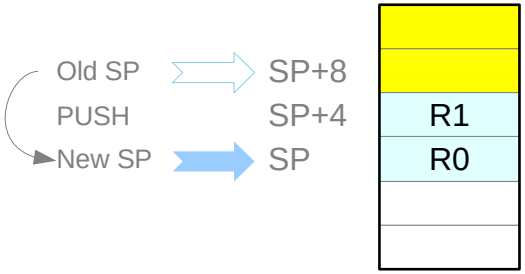
1. Binding

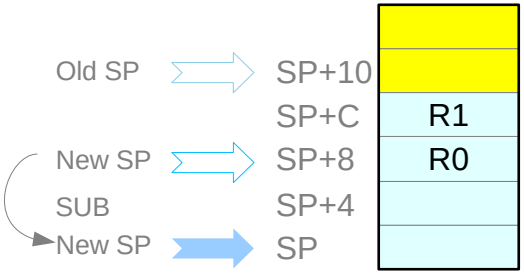
```
Sum     EQU     0           ; 32-bit local variable, stored on the stack
MOV     R0, #0
MOV     R1, #2
PUSH    {R0, R1}            ; allocate and initialize two 32-bit variables
SUB     SP, #8              ; allocate two 32-bit variables
LDR     R1, [SP, #sum]      ; R1 = sum
ADD     R1, R0              ; R1 = R1 + R0 (= sum + R0)
STR     R1, [SP, #sum]      ; sum = R0 + sum
LDR     R0, [SP, #sum]      ; R0 = sum
LSR     R0, R0, #2          ; R0 = R0 / 4
STR     R0, [SP, #sum]      ; sum = sum / 4
ADD     SP, #4              ; deallocate sum
POP     {R0, R1}            ; deallocate two 32-bit variables
```

```
;  sum = (sum + x) / 16
```

1. Binding, 2. Allocation

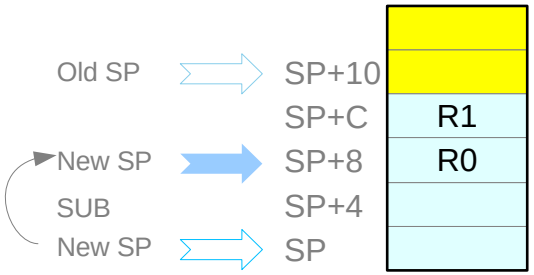
```
Sum     EQU     0           ; 32-bit local variable, stored on the stack

MOV     R0, #0
MOV     R1, #2
PUSH    {R0, R1}            ; allocate and initialize two 32-bit variables

SUB     SP, #8              ; allocate two 32-bit variables
```

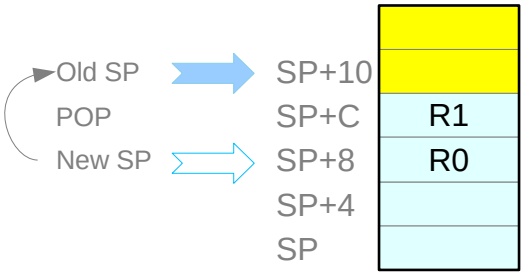




3. Access, 4. Deallocation

```
LDR     R1, [SP, #sum]      ; R1 = sum
ADD     R1, R0              ; R1 = R0 + sum
STR     R1, [SP, #sum]      ; sum = R0 + sum

LDR     R0, [SP, #sum]      ; R0 = sum
LSR     R0, R0, #2
STR     R0, [SP, #sum]      ; sum = sum / 4

ADD     SP, #8              ; deallocate sum
POP     {R0, R1}            ; deallocate two 32-bit variables
```





Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

Stack Frames

- Parameters
- Return address
- Saved registers
- Local variables

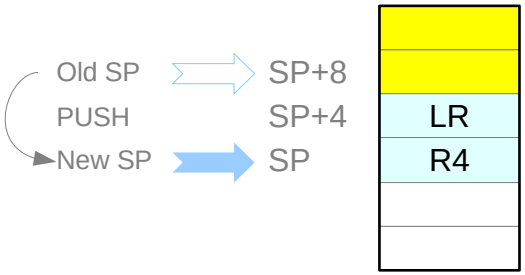
Stack Frames

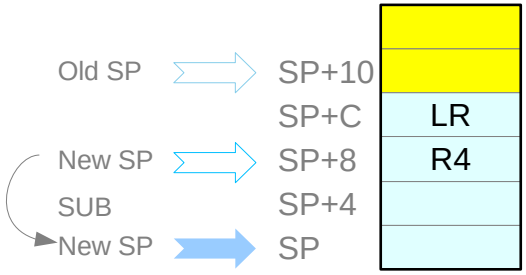
```
uint32_t calc(void) {
        uint32_t        sum, n;
        for (n=1000; n>0; n--) {
                sum = sum + n;
        }
        return sum;
}
```

Example **A** (1)

```
; *** binding phase ****
isum    EQU    0          ; 32-bit unsigned number
in      EQU    4          ; 32-bit unsigned number

; *** 1) allocation ***
calc    PUSH   {R4, LR}
        SUB    SP, #8          ; allocate n, sum
```

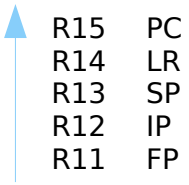
The **EQU** directive gives a **symbolic name** to a numeric constant, a register-relative value or a PC-relative value.

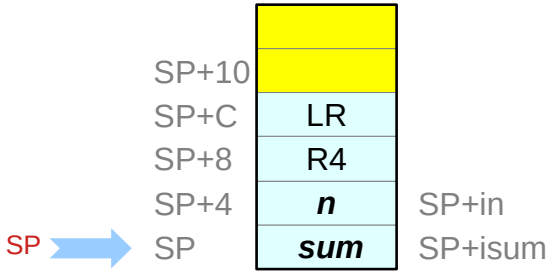




Example **A** (2)

```
; *** 2) access ***
        MOV     R0, #0
        STR     R0, [SP, #isum]     ; sum = 0           [SP, #0]
        MOV     R1, #1000
        STR     R1, [SP, #in]       ; n = 1000          [SP, #4]
loop    LDR     R1, [SP, #in]       ; R1 = n            [SP, #4]
        LDR     R0, [SP, #isum]     ; R0 = sum          [SP, #0]
        ADD     R0, R1              ; R0 = sum + n
        STR     R0, [SP, #isum]     ; sum = sum + n     [SP, #0]
        LDR     R1, [SP, #in]       ; R1 = n            [SP, #4]
        SUBS    R1, #1              ; n-1
        STR     R1, [SP, #in]       ; n = n – 1         [SP, #4]
        BNE     loop
```

