

# **SANKHYA Machine Description Language**

**Reference Manual**

**THIS DOCUMENT CONTAINS PROPRIETARY INFORMATION OF SANKHYA TECHNOLOGIES PRIVATE LIMITED. Use, duplication and disclosure are subject to license restrictions.**

**(C) Copyright 2002-2003 Sankhya Technologies Private Limited**

**All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means viz., electronic, mechanical, photo-copying, recording, or otherwise, without the prior consent of the publisher.**

**SANKHYA, SANKHYA TECHNOLOGIES, SANKHYA Tools Collection, Dynamically Targetable Tools Framework, SANKHYA Software are Trademarks, Service Marks or Registered Trademarks of Sankhya Technologies Private Limited. All other brands and names are the property of their respective owners.**

**Draft version**

---

---

# SANKHYA Machine Description Language

## **Reference Manual**

**Sankhya Technologies Private Limited**

---

## Preface

SANKHYA Machine Description Language (SMDL) is a language used for modeling a processor for use by software development tools like code generator, assembler and simulator.

The document consists of two chapters :

**Chapter 1.1** *SMDL Specification* which explains the SMDL specification and provides detailed description of each of the construct used in SMDL.

**Chapter 1.2** *Processor description file creation* which explains the steps involved in describing a processor using SANKHYA Machine Description Language (SMDL).

### Audience

This document is intended for Processor Developers, Processor Architecture Designers, Compiler Tools Developers.

Basic knowledge in Regular Expression (RE), Context Free Grammar (CFG) terminology and notation, processor architectures in general and the particular processor to be modeled is required.

### Notational Conventions

The guide follows the following conventions

% - The ‘percentage’ sign denotes a Unix environment.

- > - The 'greater than' symbol represent a DOS/Windows environment.
- Italic* - The words given in italics represent an option.
- code - The guide differentiates the normal text from a program code through this color. Any code or part of a code/command line statements in this document will therefore be represented using this color.
- ... - Indicates that some portion of the material has been removed to simplify the description.
- [ ] - Indicates an optional argument that can be used in the command line.

# Contents

## Preface

<b>1.1</b>	<b>SANKHYA Machine Description Language (SMDL) .....</b>	<b>1</b>
1.1.1	Notation .....	2
1.1.2	Lexical Elements.....	4
1.1.3	Comments .....	4
1.1.4	SMDL 1.0 Specification .....	4
1.1.5	Description.....	11
	1.1.5.1 Processor Description .....	11
1.1.6	SMDL Representations (representation) .....	20
	1.1.6.1 Intermediate code specification .....	20
	1.1.6.2 Assembly code specification .....	21
	1.1.6.3 Machine code specification .....	21
1.1.7	Attributes .....	24
1.1.8	DTTFSymbol.....	25
1.1.9	SMDL Declarations .....	26
<b>1.2</b>	<b>Creating a Processor description file using SMDL .....</b>	<b>27</b>
1.2.1	Sample Processor .....	27
1.2.2	Modeling the Processor .....	29
	1.2.2.1 Creating a file.....	29
	1.2.2.2 Specifying the operands.....	29
	1.2.2.3 Specifying the Instructions .....	32

---

# *SMDL Reference*

---

## **1.1 SANKHYA Machine Description Language (SMDL)**

SANKHYA Machine Description Language (SMDL) is a simple yet powerful language for describing a processor. Using SMDL, processor information required by tools like code generator, assembler and simulator can be specified. SMDL allows the specification of a processor as a hierarchy of instruction bundles, instructions and operands. The important aspect is that for each of the above mentioned elements, the intermediate code, assembly code and machine code representation of the element can be specified.

In addition, SMDL provides constructs for concisely specifying a range of values for an operand, specifying an operand as a union of operands and provides support for specifying array of values for operands. It also allows attaching attributes to values thereby providing a powerful method to match instructions, operands and bundles based on attribute values.

This document explains the SMDL specification and provides detailed description of each of the construct used in SMDL.

### 1.1.1 Notation

#### Lexical

The following notations are used in the SMDL specification.

