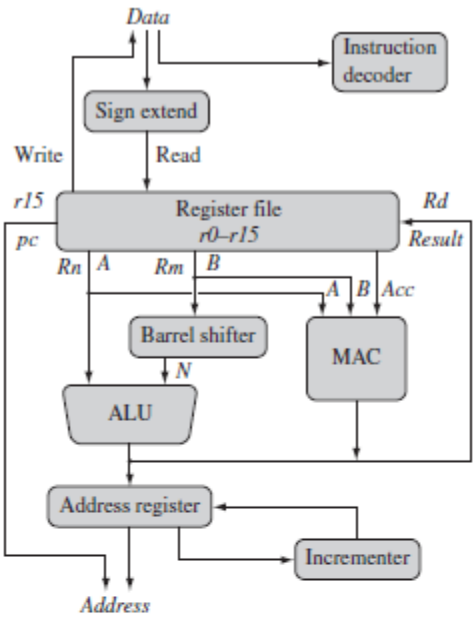
Unit 1

1. ARM Core data flow model:

The ARM processor, like all RISC processors, uses a *load-store architecture*. This means it has two instruction types for transferring data in and out of the processor: load instructions copy data from memory to registers in the core, and conversely the store



igure 2.1 ARM core dataflow model.

instructions copy data from registers to memory. There are no data processing instructions that directly manipulate data in memory. Thus, data processing is carried out solely in registers.

Data items are placed in the *register file*—a storage bank made up of 32-bit registers. Since the ARM core is a 32-bit processor, most instructions treat the registers as holding signed or unsigned 32-bit values. The sign extend hardware converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.

ARM instructions typically have two source registers, *Rn* and *Rm*, and a single result or destination register, *Rd*. Source operands are read from the register file using the internal buses *A* and *B*, respectively.

The ALU (arithmetic logic unit) or MAC (multiply-accumulate unit) takes the register values *Rn* and *Rm* from the *A* and *B* buses and computes a result. Data processing instructions write the result in *Rd* directly to the register file. Load and store instructions use the ALU to generate an address to be held in the address register and broadcast on the *Address* bus.

One important feature of the ARM is that register *Rm* alternatively can be preprocessed in the barrel shifter before it enters the ALU. Together the barrel shifter and ALU can calculate a wide range of expressions and addresses.

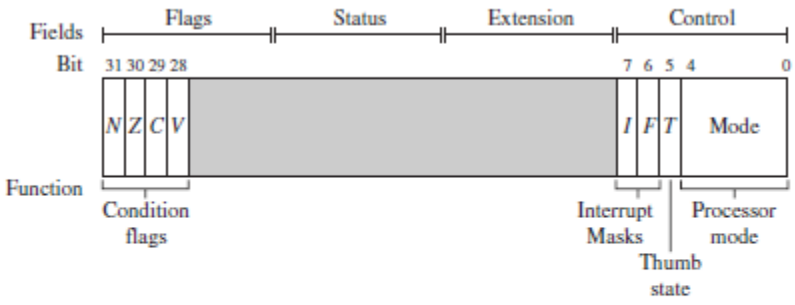
After passing through the functional units, the result in *Rd* is written back to the register file using the *Result* bus. For load and store instructions the incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location. The processor continues executing instructions until an exception or interrupt changes the normal execution flow.

2. Bit structure of CPSR:

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



2.3 A generic program status register (*psr*).

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.

For a full description of the *cpsr*, refer to Appendix B.

3.What is memory management. The types of memory management in ARM Processor.

MEMORY MANAGEMENT

Embedded systems often use multiple memory devices. It is usually necessary to have a method to help organize these devices and protect the system from applications trying to make inappropriate accesses to hardware. This is achieved with the assistance of memory management hardware.

ARM cores have three different types of memory management hardware—no extensions providing no protection, a memory protection unit (MPU) providing limited protection, and a memory management unit (MMU) providing full protection:

- *Nonprotected memory* is fixed and provides very little flexibility. It is normally used for small, simple embedded systems that require no protection from rogue applications.

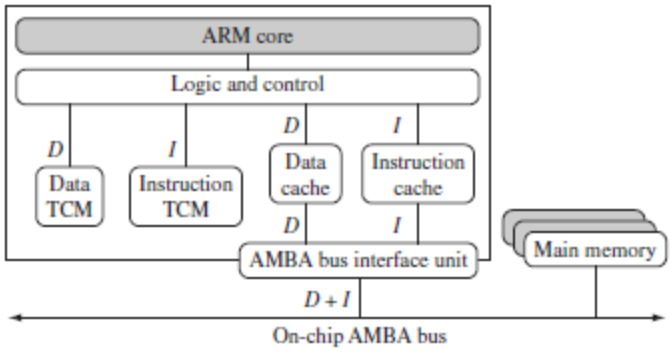


Figure 2.15 A simplified Harvard architecture with caches and TCMs.

- *MPU*s employ a simple system that uses a limited number of memory regions. These regions are controlled with a set of special coprocessor registers, and each region is defined with specific access permissions. This type of memory management is used for systems that require memory protection but don't have a complex memory map. The MPU is explained in Chapter 13.
- *MMU*s are the most comprehensive memory management hardware available on the ARM. The MMU uses a set of translation tables to provide fine-grained control over memory. These tables are stored in main memory and provide a virtual-to-physical address map as well as access permissions. MMUs are designed for more sophisticated platform operating systems that support multitasking. The MMU is explained in Chapter 14.

4. ARM processor nomenclature

ARM uses the nomenclature shown in Figure 2.16 to describe the processor implementations. The letters and numbers after the word "ARM" indicate the features a processor

ARM{x}{y}{z}{T}{D}{M}{I}{E}{J}{F}{-S}

x—family
y—memory management/protection unit
z—cache
T—Thumb 16-bit decoder
D—JTAG debug
M—fast multiplier
I—EmbeddedICE macrocell
E—enhanced instructions (assumes TDMI)
J—Jazelle
F—vector floating-point unit
S—synthesizible version

may have. In the future the number and letter combinations may change as more features are added. Note the nomenclature does not include the architecture revision information.

There are a few additional points to make about the ARM nomenclature:

- All ARM cores after the ARM7TDMI include the *TDMI* features even though they may not include those letters after the "ARM" label.
- The processor *family* is a group of processor implementations that share the same hardware characteristics. For example, the ARM7TDMI, ARM740T, and ARM720T all share the same family characteristics and belong to the ARM7 family.
- *JTAG* is described by IEEE 1149.1 Standard Test Access Port and boundary scan architecture. It is a serial protocol used by ARM to send and receive debug information between the processor core and test equipment.
- *EmbeddedICE macrocell* is the debug hardware built into the processor that allows breakpoints and watchpoints to be set.
- *Synthesizable* means that the processor core is supplied as source code that can be compiled into a form easily used by EDA tools.

5. Evolution of ARM processor families:
6. Function of flags in cpsr:

N, bit [31]

Negative condition flag. Set to bit[31] of the result of the last flag-setting instruction. If the result is regarded as a two's complement signed integer, then N is set to 1 if the result was negative, and N is set to 0 if the result was positive or zero.

Z, bit [30]

Zero condition flag. Set to 1 if the result of the last flag-setting instruction was zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

C, bit [29]

Carry condition flag. Set to 1 if the last flag-setting instruction resulted in a carry condition, for example an unsigned overflow on an addition.

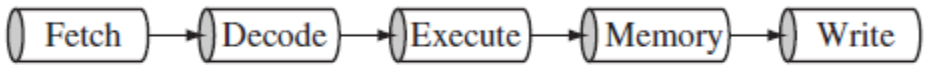
V, bit [28]

Overflow condition flag. Set to 1 if the last flag-setting instruction resulted in an overflow condition, for example a signed overflow on an addition.

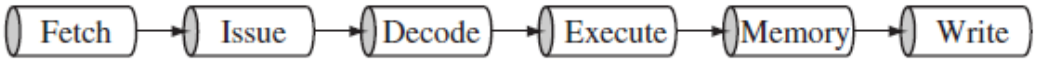
7. Explain 5-stage and 6-stage pipelining.

The five stages of pipeline are:

1. **Fetch** – The instruction is fetched from the memory and stored in the instruction register.
2. **Decode** – The instruction is moved to the decoder which decodes the instruction. It activates the appropriate control signals and takes the necessary steps for the the next execution stage.
3. **Execute** – An operand is shifted and the ALU result generated. If the instruction is a load or store, the memory address is computed in the ALU.
4. **Buffer/Data** – Data memory is accessed if required. Otherwise the ALU result is simply buffered for one cycle.
5. **Write back** – The result generated by the instruction are written back to the register file, including any data loaded from memory.



ARM9 five-stage pipeline.



ARM10 six-stage pipeline.

8. Define processor mode. List different processor modes of ARM core .

Processor Modes

The processor mode determines which registers are active and the access rights to the *cpsr* register itself. Each processor mode is either privileged or nonprivileged: A privileged mode allows full read-write access to the *cpsr*. Conversely, a nonprivileged mode only allows read access to the control field in the *cpsr* but still allows read-write access to the condition flags.

