1097

Bell Laboratories **Cover Sheet for Technical Memorandum**

The information contained herein is for the use of employees of Bell Laboratories and is not for publication. (See GEI 13.9-3)

Title- **SIM - a language for simulating computers**

Date- **May 5, 1976**

TM- **76-1271-2**

Other Keywords-

Author	Location	Extension	Charging Case- 39199
L. L. Cherry	**MH 2C-516**	**6067**	Filing Case- **39199-11**

ABSTRACT

SIM is a language and compiler for writing computer and microprocessor simulators. It combines a major portion of the facilities available in the C language[1,2] with the special bit-picking operations that are necessary to describe machines. The operations include bit referencing, concatenation and bit equivalencing.

Machine description is typically at the register level of architecture. The user defines machine registers and operations in SIM. The operations are defined as functions that may be called by the user or by a system-supplied control program.

SIM allows the user to concentrate on the machine to be simulated rather than the mechanics of writing a simulator. It also provides a readable description of the operation of the machine being simulated.

This paper is intended to be a concise reference manual and assumes familiarity with C.

SIM is written in YACC[3] and C. It runs on the UNIX operating system and will soon be availiable under GCOS.

Pages Text 10	Other 6	Total 16	
No. Figures 0	No. Tables 0	No. Refs. 5	

Bell Laboratories

Subject: SIM - a language for simulating computers
Case- 39199 -- File- 39199-11

date: May 5, 1976

from: L. L. Cherry

TM: 76-1271-2

MEMORANDUM FOR FILE

1. Introduction

SIM is a language and compiler for writing computer and microprocessor simulators. It combines a major portion of the facilities available in the C language[1,2] with some of the special bit-picking operations that are required to describe machines. The operations include bit referencing, concatenation, and bit equivalencing.

The SIM source language is used to define the registers and operations of the machine to be simulated. The machine cycles are defined as functions. When a SIM source program is compiled, a C program is produced as output. This C program is the source of a simulator for the machine that was described in the SIM language. The C program can be compiled together with user-supplied programs for control, initialization and testing and the resulting object program is an executable simulator. If no control program for the simulator is supplied by the user, the system will supply one. This system-supplied control program calls the functions that define the machine cycles in a user determined order described in section 5. The initialization program must load memory with the program to be simulated and initialize registers with starting values.

SIM currently runs under UNIX and will soon be available under GCOS.

In the future SIM will have an interactive controller that will allow the user to control the simulation directly from the terminal.

Because SIM is a general language, it knows nothing about machines or their architecture. If timing as well as functional simulation is desired, it must be built into the machine description or control program.

The SIM language allows the programmer to concentrate on the machine to be simulated rather than the mechanics of writing a simulator. It also provides a readable description of the machine operations. The system-supplied control program allows the user to bring up a functional simulator in a short period of time and later, with few changes to the machine description, add a more complicated control program to handle interrupts and timing.

The general format of a SIM program is:

```
%{
C preprocessor statements and external variables, etc.
%}
definitions
functions
%%
C user routines
```

where the initial C source and final user subroutines may be omitted. If there are no user subroutines the %% delimiter may also be omitted.

2. Notation and Conventions

Throughout this manual syntactic categories are in *italics*, literals in **bold**. Things in brackets [] are optional.

Parsing follows C with a few notable exceptions. A label must be the first token on a line; it may be preceded only by blanks or tabs. Blanks are used as in C, except that angle brackets cannot be surrounded by blanks. **decode** is a keyword in addition to the keywords of C.

Identifiers are strings of upper and lower case letters, numbers and _ (underscore), beginning with an alphabetic. Identifiers that do not appear in the definitions section of the program are assumed to be external C identifiers defined by the user in the initial C source code.

Constants are strings of digits. A preceding 0 indicates octal.

As in C, comments are bracketed by /* and */.

3. Initial C Source Code

Anything between %{ and %} at the beginning of the source file is simply copied out. This section of code generally contains external C variables and C preprocessor statements defining mnemonics for constant op-code and field values. Such constants may be used where numbers are expected in the SIM syntax.

4. Definitions

The definitions section is used to define variables or to name parts of variables.

