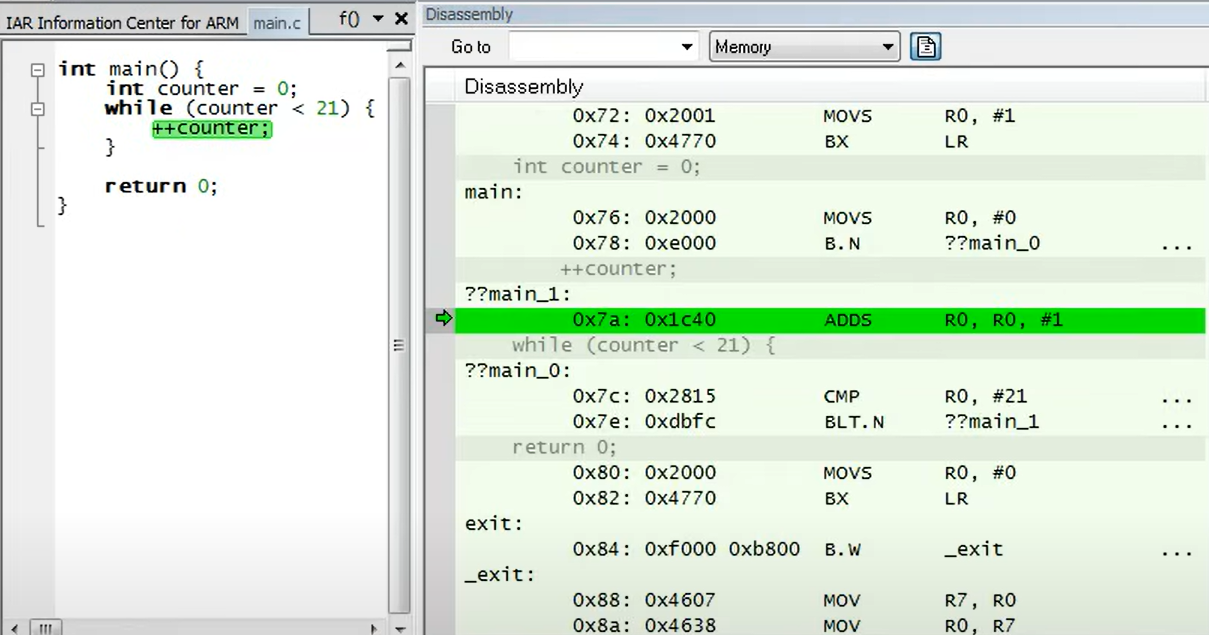
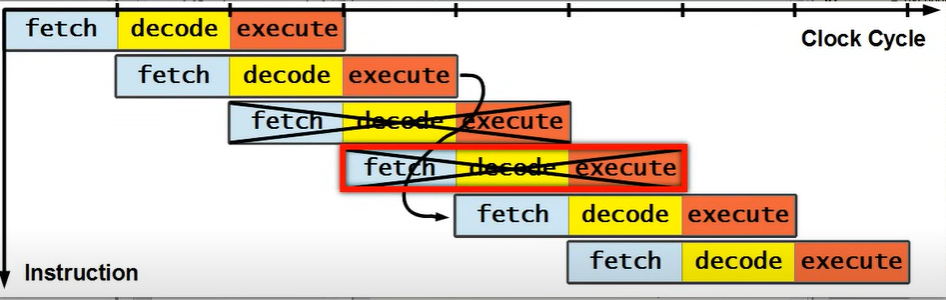
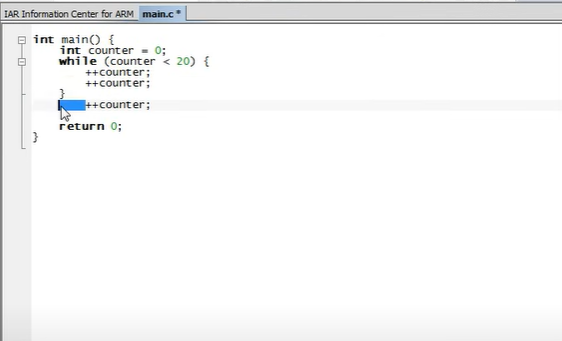
**Embedded C Programming**

* How do computers count?  
  - The computer stores data in binary, in the memory.
* The Hexadecimal system is preferred by programmers since it maps exactly to the binary system.
* **Int (Integer):** +ve range = 0-0x7FFFFFFF and -ve range = 0x80000000-0Xffffffff (-1).
* How to change the flow of control in your code?
* Loops are used to execute the same code for the required number of iterations.
* But loops carry with them an overhead what is it for that I have attached a SS of the disassembly view of the loop.  
  

