DC-v0.69 Micro-Computer
PROGRAMMING MANUAL

Destroyer Corp Inc®

Summer of '69

Preface

This manual is a reference text for the DC-v0.69 micro-computer. It provides a detailed explanation of operating codes and the function of the system's components.

This manual is divided into these sections:

- system details
- instructions & op-codes
- boot process

The sections are independent and need not be used in the order in which they appear. This manual is intended for the use of programmers and systems personnel who do not know the inner-workings of the DC-v0.69 micro-computer. The manual can also be used as a training aid in the instruction of programmers and operators.

Contents

Section 1

System

1.1 Specification

This micro-computer, based on the von-Neumann architecture, consists of the following major components -

- CPU : 8-bit with 16-bit support
- Memory : 64 KB, random-access
- Ret-stack : 128 bytes, stack-based
- Serial I/O

1.2 Memory

The micro-computer comes with a random-access memory, 64 KB in size, with addresses ranging from 0x0000 to 0xFFFF inclusive. Only register-indirect memory addressing is supported in DC-v0.69.

1.3 Ret-stack

DC-v0.69 comes with a stack for managing function calls and returns called *ret-stack*. Whenever a function call is made,via the CALL instruction, the return address is pushed to the top of the stack.

This stack permits the storage of at most 64 return addresses, which is why, the number of maximum nested function calls supported by the processor is also 64. This stack, however, can't be accessed or manipulated directly, but is only affected as a result of certain instructions.

The ret-stack is designed as a memory of size 128 bytes which grows upwards, 0x00 - 0x7F as the addresses in this memory, with the SP register containing the address of the top of this stack, where the most recent return address resides.

1.4 Registers

The processor has 16 general purpose registers, which can be numbered and referenced via numbers 0, 1,. . . , 15. These registers may also be referred to as R0, R1 etc.

Additionally, the processor also provides 3 specific registers, which can only be accessed and manipulated via specific instructions. These 3 registers are namely -

- Instruction Pointer(IP)
- Stack Pointer(SP)
- FLAGS

1.4.1 Instruction Pointer

The instruction pointer, a 16-bit register, contains the address in memory of the instruction that will be executed next. Once this instruction has been executed, IP is now updated to point to the next instruction, if no jump took place.

1.4.2 Stack Pointer

The stack pointer, an 8-bit register, contains the address of the top of the ret-stack.

1.4.3 FLAGS

The FLAGS register is the status register containing the current state of the CPU, and is 8-bit wide in DC-v0.69, consisting of the following bits where 7th bit is the highest bit while 0th bit is the lowest -

7	6	5	4	3	2	1	0
0	0	0	0	OF	SF	CF	ZF

For example, when the FLAGS register contains the value 0x05 (`0000 0101` in binary); OF is 0, SF is 1, CF is 0 and ZF is 1.

Zero Flag(ZF)

This bit is set when the result of the instruction is zero.

Carry Flag(CF)

This bit is set when the instruction resulted in a carry or borrow.

Sign Flag(SF)

This bit is set when the result of the instruction was negative.

Overflow Flag(OF)

This bit is set when the instruction resulted in an overflow or an underflow.

Note that only the `cmp` and `cmpi` instructions affect the FLAGS register, any ALU operations do not. More details on how and when the flags are set have been provided with the comparison instructions.

1.5 I/O

The processor has full serial I/O capabilities via the use of serial `in` and `out` instructions.

Section 2

Instructions & Operation Codes

In this system, instructions are encoded in "Big Endian" format and are always byte-aligned. However, different instructions have different sizes. In what follows, the following terminology is used:

- Rx/Ry/Rz : Any general purpose register
- xx : Any 8-bit immediate value
- xxyy : Any 16-bit immediate value
- Opcode : A 5-bit value

2.1 Memory ops

With just the indirect memory addressing mode, memory accesses in DC-v0.69 require two registers to build a 16-bit address.

Encoding :

Opcode	Rx	Ry	Rz
4 bits	4 bits	4 bits	4 bits

It is important to note the difference between the operands in the load and store instructions.

Opcode	Instruction	Description	Example(hex)	
0	ld Rz, [Rx Ry]	Load the byte at [Rx Ry] in Rz	01 23	ld R3, [R1 R2]
1	st [Rx Ry], Rz	Store the byte at Rz in [Rx Ry]	11 23	st [R1 R2], Rx

2.2 Arithmetic & Logical

It is important to note that none of the following instructions affect the FLAGS register in any manner.

