

# ECE3849 D-Term 2021

Real Time Embedded Systems

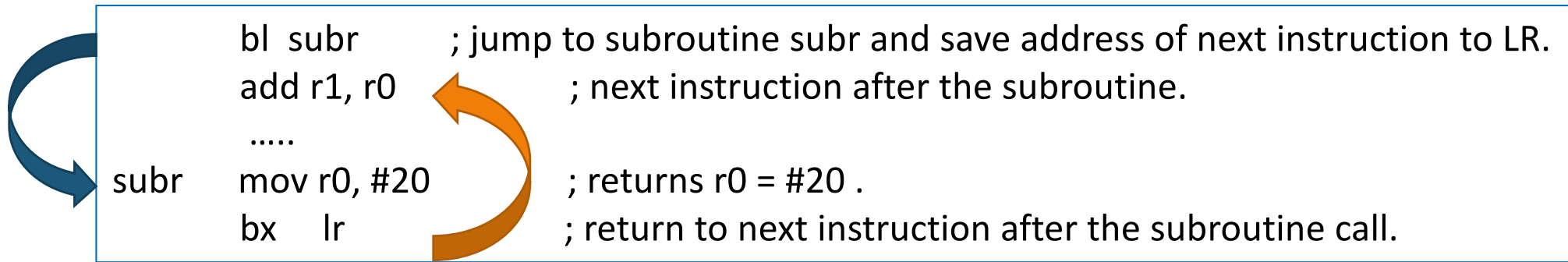
Module 5 Part 5

# Module 5 Part 4 Overview

- ARMv7 Instruction Set.
  - Subroutines
  - Stack Operations
  - C Function Register Conventions
  - CPU Performance / Instruction Timing

# Subroutines (functions)

- To call a subroutine use the BL or BLX instructions.
  - BL <label> ; calls the subroutine at the labels address.
  - BLX Rm ; Rm is the register containing the address of the subroutine to call.
- BL / BLX saves the address of the next instruction to LR = r14.
  - The program counter will jump to address when the subroutine returns.
- To return from the subroutine we use the BX command.
  - BX LR



- If our subroutine needs to call another subroutine, then we must save the contents of LR to the stack.

# Stack Operations: PUSH

- PUSH: places the contents of one or more registers on the stack and then updates the stack pointer to a new location.
  - PUSH <reglist>
  - Equivalent to: STMFD SP!, <reglist> ;
- When pushing data on to the stack the stack pointer is decremented.
  - The stack “grows” downward in memory.
  - If not properly allocated, the stack can overflow and may start writing over data in other parts of the code.
- Example
  - push {r4-r6, lr}
  - Is equivalent to
    - $\text{mem}_{32}[\text{sp} - 16] = \text{r4};$  lowest CPU number in lowest address.
    - $\text{mem}_{32}[\text{sp} - 12] = \text{r5};$
    - $\text{mem}_{32}[\text{sp} - 8] = \text{r6};$
    - $\text{mem}_{32}[\text{sp} - 4] = \text{lr};$
    - $\text{sp} -= 16$  ; updates stack pointer to new address

# Stack Operations: POP

- POP: Loads multiple registers from the stack.
  - POP <reglist>
  - LDMFD SP!, <reglist> ;
- When taking data off the stack the stack pointer is incremented.
- Example
  - pop {r4-r6, lr}
  - Is equivalent to
    - r4 = mem<sub>32</sub>[sp] ; reads back lowest register from lowest address.
    - r5 = mem<sub>32</sub>[sp+4] ;
    - r6 = mem<sub>32</sub>[sp+8] ;
    - lr = mem<sub>32</sub>[sp+12] ;
    - sp += 16 ; updates stack pointer to new address
- Very important that you keep track of the correct register order and number when pushing and pulling from the stack.
  - The stack can underflow when more data has been popped and pushed. This can lead to accessing data in other parts of the code.