[c1-c2]	-	matches any of the characters between c1 and c2 (both inclusive)
[^list]	-	matches any character except those specified by 'list'
RE1   RE2	-	matches RE1 or RE2
RE*	-	matches zero or more occurrences of RE
RE+	-	matches one or more occurrences of RE
(RE)	-	matches RE
<name>	-	refers to a SMDL element
#	-	comments

#### Grammar Notation

The SMDL grammar is specified in a form similar to Backaus Naur Form (BNF). The following are followed in this form.

1. A production rule of the grammar of the form

$$A \rightarrow x$$

where,

A is a non-terminal

x is a string of terminals/non-terminals

is represented as

$$A := x$$

Multiple production rules involving the same non-terminal on the left handside, as mentioned below.

$$A \rightarrow x$$
$$A \rightarrow y$$
$$A \rightarrow z$$

can be combined as

$$A := x | y | z$$

2.. 'string' - refers to the literal string

**Example:**

'ID' refers to the string ID.

### 1.1.2 Lexical Elements

The following is the list of lexical elements used in the SMDL specification:

D := [0-9]	# Decimal Digit
O := [0-7]	# Octal Digits
H := [0-9A-Fa-f]	# Hexadecimal Digits
B := [0-1]	# Binary Digits
STRING := ['^']*	# Quoted String
INTEGER := (D)+   "-"(D)+	# A Decimal Integer
HEX_NUMBER := "0x"(H)+   "0X"(H)+	# An Hexadecimal Integer
BINARY_NUMBER := "0b"(B)+   "0B"(B)+	# A Binary number
IDENTIFIER := ([A-Z] [a-z])(([A-Z]+) ([a-z]+) ([0-9_]+))*	# An Identifier

### 1.1.3 Comments

SMDL supports C++ style single line comments (`'//'`).

### 1.1.4 SMDL 1.0 Specification

The following is the SANKHYA Machine Description Language Specification.

1. array\_type :=
  - (a) 'DTTFTextTokens'
  - (b) 'DTTFNumTokens'

- 
2. array\_name := IDENTIFIER
  
  3. array\_string\_list :=
    - (a) STRING |
    - (b) STRING ',' array\_string\_list
  
  4. array\_number\_list :=
    - (a) INTEGER |
    - (b) INTEGER ',' array\_number\_list
  
  5. array\_declaration :=
    - (a) array\_type array\_name '=' '{' array\_string\_list '}' ';' |
    - (b) array\_type array\_name '=' '{' array\_number\_list '}' ';' ;
  
  6. smdl\_declaration := array\_declaration
  
  7. smdl\_declaration\_list :=
    - (a) smdl\_declaration |
    - (b) smdl\_declaration smdl\_declaration\_list
  
  8. attribute\_value := IDENTIFIER
  
  9. attribute\_name := IDENTIFIER
  
  10. attribute\_value\_list :=
    - (a) attribute\_value |
    - (b) attribute\_value ':' attribute\_value
  
  11. attribute := attribute\_name '=' attribute\_value\_list
-

12. attribute\_list :=

- (a) attribute |
- (b) attribute ',' attribute\_list

13. const\_value :=

- (a) STRING |
- (b) INTEGER |
- (c) HEX\_NUMBER |
- (d) BINARY\_NUMBER

14. const\_binary\_value :=

- (a) 'or' '(' INTEGER ',' INTEGER ',' const\_value ')' |
- (b) IDENTIFIER '(' INTEGER ',' INTEGER ')' |
- (c) 'expr' '(' STRING ',' INTEGER ',' INTEGER ')' |
- (d) '\$' IDENTIFIER |
- (e) const\_value

15. representation\_name :=

- (a) 'icode' |
- (b) 'asm' |
- (c) 'mcode'

