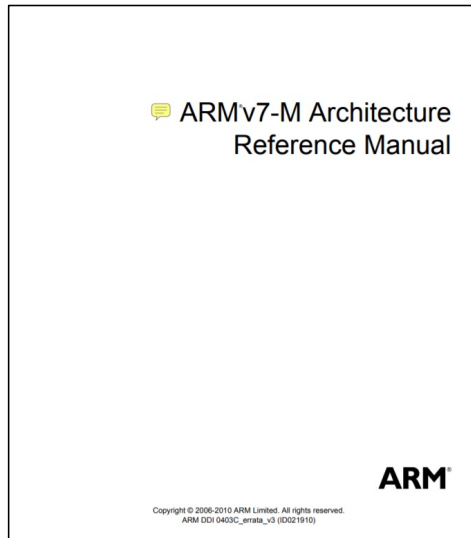


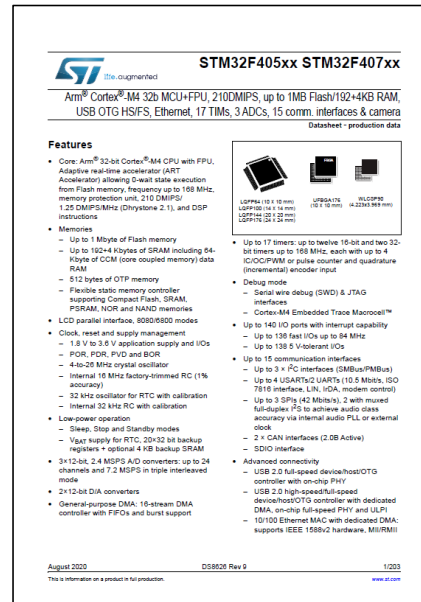
ARM architecture

STM32F407 μ C based on ARM Cortex-M4

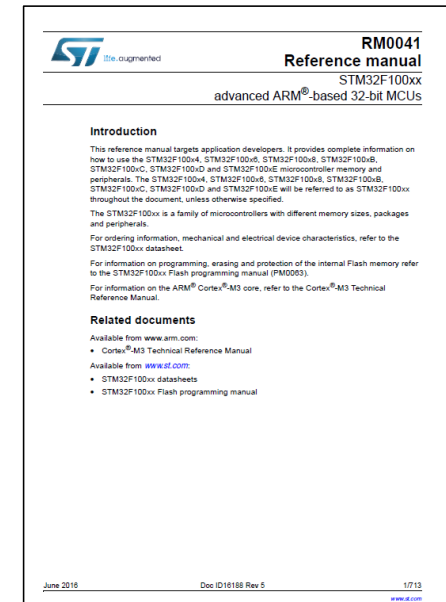
Bibliography



ARM®v7-M Architecture Reference Manual (free downloadable)



STM32F405xx - STM32F407xx DS8626 Rev 9 (free downloadable)



RM0041 Reference manual STM32F100xx advanced ARM®-based 32-bit MCUs (free downloadable)

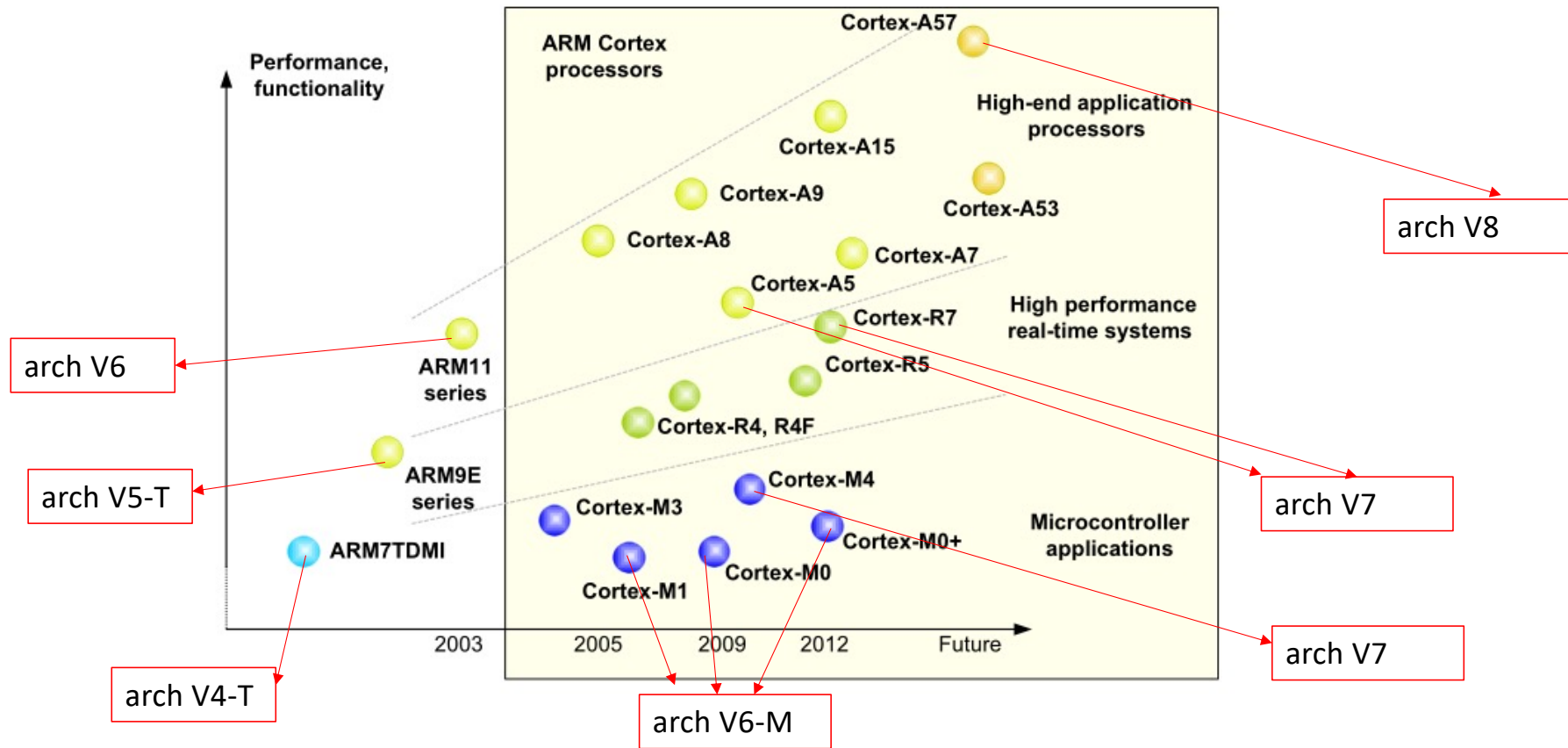
ARM architecture

- 1983: ARM project by Acorn Computer, Cambridge, UK.
- Create an Architecture and not a processor.
 - Client build their own chip.
 - Hard macro format: cell is provided.
 - Synthesizable format: IP is delivered.

IP: intellectual Property

- **Architecture: design or programmer's model:**
 - Defines Registers, addressing, memory architecture, operations.
 - Has several processors using the same basic features.
- Processor: Device.
 - Depends on an architecture,
 - Adds other not common features (pipeline).
 - Then, each processor has a slightly different configuration.

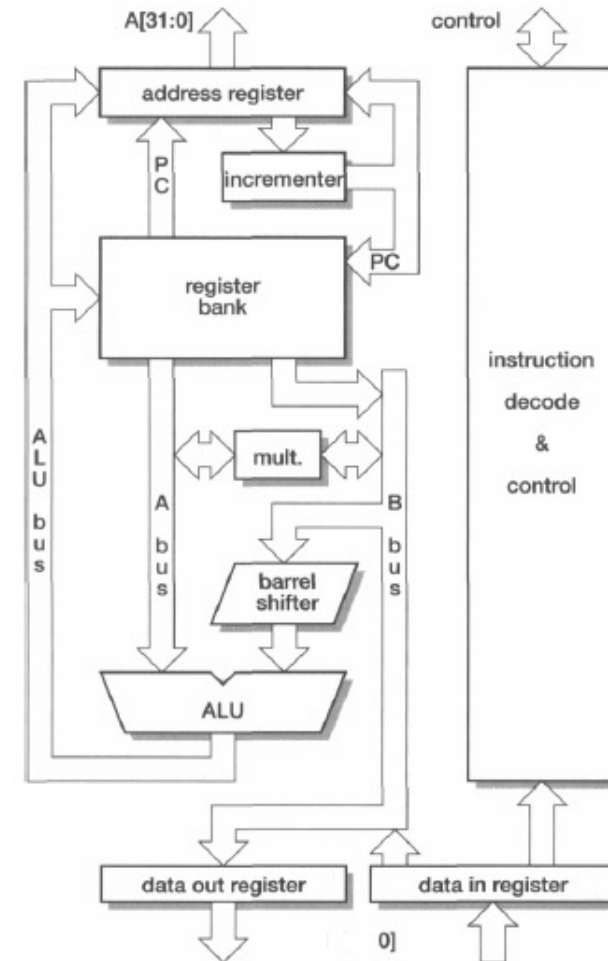
ARM architecture



ARM architecture

Every architecture is characterized by its datapath and control path.

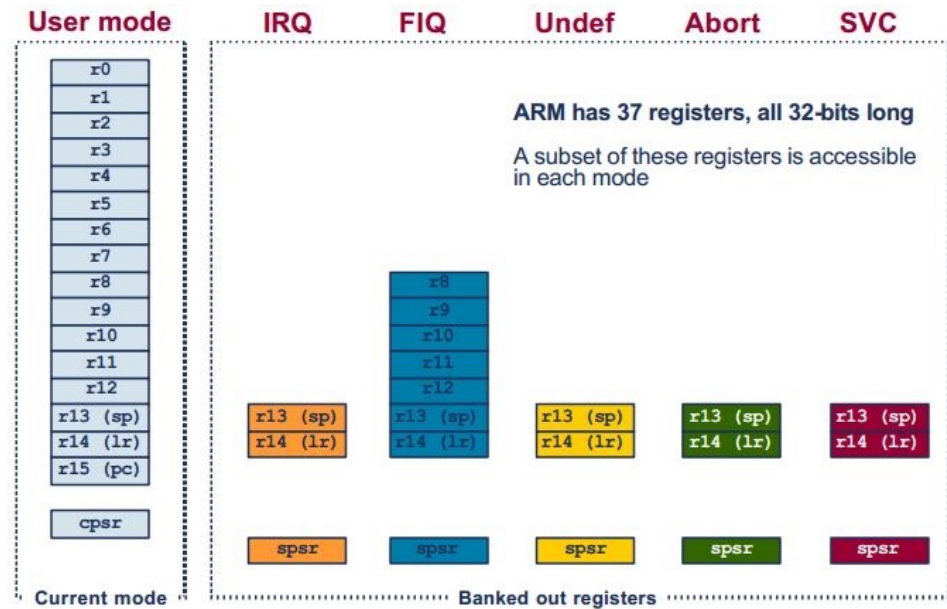
- **Bus:** instruction/data.
- **ID:** translates instructions before they are executed.
- **ALU:** use A and B buses to compute a result.
- **RF:**
 - Store operand and result,
 - Can be used to PC for instructions.
- **Incrementer:** inc/dec the register values independent of ALU.
- **Barrel shifter:** pre-process data before ALU.



ARM: registers

General Purpose Register:

- All registers are 32 bits
- Hold either data or address.
- r13, r14, r15 perform special functions.
- r0 to r3: are used to pass arguments to a function



ARM: registers

Special Purpose Registers:

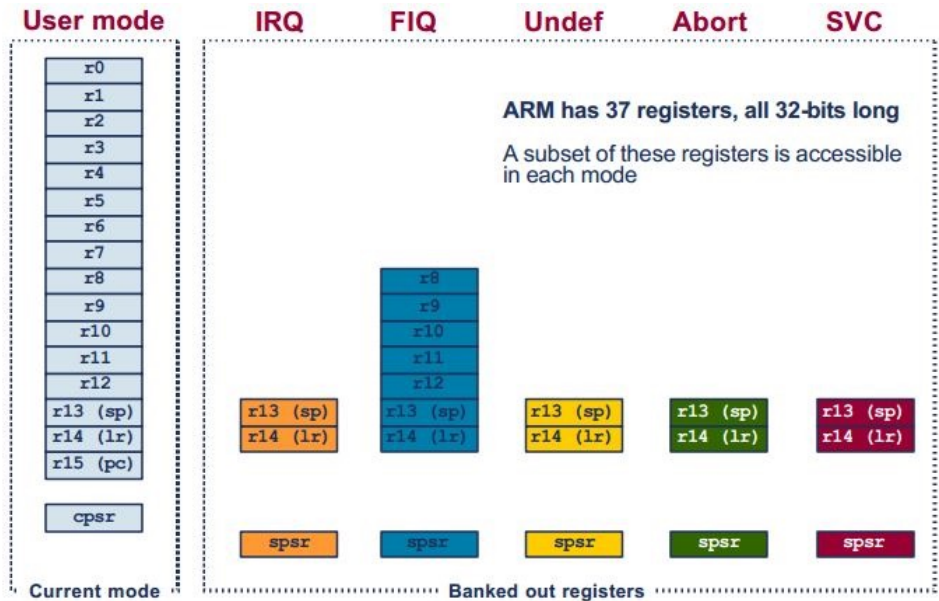
- **CPSR**: current program status register,
- **SPSR**: saved program status register.



