

## ARM Memory Map and Memory Access

The ARM CPU uses 32-bit addresses which gives us a maximum of 4 GB (gigabytes) of memory space. This 4GB of memory space has addresses 0x00000000 to 0xFFFFFFFF, meaning each byte is assigned a unique address (ARM is a byte-addressable CPU).

D31	D24 D23	D16 D15	D8 D7	D0
0x00000003	0x00000002	0x00000001	0x00000000	0x00000000
0x00000007	0x00000006	0x00000005	0x00000004	0x00000004
0x0000000B	0x0000000A	0x00000009	0x00000008	0x00000008
0x0000000F	0x0000000E	0x0000000D	0x0000000C	0x0000000C
⋮	⋮	⋮	⋮	
0xFFFFFFFF3	0xFFFFFFFF2	0xFFFFFFFF1	0xFFFFFFFF0	0xFFFFFFFF0
0xFFFFFFFF7	0xFFFFFFFF6	0xFFFFFFFF5	0xFFFFFFFF4	0xFFFFFFFF4
0xFFFFFFFFB	0xFFFFFFFFA	0xFFFFFFFF9	0xFFFFFFFF8	0xFFFFFFFF8
0xFFFFFFFFF	0xFFFFFFFEE	0xFFFFFFFED	0xFFFFFFFEC	0xFFFFFFFEC

The 4GB of memory space is divided into three regions: code, data, and peripheral devices.

Address range	Name	Description
<b>0x00000000-0x1FFFFFFF</b>	Code	ROM or Flash memory
<b>0x20000000-0x3FFFFFFF</b>	SRAM	SRAM region used for on-chip RAM
<b>0x40000000-0x5FFFFFFF</b>	Peripheral	On-chip peripheral address space
<b>0x60000000-0x9FFFFFFF</b>	RAM	Memory, cache support
<b>0xA0000000-0xDFFFFFFF</b>	Device	Shared and non-shared device space
<b>0xE0000000-0xFFFFFFFF</b>	System	PPB and vendor system peripherals

PPB – Private  
Peripheral Bus

### Sample Memory Space Allocation in ARM Cortex

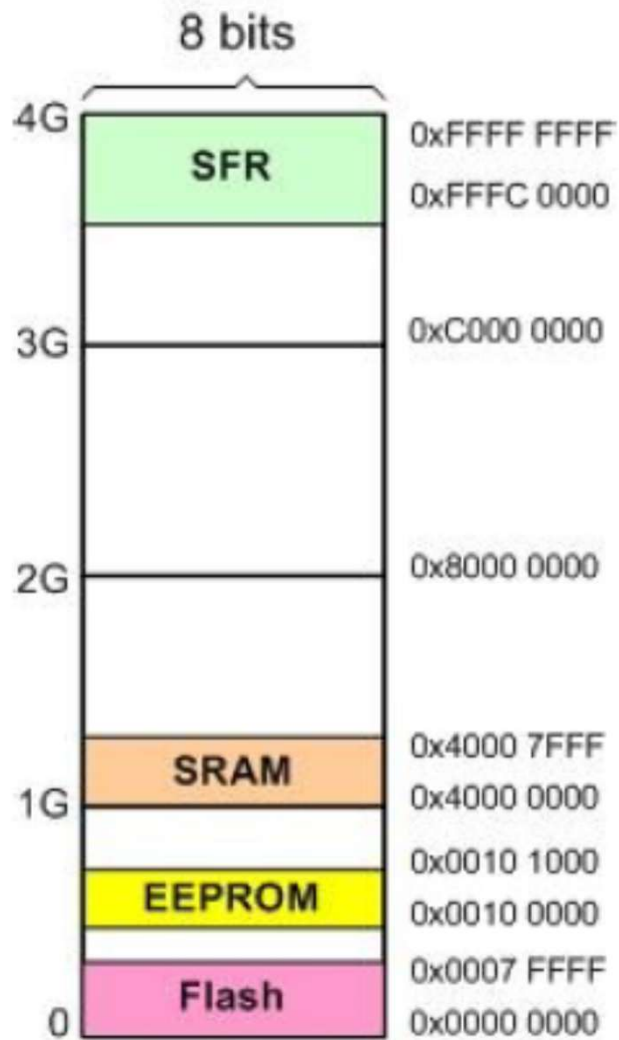
ARM Assembly Language Programming & Architecture by Mazidi, et al.

- ARM uses the memory mapped I/O.
- There is no standard for exact locations and types of memory and peripherals. Therefore the licensees can implement the memory and peripherals as they choose. For this reason the amount and the address locations of memory used by Flash ROM, SRAM, and I/O peripherals varies among the family members and chip manufacturers.
- Some of the memory addresses are set aside for the external (off-chip) memory and peripherals.

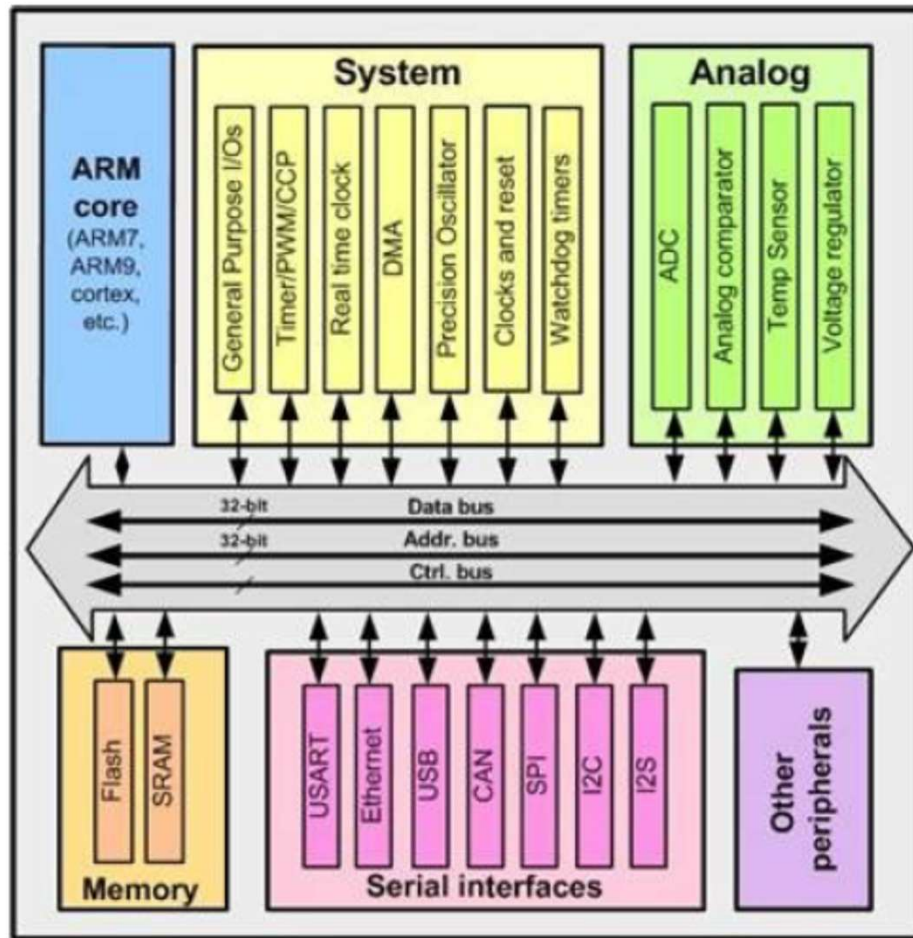
## Example:

A given ARM chip has the following address assignments. Calculate the space and the amount of memory given to each section.

- (a) Address range of 0x00100000 – 0x00100FFF for EEPROM
  - (b) Address range of 0x40000000 – 0x40007FFF for SRAM
  - (c) Address range of 0x00000000 – 0x0007FFFF for Flash
  - (d) Address range of 0xFFFC0000 – 0xFFFFFFFF for peripherals
- 
- (a) 0xFFF = 4K bytes.
  - (b) 7FFF = 32K bytes.
  - (c) 7FFFF = 512K bytes.
  - (d) 3FFFF = 256K bytes.



