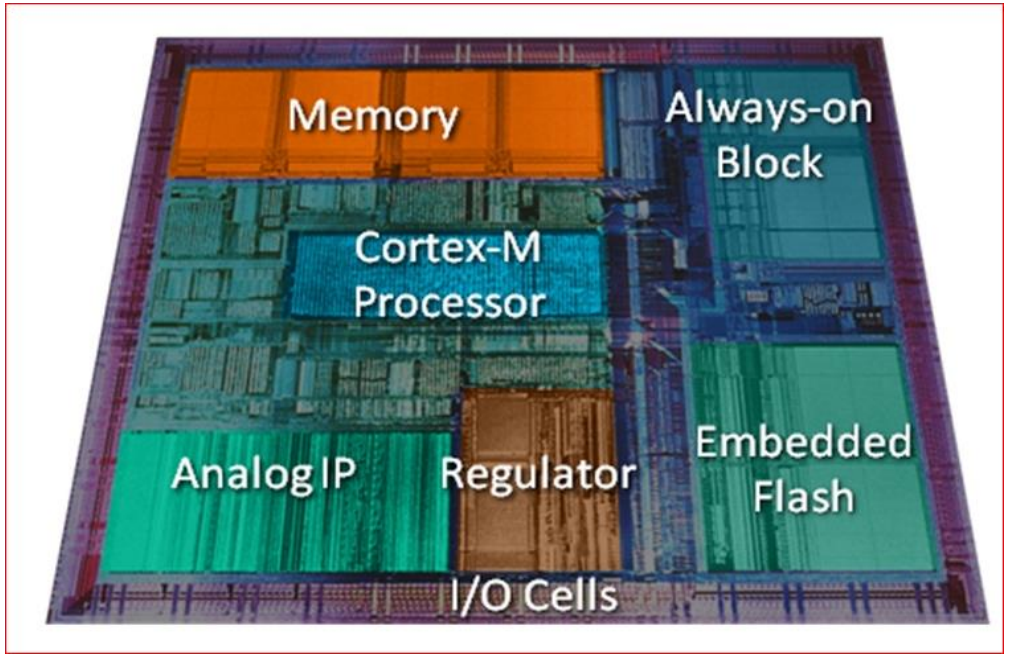
ARM SOC verification

SARVAKARMA
Committed to Deliver

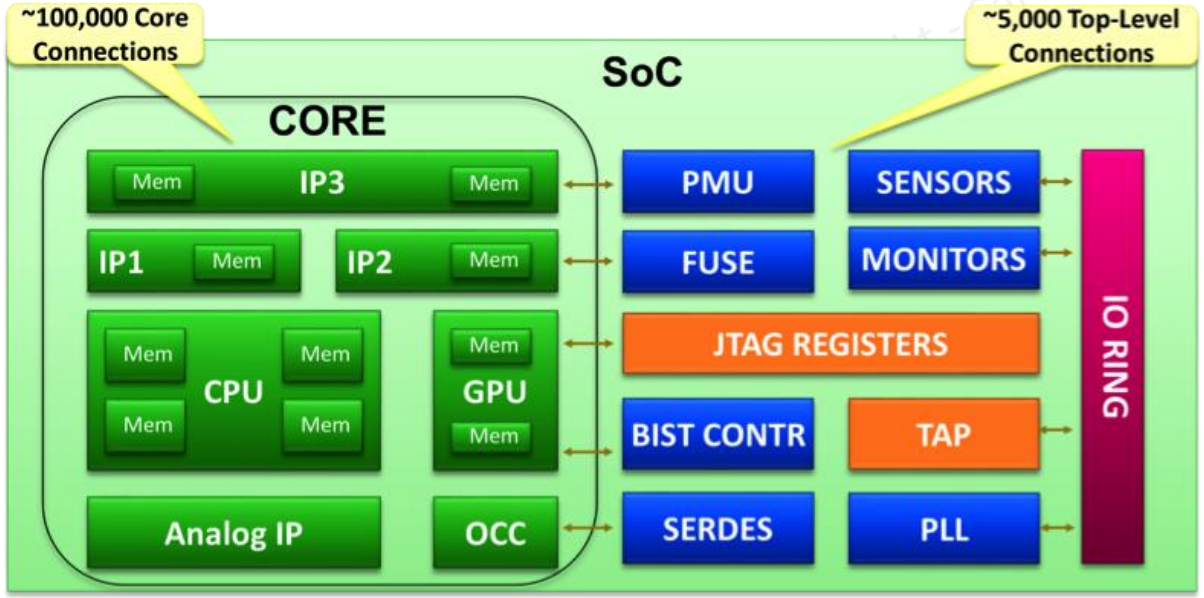
What is SOC?

- A system on a chip (SoC) combines the required electronic circuits of various computer components onto a single, Integrated Chip(IC).
- SoC is a complete electronic substrate system that may contain analog, digital, mixed-signal or radio frequency functions.
- Its components usually include a graphical processing unit (GPU), a central processing unit (CPU) that may be multi-core, and system memory (RAM).



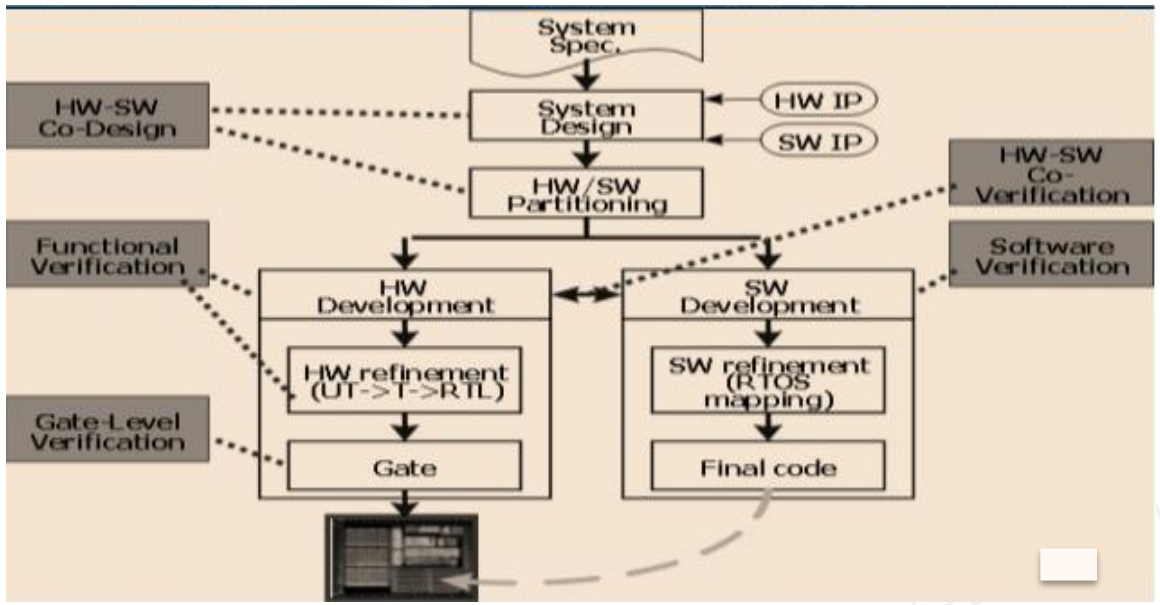
SoC and its components

- SOC includes both the hardware and software, it uses less power, has better performance, requires less space and is more reliable than multi-chip systems. Most system-on-chips today come inside mobile devices like smartphones and tablets.
- Instead of a system that assembles several chips and components onto a circuit board, the SoC fabricates all necessary circuits into one unit.
- An SoC usually contains various components such as:
  - Operating system
  - Utility software applications
  - Voltage regulators and power management circuits
  - Timing sources such as phase lock loop control systems or oscillators
  - A microprocessor, microcontroller or digital signal processor
  - Peripherals such as real-time clocks, counter timers and power-on-reset generators
  - External interfaces such as USB, FireWire, Ethernet, universal asynchronous receiver-transmitter or serial peripheral interface bus
  - Analog interfaces such as digital-to-analog converters and analog-to-digital converters
  - RAM and ROM memory



- ARM based SOC's typically used one or more ARM processor and other hardware to solve a specific computing problem.
- SOC's have RAM and ROM to store the software and a mix of hardware functions such as timers, interrupt controllers, UARTs, GPIO , DMA controllers, real time clocks and LCD controllers.
- A verification environment with a mix of C tests for debugging (for embedded processor) and Verilog/System Verilog test bench for monitors and automated checkers is used for successfully verification of an ARM based SoC design.

