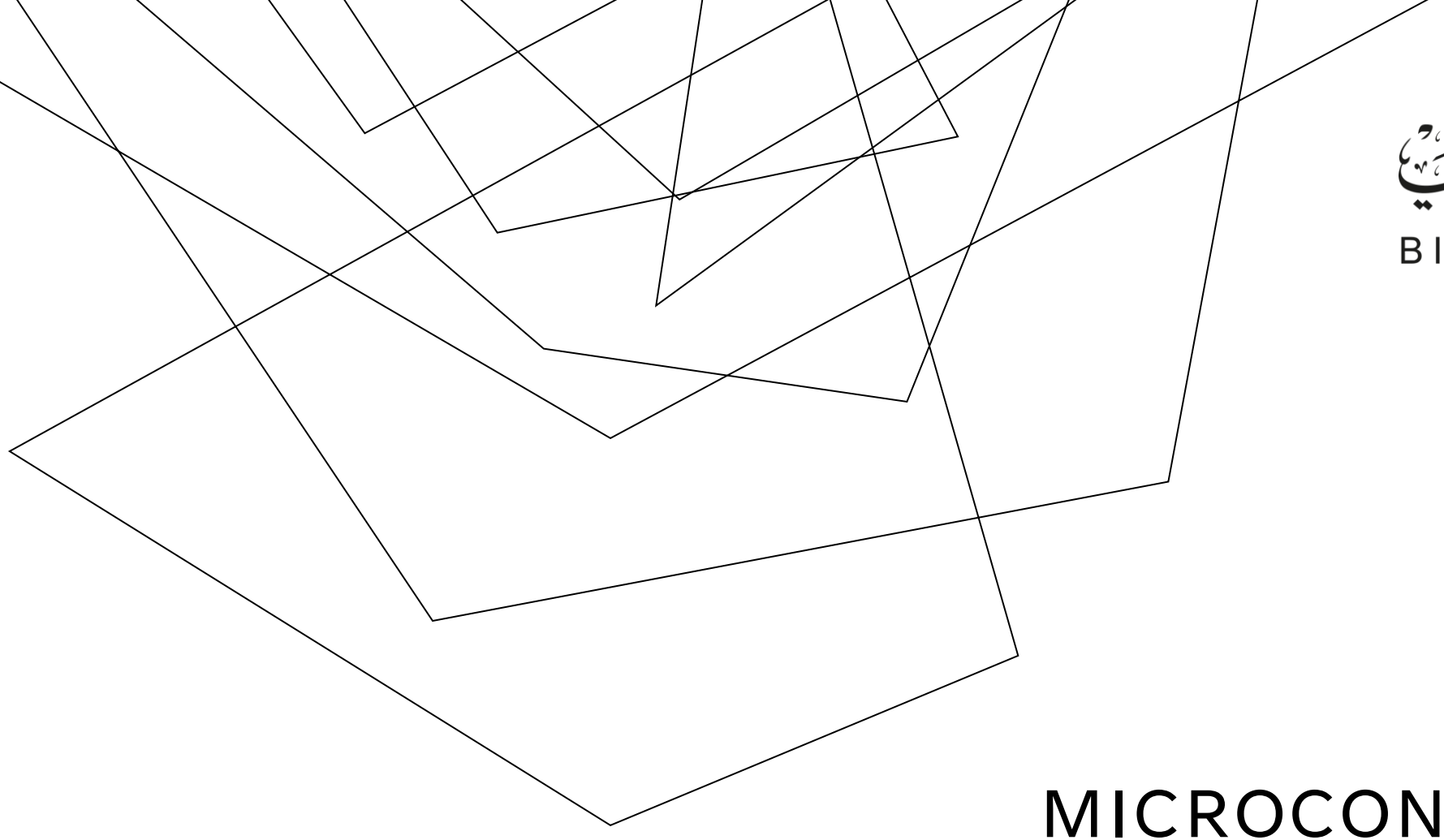


جامعة بيرزيت
BIRZEIT UNIVERSITY

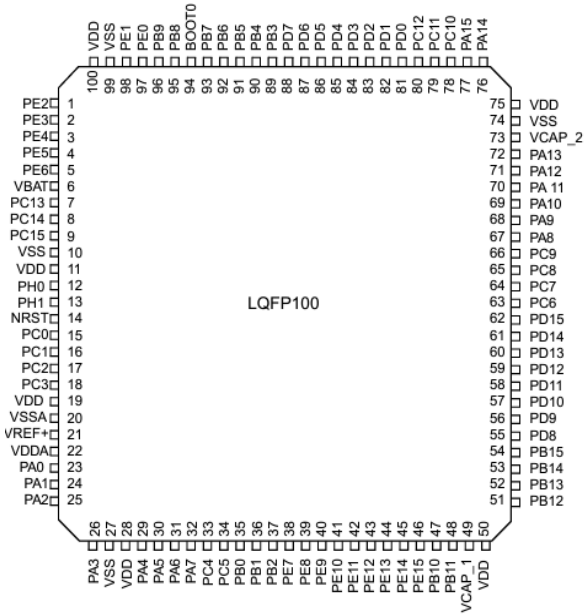
MICROCONTROLLER ARCHITECTURE

Shadi Daana

STUDENTS-HUB.com

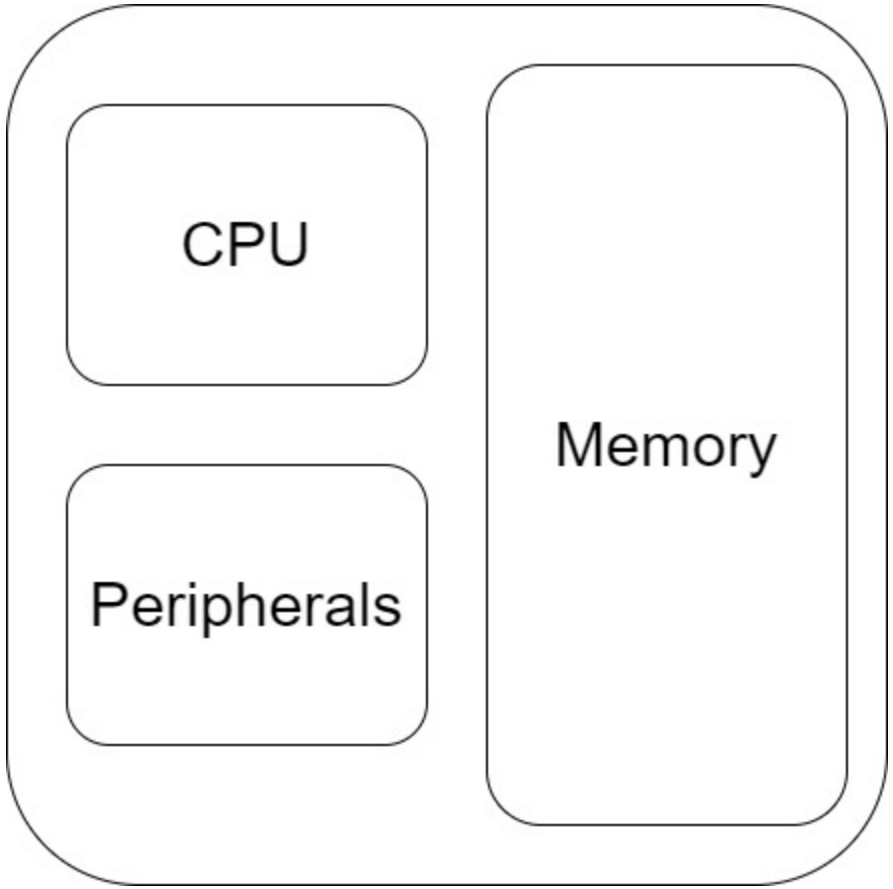
Microcontroller

- A **microcontroller** can be considered as a computer in a single chip
- Typically used in **embedded systems** to interface with sensors and actuators
- The core elements of a microcontroller are:
  - The processor (**CPU**)
  - Memory
  - I/O peripherals
  - System bus
- The system bus is the connective wire that links all components of the microcontroller together.
- Embedded **software** is programmed on a separate computer and then **compiled** into a list of **instructions**.
- These instructions then copied into the microcontroller memory
- This is often known as **firmware**
- The microcontroller then executes these **instructions**



STUDENTS-HUB.com

Microcontroller Components



STUDENTS-HUB.com

The processor (CPU)

- A processor can be thought of as the **brain** of the device.
- It processes and responds to various instructions that direct the microcontroller's function.
- This involves performing basic **arithmetic**, **logic** and **I/O** operations.
- It also performs data transfer operations, which communicate commands to other components in the larger embedded system

STUDENTS-HUB.com

I/O Peripherals

- The input and output devices are the interface for the processor to the outside world.
- The input ports receive information and send it to the processor in the form of binary data.
- The processor receives that data and sends the necessary instructions to output devices that execute tasks external to the microcontroller.

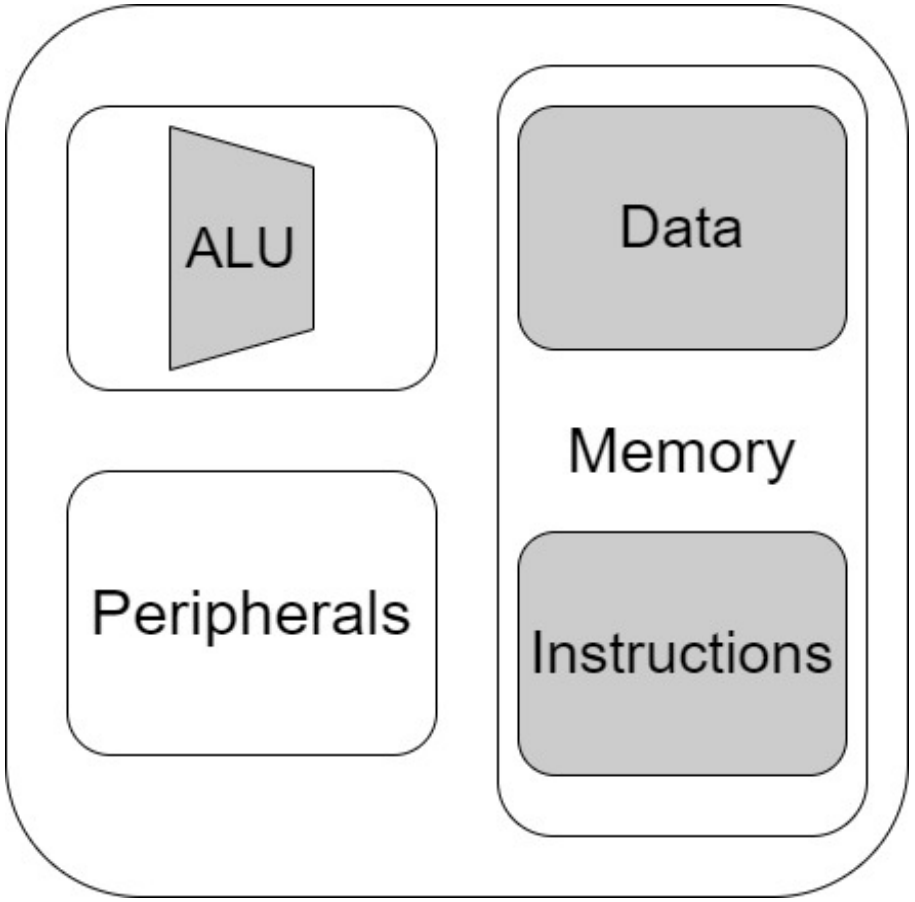
STUDENTS-HUB.com

Memory

- A microcontroller's memory is used to store the data that the processor receives and uses to respond to instructions that it's been programmed to carry out.
- A microcontroller has two main memory types:
  - **Program memory** :
    - stores long-term information about the instructions that the CPU carries out.
    - Program memory is non-volatile memory, meaning it holds information over time without needing a power source.
  - **Data memory:**
    - required for temporary data storage while the instructions are being executed.
    - Data memory is volatile, meaning the data it holds is temporary and is only maintained if the device is connected to a power source

STUDENTS-HUB.com

Microcontroller Components



STUDENTS-HUB.com

Supporting Elements

- Other supporting elements of a microcontroller include:
  - Analog to Digital Converter (**ADC**):
    - An ADC is a circuit that converts analog signals to digital signals.
    - It allows the processor at the center of the microcontroller to interface with external analog devices, such as sensors.
  - Digital to Analog Converter (**DAC**) :
    - A DAC performs the inverse function of an ADC and allows the processor at the center of the microcontroller to communicate its outgoing signals to external analog components.
  - Serial port:
    - The serial port is one example of an I/O port that allows the microcontroller to connect to external components.
    - It has a similar function to a USB or a parallel port but differs in the way it exchanges bits.

STUDENTS-HUB.com

ARM® Cortex®

- ARM® is a family of RISC-based microprocessors and microcontrollers designed by ARM Holdings.
- The company doesn’t make processors but instead designs microprocessor and multicore architectures and licenses them to manufacturers
- ARM® designs the core and bus system
- For example, STMicroelectronics purchases a license (IP) to use the design in their own microcontrollers (MCUs)
- STMicroelectronics then adds its own peripherals, memories, and power blocks to create a functional MCU
- Companies can also purchase designs for peripherals and memory, or they can design them by themself

STUDENTS-HUB.com

ARM® Processor Vs Intel Processor

- There are many differences between Intel and ARM, but the main difference is the instruction set.
  - Intel is a CISC (Complex Instruction Set Computing) processor that has a larger and more feature-rich instruction set and allows many complex instructions to access memory.
  - It therefore has more operations, addressing modes, but fewer registers than ARM. CISC processors are mainly used in normal PCs, workstations, and servers.
  - ARM is a RISC (Reduced instruction set Computing) processor and therefore has a simplified instruction set (100 instructions or less) and more general-purpose registers than CISC.
  - Unlike Intel, ARM uses instructions that operate only on registers and uses a Load/Store memory model for memory access, which means that only Load/Store instructions can access memory.

STUDENTS-HUB.com

ARM® Processor Vs Intel Processor

- More differences between ARM and x86 are:
  - In ARM, most instructions can be used for conditional execution.
  - The Intel x86 and x86-64 series of processors use the little-endian format
  - The ARM architecture was a little-endian before version 3. Since then ARM processors have become BI-endian and feature a setting that allows for switchable endianness.
- There are not only differences between Intel and ARM, but also between different ARM versions themselves. (Armv6, Armv7, and Armv8)

STUDENTS-HUB.com

ARM® Cortex®

- ARM licenses a number of specialized microprocessors and related technologies, but the bulk of their product line is the Cortex family of microprocessor architectures.
- There are three Cortex architectures, conveniently labeled with the initials A, R, and M.
  - **Cortex-M:** Processors in these profiles are used for the development of microcontroller-based based embedded systems. The Cortex-M family consists of Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, Cortex-M4 and Cortex-M7.
  - **Cortex-A**: Processors in this profile are used in high-performance application devices like mobile/cellular phones.
  - **Cortex-R**: The main market of processors of this profile is in the real-time application, where less response time is the main target

STUDENTS-HUB.com

ARM® Cortex®

- There are currently four popular versions of the Cortex-M series:
  - **Cortex-M0:** Designed for 8- and 16-bit applications, this model emphasizes low cost, ultra-low power, and simplicity. It is optimized for small silicon die size (starting from 12k gates) and use in the lowest cost chips.
  - **Cortex-M0+**: An enhanced version of the M0 that is more energy efficient.
  - **Cortex-M3**:Designed for 16- and 32-bit applications, this model emphasizes performance and energy efficiency. It also has comprehensive debug and trace features to enable software developers to develop their applications quickly.
  - **Cortex-M4**: This model provides all the features of the Cortex-M3, with additional instructions to support digital signal processing tasks.