4.1. Simple Definitions

Simple definitions have the form:

%*name*<*n1*:*n2*>

where *n1* is the number specifying the leftmost bit and *n2* is a number specifying the rightmost bit. For example, the definition of an instruction register, ir, that is 16 bits long with bit 0 on the left is:

%ir<0:15>

To define bit 0 as the rightmost bit, as in the PDP11, the definition would be:

%ir<15:0>

All definitions must be consistent as to which end of the register is bit 0. Variables can be no more than 2 host machine words long (32 bits on UNIX, 72 bits on GCOS).

4.2. Arrays

Arrays of variables are defined by:

%*name* [*size*]<*n1*:*n2*>

where *size* is the number of elements in the array and *n1* and *n2* are as described above. Arrays are indexed from 0; the last array element is *size*−1 .

4.3. Bit Equivalencing

A variable may be equivalenced to a part of a previously defined variable as follows:

%name = name1<n1:n2>

The definition of an op-code field, op0, as bits 15 through 8 of the instruction register, ir, is:

%op0 = ir<15:8>

The bits of op0 are numbered 7 through 0; 0 is the rightmost bit. Another op-code field may be defined to be the bottom 3 bits of op0 by:

%op1 = op0<2:0>

or by

%op1 = ir<10:8>

<n1:n2> may be omitted to equivalence *name* to all of *name1.*

4.4. Bit Equivalencing In Arrays

A variable may be defined as a part of a previously defined array element by:

%name = name1 [n]<n1:n2>

and again *<n1:n2>* may be omitted to specify the whole array element. The definition of the program counter, pc, as register 7 on the PDP11 is:

%pc = r[7]

4.5. Array Equivalencing

Arrays may be equivalenced by

%name [n] = name1 [m]

where *n* and *m* specify the alignment of the two arrays. This is useful for machines whose general purpose registers are actually the initial part of memory.

5. Functions

The functions section of a SIM program contains the machine definition, specified by at least one function definition. A function is designated by:

%n name

where *n* is an integer less than 100. *n* specifies the order in which the functions are to be called if no user control program is provided. All numbers between 1 and the largest *n* used must be represented. Functions may be defined in any order. If *n* is 0, the function is an internal function never to be called directly by the system-supplied control program. If the function has no body, it is assumed to be a user defined function included after the %%. For example, the following code describes a machine which performs two functions, fetch and execute; each function is to be followed by a user-supplied printing function, dump:

```
%1 fetch
...
%2 dump
%3 execute
...
%4 dump
%%
```

When using the default control program, the user must supply one other routine, in addition to the functions and their order. This is an initialization routine

siminit(argc,argv)

which must load a program into memory, initialize the program counter etc. The default control program first calls **siminit**, then goes into a loop calling the functions in the order specified.

A function definition includes all the code between its name and the next *%n name* or the final %%. The body of a function is written in the language described in sections 6 and 7.

6. Expressions

The SIM operators described below work only on SIM variables, i.e. those variables defined in the definitions section of the program. External C variables may be operated on by all the C operators with the single exception of the ?: operator which is not implemented.

6.1. Simple Expressions

Simple expressions may appear on both the left and right side of an equal sign. In C terminology they may be lvalues as well as rvalues.

6.1.1. *identifier*

An identifier is a simple expression, provided it has been defined in the definitions section of the program. An identifier not so defined is assumed to be an external C identifier defined in the C source code at the beginning of the program.

6.1.2. *identifier* [*expression*]

An identifier followed by a bracketed expression is a simple expression. The expression is taken as a subscript and the identifier must have been defined as an array in the definitions section of the program. Identifiers defined as arrays should not be used without a subscript.

6.1.3. *identifier*<*expression*:*expression*>

An identifier followed by two expressions separated by a colon and enclosed in angle brackets is a simple expression. The expressions in angle brackets specify the beginning and ending bits of the field in the identifier to be addressed. When this simple expression is used as an lvalue the bits named are replaced; when it is used as an rvalue these bits are right justified.

6.1.4. *identifier* [*expression*]<*expression*:*expression*>

A subscripted identifier followed by two expressions separated by a colon and enclosed in angle brackets is a simple expression. The expressions in angle brackets specify the beginning and ending bits of the array element to be addressed. When this simple expression is used as an lvalue the named bits are replaced; when it is used as an rvalue the bits are right justified.

6.2. Other Expressions

Expressions in this section may not appear on the left of an assignment operator.

6.2.1. *constant*

A constant is an expression.

6.2.2. (*expression*)

An expression in parenthesis is an expression.

