DvmAsm, The Desi Virtual Machine Assembler

Brian Shearing

Peter Grogono

13 November 2015

The machine

DVM, the Desi Virtual Machine, is a computer not unlike, but a little more regular than, Intel's IA32. It has:

- a condition register, $\mathcal{C}$, with values *eq*, *ne*, *lt*, *ge*, *le*, and *gt*, representing the result of comparing a pair of values;
- four thirty-two bit general-purpose integer registers, `ax`, `bx`, `cx` and `dx`;
- four special purpose registers: `bp` is the base of the current frame, `pp` of the current process, `sp` of the current stack, and `tp` of the current thread;
- a sixty-four bit floating-point accumulator, $\mathcal{A}$;
- a stack of thirty-two bit values, for temporary evaluation of expressions; and
- a data region to hold global variables and literals such as texts and reals.

The assembler

DvmAsm is an assembler for a DVM. A DvmAsm script comprises a sequence of *statements* and *macros*, where a macro is a sequence of statements introduced by a `mac` statement and ended by an `end` statement, as shown in Figure 1. Macro's may not be nested, although their invocations may be.

script ::= { *sequence* | *macro* }.
macro ::= `mac`-*statement sequence* `end`-*statement*.
sequence ::= { *statement* }.

Figure 1: Outer-level syntax

A macro may have parameters which are denoted '`$0`', '`$1`', '`$2`' and so on within the body of the macro, where they may be used to take the place of parts of addresses, of complete operands, or of the operator of the statement—in the latter case the actual argument being the name of another macro.

Lines of a script may be blank, and may include commentary. A comment starts with a percentage symbol and ends at the end of the line.

Statements

A DvmAsm statement is a *declaration* or an *operation*, as summarised in Table 1.

A declaration must have a *label*, which is an identifier. An operation may have a label. Labels start in column one, which is otherwise blank. A label starting with an upper-case letter denotes a synonym for a constant, is declared by an `equ` statement, and is denoted by *N* in the table. A label starting with a lower-case letter denotes a global quantity—an address or a macro—and is denoted by *n*. A label starting with an underscore denotes a local address within the code region, and is denoted by *_n*. The names of registers (see Table 2) are reserved and may not be used as labels.

Statement		Effect	Mode
	com *u*	Start of component *u*, where *u* is a UTF-8 text.	—
	stk *k*	End of component using *k* bytes of instance data.	—
N	equ *k*	Set *N* to have value of integer expression *k*.	*absolute*
n	mac *k*	Set *n* to name of macro with *k* parameters.	*macro*
n	ext *u*	Set *n* to address of external named *u*, a UTF-8 text.	*external*
n	fpt *f*	Plant 64-bit floating-point *f* in data region; set *n*.	*data*
n	txt *t*	Plant Desi text *t* in data region; set *n* to address.	*data*
n	utf *u*	Plant UTF-8 text *u* in data region; set *n* to address.	*data*
n	wrd *k*	Plant 32-bit integer *k* in data region; set *n* to address.	*data*
n	bss *k*	Plant *k* unset bytes in data region; set *n* to address.	*data*
n	*s*	Plant operation *s* in code region; set *n* to address.	*global*
_*n*	*s*	Plant operation *s* in code region; set _*n* to address.	*local*
	s	Plant operation *s* in code region.	—
	end	Check local labels are defined, then discard them.	—

Table 1: Declarations and statements. (See Table 4 for definition of operation *s*.)

Syntax

Figure 2 shows the syntax of statements, whether declarations or operations. An *identifier* is *N*, *n* or _*n* as shown in Table 1. When an identifier appears as an operator it must be the name of a macro already encountered; forward references to macros are not permitted.

