



antmicro

Rowhammer tester

Antmicro

2026-01-16

CONTENTS

1	Project overview	1
1.1	Tester suite architecture	1
2	Installation and setup	3
2.1	Installing dependencies	3
2.2	Install Rowhammer tester	3
2.3	Local documentation build	4
2.4	Unit tests	4
2.5	Network USB adapter setup	4
3	Building Rowhammer designs	7
3.1	Building and uploading the bitstreams	7
3.2	Ethernet connection	11
3.3	Packaging the bitstream	11
3.4	Building for simulation	12
4	SO-DIMM DDR5 Tester	13
4.1	IO map	13
4.2	Rowhammer Tester Target Configuration	15
5	RDIMM DDR5 Tester	16
5.1	Variants	16
5.2	IO map	16
5.3	Rowhammer Tester Target Configuration	20
5.4	Simulation	21
6	RDIMM DDR4 Tester	22
6.1	IO map	22
6.2	Board configuration	24
7	LPDDR4 Test Board	25
7.1	IO map	26
7.2	Board configuration	26
8	ZCU104 board	28
8.1	Board configuration	28
8.2	Preparing SD card	29
8.3	Loading the bitstream	30
8.4	ZCU104 microUSB	30
8.5	Network setup	30

8.6	SSH access	31
8.7	Controlling the board	32
8.8	ZCU104 SD card image	32
9	Arty-A7 board	36
9.1	Board configuration	36
10	LPDDR4 Test Bed	37
11	DDR5 Test Bed	38
12	LPDDR5 Test Bed	39
13	Performing attacks (hammering)	40
13.1	Attack modes	40
13.2	Patterns	42
13.3	Example output	42
13.4	Row selection examples	42
13.5	Cell retention measurement	43
13.6	Read count parameter optimization	44
13.7	DRAM modules	45
13.8	Utilities	46
14	Result visualization	52
14.1	Plot bitflips - logs2plot.py	52
14.2	Plot per DQ pad - logs2dq.py	53
14.3	Use F4PGA Visualizer - logs2vis.py	55
15	Test-writing playbook	56
15.1	Payload	56
15.2	Row mapping	56
15.3	Row Generator class	57
15.4	Payload generator class	57
15.5	Configurations	61
16	Memory testing	63
16.1	CI driven testing	63
16.2	Manual testing	63
16.3	RDIMM DDR5 test coverage	64
17	Building Linux target	66
17.1	Base DDR5 Tester Linux Options	66
17.2	Building the RDIMM DDR5 Tester Linux Target	67
17.3	Interacting with RDIMM DDR5 Tester Linux Target	67
17.4	Setting up a TFTP Server	67
17.5	Booting Linux on RDIMM DDR5 Tester Linux Target	68
18	PCIe support	71
18.1	DRAM Bender integration	71
19	Documentation for Row Hammer Tester Arty-A7	73
19.1	Modules	73
19.2	Register Groups	73

20 Documentation for Row Hammer Tester ZCU104	163
20.1 Modules	163
20.2 Register Groups	163
21 Documentation for Row Hammer Tester Data Center DRAM Tester	275
21.1 Modules	275
21.2 Register Groups	275
22 Documentation for Row Hammer Tester LPDDR4 Test Board	386
22.1 Modules	386
22.2 Register Groups	386
23 Documentation for Row Hammer Tester DDR5 Test Board	494
23.1 Modules	494
23.2 Register Groups	494
24 Documentation for Row Hammer Tester DDR5 Tester	615
24.1 Modules	615
24.2 Register Groups	616

CHAPTER ONE

PROJECT OVERVIEW

The aim of this project is to provide a platform for testing DRAM vulnerability to rowhammer attacks.

This suite can be run on real hardware (FPGAs) or in a simulation mode.

Read more about particular aspects of the framework in dedicated blog articles:

- Rowhammer Tester platform overview
- LPDDR4 Test Board
- Data Center RDIMM DDR4 Tester
- Data Center RDIMM DDR5 Tester
- SO-DIMM DDR5 Tester

1.1 Tester suite architecture

This section provides an overview of the Rowhammer Tester suite architecture.

System architecture is presented in Fig. 1.1 below:

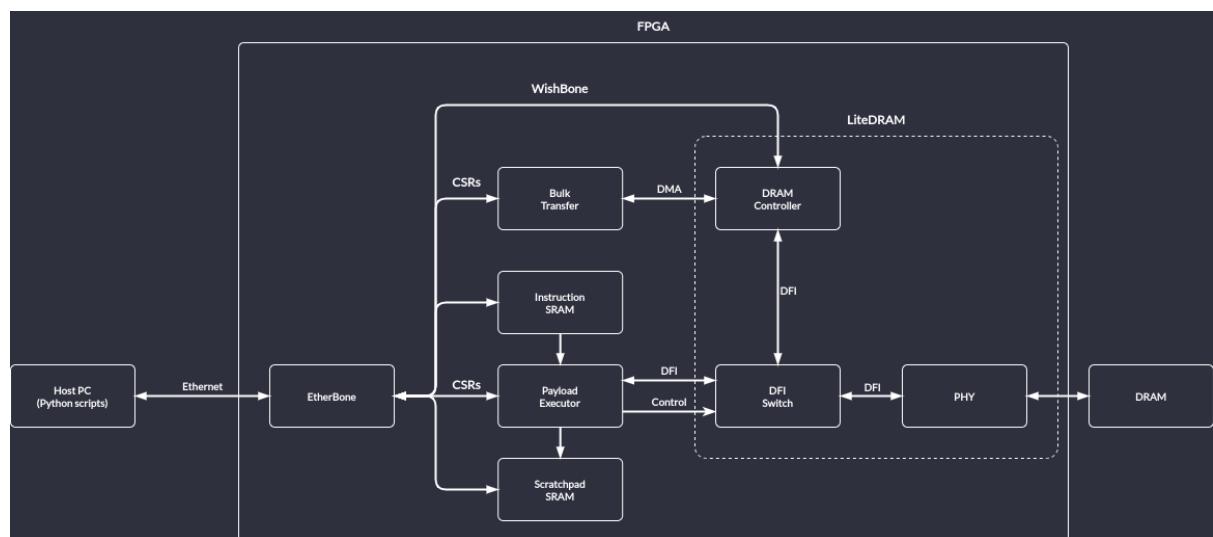


Fig. 1.1: Rowhammer Tester suite architecture

The DRAM is connected to LiteDRAM, which provides swappable PHYs and a DRAM controller implementation.

In the default bulk transfer mode, the LiteDRAM controller is connected to PHY and ensures correct DRAM traffic. Bulk transfers can be controlled using dedicated Control & Status Registers (CSRs) and use LiteDRAM DMA to ensure fast operation.

The Payload Executor allows executing a user-provided sequence of commands. It temporarily disconnects the DRAM controller from PHY; executes the instructions stored in the SRAM memory, translating them into DFI commands and finally reconnects the DRAM controller.

The application side consists of a set of Python scripts communicating with the FPGA using the LiteX EtherBone bridge.

CHAPTER TWO

INSTALLATION AND SETUP

2.1 Installing dependencies

Make sure you have Python 3 installed with the `venv` module, and the dependencies required to build `verilator`, `openFPGALoader` and `OpenOCD`. To install the dependencies on Ubuntu 18.04 LTS, run:

```
apt install git build-essential autoconf cmake flex bison libftdi-dev libjson-c-dev libevent-dev libtinfo-dev uml-utilities python3 python3-venv python3-wheel-protoBuf-compiler libcairo2 libftdi1-2 libftdi1-dev libhidapi-hidraw0 libhidapi-dev libudev-dev pkg-config tree zlib1g-dev zip unzip help2man curl ethtool
```

 Note

On some Debian-based systems, there's a problem with a broken dependency:

libc6-dev : Breaks: libgcc-9-dev (< 9.3.0-5~) but 9.2.1-19 is to be installed

gcc-9-base package installation solves the problem.

