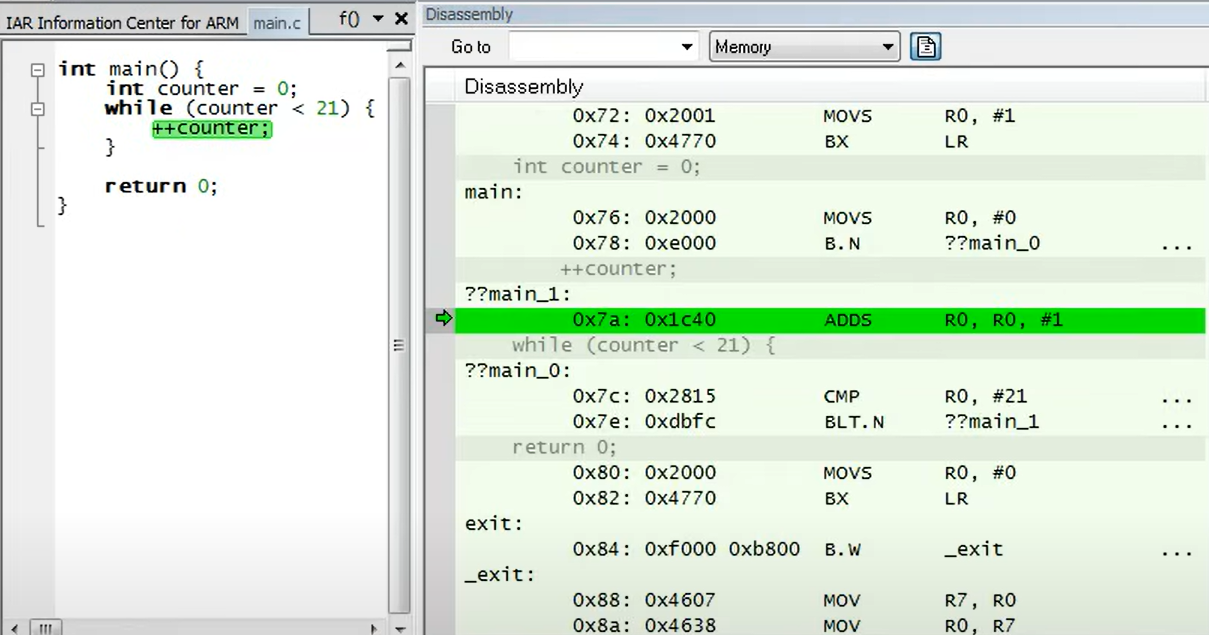
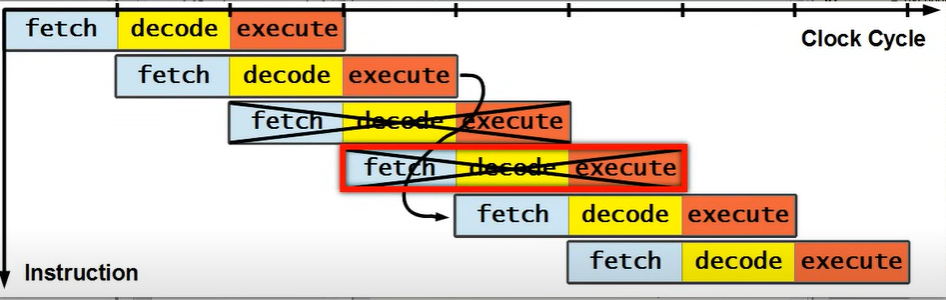
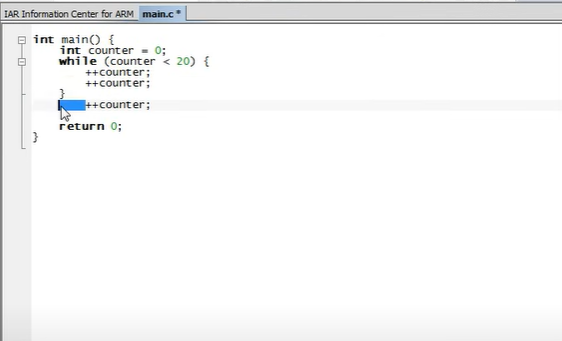
**Embedded C Programming**

* 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.  
  