Co-Verification



- At the most basic level HW/SW co-verification means verifying embedded system software executes correctly on embedded system hardware.
- It means running the software on the hardware to make sure there are no hardware bugs before the design is committed to fabrication.
- The goal can be achieved using many different ways that are differentiated primarily by the representation of the hardware, the execution engine used, and how the microprocessor is modelled.
- Co-verification is often called **virtual prototyping** since the simulation of the hardware design behaves like the real hardware, but is often executed as a software program on a workstation. Using the definition given above, running software on any representation of the hardware that is not the final board, chip, or system qualifies as co-verification.

Benefits:

- Early access to the hardware design for software engineers.
- It allows software that is dependent on hardware to be tested and debugged before a prototype is available.
- It also provides an additional test stimulus for the hardware design. This additional stimulus is useful to augment test benches developed by hardware engineers since it is the true stimulus that will occur in the final product.
- In most cases, both hardware and software teams benefit from co-verification. These co-verification benefits address the hardware and software integration problem and translate into a shorter project schedule, a lower cost project, and a higher quality product.

SARVAKARMA
Committed to Deliver

Embedded SW Vs Workstation SW

Embedded Software

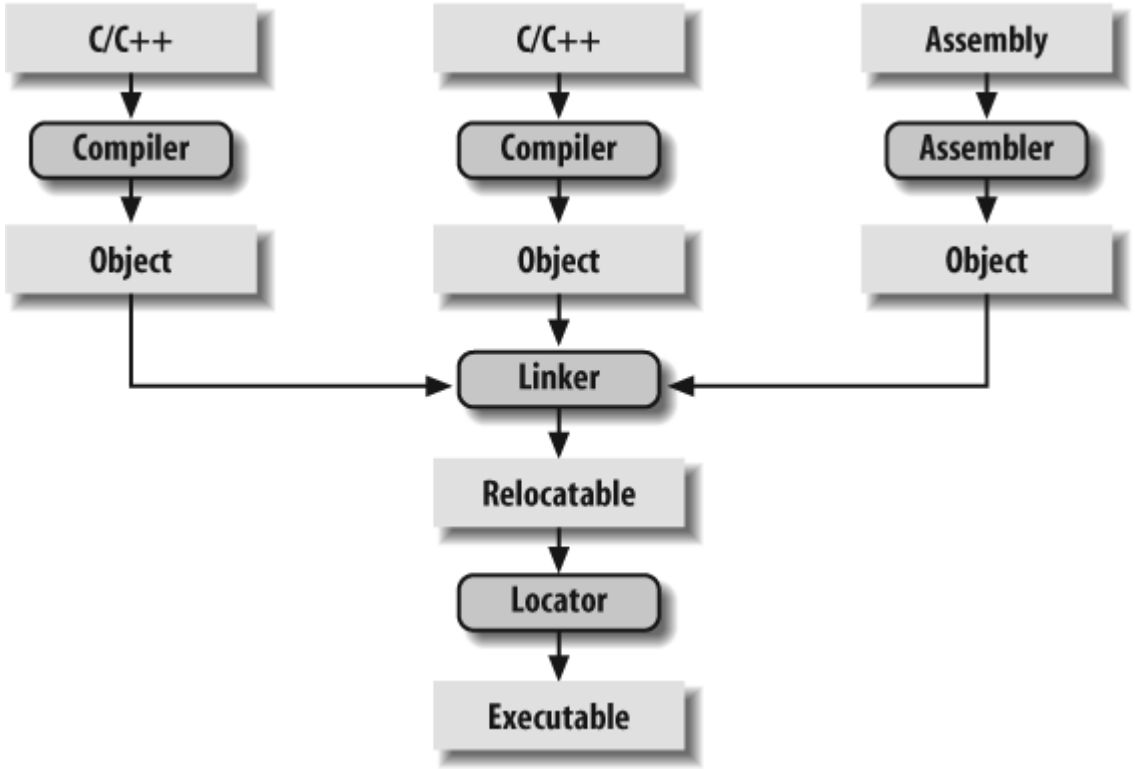
- The memory constraints of an embedded project. Memory layout is important, how much memory used is important, how to get the right data into the memory is important.
- For embedded systems, the software engineer must use a link map to specify the location of the memory and where to put ROM, RAM, and stack data.
- ELF (executable and linking format) files are object files produced by the compiler and linker that are binary representations of programs intended to be executed directly on a processor.

Workstation Software

- The modern operating system runs with large amounts of virtual memory.
- Software engineer does not care where in memory the program is loaded or about anything related to the physical addresses of the hardware.
- The program can easily use pre-compiled libraries to access common functions provided by the operating system to access graphics displays, network interfaces and input devices like keyboard and mouse.
- Software engineers write code in a text editor using the programming language of choice. The steps are to compile the source code into object files, then link those object files together, usually with some libraries provided by the compiler and operating system. The result is an executable file that can be directly executed on the computer.

C to Object Code

- In a SoC verification environment, C tests are written to exercise data transactions across various IPs in the system.
- C tests are converted to object code.
- The object code is then loaded into memory models for the processor to execute.



SARVAKARMA
Committed to Deliver

Compiler

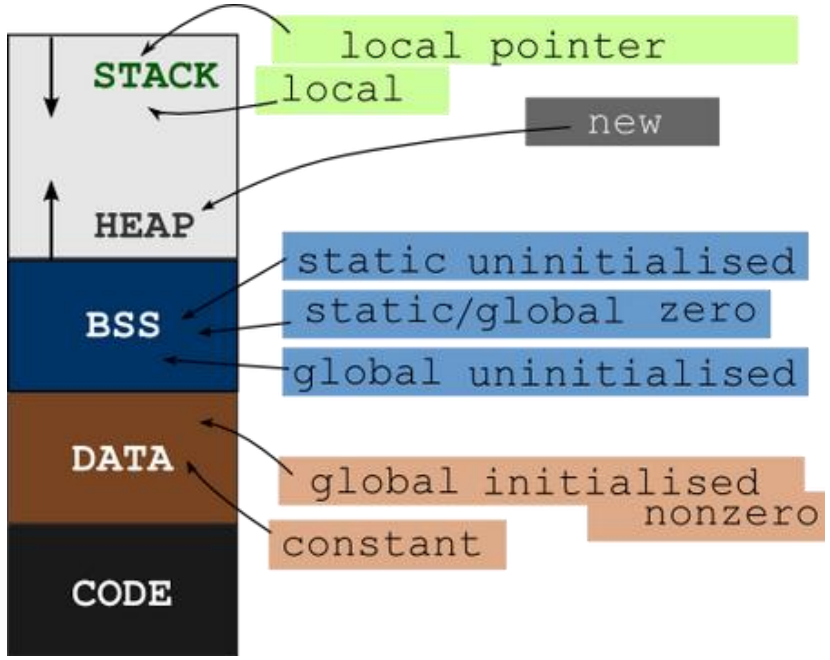
- A compiler, transforms source code written in a high-level programming language like C into the target processor's assembly/Object language.
- A few decades ago, a lot of software were written in assembly language because back then, compilers were not very efficient in utilizing the ISA of the target processor to produce dense and optimized code. Also, DRAM and memory was expensive. So, there was a big demand for Assembly Language coders to write dense and highly optimized codes for each processor. But, now-a-days there are softwares capable of producing very good quality and highly efficient assembly code.
- An example is GCC (GNU Compiler Collection), produced by GNU Project, which supports a variety of processor architecures and is in common use in the industry today. This is free software distributed under GNU General Public License (GPL).
- To know what version of GCC is installed in your server, type *gcc --version* in a linux console.
- To compile and generate an output object file, you can try
  - *$> gcc "C_file" -o "name_object"*
  - *$> gcc main.c -o HelloWorld*
- The contents of an object file can be thought of as a very large, flexible data structure. The structure of the file is often defined by a standard format such as the Common Object File Format (COFF) or Executable and Linkable Format (ELF).
- If you'll be using more than one compiler (i.e., you'll be writing parts of your program in different source languages), you need to make sure that each compiler is capable of producing object files in the same format; *gcc* supports both of the file formats previously mentioned. Although many compilers (particularly those that run on Unix platforms) support standard object file formats such as COFF and ELF, some others produce object files only in proprietary formats.

