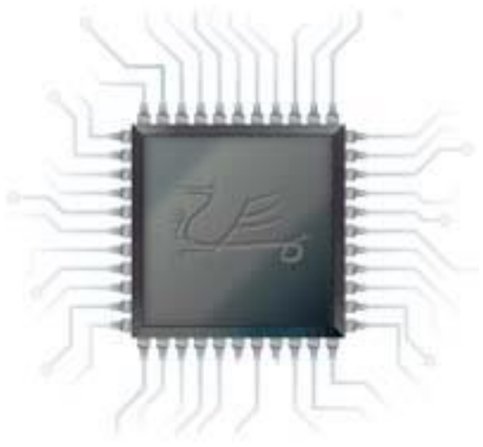
A Sparc v8 Instruction Set Simulator (sparc_iss)



Simon Southwell

July 2010
(last updated 28th August 2024)

Simon Southwell (simon@anita-simulators.org.uk)

Cambridge, UK, August 2024

Contents

INTRODUCTION **4**

SPARC Instruction Set Format 5

ARCHITECTURE FUNDAMENTALS **5**

Instruction Table Organisation 6
Instruction Decode Structure 7
Instruction Function Declarations 8

TOP LEVEL STRUCTURE **9**

INSTRUCTION FETCH **10**

DECODE AND REGISTER READ **11**

EXECUTE **12**

Arithmetic Example 12
Memory Access Example 13

WRITE BACK **13**

TRAPS **14**

OTHER FUNCTIONS **14**

GLOBAL STATE **15**

Processor State 15
User Control and Miscellaneous State 16

APPENDICES **16**

Model Limitations 16
Unsupported 17
Supported Instructions 17
Supported synthetic instructions 18
Download 18
Useful links 18

Introduction

In this document you can find links to obtain an instruction set simulator (ISS) for the Sparc processor version 8 instructions. Executables exist for Linux and windows, but source code is also provided with a `makefile` for Linux, and `.sln` file for windows Microsoft Visual Studio C++ environments, which can be used to recompile on a local machine.

The usage message of the model is as follows:

```
Usage: sparc_iss [-d] [-n <num instructions>] [-b <breakpoint addr>] \
                 [-f <filename>] [-o <filename>]


      -d Turn on Verbose display
      -n Specify number of instructions to run (default: run until UNIMP)
      -b Specify address for breakpoint
      -f Specify executable ELF file (default main.out)
      -o Output file for Verbose data (default stdout)
```

The model reads an executable ELF format file and starts executing. Simple break control is provided by specifying a number of instructions to run or an absolute address to break on, and verbose output can be turned on to follow execution.

This SPARC instruction set simulator was originally coded to capture some of the experience and knowledge gained on a much more complicated, and commercially marketed model—namely the Tricore 2 embedded processor of Infineon. The aim was to have a simpler model, that could be easily understood, with the basic architecture encapsulated, but minus the somewhat complicated additional requirements required of a commercial product, such as an interface to a debug API.

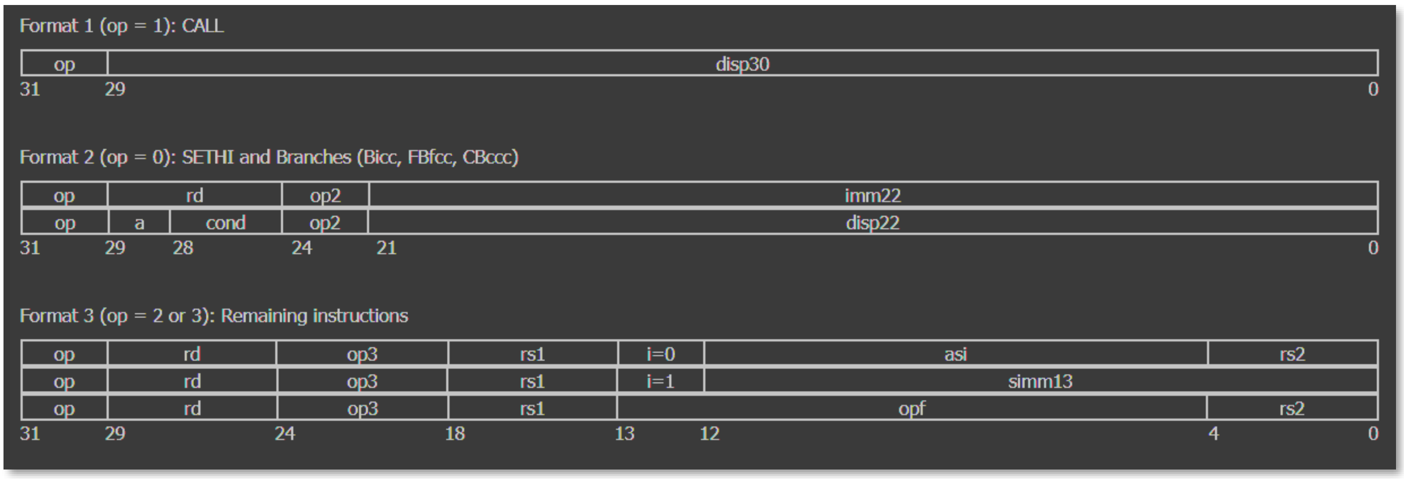
The SPARC was chosen partly because I had some previous experience of the processor, but also because of its simplicity. Whereas the Tricore 2 processor has over 800 instructions and 7 addressing modes, the SPARC v8 has just 68 instructions and 2 addressing modes. This means the code is not obfuscated with complexity which would hide the fundamental architecture principles. It was also hoped that the model of a SPARC processor might still be useful and of relevance for a little while after the publication.

