ARM ASM functions

Computer Architecture

UTEC
UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

CS3501 – 2025I
PROF.: JGONZALEZ@UTEC.EDU.PE
CWILLIAMS@UTEC.EDU.PE
SRC: HARRIS, HARRIS - DIGITAL DESIGN AND COMPUTER ARCHITECTURE

Executive Summary

- **Motivation: Programs at high-level languages can be executed in modern processor systems.**
- **Problem: We need to understand the machine code and define its relationship with programming languages constructs.**
- **Overview:**
  - **Overview of conditional statements, loops, arrays and function calls.**
  - **ARM Assembly programming for high-level constructs.**
  - **Code and execute with an ARM emulator.**
- **Conclusion: We can create complex programs using assembly language by defining correct machine code instructions.**

Resources

ARM Emulators

- **CPUlator:** https://cpulator.01xz.net/?sys=arm
  Web-based, online simulator.
- **ARM Visual:** https://salmanarif.bitbucket.io/visual/
  Windows, MacOS, Ubuntu

ARM Quick Reference Guide:
https://developer.arm.com/documentation/qrc0001/m

ARM Cheat Sheet by Uwe Zimmer:
https://cs.anu.edu.au/courses/comp2300/v_media/manuals/ARMv7-cheat-sheet.pdf

Recall: for Loops

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=1; i!=10; i=i+1)
   sum = sum + i;
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV     R0, #1     ; i = 1
  MOV     R1, #0     ; sum = 0

FOR
  CMP R0, #10        ; R0-10
  BEQ DONE           ; if (i==10)
                     ; exit loop
  ADD R1, R1, R0     ; sum=sum + i
  ADD R0, R0, #1     ; i = i + 1
  B   FOR            ; repeat loop

DONE
```

Recall: for Loops: Decremented Loops

In ARM, decremented loop variables are more efficient

C Code

```
// adds numbers from 1-9
int sum = 0


for (i=9; i!=0; i=i-1)
    sum = sum + i;
```

ARM Assembly Code

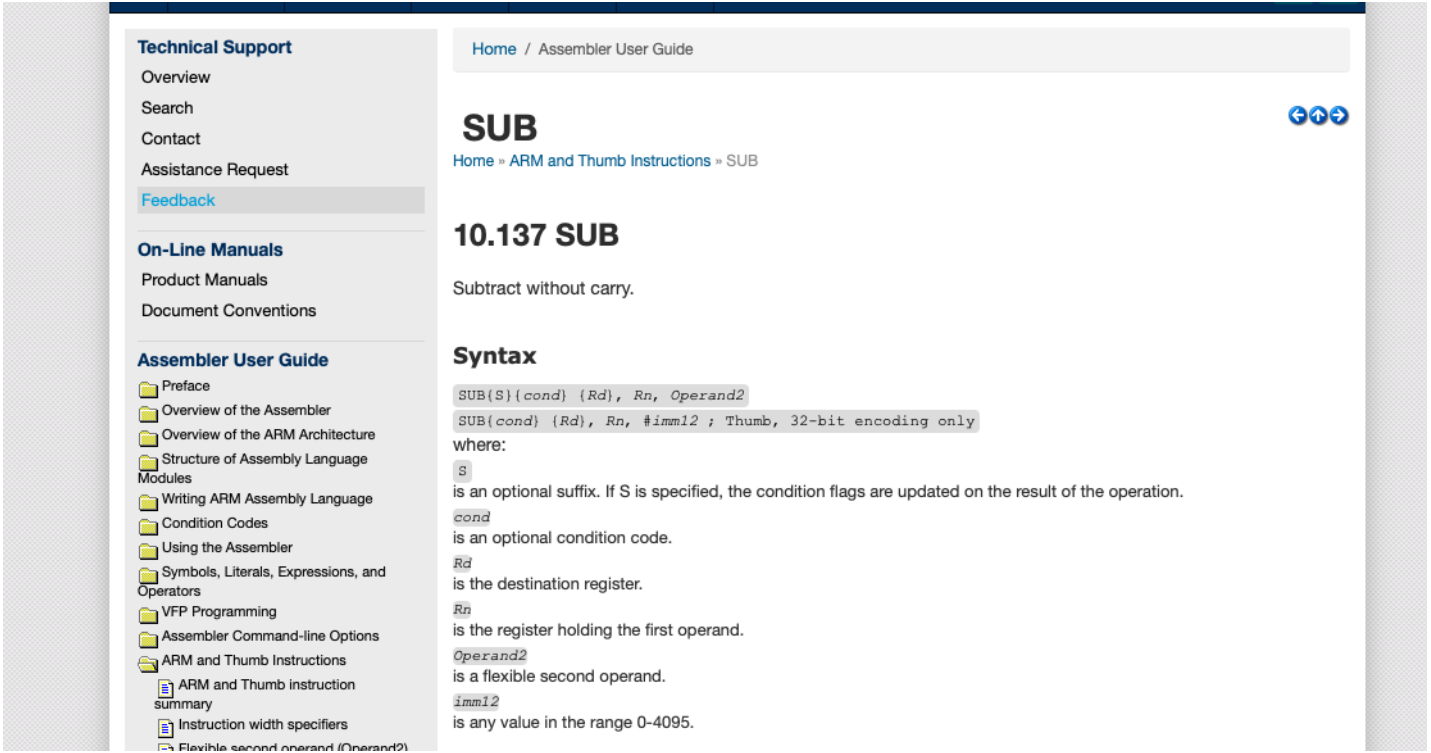
```
; R0 = i, R1 = sum
  MOV       R0, #9       ; i = 9
  MOV       R1, #0       ; sum = 0