2.2.1 8-bit non-immediate operations

These ALU instructions required two register operands and the result is stored in the first register operand. The opcode for these instructions is fixed as 2 and func decides the ALU operation to be performed.

Encoding :

Opcode	Func	Rx	Ry
4 bits	4 bits	4 bits	4 bits

Func	Instruction	Description	Example(hex)	
0	add Rx, Ry	Rx := Rx + Ry	20 12	add R1, R2
1	sub Rx, Ry	Rx := Rx - Ry	21 12	sub R1, R2
2	mul Rx, Ry	Rx := Rx * Ry	22 12	mul R1, R2
3	div Rx, Ry	Rx := Rx / Ry	23 12	div R1, R2
4	mod Rx, Ry	Rx := Rx mod Ry	24 12	mod R1, R2
5	shl Rx, Ry	Rx := Rx $\ll$ Ry	25 12	shl R1, R2
6	shr Rx, Ry	Rx := Rx $\gg$ Ry	26 12	shr R1, R2
7	and Rx, Ry	Rx := Rx & Ry	27 12	and R1, R2
8	or Rx, Ry	Rx := Rx | Ry	28 12	or R1, R2
9	xor Rx, Ry	Rx := Rx $\oplus$ Ry	29 12	xor R1, R2
10	mov Rx, Ry	Rx := Ry	2A 12	mov R1, R2

Here, all operators operate on unsigned integers, and any overflown bits are discarded. For example, `div R1, R2` with R1 as 5 and R2 as 2 would result in R1 becoming the quotient of the integer division, that is, 2. Similarly, `mul R3, R0` with R3 as 120 and R0 as 3 would result in R3 becoming 104 (360 with overflowing bits discarded); and `add R1, R1` with R1 as 124 results in R1 becoming 248.

2.2.2 8-bit immediate operations

These instructions operate on a register with an immediate(a constant) obtained directly from the bytes of the instruction. These instructions are wider than the non-immediate instructions and the opcode for these instructions is fixed as 3. Similar to the non-immediate instructions, all operators operate on unsigned integers and overflown bits are discarded. Encoding :

Opcode	Func	R	Unused	xx
4 bits	4 bits	4 bits	4 bits	8 bits

Func	Instruction	Description	Example(hex)	
0	addi Rx, xx	Rx := Rx + xx	30 10 10	addi R1, 0x10
1	subi Rx, xx	Rx := Rx - xx	31 10 10	subi R1, 0x10
2	muli Rx, xx	Rx := Rx * xx	32 10 10	muli R1, 0x10
3	divi Rx, xx	Rx := Rx / xx	33 10 10	divi R1, 0x10
4	modi Rx, xx	Rx := Rx mod xx	34 10 10	modi R1, 0x10
5	shli Rx, xx	Rx := Rx $\ll$ xx	35 10 01	shli R1, 0x01
6	shri Rx, xx	Rx := Rx $\gg$ xx	36 10 01	shri R1, 0x01
7	andi Rx, xx	Rx := Rx & xx	37 10 10	andi R1, 0x10
8	ori Rx, xx	Rx := Rx | xx	38 10 10	ori R1, 0x10
9	xori Rx, xx	Rx := Rx $\oplus$ xx	39 10 10	xori R1, 0x10
10	movi Rx, xx	Rx := xx	3A 10 10	movi R1, 0x10

2.3 Comparison

To perform loops and jumps, comparisons are required. These instructions evaluate the difference between the two operands and set the FLAGS register based on the obtained

difference. The operand registers are not affected in any manner.

Encoding :

Opcode	Rx	Ry/xx
4 bits	4 bits	8 bits

Opcode	Instruction	Description	Example(hex)	
4	cmp Rx, Ry	Computes Rx - Ry & sets the flags accordingly	41 02	cmp R1, R2
5	cmpi Rx, xx	Computes Rx - xx & sets the flags accordingly	51 10	cmpi R1, 0x10

The flags are set or reset based on the result of the performed computation. `ZF` is set when the result was simply zero. `CF` is set when the operands are treated as unsigned numbers and a borrow was required to obtain the result, since a carry can not be generated when performing subtraction between unsigned numbers. `SF` is set if the result obtained in negative while treating the operands as signed numbers. While `OF` is set if the operation resulted in an overflow or an underflow when the operands are treated as signed numbers. These flags are reset if their corresponding conditions turn out to be false.