## ARM buses and memory access



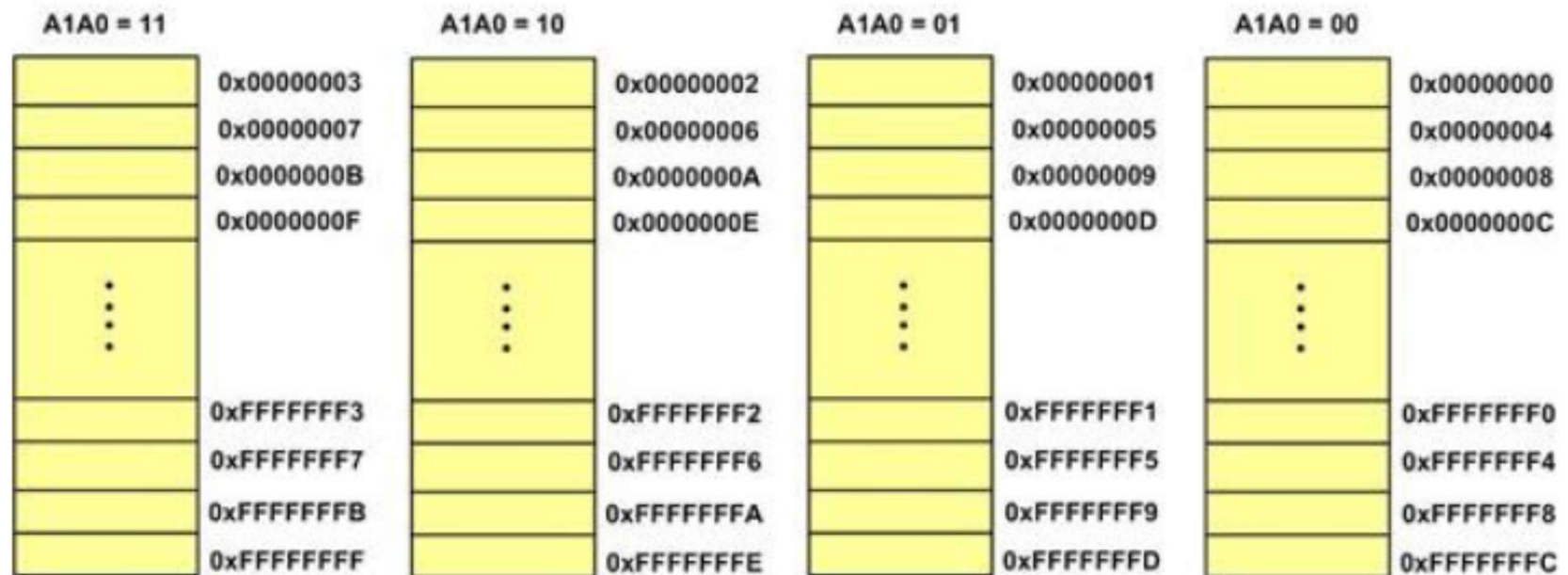
**Memory Connection Block Diagram in ARM**

## D31–D0 Data bus

- The 32-bit data bus of the ARM provides the 32-bit data path to the **on-chip** and **offchip memory** and **peripherals**.
- They are grouped into 8-bit data chunks, D0–D7, D8–D15, D16–D23, and D24–D31.

## A31–A0

- These signals provide the 32-bit address path to the **on-chip** and **off-chip memory** and **peripherals**.
- Since the ARM supports data access of byte (8 bits), half word (16 bits), and word (32 bits), the buses must be able to access any of the 4 banks of memory connected to the 32-bit data bus.
- The A0 and A1 are used to select one of the 4 bytes of the D31-D0 data bus.

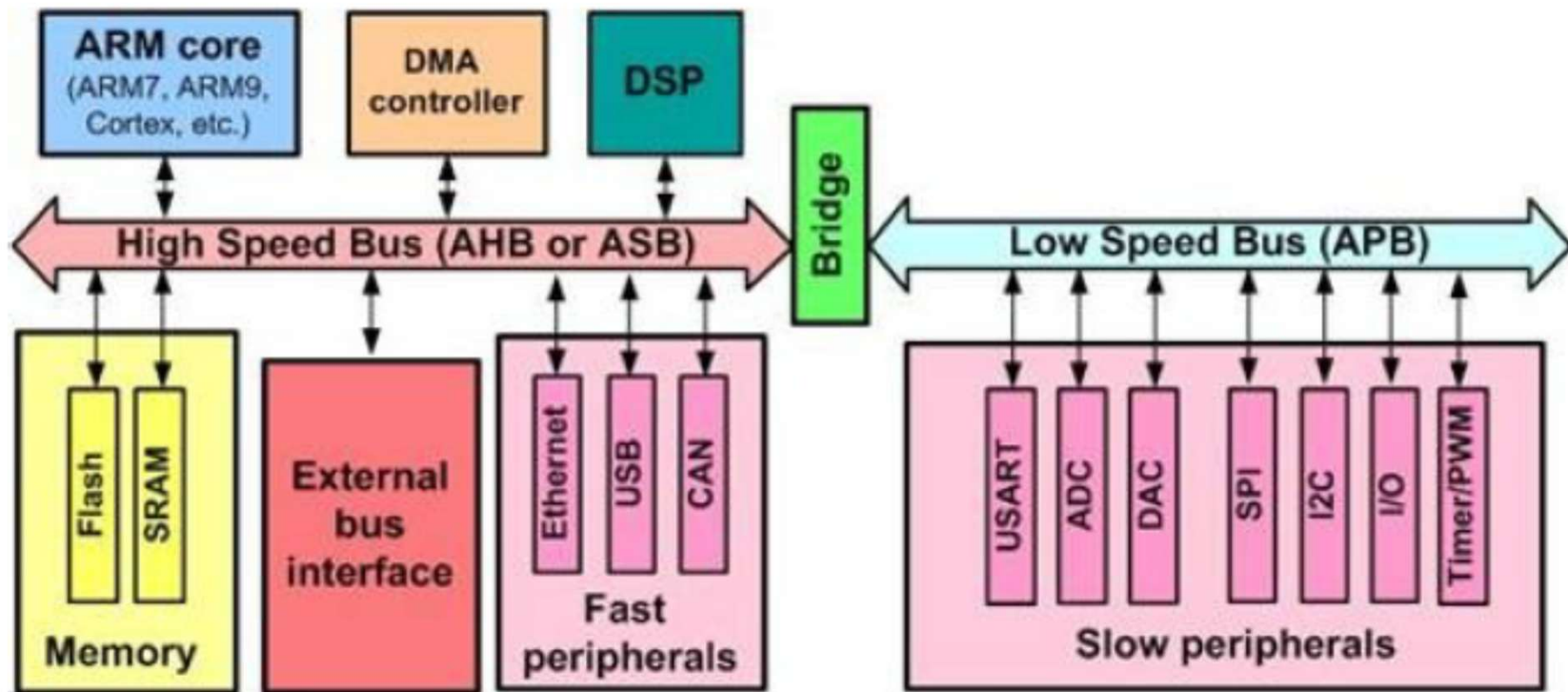


**Memory Block Diagram in ARM**



## AHB and APB buses

- CPU is connected to the on-chip memory via an AHB (advanced high performance bus).
- AHB is used for
  - connecting to on-chip ROM and RAM,
  - high speed I/O such as GPIO.
- ARM also has APB (advanced peripherals bus)
  - Used for on-chip peripherals such as timers, ADC, serial COM, SPI, I2C, and other peripheral ports.
- While we need the 32-bit data bus between CPU and the memory, many slower peripherals are 8 or 16 bits and there is no need for entire fast 32-bit data bus pathway.
- For this reason, ARM uses the AHB-to-APB bridge to access the slower on-chip devices such as peripherals.



AHB and APB in ARM

## Bus cycle time

- To access a device such as memory or I/O, the CPU provides a fixed amount of time called a **bus cycle time**.
- The **bus cycle time** used for **accessing memory** is often referred to as **MC (memory cycle)** time.
- The time from when the CPU provides the addresses at its address pins to when the data is expected at its data pins is called memory read cycle time.
- While for on-chip MC time can be 1 clock, in the off-chip memory the MC time is often 2 clocks.
- If memory is slow and its access time does not match the MC time of the CPU, extra time can be requested from the CPU to extend the read cycle time. This extra time is called a **wait state (WS)**.

Calculate the memory cycle time of a 50-MHz bus system with

- (a) 0 WS,
- (b) 1 WS, and
- (c) 2 WS.

Assume that the bus cycle time for off-chip memory access is 2 clocks.

Bus clock period =  $1/50 \text{ MHz} = 20 \text{ ns}$ .