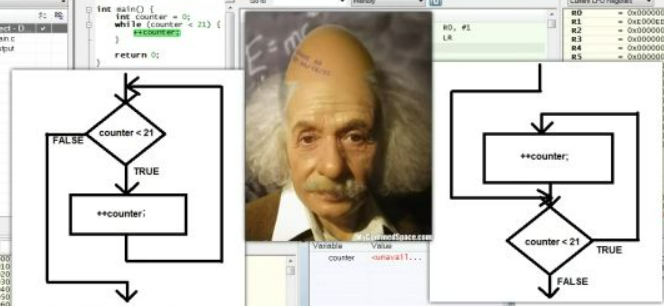
**The** **overhead**: There are branch instructions used, that stall the pipeline for a few cycles since the normal flow is disrupted. As can be seen the pipeline partially processes instructions which are discarded and then restarts execution from the new instructions since a branch instruction was executed.



* Hence to reduce this overhead in time constrained applications we need to unroll the loop to increase the speed of execution. Like this



* Compiler Magic: Even though the flow in the while loop is first compare and then increment the compiler uses a different approach as follows.



But both the flows are same then how can I say the compiler is smart?

Well the answer can be seen in the disassembly view check this out.

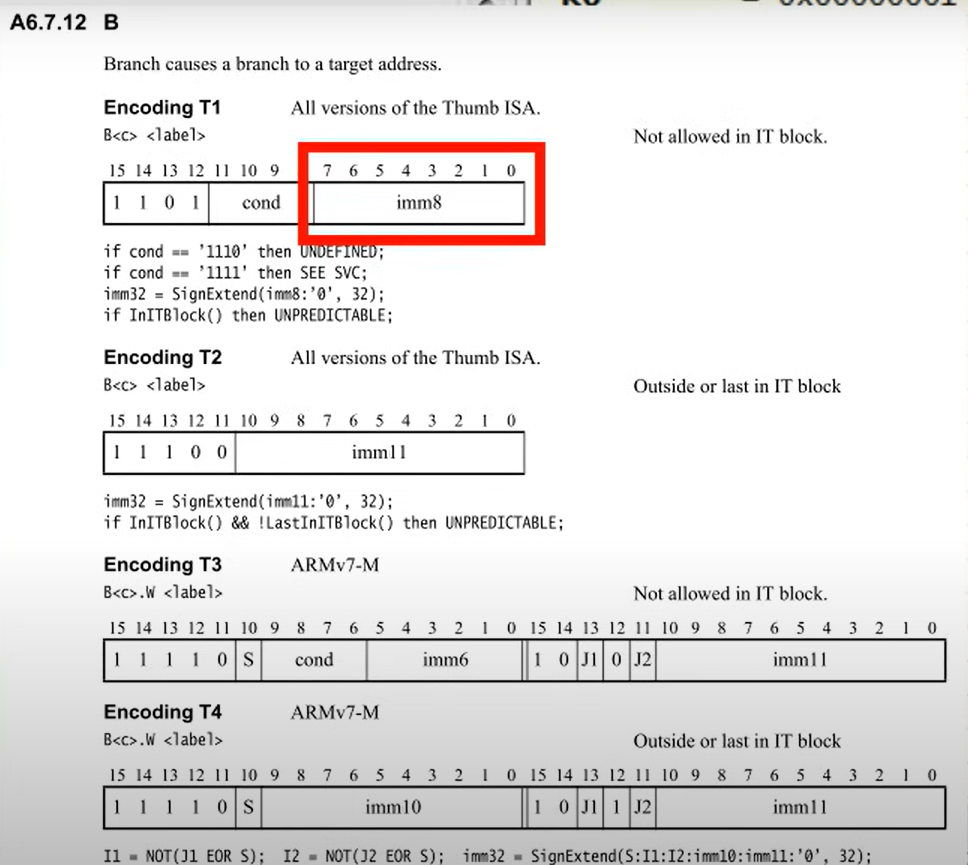


In the assembly code generated by the compiler it needs just a single branch instruction at the end of the loop, rather if the assembly code was generated as is then there would be two branch instructions BLT.N and BLT.P (to branch to return 0, when 21<21).

