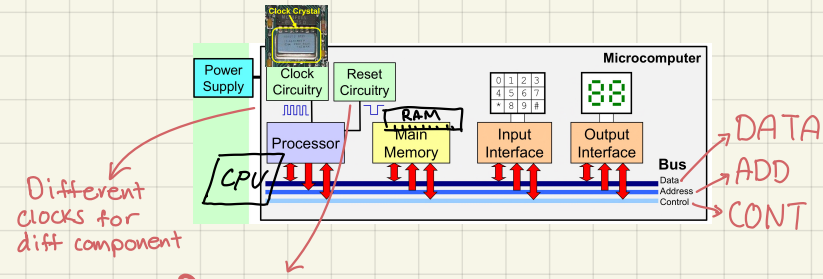
Part 1

Space

Bit = 1
Byte = 8
K Byte = 10^3
M Byte = 10^6
G Byte = 10^9

Physical memory
Kilobyte = 2^{10}
Megabyte = 2^{20}
Gigabyte = 2^{30}
Terabyte = 2^{40}
Petabyte = 2^{50}

Von Newmann

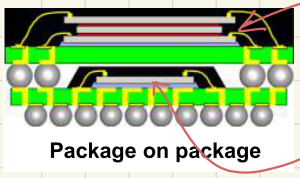


Reset Circuit

→ Manual reset pressed for many cycles of clock (human slow)

→ Boot up signal that loads the RAM with the starting signal

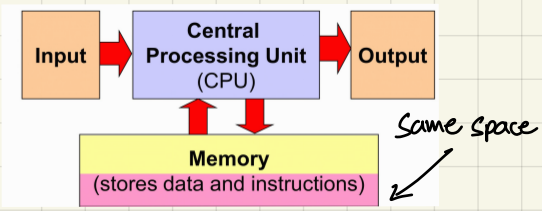
Apple's combined CPU



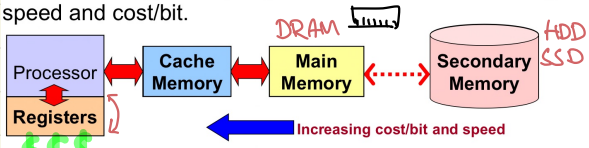
1) Save space
2) Minimize track length

Chapter 2

Stored Program Concept

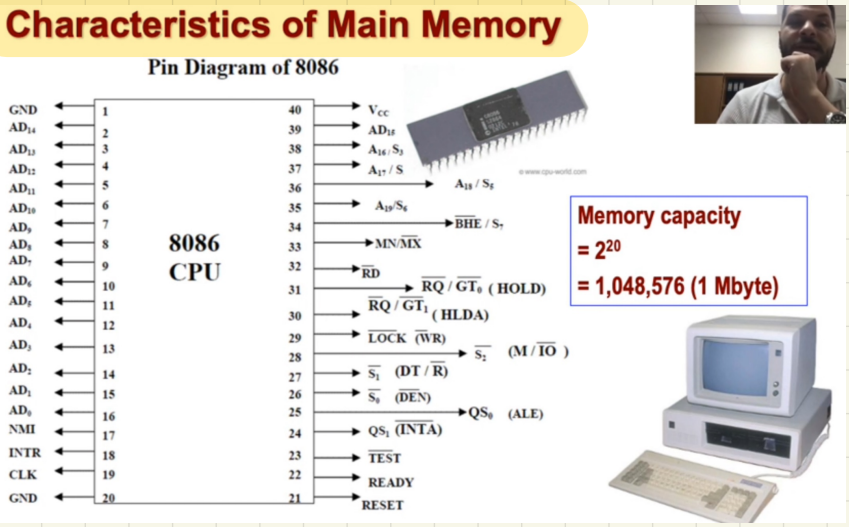


Memories are generally organized in levels of increasing speed and cost/bit.



Registers	Very fast access but limited numbers within CPU. Operates at CPU clock rate (size: 2-128 registers)
Cache Memory	Fast access static RAM close to CPU. Typical access time 3-20nS (size: up to 512 kB)
Main memory	Usually dynamic RAM or ROM (for program storage). Typical access time 30-70nS. (size: up to 16GB) Dram
Secondary Memory $	Not always random access but non-volatile. Maybe be based on magnetic or flash technology. Typical access time 0.03-100mS. (size: up to 4TB) NON-Volatile Memory → Cheap

Characteristics of Main Memory



How many address bits?

→ look @ AD_0 to AD_{19} = 20 bits
= 2^{20} ⇝ 1 MB

∴ 1MB worth of address locations

Floating Point Unit

→ part of processor, extremely big

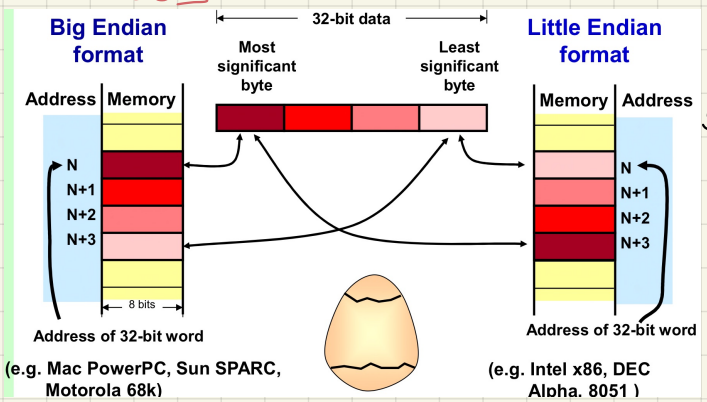
↳ Floating point is always signed

Type	Bytes	Bits	Range	
signed char	1	8	-128 -> +127	
unsigned char	1	8	0 -> +255	
short int	2	16	-32,768 -> +32,767	(+/-32KB)
unsigned short int	2	16	0 -> +65,535	(64KB)
unsigned int	4	32	0 -> +4,294,967,295	(4GB)
int	4	32	-2,147,483,648 -> +2,147,483,647	(+/-2GB)
long int	4	32	-2,147,483,648 -> +2,147,483,647	(+/-2GB)
long long int	8	64	-(2^32) -> (2^32)-1	
float	4	32		
double	8	64		
long double	12	96		

To increase size, use long

Data Organization

Numbers



Big: Most Significant byte @ smallest address

Small: Least Significant byte @ biggest address

↖ This is literally a meme

Boolean Representation

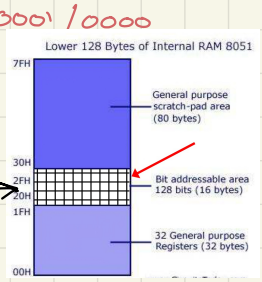
→ May be inefficient, as storage is 1 byte

→

0029	0x01
0030	0x00

one bit used
0001 / 0000

→ New storage have bitwise data



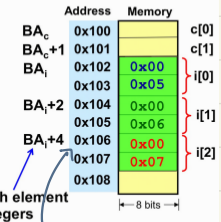
Characters

```
main(){
    char c;
    c = 'a';
}
```

How? Encode required
7 bit ASCII
16 bit Unicode

Array

```
main()
{
 char c[2];          //2 element char array c
 short int i[3];     //3 element integer array i
 i[0] = 5;           //assign values to array i
 i[1] = 6;
 i[2] = 7;
}
```



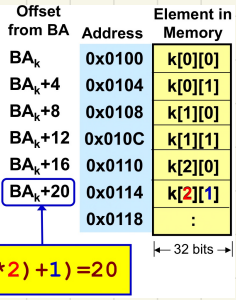
Offset ($n \times 2$) from base address (BA_i) to access each element n of the array `i` consisting of 2 byte-sized short integers

Only store the base address of the array & the offset

2 byte: short int

```
main()
{
 int k[3][2];  //a 3x2 integer array
}
```

- The offset from BA for element `k[a][b]`
  = `sizeof(int)*((2*a)+b)`



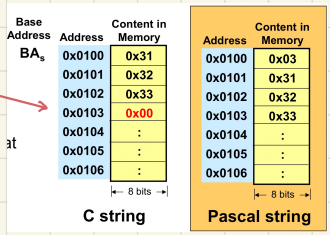
4*((2*2)+1)=20

Calculate number of units first, then multiply by element size (byte)

String (c)

```
main(){
    char s[4] = "123";
}
```

last element is null
'\n' or 00



String (Pascal)

```
main(){
    char s[4] = "123";
}
```

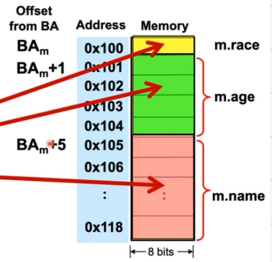
Starts w. string length at the front

↳ max length is 0xFF = 255

↳ 256 is also "correct" as it includes the header

Structure Representation

```
main()
{
 struct Man ; //define structure Man
 {
  char race;
  int age;
  char name[20];
};
 Man m;
}
```



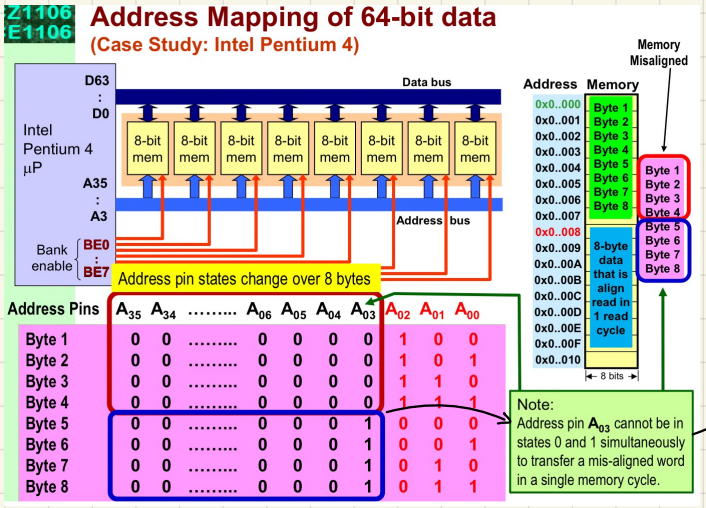
16 bits = 2 byte!
32 bits = 4 bytes!

— Chapter 2

Data Alignment

Data Type	Size (Byte)	Example of allowable start addresses due to alignment
char	1	0x..0000, 0x..0001, 0x..0002
short	2	0x..0000, 0x..0002, 0x..0004
int	4	0x..0000, 0x..0004, 0x..0008
float	4	0x..0000, 0x..0004, 0x..0008
double	8	0x..0000, 0x..0008, 0x..0010
pointer	8	0x..0000, 0x..0008, 0x..0010

→ The address only can be a divisible number by the type



Intel Pentium 4 µP System bus

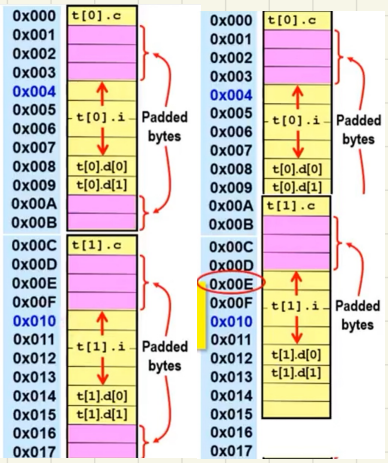
1) A35-A03 is the selector switch.
   ↳ Can ignore first 3 bits (A02, A01, A00) because you want to x-fer in 8 byte chunks
2) If data is misaligned, need to x-fer twice to get the data
   For example, here shows A03 at 0 & 1
3) For less than 8 bit, bank enable can change what is activated

Data Padding

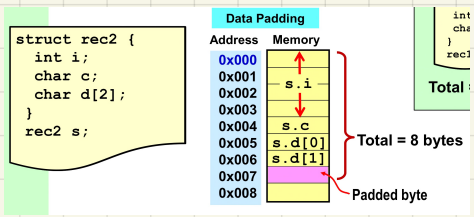
Data Type	Size (Byte)	Example of allowable start addresses due to alignment
int	4	0x..000, 0x..004, 0x..008, 0x..00C, 0x..010, 0x..014, 0x..018, 0x..01C,

```
struct rec1 {
  char c;
  int i;
  char d[2];
}
rec1 r;
  :
rec1 t[10];
```

An array of rec1




```
struct rec2 {
  int i;
  char c;
  char d[2];
}
rec2 s;
```



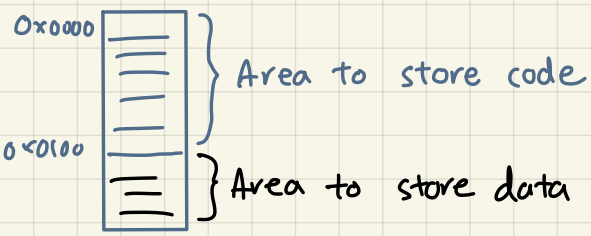
For better space allocation, we can put the multi byte data at the front, then single at the back

Example:
What are valid pointers for a 64-bit machine taking correctly aligned Int?
Int is 4 bytes ...
∴ Valid address: 0x 10010, 0x FFF10, 0x 10518, 0x CAC44, 0x 23204, 0x C3248, 0x 2FC1C, 0xDCE1C

even though machine takes 0x00 → 0x08, if there is array of INT, makes it fucked

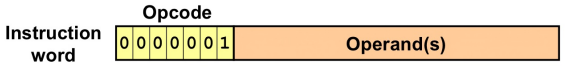
0xFC316 is not valid
↳ 0011 0001 0110
misaligned!
.. 1110 0001 1100
aligned?
↳ 4 address = 4 bytes = INT?

Chapter 3
Instruction Organization



Opcode and Operands

How are opcode encoded in an instruction?



If there are 80 different operations, (e.g. **add**, **sub**, etc) then at least 7 bits ($2^6 < 80 < 2^7$) of opcode is needed to represent all the unique bit patterns.

The **more variety** of operations supported by the instruction set, the **longer the length** of each instruction.

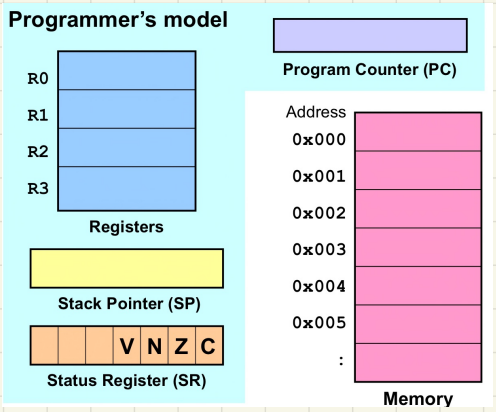
Are there many ways to specify the operand(s)?

