classmate
Date 05/01/2021
Page 1

L-T-P-C : 3-0-3-4

COURSE OBJECTIVES :

1. To introduce the advanced features of an advanced RISC ~~microcontroller~~ Microprocessor.
2. To apply the knowledge of embedded C Programming for configuring various peripherals of a microcontroller.
3. To Design and Develop Microcontroller based solution for solving real world problems.

COURSE OUTCOMES :

CO1 : Able to identify the importance of 32 bit Microprocessor.
CO2 : Able to understand architecture of the ARM processor.
CO3 : Able to analyze Peripherals and their programming aspects.
CO4 : Able to design and develop embedded systems using microcontroller.

Q. What is an Embedded system?
A. System in which software (firmware) is embedded into the hardware. The core part of the system will be a programmable device (microprocessor / microcontroller / etc.)
Examples -

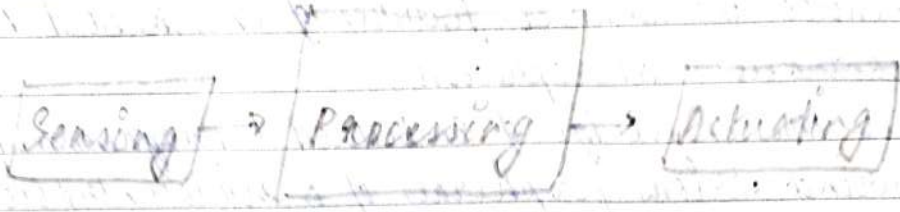
1. Avionics
2. Communication
3. Automobile
4. Consumer Electronics
5. Office Equipments
6. Household Appliances

AUTOMOTIVE EMBEDDED SYSTEMS :

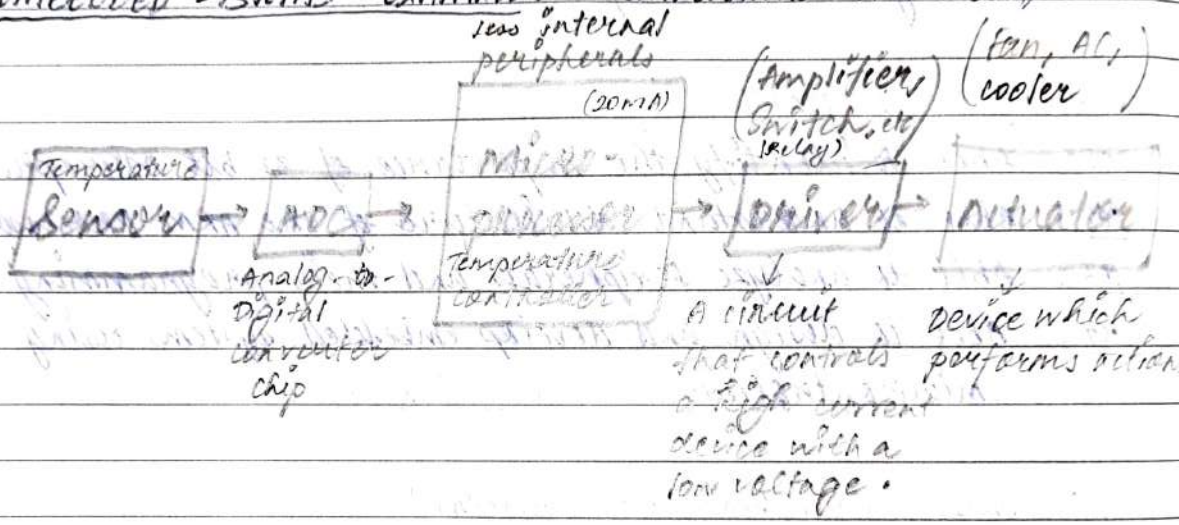
Today's high-end automobile have >80 microprocessors :

1. 4-bit microcontroller checks seat belt;
2. microcontrollers run dashboard devices;
3. 16/32-bit microprocessor controls engine.
4. Millions lines of code

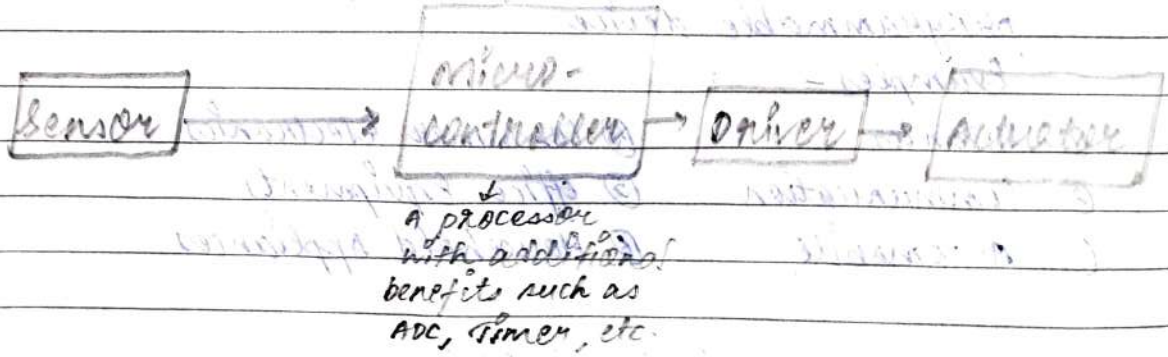
* OPERATIONS - EMBEDDED SYSTEMS :



* EMBEDDED SYSTEMS - EXAMPLE : (TEMPERATURE CONTROLLER)



Replacing micro-processor with a micro-controller,



* MICROPROCESSOR :

① Requires 'external' support hardware
Example - External RAM, ROM, Peripherals

② Application : Processing - Arithmetic, Logic operations.

* MICROCONTROLLER :

① Very little external support hardware / stand alone.
② Most RAM, ROM and peripherals on chip.
③ "Computer on a chip", or "System on chip" (SOC)
Example - PSC (Peripheral Interface Controller)
④ Application : Controlling purposes.

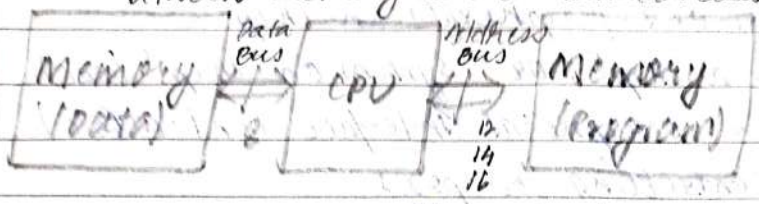
Q. Why Embedded Systems ?
A ① Reduced number of components.
② Reduced size.
③ Reduced cost.
④ Reduced power consumption.
⑤ Easier upgradation.
⑥ Easier troubleshooting & maintenance.
⑦ Best suited for specific controlling applications.

* NOTE : The CPU of a microcontroller contains Oscillator (0-40 MHz), A/D Converter, Microprocessor, RAM, and Program Memory.
[Refer to slide for diagrammatic representation.]

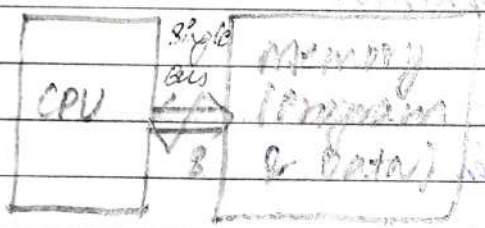
* VARIOUS MICROCONTROLLERS :
① 8 bit microcontrollers
(i) Microchip - PIC 12 & 16 series.
(ii) Atmel - 89c51
(iii) Intel - 8051
(iv) Motorola - 68HCxx series
② 16 bit microcontrollers
(i) Microchip - PIC 18 series.
③ 32 bit microcontrollers
(i) ARM Processors
④ DSP based microcontrollers
(i) Shark

TWO DIFFERENT ARCHITECTURES :

BUS : Path through which interaction takes place between memory and CPU simultaneously [collection of signals]



Harvard Architecture



Von-Neumann Architecture

	VON-NEUMANN ARCHITECTURE	HARVARD ARCHITECTURE
(1)	It is ancient computer architecture based on stored program computer concept.	It is modern computer architecture based on Harvard Mark I relay based model.
(2)	Same physical memory address is used for instructions and data.	Seperate physical memory address is used for instructions and data.
(3)	There is common bus for data and instruction transfer.	Seperate buses are used for transferring data and instruction.
(4)	Two clock cycles are required to execute single instruction	An instruction is executed in a single cycle
(5)	It is cheaper in cost.	It is costly than Von Neumann Architecture.
(6)	CPU cannot access instructions and read/write at the same time.	CPU can access instructions and read/write at the same time.

⑦	It is used in personal computers and small computers.	It is used in micro controllers and signal processing.

* RISC vs. CISC PROCESSOR :

	COMPLEX INSTRUCTION SET COMPUTER (CISC)	REDUCED INSTRUCTION SET COMPUTER (RISC)
①	A large number of instructions are present in the architecture. (usually >100)	Very fewer instructions are present. The number of instructions are generally less than 100 (usually <50)
②	Some instructions with long execution times. These include instructions that copy an entire block from one part of memory to another and others that copy multiple registers to and from memory.	No instruction with a long execution time due to very simple instruction set. Some early RISC machines did not even have an integer multiply instruction, requiring compilers to implement multiplication as a sequence of additions.
③	Variable-length encodings of the instructions. Example - IA32 instruction size can range from 1 to 15 bytes.	Fixed-length encodings of the instructions are used. Example - In IA32, generally all instructions are encoded as 4 bytes.
④	Multiple formats are supported for specifying operands. A memory operand specifier can have many different combinations of displacement, base and index registers.	Simple addressing formats are supported. Only base and displacement addressing is allowed.
⑤	CISC supports array. Several addressing modes	RISC does not supports array. Only a few addressing modes.

(6)	Arithmetic and logical operations can be applied to both memory and register operands.	Arithmetic and logical operations only use register operands. Memory referencing is only allowed by load and store instructions, i.e. reading from memory into a register and writing from a register to memory respectively.
(7)	Implementation programs are hidden from machine level programs. The ISA provides a clear abstraction between programs and how they get executed.	Implementation programs exposed to machine level programs. Few RISC machines do not allow specific instruction sequences.
(8)	Condition codes are used. The stack is being used for procedure arguments and return addresses.	No condition codes are used. Registers are being used for procedure arguments and return addresses. Memory references can be avoided by some procedures.
(9)	Usually takes more than 1 internal clock cycle (Tcyc) to execute.	Executes 1 instruction in 1 internal clock cycle (Tcyc)
(10)	Used in : 80x86, 8051, 68HC11, etc.	Used in : SPARC, ALPHA, Atmel AVR, etc.

* DIFFERENCE BETWEEN MICROPROCESSOR AND MICROCONTROLLER :

PTO

MICROPROCESSOR	MICROCONTROLLER
① Microprocessor is the heart of computer system. It is only a processor, so memory and I/O components need to be connected externally.	Microcontroller is the heart of an embedded system. It has a processor along with internal memory and I/O components.
② Memory and I/O has to be connected externally, so the circuit becomes large.	Memory and I/O are already present, and the internal circuit is small.
③ You cannot use it in compact systems. Cost of the entire system is high.	You can use it in compact systems. Cost of the entire system is low.
④ Due to external components, the total power consumption is high. Therefore, it is not ideal for the devices running on stored power like batteries.	As external components are low, total power consumption is less. So it can be used with devices running on stored power like batteries.
⑤ Most of the microprocessors do not have power saving features. It is mainly used in personal computers.	Most of the microcontrollers offer power-saving mode. It is used mainly in a washing machine, MP3 players, and embedded systems.
⑥ It has a smaller number of registers, so more operations are memory-based.	It has a more number of registers, so the programs are easier to read-write.
⑦ It is based on Von-Neumann model. It is a central processing unit on a single silicon-based integrated chip.	It is based on Harvard architecture. It is a byproduct of the development of microprocessors with a CPU along with other peripherals.
⑧ It has no RAM, ROM, I/O units, timers and other peripherals on the chip.	It has a CPU along with RAM, ROM, and other peripherals embedded on a single chip.

(9)	It uses an external bus to interface to RAM, ROM, and other peripherals.	It uses an internal controlling bus.
(10)	Microprocessor-based systems can run at a very high speed because of the technology involved.	Microcontroller-based systems run up to 200 MHz or more depending on the architecture.
(11)	It is used for general purpose applications that allow you to handle loads of data.	It is used for application-specific systems.
(12)	It is complex and expensive, with a large number of instructions to process.	It is simple and inexpensive with less number of instructions to process.

LECTURE 2 - INTRODUCTION TO ARM PROCESSOR

→ Acron RISC Machines

* BRIEF HISTORY OF ARM:

(1) ARM was developed at Acron Computer Limited of Cambridge, England between 1983 and 1985.
(2) RISC concept introduced in 1980 at Stanford & Berkley.
(3) ARM Limited founded in 1990.
(4) ARM Cores -
→ Advanced RISC Machines
→ Intellectual property core (IP)
(i) licensed to partners to develop and fabricate new micro-controllers
(ii) Soft-cores (OS file)

* FEATURES OF BASIC RISC ARCHITECTURE:

(1) A large Uniform Register File → collection of registers - same size (32 bit)
(2) Load-Store Architecture, where data processing operates on register content only.
(3) Uniform and fixed length Instructions 32-bit processor
(4) Instructions are 32-bit long → Assembly language Program
Advanced C programming

⑤ Good Speed / Power Consumption Ratio
⑥ High Code Density - Data in a unit area

* PRINCIPLE FEATURES / ENHANCEMENTS IN ARM :
① Control over ALU and Barrel Shifter for every data processing operations to maximize their usage. (multiple left/right shift in a single cycle)
② Auto-Increment and Auto-Decrement Addressing modes to optimize program loops. (methods to read/write data)
③ Load and Store Multiple Instructions to maximize data throughput. (LDM) (STM) (LDR) (STR)
④ Conditional Execution of Instructions to maximize Execution throughput.

* ARM ARCHITECTURE VERSIONS :
① Version 1 (1983-1985) - 26 bit Addressing, no Multiply or Coprocessor
② Version 2 - Includes 32-bit result multiply coprocessor
③ Version 3 - 32-bit Addressing
④ Version 4 - Add Signed, Unsigned Half-word & Signed Byte load & Store Instructions
③ Version 4T - 16-bit Thumb - compressed form of Instructions
⑥ Version 5T - Superset of 4T adding new Instructions
⑦ Version 5TE - Add Signal Processing Extension

Examples -
ARM6 : v3
ARM 7 : v3
ARM 7TDMI : v4T (LPC 2148)
StrongArm : v4
ARM 9E-S : v5TE

T: Thumb instruction support (16-bit) - Compressed ARM [subset of ARM instructions]
D: On-chip debug support
M: Enhanced multiplier
I: Embedded ICE hardware (In circuit emulator JTAG)
T2: Thumb-2
S: Synthesizable code
E: Enhanced DSP instruction set
J: Java support, Jazelle
Z: Should be Trustzone?
F: Floating point unit
H: Handshake, clockless design for synchronous or asynchronous design

* ARM vs. THUMB COMPARISON:

	ARM	THUMB
①	32-bit instruction set	16-bit instruction set
②	3-data address instructions	2-data address instructions
③	16 general purpose registers	8 general purpose registers
④	more regular binary encoding (Better performance ↑)	Greater code intensity.

