CZ1106
CE1106

Chapter 5

Instruction Set

1

CZ1106
CE1106

Chapter 5

Instruction Set

Data Transfer Instructions

Learning Objectives (5.1)
1. Describe how data in register and memory can be efficiently transferred. 2. Describe how byte-sized data can be access in memory.

2

CZ1106 CE1106

Instruction Set – Basic Categories

- Non system-level instructions in a processor can be typically classified into three basic groups:

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.

3

CZ1106 CE1106

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

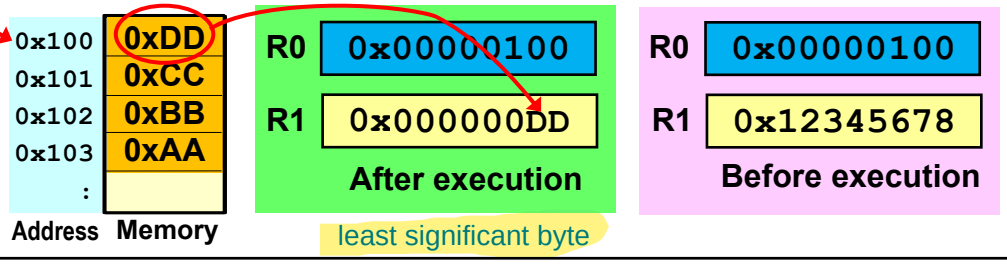
4

CZ1106 CE1106

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)
```



5

CZ1106 CE1106

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
```


6

CZ1106 CE1106

Program Example

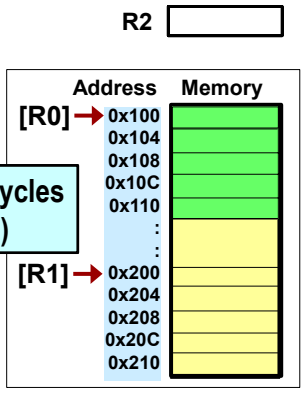
Copying a Block of Memory

- 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

7

CZ1106 CE1106

Program Example (optimised version)

Copying a Block of Memory

- The code can be further optimised for speed and size by using the autoindexing feature.
- The register indirect with **post-index** autoindexing will **automatically** add the 4 offset to the array pointers **after** memory access.

```
        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

8

CZ1106 CE1106

Program Example (further optimisation version)

Copying a Block of Memory

- 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**.

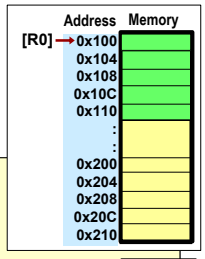
```
        MOV  R0,#0x100        ; setup source and destination pointer
loop    LDR  R2,[R0],#4       ; memory to register transfer
                              ; with post index autoindexing
        STR  R2,[R0,#0xFC]    ; register to memory transfer
                              ; with base plus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

Use one register less, save 1 cycle

20 cycles (4x5)

loop back 5 times

Offset range = ±4096



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

9

CZ1106 CE1106

Summary

- The **efficient** data transfer instruction `MOV` and `MVN` are probably the most commonly used instructions.
  - Can be used with the **register direct** and **immediate** addressing modes.
- Memory data transfer requires the use of `LDR` and `STR` instructions.
  - Numerous variants of **register indirect** addressing modes can be used.
  - Memory data transfer instructions require **two** clock cycles to execute.
- Byte-sized memory access can be done using `LDRB` and `STRB`.
  - Byte moved into register is zero-extended.
  - Byte access of memory does not have data alignment restrictions.

10

CZ1106
CE1106

Chapter 5

Instruction Set

Arithmetic Instructions

Learning Objectives (5.2)

1. Describe the operation and uses of the basic arithmetic instructions in the ARM instruction set.
2. Describe how arithmetic operations influence the status of Condition Code flags.

11

CZ1106
CE1106

Instruction Set – Basic Categories

- Non system-level instructions in a processor can be typically classified into three basic groups:

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**, logical or shift operations.
- Program control – instructions that alter the normal sequential execution flow of a program.