- **Condition code flags**
 - N = **N**egative result from ALU
 - Z = **Z**ero result from ALU
 - C = ALU operation **C**arried out
 - V = ALU operation **oV**erflowed
- **Sticky Overflow flag - Q flag**
 - Architecture 5TE and later only
 - Indicates if saturation has occurred
- **J bit**
 - Architecture 5TEJ and later only
 - J = 1: Processor in Jazelle state
- **Interrupt Disable bits**
 - I = 1: Disables IRQ
 - F = 1: Disables FIQ
- **T Bit**
 - T = 0: Processor in ARM state
 - T = 1: Processor in Thumb state
 - Introduced in Architecture 4T
- **Mode bits**
 - Specify the processor mode
- **New bits in V6**
 - **GE[3:0]** used by some SIMD instructions
 - **E** bit controls load/store endianness
 - **A** bit disables imprecise data aborts
 - **IT [abcde]** IF THEN conditional execution of Thumb2 instruction groups

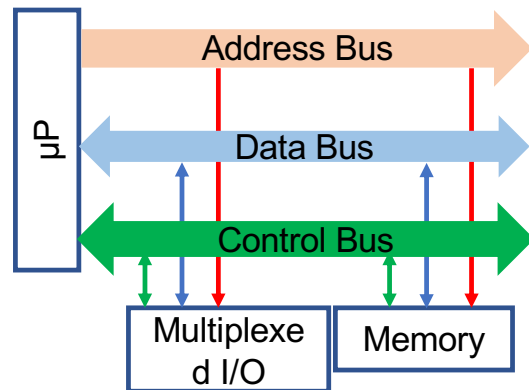
ARM: modes

- **Abort:**
 - Failed attempt to access memory,
- **IFQ/IRQ:**
 - Fast Interrupt Request, Interrupt Request,
- **Supervisor:**
 - After reset and OS kernel,
- **System:**
 - Allows full read-write access of CPSR,
- **Undefined:**
 - Undefined instruction.
- **User**
 - Common mode
 - 16 GPR and 2 SPR



- Banked registers
 - r0 to r7: never banked.
 - r8 to r12: banked FIQ mode.
 - r13, r14, and r15: unique to each mode.

μP: architecture



- m bits address bus:
 - 2^m addresses
- n bits data bus:
 - 8, 16, 32, 64 bits
- ARM μC:
 - A memory address N : a data byte
 - **4 bytes** data bus:
 - it can access in a single operation the 4 bytes located at the address $N, N+1, N+2, N+3$

- **endianess:**
 - $4735 = 0x127F$ located at $@0x1000$

- The ARM core supports either format by configuration.
- ARM-based μC are often constrained to a single format by the manufacturer

Address	Little-endian	Big-endian
0x1000	7F	12
0x1001	12	7F

ARM: memory

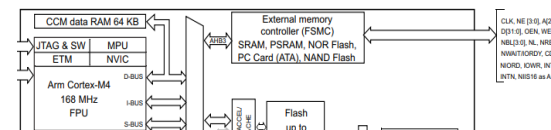
- **Static:**
 - Simple interface, short access time,
 - Low integration, high cost
- **Dynamic**
 - High integration, low-cost,
 - Complex interface, periodically refresh
- **Nature:**
 - RAM, ROM,
 - Volatile or non-volatile
 - NVSRAM: SRAM + Flash
- **Timing:**
 - Read/write access time
 - Cycle time (the minimum time between two accesses).
 - Refresh time (DRAM)
 - Chronology and compatibility of control signals, address, data

- **Parallel access:**
 - To address bus, data bus,
 - external control bus:
 - Read/Write, SC (Select Chip from address bus)
- **Serial access**
 - Non-volatile memory,
 - SPI, I2C
- Add memory?
 - Some μ C families allow or not,

Table 2. STM32F405xx and STM32F405VG

Peripherals	STM32F405RG	STM32F405OG	STM32F405VG
Flash memory in Kbytes	1024		
SRAM in Kbytes	System		
	Backup		
FSMC memory controller	No		

Figure 5. STM32F40xxx block diagram



ARM: instruction

- **Architecture CISC/RISC**

- Complex/Reduced Instruction Set Computer,
- Instruction set encoding/decoding,
- Execution speed,
- Internal wiring (inputs to ALU)
- Compiler complexity

- **RISC**

- Load/Store to register bank
- Reduced instruction set
- Indirect addressing to memory
- Fixed size instruction
- One clock cycle for one instruction (objective, except load and store)
- More complex compiler (code density)
- AVR8, ARM

- **CISC**

- Use of Accumulator register,
- Various addressing modes,
- Variable size instruction,
- Complex decoder and sequencer,
- Many clock cycle for one instruction
- 68000, 8086, STM8, 8051

ARM: instruction

- Large uniform register file,
- Load-store architecture. Data processing operates on register contents only and not involves memory location,
- Uniform and fixed length instructions,
- 32-bit processor and Instructions are 32-bit long,
- Good speed/power consumption ratio,
- High code density.

Add-on features of ARM:

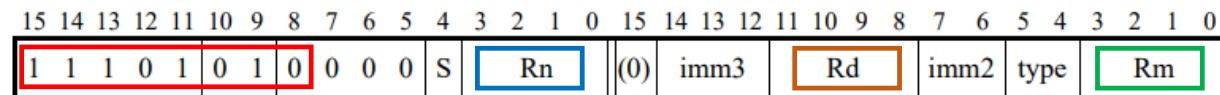
- Control over ALU and shifter for every data processing operations to maximize their usage,
- Auto-increment and auto-decrement addressing modes to optimize program loops (not very common with RISC processors),
- Load and Store multiple instructions to maximize data throughput,
- Conditional execution of instruction to maximize execution throughput (Branch instruction can be used in conjunction with other operations).

ARM: instruction

- At minimum :
 - **Opcode**: instruction format and operation
- Can includes:
 - One or many **operands** (registers, external memory, values)
- Variable/Fixed size instruction

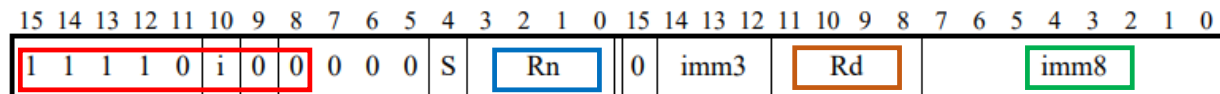
- Intel 8051 (**CISC** architecture)
 - ANL A, Rn (**1 byte**: 0101 1 nnn)
 - ANL A, #const (**2 byte**: 0101 0100 const)
 - ANL adr, #const (3 byte: 01010011 adr const)
- ARM Cortex M3 (**RISC** architecture)
 - Fixed, 4 Byte, ARMv7-M
 - AND R1, R1, R0 (0xEA 01 01 00), **encoding T2**
 - AND R1, R1, #0AA (0xF0 01 01 AA) , **encoding T1**

Encoding T2 ARMv7-M
 AND{S}<C>.W <Rd>, <Rn>, <Rm>{, <shift>}



T2

Encoding T1 ARMv7-M
 AND{S}<C> <Rd>, <Rn>, #<const>



T1

ARM: instruction - Thumb

- **Complex functions:**
 - Performed in a **single** instruction in a CISC,
 - May require **multiple** instructions in a RISC.
- Objective **Thumb**
 - In embedded system: the **cost of memory** is more critical than the **execution speed**.
 - Reduce the memory costs of these extra instructions,
 - ARM7TDMI

- Thumb instruction
 - Consists of 16-bit instructions,
 - A subset of the 32-bit ARM instructions.

Encodes a subset of the 32 bit instruction set into a 16-bit subspace.

Code density

```
-- ARM divide, 5*4 = 20 bytes
    MOV r3, #0
Loop
    SUBS r0, r0, r1
    ADOGE r3, r3, #1
    BGE loop
    ADO r2, r0, r1

-- Thumb divide, 6*2 = 12 bytes
    MOV r3, #0
Loop
    ADD r3,#1
    SUB r0, r1
    BGE loop
    SUB r1, #1
    ADD r2, r0, r1
```

ARM: addressing mode

- **Immediate** addressing
 - Does not allow access to the data memory,
 - The instruction contains the data,
 - 8 bits, 16 bits
- **8051** (CISC)
 - MOV A, #0x78
- ARM Cortex M3 (RISC)
 - load 32-bit register with 8-bit value
 - MOV R0, #0x78
 - load 32-bit register with 16-bit value
 - MOV R1, #0xF078
 - load 32-bit register with 32-bit value
 - MOV.W R2, #0x20000000
 - **Generic** instruction to load 32-bit immediate values
 - LDR R2, =0x20000001

- **Register** addressing
 - Designates source and destination register
- **8086** (CISC)
 - MOV A, R0
- ARM Cortex M3 (RISC)
 - load R0 with the content of R1
 - MOV R0, R1
 - load R0 with content of $R1 * 256$
 - MOV R0, R1 LSL #8 (Left Shift Logic)
 - load R0 with content of $R1 * 2^{R3}$
 - MOV R0, R1 LSL R3
- Arithmetic and Logic operations
 - Use only register addressing

#0x20000000: encoded on 16 bits

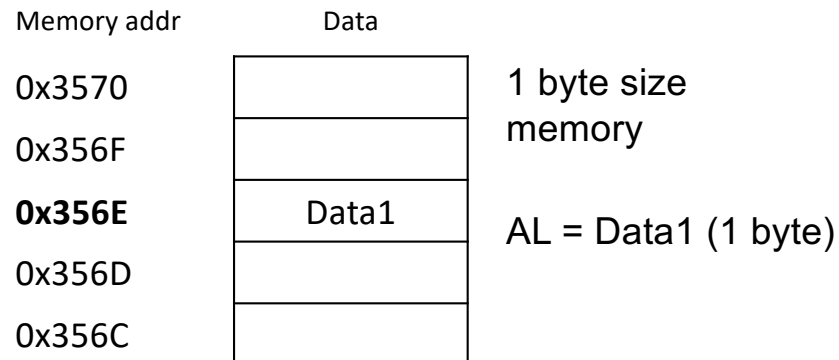
#0x20000001: can't be encoded on 16 bits

MOV R1, #0x20000000 -> instruction is coded on 32 bit -> there is no space to code R1 and opCode

ARM: addressing mode

- **Direct** addressing
 - An operand contains the address of the data.
- **8086** (CISC)
 - MOV AL, **[0x356E]** (0x356E is an address)
 - Load register AL (8bits) with data in memory at address 0x356E
- **ARM** Cortex M3 (RISC)
 - **Does not exist**

- **Indirect** addressing
 - An operand refers to a container having the address of the data,
 - A pointer,
- **8051** (CISC)
 - The pointer is a register,
 - MOV A, @DPTR



A = @DPTR

A = 0x356E

ARM: addressing mode

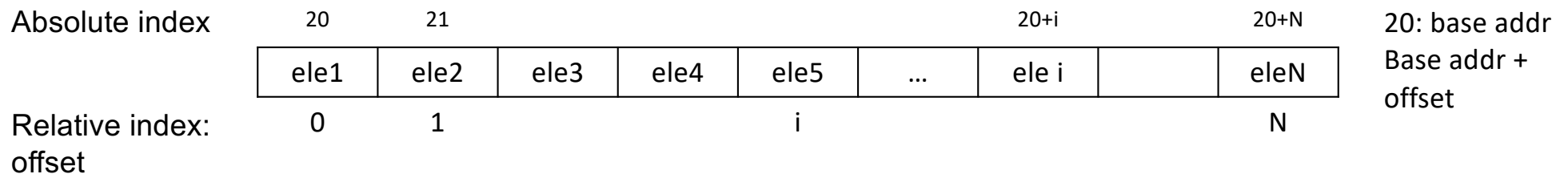
- Indirect addressing, ARM Cortex M3
- **Without offset**
 - LDR R8, [R0] : pointer = R0
- **With offset**
 - LDR R8, [R0, offset] : pointer = R0 + offset
 - LDR R8, [R0, R1] : pointer = R0 + R1
 - LDR R8, [R0,R1, LSL #N] : pointer = R0 + R1*2^N

R0=20

LDR R8, [R0]: (no offset)
R8 = ele1, ele2, ele3, ele4

LDR R8, [R0+2] -> R8=ele3..ele6 (with offset)

LDR R8, [R0+4] -> R8=ele5..ele8
LDR R8, [R0, R1 LSL #2] R1= i (loop iteration)



ARM: addressing mode

- Indirect addressing, ARM Cortex M3
- **Pre-indexed**
 - LDR R8, [R0, offset]!
 - Pointer = R0 + offset and next R0 = R0 + offset
- **Post-indexed**
 - LDR R8, [R0] , offset
 - Pointer = R0 and next R0 = R0 + offset

```
LDR R8, [R0] , #4  
Pointer = 0x2000 0000  
R8 = 0x1221AF36  
R0 = 0x2000 0004
```

```
LDR R8, [R0, #4]!  
Pointer = 0x2000 0004  
R8 = 0x220D32AA  
R0 = 0x2000 0008
```