STUDENTS-HUB.com

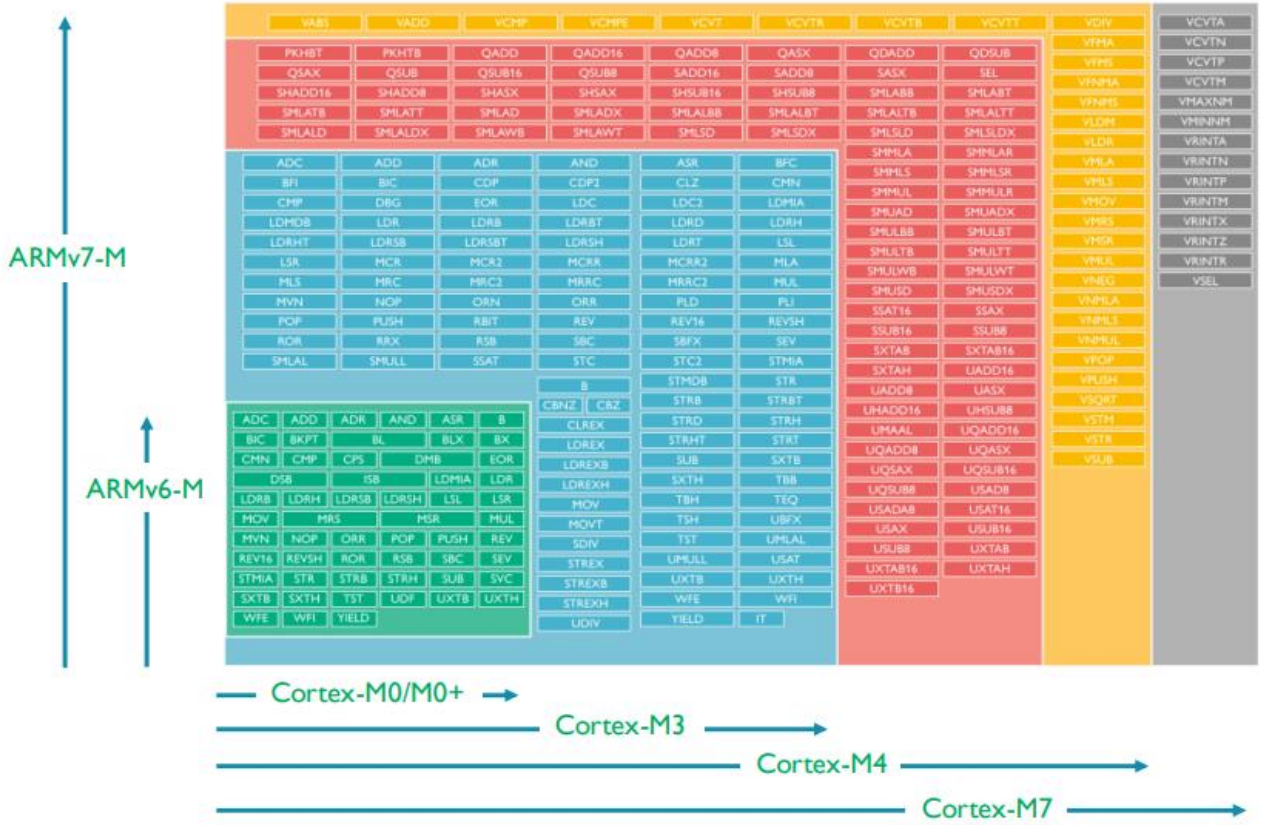
ARM® Cortex®

- Under the M-Profile there are several versions of the ARM ISA
  - ARMV6-M
  - ARMV7-M
  - ARMV8-M
- A processor definition includes a version of the instruction set architecture
- In this course, we will focus on the microcontrollers built around the Cortex M4 core.
- Cortex M4 uses ARMV7-M ISA
- All Cortex-M processors support an instruction set called **Thumb**
- **Thumb-2** is an expansion of Thumb
- Cortex-M processors support different subset of the instructions available in the Thumb ISA

Processor	ISA
Cortex-M/M0	ARMv6-M
Cortex-M3	ARMv7-M
Cortex-M4	ARMv7E-M
Cortex-M7	ARMv7E-M
Cortex-M23	ARMv8-M

STUDENTS-HUB.com

ARM® Cortex®



Floating Point

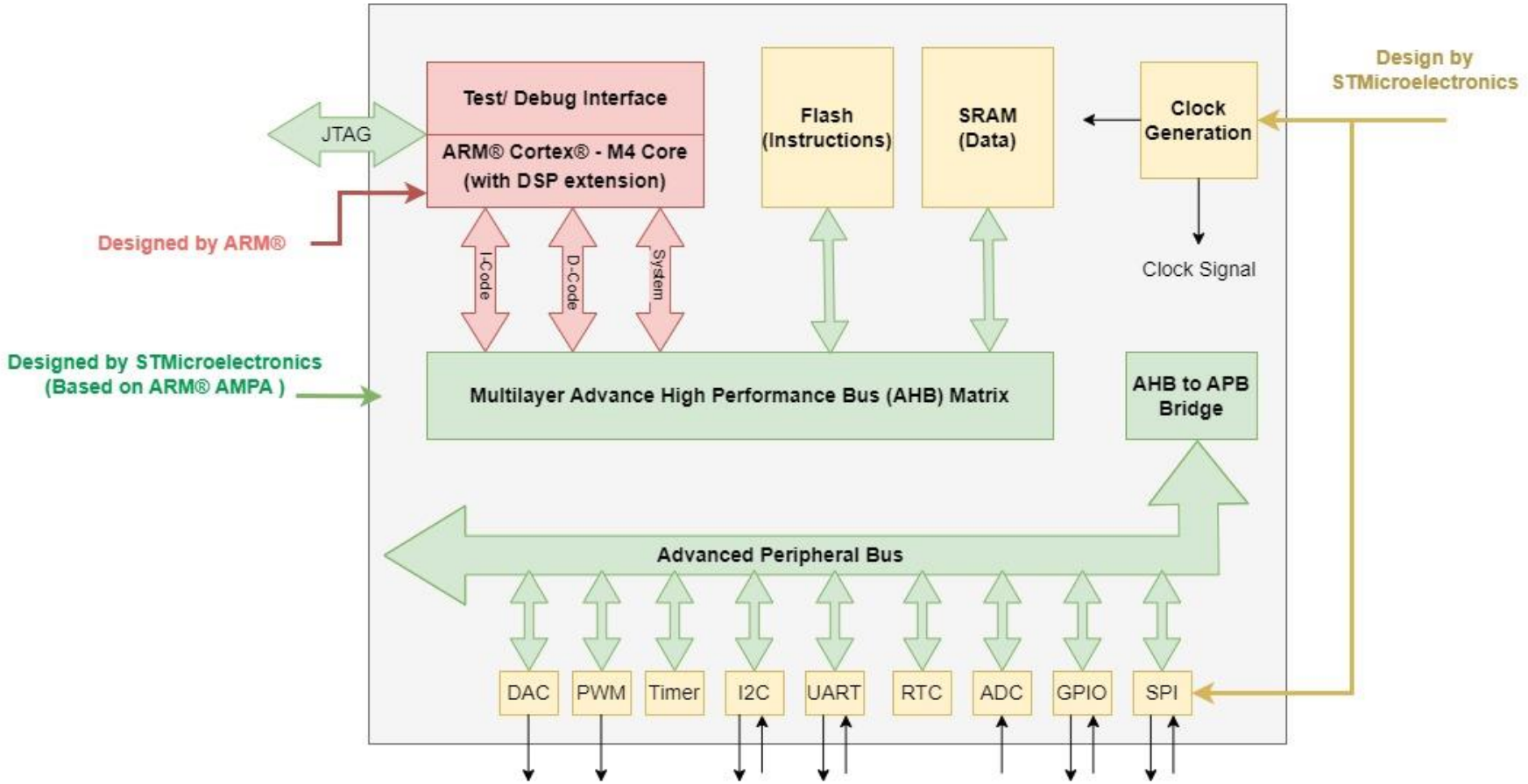
DSP (SIMD, fast MAC)

Advanced data processing
bit field manipulations

General data processing
I/O control tasks

STUDENTS-HUB.com

Microcontroller Architecture base ARM® Cortex®



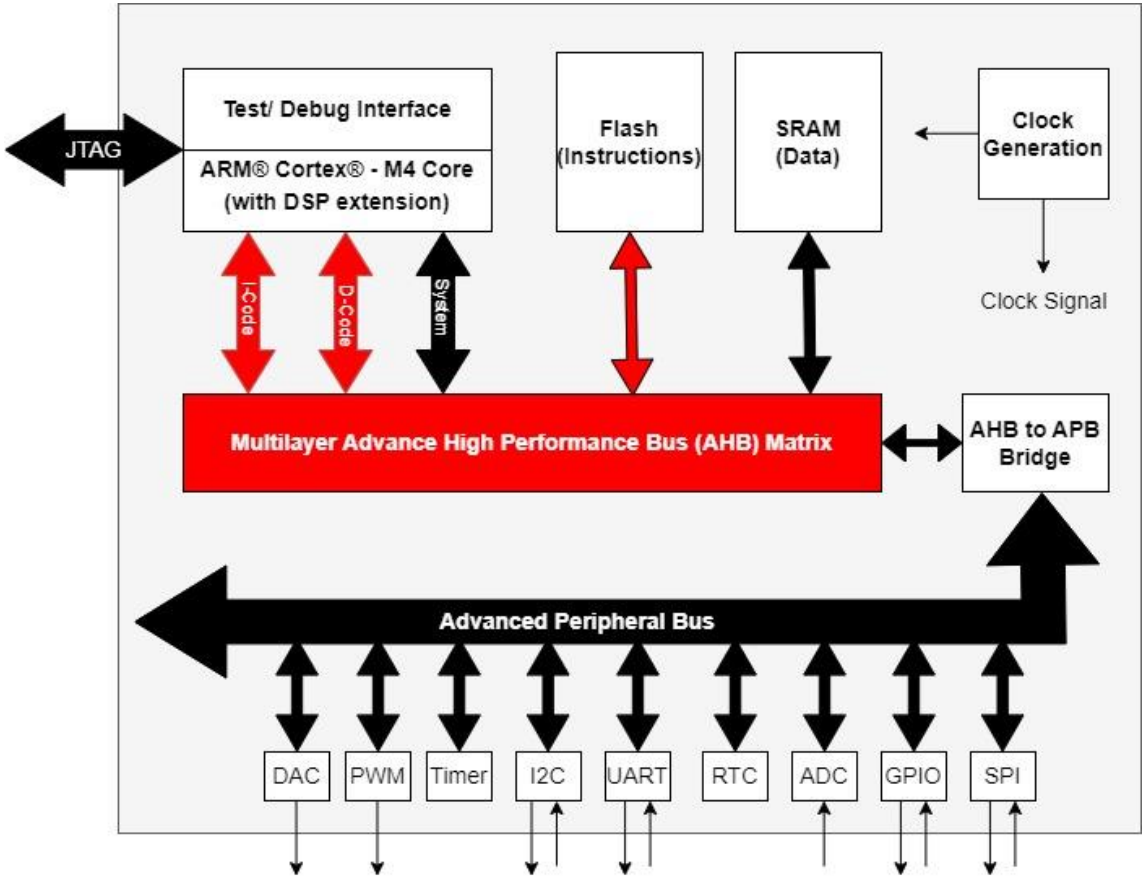
STUDENTS-HUB.com

ARM® Cortex® M4 Harvard architecture

- Microcontrollers are typically Harvard machines
- In ARM Cortex M4 processor, instructions are stored in Flash memory
- This is connected to the core using **I-Code** and **D-Code** buses
- Data is stored in SRAM
- This is connected to the core using the **System** bus **(S-bus)**
- The system bus is also used to connect the core to peripheral

