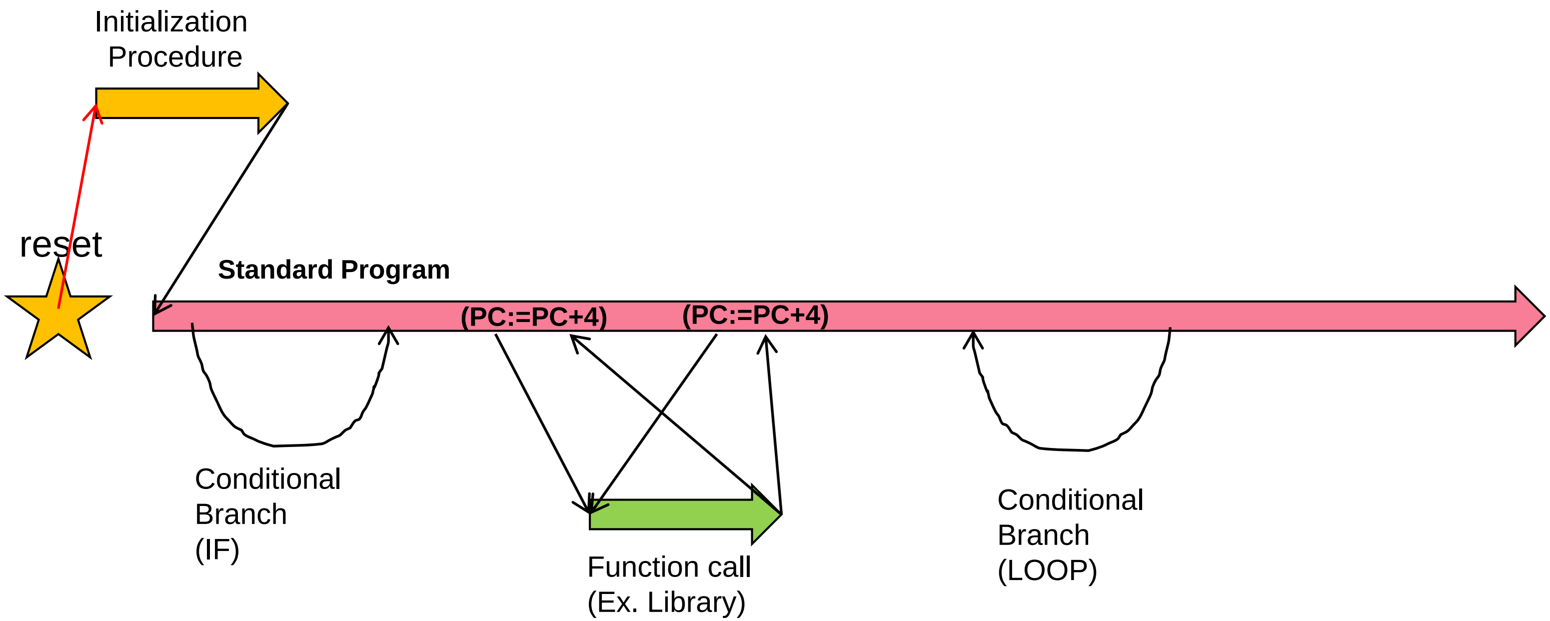
SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

ENSC254 – Subroutines and Stack

Ensc254 – Updated June 2021

School of Engineering Science
Simon Fraser University
Burnaby, BC, Canada

Function Calls



- Function calls are intentional alterations of the standard program flow to provide a "service" that is typically required often or is part of a library developed previously or externally

Function / Subroutine

What is a function call in C ?

```
int absdiff(int a, int b)
{
   if (a>b) return(a-b)
  else return(b-a);
}

int main()
{
  int x,y,z;
  …
  z=absdiff(x,y);
  …
}
```

The processor should

1) Pass arguments (x,y) for the function
2) Call the function (branch to the subroutine and link)
3) Compute result
4) Pass result
5) Return to caller providing result and resume calling code.

Function / Subroutine

BL and MOV can handle program flow control, branching to the subroutine and moving back to the calling part of the program

```
int absdiff(int a, int b)
{
   if (a>b) return(a-b)
  else return(b-a);
}

int main()
{
  int x,y,z;
  …
  z=absdiff(x,y);
  …
}
```

1) Pass arguments (x,y) for the function
2) **Branch to subroutine and link**
   BL absdiff
3) Compute result
4) Pass result
5) **Return to caller, right after the BL instruction:**
   MOV PC,LR

Quiz

Determine the content of the register file after the computation of the instruction below:

0x0C	…..
0x10	BL 0x38
0x14	….

r14	
r15	

a) R14=0x10, R15=0x20
b) R14=0x14, R15=0x20
c) R14=0x18, R15=0x38
d) R14=0x14, R15=0x38
e) R14=0x10, R15=0x14

Quiz

Determine the content of the register file after the computation of the instruction below:

0xc	…..
0x10	BL 0x38
0x14	….

r14	
r15	

a) R14=0x10, R15=0x20
b) R14=0x14, R15=0x20
c) R14=0x18, R15=0x38
d) R14=0x14, R15=0x38
e) R14=0x10, R15=0x14