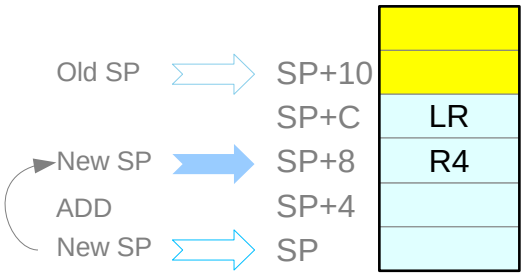


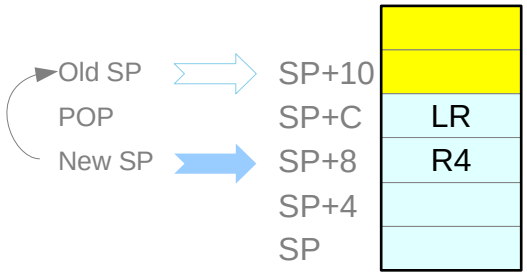


Example **A** (3)

```
; *** 3) deallocation ***
        ADD     SP, #8          ; deallocation
        POP     {R4, PC}        ; R0 = sum      PC ← LR
```

R15 PC
R14 LR
R13 SP
R12 IP
R11 FP

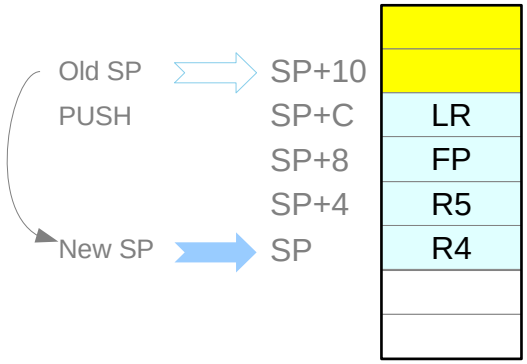


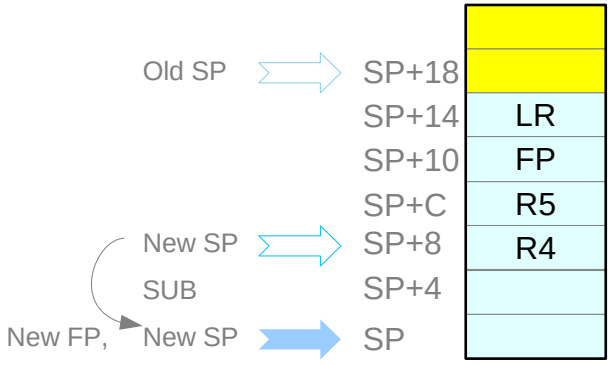


Example **B** (1)

```
; *** binding phase ****
isum    EQU    0        ; 32-bit unsigned number
in      EQU    4        ; 32-bit unsigned number

; *** 1) allocation ***
calc    PUSH   {R4, R5, FP, LR}
        SUB    SP, #8          ; allocate n, sum
        MOV    FP, SP          ; frame pointer
```

The **EQU** directive gives a **symbolic name** to a numeric constant, a register-relative value or a PC-relative value.

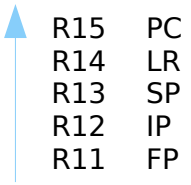


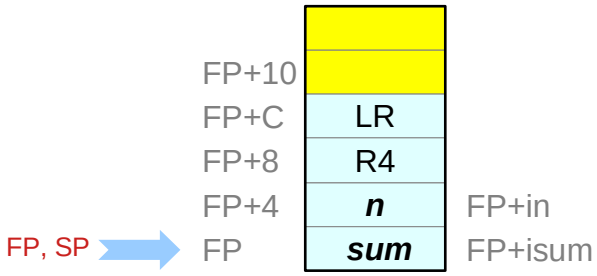


Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

Example **B** (2)

```
; *** 2) access ***
        MOV     R0, #0
        STR     R0, [FP, #isum]     ; sum = 0
        MOV     R1, #1000
        STR     R1, [FP, #in]       ; n = 1000
loop    LDR     R1, [FP, #in]       ; R1 = n
        LDR     R0, [FP, #isum]     ; R0 = sum
        ADD     R0, R1              ; R0 = sum + n
        STR     R0, [FP, #isum]     ; sum = sum + n
        LDR     R1, [FP, #in]       ; R1 = n
        SUBS    R1, #1              ; n-1
        STR     R1, [FP, #in]       ; n = n – 1
        BNE     loop
```

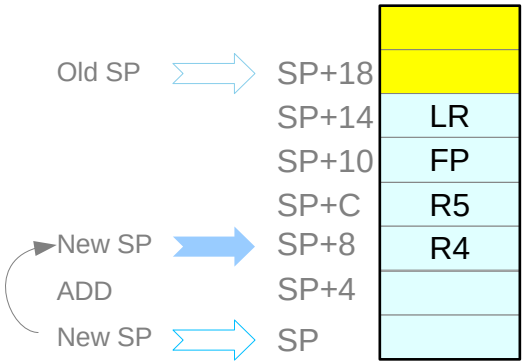


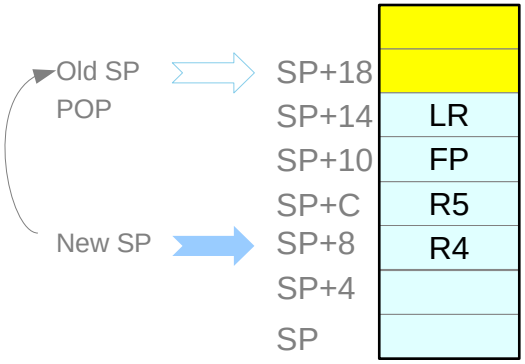


Example **B** (3)

```
; *** 3) deallocation ***
        ADD     SP, #8                  ; deallocation
        POP     {R4, R5, FP, PC}        ; R0 = sum          PC ← LR
```