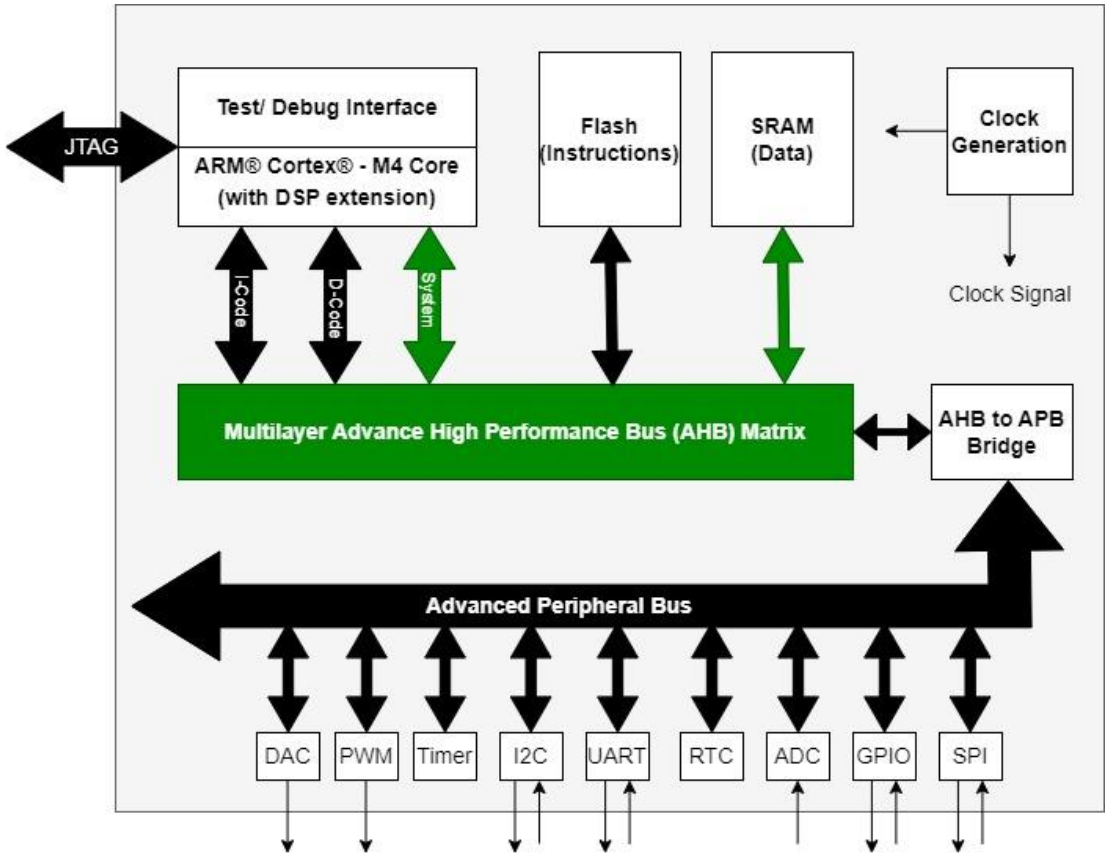
STUDENTS-HUB.com

ARM® Cortex® M4 Harvard architecture



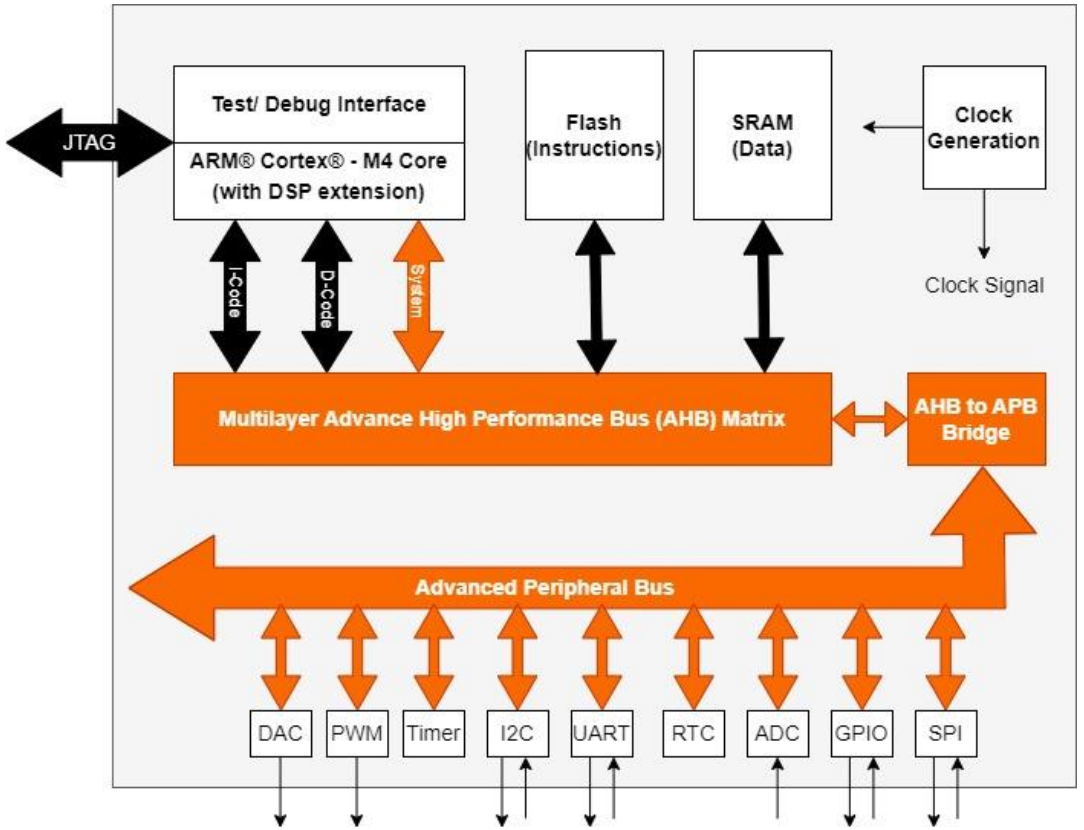
STUDENTS-HUB.com

ARM® Cortex® M4 Harvard architecture



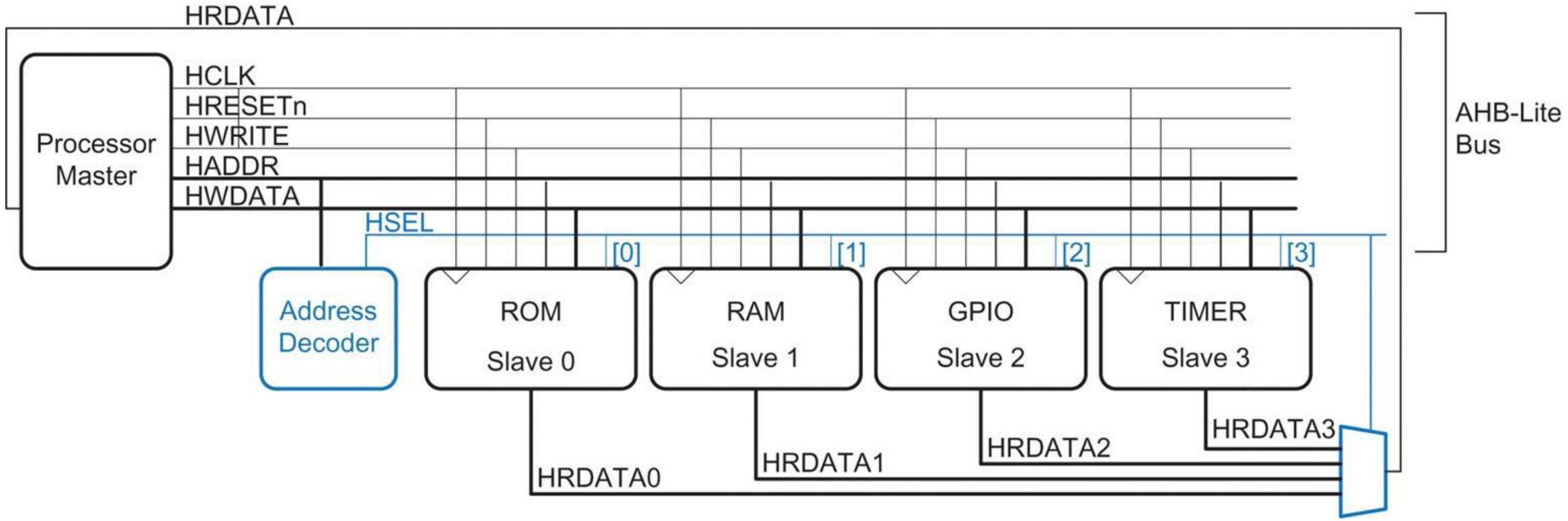
STUDENTS-HUB.com

ARM® Cortex® M4 Harvard architecture



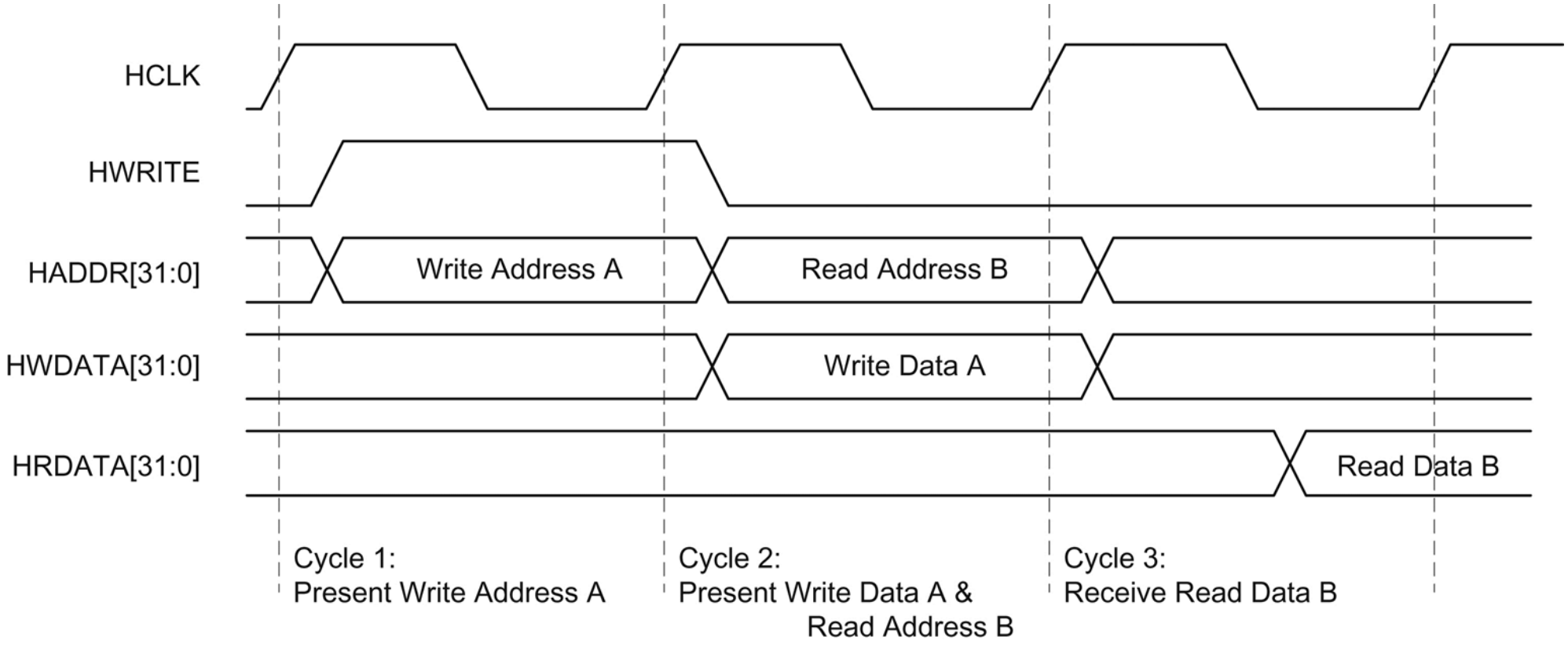
STUDENTS-HUB.com

AHB-Lite Bus



STUDENTS-HUB.com

AHB-Lite Bus



STUDENTS-HUB.com

ARM® Cortex® M Bus interfaces

- The processor contains four external Advanced High-performance Bus (AHB)-Lite bus interfaces:

1. **ICode memory interface**
   - Instruction fetches from Code memory space, 0x00000000 to 0x1FFFFFFF, are performed over this 32-bit AHB-Lite bus.
   - The Debugger cannot access this interface.
   - All fetches are **word-wide**.
   - The number of instructions fetched per word depends on the code running and the alignment of the code in memory.
2. **DCode memory interface**
   - Data and debug accesses to Code memory space, 0x00000000 to 0x1FFFFFFF, are performed over this 32-bit AHB-Lite bus.
   - Core data accesses have a higher priority than debug accesses on this bus.
   - This means that debug accesses are waited until core accesses have completed when there are simultaneous core and debug access to this bus.

STUDENTS-HUB.com

ARM® Cortex® M Bus interfaces

3. **System interface**
   - Instruction fetches, and data and debug accesses, to address ranges 0x20000000 to 0xDFFFFFFF and 0xE0100000 to 0xFFFFFFFF are performed over this 32-bit AHB-Lite bus.
   - For simultaneous accesses to this bus, the arbitration order in decreasing priority is:
     - data accesses
     - instruction and vector fetches
     - debug.
4. **Private Peripheral Bus (PPB)**
   - Data and debug accesses to external PPB space, 0xE0040000 to 0xE00FFFFF, are performed over this 32-bit Advanced Peripheral Bus (APB) bus.

STUDENTS-HUB.com

Memory Space

- The peripherals are separate circuits that enable the microcontroller to interface with sensors and actuators etc.
- Peripherals contain **registers** that are used to configure the peripherals
- They are **memory-mapped**, which means they are given an address and appear to be part of the same memory as instructions and data
- The Flash memory and RAM also appear to be in the same memory despite being physically separate
- This is known as a **unified memory map**

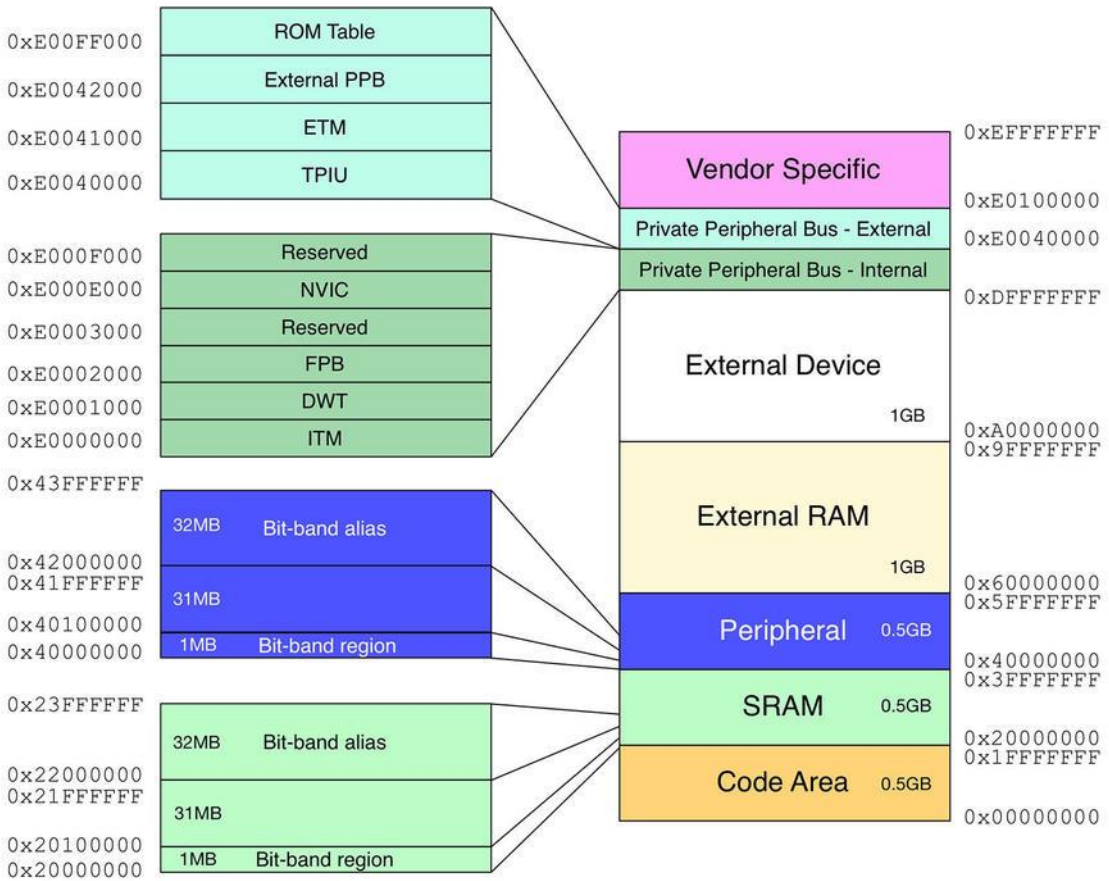
STUDENTS-HUB.com

Memory Map

- ARM defines a standardized memory address space common to all Cortex-M cores, which ensures code portability among different silicon manufacturers.
- The address space is 4 GB wide (due to the 32-bit address line)
- It is organized into several subregions with different logical functionalities.

STUDENTS-HUB.com

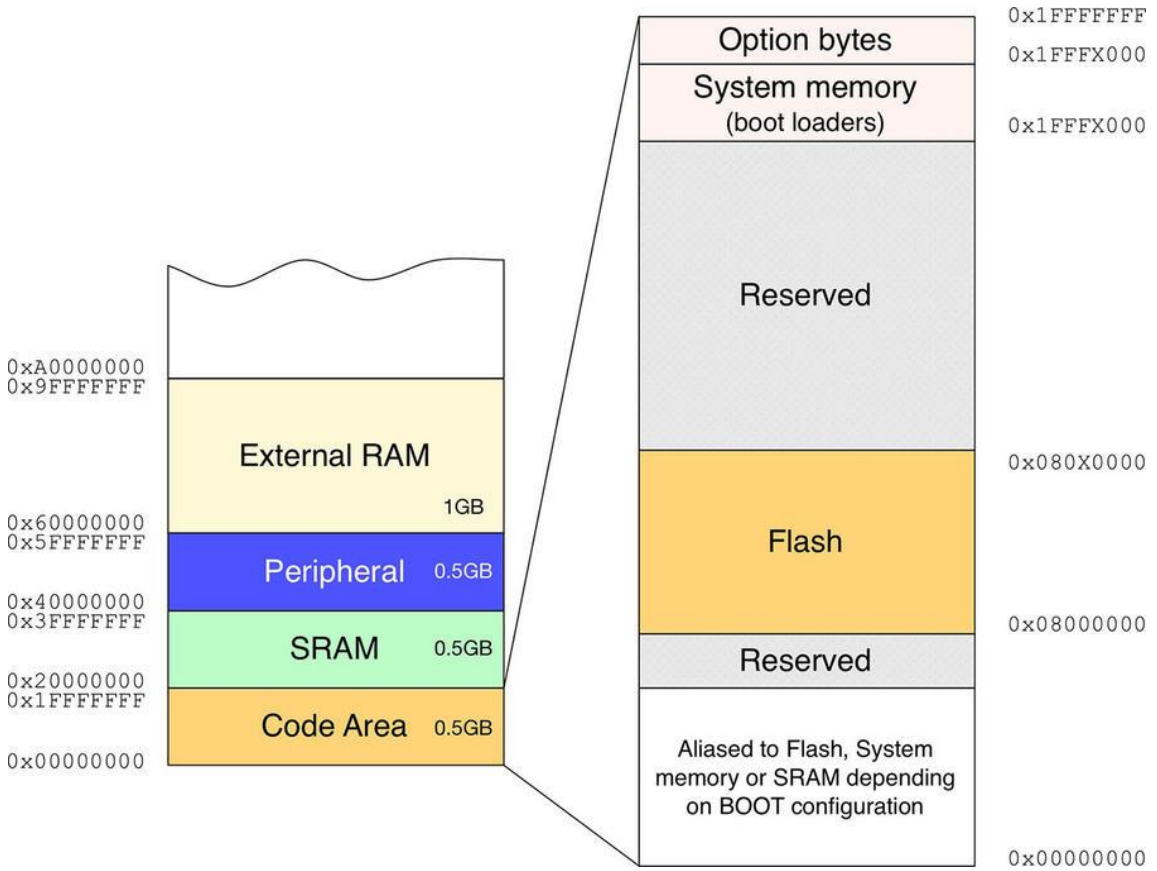
Memory Map



STUDENTS-HUB.com

Memory Map

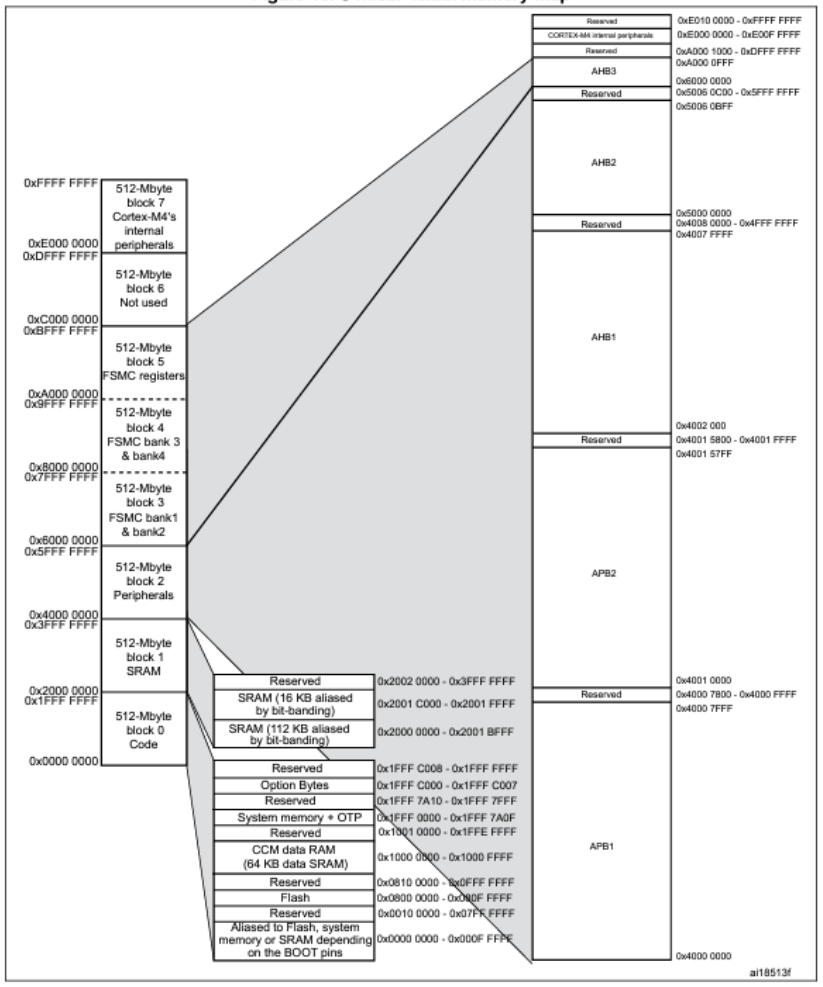
- Memory map for code area



STUDENTS-HUB.com

Memory Map- STM32F40xx Example

Figure 18. STM32F40xxx memory map



STUDENTS-HUB.com

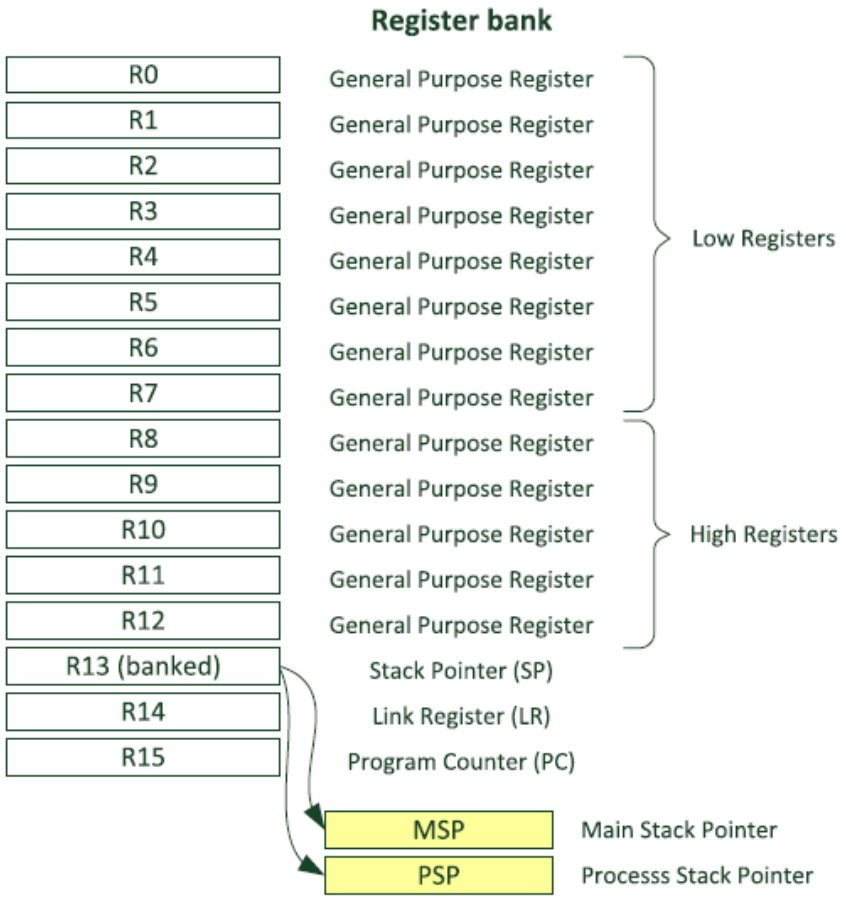
Cortex-M Core Registers

- The Cortex-M3 and Cortex-M4 processors have a number of registers inside the processor core to perform data processing and control
- Most of these registers are grouped in a unit called the register bank.
- Each data processing instruction specifies the operation required, the source register(s), and the destination register(s) if applicable.
- In the ARM architecture, if data in memory is to be processed, it has to be loaded from the memory to registers in the register bank, processed inside the processor, and then written back to the memory, if needed.
- This is commonly called a “**load-store architecture**".

