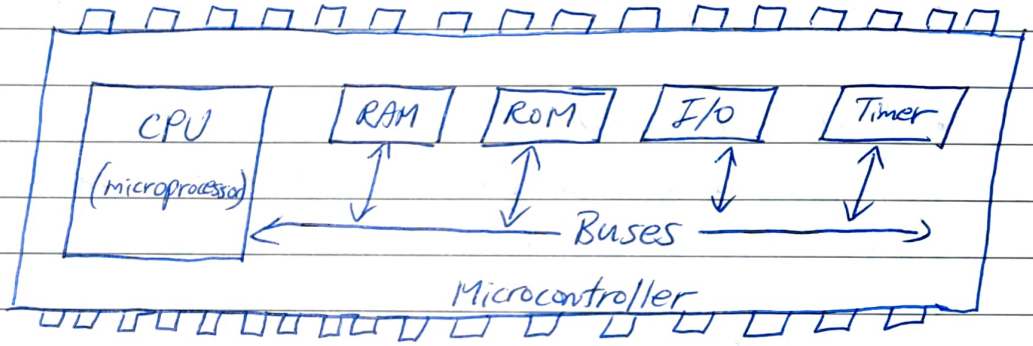
__/__/____

Embedded Systems

- Embedded (hidden) (invisible)
  micro (small)
  computer (contains processor, memory and means to exchange data)
  system (multiple components interfaced together for common purpose).

- In embedded systems, we use ROM for storing software and fixed constant data and RAM for temporary data.



- Microprocessors do not contain RAM, ROM or I/O peripherals hence, they must be externally connected through buses.

- Embedded systems do not look/behave like typical computers. Most of them do not have a keyboard, display ~~of~~ or storage.

- They can be developed in two ways:

(a) Microcontroller-based (use microcontrollers such as LPC1768 in ARM Cortex-M series) (do not run an operating system) (low cost) (low performance)

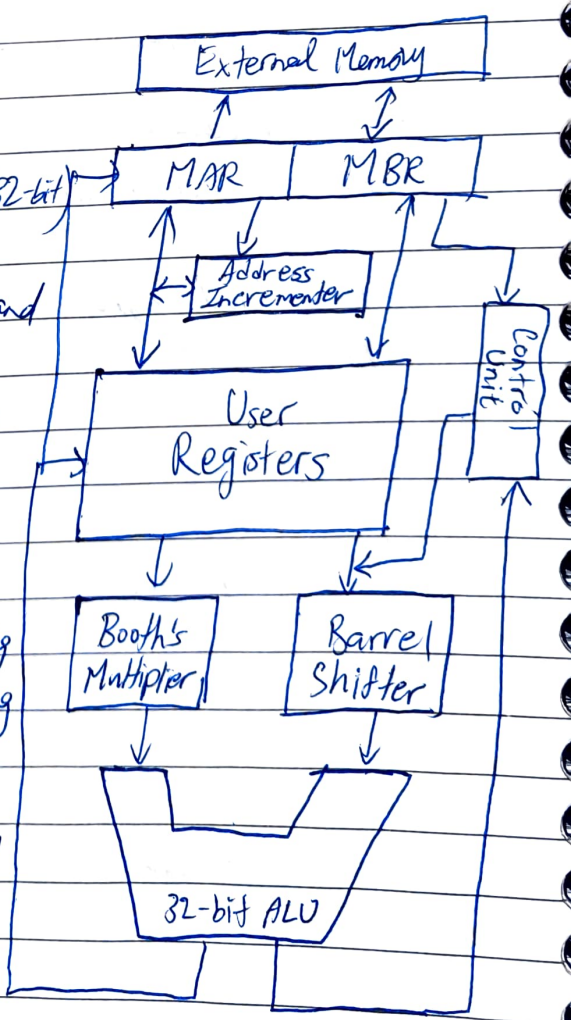
(b) Microprocessor-based (use more powerful microprocessors such as those in ARM Cortex A-series) (run an operating system) (designed on development platform, then standalone ES) (high performance)

Exo — 32-bit microcontrollers such as x86, PowerPC, PIC32.

- Factors important in choosing microcontroller : (a) chip characteristics (clock speed - how fast instructions are executed ; power consumption, cost, on-chip memory and peripherals) (b) available resources (IDE and Compiles - development environment like Keil, legacy Software multiple sources of production - supply chain reliability)
- ARM (Advanced RISC Machine) defines CPU architecture.
- Major drawback of using ARM chips is the incompatibility of peripherals designed by different manufacturers, despite the CPU architecture being the same.

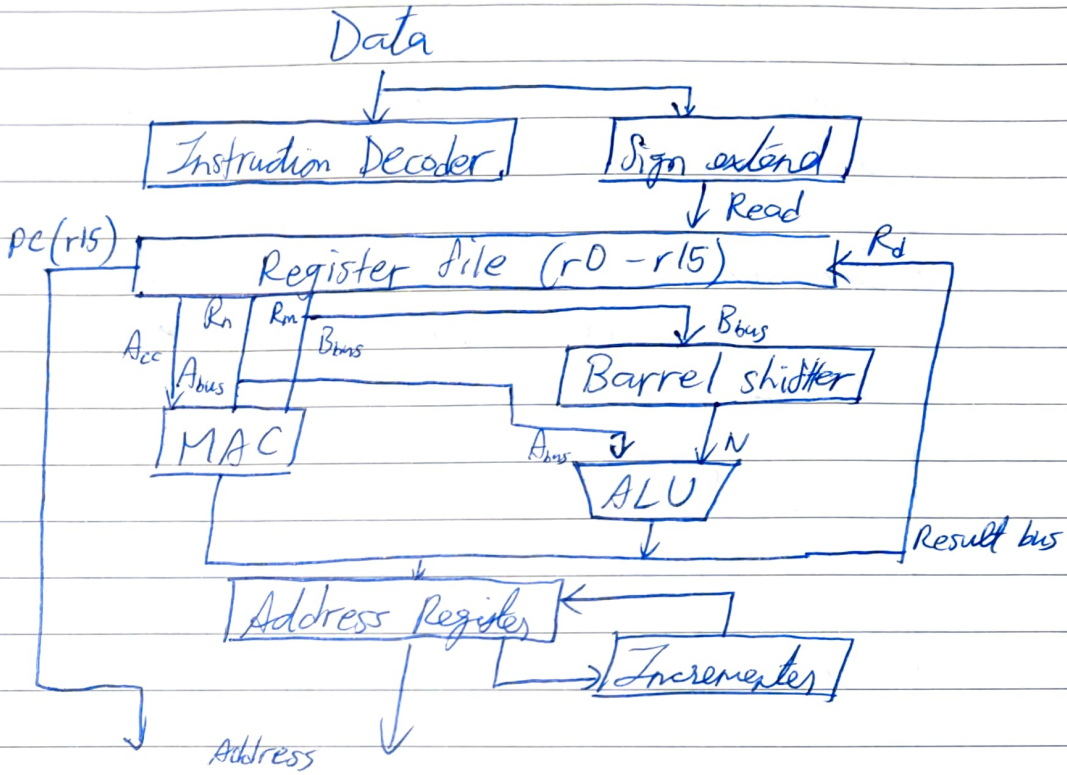
★ Architecture

- Uniform fixed-length instruction (32-bit)
- Large array of uniform registers and a load/store model of data processing where operations can only operate on registers, not directly on memory
- Separate ALU and shifter giving additional control over data processing and maximum execution speed.
- Auto-increment and auto-decrement addressing modes to improve operation of program loops.



__/__/____

- Conditional execution of instructions to reduce pipeline flushing.



- Data bus is where data enters CPU core. Load-store architecture is used by ARM processors, where load instructions copy data from memory to core registers, and store instructions copy data from registers to memory.

- Since ARM core is 32-bit processor, when 8-bit and 16-bit integers are read from memory and stored in a register, sign extend hardware transforms them to 32-bit values

- Rn and Rm are two source registers while Rd is the result / destination register. Internal buses A and B are used to read source operands from register file.

- ALU or MAC (multiply-accumulate unit) compute result using Rn and Rm register values and outcome is written directly to register file Rd. ALU is used in load and store

instructions to create an address stored in address registers and broadcast over Address bus.

- Register R_m can also be preprocessed in barrel shifter before it enters ALU.
- Incrementer updates address register before core reads or writes next register value. Process continues executing instructions until exception or interruption.
- Barrel Shifter: Digital shifter capable of shifting a data word by specific no. of bits in single clock cycle.
- MAC: Supports basic summation operations on data in registers
- Address register: Contains address from which data/instruction needs to be fetched. Connected to incrementing unit.

Ex: Shift instructions in ARM
LSL #5 (multiply by 32) (Shifts left by specified amount (powers of 2))

```
MOV R0, #3
MOV R1, #7
ADD R2, R1, R0, LSL #3   (R2 = R1 + R0*8)
```

★ Processor Modes

① User Mode (usr): Normal, least-privileged execution mode for applications.

② System Mode (sys): Privileged mode, using same registers as User mode. Used for OS-level tasks.

③ Supervisor Mode (svc): Used by OS, typically entered via system calls or exceptions.

④ FIQ Mode (fiq): Fast interrupt mode for high priority, fast response interrupts.

⑤ IRQ Mode (irq): General purpose interrupt handling mode.

⑥ Abort Mode (abt): Activated on memory access faults (data or instruction fetch aborts).

⑦ Undefined Mode (und): Handles undefined instruction exceptions.