statement	::=	[*identifier*] *operator* { *operand* }.
operator	::=	'add' ... 'zer' | *identifier* | *parameter*.
operand	::=	*identifier* | *literal* | *register* | '[' *address* ']' | *parameter*.
literal	::=	*text-literal* | *real-literal* | *word-literal*.
text-literal	::=	TEXT.
real-literal	::=	'#' [*sign*] REAL.
word-literal	::=	[*unary-op*] *term* { *adding-op term* }.
term	::=	*factor* { *multiplying-op factor* }.
factor	::=	*constant* | '(' *word-literal* ')'.
unary-op	::=	*sign* | '~'.
adding-op	::=	*sign* | '|' | '^'.
multiplying-op	::=	'*' | '/' | '&'.
sign	::=	'+' | '-'.
address	::=	*address-term* { '+' *address-term* }.
address-term	::=	*address-factor* { '*' *address-factor* }.
address-factor	::=	*register* | *constant*.
constant	::=	*identifier* | *option* | *parameter* | INTEGER.
option	::=	'@' LOWER-CASE-LETTER.
parameter	::=	'$' INTEGER.
register	::=	'ax' | 'bx' | 'cx' | 'dx' | 'bp' | 'pp' | 'sp' | 'tp'.

Figure 2: Statement syntax

The styles of INTEGER, REAL and TEXT are as in Desi.

Unary operator '`~`' denotes bit-wise negation. Binary operator '`|`' is bit-wise *or*, '`^`' is exclusive *or*, and '`&`' is bit-wise *and*.

A macro argument corresponding to a parameter that is an address-factor must be a literal word, a name equated to a literal word, or a register. A macro argument corresponding to a complete operand may be a general integer expression, or a scalar value such as a register or a text or real value.

An address-factor may include at most one register, whether written explicitly or provided as a parameter. If an address-factor does include a register then the product of all constants of the factor must evaluate to one, two, four or eight. An address may include at most one address-factor containing a register and at most one isolated register.

A constant of the form '`@`*o*', where *o* is a letter corresponding to a command line option, yields zero if the option is omitted or unity if set. The special case `@p` yields the number of processors, or the number given by the `p`-option.

Operands

Table 2 illustrates the forms of an operand.

Kind	Denotation	Meaning
literal	k	32-bit integer expression k
(Signature `K`	`$`n	argument n of current macro
in Table 4)	`@`o	value of option o of command line
	N	32-bit integer named *N* in an `equ`-statement
	`#`r	64-bit floating-point constant r (may be signed)
	t	text, in quotes, as in Desi source
register	`ax bx cx dx`	general-purpose 32-bit register
(Signature `R`)	`bp pp sp tp`	special-purpose 32-bit register
store	*n*	global code or global data address *n*[1]
(Signature `S`)	`_`*n*	local code address `_`*n*
	`[`$r_1 + sr_2 + c$`]`	address $r_1 + s \times r_2 + c$

[1] In some instructions a store address is considered to be a literal, k, as in '`mvw [bx] adr`' where `adr` is an address of an item of data.

Table 2: Operands

When a store operand has the form *n* or `_`*n* the label referred to may be a forward reference.

The scaling factor s in a computed address (in square brackets) must be 1, 2, 4 or 8.

The constant c in a computed address may be *N*, *n* or an integer. When referring to a symbolic value (cases *N* and *n*) the symbol must be defined; forward references are not permitted. Any but not all of the three parts forming a computed address may be omitted. The parts may be in any order. Examples include `[poolSize]`, `[srcName+pp]`, `[bp+$3]` and `[4dx]`. There may be only one instance of each kind of address part, with the exception of constants which may occur many times, for example `[bp+VAR+8]` where `VAR` is a constant.

Examples

Table 3 illustrates some DvmAsm statements and their corresponding IA32 code.

Example	Data and Code	Auxiliary Effect
`wVal wrd 255`	`0x100: 0x000000FF`	`wVal` = *data* 0x100
`wVal wrd ~255`	`0x100: 0xFFFFFF00`	`wVal` = *data* 0x100
`X equ 64` `wVal wrd 3+4*(X-1)`	 `0x100: 0x000000FF`	`X` = *absolute* 64 `wVal` = *data* 0x100
`fVal fpt #+0.5`	`0x100: 0x00000000` `0x104: 0x3Fe00000`	`fVal` = *data* 0x100
`uVal utf "a"`	`0x100: 0x61, 0x00`	`uVal` = *data* 0x100

`tVal txt "a"`	`0x100: 0x00000061`
	`0x104: 0x00000001`
	`0x108: 0x00000001`
	`0x10C: 0x00000001`
	`0x110: 0x00000100`
		`tVal` = *data* 0x104
		(The address of the text, 0x104, points to a four word block, the last word of which, 0x110, points to the characters of the text.)

