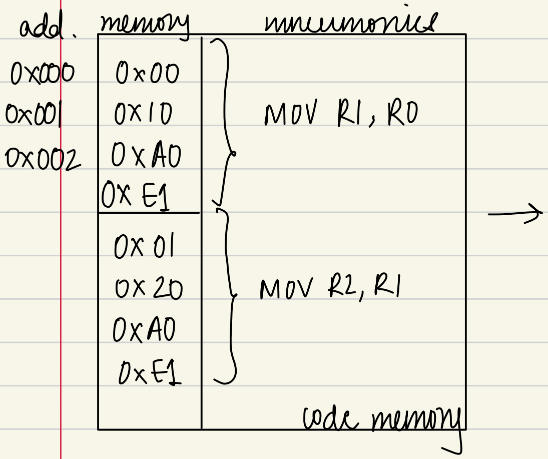
Addressing Modes

1 Assembly Program

· assembly level statements are called mnemonics
↑
have a one-to-one correspondence with a binary pattern (aka machine code) that is directly understood by CPU



in ARM we have only 15 registers

```
CMP   R0, R2
BLE   Else
MOV   R1 ,R0  ; b=a
B     Skip
Else  MOV  R1, R2 ; b=c
Skip :
```

ARM assembly program eq
↑
hardware-dependent.
we address the architecture of the processor directly.

```
if (a>c)
   b=a;
else
   b=c;
```
↑
c program equivalent. hardware-independent, HLL

more flexible
can be converted into machine code for any processor
by an assembler

2. Assembly Language

- more efficient code

- have a faster execution speed
- gives you power over choosing the memory space rather than leaving it up to the compiler enabling a more compact program size

- can have a hybrid mix : C, assembly.

- exploit a particular feature of the processor's ISA

→ HLL may not exploit optimized instructions
→ directly addressing modes and features available set arch to produce efficient code

- used when :

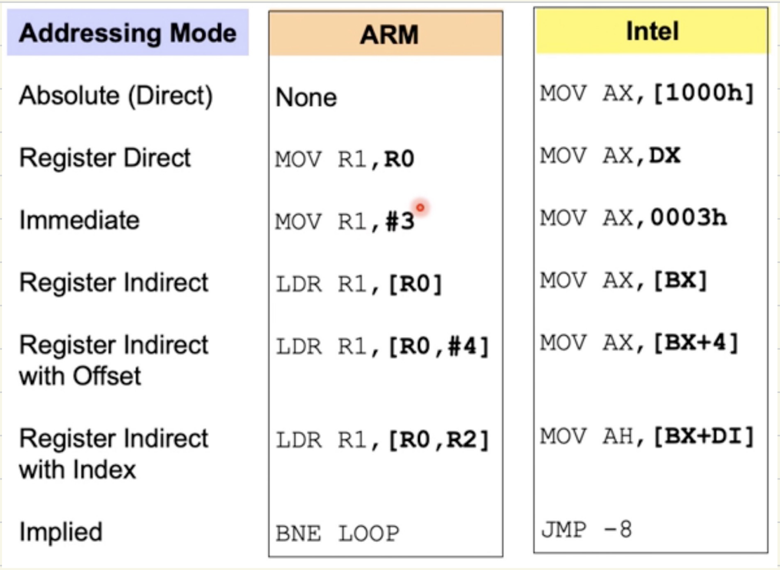
→ parts of the system kernel are being constantly executed
→ i/o intensive codes
→ time critical code : sensors and reactions eg. Anti-lock break

→ embedded system.

3. Addressing Mode AM

- how is data is accessed rather than processed
- correctly allows the CPU to identify the actual operand or the address location where the operand is stored

- AM types
→ Register direct
→ Immediate data
→ Register indirect
→ Register indirect with offset
→ Register indirect with index register
→ pre and post auto-indexing



Addressing Mode	ARM	Intel
Absolute (Direct)	None	`MOV AX,[1000h]`
Register Direct	`MOV R1,R0`	`MOV AX,DX`
Immediate	`MOV R1,#3`	`MOV AX,0003h`
Register Indirect	`LDR R1,[R0]`	`MOV AX,[BX]`
Register Indirect with Offset	`LDR R1,[R0,#4]`	`MOV AX,[BX+4]`
Register Indirect with Index	`LDR R1,[R0,R2]`	`MOV AH,[BX+DI]`
Implied	`BNE LOOP`	`JMP -8`

$$\begin{array}{rl} & 0x00000104 \\ R2 & \underline{0x00000108} \\ & 210 \end{array}$$