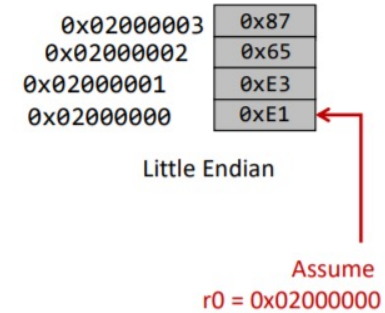
R0 = 0x2000 0000

0x2000 0000	0x36
0x2000 0001	0xAF
0x2000 0002	0x21
0x2000 0003	0x12
0x2000 0004	0xAA
0x2000 0005	0x32
0x2000 0006	0x0D
0x2000 0007	0x22

ARM: addressing mode

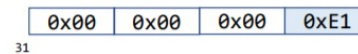
LDR	Load Word
LDRB	Load Byte
LDRH	Load Halfword
LDRSB	Load Signed Byte
LDRSH	Load Signed Halfword

STR	Store Word
STRB	Store Lower Byte
STRH	Store Lower Halfword



Load a Byte

LDRB r1, [r0]



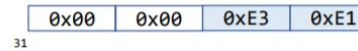
Load a Signed Byte

LDRSB r1, [r0]



Load a Halfword

LDRH r1, [r0]



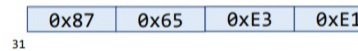
Load a Signed Halfword

LDRSH r1, [r0]



Load a Word

LDR r1, [r0]



Sign extension

LDRSB



0x06: 0000 0110

S

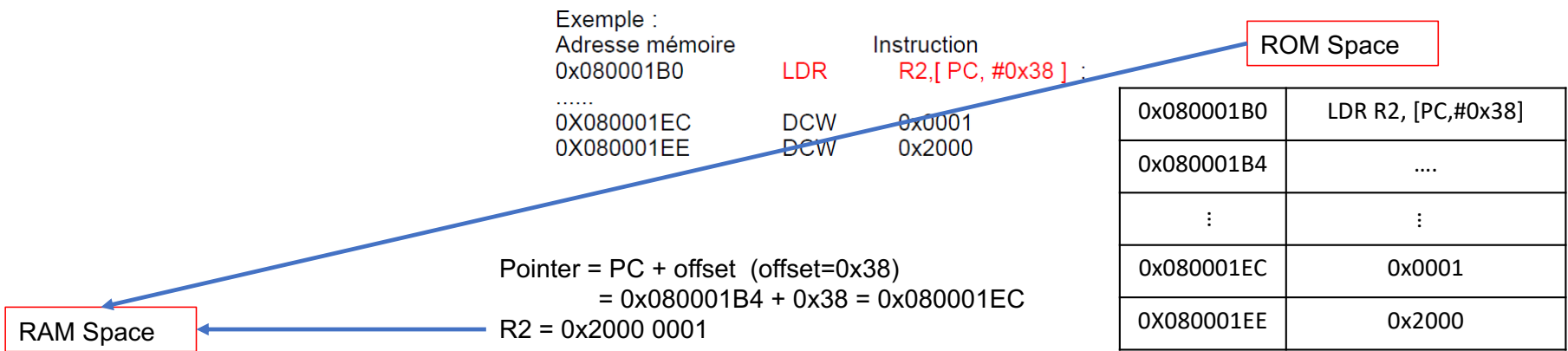
0xE1: 1110 0000

S

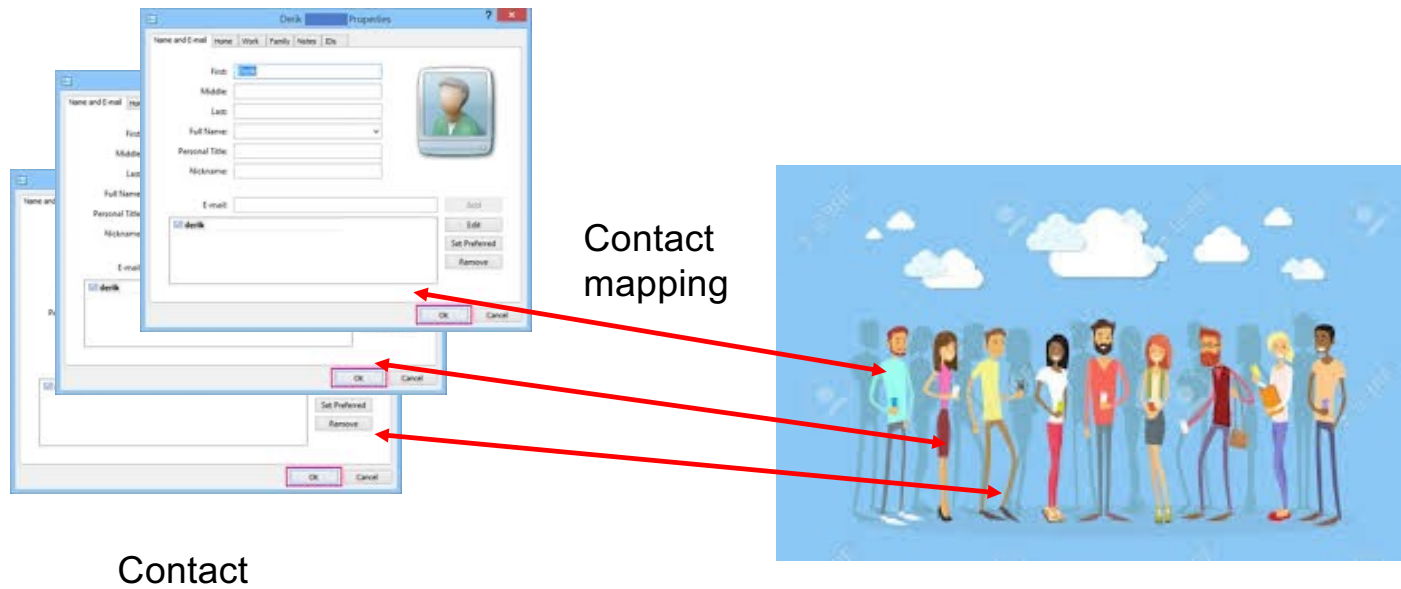
ARM: addressing mode

- LDR R2, =0x20 (**pseudo-op**)
 - MOV r2, #0x20 (**Asm-op**)
- LDR R2, 0x20000000 (pseudo-op)
 - MOV R2, #0x20000000 (Asm-op)
 - We say that the 32 bits constant #0x20000000 can be encoded on 8 bits (0x20) by Shifting left 24 bits.
- LDR R2, 0x20000001 (pseudo-op)
 - Can't be encoded
 - Too large for MOV

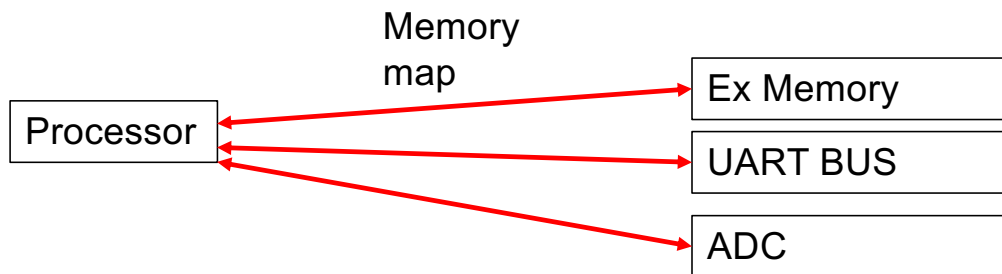
- Indirect addressing, ARM Cortex M3
- **PC-relative** addressing
 - 32-bit constant is considered as **literal set** (label)
 - **Literal set**: constants stored after program in code area
 - LDR R2, 0x2000 0001
 - Takes two instructions to access data in memory:
 1. LDR R3, [PC, #offset]
 2. Literal pool: DCW 0x0800 0001
 3. LDR R2, [R3]



ARM: memory map

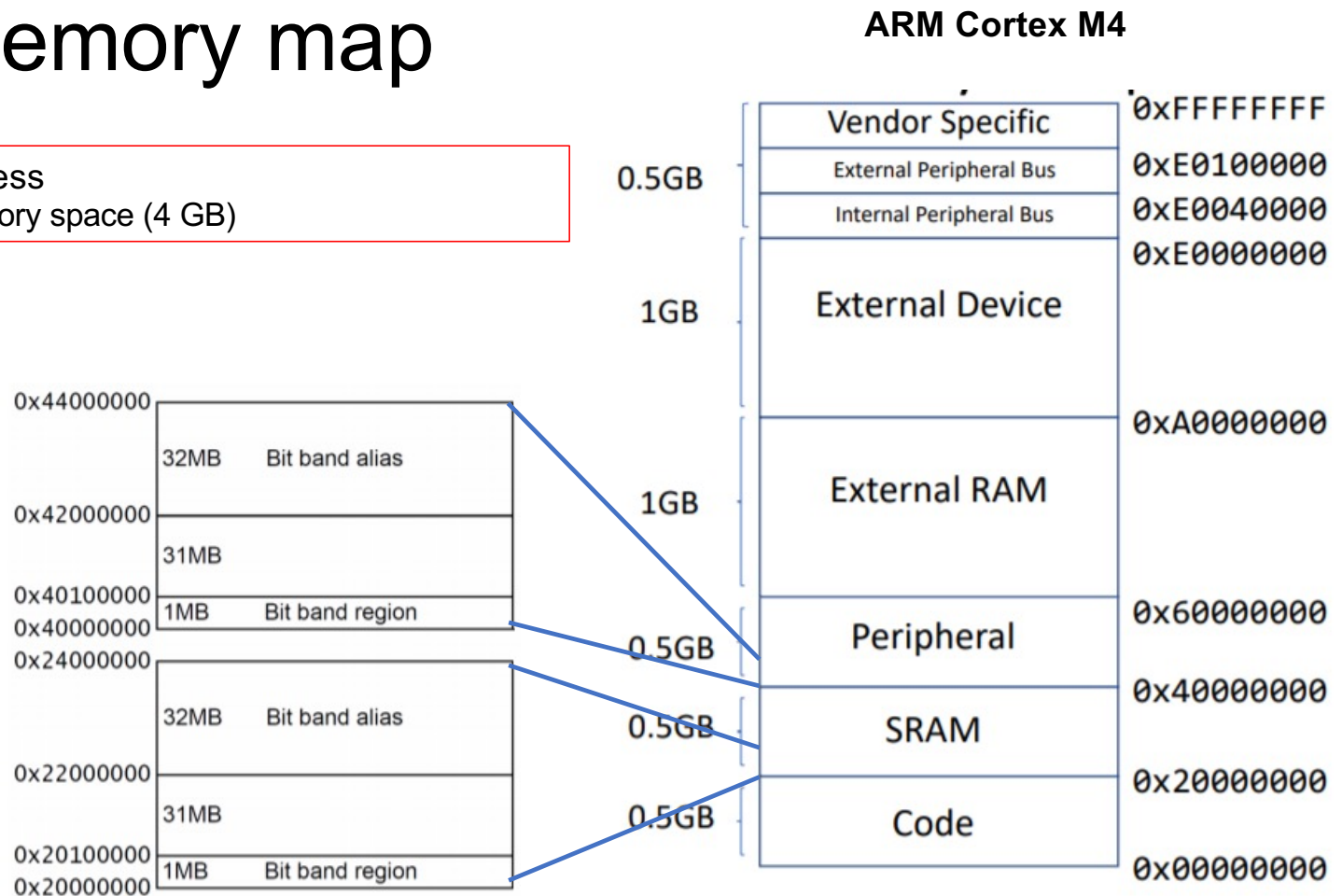


Contact



ARM: memory map

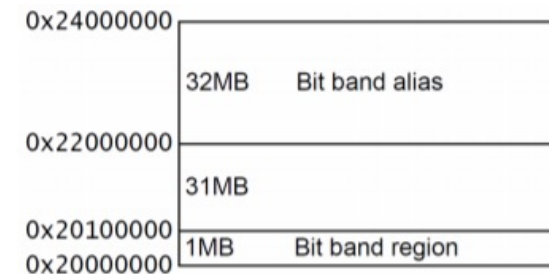
- 32-bit memory address
 - 2^{32} bytes of memory space (4 GB)



ARM: memory map – bit banding

• Context

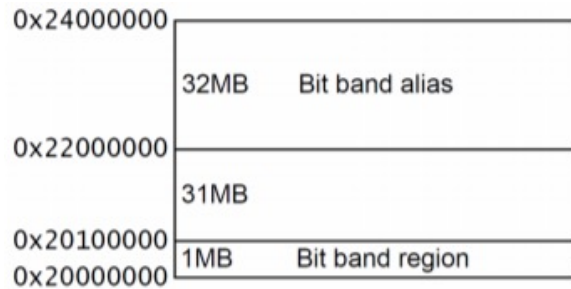
- CPU core cannot write to individual bits of a register. Instead it must write entire bytes or even words at a time.
- If a CPU needs to change the value of a bit
 1. Read the current value into a temporary register,
 2. Modify that value with a logic operation,
 3. Write the final result.
- Three step process named Read-Modify-Write:
 - Works fine when doing one thing at a time,
 - **Problems** when doing concurrent tasks
 - **Example:** what happens if an interrupt occurs between the read and modify operations that changes the value in the register? The new value will get overwritten.



