13 Subroutines
CSC 230

Department of Computer Science
University of Victoria

Stallings chapters 12,13 (skip Intel portions)
M&H: chapter 4 translated from ARC
ARM Manual

REVIEW

What is a subroutine?

- A subroutine is a portion of code to perform a specific task
- It can be relatively independent of the remaining code
- Named : ***function, procedure*** (***similar to a method***)
- All programming languages have support for subroutines, for ***calls*** to them and ***returns*** from them
- **Subroutine** can be seen as the more general label for functions or procedures or methods
- Some programming languages make very little syntactic distinction between functions and procedures.

What is the difference between functions and procedures?

REVIEW

The ideal goal

FUNCTION

- has a single return value
- has no side effects on input parameters

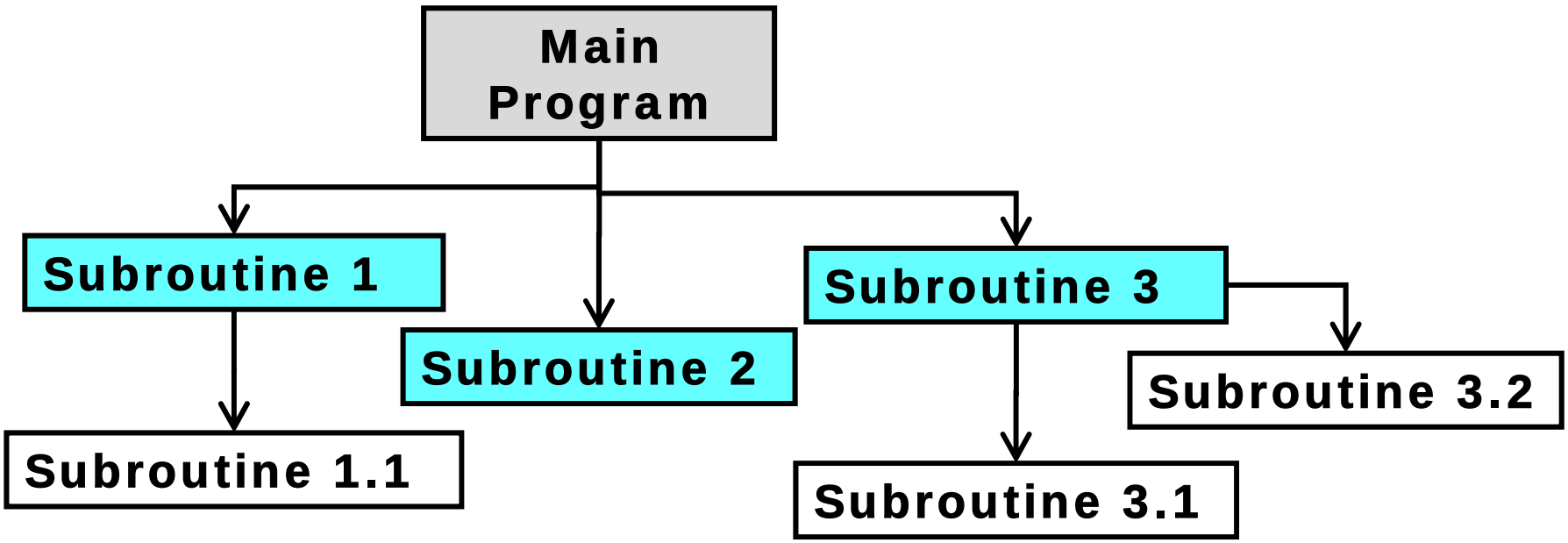
PROCEDURE

- has no directly returned result
- side effects on parameters are allowed

C does not truly distinguish between them (a procedure is a function with a void result).

Aim for the above, justify logically to yourself why you absolutely need to break the paradigm (i.e. when a 'function' has side-effects)

Subroutines

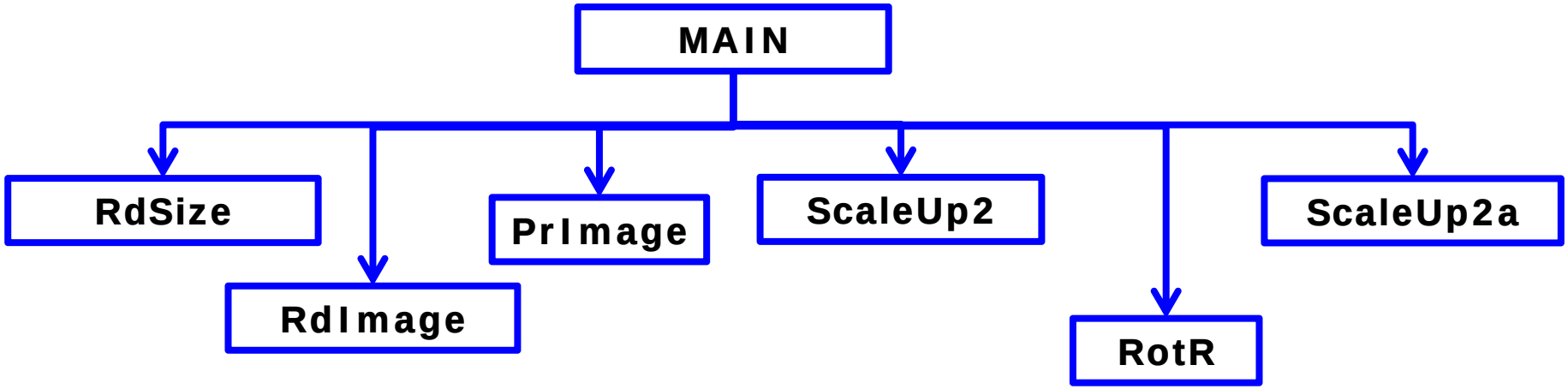


- Good program design is based on the concept of MODULARITY – the partitioning of programming code into subroutines / modules
- Start with a simple main program - outline the logical flow / steps of algorithm
- Assign execution details to subroutines
- Subroutine: general name for “function” or “procedure”

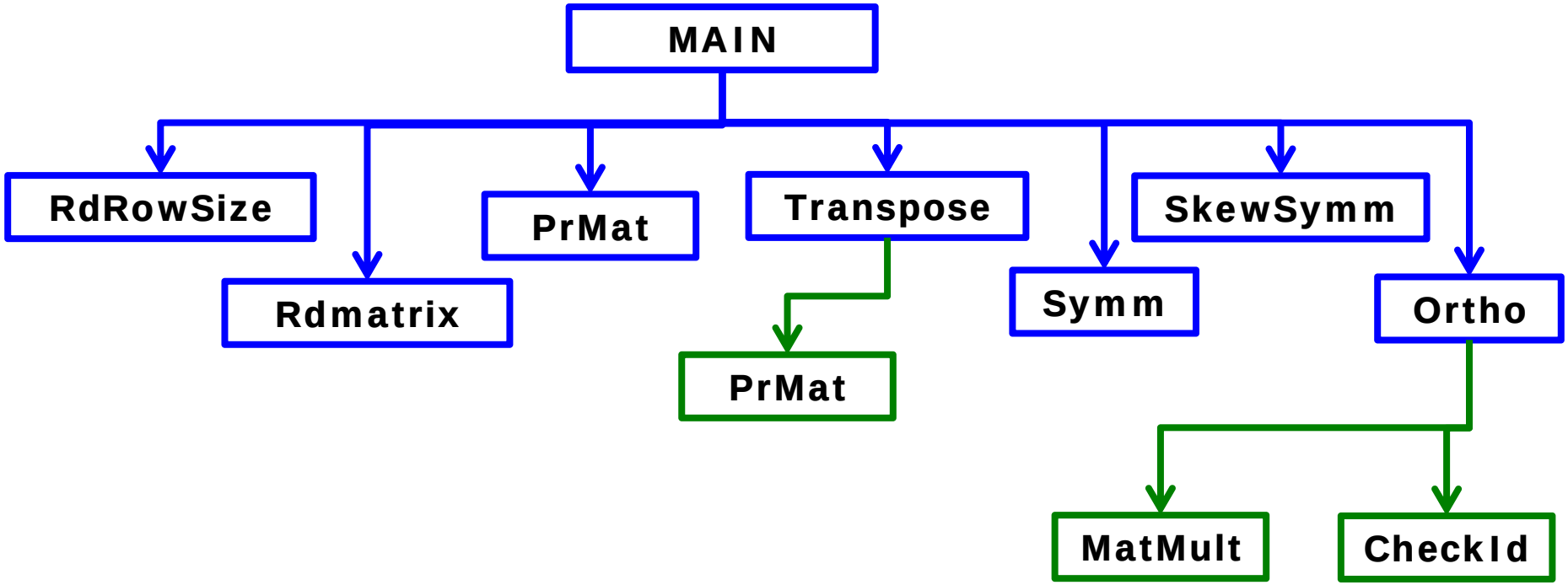
What is a Call Graph?

