64-bit ISA extensions to the AJIT processor

Madhav Desai

January 2, 2019

Contents

List of Tables

Chapter 1

The ISA Specification from IITB

1.1 Overview

The AJIT processor implements the Sparc-V8 ISA. We propose to extend this ISA to provide support for a native 64-bit integer datatype. The proposed extensions use the existing instruction encodings to the maximum extent possible.

All proposed extensions are:

$$\text{Register} \times \text{Register} \rightarrow \text{Register,Condition-codes}$$

type instructions. The load/store instructions are not modified.

We list the additional instructions in the subsequent sections. In each case, only the differences in the encoding relative to an existing Sparc-V8 instruction are provided.

1.2 ISA Extensions

The extensions to SPArc V8 for AJIT are described in this section.

1.2.1 Integer-Unit Extensions: Arithmetic-Logic Instructions

These instructions provide 64-bit arithmetic/logic support in the integer unit. The instructions work on 64-bit register pairs in most cases. Register-pairs are identified by a 5-bit even number (lowest bit must be 0). See Tables 1.1, 1.2, 1.3 and 1.4.

1.2.2 Integer-Unit Extensions: SIMD Instructions

These instructions are vector instructions which work on two source registers (each a 64 bit register pair), and produce a 64-bit vector result. The vector elements can be 8-bit/16-bit/32-bit. See Table 1.5.

1.2.3 Integer-Unit Extensions: SIMD Instructions II

These instructions are vector instructions which reduce a source register to a byte result. See Table 1.6.

1.2.4 Vector Floating Point Instructions

These are vector float operations which work on two single precision operand pairs to produce two single precision results. See Table 1.7.

1.2.5 CSWAP instructions

The Sparc-V8 ISA does not include a compare-and-swap (CAS) instruction which is very useful in achieving consensus among distributed agents when the number of agents is > 2. See Table 1.8.

We introduce a CSWAP instruction in two flavours:

ADDD	
same as ADD, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) + rs2(pair)
ADDDCC	
same as ADDCC, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) + rs2(pair), set Z,N
SUBD	
same as SUB, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) - rs2(pair)
SUBDCC	
same as SUBCC, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) - rs2(pair), set Z,N

Table 1.1: Addition and Subtraction Instructions

SLLD	
same as SLL, but with Instr[6:5]=2. if imm bit (Instr[13]) is 1, then Instr[5:0] is the shift-amount. else shift-amount is the lowest 5 bits of rs2. Note that rs2 is a 32-bit register.	rd(pair) ← rs1(pair) << shift-amount
SRLD	
same as SRL, but with Instr[6:5]=2. if imm bit (Instr[13]) is 1, then Instr[5:0] is the shift-amount. else shift-amount is the lowest 5 bits of rs2. Note that rs2 is a 32-bit register.	rd(pair) ← rs1(pair) >> shift-amount
SRAD	
same as SRA, but with Instr[6:5]=2. if imm bit (Instr[13]) is 1, then Instr[5:0] is the shift-amount. else shift-amount is the lowest 5 bits of rs2. Note that rs2 is a 32-bit register.	rd(pair) ← rs1(pair) >> shift-amount (with sign extension).

Table 1.2: Shift instructions

UMULD	
same as UMUL, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) * rs2(pair)
UMULDCC	
same as UMULCC, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) * rs2(pair), sets Z,
SMULD	
same as SMULD, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) * rs2(pair) (signed)
SMULDCC	
same as SMULCC, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) * rs2(pair) (signed) sets condition codes Z,N,Ovflow
UDIVD	
same as UDIV, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) / rs2(pair) **Note:** can generate div-by-zero trap.
UDIVDCC	
same as UDIVCC, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) / rs2(pair), sets condition codes Z,Ovflow **Note:** can generate div-by-zero trap.
SDIVD	
same as SDIV, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) / rs2(pair) (signed)
SDIVDCC	
same as SDIVCC, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) / rs2(pair) (signed), sets condition codes Z,N,Ovflow, **Note:** can generate div-by-zero trap.

Table 1.3: Multiplication and Division Instructions

ORD	
same as OR, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) | rs2(pair)
ORDCC	
same as ORCC, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) | rs2(pair), sets Z.
ORDN	
same as ORN, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) | (∼rs2(pair))
ORDNCC	
same as ORNCC, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) | (∼rs2(pair)), sets Z sets Z.
XORDCC	
same as XORCC, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair)ˆrs2(pair), sets Z sets Z.
XNORD	
same as XNOR, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair)ˆrs2(pair)
XNORDCC	
same as XNORCC, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair)ˆrs2(pair), sets Z
ANDD	
same as AND, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) . rs2(pair)
ANDDCC	
same as ANDCC, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) . rs2(pair), sets Z
ANDDN	
same as ANDN, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd(pair) ← rs1(pair) . (∼rs2(pair))
ANDDNCC	
same as ANDNCC, but with Instr[13]=0 (i=0), and Instr[5]=1.	rd ← rs1 . (∼rs2), sets Z

Table 1.4: 64 bit Logical Instructions

<table>
<tr><td colspan="2">VADDD8, VADDD16, VADDD32</td></tr>
<tr><td>Same as ADDD, but with Instr[13]=0 (i=0), and Instr[6:5]=2. Bits Instr[9:7] are a 3-bit field, which specify the data type
<table><tr><td>001</td><td>byte</td><td>(VADDD8)</td></tr><tr><td>010</td><td>half-word (16-bits)</td><td>(VADDD16)</td></tr><tr><td>100</td><td>word (32-bits)</td><td>(VADDD32)</td></tr></table></td><td>Performs a vector operation by considering the 64-bit operands as a vector of objects with specified data-type.</td></tr>
<tr><td colspan="2">VSUBD8, VSUBD16, VSUBD32</td></tr>
<tr><td>Same as ADDD, but with Instr[13]=0 (i=0), and Instr[6:5]=2. Bits Instr[9:7] are a 3-bit field, which specify the data type
<table><tr><td>001</td><td>byte</td><td>(VSUBD8)</td></tr><tr><td>010</td><td>half-word (16-bits)</td><td>(VSUBD16)</td></tr><tr><td>100</td><td>word (32-bits)</td><td>(VSUBD32)</td></tr></table></td><td>Performs a vector operation by considering the 64-bit operands as a vector of objects with specified data-type.</td></tr>
<tr><td colspan="2">VUMULD8, VUMULD16, VUMULD32</td></tr>
<tr><td>Same as ADDD, but with Instr[13]=0 (i=0), and Instr[6:5]=2. Bits Instr[9:7] are a 3-bit field, which specify the data type
<table><tr><td>001</td><td>byte</td><td>(VMULD8)</td></tr><tr><td>010</td><td>half-word (16-bits)</td><td>(VMULD16)</td></tr><tr><td>100</td><td>word (32-bits)</td><td>(VMULD32)</td></tr></table></td><td>Performs a vector operation by considering the 64-bit operands as a vector of objects with specified data-type.</td></tr>
<tr><td colspan="2">VSMULD8, VSUMLD16, VSMULD32</td></tr>
<tr><td>Same as ADDD, but with Instr[13]=0 (i=0), and Instr[6:5]=2. Bits Instr[9:7] are a 3-bit field, which specify the data type
<table><tr><td>001</td><td>byte</td><td>(VSMULD8)</td></tr><tr><td>010</td><td>half-word (16-bits)</td><td>(VSMULD16)</td></tr><tr><td>100</td><td>word (32-bits)</td><td>(VSMULD32)</td></tr></table></td><td>Performs a vector operation by considering the 64-bit operands as a vector of objects with specified data-type.</td></tr>
</table>

Table 1.5: SIMD Instructions

ORDBYTER (Byte-Reduce OR)	
op=2, op3[3:0]=0xe, op3[5:4]=0x2, contents[7:0] of rs2 specify a mask. Instr[31:30] (op) = 0x2 Instr[29:25] (rd) lowest bit assumed 0. Instr[24:19] (op3) = 111010 Instr[18:14] (rs1) lowest bit assumed 0. Instr[13] (i) = 0 (ignored) Instr[12:5] (zero) Instr[4:0] (rs2) 32-bit register is read.	rd ← (rs1_7.m7 | rs1_6.m6 | rs1_5.m5 ... | rs1_0.m0)
ANDDBYTER (Byte-Reduce AND)	
op=2, op3[3:0]=0xf, op3[5:4]=0x2, contents[7:0] of rs2 specify a mask. Instr[31:30] (op) = 0x2 Instr[29:25] (rd) lowest bit assumed 0. Instr[24:19] (op3) = 111110 Instr[18:14] (rs1) lowest bit assumed 0. Instr[13] (i) = 0 (ignored) Instr[12:5] (zero) Instr[4:0] (rs2) 32-bit register is read.	rd ← ((m7 ? rs1_7 : 0xff) . (m6 ? rs1_6 : 0xff) ... (m0 ? rs1_0 : 0xff))
XORDBYTER (Byte-Reduce XOR)	
op=2, op3[3:0]=0xe, op3[5:4]=0x3, contents[7:0] of rs2 specify a mask. Instr[31:30] (op) = 0x2 Instr[29:25] (rd) lowest bit assumed 0. Instr[24:19] (op3) = 111011 Instr[18:14] (rs1) lowest bit assumed 0. Instr[13] (i) = 0 (ignored) Instr[12:5] (zero) Instr[4:0] (rs2) 32-bit register is read.	rd ← (rs1_7.m7^rs1_6.m6^rs1_5.m5 ... ^rs1_0.m0)
ZBYTEDPOS (Positions-of-Zero-Bytes in D-Word)	
op=2, op3[3:0]=0xf, op3[5:4]=0x3, contents[7:0] of rs2/imm-value specify a mask. Instr[31:30] (op) = 0x2 Instr[29:25] (rd) lowest bit assumed 0. Instr[24:19] (op3) = 111011 Instr[18:14] (rs1) lowest bit assumed 0. Instr[13] (i) = if 0, use rs2, else Instr[7:0] Instr[12:5] = 0 (ignored if i=0) Instr[4:0] (rs2, if i=0) 32-bit register is read.	rd ← [b7_zero b6_zero b5_zero b4_zero ... b0_zero] (if mask-bit is zero then b⋆_zero is zero)

Table 1.6: SIMD Instructions II

VFADD	op=2, op3=0x34, opf=0x142
VFSUB	op=2, op3=0x34, opf=0x146
VFMUL	op=2, op3=0x34, opf=0x14a
VFDIV	op=2, op3=0x34, opf=0x14e
VFSQRT	op=2, op3=0x34, opf=0x12a

Table 1.7: SIMD Floating Point Operations. NaN propagated, but no traps. For each of these, rs1,rs2,rd are considered even numbers pointing to.

CSWAP64 (effective address in registers rs1 and rs2)	
op=3, op3=10 1111, i=0. Instr[31:30] (op) = 0x3 Instr[29:25] (rd) lowest bit assumed 0. Instr[24:19] (op3) = 101111 Instr[18:14] (rs1) lowest bit assumed 0. Instr[13] (i) = 0 (registers based effective address) Instr[12:5] (asi) = Address Space Identifier (See: Appendix G of V8) Instr[4:0] (rs2) 32-bit register is read.	

CSWAP64 (immediate effective address)	
op=3, op3=10 1111, i=1. Instr[31:30] (op) = 0x3 Instr[29:25] (rd) lowest bit assumed 0. Instr[24:19] (op3) = 101111 Instr[18:14] (rs1) lowest bit assumed 0. Instr[13] (i) = 1 (immediate effective address) Instr[12:0] (simm13) 13-bit immediate address.	

CSWAP64A (effective address in registers rs1 and rs2)	
op=3, op3=10 1111, i=0. Instr[31:30] (op) = 0x3 Instr[29:25] (rd) lowest bit assumed 0. Instr[24:19] (op3) = 111111 Instr[18:14] (rs1) lowest bit assumed 0. Instr[13] (i) = 0 (registers based effective address) Instr[12:5] (asi) = Address Space Identifier (See: Appendix G of V8) Instr[4:0] (rs2) 32-bit register is read.	

CSWAP64A (immediate effective address)	
op=3, op3=10 1111, i=1. Instr[31:30] (op) = 0x3 Instr[29:25] (rd) lowest bit assumed 0. Instr[24:19] (op3) = 111111 Instr[18:14] (rs1) lowest bit assumed 0. Instr[13] (i) = 1 (immediate effective address) Instr[12:0] (simm13) 13-bit immediate address.	

Table 1.8: CSWAP Instructions

Chapter 2

AJIT Support for the GNU Binutils Toolchain

2.1 Towards a GNU Binutils Toolchain

This section describes the details of adding the AJIT instructions to SPARC v8 part of GNU Binutils 2.22. We use the SPARC v8 manual to get the details of the sparc instruction. It's bit pattern is described *again*, and the new bit pattern required for AJIT is set up alongside. Bit layouts to determine the "match" etc. of the sparc port are also laid out. The SPARC manual also contains the "suggested asm syntax" that we adapt for the new AJIT instruction. The sections below follow the sections in chapter 1.2. For each instruction, we need to define its bitfields in terms of macros in `$BINUTILSHOME/include/opcode/sparc.h` and define the opcodes table in `$BINUTILSHOME/opcodes/sparc-opc.c`.

The AJIT instructions are variations of the corresponding SPARC V8 instructions. Please refer to the SPARC V8 manual for details of such corresponding SPARC instructions. For example, the`ADD` insn, pg. 108 (pg. 130 in PDF sequence) of the manual. Other instructions can be similarly found, and will not be mentioned.

2.1.1 Integer-Unit Extensions: Arithmetic-Logic Instructions

The integer unit extensions of AJIT are based on the SPARC V8 instructions. See: SPArc v8 architecture manual. SPARC v8 instructions are 32 bits long. The GNU Binutils 2.22 SPARC implementation defines a set of macros to capture the bits set by an instruction. These are the so called "match" masks. Please see the code in `$BINUTILSHOME/include/opcode/sparc.h` and `$BINUTILSHOME/opcodes/sparc-opc.c`.

2.1.1.1 Addition and subtraction instructions:

1. **ADDD**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	000000	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

ADDD: same as ADD, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`addd SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) + rs2(pair).

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  0000  0        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                  U  UUUU   UU
Final layout :  10DD  DDD0  0000  0SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x00	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

2. **ADDDCC**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	010000	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

New addition:

ADDDCC: same as ADDCC, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`adddcc SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) + rs2(pair), set Z,N

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  1000  0        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                        S  SSSS
Unused (0)   :                            U  UUUU   UU
Final layout :  10DD  DDD0  1000  0SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x10	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

3. **SUBD**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	000100	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

New addition:

SUBD: same as SUB, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`subd SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) - rs2(pair).

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  0010  0        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                        S  SSSS
Unused (0)   :                            U  UUUU   UU
Final layout :  10DD  DDD0  0010  0SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x04	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

4. **SUBDCC**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	010100	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

New addition:

SUBDCC: same as SUBCC, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`subdcc SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) - rs2(pair), set Z,N

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  1010  0        0           1
Destination  :    DD  DDD
Source 1     :                     SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                 U  UUUU   UU
Final layout :  10DD  DDD0  1010  0SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x14	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

2.1.1.2 Shift instructions:

The shift family of instructions of AJIT may each be considered to have two versions: a direct count version and a register indirect count version. In the direct count version the shift count is a part of the instruction

bits. In the indirect count version, the shift count is found on the register specified by the bit pattern in the instruction bits. The direct count version is specified by the 14^{th} bit, i.e. insn[13] (bit number 13 in the 0 based bit numbering scheme), being set to 1. If insn[13] is 0 then the register indirect version is specified.

Similar to the addition and subtraction instructions, the shift family of instructions of SPARC V8 also do not use bits from 5 to 12 (both inclusive). The AJIT processor uses bits 5 and 6. In particular bit 6 is always 1. Bit 5 may be used in the direct version giving a set of 6 bits available for specifying the shift count. The shift count can have a maximum value of 64. Bit 5 is unused in the register indirect version, and is always 0 in that case.

These instructions are therefore worked out below in two different sets: the direct and the register indirect ones.

1. The direct versions are given by insn[13] = 1. The 6 bit shift count is directly specified in the instruction bits. Therefore insn[5:0] specify the shift count. insn[6] = 1, distinguishes the AJIT version from the SPARC V8 version.

   (a) **SLLD**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	Lowest 5 bits of shift count
5	12	–	**Unused. Set to 0 by software.**	• **Use bit 5 to specify the msb of shift count.** • **Use bit 6 to distinguish AJIT from SPARC V8.** • **Set bits 7:12 to 0.**
13	13	0,1	The **i** bit	**Set i to "1"**
14	18	32	Source register 1, rs1	No change
19	24	100101	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

   SLLD: same as SLL, but with Instr[13]=0 (i=0), and Instr[5]=1.

   Syntax: "`slld SrcReg1, 6BitShiftCnt, DestReg`".
   (**Note:** In an assembly language program, when the second argument is a number, we have direct mode. A register number is prefixed with "r", and hence the syntax itself distinguished between direct and register indirect version of this instruction.)

   Semantics: rd(pair) ← rs1(pair) << shift count.

   Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       1  0010  1        1           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                              S  SSSS
Unused (0)   :                                 U  UUUU   UU
Final layout :  10DD  DDD1  0010  1SSS   SS1U  UUUU   U1II  IIII
```

   This will need another macro that sets bits 5 and 6. Let's call it `OP_AJIT_BITS_5_AND_6`. Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F5(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF5(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x25	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x1	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x2	in	`OP_AJIT_BITS_5_AND_6(a) /* ((a) & 0x3 << 6 */`

The AJIT bits (insn[6:5]) is set or reset internally by `F5` (just like in `F4`), and hence there are only three arguments.

(b) **SRLD**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	Lowest 5 bits of shift count
5	12	–	**unused**	• **Use bit 5 to specify the msb of shift count.** • **Use bit 6 to distinguish AJIT from SPARC V8.**
13	13	0,1	The **i** bit	**Set i to "1"**
14	18	32	Source register 1, rs1	No change
19	24	100110	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

SRLD: same as SRL, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`sral SrcReg1, 6BitShiftCnt, DestReg`".
(**Note:** In an assembly language program, when the second argument is a number, we have direct mode. A register number is prefixed with "r", and hence the syntax itself distinguished between direct and register indirect version of this instruction.)

Semantics: rd(pair) ← rs1(pair) >> shift count.

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       1  0011  0        1           1
Destination  :    DD  DDD
Source 1     :                       SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                  U  UUUU   UU
Final layout :  10DD  DDD1  0011  0SSS   SS1U  UUUU   U1II  IIII
```

This will need another macro that sets bits 5 and 6. Let's call it `OP_AJIT_BITS_5_AND_6`. Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F5(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF5(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x26	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x1	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x2	in	`OP_AJIT_BITS_5_AND_6(a) /* ((a) & 0x3 << 6 */`

The AJIT bits (insn[6:5]) is set or reset internally by `F5` (just like in `F4`), and hence there are only three arguments.

(c) **SRAD**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	Lowest 5 bits of shift count
5	12	–	**unused**	• **Use bit 5 to specify the msb of shift count.** • **Use bit 6 to distinguish AJIT from SPARC V8.**
13	13	0,1	The **i** bit	**Set i to "1"**
14	18	32	Source register 1, rs1	No change
19	24	100111	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

SRAD: same as SRA, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`srad SrcReg1, 6BitShiftCnt, DestReg`".
(**Note:** In an assembly language program, when the second argument is a number, we have direct mode. A register number is prefixed with "r", and hence the syntax itself distinguished between direct and register indirect version of this instruction.)

Semantics: rd(pair) ← rs1(pair) >> shift count (with sign extension).

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       1  0011  1        1           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                  U  UUUU   UU
Final layout :  10DD  DDD1  0011  1SSS   SS1U  UUUU   U1II  IIII
```

This will need another macro that sets bits 5 and 6. Let's call it `OP_AJIT_BITS_5_AND_6`. Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F5(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF5(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x27	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x1	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x2	in	`OP_AJIT_BITS_5_AND_6(a) /* ((a) & 0x3 << 6 */`

The AJIT bits (insn[6:5]) is set or reset internally by `F5` (just like in `F4`), and hence there are only three arguments.

2. The register indirect versions are given by insn[13] = 0. The shift count is indirectly specified in the 32 bit register specified in instruction bits. Therefore insn[4:0] specify the register that has the shift count. insn[6] = 1, distinguishes the AJIT version from the SPARC V8 version. In this case, insn[5] = 0, necessarily.

   (a) **SLLD**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	Register number
5	12	–	**unused**	• **Set bit 5 to 0.** • **Use bit 6 to distinguish AJIT from SPARC V8.**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	100101	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

SLLD: same as SLL, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`slld SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) << shift count register rs2.

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       1  0010  1        0           10
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                  U  UUUU   UU
Final layout :  10DD  DDD1  0010  1SSS   SS0U  UUUU   U10I  IIII
```

This will need another macro that sets bits 5 and 6. Let's call it `OP_AJIT_BITS_5_AND_6`. Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F5(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF5(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x25	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x2	in	`OP_AJIT_BITS_5_AND_6(a) /* ((a) & 0x3 << 6 */`

The AJIT bits (insn[6:5]) is set or reset internally by `F5` (just like in `F4`), and hence there are only three arguments.

(b) **SRLD**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	Register number
5	12	–	**unused**	• **Set bit 5 to 0.** • **Use bit 6 to distinguish AJIT from SPARC V8.**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	100110	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

SRLD: same as SRL, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`slld SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) >> shift count register rs2.

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       1  0011  0        0           10
Destination  :    DD  DDD
Source 1     :                     SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                               U  UUUU   UU
Final layout :  10DD  DDD1  0011  0SSS   SS0U  UUUU   U10I  IIII
```

This will need another macro that sets bits 5 and 6. Let's call it `OP_AJIT_BITS_5_AND_6`. Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F5(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF5(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x26	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x2	in	`OP_AJIT_BITS_5_AND_6(a) /* ((a) & 0x3 << 6 */`

The AJIT bits (insn[6:5]) is set or reset internally by `F5` (just like in `F4`), and hence there are only three arguments.

(c) **SRAD**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	Register number
5	12	–	**unused**	• **Set bit 5 to 0.** • **Use bit 6 to distinguish AJIT from SPARC V8.**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	100101	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

SRAD: same as SRA, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`slld SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) >> shift count register rs2 (with sign extension).

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       1  0011  1        0           10
Destination  :    DD  DDD
Source 1     :                     SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                               U  UUUU   UU
Final layout :  10DD  DDD1  0011  1SSS   SS0U  UUUU   U10I  IIII
```

This will need another macro that sets bits 5 and 6. Let's call it `OP_AJIT_BITS_5_AND_6`. Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F5(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF5(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x27	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x2	in	`OP_AJIT_BITS_5_AND_6(a) /* ((a) & 0x3 << 6 */`

The AJIT bits (insn[6:5]) is set or reset internally by `F5` (just like in `F4`), and hence there are only three arguments.

2.1.1.3 Multiplication and division instructions:

1. **UMULD**: Unsigned Integer Multiply AJIT, no immediate version (i.e. i is always 0).
   NOTE: The *suggested* mnemonic "umuld" conflicts with a mnemonic of the same name for another sparc architecture (other than v8). Hence we change it to: "**umuldaj**" in the implementation, but not in the documentation below.

   This conflict occurs despite forcing the GNU assembler to assemble for v8 only via the command line switch "-Av8"! It appears that forcing the assembler to use v8 is not universally applied throughout the assembler code.

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	001010	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

   UMULD: same as UMUL, but with Instr[13]=0 (i=0), and Instr[5]=1.

   Syntax: "`umuld SrcReg1, SrcReg2, DestReg`".

   Semantics: rd(pair) ← rs1(pair) * rs2(pair).

   Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  0101  0        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                 U  UUUU   UU
Final layout :  10DD  DDD0  0101  0SSS   SS0U  UUUU   UU1S  SSSS
```

   Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x0A	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

   The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

2. **UMULDCC**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	011010	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

New addition:

UMULDCC: same as UMULCC, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`umuldcc SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) * rs2(pair), set Z

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  1101  0        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                               U  UUUU   UU
Final layout :  10DD  DDD0  1101  0SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x1A	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

3. **SMULD**: Unsigned Integer Multiply AJIT, no immediate version (i.e. i is always 0).

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	001011	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

SMULD: same as SMUL, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`smuld SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) * rs2(pair) (signed).

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  0101  1        0            1
Destination  :    DD  DDD
Source 1     :                     SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                           U  UUUU   UU
Final layout :  10DD  DDD0  0101  1SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

```
Macro to set     =  F4(x, y, z)      in  sparc.h
Macro to reset   =  INVF4(x, y, z)   in  sparc.h
x                =  0x2              in  OP(x) /* ((x) & 0x3) << 30 */
y                =  0x0B             in  OP3(y) /* ((y) & 0x3f) << 19 */
z                =  0x0              in  F3I(z) /* ((z) & 0x1) << 13 */
a                =  0x1              in  OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */
```

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

4. **SMULDCC**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	011011	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

New addition:

SMULDCC: same as SMULCC, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`smuldcc SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) * rs2(pair) (signed), set Z,N,O

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  1101  1        0            1
Destination  :    DD  DDD
Source 1     :                     SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                           U  UUUU   UU
Final layout :  10DD  DDD0  1101  1SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

```
Macro to set     =  F4(x, y, z)      in  sparc.h
Macro to reset   =  INVF4(x, y, z)   in  sparc.h
x                =  0x2              in  OP(x) /* ((x) & 0x3) << 30 */
y                =  0x1B             in  OP3(y) /* ((y) & 0x3f) << 19 */
z                =  0x0              in  F3I(z) /* ((z) & 0x1) << 13 */
a                =  0x1              in  OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */
```

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

5. **UDIVD**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	001110	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

New addition:

UDIVD: same as UDIV, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`udivd SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) / rs2(pair).

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  0111  0        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                            U  UUUU   UU
Final layout :  10DD  DDD0  0111  0SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x0E	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

6. **UDIVDCC**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	011110	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

New addition:

UDIVDCC: same as UDIVCC, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`udivdcc SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) / rs2(pair), set Z,O

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  1111  0        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                               U  UUUU   UU
Final layout :  10DD  DDD0  1111  0SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x1E	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

7. **SDIVD**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	001111	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

New addition:

SDIVD: same as SDIV, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`sdivd SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) / rs2(pair) (signed).

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  0111  1        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                               U  UUUU   UU
Final layout :  10DD  DDD0  0111  1SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x0F	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

8. **SDIVDCC**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	011111	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

New addition:

SDIVDCC: same as SDIVCC, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`sdivdcc SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) / rs2(pair) (signed), set Z,N,O

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  1111  1        0           1
Destination  :    DD  DDD
Source 1     :                     SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                U  UUUU   UU
Final layout :  10DD  DDD0  1111  1SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x1F	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

2.1.1.4 64 Bit Logical Instructions:

No immediate mode, i.e. insn[5] ≡ i = 0, always.

1. **ORD**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	000010	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

ORD: same as OR, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`ord SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) | rs2(pair).

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  0001  0        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                               U  UUUU   UU
Final layout :  10DD  DDD0  0001  0SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x02	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

2. **ORDCC**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	010010	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

ORDCC: same as ORCC, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`ordcc SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) | rs2(pair), sets Z.

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  1001  0        0           1
Destination  :    DD  DDD
Source 1     :                     SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                              U  UUUU   UU
Final layout :  10DD  DDD0  1001  0SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x12	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

3. **ORDN**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	000110	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

ORDN: same as ORN, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`ordn SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) | (∼rs2(pair)).

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  0011  0        0           1
Destination  :    DD  DDD
Source 1     :                     SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                              U  UUUU   UU
Final layout :  10DD  DDD0  0011  0SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x06	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

4. **ORDNCC**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	010110	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

ORDNCC: same as ORN, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`ordncc SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) | (∼rs2(pair)), sets Z.

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  1011  0        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                  U  UUUU   UU
Final layout :  10DD  DDD0  0011  0SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x16	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

5. **XORDCC**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	010011	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

XORDCC: same as XORCC, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`xordcc SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) ^ rs2(pair), sets Z.

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  1001  1        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                  U  UUUU   UU
Final layout :  10DD  DDD0  1001  1SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x13	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

6. **XNORD**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	000111	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

XNORD: same as XNOR, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`xnordcc SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) ^ rs2(pair).

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  0011  1        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                  U  UUUU   UU
Final layout :  10DD  DDD0  0011  1SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x07	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

7. **XNORDCC**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	000111	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

XNORDCC: same as XNORD, but with Instr[13]=0 (i=0), and Instr[5]=1.
Syntax: "`xnordcc SrcReg1, SrcReg2, DestReg`".
Semantics: rd(pair) ← rs1(pair) ^ rs2(pair), sets Z.

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  0011  1        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                  U  UUUU   UU
Final layout :  10DD  DDD0  0011  1SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x07	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

8. **ANDD**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	000001	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

ANDD: same as AND, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`andd SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) · rs2(pair).

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  0000  1        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                  U  UUUU   UU
Final layout :  10DD  DDD0  0000  1SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x01	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

9. **ANDDCC**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	010001	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

ANDDCC: same as ANDCC, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`anddcc SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) · rs2(pair), sets Z.

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  1000  1        0           1
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                  U  UUUU   UU
Final layout :  10DD  DDD0  1000  1SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x11	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

10. **ANDDN**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	000101	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

ANDDN: same as ANDN, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`anddn SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) · (∼rs2(pair)).

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  0010  1        0           1
Destination  :    DD  DDD
Source 1     :                     SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                 U  UUUU   UU
Final layout :  10DD  DDD0  0010  1SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x05	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

11. **ANDDNCC**:

Start	End	Range	Meaning	New Meaning
0	4	32	Source register 2, rs2	No change
5	12	–	**unused**	**Set bit 5 to "1"**
13	13	0,1	The **i** bit	**Set i to "0"**
14	18	32	Source register 1, rs1	No change
19	24	010101	"**op3**"	No change
25	29	32	Destination register, rd	No change
30	31	4	Always "10"	No change

ANDDNCC: same as ANDN, but with Instr[13]=0 (i=0), and Instr[5]=1.

Syntax: "`anddncc SrcReg1, SrcReg2, DestReg`".

Semantics: rd(pair) ← rs1(pair) · (∼rs2(pair)), sets Z.

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  1010  1        0           1
Destination  :    DD  DDD
Source 1     :                     SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                                U  UUUU   UU
Final layout :  10DD  DDD0  0010  1SSS   SS0U  UUUU   UU1S  SSSS
```

Hence the SPARC bit layout of this instruction is:

```
Macro to set   =  F4(x, y, z)      in  sparc.h
Macro to reset =  INVF4(x, y, z)   in  sparc.h
x              =  0x2              in  OP(x) /* ((x) & 0x3) << 30 */
y              =  0x15             in  OP3(y) /* ((y) & 0x3f) << 19 */
z              =  0x0              in  F3I(z) /* ((z) & 0x1) << 13 */
a              =  0x1              in  OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */
```

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

2.1.1.5 Integer Unit Extensions Summary

- Addition and subtraction instructions:

1. **ADDD**:

```
Macro to set   =  F4(x, y, z)      in  sparc.h
Macro to reset =  INVF4(x, y, z)   in  sparc.h
x              =  0x2              in  OP(x) /* ((x) & 0x3) << 30 */
y              =  0x00             in  OP3(y) /* ((y) & 0x3f) << 19 */
z              =  0x0              in  F3I(z) /* ((z) & 0x1) << 13 */
a              =  0x1              in  OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */
```

   The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

2. **ADDDCC**:

```
Macro to set   =  F4(x, y, z)      in  sparc.h
Macro to reset =  INVF4(x, y, z)   in  sparc.h
x              =  0x2              in  OP(x) /* ((x) & 0x3) << 30 */
y              =  0x10             in  OP3(y) /* ((y) & 0x3f) << 19 */
z              =  0x0              in  F3I(z) /* ((z) & 0x1) << 13 */
a              =  0x1              in  OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */
```

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

3. **SUBD**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x04	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

4. **SUBDCC**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x14	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

- Multiplication and division instructions:

1. **UMULD**: Unsigned Integer Multiply AJIT, no immediate version (i.e. i is always 0).

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x0A	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

2. **UMULDCC**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x1A	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

3. **SMULD**: Unsigned Integer Multiply AJIT, no immediate version (i.e. i is always 0).

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x0B	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

4. **SMULDCC**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x1B	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

5. **UDIVD**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x0E	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

6. **UDIVDCC**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x1E	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

7. **SDIVD**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x0F	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

8. **SDIVDCC**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x1F	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

- 64 Bit Logical Instructions:

No immediate mode, i.e. insn[5] $\equiv$ i = 0, always.

1. **ORD**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x02	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

2. **ORDCC**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x12	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

3. **ORDN**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x06	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

4. **ORDNCC**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x16	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

5. **XORDCC**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x13	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

6. **XNORD**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x07	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

7. **XNORDCC**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x07	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

8. **ANDD**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x01	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

9. **ANDDCC**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x11	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

10. **ANDDN**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x05	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

11. **ANDDNCC**:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x15	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

- Shift instructions:

The shift family of instructions of AJIT may each be considered to have two versions: a direct count version and a register indirect count version. In the direct count version the shift count is a part of the instruction bits. In the indirect count version, the shift count is found on the register specified by the bit pattern in the instruction bits. The direct count version is specified by the 14^{th} bit, i.e. insn[13] (bit number 13 in the 0 based bit numbering scheme), being set to 1. If insn[13] is 0 then the register indirect version is specified.

Similar to the addition and subtraction instructions, the shift family of instructions of SPARC V8 also do not use bits from 5 to 12 (both inclusive). The AJIT processor uses bits 5 and 6. In particular bit 6 is always 1. Bit 5 may be used in the direct version giving a set of 6 bits available for specifying the shift count. The shift count can have a maximum value of 64. Bit 5 is unused in the register indirect version, and is always 0 in that case.

These instructions are therefore worked out below in two different sets: the direct and the register indirect ones.

1. The direct versions are given by insn[13] = 1. The 6 bit shift count is directly specified in the instruction bits. Therefore insn[5:0] specify the shift count. insn[6] = 1, distinguishes the AJIT version from the SPARC V8 version.

   (a) **SLLD**:
   This will need another macro that sets bits 5 and 6. Let's call it `OP_AJIT_BIT_2`. Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F5(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF5(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x25	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x1	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x2	in	`OP_AJIT_BIT_2(a) /* ((a) & 0x3 << 6 */`

The AJIT bits (insn[6:5]) is set or reset internally by `F5` (just like in `F4`), and hence there are only three arguments.

(b) **SRLD**:
This will need another macro that sets bits 5 and 6. Let's call it `OP_AJIT_BIT_2`. Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F5(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF5(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x26	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x1	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x2	in	`OP_AJIT_BIT_2(a) /* ((a) & 0x3 << 6 */`

The AJIT bits (insn[6:5]) is set or reset internally by `F5` (just like in `F4`), and hence there are only three arguments.

(c) **SRAD**:
This will need another macro that sets bits 5 and 6. Let's call it `OP_AJIT_BIT_2`. Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F5(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF5(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x27	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x1	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x2	in	`OP_AJIT_BIT_2(a) /* ((a) & 0x3 << 6 */`

The AJIT bits (insn[6:5]) is set or reset internally by `F5` (just like in `F4`), and hence there are only three arguments.

2. The register indirect versions are given by insn[13] = 0. The shift count is indirectly specified in the 32 bit register specified in instruction bits. Therefore insn[4:0] specify the register that has the shift count. insn[6] = 1, distinguishes the AJIT version from the SPARC V8 version. In this case, insn[5] = 0, necessarily.

(a) **SLLD**:
This will need another macro that sets bits 5 and 6. Let's call it `OP_AJIT_BIT_2`. Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F5(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF5(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x25	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x2	in	`OP_AJIT_BIT_2(a) /* ((a) & 0x3 << 6 */`

The AJIT bits (insn[6:5]) is set or reset internally by `F5` (just like in `F4`), and hence there are only three arguments.

(b) **SRLD**:
This will need another macro that sets bits 5 and 6. Let's call it `OP_AJIT_BIT_2`. Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F5(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF5(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x26	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x2	in	`OP_AJIT_BIT_2(a) /* ((a) & 0x3 << 6 */`

The AJIT bits (insn[6:5]) is set or reset internally by `F5` (just like in `F4`), and hence there are only three arguments.

(c) **SRAD**:
This will need another macro that sets bits 5 and 6. Let's call it `OP_AJIT_BIT_2`. Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F5(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF5(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x27	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x2	in	`OP_AJIT_BIT_2(a) /* ((a) & 0x3 << 6 */`

The AJIT bits (insn[6:5]) is set or reset internally by `F5` (just like in `F4`), and hence there are only three arguments.

2.1.2 Integer-Unit Extensions: SIMD Instructions

2.1.2.1 SIMD I instructions:

The first set of SIMD instructions are the three arithmetic instructions: add, sub, and mul. The "mul" instruction has signed and unsigned variations. Each of the three instructions have 8 bit (1 byte), 16 bit (1 half word) and 32 bit (1 word) versions. These versions are encoded as shown in table 2.1, where the first column denotes the bit numbers. We list all the SIMD I instructions version wise below.

987	**Type**	**Example**
001	Byte	e.g. VADDD8
010	Half-word (16-bits)	e.g. VADDD16
100	Word (32-bits)	e.g. VADDD32

Table 2.1: Data type encoding for SIMD I instructions.

1. **8 bit** (**1 Byte**)

   (a) **VADDD8**:

Start	**End**	**Range**	**Meaning**
0	4	32	Source register 2, rs2
5	6	4	*Always* 2, i.e. insn[6:5] = 10_b
7	9	8	**Data type** specifier: *Always* 0x1
10	12	–	**unused**
13	13	0,1	The **i** bit. *Always* 0.
14	18	32	Source register 1, rs1
19	24	000000	"**op3**"
25	29	32	Destination register, rd
30	31	4	Always "10"

VADDD8: same as ADD, but with Instr[13]=0 (i=0), and Instr[6:5]=2. Bits Instr[9:7] are a 3-bit field, which specify the data type

Syntax: "`vaddd8 SrcReg1, SrcReg2, DestReg`".

Semantics: *not given*

Bits layout:

```
Offsets      : 31       24 23       16  15        8   7        0
Bit layout   :  XXXX  XXXX  XXXX  XXXX   XXXX  XXXX   XXXX  XXXX
Insn Bits    :  10       0  0000  0        0     00   110
Destination  :    DD  DDD
Source 1     :                      SSS   SS
Source 2     :                                          S  SSSS
Unused (0)   :                               U  UU
Final layout :  10DD  DDD0  0000  0SSS   SS0U  UU00   110S  SSSS
To match     :  ^^       ^  ^^^^  ^        ^     ^^   ^^^
Bitfield name:  OP          OP3            i     9-   765
```

To set up bits 5 and 6, we use an already defined macro `OP_AJIT_BIT_5_AND_6`. The value to be set in these two bits is 0x2. To set bits 7 through 9, we define a new macro `OP_AJIT_BIT_7_THRU_9`. The value set in these three bits decides the *type*, byte, half word or word, of the instruction. For **vaddd8** instruction, bits 7 through 9 are set to the value 0x1. Both these macros influence the *unused* bits of the SPARC V8 architecture. So we define a macro `OP_AJIT_SET_UNUSED` that uses the previous two to set these bits unused by the SPARC V8, but used by AJIT.

```
#define OP_AJIT_BIT_7_THRU_9(x)   ((x) << 0x7)
#define OP_AJIT_SET_UNUSED        (OP_AJIT_BIT_5_AND_6(0x2) | \\
                                   OP_AJIT_BIT_7_THRU_9(0x1))
```

We can now define the final macro `F6(x, y, z, b, a)` to set the match bits for this instruction.

```
#define OP_AJIT_BIT_5(x)          (((x) & 0x1) << 5)
#define F4(x, y, z, b)            (F3(x, y, z) | OP_AJIT_BIT_5(b))
#define OP_AJIT_BIT_5_AND_6(x)    (((x) & 0x3) << 6)
#define F5(x, y, z, b)            (F3(x, y, z) | OP_AJIT_BIT_5_AND_6 (b))
#define OP_AJIT_BIT_7_THRU_9(x)   (((x) & 0x3) << 9)
#define F6(x, y, z, b, a)         (F5 (x, y, z, b) | OP_AJIT_BIT_7_THRU_9(a))
```

Hence the SPARC bit layout of this instruction is:

Macro to set	=	`F4(x, y, z)`	in	`sparc.h`
Macro to reset	=	`INVF4(x, y, z)`	in	`sparc.h`
x	=	0x2	in	`OP(x) /* ((x) & 0x3) << 30 */`
y	=	0x00	in	`OP3(y) /* ((y) & 0x3f) << 19 */`
z	=	0x0	in	`F3I(z) /* ((z) & 0x1) << 13 */`
a	=	0x1	in	`OP_AJIT_BIT(a) /* ((a) & 0x1) << 5 */`

The AJIT bit (insn[5]) is set internally by `F4`, and hence there are only three arguments.

2. **1 Half word** (**16 bit**)
3. **1 Word** (**32 bit**)

2.1.3 Integer-Unit Extensions: SIMD Instructions II

2.1.4 Vector Floating Point Instructions

2.1.5 CSWAP instructions

Chapter 3

Towards Assembler Extraction

3.1 Succinct ISA Descriptions

A. M. Vichare

ISA description languages seem to be at least 20 years old problem as of 2018. Attempts like MIMOLA or LISA have been made to describe processors and generate system software through them. This document records my attempts to develop such a language afresh, but for the AJIT processor of IIT Bombay. The benefit of hindsight should ideally be employed in this design process. I shall try to bring that in as a parallel activity along side the attempts to a practical design.

3.1.1 Instruction Set Design Study

This is the background work mainly of conceptual ideas, and study of some known examples.

3.1.1.1 Basic Concepts of Instruction Set Design

From: Henn-Patt, CA-Quant.Approach. Ed.5, App.A:

- **Type of internal storage**:
  - Stack: Operands are on the stack, and hence *implicit* in the instruction.
  - Accumulator: One of the operands is in the *accumulator* register, and hence implicit in the instruction.
  - Register-Memory: Memory *can be* a part of the instruction.
  - Register-Register: Memory is **never** a part of the instruction, except for the *load-store* pair of instructions.
  - Memory-Memory: All operands are in the memory and directly addressed as a part of the instruction. This is an old style is not often found today ($\sim$ 2018).
  - Variations: Dedicating some registers for some special purposes – **extended accumulator** or **special purpose registers**.

- Number of operands: This depends on the type of internal storage, and a design choice. An binary instruction (aka *operation*) may explicitly take two data source operands and one result destination operand. Or it may take only two operands, with one of them being **both** a data source and a result destination operand.

- **Memory layout addressing**:
  - Byte ordering: There are two ways to order a set of bytes of a multi-byte object (e.g. 32 bit, i.e. 4 byte integer).
    - **Little Endian**: The byte with the least significant bit can be stored at the smallest byte address, or
    - **Big Endian**: The byte with the least significant bit can be stored at the largest byte address.
  - Alignment needs: For multibyte objects, an architecture may need the components to be aligned on suitable address boundaries. Or it may not need them to be so aligned! If k is the number of bytes of a multibyte object, a is the address of the byte with the least significant bit, then the object is aligned if: $a = n \times k$, where n is a natural number. The address a is an integral multiple of the object size k.
  - Shifting needs: Consider reading a *single* byte aligned at a word address into a *64 bit* register. A single 64 bit read, i.e. a double word read, would be performed on double word aligned address. If the word aligned byte would **not** be double word aligned, then the byte that is read would not occupy the least significant position in the 64 bit register. In such cases for correct alignment, we will need to shift the byte read in by 3 positions (calculate this "3") so that it occupies the correct position in a 64 bit register.

