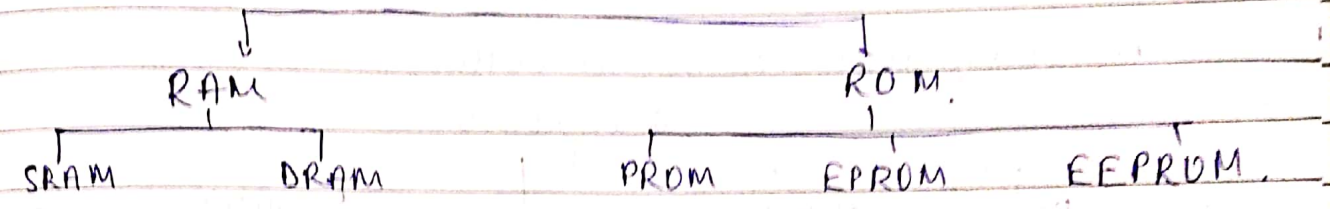
1. Classification of memory devices:



2. Difference between microprocessor & microcontroller:

Microprocessor	microcontroller
• CPU is stand alone, RAM, ROM, I/O, timer are separate	• CPU, RAM, ROM, I/O & timer are all on a single chip
• designer can decide on the amount of ROM, RAM and I/O ports	• Fix amount of on chip ROM, RAM, I/O ports
• Expansive	• For applications in which cost, power and space are critical.
• general purpose	• Single purpose.

8051 → CISC - complex instruction set computer.

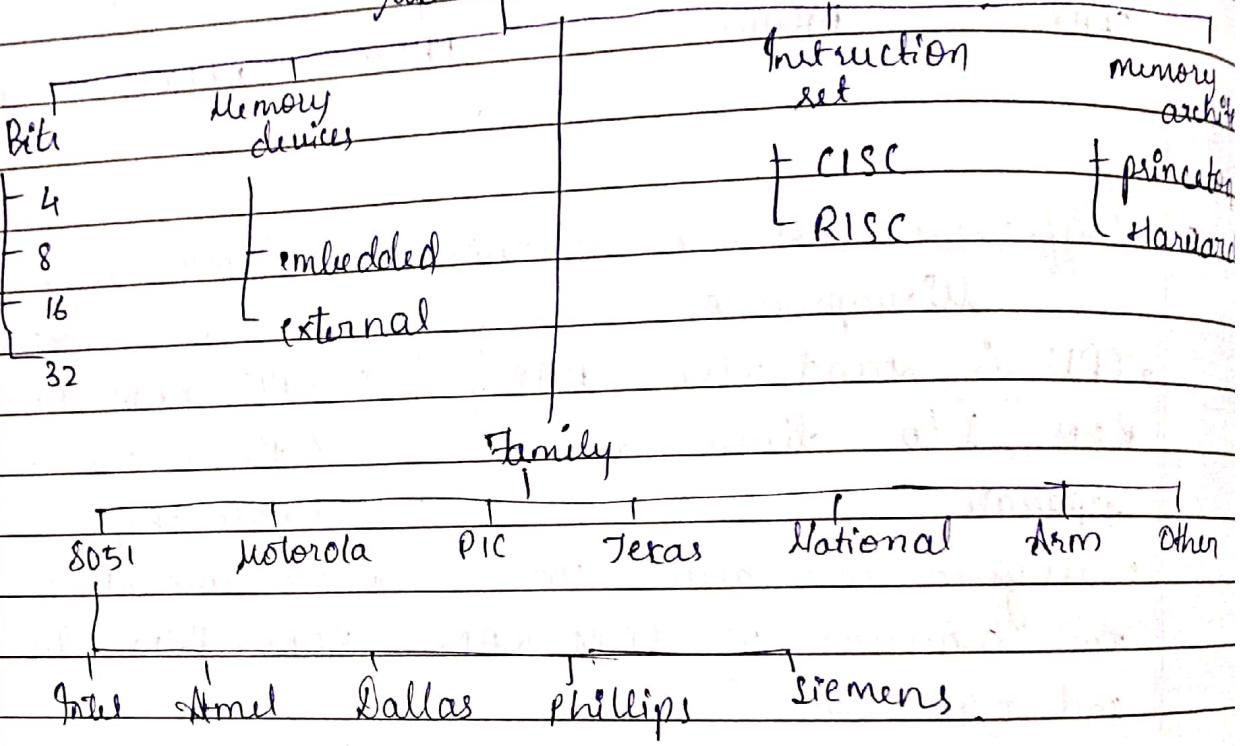
3) Difference between CISC & RISC

CISC	RISC
• Emphasis on hardware	• Emphasis on software
• Multiple instruction	• Instructions of same set with few formats
• Less registers	• More registers
• Extensive use of microprogramming	• Complexity in compiler
• Instructions take varying amount of cycle time	• Instructions take one cycle time

• pipelining is difficult • pipelining is easy

4. Classification of microcontroller

→



Types of buses

- data bus
- address bus
- control bus

Address bus → It is unidirectional. It is sent from processor to the memory

Control bus - It is unidirectional. It is sent from control unit to memory.

Data bus - It is bidirectional & sent from It can be sent from to memory or read from memory.

The operation of processor:
(In 3 stage pipeline)
fetch → decode → execute

ARM7TDMI → 32 bit microprocessor.

ARM7TDMI → 32 bit advanced RISC machine
T → thumb architecture extension.
- Uses separate instruction sets 32 bit ARM instructions & 16 bit thumb instructions.

D → debug extension
M → Enhanced multiplier
I → embedded ICE macrocell extension.

↳ Instructions in ~~RTC~~ ARM get executed in single cycle.

Features of ARM:
- 32 bit processor ⇒ 32 bit ALU, 32 bit data bus, 32 bit address bus.
- ICE (in circuit emulation)
- Registers are 32 bit.
- There are total of 37 registers — 31 general purpose
  6 special function registers.
- RISC instructions
- All most all instructions execute in single cycle.
- Each instruction ~~has~~ a fixed size of 32 bit. (32 bit long)
- Fast multiplier (external to ALU) - booth algorithm [32x8]
- Barrel shifter (to improve speed of processor)
- It operates in 2 states — ARM state - 32 bit instructions
  THUMB state - 16 bit instructions
- It operates under 7 modes:
  ↳ User (usr) - normal ARM program execution state
  ↳ FIQ (fiq) - designed to support data transfer or channel process
  ↳ IRQ (irq) - used for general purpose interrupt handling.

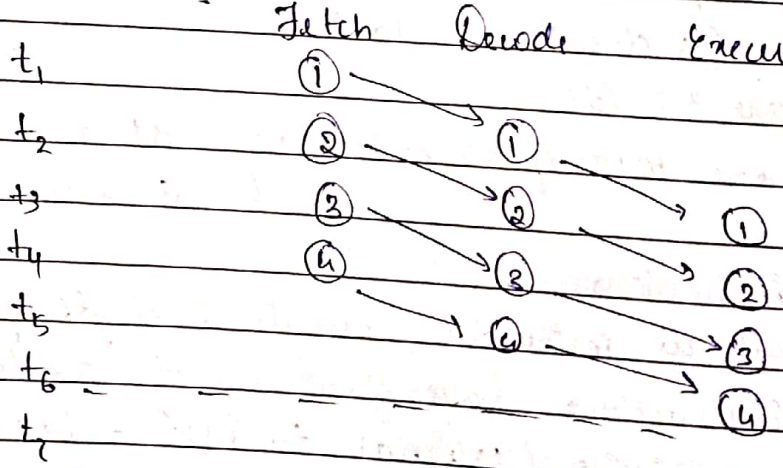
- supervisor (svc) - protected mode for the O.S
- Abort mode (abt) - Entered after a data or instruction prefe
- System (sys) - A privileged user mode for operating system
- undefined (und) - Entered when an undefined instruction is executed

Among these USER mode is unprivileged, others are privileged.

