PART 3: Embedded Programming STM32

ALU, basic instructions, Registers

STM32L152 Architecture

- ✓ STM32l152 uses a Harvard architecture / separate program and data memory buses
- ✓ Logical address space is unified (from 0x0000 0000 to 0xFFFF FFFFF.)

JM1

Memory Map Registers vs Core Registers

4GB Memory Map

Vendor Specific	
Private peripheral bus - external	
Private peripheral bus- internal	
External Device	1.0GB
External RAM	1.0GB
Peripheral	0.5GB
SRAM	0.5GB
Code	0.5GB

Access by pointing to an address

Registers

R0-R12 general purpose register
R13 (MSP/PSP)
R14 (LR)
PC
PSR
PRIMASK
FAULTMASK
BASEPRI
CONTROL

No accessible by pointer

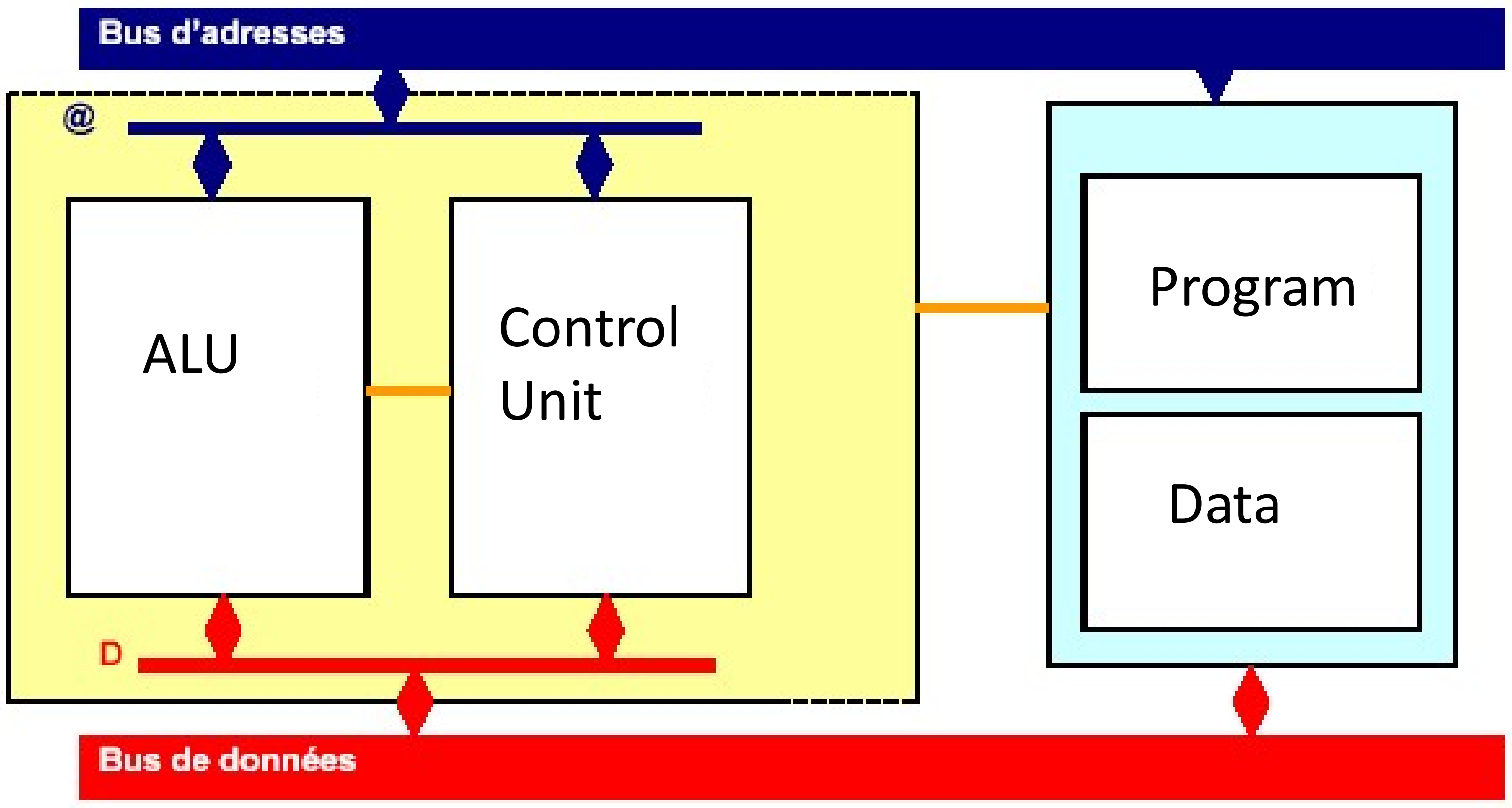
Diapositive 106

JM1 Jean-Jacques MENEU; 24/08/2020

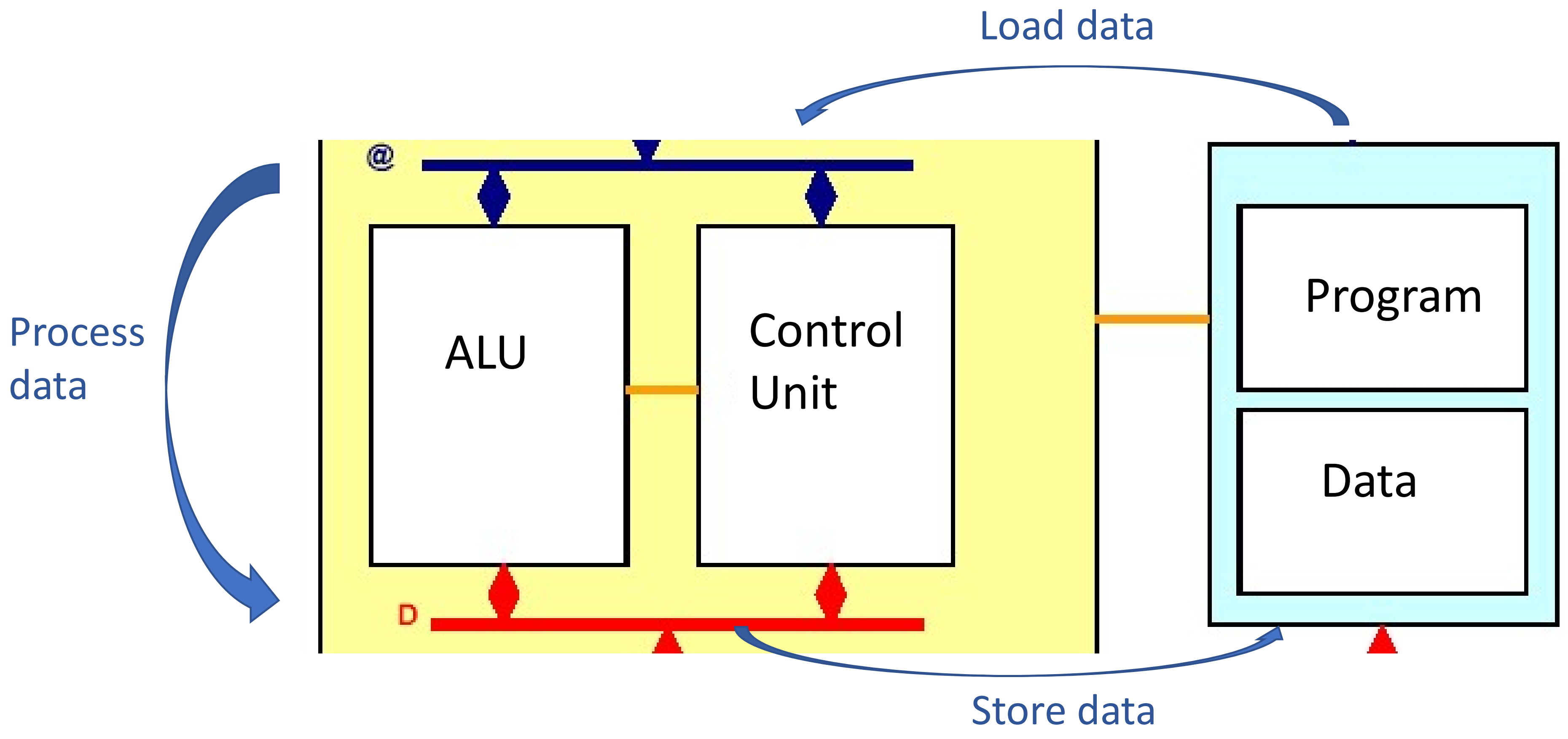
Cortex M Registers

- ✓ The Cortex-M processors apply a load and store method of operations.
- ✓ This means that to do any kind of data processing is performed from registers
- ✓ Instructions such as ADD and SUBTRACT: data must first be loaded into the CPU registers. The data processing is then executed, and the result is stored back in the main memory.