FOR
  ADD  R1, R1, R0        ; sum=sum + i
  SUBS R0, R0, #1        ; i = i - 1
                         ; and set flags
  BNE  FOR               ; if (i!=0)

            ; repeat loop
```

Saves 2 instructions per iteration:

- Decrement loop variable & compare: `SUBS R0, R0, #1`
- Only 1 branch, instead of 2

Another ARM ISA Quickguide

https://www.keil.com/support/man/docs/armasm/armasm_dom1361289908389.htm



Technical Support
- Overview
- Search
- Contact
- Assistance Request
- Feedback

On-Line Manuals
- Product Manuals
- Document Conventions

Assembler User Guide
- Preface
- Overview of the Assembler
- Overview of the ARM Architecture
- Structure of Assembly Language Modules
- Writing ARM Assembly Language
- Condition Codes
- Using the Assembler
- Symbols, Literals, Expressions, and Operators
- VFP Programming
- Assembler Command-line Options
- ARM and Thumb Instructions
  - ARM and Thumb instruction summary
  - Instruction width specifiers
  - Flexible second operand (Operand2)

Home / Assembler User Guide

SUB

Home » ARM and Thumb Instructions » SUB

10.137 SUB

Subtract without carry.

Syntax

`SUB{S}{cond} {Rd}, Rn, Operand2`

`SUB{cond} {Rd}, Rn, #imm12 ; Thumb, 32-bit encoding only`

where:

`S`
is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

`cond`
is an optional condition code.

`Rd`
is the destination register.

`Rn`
is the register holding the first operand.

`Operand2`
is a flexible second operand.

`imm12`
is any value in the range 0-4095.

Outline

Arrays

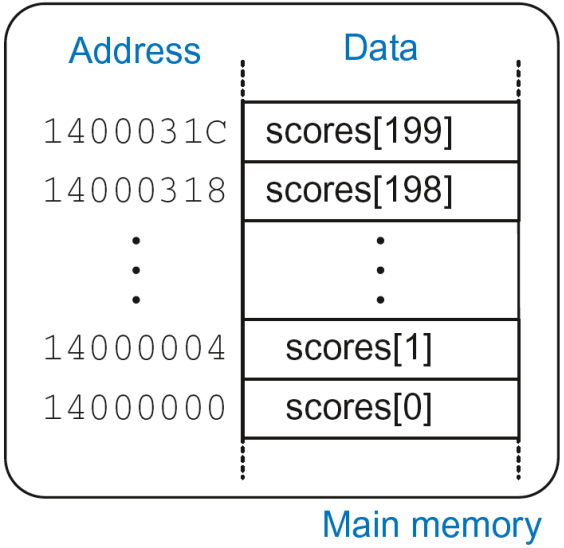
Function Calls

Conclusions

Arrays

- Access large amounts of similar data
  - **Index:** access to each element
  - **Size:** number of elements

- **Example:** 5-element array
  - **Base address** = 0x14000000 (address of first element, scores[0])
  - Array elements accessed relative to base address



Accessing Arrays

C Code

```
int array[5];
array[0] = array[0] * 8;
array[1] = array[1] * 8;
```

ARM Assembly Code