SARVAKARMA
Committed to Deliver

Assembler

- An assembler converts assembly language into an object file, which consists of machine language instructions, data and information needed to place instructions in memory.
- It has to keep track of all the labels used in branches and data transfer instructions in a *symbol table* and uses that information to determine addresses for each label.
- There are different formats for object files, and the most popular are **COFF** and **ELF**. Most object formats are structured as separate sections of data, each containing a certain type of data.
- The GNU C compiler (*gcc*) and assembler (*as*) can be configured as either native compilers or cross-compilers. The *gcc* compiler will run on all common PC and Mac operating systems. The target processor support is extensive, including Intel x86, MIPS, PowerPC, ARM, and SPARC

Header	Descriptive and Control information
Code Segment	Text segment, Executable code
Data Segment	Initialized static variables
Read-Only Data	Read only data, initialized static constants
BSS Segment (Block started by Symbol)	uninitialized static data, both variables and constants
External	Definitions and references for linking
Relocation Information	List of pointers stored in object file
Dynamic linking information	Links shared libraries needed by an executable
Debugging Information	Debugger can associate machine instructions with C source files



SARVAKARMA
Committed to Deliver

Diagnostic Software - Compile

- Contrast this software development process to that used when writing a diagnostic program for an embedded processor. The diagnostic program has no operating system and no standard libraries to communicate with input and output devices. Engineers will often call this operating environment *bare hardware*.
- The code usually goes something like the following:
  - Software starts execution from the reset vector. It is different for each architecture, but in the case of ARM it is usually address 0.
  - Setup interrupt vectors by programming addresses of interrupt handlers into the table of vectors. As an example, in the ARM architecture address 0x18 contains the address of the handler for the nIRQ interrupt (normal interrupt) signal.
  - Configure and enable other hardware. This can include enabling the instruction and data cache and programming the memory controller.
  - Setup a stack and call __main to prepare for program execution.
  - Start execution from the main() C function.
  - Everything up to the final step is done in assembly language to prepare the CPU to run even the simplest C program starting from main.
- Once the diagnostic program is coded in C and assembly language it is compiled into object files for the target processor.
- Compilation command to for Assembly program to create the object file to initialize CPU:
  - ***armasm -32 -bigend -checkreglist -CPU 5T -keep -apcs /inter -g -i include init.s -list init.lst -o init.o***
- Compilation command to for C program, which contains main() to create its object file:
  - ***armcc -ansi -c -cpu 5T -zo -bigend -fy -g -O2 -I include main.c -o main.o***
- The next step is to take all of the object files and link them together to form an ELF file named test.elf

SARVAKARMA
Committed to Deliver

Linker (/1)

- It would be a waste of time to compile and assemble all the source files if we had to edit a few lines in a single file.
- A lot of standard libraries are also part of the source code database and repetitive compilation and assembling of these files should be avoided.
- The work-around is to compile each file individually and link them up later on, and that is exactly what the linker does.
- It takes all the individual machine codes and combines them together via the following phases.
  - Place code and data modules symbolically in memory
  - Determine the addresses of
  - data and instruction labels
  - Place both internal and external references
- Basically, the linker uses the relocation information and the symbol table in each object file to resolve undefined labels, determines the memory locations of each module and finally produces an executable file that can be run on a processor.
- This is similar to the object file except that there will not be any unresolved references.
- The output of the linker is a new object file that contains all of the code and data from the input object files and is in the same object file format.
- It does this by merging the ***text, data***, and ***bss*** sections of the input files. When the linker is finished executing, all of the machine language code from all of the input object files will be in the *text* section of the new file, and all of the initialized in ***data*** section and uninitialized variables will reside in the new ***bss*** sections, respectively.
- The GNU linker **(*ld*)** runs on all of the same host platforms as the GNU compiler.

SARVAKARMA
Committed to Deliver

Linker (/2)

- Following files can be used as input to linker - armlink:
  - one or more libraries created by the librarian, armar
  - a symbol definitions file
  - a scatter file
  - a steering file.
- Linker command:

```
armlink -debug -scatter link.txt -remove -noscanlib -info sizes,totals,veneers,unused -map -symbols -xref main.o cp15init.o \
/tools/ads11/common/lib/armlib/c_a__un.b   /tools/ads11/common/lib/armlib/f_a_m.b \
/tools/ads11/common/lib/armlib/m_a_mu.b -list test.map -o test.elf
```

- An image can consist of any number of regions and output sections. Regions can have different load and execution addresses.
- To construct the memory map of an image, armlink must have information about:
  - How input sections are grouped into output sections and regions.
  - Where regions are to be located in the memory maps.
- Depending on the complexity of the memory maps of the image, there are two ways to pass this information to armlink:
  - Using command line options
  - Using a scatter file
- A scatter file is a textual description of the memory layout and code and data placement.
- It is used for more complex cases where you require complete control over the grouping and placement of image components.
- To use a scatter file,
  - specify --scatter=*filename* at the command-line

Locating

- The tool that performs the conversion from relocatable program to executable binary image is called a *locator*.
- It takes responsibility for the easiest step of the build process. In fact, you have to do most of the work in this step yourself, by providing information about the memory on the target board as input to the locator.
- The locator uses this information to assign physical memory addresses to each of the ***code*** and ***data*** sections within the relocatable program. It then produces an output file that contains a binary memory image that can be loaded into the target.
- Whether you are writing software for a general-purpose computer or an embedded system, at some point the sections of your relocatable program must be assigned actual addresses.
- In some cases, there is a separate development tool, called a *locator*, to assign addresses. However, in the case of the GNU tools, this feature is built into the linker (*ld*).

```
ENTRY (main)

MEMORY
{ ram : ORIGIN = 0x00400000, LENGTH = 64M
  rom : ORIGIN = 0x60000000, LENGTH = 16M}

SECTIONS
{
data :                  /* Initialized data. */
   { _DataStart = . ;
     *(.data)
     _DataEnd   = . ;
   } >ram

bss :                   /* Uninitialized data. */
  { _BssStart = . ;
    *(.bss)
     _BssEnd   = . ;
  } >ram

text :                  /* The actual instructions. */
  { *(.text)
  } >ram
}
```

- This script informs the GNU linker's built-in locator about the memory on the target board, which contains 64 MB of RAM and 16 MB of flash ROM.
- The linker script file instructs the GNU linker to locate the data, bss, and text sections in RAM starting at address 0x00400000.
- The first executable instruction is designated with the ENTRY command, which appears on the first line of the preceding example. In this case, the entry point is the function main.
- The start and stop addresses for this operation can be established symbolically by referring to the addresses as _DataStart and _DataEnd

Start up program and Load

- Startup code for C programs usually consists of the following series of actions:
  - Disable all interrupts.
  - Copy any initialized data from ROM to RAM.
  - Zero the uninitialized data area.
  - Allocate space for and initialize the stack.
  - Initialize the processor's stack pointer.
  - Call main.
- Typically, the startup code will also include a few instructions after the call to main. These instructions will be executed only in the event that the high-level language program exits (i.e., the call to main returns)
- If an operating system exists in the system, it will load the object file into memory and starts it.
  - Determine size of text and data segments from the executable file
  - Create an address space large enough for text and data
  - Copy instructions/data from executable file into memory
  - Initialize machine registers, point stack pointer to the right location
  - Calls the main routine of the program

SARVAKARMA
Committed to Deliver

Scatter file

Situations where scatter-loading is very useful:

Complex memory maps

- Code and data that must be placed into many distinct areas of memory require detailed instructions on where to place the sections in the memory space.