6.2.3. *function-name*([*expression*[,*expression*]])

A function call is an expression.

6.3. Unary Operators

Unary operators have the highest precedence and group right to left.

6.3.1. −*expression*

The result is the negation of the expression.

6.3.2. ~*expression*

The result is the one's complement of the expression.

6.3.3. !*expression*

The result of the logical negation operator is 0 if the value of the expression is non-zero and 1 if the value of the expression is zero.

6.4. Concatenation

simple expression :: *simple expression*

The result is the concatenation of the simple expressions. If the combined length of the two expressions is longer than 2 host machine words, it is truncated from the left.

6.5. Binary Operators

Binary operators group left to right. There are four groups of operators with equal precedence. The precedence of the groups is the same as the order in which they are presented below with the lowest first.

6.5.1. Additive Operators

6.5.1.1. addition

expression + *expression*

6.5.1.2. subtraction

expression − *expression*

6.5.2. Shift Operators

6.5.2.1. left shift

expression << *expression*

6.5.2.2. right shift

expression >> *expression*

6.5.3. Logical Operators

6.5.3.1. and

expression & *expression*

6.5.3.2. inclusive or

expression | *expression*

6.5.3.3. exclusive or

expression ^ *expression*
expression ~= *expression*

6.5.3.4. nand

expression ~& *expression*

6.5.3.5. nor

expression ~| *expression*

6.5.4. Multiplicative Operators

6.5.4.1. multiplication

expression * *expression*

6.5.4.2. division

expression / *expression*

6.5.4.3. remainder

expression % *expression*

The result of the above binary operators is the same as in C but with no sign extension. Variables behave like the low order bits of a long padded on the left with 0's. If the result of these expressions is being stored in an identifier that is not long enough the result will be truncated on the left.

6.6. Relational Operators

6.6.1. *expression* < *expression*

6.6.2. *expression* > *expression*

6.6.3. *expression* <= *expression*

6.6.4. *expression* >= *expression*

6.6.5. *expression* == *expression*

6.6.6. *expression* != *expression*

6.6.7. *expression* && *expression*

6.6.8. *expression* | *expression*

The relational operators operate as in C. The operators < and > must be surrounded by blanks. Because SIM makes great effort to override any sign-extension features of the host machine, care should be taken when using the operators < and > to test for negativity.

7. Statements

7.1. Assignment Statements

simple expression <- *expression*;
simple expression = *expression*;
simple expression =*op expression*;

The value of the expression replaces the previous value of the simple expression. If the value of the expression is too long to fit in the simple expression, it is truncated on the left. The <- and = operators are the same. *op* in the third kind of assignment may be any operator in 6.5. The result of the third kind of assignment is the same as in C. Except in pathological cases

simple expression =*op expression*;

means the same as

simple expression = *simple expression op expression*;

7.2. Compound Statement

{ *statement* }

Several statements enclosed in curly brackets may be used wherever a single statement is expected.

7.3. Conditional Statement

(expression) => *statement*

If the expression is non-zero the statement is executed. This is the same as the C **if** statement without the **else** clause. A multiple condition switch statement(**decode**) described below can be used if an **if-else** is desired.

7.4. While Statement

while(*expression*) *statement*

As long as the expression is non-zero the substatement is repeatedly executed.

7.5. Decode Statement

decode(*expression*[,*expression*]) {*condition-list*}

The decode statement is a limited version of Steve Johnson's decision table generator, DTGEN[4] with a slightly different format. The condition-list has the form

$$c_1, c_2, \ldots, c_k : statements$$

where the conditions c_k may be integers, integers preceded by !, or closed intervals indicated by [*n1*,*n2*]. The statements are executed if the values of the expressions match the condition list. More than one set of conditions may be matched for a given set of expressions. Conditions without associated statements will execute the next set of statements found. Missing conditions at the end of the condition list are treated as 'don't care' conditions. The statement list executed for a set of conditions is terminated by the next set of conditions or the } that terminates the **decode** statement. The conditions must be the first token on the line, preceded only by blanks or tabs. **decode** statements may be nested.

7.6. Labeled Statement

label : *statement*

Labeled statements in SIM are the same as labeled statements in C except that the label must be the first token on the line.

7.7. Goto Statement

goto *label*;

7.8. Return Statement

return;
return(*expression*);

SIM functions return long integer values.