- Debug interface.
- Von Neumann architecture
- It is a 3 stage pipeline - fetch, decode, execute.
- OS & RTOS

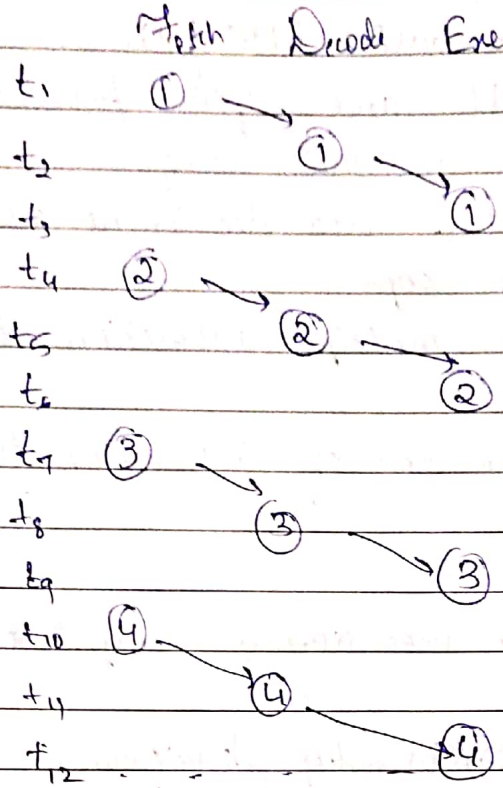
Pipelining :

Ex - executing 4 instructions with pipelining (fetch, decode, execute)



∴ 6 timecycles are required

without pipelining :



12 cycles are required.

OS & RTOS → OS stands for operating system & RTOS stands for Real time OS.

ARM → Advanced RISC Machine.

Based upon RISC architecture with enhancements to meet requirements of embedded applications

Basic risc features

- load store architecture, where data processing operations operate on register contents only
- Uniform & fixed length instructions
- Good speed / power consumption ratio
- High code density. • Instructions are 32 bit long & single length.
- Large uniform register file.

To operate in memory (perform operations) load store architecture is used which is used to load in ~~memory~~ register (read from) & store (write to) ~~memory~~ register. [Unlike in 8051 operations such as addition could be directly performed by mentioning the address]. In load store architecture ALU cannot access data from memory, they access the registers.

Enhancements to basic RISC features:

extended features -
- Control over ALU and shifter for every data processing operations to maximise their usage.
- Auto increment & auto decrement addressing modes to optimize program loops
- Load and store multiple instructions to maximise data throughput
- Conditional execution of instruction to maximize execution throughput.

throughput → ~~ex~~ execution of many instructions in a fixed time

How to Read
- Register set of ARM by Steve Furber.
- Arm organisation from Steve Furber.

Difference between simulation & emulation

Simulation	Emulation
• Decision making happens instantaneously	• Decision making happens in a series of steps
• More approximation	• More of accurate calculations

JTAG - Joint test action group.

Arm state General registers & program counter.
~~The~~ (Diagram in txt)

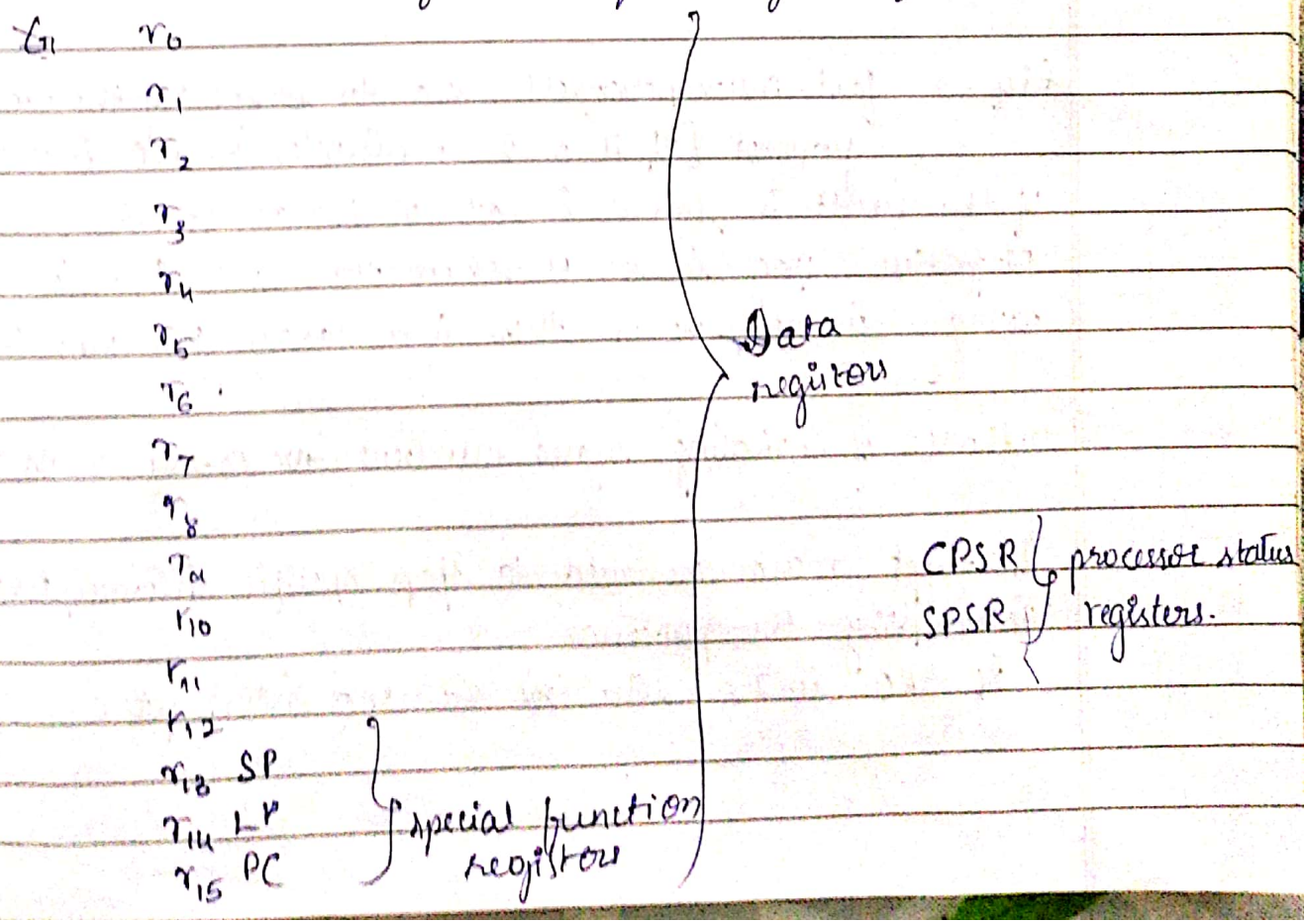
37 registers are divided as 16 in system & user, 15 registers are banked in general registers & P.C.
~~The~~ CPSR is common in all ARM State Program Status Register
There are SPSR ~~common~~ in each (of the state) column named as respectively as SPSR_fiq, SPSR_svc, SPSR_abt, SPSR_irq SPSR_und
SPSR stands for ~~saved~~ program status registers.

(CPSR - current program status registers.)
SPSR is used to store the status of the CPSR for further reference.
PC - (program counter) points to the address of the next line of program to be executed.
Registers are used for ALU operations
State registers - used to update the state

General purpose registers:

PC → program counter (r_{15})
LP → LR - link register (r_{14})
SP → stack pointer (r_{13})
CPSR → current program state register.

Stack pointer points towards the stack formed in the execution of function call in a program. It stores the head of stack in current processor mode.
LP is used to store the value of PC temporarily when the PC moves towards function definition from function call.



R13 can be used as general purpose register. But when used in abbreviation form (SP, LP, PC) performs special functions.

PC is the pointer towards the program.

control field.

31	30	29	28	27 ... 8	7	6	5	4	3	2	1	0
N	Z	C	V	Do not modify. Read as zero	I	F	T	M4	M3	M2	M1	M0

N → negative
Z → zero
C → carry
V → overflow

I → IRQ mode } interrupt masks
F → FIQ mode }
T → 0 goes to arm mode / 1 goes to thumb mode.
M → mode. (changed by using BX or BLX instructions)

5 modes → binary encoded ∴ 2^5 → 32 combinations
out of 32 only 7 combinations are valid because of security reasons.

FIQ → fast interrupt request, used to speed up the working of the processor. [If it is 1 it disables the FIQ interrupt]
If the result is zero Z is set to 1. [all 32 bits of result is zero]
Negative flag is set when the result is -ve.
Carry is set when there is a carry generated in ALU operation.

Methods of indicating signed numbers (+ve or -ve) : MSB

Interrupt masks are used to stop specific interrupt requests from interrupting the processor.
If IRQ is 1, disables the IRQ interrupt.