* **How does the branch instruction know which address to branch to**?

Well based on the arm cortexs documentation this information is encoded in the instruction.

So in our example 0xdbfc (BLT.N) 0xd means encoding type T1 (look in figure below), 0xb means the LT (less than) condition type, and 0xfc is the offset to be added to current PC to jump to the other address. \*Now offset is a signed quantity hence it means it’s represented in twos complement therefore 0xfc = -4 now subtract -4 from current PC value to get the new PC value.



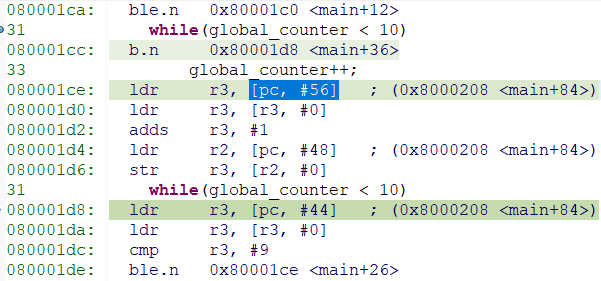
**Pointers**

Before understanding pointers let us understand the different memory sections in the MCU?

But before that lets understand the compilation process.  
Ill add the link to the page that explains this in a very clear and concise manner.

<https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/>  
  
Now at this point you understand that we need to perform the compilation process before your blinky.c code is uploaded on your dev board.

Now back to understanding of the different memory sections in the MCU  
refer to this legendary video by ArtfulBytes: <https://www.youtube.com/watch?v=hyIEUCIVhQQ&t=895s>  
Ill try giving you guys an overview here. After the compilation process we get the .o file which is a file created by the compiler, what’s special about it is that the compiler has segregated and made more sense of the blinky code and placed the global variables, the code and initialization of any global variables if any, under certain sections accordingly.

* The sections are mentioned below here . i.e.  
  .text, and .const -> Flash memory (Kind of like an EEPROM) Non-volatile memory  
  .BSS, .DATA, .HEAP, and .STACK -> RAM memory  
    
  Now this .o file is given to the LINKER script, which has the information regarding the MCU’s memory layout and will allot this memory to the above mentioned sections accordingly and generate an executable file.   
    
  The flash programmer takes this executable file and writes it to the flash memory. So in this flash memory of the MCU there is a startup code that is executed after it boots-up which handles the initializations of the RAM, which means any global variables initialised with some values is setup inside RAM, sets the Stack pointer as well. There could be something more happening down there but for now this is all you need to know.  
    
  Which variables go to which sections is explained in the video link provided above.  
    
  Why all of that understanding though?  
  that’s because when you go through the assembly code there are certain questions that might pop-up in your big brain and they go like this
* incase of the global counter, does the compiler load the static memory address from PC in r3? ldr r3, [pc, #56]
* ldr r3, [pc, #56] here the memory address of the global\_counter variable is loaded from the static memory. This happens because global variables in embedded systems are stored in a **fixed location in memory (static memory)**, and the compiler generates code to access that memory location indirectly.
* how does the compiler know which is the address of global\_counter what if there were more global variables?
* The compiler generates **symbols** for global variables, and their actual memory addresses are resolved by the **linker** using a linker script.
* Each global variable gets a unique memory address, so there is no conflict.

* **Pointers a hero and a villain**

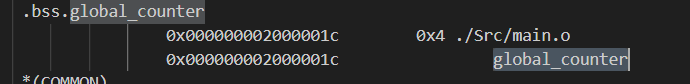
In the microcontroller there is a need to set a few registers before we begin with the blinky code. The data-sheet provides us with the memory address to these registers so how can we make use of the of these addresses? POINTERS!!!