STUDENTS-HUB.com

Cortex-M Core Registers

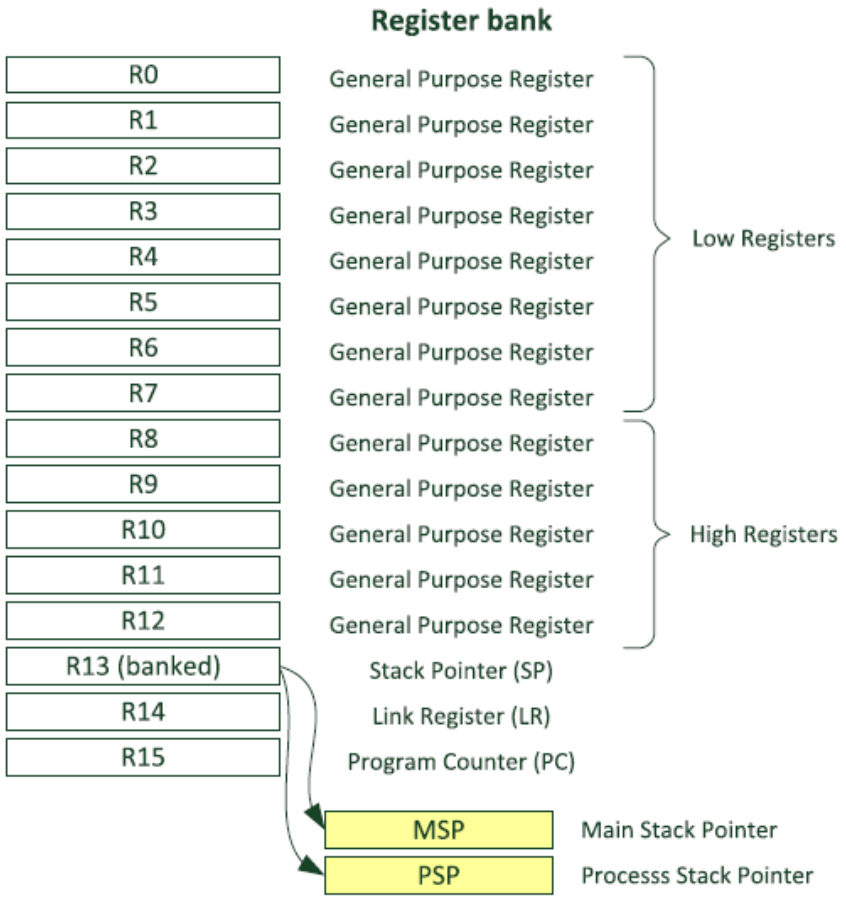
- The register bank in the Cortex-M3 and Cortex-M4 processors has 16 registers.
- Thirteen of them are general-purpose 32-bit registers, and the other three have special uses



STUDENTS-HUB.com

Cortex-M Core Registers

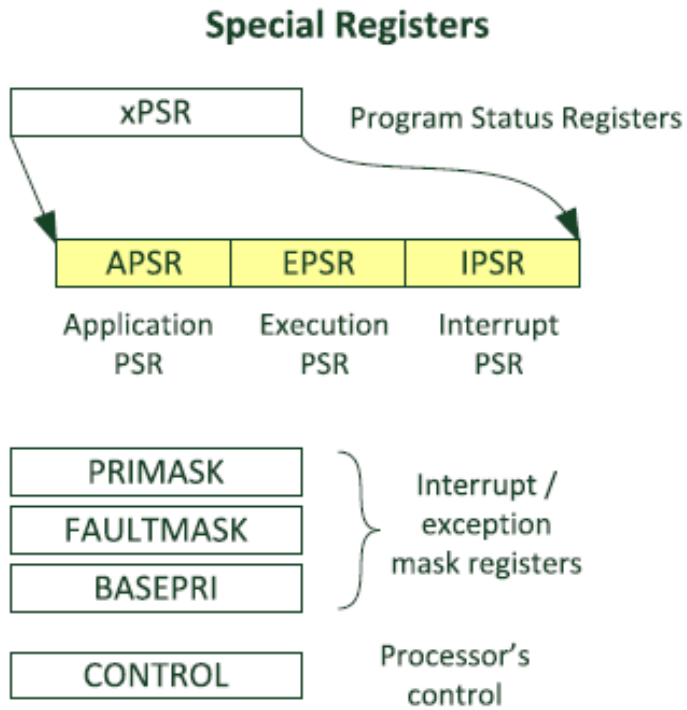
- **R0- R12** are **general-purpose** registers, and can be used as operands for ARM instructions
- **R13** is the **Stack Pointer (SP)** register, it is used for accessing the stack memory via PUSH and POP operations.
- **R14** is the **Link Register (LR)** register, which is a special-purpose register which holds the address to return to when a function call completes.
- The linker register does not require the writes and reads of the memory containing the stack
- **R15** is the **Program Counter (PC)** register, which has the address of the next instruction to be executed from memory.



STUDENTS-HUB.com

Cortex-M Core Registers

- Besides the registers in the register bank, there are a number of special registers
- These registers contain the processor status and define the operation states and interrupt/exception masking.
- In the development of simple applications with high level programming languages such as C, there are not many scenarios that require access to these registers.
- However, they are needed for development of an embedded OS, or when advanced interrupt masking features are needed.



STUDENTS-HUB.com

Instruction Set Architecture

- Where the processor stores or obtain information
  - Registers
  - Memory
  - Input/ output devices (Memory/ special instructions)
- How the processor manipulate information
  - Assembly language instructions
  - Processor hardware actions

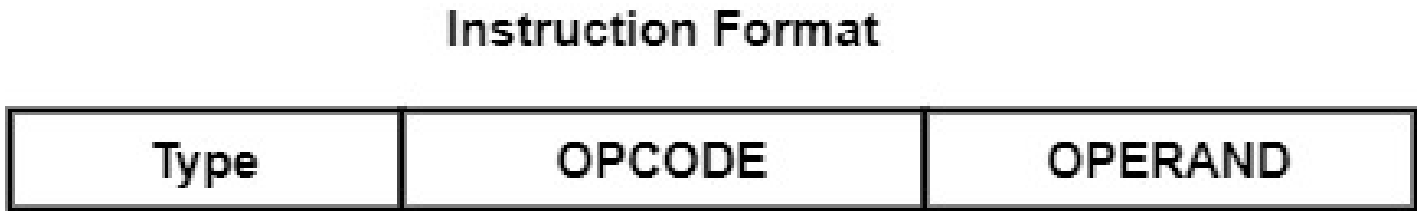
STUDENTS-HUB.com

Instruction Set Architecture

- A microcontroller is an example of a **reduced instruction set computer** (**RISC**) that has a fairly limited number of instructions
- Generally, a **mnemonic** is a symbolic name for a single executable machine language instruction
- mnemonics are used to specify an **opcode**
- An assembler is used to convert the assembly code to machine code (0's and 1's) that are stored in memory
- program often we use a high-level language (C, C++, etc.) and a compiler translates it into machine code

STUDENTS-HUB.com

Instruction Format



- **Operation code** or **op-code** is a part of the instruction that tells the processor what should be done.
- **Operand** is a part of the instruction that contains the data to be acted on, or the memory location of the data in a register.
- https://developer.arm.com/documentation/ddi0403/d

STUDENTS-HUB.com

Instruction Format

Mnemonic	Opcode	Action
AND	0000	Bitwise AND
EOR	0001	Bitwise Exclusive OR
LSL	0010	Logical Shift Left
LSR	0011	Logical Shift Right
ASR	0100	Arithmetic Shift Right
ADC	0101	Add with Carry
SBC	0110	Subtract with Carry
ROR	0111	Rotate Right
TST	1000	Test
RSB	1001	Reverse Subtract from 0
CMP	1010	Compare High Registers
CMN	1011	Compare Negative
ORR	1100	Bitwise OR
MUL	1101	Multiply Two Registers
BIC	1110	Bitwise Bit Clear
MVN	1111	Bitwise NOT

STUDENTS-HUB.com

Instruction Format

- The type of instruction can be:
  - Data processing instructions and miscellaneous instructions
  - Load/ Store instructions
  - Branch instructions
  - Media instructions
  - Coprocessor instructions
  - Floating point instructions

STUDENTS-HUB.com

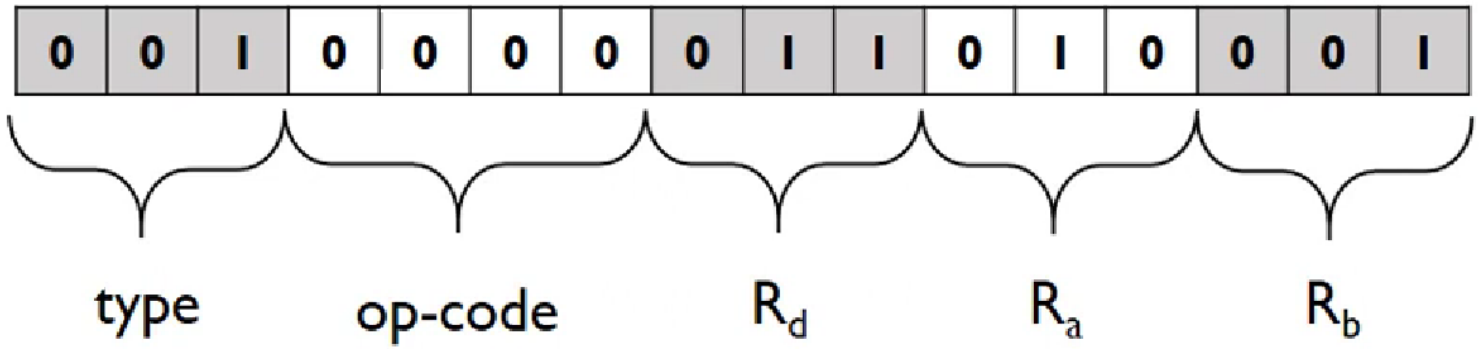
Instruction Format

- ARM instructions process data held in registers and only access memory with load and store instructions
- ARM instructions commonly take two or three operands

Instruction Syntax	Destination register (*Rd*)	Source register 1 (*Rn*)	Source register 2 (*Rm*)
ADD r3, r1, r2	*r3*	*r1*	*r2*

STUDENTS-HUB.com

Instruction Format (Example)



Data operation $R_d = R_a$ *operation* R_b `ADD R3, R2, R1`

STUDENTS-HUB.com

Addressing Modes (ARM)

- The phrase addressing modes in ARM relates to the manner an instruction operand is expressed.
- Before the operand is actually performed, the addressing mode provides a rule for interpreting or altering the address field of the instruction.

STUDENTS-HUB.com

Addressing Modes (ARM)

- The address for a load or store is formed from two parts:
  - A value from a base register

    LDR R0, [R1] (This instruction loads the 32-bit word at the memory location contained in register R1 into register R0)
  - An offset.
- The base register can be any one of the general-purpose registers. For loads, the base register can be the PC. This permits PC-relative addressing for position-independent code. Instructions marked (literal)

STUDENTS-HUB.com

Addressing Modes (ARM)

- The offset takes one of three formats:
  1. Immediate addressing offset
  2. Register addressing offset
  3. Scaled register-indexed addressing offset
- The offset and base register can be used in three different ways to form the memory address. The addressing modes are:
  - Pre-indexed
  - Post-indexed

STUDENTS-HUB.com

Addressing Modes (ARM)

Immediate addressing offset

- The offset is an unsigned number that can be added to or subtracted from the base register value.
- Immediate offset addressing is useful for accessing data elements that are a fixed distance from the start of the data object, such as structure fields, stack offsets, and input/output registers.
- e.g.

  LDR R0, [R1, #4]

  (This instruction loads the register R0 with the word at the memory regions computed by adding the constant address included in the R1 register value 4 to the memory address stored in the R1 register)

STUDENTS-HUB.com

Addressing Modes (ARM)

Register addressing offset

- The offset is a value from a general-purpose register.
- This register cannot be the PC.
- The value can be added to, or subtracted from, the base register value.
- Register offsets are useful for accessing arrays or blocks of data.
- e.g.

  LDR R0, [R1, R2]

  (This instruction will load the word at the memory address determined by adding register R1 and register R2 into register R0.)

STUDENTS-HUB.com

Addressing Modes (ARM)

- **Scaled register-indexed addressing offset**
- The offset is a general-purpose register, other than the PC, shifted by an immediate value, then added to or subtracted from the base register.
- This means an array index can be scaled by the size of each array element.
  - LDR R8, [R9, R10, LSL #2] (loads the value from the memory location pointed to by R9 + (R10 * 4) into R8)

STUDENTS-HUB.com

Addressing Modes (ARM)

- **Offset**: The offset is added to or subtracted from the base register to form the memory address
- **Pre-indexed**:
  - Post-indexed addressing updates the base register before memory access
  - LDR R1, [R0, #8] (load the value from the memory location pointed to by R0 + 8 into R1)
- **Post-indexed**:
  - Post-indexed addressing updates the base register after memory access
  - LDR R6, [R7], #8 loads the value from the memory location pointed to by R7 into R6 and then increments the value in R7 by 8.

STUDENTS-HUB.com

ARM Cortex M Assembly Programming

- An **assembly language** is a type of **low-level** programming language that is intended to communicate directly with a computer's hardware. Unlike **machine language**, which consists of binary and hexadecimal characters, assembly languages are designed to be readable by humans
- Why Learn Assembly Language?
- Learning and spending some time working at the assembly language level provides a richer understanding of the underlying computer architecture. This includes the basic instruction set, processor registers, memory addressing, hardware interfacing, and Input/ Output

STUDENTS-HUB.com

ARM Cortex M Assembly Programming

- The typical instruction format:

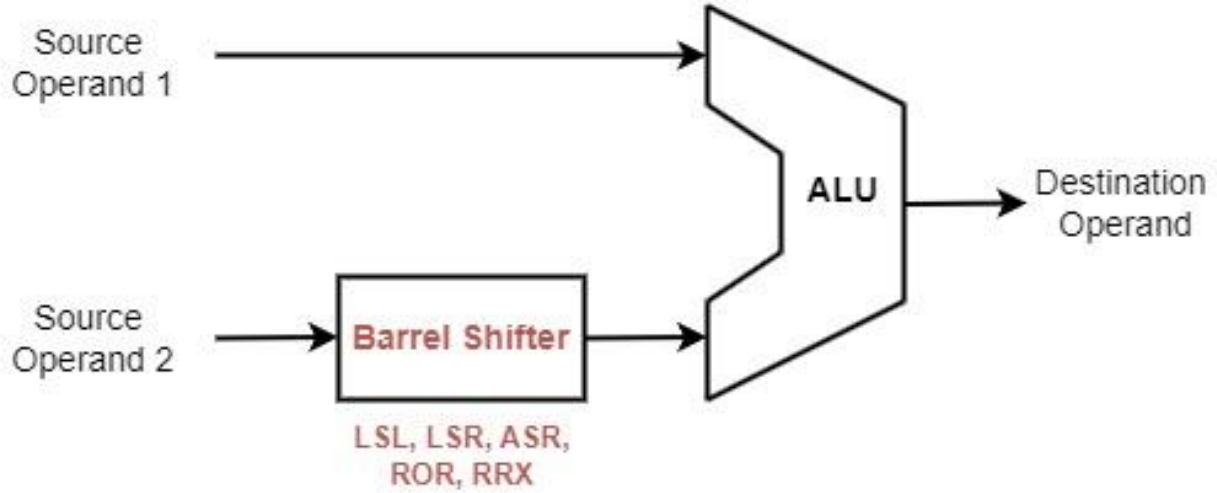
  Opcode DestReg, Operand2

  Opcode DestReg, SrcReg, Operand2

- Instructions may have two or three operands
- First operand is (almost) always a destination register
- **Operand2** is a 'flexible' operand

STUDENTS-HUB.com

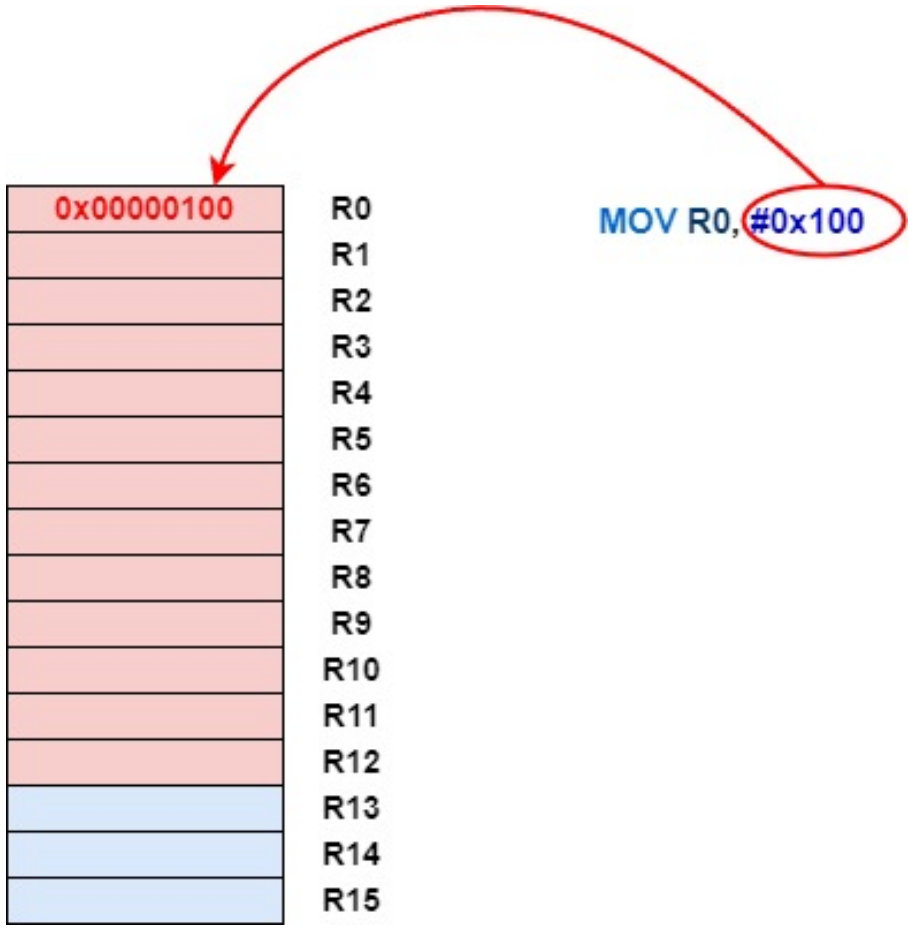
ARM Cortex M Assembly Programming




```
ADD R0, R1 , R2              ; R0 = R1+R2
ADD R0, R1 , #1              ; R0 = R1+1
ADD R0, R1 , R2, LSL #2      ; R0 = R1+R2 << 2
```