There are seven processor modes in total: six privileged modes (*abort*, *fast interrupt request*, *interrupt request*, *supervisor*, *system*, and *undefined*) and one nonprivileged mode (*user*).

The processor enters *abort* mode when there is a failed attempt to access memory. *Fast interrupt request* and *interrupt request* modes correspond to the two interrupt levels available on the ARM processor. *Supervisor* mode is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in. *System* mode is a special version of *user* mode that allows full read-write access to the *cpsr*. *Undefined* mode is used when the processor encounters an instruction that is undefined or not supported by the implementation. *User* mode is used for programs and applications.

Unit 2

1. Define Barrel Shifter. Explain the Barrel Shifter operations in ARM Processor.

BARREL SHIFTER

In Example 3.1 we showed a `MOV` instruction where *N* is a simple register. But *N* can be more than just a register or immediate value; it can also be a register *Rm* that has been preprocessed by the barrel shifter prior to being used by a data processing instruction.

Data processing instructions are processed within the arithmetic logic unit (ALU). A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU. This shift increases the power and flexibility of many data processing operations.

There are data processing instructions that do not use the barrel shift, for example, the `MUL` (multiply), `CLZ` (count leading zeros), and `QADD` (signed saturated 32-bit add) instructions.

Pre-processing or shift occurs within the cycle time of the instruction. This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.

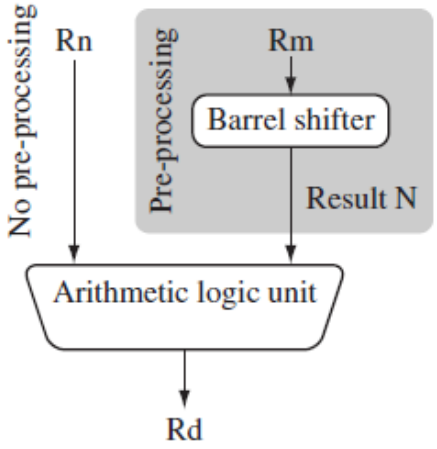


Table 3.2 Barrel shifter operations.

Mnemonic	Description	Shift	Result	Shift amount y
`LSL`	logical shift left	*x*`LSL` *y*	$x \ll y$	#0–31 or *Rs*
`LSR`	logical shift right	*x*`LSR` *y*	$(\text{unsigned})x \gg y$	#1–32 or *Rs*
`ASR`	arithmetic right shift	*x*`ASR` *y*	$(\text{signed})x \gg y$	#1–32 or *Rs*
`ROR`	rotate right	*x*`ROR` *y*	$((\text{unsigned})x \gg y) \mid (x \ll (32 - y))$	#1–31 or *Rs*
`RRX`	rotate right extended	*x*`RRX`	$(c \text{ flag} \ll 31) \mid ((\text{unsigned})x \gg 1)$	none

2. Discuss the syntax of Multiple and Single Register Load-Store transfer Instructions with different addressing modes.

SINGLE-REGISTER LOAD-STORE ADDRESSING MODES

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the indexing methods: preindex with writeback, preindex, and postindex (see Table 3.4).

Index methods.

Index method	Data	Base address register	Example
Preindex with writeback	*mem[base + offset]*	*base + offset*	`LDR r0,[r1,#4]!`
Preindex	*mem[base + offset]*	*not updated*	`LDR r0,[r1,#4]`
Postindex	*mem[base]*	*base + offset*	`LDR r0,[r1],#4`

Note: ! indicates that the instruction writes the calculated address back to the base address register.

3. What is the differences between the ARM and Thumb Instruction set.

ARM and Thumb instruction set features.

	ARM (*cpsr T* = 0)	Thumb (*cpsr T* = 1)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution[a]	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers +*pc*	8 general-purpose registers +7 high registers +*pc*

[a] See Section 3.2.6

4.Explain arithmetic and Logical data processing instructions.

`Syntax: <instruction>{<cond>}{S} Rd, Rn, N`

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

Logical instructions perform bitwise logical operations on the two source registers.

Syntax: <instruction>{<cond>}{S} Rd, Rn, N

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

5. Write the Syntax for Load-Store Multiple registers. Discuss the different Addressing modes for the Load-Store multiple registers.

MULTIPLE-REGISTER TRANSFER

Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction. The transfer occurs from a base address register *Rn* pointing into memory. Multiple-register transfer instructions are more efficient from single-register transfers for moving blocks of data around memory and saving and restoring context and stacks.

Addressing mode for load-store multiple instructions.

Addressing mode	Description	Start address	End address	*Rn*!
IA	increment after	Rn	$Rn + 4^*N - 4$	$Rn + 4^*N$
IB	increment before	$Rn + 4$	$Rn + 4^*N$	$Rn + 4^*N$
DA	decrement after	$Rn - 4^*N + 4$	Rn	$Rn - 4^*N$
DB	decrement before	$Rn - 4^*N$	$Rn - 4$	$Rn - 4^*N$

6. Write the code to move the contents of one-16 bit variable to another 16-bit Variable.

7. Explain the Stack operations. Briefly discuss the addressing modes of Stack operations.

3.3.3.1 Stack Operations

The ARM architecture uses the load-store multiple instructions to carry out stack operations. The *pop* operation (removing data from a stack) uses a load multiple instruction; similarly, the *push* operation (placing data onto the stack) uses a store multiple instruction.

When using a stack you have to decide whether the stack will grow up or down in memory. A stack is either *ascending* (A) or *descending* (D). Ascending stacks grow towards higher memory addresses; in contrast, descending stacks grow towards lower memory addresses.

When you use a *full stack* (F), the stack pointer *sp* points to an address that is the last used or full location (i.e., *sp* points to the last item on the stack). In contrast, if you use an *empty stack* (E) the *sp* points to an address that is the first unused or empty location (i.e., it points after the last item on the stack).

There are a number of load-store multiple addressing mode aliases available to support stack operations (see Table 3.11). Next to the *pop* column is the actual load multiple instruction equivalent. For example, a full ascending stack would have the notation FA appended to the load multiple instruction—LDMFA. This would be translated into an LDMDA instruction.

ARM has specified an ARM-Thumb Procedure Call Standard (ATPCS) that defines how routines are called and how registers are allocated. In the ATPCS, stacks are defined as being full descending stacks. Thus, the LDMFD and STMFD instructions provide the pop and push functions, respectively.

Addressing methods for stack operations.

Addressing mode	Description	Pop	= LDM	Push	= STM
FA	full ascending	LDMFA	LDMDA	STMFA	STMIB
FD	full descending	LDMFD	LDMIA	STMFD	STMDB
EA	empty ascending	LDMEA	LDMDB	STMEA	STMIA
ED	empty descending	LDMED	LDMIB	STMED	STMDA

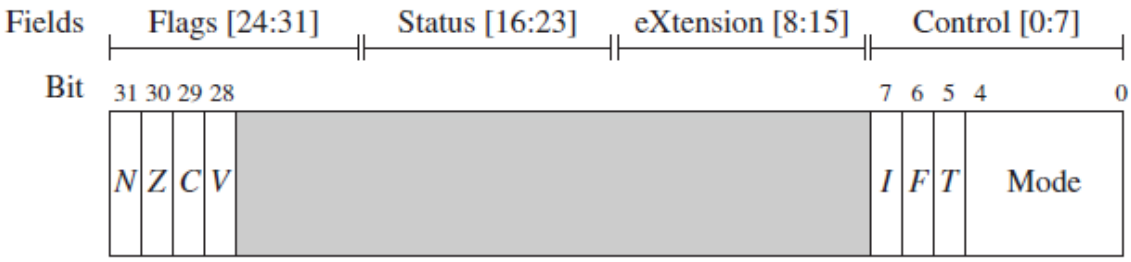
8. Write the Syntax of Program Status Register Instruction

PROGRAM STATUS REGISTER INSTRUCTIONS

The ARM instruction set provides two instructions to directly control a program status register (*psr*). The MRS instruction transfers the contents of either the *cpsr* or *spsr* into a register; in the reverse direction, the MSR instruction transfers the contents of a register into the *cpsr* or *spsr*. Together these instructions are used to read and write the *cpsr* and *spsr*.