The model architecture emphasizes structure, simplicity and maintainability, as well as ease of expandability. Speed is not of primary importance, though optimisations are made where possible. The original Tricore model usage was varied, being used both as a verification tool for the development of the processor, but also bundled with a debug tool chain, to allow software engineers to have a working demo, both prior to and after delivery of the Tricore hardware. The demonstration model was for evaluating the instructions rather than running software at high speed.

The model source code is released under the terms of the GNU general public license v3 so that you may access and modify the code for yourselves. Each instruction has been tested to some degree, but not rigorously so, and the model comes with absolutely no guarantees whatsoever. Further testing is ongoing, and subject to time available against other projects.

SPARC Instruction Set Format

This document is not meant to be a tutorial on the SPARC v8 instruction set, and the links provided will point you in the right direction to study this further. However, the instruction set does map on to the architecture chosen, and so a brief look at the instruction format is appropriate at this point. Below is shown the basic instruction format for the SPARC v8 processor.



As you can see, the SPARC has only three basic formats (with type 2 and 3 lumped together, as they are similarly structured). Format 1 has a lone instruction, `CALL`, with most of the bits used for the call's displacement address. Format 2 instructions are for the `SETHI` instruction (used in constructing 32-bit values in a destination register `rd`) and branches. The `op2` identifies the particular instruction, and for branches the `rd` (destination register) field of `SETHI` is replaced by the `a` (annul) and `cond` (condition) fields.

All other instructions are Format 3 instructions, and the op can be either 2 or 3. In general load/store type instructions have an `op` value of 3, whilst the rest (mostly logical and arithmetic instructions) have a value of 2. Common to all instructions of this type are a destination register `rd` and a primary source register `rs1`, as well as an `op3` field to identify the particular instruction. If the `i` field is clear (or a floating point/co-processor instruction) a secondary source register is specified (`rs2`). Otherwise an immediate field (`simm13`) is given. The floating-point co-processor instructions have their own field (`opf`) for specifying the co-processor instruction.

So that is all there is to the SPARC instruction set. Straight forward and will dictate the first architectural decision to be made.

Architecture Fundamentals

The model will be a straightforward fetch-decode-execute-writeback process, with no pipelining of instructions.

The approach taken in the SPARC model for decoding is to have a lookup table for each instruction of the three formats. The tables will consist of an array of pointers to functions, where the function is that needed to execute the operation. The `op` field will be used to select which table to use, and (apart from the single instruction format 1 table), the secondary op field (op2 or op3 as appropriate) indexes into the array to select the function. A function may appear in multiple places if it is a generic enough to cover multiple instructions.

Each function has an identical prototype (necessarily) where it has a single argument which is a decode structure containing all the decode information so far. Indeed this structure is like a form to be filled in as we go, where initially this is just the raw opcode (during fetch), then the decode phase looks up the execution function, splits out the individual fields and reads any source registers—placing them all in the structure. The execution phase calls the looked-up function, passing in the structure, and updates the final fields with any writeback data values and destinations, flags statuses and program counter updates. Finally, the model state is updated with these values during writeback.

Instruction Table Organisation

Below is shown the three lookup tables for the three instruction formats. The declaration of these arrays can be found in the `execute.c` source code file.