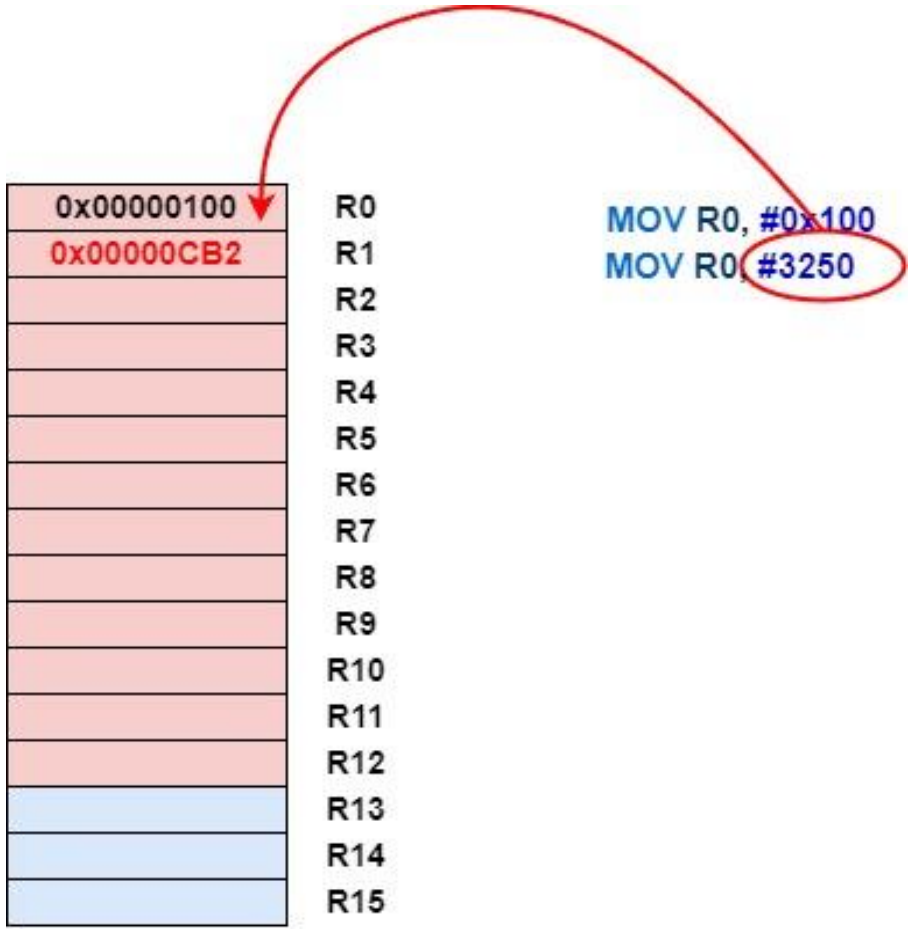
STUDENTS-HUB.com

The Move Instruction



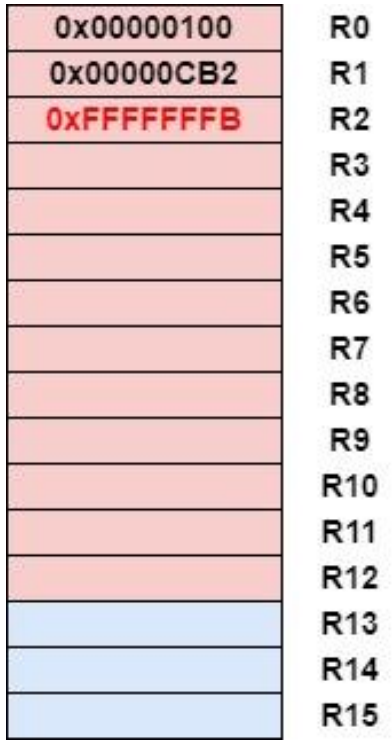
STUDENTS-HUB.com

The Move Instruction



STUDENTS-HUB.com

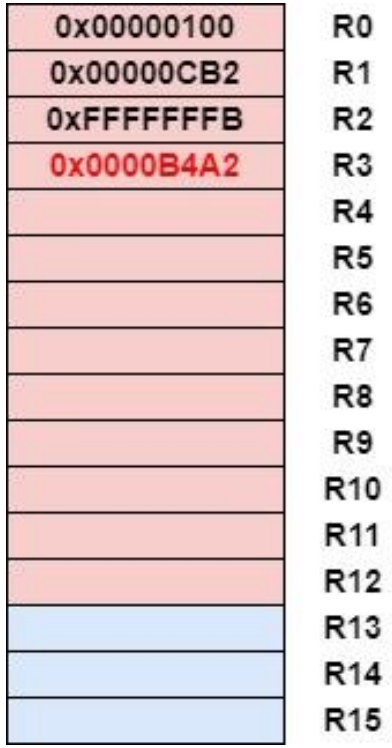
The Move Instruction




```
MOV R0, #0x100
MOV R1, #3250
MVN R2, #4      00....0100
                11....1011
```

STUDENTS-HUB.com

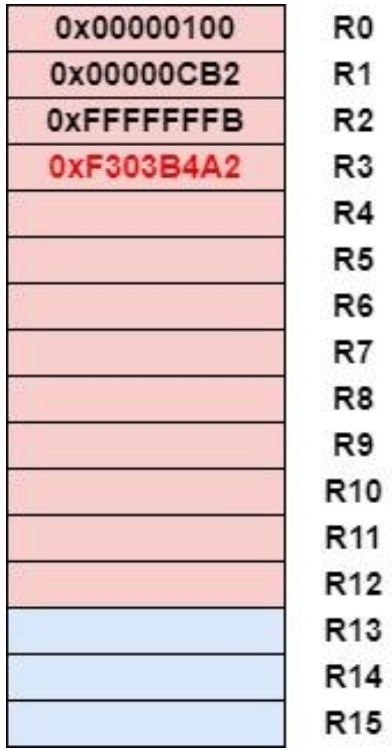
The Move Instruction




```
MOV R0, #0x100
MOV R1, #3250
MVN R2, #4
MOVW R3, #0xB4A2
```

STUDENTS-HUB.com

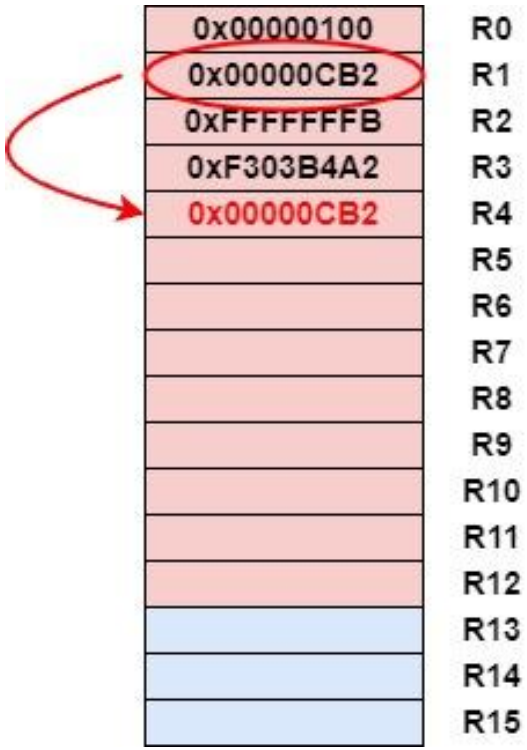
The Move Instruction




```
MOV R0, #0x100
MOV R1, #3250
MVN R2, #4
MOVW R3, #0xB4A2
MOVT R3, #0xF303
```

STUDENTS-HUB.com

The Move Instruction




```
MOV R0, #0x100
MOV R1, #3250
MVN R2, #4
MOVW R3, #0xB4A2
MOVT R3, #0xF303
MOV R4, R1
```

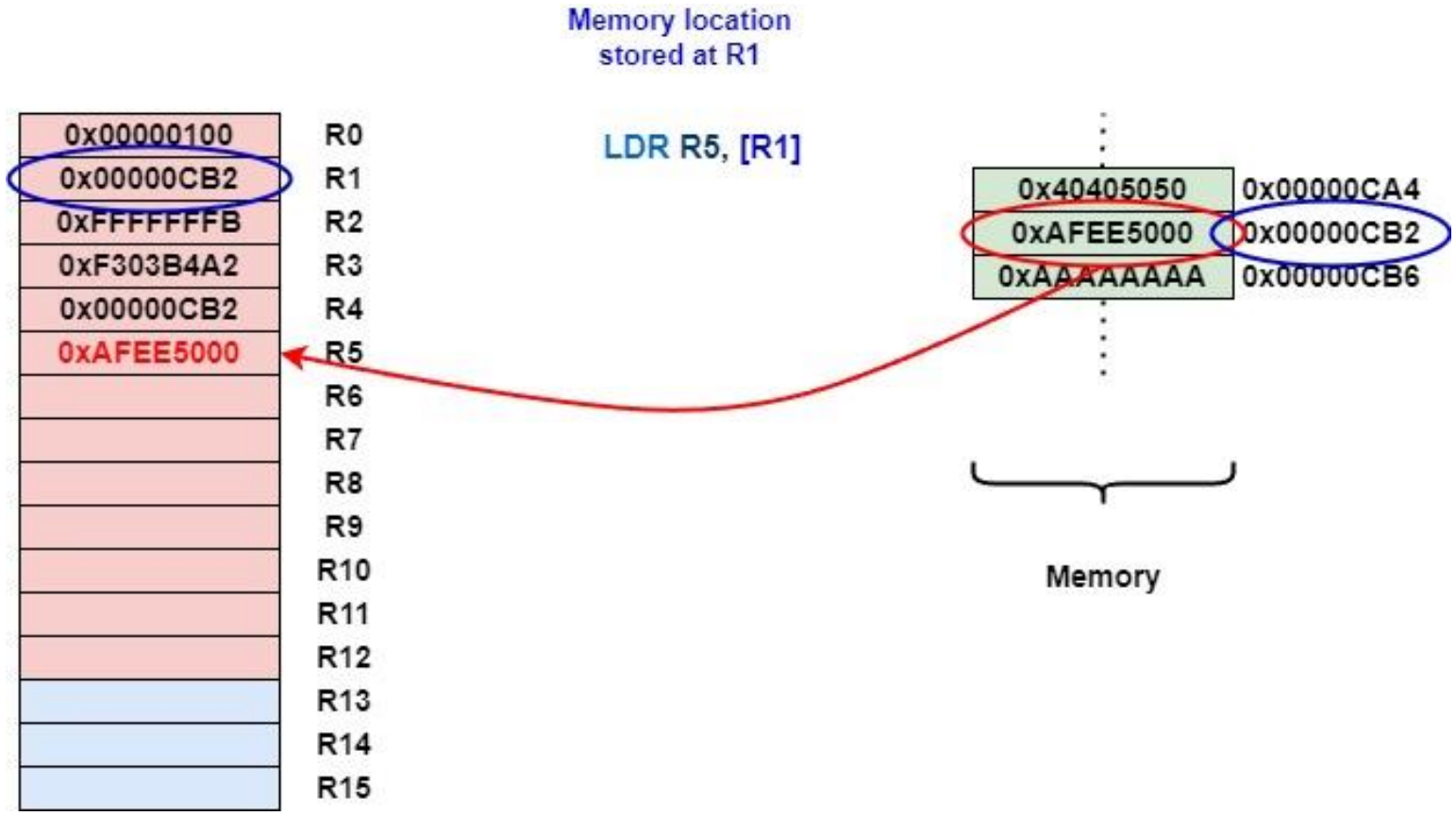
STUDENTS-HUB.com

Summary

- Most instructions specify the **destination** as the first operand
- **Immediate** values are small constants that are encoded as part of instruction itself
- The (optional) **#** prefix marks an immediate operand
- Hexadecimal values have a prefix of **0x**