- **Addressing Modes**:
  How do we address the primary memory?
  - *Immediate*: No addressing at all. The argument/s (i.e. operand/s) is/are given as a part of the instruction. There is a finite size, finite number of bits, and layout norms.
  - *Register Direct*: The operand/s is/are available in one or more registers. Instead of being placed in the instruction, the operands are available in the register.
    - *PC Relative*: A variant of register direct addressing where the register to be used is fixed as the program counter (i.e. the instruction pointer).
  - *Direct* or *Absolute*: The address is provided directly as an argument. There could be finite size definitions that could be same as or different from the size of the address bus.
  - *Register Indirect*: The operand location is given in one or more registers. The register size is expected to be the same as the size of the address bus.
    - *Auto Increment or Decrement*: A variant of register indirect where the indirection value in the register is either automatically incremented or decremented. Autoincrementing is useful for array traversals with the base address of the array in the register, and the array element size as the increment value. Autodecrementing is similarly useful for stack operations.
    - *Displacement*: A variant of register indirect addressing mode, the operand location is given as an *offset* (i.e. *displacement*), relative to a register indirect address. The memory location is thus the offset relative to the location given in a register.
    - *Indexed*: Another variant of register indirect addressing where the operand location is a well defined algebraic relation of values in a few registers. Thus, for example, the location might be given as a *sum* of values in two registers where one register has the "base" value, and the other has an "index" (i.e. an offset) relative to the base value.
    - *Scaled*: Yet another variant of register indirect addressing where the operand location is again a well defined algebraic relation of values in a few registers. The algebraic relation is a displacement relative to a "base" in one register and an integral scale up of "index" in another register.

– *Memory Indirect*: Adding one more level of indirection to the register indirect mode yields this mode. The location of the operand is now available at the memory location given by the register indirect mode.

The immediate, displacement, and register indirect addressing modes are predominantly used (about 75% to 99% of modes used).

- **Types and Size of Operands**:
  - Some specifications of size have standardized (e.g. IEEE floating point), some have become conventional (e.g. 8 bit byte, 2 byte half words, 4 byte words etc.), some are optionally supported by the processor architecture (e.g. strings, binary coded decimal, packed decimal). Representation is either tagged (not used much today $\sim$ 2018), or encoded within the opcode (preferred method today).
  - *Standardised*: IEEE Floating point – single and double precision. Single precision is 4 bytes, and double precision is 8 bytes.
  - *Conventional*:

Quad Word	Double word	Word	Half word	Byte	Bits
–	–	–	–	1	8
–	–	–	1	2	16
–	–	1	2	4	32
–	1	2	4	8	64
1	2	4	8	16	128

- **Operations in the Instruction Set**:
  Thumb rule: Simplest instructions are the most widely executed ones.

Type	**Description or Examples**
Arithmetic	Arithmetic operations on numbers: +, -, *, / etc.
Logical	Logical: AND, OR, NOT
Data Transfer	Load, Store, Move
Control Flow	Branch, Loop, Jump, Procedure call and return, Trap
System	OS System call, Virtual memory management
Floating point	Floating point +, -, *, / etc.
Decimal	Decimal +, -, *, / etc.
String	String move, compare, search
Graphics	Pixel and vertex operation, compression & decompression
Signal Processing	FFT, MAC

It might be useful to classify at a little more higher level:

Class	**Description or Examples**
Data Type based	Arithmetic, Logical, Floating point, Signal Processing, Graphics, Decimal, String
Data Transfer	All I/O
System Control	Control flow, System management

- **Instructions for Control Flow**:
  - No well defined convention for naming, but we will follow the text referred at the beginning of this section. Four main control flow instructions are usually offered.
    * Jump: These are unconditional.

    - Branch: These are conditional.
    - Procedure call.
    - Procedure return.
  - It is useful to use *PC-Relative* addressing mode to specify the destination address of a control flow instruction. This allows running the code independent of where it is loaded – a property called *position independence*. Position independence may not always be possible, especially if the target of control flow cannot be computed at compile time. In such cases other addressing modes are used. Register indirect addressing is useful for:
    1. Case analysis as in *switch* statements.
    2. Virtual functions or methods,
    3. Higher order functions or function pointers, and
    4. Dynamically shared libraries.
  - Condition code techniques: Three methods have been used –
    - Condition codes register (aka the flags register): A set of reserved special bits each indicating some defined condition is set or reset during an operation. The subsequent branch can test these bits. Typically, a separate branch instruction exists for each condition code bit.
    - Condition register: No dedicated register. Instead an arbitrary register can be designated as the "flags" register.
    - Compare and Branch: The comparison is a part of the branch instruction itself.

- **Encoding an Instruction Set**:
  - Variable sized.
  - Fixed width.
  - Hybrid: Some size varying part and some fixed part.

3.1.1.2 Some Examples of Instruction Set Design Languages

We will look at MIMOLA and LISA.

3.1.2 Instruction Set Description and Generation

We use an "engineering" approach to design and development of the language and its processors for describing an ISA and generating the processing software.

3.1.2.1 Basic Elements of the Structure of an Instruction Set Language

- Mnemonic: A string of "word" characters. A "word" is understood intuitively, and from the context.
- Class: ISAs frequently group instructions into *groups* or *classes* typically based on the semantics. Thus we can have logical instructions, integer arithmetic instructions, etc. We capture the class in this field.
- Bit pattern: An instruction is expressed using a set of binary digits, aka bits. The key attributes are:
  - Length: The total number of bits that make up the instruction. For our architecture this is a constant with value **32** bits.
  - Composition: An instruction bit pattern is composed of a subsets of bits that describe components of the bit pattern. The various *kinds* of subsets that may be needed are:
    -

3.1.3 Instruction Set Generation

3.1.3.1 Basic Elements of the "Language" to Describe the Instruction

- "insn-mnemonic" denotes the **mnemonic** of the instruction.
- "insn-bit-pattern" denotes the top level composite of the bit pattern of the given instruction.
  - "length" is a field of the bit pattern that records the total number of bits that make up the instruction.
  - "composition" is a variable length field that records the composition of the bits pattern.