INTELLIVISION WIKI TUTORIAL

Assembly Syntax Overview

This offers a brief overview of assembly syntax as implemented in as1600. Please refer to the official documentation for complete details: as1600.pdf (http://spatula-city.org/~im14u2c/intv/jzintv-1.0-beta3/doc/utilities/as1600.pdf) and as1600.txt (http://spatula-city.org/~im14u2c/intv/jzintv-1.0-beta3/doc/utilities/as1600.txt)

Contents

Basic Format

Assembly code is a line oriented language, similar to languages like BASIC, FORTRAN, COBOL and many scripting languages. Each assembly code line divides into 4 major components:

Label	Instruction or directive	Operands	Comment
`foo:`	`MOVR`	`R0, R1`	`; Copy R0 to R1`

Labels

Labels always start in the first column. Labels give a name to an address or a value. The assembler resolves labels into numbers when it assembles the program. There are two sorts of labels:

- **Global labels.** These start with one of these characters— `a-zA-Z!&_^~.` —followed by zero or more of these: `0-9a-zA-Z!&_^~.` Examples: "FOO", "abc", "_a1", ".blah".

- **Local labels.** These start with two @ signs, such as "@@loop". These labels are local to a PROC section. See the PROC directive below.

Other properties of labels:

- Labels must have distinct names from instruction mnemonics, directives and operators (defined below).
- Labels optionally may be followed by a colon or by white space. That is, both "foo" and "foo:" are allowed.

Labels are optional in most cases. Some directives, such as PROC, EQU and SET require a label.

Instructions and Directives

The next field is the instruction/directive field. Instructions and directives are always preceded by at least one white-space character, or a label with a colon. The CP1610 page lists all of the instructions.

Operand Order

Many instructions have two operands. In CP1610 assembly code, the last operand is generally the "destination" of the result, and the operands before it are "sources." For example:

```
 ADDR R0, R1  ; Compute "R0 + R1" and put the result in "R1".
```

A couple instructions, like JSR don't quite fit that model:

```
 JSR R5, foo  ; Jump to "foo", and put return address in "R5."
```

Common Directives

- `ROMW`: Sets the current "ROM width." In most cases, you can use "ROMW 16", which is also the default. Set this once per program.
- `ORG`: Sets the current "origin," that is, the address that the assembler is currently assembling at.
- `INCLUDE`: Copies in an outside file into the current assembly. Makes it easy to factor a large program into many files.
- `PROC / ENDP`: These define sections of code. They are useful primarily for defining a scope for local variables. Must be used with a label.
- `DECLE`: Outputs one or more words of data to the ROM. Can be followed by comma separated expressions and strings. `STRING` is a synonym.
- `BIDECLE`: Similar to DECLE, except it writes words as pairs of bytes. For each word, it writes the lower byte first and the upper byte second. It's most useful with data read using SDBD.
- `EQU`: Sets a label to a value, and therefore must be used with a label. Often useful for giving meaningful names to memory locations and other constants. Values set by EQU cannot be changed.
- `SET`: Similar to EQU, except that the value assigned to the label can be changed later. This can be useful with the REPEAT directive to compute complicated expressions.
- `REPEAT / ENDR`: Repeats a block of code multiple times. The repeated code is merely replayed for the assembler as if you had cut and pasted it the specified number of times. Useful for unrolling loops or computing complicated expressions at assembly time.

- `IF / ELSE / ENDI`: Conditional assembly directives. Useful for guarding pieces of code from being assembled, or otherwise controlling the assembly process based on the values of some labels.

The example at the end shows each of these in use.

Using PROC / ENDP

PROC sections bear special mention. PROC is short for "procedure." The PROC directive indicates the start of a "local" name space. Local labels, those starting with "@@", become local to that section. Within that section, those labels may be referred to by their "@@" name.

Local labels also get a corresponding global name. The assembler strips off the @@, and prepends the PROC's name and a period. This makes it easy to refer to a PROC's local labels, and a handy way of grouping sets of names.

Example:

```
;; Loop 42 times and return
FOO     PROC

        MVII    #42,  R0
@@loop: DECR    R0
        BNEQ    @@loop

        JR      R5
        ENDP

;; Loop 57 times and return
BAR     PROC
        MVII    #57,  R0
        B       FOO.loop   ; Reuse FOO's loop.
        ENDP
```

In this example, the branch in FOO refers the local label "@@loop" by its local name. The branch in BAR refers to the same label by its global name, "FOO.loop".

Comments

Comments are optional, but recommended. Comments start with a semicolon '`;`', and end at the end of the line. Comments can start anywhere on a line, including the first column.

Expressions

Expressions are mathematical statements consisting of one or more numbers or labels, connected by various operators. Most expressions evaluate down to a single value. One special case is the "string expression", which expands to a list of words. String expressions are only valid in a limited number of contexts.

Numeric Formats

The assembler accepts numbers in four basic formats:

Format	Example	Notes
Decimal	12345	Uses digits 0 - 9. Must not start with 0 unless the value is exactly 0.
Octal	0377	Uses digits 0 - 7. Always starts with a leading 0.
Hexadecimal	$5A3C	Uses digits 0 - 9, A - F. Case insensitive. Starts with $.
Binary	%01011010	Uses digits 0, 1. Starts with %.

Strings

Strings are quoted expressions. They may be used directly with DECLE, BIDECLE, STRING and BYTE directives, or as an argument to the ASC() and STRLEN() operators. Characters within strings may be escaped using the backslash \. See the as1600 documentation for more details.

When used with DECLE, BIDECLE, STRING and BYTE directives, strings may be intermixed with other expressions, separated by commas. A common use for this is to terminate strings with a NUL (0) character.

Examples:

```
    DECLE  "Hello World!", 0
    STRING "Hello World!", 0  ; synonymous with previous line
```

Operators

The following table lists some common operators that appear in expressions. This list is not exhaustive. Please refer to the as1600 documentation for a complete list.

+	Addition
-	Subtraction
*	Multiplication
/	Division
MOD	Modulo (remainder)
SHL	Shift left
SHR	Shift right
AND	Bitwise logical AND
OR	Bitwise logical OR
XOR	Bitwise logical XOR
NOT	Bitwise "NOT"

>	**GT**	Greater Than (signed)
>=	**GE**	Greater Than or Equal (signed)
<	**LT**	Less Than (signed)
<=	**LE**	Less Than or Equal (signed)
=	**EQ**	Equal
<>	**NE**	Not Equal
ASC("*string*",*index*)		Returns ASCII code of indexed character
STRLEN("*string*")		Returns the length of a string
DEFINED		Returns 1 if symbol is defined at this point (first pass)

Forward References

Expressions are computed at assembly time, which makes them a powerful method for computing complex values that will be fixed at run-time. This means that the values for all labels that appear in the expression must be known at assembly time. Expressions may include forward references, meaning references to labels whose values are not yet known. This is usually ok, since the assembler makes a second pass to finish computing these expressions. Certain directives, such as IF, EQU and REPEAT, need to know the exact value of the expression when the directive is reached the first time, though. The assembler will inform you if you use an expression whose value can't be computed when it needs it.

Example:

```
        ; This code has legal forward references.  The assembler will accept these:
TABLE   DECLE   (FOO AND $7F8) SHR 3, ((FOO AND $FC00) SHR 11) OR ((FOO AND $3) SHL 5)
        DECLE   (BAR AND $7F8) SHR 3, ((BAR AND $FC00) SHR 11) OR ((BAR AND $3) SHL 5)
FOO     PROC
        ; ... more code here
        ENDP
BAR     PROC
        ; ... more code here
        ENDP
```

```
        ; This code, however, does not have legal forward references.
        ; The assembler will give an error for each EQU directive:
CST1    EQU     (FOO AND $7F8) SHR 3, ((FOO AND $FC00) SHR 11) OR ((FOO AND $3) SHL 5)
CST2    EQU     (FOO AND $7F8) SHR 3, ((FOO AND $FC00) SHR 11) OR ((FOO AND $3) SHL 5)
FOO     PROC
        ; ... more code here
        ENDP
BAR     PROC
        ; ... more code here
        ENDP
```

Also, the assembler will issue a warning if you have a forward reference to an EQU or SET directive. It's usually safe to ignore the warnings for EQU directives, but not for SET directives if the label's value gets SET more than once. The following examples trigger warnings:

```
        ; This triggers a warning that's probably safe to ignore, but still worth fixing.
        DECLE   FOO, BAR, BAZ

        ; ... more code here

FOO     EQU     42
BAR     EQU     17
BAZ     EQU     23
```

```
        ; This also triggers a warning, and may not do what the programmer intended
        DECLE   FOO

        ; ... more code here

FOO     SET      42    ;\
FOO     SET      17    ; |- Which of these gets used in the DECLE above?
FOO     SET      23    ;/
```

Precision

Expressions are computed with 32-bit precision. This makes it easy to do extended precision work. If you try to use a 32-bit value as an operand to an instruction or one of the data directives (e.g. DECLE), though, the assembler will complain that the value is out of range. The assembler allows 32-bit values in the range $FFFF8000 (-32768) to $0000FFFF (+65535) to silently truncate to 16 bits—that is, they will truncate without an error.

Examples:

```
MAGIC   EQU     $4A5A6A7A           ; Set up magic 32-bit constant

        DECLE   MAGIC               ; Causes an error
        DECLE   MAGIC AND $FFFF     ; Valid
        DECLE   MAGIC SHR 16        ; Also valid

        MVII    #MAGIC, R0          ; Causes an error
        MVII    #MAGIC AND $FFFF, R0 ; Valid
        MVII    #MAGIC SHR 16, R0   ; Also valid

NUM1    EQU     $FFFF8000           ; -32768 as a 32-bit number
NUM2    EQU     $00008000           ; +32768 (unsigned) as a 32-bit number

        MVII    #NUM1, R0           ; sets R0 to $8000
        MVII    #NUM2, R0           ; also sets R0 to $8000
```

(Note: Older versions of as1600 will give an error on any values outside the range of $00000000 - $0000FFFF. Upgrade to the latest as1600 to fix this.)

Example

The following (largely nonsensical) example shows many of the above basic features in use.

```
        ROMW    16    ; Set ROM width to 16-bit
        ORG     $5000 ; Begin assembling code at location $5000

ROW     EQU     10    ; Row to display copyright on
COL     EQU     1     ; Column to begin displaying copyright in

;------------------------------------------------------------------------------
; EXEC-friendly ROM header.
;------------------------------------------------------------------------------
```

```
ROMHDR: BIDECLE ZERO            ; MOB picture base   (points to NULL list)
        BIDECLE ZERO            ; Process table      (points to NULL list)
        BIDECLE MAIN            ; Program start address
        BIDECLE ZERO            ; Bkgnd picture base (points to NULL list)
        BIDECLE ONES            ; GRAM pictures      (points to NULL list)
        BIDECLE TITLE           ; Cartridge title/date
        DECLE   $03C0           ; Flags:  No ECS title, run code after title,
                                ; ... no clicks
ZERO:   DECLE   $0000           ; Screen border control
        DECLE   $0000           ; 0 = color stack, 1 = f/b mode
ONES:   DECLE   1, 1, 1, 1, 1   ; Color stack initialization
;------------------------------------------------------------------------------

TITLE   DECLE   107, "Hello World!", 0

MAIN    PROC    ; Start a PROC/ENDP section

        EIS                     ; Enable interrupts

        CALL    PRINT.FLS
        DECLE   7                    ; 7 is the color number for "white"
        DECLE   $200 + ROW*20 + COL  ; Expression, including labels
        DECLE   "  Copyright 2007  ", 0

    ; The following piece of code is disabled
    IF 0
        MVII    #1234,  R0
    ENDI

        ; Copy 8 words from $120 to $128 quickly
        MVII   #$120,   R4
        MVII   #$128,   R5

        REPEAT 8
        MVI@   R4,      R0
        MVO@   R0,      R5
        ENDR

        ; Spin forever:
@@loop  B      @@loop            ; Branches to local "@@loop" label, aka "MAIN.loop"
        ENDP   ; Close a PROC/ENDP section

        INCLUDE "print.asm"
```

Hello World Tutorial

This tutorial serves as a quick-start, quickly introducing a simple program that displays the famous phrase "Hello World" on an Intellivision title screen. In this tutorial, you will see how to invoke the assembler and how to call a library function from SDK-1600 (http://sdk-1600.spatula-city.org/) named PRINT (http://sdk-1600.spatula-city.org/examples/library/print.asm) . This particular tutorial does not go into all the details underlying how it works. Dig into the more specific tutorials for deeper dives on all the topics this example touches.

Contents

Example 1: Hello World!

Let's start with the first example, and then pull it apart. Here is `hello1.asm`:

```
        ROMW    16
        ORG     $5000

;------------------------------------------------------------------------------
; EXEC-friendly ROM header.
;------------------------------------------------------------------------------
ROMHDR: BIDECLE ZERO            ; MOB picture base   (points to NULL list)
        BIDECLE ZERO            ; Process table      (points to NULL list)
        BIDECLE MAIN            ; Program start address
        BIDECLE ZERO            ; Bkgnd picture base (points to NULL list)
        BIDECLE ONES            ; GRAM pictures      (points to NULL list)
        BIDECLE TITLE           ; Cartridge title/date
        DECLE   $03C0           ; Flags:  No ECS title, run code after title,
                                ; ... no clicks
ZERO:   DECLE   $0000           ; Screen border control
        DECLE   $0000           ; 0 = color stack, 1 = f/b mode
        BIDECLE ONES            ; GRAM pictures      (points to NULL list)
ONES:   DECLE   1, 1, 1, 1, 1   ; Color stack initialization
;------------------------------------------------------------------------------
```

```
TITLE   DECLE   107, "Hello World!", 0

MAIN    EIS                     ; Enable interrupts
here    B       here            ; Spin forever.
```

ROM Width and Origin

The program starts off with a couple directives. These are commands to the assembler that tell it to do various things.

```
        ROMW    16
        ORG     $5000
```

The "ROMW" directive sets the "ROM width." Traditional Intellivision games used 10-bit wide ROMs. Modern games can use a full 16-bit wide ROM. Although the assembler defaults to 16 bits, it doesn't hurt to make it explicit.

The "ORG" directive sets the current "origin". What is this, and why does it matter? The assembler is not a compiler. Compilers take high level code, rearrange it and translate it, and ultimately bake it down into a series of instructions that run on the CPU. An assembler does much, much less than that. It merely assembles (as in "puts together") what you've written. The assembler internally keeps an image of what should be in memory based on what you've told it. When the assembler starts, that image is empty. As you give it instructions to assemble, the assembler converts these to machine code and inserts them into this virtual memory image. The ORG directive tells it where to start writing—a "cursor," if you will.

A program can have more than one ORG statement. The ORG statement moves the assembler's cursor (aka. the "SPC" or "Section Program Counter"), to a new spot. This is useful in cases where not all of your ROM is contiguous. That's a topic for another tutorial.

So why "`ORG $5000`"? The Intellivision's EXEC expects most cartridges to start at this address. It looks for a cartridge header there, and expects that header to tell it something interesting and useful about the program, which leads us to the next part of the program, the cartridge header.

Cartridge Header

```
;------------------------------------------------------------------------------
; EXEC-friendly ROM header.
;------------------------------------------------------------------------------
ROMHDR: BIDECLE ZERO            ; MOB picture base   (points to NULL list)
        BIDECLE ZERO            ; Process table      (points to NULL list)
        BIDECLE MAIN            ; Program start address
        BIDECLE ZERO            ; Bkgnd picture base (points to NULL list)
        BIDECLE ONES            ; GRAM pictures      (points to NULL list)
        BIDECLE TITLE           ; Cartridge title/date
        DECLE   $03C0           ; Flags:  No ECS title, run code after title,
                                ; ... no clicks
ZERO:   DECLE   $0000           ; Screen border control
        DECLE   $0000           ; 0 = color stack, 1 = f/b mode
ONES:   DECLE   C_BLU, C_BLU    ; Initial color stack 0 and 1: Blue
        DECLE   C_BLU, C_BLU    ; Initial color stack 2 and 3: Blue
        DECLE   C_BLU           ; Initial border color: Blue
;------------------------------------------------------------------------------
```

I'm not actually going to explain the entire contents of the EXEC header, because quite honestly, I don't know (or care) what most of the fields do. You're welcome to use this header in your own programs. It passes muster with the EXEC, and it bypasses the ECS title screen on machines with an ECS. The main point of this particular header is to provide the EXEC the minimum amount of information it needs to jump to your code and get everything started. My examples will bypass the EXEC whenever possible.

The most important fields to worry about are the one labeled "Cartridge title/date", and the one labeled "Flags". The first one points to the cartridge's copyright date as well as the title string. The second one holds a set of flags. Here's what they do:

bit 15..10	Unused.
bit 9..8	Unknown. Set both to 1 to skip ECS title screen.
bit 7	Run code that appears after title string.
bit 6	Set to 1 to work on Intellivision 2.
bit 5..0	What inputs the EXEC will make "click" sounds for

You can ignore bits 5..0 if you don't use the EXEC. The "keyclick" functionality only works if you rely on the EXEC for keypad input. None of my programs uses the EXEC.

There is a third field of interest, "MAIN". If you bypass the EXEC as I do, and don't rely on the EXEC to handle starting the game after the title screen, then this field is never actually used. Otherwise, this field points to the first line of program code that the EXEC will call after the user dismisses the EXEC's title screen.

As shown, the header above will bypass most of the EXEC's automatic setup, which is fine. None of my examples will rely on the EXEC to do anything. (Well, other than put up the title screen in this case.) This brings us to the next piece of code, the date and title.

Date and Title

The program indicates the date for the program's copyright as well as its title in a simple manner:

```
TITLE   DECLE   107, "Hello World!", 0
```

As I noted before, the "title" field points at the copyright date and the title string. The copyright date is merely the year minus 1900. (Keith Robinson mentioned that the original documentation actually said "last two digits of the copyright year." Given that we're past Y2K, let's stick with "year - 1900". Programs such as jzIntv (http://spatula-city.org/~im14u2c/intv/) expect the date in that format.) The copyright string itself is NUL-terminated, which means simply that it's followed by a word containing "0".

The DECLE directive tells the assembler to insert the data that follows it into the ROM. DECLE originally referred to the 10-bit words that the old games had to work with in their 10-bit wide ROMs. as1600 expanded the concept to mean "words that are the width of the ROM, as set by ROMW." Since we set the ROM width to 16 up front, the DECLE directive inserts a series of 16-bit words into memory.

A note on memory here: The CP-1600 is what's known as a "word addressable" machine. What this means is that each location in the address map holds a single word. Words are 16 bits wide. Narrower memory, such as 10-bit ROMs and 8-bit RAM work by filling the upper bits of the word with 0s. So, if you read the EXEC ROM, you'll find bits 10-15 always read as 0. If we had set "ROMW 10" instead of "ROMW 16", the DECLE directive would expand our data to 10 bit words, filling the upper 6 bits with 0s.

The assembler also has some other directives for initializing memory, such as BYTE, WORD and BIDECLE. I discourage using "WORD", as it doesn't work how you might expect. I personally use DECLE and BIDECLE for everything. Take a look at the assembler documentation for more information on these directives.

Code After Title String

In the header, we set a bit that said "Run the code after the title string." For now, we'll just go to an infinite loop after the title screen, since there's no program to go to right now:

```
MAIN    EIS                     ; Enable interrupts
here    B       here            ; Spin forever.
```

What this does is, as the comments say, enable interrupts and spin forever. The code enables interrupts so that the display itself is enabled. The STIC chip inside the Intellivision needs to be told, on a frame by frame basis, to enable the display. This happens in an interrupt service routine (ISR). The EXEC's initialization code installs a basic ISR that, among other things, keeps the display enabled. In a future tutorial, I will show how to replace that ISR with a custom one.

Assembling and Running

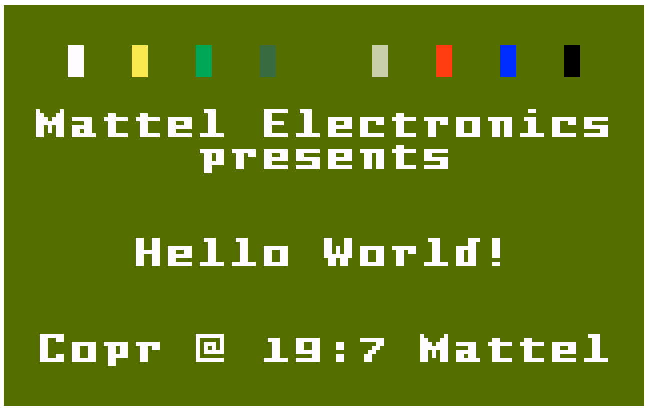
To assemble the example, copy Example 1 above into a file named "hello1.asm". Then run the following command to assemble it:

```
    as1600 -o hello1 -l hello1.lst hello1.asm
```

This will write out 4 files:

hello1.rom	The program in Intellicart .ROM format
hello1.bin	The same program in BIN+CFG format
hello1.cfg	The config file to go with hello1.bin
hello1.lst	The listing file

You can now run hello1.rom or hello1.bin in your favorite Intellivision emulator or on a real Intellivision by way of a Cuttle Cart 3 (http://www.schells.com/cc3.shtml) , Intellicart, Foom Board or any other hardware that happens to let you run on the real hardware. You should see:


```
Mattel Electronics
     presents

   Hello World!

Copr @ 19:7 Mattel
```

(Note: The screen will look different if you run this on a Sears machine, since the Sears units omit the "Mattel Electronics" string.)

Tada! Notice the broken year, and incorrect copyright attribution. We will fix that in a minute. Before we do, let's step back a minute.

Examining the Listing File

The listing file is a useful file while you're developing. It contains a detailed account of what the assembler did with your program. It shows the complete symbol table -- the list of all the symbols you used in your program and the values they ended up with -- as well as an account of what values were put in what locations in the memory space. Let's look at the listing file for hello1.asm:

```
00005000 ROMHDR                    0000500d ZERO
00005022 MAIN                      0000500f ONES
00005014 TITLE                     00005023 here
ï¿½                              ROMW    16
 0x5000                          ORG     $5000

                        ;------------------------------------------------------------------------
                        ; EXEC-friendly ROM header.
                        ;------------------------------------------------------------------------
5000 000d 0050          ROMHDR: BIDECLE ZERO            ; MOB picture base   (points to NULL list)
5002 000d 0050                  BIDECLE ZERO            ; Process table      (points to NULL list)
5004 0022 0050                  BIDECLE MAIN            ; Program start address
5006 000d 0050                  BIDECLE ZERO            ; Bkgnd picture base (points to NULL list)
5008 000f 0050                  BIDECLE ONES            ; GRAM pictures      (points to NULL list)
500a 0014 0050                  BIDECLE TITLE           ; Cartridge title/date
500c 03c0                       DECLE   $03C0           ; Flags:  No ECS title, run code after title,
                                                        ; ... no clicks
500d 0000               ZERO:   DECLE   $0000           ; Screen border control
500e 0000                       DECLE   $0000           ; 0 = color stack, 1 = f/b mode
500f 0001 0001 0001     ONES:   DECLE   1, 1, 1, 1, 1   ; Color stack initialization
5012 0001 0001
                        ;------------------------------------------------------------------------

5014 006b 0048 0065     TITLE   DECLE   107, "Hello World!", 0
5017 006c 006c 006f 0020 0057 006f 0072 006c
501f 0064 0021 0000

5022 0002               MAIN    EIS                 ; Enable interrupts
5023 0220 0001          here    B       here        ; Spin forever.
 ERROR SUMMARY - ERRORS DETECTED 0
               -  WARNINGS       0
```

The first part of the file is the symbol table. This is a list of all symbols (aka. labels) defined in the file. Symbols are simply character strings that refer to numbers. In this case, they are all program labels, and they refer to locations in memory:

`ROMHDR`	Refers to the start of the ROM header
`ZERO`	Refers to the "DECLE $0000" directive in the header
`ONES`	Refers to the "DECLE 1, 1, 1, 1, 1" directive in the header
`TITLE`	Refers to our date/title string
`MAIN`	Refers to the location of our first program instruction
`here`	Refers to the address of the infinite loop

When you write something like "`MAIN EIS`", the assembler binds the address of the instruction to the label. In this case, it binds "MAIN" to the address of the EIS instruction. According to the symbol table, that address is $5022.

After the symbol table comes the core of the assembly listing -- an accounting of how each instruction and directive turned into words in memory. At the far left is the address. Immediately after that are the actual values that the assembler put into the memory image.

For such a short program, the listing file isn't particularly interesting. For a larger program, it can be a lifesaver. If there are assembly errors, the assembler will mark them in the listing file next to the code that caused them. If you're trying to debug your code in an emulator's debugger, the listing file will tell you the addresses that everything ended up at so you can figure out where to set your breakpoints and watchpoints. All in all, it's indispensable for serious program development.

Example 2: Fixing up the title screen

Now, let's set about patching up that title screen, shall we? Before we start, copy `hello1.asm` to `hello2.asm`. This segment extends the example from the previous segment.

Let's first fix up the copyright date. One easy way to do this is to simply overwrite the entire string with one of our own. We'll use SDK-1600's PRINT function to do that.

Calling PRINT

The SDK-1600 PRINT function accepts its arguments multiple ways. Take a look at print.asm in SDK-1600 for a complete listing. For now, we will use the "PRINT.FLS" variant, which expects the following:

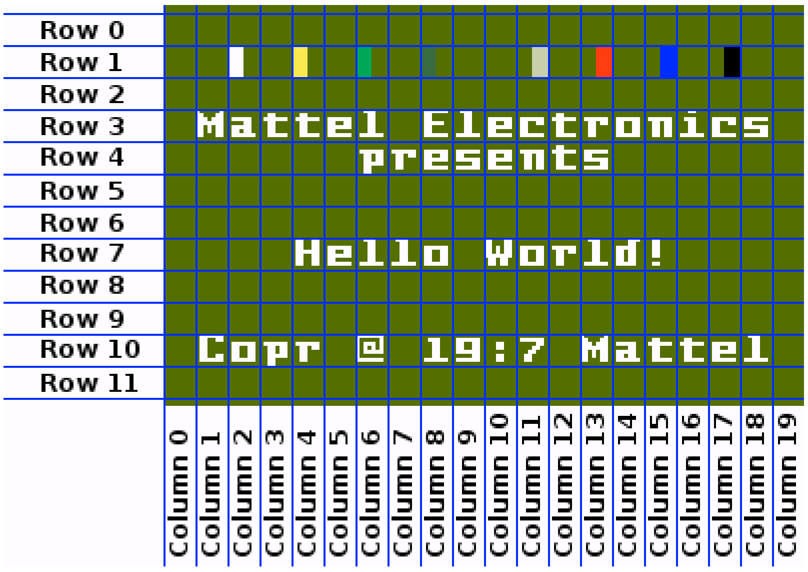
```
;;  PRINT.FLS     Format, location, and string itself follows CALL          ;;

...

;;  INPUTS for PRINT.FLS:                                                   ;;
;;      R5 -- Invocation record, followed by return code.                   ;;
;;              Screen format word          1 DECLE                         ;;
;;              Ptr to display location     1 DECLE                         ;;
;;              String                      n DECLEs (NUL terminated)       ;;
```

The CP-1610's CALL instruction works by copying the return address (the address immediately following the CALL) into the CPU register R5, and then jumping to the label or address that appears after the CALL. So "`CALL PRINT.FLS`" will jump to the label "`PRINT.FLS`". When it gets there, R5 points just after the CALL instruction.

Because of this setup, it's extremely common to pass arguments for a CALL in the words that follow the CALL. In this case, PRINT.FLS expects 3 things to appears after the CALL: A "screen format word", a pointer to where to print, and the actual NUL-terminated string to print.

The "screen format word" is a word that describes the formatting to apply to the string. Typically this is just a color number, although you can do fancier things with this. (See print.asm (http://sdk-1600.spatula-city.org/examples/library/print.asm) for details.) In our case, we'd like to print in white, and white is color #7.

The "pointer to display location" is the address on the display to which PRINT will print the string. Display memory starts at location `$200`. Location `$200` corresponds to the upper left hand corner. Each row is 20 characters long. Thus, `$200 + 19` is the rightmost character in the top row, and `$200 + 20` is the leftmost character in the second row. I'll refer to the top row as row 0 and the leftmost column as column 0. This makes the math easier.



We wish to replace the "Copr @ 19:7 Mattel" string at the bottom of the screen with our own. If you look closely at the screen shot above, you'll see that it begins in row 10, column 1. The address of this location is given by the expression: `$200 + 10*20 + 1`.

The assembler will evaluate expressions for you, so you can put that entire expression in there as-is and let the assembler work it out for you. It makes the code easier to understand and easier to write. Also, in some cases, it can make otherwise very complex and tricky things easy, especially if it all gets hidden in a macro. But that's for a later tutorial.

With our format word and display pointer in hand, we can add the following code to our program between the "`EIS`" and infinite loop:

```
MAIN    EIS                     ; Enable interrupts

        CALL    PRINT.FLS
        DECLE   7               ; 7 is the color number for "white"
        DECLE   $200 + 10*20 + 1
        DECLE   "  Copyright 2007  ", 0

here    B       here            ; Spin forever.
```

So far, so good. Now how about that PRINT function? Where does that come in? Take the file "print.asm" from the "examples/library" directory and copy it into the same directory as your Hello World program. At the end of the code, add this line:

```
        INCLUDE "print.asm"
```

The Complete hello2.asm

The complete program in `hello2.asm` looks like this:

```
        ROMW    16
        ORG     $5000

;------------------------------------------------------------------------------
; EXEC-friendly ROM header.
;------------------------------------------------------------------------------
ROMHDR: BIDECLE ZERO            ; MOB picture base   (points to NULL list)
        BIDECLE ZERO            ; Process table      (points to NULL list)
        BIDECLE MAIN            ; Program start address
        BIDECLE ZERO            ; Bkgnd picture base (points to NULL list)
        BIDECLE ONES            ; GRAM pictures      (points to NULL list)
        BIDECLE TITLE           ; Cartridge title/date
        DECLE   $03C0           ; Flags:  No ECS title, run code after title,
                                ; ... no clicks
ZERO:   DECLE   $0000           ; Screen border control
        DECLE   $0000           ; 0 = color stack, 1 = f/b mode
ONES:   DECLE   1, 1, 1, 1, 1   ; Color stack initialization
;------------------------------------------------------------------------------

TITLE   DECLE   107, "Hello World!", 0

MAIN    EIS                     ; Enable interrupts

        CALL    PRINT.FLS
        DECLE   7               ; 7 is the color number for "white"
        DECLE   $200 + 10*20 + 1
        DECLE   "  Copyright 2007  ", 0

here    B       here            ; Spin forever.


        INCLUDE "print.asm"
```

A note on INCLUDE

If you're accustomed to programming in a language like C or C++, you are now probably wondering "Why do I put the INCLUDE line at the end rather than the beginning?" That's a great question!

The INCLUDE directive copies in the contents of the specified file at the exact location it appears in your source file. It's as if you cut and pasted that file in at that point. The file "print.asm" contains the actual assembly code for the PRINT function. That means that the code gets assembled *right there*.

In a C or C++ program, the #include directive works similarly in that it copies in the requested file. The difference is that C and C++ header files typically only contain declaration information. They don't actually contain the program code that implements the functions you'll call. For example, when you do a "`#include <stdio.h>`", the file stdio.h describes what printf() looks like, but doesn't actually have an implementation of printf() in there. That comes from a library somewhere. That library gets linked in later.

In an as1600 program, the entire program gets presented to the assembler at one time. There is no separate linking step. When you include a library function such as PRINT, you're actually including the program code right there. You can see this in the listing file after you assemble your program.

If you had `INCLUDE`d `print.asm` at the top of the program, before the ORG directive, the PRINT function would get assembled at location $0000 -- not a good thing, since that's where the STIC's registers live. If you included it after the ORG directive but before the ROM header, then it would get assembled at location $5000, which is where th ROM header needs to be. You *could* put the INCLUDE directive after the header and before "TITLE". Often, though, it's just easiest to put it at the end.

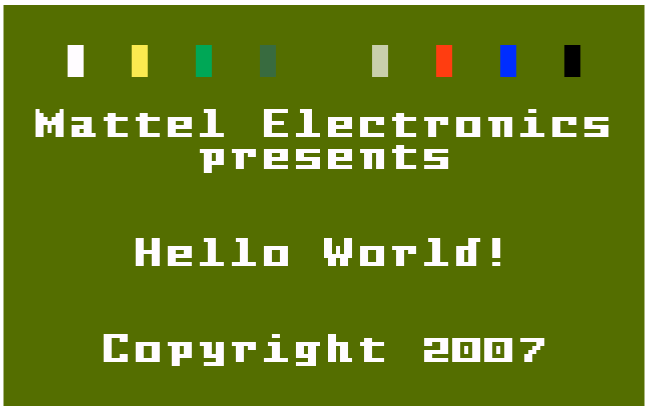
Note that some header files, such as files containing MACRO definitions and other useful definitions need to be INCLUDEd before they can be used. These files you'll want to INCLUDE at the top of your source. So far, we haven't used any of these features yet, but I thought I'd mention it.

Assembling and Running

Assemble hello2.asm with the following command:

```
as1600 -o hello2 -l hello2.lst hello2.asm
```

Then, run hello2 as you did earlier. You should see a screen similar to the following:


```
Mattel Electronics
     presents

   Hello World!

  Copyright 2007
```

Ta da!

Next Steps

See if you can take what you've learned and modify the title screen even more. For instance, try replacing "Mattel Electronics" with something else. Good luck!