Given: bus cycle time of zero wait states is 2 clocks,

So we have:

Memory cycle time with 0 WS  $2 \times 20 = 40 \text{ ns}$

Memory cycle time with 1 WS  $40 + 20 = 60 \text{ ns}$

Memory cycle time with 2 WS  $40 + 20 + 20 = 80 \text{ ns}$

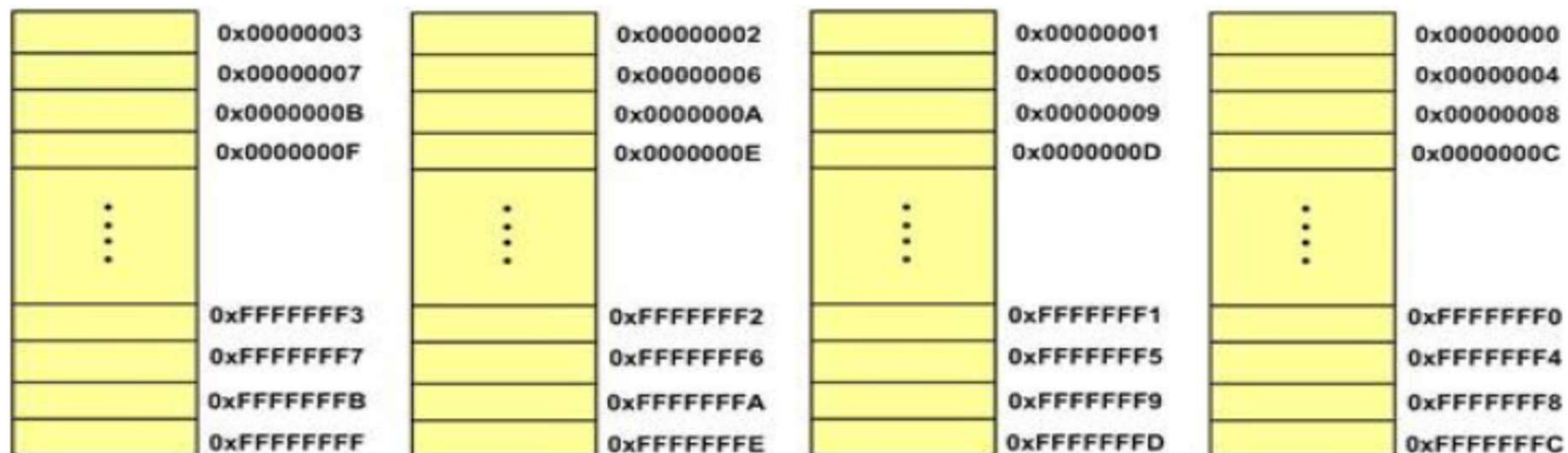
## Bus bandwidth

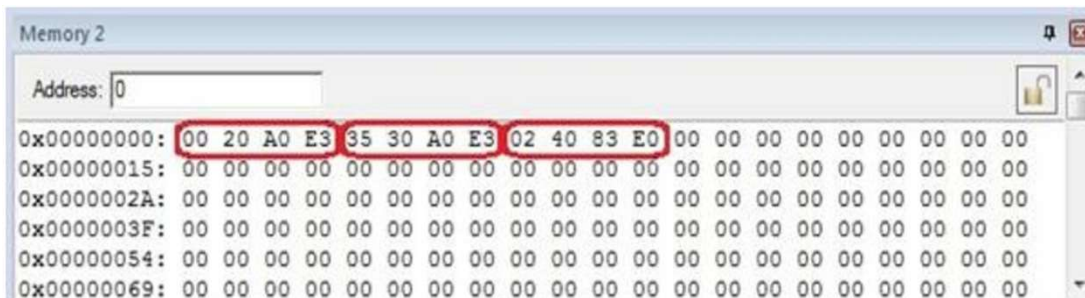
- The rate of data transfer is generally called bus bandwidth
- In other words, bus bandwidth is a measure of how fast buses transfer information between the CPU and memory or peripherals.
- Wider bus
  - Higher BW
  - But increases Si Die size for SoC
- Why Higher BW?
  - Fast CPU is of no use if BW is small.

Bus bandwidth =  $(1/\text{bus cycle time}) \times \text{bus width in bytes (MB)}$

## Code memory region

- The 4 GB of ARM memory space is organized as 1G × 32 bits. This is because, the ARM instructions are 32-bit.
- As ARM instructions are 32 bit and the internal bus of is 32-bit, we can **transfer one instruction in every clock cycle**.
- The fetching of an instruction in every clock cycle can work only if the code is word aligned, i.e., each instruction is placed at an address location ending with 0, 4, 8, or C.





## SRAM memory region

- A section of the memory space is used by SRAM. The SRAM can be on-chip or off-chip (external). This on-chip SRAM is used by the CPU to store parameters. It is also used by the CPU for the purpose of the stack.

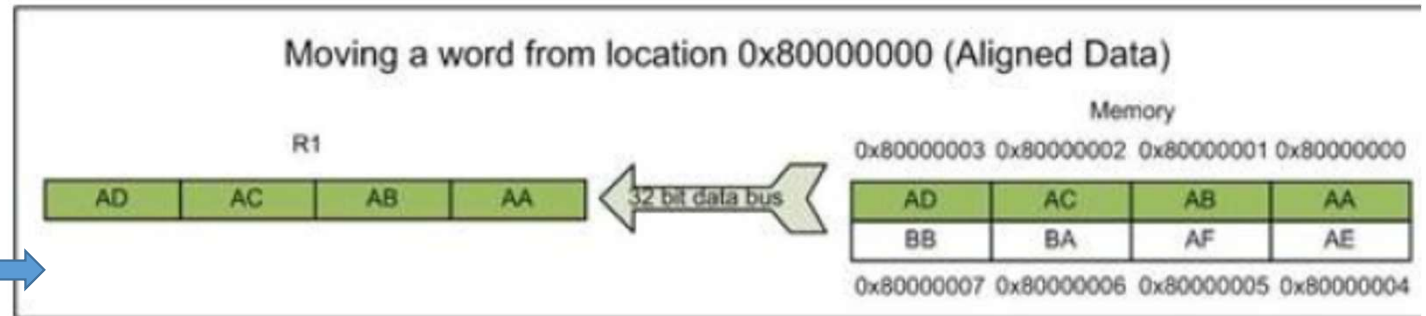
## Data misalignment in SRAM

- Data is aligned ➔ with every memory read we can get 4 bytes of information (data or code) using the D31–D0 data bus.
- **Compiler assures instructions are aligned** but the **programmer needs to make sure data in the SRAM is aligned**
- To make data word aligned, the least significant digits of the hex addresses must be **0, 4, 8, or C** (in hex).
- To make sure that data are also aligned we use the **align directive**.

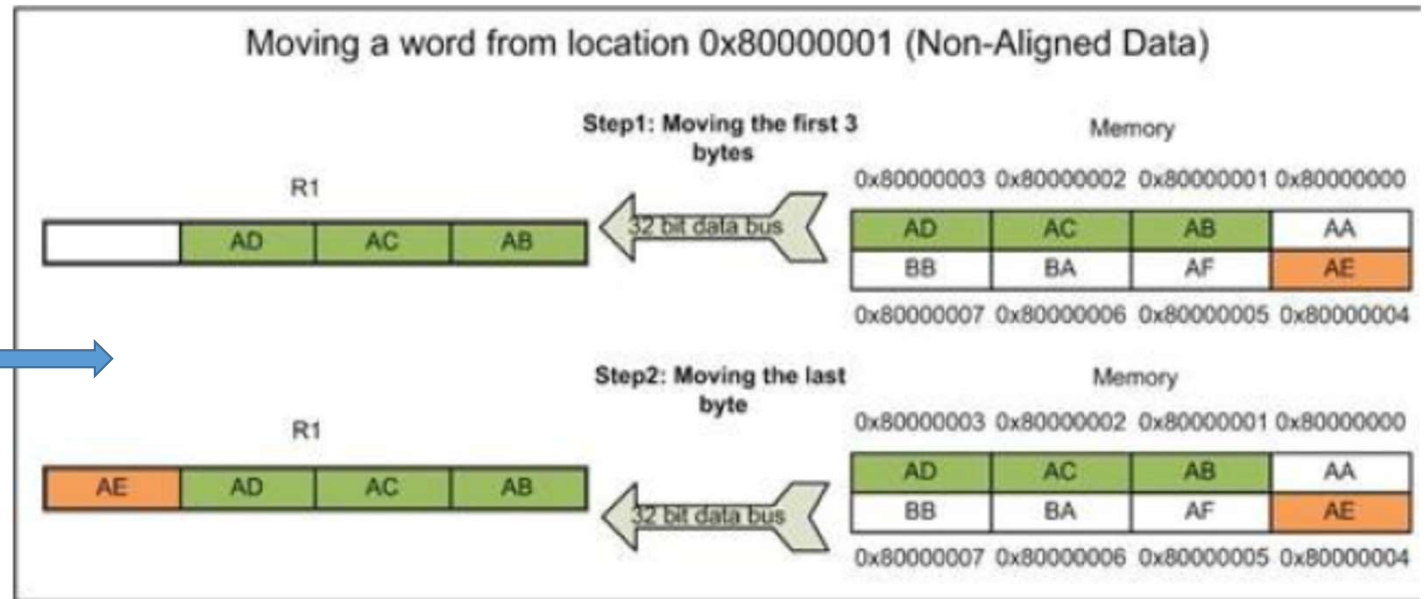


## Accessing non-aligned data

- LDR R1, [R0]  
R0=0x80000000
- 4 bytes of Mem contents are fetched in **one cycle**



- LDR R1, [R0]  
R0=0x80000001
- 8 bytes of mem locations 0x80000000 through 0x80000007 are being fetched in **two cycles**



### Memory Access for Aligned and Non-aligned Data

## Using LDR instruction with DCD and ALIGN directives

- The DCD and DCUD directives are used for 32-bit (word) data.
- The DCD directive ensures 32-bit data types are aligned, in contrast to DCUD which does not.
- DCD is used as :VALUE1 DCD 0x99775533
- This ensures that VALUE1, a word-sized operand, is located in a word aligned address location.
- An instruction accessing it will take only a single memory cycle.
- The one-time use of ALIGN directive at the beginning of data area using DCUD makes the data aligned for that group of data.