☆ Registers

- In CPU, registers are used to temporarily store information.
- With 32-bit datatype, any data larger than 32-bits must be broken into 32-bit chunks before processing, although many assemblers support 8-bit, 16-bit types.
- 1 byte = 8 bits.
  One word = 4 bytes = 32 bits
  Half word = 2 bytes = 16 bits
- In ARM, there are 13 general purpose registers (r0-r12) that are 32-bits wide, that can be used by all arithmetic and logic instructions. ARM core also has 3 special purpose registers r13, r14, r15.

r0					
r1					
r2					
r3					
r4					
r5					
r6					
r7					
r8	r8_fiq				
r9	r9_fiq				
r10	r10_fiq				
r11	r11_fiq				
r12	r12_fiq				
r13 sp	r13_fiq	r13_irq	r13_svc	r13_undef	r13_abt
r14 lr	r14_fiq	r14_irq	r14_svc	r14_undef	r14_abt
r15 pc					
cpsr					
-	spsr_fiq	spsr_irq	spsr_svc	spsr_undef	spsr_abt

- Unbanked Registers (r0-r7): Registers that remain the same across all modes. They are completely general-purpose registers with no special uses. Used for general execution.
- Banked Registers (r8-r14): Registers with different versions accessible depending on processor mode. By having dedicated registers, it helps improve interrupt response time.

★ Instructions

- ARM CPU uses tri-part instruction format:

instruction destination, source1, source2
(register / constant / memory)

Ex: MOV destination, source

Note: To write a comment in Assembly, use ';'

① ADD Rd, Rn, Op2 ($R_n + Op2 \rightarrow Rd$)

__/__/____

② ADC Rd, Rn, Op2 (Rn + Op2 (with carry) → Rd)
③ AND Rd, Rn, Op2 (Rn & Op2 (AND) → Rd)
④ BIC Rd, Rn, Op2 (Rn & (~~Op2~~) → Rd)
⑤ CMP Rn, Op2 (Compare Rn and Op2 and set status bits)
⑥ CMN Rn, Op2 (Compare Rn and (-Op2) and set status bits)
⑦ EOR Rd, Rn, Op2 (Rn ⊕ Op2 (XOR) → Rd)
⑧ MVN Rd, Op2 (~~Op2~~ → Rd)
⑨ MOV Rd, Op2 (Op2 → Rd)
⑩ ORR Rd, Rn, Op2 (Rn | Op2 (OR) → Rd)
⑪ RSB Rd, Rn, Op2 (Op2 - Rn → Rd)
⑫ RSC Rd, Rn, Op2 (Op2 - Rn (with carry) → Rd)
⑬ SBC Rd, Rn, Op2 (Rn - Op2 (with carry) → Rd)
⑭ SUB Rd, Rn, Op2 (Rn - Op2 → Rd)
⑮ TEQ Rn, Op2 (Rn ⊕ Op2 and set status bits of CPSR)
⑯ TST Rn, Op2 (Rn & Op2 (AND) and set status bits)

— Addressing Modes

- Register direct: MOV R0, R1 (R1 → R0)
- Direct: LDR R0, MEM (Value from memory address → R0)
- Immediate: MOV R0, #15
- Register indirect: LDR R0, [R1] (Value from address stored in R1 → R0)
- Register indirect with offset: LDR R0, [R1, #4] (R1+4 → R0)
- Register indirect pre-increment: LDR R0, [R1, #4]! (R1+4 → R0, then update R1 with R1+4)
- Post-increment: LDR R0, [R1], #4 (R1 → R0 then R1 → R1+4 update)

__/__/__

- Register Indirect - Register Indexed: LDR R0, [R1, R2]
  (R1 + R2 → R0)
- Register Indirect Indexed with Scaling: LDR R0, [R1, R2, LSL #2]
  (Load from R1 + (R2 << 2))
- Program Counter Relative: LDR R0, [PC, #offset]
  (PC + offset → R0)

Exa-

```
MOV R2, #0x5
MOV R1, #0x2
ADD R2, R1, R2   (R2 = 5+2 = 7)
ADD R2, R1, R2   (R2 = 7+2 = 9)
MOV R5, #0x20
STRB R2, [R5]    (R5 = 9)
```

Location	Data
R2	9
R1	2
0x20	9

ARM CPSR

- CPSR (current program status register) is the flag register in ARM, accessible to all processor modes, that contain condition code flags, interrupt disable bits, and other status info.
- Each exception mode also has a SPSR (saved program status register) used to preserve value of CPSR when exception occurs.

N	Z	C	V	Reserved	I	F	T	M4	M3	M2	M1	M0
31	30	29	28		7	6	5	4	3	2	1	0

- Control Flags (0-7): Change when exception arises and can be altered by software only when in privileged mode.
- Bits 0-4 (Mode Select Bits) (Processor modes)

User	10000
FIQ	10001

__/__/____

IRQ	10010
Supervisor	10011
Abort	10111
Undefined	11011
System	11111

- Bit 5 (Thumb State): Thumb state is a mode of operation for ARM processor that allows it to execute 16-bit instructions. Thumb state uses a compressed instruction set of ARM (subset) (reducing memory consumption and improving code density)

- Bit 6 and 7 (6 → FIQ, 7 → IRQ): If set to 1, interrupt is disabled, if 0, then enabled.

- Bit 28 (Overflow, V) (1 if signed overflow occurs in +/-)

- Bit 29 (Carry C): For addition & CMN, 1 if carry
  For subtraction & CMP, 0 if borrow
  In shift operation, C → last shifted out bit

- Bit 30 (Zero Z): Z → 1 if result = 0 (CMP equal)

- Bit 31 (Negative N): N → 1 if result is (-)ve (2's complement signed integers)

Ex- Show status of C and Z flags when

R1 = 0x0000009C
R2 = 0xFFFFFF64
ADD R2, R1, R2 (C = 1, Z = 1)

```
  0000 0000 0000 0000 0000 0000 1001 1100
+ 1111 1111 1111 1111 1111 1111 0110 0100
(1)0000 0000 0000 0000 0000 0000 0000 0000
```

Note	Instruction	Flags Affected
	ANDS	C, Z, N
	ORRS	C, Z, N
	ADDS	C, Z, N, V
	MOVS	C, Z, N
	SUBS	C, Z, N, V
	B	No flags (Branch)

Ex: Find status of C, Z, N flags in:
LDR R1 = 0x0F000006
MOV R2, R1, LSL #8 (C=0, Z=0, N=0)

R1 → 0000 1111 0000 0000 0000 0000 0000 0110
→ 0000 0000 0000 0000 0000 0110 0000 0000
(or) 0x600

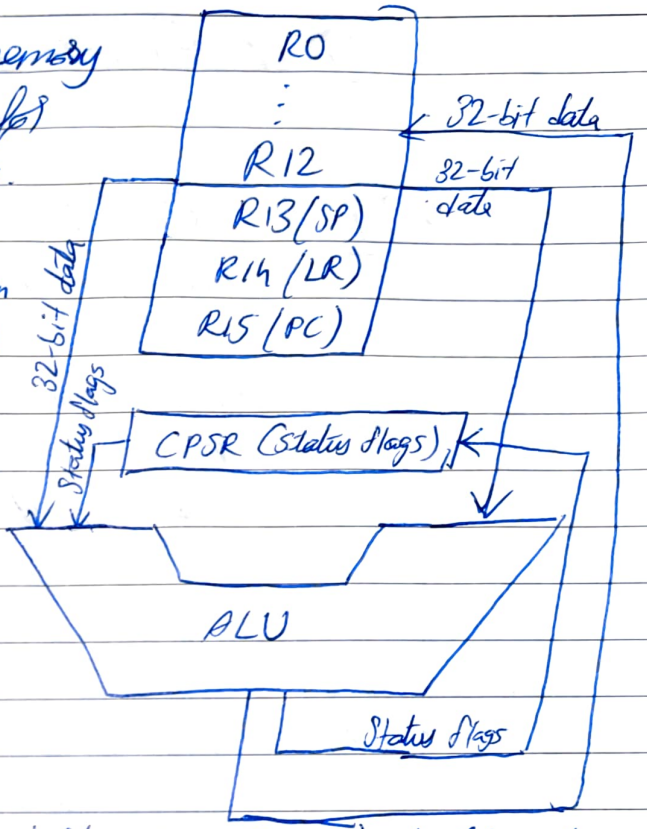
Note: (Conditional) (Opcode)	Instruction	Flags Affected	
0010	BCS	Branch if C=1	(Carry Set)
0011	BCC	Branch if C=0	(Carry Clear)
0000	BEQ	Branch if Z=1	(if equal)
0001	BNE	Branch if Z=0	(if not equal)
0100	BMI	Branch if N=1	(if minus)
0101	BPL	Branch if N=0	(if plus)
0110	BVS	Branch if V=1	(if overflow set)
0111	BVC	Branch if V=0	(if overflow clear)

☆ Special Purpose Registers

- Function of each is fixed by CPU designer at time of design since it is used to track CPU status.

__/__/____

- R13 (Stack Pointer): Holds memory address of top of stack. Used for function calls, local variables.
- R14 (Link Register): Stores return address when function (subroutine) is called.
- R15 (Program Counter): Holds address of next instruction to be executed.



✡ Assembly Language (ARM)

- ARM has four data types: bit, byte (8-bit), half word (16-bit) and word (32-bit).
- Hexadecimal: #0x99, Decimal: #12, Binary: #2_10010011
  ASCII: #'2' or 32 in hex, Base n: #n_ONNNNNN
- Assembler Directives:
- Commands that direct the assembly process, do not generate any machine code.
- AREA: Tells assembler to define new section of memory.
- READONLY: Attribute given to area of memory which can only be read from. Default for CODE.
- READWRITE: Attribute given to area of memory which can be read from / written to. Default for DATA.

