

# VIP Instruction Set



**A/P Goh Wooi Boon**

# VIP Instruction Set

## Data Transfer and Arithmetic Instructions

### Learning Objectives (4a)

1. Describe how data in register and memory can be efficiently transferred.
2. Describe the operation and uses of the various arithmetic instructions.
3. Describe how arithmetic operations influence the status of CCR flags.

# Instruction Set – Basic Groups

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

## Data Transfer

### VIP examples:

```
MOV R0,#3  
MOV R0,[0x100]  
POP R1
```

## Data Processing

### VIP examples:

```
ADD R0,#3  
OR R1,R2  
ROR R3
```

## Program Control

### VIP examples:

```
JMP 3  
JGE 0xFFC  
CALL Routine
```

- **Data transfer** – instructions that move data between registers and/or memory.
- **Data processing** – instructions that modify the data in register/memory through arithmetic or logical operations.
- **Program control** – instructions that alter the normal sequential execution flow of a program.

# Data Transfer Instructions

- Different addressing modes provide many ways to move data between register, memory & stack.
- Data transfer instructions involve two operands, a **source** and **destination** operand (some are implied, e.g. push to stack).
- Examples of various data transfer instructions in VIP:

MOV R0,R1	; register to register transfer
MOV R0,[R1]	; memory to register transfer
MOV [0x130],[0x100]	; memory to memory transfer
PSH R0	; register to stack transfer
POP [0x110]	; stack to memory transfer
MOV R1,#1	; constant to register transfer
MOV [0x110],#0	; constant to memory transfer

# Copying a Block of Memory

- Block copy is used to replicate a contiguous segment of memory from one location to another.
- VIP supports memory to memory transfer and this allow the data transfer to be executed in a single instruction.

```
MOV  R0 ,#0x100 ;setup source pointer
MOV  R1 ,#0x200 ;setup destination pointer
loop MOV  [R1],[R0] ;memory to memory transfer
      ADD  R0 ,#1 ;increment to next address
      ADD  R1 ,#1
```

loop back n times

```
MOV  R2,[R0]
MOV  [R1],R2
```

**Block copy code (partial)  
in VIP mnemonics**

Two instructions needed if  
memory to memory  
transfer is not available

# Data Transfer Instructions (cont)

- Many processors have optimized instruction to load small constant values.
- The **MOVS** instruction in VIP loads **small values** (0-15) into an operand (e.g. setup counter or clear operand).

**MOV R0, #0** ; standard MOV (2 word, 2 cycles)

**MOVS R0, #0** ; optimized MOVS (1 word, 1 cycle)

- In the VIP ISA, not all data transfer instructions influences the CCR flags .
- The **MOV** instruction influences the **N** and **Z** flags.
- The stack transfer instructions (**PSH** and **POP**) and **MOVS** do not affect the CCR flags.

# Data Processing Instructions

- The data processing category tends to have the most variety of instructions.
- Especially in processor designed to support fast numeric computations and digital signal processing.
- Examples of data processing instructions in VIP:

**NEG** (negate)

**ADD**, **ADDC**, **INC** (addition & variants)

**SUB**, **DEC** (subtraction & variants)

**UMUL** (multiplication) - to be covered later (Computer Arithmetic)

**AND**, **OR**, **EOR**, **NOT** (logical)

**ROR**, **ROL**, **RRC**, **RLC**, **RAR** (shift and rotate)

# Addition

- Add source operand to the destination operand.
- Destination operand is overwritten with result of addition.

● e.g.

```

ADD R0, #1           ; R0 = R0 + 1
ADD R0, R1           ; R0 = R0 + R1
ADD R0, [0x100]
ADD [0x102], [0x103]
    
```