7.9. Break Statement

break;

Break causes termination of the smallest enclosing **while** or **decode** statement.

7.10. Continue statement

continue;

Continue passes control to the test portion of the smallest enclosing **while** statement, i.e. to the end of the loop.

8. User Interface

As discussed in section 5, a default control program is provided if the SIM file does not include one. To allow for user written control programs and to provide communication between the user routines **siminit** and **simtest** and the SIM variables, two additional routines are included. The values of SIM variables may be interrogated by the user through the routine

long simget(*name,index***)**

where *name* is a ascii string containing the name of the variable and *index* is an integer used as a subscript if the variable was defined as an array. **simget** returns a long integer. The values of SIM variables may be set by the user through the routine

simput(*value,name,index***)**

where *value* is a long integer and *name* and *index* are as described above. *value* is truncated from the left if necessary.

9. An example

The example in Appendix 1 is a part of a functional simulator for the MAC-8[5]. The simulator has 3 parts; the C pre-processor definitions of the op-codes, the register definitions, and the functions that define the machine operation. The MAC-8 is a byte-oriented general purpose microprocessor with both 8 and 16 bit operations. The 16-bit quantities are used for addressing and operations on them are performed separately on the 2 8-bit bytes. For simplicity, this simulator performs these operations on one 16-bit quantity. The MAC-8 has 16 general purpose registers that are a moveable 32 byte section of memory and are addressed through a 16-bit register pointer (**rp**). Two-byte quantities are stored least significant byte first. There are 3 other 16-bit registers; the stack pointer (**sp**), the program counter (**pc**), and a temporary addressing register (**t16**). An 8-bit conditions register (**cr**) holds the condition flags.

Instructions are fetched 2 bytes at a time; the first byte goes into the instruction register (**ir**), the second byte goes into the destination and source register (**ds**). For dyadic instructions the instruction register contains a 5-bit op-code field (**op**) and a 3-bit mode field (**mode**). Bit 0 of **op** (**twobytes**) indicates whether the operation is to be performed on an 8 or 16 bit quantity. Bit 0 of **mode** (**ind**) indicates indirection. Register **ds** is divided into 2 4-bit fields, a destination field (**d**) and a source field (**s**).

To simplify the simulator 3 16-bit temporary registers have been defined. Registers **sc** and **dst** hold the value of the source and destination respectively. Register **er** is used for intermediate address calculations. The 17-bit working register (**wr**) is used for 16-bit calculations when a carry bit must be captured.

The MAC-8 instructions are specified in the **execute** function. This function decodes the op field and executes the proper instruction. A non-SIM function for printing register values called **dump** is executed after each call to **execute**.

The supporting functions defined after **execute** are used for decoding the addressing mode and fetching the proper source and destination values, replacing the destination with a new value, and setting the condition codes. Most MAC-8 instructions call one or more of these functions.

The addressing modes for the MAC-8 are given in Appendix 2. Addressing register 15 is a special case and indicates either the **pc** or **sp** for some modes. Source addressing is done in function **getsc**. Function **getdst** does the destination addressing and leaves the address of the destination in register **t16** to be used by **putdst** in replacing the destination value. Function **setcond** sets the condition flags from **wr**. It is called by arithmetic instructions that set all the condition flags. Instructions that leave some of the condition flags unchanged (i.e., **MOVE**) must set the flags involved in the instruction.

The instructions simulated in **execute** were chosen to illustrate the various types of MAC-8 instructions. The **MOVE** and **ADD** instructions are dyadic with a 5-bit op-code field and a 3-bit mode field. **MOVE** sets the condition flags in the instruction rather than by calling **setcond** because **carry** and **ovfl** are unchanged by **MOVE**. **INCRDECR** and **COMPL** are monadic and have a 6-bit op-code field and a 2-bit mode field. The simulator looks at the op-code as a 5-bit quantity (05) followed by a 1-bit code to distinguish between the 2 instructions. The branch instructions simulated are branch immediate, branch relative, branch on register bit, and branch on memory bit. The relative branch can be forward or backward, determined by the sign bit of the offset in **ds**.

A complete functional simulator of the MAC-8 would contain code for all instructions in **execute**. The supporting functions would remain approximately the same.

10. Usage

On UNIX

sim *file*.**sim**

puts the SIM output in *file*.**d**. The command **simcc** *file* will compile *file*.**d** and leave an executable simulator in **a.out**.

