* ARM GCC inline assembly.

- Inline assembly Code is used to write assembly Code inside a 'C' program.
- must follow Compiler syntax.
- GCC inline assembly Code Syntax:

ie: Assembly Instruction: MOV R0, R1

Inline Assembly:

$\underbrace{__}_{\text{optional}}$asm volatile ("MOV R0, R1");

volatile → type qulifier to Cancel optimisation [optional].

$\boxed{\text{ex}}$ Assembly Code of 4 Instruction:

```
LDR  R0, [R1]   //Load Reg.
            address
LDR  R1, [R2]
ADD  R0, R1
STR  R0, [R2]   //Store Reg.
```

≡

```
__asm volatile ("LDR R0,[R1]\n\t",
                "LDR R1,[R2]\n\t",
                   ⋮
                ........ );
```

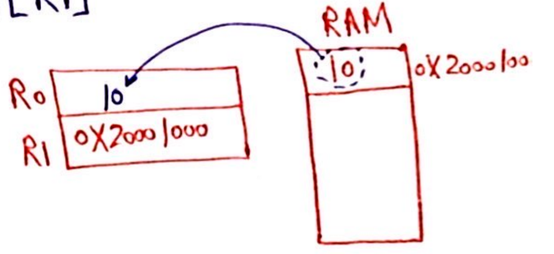
▣ why use inline assembly?

1. ~~(a)~~ move Content of C Variable data into ARM registers, like R0, R1 . . . .
2. Move ~~register~~ Content of CONTROL Register to C variable

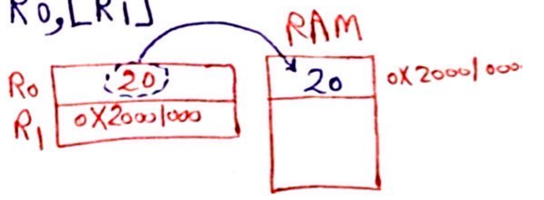
LDR Rt, [Rn]

- LDR ↓ Load Register
- Rt ↓ Destination Register
- [Rn] ↓ [register holding memory Address]

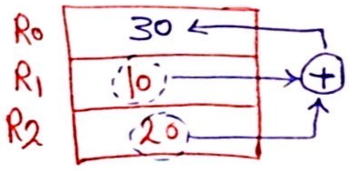
▪ LDR R0, [R1]



▪ STR R0, [R1]



▪ ADD R0, R1, R2



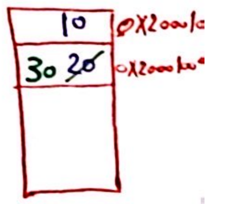
$\boxed{\text{ex}}$ * load 2 values from memory, Add them and store the result back to the memory using inLine assembly Ins.

1) Put Inside R0 address: 0X2000/000

LDR R0, =#0X2000/000
(# → Immidate)

2. Put Inside R1 address: 0X2000/004

LDR R0, =#0X2000/004



3. Load value from address stored Inside ~~R0~~ R1 inside ~~R2~~ iside R0

LDR R2, [R0]
LDR R0, [R1]

ORR R0, Number
ADD

4. Add R0 & R2 & Store Insid R0

ADD R0, R2

5. Store The result Inside 0X2000/004

$\boxed{4}$ 8 STR R0, [R1]

· General form of inline assembly instruction:

-- asm volatile ("Code": O/P operand List: I/P operand List: clobber List);

Code → mnemonic

O/P operand List → o/p operand Sperated by Comma (اللى بكتب فيه)

I/P operand List → Inputoperant Sperated by Comma (اللى بقرا منه)

clobber List → used to tell Compiler about modification done by Code

□ I/o operands & Constraint Strings:

1- Each Input output operand is described by a Constant string followed by a C expression in round brackets.

syntax: "Constraint string" (C expression)

2- This is a procedure to mix processor Register & C variables:

Contraint string = Constraint Character + Constraint modifier

$\boxed{Ex}$ Move Content of C variable (var) to R0

```
-- asm volatile ("Mov R0, %0" : : "r" (var));
```

: → o/p
"r" → Constraint string, I/p

$\boxed{or}$

```
-- asm volatile ("LDR R0, [%0]" : : "r" (&var));
-- asm volatile ("STR [%0], R0" : : "r" (&var));
```

r: Constraint string used to instruct the Compiler to use register in data mainpulation

0% → refer to 1st operand
1% → // // 2nd //

=r → output operand register

$\boxed{Ex}$ move Content of CONTROL register to C variable:

```
u32 var_CTR;
-- asm volatile ("MRS %0, CONTrol" : "=r" (var_CTR) : : );
```

* In Special Register we used 2 Instruction only

~~MRS ⇒ move Register special~~

~~MRS mov special Register~~

MSR ⇒ move Register to special

MRS ⇒ move special to Register

Reset Sequence of the Processor

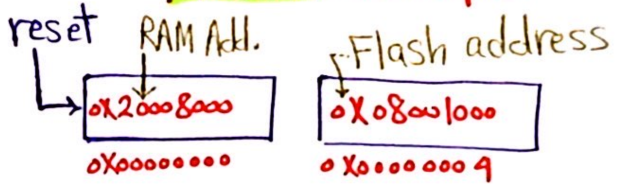
Reset Types:

1. System Reset: Reset All memory & periph. expect RCC & Backup domain Registers.
   1- reset Button
   2- peripherals generate reset:
      a- Watch dog Timer
      b- Brown out Detector
   3- SW Reset.
2. Power Reset: reset All memory & periph. Expect Backup domain Registers.
   → power on, off Cycle.
3. Backup domain Reset: reset all Backup domain Registers
   → Power off, on of External Battery.