- CODE : Attribute given to area of memory used for executable machine instructions (READONLY)
- DATA : Attribute given to area of memory used for data and no instructions can be placed (READWRITE)
- ALIGN : Ensures data and instructions are stored at properly aligned memory addresses to prevent misalignment. If ALIGN=3, then information should be placed in memory with addresses of 2^3. 0x50000000, 0x50000008, ...
- EXPORT : Declares a symbol to be globally accessible. Used for shared functions and source files.
- ENTRY : Declares entry point of program. (First instruction to be executed)
- END : Indicates to the assembler the end of the source code.
- EQU : Used to define a constant value or fixed address.
  Ex: COUNT EQU 0x25
  MOV R2, #COUNT ; R2 = #0x25
- RN : Used to give CPU register a name.
  Ex: VAL1 RN R1
  MOV VAL1, #0x25
- DCB : Allocates one or more bytes of memory.
- DCW : Allocates halfwords of memory, aligned on 2 bytes
- DCD : Allocates words of memory, aligned on 4-bytes

__/__/____

- ADR: Loads address of label into register. No memory access. (Better performance ↑)
- LDR: Loads value stored at memory location into register.

Ex:
```
AREA EXAMPLE, READONLY, CODE
ENTRY
ADR R2, DTA
LDRB R0, [R2]
ADD R1, R1, R0
H1 B H1
```
(i) `DTA DCB 0x55 DCB 0x22` → 55 22

(ii) `DTA DCB 0x55 ALIGN 2 DCB 0x22` → 55 00 22

(iii) `DTA DCB 0x55 ALIGN 4 DCB 0x22` → 55 00 00 00 22

```
END
```

Memory

- ARM CPU has 4 GB of memory space, having addresses from 0x 00000000 to 0xFFFFFFFF.

Memory Type	Address Range	Purpose
Flash ROM (Code)	0x00000000 - 0x1FFFFFFF	Stores the program code (ROM)
SRAM (Data)	0x20000000 - 0x3FFFFFFF	On-chip RAM for storing variables, stack and heap.
Peripherals (I/O)	0x40000000 - 0x5FFFFFFF	Controls hardware like GPIO, timers.
External RAM	0x60000000 - 0x9FFFFFFF	Additional memory for large data storage.
Device Memory	0xA0000000 - 0xDFFFFFFF	Used for special hardware like SD card.
System	0xE0000000 - 0xFFFFFFFF	Manages processor settings, interrupts and timers.

__/__/__

Note: EEPROM (Electrically - Erasable Programmable Read Only Memory) is a non-volatile memory used to store important data permanently, if available. (SRAM is volatile memory)

- On-chip Flash ROM is programmed and erased in block size while EEPROM is byte-programmable and erasable.

Ex: For the given address range, calculate the space and amount of memory given to each section.

0x00100000 - 0x00100FFF

Range = last address - first address + 1

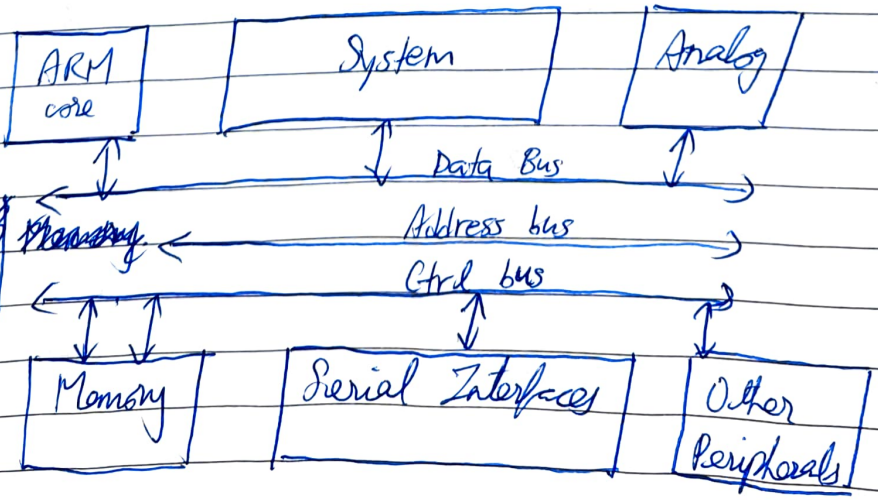
= OFF → (in decimal) 4095 + 1 = 4 KB

Ex: First address = 0x80000000

Space = 2KB = 2048 bytes - 1 = 000007FF

last address = 0x800007FF

- DRAM is the main memory for running programs while SRAM is the fast memory used for caching data. Flash memory store startup and boot programs (BIOS)



__/__/____

- Data bus (32-bit) carries actual data between processor and memory. It is divided into 4 lanes (8-bit) for better handling.
- Address bus (A31–A0) tells CPU where to find data. It supports any data size. A0 and A1 are used to select the correct portion of data from the bus (one of 8-bit chunks).
- AHB (Advanced High-Performance Bus) is the fast bus that connects CPU to on-chip memory (RAM, ROM) and some high speed peripherals.
- APB (Advanced Peripheral Bus) is a slower bus used for peripherals. There is a bridge connecting AHB to APB, that converts signals between them for efficient communication.

— • **Bus Cycle Time**

- Time taken for read/write operation of memory of I/O by CPU.
- Memory read cycle time is time from when CPU provides addresses at address pins to when data is expected at data pins. On-chip memory cycle time can be 1 clock, but for slow off-chip memory, extra time (wait state WS) will be taken.

Memory Cycle Time = Bus Cycle time × (Clocks + no. of WS)

$$\text{Bus Cycle Time} = \frac{1}{\text{Bus Speed}}$$

Ex: Bus Speed = 50 MHz, Bus Cycle time $= \frac{1}{50\,\text{MHz}} = 20\,\text{ns}$

Clocks = 2

Memory Cycle Time for 0WS = (2 + 0) × 20 = 40 ns

1WS = (2 + 1) × 20 = 60 ns

2WS = (2 + 2) × 20 = 80 ns

Bus Bandwidth (Rate of data transfer) (MB/s)

- Measure of how fast buses transfer information between CPU and memory / peripherals (wider data bus)

$$\text{Bus Bandwidth} = \frac{1}{\text{Bus Cycle Time}} \times \text{Bus Width in byte}$$

Ex: Bus speed = 100 MHz, Bus cycle Time = $\frac{1}{100\text{MHz}} = 10\text{ns}$
Clocks = 2
For ARM Thumb with 0WS = $\frac{1}{(2+0)\times 10\text{ns}} \times 2\text{ bytes}$
$= 100\text{ MB/s}$

For ARM Thumb with 1WS = $\frac{1}{(2+1)\times 10\text{ns}} \times 2\text{ bytes} = 66.6\text{ MB/s}$

For ARM with 0 WS = $\frac{1}{(2+0)\times 10\text{ns}} \times 4\text{ bytes} = 200\text{ MB/s}$

For ARM with 1 WS = $\frac{1}{(2+1)\times 10\text{ns}} \times 4\text{ bytes} = 133.2\text{ MB/s}$

Note: 4 GB of ARM memory space is organized as 1G × 32 bits since ARM instructions are 32-bit.

- Fetching of instructions in every clock cycle can work only if code is word-aligned, placed at address location ending with 0, 4, 8 or C, otherwise leads to memory access penalty.

☆ Load and Store

- In storing data ARM follows little endian convention where LSB is placed in the lower address and big endian does the opposite.

__/__/____

Ex: LDR R2, = 0x7698E89F
LDR R1, = 0x80000000
STR R2, [R1]

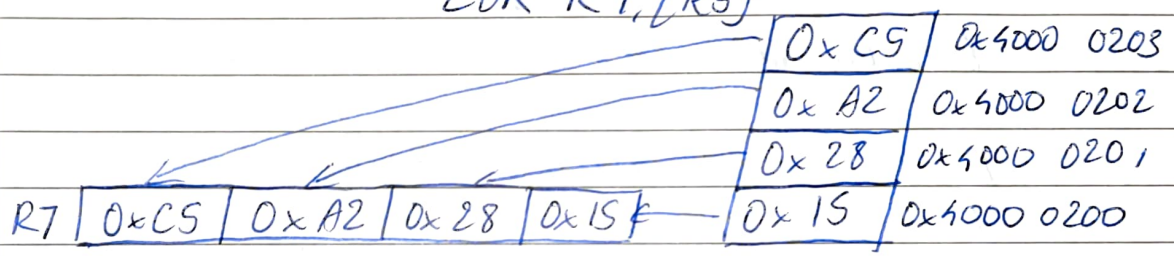
Little Endian :

76	98	E8	9F
0x80000003	0x80000002	0x80000001	0x80000000

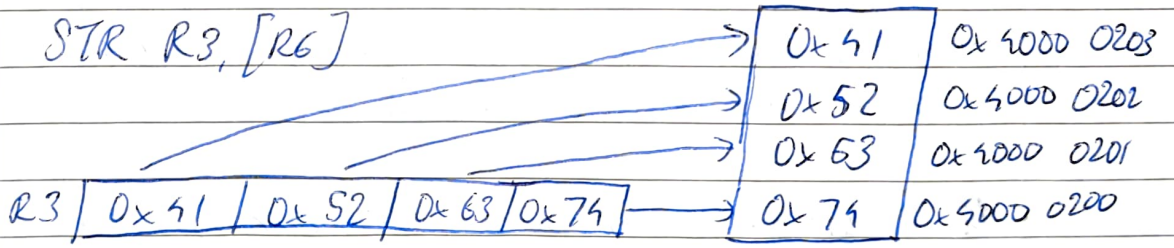
Big Endian :

