

# SPARC Architecture, Assembly Language Programming, and C, Richard P. Paul

## Chapter 1: The Computer

### Assembly Language:

- An assembly language program is a program with symbols representing numeric values. Translating an assembly language program into a machine language program involves looking up the symbols and mnemonics in a symbol table and substituting the matching numeric values.

### m4:

- m4 is a macro processor that can be used to translate symbols into numeric constants.
- With: `include(macros_defs.m)` at the beginning of an assembler file, you can import an m4 macro file.
- Example macro:  
`define(sto, 44 0)`  
`define(add, 40)`  
`define(rcl, '45 $1')`  
`rcl(0) sto 1 add ⇒ 45 0 44 0 1 40`

### The von Neumann Machine:

- The machine von Neumann helped to define consists of an addressable memory, capable of holding instructions and data, coupled with an arithmetic logic unit, capable of executing the instructions fetched from memory. The address for the next instruction to be executed was held in a register called the program counter.

### Java Virtual Machine:

- A JVM is an interpreter designed to execute Java bytecode which is generated through compiling the source code. Java bytecode is stack machine code, with each instruction filled in one byte ⇒ maximum of 256 instructions.

### Stack – architecture vs. Accumulator – architecture:

- It could be that the stack – machine needs less instructions than the accumulator machine, p.31, but the accumulator machine is much more regular, as every instruction has one operand, an address.
- The accumulator architecture is also referred to as a single – address machine.
- Although the program is shorter for the stack machine, the lack of regularity of the instructions might make decoding them more time consuming than for the accumulator machine. This is a fundamental question related to the design of computers – space versus time, p.33.

### Load/Store machines:

- The architecture for the load/store machine looks very similar to the architecture of the stack machine, with a register file replacing the stack. The main difference between the stack and the register file is that any of the register may be selected with each instruction, whereas only the top two elements of the stack may be accessed at any time.
- The SPARC is a load/store machine.

### Assembler:

- An Assembler is a macro processor specialized for translating symbolic programs into machine language programs. An assembler effectively reads a file twice, once to determine all the symbol definitions and the second time to apply those definitions and thus translate the symbolic text into numeric instructions and data.
- Similar to m4. Difference: In assembler the order of statements doesn't matter, because he goes twice through the text.

### Literals:

- A literal is a constant appearing directly in a machine instruction instead of its address. The *mov* instruction will load a constant directly into a register instead of fetching the constant from memory.
- A literal constant on the SPARC must have the range  $-4096 \leq c < 4096$ .

## Chapter 2: SPARC Architecture

### General Information about the SPARC:

- A SPARC has 32 registers available to the programmer at any time. These are divided into 4 sets: global, in, local, out, p.52
- %g0 is a special register, always returning a zero when read and discarding whatever is written to it.
- The SPARC architecture does not have a multiply or divide instruction, because the SPARC is a “Reduced Instruction Set Computer” a RISC.
- In general, when we write assembly language programs we will be expected to fill all possible delay slots.
- When we write assembly language programs it is generally a good idea first to write the algorithm in a high level or at least a pseudo – high – level language.
- Note that it is not considered good assembly language programming practice to exchange the order of the “then” code with the “else” code to avoid complementing the test.

### GNU C Compiler gcc:

- `gcc -S program.c`  $\Rightarrow$  produces only the assembler file of program.c
- `gcc expr.s -o expr`  $\Rightarrow$  produces the executable assembler file expr
- `m4 expr.m > expr.s`  
`gcc expr.s -o expr`  $\Rightarrow$  if we want to include some macros

### Pipelining:

- The components of the von Neumann machine for a RISC architecture are, p57:
  1. Instruction fetch
  2. Execute
  3. Memory Access
  4. Store Result
- There are two problems which can occur when a machine is pipelined. The first relates to load instructions, the second to branches, p.58. (use the delay slots appropriately)

### Load delay slot:

- The load delay slot occurs after a load instruction. The machine waits automatically if a problem would occur (for example, if the variable which is loaded should be used directly afterwards). But a better solution (instead of “waiting”) would be to fill the delay slot with a useful instruction.