- A call graph shows the connections between subroutines.
- It shows which subroutines can call which other subroutines.
- It does not show the logic of operations.
- It does not show the interface specifications (e.g. the parameters being passed or the results returned).
- Its purpose is to display clearly the expected structure and modularization of the code.

Example: Call Graph for answer to Assignment 1 (C code)

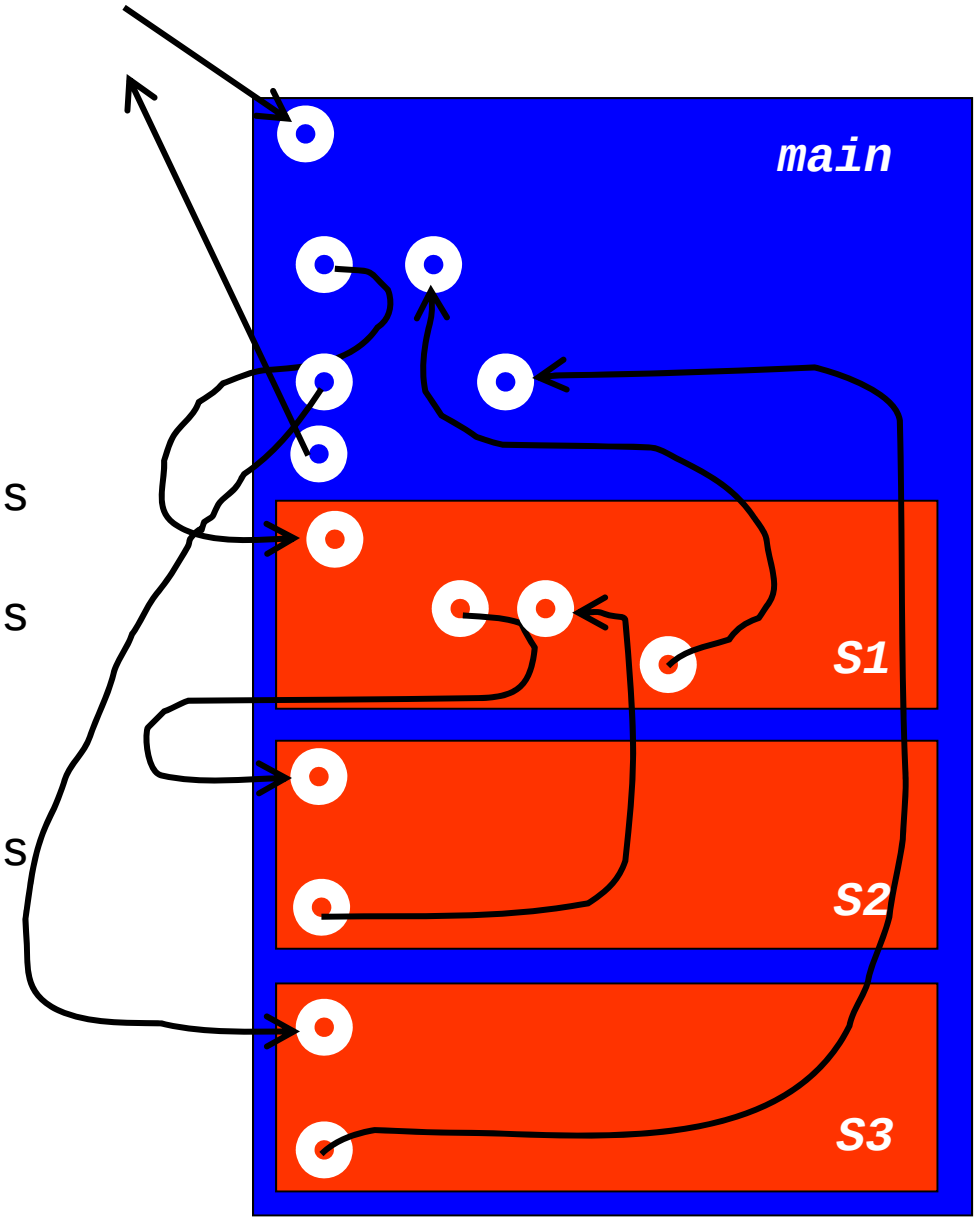


Example: Call Graph for answer to Assignment 1 (C code)



Execution

- main starts
- main calls S1
- S1 calls S2
- S2 ends and returns
- S1 ends and returns
- main calls S3
- S3 ends and returns
- main ends and returns to OS

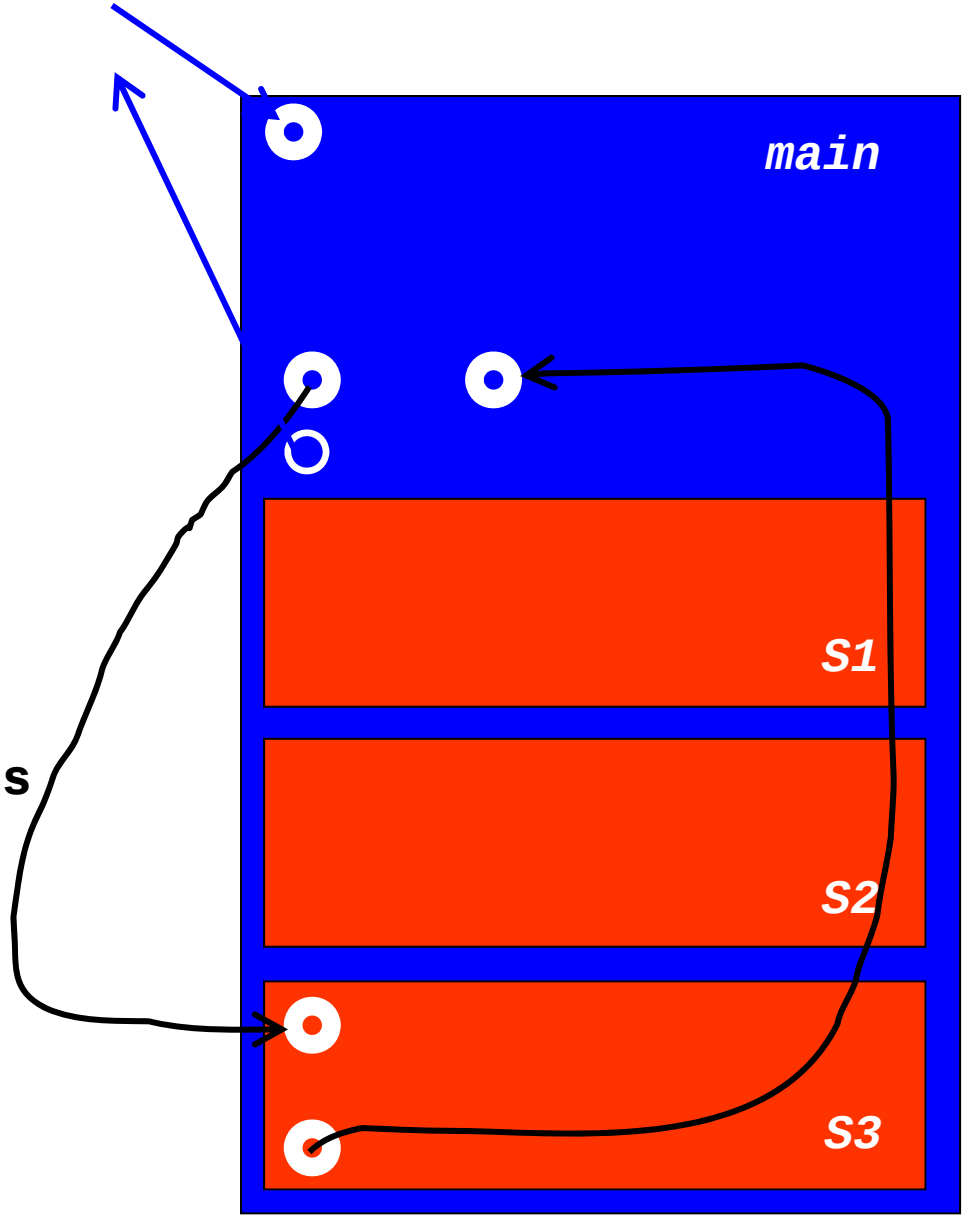


Execution: the main concept of “call” and “return”

