

# The ETRAX FS I/O-processor HOWTO

## TABLE OF CONTENTS

- 1 DOCUMENT HISTORY
- 2 REFERENCES
- 3 DEFINITIONS
- 4 INTRODUCTION
  - 4.1 About this document
- 5 The I/O-processor Assembler
  - 5.1 The Assembler
  - 5.2 Assembler Directives
  - 5.3 The Assembly Language
  - 5.4 The FSM Mode
    - 5.4.1 The FSM Timer
- 6 The I/O-processor Registers
  - 6.1 Register Macros
    - 6.1.1 C Macros
    - 6.1.2 Assembler Macros
  - 6.2 Registers and Ownership
- 7 The I/O-processor
  - 7.1 The MPU
    - 7.1.1 MPU Interrupts
  - 7.2 The SPUs
    - 7.2.1 FSM code
  - 7.3 The Data Flow
  - 7.4 The DMC
  - 7.5 The FIFOs
  - 7.6 The CRC
  - 7.7 The SAP
    - 7.7.1 The SAP in
    - 7.7.2 The SAP out
  - 7.8 The Switch
    - 7.8.1 Set and Block Signals
    - 7.8.2 Pin Mapping
  - 7.9 The Timers
  - 7.10 The Triggers
  - 7.11 The MC
    - 7.11.1 Reading from, and Writing to, System Memory
    - 7.11.2 SPU Memory Load

# 1 DOCUMENT HISTORY

Revision	Date	Description	Author
1.0	Mar 1, 2005	Corrections, changes and additions after first review	Jenny Ottosson

## 2 REFERENCES

- ETRAX FS Designer's Reference (PDF)
- IOPASM Manual (NOTE: No link available as yet.)

## 3 DEFINITIONS

<b>IOP</b>	The I/O-processor of the ETRAX FS
<b>CRC</b>	Cyclic Redundancy Check
<b>DMC</b>	DMA Communicator
<b>CPU</b>	The main CPU of the ETRAX FS
<b>MC</b>	Memory Controller
<b>MPU</b>	Master Processing Unit
<b>SAP</b>	Synchronization and Asynchronous Paths
<b>SPU</b>	Slave Processing Unit
<b>GIO</b>	General Input/Output, a 32-bit bus
<b>ALU</b>	Arithmetic Logic Unit
<b>FSM</b>	Finite State Machine
<b>SEQ</b>	Sequential code or mode (ordinary assembly language)

## 4 INTRODUCTION

### 4.1 About this document

This document provides an easy way to get started with writing code for the I/O-processor of the ETRAX FS.

Extensive register excerpts have been added to make it possible for a reader to use this document on its own without having to look up the register configuration for the code examples in the register description documentation.

Only a few examples are given for each module, and some modules do not have any example code at all. Not every instruction, nor every macro, is shown and exemplified, but hopefully enough to get started. Please note that this document does not in any way aim at giving a complete understanding of the I/O-processor.

# 5 The I/O-processor Assembler

## 5.1 The Assembler

The ETRAX FS I/O-processor assembler is a conventional microprocessor assembler. It handles assembling of both the MPU and SPU code, as well as assembling the special FSM (Finite State Machine) code in the SPUs.

There are several command-line switches, of which the most important ones for generating executable code are:

```
--arch [MPU or SPU]
--output-format [binary, object, vlog64, or vlog256]
```

Examples:

```
iopasm --arch SPU --output-format binary -o output_file.spu.bin my_file.spu.s
iopasm --arch MPU --output-format binary -o output_file.mpu.bin my_file.mpu.s
```

For a complete list of I/O-processor assembler switches and examples, see IOPASM Manual (NOTE: No link available as yet.).

## 5.2 Assembler Directives

The assembler has the following directives:

<b>.org</b> <i>address</i>	Set program counter for next instruction
<b>.seq</b>	Switch to sequential mode (SPU only)
<b>.fsm</b>	Switch to FSM mode (SPU only)
<b>.align</b> <i>boundary</i>	Align next instruction to <i>boundary</i> bits
<b>.dword</b> <i>expression-list</i>	Fill memory with data
<b>.end</b>	Mark end of file (required)

## 5.3 The Assembly Language

The general syntax of the assembly language is:

```
instruction [source operand], [destination operand]
```

Characters to indicate a comment could be:

semicolon ;  
hash #

A C comment (*/\* \*/*) will be deleted by the C preprocessor.

Numbers can be written in decimal:

```
moveq 2005, r1 ; Move the number 2005 into register r1
```

or in hexadecimal:

```
; Move the hexadecimal number ffff into register r2  
moveq 0xffff, r2
```

or in binary:

```
; Move the binary number 00110101 into register r12  
moveq %00110101, r12
```

Labels are treated as immediates:

```
spu0_label:  
; Set register r7 to address of label "spu0_label"  
moveq spu0_label, r7
```

Branches have delay slots

```
ba spu0_label ; Branch to "spu0_label"  
nop ; Delay slot
```

Example showing SPU-specific assembly code:

```
#define init_oe_lo 0x00ff  
#define init_gout_lo 0x00dc  
  
.seq ; Sequential (ordinary assembly) code start  
init:  
  movel r0, 0x0000, r0 ; Set the lowest 16 bits of register r0  
  moveh r0, 0x0000, r0 ; Set the highest 16 bits of register r0  
  movel r1, 0xffff, r1 ; Set the lowest 16 bits of register r1  
  moveh r1, 0xffff, r1 ; Set the highest 16 bits of register r1  
  
  ; Set output enables for the lowest 16 bits  
  rwq init_oe_lo, REG_ADDR(iop_sw_spu, iop_sw_spu0, rw_gio_oe_set_mask_hi)  
  
  ; Set initial values for the lowest 16 bits on the GIO bus  
  movel GOUT, init_gout_lo, GOUT  
  
  halt  
.end ; End of program
```

## 5.4 The FSM Mode

The SPUs have a special programming mode called FSM, finite state machine. The FSM mode has its own set of instructions and allows finite state machines to be executed directly in the processor.

The FSM statement is made up of four input signals and four output signals:

```
fsm_label:
    (inputs)          (outputs)      (label to jump to)
    [3]_[2]_[1]_[0] : [3]_[2]_[1]_[0] : [label]
```

If the conditions of the input signals are met, the output signals will be given the value set (if any are set) and a jump to the label is performed.

Every such line containing 4 input signals and 4 output signals and an FSM label is called transition instruction.

There are a number of flags that can be set to an FSM statement to indicate that it contains, for example: a sequential instruction; an event; or that the SPU should switch to sequential mode (ordinary assembly mode). An FSM flag must appear at the top, above the inputs and outputs:

```
fsm_label_0:
    umask = [4-bit value] ; Choose which event to update
    emask = [4-bit value] ; Choose which event to enable
    seq moveq 0x1, r0      ; Sequential instruction
    do_seq                 ; Execute the seq instruction continuously
    go_seq                 ; Switch to sequential mode
    fsm_halt               ; Disables the SPU
    inp = [4-bit value]    ; Choose which 4 inputs to listen to
    outp = [8-bit value]   ; Choose which 4 outputs to set
    timer 4 : 1_?_0_? : fsm_label_1 ; Timer instruction
    1_?_?_? : ??_?_1_0 : fsm_label_2 ; Transition instruction
```

All flags are per default set to zero.

A value written to an FSM flag will not be permanent, but will only be valid for the FSM statement in which it appears.

Simple example:

```
fsm_label_0:
    inp = 0
    outp = 0
    ??_?_1_0 : ?_0_?_1 : fsm_label_1
```

Explanation of the example above	
<b>inp</b>	Selects which input signals to listen to. There are 16 combinations to choose from, and the signals could be anything from status information from the FIFO, DMC, Timer, Trigger, etc., to input signals from the pads. The value must be in either decimal, hexadecimal or binary. Constant expressions can be used as long as the value evaluates to decimal, hexadecimal or binary within the specified range.
<b>outp</b>	Selects which output signals to write to. The outputs can only be set to the lowest 8 bits of SPUs GIO. These 8 outputs can then be rerouted to other GIO signals, or to output enables, and more. The value must be in either decimal, hexadecimal or binary. Constant expressions can be used as long as the value evaluates to decimal, hexadecimal or binary within the specified range.
(inputs) ??_?_1_0	The condition is met if: input[0] = 0 input[1] = 1 input[2] = Don't care input[3] = Don't care
(outputs) ?_0_?_1	If the conditions of the input signals are met, the output signals will be set accordingly: output[0] = 1 output[1] - Not affected output[2] = 0 output[3] - Not affected
<b>fsm_label_1</b>	If the input conditions are met, a jump to this FSM label will be made. If the input conditions are not met, the SPU will remain on this FSM statement and the input conditions will be reevaluated every 5 ns.

There can be up to 8 transitions in an FSM statement. They are evaluated as a priority encoder from top to bottom.

```
fsm_label_0:
    inp = 0
    outp = 0
    1_0_0_0 : 0_0_0_0 : fsm_label_1 ; Highest priority, evaluated first
    ?_1_1_1 : 1_1_1_1 : fsm_label_2 ; Second priority
    ?_1_1_0 : 1_1_?_? : fsm_label_3 ; Third priority
    ?_1_0_1 : ?_?_1_1 : fsm_label_4 ; Fourth priority
    ?_1_0_0 : 1_?_?_? : fsm_label_5 ; Fifth priority
    ?_?_1_1 : ?_1_?_? : fsm_label_6 ; Sixth priority
    ?_?_1_0 : ?_?_1_? : fsm_label_7 ; Seventh priority
    ?_?_?_1 : ?_?_?_1 : fsm_label_8 ; Least priority, evaluated last
```

The evaluation of the inputs is done immediately, but if none of the input conditions are true, the SPU will remain on this FSM statement and reevaluate the input conditions every 5 ns.

There are two assembly instructions to enter FSM mode:

```
moveq fsm_label, FSMPC ; Set the program counter for where to
                        ; start FSM mode
nop          ; Wait for FSMPC value to be updated

fsm         ; Goto FSM mode (at the address in FSMPC register)
nop         ; Delay slot

.align 64   ; Align the FSM code at a 64-bit boundary
.fsm       ; Tell the assembler that we switch to FSM code

fsm_label:
    inp = 0
    outp = 0
    ?_?_?_? : ?_?_?_? : fsm_label
```

Or...

```
fsmq fsm_label ; Jump to FSM label "fsm_label"
nop          ; Delay slot

.align 64     ; Align the FSM code at a 64-bit boundary
.fsm         ; Tell the assembler that we switch to FSM code

fsm_label:
    inp = 0
    outp = 0
    ?_?_?_? : ?_?_?_? : fsm_label
```

(Note that there is a delay slot following the instructions **fsm** and **fsmq**, and note also that a new value to FSMPC takes 5 ns before it can be used by the instruction **fsm**.)

To go from FSM mode to sequential mode (that is, ordinary assembly mode) there is an FSM flag named **go\_seq**:

```
fsm_label_100:
    go_seq          ; Switch to sequential mode
    only seq_label ; Goto the sequential label "seq_label"

.seq ; Tell the assembler that we switch to sequential mode

seq_label:
```

## 5.4.1 The FSM Timer

An FSM statement can also have a timer instruction. There can only be one timer instruction in an FSM statement, and it must be placed first or last:

```
fsm_label_0:
    outp = 0
    timer 5 : 0_0_0_0 : fsm_label_1
    1_0_0_0 : 0_0_0_0 : fsm_label_2
    ?_1_1_1 : 1_1_1_1 : fsm_label_3
    ?_1_1_0 : 1_1_?_? : fsm_label_4
    ?_1_0_1 : ?_?_1_1 : fsm_label_5
    ?_1_0_0 : 1_?_?_? : fsm_label_6
    ?_?_1_1 : ?_1_?_? : fsm_label_7
    ?_?_1_0 : ?_?_1_? : fsm_label_8

fsm_label_0:
    inp = 0
    outp = 0
    1_0_0_0 : 0_0_0_0 : fsm_label_1
    ?_1_1_1 : 1_1_1_1 : fsm_label_2
    ?_1_1_0 : 1_1_?_? : fsm_label_3
    ?_1_0_1 : ?_?_1_1 : fsm_label_4
    ?_1_0_0 : 1_?_?_? : fsm_label_5
    ?_?_1_1 : ?_1_?_? : fsm_label_6
    ?_?_1_0 : ?_?_1_? : fsm_label_7
    timer 5 : 0_0_0_0 : fsm_label_8
```

The order of the timer instruction (placed first or last in the FSM statement) gives the priority of the timer: If the timer is placed first and one of the input conditions is met at exactly the same time (that is, on the same SPU cycle) as the timer expires, the timer will have priority over the input condition.

The value of the timer could either be an immediate (in which case it is a 12-bit number), or a general register of the SPU.

