THE VISTA STRUCTURED ASSEMBLER

Author: J Kershaw

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

January 1992
Reports quoted are not necessarily available to members of the public or to commercial organisations.
SUMMARY

VISTA is a structured assembly language for the VIPER microprocessor chip. Though the syntax of VISTA and the appearance of VISTA programs are reminiscent of a high-level language, the actual statements are VIPER machine instructions. VISTA provides the clarity and much of the convenience of a true high-level language without the need for a complex and possibly untrustworthy compiling program.

This Report describes the VISTA language by means of an annotated example, and gives instructions for using the VISTA translator. A formal grammar of VISTA, and descriptions of VIPER and the VIPER Object Program format, are included.
Contents

1 Introduction 2
2 VISTA program example 3
3 Notes on the example (by line number) 6
4 Constructs not included in the example 11
5 Commentary on the grammar of VISTA 12
6 VISTA Grammar 18
7 Using the VISTA translator 25
8 VIPER Machine Definition 26
   8.1 ALU operations ................................ 27
   8.2 Comparisons ................................... 28
9 VIPER Object Program (.VOP) Format 29
References 31
1 Introduction

VISTA is a high-level assembly language: its statements are essentially VIPER machine instructions [Ker 87] embedded in a syntax resembling that of Algol 68. While it does not hide the structure of the underlying machine in the way that a true high-level language would, it does perform many of the routine housekeeping functions which can otherwise distract the programmer and make mistakes more likely.

The advantage of a relatively simple language like VISTA is that it does not need a complex compiler. Not only is the VISTA translator very much smaller and simpler than a typical compiler, but also its outputs are similar enough to the inputs to be checked easily. The outputs are printable, and annotated in such a way that mechanical checking is possible.

VISTA is not an inherently safe language, in the sense that some specialised languages [Cur 84] might be, though it does limit the programmer to structures which are reasonably free of complications. For example, there are no pointers or GOTO statements in VISTA. If VISTA is used to write safety-critical software, the program texts should be subjected to static analysis with a toolset such as MALPAS or SPADE. This process is made easy by the design of VISTA, which includes only constructs which can be analysed effectively by such toolsets. In fact the VISTA language was designed in parallel with a translator to the MALPAS input language [RTP], and many difficulties were resolved by changes to the language structure. A translator for SPADE [PVL] has also been developed.

This note describes VISTA in two ways: informally by means of an annotated example and formally with a modified BNF grammar. The example uses most of the constructs of the language, though one or two have escaped e.g. INCLUDE, CONTINUE. The grammar defines all possible constructs but does not in itself describe the semantics (“meaning”) of a VISTA program; where this is not obvious it is described in the accompanying commentary.

The following program in VISTA is fairly trivial and makes no pretensions to safety: it reads in a text and then counts occurrences of selected words in that text. The line numbers are merely for reference in the notes below, and are not part of the program. Page layout of VISTA programs is completely free: the example follows normal conventions for indentation. Two pieces of hardware called keyboard and screen are assumed to be available for input and output.

The only aspects which will not seem reasonably familiar are the “region declarations” (lines 4 to 7) which define regions of memory into which code, constants, and so on are to be placed, and the fact that procedures in VISTA are declared after (rather than before) their calls. There is no profound reason for this: it simply seems more natural to put the main program first. No “loopholes” are allowed for recursion, which is forbidden in VISTA though a devious programmer with knowledge of the machine architecture can sneak it in and thereby probably lose the ability to analyse the program. Notice that A, X, Y are the VIPER machine registers, 32 bits each, and there is a single bit flag register called B. X and Y (only) can be used as indexes.
2 VISTA program example

1 PROGRAM word frequency counter.

4 CODE prog FROM 0 TO 3999; -- memory regions with bounds
5 CONST const FROM 4000 TO 4095;
6 DATA data FROM 16x8000 TO 16x8000+8191;
7 PERI peri FROM 0 TO 1;