main starts

main ends and returns to OS

- **main calls S3**
- **S3 ends and returns**



Must keep track of where to return

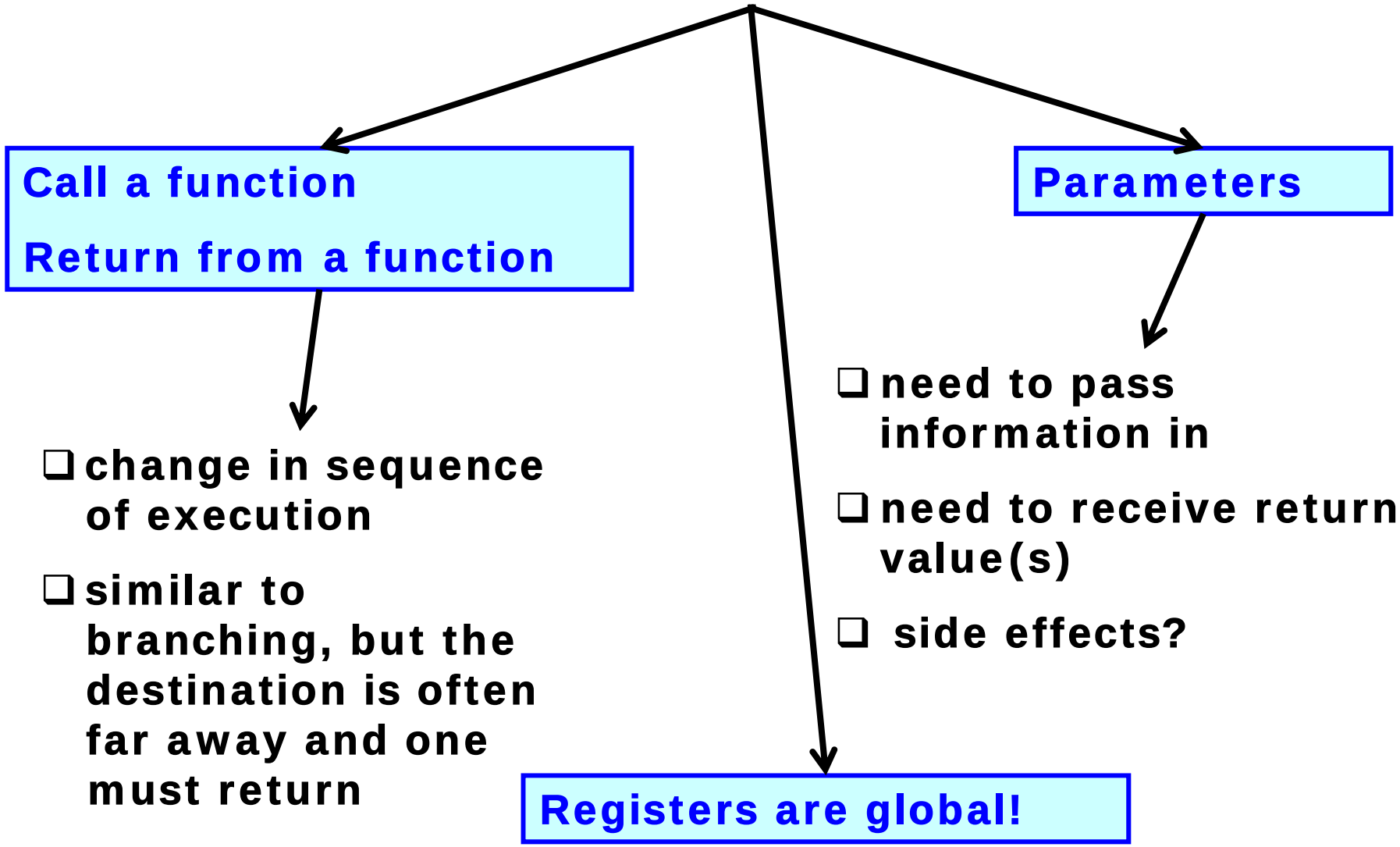
```
_start:
   BL Subr1
   next1: . . .
. . . . . . . .
   BL Subr2
   next2: . . .
. . . . . . . .
   BL Subr3
   next3: . . .
Endprogr:exit

Subr1:
. . . . . . . .
endSubr1:return

Subr2:
. . . . . . . .
endSubr2:return

Subr3:
. . . . . . . .
endSubr3:return
```

- **Program is constructed as one or more subroutines**
- **Main has label _start in ARM assembly language**
- **Labels are global, visible throughout the whole program**

Programming conventions

- Functions must be written in one piece
- Functions do not share code with each other

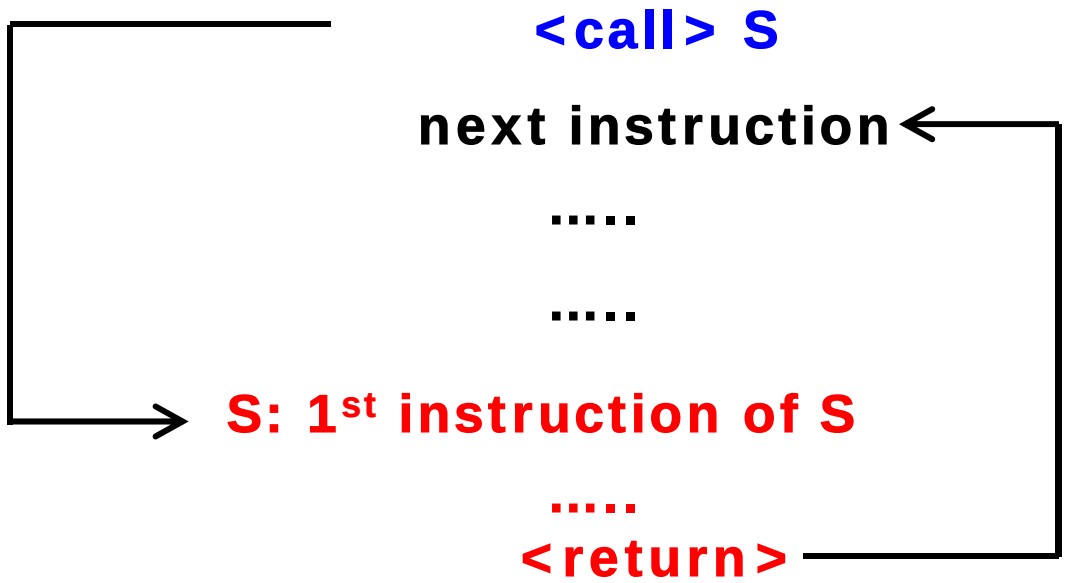
Main issues for functions



- change in sequence of execution
- similar to branching, but the destination is often far away and one must return

- need to pass information in
- need to receive return value(s)
- side effects?

Function Calls and Returns



- Function: sequence of instructions stored in memory at specified address
- Can be called from several places in program
- From the call point, address of next instruction is saved ➔ program control passes to 1st instruction of subroutine
- When finished, subroutine ends with return instruction which brings control back to previously saved one

Subroutine linkage using a link register

Code where the call happens

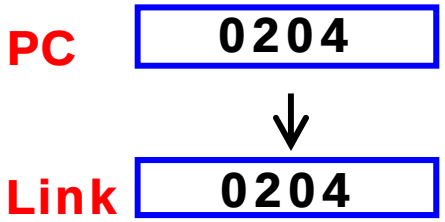
Memory location	Instruction
0200	Call Subr1
0204	Next instruction

Subroutine Subr1

Memory location	Instruction
1000	First instruction of Subr1
.....	.....
20C8	Return from Subr1 (last instruction)

[1] PC=0200
➔ Fetch Call Subr1 instruction
➔ PC=0204

[2] Link =PC=0204 ➔
remember return point



Call phase

Subroutine linkage using a link register

Code where the call happens

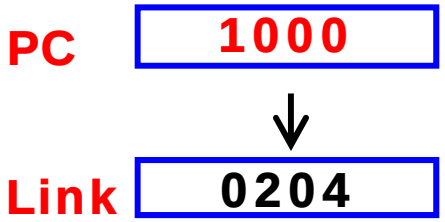
Subroutine Subr1

Memory location	Instruction	Memory location	Instruction
0200	Call Subr1	1000	First instruction of Subr1
0204	Next instruction	.....	
		20C8	Return from Subr1 (last instruction)