R15 PC
R14 LR
R13 SP
R12 IP
R11 FP





Parameter Passing

Call by value	- safe, simple, good for small amounts of data
Call by reference	- parameter can be input or output, good for large amounts

Registers	- fast and simple
Stack	- flexible, good for large amounts of data
Global variables	- simple and poor style

Counting example – Call by value (1)

```
uint32_t next(uint32_t ang) {
        ang++;
        if (ang == 200) {
                ang = 0;
        }
        return ang;
}

void main (void) {
        uint32_t angle = 0;  // 0 to 199
        Stepper_Init();
        while (1) {
                Stepper_Step();
                angle = next(angle);
        }
}
```

Counting example – Call by value (2)

```
                                    ; R0 is value of the angle      R0 is an argument
next    ADD     R0, #1              ; add to copy
        CMP     R0, #200
        BNE     skip
        MOV     R0, #0              ; roll over
skip    BX      LR                  ; 0 to 199                      R0 as a return value

angle   EQU     0
main    SUB     SP, #4              ; allocate
        MOV     R0, #0
        STR     #0, [SP, #angle]
        BL      Stepper_Init
loop    BL      Stepper_Step
        LDR     R0, [SP, #angle]    ; R0 = angle                    R0 is a parameter
        BL      next
        STR     R0, [SP, #angle]    ; update                        R0 is a return value
        B       loop
```

Counting example – Call by reference (1)

```
void next(uint32_t *pt) {
        (*pt) = (*pt) +1;
        if ((*pt) == 200) {
                (*pt) = 0;
        }
}

void main (void) {
        uint32_t angle = 0;  // 0 to 199
        Stepper_Init();
        while (1) {
                Stepper_Step();
                next(&angle);
        }
}
```

Counting example – Call by reference (2)

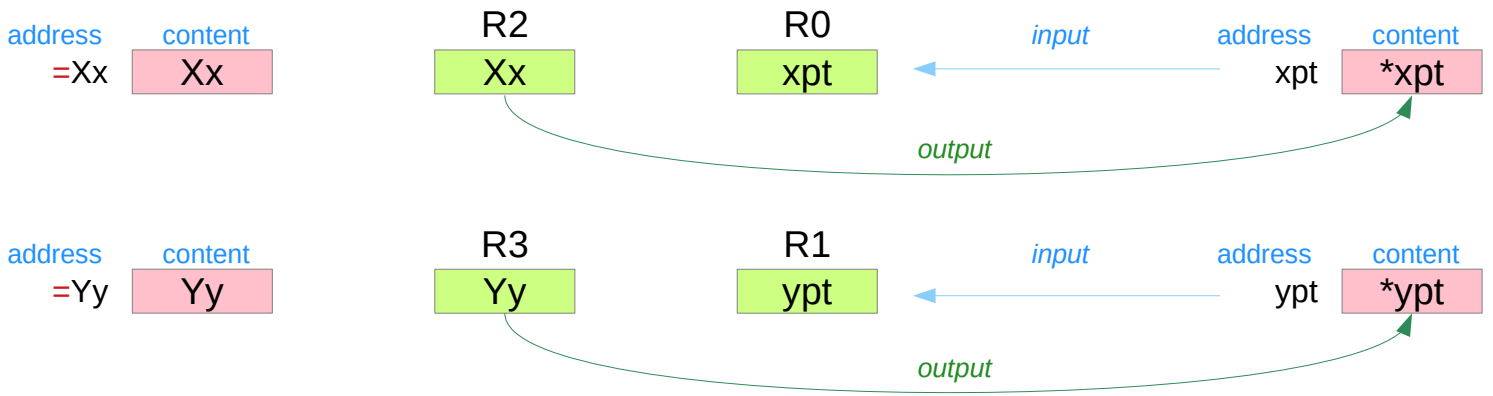
```
                                ; R0 is the address of the angle
next    LDR     R1, [R0]        ; *pt  (address)      R0 is an argument
        ADD     R1, #1          ; increment
        CMP     R1 #200
        BNE     skip
        MOV     R1, #0          ; roll over
skip    STR     R1, [R0]        ; update              R0 is not used as a return value
        BX      LR              ; 0 to 199

angle   EQU     0
main    SUB     SP, #4          ; allocate
        MOV     R0, #0
        STR     #0, [SP, #angle]
        BL      Stepper_Init
loop    BL      Stepper_Step
        LDR     R0, SP          ; R0 = &angle         R0 is a parameter
        BL      next
        B       loop
```

Multiple output example – Call by reference (1)

```
static int32_t Xx, Yy;                    // position

void where(    int32_t *xpt,
               int32_t *ypt     ) {
      (*xpt) = Xx;                        // return Xx
      (*ypt) = Yy;                        // return Yy
}


void func(void) {
      int32_t myX, myY;
      where(&myX, &myY);

      // do something based on myX, myY
}
```

Multiple output example – Call by reference (2)

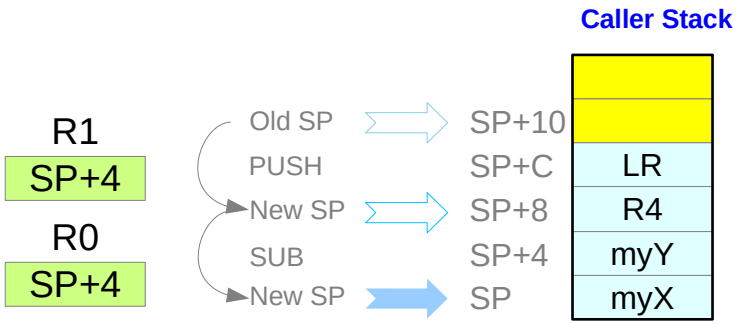
```
Xx      SPACE   4       ; allocated          private to where
Yy      SPACE   4       ; allocated
                        ; argument R0 holds the address xpt
                        ; argument R1 holds the address ypt
where   LDR     R2, =Xx ; R2 = &Xx
        LDR     R2, [R2]; R2 =    Xx         value of Xx
        STR     R2, [R0]; *xpt = Xx          pass data      R0 (= xpt)
        LDR     R3, =Yy ; R3 = &Yy
        LDR     R3, [R3]; R3 =    Yy         value of Yy
        STR     R3, [R1]; *ypt = Yy          pass data      R1 (= ypt)
        BX      LR
```

