Vircon32: Programming in assembly

Document date 2024.10.24 Written by Carra

What is this?

This document is a quick guide to program Vircon32 in assembly language. The purpose of this guide is not to teach assembly or programming in general, but rather to describe how the Vircon32 assembler works and serve as a basic assembly reference for the console.

Who should read this guide?

If you just want to make Vircon32 games you don't need this guide: you can make your games in C language with less effort. This document is meant to guide on lower level coding. Either for programmers who want finer control writing games directly in assembly or for C programmers who need to embed ASM sections to improve performance.

Summary

This document is organized into sections, each covering different aspects of the Vircon32 CPU, its assembly language syntax and the use of its assembler program.

Introduction

Using assembly language means that, instead of relying on a programming language that implements abstract concepts (such as functions or loops), you will be writing your programs stating direct instructions for the Vircon32 CPU itself.

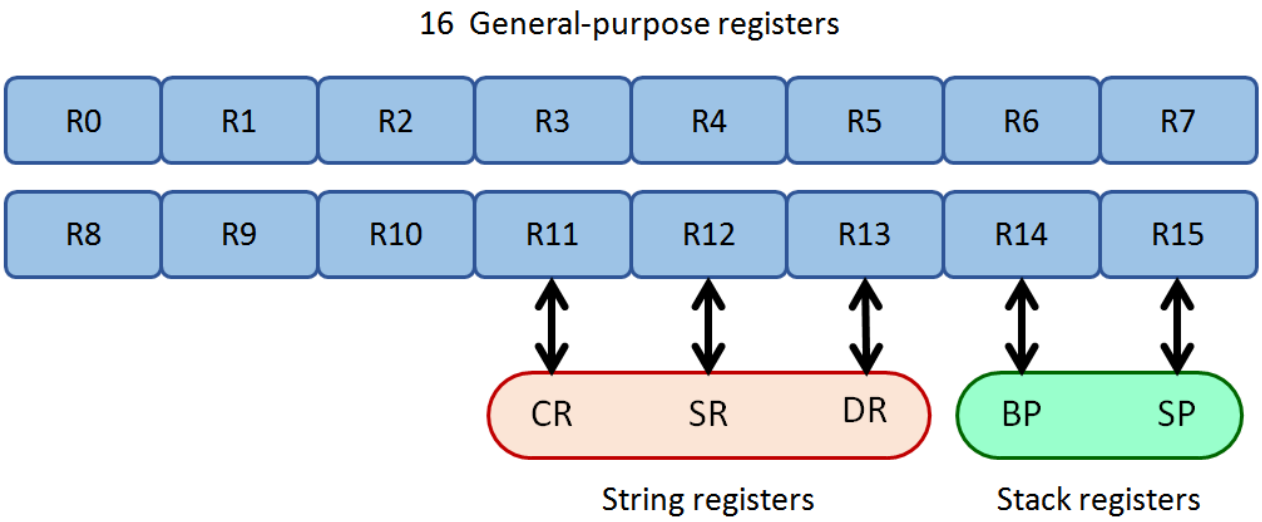
To do this, just knowing the syntax of the assembly language is not enough: you will require some knowledge about the CPU and the console architecture. This section will provide basic details about them to get you started, but if you want deeper knowledge you can read specification documents part 2 (architecture) and part 3 (CPU).

The Vircon32 CPU

Vircon32 has a simplified 32-bit CPU. Its design is loosely based on the x86 family of processors but it only has 64 instructions, with few variants. Every instruction is executed in a single cycle, so performance can be measured just by counting instructions.

The whole console is a pure 32-bit machine. This means registers cannot be used "partially", to handle 8-bit or 16-bit data. The same happens with memory: every address is a 32-bit word. All sizes are counted in words, since no individual bytes are accessible.

The Vircon32 CPU has these registers accessible for the programmer:



There are 16 general-purpose registers from R0 to R15. Each of them is a single 32-bit word that can be interpreted with different formats depending on the instruction currently running. The last 5 of these registers can have special functions in some CPU operations, and each of them receives an "alias" or alternative name that refers to its role in said operations.

Still, note that you won't need to know this in detail until you learn the instruction set.

Stack operations

The CPU implements a hardware stack, which is used to store and extract values following a Last-In, First-Out policy. Its primary function is to be used by call and return instructions when calling subroutines. It uses the following registers:

R14 = BP (Base Pointer)
R15 = SP (Stack Pointer)

The Base Pointer points to stack's current highest position (commonly called the bottom of the stack), while the Stack Pointer keeps track of its lowest position (top of the stack).

String operations

Some CPU instructions are designed to operate on a set of consecutive memory addresses. These sets of data are often referred to as "strings", even if they don't represent text. String operations use the following registers:

R11 = CR (Count Register)
R12 = SR (Source Register)
R13 = DR (Destination Register)

String instructions are not executed just once: CR dictates the number of repetitions. And their data flow will use SR and DR as initial target addresses.

About Vircon32 assembly

Since Vircon32 CPU was based on x86 processors, its assembly language was also chosen to be similar. In general, similar or identical instructions in Vircon32 retain the same name as in x86, like MOV (move data) or RET (return from subroutine). The assembler syntax is also very similar to Intel's ASM notation for x86 (see next section).

Vircon32 assembly language is case-insensitive, so it is equivalent to write any of these lines: `MOV R0, SP`, `MOV r0, sp` or `mov r0, sp`. Note, however, that the Vircon32 assembler has no support for Unicode. It is recommended to keep source code files in either standard ASCII or extended ASCII encoded in Latin-1/ISO 8859-1.

Example program

Before explaining each feature of the Vircon32 assembly language, let's first see a short example. This is a simple program that shows a countdown on screen: it starts at 5 and counts down every second. When it reaches 0 the program ends.

That's it: there is no interaction, no sound and it only uses the default BIOS text font. But it should still be a good first example: it uses most of the language features, its logic is split into small parts, and all steps and decisions are documented in the comments.

```
; execution begins here; since our main program is further
; down, jump to it or else the CPU will run unintended code
JMP _program

; ------------ DECLARATIONS ---------------
; define constants
%define StartSeconds 5              ; start countdown at 5
%define FramesPerSecond 60          ; this console works at constant 60 fps
%define Seconds R0                  ; register R0 will store the current seconds

; declare data
_digit_characters:
  integer '0','1','2','3','4','5'    ; sequence of characters for the countdown

; ------------ SUBROUTINES ---------------
_wait_1_second:
  MOV R1, FramesPerSecond     ; R1 = frames remaining (don't use R0!)
_start_wait_loop:
  WAIT                        ; wait for end of frame
  ISUB R1, 1                  ; now there is 1 less frame to wait
  MOV R2, R1                  ; comparing overwrites, so make a copy
  IGT R2, 0                   ; are frames remaining (i.e. R2) > 0 ? ...
  JT R2, _start_wait_loop     ; ... if so, keep in the loop
  RET                         ; otherwise we are done, go back

_print_seconds:
  OUT GPU_Command, GPUCommand_ClearScreen   ; clear screen (default color is black)
  MOV R1, _digit_characters                 ; R1 points to '0' (don't use R0!)
  IADD R1, Seconds                          ; R1 points to the character for Seconds
  MOV R1, [R1]                              ; load in R1 the pointed value in memory
  OUT GPU_SelectedRegion, R1                ; select our character in BIOS texture
  OUT GPU_Command, GPUCommand_DrawRegion    ; draw it at the current drawing point
  RET                                       ; we are done, go back

; ------------ MAIN PROGRAM ---------------
_program:
  MOV Seconds, StartSeconds    ; initialize our seconds count
  OUT GPU_DrawingPointX, 320   ; set screen X to draw our numbers
  OUT GPU_DrawingPointY, 180   ; set screen Y to draw our numbers
_start_main_loop:
  CALL _print_seconds          ; call subroutine to print seconds
  CALL _wait_1_second          ; call subroutine to wait 1 second
  ISUB Seconds, 1              ; now there is 1 less second remaining
  MOV R1, Seconds              ; comparing overwrites, so make a copy
  IGE R1, 0                    ; are seconds remaining (i.e. R1) >= 0? ...
  JT R1, _start_main_loop      ; ... if so, keep in the loop
  HLT                          ; otherwise end program, countdown ended
```