[1] PC=0200 ➔ Fetch Call Subr1 instruction
PC=0204

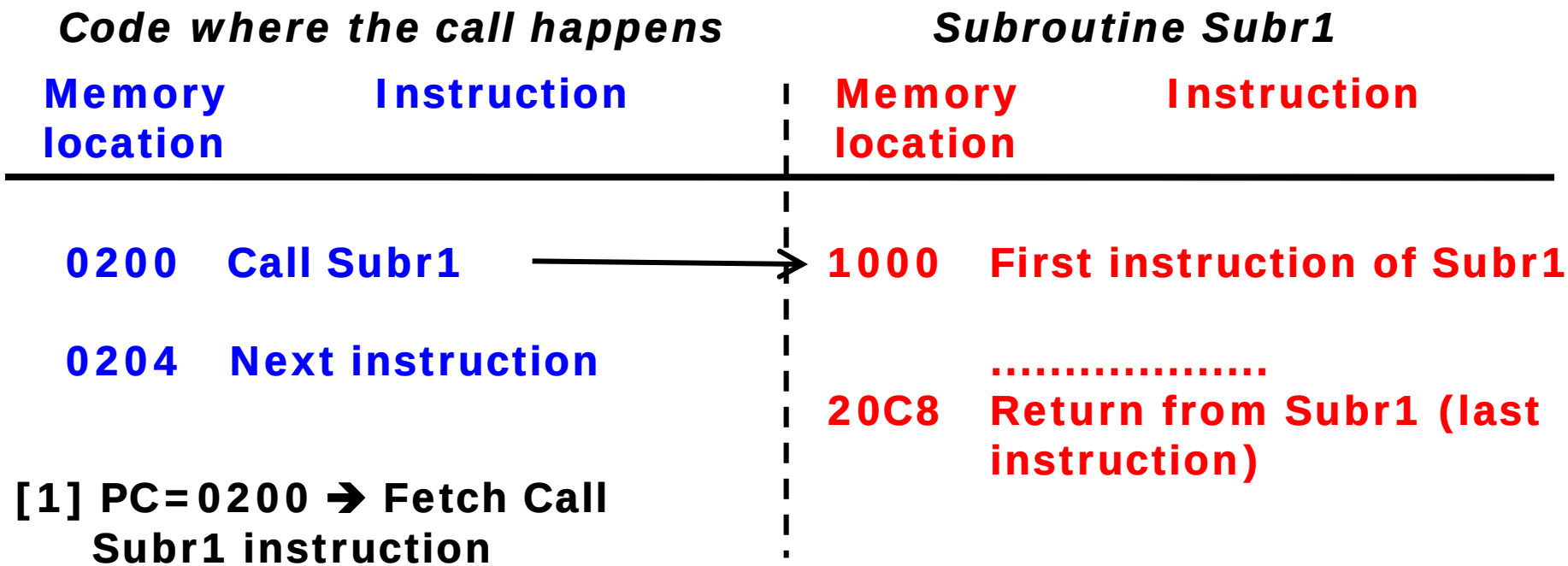
[2] Link =PC=0204 ➔ remember return point

[3] PC=1000 ➔ execute Subr1



Call phase

Subroutine linkage using a link register



[1] PC=0200 ➔ Fetch Call Subr1 instruction
PC=0204

[2] Link =PC=0204 ➔ remember return point

[3] PC=1000 ➔ execute Subr1

[4] PC=0204 ➔ copy from Link and return

	Call phase	Return phase
PC	1000	20C8
Link	0204	0204

Function calls in ARM

BL *label* **branch and link to *label* at start of subroutine**

R14 **is the Link Register**

BL **(1)saves address of next instruction in R14 (LR)**

(2)sends execution to 1st instruction of subroutine, *that is*, PC ← address of subroutine

MOV PC,LR

it must be the last instruction when returning from the subroutine ;

it restores PC to contents of LR, *that is*,

PC ← address of instruction to return to

what if a function calls another function? what happens to the PC and link register?

Parameters: 0 input, 0 output

```
int main() {
        printInit();
        getInput();
        compute();
        printClosing();
        return 0;
}
void printInit() {
        print ("HELLO\n");
}
void getInput() {
        read data
}
void compute() {
        do something
}
void printClosing() {
        printf("Arrivederci\n");
}
```

Parameters: 1 input, 0 output

```
int main() {
    read(n);
    printSquare(n);
    return 0;
}

void printSquare(int x) {
    int temp;
    temp = x * x;
    printf("Sqr(%d) = %d\n",
        x, temp);
}
```

```
@ Call subroutine with no parameters to clear all user
@ registers
        .text
        .global _start
_start:
        bl      clearall        @ Call subroutine: "clearall"
exit:   swi     0x11            @ Terminate the program

clearall:
        mov     r0,#0
        mov     r1,#0
        mov     r2,#0
        mov     r3,#0
        mov     r4,#0
        mov     r5,#0
        mov     r6,#0
        mov     r7,#0
        mov     r8,#0
        mov     r9,#0
        mov     r10,#0
        mov     r11,#0
        mov     r12,#0
        mov     pc,lr           @ Return from the subroutine
        .end
```

Parameters: 0 input, 0 output

Parameters: 2 inputs, 0 output

```
int main() {
    int i;
    i = 5;
    power(2,i);
    return 0;
}

/* raise x to the n-th power */
void power(int x, int n) {
    int i, p;
    p = 1;
    for(i=1; i<=n; ++i)
        p = p*x;
    printf("2**%d = %d", n, p);
}
```

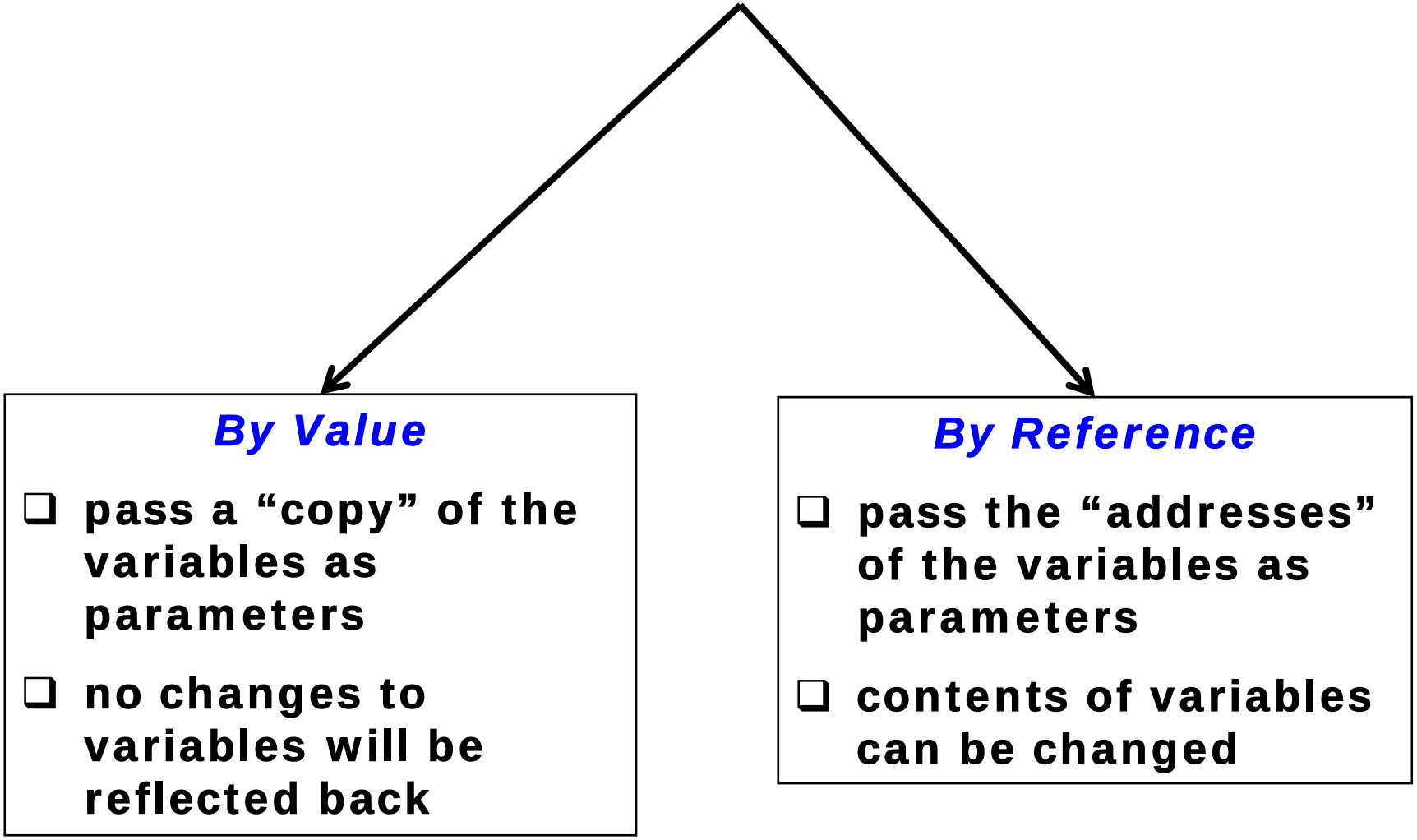
Function Call with parameters: 2 inputs, 1 output

```
int power(int, int);

int main() {
   int i, k;
   i = 5;
   k = power(2,i);
   printf("2**%d = %d\n", i, k);
   return 0;
}

/* raise x to the n-th power */
int power(int x, int n) {
   int p = 1;
   while(n>0) {
      p = p*x;  --n;
   }
   return p;
}
```