Format 1 only has 1 instruction, and so is not an array. The other two are for formats 2 and 3. Format 2 has a secondary op field (op2) of 3 bits, and so the array has 8 entries. Not all opcode values have a valid instruction, and so an UNIMP function is placed in these positions. Format 3 has a 7-bit secondary op field (op3) and so the array is 128 entries.

```
 // Format 1 instruction
static p_func const format1 = CALL;

// Format 2 instructions
static const p_func format2[8] = {
    UNIMP     , UNIMP     , BICC      , UNIMP,
    SETHI     , UNIMP     , FBFCC     , CBCCC};

// Format 3 instructions
static const p_func format3[128] = {
    ADD       , AND       , OR        , XOR       ,
    SUB       , ANDN      , ORN       , XNOR      ,
    ADDX      , UNIMP     , UMUL      , SMUL      ,
    SUBX      , UNIMP     , UDIV      , SDIV      ,
    ADDCC     , ANDCC     , ORCC      , XORCC     ,
    SUBCC     , ANDNCC    , ORNCC     , XNORCC    ,
    ADDXCC    , UNIMP     , UMULCC    , SMULCC    ,
    SUBXCC    , UNIMP     , UDIVCC    , SDIVCC    ,
    TADDCC    , TSUBCC    , TADDCCTV  , TSUBCCTV  ,
    MULSCC    , SLL       , SRL       , SRA       ,
    RDY       , RDPSR     , RDWIM     , RDTBR     ,
    UNIMP     , UNIMP     , UNIMP     , UNIMP     ,
    WRY       , WRPSR     , WRWIM     , WRTBR     ,
    FPOP1     , FPOP2     , CPOP1     , CPOP2     ,
    JMPL      , RETT      , TICC      , FLUSH     ,
    SAVE      , RESTORE   , UNIMP     , UNIMP     ,
    LD        , LDUB      , LDUH      , LDD       ,
    ST        , STB       , STH       , STD       ,
    UNIMP     , LDSB      , LDSH      , UNIMP     ,
    UNIMP     , LDSTUB    , UNIMP     , SWAP      ,
    LDA       , LDUBA     , LDUHA     , LDDA      ,
    STA      , STBA     , STHA     , STDA     ,
    UNIMP    , LDSBA    , LDSHA    , UNIMP    ,
    UNIMP    , LDSTUBA  , UNIMP    , SWAPA    ,
    LDF      , LDFSR    , UNIMP    , LDDF     ,
    STF      , STFSR    , STDFQ    , STDF     ,
    UNIMP    , UNIMP    , UNIMP    , UNIMP    ,
    UNIMP    , UNIMP    , UNIMP    , UNIMP    ,
    LDC      , LDCSR    , UNIMP    , LDDC     ,
    STC      , STCSR    , STDCQ    , STDC     ,
    UNIMP    , UNIMP    , UNIMP    , UNIMP    ,
    UNIMP    , UNIMP    , UNIMP    , UNIMP
};
```

In all cases, the array type is `p_func`, which is a pointer to function type, whose prototype is defined as:

```
  typedef void (*p_func) (pDecode_t);
```

So, all the execution functions return void, and have a single argument of type `pDecode_t`, which is a pointer to the "form" to be filled in during instruction processing. More of this below. The single argument has enough information for each execution function to perform the operation (though it may do some further decode itself, if it is a generic function, servicing multiple instructions). The execution functions will update further fields in the structure, ready for the following steps.

Note that the array entries shown in red are not supported by this model. These instructions are associated with the floating point unit or co-processor, or are for the alternate space, all of which are optional. In this model, all these functions are mapped to `UNIMP`, but the table has labels for these instructions to allow ease of adding support for them in the future.

Instruction Decode Structure

The decode structure definition is shown below. The definition of this structure is to be found in `sparc.h`.

```
     // Defer structure definition
    struct  DecodeStruct;

    typedef struct DecodeStruct *pDecode_t;
    typedef void (*p_func) (pDecode_t);


    struct DecodeStruct {
        uint32 opcode;             // Instruction opcode
        p_func function;           // Pointer to inst.c function
        uint32 rd;                 // rd register number
        uint32 rs1;                // rs1 register number
        uint32 rs1_value;          // rs1 register value
        uint32 imm_disp_rs2;       // Immediate, displacement or rs2
        uint32 ev;                 // Extended value (ev) = r[rs1] + (i ? sign_ext(imm) : r[rs2])
        uint32 op_2_3;             // op2 or op3 value
        uint32 i;                  // i, imm/rs2 indicator

        uint32 PC;                 // Current program counter value
        uint32 nPC;                // Next program counter value
        uint32 PSR;                // Program status register value
        pPSR_t p;                  // Pointer to structured version of PSR
        uint32 wb_type;            // Write back flag
        uint32 value;              // Write back value
        uint32 value1;              // Second (optional) write back value
    };
```

On the whole, the field order as shown below is the order in which the fields will be updated, as the instruction is processed. So, the opcode field is updated first, during fetch, and then decode looks up the execution function from the tables and places its pointer in function, as well as splitting up the sub-fields into the constituent parts, such as `rd`—the destination register number. Note that some of the structure fields for this are shared amongst the instructions. For instance, the `imm_disp_rs2` field can hold an immediate value (e.g. for `SETHI`) or a displacement value (e.g. for a branch instruction) or indeed the value of the secondary source register (`rs2`). As it is context dependent, the execution functions interpret this field appropriately to the instruction it is processing. All the fields from `function` to `i` are updated (if necessary) during decode.