NOTE - (Thumb) If used correctly, can lead to better performance / power-efficiency.

* OVERVIEW - CORE DATA PATH:

① Data Items are placed in register file.
② No Data Processing Instructions directly manipulate Data in memory.
③ Instructions Typically use two source registers and single result or Destination Registers
④ A Barrel Shifter on the date path can pre-process data before it enters into ALU.

Indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode → Banked Registers

classmate
Date
Page 11

⑧ Increment/decrement logic can update the Register content for sequential access independent of ALU.

* BASIC ARM ORGANIZATION:

[Refer to slide 9 for diagrammatic representation.]

LECTURE 3 - ARM PROGRAMMER'S MODEL

ARM REGISTER ORGANIZATION

* ARM REGISTER ORGANIZATION: Total - 37 Registers [17 + 20]
  ↳ General purpose registers
  ↳ Banked: Available only in corresponding mode of operation

[Refer to slide 2 for tabular representation.]

Six operating modes

User		Supervisor	Abort	Undefined	Interrupt (normal)	Fast interrupt
R0		R0	R0	R0	R0	R0
R1		R1	R1	R1	R1	R1
R2		R2	R2	R2	R2	R2
R3		R3	R3	R3	R3	R3
R4		R4	R4	R4	R4	R4
R5		R5	R5	R5	R5	R5
R6		R6	R6	R6	R6	R6
R7		R7	R7	R7	R7	R7
R8		R8	R8	R8	R8	R8_fiq
R9		R9	R9	R9	R9	R9_fiq
R10		R10	R10	R10	R10	R10_fiq
R11		R11	R11	R11	R11	R11_fiq
R12		R12	R12	R12	R12	R12_fiq
R13	special functions	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	special functions	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15	special functions	R15	R15	R15	R15	R15
CPSR	Current Program Status register	stores the status of arithmetic/logic instructions, e.g. carry/c bit				
SPSR	Saved Program Status register	SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

↳ modifies data within the corresponding mode of operation.

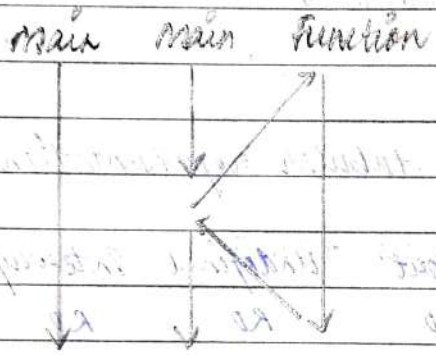
↑ 14 general purpose register

* SPECIAL REGISTERS :

① R13 - Stack Pointer
② R14 - Link Register
③ R15 - Program Counter

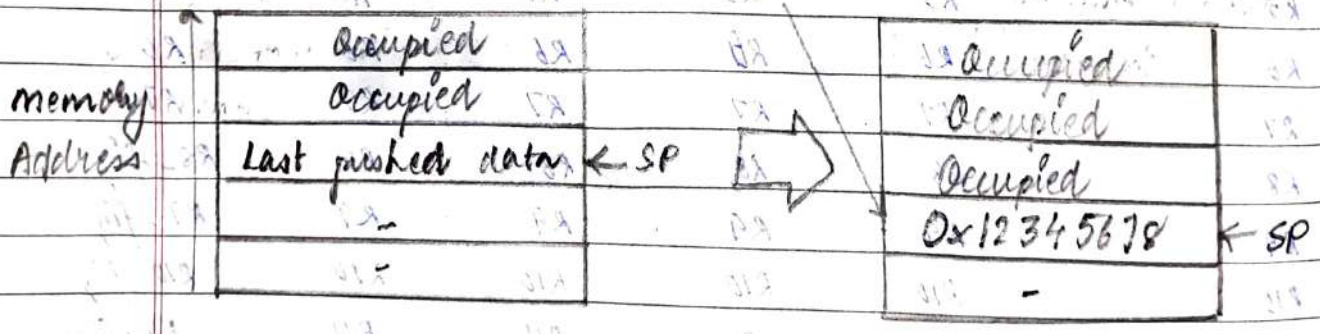
* R13 - STACK POINTER :

Stack Pointer points to Top of stack



R0 | 0x12345678

PUSH {R0}

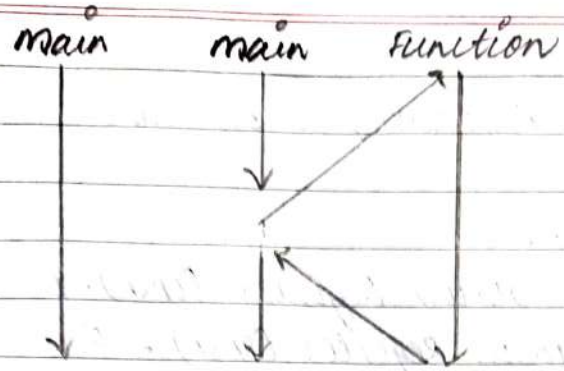


Holds the data where the next pointer has to be pushed on to.

* R14 - LINK REGISTER :

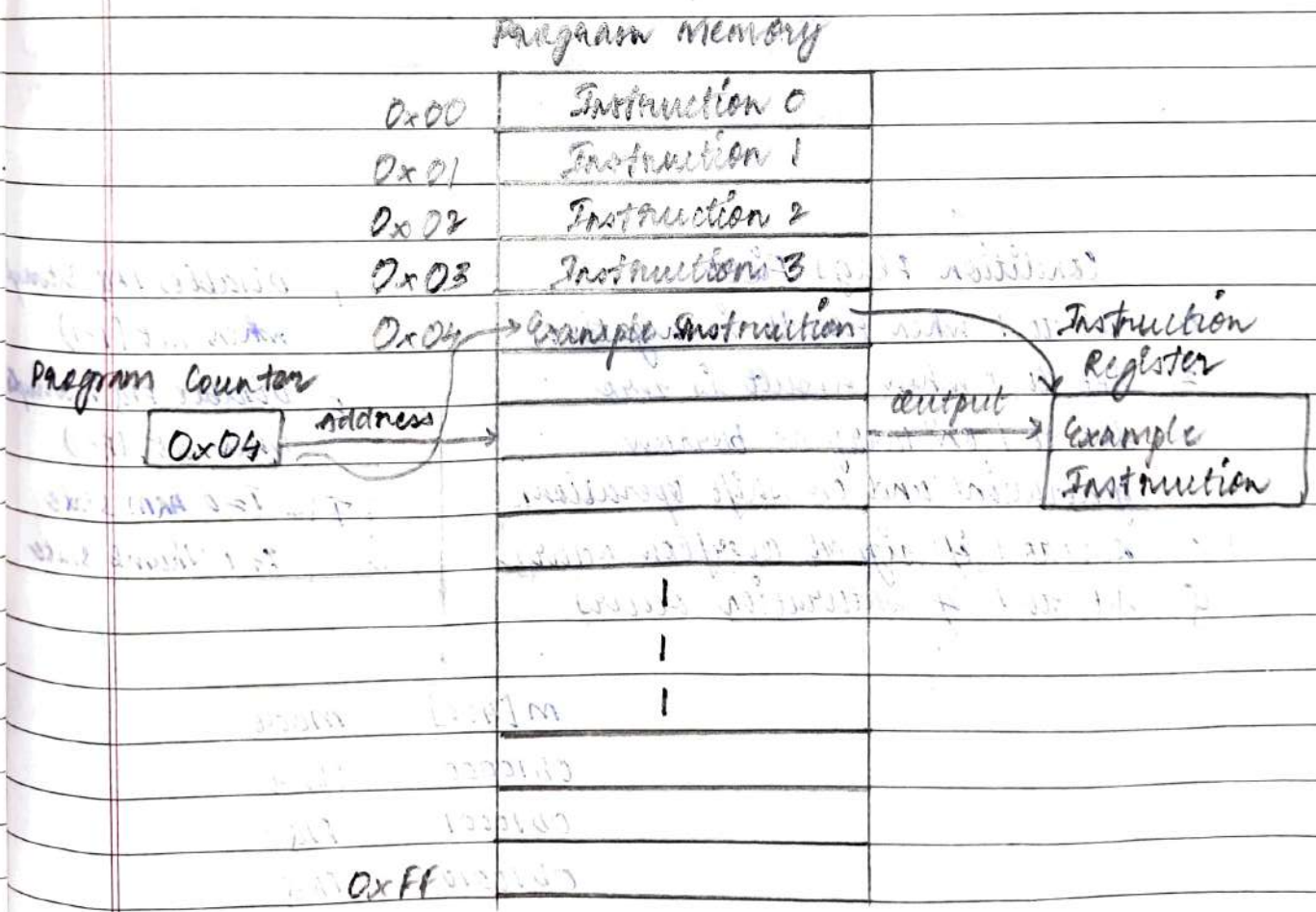
Links Register stores the return Address

PTO



* R15 - PROGRAM COUNTER :

Program Counter points to next Instruction that is to be executed by the Processor



P.T.O

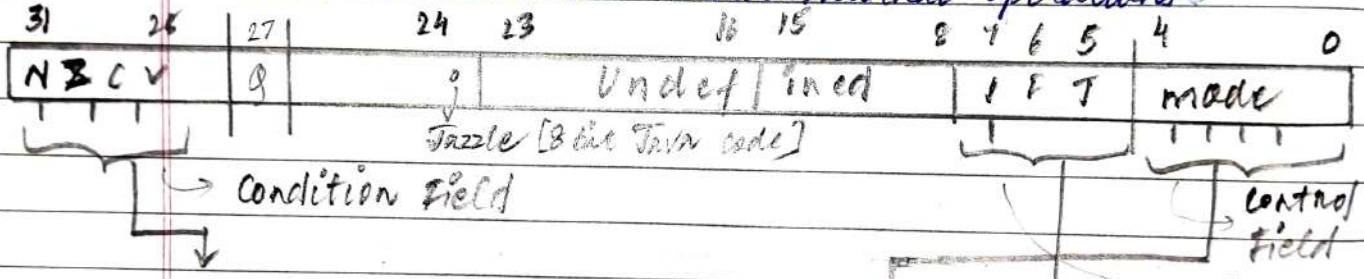
★ ARM REGISTER SETS

ARM has total 37 Registers, 32 bits each

1. Program Counter - 1
2. Current Program status Register (CPSR) - 1
3. Saved Program status Register (SPSR) - 5
4. General Purpose Registers - 30

★ PROGRAM STATUS REGISTER :

CPSR - monitors and control Internal Operations



Condition Flags Field	
N	Set to 1 when result is negative
Z	Set to 1 when result is zero
C	Set to 1 on carry or borrow generation and on shift operations
V	Set to 1 if signed overflow occurs
Q	Set to 1 if saturation occurs

I	Disables IRQ interrupt when set (I=1)
F	Disable FIQ interrupt when set (F=1)
T	T=0 ARM State
	T=1 Thumb state

M[4:0]	mode
0b10000	User
0b10001	FIQ
0b10010	IRQ
0b10011	Supervisor
0b10111	Abort
0b11011	Undefined
0b11111	System

b - binary

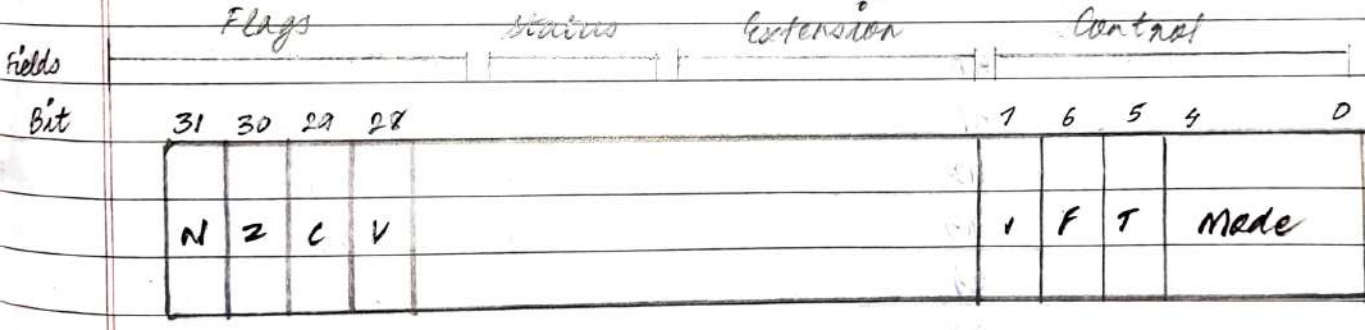
LECTURE 4 - ARM PROCESSOR MODES

* PROCESSOR MODES :

→ Processor Modes determine -
(i) Which registers are active
(ii) Access Rights to CPSR Register itself

→ Privileged -
(i) Full Read-Write access to the CPSR

→ Non-Privileged -
(i) Only Read access to the Control Field of CPSR
(ii) Read-Write access to the Condition Flags



ARM has Seven Processor Modes.

Privileged -
(i) Abort
(ii) Fast Interrupt Request (FIQ)
(iii) Interrupt Request (IRQ)
(iv) Supervisor
(v) Undefined
(vi) System

Non-Privileged -
(i) User

Data
① Abort - When there is a failed attempt to access memory
② Fast Interrupt Request (FIQ) - Entered on High Priority Interrupt Request

(3) Interrupt Request (IRQ) - Entered on Normal Interrupt Request
(4) Supervisor ~~Register~~ mode - Mode after Reset and mode in which OS kernel executes
(5) Undefined - When the processor encounters an Undefined Instruction
(6) User - Mode in which most Applications run
(7) System - Special version of User mode that allows Full Read-Write access of CPSR.

