CMU-CS-78-137

The Symbolic Manipulation of Computer Descriptions:

# An Introduction to ISPS

Mario R. Barbacci

Department of Computer Science
Carnegie-Mellon University

16 August 1978

The development of ISPS is part of the research on the Symbolic Manipulation of Computer Descriptions effort at CMU and is sponsored by the Defense Advanced Research Projects Agency under Grant F44620-73-C-0074.

## Table of Contents

## 1 Introduction

This document introduces the reader to the ISPS[1] notation. Although some details have been excluded, it covers enough of the language to provide a "reading" capability. Thus while this document in itself might not be sufficient to allow writing ISPS descriptions, it should be detailed enough to permit the reading and study of complex descriptions.

Not all the features of the notation are presented in the examples. For a detailed explanation of the complete language the reader must consult the reference manual:

**The ISPS Computer Description Language**

Mario R. Barbacci
Gary E. Barnes
Roderic C. Cattell
Daniel P. Siewiorek

Departments of Computer Science
and Electrical Engineering
Carnegie-Mellon University
August 1977

There exists a compiler and a simulator for ISPS. These programs are written in BLISS-10 and run on a DEC PDP-10 Computer under either TOPS-10, TOPS-20, or TENEX. For information about software distribution contact:

Mario R. Barbacci
Department of Computer Science
Carnegie-Mellon University
Pittsburgh PA 15213
(412) 578-2578

or BARBACCI@CMUA on the ARPAnet.

## 2 Instruction Set Processor Descriptions

To describe the ISP of a computer, or any machine, we need to define the operations, instructions, data types, and interpretation rules used in the machine. These will be introduced gradually, as we describe the primary memory state, the processor state, and the interpretation cycle. Primary memory is not, in a strict sense, part of the Instruction Set

[1] ISPS is the second implementation of the ISP notation introduced in Bell C.G. and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill Book Company, New York, 1971

Processor but it plays such an important role in its operation that it is typically included in the description. In general, data types (integers, floating point numbers, characters, addresses etc.) are abstractions of the contents of the machine registers and memories. One data type that requires explicit treatment is the "instruction" and we shall explore the interpretation of instructions in great detail.

We will use the PDP-8 ISPS description as a source of examples. In the presentation of the PDP-8 registers and data types we will use the following conventions: 1) names in upper case correspond to physical components on the PDP-8 (e.g., program counter, interrupt lines, etc.), 2) names in lower case do not have a correspondent physical components (e.g., instruction mnemonics, instruction fields, etc).

## 2.1 Memory State

The description of the PDP-8 begins by specifying the primary memory that is used to store data and instructions:

```
M\Memory[0:4095]<0:11>,
```

The primary memory is declared as an array of 4096 words, each 12 bits wide. The memory has a name "M", and an alias "Memory". These "aliases" are a special form of a comment and are useful for indicating the meaning or usage of a register's name. As in most programming languages, ISPS identifiers consist of letters and digits, beginning with a letter. The character "." is also allowed, to increase the readability. The expression [0:4095] describes the structure of the array. It declares the size (4096 words) and the names of the words (0,1,..., 4094,4095).

The expression <0:11> describes the structure of each individual word. It declares the size (12 bits) and the names of the bits (0,1,...,10,11).

> It should be noted that bit and word "names" are precisely that, i.e., identifiers for the subcomponents of a memory structure. These "names" do not necessarily indicate the relative position of the subcomponents. Thus, R<7:3> is a valid definition of a 5-bit register. The fact that the five bits are "named" 7,6,5,4,3 should not be confused with the 7th, 6th, etc. positions inside the register. Thus, bit 7 is the leftmost bit, bit 6 is located in the next position towards its right, etc., while bit 3 is the rightmost bit.

Memory is divided into 128-word pages. Page zero is used for holding global variables and can be accessed directly by each instruction. Locations 8 through 15 of page zero have

the special property, called auto-indexing, that when accessed indirectly, the contents of the location is incremented by 1. These regions of memory can be described as part of M as follows:

```
P.0\Page.Zero[0:127]<0:11>       := M[0:127]<0:11>,
        A.I\Auto.Index[0:7]<0:11>        := P.0[8:15]<0:11>,
```

The word (and bit) naming conventions on the left hand side of a field declaration are independent from the word (bit) names used on the right hand side. A.I[0] corresponds to P.0[8], A.I[1] corresponds to P.0[9], etc.