Microprocessors: Basic Architecture

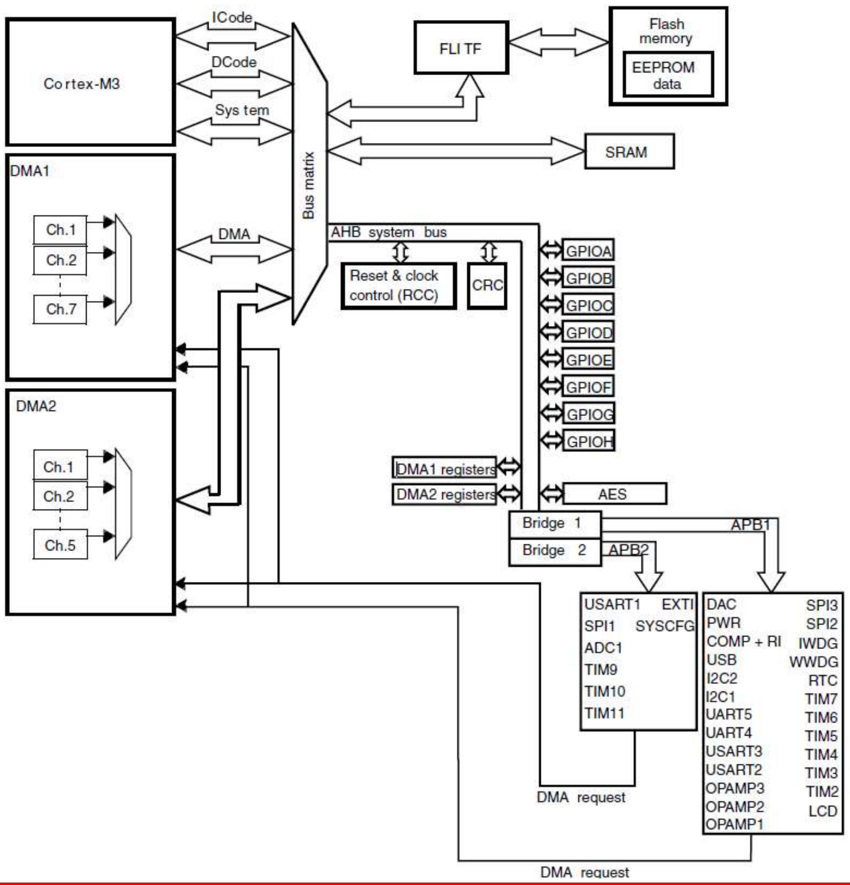


Load and Store Principle



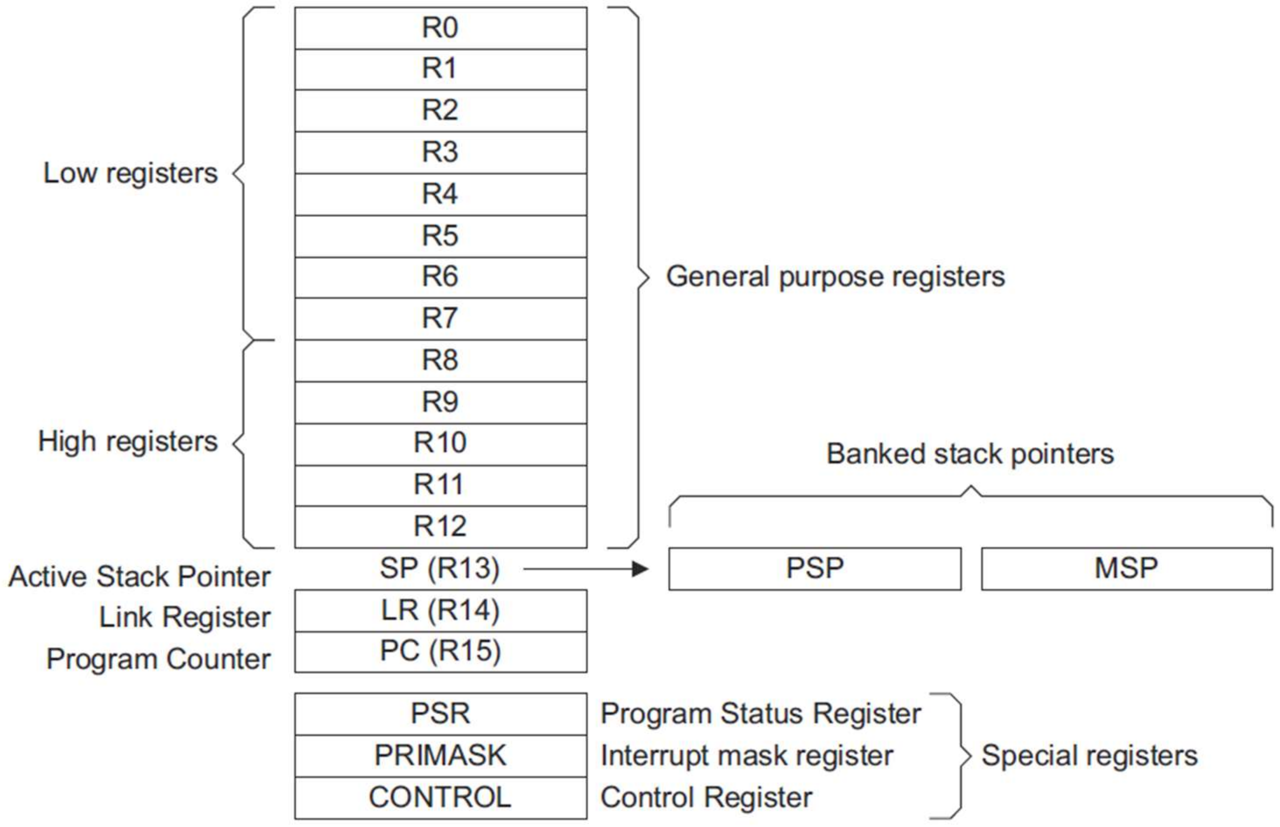
STM32L152 Architecture

Internal Registers and ALU



Memory map data

Cortex M Core Registers

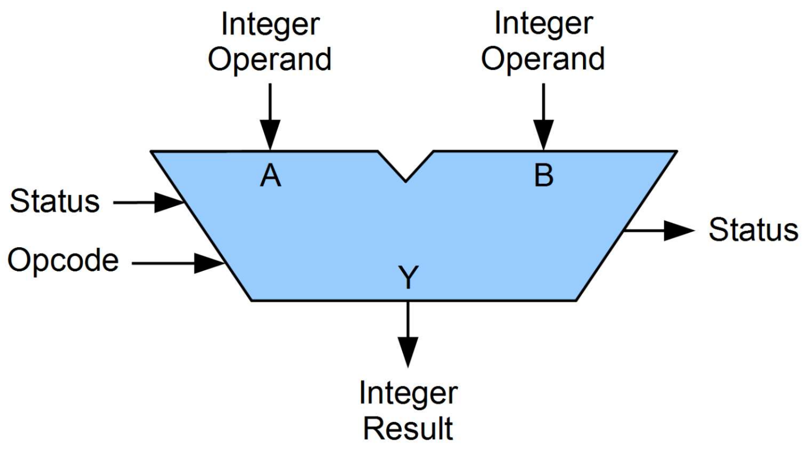


Arithmetic Logic Unit

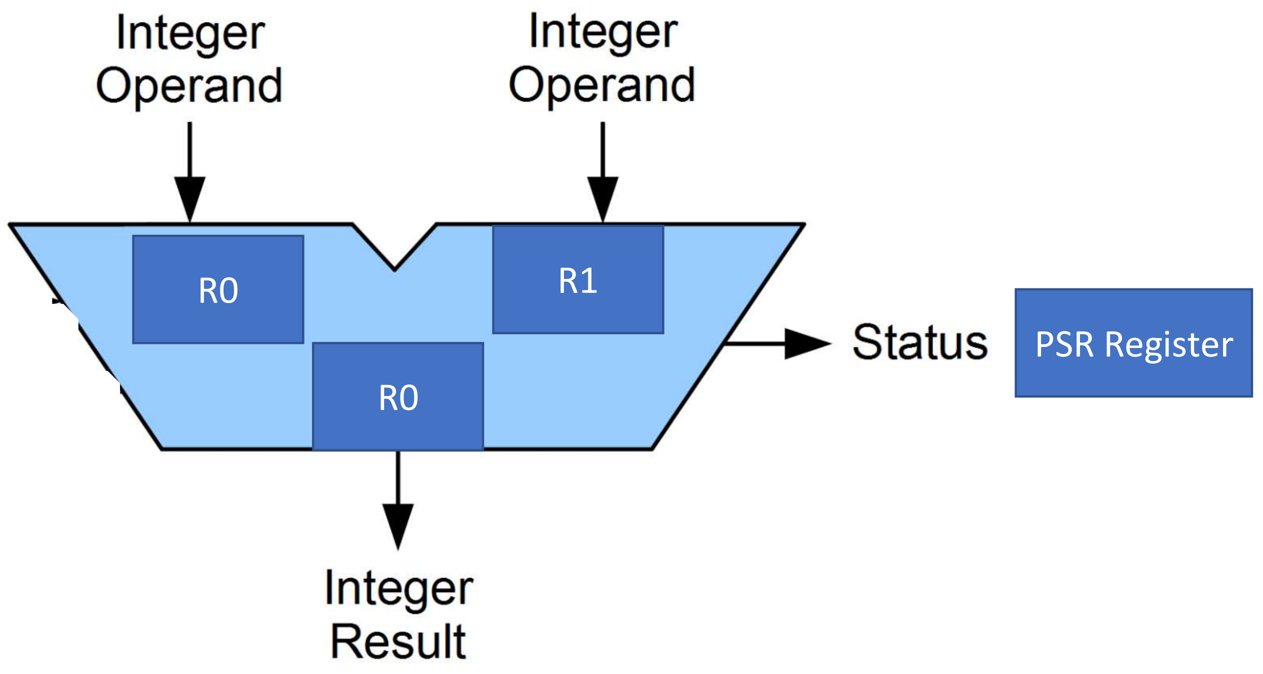
- ✓ An Arithmetic Logic Unit (ALU) is a combinational digital electronic circuit that performs arithmetic and bitwise operations on integer binary numbers
- ✓ ALU is a fundamental building block of a CPU.

Symbolic Representation ALU

- ✓ Input to an ALU are the data to be operated on, called operands
- ✓ A code indicating the operations to be performed (OpCode).
- ✓ ALU's output is the result of the performed operation
- ✓ Status of the previous or current operation



ALU Example : Addition



What is an Instruction?

- ✓ Each operation in the AUL is defined by a series of bytes, called an instruction
- ✓ An instruction has a length 16 or 32 bit on the Cortex M
- ✓ An instruction includes:
  - The operation code
  - Localisation of operands and result
- ✓ Details of the instruction set are written in the Programming Manual

Op-Code	OPER 1	OPER 2	RES

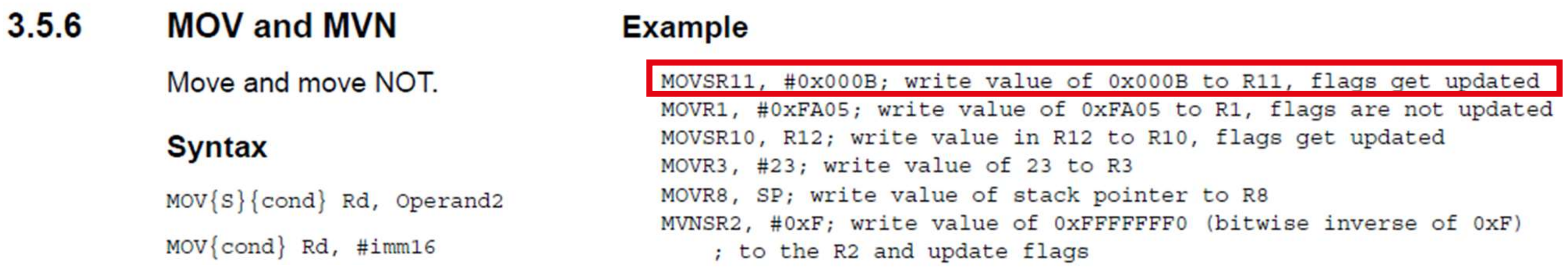
Load Operation

- ✓ A load operation allows to load a destination byte with a source byte
- ✓ It is called an instruction and in assembly language is referred as LD
- ✓ To copy data from a source to a destination, the source and the destinations must be defined
- ✓ The addressing modes give various ways to define the destination and the source

MOVS Instruction

✓ The programming manual explains the instruction set (syntax and examples are given)

✓ The MOVS instruction can store an immediate value in the range 0-65535 to a destination register



3.5.6 MOV and MVN

Move and move NOT.

Syntax

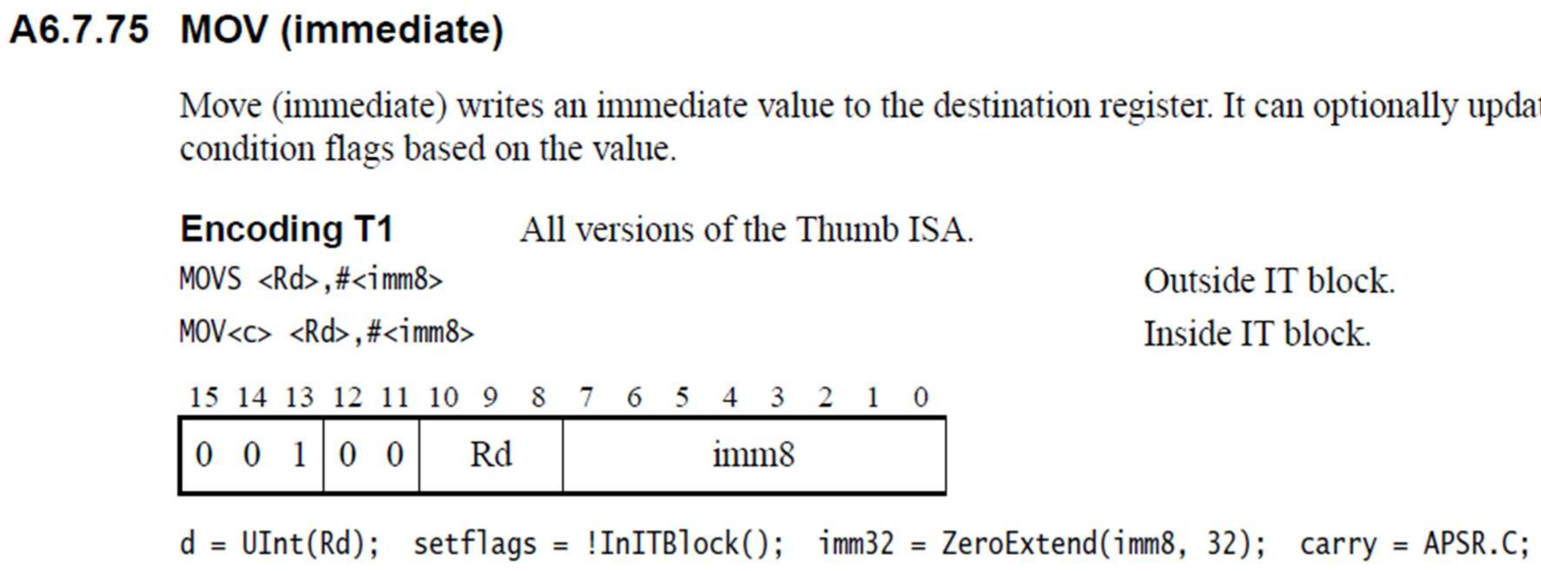
```
MOV{S}{cond} Rd, Operand2

MOV{cond} Rd, #imm16
```

Example

```
MOVSR11, #0x000B; write value of 0x000B to R11, flags get updated
MOVR1, #0xFA05; write value of 0xFA05 to R1, flags are not updated
MOVSR10, R12; write value in R12 to R10, flags get updated
MOVR3, #23; write value of 23 to R3
MOVR8, SP; write value of stack pointer to R8
MVNSR2, #0xF; write value of 0xFFFFFFF0 (bitwise inverse of 0xF)
    ; to the R2 and update flags
```

MOVS Instruction Encoding

The ARMv7-M Architecture Reference Manual details the encoding of each instruction



A6.7.75 MOV (immediate)

Move (immediate) writes an immediate value to the destination register. It can optionally update the condition flags based on the value.