We can distinguish different types of elements in this program according to how the assembler will interpret them. Here those categories are highlighted using these colors:

Comments: GRAY	Directives: PURPLE
Labels: BLUE	Definitions: BLACK
Data declarations: **PINK**	Literal values: RED
Instructions: BROWN	Registers: **BOLD**
I/O Port names: GREEN	I/O Port values: ORANGE

Comments

Comments are a way to include in the program our own explanations without having the assembler try to interpret them (which would cause errors). That is to say, comments are not part of the program itself.

Comments begin with `;` and finish at the end of the line.

```
; comment in its own line
MOV R2, R1   ; comment after a program line
```

Directives

Directives are a predefined words preceded with `%`. They are special commands for the preprocessor, like defining names and including external files, that affect the whole program from that point.

```
%ifdef DEBUG
  %include "Aux/DebugFunctions.asm"   ; file only included if DEBUG is enabled
%endif
```

The set of supported directives is described later it its own section.

Labels

Labels are identifiers that start with underscore `_`. A program can define labels to mark specific points of the program, and then it can use the label as an address value in instructions. Most common uses are jumps/subroutine calls and loading declared data.

```
; this would be an endless loop that does nothing
_start_loop:       ; define a label
  WAIT
  JMP _start_loop  ; use the label
```

Data declarations

Comments are a way to include in the program our own explanations without having the assembler try to interpret them (which would cause errors). That is to say, comments are not part of the program itself.

```
_gravity:  ; use a label to locate the data later
  integer '0','1'   ; comment after a program line
```

The set of supported data declarations is described later it its own section.

Instructions

An instruction is a minimal action performed by the CPU. Each different CPU action is identified with a short word, like `MOV` to move data. After the identifier, some instructions can take up to 2 operands, separated by commas.

```
MOV R2, 18    ; MOV instruction uses 2 operands
HLT           ; HLT instruction takes no operands
```

The set of instructions for Vircon32 CPU is described later it its own section.

Registers

Each CPU registers is a 1-word storage which value the CPU can manipulate and operate with. Their contents can be interpreted in different formats depending on the context (integer, float, boolean…). Those will be detailed later.

The set of available registers was already defined in the introduction section.

```
IADD CR, R0   ; register 0 is added to the count register (a.k.a. register 11)
```

I/O Port names

The CPU communicates with other chips via a set of in/out ports each of them exposes. By using instructions IN and OUT the CPU can read and write values in a port to achieve different effects. Assembly programs refer to these ports by their names.

```
OUT GPU_DrawingScaleX, 2.0   ; GPU port for scaling on X is set to 2x zoom
```

The set of I/O ports for all console components is described later it its own section.

I/O Port values

Most I/O ports store data in numerical formats, but a few others can only process a small set of specific values. For example port GPU_Command supports only values that the GPU can understand and process as command requests:

```
OUT GPU_Command, GPUCommand_DrawRegion  ; GPU will draw currently selected region
```

The set of I/O ports for all console components is described later in the same section as I/O port names.

Structure of a program

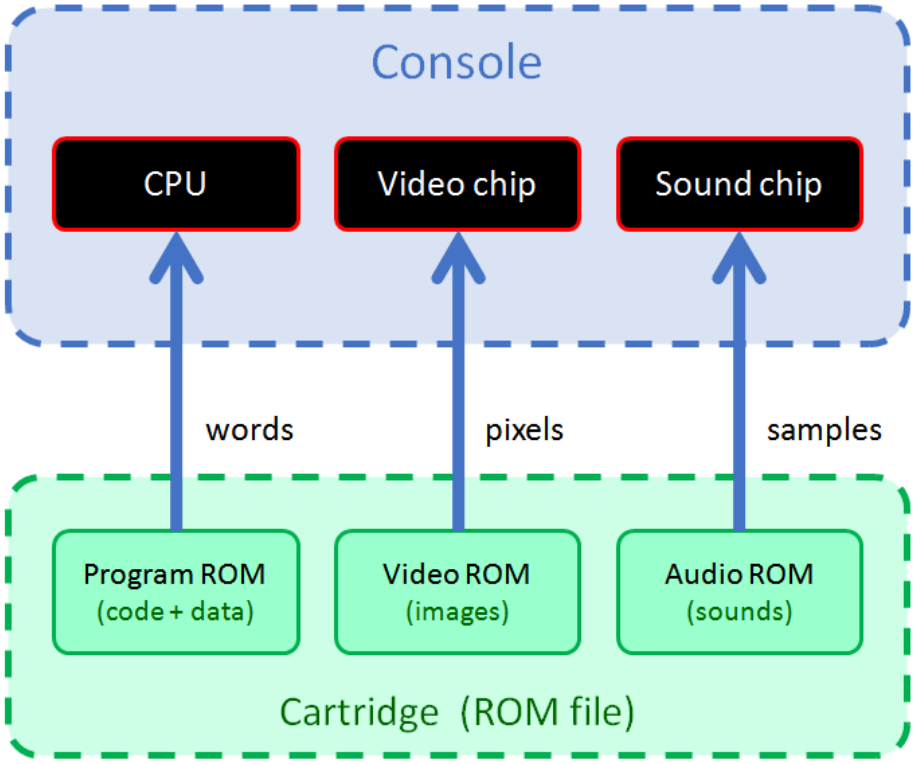
As you may have seen in the example program, a Vircon32 assembly program contains both program instructions and other non-executable data that the program may need at some point. Data can include number literals, strings or even files embedded in the final binary (see datafile declarations later).

All that information will be joined together into a sequence of words. Since this assembler (unlike others) has no concept of "segments" to separate program and data, it is up to the programmer to ensure that no data values get incorrectly executed as instructions.

This can be seen in the example program as well: the program will start execution in the first word, so our program must contain an instruction before any data declarations.