Different types of memory

- Many systems contain a variety of physical memory devices such as flash, ROM, SDRAM, and fast SRAM.
- A scatter-loading description can match the code and data with the most appropriate type of memory.
- For example, interrupt code might be placed into fast SRAM to improve interrupt response time but infrequently-used configuration information might be placed into slower flash memory.

Memory-mapped peripherals

- The scatter-loading description can place a data section at a precise address in the memory map so that memory mapped peripherals can be accessed.

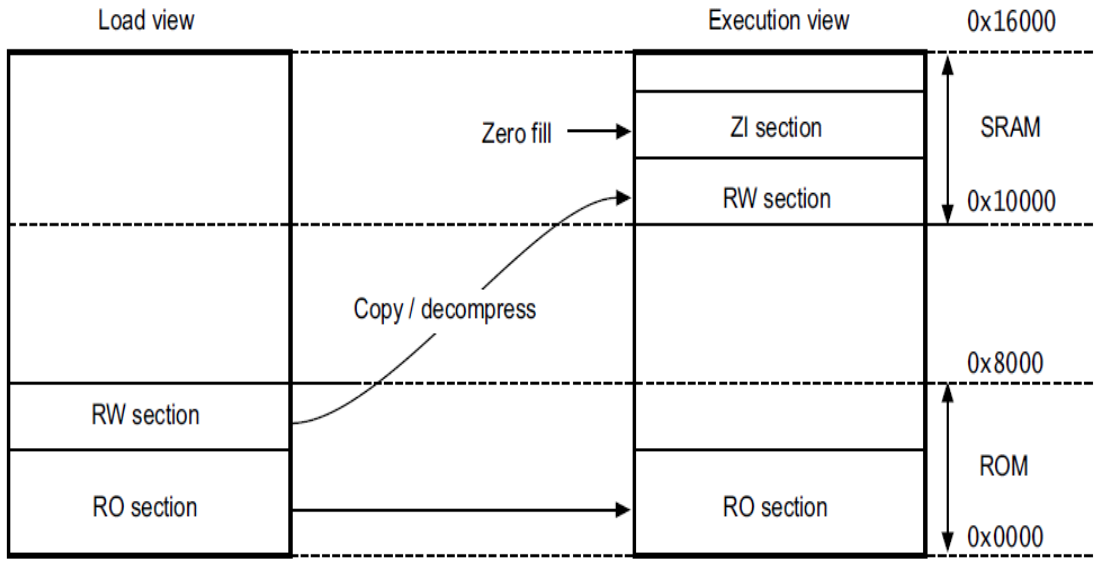
Functions at a constant location

- A function can be placed at the same location in memory even though the surrounding application has been modified and recompiled. This is useful for jump table implementation.

Using symbols to identify the heap and stack

- Symbols can be defined for the heap and stack location when the application is linked.
- Scatter-loading is usually required for implementing embedded systems because these use ROM, RAM, and memory-mapped peripherals

Scatter file Example




```
LOAD_ROM 0x0000 0x8000
; Name of load region (LOAD_ROM),
; Start address for load region (0x0000),
; Maximum size of load region (0x8000)
{
EXEC_ROM 0x0000 0x8000
; Name of first exec region (EXEC_ROM),
; Start address for exec region (0x0000),
; Maximum size of first exec region (0x8000)
    {
        (+RO)
        ; Place all code and RO data into this
        exec region
    }
SRAM 0x10000 0x6000
; Name of second exec region (RAM),
; Start address of second exec region (0x10000),
; Maximum size of second exec region (0x6000)
    {
        (+RW, +ZI)
        ; Place all RW and ZI data into ; this
        exec region
    }
}
```

ELF to Hexadecimal format

- Once the ELF file is complete and located correctly in memory, the final step is to translate the ELF file into a format that can be used to program a flash memory or otherwise load the data into the embedded system.
- A logic simulation environment will contain memory models for the various types of memory in the design. Common practice is to load the code into these memories at the start of simulation. One way to do this is to generate hexadecimal memory files that can be read directly into the memory models .

```
% fromelf -vhx -16x2 test.elf -output rom0.dat
```

- Verilog provides a system task, *$readmemh*, that will read the memory data from a file into the memory array. The task takes two arguments; the filename and the memory array name to load the data into

```
// Memory
reg [15:0] data [0:16383];
initial $readmemh("rom0.dat", data);
```

- The above example shows all of the steps to go from C and assembly language source code to a simulation memory model and finally to a complete simulation of the ARM design.
- When the reset of the microprocessor is completed it will begin to fetch instructions from memory and execute the program.

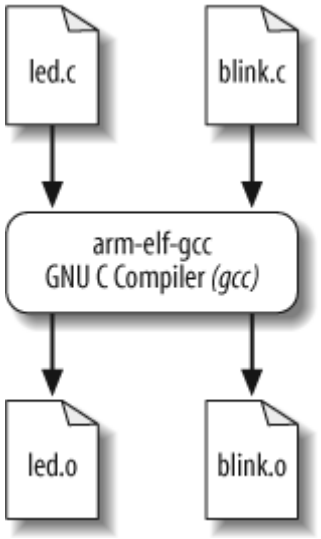
Execution Steps and commands (/1)

- Blinking LED example consists of two source modules: *led.c* and *blink.c*. The first step in the build process is to compile these two files. The basic structure for the *gcc* compiler command is:

```
arm-elf-gcc [        options        ]        file        ...
```

The command-line options are:

- -g To generate debugging info in default format
- -c To compile and assemble but not link
- -Wall To enable most warning messages
- -I../include To look in the directory *include* for header files



- Here are the actual commands for compiling the C source files:

```
# arm-elf-gcc –g -c –Wall -I../include led.c
# arm-elf-gcc -g –c -Wall -I../include blink.c
```

- The result of each of these commands is the creation of an object file that has the same prefix as the .c file, and the extension .o
- We now have the two object files—led.o and blink.o—that we need in order to perform the second step in the build process.
- For the third step, locating, there is a linker script file named viperlite.ld that we input to ld in order to establish the location of each section in the board's memory.

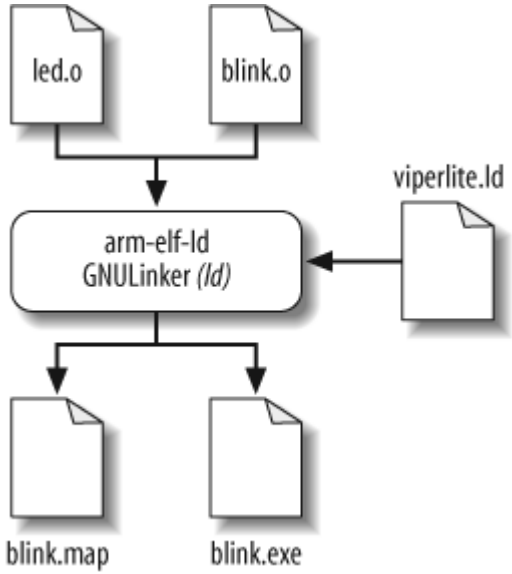
Execution Steps and commands (/2)

- The basic structure for the linker and locater ld command is:

```
arm-elf-ld [ options ] file ...
```

The command-line options we'll need for this step are:

-Map blink.map To generate a map file and use the given filename
-T viperlite.ld To read the linker script
-N To set the text and data sections to be readable and writable
-o blink.exe To set the output filename (if this option is not included, ld will use the default output filename a.out)



- The actual command for linking and locating is:

```
# arm-elf-ld –Map blink.map –T viperlite.ld -N –o blink.exe led.o blink.o
```

The result of this command is the creation of two files—blink.map and blink.exe—in the working directory
The *.map* file gives a complete listing of all code and data addresses for the final software image.