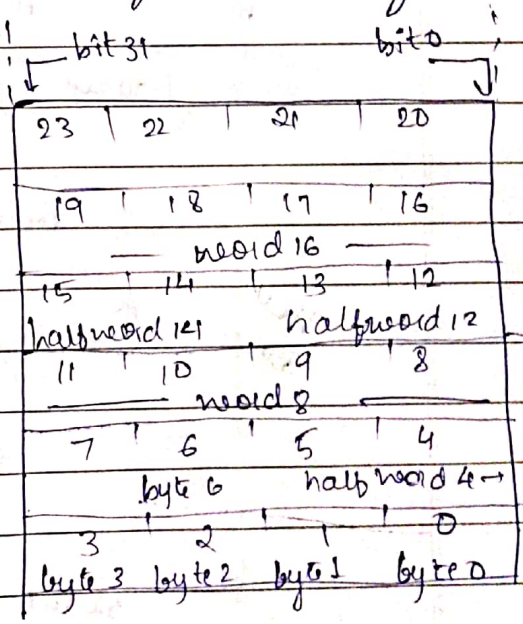
Bits 4 to 0	Mode
10000	User mode
10001	FIQ mode
10010	IRQ mode
10011	Supervisor
10111	abort
11011	Undefined
11111	system

The Memory system:

↳ The ARM system has memory state.
↔ It is viewed as a linear array of bytes numbered from 0 to $2^{32}-1$
→ Data items may be - 8 bit bytes
16 bit half words
32 bit words.

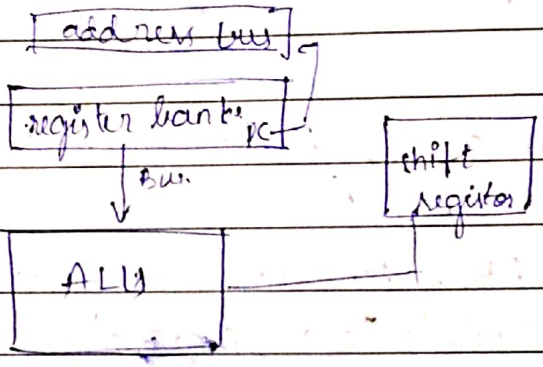
- Words are always aligned on a 4 byte boundary.

bit → binary (0 or 1)
1 nibble → 4 bits
1 byte → 8 bits
1 halfword → 16 bits
1 word → 32 bits



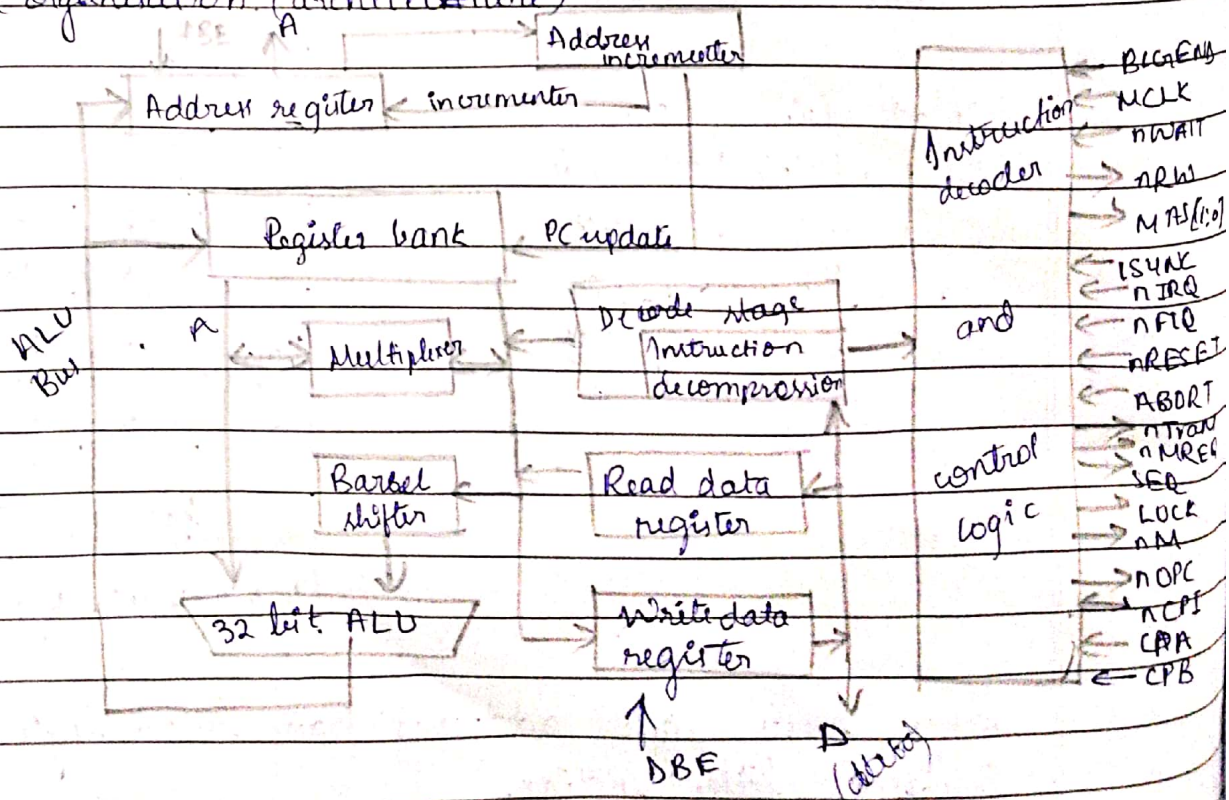
When lower byte addresses have lower byte data that it is called little endian.
When lower byte address have higher byte data, it is called big endian.

- 32 bit buses - data, address & control
- multiplier
- shift register
- ICE
- interrupts
- register banks - [PC, CPSR, SPSR]
- control & decode logic.
- ALU.



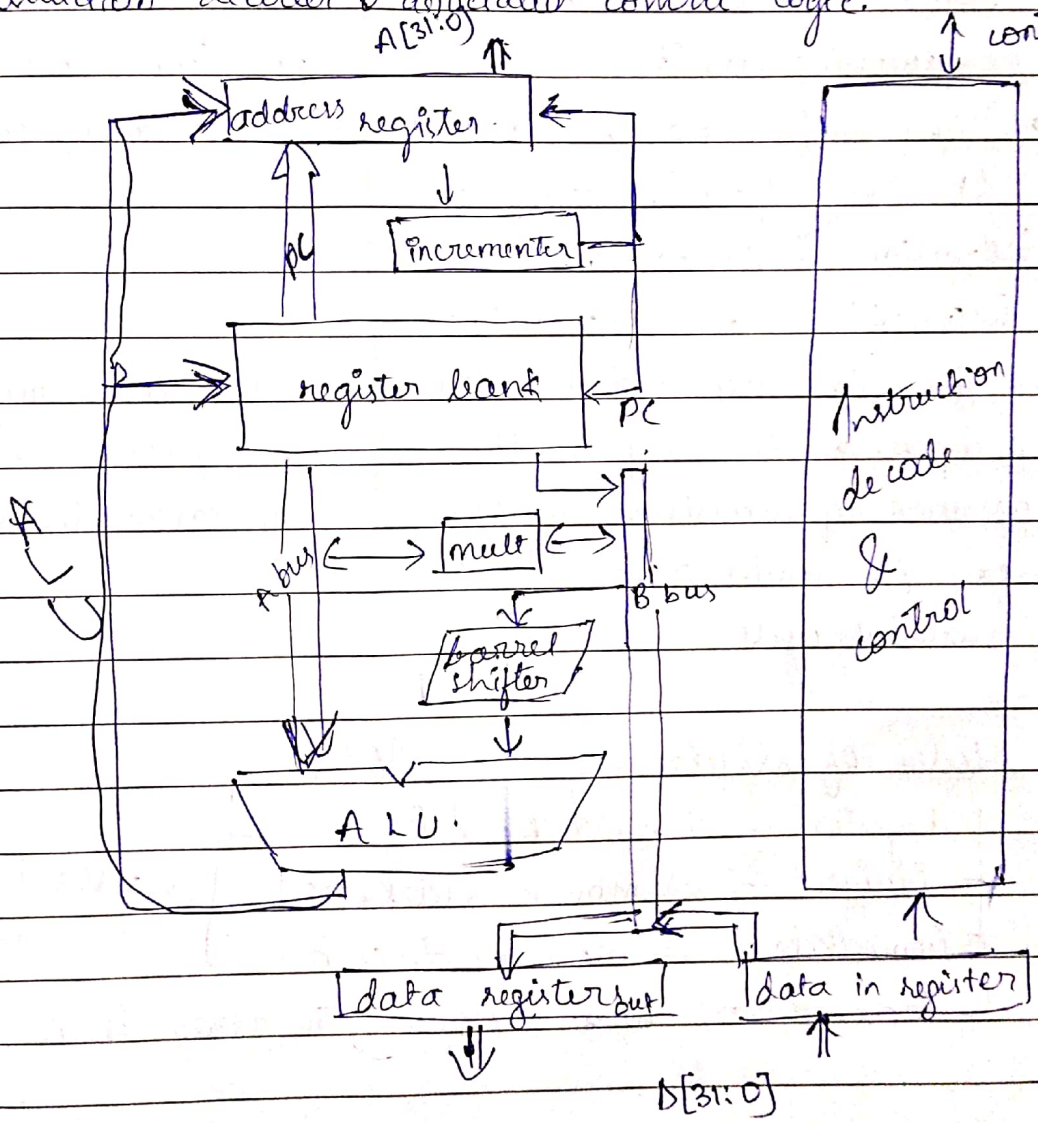
Buffers are used to correct the mismatch in the speed (speed variation) between local buses and address buses.