Vircon32 ROM structure

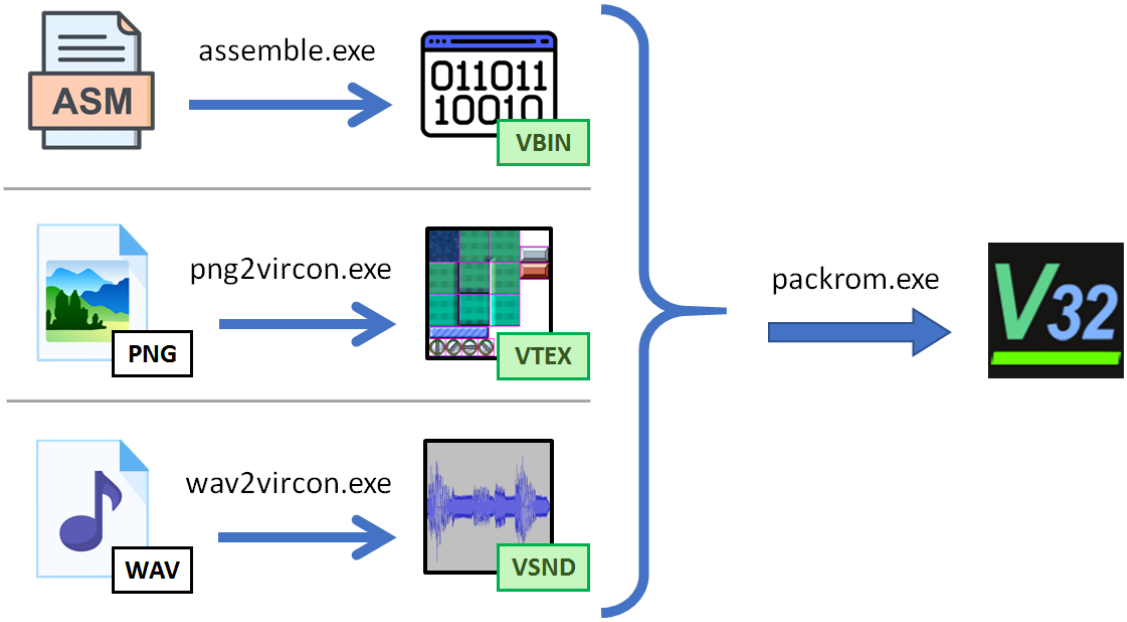
When we "assemble" our program we convert our assembly source code into a sequence of words or binary data for the CPU. But this is still not a Vircon32 full program: a Vircon32 ROM is formed by 3 parts:



Typically a program will include a set of images and sounds in the cartridge in addition to the program binary. The example program we saw does not actually need to include any asset, because it only uses the BIOS text font. But even then it still needs to "pack" the program into a ROM format and indicate it has no audio ROM or video ROM.

The build process

The full process to go from an assembly source code to an executable Vircon32 ROM will have the steps shown in this diagram:



As expected, for our assembly source code we will call the assembler to convert our assembly source code into a program ROM. And similarly, if we want to use images and sounds in our program, the set of Vircon32 dev tools also includes converter programs to import into our projects PNG images and WAV sounds. All of these programs are command line tools, so we can easily automate the process writing a short script.

Finally, another tool will pack those 3 components (or less: video and audio ROMs are optional) into the final executable ROM. For this we will need to tell our packer program where all the assets to include are located. This is done with a ROM definition XML. Here is a simple example for a program with 2 textures and 1 sound:

```
<rom-definition version="1.0">
    <rom type="cartridge" title="Game Test" version="1.0" />
    <binary path="converted/TestProgram.vbin" />
    <textures>
        <texture path="converted/TextureBackground.vtex" />
        <texture path="converted/TextureCharacter.vtex" />
    </textures>
    <sounds>
        <sound path="converted/SoundJump.vsnd" />
    </sound>
</rom-definition>
```

Note that you need to declare the paths for the already converted assets! If you include a PNG image in your program the XML must not contain the path for the PNG itself, but the path of the VTEX file created by the PNG converter.

Using the assembler

The Vircon32 assembler is a command-line tool, so you invoke it from the console. You will probably want to add your dev tools folder to your environment variables so that the `assemble` command becomes available anywhere.

These are the options available for the assembler, displayed by using `assemble --help`:

```
USAGE: assemble [options] file
Options:
  --help       Displays this information
  --version    Displays program version
  -o <file>    Output file, default name is the same as input
  -b           Assembles the code as a BIOS
  -v           Displays additional information (verbose)
Also, the following options are accepted for compatibility
but have no effect: -s
```

The most basic way to use it is just: `assemble program.asm`. Since we specified no output file via option `-o` it will use the same name and (when successful) create “program.vbin”.

Console data types

As stated before, even though all data within the console are always 32-bit words, the console will interpret each word with different data types. This section describes the possible formats and interpretations of a word:

Integer

The word is interpreted as a 32-bit signed integer. This is equivalent to the C data type “int32_t”. Note that this is the only integer data type available: there are no unsigned integers, and there are no variants for different sizes.

Boolean

The word is also taken as an integer number, but this number is then interpreted as a boolean value: If the integer value is 0 the word is taken as false, and any other value will be interpreted as true. This is equivalent to the C data type “bool”.

When producing a boolean, Vircon32 always uses the values 1 for “true” and 0 for “false”. These are the values you get when using `true` and `false` in assembly language.

Float

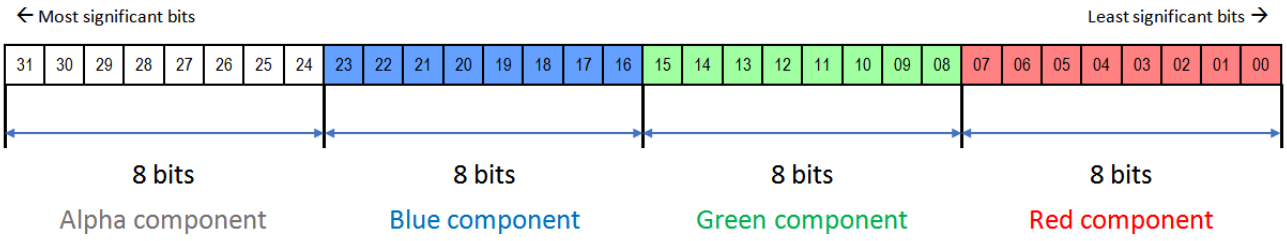
The word is interpreted as a 32-bit floating point number. This is equivalent to the C data type “float”. However Vircon32 does not support float special values, such as NaN (not a number) or infinities. Using them them in a program will produce undetermined behavior.

Binary

The word is taken to be just a sequence of 32 bits, independent from each other and with no meaning as a group. This interpretation is usually only used by binary operations such as AND/OR. This is equivalent to the way C treats integers in such operations.

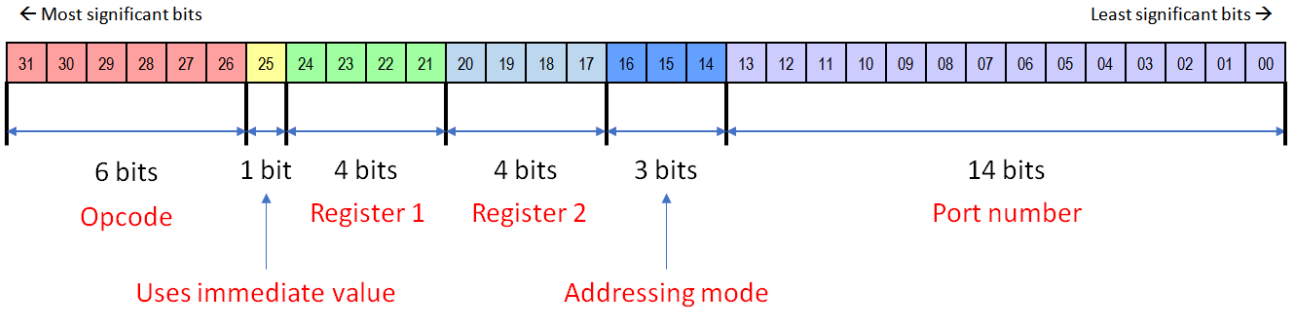
GPU color

When the GPU uses a word as a color, it will interpret it as RGBA with 4 8-bit fields in that order. This is equivalent to the RGBA8 format used by many graphic cards.



