Fantasy-Console Assembly Syntax and ISA Reference

KamS04

July 20th, 2024

1 TODO Notes:

- This is not implemented yet but the usage of the **CLINE** keyword must be changed. Currently **CLINE** returns the adderss of the current instruction being encoded. This is useful!! but it really isn't. Because whoever is writing FCASM, must manually calucate how many bytes to offset because CLINE returns a byte address. This should change, instead CLINE should return the line index of the line being encoded.
- Line indexes are a concept that must be properly implemented into the assembler (Scaffolding exists already with the nlineidxes) but this must be pushed further, Line indexes should start at 0 and the counter should be incremented at every encoded element, i.e. a constant element should not increment the line index, but a data element should.
- A macro needs to be introduced called something of the sorts of ADDROF(x), where x is a line index, and the result of this macro is the encoded address of that line.
- Fixup how memory operands are written, I think it's weird in some places
- Introduce a system to specify the size of a data block rather than defining the data

Contents

2 Syntax

Each line in the assembly is one of a few different kinds of statements.

2.1 Instruction

An instruction starts with an instruction tag (found in Instructions), and then the right number and type of arguments. The type and number of arguments are determined by the instruction format (found in Instruction Formats).

2.2 Data Element

- data8 **identifier** = { *%comma_separated_hex_literals%* }
- data16 **identifier** = { *%comma_separated_hex_literals%* }

A data element specifies a data block within the final assembly that can also be referenced by the rest of the assembly file. The size modifier after the data keyword specifies the size of each element in the data block. Each literal provided in the body of a data element increases the size of the data block by the given size modifier. The size modifier is either *8 Bits* or *16 Bits* (1 or 2 bytes). Total size of a data block is not specified (probably should be) but rather is calculated by multiplying the size modifier by the number of hex literals. Note the data block MUST be initialized to the values in the data declaration.

2.3 Constant Element

- constant **identifier** = *$hex_literal*

This defines a compile time constant. The constant can be referenced by the assembly file (and exported to other linked assembly files). Note that these constants cannot be edited after declaration AND these constants do not exist at runtime.

2.4 Structure Element

- structure **identifier** { *%comma_separated_member_declarations%* }
- member_declaration of form: "**identifier**: *$size_in_bytes_as_hex_literal*"

The structure element creates a compiletime entity that contains offsets, that can be used to group a memory block into an object and then access the individual members of said object, using the .operator.

2.5 Argument Struture Element

- structure **identifier** { *%comma_separated_member_declarations%* }

The Arguments structure is equivalent to a regular structure element, BUT the offsets are offsetted by the stack frame size. For subroutines using the hardware calling mechanism, and the stack arguments system can use this construct to skip over the stack frame and access the arguments for this subroutine directly.

2.6 Label

- **identifier**:

This is equivalent to a compile time constant except this value is set to the current memory address being encoded.

3 More Syntax

Registers are indexed by their names.

- ip
- acu
- r1
- r2
- r3
- r4
- r5
- r6
- r7
- r8
- fp
- sp

3.1 Literal

- Hex literal is represented with a **'$'** followed by 1-4 hex digits
- Literal form variable can also be acquired using a SQUARE_BRACKET_EXPRESSION
  - For a literal no prefix is required for the expression

3.2 Memory / Address

- Memory/Address is represented with a **'&'** followed by a 1-4 hex digit
- Memory/Address can also be acquired using SQUARE_BRACKET_EXPRESSION
- prefix **'&'** required for memory/address

3.3 Register Pointer

- Register pointer is equivalent to accessing a memory/address from register
- Hence, it is a **'&'** followed by a register name

3.4 SQUARE_BRACKET_EXPRESSION

A square bracket starts and terminates with a square bracket. A square bracket supports the following operators

- +
- -
- *
- /

Note: division implementation is not the most robust, use sparingly with lots of tests.

A square bracket **MUST** have at least one argument/value. The use of operators is optional. **ANY** operator used **MUST** have an argument on either side.

