64-bit ISA extensions to the AJIT processor

Madhav Desai

December 17, 2020

Contents

1 The ISA Specification from IITB **7**
1.1 Overview . . . 7
1.1.1 Changes relative to Version 1 . . . 7
1.1.1.1 Instruction Modifications . . . 7
1.1.1.2 ASR mappings . . . 8
1.2 ISA Extensions . . . 9
1.3 ISA Version 2 . . . 9
1.3.1 Integer-Unit Extensions: Arithmetic-Logic Instructions . . . 9
1.3.2 Integer-Unit Extensions: SIMD Instructions . . . 9
1.3.3 Integer-Unit Extensions: SIMD Instructions II . . . 9
1.3.4 Vector Floating Point Instructions . . . 9
1.3.5 FP Reduce . . . 9
1.3.6 Half Precision Conversion Operations . . . 9
1.3.7 CSWAP instructions . . . 15

2 AJIT Support for the GNU Binutils Toolchain **19**
2.1 Towards a GNU Binutils Toolchain . . . 19
2.1.1 Integer-Unit Extensions: Arithmetic-Logic Instructions . . . 19
2.1.1.1 Addition and subtraction instructions: . . . 20
2.1.1.2 Shift instructions: . . . 22
2.1.1.3 Multiplication and division instructions: . . . 28
2.1.1.4 64 Bit Logical Instructions: . . . 33
2.1.1.5 Integer Unit Extensions Summary . . . 41
2.1.2 Integer-Unit Extensions: SIMD Instructions . . . 47
2.1.2.1 SIMD I instructions: . . . 47
2.1.3 Integer-Unit Extensions: SIMD Instructions II . . . 48
2.1.4 Vector Floating Point Instructions . . . 48
2.1.5 CSWAP instructions . . . 48

3 Towards Assembler Extraction **49**
3.1 Succinct ISA Descriptions . . . 49
3.1.1 Instruction Set Design Study . . . 49
3.1.1.1 Basic Concepts of Instruction Set Design . . . 49
3.1.1.2 Some Examples of Instruction Set Design Languages . . . 52
3.1.2 Instruction Set Description and Generation . . . 52
3.1.2.1 Basic Elements of the Structure of an Instruction Set Language . . . 52
3.1.3 Instruction Set Generation . . . 53
3.1.3.1 Basic Elements of the “Language” to Describe the Instruction . . . 53

4 Packaging AJIT Within BuildRoot System **55**

4.1 List and Sequence of Files . . . 55
4.2 List and Sequence of Files Processing . . . 56
4.2.1 bfd/elf-bfd.h: Yes . . . 56
4.2.2 bfd/archures.c: Yes . . . 57
4.2.3 bfd/config.bfd: Yes . . . 58
4.2.4 bfd/cpu-sparc.c: Yes . . . 58
4.2.5 bfd/elf32-sparc.c: Yes . . . 58
4.2.6 bfd/elf.c: Yes . . . 58
4.2.7 bfd/elfcode.h: Yes . . . 58
4.2.8 bfd/elfxx-sparc.c: Yes . . . 59
4.2.9 bfd/elfxx-sparc.h: Yes . . . 59
4.2.10 bfd/targets.c: Yes . . . 59
4.2.11 bfd/bfd-in2.h: Maybe . . . 59
4.2.12 bfd/bfd-in.h: Maybe . . . 59
4.2.13 bfd/bfd.m4: Maybe . . . 59
4.2.14 bfd/configure: Maybe . . . 59
4.2.15 bfd/configure.in: Maybe . . . 59
4.2.16 bfd/elf64-ajit.c: Maybe . . . 59
4.2.17 bfd/elf64-sparc.c: Maybe . . . 59
4.2.18 bfd/freebsd.h: Maybe . . . 59
4.2.19 bfd/libbfd.h: Maybe . . . 59
4.2.20 bfd/Makefile.am: Maybe . . . 59
4.2.21 bfd/Makefile.in: Maybe . . . 59
4.2.22 bfd/nlm32-sparc.c: Maybe . . . 59
4.2.23 bfd/reloc.c: Maybe . . . 59
4.3 Studying the Build Process . . . 59

List of Tables

1.1 Addition and Subtraction Instructions . . . 10
1.2 Shift instructions . . . 10
1.3 Multiplication and Division Instructions . . . 11
1.4 64 bit Logical Instructions . . . 12
1.5 SIMD Instructions . . . 13
1.6 SIMD Instructions II . . . 14
1.7 SIMD Floating Point Operations . . . 16
1.8 SIMD Floating Point Reduce Operations . . . 16
1.9 SIMD Floating Point Reduce Operations . . . 16
1.10 CSWAP Instructions . . . 17

2.1 Data type encoding for SIMD I instructions. . . . 47

4.1 List of files 1 . . . 62
4.2 List of files 2 . . . 63
4.3 List of files 3 . . . 63
4.4 List of files 4 . . . 64
4.5 List of files 5 . . . 65

Chapter 1

The ISA Specification from IITB

ISA Version	2
Section below	1.2
ISA Version Date	September 2020.
Updated on	December 17, 2020.

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: Register $\times$ Register $\rightarrow$ 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.

Note This section records the ISA version 2 updates as received on September 2020.

1.1.1 Changes relative to Version 1

There has been some rationalization of the instructions. Further the ASR register mappings have been updated.

Notes on instruction naming: V* means a vector SIMD instruction

1.1.1.1 Instruction Modifications

- Some instructions have been removed.

  `VFDIV VFSQRT`

- Some instructions have been renamed.