Introducing jzIntv's Debugger

jzIntv offers a simple command-line oriented debugger. It should be familiar in style to anyone who has used the Apple] [Monitor or DOS's DEBUG.EXE.

Contents

Invoking the Debugger

To invoke the debugger, add the "-d" flag to jzIntv's command line. For example, using "hello2.rom" from the Hello World Tutorial:

```
    jzintv -d hello2.rom
```

This will invoke jzIntv, and present you with a prompt:

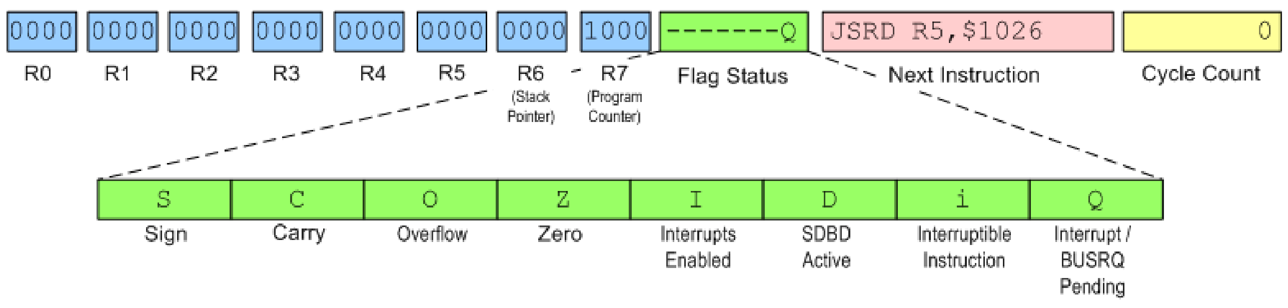
```
Loading:
  hello2.rom
jzintv:  Initializing Master Component and peripherals...
gfx:  Searching for video modes near 320x200x8 with:
gfx:      Hardware surf, Double buf, Sync blit, Software pal, Windowed
gfx:  Selected:  320x200x8 with:
gfx:      Software surf, Single buf, Sync blit, Hardware pal, Windowed
snd:  buf_size: wanted 2048, got 2048
ay8910:  Automatic sliding-window setting: 10
CP-1610          [0x0000...0x0000]
PSG0 AY8914      [0x01F0...0x01FF]
[Graphics]       [0x0000...0x0000]
[Sound]          [0x0000...0x0000]
Scratch RAM      [0x0100...0x01EF]
System RAM       [0x0200...0x035F]
EXEC ROM         [0x1000...0x1FFF]
Pad Pair 0       [0x01F0...0x01FF]
STIC             [0x0000...0x007F]
STIC             [0x4000...0x403F]
STIC             [0x8000...0x803F]
STIC             [0xC000...0xC03F]
STIC (BTAB)      [0x0200...0x02EF]
STIC (GRAM)      [0x3000...0x3FFF]
[Event]          [0x0000...0x0000]
[Rate Ctrl]      [0x0000...0x0000]
ICart   [R   ]   [0x5000...0x50FF]
CP-1610 Snoop    [0x0200...0x035F]
[Debugger]       [0x0000...0xFFFF]
 0000 0000 0000 0000 0000 0000 0000 1000 -------Q  JSRD R5,$1026            0
>
```

Most of this output is jzIntv's initialization. The last portion is the debugger prompt.

The Debugger Prompt

```
 0000 0000 0000 0000 0000 0000 0000 1000 -------Q  JSRD R5,$1026          0
>
```

The prompt shown above is the debugger's input prompt. From here, you can tell the debugger what to do next. Before each prompt, jzIntv reports specific information about the state of the machine. The following diagram illustrates:



R0 through R7 are the CPU's 8 registers. Each register is 16 bits wide. jzIntv's debugger shows their values in hexadecimal.

The "flags" field shows what CPU flags are currently set. jzIntv tracks 8 separate flags:

S	Sign Flag
C	Carry Flag
O	Overflow Flag
Z	Zero Flag
I	Interrupt Enable Flag
D	Double Byte Data Flag
i	Previous instruction was interruptible

When a flag is set, jzIntv shows the letter for that flag. When the flag is clear, jzIntv shows a dash. This makes it easy to see what flags are currently set, without having to remember their exact order. The last flag position is special. It shows the current interrupt status:

q	Interrupt asserted
b	BUSRQ asserted
Q	Interrupt being taken
B	CPU halted by BUSRQ

In the example above, jzIntv shows the CPU as taking an interrupt. Really, it's coming out of reset, which is similar. Don't worry too much about interrupts for the moment.

Introducing the Registers

The CPU's registers act as a scratch pad, holding values for instructions to operate on. Some registers have special purposes. All the registers can be used for general purpose computation. Here's a quick reference to what each register can be used for.

Register	General Purpose	Shift Instructions	Indirect Pointer	Return Address
R0	X	X		
R1	X	X	X	
R2	X	X	X	
R3	X	X	X	
R4	X		Auto-increment	X
R5	X		Auto-increment	X
R6	X		Stack	X
R7	*		Program Counter	

R6 and R7 are special. R6 is the stack pointer. R7 is the program counter. The assembler accepts SP and PC as aliases for R6 and R7. You can perform arbitrary arithmetic on either, although performing math on the program counter usually is a bad idea unless you really know what you're doing.

You will want to pay attention to R7 to know where you are at in your program. You can use the listing file, as described in the Hello World tutorial to relate what you wrote to what the debugger shows you.

Debugger Commands

The debugger offers a series of single-letter commands. You can find a full summary here (http://spatula-city.org/~im14u2c/intv/debugger.txt) . For this tutorial, we will focus on a small subset of these.

Command	Description
`R <#>`	**R**un for <#> cycles. If no argument given, runs "forever"
`S <#>`	**S**tep for <#> cycles. If no argument given, steps "forever"
`B <#>`	set **B**reakpoint at location <#>
`M <A> <B>`	show **M**emory at location <A>. Show at least <B> locations.
`U <A> <B>`	**U**nassemble memory at location <A>. Show at least <B> instrs.
`Q`	**Quit** jzIntv

The commands are case-insensitive. That is, "R 100" (run 100 instructions) is the same as "r 100". The debugger also offers a short-cut: Pressing enter alone on a line is the same as "s 1". That is, it steps a single instruction.

First Steps: Watching Things Happen

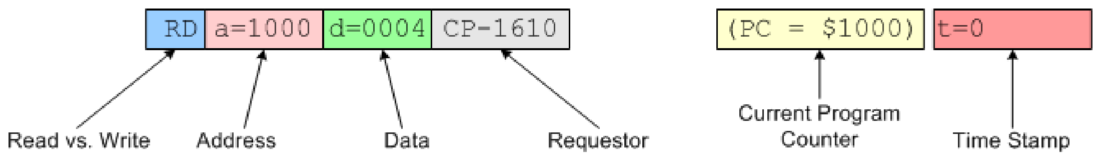
Start the debugger using "hello2.rom" from the Hello World Tutorial's Example 2:

```
jzintv -d hello2.rom
```

Once at the debugger prompt, press `[Enter]` a couple of times. You should see output similar to the following:

```
 0000 0000 0000 0000 0000 0000 0000 1000 --------  JSRD R5,$1026            0
>
Starting jzIntv...
 RD a=1000 d=0004 CP-1610          (PC = $1000) t=0
 RD a=1001 d=0112 CP-1610          (PC = $1000) t=0
 RD a=1002 d=0026 CP-1610          (PC = $1000) t=0
 0000 0000 0000 0000 0000 1003 0000 1026 --------  MVII #$02f1,R6          12
>
 RD a=1026 d=02BE CP-1610          (PC = $1026) t=12
 RD a=1027 d=02F1 CP-1610          (PC = $1026) t=12
 0000 0000 0000 0000 0000 1003 02f1 1028 ------i-  JSR  R5,$1a83           21
>
 RD a=1028 d=0004 CP-1610          (PC = $1028) t=21
 RD a=1029 d=0118 CP-1610          (PC = $1028) t=21
 RD a=102A d=0283 CP-1610          (PC = $1028) t=21
 0000 0000 0000 0000 0000 102B 02f1 1A83 ------i-  PSHR R5                 33
>
 RD a=1A83 d=0275 CP-1610          (PC = $1A83) t=33
 WR a=02F1 d=102B CP-1610          (PC = $1A83) t=33
 0000 0000 0000 0000 0000 102B 02f2 1A84 --------  PSHR R0                 42
>
```

As you can see, jzIntv shows you the current status and the next instruction it will execute at each prompt. As each instruction executes, it also outputs memory transactions as they go by. This allows you to watch what the CPU is reading or writing. The following diagram illustrates how to interpret each of these lines.



These lines show you every read and write the CPU makes. The debugger normally suppresses these unless you step through code as we have above.

In this particular code sequence, we can see the EXEC begin to initialize itself. First, it jumps from the start of ROM to the first real bit of code. Then it sets up the stack pointer, and jumps to yet another subroutine. Notice how the register values, particularly the program counter (R7), stack pointer (R6) and return address (R5) change values. Click on the instructions themselves to get a description of what each one does.

Running Ahead to Our Code

At this point, it would be useful to jump ahead to our own program code. The simulated Intellivision needs to run the EXEC code that leads up our program, but we want the simulation to stop at the first instruction of our code. We accomplish this with a breakpoint.

Take a look at `hello2.lst` from the Hello World's example 2. Below is just the first portion:

```
00005000 ROMHDR                     0000500d ZERO
00005022 MAIN                       0000500f ONES
00005014 TITLE                      0000503d PRINT.FLS
0000503b here                       0000503d PRINT
0000503e PRINT.LS                   00005048 PRINT.S
00005041 PRINT.FLP                  00005042 PRINT.LP
00005043 PRINT.P                    00005044 PRINT.R
00005053 PRINT.1st                  0000504d PRINT.tloop
ï¿½                             ROMW    16
 0x5000                         ORG     $5000

                        ;------------------------------------------------------------------------------
                        ; EXEC-friendly ROM header.
                        ;------------------------------------------------------------------------------
5000 000d 0050          ROMHDR: BIDECLE ZERO            ; MOB picture base   (points to NULL list)
5002 000d 0050                  BIDECLE ZERO            ; Process table      (points to NULL list)
5004 0022 0050                  BIDECLE MAIN            ; Program start address
5006 000d 0050                  BIDECLE ZERO            ; Bkgnd picture base (points to NULL list)
5008 000f 0050                  BIDECLE ONES            ; GRAM pictures      (points to NULL list)
500a 0014 0050                  BIDECLE TITLE           ; Cartridge title/date
500c 03c0                       DECLE   $03C0           ; Flags:  No ECS title, run code after title,
                                                        ; ... no clicks
500d 0000               ZERO:   DECLE   $0000           ; Screen border control
500e 0000                       DECLE   $0000           ; 0 = color stack, 1 = f/b mode
500f 0001 0001 0001     ONES:   DECLE   1, 1, 1, 1, 1   ; Color stack initialization
5012 0001 0001
                        ;------------------------------------------------------------------------------

5014 006b 0048 0065     TITLE   DECLE   107, "Hello World!", 0
5017 006c 006c 006f 0020 0057 006f 0072 006c
501f 0064 0021 0000

5022 0002               MAIN    EIS                     ; Enable interrupts

5023 0004 0150 003d             CALL    PRINT.FLS
5026 0007                       DECLE   7               ; 7 is the color number for "white"
5027 02c9                       DECLE   $200 + 10*20 + 1
5028 0020 0020 0043             DECLE   "  Copyright 2007  ", 0
502b 006f 0070 0079 0072 0069 0067 0068 0074
5033 0020 0032 0030 0030 0037 0020 0020 0000

503b 0220 0001          here    B       here            ; Spin forever.
```

From this, you can see that the first instruction (the "EIS" instruction) is at location $5022. Use the "B" command to set a breakpoint here, and the "R" command to tell jzintv to run. jzIntv will stop when it reaches the breakpoint. The boldface portions below illustrate the commands you should type:

```
 0000 0000 0000 0000 0000 102B 02f2 1A84 --------  PSHR R0                  42
> b 5022
Set breakpoint at $5022
> r
Hit breakpoint at $5022
 0000 C0C0 0290 8007 5022 1E87 02f2 5022 ------i-  EIS                   54475
>
```

Pulling It Apart: Disassembly and Dumping

Now we can look ahead at the next couple of instructions that will execute. The "U" command will **U**nassemble the next several instructions. "U" by itself starts at the current instruction and outputs the next several:

```
> u
    $5022:    0002                   EIS
    $5023:    0004 0150 003d         JSR  R5,$503d
    $5026:    0007                   SETC
    $5027:    02c9                   ADD@ R1,R1
    $5028:    0020                   NEGR R0
    $5029:    0020                   NEGR R0
    $502A:    0043                   SWAP R3
    $502B:    006f                   SAR  R3,2
    $502C:    0070                   RRC  R0
    $502D:    0079                   SARC R1
    $502E:    0072                   RRC  R2
    $502F:    0069                   SAR  R1
    $5030:    0067                   SLR  R3,2
    $5031:    0068                   SAR  R0
    $5032:    0074                   RRC  R0,2
    $5033:    0020                   NEGR R0
    $5034:    0032                   GSWD R2
>
```

You can clearly see our EIS instruction. Looking back at the listing, the next instruction should be a CALL. The disassembly shows "JSR R5". This is correct: CALL is an alias for JSR R5. What about the rest of this?

If you look in Example 2's source code, you'll see that the CALL was followed by some data:

```
        CALL    PRINT.FLS
        DECLE   7               ; 7 is the color number for "white"
        DECLE   $200 + 10*20 + 1
        DECLE   "  Copyright 2007  ", 0
```

jzIntv's debugger doesn't know that there is data after the CALL instruction, so it assumes it's code when it disassembles it. This gives amusing results such as we saw above. We can verify, however, that the data is what we expect it to be by using the "M" command to display a memory dump.

```
> m5022
5020:  0021 0000 0002 0004  0150 003D 0007 02C9   # .........P......
5028:  0020 0020 0043 006F  0070 0079 0072 0069   # .....C.o.p.y.r.i
5030:  0067 0068 0074 0020  0032 0030 0030 0037   # .g.h.t...2.0.0.7
5038:  0020 0020 0000 0220  0001 02A9 02AC 0200   # ..............Â¼..
5040:  0007 02A9 02AC 02A8  0275 0085 0007 000F   # .....Â¼.Â¿.u......
5048:  0006 0049 0071 0200  0006 0338 0020 004C   # ...I.q.....8...L
5050:  0048 00C8 0260 02A8  0080 022C 0009 0059   # .H.â•š...Â¿.......Y
5058:  0028 0061 00C7 00AF  02B7 0000 0000 0000   # ...a.â•Ÿ.Â».â•−......
5060:  0000 0000 0000 0000  0000 0000 0000 0000   # ................
>
```

The format of the memory dump is simple. At the left is the starting address. In the middle are hexadecimal values for 8 locations. At the right is an ASCII representation of the data. As you can see, starting on the second row at location $5028, the phrase " Copyright 2007 " is right where we expect it. The extra '.' characters between letters are due to the fact that memory locations are 16-bits wide, but characters are only 8 bits. Additional information could be in those other bits.

Stepping Through

Now let's see what the whole program does. We'll first set a breakpoint at the last instruction, and then tell jzIntv to step to the end. I've included some light commentary. Later tutorials will give more in-depth explanations of what's going on.

```
> b503b
Set breakpoint at $503B
> s
 RD a=5022 d=0002 CP-1610          (PC = $5022) t=54475
 0000 C0C0 0290 8007 5022 1E87 02f2 5023 ----I---  JSR  R5,$503d         54479
 RD a=5023 d=0004 CP-1610          (PC = $5023) t=54479
 RD a=5024 d=0150 CP-1610          (PC = $5023) t=54479
 RD a=5025 d=003D CP-1610          (PC = $5023) t=54479
```

That calls out to the PRINT function:

```
 0000 C0C0 0290 8007 5022 5026 02f2 503D ----I-i-  MVI@ R5,R1            54491
 RD a=503D d=02A9 CP-1610          (PC = $503D) t=54491
 RD a=5026 d=0007 CP-1610          (PC = $503D) t=54491
 0000 0007 0290 8007 5022 5027 02f2 503E ----I-i-  MVI@ R5,R4            54499
 RD a=503E d=02AC CP-1610          (PC = $503E) t=54499
 RD a=5027 d=02C9 CP-1610          (PC = $503E) t=54499
```

Read format word and display pointer.

```
0000 0007 0290 8007 02C9 5028 02f2 503F ----I-i-  B    $5048            54507
RD a=503F d=0200 CP-1610          (PC = $503F) t=54507
RD a=5040 d=0007 CP-1610          (PC = $503F) t=54507
0000 0007 0290 8007 02C9 5028 02f2 5048 ----I-i-  CLRC                  54516
RD a=5048 d=0006 CP-1610          (PC = $5048) t=54516
0000 0007 0290 8007 02C9 5028 02f2 5049 ----I---  SLL  R1               54520
RD a=5049 d=0049 CP-1610          (PC = $5049) t=54520
0000 000E 0290 8007 02C9 5028 02f2 504A ----I---  RRC  R1               54526
RD a=504A d=0071 CP-1610          (PC = $504A) t=54526
```

Store a flag in bit 15 of format word. See print.asm for details of why.

```
0000 0007 0290 8007 02C9 5028 02f2 504B ----I---  B    $5053            54532
RD a=504B d=0200 CP-1610          (PC = $504B) t=54532
RD a=504C d=0006 CP-1610          (PC = $504B) t=54532
```

Jump to middle of loop on first iteration.

```
0000 0007 0290 8007 02C9 5028 02f2 5053 ----I-i-  MVI@ R5,R0            54541
RD a=5053 d=02A8 CP-1610          (PC = $5053) t=54541
RD a=5028 d=0020 CP-1610          (PC = $5053) t=54541
0020 0007 0290 8007 02C9 5029 02f2 5054 ----I-i-  TSTR R0               54549
RD a=5054 d=0080 CP-1610          (PC = $5054) t=54549
0020 0007 0290 8007 02C9 5029 02f2 5055 ----I-i-  BNEQ $504d            54555
RD a=5055 d=022C CP-1610          (PC = $5055) t=54555
RD a=5056 d=0009 CP-1610          (PC = $5055) t=54555
0020 0007 0290 8007 02C9 5029 02f2 504D ----I-i-  SUBI #$0020,R0        54564
RD a=504D d=0338 CP-1610          (PC = $504D) t=54564
RD a=504E d=0020 CP-1610          (PC = $504D) t=54564
0000 0007 0290 8007 02C9 5029 02f2 504F -C-ZI-i-  SLL  R0,2             54572
RD a=504F d=004C CP-1610          (PC = $504F) t=54572
0000 0007 0290 8007 02C9 5029 02f2 5050 -C-ZI---  SLL  R0               54580
RD a=5050 d=0048 CP-1610          (PC = $5050) t=54580
0000 0007 0290 8007 02C9 5029 02f2 5051 -C-ZI---  ADDR R1,R0            54586
RD a=5051 d=00C8 CP-1610          (PC = $5051) t=54586
0007 0007 0290 8007 02C9 5029 02f2 5052 ----I-i-  MVO@ R0,R4            54592
RD a=5052 d=0260 CP-1610          (PC = $5052) t=54592
WR a=02C9 d=0007 CP-1610          (PC = $5052) t=54592
```

The above perform the following steps for the first character of " Copyright 2007 ":

- Read a character
- Is it NUL? No: Loop and display it
- Subtract off 32 to map ASCII to GROM characters
- Shift the result left by 3 to align it as the STIC expects
- Add in the format word
- Write the formatted character to the display at $2C9

Note that I will explain this process in greater detail in another tutorial.

```
0007 0007 0290 8007 02CA 5029 02f2 5053 ----I---  MVI@ R5,R0            54601
RD a=5029 d=0020 CP-1610          (PC = $5053) t=54601
0020 0007 0290 8007 02CA 502A 02f2 5054 ----I-i-  TSTR R0               54609
0020 0007 0290 8007 02CA 502A 02f2 5055 ----I-i-  BNEQ $504d            54615
0020 0007 0290 8007 02CA 502A 02f2 504D ----I-i-  SUBI #$0020,R0        54624
0000 0007 0290 8007 02CA 502A 02f2 504F -C-ZI-i-  SLL  R0,2             54632
0000 0007 0290 8007 02CA 502A 02f2 5050 -C-ZI---  SLL  R0               54640
0000 0007 0290 8007 02CA 502A 02f2 5051 -C-ZI---  ADDR R1,R0            54646
0007 0007 0290 8007 02CA 502A 02f2 5052 ----I-i-  MVO@ R0,R4            54652
WR a=02CA d=0007 CP-1610          (PC = $5052) t=54652
```

The same as above, except for the second character. Notice one thing: The RD's that correspond to program fetches disappeared. jzIntv caches program fetches, which is why these don't show up after the first time typically. This speeds up the emulation.

```
0007 0007 0290 8007 02CB 502A 02f2 5053 ----I---  MVI@ R5,R0            54661
RD a=502A d=0043 CP-1610          (PC = $5053) t=54661
0043 0007 0290 8007 02CB 502B 02f2 5054 ----I-i-  TSTR R0               54669
0043 0007 0290 8007 02CB 502B 02f2 5055 ----I-i-  BNEQ $504d            54675
0043 0007 0290 8007 02CB 502B 02f2 504D ----I-i-  SUBI #$0020,R0        54684
0023 0007 0290 8007 02CB 502B 02f2 504F -C--I-i-  SLL  R0,2             54692
008C 0007 0290 8007 02CB 502B 02f2 5050 -C--I---  SLL  R0               54700
0118 0007 0290 8007 02CB 502B 02f2 5051 -C--I---  ADDR R1,R0            54706
011F 0007 0290 8007 02CB 502B 02f2 5052 ----I-i-  MVO@ R0,R4            54712
```

```
WR a=02CB d=011F CP-1610          (PC = $5052) t=54712
011F 0007 0290 8007 02CC 502B 02f2 5053 ----I---  MVI@ R5,R0            54721
RD a=502B d=006F CP-1610          (PC = $5053) t=54721
006F 0007 0290 8007 02CC 502C 02f2 5054 ----I-i-  TSTR R0               54729
006F 0007 0290 8007 02CC 502C 02f2 5055 ----I-i-  BNEQ $504d            54735
006F 0007 0290 8007 02CC 502C 02f2 504D ----I-i-  SUBI #$0020,R0        54744
004F 0007 0290 8007 02CC 502C 02f2 504F -C--I-i-  SLL  R0,2             54752
013C 0007 0290 8007 02CC 502C 02f2 5050 -C--I---  SLL  R0               54760
0278 0007 0290 8007 02CC 502C 02f2 5051 -C--I---  ADDR R1,R0            54766
027F 0007 0290 8007 02CC 502C 02f2 5052 ----I-i-  MVO@ R0,R4            54772
WR a=02CC d=027F CP-1610          (PC = $5052) t=54772
027F 0007 0290 8007 02CD 502C 02f2 5053 ----I---  MVI@ R5,R0            54781
RD a=502C d=0070 CP-1610          (PC = $5053) t=54781
0070 0007 0290 8007 02CD 502D 02f2 5054 ----I-i-  TSTR R0               54789
0070 0007 0290 8007 02CD 502D 02f2 5055 ----I-i-  BNEQ $504d            54795
0070 0007 0290 8007 02CD 502D 02f2 504D ----I-i-  SUBI #$0020,R0        54804
0050 0007 0290 8007 02CD 502D 02f2 504F -C--I-i-  SLL  R0,2             54812
0140 0007 0290 8007 02CD 502D 02f2 5050 -C--I---  SLL  R0               54820
0280 0007 0290 8007 02CD 502D 02f2 5051 -C--I---  ADDR R1,R0            54826
0287 0007 0290 8007 02CD 502D 02f2 5052 ----I-i-  MVO@ R0,R4            54832
WR a=02CD d=0287 CP-1610          (PC = $5052) t=54832
0287 0007 0290 8007 02CE 502D 02f2 5053 ----I---  MVI@ R5,R0            54841
RD a=502D d=0079 CP-1610          (PC = $5053) t=54841
0079 0007 0290 8007 02CE 502E 02f2 5054 ----I-i-  TSTR R0               54849
0079 0007 0290 8007 02CE 502E 02f2 5055 ----I-i-  BNEQ $504d            54855
0079 0007 0290 8007 02CE 502E 02f2 504D ----I-i-  SUBI #$0020,R0        54864
0059 0007 0290 8007 02CE 502E 02f2 504F -C--I-i-  SLL  R0,2             54872
0164 0007 0290 8007 02CE 502E 02f2 5050 -C--I---  SLL  R0               54880
02C8 0007 0290 8007 02CE 502E 02f2 5051 -C--I---  ADDR R1,R0            54886
02CF 0007 0290 8007 02CE 502E 02f2 5052 ----I-i-  MVO@ R0,R4            54892
WR a=02CE d=02CF CP-1610          (PC = $5052) t=54892
02CF 0007 0290 8007 02CF 502E 02f2 5053 ----I---  MVI@ R5,R0            54901
RD a=502E d=0072 CP-1610          (PC = $5053) t=54901
0072 0007 0290 8007 02CF 502F 02f2 5054 ----I-i-  TSTR R0               54909
0072 0007 0290 8007 02CF 502F 02f2 5055 ----I-i-  BNEQ $504d            54915
0072 0007 0290 8007 02CF 502F 02f2 504D ----I-i-  SUBI #$0020,R0        54924
0052 0007 0290 8007 02CF 502F 02f2 504F -C--I-i-  SLL  R0,2             54932
0148 0007 0290 8007 02CF 502F 02f2 5050 -C--I---  SLL  R0               54940
0290 0007 0290 8007 02CF 502F 02f2 5051 -C--I---  ADDR R1,R0            54946
0297 0007 0290 8007 02CF 502F 02f2 5052 ----I-i-  MVO@ R0,R4            54952
WR a=02CF d=0297 CP-1610          (PC = $5052) t=54952
0297 0007 0290 8007 02D0 502F 02f2 5053 ----I---  MVI@ R5,R0            54961
RD a=502F d=0069 CP-1610          (PC = $5053) t=54961
0069 0007 0290 8007 02D0 5030 02f2 5054 ----I-i-  TSTR R0               54969
0069 0007 0290 8007 02D0 5030 02f2 5055 ----I-i-  BNEQ $504d            54975
0069 0007 0290 8007 02D0 5030 02f2 504D ----I-i-  SUBI #$0020,R0        54984
0049 0007 0290 8007 02D0 5030 02f2 504F -C--I-i-  SLL  R0,2             54992
0124 0007 0290 8007 02D0 5030 02f2 5050 -C--I---  SLL  R0               55000
0248 0007 0290 8007 02D0 5030 02f2 5051 -C--I---  ADDR R1,R0            55006
024F 0007 0290 8007 02D0 5030 02f2 5052 ----I-i-  MVO@ R0,R4            55012
WR a=02D0 d=024F CP-1610          (PC = $5052) t=55012
024F 0007 0290 8007 02D1 5030 02f2 5053 ----I---  MVI@ R5,R0            55021
RD a=5030 d=0067 CP-1610          (PC = $5053) t=55021
0067 0007 0290 8007 02D1 5031 02f2 5054 ----I-i-  TSTR R0               55029
0067 0007 0290 8007 02D1 5031 02f2 5055 ----I-i-  BNEQ $504d            55035
0067 0007 0290 8007 02D1 5031 02f2 504D ----I-i-  SUBI #$0020,R0        55044
0047 0007 0290 8007 02D1 5031 02f2 504F -C--I-i-  SLL  R0,2             55052
011C 0007 0290 8007 02D1 5031 02f2 5050 -C--I---  SLL  R0               55060
0238 0007 0290 8007 02D1 5031 02f2 5051 -C--I---  ADDR R1,R0            55066
023F 0007 0290 8007 02D1 5031 02f2 5052 ----I-i-  MVO@ R0,R4            55072
WR a=02D1 d=023F CP-1610          (PC = $5052) t=55072
023F 0007 0290 8007 02D2 5031 02f2 5053 ----I---  MVI@ R5,R0            55081
RD a=5031 d=0068 CP-1610          (PC = $5053) t=55081
0068 0007 0290 8007 02D2 5032 02f2 5054 ----I-i-  TSTR R0               55089
0068 0007 0290 8007 02D2 5032 02f2 5055 ----I-i-  BNEQ $504d            55095
0068 0007 0290 8007 02D2 5032 02f2 504D ----I-i-  SUBI #$0020,R0        55104
0048 0007 0290 8007 02D2 5032 02f2 504F -C--I-i-  SLL  R0,2             55112
0120 0007 0290 8007 02D2 5032 02f2 5050 -C--I---  SLL  R0               55120
0240 0007 0290 8007 02D2 5032 02f2 5051 -C--I---  ADDR R1,R0            55126
0247 0007 0290 8007 02D2 5032 02f2 5052 ----I-i-  MVO@ R0,R4            55132
WR a=02D2 d=0247 CP-1610          (PC = $5052) t=55132
0247 0007 0290 8007 02D3 5032 02f2 5053 ----I---  MVI@ R5,R0            55141
RD a=5032 d=0074 CP-1610          (PC = $5053) t=55141
0074 0007 0290 8007 02D3 5033 02f2 5054 ----I-i-  TSTR R0               55149
0074 0007 0290 8007 02D3 5033 02f2 5055 ----I-i-  BNEQ $504d            55155
0074 0007 0290 8007 02D3 5033 02f2 504D ----I-i-  SUBI #$0020,R0        55164
0054 0007 0290 8007 02D3 5033 02f2 504F -C--I-i-  SLL  R0,2             55172
0150 0007 0290 8007 02D3 5033 02f2 5050 -C--I---  SLL  R0               55180
02A0 0007 0290 8007 02D3 5033 02f2 5051 -C--I---  ADDR R1,R0            55186
02A7 0007 0290 8007 02D3 5033 02f2 5052 ----I-i-  MVO@ R0,R4            55192
WR a=02D3 d=02A7 CP-1610          (PC = $5052) t=55192
```