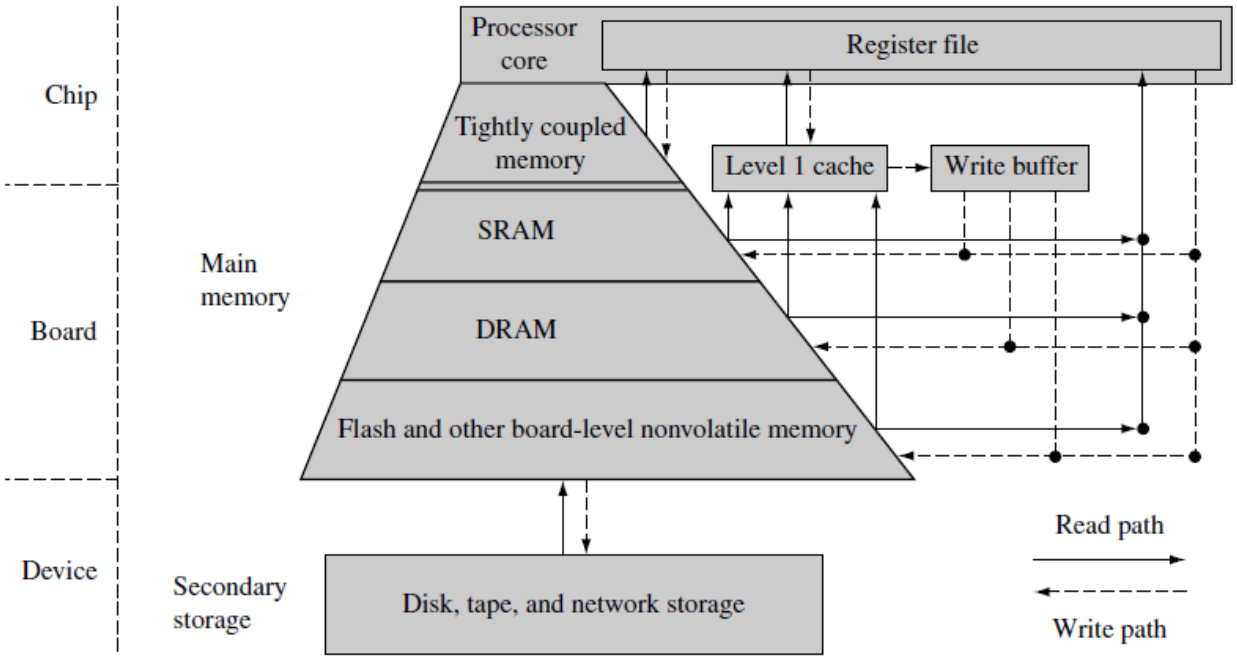
In the syntax you can see a label called *fields*. This can be any combination of control (*c*), extension (*x*), status (*s*), and flags (*f*). These fields relate to particular byte regions in a *psr*, as shown in Figure 3.9.

```
Syntax: MRS{<cond>} Rd,<cpsr|spsr>
        MSR{<cond>} <cpsr|spsr>_<fields>,Rm
        MSR{<cond>} <cpsr|spsr>_<fields>,#immediate
```



Unit 3

1. With the neat Diagram, Explain the Memory Hierarchy and Cache Memory.



A cache may be incorporated between any level in the hierarchy where there is a significant access time difference between memory components. A cache can improve system performance whenever such a difference exists. A cache memory system takes information stored in a lower level of the hierarchy and temporarily moves it to a higher level.

Figure 12.1 includes a level 1 (L1) cache and write buffer. The L1 cache is an array of high-speed, on-chip memory that temporarily holds code and data from a slower level. A cache holds this information to decrease the time required to access both instructions and data. The write buffer is a very small FIFO buffer that supports writes to main memory from the cache.

Not shown in the figure is a level 2 (L2) cache. An L2 cache is located between the L1 cache and slower memory. The L1 and L2 caches are also known as the *primary* and *secondary* caches.

2. What is Memory management Unit. Discuss how virtual memory works in MMU.

One of the key services provided by an MMU is the ability to manage tasks as independent programs running in their own private memory space. A task written to run under the control of an operating system with an MMU does not need to know the memory requirements of unrelated tasks. This simplifies the design requirements of individual tasks running under the control of an operating system.

The MMU simplifies the programming of application tasks because it provides the resources needed to enable *virtual memory*—an additional memory space that is independent of the physical memory attached to the system. The MMU acts as a translator, which converts the addresses of programs and data that are compiled to run in virtual memory to the actual physical addresses where the programs are stored in physical main memory. This translation process allows programs to run with the same virtual addresses while being held in different locations in physical memory.

How Virtual Memory Works

In Chapter 13 we introduced the MPU and showed a multitasking embedded system that compiled and ran each task at distinctly different, fixed address areas in main memory. Each task ran in only one of the process regions, and none of the tasks could have overlapping addresses in main memory. To run a task, a protection region was placed over the fixed address program to enable access to an area of memory defined by the region. The placement of the protection region allowed the task to execute while the other tasks were protected.

In an MMU, tasks can run even if they are compiled and linked to run in regions with overlapping addresses in main memory. The support for virtual memory in the MMU enables the construction of an embedded system that has multiple virtual memory maps and a single physical memory map. Each task is provided its own virtual memory map for the purpose of compiling and linking the code and data, which make up the task. A kernel layer then manages the placement of the multiple tasks in physical memory so they have a distinct location in physical memory that is different from the virtual location it is designed to run in.

To permit tasks to have their own virtual memory map, the MMU hardware performs *address relocation*, translating the memory address output by the processor core before it reaches main memory. The easiest way to understand the translation process is to imagine a relocation register located in the MMU between the core and main memory.

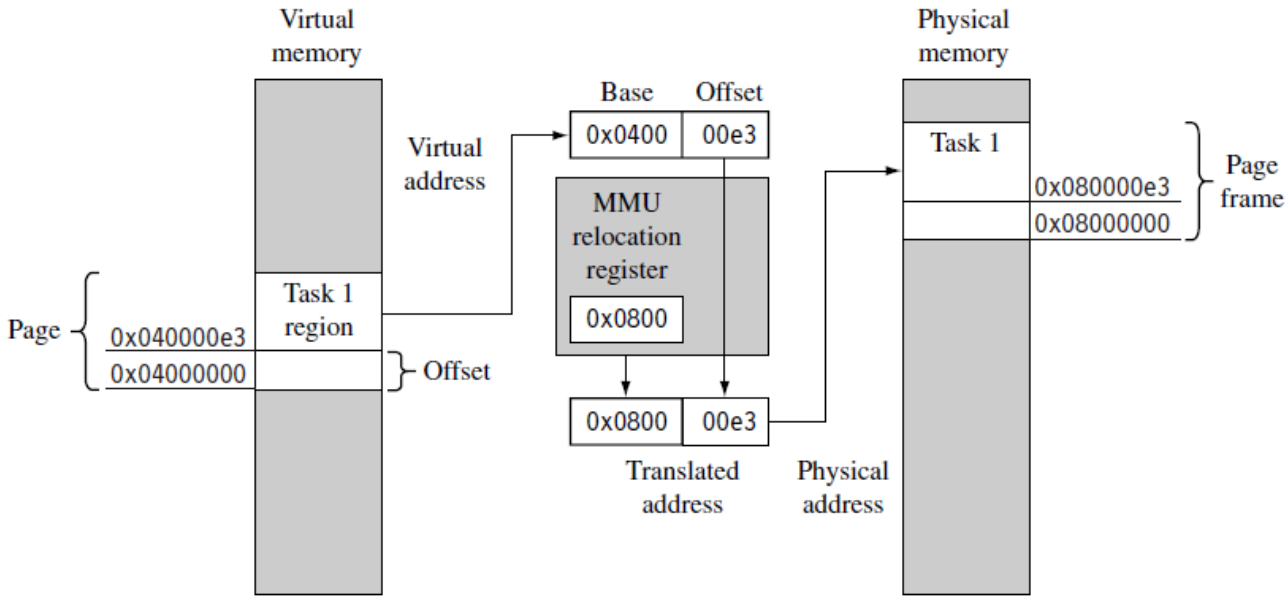


Figure 14.1 Mapping a task in virtual memory to physical memory using a relocation register.

When the processor core generates a virtual address, the MMU takes the upper bits of the virtual address and replaces them with the contents of the relocation register to create a physical address, shown in Figure 14.1

The lower portion of the virtual address is an offset that translates to a specific address in physical memory. The range of addresses that can be translated using this method is limited by the maximum size of this offset portion of the virtual address.

Figure 14.1 shows an example of a task compiled to run at a starting address of 0x4000000 in virtual memory. The relocation register translates the virtual addresses of Task 1 to physical addresses starting at 0x8000000.

A second task compiled to run at the same virtual address, in this case 0x400000, can be placed in physical memory at any other multiple of 0x10000 (64 KB) and mapped to 0x400000 simply by changing the value in the relocation register.

A single relocation register can only translate a single area of memory, which is set by the number of bits in the offset portion of the virtual address. This area of virtual memory is known as a *page*. The area of physical memory pointed to by the translation process is known as a *page frame*.

The set of relocation registers that temporarily store the translations in an ARM MMU are really a fully associative cache of 64 relocation registers. This cache is known as a Translation Lookaside Buffer (TLB). The TLB caches translations of recently accessed pages.

In addition to having relocation registers, the MMU uses tables in main memory to store the data describing the virtual memory maps used in the system. These tables of translation data are known as *page tables*. An entry in a page table represents all the information needed to translate a page in virtual memory to a page frame in physical memory.