16. string\_value := STRING

17. binary\_value :=

const\_binary\_value

18. element\_name := STRING

- 
19. value :=  
(a) string\_value |  
(b) binary\_value |  
(c) attribute\_list element\_name |
20. value\_list :=  
(a) value |  
(b) value ',' value\_list
21. smdl\_bundle := 'DTTFBundle'
22. representation := representation\_name '=' '{' value\_list '}' ',' ;
23. representation\_list :=  
(a) representation |  
(b) representation representation\_list
24. range\_name := IDENTIFIER
25. range\_start := INTEGER
26. range\_end := INTEGER
27. range\_increment := INTEGER
28. range\_specification :=  
'range' '(' range\_name '=' range\_start ',' range\_end ','  
range\_increment ')'

- 
29. `range_representation` := `range_specification` '{' `representation_list` '}'
30. `union_name_list` :=  
(a) `IDENTIFIER` |  
(b) `IDENTIFIER` ',' `union_name_list`
31. `smdl_instruction` := 'DTTFInstruction'
32. `smdl_operand` := 'DTTFOperand'
33. `variable` := `IDENTIFIER`
34. `variable_list` :=  
(a) `variable` |  
(b) `variable` ',' `variable_list`
35. `operand_type` :=  
(a) 'Integer' |  
(b) 'Register' |  
(c) 'Address'
36. `user_defined_type` := `IDENTIFIER`
37. `element_type` :=  
(a) `user_defined_type` |  
(b) 'DTTFSymbol'
38. `child_element_declaration` := `element_type` `variable_list` ';' ;

39. element\_description :=

- (a) child\_element\_declaration |
- (b) representation\_list

40. element\_description\_list :=

- (a) element\_description |
- (b) element\_description element\_description\_list

41. bundle\_element :=

- (a) smdl\_bundle element\_name '{' element\_description\_list '}'

42. instruction\_element :=

- (a) smdl\_instruction element\_name '{' element\_description\_list '}' |
- (b) smdl\_instruction element\_name : 'union' '(' union\_name\_list ')' '{' '}'

43. operand\_element :=

- (a) smdl\_operand element\_name '{' element\_description\_list '}' |
- (b) smdl\_operand element\_name : operand\_type  
'{' element\_description\_list '}' |
- (c) smdl\_operand element\_name : 'union' '(' union\_name\_list ')' '{' '}'
- (d) smdl\_operand element\_name '{' range\_representation '}' |
- (e) smdl\_operand element\_name

44. element :=

- (a) bundle\_element ';' |
- (b) instruction\_element ';' |
- (c) operand\_element ';' |

45. `element_list` :=

- (a) `element` |
- (b) `element element_list`

46. `processor_name` := `STRING`

47. `processor_decl` :=

- (a) `'ID' = INTEGER ';' |`
- (b) `'asm' = '{' STRING '}' ';' ;'`

48. `smdl_processor` :=

- (a) `'DTTFProcessor' processor_name '{' processor_decl element_list '}' ;'`
- (b) `'DTTFProcessor' processor_name : processor_name '{' processor_decl element_list '}' ';' ;'`

49. `smdl_processor_list` :=

- (a) `smdl_processor` |
- (b) `smdl_processor smdl_processor_list`

50. `smdl_spec` :=

- (a) `smdl_processor_list` |
- (b) `smdl_declaration_list`

## 1.1.5 Description

A SMDL specification consists of one or more processor descriptions and data declarations.

### 1.1.5.1 Processor Description

A Processor description consists of the following elements:

- a) Processor Name, ID and ASM string declarations
- b) Operand element descriptions
- c) Instruction element descriptions
- d) Bundle element descriptions

#### a) Processor Name, ID and ASM string declarations

##### Syntax:

```

DTTFProcessor IDENTIFIER {
  ID = INTEGER;    // a unique identification string
  asm = { STRING }; // a string placed in assembly files for
                  // identifying the processor

  operand_element*
  ...
  instruction_element*
  ...
  bundle_element*
  ...
};

```

**Example:**

```
DTTFProcessor MIPS32 {  
  ID = 1;  
  asm = { "mips32" }  
  ...  
};
```

**b) Operand description (operand\_element):**

An operand description describes a register, memory or immediate operand that can occur as operands in an instruction supported by the processor.