- There might be another time when you need an image file that can be burned into ROM or flash. The utility *objcopy* (object copy) is able to copy the contents of one object file into another object file.
- The basic structure for the *objcopy* utility is:

```
arm-elf-objcopy [ options ] input-file [ output-file ]
```

- For example, let's suppose we want to convert our Blinking LED program from ELF format into an Intel Hex Format file

```
# arm-elf-objcopy -O ihex blink.exe blink.hex
```

This command uses the –O ihex option to generate an Intel Hex Format file. Final output file is named as *blink.hex*

Makefile

- One can imagine how tedious the build process could be if you had a large number of source code files for a particular project.
- Manually entering individual compiler and linker commands on the command line becomes tiresome very quickly.
- In order to avoid this, a *makefile* can be used.
- A makefile is a script that tells the *make* utility how to build a particular program. (The *make* utility is typically installed with the other GNU tools.)
- The *make* utility follows the rules in the makefile in order to automatically generate output files from a set of input source files.
- Makefiles might be a bit of a pain to set up, but they can be a great timesaver and a very powerful tool when building project files over and over again.

```
target:   prerequisite
           command
```

```
XCC     = arm-elf-gcc
LD      = arm-elf-ld
CFLAGS  = -g -c -Wall \
          -I../include
LDFLAGS = -Map blink.map -T viperlite.ld -N

all: blink.exe
led.o: led.c led.h
       $(XCC) $(CFLAGS) led.c
blink.o: blink.c led.h
       $(XCC) $(CFLAGS) blink.c
blink.exe: blink.o led.o viperlite.ld
       $(LD) $(LDFLAGS) -o $@ led.o blink.o

clean:    -rm -f blink.exe *.o blink.map
```

```
# make
; make utility will search the current directory for a file
called makefile, if the file name is different use -f option

# make all
# make clean
# make -f Makefile1 clean ; file name different
```

Test bench

- With respect to test bench the following things are of concern
  1. How to boot the ARM processor
  2. How to start a specific C test of interest
  3. How to configure for various interrupts
  4. How to know the pass/fail status

SARVAKARMA
Committed to Deliver

Boot ROM responsibilities

- Bootrom performs the essential initialization including programming the clocks, stacks, interrupt set up etc.
- Bootrom will detect the boot media using a system register. This is to determine where to find the software bootloader. A particular sequence of probing for boot media is followed as defined by the manufacturer. This includes the order of looking for bootloader in external NOR/NAND flash, or probing for some specific characters on UART /USB for establishing connection with downloader to download the binary in flash. If no bootloader is found in any external memory, bootrom listens for a download request on UART/USB port to start the download process.
- Thus during the probing process, if the flash has already been programmed, software bootloader will be detected to be available in flash , if not so, it will be downloaded to the flash by bootrom.
- For platforms using NAND flash , the bootrom will load this boot loader to internal RAM and set the program counter at the load address of the SW bootloader in RAM.
- For platforms using NOR flash, control is transferred to the external flash

ARM boot

- Currently, there are two exception models in the ARM architecture (reset is considered a kind of exception):
- The classic model, used in pre-Cortex chip and current Cortex-A/R chips. In it, the memory at 0 contains several exception handlers:
- Offset Handler

  00 Reset
  04 Undefined Instruction
  08 Supervisor Call (SVC)
  0C Prefetch Abort
  10 Data Abort
  14 (Reserved)
  18 Interrupt (IRQ)
  1C Fast Interrupt (FIQ)
- When the exception happens, the processor just starts execution from a specific offset, so usually this table contains single-instruction branches to the complete handlers further in the code. A typical classic vector table looks like following:

```
00000000  LDR  PC, =Reset
00000004  LDR  PC, =Undef
00000008  LDR  PC, =SVC
0000000C  LDR  PC, =PrefAbort
00000010  LDR  PC, =DataAbort
00000014  NOP
00000018  LDR  PC, =IRQ
0000001C  LDR  PC, =FIQ
```

ARM boot

- At runtime, the vector table can be relocated to 0xFFFF0000, which is often implemented as a tightly-coupled memory range for the fastest exception handling. However, the power-on reset usually begins at 0x00000000 (but in some chips can be set to 0xFFFF0000 by a processor pin).
- The new microcontroller model is used in the Cortex-M line of chips. There, the vector table at 0 is actually a table of vectors (pointers), not instructions. The first entry contains the start-up value for the SP register, the second is the reset vector. This allows writing the reset handler directly in C, since the processor sets up the stack. Again, the table can be relocated at runtime. The typical vector table for Cortex-M begins like this:

```
__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
                DCD     NMI_Handler               ; NMI Handler
                DCD     HardFault_Handler         ; Hard Fault Handler
                DCD     MemManage_Handler         ; MPU Fault Handler
                DCD     BusFault_Handler          ; Bus Fault Handler
                DCD     UsageFault_Handler        ; Usage Fault Handler
                [...more vectors...]
```

reset

- On startup, the processor will jump to fixed location ,(most ARM cores support two vector locations 0x0 or 0xFFFF0000, controlled via a signal sampled at reset and a bit in CP15.
- Lets say that the core is configured to have its vectors at 0x0). This address should contain the reset vector and the default vector table.
- Reset vector is always the first instruction to be executed.
- The reset vector in this table will contain a branch to an address which will contain the reset code.
- Normally, at this stage, the rest of the vector table contains just the dummy handler- a branch instruction that causes an infinite loop (this is because this vector table is used very briefly and later on replaced by vector table in RAM after memory remap operation) .
- This reset code to which the jump has been executed from the reset vector will do the following tasks:
  1. Set up system registers and memory environment
  2. Set up MMU
  3. Setup stack pointers : initialize stack pointers for all ARM modes
  4. Set up bss section : zeroing the ZI data region, copying initialization values for initialized variables from ROM to RAM
  5. Set up hw : CPU clock initialization , external bus interface configuration,low level peripheral initialization etc

SARVAKARMA
Committed to Deliver

Memory remap

- One of the job of the initial reset code will be memory remapping.
- At the time of power up, the processor jumps at fixed location 0x0.
- This is important to ensure there is some executable code present at this location at the time of power up. And to ensure this, some non volatile memory should be mapped to this address.
- At the time of power up, ROM is mapped to 0×0 location which contains the reset exception and default vector table. However, later on, during the s/w execution, on every interrupt, the processor needs to jump to the vector table which starts from 0×0 location.
- If this vector table is located in ROM, the execution of interrupts will become very slow as ROM is slower than RAM (ROM requires more wait states as well as more power consumption for access as compared to RAM). Also, if the vector table is present in ROM, it cannot be modified by code.
- Therefore for faster and more efficient execution of interrupts , it is better if interrupt handlers and vector table is located in RAM at address 0×0.
- However, if RAM was mapped to address 0×0 at the power on of processor, being a volatile memory , it won't contain any executable code.
- Thus, it becomes important that at the time of start up, ROM is located at 0×0 address and then during normal execution RAM is re-mapped to this location.
- Memory remapping can be achieved through hardware remapping, that is changing the address map of the bus. This can also be achieved through MMU.

Test initiation

- There are two ways to execute a test of interest in C.
  1. Making it part of **main()**
  2. Using a test case number and writing this number as part of bootcode
- Making it part of main is simple.
  - It is a decision that needs to be taken before compilation and is which test is to be executed will be picked and compiled as part of ARM compilation and the ARM processor will jump to main and execute the test case.
  - So when a test is run with an argument “test1” as testname then the code is to be compiled with test1.c which has code under main(). The image so generated will have test1.c contents in main() and will execute after boot.
  - The advantage of this mechanism is that the setup environment is simple.
  - One of the disadvantage is if there are many tests and regression is to be executed then for each test the code is to be compiled and then test is to be run.
- Second method is to assign test numbers to individual test

  Main()

  Case (test_num)

  Test_num1 :

  Test_num2 :
  - Now the question is how does the processor know which testcase to execute
  - This can be done in multiple ways
    - Using a backdoor write to a pre-decided memory location which the processor reads and then proceeds to execute .
    - Using the boot code
    - External master which can be either a TB BFM or actual host processor can as a part of initialization go and write the test number information in pre-decide memory location.