Encoding T1 All versions of the Thumb ISA.

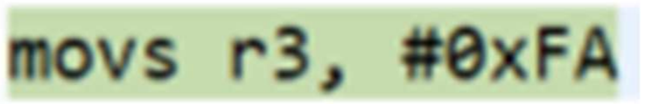
`MOVS <Rd>,#<imm8>`	Outside IT block.
`MOV<c> <Rd>,#<imm8>`	Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

MOV Instruction Encoding Example

With the example given in the Programming Manual



```
movs r3, #0xFA
```

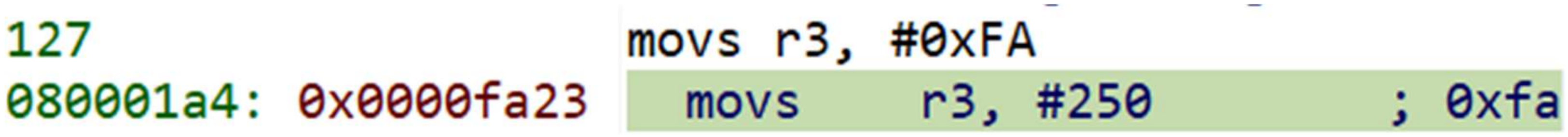
The encoding must be as follow

- Rd = 3 = b011 = 0x3
- Imm8 = 0xFA = b1111 1010

The instruction is then coded as b0010 0011 1111 1010 = 0x23FA

MOVs Instruction Encoding Example

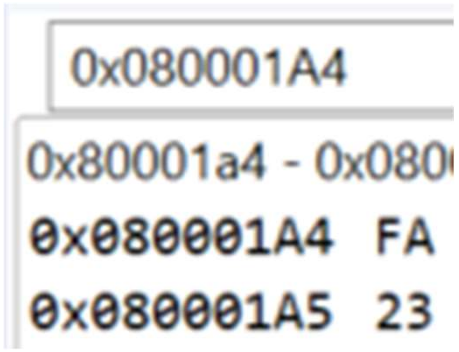
In the disassembly window, we can check the compiler encoded as 0xFA23



```
127                        movs r3, #0xFA
080001a4: 0x0000fa23         movs      r3, #250            ; 0xfa
```

In the memory window:

- Address 0x8000 01A4 : 0xFA
- Address 0x8000 01A6 : 0x23



```
0x080001A4
0x80001a4 - 0x080
0x080001A4  FA
0x080001A5  23
```

Instruction Sets

- ✓ Instruction sets depend on the microcontroller (even if basic instructions are present in all microcontrollers)
- ✓ When an instruction is not available, 2 options:
  - Find a microcontroller having this instruction
  - Program this instruction
- ✓ 8-bit microcontrollers have no instruction to add float numbers and must be done by software
- ✓ Cortex M0 has only basic instructions unlike Cortex M4 that can perform FPU (Floating Point Unit) instructions

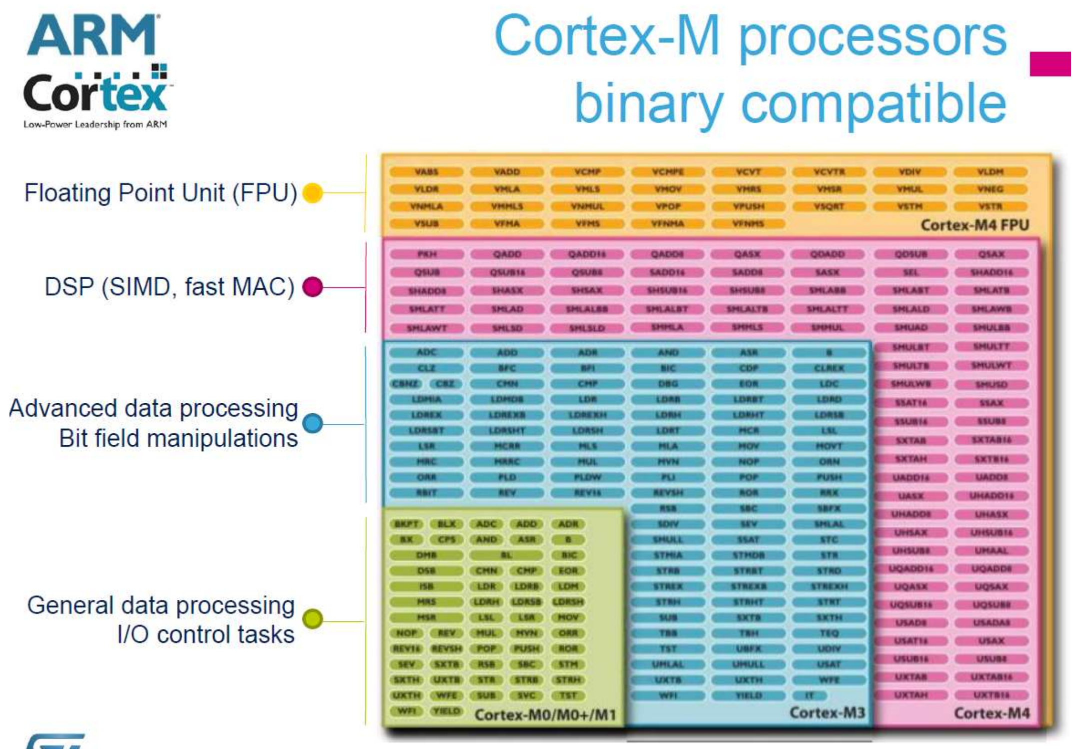
ARM Instructions

2 instruction sets

- ✓ ARM instruction set would allow code to be written for maximum performance as they are 32-bit instructions
- ✓ Thumb code would achieve a greater code density as it is 16-bit instructions
- ✓ Cortex-M processors are code compatible with the original Thumb instruction set, but they are designed to execute an extended version of the Thumb instruction set called Thumb-2.
- ✓ Thumb-2 is a blend of 16 and 32 bit instructions that has been designed to be very C friendly and efficient.

Cortex M supports only Thumb Instructions

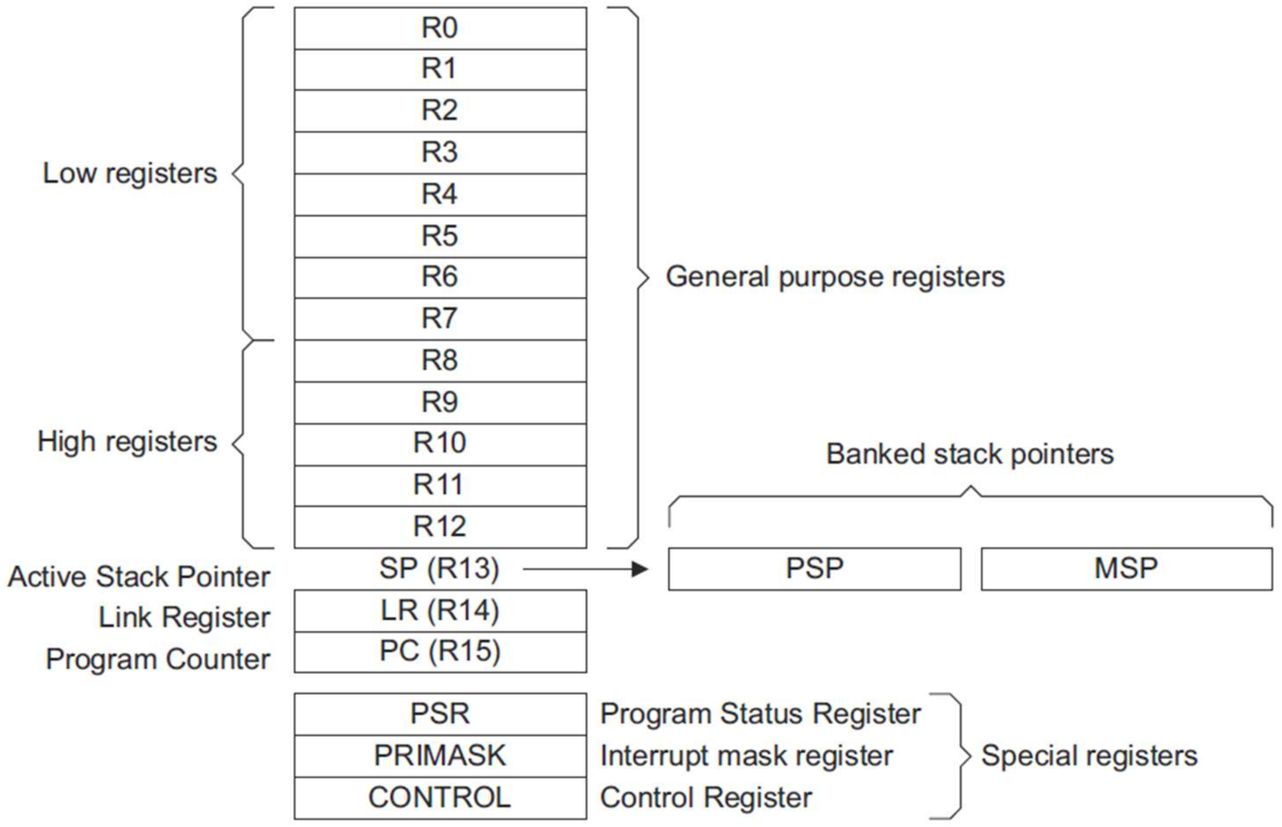
STM32 Instruction Sets



Source: ARM

The chosen instruction set depends on math requirements

STM32L152 Core Registers

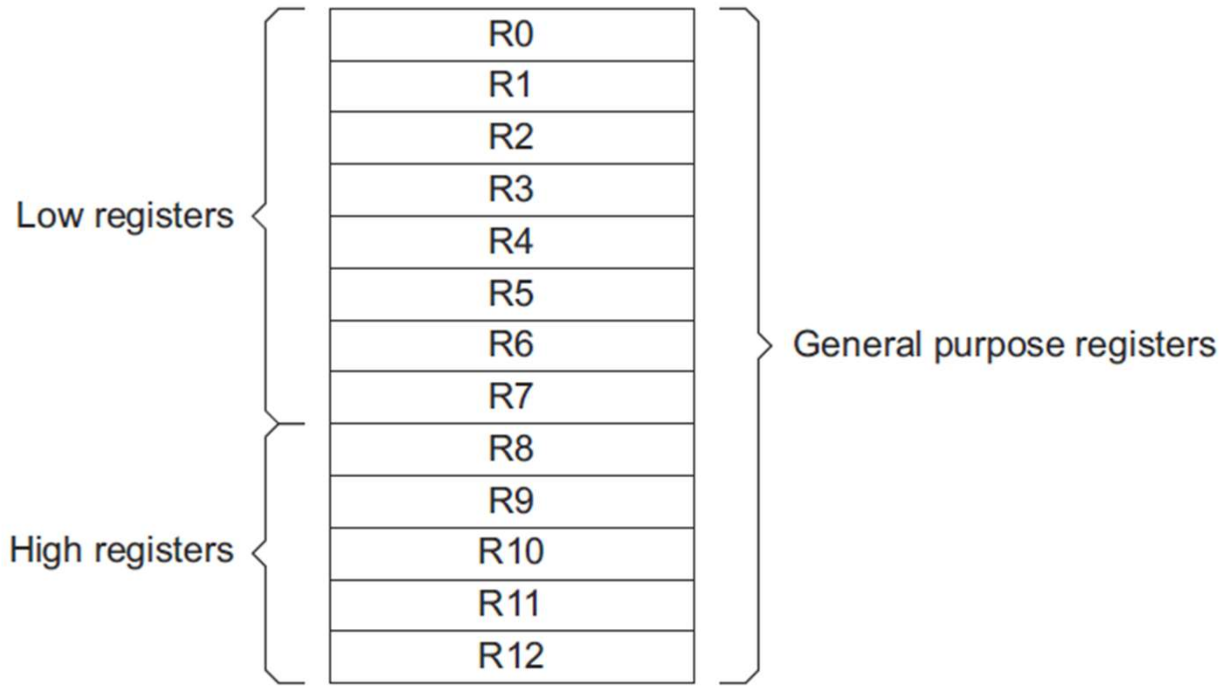


Internal registers cannot be accessed by address pointers

STM32M152 General Purpose Registers

The general purpose registers are the input of the AUL and the output (where the result of the operation is stored)