11. Acknowledgements

Thanks are due to Harold Shichman for suggesting SIM, to Dennis Ritchie for his quick improvements to the C compiler at many stages of SIM's development, and to Jim Cooper for his help with the MAC-8.

MH-1271-LLC

L. L. Cherry

Attachments
References
Appendix 1 and 2

References

[1] D. M. Ritchie, *C Reference Manual*, TM74-1273-1.
[2] B. W. Kernighan, *Programming in C: A Tutorial*, TM 74-1273-12.
[3] S. C. Johnson, *YACC, Yet Another Compiler-Compiler*, TM75-1273-6.
[4] S. C. Johnson, *DTGEN*, personal communication.
[5] S. T. Campbell, *MAC-8 Microprocessor Summary*, Memorandum For File(March 8, 1976).

Appendix 1

```
%{      /* op-codes */
#define  ON  1
#define  OFF 0
#define  MOVE    020
#define  MOVE16  030
#define  ADD     025
#define  ADD16   035
#define  INCRDECR 0
#define  COMPL   1
#define  BRANCH 013
long signext();
%}
%mem[1024]<0:7>          /* memory */
%rp<0:15>                /* register pointer */
%sp<0:15>                /* stack pointer */
%pc<0:15>                /* program counter */
%cr<0:7>                 /* condition register */
%neg = cr<0:0>           /* flags */
%zero = cr<1:1>
%ovfl = cr<2:2>
%carry = cr<3:3>
%ones = cr<5:5>
%odd = cr<6:6>
%flag = cr<7:7>
%ir<0:7>                 /* instruction register */
%op = ir<0:4>
%mode = ir<5:7>
%mmode = ir<6:7>
%op1 = ir<5:5>
%ind = mode<0:0>
%twobytes = op<1:1>
%ds<0:7>                 /* source and destination byte */
%d = ds<0:3>
%s = ds<4:7>
%t16<0:15>               /* temporary addressing register */
%sc<0:15>                /* simulator temporary registers */
%dst<0:15>
%wr<0:16>
%er<0:15>
%tmode<0:2>
%1execute
      ir <- mem[pc];
      ds <- mem[pc+1];
      pc =+ 2;
      decode(op){
MOVE:
MOVE16:
      tmode <- mode;
      getsc(); getdst();
      wr <- sc;
      decode(twobytes){
      OFF: neg <- wr<9:9>;
           odd <- wr<16:16>;
           decode(wr<9:16>){
```

```
            0:    zero <- 1;
            !0:   zero <- 0;
            0377:ones <- 1;
            !0377:      ones <- 0;
            }
       ON:  neg <- wr<1:1>;
            zero <- 0;
            (wr<1:16> == 0) => zero <- 1;
       }
       putdst();
ADD:
ADD16:
       tmode <- mode;
       getsc(); getdst();
       wr <- sc + dst;
       setcond();
       putdst();
5:     tmode < mmode;
       decode(op1){
       INCRDECR: getdst();
            decode(s<0:0>){
            OFF: wr <- wr + s;
            ON: wr <- wr +(s|0177760);
            }
            setcond();
            putdst();
       COMPL:  getdst();
            wr <- ~dst;
            setcond();
            putdst();
       }
 BRANCH: decode(mmode){
       0:    pc <- mem[pc]::ds;              /* branch immediate */
       1:    pc <- pc + signext(ds);         /* branch relative */
       2:    er <- rp +(s<<1);               /* branch on register bit */
             sc <- mem[er+1] :: mem[er];
             wr <- 1 << (15 - d);
             decode(sc & wr){
             !0:   pc <- pc + signext(mem[pc]);
             0:    pc =+ 1;
             }
       3:    er <- rp + (s<<1);              /* branch on memory bit */
             sc <- mem[er+1]::mem[er];
             sc <- mem[sc];
             wr <- 1 << (15 - d);
             decode(sc & wr){
             !0:   pc <- pc + signext(mem[pc]);
             0:    pc =+ 1;
             }
       }
 }
 %2dump
 %0getsc          /* routine to put source value in sc
                  see Appendix 2 for meanings of codes */
       er <- rp + (s<<1);
       decode(s,mode,twobytes){
       [0,14],[0,3],OFF:     sc <- mem[er];