- The **ADDC** instruction is used for **multi-precision** addition. Numbers larger than 12-bits can be added using **ADDC**.

```

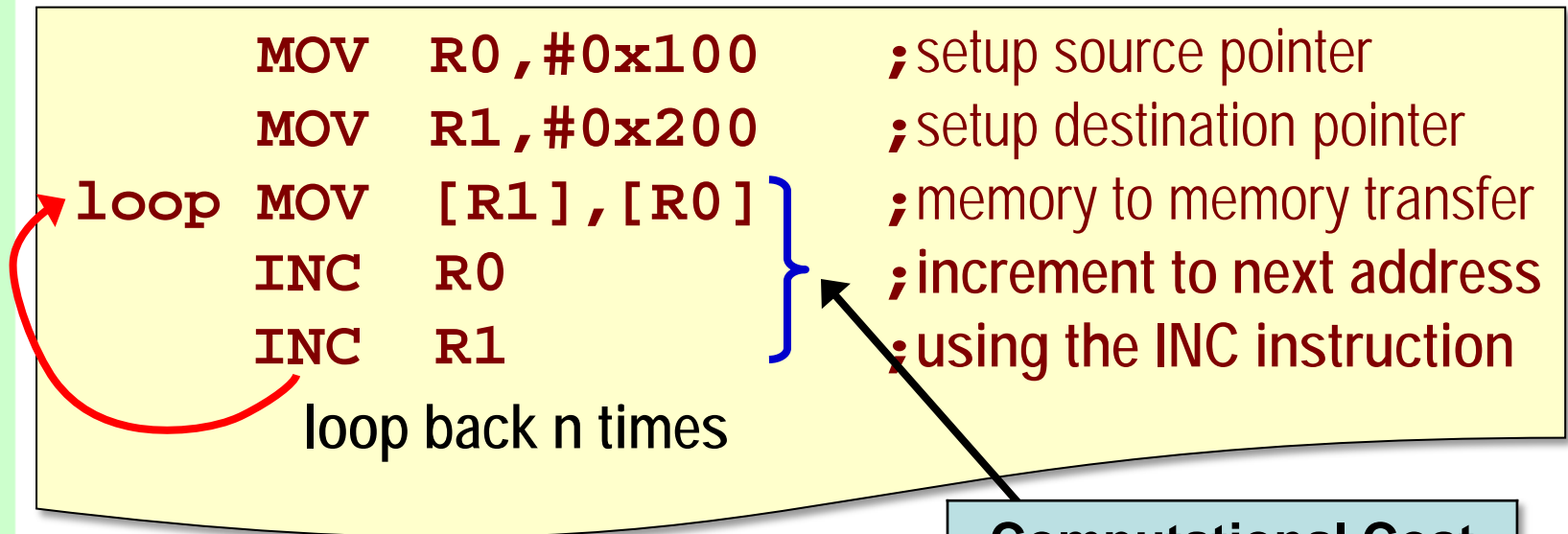
ADDC R0, R1           ; R0 = R0 + R1 + C flag bit
    
```

- The **INC** instruction is optimized to **add one** to operand (**1 execution cycle only**). Can be used to access **consecutive array efficiently**.



## Program Example (Optimizing) Efficient Block Copy

- Execution speed of the Block copy code is faster if the **INC** instruction is used to increment the two address pointers within the loop.



**Optimized version of the  
Block copy code segment**

Computational Cost (in Cycles)	
With <b>ADD</b>	With <b>INC</b>
7	5

# CCR Flags and ADD

● **ADD** can affect all the V, N, Z, C flags.

(+ve)	0000	0001	(1)
(+ve)	+0111	1111	(127)
<hr/>			
(-ve)	1000	0000	(-128)

**N=1, V=1**

2's complement  
oVerflow

(+ve)	0000	0001	(1)
(-ve)	+1111	1111	(-1)
<hr/>			
(+ve)	0000	0000	(0)

**Z=1, C=1**

unsigned  
overflow

## 4.5 4xx – ADD d,s

(taken from VIP Technical Reference)

The destination as indicated by d is made equal to the destination arithmetically added to the source as indicated by s. That is to say,  $d \leftarrow d + s$ .

### Effect on flags in the Status Register:

Using 2's complement arithmetic, if the sign of the result is incorrect the V bit is set otherwise it is cleared. Specifically, the V bit is set if  $\text{msb}(s) = \text{msb}(d)$  and  $\text{msb}(\text{result}) \neq \text{msb}(d)$ .

The most significant bit of the result is copied to the N bit.

If all the bits in the result are zero, the Z bit is set otherwise it is cleared.

If the addition caused a carry out of the most significant bit, C bit is set otherwise it is cleared.