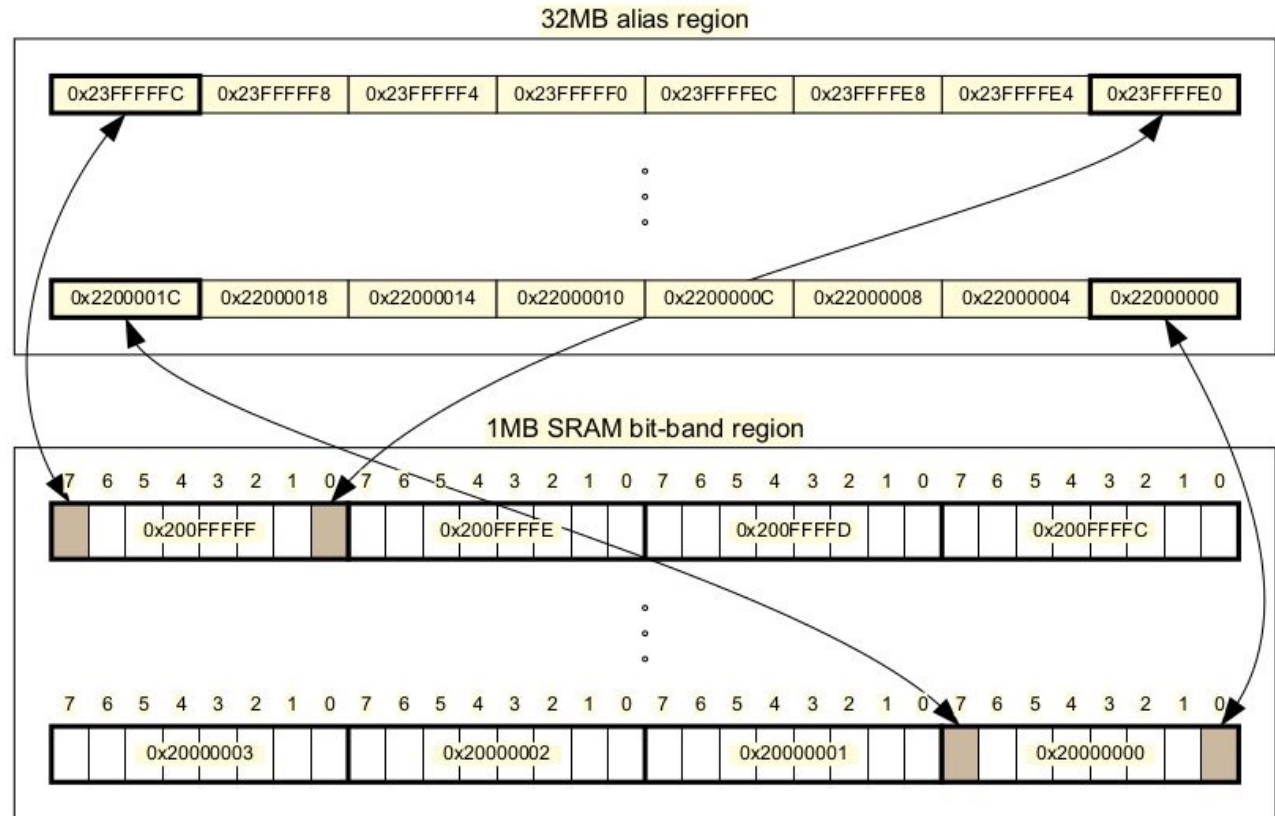
• Bit banding

- Maps each bit in the Bit-band region to an entire word in the Bit-band Alias Region,
- ARM Cortex M3/M4
- Write/Read a word in the alias region performs a write/Read the corresponding bit in the Bit-band region.
- Single machine instruction.

ARM: memory map – bit banding



- Write/Read a **byte** at the address 0x2000 0000
- Write/Read **bit 1** of 0x2000 0000 by writing/reading at the address 0x2200 0004

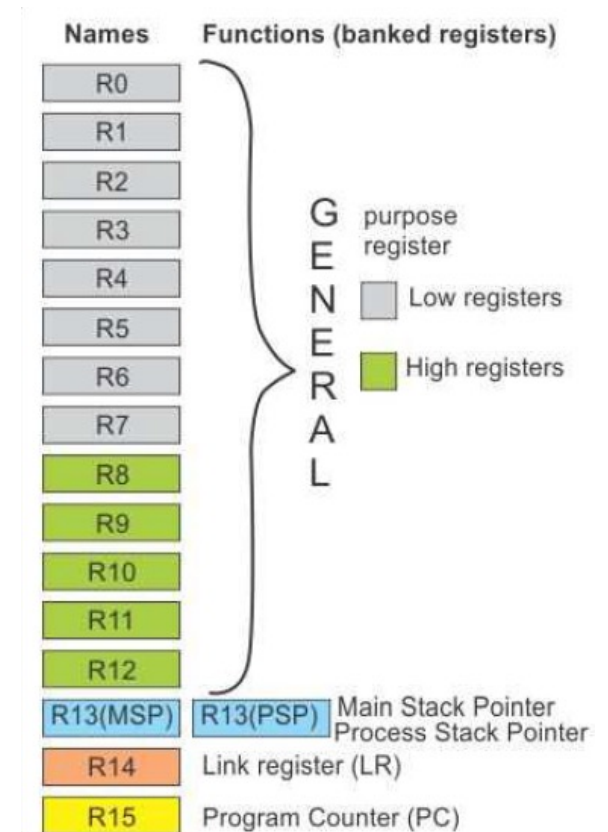


ARM: branch

- **Sequential** execution
 - Thanks to the implicit increment of the PC register
- **Conditional** execution

Instructions are placed consecutively in memory


Adresse	Instruction
0000 1000h	InstrA
0000 1004h	InstrB
0000 1008h	InstrC
0000 100Ch	
0000 1010h	InstrD
0000 1014h	InstrE
0000 1018h	InstrF
0000 101Ch	InstrG
0000 1020h	InstrH



ARM: branch


Unconditional Branch

Adresse	Instruction
0000 1000h	InstrA
0000 1004h	InstrB
0000 1008h	InstrC
0000 100Ch	Instruction SAUT à B 0000 101Ch
0000 1010h	InstrD
0000 1014h	InstrE
0000 1018h	InstrF
0000 101Ch	InstrG
0000 1020h	InstrH



Conditional Branch

Adresse	Instruction
0000 0FF8h	Instruction Test Condition
0000 0FFCh	Instruction Saut si Test Faux à 0000 1010h
0000 1000h	InstrA
0000 1004h	InstrB
0000 1008h	InstrC
0000 100Ch	Instruction SAUT à 0000 101Ch
0000 1010h	InstrD
0000 1014h	InstrE
0000 1018h	InstrF
0000 101Ch	InstrG
0000 1020h	InstrH



ARM: branch

Unconditional Branch

- **Implicit** modification of PC (Programm Counter)
- **Absolute** branch
 - `JMP 0x3024` (8051)
 - `LDR PC,=0x10203456` (ARM Cortex)
- **Relative** branch
 - `B label` (ARM Cortex M3)
- **Indirect** branch
 - `JMP @A + DPTR` (8051: PC = Register DPTR + Acc Register)
 - `BX R0` (Cortex M3: PC = R0 register content)

Conditional Branch

- **Explicit** modification of PC when a condition is TRUE
- Use also CPSR flags: C, V, N, Z
- ARM architecture:
 - Use of branch **suffix**
 - `BEQ label` (Branch if Z = 1)
 - `BLE label` (Branch if Z=1 or N! = V)

B[suffix]

BEQ: Branch when Equal

BLE: Branch Less/equal

ARM: branch

Conditional Branch

- Code Operation: ADD, AND, MOV
- S: optional **suffix** to update CPSR flags
 - ADDS, ANDS, MOVS
- C: optional **suffix** for conditional branch
 - ADDSEQ, ANDSNE, MOVSL0

<CODE OPERATION> {S} <C> Opérande 1, Opérande 2, {Opérande 3}

Example:

R0 = R0 + 1

IF R0 = 0 THEN R1 = R1 + R2

Branch execution

```
ADDS R0, #1: signed addition
BNE  label
ADDS R1, R1, R2
```

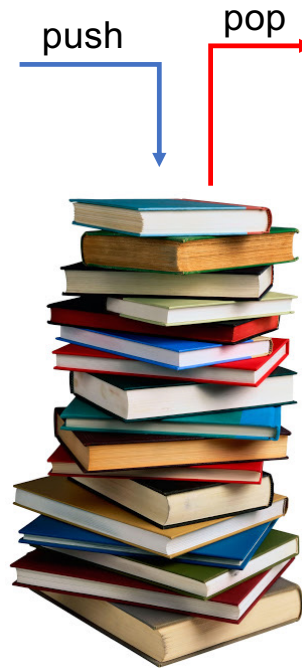
label

Conditional execution

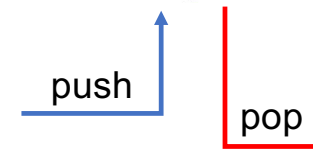
```
ADDS    R0, #1
ADDSEQ  R1, R1, R2
```

ARM: stack pointer

- Reserved memory area to store data,
- LIFO : Last In First Out
- SP: Stack Pointer
 - Designates the address of the last pushed data



Ascending stack



Descending stack

BL: Branch Link

To move between sub-programs

Stack: memory area

PC register Track instruction address in the program memory

SP register Track instruction address in the stack

PC register

Program memory
(ROM, Flash)

address1	instruction1
address2	instruction1

data memory
(RAM, SRAM)

address1	data1
address2	data2

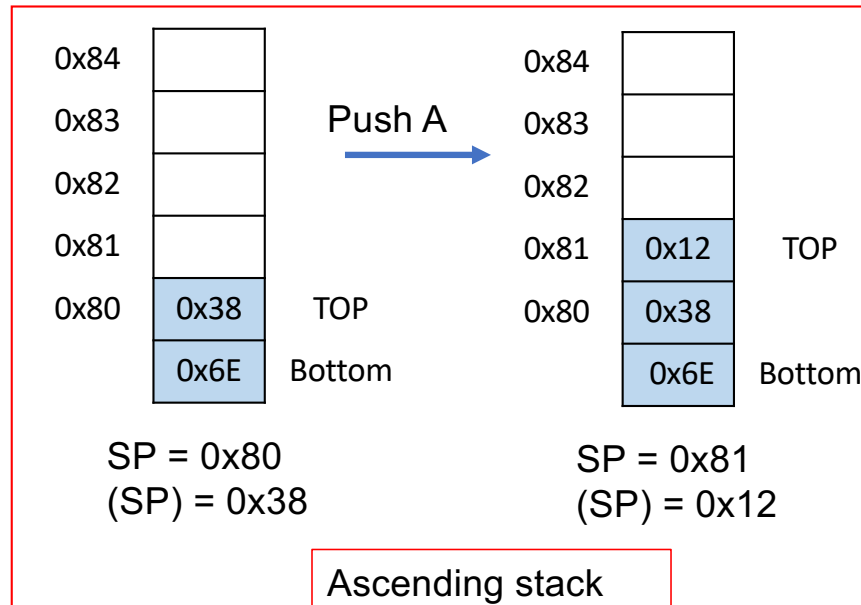
Stack

SP register

address1	Instruction address1
address2	data1
	Instruction address2
	Instruction address3
	data2

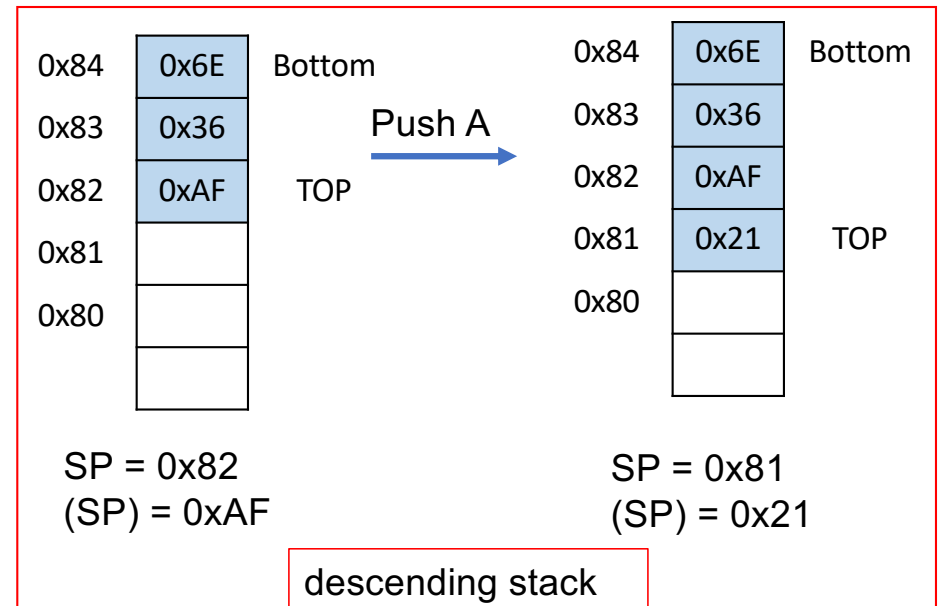
ARM: stack pointer

- **8051**: ascending stack
 - PUSH A
 - $A \rightarrow (++)SP$: pre-increment SP
 - POP A
 - $(SP--) \rightarrow A$: post-decrement of SP