### Branch delay slot:

- The branch delay slot occurs after a branch instruction. The machine doesn’t wait automatically, therefore the programmer has to fill in something useful (the worst would be a *nop* instruction).

### Annulled branches:

- All conditional branches may be annulled. If a conditional branch is annulled, the delay instruction is executed when the branch is taken (the usual case) but not if the branch is not taken.
- The *ba* unconditional branch instruction may also be annulled (but this isn’t a good practice). If an unconditional branch is annulled with *ba, a*, the delay slot instruction is not executed.

## Chapter 3: Digital Logic And Binary Numbers

### Memory Address:

- On actual machines memory addresses are typically much larger than 10 bits; 32 bits (binary digit) are quite popular, resulting in  $2^{32} = 4'294'967'296$  memory cells.

### Synthetic instructions:

- Alternative forms of instructions, or instructions with one operand always %g0, are called synthetic instructions:
- Exp: mov %r1, %r2  
⇒ or %g0, %r1, %r2 (that means mov is a synthetic instruction)

### Binary Numbers to Octal Numbers:

011010110110<sub>2</sub>  
⇒ 011 010 110 110<sub>2</sub>  
    ↓ ↓ ↓ ↓  
    3 2 6 6  
⇒ 3266<sub>8</sub> ⇒ 03266

### Binary Numbers to Hexadecimal Numbers:

011010110110<sub>2</sub>  
⇒ 0110 1011 0110<sub>2</sub>  
    ↓ ↓ ↓  
    6 b 6  
⇒ 6b6<sub>16</sub> ⇒ 0x6b6

### Octal Numbers to Binary Numbers:

566<sub>8</sub>  
⇒ 5 6 6  
    ↓ ↓ ↓  
    101 110 110<sub>2</sub>  
⇒ 101110110<sub>2</sub>

### Decimal Numbers to Binary Numbers:

2	374	
2	187+0	
2	93+1	
2	46+1	
2	23+0	
2	11+1	⇒ 101110110 <sub>2</sub>
2	5+1	
2	2+1	
2	1+0	
2	0+1	

### Decimal Numbers to Octal Numbers:

8	374	
8	46 + 6	$\Rightarrow 566_8$
8	5 + 6	
8	0 + 5	

## Chapter 4: Binary Arithmetic

### One's complement:

- All ones are replaced by zeros and all zeros are replaced by ones:  $10100 \Rightarrow 01011$

### Two's complement:

- It's one's complement and add 1:  $10100 \Rightarrow 01011 \Rightarrow 1 + 01011 = 01100$

### Addition and Subtraction with signed numbers:

- $\text{sum} = \text{xor}$ ,  $\text{carry} = \text{and}$
- The hardware always performs binary addition. If we wish to subtract, we first form the two's complement of the number and then we can just add the two numbers. If the MSB, the sign bit, is a one, we know that the number is negative (Pointers are always unsigned numbers, as they refer to a memory address).
- Example for negating a signed number:  $5 = 0101 \Rightarrow 1010 + 1 = 1011 = -5$
- Example for subtracting:  
 $3 - 5 = -2 \Rightarrow 3 + (-5) = -2$   
 $\Rightarrow 3 = 0011$   
 $5 = 0101$   
 $\Rightarrow \text{build two's complement of 5 (means negating it)} \Rightarrow 1010 + 1 = 1011$   
 $\Rightarrow 0011$   
 $+ \underline{1011}$   
 $1110$   
 $\Rightarrow 1110 \text{ must be a negative number (because of the leading 1), so let us complement it (retransformation):}$   
 $\Rightarrow 1110 \Rightarrow 1's: 0001 \Rightarrow 2's: 0010 = -2$

### Shifting:

- Arithmetic shift left = Logic shift left = zeros are shifted from the right
- Arithmetic shift right = the sign bit is copied into the MSB positions
- Logic shift right = zeros are copied into the MSB positions

### Unsigned Arithmetic:

- The same as signed arithmetic's, except, that the MSB is a 1, this is not a negative number and that's why no retransformation with computing two's complement has to be performed.