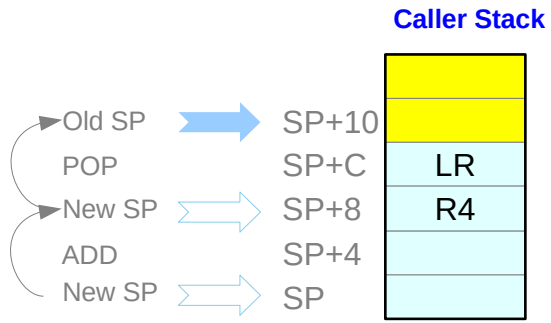


Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

Multiple output example – Call by reference (2)

```
iX      EQU     0               ; symbolic          32-bit
iY      EQU     4               ; symbolic
func    PUSH    {R4, LR}
        SUB     SP, #8          ; allocate
        ADD     R0, SP, #iX     ; R0 = &myX         myX is on the stack
        ADD     R1, SP, #iY     ; R1 = &myY         myY is on the stack
        BL      where           ; with R0, R1 arguments to where
                                ; do something base on myX, myY
        ADD     SP, #8          ; deallocate
        POP     {R4, PC}
```





Parameter Passing

```
; Reg R0 = Port A,      Reg R1 = Port B
; Reg R3 = PortC,      Reg R3 = Port D
getPorts        LDR     R0, =GPIO_PORTA_DATA_R      ; R0 = &GPIO_PORTA_DATA_R
                LDR     R0, [R0]                    ; value of Port A
                LDR     R1, =GPIO_PORTB_DATA_R      ; R1 = &GPIO_PORTB_DATA_R
                LDR     R1, [R1]                    ; value of Port B
                LDR     R2, =GPIO_PORTC_DATA_R      ; R2 = &GPIO_PORTC_DATA_R
                LDR     R2, [R2]                    ; value of Port C
                LDR     R3, =GPIO_PORTD_DATA_R      ; R3 = &GPIO_PORTD_DATA_R
                LDR     R3, [R3]                    ; value of Port D
                BX      LR

                *** calling sequence ***

                BL      getPorts
                ; Reg R0, R1, R2, R3 have four results
```

Parameter Passing – (1) using registers

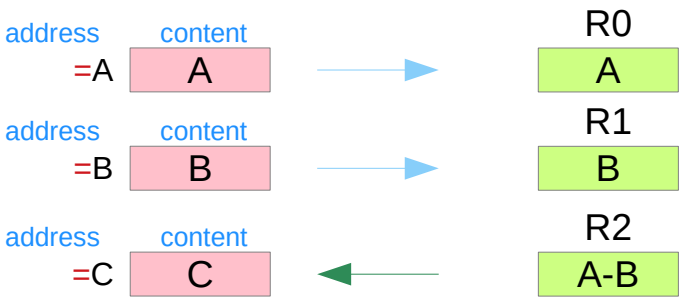
```
; Inputs:        R0, R1
; Outputs:       R2 = R0 – R1
Sub1    SUB     R2, R0, R1
        BX      LR
```

```
        LDR     R0, =A
        LDR     R0, [R0]
        LDR     R1, =B
        LDR     R1, [R1]
        BL      Sub1
        LDR     R0, =C
        STR     R2, [R0]
```

; R0 has the value of A

; R1 has the value of B

; C = A – B



Argument Register : R0, R1
Result Register : R2

Callee gets the parameters in R0, R1 and puts the result In R2

Caller prepares parameters A, B into R0, R1

Caller gets the result in R2 and stores it to C

Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

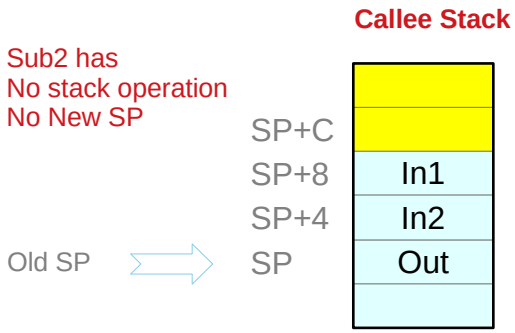
Parameter Passing – (2)-(a) using the stack

```
; Inputs:        In1 In2 on stack
; Outputs:       Out= In1 – In2 on stack
kIn1      EQU    8
kIn2      EQU    4
kOut      EQU    0
Sub2      LDR    R1, [SP, #kIn1]
          LDR    R0, [SP, #kIn2]
          SUB    R2, R0, R1
          STR    R2, [SP, #kOut]
          BX     LR
```

Argument In1 at SP+8
Argument In2 at SP+4
Result Out at SP

Caller prepares arguments
and allocate for an output
on the stack

Callee accesses arguments
and writes the result on the stack



Parameter Passing – (2)-(b) using the stack

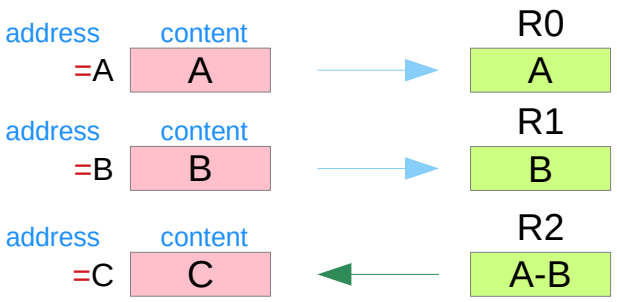
```
LDR     R0, =A
LDR     R0, [R0]
LDR     R1, =B
LDR     R1, [R1]
PUSH    {R0, R1}        ; input parameters
SUB     SP, #4          ; place for output
BL      Sub2
POP     {R2}            ; result
LDR     R0, =C
STR     R2, [R0]
ADD     SP, #8          ; balance stack
```