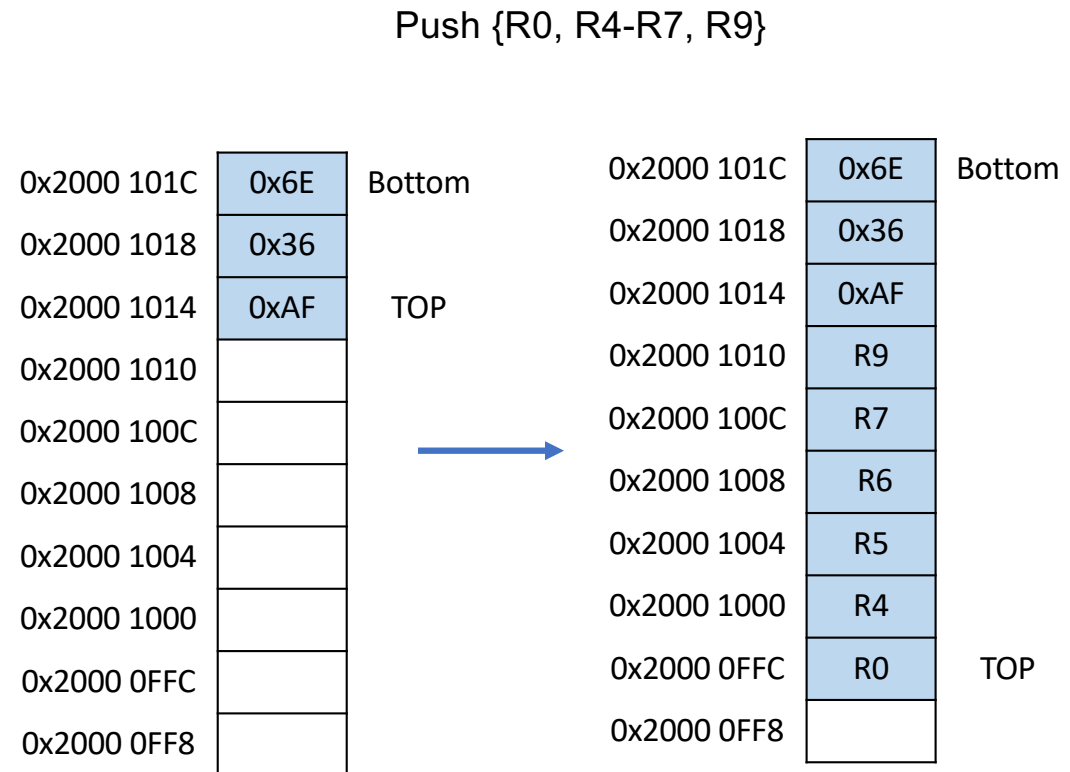
SP point to last pushed data (TOP of the stack)

- **STM8**: descending stack
 - PUSH A
 - $A \rightarrow (SP--)$: post-decrement SP
 - POP A
 - $(++)SP \rightarrow A$: pre-increment SP



ARM: stack pointer

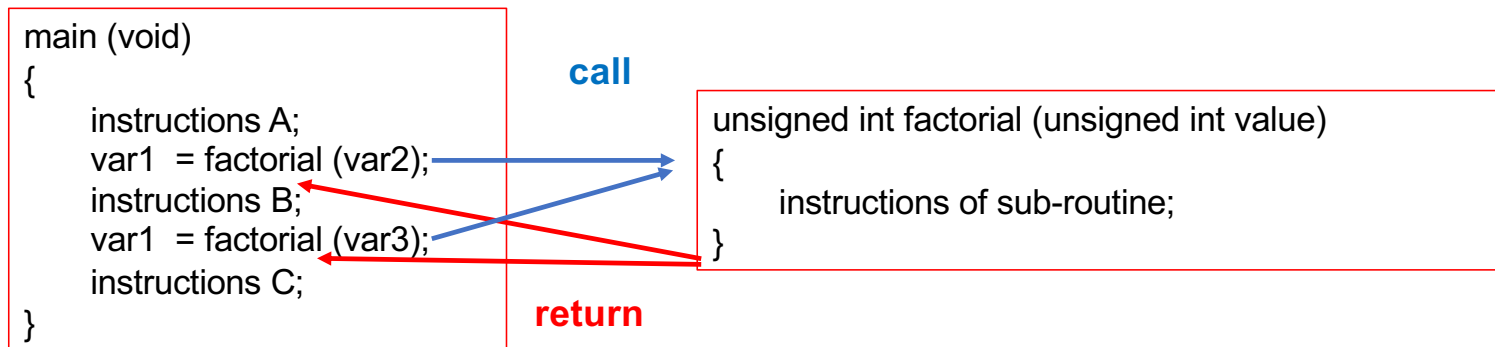
- Cortex M3: **descending** stack
 - SP: register R13
 - PUSH A
 - $A \rightarrow (SP--)$: post-decrement SP
 - POP A
 - $(++SP) \rightarrow A$: pre-increment SP



ARM: stack pointer

- **Sub-routines**
 - Functions/procedures
- Sub-routine **call**
 - Address branch from the principal program
 - How to return back to the principal program?

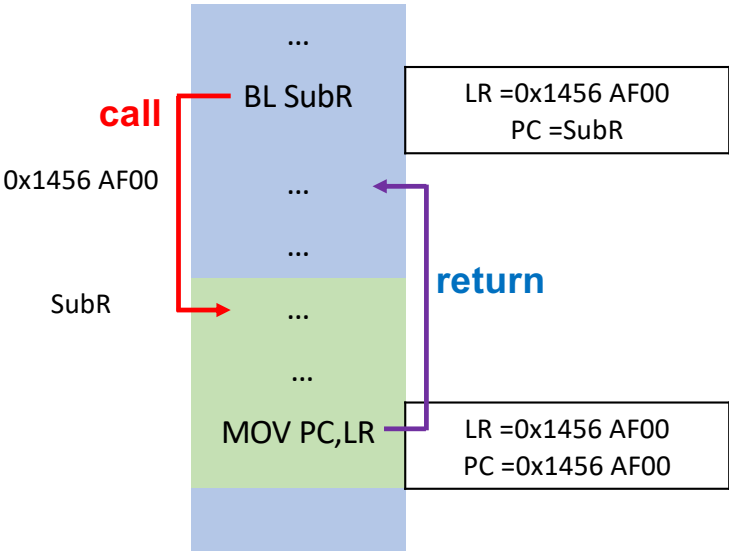
- Call instruction
 - Branch to an address by modifying the PC,
 - With prior backup of the PC.
- Return instruction
 - Recover the PC,
 - Return to the next address following the call instruction.
- **8051:**
 - CALL, RET



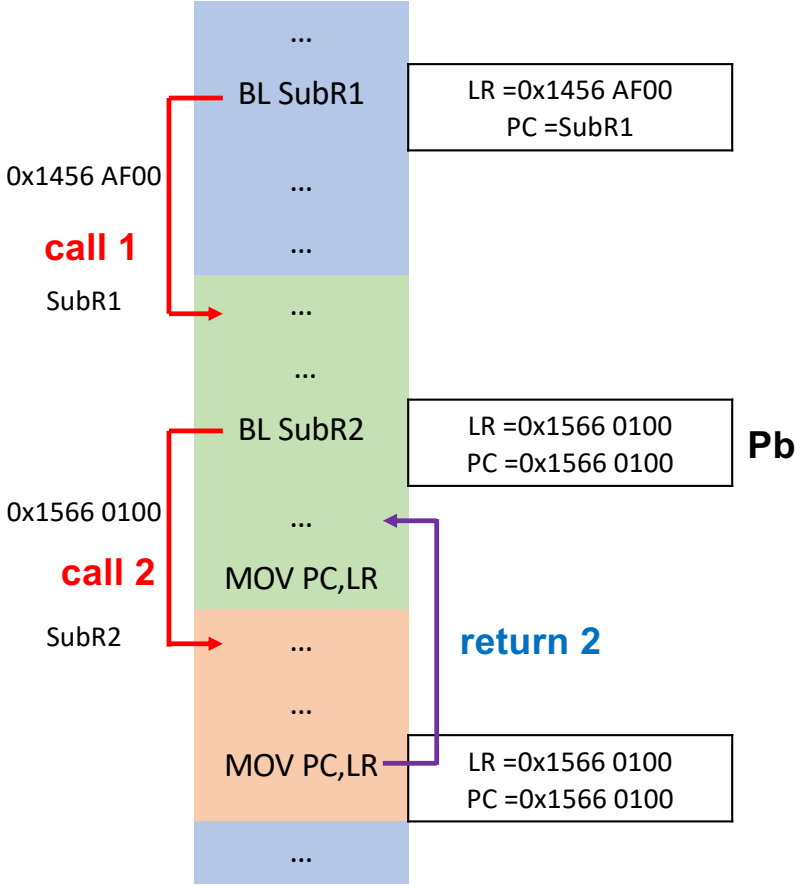
ARM: stack pointer

- ARM Cortex M3
 - BL <C>
- Branch and Link
 - Store PC in register R14 (LR)

Leaf Routine: no call in the sub-routine



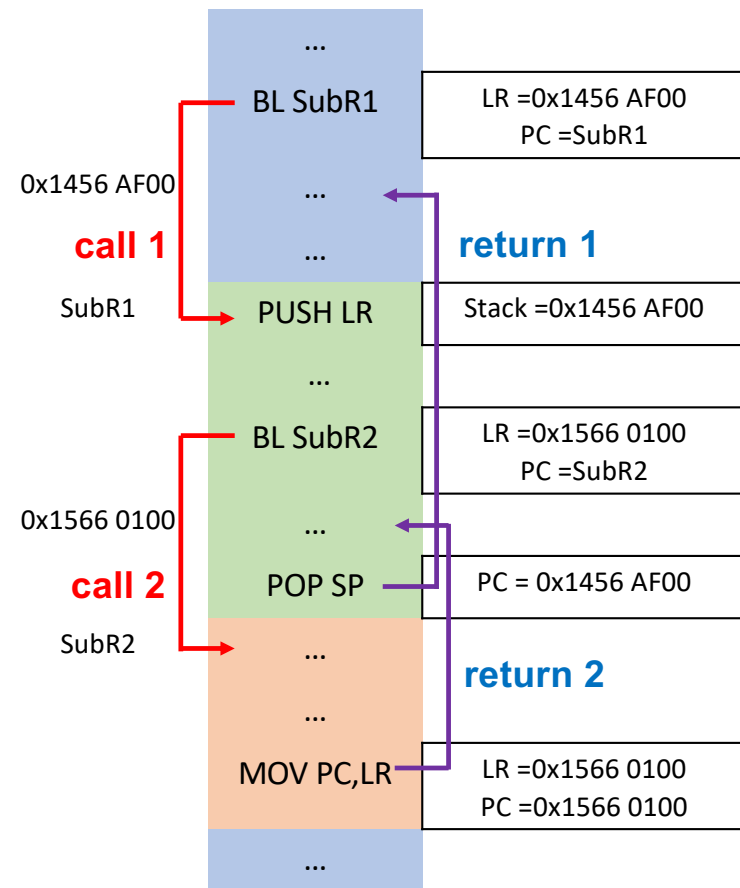
calls in the sub-routine: nested call



ARM: stack pointer

- ARM Cortex M3
 - BL <C>
 - Use stack:
 - PUSH to store LR register
 - POP to recover LR register

calls in the sub-routine



ARM: stack pointer

PC= @MI4

LR= @MI4

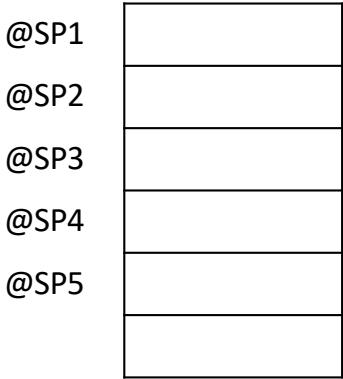
SP=@SP1

Main program

```
@MI1 instr1
@MI2 instr2
@MI3 instr3
BL subP1
@MI4 instr4
@MI5 instr5
```

