8/16 bits virtual machine.

Introduction

The goal is to define an embedded machine that can execute code on different 8-bits architectures, as well as normal CPU, such as an FPGA implementation.

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.

System architecture

CPU

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

- The code space holds executable instructions and constant data (TXT et CST);
- The data space holds the stack and module data (BSS and DAT);
- 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.

Memory

The internal address paths are 24-bit wide. The data paths are 8-bit wide.

The Code space is a global non volatile memory array. A small part of this memory holds the module registry. The remaining space can be used to store executable modules. A best fit strategy and/or a defragmentation method can be used to allocate memory, provided that the module registry is properly updated. 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.

The data space is volatile memory allocated to the executable module. It is also a global memory pool. At load time, each module is allocated a start address for its BSS variables, that points to the beginning of a memory zone usable for the module. Within the code, BSS variables are targeted using a zero based offset, which is added to the VM-managed BSS address for this module. In user modes, BSS accesses are checked so that any module cannot use memory that does not pertain to the same module.

Execution context

One or more execution context can be active at once to support single or multithreading. Scheduling and stack allocation are not defined yet.

An execution context stores the volatile information for the currently running thread, including the current module being executed, and all registers, including PC and SP.

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

Modules

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

Each module can only access the opcodes that are stored in the same module. The PC register is 16-bits wide and is range checked against the current module. The module load address, and the fact that this physical address is wider than the user available PC is unknown to the code. This ensures that no code can be fetched from outside the module. Instead, code from other modules may be called via import and export tables.

Module registry

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. A module can hold a maximum of 65535 code bytes.

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

Any module can export functions. This is a list of program counter offset values that mark exported functions, indexed by a 16-bit number.

Non volatile (NV) module registry

The NV Module registry is a table, holding permanent 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 holding modules attributes:

- module name (8 bytes)
- module position in memory (system dependent, typically 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.

Boot

At startup, the VM does the following:

- Define the top of the supervisor stack at address 128. This can be changed if a bigger supervisor stack is required.
- Affect a BSS RAM block for each loaded module, and save their address in the volatile module registry.
- The top of the user stack is defined at the very end of the memory. 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.
- 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.
  - if a default module is installed, the default module is executed in user mode.
  - else, the system is put in SLEEP mode.

- If there is no kernel module
  - if a default module is installed, the default module is executed in supervisor mode.
  - Else, the system is halted.

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.

Else, the exported function is executed in supervisor mode.

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	0
Bad instruction	1
Address error (tried to jump in the wild)	2
Stack under/overflow	3
Module not found	4
Trap #0	16
Trap #16	31
Interrupt #n	32

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	

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 SSP and USP registers are zero based, and the real memory is made by substracting the contents of the current SP register from the real stack address. In the same time, stack over/underflows are checked and reported.

PUSH is postinc, POP is predec. For the user, the stack addresses start at zero and grow when pushing data. In real memory however, the stack addresses start at the highest possible adresses (top of stack) and shrink as data is pushed.

When the SM bit in the Processor Status Register is set, and not using an index, any LOAD or STORE instruction requesting access from the SP register will alter the SP register as expected from this stack definition. 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 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.

Special registers

The CPU 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, 16 bits wide within a module,
SP / SSP	0 0 1	16	In user mode, this is an alias for USP. In supervisor mode, this is the Supervisor stack pointer, 16 bits wide.
USP	0 1 0	16	User mode stack pointer (always)
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.

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
B2	I	Global Interrupt enable
B3	B	Endianess control (0=LE, 1=BE)
B4	SM	Stack 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 or default application.

IO space

IO peripherals are abstracted from the underlying architecture. A number of peripheral may be accessed in a platform independent way, using 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. This encoding allows fast peripheral enumeration.

These tags are registered:

Tag	Length	Value
0x00	0	End of list. This is the last tag of the list.
0x01	2	Serial port. Tag contents is encoded like this: - 2 bytes: I/O port address base

Instructions table

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)

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	SS	Rn			CLRB	--
0	0	1	1	0	RD				0	1	SD	MD	Bits				CLRBC	--
0	0	1	1	0	RD				1	0	SD	MD	SS	Rn			SETB	--
0	0	1	1	0	RD				1	1	SD	MD	Bits				SETBC	--
0	0	1	1	1	RD				0	0	0	MD	SD	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			ImpTableindex				FuncNo				LIBCALL	--
0	1	1	Ind	W	RD				C	Dir	DB	MD	MS	RS			LOAD/STORE	(Index)
1	0	0	Ind	W	RD				C	Dir	DB	MD	MS	RS			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

Instructions description

The global syntax is : OPERAND DESTINATION, SOURCE

ADD, ADDC

Same encoding as MOVE, except operation is

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

ADDQ

Add a small value to a register

AND

Same encoding as MOVE, except operation is

RS AND RD → RD

Bcc

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

CLRB,CLRBC

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

CMPL

Compute two's complement.

SD=0: (RS XOR 0xFF) +1 → RD

SD=1: (WS WOR xFFFF) +1 → WD

DIV

Divide registers

IOCTL

Access I/O memory

JSR

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

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

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

LOAD

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	None
Instruction length	2 bytes
Encoding	Byte 1

0	0	0	0	1	RD

Byte 2

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

Move a constant value in a register

MOVQ

load a small value to a register

MUL

Multiply registers

NOP

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

Same encoding as MOVE, except operation is

RS OR RD → RD

RESET

Cancel all execution and restart runtime environment

RET,RETI

Return from subroutine, (normal, from supervisor mode)

ROT, ROTC

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

SETB,SETBC

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

SEXT

Sign extend 8-bit register to 16-bit

RN[7]=0: 0x00 || RN → WN

RN[7]=1: 0xFF || RN → WN

SHIFT, SHIFTC

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

SLEEP

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

STORE

SUB, SUBB

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

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

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

Test a register against a constant

TESTQ

Test a register against a small constant

TESTB,TESTBC

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

Switch to supervisor mode while calling into the kernel.

XOR

Same encoding as MOVE, except operation is

RS XOR RD → RD

Assembly syntax

Instructions

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

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
```

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

Instructions

Source lines can be

-empty lines

-comment lines, starting by #, !, @ or //

-a directive

-an instruction

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 ».

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

Relocatable format

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