It is not possible to modify the value of a variable directly in the memory (load and store principle)



Steps to Increment a Value in Memory

1) Get the address where the variable is stored in memory
2) Retrieve the value in memory by pointing the address and store it in a register
3) Increment the value stored in the register
4) Store this new value in memory

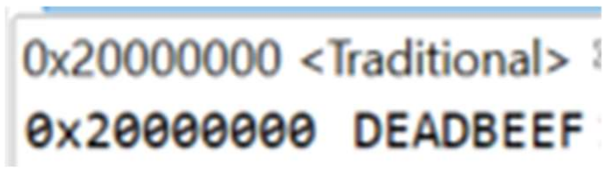
The necessary assembly functions are LDR, ADD and STR

LDR instruction 1/2

Gcc code to declare a variable in the data section

```
.section .data

varindata:
    .word 0xDEADBEEF
```

1) At address 0x20000000 the value 0xDEADBEEF is stored

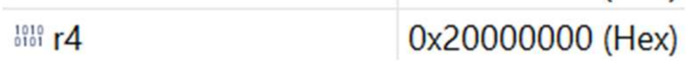


```
0x20000000 <Traditional>
0x20000000  DEADBEEF
```

2) The assembly ldr instruction retrieves the address of the variable varindata

3) Register r4 stores the address of varindata which is equal to 0x20000000

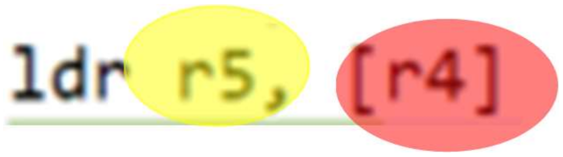
```
ldr r4, =varindata
```



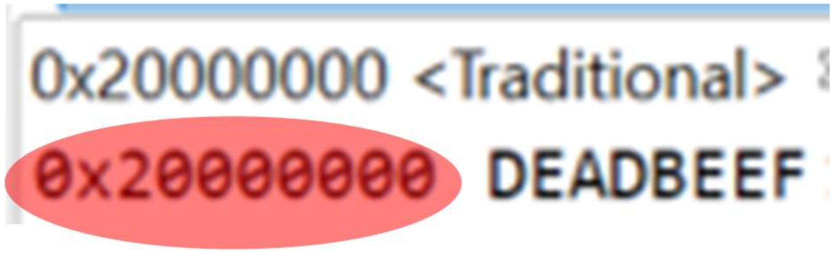
```
r4        0x20000000 (Hex)
```

*0x20000000 is the beginning of the SRAM section that stores initialized variables

LDR instruction 2/2

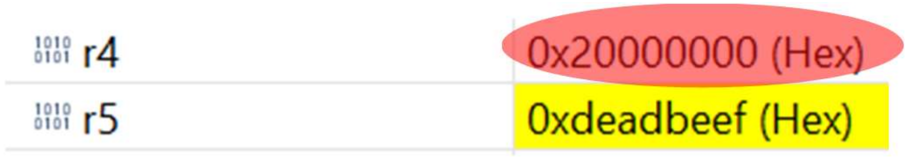


```
ldr r5, [r4]
```



```
0x20000000 <Traditional>
0x20000000  DEADBEEF
```

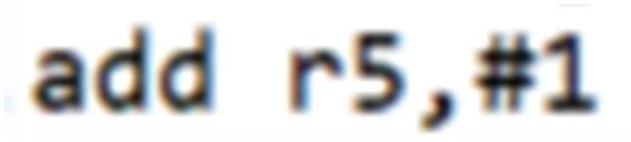
[r4] means to point to the adress stored in register r4



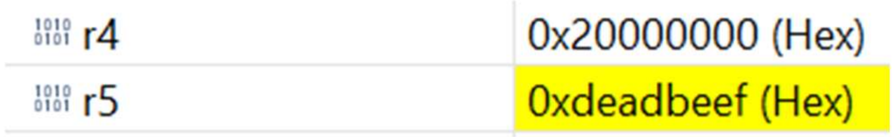
r4	0x20000000 (Hex)
r5	0xdeadbeef (Hex)

value stored at address 0x20000000 is 0xDEADBEEF -> R5 = 0xDEADBEEF

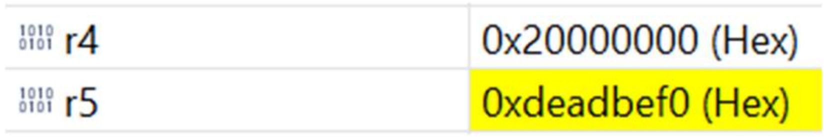
ADD instruction



```
add r5,#1
```

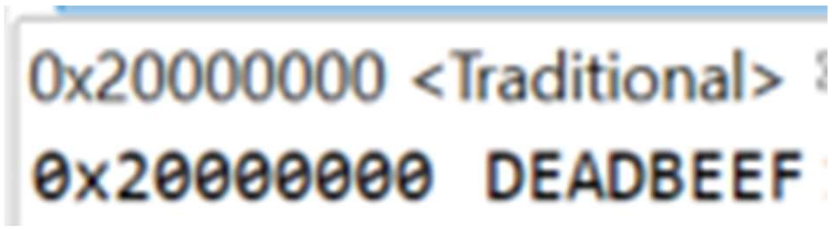


r4	0x20000000 (Hex)
r5	0xdeadbeef (Hex)



r4	0x20000000 (Hex)
r5	0xdeadbef0 (Hex)

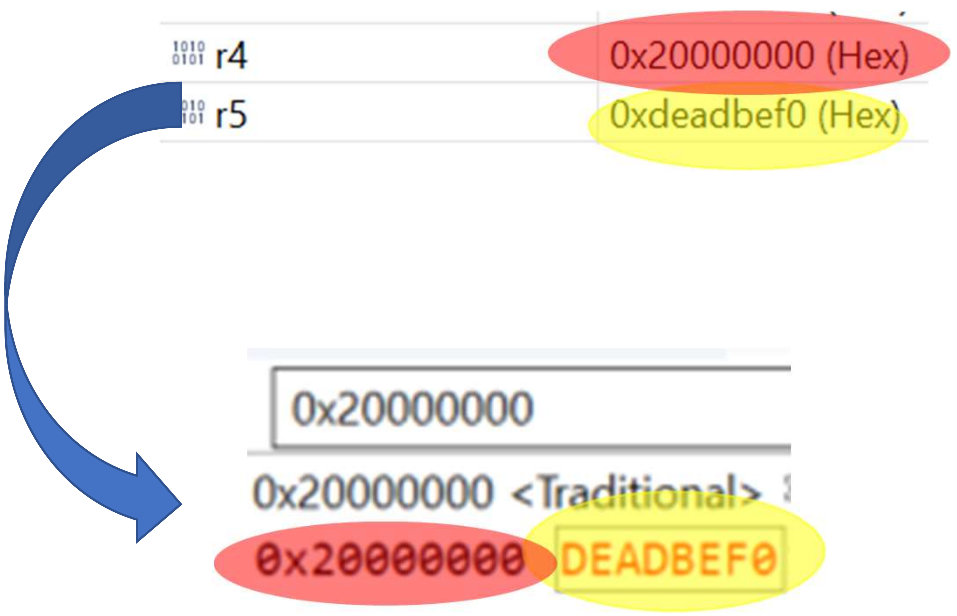
The value in memory is still 0xDEADBEEF



```
0x20000000 <Traditional>
0x20000000  DEADBEEF
```

STR instruction

```
str r5, [r4]
```



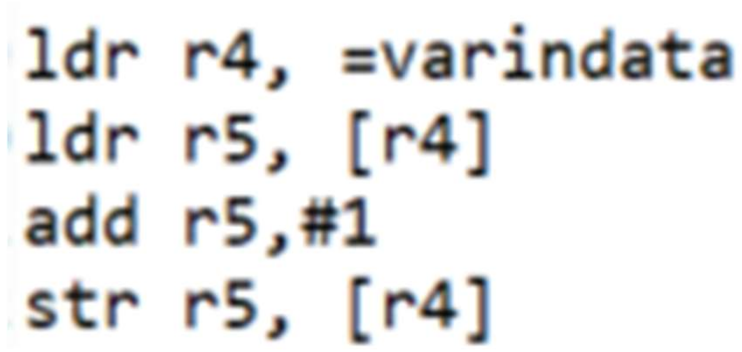
```
r4    0x20000000 (Hex)
r5    0xdeadbef0 (Hex)
```

```
0x20000000
0x20000000 <Traditional>
0x20000000 DEADBEF0
```

[r4] means to point to the adress stored in register r4

value stored at address 0x20000000 becomes 0xDEADBEF0

Summary Increment Variable



```
ldr r4, =varindata
ldr r5, [r4]
add r5,#1
str r5, [r4]
```

4 instructions are necessary to increment the value of a variable stored in memory

NOP Instruction

- ✓ NOP stands for No Operation
- ✓ This instruction just does nothing (sometimes useful the application needs to wait)

PC Calculation

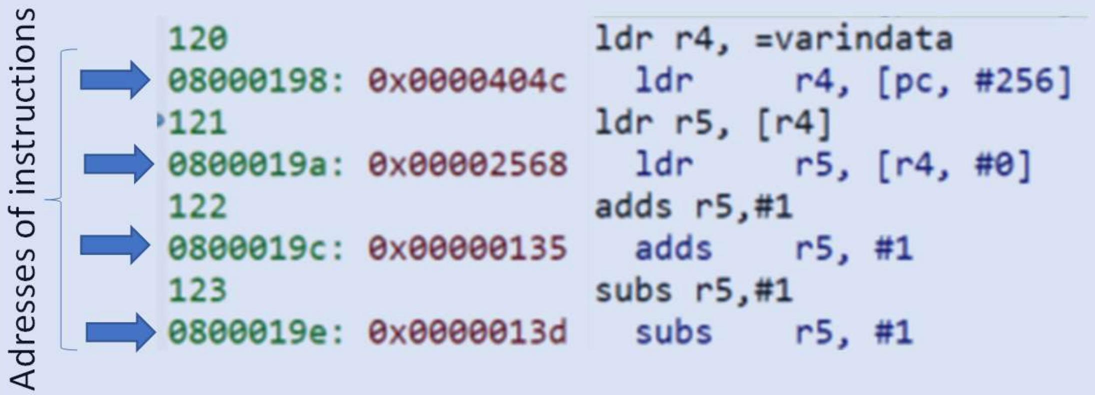
- ✓ PC stands for Program Counter
- ✓ The PC store the address of the next instruction to execute
- ✓ If the running instruction starts at the address AD and occupies 2 bytes, the new PC values is then AD+2 that represents the address of the following instruction.
- ✓ Instruction length in Cortex M (Thumb 2) is a mix od 16 and 32 bits → PC value is always even
- ✓ The PC of the STM32 is a 32-bit register to map 4GB of memory

