evm8: an embedded 8/16 bits virtual machine.

Sebastien Lorquet <sebastien@lorquet.fr>

2010 – 2012

version B4

Abstract : This document presents a virtual machine for use in embedded computing systems. It describes a virtual processor architecture and instruction set, as well as executable format encoding.

Contents

1 Revisions

Révision	Date	Description
B4	2012	Reorganized instructions description, added table and front page corrected code size in execution format More stack pointer clarifications. Interrupts simplification Add default processor status reg values
B3	2012-06-05	When no kernel module is loaded, the default app starts in user mode. Stack pointer clarifications. Remove SP, keep USP and SSP.
B2	2012-06-04	Clarifications in memory management Instruction encoding table completion Started description of each instruction Submitted for review.
B1	Somewhere in 2010	Initial developments and concepts

2 Introduction

The goal is to define an embedded machine that can execute code on different 8-bits architectures, as well as on custom hardware CPUs, such as an FPGA implementations.

Requirements are:

-Simplicity

-Speed

-Low VM footprint

-Dynamic library/module management

-Optimal execution on 8 bits micro architectures such as PIC, AVR, HC11 and MCS51.

3 System architecture

1 Overview

The processor has an Harvard register based architecture, with 3 separate data spaces:

- The code space holds executable instructions and constant data (TXT et RODATA);
- The data space holds the stack and module data (BSS and DATA);
- The IO space holds peripheral control/status registers.

The code is organized in executable modules, loaded by the VM loader, which is system dependent.

The processor has 2 states, allowing different kinds of privileges: a supervisor mode, and an user mode. The supervisor mode can do I/O operations, while the user mode cannot.

Of course these execution modes are distinct from the main execution mode of the host system, which is not discussed in this document.

2 Modules

Code is organized into modules. Modules can be programs and libraries.

A module contains metadata for management, and opcodes.

Each module can only access the opcodes that are stored in the same module. The PC register is 16-bits wide thus allowing any module to extend to 65536 code bytes. The module load address, and the fact that this physical address is possibly wider than the user available PC and not pointing into main CPU memory, is unknown to the code. This ensures that no code can be fetched from outside the module and gives a large amount of flexibility for the underlying implementation. Code from other modules may be called via import and export tables.

Executable modules are designed to be position independent, so that they can be loaded and unloaded at any time in the system lifetime, without relying on the load address.

All modules have a 8 bytes long name. These bytes are not required to be ascii characters. Trailing zero bytes or space chars can used for padding.

Any module can export functions. They are declared in an export table, which is a list of program counter offset values that mark exported functions, indexed by a 16-bit number.

Any module can import functions. They are declared in an import table, which allows for dependency checking at module load time.

Non volatile (NV) module registry

The NV Module registry is a table, holding permanent (e.g. information that do not change during module lifetime) management information about modules. This table is system dependent, and should contain a minimal set of fields to allow localization of the modules inside the system NV memory and identification of modules attributes.

The executable format described later in this document has header information that can be used as a module registry info. The minimal required information is:

- module name (8 bytes)
- module position in memory (system dependent, typically 2-3 bytes)
- flags (at least 8 bits or one byte)

The flags are defined as follows:

B7	B6	B5	B4	B3	B2	B1	B0	Description
-	-	-	-	-	-	-	1	Default application. This module will be run on machine boot.
-	-	-	-	-	-	1	-	Kernel module. This module has interrupt handlers.
0	0	0	0	0	0	-	-	RFU, must be set to zero.

There can only be one kernel module and one default application.

Volatile module registry

A memory zone is dedicated to store volatile information about modules. This includes the BSS start address, which is allocated at boot time.

3 Execution context

One or more execution context (e.g. tasks) can be active at once to support single or multithreading. Scheduling and multiple stack allocation are not defined yet.

An execution context stores the volatile information for the currently running thread, including all the general purpose and special registers, along with the current module being executed. (this information is system dependent and can be a single byte if no more than 256 modules are supported).

The maximum number of contexts (threads or tasks) that can exist in the machine can be fixed or dynamic.

4 Memory

Code and Data addresses used at runtime are virtual and valid in the current module only, each module lives in a separate address space.