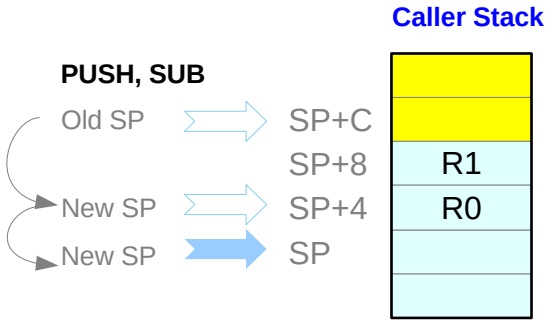
Put A, B on the stack

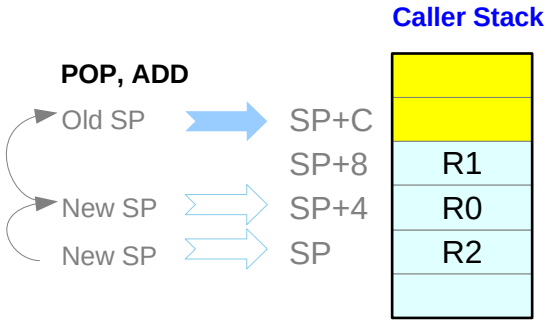
Allocate for C on the stack

Get the result C on the stack

Deallocate A, B on the stack



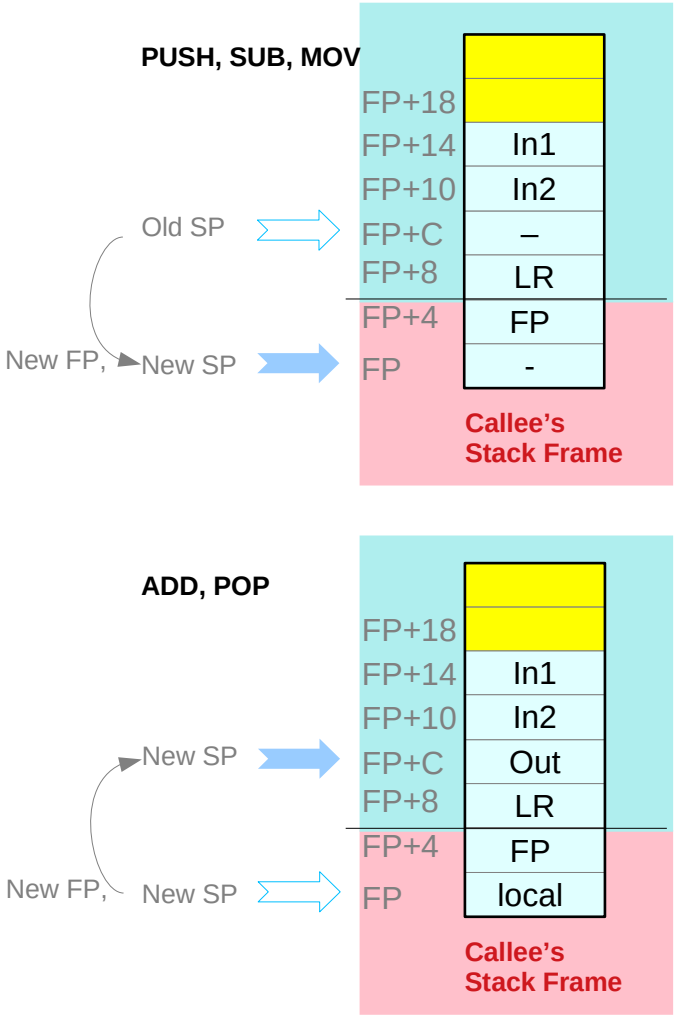




Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

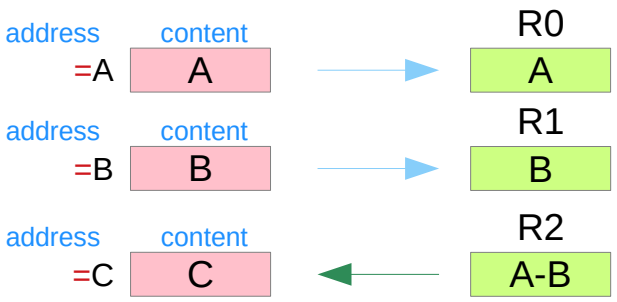
Parameter Passing – (3)-(a) using the stack frames

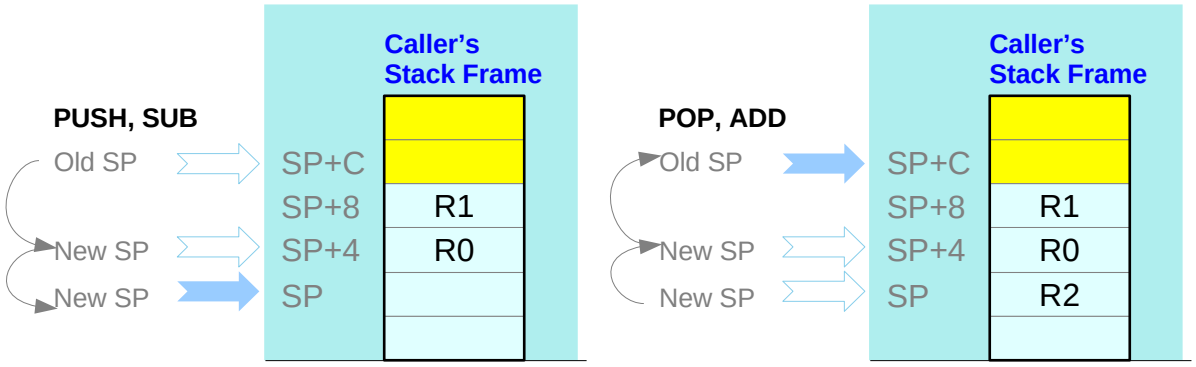
```
; Inputs:        In1 In2 on stack
; Outputs:       Out= In1 – In2 on stack
kIn1      EQU    20
kIn2      EQU    16
kOut      EQU    12
local     EQU    0
Sub3      PUSH   {FP, LR}
          SUB    SP, #4          ; not used
          MOV    FP, SP          ; allocate
          LDR    R1, [FP, #kIn1] ; frame pointer
          LDR    R0, [FP, #kIn2]
          SUB    R2, R0, R1
          STR    R2, [FP, #kOut]
          ADD    SP, #4          ; deallocate
          POP    {FP, PC}
```



Parameter Passing – (3)-(b) using the stack frames

```
LDR     R0, =A
LDR     R0, [R0]
LDR     R1, =B
LDR     R1, [R1]
PUSH    {R0, R1}        ; input parameters
SUB     SP, #4          ; place for output
BL      Sub3
POP     {R2}            ; result
LDR     R0, =C
STR     R2, [R0]        ; C = A – B
ADD     SP, #4          ; deallocate stack
```





