ARM (32-bit) Architecture

Table of Contents

Extracts and information paraphrased mostly from the well known "ARM System Developer's Guide – Designing and Optimizing System Software" by Sloss, Symes and Wright, published by Elsevier, and
The ARM University Program, ARM Architecture Fundamentals, YouTube.

Additional Resources
ARM Architecture Reference Manual
Advanced Compilers and Architectures – ARM MMU Overview, Mattias Holm
Embedded Programming with the GNU Toolchain, a git book

ARM:
- very early- dates back to 1985.
ARM cores – over 30 *billion* have shipped, growing at ~ 8 billion a year!
Predicted to exceed a 100 billion by 2020.

Cortex-**A** : **A**pplication Processors; 'rich' platform OS
Cortex-**R** : **R**eal-time
Cortex-**M** : **M**icrocontroller-based

Source

ARM's new processor naming convention

With the announcement of the Cortex™-M3 processor, ARM introduced a new naming scheme for its processors. Previously processors were named for the base processor core such as ARM7™, ARM9™ and ARM11™, and digits were added to designate the features of that variation of the processor. Thus an ARM926EJ-S™ was based on the ARM9 microarchitecture, and digits were added to designate the memory system, Java support and arithmetic extensions.

The new ARM naming scheme is based on the generation of architecture of the processor, and not sequence of arrival of the processor itself. (At ARM an architecture refers to an instruction/feature set common to a group of processors.)

Cores which belong to the latest generation of ARM architecture, ARMv7, will all be know as "Cortex" processors. The next generation of processors will be given yet another new name. This was done to give a clearer understanding of the relative performance and target application of a new processor from its name.

The ARMv7 architecture and thus the Cortex processors will be divided into three main groups: Applications (A) processors, Realtime (R) processors and Microcontrollors (M).

Applications processors are intended for use with open OS and feature a memory management unit

(MMU) providing for virtual addressing.

Realtime processors will focus more deeply embedded applications. They will feature a memory protection unit (MPU) which protects regions of memory but does not provide for virtual addressing.

Microcontrollers will generally not have memory protection, and focus on providing very low latency responses to interrupts and including features such as flash memory controllers and interrupt controllers.

The group to which a processor belongs is designated by adding a -A, -R or -M to be base Cortex name.

[OLDER]
<< For eg., what is ARM9TDMI? >>

ARM Naming Convention

All ARM processors share a common naming convention that has evolved over time. ARM cores have the name **ARM{x}{labels}** where x is the number of the core and labels are letters representing extra features, described in Table C.1.

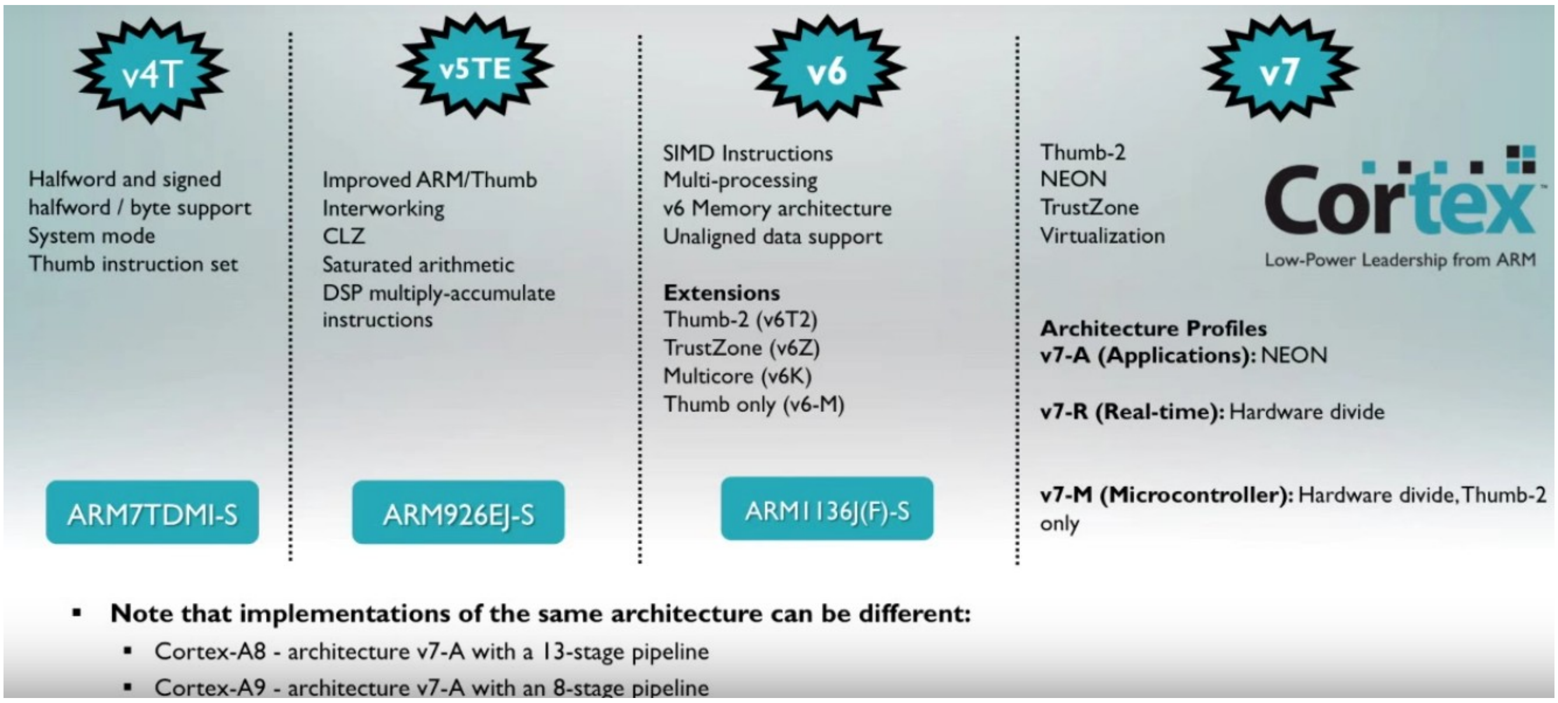
ARM processors have the name

ARM{x}{y}{z}{labels} ; where

y and **z** are numbers defining the processor cache size and memory management model.

Eg. ARM7TDMI, ARM9TDMI, etc.
See *Appendix A – Older ARM Naming Convention* for more details and examples.

[Source]



v4T
Halfword and signed halfword / byte support
System mode
Thumb instruction set
ARM7TDMI-S

v5TE
Improved ARM/Thumb Interworking
CLZ
Saturated arithmetic
DSP multiply-accumulate instructions
ARM926EJ-S

v6
SIMD Instructions
Multi-processing
v6 Memory architecture
Unaligned data support

Extensions
Thumb-2 (v6T2)
TrustZone (v6Z)
Multicore (v6K)
Thumb only (v6-M)
ARM1136J(F)-S

v7
Thumb-2
NEON
TrustZone
Virtualization

Cortex
Low-Power Leadership from ARM

Architecture Profiles
v7-A (Applications): NEON

v7-R (Real-time): Hardware divide

v7-M (Microcontroller): Hardware divide, Thumb-2 only

- **Note that implementations of the same architecture can be different:**
  - Cortex-A8 - architecture v7-A with a 13-stage pipeline
  - Cortex-A9 - architecture v7-A with an 8-stage pipeline

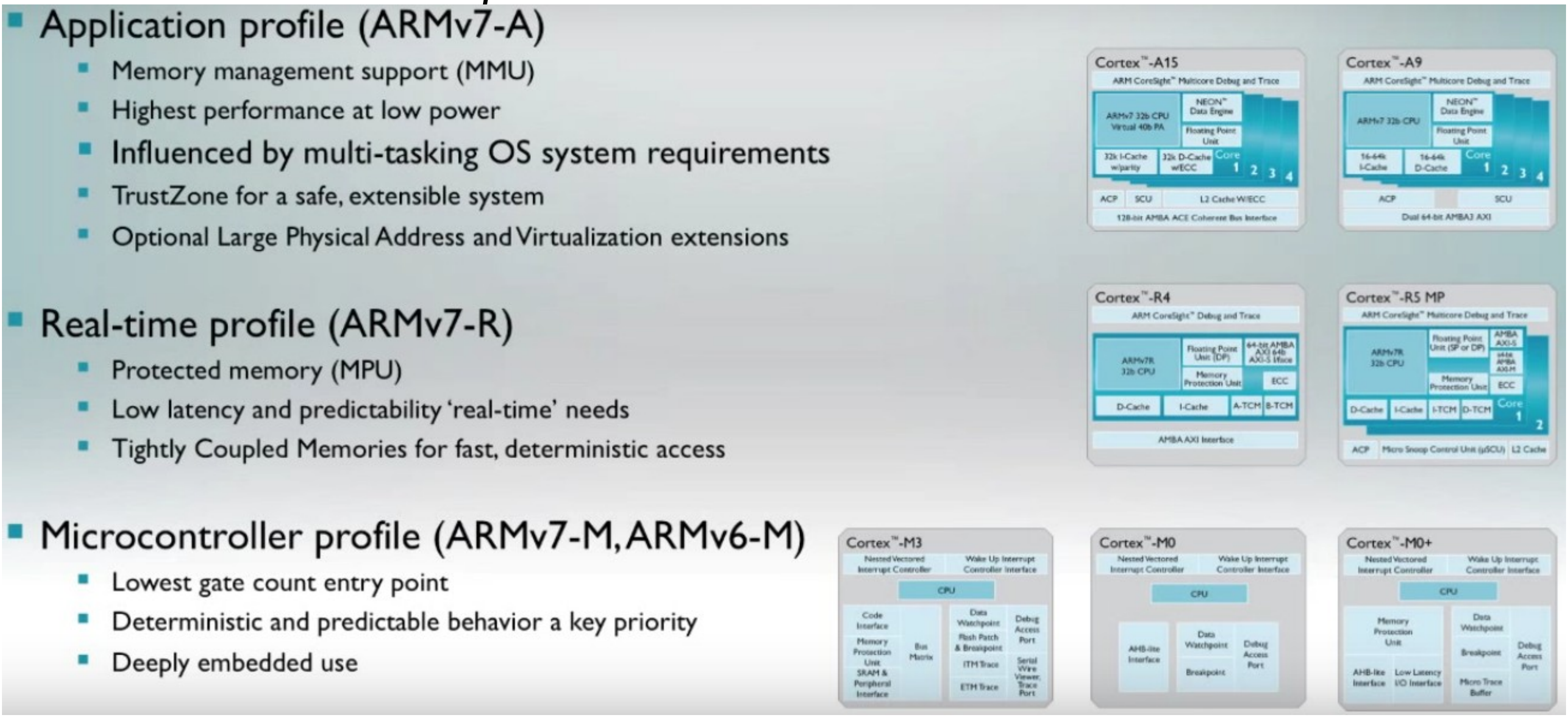
Cortex-A8 and Cortex-A9 support the same v7 architecture and are thus binary compatible; however, their internal implementation (microarchitecture) differs significantly.

ARM v7 Architecture Profiles

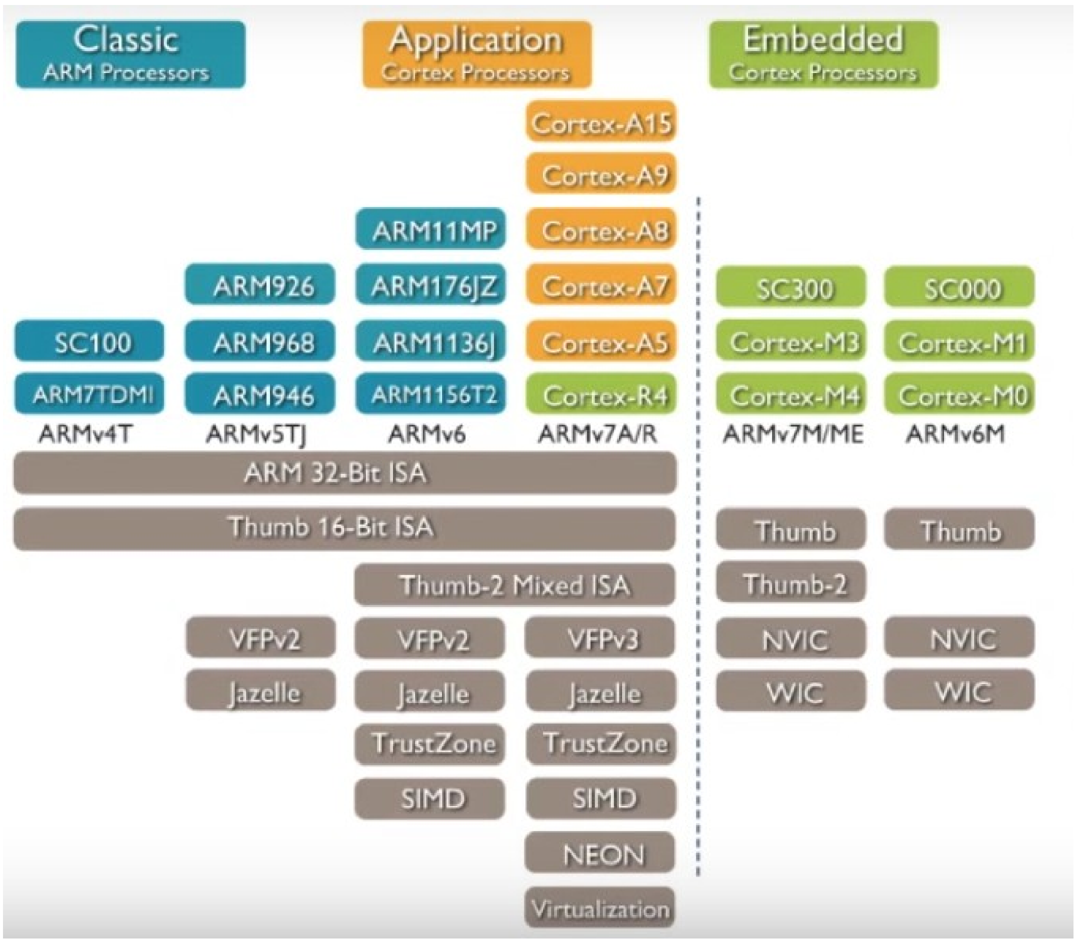
Starting from ARMv7, different ***Architecture Profiles*** determine the primary function / domain the processor will function in. From top-end to low-end:

- v7-A : Application Profile (NEON) : for platform OS (Linux), MMU, full virtualization [Cortex-A15 and A7], etc
- v7-R : Real-time Profile : hard real-time; MPU, TCM (tightly-coupled) memory; deeply embedded (car ECUs, medical, HDD controllers, etc) [Cortex-R4 and R5 MPcore]
- v7-M : Microcontroller Profile [Cortex-M3, M0 & M0+ (smallest ARM core ever produced)].

ARM Architecture v7 Profiles



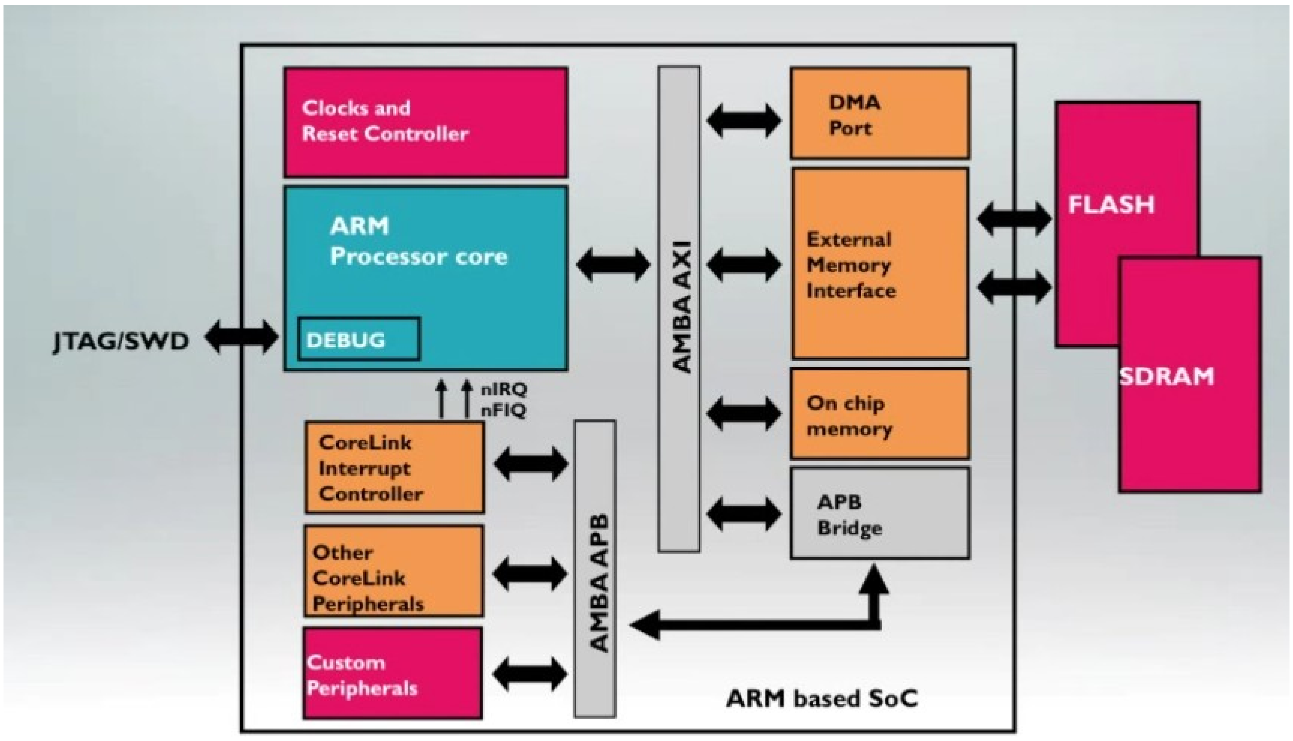
- Application profile (ARMv7-A)
  - Memory management support (MMU)
  - Highest performance at low power
  - Influenced by multi-tasking OS system requirements
  - TrustZone for a safe, extensible system
  - Optional Large Physical Address and Virtualization extensions
- Real-time profile (ARMv7-R)
  - Protected memory (MPU)
  - Low latency and predictability 'real-time' needs
  - Tightly Coupled Memories for fast, deterministic access
- Microcontroller profile (ARMv7-M, ARMv6-M)
  - Lowest gate count entry point
  - Deterministic and predictable behavior a key priority
  - Deeply embedded use