On Ubuntu 22.04 LTS the following dependencies may also be required:

```
apt install libtool libusb-1.0-0-dev pkg-config
```

2.2 Install Rowhammer tester

Clone the rowhammer-tester repository and install the rest of the required dependencies:

```
git clone --recursive https://github.com/antmicro/rowhammer-tester.git  
cd rowhammer-tester  
make deps
```

The last command will download and build all the dependencies (including a RISC-V GCC toolchain) and set up a [Python virtual environment](#) under the `./venv` directory with all the required packages installed.

The virtual environment allows you to use Python without installing the packages system-wide. To enter the environment, you have to run `source venv/bin/activate` in each new shell. You can also use the provided `make env` target, which will start a new Bash shell with the `virtualenv`

already sourced. You can install packages inside the virtual environment by entering the environment and then using pip.

To build the bitstream, you will also need to have Vivado (version 2020.2 or newer) installed and the vivado command available in your PATH. To configure Vivado in the current shell, you need to source /PATH/TO/Vivado/VERSION/settings64.sh. Then include it in your .bashrc or other shell init script.

To make the process automatic without hard-coding in the shell init script, you can use tools like `direnv`. A sample .envrc file looks like so:

```
source venv/bin/activate
source /PATH/TO/Vivado/VERSION/settings64.sh
```

All other commands assume that you run Python from the virtual environment with vivado in your PATH.

2.3 Local documentation build

The part of the documentation related to the digital design is auto-generated from source files. Other files are static and are located in the docs/ directory.

To build the documentation locally you may need to install additional requirements in your virtual Python environment. Those requirements are listed in docs/requirements.txt

You can install them with:

```
pip install ./docs/requirements.txt
```

Then you can build the local HTML documentation using:

```
cd docs
make html
```

Once the building process finishes, you can open ./build/html/index.html in your web browser.

2.4 Unit tests

To run unit tests for rowhammer tester modules, use:

```
make test
```

2.5 Network USB adapter setup

In order to control the Rowhammer platform, an Ethernet connection is necessary. In case you want to use an USB Ethernet adapter for this purpose, follow the instructions below.

1. Make sure you use a 1GbE USB network adapter.
2. Determine the MAC address for the USB network adapter:
 - Run `sudo lshw -class network -short` to get the list of all network interfaces

- Check which of the devices uses the r8152 driver by running `sudo ethtool -i <device>`
 - Display the link information for the device running `sudo ip link show <device>` and look for the mac address next to the link/ether field
3. Configure the USB network adapter to appear as network device `fpga0` using systemd
- Create `/etc/systemd/network/10-fpga0.link` with the following contents:
 - Display the link information for the device running `sudo ip link show <device>` and look for the mac address next to the link/ether field

```
[Match]
# Set this to the MAC address of the USB network adapter
MACAddress=XX:XX:XX:XX:XX:XX

[Link]
Name=fpga0
```

4. Configure the `fpga0` network device with a static IP address, always up (even when disconnected) and ignored by the network manager.

- Run the following command, assuming your system uses NetworkManager:

```
nmcli con add type ethernet con-name 'Rowhammer Tester' ifname fpga0_
↪ ipv4.method manual ipv4.addresses 192.168.100.100/24
```

- Alternatively, if your system supports the legacy interfaces configuration file:

1. Make sure your `/etc/network/interfaces` file contains the following line:

```
source /etc/network/interfaces.d/*
```

2. Create `/etc/network/interfaces.d/fpga0` with the following contents:

```
auto fpga0
allow-hotplug fpga0
iface fpga0 inet static
    address 192.168.100.100/24
```

3. Check that `nmcli device` says the state is connected (externally) otherwise run `sudo systemctl restart NetworkManager`
- Run `ifup fpga0`
5. Run `sudo udevadm control --reload` and then unplug the USB Ethernet device and plug it back in
6. Check whether an `fpga0` interface is present with the correct IP address by running `networkctl status`

Note

In case you see `libusb_open()` failed with `LIBUSB_ERROR_ACCESS` when trying to use the rowhammer tester scripts with the USB Ethernet adapter, it indicates a permissions issue. To

remedy it, allow access to the FTDI USB to serial port chip. Run `ls -l /dev/ttyUSB*`, check the listed group for tty's and add the current user to this group by running `sudo adduser <username> <group>`.

BUILDING ROWHAMMER DESIGNS

This chapter provides building instructions for synthesizing the digital design for physical DRAM testers and simulation models.

3.1 Building and uploading the bitstreams

The bitstream building process is coordinated with a Makefile located in the main folder of the Rowhammer Tester repository. Currently, 6 main targets are provided, each targeting a different DRAM type and memory form factor. Use the tab view below to select a DRAM memory type of interest. You will be provided with a name of the built target, building instruction and a link to the relevant hardware platform.

This targets a single LPDDR5 IC soldered to the *LPDDR5 Test Bed* that is installed in the *SO-DIMM DDR5 Tester*.

A typical building command is:

```
export TARGET=sodimm_lpddr5_tester
make build TARGET_ARGS="--l2-size 256 --build --sys-clk-freq 50e6 --rw-bios --no-
↪sdram-hw-test"
```

The target can be customized with the following build parameters

- `--l2-size` sets L2 cache size
- `--sys-clk-freq` specifies system clock frequency
- `--no-sdram-hw-test` disables hardware accelerated memory test

To upload the bitstream to volatile FPGA configuration RAM use:

```
export TARGET=sodimm_lpddr5_tester
make upload
```

To write the bitstream into non-volatile (Q)SPI Flash memory use:

```
export TARGET=sodimm_lpddr5_tester
make flash
```

This targets an off-the-shelf DDR5 SO-DIMMs installed on Antmicro *SO-DIMM DDR5 Tester*. A typical building command is:

```
export TARGET=sodimm_ddr5_tester
make build TARGET_ARGS="--l2-size 256 --build --iodelay-clk-freq 400e6 --bios-lto_
↪--rw-bios --no-sdram-hw-test"
```

The target can be customized with the following build parameters

- --l2-size sets L2 cache size
- --iodelayclk-freq specifies IODELAY clock frequency
- --no-sdram-hw-test disables hardware accelerated memory test

To upload the bitstream to volatile FPGA configuration RAM use:

```
export TARGET=sodimm_ddr5_tester
make upload
```

To write the bitstream into non-volatile (Q)SPI Flash memory use:

```
export TARGET=sodimm_ddr5_tester
make flash
```

This targets off-the-shelf DDR5 RDIMMs installed on Antmicro *RDIMM DDR5 Tester*.

Note

Currently only revision 1.0 of the RDIMM DDR5 Tester is supported with bitstream target. Software integration and RDIMM DDR5 PHY development for the Artix UltraScale Plus FPGA found on the RDIMM DDR5 Tester in revision 2.0 is currently ongoing.

A typical building command is:

```
export TARGET=ddr5_tester
make build TARGET_ARGS="--l2-size 256 --build --iodelay-clk-freq 400e6 --bios-lto_
↪--rw-bios --module MTC10F1084S1RC --no-sdram-hw-test"
```

The target can be customized with the following build parameters

- --l2-size sets L2 cache size
- --iodelayclk-freq specifies IODELAY clock frequency
- --module specifies RDIMM DDR5 module family
- --no-sdram-hw-test disables hardware accelerated memory test

To upload the bitstream to volatile FPGA configuration RAM use:

```
export TARGET=ddr5_tester
make upload
```

To write the bitstream into non-volatile (Q)SPI Flash memory use:

```
export TARGET=ddr5_tester
make flash
```

This configuration allows testing of a single DDR5 IC. The hardware setup used in this scenario consists of an [LPDDR4 Tester](#) with a [DDR5 Test Bed](#). Since DDR5 and LPDDR4 support the same VDDQ IO voltages, it is possible to use the Tester Board to interface with a single DDR5 IC.

