Apollo-IL specification

The cross-platform and opensource design specification of the Apollo Intermediate Language, targeting the Apollo operating system.

This is just the design specification and standard of the Apollo-IL Language, but it is completely up to the user as to how they wish to implement it, achieving full or partial AIL Compliance.

Version 0.0.1

Apollo Intermediate Language Specification

* [Memory](#_Memory)
* [Instruction Encoding](#_Instruction_Encoding)
* [Pointers and Registers](#_Pointers_and_registers)
* [The stack](#_The_Stack)
* [Program Flow](#_Program_Flow)
* [Command reference and expected outputs](#_Command_reference_and)
  + [Register + Memory Operations](#_Register_+_Memory)
    - [Move](#_Move)
    - [Move (memory)](#_Move_(memory))
    - [Swap](#_Swap)
    - [Test if equal](#_Test_if_equal)
    - [Test if not equal](#_Test_if_not)
    - [Test if less than](#_Test_if_less)
    - [Test if more than](#_Test_if_more)
  + [Arithmetic](#_Arithmetic)
    - [Add](#_Add)
    - [Subtract/Minus](#_Subtract/Minus)
    - [Increment](#_Increment)
    - [Decrement](#_Decrement)
    - [Multiply](#_Multiply)
    - [Divide](#_Divide)
  + [Bitwise operations](#_Bitwise_operations)
    - [Shift Left](#_Shift_Left)
    - [Shift Right](#_Shift_Right)
    - [Rotate Left](#_Rotate_Left)
    - [Rotate Right](#_Rotate_Right)
    - [Bitwise AND](#_Bitwise_AND)
    - [Bitwise OR](#_Bitwise_OR)
    - [Bitwise XOR (Exclusive OR)](#_Bitwise_XOR_(Exclusive)
    - [Bitwise NOT](#_Bitwise_NOT)
  + [Flow control](#_Flow_Control)
    - [Jump](#_Jump)
    - [Call](#_Call)
    - [Return](#_Return)
    - [Jump if true](#_Jump_if_true)
    - [Jump if false](#_Jump_if_false)
    - [Call if true](#_Call_if_true)
    - [Call if false](#_Call_if_false)
  + [Stack Manipulation](#_Stack_Manipulation)
    - [Push](#_Push)
    - [Pop](#_Pop)
  + [I/O](#_I/O)
    - [Receive Byte](#_Receive_Byte)
    - [Receive Word](#_Receive_Word)
    - [Receive Double Word](#_Receive_Double_Word)
    - [Send Byte](#_Send_Byte)
    - [Send Word](#_Send_Word)
    - [Send Double Word](#_Send_Double_Word)
    - [Software Interrupt](#_Software_Interrupt)
    - [Kernel Interrupt](#_Kernel_Interrupt)

Quick Reference

Recommended Pre-processor Instructions

Apollo Intermediate Language (ILA) Executable format

* Structure

ILA Section Entry

* Structure

Appendix A

Appendix B

Licensing

# Memory

There will be 64 kilobytes of addressable memory, which is the maximum amount that can be represented by 16-bit registers.

|  |  |  |
| --- | --- | --- |
| Memory start | End of memory | Purpose |
| Start of memory | Start of memory + 510 bytes | Interrupt Vector Table |
| 0x000001FE | 0x0000FFFF | Main memory |
|  |  |  |
| SP | End of Memory | Stack |

# Instruction Encoding

Each Virtual Machine instruction is to be 48 bits wide. The first 6 of these will contain the instruction, followed by 2 bits which inform the VM how to use the parameters. If the 7th and 8th bits equal 0, the VM reads the parameters as <Register>:<Register>. If they equalled 1, the VM treats the parameters as <Register >:<Value >. If the two bits were equal to 2, the Virtual Machine treats the parameters as <Value >:<Value>. Lastly, if 3, the VM treats them as <Value>:<Register >. The following bits contain the instruction parameters, 1 byte for the first, 4 bytes for the second.