# Subtraction

- Subtract source from the destination operand.
- Subtraction is not commutative (so note which is **d** and **s**).
- e.g.
  - `SUB R0, #1` ;  $R0 = R0 - 1$
  - `SUB R0, R1` ;  $R0 = R0 + \text{NOT}(R1) + 1$
  - `SUB R0, [0x100]`
  - `SUB [0x102], [0x103]`
- CCR flags that may be affected are **N**, **Z**, **V** and **C**.
- The **DEC** instruction is optimized to **subtract one** from operand (**1 execution cycle only**).  
Note: Useful for efficient **counting loop constructs**.
- The **NEG** instruction can be used to negate a value.

# Summary

- The data transfer instruction **MOV** is probably the most commonly used instruction.
  - Using the right **addressing mode** can improve data transfer efficiency.
- Arithmetic instructions influence the state of all the CCR flags.
- The VIP ISA provide some limited capability but efficient version of commonly used operations (e.g. **MOVS**, **INC**, **DEC**).

# VIP Instruction Set

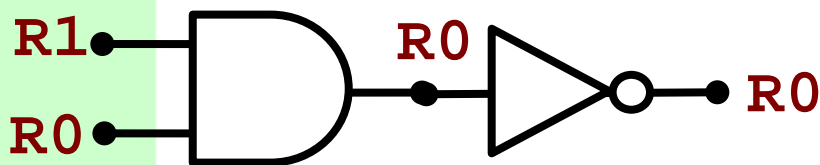
## Logical and Rotate Instructions

### Learning Objectives (4b)

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

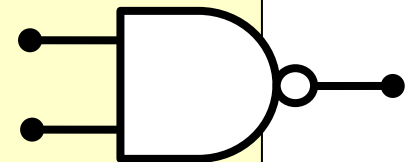
# Logical Instructions

- Logical instructions provide the various **Boolean** operators.
- The **INV** instruction is single-operand and implements the **NOT** operation.
- The **AND**, **OR** and **EOR** instructions are **dual-operand** instructions for the AND, OR and EX-OR operations
- CCR flags that may be affected are **N** and **Z**.
- Other Boolean operations can be derived from these basic logical instructions.



**;NAND operation**

```
AND R0,R1  
INV R0
```



# Logical Instructions

## AND, OR and EOR

- Dual operand logical instructions can be used to:
  - **AND** – **clear** specific bits in destination operand.
  - **OR** – **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

# Instruction examples

## AND, OR and EOR

Bits 7 6 5 4 3 2 1 0  
R0 **..01010101**

Initial condition of least significant 8 bits register R0

• e.g. **AND R0, #..11110000** (e.g. **AND R0, #0x**F**F0**)

R0 **..01010000** Bits 0 to 3 cleared after execution

• e.g. **OR R0, #..11110000** (i.e.. **OR R0, #0x0**F**0**)

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

• e.g. **EOR R0, #..11110000** (e.g. **EOR R0, #0x0**F**0**)

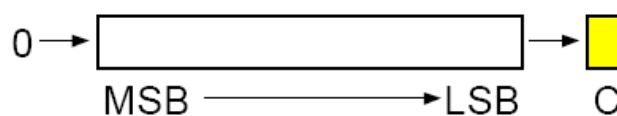
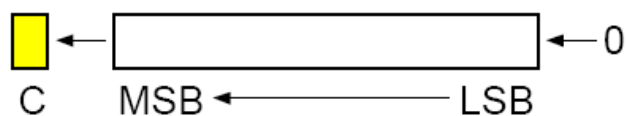
R0 **..10100101** Bits 4 to 7 inverted after execution



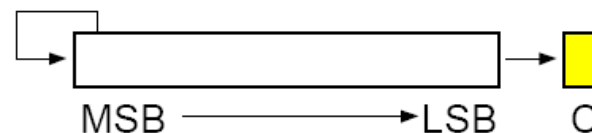
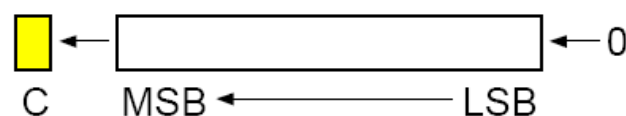
# Shift and Rotate Operations

- A visual summary of operations for the shift and rotate instructions:

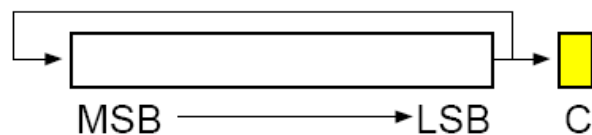
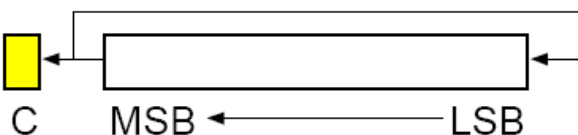
## Logical Shift



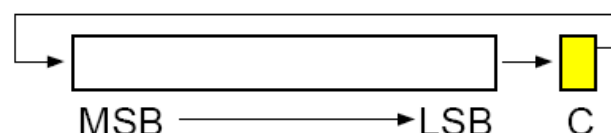
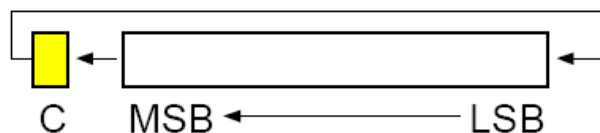
## Arithmetic Shift



## Rotate



## Rotate with Carry



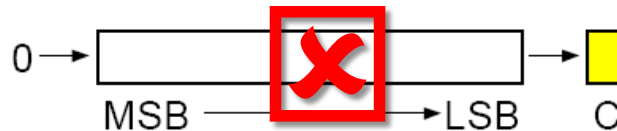
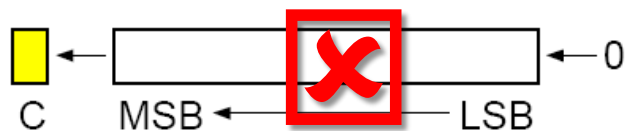
Left

Right

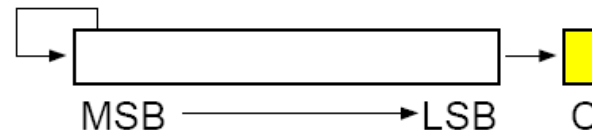
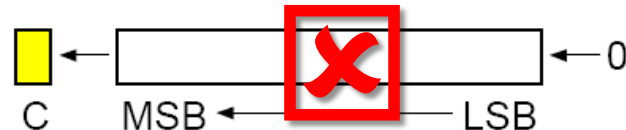
# Shift and Rotate in VIP

- VIP ISA implements a subset of the available combination of shift and rotate operations.