## 2.2 Processor State

The processor state is defined by a collection of registers used to store data, instructions, condition codes, etc. during the instruction interpretation cycle.

The PDP-8 has a 1-bit register L, which contains the overflow or carry generated by the arithmetic operations, and a 12-bit register AC, which contains the result of the arithmetic and logic operations. The concatenation of L and AC constitutes an extended accumulator LAC. The structure of the extended accumulator is shown below:

```
LAC<0:12>,
        L\Link<>                    := LAC<0>,
        AC\Accumulator<0:11>        := LAC<1:12>,
```

The expression <> indicates a single, unnamed bit (L is only one bit long and there is no need to specify a name for it.)

The program counter is used to store the address of the current instruction being executed as the machine steps through a program:

```
PC\Program.Counter<0:11>,
```

Twelve bits are needed in the PC to address all 4096 locations of MP.

In the PDP-8, I/O devices are allowed to "interrupt" the central processor. When a device requires service from the central processor, it emulates a subroutine call, forcing the processor to execute an appropriate I/O subroutine. The presence of an interrupt request is indicated by setting the INTERRUPT.REQUEST flag. The processor can honor these requests or not, depending on the setting of the INTERRUPT.ENABLE bit:

```
INTERRUPT.ENABLE<>,
INTERRUPT.REQUEST<>,
```

There are 12 console switches which can be read by the processor. These switches are treated as a 12-bit register by the central processor:

```
SWITCHES<0:11>,
```

### 2.3 Instruction Format

As most data types and registers on the PDP-8, instructions are 12-bits long:

```
i\instruction<0:11>,
```

An instruction is a special kind of data type. It is really an aggregate of smaller information units (Operation Codes, Address Modes, Operand Addresses, etc.). The structure of the instructions must be exposed by describing the format. Most PDP-8 instructions contain an operation code and an operand address:

```
op\operation.code<0:2>   := i<0:2>,
ib\indirect.bit<>        := i<3>,
pb\page.0.bit<>          := i<4>,
pa\page.address<0:6>     := i<5:11>,
```

op, ib, pb, and pa are abstractions that allow us to treat selected fields of the PDP-8 instructions as individual entities.

### 2.4 Partitioning the Description

In ISPS, a description can be divided into sections of the form:

```
** section.name **
<declaration>,
<declaration>,
. . . . .

** section.name **
<declaration>,
<declaration>,
. . . . . .
```

Each section begins with a header, an identifier enclosed between ** and **. A section

consists of a list of declarations separated by commas. Section names are not reserved keywords in the language, they are used to convey to the users of the description some information about the entities declared inside the section. The register and memory declarations presented so far could be grouped into the following sections:

```
☆☆ Memory.State ☆☆

M\Memory[0:4095]<0:11>,
        P.0\Page.Zero[0:127]<0:11>       := M[0:127]<0:11>,
                A.I\Auto.Index[0:7]<0:11>        := P.0[8:15]<0:11>,

☆☆ Processor.State ☆☆

LAC<0:12>,
        L\Link<>                    := LAC<0>,
        AC\Accumulator<0:11>        := LAC<1:12>,
PC\Program.Counter<0:11>,
RUN<>,
INTERRUPT.ENABLE<>,
INTERRUPT.REQUEST<>,
SWITCHES<0:11>,
```

```
*** Instruction.Format ***

i\instruction<0:11>,

        op\operation.code<0:2>  := i<0:2>,
        ib\indirect.bit<>       := i<3>,
        pb\page.0.bit<>         := i<4>,
        pa\page.address<0:6>    := i<5:11>,

        IO.SELECT<0:5>          := i<3:8>,      ! device select
        io.control<0:2>         := i<9:11>,     ! device operation
                IO.PULSE.P1<>           := io.control<0>,
                IO.PULSE.P2<>           := io.control<1>,
                IO.PULSE.P4<>           := io.control<2>,

        sma<>   := i<5>,                ! skip on minus AC
        spa<>   := i<5>,                ! skip on positive AC
        sza<>   := i<6>,                ! skip on zero AC
        sna<>   := i<6>,                ! skip on AC not zero
        snl<>   := i<7>,                ! skip on L not zero
        szl<>   := i<7>,                ! skip on L zero
        is<>    := i<8>,                ! invert skip sense
        group<> := i<3>,                ! microinstruction group
        cla<>   := i<4>,                ! clear AC
        cll<>   := i<5>,                ! clear L
        cma<>   := i<6>,                ! complement AC
        cml<>   := i<7>,                ! complement L
        rar<>   := i<8>,                ! rotate right
        ral<>   := i<9>,                ! rotate left
        rt<>    := i<10>,               ! rotate twice
        iac<>   := i<11>,               ! increment AC
        osr<>   := i<9>,                ! logical or AC with SWITCHES
        hlt<>   := i<10>,               ! halt the processor
```