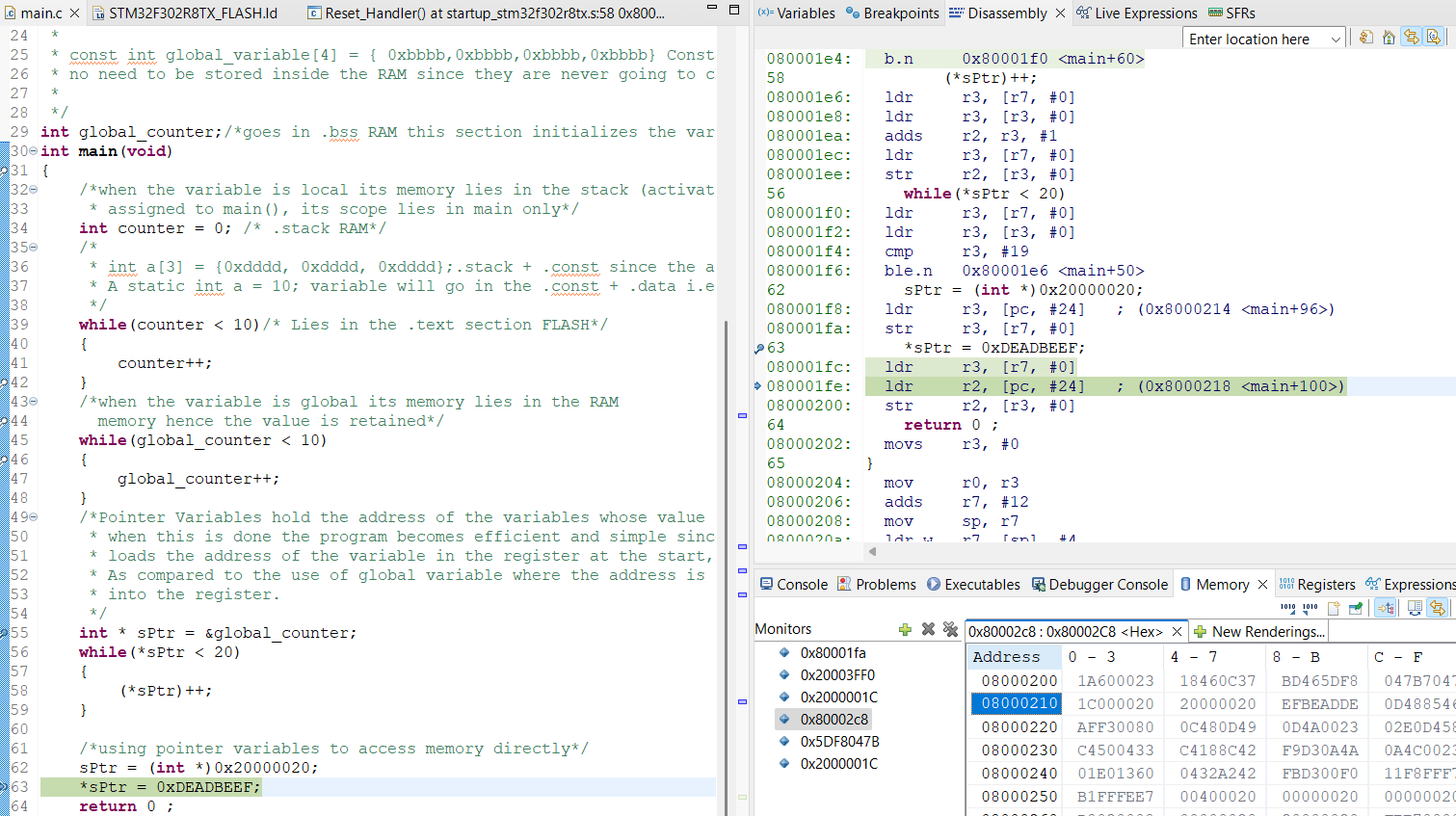
Pointers are variables that hold the address of the variable whose value that they will be pointing to.

**Syntax:** int\* ptr;  
Here we read the above statement backwards to make sense, the variable ptr is a pointer (\*) that points to a integer type value.

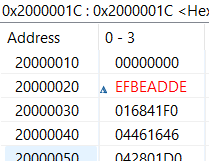
**How do Pointers make a difference in the assembly code generated by the compiler?**So when a pointer variable is used the address of the of the variable global\_counter is pushed onto the stack this can be seen in the first LDR and STR instruction (check Pointer Variable disassembly view Screenshot below)

* so since the .bss section in RAM contains the global\_counter variable, we need to traverse to this section by getting its address from the FLASH memory, this is calculated by offsetting the PC by 36. Ldr r3,[PC,#36] giving r3=0x8000208 this is the memory address in the flash memory (begins from 0x8000000) that contains the RAM address (0x2000001c) of the global\_counter variable which lies in the .bss section of the RAM .

  
this is the screenshot of the MAP file generated by the linker telling me at what address the global\_counter is stored.