```
; R0 = array base address
  MOV R0, #0x60000000          ; R0 = 0x60000000

  LDR R1, [R0]                 ; R1 = array[0]
  LSL R1, R1, 3                ; R1 = R1 << 3 = R1*8
  STR R1, [R0]                 ; array[0] = R1

  LDR R1, [R0, #4]             ; R1 = array[1]
  LSL R1, R1, 3                ; R1 = R1 << 3 = R1*8
  STR R1, [R0, #4]             ; array[1] = R1
```

Arrays using for Loops

C Code

```
int array[200];
int i;

for (i=199; i >= 0; i = i - 1)
        array[i] = array[i] * 8;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
  MOV R0, #0x60000000
  MOV R1, #199
FOR
  LDR  R2, [R0, R1, LSL #2]; R2 = array(i)
  LSL  R2, R2, #3          ; R2 = R2<<3 = R3*8
  STR  R2, [R0, R1, LSL #2]; array(i) = R2
  SUBS R0, R0, #1          ; i = i - 1
                           ; and set flags
  BPL  FOR                 ; if (i>=0) repeat loop
```

ASCII Code and Cast of Characters

- American Standard Code for Information Interchange (ASCII)
- Each text character has unique byte value
  - For example, S = 0x53, a = 0x61, A = 0x41
  - Lower-case and upper-case differ by 0x20 (32)

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	‘	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	–	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Outline

Arrays

Function Calls

Conclusions

Function Calls

- **Caller:**
  - Passes **arguments** to callee
  - Jumps to callee
  - Example: calling function `main`
- **Callee:**
  - **Performs** the function
  - **Returns** result to caller
  - **Returns** to point of call
  - **Must not overwrite** registers or memory needed by caller
  - Example: called function `sum`

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

ARM Function Conventions

- **Call Function:** branch and link

```
BL
```

- **Return** from function: move the link register to PC:

```
MOV PC, LR
```

- **Arguments:** `R0-R3`
- **Return value:** `R0`

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

ARM Assembly Code

```
0x00000200 MAIN     BL  SIMPLE
0x00000204          ADD R4, R5, R6
...

0x00401020 SIMPLE   MOV PC, LR
```

`void` means that `simple` doesn't return a value

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

ARM Assembly Code

```
0x00000200 MAIN     BL  SIMPLE
0x00000204          ADD R4, R5, R6
...

0x00401020 SIMPLE   MOV PC, LR
```

BL branches to SIMPLE
LR = PC + 4 = 0x00000204

MOV PC, LR makes PC = LR
(the next instruction executed is at 0x00000200)

Input Arguments and Return Value

C Code

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;               // return value
}
```

Input Arguments and Return Value

ARM Assembly Code

```
; R4 = y
MAIN
  ...
  MOV R0, #2        ; argument 0 = 2
  MOV R1, #3        ; argument 1 = 3
  MOV R2, #4        ; argument 2 = 4
  MOV R3, #5        ; argument 3 = 5
  BL DIFFOFSUMS     ; call function
  MOV R4, R0                  ; y = returned value
  ...
; R4 = result
DIFFOFSUMS
  ADD R8, R0, R1    ; R8 = f + g
  ADD R9, R2, R3    ; R9 = h + i
  SUB R4, R8, R9    ; result = (f + g) - (h + i)
  MOV R0, R4        ; put return value in R0
  MOV PC, LR        ; return to caller
```