4 Register Direct

- operand is the content of the specified register
- both for source and destination
- eg. MOV R1, R0

$$\underset{\text{dst}}{R1}, \quad R0 \leftarrow \text{src}$$

- very fast access cause no need to memory ∴ used to optimise execution speed

- all 16 registers of ARM can be a register direct operand (src/dst) (doesn't matter)

```
     MOV  R3, LR ;  LR = linked register
read about the 'S' → MOVS R0, R0 ;  tests for N or Z condition in R0
     MOV  PC, R1 ;  makes execution flow of processor jump
```

5. Immediate Addressing

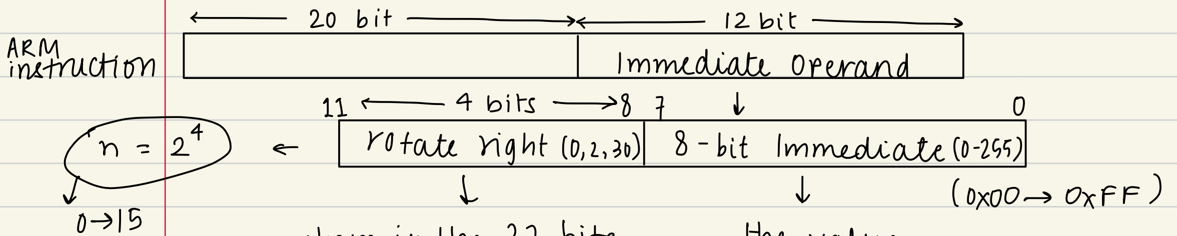
- operand is directly specified within the instruction itself
- eg. MOV R1, #3

  ↳ # symbol ⇒ immediate value

- can only be a source
- loading constant values into registers

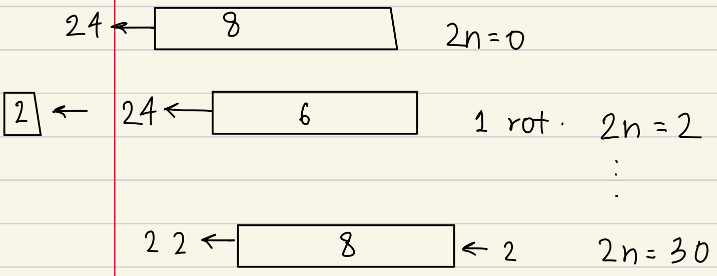
- how to encode 32 bit? instruction → 32 bits = opcode + operand

ARM allocates 12 bits for the immediate operand.



where in the 32 bits will that value be repensented.
(rotated by 2n) ← to the right

the value



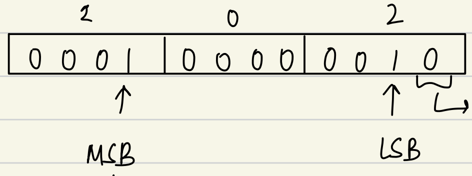
hi girl!

✓ MOV R0, #0x100 ≡ right rotate 0x01 by n=12

$$\overset{7}{1}0000000\overset{0}{0} \equiv \overset{31}{}\ \overset{16\,15}{}\ \overset{8}{}\ \overset{7}{0}000000\overset{0}{1}$$

(32−8)/2

x MOV R1, # 0x102 ⟶ MOV R1, # 0x100
ADD R1, R1, #2



odd no of 0 on RHS ⇒ can't rotate properly

1 0 0 0 0 0 0 1 ← number to rep. ≡ 0x81

rotation happens in pairs

6. Register Indirect

- register direct and immediate addressing do not allow CPU to access operands stored in memory
- C variables and arrays are usually stored in memory
- how do you specify a 32-bit address in a memory using a 32-bit long instruction

→ ARM specifies a 32-bit address of the operand in a 32-bit register

→ the register with the memory address points to the memory location where the operand is stored