## Using LDRH with DCW and ALIGN directives

- Alignment of data is also an issue when the data size is in half-words (16-bit)
- With DCWU, we must use the ALIGN directive multiple times in the data area of a given program to ensure they are aligned.
- This is in contrast to the DCW directive which ensures data type to be half-word aligned.
- This is especially the case when we use the LDRH instruction.

Example:

```
LDR R1,=0x80000000 ;R1=0x80000000
```

```
LDR R3,=0xF31E4598 ;R3=0xF31E4598
```

```
LDR R4,=0x1A2B3D4F ;R4=0x1A2B3D4F
```

```
STR R3,[R1]
```

```
STR R4,[R1,#4]
```

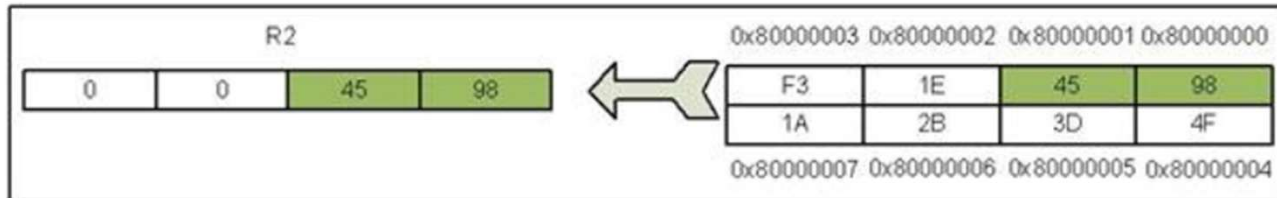
```
LDRH R2, [R1]
```

```
LDRH R2, [R1,#1]
```

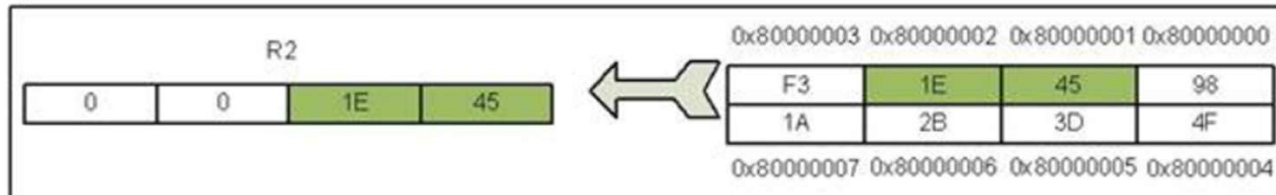
```
LDRH R2, [R1,#2]
```

```
LDRH R2, [R1,#3]
```

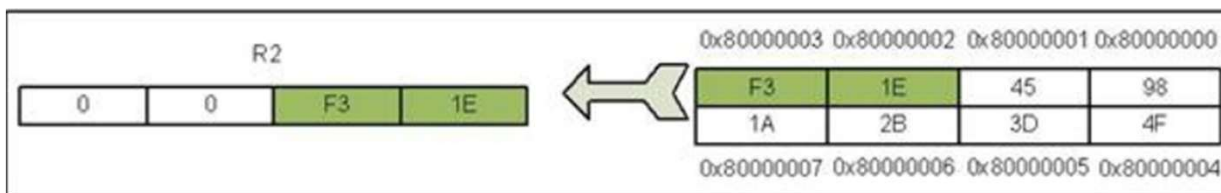
In the **LDRH R2,[R1]** instruction, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed but only 0x80000000 and 0x80000001 are used to get the 16 bits to R2. It takes only one memory cycle to transfer the data.



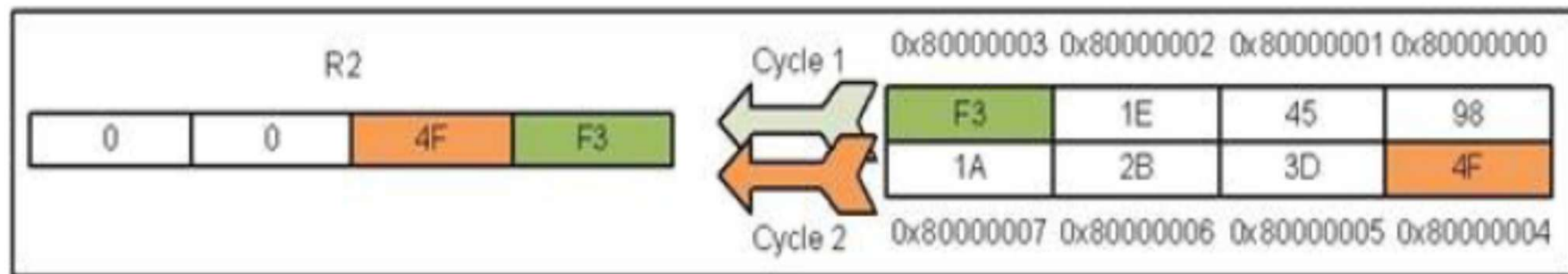
**LDRH R2,[R1,#1]**, instruction, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed, but only 0x80000001 and 0x80000002 are used to get the 16 bits to R2. It takes only one memory cycle to transfer the data.



**LDRH R2,[R1,#2]**, instruction, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed, but only 0x80000002 and 0x80000003 are used to get the 16 bits to R2. It takes only one memory cycle to transfer the data



**LDRH R2,[R1,#3]** instruction, in the first memory cycle, locations with addresses of 0x80000000, 0x80000001, 0x80000002, and 0x80000003 are accessed, but only 0x80000003 is used to get the lower 8 bits to R2. In the second memory cycle, the address locations 0x80000004, 0x80000005, 0x80000006, and 0x80000007 are accessed where only the 0x80000004 location is used to get the upper 8 bits to R2.



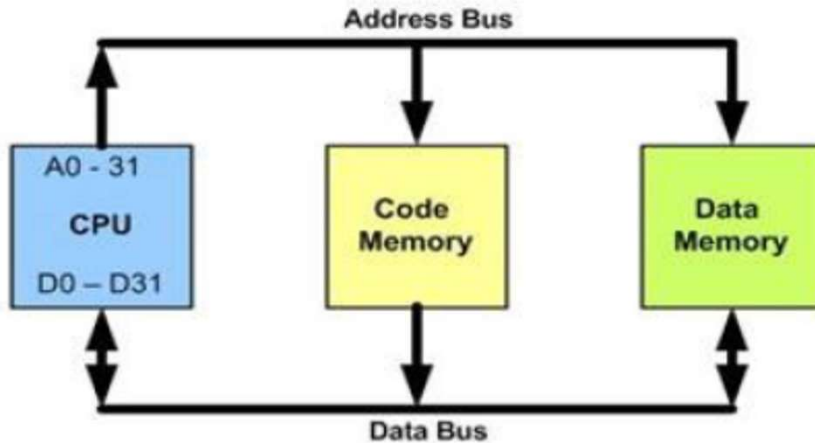
### **Using LDRB with DCB and ALIGN directives**

- The problem of misaligned data does not exist when the data size is bytes.
- Accessing a byte takes the same amount of time (one memory cycle) as an aligned word (4 bytes), regardless of the address location of the data.