Code memory

The physical structure for the global code space is not defined in this specification. It just have to be a set of non volatile memory zones, one for each module. There no requirements for this memory to be global, contiguous, or directly addressable by the CPU.

Some memory allocation strategy can be used to allocate memory to modules in a single pool, or modules can be separate non volatile objects having nothing in common. No specific alignment is required, since this memory may not be directly addressable central memory, but rather an external memory storage device. This device shall provide random byte read operations. The code space also stores constant data information.

Data memory

The data space is volatile memory allocated to an executable module when the machine boots or a new module is loaded at runtime. Again, there are no requirements on the structure of this memory, it does not have to be a single contiguous memory pool nor CPU addressable memory.

At load time, each module is allocated a volatile memory segment for its BSS and initialized DATA variables. This memory information is retained in the “volatile module registry”. At run time, data elements are addressed using offsets, e.g. the first BSS/DATA byte has zero address. The real memory has to be managed by the virtual machine by adding the real data address to the offset provided in the runtime instructions. BSS/DATA accesses are checked so that any module cannot use memory that does not pertain to the same module.

5 Boot process

At startup, the VM does the following:

- A 128 byte supervisor stack is defined. This will probably be allocated in the first or last available RAM addresses.
- Affect a BSS RAM block for each loaded module (the required size is indicated in the module metadata) , initialize it to zero and save their address in the volatile module registry. For systems where the RAM memory is shared between the BSS blocks and the Stack blocks, it is recommended that all BSS blocks are allocated to the lowest possible addresses.
- The user stack is defined. For systems where the RAM memory is shared between the BSS and stack data, it is recommended to define the stack at the end of the memory and let it grow to lower addresses. This allows runtime loading and activation of more code modules, provided that the necessary BSS memory is still available. The SSP and USP registers are cleared.

  Another strategy can be to allocate the supervisor stack and bss blocks at high adresses and let the user stack grow to high addresses from the low ram addresses. The important part of this recommendation is to let a central memory zone that can be eaten from both ends, which is necessary to load and activate a new module at runtime without a reboot or RAM defragmentation in systems where the RAM is shared between BSS and stacks.
- if a kernel module is installed, it is executed in supervisor mode. It MUST returns or nothing more will happen. This feature is enabled to setup the system before any application is run. After that, if a default module is installed, it is executed in user mode.

  If a kernel module was executed, but there is no default application, the system is put into low power SLEEP mode, waiting for an interrupt.
- If there is no kernel module,
  - if a default module is installed, the default module is executed in user mode. This mode is easy to use for tests, but this means that the code will only be able to read and write a serial port in polling mode, and that no interruptions will be installed nor installable.

    If there is no kernel module and no default application, the system stays in the bootloader waiting for module load commands.

6 Exceptions

When special conditions are met, such as cpu /stack errors, an exception is generated.

If no kernel module is defined, the system reboots.

Else, the exported function for the exception is searched. If no exported function exists, the system reboots.

If an exception function is found, supervisor mode is entered, then the function is executed.

TODO: stack frame?

Trap vectors are user-triggered exceptions, that can be used to enter supervisor mode from user mode under software control.

Entry point	Kernel exported function number
Division by zero	0x40
Invalid opcode	0x41
Address error (tried to jump in the wild or read non existent data)	0x42
System abort	0x43
Stack fault (over, under, access)	0x44
Module not found	0x45
RFU	0x46 – 0x4F
Trap #0 – #16	0x50 – 0x5F
High priority Interrupt	0x60
Low priority interrupt	0x61

7 General purpose registers

The machine has 8 registers named R0-R7, each one is 8 bits wide.

When the D bit of an instruction is set, the operation operates on register pairs. In this case the least significant bit of the register number is set to zero and the operation uses registers N and N+1 (modulo 8) to perform the operation. N has to be even.

R0	R1	R2	R3	R4	R5	R6	R7
W0		W2		W4		W6	

8 Special registers

The machine has special registers, usable only in bitwise , MOVE and LOAD/STORE instructions.