# Push and Pop Example

## Example

```
push {r4-r6,r1}
```

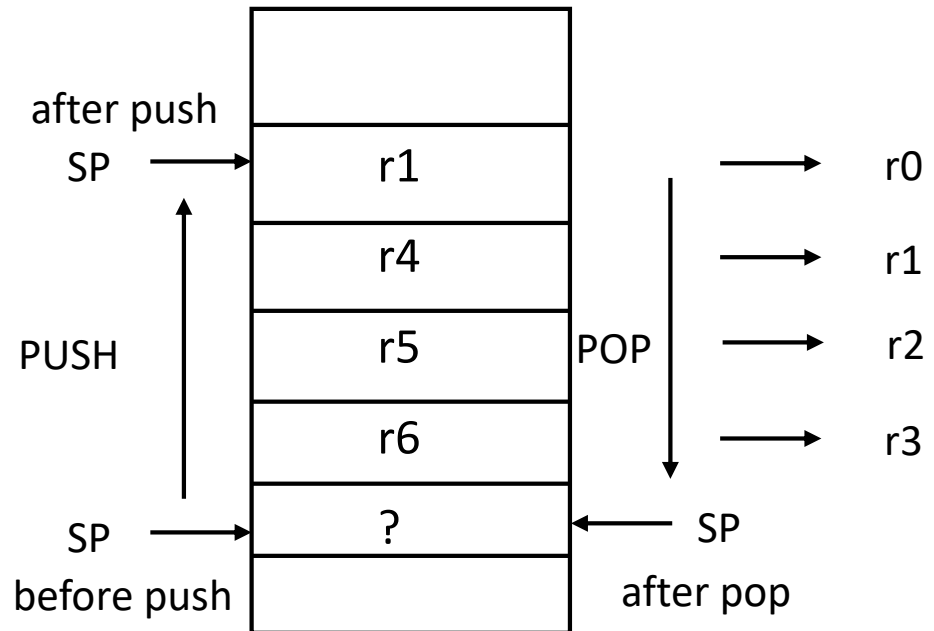
```
pop {r0-r3}
```

- **push {r4-r6,r1}** equivalent to

```
mem32[sp - 16] = r1;  
mem32[sp - 12] = r4;  
mem32[sp - 8] = r5;  
mem32[sp - 4] = r6;  
sp -= 16 ;
```

- **pop {r0-r3}** equivalent to

```
r0 = mem32[sp] ;  
r1 = mem32[sp+4] ;  
r2 = mem32[sp+8] ;  
r3 = mem32[sp+12] ;  
sp += 16 ;
```

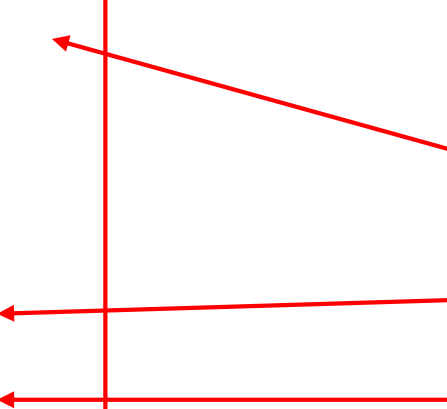


- **Push and pop commands will always order register list by register number.**
  - Lowest number accessing lowest address.
  - If you want the registers to be pushed or popped in a different order, multiple push or pop commands will be needed.

# Subroutine Calling a Subroutine

## Example

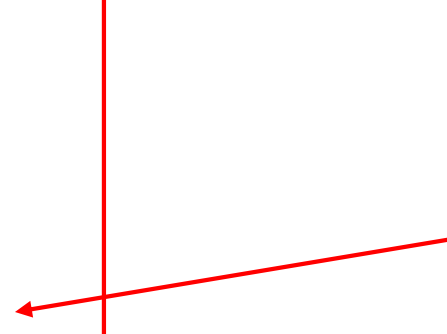
```
subr  push {lr}  
...  
bl subr2  
...  
pop {lr}  
bx lr
```



- When a subroutine calls another subroutine it needs to preserve its return address.
  - Return address is stored in LR.
- Before calling the second subroutine subr2
  - subr pushes its LR on the stack for temporary storage.
- When subr2 returns, subr pops the LR off the stack.
- When subr completes the program jumps to the return address.