ARM organization (architecture)



Components :

- Register bank stores the processor state. It has 2 read ports & one write port. & additional read port & write port that gives access to r15 (program counter)
- Barrel shifter. which shifts, or rotate one operand by any no. of bits
- ALU which performs arithmatic & logic functions.
- Data register - which hold data passing from or to memory.
- Address register - It selects & holds all memory addresses & generates sequential addresses when required.
- Instruction decoder & associated control logic.



2/5 30/1

Machine cycle

- The steps that get performed by the processor getting employed in a device and all the instructions that get implemented
- Fetch, decode, execute & store
- Memory unit & central processing unit
- The steps required by CPU to fetch & execute an instruction is called an instruction ~~in~~caller cycle)
- The time required by the microprocessor to complete the operation of accessing memory or I/O devices is called machine cycle

Instruction cycle

- A process by which a computer takes an instruction given by a program then understands it and executes it from memory
- Fetch, decode, execute and run.
- Arithmatic logic unit, registers, data & memory
- ~~The time required~~
- The steps required by CPU to fetch & execute an instruction is called instruction cycle.

Modes of Addressing in ARM.

- Register - mov R1, R2
- Indirect - mov R1, @R0, x.
- Immediate - mov R1, #05h

→ used in data processing instructions.

- Indexed → kalan movie the array is accessed → not used in ARM

↓ used in d

Instructions

- Data processing (cannot access memory directly or indirectly)
  - register addressing
  - immediate addressing
- data transfer

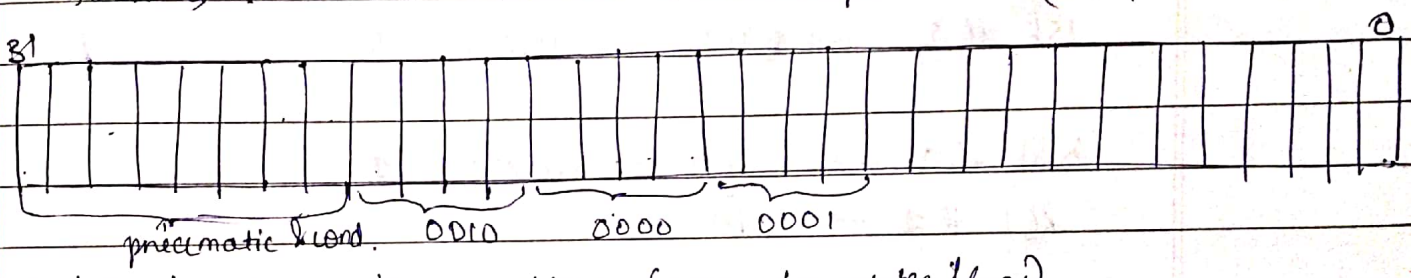
Data processing instructions:

i) move instruction (R-R transfer, R- immediat tranfer)
ii) Arithmatic instruction
iii) Logical instructions
iv) Compare
v) Jump (branch or looping)

Instruction set (32 bit)

Ex Add R_2, R_0, R_1

First 4 bits are used to hold the pneumatic (add/sub/mul/and)

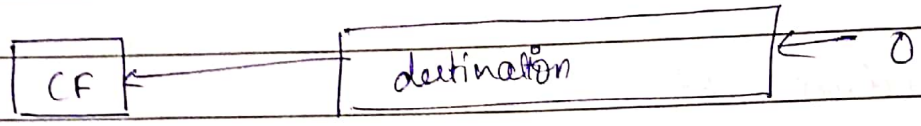


4 bits to check the condition (carry/zero-bit flags)
The next 4 bits are used to hold the address of the first operand. & similarly with the consequtive 4 bits of data.
The later bits are used to perform the operation indicate by the pneumatic.

Data can be addressed through barrel shifter also. Barrel shifter is connected to B bus. because and is only connected to second operand because only second source operand will be shifted. To speed up the execution the barrel shifter is external & all instructions can include shift instruction in their operation. [Ex Add R_2, R_0, R_1 shift #2].

The ARM doesnt have actual shift ~~register~~ instructions That is why it has barrel shifter to shift the instruction by default.

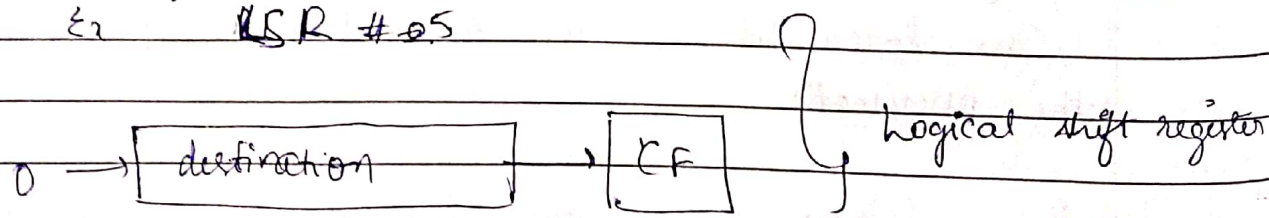
Barrel shifter - Left shift.
Shifts left by the specified amount (multiples by the power of two).
Ex - LSL #5 = Multiply by 32.
LSL - logical shift left.



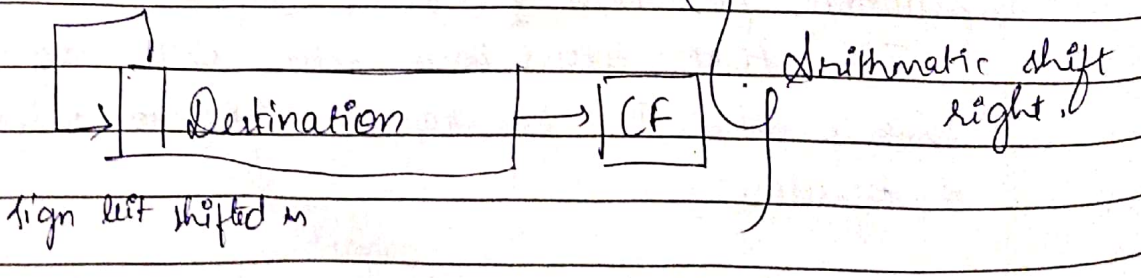
LSL #1 → 0010 = 0100 = 4.
LSL #2 → 0010 — 1000 → 8.

RSL #1 → 0010 = 0001 = 1.
RSL #2 → 0010 = 0000 = 0.

Barrel shifter - Right shift.
Shifts right by the specified amount (~~multiples~~ divides by multiples of 2).
Ex LSR #05



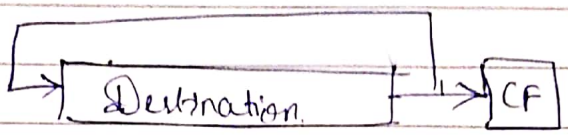
Arithmatic shift right.
Shifts right (divides by power of 2) & preserves the sign bit for 2's complement operation.