Register	Numeric encoding	Size in bits	Description
PC	0 0 0	16	Instruction pointer
SP	0 0 1	16	Current stack pointer. In supervisor mode, this is the Supervisor stack pointer. In user mode it is an alias for USP.
USP	0 1 0	16	User mode stack pointer, only available from the supervisor mode. Any attempt to use this register in user mode will trigger a 'System abort' exception.
FP	0 1 1	16	Frame pointer or generic 16 bits pointer if not used
AS	1 0 0	8	ALU Status register, 8 bits, holds arithmetic and logic CPU state bits. Available in all modes.
PS	1 0 1	8	Processor Status register, 8 bits, holds system CPU state bits. Only available in supervisor mode.
CM	1 1 0	8	Current module (read only). Used to compute the real instruction address in conjuction with module table and PC.
IC	1 1 1	16	Interrupt code. Number of the currently triggered interrupt. Values are declared in IO registers.

ALU Status register bits:

B0	Z	Last result was zero
B1	C	Last result produced a carry
B2	N	Last result was negative
B3	V	Last result overflowed
B4	0	Always read as zero, writes discarded
B5	0	
B6	0	
B7	0	

Processor Status bits:

B0	M	Processor mode (0=user, 1=supervisor)
B1	T	Trace/ Single step enable
B2	I	Global Interrupt enable
B3	B	Endianness control (0=BE, 1=LE)
B4	SM	Autostack mode enable (0=disabled, 1=enabled)
B5	0	Always read as zero, writes discarded
B6	0	
B7	0	

The processor status bits cannot be read nor written in user mode. They have to be changed by the kernel module. When no kernel module is loaded, the default values for these parameters are 0x00:

- user mode
- no trace
- no interrupts
- big endian (to be discussed)
- auto stack disabled

TODO: discuss merging AS and PS in a single 16-bit register to save a special register address. In that case the PS part will always read all zeros in user mode.

9 Stack

The stack registers have 16 bits, but the available memory can be bigger than that. The real stack address is computed using an internal “top of stack” register that is big enough to target the full address space, to which the user available stack pointer is added. This allows a full 16-bit stack to be used.

The top of stack pointers are saved by the VM but are not available to the user. Instead, the SP and USP registers are zero based, and the real memory is accessed by adding/substracting the contents of the current SP register to/from the real stack base address. In the same time, stack over/underflows are checked and reported.

When the SM bit in the Processor Status Register is set, and not using an index, any STORE instruction requesting write access from the SP address will postincrement the SP register, effectively executing a “PUSH”. When another register is used, or when SP is used with an index, or when the SM bit is not set, the register used to read memory will not be altered.

In a symmetric way, any non-indexed LOAD access using the SP register will predecrement the SP register, effectively executing a “POP”.

In this mode, loading or storing a byte register will change the SP value by one, but if the W bit of the load/store instruction is set, then SP will be changed by two.

TODO define what to do in kernel mode and accessing memory at USP.

10 IO space

The IO memory is a virtual memory zone (not backed by any actual memory except for the descriptors) used to abstract the I/O peripherals. This space starts with IO descriptors, followed by memory mapped registers, which use are defined by the descriptors.

At the beginning of the IO space, a number of read-only IO descriptors are stored. A descriptor is TLV coded, or Tag Length Value. The tag indicates the peripheral type, the length indicates the descriptor length, and value describes the peripheral registers and parameters. This encoding allows fast peripheral enumeration and hardware independent access.

These tags are registered:

Tag	Length	Value
0x00	0	End of list. This is the last tag of the list.
0x01	4+...	Serial port. Tag contents is encoded like this: - 2 bytes: I/O port address base - 1 byte: interrupt code - 1 byte: flags (interrupt priority, serial ports options, TBD later) - N bytes and format not defined yet: baud rate (should allow for nonstandard baud rates)

11 Instructions encoding

Instructions are 1-4 bytes wide.

RD = destination register

RS = source register

MD, MS = register access mode. 0= GP register, 1=special register

S, SD, SS = double register access. 0=use 8-bit registers, 1=use RN and RN+1 as a 16-bit register

Y = carry/borrow (for add, sub, rot, shift). 0=do not use carry/borrow, 1=use carry/borrow

LR = left (0) or right (1) for shift and rotate.