During execution, the selected function will update the "program counter" and "next program counter" fields (`PC` and `nPC`). The status register value is also recorded (`PSR`), but a structured version pointer (`p`) is also mapped over this value for ease of further testing and interpreting the individual bits. If a write back operation is required from the execution phase, this is flagged in `wb_type`, and the value to be written recorded in value. Occasionally an additional writeback value is to be written, and this will be placed in value1.

The writeback phase takes all the data from the second part of the structure (from `PC` to `value1`), which will by now be completely filled in, and updates the model's state accordingly. The instruction processing is complete, and a new instruction can be executed.

Instruction Function Declarations

For reference, the definitions of all the execution functions are shown below. The prototypes for these instructions are to be found in `inst.h`, with the actual functions in `inst.c`.

```
extern void UNIMP   (pDecode_t d);
extern void CALL    (pDecode_t d);
extern void BICC    (pDecode_t d);
extern void SETHI   (pDecode_t d);
extern void SLL     (pDecode_t d);
extern void SRL     (pDecode_t d);
extern void SRA     (pDecode_t d);
extern void RDY     (pDecode_t d);
extern void RDPSR   (pDecode_t d);
extern void RDWIM   (pDecode_t d);
extern void RDTBR   (pDecode_t d);
extern void WRY     (pDecode_t d);
extern void WRPSR   (pDecode_t d);
extern void WRWIM   (pDecode_t d);
extern void WRTBR   (pDecode_t d);
extern void JMPL    (pDecode_t d);
extern void RETT    (pDecode_t d);
extern void TICC    (pDecode_t d);
extern void SAVE    (pDecode_t d);
extern void RESTORE (pDecode_t d);
extern void FLUSH   (pDecode_t d);
extern void MULSCC  (pDecode_t d);
extern void LD      (pDecode_t d);
extern void LDUB    (pDecode_t d);
extern void LDUH    (pDecode_t d);
extern void LDD     (pDecode_t d);
extern void LDSB    (pDecode_t d);
extern void LDSH    (pDecode_t d);
extern void ST      (pDecode_t d);
extern void STB     (pDecode_t d);
extern void STH     (pDecode_t d);
extern void STD     (pDecode_t d);
extern void SWAP    (pDecode_t d);
extern void LDSTUB  (pDecode_t d);
extern void MUL     (pDecode_t d);
extern void DIV     (pDecode_t d);
extern void ADD     (pDecode_t d);
extern void AND     (pDecode_t d);
```

As mentioned before all of these functions have identical prototypes, compatible with the `p_func` type used in the lookup tables, where these functions are referenced (see above). Some of the functions appear multiple times within the lookup tables, to service similar instruction operations, such as `MUL`. In these cases, the function listed in the lookup tables will be the actual function name for that decode position, but this will be mapped to the generic function.

Top Level Structure

Some pseudo-code is shown below representing the structure of the top level model code, with the function call hierarchy shown. Functions might be optionally called from a higher level function, and these are shown between brackets. Where functionality exists within the function level, these are summarised between < and > symbols. The actual source code for the top level functionality is found in `sparc_iss.c` for the `main()` function, and in `execute.c` for `Run`.

```
main() {
  ReadElf (fname)
    LoadMemWord()

  Run (ExecCount)
    WHILE not terminate and not executed ExecCount instructions
      <process interrupts>
      <process traps>
        [ WriteReg() ]
      <process breakpoints>

      Ifetch(PC,   &(d->opcode));                // FETCH
        [ RegisterDump() ]
      Decode (d);                                // DECODE
        [ ReadReg() ]
      d->function(d);                            // EXECUTE
        [ Trap() ]
        [ MemRead() ]
        [ WriteRegAll() ]
           [ GetRegBase() ]
        [ MemWrite() ]
           [ ReadRegsAll() ]
           [ GetRegBase() ]
        [ TestCC() ]
        [ RegisterDump() ]
           [ ReadReg() ]
```

```
              [ DispReadReg() ]
          WriteBack{d);                              // WRITEBACK
            WriteReg{d->value, d->rd)
        END WHILE
    }
```

From the `main()` top level, program code is first read from an ELF file into memory via `ReadElf()`. A function `Run()` is then called to execute the code until completed or reaches a termination point (such as a breakpoint). A while loop processes the instructions, one at a time, until finished, and then the basic fetch-decode-execute-writeback steps are coded. `Ifetch()` implements the first step, with a call to `Decode()` the next. The execute phase is a call to the previously looked up function pointer. Within the execute function, various sub-functions might be called for handling exceptions (`Trap()`), memory accesses (e.g. `MemRead`), register accesses (e.g. `ReadReg`()) or testing status bits (`TestCC`). Some of these functions also make reference to further sub-functions, and these are also indicated.

As the main loop processes instructions, these might generate exceptions (traps), hit breakpoints or generate interrupts. These are all tested for within `Run` before starting the next instruction's fetch cycle. This would normally set some status bits of the `PSR` and then update the `PC` to point to the trap address. The subsequent fetch then starts executing from trap code, rather than the next expected instruction address.

Instruction Fetch

The fetch functionality is basically a memory read, and the function does in fact live with the other memory access functions in the `read_write.c` file. However, the reading of instructions has some restrictions, and additional checks are made.

```
    static uint32 Memory [1 << (MEM_SIZE_BITS << 2];

    void Ifetch (const uint64 physaddr, uint32 * const inst)
    {
        uint64 PA = physaddr & ADDR_MASK;

        // Misaligned instruction fetch
        if ((PA & (uint64)3) != 0) {
            RegisterDump();
            terminate = 1;
            return;
        }

        // Trying to read instructios out of range
        if ((PA & ~ADDR_MASK) != 0) {
            RegisterDump();
            terminate = 1;
            return;
        }

        *inst = Memory[PA >> (uint64)2];
    }
```

Shown above is the fetch function, along with the declaration of the processor memory array. As the memory size is configurable with macro definitions, the address passed into the function is masked with a predefined macro `ADDR_MASK`, before being checked for 32 bit alignment and address range. As both these are fatal errors, a register dump is performed before returning with the terminate flag set. If the address is valid, then a memory read is performed at the masked address, and the resulting value returned in the pointer `*inst`.

Decode and Register Read

The decode function is shown below. Its main purpose is to perform the lookup of the execution function from one of the 3 lookup tables (see above) and fill in the top half of the decode structure by extracting bits from the opcode and (optionally) reading register values. This function is found in the `execute.c` source file.

```
static void Decode(pDecode_t d)
{
    uint32 fmt_bits = (d->opcode >> FMTSTARTBIT) & LOBITS2;
    uint32 op2      = (d->opcode >> OP2STARTBIT) & LOBITS3;
    uint32 op3      = (d->opcode >> OP3STARTBIT) & LOBITS6;
    uint32 I_idx, regvalue;

    d->PC      = PC;
    d->nPC     = nPC;
    d->PSR     = PSR;
    d->wb_type = NO_WRITEBACK;

    switch (fmt_bits) {
    case 1:  // CALL
        d->function     = format1;
        d->imm_disp_rs2 = d->opcode & LOBITS30;
        break;
    case 0:  // SETHI, Branches
        d->function     = format2[op2];
        d->rd           = (d->opcode >> RDSTARTBIT) & LOBITS5;
        d->op_2_3       = (d->opcode >> OP2STARTBIT) & LOBITS3;
        d->imm_disp_rs2 = (d->opcode & LOBITS22);
        break;
    case 3:  // Memory accesses, ALU etc.
    case 2:
        d->function     = format3[op3 + ((fmt_bits & 1) << 6)];
        d->rd           = (d->opcode >> RDSTARTBIT)  & LOBITS5;
        d->op_2_3       = (d->opcode >> OP3STARTBIT) & LOBITS6;
        d->rs1          = (d->opcode >> RS1STARTBIT) & LOBITS5;
        d->i            = (d->opcode >> ISTARTBIT)   & LOBITS1;
        d->imm_disp_rs2 = (d->opcode >> RS2STARTBIT) & LOBITS13;

        ReadReg (d->rs1, &d->rs1_value);

        I_idx = op3 + ((fmt_bits & 1) << 6);
        regvalue = (I_idx < FIRST_RS1_EVAL_IDX) ? 0 : d->rs1_value;

        if (d->i)
           d->ev = regvalue + sign_ext13(d->imm_disp_rs2);
        else {
           ReadReg(d->imm_disp_rs2 & LOBITS5, &d->ev);
           d->ev += regvalue;
        }

        if (I_idx == STORE_DBL_IDX) {
            ReadReg (d->rd & ~LOBITS1, &d->value);
            ReadReg ((d->rd & ~LOBITS1)+1, &d->value1);
        } else if (I_idx == TICC_IDX && !d->i)
            ReadReg (d->imm_disp_rs2 & LOBITS5, &d->value);
        else if (fmt_bits & 1)
            ReadReg (d->rd, &d->value);
        break;
    }
}
```