Used for signed integers.

Rotate right (ROR)

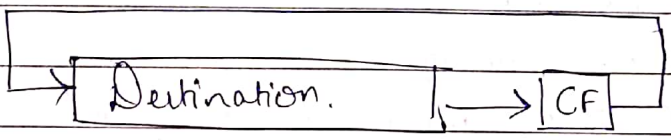
Similar to ASR but the bits wrap around as they leave the LSB & appear at the MSB.



Rotate right extended (RRX)

This operation uses the CPSR flag as a 33rd bit.

Rotates right by 1 bit (RRX # 0).



Features of ARM instruction set :-

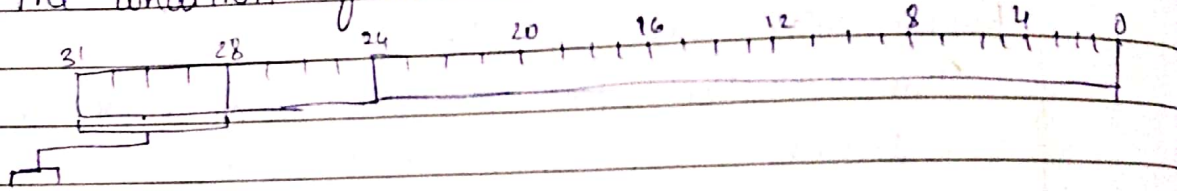
- All instructions are 32 bit long [fixed length] most instructions execute in single cycle.
- 2 sets.
  - Thumb → opcode - 16 bits
  - ARM → " → 32 bits
- All codes are conditionally executed.
- Orthogonal registers.
- Instructions get executed through 3 stage pipelining system.
- Shifting of second operand can be done through barrel shifters
- Load store architecture. — combined ALU & shifter for high speed manipulation.
- External multiplier.
- Auto indexing (memory related instructions)
- 3 operand instructions. } → most operations.
- Instruction set extension via coprocessor.

Accessing registers using ARM instructions

- All instructions can access to r14 directly.

- Most instructions also allow the use of the PC.
- Specific instructions to allow access to CPSR & SPSR
- When in privileged mode, it is also used

The condition field:



0000 = EQ - Z set (equal)
0001 = NE - Z clear (not equal)
0010 - HS/CS - C set (unsigned higher or same)
0011 = LO/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 & Z clear (unsigned higher)
1001 - LS - C clear or Z (set unsigned lower or same)
1010 - GE - N set & V set or N clear & V clear (>or=)
1011 = LT - N set & V clear or N clear & V set (<)
1100 - GT - Z clear and either N set & V set or N clear & V set (>)
1101 = LE - Z set, or N set and V clear or N clear & V set (<=)
1110 = AL - always
1111 = NV - reserved.

Eg 0x78 → 120
+0x8A → 138
Result = 102H 258

0001'0000 0010

Flags updated → N - 0
Z - 0
C - 1
V - 1

0x7A	→ 122.	Flags updated :	N - 0
+ 0x1B	27		Z - 0
Result → 0x95	149		C - 0
			V - 0

Q. Write down examples for each shift operation.

General format of instructions:

pneumonic {condition code} {S} destn. / source 1, source 2

operand 1, operand 2

S → flag updation.

```
if (a==0) ——— condition.
  { a = a+1;
  }
else
    a = a-1;
```

Flag updation.

$$\begin{array}{r} R_0 \\ +\ R_1 \\ \hline R_2 \end{array} \qquad \begin{array}{r} R_2 \\ +\ R_3 \\ +\ \bar{R} \\ \hline R_4 \end{array}$$

Add S R_2, R_1, R_0;

Add S R_4, R_2, R_3;

adc → add with carry.

& S update the flags with current values [never ~~defined~~ can choose]

There are lesser no. of basic pneumatics, and the instructions can be altered by updating flags [S]

Ex. AddEQ → this will add when Z flag is set (=1)

~~for~~ if (a==0)

{ a = a+2; → ~~MOV R1, a~~ ~~SUB~~ ;

}

else

a = a-1;

This can be written in 3 ways

```
ADDS R2,#0h          ORG 00
ADDEQ R3,R2,R1       MOV R0,#00
SUBNE R3,R2,R1       CMP R0,#00
                     ADDEQ R0,R0,#01
                     SUBNE R0,R0,#01
                     END
```

```
MOVS R1,#20h         CMP R0,#00
ADDEQ R1,R1,#1       ADDEQ R1,R0,#1;
SUBNE R1,R1,#1       SUBNE R2,R0,#1;
END
```

Q
```
if (a>0)
  { a=a+1};
else
  {a=a-1};
```

~~ADDS R0,#00~~
~~ADDMI SUBPL~~
ADDPL

→
```
CMP R0,#00        ((,Z,N,E)
ADDPL R0,R0,#1
SUBMI R0,R0,#1
```

i. Check whether a no. is positive or negative, add 1 if +ve, sub 1 if -ve.
Check whether a no. is positive odd or even add 1 if odd, sub 1 if even.

Data processing
i) Move → MOV
MOV{cond.postfix}{S} dest,source;
MVN (compliment source & put it in destination)
MVN{cond.postfix}{S} dest,source,immediate
mvn R0,R1 → R0 ← not R1 (R1' is put in R0)

MOV: R0, R1 R0 ← R1

MOV R1, R0, LSL #2 R0 ← R0 * 4.

MOV R1, R0, #1 R0 ← R1 * 00000001.

MVN:

MVN R1, R0 ; R1 ← NOT R0.

MVN R1, R0, LSL #2 R1 ← NOT (R0 * 4).

MVN R0, #4 ; R0 ← NOT 4.

2) Arithmatic & logical
- ADD
- adc
- sub.
- sbc
- RSB (reverse subtract)
- and RSC.
- ORR
- eor

write 3 addressing modes for each.

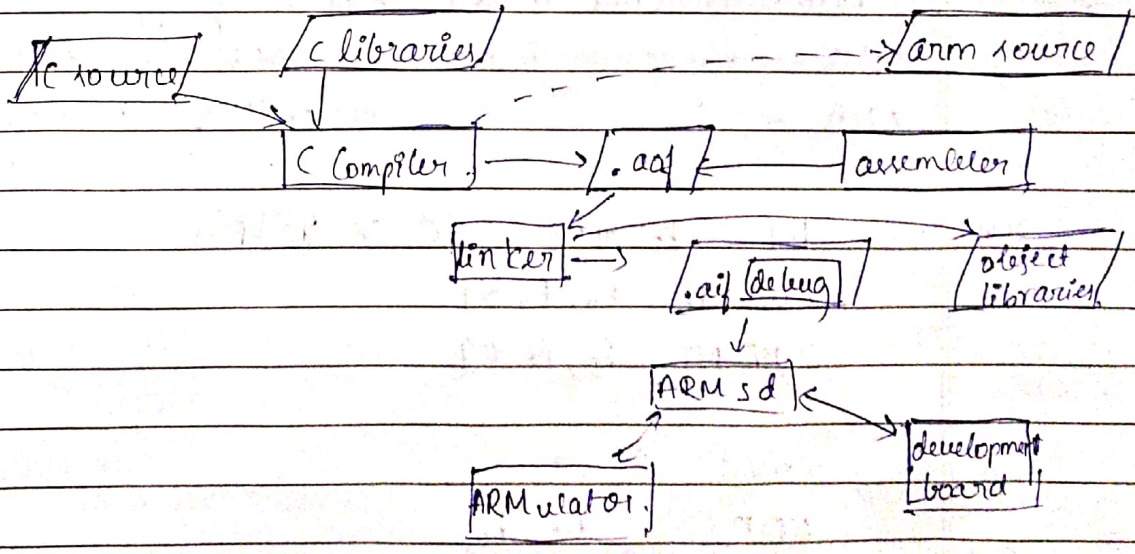
ADD:

ADD {cond. postfix} {S} dest, op1, op2;

(register) (reg, immediate, barrel shifter)

Class

ARM Development Tools:



Full form of aof, aif, difference between machine & i. cycle.

The library is included (stdio.h) in the compiler, i.e., the standard functions in the library are extended to the program to be executed in the compiler. (They become a part of the program).