`nop`	`0x200: nop`	
`neg dx`	`0x200: neg edx`	
`neg [wVal]`	`0x200: neg d[0x100]`	
`VAR equ 8` `neg [bp+VAR+4]`	 `0x200: neg d[ebp+12]`	`VAR` = *absolute* 8

`temp mac 1`
`mvw ax [bp+$0]`
`end`
`temp 8`
	
	`0x200: mov eax d[ebp+8]`	

`neg [dx+4*bx+8]`	`0x200: neg d[edx+4*ebx+8]`	

`temp mac 1`
`neg [dx+4*$0+8]`
`end`
`temp bx`
	
	`0x200: neg d[edx+4*ebx+8]`	

`temp mac 1`
`neg [dx+4*$0+8]`
`end`
`temp 10`
	
	`0x200: neg d[edx+48]`	

`logD ext "log10"` `inv logD`	 `0x200: call 0x12345678`	`logD` = *external* 0x12345678
`rtnX ...` `inv rtnX`	`0x200: ...` `0x248: call 0x200`	`rtnX` = *global* 0x200
`jmp _6` `_6 ...`	`0x200: jump 0x248` `0x248: ...`	 '_6' = *local* 0x248

Table 3: Examples, showing data at 0x0100 and code at 0x200

Operations

Table 4 shows DvmAsm operations, their meanings, and the *signatures* accepted by the assembler. A signature is a combination of the kinds of operand shown in Table 2, supplemented by signature `O` denoting an absent operand. Operands are four-byte signed words unless the suffix *b* indicates a byte, or suffix *f* an eight-byte floating-point value.

Operation	Effect	O	K	R	S	RK	RR	RS	SK	SR	SS
Data manipulation											
`mvb` p q	p_b := q_b					✓	✓	✓	✓	✓	
`mvw` p q	p := q					✓	✓	✓	✓	✓	✓
`lea` p q	p := `@`q							✓			
`xch` p q	p :=: q						✓	✓		✓	
`axc` p q	atomic p :=: q							✓		✓	
Word arithmetic											
`add|sub` p q	p := p +|- q					✓	✓	✓	✓	✓	
`mul|div|mod` p q	p := p ×|÷|% q					✓	✓	✓			
`and|orr|xor` p q	p := p and|or|xor q					✓	✓	✓	✓	✓	
`asl|asr|lsr` p q	p := p <<|>>|>>> q					✓	✓	✓			
`neg|not|zer|inc|dec` p	p := -p |!p|0|p+1|p-1			✓	✓						
`anc|adc` p	atomic p := p+1|p-1				✓						
Floating-point arithmetic											
`fld` p	$\mathcal{A}$:= p_f				✓						
`fst` p	p_f, $\mathcal{A}$:= $\mathcal{A}$, ϕ				✓						
`fad|fsb|fml|fdv|fmd` p	$\mathcal{A}$:= $\mathcal{A}$ +|-|×|÷|% p_f				✓						
`fng|fzr|fun`	$\mathcal{A}$:= -$\mathcal{A}$|0.0|1.0	✓									
Extensions and coercions											
`mzb` p q	p := zero-extended q_b					✓	✓	✓			
`cwf` p	$\mathcal{A}$:= float p			✓							
`cei|flr|rnd|trc` p	p, $\mathcal{A}$:= ceil|flor|roun|trun $\mathcal{A}$, ϕ			✓							
Comparisons											
`beq|bne|blt|bge|ble|bgt` p	p_b := $\mathcal{C}$ is =|<>|<|>=|<=|>			✓							
`cmp` p q	$\mathcal{C}$:= $p \sim q$					✓	✓	✓	✓	✓	
`fcm` p	$\mathcal{C}$, $\mathcal{A}$, r0 := $\mathcal{A} \sim p$, ϕ, ϕ				✓						
Control											
`nix|nop`	skip|no-op	✓									
`inv` p $q_0 \ldots q_{n-1}$ [1]	push q_{n-1}; ... push q_0; call p			✓	✓[2]						
`ret`	return	✓									
`jmp` p	goto p			✓	✓						
`jeq|jne|jlt|jge|jle|jgt` p	goto p if $\mathcal{C}$ is =|<>|<|>=|<=|>				✓[3]						
`jcz` p	goto p if cx zero				✓[4]						
`jmx` p $q_0 \ldots q_{n-1}$	goto $q[p]$							✓			
Stack manipulation											
`alc|dlc` n	allocate|deallocate n bytes		✓								
`pop` p	pop p			✓	✓						
`fpp|fps`	pop $\mathcal{A}$ | push $\mathcal{A}$; $\mathcal{A}$:= ϕ	✓									
`psh` p	push p		✓	✓	✓						