8 BEGIN

9 INT last, count, match, k, p, q; -- variables in DATA region
10 CHAR keyboard, screen; -- in PERI region
11 INT key[20], text[9000]; -- more data, vectors
12 INT sp = 32, eof = 26, cr = 13; -- in the CONST region
13 INT ms1[] = "\nInput the text: \0"; -- messages
14 INT ms2[] = "\nKeyvord: \0";
15 INT ms3[] = "\nNumber of occurrences is \0";
16 INT ms4[] = "outside range\0";
17 -- End of variable and constant declarations, now for the program
18 A := 1; CALL message; -- invite the user to type ...
19 A := 1; last := A; -- ... in a sample of text
20 A := sp; text[0] := A;
21
22 WHILE (CALL getchar; CALL upper) A /= eof
23 DO I := last; -- in case getchar changes I
24 CASE A > 'A' AND A < 'Z': SKIP; -- letters
25 A > '0' AND A < '9': SKIP; -- digits
26 ELSE A := sp; -- all others
27 ELSEAS; text[I] := A; I := I + 1; -- store the character
28 last := I
29 OD;
30
31 X := last; A := sp; text[I] := A; -- terminator for last word
32 REPEAT A := 2; CALL message; DISPLAY get new key word
33 A := 0; p := A;
34
43 WHILE (CALL getchar; CALL upper) A /= cr
44 DO X := p;
45  key[X] := A; -- read & store key word
46  X := X + 1;
47  p := X
48 OD;
49
50 X := p; A := sp; key[X] := A; -- terminator for key word
51 IF X = 0 THEN BREAK FI; -- exit if null key word
52 A := 0; p := A; count := A;
53
54 WHILE (Y := 0; match := Y; X := p) X < last
55 DO WHILE (A := key[Y]) A = text[X]
56 DO IF A = sp
57 THEN A := 1;
58 match := A; -- end of word & no mismatch
59 BREAK
60 FI;
61 X := X + 1;
62 Y := Y + 1 -- more letters, keep trying
63 OD;
64 IF (A := match; X := p; Y := text[X-1])
65 A /= 0 AND Y = sp
66 THEN A := count; -- whole of key matched
67 A := A + 1; -- check space before word ...
68 count := A -- .. don't want to find ..
69 FI;
70 X := X + 1; p := X
71 OD;
72
73 A := 3; CALL message; -- print number of matches
74 A := count; CALL print
75
76 UNTIL (A := key[0]) A = sp; -- until null key word input
77 STOP;
78
79 -- Now for the procedures, notice they follow the main program
80
81 PROC getchar: (WHILE (A := INPUT keyboard) A = 0 DO SKIP OD);
82
83 PROC upper: (IF A >= 'a' AND A <= 'z' THEN A := A-('a'-'A') FI);
PROC print:
-- unsigned decimal output
BEGIN INT Ixflag;
X := 0;
-- suppress leading zeros
Ixflag := X;
IF A < 0 OR A > 9999 THEN A := 4; CALL message
ELSE X := 1000; CALL digit;
X := 10; CALL digit;
X := 1; CALL digit
FI;
PROC digit:
-- prints 1 digit of number in A
BEGIN INT power;
-- I must hold a power of 10, I /= 0
power := X;
Y := 0;
WHILE A >= power DO A := A - power; Y := Y + 1 OD;
IF X = 1 OR Y > 0 THEN Ixflag := X FI;
Y := Y + '0';
-- convert to ASCII
IF Y = 0 THEN Y := sp FI;
OUTPUT Y, screen
END;
PROC message:
-- outputs message selected ..
BEGIN INT select;
select := A;
WHILE TRUE DO CASE (A := select) A IN
1: (A := ms1[X]);
2: (A := ms2[X]);
3: (A := ms3[X]);
4: (A := ms4[X])
ELSE A := 0
ESAC;
T := A AID 255;
-- now output the characters ..
IF Y = 0 THEN RETURN FI;
OUTPUT Y, screen;
CALL right8;
IF Y = 0 THEN RETURN FI;
OUTPUT Y, screen;
CALL right8;

132 IF Y = 0 THEN RETURN FI;
133 OUTPUT Y, screen;
134 CALL right8;
135 IF Y = 0 THEN RETURN FI;
136 OUTPUT Y, screen;
137 X := X + 1
138 DD;
139
140 PROC right8: -- 8 place right shift
141 BEGIN
142 A := A/2; A := A/2; A := A/2; A := A/2;
143 A := A/2; A := A/2; A := A/2; A := A/2;
144 Y := A AND 255
145 END
146
147 END -- of "message"
148 END -- of program
149 FINISH

3 Notes on the example (by line number)

1

All programs begin with PROGRAM, followed by an identifier which is taken as a title. The rest of the title line is ignored as comment. Identifiers in VISTA begin with a lower case letter and are made up of lower case letters and/or digits. They can be of any length though the translator will remember only the first 12 characters. The whole of a VISTA program must be processed at once - there is no "separate compilation" facility.

4..7

Region declarations. All storage in a VISTA program is allocated in a defined region, in ascending order of address. Constants are kept separate from data to allow use of ROM. The translator will report an error if a region fills up, or if region declarations overlap. The PER1 region has a separate address space from the other three. Notice the use of decimal and hex numbers (the other bases allowed are 2r for binary and 8r for octal) and of constant expressions which can be arbitrarily complex.

All the region declarations must be at the start, before the first block opens.

9

BEGIN marks the opening of the main block. All the data and procedure declarations of
the program must be within this or within procedures enclosed by it. Blocks may not be nested, though procedures can be: compound statements enclosed by `BEGIN .. END` or round brackets (which are everywhere interchangeable) are allowed but they may not contain declarations. Local redeclarations of variables within procedures take precedence over less local declarations with the same names, in the usual way. Local redeclarations of scalar constants are required to have the same value as any less local declarations which may still be in scope.

Ordinary variable declarations in the `DATA` region. The only data types allowed are `INT` and `BITS`, and vectors thereof. Both types occupy one 32 bit word per element. There is no restriction on the use of either. All storage allocation in VISTA is static. The initial values of variables are undefined; in a typical VIPER system they will be the contents of a RAM chip which has just been switched on.

Peripheral addresses in the `PERI` region, usable only with `INPUT` or `OUTPUT` (see lines 81, 107).

Vectors of variables. The lower bound is fixed at 0. Only one dimension is allowed.

Constants, to be placed in the `CONST` region. These are just ASCII character values. Wherever VISTA expects a constant (e.g. here or as a vector size, line 14) a constant expression can be written using any of the usual arithmetic and logical operators. Once declared, a named constant (e.g. "eof") can be used in constant expressions. Constants declared as of type `BITS` are treated as unsigned in constant expressions, as are numbers denoted in hex, octal, or binary. Decimal numbers and named constants of type `INT` are treated as signed. The choice of operator (signed vs. unsigned) in a constant expression is based on the type of the left operand: the distinction matters in practice only for the operators `>>`, `/`, `%` (see below).

The operators permitted in constant expressions, in descending order of priority are:

- unary `-` and `NOT` applied to primaries i.e. single names or numbers, or bracketed expressions.
- `<<` `>>` left/right shift the left operand by the number of places given by the right operand. Signed right shift fills with copies of the sign bit, unsigned with zeroes.
* bitwise logical AND, OR.
* multiply, divide, modulo, defined as on the host machine.
* add, subtract.

Constants (scalar or vector) cannot be assigned to. Their values are fixed irrevocably by the declaration, as they would be in reality if stored in ROM.