Assembler directives:

They are the directions given that guide the assembler

Some directives:

AREA - It defines a region in the memory (can be named) [Eg 1]

CODE - It is used to select between (data & code memory) [Ex - Named as Pg1]

Read only → used to select the type of file.

```
AREA Pg1, CODE, READONLY
ENTRY
    : MOV R0, #0x0A
    : MOV R1, #0x0B
    : ADD R2, R1, R0
END
```

Eg Area with CODE memory & Read only type is named as Pg1 in this case

ENTRY - used to indicate the beginning of the code

END - " " " the end of the program code.

HW:

1. CMP R0, #00 (C, Z, N, F)
   ADD PL R0, R0, #1
   SUBB MI R0, R0, #2

2. MOVS R2, 0.
   RORS R2, R2
   ADD HS R3, R3, #01
   ADDLO R2, R2, #01.

⊞ NXP

↳ LPC 2148

Code for +ve, -ve:

```
AREA PG1, CODE, READONLY
ENTRY
MOV R0, #0X0A
cmp R0, #0x00
ADDPL R0, R1, #1
stop b stop
END
```

OR

```
AREA PG1, CODE, READONLY
ENTRY
ldr R0, = 0x11112222
cmp R0, #0x00
ADDPL R1, R1, #1
SUBMI R1, R1, #1.
Stop b stop
END.
```

LDR is used instead of MOV because MOV moves 8 bit of ~~immediate~~ data. (that is because ~~of~~ Only 8 bit is ~~left~~ in Opcode in the instruction set) (unsigned)

Dataprocessing continued:

iii) Compare
- CMP
- CMN
- TST
- TEQ.

iv) Jump
- B
- PL
- BLX

Add 2 32 bit no, add 2 64 bit no., add series of 5 32 bit no.
count 0's & 1's in 8 bit no.
check whether a number is greater than a no. (32 bit)
" " " " less than a no. (32 bit)

Add 2 32 bit numbers.

```
AREA A32, CODE, READONLY
ENTRY
LDR R0, =0X11112222
LDR R1, =0X88885555
ADD R2, R1, R0
Stop B stop
END.
```

Add 2 64 bit numbers

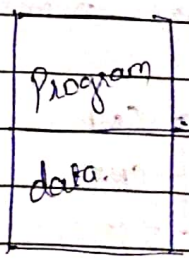
```
AREA A64, CODE, READONLY
```

19 (11112222 33334444)
12 (5555 6666 2222 3333)
21
r_5 r_4

R_2 11112222 88884444 R_0
R_3 55556666 88883333 R_1
66668889 11107777 R_4
R_5

Object Library:
It is a collection of object modules in a single file in a format that allows selective extraction by linker of those modules that are needed when linking a program.

Memory:
Program memory → read only and once written it cannot overwrite.



Memory handling Instructions:

LDR dstn., =constant —(i)
pointer —(ii)
variable name —(iii)

Ex. (i) LDR, R0, =0xAABBCCDD

(ii) LDR R1, [R0] → here R0 is used as pointer.

LDRH → loads 2 bytes into register → 16 bit.
(H = half)
LDRB → loads 1 byte into register → 8 bit
LDR → 32 bit (4 bytes) into register
Pointers are used to access a set of data (stored continuously)

(iii) LDR, R0, a.
where a=5 is initialized before (a is the variable).

DCD → directive that indicates the datatype that the variable is holding [Declare constant doubleword]
- → 32 bit (DCD)
- → 16 bit (DCW) [declare constant halfword]
- → 8 bit (DCB) [declare constant byte].

Ex a DCD 0xAABBCCDD [DCD → 32 bit]
a DCW 0xAABB [DCW → 16 bit]
a DCB 0xAA [DCB → 8 bit]

Write a program to add 2 → 32 bit numbers by declaring variables a & b.

AREA ENTRY
a DCD 0x11112222 } after stop B stop.
b DCD 0x33334444 }
LDR R0, a
LDR R1, b
ADD R2, R1, R0
stop B stop
END.

```
AREA A32, CODE, READONLY
ENTRY
LDR R0, a
LDR R1, b
ADD R2, R1, R0
stop b stop
a dcd 0x11112222
b dcd 0x33334444
end.
```

LDR R0, =a [loads the starting address of a to R0]. (not the value of a.)

```
LDR R1, [R0]
```

```
AREA A5, CODE, READONLY
ENTRY
LDR R0, =a.
LDR R1, [R0], #0x04.
LDR R2, [R0], #0x04
LDR R3, [R0], #0x04
LDR R4, [R0], #0x04
LDR R5, [R0]
ADD R6, R1, R2.
ADC R7, R6, R3
ADC R8, R7, R4
ADC R9, R8, R5.
StopB stop
a dcd 0x11223344, 0x22334455, 0x33445566, 0x11336644
0x44553322
end.
```

#0x04 increments the pointer by 4 because of autoindexing.

Find the sum of even & odd numbers in an array of 10 32 bit numbers.

Find the no. of positive & negative numbers in an array of 10 32 bit no.s.

To check whether a number is greater than the other

```
AREA GRE; CODE, READONLY
ENTRY
MOV R0, #05
MOV R1, #15
~~MOV R~~
CMP R0, R1
MOVGT R2, #01
MOV LT R2, #00
STOP B STOP
END
```

To check whether a no. is less than the other

```
AREA LESS, CODE, READONLY
ENTRY
MOV - R0, #05
MOV R1, #15
CMP R0, R1
MOV LT R2, #01
MOV GT R2, #00
STOP B STOP
END
```

For 32 bit

```
LDR - R0, = 0X11223344
LDR R1, = 0X55663311
```

To count no. of 1's & 0's, in number

```
AREA COUNT, CODE, READONLY
ENTRY
MOV R0, #0X23
MOV R1, #0X8
Label MOVS R0, R0, LSR #1
```

Add CS R2, R2, #1
ADD CC R3, R3 #1
sub r1, r1, #1.
cmp r1, #00
bne lable
end.

Branch if not equal jump back if & only if zero flag is clear.

SUB

```
MOV R1 ,#0X04
MOV R2 ,#0X01
SUBS R0, R1, R2
```

CMP

```
MOV R1 #0X01
MOV R2 ,#0X04
CMP R1, R2.
```

RSB

```
MOV R1, #0X02
MOV R2 ,#0X01
RSBS R0, R1, R2
```

EOR

```
MOV R1, #0X01
MOV R2, #0X04
EOR R1, R2, R4
```

ORR

```
MOV R1, #0X02
MOV R2, #0X03
ORR R0, R1, R2.
```

To add 5 32 bit numbers using branch loop

→

```
AREA ADD5, CODE, READONLY.
ENTRY MOV R3, #0X00 ; R3 stores final sum
LABEL: MOV R0, #0X05
       LDR R1, =a
LABEL: LDR R2, [R1], #0X04
       ADDS R3, R3, R2.
       SUBS R0, R0, #0X01
       CMP R0, #0X00
       BNE LABEL.
```

stop B stop

a dcd 0x01223344, 0x11224433, 0x33221144, 0x11334422
0xd2334455.

end

The s in the subs instruction overwrites the carry flag & gives wrong answer.

To avoid this another loop can be used

```
area A32, code, readonly.
entry
    mov  r0, #0x05
    mov  r3, #0x00
    ldr  r1, =a
LABLE ldr  r2, [r1], #0x04
    adds r3, r3, r2
    beq  carry
    sub  r0, r0, #0x01
    cmp  r0, #0x00
    bne  LABLE
carry adc r3, r3, r2
    subs r0, r0, #0x01
    cmp  r0, #0x00
    bne LABLE
stop B stop
a dcd 0x01223344, ____, ____, ____
end
```

OR

```
area A32, code, readonly
entry
    mov r0, #0x05
    mov r3, #0x00
    ldr
LABLE ldr r2, [r1], #0x04
    adds r3, r3, r2
    addcs r4, r4, 0x01
    sub r0, r0, #0x01
    cmp r0, #0x00
    bne LABLE
stop bstop
a dcd ____, ____, ____, ____
end
```

HW

1. Odd-even, +ve-ve max-min, 1's & 0's; mul div, add/sub
2. to add two array elements (one to one) and store the result in 3rd array
3. Develop a code to find minimum of N numbers.