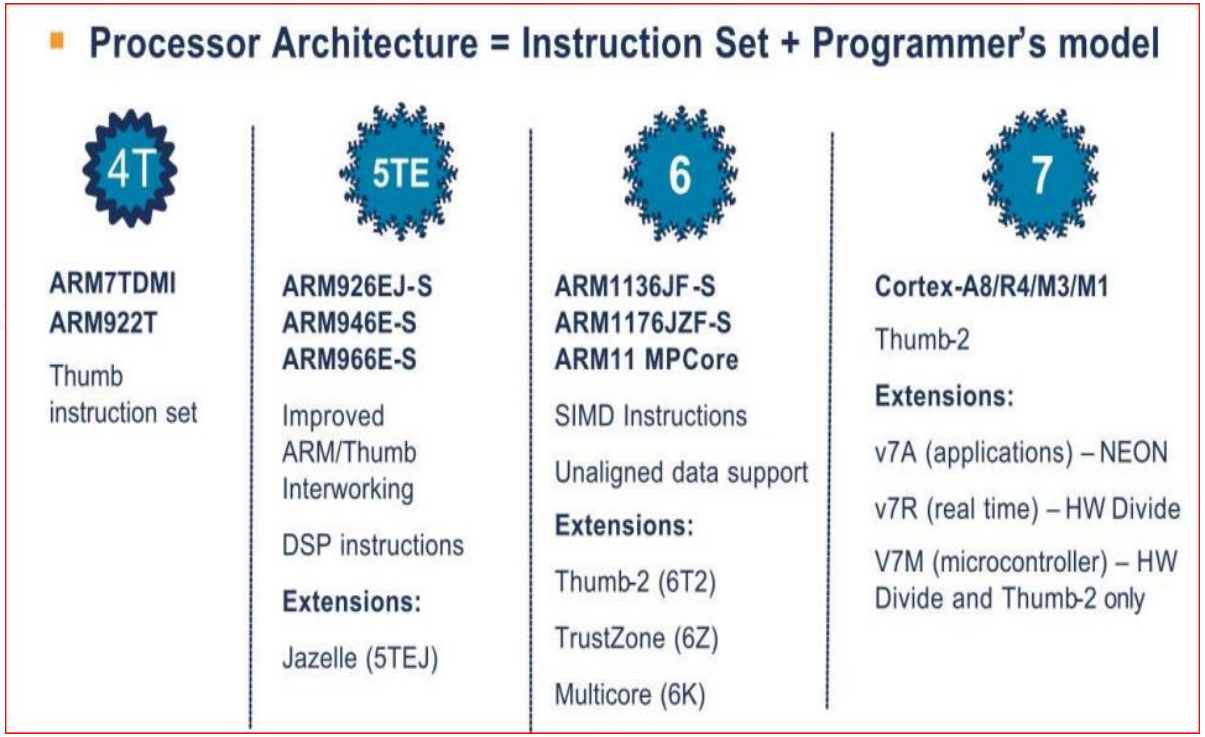
Test pass/fail criteria

- PASS/FAIL criteria can be decided by processor writing to a unused address space with pre-decide data for PASS and FAIL as shown below
  - Pass -> write 0xFACEFACE to pre-decided address location
  - Fail -> write 0xDEADBEEF to pre-decided address location
- This address space can be snooped by a monitor which can display pass or fail and call $***finish*** to end the simulation. It can also exit the simulation with a timeout value if the test hangs.

ARM

- ARM - Advanced RISC Machines (1990) Small size,Low cost,Low power consumption,High performance

Architecture	Family
ARMv1	ARM1
ARMv2	ARM2,ARM3
ARMv3	ARM6,ARM7
ARMv4	StrongARM,ARM7TDMI,ARM9TDMI
ARMv5	ARM7EJ,ARM9E,ARM10E,XScale
ARMv6	ARM11,ARM Cortex-M
ARMv7	ARM Cortex-A,ARM Cortex-M,ARM Cortex-R



ARM Cortex M3

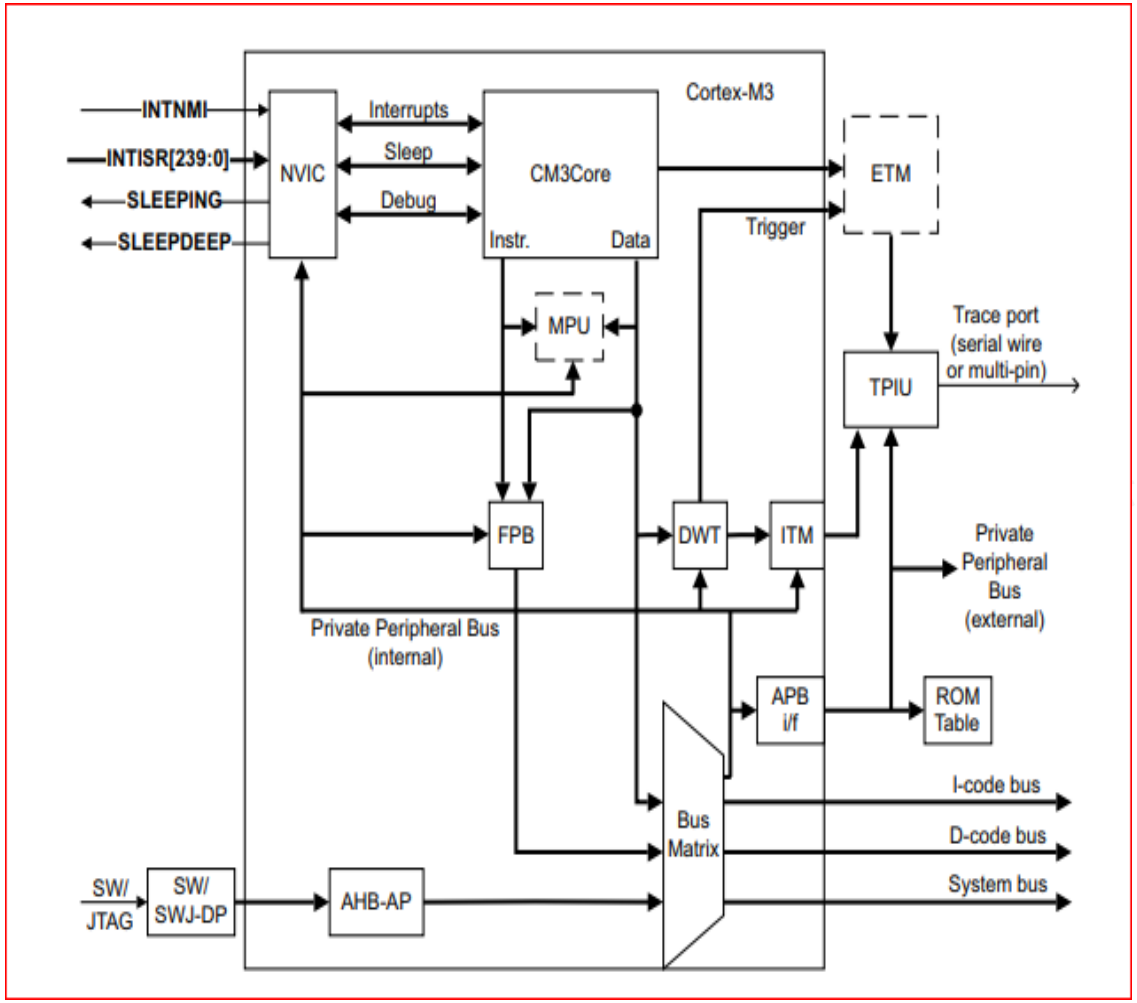
- **A Profile (ARMv7-A)**: Application processors required to run complex applications . High-end embedded operating systems.(Symbian, Linux, and Windows Embedded) Highest processing power, virtual memory system support with Memory Management Units (MMUs). High-end mobile phones and electronic wallets for financial transactions.
- **R Profile (ARMv7-R)**: Real-time. High-end breaking systems and hard drive controllers. High processing power and high reliability are essential and for which low latency is important.
- **M Profile (ARMv7-M)**: Microcontroller targeting low-cost applications. Processing efficiency is important and cost, low power consumption, low interrupt latency, and ease of use are critical. Industrial control applications, including real-time control systems.