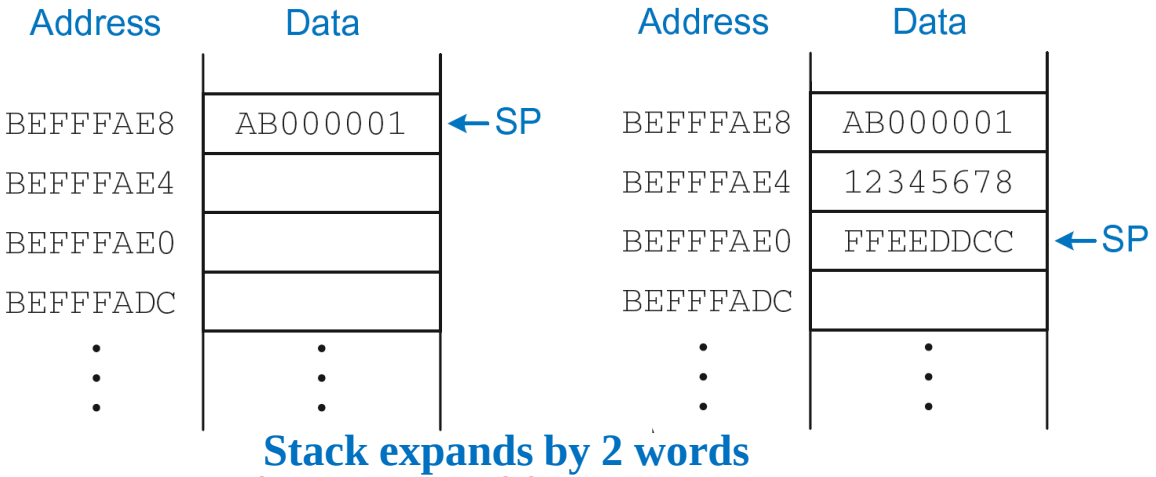
Input Arguments and Return Value

ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
  ADD R8, R0, R1    ; R8 = f + g
  ADD R9, R2, R3    ; R9 = h + i
  SUB R4, R8, R9    ; result = (f + g) - (h + i)
  MOV R0, R4        ; put return value in R0
  MOV PC, LR          ; return to caller
```

- `diffofsums` overwrote 3 registers: `R4, R8, R9`
- `diffofsums` **can use *stack*** to temporarily store registers

The Stack



- **Memory used to temporarily save variables**
- Like stack of dishes, last-in-first-out (LIFO) queue
  - **Expands:** uses more memory when more space needed
  - **Contracts:** uses less memory when the space no longer needed
- **Grows down** (from higher to lower memory addresses)
- **Stack pointer:** SP points to top of the stack

Example: Functions using the Stack

- Called functions must have no unintended side effects
- But **diffofsums overwrites 3 registers**: R4, R8, R9

ARM Assembly Code

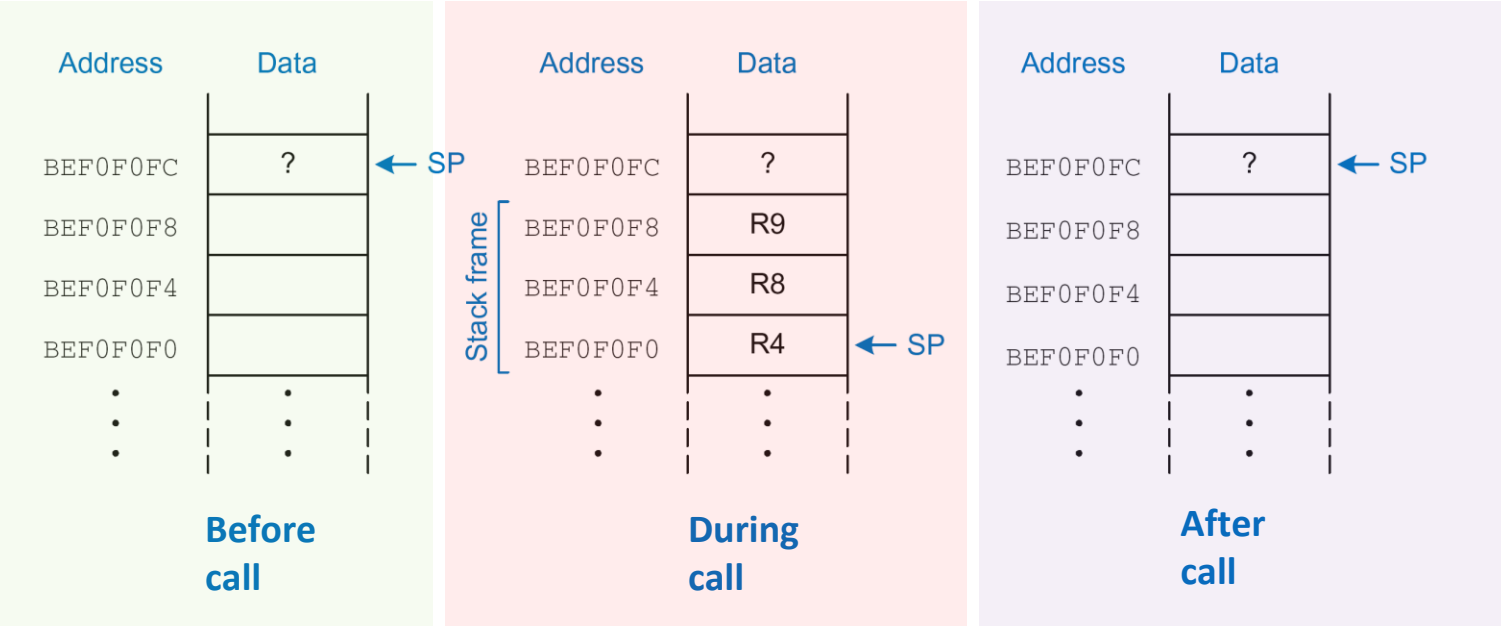
```
; R4 = result
DIFFOFSUMS
  ADD R8, R0, R1     ; R8 = f + g
  ADD R9, R2, R3     ; R9 = h + i
  SUB R4, R8, R9     ; result = (f + g) - (h + i)
  MOV R0, R4         ; put return value in R0
  MOV PC, LR          ; return to caller
```

Storing Register Values on the Stack

ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
  SUB SP, SP, #12  ; make space on stack for 3 registers
  STR R4, [SP, #8] ; save R4 on stack
  STR R8, [SP, #4] ; save R8 on stack
  STR R9, [SP]               ; save R9 on stack
  ADD R8, R0, R1   ; R8 = f + g
  ADD R9, R2, R3   ; R9 = h + i
  SUB R4, R8, R9   ; result = (f + g) - (h + i)
  MOV R0, R4       ; put return value in R0
  LDR R9, [SP]               ; restore R9 from stack
  LDR R8, [SP, #4] ; restore R8 from stack
  LDR R4, [SP, #8] ; restore R4 from stack
  ADD SP, SP, #12  ; deallocate stack space
  MOV PC, LR                 ; return to caller
```