R14=Link Register, Holds the address immediately following the function call, 0x14
R15=Program Counter, hold the destination address for the call, that is 0x10+0x08+0x20=0x38

Function / Subroutine

What is a function/subroutine in C ?

```
int absdiff(int a, int b)
{
   if (a>b) return(a-b)
  else return(b-a);
}

int main()
{
  int x,y,z;
  …
  z=absdiff(x,y);
  …
}
```

The processor should

1) **Pass arguments (x,y) for the subroutine**
2) Branch to subroutine and link:
   BL absdiff
3) Compute result
4) **Pass result**
5) Return to caller, right after the BL instruction:
   MOV PC,LR

How shall we pass arguments for a Subroutine?

```
int z,x,y;
main()
{
   x=4; y=3;
   z=absdiff(x,y)
}
```

```
int absdiff(int a, int b)
{
   if (a>b) return(a-b);
   else return(b-a)
}
```

Please note that argument passing is a purely SOFTWARE concept, the processor is simply processing its instructions without understanding anything is being passed

PASSING ARGUMENTS TROUGH REGISTERS

```
int z,x,y;
main()
{
   x=4; y=3;
   z=absdiff(x,y)
}
```

C Code

```
int absdiff(int a, int b)
{
   if (a>b) return(a-b);
   else return(b-a)
}
```

```
   LDR r4,=z
   LDR r0,[r4,#4]
   LDR r1,[r4,#8]
   BL absdiff
   STR r0,[r4,#0]
```

ARM ASM Code

```
absdiff:
        CMP     r0,r1
        SUBGT   r0,r0,r1
        SUBLE   r0,r1,r0
        MOV     PC,LR
```

```
absdiff:
        SUBS    r0,r0,r1
        RSBLE   r0,r0,#0
        MOV     PC,LR
```

Please note that argument passing is a purely SOFTWARE concept, the processor is simply processing its instructions without understanding anything is being passed

PASSING PARAMETERS THROUGH REGISTERS

Note: if you are using the GNU toolchain and want to be debugging your code, use the .size directive below your function:

.size absdiff, . - absdiff

If we have a restricted number of arguments we can pass them through conventional registers.

Calling code and called function/subroutine must agree on a given ***PROTOCOL*** for information exchange and then they can exchange data.

Compilers for ARM support the ***ARM APCS*** (Application Procedure Call Standard):

- When calling a function and if needed, often r0 is the first argument, r1 the second, r2 the third, r3 the fourth (this is the case for 32-bit values)
- *When returning from the function, r0 contains the returned result if any, together with r1 if result is 64-bit pass-by-value (e.g. long long int)*
- A copy of the standard is in the files area on Canvas.

QUIZ

1. Note that oftentimes, functions need to work on large data, such as arrays (vectors) or matrices (for example, sound samples or images).

2. Sometimes, a function may need to return more than a single value

HOW IS THIS SITUATION RESOLVED WHEN PROGRAMMING IN C/C++ ?

Pointers

1. Note that oftentimes, functions need to work on large chunks of data, such as array or matrices (for example, images or sound samples).
2. Sometimes, a function may need to return more than a single value

In such situations, the issue is resolved at the C level using ***pointers***. In this case the arguments are said to be passed by ***REFERENCE*** rather than by ***VALUE.***

In this case, a pointer (address) to an applicable input is passed as an argument, for example in the r0 register. In C++, the language has been expanded to better support references, and the below is sometimes called “Pass by Pointer”.

```
int a1[5] = {2, 4, 1, 1, 3};
int a2[5] = {1, 0, 1, 1, 2};
main()
{
   z=mac(a1, a2, 5);
}
```

```
int mac(int* x, int* y, int size)
{   int acc=0;
   for(i=0; i<size; ++i)
        { acc += x[i] * y[i];}
   return(acc);
}
```

Oh, If only it was that simple!!!

```
int z;
int a1[5] = {2,4,1,1,3};
int a2[5] = {1,0,1,1,2};
main()
{
   z=mac(a1, a2, 5);
}
```

```
int mac(int* x, int* y, int size)
{  int acc=0;
   for(i=0;i<size;i++)
       { acc += x[i] * y[i];}
   return(acc);
}
```

```
LDR r4,=z
LDR r0,=a1
LDR r1,=a2
MOV r2,#5
BL mac
STR r0,[r4]
```

```
mac:    MOV r6,#0
Loop:   LDR r4,[r0],#4
        LDR r5,[r1],#4
        MLA r6,r4,r5,r6
        SUBS r2,r2,#1
        BNE Loop
        MOV r0,r6
        MOV PC,LR
```

Can You Spot Anything Fishy Here ???

Oh, If only it was that simple!!!

```
int z;
int a1[5] = {2,4,1,1,3};
int a2[5] = {1,0,1,1,2};
main()
{
   z=mac(a1, a2, 5);
}
```

```
int mac(int* x, int* y, int size)
{   int acc=0;
   for(i=0;i<size;i++)
       { acc += x[i] * y[i];}
   return(acc);
}
```

```
LDR r4, =z
LDR r0, =a1
LDR r1, =a2
MOV r2, #5
BL mac
STR r0, [r4]
```

```
mac:    MOV   r6,#0
Loop:   LDR   r4,[r0],#4
        LDR   r5,[r1],#4
        MLA   r6,r4,r5,r6
        SUBS  r2,r2,#1
        BNE   Loop
        MOV   r0,r6
        MOV   PC,LR
```

Does coloring help finding the bug???