Numerous. An operand can be stored as part of the **instruction**, stored in a **register** or stored in **memory**.

The method by which the operand is specified is known as **addressing mode**.

Programmer's Model

Data Processing	Data Transfer	Program Control
ARM examples:	ARM examples:	ARM examples:
ADD R0,R1,R2	MOV R1,R0	B Back
SUB R1,R2,#3	STR R0,[R2,#4]	BNE Loop
EOR R3,R3,R2	LDR R1,[R2]	BL Routine



ARM Programmer Model

There are 16 32-bit registers.

R0 – R12 are general purpose registers.

R13 is the Stack Pointer (**SP**).

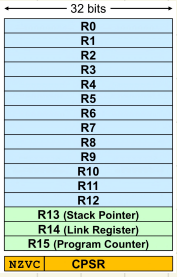
R14 is the Link Register (**LR**).

R15 is the Program Counter (**PC**)

The Current Processor Status Register (**CPSR**) holds the condition code bits (**NZVC**)

CPSR bit(s)	Name	Description (NZVC CPSR)
31	N	Can set if last operation produced a **negative** result
30	Z	Can set if last operation produced a **zero** result
29	C	Can set if last operation produced a **carry out** in the most significant bit
28	V	Can set if the last operation produced an **overflow** for a **signed** arithmetic operation

N – Negative; **Z** – Zero; **C** – Carry ; **V** – Overflow



Program Counter (R15)

- Stores the address (intervals of 32bit) so it can feed the ARM instructions. Due to pipeline of ARM, it points to 8 byte in advance

Stack Pointer (R13)

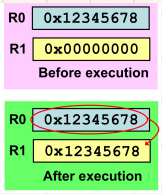
- use to temporary store register

Link Register (R14)

Move

The right operand is the source and left operand is the destination.

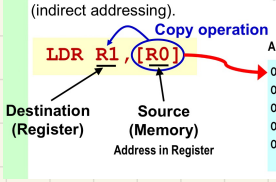
In this example, the both operands are **registers** in the ARM CPU.

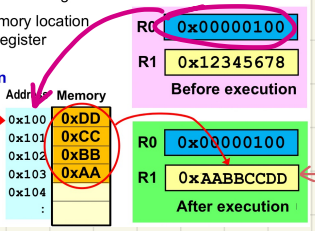


LOAD

The LDR Instruction

- The LDR operator copies **memory** content to a register.
- The left operand is always a destination register.
- The right source operand is a memory location whose address is contained in a register (indirect addressing).





[R0] means take the content as an address, 0x00000100 and find the next 32 bits

← little endian

How CPU Works

1) Fetch instructions - from memory
2) Decode instructions - 01100110 into instructions
3) Fetch data - any relevant info needed
4) Execute - move data or perform calculation

LOOP

→ Chapter 4

What is assembly?

```
if (a > c)
  b = a;
else
  b = c;
```

C program example

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

ARM assembly program equivalent

High level program like C converted into assembly language through translation which is then used to instruct the CPU on a one-to-one basis. AKA: CMP - compare, R0 - reg 0

Why is assembly?

→ faster execution speed : Noise-Cancelling headphones

→ Low program size

→ Exploit optimized features that compiler does not know yet

→ Cybersecurity to hack into systems

→ Time critical codes; ABS in cars

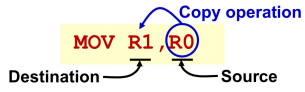
Register Direct

→ The operand is the register value
→ No need to access memory

Operand is the content of the specified **register**.

Register direct can be used for both destination and source operand.

In the MOV instruction, the right operand is the source and left operand is the destination.



R0	0x12345678
R1	0x00000000

Before execution

R0	0x12345678
R1	0x12345678

After execution

A **fast** addressing mode since no further memory access is involved during execution.
Should be used to optimise execution speed.

- All ARM's 16 registers can be a register direct operand.
- These registers can be either a source or destination operand.

```
MOV   R3,LR  ;make copy of LR in R3
MOVS  R0,R0  ;test for N or Z condition in R0
MOV   PC,R1  ;make a jump to address in R1
```

→ S flag

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
SP
LR
PC

Reg direct mode limited to 15 reg

Addressing Modes

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

Clock Cycle

1 clock cycle = 1 memory access
MOV R0,R1 only accesses R1
∴ 1 cycle to access instruction stored in memory

★ R0 is very fast, negleg. time

Immediate Addressing (Constants!)

Operand is directly specified **within** the **instruction** itself.

"#" symbol precedes the immediate value.

Example: MOV R1,#3 (#3 Immediate Value, Copy operation)

#3 into R1

After execution, the **immediate value** is **copied** into the destination register (left operand).

Immediate addressing can only be used as a **source operand**.

Used for loading **constant values** into registers. Values must be known at the time of coding (e.g. load loop count into a loop counter register).

R1	0x12345678

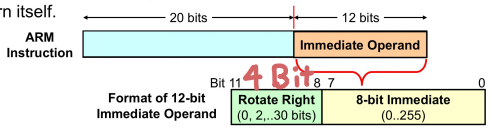
Before execution

R1	0x00000003

After execution

However, even though ARM provides 32-bit register, some space is used for opcode.

The immediate value is specified **within the instruction** bit pattern itself.

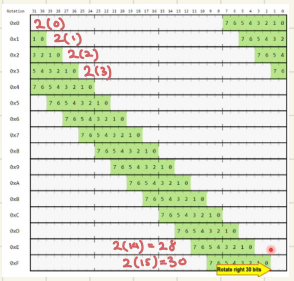


How can a 12-bit operand encode a 32-bit immediate value?

It can only describe a **subset** of all 2^{32} possible values.

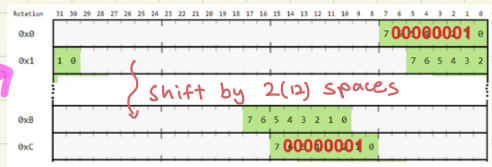
Immediate value is a number between (**0..255**) rotated right by **2*n*** bits, where the value of ***n*** is given by 4 bits (**0** ≤ ***n*** ≤ **15**).

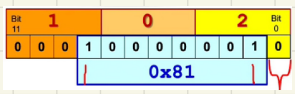
3(15) = 30



How to solve 3 examples

```
MOV R3,#0xFF    ;immediate values within 8 bits always valid
MOV R0,#0x100   ;right rotate 8-bit value of 0x01 with n=12
MOV R1,#0x102   ;this is not a valid immediate value
```





1 bit only, ARM can only shift by 2 bits

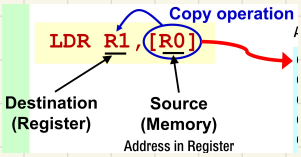
A combination of instructions can be used to achieved the desired immediate values that is not valid.

```
MOV R1,#0x100 ; load 0x100 to R1
ADD R1,R1,#2  ; add 2 to R1
```

★ MOV R0, #0x258 is OK
0010 0101 1000
8 bits!

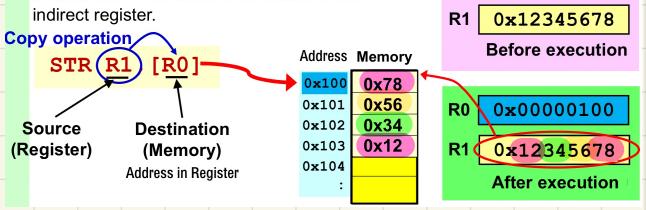
Register Indirect (like pointer)

LDR
→ On top



STR store

- The STR operator is used to copy register content to **memory**.
- The left operand is always a source register.
- The right destination operand is a memory location whose address is contained in the indirect register.



R0	0x00000100
R1	0x12345678

Before execution

R0	0x00000100
R1	0x12345678

After execution

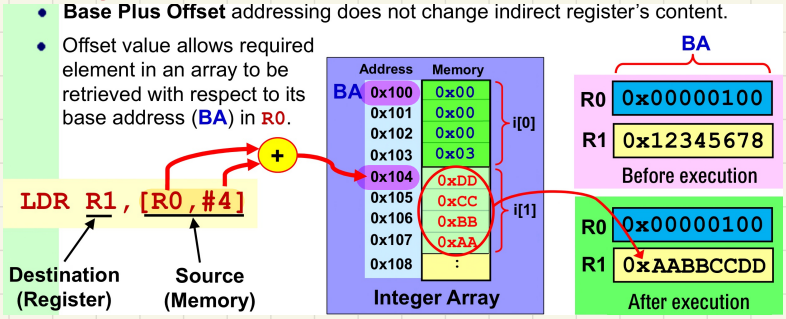
Data Alignment

The **4-byte data** read or written to memory must start at an address that is a **multiple of 4**.

The effects of an unaligned memory access depend on the ARM architecture but they invariably result in **performance degradation**.

Register Indirect w. Offset

- **Base Plus Offset** addressing does not change indirect register's content.
- Offset value allows required element in an array to be retrieved with respect to its base address (**BA**) in R0.



Accessing Arrays Example

- Use base plus offset to access array element whose index is known during coding time.

```
main()
{
// assume base address
// of array i is 0x100
int i[5];

i[0]=7;
i[4]=7;
}
```

C program example
Assign first & last elements of array `i` with value of 7.

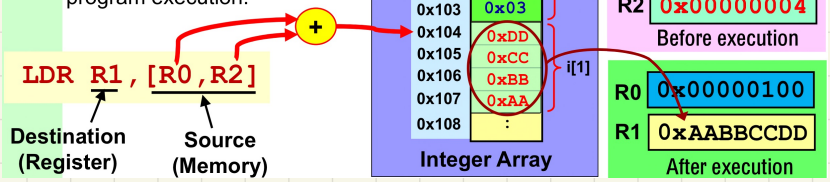
```
MOV  R2,#0x100   1
MOV  R1,#7       2
STR  R1,[R2,#0]  3
STR  R1,[R2,#16] 3
```

Using register indirect with base plus offset

1. Initialize base address of array `i` into register `R2`.
2. Load value of 7 into register `R1`.
3. Store the value of 7 into i[0] and i[4] using offsets of 0 and 16 of register R2 respectively .
   - In computing offset, note that each integer element occupies 4 bytes in memory.

Register Indirect w. Variable Offset

- **Base Plus Index Register** does not change base register's content.
- **Modifiable** index value in R2 allow different array elements to be retrieved with respect to base address (**BA**) during program execution.



Variable Offset Example

- Use base plus index register to access each array element in turn.

```
main() {
// assume base address
// of array i is 0x100
int i[400];
int n=0;

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

C program example
Initialise all 400 elements in array `i` with zero.

```
MOV  R2,#0x100   1
MOV  R0,#0       2
MOV  R1,#0       2
  :
STR  R0,[R2,R1]  3
ADD  R1,R1,#4    4
```

loop back 399 times

3×400 = 1200 cycles

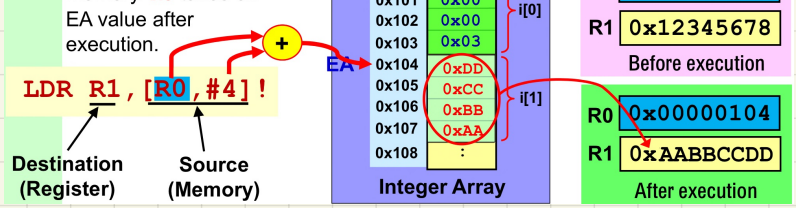
Using register indirect with base plus index

1. Initialize base address of array `i` into register `R2`.
2. Load value of 0 into register `R0` and `R1`(index register).
3. Store 0 in `R0` into i[n] using current index value in `R1` plus base address in `R2`.
4. Increment index by 4, the size of each integer element in array.

Offset with autoindexing

R0 takes on a new value

- Adds offset value to the autoindex register (AR) to compute effective address (EA) and AR gets modified.
- "`!`" in mnemonic causes autoindex register to be modified with the EA.
- Offset value added to autoindex register R0 to compute the EA in memory. R0 takes on EA value after execution.



Autoindexing Example

Faster! 800 vs 1200

- Use offset with autoindex to efficiently access each array element in turn.

```
main() {
// assume base address
// of array i is 0x100
int i[400];
int n=0;

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

C program example
Initialise all 400 elements in array `i` with zero.

```
MOV  R2,#0x100   1
MOV  R1,#0       2
STR  R1,[R2]     3
  :
STR  R1,[R2,#4]! 4
```

loop back 398 times

2×400 = 800 cycles

Using register indirect plus offset with autoindex

1. Initialize base address of array `i` into register `R2`.
2. Load value of 0 into source register `R1`.
3. Store 0 in `R1` into first element of array i[0].
4. Store 0 in `R1` into i[n] using current effective address (EA) of autoindex register `R2` plus offset `4`. Then put this EA into `R2`.

Preindex OR Postindex

Calculate R0 before using

```
LDR R1,[R0,#4]! ; R0 = R0+4
                ; R1 = mem[R0]
```
Offset with Autoindexing (pre-index)

```
LDR R1,[R0,R2]! ; R0 = R0+R2
                ; R1 = mem[R0]
```
Index with Autoindexing (pre-index)

Calculate R0 after using

```
LDR R1,[R0],#4  ; R1 = mem[R0]
                ; R0 = R0+4
```
Offset with Autoindexing (post-index)

```
LDR R1,[R0],R2  ; R1 = mem[R0]
                ; R0 = R0+R2
```
Index with Autoindexing (post-index)

Example!

```
loop.
MOV  R2,#0x100   1
MOV  R1,#0       2
  :
STR  R1,[R2],#4  3
```

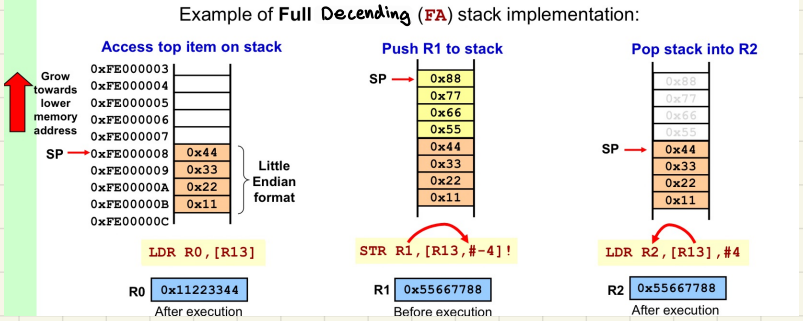
loop back 399 times

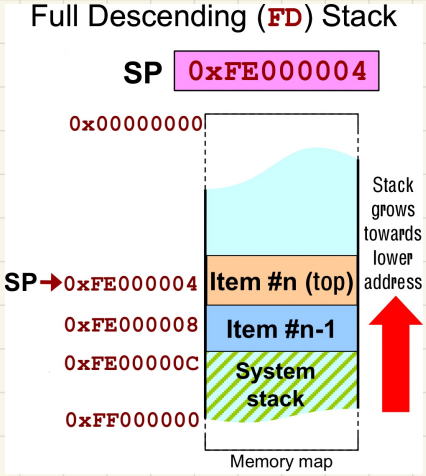
2×400 = 800 cycles

Stack!