Example of using a general register as the timer value:

```
movel r7, 0xffff, r7 ; Move lowest 16-bit value to general register r7
moveh r7, 0x001f, r7 ; Move highest 16-bit value to general register r7

fsmq fsm_timer_0    ; Switch to FSM mode
nop                ; Delay slot

.align 64          ; Align the FSM code at a 64-bit boundary
.fsm               ; Tell the assembler that we switch to FSM code

fsm_timer_0:
    timer r7 : 0_0_0_0 : fsm_timer_1
```

The timer value gives a delay in steps of 5 ns. Note however, that an FSM statement takes 10 ns to execute, and there is a default delay in a timer instruction of 5 ns. This means that a timer value of 0, 1, and 2 equals a delay of 15 ns. A timer value of 3 equals 20 ns; a timer value of 4 equals 25 ns, and so on.

The immediate timer value is a 12-bit number, and the highest value is 4095. The maximum delay achieved by the immediate timer value is therefore:

$$((4095 + 1) \cdot 5) = 20\,480 \text{ ns} = 20.48 \mu\text{s}$$

If a general register is used as a timer value, a 32-bit number can be used. This means that the maximum delay achieved by the use of a general register is (with a 3-digit accuracy):

$$(4.29 \cdot 10^9 + 1) \cdot 5 = 2.15 \cdot 10^{10} \text{ ns} = 21.5 \text{ seconds}$$

There is also a flag for keeping, or transferring, the timer value from the previous FSM statement, or rather, what is left of the timer value from the previous timer.

Here is an example:

```
fsm_timer_1:
; Set a timer of 500 ns delay.
; When, or if, it expires, set output
; bit[1] to high, and bit[0] to low
timer 99 : ?_?_1_0 : fsm_label_3
; If bit[3] is high, goto fsm_timer_2
; and set output bit[3] to high
1_?_?_? : 1_?_?_? : fsm_timer_2

fsm_timer_2:
; Keep what is left of the timer value in the
; previous FSM statement. This will make sure
; that we will reach fsm_label_3 500 ns from
; fsm_timer_1.
; When it expires, set output bit[1] to high,
; and bit[0] to low
timer keep : ?_?_1_0 : fsm_label_3

fsm_label_3:
; We will (under any circumstances) reach here
; 500 ns from fsm_timer_1.
; If bit[3] is low, goto fsm_label_4
; and set output bit[1] to high, and
; bit[0] to high
0_?_?_? : ?_?_1_1 : fsm_label_4
```

More on FSM code can be found in chapter 7.2.1 FSM code.

# 6 The I/O-processor Registers

## 6.1 Register Macros

There are a number of macros which simplifies register configuration; there is one set of macros for the C environment, and one set for assembly programming.

### 6.1.1 C Macros

Two C programming register macros for reading and writing are:

```
REG_RD (scope, inst, reg);  
REG_WR (scope, inst, reg, value);
```

<b>scope</b>	Name of the scope Typically: iop_[register bank]
<b>inst</b>	Base address of the instance of the scope Typically: regi_iop_[register bank and number]
<b>reg</b>	Name of the register
<b>value</b>	The field or bit value of the register

Example:

```
fifo_in_cfg = REG_RD(iop_fifo_in, regi_iop_fifo_in0, rw_cfg);  
REG_WR(iop_fifo_in, regi_iop_fifo_in0, rw_cfg, fifo_in_cfg);
```

The variable above must be the typedefed struct for the register in question:

```
reg_iop_fifo_in_rw_cfg  fifo_in_cfg;
```

If an "int" variable needs to be read or written the following macros can be used:

```
int timer_reg;  
int fifo_reg;  
  
timer_reg = REG_RD_INT(iop_sw_cfg, regi_iop_sw_cfg, rw_timer_grp0_owner);  
REG_WR_INT(iop_sw_cfg, regi_iop_sw_cfg, rw_fifo_in0_owner, fifo_reg);
```

There is also a macro for converting values of different variable types:

```
REG_TYPE_CONV(to type, from type, value of from type)
```

Example showing how to convert the value of a struct variable to the value of an int variable:

```
reg_iop_fifo_in_rw_cfg  fifo_in_cfg;  
int                     int_variable = 0;  
  
/* Configure the members of the struct */  
fifo_in_cfg.mode        = regk_iop_fifo_in_size16;  
fifo_in_cfg.byte_order  = regk_iop_fifo_in_order8;  
  
/* Convert the value of the struct to the int variable */  
int_variable = REG_TYPE_CONV(int, reg_iop_fifo_in_rw_cfg, fifo_in_cfg);  
  
/* Write the value back to the register  
by the use of the int variable */  
REG_WR_INT(iop_fifo_in, regi_iop_fifo_in0, rw_cfg, int_variable);
```

Example of converting the value of an int variable to the value of a struct variable:

```
int                int_variable = 0x0000ffff;
reg_iop_sw_cfg_rw_bus0_mask  sw_cfg_bus0_mask;

/* Convert the value of the int variable to the struct variable */
sw_cfg_bus0_mask = REG_TYPE_CONV(reg_iop_sw_cfg_rw_bus0_mask, int, int_variable);

/* Write the register value with the struct variable */
REG_WR(iop_sw_cfg, regi_iop_sw_cfg, rw_bus0_mask, sw_cfg_bus0_mask);
```

In some cases there are several copies of a register (for example one register for each timer in a timer group). The macros for accessing these registers are:

```
REG_RD_VECT (scope, inst, reg, index);
REG_RD_INT_VECT (scope, inst, reg, index);

REG_WR_VECT (scope, inst, reg, index, value);
REG_WR_INT_VECT (scope, inst, reg, index, value);
```

<b>index</b>	Register number, in case there are several copies of the register
--------------	---

Example:

```
/* Read the fourth timer in timer group 0 */
timer_cfg = REG_RD_VECT(iop_timer_grp, regi_iop_timer_grp0, rw_tmr_cfg, 3);

/* Check bit[27] of GIO out in the SAP module */
if ((REG_RD_INT_VECT(iop_sap_out, inst, rw_gio, 27) && 0x3ffffff) != 0) {
[...]
```

```
/* Write to the first timer of timer group 2 */
REG_WR_VECT(iop_timer_grp, regi_iop_timer_grp2, rw_tmr_cfg, 0, timer_cfg);

/* Write (0xffffffff) to the second general register of SPU0 */
REG_WR_INT_VECT(iop_spu, regi_iop_spu0, rw_r, 1, 0xffffffff);
```

## 6.1.2 Assembler Macros

The assembly instruction for reading from a register is:

```
rr [register address], Rd
```

The assembly instruction for writing to a register is:

```
rw Rs, [register address]
```

These two instructions are the same for the SPU and the MPU. There are however a number of instructions that are specific for each processor. See I/O Processor "Instruction Set Description" for more information.

These register macros evaluate to the address for each register:

```
REG_ADDR (scope, inst, reg)
```

```
REG_ADDR_VECT (scope, inst, reg, index)
```

<b>scope</b>	Name of the scope Typically: iop_[register bank]
<b>inst</b>	Base address of the instance of the scope Typically: iop_[register bank and number]
<b>reg</b>	Name of the register
<b>index</b>	Register number, in case there are several copies of the register

Example:

```
rr REG_ADDR(iop_fifo_out, iop_fifo_out0, rw_cfg), r1
rw r1, REG_ADDR(iop_fifo_out, iop_fifo_out0, rw_cfg)
rr REG_ADDR_VECT(iop_spu, iop_spu0, rw_r, 6), r6
rw r7, REG_ADDR_VECT(iop_timer_grp, iop_timer_grp0, rw_tmr_len, 0x1)
```

Macro for setting a numerical value in a register:

```
REG_FIELD (scope, reg, field, value)
```

The macro sets a bit field value of a register. (The value is a numeric value.)

<b>scope</b>	Name of the scope Typically: iop_[register bank]
<b>reg</b>	Name of the register
<b>field</b>	Name of the bit field in the register
<b>value</b>	Numeric value

Example:

```
; Enable the first timer in the first timer group
rwq REG_FIELD(iop_timer_grp, rw_cmd, en, 0x1), \
REG_ADDR(iop_timer_grp, iop_timer_grp0, rw_cmd)
```

(Note: since there is no masking procedure involved in the example above all other fields in this register (rw\_cmd) will be cleared (set to zero).)



```
; Put the masking bits for the field of the r_stat register in r13
moveq REG_MASK(iop_fifo_out_extra, r_stat, last), r13
```

The general register r13 will now have the value:

```
  3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
  1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-----+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0|
+-----+
```

```
; Mask out field (last) in the register
and r12, r13, r12
```

## 6.2 Registers and Ownership

In order for a processor (CPU, MPU, SPU0 or SPU1) to read or write to a register for a certain module it has to be the *owner* of that module.

Ownership is configured in the switch register bank **iop\_sw\_cfg**:

```

1. rw_crc_par0_owner..... 5
2. rw_crc_par1_owner..... 6
3. rw_dmc_in0_owner..... 7
4. rw_dmc_in1_owner..... 8
5. rw_dmc_out0_owner..... 9
6. rw_dmc_out1_owner..... 10
7. rw_fifo_in0_owner..... 11
8. rw_fifo_in0_extra_owner..... 12
9. rw_fifo_in1_owner..... 13
10. rw_fifo_in1_extra_owner..... 14
etc.

```

rw\_crc\_par0\_owner

```

 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-----+-----+-----+-----+-----+-----+-----+-----+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-----+-----+-----+-----+-----+-----+-----+-----+
                                                                    cfg

```

This register controls the owner of crc\_par0.

-----  
 cfg [1:0]

Constants:

```

cpu  = 0 -- cpu is owner
mpu  = 1 -- mpu is owner
spu0 = 2 -- spu0 is owner
spu1 = 3 -- spu1 is owner

```

Default constant: cpu

Selects owner of crc\_par0.  
 -----

Per default, the CPU is the owner of every module (and therefore also the owner of the switch where ownership is configured).

The ownership in the I/O-processor forms a hierarchy, where the SPUs end up at the bottom: one SPU can own the other, but not the MPU nor the CPU. The MPU can own either of the SPUs, but not the CPU. The CPU can own every processor and every other module, as it does per default.

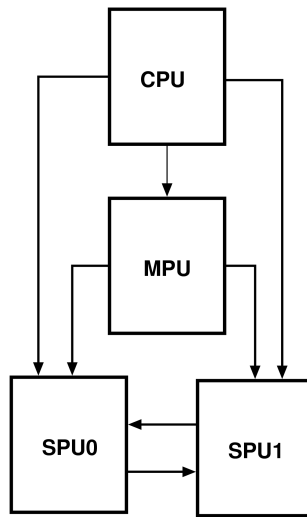


Figure 1. The Hierarchical Ownership

Example in C:

```

reg_iop_sw_cfg_rw_fifo_in0_extra_owner fifo_in0_extra_owner;

/* Configure the fifo_in0_extra to be owned by the SPU0 */

fifo_in0_extra_owner.cfg = regk_iop_sw_cfg_spu0; /* SPU0 is owner */
REG_WR(iop_sw_cfg, regi_iop_sw_cfg, rw_fifo_in0_extra_owner, fifo_in0_extra_owner);

```

# 7 The I/O-processor

Figure 2 below shows a schematic view of the modules and processors of the I/O-processor.

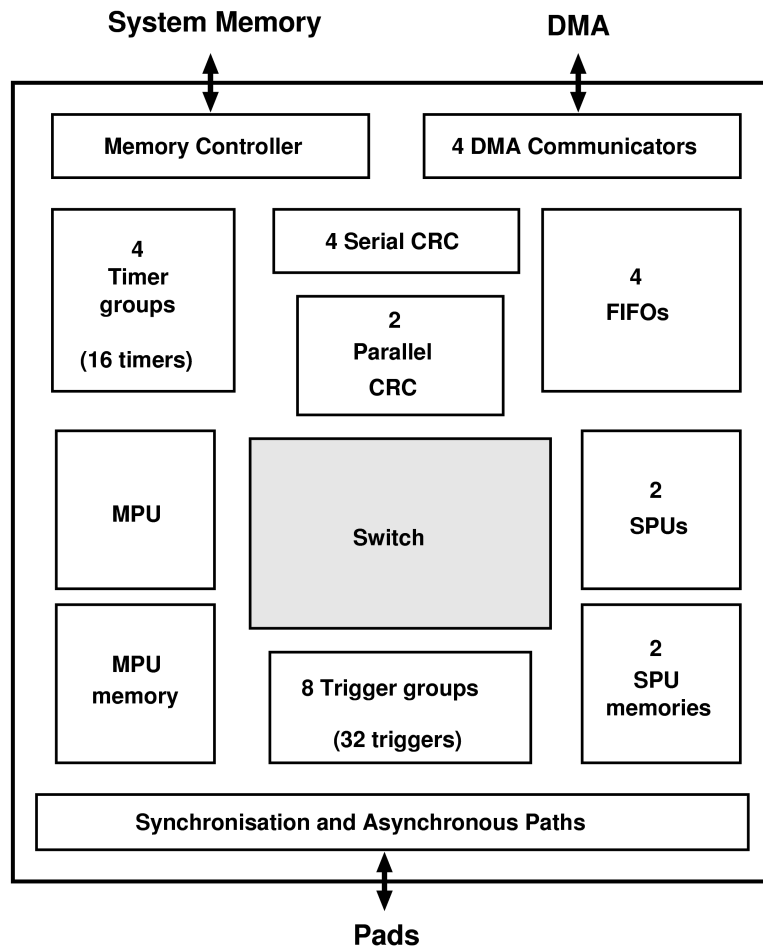


Figure 2. The I/O-processor

## 7.1 The MPU

The MPU (Master Processing Unit) is a conventional microprocessor.