Classic
ARM Processors
Application
Cortex Processors
Embedded
Cortex Processors
Cortex-A15
Cortex-A9
ARM11MP
Cortex-A8
ARM926
ARM176JZ
Cortex-A7
SC300
SC000
SC100
ARM968
ARM1136J
Cortex-A5
Cortex-M3
Cortex-M1
ARM7TDMI
ARM946
ARM1156T2
Cortex-R4
Cortex-M4
Cortex-M0
ARMv4T
ARMv5TJ
ARMv6
ARMv7A/R
ARMv7M/ME
ARMv6M
ARM 32-Bit ISA
Thumb 16-Bit ISA
Thumb
Thumb
Thumb-2 Mixed ISA
Thumb-2
VFPv2
VFPv2
VFPv3
NVIC
NVIC
Jazelle
Jazelle
Jazelle
WIC
WIC
TrustZone
TrustZone
SIMD
SIMD
NEON
Virtualization

Inside an ARM-based system

[Source]

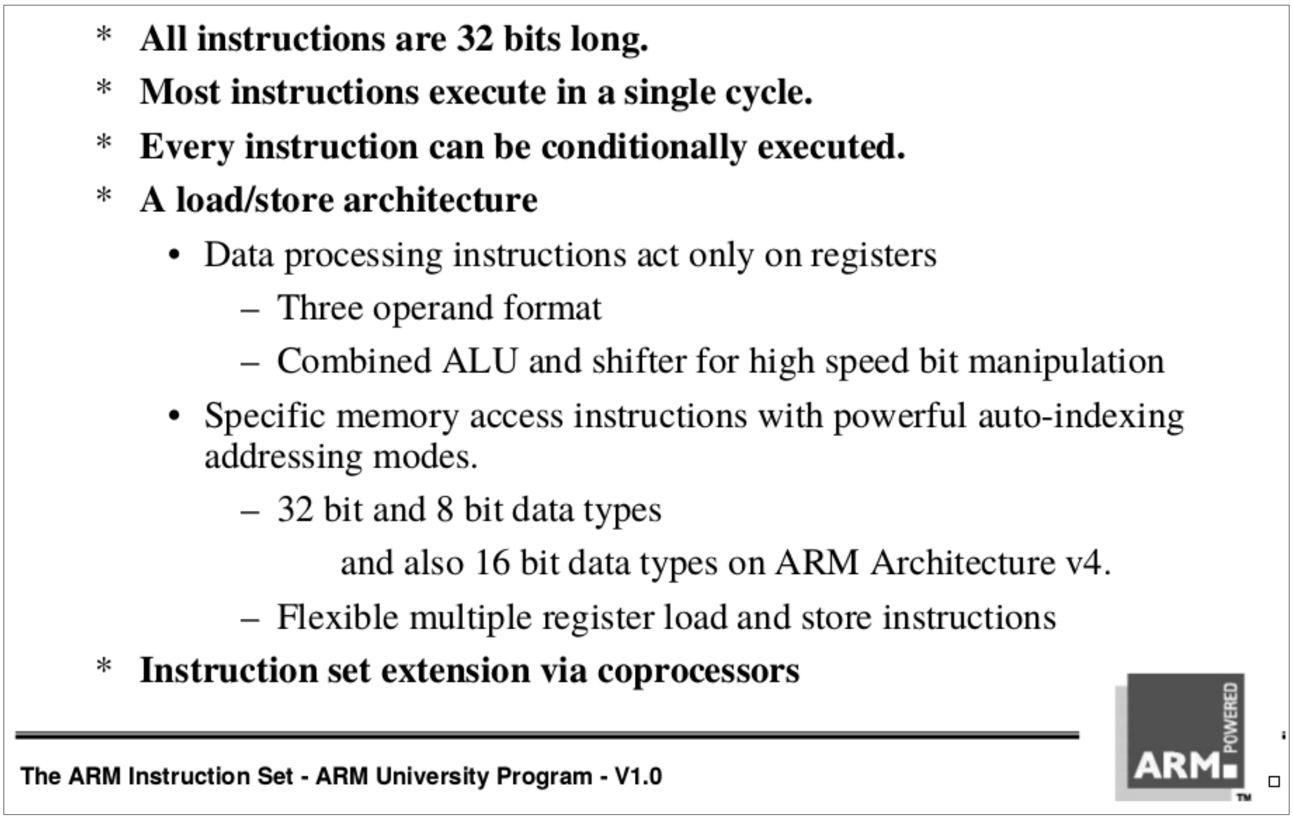
Realistically, ARMv6 and ARMv7 now dominates the market, with ARMv8 (ARM-64) gaining quickly.



Miscellaneous

- Von Neumann architecture: data and instructions share the same bus
- Harvard architecture: data and instructions use separate buses
Both implementations exist on ARM.

ARM Programmers' Model



* **All instructions are 32 bits long.**

* **Most instructions execute in a single cycle.**

* **Every instruction can be conditionally executed.**

* **A load/store architecture**

- Data processing instructions act only on registers
  - Three operand format
  - Combined ALU and shifter for high speed bit manipulation
- Specific memory access instructions with powerful auto-indexing addressing modes.
  - 32 bit and 8 bit data types

    and also 16 bit data types on ARM Architecture v4.
  - Flexible multiple register load and store instructions

* **Instruction set extension via coprocessors**

The ARM Instruction Set - ARM University Program - V1.0

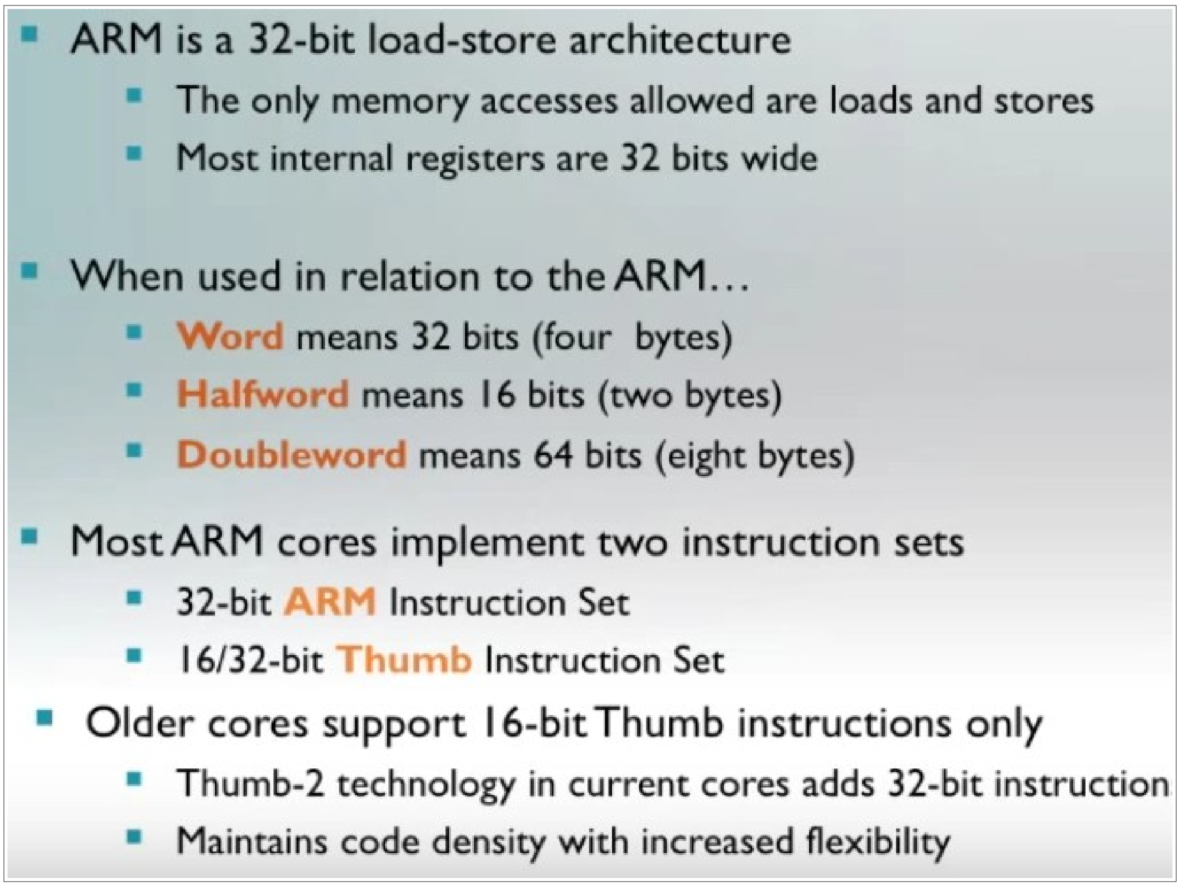
- ARM is a RISC architecture
  - Most instructions execute in a single cycle
  - orthogonal (simple and straight-forward) register set; registers r0 – r13 are treated the same by all instructions (of course, some registers – r13, r14, r15 – are special-purpose)
  - load-store architecture [cannot do memory-to-memory operations; cannot directly manipulate contents of memory (RAM). Memory only accessed via load and store machine instructions, which move contents of RAM to/from registers ↔ memory].
    (Eg.
    ```
    LDR r0, [r3]        @  =>  r0 = *r3  i.e. load contents of memory addressed
                        @ by R3 into R0.
    ```
    Text following the '@' is a comment)[1].

Resource: ARM Programmer's Model

Thumb
- originally, 16-bit instruction set
- Thumb-2 mode: mixes 16 and 32-bit instructions

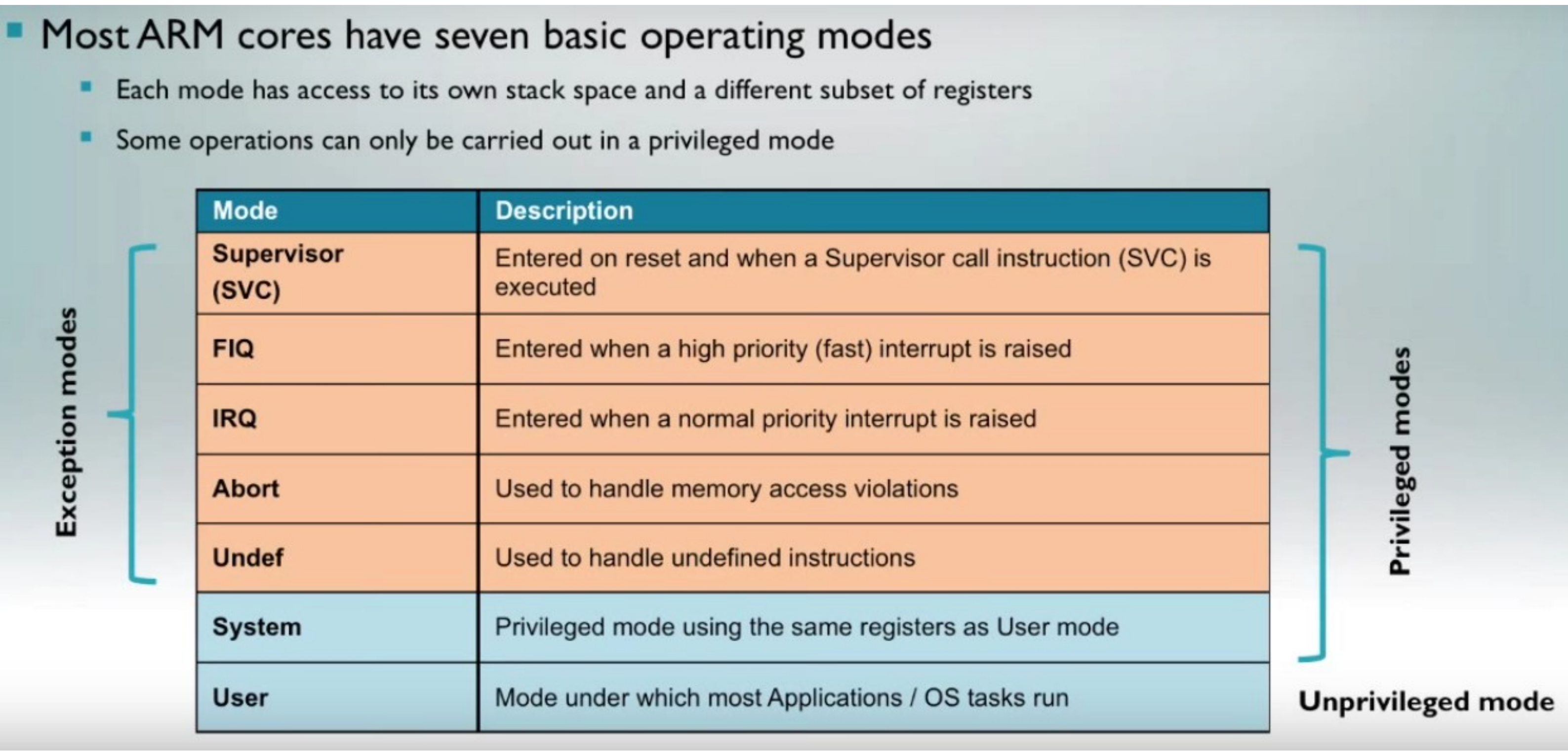
1 FYI, "Hello, world" in ARM Assembly language

- mixing “normal” ARM and Thumb[-2] instructions is not only possible, it’s encouraged; this is called *interworking*.



- ARM is a 32-bit load-store architecture
  - The only memory accesses allowed are loads and stores
  - Most internal registers are 32 bits wide
- When used in relation to the ARM…
  - **Word** means 32 bits (four bytes)
  - **Halfword** means 16 bits (two bytes)
  - **Doubleword** means 64 bits (eight bytes)
- Most ARM cores implement two instruction sets
  - 32-bit **ARM** Instruction Set
  - 16/32-bit **Thumb** Instruction Set
- Older cores support 16-bit Thumb instructions only
  - Thumb-2 technology in current cores adds 32-bit instruction
  - Maintains code density with increased flexibility

ARM Operating Modes



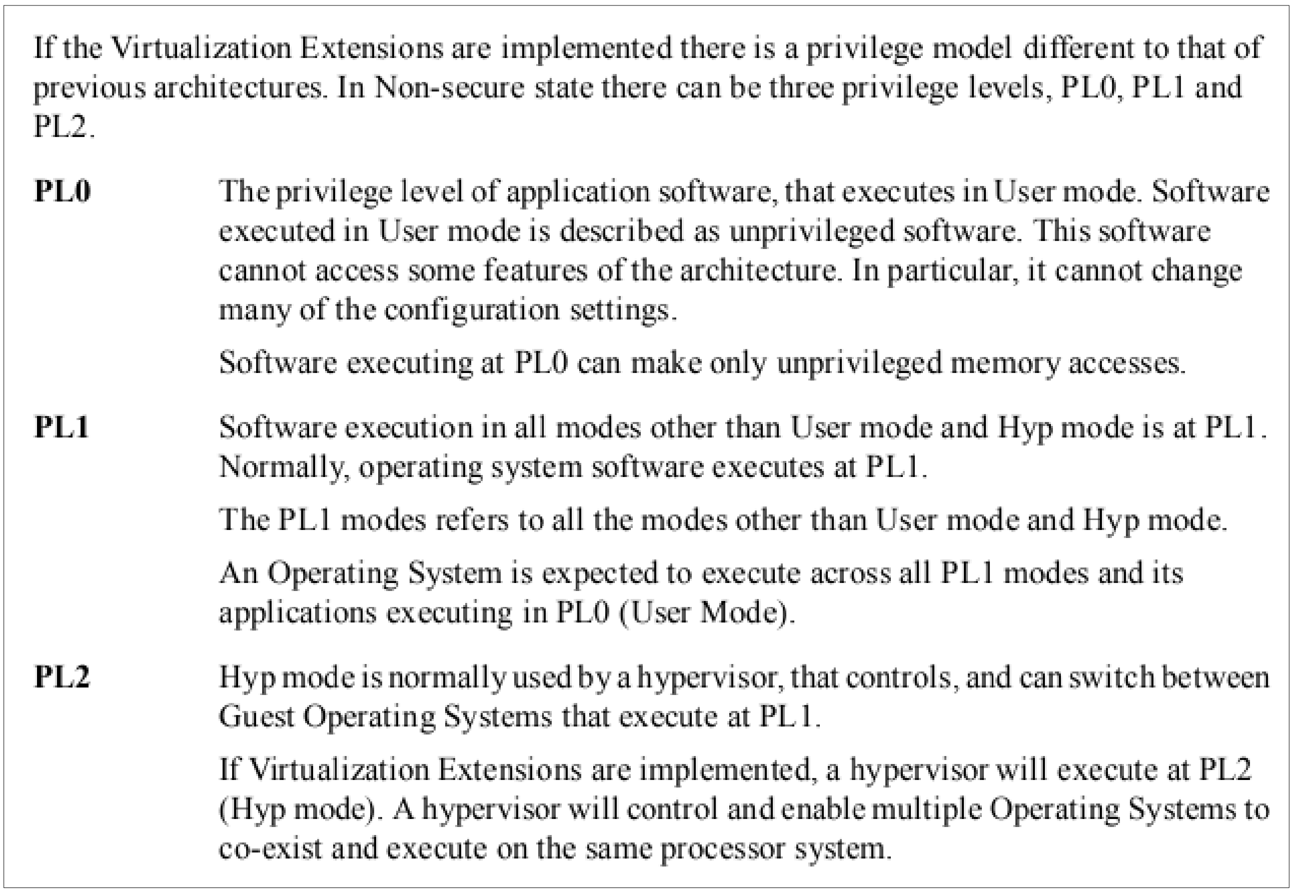
- Most ARM cores have seven basic operating modes
  - Each mode has access to its own stack space and a different subset of registers
  - Some operations can only be carried out in a privileged mode

Mode	Description
Supervisor (SVC)	Entered on reset and when a Supervisor call instruction (SVC) is executed
FIQ	Entered when a high priority (fast) interrupt is raised
IRQ	Entered when a normal priority interrupt is raised
Abort	Used to handle memory access violations
Undef	Used to handle undefined instructions
System	Privileged mode using the same registers as User mode
User	Mode under which most Applications / OS tasks run

- 6 out of 7 ARM cpu modes are privileged
- Out of the 6 privileged modes, 5 of them handle “exceptions” : an instruction or access that causes the flow of control to be diverted (via the cpu vector table)
- applications (processes) execute in unprivileged “user” mode
- each mode has it’s own stack and a few private registers to enable very fast switching in hardware; this is called “register banking”.