CPU instruction

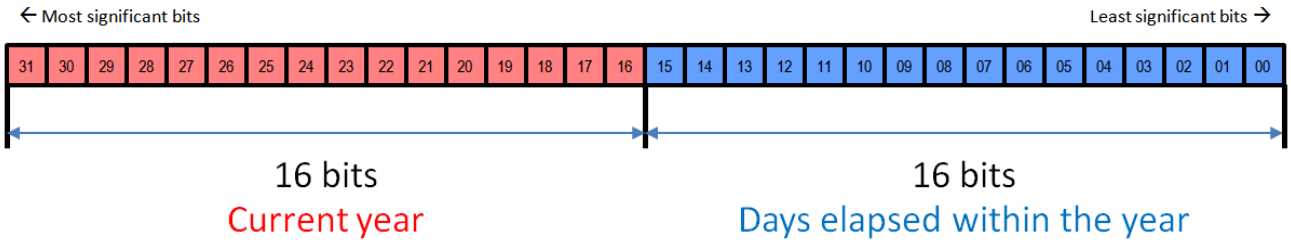
When the CPU executes a word as an instruction, it will interpret it as an instruction opcode with a series of complementary fields, as shown here:



You don’t need to know this data format to program in assembly. It is enough for you to know that the instruction itself also includes any used registers and I/O port numbers.

Date

This is the internal date format used by the console's timer. It groups 2 fields, each encoded as a 16-bit unsigned integer, as follows:



The upper 16 bits are simply the number for the current year (e.g: 2022). Lower 16 bits represent the number of days currently elapsed for that year. For instance: if the lower bits contain 3 it will now be January 4th (3 days of this year did already pass).

Declaring data

While the main component of a program are instructions, those often need to work with data. You can declare data values in different formats using the keywords that will be presented here. All of those literal values will effectively be constants, since they will be stored in a cartridge with read-only memory.

Declaring floats

Use keyword `float` to declare one or more floating point values. Note that the assembler does not support float exponential notation such as 0.123E-4. When declaring float values A variable is declared with a type and a name, and can optionally be initialized with a value:

```
_pi_multiples:  ; use labels to access your data later
  float 3.1416, 6.2832, 9.4248

_scale_factor:
  float 2       ; it will be correctly converted to float 2.0
```

Declaring integers

With keyword `integer` you can declare one or more integer values. Note that, since there is only one integer type in Vircon32, you will also use integers to represent non-numerical values such as booleans (true = 1, false = 0) and characters (for example letter 'A'). Therefore the following integer declarations are all valid:

```
integer 3, -17          ; regular integer values
integer 0xFFA02         ; hexadecimal format
integer true, false     ; boolean values are integers
integer 'H', '$'        ; characters are also integers
```

Characters can be escaped when needed, with the same syntax used in C language. Still, note that character numbers only support the ASCII range (00h to FFh):

```
integer '\n', '\\', '\''  ; new line, backslash and single quote
integer '\x4B'            ; character number 0x4B = 'K'
```

Declaring strings

Using keyword `string` you can declare a single text string. Multiple strings are not supported in a same declaration. Characters within a string can be escaped in the same way as when declaring single characters and, again, the same way as in C language.

```
string "Hello \"cruel\" world"  ; has to escape the double quotes
string "Greetings"              ; a second string needs another line
```

Declaring label addresses

Let's say you want to declare a memory address as a value to use it later. If you know the specific address beforehand (for instance, the start of the memory card), you can declare it as a regular integer. But the assembler also lets you use labels to declare addresses that will be resolved during the assembly process. These declarations use keyword `pointer`, since that is the most common use for these data values.

```
; with this we could have pointers to the float
; values we declared in the previous page
pointer _pi_multiples, _scale_factor
```

Embedding files as data

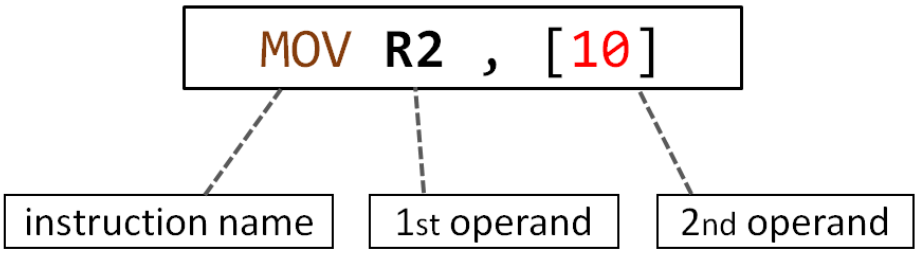
Our program may need to use large data, like a tile map. In this case it can better to store it in an external file and let the assembler embed it directly into the program binary. The assembler supports this with keyword `datafile`:

```
; the assembler will access this file and embed its contents
datafile "Data/Level1.map"
```

The path to the file is given as a string, which can have escaped characters same as regular string values. Note that, to be correctly embedded, the file size in bytes will need to be a multiple of 4. Otherwise it won't fit correctly in 32-bit words.

CPU instructions

Assembly programs are formed by sequences of instructions. Each instruction is a basic action the CPU can perform. Some instructions are self-contained (like HLT to halt the CPU), while others require up to 2 operands to define the action. A complete example could look like this:



This instruction, for example, will take the contents of memory address 10 (square brackets indicate a memory position) and copy them into register R2. Each instruction type always uses the same number of operands, although they may have variants that can use operands of different type.

In assembly programs each instruction must be in a separate line. You can leave empty lines between them but it is not possible to concatenate multiple instructions in a same line similarly to how languages like C allow concatenating different statements.

This section will provide a short description of all Vircon32 CPU instructions, grouped into sections, along with examples of all usable variants of each instruction.

CPU control	
Instruction	Short description
HLT Halt	Halts the CPU, stopping the program. This will not clear the screen or stop any playing sounds. `HLT  ; no operands`
WAIT	The CPU waits until next frame. `WAIT  ; no operands`

Jump instructions

Instruction	Short description

JMP
Jump
	Execution jumps to the indicated address
	`JMP 0x20000015  ; jump to this memory address`
	`JMP _mylabel  ; jump to this label`
	`JMP R1  ; jump to address = value of R1`

CALL	Call a subroutine at the indicated address. When the subroutine returns, execution will resume after the call instruction.
	`CALL 0x20000015  ; call sub at this memory address`
	`CALL _mysub  ; call sub at this label`
	`CALL R1  ; call sub at address = value of R1`

RET Return	Returns from current subroutine. Execution resumes after the instruction that called it. `RET  ; no operands`

JT
Jump if True
	If its first operand is true (non zero) it will perform a jump to the address in its second operand.
	`JT R1, 0x20000015  ; if R1 != 0 jump to this address`
	`JT R1, _mylabel  ; if R1 != 0 jump to this label`
	`JT R1, R2  ; if R1 != 0 jump to value of R2`

JF
Jump if False
	If its first operand is false (zero) it will perform a jump to the address in its second operand.
	`JF R1, 0x20000015  ; if R1 == 0 jump to this address`
	`JF R1, _mylabel  ; if R1 == 0 jump to this label`
	`JF R1, R2  ; if R1 == 0 jump to value of R2`