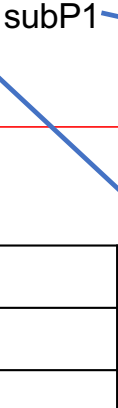
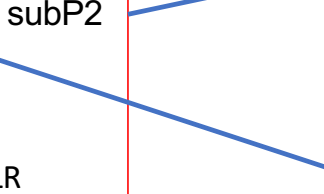
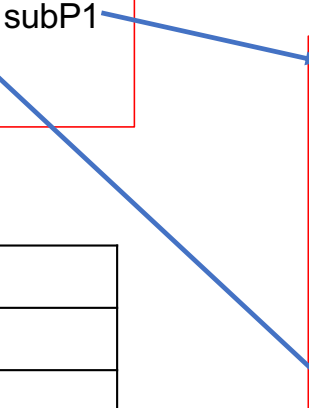
```
subP1
PUSH LR
@SI1_1 instr1
@SI1_2 instr2
@SI1_3 instr3
@SI1_4 BL
@SI1_5 instr5
@SI1_6 instr6
POP SP
@SI1_7 MOV PC,LR
```

```
subP2
@SI2_1 instr1
@SI2_2 instr2
@SI2_3 instr3
@SI2_4 instr4
@SI_5 instr5
@SI_6 MOV PC,LR
```



subP1

subP2

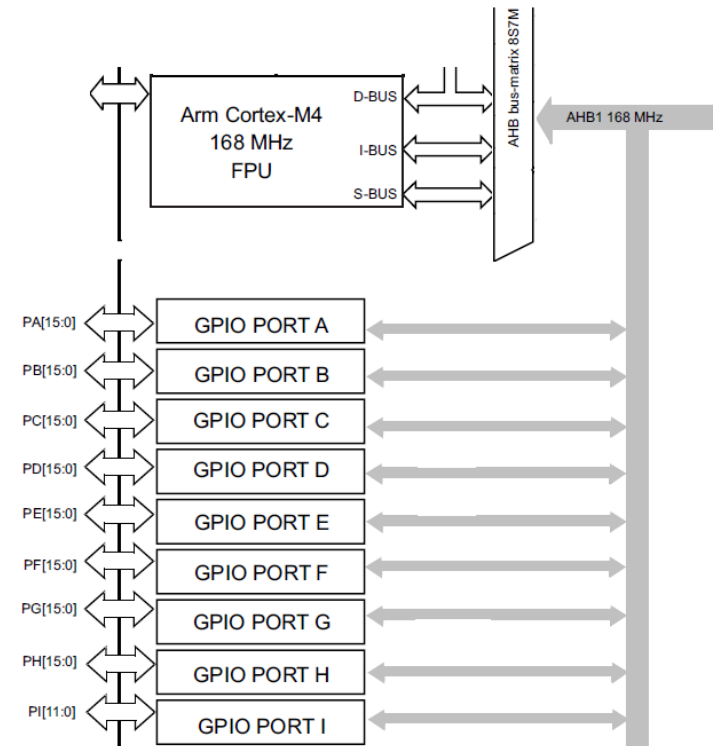


STM32F407/Cortex-04: GPIO

- **General Purpose Input Output**
- **Port**
 - I/O pins are **grouped** into Ports
 - Ports are **registers** inside the μC
 - Control the state of a pin
 - Read/Write the state of/to a pin,

STM32F405xx datasheet

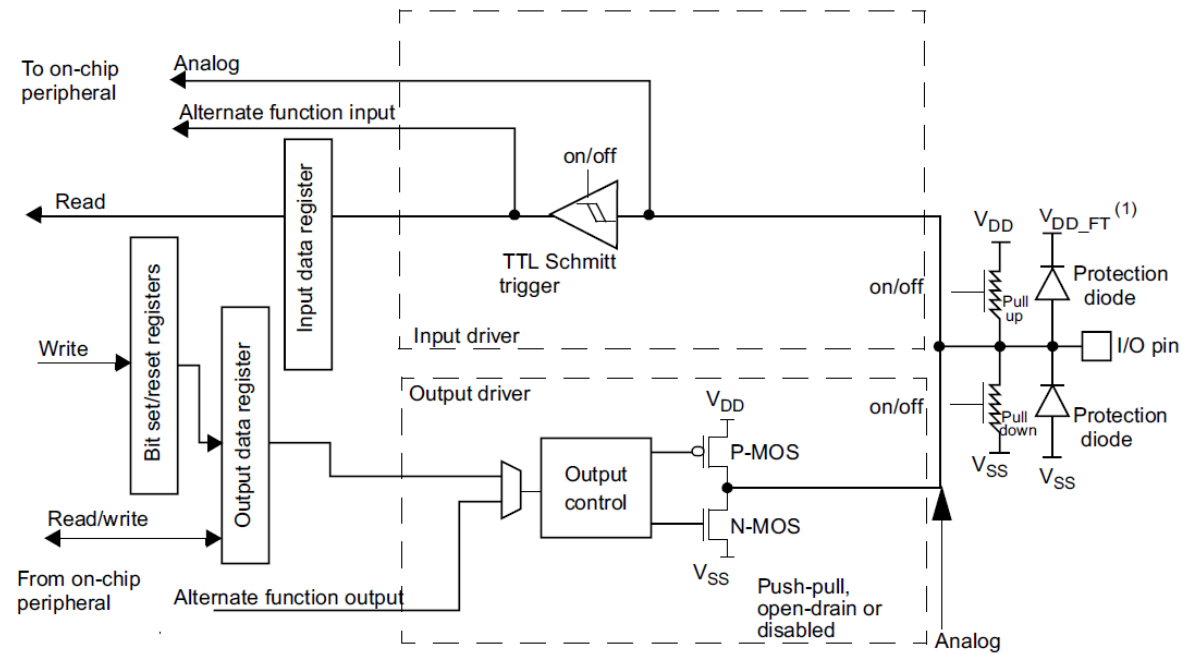
Figure 5. STM32F40xxx block diagram



GPIO

RM0090 Reference manual

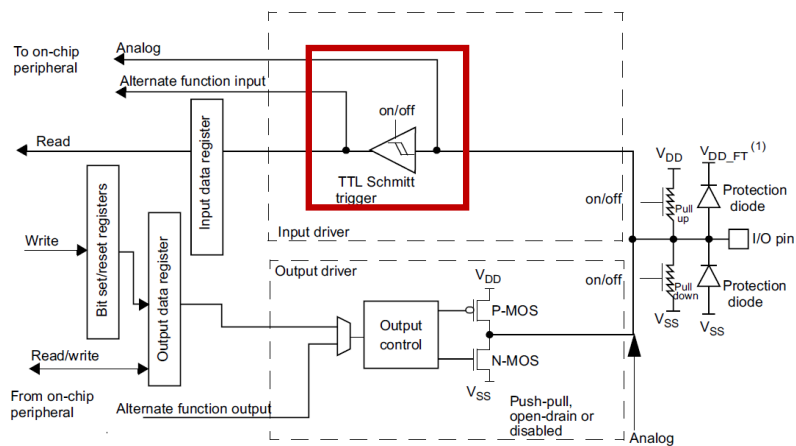
Figure 25. Basic structure of a five-volt tolerant I/O port bit



GPIO: input/Output mode

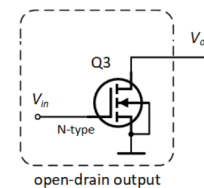
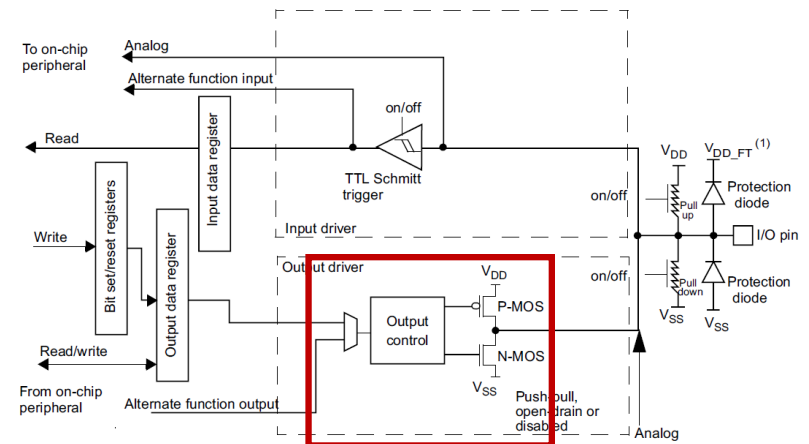
- Input**

- Schmitt Trigger: off
- Floating, Pull-up, Pull-down.

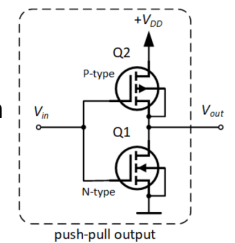


- Output**

- Schmitt Trigger: off
- Pull-up, Pull-down: off
- P-MOS, N-MOS: Open-drain, Push-pull

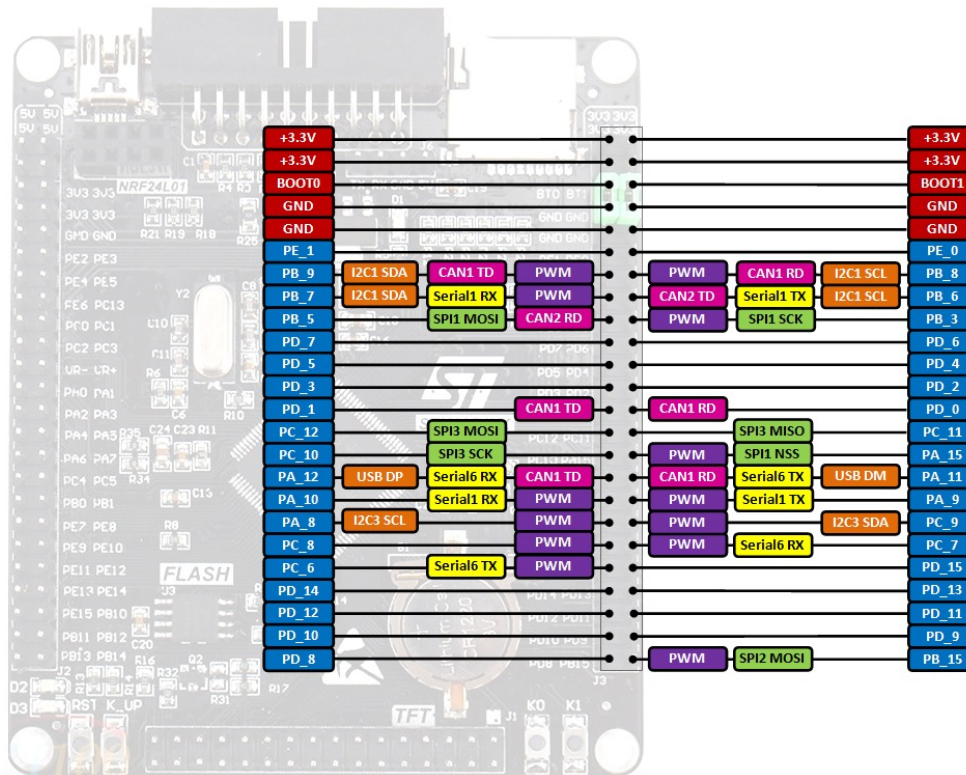


'0': N-MOS on, '1': port 'Z'
Sink current



'0': N-MOS on, '1': P-MOS on
Source/Sink current

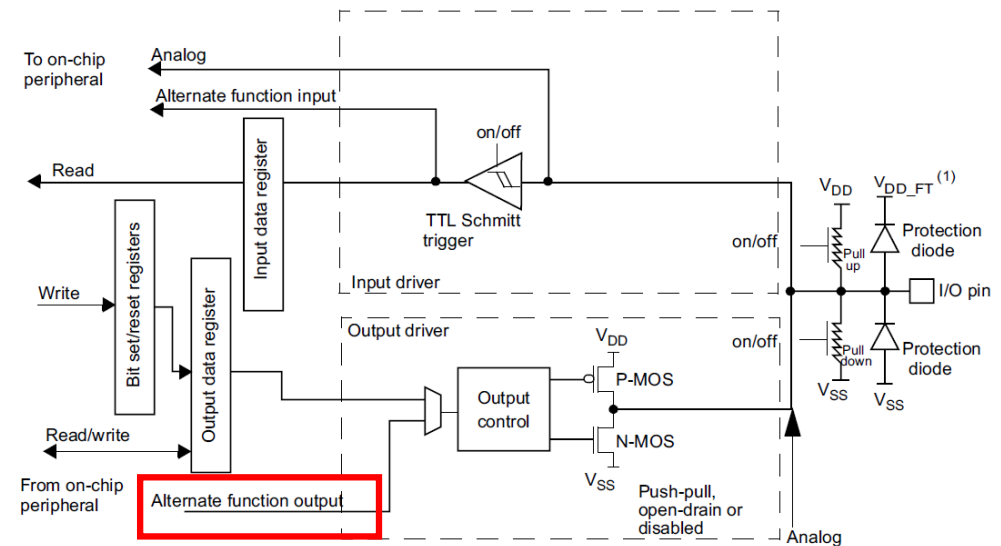
GPIO: alternate function



GPIO: alternate function

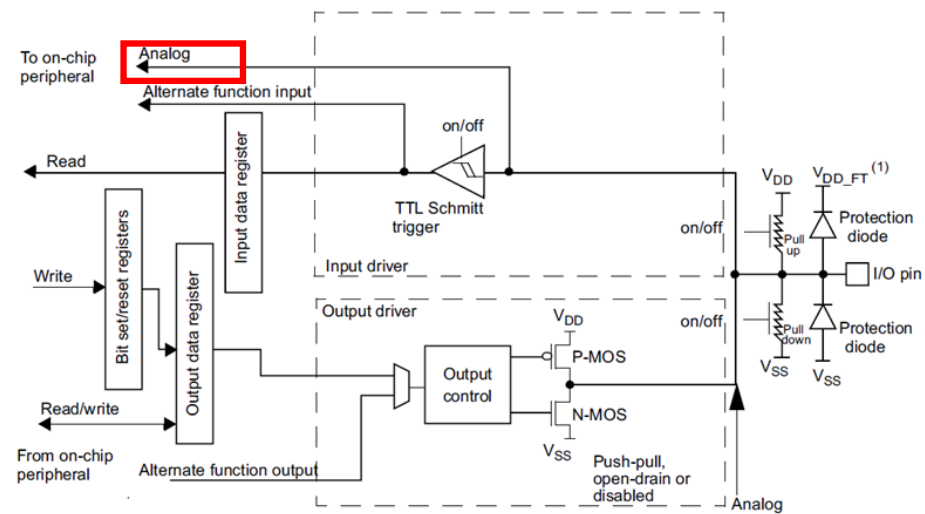
Figure 15. Alternate function configuration