For the ARM Cortex-A

PL0 : Userspace | PL1 : Kernel-space | if (Hyp) PL2 : Hypervisor



If the Virtualization Extensions are implemented there is a privilege model different to that of previous architectures. In Non-secure state there can be three privilege levels, PL0, PL1 and PL2.

PL0 The privilege level of application software, that executes in User mode. Software executed in User mode is described as unprivileged software. This software cannot access some features of the architecture. In particular, it cannot change many of the configuration settings.

Software executing at PL0 can make only unprivileged memory accesses.

PL1 Software execution in all modes other than User mode and Hyp mode is at PL1. Normally, operating system software executes at PL1.

The PL1 modes refers to all the modes other than User mode and Hyp mode.

An Operating System is expected to execute across all PL1 modes and its applications executing in PL0 (User Mode).

PL2 Hyp mode is normally used by a hypervisor, that controls, and can switch between Guest Operating Systems that execute at PL1.

If Virtualization Extensions are implemented, a hypervisor will execute at PL2 (Hyp mode). A hypervisor will control and enable multiple Operating Systems to co-exist and execute on the same processor system.

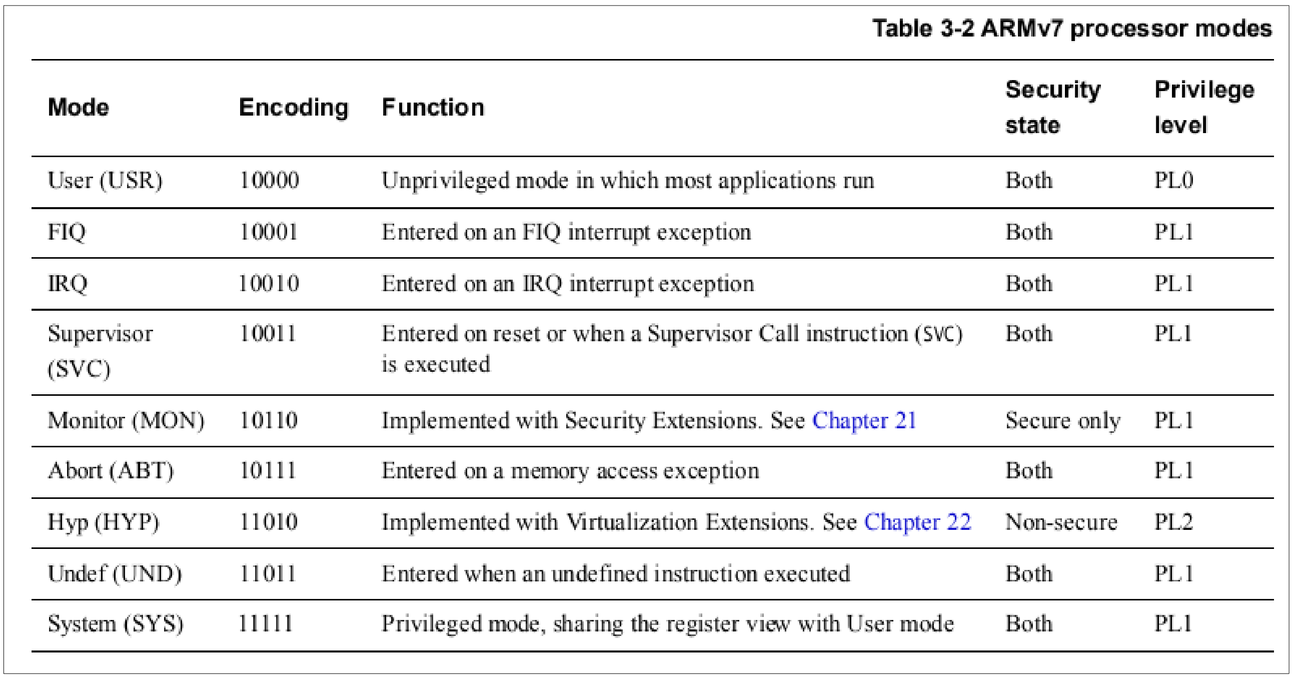


Table 3-2 ARMv7 processor modes

Mode	Encoding	Function	Security state	Privilege level
User (USR)	10000	Unprivileged mode in which most applications run	Both	PL0
FIQ	10001	Entered on an FIQ interrupt exception	Both	PL1
IRQ	10010	Entered on an IRQ interrupt exception	Both	PL1
Supervisor (SVC)	10011	Entered on reset or when a Supervisor Call instruction (SVC) is executed	Both	PL1
Monitor (MON)	10110	Implemented with Security Extensions. See Chapter 21	Secure only	PL1
Abort (ABT)	10111	Entered on a memory access exception	Both	PL1
Hyp (HYP)	11010	Implemented with Virtualization Extensions. See Chapter 22	Non-secure	PL2
Undef (UND)	11011	Entered when an undefined instruction executed	Both	PL1
System (SYS)	11111	Privileged mode, sharing the register view with User mode	Both	PL1

ARM Register Set and Mode Switching

There are up to 18 active registers: 16 data registers and 2 processor status registers. The data registers are visible to the programmer as r0 to r15.

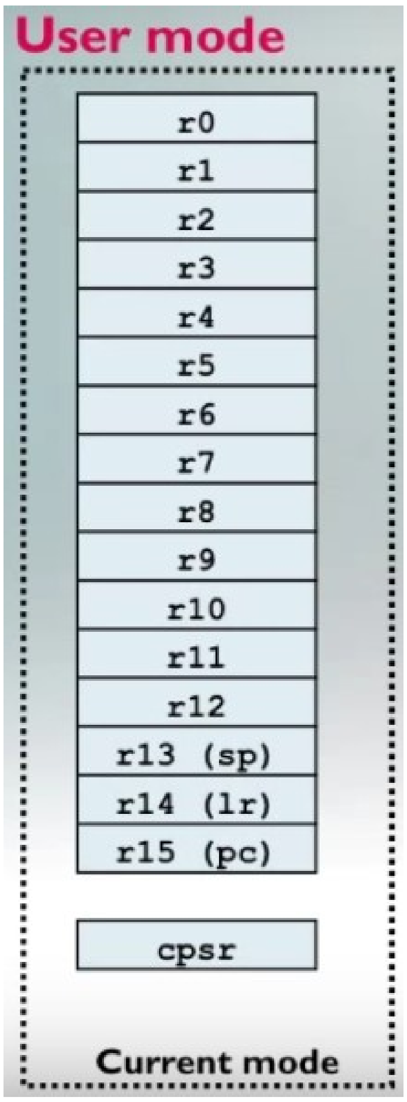


Fig 1-A

The ARM processor has three registers assigned to a particular task or special function: r13, r14, and r15. They are frequently given different labels to differentiate them from the other registers.

- Register r13 is traditionally used as the stack pointer (sp) and stores the head of the stack in the current processor mode.
- Register r14 is called the link register (lr) and is where the core puts the return address whenever it calls a subroutine.
- Register r15 is the program counter (pc) and contains the address of the next instruction to be fetched by the processor.

Depending upon the context, registers r13 and r14 can also be used as general-purpose registers, which can be particularly useful since these registers are banked during a processor mode change.

However, it is dangerous to use r13 as a general register when the processor is running any form of operating system because operating systems often assume that r13 always points to a valid stack frame.

In ARM state the registers r0 to r13 are orthogonal — any instruction that you can apply to r0 you can equally well apply to any of the other registers. However, there are instructions that treat r14 and r15 in a special way.

In addition to the 16 data registers, there are two program status registers: cpsr and spsr (the current and saved program status registers, respectively).

The register file contains all the registers available to a programmer. Which registers are visible to the programmer depend upon the current mode of the processor.

Other “banked” registers are currently not accessible; they come into play when a mode switch occurs.

Lets say a hardware interrupt - an IRQ - occurs:

Before the IRQ, the situation is as shown above- the core is executing in regular ‘user’ mode.
[see Fig 1-A above]

When an IRQ occurs, the core switches to IRQ mode:

- during that switch the hardware now “banks”*, i.e., saves, the r13 (sp), r14 (link) and CPSR (into SPSR);

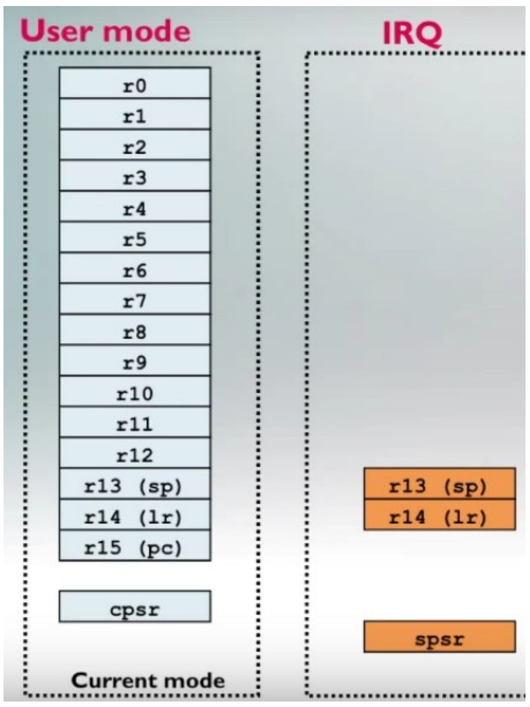
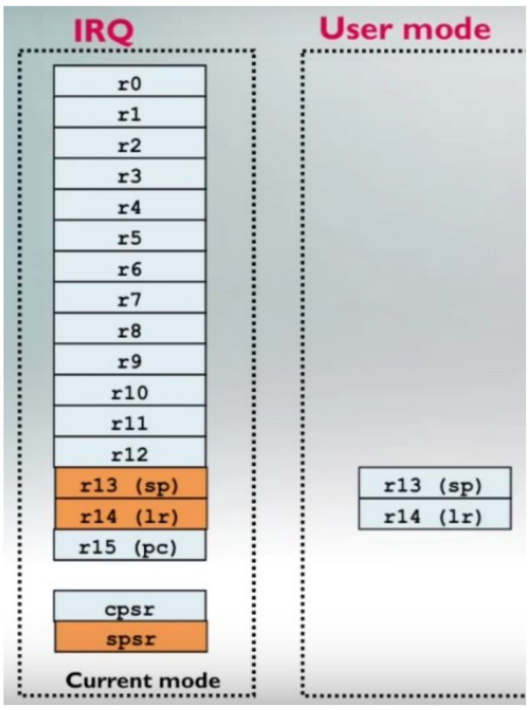
[see Fig 2-A]

- the IRQ now executes, with r13 (sp), r14 (link) and CPSR for the ‘IRQ’ mode *[Fig 2-B]*. Implies the IRQ mode – like all the other modes - have their own private stack, return address and place to store the core’s context at the instant the IRQ (or other mode) switch occurred – the SPSR. This arrangement makes returning to the original mode very easy and fast

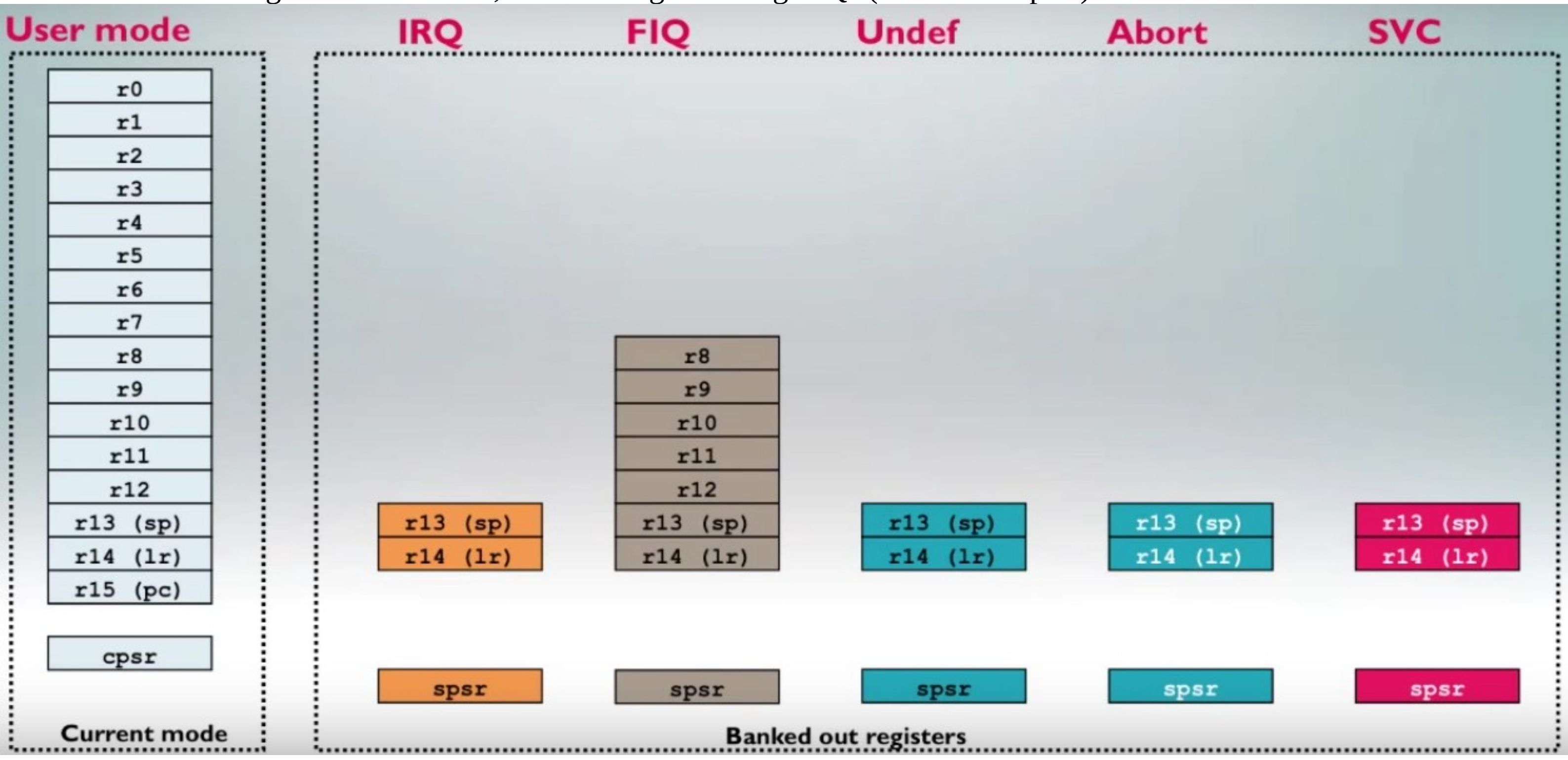
- and once done, we switch back (restore context) to the previous ‘User’ mode by again switching the appropriate registers *[Fig 2-A]*

This is really fast as it’s all done in hardware!

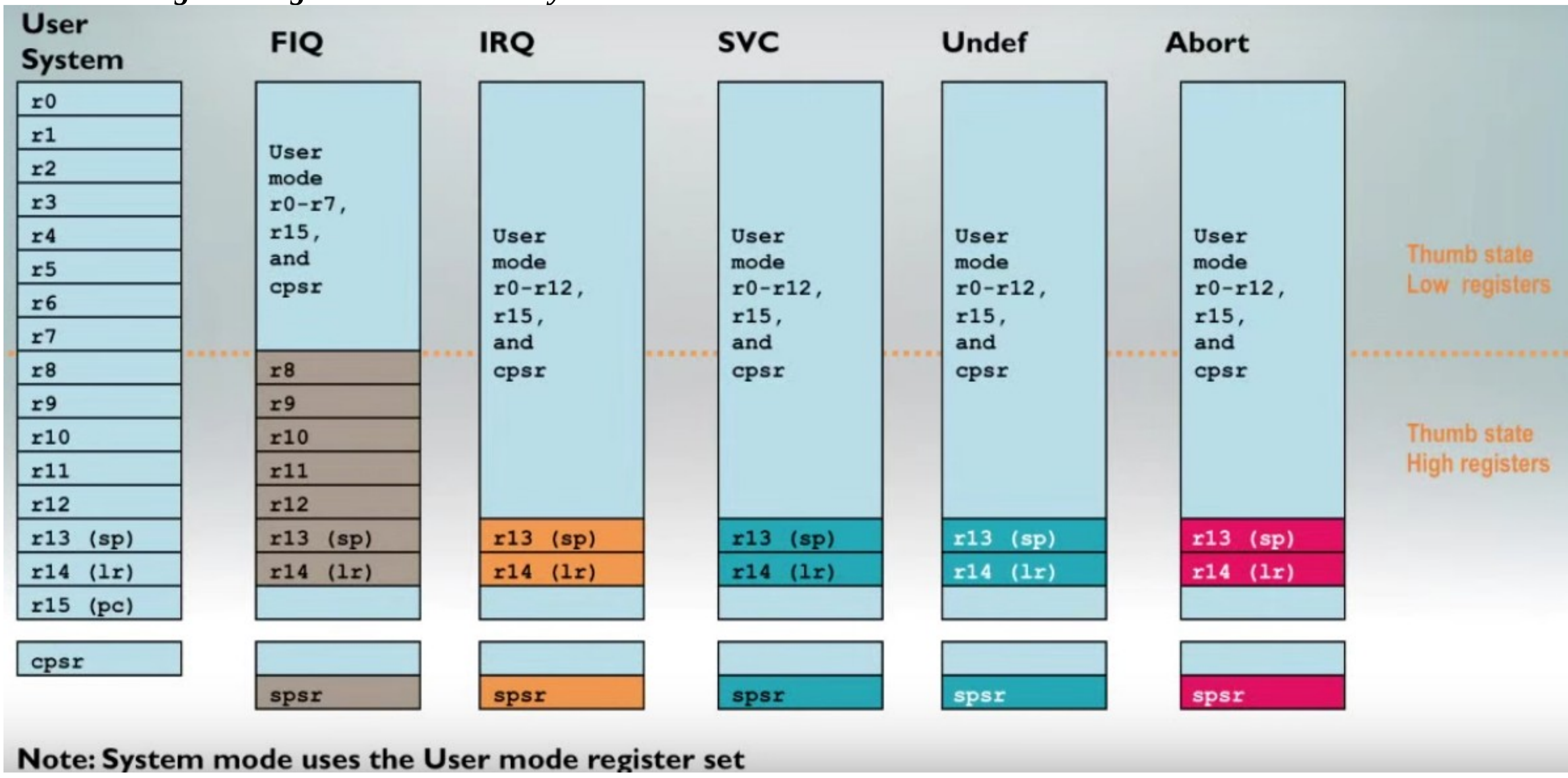
* register banking: a subset of the registers change places. In the example here, the the r13 (sp), r14 (link) and SPSR change – they hold the saved values so that they can be restored once done.