Integer comparisons

Instruction	Short description

IEQ
Integer Equal
	Checks if its 2 operands, interpreted as integers, are equal.
	Stores the boolean result in the first operand.
	`IEQ R0, -19  ; R0 = (R0 == -19)?`
	`IEQ R0, R1  ; R0 = (R0 == R1)?`

INE
Integer
Not Equal
	Checks if its 2 operands, interpreted as integers, are different.
	Stores the boolean result in the first operand.
	`INE R0, -19  ; R0 = (R0 != -19)?`
	`INE R0, R1  ; R0 = (R0 != R1)?`

IGT
Integer
Greater Than
	Checks if its first operand is greater than the second, interpreted as integers. Stores the boolean result in the first operand.
	`IGT R0, -19 ; R0 = (R0 > -19)?`
	`IGT R0, R1 ; R0 = (R0 > R1)?`

IGE
Integer
Greater or Equal
	Checks if its first operand is greater or equal than the second, taken as integers. Stores the boolean result in the first operand.
	`IGE R0, -19 ; R0 = (R0 >= -19)?`
	`IGE R0, R1 ; R0 = (R0 >= R1)?`

ILT
Integer
Less Than
	Checks if its first operand is less than the second, interpreted as integers. Stores the boolean result in the first operand.
	`ILT R0, -19 ; R0 = (R0 < -19)?`
	`ILT R0, R1 ; R0 = (R0 < R1)?`

ILE
Integer
Less or Equal
	Checks if its first operand is less or equal than the second, taken as integers. Stores the boolean result in the first operand.
	`ILE R0, -19 ; R0 = (R0 <= -19)?`
	`ILE R0, R1 ; R0 = (R0 <= R1)?`

Float comparisons	
Instruction	Short description

FEQ
Float Equal
	Checks if its 2 operands, interpreted as floats, are equal.
	Stores the boolean result in the first operand.
	`FEQ R0, 19.0 ; R0 = (R0 == 19.0)?`
	`FEQ R0, R1 ; R0 = (R0 == R1)?`

FNE
Float
Not Equal
	Checks if its 2 operands, interpreted as floats, are different.
	Stores the boolean result in the first operand.
	`FNE R0, 19.0 ; R0 = (R0 != 19.0)?`
	`FNE R0, R1 ; R0 = (R0 != R1)?`

FGT
Float
Greater Than
	Checks if its first operand is greater than the second, interpreted as floats. Stores the boolean result in the first operand.
	`FGT R0, 19.0 ; R0 = (R0 > 19.0)?`
	`FGT R0, R1 ; R0 = (R0 > R1)?`

FGE
Float
Greater or Equal
	Checks if its first operand is greater or equal than the second, taken as floats. Stores the boolean result in the first operand.
	`FGE R0, 19.0 ; R0 = (R0 >= 19.0)?`
	`FGE R0, R1 ; R0 = (R0 >= R1)?`

FLT
Float
Less Than
	Checks if its first operand is less than the second, interpreted as floats. Stores the boolean result in the first operand.
	`FLT R0, 19.0 ; R0 = (R0 < 19.0)?`
	`FLT R0, R1 ; R0 = (R0 < R1)?`

FLE
Float
Less or Equal
	Checks if its first operand is less or equal than the second, taken as floats. Stores the boolean result in the first operand.
	`FLE R0, 19.0 ; R0 = (R0 <= 19.0)?`
	`FLE R0, R1 ; R0 = (R0 <= R1)?`

Data movement	
Instruction	Short description

MOV
Move
	Moves (copies) the value from its second operand into the first.
	`MOV R0, 75 ; R0 = 75`
	`MOV R0, _label ; R0 = label address`
	`MOV R0, [75] ; R0 = Memory[ 75 ]`
	`MOV R0, [_label] ; R0 = Memory[ label address ]`
	`MOV R0, R1 ; R0 = R1`
	`MOV R0, [R1] ; R0 = Memory[ R1 ]`
	`MOV R0, [R1+14] ; R0 = Memory[ R1 + 14 ]`
	`MOV [32], R1 ; Memory[ 32 ] = R1`
	`MOV [R0], R1 ; Memory[ R0 ] = 75`
	`MOV [R0+14], R1 ; Memory[ R0 + 14 ] = R1`

LEA
Load Effective Address
	Receives a memory position as a second operand, and copies that memory position (not its contents!) into the first operand.
	`LEA R0, [R1] ; R0 = R1`
	`LEA R0, [R1-19] ; R0 = R1 - 19`

PUSH	Pushes a register's contents onto the top of the CPU stack. `PUSH R0 ; Stack.Push( -19 )`
POP	Takes the top value in the CPU stack and stores it in a register. `POP R0 ; R0 = Stack.Pop()`
IN	Reads a value from an I/O control port and stores it in a register. `IN R1, INP_GamepadLeft ; R1 = State of d-pad left`
OUT	Writes the value in a register into an I/O control port. `OUT GPU_SelectedRegion, R0 ; Selected Region = R0`

String operations

Instruction	Short description
MOVS Move String	Implements a loop, similar to a hardware version of memcpy(), to keep copying values between consecutive memory positions: • Memory[DR] = Memory[SR] • SR (source reg.) and DR (destination reg.) are incremened • CR (count register) is decremented • This instruction will keep repeating while CR > 0 `MOVS  ; no operands`
SETS Set String	Implements a loop, similar to a hardware version of memset(), to keep setting consecutive memory positions to a same value: • Memory[DR] = SR • DR (destination register) is incremened • CR (count register) is decremented • This instruction will keep repeating while CR > 0 `SETS  ; no operands`
CMPS Compare String	Implements a loop, similar to a hardware version of memcmp(), to keep comparing values between consecutive memory positions. Stores the result of the comparison in the operand register: • Operand Register = Memory[DR] – Memory[SR] • If Operand Register != 0 then end the loop • SR (source reg.) and DR (destination reg.) are incremened • CR (count register) is decremented • This instruction will keep repeating while CR > 0 `CMPS R1  ; R1 = result of last compared word pair`

Data conversion

Instruction	Short description
CIF Convert Integer to Float	Performs a value conversion in a register: reads its contents as an integer and converts that value to float format. `CIF R0  ; R0 = (float) R0`
CFI Convert Float to Integer	Performs a value conversion in a register: reads its contents as a float and converts that value to integer format. `CFI R0  ; R0 = (int) R0`

CIB Convert Integer to Boolean	Performs a value conversion in a register: reads its contents as an integer and converts that value to a boolean (either 0 or 1). `CIB R0 ; if R0 != 0 then R0 = 1`
CFB Convert Float to Boolean	Performs a value conversion in a register: reads its contents as a float and converts that value to a boolean (either 0 or 1). `CFB R0 ; if R0 != 0.0 then R0 = 1 else R0 = 0`

Binary operations	
Instruction	Short description
NOT	Performs a binary NOT on a register by inverting all its bits. `NOT R0 ; R0 = NOT R0`

AND	Performs a binary AND (bit by bit) between its operands. Stores the result in its first operand.
	`AND R0, 35 ; R0 = R0 AND 35`
	`AND R0, R1 ; R0 = R0 AND R1`

OR	Performs a binary OR (bit by bit) between its operands. Stores the result in its first operand.
	`OR R0, 35 ; R0 = R0 OR 35`
	`OR R0, R1 ; R0 = R0 OR R1`