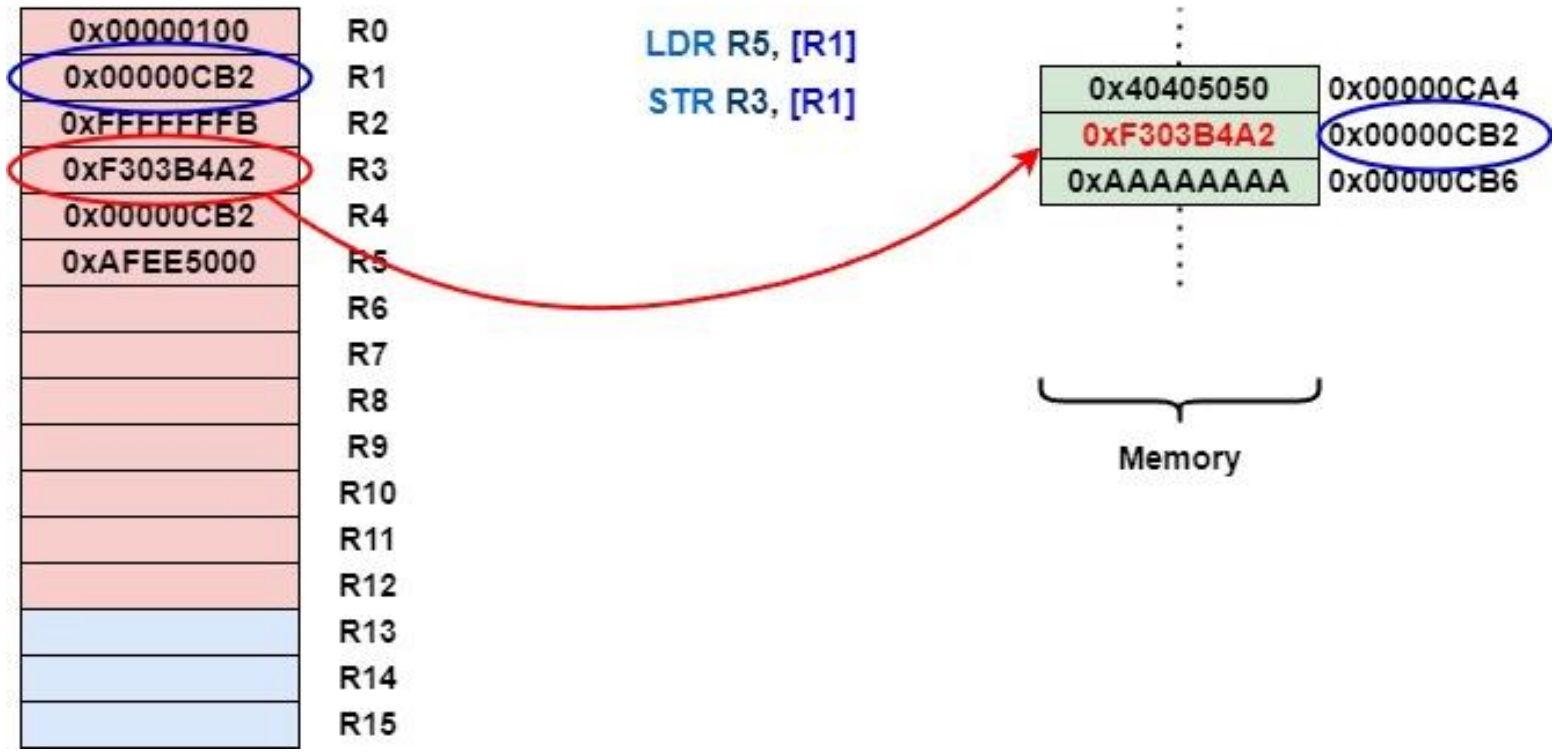
STUDENTS-HUB.com

Load/ Store Instructions



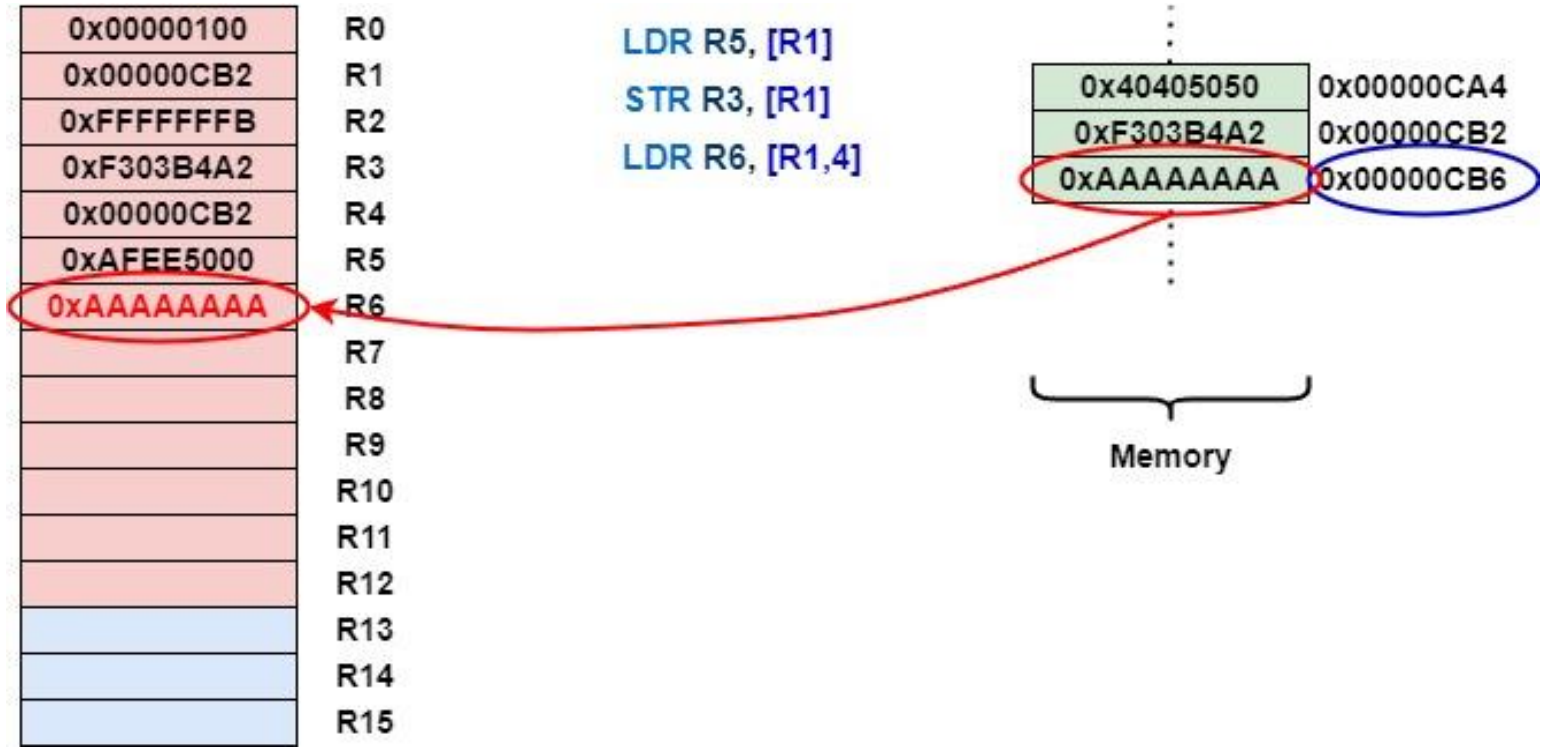
STUDENTS-HUB.com

Load/ Store Instructions



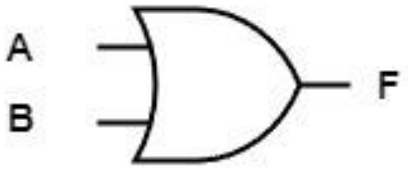
STUDENTS-HUB.com

Load/ Store Instructions



STUDENTS-HUB.com

Bitwise Instructions – OR

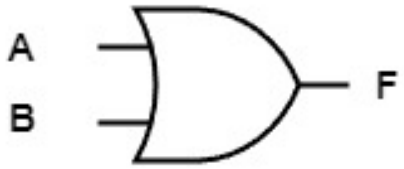


A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

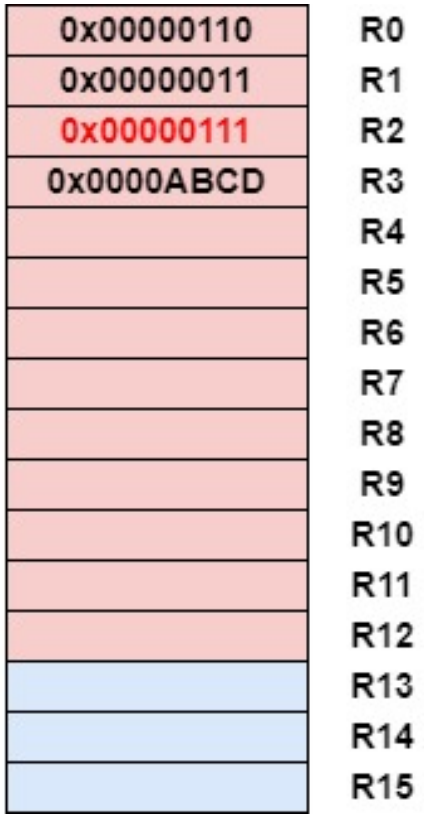

```
MOVW  R0, #0x0110
```

STUDENTS-HUB.com

Bitwise Instructions – OR



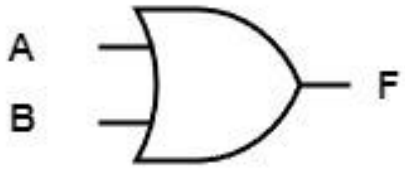
A	B	F
0	0	0
0	1	1
1	0	1
1	1	1



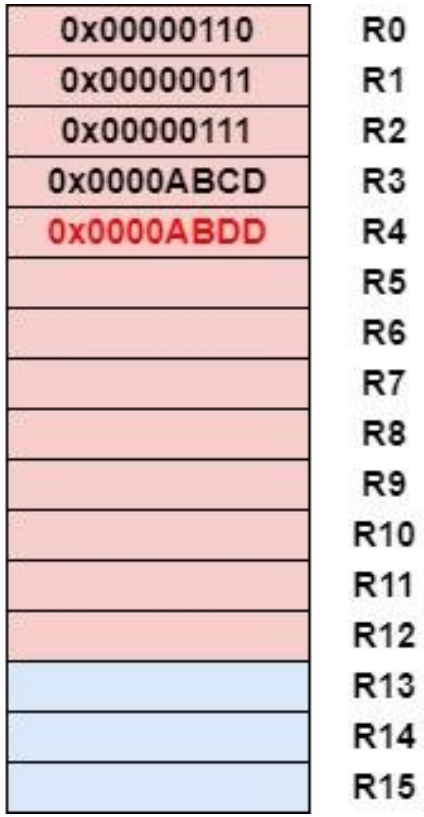

```
MOVW  R0, #0x0110
ORR   R2, R1, R0
```

STUDENTS-HUB.com

Bitwise Instructions – OR



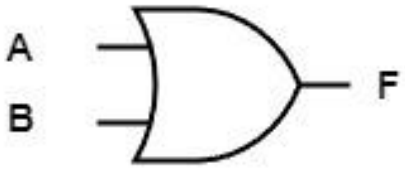
A	B	F
0	0	0
0	1	1
1	0	1
1	1	1




```
MOVW  R0, #0x0110
ORR   R2, R1, R0
ORR   R4, R3, R0
```

STUDENTS-HUB.com

Bitwise Instructions – OR



A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

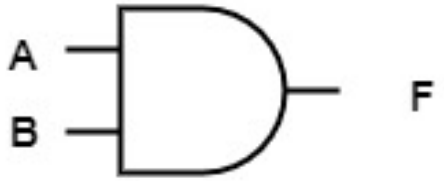

```
MOVW  R0, #0x0110
ORR  R2, R1, R0
ORR  R4, R3, R0
```

In C we would write

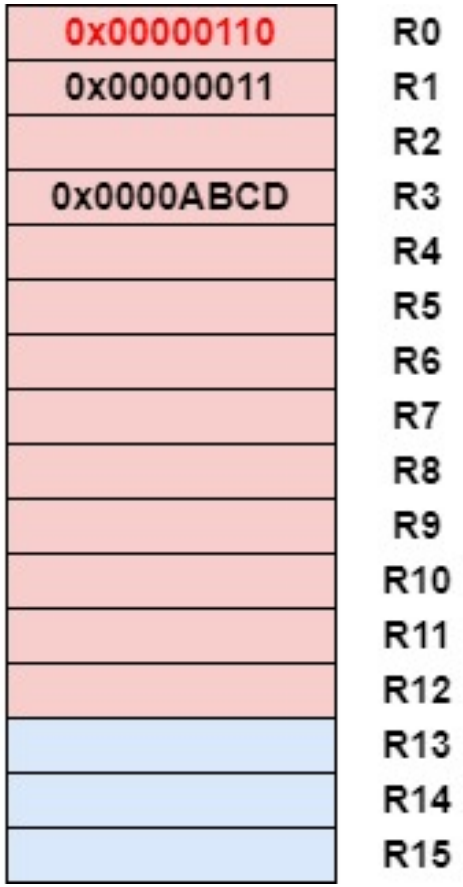
```
#define MASK 0x0110

DestA = SrcA | MASK;
DestB = SrcB | MASL;
```

STUDENTS-HUB.com

Bitwise Instructions – AND



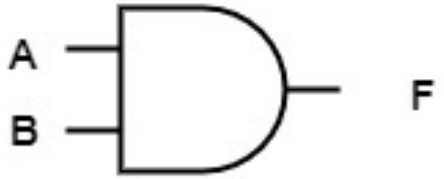
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1



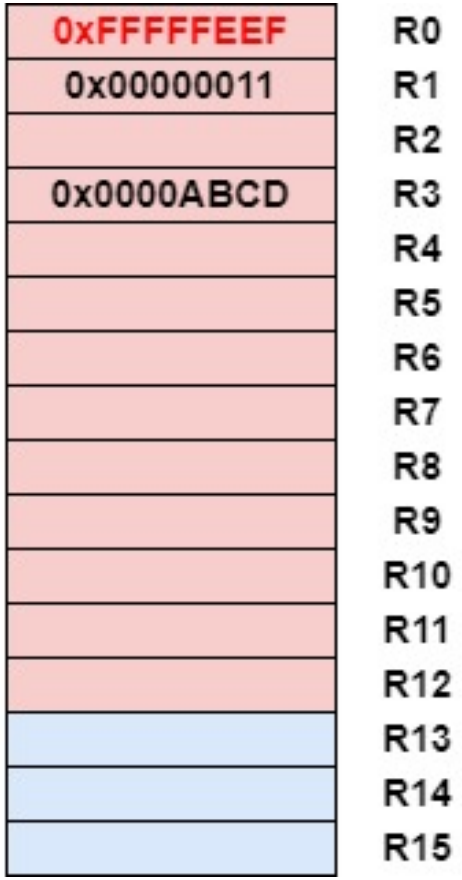

```
MOVW  R0, #0x0110
```

STUDENTS-HUB.com

Bitwise Instructions – AND



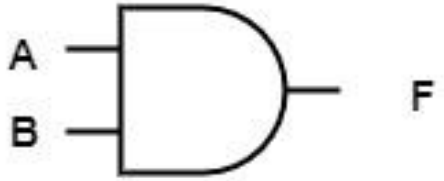
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1



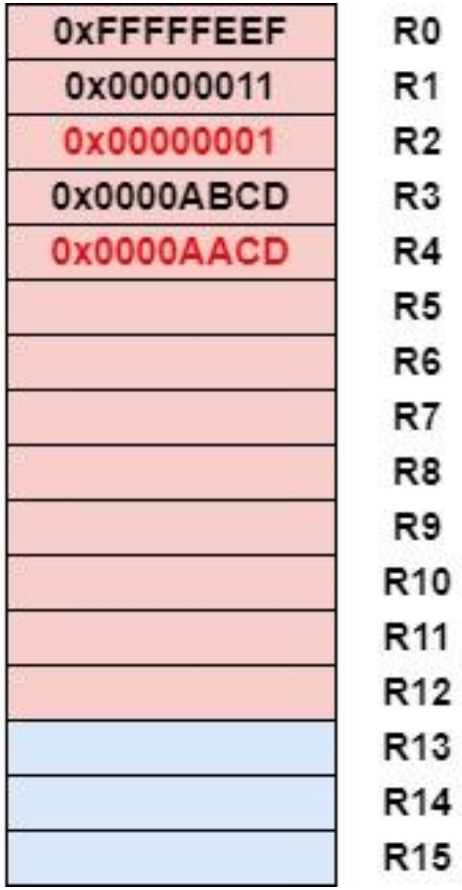

```
MOVW  R0, #0x0110
MVN   R0, R0
```

STUDENTS-HUB.com

Bitwise Instructions – AND



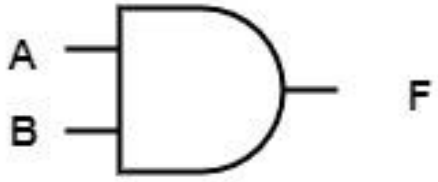
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1



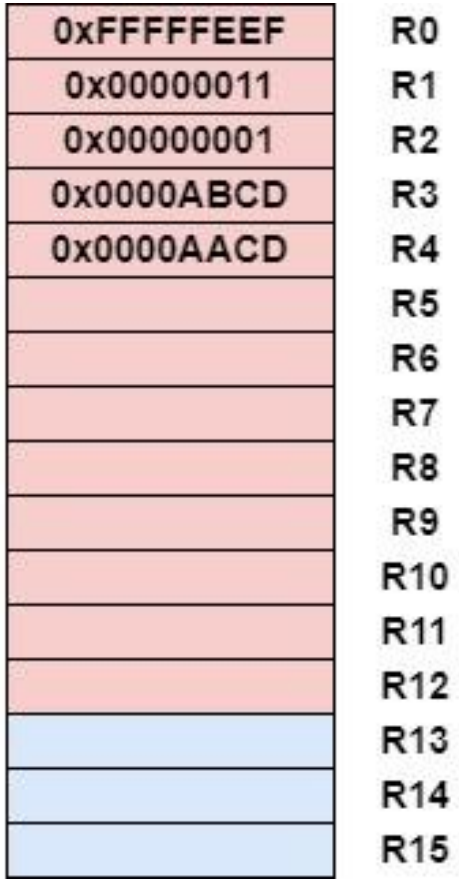

```
MOVW  R0, #0x0110
MVN   R0, R0
AND   R2, R1, R0
AND   R4, R3, R0
```

STUDENTS-HUB.com

Bitwise Instructions – AND



A	B	F
0	0	0
0	1	0
1	0	0
1	1	1



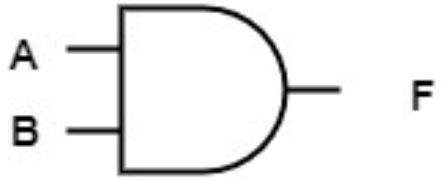

```
MOVW  R0, #0x0110
MVN   R0, R0
AND   R2, R1, R0
AND   R4, R3, R0
```

In C we would write

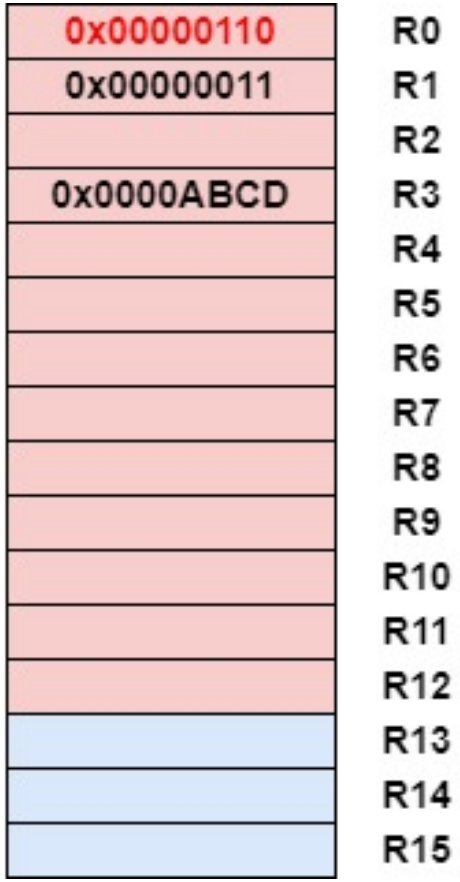
```
#define MASK 0x0110

DestA = SrcA & ~ MASK;
DestB = SrcB & ~MASL;
```

STUDENTS-HUB.com

Bitwise Instructions – BIC



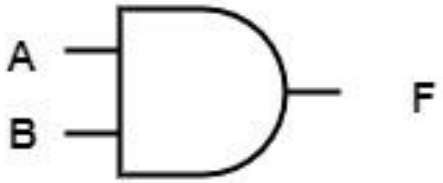
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1



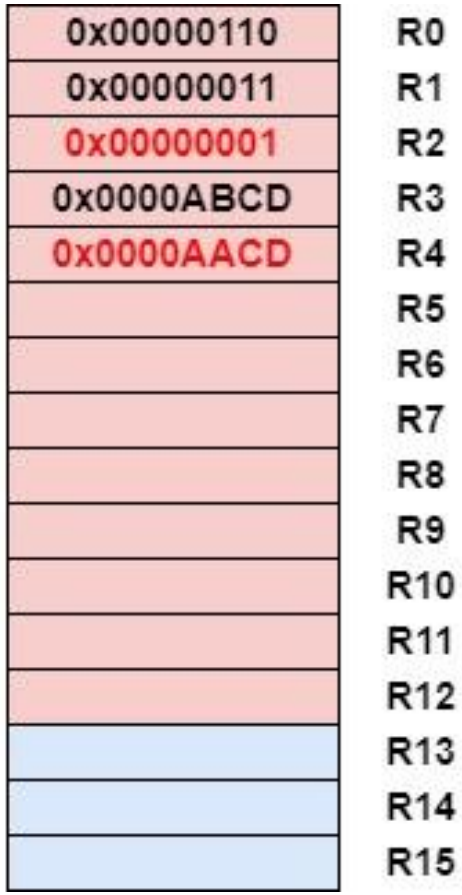

```
MOVW  R0, #0x0110
```

STUDENTS-HUB.com

Bitwise Instructions – BIC



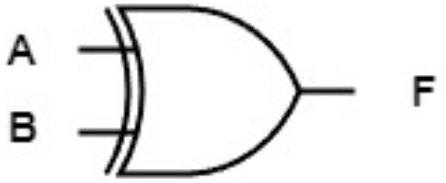
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1



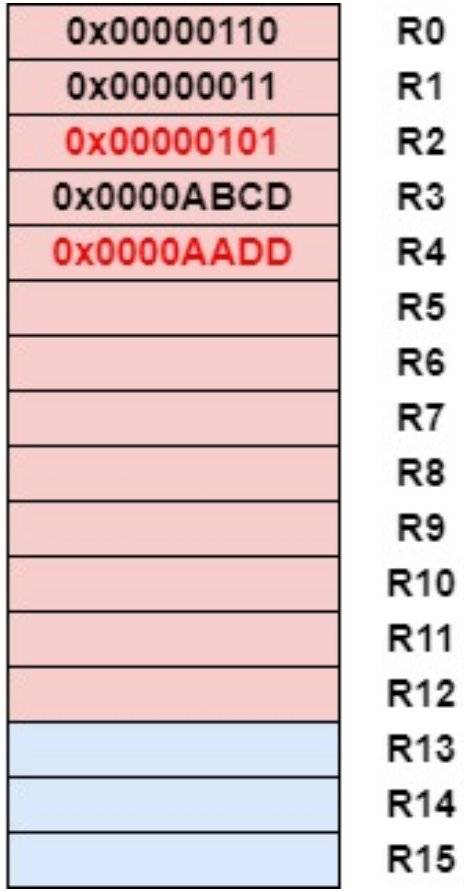

```
MOVW  R0, #0x0110
BIC   R2, R1, R0
BIC   R2, R3, R0
```