<b>Speed</b>	The MPU has a clock speed of 200 MHz and executes all instructions except LW, SW and SWX (and unaligned 32-bit immediate instructions) in one clock cycle (5 ns).
<b>Interrupts</b>	The MPU has a total of 16 interrupts. The interrupt vector I0 has the highest priority, and I15 has the lowest priority. Nested interrupts are not supported.
<b>Threads</b>	The MPU has support for 4 threads.
<b>Memory</b>	The size of the MPU memory is 4096 bytes (512 x 64 bits).
<b>Registers</b>	There are 16 32-bit general registers.  There are also 25 special registers for configuring the interrupt addresses, threads, subroutines, and more.

The MPU has support for 32-bit immediate instructions, called extended instructions (the SPU has support for 32-bit immediate instructions in FSM mode, but not in sequential (assembly) mode).

Example in assembly for the MPU:

```
mpu_start:
; Move a 32 bit immediate value to register r0
movex 0x000000dc, r0
; Write value of r0 to SPU0's register r1
; (Note: this requires the MPU to be the owner of SPU0)
rw r0, REG_ADDR_VECT(iop_spu, iop_spu0, rw_r, 1)
halt
.end ; End of program
```

## 7.1.1 MPU Interrupts

Here is an example of interrupt handling in the MPU:

```
mpu_start:
; Disable interrupts
di

; An interrupt may be executed immediatly after the "di"
; instruction, to avoid this from happening add a "nop"
; instruction.
nop

; Configure the interrupt addresses
moveq intr_1, I15

; Configure the SPU0 for the I15 interrupt
rwx REG_STATE(iop_sw_mpu, rw_intr_grp3_mask, spu0_intr15, yes), \
    REG_ADDR(iop_sw_mpu, iop_sw_mpu, rw_intr_grp3_mask)

; Enable the interrupts
ei
; Note: it takes one MPU cycle (5 ns) after the "di" instruction
; before the interrupts are enabled. Add a "nop" instruction if
; this delay may be a problem in your code.

idle_loop:
    ba idle_loop
    nop ; Delay slot

; First interrupt routine

intr_1:
; Acknowledge the SPU0 interrupt
rwx REG_STATE(iop_sw_mpu, rw_ack_intr_grp3, spu0_intr15, yes), \
    REG_ADDR(iop_sw_mpu, iop_sw_mpu, rw_ack_intr_grp3)

; Activate the DMC_out0 interrupt
rwq REG_STATE(iop_dmc_out, rw_intr_mask, cmd_rdy, yes), \
    REG_ADDR(iop_dmc_out, iop_dmc_out0, rw_intr_mask)

; Acknowledge the dmc_out0 interrupts
rwq REG_STATE(iop_dmc_out, rw_ack_intr, cmd_rdy, yes), \
    REG_ADDR(iop_dmc_out, iop_dmc_out0, rw_ack_intr)

; Activate the DMC_out0 interrupt in the mpu
rwq REG_STATE(iop_sw_mpu, rw_intr_grp2_mask, dmc_out0, yes), \
    REG_ADDR(iop_sw_mpu, iop_sw_mpu, rw_intr_grp2_mask)

; Configure the interrupt address for the DMC_out0
moveq intr_2, I8

[Do things...]

    reti ; Return from interrupt routine
    nop ; Delay slot

; Second interrupt routine

intr_2:
; Acknowledge the dmc_out0 interrupts
rwq REG_STATE(iop_dmc_out, rw_ack_intr, cmd_rdy, yes), \
    REG_ADDR(iop_dmc_out, iop_dmc_out0, rw_ack_intr)

[Do things...]

    reti ; Return from interrupt routine
    nop ; Delay slot

    halt
.end ; End of program
```

## 7.2 The SPUs

There are two SPUs (Slave Processing Units) in the I/O-processor. They are identical and specifically designed to handle communication protocols.

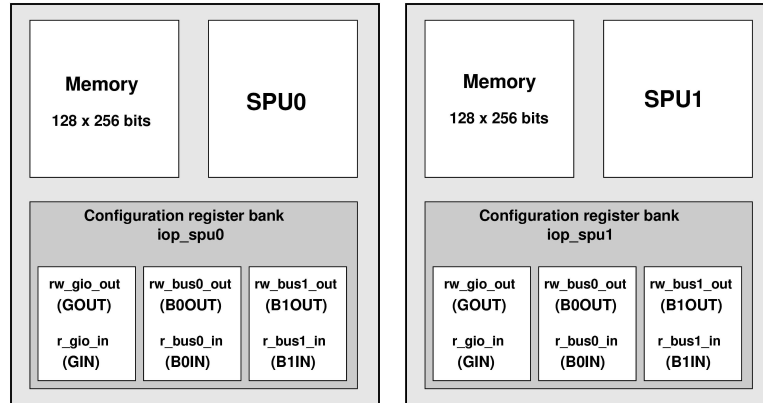


Figure 3. The Two SPUs

- Speed** The SPUs have a clock speed of 200 MHz and executes one instruction per clock cycle (5 ns).
- FSM code** The SPUs have, apart from ordinary assembly instructions, a special set of instructions called FSM (Finite State Machine) code. This enables the SPUs to execute finite state machines directly in the processor at the speed of 100 MHz; one FSM statement per 10 ns. (See 5.4 The FSM Mode.)
- Events** The SPUs have no support for interrupts, but have instead an equivalent in the FSM mode called events. Events can also be used to form hierarchical state machines.
- Memory** The size of the SPU memories are 4096 bytes (128 x 256 bits) each.
- Registers** There are 16 32-bit general registers.
- There are 16 special registers (for status information, configuration of FSM inputs, reading and writing to GIO and BUS0 and BUS1, and more).

Each SPU processor has its own register bank (iop\_spu0 and iop\_spu1) and this register bank is controlled by the owner of the SPU. Since an SPU can not own its own register bank, the SPUs have most of these registers mirrored internally in a register bank called *special registers*. The three 32-bit buses BUS0, BUS1 and GIO can be reached directly from each SPU by the use of the special registers: GIN and GOUT, B0IN and B0OUT, B1IN and B1OUT.

Here is a short example of sequential code (ordinary assembly, as opposed to FSM code) for the SPU:

```

spu0_start:
  move1 r0, 0x00dc, r0 ; Set the lowest 16 bits of register r0
  moveh r0, 0x0000, r0 ; Set the highest 16 bits of register r0

  ; Set output enables for the lowest 16 bits on the GIO bus
  ; This will set GIO[7:0] as outputs (and leave GIO[15:8] as inputs)
  rwq 0x00ff, REG_ADDR(iop_sw_spu, iop_sw_spu0, rw_gio_oe_set_mask_lo)

  ; Set initial values for the GIO bus
  move r0, GOUT

  halt
.end ; End of program

```

## 7.2.1 FSM code

Chapter 5.4 The FSM Mode gives a short introduction to the FSM mode of the SPUs. Here is a description of how to configure the inputs and outputs, together with an example.

The FSM inputs are configured in the register bank **iop\_spu**; registers **rw\_fsm\_inputs3\_0** and **rw\_fsm\_inputs7\_4**. Since the SPUs can not access this register bank they have instead their own set of special registers for configuring FSM inputs, called **FSM3\_0** and **FSM7\_4**.

There are a vast number of signals from almost every module in the I/O-processor that could be configured as inputs in FSM mode; any one of the 32 bits of the GIO inputs, FSM outputs (which could be used as flags), timer strobes, DMC flags, FIFO flags, MC flags, first bit of every general register, and more. However, there are limitations as to what signals you can choose for FSM inputs; some registers can, for instance, only be routed to some of the FSM inputs. See Internal Register for all configuration alternatives.

Here is an excerpt from the **rw\_fsm\_inputs3\_0 (FSM3\_0)** register:

```

rw_fsm_inputs3_0
  3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
  1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-----+-----+-----+-----+-----+-----+-----+-----+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+-----+-----+-----+-----+-----+-----+-----+-----+
src3 val3   src2 val2   src1 val1   src0 val0

-----
src0 [7:5]

Constants:
  gio_in   = 0 -- Input is selected from 32 gio_in
  flag     = 2 -- Input is selected from the four ALU flags
  reg_lo   = 2 -- Input is selected from bit[0] in general reg r7..r0
  xor      = 3 -- Input is selected from the four xor signals
  statin_lo = 4 -- Input is selected from 16 stat_in[15:0] (lo)
  gio_out  = 5 -- Input is selected from spu_gio_out[7:0]
  attn_lo  = 5 -- Input is selected from 8 attn[7:0] (lo)
  trigger  = 6 -- Input is selected from 32 Trigger strobes

Select source for fsm_input[0]. field:val0 is used to select which
bit to use.

```

Configuration example for SPU0:

```

; Configure the first bit to be GIO input [0],
; and the second bit to be status register of SPU0 bit[0]
; (which is the timer strobe of the first timer in timer
; group 0, see register r_stat_in in iop_spu bank)
move1 FSM3_0, \
  REG_STATE(iop_spu, rw_fsm_inputs3_0, src0, gio_in) | \
  REG_STATE(iop_spu, rw_fsm_inputs3_0, val0, 0) | \
  REG_STATE(iop_spu, rw_fsm_inputs3_0, src1, statin_lo) | \
  REG_STATE(iop_spu, rw_fsm_inputs3_0, val1, 0),
  FSM3_0

```

The same configuration as above written in C:

```
reg_iop_spu_rw_fsm_inputs3_0 fsm_inputs3_0;

fsm_inputs3_0.val0 = 0;
fsm_inputs3_0.src0 = regk_iop_spu_gio_in;
fsm_inputs3_0.val1 = 0;
fsm_inputs3_0.src1 = regk_iop_spu_statin_lo;

REG_WR(iop_spu, regi_iop_spu0, rw_fsm_inputs3_0, fsm_inputs3_0);
```

A total of 8 inputs can be configured. The selection of which 4 of these 8 to use as inputs in an FSM statement is done with the **inp** (input selector) flag. The table below shows the input selector mappings:

inp	input bit[3]	input bit[2]	input bit[1]	input bit[0]
0	fsm_in3	fsm_in2	fsm_in1	fsm_in0
1	fsm_in5	fsm_in4	fsm_in1	fsm_in0
2	fsm_in7	fsm_in6	fsm_in1	fsm_in0
3	fsm_in5	fsm_in4	fsm_in3	fsm_in2
4	fsm_in7	fsm_in6	fsm_in3	fsm_in2
5	fsm_in7	fsm_in6	fsm_in5	fsm_in4
6	fsm_in4	fsm_in2	fsm_in1	fsm_in0
7	fsm_in5	fsm_in2	fsm_in1	fsm_in0
8	fsm_in6	fsm_in2	fsm_in1	fsm_in0
9	fsm_in7	fsm_in2	fsm_in1	fsm_in0
10	fsm_in4	fsm_in3	fsm_in2	fsm_in0
11	fsm_in5	fsm_in3	fsm_in2	fsm_in0
12	fsm_in6	fsm_in3	fsm_in2	fsm_in0
13	fsm_in7	fsm_in3	fsm_in2	fsm_in0
14	z-flag	fsm_in2	fsm_in1	fsm_in0
15	z-flag	fsm_in6	fsm_in5	fsm_in7

As can be seen from the table above, some inputs are repeated more often than others; the first 4 inputs appear most frequently.

Any of the first 8 bits of SPU's GIO output register bits can be chosen as FSM outputs (note that the SPU's have their own GIO output and input registers). These first 8 bits can then be rerouted to anyone of the 32 bits of the GIO outputs, or rerouted to output enables for BUS0, BUS1 or GIO, or timers, and more. See register bank `iop_sw_cfg`, registers `rw_gio_out_grp0_cfg` to `rw_gio_out_grp7_cfg`. However, there are limitations as to, for instance, what timer strobes can be routed to the GIO outputs. See Internal Registers for all configuration alternatives.