Ind = use 8-bit signed index constant (in following byte)

C = index length 0=8 bits index, 1=16 bits index

W=wide access: read/write 16 bits at once with load/store, use a 16-bit constant in load/store, a 16 bits displacement in Bcc, wide multiplication/division result

C = Litteral Constant

Cc = condition code

D = Displacement, 8 bits signed (or 16 bits signed if W=1)

L=Link. 0=Just goto, 1=Push return address before jump

Dir = direction, 0=load/in 1=store/out

load RA, ind(RB) means mem[RB+ind] → RA

Condition codes:

0 0 0	always
0 0 1	Z (EQ)
0 1 0	NZ (NE)
0 1 1	GT
1 0 0	LT
1 0 1	GE
1 1 0	LE
1 1 1	None (RFU)

TODO:

- a TEST_AND_SET opcode may be desirable for multithread operations. But there is no shared memory yet.

First byte (@N)									Second byte (@N+1)								Description	Data
B7	B6	B5	B4	B3	B2	B1	B0		B7	B6	B5	B4	B3	B2	B1	B0		
0	0	0	0	0	0	0	0		N/A								NOP	--
0	0	0	0	0	0	0	1		N/A								RET	--
0	0	0	0	0	0	1	0		N/A								RETI	--
0	0	0	0	1	RD				0	0	S	MD	MS	RS			MOV	--
0	0	0	0	1	RD				0	1	S	MD	MS	RS			XOR	--
0	0	0	0	1	RD				1	0	S	MD	MS	RS			AND	--
0	0	0	0	1	RD				1	1	S	MD	MS	RS			OR	--
0	0	0	1	0	RD				Y	0	S	0	0	RS			ADD(C)	--
0	0	0	1	0	RD				Y	1	S	0	0	RS			SUB(B)	--
0	0	0	1	1	RD				0	0	S	0	0	RS			TEST	--
0	0	0	1	1	RD				0	1	S	0	0	RS			SWAP	--
0	0	0	1	1	RD				1	0	SD	W	SS	RS			MUL	--
0	0	0	1	1	RD				1	1	SD	W	SS	RS			DIV	--
0	0	1	0	0	RD				Y	0	SD	LR	SS	RS			SHIFT	--
0	0	1	0	0	RD				Y	1	SD	LR	SS	RS			ROT	--
0	0	1	0	1	RD				Y	0	SD	LR	Bits				SHIFTC	--
0	0	1	0	1	RD				Y	1	SD	LR	Bits				ROTC	--
0	0	1	1	0	RD				0	0	SD	MD	0	Rn			CLRB	--
0	0	1	1	0	RD				0	1	SD	MD	Bits				CLRBC	--
0	0	1	1	0	RD				1	0	SD	MD	0	Rn			SETB	--
0	0	1	1	0	RD				1	1	SD	MD	Bits				SETBC	--
0	0	1	1	1	RD				0	0	0	MD	0	Rn			TESTB	--
0	0	1	1	1	RD				0	1	0	MD	Bits				TESTBC	--
0	0	1	1	1	0	0	0		1	0	0	0	0	RD			SEXT	--
0	0	1	1	1	0	0	0		1	0	S	0	1	RD			CMPL	--
0	0	1	1	1	0	0	0		1	0	S	1	0	RD			JSR	--
0	0	1	1	1	0	0	0		1	0	0	1	1	0	0	0	RESET	--
0	0	1	1	1	0	0	0		1	0	0	1	1	0	0	1	SLEEP	--
0	0	1	1	1	0	0	0		1	1	0	1	Num				TRAP	--
0	1	0	0	0	0	0	0		Val				S	RD			ADDQ	--
0	1	0	0	0	0	0	1		Val				S	RD			SUBQ	--
0	1	0	0	0	0	1	0		Val				S	RD			MULQ	
0	1	0	0	0	0	1	1		ImpTableIndex								LIBCALLX	FuncNo
0	1	0	0	0	1	FnHi			FuncNoLo				ImpTableindex				LIBCALL	--
0	1	1	Ind	W	RV				C	Dir	SM	MV	MM	RM			LOAD/STORE	(Index)
1	0	0	Ind	W	RV				C	Dir	SM	MV	MM	RM			IOCTL	(index)
1	0	1	MD	W	RD				C-LSB								MOVC	C-MSB
1	1	0	MD	W	RD				C-LSB								TESTC	C-MSB
1	1	1	L	W	Cc				D-LSB								Bcc	D-MSB

4 Instructions description

The global syntax is : OPERAND DESTINATION, SOURCE

[A|B] represents an alternative chose, either A or B.

for example, LOAD [RV|WV], ... means

either LOAD RV, ...

or LOAD WV, ...