* ARM REGISTERS:

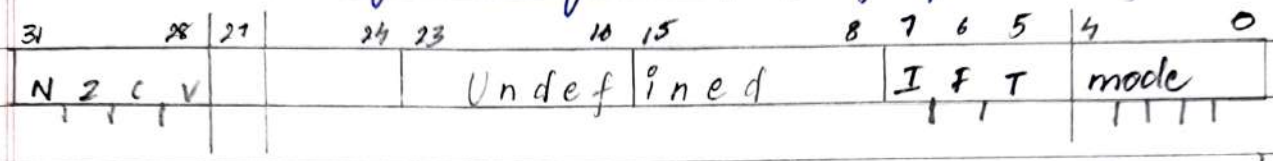
Banked
Hidden, only visible when processor moves to corresponding mode of operation

User mode	IRQ	FIQ	Undef	Abort
r0	ARM has 37 registers, all 32-bits long.			
r1				
r2	A subset of these registers is accessible in each mode			
r3				
r4				
r5	Note: System mode uses the User mode registers			
r6				
r7				
r8		r8		
r9		r9		
r10		r10		
r11		r11		
r12		r12		
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
r15 (pc)				
cpsr				
current mode	spsr	spsr	spsr	spsr
	Banked out registers			

SPSR registers are used to save CPSR of previous mode

* MODE CHANGING :

There are two ways to change the mode of operations -



① Direct write to mode Bits
② On Exception or Interrupt

m[4:0]	mode
0b10000	User
0b10001	FIQ
0b10010	IRQ
0b10011	Supervisor
0b10111	Abort
0b11011	Undefined
0b11111	System

Exceptions

SVC — Reset, FIQ, IRQ, SWI, Data Abort
Pre-fetch Abort and Undefined

Actions by processor when exceptions occur -

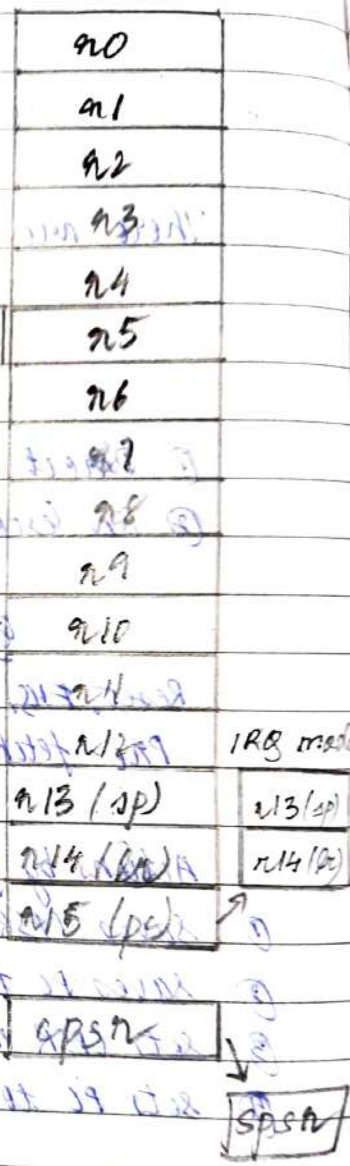
① Saves CPSR to SPSR of exception mode
② Saves PC to LR (R15) of exception mode — (Linked Register - store the return address)
③ Sets CPSR mode Bits to corresponding exception
④ Sets PC to address of exception handler

[PTO for diagrammatic representation]

* EXCEPTION PRIORITY ORDER :

① Reset Processor
② Data Abort
③ FIQ
④ IRQ
⑤ Pre-fetch Abort
⑥ SWI, Undefined Instruction
↓
Software Interrupt

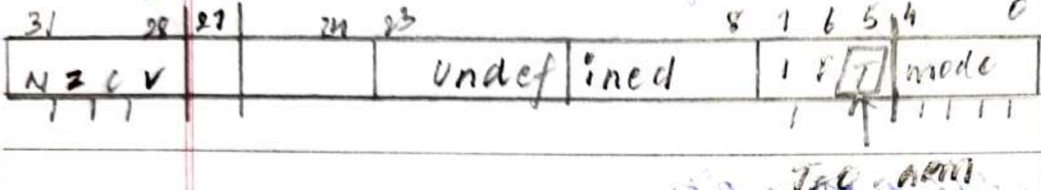
user mode



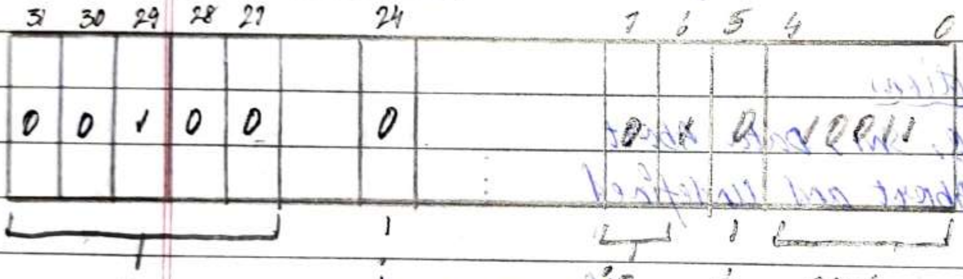
* PROCESSOR STATES :

Determines which instruction set is being executed.

① ARM state - 32 bit instructions
② Thumb state - 16 bit instructions



* EXERCISE - CPSR :



Mode - Supervisory
State - ARM

LECTURE 5 - ARM INSTRUCTION SET (INTRODUCTION)

* INSTRUCTION SETS:

① ARM
- standard 32-bit instruction set

② THUMB
- 16-bit compressed form of instruction set
- code density better than most CISC
- Performance degraded

* FEATURES OF ARM INSTRUCTION SET:

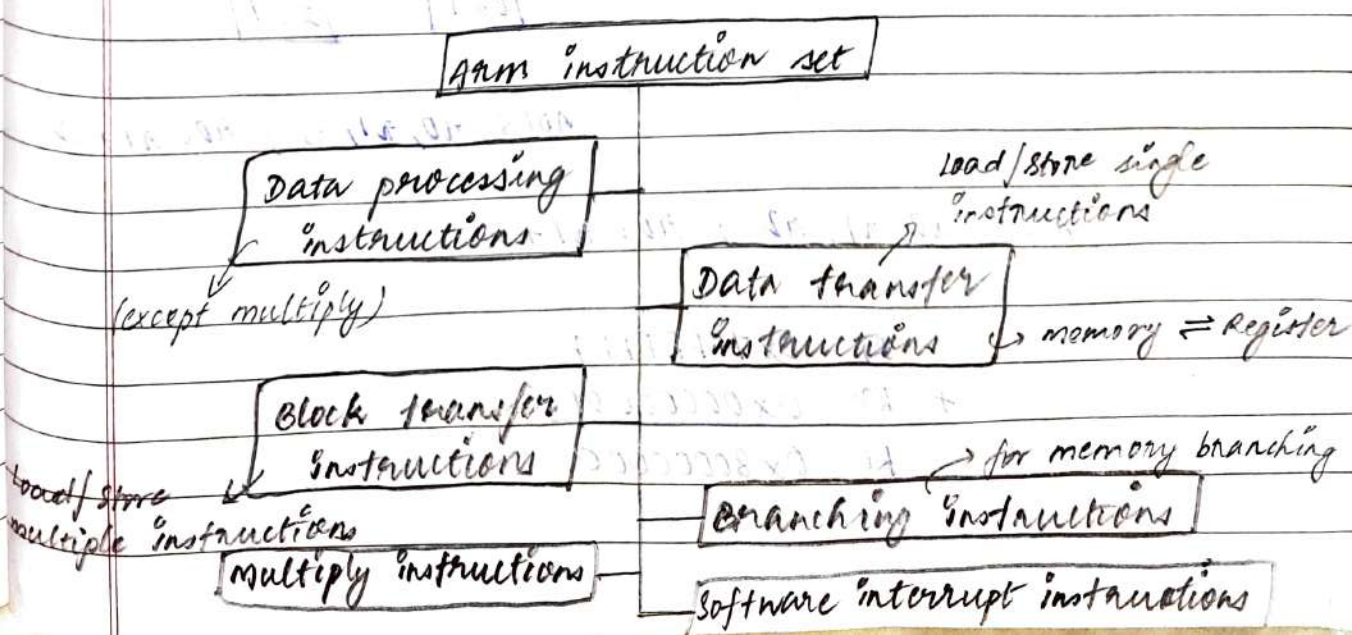
1. Load/store architecture - Memory ⇆ Register
2. 3-address data processing instructions (RESULT, FIRST OPERAND, SECOND OPERAND).
3. Conditional execution
4. Load/store multiple registers in a single instruction
5. Shift and ALU operation in single clock cycle

* CONDITIONAL EXECUTION:

- Each data processing instruction prefixed by condition code
- 16 condition codes

ADDEQ r0, r1, r2 ; if Z=1, r0 = r1 + r2

EQ	equal	MI	negative	HI	unsigned higher	GT	signed greater than
NE	not equal	PL	positive or zero	LS	unsigned lower or same	LE	signed less than or equal
CS	unsigned higher or same	VS	overflow	GE	signed greater than or equal	AL	always
CC	unsigned lower	VC	no overflow	LT	signed less than	NV	special purpose



LECTURE 6 - PART 1. DATA PROCESSING INSTRUCTIONS

* FEATURES :

1. Arithmetic and logical operations
2. 3-address format -
   (i) Two 32-bit source operands (op1 is always register, op2 is register or shifted register or immediate)
   (ii) 32-bit result placed in a register
3. Barrel shifter for op2 allows full 32-bit shift within instruction cycle. (multiple shifts in single instruction)

* ARITHMETIC OPERATIONS :

destination ← source operands / registers

```
ADD R0, R1, R2          ; R0 = R1 + R2
ADD R0, R1, #2          ; R0 = R1 + 2     (#2 → immediate data)
ADD R0, R1, R1, LSL #1  ; R0 = R1 + R1 << 1   (LSL → logic shift left)
```

```
ADD R0, R1, R2 ; R0 = R1 + R2
```

	R1	0x00000001	R1	0xFFFFFFFF
+	R2	0x00000002	+ R2	0x00000001
	R0	0x00000003	R0	0x00000000
				C = 1, Z = 1

Status Flag Updation

```
ADDS R0, R1, R2 ; R0 = R1 + R2
```

```
ADDS R0, R1, R2 ; R0 = R1 + R2
```

	R1	0x7FFFFFFF
+	R2	0x00000001
	R0	0x80000000

MSB - number is positive

R1	0111	1111	1111	1111	1111	1111	1111	1111
+ R2	0000	0000	0000	0000	0000	0000	0000	0001
R0	1000	0000	0000	0000	0000	0000	0000	0000

MSB - number is negative

V=1

N=1

Hexadecimal

ADD R0, R1, #2 ; R0 = R1 + 2

R1	0x00000001
+	2
R0	0x00000003

ADD R0, R1, #1F ; R0 = R1 + 0x0000000F

R1	0x00000001
+	0x0000000F
R0	0x00000010

ADD R0, R1, R1, LSL #1 ;
R0 = R1 + R1 << 1

R1	0x00000001
+R1, LSL #1	0x00000002
R0	0x00000003

R0 = R1 + R1 << 1
= R1 + 2 x R1
= 3 x R1

→ Add with carry

ADC R0, R1, R2 ;
R0 = R1 + R2 + C

R1	0x00000001
+ R2	0x00000002
+ C	1
R0	0x00000004

SUB R0, R1, R2 ; R0 = R1 - R2

SUB R0, R2, R1 ; R0 = R2 - R1
RSB R0, R1, R2 ; R0 = R2 - R1
} - Same

Reverse Subtract

R1	0x00000003
- R2	0x00000002
R0	0x00000001

R2	0x00000002
- R1	0x00000003
R0	0xFFFFFFFF

RSBS R0, R1, R2 ; R0 = R2 - R1

R2	0x00000002
- R1	0x00000003
R0	0xFFFFFFFF

N=1 C=0

0xFFFFFFFF
↓ 1's Compliment
0x00000000
↓ +1
0x00000001

Addition	
C=0	No Carry
C=1	Carry

Subtraction	
C=0	Borrow
C=1	No Borrow

$$B = \sim C$$

→ Subtract with Carry

SBC R0, R1, R2 ; R0 = R1 - R2 - B

SBC R0, R1, R2 ; R0 = R1 - R2 - ~C

	R1	0x00000003
-	R2	0x00000001
	C	0
-	~C	1
	R0	0x00000001

→ Reverse - Subtract with Carry

RSC R0, R1, R2 ; R0 = R2 - R1 - ~C

	R2	0x00000003
-	R1	0x00000001
	C	0
-	~C	1
	R0	0x00000001

LECTURE 7 - ARM INSTRUCTION SET - DATA PROCESSING - II

* DATA PROCESSING INSTRUCTIONS :-

BIT-WISE LOGICAL OPERATIONS :

AND R0, R1, R2 ; R0 = R1 and R2

ORR R0, R1, R2 ; R0 = R1 or R2

EOR R0, R1, R2 ; R0 = R1 XOR R2

R1	0x FFFFFFFF
& R2	0x00000000
R0	0x00000000

R1	1111	1111	1111	1111	1111	1111	1111	1111
& R2	0000	0000	0000	0000	0000	0000	0000	0000
R0	0000	0000	0000	0000	0000	0000	0000	0000

↗ clear a bit in a register

BIC R0, R1, R2 ; R0 = R1 and not R2

R1	1111	1111	1111	1111	1111	1111	1111	1010
R2	0000	0000	0000	0000	0000	0000	0000	0101
& Not R2	1111	1111	1111	1111	1111	1111	1111	1010
R0	1111	1111	1111	1111	1111	1111	1111	1010

R1	0xFFFFFFFF
R2	0x00000005
R0	0xFFFFFFFA

+ REGISTER MOVEMENT OPERATIONS:

↗ content of R2 will be moved to R0

MOV R0, R2 ; R0 = R2

R2	0x00000002	R2	0000 0000	0010
R0	0x00000002	R0	0000 0000	0010

↗ complement of R2 will be moved to R0

MVN R0, R2 ; R0 = Not R2

R2	0x00000002	R2	0000 0000	0010
R0	0xFFFFFFFD	R0	1111 1111	1101

PTO →

* COMPARISON OPERATIONS :

→ Compares two parameters, no result stored

CMP R1, R2 ; Set CC on R1 - R2

R1 0x00000002	R1 0000 0000 ... 0010
- R2 0x00000002	- R2 0000 0000 ... 0010
Z = 1 (No Result)	Z = 1 (No Result)

R1 0x00000002	R1 0000 0000 ... 0010
- R2 0x00000003	- R2 0000 0000 ... 0011
N = 1 (No Result)	N = 1 (No Result)

CC (Condition Code field of CPSR) - N, Z, C, V

No Result

CMN R1, R2 ; Set CC on R1 + R2

R1 0x00000000	R1 0000 0000 ... 0000
+ R2 0x00000000	+ R2 0000 0000 ... 0000
Z = 1	Z = 1

R1 0xFFFFFFFF	R1 1111 1111 1111 ... 1111
+ R2 0x00000001	+ R2 0000 0000 0000 ... 0001
C = 1	C = 1

CC (Condition Code field of CPSR) - N, Z, C, V

No Result

P.T.O.

TST r1, r2 ; Set CC on r1 & r2 - Test single bit is 1 or 0 (Particular)

R1 0xFFFFFFFF	R1 1111 1111 1111 .... x4 1111
& R2 0x00000001	& R2 0000 0000 0000 .... x4 0001
Z=0	Z=0 - So the bit is '1'

R1 0xFFFFFFFE	R1 1111 1111 1111 .... x4 1110
& R2 0x00000001	& R2 0000 0000 0000 .... x4 0001
Z=1	Z=1 - So the bit is '0'

CC (Condition Code field of CPSR) - N, Z, C, V
No Result

TEQ r1, r2 ; Set CC on r1 ^ r2 - Test both numbers are equal (XOR)

R1 0xFFFFFFFE	R1 1111 1111 1111 .... x4 1110
^R2 0xFFFFFFFE	^R2 1111 1111 1111 .... x4 1110
Z=1	Z=1 Both Nos are Same

R1 0xFFFFFFFE	R1 1111 1111 1111 .... x4 1110
^R2 0xFFFFFFFF	^R2 1111 1111 1111 .... x4 1111
Z=0	

CC (Condition Code field of CPSR) - N, Z, C, V
No Result

* SHIFTED REGISTER OPERANDS :

① LSL - logical shift left by 0 to 31 places ; fill the vacated bits at the least significant end of the word with zeros.