Here is an excerpt from the `rw_gio_out_grp7_cfg` register:

```
rw_gio_out_grp7_cfg
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-----+-----+-----+-----+-----+-----+
|0 0 0 0 0 0 0 0| | | | | | | | | | | | | | | |
+-----+-----+-----+-----+-----+-----+
|          |          |          |          |
|  gio31   |  gio30   |  gio29   |  gio28   |
|  gio31_oe |  gio30_oe |  gio29_oe |  gio28_oe |
```

This register select output enable and sources for hw gio out [31:28].

-----  
gio31\_oe [23:22]

Constants:  
 spu0\_gio0 = 0 -- SPU0 gio out[0] is selected  
 spu0\_gio1 = 1 -- SPU0 gio out[1] is selected  
 spu1\_gio0 = 2 -- SPU1 gio out[0] is selected  
 spu1\_gio1 = 3 -- SPU1 gio out[1] is selected

Default constant: spu0\_gio0

Output enable for gio out[31].

-----  
gio31 [21:18]

Constants:  
 spu0\_gioout1 = 0 -- SPU0 gio out[1] is selected  
 spu1\_gioout1 = 1 -- SPU1 gio out[1] is selected  
 spu0\_gioout3 = 2 -- SPU0 gio out[3] is selected  
 spu1\_gioout3 = 3 -- SPU1 gio out[3] is selected  
 spu0\_gioout5 = 4 -- SPU0 gio out[5] is selected  
 spu1\_gioout5 = 5 -- SPU1 gio out[5] is selected  
 spu0\_gioout7 = 6 -- SPU0 gio out[7] is selected  
 spu1\_gioout7 = 7 -- SPU1 gio out[7] is selected  
 sdp\_out0 = 8 -- SDP0 out is selected  
 sdp\_out1 = 9 -- SDP1 out is selected  
 timer\_grp0\_strb3 = 10 -- Timer\_grp0 timer 3 is selected  
 timer\_grp1\_strb3 = 11 -- Timer\_grp1 timer 3 is selected  
 timer\_grp2\_strb3 = 12 -- Timer\_grp2 timer 3 is selected  
 timer\_grp3\_strb3 = 13 -- Timer\_grp3 timer 3 is selected  
 spu0\_gioout31 = 14 -- SPU0 gio out[31] is selected  
 spu1\_gioout31 = 15 -- SPU1 gio out[31] is selected

Default constant: spu0\_gioout1

A configuration example in C:

```
reg_iop_sw_cfg_rw_gio_out_grp7_cfg  gio_out_grp7_sw_cfg;

gio_out_grp7_sw_cfg = REG_RD(iop_sw_cfg, regi_iop_sw_cfg, rw_gio_out_grp7_cfg);
gio_out_grp7_sw_cfg.gio31 = regk_iop_sw_cfg_spu0_gioout3;
gio_out_grp7_sw_cfg.gio30 = regk_iop_sw_cfg_spu0_gioout2;
gio_out_grp7_sw_cfg.gio29 = regk_iop_sw_cfg_spu0_gioout1;
gio_out_grp7_sw_cfg.gio28 = regk_iop_sw_cfg_spu0_gioout0;
REG_WR(iop_sw_cfg, regi_iop_sw_cfg, rw_gio_out_grp7_cfg, gio_out_grp7_sw_cfg);
```

Here is an FSM (Finite State Machine) code example for the SPU (the configuration examples above are used in this example):

```
.seq ; Sequential code start
seq_init:
    fsmq fsm_label_0
    nop ; Delay slot

.align 64 ; Align the FSM code at a 64-bit boundary
.fsm ; Tell the assembler that we switch to FSM code
fsm_label_0:
    inp = 0
    outp = 0
    ?_?_1_0 : 1_?_1_? : fsm_label_1
    ; If GIO in bit[0] (first FSM input) is 0 and the timer strobe is high
    ; (second FSM input), set GIO out bit[31] to 1 and GIO out bit[29] to 1,
    ; and goto label "fsm_label_1"
    ?_?_1_1 : ?_1_?_? : fsm_label_1
    ; If GIO in bit[0] is 1 and the timer strobe is high, set GIO out bit[30]
    ; to 1, and goto label "fsm_label_1"
    ?_?_?_? : ?_?_?_? : fsm_label_2 ; Else goto label "fsm_label_2"

fsm_label_1:
    inp = 1
    outp = 0
    ?_?_0_? : ?_?_?_? : fsm_label_0
    ; Wait until timer strobe goes low, then goto label "fsm_label_0"

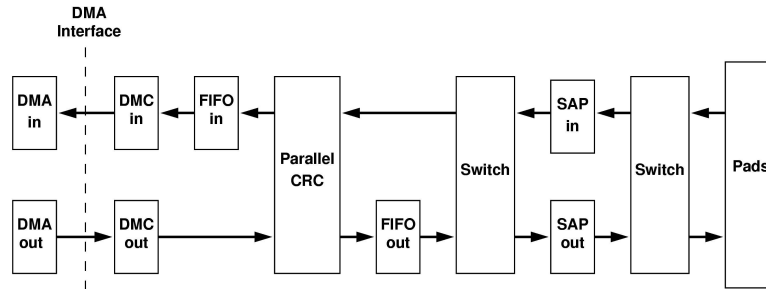
fsm_label_2:
    go_seq ; Switch to sequential (assembly) mode
    only spu0_seq_halt ; A sequential (assembly) label

.seq
spu0_seq_halt:
    halt
.end
```

## 7.3 The Data Flow

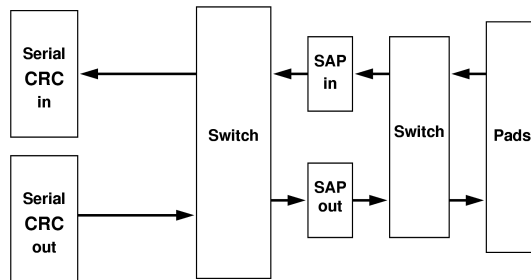
The I/O-processor consists of several modules which controls the data flow. There are two 32-bit data buses. The data width for these could be 8, 16, 24 or 32 bits. There is also a 32-bit GIO, General Input/Output, bus, which can handle serial communication and various protocol signals.

There are two input and output parallel data channels or paths, pdp0 and pdp1. Figure 4 gives a schematic view of one of them.



**Figure 4. Parallel Data Paths, PDP, for BUS0 or BUS1**

There is no parallel-to-serial or serial-to-parallel converter in the I/O-processor. For a serial communication protocol data has to be read, and written, from either the FIFO or the DMC. The serial data has to be handled by the use of logical bit shift instructions. There is however two serial CRC modules that can either check or generate any given CRC polynomial.



**Figure 5. Serial Data Paths for GIO**

## 7.4 The DMC

There are two input and two output DMC (DMA Communicator) modules. The DMC handles the communication with the ETRAX FS DMA.

The DMA is the unit (outside of the I/O-processor) that performs direct memory access. Data is transported between the memory and the DMA in groups of 32 bytes.

The ETRAX FS DMA has 10 channels, 5 input channels and 5 output channels, and 4 of these channels can be allocated to the I/O-processor (See DMA Connection):

- dma2 = dmc\_out0
- dma3 = dmc\_in0
- dma4 = dmc\_out1
- dma5 = dmc\_in1

Here is an excerpt from the `rw_ack_intr` register of the DMC in:

```
rw_ack_intr
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-----+-----+-----+-----+-----+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-----+-----+-----+-----+-----+
                                     | | | | | data_md
                                     | | | | |  ctxt_md
                                     | | | | |  group_md
                                     | | | | |  cmd_rdy
                                     | | | | |  sth
                                     | | | | |  full
```

Acknowledge interrupts. Interrupts from DMA Communicator in-channel.

-----  
data\_md [0]

Constants:  
yes = 1 -- Acknowledge interrupt  
no = 0 -- No operation

Acknowledge data\_md interrupt  
-----

Example in assembly code:

```
; Clear dmc_in0 data_md interrupt
rrw REG_STATE(iop_dmc_in, rw_ack_intr, data_md, yes), \
    REG_ADDR(iop_dmc_in, iop_dmc_in0, rw_ack_intr)
```

The DMA has meta data fields where information of the transfer can be written and read:

```
; Read the context meta data fields from DMC out 0 (DMA channel 2)
rr REG_ADDR(iop_dmc_out, iop_dmc_out0, r_ctxt_descr), r1
rr REG_ADDR(iop_dmc_out, iop_dmc_out0, r_ctxt_descr_md1), r2
rr REG_ADDR(iop_dmc_out, iop_dmc_out0, r_ctxt_descr_md2), r3
```

A DMA transfer can also be started from the I/O-processor. Here is an excerpt from the `rw_stream_cmd` register of the DMC in:

```
rw_stream_cmd
 3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-----+-----+-----+-----+
|0 0 0 0 0 0 0 0| |0 0 0 0 0 0|
+-----+-----+-----+-----+
                                n                cmd
```

Command register for the DMA in-channel.

-----  
cmd [9:0]

```
Constants:
store_descr = 0 -- Store descriptor(nop)
store_md    = 1 -- Store meta data
store_c     = 2 -- Store context descriptor
store_g     = 4 -- Store group descriptor
array      = 8 -- When next_en, stop at eol.
next_en    = 16 -- Find the next enabled descriptor
copy_next  = 16 -- Make the down pointer of the upper group, point to the current next group.
next_pkt   = 16 -- Go to the first data descriptor of the next packet
dis_c      = 16 -- Disable the current context descriptor.
dis_g      = 32 -- Disable the current group descriptor.
copy_up    = 32 -- Copy current up pointer when loading the next group descriptor
save_down  = 32 -- Make the down pointer of the upper group, point to current group
save_up    = 32 -- Make the up pointer of the lower group, point to current group
update_down = 32 -- Make the down pointer of the group point to the loaded context
restore    = 32 -- Restore context descriptor
burst     = 32 -- Start bursting
set_reg    = 80 -- Set reg
ack_pkt    = 256 -- Acknowledge packet(continue burst)
load_d     = 320 -- Load data descriptor
set_w_size1 = 400 -- Set word size to 1 byte
set_w_size2 = 416 -- Set word size to 2 bytes
set_w_size4 = 448 -- Set word size to 4 bytes
load_c     = 512 -- Load context descriptor
load_c_next = 576 -- Load next context descriptor
load_c_n   = 640 -- Load n:th context descriptor
load_g     = 768 -- Load group descriptor
load_g_next = 832 -- Load next group descriptor
load_g_up  = 896 -- Load upper group descriptor
load_g_down = 960 -- Load lower group descriptor
```

Write a command to the DMA.

-----  
Example code showing how to start, or restart, a DMA transfer:

```
; Give "ack_pkt" and "restore" to DMC in 0,
; (DMA channel 3) to start, or restart, a burst
rwq REG_STATE(iop_dmc_in, rw_stream_cmd, cmd, ack_pkt) | \
  REG_STATE(iop_dmc_in, rw_stream_cmd, cmd, restore), \
  REG_ADDR(iop_dmc_in, iop_dmc_in0, rw_stream_cmd)
```



```

-----
byte_order [4:3]

Constants:
  order8 = 0 -- 8-bit mode (3 2 1 0)
  order16 = 1 -- 16-bit mode (2 3 0 1)
  order24 = 2 -- 24-bit mode (3 0 1 2)
  order32 = 3 -- 32-bit mode (0 1 2 3)

Default constant: order8

Controls the byte swapping mechanism.

-----
free_lim [2:0]

Default value: 4

Set the number of bytes which must be free in the FIFO before the
FIFO will generate an interrupt or signal the parallel in-data
interface that it can receive data.

-----

```

Here is a C example of a 16-bit parallel configuration, with no byte swap:

```

reg_iop_fifo_out_rw_cfg  fifo_out_cfg;

fifo_out_cfg = REG_RD(iop_fifo_out, regi_iop_fifo_out0, rw_cfg);
/* 16 bits width */
fifo_out_cfg.mode = regk_iop_fifo_out_size16;
/* Do not disable FIFO after LAST flag */
fifo_out_cfg.last_dis_dif_in = regk_iop_fifo_out_no;
/* Strobe FIFO on positive edge */
fifo_out_cfg.trig = regk_iop_fifo_out_pos;
/* Do not swap the bytes */
fifo_out_cfg.byte_order = regk_iop_fifo_out_order8;
/* Receive more data when 2 bytes are free */
fifo_out_cfg.free_lim = 2;
REG_WR(iop_fifo_out, regi_iop_fifo_out0, rw_cfg, fifo_out_cfg);

```

Each FIFO has one extra register bank. This extra register bank makes it for instance possible for the CPU to own the FIFO module in order to configure the FIFO, and one processor in the I/O-processor to own the FIFO extra register bank to read status information, or any other combination of ownership between the four processors.

