Elektronite

presents

Intellivision White Papers

Paper #2

Programming the Intellivision I

Get started with the CP1610

Version 1.0

[Authors]

Valter Prette

Joseph Zbiciack

Contents

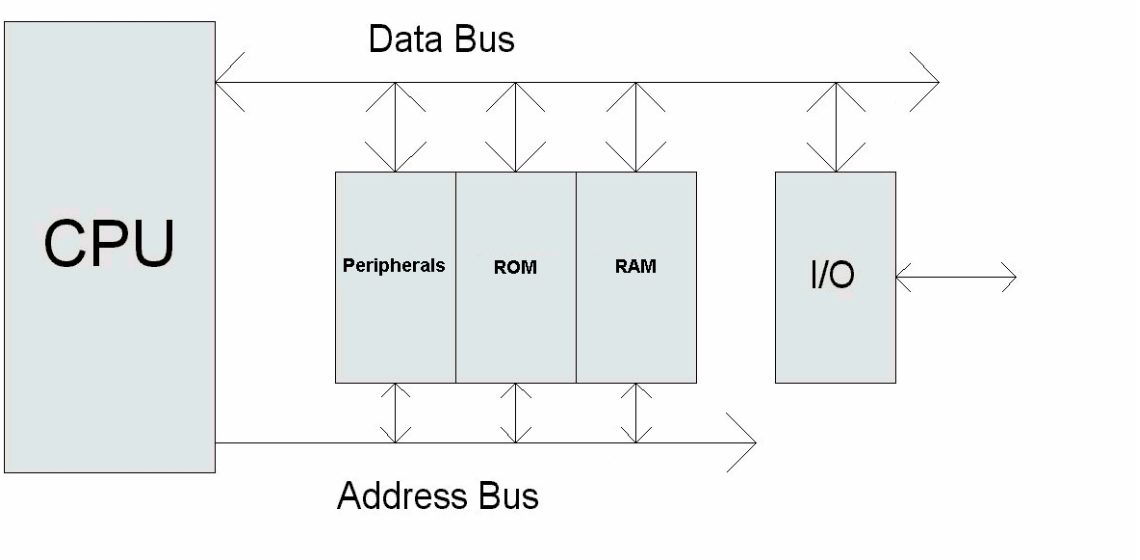
Introduction

Welcome to the first Intellivision programming lesson.
This white paper is intended for people who do not know much about this platform and want to learn writing applications for it.
We hope that you find this documentation useful and that you get involved in the exciting Intellivision homebrew community.

Developing software for Intellivision means to write assembler code for a CP1610 based hardware.
Before to study the language itself, you need to learn the basics of the hardware to understand why you need to perform a specific action in order to have the result you're looking for.
In particular, you need to know how to refer to memory registers, how to pilot the STIC and how to deal with interrupts.
All those concepts will be introduced step by step during the lessons, to make the learning process as smooth as possible.
This first lesson will introduce the CP1610 system and the routines that you will use starting from lesson two.
This document is also intended as reference of the assembly commands available for the Intellivision platform.

CP1610 General Overview

The CP-1600 provides what's known as Von Neumann style computer architecture. That means a Central Processing Unit which is connected by a single bus to several memories and peripherals. In our case, the Central Processing Unit is the CP-1600. Conceptually, the diagram looks like so:



All of the devices that are outside the CPU appear in a single, unified Address Space. In our case, addresses are 16-bits wide, and so the address space is 64K words large.
Programs and data are stored in memory, which the CPU accesses via bus. The CPU makes no distinction between whether a memory holds program code or data, so both can be stored in the same memory (but usually at different locations).
Also, memories and peripherals are treated identically, so accesses to "memory" may go to either RAMs, ROMs, or various peripherals.

Registers

Memory accesses are slow, because they have to go "off chip" for data. As a result, most operations in the CPU operate on registers. Registers are special memory locations inside the CPU that are connected directly to the CPU's arithmetic and logic units. Most of these registers are so-called "General Purpose" registers, although also many have additional special uses assigned to them.
The CP-1600 has 8 16-bit general purpose registers, named R0 through R7.
Additionally, there is the status word, SWD, which contains the status bits.
You need to become familiar with this nomenclature because the register will appear

anywhere in the Intellivision assembler code.
The following table describes the registers

Register	Special Purpose
R0	None
R1	Data Counter
R2	Data Counter
R3	Data Counter
R4	Auto-incr Data Counter, or JSR Return Address
R5	Auto-incr Data Counter, or JSR Return Address
R6	Stack Data Counter, or JSR Return Address
R7	Program Counter
SWD	Status word: Holds Sign, Zero, Carry, Over bits

In addition to providing access to memory, R4 and R5 auto-increment and R6 behaves as a stack pointer when used as Data Counters.
Registers R4 through R6 also may be used to hold the return-address for a JSR instruction. JSR acts very similar to BASIC's GOSUB instruction. The return address may then be saved in memory by your program.