PC Calculation

Code

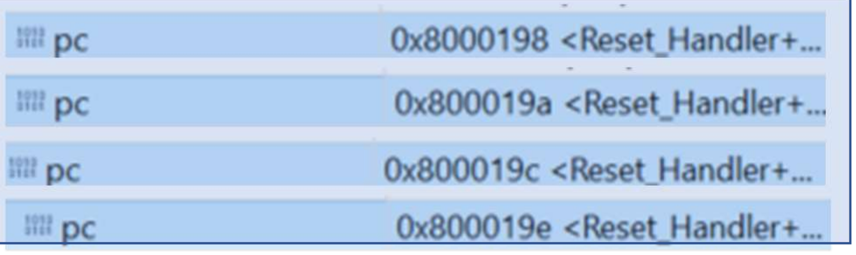
```
ldr r4, =varindata
ldr r5, [r4]
add r5,#1
str r5, [r4]
```

Disassembly Window



```
120                                ldr r4, =varindata
08000198: 0x0000404c                 ldr      r4, [pc, #256]
121                                ldr r5, [r4]
0800019a: 0x00002568                 ldr      r5, [r4, #0]
122                                adds r5,#1
0800019c: 0x00000135                 adds     r5, #1
123                                subs r5,#1
0800019e: 0x0000013d                 subs     r5, #1
```

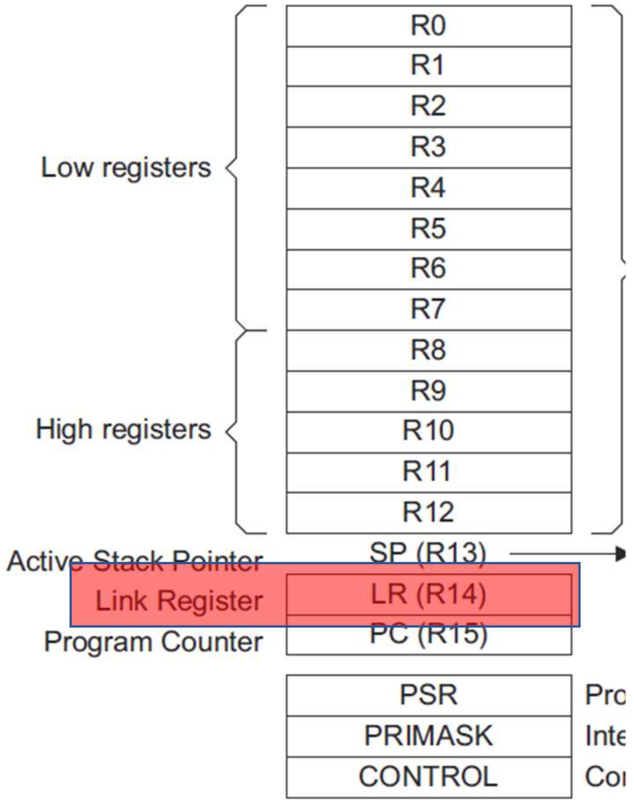
In debug mode, the PC register is updated at each instruction



```
pc    0x8000198 <Reset_Handler+...
pc    0x800019a <Reset_Handler+...
pc    0x800019c <Reset_Handler+...
pc    0x800019e <Reset_Handler+...
```

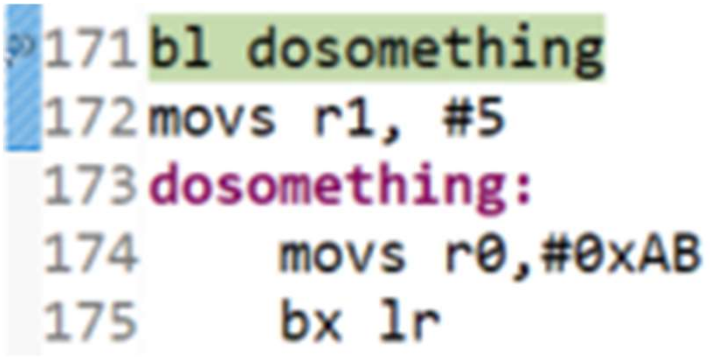
Link Register

- ✓ The Link Register stores the return information for subroutines, function calls and exceptions. (On reset, the LR value is unknown)
- ✓ When there is a jump in the program to execute some code and must return at the jump level, it is necessary to retrieve the address where the jump occurred. This information is stored thanks to the Link Register

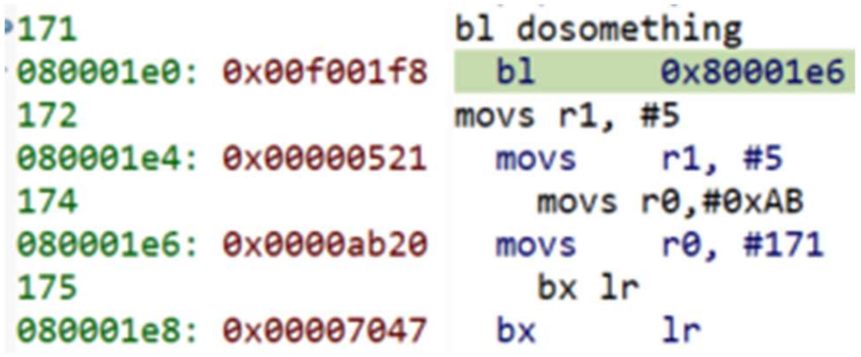


Link Register Example

1. At line 171, the program must jump to the label dosomething which is at the address 0x08000 01E6
2. Instruction movs r0,#0xAB is then executed
3. Instruction bx lr make a jump to instruction movs r1,#5



```
171 bl dosomething
172 movs r1, #5
173 dosomething:
174       movs r0,#0xAB
175       bx lr
```

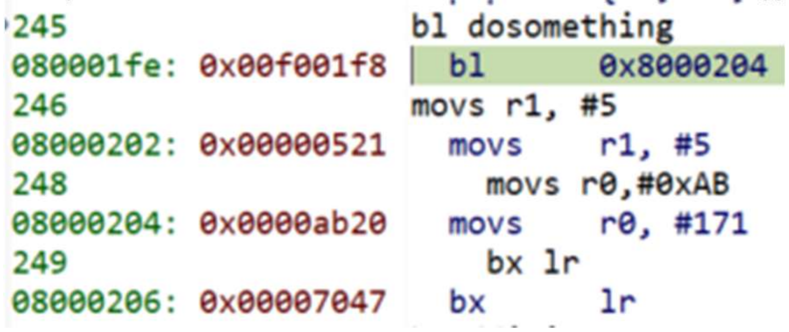


```
171                           bl dosomething
080001e0:  0x00f001f8          bl        0x80001e6
172                           movs r1, #5
080001e4:  0x00000521          movs      r1, #5
174                              movs r0,#0xAB
080001e6:  0x0000ab20          movs      r0, #171
175                              bx lr
080001e8:  0x00007047          bx        lr
```

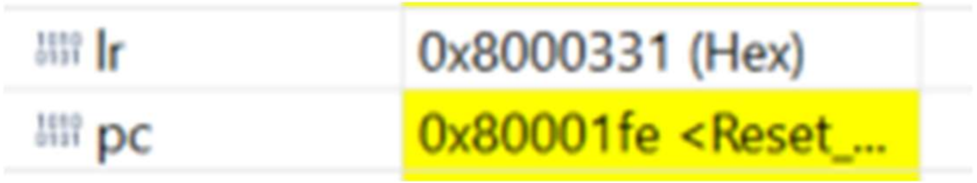
Let's see in Details the values of PC and LR registers during the execution of this program…

Link Register Code Exécution (1/2)

1) The next instruction is 'bl dosomething' at address 0x0800 01FE, hence PC = 0x0800 01E0

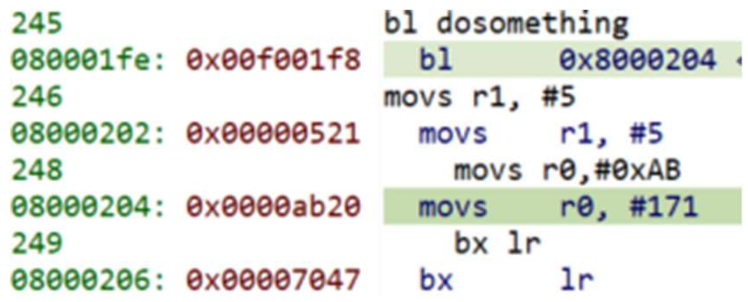


```
245                          bl dosomething
080001fe: 0x00f001f8         bl        0x8000204
246                          movs r1, #5
08000202: 0x00000521         movs      r1, #5
248                            movs r0,#0xAB
08000204: 0x0000ab20         movs      r0, #171
249                            bx lr
08000206: 0x00007047         bx        lr
```

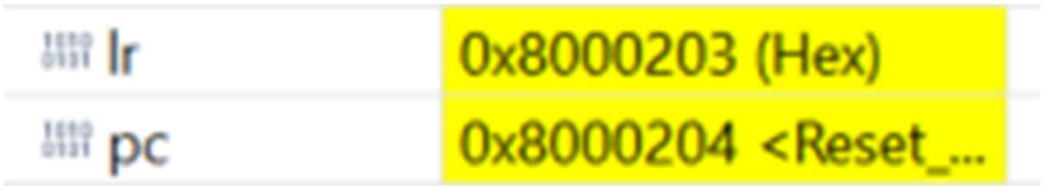


lr	0x8000331 (Hex)
pc	0x80001fe <Reset_...

2) The program jumps at the instruction movs, r0,#0xAB that is at address 0x8000 204. LR = 0x8000 0203 to point to the instruction following 'bl dosomething'



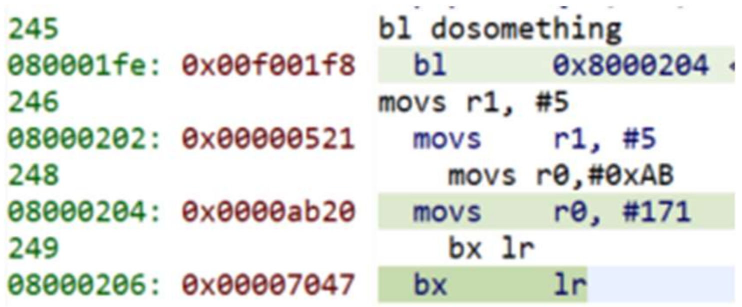
```
245                          bl dosomething
080001fe: 0x00f001f8         bl        0x8000204
246                          movs r1, #5
08000202: 0x00000521         movs      r1, #5
248                            movs r0,#0xAB
08000204: 0x0000ab20         movs      r0, #171
249                            bx lr
08000206: 0x00007047         bx        lr
```



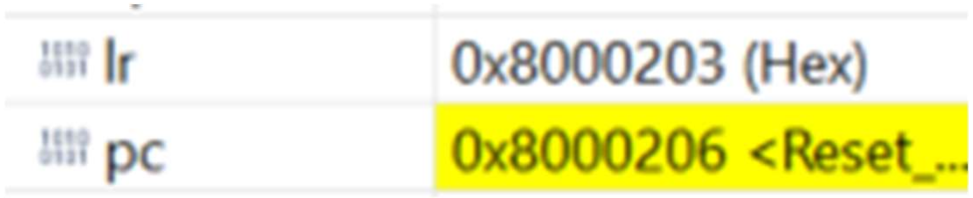
lr	0x8000203 (Hex)
pc	0x8000204 <Reset_...