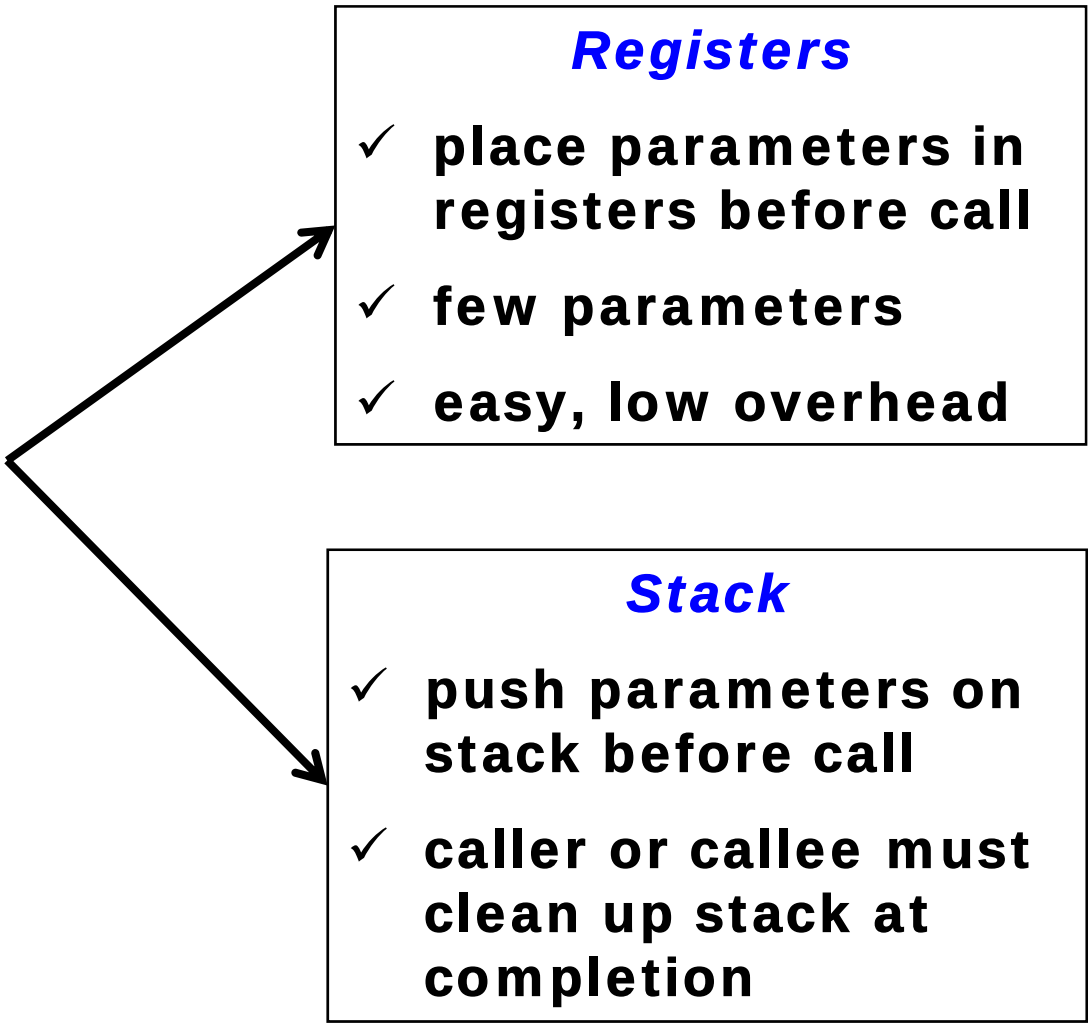
Note: the input parameters do not change in value

Better design – why? (because "power" should simply compute its function and let "main" take care of other tasks like printing)

Passing Input Parameters



Passing Input Parameters in Assembly Languages



```
@ Sum of the first "MAX" Fibonacci with subroutine
        .text
        .global _start
        .equ    MAX,10
_start:         mov     r1,#MAX
                bl      Fib     @SumFib:r0 = Fib(MAX:r1)
                @r0 will have return parameter ➔ store it
exit:           swi     0x11            @ Terminate the program

 @Subroutine to compute sum of n Fibonacci numbers
 @ SumFib:r0 =Fib(n:r1)
 Fib:   sub     r1,r1,#2        @ Counter - 2
        mov     r2,#1           @ PrevFib = 1
        mov     r3,#2           @ CurrFib = 2
        mov     r0,#3           @ TotSum = 3
 Body:  add     r4,r2,r3        @ NextFib = PrevFib + CurrFib
        add     r0,r0,r4        @ Update Sum
        mov     r2,r3           @ PrevFib = CurrFib
        mov     r3,r4           @ CurrFib = NextFib
 Decr:  subs    r1,r1,#1        @ Count = Count -1
        bne     Body            @ If Count != 0, repeat loop
 DoneFib:       mov     pc,lr           @ Return from subroutine
```

DOCUMENTATION!!!

Returning a Result

- Return the result in a register, usually R0
- Result may be a value or an address
- Normal for function calls where there is single output parameter
- Usual for C language where input parameters to a functions are ALWAYS passed by value as default
- Must document well the interface for parameters passing from call to subroutine

No side effects

Input parameters should not be changed ➔ no side effects

What about the registers used in subroutine?

What if they are also used by caller routine?

➔ MUST save the “state” of computation

➔ MUST save the computational registers used locally

➔ ***MUST save the "state" of computation***

➔ ***MUST save the computational registers used locally***

what exactly does it imply?

- **at the entrance of every function, one must copy the contents of *all registers which will be used locally* in the function**
- **use only registers in functions for storage – allocation of variables is normally only done in main routine**
- **at the exit of every function, copy back the original contents of all registers which were used locally, so that after returning to the caller routine **nothing** has changed, except for the return value (in R0), if any**

➔ **THERE ARE SPECIAL INSTRUCTIONS TO HELP OUT**

Load and Store Multiple Registers

STM*xx* Rn!,{list of registers to be stored}
LDM*xx* Rn!,{list of registers to be loaded}

- **Rn is base register containing starting address of location in memory where loading or storing will occur from**
- **List of registers to be stored or loaded, any subset of the 16 registers**
- **The '!' notation indicates that the base register is updated. This is necessary when STM and LDM are used as push and pop operations.**

Load and Store Multiple Registers

STM*xx* Rn!,{list of registers to be stored}
LDM*xx* Rn!,{list of registers to be loaded}

What do they do? (semantics)

STM*xx* Rn!,{list of registers to be stored}

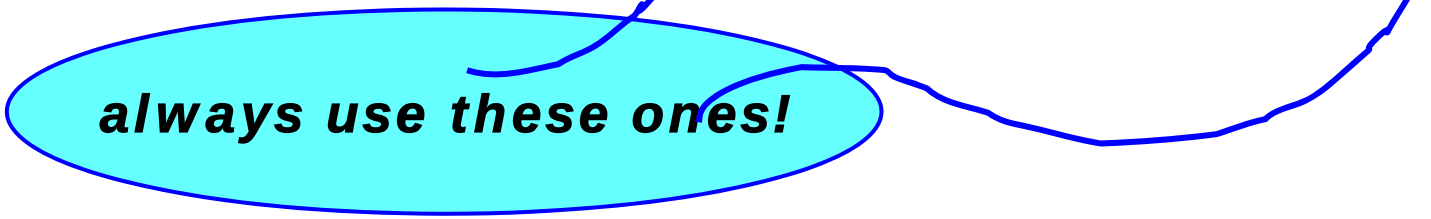
Copy to memory, starting at the address currently contained in register Rn, the contents of all the registers in the given list and update Rn