Oh, If only it was that simple!!!

```
main:
LDR r4,=z
LDR r0,=a1
LDR r1,=a2
MOV r2,#5
BL mac
STR r0,[r4]
```

```
mac:    MOV  r6,#0
Loop:   LDR  r4,[r0], #4
        LDR  r5,[r1], #4
        MLA  r6,r4,r5,r6
        SUBS r2,r2,#1
        BNE  Loop
        MOV  r0,r6
        MOV  PC,LR
```

- Please note how the subroutine *mac* is destroying the former value of R4.
- Note that when writing *mac* a programmer is not suppose to know in general what registers are currently in use, so this problem can't actually be avoided!
  - ***On the other hand, the calling code understands that r0 to r3 may be overwritten, because they are used for arguments (and r0 to r1 may carry outputs). But the calling code ASSUMES registers r4 to r11 remain unchanged during the function call, while the subroutine MAY NEED to use them.***

Re-Entrant Subroutines

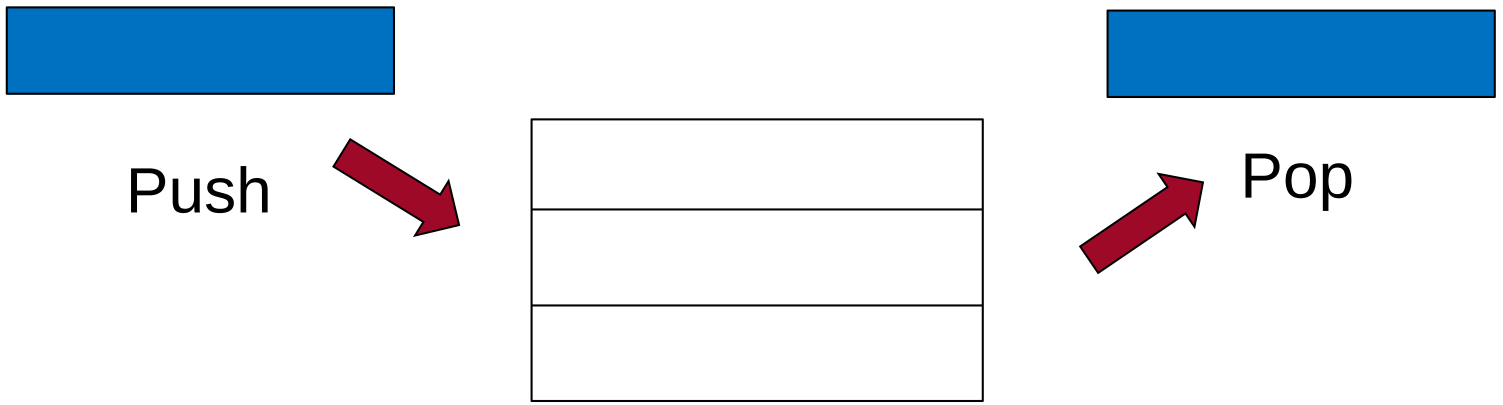
- Often we want a function/subroutine that returns the processor in close to the same state it has found it, including most registers (but in the case of ARM, given the APCS, excluding r0-r3 and also r12).

- ***How can we make our subroutines do this?***
  - ***Before we start computing, we need to SAVE the status of any registers in (r4-r11) that we plan to use, and RESTORE them before we give control back to the caller***
  - ***If we do that, in between these steps we are free to use the registers as much as we want!***

The most convenient way to save/restore registers is to use a STACK

STACK

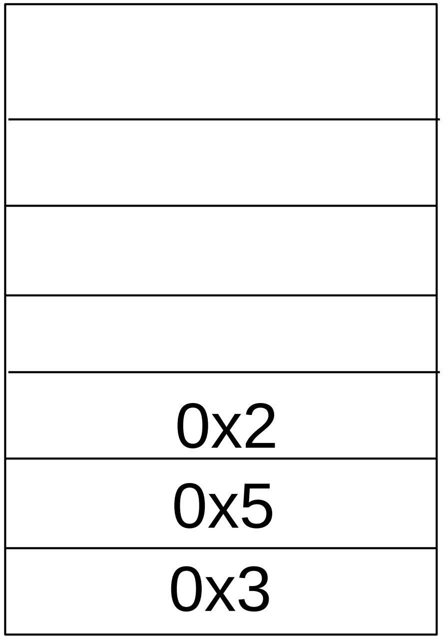
- A stack is one logical way to organize a memory space, and it is based on a `LIFO` pattern (`Last In, First Out`)
- Note that A STACK IS NOT A PHYSICAL DEVICE, but only a logical way to use a portion of standard memory
- Only two operations are available on a stack: PUSH (Add value to Stack) and POP (Read and Remove value from Stack).