For example, if R3 contains the byte 0x7F and R5 contains 0x81, the instruction `cmp R3, R5` would compute the unsigned result as 254 after performing a borrow, and the signed result would be computed as -2 by treating R3 as 127 & R5 as -127 and encountering an underflow. Thus, ZF will be reset while CF, SF and OF would be set. Similarly, if R3 were 0x81 and 0x7F, both the unsigned and the signed result would be obtained as 2, after encountering an overflow in signed difference of -127 and 127 respectively. Therefore, OF would be set while ZF, CF and SF are reset in this case.

2.4 Control Flow

Sequential control-flow of the machine can be altered in two ways - either by jumps or by function calls & returns.

2.4.1 Jumps

Encoding :

Opcode	Unused	xxyy
4 bits	4 bits	16 bits

Opcode	Instruction	Description	Example(hex)	
6	jmp xxyy	Jump to xxyy	60 00 01	jmp 0x0001
7	je xxyy	Jump to xxyy if ZF = 1	70 00 01	je 0x0001
8	jne xxyy	Jump to xxyy if ZF = 0	80 00 01	jne 0x0001
9	jl xxyy	Jump to xxyy if SF != OF	90 00 01	jl 0x0001
10	jg xxyy	Jump to xxyy if SF = OF & ZF = 0	A0 00 01	jg 0x0001

The FLAGS register is not reset when a jump is performed. It must be ensured by the programmer that the flags being checked are therefore not stale but have been computed recently.

2.4.2 Call

Encoding :

Opcode	Unused	xxyy
4 bits	4 bits	16 bits

Opcode	Instruction	Description	Example(hex)	
11	call xxyy	Call function at xxyy; pushes return address to ret-stack, updates IP and decrements SP by 2	B0 00 01	call 0x0001

For example, suppose IP was initially 0x0000 and SP was 0x02 and the instruction encountered at IP is `call 0x1000`; then the next instruction's address, 0x0003, is pushed to the ret-stack. With this push, SP is decremented to 0x00 since this is the new top of the stack, and IP is now updated to 0x1000, from where the execution now continues.
It is also important to note that at most 64 nested function calls are permitted by the processor, owing to the ret-stack size. Any more function calls result in the CPU crashing and the machine gets toasted.

2.4.3 Return

Encoding :

Opcode	Unused
4 bits	4 bits

Opcode	Instruction	Description	Example(hex)	
12	ret	Returns to caller function, resuming execution from saved return address	C0	ret

Continuing from the previous example, suppose a `ret` instruction is encountered next. The top of the stack would be popped off and the obtained return address would be the new IP. Due to a pop, the SP has been incremented to 0x02 from 0x00. The execution then continues from the new IP.
Note that returns on empty ret-stack will cause the CPU to crash, and the machine gets toasted.

2.5 Serial I/O

The processor performs serial I/O by reading or writing one byte at a time. Two buffers have been provided with the processor, one input and another output buffer for reading the input and writing the output respectively. Reads and writes can only be done using registers.

Encoding :

Opcode	R
4 bits	4 bits

Opcode	Instruction	Description	Example(hex)	
13	in Rx	Read one byte from serial in to Rx	D5	in R5
14	out Rx	Outputs contents of Rx to serial out	E5	out R5

For example, the instruction `in R5` would lead to a byte being read from input into register R5. Similarly, the instruction `out R3` would lead the contents of register R5, which is precisely one byte, to be written to output.

2.6 Misc

It is important to note that any invalid instruction, whether it be an unused opcode, an unused `func` or an instruction where the states of the machine are being led beyond permissible ranges, will cause the CPU to crash and the machines to get toasted. A toasted machine can not be used again. However, there's also a way to peacefully halt the machine using the `halt` instruction.

Encoding :

Opcode	Unused
4 bits	4 bits

Opcode	Instruction	Description	Example(hex)	
15	halt	Halts the machine	F0	halt

Section 3

Boot Process

On boot-up, IP is initialized to 0x0000, SP to 0x80, the reg-stack as empty, and all general-purpose registers as well as FLAGS register are reset. The memory is also initialized with all zeros. The ROM is then loaded at 0x0000 and then, the execution begins.