Firstly, the decode structure is updated with copies of the status register and PC (and next PC). The main format type decode is then done with a case statement. For format 1 and 2 instructions (with `fmt_bits` values of 1 and 0) it is just a matter of extracting bits from the opcode fields and placing in the decode structure, and the execute function table lookup. For format 2 instructions (`fmt_bits` 2 and 3) some additional functionality is required. All format 2 instructions need to read at least 1 source register, but depending on the other opcode fields and additional source registers may need to be read, and the rest of the code implements this.

Execute

The execution functions all reside in `inst.c`, with one for each of the functions in the decode tables shown in a previous section. There are too many to discuss in detail here, and we will only look at a couple of examples.

Arithmetic Example

Most of the arithmetic and logic instructions (all of which are format 3 instructions with an op value of 2) have functions who generate a writeback value and update the status register. A typical function is shown below for multiplication.

```
void MUL (pDecode_t d)
{
    uint64 x, y, z;
    uint32 cc;

    cc = (d->PSR >> PSR_CC_CARRY) & LOBITS4;
    x = d->rs1_value;
    y = d->ev;

    // Sign extend for SMUL/SMULCC
    if (d->op_2_3 & LOBITS1) {
        x |= (x & 0x80000000) ? ((uint64)0xffffffff << (uint64)32) : 0;
        y |= (y & 0x80000000) ? ((uint64)0xffffffff << (uint64)32) : 0;
        z = (int64)x * (int64)y;  // actual multiplication (signed)
    } else
        z = x * y;                // actual multiplication (unsigned)

    Y = (uint32)(z >> 32);        // Y = hi 32 bits
    z &= 0xffffffff;              // z = lo 32 bits

    // UMULCC or SMULCC
    if (d->op_2_3 & BIT4) {
        cc = (cc & ~(1 << CC_ZERO))     | (((z == 0) ? 1 : 0) << CC_ZERO);
        cc = (cc & ~(1 << CC_NEGATIVE)) | (((uint32)(z >> 31) & 1) << CC_NEGATIVE);
        cc &= ~(1 << CC_OVERFLOW);
        cc &= ~(1 << CC_CARRY);
        d->PSR = (d->PSR & ~(0xf << PSR_CC_CARRY)) | (cc << PSR_CC_CARRY);
    }

    d->wb_type = WRITEBACKREG;
    d->value = (uint32)z;
    d->PC = d->nPC;
    d->nPC += 4;
}
```

In this function, the carry bits are extracted from the status register into `cc`, and the two operands into `x` and `y` from values already extracted during decode. This function is a generic function for signed and unsigned multiplication, as well as for multiplication with and without carry. Therefore, some further decode needs to be performed from values already in the decode structure to select the appropriate operation.

Firstly, if a signed operation is required, `x` and `y` are sign-extended before a signed multiply, otherwise a simple multiply is performed. Then, the `cc` carry status bits are only updated for `CC` variants of the instruction, and the status register, `PSR`, updated. The result is always written back, and so the write back flag is set, and the result in `z` placed in the value field. The program counter fields are then updated with a simple increment.

Instructions like this, you might note, do not generate any kind of trap or exception. If the instruction is legal (and it will be if we got this far) then a result will definitely be generated. This isn't true for instructions such as memory accesses, so we'll look at an example next.

Memory Access Example

The memory access instruction example we'll look at is the load double word (`LDD`) instruction. It is very similar in structure to the `Ifetch` function we saw earlier. It makes some checks on its arguments, and throws a trap if bad, otherwise a memory read is performed.

```
void LDD (pDecode_t d)
{
    if (d->rd & LOBITS1) {
        Trap(d, SPARC_ILLEGAL_INSTRUCTION);
    } else if (d->ev & LOBITS3) {
        Trap(d, SPARC_MEMORY_ADDR_NOT_ALIGNED);
    } else {
        MemRead(d->ev, 8, d->rd & ~LOBITS1, 0);
        d->PC = d->nPC;
        d->nPC += 4;
    }
}
```

The first check is for destination register alignment. As this is a double word load, then the destination register number must be even. Similarly, the address for the read must be 32 bit aligned. If either of these checks fails, a call to `Trap` is made. If no exception is thrown, then a memory read is performed, placing the read value directly into register selected by `rd`. The program counters are updated incrementally in the decode structure, ready for a writeback update.

Write Back

The write back function is perhaps the simplest of them all. In all cases the model's program counters and status registers are updated with the values from the decode structure. If the `writeback` flag is set, the destination register is also updated, via a call to `WriteReg()`.

```
static void WriteBack (const pDecode_t d)
{
    PC  = d->PC;
    nPC = d->nPC;
    PSR = d->PSR;

    if (d->wb_type == WRITEBACKREG)
        WriteReg(d->value, d->rd);
}
```

The `WriteBack()` function is located in the `execute.c` source code file.

Traps

The trap function does not itself cause the program flow to change. Instead, it is called with a trap type from the various execution functions etc. and updates some global state (i.e. `TrapType` that is processed at the top level, as we saw earlier).

Traps can be disabled with a bit in the status register, so if `Trap()` is called with this bit set, then this is fatal. In that case a register dump is performed, and the `terminate` flag set.

The `Trap()` function is found in the `inst.c` file, along with all the execution functions, mainly because it is mostly called due to exceptions generated from these functions.

```
static void Trap (pDecode_t d, uint32 trap_no)
{
    int tn = trap_no & LOBITS8;

    // Trapped whilst traps disabled
    if (d->p->et == 0) {
         RegisterDump();
         terminate = d->opcode;
         return;
    }
    TrapType = (TrapType & ~(LOBITS8)) | tn;
}
```

Other Functions

The above code forms the core of the functionality for the model, but mentioned before are various other functions to support this code. Broadly speaking, these fall in to one of 3 categories; namely getting global state (`PC`, `PSR` etc.), memory accesses and register accesses. There are a couple of debug oriented functions (`RegisterDump()`, which uses `DispReadReg()`, and `DispDecode()`), which are fairly self-explanatory, and don't need documenting here.

There are four functions for retrieving state: `GetPC()`, `GetnPC()`, `GetPSR()` and `GetIRL()`. These are simple data hiding modules and return the raw value of the particular state. They are located in `execute.c`.

The memory access functions (in `read_write.c`) consist of `LoadMemWord()` (only used to load program data to memory), `MemRead()` and `MemWrite()`. These last two don't return a value (for reads), or take a value (for writes), but simply move data directly from memory to a register or vice versa.

For register accesses we have `ReadReg()` and `WriteReg()`, both located in `read_write.c`. Unlike the memory functions, these do return a value (for reads) and use an input value (for writes). These functions are aware of the register windowing and virtualise this away. Some functions which treat all the registers as a flat space exist and are called from various places where appropriate. These are `WriteRegAll()` and `ReadRegAll()`. The offset within this global register space is fetched with `GetRegBase()`.

There is one other support function in `inst.c` that I should mention. `TestCC()` is used by the branch instruction functions (`BICC()` and `TICC()`) to determine whether to branch or not. It tests various status register bits depending on the instruction.

The program execution data is loaded into the SPARC memory array using a function `ReadElf()`, located in the `elf.c` source code file. It takes a single file name argument where it expects to find a SPARC v8 executable ELF file. We'll not look at the details of this function here, but information about the ELF format can be found via the links section below.

Global State

This SPARC model does use global state to implement its functionality. Normally this state would be wrapped in data hiding functions, with checking and limited access etc. but, as mentioned in the introduction, this particular model is meant to be as simple and easy to understand as possible, and the amount of global state, and its simplicity, means that the more rigorous approach would only serve to complicate the source code. I will briefly describe the major state variables here to aid navigation of the code's functionality

Processor State

The SPARC processor's major registers are all mapped onto global variables, though some are limited in scope via static declarations. The main control and state registers are declared in `execute.c`:

- `PC` Program Counter
- `nPC` Next Program Counter
- `PSR` Processor Status Register
- `IRL` Interrupt Level

These variables are all declared static, and so are limited in scope to functions in `execute.c`. Additional globals map to processor registers in `inst.c`.

- `TBR` Trap Base Register
- `WIM` Window Invalid Mask
- `Y` Multiply/Divide Register

The `TBR` and `WIM` variables are declared fully global, but the `Y` variable is limited in scope to just `inst.c`. All these global variables implement registers with the SPARC processor. The function of these registers is not documented here, and for those interested in following this up more, section 4.2 of the SPARC Architecture Manual v8 explains their functionality. A link to this document can be found in the Useful Links section.

Some additional state of the processor system is also mapped to global variable. Unlike the above state, these aren't mapped to registers directly visible to the instruction set, but still have direct counter parts in the processor system hardware. These are:

• `Globals[]`	Global Registers
• `Locals[]`	Local Registers
• `Outs[]`	Out (and In) Registers
• `Memory[]`	System memory

The first three of the above list are for implementing the windowed registers of the SPARC (see section 4.1 of the architecture manual). It is not a direct mapping as the `Outs[]` doubles for both in the "out" and "in" registers. The `Memory[]` implements the accessible memory for executable and program data.