17..20

Vectors of constants. These can be given values either as a list of constant expressions e.g. (1, 2, 3, 4) or (as here) a string of 7 bit ASCII characters. Characters are stored four per word, least significant first. Notice the special characters \n (line feed) and \0\ (character of value 0, used here as a string terminator). Characters can also be used as individual constants: the declaration on line 16 could equally well be written as

```plaintext
INT sp = ' ', eof = '\26\', cr = '\c';
```

Character constants generate the standard ASCII representation without parity. See Section 5 ("initlist") for a list of special characters.

22

Comments follow -- as in Ada, and extend to the end of the line.

24 et seq.

Instructions are translated one-for-one to VIPER machine instructions. There are three 32 bit registers called A, X, Y of which X and Y can be used as index registers. Y is changed by the CALL instruction, but otherwise the registers can be used interchangeably.

No explicit arguments are allowed with procedure calls, but A and X can be used freely to hand over parameters.

28

Notice the "preface block" (CALL getchar; CALL upper) which is optional before a condition to set up the register contents. Conditions generally compare a register with a variable or constant (not two registers or two variables) but there are also conditions B, NOT B, TRUE, FALSE for special cases.
This form of the VISTA CASE statement requires an explicit predicate on each limb. A single preface block is allowed immediately after the word CASE. Conditions are tested in the order written and never cause a change in the register values; when a condition is found to be true the corresponding limb is entered and followed by exit from the CASE statement. See comment on line 117 for the other form of CASE statement.

Notice the use of AND as a connective in conditionals. Any number of AND or OR connectives may be used, but not both in the same condition. No brackets are allowed. Without these restrictions (which arise because of the need to map VISTA statements into single VIPER instructions) it would have been possible to use a simple IF - THEN - ELSE here.

The SKIP statement has no effect (and generates no code) but is provided for clarity. It can be used or omitted as preferred.

The ELSE limb is optional. If it were omitted and no condition was true, a CASE statement would cause the machine to stop. If you want a CASE statement to do nothing in this situation, use ELSE SKIP.

Loop constructs are WHILE - DO - OD and REPEAT - UNTIL.

DISPLAY is an alternative form of comment which is preserved in the translator output file. It does not form part of the program but can be picked up by other software e.g. the VIPER simulator.

BREAK and CONTINUE are valid only in a WHILE or REPEAT statement. Their effect is, respectively:

- Exit from the most local loop, i.e. jump to the statement following OD or following the condition after UNTIL.
- Proceed to the next cycle of the most local loop, i.e. jump to the beginning of the condition after WHILE or UNTIL.

The jump in this case is to line 77.

This is a fairly elaborate condition! The whole thing compiles to 6 instructions.
The STOP statement generates an illegal VIPER instruction which stops the processor. A constant expression can be placed after the STOP, in which case its value will be compiled into the (otherwise unused) address part of the instruction and will be visible on the address bus when the processor has stopped. If no expression is given, the translator will insert the address of the instruction.

Procedure declarations in VISTA follow their calls, i.e. only forward reference is allowed. This is logically more satisfactory (the main program comes first) and matches MALPAS better than the traditional rule of declaration before use. Recursion is not in principle possible, since the language does not allow pointers of any kind.

Variables and constants (lines 10..20) must be declared before use in the conventional way. In this and most other respects VISTA conforms to the normal Algol or Pascal block structure. Round brackets are interchangeable with BEGIN .. END.

The peripheral location keyboard is assumed to deliver 0 until a character is typed.

upper turns lower case letters in A to upper case.

4 digit unsigned decimal output.

digit is nested within print.

Division by repeated subtraction, since VIPER has (at present) no divide instruction. The loop is never obeyed more than 9 times.

The peripheral location screen is assumed to make characters visible in some way.
WHILE TRUE gives an endless loop which can be left only by BREAK or (in a procedure) RETURN.

The strings output by message must be known to the procedure in advance. It is not possible to hand over the address of an arbitrary string since VISTA does not allow pointers. This is an example of the other form of CASE statement in which each limb has either a single selector value or a bounded range of values e.g. 1..9: If the contents of the nominated register (A in this case) match the selector, the limb is entered. Selector values and bounds may be constant expressions, as usual. A minor variation replaces IN by UNSIGNED, telling the translator to use unsigned tests when checking whether the register is within a range.

This form of CASE is much easier for programs like MALPAS to analyse, e.g. when checking for coverage and overlap of the conditions.

A zero character terminates the string. Strings are stored with the least-significant 8 bits of the word holding the first character. RETURN exits from the most local procedure only.

right8 is nested inside message.

These are “arithmetic” shifts which duplicate the sign bit. Line 144 renders this unimportant. The alternative is an end-around shift through B (e.g. A >> 1) which in this case would do just as well.

Every program ends nominally with FINISH though end-of-file alone is acceptable.

4 Constructs not included in the example

* B := TRUE;  B := FALSE
  Direct setting/clearing of the B flag.
Source text inclusion facility. The first non-printing character e.g. space or end-of-line, terminates the file name. There is no default extension. The example given is of an MSDOS file name; the syntax is that appropriate to the host machine on which the VISTA translator is running. INCLUDE files may be nested up to 9 deep. A list of the files used will be printed at the top of the diagnostic output file.

5 Commentary on the grammar of VISTA

A modified form of the BNF grammar which follows is input to the S1D parser generator, whose output forms the kernel of the translator. This version of the grammar has been shorn of the "compiling actions" which form the interface to the rest of the translator, but do not change the structure of the language except in the handling of character strings where the effect of an absent action has been embodied into the syntax for clarity.

Terminal symbols e.g. NUM, SEMI, BEGIN are in capitals, non-terminals in lower case. Comments are between \ . . . . Punctuation characters (for instance * , ; ) are part of the meta-language of this version of BNF: commas separate alternatives and the last alternative of a rule ends with a semicolon. Empty alternatives are indicated by void.