This table displays the possible modes for addressing the parameters, along with the bit to set in the instruction.

|  |  |
| --- | --- |
| Addressing mode bit value  Binary/Hexadecimal | Mode for addressing parameters |
| 0000/0x00 | <Register>:<Register> |
| 0001/0x01 | <Register>:<Value> |
| 0010/0x02 | <Value>:<Value> |
| 0011/0x03 | <Value>:<Register> |

Overall, each instruction is 48 bits/6 bytes wide. This table shows how each instruction should be laid out:

|  |  |  |  |
| --- | --- | --- | --- |
| 6 bits | 2 bits | 1 byte (8 bits) | 4 bytes (32 bits) |
| Instruction | Address Mode | Parameter 1 | Parameter 2 |

# Pointers and registers

The Virtual Machine executing these instructions must have a stack pointer, a program counter along with 16 general purpose registers which are separate from these. These registers are all represented through

* Stack Pointer (referred to as the SP)
  + This read-only register determines the absolute location of the top of the stack. If it holds the value of 0, then the stack is empty.
* Stack Segment (SS)
  + This register determines where the stack should be placed in RAM.
* Program Counter (PC)
  + This contains the address of the next instruction held in the virtual machine’s memory.
* Instruction Pointer (IP)
  + This contains the current point of execution inside the VM’s RAM.
* Eleven general-purpose registers.
  + These have no specific usage to the VM itself, but instead are available for usage by the user and/or compiler.
  + Three 2-byte registers (A, B and C)
  + Six 1-byte registers that make up the three 2-byte registers
    - (AL & AH, BL & BH, CL & CH)
    - (A/B/C) L refers to the lower byte of the registers
    - (A/B/C) H refers to the higher byte of the registers
  + Two 32-bit (4 byte) wide registers X and Y.
* Flag register

Registers are to be represented by a single byte, through from 0xF0 to 0xFE. This table lists all available registers, and their values.

|  |  |
| --- | --- |
| Register | Byte |
| PC | 0xF0 |
| IP | 0xF1 |
| SP | 0xF2 |
| SS | 0xF3 |
| A | 0xF4 |
| AL | 0xF5 |
| AH | 0xF6 |
| B | 0xF7 |
| BL | 0xF8 |
| BH | 0xF9 |
| C | 0xFA |
| CL | 0xFB |
| CH | 0xFC |
| X | 0xFD |
| Y | 0xFE |

# The Stack

The location in memory of the top of the stack is determined by the Stack Segment (SS) register. It can be specified by the program, but it is decided by VM to a default value. The absolute location of the top of the stack is located at the Stack Pointer, which cannot be read by the program but should be set by the VM. If there is nothing on the stack, it’s value is 0. Each time an item is pushed onto the stack, decrement the stack pointer by 1 and then copy the value into the RAM address equal to the Stack Pointer. Popping data from the stack works in a similar fashion, instead incrementing the Stack Pointer by 1 **after** copying the data at the RAM location shown by the Stack Pointer. The SP must be the same word width as the PC (program counter) address size.

# Program Flow

Any AIL executable will run from the first command, iterating through each one until a Flow Control operator is reached (e.g. a jump instruction). Upon encountering a flow instruction, the Virtual Machine’s CPU should immediately jump to the specified location, unless the command specifies otherwise. Labels should be compiled into an address relative to the program base at compile time, and the CPU will execute the program base and the relative address. Consider this example assembler code:

NOP

JMP halt

halt:

CLI

HLT

NOP would be at the relative address 0x00, while the JMP instruction would be address 0x01, the CLI at 0x03 and so on. As for a flow control operator such as the JMP instruction, that would be translated as 0x01 0x03, with the opcode being 0x01 and 0x03 as the label’s relative address.

# Command reference and expected outputs

## Register + Memory operations

### Move

### Move (memory)

### Swap

### Test if equal

### Test if not equal

### Test if less than

### Test if more than

## Arithmetic

### Add

### Subtract/Minus

### Increment

### Decrement

### Multiply

### Divide

## Bitwise operations

### Shift Left

### Shift Right

### Rotate Left

### Rotate Right

### Bitwise AND

### Bitwise OR

### Bitwise XOR (Exclusive OR)

### Bitwise NOT

## Flow Control

### Jump

### Call

### Return

### Jump if true

### Jump if false

### Call if true

### Call if false

## Stack Manipulation

### Push

### Pop

## I/O

### Receive Byte

### Receive Word

### Receive Double Word

### Send Byte

### Send Word

### Send Double Word

### Software Interrupt

### Kernel Interrupt