- **AF:**
 - Open drain or Push-pull
 - Schmitt Trigger: on
 - Pull-up/Pull-down: off



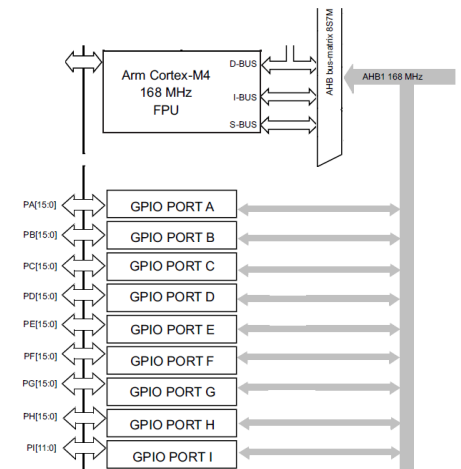
GPIO: analog input

- **Analog:**
 - Schmitt Trigger: off
 - Pull-up/Pull-down: off



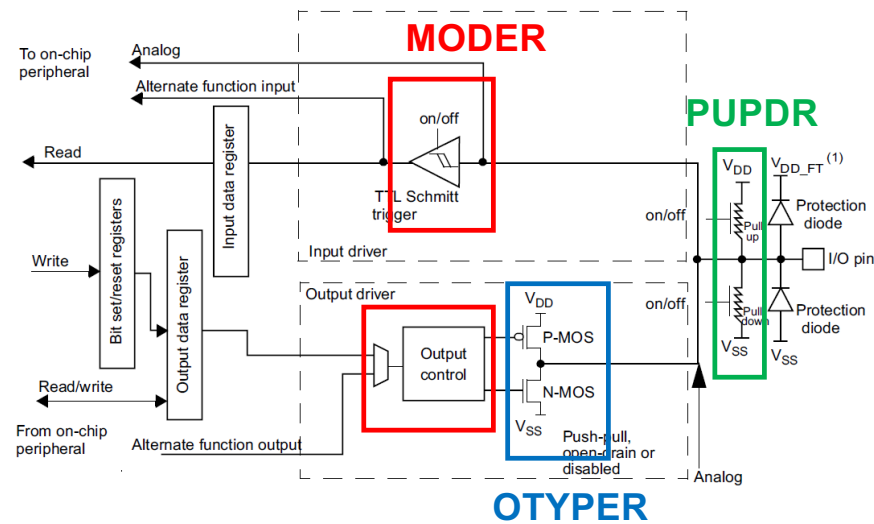
GPIO: registers

- Each I/O port bit is freely programmed
- However the I/O port registers have to be accessed as 32-bit words.
- **Data registers:**
 - **IDR** (Input Data register) and **ODR** (Output Data register)
 - Atomic read/modify accesses (bit-banding)
 - **BSRR** (Bit Set/Reset Register)
- **Configuration registers:**
 - **MODER**: Mode Register (input, output, alternate, analog)
 - **OTYPER**: Output Type Register (push-pull, open drain)
 - **OSPEEDR**: Output Speed Register (low, medium, fast, high)
 - **PUPDR**: Pull-Up/Pull-Down Register
 - **AFRL, AFRH**: Alternate Function Register (64 bits)
- Configure port Clock:
 - **RCC_AHB1ENR**
- Register notation:
 - GPIOx_RegisterName, x=A..G,
 - Exemple: GPIOx_IDR



GPIO: registers

- **Data**
 - IDR, ODR,
 - BSRR.
- **Configuration**
 - MODER,
 - OTYPER,
 - OSPEEDR,
 - PUPDR,
 - AFRL,
 - AFRH.
 - RCC_AHB1ENR



GPIO: registers

For programmer:

- Use memory mapping of each port.
- Write/Read in memory map ↔ Write/Read to port registers.

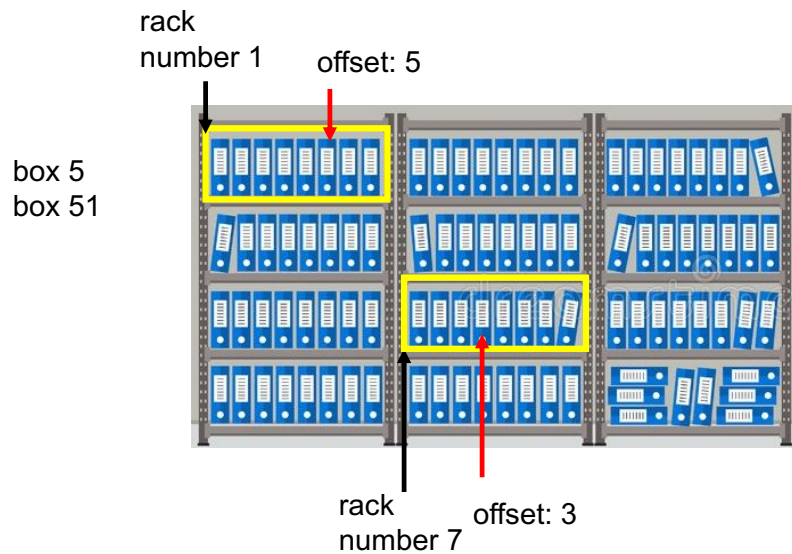
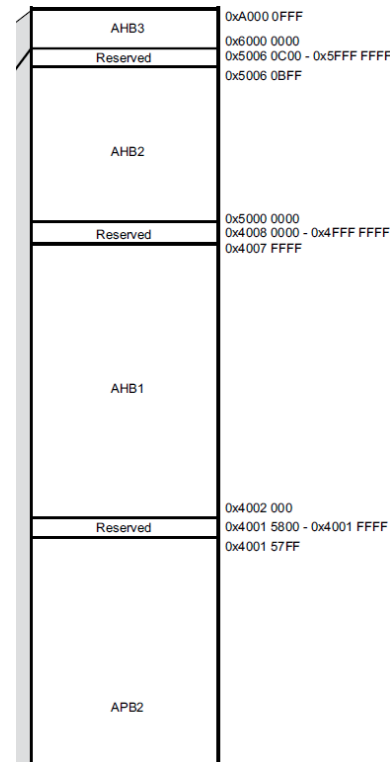


Figure 18. STM32F40xxx memory map



STM32F405xx datasheet

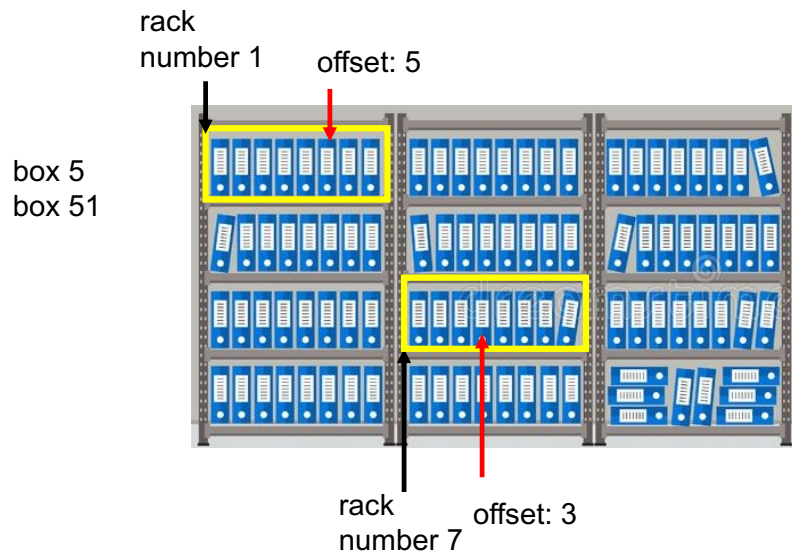
Table 10. register boundary addresses

Base Address	GPIOx
0x4002 2800	GPIOK
0x4002 2400	GPIOJ
0x4002 2000	GPIOI
0x4002 1C00	GPIOH
0x4002 1800	GPIOG
0x4002 1400	GPIOF
0x4002 1000	GPIOE
0x4002 0C00	GPIOD
0x4002 0800	GPIOC
0x4002 04000	GPIOB
0x4002 0000	GPIOA

GPIO: registers

For programmer:

- Use memory mapping of each port.
- Write/Read in memory map ↔ Write/Read to port registers.



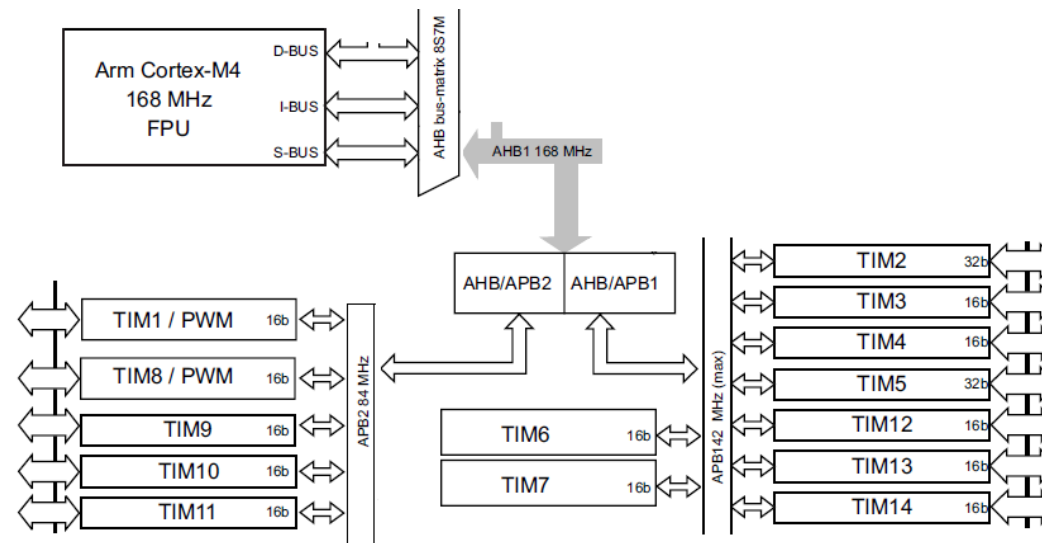
RM0090 Reference manual

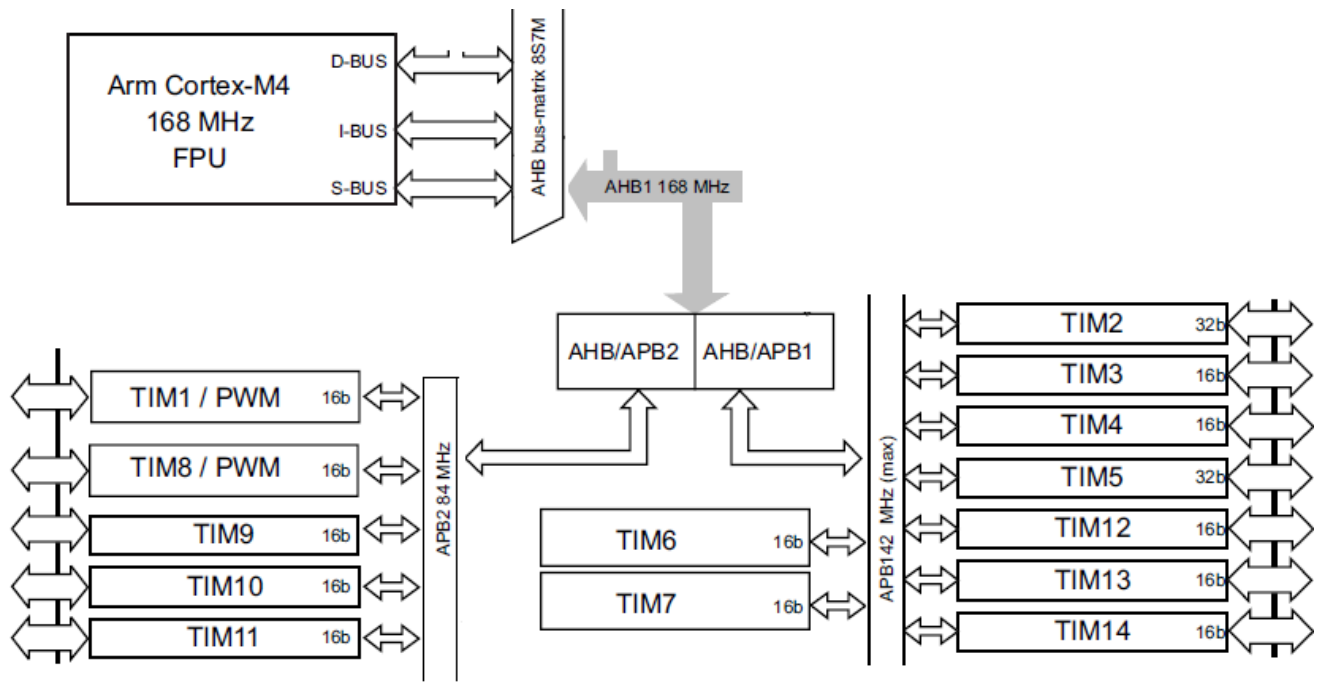
Table 39. GPIO register map and reset values