② LSR - logical shift right by 0 to 32 places ; fill the vacated bits at the most significant end of the word with zeros.

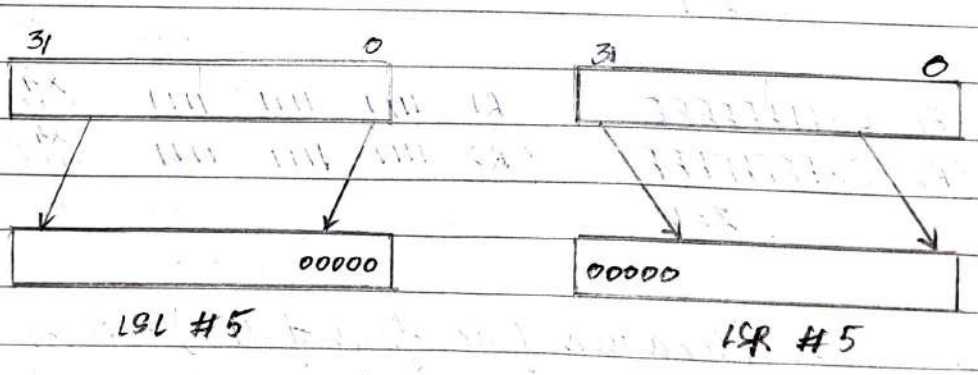
PTO

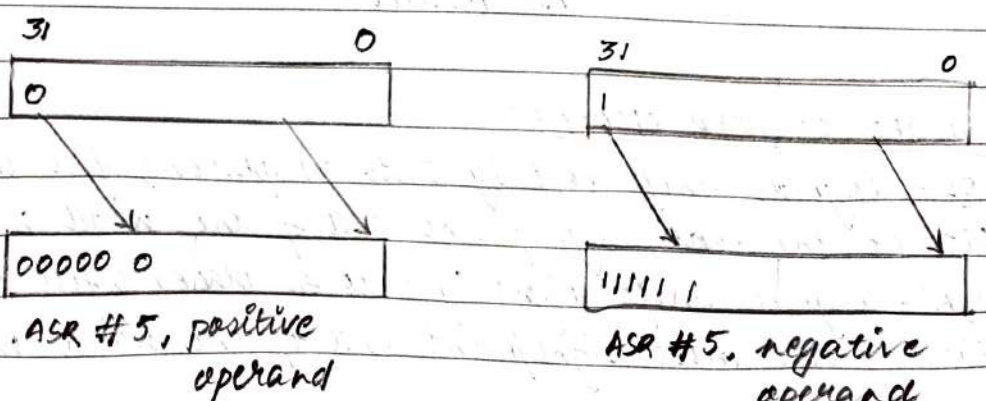
③ ASR - arithmetic shift right by 0 to 32 places; fill the vacated bits at the most important end of the word with zeros if the source operand was positive, or with ones if the source operand was negative.

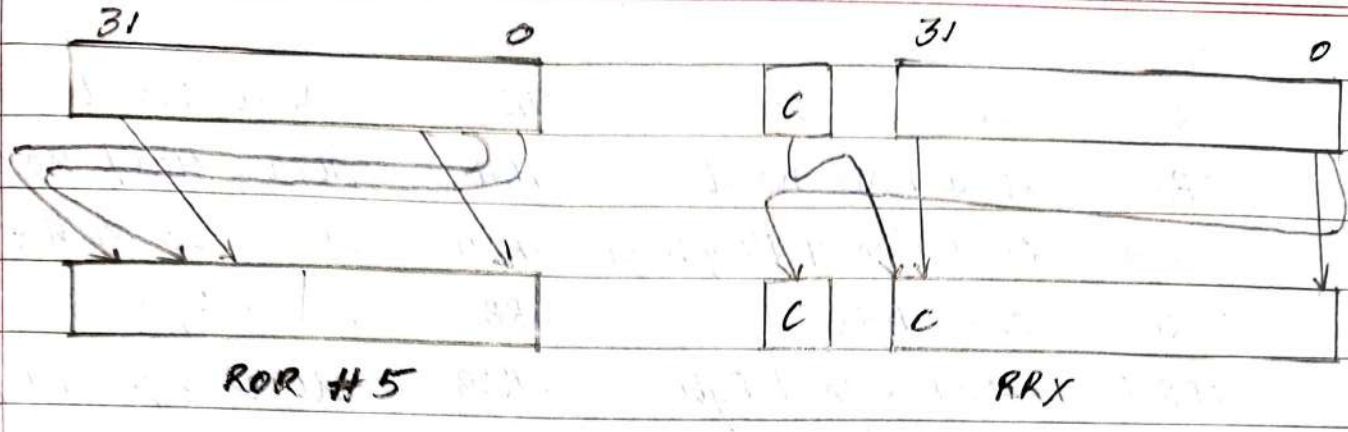
④ ROR - rotate right by 0 to 32 places; the bits which fall off the least significant end of the word are used, in order, to fill the vacated bits at the most significant end of the word.

⑤ RRX - rotate right extended by 1 place; the vacated bit (bit 31) with the old value of the C flag and the operand is shifted one place to the right. With appropriate use of the condition codes (see below) a 33-bit rotate of the operand and the C flag is performed.

ADD R3, R2, R1, LSL #3; R3 := R2 + 8 * R1







ADD r0, r1, r1, LSL #1 ; r0 = r1 + r1 << 1

R1	0x00000001		R1	0000 0001
+ R1, LSL #1	0x00000002		+ R1, LSL #1	0000 0010
R0	0x00000003		R0	0000 0011

r0 = r1 + r1 << 1 = r1 + 2 x r1 = 3 x r1

e.g. :

if (z == 1) R1 = R2 + (R3 + 4)

complies to

ADDEQS R1, R2, R3, LSL #2

(SINGLE INSTRUCTION !)

LECTURE 8 - BASIC ADDITION PROGRAM

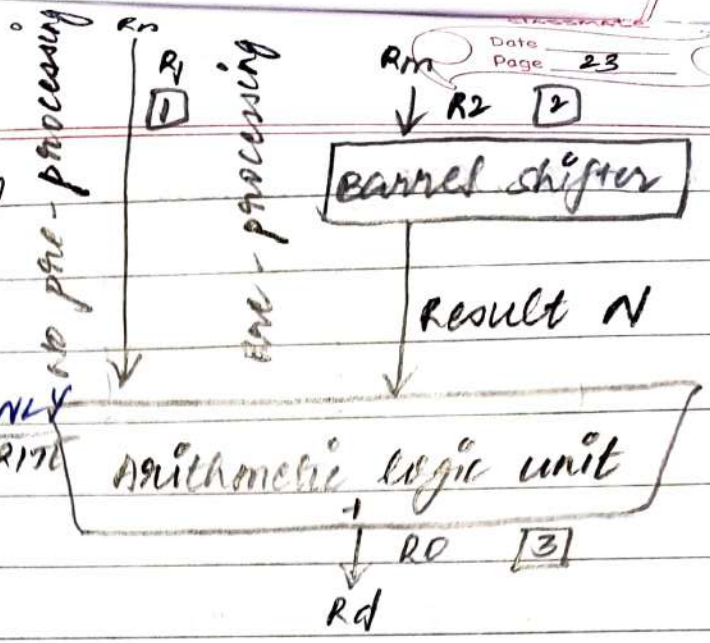
+ MEMORY MAP - LPC 2148 :

[refer to page 2 for diagrammatic representation.]

① DATA - On-chip static RAM (memory)

② PROGRAM - On-chip Non-Volatile memory - Flash ROM (Code m

4.0 GB		0xFFFF FFFF
	AHB PERIPHERALS	
3.75 GB		0xF000 0000
	VPB PERIPHERALS	
3.5 GB		0xE000 0000
3.0 GB	RESERVED ADDRESS SPACE	0xC000 0000
2.0 GB		0x8000 0000
	BOOT BLOCK	
	(12 KB REMAPPED FROM ON-CHIP FLASH MEMORY)	0x7FFF D000
		0x7FFF CFFF
	RESERVED ADDRESS SPACE	
		0x7FD0 2000
	8 KB ON-CHIP USB DMA RAM (LPC 2146/2148)	0x7FD0 1FFF
		0x7FD0 0000
	RESERVED ADDRESS SPACE	0x7FCF FFFF
RAM DATA Memory		0x4000 2000
	32 KB ON-CHIP STATIC RAM (LPC2146/2148)	0x4000 1FFF
		0x4000 4000
	16 KB ON-CHIP STATIC RAM (LPC2142/2144)	0x4000 3FFF
		0x4000 2000
1.0 GB	8 KB ON-CHIP STATIC RAM (LPC 2141)	0x4000 1FFF
		0x4000 0000
		0x3FFF FFFF
	RESERVED ADDRESS SPACE	
		0x0008 0000
Build read/write 1,0,0,000 Program		0x0007 FFFF
	TOTAL OF 512 KB ON-CHIP NON-VOLATILE MEMORY (LPC 2148)	0x0004 0000
		0x0003 FFFF
	TOTAL OF 256 KB ON-CHIP NON-VOLATILE MEMORY (LPC2146)	0x0002 0000
		0x0001 FFFF
	TOTAL OF 128 KB ON-CHIP NON-VOLATILE MEMORY (LPC 2144)	0x0001 0000
PROGRAM FLASH ROM (CODE M/m)		0x0000 FFFF
	TOTAL OF 64 KB ON-CHIP NON-VOLATILE MEMORY (LPC 2142)	0x0000 8000
		0x0000 7FFF
	TOTAL OF 32 KB ON-CHIP NON-VOLATILE MEMORY (LPC 2141)	
0.0 GB		0x0000 0000



ASSEMBLY LANGUAGE PROGRAM

TO ADD TWO NUMBERS:

Assembler Directive

```
AREA ADDITION, CODE, READONLY
                DATA  READWRITE
START → ENTRY
        MOV R1, #1
        MOV R2, #2
        ADD R0, R1, R2
STOP B STOP ← Label
Discard → END
```

32-bit of code

(HEX)	Program m/m
0x00000000	MOV R1, #1
0x00000004	MOV R2, #2
0x00000008	ADD R0, R1, R2
0x0000000C	B 0x0000000C
0x00000010	
.....	

Short Jump →

LECTURE 9 - BINARY ENCODING

SUBEQS R0, R1, #2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0

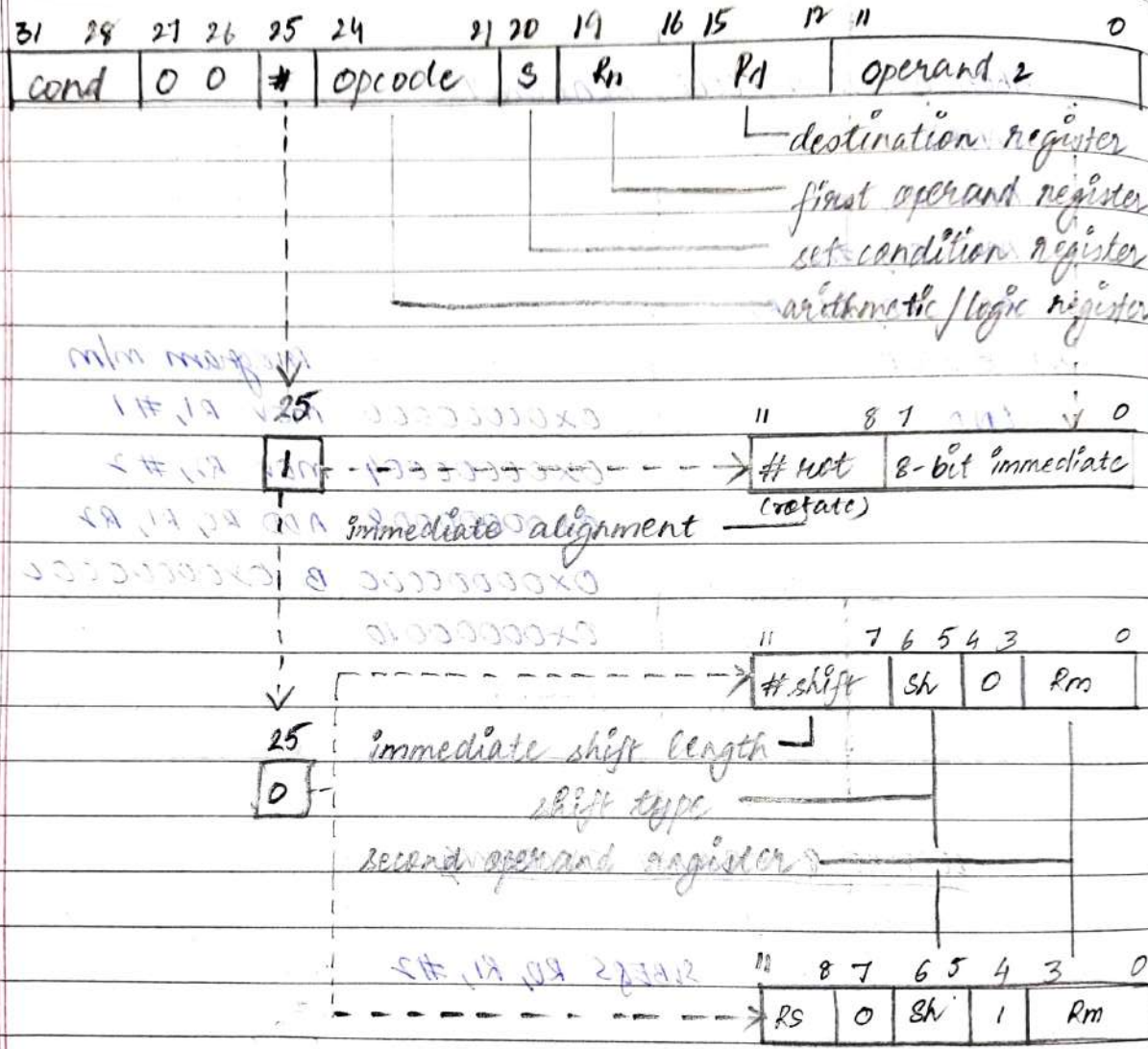
4 Bit Cond. code | Res. | Im. Op. | 4 Bit opCode | Op. 1 Reg. | Dest. Reg.