XOR
Exclusive OR
	Performs a binary exclusive OR (bit by bit) between its operands. Stores the result in its first operand.
	`XOR R0, 35 ; R0 = R0 XOR 35`
	`XOR R0, R1 ; R0 = R0 XOR R1`

BNOT Boolean NOT	Performs a boolean NOT on a register, by interpreting it as a boolean and then negating it. `BNOT R0 ; if R0 == 0 then R0 = 1 else R0 = 0`

SHL
Bit Shift Left
	Shifts the bits of its first operand to the right. 2nd operand is the number of positions to shift (negative means shift to the right).
	`SHL R0, 3 ; R0 = R0 << 3`
	`SHL R0, R1 ; R0 = R0 << R1`

Integer arithmetic

Instruction	Short description

IADD
Integer Add
	Interprets both operands as integers and adds them. Stores the result in its first operand.
	`IADD R0, 27  ; R0 = R0 + 27`
	`IADD R0, R1  ; R0 = R0 + R1`

ISUB
Integer Subtract
	Interprets both operands as integers and subtracts the second from the first. Stores the result in its first operand.
	`ISUB R0, 27  ; R0 = R0 - 27`
	`ISUB R0, R1  ; R0 = R0 - R1`

IMUL
Integer Multiply
	Interprets both operands as integers and multiplies them. Stores the result in its first operand.
	`IMUL R0, 27  ; R0 = R0 * 27`
	`IMUL R0, R1  ; R0 = R0 * R1`

IDIV
Integer Divide
	Interprets both operands as integers and divides the first by the second. Stores the integer part of the result in its first operand.
	`IDIV R0, 27  ; R0 = R0 / 27`
	`IDIV R0, R1  ; R0 = R0 / R1`

IMOD
Integer Modulus
	Interprets both operands as integers and divides the first by the second. Stores the remainder in its first operand.
	`IMOD R0, 27  ; R0 = Remainder( R0 / 27 )`
	`IMOD R0, R1  ; R0 = Remainder( R0 / R1 )`

ISGN Integer Sign Change	Interprets its operands as an integer and changes its sign. `ISGN R0  ; R0 = –R0`

IMIN
Integer Minimum
	Interprets both operands as integers and stores the minimum of the 2 in its first operand.
	`IMIN R0, 27  ; R0 = Minimum( R0, 27 )`
	`IMIN R0, R1  ; R0 = Minimum( R0, R1 )`

IMAX
Integer Maximum
	Interprets both operands as integers and stores the maximum of the 2 in its first operand.
	`IMAX R0, 27  ; R0 = Maximum( R0, 27 )`
	`IMAX R0, R1  ; R0 = Maximum( R0, R1 )`

IABS Integer Absolute Value	Interprets a register as an integer and obtains its absolute value. `IABS R0  ; R0 = ABS( R0 )`

Float arithmetic

Instruction	Short description

FADD
Float Add
	Interprets both operands as floats and adds them. Stores the result in its first operand.
	`FADD R0, 13.2  ; R0 = R0 + 13.2`
	`FADD R0, R1  ; R0 = R0 + R1`

FSUB
Float Subtract
	Interprets both operands as floats and subtracts the second from the first. Stores the result in its first operand.
	`FSUB R0, 13.2  ; R0 = R0 - 13.2`
	`FSUB R0, R1  ; R0 = R0 - R1`

FMUL
Float Multiply
	Interprets both operands as floats and multiplies them. Stores the result in its first operand.
	`FMUL R0, 13.2  ; R0 = R0 * 13.2`
	`FMUL R0, R1  ; R0 = R0 * R1`

FDIV
Float Divide
	Interprets both operands as floats and divides the first by the second. Stores the result in its first operand.
	`FDIV R0, 13.2  ; R0 = R0 / 13.2`
	`FDIV R0, R1  ; R0 = R0 / R1`

FMOD
Float Modulus
	Interprets both operands as floats and calculates the “floating point remainder” of dividing the first by the second. Stores the result in its first operand.
	`FMOD R0, 13.2  ; R0 = fmod( R0, 13.2 )`
	`FMOD R0, R1  ; R0 = fmod( R0, R1 )`

FSGN Float Sign Change	Interprets its operands as a float and changes its sign. `FSGN R0  ; R0 = -R0`

FMIN
Float Minimum
	Interprets both operands as floats and stores the minimum of the 2 in its first operand.
	`FMIN R0, 13.2  ; R0 = Minimum( R0, 13.2 )`
	`FMIN R0, R1  ; R0 = Minimum( R0, R1 )`

FMAX
Float Maximum
	Interprets both operands as integers and stores the maximum of the 2 in its first operand.
	`FMAX R0, 13.2  ; R0 = Maximum( R0, 13.2 )`
	`FMAX R0, R1  ; R0 = Maximum( R0, R1 )`

FABS Float Absolute Value	Interprets a register as a float and obtains its absolute value. `FABS R0  ; R0 = ABS( R0 )`

Extended float operations	
Instruction	Short description
FLR Floor	Interprets a register as float and rounds it to downwards to an integer. The value is still a float, it is not converted to integer. `FLR R0 ; R0 = floor( R0 )`
CEIL Ceiling	Interprets a register as float and rounds it to upwards to an integer. The value is still a float, it is not converted to integer. `CEIL R0 ; R0 = ceil( R0 )`
ROUND	Interprets a register as float and rounds it to the nearest integer. The value is still a float, it is not converted to integer. `ROUND R0 ; R0 = round( R0 )`
SIN Sine	Interprets a register as float and obtains its sine. `SIN R0 ; R0 = sin( R0 )`
ACOS Arc Cosine	Interprets a register as float and obtains its arc cosine. `ACOS R0 ; R0 = acos( R0 )`
ATAN2 2-Argument Arc Tangent	Interprets 2 registers as floats and obtains the arc tangent of the angle given by vector: { DeltaX = 2nd operand, DeltaY = 1st operand }. `ATAN2 R0, R1 ; R0 = atan2( R0, R1 )`
LOG Logarithm	Interprets a register as float and obtains its natural logarithm. `LOG R0 ; R0 = log( R0 )`
POW Power	Interprets both registers as floats and obtains the result of raising the first to the power of the second. `POW R0, R1 ; R0 = Pow( R0, R1 )`

Assembler directives

Same as in C language, the Vircon32 assembler includes a preprocessor that can perform a series of text operations to alter our source code. These operations are invoked in the code using directives, which are a set of predefined identifiers starting with `%`.

This section will list all available directives and explain their use. Note that, like all other features in the assembler, each directive needs to be written in its own single line.