STUDENTS-HUB.com

Bitwise Instructions – EOR



A	B	F
0	0	0
0	1	1
1	0	1
1	1	0




```
MOVW  R0, #0x0110
EOR  R2, R1, R0
EOR  R4, R3, R0
```

In C we would write

```
#define MASK 0x0110

DestA = SrcA ^ MASK;
DestB = SrcB ^ MASK;
```

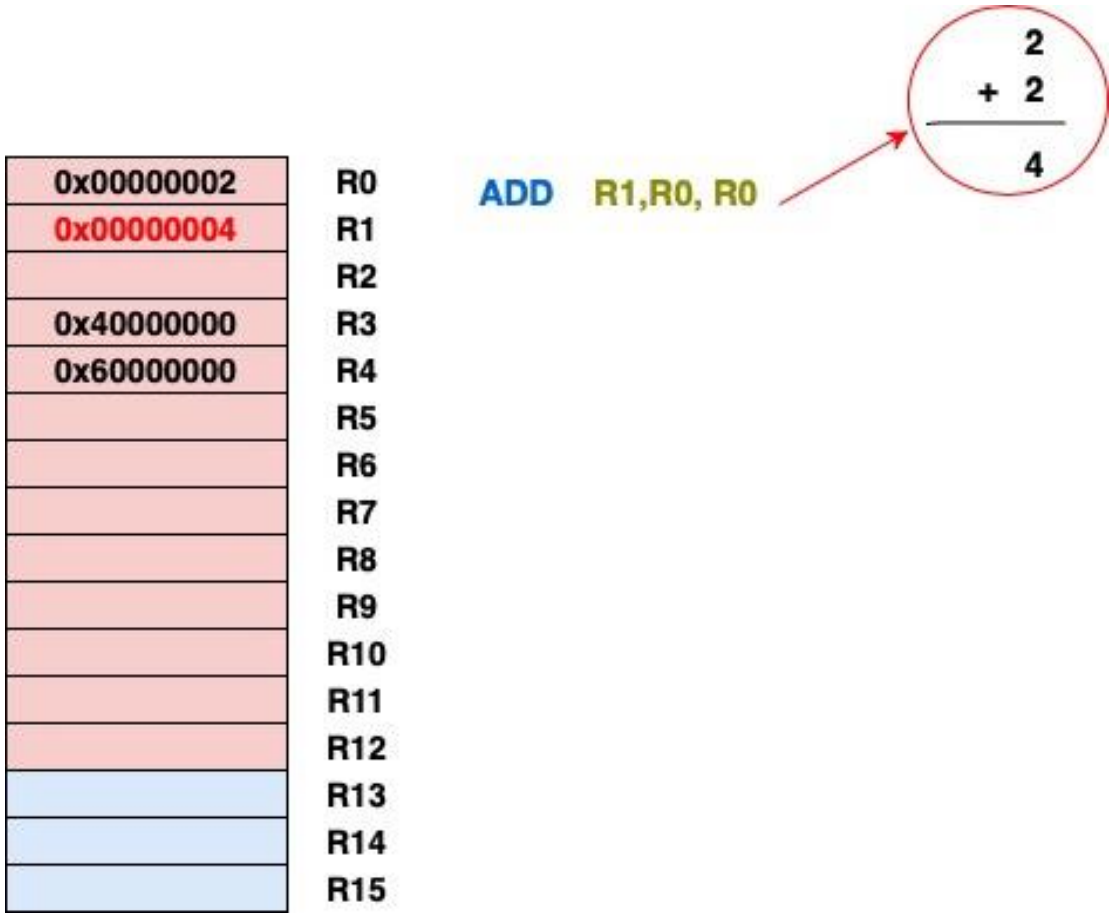
STUDENTS-HUB.com

Summary

- Set bits to 1 in mask to modify corresponding bits
- Use OR instruction to force bits to 1
- Use MVN-AND instructions to force bits to 0
- Use BIC instruction to force bits to 0
- Use EOR instruction to invert bits

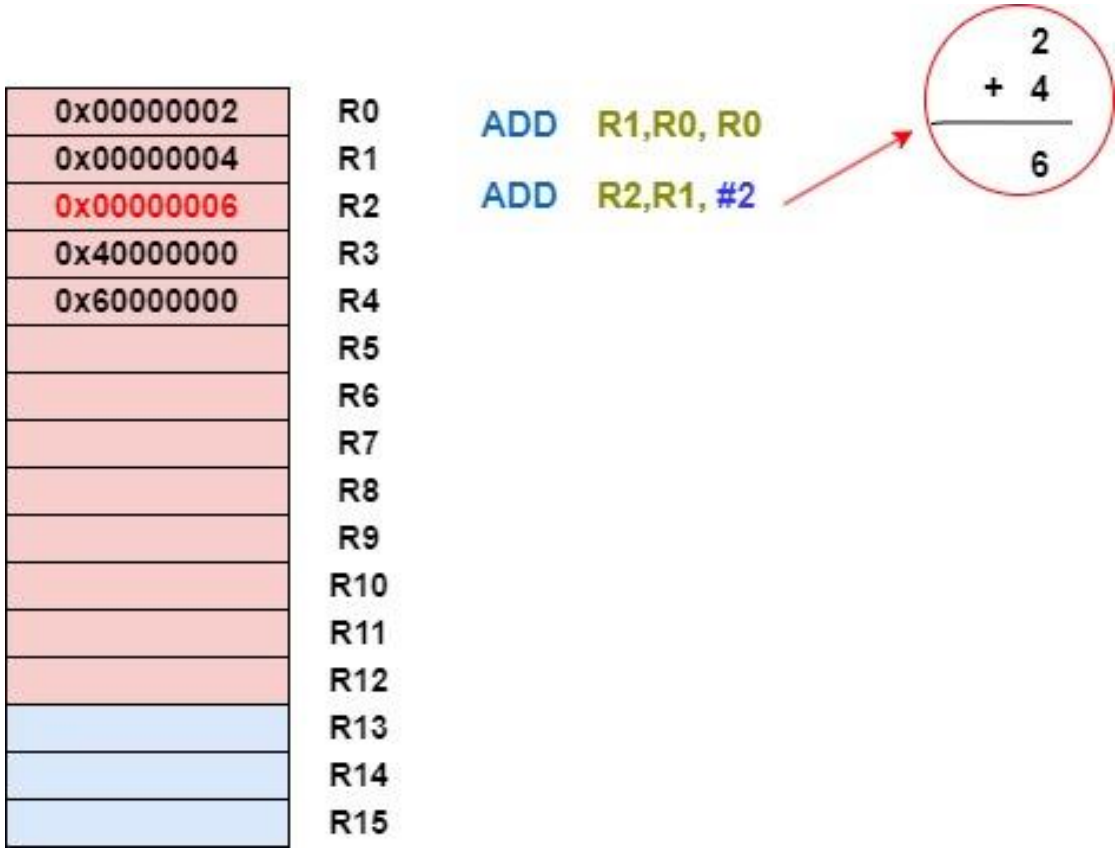
STUDENTS-HUB.com

Arithmetic Instructions- ADD



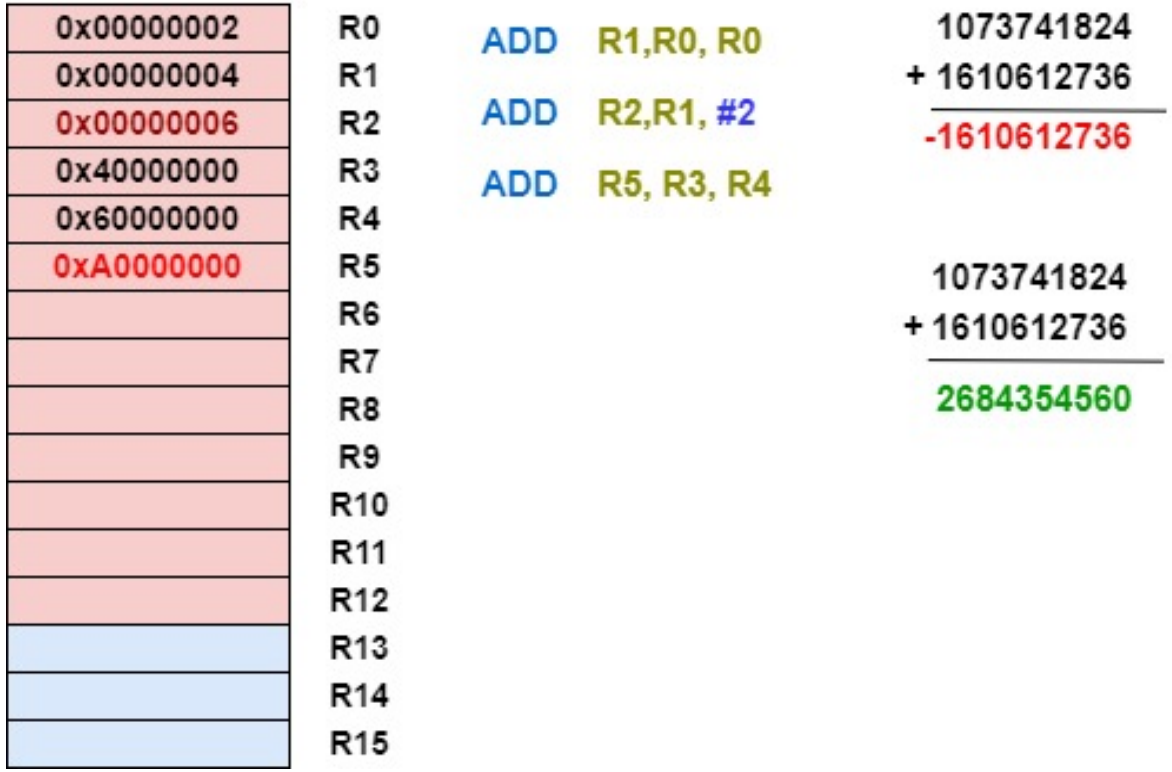
STUDENTS-HUB.com

Arithmetic Instructions- ADD



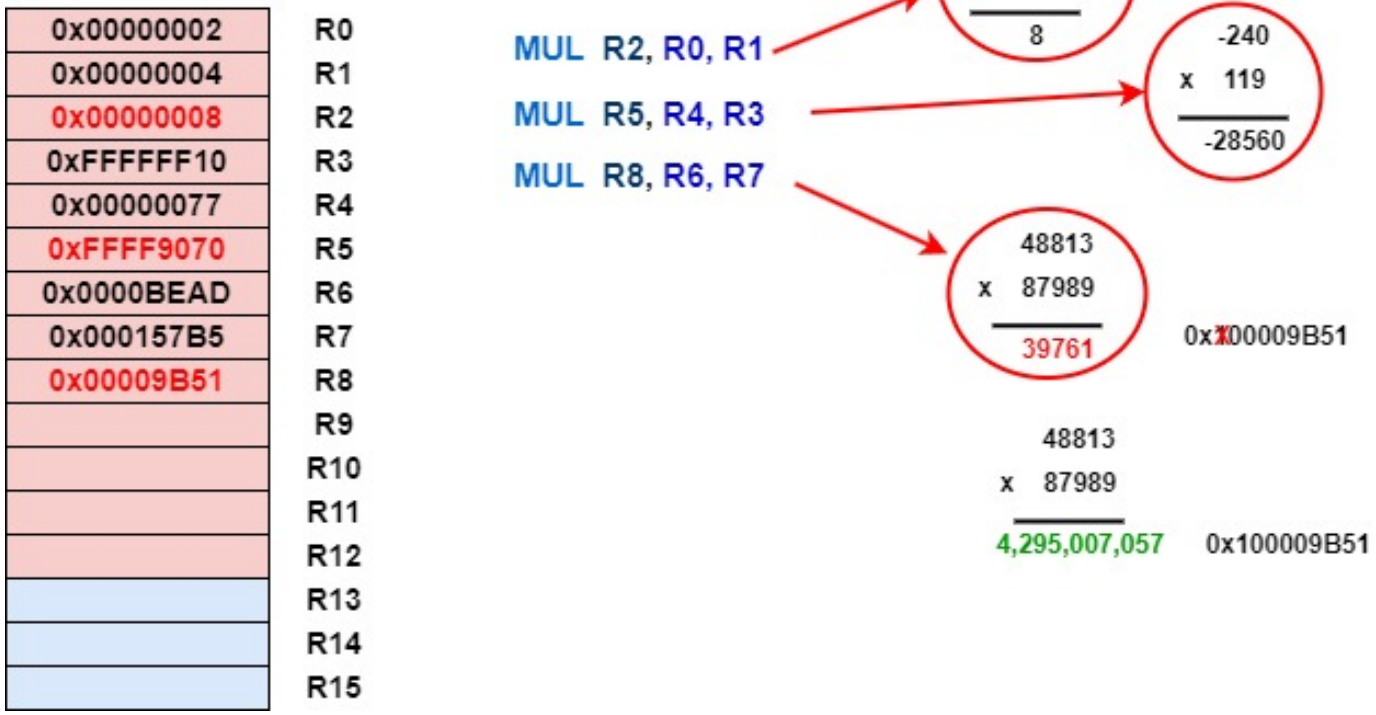
STUDENTS-HUB.com

Arithmetic Instructions- ADD



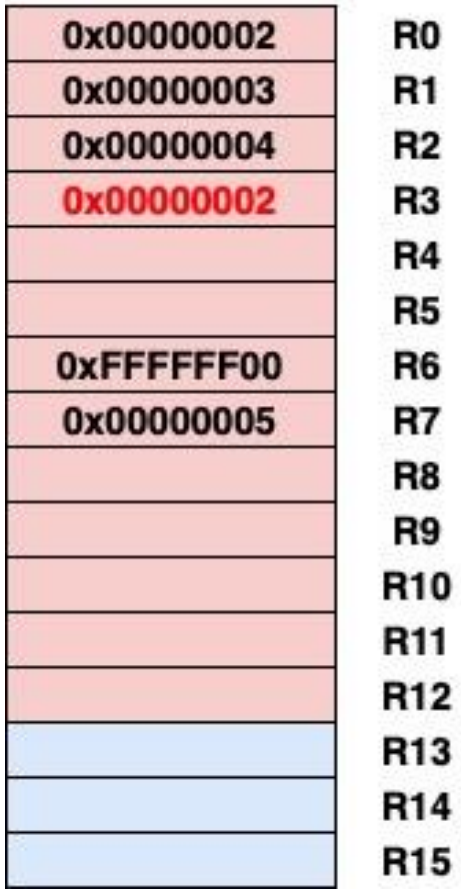
STUDENTS-HUB.com

Arithmetic Instructions- Multiplication



STUDENTS-HUB.com

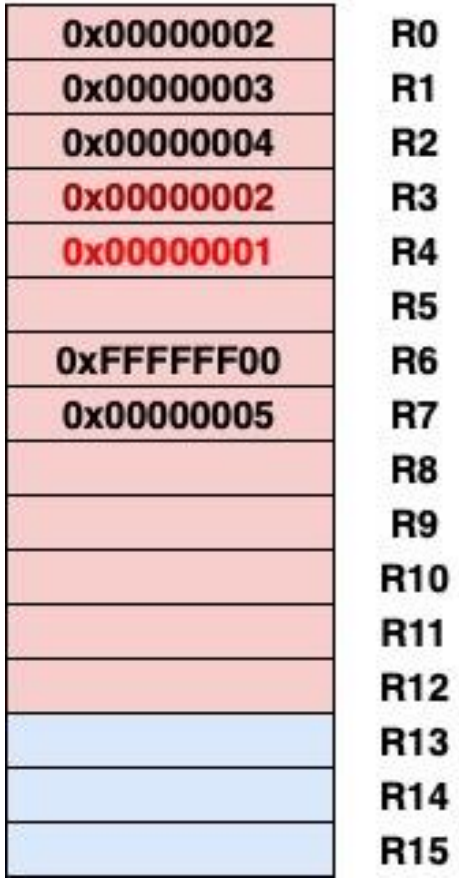
Arithmetic Instructions- Division



UDIV R3, R2, R0 4 / 2 = 2

STUDENTS-HUB.com

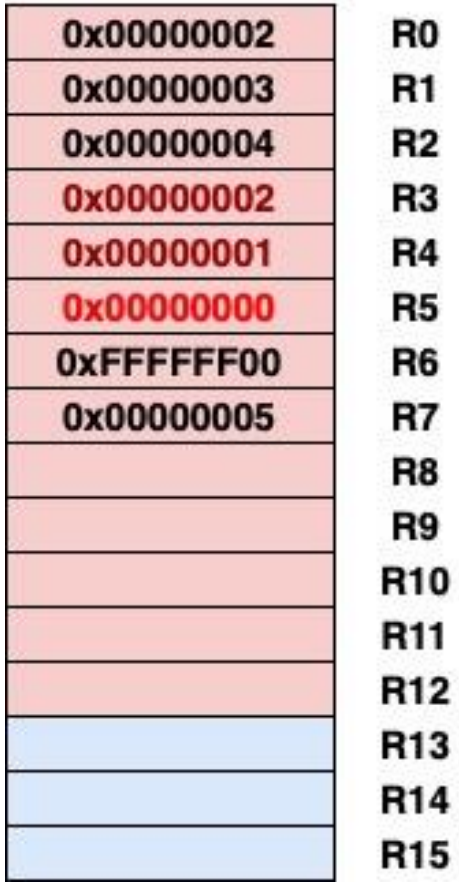
Arithmetic Instructions- Division




```
UDIV   R3, R2, R0      4 / 3 = 1

UDIV   R4, R2, R1
```