Parameter Passing – (4) using global variables

```
; Inputs:       A, B
; Outputs:      C = A – B
Sub4    LDR     R0, =A
        LDR     R0, [R0]
        LDR     R1, =B
        LDR     R1, [R1]
        SUB     R2, R1, R0
        LDR     R0, =C
        STR     R0, [R0]
        BX      LR
```

; R0 has the value of A

; R1 has the value of B
; A – B

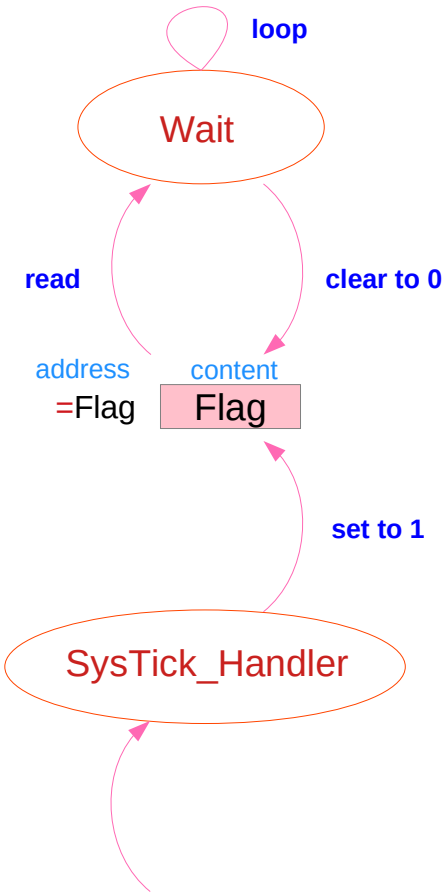
address	content
=A	A
=B	B
=C	C

```
        BL      Sub4
```

Parameter Passing – (5) using memory locations

```
; Wait for Flag to become 1
Wait
loop    LDR     R0, =Flag
        LDR     R1, [R0]      ; R1 = Flag       (read Flag)
        CMP     R1, #1
        BNE     loop          ; wait until 1
        MOV     R1, #0        ;
        STR     R1, [R0]      ; Flag = 0        (clear Flag)
        BX      LR
```

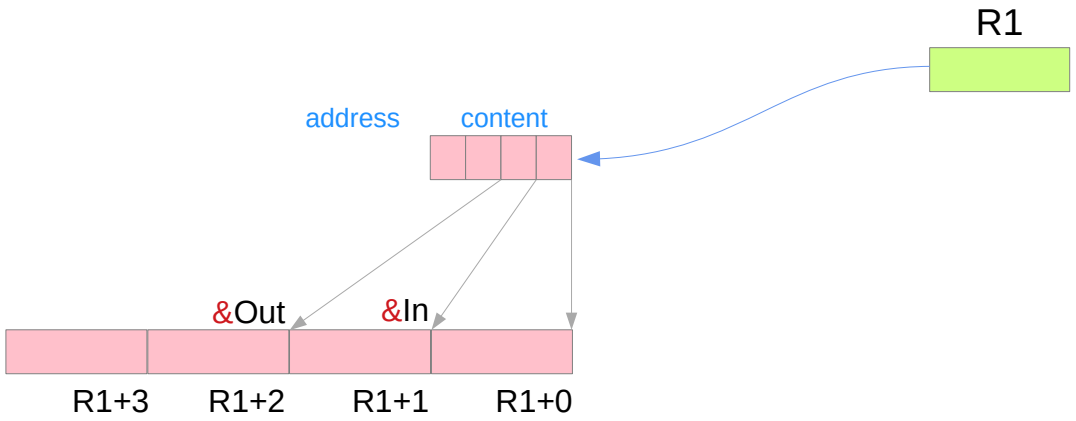
```
SysTick_Handler
        LDR     R0, =Flag     ; R0 = &Flag
        MOV     R1, #1
        STR     R1, [R0]      ; Flag = 1        (set Flag)
        BX      LR            ; return form interrupt
```



Promotion / Demotion Problem

Out = (99 * In) / 100; // 99 = (32+1) + (32+1)*2 = 33*3

```
            LDR     R1, [PC, #208]      ; (R1 + 1)  = &In
            MOVS    R2, #0x64           ; R2 = 100
Promotion   LDRB    R0, [R1, #0x01]     ; R0 = In                                   ; R0 ← [R1 + 1]
            ADD     R0, R0, R0, LSL #5  ; R0 = R0 + 32*R0 = (1+32) * In
            ADD     R0, R0, R0, LSL #1  ; R0 = R0 + 2 *R0 = (33 + 2*33) * In = 99 * In
            UDIV    R0, R0, R2          ; 99 * In / 100
Demotion    STRB    R0, [R1, #0x02]     ; Out = 99 * In / 100                       ; R0 → [R1 + 2]
```



Promotion :
R0 (32bit) ← In (8bit)

32-bit computation
Out = 99*In/100
Out < In always

Demotion :
R0(32bit) → Out(8bit)

No overflow!!!

Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

Precedence problem

```
uint32_t combine (
        uint32_t     msb,
        uint32_t     lsb) {
        return msb << 8 + lsb;
}
```


```
Combine   MOV     R3, R0          ; R0 = msb
          ADD     R3, R1, #0x08   ; lsb + 8
          LSL     R0, R2, R3      ; msb << (8 + lsb)        wrong precedence
          BX      LR
```