### %Y - register:

- The %Y - register initially holds the multiplier and eventually holds the low - order part of the product, p.131.
- Multiplicand x Multiplier = Product
- Although *umul* and *mul* produce the same low 32 bits of the product, *umul* results in fewer instructions. It is frequently used in C to multiply both signed and unsigned integers.

## Chapter 5: The Stack

<i>C Type</i>	<i>SPARC Type</i>	<i>Bits</i>	<i>Unsigned</i>	<i>Signed</i>
<i>char</i>	<i>byte</i>	8	0,255	-128,127
<i>short</i>	<i>half</i>	16	0,65'535	-32'768,32'767
<i>int,long</i>	<i>word</i>	32	$0,4.294 \cdot 10^9$	$\pm 2.147 \cdot 10^9$

### Alignment:

- To make the implementation of the architecture efficient, all memory references must be aligned. That is to say, 2 – byte quantities may only be addressed in memory with addresses that are divisible by 2, even addresses. 4 – byte quantities may be addressed in memory only if they are aligned on a 4 – byte boundary, that is, that the address is evenly divisible by 4. Similarly, an 8 – byte quantity must have a memory address evenly divisible by 8.

### The Stack:

- The address of the last occupied stack memory element is always kept in register %o6. This register is also known to the assembler as %sp, the stack pointer.
- The only reason that the space at the top of memory (means the highest addresses in memory) is called the stack is because it provides a first – in last – out data structure.
- The stack, located at top of memory grows downward. It is also possible to grow a program upward from the bottom of memory to provide for a heap, space that is not handled on a first – in last – out order. This allows for the largest possible size of program in a given memory space; when the stack and the heap overlap, the program can no longer execute.
- The stack is always kept doubleword (8 bytes) aligned. To ensure that the stack is doubleword aligned, the address in the stack pointer, %sp, must be evenly divisible by 8. If we want 94 bytes of stack memory space, we must ask for 96 to keep the stack aligned. This could be done as follows: %sp , -94 & -8 , %sp , this would result in 96 being subtracted from the stack pointer, because the 3 LSB are cut off, because -8 = 1111...11000 is valid.
- The SPARC architecture provides a second register, %i6, known as the frame pointer, %fp, into which is stored a copy of the stack pointer before it is changed to provide more storage. The frame pointer %fp points to what was the top of the stack before %sp was changed.
- The save instruction performs an addition and saves the contents of the stack pointer in %fp, the frame pointer. The save instruction is normally executed once at the beginning of a program to provide storage for all automatic variables (this are the variables with no register, extern and static before the type int, long, and so on. int x = 5, so x is a automatic variable). The save instruction is used to provide not only storage for automatic variables but also to provide space on the stack to save some of the registers. In Chapter 7 we describe why registers must be saved, but for now we need to provide 92 extra bytes of storage whenever executing a save instruction:
  - save %sp , -92 – (bytes\_of\_local\_storage) , %sp
  - Suppose, for example, that we wished to store five variables, a0 – a4, on the stack instead of in the registers. We would first need to make room at the beginning of our program by:
  - save %sp , (-92 –(5 \* 4)) & -8 , %sp (& -8 is needed for doubleword alignment)
  - To access the stack we use now the frame pointer, %fp, because the stack pointer points now to another location.
  - The content of the first variable a0, will now be at address: %fp – 4  
( - 4 because a0 was a register and this is a register and a register is a word, which has 4 bytes)
- Stack variables have to be initialised by loading the constant into a register and then storing it on the stack.
  - mov 7 , %o0
  - st %o0 , [%fp – 4]

### To write faster code (doesn't matter with good compilers):

- Tests against zero are better in loops, as they generally eliminates the need for a separate *cmp* instruction.
- The use of pointers is generally better than an indexing as it avoids computing an index expression, especially when stepping through an array element by element.