Link Register Code Exécution (2/2)

3) The next instruction is ‘bx lr’ at address 0x0800 0206, hence PC = 0x0800 0206

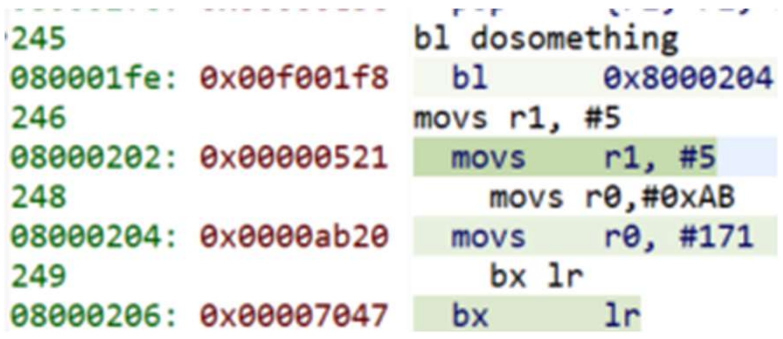


```
245                      bl dosomething
080001fe: 0x00f001f8       bl       0x8000204
246                      movs r1, #5
08000202: 0x00000521       movs     r1, #5
248                         movs r0,#0xAB
08000204: 0x0000ab20       movs     r0, #171
249                         bx lr
08000206: 0x00007047       bx       lr
```

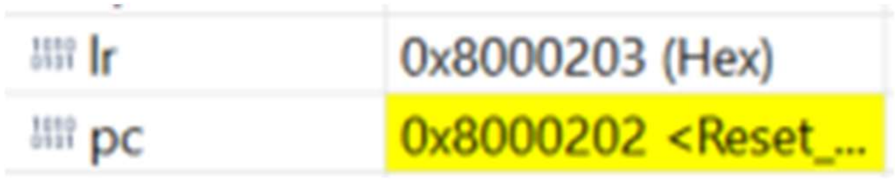


lr	0x8000203 (Hex)
pc	0x8000206 <Reset_...

4) The program jumps at the instruction movs, r1,#5 that is at address 0x8000 0202. lr does not change



```
245                      bl dosomething
080001fe: 0x00f001f8       bl       0x8000204
246                      movs r1, #5
08000202: 0x00000521       movs     r1, #5
248                         movs r0,#0xAB
08000204: 0x0000ab20       movs     r0, #171
249                         bx lr
08000206: 0x00007047       bx       lr
```



lr	0x8000203 (Hex)
pc	0x8000202 <Reset_...

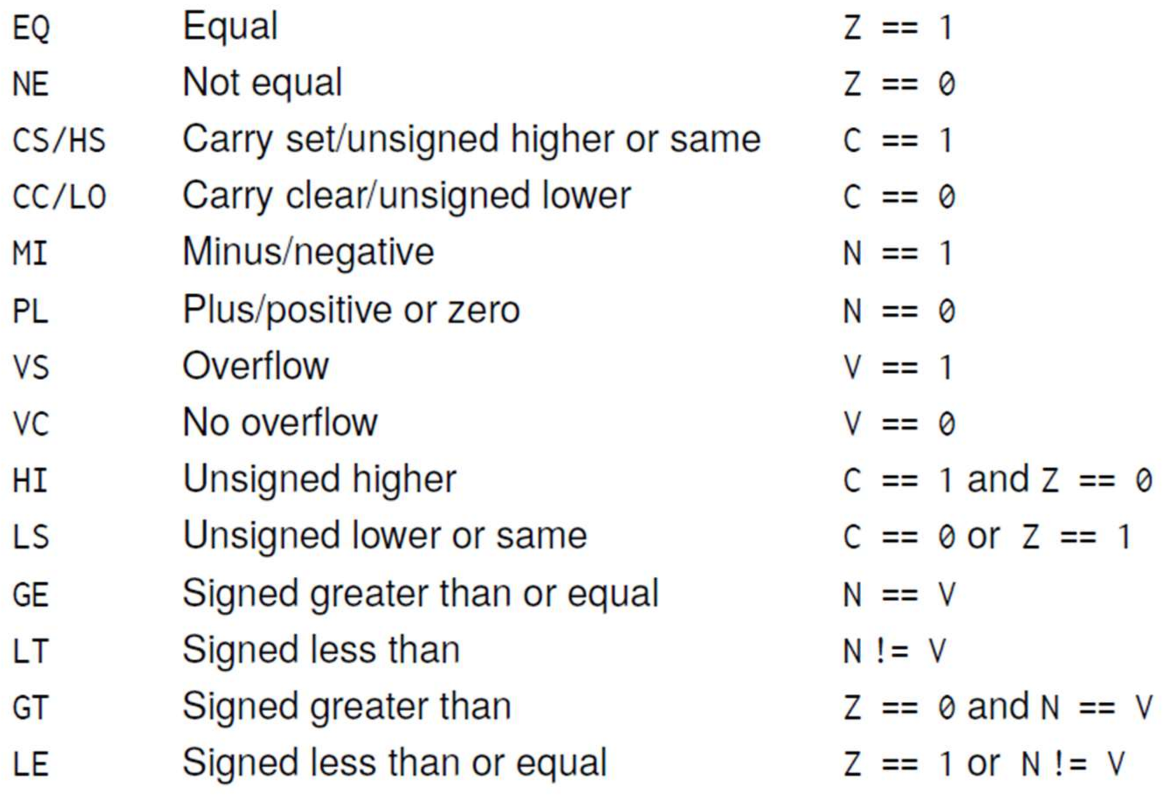
LSB of lr

- ✓ When lr = 0x0800 01E5, PC becomes 0x0800 01E4 while we could expect 0x0800 01E5
- ✓ PC always has an even value, this means the lsb is always equal to 0
- ✓ ARM processors can run with thumb instructions or ARM instructions. This information is given by the lsb of lr.
- ✓ When lr value is copied to pc register, if lsb of lr is equal to 1 this informs the processor that the next instructions is a thumb instructions
- ✓ As Cortex M supports only thumb instructions, lsb of lr is equal to 1

Branch Instructions Summary

- ✓ B address : go to the address
- ✓ B<c> address : go to the address with conditions (depends on APSR register)
- ✓ BL address : save return address in lr (link register) and go to the address
- ✓ BX register : go to the address pointed in the register (often used with lr)

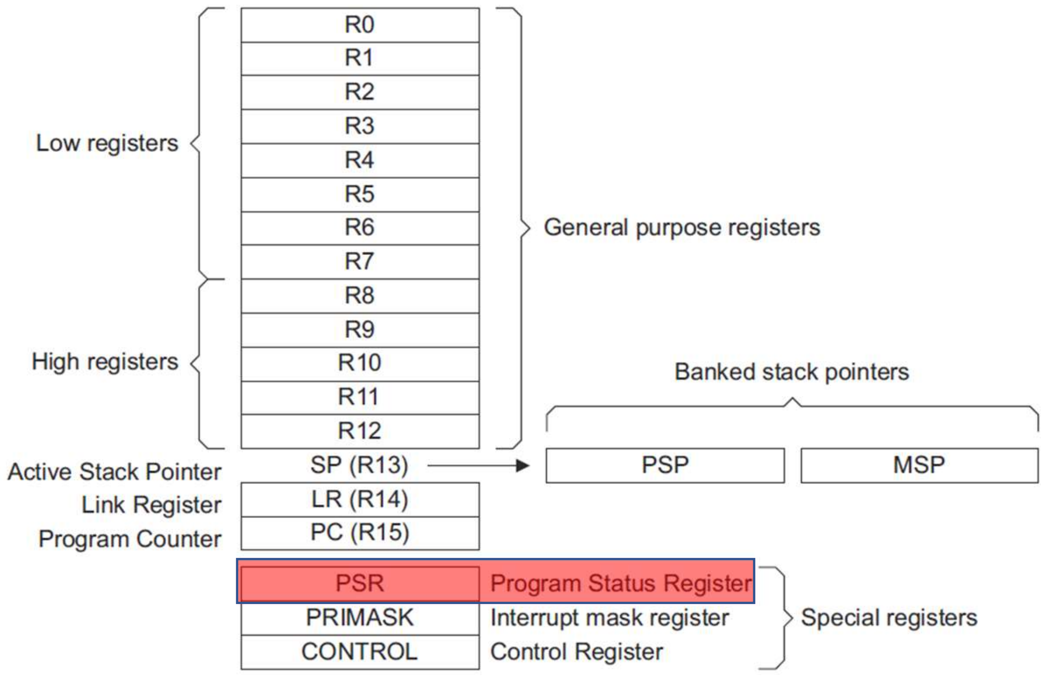
Conditonal Execution



EQ	Equal	Z == 1
NE	Not equal	Z == 0
CS/HS	Carry set/unsigned higher or same	C == 1
CC/LO	Carry clear/unsigned lower	C == 0
MI	Minus/negative	N == 1
PL	Plus/positive or zero	N == 0
VS	Overflow	V == 1
VC	No overflow	V == 0
HI	Unsigned higher	C == 1 and Z == 0
LS	Unsigned lower or same	C == 0 or Z == 1
GE	Signed greater than or equal	N == V
LT	Signed less than	N != V
GT	Signed greater than	Z == 0 and N == V
LE	Signed less than or equal	Z == 1 or N != V

Program Status Register (PSR)

- ✓ The Program Status Register allows to record information about the result of an operation (addition, substraction, comparison, test…)
- ✓ It allows to know if an operation comes up with a 0, negative, carry or overflow result
- ✓ During an interrupt, the interrupt number is recorded



PSR Register

- ✓ PSR stands for Program Status Register
- ✓ As its name implies, the PSR contains all the CPU status flags
- ✓ The PSR has three alias registers
  - Application Program Status Register (APSR)
  - Interrupt Program Status Register (IPSR)
  - Execution Program Status Register (EPSR)
- ✓ Each of these alias registers contains a subset of the full register flags and can be used as shortcut if it is necessary to access part of the PSR.
- ✓ The PSR is generally referred to as the xPSR to indicate the full register rather than any of the alias subsets.

PSR Register Contents

APSR	N	Z	C	V	Q					
IPSR										Exception Number
EPSR						ICI/IT	T		ICI/IT	

PSR	N	Z	C	V	Q	ICI/IT	T		ICI/IT	Exeption Number

- ✓ In a normal application program, the code does not make explicit access to the xPSR or any of its alias registers
- ✓ Any use of the xPSR is made by compiler generated code.
- ✓ But as a programmer it is mandatory to have an awareness of the xPSR and the flags contained in it.

xPSR Register Contents

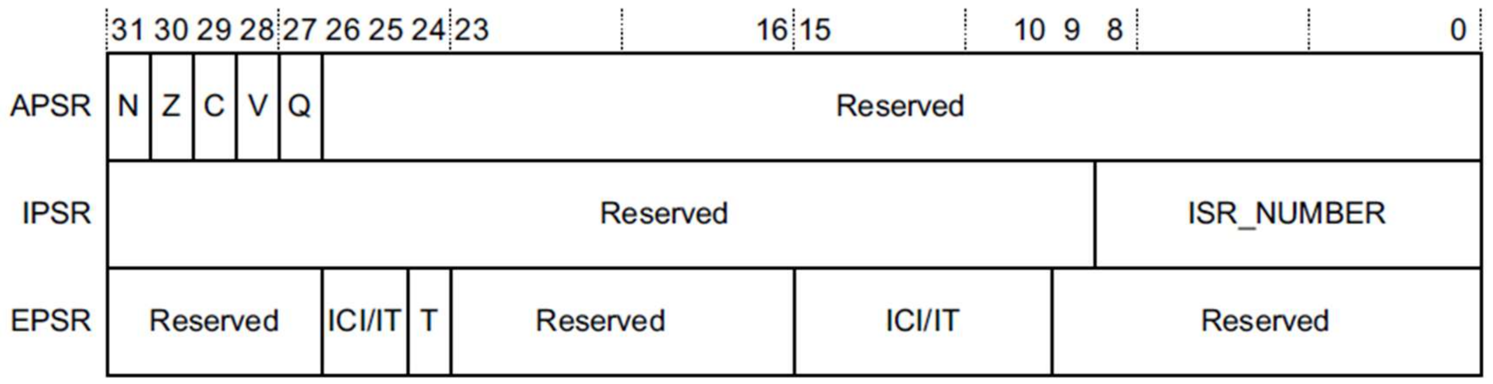
✓ The most significant four bits of the xPSR are the condition code bits
  - Negative
  - Zero
  - Carry
  - oVerflow

✓ These are set and cleared depending on the results of a data processing instruction.