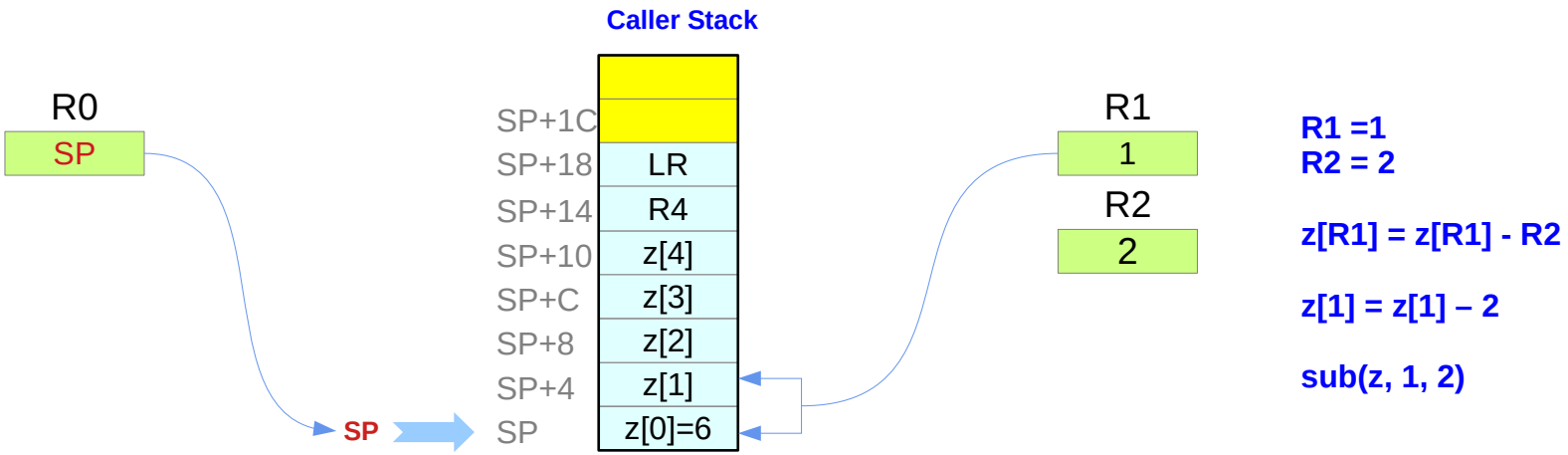
```
Combine   ADD     R3, R1, R0, LSL #8   ; R3 = R0 + 256*R1     (msb << 8) + lsb
          BX      LR                   ;                      correct precedence
```

Compiler's local and global variable implementation (2)

```
int32_t       G;                    // global

int32_t sub(int32_t *pt             // R0 parameter
            int32_t index,          // R1 parameter
            int32_t values) {       // R2 parameter
      pt[index] -= value;
      return value;
}


void main(void) {
      int32_t z[5];                 //local
      G = 3;                        // access global
      z[0] = 6;                     // access local
      G = sub(z, 1, 2);
}
```

Accessing local, global, parameter variables (1)

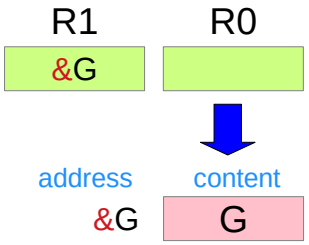
```
; R0 is *pt
; R1 is index
; R2 is value
sub:    MOV     R3, R0                  ; R3 is *pt
        LDR     R0, [R3, R1, LSL #2]    ; R0 ← [R3 + R1*4]
        SUBS    R0, R0, R2              ; R0 ← R0 – R2
        STR     R0, [R3, R1, LSL #2]    ; R0 → [R3 + R1*4]
        MOV     R0, R2                  ; return value
        BX      LR
```



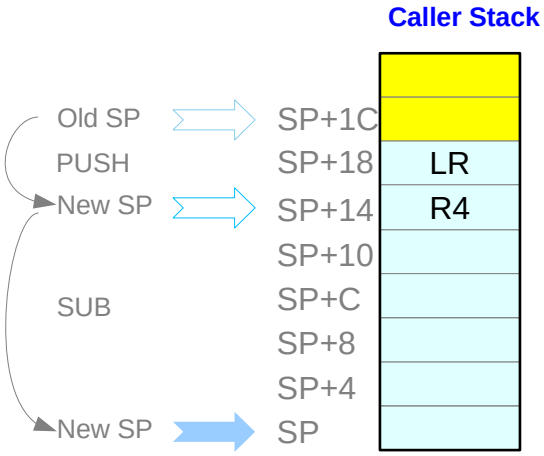
Accessing local, global, parameter variables (2)

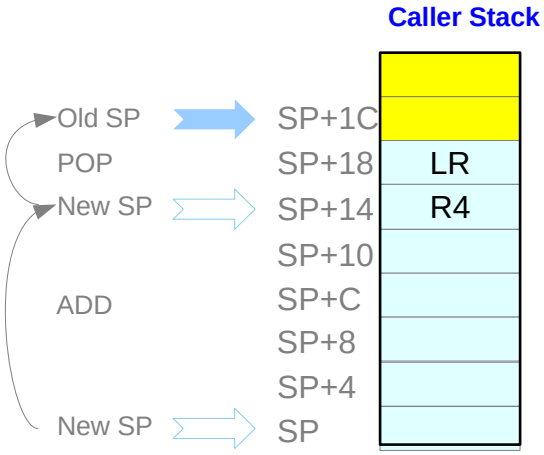
```
main:   PUSH    {R4, LR}
        SUB     SP, SP, #0x05       ; allocate z
        MOVS    R0, #0x03
R0      LDR     R1, [PC, #340]      ; R1 = &G
3       STR     R0, [R1, #0x00]     ; G=3
        MOVS    R0, #0x06
        STR     R0, [SP, #0x00]     ; z[0] = 6
        MOVS    R2, #0x02           ; value
        MOVS    R1, #0x01           ; index
        MOV     R0, SP              ; *pt
        BL.W    sub
R0      LDR     R1, [PC, #320]      ; R1 = &G
return val
        STR     R0, [R1, #0x00]     ; store G
        ADD     SP, SP, #0x05       ; deallocate
        POP     {R4, PC}
```

G=3;
G=sub(G,1,2);