## Chapter 6: Data Structures

### Address Arithmetic:

- In general, a  $n$  – dimensional array requires  $n$  adds and  $n - 1$  multiplies to compute an address of an element.
- Structure fields offsets are very similar to the macros used to define the local variable offsets on the stack; the difference is that structure offsets increase positively, whereas stack variable offsets increase negatively.
- Notice that in comparison to array access, structure access is always as efficient as simple variable access. In the case of structures, it is the assembler that has to compute the offsets from the origin of the structure. This is done at assembly time, not during program execution. Structure elements, or fields, cannot be accessed by a variable index, whereas array elements can.
- It is to our advantage, in C, to declare arrays with dimensions that are powers of 2, because of fast computation with shifting.
- The following example shows, how you can perform a very fast multiplication, without really multiplying, only use adding and shifting.

$\%o0 * 7 \Rightarrow \%o0 * 8 - \%o0$

$\Rightarrow \text{sub } \%g0, \%o0, \%o1 \quad ! \quad \%o1 = -\%o0$

$\text{sll } \%o0, 3, \%o0 \quad ! \quad \%o0 = \%o0 * 2^3$

$\text{add } \%o0, \%o1, \%o0 \quad ! \quad \%o0 * 8 - \%o0 = \%o0 * 7$

## Chapter 7: Subroutines

### Open Subroutine:

- An open subroutine is handled by the text editor or by the macro processor and is the insertion of the required code whenever it is needed in the program.

### Closed Subroutine:

- A closed subroutine is one in which the code appears only once in the program; whenever it is needed, a jump to the code is executed, and when it completes, a return is made to the instruction occurring after the jump instruction. Arguments to closed subroutines may be placed in registers or on the stack.

### Open Subroutines vs. Closed Subroutines:

- Open subroutines are very efficient in execution, with no wasted instructions. Arguments to open subroutines are very flexible and can be as general as the programmer wishes to make them. However, every time we need to multiply and insert the open subroutine, we generate additional code. If the open subroutine results in a short section of code, this is probably the correct thing to do. If the code generated were longer, simply repeating the code every time it was needed would begin to take up a lot of memory. It might be better to write code once, as a closed subroutine, and to branch to the code when needed, and then to return to the next instruction immediately after the branch.

## Register saving, p.206

- The SPARC typically provides 128 registers, with the programmer having access to the 8 global registers, and only 24 of the mapped (active) registers at any one time. The save instruction changes the register mapping so that new registers are provided.
- The 32 registers are divided into 4 groups: *in*, *local*, *out* and *global*. The 8 *global* registers, %g0 - %g7, are not mapped and are global to all subroutines.
- The *in* register are used to pass arguments to closed subroutines.
- The *local* registers are for a subroutine's local variables.
- The *out* registers are used to pass arguments to subroutines that are called by the current subroutine.
- The *in*, *local* and *out* registers are mapped. When the save instruction is executed the *out* registers become the *in* register, and a new set of *local* and *out* registers are provided. The mapping pointer into the register file is changed by 16 registers.
- The current register set is indicated by the current window pointer, "CWP", a machine register.
- The last free register set is marked by the window invalid bit, in the "WIM", another machine register.
- Each register set contains 16 general registers; the number of register sets is implementation dependent.
- The out register being used are from the invalid register window marked by the *wim* bit. If an additional subroutine call is made, a hardware trap occurs. The hardware trap is discussed fully in Chapter 12, but its effect is to move the 16 register form window set seven onto the stack where the stack pointer of register window seven is pointing.
- Saves and restores can be made in a range of 6 without window overflow or underflows occurring. Although this is efficient for general programming, it would become expensive if deeply nested recursive subroutine calls were frequently made.