The commentary is indexed by the names of the non-terminals, i.e. those which appear to the left of an = sign.

prog

The leading rule. Every program begins with PROGRAM and a name; any text following on the same line is taken as comment.

regions

A "region" declaration defines a bounded region of store or peripheral addressing space into which objects of appropriate type (CODE, CONST, DATA or PER) can be placed by the compiler. Only objects in DATA regions can be assigned to; only objects in PERI regions can be used in INPUT or OUTPUT instructions. Regions are global to the whole program, and the space they occupy is allocated permanently. The address space for PERI regions is separate from the other three. Region bounds are inclusive.

Several region declarations of the same type may be made e.g.

```
CODE rom1 FROM 0 TO 4095;
CODE rom2 FROM 65536 TO 65536 + 4095;
```

Only one region of each type (CODE, CONST, DATA, PERI) is in use at any one time, but it can be changed between declarations (for CODE, between procedures) by writing e.g.
CODE IN rom1; and reset to the region previously in use by e.g. RESET CODE; Changes are stacked separately for each type e.g.

```
CODE IN rom1; DATA IN ram; PROC fun1: (etc);  
CODE IN rom2; PROC fun2: (etc);  
RESET CODE;  
RESET CODE; -- restore status quo  
RESET DATA;
```

A region declaration automatically sets the new region as the current one of that type. While a region is in use, objects of appropriate type are placed at successively increasing addresses within that region until it is full, when an error message is given. While a region is not in use (because the programmer has changed to another of the same type) its current loading position is remembered.

All the region declarations in a program must be grouped together at the top, before the main block begins.

block

VISTA is block structured, where a block is either the main program or the body of a procedure. Blocks cannot be "nested" in VISTA, except as the bodies of procedures: BEGIN . . END or round brackets can be used to group instructions (e.g. for a CASE limb where the syntax demands a single instruction) but such a "compound statement" cannot include declarations. The present VISTA translator does not re-use memory allocated to local variables which cannot be in scope simultaneously, but this is not guaranteed to persist in future versions.

The scope rules are as follows:

- A "region" declaration is in scope for the whole program.
- An "object" (variable or constant) declaration is in scope from the point of declaration until the end of the main block or of the enclosing procedure, whichever is smaller.
- A "procedure" declaration is in scope from the start of the main block or of the enclosing procedure, whichever is smaller, until the start of its own declaration.

The sequence within a block is strictly defined, and is the same whether the block is the main program or a procedure body.

1. Variable and/or constant declarations (if any).
2. Main code, which can be void e.g. in a program consisting entirely of procedures.
3. Procedure declarations which can be interspersed with more variable or constant declarations (e.g. those used only to communicate between procedures) or with region changes.
BEGIN . . END and ( ) are interchangeable but must be correctly paired.

procdec

No explicit arguments or return values are allowed. What you do with the registers is up to you, but remember that the CALL instruction changes Y. RETURN changes no register except the program counter which is not visible in VISTA.

The form PROC ID FROM unsexp declares that a procedure can be found at the absolute address given by the unsigned expression, and is used to call facilities in other (separately compiled) programs e.g. in a ROM-resident monitor. The monitor program would contain a CODE region declaration starting FROM the same address.

objdec

An “object” declaration allocates space in one or other of the current regions. Types BITS and INT allocate space in the DATA region, or in the CONST region if the object is initialised. Type CHA allocates space in the PERI region.

Wherever a number would be appropriate in VISTA, a constant expression can be written. Objects declared as scalar constants e.g.

INT maxrom = 4095;

can be used in constant expressions - their values are held in the compiler as well as (possibly) in the program. Scalar constants which will fit into 20 bits are usually embedded in literal instructions and not stored separately. Vectors of constants (or elements thereof) e.g.

BITS table[ ] = (1, 2, 4, 8, 16, 32, 64, 128);

cannot be used in constant expressions. They are indexed from 0 up.

Vectors of variables e.g. INT vec[100]; must be declared with the size (in 32 bit words) and are also indexed from 0 up, in this case to 99.

initlist

A vector of constants can be initialised to a string, which will be stored 4 ASCII bytes per word without parity, least significant byte first. There is no automatic terminator, though unfilled words will be padded with nulls. Certain special characters can be included in strings using “backslash” and a lower case letter e.g. \n, \c, \f, \s, \t, \" for respectively linefeed, carriage return, form feed, space, tab, quote (which would otherwise terminate the string) and backslash. An arbitrary 8 bit constant can be placed in a string
(e.g. as a terminator) thus: "A null-terminated string\0". Any of the usual number bases may be used.

Strings must normally fit within one line. To extend a string on to the following line(s), end each line with the sequence \e thus:

```
INT msg[] = "An example of a string which will not fit \e comfortably within a single line";
```

\e does not insert a new line into the string: an explicit \n (or \c\n ) must be used.

### exp

Expressions are either all constant or consist of a variable only. Variables appear only in instructions. Unary minus or NOT are allowed only before a primary e.g. -(const-1) is allowed but -const-1 is not. See the commentary on the example program for a list of permitted operators. Any identifiers used in an expression must be already declared and in scope, see above under "block".

### inst

The syntax of VIPER instructions - most of the rest of VISTA is not VIPER-specific. Void is allowed as an instruction, but SKIP is provided for clarity e.g. as a null branch in CASE statements. It generates no code.

### regassign, indexreg

Data registers (sreg or dreg) are A, I, T. Index registers are I, Y only.

### loopst

BREAK, CONTINUE are valid only in WHILE or REPEAT statements, and transfer control to the instruction following the innermost loop or the start of the controlling test of the innermost loop respectively.