STACK

Please note that you don't get to choose where to Push or Pop your data: You can only *Push to* and *Pop from* the "top" location:

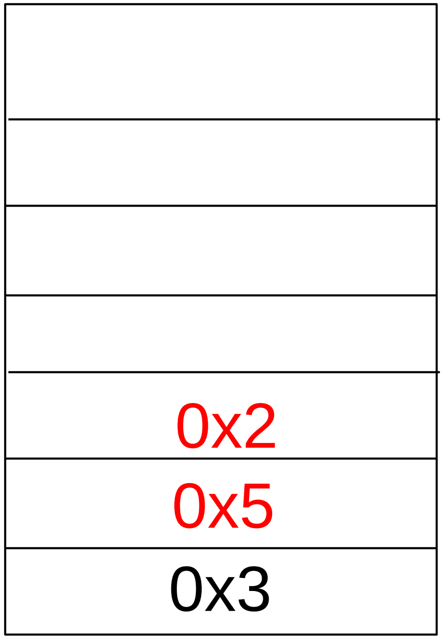
Push 0x3
Push 0x5
Push 0x2



STACK

Please note that you don't get to choose where to Push or Pop your data: You can only *Push to* and *Pop from* the highest location:

Push 0x3
Push 0x5
Push 0x2



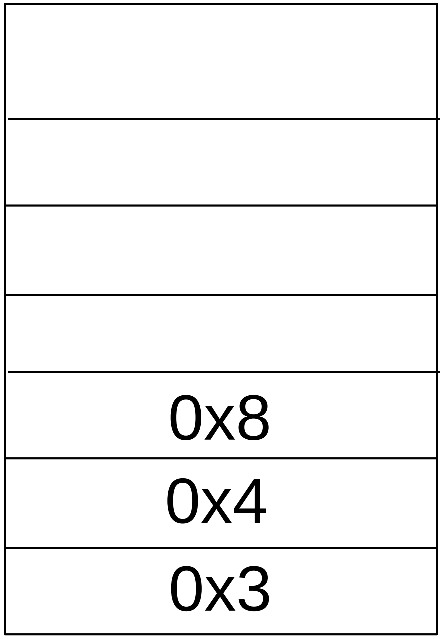
A=Pop(); --> A=2
B=Pop(); --> B=5

STACK

- Please note that you don't get to choose where to Push or Pop your data: You can only *Push to* and *Pop from* the highest location:
- The Pop is assumed to be a *destructive* operation. After you pop a value from the stack, that value is not guaranteed to be available any more

Push 0x3
Push 0x5
Push 0x2

Push 0x4
Push 0x8

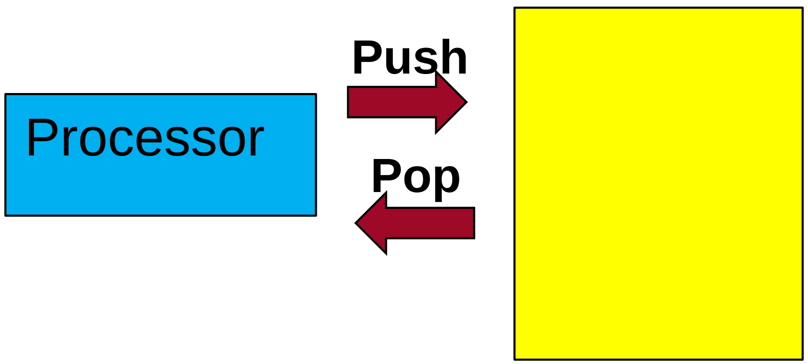


A=Pop(); --> A=2
B=Pop(); --> B=5

STACK

A stack is “implemented” by a processor architecture using its memory and based on a counter value:

```
#define STACK_SIZE 1024
int stack[STACK_SIZE];
int counter = STACK_SIZE;
void  Push(int value)
{
  stack[--counter]=value;
}
int Pop()
{
  return(stack[counter++]);
}
```



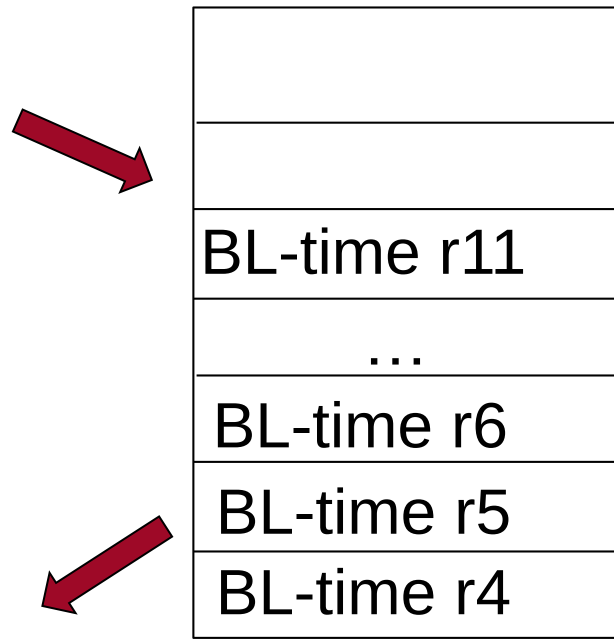
Using Stack in Re-Entrant Function Calls