You can build this target with:

```
export TARGET=ddr5_test_board  
make build TARGET_ARGS="--l2-size 256 --build --iodelay-clk-freq 400e6 --bios-lto  
--rw-bios --no-sdram-hw-test"
```

The target can be customized with the following build parameters:

- `--l2-size` sets L2 cache size
- `--no-sdram-hw-test` disables hardware accelerated memory test

To upload the bitstream to volatile FPGA configuration RAM use:

```
export TARGET=ddr5_test_board  
make upload
```

To write the bitstream into non-volatile (Q)SPI Flash memory use:

```
export TARGET=ddr5_test_board  
make flash
```

This targets single LPDDR4 ICs soldered to interchangeable testbeds installed on Antmicro [LPDDR4 Test Board](#). You can build the target with:

```
export TARGET=lpddr4_test_board  
make build
```

To upload the bitstream to volatile FPGA configuration RAM use:

```
export TARGET=lpddr4_test_board  
make upload
```

To write the bitstream into non-volatile (Q)SPI Flash memory use:

```
export TARGET=lpddr4_test_board  
make flash
```

This targets off-the-shelf DDR4 SO-DIMMs installed on AMD-Xilinx [ZCU104](#). You can build the target with:

```
export TARGET=zcu104  
make build
```

The generated bitstream file must be named `zcu104.bit` and written to an SD card used for booting the board. Please refer to the [Loading the bitstream](#) section for more details.

This targets off-the-shelf DDR4 RDIMMs installed on Antmicro [RDIMM DDR4 Tester](#). You can build the target with:

```
export TARGET=ddr4_datacenter_test_board  
make build
```

To upload the bitstream to volatile FPGA configuration RAM use:

```
export TARGET=ddr4_datacenter_test_board  
make upload
```

To write the bitstream into non-volatile (Q)SPI Flash memory use:

```
export TARGET=ddr4_datacenter_test_board  
make flash
```

This is supported by the Digilent *Arty* boards. In this setup the Rowhammer Tester targets a single DDR3 IC installed on board. For Arty A7-100T with the XC7A100TCSG324-1 FPGA use:

```
export TARGET=arty  
make build TARGET_ARGS="--variant a7-100"
```

For Arty A7-35T with the XC7A35TICSG324-1L FPGA use:

```
export TARGET=arty  
make build
```

To upload the bitstream to volatile FPGA configuration RAM use:

```
export TARGET=arty  
make upload
```

To write the bitstream into non-volatile (Q)SPI Flash memory use:

```
export TARGET=arty  
make flash
```

Note

Running `make` will generate build files without invoking Vivado.

The generated bitstreams are stored in the `./build/<target-name>/gateware/` folder named after the respective target name used for building.

Note

The FPGA configuration RAM is a volatile memory so you would need to write the generated bitstream every time you power-cycle the board or reset the configuration state of the FPGA. The on-board FPGA will get automatically configured with a bitstream stored in the Flash memory on power-on. Please refer to the board-specific chapters (provided along with build instructions) for further information on how to connect the board to a host PC and how to configure it for uploading the bitstream.

3.2 Ethernet connection

The hardware platforms flashed with a generated bitstream can be accessed via Ethernet connection. The board's default IP address is 192.168.100.50 and you need to ensure the board and a host PC are registered within the same subnet (so, for example, you can use 192.168.100.2/24).

Note

In order to change the default IP address assigned to the board please set the IP_ADDRESS environment variable, rebuild the bitstream and re-upload it to the board.

Boards are controlled the same way for both simulation and hardware runs. In order to communicate with the board via EtherBone, start litex_server with the following command:

```
export IP_ADDRESS=192.168.100.50 # optional, should match the one used during  
→build  
make srv
```

The build files (CSRs address list) must be up-to-date. The build files can be re-generated with make.

Then, in another terminal, you can use the Python scripts provided. *Remember to enter the Python virtual environment before running the scripts!* Also, the TARGET variable should be set to load configuration for the given target. For example, to use the leds.py script, run the following:

```
source ./venv/bin/activate  
export TARGET=arty # (or zcu104) required to load target configuration  
cd rowhammer_tester/scripts/  
python leds.py # stop with Ctrl-C
```

3.3 Packaging the bitstream

To save the bitstream and use it later or share it, use the make pack utility target. It packs the files necessary to load the bitstream and run rowhammer scripts on it. These files are:

- build/\$TARGET/gateware/\$TOP.bit
- build/\$TARGET/csr.csv
- build/\$TARGET/defs.csv
- build/\$TARGET/sdram_init.py
- build/\$TARGET/litedram_settings.json

Running make pack creates a zip file named, for instance, \$TARGET-\$BRANCH-\$COMMIT.zip.

To use a bitstream packaged this way, run unzip your-bitstream-file.zip.

3.4 Building for simulation

Select TARGET, generate intermediate files & run the simulation:

```
export TARGET=arty  
make sim
```

This command will generate intermediate files & simulate them with Verilator. After simulation has finished, a signal dump can be investigated using [gtkwave](#):

```
gtkwave build/$TARGET/gateware/sim.fst
```

Warning

To run the simulation and the rowhammer scripts on a physical board at the same time, change the IP_ADDRESS variable, otherwise the simulation can conflict with the communication with your board.

1. Create the TUN interface:

```
tunctl -u $USER -t litex-sim
```

2. Configure the IP address of the interface:

```
ifconfig litex-sim 192.168.100.1/24 up
```

3. Optionally allow network traffic on this interface:

```
iptables -A INPUT -i litex-sim -j ACCEPT  
iptables -A OUTPUT -o litex-sim -j ACCEPT
```

Note

Typing `make ARGS="--sim"` will cause LiteX to only generate intermediate files and stop right after.

SO-DIMM DDR5 TESTER

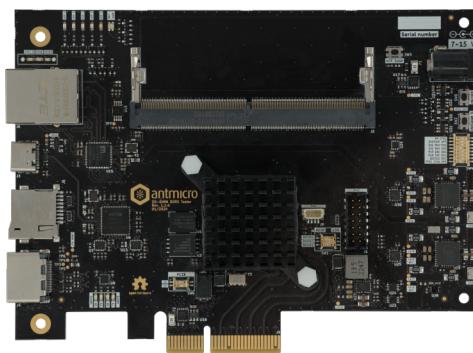


Fig. 4.1: SO-DIMM DDR5 Tester

The SO-DIMM DDR5 tester is an open source hardware test platform that enables testing and experimenting with various off-the-shelf DDR5 SO-DIMM modules. This board also supports testing single LPDDR5 ICs via [LPDDR5 Test Bed](#).

The hardware is open and can be found on GitHub: <https://github.com/antmicro/sodimm-ddr5-tester>

For FPGA digital design documentation for this board, refer to the [Digital design](#) chapter.

4.1 IO map

A map of on-board connectors, status LEDs, control buttons and I/O interfaces is provided in Fig. 4.2 below.

Connectors:

- J7 - main DC barrel jack power connector, voltage between 7-15 V is supported
- J3 - USB-C debug connector used for programming FPGA or Flash memory
- J1 - optional slot for a standard 14-pin JTAG connector used for programming FPGA or Flash memory
- J4 - HDMI connector
- J6 - Ethernet connector used for data exchange with on-board FPGA
- J2 - 262-pin SO-DIMM connector for connecting DDR5 memory modules

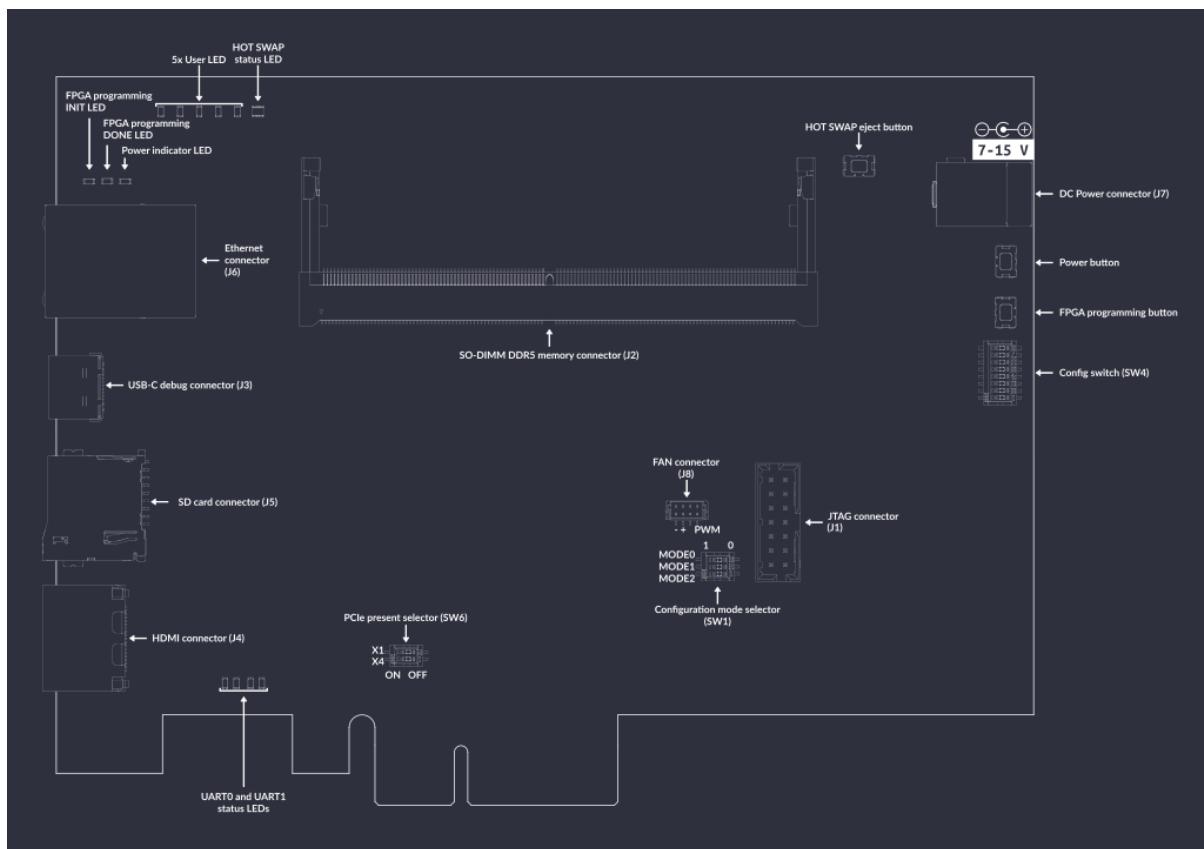


Fig. 4.2: SO-DIMM DDR5 tester interface map

- J8 - optional 5V fan connector
- J5 - socket for SD card

Switches and buttons:

- Power button SW5 - push button to power up the device, push button again to turn the device off
- Configuration mode selector SW1 - switch proper slides to specify programming mode (options described later)
- FPGA programming button SW2 - push button to start programming from Flash
- PCIe present selector SW6 - switch slide to set PCIe present to X1 and X4
- HOT SWAP eject button SW3 - reserved for future use to turn off DDR memory and allow hot swapping it
- Config switch SW4 - switch for setting several configuration options (options described later)

LEDs:

- 3V3 Power indicator D1 - indicates presence of stabilized 3.3V voltage
- FPGA programming INIT D3 - indicates current FPGA configuration state
- FPGA programming DONE D2 - indicates completion of FPGA programming
- 5x User (D4, D5, D6, D7, D8) - user-configurable LEDs

- HOT SWAP status D9 - RGY LED indicating status of hot swap process
- UART0 and UART1 status (D10, D11, D12, D13) - indicates status of RX/TX lines of UART protocols

4.2 Rowhammer Tester Target Configuration

The following instructions explain how to set up the board.

Set configuration mode selectors SW1 in proper positions. Using these 3 switches, you can set the following modes:

Configuration mode	MODE[2]	MODE[1]	MODE[0]
Master Serial	0	0	0
Master SPI	0	0	1
Master BPI	0	1	0
Master SelectMAP	1	0	0
JTAG	1	0	1
Slave SelectMAP	1	1	0
Slave Serial	1	1	1

For JTAG programming, set JTAG mode. If the bitstream needs to be loaded from the Flash memory, select Master SPI mode. This configuration is set by default. The bitstream will be loaded from flash memory upon device power-on or after a SW2 button press.

Set config switch SW4 in proper positions. The possible options are:

- PB- CTRL - automatically turn on board or use power button SW5
- INSTANT OFF - pushing power button SW5 will turn off device instantly or after 8s
- I2C MUX IN1 - configure switch of I2C on board (described below)
- I2C MUX IN2 - configure switch of I2C on board (described below)
- PWR I2C SDA - disable I2C in 3V3 powered IC's
- PWR I2C SCL - disable I2C in 3V3 powered IC's
- QDCDC2 I2C - enable I2C in main DCDC converter
- FTDI JTAG OFF - disable on-board FTDI which communicates with FPGA and Flash memory via JTAG

Configure power-up of I2C on board:

PWR I2C	IN1	IN2
OFF	L	L
FPGA DDR I2C	H	L
FPGA PWR I2C	L	H
FTDI	H	H

Connect power supply (7-15VDC) to J7 barrel jack. Then connect the board's USB-C J3 and Ethernet J6 interfaces to your computer, insert the memory module into the J2 socket and turn it on using power switch SW5.

RDIMM DDR5 TESTER

The RDIMM DDR5 Tester is an open source hardware test platform that enables testing and experimenting with various DDR5 RDIMMs (Registered Dual In-Line Memory Module).

The hardware is open and can be found on [GitHub](#).

For FPGA gateware documentation for this board, refer to the [*Gateware Documentation chapter*](#).

5.1 Variants

The latest design revision (Rev. 2.0) is based on AMD (Xilinx) UltraScale Plus FPGA which offers more logic resources operating at higher throughput. A more detailed comparison between Kintex-7 and UltraScale Plus RDIMM DDR5 Tester can be found in this [blog note](#).



Fig. 5.1: RDIMM DDR5 Tester in revision 2.0.0

5.2 IO map

A map of on-board connectors, status LEDs, control buttons and I/O interfaces is provided in Fig. 5.3 below.

Connectors:

- J7 - main DC barrel jack power connector, voltage between 7-15 V is supported
- J3 - USB-C debug connector used for programming FPGA or Flash memory
- J4 - HDMI output connector

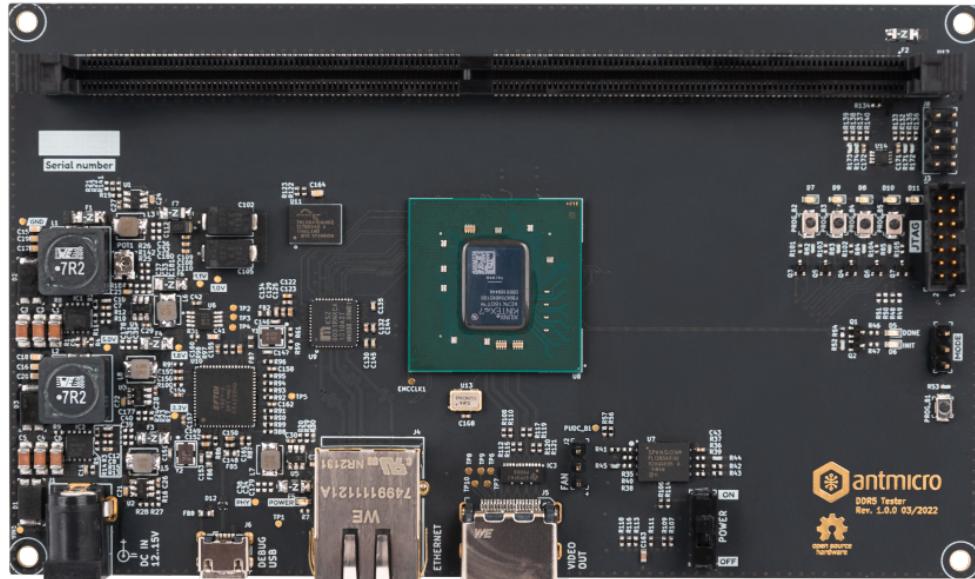


Fig. 5.2: RDIMM DDR5 Tester in revision 1.0.0

- J11 - HDMI input connector
- J6 - Ethernet connector used for data exchange with on-board FPGA
- J2 - 288-pin RDIMM connector for connecting DDR5 memory modules
- J8 - optional 5V fan connector
- J5 - socket for SD card
- J12 - I2C/I3C connector
- J17 - Heater connector (maximum 10W)
- J1 - optionally mounted standard 14-pin JTAG connector used for programming FPGA or Flash memory

Switches and buttons:

- Power button SW5 - push button to power up the device, push button again to turn the device off
- FPGA programming button SW2 - push button to start programming from Flash
- PCIe present selector SW1 - switch slide to set PCIe present to X1, X4 and X8
- HOT SWAP eject button SW3 - reserved for future use to turn off DDR memory and allow hot swapping it
- Config switch SW4 - switch for setting several configuration options (options described later)

LEDs:

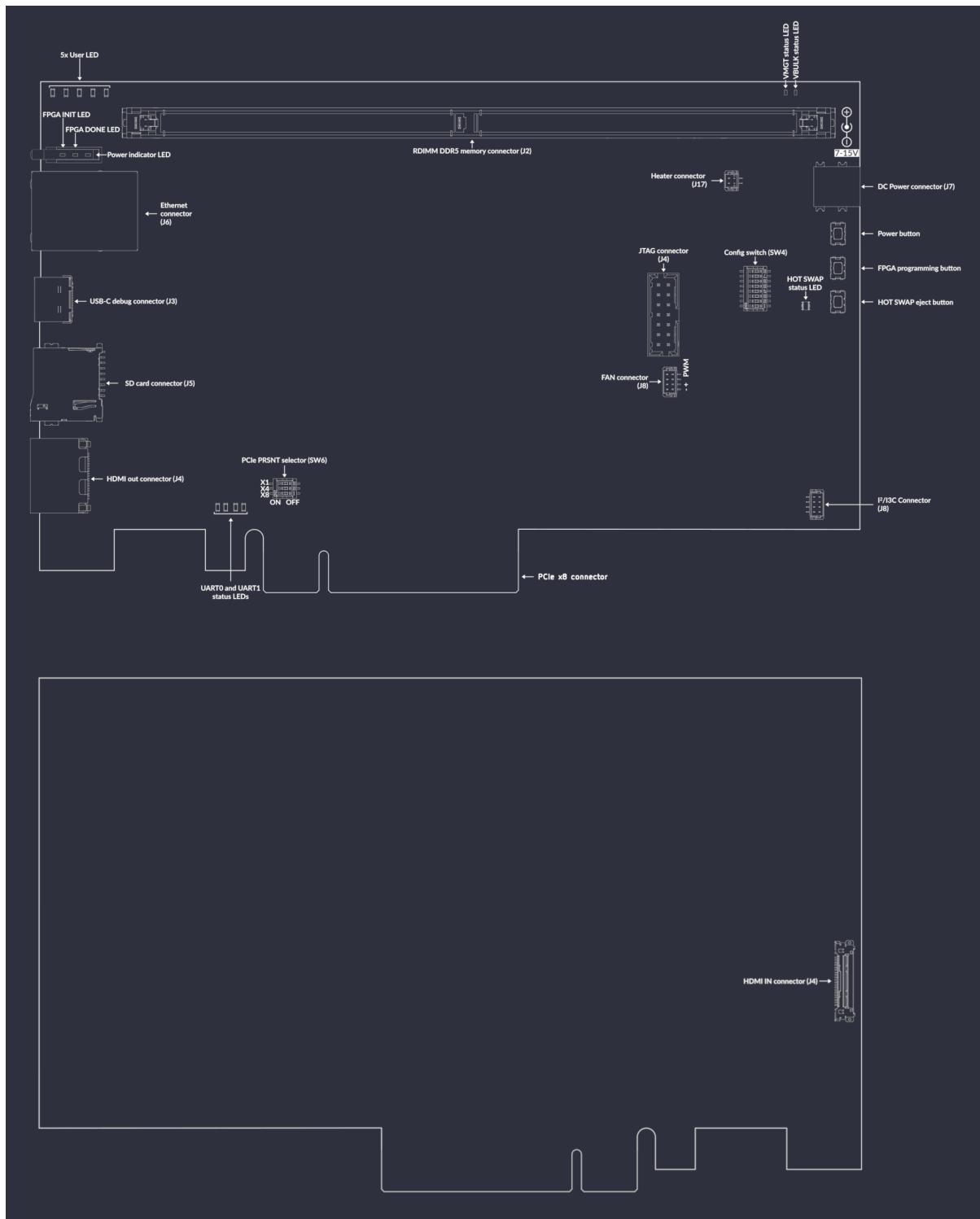


Fig. 5.3: DDR5 tester interface map

- Power good indicator D1 - indicates presence of all FPGA voltage rails stabilized
- FPGA programming INIT D2 - indicates current FPGA configuration state
- FPGA programming DONE D3 - indicates completion of FPGA programming
- 5x User (D8, D7, D6, D5, D4) - user-configurable LEDs
- VBULK indicator D21 - indicates presence of VBULK DDR5 voltage rail stabilized
- VMGT indicator D22 - indicates presence of VMGT DDR5 voltage rail stabilized

A map of on-board connectors, status LEDs, control buttons and I/O interfaces is provided in Fig. 5.4.

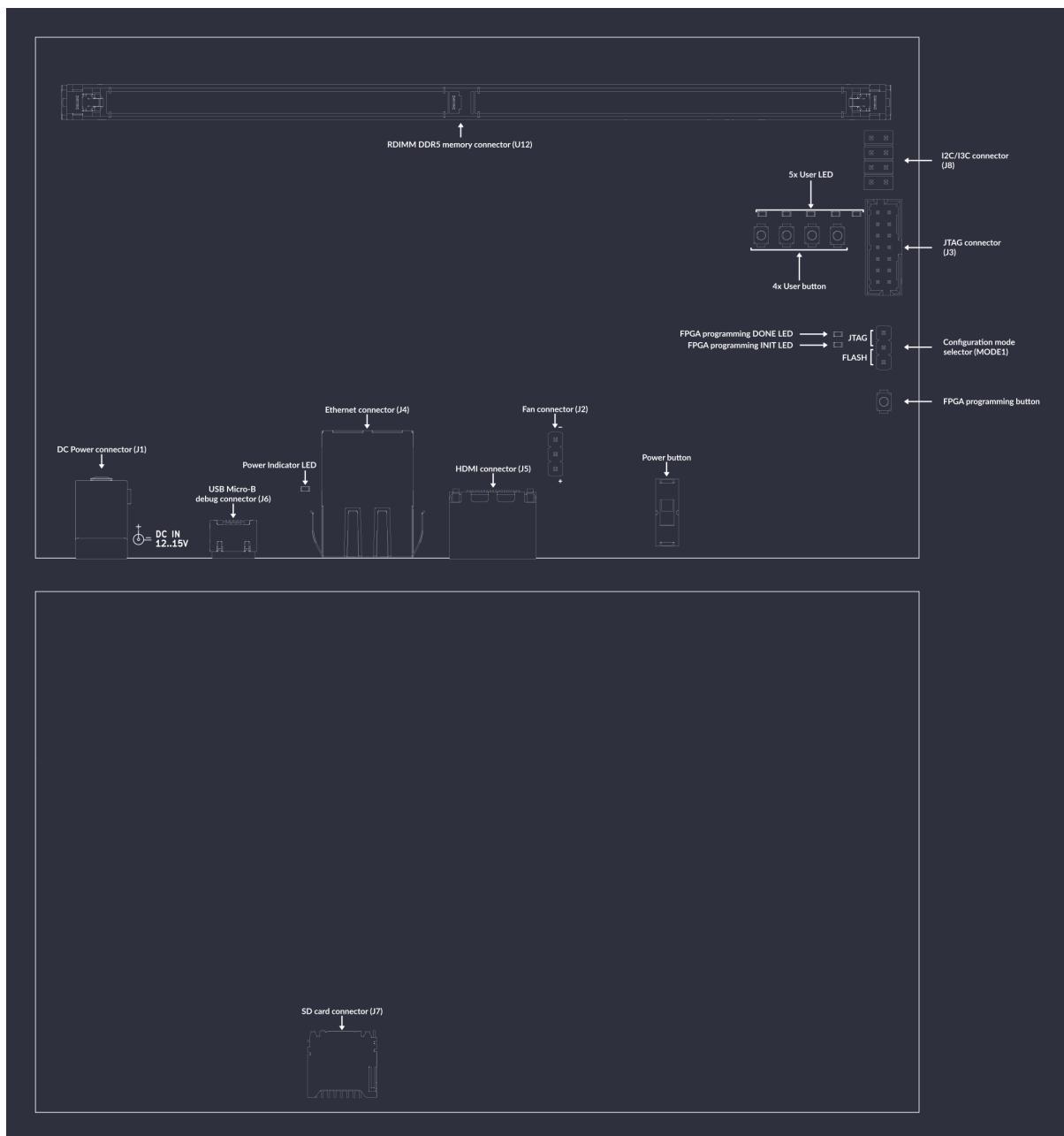


Fig. 5.4: DDR5 tester interface map

Connectors:

- J1 - main DC barrel jack power connector, voltage between 12-15V is supported
- J6 - USB Micro-B debug connector used for programming FPGA or Flash memory
- J3 - standard 14-pin JTAG connector used for programming FPGA or Flash memory
- J5 - HDMI connector
- J4 - Ethernet connector used for data exchange with on-board FPGA
- U12 - 288-pin RDIMM connector for connecting DDR5 memory modules
- MODE1 - configuration mode selector, short proper pins with jumper to specify programming mode
- J2 - optional 5V fan connector
- J7 - socket for SD card
- J8 - 2.54mm goldpin connector with exposed I2C and I3C signals

Switches and buttons:

- Power ON/OFF button S1 - slide up to power up a device, slide down to turn off the device
- FPGA programming button PROG_B1 - push button to start programming from Flash
- 4x User button (PROG_B2, PROG_B3, PROG_B4, PROG_B5) - user-configurable buttons

LEDs:

- 3V3 Power indicator PWR1 - indicates presence of stabilized 3.3V voltage
- FPGA programming INIT D6 - indicates current FPGA configuration state
- FPGA programming DONE D5 - indicates completion of FPGA programming
- 5x User (D7, D8, D9, D10, D11) - user-configurable LEDs

5.3 Rowhammer Tester Target Configuration

The following instructions explain how to set up the board depending on the revision you have.

Set the config switch SW4 in proper positions. The possible options are:

- PB- CTRL - automatically turn on board or use power button SW5
- INSTANT OFF - pushing power button SW5 will turn off device instantly or after 8s
- I2C MUX IN1 - configure switch of I2C on board (described below)
- I2C MUX IN2 - configure switch of I2C on board (described below)
- JTAG/ SPI Flash - boot mode switch
- USER IN - User configurable input
- QDCDC2 I2C - enable I2C in main DCDC converter
- FTDI JTAG OFF - disable on-board FTDI which communicates with FPGA and Flash memory via JTAG

Configure power-up of I2C on board:

PWR I2C	IN1	IN2
OFF	L	L
FPGA DDR I2C	H	L
FPGA PWR I2C	L	H
FTDI	H	H

Connect power supply (12-15VDC) to the J1 barrel jack. Then connect the board USB cable (J6) and Ethernet cable (J4) to your computer and insert the memory module to the socket U12. To turn on the board, use the power switch S1. After power is up, configure the network and upload the bitstream.

There is a JTAG/SPI switch (MODE1) on the right-hand side of the board. Unless it's set to the SPI setting, the FPGA will load the bitstream received via JTAG (J3).

The bitstream will be loaded from flash memory upon device power-on or after pressing the PROG_B1 button.

5.4 Simulation

The simulation is based on a DDR5 DRAM model ([sdram_simulation_model.py](#)) and a DDR5 PHY simulation model ([simphy.py](#)). These models are used by the SoC simulation model ([simsoc.py](#)).

Start the simulation with:

```
python3 third_party/litedram/litedram/phy/ddr5/simsoc.py --no-masked-write --with-
    ↪sub-channels --dq-dqs-ratio 4 --modules-in-rank 1 --log-level error --skip-csca_
    ↪--skip-reset-seq --skip-mrs-seq --with-prompt --l2-size 256 --uart-name serial
```

RDIMM DDR4 TESTER



Fig. 6.1: Data Center RDIMM DDR4 Tester

The Data Center RDIMM DDR4 Tester is an open source hardware test platform that enables testing and experimenting with various DDR4 RDIMMs (Registered Dual In-Line Memory Module).

The hardware is open and can be found on GitHub: <https://github.com/antmicro/rdimm-ddr4-tester>

The following instructions explain how to set up the board.

For FPGA digital design documentation for this board, refer to the *Digital design* chapter.

6.1 IO map

A map of on-board connectors, status LEDs, control buttons and I/O interfaces is provided in Fig. 6.2 below.

Connectors:

- J3 - main DC barrel jack power connector, voltage between 7-15V is supported
- J9 - USB-C debug connector used for programming FPGA or Flash memory
- J1 - standard 14-pin JTAG connector used for programming FPGA or Flash memory
- J6 - HDMI connector

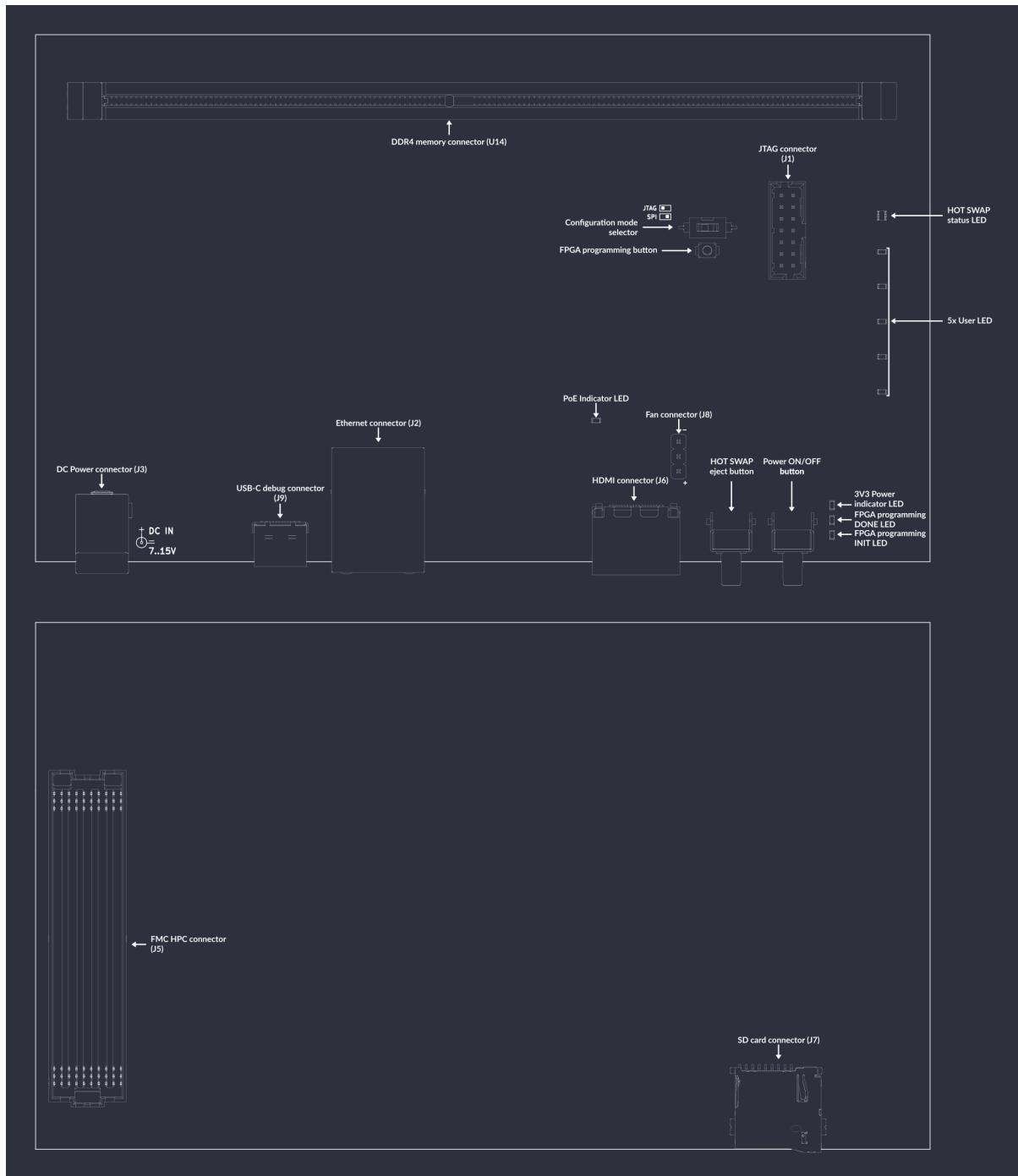


Fig. 6.2: DDR4 data center dram tester interface map

- J2 - Ethernet connector used for data exchange with on-board FPGA and power supply via PoE
- U14 - 288-pin RDIMM connector for connecting DDR4 memory modules
- J8 - optional 5V fan connector
- J7 - socket for SD card
- J5 - FMC HPC connector reserved for future use

Switches and buttons:

- Power ON/OFF button S3 - push button to power up the device, hold for 8s to turn off the device
- FPGA programming button PROG_B1 - push button to start programming from Flash
- Configuration mode selector S2 - Switch left/right to specify SPI/JTAG programming mode
- HOT SWAP eject button S1 - reserved for future use to turn off DDR memory and allow hot swapping it

LEDs:

- 3V3 Power indicator PWR1 - indicates presence of stabilized 3.3V voltage
- PoE indicator D15 - indicates negotiated PoE voltage supply
- FPGA programming INIT D10 - indicates current FPGA configuration state
- FPGA programming DONE D1 - indicates completion of FPGA programming
- HOT SWAP status D17 - RGY LED indicating status of hot swap process
- 5x User (D5, D6, D7, D8, D9) - user-configurable LEDs

6.2 Board configuration

Connect power supply (7-15VDC) to J3 barrel jack. Then connect the board USB cable (J9) and Ethernet cable (J2) to your computer and insert the memory module to the socket U14. To turn the board on, use power switch S3.

After power is up, configure the network and prepare the board for uploading the bitstream.

A JTAG/SPI switch S2 on the right side of the board (near the JTAG connector) defines whether the bitstream is loaded via JTAG or SPI Flash memory. Bitstream will be loaded from flash memory upon device power-on or after pressing the PROG_B1 button.

CHAPTER SEVEN

LPDDR4 TEST BOARD

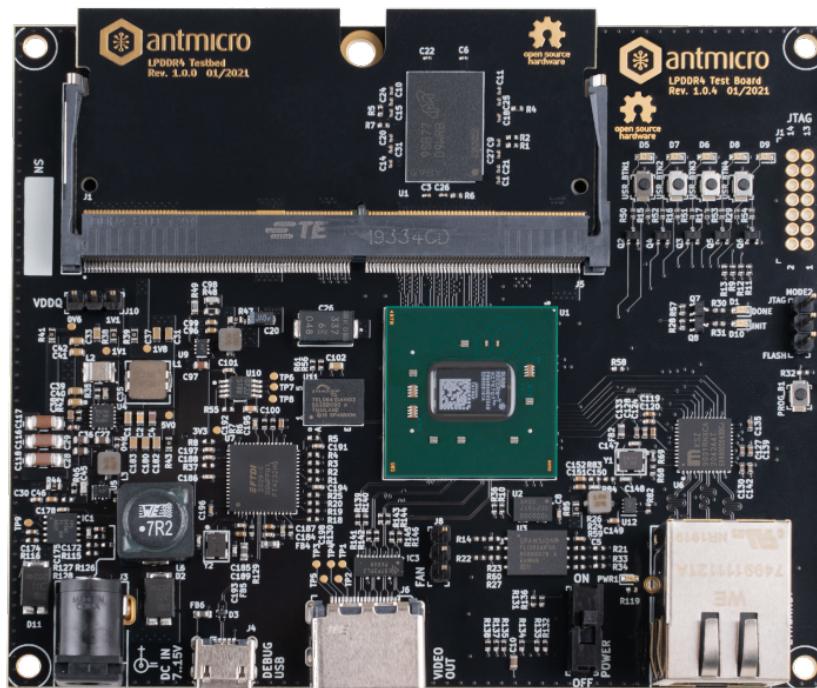


Fig. 7.1: LPDDR4 Test Board

The LPDDR4 Test Board is a platform developed by Antmicro for testing LPDDR4 memory. It uses the Xilinx Kintex-7 FPGA (XC7K70T-FBG484).

The hardware is open and can be found on GitHub (<https://github.com/antmicro/lpddr4-test-board>).

This Test Board supports interchangeable Test Beds that can be populated with various memories. Initially the LPDDR4 Test Board was meant to be used with *LPDDR4 Test Beds*. Currently it also supports *DDR5 Test Beds*.

For FPGA digital design documentation for this board, refer to the *Digital design* chapter.

7.1 IO map

A map of on-board connectors, status LEDs, control buttons and I/O interfaces is provided in Fig. 7.2 below.

Connectors:

- J6 - main DC barrel jack power connector, voltage between 7-15V is supported
- J1 - USB Micro-B debug connector used for programming FPGA or Flash memory
- J4 - standard 14-pin JTAG connector used for programming FPGA or Flash memory
- J2 - HDMI connector
- J5 - Ethernet connector used for data exchange with on-board FPGA
- J9 - 260-pin SO-DIMM connector for connecting LPDDR4 memory
- MODE1 - configuration mode selector, short proper pins with jumper to specify programming mode
- J7 - VDDQ selector used for specifying value of VDDQ voltage
- J8 - optional 5V fan connector
- J3 - socket for SD card

Switches and buttons:

- Power switch S1 - slide up to power up a device, slide down to turn the device off
- FPGA programming button PROG_B1 - push to start programming from Flash
- 4x User button (USR_BTN1,USR_BTN2,USR_BTN3,USR_BTN4) - user-configurable buttons

LEDs:

- Power indicators (PWR1, PWR2, PWR3, PWR4, PWR5, PWR6) - indicates presence of stabilized voltages: 5V, 3V3, 1V8, 1V2, 1V1, 1V0
- FPGA programming INIT D9 - indicates current FPGA configuration state
- FPGA programming DONE D8 - indicates completion of FPGA programming
- 5x User (D1, D2, D3, D5, D6) - user-configurable LEDs

7.2 Board configuration

First insert the LPDDR4 testbed into the socket J9 and make sure that jumpers are set in correct positions:

- the VDDQ switch (J7) should be set in the 1V1 position
- MODE1 switch should be set in the FLASH position

Connect power supply (7-15VDC) to J6 barrel jack. Then connect the board's USB-C J1 and Ethernet J5 interfaces to your computer. Turn on the board using power switch S1. Then configure the network. There is a JTAG/FLASH jumper MODE1 on the right-hand side of the board. It defines whether the bitstream is loaded via JTAG or FLASH memory. The bitstream will be loaded from flash memory upon device power-on or after the FPGA programming PROG_B1 button is pressed.