## Subroutine Linkage:

- The SPARC architecture supports two instructions for linking to subroutines. Both instruction may be used to store the address of the instruction that called the subroutine into register %o7. As the instruction following the instruction that called the subroutine will also be executed, the return from a subroutine is to %o7 + 8, which is the address of the next instruction to be executed in the main program. If a save instruction is executed at the beginning of the subroutine, the contents of %o7 will become the contents of %i7 and the return will have to be to %i7 + 8.
- If the subroutine name is known at assembly time, the call instruction may be used to link to a subroutine. The call instruction has as operand the label at the entry to the subroutine and transfers control to that address. It also stores the current value of the program counter, %pc, into %o7. Like any instruction that changes the %pc, the call instruction is always followed by a delay slot instruction. The call instruction delay instruction may not be annulled.
- The instruction *jmp*l means: *jmp*l %to, %from . The "to" means where to jump, and the "from" means from where the jump came.
- Example: Assume the address of the subroutine is in the register %o0.

```
call subr          ! jmp l %o0, %o7
```

```
nop
```

```
subr: save %sp, x, %sp    ! %sp = %sp + x
```

**M**

```
ret                ! jmp l %i7+8, %g0 , because the %o7 becomes the %i7 with the save
```

```
restore %l0, 6, %o0 ! %l0+6 = %o0 = %i0 after the return jump
```

- If you make the assignment for the return value before the restore, you have to save it in %i0, instead of in %o0 as you do it with restore, because %i0 becomes %o0 after (with) the restore.

### Arguments to Subroutines:

- A problem occurs, if there are more than 6 arguments passed to the subroutine, because only the %o0 - %o5 registers can be used to pass arguments. The SPARC architecture recognizes this problem and allows the first six arguments simply to be placed in the *out* registers where the subroutine may access them directly. Unfortunately, only 6 *out* registers are available, as %o6 is the stack pointer and %o7 will be the return address if we call another subroutine. After execution of a *save* instructions the arguments will be in the first six *in* registers, %i0 - %i5.
- The convention established in the SPARC architecture is to pass the first 6 arguments in the first six out registers, %o0 - %o5, with any additional arguments placed on the stack.
- However, space is always reserved for the first 6 arguments on the stack even though they are not there. In fact, the space is reserved even if there are no arguments at all. Each argument occupies one word on the stack or register, so that when passing byte arguments to subroutines, they must be moved into word quantities before passing.
- The arguments are located on the stack, after the 64 bytes reserved for register window saving. However, immediately after the 64 bytes reserved for register window saving, there is a pointer to where a structure may be returned. Thus, the structure return pointer will be at [%sp + 64] and the first argument, if it were on the stack, at [%sp + 64] + 4.
- If you want to return a structure (not as a pointer), you have to place a copy of the desired structure at the address at place %sp + 64, so type: `ld [%sp + 64], %g1`, so %g1 contains now an address.
- Example: The following save instruction will provide:

*.global subroutine\_name*

*subroutine\_name :*

*save %sp, -(64 + 4 + 24 + local) & -8, %sp*

- Space for saving the register window set, if necessary,  $64 = 16 * 4$ , because there are 16 register à 4 byte to pass.
  - A structure pointer, 4 bytes
  - A place to save 6 arguments, used for pointers,  $24 = 6 * 4$
  - Space for any local variables
- The value returned by a function or subroutine is always returned in register %o0, that is, %o0 of the calling program. If a save instruction has been executed, %o0 will be %i0 before the restore instruction is executed.

### Leaf Subroutines:

- A leaf subroutine is one that does not call any other routines.
- For a leaf subroutine the register usage is restricted as follows:
  - The leaf routine may only use the first 6 *out* registers %o0 - %o5 and the *global* registers %g0 and %g1.
  - A leaf routine does not execute either a save or a restore instruction but simply uses the calling subroutine's register set, observing the restrictions listed above.
- The elimination of register saving and restoring makes calling a leaf routine very efficient.
- A leaf routine is called in the same manner as a regular subroutine, placing the return address into %o7. As a save instruction is not executed, the return address for a leaf routine is %o7 + 8, not %i7 + 8, this means for practically purposes, use *retl* instead of *ret*.