Addressing modes

The CP-1600 offers a wide variety of addressing modes for its instruction set. Some of these modes operate entirely on registers inside the CPU. Others access memory, allowing access to RAMs, ROMs, and peripheral devices.
Before launching into a complete description of the modes, let's first look at the common instruction forms.

Single-operand instructions, such as "INCR", "DECR" and so on work with a single operand that doubles as both "source" and "destination."
Dual-operand instructions, such as "ADDR" and "SUBR" operate on two operands, where the first is a "source" and the second is both "source" and "destination."
For example, consider the single operand instruction "INCR Rx" (INCR stands for "INCrement Register"). Rx acts both as a source (input) and destination (output) for the instruction.
Now, consider the two-operand instruction "ADDR Rx, Ry" (ADDR stands for "ADD Registers"). Here, Rx is simply a source operand, and Ry is both a source, and a destination.

Let’s have a look to the addressing modes now.
The simplest addressing mode is "Implied" mode, in which the instruction operates on one or more operands that are not directly specified. Instructions which directly set/clear flags fall into this category.

Example:

```
CLRC            ; C = 0 (Clear the carry bit.)
```

The next simplest addressing mode is "Register" mode, in which the instruction reads and writes all of its results to registers.
Register mode instructions generally have an "R" as the last letter of their mnemonic, as in "ADDR", "COMR", etc. "Register" instructions do not access memory.

Examples:

```
INCR  R0        ; R0 = R0 + 1
ADDR  R1, R2    ; R2 = R2 + R1
SUBR  R3, R4    ; R4 = R4 - R3
```

"Immediate" mode instructions accept a constant as one of the two operands. These instructions are always dual-operand instructions, and the first operand is always the constant, with exception to "MVOI," which writes to its immediate operand.
Immediate mode instructions generally have an "I" as the last letter of the mnemonic, as in "ADDI".

Examples:

```
MVII  #$0042, R3        ; R3 = $0042
XORI  #$FF00, R6        ; R6 = R6 XOR $FF00
```

"Direct" mode instructions specify a fixed memory address from which to read one of the operands. As with Immediate mode, the first operand is always the direct operand, except for MVO, which writes a value to the requested address.

Examples:

```
MVO   R4, $01F1         ; POKE $01F1, R4
MVI   $01F0, R3         ; R3 = PEEK($01F0)
ADD   $02F0, R5         ; R5 = R5 + PEEK($02F0)
```

"Indirect" mode instructions access memory through a Data Counter register for one of the operands. The first operand is generally the data counter, with exception to "MVO@," in which it is the second operand. Except in the case of "MVO@," Indirect mode instructions read the value at the memory location pointed to by the Data Counter before performing the instruction.
With "MVO@", the value is written to the desired location. Indirect mode instructions are generally noted with a "@" at the end of the mnemonic.

When an Auto-Incrementing Data Counter is used with Indirect mode, the Data Counter is incremented after the access. This allows loops to step through arrays very efficiently, since it is not necessary to manually update the Data Counters.

The Stack Data Counter is a special case. When writing, it is incremented after the access, just as the Auto-Incrementing Data Counters are. When reading, however, it is decremented before the access, thus providing a simplistic stack. Indirect addressing via the Stack Data Counter is referred to as "Stack Addressing."

Examples:

```
MVO@  R4, R3          ; POKE R3, R4
MVO@  R3, R4          ; POKE R4, R3 : R4 = R4 + 1     (Auto-incr)
MVO@  R3, R6          ; POKE R6, R3 : R6 = R6 + 1     (Stack)

MVI@  R3, R4          ; R4 = PEEK(R3)
MVI@  R4, R3          ; R3 = PEEK(R4) : R4 = R4 + 1   (Auto-incr)
MVI@  R6, R3          ; R6 = R6 - 1 : R3 = PEEK(R6)   (Stack)

XOR@  R3, R2          ; R2 = R2 XOR PEEK(R3)
```

The Exec and the JzIntv library

The whole system is controlled by a 4K program called the Executive (the EXEC) which resides in the Intellivision Executive ROM chip (with an additional 344 bytes over in the GROM chip).
In a sense, the exec is the main game program, and the plug-in cartridge merely contains subroutines and data which are used by the EXEC.
Normally, only EXEC routines access GROM, GRAM and the STIC control registers.
The EXEC contains routines for moving objects around the screen, loading GRAM, creating sound and music, testing for moving object interaction, etc.
Using the EXEC is the standard solution adopted by early games on the system.
Today the developers have the possibility to use and alternative library of routines programmed by J. Zbiciack and delivered under GNU license.
The JzIntv library is a user friendly set of functions that cover the EXEC scope and add some more option to the developer.
That library will be presented in future documentation; from next lesson on, you would better studying the standard routines.