- The Cortex processor families are the first products developed on architecture v7.
- The Cortex-M3 processor is based on one profile of the v7 architecture, called ARM v7-M, an architecture specification for microcontroller products.

ARM Cortex M3

Features	ARM7TDMI-S	Cortex-M3
Architecture	ARMv4T (von Neumann)	ARMv7-M (Harvard)
ISA Support	Thumb / ARM	Thumb / Thumb-2
Pipeline	3-Stage	3-Stage + branch speculation
Interrupts	FIQ / IRQ	NMI + 1 to 240 Physical Interrupts
Interrupt Latency	24-42 Cycles	12 Cycles
Sleep Modes	None	Integrated
Memory Protection	None	8 region Memory Protection Unit
Dhrystone	0.95 DMIPS/MHz (ARM mode)	1.25 DMIPS/MHz
Power Consumption	0.28mW/MHz	0.19mW/MHz
Area	0.62mm2 (Core Only)	0.86mm2 (Core & Peripherals)*

ARM cortex M3 processor

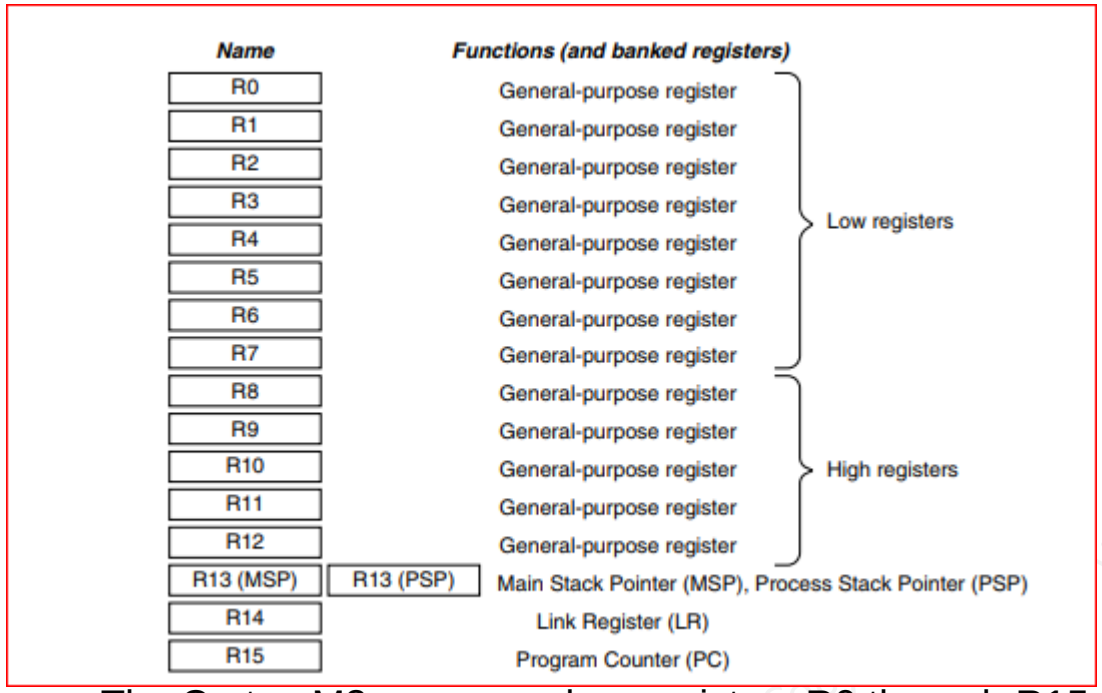


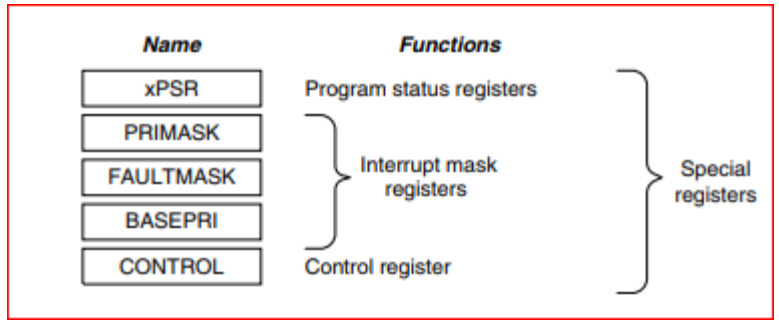
- The CortexM3 is a 32-bit microprocessor. It has a 32-bit data path, a 32-bit register bank, and 32-bit memory interfaces
- The processor has a Harvard architecture, which means that it has a separate instruction bus and data bus. This allows instructions and data accesses to take place at the same time, and as a result of this, the performance of the processor increases because data accesses do not affect the instruction pipeline.
- For complex applications that require more memory system features, the Cortex-M3 processor has an optional Memory Protection Unit (MPU), and it is possible to use an external cache if it's required. Both little endian and big endian memory systems are supported.
- The Cortex-M3 processor includes a number of fixed internal debugging components. These components provide debugging operation supports and features, such as breakpoints and watchpoints.

Cortex M3 features

- Processor core. A low gate count core, with low latency interrupt processing that features:
  - ARMv7-M.
  - A Thumb-2 Instruction Set Architecture (ISA) subset,
- Nested Vectored Interrupt Controller (NVIC) closely integrated with the processor core to achieve low latency interrupt processing. Features include:
  - External interrupts of 1 to 240 configurable size.
  - Support for tail-chaining and late arrival of interrupts. This enables back-to-back interrupt processing without the overhead of state saving and restoration between interrupts.
  - Processor state automatically saved on interrupt entry, and restored on interrupt exit, with no instruction overhead consisting of all base Thumb-2 instructions.
- MPU - For complex applications that require more memory system features, the Cortex-M3 processor has an optional Memory Protection Unit (MPU), and it is possible to use an external cache if it's required.
- Bus interfaces:
  - Advanced High-performance Bus-Lite (AHB-Lite) ICode, DCode and System bus interfaces.
  - Advanced Peripheral Bus (APB) and Private Peripheral Bus (PPB) Interface.
  - Bit band support that includes atomic bit band write and read operations.
  - Write buffer for buffering of write data.
- Low-cost debug solution that features:
  - Debug access to all memory and registers in the system, including Cortex-M3 register bank when the core is running, halted, or held in reset.
  - Serial Wire Debug Port (SW-DP) or Serial Wire JTAG Debug Port (SWJ-DP) debug access, or both. — Instrumentation Trace Macrocell (ITM) for support of printf style debugging.
  - Trace Port Interface Unit (TPIU) for bridging to a Trace Port Analyzer (TPA).
  - Optional Embedded Trace Macrocell (ETM) for instruction trace.

Registers





Register	Function
xPSR	Provide arithmetic and logic processing flags (zero flag and carry flag), execution status, and current executing interrupt number
PRIMASK	Disable all interrupts except the nonmaskable interrupt (NMI) and hard fault
FAULTMASK	Disable all interrupts except the NMI
BASEPRI	Disable all interrupts of specific priority level or lower priority level
CONTROL	Define privileged status and stack pointer selection

- The Cortex-M3 processor has registers R0 through R15
- R13 (the stack pointer) is banked, with only one copy of the R13 visible at a time.
- R0–R12 are 32-bit general-purpose registers for data operations.
- Some 16-bit Thumb® instructions can only access a subset of these registers (low registers, R0–R7).
- Main Stack Pointer (MSP): The default stack pointer, used by the operating system (OS) kernel and exception handlers
- Process Stack Pointer (PSP): Used by user application code
- Link register - When a subroutine is called, the return address is stored in the link register.
- Program Counter - The program counter is the current program address. This register can be written to control the program flow.