That brings us through " Copyright"...

```
02A7 0007 0290 8007 02D4 5033 02f2 5053 ----I---  MVI@ R5,R0            55201
RD a=5033 d=0020 CP-1610          (PC = $5053) t=55201
0020 0007 0290 8007 02D4 5034 02f2 5054 ----I-i-  TSTR R0               55209
0020 0007 0290 8007 02D4 5034 02f2 5055 ----I-i-  BNEQ $504d            55215
0020 0007 0290 8007 02D4 5034 02f2 504D ----I-i-  SUBI #$0020,R0        55224
0000 0007 0290 8007 02D4 5034 02f2 504F -C-ZI-i-  SLL  R0,2             55232
0000 0007 0290 8007 02D4 5034 02f2 5050 -C-ZI---  SLL  R0               55240
0000 0007 0290 8007 02D4 5034 02f2 5051 -C-ZI---  ADDR R1,R0            55246
0007 0007 0290 8007 02D4 5034 02f2 5052 ----I-i-  MVO@ R0,R4            55252
WR a=02D4 d=0007 CP-1610          (PC = $5052) t=55252
0007 0007 0290 8007 02D5 5034 02f2 5053 ----I---  MVI@ R5,R0            55261
```

```
 RD a=5034 d=0032 CP-1610          (PC = $5053) t=55261
 0032 0007 0290 8007 02D5 5035 02f2 5054 ----I-i-  TSTR R0             55269
 0032 0007 0290 8007 02D5 5035 02f2 5055 ----I-i-  BNEQ $504d           55275
 0032 0007 0290 8007 02D5 5035 02f2 504D ----I-i-  SUBI #$0020,R0       55284
 0012 0007 0290 8007 02D5 5035 02f2 504F -C--I-i-  SLL  R0,2            55292
 0048 0007 0290 8007 02D5 5035 02f2 5050 -C--I---  SLL  R0              55300
 0090 0007 0290 8007 02D5 5035 02f2 5051 -C--I---  ADDR R1,R0           55306
 0097 0007 0290 8007 02D5 5035 02f2 5052 ----I-i-  MVO@ R0,R4           55312
 WR a=02D5 d=0097 CP-1610          (PC = $5052) t=55312
 0097 0007 0290 8007 02D6 5035 02f2 5053 ----I---  MVI@ R5,R0           55321
 RD a=5035 d=0030 CP-1610          (PC = $5053) t=55321
 0030 0007 0290 8007 02D6 5036 02f2 5054 ----I-i-  TSTR R0              55329
 0030 0007 0290 8007 02D6 5036 02f2 5055 ----I-i-  BNEQ $504d           55335
 0030 0007 0290 8007 02D6 5036 02f2 504D ----I-i-  SUBI #$0020,R0       55344
 0010 0007 0290 8007 02D6 5036 02f2 504F -C--I-i-  SLL  R0,2            55352
 0040 0007 0290 8007 02D6 5036 02f2 5050 -C--I---  SLL  R0              55360
 0080 0007 0290 8007 02D6 5036 02f2 5051 -C--I---  ADDR R1,R0           55366
 0087 0007 0290 8007 02D6 5036 02f2 5052 ----I-i-  MVO@ R0,R4           55372
 WR a=02D6 d=0087 CP-1610          (PC = $5052) t=55372
 0087 0007 0290 8007 02D7 5036 02f2 5053 ----I---  MVI@ R5,R0           55381
 RD a=5036 d=0030 CP-1610          (PC = $5053) t=55381
 0030 0007 0290 8007 02D7 5037 02f2 5054 ----I-i-  TSTR R0              55389
 0030 0007 0290 8007 02D7 5037 02f2 5055 ----I-i-  BNEQ $504d           55395
 0030 0007 0290 8007 02D7 5037 02f2 504D ----I-i-  SUBI #$0020,R0       55404
 0010 0007 0290 8007 02D7 5037 02f2 504F -C--I-i-  SLL  R0,2            55412
 0040 0007 0290 8007 02D7 5037 02f2 5050 -C--I---  SLL  R0              55420
 0080 0007 0290 8007 02D7 5037 02f2 5051 -C--I---  ADDR R1,R0           55426
 0087 0007 0290 8007 02D7 5037 02f2 5052 ----I-i-  MVO@ R0,R4           55432
 WR a=02D7 d=0087 CP-1610          (PC = $5052) t=55432
 0087 0007 0290 8007 02D8 5037 02f2 5053 ----I---  MVI@ R5,R0           55441
 RD a=5037 d=0037 CP-1610          (PC = $5053) t=55441
 0037 0007 0290 8007 02D8 5038 02f2 5054 ----I-i-  TSTR R0              55449
 0037 0007 0290 8007 02D8 5038 02f2 5055 ----I-i-  BNEQ $504d           55455
 0037 0007 0290 8007 02D8 5038 02f2 504D ----I-i-  SUBI #$0020,R0       55464
 0017 0007 0290 8007 02D8 5038 02f2 504F -C--I-i-  SLL  R0,2            55472
 005C 0007 0290 8007 02D8 5038 02f2 5050 -C--I---  SLL  R0              55480
 00B8 0007 0290 8007 02D8 5038 02f2 5051 -C--I---  ADDR R1,R0           55486
 00BF 0007 0290 8007 02D8 5038 02f2 5052 ----I-i-  MVO@ R0,R4           55492
 WR a=02D8 d=00BF CP-1610          (PC = $5052) t=55492
 00BF 0007 0290 8007 02D9 5038 02f2 5053 ----I---  MVI@ R5,R0           55501
 RD a=5038 d=0020 CP-1610          (PC = $5053) t=55501
 0020 0007 0290 8007 02D9 5039 02f2 5054 ----I-i-  TSTR R0              55509
 0020 0007 0290 8007 02D9 5039 02f2 5055 ----I-i-  BNEQ $504d           55515
 0020 0007 0290 8007 02D9 5039 02f2 504D ----I-i-  SUBI #$0020,R0       55524
 0000 0007 0290 8007 02D9 5039 02f2 504F -C-ZI-i-  SLL  R0,2            55532
 0000 0007 0290 8007 02D9 5039 02f2 5050 -C-ZI---  SLL  R0              55540
 0000 0007 0290 8007 02D9 5039 02f2 5051 -C-ZI---  ADDR R1,R0           55546
 0007 0007 0290 8007 02D9 5039 02f2 5052 ----I-i-  MVO@ R0,R4           55552
 WR a=02D9 d=0007 CP-1610          (PC = $5052) t=55552
 0007 0007 0290 8007 02DA 5039 02f2 5053 ----I---  MVI@ R5,R0           55561
 RD a=5039 d=0020 CP-1610          (PC = $5053) t=55561
 0020 0007 0290 8007 02DA 503A 02f2 5054 ----I-i-  TSTR R0              55569
 0020 0007 0290 8007 02DA 503A 02f2 5055 ----I-i-  BNEQ $504d           55575
 0020 0007 0290 8007 02DA 503A 02f2 504D ----I-i-  SUBI #$0020,R0       55584
 0000 0007 0290 8007 02DA 503A 02f2 504F -C-ZI-i-  SLL  R0,2            55592
 0000 0007 0290 8007 02DA 503A 02f2 5050 -C-ZI---  SLL  R0              55600
 0000 0007 0290 8007 02DA 503A 02f2 5051 -C-ZI---  ADDR R1,R0           55606
 0007 0007 0290 8007 02DA 503A 02f2 5052 ----I-i-  MVO@ R0,R4           55612
 WR a=02DA d=0007 CP-1610          (PC = $5052) t=55612
```

...and that finishes of the string with " 2007 ". Next is the NUL terminator:

```
 0007 0007 0290 8007 02DB 503A 02f2 5053 ----I---  MVI@ R5,R0           55621
 RD a=503A d=0000 CP-1610          (PC = $5053) t=55621
 0000 0007 0290 8007 02DB 503B 02f2 5054 ----I-i-  TSTR R0              55629
 0000 0007 0290 8007 02DB 503B 02f2 5055 ---ZI-i-  BNEQ $504d           55635
```

See how this time, the loop didn't iterate?

```
 0000 0007 0290 8007 02DB 503B 02f2 5057 ---ZI-i-  SLLC R1              55642
 RD a=5057 d=0059 CP-1610          (PC = $5057) t=55642
 0000 000E 0290 8007 02DB 503B 02f2 5058 ----I---  ADCR R0              55648
 RD a=5058 d=0028 CP-1610          (PC = $5058) t=55648
 0000 000E 0290 8007 02DB 503B 02f2 5059 ---ZI-i-  SLR  R1              55654
 RD a=5059 d=0061 CP-1610          (PC = $5059) t=55654
 0000 0007 0290 8007 02DB 503B 02f2 505A ----I---  ADDR R0,R7           55660
```

This extracts the flag I mentioned obliquely earlier. See "print.asm" for details. And for the grand finale, we return and hit our breakpoint:

```
 RD a=505A d=00C7 CP-1610          (PC = $505A) t=55660
 0000 0007 0290 8007 02DB 503B 02f2 505B ----I-i-  MOVR R5,R7           55667
 RD a=505B d=00AF CP-1610          (PC = $505B) t=55667
 0000 0007 0290 8007 02DB 503B 02f2 503B ----I-i-  B    $503b           55674
Hit breakpoint at $503B
 0000 0007 0290 8007 02DB 503B 02f2 503B ----I-i-  B    $503b           55674
>
```

Wrapping Up

That concludes the quick tour of the most useful commands in jzIntv's debugger. You can quit the debugger by giving it the "Q" command. You're encouraged at this point to try single-stepping through other programs to see how the instructions work. The next tutorial aims at introducing the various families of instructions.

Introducing the Instruction Set Part 1

This tutorial focuses primarily on introducing the CP1610's instruction set. The instructions divide up into multiple categories that we will explore briefly below. You can visit the various instruction p

ages for details on specific instructions.

Contents

The CPU, Memory and Registers

The Central Processing Unit, aka. CPU, forms the brain of the system. It is connected to several memories, including the Executive ROM, Scratchpad RAM, System RAM, Graphics ROM and Graphics RAM. It's also connected to several peripherals, such as the Programmable Sound Generator (PSG) and Standard Television Interface Controller (STIC). On the inside, it has a small amount of storage for intermediate values called registers.

Memory

All of the RAMs, ROMs and peripherals are hooked to the CPU over a common interface known as the "system bus." Each of these items is assigned a range of *addresses* by which the CPU can refer to it. For example, the Scratch RAM is assigned addresses $0100 - $01EF. The System RAM is assigned addresses $0200 - $035F. The EXEC is assigned addresses $1000 - $1FFF. And so on. These assignments are referred to as the Memory Map.

One can think of addresses as being similar to street addresses and ZIP codes and each memory or peripheral as being similar to a city—each city gets its own range of ZIP codes that's separate from any other, and the ZIP code makes sure that mail gets to the right cities. The street address says where in the city the mail gets delivered. Addresses for accesses on the system bus work similarly: Some portion of the address uniquely identifies what item is being addressed, and the rest of the address says what to access within the item.

The CPU interacts with memory by reading and writing specific address locations in memory. The CPU does not know what it is accessing when it sends out a read or write. All it knows is the address, and whether it's reading or writing. If it's writing, it also knows the data it wishes to write. How the addressed item responds depends on the item itself.

RAMs (Random Access Memories) respond to reads by returning the contents of the addressed location. For example, if the CPU reads from location $0100, the Scratch RAM will return the contents of that location to the CPU. If the CPU writes to location $0100, the Scratch RAM will update the value stored in that location. ROMs (Read Only Memories) work similarly, except that they ignore writes. Peripherals such as the PSG and STIC modify their behavior based on what gets written to them, either generating various sounds (in the case of the PSG) or changing the display (in the case of the STIC).

Programming the CPU is primarily a process of figuring out what values need to be written where and when. ROMs typically hold the program code and fixed data. RAMs hold values that change during the game, or in case of the GRAM and BACKTAB, information displayed on the screen. Peripherals handle inputs (such as controllers) and provide outputs such as sound, voice and video.

The CPU Registers

The CPU has 8 16-bit registers. These registers act as a scratch pad, holding values for instructions to operate on. Some registers have special purposes. All of the registers can be used for general purpose computation. Here's a quick reference to what each register can be used for.

Register	General Purpose	Shift Instructions	Indirect Pointer	Return Address
R0	**X**	**X**		
R1	**X**	**X**	**X**	
R2	**X**	**X**	**X**	
R3	**X**	**X**	**X**	
R4	**X**		**Auto-increment**	**X**
R5	**X**		**Auto-increment**	**X**
R6	**X**		**Stack**	**X**
R7	*		**Program Counter**	

R6 and R7 are special. R6 is the stack pointer. R7 is the program counter. The assembler accepts SP and PC as aliases for R6 and R7. You can perform arbitrary arithmetic on either, although performing math on the program counter usually is a bad idea unless you really know what you're doing.

In addition to the 8 16-bit registers, the CPU has a number of flag bits that track processor status:

S	Sign Flag	If set, indicates that the previous operation resulted in negative value, which is determined by a one (1) in bit 15 of the result.
C	Carry Flag	If set, indicates that the previous operation resulted in unsigned integer overflow.
Z	Zero Flag	If set, indicates that the previous operation resulted in zero.
O	Overflow Flag	If set, indicates that the previous operation gave a signed result that is inconsistent with the signs of the source operands. (Signed overflow.)
I	Interrupt Enable Flag	If set, allows the INTRM line to trigger an interrupt, causing the CPU to jump to the interrupt subroutine located in the Executive ROM at $1004.
D	Double Byte Data Flag	If set, it causes the next instruction to read a 16-bit operand with two 8-bit memory accesses, if it supports it.

Primary Instructions

The CP1610 has 7 primary instructions. The primary instructions have the most flexibility in terms of where they draw their inputs from. The 7 primary instructions are:

MVI	"MoVe In": Read a value into a register from memory
MVO	"MoVe Out": Write a value from a register to memory
ADD	Add two values together
SUB	Subtract two values
CMP	Compare two values by subtracting them
AND	Bitwise logical AND
XOR	Bitwise logical XOR

Each instruction takes two operands. The first operand is a source operand. The second is both source and destination, except in the case of CMP, which doesn't write a result. (CMP does, however, set flags.) The primary instructions are available in 4 forms, each with a different addressing mode. The mnemonics for each form differs slightly from the others:

Register Mode	Direct Mode	Indirect Mode	Immediate Mode
MOVR	MVI	MVI@	MVII
	MVO	MVO@	MVOI

ADDR	ADD	ADD@	ADDI
SUBR	SUB	SUB@	SUBI
CMPR	CMP	CMP@	CMPI
ANDR	AND	AND@	ANDI
XORR	XOR	XOR@	XORI

Register Mode Instructions

Register mode instructions operate on two different registers. For instance, "ADDR R0, R1" adds the value in R0 to the value in R1, and writes the result to R1. It's equivalent to the expression "R1 = R0 + R1".

The move instructions are the simplest. "Move" is something of a misnomer, though. Move instructions really copy values from one place into another. For example, "MOVR R0, R1" copies the value in R0 to R1. After the CPU runs this instruction, both R0 and R1 will have the same value. The CPU will also set the Sign Flag and Zero Flag based on the value of the number it copied.

Move instructions are sometimes used with the program counter. "MOVR R5, R7" will jump to the location whose address is in R5. The assembler offers a pseudonym for this, "JR". The instruction "JR R5" is equivalent to "MOVR R5, R7."

Examples:

```
;  Add R0 to R1, leaving the result in R1
ADDR R0, R1
```

```
;  Copy the value in R2 to R3
MOVR R2, R3
```

```
;  Subtracts R1 from R2 setting flags.  Does not change either of R1 or R2.
CMPR R1, R2
```

Direct Mode Instructions

Direct mode instructions operate on a value in memory. For all but MVO, they read a value from the named location. The instruction "ADD $123, R0" reads the value from location $123 and adds it to R0. The MVO instruction works in the opposite direction: It copies the value from a register into a memory location. "MVO R0, $123" writes the value that's in R0 to location $123.

The address can either be a bare address, as shown above, or a label. Labeled addresses are much easier to read. For example, suppose the label "LIVES" points to the number of lives your player has remaining. You could read that value from memory into R0 with:

```
MVI LIVES, R0
```

You'll see this technique used heavily in assembly programs of appreciable size. It makes programs easier to write, read and follow.

Indirect Mode Instructions

Indirect mode instructions also operate on a value in memory. However, rather than specifying the address directly in the instruction, they instead get the address from a register. This is useful for reading through a range of memory. For example, the instruction "ADD@ R1, R2" reads from the memory location pointed to by R1, and adds that value into R2. As with direct mode, the MVO@ instruction works in the opposite direction by writing a value to memory. "MVO@ R0, R1" writes the value of R0 to the location pointed to by R1.

Indirect mode behaves specially when using R4, R5 or R6 as the pointer register. For R4 and R5, the CPU will increment their value after using it. This makes it easy to step through an array in memory. The instruction "MVI@ R4, R0" will copy the value in memory pointed to by R4 into R0, and then it will increment R4.

Examples:

```
;  Read memory pointed to by R3,
;  and put the value in R0
MVI@ R3, R0
```

```
;  Read memory pointed to by R2,
;  and add the value to R1, leaving the result in R1
ADD@ R2, R1
```

```
;  Read memory pointed to by R4,
;  subtract the value from R3 writing the difference to R3.
;  Then, increment R4.
SUB@ R4, R3
```

R6 works even more specially. When reading via R6, the CPU will decrement R6 *before* using it. When writing via R6, the CPU will increment R6 *after* using it. This behavior is what makes R6 useful as a stack pointer. The instructions "PSHR" and "PULR" are just pseudonyms for MVI@ and MVO@ using R6. Thus "PSHR R0" means the same as "MVO@ R0, R6", and "PULR R0" means the same as "MVI@ R6, R0".

The stack pointer can be used with other instructions as well. The instruction "ADD@ R6, R0" reads the value from the top of stack and adds it to R0. This can eliminate many PULR instructions. If you've ever used a stack-based calculator (such as one of HP's RPN calculators), you may find this style of programming intuitive.

Indirect mode has one limitation: You can't use R0 as the indirect register. Therefore "MVI@ R0, R1" is illegal, as is "MVO@ R1, R0".

Immediate Mode Instructions

Immediate mode instructions operate on a constant value. "MVII #1234, R0" copies the number 1234 into R0. These are useful for adding or subtracting constants from registers, setting up values in registers and so on. Note that the instruction "MVOI" generally doesn't do anything useful. You can safely ignore it.

Examples:

```
  ; Puts the value 1234 into R0
  MVII #1234, R0
```

```
  ; Bitwise ANDs $FF with R1, leaving result in R1
  ANDI #$FF, R1
```

Special Note: Compare Instructions

The CMP instruction bears special mention. Regardless of their addressing mode, the ADD, SUB and CMP instructions all set the flags based on the result of the computation. ADD and SUB also write the computed value back to a register. CMP works like SUB, but it doesn't write the result of the subtraction anywhere. Rather, it only sets flags. Its primary use is to control conditional branch instructions.

Program Example: ex1.asm

Lets see some of the primary instructions in action, shall we? The following example code (`ex1.asm`) give a couple examples of each addressing mode on a couple instructions.

Program Listing

```
        ROMW    16
        ORG     $5000
;------------------------------------------------------------------------------
; EXEC-friendly ROM header.
;------------------------------------------------------------------------------
ROMHDR: BIDECLE ZERO            ; MOB picture base   (points to NULL list)
        BIDECLE ZERO            ; Process table      (points to NULL list)
        BIDECLE MAIN            ; Program start address
        BIDECLE ZERO            ; Bkgnd picture base (points to NULL list)
        BIDECLE ONES            ; GRAM pictures      (points to NULL list)
        BIDECLE TITLE           ; Cartridge title/date
        DECLE   $03C0           ; Flags:  No ECS title, run code after title,
                                ; ... no clicks
ZERO:   DECLE   $0000           ; Screen border control
        DECLE   $0000           ; 0 = color stack, 1 = f/b mode
ONES:   DECLE   1, 1, 1, 1, 1   ; Color stack initialization
;------------------------------------------------------------------------------

TITLE   DECLE   107, "Example 1", 0

MAIN
```

```
        ; Immediate mode instructions
        MVII    #$1234, R0
        ADDI    #$5555, R0

        ; Register mode instructions
        MOVR    R0,     R1
        ADDR    R0,     R1

        ; Direct mode instructions
        MVO     R0,     $300
        MVI     $300,   R2
        ADD     $300,   R2


        MVII    #$300,  R1
        MOVR    R1,     R5

        ; Indirect mode instructions
        MVI@    R1,     R3
        MVO@    R3,     R5
        MVO@    R3,     R5
        MVO@    R3,     R5

        ; Stack instructions
        PSHR    R0              ; put R0 on the stack
        PULR    R4              ; pull the value back off into R4

here    B       here            ; Spin forever.
```

Assembling

Assemble this program with "`as1600 -o ex1 -l ex1.lst ex1.asm`". Then take a quick peek at the symbol table in the listing file. (See the Hello World Tutorial if you need a refresher.)

```
00005000 ROMHDR                     0000500d ZERO
0000501f MAIN                       0000500f ONES
00005014 TITLE                      0000502f here
```

As you can see, "MAIN", which is where our program starts, is at $501F. Since this program doesn't print anything, we'll need to watch it in action in the debugger. To save time, we can set a breakpoint at the start of our code so that we can skip watching the EXEC.

Observing in the Debugger

Fire up jzIntv with "jzintv -d ex1". (See Introducing_jzIntv's_Debugger for a tutorial on the debugger.)) After jzIntv loads, you'll be at the debugger prompt:

```
  0000 0000 0000 0000 0000 0000 0000 1000 --------  JSRD R5,$1026             0
 >
```

Set a breakpoint at MAIN by typing "b 501F" and pressing enter. Then run up until the breakpoint by typing "r" and enter. The result should look like this:

```
  0000 0000 0000 0000 0000 0000 0000 1000 --------  JSRD R5,$1026             0
 > b 501F
 Set breakpoint at $501F
 > r
 Starting jzIntv...
 Hit breakpoint at $501F
  0000 C0C0 0291 8007 501F 1E87 02f2 501F ------ib  MVII #$1234,R0        54064
 >
```

As you can see, R7 = $501F, which is the first instruction of MAIN. The other registers all have values in them that were set up by the EXEC. We can ignore these values for the time being. Lets single step through our 10 instructions. (Note: I have highlighted register values that change among R0 - R6 and important memory accesses for clarity.)

```
 0000 C0C0 0291 8007 501F 1E87 02f2 501F ------i-  MVII #$1234,R0        54064
>
 RD a=501F d=02B8 CP-1610          (PC = $501F) t=54064
 RD a=5020 d=1234 CP-1610          (PC = $501F) t=54064
 1234 C0C0 0291 8007 501F 1E87 02f2 5021 ------i-  ADDI #$5555,R0        54072
>
 RD a=5021 d=02F8 CP-1610          (PC = $5021) t=54072
 RD a=5022 d=5555 CP-1610          (PC = $5021) t=54072
 6789 C0C0 0291 8007 501F 1E87 02f2 5023 ------i-
```

See how the MVII sets R0, and the ADDI changes it.

```
 6789 C0C0 0291 8007 501F 1E87 02f2 5023 ------i-  MOVR R0,R1            54080
>
 RD a=5023 d=0081 CP-1610          (PC = $5023) t=54080
 6789 6789 0291 8007 501F 1E87 02f2 5024 ------i-  ADDR R0,R1            54086
>
 RD a=5024 d=00C1 CP-1610          (PC = $5024) t=54086
 6789 CF12 0291 8007 501F 1E87 02f2 5025 S-O---i-
```

See how MOVR R0, R1 copies the value from R0 to R1, and how ADDR R0, R1 adds the value of R0 into R1, leaving the result in R1.

```
 6789 CF12 0291 8007 501F 1E87 02f2 5025 S-O---i-  MVO  R0,$0300         54092
>
 RD a=5025 d=0240 CP-1610          (PC = $5025) t=54092
 RD a=5026 d=0300 CP-1610          (PC = $5025) t=54092
 WR a=0300 d=6789 CP-1610          (PC = $5025) t=54092
 6789 CF12 0291 8007 501F 1E87 02f2 5027 S-O-----  MVI  $0300,R2         54103
>
 RD a=5027 d=0282 CP-1610          (PC = $5027) t=54103
 RD a=5028 d=0300 CP-1610          (PC = $5027) t=54103
 RD a=0300 d=6789 CP-1610          (PC = $5027) t=54103
 6789 CF12 6789 8007 501F 1E87 02f2 5029 S-O---i-  ADD  $0300,R2         54113
>
 RD a=5029 d=02C2 CP-1610          (PC = $5029) t=54113
 RD a=502A d=0300 CP-1610          (PC = $5029) t=54113
 RD a=0300 d=6789 CP-1610          (PC = $5029) t=54113
 6789 CF12 CF12 8007 501F 1E87 02f2 502B S-O---i-
```

Here, you can see the CPU write the value of R0 to location $300 with the first instruction. The next one reads location $300 into R2. The third one reads location $300, and adds its value to R2, leaving the result in R2.

The next two instructions set us up to read and write locations around $300 with indirect accesses:

```
 6789 CF12 CF12 8007 501F 1E87 02f2 502B S-O---i-  MVII #$0300,R1        54123
>
 RD a=502B d=02B9 CP-1610          (PC = $502B) t=54123
 RD a=502C d=0300 CP-1610          (PC = $502B) t=54123
```

```
  6789 0300 CF12 8007 501F 1E87 02f2 502D S-O---i-  MOVR R1,R5           54131
 >
  RD a=502D d=008D CP-1610          (PC = $502D) t=54131
  6789 0300 CF12 8007 501F 0300 02f2 502E --O---i-
```

At this point, both R1 and R5 are set up to point to location $300. The only difference between the two is that R1 does not increment with each indirect access, but R5 does.

```
  6789 0300 CF12 8007 501F 0300 02f2 502E --O---i-  MVI@ R1,R3           54137
 >
  RD a=502E d=028B CP-1610          (PC = $502E) t=54137
  RD a=0300 d=6789 CP-1610          (PC = $502E) t=54137
  6789 0300 CF12 6789 501F 0300 02f2 502F --O---i-  MVO@ R3,R5           54145
 >
  RD a=502F d=026B CP-1610          (PC = $502F) t=54145
  WR a=0300 d=6789 CP-1610          (PC = $502F) t=54145
  6789 0300 CF12 6789 501F 0301 02f2 5030 --O-----
```

Notice how both instructions accessed location $300. Also notice that R1 retained its value, but the CPU incremented R5. This is the main difference between R1-R3 and R4-R5 for indirect accesses.

```
  6789 0300 CF12 6789 501F 0301 02f2 5030 --O-----  MVO@ R3,R5           54154
 >
  RD a=5030 d=026B CP-1610          (PC = $5030) t=54154
  WR a=0301 d=6789 CP-1610          (PC = $5030) t=54154
  6789 0300 CF12 6789 501F 0302 02f2 5031 --O-----  MVO@ R3,R5           54163
 >
  RD a=5031 d=026B CP-1610          (PC = $5031) t=54163
  WR a=0302 d=6789 CP-1610          (PC = $5031) t=54163
  6789 0300 CF12 6789 501F 0303 02f2 5032 --O-----
```

As you can see, with each "MVO@ R3, R5", the CPU writes the value of R3 out to the address that R5 currently points to, and it also increments R5. So, the first write goes to $300, the next goes to $301, and the third goes to $302. At the end, R5 points to $303.

That brings us to our last two instructions, PSHR and PULR:

```
  6789 0300 CF12 6789 501F 0303 02f2 5032 --O-----  PSHR R0              54172
 >
  RD a=5032 d=0270 CP-1610          (PC = $5032) t=54172
  WR a=02F2 d=6789 CP-1610          (PC = $5032) t=54172
  6789 0300 CF12 6789 501F 0303 02f3 5033 --O-----  PULR R4              54181
 >
  RD a=5033 d=02B4 CP-1610          (PC = $5033) t=54181
  RD a=02F2 d=6789 CP-1610          (PC = $5033) t=54181
  6789 0300 CF12 6789 6789 0303 02f2 5034 --O---i-  B    $5034           54192
 >
```

The PSHR instruction writes the value of R0 ($6789) to the "top of stack", location $2F2. This is the address held in R6, the stack pointer. It also advances the stack pointer to point to location $2F3.

The PULR instruction reverses the process. First, the CPU decrements R6, and then it reads the value R6 now points to from memory into R4. Notice that we pushed from one register and pulled back into a different one. While "push" and "pull" operations need to be paired so that the stack stays consistent, nothing prevents you from pushing from one register and pulling into another.

Before you quit jzIntv, let's take a peek at memory real quick, by typing "m 2F0" and pressing enter:

```
> m 2F0
02F0:  5014 106C 6789 1E7D  8007 0002 0000 0007   # P..lg...........
02F8:  02D2 17D0 18F9 0000  8007 3800 5012 0228   # .-.-......8.P...
0300:  6789 6789 6789 0000  0000 0000 0000 0000   # g.g.g...........
0308:  0000 0000 0000 0000  0000 0000 0000 0000   # ................
0310:  0000 0000 0000 0000  0000 0000 0000 0000   # ................
0318:  0000 0000 0000 0000  0000 0000 0000 0000   # ................
0320:  0000 0000 0000 0000  0000 0000 0000 0000   # ................
0328:  0000 0000 0000 0000  0000 0000 0000 0000   # ................
>
```

You'll notice that our value $6789 appears at location $2F2 (two words in on the top row), and at locations $300, $301 and $302 (first 3 positions in the third row).

Moving On

At this point, you may wish to continue with the remaining parts of this tutorial:

- Introducing the Instruction Set Part 2: Single Register and Implied Operand Instructions
- Introducing the Instruction Set Part 3: Branches, Conditional Branches and Calls
- Introducing the Instruction Set Part 4: Shift and Rotate Instructions

Or, you can return to the Programming Tutorials index.

Introducing the Instruction Set Part 2

This short part covers the Single Register Arithmetic instructions as well as the Implied Operand instructions. If you haven't read it already, please see Introducing the Instruction Set Part 1.

Single Register Arithmetic Instructions

These instructions are fairly simple instructions that operate on a single register:

TSTR	Check sign, zero on a register	
CLRR	Set a register to 0	R = 0
INCR	Increment a register	R = R + 1
DECR	Decrement a register	R = R - 1
NEGR	Negate a register	R = -R
COMR	1s complement a register	R = R XOR $FFFF
ADCR	Add carry bit to register	R = R + C

These instructions work equally on all 8 registers. That includes the program counter. For instance, "INCR R7" (aka. INCR PC) will skip the next instruction word. That can be useful for skipping single-word instructions. "DECR R7" (aka. DECR PC) is equivalent to "here: B here", but is one byte shorter.

Example: Consider the value of R0 at each step in this sequence as it executes:

```
CLRR R0   ; Clear R0:            R0 = 0
INCR R0   ; Add 1 to R0:         R0 = 1
DECR R0   ; Subtract 1 from R0:  R0 = 0
COMR R0   ; 1s complement R0:    R0 = $FFFF
NEGR R0   ; Negate R0:           R0 = 1
```

The ADCR instruction is useful for extended precision arithmetic. The following example shows how to add the 32-bit number in R3:R2 to the 32-bit number in R1:R0. (R3 and R1 hold the upper halves of the 32-bit numbers, whereas R2 and R0 hold the lower halves.)

```
; Add lower halves together.  This generates a carry in 'C'
ADDR R2, R0

; Add carry into upper half of result
ADCR R1

; Now add upper halves together.
ADDR R3, R1
```

Implied Operand Instructions

These instructions don't operate on any register:

NOP	No operation
SETC	Set carry flag
CLRC	Clear carry flag
EIS	Enable interrupts
DIS	Disable interrupts

SETC, CLRC and NOP are fairly self explanatory. EIS and DIS are also straightforward, though interrupts are beyond the scope of this tutorial. They are important, and I will cover them in a future tutorial.

Moving On

At this point, you may wish to continue with the remaining parts of this tutorial:

- Introducing the Instruction Set Part 1: The CPU, Memory and Registers; Primary Instructions and Addressing Modes
- Introducing the Instruction Set Part 3: Branches, Conditional Branches and Calls
- Introducing the Instruction Set Part 4: Shift and Rotate Instructions

Or, you can return to the Programming Tutorials index.

Introducing the Instruction Set Part 3

This segment of the tutorial introduces branches, particularly conditional branches and function calls. This is Part 3 of a series. If you haven't yet, you may wish to review at least Part 1 and Part 2.

Contents

Why do we need branches?

Branches and jumps provide a way to guide the CPU through a sequence of code. (Jump is more or less a synonym for branch.) Ordinarily, the CPU just keeps executing instructions in the order you wrote them. It's like going down a to-do list. For a simple example, suppose we wanted to add two numbers—call them A and B—together, and write the result to a third place—call it C. The steps might look like this:

1. Read A into a R0
2. Add B to R0
3. Write R0 to C

These steps translate to the following instruction sequence:

```
MVI  A,  R0    ; Read A into R0
ADD  B,  R0    ; Add B to R0
MVO  R0, C     ; Write R0 out to C
```

The CPU, when it encounters this sequence of instructions, will always execute them in this order. That's useful by itself, but not useful enough to write interesting programs.

For example, suppose you wanted to fill the entire screen with blank characters. The screen occupies 240 locations of memory starting at location $200. You need to write a blank character (which happens to be the value $0000) to all 240 locations. One way to do that would be to do this:

1. Set R0 to 0
2. Write R0 to $200
3. Write R0 to $201
4. Write R0 to $202

... 234 lines omitted ...

239. Write R0 to $2ED
240. Write R0 to $2EE
241. Write R0 to $2EF

That works, but it leads to very large and tedious code. The simple way to write this in assembly code would be:

```
CLRR R0
MVO  R0,  $200
MVO  R0,  $201
MVO  R0,  $202
; ... 234 lines omitted ...
MVO  R0,  $2ED
MVO  R0,  $2EE
MVO  R0,  $2EF
```

That runs very fast, but it takes up a ton of ROM. Also, it clears the screen only once. What if you want to clear the screen many times throughout your program, perhaps from different places? You really don't want to copy this same code everywhere you need to clear the screen. That's where branches come in.

Branches tell the CPU to start executing at a different location, rather than barreling forward. Unconditional branches and jumps tell the CPU to always jump to that location. Conditional branches tell the CPU to jump to a new location *if some condition is true.* These let you skip pieces of code, or repeat them a number of times. Jumps to subroutines (also known as "calls") tell the CPU "go do this and come back when you're done." The following sections explore these concepts a little more closely.

Branches and Labels

Like a "`goto`" statement in other languages, branches need to tell the CPU *where* to branch to. In assembly code, you have two options:

- Specify the address of where you'd like the CPU to branch to
- Specify the label of the instruction you'd like the CPU to branch to

Most of the time, you will want to write your code in terms of branches to labels. This is far, far easier than trying to figure out the addresses of instructions you might wish to jump to. See the Assembly Syntax Overview for details on how to write labels. All of the examples in this tutorial will use branches and jumps to labels.

Unconditional Branches and Jumps

Jump and Unconditional Branch instructions are like "goto" instructions found in higher-level languages. They tell the CPU to immediately start executing code from somewhere else as opposed to executing the next instruction in sequence. They're useful to tell the CPU to skip a section of code, or to repeat a section of code. Be careful when repeating a block of code with an unconditional branch: If there isn't a conditional branch somewhere in that block, it could end up running forever.

The following table lists the instructions:

Mnemonic	Description	Cycles	Size
B	Branch to label	9	2 words
J	Jump to label	12	3 words
JD	Jump to label while disabling interrupts	12	3 words
JE	Jump to label while enabling interrupts	12	3 words
JR	Jump to address in register	7	1 word

As you can see, the primary difference between branches and jumps is that branches are smaller and faster. Branches encode their "target address," the address being jumped to, as a relative offset from the current address. Jumps, on the other hand, store the actual address of the target. In most cases, especially in a 16-bit ROM, there are few reasons to use a J instruction, although the combination instructions, JD and JE can be useful.

There is also a pseudo-instruction, JR, that allows "jumping to a location held in a register." It is really a pseudonym for "MOVR Rx, R7". Because it is a MOVR instruction, it will modify the Sign Flag and Zero Flag, which may be confusing if you're not expecting it. Otherwise, it is an efficient method for jumping to an address held in an register, such as when returning from a CALL.

Conditional Branches

Conditional branches are similar to "`if ... goto`" type constructs found in other languages. They tell the CPU to start executing at another location if a particular condition is true. This allows us to build "`if-then-else`" types of statements, as well as "`for`" and "`while`" loops.

The CP1610 has a rich set of conditional branch instructions. They work by looking at the CPU's flag bits to decide when to branch. Instructions like CMP and DECR set these flags, making it easy to write those if/else statements and for/while loops. Even fancier uses are possible with some creativity.

The following table summarizes the conditional branches.