We have added a few more field declarations. These are used to interpret the I/O and Operate instructions. The PDP-8 I/O instruction uses the 9 bits of addressing information to specify operations for the I/O devices. These 9 bits are divided into a "device selector" field (6 bits, IO.SELECT<0:5>) and a "device operation" field (3 bits, io.control<0:2>). Note that several alternate field declarations may be associated with the same portion of a register or data type thus adding flexibility to the description. A comment is indicated by "!" and all characters following "!" to the end of the line are treated as commentary and not as part of the description. The PDP-8 Operate instruction's address field is not interpreted as an address but as a list of sub-operations. The reader can refer to the DEC PDP-8 processor manuals for additional details.

## 3 Effective Address

The effective address computation is an algorithm which computes "addresses" of data and instructions:

```
☆☆ Effective.Address ☆☆

last.pc<0:11>,

eadd\effective.address<0:11> :=
        Begin
        Decode pb =>
                Begin
                0 := eadd ='00000 @ pa,                    ! Page Zero
                1 := eadd = last.pc<0:4> @ pa              ! Current Page
                End Next
        If Not ib => Leave eadd Next
        If eadd<0:8> Eqv #001 => M[eadd] = M[eadd] + 1 Next ! Auto Index
        eadd = M[eadd]
        End,
```

Since the memory of the machine is 4096 words long, addresses have to be 12 bits long. Of the 12 bits in an instruction, 3 bits have been allocated for the operation code (op) and there are only 9 bits (ib, pb, and pa) in the instruction register left for addressing information. These bits, together with some other portions of the processor state, are interpreted by the algorithm to yield the necessary 12 bits of addressing needed.

### 3.1 Address Computation

Instructions and data tend to be accessed sequentially or within address clusters. This property is called "locality". The PDP-8 memory is logically divided into 32 pages of 128 words each. The concept of locality of memory references is used to reduce the addressing information by assuming that data are usually in the same page as the instructions that reference them. The pa portion of an instruction is that "address within the current page". The pb portion on an instruction is used as an escape mechanism to indicate when pa is to be used as an address within page 0 (M[0:127]) instead of the current page.

last.pc contains the address of the current instruction and is used to compute the current page number.

The first step of the algorithm,

```
Decode pb =>
        Begin
        0 := eadd ='00000 @ pa,
        1 := eadd = last.pc<0:4> @ pa
        End Next
```

indicates a group of alternative actions, to be selected according to the value of the expression following the "Decode" operator. The alternatives appear enclosed between "Begin" and "End" and separated by ",". The expressions "0 :=" and "1 :=" are used to label the statements with the corresponding value of pb. The alternative statements can be left unnumbered in which case they are treated as if they were labelled "0:=", "1:=", "2:=",... etc.

The effective address (eadd) is built by concatenating a page number with the page address (pa). The "@" operator is used to indicate concatenation of operands. If pb is equal to 0, page 0 is used in the computation. If pb is equal to 1, the current page number is used instead.

Constants prefixed with the character "'" represent binary numbers. '00000 represents a 5-bit string which is concatenated with the 7 bits of pa to yield the 12 bits needed.