STUDENTS-HUB.com

Arithmetic Instructions- Division

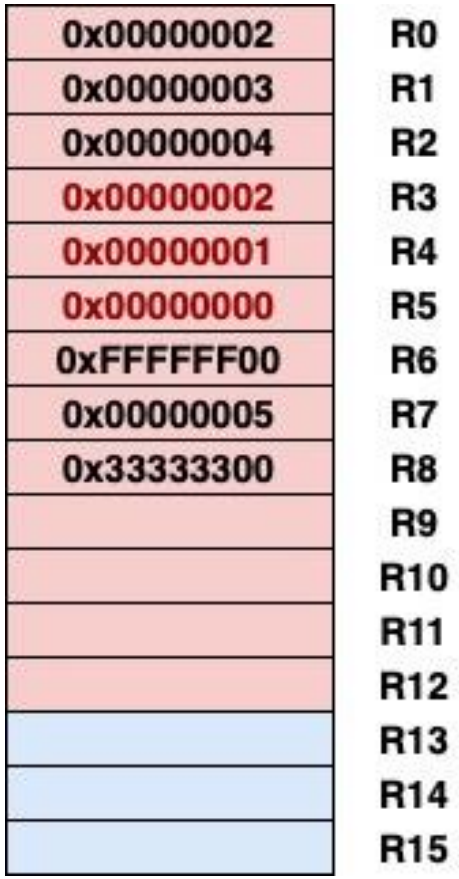



```
UDIV   R3, R2, R0
UDIV   R4, R2, R1
UDIV   R5, R1, R2
```

3 / 4 = 0

STUDENTS-HUB.com

Arithmetic Instructions- Division

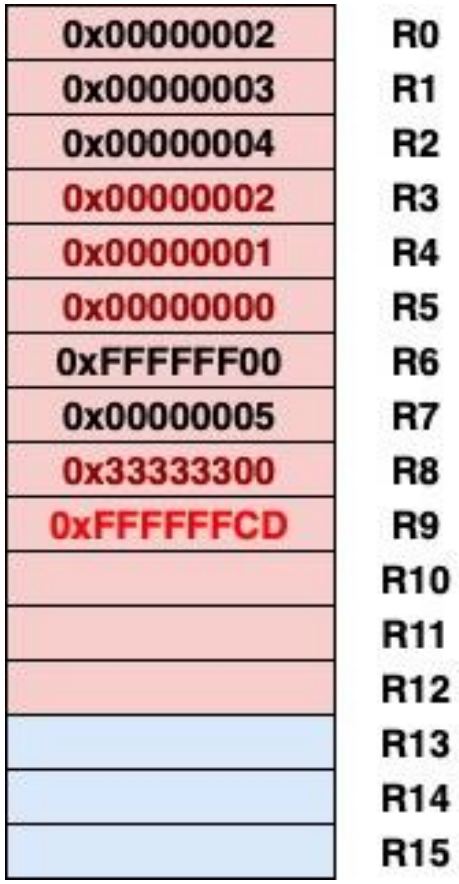



```
UDIV   R3, R2, R0
UDIV   R4, R2, R1
UDIV   R5, R1, R2
UDIV   R8, R6, R7
```

$$\frac{4294967040}{5} = 858993408$$

STUDENTS-HUB.com

Arithmetic Instructions- Division



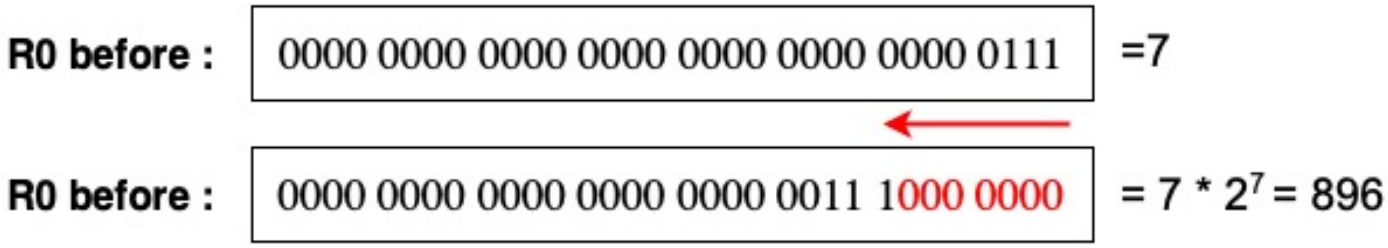

```
UDIV  R3, R2, R0
UDIV  R4, R2, R1
UDIV  R5, R1, R2
UDIV  R8, R6, R7
SDIV  R9, R6, R7
```

$$\frac{-256}{5} = -51$$

STUDENTS-HUB.com

Logical shift and rotation – Left Shift

- **Left Shifts** effectively multiply the contents of a register by $\mathbf{2^s}$ where **s** is the shift amount.
- **MOV R0,R0,LSL 7**

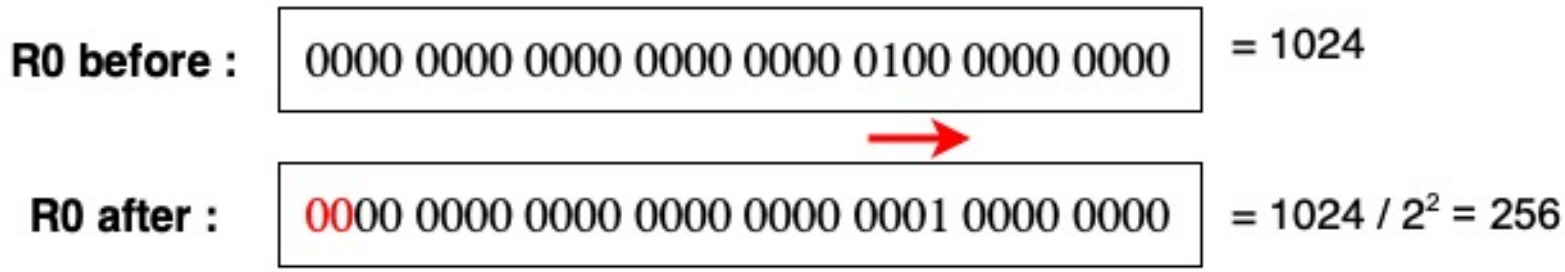


- Shifts can also be applied to the second operand of any data processing instruction ADD R1,R1,R0,LSL 7

STUDENTS-HUB.com

Logical shift and rotation – Right Shift

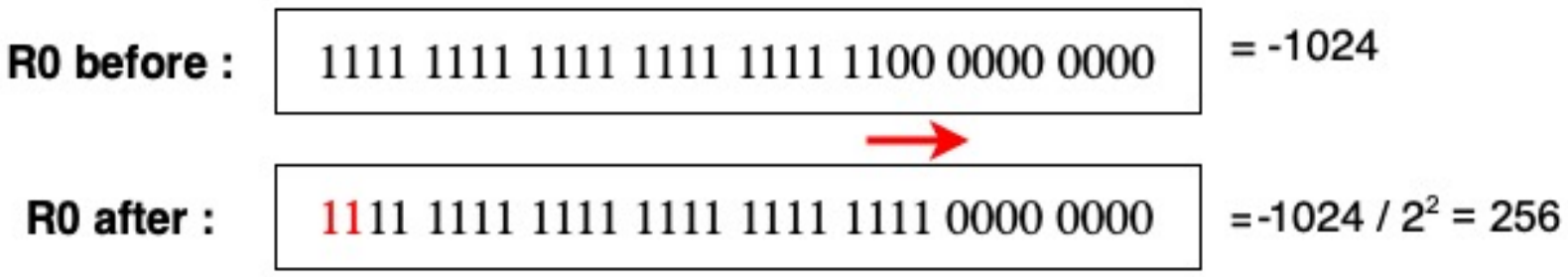
- **Right Shifts** behave like dividing the contents of a register by 2^s where **s** is the shift amount, if you assume the contents of the register are unsigned
- **MOV R0,R0,LSR 2**



STUDENTS-HUB.com

Logical shift and rotation – Arithmetic Right Shifts

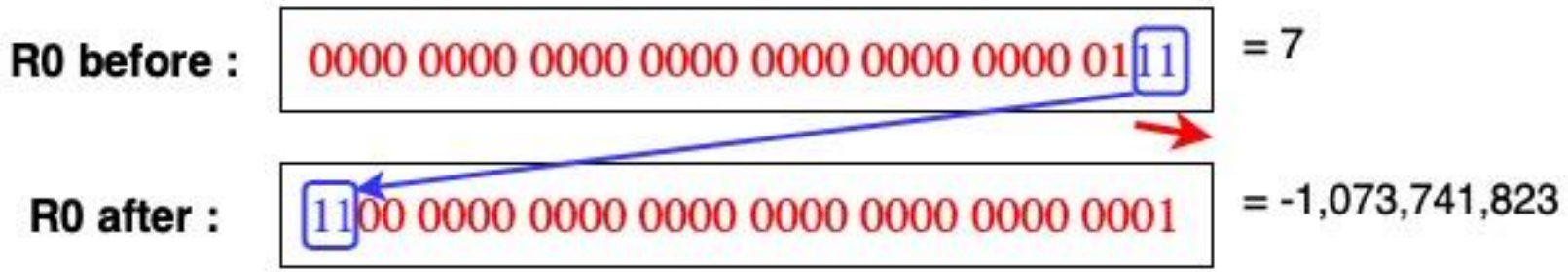
- **Arithmetic Right Shifts** behave like dividing the contents of a register by $\mathbf{2^s}$ where **s** is the shift amount, if you assume the contents of the register are signed
- **MOV R0,R0,ASR 2**



STUDENTS-HUB.com

Logical shift and rotation – Rotate Right Shift

- **Rotating Right Shifts** have no arithmetic analogy. However, they don't lose bits like both logical and arithmetic shifts
- **MOV R0,R0,ROR 2**



STUDENTS-HUB.com

Exercise 1- Setting a bit

- Set bit #5 in a register (assume R0) without affecting the other bits.
- a|= (1<<5);
- Use LSL to access the bit location

STUDENTS-HUB.com

Exercise 2- Clearing a bit

- Clear bit #5 in a register (assume R0) without affecting the other bits.
- A &= ~(1<<5);
- Use LSL to access the bit location

STUDENTS-HUB.com

Exercise 3- Toggling a bit

- Toggle bit #5 in a register (assume R0) without affecting the other bits.
- A^ = (1<<5);
- Use LSL to access the bit location

STUDENTS-HUB.com

Branches

- Branches (aka Jumps) allow us to jump to another code segment. This is useful when we need to skip (or repeat) blocks of codes or jump to a specific function.
- The best examples of such a use case are conditional statements (**if** statements) and Loops (**while** and **for**)

STUDENTS-HUB.com

Branches

```
main:
        mov     r1, #2      /* setting up initial variable a */
        mov     r2, #3      /* setting up initial variable b */
        cmp     r1, r2      /* comparing variables to determine which is bigger */
        blt     r1_lower    /* jump to r1_lower in case r2 is bigger (N==1) */
        mov     r0, r1      /* if branching/jumping did not occur, r1 is bigger (or the same) so store r1 into r0 */
        b       end         /* proceed to the end */
r1_lower:
        mov r0, r2          /* We ended up here because r1 was smaller than r2, so move r2 into r0 */
        b end               /* proceed to the end */
end:
        bx lr               /* THE END */
```

STUDENTS-HUB.com

Branches

```
int main() {
    int max = 0;
    int a = 2;
    int b = 3;
    if(a < b) {
     max = b;
    }
    else {
     max = a;
    }
    return max;
}
```

STUDENTS-HUB.com

Conditions in Assembly

Condition Field		
Mnemonic	**Description**	**Description (VFP)**
EQ	Equal	Equal
NE	Not equal	Not equal, or unordered
CS / HS	Carry Set / Unsigned higher or same	Greater than or equal, or unordered
CC / LO	Carry Clear / Unsigned lower	Less than
MI	Negative	Less than
PL	Positive or zero	Greater than or equal, or unordered
VS	Overflow	Unordered (at least one NaN operand)
VC	No overflow	Not unordered
HI	Unsigned higher	Greater than, or unordered
LS	Unsigned lower or same	Less than or equal
GE	Signed greater than or equal	Greater than or equal
LT	Signed less than	Less than, or unordered
GT	Signed greater than	Greater than
LE	Signed less than or equal	Less than or equal, or unordered
AL	Always (normally omitted)	Always (normally omitted)

All ARM instructions (except those with Note C or Note U) can have any one of these condition codes after the instruction mnemonic (that is, before the first space in the instruction as shown on this card). This condition is encoded in the instruction.

All Thumb-2 instructions (except those with Note U) can have any one of these condition codes after the instruction mnemonic. This condition is encoded in a preceding IT instruction (except in the case of conditional Branch instructions). Condition codes in instructions must match those in the preceding IT instruction.

On processors without Thumb-2, the only Thumb instruction that can have a condition code is `B <label>`.

STUDENTS-HUB.com

Branches

```
main:
        mov     r0, #0      /* setting up initial variable a */
loop:
        cmp     r0, #4      /* checking if a==4 */
        beq     end         /* proceeding to the end if a==4 */
        add     r0, r0, #1  /* increasing a by 1 if the jump to the end did not occur */
        b loop              /* repeating the loop */
end:
        bx lr               /* THE END */
```

STUDENTS-HUB.com

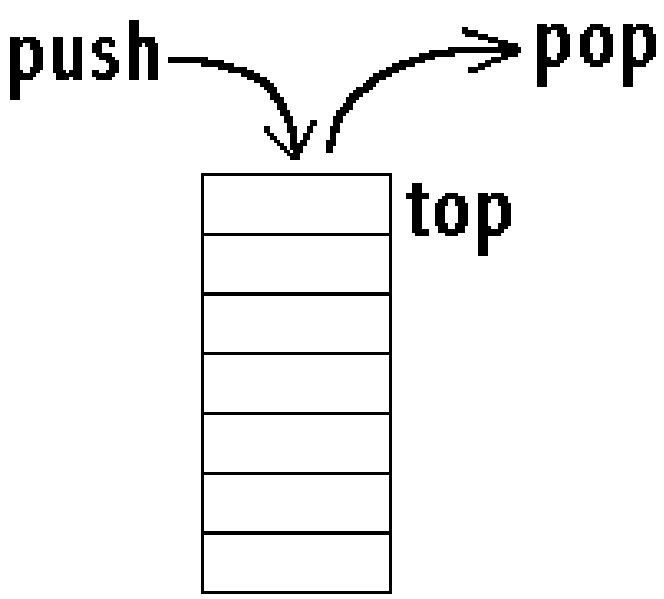
Branches

```
int main() {
    int a = 0;
    while(a < 4) {
    a= a+1;
    }
    return a;
}
```

STUDENTS-HUB.com

Stack Instructions- PUSH/ POP

- Stack Memory is part of the main memory reserved for the temporary storage of data (transient data), mainly used in function call, interrupt/exception handling.
- Stack Memory is accessed in **Last In First Out LIFO** manner.
- Addition and removal takes place **only** at one end, called the **top**.
- The stack can be accessed using PUSH, POP or memory instructions such as LDR, STR.



STUDENTS-HUB.com

Stack Instructions- PUSH/ POP

- The stack is traced by the Stack Pointer (SP), and is used to save below information:
  - Temporary storage for processor register values
  - Temporary storage for local variables of functions
  - Save the context of the current executing code before moving to exception/ interrupt handing routine
- The ARM Cortex-M processor uses a full descending stack. This means the stack pointer indicates the last stacked item on the stack memory. When the processor pushes a new item onto the stack, it decrements the stack pointer and then writes the item to the new memory location

STUDENTS-HUB.com

Stack Instructions- PUSH/ POP

Source: https://www.codeinsideout.com/blog/stm32/stack-memory/#stack

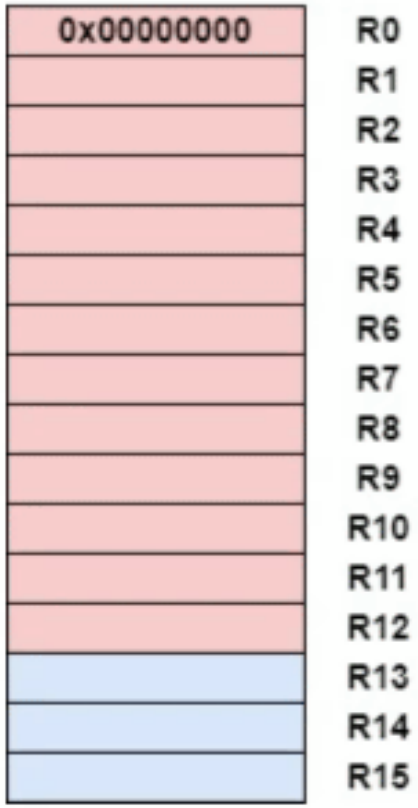
STUDENTS-HUB.com

Stack Instructions- PUSH/ POP

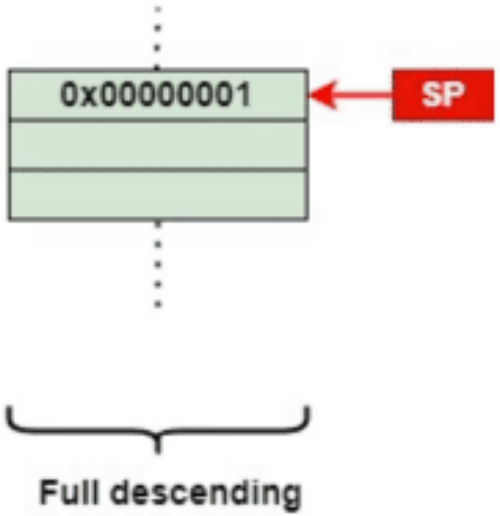
- The ARM Cortex-M processor uses a **full descending** stack.
- This means the stack pointer indicates the last stacked item on the stack memory.
- When the processor pushes a new item onto the stack, it decrements the stack pointer and then writes the item to the new memory location

STUDENTS-HUB.com

Stack Instructions- PUSH/ POP




```
MOV R0, #2
PUSH {R0}
MOV R0, #3
POP {R0}
```



STUDENTS-HUB.com

Questions?

STUDENTS-HUB.com