3.5 SQUARE_BRACKET_EXPRESSION Argument/Value Syntax

3.5.1 Hex Literal

This is the exact same as a regular hex literal

3.5.2 Variable

- **"!identifier"**

A variable is written as an exclamation mark followed by a variable name.

3.5.3 Interpret As Single

- *<**%Structure%**>%symbol%**.**%property%*

The interpret as single command tells the assembler expression evaluator to assume that symbol is a pointer/memory address to a memory block of type **Structure**. The evaluator then calculates and returns the corresponding address of the **property** within the structure. The **symbol** can be a variable or a hex literal using their corresponding syntaxes.

3.5.4 Interpret As Array

- *<**%Structure%**>%symbol%**[%index%]**.%property%*

This is similar to interpret as single. This assumes that **symbol** is a pointer to an array of **Structure** elements. The evaluator calculates the address of the **index**-th item of this array (found by multiplying the total size of the **Structure** by **index** and adding to the value of **symbol**). The evaluator then finds the offset of the **property** within this data block and this final address is returned.

4 Instruction Formats

A collection of instruction types are available to the assembler. These types determine the number and types (sizes) of arguments for each instruction.

- **noArgs**
  - No arguments passed to this format
- **singleReg**
  - 1 Register index passed to this format
- **singleMem**
  - 1 memory address passed to this format
- **singleLit**
  - 1 literal value passed to this format
- **singleLit8**
  - 8-bit literal value passed to this format
- **lit8Mem**
  - 8-bit literal and memory address
- **litMem**
  - 16-bit literal and memory address
- **litOffReg**
  - 16-bit literal, register pointer, and register
- **memReg**
  - Memory address followed by register
- **regMem**
  - Register followed by Memory address
- **litReg**
  - Literal followed by Register
- **regPtr**
  - 1 Register index passed, Value in register is read as Memory address
- **regReg**
  - Register followed by another register
- **regLit**
  - Register followed by literal
- **regLit8**
  - Register followed by 8-bit literal
- **regPtrReg**
  - Register pointer followed by register
- **regRegPtr**
  - Register followed by a register pointer
- **regRegReg**
  - Register, register, and then a register

5 Instructions

5.1 log

5.1.1 LOG REG - *singleReg*

- 0x20
- prints value of **register** to debugging output (STDOUT on most VMs)

5.1.2 LOG REG PTR - *regPtr*

- 0x21
- prints value of memory at address in **register** to debugging output

5.1.3 LOG MEM - *singleMem*

- 0x41
- prints value of memory at **address** to debugging output

5.2 mov8

5.2.1 MOV8 LIT MEM - *lit8Mem*

- 0xD9
- move 1 byte **literal** to **address**

5.2.2 MOV8 MEM REG - *memReg*

- 0x9b
- move 1 byte from **address** to **register**

5.2.3 MOV8 REG PTR REG - *regPtrReg*

- 0xBF
- copy byte at address in **register 1** to **register 2**

5.2.4 MOV8 REG REG PTR - *regRegPtr*

- 0xBB
- copy low byte in **register 1** to address in **register 2**

5.3 movl

5.3.1 MOVL REG MEM - *regMem*

- 0x79
- copy low byte in **register** to **memory**

5.4 movh

5.4.1 MOVH REG MEM - *regMem*

- 0x7D
- copy high byte in **register** to **memory**

5.5 mov

5.5.1 MOV LIT REG - *litReg*

- 0x98
- move **literal** to **register**

5.5.2 MOV REG REG - *regReg*

- 0xB8
- copy value in **register 1** to **register 2**

5.5.3 MOV REG REG PTR - *regRegPtr*

- 0xBA
- copy literal in **register 1** to address in **register 2**

5.5.4 MOV REG MEM - *regMem*

- 0x78
- copy literal in **register** to **address**

5.5.5 MOV MEM REG - *memReg*

- 0x9A
- copy memory at **address** to **register**

5.5.6 MOV LIT MEM - *litMem*

- 0xD8
- move **literal** to **address**

5.5.7 MOV REG PTR REG - *regPtrReg*

- 0xBE
- copy value at address in **register 1** to **register 2**

5.5.8 MOV LIT OFF REG - *litOffReg*

- 0xF8
- copy value at address, calculated by adding **literal** offset to address in **register 1**, to **register 2**