ARM Stack Implementation (FD)

- The are 4 possible stack implementations supported by the ARM instruction set.
  - Full Descending, Full Ascending, Empty Descending and Empty Ascending

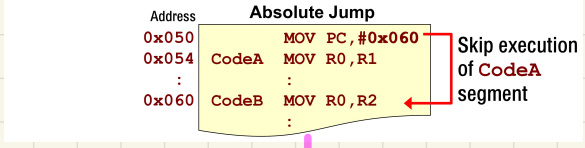


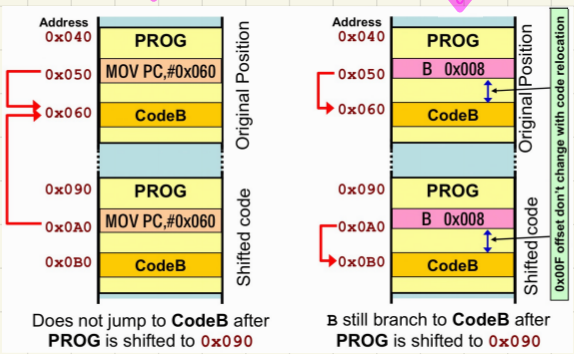


Absolute Jump

Jump specific address of instruction

Example: `MOV PC,#0x060` ;Jump to CodeB





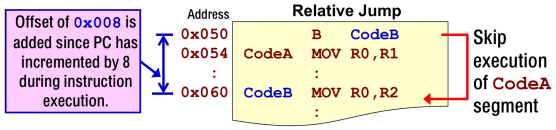
Relative Jump

0x10 = 16 bytes

Code B causes a 8 byte jump, not 16 bytes as the ARM processor reads 8 bytes ahead of "current code"

A **relative jump** is done using the branch instruction (e.g. `B`) with an appropriate **signed offset.** (Note: the range of this offset in ARM is **+/- 32 Mbytes**).

Example: `B CodeB` ;Jump to CodeB



Relative jump supports **position-independent** code.

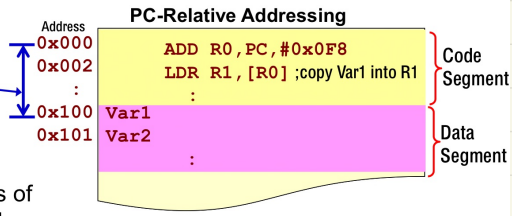
Note: In the ARM processor the PC points 8 bytes ahead of the current executed instruction due to its pipeline architecture.

If you want to use in only 1 pc

- Position-independent (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.
  - 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:
Var address – (PC value + 8)



- Referencing absolute address of variable in whatever ways will violate **P-I** requirements.

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

When executing code, PC is always two instructions ahead
This means 8 bytes; as one line takes up 4 bytes.
∴ @ 0x1000, PC=0x1008

→ Chap 5

Instruction Set

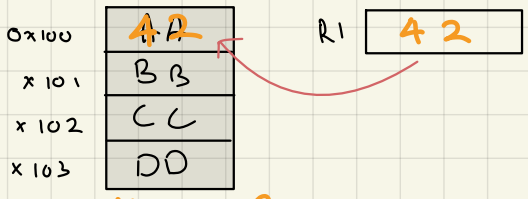
Data Transfer	Data Processing	Program Control
ARM examples:	ARM examples:	ARM examples:
`MOV R1,R0`	`ADD R0,R1,R2`	`B Back`
`STR R0,[R2,#4]`	`SUB R1,R2,#3`	`BNE Loop`
`LDR R1,[R2]`	`EOR R3,R3,R2`	`BL Routine`

Data transfer – instructions that move data between registers and/or memory.

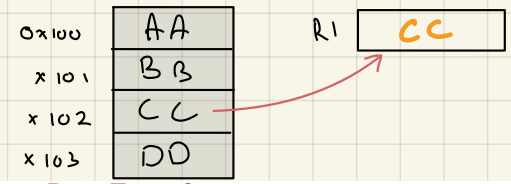
Data processing – instructions that modify the data in register through arithmetic or logical operations.

Program control – instructions that alter the normal sequential execution flow of a program.

MOV R4, #0x100 MOV R1, #0x42
STRB R1, [R4]



MOV R4, #0x102
LDRB R1, [R4]



Data Xfer

Register Data Transfer

Moves source operand to the destination register.

With `MOV`, the source operand can use either **register direct** or **immediate** addressing.

```
MOV   R1,R0  ; make copy of R0 in R1
MOVS  R0,#0  ; move 0 into R0 and set Z flag
```

With move complement (NOT) `MVN`, the source operand is bit-wise inverted before moving into the destination register.

```
MVN   R1,R0  ; R1 = NOT (R0)
MVN   R0,#0  ; move 32-bit value of -1 into R0
```

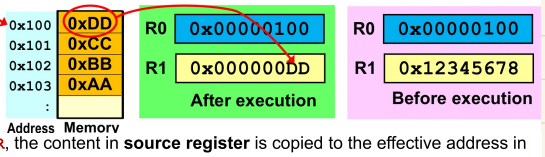
R0 = 0xFFFFFFFF after execution

MOVS → flag update

06 Memory Data Transfer

- Data in memory can be transferred to or from a register.
  - With `LDR`, the memory data at the effective address is moved to the **destination register** using various **indirect register** addressing modes.

```
LDR   R1,[R0]  ; copy 32-bit value pointed by R0 into R1
LDRB  R2,[R0]  ; copy 8-bit value pointed by R0 into R2 (byte zero-extends to 32 bits)
```



  - With `STR`, the content in **source register** is copied to the effective address in memory using various indirect addressing modes.

```
STR   R1,[R0]    ; copy R1 (4 bytes) starting at address pointed by R0
STRB  R2,[R0,#1]! ; copy byte in R2 to only one address at [R0+1]; then R0=R0+1
```

LDRB at the end asks for a single byte, changes all the front to 0

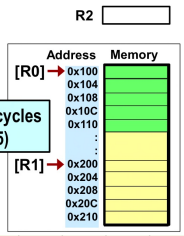
STRB stores the two least significant bytes into the target register. Make sure to check length of address, as doing STRB R0, [R1] may change R1 : 0x 00 00 00FF

Copying a Block of Memory

- Block copy is used to replicate a contiguous segment of memory from one location to another.
- The `MOV`, `LDR` and `STR` data transfer mnemonics are required to perform the block copy operations.

```
      MOV  R0,#0x100  ; setup source pointer
      MOV  R1,#0x200  ; setup destination pointer
loop  LDR  R2,[R0]    ; memory to register transfer
      STR  R2,[R1]    ; register to memory transfer
      ADD  R0,R0,#4   ; increment source pointer
      ADD  R1,R1,#4   ; increment destination pointer
```

loop back 5 times — 30 cycles (6x5)



Block copy 5 words starting at address 0x100 to 0x200

Using auto-indexing to reduce computational cycles

```
      MOV  R0,#0x100    ; setup source pointer
      MOV  R1,#0x200    ; setup destination pointer
loop  LDR  R2,[R0],#4   ; memory to register transfer
                        ; with post index autoindexing
      STR  R2,[R1],#4   ; register to memory transfer
                        ; with post index autoindexing
```

loop back 5 times — 20 cycles (4x5)

Block copy 5 words starting at address 0x100 to 0x200

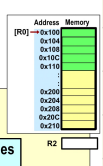
Using one pointer & many offsets, also only use 1 REG

- Further minor optimisation by using only **one pointer register** and make use of block copy offset.
- Be careful when using this technique as the immediate offset range for register indirect is limited to only **+/- 4096 bytes**.

```
      MOV  R0,#0x100    ; setup source and destination pointer
      #0x104            ← Use one register less, save 1 cycle
loop  LDR  R2,[R0],#4   ; memory to register transfer
      #0x204            ; with post index autoindexing
      STR  R2,[R0,#0xFC] ; register to memory transfer
                        ; with base plus offset of (0x200-0x100)+4=0xFC
```

20 cycles (4x5)

loop back 5 times — Offset range = ±4096



Data Processing / ADDITION

```
ADDS R2,R0,#8
```
Destination ← R2 ; + ← R0, #8

Condition Code Flags and ADD

- `ADD` can affect all the `N`, `Z`, `V`, `C` flags.

		Signed Number	Unsigned Number
(+ve)	0000 0001	(1)	(1)
(+ve)	+0111 1111	(127)	(127)
(-ve)	1000 0000	(-128)	(128)

N=1, V=1 ← 2's complement oVerflow

		Signed Number	Unsigned Number
(+ve)	0000 0001	(1)	(1)
(-ve)	+1111 1111	(-1)	(255)
(+ve)	0000 0000	(0)	(0)

Z=1, C=1 ← unsigned overflow

- The `V` flag when set, indicates an **overflow** when adding **signed** numbers.
- Overflow is detected when both **signed numbers** added have the **same sign** but the result has the **opposite sign**.
- The `C` flag when set, indicates an **overflow** when adding **unsigned** numbers.

`ADD`	(addition)
`SUB`	(subtraction)
`RSB`	(reverse subtraction)
`ADC`	(add with carry)
`SBC`	(subtract with carry)
`RSC`	(reverse subtract with carry)

```
SUBS R2,R0,#4 ; R2=R0-4
```
Destination ← R2 ; - ← R0, #4

To influence all the condition code flags (`N`,`Z`,`V`,`C`), the "`S`" suffix must be added to the `SUB` mnemonic.

`RSB` can be used to reverse the subtraction order.

```
RSBS R2,R0,#4 ; R2=4-R0
```

2 = 0010, -4 = 1100, -2 = 0'1110 (produces a borrow)

2 = 0010, -1 = 1111, 1 = 1'0001

- Subtraction is done by **adding** the minuend to the **negated** subtrahend.

A – B = A + (-B) (2's complement operation)

- **2's complement** is used to negate subtrahend.

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

In binary:
```
  ...0000 0011
+ ...1111 1111
  ...0000 0010
```

Minuend: 0x00000003 R0
Subtrahend: 0x00000001 R1
0x00000002 R2

= 0x00000003 (3) + 0xFFFFFFFF (-1) = 0x00000002 (2)

R0	0x00000003
R1	0x00000001
R2	0x12345678

Before execution

R0	0x00000003
R1	0x00000001
R2	0x00000002

After execution

- In the ARM's `SUB` instruction, the `C` flag clears (`C`=0) if the subtraction produce a **borrow** and `C` sets (`C`=1)otherwise.
- A borrow occurs in subtraction when the **unsigned value** of the Minuend is **less** than the unsigned value of the Subtrahend.

Borrow ← A (Minuend) – B (Subtrahend) = (A-B)

unsigned (Minuend) (A) < unsigned (Subtrahend) (B) → C=0

unsigned (Minuend) (A) ≥ unsigned (Subtrahend) (B) → C=1

- The `V` flag is set (`V`=1) when the result is out of the signed 32-bit range. $-2^{31} < (A-B) \geq +2^{31}$ → V=1
- An **unsigned underflow** is indicated by (`C`=0).

Convert Negative to Positive

- The code converts an integer array of 10 negative values to its equivalent positive values.
- The 2's complement operation on each retrieved word converts it from -ve to +ve. The –ve word in memory is then replace with its +ve equivalent.

```
        MOV  R0,#0x100  ; setup source pointer
Loop    LDR  R1,[R0]    ; get next –ve int in mem to R1
        MVN  R1,R1      ; complement –ve word
        ADD  R1,R1,#1   ; add 1 to do 2's complement
        STR  R1,[R0],#4 ; put result back then increment
                        ; to next array element
```
loop back 9 times

60 cycles (6x10)

1st –ve integer converted → R1 0x00000001

Address	Memory	
[R0] → 0x100	0xFFFFFFFF	(-1)
0x104	0xFFFFFFFE	(-2)
0x108	0xFFFFFFFD	(-3)
0x10C	0xFFFFFFFC	(-4)
0x110	0xFFFFFFFB	(-5)
0x114	0xFFFFFFFA	(-6)
0x118	0xFFFFFFF9	(-7)
0x11C	0xFFFFFFF8	(-8)
0x120	0xFFFFFFF7	(-9)
0x124	0xFFFFFFF6	(-10)

- The `SUB` instruction can be used to do the negation operation more efficiently.
- By **subtracting** the value **0** with the –ve value retrieved from memory, the result will be its +ve equivalent. e.g. 0 – (-5) = +5

```
        MOV  R0,#0x100  ; setup source pointer
        MOV  R2,#0      ; load zero into R2   (R2=0)
Loop    LDR  R1,[R0]    ; get next –ve int in mem to R1
        SUB  R1,R2,R1   ; do (0 – R1) to get +ve value
        STR  R1,[R0],#4 ; put result back then increment
                        ; to next array element
```
loop back 9 times

R1=R2 – R1
R1=0 – R1

50 cycles (5x10)

R1 0x00000001

Address	Memory	
[R0] → 0x100	0xFFFFFFFF	(-1)
0x104	0xFFFFFFFE	(-2)
0x108	0xFFFFFFFD	(-3)
0x10C	0xFFFFFFFC	(-4)
0x110	0xFFFFFFFB	(-5)
0x114	0xFFFFFFFA	(-6)
0x118	0xFFFFFFF9	(-7)
0x11C	0xFFFFFFF8	(-8)
0x120	0xFFFFFFF7	(-9)
0x124	0xFFFFFFF6	(-10)

- The **reverse subtract** instruction allow the immediate value of **0** to be used as the minuend, thereby removing the need for a zeroed register.

```
        MOV  R0,#0x100  ; setup source pointer
                        (Use one register less, save 1 cycle)
Loop    LDR  R1,[R0]    ; get next –ve int in mem to R1
        RSB  R1,R1,#0   ; do (0 – R1) to get +ve value
        STR  R1,[R0],#4 ; put result back then increment
                        ; to next array element
```
loop back 9 times

R1=0 – R1

50 cycles (5x10)

R1 0x00000001

Address	Memory	
[R0] → 0x100	0xFFFFFFFF	(-1)
0x104	0xFFFFFFFE	(-2)
0x108	0xFFFFFFFD	(-3)
0x10C	0xFFFFFFFC	(-4)
0x110	0xFFFFFFFB	(-5)
0x114	0xFFFFFFFA	(-6)
0x118	0xFFFFFFF9	(-7)
0x11C	0xFFFFFFF8	(-8)
0x120	0xFFFFFFF7	(-9)
0x124	0xFFFFFFF6	(-10)

Reduces registers needed from 3 to only 2

Carry Arithmetic

ARM provides arithmetic instructions that takes the **carry bit** into consideration.