Core in User Mode before IRQ	*Switch* *User ↔ IRQ*	*IRQ occurs, core switches to IRQ Mode “banking” r13, r14 and CPSR of user mode (into SPSR)*
 User mode IRQ r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 (sp) r14 (lr) r15 (pc) cpsr r13 (sp) r14 (lr) spsr Current mode *Fig 2-A*	◄─►	 IRQ User mode r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 (sp) r14 (lr) r15 (pc) cpsr spsr r13 (sp) r14 (lr) Current mode *Fig 2-B*

FIQ mode:
- more registers are banked, thus making servicing FIQs ("fast interrupts") even faster.



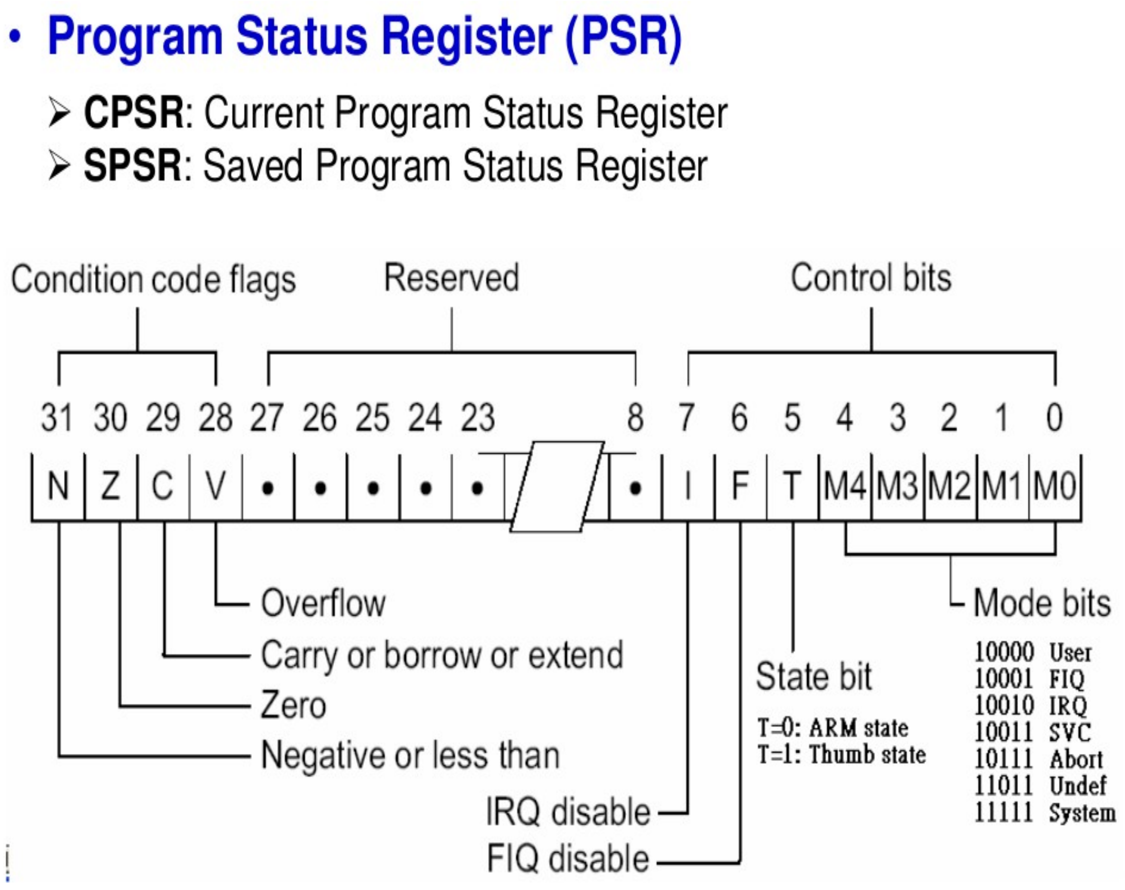
Register Organization Summary


User
System
FIQ
IRQ
SVC
Undef
Abort
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
User mode r0-r7, r15, and cpsr
User mode r0-r12, r15, and cpsr
spsr
Thumb state Low registers
Thumb state High registers
Note: System mode uses the User mode register set

- System mode is not displayed as it uses the identical register set as User mode.
- When running in Thumb mode (16-bit instructions), typically the low 8 registers are available.

The CPSR - Current Program Status Register

The ARM core uses the cpsr to monitor and control internal operations. The cpsr is a dedicated 32-bit register and resides in the register file. Figure 2.3 shows the basic layout of a generic program status register. Note that the shaded parts are reserved for future expansion.



The cpsr is divided into four fields, each 8 bits wide: flags, status, extension, and control. In current designs the extension and status fields are reserved for future use. The control field contains the processor mode, state, and interrupt mask bits. The flags field contains the condition flags.

Some ARM processor cores have extra bits allocated. For example, the J bit, which can be found in the flags field, is only available on Jazelle-enabled processors, which execute 8-bit instructions. We will discuss Jazelle more in Section 2.2.3. It is highly probable that future designs will assign extra bits for the monitoring and control of new features.

<<

Source [Optional / FYI]

Status flags and condition codes

I've already mentioned part of this in the introduction, so I'll make this brief. The ARM processor has 4 status flags, (**Z**)ero, (**N**)egative, (**C**)arry and signed o(**V**)erflow, which can be found in the program status register. There are actually two of these: one for the *current* status (CPSR) and a *saved* status register (SPSR), which is used in interrupt handlers. You won't have to deal with either of these, though, as reacting to status registers usually goes through the conditional codes (table 23.4). But first, a few words about the flags themselves:

- **Zero** (Z). If the result of operation was 0.
- **Negative** (N). Result was negative (i.e. most significant bit set).
- **Carry bit set** (C). If the 'mostest' significant bit is set (like bit 32 for 32bit operations).
- **Arithmetic overflow** (V). Like adding two positive numbers and getting a negative number because the result got too big for the registers.

Each of the data instructions can set the status flags by appending -s to the instruction, except for cmp, cmn, tst and teq, which always set the flags.

Table 23.4 lists 16 affixes that can be added to the basic branch instruction. For example, bne Label would jump to Label if the status is non-zero, and continue with the next instruction if it isn't.

Affix	Flags	Description
eq	Z=1	Zero (EQual to 0)
ne	Z=0	Not zero (Not Equal to 0)
cs / hs	C=1	Carry Set / unsigned Higher or Same
cc / lo	C=0	Carry Clear / unsigned LOwer
mi	N=1	Negative (MInus)
pl	N=0	Positive or zero (PLus)
vs	V=1	Sign overflow (oVerflow Set)
vc	V=0	No sign overflow (oVerflow Clear)
hi	C=1 & Z=0	Unsigned HIgher
ls	C=0 | Z=1	Unsigned Lower or Same
ge	N=V	Signed Greater or Equal
lt	N != V	Signed Less Than
gt	Z=0 & N=V	Signed Greater Than
le	Z=1 | N != V	Signed Less or Equal
al	-	ALways (default)
nv	-	NeVer

Table 23.4: conditional affixes.

…

>>

ARM Exceptions and Modes

Source :: "ARM System Developer's Guide – Designing and Optimizing System Software" by Sloss, Symes and Wright, published by Elsevier.

An exception is any condition that needs to halt the normal sequential execution of instructions. Examples are when the ARM core is reset, when an instruction fetch or memory access fails, when an undefined instruction is encountered, when a software interrupt instruction is executed, or when an external interrupt has been raised. Exception handling is the method of processing these exceptions.

Most exceptions have an associated software exception handler—a software routine that executes when an exception occurs. For instance, a Data Abort exception will have a Data Abort handler. The handler first determines the cause of the exception and then services the exception. Servicing takes place either within the handler or by branching to a specific service routine. The Reset exception is a special case since it is used to initialize an embedded system.

…

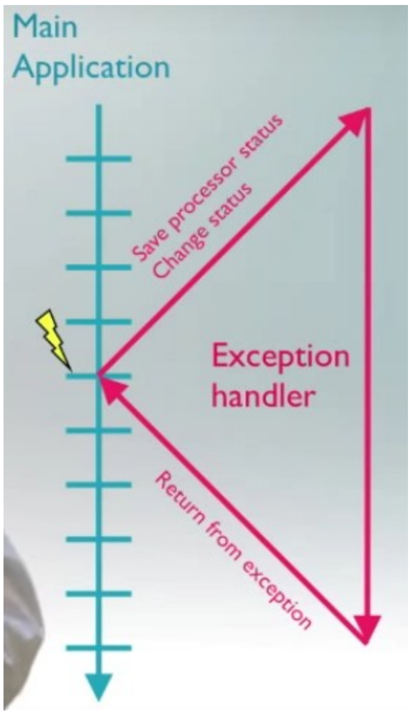
<<

Exception Handling in a Nutshell

1. Save processor state
   - copy CPSR into SPSR_<mode>
   - save return address (PC+4) into LR_<mode>
2. Change processor status for exception –
   - in the CPSR:
     - set mode field bits
     - set ARM or Thumb state
     - disable Interrupts (if appropriate)

     set PC to Vector address (from the vector switching table)
3. Execute exception handler
4. Return to application
   - restore CPSR from SPSR_<mode>
   - restore PC from LR_<mode>

>>



When an exception causes a mode change, the core automatically :

- saves the cpsr to the spsr of the exception mode
- saves the pc to the lr of the exception mode

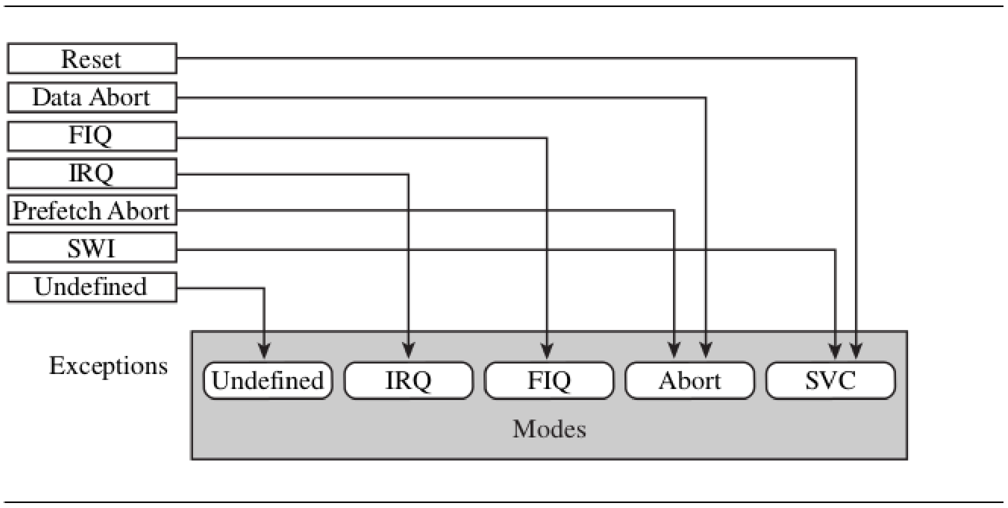


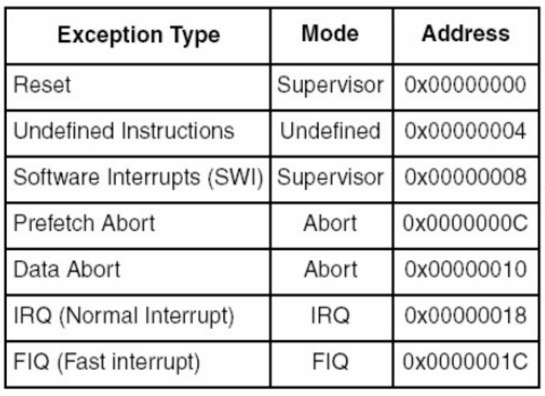
Figure 9.1 Exceptions and associated modes.

Exceptions, Interrupts, and the Vector Table

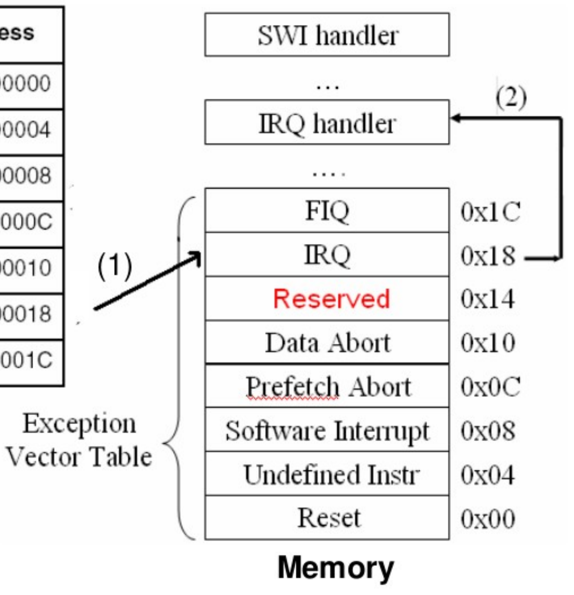
When an exception or interrupt occurs, the (ARM) processor sets the pc to a specific memory address. The address is within a special address range called the vector table. The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt. << equivalent to the IDT on x86 >>.

The memory map address 0x00000000 is reserved for the vector table, a set of 32-bit words. On some processors the vector table can be optionally located at a higher address in memory (starting at the offset `0xffff0000` – virtual address). Operating systems such as Linux and Microsoft's embedded products can take advantage of this feature.

- **ARM Exception Vectors**



Exception Type	Mode	Address
Reset	Supervisor	0x00000000
Undefined Instructions	Undefined	0x00000004
Software Interrupts (SWI)	Supervisor	0x00000008
Prefetch Abort	Abort	0x0000000C
Data Abort	Abort	0x00000010
IRQ (Normal Interrupt)	IRQ	0x00000018
FIQ (Fast interrupt)	FIQ	0x0000001C



When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table (see Table 2.6). Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:

- Reset vector is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.
- Undefined instruction vector is used when the processor cannot decode an instruction.
- Software interrupt vector is called when you execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine << *on Linux – system calls* >>
- Prefetch abort vector occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.
- Data abort vector is similar to a prefetch abort but is raised when an instruction attempts to access data memory without the correct access permissions << *this is what ultimately (from the MMU to OS fault handler to...) leads to your "Segmentation fault, [core dumped]" signals, etc* >>

Table 2.6 The vector table.

Exception/interrupt	Shorthand	Address	High address
Reset	RESET	0x00000000	0xffff0000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	DABT	0x00000010	0xffff0010
Reserved	—	0x00000014	0xffff0014
Interrupt request	IRQ	0x00000018	0xffff0018
Fast interrupt request	FIQ	0x0000001c	0xffff001c

- Reset vector is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.
- Interrupt request vector is used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the cpsr.
- Fast interrupt request vector is similar to the interrupt request but is reserved for hardware requiring faster response times. It can only be raised if FIQs are not masked in the cpsr.

<<
Implementation On Linux

ARM vector table base address is at physical address 0x0, and usually at virtual address 0xffff0000 .

In *arch/arm/mm/mmu.c* :

```
#define vectors_base()  (vectors_high() ? 0Xffff0000 : 0)
...
```

Where is the ARM vector table initialized and setup in Linux? Short answer:
arch/arm/kernel/traps.c:early_trap_init()

See this post for more details.
>>

ARM Assembly – the Basics

Resources-

https://community.arm.com/processors/b/blog/posts/hello-world-in-assembly
ARM assembler in Raspberry Pi – Chapter 1
PRACTICAL ARM EXPLOITATION, LAB MANUAL [PDF]
-pages 4 – 18.

http://www.coranac.com/tonc/text/asm.htm

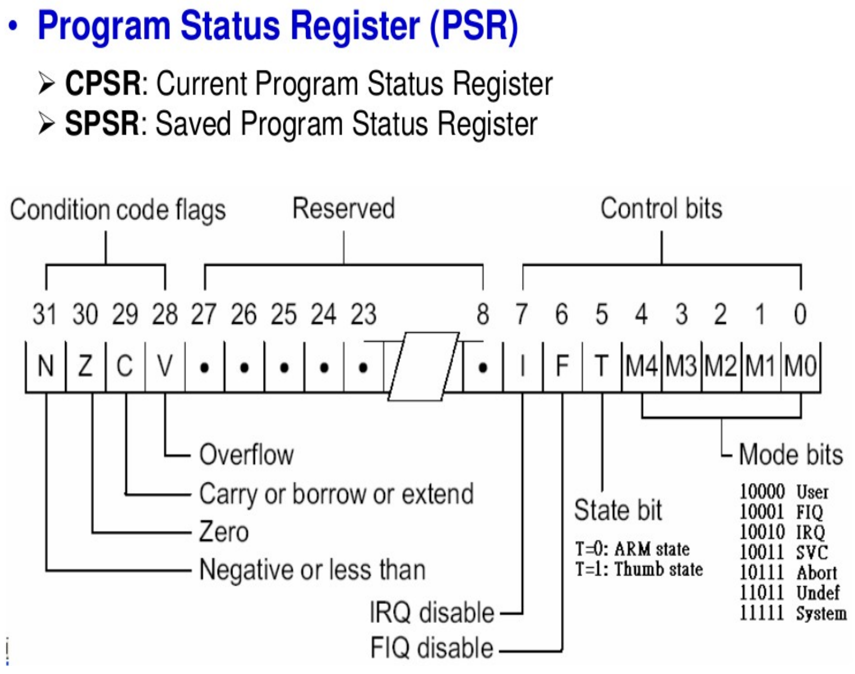
The reader is strongly advised to read through a few chapters and try out small ARM assembly language programs from here:
ARM assembler in Raspberry Pi – Chapter 1

Branching

Ch 5: Branching

On Conditional flags (set in CPSR).