A *page table entry* (PTE) in a page table contains the following information about a virtual page: the physical base address used to translate the virtual page to the physical page frame, the access permission assigned to the page, and the cache and write buffer configuration for the page. If you refer to Table 14.1, you can see that most of the region configuration data in an MPU is now held in a page table entry. This means access permission and cache and write buffer behavior are controlled at a granularity of the page size, which provides finer control over the use of memory. Regions in an MMU are created in software by grouping blocks of virtual pages in memory.

3. How the main Memory maps to a direct mapped cache and explain the basic operation of a Cache Controller.

Basic Operation of a Cache Controller

The *cache controller* is hardware that copies code or data from main memory to cache memory automatically. It performs this task automatically to conceal cache operation from the software it supports. Thus, the same application software can run unaltered on systems with and without a cache.

The cache controller intercepts read and write memory requests before passing them on to the memory controller. It processes a request by dividing the address of the request into three fields, the *tag* field, the *set index* field, and the *data index* field. The three bit fields are shown in Figure 12.4.

First, the controller uses the set index portion of the address to locate the cache line within the cache memory that might hold the requested code or data. This cache line contains the cache-tag and status bits, which the controller uses to determine the actual data stored there.

The controller then checks the valid bit to determine if the cache line is active, and compares the cache-tag to the tag field of the requested address. If both the status check and comparison succeed, it is a cache *hit*. If either the status check or comparison fails, it is a cache *miss*.

On a cache miss, the controller copies an entire cache line from main memory to cache memory and provides the requested code or data to the processor. The copying of a cache line from main memory to cache memory is known as a *cache line fill*.

On a cache hit, the controller supplies the code or data directly from cache memory to the processor. To do this it moves to the next step, which is to use the data index field of the address request to select the actual code or data in the cache line and provide it to the processor.

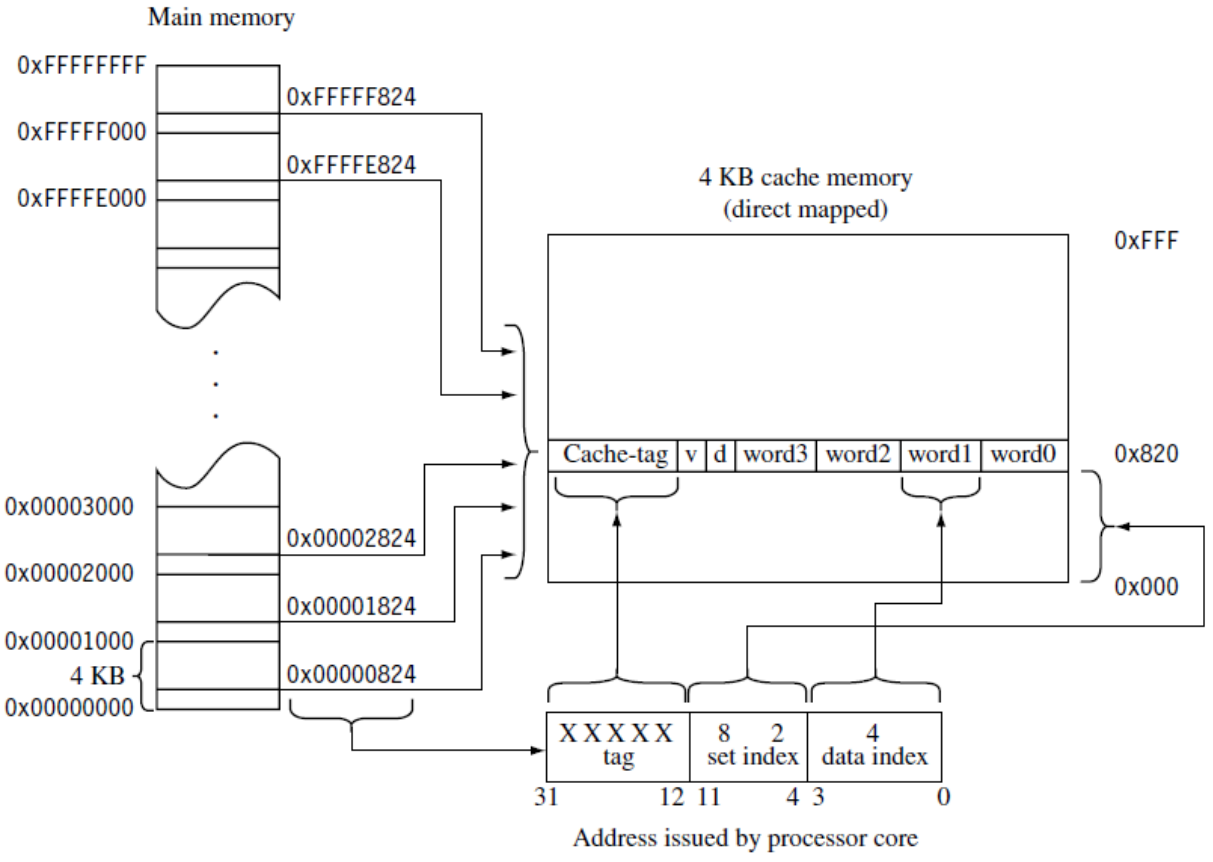


Figure 12.5 How main memory maps to a direct-mapped cache.

4. What is Memory Protected Units? Explain the different types of ARM cores that contains the MPU.

than cooperatively enforced software routines.

ARM provides several processors equipped with hardware that actively protects system resources, either through a memory protection unit (MPU) or a memory management unit (MMU). A processor core with an MPU, the subject of this chapter, provides hardware protection over several software-designated regions. A processor core with an MMU, the subject of the next chapter, provides hardware protection and adds a virtual memory capability.

In a protected system, there are two major classes of resource that need monitoring: the memory system and peripheral devices. Since ARM peripherals are generally memory mapped, the MPU uses the same method to protect both resources.

An ARM MPU uses regions to manage system protection. A *region* is a set of attributes associated with an area of memory. The processor core holds these attributes in several CP15 registers and identifies each region by a number, which ranges between zero and seven.

Summary of ARM cores with protection units.

ARM core	Number of regions	Separate instruction and data regions	Separate configuration of instruction and data regions
ARM740T	8	no	no
ARM940T	16	yes	yes
ARM946E-S	8	no	yes
ARM1026EJ-S	8	no	yes

There are currently four ARM cores that contain an MPU; the ARM740T, ARM940T, ARM946E-S, and the ARM1026EJ-S. The ARM740T, ARM946E-S, and ARM1026EJ-S each contain 8 protection regions; the ARM940T contains 16 (see Table 13.1).

The ARM740T, ARM946E-S, and ARM1026EJ-S have *unified* instruction and data regions—the data region and instruction region are defined using the same register that sets the size and starting address. The memory access permission and cache policies are configured independently for instruction and data access in the ARM946E-S and ARM1026EJ-S cores; in the ARM740T the same access permission and cache policies are assigned to both instruction and data memory. Regions are independent of whether the core has a Von Neumann or Harvard architecture. Each region is referenced by an identifying number between zero and seven.

Because the ARM940T has separate regions to control instruction and data memory, the core can maintain different region sizes and starting addresses for instruction and data regions. The separation of instruction and data regions results in eight additional regions in this cached core. Although the identifying region numbers in an ARM940T still range from zero to seven, each region number has a pair of regions, one data region and one instruction region.

There are several rules that govern regions:

1. Regions can overlap other regions.
2. Regions are assigned a priority number that is independent of the privilege assigned to the region.
3. When regions overlap, the attributes of the region with the highest priority number take precedence over the other regions. The priority only applies over the addresses within the areas that overlap.
4. A region's starting address must be a multiple of its size.
5. A region's size can be any power of two between 4 KB and 4 GB—in other words, any of the following values: 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, . . . , 2 GB, 4 GB.
6. Accessing an area of main memory outside of a defined region results in an abort. The MPU generates a prefetch abort if the core was fetching an instruction or a data abort if the memory request was for data.