12

CZ1106 CE1106

Arithmetic Instructions

- The basic arithmetic operations are **Add** and **Subtract**.
  - These arithmetic instructions involve **three** operands.

```
ADD R2,R0,R1
```

  - Add and subtract can only involve **registers** or **immediate values** (as source operand).

```
ADD R2,R0,#4
```

  - The ARM instruction set provides some **variants** of the basic add and subtract operations to provide more flexibility during programming.

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

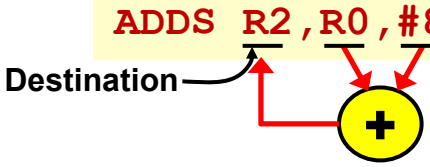
13

CZ1106 CE1106

ADD Instruction

- Addition is a **commutative** operation that adds two source operands (order is immaterial).
  - But only **rightmost** operand can take on an **immediate** value. The other operands must be registers

```
ADDS R2,R0,#8
```



R0	0x00000003
R1	0x00000006
R2	0x12345678

Before execution

  - In order to influence the condition code flags (N,Z,V,C), the "S" suffix must be added in the mnemonic.

R0	0x00000003
R1	0x00000006
R2	0x0000000B

After execution

14

CZ1106 CE1106

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.

15

Processor does not know whether it is signed or unsigned, up to user to decide and just gives all possible flags

CZ1106 CE1106

SUB Instruction

- Subtraction is a **non-commutative** operation.
- All operands are register but the **rightmost** operand (subtrahend) can take on an **immediate** value.

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

Destination

R0	0x00000009
R1	0x00000006
R2	0x12345678

Before execution

- 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
```

R0	0x00000009
R1	0x00000006
R2	0x00000005

After executing SUB

16

useful as only the rightmost operand can be an immediate value

CZ1106 CE1106

SUB Instruction (cont)

- 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

17

CZ1106 CE1106

Condition Code Flags and SUB

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

Minuend Subtrahend
unsigned (A) < unsigned (B) — **C=0**

Minuend Subtrahend
unsigned (A) ≥ unsigned (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).

Learn More: Google "ARM subtraction carry flag"

18

CZ1106 CE1106

Program Example

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.

1st –ve integer converted

R1 0x00000001

```
        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
```

60 cycles (6x10)

loop back 9 times

[R0] →

Address	Memory	
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)
:		

Convert 10 words starting at address 0x100 to its +ve values

19

CZ1106 CE1106

Program Example (optimized version)

Convert Negative to Positive

- 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

R1 0x00000001

```
        MOV  R0,#0x100   ; setup source pointer
        MOV  R2,#0       ; load zero into R2
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
```

R2 = 0

R1 = R2 – R1
R1 = 0 – R1

50 cycles (5x10)

loop back 9 times

[R0] →

Address	Memory	
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)
:		

Convert 10 words starting at address 0x100 to its +ve values

20

CZ1106 CE1106

Program Example (further optimisation version)

Convert Negative to Positive

- The **RSB** instruction can be used to do the negation with an additional register.
- 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
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
```

Use one register less, save 1 cycle

R1 = 0 – R1

loop back 9 times

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)
	:		
	:		

Convert 10 words starting at address 0x100 to its +ve values

21

CZ1106 CE1106

Carry-based Arithmetic Instructions

- 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

- Like the other arithmetic instructions, the "**S**" suffix can be added to the mnemonic to influence the condition code flags (**N**,**Z**,**V**,**C**),.

22

CZ1106
CE1106

Summary

- The **ADD** and **SUB** instructions are 3-operand instructions.
  - Supports **register direct** and **immediate** addressing (rightmost operand only).
  - Influence all condition code flags (**N**,**Z**,**V**,**C**) when "**S**" suffix is used.
  - **SUB** is non-commutative. **RSB** allows the minuend to be an **immediate value**.
- Arithmetic instructions that incorporate the carry flag (**C**) can be employed for multi-precision arithmetic.