An operand description can be:

- b.1) A simple operand description
- b.2) A range operand description
- b.3) A union description

b.1) A simple operand description consists of the following

- i) Operand Name
- ii) Operand Type (optional)
- iii) Intermediate code representation of the operand
- iv) Assembly code representation of the operand
- v) Machine code representation of the operand
- vi) Component operand declarations (if any)

**Syntax:**

- i) DTTFOperand IDENTIFIER {  
child\_element\_declaration\*  
  
representation\_list\*  
};
  
- ii) DTTFOperand IDENTIFIER : operand\_type {  
child\_element\_declaration\*  
  
representation\_list\*  
};

The child\_element\_declaration refers to

- a) any operands defined earlier and which are part of this operand
- b) DTTFSymbol elements

**Example:**

```
// PC register
DTTFOperand o_pc : Register {
  icode = { "pc" };
  asm  = { "pc" };
  mcode = { "1" };
};
```

## b.2) A range operand description

The range operand declaration can be used to specify a range of values for the operand. It specifies a range variable and the range of values which it can take. The range is specified using the initial value, final value and increment.

### Syntax:

- i) DTTFOperand IDENTIFIER {  
     range (IDENTIFIER = INTEGER, INTEGER, INTEGER) {  
         representation\_list\*  
     };  
 };
  
- ii) DTTFOperand IDENTIFIER : operand\_type {  
     range (IDENTIFIER = INTEGER, INTEGER, INTEGER) {  
         representation\_list\*  
     };  
 };

### Example:

```
// General Purpose Registers (r0-r31)
```

```
DTTFOperand o_gprv : Register {  

    range (i = 0,31,1) {  

        icode = { "r$i" };  

        asm = { "r$i" };  

    }  

};
```

```

        mcode = {"$i"};
    };
};

```

### b.3) A union description

The union description specifies an operand as a union of one or more other operands. This operand will match any of the members of the union.

#### Syntax:

```

DTTFOperand IDENTIFIER : union ( IDENTIFIER, IDENTIFIER, ... ) {

};

```

#### Example:

```

// register 'r0'
DTTFOperand o_gpr0 : Register {
    icode = { "r0" };
    asm = { "r0" };
    mcode = { 0 };
};

// general purpose registers
DTTFOperand o_gprv : Register {
    range (i = 1,31,1) {
        icode = { "r$i" };
        asm = { "r$i" };
    }
};

```

```

        mcode = {"$i"};
    };
};

// union of all registers
DTTFOperand o_gpr : union ( o_gpr0, o_gprv) { };

```

### c) Instruction description (instruction\_element):

This represents an instruction of a processor. An Instruction description consists of the following:

- i) Instruction Name
- ii) Operand declarations (child\_element\_declaration) (optional)
- iii) Intermediate code representation of the instruction
- iv) Assembly code representation of the instruction
- v) Machine code representation of the instruction

An instruction can also be specified as a union of other instructions

### Syntax:

```

DTTFInstruction IDENTIFIER {
    child_element_declaration*

    representation_list*
};

DTTFInstruction IDENTIFIER : union ( IDENTIFIER, IDENTIFIER,

```

```
... ) {
};
```

The `child_element_declaration` refers to

- a) any operands defined earlier and which are part of this instruction
- b) DTTFSymbol elements

**Example:**

```
// Add instruction (add rd,rs,rt)
DTTFInstruction i_rrr_add {
    o_gprv rd, rs, rt;
    icode = { "=", rd, "+", rs, rt };
    asm = { "add", rd,rs,rt };
    mcode = { or(0,6,0b100000); rd (11,5); rt(16,5);
              rs(21,5); or(6,5,0b000000); or(26,6,0b000000) };
};
```

```
DTTFInstruction i_rrr_sub {
    o_gprv rd, rs, rt;
    icode = { "=", rd, "-", rs, rt };
    asm = { "sub", rd,rs,rt };
    mcode = { or(0,6,0b100010); rd (11,5); rt(16,5); rs(21,5);
              or(6,5,0b000000); or(26,6,0b000000) };
};
```

```
// Arithmetic instructions
DTTFInstruction i_arith : union (i_rrr_add, i_rrr_sub) {};
```

**d) Bundle Description (bundle-description):**

A Bundle represents a set of instructions that a processor can execute as a unit. This concept has significance for VLIW processors where multiple instructions of different type can be executed during a clock cycle.