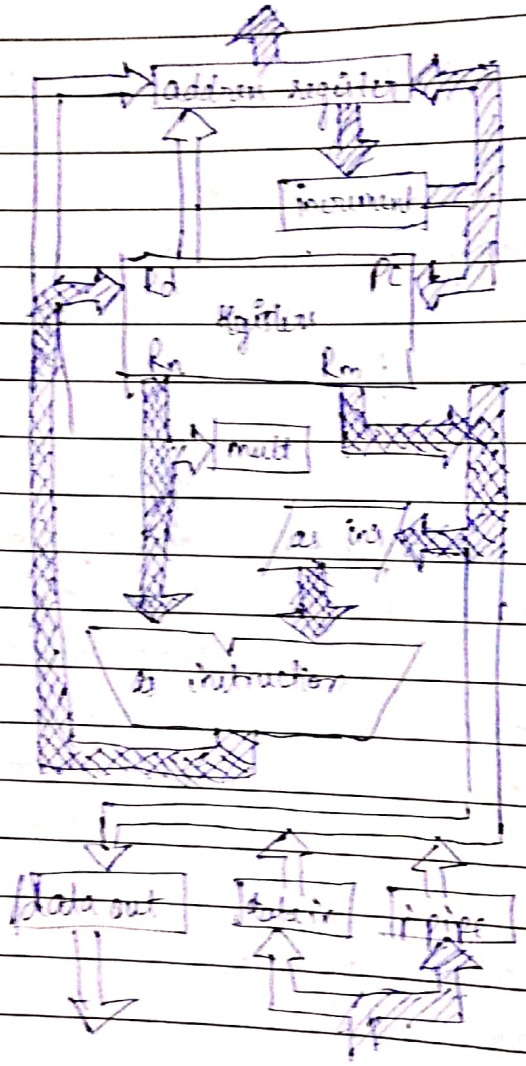
4 3+3+3+3

3×4 3 4+4+4.

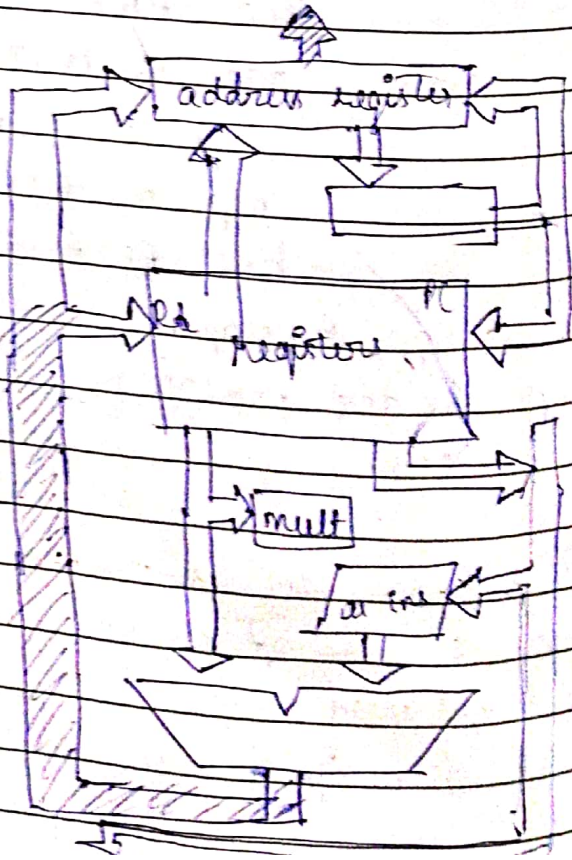
Code for division (division using subtraction)

ASEP Logic Data processing instruction

Data path activity.



add R_2, R_0, R_1



add $R_1, R_0 \neq 0$?

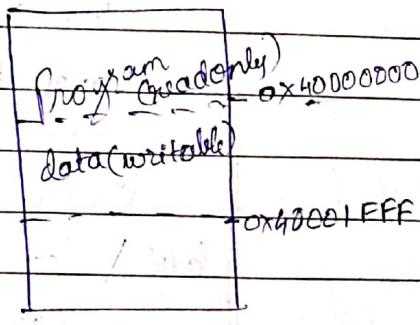
STR → store instruction.

1- STR source, destination (destination → memory pointer/address)

The writable memory area is 8kB.

1k = 10 b. [∵ $2^{10} = 1024$.]

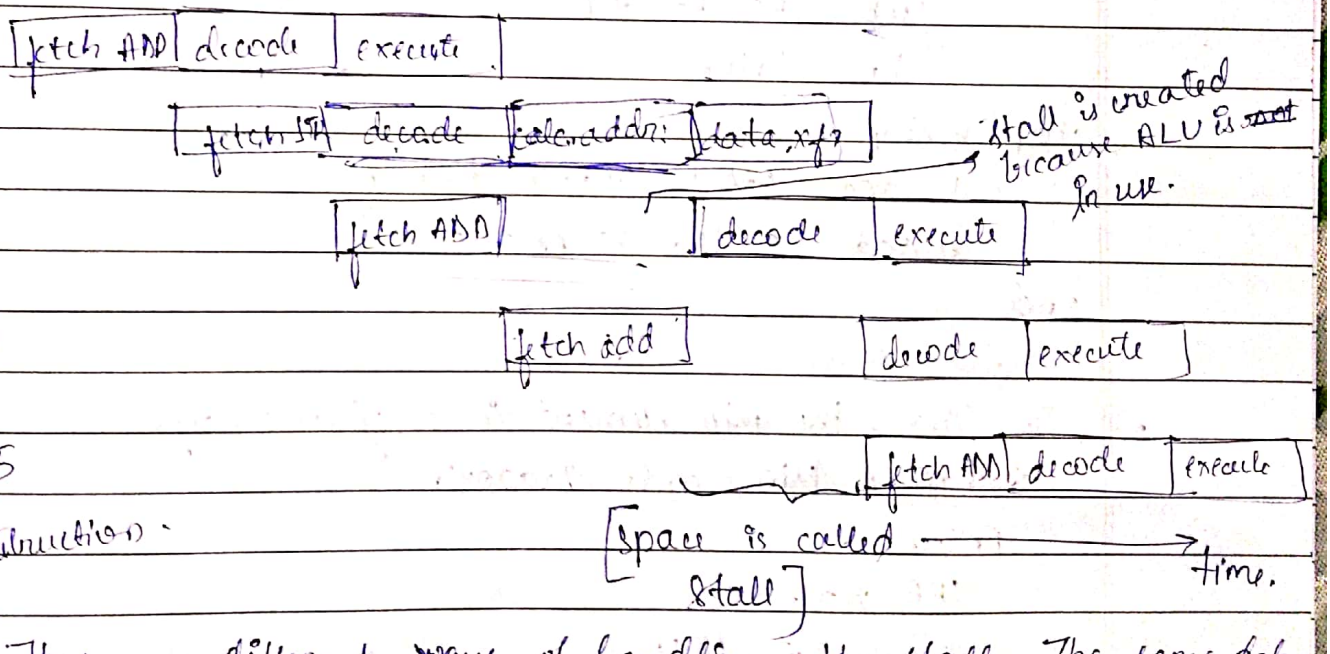
∴ 8kb = 13 b. [∵ $2^{13} = 8192$]



strb R_5, [R_0], #01

40000000

ARM multicycle instruction pipeline.



There are different ways of handling the stall. The same data is fetched in the next cycle if the data is not flushed.

Q) Str instruction. Str R_5 [R_4, #4]

pre increments the data.

(Decoder generates control signals.)