Mnemonic	**Name**	**Branch taken when...**	**Mnemonic**	**Name**	**Branch taken when...**
BC	Branch on Carry	`C = 1`	**BNC**	Branch on No Carry	`C = 0`
BOV	Branch on OVerflow	`OV = 1`	**BNOV**	Branch on No OVerflow	`OV = 0`
BPL	Branch if PLus	`S = 0`	**BMI**	Branch on MInus	`S = 1`
BEQ	Branch if EQual	`Z = 1`	**BNEQ**	Branch on Not Equal	`Z = 0`
BZE	Branch on ZEro		**BNZE**	Branch on Not ZEro	
BLT	Branch if Less Than	`S <> OV`	**BGE**	Branch if Greater than or Equal	`S = OV`
BNGE	Branch if Not Greater than or Equal		**BNLT**	Branch if Not Less Than	

BLE	Branch if Less than or Equal	`Z = 1 OR S <> OV`	**BGT**	Branch if Greater Than	`Z = 0 AND S = OV`
BNGT	Branch if Not Greather Than		**BNLE**	Branch if Not Less than or Equal	
BUSC	Branch on Unequal Sign and Carry	`S <> C`	**BESC**	Branch on Equal Sign and Carry	`S = C`

Conditional branches are most often used with numeric comparisons, or as the branch at the end of a loop. The following sections illustrate how numeric comparisons, increment and decrement work in concert with branches.

Conditional branches can also be paired with other instructions that manipulate the flags. For instance, shift instructions, as described in Part 4 update sign, zero, carry and overflow flags depending the operation performed. This can lead to interesting and creative combinations of shifts and branches.

Another use of flags and branches is to pass status information in CPU flags (such as the Carry Flag) and then act on that information later. The SETC and CLRC instructions make it easy to manipulate the Carry Flag to pass this status information around.

Signed Comparisons

The following branches are particularly useful when comparing signed numbers. These are the conditional branches you will use most when writing "`if-then-else`" statements, since most numbers are signed, or can be treated as signed.

Mnemonic		**Branch taken when...**	**Mnemonic**		**Branch taken when...**
BEQ	**BZE**	`Z = 1`	**BNEQ**	**BNZE**	`Z = 0`
BLT	**BNGE**	`S <> OV`	**BGE**	**BNLT**	`S = OV`
BLE	**BNGT**	`Z = 1 OR S <> OV`	**BGT**	**BNLE**	`Z = 0 AND S = OV`
BOV		`OV = 1`	**BNOV**		`OV = 0`

Note: One pair of branches shown above—BOV and BNOV—are useful in this context only for detecting overflow and little else. I included them here for completeness. These actually find more use paired up with shift instructions. Those are described Part 4 of this tutorial.

How Signed Arithmetic Works With Conditional Branches

The next couple of paragraphs describe how these branches work with compares. Most of the time, you do not need to think about these details when writing programs, so feel free to skim it for now and come back to it later.

The compare instruction compares two numbers by subtracting them, and then setting the flags based on the result. This provides a lot of information about the relative values of the two numbers, as this table shows (ignoring overflow):

If this is true...	**...then this also must be true...**	**...which implies the flags get set as follows (if you ignore overflow).**	
`x = y`	`x - y = 0`	S = 0	Z = 1
`x < y`	`x - y < 0`	S = 1	Z = 0
`x > y`	`x - y > 0`	S = 0	Z = 0

That is, we can determine whether two numbers are equal or not by looking at the Zero Flag. We can determine if one's less than the other by looking at the Sign Flag. At least, that would be true if there was no such thing as overflow.

The CP1610 can only work with 16 bits at a time. If you try to subtract two numbers whose values are very far apart, such as, in the worst case 32767 - (-32768), you will trigger an overflow, because the results don't fit in 16 bits. Overflow causes the sign of the result to be the opposite of what you would get if no overflow had occurred. The branches take this into account and look at the overflow bit in addition to the sign bit to decide whether one number is greater than or less than another. The following table illustrates the relationships both with and without overflow.

If this is true...	**...then the flags get set as follows...**			**...which matches these branches.**
`x = y`	S = 0	Z = 1	OV = 0	BEQ, BGE, BLE
`x < y`	S = 1	Z = 0	OV = 0	BNEQ, BLT, BLE
	S = 0	Z = 0	OV = 1	
`x > y`	S = 0	Z = 0	OV = 0	BNEQ, BGT, BGE
	S = 1	Z = 0	OV = 1	

As you can see, the flags and the branches cooperate quite nicely.

Gotchas With the CP1610 Compare Instruction

The syntax for the CP1610's compare instruction can confuse things slightly, since it does a "subtract from". Consider the following example:

```
MVII    #1, R0  ; R0 = 1
MVII    #2, R1  ; R1 = 2
CMPR    R0, R1  ; Subtract R0 from R1 to set flags
BLT     label   ; Is this taken?
```

This computes "R1 - R0", not "R0 - R1". It compares R0 to R1 by subtracting R0 from R1. In this example, that leaves S=0 and OV=0. R1 is *not* less than R0, so the branch is *not* taken. In other words, to determine if a given branch is taken, you have to read right-to-left. "Is R1 less than R0?" In this case, the answer is no, so the branch is not taken.

Unsigned Comparisons

These branches are useful when comparing unsigned numbers. Most often, these get used with pointers into your game ROM, or with certain fixed point values such as screen coordinates.

Mnemonic		Branch taken when...	Mnemonic		Branch taken when...
BC		`C = 1`	**BNC**		`C = 0`
BEQ	**BZE**	`Z = 1`	**BNEQ**	**BNZE**	`Z = 0`

The unsigned comparisons require some additional explanation to be useful. After comparing two unsigned numbers, the BC instruction will branch if the second number is greater than or equal to the first. The BNC instruction will branch if the second number is smaller than the first. The CP1610 does not offer unsigned equivalents for "branch if greater than" or "branch if less than or equal." You can get similar effects, though, by combining BC and BNC with BEQ or BNEQ to separate out the "equals", "greater-than" and "less-than" cases.

How Unsigned Arithmetic Works With Conditional Branches

The following couple of paragraphs explain how BC and BNC work in concert with other instructions to provide unsigned comparisons. Feel free to skim this for now and come back to it later.

Unsigned arithmetic is actually pretty similar to signed arithmetic, except that there isn't a sign bit. Furthermore, it turns out that all the interesting information ends up in the carry bit. Read "How Signed Arithmetic Works With Conditional Branches" above to get the basics.

As mentioned before, the compare instruction compares two numbers by subtracting them and setting the flags. When subtracting two unsigned numbers, the Carry flag doubles as a "do not borrow" flag. With that in mind (and handwaving just *how* carry functions that way for now), we have:

If this is true...	...then this also must be true...	...which implies the flags get set as follows

`x = y`	`x - y = 0`	C = 1 (no borrow)	Z = 1
`x > y`	`x - y > 0`	C = 1 (no borrow)	Z = 0
`x < y`	`x - y < 0`	C = 0 (borrow needed)	Z = 0

As you can see, the carry flag gets set whenever 'x' is greater than or equal to 'y'. The carry flag is clear whenever 'x' is less than 'y'. This is why BC works as the equivalent of an unsigned "branch if greater than or equal," and BNC works as an unsigned version of "branch if less than."

Sign/Zero Comparisons

Common looping instructions, such as DECR and INCR, as well as other useful instructions such as the TSTR instruction only modify the sign and zero flags without updating the carry or overflow flags. Such instructions are best used with the following branches:

Mnemonic		**Branch taken when...**	**Mnemonic**		**Branch taken when...**
BPL		`S = 0`	**BMI**		`S = 1`
BEQ	**BZE**	`Z = 1`	**BNEQ**	**BNZE**	`Z = 0`

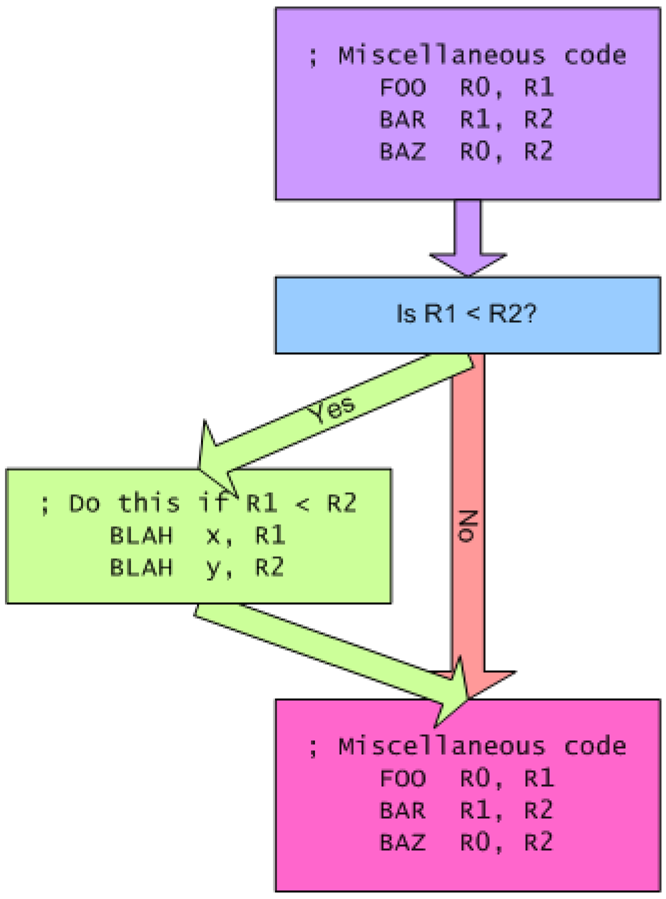
These will be explored in more detail in the "looping" section below.

If-Then and If-Then-Else

The `if-then` construct lets you say "Run this bit of code here only if this is true." The `if-then-else` construct is similar, with the added twist of letting you say "...or, run this other bit of code if it isn't true."

If-Then

Let's start with the `if-then` statement. The following diagram shows its overall structure. In this case, the code says "if R1 < R2, run the extra bit of code on the left." (green background)

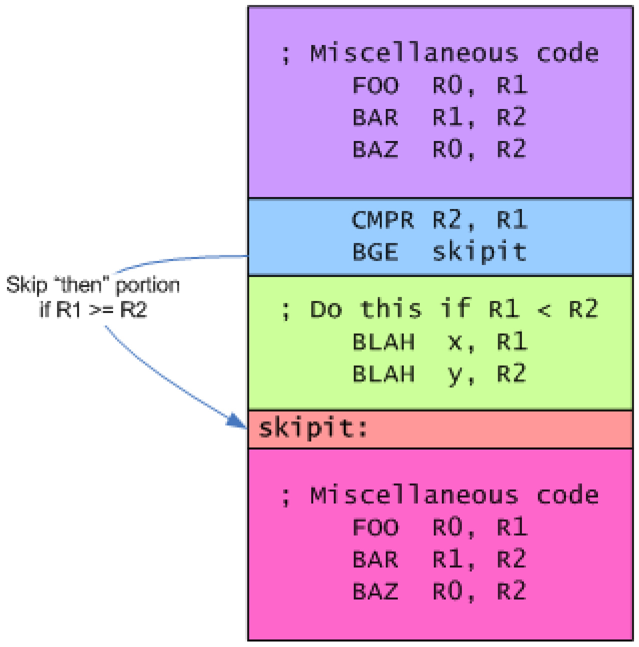


```
; Miscellaneous code
    FOO   R0, R1
    BAR   R1, R2
    BAZ   R0, R2
```

```
; Do this if R1 < R2
    BLAH   x, R1
    BLAH   y, R2
```

```
; Miscellaneous code
    FOO   R0, R1
    BAR   R1, R2
    BAZ   R0, R2
```

In this example, the CPU runs the green portion only if R1 is less than R2, as indicated in the blue square. The other bits of code always run. The diagram shows how this looks as assembly language code:

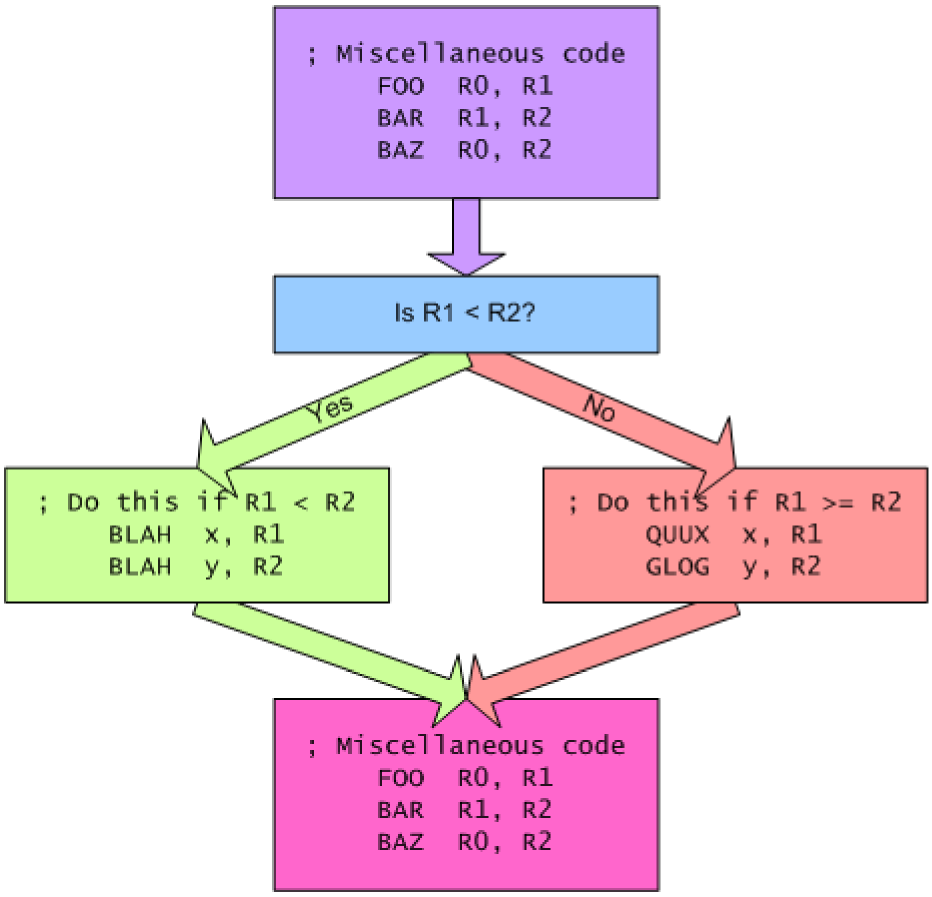


```
; Miscellaneous code
    FOO   R0, R1
    BAR   R1, R2
    BAZ   R0, R2
    CMPR  R2, R1
    BGE   skipit
; Do this if R1 < R2
    BLAH   x, R1
    BLAH   y, R2
skipit:
; Miscellaneous code
    FOO   R0, R1
    BAR   R1, R2
    BAZ   R0, R2
```

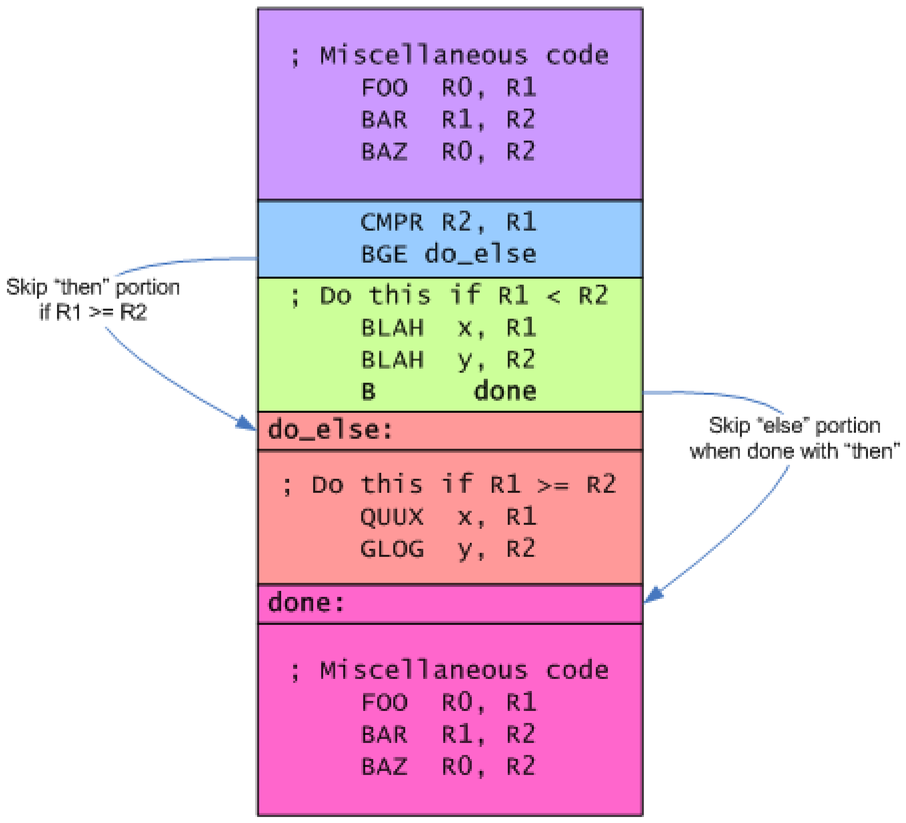
The code in the blue square implements the actual "if" code. It first compares R2 to R1 using the CMPR instruction, and then branches around the "then" portion if R1 is greater-than or equal to R2 with BGE. That way, the green portion only runs if R1 is less than R2, as we intended. (See Gotchas With the CP1610 Compare Instruction for an explanation as to why this is `CMPR R2, R1` and not `CMPR R1, R2`.)

If-Then-Else

The `if-then-else` statement is very similar. It just adds code that runs when the "if" condition is false, as shown in this diagram:



As with the previous example, the code in the green box on the left runs when R1 is less than R2. The code in red box at the right runs when R1 is greater than or equal to R2. The CPU either executes one or the other. It doesn't execute both. So how does this look in assembly code?



Again, the blue box holds the code that implements the "if". As with the previous example, the branch gets taken if R1 is greater than or equal to R2, thereby skipping the "then" portion of code. The branch now goes to the "else" code in the red box. The "then" code has an additional instruction: `B done`. What this does is skip the code for the "else" portion when the "then" portion is done. In either case, the CPU continues executing with the final block of code at the bottom.

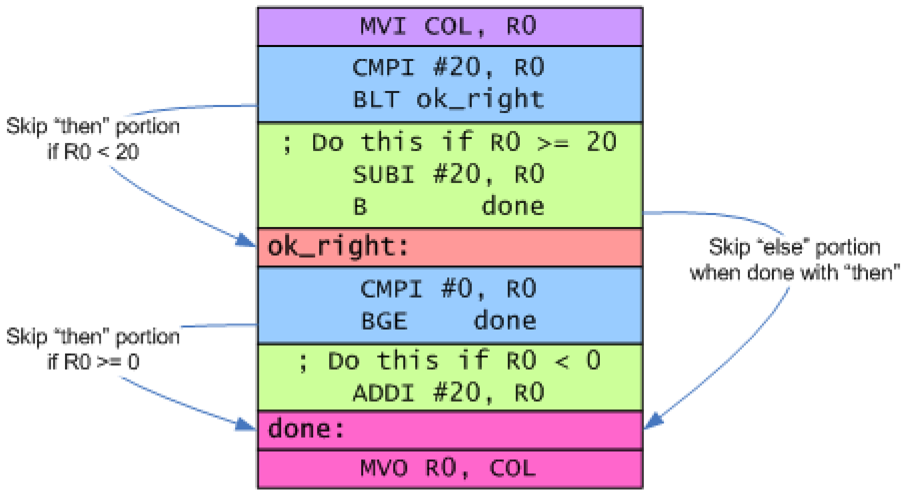
Concrete Example: Staying On Screen

The above examples are somewhat abstract in nature. The following example is somewhat more concrete: Make sure `COL` stays in the range 0..19. This corresponds to the column coordinate in BACKTAB. If `COL` gets bigger than 19 (beyond the right edge of the screen), wrap around to the left edge by subtracting 20, else if `COL` gets smaller than 0, wrap around to the right edge by adding 20. The following example assumes `COL` is in 16-bit memory for now.

```
          MVI   COL, R0    ; Put COL into R0

          CMPI #20, R0     ; Is COL >= 20?
          BLT  ok_right    ; Branch around if COL < 20
          SUBI #20, R0     ; Wrap around from right edge to left by subtracting 20
          B    done        ; Skip the rest of the code, since we know we're not off the left

ok_right: CMPI #0,  R0     ; Is COL < 0?
          BGE  done        ; Branch around if COL >= 0
          ADDI #20, R0     ; Wrap around from left edge to right edge by adding 20

done:     MVO  R0,  COL    ; Write the final result out
```

Notice that this code has both an `if-then` and an `if-then-else`. The first test, "`COL >= 20`" is part of an `if-then-else`. The "then" clause subtracts 20 from `COL` and then branches around the "else" clause. The "else" clause holds the second test, "`COL < 0`" as part of an `if-then` statement. That statement's "then" clause adds 20 to `COL` if it is indeed less than 0.

All paths finally lead to `done`, where the updated value of `COL` gets written back to memory. The following diagram shows how it all works:



A quick note on efficiency: The above code works, but it can be made slightly faster and more compact using `TSTR` instead of `CMPI #0`. The `TSTR` instruction sets the sign and zero based on the value in a register. In this case, we're looking for whether `COL` is less than 0, which means its sign bit would be '1'. Thus:

```
ok_right: CMPI #0,  R0    ; Is COL < 0?
          BGE  done       ; Branch around if COL >= 0
```

becomes:

```
ok_right: TSTR R0         ; Is COL < 0?
          BPL  done       ; Branch around if COL >= 0
```

This is 2 cycles faster and 1 word shorter. Most of the time you won't notice the difference, but sometimes every cycle and every word counts.

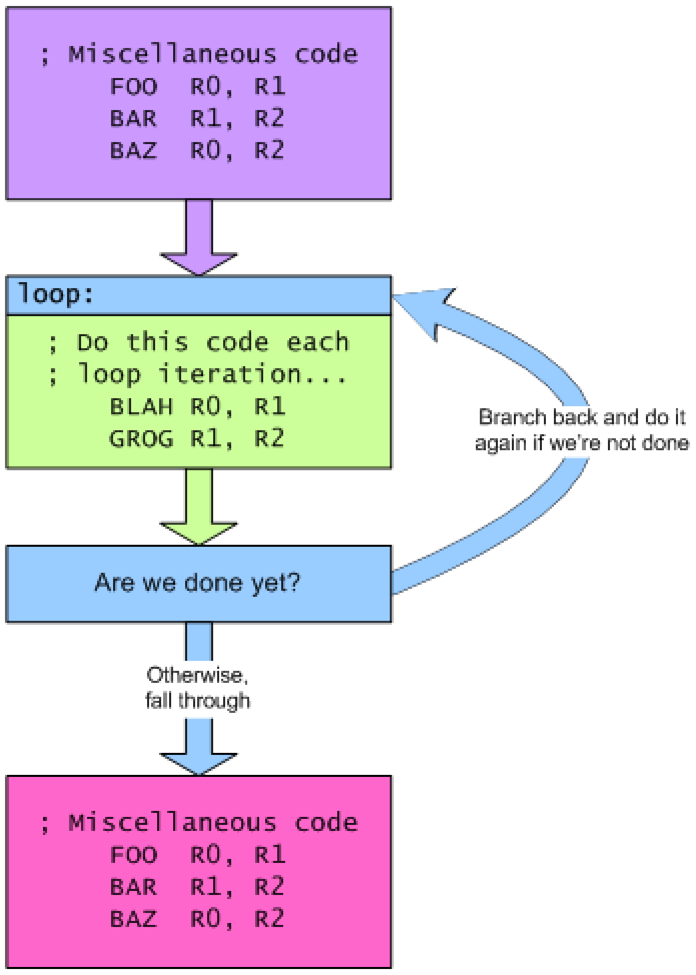
Looping

Loops tell the CPU to repeat a sequence of instructions some number of times. There are two main kinds of loops, traditionally referred to as `for` loops and `while` loops. `For` loops know how many times they'll iterate before they start. `While` loops figure out when to stop iterating based on something that happens during the loop. Clearing the screen is an example of a `for` loop: It will iterate 240 times, writing a 0 to each location of BACKTAB. Finding the length of a string is an example of a `while` loop: It iterates until it finds the NUL character at the end of the string.

(Note: Do not confuse these terms with the `for` and `while` keywords from languages like C and C++. You can write either kind of loop with either keyword in those languages.)

Basic Structure

In assembly code, loops tend to have the structure shown below. This structure works for both `for` and `while` loops. The green part in the middle is called the "body" of the loop. The blue part is sometimes called the "loop test" or "loop condition." The difference between a `for` loop and a `while` loop is in the loop condition.

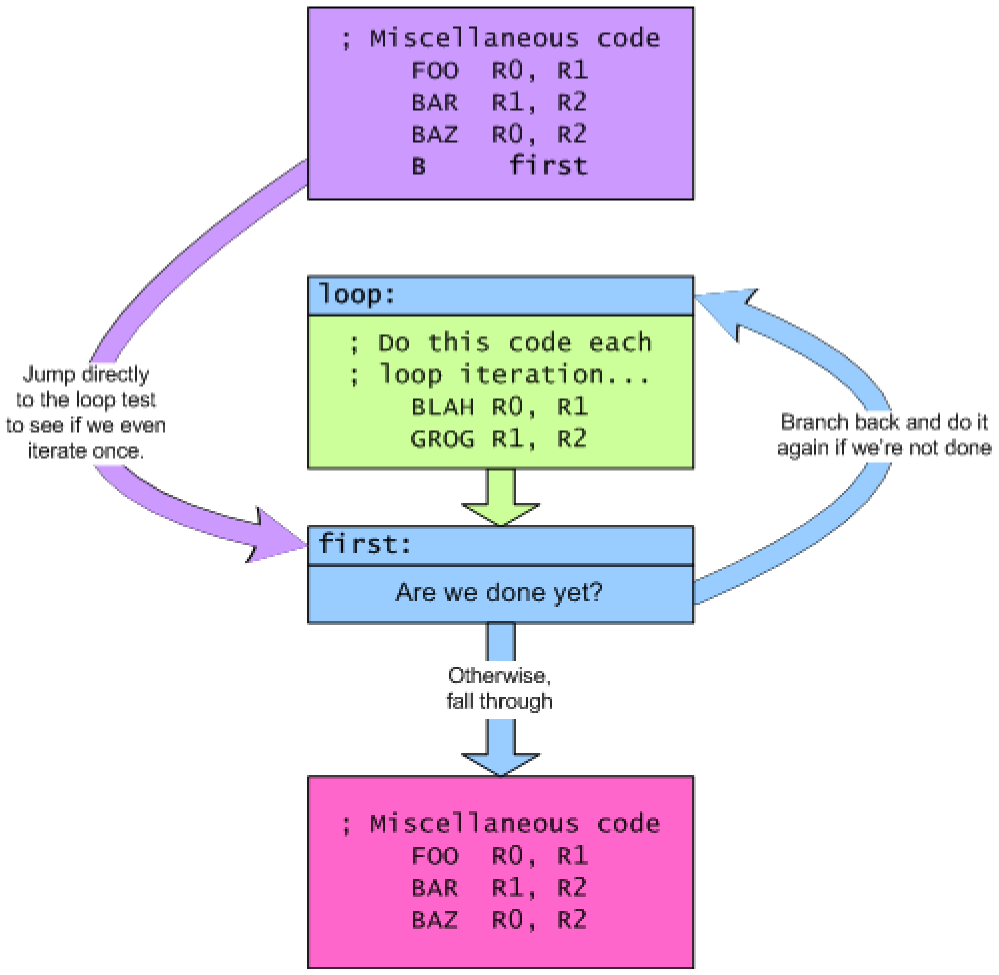


```
; Miscellaneous code
    FOO  R0, R1
    BAR  R1, R2
    BAZ  R0, R2
```

```
loop:
  ; Do this code each
  ; loop iteration...
    BLAH R0, R1
    GROG R1, R2
```

```
; Miscellaneous code
    FOO  R0, R1
    BAR  R1, R2
    BAZ  R0, R2
```

If you look carefully, you'll see that this loop always executes the green part at least one time—aka. one iteration. Sometimes, though, you don't want that. This can be accomplished by branching around the body of the loop and directly to the loop test, as shown below:



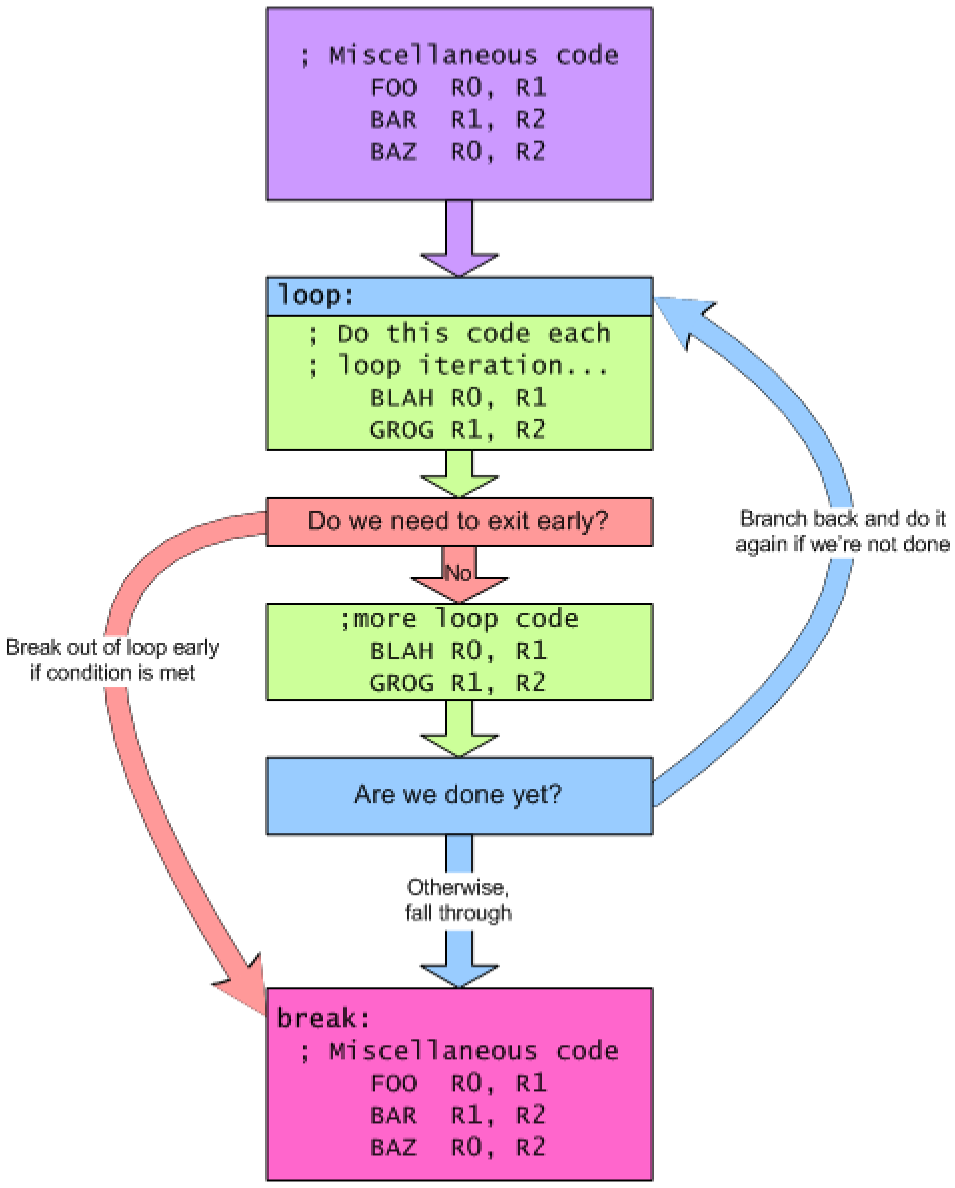
In `for` loops, the loop test is generally a decrement and a branch. The DECR instruction will set the zero flag when the register it decrements becomes zero. This makes it easy to write loops that iterate a prescribed number of times. The following example iterates 10 times:

```
        MVII #10, R0
loop:   ; (Put the code to repeat here.  It will get executed 10 times.)

        DECR R0         ; Count down from 10 down to 0
        BNEQ loop       ; Iterate as long as count isn't expired
```

`While` loops have more varied loop tests. What they look like depends on what the loop is trying to accomplish.

Terminating a Loop Early

Sometimes, you'll want to break out of a loop early if some alternate condition gets matched. You can do this by putting a test and branch in the body of the loop, with the branch pointing outside the loop. This is similar to the action of C's `break` keyword. The following diagram illustrates this structure:



"Searching" loops often look like this, breaking out of the loop when they find whatever it is they're searching for. Often, the branch that breaks out of the loop will branch somewhere other than the code immediately following the loop. This makes it easy to give different behavior based on whether the loop found the sought item or not. The example below illustrates this.