**Logical Shift**

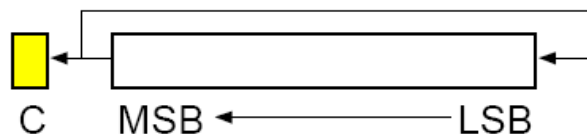


**Arithmetic Shift**

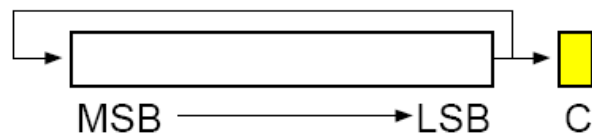


**RAR**

**Rotate**

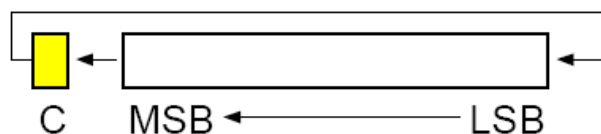


**ROL**

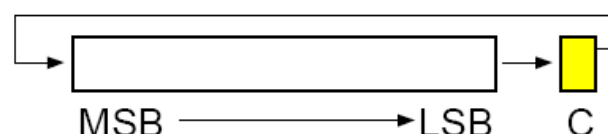


**ROR**

**Rotate with Carry**



**RLC**



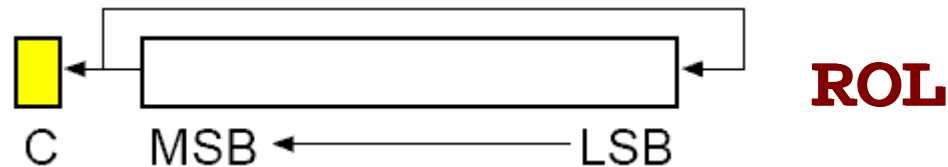
**RRC**

**Left**

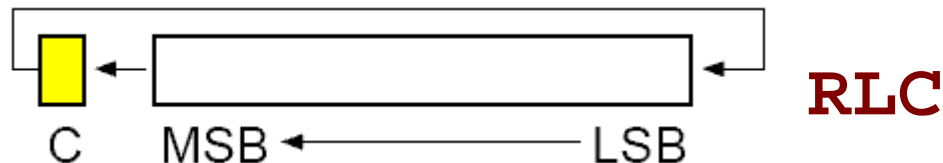
**Right**

# Rotate

- Rotate is also called **cyclical shift**, as no bit in register is lost during the shift operation.
- In basic rotate (**ROL**, **ROR**), bit shifted out of register is returned in at the other end and is placed into the **C-flag**



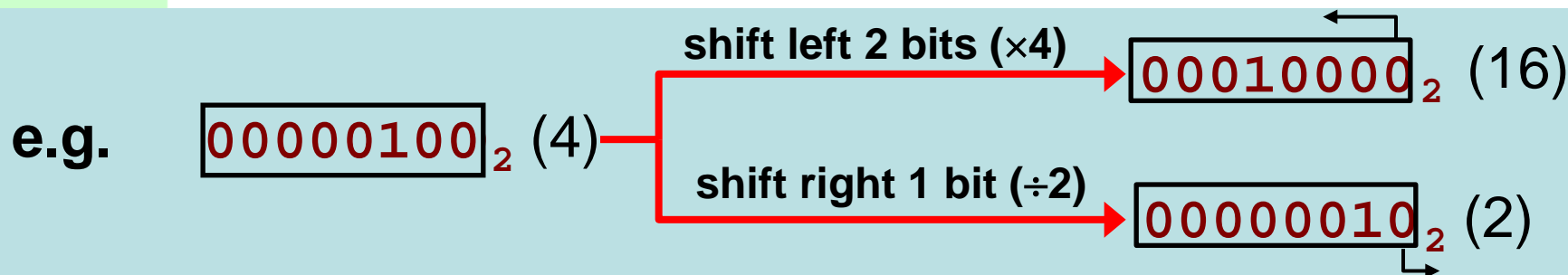
- In rotate with carry (**RLC**, **RRC**), the **C-flag** is shifted into the register at the other end, while the bit shifted out replaces the current **C-flag**.



- Rotate with carry can be used to implement shift operations.

# 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 (use Shift Left).
- In **unsigned divide**, binary “0” is shifted into the MSB of the register from the left (use Shift Right).
- In **signed divide**, the sign bit is shifted into the MSB of the register from the left (use Arithmetic Shift Right – **RAR**).

# Implementing Logical Shift Left

- There is no shift left instruction in the VIP ISA.
- Shift left can be implemented using the Rotate with Carry left (**RLC**) instruction.
- The Carry (**C**) bit must be cleared to ensure only binary “0”s shifts into the LSB.