## Example

```
subr  push {lr}  
...  
bl subr2  
...  
pop {pc}
```



- Alternately, you could pop LR off the stack directly into the PC.
  - When popped LR will be written into PC which will cause the program to jump to the return address.

# C Functions Register Conventions

- When passing arguments to a C function.
  - The first four arguments are written to r0 – r3.
  - Additional arguments are written to the stack. These are then popped off in the function to access them.
  - For efficiency it is recommended to keep the number of arguments in a function to 4.
- When a function returns, the return value is returned in r0 (and r1 if needed).
- A parent function is the function making the call.
- A child function is the function being called.
- r0 – r3, r12 are preserved by the parent.
  - These values are likely to be changed by the child function.
  - If the values need to be preserved, the parent pushes them to the stack before calling the child function.
- r4 – r11 are preserved by the child.
  - The child uses these registers for local variables. .
  - Before using these variables the child pushes the original values to the stack.
  - When done, the child pops them off the stack and restores the original values.
- SP / r13 is preserved in the sense that the child function should push and pop the same number of items from the stack during its operation. (Note for next page)
  - To the parent the SP should be unchanged when the function returns.
- LR / r14 must be pushed to the stack by the child before making other function calls and popped back to either LR or PC before returning. (Note for next page)
  - After pushing LR to the stack, r14 can be used for local variables.

# C Function Register Summary

Register Type	Description
Argument register	Passes arguments during a function call
Return register	Holds the return value from a function call
Expression register	Holds a value
Stack pointer	Holds the address of the top of the software stack
Link register	Contains the return address of a function call
Program counter	Contains the current address of code being executed

Register	Alias	Usage	Preserved by Function <sup>[a]</sup>
R0	A1	Argument register, return register, expression register	Parent
R1	A2	Argument register, return register, expression register	Parent
R2	A3	Argument register, expression register	Parent
R3	A4	Argument register, expression register	Parent
R4	V1	Expression register	Child
R5	V2	Expression register	Child
R6	V3	Expression register	Child
R7	V4	Expression register	Child
R8	V5	Expression register	Child
R9	V6	Expression register	Child
R10	V7	Expression register	Child
R11	V8	Expression register	Child
R12	V9, IP	Expression register, Intra-Procedure-call scratch register	Parent
R13	SP	Stack pointer	Child <sup>[b]</sup>
R14	LR	Link register, expression register	See note
R15	PC	Program counter	N/A

The parent function is responsible for preserving these, as they are likely to change in the function.

The child function is responsible for preserving these. They should appear unchanged to the parent.

Notes on previous page.

# C Function Example

## Example

```
uint32_t umax2(uint32_t x, uint32_t y) {  
    if (y > x) {  
        x = y;  
    }  
    return x;  
}
```

*Annotations:*

- r0* (points to `x`)
- r1* (points to `y`)
- Arguments are passed in *r0* – *r3*
- r0* Return values in *r0* (points to `return x;`)

```
umax2    cmp r1, r0    ; compare y-x > 0, y > x  
         ble endif1   ; if y <= x jump  
         mov r0, r1    ; x = y  
endif1  bx lr        ; return to parent.
```

- Since `x` is already in `r0`, no additional `mov` is needed to set the return value.

# C Function Example: Calling umax2

## Example

```
uint32_t umax_array(uint32_t a[],
                    uint32_t n) {
    uint32_t max = 0;
    uint32_t i;
    for (i = 0; i < n; i++) {
        max = umax2(a[i], max);
    }
    return max;
}
```

We will be using r4-r7 in the function. r4-r12 need to be preserved by the child umax\_array so we save the originals.