It is easy to see how a Stack configuration is a handy way to build convenient functions:

```
main:
LDR r4,=z
LDR r0,=a1
LDR r1,=a2
MOV r2,#5
BL mac
STR r0,[r4]
```

```
mac:    Push {r4}
        Push {r5}
        ;@ … (Until r11)

        MOV r6,#0
Loop:   LDR r4,[r0], #4
        LDR r5,[r1], #4
        MLA r6,r4,r5,r6
        SUBS r2,#1
        BNE Loop
        MOV r0,r6
        Pop {r11}
        Pop {r10}
        ;@ … (until r4)
        MOV PC,LR
```

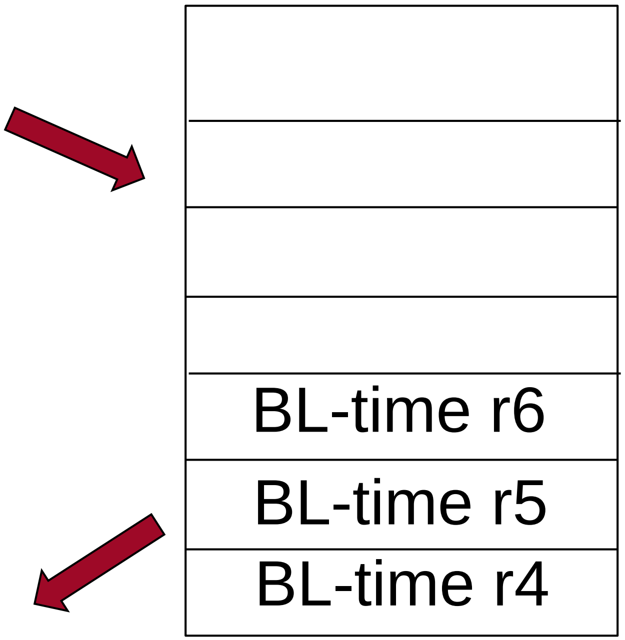


Using Stack in Re-Entrant Function Calls

It is easy to see how a Stack configuration is a handy way to build re-entrant functions: IN FACT, we don’t even need to save all r4-r11 registers, but only those that are used in the function

```
main
LDR r4,=z
LDR r0,=a1
LDR r1,=a2
MOV r2,#5
BL mac
STR r0,[r4]
```

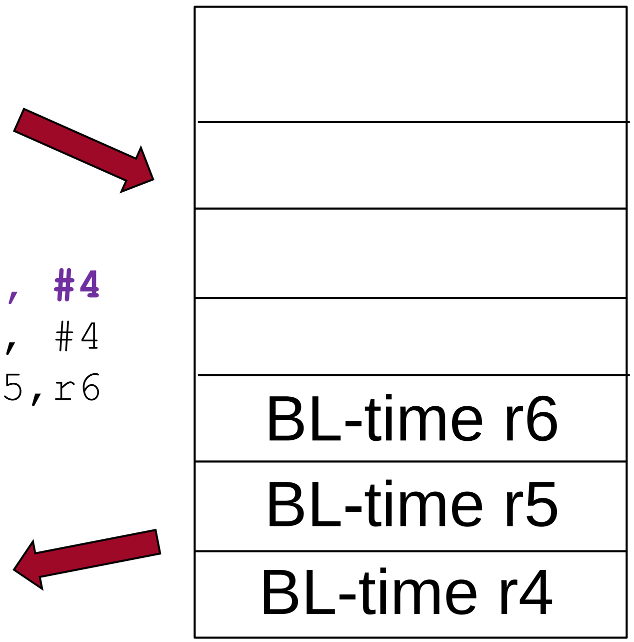
```
mac:    Push {r4}
        Push {r5}
        Push {r6}
        MOV r6,#0
Loop:   LDR r4,[r0], #4
        LDR r5,[r1], #4
        MLA r6,r4,r5,r6
        SUBS r2,#1
        BNE Loop
        MOV r0,r6
        Pop {r6}
        Pop {r5}
        Pop {r4}
        MOV PC,LR
```



Implementing Push/Pop in ARM Assembly

- The Push and Pop functionality with individual registers can easily be rendered using LDR / STR … register R13 is especially reserved in the APCS to act as STACK POINTER (SP)

```
mac:    Push {r4}
        Push {r5}
        Push {r6}
        MOV r6,#0
Loop:   LDR r4,[r0], #4
        LDR r5,[r1], #4
        MLA r6,r4,r5,r6
        SUBS r2,#1
        BNE Loop
        MOV r0,r6
        Pop {r6}
        Pop {r5}
        Pop {r4}
        MOV PC,LR
```




```
mac:    Str r4,[SP, #-4]!
        Str r5,[SP, #-4]!
        Str r6,[SP, #-4]!
        MOV r6,#0
Loop:   LDR r4,[r0], #4
        LDR r5,[r1], #4
        MLA r6,r4,r5,r6
        SUBS r2,#1
        BNE Loop
        MOV r0,r6
        Ldr r6,[SP],#4
        Ldr r5,[SP],#4
        Ldr r4,[SP],#4
        MOV PC,LR
```

Quiz (1):