**;Shift left by 1 bit**

**BCSR 1**

**RLC R0**

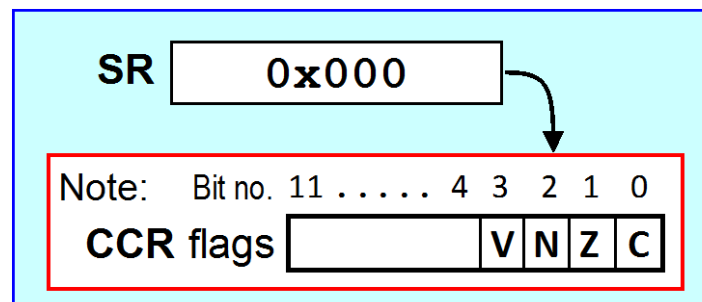
**Clearing C bit with **BCSR****

**;Shift left by 1 bit**

**AND SR,#0xFFE**

**RLC R0**

**Clearing C bit with **AND****



# Instruction examples

## Shift and Rotate

R0 **1000 0100 0010** R1 **0111 1111 1111** **N=0, Z=0, V=0, C=0**

Initial conditions of register R0 , R1 and CCR flags

• e.g. **ROR R0** R0 **0100 0010 0001** **N=0, Z=0, V=0, C=0**

After execution

• e.g. **RAR R0** R0 **1100 0010 0001** **N=1, Z=0, V=0, C=0**

After execution

• e.g. **RAR R1** R1 **0011 1111 1111** **N=0, Z=0, V=0, C=1**

After execution

• e.g. **RCN 2**  
**ROR R0** R0 **1010 0001 0000** **N=1, Z=0, V=0, C=1**

After execution

• e.g. **RLC R1** R1 **1111 1111 1110** **N=1, Z=0, V=0, C=0**

After execution

# Rotating Multiple Bits

- VIP instruction set provides the **RCN n** mnemonic to optimize the implementation of multiple bit shift.
- Multiple shift from 1 to 15 can be specified (4 bits).
- If **n** is 0, the number of bits to shift is obtained from the least significant 4 bits of the **AR** register.

**;Rotate right by 4 bits**

**ROR R0**

**ROR R0**

**ROR R0**

**ROR R0**

**Using only ROR**

Instruction Length (word)	Execution Time (cycles)
4	4

**;Rotate right by 4 bits**

**RCN 4**

**ROR R0**

**Using RCN and ROR**

Instruction Length (word)	Execution Time (cycles)
2	2

# Summary

- Logical instructions such as **AND**, **OR**, **XOR** can be used to **clear**, **set** and **complement** specific bits in a register, respectively.
- Logical **shifts** can be implemented using the rotate instructions provided the **C-flag** is **cleared** first.
- Arithmetic shift instruction can be used as a fast way of implementing **multiplication** and **division** by values of  $2^N$ .



# VIP Instruction Set

## Program Control Instructions

### Learning Objectives (4c)

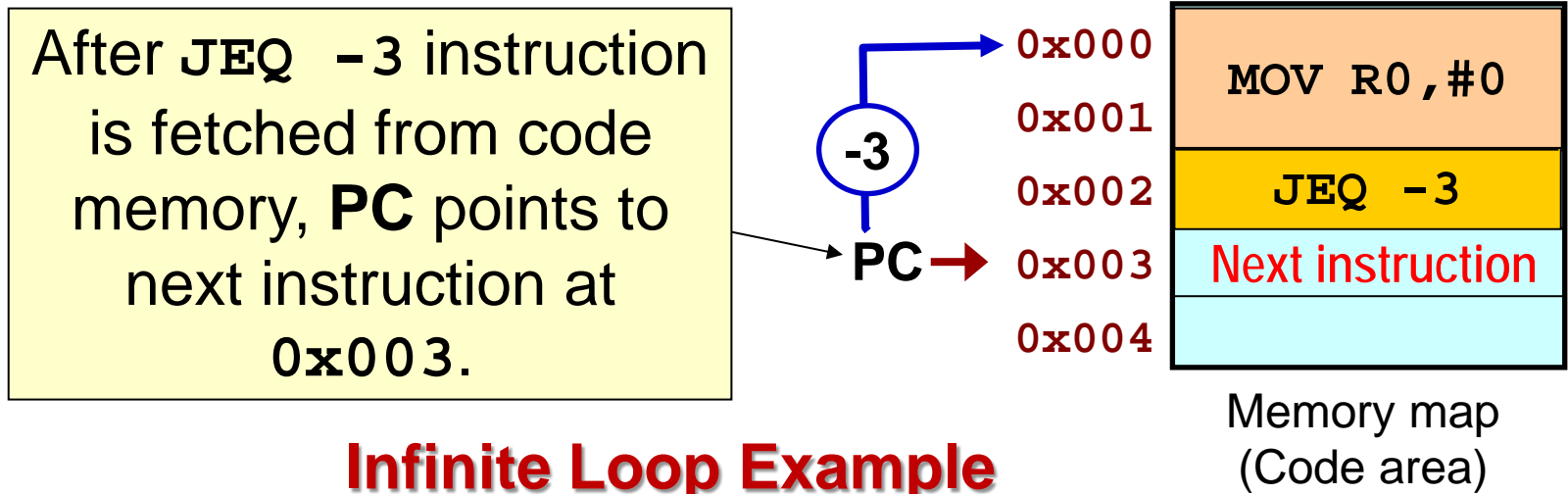
1. Describe the various conditional jump instructions and its uses.
2. Describe how conditional test can be implemented.