### Pointers as arguments:

- You can't pass a pointer within a register to a subroutine, because a pointer has an address. You have to pass the value the pointer points to onto the stack while the address to the place in the stack is stored in the registers %o0 - %o5.

## Chapter 8: Machine Instructions

- Instructions on the SPARC architecture occupy one word, 32 bits.
- It's important that decoding the instruction format be simple and direct if we are to execute an instruction each machine cycle.
- There are 3 types of instructions:
- Format One Instruction:
  - There's only one Format One instruction, the call instruction.
  - The call instruction must be able to transfer control to any location in the 32 – bit address space. The target of such a transfer must be an instruction and thus word aligned. A word – aligned address has the least significant two bits, both zero, so that any possible target address contains 30 bits of information. A Format One call instruction contains an op field of <01> followed by 30 bits of address.
  - Although the actual address could be stored, right – shifted two bits, in the instruction, it is the address relative to the current contents of the program counter that is stored. Why is this? Programs are frequently moved around in memory, requiring that the addresses of all labels be changed. Thus all the subroutines addressed in call instructions would also have to be changed. However, if the address were stored relative to the program counter, no matter where the program was moved in memory the relative address would remain the same. Program counter relative addresses do not have to be changed when a program is moved in memory and can be computed by the assembler at assembly time.
- Format Two Instruction:
  - Format Two instructions are the branch and the sethi instructions.
  - In this case, the target of the branch is also stored relative to the program counter right – shifted two bits. However, in the case of a branch, only 22 bits are available for the displacement, so that the target of branches may be only  $\pm 2^{21}$  instructions from the program counter. Branches to targets that are further away than +8'388'604 , –8'388'608 bytes will be discussed shortly.
- Format Three Instruction:
  - The majority of instructions are of this type. Example: `sub %lo, 5, %o0`
  - The constant in the above example in the range of -4096 to 4095.
- There could be a maximum of  $2^7 = 128$  instructions in the SPARC architecture. Not all numbers are used, for example 001001 is an unimp (Unimplemented instruction).
- To load constants that are larger than  $\pm 4095$  , have to be loaded as follows:
  - So far we have worked only with small constants in our programs. We were limited by the 13 – bit signed – immediate field of the instruction. It would be difficult to load constants that were longer than 13 bits. If we need a larger constant, we will need to use the sethi instruction, which will load the high 22 bits of a register while clearing the low 10 bits with zeros.
  - Example to load %o0 with 0x30cf0034:  
`sethi 0x30cf0034 >> 10, %o0`  
`or %o0, 0x30cf0034 & 0x3ff, %o0`  
The 0x3ff has all 0's except the last 10 positions are 1's.
  - There are two other ways to do this:  
`%hi(x)  $\Leftrightarrow$  x >> 10`  
`%lo(x)  $\Leftrightarrow$  x & 0x3ff`  
 $\Rightarrow$  `sethi %hi(0x30cf0034), %o0`  
 $\quad$  `or %o0, %lo(0x30cf0034), %o0`  
 $\Rightarrow$  `set 0x30cf0034, %o0`

## Chapter 9: External Data and Text

### Introduction:

- So far we have made use of memory only to store our programs for execution and to store our local variables and function arguments. In this chapter we discuss external and static variables. Local variables, stored on the stack, may be addressed relative to the frame pointer; however, to make those same variables available to other functions would be very difficult. As the value of static variables in functions does not change between function calls, they may not be stored on the stack where storage is created and released between function calls. To solve this problem, external and static variables are stored in memory much like the program. Their addresses are then made available to all function that need to access the variables.
- When a program is loaded into memory, the program text, initialised variables, and zero – initialised variables are loaded into different regions of memory. These regions are called sections, and each generally starts on a 0x2000 byte boundary. In this way, memory protection may be applied different parts of the program. Program text is normally read – only, meaning that if we attempt to store something into this area of memory, we will get a system trap. The other two regions are read – write, meaning that all accesses are valid. The three regions of memory are called the text, data, and bss sections, respectively.
  - text section: There are the program and any read only data located and the text section is where the assembler puts things unless we tell it otherwise.
  - data section: It's for initialised data.
  - bss section (block starting symbol): It's for zero initialised data.
- When the program is loaded into memory, the text and data sections are loaded first into low memory. Then space is zeroed for the bss section. These three sections are all at low memory, leaving the stack at high memory. The stack has nothing to do with program sections. In C, all the external variables that are not specifically initialised are located in the bss section. The first 0x2000 bytes of memory are reserved for the operating system and are not used by regular programs. Look at picture on page 249.