> The transfer operator, "=", modifies the memory or register specified on its left hand side. If the right hand side has more bits than the left hand side, the right hand side is truncated to the proper size by dropping the leftmost extra bits. If the right hand side is shorter, enough 0 bits are added on its left until the length of the left hand side is matched. Thus, the first conditional statement can be written as "0 := eadd = pa".

The expression <0:4> is used to select bits 0,..,4 of last.pc. These 5 bits contain the current page number, and, together with the 7 bits of pa, yield the necessary 12 bits.

### 3.2 Indirect Addresses

A full 12 bit target address can be stored in a memory location used as a pointer and the instruction only needs to specify the address of this pointer location. Indirect addresses are specified via a bit in the instruction register (ib) which indicates whether we have a direct (ib=0) or an indirect (ib=1) address.

The second step of the algorithm,

```
If Not ib => Leave eadd
```

is separated from the previous by the operator "Next". The statement(s) preceding Next must be completed before the statement following it can be executed. The first step computed a preliminary effective address. The second step tests the value of ib and if it is equal to 0 then the preliminary effective address is used as the real effective address. If ib is equal to 1, the preliminary effective address is used to access a memory location which contains the real effective address. In the former case, the expression "Leave eadd" is used to indicate the termination of the procedure (this is similar to a RETURN statement in many programming languages).

### 3.3 Auto Indexing

Constants prefixed with the character "#" represent octal numbers. #001 represents the following 9-bit string: '000000001. The procedure treats indirect addresses as special cases. If a preliminary effective address in the range #0010:#0017 (8:15) is used as an indirect address (ib=1), the memory location is first incremented and the new value used as the indirect address:

```
If eadd<0:8> Eqv #001 => M[eadd] = M[eadd] + 1 Next
eadd = M[eadd]
```

By comparing the high order bits of eadd with #001 and ignoring the lower 3 bits we are in fact specifying a range of addresses (#0010, #0011, #0012,... #0017). Memory locations #0010:#0017 constitute the auto-indexing registers.

Regardless of whether auto-indexing took place or not, the last step of the algorithm uses the preliminary effective address (which could have been modified by auto-indexing) as the address of a memory location which contains the real effective address:

```
eadd=M[eadd]
```

## 4 Instruction Interpretation

The instruction interpretation section describes the instruction cycle i.e. the fetching, decoding, and executing of instructions.

```
** Instruction.Interpretation **

interpret :=
        Begin
        Repeat  Begin
                i = M[PC]; last.pc = PC Next
                PC = PC + 1 Next
                execute() Next
                If INTERRUPT.ENABLE And INTERRUPT.REQUEST =>
                        Begin
                        M[0] = PC Next
                        PC = 1
                        End
                End
        End,
```

The instruction cycle is described by a loop. The "Repeat" operator precedes a block of statements that are to be continuously executed. The instruction cycle of the machine consists of four steps:

1. A new instruction is fetched (i = M[PC]).

2. The program counter is incremented (PC = PC + 1). It now points to the next instruction. Under normal circumstances (i.e. unless a Jump takes place) this will be the instruction to be executed next.

3. The instruction is executed (execute()).

4. Interrupt requests, if allowed are honored. The cycle is then repeated.

The ";" separator is used to indicate concurrency (i.e. two statements separated by ";" are executed concurrently):

```
i = M[PC]; last.pc = PC Next
```

Notice how the value of the program counter is saved in last.pc before it is incremented. The effective address procedure relies on the fact that last.pc contains the address of the current instruction.

The execute procedure describes the individual instructions:

```
execute :=
        Begin
        Decode op =>
                Begin
                #0\and :=       AC = AC And M[eadd()],
                #1\tad :=       LAC = LAC + M[eadd()],
                #2\isz :=       Begin
                                M[eadd] = M[eadd()] + 1 Next
                                If M[eadd] Eql 0 => PC = PC + 1
                                End,
                #3\dca :=       Begin
                                M[eadd()] = AC Next
                                AC = 0
                                End,
                #4\jms :=       Begin
                                M[eadd()] = PC Next
                                PC = eadd + 1
                                End,
                #5\jmp :=       PC = eadd(),
                #6\iot :=       input.output(),
                #7\opr :=       operate()
                End
        End,
```

Instruction mnemonics can be indicated as aliases for the constants used to specify the operation codes:

```
#3\dca := .....
```

### 4.1 Operation Code 0\and: Logical And

If the operation code is equal to 0, the contents of the accumulator (excluding the L bit) are replaced by the logical product of the accumulator and a memory location. eadd() is used to indicate that the effective address computation must be executed in order to obtain the memory address.