A bundle description consists of the following:

- i) Bundle name
- ii) child element declarations (if any)
- iii) Intermediate code representation of the bundle
- iv) Assembly code representation of the bundle
- v) Machine code representation of the bundle

**Syntax:**

```
DTTFBundle IDENTIFIER {  
    child_element_declaration*  
  
    representation_list*  
};
```

The `child_element_declaration` refers to

- a) any instructions, operands which are part of this bundle
- b) DTTFSymbol elements

**Example:**

```

// Add instruction (add rd,rs,rt)
DTTFInstruction i_rrr_add {
    o_gprv rd, rs, rt;
    icode = { "=", rd, "+", rs, rt };
    asm = { "add", rd,rs,rt };
    mcode = { or(0,6,0b100000); rd (11,5); rt(16,5);
              rs(21,5); or(6,5,0b000000); or(26,6,0b000000) };
};

DTTFInstruction i_rrr_sub {
    o_gprv rd, rs, rt;
    icode = { "=", rd, "-", rs, rt };
    asm = { "sub", rd,rs,rt };
    mcode = { or(0,6,0b100010); rd (11,5); rt(16,5); rs(21,5);
              or(6,5,0b000000); or(26,6,0b000000) };
};

// Arithmetic instructions
DTTFInstruction i_arith : union (i_rrr_add, i_rrr_sub) {};

// A 64-bit bundle containing two 32-bit integer unit instructions
DTTFBundle b_ii {
    i_arith i1;
    i_arith i2;

    icode = { i1, i2 };
    asm = { i1, i2 };
};

```

```
mcode = { i1(0,31); i2(32,63)};
};
```

### 1.1.6 SMDL Representations (representation)

The processor instructions, operands and bundles can be represented in the following formats in SMDL:

- a) Intermediate code (icode)
- b) Assembly code (asm)
- c) Machine code (mcode)

#### 1.1.6.1 Intermediate code specification

An intermediate code declaration specifies the representation of a bundle, instruction or operand in the SANKHYA intermediate code language (icode).

**Syntax:**

```
icode = { value_list };
```

where value\_list is a list of STRING or IDENTIFIER values.

**Example:**

```
icode = { "=", rd, "+", rs, rt };
icode = { i1, i2 };
icode = { "$i" };
```

### 1.1.6.2 Assembly code specification

An Assembly code declaration represents the assembly language form of a bundle, instruction or operand.

**Syntax:**

```
icode = { value_list };
```

where value\_list is a list of STRING or IDENTIFIER values.

**Example:**

```
asm = { "add", rd, rs, rt };
```

### 1.1.6.3 Machine code specification

A machine code declaration specifies the binary machine code representation of a bundle, instruction or operand. The bit position, number of bits and the bit values can be specified for each opcode and operands.

**Syntax:**

```
mcode = { value_list };
```

where, value\_list is a list of string or binary values.

The binary values can be of the following form:

(a) 'or' '(' INTEGER ',' INTEGER ',' const\_value ')'

The 'or' value specifies the starting bit position (first INTEGER), the number of bits (second INTEGER) and the value (const\_value) which should be set at the particular bit position in the machine word.

**Example:**

```
or(0,6,"100000"); // encode bits 0-5 (6 bits) with binary 100000
or(0,6,32); // encode bits 0-5 (6 bits) with decimal 32
or(0,6,0x20); // encode bits 0-5 (6 bits) with hex 0x20 (decimal 32)
or(0,6,0b100000); // encode bits 0-5 (6 bits) with binary 100000
```

(b) IDENTIFIER '(' INTEGER ',' INTEGER ')'

This format specifies that the element (IDENTIFIER) should be used to encode bits from start (INTEGER) to length (INTEGER) number of bits.

**Example:**

```
rd(6,5);
```

(c) 'expr' '(' STRING ',' INTEGER ',' INTEGER ')'

The 'expr' value specification specifies that an expression specified as STRING should be evaluated and placed starting from start position (first INTEGER) for certain number of bits (second INTEGER).