### casest

CASE statements come in two forms, one with an explicit predicate on each limb (caseitem) and one with a nominated register and a selection value or range on each limb (setitem). Either form can have an optional ELSE limb at the end, but if this is not present the translator will insert ELSE STOP. In both forms, the cases are tested in the order written. There is no check for coverage or overlap between predicates.
condlist

Conditions all correspond to single VIPER instructions, which compare a register with an immediate or stored operand. To set up the registers ready for the comparison, a “preface” can be inserted thus:

\[ \text{IF (A := fred) A > 0 THEN .... FI;} \]
\[ \text{WHILE (X := index; A := table[X]) A /= 0 DO .... OD;} \]

The preface (if present) is part of the test - see CONTINUE above.

There is no particular advantage in doing IF statements this way: compare

\[ A := \text{fred}; \text{IF A > 0 ...} \]

but the same is not true for WHILE and REPEAT statements: consider

\[ A := \text{fred}; \text{WHILE A > 0 DO .... ; A := 1; joe := A OD;} \]

condition

Compound conditions can be written using AND or OR but not both, because of the restriction that each comparison is a single VIPER instruction. Every comparison is performed every time. The only possible side effect is a hardware error on address-out-of-bounds e.g. in

\[ X := 5000000; \text{IF A = 0 AND Y = table[X] THEN .... FI;} \]

IF cond1 OR cond2 OR ... THEN is compiled as (see VIPER definition below)

\[ \text{TEST cond1; SET B IF cond2; SET B IF ...; IF NOT B GOTO else-or-fi then-code} \]

IF cond1 AND cond2 AND ... THEN is compiled as

\[ \text{TEST inverse of cond1; SET B IF inverse of cond2; SET B IF ...; IF B GOTO else-or-fi then-code} \]

“Condition” includes TRUE and FALSE for situations like WHILE TRUE DO .... OD and B, NOT B for special occasions.
relop

GE, LT are unsigned 32 bit comparisons.

operand, unsop

Things on the right-hand side of a regassign i.e. constant expressions, variables, vector references. If a constant expression can be fitted into 20 bits it will be handled as a literal, otherwise storage will be allocated for it in the current CONST region.

offset

A constant offset done at compile time, with an optional run-time index in the X or Y register.

variable

Writes to memory e.g. fred := A, also arguments of INPUT or OUTPUT.

function

These are machine functions, not to be confused with operators in constant expressions even though * - can be used in either context. They appear only after a register name (A, X or Y).

shift

/2 >>1 *2 <<<1 are the only permitted combinations in instructions, corresponding to the VIPER machine functions. The right operand is unrestricted in constant expressions.
6 VISTA Grammar

---

**Basic language structure**

**RULES**

**prog** = PROGRAM ID regions block opsemi FINISH;

**regions** = rdec SEMI, rdec SEMI regions; \ All region decs are global \ .. and there must be >= 1 \\

**rdec** = AREA bounds, \ data region declaration \\

CODE bounds, \ AREA = CONST, DATA, or PERI \\
datachange, \ CODE region declaration \\

codechange;

**bounds** = ID FROM unsexp TO unsexp ;

**block** = BEGIN decpart codepart END, \ a void decpart is allowed \ open decpart codepart CLOSE; \ note CODE change not allowed \\

**decpart** = void, \ notice this can also be void \\
ddec SEMI decpart;

**codepart** = inst, \ interspersed data decs .. \\

inst SEMI procode; \ .. or region changes .. \\

**procode** = codepart, \ .. but only between PROCs \\

prosect proclist;

**procetc** = procdec, \ interspersed data decs .. \\
ddec, \ .. or region changes .. \\
codechange;
procdec = PROC ID COLON block, PROC ID FROM unexp;

proclist = void, SEMI, \ have mercy on spare semis! \ SEMI proctc proclist;

******** Data and channel declarations ********

ddec = objdec, datachange;

objdec = TYPE obj objlist; \ data or CHAN dec \ TYPE = BITS, INT, or CHAN \ datachange = AREA IN ID, \ change data region \ RESET AREA; \ back to old region \ codechange = CODE IN ID, \ change CODE region \ RESET CODE;

obj = ID, \ not with CHAN! \ ID EQUALS exp, ID OSQ unexp CSQ, ID OSQ CSQ EQUALS initlist;

objlist = void, CONMA obj objlist;

initlist = OPEN exp erlist CLOSE, \ character string, see above \ QUOTE <string> QUOTE;

erlist = void, CONMA exp erlist;
Constant expressions, including scalar identifiers declared as CONST
A scalar variable as an operand is included for syntactic reasons

\[
\begin{align*}
\text{exp} & = \text{unexp}, \\
& \quad \text{PLUS primary,} \quad \text{\( \text{+ID or NOT ID still } \ldots \)} \\
& \quad \text{NOT primary,} \quad \text{\( \text{qualifies as a variable } \ldots \)} \\
& \quad \text{MINUS primary;} \quad \text{\( \text{while -ID does not} \)} \\
\text{unexp} & = \text{term}, \quad \text{\( \text{operators } + - \)} \\
& \quad \text{unexp PLUS term,} \\
& \quad \text{unexp MINUS term;} \\
\text{term} & = \text{factor}, \quad \text{\( \text{operators } \ast / \% \)} \\
& \quad \text{term TIMES factor,} \\
& \quad \text{term SLASH factor,} \\
& \quad \text{term MOD factor;} \\
\text{factor} & = \text{shifter}, \quad \text{\( \text{operators } \& / \)} \\
& \quad \text{factor AMPERSAND shifter,} \\
& \quad \text{factor B\& shifter;} \\
\text{shifter} & = \text{primary}, \quad \text{\( \text{operators } >> << \)} \\
& \quad \text{shifter GREATER GREATER primary,} \\
& \quad \text{shifter LESS LESS primary;} \\
\text{primary} & = \text{ID,} \quad \text{\( \text{over-size number} \)} \\
& \quad \text{NUM,} \\
& \quad \text{BIGNUM,} \\
& \quad \text{OPEN unexp CLOSE;} \\
\end{align*}
\]
\ Executable statements. This includes nearly all the VIPER-specific bits \\