23

CZ1106
CE1106

24

24

CZ1106
CE1106

Chapter 5

Instruction Set

Logical, Shift and Rotate Instructions

Learning Objectives (5.3)

1. Describe the operation and uses of the various logical instructions.
2. Describe the operation and uses of the various shift and rotate instructions.
3. Describe how multiplication and division can be done using bit shift.

25

CZ1106
CE1106

Instruction Set – Basic Categories

- Non system-level instructions in a processor can be typically classified into three basic groups:

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, **logical or shift** operations.
- Program control – instructions that alter the normal sequential execution flow of a program.

26

CZ1106 CE1106

Logical Instructions

- Logical instructions provide various **Boolean** operators.
  - `MVN` is a **two-operand** instruction that does the **NOT** operation.

    Example: `MVNS R2,R2`

    R2 `0x00000000` (Before execution) → R2 `0xFFFFFFFF` (After execution) **N=1** and **Z=0**

  - 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` (Before execution) → R1 `0x1234FFFF` (After execution) **N=0** and **Z=0**

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

27

CZ1106 CE1106

Logical Instructions
AND, OR and EOR Applications

- The basic logical instructions can be used to:
  - `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

28

CZ1106 CE1106

Instruction examples

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

29

CZ1106 CE1106

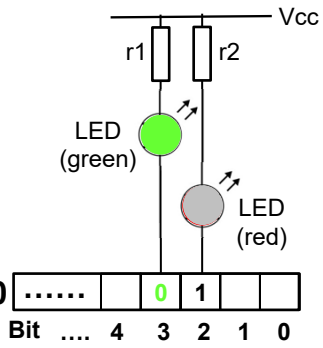
Program Example

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

30

CZ1106 CE1106

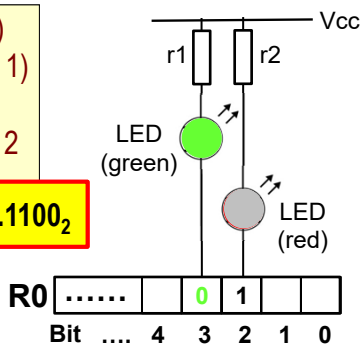
Program Example (Improved version)

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.
- Once the alternate pattern for the two LEDs are in place, `EOR` can be used to **flip** or **invert** their state after each cycle.

```
        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



Alternating patterns for bits 3 and 2 in R0

31

CZ1106 CE1106

Shift and Rotate Instructions

- ARM has several shift and rotate operations:
- Logical Shift Left (`LSL`) and Logical Shift Right (`LSR`).

- Arithmetic Shift Right (`ASR`)

retains the sign bit

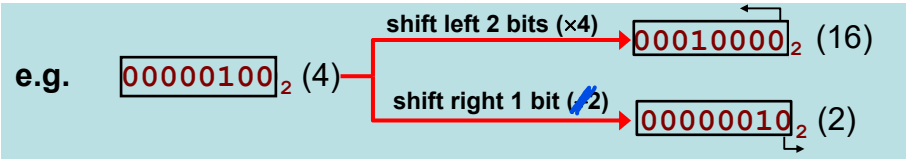
- Rotate Right (`ROR`) and Rotate Right Extended (`RRX`).

32

CZ1106 CE1106

Doing Arithmetic with Shift

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



- 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`).

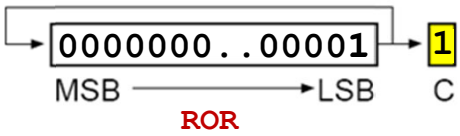
- The "`S`" suffix is used on the data processing operator to influence the `C` flag.

33

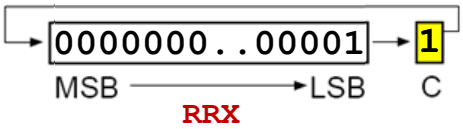
CZ1106 CE1106

Rotate Operations

- 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 the leftmost end, while the bit shifted out replaces the current **C**-flag.



34

CZ1106
CE1106

Shift and Rotate Mnemonics

- 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

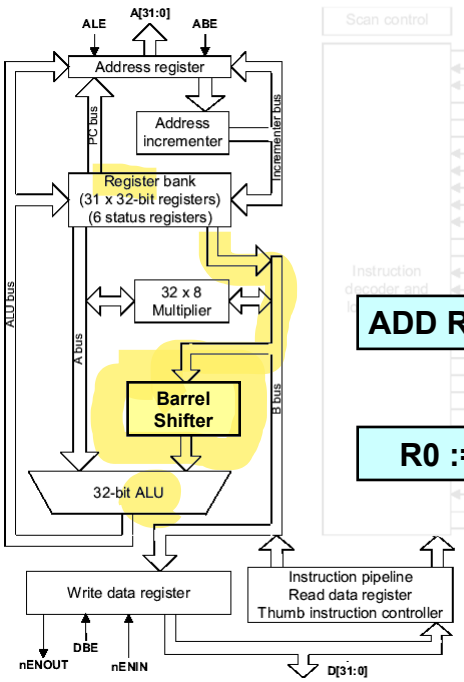
```
ADD  R2,R1,R0,LSR #2  ; R2=R1+R0>>1
```
ADD R1 with 2-bit right shifted R0. Put result in R2.