[1] arguments q_i optional.
[2] p must be mode *global* or *external*; if global then reference may be forward.
[3] p must be mode *global* or *local*; reference may be forward in either mode.
[4] p must be mode *local*; reference may only be backwards, and short (≤128 bytes).

Table 4: DvmAsm operations, semantics and signatures

Although `psh`- and `pop`-operations may be employed to store temporary values, the stack must be empty whenever communication takes place between processes, or at any other time when a process may become suspended. The stack is not preserved while a process waits, nor are the values in the floating-point register $\mathcal{A}$, the condition register $\mathcal{C}$, and the general registers `ax`, `bx`, `cx` and `dx`.

Execution

The name of the assembler is *DesiRun.exe*. It depends on the two libraries *DesiLib.dll* and the third-party library *BeaEngine.dll*, the latter acting as disassembler. In the examples below all three files are in a directory named `run` along with the assembler script to be executed.

The assembler reads a LaTeX file, treating lines in 'verbatim' environments as source, and ignoring others. The command to effect assembly and execution is '`DesiRun t`', where *t* is the name of a LaTeX file, anadorned by its extension of '`.tex`'.

The left panel of Figure 3 shows a DvmAsm program embedded within a LaTeX script, the script in this example being `dvmspec.tex`—the source of this document. The right panel shows two executions of the program.[1]

```
...
\begin{verbatim}
scrln ext "scrln"
hello txt "Hi-di-\u03C0!\n"
      psh hello
      inv scrln
      dlc 4
      zer ax
      ret
\end{verbatim}
...
```

```
c:run>DesiRun dvmspec
Hi-di-π!

c:run>DesiRun +e +m +n dvmspec
0000 push 00000034h
0005 call 6CD4A264h
000A add esp, 04h
000D xor eax, eax
000F ret

0010 48 00 00 00 H...
0014 69 00 00 00 i...
0018 2D 00 00 00 -...
001C 64 00 00 00 d...
0020 69 00 00 00 i...
0024 2D 00 00 00 -...
0028 C0 03 00 00 ....
002C 21 00 00 00 !...
0030 0A 00 00 00 ....
0034 09 00 00 00 ....
0038 09 00 00 00 ....
003C 09 00 00 00 ....
0040 10 00 00 00 ....
```

Figure 3: A DvmAsm program and two executions

Of the options associated with compilation, assembly and execution, the assembler responds to `c`, `e`, `f`, `l`, `m`, `n`, `r`, `s`, `t`, `u` and `v`. (See the Desi SDK, or type '`DesiRun -h`', for more details.) The second example of Figure 3 writes a report to the console (option `+e`) comprising the generated machine code (`+m`) with no execution (`+n`).

The source of a DvmAsm program may be split across several files linked together by `\insert` statements, which must appear in the LaTeX regions of the files and not in the DvmAsm regions. Inserted files may themselves include `\insert` statements, and so on recursively.

The phrases `\begin{verbatim}`, `\end{verbatim}` and `\insert` must start in the first column of their respective lines.

[1] By these simple examples we demolish a standard result of computing theory. You *can* execute a specification. `;)`