CSE 331: Microprocessor Interfacing and Embedded Systems

Assembly Language

Procedure

https://cpulator.01xz.net/?sys=arm-de1soc

Why Subroutines?

- Avoid code repetition
- Improve modularity
- Make code more readable & maintainable

Have you ever written the same block of code more than once?
Think of subroutines as reusable Lego blocks in your program.

What is a Subroutine?

- A separate block of code performing a specific task
- Called from multiple locations
- Execution returns to the point it was called

```
BL my_function ; Branch with Link
BX LR ; Return from subroutine
```

Basic Subroutine Example

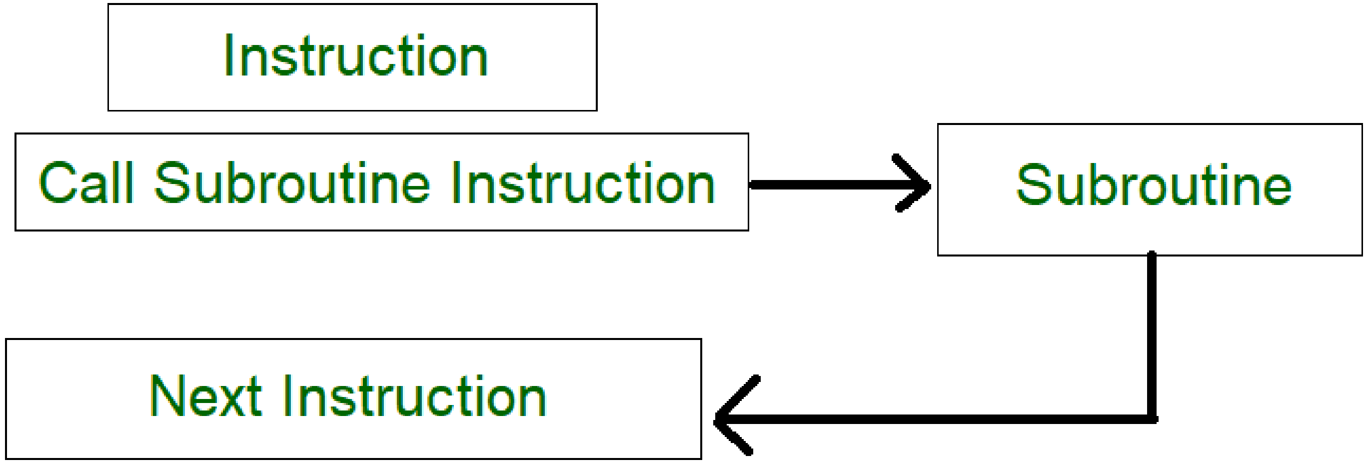
```
main:
    MOV R0, #5
    BL square
    ; R0 now contains 25
    B end

square:
    MUL R0, R0, R0
    BX LR

end:
    ; halt or loop
```

Registers and Return Addresses

- LR （R14）holds return address
- BL stores PC+4 into LR
- BX LR jumps to the address in LR



Source: geeks for geeks

Subroutines with Parameters

ARM Calling Convention (AAPCS):
- R0–R3 for parameters
- R0 for return value

```
main:
 MOV R0, #3
  MOV R1, #4
 BL multiply
  ; R0 now = 12
 B end

multiply:
 MUL R0, R0, R1
 BX LR
```

Register Clobbering!

- **Problem**: Subroutines may modify registers that the main code still needs.
- **Solution**: Save/restore registers using the stack.

```
PUSH {R4, LR}
 ; subroutine code
POP {R4, LR}
BX LR
```

What could go wrong if we forget to save LR?

Subroutine with Stack Saving

```
multiply:
    PUSH {R4, LR}
    MOV R4, R1
    MUL R0, R0, R4
    POP {R4, LR}
    BX LR
```

```
multiply:
    PUSH {R4, LR}
    MOV R4, R1
    MUL R0, R0, R4
    POP {R4, PC}
```

Subroutine with Stack Saving

C:

```
int square(int x) {
   return x * x;
}

int main() {
   int a = square(3);
   int b = square(5);
   while (1);  // Infinite loop
to halt
}
```

Assembly:

```
      LDR    R0, =3       ; First call
      BL     square
      STR    R0, [R2]     ; Simulate print: store result at R2

      LDR    R0, =5       ; Second call
      BL     square
      STR    R0, [R3]     ; Simulate print: store result at R3

stop:   B      stop

square:
      MUL    R0, R0, R0
      BX     LR          ; RETURN to caller
```

Subroutine with Stack Saving

- **What will happened If You Forget BX LR?**
  - The subroutine doesn't return.
  - Execution continues into unknown memory or next label → leads to a **crash or undefined behavior**.