```
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.

35

CZ1106
CE1106



ADD R0, R1, R2, LSR #2

R0 := R1 + (R2 >> 2)

AMR7TDMI Processor Logic

36

CZ1106 CE1106

Program Example

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

AND mask of 0x0000001

Initial R0 = 0x22222222

R0	0x11111111
R1	0x00000001
R2	0x00000001
R3	0x0000001F

Changes after one loop

Count the number of binary 1s in R0

37

CZ1106 CE1106

Program Example (optimized version)

Count Number of 1's

- Count the number of binary 1s in register **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

Count the number of binary 1s in R0

38

CZ1106
CE1106

Summary

- Logical instructions such as **AND**, **ORR**, **EOR** can be used to **clear**, **set** and **complement** specific bits in a register, respectively.
- Arithmetic shift instruction can be used as a fast way of implementing **multiplication** and **division** by values of 2^N.
- In the ARM, shift and rotate operations are used in **conjunction** with data transfer and data processing operations.

39

CZ1106
CE1106

40

40

CZ1106
CE1106

Chapter 5

Instruction Set

Program Control Instructions

Learning Objectives (5.4)

1. Describe the various conditional branch instructions and its uses.
2. Describe how conditional test can be implemented.

41

CZ1106
CE1106

Instruction Set – Basic Categories

- Non system-level instructions in a processor can be typically classified into three basic groups:

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`

Covered in Modular Programming

- Data transfer – instructions that move data between regi ory.
- Data processing – instructions that modify the data ugh arithmetic, logical or shift operations.
- **Program control** – instructions that alter the normal sequential execution flow of a program.

42

CZ1106 CE1106

Program Control Instructions

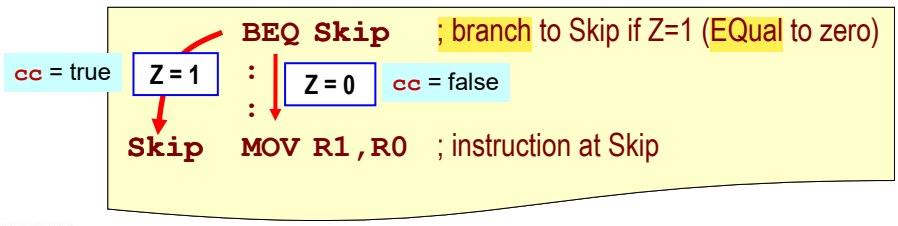
- These instructions facilitate the **disruption** of a program's normal **sequential** flow.
  - The disruption of sequential flow is implemented by modifying the contents of the Program Counter (**PC**).
  - The content of the **PC** can be modified **directly** or by using a **Branch** instruction.
  - A jump can be executed based on a given condition (e.g. if result of previous execution is negative) and this is called a **conditional branch**.
  - Conditional branch is useful for implementing:
    - conditional constructs (e.g. **if** or **if-else**)
    - loop constructs (e.g. **for** or **while** loops)

43

CZ1106 CE1106

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.