9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0

4 Bit Imm. RDR | 8-bit operand 2

PTO

* DATA PROCESS :



SUBEQS R0, R1, #2

* CONDITION CODES :

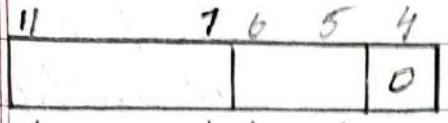
Code	Suffix	Flags	meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative

0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equal to V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

+ OPCODES :

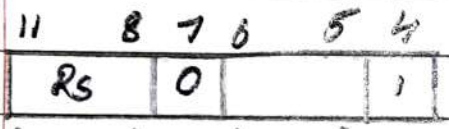
Assembler Mnemonic	OpCode	Action
AND	0000	operand1 AND operand2
EOR	0001	operand1 EOR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry
SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2 (operand1 is ignored)
BIC	1110	operand1 AND NOT operand2 (Bit clear)
MVN	1111	NOT operand2 (operand1 is ignored)

* SHIFTS:



Shift Type
00 = logical left
01 = logical right
10 = arithmetic right
11 = rotate right

Shift amount
5 bit unsigned integer



Shift Type
00 = logical left
01 = logical right
10 = arithmetic right
11 = rotate right

Shift register
Shift amount specified in bottom byte of Rs

LECTURE 10 : ARM INSTRUCTION SET - DATA TRANSFER - I

① Load / store instructions

② Used to move Word (32-bit), signed and unsigned Half Word (16-bit) and Byte to and from registers

③ Can be used to load PC [Program Counter]
(if target address is beyond branch instruction range)

PTO

Load - Memory to Register
Store - Register to Memory

LDR	Load Word	STR	Store Word
LDRH	Load Half Word	STRH	Store Half Word
LDRSH	Load Signed Half Word	STRSH	Store Signed Hdf Word
LDRB	Load Byte	STRB	Store Byte
LDRSB	Load Signed Byte	STRSB	Store Signed Byte

* SINGLE REGISTER LOAD :
(BASE PLUS OFFSET PRE-INDEXED WITH IMMEDIATE OFFSET)

```
LDR R0, = 0x40000000
LDR R1, [R0]
LDR R2, [R0, #4]
```

Base Plus Offset
Pre-indexed Addr. Mode
with Imm. Offset

Indexing is done before load operation

	Data M/M		Data
R0 →	0x40000000	0x10101010	R0 0x40000000
	0x40000004	0x20202020	R1 0x10101010
	0x40000008	0x00000003	R2 0x20202020

* SINGLE REGISTER LOAD : ⇒ Increment First, Load Next
(BASE PLUS OFFSET PRE-INDEXED WITH IMMEDIATE OFFSET AND WRITE BACK)

```
LDR R0, = 0x40000000
LDR R1, [R0]
LDR R2, [R0, #4]!
```

Pre-indexed Addr. Mode
with Imm. Offset
& Write Back
(Auto-indexing)

Pointer value is updated at the base

	Data M/M
R0 →	0x40000000 0x10101010
	0x40000004 0x20202020
	0x40000008 0x00000003

	Data
R0	0x40000004
R1	0x10101010
R2	0x20202020

* SINGLE REGISTER LOAD: → Load First, Increment Next
(BASE PLUS OFFSET POST-INDEXED WITH IMMEDIATE OFFSET)

```
LDR R0, =0x40000000
LDR R1, [R0]
LDR R2, [R0], #4
```

Post-indexed Addr. Mode with Imm. offset

	Data M/M
R0 → 0x40000000	0x10101010
0x40000004	0x20202020
0x40000008	0x00000003

	Data
R0	0x40000004
R1	0x10101010
R2	0x10101010

* SINGLE REGISTER LOAD AND STORE:

```
LDR R0, =0x40000000
LDR R1, [R0]
LDR R2, [R0], #4
STR R2, [R0, #4]!
```

Post-indexed Addr. Mode

Pre-indexed Addr. Mode with write back

	Data m/m
R0 → 0x40000000	0x10101010
0x40000004	0x20202020
0x40000008	0x10101010

	Data
R0	0x40000008
R1	0x10101010
R2	0x10101010

LECTURE 11: ARM INSTRUCTION SET - DATA TRANSFER - II

* SINGLE REGISTER LOAD (DECREMENT):

```
LDR R0, = 0x40000004
LDR R1, [R0]
LDR R2, [R0, #-4]!
```

Pre-indexed Addn Mode with write back (auto-indexing)

	Data M/M
0x40000000	0x10101010
0x40000004	0x20202020
0x40000008	0x00000003

R0 →

	Data
R0	0x40000000
R1	0x20202020
R2	0x10101010

* SINGLE REGISTER LOAD:

(BASE PLUS OFFSET PRE-INDEXED WITH REGISTER OFFSET AND WRITE BACK)

```
LDR R0, = 0x40000000
LDR R1, [R0]
mov R2, #4          ← LDR R3, [R0, R2]!
[done while programming]
```

Pre-indexed Reg. offset with write back (auto-indexing)

	Data M/M
0x40000000	0x10101010
0x40000004	0x20202020
0x40000008	0x00000003

R0 →

	Data
R0	0x40000004
R1	0x10101010
R2	0x00000004
R3	0x20202020

PTO →

* SINGLE REGISTER LOAD :

(BASE PLUS OFFSET PRE-INDEXED WITH SCALED REGISTER OFFSET AND WRITE BACK)

```
LDR R0, = 0x40000000
LDR R1, [R0]
MOV R2, #1          LDR R3, [R0, R2, LSL #2]!
[Done while programming]
```

Pre-indexed
Scaled Reg. offset
with Write Back

	Data M/M
0x40000000	0x10101010
0x40000004	0x20202020
0x40000008	0x00000002

	Data
R0	0x40000004
R1	0x10101010
R2	0x00000001
R3	0x20202020

* SINGLE REGISTER LOAD HALFWORD :

(BASE PLUS OFFSET PRE-INDEXED WITH IMMEDIATE OFFSET)

```
LDR R0, = 0x40000000
LDRH R1, [R0]
LDRH R2, [R0, #4]
```

Base Plus offset
Pre-indexed Addr. mode
with Imm. offset

	Data M/M
0x40000000	0x10101010
0x40000004	0x20202020
0x40000008	0x00000003

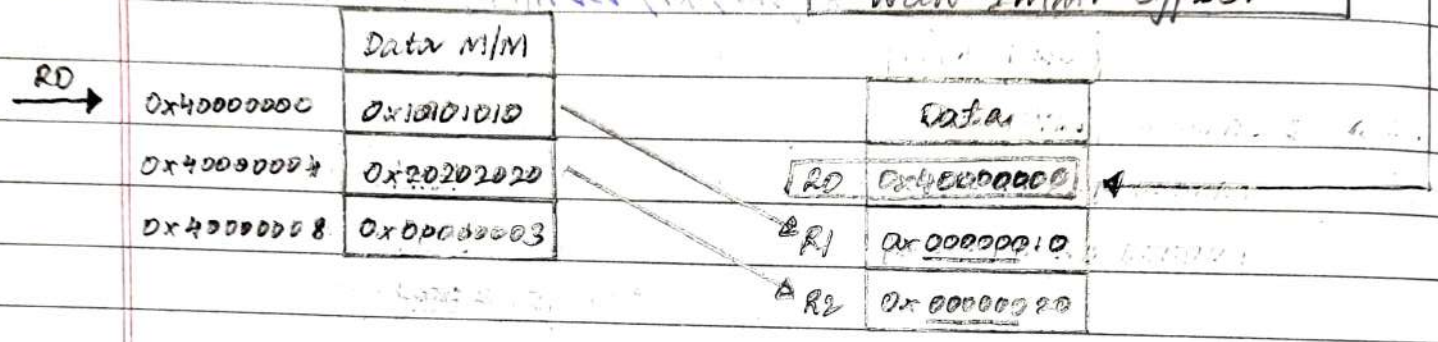
	Data
R0	0x40000000
R1	0x00001010
R2	0x00002020

PTO →

* SINGLE REGISTER LOAD BYTE :
(BASE PLUS OFFSET PRE-INDEXED WITH IMMEDIATE OFFSET)

```
LDR R0, = 0x40000000
LDRB R1, [R0]
LDRB R2, [R0, #4]
```

Base Plus Offset
Pre-indexed Addr. Mode with Imm. Offset

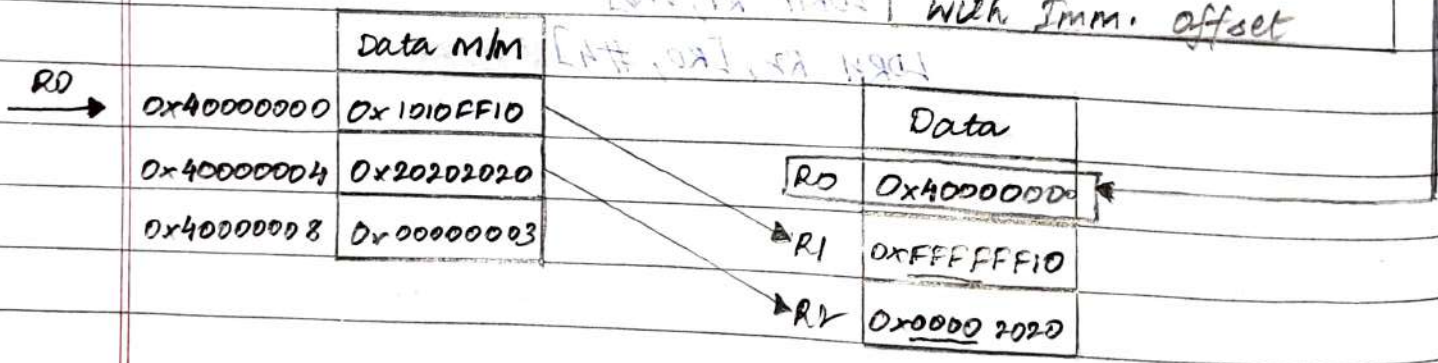


* SINGLE REGISTER LOAD SIGNED HALFWORD :
(BASE PLUS OFFSET PRE-INDEXED WITH IMMEDIATE OFFSET)

```
LDR R0, = 0x40000000
LDRSH R1, [R0]
LDRSH R2, [R0, #4]
```

MSB Signed extension ←

Base Plus offset
Pre-indexed Addr. Mode with Imm. offset



PTO

* SINGLE REGISTER LOAD SIGNED BYTE :
(BASE PLUS OFFSET PRE-INDEXED WITH IMMEDIATE OFFSET)

LDR R0, = 0x40000000

MSB Signed Extension ← LDRSB R1, [R0] — Base Plus Offset

LDRSB R2, [R0, #4] → Pre-indexed addr. Mode with Imm. Offset

	Data M/m
R0 → 0x40000000	0x101010FF
0x40000004	0x20202020
0x40000008	0x00000003

	Data
R0	0x40000000
R1	0xFFFFFFFF
R2	0x00000020

* EXAMPLE :

R0 = 0x00000000

R1 = 0x 00090000

mem 32 [0x00009000] = 0x01010101

mem 32 [0x00009004] = 0x02020202

LDR R0, [R1, #4] !

Pre-indexing with writeback :

R0 = 0x02020202

R1 = 0x00009004

LDR R0, [R1, #4]

Pre-indexing :

R0 = 0x02020202

R1 = 0x00009000

LDR R0, [R1], #4

Post-indexing :

R0 = 0x01010101

R1 = 0x 0009004

* LOAD / STORE ADDRESSING MODES :

```
LDR   R0 , = 0x40000000      ; R0 = 0x40000000
LDR   R1 , [R0]              ; R1 = mem32[0x40000000]
LDR   R1 , [R0, #4]          ; R1 = mem32[0x40000000 + 4]
LDR   R1 , [R0, #-4]         ; R1 = mem32[0x40000000 - 4]
LDR   R1 , [R0, R2]          ; R1 = mem32[0x40000000 + R2]
LDR   R1 , [R0, -R2]         ; R1 = mem32[0x40000000 - R2]
LDR   R1 , [R0, R2, LSL #2]  ; R1 = mem32[0x40000000 + R2 << 2]
LDR   R1 , [R0, -R2, LSL #2] ; R1 = mem32[0x40000000 - R2 << 2]
LDRH  R1 , [R0]              ; R1 = mem16[0x40000000]
LDRB  R1 , [R0]              ; R1 = mem8[0x40000000]
LDRSH R1 , [R0]              ; R1 = Sign Ext. mem16[0x40000000]
LDRSB R1 , [R0]              ; R1 = Sign Ext. mem8[0x40000000]
STR   R1 , [R0]              ; mem32[0x40000000] = R1
```

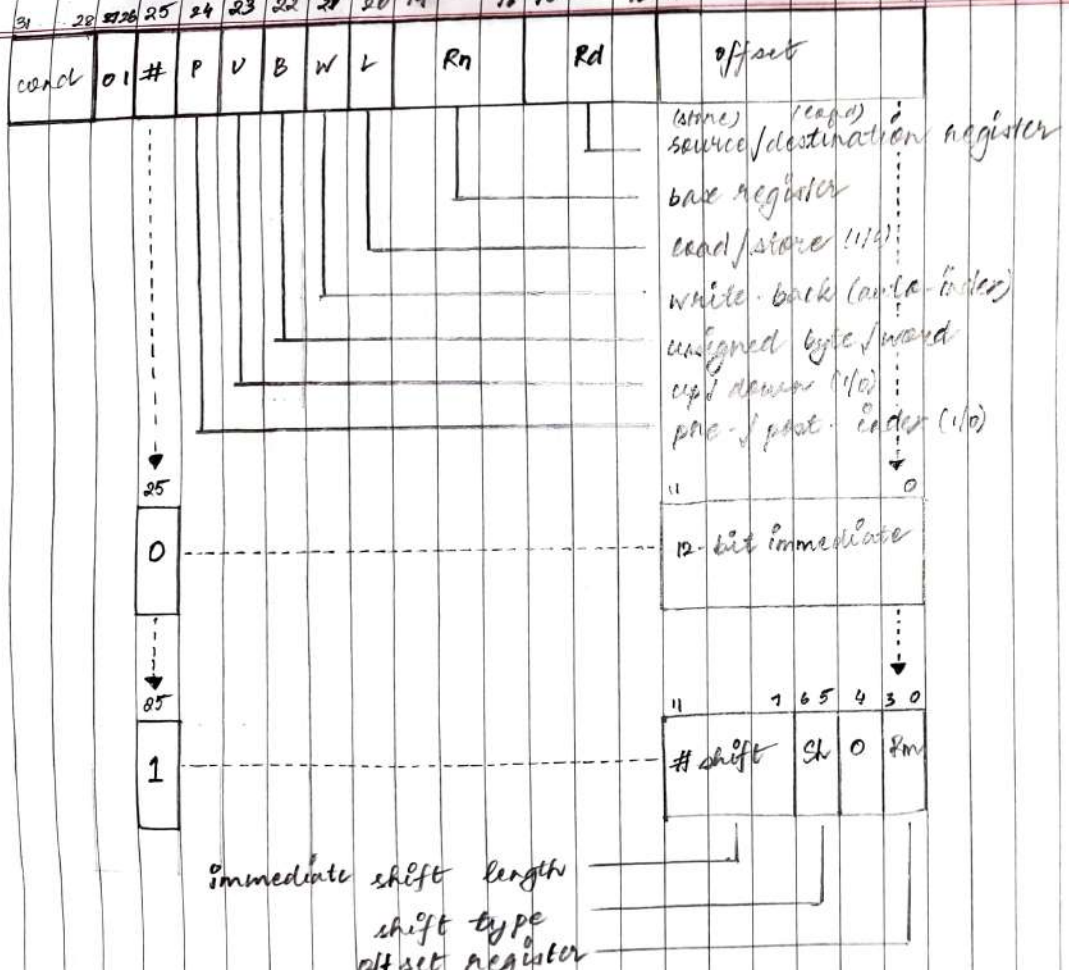
LECTURE 12 - DATA TRANSFER - BINARY ENCODING