### Switch Statement:

- Although there is grater overhead in a switch statement, compared to a series of if else statements, it becomes increasingly efficient as the number of choices increases. Tue use of a switch statement becomes questionable when the number of choices is limited or when the labels have a great range. Of course, a complier might translate such code into the if else form.

### The Loader:

- A C program consists of a number of external objects, variables, and functions. These external objects may be grouped together onto any number of separate source files, and each file may be compiled and assembled separately. They are combined by a program called the loader, ld; the loader is generally called by the compiler as a final pass. The loader's main task is to resolve the external symbols so that all references to a particular symbol refer to the same location. The scope of symbols in assembly language is the extent of the source file. If symbols are to be made available to other software modules, the .global declaration must be used.

## Chapter 10: Input / Output

### Introduction:

- Communication with input / output devices is accomplished through memory in the SPARC architecture. A section of memory, form 0xfff00000 to 0xffffe000, might be replaced by device registers.
- When load and store instructions have addresses in the device registers section of memory , devices are activated, instead of the normal storing and retrieving of data. Each device has a unique address, or addresses, assigned to it. When the computer reads or writes to one of these reserved device memory locations, it is not addressing memory but instead communications with the device. In this way input and output may be performed with the regular instruction set of the computer and no special input / output instructions are needed.

### **Character Devices:**

- To write characters to a simple character device the stb instruction is used, addressing a physical hardware device register. To read from a character device the ldub instruction is used, once again addressing a physical hardware device register.
- Example for writing:  

```
mov "a" , %o0  
set 0xffff0000, %o1  
stb %o0 , [%o0]
```
- Example for reading:  

```
set 0xffff0008 , %o1  
ldub [%o1] , %o0
```

### **Programmed I / O:**

- There is the problem of synchronisation. How do we know when a character has been typed so that we may load it from the device data register?
- This information is provided by another device register called the status register. For a simple device like a keyboard, the status register would be a single byte and would be located in memory adjacent to the data register. The interfacing of a device normally results in a block of memory containing the device registers. In the status registers there is normally a ready bit to indicate that the device is ready to accept data or that it has data ready to be taken from the data register. The status register may also contain an error bit to indicate that a device error has occurred. As we will see later, it also contains an interrupt for service bit.

### **Interrupted – Driven I / O:**

- Assuming that the computer has something useful to do while the input or output is going on, it would be efficient if the machine were able to proceed, only returning to service the input / output device when it was ready. This facility has always been provided in terms of interrupts. Input / output devices may interrupt the computer when they need service, for example, by setting the ready flag. When an interrupt occurs the state of the machine is saved and the computer then executes code related to handling the input / output device. When the device has been serviced, the computer returns to the program it was executing, first restoring the state.
- By using interrupts we avoid the problem of not keeping the CPU waiting for a slow device.
- Example: There is assembler code which prints the first char of a string and then points to the next char of the string, then the code is finished  $\Rightarrow$  now the device which has to print the char (e.g. the printer) prints the given char (e.g. by a certain address in memory) and when it has finished, it calls an interrupt and the necessary trap code has to be executed  $\Rightarrow$  in this trap the next char is going to be printed  $\Rightarrow$  after the output device has finished it calls an interrupt [the device has a certain predefined place in the trap – table which points to the place where the device driver code is located which has to handle the interrupt (means print the char and move one position ahead and go back] again (the output device always calls a trap if it's ready, means finished)  $\Rightarrow$  and so on and on.