∴ register points to the memory location where the operand is stored

- ARM uses LDR and STR mnemonics for indirect

LDR

- LDR : loads / reads memory content to a register

LDR R1 , [R0]
$\overset{\uparrow}{\text{dst}}$ $\overset{\uparrow}{\text{src}}$

go to memory address and copy subsequent locations until 32-bit reg is full (ARM: 4 memory locations worth)

by data alignment, the 4-byte data to be written or read must start at a memory location that is a multiple of 4

↳ else performance degradation

STR (read store)

- STR : copies register content to memory

$$\underset{}{\text{STR}}\quad \underset{\text{src, reg}}{\underset{\uparrow}{\text{R1}}}, \underset{\text{dst, memory}}{\underset{\uparrow}{\text{[R0]}}}$$

7. Register indirect with offset

- adds a specific offset value to the indirect register to compute the effective address (EA) in memory

- LDR R1, [R0, #4]
  - R1 ← dst, reg
  - R0 ↳ base add.

  if R0 = 0x0100
  it will take data from 0x 104
  value of R0 is unchanged

- prog. example

C:
```
int i[5];      //base add = 0x100        i[1] : 0x104 - 0x107
i[0] = 7;
i[4] = 7;
```
↓

assembly code:
```
reg to reg     { MOV  R2, #0x100       // initialise base add. into R2
               { MOV  R1, #7           // value into R1
reg to memory  { STR  R1, [R2, #0]     // store 7(R1) in R2
               { STR  R1, [R2, #16]    // store 7(R1) in R2+16
```

LDR: memory to reg

8. Register Indirect with Index Register

- this variant adds the content of the index register to the indirect register to compute EA
  ↳ effective address

- LDR R1, [R0, R2]
  ↓ dst   ↳ src

- advantage is that index value can be modified during run-time

- prog. example : initialise all 400 elements in array i to zero

C
```
int i[400];
int n=0;
while (n<400) {
  i[n] = 0;
  n = n+1;
  }
```
↓

assembly
```
MOV R2, #0x100     // base add
MOV R0, # 0        // value
MOV R1, #0         // counter
:                                       base address
STR R0, [R2, R1] (2) // put R0 in R2 + offset
ADD R1, R1, #4 (1)   // R1 = R1 + 4
```
memory addresses

loop back 399 times

$\underline{\underline{(1)}} \downarrow$ 3 x 400 cycles

9. Register Indirect with autoindexing

- autoindexing allows the indirect register's content to be modified during execution

- autoindexing provides an efficient way to access consecutive array elements

a) with offset

adds offset value to AR (autoindex register) to compute EA & modifies AR

- LDR R1, [R0, #4]!

$$\underset{\substack{\uparrow \\ \text{dst}}}{\text{R1}},\ \underset{\substack{\uparrow \\ \text{src, memory}}}{\underline{[\text{R0},\ \#4]}}\underset{\substack{= \\ \hookrightarrow}}{!}\ \text{autoindexing}$$

R0	0x00000100
R1	0x12345678
R0	0x00000104
R1	0xAABBCCDD

both values are modified

```
c {  int i[400];                 MOV R2, #0x100     // base add in R2
     int n=0;                    MOV R1, #0         // initial value
     while (n<400) {      →      STR R1, [R2]       // store R1 in memory
        i[n] = 0;                                      loc. R2
        n = n+1;        loop     STR R1, [R2, #4]!  // stores R1(0)
     }               398 times                         in R2+4
                                                       and then
                                                       modifies R2
```

preindex ↗ (STR R1, [R2, #4]!)

2×400 = 800 cycles

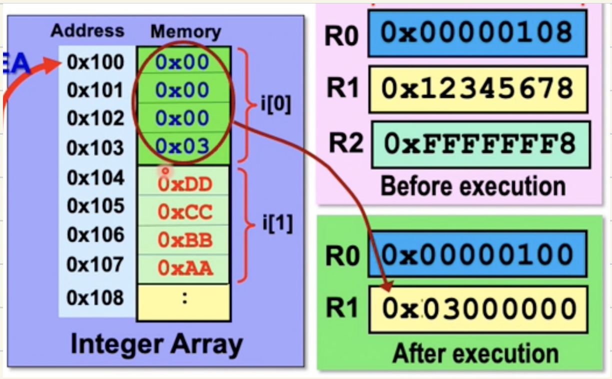
STR R1, [R2], #4

↑ post index

2×400 = 800 cycles

b) with index register

- LDR R1, [R0, R2]!