9F	E8	98	76

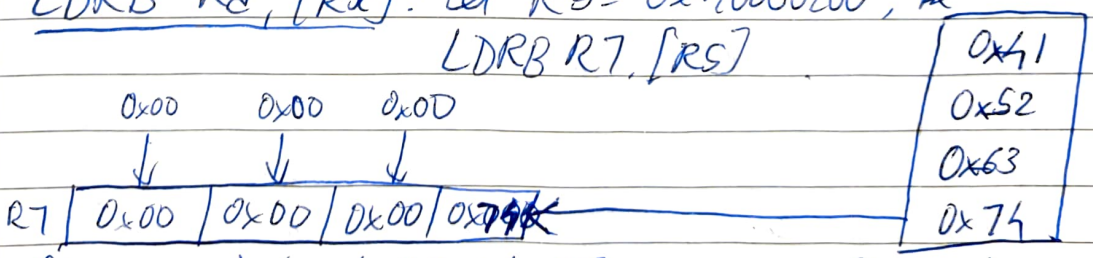
- LDR Rd, [Rx]: Let R5 = 0x40000200
LDR R7, [R5]



- STR Rx [Rd]: Let R6 = 0x40000200, R3 = 0x41526374



- LDRB Rd, [Rx]: Let R5 = 0x40000200,
LDRB R7, [R5]

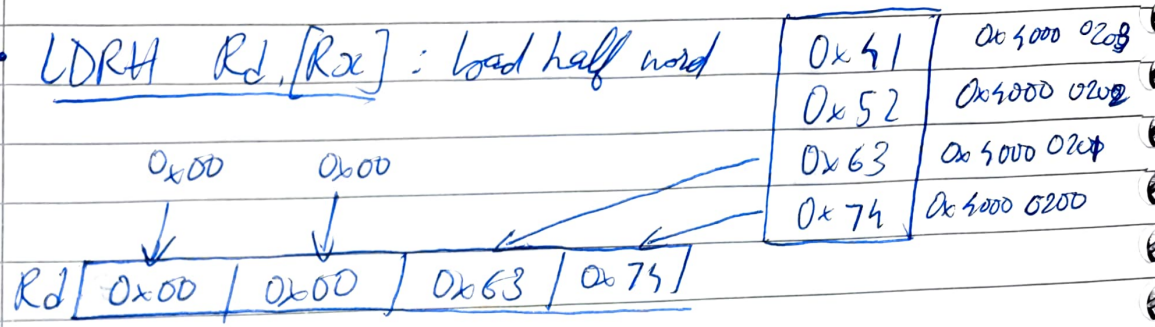


(Copy one byte of R5 into R7, remaining all zeros)

- STRB Rx [Rd]: Store the byte in Rx into location pointed by Rd.

__/__/__

- LDRH Rd, [Rx] : load half word



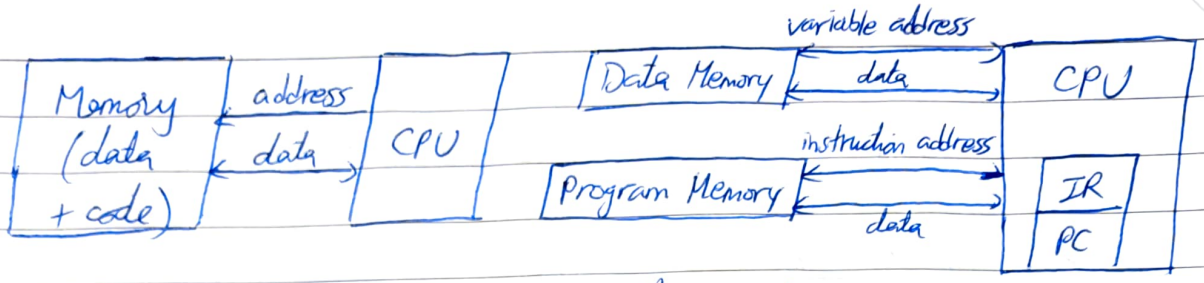
- STRH Rd, [Rd] : Store half word in Rx into location pointed to by Rd.

Note: For PC relative addressing mode,
Effective address, EA = Address of current instruction + 8 + offset.

Eg: LDR R0, [PC, #4] in address 0x10
EA = 0x10 + 8 + 4 = 0x1C

Von-Neumann Architecture	Harvard Architecture
A single memory stores both program instruction and data.	Memory is split into two parts for program instruction and data.
Shared signals and memory for both code and data.	Simultaneous access to both is possible, improving efficiency.
Single shared bus for instruction and data fetching.	Seperate buses for instructions and data fetching reducing bottleneck.
Programs can modify themselves since code and data are stored in same memory.	Programs cannot easily modify themselves as code memory is read only.

__/__/____



Von - Neumann Harvard

☆ RISC Architecture

- To increase processing power of CPU, ARM designers have : (a) increased clock frequency of chip (b) used Harvard architecture (c) changed internal architecture of CPU and used RISC architecture

– Features

- Fixed instruction size, helps CPU to decode instructions quickly
- Large no. of CPU registers, which avoids the need to access memory multiple times.
- Small instruction set, leading to large program size, using more memory. Commonly used for high-level language like C.
- Single-cycle execution and ~~off~~ seperate buses for data and code.
- Load/store architecture preventing RISC microprocessors from data manipulation in memory.

Arithmetic and Logic Instructions in ARM

☆ Arithmetic Instructions

- Unsigned numbers comprise of data in which all bits are used to represent data and none are ~~used~~ set aside for sign denotion.

	Data Size	Bits	Decimal	Hex
STRB	Byte	8	$0-(2^8-1)$	0x0 - 0xFF
STRH	Half-Word	16	$0-(2^{16}-1)$	0 - 0xFFFF
STR	Word	32	$0-(2^{32}-1)$	0 - 0xFFFFFFFF

- ARM arithmetic instruction do not affect (update) flags unless we specify it (suffix - S)

Ex:- ADD - flags unchanged
ADDS - flags changed. (C, Z, N or V)

① ADD Rd, Rn, Op2 ; (Rd = Rn + Op2)

Ex:
```
LDR R2, =0xFFFFFFF5
MOV R3, #0x0B
ADDS R1, R2, R3
```

Here, R2 ⇒ 1111 1111 1111 1111 0101
R3 ⇒ (+) 0000 0000 0000 0000 1011
0x10000000 : 1 0000 0000 0000 0000 0000

C = 1, ~~Z~~ Z = 1.

② ADC Rd, Rn, Op2 ; (Rd = Rn + Op2 + C)

(flag bit) from previous instruction.

__/__/____

Exa: LDR R0, = 0xF62562FA
LDR R1, = 0xF412963B
MOV R2, #0x35
MOV R3, #0x21
ADDS R5, R1, R0 (now C=1)
ADC R6, R2, R3 (0x35 + 21 + 1)

③ SUB Rd, Rn, Op2 ; Rd = Rn - Op2

④ SBC Rd, Rn, Op2 ; Rd = Rn - Op2 - 1 + C (multiword subtraction)

⑤ RSB Rd, Rn, Op2 ; Rd = Op2 - Rn (reverse subtract)

Exa: MOV R1, #0x6E
RSB R0, R1, #0 ; R0 = 0 - R1 = 0xFFFFFF92

⑥ RSC Rd, Rn, Op2 ; Rd = Op2 - Rn - 1 + C

Exa: To create 2's complement of 64-bit data in R0 and R1 registers where R0 holds lower 32-bit.
LDR R0, = 0xF62562FA
LDR R1, = 0xF812963B
RSB R5, R0, #0 (now C=0)
RSC R6, R1, #0 (now R5 holds lower 32-bits)

⑦ MUL Rd, Rn, Op2 ; Rd = Rn x Op2 (can fit max 32-bits)

⑧ UMULL RdLo, RdHi, Rn, Op2 ; Rd = Rn x Op2
(RdLo holds lower word
RdHi holds higher word)

Exa: MOV R0, #1
RSB R1, R0, #0 } 0xFFFFFFFF
RSB R2, R0, #0
UMULL R3, R4, R1, R2 => 0xFFFFFFFE, 0x00000001
R4 R5

(Unsigned multiply long)

⑨ MLA Rd, Rm, Rs, Rn , (Rd = Rm × Rs + Rn)
(Multiply and accumulate)

(or) UMLAL RdLo, RdHi, Rn, Op2

Note: We can use SUB instruction to perform division.

Logic Instructions

① AND Rd, Rn, Op2 (bitwise)

X	Y	X & Y (x)
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X / Y (+)
0	0	0
0	1	1
1	0	1
1	1	1

② ORR Rd, Rn, Op2

③ EOR Rd, Rn, Op2 (XOR) (Same - 0, Diff - 1)

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

X	Y	X AND $\bar{Y}$
0	0	0
0	1	0
1	0	1
1	1	0

④ BIC Rd, Rn, Op2

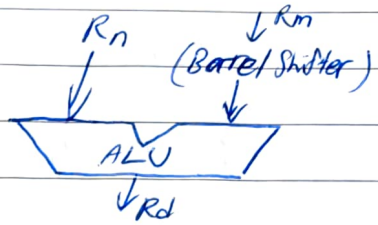
(Bit Clear)
(X $\bar{Y}$.) (X - Y)

__/__/____

⑤ MVN Rd, Rn (Move negative) (1's complement)

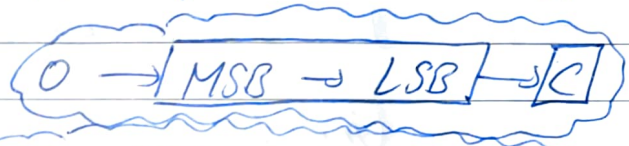
★ Shift & Rotate Instructions

- Barrel Shifter



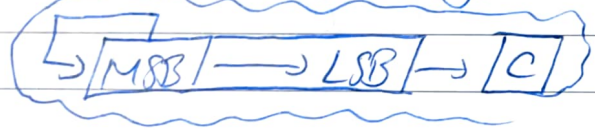
- Used for more arithmetic, logical, comparison and multiply instructions.
- There are two kinds of shifts; logical and arithmetic.