* Then the STR instruction stores this RAM address of the global\_counter onto the stack i.e. the stack address here is stored in r7 = 0x20003ff0  
    
  as can been seen from the memory view the SP 0x20003ff0 points to the address 0x2000001c  
  so now anytime global counter needs to be called or updated global\_counters address is read from the stack and the operations are performed on its value.  
    
  Now when we need to access some registers related to GPIO etc. we can provide the address of these registers directly to pointer variable as follows  
  

As can be seen sptr is initialised with 0x20000020 (memory address in RAM)  
and then we deference the pointer and initialize the pointer with a value i.e. 0xDEADBEEF this value was stored in the FLASH memory(begins from 0x8000000) (you can verify this by checking the ss out check DEADBEEF in the 3rd column) to know why check the code for the lesson\_2.

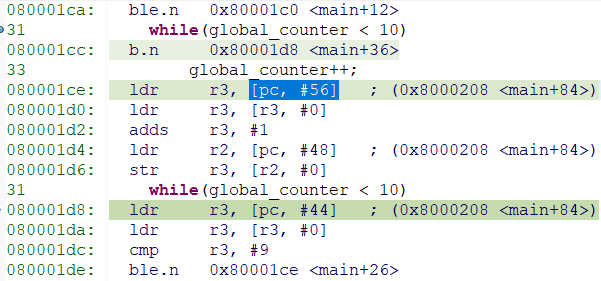
The compiler may throw an error or warning if you try doing this sptr = 0x20000020 the compiler will reject this until you typecast it. sPtr = (**int** \*)0x20000020  
now if I go check the address 0x20000020 I get the following  
  
pretty interesting right? Remember this as we will need these bad boyz in the future.

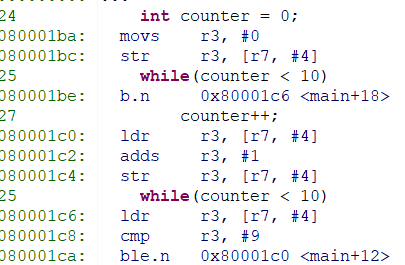
**What is RISC?**

Here the value from the memory needs to be loaded into a register before performing any operations on them. LDR

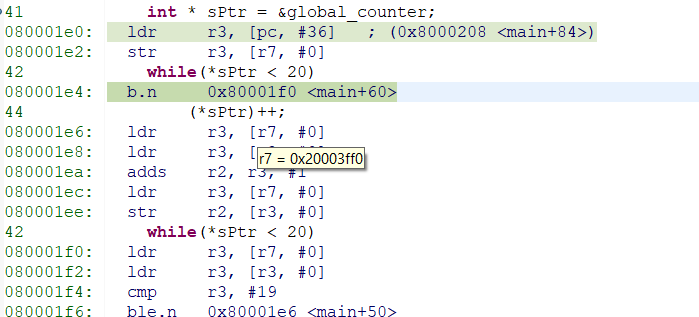
After the operations are done the result is stored in the memory using the STR instructions.

Global counter disassembly view:



Local counter disassembly view:  


Pointer Variable disassembly view:



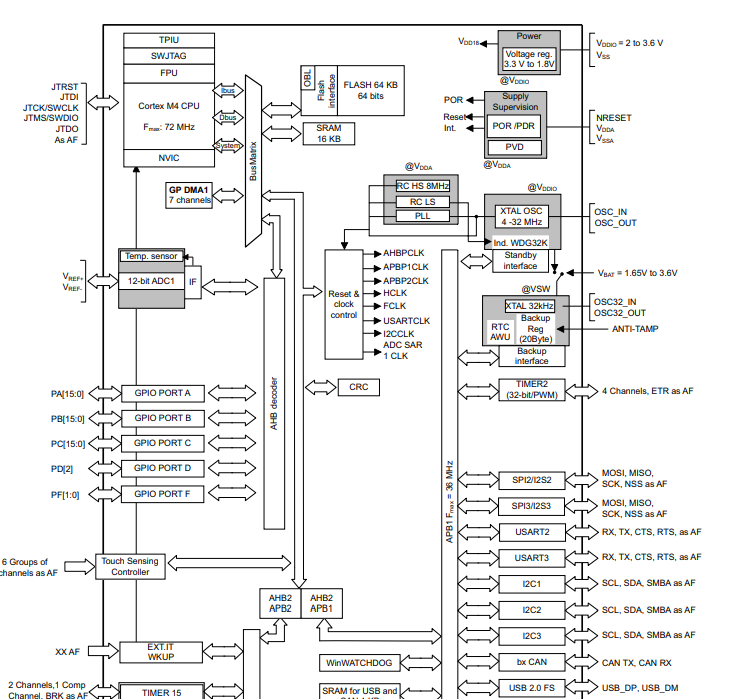
**Binky LED**

So in the last lesson pointers were the main focus, so lets start with the first use-case of pointers from the embedded perspective.