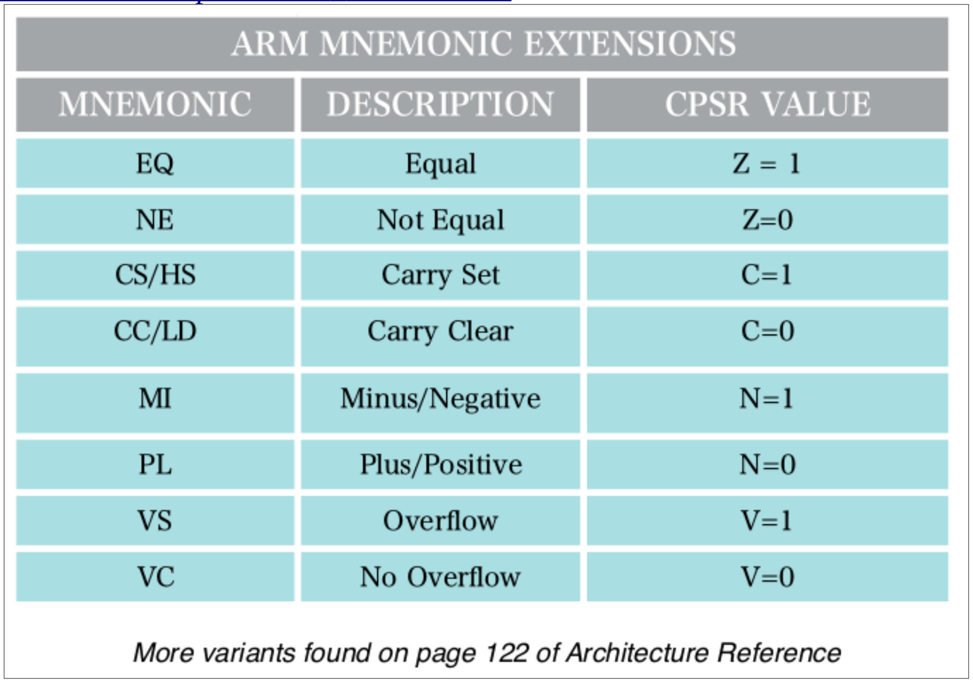
<<
Recall the CPSR:



- **Program Status Register (PSR)**
  - **CPSR**: Current Program Status Register
  - **SPSR**: Saved Program Status Register

>>

<<
Source – Practical ARM Exploitation – Lab Manual



ARM MNEMONIC EXTENSIONS		
MNEMONIC	**DESCRIPTION**	**CPSR VALUE**
EQ	Equal	Z = 1
NE	Not Equal	Z=0
CS/HS	Carry Set	C=1
CC/LD	Carry Clear	C=0
MI	Minus/Negative	N=1
PL	Plus/Positive	N=0
VS	Overflow	V=1
VC	No Overflow	V=0

More variants found on page 122 of Architecture Reference

>>
…
The semantics of these four condition codes in instructions updating the cpsr are roughly the following

- N will be enabled if the result of the instruction yields a negative number. Disabled otherwise.
- Z will be enabled if the result of the instruction yields a zero value. Disabled if nonzero.
- C will be enabled if the result of the instruction yields a value that requires a 33rd bit to be fully represented. For instance an addition that overflows the 32 bit range of integers. There is a special case for C and subtractions where a non-borrowing subtraction enables it, disabled otherwise: subtracting a larger number to a smaller one enables C, but it will be disabled if the subtraction is done the other way round.
- V will be enabled if the result of the instruction yields a value that cannot be represented in 32 bits two's complement.

So we have all the needed pieces to perform branches conditionally. But first, let's start comparing two values. We use the instruction cmp for this purpose.

```
cmp r1, r2 /* updates cpsr doing "r1 - r2", but r1 and r2 are not modified */
```

This instruction subtracts to the value in the first register the value in the second register. Examples of what could happen in the snippet above?

- If r2 had a value (strictly) greater than r1 then N would be enabled because r1-r2 would yield a negative result.
- If r1 and r2 had the same value, then Z would be enabled because r1-r2 would be zero.
- If r1 was 1 and r2 was 0 then r1-r2 would not borrow, so in this case C would be enabled. If the values were swapped (r1 was 0 and r2 was 1) then C would be disabled because the subtraction does borrow.
- If r1 was 2147483648 (the largest positive integer in 32 bit two's complement) and r1 was -1 then r1-r2 would be 2147483649 but such number cannot be represented in 32 bit two's complement, so V would be enabled to signal this.

How can we use these flags to represent useful conditions for our programs?

- EQ (equal) When Z is enabled (Z is 1)
- NE (not equal). When Z is disabled. (Z is 0)
- GE (greater or equal than, in two's complement). When both V and N are enabled or disabled (V is N)
- LT (lower than, in two's complement). This is the opposite of GE, so when V and N are not both enabled or disabled (V is not N)
- GT (greather than, in two's complement). When Z is disabled and N and V are both enabled or disabled (Z is 0, N is V)
- LE (lower or equal than, in two's complement). When Z is enabled or if not that, N and V are both enabled or disabled (Z is 1. If Z is not 1 then N is V)
- MI (minus/negative) When N is enabled (N is 1)
- PL (plus/positive or zero) When N is disabled (N is 0)
- VS (overflow set) When V is enabled (V is 1)
- VC (overflow clear) When V is disabled (V is 0)
- HI (higher) When C is enabled and Z is disabled (C is 1 and Z is 0)
- LS (lower or same) When C is disabled or Z is enabled (C is 0 or Z is 1)
- CS/HS (carry set/higher or same) When C is enabled (C is 1)
- CC/LO (carry clear/lower) When C is disabled (C is 0)

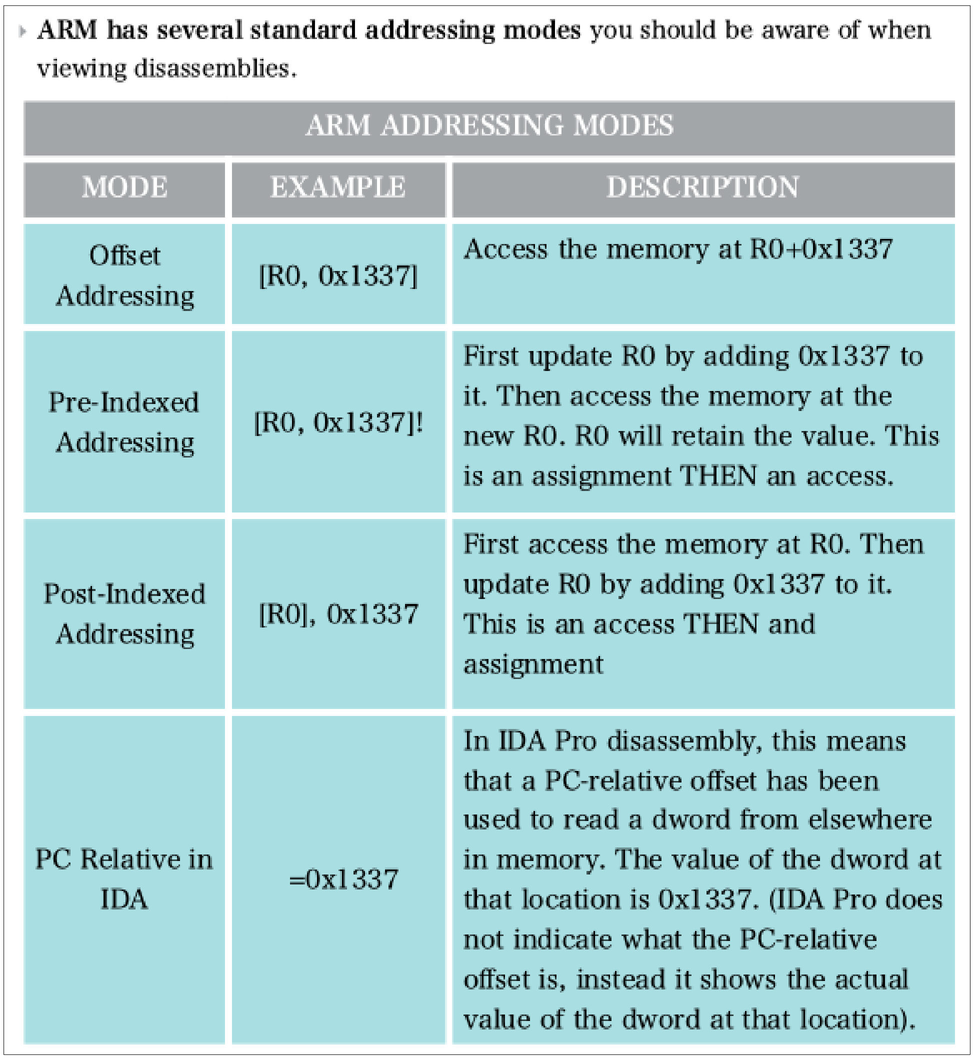
These conditions can be combined to our b instruction to generate new instructions. This way, beq will branch only if Z is 1. If the condition of a conditional branch is not met, then the branch is ignored and the next instruction will be run. It is the programmer task to make sure that the condition codes are properly set prior a conditional branch.

```
/* -- compare01.s */
.text
.global main
main:
     mov r1, #2       /* r1 ← 2 */
     mov r2, #2       /* r2 ← 2 */
     cmp r1, r2       /* update cpsr condition codes with the value of
r1-r2 */
```

```
    beq case_equal   /* branch to case_equal only if Z = 1 */
case_different :
    mov r0, #2       /* r0 ← 2 */
    b end            /* branch to end */
case_equal:
    mov r0, #1       /* r0 ← 1 */
end:
    bx lr
```

If you run this program it will return an error code of 1 because both r1 and r2 have the same value. Now change mov r1, #2 in line 5 to be mov r1, #3 and the returned error code should be 2. Note that case_different we do not want to run the case_equal instructions, thus we have to branch to end (otherwise the error code would always be 1).

Addressing Modes

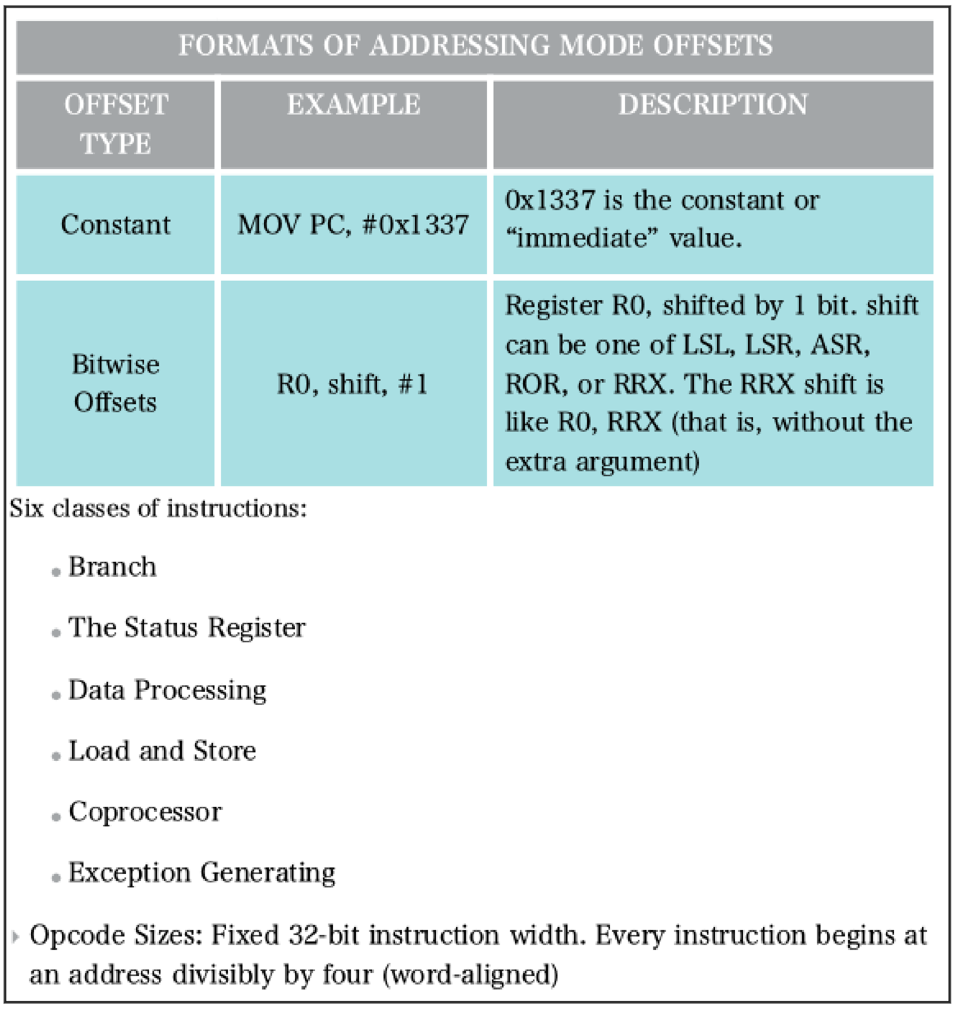


- **ARM has several standard addressing modes** you should be aware of when viewing disassemblies.

ARM ADDRESSING MODES		
MODE	EXAMPLE	DESCRIPTION
Offset Addressing	[R0, 0x1337]	Access the memory at R0+0x1337
Pre-Indexed Addressing	[R0, 0x1337]!	First update R0 by adding 0x1337 to it. Then access the memory at the new R0. R0 will retain the value. This is an assignment THEN an access.
Post-Indexed Addressing	[R0], 0x1337	First access the memory at R0. Then update R0 by adding 0x1337 to it. This is an access THEN and assignment
PC Relative in IDA	=0x1337	In IDA Pro disassembly, this means that a PC-relative offset has been used to read a dword from elsewhere in memory. The value of the dword at that location is 0x1337. (IDA Pro does not indicate what the PC-relative offset is, instead it shows the actual value of the dword at that location).

Source – Practical ARM Exploitation – Lab Manual

(contd.)

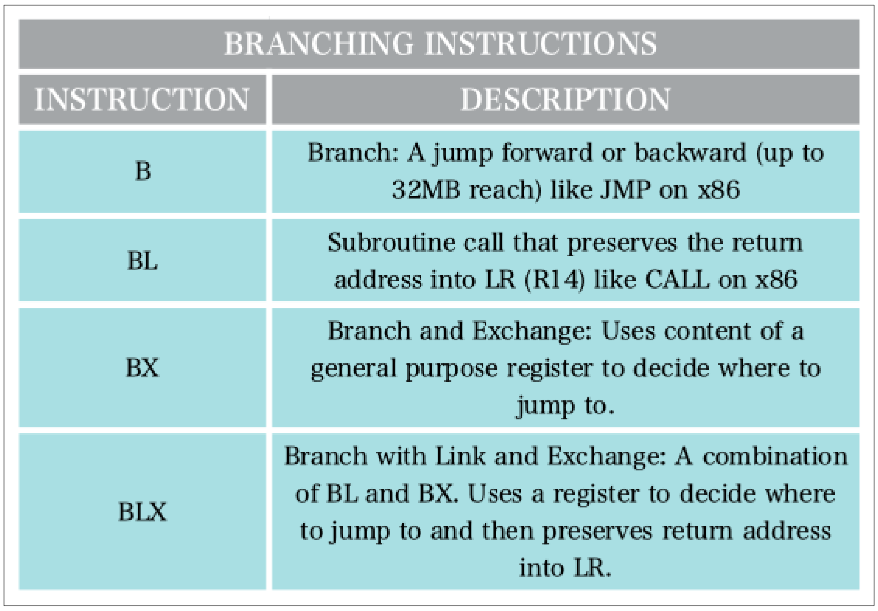


FORMATS OF ADDRESSING MODE OFFSETS		
OFFSET TYPE	EXAMPLE	DESCRIPTION
Constant	MOV PC, #0x1337	0x1337 is the constant or "immediate" value.
Bitwise Offsets	R0, shift, #1	Register R0, shifted by 1 bit. shift can be one of LSL, LSR, ASR, ROR, or RRX. The RRX shift is like R0, RRX (that is, without the extra argument)

Six classes of instructions:

- Branch
- The Status Register
- Data Processing
- Load and Store
- Coprocessor
- Exception Generating

- Opcode Sizes: Fixed 32-bit instruction width. Every instruction begins at an address divisibly by four (word-aligned)



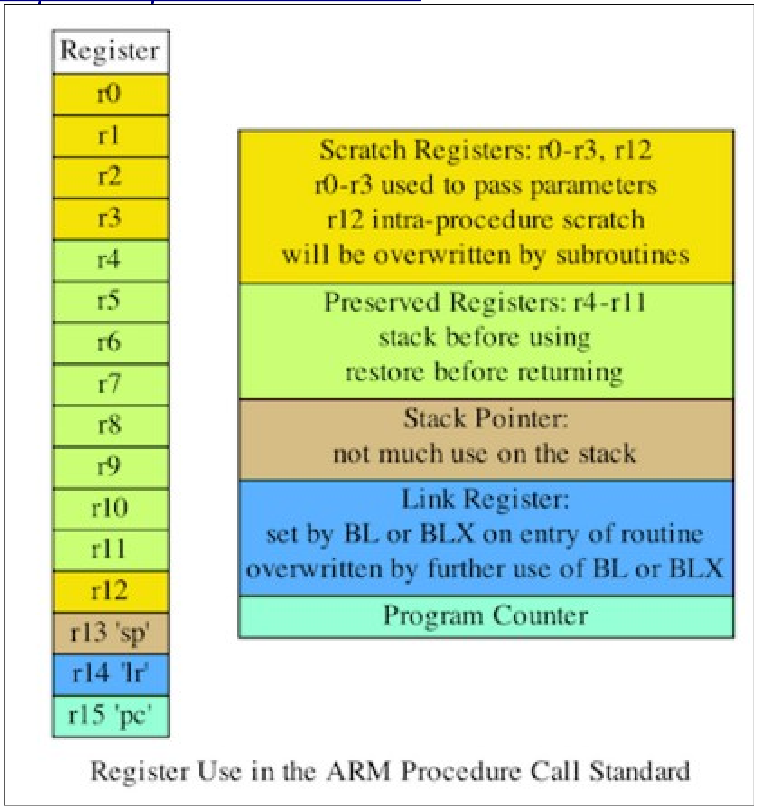
BRANCHING INSTRUCTIONS	
INSTRUCTION	DESCRIPTION
B	Branch: A jump forward or backward (up to 32MB reach) like JMP on x86
BL	Subroutine call that preserves the return address into LR (R14) like CALL on x86
BX	Branch and Exchange: Uses content of a general purpose register to decide where to jump to.
BLX	Branch with Link and Exchange: A combination of BL and BX. Uses a register to decide where to jump to and then preserves return address into LR.