### 4.2 Operation Code 1\tad: Two's Complement Add

The tad instruction follows the pattern of the previous instruction. Notice however, that the complete accumulator (including the L bit) is involved in the operation. L will contain the overflow or carry out of the sign position of AC.

### 4.3 Operation Code 2\isz: Increment and Skip if Zero

This instruction is described in two consecutive steps, The first step indicates that some memory location, specified by the effective address computation, will be incremented by 1.

Notice the different uses of eadd in the statement:

```
M[eadd] = M[eadd()] + 1
```

The effective address is computed once, eadd(), and is used to fetch the memory location, M[eadd()]. The result of the addition must be stored back in the same memory location. This is indicated by using the effective address register, eadd, on the left hand side, M[eadd]. eadd already contained the correct address and there was no need to recompute it. In fact, because of the auto-indexing operations performed during the effective address computation, the effective address must be computed precisely once.

The second step of the instruction,

```
If M[eadd] Eql 0 => PC = PC + 1
```

tests the result of the addition. If the result is equal to 0 the program counter is incremented by one, thus in effect, skipping over the next instruction in sequence. Once again, eadd is used instead of eadd() to avoid undesirable side-effects.

### 4.4 Operation Code 3\dca: Deposit and Clear Accumulator

This instruction deposits the accumulator in a memory location and then clears the accumulator (excluding the L bit).

### 4.5 Operation Code 4\jms: Jump to Subroutine

This instruction alters the normal sequence of instructions by modifying the program counter so that the next instruction will not be the one following the current instruction, but the one located at a memory location specified by the effective address. The program counter is stored into the location preceding the subroutine code (the result of eadd()). The program counter is then modified to point to the first instruction of the subroutine (eadd + 1).

### 4.6 Operation Code 5\jmp: Jump

This instruction also modifies the normal sequence of instructions. It can be used to jump

to disjoint pieces of code. If we use ib=1 and specify the address of the location preceding the subroutine, the result of the effective address computation will yield the return address that was stored by the subroutine call.

### 4.7 Operation Code 6\iot: Input/Output

The input.output procedure describes two specific cases of I/O instruction, namely those used to control the interrupt mechanism:

```
input.output :=
        Begin
        Decode i<3:11> =>
                Begin
                #001\ion :=
                        Begin                       ! turn Interrupt ON
                        INTERRUPT.ENABLE = 1 Next
                        Restart interpret
                        End,
                #002\iof :=
                        Begin                       ! turn Interrupt OFF
                        INTERRUPT.ENABLE = 0
                        End,
                Otherwise := No.Op()                ! not implemented
                End
        End,
```

"Otherwise" can be specified in a Decode operation to indicate a default action to be executed if none of the explicitly named cases (#001 or #002) apply. All other I/O operations default to a predefined ISPS procedure No.Op(), this is done simply to keep the examples short.

I/O operation #002 disables interrupts. It typically occurs as the first instruction of an interrupt handling routine. I/O operation #001 enables interrupts. It typically occurs at the end of an interrupt handling subroutine. Its effect is delayed for one instruction (the return from the subroutine) to avoid losing the return address if an interrupt were to occur immediately. This is achieved by skipping over the last portion of the instruction interpretation cycle:

```
If INTERRUPT.ENABLE And INTERRUPT.REQUEST => ....
```

The "Restart interpret" operation is used to indicate a return from the input.output procedure, not to the place from were it was invoked (inside execute) but to the beginning of the interpret procedure, thus bypassing the interrupt trapping for one instruction.