- **What will happen If You Clobber LR**

```
square:
    MOV    LR, #0        ; Overwriting LR
    MUL    R0, R0, R0
    BX     LR            ; Now jumping to address 0 → likely a crash
```

- LR stores the return address.
- Overwriting it before returning causes execution to jump to wrong address.

Subroutine Returning Multiple Values

C

```
void divmod(int a, int b, int* quotient, int* remainder)
{
   *quotient = a / b;
   *remainder = a % b;
}

int main() {
   int q, r;
   divmod(10, 3, &q, &r);
   while(1); // halt
}
```

Assembly

```
MOV R0, #10 ; a = 10
MOV R1, #3 ; b = 3
BL divmod ;

Stop: B stop ; infinite loop
divmod:
      PUSH   {R4, R5, LR}
      MOV    R4, #0        ; R4 = quotient
      MOV    R5, R0        ; R5 = dividend
loop_div:
      CMP    R5, R1
      BLT    end_div       ; if dividend < divisor → done
      SUB    R5, R5, R1    ; R5 -= divisor
      ADD    R4, R4, #1    ; quotient++
      B      loop_div
end_div:
      MOV    R0, R4        ; return quotient in R0
      MOV    R1, R5        ; return remainder in R1
      POP    {R4, R5, PC}
```

Overview of AAPCS

AAPCS defines:

- How **arguments** are passed to functions
- Where **return values** are placed
- Which **registers must be preserved**
- How to **use the stack**

It ensures that functions can interact **consistently**, even when compiled separately or in different languages (e.g., C, assembly, etc.).

Overview of AAPCS

Register Classification

Register	Use
R0–R3	Argument passing and return values
R4–R11	Callee-saved (must be preserved by subroutine)
R12 (IP)	Intra-procedure call scratch register
R13 (SP)	Stack Pointer
R14 (LR)	Link Register (return address)
R15 (PC)	Program Counter

Overview of AAPCS

Function Arguments (Inputs)

- Passed in registers **R0 to R3**
- Additional arguments (5th onward) are passed on the **stack**
- If a struct or array is passed by value, it may go on the stack or be split across R0–R3 and stack

Tip: If you're calling a function with 6 arguments, only the first 4 go into R0–R3, and the rest go to stack.

Overview of AAPCS

Return Values

- Single return value → **R0**
- Two return values → **R0, R1**
- For structs/unions > 4 bytes, a pointer to memory is passed in R0 (caller allocates memory)

Overview of AAPCS

Register Preservation Rules

- **Callee-saved** (must be preserved by the function):
  `R4–R11`, `LR, SP`
  ➤ `If used, the function must` **save & restore** them (usually with PUSH/POP)
- These values must be saved and restored Inside the function

- **Caller-saved** (can be overwritten):
  `R0–R3`, `R12`
  ➤ `Caller must back them up if needed after function call`
- `This values must be saved from where the function is being called`

Excercise

- Write a function that takes an address of an array as input and returns the sum of all elements of the array. [assume all the arrays are terminated with a negative number]

Recursive Functions

- A recursive function is a function that calls itself directly or indirectly.

Recursive Function	Iterative Function

<pre>int factorial(int n) {
    if(n==1)
        return 1;
    else
        return n * factorial(n-1);
}
 	
int main(void){
    int y;
    y = factorial(5);
    return 0;
}</pre>
	<pre>int factorial(int n) {
	    result = 1;
	    for (int i = 1; i < n; i++)
	        result *= i;
	    return result;
	}
	