### **System I / O:**

- Input and Output are handled by the operating system. It is the operating system that controls all input / output devices.
- To perform any input or output from a program, one must make a request to the operating system. Such a request is called a trap, or system call. We have already seen the use of a trap to exit at the end of a program:  

```
mov    1 , %g1
ta     0
```
- The service requested is represented by the number in register %g1. The trap instruction ta is like a subroutine call in that it transfers control to a different address. Unlike a call instruction, the trap instruction has no delay slot. The address to which the trap instruction transfers control is stored in a table in the operation system. The trap instruction, as a side effect, changes the mode in which the computer is operating from user mode to supervisor mode. In supervisor mode a program may execute additional instructions and may address the input / output devices.

## **Chapter 12: Traps and Exceptions**

### **Introduction:**

- Shared resources are handled by the operating system, and their access to users is protected.
- How a program is executed look at p.329.
- The mechanism for preventing access to shared resources and the operation system itself is by means of two mechanisms, a different mode of execution and the trap mechanism. Programs may execute in two modes in the SPARC architecture, supervisor and user. Certain instructions may only be executed in supervisor mode, such as those that access state registers and input/output devices. We cannot access the processor state register when executing a program in user mode, but the operation system may read and write the register, The current state of the processor , supervisor or user, is kept in the processor status word. A number of load and store instructions exist to access other segments of memory that contain the devices and memory management. These load and store alternative instructions may only be executed in supervisor mode and are know as privileged instructions.

### **Trap:**

- The trap instruction is a nondelayed branch and does not involve the execution of a delay slot instruction.
- There are 256 possible trap types specified, half software traps and half hardware.
- For each exception that may cause a trap and for each external interrupt request a priority and trap type tt are defined. The traps are grouped together in the table: machine and memory failure, page faults, window overflow and underflow, privileged and illegal instructions, floating – point, arithmetic, trap instructions, interrupts.
- The rett instruction is a privileged instruction that will cause a trap if executed in user mode. The supervisor normally also clears the stack and any register windows it has used before returning from a trap.
- There are the following registers assignments:
  - PSR ⇒ %10
  - PC ⇒ %11
  - nPC ⇒ %12
- The return value of a trap instruction is located in %o0 .
- Parameters to traps are given in the %o0 - %o5 registers.
- From page 333 on, the mechanism of how a trap is going to be executed, is listed.

### **Processor State Registers:**

- The integer processor has the following state registers:
  - The multiply / divide register Y
  - The program counter, PC and nPC
  - The processor state register, PSR (holds the CWP)
  - The window invalid mask, WIM
  - The trap base register, TBR

### **Window Invalid Mask Register:**

- This register, 32 bits long, has active bits for each register set present and has one of the bits set of the bits set for the register window set which is invalid, WIM[CWP] = 1; all other bits are zero.

### **Trap Base Register:**

- The trap base register, TBR, holds the memory address of the first of 4 instructions of the code to handle the trap.

## **Chapter 13: Memory Management**

### **Introduction:**

- As the performance of a computer goes up, so does the cost, and the economics of a single – user machine grows steadily worse, so that more powerful machines are generally designed to support a number of users at the same time. This is called time – sharing, handled by switching the processor between users for some fraction of a second each, so that all users appear to have a less powerful machine entirely to themselves.
- Switching between users occurs with interrupts.
- Memory is handled in the SPARC architecture by a unit called the memory management unit, MMU, which is presented with virtual addresses by the processor and translates these addresses into addresses of the physical memory attached to the memory management unit.

### **Paging:**

- With paging, an entire program does not have to be resident in memory for the program to be executed. Only a few pages are necessary to start the program, an initial text page, data page, bss segment page, and a page of stack.
- When a page that is not in main memory is accessed and the zero page table entry encountered, a page fault trap occurs. The operation system then arranges to have the missing page placed somewhere into memory and the page table updated. During this time, called paging, some other process is run, p.355.