These instructions are mainly used to support **multi-precision** arithmetic that involves data size larger than the 32-bit registers in the ARM CPU.

`ADC R2,R0,R1 ; R2=R0+R1+C`	ADD with carry
`SBC R2,R0,R1 ; R2=R0-R1+NOT(C)`	SUB with carry
`RSC R2,R0,R1 ; R2=R1-R0+NOT(C)`	RSB with carry

Logical Instructions

AND OR NOT

Logical instructions provide various **Boolean** operators.

`MVN` is a **two-operand** instruction that does the **NOT** operation.

Example: `MVNS R2,R2`

R2 `0x00000000` → R2 `0xFFFFFFFF` **N=1** and **Z=0**

Before execution / After execution

The `AND`, `ORR` and `EOR` operators are **three-operand** instructions for the **AND**, **OR** and **EX-OR** operations respectively.

Example: `ORRS R1,R1,#0x0000FFFF`

R1 `0x12345678` → R1 `0x1234FFFF` **N=0** and **Z=0**

Before execution / After execution

The "`S`" suffix can be used to influence the **N** and **Z** bits in the CC flags.

`AND` – **clear** specific bits in destination operand.

`ORR` – **set** specific bits in destination operand.

`EOR` – **complement** specific bits in destination operand.

AND truth table

A	B	Z = A.B
0	0 *	0
0	1	0
1	0 *	0
1	1	1

* Binary **0** **mask** is used to **clear** the bit

OR truth table

A	B	Z = A+B
0	0	0
0	1*	1
1	0	1
1	1*	1

* Binary **1** mask is used to **set** the bit

EX-OR truth table

A	B	Z = A⊕B
0	0	0
0	1*	1
1	0	1
1	1*	0

* Binary **1** mask is used to **complement** the bit

AND, ORR and EOR Applications

Bits 7 6 5 4 3 2 1 0

R0 `..01010101`

Initial condition of least significant 8 bits register **R0**

e.g. `AND R1,R0,#..11110000` (e.g. `AND R1,R0,#0xF0`)

R1 `..01010000` Bits 0 to 3 cleared after execution

e.g. `ORR R0,R0,#..11110000` (e.g... `ORR R0,R0,#0xF0`)

R0 `..11110101` Bits 4 to 7 set after execution

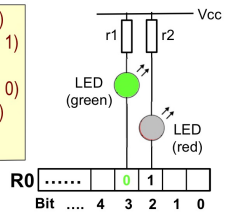
e.g. `EOR R2,R0,#..11110000` (e.g. `EOR R2,R0,#0xF0`)

R2 `..10100101` Bits 4 to 7 inverted after execution

Alternate LED Flashing

- Turn **Green** and **Red** LEDs on and off alternately.
- **Green** and **Red** LED states are mapped to **bits 3** and **2** of **R0** respectively. All other bits must not be affected during pattern change.
- `AND` is used to turn on the **active-low** LEDs and `ORR` is used to turn them off. Two patterns per cycle is needed to alternate the ON and OFF between LEDs.

```
Loop AND  R0,R0,#0xFFFFFFFB  ; turn on Red (bit 2 = 0)
     ORR  R0,R0,#0x00000008  ; turn off Green (bit 3 = 1)
     output pattern in R0 and time delay
     AND  R0,R0,#0xFFFFFFF7  ; turn on Green (bit 3 = 0)
     ORR  R0,R0,#0x00000004  ; turn off Red (bit 2 = 1)
     output pattern in R0 and time delay
loop back
```



Alternating patterns for bits 3 and 2 in R0

```
     AND  R0,R0,#0xFFFFFFFB  ; turn on Red (bit 2 = 0)
     ORR  R0,R0,#0x00000008  ; turn off Green (bit 3 = 1)
     output pattern in R0 and time delay
Loop EOR  R0,R0,#0x0000000C  ; flip state of bits 3 and 2
     output pattern in R0 and time delay
loop back
```

EOR mask = 00 .....1100$_2$

R0 ...

ROTATE / SHIFT

LSL LSR

ASR

ROR RRX

Shift and Rotate Instructions

ARM has several shift and rotate operations:

Logical Shift Left (`LSL`) and Logical Shift Right (`LSR`).

Arithmetic Shift Right (`ASR`)

Rotate Right (`ROR`) and Rotate Right Extended (`RRX`).

Shift performs multiply (shift left) or divide (shift right) by a factor of 2^N, where N is the no. of bits shifted.

e.g. 00000100_2 (4) → shift left 2 bits (×4) → 00010000_2 (16)

00010000_2 → shift right 1 bit (÷2) → 00000010_2 (2)

In signed or unsigned **multiply**, binary "0"' is shifted into the LSB of the register from the right using Logical Shift Left (`LSL`).

In **unsigned divide**, binary "0" is shifted into the MSB of the register from the left using Logical Shift Right (`LSR`).

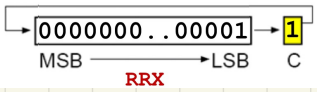
In **signed divide**, the sign bit is shifted into the MSB from the left using Arithmetic Shift Right (`ASR`).

The "`S`" suffix is used on the data processing operator to influence the `C` flag.

Rotate is also called **cyclical shift**, as no bits in the register is lost during the shifting operation.

In basic rotate right (ROR), the bit shifted out of register is returned in at the leftmost end and is also placed into the **C**-flag.

In rotate right extended (RRX), the **C**-flag is shifted into the register at leftmost end, while the bit shifted out replaces the current **C**-flag.



- The ARM **efficiently combines** the shift operation with the data transfer or processing instruction.
  - Shift operation is applied to the **rightmost** operand (2nd source operand).
  - Number of bits to shift is specified as an **immediate** value or a value within a **register** (dynamic shift):

```
MOV   R0,R0,LSL #1  ; R0=R0<<1          Shift R0 left by 1 bit
ADD   R2,R1,R0,LSR #2 ; R2=R1+R0>>1     ADD R1 with 2-bit right shifted R0. Put result in R2.
ADDS R2,R1,R0,LSL R4 ; R2=R1+R0<<R4     Shift R0 by R4 bits before ADD with R1. Update N,Z,V,C flags and result in R2.
ORRS R0,R0,R0,ROR #1 ; R0=R0||R0>>1     OR R0 with a 1-bit right rotated version of itself. Set C flag if bit rotated out is 1.
```

ADD R0, R0, R0, LSL #4

$2^4 = 16$

The item is multiplied by 16.

ADD R0 to (16)(R0)

= 17 (R0)

Count Number of 1's

- Count the number of binary 1s in register **R0**.
- Rotate R0 32 times (ROR), at each rotate use AND to mask all bits except LSB. Then add the LSB. The cumulated total will be the number of 1s in R2.

```
      MOV  R2,#0          ; clear 1-counter for binary 1s
      MOV  R3,#32         ; set loop counter to 32 times
Loop  MOV  R0,R0,ROR #1   ; rotate right 1 bit
      AND  R1,R0,#1       ; clear all bits except LSB
      ADD  R2,R2,R1       ; add LSB to 1-counter
      SUB  R3,R3,#1       ; decrement loop counter
```

loop back 31 times

Initial R0 = 0x22222222

R0	0x11111111
R1	0x00000001
R2	0x00000001

AND mask of 0x0000001

Changes after one loop

R3	0x0000001F

Count the number of binary 1s in R0

- LSR is used to shift content in R0 1 bit at a time into the C flag. Then ADDC can be used to sum the 1s going into the C flag. Loop ends when R0 has no more 1s and Z flag is set

```
      MOV  R2,#0          ; clear 1-counter for binary 1s
Loop  MOVS R0,R0,LSR #1   ; 1-bit right shift to move LSB
                          ; into C flag
      ADC  R2,R2,#0       ; add C flag to 1-counter
```

loop back if not zero

R2 = R2 + 0 + C

Initial R0 = 0x22222222

R0	0x11111111	C=0
R2	0x00000000	
R0	0x08888888	C=1
R2	0x00000001	

1st loop

2nd loop

Instruction Set / BRANCH / CMPRE

Conditional Branch (Bcc)

ARM provides conditional branch using Bcc.

If the condition specified in the condition field (cc) is **true**, a displacement is added to the **PC**, otherwise next instruction is executed.

Bcc uses **PC-relative** addressing mode with a displacement range of ±**32MB**.

The **PC** value used to compute required displacement is **8 bytes** ahead of the current Bcc being executed.

Bcc is used with **address labels** that allows the assembler to compute the required displacement values.

```
      BEQ Skip    ; branch to Skip if Z=1 (EQual to zero)
      :           Z = 0   cc = false
      :
Skip  MOV R1,R0   ; instruction at Skip
```

cc = true, Z = 1

ARM provides **different** conditional branch options.

15 possible conditions is permitted in the condition field (cc) using combinations of the N, Z, V, C flags.

e.g. Bcc	Operation and CC flag conditions	
B or BAL	PC ← PC ± n	Branch Always
BEQ	If Z = 1, PC ← PC ± n	Branch Equal
BVS	If V = 1, PC ← PC ± n	Branch Overflow Set

Flexible conditional branch can be programmed based on outcome of instructions **prior** to Bcc.

The choice of condition (cc) is dependent on whether the test is for a **signed** or **unsigned** computation.

```
      SUBS R0,R0,#1
      BEQ  Skip
Z = 1 :    cc = EQ (Equal to zero)
Skip MOV   R1,R0
```

Suffix	Flags	Meaning	
EQ	Z = 1	Equal	
NE	Z = 0	Not equal	
CS or HS	C = 1	Higher or same, **unsigned**	Unsigned comparison
CC or LO	C = 0	Lower, **unsigned**	Unsigned comparison
MI	N = 1	Negative	
PL	N = 0	Positive or zero	
VS	V = 1	Overflow	
VC	V = 0	No overflow	
HI	C = 1 and Z = 0	Higher, **unsigned**	Unsigned comparison
LS	C = 0 or Z = 1	Lower or same, **unsigned**	Unsigned comparison
GE	N = V	Greater than or equal, **signed**	Signed comparison
LT	N != V	Less than, **signed**	Signed comparison
GT	Z = 0 and N = V	Greater than, **signed**	Signed comparison
LE	Z = 1 and N != V	Less than or equal, **signed**	Signed comparison
AL	Can have any value	Always. This is the default when no suffix is specified.	

R0 0x00000001 (+1)

R1 0xFFFFFFFF (-1)

```
      SUBS R2,R0,R1
      BGT  Else
      :
      :
      :
Else MOV   R1,R0
```

R0 > R1

R0 > R1 (signed compare)

Count Number of 1's

- Count the number of binary 1s in register **R0**.
  - Loop counter R3 is initialized to 32 at the start.
  - SUBS is used to decrement R3 to zero and set the Z flag when that happens.
  - BNE is used to test for Z=0, until Z=1 (i.e. R3=0), it will keep looping back.

```
        MOV   R2,#0          ; clear 1-counter for binary 1s
        MOV   R3,#32         ; set loop counter to 32 times
Loop    MOV   R0,R0,ROR #1   ; rotate right 1 bit
        AND   R1,R0,#1       ; clear all bits except LSB
        ADD   R2,R2,R1       ; add LSB to 1-counter
        SUBS  R3,R3,#1       ; decrement loop counter
        BNE   Loop           ; loop back until R3=0
```

loop back 31 times; Z = 0; Z = 1

  - Loop counter R3 is initialized to **31** at the start.
  - SUBS decrements R3 to negative and sets the N flag when that happens.
  - BPL is used to test for N=0, until N=1 (i.e. R3=-1), it will keep looping back.

```
        MOV   R2,#0          ; clear 1-counter for binary 1s
        MOV   R3,#31         ; set loop counter to 31
Loop    MOV   R0,R0,ROR #1   ; rotate right 1 bit
        AND   R1,R0,#1       ; clear all bits except LSB
        ADD   R2,R2,R1       ; add LSB to 1-counter
        SUBS  R3,R3,#1       ; decrement loop counter
        BPL   Loop           ; loop back until R3=-1
```

loop back 31 times; N = 0; N = 1

TST, CMN, TEQ

06 Other Conditional Test Instructions

- ARM provides several other operators that can be used to influence the conditional test flags.
  - These conditional test instructions do not modify the destination operand.
  - They do not need the "S" suffix to influence the condition code flags (N,Z,V,C).

`CMN R0,R1 ; set (N,Z,C,V) based on R0 + R1`	Compare Negative
`TST R0,R1 ; set (N,Z,C) based on R0 AND R1`	Test Bits
`TEQ R0,R1 ; set (N,Z,C) based on R0 EOR R1`	Test Equivalence

  - The C flag for TST and TEQ can be influence by applying the shift and rotate operations on the source operand (rightmost).

Example

Find Largest Number (FindMax)

Write an assembly language program to :

Find the **largest value** in an integer array and store the result in register **R3**.

The array consists of **10 unsigned** numbers stored starting at address **0x100**.

Largest Value: R3 0x00000007

Number Array

	Address	Memory
[R0] →	0x100	0x00000003
	0x104	0x00000007
	0x108	0x00000004
	0x10C	0x00000002
	:	:
	:	:
	0x120	0x00000005
	0x124	0x00000001

Things to note:

Use correct conditional test for comparing unsigned number.

Use appropriate register indirect to access each array element efficiently.