① Logical Shift Right (LSR)



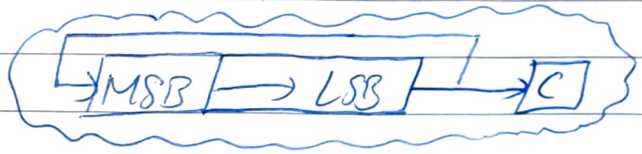
② Logical Shift Left (LSL)

③ Arithmetic Shift Right (ASR)



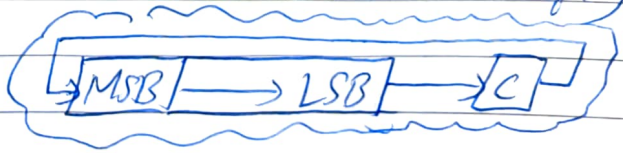
Eg: 0xFFFFFFF6 → ASR #1 → 0xFFFFFFFB
11111011 0 11111011

④ ROR (rotate right)
(Carry = MSB)

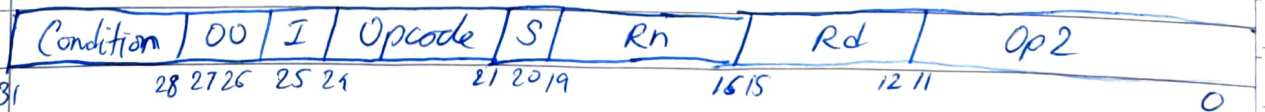


⑤ Rotate Left (ROR (32-n)) (rotate 31 bits to right = rotate 1 bit to left)

⑥ RRX (right rotate + carry)
(Carry → MSB)



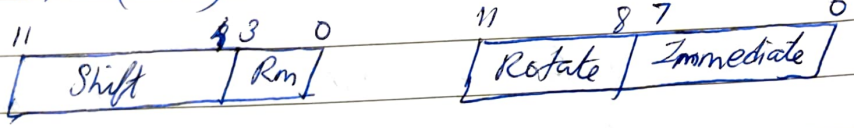
Note (General formation of Process instruction)



- Here, for bit 25, I=0 if Op2 is register.
  I=1 if Op2 is immediate value.

For bit 20, if S=0, flag is unchanged
if S=1, flag updated after execution.

For bit 11-0, (I=0) (I=1)



BCD and ASCII Conversion

- Binary representation of 0 to 9 is called BCD

- Unpacked BCD: Lower 4 bits store BCD digit
  Upper 4 bits are always 0.

Ex: 0000 0101 represents 5. (1 byte / digit)

- Packed BCD: Store two BCD digits. Requires only half the memory compared to unpacked BCD. (2x efficient)

Ex: 0101 1001 represents 59

Key	ASCII	Binary (hex)	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

00101001
00001111
00001001 → 0x09 (R1)

__/__/__

- ASCII → unpacked BCD:

Binary of ASCII (hex) is ANDed with 0000 1111 (0x0F)

- ASCII → packed BCD: First ASCII → unpacked BCD
  then combined to make packed BCD

Key	ASCII	Unpacked	Packed
2	32	0000 0010	
7	37	0000 0111	0010 0111 (0x27)

- ~~ASCII~~ Packed BCD → ASCII:

Packed	Unpacked	ASCII
0x29	0x02 & 0x09	0x32 & 0x39
0010 1001	0000 0010 & 0000 1001	011 0010 & 011 1001

→ (Code for ASCII → packed BCD)

```
MOV  R1, #0x37
MOV  R2, #0x32
AND  R1, R1, 0x0F      (→ unpacked)
AND  R2, R2, 0x0F
MOV  R3, R2, LSL #4    (R2 << 4, R3 = 0x20)
ORR  R4, R3, R1        (R4 = 0x27)
```

→ (Code for ~~Packed~~ BCD → ASCII)

```
MOV  R0, #0x29
AND  R1, R0, #0x0F     (R1 = 0x09)
ORR  R1, R1, #0x30     (R1 = 0x39)
MOV  R2, R0, LSR #4    (R2 = 0x02)
ORR  R2, R2, #0x30     (R2 = 0x32)
```

✱ Branching & Looping

- Loop is repetition of sequence of instructions/operation a certain no. of times.

Condition Code	Condition	Meaning	Condition Flag State
0000	BEQ	equal	Z=1
0001	BNE	not equal	Z=0
0010	BCS/BHS	carry set/higher/same	C=1
0011	BMI	minus (−)	N=1
0100	BPL	plus (+)	N=0
0101	BVS	Overflow	V=1
0110	BVC	no overflow	V=0
0111	BCC/BLO	carry clear/lower	C=0
1001	BLS	less/same	Z=1 / C=0
	BHI	higher	Z=0 / C=1

Eg:- Write a program to (a) clear R0 (b) add 9 to R0 x1000 time (c) place sum in R4.

→
```
AREA CODE1, CODE, READONLY
ENTRY
    LDR R2, =1000   (counter)
    MOV R0, #0      (sum)
AGAIN
    ADD R0, R0, #9
    SUBS R2, R2, #1
    BNE AGAIN       (if Z=0, loop again)
    MOV R4, R0      (else) (if R2=0)
HERE B HERE
    END
```

__/__/____

Ex: Write a program to (a) load R0 register with value 0x55 (b) complement it 16 billion times.

```
AREA EXAMPLE2 , CODE, READONLY
ENTRY
    MOV R0, #0x55
    MOV R2, #16          (outer loop count)
L1  LDR R1, =1000000000  (inner loop count)
L2  EOR R0, R0, #0xFF    (R0 ← R0 ⊕ 0xFF)
    SUBS R1, R1, #1      (inner loop)
    BNE L2               (loop until R1=0, inner)
    SUBS R2, R2, #1
    BNE L1               (loop until R2=0, outer)
HERE B HERE
    END
```

Ex: Write a program to find largest of two numbers.

```
    AREA PROGRAM , CODE, READONLY
    ENTRY
Main
    LDR R1, Value1
    LDR R2, Value2
    CMP R1, R2           (compare)
    BHI Done             (R1 > R2)
    MOV R1, R2           (else, R2 → R1)
Done
    STR R1, Result
Value1 DCD 0x55555555
Value2 DCD 0x55555555
Result DCD 0
    END
```

__/__/__

Note For CMP R1, R2

Instruction	C	Z
R1 > R2	1	0
R1 = R2	1	1
R1 < R2	0	0

Exa For if-else statement: if (a==1) : b=3;
else : b=4;

→
```
MOV R1, a
MOV R2, b
CMP R1, #1      (compare a and 1)
BNE else        (go to else if a≠1)
MOV R2, #3      (b=3)
B endif
else  MOV R2, #4    (b=4)
endif B endif
```

Exa (Division of unsigned numbers in ARM)

```
R0 = numerator;
R1 = denominator;
while (R0 >= R1){
    R0 = R0-R1;        (num = num-deno)
    R2 = R2+1;         (quotient)
}
```

→
```
    AREA mycode, CODE, READONLY
    ENTRY
Main
    LDR R0, =12        (numerator)
    MOV R1, #2         (denominator)
    MOV R2, #0         (quotient)
L1  CMP R0, R1
    BLO FINISH         (if R0<R1, go to FINISH)
```

```
        SUB R0, R0, R1      (R0 = R0 - R1)
        ADD R2, R2, #1      (quotient ++)
B   L1                      (go to L1)
FINISH B FINISH
```

- Unconditional Branch (B) is a jump in which control is transferred unconditionally to target location.

— Recursion

```
int factorial (int n) {
    if (n==0) return 1;
    else
        return n * factorial (n-1);
}
```

→

```
        AREA factorial, CODE, READONLY
        ENTRY
Main
        MOV R0, #1          (counter)
        MOV R1, #6          (to find 6!)
        MOV R3, #1          (factorial value/result)
        BL loop             (Branch with link)
        B STOP
loop
        MUL R4, R3, R0
        MOV R3, R4
        ADD R0, R0, #1      (increment counter)
        CMP R0, R1
        BLE loop            (Branch if less than or equal)
        MOV PC, LR          (return from subroutine)(go to main)
STOP B STOP
        END
```

	(counter) R0	(result) R3	R4
(i)	1	1	1
(ii)	2	2	2
(iii)	3	6	6
(iv)	4	24	24
(v)	5	120	120
(vi)	6	720	720

Ex:-

B label : unconditional branch to label

B<condition> label : conditional branch to label

BL label : calls subroutine and saves return address in Link Register (LR or R14) before jumping.

MOV PC, LR : R15 = R14, returns to instruction after BL

BX Rm : branches to address stored in Rm. Used for function returns / switching between ARM / Thumb mode.

BLX Rm : Jumps to address in R0 and stores return address in LR/R14. Switches modes as well.

I/O PROGRAMMING

- In ARM microcontrollers like LPC1768, I/O devices (LED's, buttons, sensors, etc.) are accessed via memory-mapped I/O. Each device is assigned a specific memory address.