Initially 3, then return value of sub

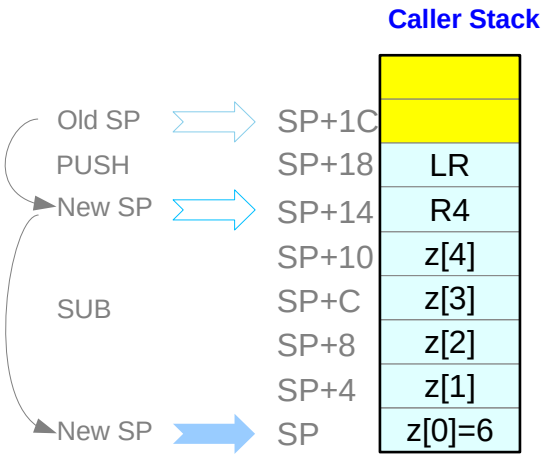


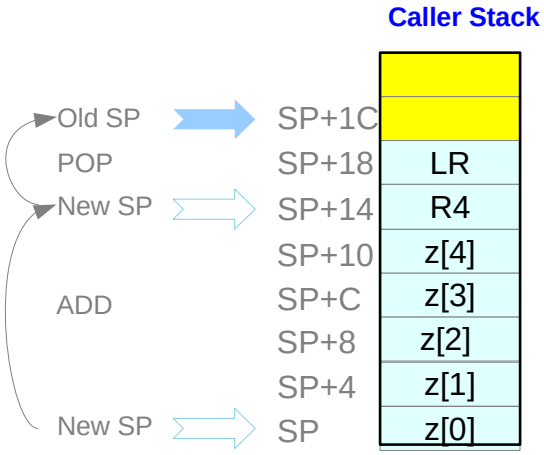


Accessing local, global, parameter variables (3)

```
main:   PUSH    {R4, LR}
        SUB     SP, SP, #0x05       ; allocate z
        MOVS    R0, #0x03
        LDR     R1, [PC, #340]      ; R1 = &G
        STR     R0, [R1, #0x00]     ; G=3
R0      MOVS    R0, #0x06
6       STR     R0, [SP, #0x00]     ; z[0] = 6
        MOVS    R2, #0x02           ; value
        MOVS    R1, #0x01           ; index
        MOV     R0, SP              ; *pt
        BL.W    sub
        LDR     R1, [PC, #320]      ; R1 = &G
        STR     R0, [R1, #0x00]     ; store G
        ADD     SP, SP, #0x05       ; deallocate
        POP     {R4, PC}
```

z[0] = 6





Accessing local, global, parameter variables (4)

```
main:   PUSH    {R4, LR}
        SUB     SP, SP, #0x05       ; allocate z
        MOVS    R0, #0x03
        LDR     R1, [PC, #340]      ; R1 = &G
        STR     R0, [R1, #0x00]     ; G=3
        MOVS    R0, #0x06
        STR     R0, [SP, #0x00]     ; z[0] = 6
        MOVS    R2, #0x02           ; value
        MOVS    R1, #0x01           ; index
        MOV     R0, SP              ; *pt
        BL.W    sub
        LDR     R1, [PC, #320]      ; R1 = &G
        STR     R0, [R1, #0x00]     ; store G
        ADD     SP, SP, #0x05       ; deallocate
        POP     {R4, PC}
```

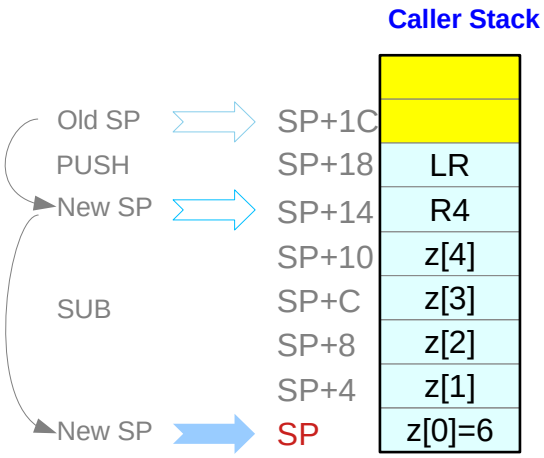
sub(G,1,2);

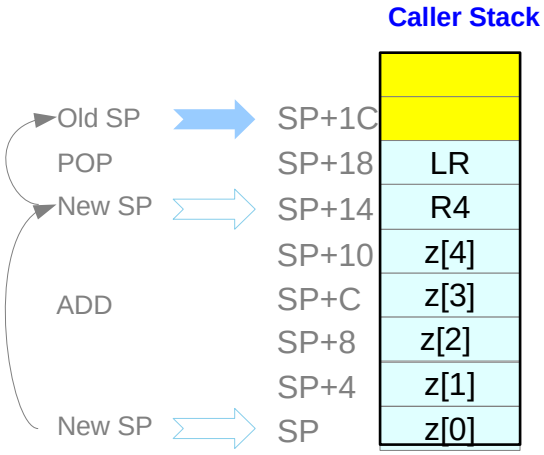
parameters

R0	R1	R2
SP	1	2

R0

return value of **sub**





Accessing local, global, parameter variables (6)

```
main:   PUSH    {R4, LR}
        SUB     SP, SP, #0x05       ; allocate z
        MOVS    R0, #0x03
0       LDR     R1, [PC, #340]      ; R1 = &G
1       STR     R0, [R1, #0x00]     ; G=3
2       MOVS    R0, #0x06
3       STR     R0, [SP, #0x00]     ; z[0] = 6
4       MOVS    R2, #0x02           ; value
5       MOVS    R1, #0x01           ; index
6       MOV     R0, SP              ; *pt
7       BL.W    sub
8       LDR     R1, [PC, #320]      ; R1 = &G
        STR     R0, [R1, #0x00]     ; store G
        ADD     SP, SP, #0x05       ; deallocate
        POP     {R4, PC}
```

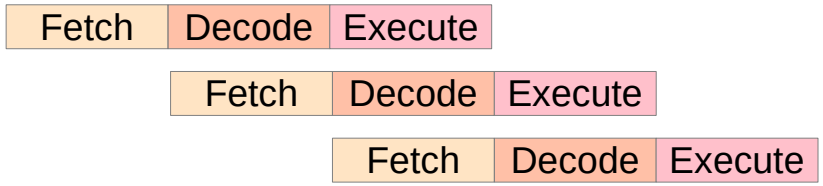
340

320

20_{16} =
8 * 4 = 32

address	content
&G	G

the actual PC value is always the address of the current instruction **+ 8**
Instruction Pipeline



BL.W

BL{cond}{.W} label

cond

is an optional condition code.

cond is not available on all forms of this instruction.

.W

is an optional instruction width specifier

to force the use of a **32-bit BL instruction in Thumb**.

label

is a **PC-relative expression**.

https://www.keil.com/support/man/docs/armasm/armasm_dom1361289865686.htm

Trigger.c

References

[1] http://wiki.osdev.org/ARM_RaspberryPi_Tutorial_C
[2] http://blog.bobuhiro11.net/2014/01-13-baremetal.html
[3] http://www.valvers.com/open-software/raspberry-pi/
[4] https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads.html