Set up appropriate count loop to access all 10 numbers

```
        MOV   R0,#0x100      ;setup pointer to first array element
        MOV   R1,#9          ;load 9 into counter register
        LDR   R3,[R0]        ;assume 1st no. in array is current max
Loop
        SUBS  R1,R1,#1
        BNE   Loop
```

Initialisation of registers

Loop Count R1 0x0000009

Temp Reg R2

Current Max R3 0x00000003

Number Array

	Address	Memory
[R0] →	0x100	0x00000003
	0x104	0x00000007
	0x108	0x00000004
	0x10C	0x00000002
	:	:

R0 = Address pointer for current array element.
R1 = Loop counter register
R2 = Temporary register holding current no.
R3 = Current maximum value (i.e. the result).

```
        MOV   R0,#0x100      ;setup pointer to first array element
        MOV   R1,#9          ;load 9 into counter register
        LDR   R3,[R0]        ;assume 1st no. in array is current max
Loop    ADD   R0,R0,#4       ;increment array pointer to next element
        LDR   R2,[R0]        ;get next no. in array
        CMP   R2,R3          ;compare R3 and R2 (i.e. R2-R3)
        BLS   Skip           ;branch if R2 ≤ current max (i.e. R3)
        MOV   R3,R2          ;update current max. in R3 with R2
Skip    SUBS  R1,R1,#1       ;decrement 1 from counter register
        BNE   Loop           ;jump back to Loop if not zero
```

R0 = Address pointer for current array element.
R1 = Loop counter register
R2 = Temporary register holding current no.
R3 = Current maximum value (i.e. the result).

Comparing Signed & Unsigned Values

Appropriate conditional test must be selected based on the number representation used.

For testing **signed** values, use **GT**, **LT**, **GE**, **LE**.

e.g.

```
        SUBS  R1,R1,R2 ;R1 = R1 - R2
        BGE   R1≥R2    ;jump to R1≥R2 if result is positive
          :
R1≥R2   :
```

Destination register gets modified when we try to find out if R1 ≥ R2

For testing **unsigned** values, use **HI**, **LO**, **HS**, **LS**.

e.g.

```
        SUBS  R1,R1,R2 ;R1 = R1 - R2
        BHS   R1≥R2    ;jump to R1≥R2 if R1 higher or equal to R2
          :
R1≥R2   :
```

Use (CMP) instead of (SUBS) to compare values of two operands without affecting the operands.

Comparing a register value (signed) to an immediate value.

```
        CMP   R1,#4          ; test (R1 – 4), where R1 is a signed no.
        BGE   R1≥4           ; branch to R1≥4 if result is positive (i.e. R1 ≥ 4)
          :
R1≥4    :
```

Finding C string terminator (0) in memory pointed to by R0.

```
Loop    LDRB  R1,[R0],#1     ; read mem byte using post-index autoindex
        CMP   R1,#0          ; test ( R1 – 0 )
        BEQ   Found          ; branch to Found if value is 0
        B     Loop           ; keep branching back to start of Loop
Found   :
```

1106 Conditional Execution

- ARM instructions can be conditionally executed based on the CC flags.

ARM code example

```
; C code
if (R0 == 1)
  R1 = 3;
else
  R1 = 5;
```

```
        CMP   R0,#1          ; set CC based on r0 –1
        BNE   ELSE           ; if (R0 == 1)
        MOV   R1,#3          ; then { R1 := 3}
        B     SKIP           ; skip over else code seg
ELSE    MOV   R1,#5          ; else  { R1 := 5}
SKIP    .....                ;
```

  - The conditional execution feature allows us to make the execution of each instruction dependent on the current status of the N, Z, V, C flags.

```
        CMP   R0,#1          ; if (r0 == 1)
Executes if Z = 1
        MOVEQ R1,#3          ; then { r1 := 3}
Executes if Z = 0
        MOVNE R1,#5          ; else  { r1 := 5}
SKIP    .....                ;
```

Modular Programming

:
Call Module 1
Call Module 3
Call Module 1

Module 1
Module 2
Module 3

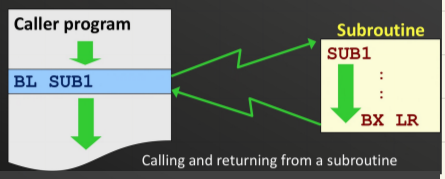
Loose Coupling
- data is independent
  - No global varibles

Strong Modularity
- Single local task

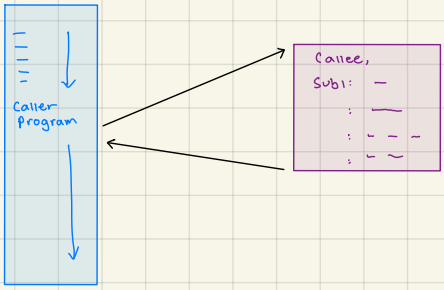
Subroutines

- Calling and returning from a subroutine
  - To go to subroutine (SUB1): **BL SUB1**
  - To return to caller program: **BX LR**

BL: branch with link
BX: branch and exchange



Calling and returning from a subroutine



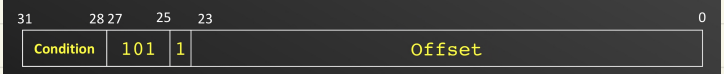
Why BL and BX and not B?

- Main program branches to subroutine:
  - Can be done with B ➔ what is the draw back?
  - B overwrites value in PC ➔ oldPC value in main program is LOST
  - Maybe add another branch at the end of subroutine ➔ need to know the exact mem location during compilation, not an effective approach

→ The caller program needs to have the B Link inside it, making this non-modular

Branch with Link (BL)

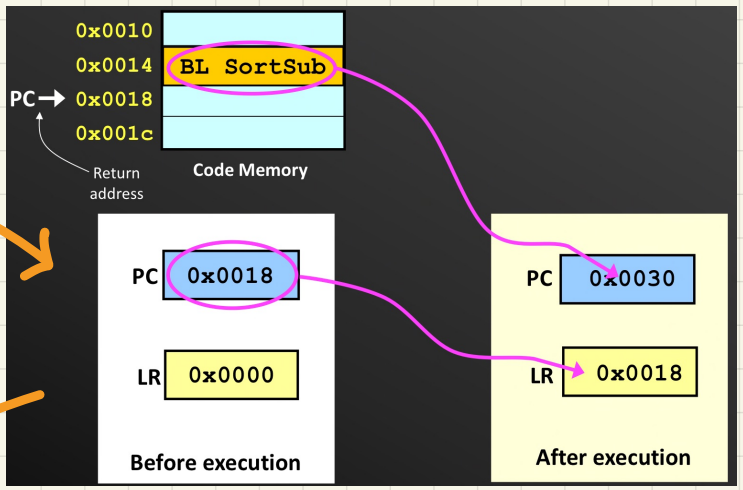
- **BL** used to make subroutine call
- Return address (PC contents + 4) is stored in the link register (R14)



- We can also conditionally make a functional call (more of this later)
- **BL** used to make subroutine call

```
Subroutine Call
BL SortSub
```

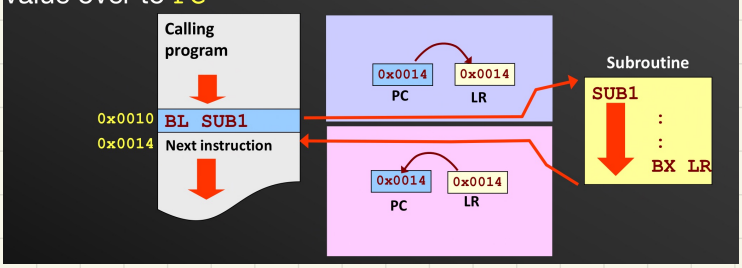
- Execution sequence:
- Return address (PC contents + 4) is stored in the link register (R14)
- The subroutine address is stored to the PC



→ Stores PC inside LR
↳ Populate PC with module address

BX lr returns from subroutine

lr contains the return address, the instruction copies the value over to **PC**



BL SUB1
BX LR

Note: VisUAL does not support `bx lr`

- Use `MOV PC, LR`

Parameter Through Reg

Parameter Passing using Registers

- Parameters are placed into the register before calling the subroutine
- **Number** of parameters passed are **limited** to the **available registers**
  - Useful when number of parameters are **small**
  - Not all **R0-R12** are preferred to pass parameters

Calling convention

R0-R3	R4-R11
Can be used to pass argument values to return values from subroutine Subroutine can modify values	Used to hold local variables Not for passing arguments Must be preserved in the subroutine

R12: Scratchpad register, does not need to be preserved
Can be used sometimes as return register → can be either

R13 (Stack Pointer)
R14 (Link Register)
R15 (Program Counter)

NZVC	CPSR

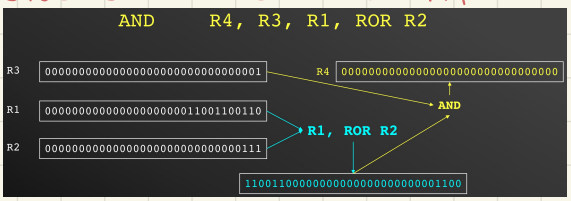
Example: Bit Counting Subroutine

- Write a subroutine to:
  - Count the number of "1" bits in a word.
  - Return result in register **R0**.
  - **Design considerations:**
  - How do we transfer the word into the subroutine?
    - Put the word into a **register**, which can then be accessed within the subroutine (e.g. register **R1**).
  - How do we check if each individual bit is a "1" or a "0"?

```
Count1s  EOR     R0,R0,R0      ;Clear R0
         ADD     R2, R0, #32   ; Set counterR2 with 32
         ADD     R3, R0, #1    ; Set R3 with 1
Loop     AND     R4, R3, R1, ROR R2        ; ?
         ADD     R0,R0,R4      ; Add the lsb of R0
         SUBS    R2,R2,#1      ; decrement counter by 1
         BNE     Loop          ; loop if not zero
         MOV     PC, LR        ; same as bx lr
```

Value passed in R1, return value in R0 VisUAL does not support `bx lr`

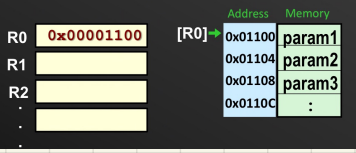
Add R3 & the rotation of R1 by the value of R2 bits to R4



Parameter Thru Mem

A region in memory is treated like a mailbox and is used by both the calling program and subroutine.

- Parameters to be passed are gathered into a **block** at a predefined memory location
- The start address of the memory block is passed to the subroutine via an **address register**.
- Useful for passing **large number of parameters**.



Example: Lower to Upper Case Subroutine

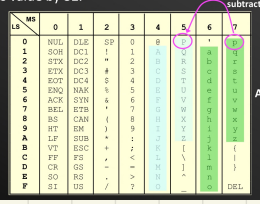
- Write a subroutine to:
  - To convert an ASCII string from lower to upper case.
  - The string is terminated by a NULL character (**0x00000000**).
  - The start address of the string is passed via **R0**.
  - A segment of the calling program shows how the parameter is setup and the subroutine called:

```
;Calling program
   :
MOV   R0,#0x100   ;move start addr. of string to R1
BL    Lo2Up       ;branch to Lo2Up subroutine
   :
```

Address	Memory
0x100	"a"
0x104	"p"
0x108	"p"
0x10C	"l"
0x110	"e"
0x114	0x000
0x118	:

How to convert an ASCII character from lower to upper case?

- Check that the character's value is between 'a' and 'z'.
- If so, subtract its value by 32.



```
Lo2Up   STMFD   SP!,{r0,r1}   ;save registers used within subroutine
Loop    LDR     R1,[R0],#4   ;get current char from string in memory
        CMP     R1,#0        ;Compare with NULL
        BEQ     Done         ;if NULL char. branch to Done
        CMP     R1,#0x061    ;compare with lower limit "a"
        BLT     Loop         ;if smaller than "a", do not convert
        CMP     R1,#0x07A    ;compare with upper limit "z"
        BGT     Loop         ;if greater than "z" do not convert
        SUB     R1,R1,#32    ;convert to upper case by subtracting 32
        STR     R1,[R0,#-4]  ;write modified char back to string in memory
        B       Loop         ;branch back to Loop
Done    LDMFD   SP!,{r0,r1}  ;restored saved registers before returning
        MOV     PC,LR        ;return from subroutine
```

Always

Z=1

Note: Subroutine modifies **R0** & **R1** but restores their original contents before returning. This produces a **transparent subroutine** that does not effect the proper operation of calling program

Efficient Memory Solution

```
Lo2Up   STMFD   SP!,{r0,r1}   ;save registers used within subroutine
Loop    LDRB    R1,[R0],#1   ;get current char from string in memory
        CMP     R1,#0        ;Compare with NULL
        BEQ     Done         ;if NULL char. branch to Done
        CMP     R1,#0x061    ;compare with lower limit "a"
        BLT     Loop         ;if smaller than "a", do not convert
        CMP     R1,#0x07A    ;compare with upper limit "z"
        BGT     Loop         ;if greater than "z" do not convert
        SUB     R1,R1,#32    ;convert to upper case by subtracting 32
        STRB    R1,[R0,#-1]  ;write modified char back to string in memory
        B       Loop         ;branch back to Loop
Done    LDMFD   SP!,{r0,r1}  ;restored saved registers before returning
        MOV     PC,LR        ;return from subroutine
```

Note: Subroutine modifies **R0** & **R1** but restores their original contents before returning. This produces a **transparent subroutine** that does not effect the proper operation of calling program

STACK

Push Data to the Stack

- Writing to stack can be done using **STR** instruction
  - Need to also increase the stack pointer before storing
- Syntax: `STR R0, [SP,#-4]!`

Basically, Descending Full stack

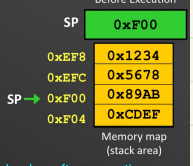
STMFD !

Pushing Registers to Stack

- E.g., want to push R0 and R1

Method 1:

```
STR R0, [SP, #-4]!
STR R1, [SP, #-4]!
```



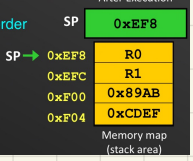
Method 2:

Use Stack pointer and write back updated value after operation

```
STMFD SP!,{R1,R0}
```

Write registers in descending order

Store multiple registers fully descending



store multiple

```
STMFD SP!, {list of registers}
```

Load multiple fully decending

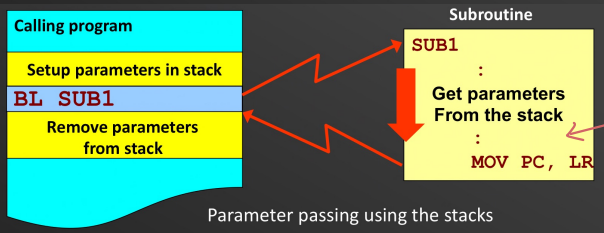
```
LDMFD SP!, {list of registers}
```

`STMFD SP!,{R1,R0}` Sp decreases to 0

`LDMFD SP!,{R1,R0}` Sp increases to high

Most **general** method of parameter passing – no registers needed, supports recursive programming.

Large numbers of parameters can be passed as long as stack does not overflow.



Parameter passing using the stacks

If you add any items, delete those added within the stack

Parameters pushed to the stack must be **removed** by the calling program immediately after returning from subroutine.