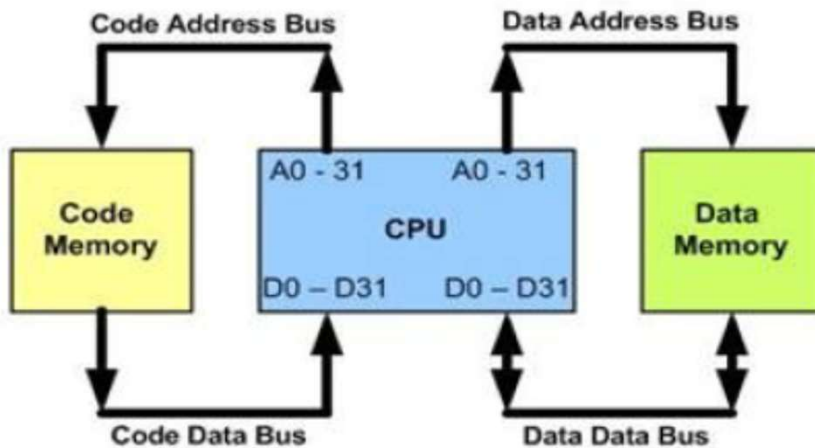
### **Peripheral region**

- A section of memory is set aside for peripherals. The type of peripherals and memory address locations used is unique to a vendor. The ARM manufacturers provide the details of memory map for the peripherals.

Von Neumann



Harvard



Von Neumann vs. Harvard Architecture

## Harvard Architecture and ARM

- In recent years many ARM manufacturers are using the Harvard architecture for ARM CPUs.
- Old ARM architectures up to ARM7 use Von Neumann architecture.
- The Harvard architecture feeds the CPU with both code and data at the same time via two sets of buses, one for code and one for data. This increases the processing power of the CPU since it can bring in more information.

## ARM Bit-Addressable Memory Region

One of the most important features of the CPU is the ability to access the RAM and I/O ports in bits instead of bytes. This is a very important and powerful feature of the ARM used widely in the embedded system design and applications.

### Bit-addressable (bit-band) SRAM

- Of the 4GB memory space of the ARM, a number of bytes are bit-addressable. The ARM literature refers to this region as the **bit-band region**. Since the ARM instruction is 32-bit and the CPU executes code one instruction at a time, the code (Flash ROM) space is always word size and word aligned,



- The bit addressable RAM and I/O locations vary among the family members and vendors.
- The ARM generic manual defines the location addresses of bit-band as 0x20000000 to 0x200FFFFFFF for SRAM and 0x40000000 to 0x400FFFFFFF for peripherals.
- The bit-band (bit-addressable) regions are the only region that can be accessed in both bit and byte/halfword/word formats while the other area of memory must be accessed in byte/halfword/word size.
- For the ARM Cortex-M, the bit-band SRAM addresses 0x20000000 to 0x200FFFFFFF (1M bytes) is **given alias addresses of 0x22000000 to 0x23FFFFFFF**.
- The bit addresses 0x22000000 to 0x2200001F are for the first byte of SRAM location 0x20000000, and 0x22000020 to 0x2200003F are the bit addresses of the second byte of SRAM location 0x20000001, and so on.

SRAM Byte addresses	SRAM Bit addresses							
	(We use these addresses to access the individual bits)							
	D7	D6	D5	D4	D3	D2	D1	D0
200FFFFFF	23FFFFFFC	F8	F4	F0	EC	E8	E4	23FFFFFFE0
200FFFFFE	23FFFFFDC	D8	D4	D0	CC	C8	C4	23FFFFFC0
200FFFFFD	23FFFFFBC	B8	B4	B0	AC	A8	A4	23FFFFFA0
200FFFFFC	23FFFF9C	98	94	90	8C	88	84	23FFFF80
200FFFFFB	23FFFF7C	78	74	70	6C	68	64	23FFFF60
200FFFFFA	23FFFF5C	x8	x4	x0	xC	x8	x4	23FFFF40
200XXXXXX	2XXXXXXC	X8	X4	X0	XC	X8	X4	2XXXXXX0
20000008	2200011C	118	114	...			104	22000100
20000007	220000FC	F8	F4	F0	EC	E8	E4	220000E0
20000006	220000DC	D8	D4	D0	CC	C8	C4	220000C0
20000005	220000BC	B8	B4	B0	AC	A8	A4	220000A0
20000004	2200009C	98	94	90	8C	88	84	22000080
20000003	2200007C	78	74	70	6C	68	64	22000060
20000002	2200005C	58	54	50	4C	48	44	20000040
20000001	2200003C	38	34	30	2C	28	24	22000020
20000000	2200001C	18	14	10	0C	08	04	22000000

**SRAM bit-addressable region and their alias addresses**

## Finding the alias address of a bit:

Bit alias address = Bit alias base address + Byte offset x 32 + Bit number x 4

- Calculate the bit alias address of bit 3 of 0x20000004

Bit alias base address = 0x22000000 (Look at the table, always constant)

Byte offset = 0x20000004 - 0x20000000 = 4

Bit number 3

Substituting

Bit alias address = 0x22000000 + 4 \* 32 + 3 \* 4

Bit alias address = 0x22000000 + 128 + 12 = 0x22000000 + 0x8C

Bit alias address = 0x2200008C

- Since each byte of SRAM has 8 bits we need an address for each bit.
- This means we need at least 8M address locations to access 8M bits, one address for each bit.
- To make the addresses word-aligned the ARM provides 4-byte alias address for each bit.
- For example, 0x22000000 to 0x2200001F is assigned to a single byte location of 0x20000000.
- That means we have 0x22000000 to 0x23FFFFFF (total of 32M locations, as alias addresses) for 1M bytes of address.
- SRAM locations 0x20000000 – 0x200FFFFFF are both byte-addressable and bit-addressable. The only difference is when we access it in byte (or halfword or word) we use addresses 0x20000000 to 0x200FFFFFF, but when they are accessed in bit, they are accessed via their alias addresses of 0x22000000 to 0x23FFFFFF.

Example:

Write a program to set HIGH the D6 of the SRAM location 0x20000001 using a) byte address and b) the bit alias address.

**Solution:**

a) LDR R1,=0x20000001 ;load the address of the byte

LDRB R2,[R1] ;get the byte

ORR R2,R2,#2\_01000000 ;make D6 bit high

STRB R2,[R1] ;write it back

b) address 0x22000038 is the bit address of D6 of SRAM location 0x20000001.

LDR R1,=0x22000038 ;load the alias address of the bit

MOV R2,#1 ;R2 = 1

STRB R2,[R1] ;Write one to D6

### **Peripheral I/O port bit-addressable region**

- The general purpose I/O (GPIO) and peripherals such as ADC, DAC, RTC, and serial COM port are widely used in the embedded system design.
- ARM has set aside 1M bytes of address space to be used bit-band (bit-addressable) I/O and peripherals.
- The address space assigned to bit-band peripherals and GPIO is 0x40000000 to 0x400FFFFFF with address aliases of 0x42000000 to 0x43FFFFFF.

# Stack and Stack Usage in ARM

- The stack is a section of RAM (**SRAM**) used by the CPU to store information **temporarily**.
- This information could be **data** or **an address** or **CPU registers** when calling a subroutine.
- Stack is also widely used when executing an **interrupt service routine (ISR)**.
- The **CPU needs** this storage area **because there are only a limited number of registers**.

## How stacks are accessed

- In the ARM CPU the register used to access the stack is **R13**.
- Storing of CPU information into stack → **PUSH**
- Retrieving/loading the information back → **POP**

## Pushing onto the stack

- R13 points to the stack
- As data is **PUSH**ed to the stack, SP can be either incremented or decremented → choice is yours

```
STR R1,[R13]    ;store R1 onto the stack,  
SUB R13,R13,#4  ;and decrement SP
```

## Popping from the stack

- As **POP** is executed SP → either incremented or decremented
- Here the top of the stack is loaded back to the register → **stack is LIFO (Last-In-First-Out) memory**

```
ADD R13,R13,#4    ;increment SP  
LDR R1, [R13]     ;load (POP) the top of stack to R1
```



The following ARM program places some data into registers and calls a subroutine that uses the same registers. It shows how to use the stack. Examine the stack, stack pointer, and the registers used after the execution of each instruction.

```
;initialize the SP to point
;to the last location of RAM (Stack_Top)
;Assume Stack_Top = 0xFF8
```

```
LDR R13,=Stack_Top      ;load SP
LDR R0,=0x125            ;R0 = 0x125
LDR R1,=0x144            ;R1 = 0x144
MOV R2,#0x56             ;R2 = 0x56
BL MY_SUB                ;call a subroutine
ADD R3,R0,R1             ;R3 = R0 + R1 = 0x125 + 0x144 = 0x269
ADD R3,R3,R2             ;R3 = R3 + R2 = 0x269 + 0x56 = 0x2BF
HERE B HERE              ;stay here
```

```
;-----
MY_SUB
;save R0, R1, and R2 on stack
;before they are used by a loop
STR R0,[R13]             ;save R0 on stack
SUB R13,R13,#4           ;R13 = R13 - 4,
STR R1,[R13]             ;save R1 on stack
SUB R13,R13,#4           ;R13 = R13 - 4
STR R2,[R13]             ;save R2 on stack
SUB R13,R13,#4           ;R13 = R13 - 4

;---R0,R1, and R2 are changed
MOV R0,#0 ;R0 = 0
MOV R1,#0 ;R1 = 0
MOV R2,#0 ;R2 = 0
;-----
```