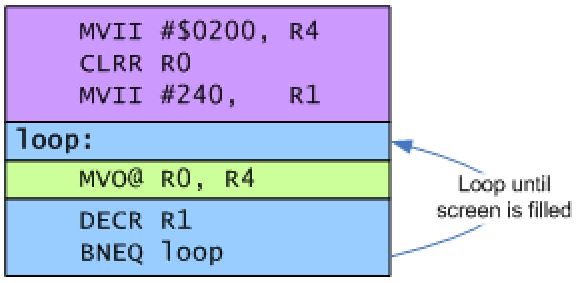
Concrete For Loop Example: Clearing the screen

To clear the screen, we need to write a blank character to each of the 240 locations in BACKTAB, starting at location $200. The blank character happens to be $0000. The following simple code accomplishes this:

```
        ; Clear the screen
        MVII    #$0200,  R4   ; Point to BACKTAB
        CLRR    R0            ; Set R0 = $0000
        MVII    #240,    R1   ; Use R1 as our "loop counter"
loop:
        MVO@    R0,      R4   ; Write a blank to the screen.
        DECR    R1            ; Count down the loop counter
        BNEQ    loop          ; Keep looping until counter goes to zero
```

This code uses Indirect Mode accesses, as described in Part 1 of the tutorial. It uses R4 as the indirect pointer. R4 has the special property that its value increments after each indirect access. So, on the first iteration of the loop, the `MVO@ R0, R4` instruction writes to location $200. On the second iteration, it writes to $201, and so on. Very handy.

The following diagram illustrates the structure of the example above:



```
    MVII #$0200, R4
    CLRR R0
    MVII #240,   R1
loop:
    MVO@ R0, R4
    DECR R1
    BNEQ loop
```

Concrete While Loop Example: Finding the length of a string

This example is a little more involved. In this case, we want to find the length of a string pointed to by R4. There are many ways we can do this. The way shown in this example is not the most efficient, but it's not intended to be. The overall sequence of steps are:

1. Set the string length to 0 initially
2. Get next character from string
3. Is it NUL (0)? If so, we're done.
4. Increment our string length and go back to step #2.

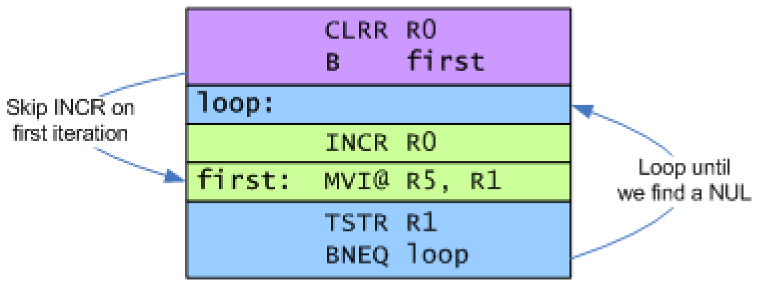
It's more efficient to test the loop condition at the bottom of the loop. The sequence above has that in the middle. The most common trick is to "rotate" the loop, and just skip a portion of it on the first iteration:

1. Set the string length to 0 initially
2. Skip step #3. (This only happens one time.)
3. Increment the string length.
4. Get the next character from the string.
5. If it isn't NUL, go back to step #3.

As a result, this is an example of a loop that may iterate zero times, since the string itself may be empty and so we'll never increment the string length. We use an unconditional branch to skip the increment on the first iteration. The code ends up looking like this:

```
       ; Assume R4 already points to the string of interest
       CLRR    R0           ; Zero out our string length
       B       first        ; Skip the increment on first iteration

loop:  INCR    R0           ; Increment our string length
first: MVI@    R4, R1       ; Get next character from string
       TSTR    R1           ; See if it's NUL
       BNEQ    loop         ; If not, keep looping
```

The following diagram illustrates the structure:



```
        CLRR R0
        B    first
loop:
        INCR R0
first:  MVI@ R5, R1
        TSTR R1
        BNEQ loop
```

As with the previous example, this code uses Indirect Mode with R4 to step through a range of memory. This is one of the most common uses of R4 and R5. In this example, R4 will read the first character of the string on the first iteration, the second on the next, and so on. At the end of the loop, R4 will point one location beyond the NUL character that terminates the string and R0 will hold the length of the string.

Exercise for the reader: How would you write this more efficiently?

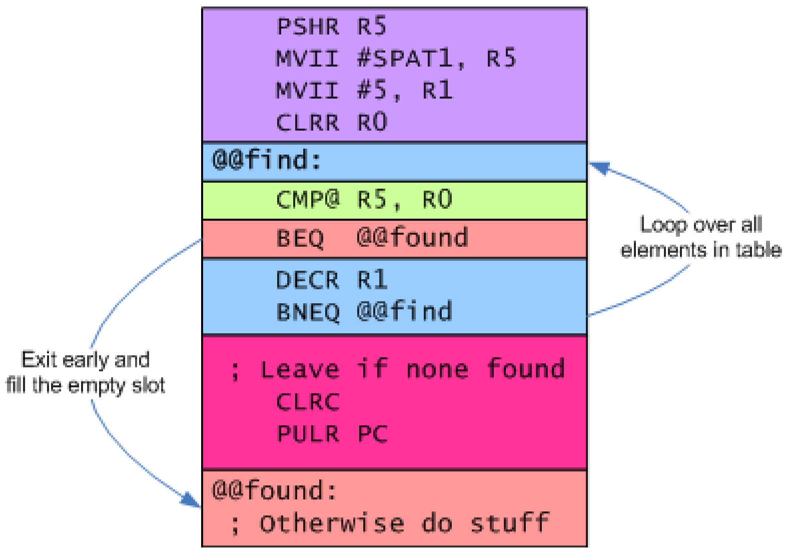
Concrete Example of an Early Exit: Finding a slot in a table

The following example comes directly from Space Patrol (http://spacepatrol.info/src/bg/bgengine.asm) . The following loop looks for the first zero element in a table. The table is 5 elements long. If there is no non-zero element, it clears the carry flag and leaves. Otherwise, it proceeds to make a new bad guy by filling that slot in the table.

```
        PSHR    R5
@@1:
        ;; ---------------------------------------------------------------- ;;
        ;;  Find first slot in group 1 of the MOBs.                         ;;
        ;; ---------------------------------------------------------------- ;;
        MVII    #SPAT1, R5
        MVII    #5,     R1
        CLRR    R0
@@find:
        CMP@    R5,     R0
        BEQ     @@found
        DECR    R1
        BNEQ    @@find

        CLRC
        PULR    PC                  ;; Didn't find one, so leave.

        ;; ---------------------------------------------------------------- ;;
        ;;  Slot is open so copy over the attribute for the new bad-guy.    ;;
        ;; ---------------------------------------------------------------- ;;
@@found:
```

In this case, we have a `for` loop that iterates 5 times, but exits early if it finds an empty slot in the table. (`SPAT1` stands for "SPrite Attribute Table 1") Furthermore, if it does exit early, it exits to a different path than if it makes 5 full iterations. This is a useful idiom for searching and finding.

The following diagram illustrates the overall structure of the code:



```
    PSHR R5
    MVII #SPAT1, R5
    MVII #5, R1
    CLRR R0
@@find:
    CMP@ R5, R0
    BEQ  @@found
    DECR R1
    BNEQ @@find
  ; Leave if none found
    CLRC
    PULR PC
@@found:
  ; Otherwise do stuff
```

Loop over all elements in table
Exit early and fill the empty slot

Function Calls

Functions, also known as procedures or subroutines, are isolated bits of code that perform some action. These bits of code are likely to be called from many places. Functions provide a useful way to encapsulate functionality and structure your program. The Jump to SubRoutine family of instructions provide a handy mechanism for doing this:

Mnemonic		Description	Cycles	Size
JSR	**CALL**	Jump to SubRoutine at label	12	3 words
JSRD		Jump to SubRoutine at label while disabling interrupts	12	3 words
JSRE		Jump to SubRoutine at label while enabling interrupts	12	3 words

Each of the JSR instructions take two arguments: A register and a label (or address). The CPU puts the *return address* in the specified register and then jumps to the specified label. The return address is the address of the word following the JSR instruction. The `CALL label` instruction is a pseudonym for `JSR R5, label`.

Simple Call/Return

Many functions perform a simple task and then return. The screen clearing example above is an example of this. With a little extra code, we can turn that into a function named CLRSCR:

```
CLRSCR PROC
       ; Clear the screen
       MVII    #$0200,   R4   ; Point to BACKTAB
       CLRR    R0             ; Set R0 = $0000
       MVII    #240,     R1   ; Use R1 as our "loop counter"
@@loop:
       MVO@    R0,       R4   ; Write a blank to the screen.
       DECR    R1             ; Count down the loop counter
       BNEQ    @@loop         ; Keep looping until counter goes to zero
```

```
        JR      R5              ; Return from function call
        ENDP
```

The additions to the code are minor. `PROC` and `ENDP` directives tell the assembler where the "procedure" begins and ends. This mainly provides a scope for local labels, such as `@@loop` in this example. The Assembly Syntax Overview describes these directives in a little more detail.

The other main addition is the `JR R5` instruction. What this does is return from the function back to the code that called it. How does this work?

When calling this function with `CALL CLRSCR` (or `JSR R5, CLRSCR`), the CPU will copy the address of the instruction following the call into R5, and then branch to the function. The `JR R5` instruction tells the CPU to jump back to that location, in effect returning from the called function.

This works as long as the function doesn't use R5 itself. If the function uses R5 for some purpose, it needs to save the return address somewhere—either another register, in a location in memory, or on the stack. Saving the return address on the stack is the most popular. Here's the same function, modified to save the return address on the stack, even though it doesn't strictly need to:

```
CLRSCR PROC
        PSHR    R5              ; save return address on the stack

        ; Clear the screen
        MVII    #$0200,  R4     ; Point to BACKTAB
        CLRR    R0              ; Set R0 = $0000
        MVII    #240,    R1     ; Use R1 as our "loop counter"
@@loop:
        MVO@    R0,      R4     ; Write a blank to the screen.
        DECR    R1              ; Count down the loop counter
        BNEQ    @@loop          ; Keep looping until counter goes to zero

        PULR    PC              ; Return from function call
        ENDP
```

The `PSHR R5` instruction saves the return address on the stack. The `PULR PC` pops the top item off the stack, and puts it in the program counter. (Note that PC is just a synonym for R7.)

Nested Call/Return

Sometimes, a function will need to call one or more other functions to complete its task. To do this, one merely needs to save and restore the return address before it calls the other functions. The following example, "MYFUNC" calls CLRSCR to clear the screen. It doesn't actually do much else:

```
MYFUNC  PROC
        PSHR    R5      ; Save return address
        CALL    CLRSCR  ; Clear the screen
        PULR    PC      ; Return
        ENDP
```

Passing Arguments to Functions

In the previous `CLRSCR` example, the function itself didn't take any inputs. (Inputs to functions are also referred to as arguments or parameters.) One way to pass arguments into a function is to set up the values in CPU registers. The caller and callee need to agree on the meaning of the various registers for this to work.

Suppose we wanted to generalize `CLRSCR` a little bit, and have it just be a generic memory-fill function. That function would need three arguments:

1. The address of memory to fill
2. The number of locations to fill
3. The value to fill with

An example implementation of `FILLMEM` appears below. It expects the address of the location to fill In R4, the number of locations to fill in R1 and the value to fill in R0.

```
FILLMEM PROC
@@loop: MVO@    R0, R4
        DECR    R1
        BNEQ    @@loop
        JR      R5
        ENDP
```

This is somewhat shorter than the `CLRSCR` code, mainly because the instructions that set up R0, R1 and R4 are omitted. `FILLMEM` expects the calling function to set these up. It's now possible to write `CLRSCR` in terms of a call to `FILLMEM`. This isn't the best way to write `CLRSCR`, but it is a useful example to illustrate the concepts.

```
CLRSCR  PROC
        PSHR    R5           ; Save return address

        MVII    #$200, R4    ; Point to BACKTAB
        MVII    #240,  R1    ; Clear 240 locations
        CLRR    R0           ; Prepare to write zeros

        CALL    FILLMEM      ; Fill the screen with zeros

        PULR    PC           ; Return to our caller
        ENDP
```

(Note: For a more compact way of writing `CLRSCR` and `FILLMEM`, take a look at the optimized implementation in SDK-1600.)

Passing Arguments via Return Address

In many cases, the arguments to a function will be fixed for a given call of that function. For example, when your game prints "Game Over", it likely prints it at the same position every time, in the same color, and the text always reads "Game Over." Setting up these values—the position, format and pointer to the string—takes up a bit of code space. Each `MVII` instruction is 2 words long. That adds up quickly.

Fortunately, the CP1610 makes it easy to pass static arguments as just plain data, without needing a block of `MVII` instructions before each call. If a particular function is going to get called regularly with certain arguments fixed at the call site as in the example above, then it makes sense to pass arguments via the return address.

This relies on the fact that CALL sets R5 to point to the word just after the CALL instruction. If you put data here, then MVI@ R5 will read that data. The SDK-1600 PRINT function uses this technique to great effect.

The following example shows a different version of FILLMEM and CLRSCR that mimic the example above. The difference is that this code passes arguments as data following the CALL instruction.

```
FILLMEM PROC
        MVI@     R5,   R0    ;  Get value to fill
        MVI@     R5,   R1    ;  Get number of locations to fill
        MVI@     R5,   R4    ;  Get address to fill at

@@loop: MVO@     R0,   R4    ;  Fill a location
        DECR     R1          ;  Count down
        BNEQ     @@loop      ;  Loop until it's all filled

        JR       R5          ;  Jump to the code after the data we read above.
        ENDP

CLRSCR  PROC
        PSHR     R5          ; Save return address

        CALL     FILLMEM
        DECLE    0           ;  Fill with zeros
        DECLE    240         ;  Fill 240 locations
        DECLE    $200        ;  Fill starting at $200

        PULR     PC          ; return
        ENDP
```

For many library functions, such as PRINT, you may want to have multiple entry points into the same function that read a different number of arguments as data vs. expecting arguments in registers. The PRINT function linked above works this way, providing a high degree of flexibility in how it's called. The way this is accomplished is simple in many cases. You can put local labels on the various MVI@ R5 instructions at the top of the function, making it easy for callers to pick which parameters get read from after the CALL and which get passed in registers.

Here's a modified version of FILLMEM. The local label @@vla denotes the entry point that will read "value", "length" and "address" as data after the CALL. This local label is known to other functions as FILLMEM.vla. The @@la entry point (aka FILLMEM.la) will read just "length" and "address" as data after the CALL. The value to fill needs to be set in R0. And so on. The @@r entry point (aka FILLMEM.r) expects all arguments to come in registers.

```
FILLMEM PROC
@@vla:  MVI@     R5,   R0    ;  Get value to fill
@@la:   MVI@     R5,   R1    ;  Get number of locations to fill
@@a:    MVI@     R5,   R4    ;  Get address to fill at
@@r:

@@loop: MVO@     R0,   R4    ;  Fill a location
        DECR     R1          ;  Count down
        BNEQ     @@loop      ;  Loop until it's all filled

        JR       R5          ;  Jump to the code after the data we read above.
        ENDP
```

As you can see, this technique provides a great deal of flexibility in how things are called.

Call Chaining: Having One Function Return for Another

Sometimes, the last thing you do at the end of one function is call another. When that function returns, the only work left to do is to return again. While that works, it's often possible to "chain" the calls—that is, branch to the next function and let it return for you. Note that **this is purely an optimization.** If it is at all unclear to you, I recommend you steer clear of it unless you're in a crunch. Feel free to skip this section and come back to it if you need it later.

Suppose we write a function that spins for a moment and then clears the screen. There are many ways to do this, each with different advantages and disadvantages. The code below has the advantage of being easy to write and understand. This first version shows a normal call/return sequence:

```
; Delay for a moment and then clear the screen
DLYCLR  PROC
        PSHR    R5          ; Save the return address

        MVII    #$FFFF, R0
        ; Loop a bunch of times doing nothing
@@spin: NOP
        NOP
        DECR    R0
        BNEQ    @@spin

        ; Now clear the screen
        CALL    CLRSCR

        PULR    PC          ; Return to our caller
        ENDP
```

Here, we save our return address, then we call out to `CLRSCR`. At the end, `CLRSCR` returns to us, and we return to our caller. This can be made more efficient by having `CLRSCR` return directly to our caller for us. This saves stack space (which is often at a premium) and can save cycles. (This example is clearly not cycle-count sensitive.)

Consider the following version of the above example:

```
; Delay for a moment and then clear the screen
DLYCLR  PROC
        MVII    #$FFFF, R0
        ; Loop a bunch of times doing nothing
@@spin: NOP
        NOP
        DECR    R0
        BNEQ    @@spin

        ; Now clear the screen and return
        B       CLRSCR
        ENDP
```

Notice how this version of the code doesn't save the return address, and in fact doesn't do anything with it. Instead of using `CALL` to invoke `CLRSCR`, it merely branches to it. Since the return address for `DLYCLR` is already in R5, when `CLRSCR` completes and branches to it, it'll return to `DLYCLR`'s caller. This is what "call chaining" is all about: In this case, `CLRSCR` returns on behalf of `DLYCLR`. Rather than having `DLYCLR` call `CLRSCR`, it instead branches—aka. chains—to it.

It's not always possible to chain calls like this. In particular, if you pass arguments after your `CALL` instruction, as described in the previous section, you cannot chain to that call.

As a pragmatic measure, I suggest avoiding call-chaining unless you need the stack space, code space or cycles that it saves. Write your code initially without it, and optimize later if you need it.

Indirect Branches and Jump Tables

*"It was a 'Jump to Conclusions' mat. You see, it would be this mat that you would put on the floor... and would have different **conclusions** written on it that you could **jump to.**"* -- Tom Smykowski, Office Space

Indirect branches are branches whose destination isn't encoded into the instruction itself. You've seen a special version of this above: The `JR R5` and `PULR PC` sequences for returning from function calls are both forms of indirect branches.

These sorts of branches are useful for many things. You can implement multi-way branches similar to C's `switch/case` construct (or BASIC's `ON .. GOTO`), or call-backs and so on. The following sections explore some of these uses.

Jump Vectors

Jump vectors are locations in memory that hold addresses of functions to jump to. For example, when an interrupt occurs, the EXEC does a tiny bit of housekeeping (it saves the registers on the stack), and then it looks in locations $100-$101 for the address of an interrupt service routine to jump to. This allows the game to change what function handles interrupts to suit its needs, despite the fact that the interrupt is handled initially by the EXEC ROM.

The following code illustrates how to set up the interrupt service vector. Since that vector is in 8-bit memory and program addresses are 16 bits, the vector occupies two locations.

```
        MVII    #MYISR, R0    ; MYISR is the label of the function that will get called
        MVO     R0,     $100  ; Write out lower 8 bits of "MYISR" to location $100
        SWAP    R0            ; Swap upper/lower halves of address
        MVO     R0,     $101  ; Write out upper 8 bits of "MYISR" to location $101
```

Other uses for jump vectors include setting up dispatch addresses for interesting events, such as receiving controller input, a timer expiring, or two objects colliding. Jumping via a vector is easy. The following code snippets show how to do this for addresses in 8-bit and 16-bit memory.

```
        ; Jumping via vector stored in 8-bit memory
        MVII    #@@ret, R5    ; Set up return address in R5 (if needed)
        MVII    #vector,R4    ; Address of vector in 8-bit memory
        SDBD                  ; Read vector as Double Byte Data
        MVI@    R4,     R7    ; Read vector into program counter
@@ret:
```

```
        ; Jumping via vector stored in 16-bit memory
        MVII    #@@ret, R5    ; Set up return address in R5 (if needed)
        MVI     vector, R7    ; Read vector into program counter
@@ret:
```

Space Patrol (http://spacepatrol.info) uses a variation on this technique to manage the "bad guys." Instead of storing the address of the "thinker" associated with each bad guy, it instead stores a small integer index saying which thinker to call. A separate table expands that small integer into an actual thinker address. That blends the jump-vector idea with the next topic, jump tables.

Simple Jump Tables

Jump tables are one way to implement `switch`/`case` type functionality. The basic idea is to associate branch targets with small integer indices, by way of a simple lookup table. For example, consider the following fragment of C code:

```
switch (x)
{
    case 0:
    {
         /* do stuff for case 0 */
         break;
    }
    case 1:
    {
         /* do stuff for case 1 */
         break;
    }
    case 2:
    {
         /* do stuff for case 2 */
         break;
    }
    case 3:
    {
         /* do stuff for case 3 */
         break;
    }
}
```

For those of you who don't know C, here's what it does: Based on the value of 'x', which presumably is in the range 0..3, run one of the four indicated blocks of code. We can do the same thing with a jump table in assembly code. (Note that the assembly code's behavior is undefined if 'x' is outside the range 0..3. It'll most likely crash.)

```
        MVI     X,       R1    ; get the value of 'X'.  Presumably it's 0..3.
        ADDI    #@@jtbl, R1    ; Index into the jump_table
        MVI@    R1,      R7    ; jump to the specified address
@@jtbl: DECLE   @@case0, @@case1, @@case2, @@case3

@@case0 ; do stuff for case 0
        B       @@done

@@case1 ; do stuff for case 1
        B       @@done

@@case2 ; do stuff for case 2
        B       @@done

@@case3 ; do stuff for case 3

@@done:
```

Adding to the Program Counter

Sometimes, you'll run across a situation where the various places you're branching to are a fixed number of words apart. This next technique is a little tricky, but is very efficient for handling this case.

For example, suppose I want to shift a number left 4, 3, 2 or 1 positions. I know that the SLL opcode is only 1 word long. If I put four SLL instructions in a row, and then jump to the appropriate one in the list, I will get the desired overall shift amount. The basic idea is this:

```
        MAGIC_BRANCH   ; skips 0, 1, 2, or 3 of the following instructions
        SLL     R0, 1
        SLL     R0, 1
        SLL     R0, 1
        SLL     R0, 1
```

The "MAGIC_BRANCH" is really just a sequence that adds a register to the program counter based on how many instructions we intend to skip. Suppose R1 held the shift amount. The following example show how to use that to skip some number of SLL instructions based on that:

```
        NEGR    R1         ; \_ Subtract R1 from 4 so it's now 3..0
        ADDI    #4, R1     ; /
        ADDR    R1, R7     ; Skip 0, 1, 2 or 3 of following instructions
        SLL     R0, 1
        SLL     R0, 1
        SLL     R0, 1
        SLL     R0, 1
```

The code subtracts the shift amount from 4, because a shift of 4 needs to skip 0 instructions, whereas a shift of 1 needs to skip 3 instructions. When you add to the program counter, as this example does, the value in the program counter at the time of the add is the address of the next instruction. This is why a value of 0 skips zero instructions.

You can also use this technique in combination with branches as an alternate way of implementing a `switch`/`case` statement. This method is larger and slower than most other methods, though. It made more sense with 10-bit wide ROMs than with today's 16-bit wide ROMs. The following example shows the `switch`/`case` from the previous section rewritten to use this technique. Note that branch instructions are 2 words long.

```
        MVI     X,       R1    ; get the value of 'X'.  Presumably it's 0..3.
        SLL     R1,      1     ; multiply X by 2
        ADD@    R1,      R7    ; jump to one of the following three branch inst. or @@case3
        B       @@case0
        B       @@case1
        B       @@case2
@@case3 ; do stuff for case 3
        B       @@done

@@case0 ; do stuff for case 0
        B       @@done

@@case1 ; do stuff for case 1
        B       @@done

@@case2 ; do stuff for case 2

@@done:
```

The example also shows a minor optimization: The highest numbered case can always be put right at the end of the list of branches, rather than having a branch to it there.

Clever (or Silly) Branch Tricks

The program counter, R7, can be accessed by almost any of the CPU's instructions. This leads to several cute tricks that border on being a bit too clever. Since these show up in actual code, it's worth at least explaining how they work.

Using INCR to Skip an Instruction

The instruction `INCR PC` adds 1 to the program counter. The net effect of this is skip the next word of code. If the next instruction is a 1 word instruction, this acts like a compact version of an unconditional branch. Why is this attractive? It is 1 word smaller and 2 cycles faster than a `B` instruction. It only works for skipping single-word instructions, and will modify the sign and zero flags, which may not be what you want.

A common place where this shows up is in function prologues. You might have multiple entry points to a function, but you want to skip, say, the `PSHR R5` that's on one of them. Example:

```
ENTRY1  PROC
        PSHR    R5

        ; do some stuff

        INCR    PC    ; skip PSHR R5
ENTRY2  PSHR    R5

        ; do more stuff

        PULR    PC    ; return
        ENDP
```

Using ADCR to Conditionally Skip an Instruction

The instruction `ADCR PC` adds 1 to the program counter whenever the Carry Flag is 1. This amounts to a limited version of the `BC` instruction. This can come up in all sorts of situations.

Take this code from Space Patrol (http://spacepatrol.info/src/bg/bgengine.asm) for example. Here, the function `BGMAKE` returns with the Carry Flag set or clear based on whether there was room for another bad guy. (BG means Bad Guy. `BGMAKE` makes another Bad Guy.) If C==1, that means making the bad guy was successful, so there's additional work to do. If C==0, then there was no room for another bad guy.

```
@@sploop:
        CALL    BGMAKE
        ADCR    PC              ;\_ if carry set, we maybe have room for more
        PULR    PC              ;/
```

Here, the `ADCR PC` instruction skips the `PULR PC` instruction if `BGMAKE` successfully found a slot for a new bad guy.

Using DECR to Spin "Forever"

Sometimes, your program will need to simply "sit and spin" when trying to synchronize with an interrupt, say as part of an initialization sequence. (Interrupts and their uses will be covered in more detail in a future tutorial.) The `DECR PC` instruction decrements the program counter after executing the instruction. This puts the program counter right back at the same instruction, resulting in an infinite loop. The only thing that will break out of this infinite loop is an interrupt.

Here's an example:

```
        ; Prepare to continue initialization after interrupt
        MVII    #INIT,  R0      ;\
        MVO     R0,     $100    ; |_ Set interrupt service vector to point to "INIT"
        SWAP    R0              ; |
        MVO     R0,     $101    ;/

        EIS                     ; Enable interrupts

        DECR    PC              ; Spin until interrupt happens
```

Then, code at `INIT` can continue with initialization: Set up the STIC and GRAM, reset the stack pointer, and so on.

Moving On

At this point, you may wish to move along to the last part, or review earlier parts of this tutorial:

- Introducing the Instruction Set Part 1: The CPU, Memory and Registers; Primary Instructions and Addressing Modes
- Introducing the Instruction Set Part 2: Single Register and Implied Operand Instructions
- Introducing the Instruction Set Part 4: Shift and Rotate Instructions

Or, you can return to the Programming Tutorials index.

Introducing the Instruction Set Part 4

This short tutorial examines the shift and rotate instructions. These are useful instructions, but somewhat more specialized than the rest of the instruction set. If you haven't read Part 1, Part 2 or Part 3 yet, you might consider doing so first.

Contents

Shift and Rotate Instructions

The CP1610 offers a rich variety of shift and rotate instructions. Each can shift by one or two positions. The shift instructions only operate on R0 through R3. You cannot use them with R4 through R7.

Click on a given mnemonic to see how it operates.

SLL Rx[, 2]	Shift Logical Left
SLR Rx[, 2]	Shift Logical Right
SAR Rx[, 2]	Shift Arithmetic Right
SWAP Rx[, 2]	Swap bytes
SLLC Rx[, 2]	SLL into Carry
SARC Rx[, 2]	SAR into Carry
RLC Rx[, 2]	Rotate Left thru Carry
RRC Rx[, 2]	Rotate Right thru Carry

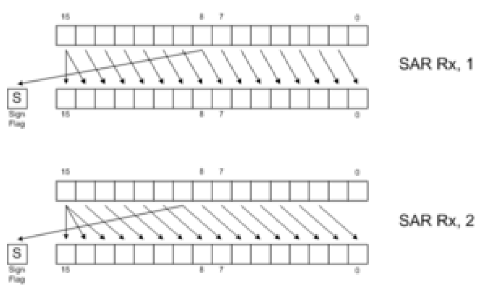
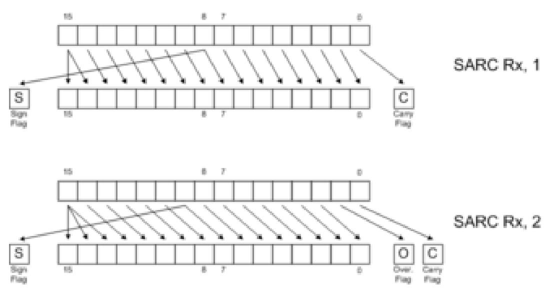
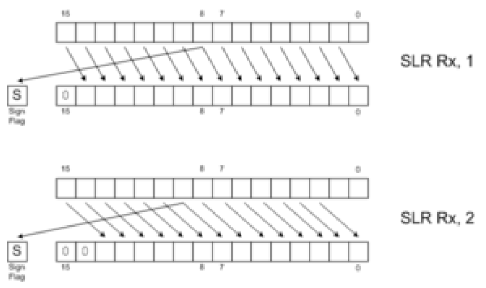
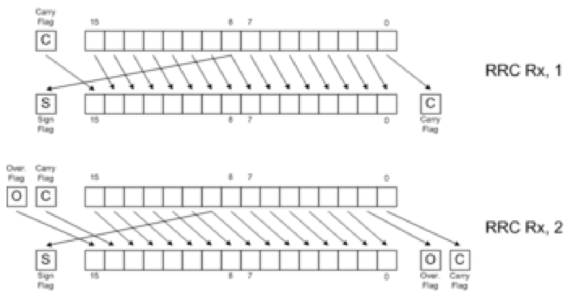
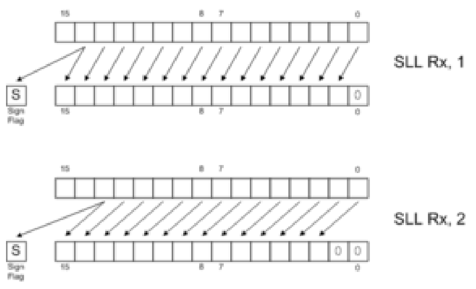
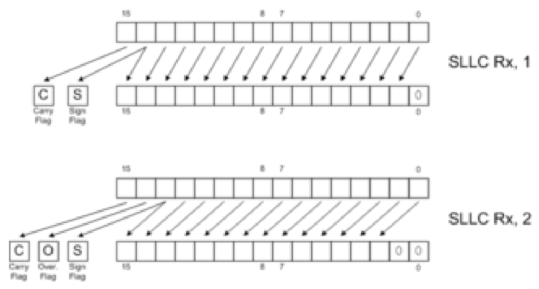
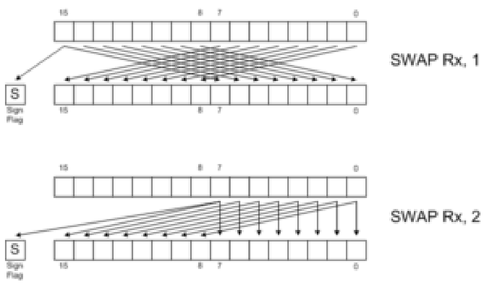
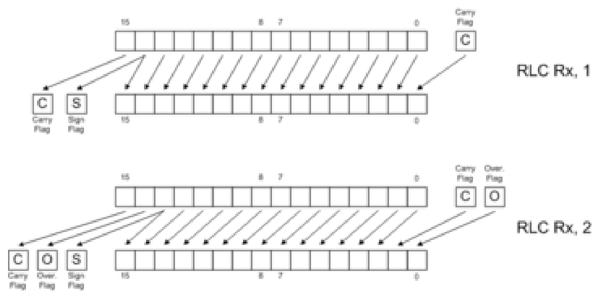
Shifts and rotates are useful for bit manipulation and certain types of mathematics, such as random number generation. Logical shifts (SLL, SLR, SLLC) fill the newly-opened positions with zeros. Arithmetic shifts (SAR, SARC) fill the newly opened positions with copies of bit 15. Logical right shifts can be thought of as "unsigned divide by 2", whereas arithmetic right shifts can be thought of as "signed divide by 2, rounding toward negative."

Shifts can also align numbers with fields packed in a word, or help extract a field in a packed word. For example, the STIC display words in the BACKTAB store the card # to display in bits 2 - 10. To extract a card number, you need to shift right by 3. To insert a card number into a display word, you need to shift left by 3.

Rotates are like shifts, except that they fill the vacated bits with bits from a status register. By itself, this property has no particular mathematical meaning. When rotates combine with shifts, though, they can perform extended-precision operations.

Shift and Rotate Diagram Summary

The following diagrams illustrate the behavior of each of the 8 shift and rotate instructions. Click on the links below for more detail.

Shifts that do *not* use Carry and Overflow	**Shifts and rotates that *do* use Carry and Overflow**
 SAR	 SARC Rx, 1 SARC Rx, 2 S Sign Flag C Carry Flag O Over. Flag 15 8 7 0 SARC
 SLR	 RRC Rx, 1 RRC Rx, 2 Carry Flag C Over. Flag O S Sign Flag 15 8 7 0 RRC
 SLL	 SLLC Rx, 1 SLLC Rx, 2 C Carry Flag O Over. Flag S Sign Flag 15 8 7 0 SLLC
 SWAP Rx, 1 SWAP Rx, 2 S Sign Flag 15 8 7 0 SWAP	 RLC Rx, 1 RLC Rx, 2 Carry Flag C Over. Flag O S Sign Flag 15 8 7 0 RLC

Caveats: Non-interruptibility

Shift instructions are Non-interruptible Instructions. Be sure to include an interruptible instruction in a long sequence of shifts or other non-interruptible instructions, otherwise you could cause display glitches.

A useful rule of thumb is to insert an interruptible instruction between every 4 or so non-interruptible instructions. If you can't find an interruptible instruction to move into place, you can use a NOP instruction.

Using Shifts for Multiplication

Shifting values left multiplies them by powers of two. Each bit position shifted corresponds to another power of two. Shifting left one position multiplies by 2; shifting by two positions multiplies by 4. The following table illustrates the relationship between positions shifted and the power of two. The table stops at 8 for brevity's sake.