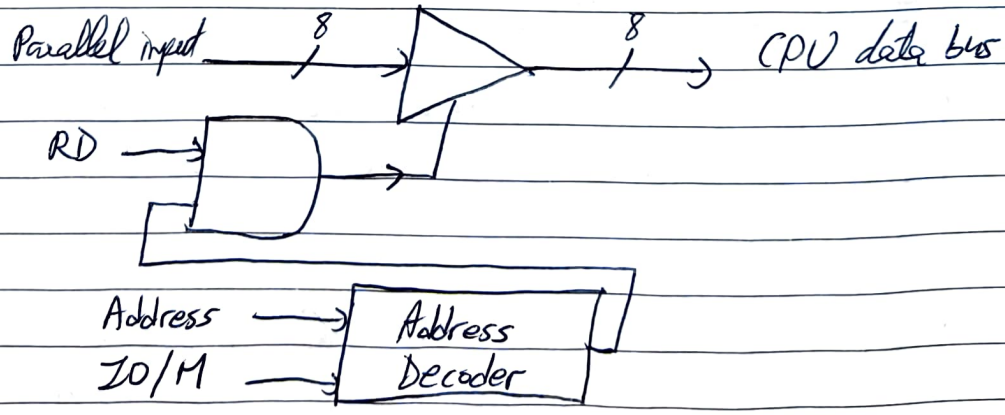
★ Parallel Interface

- The simple I/O port on microcontroller is parallel port, a simple mechanism that allows software to interact with external devices. (parallel = multiple devices at once)

— Input Port

- Microcontroller reads a value from memory-mapped input port from devices (Ex:- To check if switch is ON)

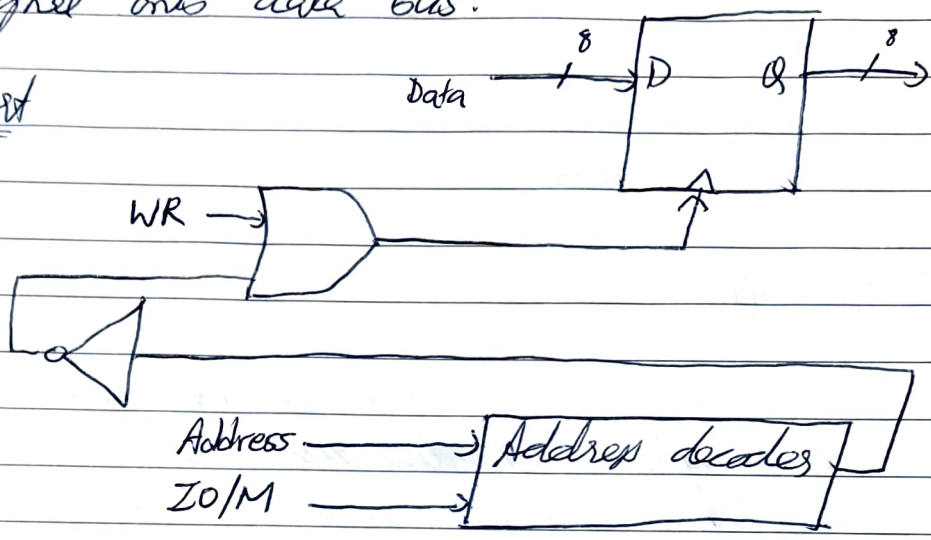
- Actual physical signal on device is passed through tri-state buffer onto data bus only during read.



- When CPU wants to read, it activates a signal called RD (read strobe) and CPU puts address of input port on address bus.

- If address matches ~~and~~ the input port address, and RD is active, then tri-state buffer is enabled and sends input signal onto data bus.

— Output Port



- Implemented with registers, multi-bit flip-flops and a common clock. Register takes data input D and loads ~~output~~ data to register output (Q) upon rising edge of clock input driven by CPU write strobe (WR), only if ~~address~~ on CPU bus ~~corresponds~~ to address of output port.

☆ I/O Programming

- To control other devices such as sensors, LED, etc. Done through GPIO pin (General purpose Input Output)

- GPIO is a pin on an IC (integrated circuit) which can either be input/output pin controlled at run time.

- LPC1768 has 5 ports P0 - P4, each 32 bit pins. Since few pins are not ~~available~~ available,
  Total = ~100 pins.

Note: LPC1768 is an ARM Cortex-M3-based microcontroller which offers high performance and very low power consumption, incorporates 3-stage pipeline, used Harvard architecture with separate instruction and data buses as well as a third bus for peripherals.

- LPC1768 : 512 KB Flash
  64 KB SRAM
  8 KB ROM
  100 pin IC

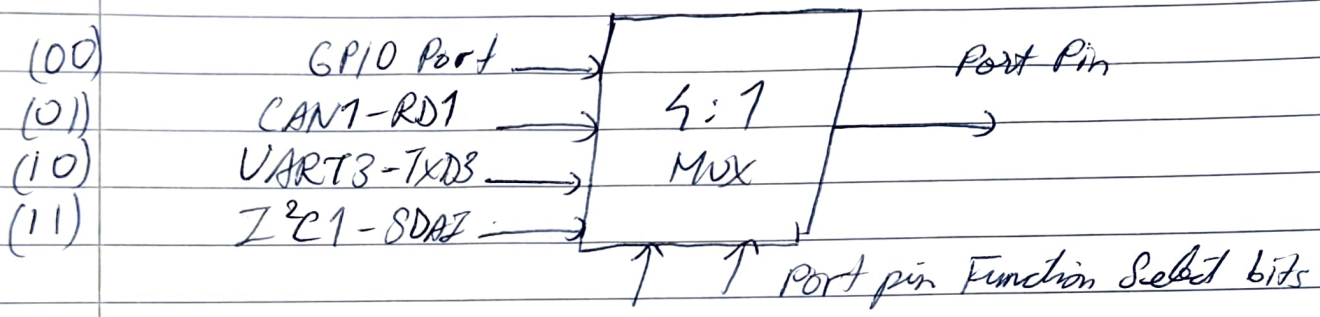
— Pin Connect Block

- Responsible for configuring function of each physical pin on the microcontroller.
- Pins labelled IO indicate that they can be used for either inputs / outputs depending on configuration.
- Horizontal lines indicate power and ground pins for device to operate.
- Nomenclature: P2.11 refers to pin 11 in port 2.

(Registers) PINSEL0 - P0[15:0] (control)
PINSEL1 - P0[31:16]
PINSEL2 - P1[15:0] (ethernet)
PINSEL3 - P1[31:16]
PINSEL4 - P2[15:0]
:
PINSEL9 - P4[31:16]
PINSEL10 - Trace port enable on P2.2 - P2.6

★★ • These are pin function select registers, which configure the microcontroller pins to desired functions

• Each port pin can be used for upto 4 functions with help of MUX (4:1)



• Used to configure pin for GPIO mode before setting direction or writing values

Note We cannot directly set values for PINSELx registers we must use AND or OR. (overwrite existing bits)

Ex: To set ~~P0.0~~ P0.0 to TXD3
If PINSEL0 = 0x00000005 (..... 000101)
PINSEL0 &= 0xFFFFFFFC clears last two bits (..000100)
PINSEL0 |= 0x00000002 sets last 2 bits to 10 (.... 000110)
(Here P0.0 uses bits 1:0 of PINSEL0)

– FIODIR (Fast GPIO direction control register)

• After pin is set to GPIO mode via PINSELx, the FIOxDIR register determines whether pin is configured as input (0) or output (1)

Ex: Bit 31 on FIOxDIR controls pin Px.31

__/__/____

— FIOMASK (fast mask register) (pin access controller)

- Works like a filter, tells the microcontroller which pins you want to ignore when reading / writing to a port.
  (1 - masked / ignored / off, 0 - active / can be used)
- FIOPIN, FIOSET and FIOCLR can only work on unmasked pins with bit 0 (active)

— FIOPIN (fast port pin value register)

- Once pin is designated as GPIO and direction is set, FIOxPIN either returns the state of pin if configured as input, or changes output level / write if configured as output.

Ex: Toggling an LED or checking logic state of input pin.

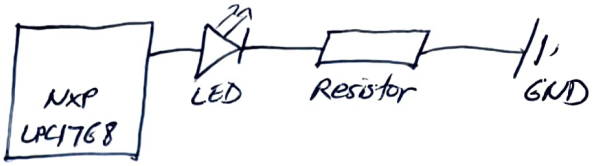
Note: FIOPIN only works on pins with bit values marked with 0 in FIOMASK register, else it will read as 0 regardless of physical pin state.

— FIOSET (fast port output set register)

- 0 → controlled pin is unchanged
  1 → controlled pin output set to HIGH (1) (only if FIODIR set it to output (1))

— FIOCLR (fast port output clear register)

- 0 → controlled pin output is unchanged
  1 → controlled pin output set to LOW (0)



✡ LED

- Initialize microcontrollers system, power up peripherals, set clock rate, etc. Connect necessary pins using pin connect block; to configure pins to desired function
- Set / clear bit of respective pin to turn ON/OFF LED.