- The stack is accessed by means of Store / Load operations that can implement the logic of PUSH / POP primitives using a counter register that we call STACK POINTER. **But physically, where is the Stack located ?**

a) Register File
b) ALU
c) SRAM (Read/Write Memory)
d) Flash or ROM (Non Volatile Memory)
e) Nowhere! It is just a logic concept with no physical equivalent or representation

Quiz (1):

- The stack is accessed by means of Store / Load operations that can implement the logic PUSH / POP primitives using a counter register that we call STACK POINTER. **But physically, where is the Stack located ?**

a) Register File
b) ALU
c) SRAM (Read/Write Memory)
d) Flash or ROM (Non Volatile Memory)
e) Nowhere! It is just a logic concept with no physical equivalent or representation

The stack is continuously updated during computation, so it must be placed in easily readable/writable memory within easy access from the core

Quiz (2):

- How can we reserve empty space in the memory especially dedicated to the stack ?

Quiz (2):

- **How can we reserve an empty space in the memory especially dedicated to the stack ?**

Well, one possibility is to use the SPACE directive:

```
STACKSZ         EQU     0x4000
        AREA    aSTACK, DATA, READWRITE
stack   SPACE   STACKSZ
stackInit
```

Alternatively, the stack can be defined in the linker script (for the GNU toolchain) or Scatter File (for the ARM toolchain) so that we have it for all programs and we don’t need to redefine it each time.

If we used a startup file in Keil tools, then the startup code will read from the scatter file the stack address and load the stack pointer for us

Definition of the stack

- Of course, in order to implement the STACK as introduced before, we must place it somewhere in the addressing space of the processor.

- Note that, since we must read to and write from it, it MUST be READWRITE. Also, ***it must be a specifically reserved area***, because it must not overwrite other significant data.

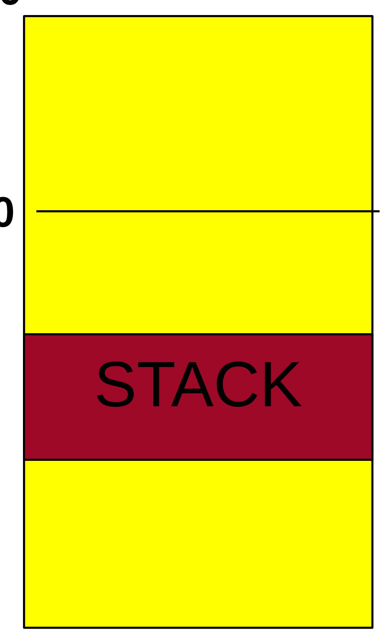
- Defining the Stack size is purely a software problem: We can change the stack size in every program, or let the OS do that for us if we have an OS.

- *It is a very delicate choice, because we don't want to waste large amounts of memory, but we also don't want to run out of stack when calling many – possibly recursive – subroutines*

0x0000 0000

0x4000 0000

In LPC2104 here is where SRAM begins



Stack Overflow Error

- Ever heard about stack overflow error?
  - That is a common source of error in some programming environments and computer systems, like stepping on the cable and unplugging a device from power

- Simple Example: try this on some C environments (with optimization turned off)

```
void f() { f(); }
main(void) {
        f(); return 0;
}
```

You'll oftentimes get the ever-famous error "SEGMENTATION FAULT"

Stack Overflow Error

How would such a sweet and innocent guy cause problems like that ?

```
void f() { f(); }
main(void) {
  f(); return 0;
}
```

Stack Overflow Error

How would such a sweet and innocent guy cause problems like that ?

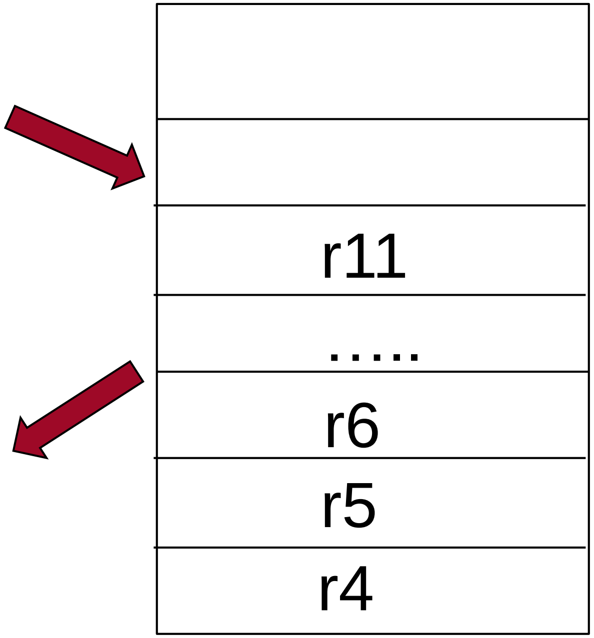
```
void f() { f(); }
main(void) {
  f(); return 0;
}
```

- The code above will continuously keep saving the status (return address) on the stack and call itself again, until the point where the stack exceeds its allocated space and wants to overwrite some other useful regions of memory
- If we have an OS running, the OS will interrupt the program, inhibit any further expansion of the stack, and issue an error.
- If we don’t have an OS, many processors (ARM Included) will have hardware checks on the memory areas that the user can access, and if we overflow that the checks will issue an error
- If we haven’t that either, the stack will “INVADE” all sorts of memory and perhaps overwrite the program, causing global system failure!