**Example:**

```
mcode = { expr("<< $rd 2", 0,5) };
```

**(d) '\$' IDENTIFIER**

This format can be used in range specifications and in expr value for referring to an element.

**Example:**

```
DTTFOperand register {
  range (i = 1,30,1) {
    mcode = { $i };
  };
};
```

**(e) const\_value**

This format can be used to just specify the value to be encoded without mentioning bit positions (which will be provided by a parent element).

```
mcode = { 1 };
mcode = { "1" };
mcode = { 0x1 };
mcode = { 0b1 };
```

**Example:**

```
// Register 'rd' is encoded in bits 0-4 and bits 26-31 should be
// binary 010000 and all other bits should be zeroes
mcode = { rd(0,5); or(5,22,0); or(26,6,0b010000) };
```

**1.1.7 Attributes**

SMDL allows attributes to be associated with values. An attribute can be used to specify the properties of string, element and binary values. An attribute is a name-value pair. Attributes can be used to differentiate between similar elements during translation.

**Example:**

```
DTTFInstruction add {
    register rs,rt,rd;

    // Specify width as 32 bits for this instruction
    asm = {"{width=32}add", rd, rs, rt };

};
```

Attributes are specified within {} bracket pairs before a string, binary or element value. In the example above, a 'width' attribute has been specified for the "add" instruction and its value has been specified as '32'. This allows the selection of this DTTFInstruction to be controlled based on the width attribute.

Multiple attributes can be specified for a value.

**Example:**

```
icode = { "{sign=s, width=32, type=int:ptr}=", rd, a };
```

The above specifies three attributes - sign, width and type for the '=' value. Each attribute in the list is separated by a comma. Multiple values for an attribute are specified using a ':' separated list as shown for 'type' attribute above.

**1.1.8 DTTFSymbol**

DTTFSymbol represents a symbol which can be added to a symbol table and can be looked-up. This can be used to represent program labels, function names etc.

**Example:**

```
// jump instruction ('j address')
DTTFInstruction i_s_j {
    DTTFSymbol s;
    icode = { "j", s };
    asm = { "j", s };
    mcode = { or(26,6,0b000010); expr("/ $s 4",0,26) };
};
```

### 1.1.9 SMDL Declarations

SMDL allows declaration of text and numeric arrays of values which can be used in element descriptions.

#### Syntax:

```
DTTFTextTokens IDENTIFIER = { STRING, STRING, ... };  
DTTFNumTokens IDENTIFIER = { INTEGER, INTEGER, ... };
```

#### Example:

```
DTTFTextTokens Regs = { "r1", "r2", "r3" };
```

```
DTTFOperand o_gprv : Register  
{  
  range (i = 0, 15, 1) {  
    icode = {"$Regs[$i]"};  
    asm = {"$Regs[$i]"};  
    mcode = {"$mRegs[$i]"};  
  };  
};
```

## 1.2 Creating a Processor description file using SMDL

This section explains the steps involved in describing a processor using SANKHYA Machine Description Language (SMDL).

### 1.2.1 Sample Processor

Following is the specification of RISC processor P.

1) 32 32-bit General Purpose Registers (r0-r31)

2) The following instructions are supported:

add rD, rS, rT - adds rS and rT and stores result in rD

sub rD, rS, rT - subtracts rT from rS and stores result in rD

ld rD, mem - loads 32 bits from address 'mem' and places the value in rD

st rD, mem - stores the contents of rD at memory address 'mem'

move rD, rS - moves the contents of rS to rD

where,

rD, rS, rT - are general purpose registers

mem - a memory operand

3) The following addressing modes are supported:

a) imm8[rS] - Register indirect with 8 bit displacement

where,

imm8 - 8-bit signed immediate value

4) The machine code for the instructions are as follows:

(each rD, rS, rT stands for one bit of the respective registers 'imm8' represents the 8 bits of the immediate value)

a) add rD, rS, rT

< MSB ----- LSB >

00001 rDrDrDrDrD rSrSrSrSrS rTrTrTrTrT 000000000000

b) sub rD, rS, rT

00010 rDrDrDrDrD rSrSrSrSrS rTrTrTrTrT 000000000000

c) ld rD, imm8[rS]

00011 rDrDrDrDrD imm8 rSrSrSrSrS 0000000000

d) st rD, imm8[rS]

00100 rDrDrDrDrD imm8 rSrSrSrSrS 0000000000

e) move rD, rS

00101 rDrDrDrDrD rSrSrSrSrS 000000000000000000

## 1.2.2 Modeling the Processor

The following are the steps involved in creating a model for the processor.