Stack during `diffofsums` Call



Registers

Preserved *Callee-Saved*	Nonpreserved *Caller-Saved*
R4-R11	R12
R14 (LR)	R0-R3
R13 (SP)	CPSR
stack above SP	stack below SP

Storing Saved Registers only on Stack

ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
  STR R4, [SP, #-4]!          ; save R4 on stack
  ADD R8, R0, R1    ; R8 = f + g
  ADD R9, R2, R3    ; R9 = h + i
  SUB R4, R8, R9    ; result = (f + g) - (h + i)
  MOV R0, R4        ; put return value in R0
  LDR R4, [SP], #4 ; restore R4 from stack
  MOV PC, LR                  ; return to caller
```

Pre-index

Post-index

Notice code optimization for expanding/contracting stack

Nonleaf Function

ARM Assembly Code

```
STR LR, [SP, #-4]!     ; store LR on stack
BL  PROC2                    ; call another
function
...
LDR LR, [SP], #4       ; restore LR from stack
jr  $ra                ; return to caller
```

Nonleaf Function Example

C Code

```
int f1(int a, int b) {
  int i, x;
  x = (a + b)*(a - b);
  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}
int f2(int p) {
  int r;
  r = p + 5;
  return r + p;
}
```

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
  PUSH {R4,  R5, LR}
  ADD   R5,  R0, R1
  SUB   R12, R0, R1
  MUL   R5,  R5, R12
  MOV   R4,  #0
FOR
  CMP   R4, R0
  BGE   RETURN
  PUSH {R0, R1}
  ADD   R0, R1, R4
  BL    F2
  ADD   R5, R5, R0
  POP  {R0, R1}
  ADD   R4, R4, #1
  B     FOR
RETURN
  MOV   R0, R5
  POP  {R4, R5, LR}
  MOV   PC, LR
```

```
; R0=p, R4=r
F2
  PUSH {R4}
  ADD   R4, R0, 5
  ADD   R0, R4, R0
  POP  {R4}
  MOV   PC, LR
```