User Control and Miscellaneous State

The user command line options affect control within the model, and global variables are used to communicate this control.

• `Verbose`	Controls verbosity
• `NumRunInst`	Number of instructions to run before breakpoint
• `BreakPoint`	Address of breakpoint
• `*ofp`	Pointer to user specified verbosity output file

There are some final global variables worth mentioning here for completeness:

• `terminate`	Flags main loop to terminate
• `TrapType`	Flags main loop that a trap has occured
• `ProcNo`	An ID number for this particular processor instantiation

The first two variables are used to communicate to the top-level code. The terminate variable is set by various sources, for fatal exceptions etc. The `TrapType` variable is set by non-fatal exceptions which are handled as traps or interrupts. The last variable `ProcNo` isn't actually used as yet but is a hook for adding functionality should the model be used in an environment where multiple instantiations of the model are required.

Appendices

Model Limitations

This model will allow arbitrarily compiled code to be executed from, say, a gcc SPARC cross-compiler, but some instructions, alluded to above, are not supported. These are for the alternate space, the floating point unit, and for the co-processor interface. The exact instructions not supported are listed below.

In addition, the model does not currently have a memory management unit (MMU), and all addresses are physical. This is left as an exercise for the student. In `Run()` (in `execute.c`) there is an assignment `physaddr = (uint64) PC;`. This is where an MMU function should be inserted. Also, the memory accesses are not put through a model of a cache. Caching is invisible to the instructions, and as this is an ISS, it was not deemed necessary for this simple model. However, the original Tricore 2 model had both an MMU and cache model.

Unsupported

All the instructions that are not supported are not core instructions but associated with the alternate space support or the floating point co-processor or generic co-processing port.

```
  // Alternate space instructions
  LDSBA LDSHA LDUBA LDUHA LDA LDDA
  STBA STHA STA STDA
  LDSTUBA SWAPA
  RDASR WRASR

  // Floating point unit
  LDF LDDF LDFSR
  STF STDF STFSR STDFQ
  FBfcc
  FPop

  // Coprocessor
  LDC LDDC LDCSR
  STC STDC STCSR STDCQ
  CBccc
  CPop
```

Supported Instructions

Below are listed the instructions supported by the ISS. This contains all the core instructions, and a SPARC compiler would generate code which is compatible with this list.

```
  Load:           LDSB LDSH LDUB LDUH LD LDD
  Store:          STB STH ST STD
  Atomic:         SWAP LDSTUB
  Misc:           SETHI NOP
  Logical*:       AND ANDcc ANDN ANDNcc OR ORcc ORN ORNcc XOR XORcc XNOR XNORcc
  Shift:          SLL SRL SRA
  Add**:          ADD ADDcc ADDX ADDXcc
  Subtract**:     SUB SUBcc SUBX SUBXcc
  Multiply***:    UMUL UMULcc SMUL SMULcc
  Divide***:      UDIV UDIVcc SDIV SDIVcc
  Tagged add**:   TADDcc TADDcc TSUBccTV TSUBccTV
  Multiply Step: MULScc
  Window:         SAVE RESTORE
  Branch:         Bicc
  Prog control:  CALL JMPL
  Trap:           RETT Ticc
  Read Regs:      RDY RDPSR RDWIM RDTBR
  Write Regs:     WRY WRPSR WRWIM WRTBR
  Mem Sync:       STBAR FLUSH
  Unimplemented: UNIMP

NOTES:
  * The individual logical instruction have the N, cc, and Ncc
    operation performed in a common function which does a partial
    decode.

 ** All add, subtract and tagged add are implemented in a
    single function which does a partial decode.
```

```
*** All multiply and divide instructions (not MULScc) implement
    their cc versions in a common function which does a partial
    decode.
```

Supported synthetic instructions

Sparc assembly has a number of instructions which are not really separate instructions, but map (are synthesised) onto a core instruction. These are supported by the ISS and listed here.

```
  BTST BSET BCLR BTOG
  CLR CLRB CLRH CLR
  CMP
  DEC DECCC
  INC INCC
  JMP
  MOV
  NOT NEG
  SET
  TST
  SKIPZ SKIPNZ
```

Download

The model is released under version 3 of the GPL and comes with no warranties whatsoever. You can download the model and source code from github.

Useful links

- SPARC Assembly Language Reference Manual
- SPARC Architecture Manual version 8
- Executable and Linkable Format (ELF)
- Howto on building a SPARC cross-compiler