;restore the original registers contents from stack

ADD R13,R13,#4 ;R13 = R13 + 4

LDR R2,[R13] ;restore R2 from stack

ADD R13,R13,#4 ;R13 = R13 + 4

LDR R1,[R13] ;restore R1 from stack

ADD R13,R13,#4 ;R13 = R13 + 4

LDR R0,[R13] ;restore R0 from stack

BX LR ;return to caller

After the execution of	Contents of some the registers (in Hex)				Stack
	R0	R1	R2	SP (R13)	
LDR R13,=0xFF8	0	0	0	FF8	<div> <div>FEC</div> <div>FF0</div> <div>FF4</div> <div>FF8</div> <div>← SP</div> </div>
LDR R0,=0x125 LDR R1,=0x144 LDR R2,=0x56	125	144	56	FF8	<div> <div>FEC</div> <div>FF0</div> <div>FF4</div> <div>FF8</div> <div>← SP</div> </div>
STR R0,[R13] SUB R13,R13,#4	125	144	56	FF4	<div> <div>FEC</div> <div>FF0</div> <div>FF4</div> <div>FF8</div> <div>← SP</div> </div> <div> <div>00</div> <div>00</div> <div>01</div> <div>25</div> </div>

STR R1,[R13] SUB R13,R13,#4	125	144	56	FF0	<table><tr><td>FEC</td><td></td><td></td><td></td><td></td></tr><tr><td>FF0</td><td></td><td></td><td></td><td></td></tr><tr><td>FF4</td><td>00</td><td>00</td><td>01</td><td>44</td></tr><tr><td>FF8</td><td>00</td><td>00</td><td>01</td><td>25</td></tr></table> <div>← SP</div>	FEC					FF0					FF4	00	00	01	44	FF8	00	00	01	25
FEC																									
FF0																									
FF4	00	00	01	44																					
FF8	00	00	01	25																					
STR R2,[R13] SUB R13,R13,#4	125	144	56	FEC	<table><tr><td>FEC</td><td></td><td></td><td></td><td></td></tr><tr><td>FF0</td><td>00</td><td>00</td><td>00</td><td>56</td></tr><tr><td>FF4</td><td>00</td><td>00</td><td>01</td><td>44</td></tr><tr><td>FF8</td><td>00</td><td>00</td><td>01</td><td>25</td></tr></table> <div>← SP</div>	FEC					FF0	00	00	00	56	FF4	00	00	01	44	FF8	00	00	01	25
FEC																									
FF0	00	00	00	56																					
FF4	00	00	01	44																					
FF8	00	00	01	25																					
MOV R0,#0 MOV R1,#0 MOV R2,#0	0	0	0	FEC	<table><tr><td>FEC</td><td></td><td></td><td></td><td></td></tr><tr><td>FF0</td><td>00</td><td>00</td><td>00</td><td>56</td></tr><tr><td>FF4</td><td>00</td><td>00</td><td>01</td><td>44</td></tr><tr><td>FF8</td><td>00</td><td>00</td><td>01</td><td>25</td></tr></table> <div>← SP</div>	FEC					FF0	00	00	00	56	FF4	00	00	01	44	FF8	00	00	01	25
FEC																									
FF0	00	00	00	56																					
FF4	00	00	01	44																					
FF8	00	00	01	25																					
ADD R13,R13,#4 LDR R2,[R13]	0	0	56	FF0	<table><tr><td>FEC</td><td></td><td></td><td></td><td></td></tr><tr><td>FF0</td><td></td><td></td><td></td><td></td></tr><tr><td>FF4</td><td>00</td><td>00</td><td>01</td><td>44</td></tr><tr><td>FF8</td><td>00</td><td>00</td><td>01</td><td>25</td></tr></table> <div>← SP</div>	FEC					FF0					FF4	00	00	01	44	FF8	00	00	01	25
FEC																									
FF0																									
FF4	00	00	01	44																					
FF8	00	00	01	25																					
ADD R13,R13,#4 LDR R1,[R13]	0	144	56	FF4	<table><tr><td>FEC</td><td></td><td></td><td></td><td></td></tr><tr><td>FF0</td><td></td><td></td><td></td><td></td></tr><tr><td>FF4</td><td></td><td></td><td></td><td></td></tr><tr><td>FF8</td><td>00</td><td>00</td><td>01</td><td>25</td></tr></table> <div>← SP</div>	FEC					FF0					FF4					FF8	00	00	01	25
FEC																									
FF0																									
FF4																									
FF8	00	00	01	25																					
ADD R13,R13,#4 LDR R0,[R13]	125	144	56	FF8	<table><tr><td>FEC</td><td></td><td></td><td></td><td></td></tr><tr><td>FF0</td><td></td><td></td><td></td><td></td></tr><tr><td>FF4</td><td></td><td></td><td></td><td></td></tr><tr><td>FF8</td><td></td><td></td><td></td><td></td></tr></table> <div>← SP</div>	FEC					FF0					FF4					FF8				
FEC																									
FF0																									
FF4																									
FF8																									

## **Nested calls in ARM:**

- Upon calling a subroutine from the main program using the BL instruction, R14, the linker register, keeps track of where the CPU should return after completing the subroutine.
- Now, if we have another call inside the subroutine using the BL instruction, then it is our job to store the original R14 on the stack. Failure to do that will end up crashing the program.

## Using LDM and STM instructions for the stack

- Pushing the registers onto the stack one register at a time is time consuming.
- Another way to push register contents onto the stack is to use
  - STM (store multiple)
  - LDM (load multiple).

### STM

STM R11, {R0-R3}

;Store R0 through R3 onto memory pointed to by R11

STM R7, {R0,R3,R5}

;Store R0, R3, R5 onto memory pointed to by R7

## **LDM**

LDM R11, {R0-R3}

;Load R0 through R3 from memory pointed to by R11

LDM R7, {R0,R3,R5}

;Load R0,R3,R5 from memory pointed to by R7

Modify the previous example using the LDM and STM instructions.

```
;initialize the SP to point  
;to the last location of RAM (Stack_Top)  
;Assume Stack_Top = 0xFF8
```

```
LDR R13,=Stack_Top      ;load SP  
LDR R0,=0x125            ;R0 = 0x125  
LDR R1,=0x144            ;R1 = 0x144  
MOV R2,#0x56             ;R2 = 0x56  
BL MY_SUB                ;call a subroutine  
ADD R3,R0,R1              ;R3 = R0 + R1 = 0x125 + 0x144 =  
0x269  
ADD R3,R3,R2              ;R3 = R3 + R2 = 0x269 + 0x56 =  
0x2BF  
HERE B HERE              ;stay here
```

## Previous

```
;-----  
MY_SUB  
;save R0, R1, and R2 on stack  
;before they are used by a loop  
STR R0,[R13]    ;save R0 on stack  
SUB R13,R13,#4  ;R13 = R13 - 4,  
STR R1,[R13]    ;save R1 on stack  
SUB R13,R13,#4  ;R13 = R13 - 4  
STR R2,[R13]    ;save R2 on stack  
SUB R13,R13,#4  ;R13 = R13 - 4  
  
;---R0,R1, and R2 are changed  
MOV R0,#0 ;R0 = 0  
MOV R1,#0 ;R1 = 0  
MOV R2,#0 ;R2 = 0  
;-----
```

## With STM

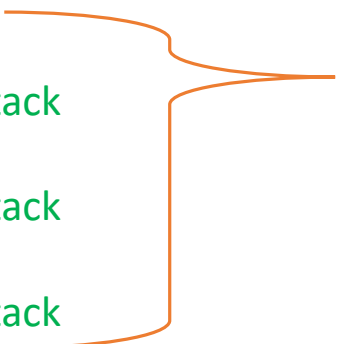
```
;-----  
MY_SUB  
;save R0,R1, and R2 on stack  
;before they are used by a loop  
  
STM R13,{R0-R2} ;save R0,R1,R2 on stack  
SUB R13,R13,#12  
  
;---R0,R1, and R2 are changed  
MOV R0,#0 ;R0=0  
MOV R1,#0 ;R1=0  
MOV R2,#0 ;R2=0  
;-----
```



### Previous

;restore the original registers contents from stack

```
ADD R13,R13,#4 ;R13 = R13 + 4
LDR R2,[R13]    ;restore R2 from stack
ADD R13,R13,#4 ;R13 = R13 + 4
LDR R1,[R13]    ;restore R1 from stack
ADD R13,R13,#4 ;R13 = R13 + 4
LDR R0,[R13]    ;restore R0 from stack
BX LR           ;return to caller
```



### With LDM

;restore the original registers contents from stack

```
ADD R13,R13,#12
LDM R13,{R0-R2} ;restore R0,R1, and R2 from stack
BX LR           ;return to caller
```

## Stack Operation

- An **Ascending stack** grows upwards. It starts from a low memory address and, as items are pushed onto it, progresses to higher memory addresses.
- A **Descending stack** grows downwards. It starts from a high memory address, and as items are pushed onto it, progresses to lower memory addresses.
- In an **Empty stack**, the stack pointers points to the next free (empty) location on the stack, i.e. the place where the next item to be pushed onto the stack will be stored.
- In a **Full stack**, the stack pointer points to the topmost item in the stack, i.e. the location of the last item to be pushed onto the stack.