LDM*xx* Rn!,{list of registers to be stored}

Copy from memory, starting at the address currently contained in register Rn, the contents of all the registers in the given list and update Rn

STMxx and LDMxx: the full list

Meaning	Using stack	General form
Pre-increment Load	LDMED	LDMIB
Post-increment Load	LDMFD	LDMIA
Pre-decrement Load	LDMEA	LDMDB
Post-decrement Load	LDMFA	LDMDA
Pre-increment Store	STMFA	STMIB
Post-increment Store	STMEA	STMIA
Pre-decrement Store	STMFD	STMDB
Post-decrement Store	STMED	STMDA



Use of Load & Store Multiple in Subroutines

- Registers are used inside subroutine
- Must save them on entry, restore them on
- Overall "state" of processor must be … ➔ no effects
- Only changes through parameters

how did I choose R13?

Store and then reload all registers

Easier later!

```
Fib:   STMFD   r13!,{r1-r10,lr}
       sub     r1,r1,#2        @ Counter - 2
       mov     r2,#1           @ PrevFib = 1
       mov     r3,#2           @ CurrFib = 2
       mov     r0,#3           @ TotSum = 3
Body:  add     r4,r2,r3        @ NextFib = PrevFib + CurrFib
       add     r0,r0,r4        @ Update Sum
       mov     r2,r3           @ PrevFib = CurrFib
       mov     r3,r4           @ CurrFib = NextFib
Decr:  subs    r1,r1,#1        @ Count = Count -1
       bne     Body            @ If Count != 0, repeat loop
Done:  LDMFD r13!,{r1-r10,pc}
       @ Reloading into pc causes return from subroutine
```

Use of Load & Store Multiple in Subroutines

- Registers are used inside subroutine
- Must save them on entry, restore them on exit
- Overall "state" of processor must be saved ➔ no side effects
- Only changes through parameters

Only store and reload the registers used locally in subroutine

```
Fib:   STMFD   r13!,{r1,r2,r3,r4,lr}
       sub     r1,r1,#2        @ Counter - 2
       mov     r2,#1           @ PrevFib = 1
       mov     r3,#2           @ CurrFib = 2
       mov     r0,#3           @ TotSum = 3
Body:  add     r4,r2,r3        @ NextFib = PrevFib + CurrFib
       add     r0,r0,r4        @ Update Sum
       mov     r2,r3           @ PrevFib = CurrFib
       mov     r3,r4           @ CurrFib = NextFib
Decr:  subs    r1,r1,#1        @ Count = Count -1
       bne     Body            @ If Count != 0, repeat loop
Done:  LDMFD r13!,{r1,r2,r3,r4,pc}
       @ Reloading into pc causes return from subroutine
```

A few interim answers

List of registers used locally in subroutine

```
Fib:   STMFD   r13!,{r1,r2,r3,r4,lr}
       sub     r1,r1,#2        @ Counter = 2
       mov     r2,#1           @ PrevFib = 1
       mov     r3,#2           @ CurrFib = 2
       mov     r0,#3           @ TotSum = 3
Body:  add     r4,r2,r3        @ NextFib = PrevFib + CurrFib
       add     r0,r0,r4        @ Update Sum
       mov     r2,r3           @ PrevFib = CurrFib
       mov     r3,r4           @ CurrFib = NextFib
Decr:  subs    r1,r1,#1        @ Count = Count -1
       bne     Body            @ If Count != 0, repeat loop
Done:  LDMFD r13!,{r1,r2,r3,r4,pc}
       @ Reloading into pc causes return from subroutine
```

ALL registers used locally in the subroutine should be copied temporarily to memory (on the stack) and then re-copied back at exit to guarantee no side effects

STM and LDM instructions are designed to help for this purpose

A few interim answers


```
Fib:   STMFD   r13!,{r1,r2,r3,r4,lr}
       sub     r1,r1,#2        @ Counter = 2
       mov     r2,#1           @ PrevFib = 1
       mov     r3,#2           @ CurrFib = 2
       mov     r0,#3           @ TotSum = 3
Body:  add     r4,r2,r3        @ NextFib = PrevFib + CurrFib
       add     r0,r0,r4        @ Update Sum
       mov     r2,r3           @ PrevFib = CurrFib
       mov     r3,r4           @ CurrFib = NextFib
Decr:  subs    r1,r1,#1        @ Count = Count -1
       bne     Body            @ If Count != 0, repeat loop
Done:  LDMFD r13!,{r1,r2,r3,r4,pc}
       @ Reloading into pc causes return from subroutine
```

➔*Need to copy Link Register = R14 as well, since it could be changed again if there is a nested call (see later)*
➔*By substituting PC in the list when re-copying back at exit, the effect is that the Link register content are copied to the PC*
➔ *saves having to issue explicitly the MOV PC, LR*

A few interim answers

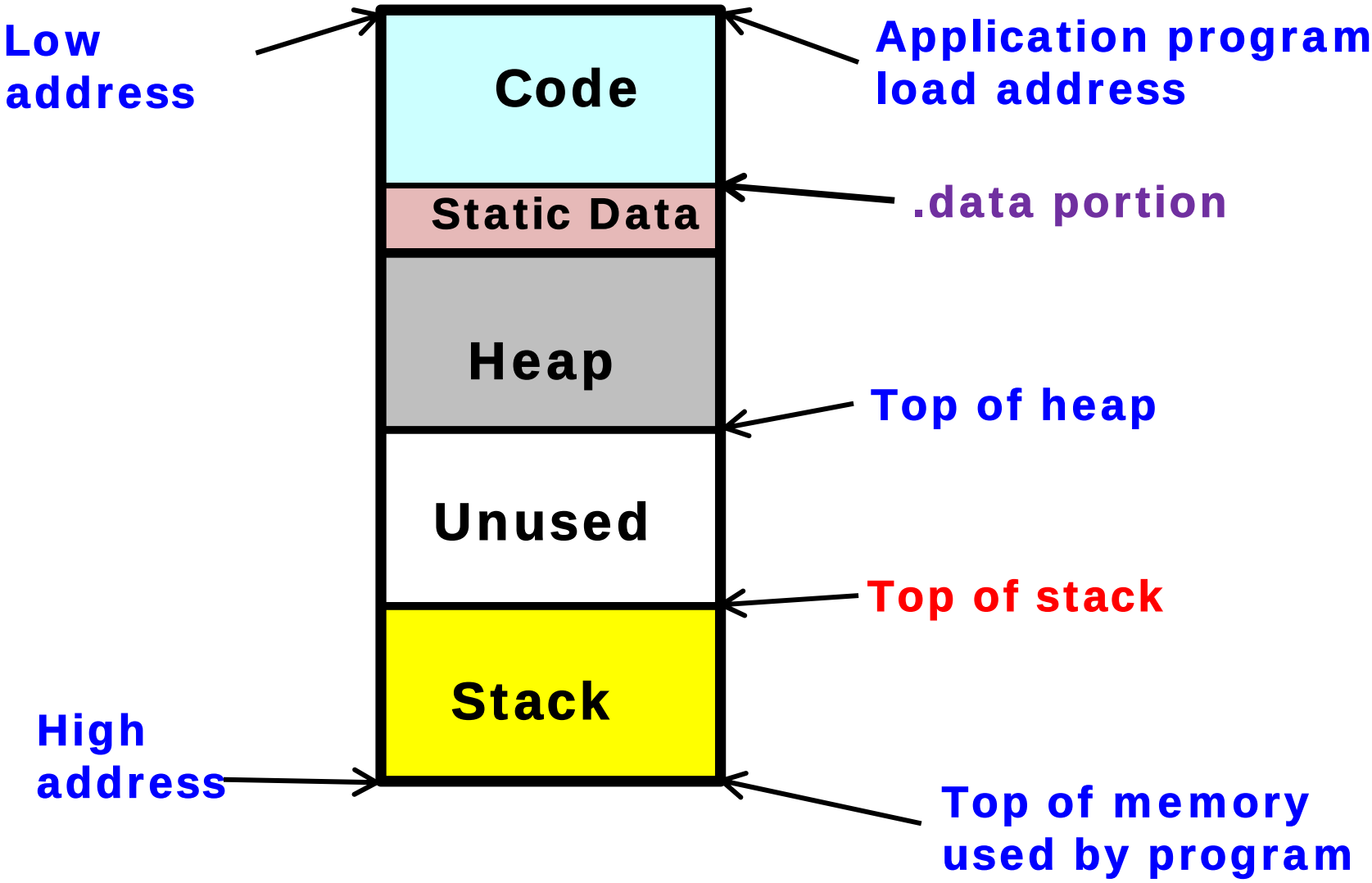
```
Fib:   STMFD   r13!,{r1,r2,r3,r4,lr}
       sub     r1,r1,#2        @ Counter - 2
       mov     r2,#1           @ PrevFib = 1
       mov     r3,#2           @ CurrFib = 2
       mov     r4,#3           @ TotSum = 3
Body:  add     r4,r2,r3        @ NextFib = PrevFib + CurrFib
       add     r0,r0,r4        @ Update Sum
       mov     r2,r3           @ PrevFib = CurrFib
       mov     r3,r4           @ CurrFib = NextFib
Decr:  subs    r1,r1,#1        @ Count = Count -1
       bne     Body            @ If Count != 0, repeat loop
Done:  LDMFD   r13!,{r1,r2,r3,r4,pc}
       @ Re   ing into pc causes return from subroutine
```