Appendix: CP1610 Instruction tables

Those tables summarize all the assembler commands available for programming the Intellivision CPU.
If you're not used to assembler, at the moment you may not understand perfectly the meaning, but you can use the tables as reference for the next programming lessons.

Register To Register			
Command	**Operation**	**Microcycles**	**Comments**
MOVR	MOVe Register	6/7*	
TSTR	TeST Register	6/7*	MOVR to itself
JR	Jump to address in Register	7	MOVR to PC
ADDR	ADD contents of Registers	6/7*	
SUBR	SUBtract contents of Registers	6/7*	Results not stored
CMPR	CoMPare Registers by subtr	6/7*	
ANDR	logical AND Registers	6/7*	
XORR	eXclusive OR Registers	6/7*	
CLRR	CLeaR Register	6/7*	XORR with itself

*7 cycles if destination register is R6 or R7, 6 cycles otherwise.

Single Register			
Command	**Operation**	**Microcycles**	**Comments**
INCR	INCrement Register	6/7*	
DECR	DECrement Register	6/7*	
COMR	COMplement Register	6/7*	One's Complement
NEGR	NEGate Register	6/7*	Two's Complement
ADCR	Add Carry Bit to Register	6/7*	
GSWD	Get Status WorD	6/7*	
NOP	No OPeration	6/7*	
SIN	Software INterrupt	6/7*	Pulse to PCIT pin
RSWD	Return Status WorD	6/7*	

*7 cycles if destination register is R6 or R7, 6 cycles otherwise.

Register Shift			
Command	**Operation**	**Microcycles**	**Comments**
SWAP	SWAP 8-bit bytes	6	
SLL	Shift Logical Left	6	
RLC	Rotate Left thru Carry	6	
SLLC	Shift Logical Left thru Carry	6	
SLR	Shift Logical Right	6	
SAR	Shift Arithmetic Right	6	
RRC	Rotate Right thru Carry	6	
SARC	Shift Arithmetic Right thru Carry	6	

Shift is not interruptible. One or two position shift capability.
Two positions SWAP replicates lower byte in both halves.

Control Instructions			
Command	**Operation**	**Microcycles**	**Comments**
HLT	HaLT	4	Halts machine completely
SDBD	Set Double Byte Data	4	Must precede external
EIS	Enable Interrupt System	4	
DIS	Disable Interrupt System	4	
TCI	Terminate Current Interrupt	4	
CLRC	CLeaR Carry to zero	4	
SETC	SET Carry to one	4	

Jump Instructions			
Command	**Operation**	**Microcycles**	**Comments**
J	Jump	12	
JE	Jump, Enable, interrupt	12	
JD	Jump, Disable interrupt	12	
JSR	Jump, Save Return	12	
JSRE	Jump, Save Return & Enable	12	Return Address saved in R4, R5 or R6
JSRD	Jump, Save Return & Disable interrupt	12	

Conditional Branch Instructions

Command	Operation	Microcycles	Comments
B	unconditional Branch	9	
NOPP	NO OPeration	7	Two words
BC	Branch on Carry	7*	C = 1
BNC	Branch on No Carry	7*	C = 0
BOV	Branch on OVerflow	7*	OV = 1
BNOV	Branch on No OVerflow	7*	OV = 0
BPL	Branch on PLus	7*	S = 0
BMI	Branch on Minus	7*	S = 1
BEQ	Branch if Not Zero or Not EQual	7*	Z = 1
BNEQ	Branch if Not Zero or Not EQual	7*	Z = 1
BLT	Branch if Less Than	7*	S XOR OV = 1
BGE	Branch if Greater than or Equal	7*	S XOR OV = 0
BLE	Branch if Less than or Equal	7*	Z OR (S XOR OV) = 1
BGT	Branch if Greater Than	7*	Z OR (S XOR OV) = 0
BUSC	Branch if Sign not = Carry	7*	C XOR S = 1
BESC	Branch if Sign = Carry	7*	C XOR S = 0
BEXT	Branch if External condition is true	7*	

* Add 2 cycles if test condition is true

Input/Output

Command	Operation	Microcycles	Comments
MVO	MoVe Out	9 or 11	Not interruptible
PSHR	PuSH Register to Stack	9	PSHR = MVO@R6 Not interruptible
MVI	MoVe in	8 to 11	
PULR	PULl from stack to Register	11	PULR = MVI@R6

Arithmetic & Logic

Command	Operation	Microcycles	Comments
ADD	ADD	8 to 11	
SUB	SUBtract	8 to 11	
CMP	CoMPare	8 to 11	Result not saved
AND	logical AND	8 to 11	
XOR	eXclusive OR	8 to 11	