\ inst = void,    \ to allow spare semicolons \\
       simstat, \ those allowed in prefaces \\
       SKIP, \ explicit null instruction \\
       RETURN, \ explicit null instruction \\
       STOP, \ those allowed in prefaces \\
       STOP exp, \ qualifier for diagnostics \\
       loopst, \ legal only in WHILE/REPEAT \\
       BEGIN codelist END, \ ditto \\
       OPEN codelist END; \ legal only in WHILE/REPEAT \\

\ simstat = regassign, \ compound statements \\
       variable COLOR EQUALS sreg, \ compound statements \\
       CALL ID, \ compound statements \\
       OUTPUT sreg COMMA variable, \ compound statements \\
       IF condlist THEN codelist elsepart FI, \ compound statements \\
       casest; \ compound statements \\

\ regassign = dreg COLOR EQUALS sreg, \ compound statements \\
       dreg COLOR EQUALS NOT sreg, \ compound statements \\
       dreg COLOR EQUALS operand, \ compound statements \\
       dreg COLOR EQUALS sreg function unsop, \ compound statements \\
       dreg COLOR EQUALS sreg shift NUM, \ compound statements \\
       dreg COLOR EQUALS INPUT variable, \ compound statements \\
       B COLOR EQUALS TRUE, \ compound statements \\
       B COLOR EQUALS FALSE; \ compound statements \\

\ codelist = inst, \ inst SEMI codelist; \ inst SEMI codelist; \\
       inst SEMI codelist; \ inst SEMI codelist; \\

\ elsepart = void, \ inst SEMI codelist; \ inst SEMI codelist; \\
       ELSE codelist; \ inst SEMI codelist; \\

\ loopst = WHILE condlist DO codelist OD, \ legal only in WHILE/REPEAT \\
       REPEAT codelist UNTIL condlist, \ ditto \\
       BREAK, \ ditto \\
       CONTINUE; \ ditto \\

21
```plaintext
CASE statements

\begin{verbatim}
casest = CASE caseitem caselist ESAC,
        CASE preface caseitem caselist ESAC,
        CASE areg signtype setitem setlist ESAC,
        CASE preface areg signtype setitem setlist ESAC;

preface = BEGIN simplist END, \ only assignments & calls .. \ 
          OPEN simplist CLOSE; \ .. allowed in prefaces \ 

simplist = simstat, \ simplist \ Simplist;
           simstat SEMI simplist;

\end{verbatim}

\begin{verbatim}
caseitem = condition COLON inst;

\end{verbatim}

\begin{verbatim}
caselist = void, \ no ELSE present \ 
           SEMI, \ ditto, spare semi \ 
           SEMI caseitem caselist, \ ditto, spare semi \ 
           opsemi ELSE codelist; \ default limb \ 

\end{verbatim}

\begin{verbatim}
signtype = IN, \ default type SIGNED \ 
           UNSIGNED;

\end{verbatim}

\begin{verbatim}
setitem = range COLON inst;

\end{verbatim}

\begin{verbatim}
setlist = void, \ 
          SEMI, \ 
          SEMI setitem setlist, \ 
          opsemi ELSE codelist;

\end{verbatim}

\begin{verbatim}
rnge = exp, \ exp DOT DOT exp;

\end{verbatim}

22
```
Conditions

\begin{verbatim}
condlist = condition,
           preface condition; \ "preface" statements \ 

condition = B, 
             NOT B, 
             TRUE, 
             FALSE, 
             cond, 
             cond AND andlist, 
             cond OR orlist;

andlist = cond, 
          cond AND andlist; \ can't mix AND, OR \ 

orlist = cond, 
        cond OR orlist;

cond = sreg relop exp, 
       sreg relop ID indxtail;

relop = LESS, 
        GREATER EQUALS, 
        EQUALS, 
        SLASH EQUALS, 
        LESS EQUALS, 
        GREATER, 
        LT, 
        GE;

opsemi = void, 
         SEMI; \ forgive spare semicolons! \ 

sreg = DATAREG; \ one of A, I, Y \ 

dreg = DATAREG;
\end{verbatim}
\texttt{Operands and Functions \end{grammar}}

\texttt{operand = exp, \hspace{1cm} \textit{constant or scalar variable}}
\hspace{1cm} \texttt{ID indxtail, \hspace{1cm} \textit{vector reference}}
\hspace{1cm} \texttt{NOT ID indxtail;}

\texttt{unsop = unsexp, \hspace{1cm} \textit{ID indxtail;}}

\texttt{indxtail = CSQ offset CSQ;}

\texttt{offset = unsexp, \hspace{1cm} \textit{non-negative constant offset}}
\hspace{1cm} \texttt{indexreg, \hspace{1cm} \textit{run-time index}}
\hspace{1cm} \texttt{indexreg PLUS unsexp, \hspace{1cm} \textit{ditto with constant offset}}
\hspace{1cm} \texttt{indexreg MINUS unsexp;}

\texttt{variable = ID, \hspace{1cm} \textit{subset of "operand"}}
\hspace{1cm} \texttt{ID indxtail;}

\texttt{indexreg = INDREG; \hspace{1cm} \textit{one of } X, Y \textit{ \}}

\texttt{function = PLUS, \hspace{1cm} \textit{VIPER machine functions, not to be confused ..}}
\hspace{1cm} \texttt{MINUS, \hspace{1cm} \textit{.. with operators in constant expressions}}
\hspace{1cm} \texttt{ADD, \hspace{1cm} XOR, \hspace{1cm} OR, \hspace{1cm} AND \hspace{1cm} NOT;}
\hspace{1cm} \texttt{SUB, \hspace{1cm} NOR, \hspace{1cm} AND \hspace{1cm} NOT;}
\hspace{1cm} \texttt{XOR, \hspace{1cm} NOR, \hspace{1cm} AND \hspace{1cm} NOT;}
\hspace{1cm} \texttt{GREATER GREATER, \hspace{1cm} LESS LESS;}

\texttt{End of grammar \end{grammar} \end{grammar}}
7 Using the VISTA translator

This description relates to VISTA Version 5 running on an IBM PC-compatible under MSDOS 3.3 or later.