As I noted earlier, we use pointers to access the various peripheral registers in an MCU. Here in this lesson, we will begin by blinking a user LED on the stm32 board without using any libraries to configure and blink the LED by using the MCU specific datasheet, user-manual, and the reference manual. Initially these documents gave me the chills when I first opened them, but fear not as you don’t have to read it entirely to blink an LED.

Note: Just because I use a stm32 nucleo board to implement the blinky code does not mean you have to go buy the STM32, rather you could work with a MCU that is with you right now. All you need is to download the datasheet, User manual and Reference manual wrt your mcu on the dev board. Don’t be afraid to use your Arduino, esp32 etc.

Blinking the LED includes the following and this is what most of your libraries do or implement the blinking of an LED:

So, to begin with, take a look at this block diagram of the MCU  


The on-board user-LED is internally connected to the 32 bit GPIO-PORT B register, and if you intend to use any of these registers you would need to perform the following steps,

1. The ARM CPU is connected to the on-chip memory via an AHB. Hence, we enable the peripheral clock to the AHB bus. Here RCC\_AHBENR is the register we set to enable the clock to the GPIO port B register specifically.
2. Set the mode of PIN 13 to output or input. Set the bit corresponding to PB13 in the GPIO\_B\_MODER register to output mode by referring the reference manual in my case the output mode is 01.
3. Now the GPIO has a data register for input and output data. Here GPIO\_ODR register the bit no 13 corresponding to PB13 is set and reset to blink the LED.
4. By default, the PB13 is configured in the push-pull output type configuration.

As can be seen we used the registers inside the microcontroller to blink the LED and we did that with the help of pointers, we managed to set and reset a bit inside a particular register by making use of the provided start address and its offset (mentioned in the reference manual)

**ex:**

**#define GPIO\_B\_ODR\_OFST 0x14**

**#define GPIO\_B\_START\_ADDR 0x48000400**

**#define** GPIO\_B\_ODR (\*((**unsigned** **int** \*)(GPIO\_B\_START\_ADDR + GPIO\_B\_ODR\_OFST)))

Unsigned int \* : typecasts the GPIO\_ODR address which is 0x48000414. Typecasting the address is necessary else the compiler will throw an error.

(\*(**unsigned** **int** \*)): Dereference the pointer to set the desired bit to 1 or 0.

**Preprocessor and Volatile**Preprocessors in C are directives that are **executed before actual compilation** begins. They are used to **modify the source code** before it reaches the compiler. These directives start with #, and they are processed by the **C preprocessor (cpp)**.  
  
**Types of Preprocessor Directives**

1. **Macro Definitions (#define)** – Used to define constants or code snippets.
2. **File Inclusion (#include)** – Includes header files in the source file.
3. **Conditional Compilation (#ifdef, #ifndef, #endif, #if, #else)** – Allows conditional code compilation.
4. **Pragma Directives (#pragma)** – Provides compiler-specific instructions.

**What happens to the preprocessor directives during the compilation process?**

* During the first compilation step i.e. preprocessing, the preprocessor processes the directives before the compilation
* It replaces Macros, expands the header files, and removes the comments
* Lets consider the following code for example:  
  file example.c

#include <stdio.h>

#define LED\_PIN 13

Int main(void)

{

Printf(“\n the LED pin is %d”,LED\_PIN);

}

After preprocessing the file generated looks something like this  
file example.i  
// Expanded stdio.h contents (not shown completely here)  
 int printf(const char \*format, ...);   
  
int main()   
{   
 printf("LED is connected to pin %d\n", 13); // Macro LED\_PIN replaced with 13

return 0;

}

Hence these directives are not stored in the RAM or the FLASH memory because the value is substituted in the code during preprocessing.

