SPARC Exception Handling

The SPARC supports various types of traps:

- Synchronous traps
  - Exceptions generated by standard instructions
    e.g. `illegal_instruction`, `window_overflow` etc.
  - Exceptions generated by memory access instructions
  - Exceptions generated by floating-point instructions
  - Exceptions generated by coprocessor instructions
  - Explicit trap instruction
    Trap on integer condition codes, `Ticc`.
- Asynchronous traps
  - Reset
  - Interrupts

SPARC Exception Handling

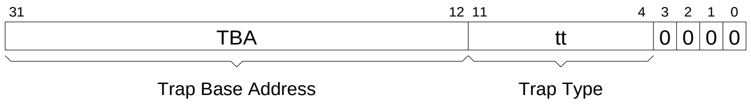
Each trap is assigned a priority and a trap type.

Trap	Priority	Trap Type	Sync/ Async
Reset	1	-	Async
Illegal Instruction	2	1	Sync
Privileged Instruction	3	2	Sync
Tag Overflow	13	10	Sync
Ticc	14	128-255	Sync
Interrupt level 15	15	31	Async
Interrupt level 14	16	30	Async
Interrupt level 2	28	18	Async
Interrupt level 1	29	17	Async

SPARC Exception Handling

- Priority
  - Where multiple traps occur the highest priority trap is taken.
- Trap Type
  - When a trap is taken the value of the Trap Base Register is copied into the Program Counter.
  - The trap type forms a field in the Trap Base Register.

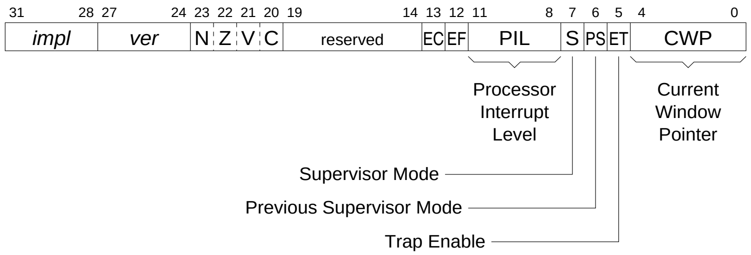
Trap Base Register



Exception Handling

- The processor state register contains a number of fields relevant to exception handling:

Processor State Register



Trap Operation

- Further traps are disabled
  - interrupts are ignored
  - synchronous traps force the processor into error mode

    `ET ← 0`
- Processor enters supervisor mode
  - also save previous state

    `PS ← S`

    `S ← 1`
- A new register window is selected

  `CWP ← CWP - 1`

- PC and nPC are saved into new window

  `%l1 ← PC`

  `%l2 ← nPC`
- The *tt* field in the Trap Base Register is set appropriately
- Jump to service routine
  - For all traps except reset:

    `PC ← TBR`

    `nPC ← TBR + 4`
  - For reset:

    `PC ← 0`

    `nPC ← 4`

Trap Table

- The trap table contains 256 4-word entries (although many are unallocated).
- A 4-word entry is rather small for a full service routine, a typical entry will save the processor status register and jump to the real routine:

```
trap_entry:
        mov %psr, %l0
        sethi %hi(trap_func), %l4
        jmp %l4 + %lo(trap_func)
        nop
```


Trap Routine

- The trap routine will restore the status register before returning from the trap.

```
trap_func:
        !
        ! body of trap routine in this space
        !
        mov %l0, %psr
        nop             ! flush %psr update through pipeline
        nop
        nop
        jmp %l1         ! restores old PC
        rett %l2        ! restores old nPC and other bits
```

Return from Trap

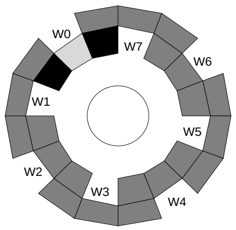
- Return from trap is achieved using the *control transfer couple*:

```
JMPL %l1, %0[1]
                    %0  ← PC
                    PC  ← nPC
                    nPC ← %l1
RETT %l2
                    ET  ← 1
                    PC  ← nPC
                    nPC ← %l2
                    CWP ← CWP + 1
                    S   ← PS
```

[1]JMP xx ≡ JMPL xx, %0

Trap Window

- The trap operation decrements the window pointer without performing a check for an invalid window. Since the local variables of an invalid window don't overlap with other windows, we can use these registers in our trap handler.