If not, repeated pushing of parameters to the stack will lead to a stack overflow.

Sum from 1 to N

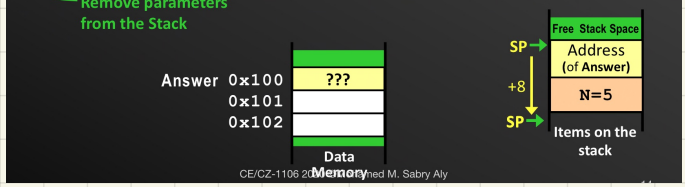
- Write a subroutine to:
  - Sum the positive numbers from 1 to **N**, where **N** is a value passed to the subroutine.
  - The computed sum should be directly updated to a memory variable **Answer**, whose address is **0x100**.
  - All parameters are to be passed via the **stack**.

Solution:

- Push two parameters on stack, the value of **N** and **address** of memory variable **Answer**.

```
        :                       ;find the sum of (1+2+3+4+5), where N=5
Parameters setup
        MOV   R1, #5            ;Set R0 with #5
        MOV   R0, #0x100        ;Set R1 with the address
        STMFD SP!, {r1,r0}      ;push R0&R1 to stack
        BL    Sum1N             ;call subroutine Sum1N
        ADD   SP,SP,#8          ;add 8 to pop the two parameters from stack
        :
```

Remove parameters from the Stack



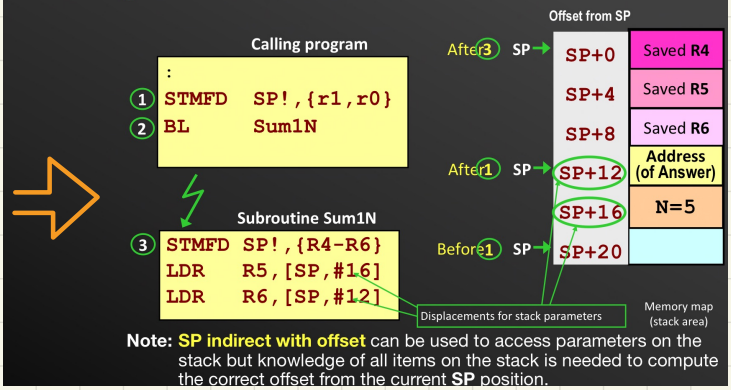
Sum from 1 to N Subroutine
Possible Solution

R6 Answer Addr
R5 N

```
Sum1N   STMFD  SP!,{R4,R5,R6}   ;save registers to stack
Retrieve stack parameters
        LDR    R5,[SP,#16]      ;Load N from stack to R5
        LDR    R6,[SP,#12]      ;Load Answer's From stack
        MOV    R4,#0            ;clear summation register R0 to 0
Loop    ADD    R4,R4,R5         ;add current value in R5 to R4
        SUBS   R5,R5,#1         ;decrement current value in R4 by 1
Z=0     BNE    Loop             ;jump back to Loop if R4 not yet zero
Z=1     STR    R4,[R6]          ;write sum to Answer
        LDMFD  SP!,{R4,R5,R6}   ;restored saved registers
        MOV PC,LR               ;return from subroutine
```

Note: The subroutine needs three registers. **R4** to compute the sum from 1 to N. **R6** to be an address pointer to the memory variable **Answer** where the results will written to. **R5** holds the value of N, which is decremented by 1 after each loop till it reaches 0.

Accessing Stack Parameters



Note: **SP indirect with offset** can be used to access parameters on the stack but knowledge of all items on the stack is needed to compute the correct offset from the current **SP** position.

Transparent Subroutines

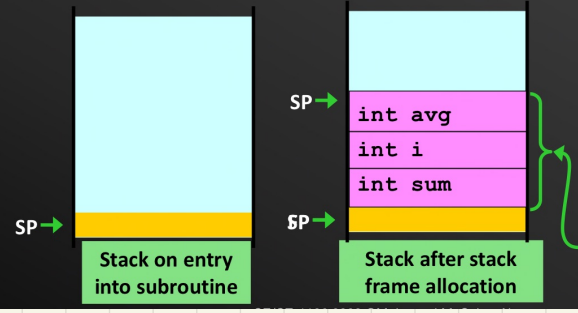
- A transparent subroutine will **not affect any CPU resources** used by the program calling it.
- To achieve this, all local registers used by the subroutine (R4-R11) must be **saved on the stack** on entry and **restored from stack** before returning.

```
SUB1 STMFD SP!,{R4-R7}     ; save R4 to R7 to stack
     :
     :                     ; registers R4 to R7 are
     :                     ; used in subroutine
     LDMFD SP!,{R4-R7}     ; restore R4 to R7 from stack
     MOV PC,LR             ; return to calling program
```

✱ Initiating Local Varibles

Stack Frame

- Stack frame of **N** words is created on the system stack immediately on entry into a subroutine.
  - Memory space within the stack frame can be accessed using with a **frame pointer (FP)** or the stack pointer **(SP)**.
  - Frame pointer in ARM can be any register
    - General convention **FP** is **R11**




```
int Mean()
{
 int avg, i;
 int sum = 0;
 :
```

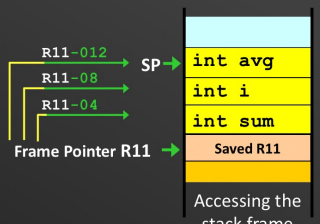
function Mean

Stack frame of N=3 words created for three word-sized local variables in functions **Mean**.

R11 FRAME POINTER

Accessing Stack Frame Variables
Using the Frame Pointer

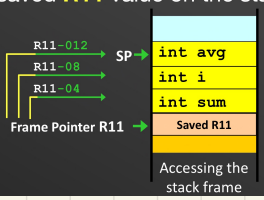
- Consider the use of a frame pointer register **R11**.
  - Original contents of **R11** is saved on the stack before it is used as the frame pointer.
  - Frame pointer (**R11**) now points to the saved **R11** and a stack frame is created by adding frame size **4N** to **SP**.
  - **R11** is the frame reference and an appropriate negative displacement from **R11** can be used to access any stack frame variable.
  - When exiting the subroutine, the stack frame can be destroyed by adding **4N+4** to **SP** and copying the saved **R11** value on the stack back into **R11**.



When exiting the subroutine, the stack frame can be destroyed by adding **4N+4** to **SP** and copying the saved **R11** value on the stack back into **R11**.

For N=3

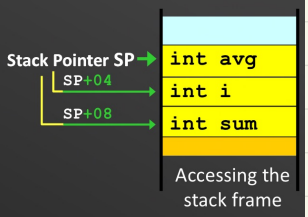
```
ADD SP, SP, #16
LDR R11,[R11]
```



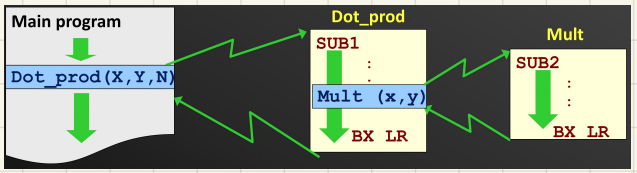
Using the Stack Pointer

A more efficient approach is to use the stack pointer (**SP**) itself.

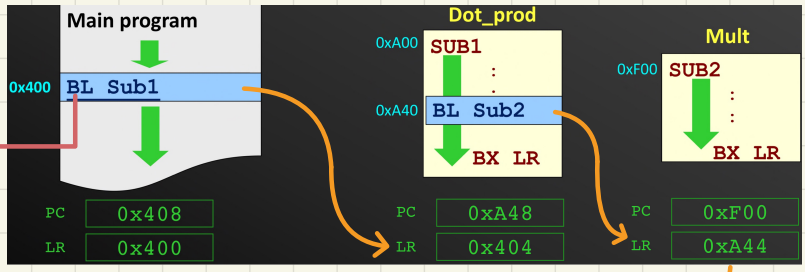
- A stack frame is created by adding frame size **4N** to **SP**.
- **SP** is used as a reference to access all local variables.
- Appropriate **positive** displacements from **SP** is used to access any of the stack frame variables.
- **Pro** - This method is more efficient because there is no need to setup a frame pointer.
- **Con** – More restrictive as system stack cannot be used within subroutine without changing the reference **SP**.



RECURSIVE



Problem: PC may be lost, can't return



stuck on 0xA44

∴ Need to store somewhere safe
AKA; System Stack

```
DotProd     STMFD       SP!,{R4-R7}     ;Store regs to stack
            LDR         R4 ,[SP,#28]    ;Read location of X→#28 cuz
            LDR         R5 ,[SP,#24]    ;Read location of Y R4-R7 stored
            LDR         R6 ,[SP,#20]    ;Read arrays length
            MOV         R7, #0          ; Clear R7 (Sum)
Loop1       LDR         R0 ,[R4],#4     ; Get X[i]  } to multiply
            LDR         R1 ,[R5],#4     ; Get Y[i]  }
            STR LR save ← LR ,[SP,#-4]! ; Push Link register to SP
            BL          Mult            ; Call Mult Subroutine
            LDR LR restore ← LR ,[SP],#4; Pop Link Register from SP
            ADD         R7,R7,R12       ; Add the product to R7
            SUBS        R6,R6,#1        ; Reduce the counter by 1
            BNE         Loop1           ; not 0 then repeat
            LDR         R4, [SP,#16]    ; read destination address
            STR         R7, [R4]        ; Store in destination address
            LDMFD       SP!,{R4-R7}     ; Restore registers
            MOV         PC, LR          ; same as bx lr
```

if ≠0

Iteratively add R0 to R12 for R1 times

```
Mult        MOV         R12,#0          ;Clear R12
Loop        ADD         R12,R12,R0      ;Add R0 to R12
            SUBS        R1,R1,#1        ;Decrement R1 with 1
            BNE         Loop
            MOV         PC, LR          ; same as bx lr
```

Alternatively: Front, add LR into stack at front

Second, directly put PC into LDMFD

```
DotProd   STMFD   SP!,{R4-R7,LR}   ;Store regs to stack
          LDR     R4 ,[SP,#32]    ;R
          LDR     R5 ,[SP,#28]    ;R
          LDR     R6 ,[SP,#24]    ;R
          MOV     R7, #0          ;
Loop1     LDR     R0 ,[R4],#4     ;
          LDR     R1 ,[R5],#4     ;

          BL      Mult            ;

          ADD     R7,R7,R12       ;
          SUBS    R6,R6,#1        ;
          BNE     Loop1           ;
          LDR     R4, [SP,#20]    ;
          STR     R7, [R4]        ;
          LDMFD   SP!,{R4-R7,PC}  ;
```

Why is this correct?

We now push 5 registers to stack
Need to update the offsets

SP →	R4
	R5
	R6
	R7
	LR
SP+20 →	0x300
SP+24 →	20
SP+28 →	0x200
SP+32 →	0x100

Recursive Subroutine

- A recursive routine calls itself within its own body.
- Recursion is an elegant way to solve algorithms or mathematical expressions that have **systematic repetitions**. (e.g. factorial **n! = n×(n-1)×(n-2)….2×1**)

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

Recursive Code Example

```
Recur   STMFD SP!,{…,LR}        ; Subroutine Recur
        Condition to terminate the subroutine (branch to Done)
          :
        BL Recur                ; call Recur with Recur routine
          :
Done    LDMFD SP!,{…,LR}
        BX LR                   ; return to calling program
```

```
Fib       STMFD     SP!,{R4-R6,LR}    ;Store regs to stack
          LDR       R4 ,[SP,#16]      ;Read Value of n
          CMP       R4 ,#1            ;Compare with 1 (if n==1)
          MOVEQ     R12,#0            ;Assign result with 0
          CMP       R4 ,#2            ;Compare with 2 (if n==2)
          MOVEQ     R12,#1            ; Assign result with 1
          BLE       Done              ; if n less than equal 2 finish
          SUB       R5,R4,#1          ; Get n-1
          STMFD     SP!,{R5}          ;Push n-1 to stack
          BL        Fib               ; calculate Fib(n-1)
          LDMFD     SP!,{R5}          ; Pop from stack
          MOV       R6,R12            ;Save result in temp register
          SUB       R5,R4,#2          ; Get n-2
          STMFD     SP!,{R5}          ;Push n-2 to stack
          BL        Fib               ; calculate Fib (n-2)
          LDMFD     SP!,{R5}          ; Pop from stack
          ADD       R12,R12,R6        ; Calculate Fib(n-1) + Fib(n-2)
Done      LDMFD     SP!,{R4-R6,LR}    ; Restore registers
          MOV       PC, LR            ; same as bx lr
```

Flow Control

IF Statements

- How is the **IF** construct implemented?
- **Imitating** the high-level test condition does not result in very efficient assembly-level implementation.

```
if (a > b)
  {S1}
```

Convert to assembly code

```
      CMP a,b
      BGT DoIF
      B Skip
DoIF  {S1}
Skip
```

- High-level test condition is **reversed** in assembly-level to avoid the need for an additional unconditional jump.

```
if (a > b)
  {S1}
```

Convert to assembly code

```
      CMP a,b
      BLE Skip
      {S1}
Skip
```

→ optimize by 1/3 by taking negation

Note: The reverse condition test of **HS** is **LO**, reverse of **LT** is **GE**

Conditional Execution

In the 32-bit ARM ISA, instructions can be conditionally executed based on the CC flags.

Consider the following 32-bit ARM code segment.

```
; C code
if (r0 == 1)
 r1 = 3;
```

```
      CMP   r0, #1      ; set CC based on r0 –1
      BNE   Skip        ; if (r0 == 1)
      MOV   r1, #3      ; then { r1 := 3}
SKIP  ......            ;
```

→ move if equal

It can be replaced using conditional execution instructions.

```
      CMP   r0, #1      ; if (r0 == 1)
      MOVEQ r1, #3      ; then { r1 := 3}
SKIP  ......            ;
```

Find Largest Number

```
      MOV   R0,#0x100     ;setup pointer to first array element
      MOV   R1,#9
      LDR   R3,[R0]
Loop  LDR   R2,[R0,#4]!
      CMP   R2,R3         ;compare R3 and R2 (i.e. R2-R3)
      BLO   Skip          ;branch if R2 <  current max (i.e. R3)
      MOV   R3,R2         ;update current max. in R3 with R2
Skip  SUBS  R1,R1,#1      ;decrement 1 from counter register
      BNE   Loop          ;jump back to Loop if not zero
```

```
if R2 >= R3   // R2 larger or equal to current max
{
  R3 = R2;    // then update R3 with new max R2
}
```

R0 = Address pointer for current array element.
R1 = Loop counter register
R2 = Temporary register holding current no.
R3 = Current maximum value (i.e. the result).