The translator will print a summary of the operating instructions if invoked without arguments. Normally the name of the VISTA source file follows the program name; if no extension is given .VIS will be assumed e.g. vista sourcefile

VISTA can generate two output files, one containing diagnostics and information about the memory layout of the object program and the other containing the object program itself. The files normally have the same name as the source, and extensions (respectively) .ERR and .VOP but the names of the output files can be changed by an option.

Options are typed after the source file name, separated by a space. Every option begins with a letter (upper or lower case immaterial) and may include also a decimal number or a file name. Spaces may not appear within numbers or names, and a space (or end-of-line) must appear after a file name, but otherwise layout is unrestricted e.g.

```
vista sourcefile D5000 v p400b404 j20000 vopfile
```

All options have default values which can be found by running the translator and inspecting the statistics printed at the end. The options available are as follows:

- **T** Do a trial translation only i.e. no .VOP file
- **V** Non-verbose output i.e. no comments in .VOP file. Verbose .VOP files are typically 3 times as big.
- **F 20** Allow for 20 source files i.e. up to 19 INCLUDE files.
- **D 12000** Allow for 12000 data declarations in the program (variables and constants). Similarly **P** (procedure declarations) and **R** (region declarations).
- **J 6000** Allow for 6000 jumps - 3 each per IF, WHILE, REPEAT, CASE in the program.
- **C 200** Allow for 200 different anonymous constants i.e. constants appearing in instructions but not declared explicitly with names.
- **B 1000** Allow 1000 blocks. Blocks = procedures + 2
- **O name** Use "name.vop" and "name.err" for the two output files. Any extension typed will be ignored. "O" is a letter.
8 VIPER Machine Definition

The VIPER machine has 3 general purpose 32 bit registers (called A, I, Y), a program address counter (P), and a single bit Boolean register (B). Memory addresses occupy 20 bits, so only the least significant 20 bits of P are meaningful: loading a "1" into any of the top 12 bits will cause the machine to stop.

All instructions occupy 32 bits, divided into a 12 bit function code and a 20 bit address.

<table>
<thead>
<tr>
<th>Function 12 bits</th>
<th>Address 20 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>__________________</td>
<td>________________</td>
</tr>
</tbody>
</table>

The function code is further divided into:

<table>
<thead>
<tr>
<th>RF</th>
<th>MF</th>
<th>DF</th>
<th>CF</th>
<th>FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bits</td>
<td>2 bits</td>
<td>3 bits</td>
<td>1 bit</td>
<td>4 bits</td>
</tr>
<tr>
<td>__________________</td>
<td>________________</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Most instructions are of the form D := R op N, where D and R are registers chosen from A I T P. N is either a 20 bit literal constant or the contents of a 32 bit memory location. Memory addresses are limited to 20 bits, and the machine will stop if a computed address (MF = 2 or 3, see below) exceeds this limit.

**RF: source register field**

- 0: R is contents of register A
- 1: R " " " " I
- 2: R " " " " Y
- 3: R " " " " P after incrementing

**MF: memory address control field**

- 0: N is the address, i.e. 20 bit non-negative constant
- 1: N is contents of address, memory or peripheral
- 2: N is contents of (address+I) " " "
- 3: N is contents of (address+Y) " " "

**DF: destination control field**

- 0: D is the A register
- 1: D " " I "
- 2: D " " Y "
- 3: D " " P "
- 4: D " " P " if B, else do nothing
- 5: D " " P " if NOT B, else do nothing
- 6: D is N in peripheral space ) see
- 7: D is N in memory space ) below

26
The sequence in which the CF, DF, and FF fields are inspected is important:

| V |
|-------------------------|-------------------------|-------------------------|
| yes | The instruction is a comparison. The DF field is ignored. |
| no |

| V |
|-------------------------|-------------------------|-------------------------|
| yes | The instruction is "store R": M will be used as the destination, stopping if MF = 0. The FF field is ignored. |
| no |

| V |
|-------------------------|-------------------------|-------------------------|
| yes | If B is false do nothing, otherwise ALU operation to P. |
| no |

| V |
|-------------------------|-------------------------|-------------------------|
| yes | If B is true do nothing, otherwise ALU operation to P. |
| no |

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU operation to register specified by DF (in the range 0 to 3).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8.1 ALU operations

If CF = 0 and DF < 6, FF and possibly MF specify the ALU function:

<table>
<thead>
<tr>
<th>FF</th>
<th>D := N OT N</th>
<th>i.e. N operand complemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y := P then P := M from memory space i.e. CALL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>D := M from peripheral space ) equivalent</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>D := M from memory space ) if M = 0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>D := R + M, B := carry</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>D := R + M, stop on overflow</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>D := R - M, B := borrow</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>D := R - M, stop on overflow</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>D := R XOR M</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>D := R AND M</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>D := R XOR M</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>D := R AND (NOT M)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>D := R AND (NOT M)</td>
<td></td>
</tr>
</tbody>
</table>
12, \( MF = 0 \)  
D := R/2, sign bit copied

MF = 1
D := R >> 1 through B, i.e. D31 := B,  
DO..30 := R1..31, B := RO

MF = 2
D := R \times 2, stop on overflow

MF = 3
D := R << 1 through B, i.e. D0 := B,  
D1..31 := RO..30, R := R31

13 spare instruction, stop

14 spare instruction, stop

15 spare instruction, stop

Note that if the destination is P (DF = 3, 4, or 5) only functions 1, 3, 5, and 7 are legal. Attempting to obey any other will cause the machine to stop. Function 1 operates only on P; if DF specifies any other destination register the machine will stop.