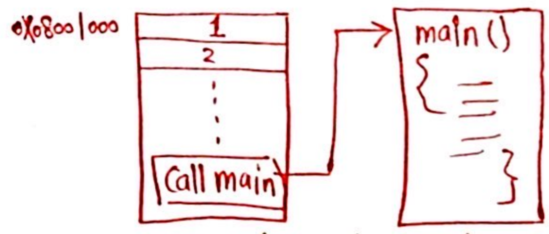
What happens when MCU undergoes Reset?

1. The PC is Loaded with the value 0x0
2. The processor Loads the value @ memory Location 0X0000 0000 into MSP (main stack pointer).
   ∴ MSP = value @ address 0X00000000
   ∴ Processor first initializes the stack Pointer.
3. The processor Loads value @ memory Location 0X0000 0004 into PC, that value is actually the address of the reset Handler.
4. So now pc jumps to the rest handler & start Excution.
5. a Rest handler is just C or asm. function written by you to carry out any Initilization required.
6. From reset handler you Call your main Function of the App.

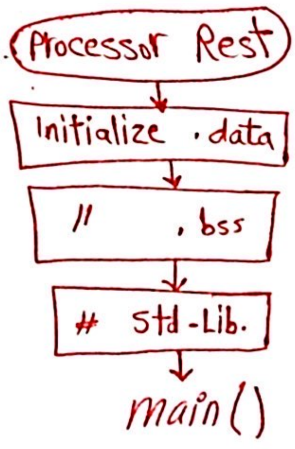
Visual Example.



1. after Reset, PC = 0X00000000
2. processor Read value @ Location 0 into MSP, MSP = 0X20008000
3. The PC increments one step (4B) to address 0X00000004
4. The PC is Loaded with value inside 0X00000004
   ∴ PC = 0X08001000
5. now processor Jumps to this addre Start Execution.



** Flash memory Start with Vector table @ address 0, So first Location in vector table Contain MSP Initial address, 2nd Location Contains reset Handler address.



** Access Level & T-bit **

recall that when processor is in unprivileged mode, it can't access some processor Registers, Accessing them will cause processor fault exception.

* T-Bit of the EPSR *

1. Various ARM processor support ARM & Thumb Instructions, that means the ability to switch between ARM & Thumb State.

2. Processor must be in ARM state to Execute instruction from ARM set, and in Thumb state to execute inst. from Thumb set.

3. if (T) bit is set, processor think that the Next instruction which it is about to Execute is from Thumb Set, if it is cleared (0) it thinks it is about to Execute from ARM Set.

4. Cortex Mx processor don't Support ARM State, Hence T-bit must be always (1), failing to maintain this is illegal, & will result in Usage fault Exception.

5. The LSB [bit0] of PC is Linked to this T-bit, when you load a value or an address into PC The ~~[illegible]~~ bit(0) of this value is loaded into T-bit.

6. Hence, Any address, you place into PC must have its 0th bit as (1)

7. This is taken care by compiler need ~~to~~ not worry most Time but you should be careful when you write direct to PC or initialising funtion pointer with a raw address.

مهم جداً

Ex: if we want decleratio Pointer to function & pass address of another func

```
void (fptr*)(void)= 0x80000
```

X X

```
void (fptr*)(void)= 0x80000+1
```

Bit Banding :

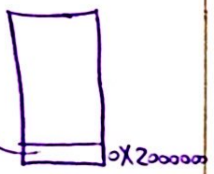
1- Capability to address a single bit of amemory address

2- This feature is optional, mcu manufacturer may Support it may not support it, refer Data Sheet.

3- If you have 1KB memory, each byte ~~hase~~ has address, Then you have 1024 Location (different) to read byte @ address 0X20000000
This is Called byte accessing

→ using instruction LDRB

LDRB R0, [R1]



→ what if we want to Read 2B:

using LDRH ⇒ Load Half word

→ what if we want to Read 4B:

Using LDR ⇒ word accessing.

* what if we want to read 1bit or 4b for Ex?

→ This is half byte accessing, not Support You will have to read one byte, modify then write back.

Bit Banding Can allow you to address individual bits [bit addressing feature]

→ without Bit Banding, if you want to Set bit(0) in memory:

```
LDR R0, [R1]
ORR R0, #1
STR R0, [R1]
```

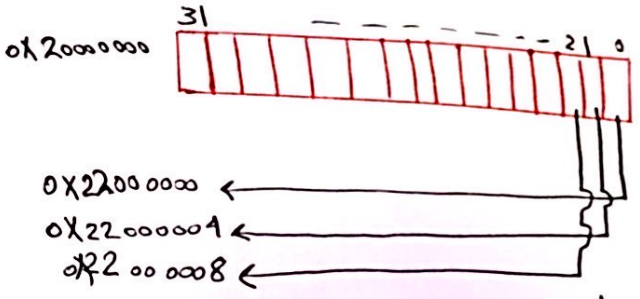
* with Bit banding:

→ STR #1, [A_A]
↳ alias address of th bit address

→ Bitbanding Can be allowed on SRAM, Peripheral region, each region has 1MB bit band region.

→ Access the bit band~~ing~~ region using bit band Alias

→ for SRAM : bit band region starts @ 0X20000000
→ bit band Alias start @ 0X22000000



* Bit Banding Provides atomic operations to bit data as accessing happens in one instruction.

* Calculating of bitband Alias Address

General formala:

Alias address = Alias Base + bit pos *4 + 32 *(Bit Band Memory − bit band base)

0X22000000 ↗ Alias Base; 0X20000000 ↗ bit band base

(EX) 7th bit of memor location 0X20000200

Alias address = 0X20000000 + (7*4) + 32 (0X20000200 − 0X20