Without reverse

```
      CMP     R2,R3
      BHS     Assign
      SUBS    R1,R1,#1
      BNE     Loop
Assign MOV    R3,R2
      SUBS    R1,R1,#1
      BNE     Loop
```

Redundant

```
      MOV   R0,#0x100     ;setup pointer to first array element
      MOV   R1,#9
      LDR   R3,[R0]
Loop  LDR   R2,[R0,#4]!
      CMP   R2,R3         ;compare R3 and R2 (i.e. R2-R3)
      MOVGE R3,R2         ;update current max. in R3 with R2
      SUBS  R1,R1,#1      ;decrement 1 from counter register
      JNE   Loop          ;jump back to Loop if not zero
```

```
if R2 >= R3   // R2 larger or equal to current max
{
  R3 = R2;    // then update R3 with new max R2
}
```

Find Largest Number, and Store Its Index

No conditional execution

```
R3= X[0];
R4=0;
R0= &X[0]
For (R1=9;R1>0;R1--){
        R0+=4;R2=X[R0];
        if(R2>=R3){
                R3=R2;
                R4=10-R1;
        }
}
```

```
        MOV   R0,#0x100     ;setup pointer to first array element
        MOV   R1,#9         ;load 9 into counter register
        MOV   R4,#0         ;load 0 into index_max register
        LDR   R3,[R0]       ; 1st no. in array is current max
Loop    LDR   R2,[R0,#4]!   ;get next no. in array
        CMP   R2,R3         ;compare R3 and R2 (i.e. R2-R3)
        BLO   Skip          ;branch if R2 <  current max (i.e.
        MOV   R3,R2         ;update current max. in R3 with R2
        RSUB  R4,R1,#10     ;Store the index in R4
Skip    SUBS  R1,R1,#1      ;decrement 1 from counter register
        BNE   Loop          ;jump back to Loop if not zero
```

Ave of 5.4 execution

Faster!

With conditional execution

```
R3= X[0];
R4=0;
R0= &X[0]
For (R1=9;R1>0;R1--){
        R0+=4;R2=X[R0];
        if(R2>=R3){
                R3=R2;
                R4=10-R1;
        }
}
```

```
        MOV   R0,#0x100     ;setup pointer to first array element
        MOV   R1,#9         ;load 9 into counter register
        MOV   R4,#0         ;load 0 into index_max register
        LDR   R3,[R0]       ; 1st no. in array is current max
Loop    LDR   R2,[R0,#4]!   ;get next no. in array
        CMP   R2,R3         ;compare R3 and R2 (i.e. R2-R3)
        MOVGE R3,R2         ;update current max. in R3 with R2
        RSUBGE R4,R1,#10    ;Store the index in R4
        SUBS  R1,R1,#1      ;decrement 1 from counter register
        BNE   Loop          ;jump back to Loop if not zero
```

Ave of 6 executions

Slower!

IF ELSE

IF-ELSE Statements

- How is the **IF-ELSE** construct implemented?
- Combination of conditional and unconditional jumps used for the **IF-ELSE** construct.
- Reversing high-level test condition does not improve efficiency unless
- the ELSE code segment {S2} is more likely to execute.

```
        CMP a,#3
        BNE DoElse
        {S1}
        B Skip
DoElse  {S2}
Skip    :
```

Reversing test condition

If a==3, 4 loop
a≠3, 3 loop

```
if (a == 3)
  {S1}
else
  {S2}
```

Convert to assembly code

```
        CMP a,#3
        BEQ DoIf
        {S2}
        B Skip
DoIf    {S1}
Skip    :
```

If a==3, 3 loop
a≠3, 4 loop

IF-ELSE implementation in low-level assembly

```
; C code
if (r0 == 1)
 r1 = 3;
else
 r1 = 5;
```

```
        CMP   r0, #1        ; set CC based on r0 –1
        BNE   ELSE          ; if (r0 == 1)
        MOV   r1, #3        ; then { r1 := 3}
        B     SKIP          ; skip over else code seg
ELSE    MOV   r1, #5        ; else  { r1 := 5}
SKIP    .....               ;
```

Alternatively,

With conditional execution

```
        CMP r0, #1          ; if (r0 == 1)
        MOVEQ r1, #3        ; then { r1 := 3}
        MOVNE r1, #5        ; else  { r1 := 5}
SKIP    .....               ;
```

→ only 3!

Compound AND Conditions

- How are compound AND conditions handled?
- Logical AND can bind multiple basic relational conditions.

e.g.
```
if ((a == b) && (b > 0)) {S1}
```
order of compound AND test

- Compilers resolve compound conditions into simpler ones.

```
    if (a != b) then Skip
    if (b <= 0) then Skip
       {S1}
Skip    :
```

- Elementary conditions bound by the logical AND are tested from **left-to-right**, in the order given in the C program.
- The first false condition means the remaining conditions are not computed. This is called the **fast Boolean operation**.
- Keep the **least likely** condition **leftmost** in your program for more efficient execution.

AND negates condition, left to right ....

also, leave the least likely condition @ Left!

Possible Problems

```
if ((a == b) && (b > 0)) {S1}
```

What if a>b and b>0

```
CMP     R1,R2  ;R1<-a, R2<-b
CMPEQ   R2,#0
XXXGT   XXXX   ; S1
```

1) If a>b, blue will "fail", and not trigger CMP EQ R2, #0.

However, the 3rd step will still trigger under the first greater flag!

Better Way

```
    CMP   R1, R2
    BNE   Skip
    CMP   R2, #0
    XXXGT  XXX   ; S1
Skip - - -
     - -
```

Compound OR Conditions

- How are compound OR conditions handled?

e.g.
```
if ((a == 1) || (a == 2)) {S1}
```

- Most compilers eliminate an unconditional jump at the end of the compound OR series by **reversing** the **last conditional** test.

```
      if (a == 1) then DoIf
      if (a != 2) then Skip
DoIf  {S1}
Skip    :
```

- The conditional test that is **most likely** to be **true** should be kept leftmost.

Most likely @ beginning

Compound Condition Example

```
if ((a == 1) || (a == 2)) {S1}
```

With conditional assignment
S1 needs to be replicated.
Not suitable for multiple instructions

```
CMP     R1,#1  ;R1<-a
XXXEQ   XXXX   ; S1
CMPNE   R1,#2
XXXEQ   XXXX   ; S1
```

A more holistic approach

```
      CMP     R1,#1
      BEQ     doIF
      CMPNE   R1,#2
doIF  XXXEQ   XXXX   ; S1
```

4 always

either 3 / 4

Efficient Bit Hack

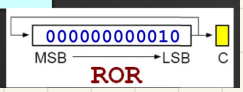
Branchless Logic

- Branchless logic avoid using conditional jump instructions when implementing logical constructs.
- `Bcc` instructions may result in costly **flushing** operations when the wrong next instruction is pre-fetched into the CPU's pipeline.
- How is branchless logic implemented?
- **Exploit arithmetic relationship** to transform the test condition into the corresponding desired outcome. Can only be applied in special cases and desired outcomes are usually Boolean values.

e.g.

```
if ((X & 2) == 2)
  X = true;
else
  X = false;
```

```
AND R1,R1,#0x02
ROR R1,R1,#1
```



- **Conditional execution** can be used to avoid branching.

SWITCH

Switch Statement

- How is the **SWITCH** construct implemented?

The assembly code produced varies between compilers and depends on the nature and range of the case values.

Two different SWITCH scenarios are examined:

```
switch(x) {
 case 0 : {S0};
          break;
 case 1 : {S1};
          break
 case 2 : {S2};
          break;
 case 3 : {S3};
}
```

Running values narrow range

```
switch(x) {
 case 1    : {S0};
             break;
 case 10   : {S1};
             break
 case 100  : {S2};
             break;
 case 1000 : {S3};
}
```

Random values wide range

Switch – Running & Narrow Values

- If cases are consecutive narrow range values, a **Jump Table** is used to avoid testing each case in turn.

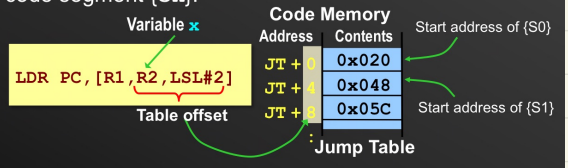
```
switch(x)
{
case 0:
 {S0};
 break;

case 1:
 {S1};
 break;

case 2:
 {S2};
 break;

case 3:
 {S3};
}
```

- Jump table contains list of **start addresses** for the code segments that is associated with each case values.
- Var **x** on which the switch is decided, acts as an **offset** into the table to access the corresponding start address.
- Start address is loaded into **PC** to execute the required code segment {S**x**}.



Note: Time taken is on average less than the equivalent if-else-if cascade and is independent of number of cases in the switch construct.

Jump Table Example

```
switch(x)
{
case 0:
 G='F';
 break;

case 1:
 G='D';
 break;

case 2:
 G='C';
 break;

case 3:
 G='B';
}
```

Cascaded if-else

```
      LDR R1, [R2]
      CMP R1, #0
      BEQ C1
      CMP R1, #1
      BEQ C2
      CMP R1, #2
      BEQ C3
      MOV R0, #66 ;'B'
RES   STR R0, [R2,#4]
      .
      .
      .
C1    MOV R0, #70 ;'F'
      B RES
C2    MOV R0, #68 ;'D'
      B RES
C3    MOV R0, #67 ;'C'
      B RES
```

Jump Table

```
      LDR R1, [R2]
      ADR R3, TBL
      LDR PC, [R3,R1, LSL #2]
RES   STR R0, [R2,#4]
      .
      .
      .
0xA00 C1  MOV R0, #70 ;'F'
      B RES
C2    MOV R0, #68 ;'D'
      B RES
C3    MOV R0, #67 ;'C'
      B RES
```

Address	Contents
TBL + 0	0xA00
TBL + 4	0xA08
TBL + 8	0xA10
:	

Jump Table

add R1 multiplied by 4 to go to eq^ address

Jump Table Example (Default)

```
switch(x)
{
case 0:
 G='F';
 break;

case 1:
 G='D';
 break;

case 2:
 G='C';
 break;

case 3:
 G='B';
Default:
G='X';
}
```

Cascaded if-else

```
      LDR R1, [R2]
      CMP R1, #0
      BEQ C1
      CMP R1, #1
      BEQ C2
      CMP R1, #2
      BEQ C3
      CMP R1,#3
      BEQ C4
      MOV R0, #88 ;'X'
RES   STR R0,[R2,#4]
      .
      .
C1    MOV R0, #70 ;'F'
      B RES
C2    MOV R0, #68 ;'D'
      B RES
C3    MOV R0, #67 ;'C'
      B RES
C4    MOV R0, #66; 'B'
      B RES
```

Jump Table

```
      LDR R1, [R2]
      MOV R0, #88; 'X'
      CMP R1, #3
      BGT RES
      ADR R3, TBL
      LDR PC, [R3,R1, LSL #2]
0xA00 RES  STR R0, [R2,#4]
      .
C1    MOV R0, #70 ;'F'
      B RES
C2    MOV R0, #68 ;'D'
      B RES
C3    MOV R0, #67 ;'C'
      B RES
```

Address	Contents
TBL + 0	0xA00
TBL + 4	0xA08
TBL + 8	0xA10
:	

Jump Table

JUMP

Switch – Random & Wide Values

- If cases are random wide range values, a **fork algorithm** is used to speed up the average search time and avoid testing every case (e.g. when **x** = 1000).
- Due to the wide value spread, the **jump table size** will be **too large**. A cascade of if-else-if comparisons is more efficient.

```
switch(x)
{
case 1:
 {S0};
 break;

case 10:
 {S1};
 break;

case 100:
 {S2};
 break;

case 1000:
 {S3};
 break;
}
```

```
if(x==1)
 {S0};
else if(x==10)
 {S1};
else if(x==100)
 {S2};
else if(x==1000)
 {S3};
```

standard if-else-if implementation

```
if(x<=10) {
   if(x==1)
    {S0};
   else if(x==10)
    {S1};
}
else {
   if(x==100)
    {S2};
   else if(x==1000)
    {S3};
}
```

forked if-else-if implementation

→ Change from $\Theta(n)$ to $\Theta(\log n)$

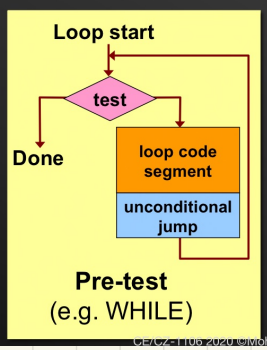
Split

Loops

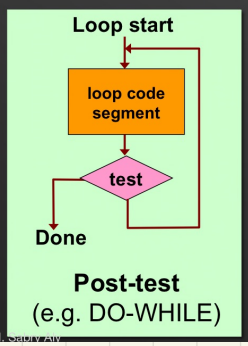
Loop constructs are distinguished by the position of their conditional test.

Pre-test loop may **never execute** its loop code segment.

Post-test loop executes the loop segment **at least once**.

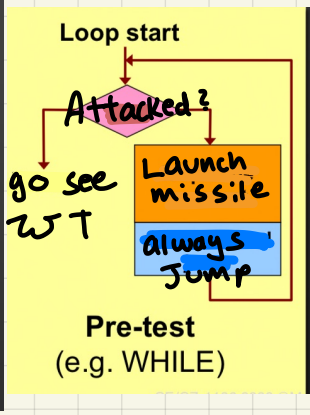


Pre-test (e.g. WHILE)

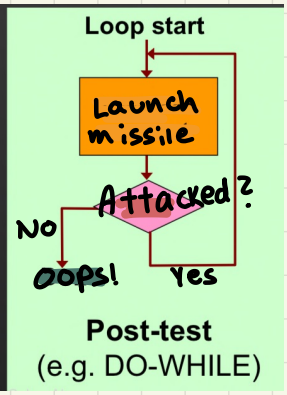


Post-test (e.g. DO-WHILE)

LOOP



Pre-test (e.g. WHILE)



Post-test (e.g. DO-WHILE)

WHILE Implementation

- Implementation of the **WHILE** loop constructs:
- This is an example of a **pre-test** loop.
- If the condition (**VarX > 0**) is false, the loop segment is not executed at all.

```
WHILE (VarX > 0)
{
   Loop segment
}
```

```
Back  CMP "VarX",#0
      BLE Exit
       :
       :   } Loop segment
       :
      B Back
Exit  :
```

VarX<=0

Implementation of WHILE in ARM assembly language

Note: **VarX** is variable. you will need to load it to a register.