```

```
        [0,14],[0,3],ON:
        [0,14],5:       sc <- mem[er+1]::mem[er];
        [0,14],7:       sc <- mem[er+1]::mem[er];
                        decode(twobytes){
                        OFF: wr = sc + 1;
                        ON:  wr = sc + 2;
                        }
                        mem[er] <- wr<9:16>;
                        mem[er+1] <- wr<1:8>;
        15,[0,3],OFF:   sc <- mem[pc];
                        pc =+ 1;
        15,[0,3],ON:
        15,5:           sc <- mem[pc+1]::mem[pc];
                        pc =+ 2;
        [0,14],4:
        [0,14]),6:      sc <- mem[er+1]::mem[er] + signext(mem[pc];
                        pc =+ 1;
        15,4:
        15,6:           sc <- sp + signext(mem[pc]);
                        pc =+ 1;
        }
        (ind == ON) => decode(twobytes){
             OFF: sc <- mem[sc];
             ON:  sc <- mem[sc+1]::mem[sc];
             }
%0getdst        /* routine to put destination value in dst
                   leaving destination address in t16
                   see Appendix 2 for meanings of codes */
        t16 <- rp + (d << 1);
        decode(d,tmode,twobytes){
        [0,15],0,OFF:
        [0,15],[5,7],OFF:       dst <- mem[t16];
        [0,15],0,ON:
        [0,15],[5,7],ON:
        [0,14],1:
        [0,15],3:       dst <- mem[t16+1]::mem[t16];
        15,1:           dst <- mem[pc+1]::mem[pc];
                        pc =+ 2;
        [0,14],2:
        [0,14],4:       dst <- mem[t16+1]::mem[t16] + signext(mem[pc]);
                        pc =+ 1;
        15,2:
        15,4:           wr <- sp + signext(mem[pc]);
                        pc =+ 1;
        }
        (tmode >= 2 && tmode <= 4)=>{
             t16 <- dst;
             decode(twobytes){
             OFF: dst <- mem[t16];
             ON:  dst <- mem[t16+1]::mem[t16];
             }
        }
%0putdst        /* put wr in destination address */
        mem[t16] <- wr<9:16>;
        (twobytes == ON)=> mem[t16+1] <- wr<1:8>;
        (tmode == 3) => {
             er <- rp + (d<<1);
```

```
            decode(twobytes){
            OFF: t16 =+ 1;
            ON:  t16 =+ 2;
            }
            mem[er] <- t16<8:15>;
            mem[er+1] <- t16<0:7>;
      }
%0setcond      /* set condition codes from wr */
      ovfl <- 0;
      decode(twobytes){
      OFF: neg <- wr<9:9>;
            ((sc<8:8> == dst<8:8>) && (sc<8:8> != wr<9:9>)) => ovfl <- 1;
            carry <- wr<8:8>;
            decode(wr<9:16>){
            0:          zero <- 1;
            !0:         zero <- 0;
            0377:       ones <- 1;
            !0377:      ones <- 0;
            }
            odd <- wr<16:16>;
       ON:  neg <- wr<1:1>;
            decode(wr<1:16>){
            0:    zero <- 1;
            !0:   zero <- 0;
            }
            ((sc<0:0> == dst<0:0>) && (sc<0:0> != wr<1:1>)) => ovfl <- 1;
            carry <- wr<0:0>;
      }
%%
long signext(byte)
long byte; {
      if(byte & 0200)return(byte|0177400L);
      return(byte);
}
dump(){
      /* print registers */
}
```

Appendix 2

Addressing Modes

Mode	$s = [0,14]$	$s = 15$	$d = [0,14]$	$d = 15$
0	r_s	$*pc$	r_d	r_d
1	r_s	$*pc$	$*r_d$	$**pc$
2	r_s	$*pc$	$*(r_d+N)$	$*(sp+N)$
3	r_s	$*pc$	$*r_d{+}{+}$	$*r_d{+}{+}$
4	$*(r_s+N)$	$*(sp+N)$	$*(r_d+N_1)$	$*(sp+N_1)$
5	$*r_s$	$**pc$	r_d	r_d
6	$*(r_s+N)$	$*(sp+N)$	r_d	r_d
7	$*r_s{+}{+}$	$*r_s{+}{+}$	r_d	r_d

where an immediate r_i may be either 8 or 16 bits long; an indirect r_i is always 16 bits long. N is the next byte in the instruction stream; N_1 is the byte after N.