Operating modes

- The Cortex-M3 processor has two modes and two privilege levels
  - The operation modes (thread mode and handler mode) determine whether the processor is running a normal program or running an exception handler like an interrupt handler or system exception handler
  - The privilege levels (privileged level and user level) provide a mechanism for safeguarding memory accesses to critical regions as well as providing a basic security model.
- When the processor is running a main program (thread mode), it can be either in a privileged state or a user state, but exception handlers can only be in a privileged state.
- When the processor exits reset, it is in thread mode, with privileged access rights.
- In the privileged state, a program has access to all memory ranges (except when prohibited by MPU settings) and can use all supported instructions. Software in the privileged access level can switch the program into the user access level using the control register.
- When an exception takes place, the processor will always switch back to the privileged state and return to the previous state when exiting the exception handler
- The separation of privilege and user levels improves system reliability by preventing system configuration registers from being accessed or changed by some untrusted programs

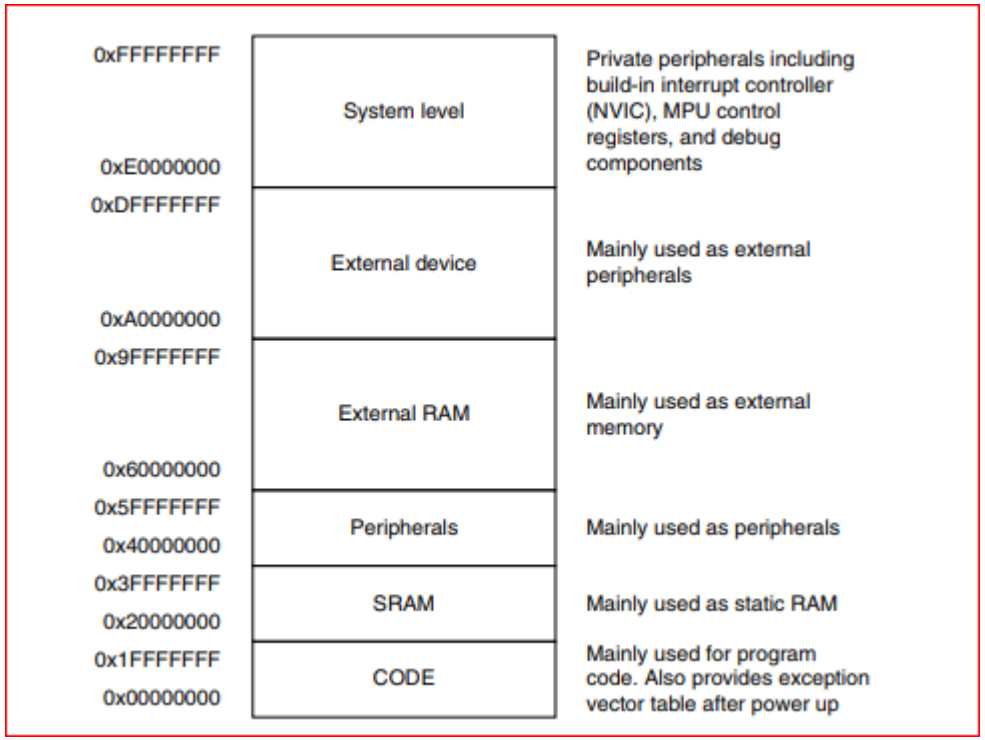
	Privileged	*User*
When running an exception handler	Handler mode	
When not running an exception handler (e.g., main program)	Thread mode	Thread mode

Non Vectored Interrupt Controller (NVIC)

- **Nested Interrupt Support** - All the external interrupts and most of the system exceptions can be programmed to different priority levels. When an interrupt occurs, the NVIC compares the priority of this interrupt to the current running priority level. If the priority of the new interrupt is higher than the current level, the interrupt handler of the new interrupt will override the current running task.
- **Vectored Interrupt Support-** When an interrupt is accepted, the starting address of the interrupt service routine (ISR) is located from a vector table in memory. There is no need to use software to determine and branch to the starting address of the ISR. Thus, it takes less time to process the interrupt request.
- **Dynamic Priority Changes** Support Priority levels of interrupts can be changed by software during run time. Interrupts that are being serviced are blocked from further activation until the ISR is completed, so their priority can be changed without risk of accidental re entry.
- **Reduction of Interrupt Latency** These include automatic saving and restoring some register contents, reducing delay in switching from one ISR to another, and handling of late arrival interrupts.
- **Interrupt Masking** Interrupts and system exceptions can be masked based on their priority level or masked completely using the interrupt masking registers BASEPRI, PRIMASK, and FAULTMASK. They can be used to ensure that time-critical tasks can be finished on time without being interrupted.

Memory Map

- The Cortex-M3 has an optional MPU.
- This unit allows access rules to be set up for privileged access and user program access.
- When an access rule is violated, a fault exception is generated, and the fault exception handler will be able to analyse the problem and correct it, if possible.
- The MPU can be used in various ways. In common scenarios, the OS can set up the MPU to protect data use by the OS kernel and other privileged processes to be protected from untrusted user programs.
- The MPU can also be used to make memory regions read-only, to prevent accidental erasing of data or to isolate memory regions between different tasks in a multitasking system.
- Overall, it can help make embedded systems more robust and reliable.



Instruction set & Debug support

Instruction Set

- The Cortex-M3 supports the Thumb-2 instruction set.
- This is one of the most important features of the Cortex-M3 processor because it allows 32-bit instructions and 16-bit instructions to be used together for high code density and high efficiency. It is flexible and powerful yet easy to use.

Debug:

- Supports JTAG or Serial-Wire debug interfaces.
- Based on the CoreSight debugging solution, processor status or memory contents can be accessed even when the core is running.
- Built-in support for six breakpoints and four watch points
- Optional ETM for instruction trace and data trace using DWT
  - New debugging features, including fault status registers, new fault exceptions, and Flash Patch operations, make debugging much easier
  - ITM provides an easy-to-use method to output debug information from test code
  - PC sampler and counters inside the DWT provide code-profiling information

Tightly Coupled Memories (TCM)

- Embedded applications often have requirements for fast, deterministic memory to store real-time data and performance critical instruction sequences. Some ARM cores provide tightly-coupled memory (TCM) to satisfy this requirement.
- The ARM core provides an interface to TCM, but the memory itself is implemented outside of the CPU core. When TCM functionality is provided for Harvard architecture cores, there are separate memory interfaces for instruction (ITCM) and data (DTCM).
- By locating the memory outside of the core, designers have the greatest flexibility in system design and can work with differences in RAM libraries for specific semiconductor processes.
- TCM is different from cache memory since it can be addressed directly by software at a specific location in the microprocessor memory map. TCM is usually implemented as single-cycle SRAM, and TCM size is specified using input signals to the CPU core.
- TCM status and control is done via programming registers in coprocessor 15.
- Software can enable and disable TCM as well as assign the location in the memory map. As an example, information about the TCM interface for the ARM926EJ-S is presented below.
- Data TCM is always disabled at reset and must be explicitly enabled by software.
- Instruction TCM is disabled at reset unless the **INITRAM** signal is high. With **INITRAM** high, ITCM starts enabled and will respond to memory requests at address 0. This option allows the CPU to boot directly from TCM.
- Of course, booting from TCM implies the design can load instructions into the ITCM before reset and without the use of software.
- If ITCM is to be enabled at reset, but not used for the reset vectors, the ARM926EJ-S offers the **VINITHI** signal to tell the CPU to boot from address 0xffff0000 instead.
- When **VINITHI** is high, the CPU will locate the exception vectors at 0xffff0000.Because the TCM interface is optimized for single-cycle performance, there is no protection against software reading from this memory.
- The MMU can be used with TCM to protect against unauthorized access, but only aborted writes are guaranteed not to take place. Aborted reads will still read memory, so there is no way to protect against TCM reads.

Thank You