DO-WHILE Implementation

- Implementation of the **DO-WHILE** loop constructs
- This is an example of a **post-test** loop.
- The loop segment is executed at least once before condition is tested.
- Post-test loop construct is **more efficient** than the pre-test as there is no need for an additional unconditional jump.

```
DO {
   Loop segment
} WHILE (VarX > 0)
```

```
Back  :
      :   } Loop segment
      :
      CMP "VarX",#0
      BGT Back
      :
```

Implementation of DO- WHILE in ARM assembly language

Note: **VarX** is variable. you will need to load it to a register.

FOR Implementation

- Implementation of **FOR** loop constructs:
- The FOR loop is a **pre-test** loop that evaluates the condition first before executing loop segment.
- If loop segment is executed and count **N** is not used in loop segment, some optimizing compilers implement the **FOR** loop using a **post-test** with **decrement** & **test for zero**.

```
FOR (N=0; N<5; N++)
{
   Loop segment (x5)
}
```

```
      MOV R0,#0
Back  CMP R0,#5
      BGE Exit
       :
       :   } Loop segment
       :
      ADD R0,R0,#1
      B   Back
Exit  :
```

N==5

Implementation of FOR in ARM assembly language

Example: Summation 1 to N

```
Sum =0;
for(i=0;i<=N;i++)
    Sum=Sum+i;
```

Pre-test

```
        MOV     R2,#0
        MOV     R0,#0
Loop    CMP     R2,#N
        BGT     END
        ADD     R0,R0,R2
        ADD     R2,R2,#1
        B       Loop
END     STR     R0,["sum"]
```

Post-test

```
        MOV     R2, #N
        MOV     R0,#0
Loop    ADD     R0,R0,R2
        SUBS    R2,R2,#1
        BNE     Loop
        STR     R0,["sum"]
```

Instruction Encoding

Overall Instruction Format

- Arithmetic & Logic instructions
  - ADD, ADC, SUB, SBC, RSB, AND, EOR, RSC, ORR, BIC

XXXCCS Rd, Rs1, Rs2 , {shft #}

31–28	27–25	24–21	20	19–16	15–12	11–7	6–5	4	3–0
Condition (loop ect.)	000	Inst. type (Instruction Type)	S	Rs1	Rd (Destination)	Shift size	shft	0	Rs2

XXXCCS Rd, Rs1, Rs2 , {shft Rshift}

31–28	27–25	24–21	20	19–16	15–12	11–8	7	6–5	4	3–0
Condition	000	Inst. type	S	Rs1	Rd	Rshift	0	shft	1	Rs2

XXXCCS Rd, Rs1, #immediate (rotated)

31–28	27–25	24–21	20	19–16	15–12	11–8	7–0
Condition	001	Inst. type	S	Rs1	Rd	#rot	8-bit immediate

ADD R0,R1,R2,LSL #5

31–28	27–25	24–21	20	19–16	15–12	11–7	6–5	4	3–0
1110 (No condition)	000	0100	0 (no flag)	0001	0000	00101	00	0	0010

ADD R0,R1,R2

31–28	27–25	24–21	20	19–16	15–12	11–7	6–5	4	3–0
1110	000	0100	0	0001	0000	00000	00	0	0010

SUBS R4, R3, R2, LSR R5

31–28	27–25	24–21	20	19–16	15–12	11–8	7	6–5	4	3–0
1110 (No condition)	000	0010	1	0011	0100	0101	0	01	1 (Shift with register)	0010

RSBEQS R5,R3,#20

31–28	27–25	24–21	20	19–16	15–12	11–8	7–0
0000 (If Equal)	001	0011	1	0011	0101	0000	0001 0100

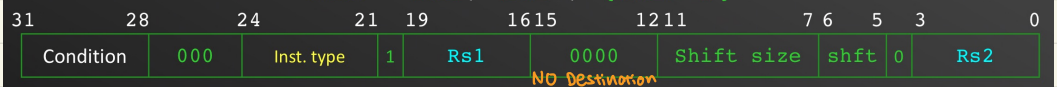
('immediate value')

ARITHMETIC & LOGICAL

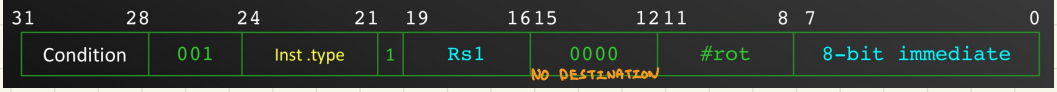
Overall instruction format

- Comparison instructions
  - TST, TEQ, CMP, CMN

XXXCC Rs1, Rs2 , {shft #}



XXXCC Rs1, #immediate (rotated)

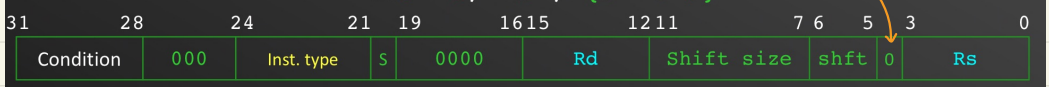


Overall instruction format

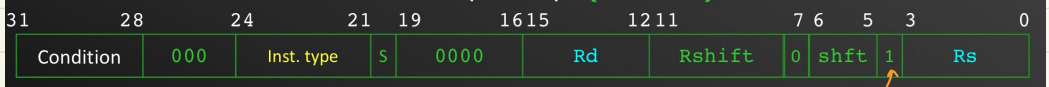
- MOV instructions
  - MOV, MOVN

Shift w/o register

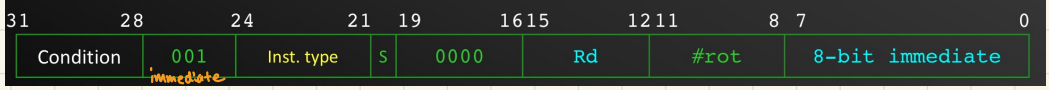
XXXCCS Rd, Rs , {shft #}

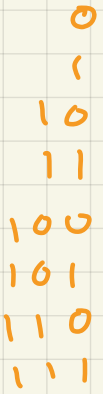


XXXCCS Rd, Rs , {shft #}

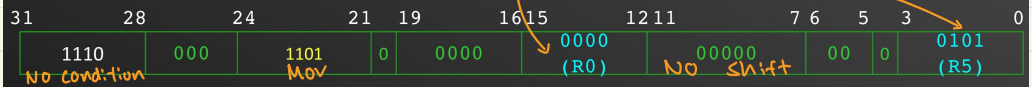


XXXCC Rd, #immediate (rotated)

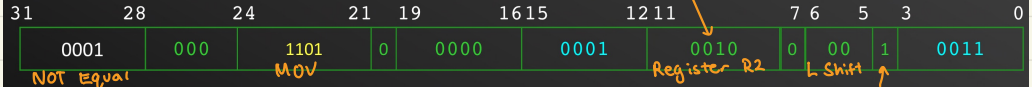




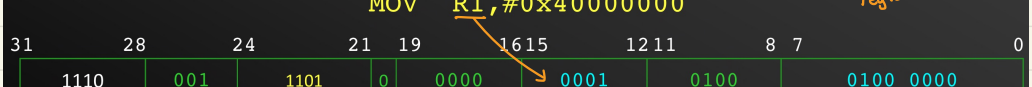
MOV R0,R5



MOVNE R1, R3, LSL R2



MOV R1,#0x40000000



Code	Condition	Flags	Meaning
0000	EQ	Z = 1	Equal
0001	NE	Z = 0	Not equal
0010	CS or HS	C = 1	Higher or same, unsigned
0011	CC or LO	C = 0	Lower, unsigned
0100	MI	N = 1	Negative
0101	PL	N = 0	Positive or zero
0110	VS	V = 1	Overflow
0111	VC	V = 0	No overflow
1000	HI	C = 1 and Z = 0	Higher, unsigned
1001	LS	C = 0 or Z = 1	Lower or same, unsigned
1010	GE	N = V	Greater than or equal, signed
1011	LT	N != V	Less than, signed
1100	GT	Z = 0 and N = V	Greater than, signed
1101	LE	Z = 1 and N != V	Less than or equal, signed
1110	AL		Always.
1111	NV		Reserved (unused)

Inst. Type field Field

- 4 bits for instructions ➔ up to 16 combinations
- 4 bits for source and destination registers ➔ 16 registers

Code	Instruction
0000	AND
0001	EOR
0010	SUB
0011	RSB
0100	ADD
0101	ADC
0110	SBC
0111	RSC

Code	Instruction
1000	TST
1001	TEQ
1010	CMP
1011	CMN
1100	ORR
1101	MOV
1110	BIC
1111	MVN

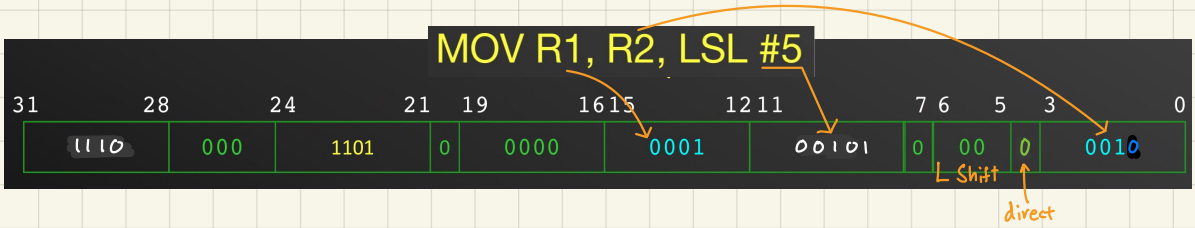
LSL, LSR, ASR, ROR, and RRX has no dedicated instructions

Instructions are encoded as MOV instructions

E.g. LSL R1, R2, #5 ≡ MOV R1, R2, LSL #5

Specify the shift type in bits 5 and 6

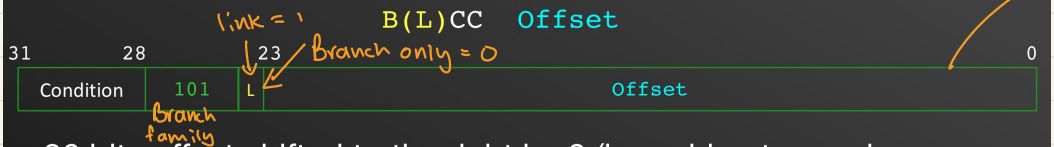
Code	Shift type
00	LSL
01	LSR
10	ASR
11	ROR/RRX



Branch & Link

Branch Instructions

- For conditional branch and branch with link



- 26 bits offset shifted to the right by 2 (branching to word addresses) ➔ calculated by the assembler

- Branch range: ±32 MBytes

Basically,

0x100
0x104
0x108
0x10C
0x110

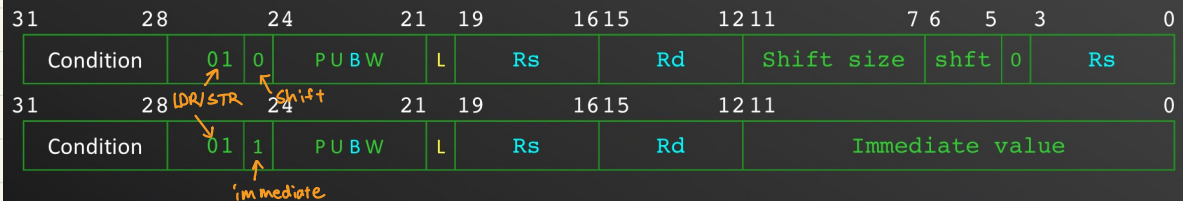
all have zero as the two LSB. ∴ This "ignores" the spare zeros

LDR STR

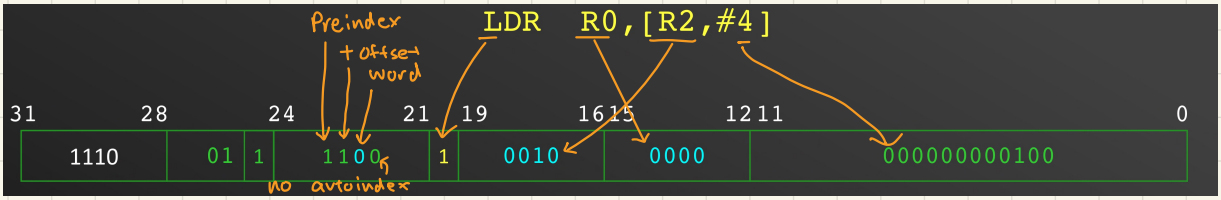
Load/Store Instructions

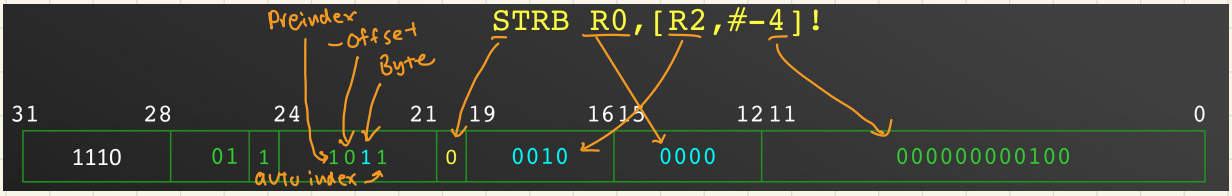
- Load/Store word bye

LDR/STR{B}CC Rs,[RD,offset pre or post]W



- L bit determines load (1) or store (0)
- P bit determines indexing pre (1) or post (0)
- U determines offset direction add (1) or subtract (0)offset
- B determines byte (1) or word (0) access
- W is for write back (!) of the offset → Auto indexing



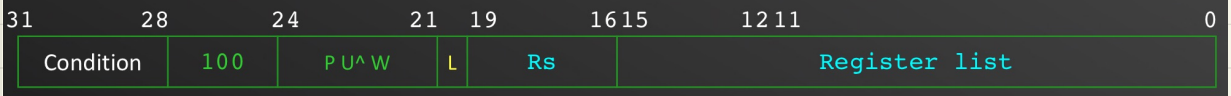


LDM FD STM FD

Load/Store Instructions

- Load/Store Multiple instructions

LDM/STMFXCC RsW,{register list}



- L bit determines load (1) or store (0)
- P bit determines indexing before (1) or after (0)
- U determines offset direction add (1) or subtract (0)
- ^ is “don’t care”
- W is for write back (!) after operation

PU	Instruction
10	STMFD
01	LDMFD
00	STMFA
11	LDMFA

STMFD SP!,{R0-R6}