Definitions

With directive `%define` we can declare identifiers and assign them a text value. Then, when we write that identifier later, it will get replaced with that text. For instance:

```
%define Pi  3.1416
%define ClearScreen  OUT GPU_Command, GPUCommand_ClearScreen

; use our definitions
_test:
  MOV R0, Pi
  ClearScreen   ; using this will run the instruction
```

Identifiers can also be defined with no value. This can be useful to use flags in conditional directives (see below).

```
%define DEBUG    ; no value: only to be used as a flag for conditionals later
```

We can also use a definition only at certain parts of our code, and remove it later. Directive `%undef` lets us undefine a previously defined identifier.

```
%define PositionX [10]   ; position X is stored in memory address 10

; definition only usable in this subroutine
_increment_position_x:
  MOV R0, PositionX
  IADD R0, 1
  MOV PositionX, R0
  RET

%undef PositionX
```

Conditionals

We can use directives to choose if certain parts of the source code are included in our program or not. Here the possible conditions we can use are only `%ifdef` (tests if some identifier has been defined) and `%ifndef` (tests if it has not been defined). After one of these directives we must finish the block with an `%endif` directive. This will form an enclosed block, and every line contained between the 2 directives will be included in our program only if the condition is met:

```
%ifdef DEBUG
  CALL _draw_hitboxes   ; these should not be seen in the final release
%endif
```

We can also create a second block using `%else`, to include a different code when the condition is not met. You can also nest different conditional blocks!

```
%ifdef DEBUG
  %include "DebugRoutines.asm"    ; debug routines do extra stuff for our testing
%else
  %include "ReleaseRoutines.asm"  ; release routines are faster
%endif
```

Messages

We can use directives to output messages while our program is being assembled. Directive `%warning` will output the message as a warning, which will not stop the assembly. In contrast, using `%error` the message is treated as an error and will stop the process.

```
%warning "This message lets assembly continue"
%error "But not THIS one"
MOV R0, 17   ; the assembler will not reach this line
```

Includes

The `%include` directive will look for an external file and add its content at that point. This makes it possible to split our programs into multiple files or reuse subroutines:

Main file (program.asm)

```
%include "Routines/increment.asm"

; now we can call routine _increment_x
CALL _increment_x
```

Included file (Routines/ movement.asm)

```
_increment_x:
  MOV R0, [14]   ; variable x stored at memory position 14
  IADD R0, 1
  MOV [14], R0
  RET
```

The string used by %include specifies a file path relative to the main file, and if needed will use folder separators or ".." to go to parent folders, same as any other path.

An important distinction: Do not confuse this file inclusion with what datafile keyword does! Using datafile embeds a file in the binary program file, after assembling. However, %include embeds a file in the assembly source code.

I/O control ports

Instructions `IN` and `OUT` work with the I/O control ports exposed by the console chips to communicate with the CPU. Vircon32 assembly language identifies these ports by name, so this section will provide all port names for each chip, along with a small description of what their value represents and the data types they will expect.

A few of the ports will only accept a small list of valid values. These ports list their expected data type as "Enum". You will also find the name of those values here.

GPU control ports			
Port name	Access	Format	Short description
GPU_Command	Write	Enum	Input port to receive GPU draw commands
GPU_RemainingPixels	Read	Integer	Pixels the GPU can still draw in this frame
GPU_ClearColor	R & W	Color	Color used to clear the screen
GPU_MultiplyColor	R & W	Color	Modulation color used when drawing regions
GPU_ActiveBlending	R & W	Enum	Blending mode currently active
GPU_SelectedTexture	R & W	Integer	Currently selected texture ID
GPU_SelectedRegion	R & W	Integer	Currently selected region ID (in selected texture)
GPU_DrawingPointX	R & W	Integer	Screen X coordinate to draw regions
GPU_DrawingPointY	R & W	Integer	Screen Y coordinate to draw regions
GPU_DrawingScaleX	R & W	Float	Scale factor in X to draw regions [1]. Flips X if negative
GPU_DrawingScaleY	R & W	Float	Scale factor in Y to draw regions [1]. Flips Y if negative
GPU_DrawingAngle	R & W	Float	Rotation angle to draw regions [2], in radians. Grows anticlockwise when positive
GPU_RegionMinX	R & W	Integer	Selected region's minimum X coordinate
GPU_RegionMinY	R & W	Integer	Selected region's minimum Y coordinate
GPU_RegionMaxX	R & W	Integer	Selected region's maximum X coordinate
GPU_RegionMaxY	R & W	Integer	Selected region's maximum Y coordinate
GPU_RegionHotspotX	R & W	Integer	Selected region's hotspot X coordinate
GPU_RegionHotspotY	R & W	Integer	Selected region's hotspot Y coordinate

NOTES:

[1]: These scale factors only apply when regions are drawn with scaling ON.
[2]: This angle applies when regions are drawn with rotation ON.

GPU commands (for GPU_Command)	
Command name	Short description
GPUCommand_ClearScreen	Draw screen with current clear color
GPUCommand_DrawRegion	Draw selected region (scaling OFF, rotation OFF)
GPUCommand_DrawRegionZoomed	Draw selected region (scaling ON, rotation OFF)
GPUCommand_DrawRegionRotated	Draw selected region (scaling OFF, rotation ON)
GPUCommand_DrawRegionRotozoomed	Draw selected region (scaling ON, rotation ON)

GPU blending modes (for GPU_ActiveBlending)	
Blending mode name	Short description
GPUBlendingMode_Alpha	Alpha blend: transparency but no color changes
GPUBlendingMode_Add	Additive blend: colors are added, lighting effect
GPUBlendingMode_Subtract	Subtractive blend: colors are subtracted, shadow effect

SPU control ports			
Port name	Access	Format	Short description
SPU_Command	Write	Enum	Input port to receive SPU sound commands
SPU_GlobalVolume	R & W	Float	Global volume multiplier for all channels
SPU_SelectedSound	R & W	Integer	Currently selected sound ID
SPU_SelectedChannel	R & W	Integer	Currently selected sound channel ID
SPU_SoundLength	Read	Integer	Selected sound's number of samples
SPU_SoundPlayWithLoop	R & W	Boolean	Selected sound loops when playing?
SPU_SoundLoopStart	R & W	Integer	Selected sound's first sample to be looped
SPU_SoundLoopEnd	R & W	Integer	Selected sound's last sample to be looped
SPU_ChannelState	Read	Enum	Selected channel's state (stop/pause/play)
SPU_ChannelAssignedSound	R & W	Integer	Selected channel's assigned sound ID
SPU_ChannelVolume	R & W	Float	Selected channel's volume multiplier
SPU_ChannelSpeed	R & W	Float	Selected channel's playback speed multiplier
SPU_ChannelLoopEnabled	R & W	Boolean	Selected channel has loop enabled?
SPU_ChannelPosition	R & W	Integer	Selected channel's current playback position

SPU commands (for SPU_Command)	
Command name	Short description
SPUCommand_PlaySelectedChannel	Selected channel will: • If stopped: begin playing its assigned sound • If paused: resume previous playback • If already playing: restart its playback
SPUCommand_PauseSelectedChannel	Selected channel pauses playback. No effect if stopped
SPUCommand_StopSelectedChannel	Selected channel stops playback
SPUCommand_PauseAllChannels	All channels that were playing are paused
SPUCommand_ResumeAllChannels	All channels that were paused resume playback
SPUCommand_StopAllChannels	All channels that were playing or paused are stopped

Sound channel states (for SPU_ChannelState)	
State name	Short description
SPUChannelState_Stopped	Channel is stopped
SPUChannelState_Paused	Channel is paused and can continue playback
SPUChannelState_Playing	Channel is currently producing sound output