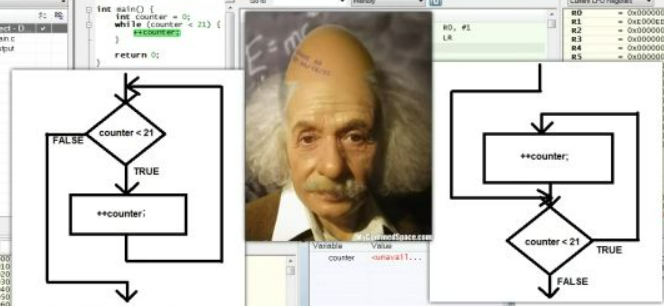
**The** **overhead**: There are branch instructions used, that stall the pipeline for a few cycles since the normal flow is disrupted. As can be seen the pipeline partially processes instructions which are discarded and then restarts execution from the new instructions since a branch instruction was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

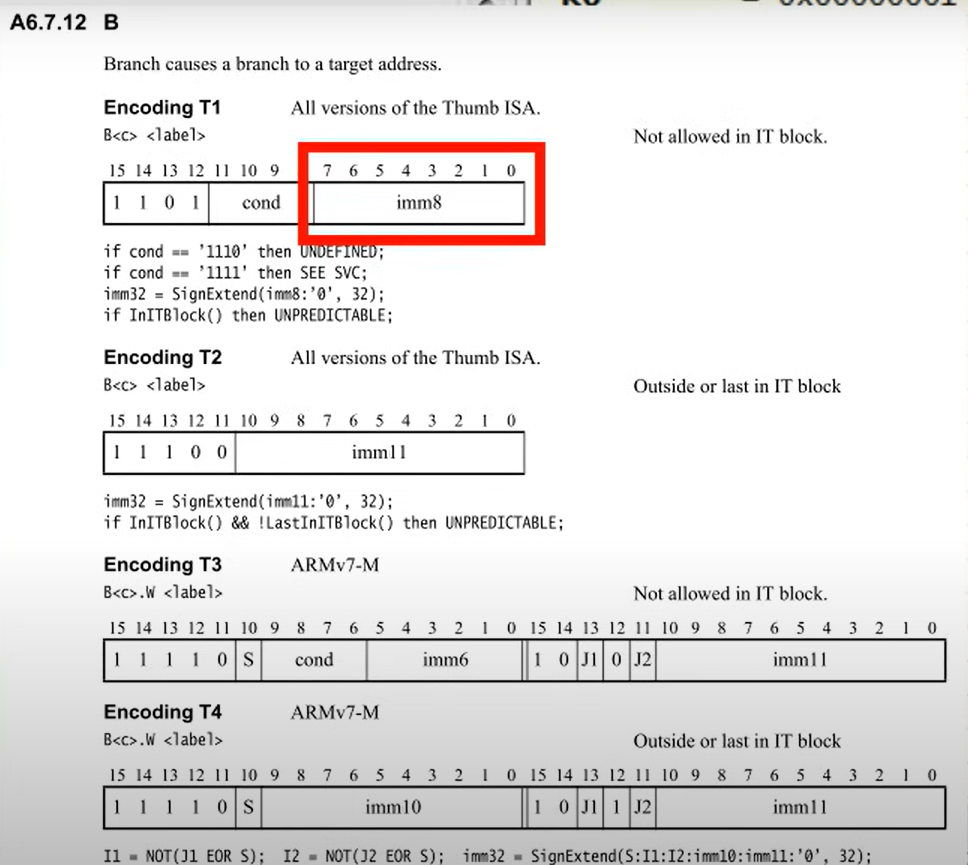


In the assembly code generated by the compiler it needs just a single branch instruction at the end of the loop, rather if the assembly code was generated as is then there would be two branch instructions BLT.N and BLT.P (to branch to return 0, when 21<21).

* **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.



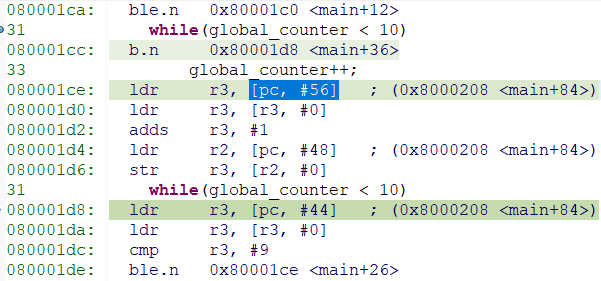
**Pointers**

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**

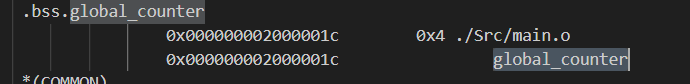
In the microcontroller there is a need to set a few registers before we begin with the blinky code. The data-sheet provides us with the memory address to these registers so how can we make use of the of these addresses? POINTERS!!!