Number of positions shifted left	Power of 2 multiplied by	Example Code Sequence
1	2	`SLL Rx, 1`
2	4	`SLL Rx, 2`

3	8	`SLL Rx, 1`
		`SLL Rx, 2`

4	16	`SLL Rx, 2` `SLL Rx, 2`

Number of positions shifted left	Power of 2 multiplied by	Example Code Sequence

5	32	`SLL Rx, 1`
		`SLL Rx, 2`
		`SLL Rx, 2`

6	64	`SLL Rx, 2` `SLL Rx, 2` `SLL Rx, 2`

7	128	`SLL Rx, 1`
		`SLL Rx, 2`
		`SLL Rx, 2`
		`SLL Rx, 2`

8	256	`SWAP Rx, 1`
		`ANDI #$FF00, Rx`

You can combine shift and add instructions to perform simple multiplications by constants that aren't powers of two. The following example shows how to multiply the value in R0 by 20, using R1 as a temporary variable.

```
SLL  R0, 2   ; Multiply R0 by 4
MOVR R0, R1  ; Save a copy of original value * 4
SLL  R0, 2   ; Multiply R0 by 4 again (original value * 16)
ADDR R1, R0  ; Add (value * 4) to (value * 16), giving (value * 20)
```

This works by recognizing that "20" is the sum of two powers of two, "4" and "16". In general, you can multiply by any constant by breaking the constant down into its constituent powers of two.

In some cases, it may be more efficient to use subtracts instead of adds. For example, consider the following example that multiplies R0 by 15, using R1 as a temporary register.

```
MOVR  R0, R1  ; Save a copy of original value
SLL   R0, 2   ; multiply R0 by 4:                   R0 = value * 4
SLL   R0, 2   ; multiply R0 by 4 again:             R0 = value * 16
SUBR  R1, R0  ; Subtract original value from result:  R0 = value * 15
```

Using Shifts for Division by Powers of Two

Right shifts divide by powers of two, by default truncating the fractional portion. This is the same as rounding towards negative. With a minor tweak, the same code can round upwards. The following table shows example code sequences for various power-of-two divides on signed numbers and unsigned numbers.

Number of positions shifted right	Power of 2 divided by	Example Code Sequence for signed, rounding to negative	Example Code Sequence for signed, rounding to positive	Example Code Sequence for unsigned, rounding toward 0	Example Code Sequence for unsigned, rounding to positive
1	2				

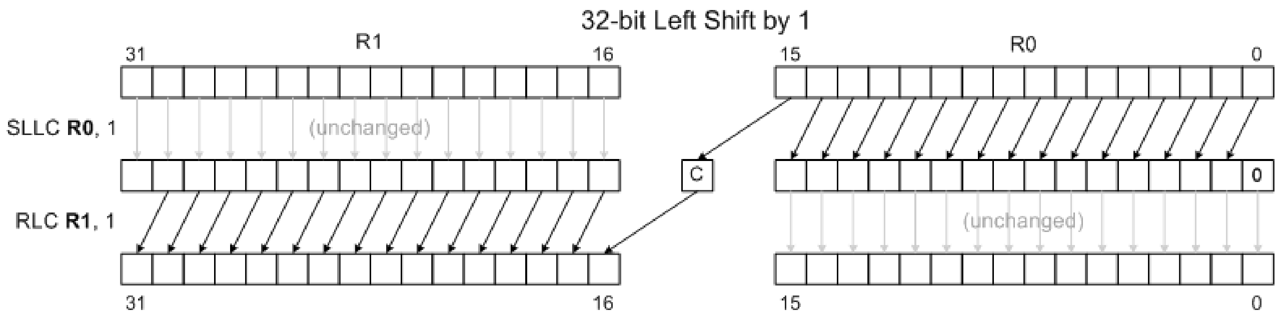
		`SAR  Rx, 1`	`INCR Rx` `SAR  Rx, 1`	`SLR  Rx, 1`	`INCR Rx` `SLR  Rx, 1`
2	4	`SAR  Rx, 2`	`ADDI #2, Rx` `SAR  Rx, 2`	`SLR  Rx, 2`	`ADDI #2, Rx` `SLR  Rx, 2`
3	8	`SAR  Rx, 2` `SAR  Rx, 1`	`SAR  Rx, 2` `INCR Rx` `SAR  Rx, 1`	`SLR  Rx, 2` `SLR  Rx, 1`	`SLR  Rx, 2` `INCR Rx` `SLR  Rx, 1`
4	16	`SAR  Rx, 2` `SAR  Rx, 2`	`SAR  Rx, 2` `ADDI #2, Rx` `SAR  Rx, 2`	`SLR  Rx, 2` `SLR  Rx, 2`	`SLR  Rx, 2` `ADDI #2, Rx` `SLR  Rx, 2`
5	32	`SAR  Rx, 2` `SAR  Rx, 2` `SAR  Rx, 1`	`SAR  Rx, 2` `SAR  Rx, 2` `INCR Rx` `SAR  Rx, 1`	`SLR  Rx, 2` `SLR  Rx, 2` `SLR  Rx, 1`	`SLR  Rx, 2` `SLR  Rx, 2` `INCR Rx` `SLR  Rx, 1`
6	64	`SAR  Rx, 2` `SAR  Rx, 2` `SAR  Rx, 2`	`SAR  Rx, 2` `SAR  Rx, 2` `ADDI #2, Rx` `SAR  Rx, 2`	`SLR  Rx, 2` `SLR  Rx, 2` `SLR  Rx, 2`	`SLR  Rx, 2` `SLR  Rx, 2` `ADDI #2, Rx` `SLR  Rx, 2`
7	128	`SAR  Rx, 2` `SAR  Rx, 2` `SAR  Rx, 2` `SAR  Rx, 1`	`SAR  Rx, 2` `SAR  Rx, 2` `SAR  Rx, 2` `INCR Rx` `SAR  Rx, 1`	`SLR  Rx, 2` `SLR  Rx, 2` `SLR  Rx, 2` `SLR  Rx, 1`	`SLR  Rx, 2` `SLR  Rx, 2` `SLR  Rx, 2` `INCR Rx` `SLR  Rx, 1`
8	256	`SAR  Rx, 2` `SAR  Rx, 2` `SAR  Rx, 2` `SAR  Rx, 2`	`SAR  Rx, 2` `SAR  Rx, 2` `SAR  Rx, 2` `ADDI #2, Rx` `SAR  Rx, 2`	`SWAP Rx` `ANDI #$00FF, Rx`	`ADDI #$80, Rx` `SWAP Rx` `ANDI #$00FF, Rx`

Combining Shifts and Rotates for Extended-Precision Shifts

Rotate instructions combined with shift instructions make it easy to shift values longer than 16 bits. The following examples show how to shift the 32 bit number held in R1:R0 left and right by 1 and 2 positions. (R1 holds bits 16..31 of the 32-bit number.) Note that the diagrams omit the final results of the Carry Flag, Overflow Flag and Sign Flag for clarity and brevity. The rotate instructions do move the shifted-away bits to the C and O flags, though, allowing one to extend the shift to even wider words. See the RLC and RRC pages for more details.

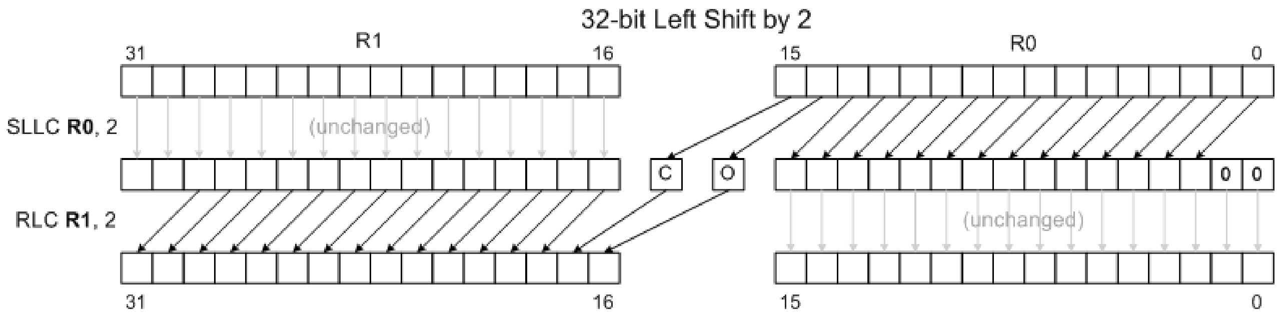
32-bit Left Shift by 1

```
    ; Shift R1:R0 left by 1 position
    SLLC R0, 1   ; Shift lower half left, extra bit to 'C'
    RLC  R1, 1   ; Shift upper half left, pulling lower bit from 'C'
```



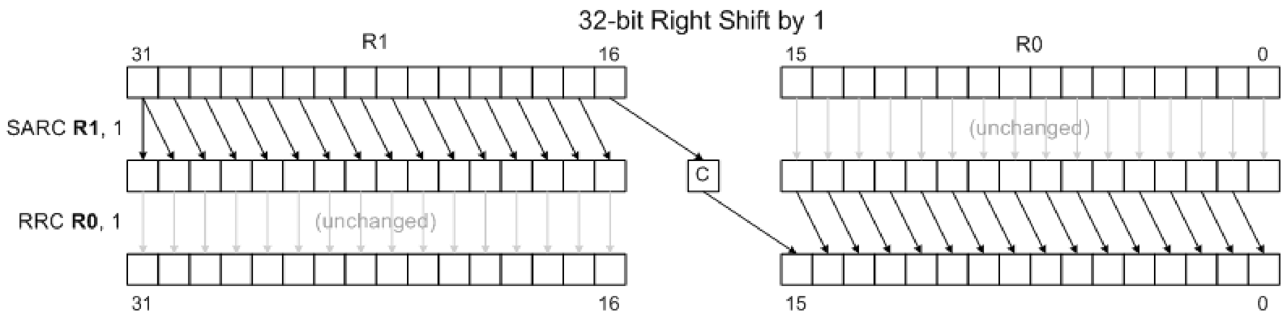
32-bit Left Shift by 2

```
; Shift R1:R0 left by 2 positions
SLLC R0, 2   ; Shift lower half left, extra bits to 'C', 'O'
RLC  R1, 2   ; Shift upper half left, pulling lower bit from 'C', 'O'
```



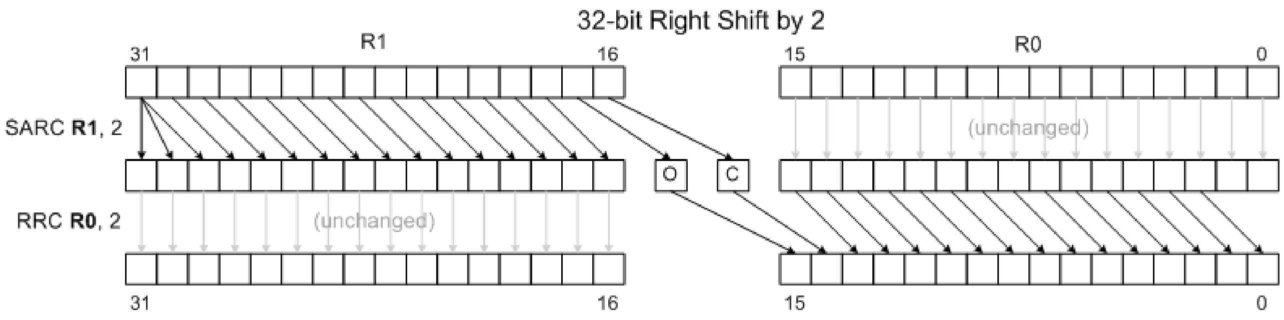
32-bit Right Shift by 1

```
; Shift R1:R0 right by 1 position (arithmetic right shift)
SARC R1, 1   ; Shift upper half left, extra bit to 'C'
RRC  R0, 1   ; Shift lower half left, pulling lower bit from 'C'
```



32-bit Right Shift by 2

```
; Shift R1:R0 left by 2 positions (arithmetic right shift)
SARC R1, 2   ; Shift upper half right, extra bits to 'C', 'O'
RRC  R0, 2   ; Shift lower half right, pulling lower bit from 'C', 'O'
```



Wrapping Up

At this point, you may wish to revisit the earlier parts of this tutorial:

- Introducing the Instruction Set Part 1: The CPU, Memory and Registers; Primary Instructions and Addressing Modes
- Introducing the Instruction Set Part 2: Single Register and Implied Operand Instructions
- Introducing the Instruction Set Part 3: Branches, Conditional Branches and Calls

Or, you can return to the Programming Tutorials index.

Introducing Interrupts

Interrupts are a way for a device in the system to ask the CPU to stop what it's currently doing and do something on its behalf, or to inform the CPU that something happened. In the Intellivision, the STIC interrupts the CPU 60 times a second (NTSC) or 50 times a second (PAL/SECAM) to tell the CPU that it's done drawing the display. This lets the CPU know that the STIC is ready to accept certain commands, and provides a time base for games and programs.

In most systems, interrupts come in two flavors: "maskable interrupts" and "non-maskable interrupts." These sometimes go by the name IRQ and NMI (the common parlance, especially in the 6800/6502 world), or in the case of the CP1610, INTRM and INTR. A maskable interrupt is an interrupt the CPU can choose to ignore until it's convenient. That is, the CPU can "mask" (aka. block or postpone) the interruption. A non-maskable interrupt is an interrupt that forces the CPU to do something immediately.

The Intellivision system only has maskable interrupts, so that's all this tutorial will explore. Other systems often have more elaborate interrupt structures.

Contents

Interrupts and Interruptibility

As mentioned above, interrupt requests ask the CPU to do something other than what it's currently doing when it is convenient to do so. Interrupts are like phone calls. When the phone rings, you can either answer it immediately, answer it after several rings, or you can ignore the phone altogther.

Phone calls interrupt whatever it is you're doing when you answer the call. If you're not busy, it's likely convenient to answer the phone immediately. If you are busy doing something tricky—for example, carrying a bowl of hot soup—you might wait to answer it until you've set it down. If you're away from the phone for awhile—for instance, you're in an important meeting—you might check your voice mail (if you have it) when you get back. At any rate, the phone's unlikely to still be ringing when you do get back.

CPU interrupts are similar, as we'll see in a moment. Programs have situations that are analagous to each of the three scenarios above. The CP1610's notion of "convenient" is called "interruptible." Interruptible means "the CPU will process an interrupt that arrives during this instruction after the instruction completes." The CPU is interruptible if the following two conditions are true:

1. Interrupts are enabled
2. The current instruction is an "interruptible instruction."

Programs can enable (unmask) interrupts with the `EIS`, `JE` and `JSRE` instructions. They can disable (mask) interrupts with `DIS`, `JD` and `JSRD` instructions. These instructions are useful for enabling and disabling interrupts for long periods of time, or over critical groups of instructions. This is a bit like turning the ringer off on your phone when you don't want to be bothered.

Individual instructions in the CP1610 instruction set are also classified as interruptible or non-interruptible. This is a unique feature of CP1610 assembly language. We don't know why General Instrument made specific instructions non-interruptible. Nonetheless, several useful idioms arise out of their selections. The non-interruptible instructions you'll encounter most commonly are writes (`MVO`, `MVO@`, and `PSHR`), shifts/rotates and the various 4 cycle instructions such as `EIS`, `DIS` and `SDBD`. Non-interruptible instructions are akin to ignoring your phone's ringer for a moment while you get something else out of the way.

What Happens When the CPU Takes an Interrupt?

When the Intellivision responds to an interrupt, it takes the following steps:

1. The CPU writes the current value of the program counter (R7) to the stack. This value indicates the instruction that will run when we return from the interrupt.

2. The CPU then jumps to location $1004. The hardware sets this address, and there's no easy way to change this address on the Intellivision.
3. The EXEC interrupt code at $1004 then saves the CPU's state on the stack. The code pushes R0 through R5 and a copy of the flags register onto the stack. Altogether, interrupts take up a minimum of 8 words of stack space.
4. The EXEC interrupt code sets up a return address ($1014) in R5, and then jumps to the interrupt handler whose address is stored in RAM at locations $100-$101.

The sequence of events above is the same for all Intellivisions. The Intellivision I, Intellivision II, Sears Super Video Arcade, Tandyvision One, INTV System III and INTV Super Pro System all perform the same steps. (It *is* possible to modify the Intellivision to do something different, but why?) It makes sense then to understand this sequence exactly as it is described above.

So What are Interrupt Handlers?

Interrupt handlers (also known as Interrupt Service Routines or ISRs) are functions that get invoked by the EXEC in response to an interrupt. In the Intellivision, these functions handle tasks that must be attended to periodically, since the Intellivision's STIC triggers an interrupt 60 times a second (50 times a second on PAL/SECAM systems).

ISRs have many duties. During normal game play, an ISR might do the following:

1. Update STIC registers
2. Update the GRAM
3. Update any music and/or sound effects
4. Update any timers or counters that keep track of time
5. Update object positions based on their "velocity"
6. Check the hand controllers for new inputs

This list isn't perfect, but it is representative. The exact demands placed on an ISR depend on the nature of the game program and what that program needs at the time.

The first step is probably the most important one. By default, the STIC will blank the display unless the CPU specifically asks it to display the next frame. This is referred to as the "STIC handshake." To keep the display active (non-blank), programs must write to location $0020 within about 2000 cycles of the STIC generating an interrupt. If the CPU does not write to this location, the STIC will blank the display during the next frame.

The remaining steps really depend on how your game is structured. Future tutorials will cover these needs in more depth.

Setting up an Interrupt Handler

Programs can set up an interrupt handler by writing its address to locations $100 and $101. The address is stored in Double Byte Data format. The following code shows one way of setting up "MYISR" as the interrupt handler:

```
MVII    #MYISR, R0
MVO     R0,     $100    ; write out lower 8 bits of ISR address
```

```
SWAP    R0              ; swap lower/upper halves
MVO     R0,     $101    ; write out upper 8 bits of ISR address
```

There are other ways to do this, but this is adequate for most purposes. This sequence is also *non-interruptible,*, which is important, as we will see below.

Interrupt handlers behave just like any other function. The EXEC sets up R5 to point to the return address, $1014. Since the return address is always $1014, some interrupt handlers (such as those in Space Patrol (http://spacepatrol.info/)) simply branch directly to $1014 rather than keeping the value of R5 around.

Reasons for Turning Off Interrupts

Normally, programs leave interrupts enabled, and don't pay much attention to which instructions are interruptible. They let interrupts happen when they happen. Sometimes, though, programs have good reason to turn off interrupts.

Critical Sections

To the program, interrupts look like they occur *between* two other instructions. The CPU stops doing what it's doing, processes the interrupt, and then resumes what it had been doing. Most of the time, this is ok: What the interrupt code does has little to do with what the interrupted code is doing. That isn't always true though.

Suppose, for example, you're trying to increment a variable. This is a three step process:

```
MVI     VAR,    R0
INCR    R0
MVO     R0,     VAR
```

Suppose also that an interrupt handler tries to decrement this same variable:

```
MVI     VAR,    R1
DECR    R1
MVO     R1,     VAR
```

This can lead to disaster. Imagine if an interrupt happens in the middle of the first instruction sequence. The CPU ends up executing the following instruction sequence with respect to `VAR`:

```
MVI     VAR,    R0
INCR    R0
; ---> interrupt occurs
    MVI     VAR,    R1
    DECR    R1
    MVO     R1,     VAR
; <--- return from interrupt
MVO     R0,     VAR
```

What value gets left in `VAR`? In this case, it looks like the decrement never happened. That's because the `MVO R0, VAR` writes the incremented value after `MVO R1, VAR` wrote the decremented value. Also, the decremented value isn't even the right value.

This type of code is referred to as a "critical section." The `MVI`/`INCR`/`MVO` sequence is critical because it needs to execute without interruption. Using the phone call analogy above, this is similar to carrying a hot bowl of soup: You don't want to spill the soup, and carrying it requires your full attention, so you want to do so without interruption until you safely set down the soup.

To fix the example above, you can wrap the critical code with `EIS` and `DIS` instructions:

```
    DIS
    MVI     VAR,    R0
    INCR    R0
    MVO     R0,     VAR
    EIS
```

What this does is prevent the CPU from servicing an interrupt while executing this code. Now the interrupt handler and the main program can increment and decrement and the right thing always happens.

A Useful Idiom: MVO/SWAP/MVO

Because some of CP1610's instructions are non-interruptible, certain very useful idioms are non-interruptible. Perhaps the most important idiom is the one for writing Double Byte Data. The following sequence is not interruptible, because MVO@ and SWAP are not interruptible:

```
    MVO@    R0,     R4      ; write lower 8 bits
    SWAP    R0
    MVO@    R0,     R4      ; write upper 8 bits
```

This idiom is very useful for setting up the interrupt handler address. If the sequence *were* interruptible, programs would have to manually disable interrupts while setting up a new interrupt handler address. Otherwise, if an interrupt happened before the address was completely set up, the program could crash.

For example, suppose $100/$101 point to an interrupt handler at location $5678. This means location $100 holds $78, and location $101 holds $56. (See Double Byte Data if you're unsure why.) Now we're setting up a new interrupt handler "ISR2" that happens to be at location $FEDC. The code for this might look like so:

```
    MVII    #ISR2,  R0
    MVO     R0,     $100    ; Write out lower 8 bits ($DC in this example)
    SWAP    R0
    MVO     R0,     $101    ; Write out upper 8 bits ($FE in this example)
```

Next, suppose the `MVO/SWAP/MVO` sequence *is*interruptible. If an interrupt occurs after the first `MVO`but before the second `MVO`, what would happen?

The answer is simple: The EXEC would branch to whatever code or data is at location $56DC. Who knows what's at this address? This address is a meaningless mixture of $5678 (the old ISR) and

$FEDC (the new ISR). The most likely outcome is that the program crashes mysteriously.

Fortunately, because `MVO` and `SWAP` are both non-interruptible on the CP1610, this isn't possible. The code *does*safely update the interrupt handler address. Any interrupt that arrives during this sequence will be ignored until after the sequence completes.

Program Initialization

Sometimes, programs need to turn off (or simply find it convenient to turn off) interrupts for extended period of time. This happens most often during initialization. (By initialization, I mean any major change in machine and game setup, not just the setup that happens right after reset.)

During initialization, the program may be setting up large bits of memory, such as the contents of GRAM and so on. This may take more than the time the STIC usually allots for this. Thus, such code may disable interrupts to simplify program design and prevent misbehavior.

This is analagous to being "away from the phone" in the metaphor above, only there is no "voice mail." If the CPU doesn't respond to an interrupt in about 1600 cycles, the interrupt simply gets dropped. This is like what happens when someone hangs up after letting the phone ring for awhile. The *next*interrupt still occurs though.

Example 1: A simple "elapsed time" clock

The following example attempts to demonstrate how interrupts work without getting caught up in too many details of how the rest of the system works. This example shows a simple elapsed-time clock on screen that shows how many hours, minutes, seconds and 60ths of a second (tics) have elapsed since the program started. It uses an interrupt handler to keep track of the time and to keep the screen from going blank.

Let's build this up by pieces.

The Interrupt Handler

The interrupt handler has two major responsibilities: Keep the screen on, and update the time. Updating the time has several smaller steps:

1. Increment the number of tics that have occurred this second. If that number is less than 60, then we're done updating the time so skip to the end.
2. Zero out the # of tics/second and increment the number of seconds. If this number is less than 60, then we're done updating the time, so skip to the end.
3. Zero out the # of seconds and increment the number of minutes. If this number is less than 60, then we're done updating the time, so skip to the end.
4. Zero out the number of minutes and increment the number of hours.

The following code shows one way to implement the ISR. The first statement,`MVO R0, $20` keeps the screen enabled. The rest of the code updates the elapsed time. There *are* more clever ways to do this. This code is written for clarity. (In general, write for clarity first, and get fancy later only if you need to.)

```
MYISR   PROC

        MVO     R0,     $20     ; Keep screen enabled.

        MVI     TIC,    R0      ; Get current tic count into R0
        MVI     SEC,    R1      ; Get current seconds count into R1
        MVI     MIN,    R2      ; Get current minutes count into R2
        MVI     HOUR,   R3      ; Get current hours count into R3

        INCR    R0              ; Increment tic count
        CMPI    #60,    R0      ; Tic < 60?
        BLT     @@time_done     ; Yes:  Done updating time

        CLRR    R0              ; Reset tic count
        INCR    R1              ; Increment seconds count
        CMPI    #60,    R1      ; Seconds < 60?
        BLT     @@time_done     ; Yes:  Done updating time

        CLRR    R1              ; Reset seconds count
        INCR    R2              ; Increment minutes count
        CMPI    #60,    R2      ; Minutes < 60?
        BLT     @@time_done     ; Yes:  Done updating time

        CLRR    R2              ; Reset minutes count
        INCR    R3              ; Increment hours count

@@time_done:
        MVO     R0,     TIC     ; Store updated tic count
        MVO     R1,     SEC     ; Store updated seconds count
        MVO     R2,     MIN     ; Store updated minutes count
        MVO     R3,     HOUR    ; Store updated hours count

        JR      R5
        ENDP
```

This interrupt handler is fairly straightforward. It's all that this particular example needs.

The Main Loop

The main loop merely needs to print out the time whenever it changes. The loop runs "forever." The main steps are:

1. Wait for the time to change. We can do this by waiting for the value of `TIC` to change.
2. Take a snapshot of the current time. While it might not be immediately obvious, this is a critical section. We don't want the time to change on us while we're reading it.
3. Print out the various time components: Hours, Minutes, Seconds and Tics.
4. Go back to step 1.

Let's take a look of each of these steps in turn.

Step 1: Wait for the time to change

We can tell if the time has changed by looking at the `TIC` variable and seeing if its value is different than the last time we displayed the time. This works because the value of `TIC` changes every time the time changes, even if the other time components don't change.

The safest and easiest way to do this is to store a copy of the previous value of `TIC` somewhere. In this example, we'll store it in a place called `PTIC`. The code to do this might look like so:

```
@@wait_time:
        MVI     TIC,    R0      ; Get current value of TIC
        CMP     PTIC,   R0      ; Has the time changed?
        BEQ     @@wait_time     ; No:  Keep looping
```

When the loop above exits, we know the time has changed since the last time we displayed it.

Step 2: Take a snapshot of the time

Because the interrupt handler updates the time inside the interrupt handler, we need to be careful when we read the time so that we get a consistent view of the time. Otherwise, if an interrupt occurs while we're reading the time, we could get a scrambled copy. For example, suppose the current time is 11:59:59:59 and is about to roll over to 12:00:00:00. If an interrupt comes in after we've read tics, seconds and minutes but before we read the hours, we might see 12:59:59:59, which is completely wrong. Even worse, on our next tic we will likely see 12:00:00:01.

If this did happen, time will appear to have jumped forward and then backward by very large amouints, which is bad. We want to see either 11:59:59:59 or 12:00:00:00 in this case, not some mixture of the two. If we were doing much more with the time than just displaying it, such an error could cause all sorts of strange problems.

Now, in *this* particular example program, such an error is unlikely to happen and if it did happen, the consequences are minor. In more complex programs, though, it could happen that by the time we get around to displaying the time again, the time is about to change. Therefore, rather than write risky code that happens to work for this example, it's better to write correct code that will continue to work even if it gets reused in a much more complex program.

The most straightforward way to read the time, then, is to disable the interrupts and read the time with interrupts off. This copy of the time is guaranteed to be consistent no matter what. For the purposes of this example, we'll read the time and put a copy of it on the stack with `PSHR`.

```
        DIS                     ; Disable ints (begin critical section)
        MVI     TIC,    R0      ; Read tic count
        MVO     R0,     PTIC    ; Remember TIC in PTIC for next time
        PSHR    R0              ; Save tic count on the stack
        MVI     SEC,    R0      ; Read seconds count
        PSHR    R0              ; Save seconds count on the stack
        MVI     MIN,    R0      ; Read minutes count
        PSHR    R0              ; Save minutes count on the stack
        MVI     HOUR,   R0      ; Read hours count
        EIS                     ; Enable ints (end critical section)
```

Notice that we re-read `TIC` and copy its value to `PTIC` inside this critical section. This is perhaps overkill, in that it's highly unlikely that the current time will change between when the `@@wait_time` loop completes and when we snapshot the time. It would require our code to be running way behind compared to the 60Hz (or 50Hz) interrupt clock. That said, the above way of writing the code ensures that we only start a new loop when the current time is different than the displayed time, no matter what, and the displayed time is always 100% consistent.

Step 3: Displaying the time

To display the time, we'll use the SDK-1600 (http://sdk-1600.spatula-city.org/) routine `PRNUM16`, found here. The`PRNUM16.z` routine prints the number in R0 in a fixed width field, complete with leading zeroes. It prints the value to the screen, starting at the display location specified in R4. It returns with R4 pointing to the first position after the displayed number. This is exactly what we want.

The `PRNUM16` function describes its inputs and outputs as follows (edited slightly for clarity):

```
INPUTS
    R0  Number to print.
    R2  Width of field.
    R3  Format word, added to digits to set the color.
    R4  Pointer to location on screen to print number

OUTPUTS
    R0  Zeroed
    R1  Unmodified
    R2  Unmodified
    R3  Unmodified
    R4  Points to first character after field.
```

This is perfect for our needs. The example program will display the time as four 2 digit fields separated by an empty space. To keep the example simple, we won't cover just yet how to display colons between the time segments. Later tutorials on the STIC will cover this.

The time string will take up 8 characters for the time digits, and then another 3 characters for the spaces between them. This makes the time 11 characters long. To display this approximately centered in the screen, we can display it in Row 5, Column 5. Recall that the BACKTAB is a 20 x 12 character grid. This means that the first character will be at location `$0200 + 5*20 + 5`. (See the Hello World Tutorialfor a refresher on how this calculation works.)

The display format word controls what color the number appears in, and in fancier cases, what font. Such details are beyond the scope of this tutorial, but rest assured we'll cover them eventually. For now, we'll display the time in white. White is color #7, and so our format word will be 7.

With that in mind, the following code shows how to display the time. Recall that when we start this code, R0 already holds a copy of `HOUR`and the stack holds `MIN`, `SEC` and `TIC`.

```
        MVII    #2,     R2      ; Set our field width to 2
        MVII    #7,     R3      ; Set our format word to "white"
        MVII    #$200 + 5*20 + 5, R4    ; Start in row 5, column 5

        CALL    PRNUM16.z       ; Display the hours
        INCR    R4              ; Leave a blank space after hours

        PULR    R0              ; Get minutes
        CALL    PRNUM16.z       ; Display the minutes
        INCR    R4              ; Leave a blank space after minutes

        PULR    R0              ; Get seconds
        CALL    PRNUM16.z       ; Display the seconds
        INCR    R4              ; Leave a blank space after seconds
        PULR    R0              ; Get tics
        CALL    PRNUM16.z       ; Display the tics
```

Notice how `PULR` gets used to recall the previously saved values of `HOUR`, `MIN` and `SEC`. The`PULR` instructions mirror the `PSHR` instructions we used earlier to save these values. With the stack, every thing that gets pushed on the stack needs to be pulled off later. Furthermore, what gets pushed onto the stack gets pulled off in reverse order. In this case, we pushed the snapshot of `TIC` followed by the snapshot of `SEC` followed by the snapshot of `MIN`. When we displayed the time, these came off the stack in reverse order—minutes, seconds and then tics.

Step 4: Go back to step 1

This is perhaps the easiest step. Since our code has nothing better to do than display the time, all it needs to do is jump back to the top of the loop. To do this, we can put a label before step 1, and branch to it. The resulting loop looks like, from end to end:

```
@@outer_loop:                   ; outermost loop

        ;; Step 1:  Wait for time to change
@@wait_time:
        MVI     TIC,    R0      ; Get current value of TIC
        CMP     PTIC,   R0      ; Has the time changed?
        BEQ     @@wait_time     ; No:  Keep looping

        ;; Step 2:  Snapshot the time
        DIS                     ; Disable ints (begin critical section)
        MVI     TIC,    R0      ; Read tic count
        MVO     R0,     PTIC    ; Remember TIC in PTIC for next time
        PSHR    R0              ; Save tic count on the stack
        MVI     SEC,    R0      ; Read seconds count
        PSHR    R0              ; Save seconds count on the stack
        MVI     MIN,    R0      ; Read minutes count
        PSHR    R0              ; Save minutes count on the stack
        MVI     HOUR,   R0      ; Read hours count
        EIS                     ; Enable ints (end critical section)

        ;; Step 3:  Display the updated time
        MVII    #2,     R2      ; Set our field width to 2
        MVII    #7,     R3      ; Set our format word to "white"
        MVII    #$200 + 5*20 + 5, R4    ; Start in row 5, column 5

        CALL    PRNUM16.z       ; Display the hours
        INCR    R4              ; Leave a blank space after hours

        PULR    R0              ; Get minutes
        CALL    PRNUM16.z       ; Display the minutes
        INCR    R4              ; Leave a blank space after minutes

        PULR    R0              ; Get seconds
        CALL    PRNUM16.z       ; Display the seconds
        INCR    R4              ; Leave a blank space after seconds

        PULR    R0              ; Get tics
        CALL    PRNUM16.z       ; Display the tics

        ;; Step 4:  Go back to step 1.
        B       @@outer_loop
```

Now, you might wonder why we added a second label, `@@outer_loop`, when there's no code between `@@outer_loop` and `@@wait_time`. After all, it's not strictly necessary. The answer is pragmatism.

While it may be true at the moment that the additional label is redundant, that may not be true forever. At some point, we might want to add code that runs before we wait for the time sync-up. That code will need to go between the two labels. After all, displaying the updated time might be the *last*thing we do, as opposed to the *only* thing we do.