The operation \( D := R \) can be achieved by performing \( D := R \times 0 \) or (for destinations A, X, Y) \( D := R \& \& \& NOT 0 \) which is faster. \( D := NOT R \) can be done as \( D := R \& \& \& R0 \).

8.2 Comparisons

If \( CF = 1 \), \( FF \) specifies a comparison. Comparisons never change anything other than \( B \), apart from the change in \( P \) implied by continuing to the next instruction, and never cause the machine to stop unless a memory address exceeds 20 bits (\( MF = 2 \) or 3). For comparisons the ALU function is forced to \( R - M \), and the arithmetic unit behaves as if it had 33 bits with initial sign extension and no overflow detection, allowing signed or unsigned comparison of 32 bit values.

The new value of \( B \) is derived from the result of the subtraction as follows: bit32 is the conceptual extra (33rd) bit:

\[
\begin{align*}
FF = 0 & \quad B := \text{bit32} \quad (\text{i.e. } R < M) \\
1 & \quad B := \text{NOT bit32} \quad (R > M) \\
2 & \quad B := \text{allzero} \quad (R = M) \\
3 & \quad B := \text{NOT allzero} \quad (R /\!/ M) \\
4 & \quad B := \text{bit32 OR allzero} \quad (R <= M) \\
5 & \quad B := \text{NOT bit32 AND NOT allzero} \quad (R > M) \\
6 & \quad B := \text{borrow} \quad (\text{unsigned } R < M) \\
7 & \quad B := \text{NOT borrow} \quad (\text{unsigned } R > M)
\end{align*}
\]

\( FF = 8 \) to \( 15 \): the set of operations is repeated, but if the result is 0 (i.e. FALSE) \( B \) is left unchanged.
VIPER Object Program (.VOP) Format

VIPER Object Program files are printable, and contain the information needed by a program loader, PROM programmer, or VIPER simulator. Only the information within the defined format is significant; the remainder of each line is undefined and can be filled with whatever material may be useful. The VISTA translator outputs the current loading address and a “reconstructed” version of each instruction, followed on some lines by a “marker” showing the beginning of loops, procedures etc.

The type of every line is indicated by the character in column 1, which is followed (in most cases) either by a string of characters, a hex number, or a sequence of decimal numbers. Some types of line have alternative decimal or hex forms. The defined formats must be adhered to rigidly.

In the following description, underscore indicates mandatory space characters. d stands for a decimal digit (0..9) and h for a hex digit (0..9, A..F or a..f). All other characters stand for themselves. Every defined format must end with a “white space” character i.e. space, TAB, or end-of-line.

Title and Date of program. These are not mandatory but are generated by VISTA from (respectively) the PROGRAM line which begins the source file and the date derived from the host operating system.

Set the current loading address i.e the address at which subsequent data will be loaded into memory. In the decimal form the second 5 digit number represents the least significant 16 bits of the address (0..65535) and the first number represents the rest. In both forms the digits are in the usual order i.e. most significant first. Addresses up to 16rFFFFF are interpreted as in memory. Addresses in the range 16r100000 to 16r1FFFFF are treated by the VIPER simulator as in peripheral space.

Load the 32 bit value given into the location pointed to by the current loading address, add 1 to the address, and update the sumcheck (see below). In the decimal form the four numbers (0..255) represent the four bytes of the value, most significant first. All the digits must be present.
Verify the current sumcheck against the hex value given, and clear the sumcheck. The sumcheck is the 32 bit residue of the sum of every 32 bit value loaded plus its address, since the sumcheck was last cleared. The sumcheck is zero initially. All the digits must be present.

End of .VOP file.

The remaining formats are relevant only to a simulator or a prototyping system which contains a host computer of some kind. PROM programmers ignore them.

Format as for C. Load the VIPER A register with the 32 bit value given. Similar formats exist for the X, Y, and P (program address) registers.

Set/clear the B flag or the internal STOP flag.

Define a memory or I/O region size, for memory allocation in the simulator. The first decimal number is the lowest address in the region, the second is the highest. In a VISTA output these correspond to the regions declared by the programmer, with the upper bounds as actually used rather than as declared. The bounds must be decimal, 0..1048575, but in these two formats leading zeroes may be omitted.

The character string is the name of a file "used" by an INCLUDE statement. Simulators can use this information to improve diagnostics.

Corresponds to DISPLAY in the VISTA source. The character string may be interpreted by a simulator program or simply displayed as a diagnostic.
References


[RTP] MALPAS is supplied by Rex, Thompson and Partners Ltd, West Street, Farnham GU9 7EQ, UK.

[PVL] SPADE is supplied by Program Validation Ltd, 26 Queen’s Terrace, Southampton SO1 1BQ, UK.
VISTA is a structured assembly language for the VIPER microprocessor chip. Though the syntax of VISTA and the appearance of VISTA programs are reminiscent of a high-level language, the actual statements are VIPER machine instructions. VISTA provides the clarity and much of the convenience of a true high-level language without the need for a complex and possibly untrustworthy compiling program.

This Report describes the VISTA language by means of an annotated example, and gives instructions for using the VISTA translator. A formal grammar of VISTA, and descriptions of VIPER and the VIPER Object Program format, are included.