When calling umax2  
r0 is first argument  
r1 is second argument

Therefore, we must move the umax\_array arguments into temporary local variables.

umax\_array

```
push {r4-r7, lr}
mov r4, r0
mov r5, r1
mov r6, #0
mov r7, #0
loop1 cmp r7, r5
      bhs done1
      lsl r2, r7, #2
      ldr r0, [r4, r2]
      mov r1, r6
      bl umax2
      mov r6, r0
      add r7, r7, #1
      b loop1
done1 mov r0, r6
      pop {r4-r7, pc}
```

```
; upon entry r0 = &a, r1 = n
; save original values to stack
;, umax2 uses r0
; r4 = &a r5 = n, umax2 uses r1
; r6 = max = 0
; r7 = i = 0
; checking i < n in for unsigned
; if i >= n get out of loop
; r2 = address offset of a[i], i << 2
; r0 = a[i], copy data mem32(a[0]+offset)
; r1 = max
; call umax2 using r0 and r1 as args
; r6 = max = umax2 return value in r0
; i++ for for loop
; always jump to start of loop
; copy return value to r0 = max.
; restore r4-r7 to original values
pc = lr (jumps back to return address)
```

# CPU Performance

- CPU instruction timing information is in the ARM Cortex-M4 Technical Reference Manual.
- Each instruction will take a certain number of clock cycles, for most instructions it is one cycle.
- However there are always exceptions.
  - Instructions for multiply and divide may take longer.
  - Some operations may cause the pipeline to flush and we will need to wait for the pipeline to reload.
    - The TM4C1294 has a pipeline depth of 3 so it may take an additional 1 or 2 cycles to reload.
    - The reload time is dependent on the instruction.
    - The most common instructions to require reload cycles are branches.
  - Operations like load and store may be processing multiple addresses and so their execution time also depends on the number of addresses being accessed.

# Table of Instruction Timing

Instruction type	Clock cycles
Data operations	1 (+P <sup>a</sup> if PC is destination)
MUL	1
MLA, MLS	2
Divide	2 to 12
LDR/STR	1 or 2 <sup>b</sup> (+P <sup>a</sup> if PC is destination)
LDM/STM	1+N <sup>b</sup> (+P <sup>a</sup> if PC is destination)
Branch	1+P <sup>a</sup>

- P = pipeline reload cycles
- N = Number of registers loaded or stored

- **+P<sup>a</sup> cycles:** Branches take one clock cycle per instruction plus a pipeline reload for the target instruction.
  - Non-taken branches are 1 cycle total, P = 0.
  - Taken **branches with an immediate label** operand are normally,
    - **2 total cycles:** 1 cycle + P where P = 1.
    - P is only 1 because the branch location is determined during decode, so we just need to re-fetch the last instruction.
  - Taken **branches with register operation** are normally.
    - **3 total cycles:** 1 cycle + P where P = 2.
    - P is 2 because the branch address is determined during execution, so the decode and fetch pipeline stages need to be reloaded.
    - If the PC is the destination address, as denoted in the table above, those cases fall into this category.
- **Pipeline reloads are longer when branching to unaligned 32-bit instructions, in addition to access to slower memories.**

# Load and Store Execution

## (Note <sup>b</sup> previous page)

- Load and Store instructions are subject to these general rules.
  - STR with immediate or no offset is always 1 cycle.
  - STR with register offset is 2 cycles.
    - `str r0, [r2,r3]`
  - LDR are normally 2 cycles when done individually.
    - Multiple LDRs can be pipelined as long as the the data of the previous LDR is not the address of the next.
      - Example `ldr r2, [r3]; ldr r4 [r2]` ; can not be pipelined
    - When pipelined, the first LDR takes two 2 cycles, then the following LDRs or STR each take one cycle.
  - LDM and STM are pipelined within the instruction.
    - The multiple loads /stores in the same instruction are pipelined.
    - The number of cycles equals 1 + N (number of load/stores in the instruction).
    - LDM / STM cannot be pipelined with other load and store instructions.
  - Unaligned Word or Halfword loads or stores add penalty cycles.  
(compiler should take care of this so it is not usually a problem).
    - Halfword-aligned word or byte-aligned half word take 1 additional cycle.
    - Byte-aligned word takes 2 additional cycles.
- Memory accesses may stall with slower memory for additional cycles.