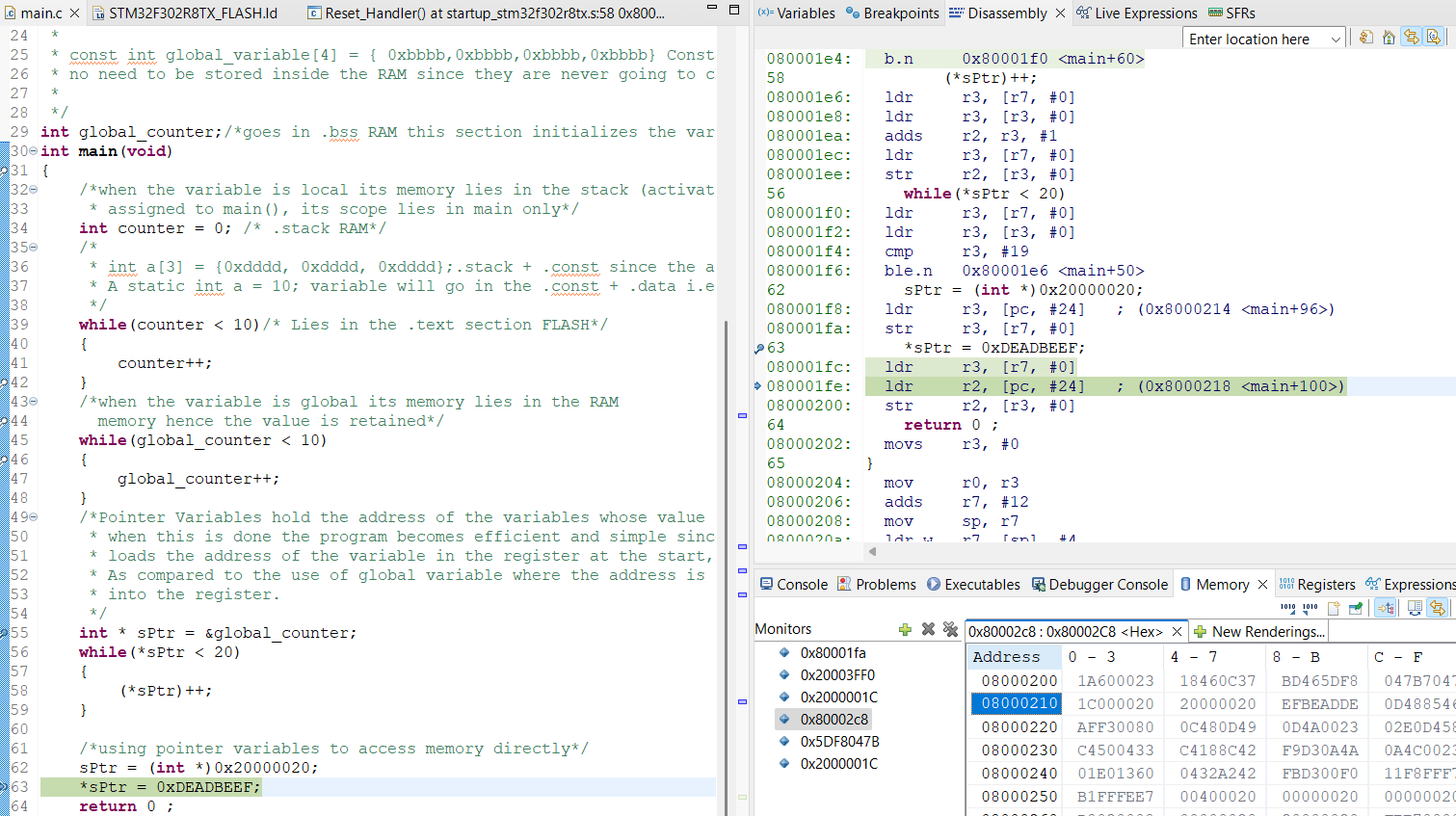
Pointers are variables that hold the address of the variable whose value that they will be pointing to.

**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.

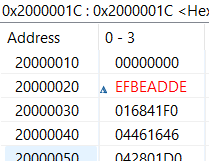
**How do Pointers make a difference in the assembly code generated by the compiler?**So when a pointer variable is used the address of the of the variable global\_counter is pushed onto the stack this can be seen in the first LDR and STR instruction (check Pointer Variable disassembly view Screenshot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

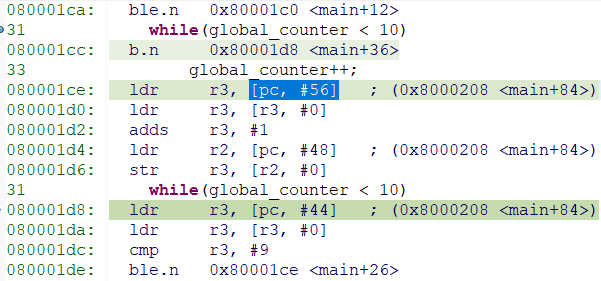
The compiler may throw an error or warning if you try doing this sptr = 0x20000020 the compiler will reject this until you typecast it. sPtr = (**int** \*)0x20000020  
now if I go check the address 0x20000020 I get the following  
  
pretty interesting right? Remember this as we will need these bad boyz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

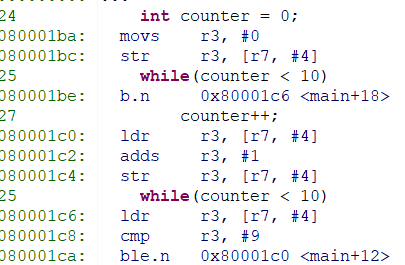
**What is RISC?**

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:

