alpide_control

Address width: 32

Data width: 32

Base address: 0x24110000

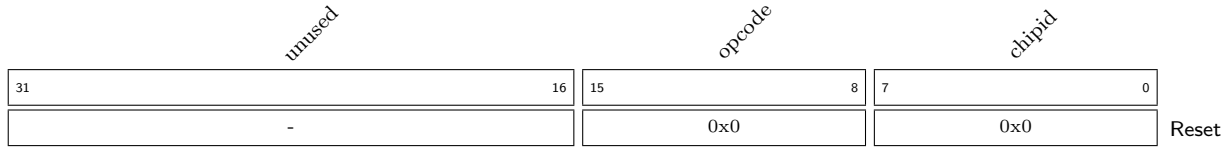
ALPIDE slow control management.

1 Register List

#	Name	Mode	Address	Type	Length	Reset
0	write_ctrl	RW	0x00000000	FIELDS	16	0x0
1	write_address	RW	0x00000004	SLV	16	0x0
2	write_data	RW	0x00000008	SLV	16	0x0
3	stop_bit_requirement	RW	0x0000000C	SLV	2	0x3
4	read_status	RO	0x00000010	FIELDS	12	0x0
5	read_data	RO	0x00000014	SLV	16	0x0
6	reset_counters	PULSE	0x00000018	SL	1	0x0
7	num_broadcasts	RO	0x0000001C	DEFAULT	32	0x0
8	num_writes	RO	0x00000020	DEFAULT	32	0x0
9	num_reads	RO	0x00000024	DEFAULT	32	0x0
10	num_opcodes	RO	0x00000028	DEFAULT	32	0x0
11	num_trigger_sent	RO	0x0000002C	DEFAULT	32	0x0
12	num_trigger_not_sent	RO	0x00000030	DEFAULT	32	0x0
13	num_wait_exec	RO	0x00000034	DEFAULT	32	0x0
14	mask_busy	RW	0x00000038	FIELDS	2	0x0
15	wait_value	RW	0x0000003C	DEFAULT	32	0x0
16	num_regs	RW	0x00000040	DEFAULT	32	0x0

2 Registers

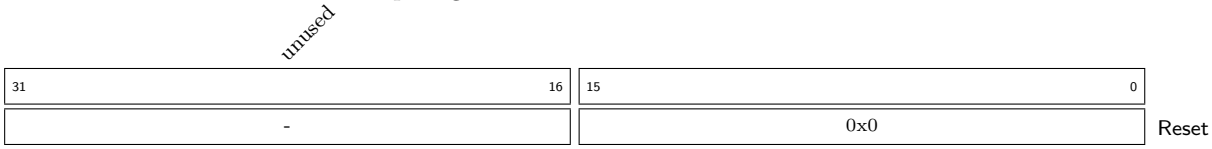
Register 2.1: WRITE_CTRL - RW (0x00000000)
When written to, start the alpide_control state machine. Opcode decides the operation executed.



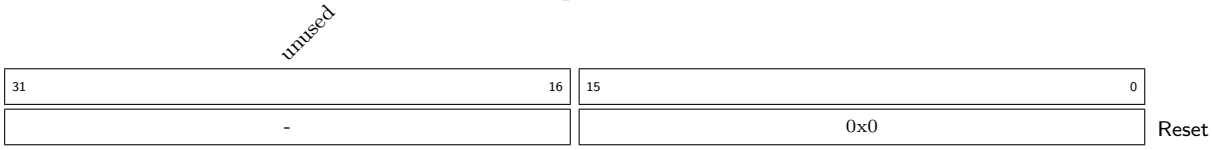
chipid

opcode

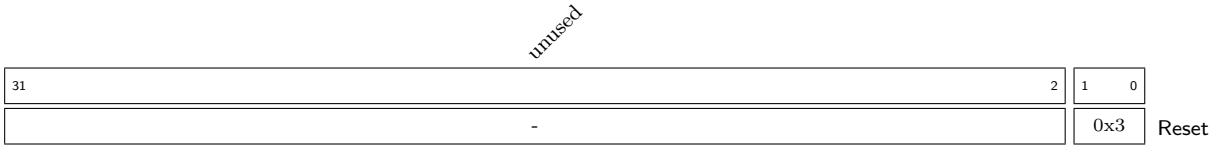
Register 2.2: WRITE_ADDRESS - RW (0x00000004)
The ALPIDE chips register address. Must be written before write_ctrl.



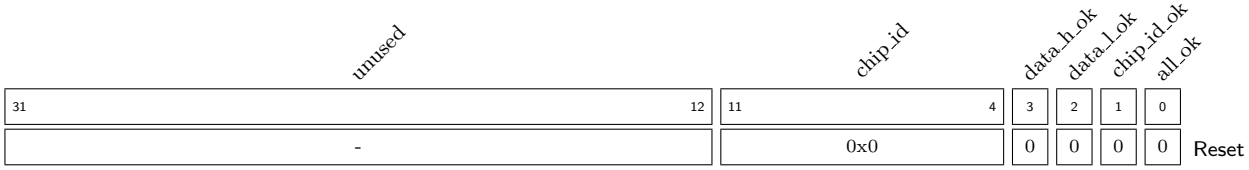
Register 2.3: WRITE_DATA - RW (0x00000008)
Data that is written to ALPIDE register. Must be written before write_ctrl when performing a write operation.



Register 2.4: STOP_BIT_REQUIREMENT - RW (0x0000000C)
Requirement for the start/stop bit detection. 3 -¿ All 6 bits (strict - may fail with noisy signal). 2 -¿ 5 bits. 1 -¿ 4 bits. 0 -¿ Not allowed (will revert to 6 bits).



Register 2.5: READ_STATUS - RO (0x00000010)
Read status.



all_ok

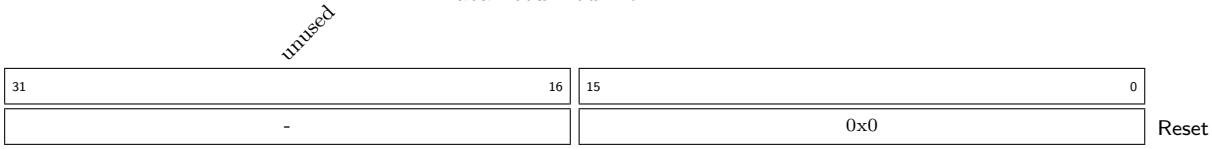
chip_id_ok

data_l_ok

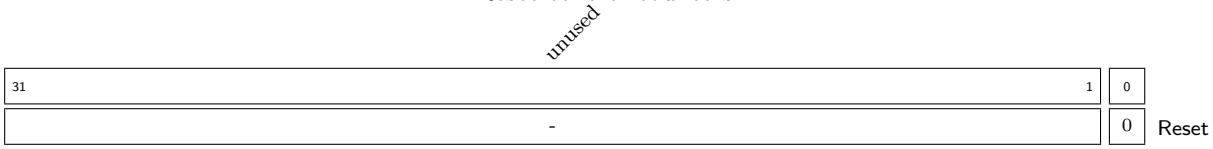
data_h_ok

chip_id

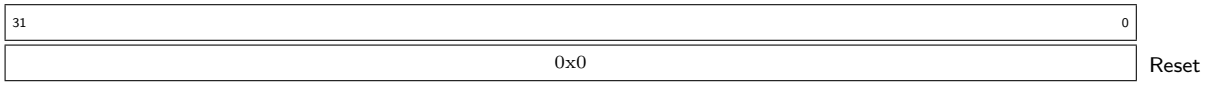
Register 2.6: READ_DATA - RO (0x00000014)
Data returned from ALPIDE.



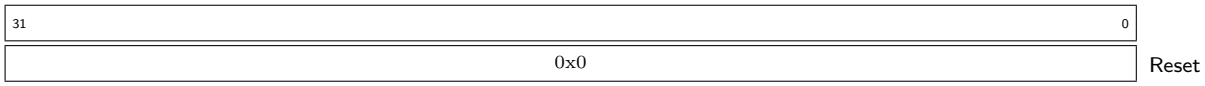
Register 2.7: RESET_COUNTERS - PULSE FOR 1 CYCLES - (0x00000018)
Reset control counters.



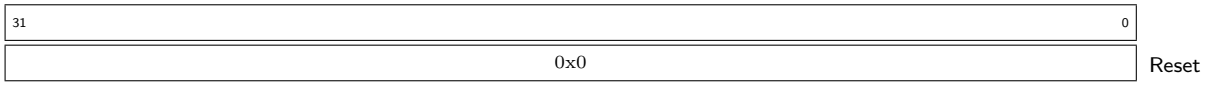
Register 2.8: NUM_BROADCASTS - RO (0x0000001C)
Number of broadcasts.



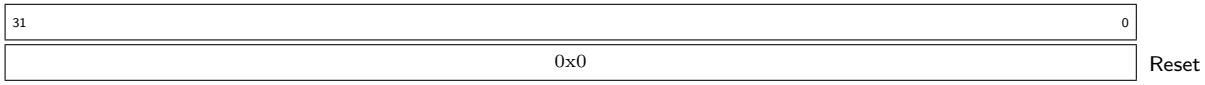
Register 2.9: NUM_WRITES - RO (0x00000020)
Number of write operations.



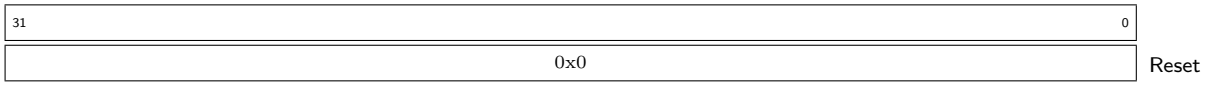
Register 2.10: NUM_READS - RO (0x00000024)
Number of read operations.



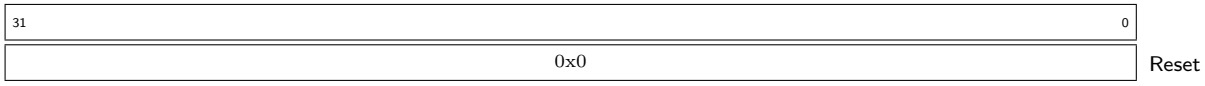
Register 2.11: NUM_OPCODES - RO (0x00000028)
Number of opcodes transmitted.



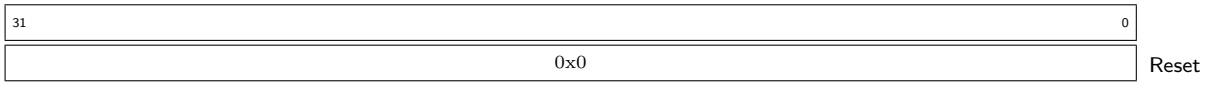
Register 2.12: NUM_TRIGGER_SENT - RO (0x0000002C)
Number of triggers transmitted.



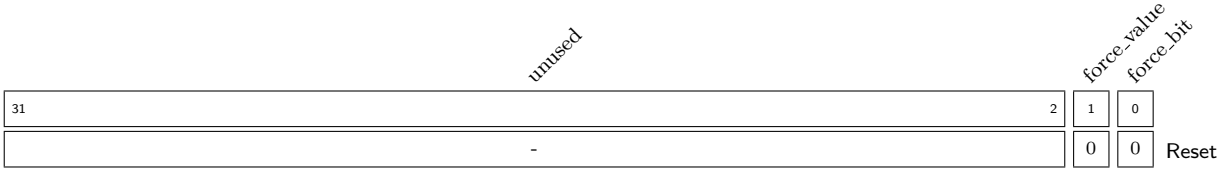
Register 2.13: NUM_TRIGGER_NOT_SENT - RO (0x00000030)
Number of triggers not transmitted.



Register 2.14: NUM_WAIT_EXEC - RO (0x00000034)
Number of waits executed.



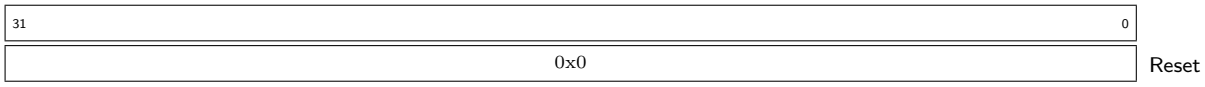
Register 2.15: MASK_BUSY - RW (0x00000038)
Set the busy status by force, either not busy or busy.



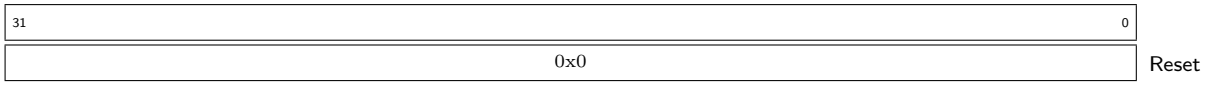
force_bit

force_value '1' -¿ busy, '0' -¿ not busy.

Register 2.16: WAIT_VALUE - RW (0x0000003C)
Unused.



Register 2.17: NUM_REGS - RW (0x00000040)
Unused.



3 Example VHDL Register Access

All registers are bundled in records based on their mode. E.g. all RW registers are accessed through the record *bustype_rw_regs*. Access is also dependent on the type of register. All register of type SL, SLV and DEFAULT are all directly accessed by just specifying the mode record signal. E.g. the RW register *reg0* can be assigned a value like this (assuming AXI-bus):

```
axi_rw_regs.reg0 <= (others => '0');
```

Registers of type FIELD cannot be directly accessed without specification of a certain field. This is because the registers are implemented as a record in VHDL (thus a record of records). E.g. if the RO register *reg1* contains the field *field3* it can be accessed like this (assuming AXI-bus):

```
axi_ro_regs.reg1.field3 <= (others => '0');
```