### 1.2.2.1 Creating a file

Create a file named p.md and add the following in it

```
DTTFProcessor P
{
  ID = 1;
  asm = { "proc P" };
};
```

The ID and asm fields can be used by tools which require identification information of the processor.

### 1.2.2.2 Specifying the operands

Define a DTTFOperand element inside the DTTFProcessor specification for each type of operand supported by the processor. For representing a range of values (like registers) use the range operand construct. Addressing modes can be treated as operands consisting of other simpler operands (ie registers, immediate etc).

The icode representation for registers is 'r' followed by a number. The assembly representation for registers is 'r' followed by a number. The machine code (mcode) representation for registers is the value of the register (0-31).

### General Purpose Registers:

```
// 32 General Purpose registers
DTTFOperand o_gprv : Register {
  range (i = 0,31,1) {
    icode = { "r$i" };
    asm = { "r$i" };
    mcode = { "$i" };
  };
};
```

### Immediate Values:

The 8-bit signed immediate value is represented as a range of values from -128 to +127 in icode, asm and mcode representations.

```
// 8-bit immediate value
DTTFOperand o_imm8 : Integer {
  range (i = -128,127,1) {
    icode = { "$i" };
    asm = { "$i" };
    mcode = { "$i" };
  };
};
```

### Addressing Mode operand:

The register indirect addressing mode with 8-bit signed displacement is represented as follows

```
icode : + register imm8
asm   : imm8[register]
mcode : The register is encoded starting at bit 9 for 5 bits and the
        imm8 value is encoded starting at bit 14 for 8 bits.
```

```
// Register indirect with displacement - addressing mode
DTTFOperand o_r_imm8 : Address {
    o_gprv rs;
    o_imm8 o8;

    icode = { "+", rs, o8 };
    asm = { o8, "[", rs, "]" };
    mcode = { rs(9,5); o8(14,8) };
};
```

Note how previously defined operands `o_gprv` and `o_imm8` have been used in the description of `'o_r_imm8'`. This completes the operand specification for the given processor.

### 1.2.2.3 Specifying the Instructions

Next the instructions supported by the processor should be described in icode, asm and mcode formats.

#### 1) Add instruction

```
DTTFInstruction i_rrr_add {
    o_gprv rd, rs, rt;

    icode = { "{width=32}=", rd, "+", rs, rt };
    asm = { "add", rd, rs, rt };
    mcode = { or(0,12,0b000000000000);
             rd(12,5);
             rt(17,5);
             rs(22,5);
             or(27,5,0b00001)
             };
};
```

The 3 register operands for the add instruction are specified as rd, rs and rt. The intermediate representation for add in SANKHYA icode is

$$= \text{reg} + \text{reg reg}$$

The 'width=32' attribute specifies that the instruction operates on 32-bit values

The machine code encoding is done as follows

```
bits 0-11  zeros
bits 12-16 encoded with machine encoding for register 'rd'
bits 17-21 encoded with machine encoding for register 'rs'
bits 22-26 encoded with machine encoding for register 'rt'
bits 27-31 binary 00001 (opcode for 'add').
```

The 'sub' instruction can be similarly defined.

## 2) Load instruction

```
DTTFInstruction i_rm_load {
    o_gprv rd;
    o_r_imm8 mem;

    icode = { "{width=32}=", rd, "{arity=1}*", mem };
    asm = { "ld", rd, mem };
    mcode = { or(0,9,0b000000000);
              mem(9,13);
              rd(22,5);
              or(27,5,0b00011)
            };
};
```

The register operand for the load instruction is specified as 'rd'. The memory operand (of type o\_r\_imm8) is specified as 'mem'.