Also, it's simply clearer. Our last step is to "do everything again." This has different meaning than "wait for the time to change." Descriptive labels make it clearer to the reader what the purpose of a given branch is. Labels are free. Don't be afraid to add labels if it makes the code clearer.

Setting Things Up

So far, we've looked at the meat of the problem—the interrupt handler and the time display—but we've skipped entirely over the problem of setting things up. Most of that is pretty easy. Four steps remain: We must find a place to put the time, set the initial time to zero, clear the screen, and make our ISR the current interrupt handler.

Allocating Our Variables

You might be thinking "We already have a place to put the time!" Well, yes and no. In the fragments of code above, I've used the names `PTIC`, `TIC`, `SEC`, `MIN` and `HOUR` to refer to the various variables this program needs. Those variables do indeed hold the current time, as well as the previous value of `TIC`. I haven't told the assembler where to put these though. If you were to try to assemble the code above, you'd get errors for this reason. One way to rectify this is to assign addresses to each of these names with the`EQU` directive.

Each of these quantities fits in 8 bits, so it's natural to allocate these values in 8 bit Scratchpad RAM. This memory resides at addresses $0100 - $01EF in the Intellivision memory map. Locations $0100 - $0101 hold the interrupt handler address. The remaining locations are available for our use when we avoid using the EXEC. (The EXEC uses some of these locations for its internal bookkeeping.)

In this example, we do not use the bulk of the EXEC, and so we have the entire Scratchpad available to us. Therefore, the following code snippet works just fine. (There are better ways to do this in large programs. We will look at those later.)

```
PTIC    EQU     $102    ; Previous value of 'tic'
TIC     EQU     $103    ; Current number of tics (0..59)
SEC     EQU     $104    ; Current number of seconds (0..59)
MIN     EQU     $105    ; Current number of minutes (0..59)
HOUR    EQU     $106    ; Current number of hours (0..59)
```

There are many ways we might initialize these values. The most straightforward is just to zero out all of the Scratchpad RAM, except for locations $100 and $101. The SDK-1600 routine `FILLZERO` `(http://sdk-1600.spatula-city.org/examples/library/fillmem.asm)` is perfect for this task:

```
        MVII    #$102,  R4      ; Start filling at location $102
        MVII    #$0EE,  R1      ; Fill $EE locations (up through $1EF)
        CALL    FILLZERO
```

This will clear out all of the Scratchpad memory, and thus set all five of our variables—`PTIC`, `TIC`, `SEC`, `MIN`, and `HOUR`—to zero. Hold that thought a moment, because we'll come back to it.

Clearing the Screen

The SDK-1600 function `CLRSCR` does a fine job of clearing the screen. This function resides in the same file as `FILLZERO`and `FILLMEM`. This is because the three functions are very closely related. This makes it tempting to think a little harder about what exactly we're doing.

The straightforward next step in program initialization would be to `CALL CLRSCR` after clearing out the Scratchpad RAM. The screen resides at $0200 - $02EF, and this fills these locations with 0. If you recall, the code above just got done filling locations $0102 - $01EF with 0 as well. It seems like we should be able to combine the two acts.

It turns out we can. Addresses $01F0 - $01FF refer to the Programmable Sound Generator (aka. PSG). We can safely write 0s to all of its locations. In fact, this is the preferred way to initialize the PSG when its state is otherwise unknown. Thus, we can combine the `FILLZERO` and `CLRSCR` calls into a single `FILLZERO` call that zeros $0102 - $02EF. The combined code looks like so:

```
        MVII    #$102,  R4      ; Start filling at location $102
        MVII    #$1EE,  R1      ; Fill $1EE locations (up through $2EF)
        CALL    FILLZERO
```

This kills two birds with one stone. (Three birds if you count initializing the PSG, although in this example it happens that the EXEC has done that for us already.)

Setting up the Interrupt Handler

Finally, all that's left is setting up the interrupt handler to point to the time counting code we wrote above. Recall that our routine is called `MYISR`. The following code sets that up:

```
        MVII    #MYISR, R0      ; \
        MVO     R0,     $100    ;  |_ Write out "MYISR" to $100-$101
        SWAP    R0              ;  |  Double Byte Data.
        MVO     R0,     $101    ; /
        EIS                     ; Make sure interrupts are enabled
```

Putting it All Together

No Intellivision program is complete without an appropriate ROM header and INCLUDE directives for all of the library functions. This particular example needs `fillmem.asm` and `prnum16.asm` from SDK-1600. Download these files and put them in a directory. Then put the following source code in a new file named `elapsed.asm`in the same directory. This source code contains all of the snippets from above, along with a ROM header to make it work. The added portions are in bold. As you can see, the additions are minor and are largely boilerplate you've seen before from the Hello World Tutorial.

```
PTIC    EQU     $102    ; Previous value of 'tic'
TIC     EQU     $103    ; Current number of tics (0..59)
SEC     EQU     $104    ; Current number of seconds (0..59)
MIN     EQU     $105    ; Current number of minutes (0..59)
HOUR    EQU     $106    ; Current number of hours


        ROMW    16      ; 16-bit ROM
        ORG     $5000   ; Standard ROM memory map starts at $5000

;------------------------------------------------------------------------------
; EXEC-friendly ROM header.
;------------------------------------------------------------------------------
ROMHDR: BIDECLE ZERO            ; MOB picture base   (points to NULL list)
        BIDECLE ZERO            ; Process table      (points to NULL list)
        BIDECLE MAIN            ; Program start address
        BIDECLE ZERO            ; Bkgnd picture base (points to NULL list)
        BIDECLE ONES            ; GRAM pictures      (points to NULL list)
        BIDECLE TITLE           ; Cartridge title/date
        DECLE   $03C0           ; Flags:  No ECS title, run code after title,
                                ; ... no clicks
ZERO:   DECLE   $0000           ; Screen border control
        DECLE   $0000           ; 0 = color stack, 1 = f/b mode
ONES:   DECLE   1, 1, 1, 1, 1   ; Color stack initialization
;------------------------------------------------------------------------------

TITLE   STRING  $107, "Elapsed Time Demo", 0  ; Title string and date (2007)

MAIN    PROC

        MVII    #$102,  R4      ; Start filling at location $102
        MVII    #$1EE,  R1      ; Fill $1EE locations (up through $2EF)
        CALL    FILLZERO

        MVII    #MYISR, R0      ; \
        MVO     R0,     $100    ;  |_ Write out "MYISR" to $100-$101
        SWAP    R0              ;  |  Double Byte Data.
        MVO     R0,     $101    ; /
        EIS                     ; Make sure interrupts are enabled

@@outer_loop:                       ; outermost loop

        ;; Step 1:  Wait for time to change
@@wait_time:
        MVI     TIC,    R0      ; Get current value of TIC
        CMP     PTIC,   R0      ; Has the time changed?
        BEQ     @@wait_time     ; No:  Keep looping

        ;; Step 2:  Snapshot the time
        DIS                     ; Disable ints (begin critical section)
        MVI     TIC,    R0      ; Read tic count
        MVO     R0,     PTIC    ; Remember TIC in PTIC for next time
        PSHR    R0              ; Save tic count on the stack
        MVI     SEC,    R0      ; Read seconds count
        PSHR    R0              ; Save seconds count on the stack
        MVI     MIN,    R0      ; Read minutes count
        PSHR    R0              ; Save minutes count on the stack
        MVI     HOUR,   R0      ; Read hours count
        EIS                     ; Enable ints (end critical section)

        ;; Step 3:  Display the updated time
        MVII    #2,     R2      ; Set our field width to 2
        MVII    #7,     R3      ; Set our format word to "white"
        MVII    #$200 + 5*20 + 5, R4    ; Start in row 5, column 5

        CALL    PRNUM16.z       ; Display the hours
        INCR    R4              ; Leave a blank space after hours

        PULR    R0              ; Get minutes
        CALL    PRNUM16.z       ; Display the minutes
        INCR    R4              ; Leave a blank space after minutes
        PULR    R0              ; Get seconds
        CALL    PRNUM16.z       ; Display the seconds
        INCR    R4              ; Leave a blank space after seconds

        PULR    R0              ; Get tics
        CALL    PRNUM16.z       ; Display the tics

        ;; Step 4:  Go back to step 1.
        B       @@outer_loop

        ENDP

MYISR   PROC

        MVO     R0,     $20     ; Keep screen enabled.

        MVI     TIC,    R0      ; Get current tic count into R0
        MVI     SEC,    R1      ; Get current seconds count into R1
        MVI     MIN,    R2      ; Get current minutes count into R2
        MVI     HOUR,   R3      ; Get current hours count into R3

        INCR    R0              ; Increment tic count
        CMPI    #60,    R0      ; Tic < 60?
        BLT     @@time_done     ; Yes:  Done updating time

        CLRR    R0              ; Reset tic count
        INCR    R1              ; Increment seconds count
        CMPI    #60,    R1      ; Seconds < 60?
        BLT     @@time_done     ; Yes:  Done updating time

        CLRR    R1              ; Reset seconds count
        INCR    R2              ; Increment minutes count
        CMPI    #60,    R2      ; Minutes < 60?
        BLT     @@time_done     ; Yes:  Done updating time

        CLRR    R2              ; Reset minutes count
        INCR    R3              ; Increment hours count

@@time_done:
        MVO     R0,     TIC     ; Store updated tic count
        MVO     R1,     SEC     ; Store updated seconds count
        MVO     R2,     MIN     ; Store updated minutes count
        MVO     R3,     HOUR    ; Store updated hours count

        JR      R5
        ENDP

;; ======================================================================
;;  Include SDK-1600 library functions
;; ======================================================================
        INCLUDE     "prnum16.asm"
        INCLUDE     "fillmem.asm"
```

To assemble the code above, type:

```
  as1600 -o elapsed -l elapsed.lst elapsed.asm
```

And that's it.

The cartridge header tells the EXEC to minimize its initialization work and simply run the code that appears immediately after the title string. That code is our initialization code. Once that code runs, it drops into the main program loop.

In the background, the interrupt handler updates the elapsed time with every interrupt from the STIC. This causes the `@@wait_time` loop in the main program to exit, and the rest of the code to update the displayed time. Wash, rinse, repeat.

Further Things to Contemplate About the Elapsed Time Demo

This timer is a pretty simple piece of code. It's also specific to NTSC systems, insofar as its notion of hours, minutes and seconds are built around 60 tics/second. How would you change the code to keep time correctly on a PAL/SECAM system that generates 50 interrupts per second? How might you detect whether a given system is NTSC or PAL/SECAM at run time?

The timer currently does not display colons between the various time components. Where might you insert code to do this? Is that the best place for this code? Why or why not?

The "tics" display goes by very quickly, so quickly that it's mostly a blur. Typically, humans aren't very concerned about time displays with resolution smaller than a second, except maybe on stopwatches. How would you change this code to only update once a second?

What happens when this program runs for more than 99 hours? More than 256 hours? (That is, other than your Intellivision overheating . . . .) With that in mind, how would you add a days indicator to the program?

The background color behind the timer display is brown. Why is this? How might you change this?

Example 2: A simple "wait timer"

The purpose of this example is to demonstrate how to introduces pauses in a program. The "wait timer" consists of two parts: A short bit of code in the ISR that decrements a count if it's non-zero, and a separate function that sets the count and waits for it to become zero.

The wait-timer serves two purposes:

- Long waits can space events out. This can be useful if you want to display a message for a moment or otherwise choreograph events.
- Short waits (such as 1 tic) are useful for waiting to ensure that the screen has updated since changing something that the interrupt handler might have acted on. That is, it's a way of figuring out that an interrupt has occurred.

We'll see an example of the first way of using the wait timer in this example. Future tutorials will reuse this mechanism for both purposes, which is the main reason I'm introducing it here.

The Interrupt Handler Code

The wait timer code can be inserted into any interrupt handler. The code for the wait timer is very simple. It decrements the value of the variable`WTIMER` if it's non-zero, as shown below:

```
        MVI     WTIMER, R0      ; Get current timer count
        DECR    R0              ; Decrement it
        BMI     @@expired       ; If it went negative, it's expired
        MVO     R0,     WTIMER  ; Store updated count
@@expired:
```

This code can be inserted into any ISR easily. Alternately, you can set up many ISRs to call this at the end of their run before exiting. That's entirely up to how you want your program to work. In this example, we'll have a simple ISR that enables the screen, counts down the wait timer, and then exits. The entire ISR is shown below:

```
;; ======================================================================== ;;
;;  MYISR -- A simple interrupt service routine                             ;;
;; ======================================================================== ;;
MYISR   PROC
        MVO     R0,     $20     ; Enable the display

        MVI     WTIMER, R0      ; Get current timer count
        DECR    R0              ; Decrement it
        BMI     @@expired       ; If it went negative, it's expired
        MVO     R0,     WTIMER  ; Store updated count
@@expired:

        JR      R5              ; return from interrupt
        ENDP
```

A more complicated program would clearly have more stuff in its interrupt handler. We'll see this in future tutorials.

The Wait Code

Programs will call the following function to actually wait for something to happen. This function has two entry points. `WAIT` will wait for the number of tics specified in the word after the `CALL`. (See the branch tutorial for an explanation of this technique.) The second entry point expects the number of tics to wait to be in R0.

```
;; ======================================================================== ;;
;;  WAIT -- Wait for some number of tics                                    ;;
;;                                                                          ;;
;;  INPUTS for WAIT                                                         ;;
;;      1 decle after call:  Number of tics to wait                         ;;
;;                                                                          ;;
;;  INPUTS for WAIT.1                                                       ;;
;;      R0  Number of tics to wait                                          ;;
;;                                                                          ;;
;;  OUTPUTS                                                                 ;;
;;      R0  Zeroed                                                          ;;
;;                                                                          ;;
;;  NOTE                                                                    ;;
;;      Requires WTIMER code in ISR, and interrupts enabled.                ;;
;;                                                                          ;;
;; ======================================================================== ;;
WAIT    PROC
        MVI@    R5,     R0          ; Get # of tics to wait from after CALL
@@1:    MVO     R0,     WTIMER      ; Set up wait timer

        CLRR    R0                  ; \
@@wait: CMP     WTIMER, R0          ;  |- Wait for WTIMER = 0
        BNEQ    @@wait              ; /

        JR      R5                  ; return
        ENDP
```

The Rest

For this example, we'll display "Hello" and "World" about one second apart in as we did previously in the Hello World Tutorial. We'll also use the an infinite loop. We'll reuse SDK-1600's (http://sdk-1600.spatula-city.org/) PRINT and CLRSCR functions. This results in the following code:

```
MAIN    PROC

        CALL    CLRSCR              ; Clear the screen

        MVII    #MYISR, R0          ; \
        MVO     R0,     $100        ;  |_ Write out "MYISR" to $100-$101
        SWAP    R0                  ;  |  Double Byte Data.
        MVO     R0,     $101        ; /
        EIS                         ; Make sure interrupts are enabled

@@loop:
        ;; Print "Hello" at row #5, column #7.
        CALL    PRINT.fls
        DECLE   7, $200 + 5*20 + 7
        STRING  "Hello", 0

        ;; Wait for 1 second
        CALL    WAIT
        DECLE   60

        ;; Print "World" at row #5, column #7.
        CALL    PRINT.fls
        DECLE   7, $200 + 5*20 + 7
        STRING  "world", 0

        ;; Wait for 1 second
        CALL    WAIT
        DECLE   60

        ;; Do it again
        B   @@loop

        ENDP
```

There really isn't much to this example, is there?

Putting it All Together

The source listing below puts all the fragments above into a complete program. Mainly, this just adds the cartridge header, the `INCLUDE`directives to include print.asm and fillmem.asmfrom SDK-1600, and assigns `WTIMER` a location in 8-bit memory. The bold portions are the new portions.

```
WTIMER  EQU     $102    ; Put WTIMER into 8-bit memory

        ROMW    16      ; 16-bit ROM
        ORG     $5000   ; Standard ROM memory map starts at $5000

;------------------------------------------------------------------------------
; EXEC-friendly ROM header.
;------------------------------------------------------------------------------
ROMHDR: BIDECLE ZERO            ; MOB picture base   (points to NULL list)
        BIDECLE ZERO            ; Process table      (points to NULL list)
        BIDECLE MAIN            ; Program start address
        BIDECLE ZERO            ; Bkgnd picture base (points to NULL list)
        BIDECLE ONES            ; GRAM pictures      (points to NULL list)
        BIDECLE TITLE           ; Cartridge title/date
        DECLE   $03C0           ; Flags:  No ECS title, run code after title,
                                ; ... no clicks
ZERO:   DECLE   $0000           ; Screen border control
        DECLE   $0000           ; 0 = color stack, 1 = f/b mode
ONES:   DECLE   1, 1, 1, 1, 1   ; Color stack initialization
;------------------------------------------------------------------------------

TITLE   STRING  $107, "Wait Timer Demo", 0  ; Title string and date (2007)

MAIN    PROC

        CALL    CLRSCR              ; Clear the screen

        MVII    #MYISR, R0          ; \
        MVO     R0,     $100        ;  |_ Write out "MYISR" to $100-$101
        SWAP    R0                  ;  |  Double Byte Data.
        MVO     R0,     $101        ; /
        EIS                         ; Make sure interrupts are enabled

@@loop:
        ;; Print "Hello" at row #5, column #7.
        CALL    PRINT.FLS
        DECLE   7, $200 + 5*20 + 7
        STRING  "Hello", 0

        ;; Wait for 1 second
        CALL    WAIT
        DECLE   60

        ;; Print "World" at row #5, column #7.
        CALL    PRINT.FLS
        DECLE   7, $200 + 5*20 + 7
        STRING  "world", 0

        ;; Wait for 1 second
        CALL    WAIT
        DECLE   60

        ;; Do it again
        B   @@loop

        ENDP

;; ======================================================================== ;;
;;  MYISR -- A simple interrupt service routine                             ;;
;; ======================================================================== ;;
MYISR   PROC
        MVO     R0,     $20     ; Enable the display

        MVI     WTIMER, R0      ; Get current timer count
        DECR    R0              ; Decrement it
        BMI     @@expired       ; If it went negative, it's expired
        MVO     R0,     WTIMER  ; Store updated count
@@expired:

        JR      R5              ; return from interrupt
        ENDP

;; ======================================================================== ;;
;;  WAIT -- Wait for some number of tics                                    ;;
;;                                                                          ;;
;;  INPUTS for WAIT                                                         ;;
;;      1 decle after call:  Number of tics to wait                         ;;
;;                                                                          ;;
;;  INPUTS for WAIT.1                                                       ;;
;;      R0  Number of tics to wait                                          ;;
;;                                                                          ;;
;;  OUTPUTS                                                                 ;;
;;      R0  Zeroed                                                          ;;
;;                                                                          ;;
;;  NOTE                                                                    ;;
;;      Requires WTIMER code in ISR, and interrupts enabled.                ;;
;;                                                                          ;;
;; ======================================================================== ;;
WAIT    PROC
        MVI@    R5,     R0          ; Get # of tics to wait from after CALL
@@1:    MVO     R0,     WTIMER      ; Set up wait timer

        CLRR    R0                  ; \
@@wait: CMP     WTIMER, R0          ;  |- Wait for WTIMER = 0
        BNEQ    @@wait              ; /
        JR      R5                  ; return
        ENDP

;; ====================================================================== ;;
;;  Library includes                                                      ;;
;; ====================================================================== ;;
        INCLUDE     "fillmem.asm"
        INCLUDE     "print.asm"
```

Copy this out to a file, `wtdemo.asm`, along with copies of `print.asm` and `fillmem.asm` that were linked to above. Then assemble with:

```
as1600 -o wtdemo -l wtdemo.lst wtdemo.asm
```

Ta da! When you run this program, you should get a display that alternates between "Hello" and "World" in the middle of the screen with a 1 second delay between each.

Tagalong Todd Tutorial

Overview

If you're looking to get into developing programs (presumably games) for the Intellivision platform, you can go through the Hello World tutorials and read a good bit about the CP1600 architecture and the AS1600 assembler syntax, etc., etc. But to start programming anything of any consequence, you typically hear that you should start by understanding what is going on in the very simple demonstration game Tagalong Todd. I've attempted that a couple of times and found it difficult. Finally, I just started attempting to understand the game by ripping it apart and seeing what changes/breaks. At that point, writing up a tutorial seemed like a good idea as that will help me to understand the code as well.

This tutorial will attempt to build up the Tagalong Todd example game from the title screen on up through a working "game" one bite-sized chunk at a time.

There are several versions of Tagalong Todd out there. There are at least three that are shipped with the SDK itself! This tutorial will work through the simplest one, the one found in the examples/tagalong directory within the SDK directory structure.

Assumptions

This tutorial will try not to duplicate documentation available elsewhere in the SDK documentation and/or on the Intelliwiki. As such it will not go into details about as1600 syntax, the architecture of the CP1600, etc. However, it will touch on those topics insomuch as they are related to the coding of the examples, of course.

The tutorial also assumes you've already worked through the "Hello World" tutorials, and will pretty much pick up where those left off. E.g. it is assumed you understand how to overwrite text on the title screen, can assemble and run that example code and not much more.

The Progression

The plan for this tutorial is to take the very basic Tagalong Todd "game" that comes with the SDK and break it down into small chunks that can build upon each other until the entire game is understood. As such, the tutorial will follow this progression:

- Part 1: Start with code that can display the title screen and then draw the player figure.
- Part 2: Add code to move the figure around the screen programmatically.
- Part 3: Add code to read controller input and move the figure around the screen in response.
- Part 4: Add code to draw Todd on the screen.
- Add code to cause Todd to chase the player.

At that point, the game should be very close to the one that comes with the SDK.

Tagalong Todd Tutorial: Part 1

This part of the tutorial will show you how to draw a basic title screen, then put the player graphic onto the screen.

The complete source for this tutorial (it's a subset of the full tagalong.asm found in the SDK) is here.

Contents

The Title screen

The code to produce the title screen doesn't involve anything more complicated than what you've already done for the "Hello World" examples. So I'll just dump that code out here with minimal comment.

```
            ROMW    16              ; Use 16-bit ROM

;------------------------------------------------------------------------------
; Include system information
;------------------------------------------------------------------------------
            INCLUDE "gimini.asm"
```

Depending on the version of "Hello World" you worked through, you may not have seen this include statement before. Basically, gimini.asm just contains a lot of nice mnemonic definitions about the INTV system that will allow us to use identifiers like **C_BLU** (to indicate the code for the color blue) instead of having to always use hard to remember numeric codes.

```
;------------------------------------------------------------------------------
; EXEC-friendly ROM header.
;------------------------------------------------------------------------------
            ORG     $5000           ; Use default memory map
ROMHDR:     BIDECLE ZERO            ; MOB picture base   (points to NULL list)
            BIDECLE ZERO            ; Process table      (points to NULL list)
            BIDECLE MAIN            ; Program start address
            BIDECLE ZERO            ; Bkgnd picture base (points to NULL list)
            BIDECLE ONES            ; GRAM pictures      (points to NULL list)
            BIDECLE TITLE           ; Cartridge title/date
            DECLE   $03C0           ; No ECS title, run code after title,
                                    ; ... no clicks
ZERO:       DECLE   $0000           ; Screen border control
            DECLE   $0000           ; 0 = color stack, 1 = f/b mode
ONES:       DECLE   C_BLU, C_BLU    ; Initial color stack 0 and 1: Blue
            DECLE   C_BLU, C_BLU    ; Initial color stack 2 and 3: Blue
            DECLE   C_BLU           ; Initial border color: Blue
;------------------------------------------------------------------------------
```

This should be pretty familiar from the "Hello World" example.

```
;; ======================================================================== ;;
;;  TITLE  -- Display our modified title screen & copyright date.           ;;
;; ======================================================================== ;;
TITLE:      PROC
            STRING  102, "Tagalong Todd", 0
            BEGIN

            ; Patch the title string to say '=JRMZ=' instead of Mattel.
            CALL    PRINT.FLS       ; Write string (ptr in R5)
            DECLE   C_WHT, $23D     ; White, Point to 'Mattel' in top-left
            STRING  '=JRMZ='        ; Guess who?  :-)
            STRING  ' Productions'
            BYTE    0

            CALL    PRINT.FLS       ; Write string (ptr in R1)
            DECLE   C_WHT, $2D0     ; White, Point to 'Mattel' in lower-right
            STRING  '2002 =JRMZ='   ; Guess who?  :-)
            BYTE    0

            ; Done.
            RETURN                  ; Return to EXEC for title screen display
            ENDP
```

This is the title screen, again hopefully this is familiar territory. If you have any questions about this code, I recommend going back through the "Hello World" tutorials again.

```
;; ======================================================================== ;;
;;  MAIN:  Here's our main program code.                                    ;;
;; ======================================================================== ;;
MAIN:       PROC
            BEGIN

            CALL    CLRSCR          ; Clear the screen

            RETURN                  ; Return to the EXEC and sit doing nothing.
            ENDP
```

At this point, we'll just clear the screen and do nothing.

```
;; ======================================================================== ;;
;;  LIBRARY INCLUDES                                                        ;;
;; ======================================================================== ;;
            INCLUDE "print.asm"       ; PRINT.xxx routines
            INCLUDE "fillmem.asm"     ; CLRSCR/FILLZERO/FILLMEM
```

And some library functions. That's about it. This gets us a title screen; the full code for this much is in tag0.asm. We now have even less functionality working than we had with the "Hello World" example! But we're going to be adding to it immediately.

Adding a player graphic

In order to add a player graphic to the screen, we need to tackle a couple of general concepts that will be used in the real game. It is certainly possible to draw a graphic to the screen without doing everything I'll be going over, but you wouldn't be able to build much on code that was that simple. In order for our code to be useful in a real game at some point, we really need to build in some support so that the Intellivision can draw to the screen within game play.

To make that happen, we'll learn about two different game concepts here: "shadowing" the data that will be drawn on the screen, and the taskq library provided with the SDK.

Shadowing display data

I don't want to get too technical here because I want this tutorial to stay at an introductory level. But basically, you can think of games on the Intellivision (and on most gaming platforms) as a large loop of instructions running over and over again. One of the things that happens in this master loop is the drawing of the screen. While the screen update is happening, we don't want to be doing anything computationally expensive because there just isn't much time. (For more detailed info, look for **VBLANK** in the documentation).

A strategy for handling this is to keep a copy of the data that needs to be displayed somewhere in the program so that the copy can be updated at any point. This copy is usually called a "shadow". Then when the Intellivision is updating its display, we can simply have it copy the shadow data to the real display registers - something that is simple and fast. This is the strategy we'll follow for this example game.

So what data will we need to treat in this manner? Any data that will be changing the game's display. In this example we'll only be using 1 sprite graphic called a **M**oveable **OB**ject or **MOB**. So we'll need to shadow the data that controls the first MOB. (For more info on the display and MOBs, look for **STIC** and **MOB** in the documentation).

Data to shadow for MOB 0

If you look in the gimini.asm library file, you'll see some ASCII art diagrams that detail what pieces of data are used to control the different MOBs. For our purposes, we're interested in updating the MOB's location (X and Y coordinates), visibility (we want our MOB to be visible), and color. Those chunks of data are controlled via registers in the STIC. As it turns out the registers that control all of these data items can be addressed in one block of address space. It looks like this:

```
STIC address
|
v
+---------------------------+---------------------------+---------------------------+
| 'X' registers for MOBs 0-7 | 'Y' registers for MOBs 0-7 | 'A' registers for MOBs 0-7 |
+---------------------------+---------------------------+---------------------------+
```

The X registers contain (among other things) the X coordinates and visibility flags for the 8 MOBs. The Y registers contain (among other things) the Y coordinates for the 8 MOBs. The A registers contain (among other things) the color information for the 8 MOBs. So in order for us to update those registers, we will do a copy of memory from a buffer to these registers. The simplest way to do that is to define a contiguous block of 24 words so that we can just copy that into these registers in one efficient loop. This is slightly wasteful in terms of space (we won't be storing or changing data for MOBs 1-7, only MOB 0), but it is simple.

The other info we need to define for the MOB is the graphic image itself. That will also be done via a buffer that will be copied into the right place during the display update (VBLANK) time.

Setup the data

So we'll need a 24 word buffer:

```
            ; STIC shadow
STICSH      RMB     24              ; Room for X, Y, and A regs only.
```

We'll define this in an area of scratch memory along with some other buffers we'll be defining shortly.

Another couple of data items we'll want to have in a stored location are the X and Y coordinates of the player MOB. Technically at this point we could just store these in the STICSH buffer we just defined. But by having a separate dedicated location for these, we can store those coordinates in any method we might choose instead of only being able to store the position in numbers of pixels.

So defining the MOB X and Y positions:

```
PLYR        PROC
@@XP        RMB     1               ; X position
@@YP        RMB     1               ; Y position
            ENDP
```

We've chosen to define this as a named data structure. This is a bit of overkill for just two items, but we're going to be adding more later. So we'll have PLYR.XP and PLYR.YP available, each storing one word which can represent the player's position.

The last piece of data setup that we'll need is the data defining the graphic image of our player. We'll do that after we discuss ISRs below.

Code to copy the shadow data

Now that we have storage defined for our "shadow STIC" area and we've defined our graphic image, we need to write some code that will copy that data into the right spots. To that end, we'll create a proc that will update the MOB data for us. This will do whatever computations are necessary to populate the shadow STIC area. The code that copies the data from the shadow area into the actual display area will come later.

At the start of the MOB_UPDATE routine, we push R5 onto the stack to save its value because we're going to use it in our routine. R5 will contain the address that the function should return to after the routine is finished. If you check the bottom of the routine, you can see a corresponding **PULR PC**call that pulls this address off the stack and puts it back into the program counter (R7).

```
            MVII    #@@mobr,    R4      ; MOB information template
            MVII    #STICSH,    R5
```

These two lines define two pointers that we will use to update the shadow STIC area. We've chosen R4 and R5 because they auto-increment during indirect access which will make the coding a bit simpler. R5 points to the shadow STIC buffer we defined earlier, and R4 points to a little local buffer at the end of the routine. That buffer contains bit patterns that we will use to mask our values before storing them.

```
            MVI     PLYR.XP,    R0      ;\
            SWAP    R0                  ; |
            ANDI    #$00FF,     R0      ; |- Player X position
            XOR@    R4,         R0      ; |
            MVO@    R0,         R5      ;/
```

To understand this chunk of code, we need to understand that the player's X position will eventually be stored as a fixed point data item, with the integer part in the upper 8 bits and the fractional part in the lower 8 bits. The code then takes the player's X position, SWAPS its bytes so that the integer part is now in the lower 8 bits, then uses the ANDI operation to drop off the fractional part. It then XORs that position with the first word at our local buffer at @@mobr (pointed to via R4). That first word is a bit pattern from gimini.asm that will set the visibility bit. Of course, all this math was done in the R0 register, and that value is then copied into the first word of our shadow STIC buffer (pointed to via R5).

At this point, both R4 and R5 are now pointing to the next word in their respective buffers, because we used them in indirect mode. So R4 is now pointing to the next word in our @@mobr buffer, and R5 is pointing to the second word in our shadow STIC. But that word will shadow the X coordinate of MOB1 (which we're not using at this point, but the data space is there nonetheless). We need to adjust the pointer.

```
            ADDI    #7,         R5      ;  Move pointer to Y coordinate section of the STICSH
```

This line advances it to the Y coordinate section.

```
            MVI     PLYR.YP,    R0      ;\
            SWAP    R0                  ; |
            ANDI    #$007F,     R0      ; |- Player Y position
            XOR@    R4,         R0      ; |
            MVO@    R0,         R5      ;/
```

The next chunk of code is almost identical for the player Y position. The mask value of $007F is due to the fact that the Y position is only stored in the lowest 7 bits rather than the lowest 8 bits like the X position was.

```
            ADDI    #7,         R5      ; Move pointer to A register section of the STICSH
```

R4 is at the next word in our local buffer but we need to advance R5 again.

```
            MVI@    R4,         R0      ; \_ Player's A register
            MVO@    R0,         R5      ; /
```

This just copies the data from our local buffer into the shadow STIC. This is to set the color of the MOB.

```
            CLRR    R0
            MVO     R0,         MOB_BUSY
```

This bit of code puts a zero into the MOB_BUSY location. Let's postpone discussion of this item until we get to the section on the ISR.

```
            PULR    PC
```

As mentioned above, this sets the PC to the address the program should return to.

All that's left is to setup the code such that our MOB_UPDATE routine gets called at appropriate times. We'll do that via an **I**nterrupt **S**ervice **R**outine (or **ISR**). The SDK already has good documentation of ISRs, so we won't get into all the details of that here. Our ISR is going to employ the taskq library, so we'll discuss that next.

Using the taskq library

The comments in the taskq.asm file really describe all that is necessary to use the taskq library. So I won't duplicate that documentation here. I'll only touch on how we are using the taskq library in this example program.

Without getting too detailed, the taskq library allows you to define tasks (procedures) that need to be executed in order. The queue takes care of executing the tasks for you, and you can keep adding tasks to be executed based on whatever criteria the game requires - including having tasks which queue other tasks.

The taskq library relies on various specially named storage locations being defined in the program. For this tutorial, we'll just note the values used and leave it as an exercise for the reader to read up in the taskq comments for more details.

The taskq related definitions and storage for this game:

```
TSKQM       EQU     $7              ; Task queue is 8 entries large
MAXTSK      EQU     1               ; Only one task

TSKQHD      RMB     1               ; Task queue head
TSKQTL      RMB     1               ; Task queue tail
TSKDQ       RMB     2*(TSKQM+1)     ; Task data queue
TSKACT      RMB     1               ; Number of active tasks

TSKQ        RMB     (TSKQM + 1)     ; Task queue
TSKTBL      RMB     (MAXTSK * 4)    ; Timer task table
```

Basically, it is enough at this stage to understand that these are defined as part of using the taskq library.