Load Multiple, Store Multiple

- Note that in a general case we may need to queue up to 8 or even more store (and load) operations, which could be quite costly in terms of instruction memory.
- For this reason, ARM introduces the very UNRISC-y instructions LDM / STM (Load Multiple – Store Multiple) that, starting from a reference address, can store a collection of registers in a row

```
Str r4,[SP, #-4]!
Str r5,[SP, #-4]!
;@ … (Until r11)
;@ …
;@ (CODE)
;@ …
Ldr r11,[SP],#4
Ldr r10,[SP],#4
;@ … (until r4)
MOV PC,LR
```




```
STMDB SP!,{r4-r11}
;@ …
;@ (CODE)
;@ …
LDMIA SP!,{r4-r11}
MOV PC,LR
```

Load Multiple, Store Multiple

LDM{<cond>}<addressing_mode> <Rn>{!}, <registers>

STM{<cond>}<addressing_mode> <Rn>{!}, <registers>

- <cond> represents the standard ARM condition codes
- <Rn> is a register containing an address. The start of the set of load/store operations is derived from that address. It is r13 in the case of stack usage (for APCS subroutines) but LDM/STM are not used only for the stack, they can be used IN ANY SITUATION where a set of consecutive Loads/Stores is required.

Load Multiple, Store Multiple

			15 … 0
cond	110 PU0W1	Rn	Register_list

- cond : usual condition code
- W Writeback address increment (Activated by !)
- Register_List : if bit [i] = 1 register [i] is saved (Function will have different time depending on the number of registers saved)
- P = 1 ==> Pre Indexed addr (IB or DB) ; P=0 ==> Post Indexed addr (IA or DA)
- U = 1 ==> “Upward”, address Increment (IA, IB); U=0 ==> “Downward”, address Decrement (DA, DB)
- Those two last parameters are specified by the <address mode> suffix:
  - IA Increment After load/store
  - IB Increment Before load/store
  - DA Decrement After load/store
  - DB Decrement Before load/store

Alternate addressing-mode naming

Please accept my apologies for this confusing situation, but some times the addressing mode of LDM / STM are expressed with alternative names. This is seen from the standpoint of a stack that can be ascending or descending and full (pointing to the last pushed value) or empty (pointing to next location to push something to)

	push	pop
• FD Full "Descending"	(STMDB	/ LDMIA)
• FA Full "Ascending"	(STMIB	/ LDMDA)
• ED Empty "Descending"	(STMDA	/ LDMIB)
• EA Empty "Ascending"	(STMIA	/ LDMDB)

- The ARM APCS uses a Full Descending stack.
- So common Push is ‘STMFD SP!’ and Pop is ‘LDMFD SP!’

Argument passing to subroutines: Resume

According to the ARM APCS:

- Functions featuring up to 4 word-sized inputs pass arguments on registers r0-r3
- Functions featuring more than 4 word-sized inputs use the STACK for argument passing as well.

- In all cases, the processor is simply executing a MOV or STR / STM operation, it is not in any way aware from the hardware standpoint of the argument passing.

Example / Tutorial

The Loop can be somewhat unrolled in the MAC example.

```
Reset_Handler
          MOV    r0,#10
          LDR    r1,=A1
          LDR    r2,=A2
          MOV    r5, #0

Loop      LDR    r6,[r1],#4
          LDR    r7,[r2],#4
          MLA    r5,r6,r7,r5
          SUBS   r0,r0,#1
          BNE     Loop

Stop      b Stop

A1        DCD 2,4,1,1,3,2,1,3,4,2
A2        DCD 1,0,1,1,2,0,1,2,0,1

          END
```

```
Reset_Handler
          MOV    r0,#10
          LDR    r1,=A1
          LDR    r2,=A2
          MOV    r5, #0

Loop      LDR    r6,[r1],#4
          LDR    r7,[r2],#4
          LDR    r8,[r1],#4
          LDR    r9,[r2],#4
          MLA    r5,r6,r7,r5
          MLA    r5,r8,r9,r5
          SUBS   r0,r0,#2
          BNE     Loop

Stop      b Stop

A1        DCD 2,4,1,1,3,2,1,3,4,2
A2        DCD 1,0,1,1,2,0,1,2,0,1

          END
```

Example / Tutorial (2)

Loop unrolling. Loop unrolling is a compiler optimization technique that trades instruction memory size for speed, repeating more than once the instruction in a loop to minimize branch overhead

```
Reset_Handler
          MOV    r0,#10
          LDR    r1,=A1
          LDR    r2,=A2
          MOV    r5, #0

Loop      LDR    r6,[r1],#4
          LDR    r7,[r2],#4
          LDR    r8,[r1],#4
          LDR    r9,[r2],#4
          MLA    r5,r6,r7,r5
          MLA    r5,r8,r9,r5
          SUBS   r0,r0,#2
          BNE     Loop

Stop      b Stop

A1        DCD 2,4,1,1,3,2,1,3,4,2
A2        DCD 1,0,1,1,2,0,1,2,0,1

          END
```

```
Reset_Handler
          MOV    r0,#10
          LDR    r1,=A1
          LDR    r2,=A2
          MOV    r5, #0

Loop      LDMIA r1!,{r6,r8}
          LDMIA r2!,{r7,r9}
          MLA    r5,r6,r7,r5
          MLA    r5,r8,r9,r5
          SUBS   r0,r0,#2
          BNE     Loop

Stop      b Stop

A1        DCD 2,4,1,1,3,2,1,3,4,2
A2        DCD 1,0,1,1,2,0,1,2,0,1

          END
```