Gamepad controller control ports			
Port name	Access	Format	Short description
INP_SelectedGamepad	R & W	Integer	Currently selected gamepad ID
INP_GamepadConnected	Read	Boolean	Is a gamepad present in the selected port?
INP_GamepadLeft	Read	Integer	State of selected gamepad's left direction
INP_GamepadRight	Read	Integer	State of selected gamepad's right direction
INP_GamepadUp	Read	Integer	State of selected gamepad's up direction
INP_GamepadDown	Read	Integer	State of selected gamepad's down direction
INP_GamepadButtonStart	Read	Integer	State of selected gamepad's button Start
INP_GamepadButtonA	Read	Integer	State of selected gamepad's button A
INP_GamepadButtonB	Read	Integer	State of selected gamepad's button B
INP_GamepadButtonX	Read	Integer	State of selected gamepad's button X
INP_GamepadButtonY	Read	Integer	State of selected gamepad's button Y
INP_GamepadButtonL	Read	Integer	State of selected gamepad's button L
INP_GamepadButtonR	Read	Integer	State of selected gamepad's button R

Timer control ports			
Port name	Access	Format	Short description
TIM_CurrentDate	Read	Date	Current date, in internal date format
TIM_CurrentTime	Read	Integer	Current time of day, expressed as elapsed seconds
TIM_FrameCounter	Read	Integer	Frames elapsed since last console start or reset
TIM_CycleCounter	Read	Integer	CPU cycles elapsed within current frame

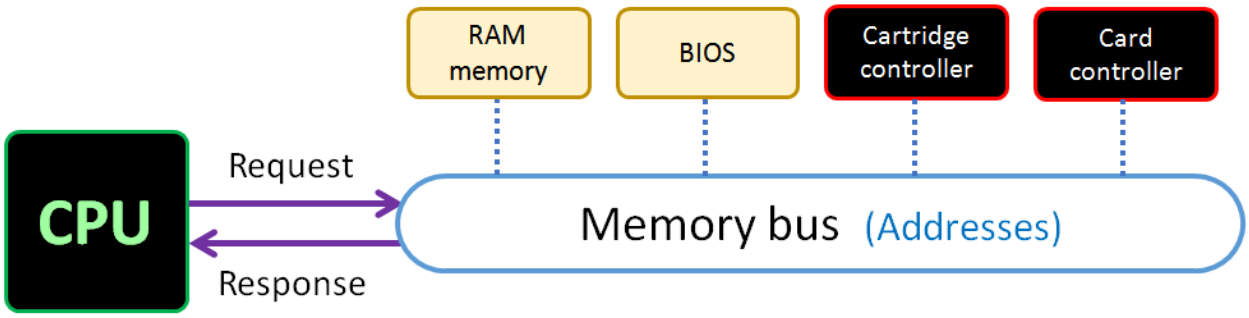
Random Number Generator control ports			
Port name	Access	Format	Short description
RNG_CurrentValue	R & W	Integer	Current seed/value in the pseudo-random sequence

Memory Card Controller control ports			
Port name	Access	Format	Short description
MEM_CardConnected	Read	Boolean	Is a memory card present in the card port?

Cartridge Controller control ports			
Port name	Access	Format	Short description
CAR_CartridgeConnected	Read	Boolean	Is a cartridge present in the cartridge port?
CAR_ProgramROMSize	Read	Integer	Current cartridge's program ROM size, in words
CAR_NumberOfTextures	Read	Integer	Current cartridge's number of textures
CAR_NumberOfSounds	Read	Integer	Current cartridge's number of sounds

Memory mapping

There can be up to 4 devices in a Vircon32 console that have memory accessible to the CPU. Each of them connects to the memory bus and the CPU can request to read or write specific memory positions from their memory.



This table shows the range of memory addresses for each of the devices, when they are present. RAM memory and the BIOS are always present, but the cartridge and especially the memory card will not always be.

Connected device	Address range
RAM memory	0x00000000 – 0x003FFFFF
BIOS program ROM	0x10000000 – 0x10000000 (max)
Cartridge program ROM	0x20000000 – 0x27FFFFFF (max)
Memory card RAM	0x30000000 – 0x30003FFF

RAM memories allow read and write access, while ROM memories are read-only. Note that the size for BIOS and cartridge memories will be variable. Cartridges are interchangeable, and different Vircon32 systems may have different BIOSes.

For RAM memory it is useful to know that the CPU stack begins at the last RAM address and grows towards lower addresses. So it is a good practice to use RAM in your programs starting from address 0 and and growing up, so that they will not conflict.

General tips

This last section contains a few practical tips that will be useful if you are learning to program in Vircon32 assembly. While this document is intended as a guide/reference text rather than a proper assembly course, we will still see some general pointers that will come in handy and save you from some common mistakes.

Variables in memory

If you know other higher-level programming languages (which you should before attempting assembly!) you may have wondered how you declare variables in this assembly language. The answer is: you don't!

You must choose for yourself where you will store each of your variables, either in memory or in registers. But since register contents can be volatile (a subroutine may have modified it without you counting on it), most of the time you will want to assign your variables a memory address.

You can still have "pseudo-variables" by using `%define` to give them a name and address. Still, it is up to you to handle the type of each value (integer, float...). And also remember that, while a variable in a register can always be used directly, a variable in memory usually needs to go through registers to be manipulated. Like in this example:

```
%define Score  [0]             ; integer
%define BonusMultiplier  [1]   ; float

; this subroutine does: Score *= BonusMultiplier
_increment_score:
  MOV R0, Score            ; read Score to register
  CIF R0                   ; convert to float to be able to multiply both
  MOV R1, BonusMultiplier  ; read BonusMultiplier to register
  IMUL R0, R1              ; now R0 = Score * BonusMultiplier
  CFI R0                   ; to store back in score we need it to be integer again
  MOV Score, R0            ; store back new Score from register
  RET                      ; finished, go back
```

Preserving registers

You may have noticed something wrong in the previous subroutine: registers R0 and R1 are used as temporary variables, but the code calling that subroutine may not be aware of that. It might believe the subroutine is only handling the variables. If this is the case, you will probably experience errors: the calling code may be storing values in, say, R0.

To be safe from this, it is a good practice to preserve the registers you use in subroutines. Or at least, the ones you consider likely to be problematic. You can do this by storing them in the stack before changing their value, and retrieving them back when you are finished. Then the caller code will not have its working values suddenly changed.

```
; this subroutine calculates the square root of R0; it provides
; the result back in R0, so no need to preserve it
_square_root:
  PUSH R1           ; save R1 in the stack
  MOV R1, 0.5       ; POW can only operate with 2 registers! We need R1
  POW R0, R1        ; R0 = sqrt( R0 ) -> we have the result
  POP R1            ; restore R1 from the stack
  RET               ; finished, go back (result is given in R0)
```

The C standard library

If you are not sure of how to interact with the console with assembly code, a good place to look is standard library of the Vircon32 compiler. In the header files you will find many of the functions are either implemented in assembly, or have some core part of them done with an assembly block.

The example below is from header file video.h. The curly brace syntax you see here is a mechanism that allows C variables to be captured in assembly. When compiling, the named variable is replaced with its address.

```
// sets the hotspot of the currently selected region
void set_region_hotspot( int hotspot_x, int hotspot_y )
{
    asm
    {
        "mov R0, {hotspot_x}"
        "out GPU_RegionHotSpotX, R0"
        "mov R0, {hotspot_y}"
        "out GPU_RegionHotSpotY, R0"
    }
}
```

Hardware errors

Like any CPU, the Vircon32 processor may encounter errors in some situations, like dividing by zero. When this happens your program will be aborted and the CPU will raise a hardware error. The BIOS will then show you a message like this, where you can get some information on the error. But there is no way for your program to recover after this.