### 4.8 Operation Code 7\opr: Operate

The Operate instruction encodes a large number of primitive "micro-operations" in the address bits of an instruction. Some bits (e.g., cla) represent a micro-operation by themselves. Others (e.g., rt and ral) jointly represent a micro-operation. There are several conditional skip micro-operations. These are grouped in a separate procedure for readability:

```
skip<>,

skip.group :=
        Begin
        skip = 0 Next
        Decode is =>                                    ! invert skip condition
                Begin
                0 :=    Begin
                        If snl And (L Eql 1) => skip = 1;
                        If sza And (AC Eql 0) => skip = 1;
                        If sma And (AC Lss 0) => skip = 1
                        End,
                1 :=    Begin
                        IF szl@sna@spa Eql 0 => skip = 1;
                        If szl And (L Eql 0) => skip = 1;
                        If sna And (AC Neq 0) => skip = 1;
                        If spa And (AC Geq 0) => skip = 1
                        End
                End Next
        If skip => PC = PC + 1                          ! Skip
        End,
```

```
operate :=
        Begin
        Decode group =>
                Begin
                0 :=    Begin                           ! group 1
                        If cla => AC = 0;
                        If cll => L = 0 Next
                        If cma => AC = Not AC;
                        If cml => L = Not L Next
                        If iac => LAC = LAC + 1 Next
                        Decode rt =>            ! rotate once or twice
                                Begin
                                0 :=    Begin           ! once
                                        If ral => LAC = LAC Slr 1;
                                        If rar => LAC = LAC Srr 1
                                        End,
                                1 :=    Begin           ! twice
                                        If ral => LAC = LAC Slr 2;
                                        If rar => LAC = LAC Srr 2
                                        End
                                End
                        End,
                1 :=    Begin                           ! groups 2 and 3
                        Decode i<11> =>
                                Begin
                                0 :=    Begin           ! group 2
                                        skip.group() Next
                                        If cla => AC = 0 Next
                                        If osr => AC = AC Or SWITCHES;
                                        If hlt => RUN = 0
                                        End,
                                1 :=    Begin           ! group 3
                                        If cla => AC = 0 Next
                                        No.Op()  ! eae group
                                        End
                                End
                        End
                End
        End
```

Several micro-operations can appear in the same instruction, however, not all combinations are legal or useful. Micro-operations are executed at different points in time thus allowing sequences of transformations applied to the accumulator and/or link bit. For instance, in the group 1 micro-operations, clearing AC/L is done before complementing them, this is done before incrementing the combined L@AC (LAC) register, and this in turn precedes the rotation of L@AC.

# 5 Other Features of ISPS

Not all the features of the notation have been presented in the examples. This section will attempt to provide a list of the missing operations to help the readers follow larger descriptions.

## 5.1 Constants

In general a constant is a sequence of characters drawn from some alphabet determined by the base of the constant. The base of a non-decimal constant is given by a prefix character. The alphabets for the predefined bases in ISPS are:

| Base | Prefix | Alphabet |
|---|---|---|
| 2 | ' | 0,1,? |
| 8 | # | 0,1,2,3,4,5,6,7,? |
| 10 | | 0,1,2,3,4,5,6,7,8,9,? |
| 16 | " | 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,? |

The character "?" can be used to specify a don't care digits. Its presence stands for any digit in the corresponding alphabet.

The length of a constant is measured in bits. Decimal constants are one bit longer than the smallest number of bits needed to represent its value (beware that the use of don't care ("?") decimal digits results in constants of unspecified length). Binary constants have one bit for each digit explicitly written. Octal constants have three bits for each digit explicitly written. Hexadecimal constants have four bits for each digit explicitly written:

| Example | Length | Bit Pattern |
|---|---|---|
| "1000 | 16 | 0001000000000000 |
| 15 | 5 | 01111 |
| #17 | 6 | 001111 |
| 0 | 2 | 00 |
| '0?101 | 5 | 0?101 |
| #?2 | 6 | ???010 |

## 5.2 Arithmetic Representation

ISPS allows the user to specify arithmetic operations in four different representations: Two's Complement, One's Complement, Sign Magnitude, and Unsigned Magnitude (the default is Two's Complement.) To specify a different representation, the following modifiers can be used:

| Modifier | Arithmetic Representation |
|---|---|
| {TC} | Two's Complement |
| {OC} | One's Complement |
| {SM} | Sign Magnitude |
| {US} | Unsigned Magnitude |