how did I choose R13?

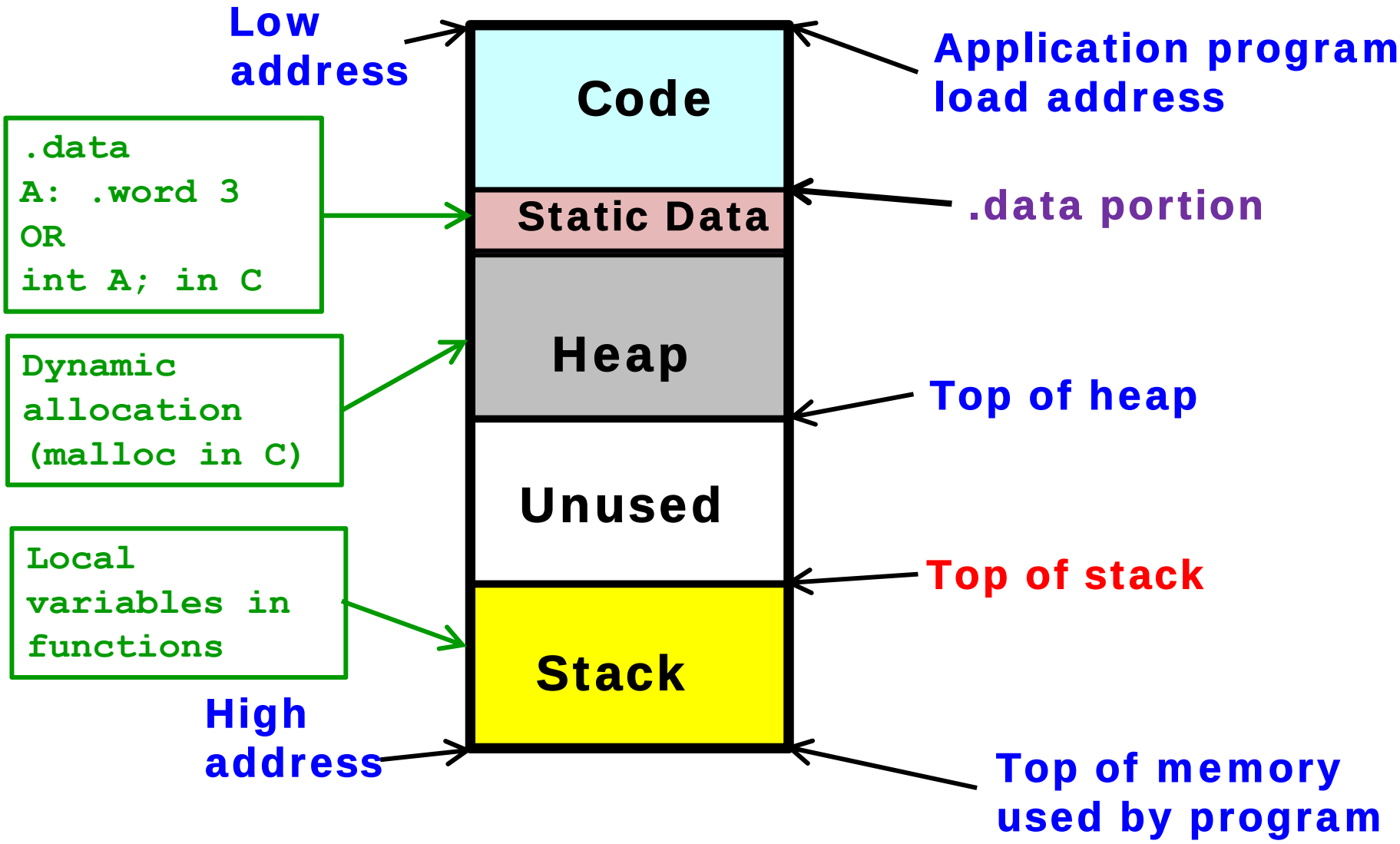
R13 = SP ➔ stack pointer

➔There is a system stack, in memory, pre-allocated, useful for exactly this calls
➔ coming up.....

Memory Configuration



Where is storage allocated within the program space?



Load and Store Multiple Registers

STM*xx* Rn,{list of registers to be stored}
LDM*xx* Rn,{list of registers to be loaded}

- **use them in pairs**
- **there are variations, both for Load and for Store, with differing semantics**

WAIT! --- DO NOT MEMORIZE !

LDMIA	**Load Multiple Increment After**
LDMIB	**Load Multiple Increment Before**
LDMDA	**Load Multiple Decrement After**
LDMDB	**Load Multiple Decrement Before**
STMIA	**Store Multiple Increment After**
STMIB	**Store Multiple Increment Before**
STMDA	**Store Multiple Decrement After**
STMDB	**Store Multiple Decrement Before**

Examples (read only and check ARM Manual)

Assume R13 = 0000 9020 (some location in memory)

STMDB R13!,{R6,R1-R3}

stores registers R1,R2,R3,R6, in this order, to memory, from lowest address to highest address

- **order in list is irrelevant (they get reordered)**
- **DB means "decrease before", i.e. decrease address by 4*4 before storing each register. (0000 9020 – 4*4 = 0000 9010)**
- **R1 is stored at address 0000 9010**
- **R2 is stored at address 0000 9014**
- **R3 is stored at address 0000 9018**
- **R6 is stored at address 0000 901C**
- **and R13 at the end is 0000 9010**

READ ONLY

STMDB R13,{R1-R3,R6}
has same effect but leaves R13 unchanged (at 0000 9020)

Examples (read only and check ARM Manual)

Assume R13 = 0000 9010 (an address of a location in memory) ➔ after the last example, but it could be anything

LDMIA R13!,{R1-R3,R6}

- **loads registers R1,R2,R3,R6 in this order from memory, from lowest address to highest address**
- **order in list is irrelevant**
- **IA means "increment after", i.e. increment address by 4*4 after loading the registers**
- **R1 is loaded from address 0000 9010**
- **R2 is loaded from address 0000 9014**
- **R3 is loaded from address 0000 9018**
- **R6 is loaded from address 0000 901C**

READ ONLY

- **R13 at the end is 0000 9020 = 901C + 4**

LDMIA R13,{R1-R3,R6}
does the same thing but R9 is not changed at the end

Examples (read only and check ARM Manual)

LDMDA R11!, {R1-R7}

➔loads list of registers and decrement after

loads Registers r1, r2, r3, r4, r5, r6, r7 from memory locations

[R11] - 28 ➔ for R1
[R11] - 24 ➔ for R2
[R11] - 20 ➔ for R3
[R11] - 16 ➔ for R4
[R11] - 12 ➔ for R5
[R11] - 8 ➔ for R6
[R11] - 4 ➔ for R7