SUB R8,R6	perform a subtraction and do not update the condition code flags
SUBS R8,R6	perform a subtraction and update the conditions code flags

✓ This allows the compiler to perform an instruction set that updates the condition code flags, then perform instructions that do not modify the flags and then perform a conditional branch on the state of the xPSR condition codes.

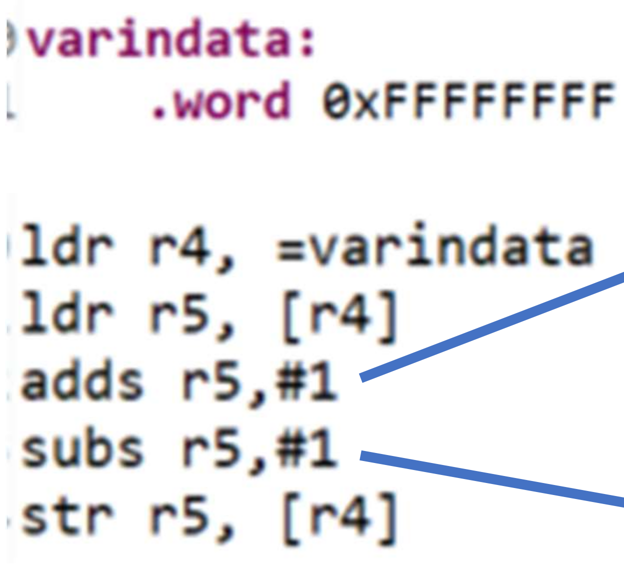
PSR in Details



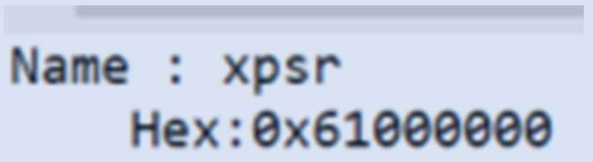
The PSR register is split in 3 sub-registers:

- ✓ Application Program Status Register (negative or null result….)
- ✓ Interrupt Program Status Register : interrupt number
- ✓ Execution Program Status Register: T = 1 in Cortex M processors as only thumb instructions are supported

PSR Application

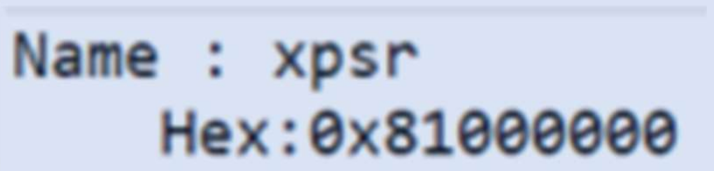


```
varindata:
      .word 0xFFFFFFFF

ldr r4, =varindata
ldr r5, [r4]
adds r5,#1
subs r5,#1
str r5, [r4]
```



```
Name : xpsr
    Hex:0x61000000
```

Result is zero with carry

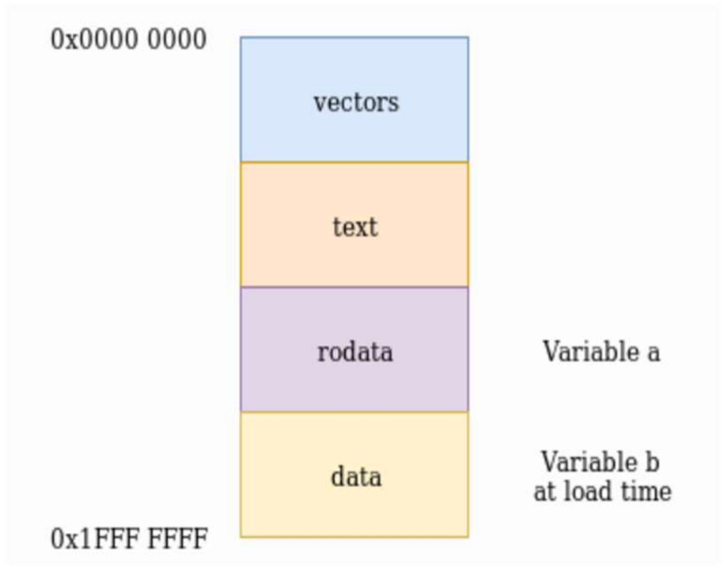


```
Name : xpsr
    Hex:0x81000000
```

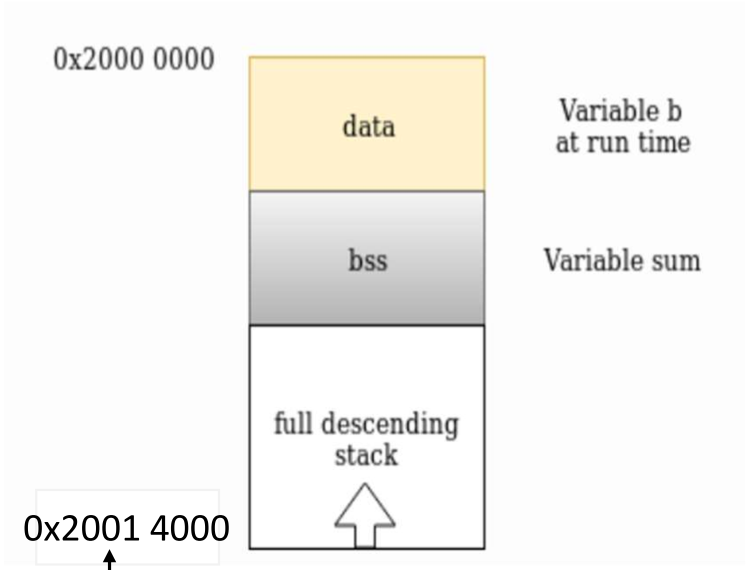
Result is negative

Memory Organization

Flash Memory

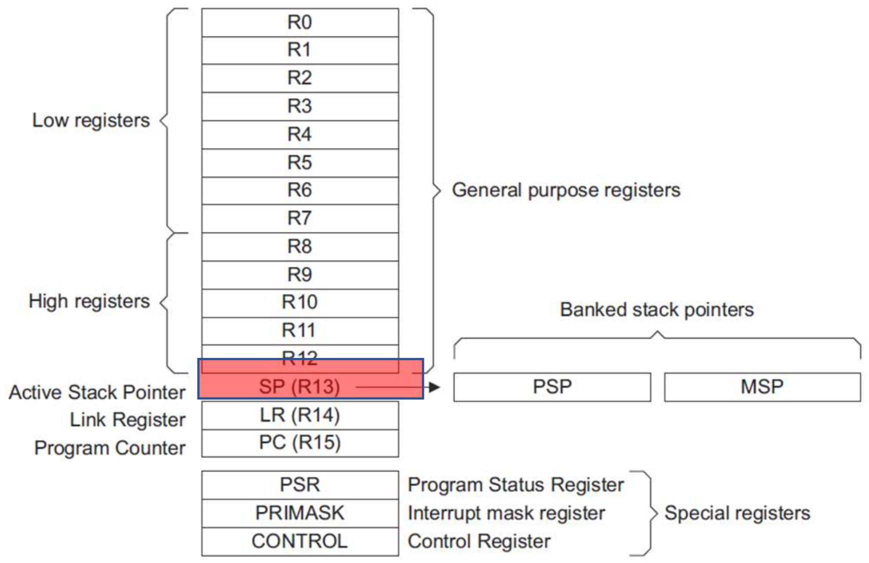


SRAM Memory



STM32L152 has 80KBytes SRAM. 80*1024 = 0x1 4000 = 81 920

Stack Pointer



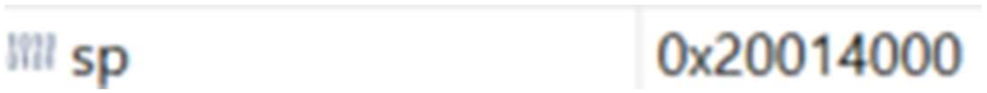
A special memory area: the stack

- ✓ Area in the RAM
- ✓ LIFO (Last In First Out) type operation (Vs FIFO)
- ✓ Stores data temporarily
- ✓ Allows to save the context (PC, …) when a routine or an interrupt is called

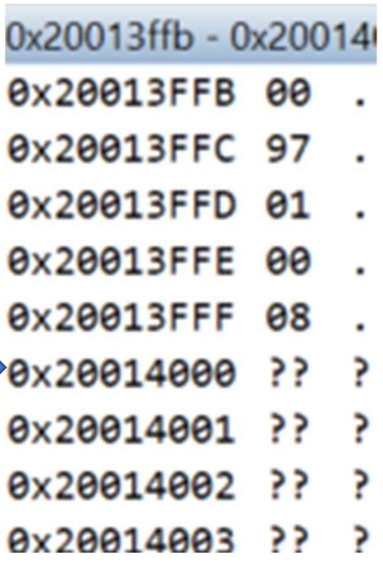
→ Need a counter that always points to the next available cell (byte) in the stack

→ The stack pointer, decreased after stacking, incremented before unstacking

Stack Initialisation



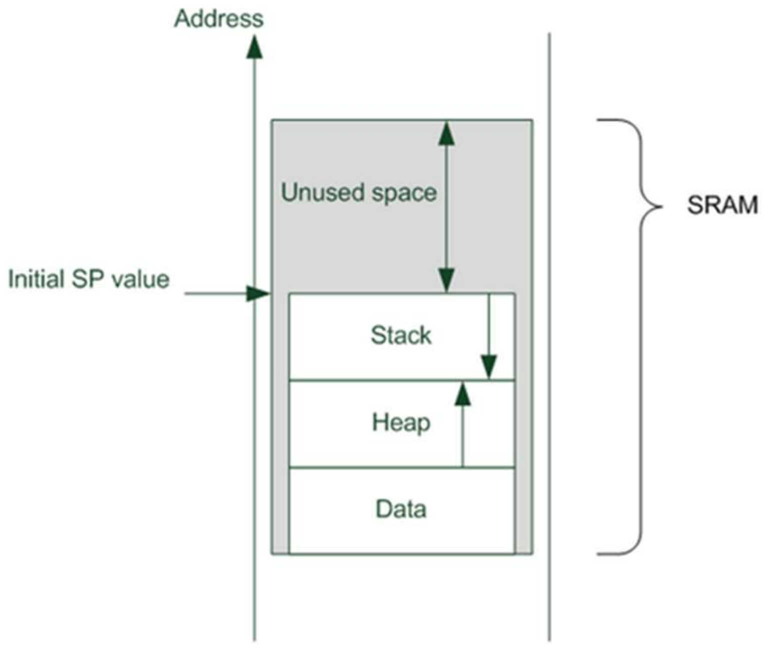
```
sp                    0x20014000
```