The intermediate representation for load instruction in SANKHYA icode is

$$= \text{reg} * \text{mem}$$

where, '\*' is the unary dereferencing operator.

The 'width=32' attribute specifies that the instruction operates on 32-bit values. The 'arity=1' attribute specified for the '\*' operator signifies it as the unary dereference operator (and not the binary multiply operator).

The machine code encoding is done as follows:

bits 0-8    zeros  
 bits 9-13  encoded with machine encoding for 'mem' operand  
 bits 22-26 encoded with machine encoding for register 'rd'  
 bits 27-31 binary 00011 (opcode for 'ld').

The other instructions can be similarly specified.

This completes the description of the sample processor using SMDL. The STC tools can use the complete description of the processor to generate code for the processor, assemble the assembly programs written for the processor and for simulating the processor.

The processor description file for the processor P may look like the following:

```
DTTFProcessor P
{
  ID = 1;
  asm = { "proc p" };
}
```

```

// 32 General Purpose registers
DTTFOperand o_gprv : Register {
    range (i = 0,31,1) {
        icode = { "r$i" };
        asm = { "r$i" };
        mcode = { "$i" };
    };
};

// 8-bit immediate value
DTTFOperand o_imm8 : Integer {
    range (i = -128,127,1) {
        icode = { "$i" };
        asm = { "$i" };
        mcode = { "$i" };
    };
};

// Register indirect with displacement - addressing mode
DTTFOperand o_r_imm8 : Address {
    o_gprv rs;
    o_imm8 o8;

    icode = { "+", rs, o8 };
    asm = { o8, "[", rs, "]" };
    mcode = { rs(9,5); o8(14,8) };
};

// Add instruction

```

```

DTTFInstruction i_rrr_add {
    o_gprv rd, rs, rt;

    icode = { "{width=32}=", rd, "+", rs, rt };
    asm = { "add", rd, rs, rt };
    mcode = { or(0,12,0b000000000000);
             rd(12,5);
             rt(17,5);
             rs(22,5);
             or(27,5,0b00001)
            };
};

// Load instruction
DTTFInstruction i_rm_load {
    o_gprv rd;
    o_r_imm8 mem;

    icode = { "{width=32}=", rd, "{arity=1}*", mem };
    asm = { "ld", rd, mem };
    mcode = { or(0,9,0b0000000000);
             mem(9,13);
             rd(22,5);
             or(27,5,0b00011)
            };
};

// other instructions
// ...
};

```

---

## For More Information

STC Download	<a href="http://www.sankhya.com/info/products/tools/download.html">http://www.sankhya.com/info/products/tools/download.html</a>
STC Roadmap	<a href="http://www.sankhya.com/info/products/">http://www.sankhya.com/info/products/</a>
STC Documentation	<a href="http://www.sankhya.com/info/products/tools/docs.html">http://www.sankhya.com/info/products/tools/docs.html</a>
STC Sales & Support	<a href="mailto:sales@sankhya.com">sales@sankhya.com</a>

---

## SANKHYA

**Sankhya Technologies India Operations Private Limited**  
**#30-15-58,"Silver Willow",Third Floor,**  
**Dabagardens,**  
**Visakhapatnam 530 020, INDIA**  
**Tel:+91 891 554 2666**

**Sankhya Technologies Private Limited**  
**#13/2, "JayaShree", Third Floor, First Street, Jayalakshmipuram,**  
**Nungambakkam,**  
**Chennai 600 034, INDIA**  
**Tel: +91 44 2822 7358**  
**Fax: +91 44 2822 7357**

**Email: [sales@sankhya.com](mailto:sales@sankhya.com)**  
**<http://www.sankhya.com>**

---

SANKHYA, SANKHYA TECHNOLOGIES, SANKHYA Tools Collection, Dynamically Targetable Tools Framework, SANKHYA Software are trademarks, servicemarks or registered trademarks of Sankhya Technologies Private Limited. All other brands and names are the property of their respective owners.

---