1. Add 2 array elements & store the answer in an array.

Program:

```
AREA add_array, code, readonly
ENTRY
    ldr R0, =a
    ldr R1, =b
    ldr R2, =0x40000000
    mov R3, #3
lab ldr R4, [R0], #4
    ldr R5, [R1], #4
    add R6, R4, R5
    str R6, [R2], #4
    subs R3, R3, #1
    bne lab
stop b stop
    a dcd 0xA1, 0xFE9, 0x56F
    b dcd 0x116, 0xBAC, 0x74C
end
```

2. To find the min element in a given array.

Program:

```
AREA Min, code, readonly
ENTRY
    LDR R0, =a
    MOV R5, #4
    LDR R2, [R0], #4
CMPR LDR R4, [R0], #4
    CMP R2, R4
    MOVCS R2, R4
    SUBS R5, R5, #1
    BNE CMPR
STOP B STOP
    a dcd 0x11223344, 0x22334455, 0x12345678
    END
```

3. Code to find the sum of even no. in an array of 10 ele

```
AREA EVEN, CODE, READONLY
ENTRY
MOV R3, #0x00   // R3 is used to store the result
MOV R5, #0x0A   // R5 is used as count
LDR R0, =a
even LDR R2, [R0], #4
MOVS R6, R2, ROR, #01
ADDCC R3, R3, R2
SUBS R5, R5, #1
BNE EVEN
STOP B STOP
a dcd 0x11223344, ____, ____, ____,
END
```

4. To find sum of ~~negative~~ odd numbers in a given array of 10 elements

```
AREA SUMODD, CODE, READONLY
ENTRY
MOV R3, #0x00   // R3 is used to store the result
MOV R5, #0x0A   // R5 is used as count
LDR R0, #=a
odd LDR R2, [R0], #4
MOVS R6, R2, ROR #01
ADDCS R3, R3, R2
SUBS R5, R5, #1
BNE ODD
Stop B Stop
a dcd 0x098, 0xA1, 0x32, 0x33, 0x41..
end
```

5. To find the sum of positive numbers in a given array

```
AREA POS, CODE, READONLY
ENTRY
```

```
        LDR  R0,=a
        MOV  R5,#0X0A
        MOV  R6,#00  // R6 is used to store the result.
POS:    LDR  R2,[R0],#4
        MOV  R3,R2
        MOVS R2,R2,LSL #1
        ADD CC R6,R6,R3
        SUBS  R5,R5,#1
        BNE  POS
Stop B Stop
        a dcd 0x2233,
        end
```

6. To find the sum of negative numbers in a given array of 10 n

```
        AREA NEGATIVE, CODE, READONLY
        ENTRY
        LDR R0,=a   //R0 is used as a pointer to array
        MOV R5,#0A
        MOV R6,#00  // stores the result
NEG     LDR R2,[R0],#4
        MOV R3,R2
        MOVS R2,R2,LSL,#1
        ADD CS R6,R6,R3
        SUBS R5,R5,#1
        BNE NEG
STOP B STOP
        a dcd 0xfe
        end
```

7. Code for division using subtraction

```
        AREA DIV_SUB, CODE, READONLY
        ENTRY
```

```
        MOV R5, #15
        MOV R6, #3
        CMP R6, #0
        BEQ EN
        CMP R6, R5
        MOV PL R6, #1
        BPL EN
DIV     SUB R7, R5, R6
        ADD R1, R1, #1
        CMP R7, R6
        MOVPL R5, R7
        BPL DIV
END
STOP    B STOP
        END
```

8. To sort a given array in ascending & descending order.

→

```
        AREA ASCEND, CODE, READONLY
        ENTRY
        MOV R0, #5
OUT     LDR R5, =0X40000000
        ADD R6, R5, #1
        MOV R3, #4
IN      LDRB R1, [R5]
        LDRB R2, [R6]
        CMP R1, R2
        BCC LOP
        MOV R4, R2
        MOV R2, R1
        MOV R1, R4
LOP     STRB R1, [R5]
        STRB R2, [R6]
```

```
        ADD  R5,R5,#1
        ADD  R6,R6,#1
        SUBS R3,R3,#1
        BNE  IN
        SUBS R0,R0,#1
        BNE  OUT
STOP    B    STOP
        END.
```

9 To arrange an array in descending order

```
        AREA DESCEND, CODE, READONLY
        ENTRY
        MOV  R0,#5
OUT     LDR  R5,=0X40000000
        ADD  R6,R5,#1
        MOV  R3,#4
IN      LDRB R1,[R5]
        LDRB R2,[R6]
        CMP  R1,R2
        BCS  LOP
        MOV  R4,R2
        MOV  R2,R1
        MOV  R1,R4
LOP     STRB R1,[R5]
        STRB R2,[R6]
        ADD  R5,R5,#1
        ADD  R6,R6,#1
        SUBS R3,R3,#1
        BNE  IN
        SUBS R0,R0,#1
        BNE  OUT
STOP    B    STOP
        END
```

Q: ~~Code to~~ Code to sort array in half ascending & half descending order.

```
        AREA HALF, CODE, READONLY
        ENTRY
        MOV R0, #8
        MOV R9, #2
        MOV R10, #0
OUT1    LDR R5, #0x40000000
        ADD R6, R5, #1
        MOV R3, #2
IN1     LDRB R1, [R5]
        LDRB R2, [R6]
        CMP R1, R2
        BCC LOP
        MOV R4, R2
        MOV R2, R1
        MOV R1, R4
LOP     STRB R1, [R5]
        STRB R2, [R6]
        ADD R5, R5, #1
        ADD R6, R6, #1
        SUBS R3, R3, #1
        BNE IN1
        SUBS R0, R0, #1
        BNE OUT1
        BEQ OUT2
OUT2    LDR R5, =0x40800003
        ADD R6, R5, #1
        MOV R12, #1
IN2     LDRB R1, [R5]
        LDRB R2, [R6]
        CMP R1, R2
```

```
        BCS LOP 2
        MOV R4, R2
        MOV R2, R1
        MOV R1, R4
LOP 2   STRB R1, [R5]
        STRB R2, [R6]
        ADD R5, R5, #1
        ADD R6, R6, #1
        SUBS R12, R12, #1
        BNE IN2
        SUBS R9, R9, #1
        BNE OUT2
STOP B STOP
        END
```

To generate first 10 numbers of fibonacci series

```
AREA FIBO, CODE, READONLY
ENTRY
    MOV R10, #10
    LDR R3, =0x40000000
    MOV R0, #0
    SUB R1, R0, #1
    MOV R2, #1
    ADD R6, R2, R1
    STR R6, [R3], #4
FIB ADD R6, R2, R0
    STR R6, [R3], #4
    MOV R2, R0
    MOV R0, R6
    SUBS R10, R10, #1
    BNE FIB
STOP B STOP
    END.
```

To generate first 10 even numbers.

```
AREA EVEN, CODE, READONLY
ENTRY
MOV R10, #10
ldr R3, =0x40000000
mov R4, #0
label add R4, r4, #2
str R4, [r3], #4
sub R10, R10, #1
BNE LABLE
```

31. Develop a code to find

i) R0 - 5R1

ii) R3 = R0 R4

iii) R7 = 15R3

iv) if (R0 > R1)

{ $R_2 = R_0 + 4R_1$ }

else

{ $R_2 = R_0 + 7R_1$ }

if { $R_5 = !R_6$ }

{ $Y = AB + BC + AC$ }

else

{ Y = A nor B nor C }

32. Given random array of 10 nos. separate even numbers.

[string dcb "BVBCET",0] → null terminated string.

[string dcb "BVBCET", CR
CR equ 0x0d] → string terminated with CR=0x0d

[string dcb "BVBCET", $] → string terminating with $

Store instruction occurs in 2 phases
- calculation of address. Ex [R2, #4]
- store data & auto indexing.

In store instruction immediate data has a size of 12 byte.

STR R0, [R2, #4]! → writeback

writeback helps in updating the address to the autoindexed value. (It only applies to pre-indexing)

Not using (!) in preindexing keeps the pointer pointing towards R2 and repeats the same address.

PC is involved in execution of branch instruction. And the PC ~~poi~~ is moved to ALU because it is used to calculate the size of jump in case of loop. The jump size is restricted to 2B.

~~Word align the~~

R14 → link register [used to temporarily store the value of PC]

BL → branch & link for function call

BX → branch & exchange for different operation.

before branching it gets word aligned. ~~Word aligning~~.

word align → addresses having multiples of 4.

Sum of odd & even parity.

LDR R0, =AB

MOV R12, =0x04

MOV R11, =0x08

```
loop    LDRB  R1 [R0], #0x01
        MOV   R2 , R1
ONES    MOVS  R1, R1, LSR, #0x01
        ADDCS R3, R3 , #0x01
        SUBS  R11, R11, #0x01
        BNE   ONES
        MOV R11, #0x08
        MOV R4, R3
        ANDS R6, R4, #0x01
        ADDEQ R5, R5, R2.
        ADDNE R7, R7, R2.
        SUBS R12, R12, #0x01
        MOV R3, #0x00
        LDR . R8, = 0x40000000
        LDR . R9, 0x40000004
        STRB R5, [R8]
        STRB R7, [R9]
stop    b stop
        A DCB 0x01, . . . , 0x0A
end.
```

(iXS can be done as RSB R1, R0, R0, LSL #3.