	int main(void){
	    int y;
	    y = factorial(5);
	    return 0;
	}</pre>

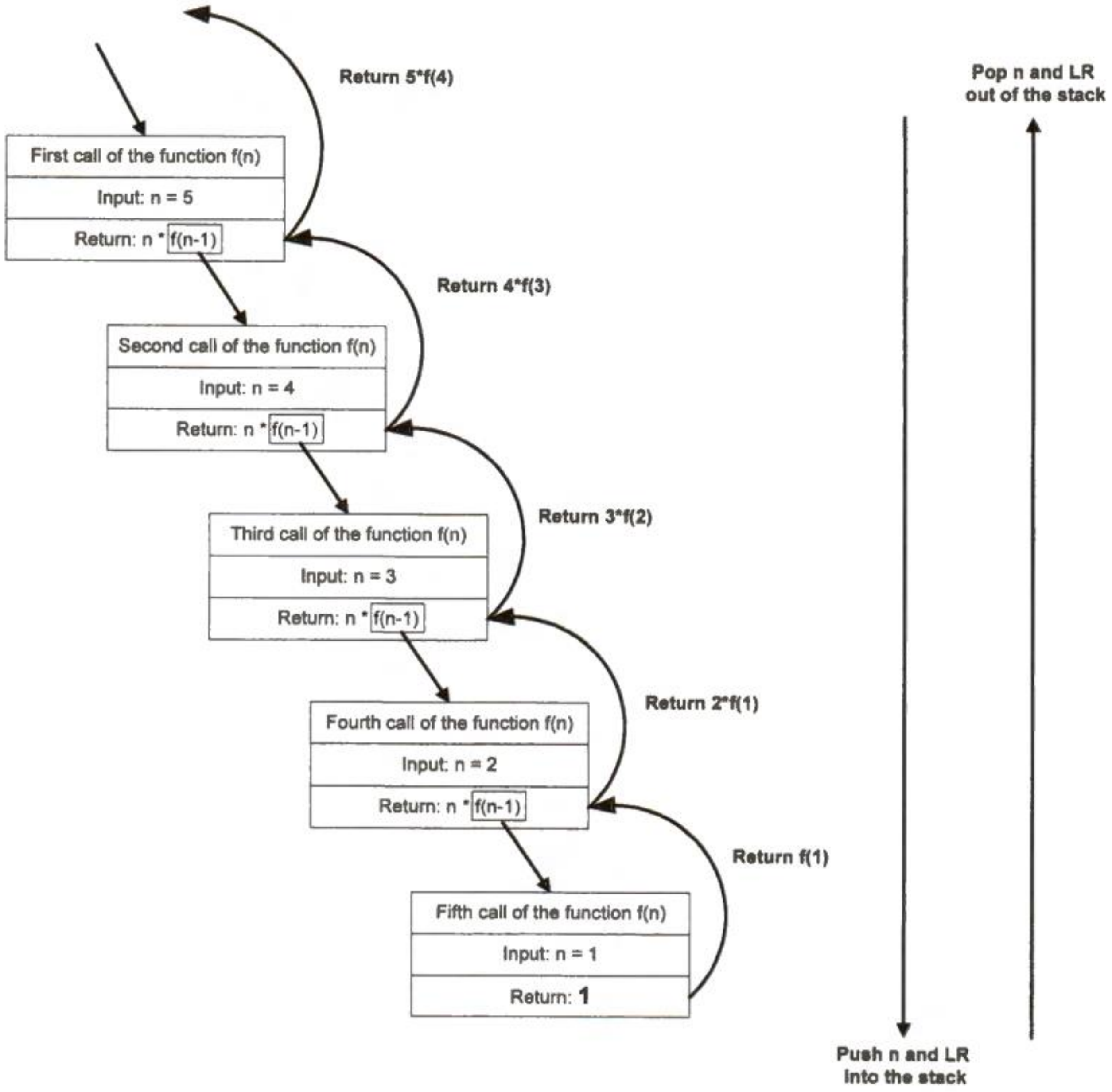


Figure 8-17. Call graph of the recursive factorial function

Address	Assembly Program
	`AREA main, CODE, READONLY`
	`EXPORT __main`
	`ENTRY`
	`__main PROC`
0x0800012E	`MOV r0, #5`
0x08000130	`BL factorial`
0x08000134	`stop B stop`
	`ENDP`
	`factorial PROC`
0x08000136	`PUSH {r4, lr} ; preserve`
0x08000138	`MOV r4, r0 ; r4 = n`
0x0800013A	`CMP r4, #1`
0x0800013C	`BNE else ; if n ≠ 1`
0x0800013E	`MOV r0, #1 ; f = 1`
0x08000140	`loop POP {r4, pc} ; return`
0x08000142	`else SUB r0, r4, #1 ; n - 1`
0x08000144	**`BL factorial`** `; r0 is input`
0x08000148	**`MUL r0, r4, r0`** `; n*f(n-1)`
0x0800014C	`B loop`
	`ENDP`
	`END`

Memory Address	Memory Content
0x20000600	
0x200005FC	0x08000134 (LR)
0x200005F8	0 (r4)
0x200005F4	0x08000148 (LR)
0x200005F0	5 (r4)
0x200005EC	0x08000148 (LR)
0x200005E8	4 (r4)
0x200005E4	0x08000148 (LR)
0x200005E0	3 (r4)
0x200005DC	0x08000148 (LR)
0x200005D8	2 (r4)
0x200005D4	0x08000148 (LR)
0x200005D0	

Stack content immediately after factorial (1) completes.