Status flags have the following meaning:

Z (zero) is set when the result of an operation is zero

N (negative) is set when the higest bit of the result is set

C (carry) is set when adding two operands that do not fit within the same number of bits of this operand

V (overflow) is set when operands have the same sign, and the result have a different sign

ADD, ADDC

Assembly syntax

ADD RD, RS

ADD WD, WS

ADDC RD, RS

ADDC WD, WS

Effect

Add 2 registers, optionnaly including the carry

Y=0: RS + RD → RD

Y=1: RS + RD + C → RD

Then,

If RD=0, set Z

(TODO trouver les équations des autres flags)

Affected flags:

Z C N V

Encoding

Byte 1

B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	1	0	RD		

Byte 2

B7	B6	B5	B4	B3	B2	B1	B0
Y	0	S	0	0	RS		

RD: destination register number
RS: source register number

Y: use carry in operation
Y=0: do not use carry, operation is ADD
Y=1: use carry, operation is ADDC

S: operation size
S=0: 8-bit operation
S=1: 16-bit operation

ADDQ

Assembly syntax

ADDQ RN, #value

ADDQ WN, #value

Effect

Add a small positive value (1..16) to a general purpose register.

S=0: RN + value → RN

S=1: WN + value → WN

Update flags according to result

Affected flags

Z N C V

Encoding

Byte 1

B7	B6	B5	B4	B3	B2	B1	B0
0	1	0	0	0	0	0	0

Byte 2

B7	B6	B5	B4	B3	B2	B1	B0
VALUE CODE				S	RD		

RD: destination register number

S: size of operation,
S=0 : 8-bit operation
S=1: 16-bit operation

VALUE CODE: This field encodes the value to be added, minus one. Since adding zero has no sense, this encoding allows easy addition of values in range 1..16

AND

Assembly syntax

AND RD, RS

AND WD, WS

Effect

RS AND RD → RD

Bcc

Assembly syntax

BRA[L] offset

BEQ[L] offset

BNE[L] offset

BGT[L] offset

BGE[L] offset

BLT[L] offset

BLE[L] offset

Effect

Branch (and link) if condition is verified. This is a relative jump. For absolute jumps within a module, use JSR.

The offset can be 8 or 16 bits. The value can be auto calculated, or hardcoded using a .L or .S to the opcode. Using Bcc.S will fail if the required offset does not fit within 8 bits.

CLRB,CLRBC

Assembly syntax

CLRB [RD|WD], RB

CLRBC [RD|WD], #val4

Effect

Clear a bit. Bit index in register or in constant.

CMPL

Assembly syntax

CMPL RD

CMPL WD

Effect

Compute two's complement.

SD=0: (RD XOR 0xFF) +1 → RD

SD=1: (WD XOR 0xFFFF) +1 → WD

DIV

Assembly syntax

DIV RD, RS

Effect

Divide registers

IOCTL

Assembly syntax

XXX

Effect

Access I/O memory

JSR

Assembly syntax

JSR Rn

JSR Wn

Effect

Jump to subroutine using a register. This stores the address just after this instruction on the stack, then loads the contents of register RD or WD in PC.

There is no JMP instruction because this one is a simple alias to MOV PC, Wn

LIBCALL

Assembly syntax

LIBCALL index@libname

Effect

Call a function in another module via this module's import table. This compact version can be used to access the first 64 functions of the first 16 imported libraries

LIBCALLX

Assembly syntax

LIBCALLX index@libname

Effect

Call a function in another module via this module's import table. This version is not restricted to 16 libs or 64 functions.

LOAD

Assembly syntax

LOAD [RV|WV], ([RM|WM])

LOAD [RV|WV], offset8([RM|WM])

LOAD [RV|WV], offset16([RM|WM])

Effect

Retrieve memory contents. Used for stack and pointer dereference. The indexed version is useful for struct access.