CWP = W0 (W0 INVALID)

Fast Trap Routines

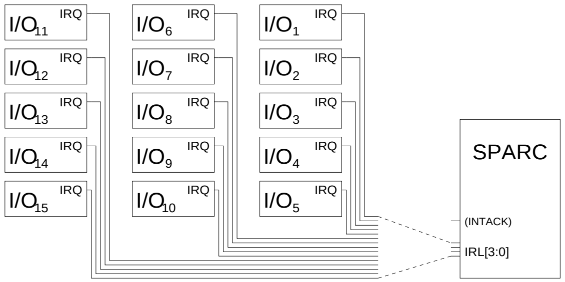
The following restrictions apply to fast trap routines:

- Use only local variables `%l3` – `%l7`
  - since `%l0`, `%l1` & `%l2` are used for `PSR`, `PC` & `nPC`
- Use no subroutines
  - since `CALL` would overwrite `%o7`
- Don't re-enable traps
  - since a nested trap would decrement `CWP` again

Slow Trap Routines

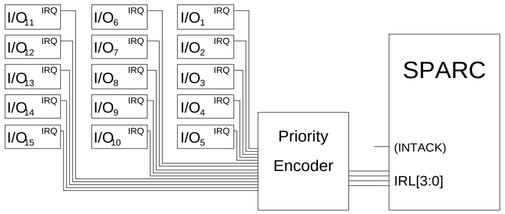
For more complex trap routines we may wish to lift these restrictions. In particular we may wish to re-enable traps and make use of subroutines. The following conditions should be met:

- Check for invalid window on entry and before return
  - manipulate *old window stack* as required
  - `RETT` will not restore an *old window* from the stack
- Top level of trap routine has restricted variable access
  - uses locals `%l3` – `%l7` and outs `%o0` – `%o7`
- Disable traps before return
  - automatic (provided we save and restore processor status)



- The SPARC supports up to fifteen separate interrupt lines via the four bit IRL bus.
  - IRL[3:0] = n indicates that interrupt request n is active.
  - IRL[3:0] = 0 indicates that no interrupt request is active.

- Since the IRL bus can only indicate one active interrupt to the processor we use a simple priority encoder.
  - Interrupt request 15 (IRL[3:0] = 1111_2) has highest priority and request 1 has the lowest.



- A more advanced system might use a custom interrupt controller combined with the INTACK signal from the SPARC[2].

[2]not all SPARC processors support INTACK

Interrupt Cycle

- Interrupt request on line 5
- Priority encoder output changes from 0000_2 to 0101_2
- SPARC retimes IRL signals and checks for stability over two cycles
- SPARC interrupts at end of current instruction
- Trap type field of the Trap Base Register is set to 21 (5+16)
- SPARC jumps to trap routine for Interrupt Level 5

Interrupt Masking

The 4 bit Processor Interrupt Level (PIL) field of the Processor State Register provides a simple priority based interrupt mask.

- PIL = 0 allows all interrupts provided that traps are enabled (ET = 1).
- PIL = n masks all interrupts whose interrupt level is not greater than n.
- Interrupt level 15 is not maskable by the PIL (hence PIL = 14 is equivalent to PIL = 15).

SPARC Interrupts

Fast Interrupt Service Routines

- These routines are the fast trap routines already discussed.

  They may make only limited use of the SPARC register set and cannot support subroutines or nested traps.

Slow Interrupt Service Routines

- Slow interrupt routines may have full use of the facilities offered by the SPARC register set provided that they are careful to avoid causing a window overflow or underflow trap while traps are disabled.
  - Flat register window model subroutines may be used once a full register window has been acquired.
  - Standard register window model subroutines may be used if traps are re-enabled.
- Once traps are re-enabled we may support nested interrupts.

SPARC Interrupts

Nested Interrupts

- Standard behaviour of a slow interrupt routine which wants to support nested interrupts is:
  - Set the PIL field of the PSR to the level of the current interrupt when re-enabling traps.
    This blocks lower priority interrupts.
  - The previous PIL value will be restored along with the preserved PSR at the end of the service routine.
- Alternatively the routine might:
  - Set PIL to 15 to block all except the non-maskable interrupt.
  - Set PIL to zero to enable all interrupts.
    This is likely to be the case where an alternative priority system is implemented.

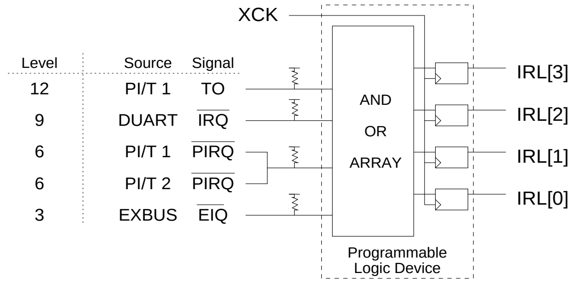
SPARC Interrupts

Summary

- Basic SPARC interrupt scheme:
  - priority fixed in hardware
    assisted by a simple priority encoder
  - maximum of 14 prioritized interrupt levels
  - single non-maskable interrupt
  - limited masking control
    since the PIL is used to support the priority scheme.

Where we require a more flexible scheme we might use a more advanced interrupt controller.

REB Interrupt Scheme



- The RISC Experimenter Board supports 5 interrupts
- Priority is resolved by a simple programmable logic device (PLD)
- PLD also synchronizes the signals to XCK
- $\overline{PIRQ}$ signals from the 2 PI/T devices are wire-or'ed at level 6

Trap Table

- EPROM programs
  - trap table is in EPROM 0 to FFF - unalterable.
  - default trap routines simply light LEDs for diagnostics.
- RAM programs
  - trap table is copied to RAM at C1000 to C1FFF
  - trap base address is adjusted accordingly
  - user may then install own handlers