Vaguely Eleven-Like Machine Architecture (VELMA)

version of 27 October 2008

Overall notes

VELMA is a word-addressed machine. Words are 24 bits. Addresses are 15 bits. Memory goes from address 10_8 to 77777_8 (32767), inclusive (although 10 through 77 are memory-mapped i/o words and interrupt vectors).

There are eight 24-bit registers, named R0 through R7. These registers can also be referred to by memory addresses 0 through 7, respectively (this idea is taken from the PDP-10). R6 is used as the stack pointer and should not be used for other purposes.

Indirect addressing ignores the extraneous top 9 bits of the indirect value.

The resemblance to the PDP-11 (''the Eleven'') is not always substantial.

Most numbers in this document are in base eight, except when they're obviously in base two. The assembler considers numbers to be in base eight.

The ops

Here are the available ops and their assembly-language mnemonics. In the assembly-language notation, we always write ''OP SRC, DST''. In some cases, ops with different opcodes have the same mnemonic if the syntax makes it clear which is which. For example, ''ADD 123, R1'' refers to the first ADD op, whereas ''ADD R1, 123'' refers to the second.

The opcodes are listed in binary after the mnemonic. Except for the branch instructions, the opcode is the upper six bits of the instruction word; then comes three bits indicating the register number (''reg''); then a fifteen-bit address (''mem''). For the branch instructions (i.e. the instructions which start with a 'B', not including those which start with 'BI'), the upper nine bits are opcode and the remaining fifteen bits are the address.

ADD (000000) R[reg] ← R[reg] + M[mem]

Add the value in the memory location into the register.

ADD (000001) M[mem] ← M[mem] + R[reg]

Add the value in the register into the memory location.

SUB (000010) R[reg] ← R[reg] − M[mem]

Subtract the value in the memory location from the register.

SUB (000011) M[mem] ← M[mem] − R[reg]

Subtract the value in the register from the memory location.

MUL (000100) R[reg] ← R[reg] × M[mem]

Multiply the value from the memory location into the register. Unlike on most real CPUs, this multiply op produces a 24-bit result (the same size as the operands).

DIV (000101) R[reg] ← R[reg] div M[mem] **and** R[reg+1] ← R[reg] mod M[mem]

Divide the value in the register by the value in the memory location, producing a quotient and a remainder. The register number **must** be even. The quotient is stored in the specified register, and the remainder is stored in the *next* register (i.e. *reg*+1).

CMP (000110) R[reg] − M[mem]

Subtract the value in the memory location from the value in the register, without storing. See ''Condition codes'' section.

CMP (000111) M[mem] − R[reg]

Subtract the value in the register from the value in memory location, without storing. See ''Condition codes'' section.

TST (001000) R[reg]
Test the value in the register. The mem field is ignored. See ‘‘Condition codes’’ section.

TST (001001) M[mem]
Test the value in the memory location. The reg field is ignored. See ‘‘Condition codes’’ section.

BIT (001010) R[reg] ∧ M[mem]
Test the ANDing together of the values in the register and in the memory location. See ‘‘Condition codes’’ section.

———————————

MOV (001100) R[reg] ← M[mem]
Copy the value from the memory location to the register.

MOV (001101) M[mem] ← R[reg]
Copy the value from the register to the memory location.

EXCH (001110)
Simultaneously copy the value from the memory location to the register and from the register to the memory location.

MOV# (001111) R[reg] ← mem
Treat the contents of the instruction’s memory location field not as an address but simply as a value, and put that value into the register. In the assembly language notation, the ‘#’ can be written as a prefix of the number, rather than as a suffix of the name ‘MOV’. Assembly-language syntax example: ‘‘MOV #23, R2’’.

@MOV (010000) M[mem] ← M[R[reg]]
Treat the contents of the register as an address, and copy the value from *that* address to the memory location. In the assembly language notation, the ‘@’ can be written as a prefix of the register, rather than as a prefix of the name ‘MOV’; or more commonly, instead of using the ‘@’ symbol at all, the register can be parenthesized. E.g. ‘‘MOV (R2), 123’’ is the same as ‘‘MOV @R2, 123’’, and this is the "@MOV" op.

MOV@ (010001) M[R[reg]] ← M[mem]
Treat the contents of the register as an address, and copy the value from the memory location to that address. In the assembly language notation, the ‘@’ can be written as a prefix of the register, rather than as a suffix of the name ‘MOV’; or more commonly, instead of using the ‘@’ symbol at all, the register can be parenthesized. E.g. ‘‘MOV 123, (R2)’’ is the same as ‘‘MOV 123, @R2’’, and this is the "MOV@" op.