Then it decrements R11 by 28

READ ONLY

LDMDA R11, {R1-R7}

then R11 is not changed after execution

Think it through

- **Parameters can be passed to a subroutine using registers. But how many registers can really be available? What if the processor architecture does not have many registers?**
- **The return address after calling a subroutine can be stored in a Link Register. But what happens when there are calls within calls?**
- **Registers are used locally and their state at subroutine entry must be preserved and restored at exit, so that the calling program can proceed with the same assumptions.**
  - **What if one needs to allocate variables locally?**

The 3 important steps

1. **Clear documentation on the Call instruction (input/output parameters)**

2. **Clear documentation at entry point in subroutines**

3. **Store local registers at entry and re-Load all local registers at exit (including LR → PC)**

In a high level language, the call itself and the function header can be quite self-documenting – not the case here

Example 1: find max element in array

```
@ Find max element in an array with parameters:
@      R2 = array address
@      R1 = value of size of array
@      Return in R0 = max element
@      R0 = maxarray3 (size:R1, addr.array: R2)
       ldr    r1,=size
       ldr    r1,[r1]       @ r1 = size of array
       ldr    r2,=array     @ r2 = & array
       bl     Maxarray3     @ int:R0=maxarray3
                            @      (addr array:r2,size:r1)
       swi    0x11
@*****************************
Maxarray3: code here (see next)
@*****************************
       .data
array:       .word 10,-2,12, 13,-5,6,9,11,13,-2
size: .word 10
       .end
```

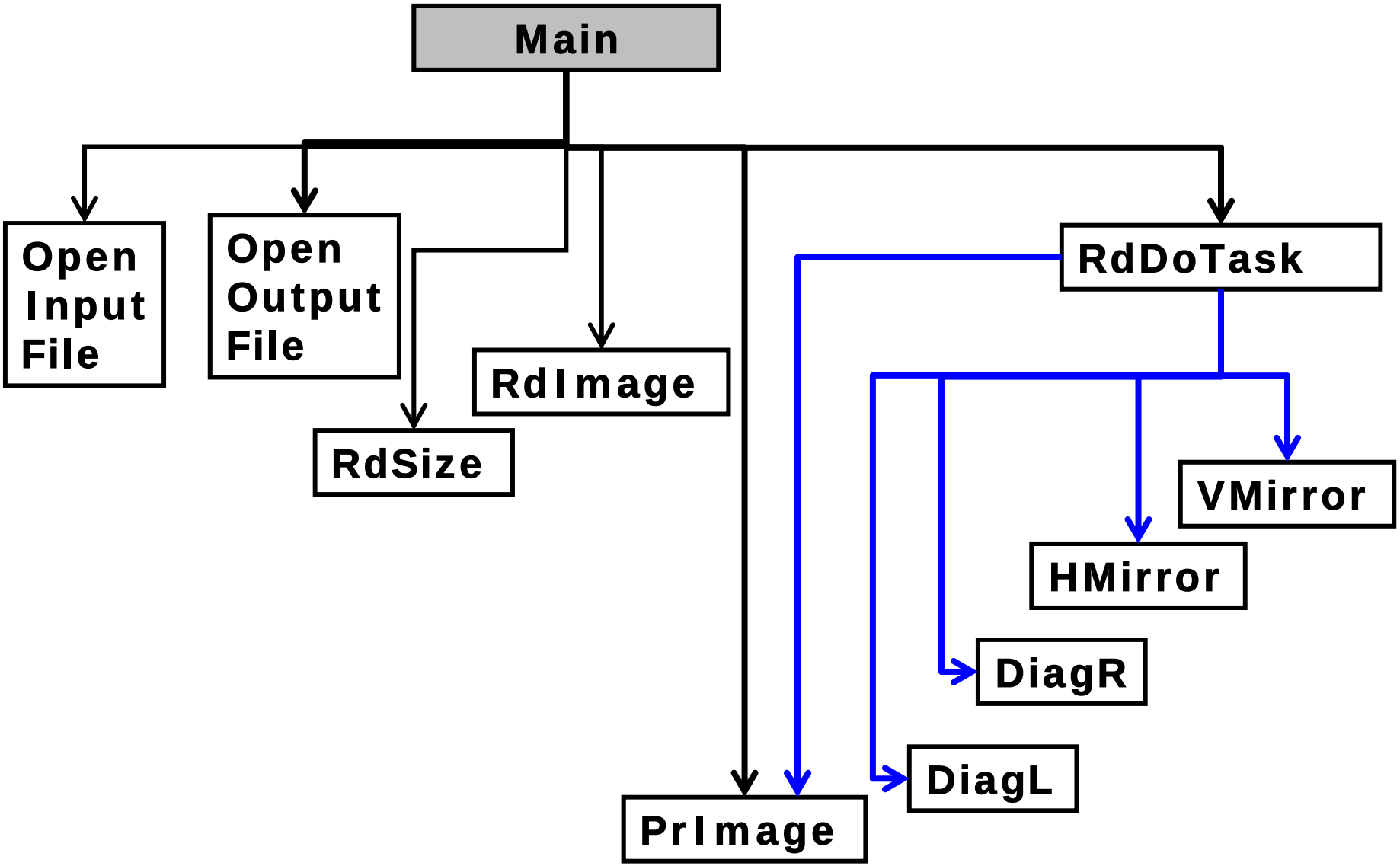
```
@ MaxArray3 function
@ r0 = Maxarray3 (size:r1, addr.array: r2)
Maxarray3:
        STMFD sp!,{r1-r10,lr}   @save registers

        LDMFD r13!,{r1-r10,lr}  @restore registers
        MOV   PC,LR             @ and return to the caller
        .data
array:      .word 10,-2,12, 13,-5,6,9,11,13,-2
```

```
@ MaxArray3 function
@r0 = maxarray3 (size:r1, addr.array: r2)
Maxarray3:
        STMFD sp!,{r1-r10,lr}   @save registers
        LDR   r0,[r2]       @r3 = tempmax = array[0]
        MOV   r3,#1         @r0 = index into array
loop:   CMP   r3,r1         @is index at end of array?
        BEQ   endmaxarray3      @checked whole array
        LDR   r4,[r2,#4]! @r4=array[i],increment r2
                            @ pointer
        CMP   r0,r4         @compare tempmax = array[index]
        BGE   incr          @tempmax ok, try next element
        MOV   r0,r4         @update tempmax
incr:   ADD   r3,r3,#1      @increment index
        BAL   loop
endmaxarray3:
        LDMFD r13!,{r1-r10,lr}  @restore registers
        MOV   PC,LR         @ and return to the caller
        .data
array:      .word 10,-2,12, 13,-5,6,9,11,13,-2
```

```
@ MaxArray3 function
@r0 = maxarray3 (size:r1, addr.array: r2)
Maxarray3:
        STMFD sp!,{r1,r2,r3,r4,lr}    @save registers
        LDR   r0,[r2]       @r3 = tempmax = array[0]
        MOV   r3,#1         @r0 = index into array
loop:   CMP   r3,r1         @is index at end of array?
        BEQ   endmaxarray3        @checked whole array
        LDR   r4,[r2,#4]! @r4=array[i],increment r2
                            @ pointer
        CMP   r0,r4         @compare tempmax = array[index]
        BGE   incr          @tempmax ok, try next element
        MOV   r0,r4         @update tempmax
incr:   ADD   r3,r3,#1      @increment index
        BAL   loop
endmaxarray3:
        LDMFD r13!,{r1,r2,r3,r4,lr}   @restore registers
        MOV   PC,LR         @ and return to the caller
        .data
array:        .word 10,-2,12, 13,-5,6,9,11,13,-2
```

```
@ MaxArray3 function
@r0 = maxarray3 (size:r1, addr.array: r2)
Maxarray3:
        STMFD sp!,{r1,r2,r3,r4,lr}    @save registers
        LDR   r0,[r2]       @r3 = tempmax = array[0]
        MOV   r3,#1         @r0 = index into array
loop:   CMP   r3,r1         @is index at end of array?
        BEQ   endmaxarray3        @checked whole array
        LDR   r4,[r2,#4]! @r4=array[i],increment r2
                            @ pointer
        CMP   r0,r4         @compare tempmax = array[index]
        BGE   incr          @tempmax ok, try next element
        MOV   r0,r4         @update tempmax
incr:   ADD   r3,r3,#1      @increment index
        BAL   loop
endmaxarray3:
        LDMFD r13!,{r1,r2,r3,r4,pc}   @restore registers
        @ MOV PC,LR         @ and return to the caller
        .data
array:        .word 10,-2,12, 13,-5,6,9,11,13,-2
```

Example: call graph for Image processing



Example: Call Graph for answer to Assignment 1 (C code)