→
```
#include <lpc17xx.h>
void
int main() {
    SystemInit();
    LPC_PINCON->PINSEL4 = 0x00000000;
    LPC_GPIO2->FIODIR = 0xFFFFFFFF;
    while(1) {
        LPC_GPIO2->FIOSET = 0xFFFFFFFF;
        for(i=0; i<5000; i++);
        LPC_GPIO2->FIOCLR = 0xFFFFFFFF;
        for(i=0; i<5000; i++);
    }
}
```

✡ Seven-Segment

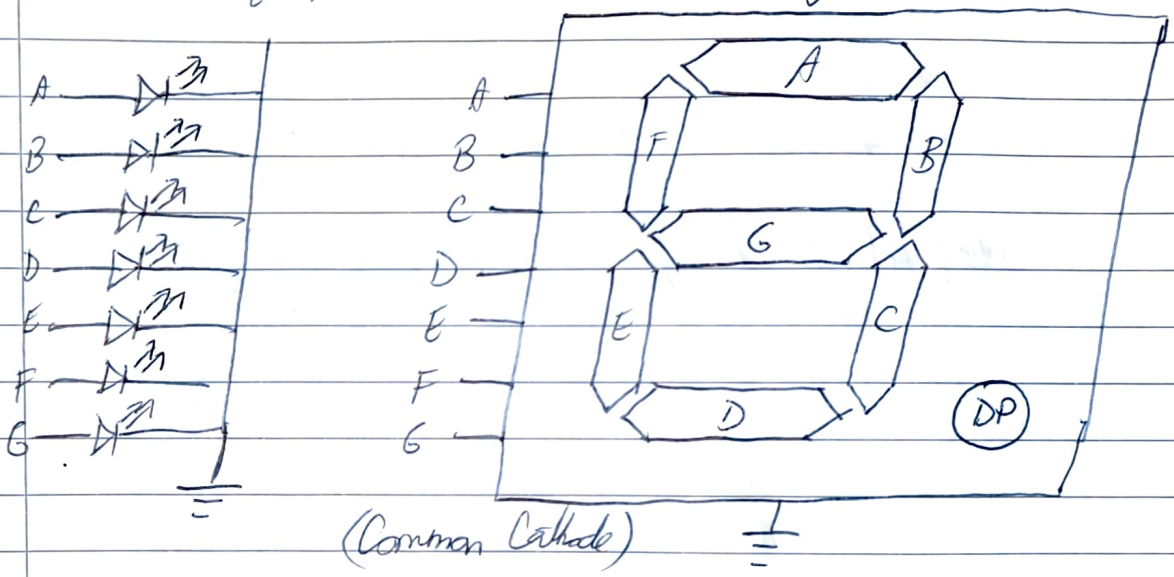
- In seven-segment display, power/voltage at different pins can be applied at the same time, so we can form combinations of displays numericals from 0 → 9 and alphabets A → F.

- Types of Displays

(a) Common Cathode : All cathode connections of LED

segments are connected to logic 0 or ground. We use logic 1 through current limiting resistors to forward bias individual anode terminals a to g.

- Common Anode: Anode connections connected to logic 1. We use logic 0 through current limiting resistors to cathode of particular segment a to g.



Value	DP	G	F	E	D	C	B	A	HEX
0	0	0	1	1	1	1	1	1	0x3F
1	0	0	0	0	0	1	1	0	0x06
2	0	1	0	1	1	0	1	1	0x5B
3	0	1	0	0	1	1	1	1	0x4F
4	0	1	1	0	0	1	1	0	0x66
5	0	1	1	0	1	1	0	1	0x6D
6	0	1	1	1	1	1	0	1	0x7D
7	0	0	0	0	0	1	1	1	0x07
8	0	1	1	1	1	1	1	1	0x7F
9	0	1	1	0	1	1	1	1	0x6F
A	0	1	1	1	0	1	1	1	0x77
B	0	1	1	1	1	1	0	0	0x7C

Value	DP	G	F	E	D	C	B	A	HEX
C	0	0	1	1	1	0	0	1	0x39
D	0	1	0	1	1	1	1	0	0x5E
E	0	1	1	1	1	0	0	1	0x79
F	0	1	1	1	0	0	0	1	0x71

(and dot)

→ Here a → g is from P0.4 → P0.11

→
```
#include <LPC17xx.h>
unsigned int delay, count = 0, Switchcount = 0, j;

unsigned int Disp[16] = { 0x00003F0, 0x00000060,
    0x000005B0, 0x000004F0, 0x00000660
    0x000006D0, 0x000007D0, 0x00000070,
    0x000007F0, 0x000006F0, 0x00000770,
    0x000007C0, 0x00000390, 0x000005E0,
    0x00000790, 0x00000710 };
#define ALLDISP = 0x00180000.
#define DATAPORT = 0x00000FF0   (Pin 4 → 11)

int main (void) {
    LPC_PINCON → PINSEL0 = 0x00000000;
    LPC_PINCON → PINSEL1 = 0x00000000;
    LPC_GPIO0 → FIODIR = 0x00180FF0;
    while (1) {
        LPC_GPIO0 → FIOSET |= ALLDISP
        LPC_GPIO0 → FIOCLR = 0x00000FF0.
        LPC_GPIO0 → FIOSET = Disp[Switchcount];
        for (j=0; j<3; j++) {
            for (delay=0; delay<30000; delay++);
        }
        Switchcount ++;
```

16 in decimal

```
if (Switchcount == 0x10){
    Switchcount = 0;
    LPC_GPIO0->FIOCLR = 0x00180FF0;
}
}
}
```

✡ Steppes Motor

- Motors that move in discrete steps or convert electrical pulses into rotatory motion.
- They have multiple coils organized in groups called phases and by energizing each of these phases in sequence, motor will rotate one step ~~at a~~ a time.

→
```
#include <LPC17xx.H>
void clockwise (void);
void anticlockwise (void);
unsigned long int var1, var2;
unsigned int i=0, j=0, k=0;

int main (void){
    LPC_PINCON->PINSEL4 = 0x00000000;
    LPC_GPIO2->FIODIR = 0x0000000F;
    while (1){
        for(j=0; j<30; j++)
            clockwise ();
        for (k=0; k<50000; k++);
        for (j=0; j<30; j++)
            anticlockwise ();
        for (k=0; k<50000; k++);
    }
}
```

```
void clockwise(void){
    var1 = 0x00000001;
    for(i=0; i<=3; i++){
        LPC_GPIO2->FIOCLR = 0x0000000F;
        LPC_GPIO2->FIOSET = var1;
        var1 <<= 1;
        for(k=0; k<=15000; k++);
    }
}
void anticlockwise(void){
    var1 = 0x00000008;
    for(i=0; i<=3; i++){
        LPC_GPIO2->FIOCLR = 0x0000000F;
        LPC_GPIO2->FIOSET = var1;
        var1 >>= 1;
        for(k=0; k<=15000; k++);
    }
}
```

(Refer to Relay/Buzzer and Keypad Code on Teams)

TIMERS / COUNTERS

- Timers are used to measure elapsed time like processor ticks or count external events.
- They generate periodic interrupts in timer mode and count events in counter mode and generate precise delays.
- Timers essentially are software designed to count time interval between events, counts the cycle of peripheral clock or externally supplied clock.
- There are 4 identical timer peripherals in LPC1768 (Timer 0 - Timer 3), each having their own Time Counter (TC) and Prescale Register (PR).
- Timer Counter (32-bit) is incremented every (PR+1) cycles of peripheral clock (PCLK), controlled through TCR.
- Pre-scaler Register helps define resolution of timer, specifies the Pre-scaler value for incrementing TC.
- When timer is reset, TC is set to 0; every (PR+1) clock cycles of PCLK, TC is incremented by 1. This is done with the help of Pre-scaler Counter (PC) which is incremented each PCLK cycle from 0 till value equals PR, then PC = 0 (reset) and TC ++.
- When TC reaches it's maximum value, it is reset back to 0.

- Every timer has 4 32-bit Match Registers (MR0-MR3) which is a register which contains a specific value set by the user.

- When the timer starts, every time TC is incremented, value is compared with MRx, and if they match actions can be triggered based on settings in MCR (Match Control Register) register. (internal)

- There are also match outputs, Timer 2 has 4, rest have 2.

- Also, when TC == MRx, timers also have a physical output pin on the MCU which can be controlled by External Match Register (EMR) which can be toggled, set or cleared. It reflects the External Match status on the pin.

- Here MATx.y is the physical output pin tied to Timer x and Match Register (MR) y.

- As a result, EMR controls functionality of output pin when match register (MR) matches TC.

→ • TCR (Bit [1:0], rest are reserved)

Bit 0 (Counter Enable): 1 → TC, PR enabled for counting
0 → TC, PR disabled

Bit 1 (Counter Reset): 1 → TC, PR reset on next positive edge of PCLK

__/__/__

→ MCR (operation when MRx == TC)

- Here Bits [2:0] used for MR0
  Bits [5-3] used for MR1.....

For MRx

Bit 0 (MRxI): 1 → interrupt generated
Bit 1 (MRxR): 1 → TC is reset
Bit 2 (MRxS): 1 → TCR[0] → 0
(disable / stop timer)

→ EMR (contains status bits EMx and control bits EMCx)

Bits [3:0] - EM0-EM3 (current output state for MATx.0 - MATx.3)
Bits [5:4] - EMC0 (what to do to EM0 on TC == MR0)
Bits [7:6] - EMC1 (" " EM1 on TC == MR1)
Bits [9:8] - EMC2 (" " EM2 on TC == MR2)
Bits [11:10] - EMC3 (" " EM3 on TC == MR3)

EMCx	Function	
00	Do nothing	
01	Clear	(EMx = 0)
10	Set	(EMx = 1)
11	Toggle	(Flip / 1's complement)

- Timers also have a Capture pin (CAPx.y) which is a physical pin that detects signal edges (external). Capture registers store the timer value when edge is detected. (2)

- CTCR (Count Control Register) is used to select either Timer Mode / Counter Mode (+ edge for counting)

  Bits [1:0] : 00 → Timer Mode
  01 → Counter Mode with rising edge
  10 → Counter Mode with falling edge
  11 → Counter Mode with both edges.

  Bits [3:2] : Counter Input Select.
  00 → CAPx.0 pin for Timer x
  01 → CAPx.1 pin for Timer x.

Eg: While measuring duration of high pulse / pulse width:
1. Set timer running.
2. Configure CAP0.0 to capture on rising edge, save to CR0
3. Configure CAP0.0 to capture on falling edge, save to CR1
5. Pulse Width = CR1 - CR0 (end time - start time)

- For calculating Timer Pre-Scalar, we need to select peripheral clock for the timer.