**Volatile:**

The Volatile keyword prevents the compiler from optimizing the variable which it assumes that the value remains constant. Lets take the following code for example: a delay implemented using a while loop   
int main(void)  
{

Int counter = 0;

While( counter<=10000)

{  
 counter++;  
}  
counter = 0;  
return 0;

}

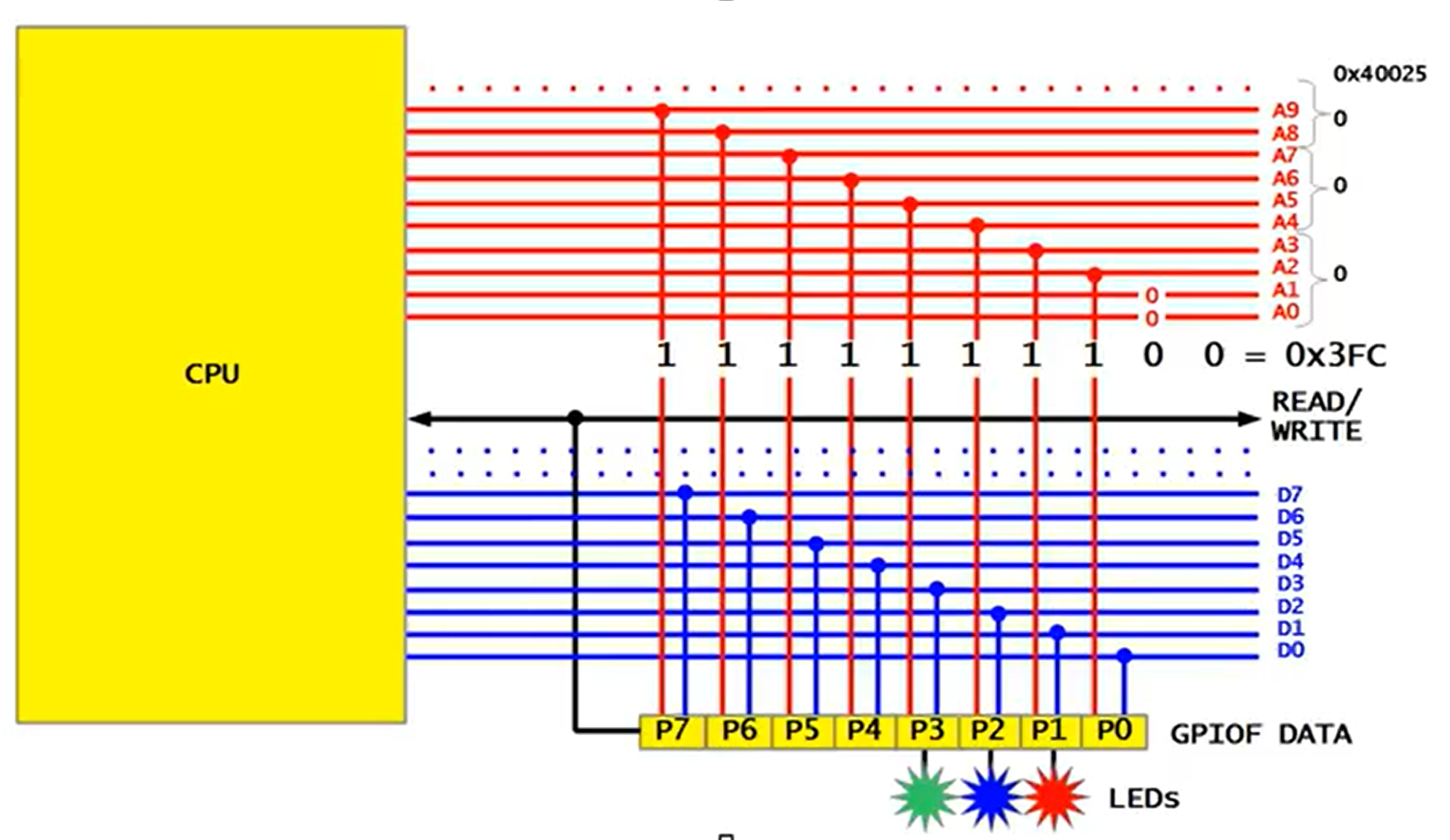
Since the counter variable will reset itself to zero after the counter value reaches the limit the compiler considers the while loop as redundant and will optimize the variable out unless, the counter is defined as a volatile variable, this is because from the compilers perspective the counter variables incremented value is overwritten by zero and it has not been used anywhere else in the code.   
  
Similarly we use volatile in the following cases,

* preventing optimization for **hardware registers**, since the register value can change outside of the CPU’s control.
* Incase of **interrupts**, here the occurrence of an interrupt is asynchronous or unpredictable, so if there is a shared variable that is modified during an interrupt the compiler will optimize this variable if volatile is not used.
* Similarly when multithreading if a shared variable is not volatile then this variable might get cached in a register.