5. What is Set associativity in Cache Memory. Explain in detail.

Set Associativity

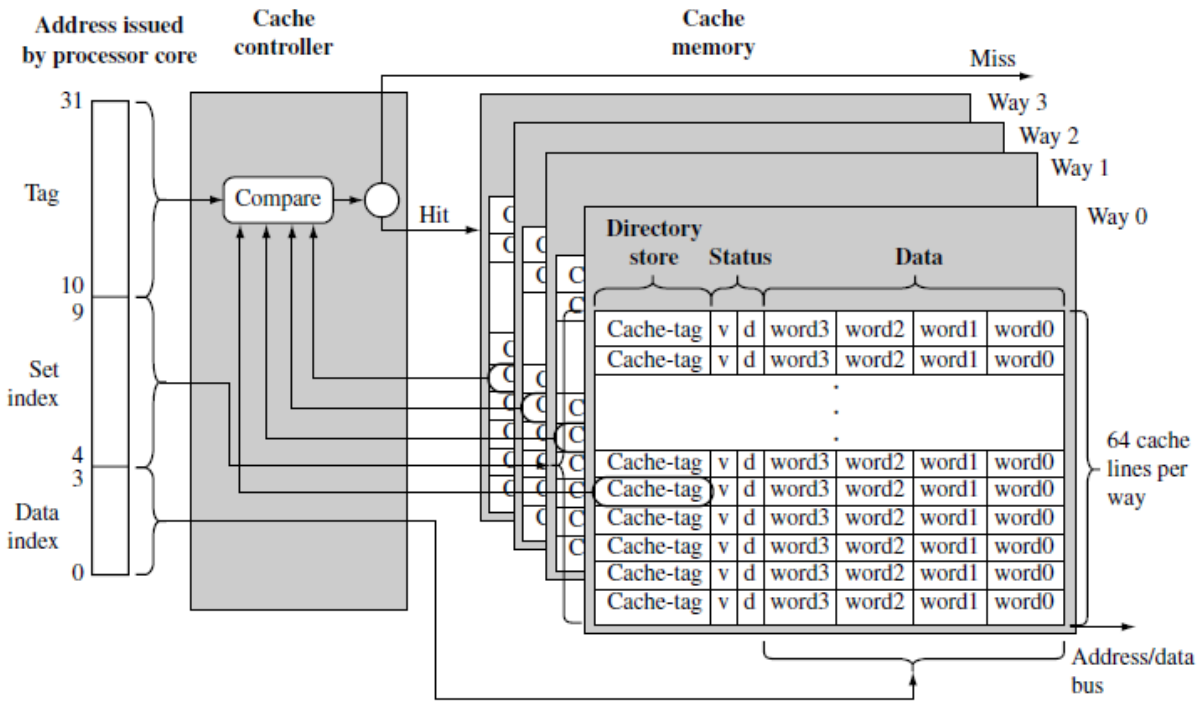
Some caches include an additional design feature to reduce the frequency of thrashing (see Figure 12.7). This structural design feature is a change that divides the cache memory into smaller equal units, called *ways*. Figure 12.7 is still a four KB cache; however, the set index now addresses more than one cache line—it points to one cache line in each way. Instead of one way of 256 lines, the cache has four ways of 64 lines. The four cache lines with the same set index are said to be in the same *set*, which is the origin of the name "set index."

The set of cache lines pointed to by the set index are *set associative*. A data or code block from main memory can be allocated to any of the four ways in a set without affecting program behavior; in other words the storing of data in cache lines within a set does not affect program execution. Two sequential blocks from main memory can be stored as cache lines in the same way or two different ways. The important thing to note is that the data or code blocks from a specific location in main memory can be stored in any cache line that is a member of a set. The placement of values within a set is exclusive to prevent the same code or data block from simultaneously occupying two cache lines in a set.

The mapping of main memory to a cache changes in a four-way set associative cache. Figure 12.8 shows the differences. Any single location in main memory now maps to four different locations in the cache. Although Figures 12.5 and 12.8 both illustrate 4 KB caches, here are some differences worth noting.

The bit field for the tag is now two bits larger, and the set index bit field is two bits smaller. This means four million main memory addresses now map to one set of four cache lines, instead of one million addresses mapping to one location.

The size of the area of main memory that maps to cache is now 1 KB instead of 4 KB. This means that the likelihood of mapping cache line data blocks to the same set is now four times higher. This is offset by the fact that a cache line is one fourth less likely to be evicted.



6. Explain the basic architecture of a Cache Memory and relationship between main memory and cache (see 3)

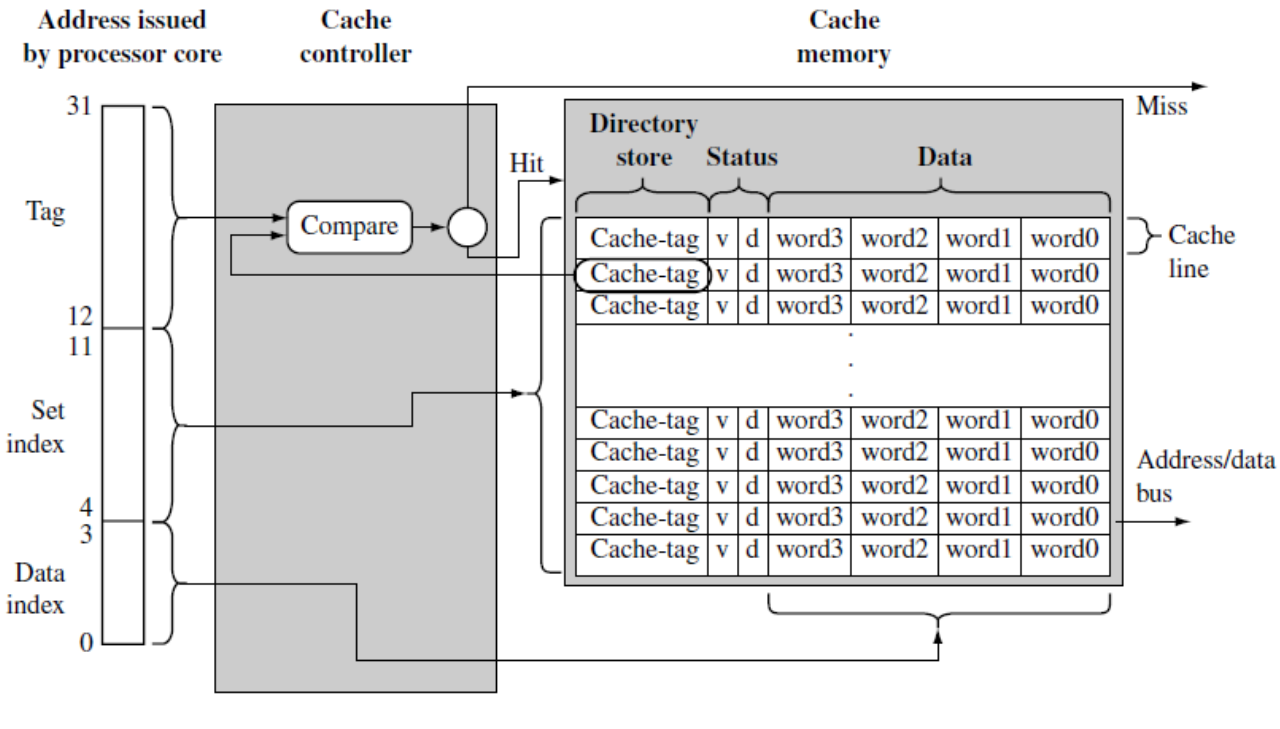
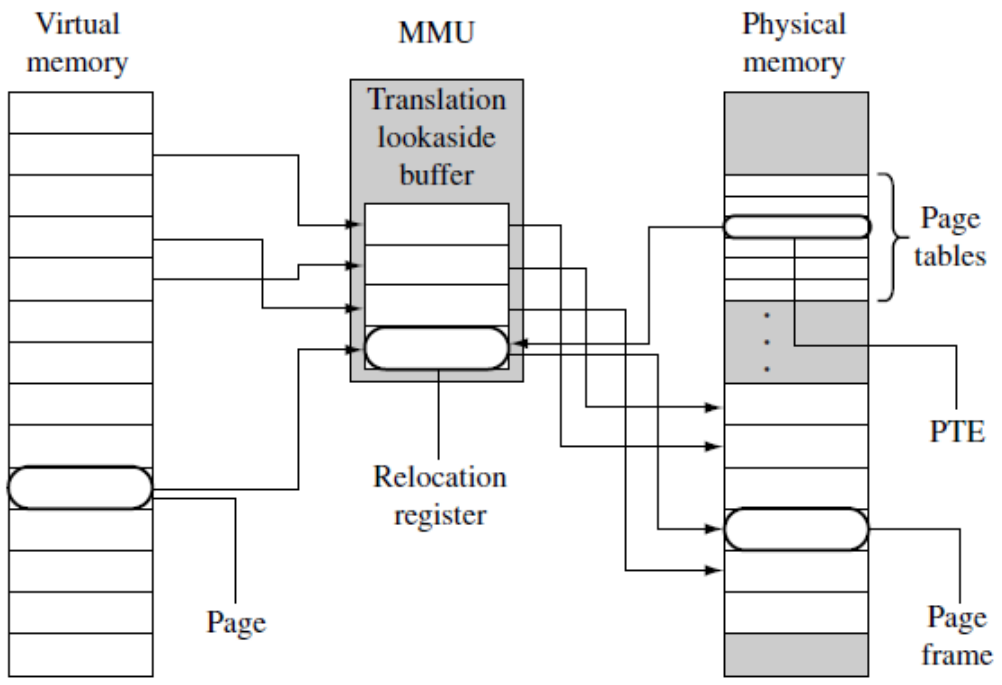


Figure 12.4 A 4 KB cache consisting of 256 cache lines of four 32-bit words.