- ADDDBYTER replaced with ADDDREDUCE8 instruction encoding modified as shown later.
  Given [a1 a2 a3 ... a8] calculate (a1+a2+...+a8)
- ORDBYTER replaced with ORDREDUCE8 instruction encoding modified as shown later.
- ANDDBYTER replaced with ANDDREDUCE8 instruction encoding modified as shown later.
- XORDBYTER replaced with XORDREDUCE8 instruction encoding modified as shown later.
- VFADD replaced with VFADD32 opcode modified.
  Given [x1 x2], [y1 y2] of single-precision numbers calculate [(x1+y1) (x2+y2)] this becomes VFADD32
  Added half-precision [x1 x2 x3 x4], [y1 y2 y3 y4] calculate [(x1+y1) (x2+y2) ... (x4 + y4)] this becomes VFADD16
- VFSUB replaced with VFSUB32 opcode modified.
- VFMUL replaced with VFMUL32 opcode modified.

- Some instructions have been added. Opcodes have been assigned (see below).

ADDDREDUCE16	ORDREDUCE16	ANDDREDUCE16	XORDREDUCE16
VFADD16	VFSUB16	VFMUL16	
FADDREDUCE16			
FSTOH	FHTOS		
VFHTOI16	VFI16TOH		

  Instruction behaviour is described below.

1.1.1.2 ASR mappings

ASR[31] and ASR[30] provide a free-running 64-bit counter running on the processor clock (same as in AJIT32).

ASR[29] is intialized to processor ID field (writes to this register are ignored).

The ancillary state register ASR[28] is interpreted as a floating point configuration register. The bits of this register are interpreted as follows:

31:8 unused
7:0 half-precision exponent width

This register is intialized to a value of 5 by default (as per the IEEE half-precision format). Valid values of the exponent width are between 5 and 14.

IEEE half precision = 1 sign + 5 exp + 10 mantissa bits would like
1 sign 08 exp 7 mantissa
1 sign 12 exp 3 mantissa

1.2 ISA Extensions

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

This has been superseded by version 2 as below.

1.3 ISA Version 2

1.3.1 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.

1.3.2 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.

1.3.3 Integer-Unit Extensions: SIMD Instructions II

These instructions are vector instructions which reduce a 64 bit source register to a destination using an associative operation. See Table 1.6.

1.3.4 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.

1.3.5 FP Reduce

This instruction adds the four half-precision numbers in the 64-bit FP register pair rs1, and produce a result into the 32-bit FP register. See Table 1.8.

1.3.6 Half Precision Conversion Operations

These instructions allow conversion between IEEE half-precision numbers and IEEE single/double precision numbers and integers. See Table 1.9.

Note: the double-to-half and half-to-double, int-to-half and half-to-int instructions are not provided. This is because, these transformations are likely to be rarer. Also, the `FDTOS`, `FDTOI`, `FITOS`, `FITOD` instructions together with the added `FSTOH`, `FHTOS` instructions are sufficient (at a minor cost).

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[7:6]=2. If imm bit (Instr[13]) is 1, then Instr[5:0] is the shift-amount, else shift-amount is the lowest 6 bits of rs2. Note that rs2 is a 32-bit register.	rd(pair) ← rs1(pair) << shift-amount
SRLD	
Same as SRL, but with Instr[7:6]=2. If imm bit (Instr[13]) is 1, then Instr[5:0] is the shift-amount, else shift-amount is the lowest 6 bits of rs2. Note that rs2 is a 32-bit register.	rd(pair) ← rs1(pair) >> shift-amount
SRAD	
Same as SRA, but with Instr[7:6]=2. If imm bit (Instr[13]) is 1, then Instr[5:0] is the shift-amount, else shift-amount is the lowest 6 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

VADDD8, VADDD16, VADDD32

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

001	byte	(VADDD8)
010	half-word (16-bits)	(VADDD16)
100	word (32-bits)	(VADDD32)

Performs a vector operation by considering the 64-bit operands as a vector of objects with specified data-type.

```
vaddd8 rs1, rs2, rd
vaddd16 rs1, rs2, rd
vaddd32 rs1, rs2, rd
```

VSUBD8, VSUBD16, VSUBD32

Same as SUBD, 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

001	byte	(VSUBD8)
010	half-word (16-bits)	(VSUBD16)
100	word (32-bits)	(VSUBD32)

Performs a vector operation by considering the 64-bit operands as a vector of objects with specified data-type.

```
vsubd8 rs1, rs2, rd
vsubd16 rs1, rs2, rd
vsubd32 rs1, rs2, rd
```

VUMULD8, VUMULD16, VUMULD32

Same as UMULD, 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

001	byte	(VMULD8)
010	half-word (16-bits)	(VMULD16)
100	word (32-bits)	(VMULD32)

Performs a vector operation by considering the 64-bit operands as a vector of objects with specified data-type.

```
vumuld8 rs1, rs2, rd
vumuld16 rs1, rs2, rd
vumuld32 rs1, rs2, rd
```

VSMULD8, VSUMLD16, VSMULD32

Same as SMULD, 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

001	byte	(VSMULD8)
010	half-word (16-bits)	(VSMULD16)
100	word (32-bits)	(VSMULD32)

Performs a vector operation by considering the 64-bit operands as a vector of objects with specified data-type.

```
vsmuld8 rs1, rs2, rd
vsmuld16 rs1, rs2, rd
vsmuld32 rs1, rs2, rd
```

Table 1.5: SIMD Instructions

ADDDREDUCE8	
op=2, op3[3:0]=0xd, op3[5:4]=0x2, contents[7:0] of rs2 specify a mask. Instr[31:30] (op) = 0x2 Instr[29:25] (rd) 32-bit register. Instr[24:19] (op3) = 101101 Instr[18:14] (rs1) lowest bit assumed 0. Instr[13] (i) = 0 (ignored) Instr[12:10] (zero) Instr[9:7] = 1 for byte reduce contents[7:0] of rs2 specify a mask. Instr[6:5] (zero) Instr[4:0] (rs2) 32-bit register is read.	rd ← (m7 ? rs1_7 : 0x0) + (m6 ? rs1_6 : 0x0) + (m5 ? rs1_5:0) . . . + (m0 ? rs1_0 : 0x0) `adddreduce8 %rs1, %rs2, %rd`

ADDDREDUCE16	
op=2, op3[3:0]=0xd, op3[5:4]=0x2, contents[3:0] of rs2 specify a mask. Instr[31:30] (op) = 0x2 Instr[29:25] (rd) 32-bit register. Instr[24:19] (op3) = 101101 Instr[18:14] (rs1) lowest bit assumed 0. Instr[13] (i) = 0 (ignored) Instr[12:10] (zero) Instr[9:7] = 2 for half word reduce contents[3:0] of rs2 specify a mask. Instr[6:5] (zero) Instr[4:0] (rs2) 32-bit register is read.	rd ← (m3 ? rs1_hw_3 : 0x0) + (m2 ? rs1_hw_2 : 0x0) + (m1 ? rs1_hw_1: 0x0) + (m0 ? rs1_hw_0 : 0x0) `adddreduce16 %rs1, %rs2, %rd`

ORDREDUCE8 (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) rd is a 32-bit register. Instr[24:19] (op3) = 101110 Instr[18:14] (rs1) lowest bit assumed 0. Instr[13] (i) = 0 (ignored) Instr[12:10] (zero) Instr[9:7] = 1 for byte reduce contents[7:0] of rs2 specify a mask. Instr[6:5] (zero) Instr[4:0] (rs2) 32-bit register is read.	rd ← (m7 ? rs1_7 : 0x0) | (m6 ? rs1_6 : 0x0) | (m5 ? rs1_5:0) . . . | (m0 ? rs1_0 : 0x0) `ordreduce8 %rs1, %rs2, %rd`

ORDREDUCE16 (Half Word-Reduce OR)	
op=2, op3[3:0]=0xe, op3[5:4]=0x2, contents[3:0] of rs2 specify a mask. Instr[31:30] (op) = 0x2 Instr[29:25] (rd) rd is a 32-bit register. Instr[24:19] (op3) = 101110 Instr[18:14] (rs1) lowest bit assumed 0. Instr[13] (i) = 0 (ignored) Instr[12:10] (zero) Instr[9:7] = 2 for half-word reduce, contents[3:0] of rs2 specify a mask. Instr[6:5] (zero) Instr[4:0] (rs2) 32-bit register is read.	rd ← (m3 ? rs1_3 : 0x0) | (m2 ? rs1_2 : 0x0) | (m1 ? rs1_1 : 0x0) | (m0 ? rs1_0 : 0x0) `ordreduce16 %rs1, %rs2, %rd`

ANDDREDUCE8 (Byte-Reduce OR)	
op=2, op3[3:0]=0xf, op3[5:4]=0x2, contents[7:0] of rs2 specify a mask.	

1.3.7 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.

We introduce a CSWAP instruction in two flavours. See Table 1.10.

The semantics of the instruction (the entire sequence is atomic)

```
TMPVAL = mem[rs1]  (load double, lock system bus)
if <rs2-pair/immediate> == TMPVAL
        (store double, unlock) mem[rs1] = <rd-pair>
        <rd-pair>  = TMPVAL
else
        (store double, unlock) mem[rs1] = TMPVAL
```

The write under else is redundant but is required in order to unlock the bus.

Similar to SWAP,

- `mem[rs1]` is left either with its value prior to the instruction or with the value in rd-pair.
- `<rd-pair>` is left either with its value prior to the instruction or with the value in mem[rs1].

The processor can check rd-pair after execution to confirm if the swap succeeded.

VFADD32	op=2, op3=0x34, opf=0x142	`vfadd32 %f0, %f2, %f4`
VFADD16	op=2, op3=0x34, opf=0x143	`vfadd16 %f0, %f2, %f4`
VFSUB32	op=2, op3=0x34, opf=0x144	`vfadd32 %f0, %f2, %f4`
VFSUB16	op=2, op3=0x34, opf=0x145	`vfadd16 %f0, %f2, %f4`
VFMUL32	op=2, op3=0x34, opf=0x146	`vfadd32 %f0, %f2, %f4`
VFMUL16	op=2, op3=0x34, opf=0x147	`vfadd16 %f0, %f2, %f4`
VFI16TOH	op=2, op3=0x34, opf=0x148	`vfi16toh %f0, %f2`
VFHTOI16	op=2, op3=0x34, opf=0x149	`vfhtoi16 %f0, %f2`

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.

FADDREDUCE16	op=2, op3=0x34, opf=0x150	`vfadd32 %f0, %f2, %f4`

Table 1.8: SIMD Floating Point Reduce Operations.

FSTOH	op=2, op3=0x34, opf=0x151	`fstoh %f1, %f2`
FHTOS	op=2, op3=0x34, opf=0x152	`fhtos %f1, %f2`

Table 1.9: SIMD Floating Point Reduce Operations.

CSWAP (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.	`cswap %rs1, %rs2, %rd` with asi specified.
CSWAP (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.	`cswap %rs1, imm, %rd`.
CSWAPA (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.	`cswapa %rs1, %rs2, %rd` with asi specified.
CSWAPA (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.	`cswapa %rs1, imm, %rd`.

Table 1.10: 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.

Chapter 4

Packaging AJIT Within BuildRoot System

4.1 List and Sequence of Files

Basic directory structure:

VARIABLE NAME	DESCRIPTION
TOP	Some top level directory of buildroot software.
BUILDROOT_VERSION	2014.08
BUILDROOT_TOP	${TOP}/buildroot-${BUILDROOT_VERSION}
OUTPUT	${BUILDROOT_TOP}/output
BUILD	${OUTPUT}/build

1. File ${BUILDROOT_TOP}/arch/Config.in: This file is used to add a new *architecture* to the buildroot system. Add the AJIT processor as follows:

```
config BR2_ajit
        bool "AJIT (IIT Bombay)"
        help
          Synopsys' IIT Bombay designed SPARC V8 like processor that is
          targetted for netblazers. Little endian.
```

   The indentation uses the TAB character. It appears to be mandatory as every "*/*Config.in" file I looked into uses it.

2. binutils/gas/configure.tgt: This is the file where the gas tool sets the target CPU files given the usual GNU triad (or quad, sometimes).

We first focus on binutils-2.22 for buildroot-2014.08 only. The gdb-7.6.2 port would be similar, and dealt with later.

4.2 List and Sequence of Files Processing

Since AJIT is based on Sparc V8, we first search for files in `binutils` that contain the string "`sparc`". The search is case insensitive. We first focus on adding ajit to the GNU BFD system in the binutils. Hence our search is in `binutils/bfd`. This yields the following list of files in tables 4.1 and 4.2.

We can eliminate files that are not related to ELF in any way. For example they may be dealing with the COFF format. Or they may be 64 bit; AJIT is a 32 bit system as of date.[1] Of the remaining, some would most certainly be candidate files for the AJIT port and some would probably be. Table 4.3 lists these files with a tag "yes" if the file is most certainly a candidate for AJIT port or a tag "maybe".

In the following sections we look at each file in detail. These sections are written by trying to guess the most probable next file, and then going back and forth across the other files to fill in the information.

4.2.1 `bfd/elf-bfd.h`: Yes

We take this as the first file to examine. Among the first files it includes is: `include/elf/common.h` which has the basic ELF definitions implemented for the GNU BFD system. For the definitions of ELF format, refer to the standard reference [ELF REFERENCE HERE]. Among the important fields is the `e_machine` field. We reproduce the relevant comment in `include/elf/common.h` below:

```
Values for e_machine, which identifies the architecture.  These
numbers are officially assigned  by registry@sco.com.  See below for a
list of ad-hoc numbers used during initial development.

If it is  necessary to assign new unofficial  EM_* values, please pick
large random numbers (0x8523, 0xa7f2, etc.) to minimize the chances of
collision with official or non-GNU unofficial values.

NOTE: Do not  just increment the most recent  number by one.  Somebody
else somewhere  will do exactly  the same thing,  and you will  have a
collision.  Instead, pick a random number.

Normally, each entity or maintainer  responsible for a machine with an
unofficial e_machine number should eventually ask registry@sco.com for
an officially blessed number to be added to the list above.
```

We will assign a temporary value 0xABCD as of now (i.e. 2020) until the AJIT architecture matures for a more global standardised support. At that point, we should follow the instructions in the comment above. We add the following to `include/elf/common.h`:

```
#define  EM_AJIT  0xABCD   /* The IITB AJIT Processor */
```

Note that this change will reflect only after the support is fully implemented.

The two other files included are: `include/elf/internal.h` and `include/elf/external.h`. These respectively describe the ELF format within the BFD system when in-memory and in-file.

An enum, "`enum elf_target_id`" is used to identify target specific extensions to the `elf_obj_tdata` and `elf_link_hash_table` structures. Both the latter structures are in this file too. Since AJIT has no extensions,

[1] 2020.

we do not seem to need adding an AJIT identifier to this enum. If, however, we do need to add then there are two main issues to consider given that the enum constants are lexicographically ordered:

1. The name "AJIT" will appear as the first enum in the lexicographical order. That will offset each subsequent enum value by +1 relative to its previous value. One value from this enum gets built into the tools for a specific system; in particular the GNU BFD library on that system. This is a potential problem. If, for a system (say the i386), we build two GNU BFD library versions, one using the standard binutils and the other using binutils with AJIT support, then the libraries will use different indices internally. If these indices result in different, but legal, ELF processing then we have a problem. Our resulting system is fragile and will break easily.

2. The other choice to place the "AJIT" enum is at the second last position. This will preserve the enums of all the other supported targets. We will not have the problem in 1 above. However, some other development effort might add another target to binutils and at the same place! Unless it so happens that the tools for these two non-standard targets come together we will not have problems. This is a low probability event, and we ignore it. If both these targets become standard then the development effort will have to ensure that these have distinct enum values.

4.2.2 `bfd/archures.c`: Yes

From the comments in the file we summarize:

About Architectures

The BFD approach keeps one atom in a BFD describing the architecture of the data attached to the BFD: a pointer to a "`bfd_arch_info_type`".

Pointers to structures can be requested independently of a BFD so that an architecture's information can be interrogated without access to an open BFD.

The architecture information is provided by each architecture package. The set of default architectures is selected by the macro "`SELECT_ARCHITECTURES`". This is normally set up in the @fileconfig/@vartarget.mt file of your choice. If the name is not defined, then all the architectures supported are included.

When BFD starts up, all the architectures are called with an initialize method. It is up to the architecture back end to insert as many items into the list of architectures as it wants to; generally this would be one for each machine and one for the default case (an item with a machine field of 0).

BFD's idea of an architecture is implemented in @filearchures.c. */

/*

SUBSECTION bfd_architecture

DESCRIPTION This enum gives the object file's CPU architecture, in a global sense—i.e., what processor family does it belong to? Another field indicates which processor within the family is in use. The machine gives a number which distinguishes different versions of the architecture, containing, for example, 2 and 3 for Intel i960 KA and i960 KB, and 68020 and 68030 for Motorola 68020 and 68030.

4.2.3 `bfd/config.bfd`: Yes

4.2.4 `bfd/cpu-sparc.c`: Yes

4.2.5 `bfd/elf32-sparc.c`: Yes

4.2.6 `bfd/elf.c`: Yes

The only place needed is the routine that groks the core file for NETBSD. Since AJIT is not ported to any other OS except GNU/Linux we do not need to add or change any code in this file.

4.2.7 `bfd/elfcode.h`: Yes

This file returns a `bfd_target` (defined in `bfd/bfd-in2.h`) object from the ELF file. Also `struct bfd` is in the same file.

4.2.8 `bfd/elfxx-sparc.c`: Yes

4.2.9 `bfd/elfxx-sparc.h`: Yes

4.2.10 `bfd/targets.c`: Yes

4.2.11 `bfd/bfd-in2.h`: Maybe

4.2.12 `bfd/bfd-in.h`: Maybe

4.2.13 `bfd/bfd.m4`: Maybe

4.2.14 `bfd/configure`: Maybe

4.2.15 `bfd/configure.in`: Maybe

4.2.16 `bfd/elf64-ajit.c`: Maybe

4.2.17 `bfd/elf64-sparc.c`: Maybe

4.2.18 `bfd/freebsd.h`: Maybe

4.2.19 `bfd/libbfd.h`: Maybe

4.2.20 `bfd/Makefile.am`: Maybe

4.2.21 `bfd/Makefile.in`: Maybe

4.2.22 `bfd/nlm32-sparc.c`: Maybe

4.2.23 `bfd/reloc.c`: Maybe

4.3 Studying the Build Process

To study the build process we build the GNU binutils-2.22 for at least two targets: a native and a cross. Our host machine is a 64 bit x86 as we write this. So we use the x86 or i386 as a the native machine, and the SPARC as the cross machine.[2] We install the binaries built on to a separate directory hierarchy for each target than the default `/usr/local`. We will refer to the i386 install folder as the `$X86INSTALLDIR` and the SPARC install folder as the `$SPARCINSTALLDIR`.

The build follows the usual steps to building GNU software: `configure` followed by `make` followed by `make install`. The standard output and the standard error of each step is individually redirected into files. This allows a systematic exploration of the sequence of the build process that has been actually followed. Here are the commands used to capture the details of the i386 build:

[2]For a cross target our build generates binaries that run **on** the host machine (i386 in this case) and generate output **for** the target machine (SPARC in this case).

1. `cd BINUTILS-SOURCES-FOR-i386`: change to the folder where we have the pristine binutils sources to be built for the i386.
2. `./configure --prefix=$X86INSTALLDIR --target=i386-pc-linux-gnu > i386-configure.stdout 2> i386-configure.stderr`
   This command sets up the build for the i386 target. The target is specified using the `--target` option to `configure`. The specification follows the GNU rules; the i386 target is specified as: `i386-pc-linux-gnu`. The build process is also informed that the installation is to be in the `$X86INSTALLDIR` folder. Apart from checking if the host has all the support required for the build, the `configure` may also set up some variables sensitive to the target, and may even generate some files that are target specific. In particular the `Makefile` that will build the entire target specific binutils is generated towards the end of its run.
3. `make > i386-make.stdout 2> i386-make.stderr`
   Using the `Makefile` generated by configure, this command is the main workhorse that builds the software system. Usually it compiles and links the programs. At times it also generates target specific files. Both these are critical to our study below.
4. `make install > i386-install.stdout 2> i386-install.stderr`
   This command installs the binaries, libraries and any header files generated by the build in the required directory hierarchy below the `$X86INSTALLDIR` folder.

Similarly for the SPARC build we have in summary:

1. `cd BINUTILS-SOURCES-FOR-SPARC`: change to the folder where we have the pristine binutils sources to be built for the SPARC.
2. `./configure --prefix=$SPARCINSTALLDIR --target=sparc-linux-gnu > sparc-configure.stdout 2> sparc-configure.stderr`
3. `make > sparc-make.stdout 2> sparc-make.stderr`
4. `make install > sparc-install.stdout 2> sparc-install.stderr`

The `configure` sequence of configuring over the folders in binutils is:

1. Configuring in ./intl
2. Configuring in ./libiberty
3. Configuring in ./bfd
4. Configuring in ./opcodes
5. Configuring in ./binutils
6. Configuring in ./etc
7. Configuring in ./gas
8. Configuring in ./gprof
9. Configuring in ./ld

Assuming a successful build our main source of study of the build process are the `*.stdout` files. Study of these files yields the following sequence:

1. Configuring in ./intl
2. Configuring in ./libiberty
3. Configuring in ./bfd
4. The files created during `configure` in `bfd/` are:
   (a) config.status: creating Makefile
   (b) config.status: creating doc/Makefile
   (c) config.status: creating bfd-in3.h:

(d) config.status: creating po/Makefile.in

(e) config.status: creating config.h

5. Building the libiberty library. The build process configures in `bfd` *before* it enters the `libiberty` to build this library. C files that contribute to this are:
   regex.c, cplus-dem.c, cp-demangle.c, md5.c, sha1.c, alloca.c, argv.c, choose-temp.c, concat.c, cp-demint.c, crc32.c, dyn-string.c, fdmatch.c, fibheap.c, filename_cmp.c, floatformat.c, fnmatch.c, fopen_unlocked.c, getopt.c, getopt1.c, getpwd.c, getruntime.c, hashtab.c, hex.c, lbasename.c, lrealpath.c, make-relative-prefix.c, make-temp-file.c, objalloc.c, obstack.c, partition.c, pexecute.c, physmem.c, pex-common.c, ne.c, ne.c, pex-unix.c, safe-ctype.c, bject.c, bject.c, bject-coff.c, bject-coff.c, bject-elf.c, bject-elf.c, bject-mach, bject-mach, sort.c, spaces.c, splay-tree.c, stack-limit.c, strerror.c, strsignal.c, rdinary.c, rdinary.c, xatexit.c, xexit.c, xmalloc.c, xmemdup.c, xstrdup.c, xstrerror.c, xstrndup.c, setproctitle.c
   After removing any previous libiberty library files, the libiberty library is built afresh using `ar` and `ranlib`. Also a list of these object files is collected in the file `required-list`.
6. Building in `bfd`.
   The sequence here is:
   (a) Create `bfdver.h`
   (b) Create `elf32-target.h`: Commands sequence is: `rm -f elf32-target.h`, `sed -e s/NN/32/g < ./elfxx-target.h > elf32-target.new`, `mv -f elf32-target.new elf32-target.h`.
   (c) Create `elf64-target.h`: Commands sequence is: `rm -f elf64-target.h`, `sed -e s/NN/64/g < ./elfxx-target.h > elf64-target.new`, `mv -f elf64-target.new elf64-target.h`.
   (d) Create `targmatch.h`: Commands sequence is: `rm -f targmatch.h`, `sed -f ./targmatch.sed < ./config.bfd > targmatch.new`, `mv -f targmatch.new targmatch.h`.
   (e) Build the BFD specific documentation. We skip these details.
   (f) Create `bfd.h`. Commands sequence is: `rm -f bfd-tmp.h`, `cp bfd-in3.h bfd-tmp.h`, `/bin/bash ./../move-if-change bfd-tmp.h bfd.h`, `rm -f bfd-tmp.h`, `touch stmp-bfd-h`,

<table>
<tr><td>bfd/aoutf1.h</td><td>A.out “format 1” file handling code for BFD.</td></tr>
<tr><td>bfd/aout-sparcle.c</td><td>BFD backend for sparc little-endian aout binaries.</td></tr>
<tr><td>bfd/aoutx.h</td><td>BFD semi-generic back-end for a.out binaries.</td></tr>
<tr><td>bfd/archures.c</td><td>BFD library support routines for architectures.</td></tr>
<tr><td>bfd/bfd-in2.h</td><td>Main header file for the bfd library: portable access to object files. This file is automatically generated from
<table><tr><td>“bfd-in.h”</td><td>“init.c”</td><td>“opncls.c”</td></tr><tr><td>“libbfd.c”</td><td>“bfdio.c”</td><td>“bfdwin.c”</td></tr><tr><td>“section.c”</td><td>“archures.c”</td><td>“reloc.c”</td></tr><tr><td>“syms.c”</td><td>“bfd.c”</td><td>“archive.c”</td></tr><tr><td>“corefile.c”</td><td>“targets.c”</td><td>“format.c”</td></tr><tr><td>“linker.c”</td><td>“simple.c”</td><td>“compress.c”</td></tr></table>Run “<code>make headers</code>” in your build <code>bfd/</code> to regenerate.</td></tr>
<tr><td>bfd/bfd-in.h</td><td>Main header file for the bfd library: portable access to object files.</td></tr>
<tr><td>bfd/bfd.m4</td><td>This file was derived from acinclude.m4.</td></tr>
<tr><td>bfd/cf-sparclynx.c</td><td>BFD back-end for Sparc COFF LynxOS files.</td></tr>
<tr><td>bfd/coffcode.h</td><td>Support for the generic parts of most COFF variants, for BFD.</td></tr>
<tr><td>bfd/coff-sparc.c</td><td>BFD back-end for Sparc COFF files.</td></tr>
<tr><td>bfd/coff-tic4x.c</td><td>BFD back-end for TMS320C4X coff binaries.</td></tr>
<tr><td>bfd/coff-tic54x.c</td><td>BFD back-end for TMS320C54X coff binaries.</td></tr>
<tr><td>bfd/config.bfd</td><td>Convert a canonical host type into a BFD host type. Set shell variable <code>targ</code> to canonical target name, and run using “<code>. config.bfd</code>”.
Sets the following shell variables:
<table><tr><td><code>targ_defvec</code></td><td>Default vector for this target</td></tr><tr><td><code>targ_selvecs</code></td><td>Vectors to build for this target</td></tr><tr><td><code>targ64_selvecs</code></td><td>Vectors to build if <code>--enable-64-bit-bfd</code> is given or if host is 64 bit.</td></tr><tr><td><code>targ_archs</code></td><td>Architectures for this target</td></tr><tr><td><code>targ_cflags</code></td><td><code>$(CFLAGS)</code> for this target (FIXME: pretty bogus)</td></tr><tr><td><code>targ_underscore</code></td><td>Whether underscores are used: yes or no</td></tr></table></td></tr>
<tr><td>bfd/configure</td><td>Guess values for system-dependent variables and create Makefiles.</td></tr>
<tr><td>bfd/configure.in</td><td>Process this file with autoconf to produce a configure script.</td></tr>
<tr><td>bfd/cpu-sparc.c</td><td>BFD support for the SPARC architecture.</td></tr>
<tr><td>bfd/elf32-cris.c</td><td>CRIS-specific support for 32-bit ELF. In comment.</td></tr>
<tr><td>bfd/elf32-m68hc1x.h</td><td>Motorola 68HC11/68HC12-specific support for 32-bit ELF. In comment.</td></tr>
<tr><td>bfd/elf32-sparc.c</td><td>SPARC-specific support for 32-bit ELF.</td></tr>
<tr><td>bfd/elf64-ajit.c</td><td>SPARC-specific support for 64-bit ELF</td></tr>
<tr><td>bfd/elf64-sparc.c</td><td>SPARC-specific support for 64-bit ELF</td></tr>
<tr><td>bfd/elf-bfd.h</td><td>BFD back-end data structures for ELF files.</td></tr>
<tr><td>bfd/elf.c</td><td>ELF executable support for BFD.</td></tr>
<tr><td>bfd/elfcode.h</td><td>ELF executable support for BFD.</td></tr>
<tr><td>bfd/elfxx-sparc.c</td><td>SPARC-specific support for ELF.</td></tr>
<tr><td>bfd/elfxx-sparc.h</td><td>SPARC ELF specific backend routines.</td></tr>
<tr><td>bfd/freebsd.h</td><td>BFD back-end definitions used by all FreeBSD targets.</td></tr>
<tr><td>bfd/libaout.h</td><td>BFD back-end data structures for a.out (and similar) files.</td></tr>
<tr><td>bfd/libbfd.h</td><td>Declarations used by bfd library *implementation*. (This include file is not for users of the library.) This file is automatically generated from
<table><tr><td>“libbfd-in.h”</td><td>“init.c”</td><td>“libbfd.c”</td></tr><tr><td>“bfdio.c”</td><td>“bfdwin.c”</td><td>“cache.c”</td></tr><tr><td>“reloc.c”</td><td>“archures.c”</td><td>“elf.c”</td></tr></table>Run “<code>make headers</code>” in your build <code>bfd/</code> to regenerate.</td></tr>
</table>

Table 4.1: List of files in `binutils/bfd` that contain the word “`sparc`”. Continued in Table 4.2.

bfd/lynx-core.c	BFD back end for Lynx core files
bfd/mach-o.c	Mach-O support for BFD.
bfd/Makefile.am	Process this file with automake to generate Makefile.in
bfd/Makefile.in	Makefile.in generated by automake 1.11.1 from Makefile.am.
bfd/mipsbsd.c	BFD backend for MIPS BSD (a.out) binaries.
bfd/netbsd-core.c	BFD back end for NetBSD style core files
bfd/nlm32-sparc.c	Support for 32-bit SPARC NLM (NetWare Loadable Module)
bfd/pdp11.c	BFD back-end for PDP-11 a.out binaries.
bfd/reloc.c	BFD support for handling relocation entries.
bfd/sparclinux.c	BFD back-end for linux flavored sparc a.out binaries.
bfd/sparclynx.c	BFD support for Sparc binaries under LynxOS.
bfd/sparcnetbsd.c	BFD back-end for NetBSD/sparc a.out-ish binaries.
bfd/sunos.c	BFD backend for SunOS binaries.
bfd/targets.c	Generic target-file-type support for the BFD library.

Table 4.2: Continued from Table 4.1. List of files in `binutils/bfd` that contain the word "`sparc`".

bfd/archures.c	yes
bfd/config.bfd	yes
bfd/cpu-sparc.c	yes
bfd/elf32-sparc.c	yes
bfd/elf-bfd.h	yes
bfd/elf.c	yes
bfd/elfcode.h	yes
bfd/elfxx-sparc.c	yes
bfd/elfxx-sparc.h	yes
bfd/targets.c	yes
bfd/bfd-in2.h	maybe
bfd/bfd-in.h	maybe
bfd/bfd.m4	maybe
bfd/configure	maybe
bfd/configure.in	maybe
bfd/elf64-ajit.c	maybe
bfd/elf64-sparc.c	maybe
bfd/freebsd.h	maybe
bfd/libbfd.h	maybe
bfd/Makefile.am	maybe
bfd/Makefile.in	maybe
bfd/nlm32-sparc.c	maybe
bfd/reloc.c	maybe

Table 4.3: List of files in `binutils/bfd` that contain the word "`sparc`" and are possible candidate files for the AJIT port.