5.6 movb

5.6.1 MOV BLOCK - *regRegReg*

- 0x7E
- copy memory blck at address in **register 1**, to memory block at address in **register 2**, of length in **register 3**
- all of these are RAV addresses

5.7 int

5.7.1 INT LIT - *singleLit*

- 0x45
- interrupt with literal
  - stack frame is not saved to stack
  - only thing saved to stack is the next_instruction_address
  - similar to ther than cal

5.7.2 INT REG - *singleReg*

- 0x25
- interrupt with value in **register**
  - stack frame is not saved to stack
  - only thing saved to stack is the next_instruction_address
  - similar to jmp rather than cal

5.8 rti

5.8.1 RET INT - *noArgs*

- 0x05
- return from interrupt
  - pops off next_instruction_address and jump to it
  - no stack frame to remove

5.9 add

5.9.1 ADD REG REG - *regReg*

- 0xB0
- add value in **register 1** with value in **register 2**, save in acu

5.9.2 ADD LIT REG - *litReg*

- 0x90
- add **literal** with value in **register**, save in acu

5.10 sub

5.10.1 SUB LIT REG - *litReg*

- 0x91
- subtracts value in **register** from **literal**, save in acu

5.10.2 SUB REG LIT - *regLit*

- 0x71
- subtract **literal** from value in **register**, save in acu

5.10.3 SUB REG REG - *regReg*

- 0xB1
- subtract value in **register 2** from value in **register 1**, save in acu

5.11 inc

5.11.1 INC REG - *singleReg*

- 0x30
- increment the value in the **register** by 1 (in place)

5.12 dec

5.12.1 DEC REG - *singleReg*

- 0x31
- decrement the value in the **register** by 1 (in place)

5.13 mul

5.13.1 MUL LIT REG - *litReg*

- 0x96
- multiply a **literal** by a value in **register**, save in acu

5.13.2 MUL REG REG - *regReg*

- 0xB6
- multiply value in **register 1** by value in **register 2**, save in acu

5.14 lsf

5.14.1 LSF REG LIT - *regLit8*

- 0x74
- shift value in **register** by **literal** bits to left, save in acu

5.14.2 LSF REG REG - *regReg*

- 0xB4
- shift value in **register 1** by value in **register 2** bits to left, save in acu

5.15 rsf

5.15.1 RSF REG LIT - *regLit8*

- 0x75
- shift value in **register** by **literal** bits to right, save in acu

5.15.2 RSF REG REG - *regReg*

- 0xB5
- shift value in **register 1** by value in **register 2**, save in acu

5.16 and

5.16.1 AND REG LIT - *regLit*

- 0x77
- perform bitwise *and* between value in **register** and **literal**, save in acu

5.16.2 AND REG REG - *regReg*

- 0xB7
- perform bitwise *and* between value in **register 1** and value in **register 2**, save in acu

5.17 or

5.17.1 OR REG LIT - *regLit*

- 0x72
- perform bitwise *or* between value in **register** and **literal**, save in acu

5.17.2 OR REG REG - *regReg*

- 0xB2
- perform bitwise *or* between value in **register 1** and value in **register 2**, save in acu

5.18 xor

5.18.1 XOR REG LIT - *regLit*

- 0x73
- perform bitwise *xor* between value in **register** and **literal**, save in acu

5.18.2 XOR REG REG - *regReg*

- 0xB3
- perform bitwise *xor* between value in **register 1** and value in **register 2**, save in acu