7. What is the difference between the Logical and Physical caches.
https://www.brainkart.com/article/Logical-versus-physical-caches_7647/

8. What are the components of virtual memory system. Explain how mapping between the Virtual memory to physical memory using MMU.(see 2)



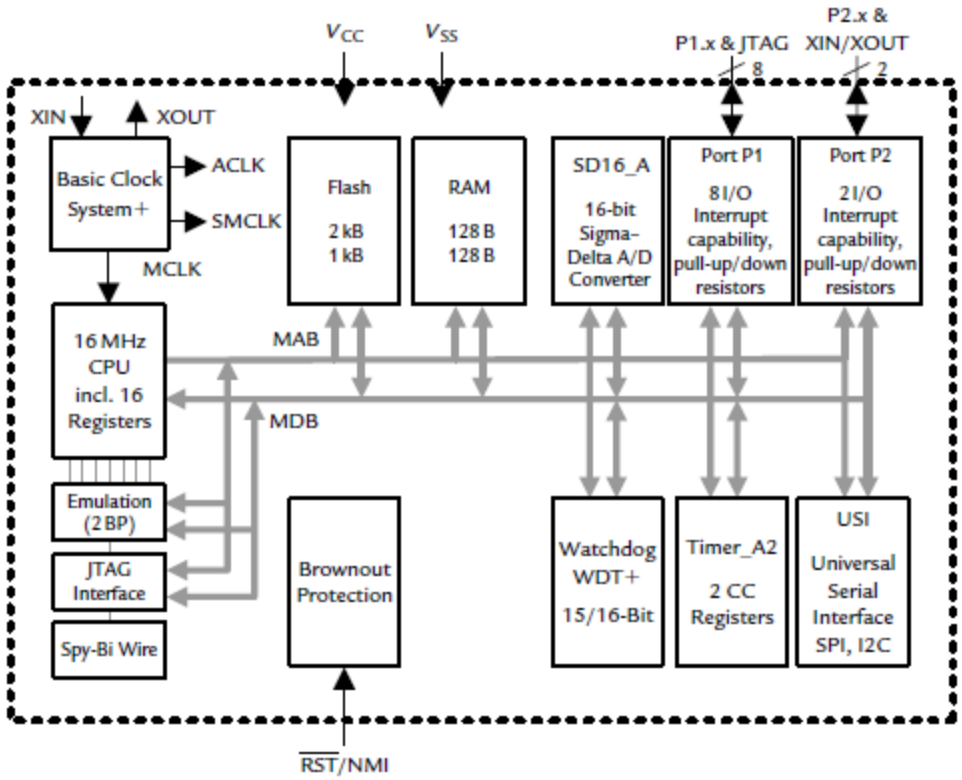
14.2 The components of a virtual memory system.

9. Explain the L1 Page table virtual to physical memory Translation.

10. What are the different Control components in the MMU. Explain the two levels of Page table.
11. Define TLB. Explain the two level table virtual to physical address Translation.
12. Write the short note on Cache policy.
13. With block diagram, explain the architecture of Nuvoton Cortex NUC140.
14. List the different features of Nuvoton Cortex NUC140 processor.

Unit 4

1. With block diagram, explain the architecture of MSP430.



- On the left is the CPU and its supporting hardware, including the clock generator. The emulation, JTAG interface and Spy-Bi-Wire are used to communicate with a desktop computer when downloading a program and for debugging.

  - The main blocks are linked by the *memory address bus* (MAB) and *memory data bus* (MDB).
  - These devices have flash memory, 1 KB in the F2003 or 2 KB in the F2013, and 128 bytes of RAM.
  - Six blocks are shown for peripheral functions (there are many more in larger devices). All MSP430s include input/output ports, Timer_A, and a watchdog timer, although the details differ. The universal serial interface (USI) and sigma–delta analog-to-digital converter (SD16_A) are particular features of this device.
  - The brownout protection comes into action if the supply voltage drops to a dangerous level. Most devices include this but not some of the MSP430x1xx family.
  - There are ground and power supply connections. Ground is labeled V_{SS} and is taken to define 0 V. The supply connection is V_{CC}. For many years, the standard for logic was $V_{CC} = +5$ V but most devices now work from lower voltages and a range of 1.8–3.6 V is specified for the F2013. The performance of the device depends on V_{CC}. For example, it is unable to program the flash memory if $V_{CC} < 2.2$ V and the maximum clock frequency of 16 MHz is available only if $V_{CC} \geq 3.3$ V.

2. Explain in detail the pin diagram of MSP430F2003 processor.

- V_{CC} and V_{SS} are the supply voltage and ground for the whole device (the analog and digital supplies are separate in the 16-pin package).
- P1.0–P1.7, P2.6, and P2.7 are for digital input and output, grouped into ports P1 and P2.
- TACLK, TA0, and TA1 are associated with Timer_A; TACLK can be used as the clock input to the timer, while TA0 and TA1 can be either inputs or outputs. These can be used on several pins because of the importance of the timer.

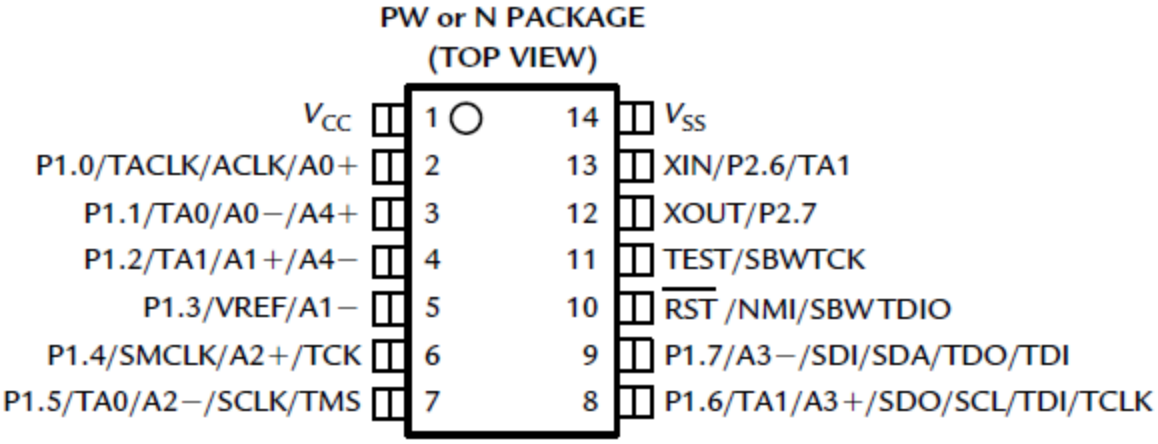


Figure 2.1: Pin-out of the MSP430F2003 and F2013, taken from the data sheet.

- A0−, A0+, and so on, up to A4±, are inputs to the analog-to-digital converter. It has four differential channels, each of which has negative and positive inputs. VREF is the reference voltage for the converter.
- ACLK and SMCLK are outputs for the microcontroller's clock signals. These can be used to supply a clock to external components or for diagnostic purposes.
- SCLK, SDO, and SCL are used for the universal serial interface, which communicates with external devices using the serial peripheral interface (SPI) or inter-integrated circuit (I^2C) bus.
- XIN and XOUT are the connections for a crystal, which can be used to provide an accurate, stable clock frequency.
- $\overline{\text{RST}}$ is an active low reset signal. *Active low* means that it remains high near V_{CC} for normal operation and is brought low near V_{SS} to reset the chip. Alternative notations to show the active low nature are _RST and /RST.
- NMI is the nonmaskable interrupt input, which allows an external signal to interrupt the normal operation of the program.
- TCK, TMS, TCLK, TDI, TDO, and TEST form the full JTAG interface, used to program and debug the device.
- SBWTDIO and SBWTCK provide the Spy-Bi-Wire interface, an alternative to the usual JTAG connection that saves pins.

3. Discuss about the register organisation of MSP430.

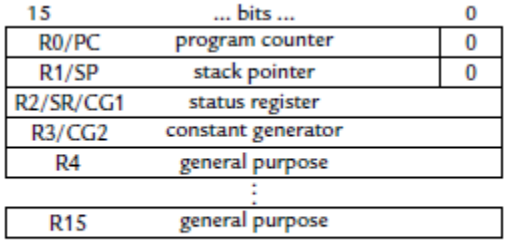


Figure 2.5: Registers in the CPU of the MSP430.

Program counter, PC: This contains the address of the next instruction to be executed, "points to" the instruction in the usual jargon. Instructions are composed of 1–3 words, which must be aligned to even addresses, so the lsb of the PC is hard-wired to 0.

Stack pointer, SP: When a subroutine is called, the CPU jumps to the subroutine, executes the code there, then returns to the instruction after the call. It must therefore keep track of the contents of the PC before jumping to the subroutine, so that it can return afterward. This is the primary purpose of the *stack*. Some processors use separate hardware for the stack but the MSP430 uses the top (high addresses) of the main RAM. The stack pointer holds the address of the most recently added word and is automatically adjusted as the stack grows downward in memory or shrinks upward.

Status register, SR: This contains a set of *flags* (single bits), whose functions fall into three categories. The most commonly used flags are C, Z, N, and V, which give information about the result of the last arithmetic or logical operation. The Z flag is set if the result was zero and cleared if it was nonzero, for instance. Decisions that affect the flow of control in the program can be made by testing these bits.