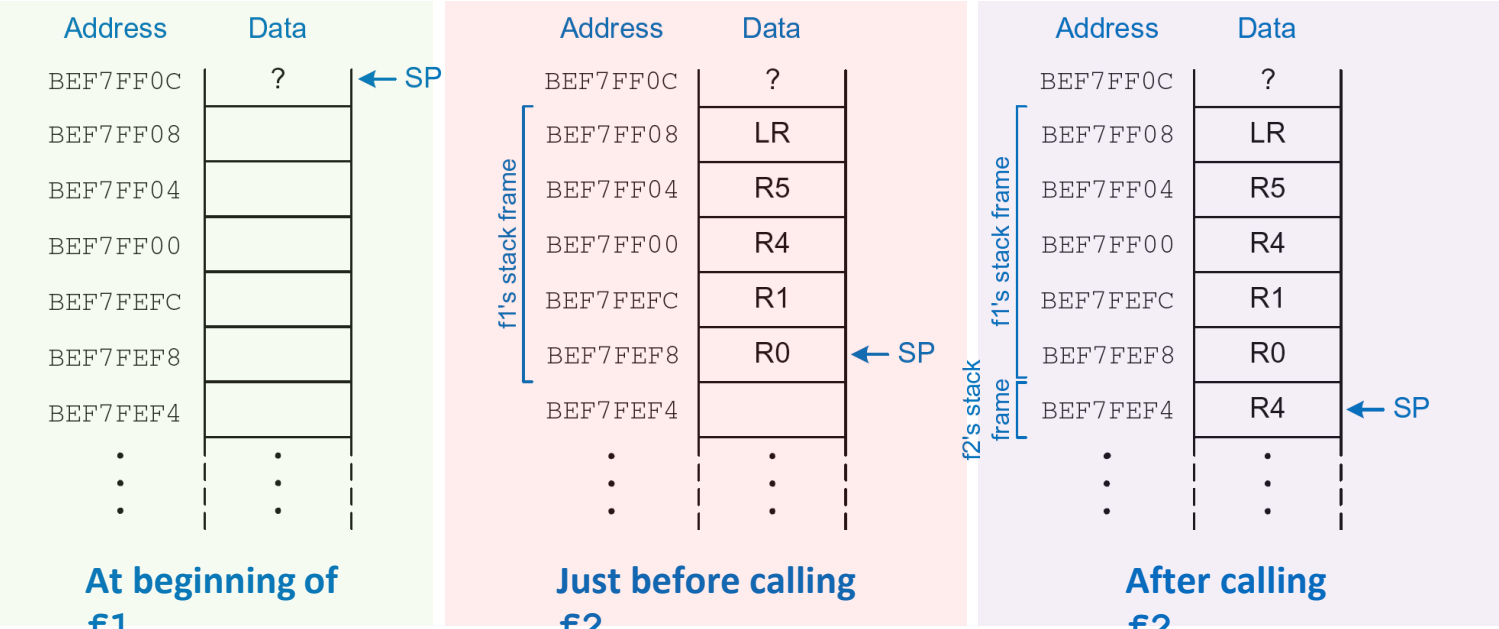
Nonleaf Function Example

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
  PUSH {R4,  R5, LR} ; save regs
  ADD   R5,  R0, R1  ; x = (a+b)
  SUB   R12, R0, R1  ; temp = (a-b)
  MUL   R5,  R5, R12 ; x = x*temp
  MOV   R4,  #0      ; i = 0
FOR
  CMP   R4, R0       ; i < a?
  BGE   RETURN       ; no: exit loop
  PUSH {R0, R1}      ; save regs
  ADD   R0, R1, R4   ; arg is b+i
  BL    F2           ; call f2(b+i)
  ADD   R5, R5, R0   ; x =x+f2(b+i)
  POP  {R0, R1}      ; restore regs
  ADD   R4, R4, #1   ; i++
  B     FOR          ; repeat loop
RETURN
  MOV   R0, R5       ; return x
  POP  {R4, R5, LR}  ; restore regs
  MOV   PC, LR       ; return
```