# 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 **jump** 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 jump**.
- Conditional jump is useful for implementing:
  - conditional constructs (e.g. **if** or **if-else**)
  - loop constructs (e.g. **for** or **while** loops)

# Conditional Jump (Jcc)

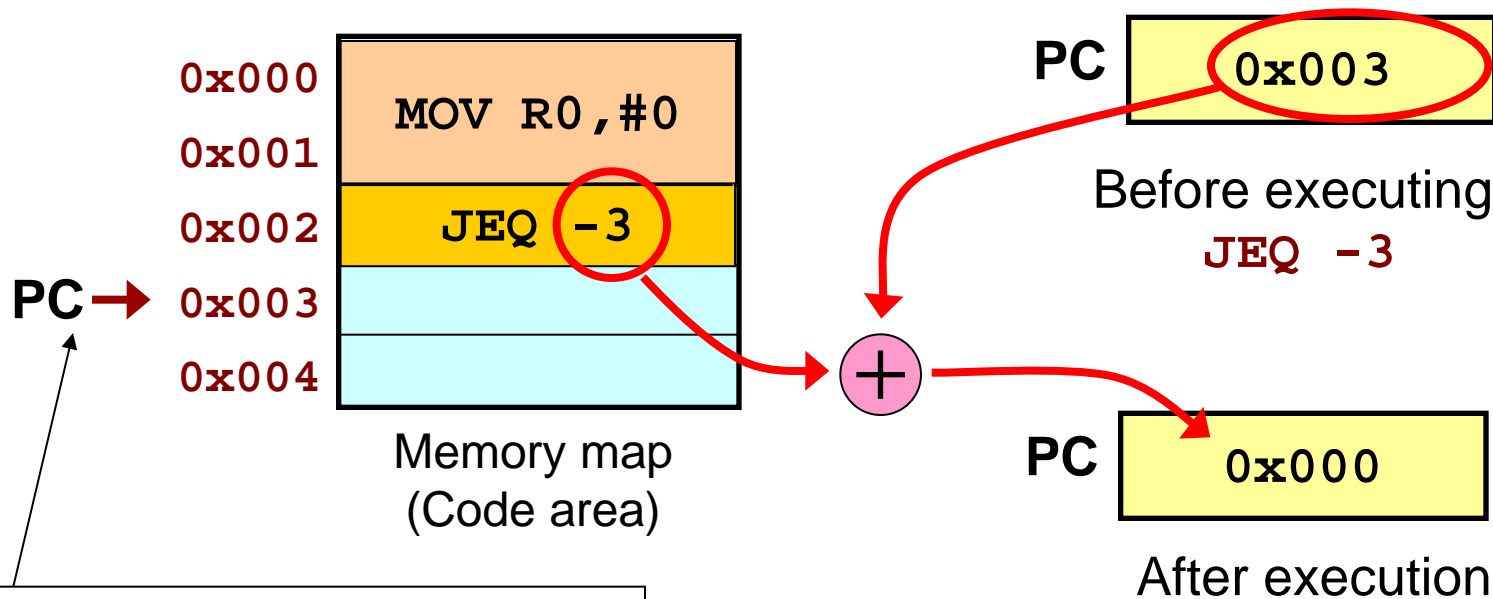
- VIP provides conditional jump using **Jcc**.
- If the condition specified in the condition field (**cc**) is **true**, a displacement is added to the **PC**, otherwise next instruction is executed.
- **Jcc** uses the **PC-relative** addressing mode.
- PC value used to compute required displacement is the start address of the instruction **immediately after Jcc**.



# (Instruction Example)

## Conditional Jump (JEQ)

• E.g. `loop    MOV    R0 , #0`  
          `JEQ    -3` } This is an example of an infinite loop



After JEQ -3 instruction is fetched from code memory, PC points to next instruction at 0x003.

# Test Conditions for Jcc

- VIP provides **different** conditional jump options.
- 12 possible conditions is permitted in the condition field (**cc**) using combinations of the **N**, **Z**, **V**, **C** flags.

e.g. Jcc	Operation and CCR flag conditions
JMP	$PC \leftarrow PC \pm n$
JEQ	If $Z=1$ , $PC \leftarrow PC \pm n$
JGE	If $N = V$ , $PC \leftarrow PC \pm n$

- This allows **flexible** conditional jump to be programmed based on the result of instructions **prior** to **Jcc**.
- Depending on the condition (**cc**), the displacement range is either a **signed 8-bit** or **4-bit** value.
- The choice of condition (**cc**) is dependent on whether the test is for a **signed** or **unsigned** computation.