5.19 not

5.19.1 NOT - *singleReg*

- 0x37
- perform bitwise *not* on value in **register**, save in acu

5.20 jmp

5.20.1 JMP REG - *singleReg*

- 0x28
- Jump to address inside register

5.20.2 JMP LIT - *singleMem*

- 0x48
- Jump to address

5.21 jne

5.21.1 JNE LIT - *litMem*

- 0xCD
- if **literal** is not equal to value in acu, jump to **address**

5.21.2 JNE REG - *regMem*

- 0x6D
- if value in **register** is not equal to value in acu, jump to **address**

5.22 jeq

5.22.1 JEQ REG - *regMem*

- 0x6A
- if value in **register** is equal to value in acu, jump to **address**

5.22.2 JEQ LIT - *litMem*

- 0xCA
- if **literal** is equal to value in acu, jump to **address**

5.23 jlt

5.23.1 JLT REG - *regMem*

- 0x6C
- if value in **register** is less than value in acu, jump to **address**

5.23.2 JLT LIT - *litMem*

- 0xCC
- if **literal** is less than value in acu, jump to **address**

5.24 jgt

5.24.1 JGT REG - *regMem*

- 0x69
- if value in **register** is greater than value in acu, jump to **address**

5.24.2 JGT LIT - *litMem*

- 0xC9
- if value in **register** is greater than value in acu, jump to **address**

5.25 jle

5.25.1 JLE REG - *regMem*

- 0x6E
- if value in **register** is less than equal to value in acu, jump to **address**

5.25.2 JLE LIT - *litMem*

- 0xCE
- if **literal** is less than equal to value in acu, jump to **address**

5.26 jge

5.26.1 JGE REG - *regMem*

- 0x6B
- if value in **register** is greater than equal to value in acu, jump to **address**

5.26.2 JGE LIT - *litMem*

- 0xCB
- if **literal** is greater than equal to value in acu, jump to **address**

5.27 psh

5.27.1 PSH LIT - *singleLit*

- 0x42
- push **literal** to stack (will add 2 to the *sp* register)

5.27.2 PSH REG - *singleReg*

- 0x22
- push value in **register** to stack (will add 2 to the *sp* register)

5.28 pop

5.28.1 POP - *singleReg*

- 0x23
- pop value off stack and save in **register** (will subtract 2 from the *sp* register)

5.29 cal

5.29.1 CAL LIT - *singleLit*

- 0x44
- call subroutine at **address**
  - saves stack frame to stack
  - assume that registers will stay intact on return of subroutine
    - except for acu, which will contain the result of the subroutine

5.29.2 CAL REG - *singleReg*

- 0x24
- call subroutine at address in **register**
  - saves stack frame to stack
  - assume that registers will stay intact on return of subroutine
    - except for acu, which will contain the result of the subroutine

5.30 ret

5.30.1 RET - *noArgs*

- 0x04
- return from subroutine
  - all pushes to stack from within subroutine will be lost
  - stack frame will be popped from stack and registers will be restored for the calling routine
  - acu register will stay intact

5.31 hlt

5.31.1 HLT - *noArgs*

- 0x06
- tells the cpu to stop cycling, effectively turns off the VM

5.32 brk

5.32.1 BRK - *noArgs*

- 0x03
- used for debugging
  - when the cpu is in debug mode, the CPU will stop at the **brk** instruction and wait for user input before continuing

5.33 mms

5.33.1 MEM MODE SET - *singleLit8*

- 0x47
- sets memory mode to **literal**
  - for now memory modes are
  - 0: **RELATIVE**
    - memory access are done by taking address and adding the start address of the memory device to which the last interrupt either pointed to or returned to
  - 1: **ABSOLUTE**
    - memory accesses are done with the addresses directly

5.34 rav

5.34.1 REAL ADDRESS VALUE REG PTR - *regReg*

- 0xA7
- calculates the real memory address of the address in **register 1** and saves it to **register 2**
- depends on Memory Mode
  - will add the start address of the memory device to which the last interrupt either pointer to or returned to, with the address in **register 1** in *RELATIVE* Memory device
  - will directly return the address in **register 1** given in *ABSOLUTE* Memory Device