```
; R0=p, R4=r
F2
  PUSH {R4}          ; save regs
  ADD   R4, R0, 5    ; r = p+5
  ADD   R0, R4, R0   ; return r+p
  POP  {R4}          ; restore regs
  MOV   PC, LR       ; return
```

Stack during Nonleaf Function



Recursive Function Call

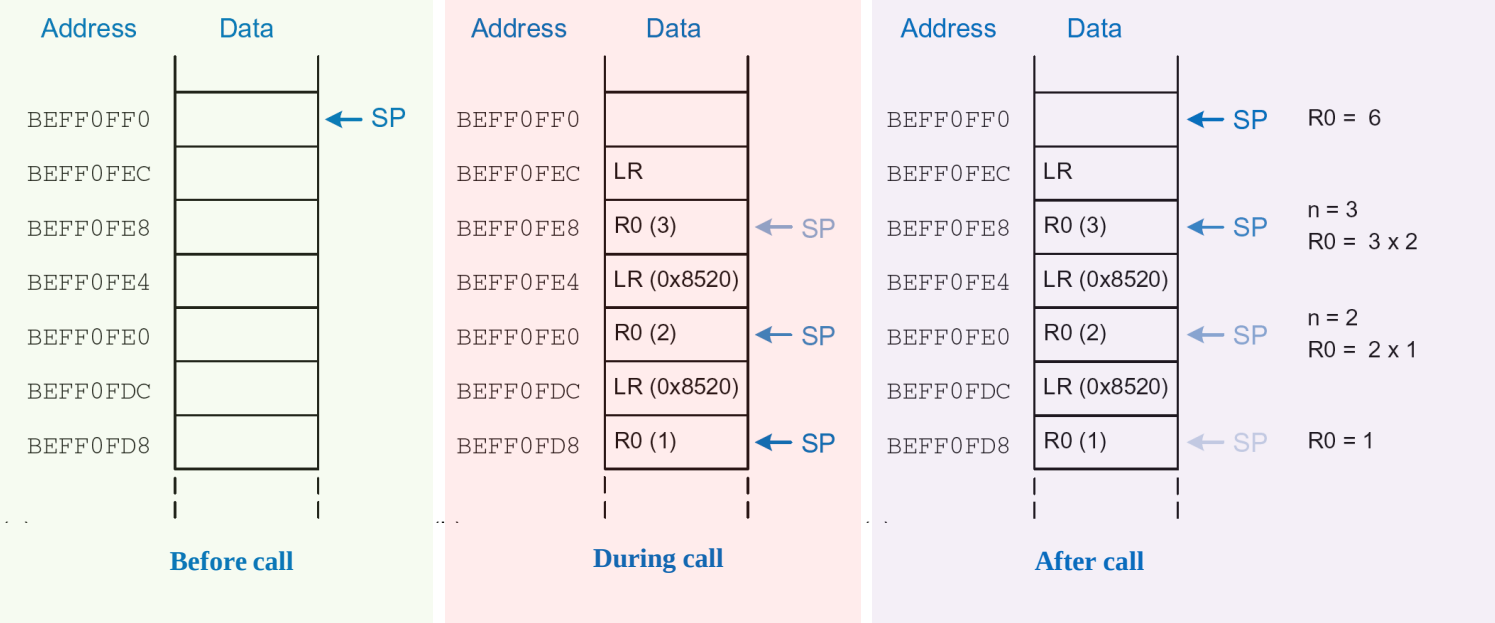
C Code

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```

ARM Assembly Code

```
0x94 FACTORIAL  STR R0, [SP, #-4]!  ;store R0 on stack
0x98            STR LR, [SP, #-4]!  ;store LR on stack
0x9C            CMP R0, #2                    ;set flags with R0-2
0xA0            BHS ELSE                      ;if (r0>=2) branch to else
0xA4            MOV R0, #1                    ; otherwise return 1
0xA8            ADD SP, SP, #8      ; restore SP 1
0xAC            MOV PC, LR                    ; return
0xB0 ELSE       SUB R0, R0, #1      ; n = n - 1
0xB4                 BL  FACTORIAL            ; recursive call
0xB8            LDR LR, [SP], #4    ; restore LR
0xBC            LDR R1, [SP], #4    ; restore R0 (n) into R1
0xC0            MUL R0, R1, R0      ; R0 = n*factorial(n-1)
0xC4            MOV PC, LR                    ; return
```

Stack during Recursive Call



Summary: Function Call

- **Caller**
  - Puts arguments in `R0-R3`
  - Saves any needed registers (`LR`, maybe `R0-R3, R8-R12`)
  - Calls function: `BL CALLEE`
  - Restores registers
  - Looks for result in `R0`
- **Callee**
  - Saves registers that might be disturbed `(R4-R7)`
  - Performs function
  - Puts result in `R0`
  - Restores registers
  - Returns: `MOV PC, LR`

Outline

Arrays

Function Calls

Conclusions

Conclusions

- **We reviewed fundamentals concepts in programming languages constructs.**
- **We reviewed** assembly implementation for conditional statements, loops, arrays and function calls.
- **We coded and executed high-level constructs using ARM syntax and instructions.**
- We conclude that **complex programs have a direct implementation in machine code that allows to execute them in the processor.**

ARM ASM functions

Computer Architecture

UTEC
UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

CS3501 – 2025 I
PROF.: JGONZALEZ@UTEC.EDU.PE
CWILLIAMS@UTEC.EDU.PE