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](#_gjdgxs)
* [Instruction Encoding](#_30j0zll)
* [Pointers and Registers](#_1fob9te)
* [The stack](#_3znysh7)
* [Program Flow](#_2et92p0)
* [Command reference and expected outputs](#_tyjcwt)
  + [Register + Memory Operations](#_3dy6vkm)
    - [Move](#_1t3h5sf)
    - [Move (memory)](#_4d34og8)
    - [Swap](#_2s8eyo1)
    - [Test if equal](#_17dp8vu)
    - [Test if not equal](#_3rdcrjn)
    - [Test if less than](#_26in1rg)
    - [Test if more than](#_lnxbz9)
  + [Arithmetic](#_35nkun2)
    - [Add](#_1ksv4uv)
    - [Subtract/Minus](#_44sinio)
    - [Increment](#_2jxsxqh)
    - [Decrement](#_z337ya)
    - [Multiply](#_3j2qqm3)
    - [Divide](#_1y810tw)
  + [Bitwise operations](#_4i7ojhp)
    - [Shift Left](#_2xcytpi)
    - [Shift Right](#_1ci93xb)
    - [Rotate Left](#_3whwml4)
    - [Rotate Right](#_2bn6wsx)
    - [Bitwise AND](#_qsh70q)
    - [Bitwise OR](#_3as4poj)
    - [Bitwise XOR (Exclusive OR)](#_1pxezwc)
    - [Bitwise NOT](#_49x2ik5)
  + [Flow control](#_2p2csry)
    - [Jump](#_147n2zr)
    - [Call](#_3o7alnk)
    - [Return](#_23ckvvd)
    - [Jump if true](#_ihv636)
    - [Jump if false](#_32hioqz)
    - [Call if true](#_1hmsyys)
    - [Call if false](#_41mghml)
  + [Stack Manipulation](#_2grqrue)
    - [Push](#_vx1227)
    - [Pop](#_3fwokq0)
  + [I/O](#_1v1yuxt)
    - [Receive Byte](#_4f1mdlm)
    - [Receive Word](#_2u6wntf)
    - [Receive Double Word](#_19c6y18)
    - [Send Byte](#_3tbugp1)
    - [Send Word](#_28h4qwu)
    - [Send Double Word](#_nmf14n)
    - [Software Interrupt](#_37m2jsg)
    - [Kernel Interrupt](#_1mrcu09)

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 | <Value>:<Register> |
| 0010/0x02 | <Register>:<Value> |
| 0011/0x03 | <Value>:<Value> |

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 set by the 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

* + Mnemonic: MOV Opcode: 0x01 Parameters: [required] Destination, [required] Source.
  + Description/Notes: The destination parameter **must** be a register, while the source parameter may be either a register or an absolute value.

### Move (memory)

* + Mnemonic: MOM Opcode: 0x3A Parameters: [required] Source, [required] Destination
  + Description/Notes: The destination parameter **must** be a location in memory, while the source parameter may be either a register or an absolute value
  + Mnemonic: MOE Opcode: 0x3B Parameters: [required] Destination, [required] Source
  + Description/Notes: The destination parameter **must** be a register, and the source parameter **must** be a memory location.

### Swap

* + Mnemonic: SWP Opcode: 0x02 Parameters: [required] Source 1, [required] Source 2
  + Description/Notes: Both sources **must** be registers. Literally swaps the two values stored in the specified registers, so the value in Source 1 is placed into Source 2, and vice versa.

### Test if equal

* + Mnemonic: TEQ Opcode: 0x1A Parameters: [required] Register 1, [required] Register 2
  + Description/Notes: If successful, the next logic-based operation will be executed.

### Test if not equal

* + Mnemonic: TNE Opcode: 0x1B Parameters: [required] Register 1, [required] Register 2
  + Description/Notes: If successful, the next logic-based operation will be executed.

### Test if less than

* + Mnemonic: TLT Opcode: 0x1C Parameters: [required] Register 1, [required] Register 2
  + Description/Notes: If successful, the next logic-based operation will be executed.

### Test if more than

* + Mnemonic: TMT Opcode: 0x1D Parameters: [required] Register 1, [required] Register 2
  + Description/Notes: If successful, the next logic-based operation will be executed.

## Arithmetic

### Add

* + Mnemonic: ADD Opcode: 0x04 Parameters: [required] Destination, [required] Source
  + Description: Adds the source to the destination (e.g. dest = dest + source).
  + Notes: The destination **must** be a register while the source can be either a register or an absolute value.

### Subtract/Minus

* + Mnemonic: ADD Opcode: 0x05 Parameters: [required] Destination, [required] Source
  + Description: Subtracts the source from the destination (e.g. dest = dest - source).
  + Notes: The destination **must** be a register while the source can be either a register or an absolute value.