5.34.2 REAL ADDRESS VALUE MEM - *litReg*

- 0x87
- calculates the real memory address of the **address** and saves it to **register**
- depends on Memory Mode
  - will add the start address of the memory device to which the last interrupt either pointer to or returned to, with the **address** in *RELATIVE* Memory device
  - will directly return the address given in *ABSOLUTE* Memory Device

5.35 sig

5.35.1 SEND SIGNAL - *singleReg*

- 0x27

- send a signal to device with DeviceID of value in **register**
  - no equivalent literal instruction since there is not specification that asserts that DeviceIDs are predicatble and hence compile time constants
  - Any use of a DeviceID should be from a register, hopefully calculated using either the DeviceID arrays, or from the data within the device codespaces
    - *Since the first byte of any device's codespace will be set to the DeviceID associated with it*
  - DO NOT attempt to use a *literal* to signal a device, it is an unspecified action and depending on implementation may cause issues

6 Oh No Instructions

There is a reference to the following instructions in the most recent and thereby most up-to-date assembler. Unfortunately, these instructions have no explanation or implementations, and therefore I have no clue what they are for or what they do. That being said it seems like they are related to the LogicSim implementation of the FanCon system.

6.1 sjge

6.1.1 JGE REGI - *regMem*

- 0x6f

- ??

6.2 slog

6.2.1 LOG REGI - *singleReg*

- 0x26

- ??

7 Instructions Table

	L	010	
log	LOG MEM	00_001	41
int	INT LIT	00_101	45
psh	PSH LIT	00_010	42
cal	CAL LIT	00_100	44
mms	MEM MODE SET	00_111	47
jmp	JMP LIT	01_000	48

	LL	110	
mov8	MOV8 LIT MEM	11_001	D9
mov	MOV LIT MEM	11_000	D8
jne	JNE LIT	01_101	CD
jeq	JEQ LIT	01_010	CA
jlt	JLT LIT	01_100	CC
jgt	JGT LIT	01_001	C9
jle	JLE LIT	01_110	CE
jge	JGE LIT	01_011	CB

	LRR	111	
mov	MOV LIT OFF REG	11_000	F8

	R	001	
log	LOG REG	00_000	20
log	LOG REG PTR	00_001	21
inc	INC REG	10_000	30
dec	DEC REG	10_001	31
not	NOT	10_111	37
jmp	JMP REG	01_000	28
psh	PSH REG	00_010	22
cal	CAL REG	00_100	24
sig	SEND SIGNAL	00_111	27
pop	POP	00_011	23
int	INT REG	00_101	25
	LOG REGI	00_110	26

	NIL	000	
rti	RET INT	00_101	05
ret	RET	00_100	04
hlt	HLT	00_110	06
brk	BRK	00_011	03
nop	NOP	00_000	00

	RR	101	
mov8	MOV8 REG PTR REG	11_111	BF
mov8	MOV8 REG REG PTR	11_011	BB
mov	MOV REG REG	11_000	B8
mov	MOV REG REG PTR	11_019	BA
mov	MOV REG PTR REG	11_110	BE
add	ADD REG REG	10_000	B0
sub	SUB REG REG	10_001	B1
mul	MUL REG REG	10_110	B6
lsf	LSF REG REG	10_100	B4
rsf	RSF REG REG	10_101	B5
and	AND REG REG	10_111	B7
or	OR REG REG	10_010	B2
xor	XOR REG REG	10_011	B3
rav	REAL ADDRESS VALUE REG PTR	00_111	A7

	RL	011	
movl	MOVL REG MEM	11_001	79
movh	MOVH REG MEM	11_101	7D
mov	MOV REG MEM	11_000	78
movb	MOV BLOCK	11_110	7E
sub	SUB REG LIT	10_001	71
lsf	LSF REG LIT	10_100	74
rsf	RSF REG LIT	10_101	75
and	AND REG LIT	10_111	77
or	OR REG LIT	10_010	72
xor	XOR REG LIT	10_011	73
jne	JNE REG	01_101	6D
jeq	JEQ REG	01_010	6A
jlt	JLT REG	01_100	6C
jgt	JGT REG	01_001	69
jle	JLE REG	01_110	6E
jge	JGE REG	01_011	6B
	JGE REGI	01_111	6f