MOV

Assembly Syntax

MOVE RD, RS

Effect

RS → RD
Copy the contents of a register to another register.

Affected flags

Z	The copied register contained zeros
N	
C	
V	

Encoding

Byte 1

B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	0	1	RD		

Byte 2

B7	B6	B5	B4	B3	B2	B1	B0
0	0	SZ	MD	MS	RS		

RD: destination register number
RS: source register number
S: operation size.
    S=0: operate on 8-bit registers
    S=1: operate on 16-bit register pair
MD: dest register mode
MS:source register mode
    Mx=0: Normal register set
    Mx=1: Special register set

MOVC

Assembly syntax

MOVC RD, #value8

MOVC WD, #value16

Effect

Move a constant value in a register. There is no opcode to store a 8 bit constant in a 16-bit register.

MUL

Assembly syntax

MUL DESTINATION, SOURCE

Effect

Perform 8x8 / 8x16 / 16x16 unsigned multiplication

RD * RS → RD

RD * RS → WD

RD * WS → RD

RD * WS → WD

WD * RS → RD

WD * RS → WD

WD * WS → RD

WD * WS → WD

Then, update flags

Affected flags:

Z C N V

Encoding

Byte 1

B7	B6	B5	B4	B3	B2	B1	B0
0	0	0	1	1	RD		

Byte 2

B7	B6	B5	B4	B3	B2	B1	B0
1	0	SD	W	SS	RS		

RD: destination register number
RS: source register number
SD : size of RD as a source operand
SS : size of RS
W : size of result

MULQ

Assembly syntax

MULQ RD, #val4

Effect

Multiply a register by a small integer in range 2..17

This instruction cannot be used to multiply by zero or one.

NOP

Assembly syntax

NOP

Effect

Assembly syntax	NOP
Effect	Performs no operation. The instruction encoding matches the memory erased state.
Affected flags	None
Instruction length	1 byte
Encoding	Byte 1 0 0 0 0 0 0 0 0

OR

Assembly syntax

OR RD, RS

OR WD, WS

Effect

Same encoding as MOVE, except operation is

RS OR RD → RD

RESET

Assembly syntax

RESET

Effect

Cancel all execution and restart runtime environment

RET

Assembly syntax

RET

Effect

Normal return from subroutine

RETI

Assembly syntax

RETI

Effect

Return from subroutine, also restores the processor status register. Used to exit the supervisor mode.

ROT, ROTC

Assembly syntax

ROT DESTINATION, SOURCE

ROTC DESTINATION, #val4

Effect

Rotate the contents of a register, optionally through carry. Number of places is in a register or in a constant.

SETB,SETBC

Assembly syntax

SETB [RD|WD], RB

SETBC [RD|WD], #val4

Effect

Set a bit. Bit index in register or in constant.

SEXT

Assembly syntax

SEXT RD

Effect

Sign extend 8-bit register to 16-bit

RN[7]=0: 0x00 || RN → WN

RN[7]=1: 0xFF || RN → WN

SHIFT, SHIFTC

Assembly syntax

SHIFT DESTINATION, SOURCE

SHIFTC DESTINATION, #val4

Effect

Shift contents of a register, optionally through carry. Number of places is in a register or in a constant.

SLEEP

Assembly syntax

SLEEP

Effect

Go into low power mode until an interrupt wakes the processor.

STORE

Assembly syntax

STORE [RV|WV], ([RM|WM])

STORE [RV|WV], offset8([RM|WM])

STORE [RV|WV], offset16([RM|WM])

Effect

Transfer the contents of a register into memory.

SUB, SUBB

Assembly syntax

SUB RD, RS

SUB WD, WS

SUBB RD, RS

SUBB WD, WS

Effect

Same encoding as MOVE, except operation is

Y=0: RS – RD → RD

Y=1: RS – RD – C → RD

Affected flags: Z C N V

SWAP

Assembly syntax

SWAP RD

SWAP WD

Effect

Same encoding as MOVE, except operation is

S=0: swap nibbles in 8-bit register RS and store in RD

S=1: swap contents of registers RS and RD

TEST

Assembly syntax

TEST RD, RS

Effect

Same encoding as MOVE, except operation is

Y=0: Compute RS – RD

Y=1: Compute RS – RD – C

Do not update RD

Update flags

Affected flags: Z C N V

TESTC

Assembly syntax

TEST RD, #value8

TEST WD, #value16

Effect

Test a register against a constant

TESTB,TESTBC

Assembly syntax

TESTB [RD|WD], RB

TESTBC [RD|WD], #val4

Effect

Test a bit. Bit index in register or in constant. Result in Zero, so that BNE/BEQ can be used to jump. Just like AND, but can accept a constant and does not alter the tested register.