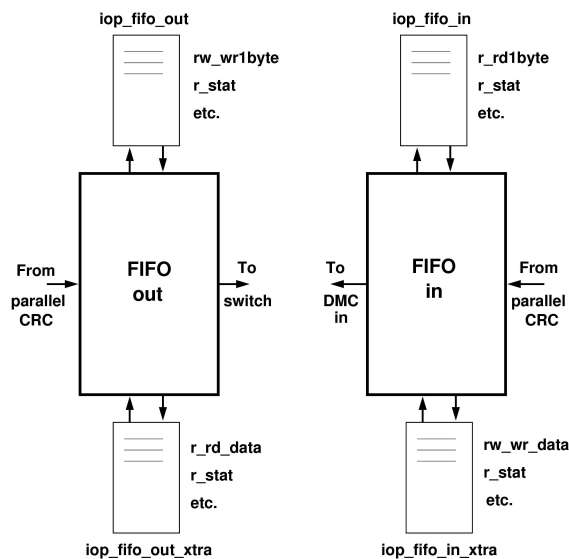


Figure 6. The FIFOs and their register banks

The extra register banks can also be used for sending data from one processor to another in the I/O-processor.

FIFO	The Owner of the register bank can...
<b>iop_fifo0_out</b>	Configure the FIFO (and flush the FIFO) Enable and disable the FIFO Write data to fifo0_out Set last flag Read FIFO status information Read and acknowledge interrupts
<b>iop_fifo0_out_extra</b>	Read data from fifo0_out Read FIFO status information Read and acknowledge interrupts
<b>iop_fifo0_in</b>	Configure the FIFO (and flush the FIFO) Enable and disable the FIFO Read data from fifo0_in Set last flag Read FIFO status information Read and acknowledge interrupts
<b>iop_fifo0_in_extra</b>	Write data to fifo0_in Read FIFO status information Read and acknowledge interrupts
<b>iop_fifo1_out</b>	Configure the FIFO (and flush the FIFO) Enable and disable the FIFO Write data to fifo1_out Set last flag Read FIFO status information Read and acknowledge interrupts
<b>iop_fifo1_out_extra</b>	Read data from fifo1_out Read FIFO status information Read and acknowledge interrupts
<b>iop_fifo1_in</b>	Configure the FIFO (and flush the FIFO) Enable and disable the FIFO Read data from fifo1_in Set last flag Read FIFO status information Read and acknowledge interrupts
<b>iop_fifo1_in_extra</b>	Write data to fifo1_in Read FIFO status information Read and acknowledge interrupts

Flushing and resetting a FIFO is done by writing to the **rw\_cfg** register. In order not to overwrite the configuration of the FIFO, a read must first be made. If the FIFO is enabled, it is wise to disable it before it is flushed.

Example in assembly code:

```

; Read the value of the rw_cfg register
rr REG_ADDR(iop_fifo_out, iop_fifo_out0, rw_cfg), r1

; Disable the FIFO
rwq REG_STATE(iop_fifo_out, rw_ctrl, dif_out_en, no) | \
REG_STATE(iop_fifo_out, rw_ctrl, dif_in_en, no), \
REG_ADDR(iop_fifo_out, iop_fifo_out0, rw_ctrl)

; Write the value back, and thereby flush the FIFO
rw r1, REG_ADDR(iop_fifo_out, iop_fifo_out0, rw_cfg)

```

(Since the result from a register read operation is delayed one cycle, at least one instruction needs to be placed between the **rr** and **rw** instruction.)

If the FIFO out is not flushed before a new DMA transaction is started, the data remaining in the FIFO from the last transaction will be the first data on the output.



```

-----
byte0_ext_src [4:2]

Constants:
  gio1      = 0 -- Use gio_in[1]
  gio6      = 1 -- Use gio_in[6]
  gio7      = 2 -- Use gio_in[7]
  gio18     = 3 -- Use gio_in[18]
  gio19     = 4 -- Use gio_in[19]
  gio23     = 5 -- Use gio_in[23]
  timer_grp0_tmr3 = 6 -- Use Timer group 0, Timer 3
  timer_grp3_tmr3 = 7 -- Use Timer group 3, Timer 3

Default constant: gio1

Select source for external synchronization (field:byte0_sel equals
state:byte0_sel:ext_clk_200 or state:byte0_sel:no_del_ext_clk_200)
or Timer source (field:byte0_sel equals
state:byte0_sel:timer_200_clk).

-----
byte0_sel [1:0]

Constants:
  timer_clk_200      = 0 -- Use timer + 200 MHz flip-flops for synchronization
  two_clk_200        = 2 -- Use two 200 MHz flip-flops for synchronization
  no_del_ext_clk_200 = 1 -- Use external signal and two 200 MHz flip-flop
  ext_clk_200        = 3 -- Use ext and two 200 MHz flip-flops

Default constant: two_clk_200

Select synchronization path for byte0, BUS0[7:0].

-----

```

Example code in C for BUS0 in SAP in:

```

reg_iop_sap_in_rw_bus0_sync  sap_in_bus0_sync;

/* byte 0 */
sap_in_bus0_sync = REG_RD(iop_sap_in, regi_iop_sap_in, rw_bus0_sync);
/* Add one extra 200 MHz flip-flop */
sap_in_bus0_sync.byte0_delay = regk_iop_sap_in_yes;
/* Synchronize on positive edge for the external source */
sap_in_bus0_sync.byte0_edge = regk_iop_sap_in_pos;
/* Use timer 3 of timer group 0 as external clock source */
sap_in_bus0_sync.byte0_ext_src = regk_iop_sap_in_timer_grp0_tmr3;
/* Use timer and a 200 MHz flip-flop */
sap_in_bus0_sync.byte0_sel = regk_iop_sap_in_timer_clk_200;
REG_WR(iop_sap_in, regi_iop_sap_in, rw_bus0_sync, sap_in_bus0_sync);

```

Here is an excerpt from **rw\_gio[31:0]** register of the SAP in:

```

rw_gio[31:0]

  3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1
  1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-----+-----+-----+-----+-----+-----+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|
+-----+-----+-----+-----+-----+-----+
|                                     | | | | | sync_sel
|                                     | | | | | sync_ext_src
|                                     | | | | | sync_edge
|                                     | | | | | delay
|                                     | | | | |
+-----+-----+-----+-----+-----+
|                                     | | | | |
|                                     | | | | |
|                                     | | | | |
|                                     | | | | |
+-----+-----+-----+-----+
logic

```

Type: rw (vector of 32 instances 0..31)

This vector register controls the synchronization and NOT/AND/OR logic of each input GIO.

```

-----
logic [9:8]

Constants:
  none = 0 -- No extra logic
  inv  = 1 -- The GIO[x] signal is inverted
  and  = 2 -- The GIO[x] signal is and:ed with GIO[x+1]
  or   = 3 -- The GIO[x] signal is or:ed with GIO[x+1]

Default constant: none

NOT/AND/OR logic for GIO. Each synchronized GIO can be and:ed or
or:ed with another synchronized GIO input. e.g., GIO[0] can be
and:ed with GIO[1]. Note: GIO[31] uses GIO[0] as operand.

-----

```

```

-----
delay [7]

Constants:
  no = 0 -- No extra 200 MHz flip-flop
  yes = 1 -- Add one extra 200 MHz flip-flop

Default constant: no

Delay GIO with one 200 MHz flip-flop, after synchronization.
-----

sync_edge [6:5]

Constants:
  pos = 1 -- Data is synchronized on positive edge
  neg = 2 -- Data is synchronized on negative edge
  pos_neg = 3 -- Data is synchronized on both positive and negative edge

Default value: 0

Selects edge of external source for synchronization, only used if
field:sync_sel equals state:sync_sel:ext_clk_200 or
state:sync_sel:no_del_ext_clk_200.
-----

sync_ext_src [4:2]

Constants:
  timer_grp1_tmr3 = 4 -- Use Timer group 1, Timer 3
  timer_grp2_tmr3 = 5 -- Use Timer group 2, Timer 3
  timer_grp0_tmr3 = 6 -- Use Timer group 0, Timer 3
  timer_grp3_tmr3 = 7 -- Use Timer group 3, Timer 3
  gio1 = 0 -- Use gio_in[1]
  gio6 = 1 -- Use gio_in[6]
  gio7 = 2 -- Use gio_in[7]
  gio18 = 3 -- Use gio_in[18]
  gio5 = 4 -- Use gio_in[5]
  gio13 = 5 -- Use gio_in[13]
  gio21 = 6 -- Use gio_in[21]
  gio29 = 7 -- Use gio_in[29]

Default constant: gio1

Select source for external synchronization (field:sync_sel equals
state:sync_sel:ext_clk_200 or state:sync_sel:no_del_ext_clk_200) or
Timer source (field:sync_sel equals state:sync_sel:timer_200_clk).
-----

sync_sel [1:0]

Constants:
  timer_clk_200 = 0 -- Use Timer, followed by two 200 MHz flip-flops
  two_clk_200 = 2 -- Use two 200 MHz flip-flops for synchronization
  no_del_ext_clk_200 = 1 -- Use external signal and two 200 MHz flip-flop
  ext_clk_200 = 3 -- Use ext and two 200 MHz flip-flops

Default constant: two_clk_200

Select synchronization path for GIO.
-----

```

Example code in C for GIO bit[0] in SAP in:

```

reg_iop_sap_in_rw_gio sap_in_gio;

sap_in_gio = REG_RD_VECT(iop_sap_in, regi_iop_sap_in, rw_gio, 0);
/* Use timer 3 in timer group 0 as external clock source */
sap_in_gio.sync_ext_src = regk_iop_sap_in_timer_grp0_tmr3;
/* Use timer and two 200 MHz flip-flops */
sap_in_gio.sync_sel = regk_iop_sap_in_timer_clk_200;
/* Synchronize on positive edge for the external clock source */
sap_in_gio.sync_edge = regk_iop_sap_in_pos;
REG_WR_VECT(iop_sap_in, regi_iop_sap_in, rw_gio, 0, sap_in_gio);

```







Example code in C for GIO in SAP out:

```
reg_iop_sap_out_rw_gio  sap_out_gio;

/* Use internal I/O-proc synchronization
   (it uses an additional 200 MHz flip-flop
   to synchronize with the 200 MHz clock) */
sap_out_gio = REG_RD_VECT(iop_sap_out, regi_iop_sap_out, rw_gio, 5);
/* Do not invert the clock, use positive edge */
sap_out_gio.out_clk_inv = regk_iop_sap_out_no;
/* Use a 200 MHz flip-flop */
sap_out_gio.out_clk_sel = regk_iop_sap_out_clk_200;
REG_WR_VECT(iop_sap_out, regi_iop_sap_out, rw_gio, 5, sap_out_gio);
```

## 7.8 The Switch

There is one switch module that connects the various modules in the data flow. The switch module is also responsible for mapping signals to output pads.

The switch module has five register banks, and each processor has its own register bank:

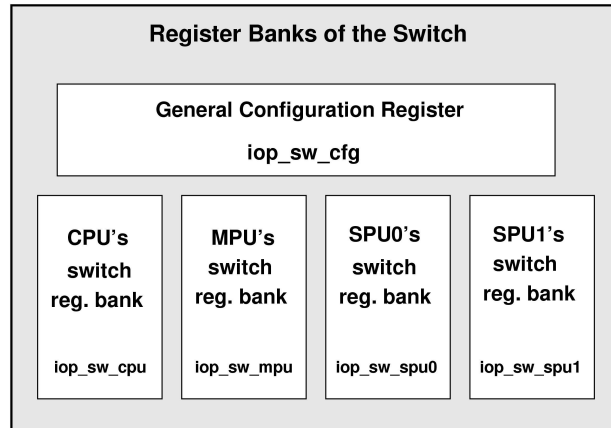


Figure 7. Register Banks of the Switch

Data can be written directly to data registers in the switch. The data register values are ORed with the data flow from the parallel data path, PDP. There are also registers to block signals coming from the parallel data path.