* DATA TRANSFER INSTRUCTIONS :

① Load/store instructions
② Used to move signed and unsigned Word, Half Word and Byte to and from registers
③ Can be used to load PC (if target address is beyond branch instruction range)

LDR	Load Word	STR	Store Word
LDRH	Load Half Word	STRH	store Half Word
LDRSH	Load Signed Half Word	STRSH	store Signed Half Word
LDRB	Load Byte	STRB	store Byte
LDRSB	Load Signed Byte	STRSB	store signed Byte

DATA TRANSFER + SINGLE WORD & UNSIGNED BYTE :



load/store bit
0 = store to memory
1 = load from memory

write-back bit
0 = no write-back
1 = write address into base

byte/word bit
0 = transfer word quantity
1 = transfer byte quantity

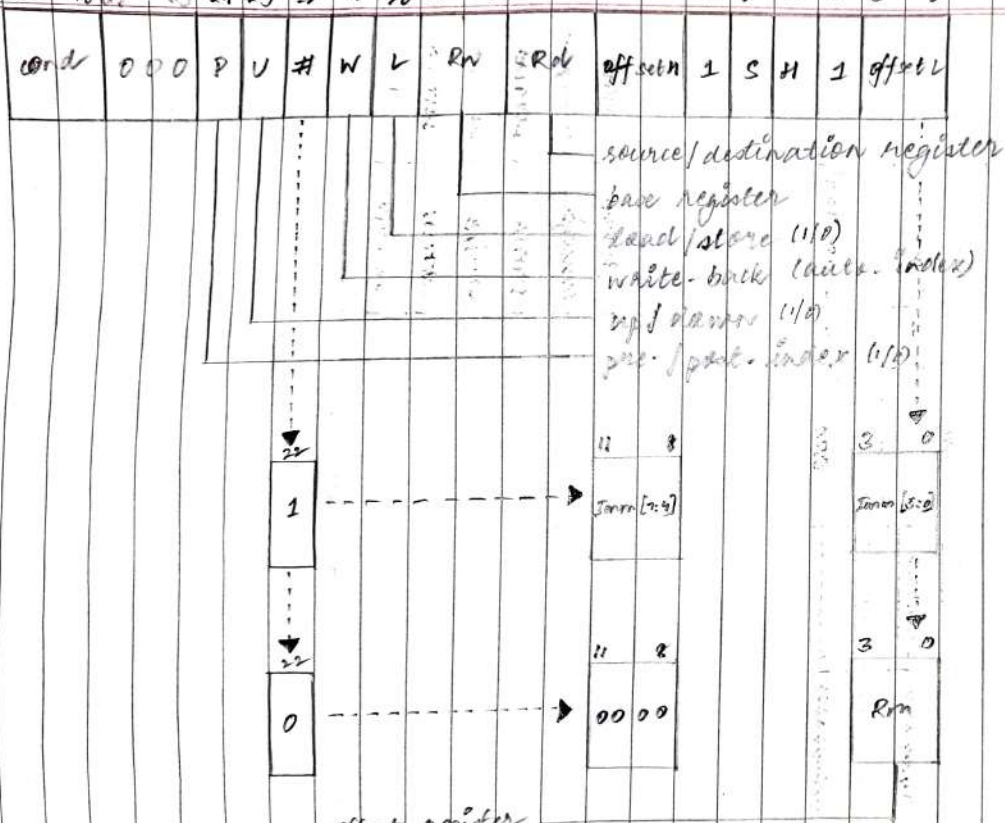
up/down bit
0 = down; subtract offset from base
1 = up; add offset to base

pre/post indexing bit
0 = post; add offset after transfer
1 = pre; add offset before transfer

Immediate offset
0 = offset is an immediate value
1 = offset is a register

Shift type
00 = logical left
01 = logical right
10 = arithmetic right
11 = rotate right

* DATA TRANSFER - HALF WORD AND SIGNED BYTE :



S H
- 00 = SWP instruction
- 01 = unsigned halfwords
- 10 = signed byte
- 11 = signed halfwords

load/store
- 0 = store to memory
- 1 = load from memory

write-back
- 0 = no write back
- 1 = write address into base

up/down
- 0 = down: subtract offset from base
- 1 = up: add offset to base

pre/post indexing
- 0 = post: add/subtract offset after transfer
- 1 = pre: add/subtract offset before transfer

LECTURE 13 - ARM INSTRUCTION SET - BLOCK DATA TRANSFER

* MULTIPLE REGISTER LOAD/STORE (BLOCK TRANSFER):

① Load Multiple instruction (LDM) - Block of data memory moved to corresponding register
② store Multiple instruction (STM) - opposite of LDM
③ LDM takes 2+Nt cycles,
where N is number of registers to load
t is number of cycles required for each sequential access to memory

	Data m/m	LDM	Data
0x40000000	0x10101010	→ R0	0x10101010
0x40000004	0x20202020	→ R1	0x20202020
0x40000008	0x00000003	STM → R2	0x00000003

Syntax - <LDM|STM> {<cond>} <addressing mode> Rn{!}, <registers>

④ Addressing modes -

IA - Increment After
IB - Increment Before
DA - Decrement After
DB - Decrement Before

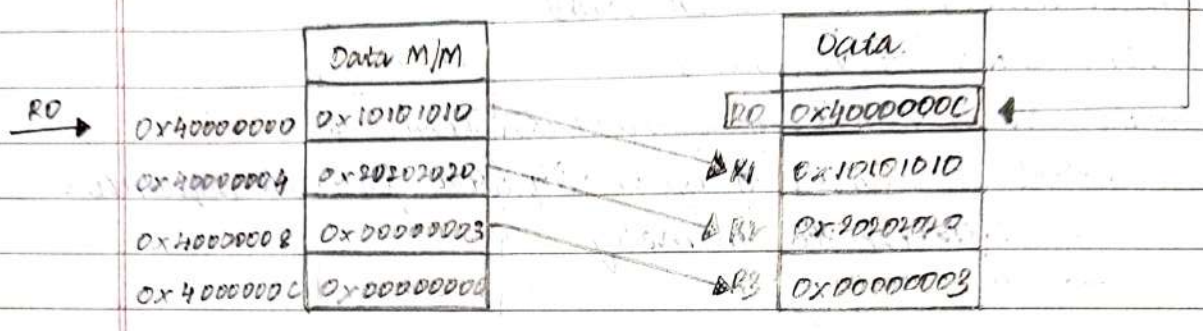
Store Multiple	Load Multiple
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

LDMIA R0!, {R1, R2, R3}
LDMIA R0!, {R1-R3}
LDMIA R0!, {R1-R3, R5}
STMIB R0!, {R1-R3}

PTO →

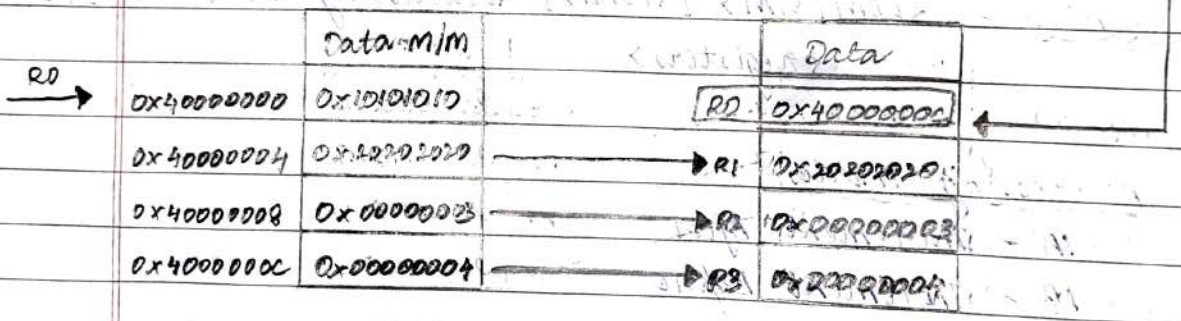
* MULTIPLE REGISTER LOAD (INCREMENT AFTER):

```
LDR R0, = 0x40000000
LDMIA R0!, {R1-R3}
```



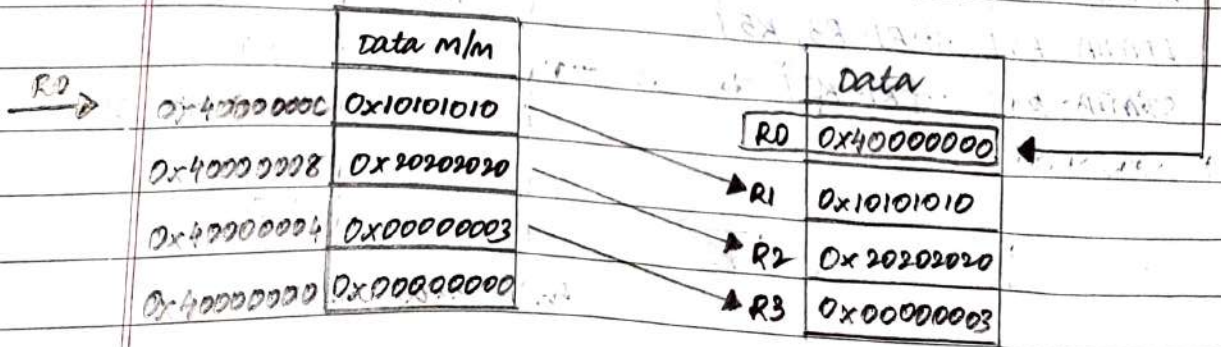
* MULTIPLE REGISTER LOAD (INCREMENT BEFORE):

```
LDR R0, = 0x40000000
LDMIB R0! {R1-R3}
```



* MULTIPLE REGISTER LOAD (DECREMENT AFTER):

```
LDR R0, = 0x4000000C
LDMDA R0!, {R1-R3}
```



* MULTIPLE REGISTER LOAD (DECREMENT BEFORE):

```
LDR R0, =0x4000000C
LDMDB R0!, {R1-R3}
```

	Data M/M		Data
R0 → 0x4000000C	0x10101010	R0	0x40000000
0x40000008	0x20202020	R1	0x20202020
0x40000004	0x00000003	R2	0x00000003
0x40000000	0x00000004	R3	0x00000004

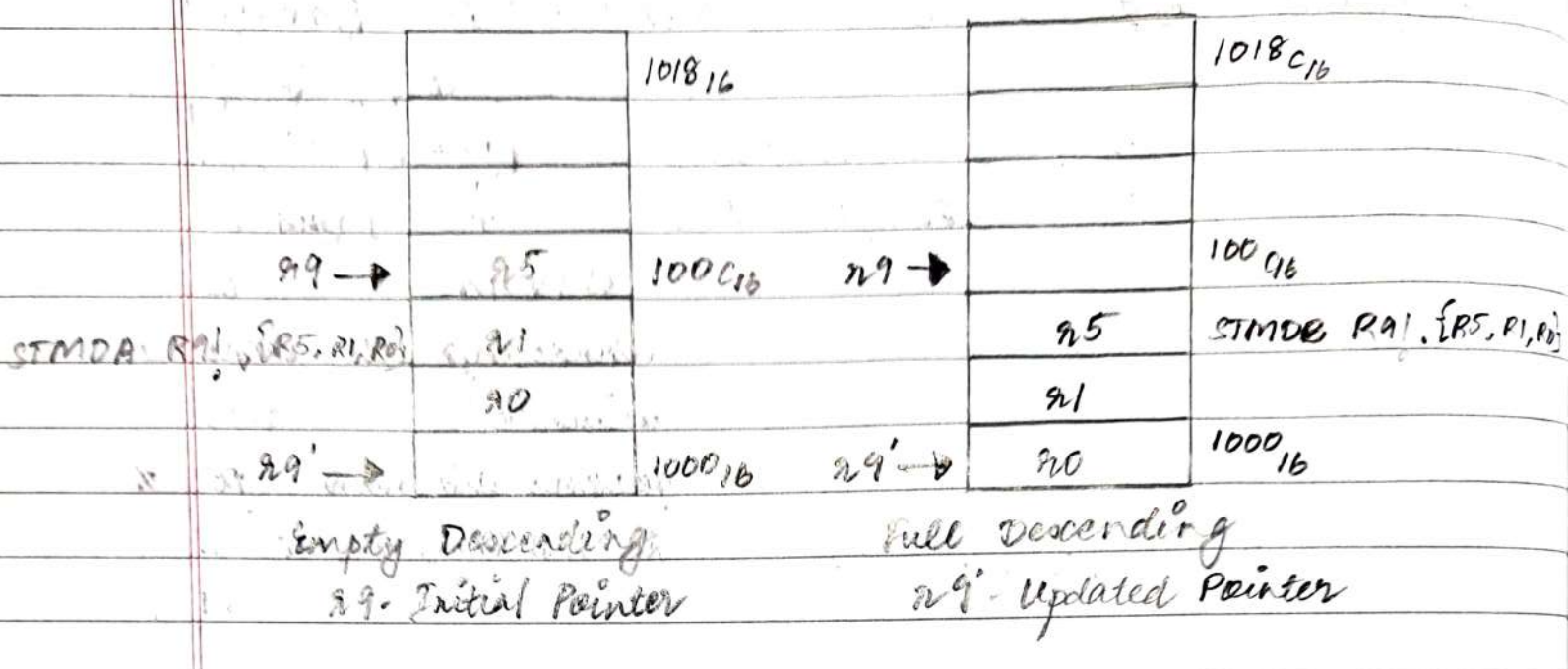
* MULTIPLE REGISTER STORE (DECREMENT AFTER):

```
LDR R0, =0x4000000C
STMDA R0!, {R1-R3}
```

	Data M/M		Data
R0 → 0x4000000C	0x10101010	R0	0x40000000
0x40000008	0x20202020	R1	0x10101010
0x40000004	0x00000003	R2	0x20202020
0x40000000	0x00000004	R3	0x00000003

* TYPES OF STACK (PUSH OPERATION) :- STM
(POP OPERATION) :- LDM

R9 →		1018_{16}	R9 →	R5	1018_{16}
	R5			R1	
	R1			R0	
R9 →	R0	$100C_{16}$	R9 →		$100C_{16}$
STMIA R9!, {R0, R1, R5}					STMIB R9!, {R0, R1, R5}
		1000_{16}			1000_{16}
	Empty Ascending			Full Ascending	



* RELATION BETWEEN BLOCK TRANSFER AND STACK OPERATIONS :

		Ascending		Descending	
		Full	empty	Full	Empty
Increment	Before	STMIB			LDMIB
	After	STMFA			LDMED
			STMIA	LDMIA	
	After		STMEA	LDMFD	
Decrement	Before		LDMDB	STMDB	
			LDMEA	STMFD	
	After	LDMDA			STMDA
		LDMFA			STMED

LECTURE 17 - NXP LPC2148 ARCHITECTURE

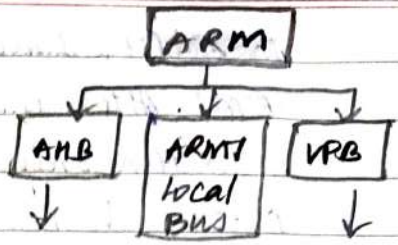
LPC2148 is a microcontroller having ARM7TDMI-S as the ARM core and manufactured by NXP, previously known as Philips.