Timer	PCLKSELx
Timer 0	PCLKSEL0 [3:2]
Timer 1	PCLKSEL0 [5:4]
Timer 2	PCLKSEL1 [13:12]
Timer 3	PCLKSEL1 [15:14]

- To get PCLK value,

PCLKSELx bits	PCLK
00	PCLK = CCLK/4
01	PCLK = CCLK
10	PCLK = CCLK/2
11	PCLK = CCLK/8

CCLK: System Core Clock

- Delay / time for 1 clock cycle, when PCLK = X MHz

$$T_{PCLK} = \frac{1}{PCLK_{Hz}} = \frac{1}{(X * 10^6)\ seconds}$$

- For given value of PR

$$T_{RES} = \frac{PR+1}{PCLK_{Hz}} = \frac{PR+1}{(X * 10^6)} \ seconds$$

(timer resolution)

$$PR = ((X * 10^6) * T_{RES}) - 1$$

— Program to get 1ms delay

```
void initTimer0 (void) {
    LPC_TIM0 -> CTCR = 0x0;      (timer mode)
    LPC_TIM0 -> TCR = 0x02;      (reset timer)
    LPC_TIM0 -> PR = 2;          (increment TC every 3 clock cycles)
    LPC_TIM0 -> MR0 = 999        (TC counts 0 -> 999)
    LPC_TIM0 -> MCR = 2          (reset TC after 1ms delay)
    LPC_TIM0 -> EMR = 0x20       (when match occurs, bit 0
                                  of EMR will be set)
    LPC_TIM0 -> TCR = 0x01       (enable timer 0)
}
void delay (void){
    initTimer0 ();
    while (!(LPC_TIM0 -> EMR & 1));
}
```

PWM

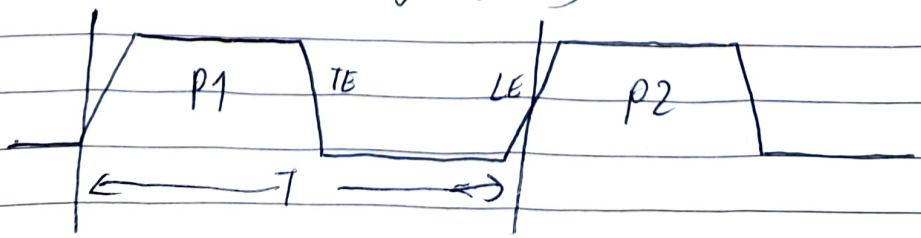
- Pulse Width Modulation (PWM) is an easy way to encode data such that it corresponds to width of pulse given fixed frequency.
- Based on standard Timer Block, just that PWM function is added to these features and is based on match register events.
- PWM signals contain two types of edges: Leading Edge (/) and Trailing Edge (\)

$$\text{Duty Cycle} = \frac{T_{ON}}{T_{ON} + T_{OFF}} \qquad (DC\% = DC \times 100)$$

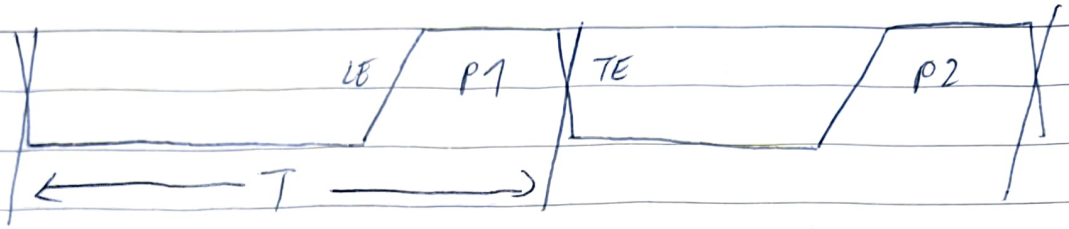
- PWM signal is of two types:

(a) Single Edge PWM: Pulse is at beginning / end of period.

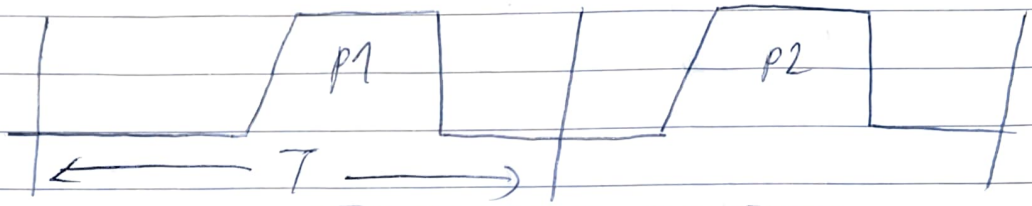
In Trailing Edge PWM, leading edge is fixed at beginning of period and trailing edge is modulated (controlled, can change position to control pulse width. Thereby changing duty cycle)



In Leading Edge PWM, trailing edge is fixed at end of period and leading edge is modulated.



(b) Double Edge PWM: Pulse can be positioned anywhere within period. Both edges are modulated.



- Average Voltage, $V_{avg} = (DC \times V_H) + ((1 - DC) \times V_L)$

where DC → duty cycle, V_H → voltage at high state,
V_L → voltage at low state.

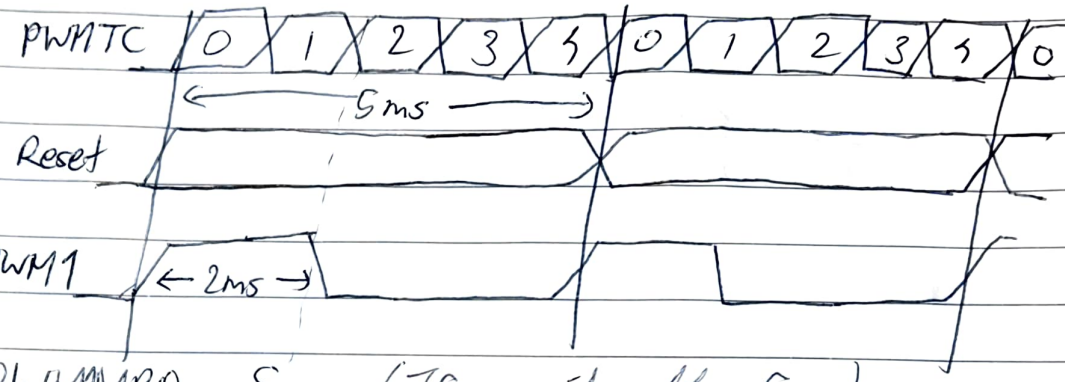
- ARM cortex LPC1768 has one PWM module (PWM1), supports both single and double edge PWM.

- Similar to timer block, it has TC and PR and match registers. PWM1.0 - PWM1.6
where PWM1.1 means PWM1MR1

- PWM1.0 is used to generate PWM period.
Remaining MR can either have 6 single edge PWM or 3 double edge PWM.

→ • At the start of every cycle, PWM signal goes HIGH (unless match value is 0) and it goes LOW when match value is reached (TC == MRx)

Ex: Let PWM duration be 5ms, TC increments by 1 every 1 ms. using appropriate PR value.



PWMMMR0 = 5 (TC reset after 5ms)
PWMMMR1 = 2 (PWM1 will go LOW when TC=2)

At t= 0ms, PWM1 goes HIGH
t = 2ms PWM1 goes LOW (MR1 = 2)
Duty cycle = $\frac{2}{5} \times 100 = 40\%$

→ • PWM1TCR : Bit 0 : 1 → TC, PC enabled
0 → both disabled
Bit 1 : 1 → TC, PC reset at next edge of PCLK
Bit 3 : enables PWM mode / output

• PWM1PR to control resolution of PWM output
TC increments every (PWM1PR + 1) PCLK cycles.

→ • PWM1MCR : Bit 0 : generate interrupt
Bit 1 : Reset TC
(for PWM1MR0) Bit 2 : Stop TC, PC, disables counters

→ PWM1PCR : Bit [2-6], select between single edge (0) and double edge mode (1)
Bit [9-14], enable / disable PWM outputs

→ PWM1IR (Interrupt register): Tells which PWM1 match register caused an interrupt. If set to 1, it means that interrupt was triggered since TC matched that register. Can be cleared by writing 1 to that bit.

Bit 0 – PWM1MR0
Bit 1 – PWM1MR1
⋮

→ PWM1LER (Load Enable Register): Updates value on match registers while PWM is running, when TC resets or corresponding bit on PWM1LER set to 1. Until then, new value is stored in Shadow Register.

☆ Configuration of PWM

① Select PWM Pin Function (PINSEL*)

LPC_PINCON → PINSEL4 |= (1<<0); (Configure P2.0 as PWM1.1)

② Select Single / Double Edge Mode (PCR)

LPC_PWM1 → PCR = (1<<9); (by default, single edge, also enable PWM output)

③ Set Prescaler (PR)

LPC_PWM1 → PR = 0; (TC incremented every PCLK)

④ Set Period (MR0) (total duration of 1 PWM cycle)

LPC_PWM1 → MR0 = 10000;

⑤ Set Pulse Width (MR*) (until when to stay HIGH)

LPC_PWM1 → MR1 = 4000;

⑥ Configure Match Control Register (MCR)

```
LPC_PWM1->MCR = (1<<1);  (reset TC on match)
LPC_PWM1->MCR |= (1<<0);  (interrupt on match)
```

⑦ Set Load Enable Registers (LER)

```
LPC_PWM1->LER = (1<<0) | (1<<1);  (load MR0, MR1)
```

⑧ Enable PWM Output (PCR)

```
LPC_PWM1->PCR |= (1<<9);
```

⑨ Reset PWM Timer (TCR)

```
LPC_PWM1->TCR = (1<<1);
```

⑩ Enable Timer and PWM Mode (TCR)

```
LPC_PWM1->TCR = (1<<0) | (1<<3);
```

—x—