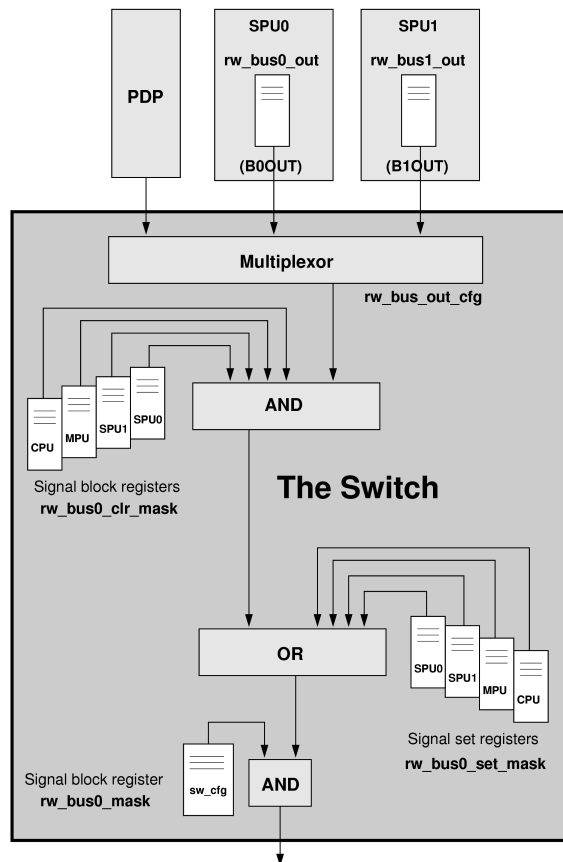


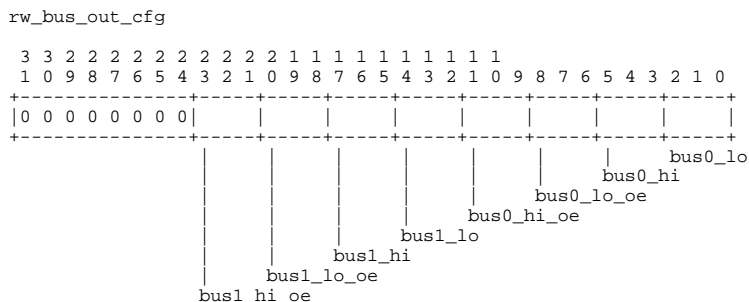
Figure 8. BUS0 Output Signals Through the Switch

Figure 7 shows how BUS0 signals are routed through the switch module. BUS1 signals are treated exactly the same as BUS0 signals.

The routing of output enables for BUS0 and BUS1 have a similar layout through the switch except for the addition that more signals can be routed through the multiplexor, so more signals can be used as output enables.

The data flow between the I/O-processor and main memory is mainly handled by the ETRAX FS DMA, in which case it flows through the PDP (Parallel Data Path) in Figure 7 above. (Data can also be sent to the I/O-processor from the CPU directly to data registers in the FIFOs.) Each processor has its own register bank in the switch where data can be written or where data can be blocked. Note that there is a second AND gate where all data signals can be blocked.

Several signals can be used as output enables for the parallel data signals (handled in bytes), and the BUS0 and BUS1 bytes can be shifted or rotated. Here are excerpts from some registers in the **iop\_sw\_cfg** register bank:



Default value: 0x00000000

This register select HW sources for BUS0, BUS1, BUS0 OE and BUS1 oe

-----  
bus1\_hi\_oe [23:21]

- Constants:
- spu0\_gio2 = 0 -- Use SPU0 GIO[2] as OE
  - spu0\_gio1 = 1 -- Use SPU0 GIO[1] as OE
  - spu1\_gio2 = 2 -- Use SPU1 GIO[2] as OE
  - spu1\_gio1 = 3 -- Use SPU1 GIO[1] as OE
  - timer\_grp0\_tmrl = 4 -- Use Timer group 0, Timer 1 as OE
  - timer\_grp1\_tmrl = 5 -- Use Timer group 1, Timer 1 as OE
  - timer\_grp2\_tmrl = 6 -- Use Timer group 2, Timer 1 as OE
  - timer\_grp3\_tmrl = 7 -- Use Timer group 3, Timer 1 as OE

Default constant: spu0\_gio2

Output enable register for BUS1[31:16].

-----  
bus1\_lo\_oe [20:18]

- Constants:
- spu0\_gio2 = 0 -- Use SPU0 GIO[2] as OE
  - spu0\_gio1 = 1 -- Use SPU0 GIO[1] as OE
  - spu1\_gio2 = 2 -- Use SPU1 GIO[2] as OE
  - spu1\_gio1 = 3 -- Use SPU1 GIO[1] as OE
  - timer\_grp0\_tmrl = 4 -- Use Timer group 0, Timer 1 as OE
  - timer\_grp1\_tmrl = 5 -- Use Timer group 1, Timer 1 as OE
  - timer\_grp2\_tmrl = 6 -- Use Timer group 2, Timer 1 as OE
  - timer\_grp3\_tmrl = 7 -- Use Timer group 3, Timer 1 as OE

Default constant: spu0\_gio2

Output enable register for BUS1[15:0].

-----  
bus1\_hi [17:15]

- Constants:
- pdp\_out0\_lo = 0 -- Parallel datapath 0 [15:0]
  - pdp\_out0\_hi = 1 -- Parallel datapath 0 [31:16]
  - pdp\_out1\_lo = 2 -- Parallel datapath 1 [15:0]
  - pdp\_out1\_hi = 3 -- Parallel datapath 1 [31:16]
  - pdp\_out0\_lo\_rot8 = 4 -- Parallel datapath 0 rotated 8bits {[7:0], [15:8]}
  - pdp\_out1\_hi\_rot8 = 5 -- Parallel datapath 1 rotated 8bits {[23:16], [31:24]}
  - spu1\_bus\_out0\_hi = 6 -- SPU1 BUS0 out [31:16]
  - spu0\_bus\_out1\_lo = 7 -- SPU0 BUS1 out [15:0]

Default constant: pdp\_out0\_lo

BUS1[31:16].

-----

```

bus1_lo [14:12]

Constants:
pdp_out0_lo      = 0 -- Parallel datapath 0 [15:0]
pdp_out0_hi      = 1 -- Parallel datapath 0 [31:16]
pdp_out1_lo      = 2 -- Parallel datapath 1 [15:0]
pdp_out1_hi      = 3 -- Parallel datapath 1 [31:16]
pdp_out1_lo_rot8 = 4 -- Parallel datapath 1 rotated 8bits {[7:0], [15:8]}
pdp_out0_hi_rot8 = 5 -- Parallel datapath 0 rotated 8bits {[23:16], [31:24]}
spu1_bus_out0_lo = 6 -- SPU1 BUS0 out [15:0]
spu0_bus_out1_hi = 7 -- SPU0 BUS1 out [31:16]

Default constant: pdp_out0_lo

BUS1[15:0].
-----

```

You can also route an outgoing BUS1 as an input source for SPU0's BUS0 (or SPU1's BUS0), or vice versa:

```

rw_spu0_cfg

 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9
-----+-----+-----+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0| | |
-----+-----+-----+
                                         | bus0_in
                                         bus1_in

```

This register controls the source of input buses into SPU0.

```

-----
bus1_in [3:2]

Constants:
bus0      = 0 -- Data from external BUS0
bus1      = 1 -- Data from external BUS1
pdp_out0  = 2 -- Parallel datapath 0 (Out)
pdp_out1  = 3 -- Parallel datapath 1 (Out)

Default constant: bus0

Select source for BUS1.
-----

```

```

-----
bus0_in [1:0]

Constants:
bus0      = 0 -- Data from external BUS0
bus1      = 1 -- Data from external BUS1
pdp_out0  = 2 -- Parallel datapath 0 (Out)
pdp_out1  = 3 -- Parallel datapath 1 (Out)

Default constant: bus0

Select source for BUS0.
-----

```



Figure 9 below shows how the 32-bit GIO bus is routed in the switch module.

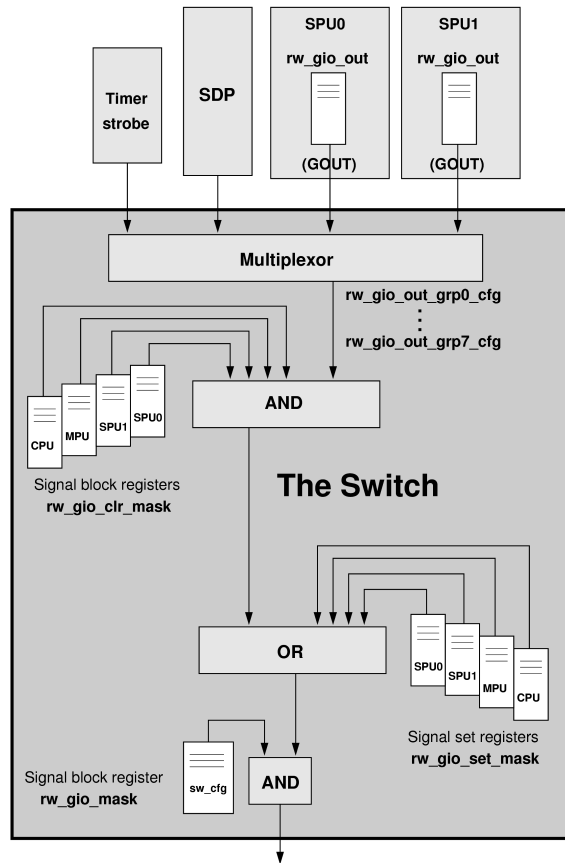


Figure 9. GIO Output Signals Through the Switch

There are a number of signals from the SPUs, the timer groups, and the SDP (Serial Data Path) that can be routed through the switch to the GIO out:

```
rw_gio_out_grp0_cfg
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-----+-----+-----+-----+-----+-----+
|0 0 0 0 0 0 0| | | | | | | | | | | | | | | | | |
+-----+-----+-----+-----+-----+-----+
                |   |   |   |   | | | |
                |gio3| |gio2| |gio1| |gio0|
                |gio3_oe| |gio2_oe| |gio1_oe| |gio0_oe|
```

This register select HW sources for GIO[3:0] and GIO[3:0] oe

```
-----
gio3_oe [23:22]

Constants:
  spu0_gio0 = 0 -- SPU0 GIO[0] is selected
  spu0_gio1 = 1 -- SPU0 GIO[1] is selected
  spu1_gio0 = 2 -- SPU1 GIO[0] is selected
  spu1_gio1 = 3 -- SPU1 GIO[1] is selected

Default constant: spu0_gio0

Output enable for GIO[3].
-----
```

-----  
gio3 [21:18]

Constants:

spu0_gioout1	= 0	-- SPU0 GIO[1] is selected
spu1_gioout1	= 1	-- SPU1 GIO[1] is selected
spu0_gioout3	= 2	-- SPU0 GIO[3] is selected
spu1_gioout3	= 3	-- SPU1 GIO[3] is selected
spu0_gioout5	= 4	-- SPU0 GIO[5] is selected
spu1_gioout5	= 5	-- SPU1 GIO[5] is selected
spu0_gioout7	= 6	-- SPU0 GIO[7] is selected
spu1_gioout7	= 7	-- SPU1 GIO[7] is selected
sdp_out0	= 8	-- SDP0 out is selected
sdp_out1	= 9	-- SDP1 out is selected
timer_grp0_strb3	= 10	-- Timer group 0, Timer 3 is selected
timer_grp1_strb3	= 11	-- Timer group 1, Timer 3 is selected
timer_grp2_strb3	= 12	-- Timer group 2, Timer 3 is selected
timer_grp3_strb3	= 13	-- Timer group 3, Timer 3 is selected
spu0_g6	= 14	-- SPU0 GIO[6] is selected
spu1_g6	= 15	-- SPU1 GIO[6] is selected

Default constant: spu0\_gioout1

GIO[3].  
-----

## 7.8.1 Set and Block Signals

Examples:

Let through the first two bytes of data for BUS0 for the parallel data path, and block the two upper bytes of BUS0 (the output enables can only be set and cleared in groups of bytes):