Measurements

Results:

Standard: **code size of 124 Bytes and 11.25 usec**

Unrolled: **code size of 136 Bytes and 9.58 usec**

Unrolled with LDM: **code size of 128 Bytes and 7.92 usec**

Notice the relevant optimization in the load mechanism, essentially based on the pipeline structure.

```
0.083 us        MOV    r0,#10
0.250 us        LDR    r1,=A1
0.250 us        LDR    r2,=A2
            loop
1.250 us        LDR    r6,[r1],#4
1.250 us        LDR    r7,[r2],#4
1.250 us        LDR    r8,[r1],#4
1.250 us        LDR    r9,[r2],#4
1.250 us        MLA    r5,r6,r7,r5
1.250 us        MLA    r5,r8,r9,r5
0.417 us        SUBS   r0,#2
1.083 us        BNE     loop
```

```
            Reset_Handler
0.083 us        MOV    r0,#10
0.250 us        LDR    r1,=A1
0.250 us        LDR    r2,=A2
            loop
1.667 us        LDMIA  r1!,{r6,r8}
1.667 us        LDMIA  r2!,{r7,r9}
1.250 us        MLA    r5,r6,r7,r5
1.250 us        MLA    r5,r8,r9,r5
0.417 us        SUBS   r0,#2
1.083 us        BNE     loop
```

SUBS r0,#2 is a shorthand for SUBS r0,r0,#2 … like a+=2 compared with a=a+2

Example: Let’s make a function out of our MAC code

The unrolled version of the MAC described in the previous slides uses more registers than the example we originally introduced. As an exercise of defining subroutines, we now want to implement it as an independent function
*int mac(int *x, int *y, int size)*

```
Reset_Handler
        ;@ STARTUP
        LDR      sp,=stackInit
        ;@ preparing parameters for function
        ;@    call: int mac(int *x,int * y,int size)
        ;@ Remember that the order of operands must be respected
        ;@    so r0 holds x, r1 holds y, and  r2 holds size
        LDR   r0,=A1
        LDR   r1,=A2
        MOV   r2,#10
        BL mac
Stop
        b stop
```

Example

```
    ;@ This is the unrolled version of the function: Note that if SIZE was
    ;@    odd, this version likely would not work!!!!  What's a quick solution?
Mac
    ;@ Saving on the stack the registers that get modified in the function
    PUSH {r4-r7}
    MOV r3, #0
Loop
    LDMIA r0!,{r4,r5}
    LDMIA r1!,{r6,r7}
    MLA   r3,r4,r6,r3
    MLA   r3,r5,r7,r3
    SUBS  r2,r2,#2
    BNE   Loop
    MOV   r0,r3 ;@ We use R3 as it is one of the registers that
                ;@    doesn't need to be saved. Output must be in r0
    POP {r4-r7}
    MOV PC,LR
```

Memory Space

Memory Space organization:

```
            AREA     bDATA, DATA, READWRITE

A1          DCD 2,4,1,1,3,2,1,3,4,2
A2          DCD 1,0,1,1,2,0,1,2,0,1

; Enough for this small example, but if
we were nesting many subroutines ...
stack       SPACE 400
stackInit
            END
```

Tutorial 3: What registers are being saved?

Consider the following C code

```
1 int a;
2 int sum(int a, int b)
3 {
4         return(a+b);
5 }
6 int headache(int a, int b)
7 { int c,d;
8         c=(a*b+1)<<3;
9   d=(a-b+7)^2;
10        return(sum(b,c));
11 }
12
13 main()
14 {a=headache(a,1);}
```

```
0x000001C4  E92D4070  STMDB     R13!,{R4-R6,R14}
0x000001C8  E1A04000  MOV       R4,R0
0x000001CC  E1A03001  MOV       R3,R1
     8:          c=(a*b+1)<<3;
0x000001D0  E0000394  MUL       R0,R4,R3
0x000001D4  E2800001  ADD       R0,R0,#0x00000001
0x000001D8  E1A05180  MOV       R5,R0,LSL #3
     9:    d=(a-b+7)^2;
0x000001DC  E0440003  SUB       R0,R4,R3
0x000001E0  E2800007  ADD       R0,R0,#0x00000007
0x000001E4  E2206002  EOR       R6,R0,#0x00000002
    10:          return(sum(b,c));
0x000001E8  E1A01005  MOV       R1,R5
0x000001EC  E1A00003  MOV       R0,R3
0x000001F0  EB00000C  BL        sum(0x00000228)
0x000001F4  E8BD4070  LDMIA     R13!,{R4-R6,R14}
```

The compiler here is saving R14,as we are calling another function that would destroy it. If we did not have a nested call, r14 would not need to be saved!