MOV autodecrement (010010) first R[reg] ← R[reg]−1, then M[R[reg]] ← M[mem]
Indirect-target move with pre-autodecrement, like C’s ‘‘*−−p’’.
Assembly-language syntax example: ‘‘MOV 123, −(R2)’’.

MOV autoincrement (010011) first M[mem] ← M[R[reg]], then R[reg] ← R[reg]+1
Indirect-source move with post-autoincrement, like C’s ‘‘*p++’’.
Assembly-language syntax example: ‘‘MOV (R2)+, 123’’.

———————————

ASHR (010101) R[reg] ← R[reg] right-shifted mem times, arithmetically
Arithmetic right-shift. The mem field does not indicate a memory address, but rather, how many bits to shift. The value is simply stated in the assembly-language syntax, e.g. ‘‘ASHR 1, R2’’. The unsigned 15-bit shift value must be less than 24.

ASHL (010110) R[reg] ← R[reg] left-shifted mem times, arithmetically
Arithmetic left-shift. As ASHR.

LSHL (010111) R[reg] ← R[reg] left-shifted mem times, logically (sign bit is not special)
Logical left-shift. As ASHR and ASHL.

CLR (011000) R[reg] ← 0
The specified register is cleared (made zero). The mem field is ignored.

INC (011001) R[reg] ← R[reg]+1
The specified register is incremented by one. The mem field is ignored.

DEC (011010) R[reg] ← R[reg]−1
The specified register is decremented by one. The mem field is ignored.

BIC (011100) R[reg] ← R[reg] $\wedge$ $\overline{\text{M[mem]}}$
‘‘Bit clear’’—the value in the register is modified by clearing to 0 all of the bits corresponding to bits which are 1 in the value in the memory location, and leaving alone all of the bits which correspond to bits which are 0 in the value in the memory location. (Really, this is better explained by the register transfer notation above.)

BIS (011101) R[reg] ← R[reg] $\vee$ M[mem]
‘‘Bit set’’—like BIC, except that the indicated bits are set in the register, not cleared.

BIF (011110) R[reg] ← R[reg] $\oplus$ M[mem]
‘‘Bit flip’’—like BIC, except that the indicated bits are flipped in the register, i.e. set if they were clear or cleared if they were set.

———————

JUMP (110000) PC ← mem

JUMP@ (110001) PC ← M[mem]

JSR (110010) R6 ← R6−1, then M[R6] ← PC, then PC ← mem
Jump to subroutine.

RTS (110100) PC ← M[R6], then R6 ← R6+1
Return from subroutine.

RTI (110101) PC ← M[R6], then R6 ← R6+1, then PSW ← M[R6], then R6 ← R6+1
Return from interrupt.

———————

Branch instructions (111000 and 111001)
See ‘‘Branch instructions’’ section, after the condition codes section.

HALT (111111)
The CPU is halted until the operator presses the green START button.

Condition codes

VELMA has four condition codes, named N, Z, C, and V.

The first six ops (000000 through 000101), INC, and DEC set the condition codes in accordance with the arithmetic operation which was performed:

N = the most-significant bit of the result;
Z = whether the result is zero;
C = carry-out occurred;
V = overflow occurred.

CLR sets Z and clears N, C, and V.

The test ops (TST, CMP, and BIT) set the condition codes in accordance with the value being tested:

N = the most-significant bit of the result;
Z = whether the result is zero;
CMP sets or clears C and V in accordance with the subtraction operation performed;
TST and BIT clear V (i.e. make it zero (false)) and don't affect C.

Branch instructions

Mnemonic	Name	Binary opcode including reg field	Branch condition
BR	Branch	111000000	unconditional
BNE	Branch on not equal	111000001	Z = 0
BEQ	Branch on equal	111000010	Z = 1
BGE	Branch on greater than or equal to	111000011	(N ⊕ V) = 0
BLT	Branch on less than	111000100	(N ⊕ V) = 1
BVC	Branch on overflow clear	111000101	V = 0
BVS	Branch on overflow set	111000110	V = 1
BCC	Branch on carry clear	111000111	C = 0
BCS	Branch on carry set	111001000	C = 1
BGT	Branch on greater than	111001001	(Z ∨ (N ⊕ V)) = 0
BLE	Branch on less than or equal to	111001010	(Z ∨ (N ⊕ V)) = 1