## Writeback options of STM and LDM

We can specify the **action to be taken for the pointer**. The **action can be increment or decrement** before or after the push or pop is done.

### IA Increment After

IA stands for Increment After and **adds four to the pointer after** load or storing each register.

### IB Increment Before

IB stands for Increment Before and **adds four to the pointer before** load or storing each register.

### **DA** Decrement After

DA stands for Decrement After and **subtracts four from the pointer after** load or storing each register.

### **DB** Decrement Before

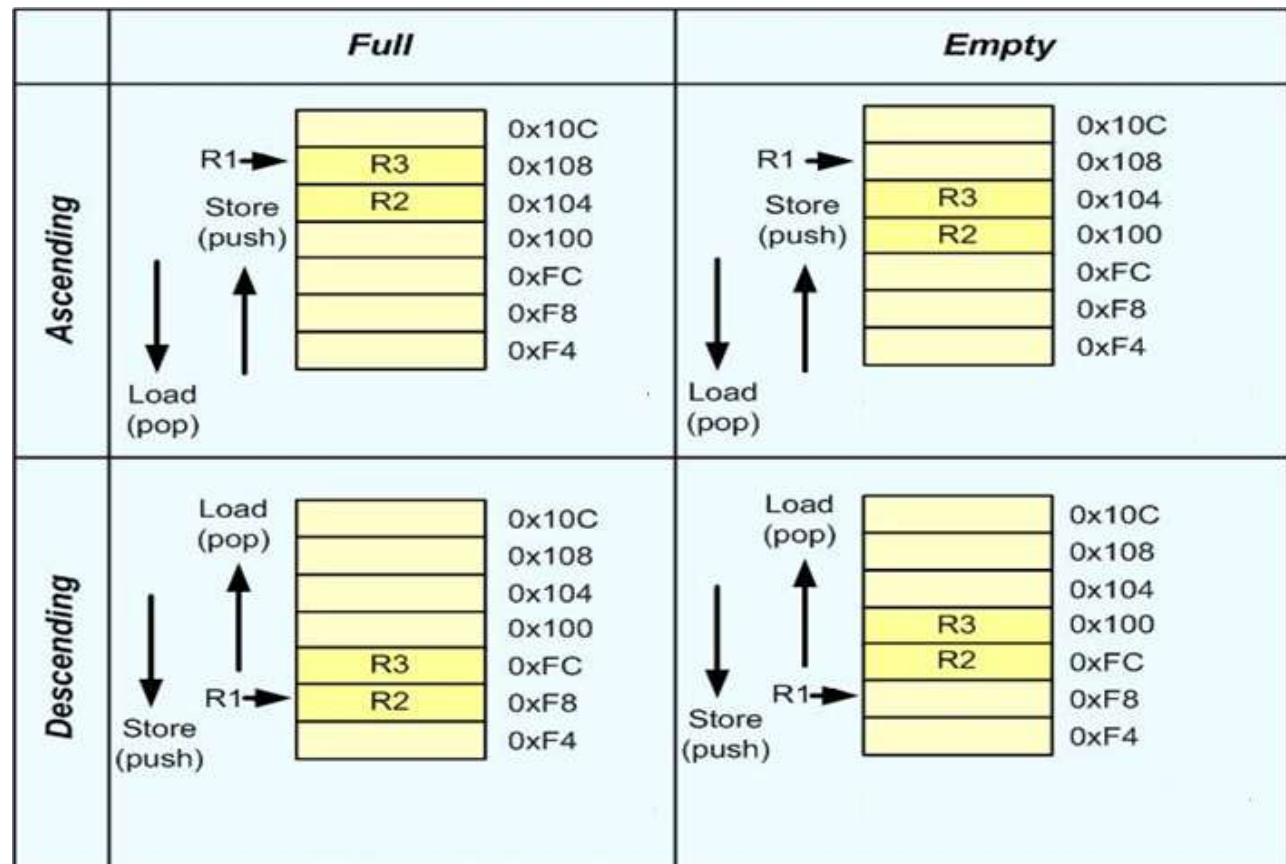
DB stands for Decrement Before and **subtracts four from the pointer before** load or storing each register.

For further clarification assume that **R1 = 0x100**. Figure below shows the memory after running **STM R1!,{R2,R3}** with each of IA, IB, DA and DB options.

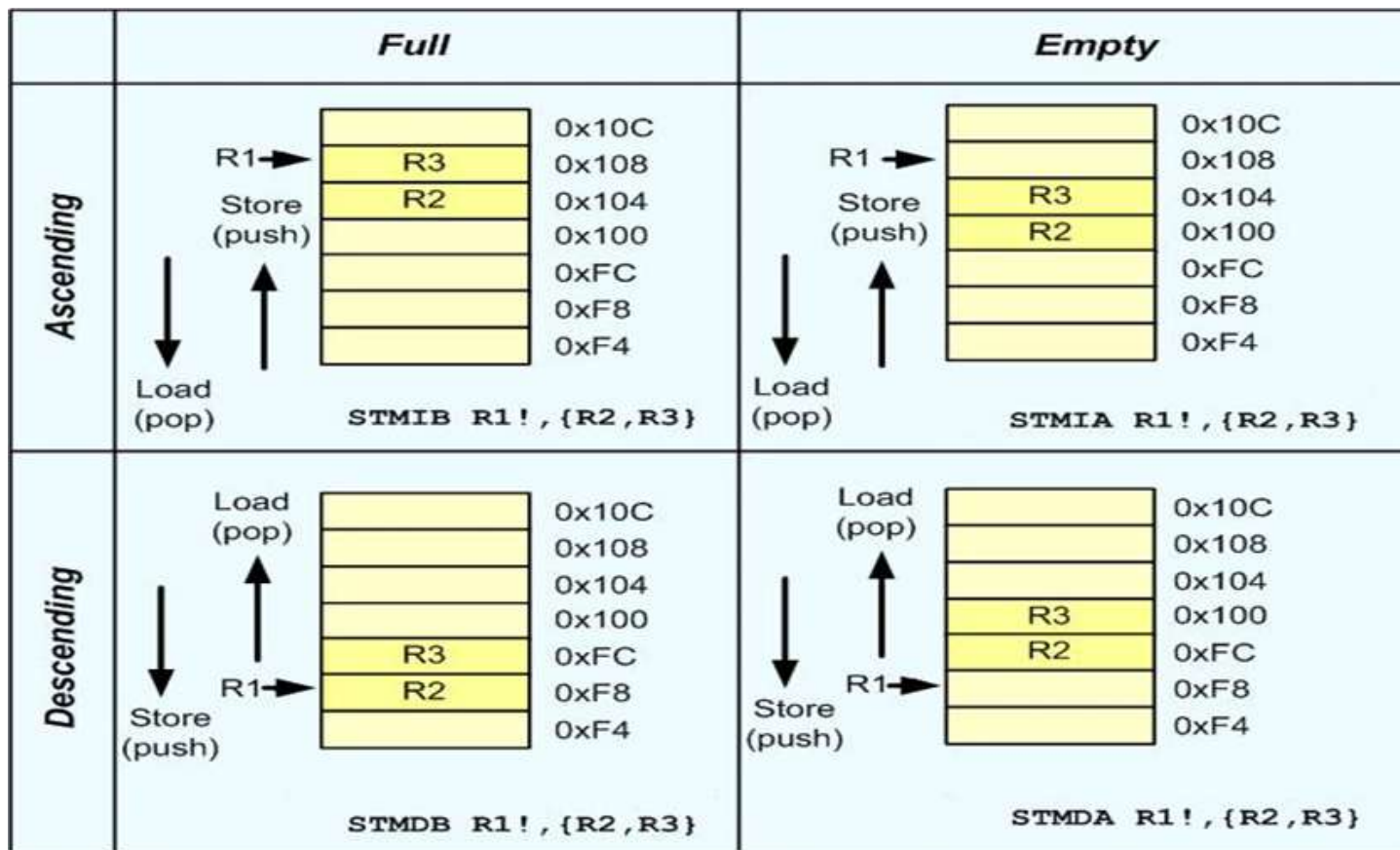
	<i>Before</i>	<i>After</i>
<b>Increment</b>	<p>After <b>STMIB R1! , {R2 , R3}</b></p>	<p>After <b>STMIA R1! , {R2 , R3}</b></p>
<b>Decrement</b>	<p>After <b>STMDB R1! , {R2 , R3}</b></p>	<p>After <b>STMDA R1! , {R2 , R3}</b></p>

- We have **four** stack structure, either it is **ascending** or **descending**.
- Stack is **ascending** when it is **incremented after** each **PUSH** and **decremented after** each **POP**.
- Stack is **descending** when it is **decremented after** each **PUSH** and **incremented after** POP.