### Increment

* + Mnemonic: INC Opcode: 0x08 Parameters: [required] Source
  + Description/Notes: Increments the source by 1. (C-Style) pseudocode: source++

### Decrement

* + Mnemonic: DEC Opcode: 0x09 Parameters: [required] Source
  + Description/Notes: Decrements the source by 1. (C-Style) pseudocode: source--

### Multiply

* + Mnemonic: MUL Opcode: 0x30 Parameters: [required] Source, [required] Destination
  + Description: Multiplies the source by the destination and places it into the destination (e.g. dest = source \* dest).

### Divide

* + Mnemonic: DIV Opcode: 0x31 Parameters: [required] Source, [required] Destination
  + Description: Divides the source by the destination and places it into the destination (e.g. dest = source / dest).

## Bitwise operations

### Shift Left

* + Mnemonic: SHL Opcode: 0x06 Parameters: [required] Source, [required] (Positions)
  + Description: Shifts the binary representation (Positions) to the left. (e.g. 001 with the shift of 2 positions is 100) Effectively squares the value in source.

### Shift Right

* + Mnemonic: SHR Opcode: 0x07 Parameters: [required] Source, [required] (Positions)
  + Description: Shifts the binary representation (Positions) to the right. (e.g. 100 with the shift of 2 positions is 001). Opposite of Shift Left (see above).

### Rotate Left

* + Mnemonic: ROL Opcode: 0x0E Parameters: [required] Source, [required] (RotationPosition)
  + Description: Performs a bitwise rotation operation to the left on Source by (RotationPosition). Equivalent of a shift left but any bits which are removed instead get appended on the right. (e.g. 101 rotated left 1 position would be 011)

### Rotate Right

* + Mnemonic: ROR Opcode: 0x0F Parameters: [required] Source, [required] (RotationPosition)
  + Description: Performs a bitwise rotation operation to the right on Source by (RotationPosition). Equivalent of a shift right but any bits which are removed are instead appended on the left. (e.g. 101 rotated right 1 position would be 110)

### Bitwise AND

* + Mnemonic: AND Opcode: 0x0A Parameters: [required] Source A, [required] Source B
  + Description: Performs an AND operation between Source A and Source B. (e.g. if A is 1101001 (binary), and B is 1010110, the result (placed in Source A’s register) would be 1000000)

### Bitwise OR

* + Mnemonic: BOR Opcode: 0x0B Parameters: [required] Source A, [required] Source B
  + Description: Performs an OR operation between Sources A and B. (e.g. if A is 1010110 and B is 1101101, A becomes 1111111)

### Bitwise XOR (Exclusive OR)

* + Mnemonic: XOR Opcode: 0x0C Parameters: [required] Source A, [required] Source B
  + Description: Performs an Exclusive OR operation between sources A and B. (e.g. if A is 1010110 and B is 1101101, A becomes 0111011)

### Bitwise NOT

* + Mnemonic: NOT Opcode: 0x0D Parameters: [required] Source
  + Description: Performs a NOT operation on source A. (e.g. if A is 1010110 then A becomes 0101001)

## Flow Control

### Jump

* + Mnemonic: JMP Opcode: 0x10 Parameters: [required] Destination
  + Notes: Destination may be a register, absolute address or label.
  + Description: The assembler will swap any references to a label to the address of the label

### Call

* + Mnemonic: CLL Opcode: 0x11 Parameters: [required] Call Location.
  + Notes: Call location may be register, absolute address or label.
  + Description: Same operation as Jump, however when a Return instruction is encountered, the execution passes back to the calling instruction. (e.g. if 0xAF is called from 0x1C, when a Return instruction is encountered in the 0xAF area, control will pass to the next instruction after 0x1C.

### Return

* + Mnemonic: RET Opcode: 0x12
  + Description/Example: Returns to the position before when a subroutine was called. Returning from the subroutine that has previously been entered using the Call instruction (See above)

### Jump if true

* + Mnemonic: JMT Opcode: 0x13 Parameters: [required] Destination.
  + Description: If a Boolean operation has occurred previously, and the result was true, this jump will be made, otherwise no jump will be made by this command.

### Jump if false

* + Mnemonic: JMF Opcode: 0x14 Parameters: [required] Destination
  + Description: Functions the same as the Jump If True instruction, except the condition for execution is false.

### Call if true

* + Mnemonic: CLT Opcode: 0x17 Parameters: [required] Call Location.
  + Notes: Call location may be register, absolute address or label.
  + Description: Like CLL, but only if the previous Boolean operation is true.

### Call if false

* + Mnemonic: CLF Opcode: 0x18 Parameters: [required] Call Location.
  + Notes: Call location may be register, absolute address or label.
  + Description: Like CLT but the condition for execution is 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