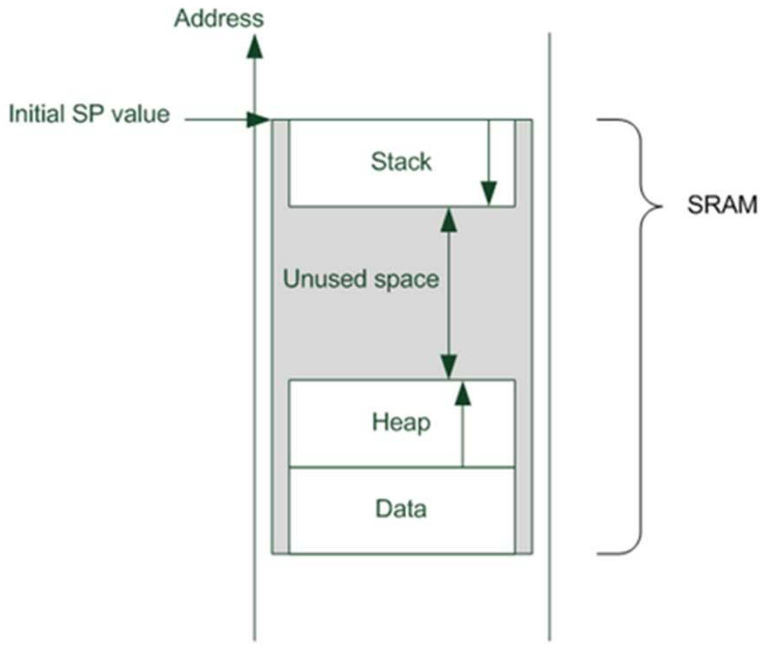
```
0x20013ffb - 0x20014
0x20013FFB  00  .
0x20013FFC  97  .
0x20013FFD  01  .
0x20013FFE  00  .
0x20013FFF  08  .
0x20014000  ??  ?
0x20014001  ??  ?
0x20014002  ??  ?
0x20014003  ??  ?
```

- ✓ At power up, the SP on the top of the SRAM
- ✓ '??' means it is no possible to read data. In fact, there is no silicon(transistors) as the STM32L152 has 80KBytes of SRAM and SRAM region starts at 0x2000 0000
- ✓ Values in SRAM are random and can depend on previous programs that run on the microcontroller
- ✓ As SP = 0x2001 4000, the next available memory space is at 0x2001 3FFF

Stack / SRAM Layout

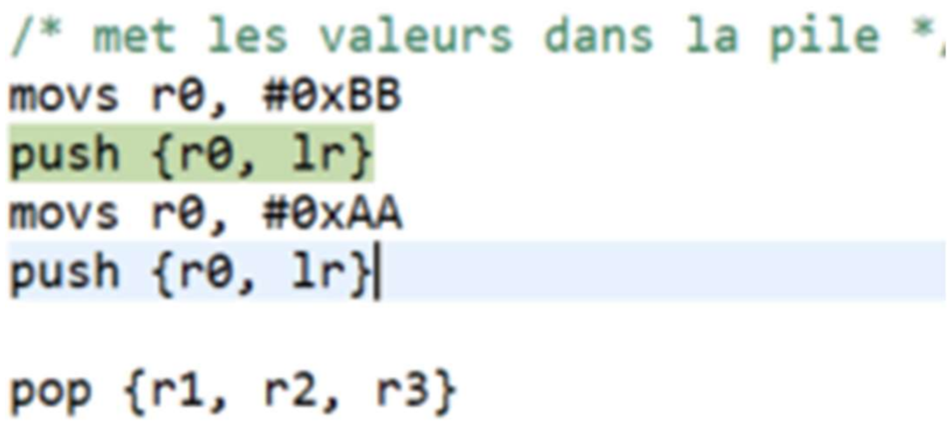


Typical arrangement 1: Stack region placed at the end of used SRAM space



Typical arrangement 2: Stack region placed at the end of SRAM space

Stack Program Example



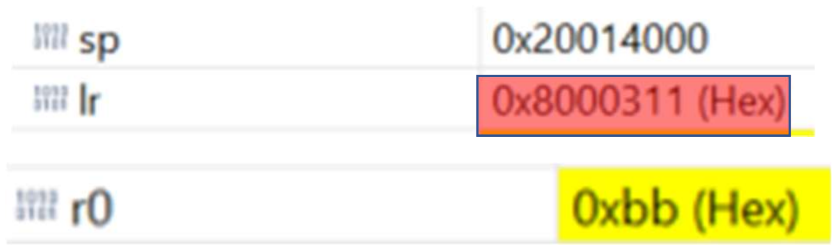
```
/* met les valeurs dans la pile *
movs r0, #0xBB
push {r0, lr}
movs r0, #0xAA
push {r0, lr}

pop {r1, r2, r3}
```

This program performs the following actions:

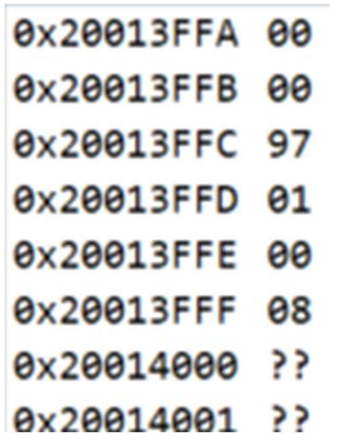
1) Load 0xBB in General Register r0
2) Push values in r0 and lr in the stack
3) Load 0xAA in the stack
4) Retrieve the 3 values of the stack and stores them in the three general registers r1, r2, r3

Stack Program Example Execution (1/2)

Before the push instruction

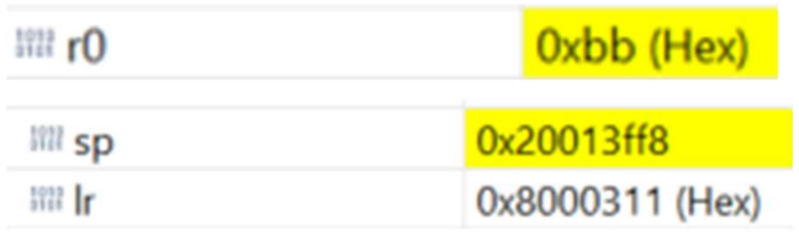


```
sp    0x20014000
lr    0x8000311 (Hex)

r0    0xbb (Hex)
```

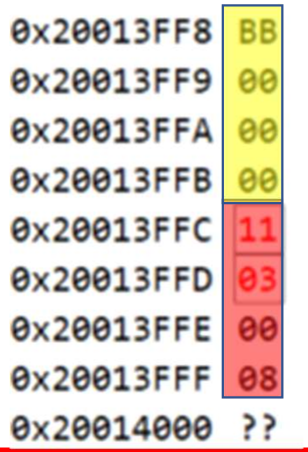


```
0x20013FFA  00
0x20013FFB  00
0x20013FFC  97
0x20013FFD  01
0x20013FFE  00
0x20013FFF  08
0x20014000  ??
0x20014001  ??
```

After the push instruction



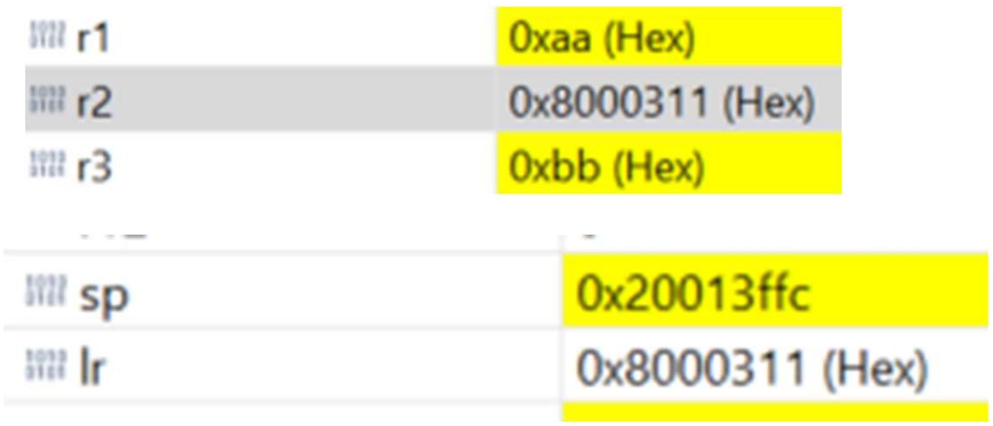
```
r0    0xbb (Hex)

sp    0x20013ff8
lr    0x8000311 (Hex)
```



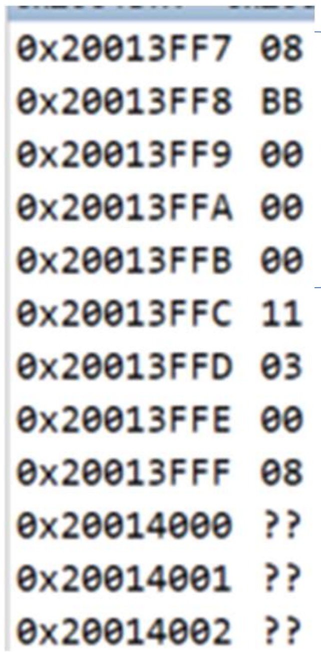
```
0x20013FF8  BB
0x20013FF9  00
0x20013FFA  00
0x20013FFB  00
0x20013FFC  11
0x20013FFD  03
0x20013FFE  00
0x20013FFF  08
0x20014000  ??
```

Stack Program Example (2/2)

After the Pop instruction



r1	0xaa (Hex)
r2	0x8000311 (Hex)
r3	0xbb (Hex)
sp	0x20013ffc
lr	0x8000311 (Hex)



```
0x20013FF7  08
0x20013FF8  BB
0x20013FF9  00
0x20013FFA  00
0x20013FFB  00
0x20013FFC  11
0x20013FFD  03
0x20013FFE  00
0x20013FFF  08
0x20014000  ??
0x20014001  ??
0x20014002  ??
```

Data is not erased but stack pointer allows to write on this memory space

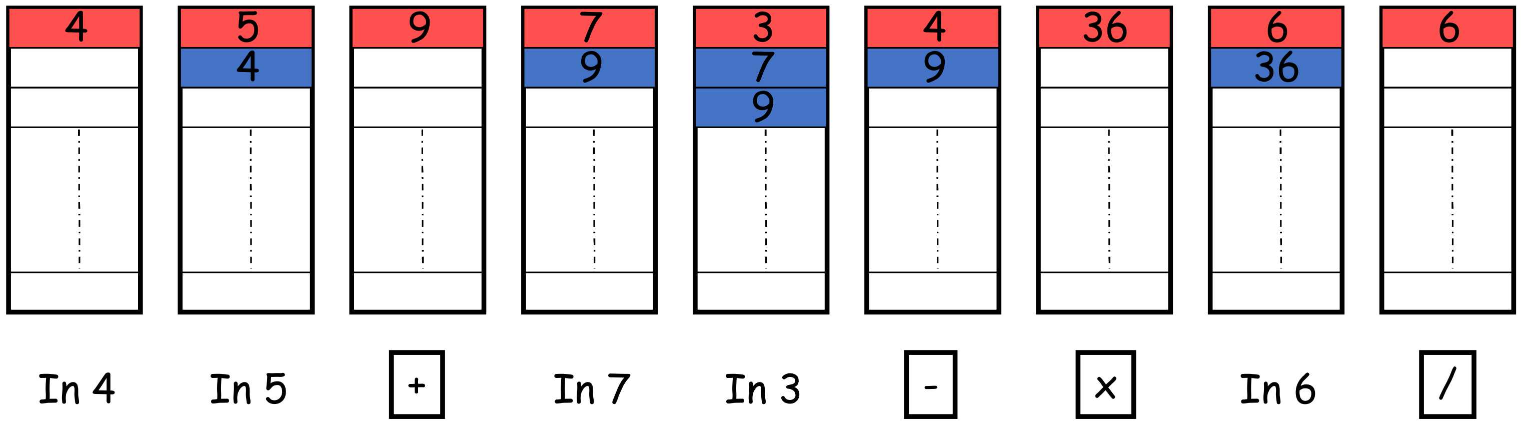
Notation Polonaise

- Notation classique : (A + B) * (C - D)
- Notation polonaise : AB+CD-*
- Structure de l'unité de traitement
  - Organisation des registres en pile
  - Analyse d'une expression
    - opérande empiler dans la pile
    - opérateur
      - ✓ dépiler deux éléments
      - ✓ effectuer l'opération
      - ✓ empiler le résultat

La mémoire centrale

Exemple d'utilisation de la pile :

Notation polonaise : 4 5 + 7 3 – x 6 /
Pour faire l'opération (4+5)x(7-3)/6



4
5
4
9
7
9
3
7
9
4
9
36
6
36
6
In 4
In 5
+
In 7
In 3
-
x
In 6
/