Let's take a look at the ISR procedure itself:

```
ISR         PROC

            ;; ------------------------------------------------------------ ;;
            ;;  Basics:  Update color stack and video enable.              ;;
            ;; ------------------------------------------------------------ ;;
            MVO     R0,     STIC.viden  ; Enable display
            MVI     STIC.mode, R0       ; ...in color-stack mode
```

These lines do exactly what the comments say. It's called the "VBLANK handshake", and it tells the STIC to draw the screen for this frame. Just writing something to .viden (which you can see maps to address $20 if you look in gimini.asm) enables display, and reading STIC.mode tells it to be in color-stack mode. Discussion of the different video modes is beyond the scope of this tutorial. Suffice it to say that this code enables drawing on the screen.

```
            MVII    #C_GRY, R0          ;\
            MVO     R0,     STIC.cs0    ; |__ Set display to grey
            MVO     R0,     STIC.cs2    ; |
            MVO     R0,     STIC.bord   ;/
```

These lines set the colors for the background and the border to grey. Again, you can find more detail by reading comments in gimini.asm and the STIC documentation.

```
            ;; ------------------------------------------------------------ ;;
            ;;  Update STIC shadow and queue updates for MOB velocities.    ;;
            ;; ------------------------------------------------------------ ;;
            MVI     MOB_BUSY, R0
            TSTR    R0
            BNEQ    @@no_mobs
            MVO     PC,     MOB_BUSY
```

Now we're in a good position to talk about what we're doing with the MOB_BUSY location. This is serving as a flag to make sure that we don't bother copying our shadow STIC area unless we need to.

In detail, this code copies the current value from MOB_BUSY into R0 so it can be tested via the TSTR instruction. This will set the zero flag if the value is zero, clearing that flag if it isn't zero. BNEQ (**B**ranch on **N**ot **EQ**ual) will branch if the zero flag is NOT set - effectively making that opcode mean "branch if not zero", indeed a synonym is BNZE for **B**ranch if **N**ot **ZE**ro. So if MOB_BUSY is not zero, then the next chunk of code will be skipped as the PC moves down to the @@no_mobs label.

The first thing executed if the MOB_BUSY location is zero, is to set the current value of the PC to it making it non-zero. So we won't enter this section of code again until that location is reset back to zero when MOB_UPDATE next runs.

```
            CALL    MEMCPY              ;\__ Copy over the STIC shadow.
            DECLE   $0000, STICSH, 24   ;/
```

And here's where the magic happens - we copy our STIC shadow area into the actual STIC registers. You can look at the comments for the MEMCPY routine to see that it gets its argument list from the bytes immediately following the call itself. In this case the arguments we're sending are '$0000, STICSH, 24' which will tell MEMCPY to copy 24 words of memory from our STICSH buffer to memory location 0 - which is the start of the STIC MOB registers.

```
            MVII    #MOB_UPDATE, R0
            JSRD    R5,   QTASK
```

These lines will stick the address of MOB_UPDATE into the task queue so that it gets run again.

```
@@no_mobs:

            CALL    DOTIMER             ; Update timer-based tasks.
```

This is necessary to make sure that the taskq library updates any tasks that rely on timers. Consider this part of the contract of using the taskq library.

```
            B       $1014               ; return from interrupt.
            ENDP
```

And the end of the ISR. You can think of $1014 as the magic address that the Intellivision expects returns from ISRs to branch back to. More detail is (as usual) available in the SDK docs.

Getting our player graphic into the system

The last piece of this puzzle is to define the graphic image for our player and write the code that copies that data into the right place so that it can be displayed. You can look up how to format the data so that it is simply copyable into the display area in the docs, but the gist of it is that we can define it like this:

```
;; ======================================================================== ;;
;;  GRAMIMG -- Custom graphics to load into GRAM.                           ;;
;; ======================================================================== ;;
GRAMIMG     PROC

@@person:   ; Crappy person graphic.
            DECLE   %00010000
            DECLE   %00111000
            DECLE   %00111000
            DECLE   %00010000
            DECLE   %00010000
            DECLE   %01111100
            DECLE   %10111010
            DECLE   %10111010
            DECLE   %10111010
            DECLE   %10111010
            DECLE   %00111000
            DECLE   %00101000
            DECLE   %00101000
            DECLE   %00101000
            DECLE   %00101000
            DECLE   %01101100
@@end:
            ENDP
```

You can see the shape as the 1's in among the 0's. The data labels are niceties that will allow us to reference the beginning and end of the data space via those labels in our code, allowing us to calculate the length of the data block rather than having to count up the number of words used.

The code that copies this data into the **G**raphics **RAM** area (GRAM) has to be handled specially, because we can only copy this data into place during the VBLANK time period. That's when the Intellivision has access to the STIC GRAM data area. What this means is that we need to copy this data into place during an ISR. We'll do that by writing an ISR that only runs once. It will copy the data into place and then immediately setup our real ISR that will handle things from that point forward.

```
;; ======================================================================== ;;
;;  INITISR -- Copy our GRAM image over, and then do the plain ISR.         ;;
;; ======================================================================== ;;
INITISR     PROC
            PSHR    R5
```

Standard start to a proc.

```
            CALL    MEMCPY
            DECLE   $3800, GRAMIMG, GRAMIMG.end - GRAMIMG
```

This copies the above memory buffer into the right area. As before, you can see how MEMCPY is working here. It will copy the buffer starting at GRAMIMG with length of (GRAMIMG.end - GRAMIMG) into the memory locations at $3800.

```
            MVII    #ISR,    R0
            MVO     R0,      ISRVEC
            SWAP    R0
            MVO     R0,      ISRVEC + 1
```

This little chunk of code is what "wires in" a procedure so that it gets called during the VBLANK time period when the Intellivision has access to the STIC registers. You can see in the listing that ISRVEC is defined to be memory location $100. In this case, we're setting things up so that our real ISR gets called from here on for VBLANK updates.

But how will this ISR itself be called? We'll set that up in MAIN, which we'll detail next.

```
            PULR    PC
            ENDP
```

And the standard ending of a proc.

The MAIN routine

And now for MAIN.

```
;; ====================================================================== ;;
;;  MAIN:  Here's our main program code.                                  ;;
;; ====================================================================== ;;
MAIN:       PROC
            DIS
```

This opcode will disable interrupts while we get things setup.

```
            MVII    #STACK, R6      ; Set up our stack
```

We need to setup a chunk of memory for the system to use as a stack, and then set R6 to the starting address (i.e. the top of the stack). Note that you may want to use the alias 'SP' for R6 because of its use as a stack pointer.

```
            MVII    #$25E,  R1      ;\
            MVII    #$102,  R4      ; |-- Clear all of RAM memory
            CALL    FILLZERO        ;/
```

This code just clears all of the RAM in the Intellivision. A good idea. You can see that it is calling FILLZERO. FILLZERO expects to find its inputs via R1 (the length of the memory to be cleared) and R4 (the starting address of the memory to be cleared). This will clear out memory from $102 to $360, which is all the RAM in the machine.

```
            MVII    #INITISR, R0    ;\    Do GRAM initialization in ISR.
            MVO     R0,     ISRVEC  ; |__ INITISR will the point to the
            SWAP    R0              ; |   regular ISR when it's done.
            MVO     R0,     ISRVEC+1;/
```

This chunk of code should look very similar. We just discussed how to wire-in an ISR routine. You should now be able to see what we're doing here. We're going to have INITISR called first, which will copy the person graphic into the GRAM area and then wire-in the real ISR. Basically, INITISR is doing a little setup for us that can only be done via an ISR.

```
            ;; ------------------------------------------------------------ ;;
            ;;  Put the character on the screen                             ;;
            ;; ------------------------------------------------------------ ;;
            MVII    #$1000, R0
            MVO     R0,     PLYR.XP
            MVO     R0,     PLYR.YP
```

Here we're putting reasonable values as starting values into the X and Y positions for the player. Specifically we're setting both X and Y to $1000. $1000 will look like this bitwise (showing both bytes as halves): 00010000 00000000. When we discussed the algorithm for populating our shadow STIC, we discussed that we are going to represent our position on the screen as a fixed decimal number. So the most significant bits (MSBs) will be the integer part and the least significant bits (LSBs) will be the fractional part. By putting the value of $1000 into X and Y, we are positioning the player at (16.0, 16.0) or (16,16) a spot that is comfortably visible on the screen in the upper left quadrant of the full display resolution of 160x96.

```
            EIS
```

This instruction enables interrupts again. Basically this is the "GO!" command to make things start running!

```
            ;; ------------------------------------------------------------ ;;
            ;;  Fall into the RUNQ.  We should never exit the RUNQ in this  ;;
            ;;  demo, since we never call SCHEDEXIT.                        ;;
            ;; ------------------------------------------------------------ ;;
            CALL    RUNQ            ; Run until a SCHEDEXIT happens
```

This tells the taskq engine to start executing tasks. As the comments indicate, the taskq engine calls tasks that get queued up until someone calls SCHEDEXIT. We haven't put a call to SCHEDEXIT in any of our tasks or our ISR, so this shouldn't happen. Effectively, this "game" will run forever.

```
            ;; ------------------------------------------------------------ ;;
            ;;  If a SCHEDEXIT *does* happen (say, due to a bug), crash     ;;
            ;;  gracefully.                                                 ;;
            ;; ------------------------------------------------------------ ;;
            CALL    PRINT.FLS
            DECLE   C_RED, $200 + 11*20
                    ;01234567890123456789
            STRING  "SCHEDEXIT WAS CALLED",0
```

The comments here pretty much tell the story. If we somehow call SCHEDEXIT accidentally, we'd like to know that. This will put a message onto the screen in red letters.

```
          DECR    PC              ; Can't get here
          ENDP
```

Again, we should never get here. But if it happens, this line of code will cause the Intellivision to just spin here forever. Specifically, decrementing the program counter (PC, aka R7) will cause the PC to move back one word such that it points again to the 'DECR PC' instruction. This works because 'DECR PC' is one word in length.

And the includes

```
;; ======================================================================== ;;
;;  LIBRARY INCLUDES                                                        ;;
;; ======================================================================== ;;
            INCLUDE "print.asm"       ; PRINT.xxx routines
            INCLUDE "fillmem.asm"     ; CLRSCR/FILLZERO/FILLMEM
            INCLUDE "memcpy.asm"      ; MEMCPY
            INCLUDE "timer.asm"       ; Timer-based task stuff
            INCLUDE "taskq.asm"       ; RUNQ/QTASK
```

A couple new includes need to be added as well.

That's it!

Assemble and run this little program and you should see a custom title screen, followed by our player graphic standing still. Wow, not much at all... Such is the nature of assembly programming! But really, we've put a lot into this program that is quite useful. We'll expand on these concepts in the next tutorials as we work ever closer towards complete understanding of the Tagalong Todd example.

Tagalong Todd Tutorial: Part 2

In this part of the tutorial, we'll just add enough code to move our player figure around the screen a bit.

The complete source for this tutorial (it's a subset of the full tagalong.asm found in the SDK) is here.

Contents

A new periodic task

In order to move our player figure around the screen, we just need a task that will run periodically and adjust the X and Y position variables for our player. This is pretty simple to do because we already have included the taskq library.

By the way, this is an intermediate piece of code that is not in the final tagalong.asm code listing. This seems to me to be a nice intermediate step towards updating the position of the player via hand controller input, so I'm including it as a step in the tutorial.

Moving the player

We'll write a procedure that will adjust the PLYR.XP and PLYR.YP locations programatically. Then we'll rely on the taskq library to update the player on the screen by executing the MOB_UPDATE routine we have already written.

The procedure:

```
;; ======================================================================== ;;
;;  MOVE_PLAYER -- Just programatically move the player around a bit.       ;;
;; ======================================================================== ;;
MOVE_PLAYER PROC
            PSHR    R5

            ; Just for something to do we'll move across the screen, then
            ; down a row and back across, again and again until we get to the
            ; bottom at which time we'll start over
            MVI     PLYR.XP,    R0      ;\
            SWAP    R0                  ; >-- Get player X position
            ANDI    #$00FF,     R0      ;/
```

This should look familiar. We're using the same method to grab the X position as we did in the MOB_UPDATE routine.

```
            ADDI    #8,         R0      ; move 8 X positions to the right
```

This just adds 8 pixels to the X position.

```
            MVII    #159,       R1      ;\__ compare X against the right edge of the screen
            CMPR    R0,         R1      ;/
            BLT     @@fixx

            SWAP    R0                  ; put the integer part back into the upper byte
            MVO     R0,    PLYR.XP      ; update the X position
            B       @@done              ; we're done
```

This code compares our X coordinate with 159. 159 is the rightmost pixel that can be displayed on the Intellivision. So when we get to this spot, it is time to move the player character down a row.

We compare the X position with 159 via the 'CMPR R0, R1' instruction. Then BLT will branch if R1 is less than R0, which would mean that 159 is less than our X coordinate, which means that our X coordinate is now off the right edge of the screen. We branch to @@fixx, where we'll fix the X coordinate.

If 159 is greater than or equal to our X coordinate, our X coordinate is still within the screen resolution. The code falls through and SWAPs the bytes again so that the integer part is in the upper byte of the register and puts that value back into PLYR.XP. When this happens, we're all done - no adjustment to the Y coordinate is necessary.

```
            ; here we've hit the right edge of the screen and should move down a row
@@fixx      MVII    #0,         R0      ;\__ set X position back to zero
            MVO     R0,    PLYR.XP      ;/

            MVI     PLYR.YP,    R0      ;\
            SWAP    R0                  ; >-- Get player Y position
            ANDI    #$007F,     R0      ;/

            ADDI    #8,         R0      ;  move 8 Y positions down
```

We reset X to 0, and we also need to move down a row. So we add 8 pixels to our Y coordinate.

```
            MVII    #96,        R1      ;\__ compare Y against the bottom edge of the screen
            CMPR    R0,         R1      ;/
            BLT     @@fixy

            SWAP    R0                  ; put the integer part back into the upper byte
            MVO     R0,    PLYR.YP      ; update the Y position
            B       @@done
```

We're doing a similar comparison to that above by checking the bottom of the screen. If we hit the bottom, we'll fix the Y coordinate at @@fixy. If not, we can put the Y coordinate back into PLYR.YP, and we're done.

```
            ; here we've hit the bottom edge of the screen and should move to the top
@@fixy      MVII    #0,         R0      ;\__ set Y position back to zero
            MVO     R0,    PLYR.YP      ;/

@@done      PULR    PC
            ENDP

```

To fix the Y coordinate, we'll set it to zero, effectively jumping back to the top of the screen.

I'm sure this isn't the most efficient way to do things, but it works and is simple.

Yes, this is slightly wrong...

The above code checking against the borders is actually wrong. The coordinate system for MOBs is a bit different. You can check the STIC documentation for the actual details. It doesn't actually affect our tutorial very much, so I'm not getting into the intricacies of how the MOBs relate to the actual screen resolution in this tutorial.

Calling our new routine

As mentioned at the top of this tutorial, we'll use the taskq library to call our new routine. There are probably several ways of doing it, and we'll try two.

A first, very simple method

The first is just to put a call to our new movement routine into the queue alongside of where we are putting our MOB_UPDATE routine into the queue. That will look like this:

```
            MVII    #MOB_UPDATE, R0
            JSRD    R5,   QTASK

            MVII    #MOVE_PLAYER, R0
            JSRD    R5,   QTASK
```

Just place those second two lines into the ISR from our last tutorial, and the taskq library will get a request to move the player over and over again.

Build it and try it!

This is enough to move our player around. He'll move pretty quickly, because a request to move the player is going into the task queue during every ISR. But it illustrates the point.

Another way to call MOVE_PLAYER

We can insert the call to MOVE_PLAYER into the task queue in a different manner. Specifically, the task queue allows for timer based tasks. We can tell the task queue to process a

MOVE_PLAYER every N seconds. For our next example, we'll have it move the player every 1/4 second.

We'll tell the task queue about the timer task back in main during our initial setup code.

```
                ;; ------------------------------------------------------------ ;;
                ;;  Set up character movement task                              ;;
                ;; ------------------------------------------------------------ ;;
                CALL    STARTTASK
                DECLE   0
                DECLE   MOVE_PLAYER
                DECLE   30, 30          ; 4Hz

                MVII    #1,     R0
                MVO     R0,     TSKACT
```

The STARTTASK call is what sets up a timer based task in the queue. You can read the comments in timer.asm to get all the details, but this call works by defining the task number (0), then providing the procedure to call (MOVE_PLAYER), then providing details about how often the task should be run (the 30, 30 pair).

After telling the task queue about our new task, we set TSKACT to 1 to tell the task queue library that 1 task has been defined.

Build it and try it!

When this version is built and run, the player graphic will move across the screen at a much more reasonable pace.

That's it!

We now have a player graphic moving around. And it didn't take too much additional code after we laid all the groundwork in part 1.

Tagalong Todd Tutorial: Part 3

In this part of the tutorial, we'll add code that will read the hand controller and move our player figure around the screen in response.

The complete source for this tutorial (it's a subset of the full tagalong.asm found in the SDK) is here.

Contents

Changes to our movement routine

In order to move the character figure around in a believable manner, we'll change the algorithm we've been using for the movement. Our last update just moved the figure via an offset. We'll change that here to include a velocity measure. As the user presses the disc, we'll speed up the character movement until it hits a maximum speed.

Introducing scanhand

A set of routines that read the hand controllers is provided with the SDK in a library file called scanhand.asm. It is built to be used with the TASKQ library and both pieces make assumptions about the other. At this point, it's enough to know that we need to allocate specifically named chunks of memory for the library routines.

```
            ; Hand-controller 8-bit variables
SH_TMP      RMB     1               ; Temp storage.
SH_LR0      RMB     3               ;\
SH_FL0      EQU     SH_LR0 + 1      ; |-- Three bytes for left controller
SH_LV0      EQU     SH_LR0 + 2      ;/
SH_LR1      RMB     3               ;\
SH_FL1      EQU     SH_LR1 + 1      ; |-- Three bytes for right controller
SH_LV1      EQU     SH_LR1 + 2      ;/

            ; Hand-controller 16-bit variables
SHDISP      RMB     1               ; ScanHand dispatch
```

We're also going to need more variables to store the new velocity measures we'll be manipulating.

```
PLYR        PROC
@@XP        RMB     1               ; X position
@@YP        RMB     1               ; Y position
@@XV        RMB     1               ; X velocity
@@YV        RMB     1               ; Y velocity
@@TXV       RMB     1               ; Target X velocity
@@TYV       RMB     1               ; Target Y velocity
            ENDP
```

The XP and YP variable should look familiar from part 2 of the tutorial. We're introducing two new pairs of values: a current velocity and a target velocity in each of the X and Y directions.

After those pieces of setup are done, we can get into the algorithm changes.

```
            ;; ------------------------------------------------------------ ;;
            ;;  Set up our hand-controller dispatch.                        ;;
            ;; ------------------------------------------------------------ ;;
            MVII    #HAND,  R0      ;\__ Set up scanhand dispatch table
            MVO     R0,     SHDISP  ;/
```

Here we're setting up the dispatch table that the scanhand routine expects. This probably warrants a bit of explanation. scanhand wants the addresses of three routines:

1. a routine to call when someone presses a button on the keypad
2. a routine to call when someone presses a side action button
3. a routine to call when someone presses a disc direction

The addresses of those routines need to be provided in that order. So we see this chunk of code later in the file:

```
;; ======================================================================== ;;
;;  HAND    Dispatch table.                                                 ;;
;; ======================================================================== ;;
HAND        PROC
            DECLE   HIT_KEYPAD
            DECLE   HIT_ACTION
            DECLE   HIT_DISC
            ENDP
```

These lines setup the three routines that scanhand should call. Those routines need to be defined in our code - after all, scanhand doesn't know what we want to do when the player presses buttons or disc.

Precomputing velocities for various angles

```
;; ======================================================================== ;;
;;  SINTBL  -- Sine table.  sin(disc_dir) * 511                             ;;
;; ======================================================================== ;;
SINTBL      PROC
            DECLE   $0000
            DECLE   $00C3
            DECLE   $0169
            DECLE   $01D8
            DECLE   $01FF
            DECLE   $01D8
            DECLE   $0169
            DECLE   $00C3
            DECLE   $0000
            DECLE   $FF3D
            DECLE   $FE97
            DECLE   $FE28
            DECLE   $FE01
            DECLE   $FE28
            DECLE   $FE97
            DECLE   $FF3D
            ENDP
```

This is a lookup table that defines the values for sin(x) * 511 for the 16 directions that the directional disc can input. These will be the values we'll apply to the Y direction velocity as we move the player around the screen. Note that we don't need a cos(x) table because cos(x) = sin(90 - x) (for values of x in degrees). For our purposes here, since there are 16 directions: cos(direction) = sin(direction - 4). That is to say that the cosine value for a disc direction can be found by looking 4 entries earlier in this table (16 directions covers 360 degrees, so 4 directions covers 90 degrees).

Defining the three scanhand callbacks

```
;; ======================================================================= ;;
;;  HIT_KEYPAD -- Someone hit a key on a keypad.                           ;;
;; ======================================================================= ;;
HIT_KEYPAD  PROC
            JR      R5
            ENDP

;; ======================================================================= ;;
;;  HIT_ACTION -- Someone hit an action button                             ;;
;; ======================================================================= ;;
HIT_ACTION  PROC
            JR      R5
            ENDP
```

These two routines just return immediately when called. Basically, our game will do nothing on keypad presses or action button presses.

That leaves the directional disc algorithm.

When discussing the decoding of the information coming from scanhand, it is really kind of hard to make much sense without rewriting the documentation that comes in scanhand. When reading along with this part of the write-up, I suggest having the source code for scanhand.asm nearby. I'll be duplicating a small amount of what is written there just to make the tutorial simpler to follow, but there is a lot of good info in the comments in scanhand.asm.

Here's the short version (put together from choice info in scanhand.asm). Our routine will be given a control word, from which we can determine what occured. It will look like this:

```
       15                    9    8    7                        0
      +--------------------+-------+---+-----------------------+
      |       RESERVED      |CTRL # |RLS|      Input Number      |
      +--------------------+-------+---+-----------------------+
```

- The Input Number bit pattern will tell us what was pressed.
- The RLS bit tells us if this is a "press down" or a "release up" event.
- The CTRL # bits tell us which controller was pressed.

On to the code.

```
;; ====================================================================== ;;
;;  HIT_DISC   -- Someone hit a directional disc                          ;;
;; ====================================================================== ;;
HIT_DISC    PROC
            PSHR    R5
```

The standard saving of R5 onto the stack so we know where to return.

```
            ANDI    #$FF,   R2      ; Ignore controller number
```

scanhand puts the control word into R2. For our demo, we don't care about left vs. right controller, so we'll just wipe out the top half of the word - that includes the RESERVED area as well as the two controller number bits.

```
            CMPI    #$80,   R2
            BLT     @@pressed
```

Here we'll check the RLS bit to see if this is a "press down" event. If it is, we should process movement one way; if this is a "release up" event, we'll do something different. The logic for release is coming next, we'll branch to @@pressed for a disc press.

```
            CLRR    R0
            MVO     R0,     PLYR.TXV
            MVO     R0,     PLYR.TYV
            PULR    PC
```

In the event of a disc release, we'll set our target velocity to zero in both the X and Y directions. If the player releases the disc, he is indicating a desire to stop movement. That's all we need to do in the case of a release, so we use PULR PC to bail out of this routine immediately.

The next chunk of code takes the direction the player is pressing on the directional disc and sets the X and Y target velocity values accordingly. Line by line, it goes like this:

```
@@pressed:  MOVR    R2,     R1
```

Make a copy of the direction data (in R1). We'll use this copy for the cosine value.

```
            ADDI    #4,     R1
```

Adjust the R1 (cosine pointer) by 4 entries in the sine table. Again, that works because cos(x) = sin (90 - x).

```
            ANDI    #$F,    R1
```

If we moved off the end of the 16 values in the sine table, this operation handles the wrap-around because of the way that binary numbers work in this case. (clever!)

```
            ADDI    #SINTBL,R2          ; sine pointer
```

R2 contains the POSITION in the table of sine values for the value that we care about. By adding the base address of the table itself, we get a memory pointer to the sine value we want.

```
            ADDI    #SINTBL,R1          ; cosine pointer
```

Do the same as above for R1 (the cosine value).

```
            MVI@    R2,     R2          ; sine for our direction
```

Here we overwrite the R2 value with the sine value we are fetching. Before this operation, R2 was pointing at the value we wanted, now it contains the value.

```
            MVI@    R1,     R1          ; cosine for our direction
```

Again, the same for R1.

```
            NEGR    R2
```

We negate the velocity value for sine just to make it consistent with the addressing scheme of the screen. Y values start at 0 at the top of the screen and get larger as you go down.

```
            MVO     R2,     PLYR.TYV    ; Set our target Y velocity to sine
```

And finally, take the computed target Y velocity (really just the adjusted lookup value from the sine table) and stick it into our data structure.

```
            MVO     R1,     PLYR.TXV    ; Set our target X velocity to cosine
```

Again, the same for R1 - this time for the X velocity.

```
            PULR    PC
            ENDP
```

End procedure as normal.

That routine really only computed a current "target velocity." So we know what the player is attempting to do by looking at PLYR.TXV and PLYR.TYV. We'll make changes to our MOB_UPDATE function that will incorporate those values and move the player on the screen.

Positioning the player

```
;; ======================================================================= ;;
;;  MOB_UPDATE -- This updates the player's position                       ;;
;; ======================================================================= ;;
MOB_UPDATE  PROC
            PSHR    R5
```

This procedure opening should look very familiar.

```
            ;; ------------------------------------------------------- ;;
            ;;  Bring our actual velocity closer to our target velocity,    ;;
            ;;  and apply the velocity to our position.                   ;;
            ;; ------------------------------------------------------- ;;
```

As can be seen from this comment, we're going to do the math necessary to bring our current velocity closer to the target that we've computed in the scanhand call-back routine. The approach we'll take here will be to compute the difference between the target velocity and the current velocity, then adjust the velocity towards the target by 1/4 of that difference. That way the speed of the player adjusts in a way that is noticeable in gameplay.

The first chunk deals with the X direction.

```
            MVI     PLYR.TXV,   R0
            SUB     PLYR.XV,    R0
```

Put the target velocity into R0, then subtract the actual velocity. R0 will now contain the difference between them.

```
            SARC    R0
```

SARC is shift right, which effects a divide by 2. It also shifts the right-most bit into the carry flag. In terms of the math, that works just fine except for the case when the difference is very small, i.e. only 1. When we SARC the value of 1 (computing 1/2), we'll get 0 left in our register as our value to adjust by, and that will mean that the actual velocity will never really converge on the target velocity. So in that case, we'll just add that 1 back to our register as a way of rounding that 1/2 value back up.

However if the value is negative we don't want to add one back for the rounding effect. We'll just let that value fall away (rounding down).

```
            BMI     @@nr0
```

BMI branches on "minus" which skips over the add carry. This is the rounding down effect for a negative number.

```
            ADCR    R0
```

Here we're adding the 1 back to our number -- rounding up.

```
@@nr0       SARC    R0
            BMI     @@nr1
            ADCR    R0
```

And these three instructions do it all again, making our adjustment 1/4 of the overall difference we computed.

```
@@nr1       ADD     PLYR.XV,    R0
            MVO     R0,         PLYR.XV
            ADD     PLYR.XP,    R0
            MVO     R0,         PLYR.XP
```

Now we add this computed difference to the player X velocity, and then add that velocity value to the player's X position. So we now have a new position for the player that is reflective of how the player has pressed the directional disc!

```
            MVI     PLYR.TYV,   R0
            SUB     PLYR.YV,    R0
            SARC    R0
            BMI     @@nr2
            ADCR    R0
@@nr2       SARC    R0
            BMI     @@nr3
            ADCR    R0
@@nr3       ADD     PLYR.YV,    R0
            MVO     R0,         PLYR.YV
            ADD     PLYR.YP,    R0
            MVO     R0,         PLYR.YP
```

And there is a second set of identical operations for the Y velocity.

And the rest...

The rest of the routine hasn't changed from the last tutorial part. We just changed out how we calculated the positions and our rendering of the player graphic is exactly the same.

Wrapping it up

```
;; ======================================================================== ;;
;;  LIBRARY INCLUDES                                                        ;;
;; ======================================================================== ;;
            INCLUDE "print.asm"       ; PRINT.xxx routines
            INCLUDE "fillmem.asm"     ; CLRSCR/FILLZERO/FILLMEM
            INCLUDE "memcpy.asm"      ; MEMCPY
            INCLUDE "hexdisp.asm"     ; HEX16/HEX12
            INCLUDE "scanhand.asm"    ; SCANHAND
            INCLUDE "timer.asm"       ; Timer-based task stuff
            INCLUDE "taskq.asm"       ; RUNQ/QTASK
```

The library includes.

Build it and try it!

You should now be able to move the character around the screen via the hand controllers.

That's it!

We're really close to having this demo fully functional. With the player moving around, we really only need to add Todd.

Tagalong Todd Tutorial: Part 4

Contents

Tagalong Todd Tutorial: Part 4

In this part of the tutorial, we'll add code that will put Todd on the screen. We'll do so in a manner that will allow moving him around later.

The complete source for this tutorial (it's a subset of the full tagalong.asm found in the SDK) is here.

How we'll draw Todd

We already have all of the pieces in place to draw Todd on the screen. He's going to use the same drawing of a man, but in a different color. He'll be drawn on the screen using MOB_UPDATE just like we do for the player character. We just need to define the right values and adjust things a little.

A couple of storage places

As you remember, we started drawing the player by having 2 variables for the X and Y positions on the screen. We'll define the same for Todd.

```
TODD        PROC                    ; TODD's STATS
@@XP        RMB     1               ; X position
@@YP        RMB     1               ; Y position
            ENDP
```

This shouldn't really require much explanation at this point. If you have a hard time remembering how we setup the player character, go back to part 2 and give it another read.

A small addition to main

Over in main, we want to provide initial values for Todd's X and Y positions.

```
            ;; ------------------------------------------------------------ ;;
            ;;  Put you and Todd onscreen.                                  ;;
            ;; ------------------------------------------------------------ ;;
            MVII    #$1000, R0
            MVO     R0,     PLYR.XP
            MVO     R0,     PLYR.YP

            MVII    #$4000, R0
            MVO     R0,     TODD.XP
            MVO     R0,     TODD.YP
```

Right along side of the initial player position we set a value for Todd.

Changes to MOB_UPDATE

Todd will need very similar code to what we've written already. Specifically, we need to put Todd's X and Y positions into our shadow buffer in the same places where we are currently doing the same for the player.

```
            ;; ------------------------------------------------------------ ;;
            ;;  Merge our position with our MOB registers.                  ;;
            ;; ------------------------------------------------------------ ;;
            MVII    #@@mobr,    R4      ; MOB information template
            MVII    #STICSH,    R5

            MVI     PLYR.XP,    R0      ;\
            SWAP    R0                  ; |
            ANDI    #$00FF,     R0      ; |- Player X position
            XOR@    R4,         R0      ; |
            MVO@    R0,         R5      ;/

            MVI     TODD.XP,    R0      ;\
            SWAP    R0                  ; |
            ANDI    #$00FF,     R0      ; |- Todd X position
            XOR@    R4,         R0      ; |
            MVO@    R0,         R5      ;/
```

So we see a very similar block here for Todd's X position.

```
            ADDI    #6,         R5
```

We need to adjust the pointer by 6 locations instead of 7, because R5 has been incremented a second time when we did the operations for Todd.

```
            MVI     PLYR.YP,    R0      ;\
            SWAP    R0                  ; |
            ANDI    #$007F,     R0      ; |- Player Y position
            XOR@    R4,         R0      ; |
            MVO@    R0,         R5      ;/

            MVI     TODD.YP,    R0      ;\
            SWAP    R0                  ; |
            ANDI    #$007F,     R0      ; |- Todd Y position
            XOR@    R4,         R0      ; |
            MVO@    R0,         R5      ;/
```

And similar updates for the Y position.

```
            ADDI    #6,         R5
```

And another adjustment of 6 locations instead of 7.

```
            MVI@    R4,         R0      ; \_ Player's A register
            MVO@    R0,         R5      ; /
            MVI@    R4,         R0      ; \_ Todd's A register
            MVO@    R0,         R5      ; /
```

And we need to copy the A register area for Todd as well.

```
            ;; ------------------------------------------------------------ ;;
            ;;  Bits to copy into MOB registers.                            ;;
            ;; ------------------------------------------------------------ ;;
@@mobr      DECLE   STIC.mobx_visb      ; make player visible
            DECLE   STIC.mobx_visb      ; make Todd visible

            DECLE   STIC.moby_yres      ; make player 8x16 MOB
            DECLE   STIC.moby_yres      ; make Todd 8x16 MOB

            DECLE   STIC.moba_fg1 + STIC.moba_gram + 0*8    ; Player is blue
            DECLE   STIC.moba_fg2 + STIC.moba_gram + 0*8    ; Todd is red
```

And we need to provide Todd the same kinds of bit pattern constants that we provided for the player graphic. Again, if this is unfamiliar check out part 1 of this tutorial for a bit more detail.

That's it!

Assemble and run it. We now see Todd on the screen! He doesn't move yet, but he's there. The last step of our tutorial series will be to make Todd chase us around the screen.