Caching in a register what is that?

A common optimization technique is storing frequently used variables **in CPU registers instead of RAM** to speed up access. For instance  
int counter = 0;   
while (counter == 0)   
{   
 // Wait for counter to change   
}  
here the assembly equivalent is,  
MOV R1, #0 ; Store counter in register R1   
LOOP: CMP R1, #0 ; Compare register R1 with 0   
BEQ LOOP ; If still 0, keep looping  
 **Problem:** If another process (like an interrupt or another thread) changes counter in RAM, the CPU **won't notice** because it's still using the cached value in R1 (register). The loop **never exits**, even if counter actually changed in memory.

**Pointers and Arrays:**In a stellaris board **bit-banding** is used to perform **atomic writes**

What is bit-banding?  
Bit-banding is a **memory-mapping feature** found in some ARM Cortex-M processors that allows **atomic** bit-level access to memory regions (**Bit-banding** maps **each bit** of a register to a unique address). Instead of using **read-modify-write operations**, bit-banding allows a **single write operation** to modify an individual bit in memory. This works because each bit in the register has an alias address, hence we can make use of the array to access these specific bits. For instance, stellaris provides GPIO with 256, 32-bit data registers starting with the address 0x40250000 is the base address for the GPIO registers, and writing to the data register with an offset of 0x3FC which corresponds to this   
  
Courtesy Miro Samek Quantum Leaps

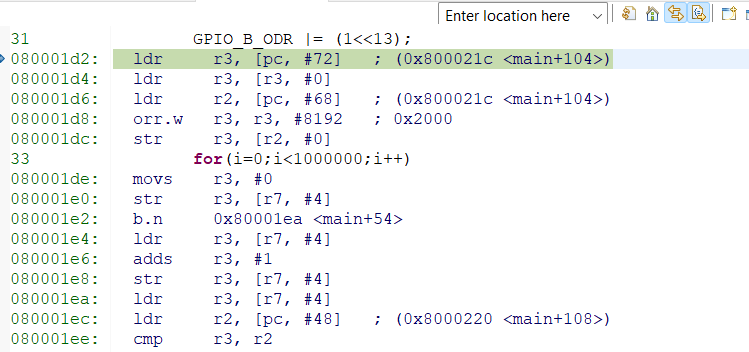
To set the RED LED bit the address is 0x40250008 the different ways to set this address is

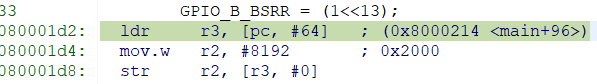
* (\*(unsigned long \*)(0x40250000 + (1<<3))) = 1; // Corresponds to 0x40250008
* (unsigned long \*)(0x40250000) + 2 = 1; // pointer arithmetic: pointer +2 means pointer will increment by 4 bytes ( since pointer is defined as unsigned long) so 0x40250004 and then 0x40250008
* #define GPIO\_DATA\_BIT\_R (unsigned long \*)(0x40250000)   
  GPIO\_DATA\_BIT\_R[2] = 1;

An **array** is a collection of elements of the same data type, stored in **contiguous memory locations**. The array name itself acts as a **constant pointer** to the first element of the array. For eg int a[4];  
Here, a refers to the **base address** (starting location) of the array.

* Since a behaves like a pointer, accessing elements using pointer arithmetic follows this rule:
* \*(a + 0) = a[0] (points to the first element)
* \*(a + 1) = a[1] (moves to the next element)
* The key idea is that **pointer arithmetic takes into account the size of the data type**.
* If a is an int array (4-byte elements), a + 1 moves **4 bytes** ahead in memory.
* If a were a char array (1-byte elements), a + 1 would move **1 byte** ahead.
* This behaviour ensures that the pointer always moves to the next element correctly, regardless of data type size.

Blink the LED using Atomic Write  
What is an Atomic Write?  
Atomic write means ensuring a **single operation is completed fully without interruption**, especially in a multi-threaded or interrupt-driven system.



After using the BSRR register for an Atomic write:  
  
  
Here BSRR writes to a special circuit in Hardware to modify only a certain bits in the ODR registers.

Thus, avoiding the hassle of reading, modifying, and writing when updating the GPIO\_B\_ODR register. Which is prone to being modified by an interrupt during this read, write, and modify cycles.