	LR	100	
mov8	MOV8 MEM REG	11_011	9B
mov	MOV LIT REG	11_000	98
mov	MOV MEM REG	11_010	9A
add	ADD LIT REG	10_000	90
sub	SUB LIT REG	10_001	91
mul	MUL LIT REG	10_110	96
rav	REAL ADDRESS VALUE MEM	00_111	87

8 I/O reference

Unfortunately, the FantasyConsole system allows for pseudo I/O devices to be connected to the VM runtime. The system sets up three subtopics to create.

- Device
- Device Driver Interaction
- User-code Interaction

These devices should be implemented mostly through kotlin, this kotlin setup is largely ignored in within this reference because it can be rather expansive and a lot of anything is possible. To turn a .jar into an acceptable device it needs to have a class that implements the **ca.kam.vmhardwarelibraries.TechDevice** interface. This interface requires defining the following properties

- isForked
  - This is a boolean value that should signify if the device requires its code to run parallel or just runs on the **signal** function. This is largely implementation dependent, the current Kotlin-FC implementation does not care about this flag, but other implementations may.

- deviceInfo
  - This is a field of type **ca.kam.vmhardwarelibraries.DeviceAsks** which is outlined later. This are the resources that the device requests from the device.
- lockBuffer
  - This is a function hook, that gives the device a reference to the memory device that represents the memory buffer that the device has requested. This function is only ever run once and then never again. The device should save this reference internally.
- getCode
  - A function that should return a **UByteArray** that contains the driver code that the device provides.
- signal
  - A function hook, this is called when the **sig** instruction is executed with this device as its argument.

The class **ca.kam.vmhardwarelibraries.DeviceAsks** has the following properties

- name
  - A single UByte that defines the name of the device. Ideally would want this to be unique. In the future we would need some better way to identify devices.
- codeSpaceRequest
  - UShort that defines the size of the code space that the device driver requires. The Kotlin-FC implementation has a hard cap of 600 bytes of maximum code space.
- commBufferRequest
  - UShort that defines the requested buffer size for the device. The Kotlin-FC implementation has a hard cap of 50 bytes as the maximum buffer size.
- interrupts
  - Array of UShorts that contains the local space addresses of the interrupt handlers that the device defines. The Kotlin-FC implementation currently supports a maximum of 2 device defined interrupts. (Technically)

The code space that is provided to the device is populated as follows

- First byte is set to the device ID
- The next 2 bytes are set to the address of the communication buffer requested by this device
- the rest of the code space is copied directly from the **getCode** function of the device

Note, that this means that the device driver code should start with a *data8* for the device ID, and follow up with a *data16* for the communication buffer address.
This brings us to the user side and the API provided by the FC for accessing, identifying, and finding devices. The FC system setup creates 3 arrays to help with this.

- Communication Buffer address list
- Device name list
- Device first interrupt list

Interrupts 3 and 4 return the device name and the device first interrupt lists into the **acu**. Interrupt 2 requires the device id to be passed as an argument in **r1** and returns the communication buffer address of that device into **acu**.
This brings us to the user-api for accessing an I/O device. The user code should first grab the device name list, and the device first interrupt list. The device ID is effectively the index of the device name within the device name list. So the user-api should walk through the device name list until it finds the device it's looking for. The user code now knows the device ID and also knows the first interrupt corresponding to the device it wants. The user code can now use the interrupts to use the device. The user code can also use interrupt 2 and the device ID to find the communication buffer and put data their directly. Not recommended and will only work if the device and device driver support/expect this behaviour. *Note: the user code should probably have a pseudo "device driver" that effectively caches the previously mentioned information so that cpu cycles are not wasted finding and refinding device drivers.*

8.1 TODO

Actually fix the device naming issue.