44

CZ1106 CE1106

Test Conditions for Bcc

- 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   :
Skip    MOV  R1,R0
```

`cc` = EQ (Equal to zero)

45

CZ1106 CE1106

Different Bcc Conditions

- There are 15 possible conditional tests for `Bcc`.

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
R0 > R1 :
        :
        :
Else    MOV  R1,R0
```

R0 > R1 (signed compare)

46

CZ1106 CE1106

Program Example

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

Count the number of binary 1s in R0

47

CZ1106 CE1106

Program Example (Alternative Count Loop)

Count Number of 1's

- Count the number of binary 1s in register **R0**.
- 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

Count the number of binary 1s in R0

48

CZ1106 CE1106

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     :
```

- 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     :
```

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

Note: R1≥R2 label is only for illustration. The "≥" is not a valid label symbol in the VisUAL ARM simulator

49

CZ1106 CE1106

Conditional Test using CMP

- 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     :
```


50

CZ1106 CE1106

CMP

- `CMP` **subtracts** the source operand from the destination register and sets the **CC** flags according to the results.
  - Destination register remain **unmodified** after `CMP`.
  - CC flags affected in the same manner as the **subtract** instruction (`SUBS`).

```
        SUBS   R1,R1,R2

        BGE    R1≥R2
          :
R1≥R2     :
```

R1 modified to achieve desired flow control

```
        CMP    R1,R2

        BGE    R1≥R2
          :
R1≥R2     :
```

Same flow control but R1 unchanged

51

CZ1106 CE1106

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).

52

CZ1106
CE1106

Summary

- Conditional branch (**Bcc**) allows us to implement conditional and loop constructs.
- Appropriate (**Bcc**) conditions must be selected for the conditional test used.
  - The (**cc**) choice needs to take into account of data type being used (i.e. signed or unsigned numbers).
- Appropriate operations (e.g. **CMP** or **SUBS**) are used to set the **N**, **Z**, **V**, **C** flags before conditional test can be done.

53

CZ1106
CE1106

54

54

CZ1106
CE1106

Chapter 5

Program Example

Finding the Largest Number

Learning Objectives (5.5)

1. Use appropriate data transfer instructions to retrieve memory arrays efficiently.
2. Use appropriate program control instructions to determine flow of program based on desired outcomes.
3. Implement a simple find max algorithm in ARM assembly.

55

CZ1106
CE1106

Program 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`.
  - **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

Largest Value

R3	0x00000007

Number Array

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

56

CZ1106 CE1106

Possible Solution

Find Largest Number (FindMax)

```
        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 Memory

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).

57

CZ1106 CE1106

Possible Solution

Find Largest Number (FindMax)

```
        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).

58

CZ1106 CE1106

Conditional Execution

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

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

ARM code example

```
        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)
MOVEQ   R1,#3              ; then { r1 := 3}
MOVNE   R1,#5              ; else  { r1 := 5}
SKIP    ......             ;
```

Executes if **Z** = 1

Executes if **Z** = 0

59

CZ1106 CE1106

Summary

- **Register indirect** addressing modes (with and without autoindexing) can be used to access array elements in memory.
- Conditional branch (**Bcc**) allows us to implement conditional and loop constructs.
- Appropriate conditions (e.g. **LS** and **NE**) must be selected to implement the required test.
- Appropriate operations (e.g. **CMP** or **SUBS**) are used to set the (**N**, **Z**, **V**, **C**) flags before conditional test can be done.
- Conditional execution (e.g. **MOVHI** or **ADDEQ**) can be used to avoid doing conditional branching.

60