# Timing Estimation umax2

- Most operations take 1 clock cycle. We just need to look out for the special cases for multiplies, divides, branching, loads and stores.

## Example

```
uint32_t umax2(uint32_t x, uint32_t y) {  
    if (y > x) {  
        x = y;  
    }  
    return x;  
}
```

		; clock cycles per instruction
umax2	cmp r1, r0	; 1
	ble <b>endif1</b>	; 2 if taken , 1 not taken
	mov r0, r1	; 1
<b>endif1</b>	bx lr	; branch to register = 3

- Path1:  $y > x$ 
  - 1 for cmp
  - + 1 for branch not taken
  - + 1 mov
  - + 3 bx
  - Total = 6 cycles
- Path2:  $y \leq x$ 
  - 1 for cmp
  - + 2 for branch taken to endif1
  - + 3 bx
  - Total = 6 cycles
- If the totals were not the same, take the maximum.

# Timing Estimation umax array

umax\_array ; Clock Cycles **umax\_array(a, n) assuming n = 10**

push {r4-r7, lr} ; N = 5 registers, cycles 1 + N = 6

mov r4, r0 ; 1

mov r5, r1 ; 1

mov r6, #0 ; 1

mov r7, #0 ; 1

**loop1** cmp r7, r5 ; 1

bhs done1 ; branch label:  
; branch to done1 = 2,  
; if not taken 1

lsl r2, r7, #2 ; 1

ldr r0, [r4, r2] ; 2 (not pipelined)

mov r1, r6 ; 1

bl umax2 ; branch label:  
; always taken = 2 + time of umax2 6 = 8

mov r6, r0 ; 1

add r7, r7, #1 ; 1

b **loop1** ; branch label: always taken = 2

**done1** mov r0, r6 ; 1

pop {r4-r7, pc} ; N = 5 registers + PC as destination P = 2.

Cycles = 1 + N + P = 8

Final: 10 start  
+ 183 loop  
+ 9 end  
Total: 202 clock

start = 10 clocks

First 10 times does all the instructions branch to done1 not taken.

Clocks per loop = 18

10 x 18 = 180

Last time done1 branch taken.

Clocks = 1 for cmp  
+ 2 for b taken = 3

Loop1 total: 180 + 3 = 183

end = 9 clocks

# Interfacing C with Assembly

- Several assembler directives are needed to define
  - The instruction set
    - .thumb ; one used for the TM4C1294
    - .arm
  - Memory sections
    - .text ; program code stored in ROM
    - .data ; initialized non-constant objects (global variables) ROM
    - .bss ; uninitialized objects (global variables) RAM
    - .sect "section name" ; create your own section
  - Linkage
    - .global <label1>, <label2> ..... ;
    - The labels are object that the C code needs to reference like global variable and function names.
  - Data allocation and initialization
    - .byte | .char <value> {, ..., <value>}
    - .half | .short <value>{, ..., <value>}
    - .word | .long | .int <value>{, ..., <value>}
    - .cstring "text" {, ..., <text>}
    - .space <size in bytes>
    - .align <size>
  - Source debugging, lets debugger find the functions
    - .asmfunc
    - .endasmfunc

# Assembler Directive Example

## C Code equivalent

```
long x = 0;
long cur_count(long y) {
    x = x + y;
    return x;
}
```

	<b>.global</b> x, cur_count	; function and global var names
	<b>.data</b>	
x	<b>.long</b> 0	; define the global variable x = 0
	<b>.text</b>	; the program goes in text
	<b>.thumb</b>	; select thumb instruction set
cur_count	<b>.asmfunc</b>	; start your function
	ldr r1, x_addr	; r1 = address of x (PC relative addr)
	ldr r2,[r1]	; <b>r2 = x</b>
	add r0,r2	; <b>y is passed in r0, r0 = y+x</b>
	str r0,[r1]	; x = y +x
	bx lr <b>.endasmfunc</b>	; return x , end the function
x_addr	<b>.word</b> x	; x_addr = address of x