### CPU

```
reg_iop_sw_cfg_rw_bus0_mask      sw_cfg_bus0_mask;
reg_iop_sw_cpu_rw_bus0_set_mask  sw_cpu_bus0_set_mask;
reg_iop_sw_cpu_rw_bus0_clr_mask  sw_cpu_bus0_clr_mask;

reg_iop_sw_cfg_rw_bus0_oe_mask   sw_cfg_bus0_oe_mask;
reg_iop_sw_cpu_rw_bus0_oe_set_mask sw_cpu_bus0_oe_set_mask;
reg_iop_sw_cpu_rw_bus0_oe_clr_mask sw_cpu_bus0_oe_clr_mask;

reg_iop_sw_cfg_rw_bus_out_cfg    bus_out_sw_cfg;

reg_iop_sw_cfg_rw_pdp0_cfg       pdp0_sw_cfg;

/* BUS0 signals */

/* BUS0 out block register--block the two upper bytes */
sw_cfg_bus0_mask = 0x0000ffff;
REG_WR(iop_sw_cfg, regi_iop_sw_cfg, rw_bus0_mask, sw_cfg_bus0_mask);

/* BUS0 set signals register--clear all signals */
sw_cpu_bus0_set_mask = 0x00000000;
REG_WR(iop_sw_cpu, regi_iop_sw_cpu, rw_bus0_set_mask, sw_cpu_bus0_set_mask);

/* BUS0 block signals register--no need to block the upper bytes here
since they are already blocked */
sw_cpu_bus0_clr_mask = 0x00000000;
REG_WR(iop_sw_cpu, regi_iop_sw_cpu, rw_bus0_clr_mask, sw_cpu_bus0_clr_mask);

/* Output enables for BUS0 */

/* Do not block any output enables */
sw_cfg_bus0_oe_mask = 0xf;
REG_WR(iop_sw_cfg, regi_iop_sw_cfg, rw_bus0_oe_mask, sw_cfg_bus0_oe_mask);

/* Block all signals for output enables coming from the multiplexer */
sw_cpu_bus0_oe_clr_mask = 0xf;
REG_WR(iop_sw_cpu, regi_iop_sw_cpu, rw_bus0_oe_clr_mask, sw_cpu_bus0_oe_clr_mask);

/* Set the output enables for the two lowest bytes */
sw_cpu_bus0_oe_set_mask = 0x3;
REG_WR(iop_sw_cpu, regi_iop_sw_cpu, rw_bus0_oe_set_mask, sw_cpu_bus0_oe_set_mask);

/* Configure the lowest word of BUS0 for the parallel data path */
bus_out_sw_cfg = REG_RD(iop_sw_cfg, regi_iop_sw_cfg, rw_bus_out_cfg);
bus_out_sw_cfg.bus0_hi = regk_iop_sw_cfg_pdp_out0_hi;
bus_out_sw_cfg.bus0_lo = regk_iop_sw_cfg_pdp_out0_lo;
REG_WR(iop_sw_cfg, regi_iop_sw_cfg, rw_bus_out_cfg, bus_out_sw_cfg);

/* Parallel Data Path Config */
pdp0_sw_cfg = REG_RD(iop_sw_cfg, regi_iop_sw_cfg, rw_pdp0_cfg);
/* Select DMC0 for parallel output path 0 */
pdp0_sw_cfg.out_src = regk_iop_sw_cfg_dmc0;
/* Use timer 1 in timer group 2 as FIFO in0 strobe */
pdp0_sw_cfg.in_strb = regk_iop_sw_cfg_strb_timer_grp2_tmrl;
/* Do not select any last signal */
pdp0_sw_cfg.in_last = regk_iop_sw_cfg_none;
/* Select 8 bits (for the FIFO) */
pdp0_sw_cfg.in_size = regk_iop_sw_cfg_size_8;
/* Select BUS0 for parallel input path */
pdp0_sw_cfg.in_src = regk_iop_sw_cfg_bus0;
/* Use gated clock 0 from SAP out as FIFO out0 strobe */
pdp0_sw_cfg.out_strb = regk_iop_sw_cfg_gated_clk0;
/* Select parallel path 0 as user of DMC out 0 */
pdp0_sw_cfg.dmc0_usr = regk_iop_sw_cfg_par0;
REG_WR(iop_sw_cfg, regi_iop_sw_cfg, rw_pdp0_cfg, pdp0_sw_cfg);
```

Write a value from the SPU0 directly to the highest byte of BUS0, and let through the lowest byte for the parallel data path:

#### CPU

```

reg_iop_sw_cfg_rw_bus0_mask      sw_cfg_bus0_mask;
reg_iop_sw_cpu_rw_bus0_set_mask  sw_cpu_bus0_set_mask;
reg_iop_sw_cpu_rw_bus0_clr_mask  sw_cpu_bus0_clr_mask;

reg_iop_sw_cfg_rw_bus0_oe_mask   sw_cfg_bus0_oe_mask;
reg_iop_sw_cpu_rw_bus0_oe_set_mask sw_cpu_bus0_oe_set_mask;
reg_iop_sw_cpu_rw_bus0_oe_clr_mask sw_cpu_bus0_oe_clr_mask;

reg_iop_sw_cfg_rw_bus_out_cfg    bus_out_sw_cfg;

reg_iop_sw_cfg_rw_pdp0_cfg       pdp0_sw_cfg;

/* BUS0 signals */

/* BUS0 out block register--let through the lowest byte for the parallel data path,
and the highest byte for the SPU0 */
sw_cfg_bus0_mask = 0xff0000ff;
REG_WR(iop_sw_cfg, regi_iop_sw_cfg, rw_bus0_mask, sw_cfg_bus0_mask);

/* BUS0 set signals register--clear all signals */
sw_cpu_bus0_set_mask = 0x00000000;
REG_WR(iop_sw_cpu, regi_iop_sw_cpu, rw_bus0_set_mask, sw_cpu_bus0_set_mask);

/* BUS0 block signals register--block the three highest bytes for any
unwanted signals from the multiplexer */
sw_cpu_bus0_clr_mask = 0xfffffff0;
REG_WR(iop_sw_cpu, regi_iop_sw_cpu, rw_bus0_clr_mask, sw_cpu_bus0_clr_mask);

/* Output enables for BUS0 */

/* Do not block any output enables */
sw_cfg_bus0_oe_mask = 0xf;
REG_WR_INT(iop_sw_cfg, regi_iop_sw_cfg, rw_bus0_oe_mask, sw_cfg_bus0_oe_mask);

/* Block all signals for output enables except the lowest (for the PDP) */
sw_cpu_bus0_oe_clr_mask = 0x7;
REG_WR_INT(iop_sw_cpu, regi_iop_sw_cpu, rw_bus0_oe_clr_mask, sw_cpu_bus0_oe_clr_mask);

/* Set the output enables for the two lowest bytes */
sw_cpu_bus0_oe_set_mask = 0x3;
REG_WR_INT(iop_sw_cpu, regi_iop_sw_cpu, rw_bus0_oe_set_mask, sw_cpu_bus0_oe_set_mask);

/* Configure the lowest word of BUS0 for the parallel data path */
bus_out_sw_cfg = REG_RD(iop_sw_cfg, regi_iop_sw_cfg, rw_bus_out_cfg);
bus_out_sw_cfg.bus0_lo = regk_iop_sw_cfg_pdp_out0_lo;
REG_WR(iop_sw_cfg, regi_iop_sw_cfg, rw_bus_out_cfg, bus_out_sw_cfg);

/* Parallel Data Path Config */
pdp0_sw_cfg = REG_RD(iop_sw_cfg, regi_iop_sw_cfg, rw_pdp0_cfg);
/* Select DMC0 for parallel output path 0 */
pdp0_sw_cfg.out_src = regk_iop_sw_cfg_dmc0;
/* Synchronize FIFO in0 strobe with GIO in[2] */
pdp0_sw_cfg.in_strb = regk_iop_sw_cfg_gio2;
/* Do not select any last signal */
pdp0_sw_cfg.in_last = regk_iop_sw_cfg_none;
/* Select 8 bits (for the FIFO) */
pdp0_sw_cfg.in_size = regk_iop_sw_cfg_size_8;
/* Select BUS0 for parallel input path */
pdp0_sw_cfg.in_src = regk_iop_sw_cfg_bus0;
/* Synchronize FIFO out0 strobe with GIO in[2] */
pdp0_sw_cfg.out_strb = regk_iop_sw_cfg_gio2;
/* Select parallel path 0 as user of DMC out 0 */
pdp0_sw_cfg.dmc0_usr = regk_iop_sw_cfg_par0;
REG_WR(iop_sw_cfg, regi_iop_sw_cfg, rw_pdp0_cfg, pdp0_sw_cfg);

```

#### SPU0

```

; Put the value for BUS0 in general register r1
moveh r1, 0x5500, r1

; Write the value to BUS0 via register rw_bus0_oe_set_mask
rw r1, REG_ADDR(iop_sw_spu, iop_sw_spu0, rw_bus0_oe_set_mask)

```

Same example as above, but instead use the SPU0 special register B0OUT.

#### CPU

```
reg_iop_sw_cfg_rw_bus0_mask      sw_cfg_bus0_mask;
reg_iop_sw_cpu_rw_bus0_set_mask  sw_cpu_bus0_set_mask;
reg_iop_sw_cpu_rw_bus0_clr_mask  sw_cpu_bus0_clr_mask;

reg_iop_sw_cfg_rw_bus0_oe_mask   sw_cfg_bus0_oe_mask;
reg_iop_sw_cpu_rw_bus0_oe_set_mask sw_cpu_bus0_oe_set_mask;
reg_iop_sw_cpu_rw_bus0_oe_clr_mask sw_cpu_bus0_oe_clr_mask;

reg_iop_sw_cfg_rw_bus_out_cfg    bus_out_sw_cfg;

reg_iop_sw_cfg_rw_pdp0_cfg       pdp0_sw_cfg;

/* BUS0 signals */

/* BUS0 out block register--let through the lowest and the highest bytes */
sw_cfg_bus0_mask = 0xff0000ff;
REG_WR(iop_sw_cfg, regi_iop_sw_cfg, rw_bus0_mask, sw_cfg_bus0_mask);

/* BUS0 set signals register--clear all signals */
sw_cpu_bus0_set_mask = 0x00000000;
REG_WR(iop_sw_cpu, regi_iop_sw_cpu, rw_bus0_set_mask, sw_cpu_bus0_set_mask);

/* BUS0 block signals register--block the two middle bytes */
sw_cpu_bus0_clr_mask = 0x00ffff00;
REG_WR(iop_sw_cpu, regi_iop_sw_cpu, rw_bus0_clr_mask, sw_cpu_bus0_clr_mask);

/* Output enables for BUS0 */

/* Do not block any output enables */
sw_cfg_bus0_oe_mask = 0xf;
REG_WR_INT(iop_sw_cfg, regi_iop_sw_cfg, rw_bus0_oe_mask, sw_cfg_bus0_oe_mask);

/* Block the two middle bytes for output enables coming from the multiplexer */
sw_cpu_bus0_oe_clr_mask = 0x6;
REG_WR_INT(iop_sw_cpu, regi_iop_sw_cpu, rw_bus0_oe_clr_mask, sw_cpu_bus0_oe_clr_mask);

/* Set the output enables for the highest and lowest bytes */
sw_cpu_bus0_oe_set_mask = 0x9;
REG_WR_INT(iop_sw_cpu, regi_iop_sw_cpu, rw_bus0_oe_set_mask, sw_cpu_bus0_oe_set_mask);

/* Configure the highest word of BUS0 for the SPU0 BUS0 register
and the lowest word for the parallel data path */
bus_out_sw_cfg = REG_RD(iop_sw_cfg, regi_iop_sw_cfg, rw_bus_out_cfg);
bus_out_sw_cfg.bus0_hi = regk_iop_sw_cfg_spu0_bus_out0_hi;
bus_out_sw_cfg.bus0_lo = regk_iop_sw_cfg_pdp_out0_lo;
REG_WR(iop_sw_cfg, regi_iop_sw_cfg, rw_bus_out_cfg, bus_out_sw_cfg);

/* Parallel Data Path Config */
pdp0_sw_cfg = REG_RD(iop_sw_cfg, regi_iop_sw_cfg, rw_pdp0_cfg);
/* Select DMC0 for parallel output path 0 */
pdp0_sw_cfg.out_src = regk_iop_sw_cfg_dmc0;
/* Select SPU0 GIO[5] as FIFO in0 strobe */
pdp0_sw_cfg.in_strb = regk_iop_sw_cfg_spu0_gio_out5;
/* Do not select any last signal */
pdp0_sw_cfg.in_last = regk_iop_sw_cfg_none;
/* Select 8 bits (for the FIFO) */
pdp0_sw_cfg.in_size = regk_iop_sw_cfg_size_8;
/* Select BUS0 for parallel input path */
pdp0_sw_cfg.in_src = regk_iop_sw_cfg_bus0;
/* Select SPU0 GIO[5] as FIFO out0 strobe */
pdp0_sw_cfg.out_strb = regk_iop_sw_cfg_spu0_gio_out5;
/* Select parallel path 0 as user of DMC out 0 */
pdp0_sw_cfg.dmc0_usr = regk_iop_sw_cfg_par0;
REG_WR(iop_sw_cfg, regi_iop_sw_cfg, rw_pdp0_cfg, pdp0_sw_cfg);
```

#### SPU0

```
/* Write the value to BUS0 via register B0OUT
moveh B0OUT, 0x5500, B0OUT
*/
```



The GIO bus is mapped in blocks of 4 bits, or nibbles, to the two mapping alternatives *Mapping A* and *Mapping B*:

GIO nibble:	Mapping A:	Mapping B:
GIO[3:0]	PB[11:8]	PD[17:16] PE[17:16]
GIO[7:4]	PD[11:8]	PB[17:16] PC[17:16]
GIO[11:8]	PC[11:8]	PE[3:0]
GIO[15:12]	PE[11:8]	PD[3:0]
GIO[19:16]	PB[15:12]	PC[3:0]
GIO[23:20]	PD[15:12]	PC[7:4]
GIO[27:24]	PC[15:12]	PE[7:4]
GIO[31:28]	PE[15:12]	PD[7:4]

Apart from these 72 pins there are also two pins that contain specific hardware for USB. To use the internal on-chip USB 1.1 transceiver some of the 72 pins must be connected to it:

#### Mapping for internal USB 1.1 transceiver

Transceiver Signal	Direction	Description	IOP GIO	Pin
<b>USB Port 0</b>				
vmo	out	Data out minus	GIO[8]A	PC[8]
vpo	out	Data out plus	GIO[9]A	PC[9]
vmi	in	Data in minus	GIO[10]A	PC[10]
vpi	in	Data in plus	GIO[11]A	PC[11]
oe_n	out	Output enable	GIO[24]A	PC[12]
speed_n	out	Speed select	GIO[25]A	PC[13]
rcv	in	Receive data	GIO[26]A	PC[14]
<b>USB Port 1</b>				
vmo	out	Data out minus	GIO[12]A	PE[8]
vpo	out	Data out plus	GIO[13]A	PE[9]
vmi	in	Data in minus	GIO[14]A	PE[10]
vpi	in	Data in plus	GIO[15]A	PE[11]
oe_n	out	Output enable	GIO[28]A	PE[12]
speed_n	out	Speed select	GIO[29]A	PE[13]
rcv	in	Receive data	GIO[30]A	PE[14]

The configuration for the use of the internal on-chip USB transceiver is done in the pinmux register bank (that is, outside the I/O-processor):

```
rw_usb_phy
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-----+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0| | |
+-----+
| en_usb0
en_usb1
```

Connects the internal USB 1.1 phy to the IO processor. Two different mappings are available.