Offset	Register
0x00	MODER
0x04	OTYPER
0x08	OSPEEDR
0x0C	PUPDR
0x10	IDR
0x14	ODR
0x18	BSRR
0x1C	LCKR
0x20	AFRL
0x24	AFRH

Timers

- Timers types:
 - **SysTick**: common for all Cortex-M
 - **Basic** : for interrupts and DMA request (TIM6, TIM7)
 - **General Purpose** : basic, compare, capture, sensor fast interrupt (TIM2-5, TIM9-14)
 - **Advanced**: TIM1 and TIM8

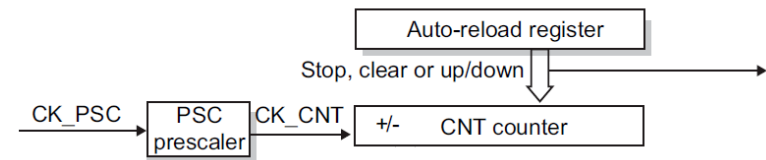




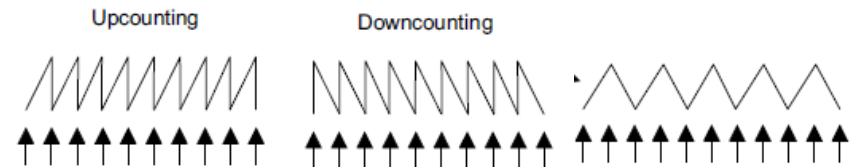
Timers: Time-base unit

Main block:

- **CNT**: Counter register,
- **PSC**: Prescaler register,
- **ARR**: Auto-Reload register



- **Upcounting** mode:
 - Counts from 0 to the auto-reload value,
 - Generates an **overflow** event.
- **Downcounting** mode:
 - Counts from the auto-reload value to 0,
 - Generates an **underflow** event.
- **Center-aligned** mode
 - Counts from 0 to the auto-reload value,
 - Generates an overflow event,
 - Counts from the auto-reload value down to 1,
 - Generates a counter underflow event.



Timers: Input Capture

Input Capture channel:

- Measure the input frequency,
- When an input signal is received
 - A timestamp in memory is recorded,
 - A flag is set to indicate that an input is captured,
 - We can read out the capture value through interrupt or event polling.
- Each Capture channel:
 - Capture/compare register: to latch the counter value.
 - Input stage: TI1 input is sampled, filtered (TI1F), used to generate trigger signal TI1FP1. TI1FP1 (or prescaled signal IC1PS1) is used as the capture command.
 - Output stage: generates the reference waveform.

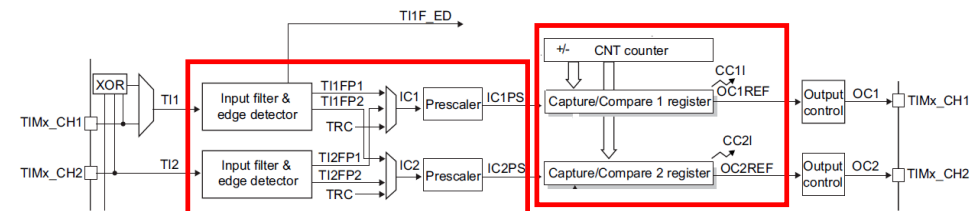
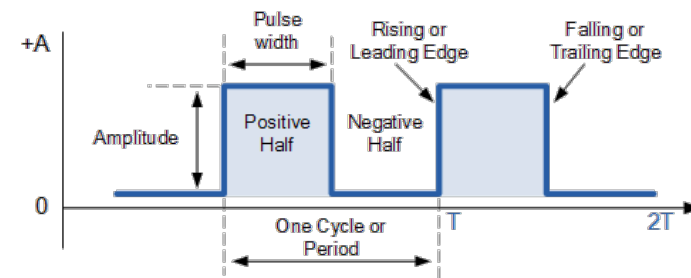
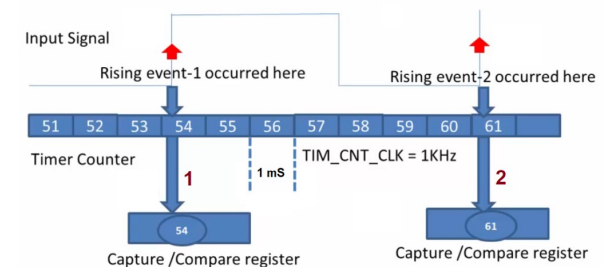
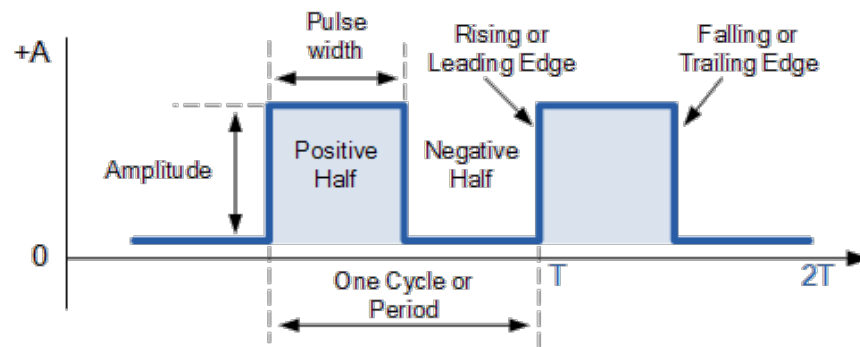
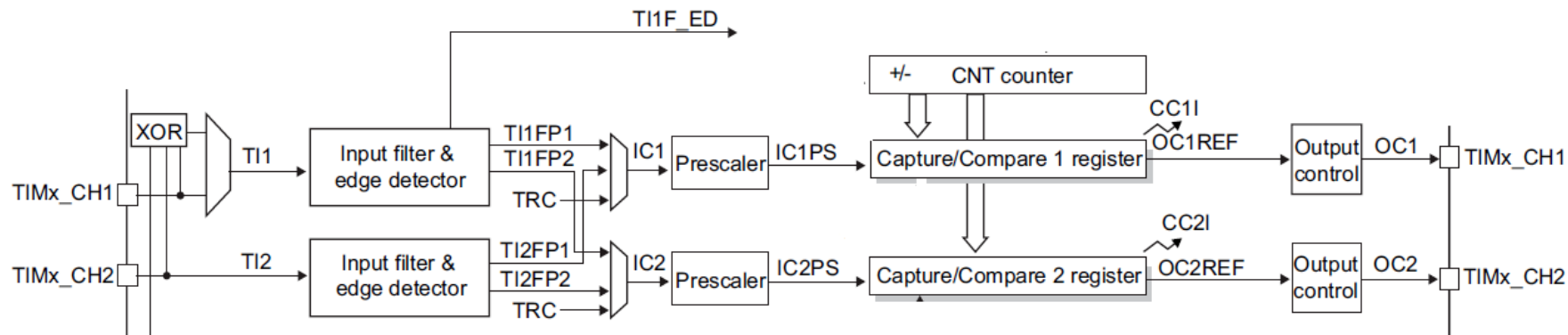


Figure 134. General-purpose timer block diagram RM0090 (TIM2 to TIM5)

Ex: time base = 1KHz, time resolution = 1ms. Each time the CAPTURE is triggered, an Interrupt is generated which to save the value contained in the register: CAPTURE (1). The next time the CAPTURE input is triggered, we will do the same procedure again, saving the new value contained in the CAPTURE register (2).

Calculate the Time Period: timer value difference $61 - 54 = 7 \times 1\text{ms} = 7\text{ms}$.

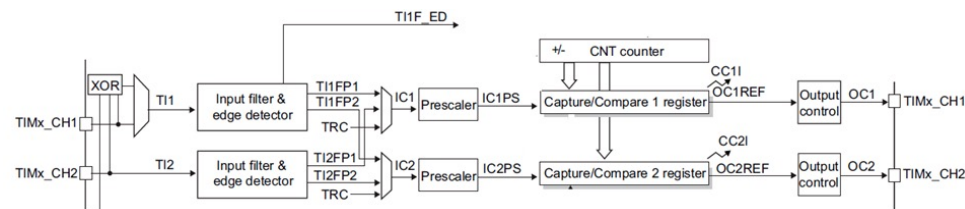
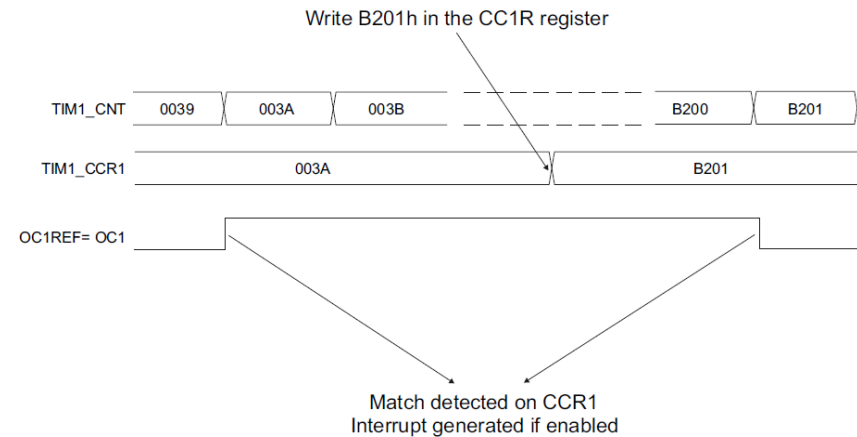




Timers: Output Compare

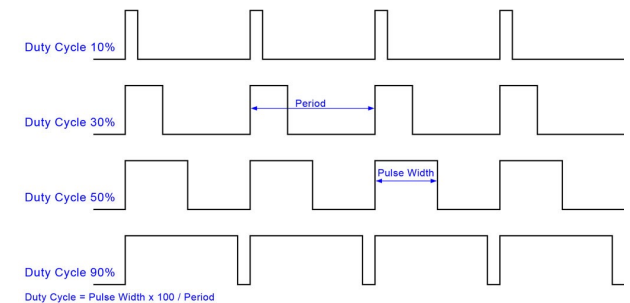
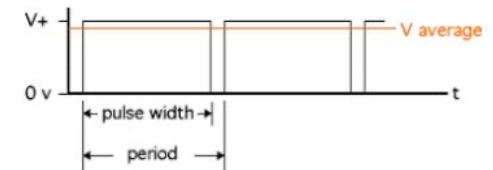
Output compare:

- Used to control an output waveform for example,
 - Connect timer output channel to a GPIO pin,
- Compare CNT to value in CCR. When $CNT = CCR$:
 - The corresponding output pin is assigned to the programmable mode: set, reset, toggle, unchanged.
 - Set a flag in the interrupt status register.
 - May generate an interrupt,
 - May send a DMA request



Timers: PWM

- Frequency: ARR, duty cycle: CCR, The PWM mode can be selected independently on each channel:
 - PWM mode 1: OCxM = 110, PWM mode 2: OCxM = 111 in CCMR.
 - The user must enable the corresponding preload register: OCxPE bit in CCMR
 - Polarity: CCER[CCxP]
 - Output enable: CCER[CCxE]
- The user has to initialize all the registers by setting:
 - UG bit in EGR.



Timers: registers

TIMx, x=10/11/12/14

Reserved										CKD[1:0] rw rw		ARPE rw	Reserved				OPM rw	URS rw	UDIS rw	CEN rw
Reserved													CC1IE rw	UIE rw						
Reserved										CC1OF rc_w0		Reserved				CC1IF rc_w0	UIF rc_w0			
Reserved								OC1M[2:0] rw rw rw rw				OC1PE rw	OC1FE rw	CC1S[1:0] rw rw						
Reserved													CC1NP rw	Res.	CC1P rw	CC1E rw				
CCR1[15:0] rw/ro rw/ro rw/ro rw/ro rw/ro rw/ro rw/ro rw/ro rw/ro rw/ro rw/ro rw/ro rw/ro rw/ro rw/ro rw/ro																				

CEN: Counter Enable

CR: Control Register.

DIER: Interrupt Enable Register

UIE: Update Interrupt Enable

SR: Status Register

UIF: Update Interrupt Flag

CCMR1: Capture/Compare Mode Register

OC1M: Output Compare Mode (PWM 1, 2)

CCER: Capture/Compare Enable Register

CC1E: Capture/Compare Ch1 output Enable

CCR1: Capture/Compare Register 1

Timers: registers

TIMx, x=10/11/12/14

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

CNT: Counter

ARR: Auto-Reload Register

PSC: Prescaler

Timer frequency:

$$F_{CNT} = \frac{F_{CK}}{PSC + 1} \rightarrow T_{CNT} = T_{CK}(1 + PSC)$$

16-bit counter: overflow occurs at $(65535+1) T_{CNT}$

By using ARR:

$$T_{overflow} = T_{CNT}(1 + ARR)$$

Example: 1 second counter, $F_{CK}=16$ MHz

With PSC = 16000-1 $\rightarrow T_{CNT}= 1$ ms

With ARR = 1000-1 $\rightarrow T_{overflow}= 1$ s

Example: 1 second counter, $F_{CK}=16$ MHz

With PSC = 16-1 $\rightarrow T_{CNT}= 1\mu$ s

With ARR = 1000000-1 $\rightarrow T_{overflow}= 1$ s (not on 16-bits Timer)