Another way to change program flow is to directly change the value of PC (R15) directly (impossible on x86):

```
MOV PC, #1337 ;Redirect execution to 1337
```

Function Calling on the ARM

Source – How to call a function from ARM assembler


Register
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 'sp'
r14 'lr'
r15 'pc'
Scratch Registers: r0-r3, r12
r0-r3 used to pass parameters
r12 intra-procedure scratch
will be overwritten by subroutines
Preserved Registers: r4-r11
stack before using
restore before returning
Stack Pointer:
not much use on the stack
Link Register:
set by BL or BLX on entry of routine
overwritten by further use of BL or BLX
Program Counter
Register Use in the ARM Procedure Call Standard

Source

…

Function calls use a special kind of branching instruction, namely bl. It works exactly like the normal branch, except that it saves the address after the bl in the link register (r14 or lr) so that you know where to return to after the called function is finished. In principle, you can return with to the function using 'mov pc, lr', which points the program counter back to the calling function, but in practice you might be better off with bx (Branch and eXchange). The difference is that bx can also switch between ARM and THUMB states, which isn't possible with the mov return.

Unlike b and bl, bx takes a register as its argument, instead of a label. This register will usually be lr, but the others are allowed as well.

There's also the matter of passing parameters to the function and returning values from it. In principle you're free to use any system you like, it is recommended to ARM's own ARM Architecture Procedure Call Standard (AAPCS) for this. For the majority of the work this can be summarized like this:

- The first 4 arguments go into r0-r3.
  Later ones go on the stack, in order of appearance.
- The return value goes into r0.
- The scratch registers r0-r3 (and r12) are free to use without restriction in a function. As such, after *calling* a function they should be considered 'dirty'.
- The other registers must leave a function with the same values as they came in. Push them on the stack before use, and pop them when leaving the function. Note that another bl sets lr, so stack that one too in that case.

…

An Example to Demonstrate Function Calling on ARM

We consider:

1. Calling a function with 4 integet parameters
2. Calling a function with 4 integet parameters + a local char buffer

Environment:
A Qemu-emulated ARM926EJ-S rev 5 (v5l) (ARM-32) running the 4.8.12-yocto-standard Linux kernel built with Yocto Poky!

```
Yocto # cat /etc/issue
Poky (Yocto Project Reference Distro) 2.2.1 \n \l

Yocto #
```

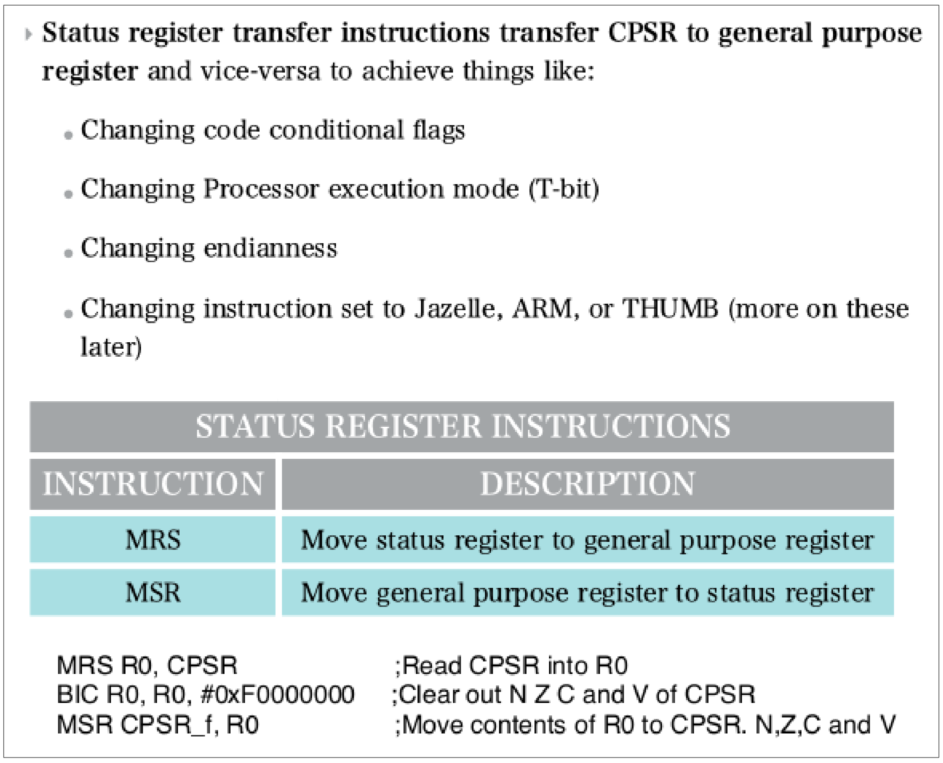
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

static void bar(int a, int b, int c, int d)
{
        char buf[12];                     << local buffer >>
        memset(buf, 'A', 12);
}
static int foo(int a, int b, int c, int d)
{
        return (a+b+c+d);
}

int main (int argc, char **argv)
{
    foo(1,2,3,4);
    bar(5,6,7,8);
    exit(0);
}
```

Makefile

```
ALL := params_arm

CXX=
CC=${CXX}gcc
LD=${CXX}ld
CFLAGS_DBG=-g -ggdb -O0 -Wall
CFLAGS=-Wall

all := ${ALL}

params_arm: params_arm.c
      ${CC} ${CFLAGS} -o $@ $<

clean:
      rm -fv ${ALL} *.o
```

```
Yocto # ./params_arm ; echo $?
0
Yocto #
```

```
$ objdump -d ./params_arm     << NOTE- the assembly you see might vary from that shown
below; depends on the compiler version (we're using gcc ver 6.2.0) >>

[...]

<<
ldr : load from memory into a register
str : load from a register into a memory location
>>

00010444 <bar>:
   10444:    e92d4800     push   {fp, lr}
   10448:    e28db004     add    fp, sp, #4
   1044c:    e24dd020     sub    sp, sp, #32  << make space on stack for the local
buffer >>
   10450:    e50b0018     str    r0, [fp, #-24]     ; 0xffffffe8 << put [r0] into mem @
                                                    addr [fp-24], i.e., [fp-24] ← r0 >>
   10454:    e50b101c     str    r1, [fp, #-28]     ; 0xffffffe4
   10458:    e50b2020     str    r2, [fp, #-32]     ; 0xffffffe0
   1045c:    e50b3024     str    r3, [fp, #-36]     ; 0xffffffdc
   10460:    e24b3010     sub    r3, fp, #16
   10464:    e3a0200c     mov    r2, #12
   10468:    e3a01041     mov    r1, #65      ; 0x41
   1046c:    e1a00003     mov    r0, r3
   10470:    ebffffa3     bl     10304 <memset@plt>
   10474:    e1a00000     nop                 ; (mov r0, r0)
   10478:    e24bd004     sub    sp, fp, #4
   1047c:    e8bd8800     pop    {fp, pc}  << notice how PC is set to the saved 'lr' >>

00010480 <foo>:
   10480:    e52db004     push   {fp}         ; (str fp, [sp, #-4]!)

<<
The 'C' code of function 'foo()' is:
return (a+b+c+d);

The compiler generates assembly to perform the addition of the 4 parameters by
[1] creating temporary space for them on the stack,
[2] loading them from registers (r0-r3) to stack memory (with the STR),
[3] then loading from (stack) memory to registers (with the LDR) and using ADD
instructions, and,
[4] finally puts the result into r0 (ret value).
>>

   10484:     e28db000     add     fp, sp, #0
   10488:     e24dd014     sub     sp, sp, #20        << [1] >>
   1048c:     e50b0008     str     r0, [fp, #-8]      << [2] >>
   10490:     e50b100c     str     r1, [fp, #-12] << Effectively, [fp-12] ← r1 >>
   10494:     e50b2010     str     r2, [fp, #-16]
   10498:     e50b3014     str     r3, [fp, #-20]      ; 0xffffffec
   1049c:     e51b2008     ldr     r2, [fp, #-8]      << [3] >>
   104a0:     e51b300c     ldr     r3, [fp, #-12] << effectively, r3 ← [fp-12] >>
   104a4:     e0822003     add     r2, r2, r3         << [3] >>
   104a8:     e51b3010     ldr     r3, [fp, #-16]
   104ac:     e0822003     add     r2, r2, r3
   104b0:     e51b3014     ldr     r3, [fp, #-20]      ; 0xffffffec
   104b4:     e0823003     add     r3, r2, r3
   104b8:     e1a00003     mov     r0, r3         << [4] return value into r0 >>
   104bc:     e24bd000     sub     sp, fp, #0
   104c0:     e49db004     pop     {fp}          ; (ldr fp, [sp], #4)
   104c4:     e12fff1e     bx      lr

000104c8 <main>:
   104c8:     e92d4800     push    {fp, lr}    << the frame pointer (fp) and link
              register (lr = RET address) is saved on the stack >>
   104cc:     e28db004     add     fp, sp, #4
   104d0:     e24dd008     sub     sp, sp, #8
   104d4:     e50b0008     str     r0, [fp, #-8]
   104d8:     e50b100c     str     r1, [fp, #-12]
   104dc:     e3a03004     mov     r3, #4       << parameters written in right-to-
                                      left order into the registers r3-r0 >>
   104e0:     e3a02003     mov     r2, #3
   104e4:     e3a01002     mov     r1, #2
   104e8:     e3a00001     mov     r0, #1
   104ec:     ebffffe3     bl      10480 <foo> << bl: branch and link instruction >>
   104f0:     e3a03008     mov     r3, #8   << parameters written in right-to-left
                                           order into the registers r3-r0 >>
   104f4:     e3a02007     mov     r2, #7
   104f8:     e3a01006     mov     r1, #6
   104fc:     e3a00005     mov     r0, #5
   10500:     ebffffcf     bl      10444 <bar> << bl: branch and link instruction >>
   10504:     e3a00000     mov     r0, #0
   10508:     ebffff7a     bl      102f8 <exit@plt>

[...]

$
```



- **Status register transfer instructions transfer CPSR to general purpose register** and vice-versa to achieve things like:
  - Changing code conditional flags
  - Changing Processor execution mode (T-bit)
  - Changing endianness
  - Changing instruction set to Jazelle, ARM, or THUMB (more on these later)

STATUS REGISTER INSTRUCTIONS	
INSTRUCTION	DESCRIPTION
MRS	Move status register to general purpose register
MSR	Move general purpose register to status register

```
MRS R0, CPSR              ;Read CPSR into R0
BIC R0, R0, #0xF0000000   ;Clear out N Z C and V of CPSR
MSR CPSR_f, R0            ;Move contents of R0 to CPSR. N,Z,C and V
```

Source – Practical ARM Exploitation – Lab Manual

Load/Store, etc instructions – please lookup in the above "lab manual".

Quick Comparison

Property	**Intel x86**	**ARM**
CPU Type	CISC [1]	RISC [1]
Alignment	Varies	Strict- all instructions aligned to word size (4 bytes on 32-bit)
Direct access to memory (RAM)	Yes; instructions can access registers and/or memory directly, i.e., operands could be registers or memory locations	No, load-store architecture; operands have to be registers only. Instructions exist to load from memory (LDR) into a register and vice-versa (STR)
Register file	On the 32-bit, significant difference: small register file, less GPRs	Even on 32-bit, sufficient GPRs
Conditional Instructions	Some	ALL ARM instructions are conditional; compilers can optimize

[1] This distinction is blurring nowadays.

[Optional / FYI]

ARM Inline Assembly

[Extended Asm - Assembler Instructions with C Expression Operands]

Linux Journal: *GCC Inline Assembly and Its Usage in the Linux Kernel, Dibyendu Roy*
Issue #271, November 2016.

Basic Inline Extended Assembly Template

```
 asm [volatile] ( AssemblerTemplate
                    : OutputOperands
                    [ : InputOperands
                    [ : Clobbers ] ])
```

A quick example: query the current stack pointer value from a kernel module:

```
...
      int loc=5;
#if defined CONFIG_ARM
      static u32 val;

      __asm__ __volatile__ (
            "mov r3, sp"           << mov dest, src >>
            : "=r" (val) /* output operands */
            :            /* input operands */
            : "r3");     /* clobbers */

      pr_info("loc=0x%p, stack reg (sp) val = 0x%x\n", &loc, val);
#endif
...

Sample Output:
…
ARM / $ insmod myprj/kernel/misc/2p.ko
loc=0x8fbffdac, stack reg (sp) val = 0x8fbffdc0
ARM / $
```

Q. How to get the CPSR / SPSR register values?
A.

<< From here >>

To copy the PSR into a register:

```
MRS     R0, CPSR                 ; Copy CPSR into R0
MRS     R0, SPSR                 ; Copy SPSR into R0
```

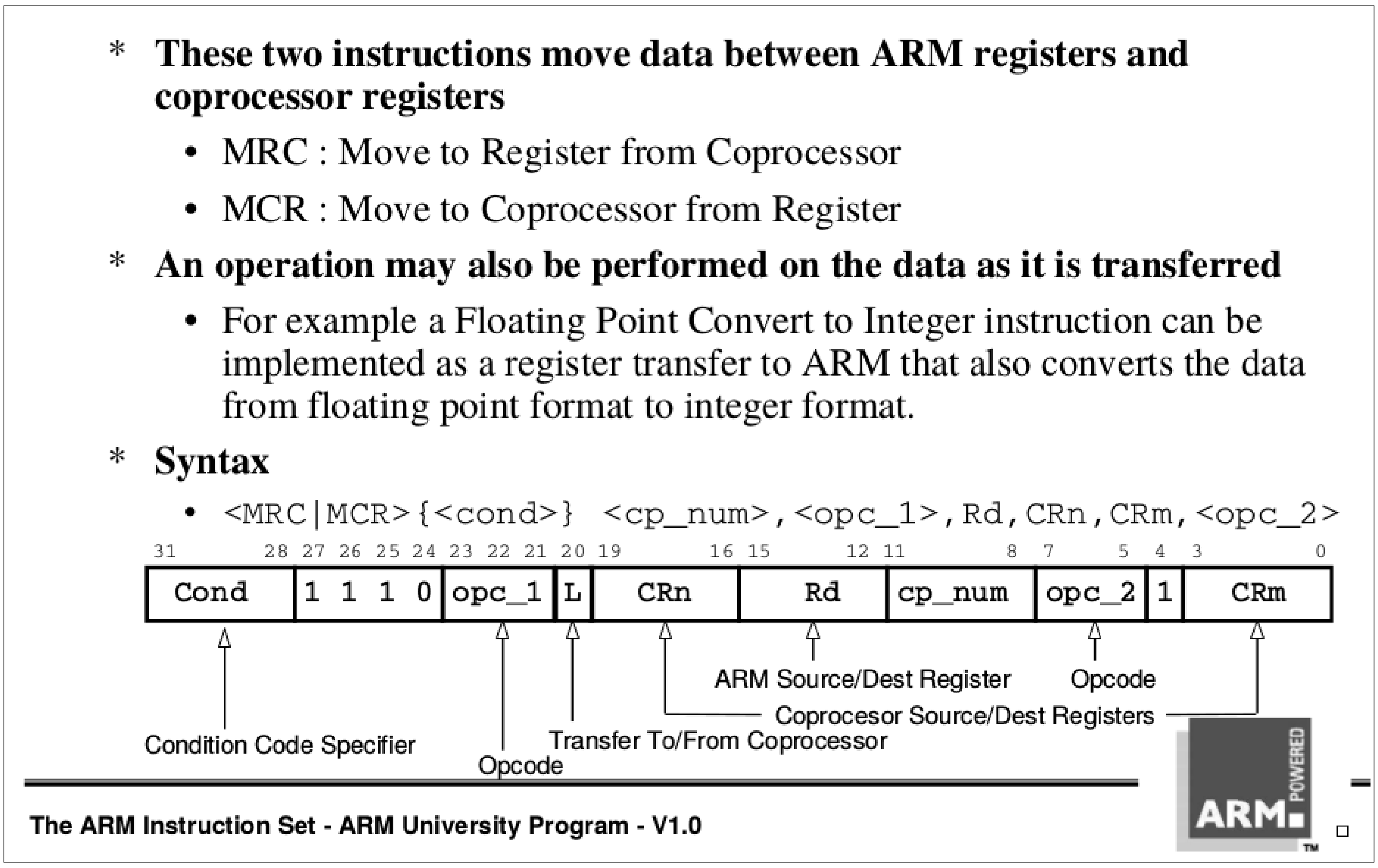
You have two PSRs - the **CPSR** which is the Current Program Status Register and **SPSR** which is the Saved Program Status Register (the previous processor mode's PSR). Each privileged mode has its own PSR, so the total available selection of PSR is:

• CPSR_all - current
• CPSR_flg - current, flags only
• SPSR - saved, current
• SPSR_svc - saved, SVC(32) mode
• SPSR_irq - saved, IRQ(32) mode
• SPSR_abt - saved, ABT(32) mode
• SPSR_und - saved, UND(32) mode
• SPSR_fiq - saved, FIQ(32) mode

You can only alter the SPSR of the mode you are currently in, thus if you are in SVC mode, you cannot MRS to update SPSR_fiq. The way to do this, should it be necessary, is to temporarily enter the mode relating to the SPSR you wish to update it, and do it that way. Using the *_flg* suffix allows you to alter the flag bits without affecting the control bits.

In User(32) mode, the control bits of CPSR are protected, you can only alter the condition flags. In other modes, the entire CPSR is available. You should not specify R15 as a source or destination register. And finally, you must not attempt to access the SPSR in User(32) mode as it doesn't exist!

Coprocessor Register Transfers



* **These two instructions move data between ARM registers and coprocessor registers**
  - MRC : Move to Register from Coprocessor
  - MCR : Move to Coprocessor from Register
* **An operation may also be performed on the data as it is transferred**
  - For example a Floating Point Convert to Integer instruction can be implemented as a register transfer to ARM that also converts the data from floating point format to integer format.
* **Syntax**
  - `<MRC|MCR>{<cond>} <cp_num>,<opc_1>,Rd,CRn,CRm,<opc_2>`

31–28	27 26 25 24	23 22 21	20	19–16	15–12	11–8	7–5	4	3–0
Cond	1 1 1 0	opc_1	L	CRn	Rd	cp_num	opc_2	1	CRm

Condition Code Specifier
Opcode
Transfer To/From Coprocessor
ARM Source/Dest Register
Opcode
Coprocesor Source/Dest Registers

The ARM Instruction Set - ARM University Program - V1.0

Case Study: Programming the ARM System Coprocessor

Ref: TRM for the ARM1176jzf-s (the Raspberry Pi 2 uses this core)
[PDF]

Page 3-12:

Use of the system control coprocessor

This section describes the general method for use of the system control coprocessor.
You can access system control coprocessor CP15 registers with MRC and MCR instructions.

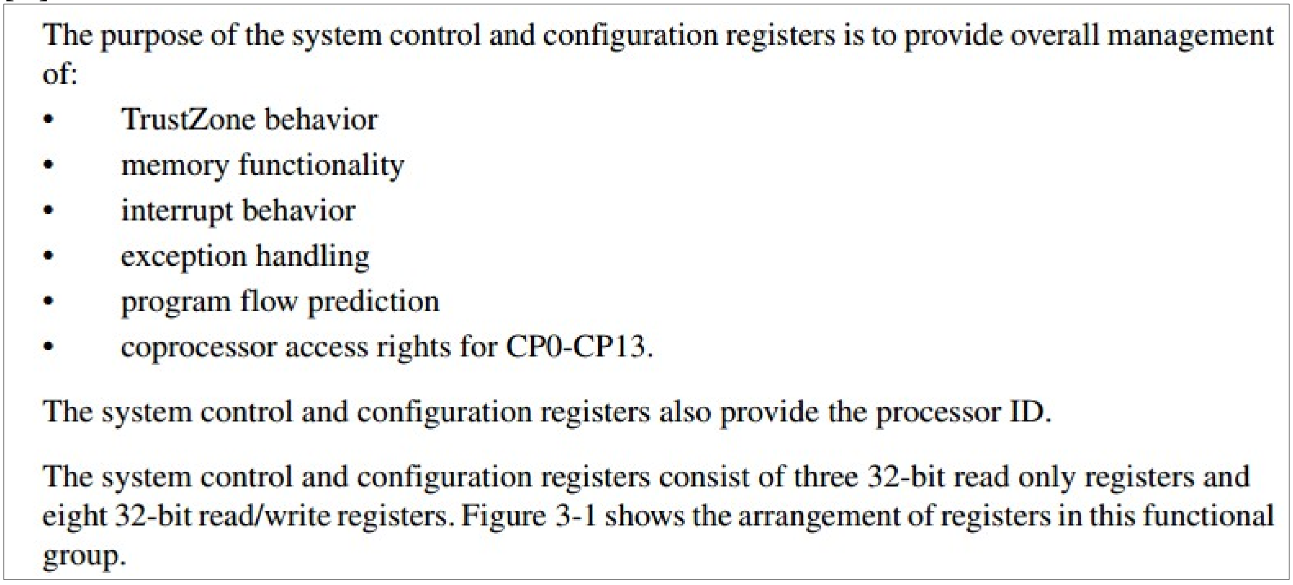
```
MCR{cond} P15,<Opcode_1>,<Rd>,<CRn>,<CRm>,<Opcode_2>
MRC{cond} P15,<Opcode_1>,<Rd>,<CRn>,<CRm>,<Opcode_2>
```

Figure 3-9 shows the instruction bit pattern of MRC and MCR instructions.
...
The CRn field of MRC and MCR instructions specifies the coprocessor register to access. The CRm field and Opcode_2 fields specify a particular operation when addressing registers. The L

bit distinguishes between an MRC (L=1) and an MCR (L=0).
...

Useful:
Table 3-1 on page 3-3 shows “System control coprocessor register functions”
Figure 3-1 to 3-8 shows in a concise manner similar information.

From the TRM:
Page 3-5:
[...]


The purpose of the system control and configuration registers is to provide overall management of:

- TrustZone behavior
- memory functionality
- interrupt behavior
- exception handling
- program flow prediction
- coprocessor access rights for CP0-CP13.

The system control and configuration registers also provide the processor ID.

The system control and configuration registers consist of three 32-bit read only registers and eight 32-bit read/write registers. Figure 3-1 shows the arrangement of registers in this functional group.

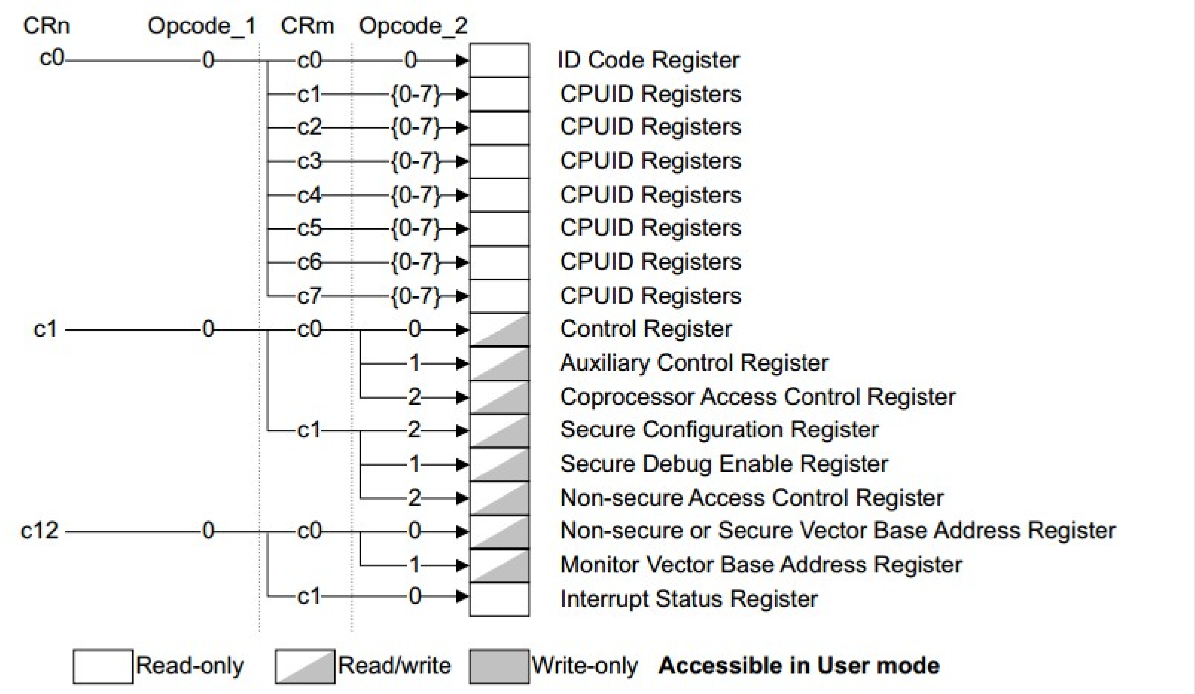


Figure 3-1 System control and configuration registers

From Fig 3-1 above, we can see that to access the CPUID registers, we must read (MRC opcode) from CP0 : opcode_1 = 0 : CRm = c1-c7 : opcode_2 = ‘n’ ; n={0-7} which CPU we want to read the ID of.

Also, on page 3-14 we find the coprocessor CP-15 (MMU control) registers and operations:

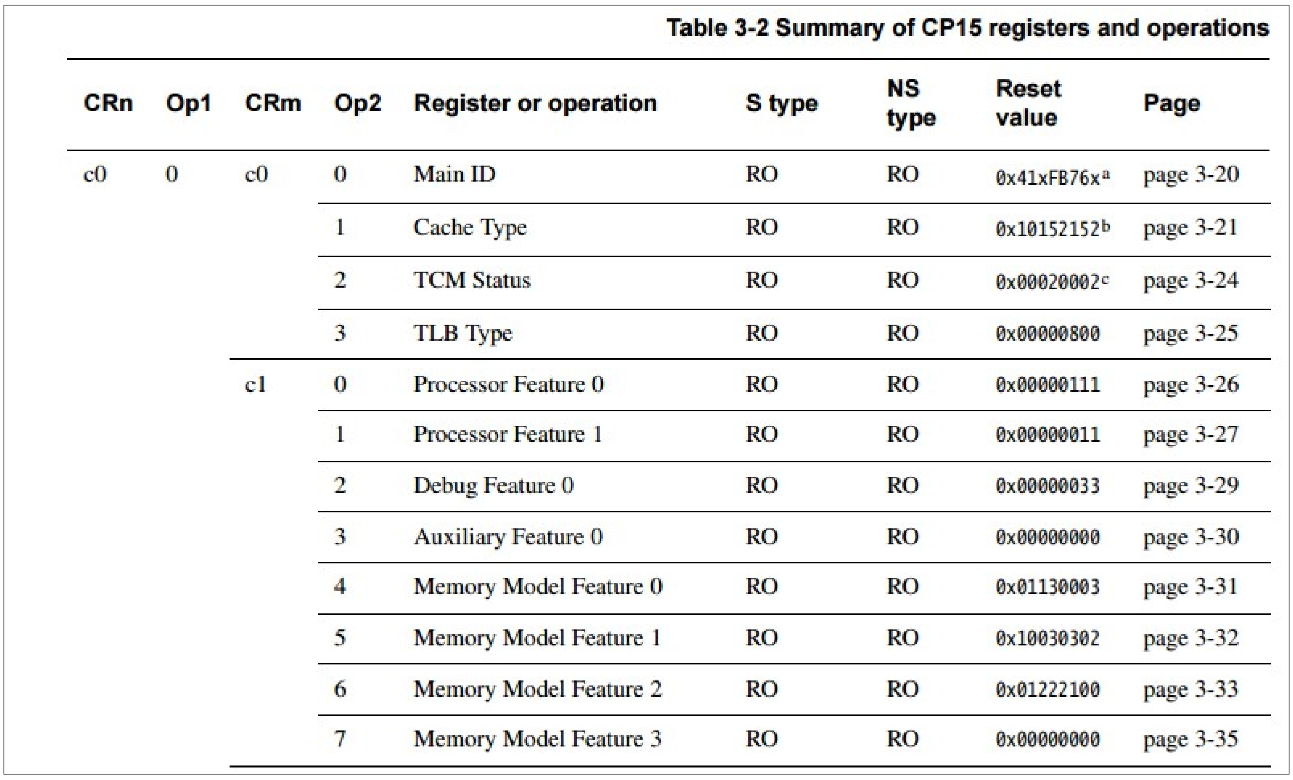


Table 3-2 Summary of CP15 registers and operations

CRn	Op1	CRm	Op2	Register or operation	S type	NS type	Reset value	Page
c0	0	c0	0	Main ID	RO	RO	`0x41xFB76x`[a]	page 3-20
			1	Cache Type	RO	RO	`0x10152152`[b]	page 3-21
			2	TCM Status	RO	RO	`0x00020002`[c]	page 3-24
			3	TLB Type	RO	RO	`0x00000800`	page 3-25
		c1	0	Processor Feature 0	RO	RO	`0x00000111`	page 3-26
			1	Processor Feature 1	RO	RO	`0x00000011`	page 3-27
			2	Debug Feature 0	RO	RO	`0x00000033`	page 3-29
			3	Auxiliary Feature 0	RO	RO	`0x00000000`	page 3-30
			4	Memory Model Feature 0	RO	RO	`0x01130003`	page 3-31
			5	Memory Model Feature 1	RO	RO	`0x10030302`	page 3-32
			6	Memory Model Feature 2	RO	RO	`0x01222100`	page 3-33
			7	Memory Model Feature 3	RO	RO	`0x00000000`	page 3-35

[…]

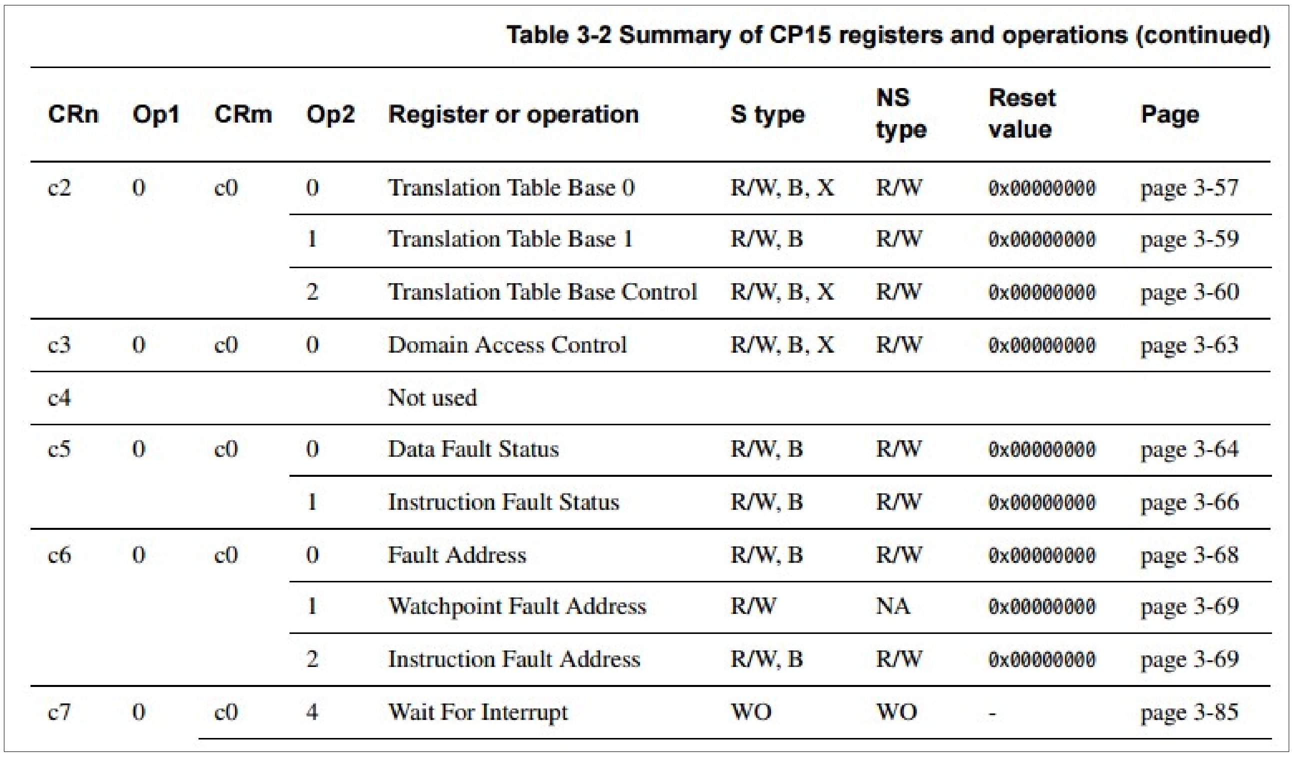


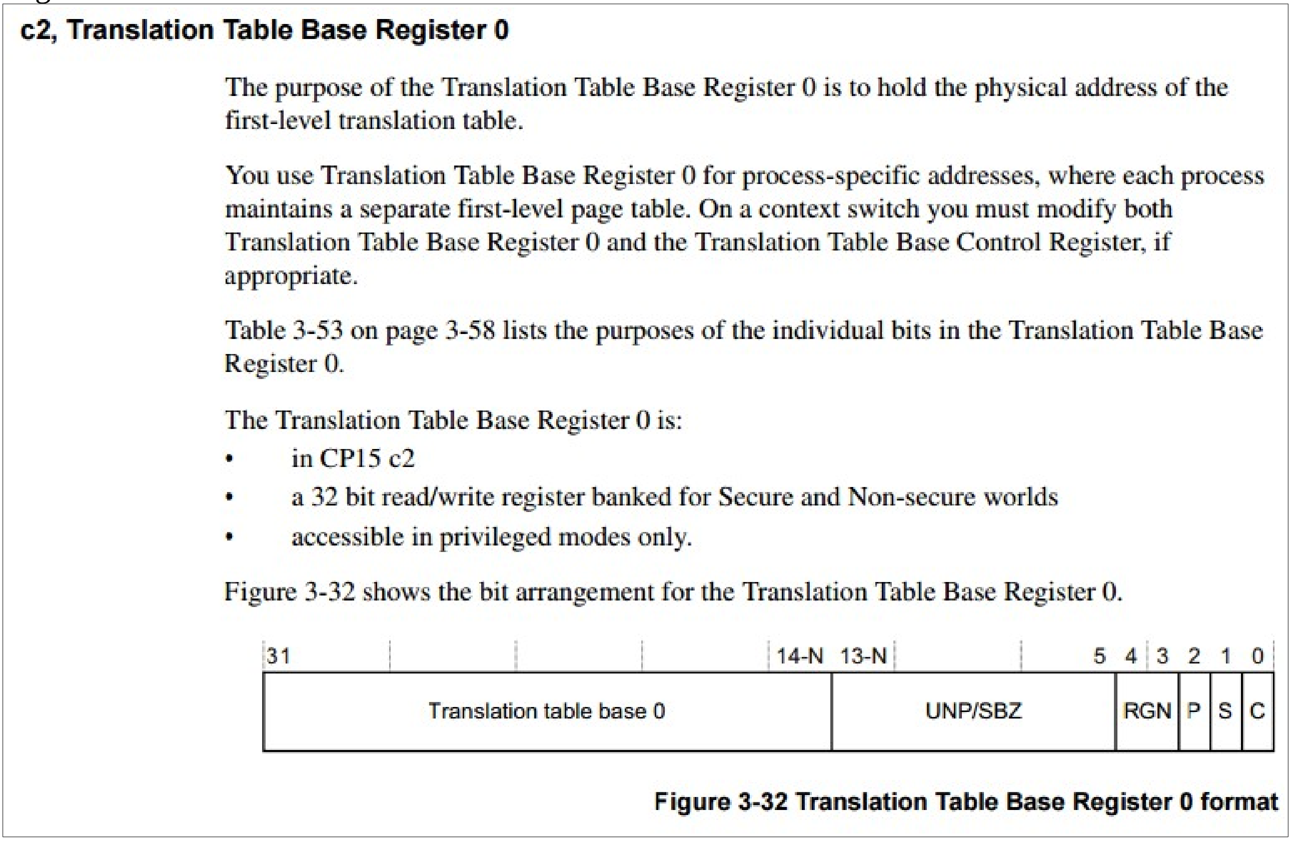
Table 3-2 Summary of CP15 registers and operations (continued)

CRn	Op1	CRm	Op2	Register or operation	S type	NS type	Reset value	Page
c2	0	c0	0	Translation Table Base 0	R/W, B, X	R/W	`0x00000000`	page 3-57
			1	Translation Table Base 1	R/W, B	R/W	`0x00000000`	page 3-59
			2	Translation Table Base Control	R/W, B, X	R/W	`0x00000000`	page 3-60
c3	0	c0	0	Domain Access Control	R/W, B, X	R/W	`0x00000000`	page 3-63
c4				Not used				
c5	0	c0	0	Data Fault Status	R/W, B	R/W	`0x00000000`	page 3-64
			1	Instruction Fault Status	R/W, B	R/W	`0x00000000`	page 3-66
c6	0	c0	0	Fault Address	R/W, B	R/W	`0x00000000`	page 3-68
			1	Watchpoint Fault Address	R/W	NA	`0x00000000`	page 3-69
			2	Instruction Fault Address	R/W, B	R/W	`0x00000000`	page 3-69
c7	0	c0	4	Wait For Interrupt	WO	WO	-	page 3-85

Example kernel module

mmu_peek:
Read the MMU (thus coprocessor CP-15) CP-15:c0 Main ID register value, the CP15:c1 control register value, and the Translation Table Base Register 0 (TTBR0) current value:

Page 3-57:



c2, Translation Table Base Register 0

The purpose of the Translation Table Base Register 0 is to hold the physical address of the first-level translation table.

You use Translation Table Base Register 0 for process-specific addresses, where each process maintains a separate first-level page table. On a context switch you must modify both Translation Table Base Register 0 and the Translation Table Base Control Register, if appropriate.

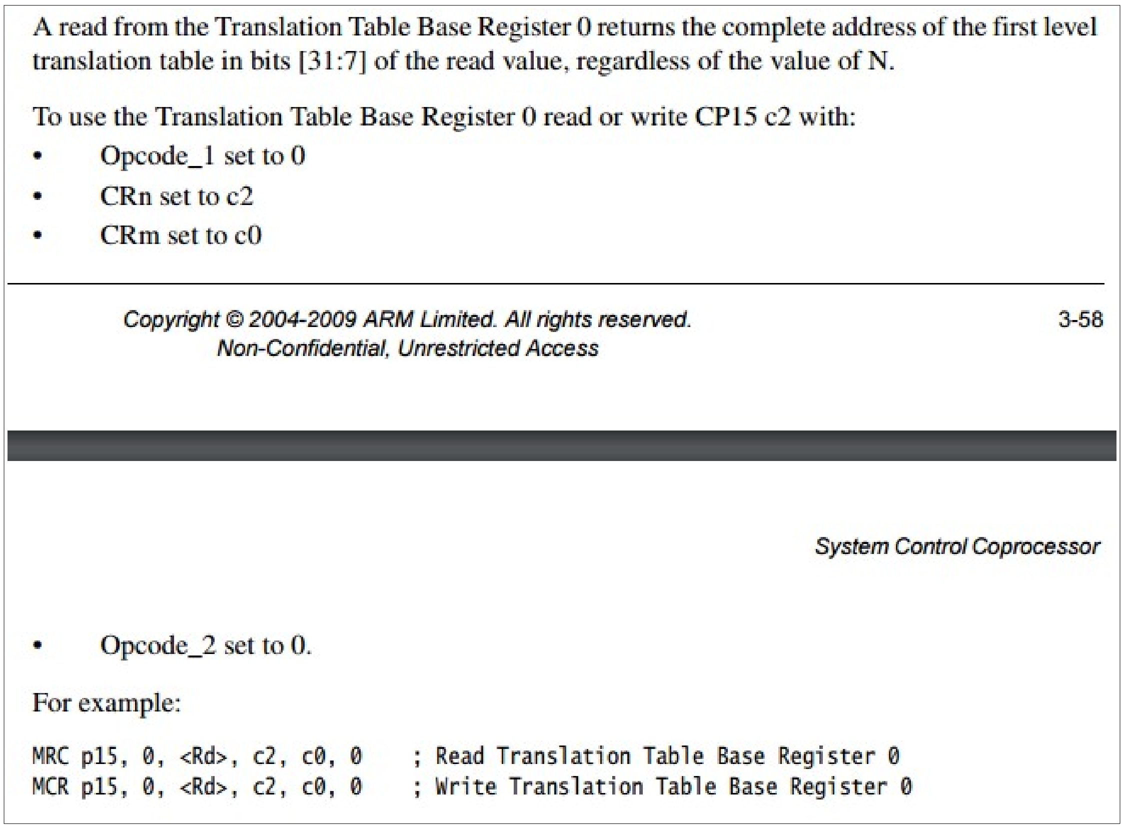
Table 3-53 on page 3-58 lists the purposes of the individual bits in the Translation Table Base Register 0.

The Translation Table Base Register 0 is:

- in CP15 c2
- a 32 bit read/write register banked for Secure and Non-secure worlds
- accessible in privileged modes only.

Figure 3-32 shows the bit arrangement for the Translation Table Base Register 0.

Figure 3-32 Translation Table Base Register 0 format


A read from the Translation Table Base Register 0 returns the complete address of the first level translation table in bits [31:7] of the read value, regardless of the value of N.

To use the Translation Table Base Register 0 read or write CP15 c2 with:

- Opcode_1 set to 0
- CRn set to c2
- CRm set to c0

Copyright © 2004-2009 ARM Limited. All rights reserved. 3-58
Non-Confidential, Unrestricted Access

System Control Coprocessor

- Opcode_2 set to 0.

For example:

```
MRC p15, 0, <Rd>, c2, c0, 0    ; Read Translation Table Base Register 0
MCR p15, 0, <Rd>, c2, c0, 0    ; Write Translation Table Base Register 0
```

RPi # cat mmu_peek_lkm.c

```
/*
 * mmu_peek_lkm
 * Meant to be run on the Raspberry Pi.
 *
 * Dump some ARM CP15 (MMU/TLB control) registers.
 * ARM Inline assembler help: http://www.ethernut.de/en/documents/arm-inline-asm.html
 *
 * See ARM1176JZF-S TRM pg 3-2 onwards for ARM system control coprocessor (CP15)
 * control/config registers description & usage.
 *
 * General syntax of MRC : Move to ARM Register from Coprocessor [read] :
 *  MRC    <copro>, <op1>, Rd, Cn, Cm, <op2>
 *
 *  <copro> is usually CP15 (written as 'p15')
 *  Rd = dest register (%0 => r0)
 *
 * (c) Kaiwan NB, kaiwanTECH
 */
#include <linux/init.h>
#include <linux/module.h>

#define MODNAME  "mmu_peek_lkm"
MODULE_LICENSE("Dual BSD/GPL");

static inline u32 getreg_cp15_c0(void)
{
  u32 rv;
  /* read Main ID register, CP15:c0 [see ARM1176jzf-s TRM, pg 3-20] reg into r0 */
  //    MRC <copro>, <op1>, Rd, Cn, Cm, <op2>
  asm volatile("mrc p15, 0, %0, c0, c0, 0" : "=r" (rv));
  return rv;
}

static inline u32 getreg_cp15_c1(void)
{
  u32 rv;
/* read MMU CP15:c1 control register [see ASDG, Sloss, pg 514; TRM pg 3-44] reg into r0
*/
  //    MRC <copro>, <op1>, Rd, Cn, Cm, <op2>
  asm volatile("mrc p15, 0, %0, c1, c0, 0" : "=r" (rv));
  return rv;
}

static inline u32 get_ttbr0(void)
{
  u32 rv;
/* read TTRB0 register, CP15:c2 [see ARM1176jzf-s TRM, pg 3-58] reg into r0 */
  //    MRC <copro>, <op1>, Rd, CRn, CRm, <op2>
  asm volatile("mrc p15, 0, %0, c2, c0, 0" : "=r" (rv));
  return rv;
}


static int __init mmu_peek_init(void)
{
    volatile u32 reg=0x0;

    reg = getreg_cp15_c0();
    pr_info("%s: mmu_peek: Main ID register CP15:c0=0x%08x\n",
                MODNAME, reg);
    reg = getreg_cp15_c1();
    pr_info("%s: mmu_peek: MMU control register CP15:c1=0x%08x\n",
                MODNAME, reg);
    reg = get_ttbr0();
    pr_info("%s: mmu_peek: TTBR0=0x%08x\n",
                MODNAME, reg);

    return 0; // success
}

static void __exit mmu_peek_exit(void)
{
    pr_info("%s: Done.\n", MODNAME);
}

module_init(mmu_peek_init);
module_exit(mmu_peek_exit);
```

```
RPi # insmod mmu_peek_lkm.ko
RPi # dmesg
[ 1263.815599] Done.
[ 1265.264973] mmu_peek_lkm: mmu_peek: Main ID register CP15:c0=0x410fd034
[ 1265.264996] mmu_peek_lkm: mmu_peek: MMU control register CP15:c1=0x10c5383d
[ 1265.265006] mmu_peek_lkm: mmu_peek: TTBR0=0x30f3006a
RPi #
```

Appendix A : (Older) ARM Naming Convention

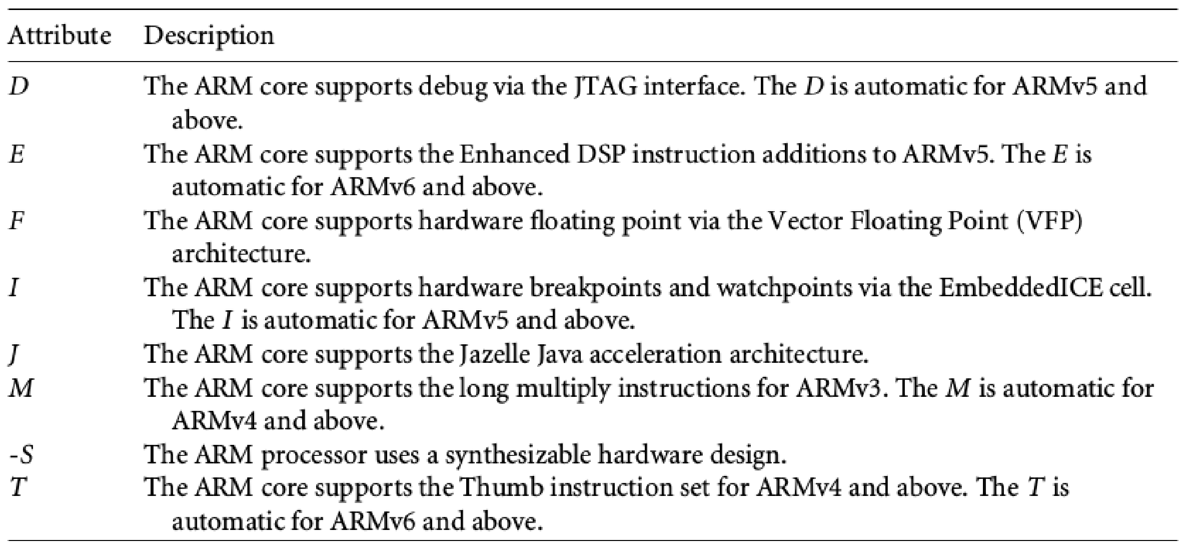
All ARM processors share a common naming convention that has evolved over time. ARM cores have the name ARM{x}{labels} where x is the number of the core and labels are letters representing extra features, described in Table C.1.

ARM processors have the name

ARM{x}{y}{z}{labels} ; where

y and **z** are numbers defining the processor cache size and memory management model.

Table C.1 Label attributes.



Attribute	Description
D	The ARM core supports debug via the JTAG interface. The *D* is automatic for ARMv5 and above.
E	The ARM core supports the Enhanced DSP instruction additions to ARMv5. The *E* is automatic for ARMv6 and above.
F	The ARM core supports hardware floating point via the Vector Floating Point (VFP) architecture.
I	The ARM core supports hardware breakpoints and watchpoints via the EmbeddedICE cell. The *I* is automatic for ARMv5 and above.
J	The ARM core supports the Jazelle Java acceleration architecture.
M	The ARM core supports the long multiply instructions for ARMv3. The *M* is automatic for ARMv4 and above.
-S	The ARM processor uses a synthesizable hardware design.
T	The ARM core supports the Thumb instruction set for ARMv4 and above. The *T* is automatic for ARMv6 and above.

Table C.2 lists the rules for ARM processor numbering.

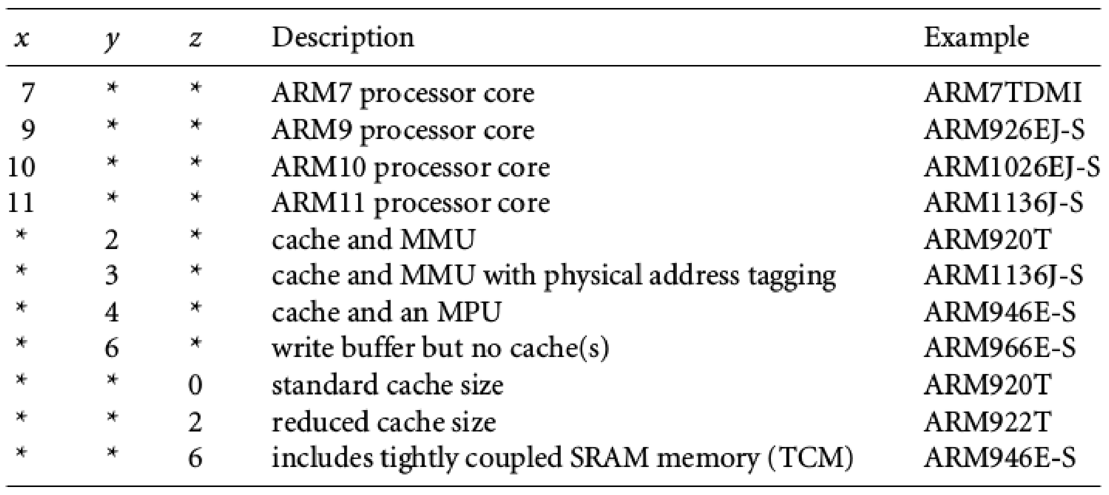


Table C.2 ARM processor numbering: ARM{x}{y}{z}.

x	y	z	Description	Example
7	*	*	ARM7 processor core	ARM7TDMI
9	*	*	ARM9 processor core	ARM926EJ-S
10	*	*	ARM10 processor core	ARM1026EJ-S
11	*	*	ARM11 processor core	ARM1136J-S
*	2	*	cache and MMU	ARM920T
*	3	*	cache and MMU with physical address tagging	ARM1136J-S
*	4	*	cache and an MPU	ARM946E-S
*	6	*	write buffer but no cache(s)	ARM966E-S
*	*	0	standard cache size	ARM920T
*	*	2	reduced cache size	ARM922T
*	*	6	includes tightly coupled SRAM memory (TCM)	ARM946E-S

The labels, or attributes, are often subsumed into the architecture version over time. For example, the T label indicates the inclusion of Thumb in ARMv4 processors. However, Thumb is included in ARMv5 and later processors, so it is not necessary to specify the T after this point.

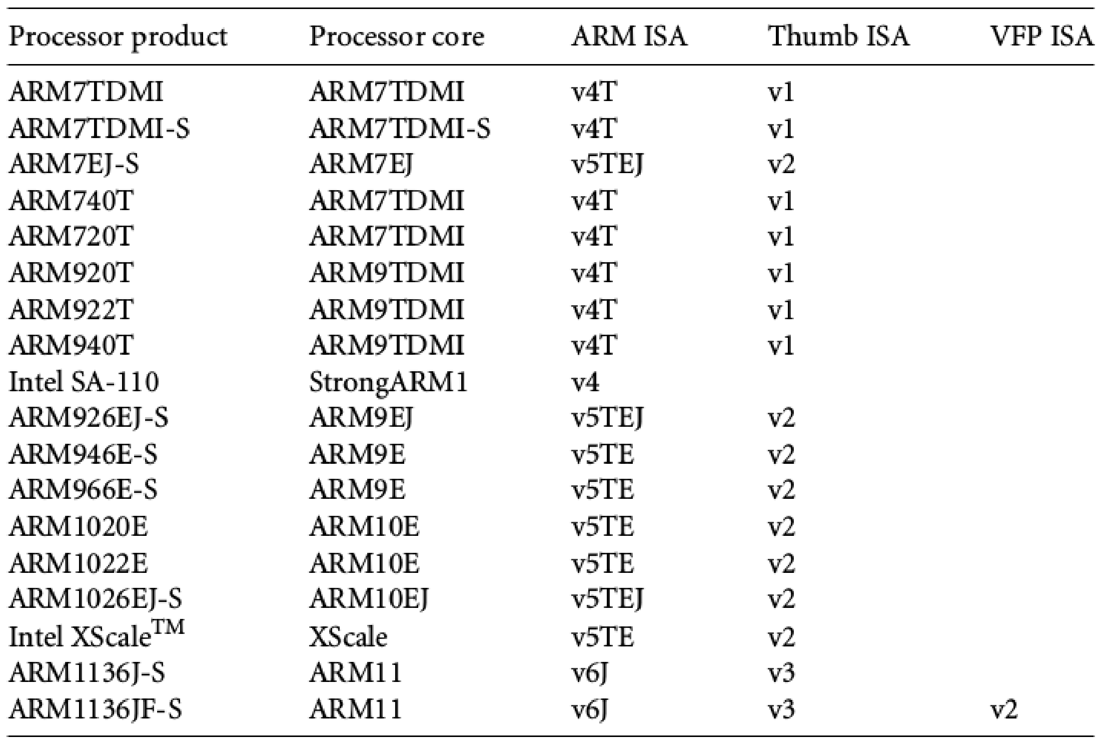


Table C.3 Processors, cores, and architecture versions.

Processor product	Processor core	ARM ISA	Thumb ISA	VFP ISA
ARM7TDMI	ARM7TDMI	v4T	v1	
ARM7TDMI-S	ARM7TDMI-S	v4T	v1	
ARM7EJ-S	ARM7EJ	v5TEJ	v2	
ARM740T	ARM7TDMI	v4T	v1	
ARM720T	ARM7TDMI	v4T	v1	
ARM920T	ARM9TDMI	v4T	v1	
ARM922T	ARM9TDMI	v4T	v1	
ARM940T	ARM9TDMI	v4T	v1	
Intel SA-110	StrongARM1	v4		
ARM926EJ-S	ARM9EJ	v5TEJ	v2	
ARM946E-S	ARM9E	v5TE	v2	
ARM966E-S	ARM9E	v5TE	v2	
ARM1020E	ARM10E	v5TE	v2	
ARM1022E	ARM10E	v5TE	v2	
ARM1026EJ-S	ARM10EJ	v5TEJ	v2	
Intel XScale^{TM}	XScale	v5TE	v2	
ARM1136J-S	ARM11	v6J	v3	
ARM1136JF-S	ARM11	v6J	v3	v2

Generic ARM Resources

Simplest Bare-metal program for ARM

Building Bare-Metal ARM Systems with GNU: Part 1 - Getting Started

Power Management (ARM) Basics

ARMology
https://marcin.juszkiewicz.com.pl/ (blog site)