<table>
<tr><td>bfd/archures.c (y)</td><td>BFD library support routines for architectures.</td></tr>
<tr><td>bfd/bfd-in2.h (m)</td><td>Main header file for the bfd library: portable access to object files. This file is automatically generated from
<table><tr><td>“bfd-in.h”</td><td>“init.c”</td><td>“opncls.c”</td></tr><tr><td>“libbfd.c”</td><td>“bfdio.c”</td><td>“bfdwin.c”</td></tr><tr><td>“section.c”</td><td>“archures.c”</td><td>“reloc.c”</td></tr><tr><td>“syms.c”</td><td>“bfd.c”</td><td>“archive.c”</td></tr><tr><td>“corefile.c”</td><td>“targets.c”</td><td>“format.c”</td></tr><tr><td>“linker.c”</td><td>“simple.c”</td><td>“compress.c”</td></tr></table>Run “<code>make headers</code>” in your build <code>bfd/</code> to regenerate.</td></tr>
<tr><td>bfd/bfd-in.h (m)
bfd/bfd.m4 (m)</td><td>Main header file for the bfd library: portable access to object files.
This file was derived from acinclude.m4.</td></tr>
<tr><td>bfd/config.bfd (y)</td><td>Convert a canonical host type into a BFD host type. Set shell variable <code>targ</code> to canonical target name, and run using “<code>. config.bfd</code>”.
Sets the following shell variables:
<table><tr><td><code>targ_defvec</code></td><td>Default vector for this target</td></tr><tr><td><code>targ_selvecs</code></td><td>Vectors to build for this target</td></tr><tr><td><code>targ64_selvecs</code></td><td>Vectors to build if <code>--enable-64-bit-bfd</code> is given or if host is 64 bit.</td></tr><tr><td><code>targ_archs</code></td><td>Architectures for this target</td></tr><tr><td><code>targ_cflags</code></td><td><code>$(CFLAGS)</code> for this target (FIXME: pretty bogus)</td></tr><tr><td><code>targ_underscore</code></td><td>Whether underscores are used: yes or no</td></tr></table></td></tr>
<tr><td>bfd/configure (m)
bfd/configure.in (m)
bfd/cpu-sparc.c (y)
bfd/elf32-sparc.c (y)
bfd/elf64-ajit.c (m)
bfd/elf64-sparc.c (m)
bfd/elf-bfd.h (y)
bfd/elf.c (y)
bfd/elfcode.h (y)
bfd/elfxx-sparc.c (y)
bfd/elfxx-sparc.h (y)
bfd/freebsd.h (m)</td><td>Guess values for system-dependent variables and create Makefiles.
Process this file with autoconf to produce a configure script.
BFD support for the SPARC architecture.
SPARC-specific support for 32-bit ELF.
SPARC-specific support for 64-bit ELF
SPARC-specific support for 64-bit ELF
BFD back-end data structures for ELF files.
ELF executable support for BFD.
ELF executable support for BFD.
SPARC-specific support for ELF.
SPARC ELF specific backend routines.
BFD back-end definitions used by all FreeBSD targets.</td></tr>
<tr><td>bfd/libbfd.h (m)</td><td>Declarations used by bfd library *implementation*. (This include file is not for users of the library.) This file is automatically generated from
<table><tr><td>“libbfd-in.h”</td><td>“init.c”</td><td>“libbfd.c”</td></tr><tr><td>“bfdio.c”</td><td>“bfdwin.c”</td><td>“cache.c”</td></tr><tr><td>“reloc.c”</td><td>“archures.c”</td><td>“elf.c”</td></tr></table>Run “<code>make headers</code>” in your build <code>bfd/</code> to regenerate.</td></tr>
<tr><td>bfd/Makefile.am (m)
bfd/Makefile.in (m)
bfd/nlm32-sparc.c (m)
bfd/reloc.c (m)
bfd/targets.c (y)</td><td>Process this file with automake to generate Makefile.in
Makefile.in generated by automake 1.11.1 from Makefile.am.
Support for 32-bit SPARC NLM (NetWare Loadable Module)
BFD support for handling relocation entries.
Generic target-file-type support for the BFD library.</td></tr>
</table>

Table 4.4: List of files in `binutils/bfd` that contain the word “`sparc`” and are possible candidate files for the AJIT port.

bfd/elf-bfd.h	01	BFD back-end data structures for ELF files.	yes
bfd/elf.c	02	ELF executable support for BFD.	yes
bfd/elfcode.h	03	ELF executable support for BFD. External to internal conversions.	yes
bfd/elfxx-sparc.c	04	SPARC-specific support for ELF.	yes
bfd/elfxx-sparc.h	05	SPARC ELF specific backend routines.	yes
bfd/libbfd.h	06	Declarations used by bfd library; generated file, see source files	yes
bfd/targets.c	07	Generic target-file-type support for the BFD library.	yes
bfd/reloc.c	08	BFD support for handling relocation entries.	yes
bfd/elf32-sparc.c	09	SPARC-specific support for 32-bit ELF.	yes
bfd/cpu-sparc.c	10	BFD support for the SPARC architecture.	yes
bfd/archures.c	11	BFD library support routines for architectures.	yes
bfd/config.bfd	12	Convert a canonical host type into a BFD host type.	yes
bfd/bfd-in2.h	13	Main header file for the bfd library: portable & generated	yes
bfd/bfd-in.h	14	Main header file for the bfd library: portable	yes
bfd/bfd.m4	21	This file was derived from acinclude.m4. **To Check**	maybe
bfd/configure.in	22	Process this file with autoconf to produce a configure script.	maybe
bfd/Makefile.am	22	Process this file with automake to generate Makefile.in	no
bfd/elf64-ajit.c	22	SPARC-specific support for 64-bit ELF - to DENY 64 bit support	maybe
bfd/nlm32-sparc.c	22	Support for 32-bit SPARC NLM (NetWare Loadable Module)	no
bfd/freebsd.h	23	BFD back-end definitions used by all FreeBSD targets.	no
bfd/configure	23	Guess values for system-dependent variables and	no
bfd/Makefile.in	23	Makefile.in generated by automake 1.11.1 from Makefile.am.	no
bfd/elf64-sparc.c	23	SPARC-specific support for 64-bit ELF	no

Table 4.5: List of files in `binutils/bfd` that contain the word "`sparc`" and that are possible candidate files for the AJIT port along with their number in the sequence of modifications.