10. pre and post indexing

LDR R1, [R0, #4]! off set w/ autoindexing	R0 = R0+4
	R1 = mem[R0]
		update first

LDR R1, [R0, R2]! index w/ autoindexing	R0 = R0+R2
	R1 = mem[R0]
		update first

LDR R1, [R0], #4	R1 = mem[R0]
	R0 = R0+4
		update later

LDR R1, [R0], R2	R1 = mem[R0]
	R0 = R0+R2	

∴ pre index

post index

no exclamation

11. Stack System

- stack is FILO (first-in-last-out) data structure
- it's maintained in memory's data area
  ARM manages it by a dedicated stack pointer (SP) or R13
- FD (fully descending) stack grows towards lower memory addresses (ie. upwards)
  ↓
  starts at 0xFF000000 in VisUAL ARM
- puts the stack v far away from code and data
- SP points to the top-most (last added) or the space above it
- you can push, pop and access
- types of implementation : FD, FA, ED, EA
  - ↳ how to grow (high → low or low → high)
  - ↳ pt to new location or filled location

12. ARM Stack Implementation FD

- FD = fully descending
- SP pts to filled location
- SP moves upwards towards lower memory addresses

```
LDR R0, [R13]            //access item
        ↳ or SP          use interchangeble
STR R1, [R13, #-4]!  ⤷ push R1 to stack    first update, then move
LDR R2, [R13], #4        // pop stack into R2
               ↑           first remove, then move
           moves down
```

13. ARM Stack Implementation EA

- E → stack pts to empty , A ⇒ pointer moves down on add.

```
LDR R0, [R13, #-4]     //accessing value
STR R1, [R13], #4      // push value     store, then move
LDR R2, [R13, #-4]!    // pop value to R2   move, store,
                                             update
```

14. PC-related Addressing modes

15. Absolute Jump

- mostly, code must be run sequentially, so the addresses
- but, a new address can be loaded into the PC to alter the sequential order of program execution
- an absolute jump to a new code position is done by loading the address to jump to in the PC
- MOV PC, #0x060 remember: in ARM, PC pts 8 bytes ahead
- absolute jump is not position-indepent. one must know the code address

16. Relative Jump

- adding an offset to the PC
- a relative jump is done using the branch instruction w/ an appropriate signed offset (range for ARM = ±32Mb
- B CodeB
  ↳ branch can be conditional or unconditional (this)

∵ ARM is 8 bytes ahead, ∴ offset = resultant - 8
↳ next page

- this is position-independent

17. Position Independent

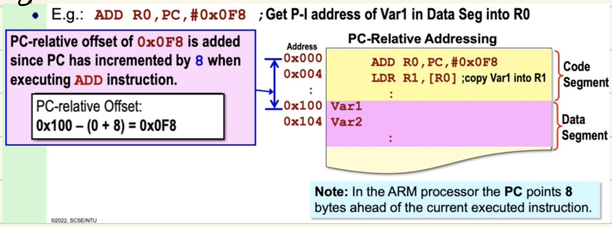
- across devices, programs can be stored anywhere in the memory
- position-independent allows codes to be executed correctly regardless of where they are stored in the memory
- position-dependent means knowing the exact address but that is hard to do across devices
- relative jump solves that problem

18. Accessing Data

- P-I programs require data to be accessed relative to the PC
- PC-relative addressing is used to access variables in the data segment of program in memory

PC-relative offset = var add − (PC value + 8)

```
ADD R0, PC, #0x0F8
LDR R1, [R0]
```



- E.g.: `ADD R0,PC,#0x0F8` ; Get P-I address of Var1 in Data Seg into R0

PC-relative offset of 0x0F8 is added since PC has incremented by 8 when executing ADD instruction.

PC-relative Offset:
0x100 – (0 + 8) = 0x0F8

```
0x000    ADD R0,PC,#0x0F8
0x004    LDR R1,[R0] ;copy Var1 into R1
  :        :
0x100  Var1
0x104  Var2
           :
```

Note: In the ARM processor the **PC** points **8** bytes ahead of the current executed instruction.