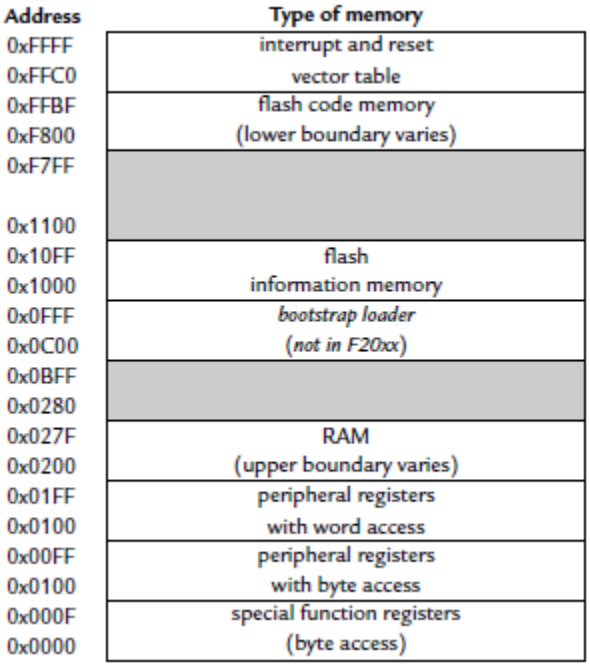
Setting the GIE bit enables maskable interrupts, which is explained in the section "Interrupts" on page 186. We do not use interrupts for the time being.

The final group of bits is CPUOFF, OSCOFF, SCG0, and SCG1, which control the mode of operation of the MCU. All systems are active when all bits are clear. Setting various combinations of these bits puts the MCU into one of its low-power modes, which is described in the section "Low-Power Modes of Operation" on page 198. We keep the CPU active for now.

Constant generator: This provides the six most frequently used values so that they need not be fetched from memory whenever they are needed. It uses both R2 and R3 to provide a range of useful values by exploiting the CPU's addressing modes. This will be explained in the section "Constant Generator and Emulated Instructions" on page 131 but the compiler or assembler handles the details automatically.

General purpose registers: The remaining 12 registers, R4–R15, are general working registers. They may be used for either data or addresses because both are 16-bit values, which simplifies the operation significantly.

4. Write a Short note on memory organisation of MSP430. 6



Address	Type of memory
0xFFFF	interrupt and reset
0xFFC0	vector table
0xFFBF	flash code memory
0xF800	(lower boundary varies)
0xF7FF	
0x1100	
0x10FF	flash
0x1000	information memory
0x0FFF	*bootstrap loader*
0x0C00	(*not in F20xx*)
0x0BFF	
0x0280	
0x027F	RAM
0x0200	(upper boundary varies)
0x01FF	peripheral registers
0x0100	with word access
0x00FF	peripheral registers
0x0100	with byte access
0x000F	special function registers
0x0000	(byte access)

Here is a brief description of each region:

Special function registers: Mostly concerned with enabling functions of some modules and enabling and signalling interrupts from peripherals.

Peripheral registers with byte access and peripheral registers with word access: Provide the main communication between the CPU and peripherals. Some must be accessed as words and others as bytes. They are grouped in this way to avoid wasting

addresses. If the bytes and words were mixed, numerous unused bytes would be needed to ensure that the words were correctly aligned on even addresses.

Random access memory: Used for variables. This always starts at address 0x0200 and the upper limit depends on the size of the RAM. The F2013 has 128 B.

Bootstrap loader: Contains a program to communicate using a standard serial protocol, often with the COM port of a PC. This can be used to program the chip but improvements in other methods of communication have made it less important than in the past, particularly for development. Details are given in the application note *Features of the MSP430 Bootstrap Loader* (slaa089). All MSP430s had a bootstrap loader until the F20xx, from which it was omitted to improve security.

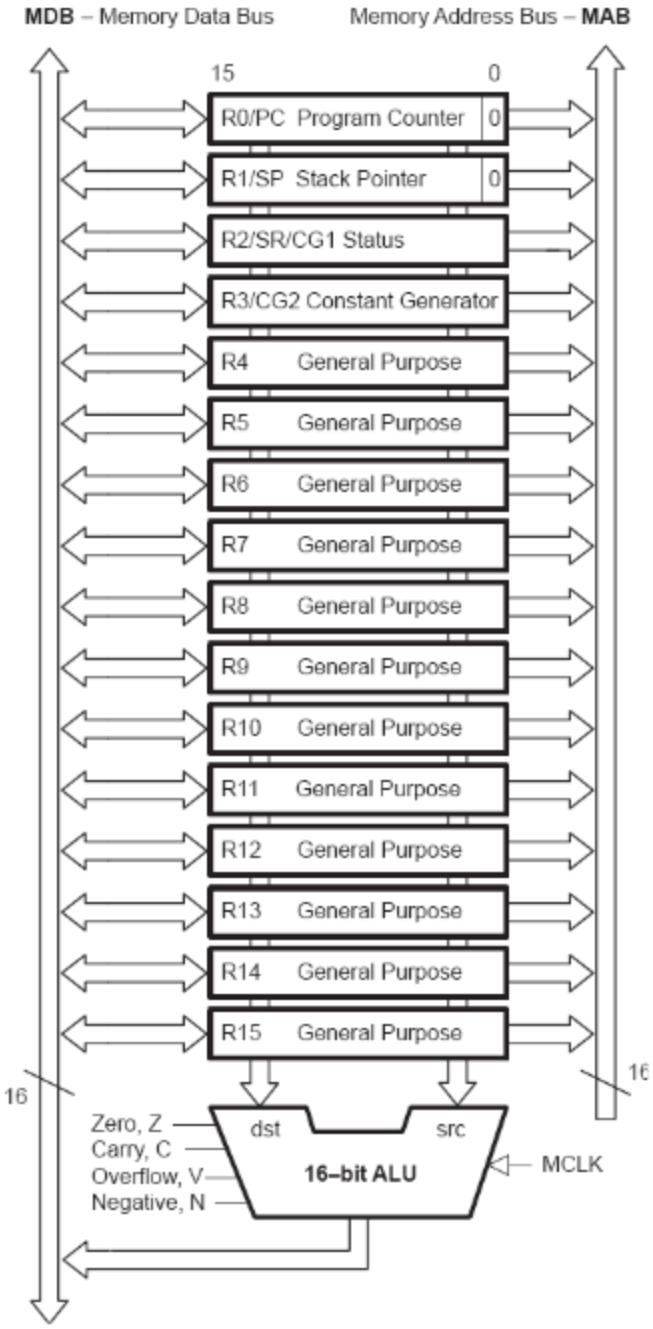
Information memory: A 256 B block of flash memory that is intended for storage of nonvolatile data. This might include serial numbers to identify equipment—an address for a network, for instance—or variables that should be retained even when power is removed. For example, a printer might remember the settings from when it was last used and keep a count of the total number of pages printed. The information memory is laid out with smaller segments of flash than the code memory, which makes it more convenient to erase and rewrite. Segment A contains factory calibration data for the DCO in the MSP430F2xx family and is protected by default.

Code memory: Holds the program, including the executable code itself and any constant data. The F2013 has 2 KB but the F2003 only 1 KB.

Interrupt and reset vectors: Used to handle "exceptions," when normal operation of the processor is interrupted or when the device is reset. This table was smaller and started at 0xFFE0 in earlier devices.

5. Briefly explain the internal block diagram of CPU in MSP430. 6

Figure 4-4. MSP430 CPU block diagram.



The MSP430X CPU extends the addressing capabilities of the MSP430 family beyond 64 kB to 1 MB. To achieve this, there are some changes to the addressing modes and two new types of instructions. One type of new instructions allows access to the entire address space, and the other is designed for address calculations.

The MSP430X CPU address bus is 20 bits, but the data bus is still 16 bits. The CPU supports 8-bit, 16-bit and 20-bit memory accesses. Despite these changes, the MSP430X CPU remains compatible with the MSP430 CPU, having a similar number of registers. A block diagram of the MSP430X CPU is shown in the figure below:

4.4.1 Arithmetic Logic Unit (ALU)

The MSP430 CPU includes an arithmetic logic unit (ALU) that handles addition, subtraction, comparison and logical (AND, OR, XOR) operations. ALU operations can affect the overflow, zero, negative, and carry flags in the status register.

4.4.2 MSP430 CPU registers

The CPU incorporates sixteen 16-bit registers:

- Four registers (R0, R1, R2 and R3) have dedicated functions;
- There are 12 working registers (R4 to R15) for general use.

R0: Program Counter (PC)

The 16-bit Program Counter (PC/R0) points to the next instruction to be read from memory and executed by the CPU. The Program counter is implemented by the number of bytes used by the instruction (2, 4, or 6 bytes, always even). It is important to remember that the PC is aligned at even addresses, because the instructions are 16 bits, even though the individual memory addresses contain 8-bit values.