* TERMINOLOGIES :

① AMBA AHB (Advanced High-Performance Bus) = Operates on higher speed compared to VPB

② VPB (VLSI peripheral bus)

③ Vectored Interrupt Controller (VIC).



④ AHB to VPB Bridge - Converts the data from low speed devices connected to VPB to a format that is accepted by the AHB. Speed mismatch will be handled.

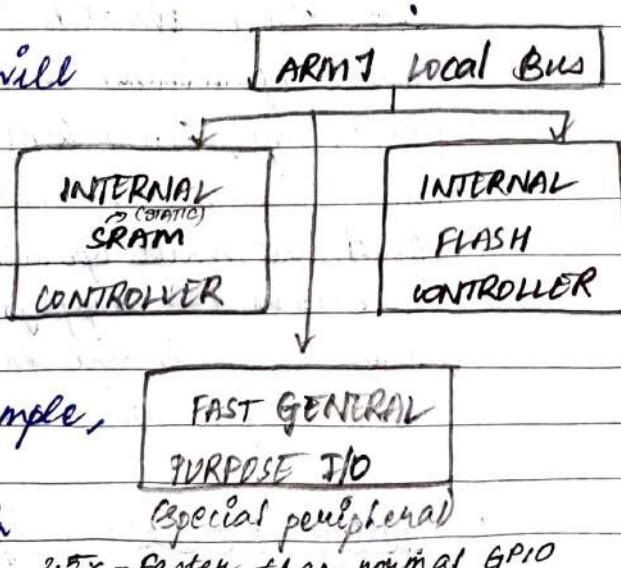
Used for connecting all advanced peripherals such as USB, DMA, VIC, etc

Used for connecting all external peripherals such as ADC, Timer, etc

⑤ AHB Bridge - Converts data from AHB Bus format to the ARM7 Processor. This Speed mismatch will be handled ~~by AHB Bridge~~.

⑥ ARM7 Local Bus - This is the fastest bus available in ARM. This will be operating on the highest speed since it is directly connected and interacting with the processor. Therefore, on comparing other buses in terms of speed, we find that ARM7 Local Bus > AHB Bus > VPB VLSI Bus.

* NOTE: Fast General Purpose I/O will be connected to ARM7 Local Bus where normal General Purpose I/O will be connected to VPB Bus.



Random Access Memory (RAM) is used for data handling. For example, when adding two numbers, the resultant sum will be stored in RAM. Whereas, the above program instruction will be stored in flash. Depending upon the serial number, we have different bytes of SRAM and FLASH.

In earlier times, programs were stored in Read-Only Memory (ROM). After that PROM (PROGRAMMABLE) was introduced where the program can be written only once. The writing process was done in the factory itself. Later, EPROM (ERASE) was invented where the program can be erased and re-written. The erasing process was done using UV-rays, through a transparent window present in the center of IC, rays were passed to the silicon chips inside the ROM. This window will be covered by the company's logo sticker to avoid light from corrupting the data inside.

The above process had a drawback. A seperate UV eraser was required for erasing the program. Moreover, it took 20 minutes to erase the data and only 1000 erase cycles can be performed. To overcome these problems, E^2 PROM was developed.

Electrically erasable Programmable Read-Only Memory (EEPROM). Here, there was no need for a seperate UV eraser, electrical signals were used. But again, there was a trouble. Data can be written only serially here, bulkee read and write operations were not possible. In order to have this advantage, FLASH ROM was developed.

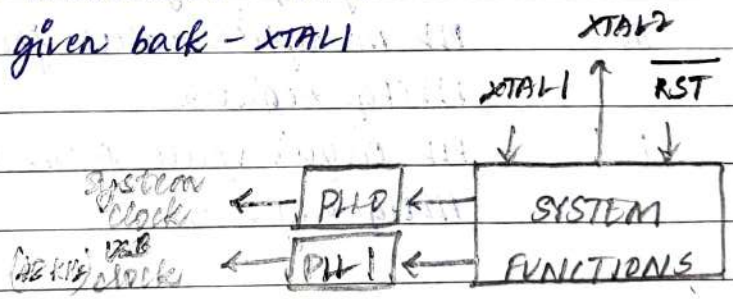
Read and write operations can be performed in FLASH with a normal +5V supply and erase cycles count upto 1,00,000 times. This is the latest development that we find in today's pen drives and hard disks.

(1) System Functions Peripheral - Handle the clock input given to the controller, a complex digital device / synchronous digital device.

Trainer kit is used to generate clock in a digital lab. 555 astable multivibrator is used to develop this clock. But, accurate clocks are not generated here owing to many passive components such as resistors, capacitors, etc. connected to it, thereby have their own tolerance and varying the final clock output.

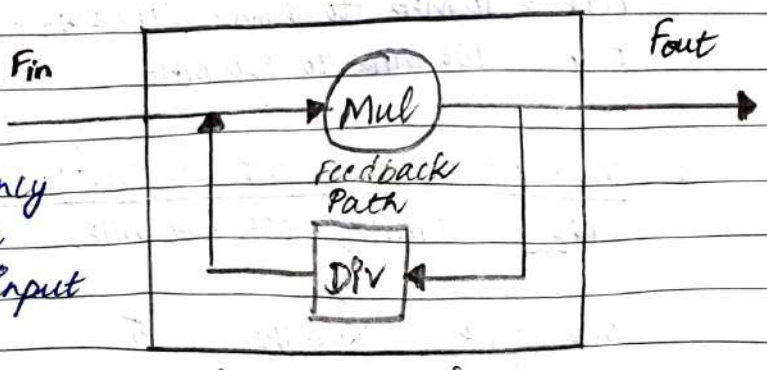
① Crystal oscillator - this is used to generate a clock in a microcontroller / microprocessor. This is connected to XTAL1 and XTAL2 of the system functions. It cannot generate its own clock but a small frequency clock is given to the crystal from the microcontroller. This induces a vibration in the crystal in its fundamental frequency (say, 12MHz).

Clock In from the controller – XTAL2
Crystal frequency given back – XTAL1



② PLL (Phase Locked Loops) - used for multiplying or dividing the clock. System clock is connected to all the peripherals and processors.

PLL is a closed loop control system to generate high frequency by multiplying with given factor to the input frequency.



Basic PLL Block Diagram

In LPC2148 microcontroller, there are 2 PLLs which provide programmable frequencies to the CPU and USB system. The input clock frequency to PLL0 and PLL1 is in the range of 10 MHz to 25 MHz only. It is multiplied up the range of 10 MHz to 60 MHz for CCLK and 48 MHz for the USB clock using Current Controlled Oscillator (CCO).

* **PLL FREQUENCY CALCULATION (CCLK):**

Table - elements determining PLL's frequency

Element	Description
F_{osc} (Frequency of Oscillator)	the frequency from the crystal oscillator/external oscillator
F_{CCO}	the frequency of the PLL current controlled oscillator
CCLK (Core ARM clock)	the PLL output frequency (also the processor clock frequency)
M (Multiplication Factor)	PLL Multiplier value from the MSEL bits in the PLLCFG register
P (Division Factor)	PLL Divider value from the PSEL bits in the PLLCFG register

$$CCLK = M \times F_{osc}$$

$$F_{CCO} = CCLK \times 2 \times P$$

F_{osc} = 10 MHz to 25 MHz

CCLK = 10 MHz to Fmax (LPC2148 - 60 MHz)

F_{CCO} = 156 MHz to 320 MHz

* **CALCULATION OF M AND P (DIVIDER) VALUE:**

(F_{osc} = 12 MHz, CCLK = 60 MHz)

$$M = \frac{CCLK}{F_{osc}} = \frac{60\ MHz}{12\ MHz} = 5$$

Range of F_{CCO} is 156 MHz to 320 MHz.

Substituting F_{CCO} = 156 MHz,

$$P = \frac{156\ MHz}{2 \times 60\ MHz} = 1.3$$

Substituting F_{CCO} = 320 MHz,

$$P = \frac{320\ MHz}{2 \times 60\ MHz} = 2.67$$

Since the value of P must be an Integer, the Integer between 1.3 and 2.7 is 2.

* PCLK FREQUENCY CALCULATION:

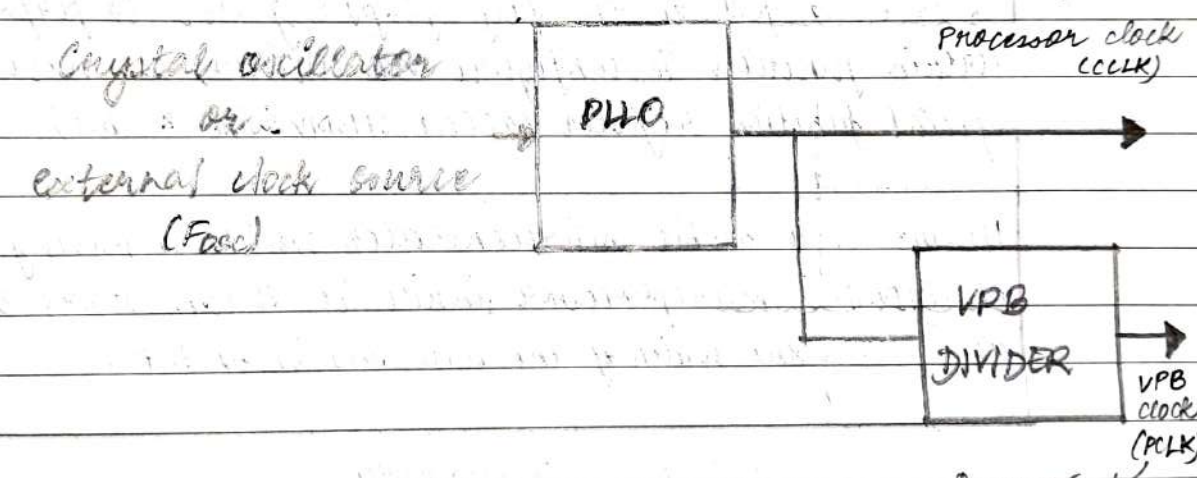


Table - VPB - VLSI Peripheral Bus

Bit	Symbol	Value	Description	Reset value
1:0	VPBDIV	00	VPB bus clock is one fourth of the processor clock. [60/4 = 15 MHz]	00
		01	VPB bus clock is the same as the processor clock. [60/1 = 60 MHz]	
		10	VPB bus clock is one half of the processor clock. [60/2 = 30 MHz]	
		11	Reserved. If this value is written to the VPBDIV register, it has no effect (the previous setting is retained. [DON'T CARE]	

PTO

7:2	-	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Q. How will you configure the VPB divider ?

A. To configure a microcontroller, we need to configure some registers (collection of flip flops). Generally in a microcontroller, we have two kinds of registers.

① General Purpose Registers - Store / Manipulate Data

② Special Function Registers - Configure the peripherals 1 or 0 is fed to the flip-flop / register to perform a certain function. To configure VPB divider, we have a special function register called VPBDIV (size - 32-bit).

LPC2148 is a 32-bit microcontroller which is having a ARM7TDMI-S microprocessor inside it. It can store 32-bit of data, i.e., the width of the data bus is 32-bit.

LECTURE 18 - LED BLINKING

GPIO stands for General Purpose Input Output Peripherals. Similar to ADC peripheral, it is used for interfacing general purpose input or output devices such as an LED (output device) or a switch (input device).

GPIO is a collection of pins, LPC2148 being a 64 pin IC microcontroller contains a package named LQFP - Low-Profile Quad Flat Package, meaning there are pins on all its four sides and the thickness of the IC is very less.

In general, GPIO is called as GPIO Ports, i.e., collection of external pins. Through these pins, we interact with the input/output devices.

There are two ports in LPC2148, namely -

1. P0 - 30 Pins
2. P1 - 16 Pins

To refer to an individual pin, we use a terminology as follows-

1. P0.0 - P0.25
2. P0.28 - P0.31
3. P1.16 - P1.31

These pins' division is based on multiple criteria such as chip manufacturer, power consumption, size, etc.

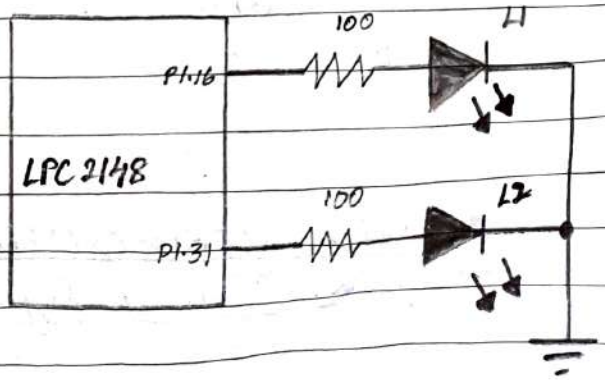
Each pin can be given multiple functionality. For example,

46 P0.16 / EINT0 / MAT0.2 / CAP0.2

GPIO Pin 16 — External Interrupt Pin 0 — Matching Pin 2 — Capture Pin 2

Therefore, four functionalities are multiplexed to a certain pin. This is done to reduce the number of pins, thereby decreasing the size, power consumption, area of cross-section, etc.

* LED BLINKING - GPIO :



* NOTE - ARM based microcontroller will be operating on +3.3 V Vcc (Voltage Common Collector).

* STEPS TO BLINK LEDs :

① Configure functionality of P1.31 to P1.16 as GPIO Port
PINSEL2 = 0 ; → All 32-bits as 0. (LED is an output device)

② Configure Direction of P1.31 to P1.16 as output Port
IODIR1 = 0xFFFFFFFF; (Digital 1)

③ Switch on all LEDs on Port1 (configure Data as Logic High)
IOSET1 = 0xFFFFFFFF; (Digital 0 [Ground])

④ Switch off all LEDs on Port1 (configure Data as Logic Low)
IOCLR1 = 0xFFFFFFFF;

Parameters -
① Direction
② Data

* STEP 1 - PINSEL2 = 0;

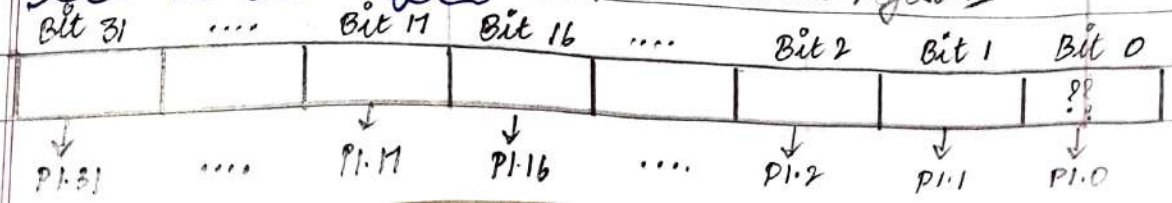
→ PINSEL Registers - [Special Function Register]
PINSEL0 - P0.0 to P0.15
PINSEL1 - P0.16 to P0.31
PINSEL2 - P1.16 to P1.31

→ PINSEL2 Register.