TRAP

Assembly syntax

TRAP #val4

Effect

Switch to supervisor mode while calling into the kernel.

XOR

Assembly syntax

XOR RD, RS

Effect

Same encoding as MOVE, except operation is

RS XOR RD → RD

5 Assembly syntax

1 Instructions

The syntax for each instruction is detailed in the instruction's descriptions.

2 Symbols

Valid symbols are matching the regex: [A-Za-z][A-Za-z0-9]*, maximum length is 64 bytes.

Symbols are either code addresses or data symbols.

Evm8 is a load store machine, a symbol is an address. Unlike with 68k , there is no “LEA” instruction, because this is what movc does:

```
movc R0, label
```

does not mean: mem[label] → R0

but rather : label → R0

The 68k instruction move.b label, d0 requires 2 evm8 instructions:

```
movc R1, label          ; label → R1
load R0, R1             ; mem[R1] → R0
load R0, 3(R1) ; mem[R1+3] → R0
```

3 Directives

Directives are commands that do not lead directly to binary code, instead they change the behaviour of the assembler.

.module	Define executable module name. Only allowed once.
.equ SYM, VAL	Define a constant SYM with value VAL.
.include ”path”	Include an external file at this point
.xdef SYM	Mark symbol SYM as being global (visible by other files)
.global SYM	Alias for .xdef
.text [NAME]	Following data and code will go in the (possibly named) code section
.rodata [NAME]	Following data will go into the (possibly named) rodata section
.data [NAME]	Following data will go into the (possibly named) initialized data section
.bss [NAME]	Following data will go in the (possibly named) bss section
.db VAL [,VAL]+ .byte	Store a byte verbatim
.dw VAL[,VAL]+ .word	Store a word (2 bytes) verbatim
.dl VAL[,VAL]+ .long	Store a long word (4 bytes) verbatim
.ds VAL .space	Store a number of zero bytes
.asciiz “VAL”	Store a null terminated string

In the future we will also support .macro … .endmacro

4 Instructions

Source lines can be:

-empty lines

-comment lines, starting by any of: # ! ; @ //

-a directive

-an instruction

6 Executable format

To allow efficient and modular execution, a specific executable format is defined.

There is no difference between libraries and programs. A library is a program with entry points, whose only executable instruction is « RET ».

Note that this is just an interchange format. It may differ from what is really stored in the target system's memory.

Offset	Length	Description
0	8	Module name: a 8 bytes identifier for the library or program, preferably ASCII but not required.
8	2	RAM SIZE: the number of bytes that must be allocated for this program for both BSS and initialized DATA.
10	2	Code size: the number of code bytes
12	1	Import table size: the number of imported libraries (n).
13	1	Export table size: the number of exported functions (p).
14	2	Reserved, must be 0x0000
16	8*n	Names of imported libs: 8 bytes per name
16+8*n	2*p	Exported PCs: 2 bytes per entry point
16+8*n+2*p	C	Executable code

7 Relocatable format

1 Format

A specific relocatable object format has been defined to allow linking of multiple object files in a single binary program or library.,

Relocatable files format is as follows:

Offset	Length	Description
0	4	Magic « REL8 »
4	2	Exported symbols count
6	2	Relocation count

Relocations

Relocations are of several types

-

todo

It is believed that this assembler will be integrated as a code generator backend for LLVM, to benefit from the good compilation quality of the CLANG compiler.

The only problem is the harvard architecture, wether it's acceptable for clang it is not known yet, and how to declare exports and imports. An SDCC backend is an alternative, it already supports harvard architectures and builtins for specific opcodes.

8 References

Carry and Overflow
http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Comb/overflow.html

Influences

68k user manual

javacard runtime environment