-----  
en\_usb1 [1]

Constants:

no = 0 -- USB phy not connected  
yes = 1 -- USB phy connected to the IO processor

Default constant: no

Enables the IO-processor to use the internal USB phy. This field connects IO processor ports pe[14:8] to the USB phy.

-----  
en\_usb0 [0]

Constants:

no = 0 -- USB phy not connected  
yes = 1 -- USB phy connected to the IO processor

Default constant: no

Enables the IO-processor to use the internal USB phy. This field connects IO processor ports pc[14:8] to the USB phy.

For further information, and to see what pin ball is connected to each bit of the ports, see Pinout and Pin Multiplexing (Note: this is only a temporary link!).

Configuration example in C for a 16-bit parallel protocol:

```
reg_iop_sw_cfg_rw_pinmapping pinmapping;

/* Read the value (in case there are more than one pin mapping) */
pinmapping = REG_RD(iop_sw_cfg, regi_iop_sw_cfg, rw_pinmapping);

/* BUS0[7:0] => PB[7:0] */
pinmapping.bus0_byte0 = regk_iop_sw_cfg_a;

/* BUS0[15:8] => PB[15:8] */
pinmapping.bus0_byte1 = regk_iop_sw_cfg_b;

/* GIO[3:0] => PD[17:16] PE[17:16] */
pinmapping.gio3_0 = regk_iop_sw_cfg_b;

/* GIO[7:4] => PB[17:16] PC[17:16] */
pinmapping.gio7_4 = regk_iop_sw_cfg_b;

/* GIO[11:8] => PC[11:8] */
pinmapping.gio11_8 = regk_iop_sw_cfg_a;

/* Write the new values */
REG_WR(iop_sw_cfg, regi_iop_sw_cfg, rw_pinmapping, pinmapping);
```



```

-----
strb [4:3]

Constants:
  hi      = 0 -- High strobe
  pos     = 1 -- Positive edge of strobe
  neg     = 2 -- Negative edge of strobe
  pos_neg = 3 -- Both positive and negative edge of strobe

Default constant: hi

Select if the clock source (field:clk_src) should be triggered when
it is high or on positive or negative edge.

-----
clk_src [2:0]

Constants:
  clk200      = 0 -- I/O Processor clock
  div_clk200  = 1 -- Predivided I/O Processor clock
  clk_gen     = 2 -- Clock generator
  div_clk_gen = 3 -- Predivided clock generator
  tmr        = 4 -- Timer N-1 in the Timer group, where N is the
                  index of Timer which is configured

Default constant: int

Selects which clock source should be used for timer countdown.
-----

```

Excerpt from the `rw_cmd` register:

```

rw_cmd

  3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
  1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-----+-----+-----+-----+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0| | | | |
+-----+-----+-----+-----+
                                strb  dis  en   rst

```

Timer commands for the whole group. Each bit in a field represents one timer.

```

-----
strb [15:12]

Strobe one or more timers. The strobe will count down reg:r_tmr_cnt
one step. The timer will count down even if the timer is disabled.

-----
dis [11:8]

Disable one or more timers.

-----
en [7:4]

Enable one or more timers.

-----
rst [3:0]

Reset one or more timers.

-----

```

Example of a 25 MHz clock:

#### CPU

```
reg_iop_timer_grp_rw_tmr_cfg timer_cfg;
reg_iop_timer_grp_rw_tmr_len timer_len;

/* Configure the third clock in timer group 0 */
timer_cfg = REG_RD_VECT(iop_timer_grp, regi_iop_timer_grp0, rw_tmr_cfg, 2);
/* Do not invert the output strobe */
timer_cfg.inv = regk_iop_timer_grp_no;
/* Use toggle mode for a clock */
timer_cfg.out_mode = regk_iop_timer_grp_toggle;
/* Put the run_mode in "complete"
timer_cfg.run_mode = regk_iop_timer_grp_complete;
/* The clock source must be put to "hi" when system (IOP) clock is used */
timer_cfg.strb = regk_iop_timer_grp_hi;
/* Choose the system clock (IOP clock) of 200 MHz */
timer_cfg.clk_src = regk_iop_timer_grp_clk200;
REG_WR_VECT(iop_timer_grp, regi_iop_timer_grp0, rw_tmr_cfg, 2, timer_cfg);

/* The cycle length of the input clock source.
A value of 3 gives a division of 4 for a half period
200 / (4 x 2) = 25 */
timer_len.value = 0x3;
REG_WR_VECT(iop_timer_grp, regi_iop_timer_grp0, rw_tmr_len, 2, timer_len);
```

#### SPU0

```
; Start the third clock
rwq REG_FIELD(iop_timer_grp, rw_cmd, en, 0x4), \
REG_ADDR(iop_timer_grp, iop_timer_grp0, rw_cmd)
```

## 7.10 The Triggers

There are 8 trigger groups with 4 triggers in each group. A Trigger can detect rising and/or falling edges of an input signal. An interrupt can then be generated, or the output strobe from the trigger can be used as an input to a SPU, or to start or stop a timer.

## 7.11 The MC

The MC, Memory Controller, makes it possible to load the memories of the SPUs via the register interface. It can also be used for reading and writing data to the system memory (that is, the external memory), or to the internal 128 kbyte memory of the ETRAX FS. (Note however that data can only be written to the memory banks of the SPUs, the MC can not read data from these memory banks.)

The MC uses the register interface to read and write data, and data can only be read or written one byte (8 bits), two bytes (16 bits), three bytes (24 bits) or four bytes (32 bits) at a time. When data are read or written to the external memory, it shares the data access with the DMA via the memory arbiter.

It is not possible to write data to the memory of the MPU. The MPU has a set of instructions (called LW, SW, and SWX) to read data from, or write data to, its memory bank.

## 7.11.1 Reading from, and Writing to, System Memory

Example in assembly of a read from system memory:

```
mc_config_read:
; Configure the MC for a read from memory...
; do not keep ownership (release ownership when
; this single read is done)
rwq REG_STATE(iop_sw_spu, rw_mc_ctrl, keep_owner, no) | \
REG_STATE(iop_sw_spu, rw_mc_ctrl, cmd, rd) | \
; read 4 bytes
REG_FIELD(iop_sw_spu, rw_mc_ctrl, size, 0x4) | \
REG_STATE(iop_sw_spu, rw_mc_ctrl, wr_spu0_mem, no) | \
REG_STATE(iop_sw_spu, rw_mc_ctrl, wr_spu1_mem, no), \
REG_ADDR(iop_sw_spu, iop_sw_spu0, rw_mc_ctrl)

; Read the r_mc_stat MC register
rr REG_ADDR(iop_sw_spu, iop_sw_spu0, r_mc_stat), r15

nop ; Wait one cycle for the update of the r15 register

; Check for ownership...
bbc r15, REG_BIT(iop_sw_spu, r_mc_stat, owned_by_spu0), mc_config
nop ; Delay slot

; Ownership granted, put the system memory address in rw_mc_addr
rw r2, REG_ADDR(iop_sw_spu, iop_sw_spu0, rw_mc_addr)

; Wait for busy bit to go low
mc_config_read_wait:
; Read the r_mc_stat MC register
rr REG_ADDR(iop_sw_spu, iop_sw_spu0, r_mc_stat), r15

nop ; Wait one cycle for the update of the r15 register

; Check busy bit of MC...
bbs r15_misc, REG_BIT(iop_sw_spu, r_mc_stat, busy_spu0), mc_config_read
nop ; Delay slot

; Busy bit is low, read data from external memory
rr REG_ADDR(iop_sw_spu, iop_sw_spu0, rs_mc_data), r1
```

Example in assembly of a write to system memory:

```
mc_config_write:
; Configure the MC for a write from memory...
; do not keep ownership (release ownership when
; this single write is done)
rwq REG_STATE(iop_sw_spu, rw_mc_ctrl, keep_owner, no) | \
REG_STATE(iop_sw_spu, rw_mc_ctrl, cmd, wr) | \
; write 4 bytes
REG_FIELD(iop_sw_spu, rw_mc_ctrl, size, 0x4) | \
REG_STATE(iop_sw_spu, rw_mc_ctrl, wr_spu0_mem, no) | \
REG_STATE(iop_sw_spu, rw_mc_ctrl, wr_spu1_mem, no), \
REG_ADDR(iop_sw_spu, iop_sw_spu0, rw_mc_ctrl)

; Read the r_mc_stat MC register
rr REG_ADDR(iop_sw_spu, iop_sw_spu0, r_mc_stat), r15

nop ; Wait one cycle for the update of the r15 register

; Check for ownership...
bbc r15, REG_BIT(iop_sw_spu, r_mc_stat, owned_by_spu0), mc_config_write
nop ; Delay slot

; Ownership granted, put the data in rw_mc_data
rw r1, REG_ADDR(iop_sw_spu, iop_sw_spu0, rw_mc_data)

; Put the system memory address in rw_mc_addr
rw r2, REG_ADDR(iop_sw_spu, iop_sw_spu0, rw_mc_addr)
```

## 7.11.2 SPU Memory Load

Loading the SPU's memory banks should not be done with the C library function `fopen`. Instead, use a program like `xxd` (hexdump) to create a static array function of the binary SPU or MPU file:

```
xxd -i spu_prog.bin spu_prog.h
```

The output file "spu\_prog.h" will contain an array of the file like:

```
unsigned char spu_prog_bin[] = {
    0x09, 0x05, 0x00, 0x70, 0x06, 0x05, 0x01, 0x50, 0x20, 0x05, 0x05, 0x70,
    [...]
    0x00, 0x00, 0x00, 0x00
};
unsigned int spu_prog_bin_len = 4096;
```

Now load the static array by the use of the memory controller (note that you have to cast the created array from "char" to "unsigned int" in this example):

```
void load_spu0_mem(unsigned int *spu_prog_bin) {
    int i;
    reg_iop_sw_cpu_rw_mc_ctrl mc_ctrl;
    reg_iop_sw_cpu_r_mc_stat mc_stat;
    reg_iop_spu_rw_ctrl spu_ctrl;

    /* Disable the SPU0 */
    spu_ctrl.en = regk_iop_spu_no;
    spu_ctrl.fsm = regk_iop_spu_no;
    REG_WR(iop_spu, regi_iop_spu0, rw_ctrl, spu_ctrl);

    /* Configure the mc_ctrl struct for the rw_mc_ctrl register */
    mc_ctrl.size = 4;
    mc_ctrl.cmd = regk_iop_sw_cpu_reg_copy;
    mc_ctrl.keep_owner = regk_iop_sw_cpu_yes;
    mc_ctrl.wr_spul_mem = regk_iop_sw_cpu_no;
    mc_ctrl.wr_spu0_mem = regk_iop_sw_cpu_yes;

    /* Write the values, and wait for ownership */
    do {
        REG_WR(iop_sw_cpu, regi_iop_sw_cpu, rw_mc_ctrl, mc_ctrl);
        mc_stat = REG_RD(iop_sw_cpu, regi_iop_sw_cpu, r_mc_stat);
    } while (mc_stat.owned_by_cpu == regk_iop_sw_cpu_no);

    /* Do the memory write, the memory size is 4096 bytes
       Note: the MC writes 4 bytes at a time in this example */
    for (i = 0; i < (4096 / 4); i++) {
        /* Program counter incrementation (note: the PC uses byte addresses) */
        REG_WR_INT(iop_spu, regi_iop_spu0, rw_seq_pc, (i * 4));
        /* Write data from the array to the SPU0 memory bank */
        REG_WR_INT(iop_sw_cpu, regi_iop_sw_cpu, rw_mc_data, *(spu_prog_bin + i));
    }

    /* Reset the program counter, rw_seq_pc */
    REG_WR_INT(iop_spu, regi_iop_spu0, rw_seq_pc, 0);

    /* Release ownership of MC */
    REG_RD(iop_sw_cpu, regi_iop_sw_cpu, rs_mc_data);
}
```