Bit	Symbol	Value	Function	Reset Value
1:0	-	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
2	GPIO1	0	Pins P1.36-26 are used as GPIO pins.	$\overline{P1.26}$
	DEBUG	1	Pins P1.36-26 are used as Debug port.	$\overline{RTCK}$
3	GPIO1	0	Pins 1.25-16 are used as GPIO pins.	$\overline{P1.20}$
	TRACE	1	Pins 1.25-16 are used as Trace port.	$\overline{TRACESYNC}$
31:4	-	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

* STEP 2 - IODIR1 = 0xFFFFFFFF;

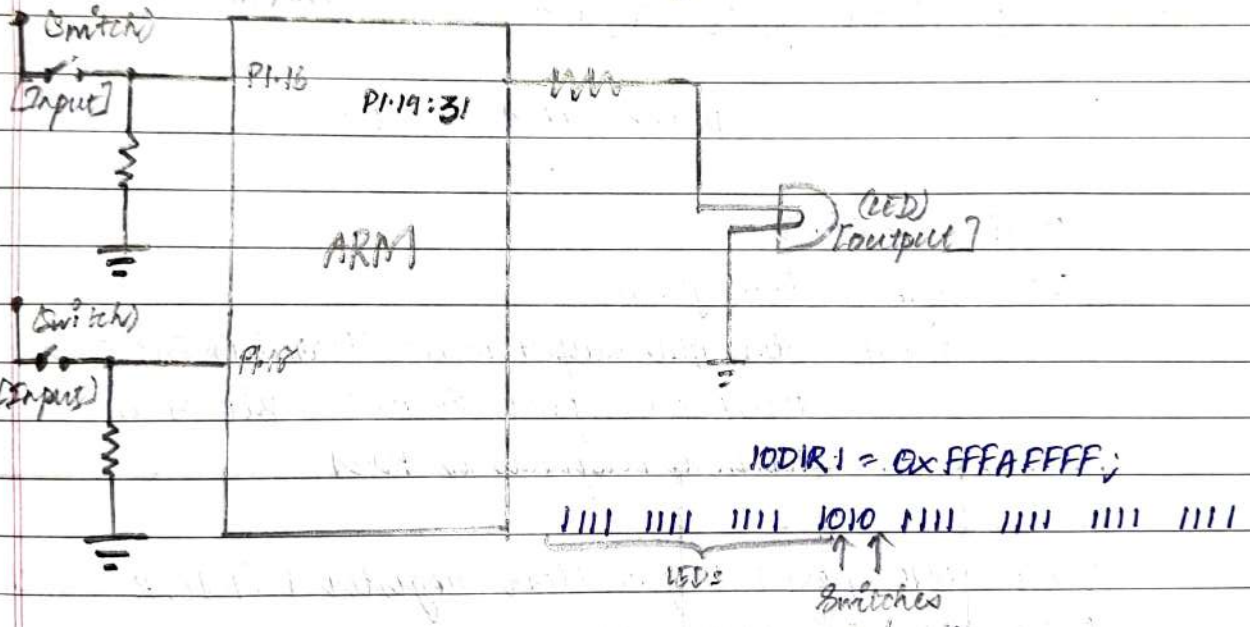
→ IODIR1 (Direction Register) - [Special Function Register]



B - Binary
D - Decimal
F - Hexadecimal

Table - GPIO port 1 Direction register (IO1DIR - address 0xE002 8018) bit description

Bit	Symbol	Value	Description	Reset value
31:0	P1xDIR		Slow GPIO Direction control bits. Bit 0 in IO1DIR controls P1.0 ... Bit 30 in IO1DIR controls P1.30.	0x0000 0000
		0	Controlled pin is input.	
		1	Controlled pin is output.	



+ STEP 3 - IO1SET1 = 0xFFFFFFFF;

→ IO1SETx (Data Register - Set) - [Special Function Register]

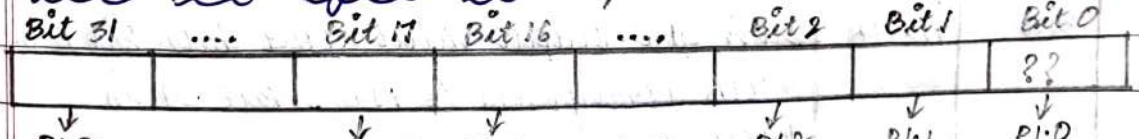


Table 1 - Set [Logic 1 - Vcc is generated - Pin ON - LED ON]

Bit	Symbol	Description	Reset value
31:0	P0xSET	Slow GPIO output value Set bits. Bit 0 in IO0SET corresponds to P0.0 ... Bit 31 in IO0SET corresponds to P0.31.	0x0000 0000

Table - GPIO port 1 output set register (IO1SET - address 0xE002 8014) bit description

Bit	Symbol	Description	Reset value
31:0	P1xSET	slow GPIO output value set bits. Bit 0 in IO1SET corresponds to P1.0 ... Bit 31 in IO1SET corresponds to P1.31.	0x0000 0000

* STEP 4 - IO1CLR1 = 0xFFFFFFFF;

→ IO1CLRx (Data Register - clear) - [Special function register]

Bit 31	....	Bit 17	Bit 16	....	Bit 2	Bit 1	Bit 0
P1.31	....	P1.17	P1.16	....	P1.2	P1.1	P1.0

Table:- 1 - clear [Logic 1 - Pin is cleared - ~~Pin~~ OFF - LED OFF]

Bit	Symbol	Description	Reset value
31:0	P0xCLR	slow GPIO output value Clear bits. Bit 0 in IO0CLR corresponds to P0.0 ... Bit 31 in IO0CLR corresponds to P0.31	0x0000 0000

Table - GPIO port 1 output clear register 1 (IO1CLR - address 0xE002 801C) bit description

Bit	Symbol	Description	Reset value
31:0	P1xCLR	slow GPIO output value clear bits. Bit 0 in IO1CLR corresponds to P1.0 ... Bit 31 in IO1CLR corresponds to P1.31	0x0000 0000

* STEPS TO BLINK LEDS :

① Configure functionality of P1.31 to P1.16 as GPIO Port.
PINSEL 2 = 0;

② Configure Direction of P1.31 to P1.16 as output port.
IO1DIR1 = 0xFFFFFFFF;

* NOTE: To reduce the brightness of the LED, you may add a resistor.

INFINITE LOOP:

(3) Switch on all LEDs on Port 1 (configure Data as Logic High)
IOSET1 = 0xFFFFFFFF;
(4) Delay [Time taken to execute one cycle is 1/15 seconds]
(5) Switch off all LEDs on Port 1 (configure Data as Logic Low)
IOCLR1 = 0xFFFFFFFF;
(6) Delay [TURN OFF → WAIT → TURN ON → WAIT]

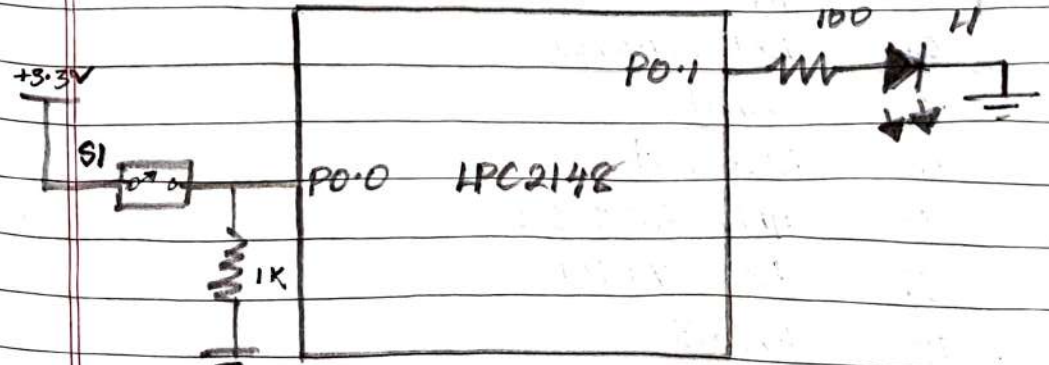
* REGISTER SUMMARY - GPIO:

	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
PINSEL0	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0
	31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16
PINSEL0	P0.15	P0.14	P0.13	P0.12	P0.11	P0.10	P0.9	P0.8
	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
PINSEL1	P0.23	P0.22	P0.21	P0.20	P0.19	P0.18	P0.17	P0.16
	31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16
PINSEL1	P0.31	P0.30	P0.29	P0.28	P0.27	P0.26	P0.25	P0.24

	15 14	13 12	11 4	3	2	1 0
PINSEL2				GPIO/TRACE	GPIO/DEBUG	SEL

LECTURE 19 - LED SWITCH

* LED CONTROL USING SWITCH - GPIO :



↳ To avoid floating set values pull down resistance

Input [Switch]

* STEPS FOR LED CONTROL USING SWITCH: → Output [LED]

① Configure functionality of P0.0 and P0.1 as GPIO Port
PINSEL0 = 0x00000000;

② Configure Direction of P0.0 as Input Pin
IO0DIR [&] = 0xFFFFFFFE; → clear a particular bit alone without affecting the other bits

③ Configure Direction of P0.1 as Output Pin
IO0DIR [|] = 0x00000002; → set a particular bit alone without affecting the other bits

Infinite loop

④ Check if Switch connected to P0.0 is Pressed
if (IO0PIN & 0x01) == 1)

⑤ If Pressed, Switch ON LED connected to P0.1
IO0SET 1 = 0x00000002;

⑥ else Switch OFF LED connected to P0.1
IO0CLR 0 1 = 0x00000002;

* STEP 1 - PINSEL0 = 0x00000000;

→ PINSEL Registers:- [Special Function Register (SFR)]
PINSEL0 - P0.0 to P0.15
PINSEL1 - P0.16 to P0.31
PINSEL2 - P1.16 to P1.31

→ PINSEL0 Register:-
Table - Pin function Select register 0 (PINSEL0 - address 0xE002 C000) bit description

Bit	Symbol	Value	Function	Reset Value
1:0	P0.0	00	GPIO Port 0.0	0
		01	TXD (UART0)	
		10	PWM1	
		11	Reserved	
3:2	P0.1	00	GPIO Port 0.0	0
		01	RxD (UART0)	
		10	PWM3	
		11	EINT0	

* STEP 2 - IO0DIR &= 0xFFFFFFFE;

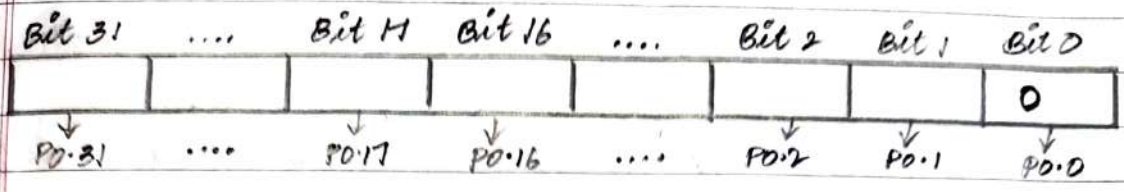
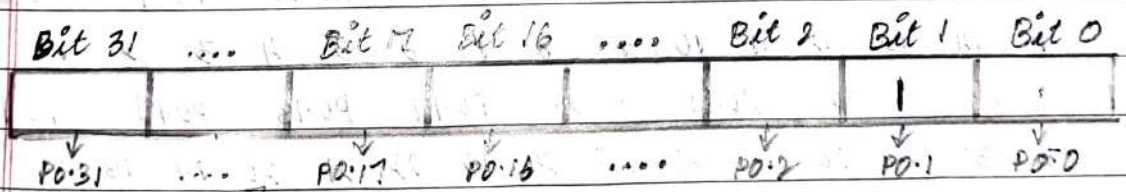


Table - GPIO port 1 Direction register (IO1DIR - address 0xE002 8018) bit description

Bit	Symbol	Value	Description	Reset Value
31:0	P1xDIR		Slow GPIO Direction control bits. Bit 0 in IO1DIR controls P1.0 ... Bit 30 in IO1DIR controls P1.30.	0x0000 0000
		0	Controlled pin is input.	
		1	Controlled pin is output.	

* STEP 3 - IO0DIR |= 0x00000002;

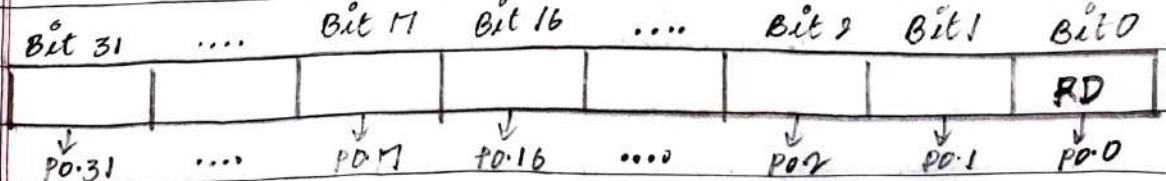


Clear: 1111 & 1101 = 1101

Set: 1101 | 0010 = 1111

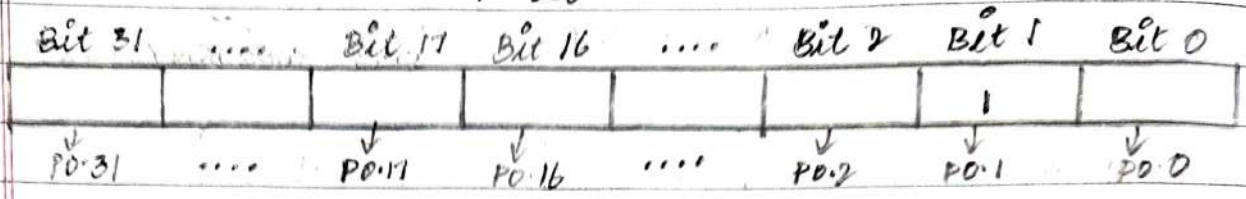
Split a Bit: 1111 & 0010 = 0010

* STEP 4 - If ((IOPIN0 & 0x01) == 1);



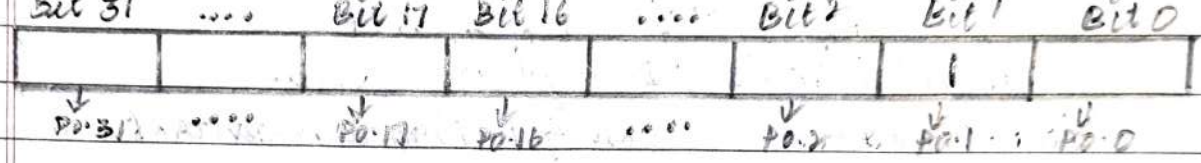
* STEP 5 - IOSET0 |= 0x00000002;

1 - Set



* STEP 6 - IOCLR0 |= 0x00000002;

1 - Clear



* REGISTER SUMMARY - GPIO:

	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
PINSEL0	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0
	31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16
PINSEL0	P0.15	P0.14	P0.13	P0.12	P0.11	P0.10	P0.9	P0.8
	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
PINSEL1	P0.23	P0.22	P0.21	P0.20	P0.19	P0.18	P0.17	P0.16
	31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16
PINSEL1	P0.31	P0.30	P0.29	P0.28	P0.27	P0.26	P0.25	P0.24

	15 14	13 12	11 4	3	2	1 0
PINSEL2				GPIO/TRACE	GPIO/DEBUG	SEL