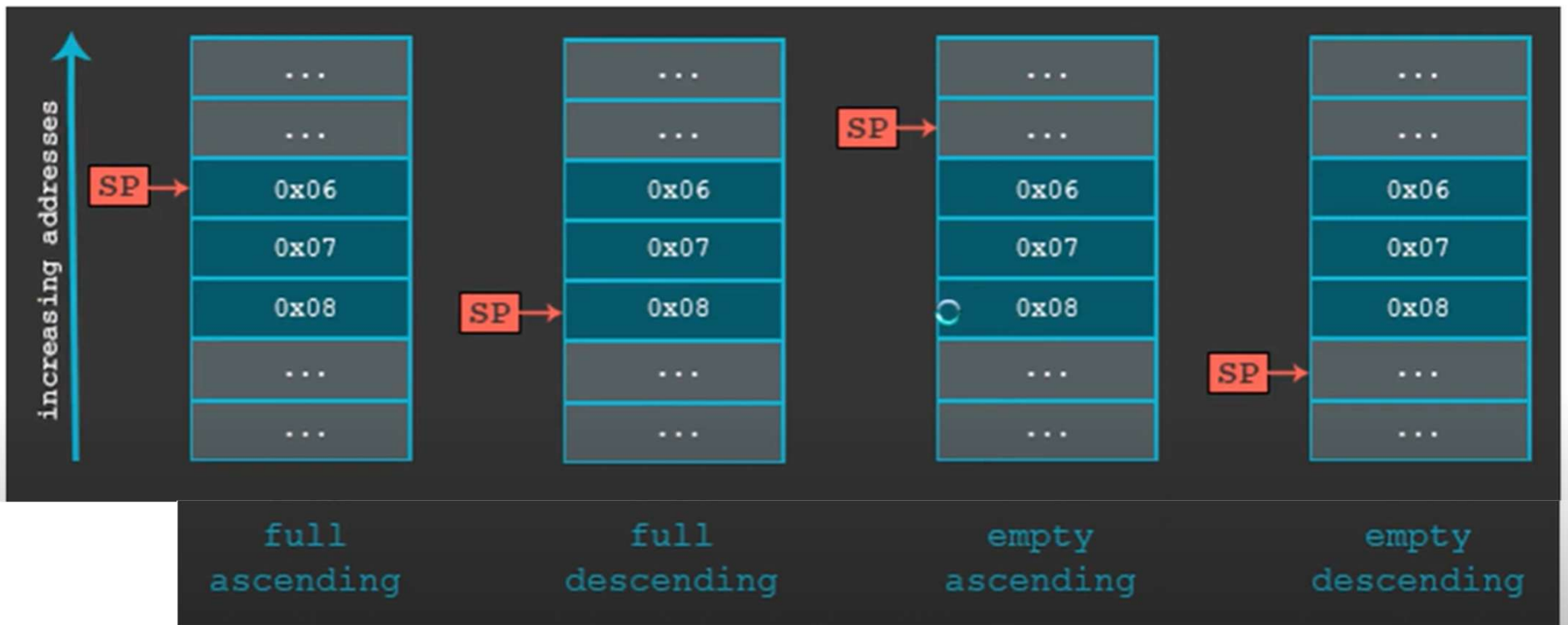
STM R1!,{R2,R3}



- The SP can point to the **last filled location**; in this case the stack is called **Full Stack**.
- The SP can point to the **next available location**, as well; which is called an **Empty Stack**.
- To support 4 types of the stack STM and LDM instructions take four suffixes IA, IB, DA and DB.
- If no suffix is used, the **default** action is **IA**



## Identify the stack structure





## Stack Instructions

- To make it easier for the programmer, stack-oriented suffixes can be used instead of the increment or decrement and before or after suffixes

Stack type	Store (PUSH)	Load (POP)
Full descending	STMFD (STMDB, Decrement Before)	LDMFD (LDM, increment after)
Full ascending	STMFA (STMIB, Increment Before)	LDMFA (LMDA, Decrement After)
Empty descending	STMED (STMDA, Decrement After)	LDMED (LDMIB, Increment Before)
Empty ascending	STMEA (STM, increment after)	LDMEA (LDMDB, Decrement Before)

Stack Structure	Load	Store	Load (alternate Names)	Store (alternate Names)
Full Ascending	LDMDA	STMIB	LDMFA	STMFA
Full Descending	LDMIA	STMDB	LDMFD	STMFD
Empty Ascending	LDMDB	STMIA	LDMEA	STMEA
Empty Descending	LDMIB	STMDA	LDMED	STMED

### The four stack structures and the options of LDM and STM instructions

The stack structure used by ARM for **PUSH**, **POP** instructions and **interrupt handling** is a Full Descending stack using R13 as the stack pointer. To support Full Descending stack, **STMDB** and **LDMIA** pair of instructions should be used.

## **STMDB      Store Multiple register and Decrement Before**

**Flags:** Unaffected.

**Format:** STMDB Rn, {Rx,Ry,...}

**Function:** Stores registers Rx, Ry,... into consecutive memory locations. The starting address of memory location is given by Rn register. The source registers are separated by comma and placed in braces.

STMDB R11!, {R0, R1, R2, R3}

STMDB R11!, {R0-R3}

;Store R0 through R3 onto memory pointed to by R11 and update R11 with the final address

STMDB R11!, {R0, R5, R3}

;Store R0, R3 and R5 onto memory pointed to by R11 and update R11 with the final address

**STMDB R11!, {R0-R3, R8, R7}**

;Store R0 through R3, R7 and R8 onto memory pointed to by R11 and update R11 with the final address

## **LDMIA     Load Multiple registers and Increment after each Access**

Flags: Unaffected.

Format: **LDMIA Rn, {Rx,Ry,..}**

Function: This is the same as the LDM instructions. This IA (Increment the address after each Access) is the default. We use this for Popping (loading) multiple words from descending stack into CPU registers.

**LDMIA R11!, {R0, R1, R2, R3}**

**LDMIA R11!, {R0-R3}**

**LDMIA R11!, {R0, R5, R3}**

When the registers are pushed on to the stack, lower numbered registers are stored in the lower address and when popped, data from lower address goes into the lower numbered register.

PUSH and POP instructions.

**PUSH      PUSH register onto stack**

**Flags:** Unaffected.

**Format:** PUSH {reg\_list} ;PUSH reg\_list onto stack

**Function:** Copies the contents of registers stated in reg\_list onto the stack and decrements SP by 4, 8, 12, 16, ... depending on the number of registers in reg\_list.

Example:

PUSH {R1} ;PUSH the R1 onto top of stack

PUSH {R1,R4,R7} ;PUSH R1,R4,R7 onto top of stack

PUSH {R2-R6} ;PUSH the R2,R3,R4,R5,R6 onto top of stack

PUSH {R0,R5} ;PUSH the R0 and R5 onto top of stack

PUSH {R0-R7} ;PUSH the R0 through R7 onto top of stack

The PUSH instruction is alias for STMDB R13!.

## **POP                    POP register from Stack**

**Flags:** Unaffected.

**Format:** POP {reg\_list} ;reg\_reg = words off top of stack

**Function:** Copies the words pointed to by the stack pointer to the registers indicated by the reg\_list and increments the SP by 4, 8, 12, 16, ... depending on the number of registers in the reg\_list.

**Example:**

POP {R1} ;POP the top word of stack to R1

POP {R1,R4,R7} ;POP the top 3 words of stack to R1,R4,R7

POP {R2-R6} ;POP the top 5 words of stack to R2-R6

POP {R0,R5} ;POP the top 2 words of stack to R0 and R5

POP {R0-R7} ;POP the top 8 words of stack to R0-R7

The POP instruction is alias for LDMIA R13!.