In all the signed representations, the sign bit is the leftmost position of the operand (1 for negative numbers, 0 for positive numbers). The above modifiers can be attached to any arithmetic or relational operator to override a default. They can also be attached to a procedure declaration to set a default throughout the body. When attached to a section name the default applies to all the declarations in the section:

```
test :=
        Begin {OC}                  ! Default for the body
        .....
        End,

** Section.1 ** {TC}                ! Default for the section
        ..........

X = Y + {SM} Z                      ! Instance
```

Always remember that the arithmetic representation is a property of the operator, not the operand. Thus, the same bit pattern can be treated as a Two's complement or an Unsigned integer depending on the arithmetic context in which it is used.

### 5.3 Sign Extension

All ISPS data operators define results whose length is determined by both the lengths of the operands and the specific operator. Some operations require that their operands be of the same length. This is usually accomplished by "sign-extending" the operands. In the context of Unsigned Magnitude arithmetic, "sign-extension" is interpreted as zero-extension (i.e. padding with 0s on the left). In One's and Two's Complement arithmetic the expansion is

done by replication of the sign bit. In Sign Magnitude arithmetic the expansion is done by inserting 0s between the sign bit and the most significant bit of the operand.

### 5.4 Data Operators (in order of precedence)

#### 5.4.1 Negation and Complement: -, NOT

Unary - generates the arithmetic complement of the operand (the operation is invalid in Unsigned arithmetic.) The result is one bit longer than the operand. The NOT operator generates the logical complement of the operand. The result has the same length as the operand.

#### 5.4.2 Concatenation: @

The @ operator concatenates the two operands. The length of the result is the sum of the lengths of the operands.

#### 5.4.3 Shift and Rotate: SL0,SL1,SLD,SLR,SR0,SR1,SRD,SRR

These operators shift or rotate the left operand the number of places specified by the right operand. The result has the same length as the left operand. The operators have the format "Sxy" where "x" is either L(eft) or R(ight) to indicate the direction of movement. "y" is either 0, 1, D(uplicate), or R(otate) to indicate the source of bits to be shifted in. Sx1 shifts its left operand inserting 1s in the vacant positions. Sx0 is similar to Sx1 but inserting 0s. SxD inserts copies of the bit leaving the position to be vacated (not the bit being shifted out). SxR inserts copies of the bit being shifted out (i.e. rotates the left operand).

#### 5.4.4 Multiplication, Division, and Remainder: *, /, MOD

These operators compute the arithmetic product, quotient, and remainder of the two operands, respectively. The lengths of the results are:

| Operation | Length of Result |
|---|---|
| * | Sum of lengths |
| / | Left Operand (dividend) |
| MOD | Right Operand (divisor) |

#### 5.4.5 Addition and Subtraction: +, -

The + and -operators compute the arithmetic sum and difference of the two operands,

respectively. The shortest operand is sign-extended and the result is one bit longer than the largest operand.

### 5.4.6 Relational Operations: EQL,NEQ,LSS,LEQ,GTR,GEQ,TST

These operations perform an arithmetic comparison between the two operands. The shortest operand is sign-extended and the result is either 1 or 2 bits long. The first six operators (i.e. all except TST) produce a 1-bit result indicating whether the relation is True (1) or False (0). The TST operator produces a 2-bit result indicating whether the relation between the left and right operands is LSS (0), EQL (1), or GTR (2).

### 5.4.7 Conjunction and Equivalence: AND, EQV

These operators produce the logical product and coincidence operations of the two operands. The shortest operand is zero-extended and the result is as long as the largest operand.

### 5.4.8 Disjunction and Non-equivalence: OR, XOR

These operators produce the logical sum and difference operations of the two operands. The shortest operand is zero-extended and the result is as long as the largest operand.

### 5.4.9 Logical and Arithmetic Assignment: =, <=

The logical assignment operator, "=", truncates or zero-extends the source (right operand) to match the length of the destination (left operand). The arithmetic assignment operator, "<=", truncates or sign-extends the source to match the length of the destination.