R1: Stack Pointer (SP)

The Stack Pointer (SP/R1) is located in R1.

1st: stack can be used by user to store data for later use (instructions: store by PUSH, retrieve by POP);

2nd: stack can be used by user or by compiler for subroutine parameters (PUSH, POP in calling routine; addressed via offset calculation on stack pointer (SP) in called subroutine);

3rd: used by subroutine calls to store the program counter value for return at subroutine's end (RET);

4th: used by interrupt - system stores the actual PC value first, then the actual status register content (on top of stack) on return from interrupt (RETI) the system get the same status as just before the interrupt happened (as long as none has changed the value on TOS) and the same program counter value from stack.

R2: Status Register (SR)

The Status Register (SR/R2) stores the state and control bits. The system flags are changed automatically by the CPU depending on the result of an operation in a register. The reserved bits of the SR are used to support the constants generator. See the device-specific data sheets for more details.

R2/R3: Constant Generator Registers (CG1/CG2)

Depending of the source-register addressing modes (As) value, six commonly used constants can be generated without a code word or code memory access to retrieve them.

This is a very powerful feature, which allows the implementation of emulated instructions, for example, instead of implementing a core instruction for an increment, the constant generator is used.

R4 - R15: General–Purpose Registers

These general-purpose registers are used to store data values, address pointers, or index values and can be accessed with byte or word instructions.

6. Write in detail about the Status register and the functions of Flages MSP430. 8

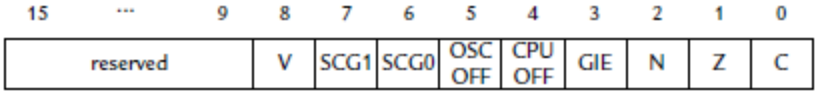


Figure 5.3: Individual bits in the status register.

5.1.3 Status Register (SR)

This contains a set of *flags* (single bits) shown in Figure 5.3, whose functions fall into three categories. The reserved bits are not used in the MSP430 but some have been taken up in the MSP430X to extend the length of addresses. The status register is known as the condition code register in some processors.

Result of Arithmetic or Logic Operations

The C, Z, N, and V bits are affected by many of the operations performed by the ALU, but not all—see the section "Instruction Set" on page 132 for details:

- The main function of the *carry* bit C is to flag that the result of an arithmetic operation is too large to fit in the space allocated. In other words, an overflow occurred. Here is an example with bytes for simplicity. The hexadecimal sum $0x75 + 0xC7 = 0x13C$, where the result is too large to be held in a single byte. The processor would put 0x3C in the destination and set the carry bit to show that the result had overflowed and that a 1 should be carried into the next more significant byte. The carry flag also takes part in rotations and shifts. It is sometimes used as temporary storage to pass a bit from one register to another or to a subroutine.
- The *zero* flag Z is set when the result of an operation is 0. A common application is to check whether two values are equal: They are subtracted and the Z bit is tested to see whether the result is 0, which shows that the values are the same. This is used in Listing 4.6.
- The *negative* flag N is made equal to the msb of the result, which indicates a negative number if the values are signed.
- The *signed overflow* flag V is set when the result of a signed operation has overflowed, even though a carry may not be generated. An example is the sum $0x75 + 0x67 = 0xDC$. There are no problems if the variables are all unsigned.

7. Explain about various addressing modes of MSP430 with examples 6

An addressing mode is a way to represent the different types of registers and memory addresses where a function is implemented.The MSP430 allows it's user to select from a variety of modes some of they are:

Mode	Encoding	Syntax	operand
Register	00/0	Rn	Register contents.
Indexed	01/1	X(Rn)	Value at address X+Rx; *X is stored in an extension word following the instruction. x is encoded in the corresponding register field of the instruction.*
Absolute	01/1. Reg=r2	&ADDR	at constant address ADDR stored in an extension work (following the instruction)
Indirect Register	10/-	@Rn	variable pointed to by register Rn
Indirect Autoincrement	11/-	@Rn+	Rn contains address of operand; afterwards, the contents of Rn are incremented. Used for Pop and immediate.
Immediate	11/- Reg=R0	#N	N is a constant; it is stored in an extension word (following the instruction).

(Explain each)

8. Write the program in Assembly Language to light LEDs with a constant pattern 5

Listing 4.3: Program `ledsasm.s43` in assembly language to light LEDs with a constant pattern.

```
; ledsasm.s43 - simple program to light LEDs, absolute assembly
; Lights pattern of LEDs, sets pins to output, then loops forever
; Olimex 1121STK board with LEDs active low on P2.3,4


; J H Davies, 2006-05-17; IAR Kickstart version 3.41A
;------------------------------------------------------------------------------
#include <msp430x11x1.h>            ; Header file for this device

    ORG     0xF000                  ; Start of 4KB flash memory
Reset:                              ; Execution starts here
    mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
    mov.b   #00001000b,&P2OUT
                        ; LED2 (P2.4) on, LED1 (P2.3) off (active low!)
    mov.b   #00011000b,&P2DIR       ; Set pins with LEDs to output
InfLoop:                            ; Loop forever...
    jmp     InfLoop                 ; ...doing nothing
;------------------------------------------------------------------------------
    ORG     0xFFFE                  ; Address of MSP430 RESET Vector
    DW      Reset                   ; Address to start execution
    END
```

9. Explain the Arithmetic and Logical Instructions with both one and two operands in detail. 5

5.4.2 Arithmetic and Logic Instructions with Two Operands

Binary Arithmetic Instructions with Two Operands

These are fairly standard. The carry bit should be interpreted as "not borrow" for subtraction:

```
add.w   src,dst ; add                   dst += src
addc.w  src,dst ; add with carry        dst += (src + C)
adc.w   dst     ; add carry bit         dst += C        emulated
sub.w   src,dst ; subtract              dst -= src
subc.w  src,dst ; subtract with borrow  dst -= (src + ~C)
```

```
sbc.w   dst      ; subtract borrow bit       dst -= ~C       emulated
cmp.w   src,dst  ; compare, set flags only   (dst - src)
```

The compare operation `cmp` is the same as subtraction `sub` except that only the bits in SR are affected; the result is not written back to the destination. There are many examples of operations on more than one word with the carry/borrow bit in Section 5.1 of *Application Reports* (slaa024). Maxfield and Brown [37] give an entertaining account of binary arithmetic.

Arithmetic Instructions with One Operand

All these are emulated, which means that the operand is always a destination:

```
clr.w   dst      ; clear                     dst = 0         emulated
dec.w   dst      ; decrement                 dst--           emulated
decd.w  dst      ; double decrement          dst -= 2        emulated
inc.w   dst      ; increment                 dst++           emulated
incd.w  dst      ; double increment          dst += 2        emulated
tst.w   dst      ; test (compare with 0)     (dst - 0)       emulated
```

The test operation is the special case of comparison with 0. In many processors the clear operation differs from a move with the value 0 because a move sets the flags but a clear does not. This does not apply to the MSP430 because a move does not set the flags.

10.Write the program in C to read input from switch for lighting LED when button B1 is pressed. 5

Listing 4.5: Program butled1.c in C to light LED1 when button B1 is pressed. This version has a single loop containing a decision statement.

```
// butled1.c - press button to light LED
// Single loop with "if"
// Olimex 1121STK board, LED1 active low on P2.3,
//   button B1 active low on P2.1
// J H Davies, 2006-06-01; IAR Kickstart version 3.41A
//------------------------------------------------------------------------
#include <msp430x11x1.h>          // Specific device

// Pins for LED and button on port 2
#define LED1     BIT3
#define B1       BIT1
```



```
void main (void)
{
    WDTCTL = WDTPW | WDTHOLD;     // Stop watchdog timer
    P2OUT |= LED1;                // Preload LED1 off (active low!)
    P2DIR = LED1;                 // Set pin with LED1 to output
    for (;;) {                    // Loop forever
        if ((P2IN & B1) == 0) {   // Is button down? (active low)
            P2OUT &= ~LED1;       // Yes: Turn LED1 on (active low!)
        } else {
            P2OUT |= LED1;        // No: Turn LED1 off (active low!)
        }
    }
}
```

11.Write the program in Assembly Language and in 'C' Programming to light LEDs with a constant pattern

Listing 4.2: Program ledson.c in C to light LEDs with a constant pattern.

```
// ledson.c - simple program to light LEDs
// Sets pins to output, lights pattern of LEDs, then loops forever
// Olimex 1121STK board with LEDs active low on P2.3,4
// J H Davies, 2006-05-17; IAR Kickstart version 3.41A
//------------------------------------------------------------------------
#include <msp430x11x1.h>          // Specific device

void main (void)
{
    WDTCTL = WDTPW | WDTHOLD;       // Stop watchdog timer
    P2DIR = 0x18;                   // Set pins with LEDs to output, 0b00011000
    P2OUT = 0x08;                   // LED2 (P2.4) on, LED1 (P2.3) off (active low!)
    for (;;) {                      // Loop forever...
    }                               // ...doing nothing
}
```