# Different Jcc Conditions

- There are 12 possible conditional tests for **Jcc**.

Mnemonic	Condition Tested	Jump Range	Remarks
<b>JMP (BRA)</b>	Always	-128 to + 127	Unconditional
<b>JHS (JC)</b>	Higher or Same	-128 to + 127	Unsigned only
<b>JLO (JNC)</b>	Lower	-128 to + 127	Unsigned only
<b>JEQ (JZ)</b>	Zero	-128 to + 127	Both
<b>JNE (JNZ)</b>	Not Zero	-128 to + 127	Both
<b>JPL</b>	Positive	-8 to 7	Signed only
<b>JGE</b>	Greater than or Equal	-8 to 7	Signed only
<b>JLT</b>	Less than	-8 to 7	Signed only
<b>JGT</b>	Greater than	-8 to 7	Signed only
<b>JLE</b>	Less than or Equal	-8 to 7	Signed only
<b>JVC</b>	Overflow is Clear	-8 to 7	Signed only
<b>JPE</b>	Parity is Even	-8 to 7	Parity of AR register

# Implementing a simple count loop

- Consider the task of writing 0's to 400 bytes of memory starting at address 0x100:

```

MOV    R2, #400 ;load 400 into counter register
MOV    R0, #0
MOV    R1, #0x100
loop MOV [R1], R0
INC    R1
DEC    R2 ;subtract 1 from counter register
JNE    loop ;loop back 400 times until R2 reaches 0

```

loop back  
if result is  
not zero

Continue sequential execution if result is zero  
(i.e. exit count loop)

Implementation using register  
indirect and the **JNE** condition test

# Another count loop implementation

- Different conditions can be used to implement the counting loop but the **initial count** value must be appropriate to get required loop count.

```

MOV    R2, #399 ;load (400-1) into counter register
MOV    R0, #0
MOV    R1, #0x100
loop   MOV    [R1], R0
      INC    R1
      DEC    R2 ;subtract 1 from counter register
      JPL    loop ;loop back 400 times until R2 reaches -ve
  
```

loop back  
if result is  
**positive**

Continue sequential execution if result is **negative**  
(i.e. exit count loop)

Implementation using register indirect  
and the **JPL** condition test



# Conditional Test – Signed & Unsigned

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

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

- e.g.
 

```

SUB R1,R2 ;R1 = R1 - R2
JPL R1≥R2 ;jump to R1≥R2 if result is positive
:
R1≥R2 :
```




- For testing **unsigned** values, use **HS, LO**.


- e.g.
 

```


SUB R1,R2 ;R1 = R1 - R2
JHS R1≥R2 ;jump to R1≥R2 if R1 higher or equal to R2
:
R1≥R2 :
```

# CMP

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

	SUB	R1, R2
	JPL	R1 ≥ R2
	:	
R1 ≥ R2	:	

**R1 modified to achieve  
desired flow control**

	CMP	R1, R2
	JPL	R1 ≥ R2
	:	
R1 ≥ R2	:	

**Same flow control  
but R1 unchanged**

# Conditional Test using CMP

- Use (**CMP**) instead of (**SUB**) to compare values of two operands without affecting the operands.
- Comparing **signed** register value to an immediate value.

- e.g.

```
CMP    R1, #4    ;test (R1 - 4), where R1 is a signed no.
JGE    R1 ≥ 4    ;jump to R1 ≥ 4 if result is positive
      :
R1 ≥ 4  :
```

- Finding C string terminator (0) in memory.

- e.g.

```
Loop    CMP    [R1], #0 ;test ([R1] - 0)
        JEQ    Found   ;jump to Found if value is 0
        INC    R1      ;increment memory pointer to next char
        JMP    Loop    ;jump back to start of Loop
Found   :
```

# Summary

- Conditional branch (**Jcc**) allows us to implement conditional and loop constructs.
- Appropriate (**Jcc**) 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 **DEC**) are used to set the **N**, **Z**, **V**, **C** flag before conditional test can be done.