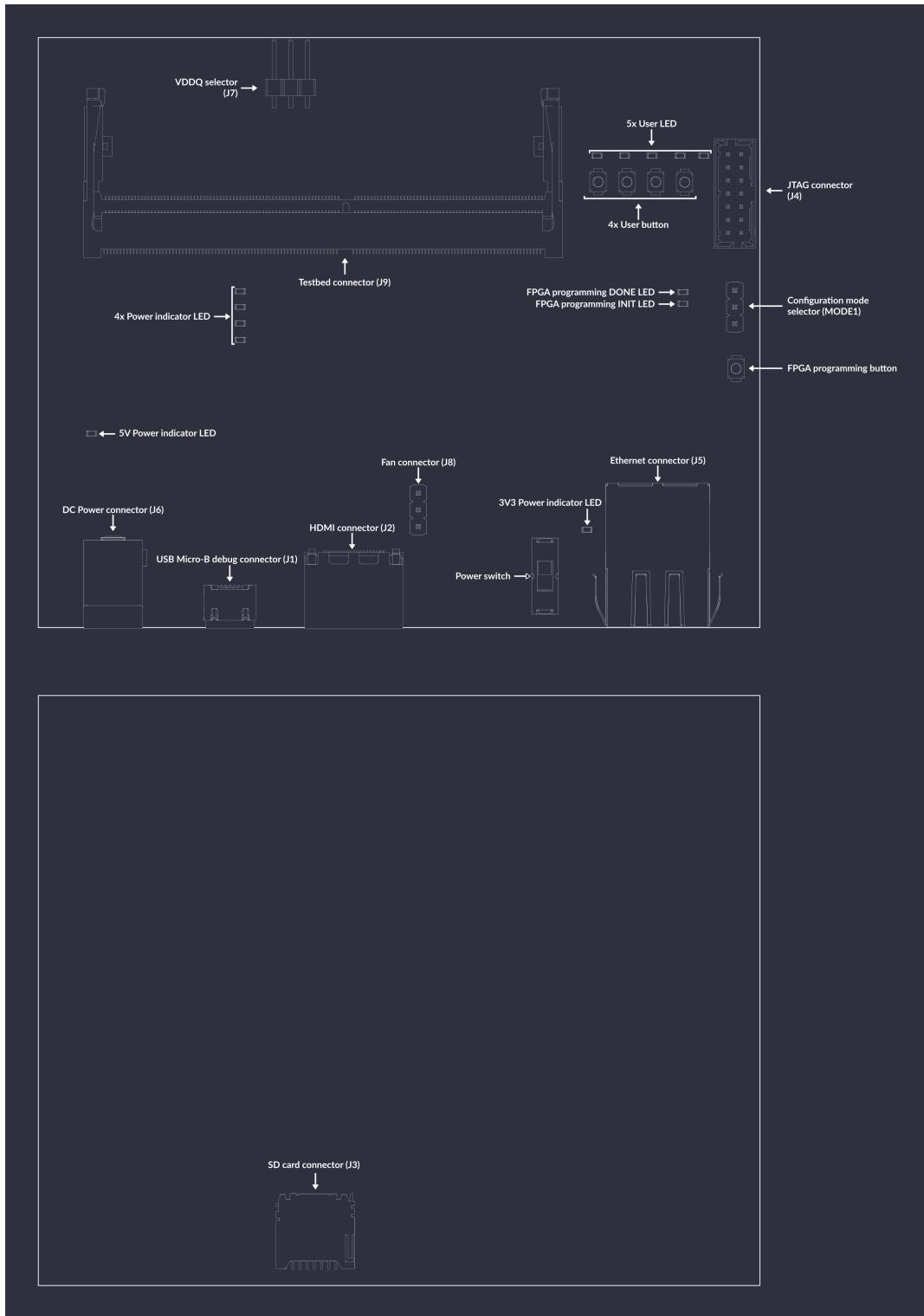


Fig. 7.2: LPDDR4 test board interface map

CHAPTER EIGHT

ZCU104 BOARD

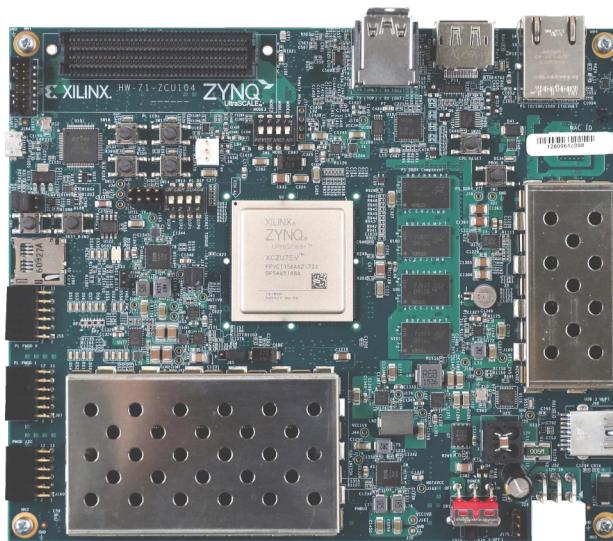


Fig. 8.1: ZCU104 board

The [ZCU104 board](#) enables testing DDR4 SO-DIMM modules. It features a Zynq UltraScale+ MPSoC device consisting of a Processing System (PS) with quad-core ARM Cortex-A53 and programmable logic (PL).

On the ZCU104 board, the Ethernet PHY is connected to PS instead of PL. For this reason, it is necessary to route the Ethernet/EtherBone traffic as follows :PC <-> PS <-> PL. A simple EtherBone server is implemented for this purpose (the source code can be found in the `firmware/zcu104/etherbone/` directory).

The following instructions show how to set up the board for the first time.

For FPGA design documentation for this board, refer to the [Digital design](#) chapter.

8.1 Board configuration

To make the ZCU104 boot from an SD card, it is necessary to ensure proper switch configuration. The mode switch (SW6) consisting of 4 switches is located near the FMC LPC Connector (J5) (the same side of the board as USB, HDMI, Ethernet). For details, refer to the [ZCU104 Evaluation Board User Guide \(UG1267\)](#). To use an SD card, configure the switches as follows:

1. ON
2. OFF

3. OFF
4. OFF

8.2 Preparing SD card

For a basic, simple setup, get the pre-built SD card image `zcu104.img` from [github releases](#) and load it to a microSD card. To load it to the SD card, insert the card into your PC card slot and find the device name. `lsblk` can be used to check for available devices. An example output looks as follows:

```
$ lsblk
NAME      MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda        8:0    0 931.5G  0 disk
└─sda1     8:1    0 931.5G  0 part /data
sdb        8:16   1 14.8G  0 disk
└─sdb1     8:17   1  128M  0 part /run/media/BOOT
└─sdb2     8:18   1  128M  0 part /run/media/rootfs
nvme0n1   259:0  0 476.9G  0 disk
└─nvme0n1p1 259:1  0  512M  0 part /boot
└─nvme0n1p2 259:2  0 476.4G  0 part /
```

In the output above, the SD card is `sdb` with two partitions `sdb1` and `sdb2`.

 **Warning**

Make sure to select the proper device name to avoid damaging the hard drive in your system! Assure the device `SIZE` matches the capacity of your microSD card and compare the outputs of the `lsblk` command with and without the SD card inserted.

Make sure to unmount all partitions on the card before loading the image. For example, assuming the SD card is `/dev/sdb` (device is without a number), use `sudo umount /dev/sdb1` `/dev/sdb2` to unmount its partitions.

To load the image, use the following command, replacing `<DEVICE>` according to the output of `lsblk` (in the example above it would be `/dev/sdb`):

```
sudo dd status=progress oflag=sync bs=4M if=zcu104.img of=<DEVICE>
```

Now, the microSD card should be ready to use. If it is loaded successfully, you will be able to mount the two partitions (`BOOT` and `rootfs`) on your PC and browse the files. First, check whether your system auto-mounted the partitions. If not, you can use:

```
sudo mkdir -p /mnt/boot /mnt/rootfs
sudo mount /dev/sdb1 /mnt/boot
sudo mount /dev/sdb2 /mnt/rootfs
```

8.3 Loading the bitstream

Instead of loading the bitstream through the JTAG interface, copy it to the microSD card BOOT partition (FAT32). The bitstream will be then loaded by the bootloader during system startup.

The prebuilt card image comes with a sample bitstream, but in order to use the provided rowhammer Python scripts, you need to create a fresh bitstream.

Copy it to the BOOT partition (FAT32) of the microSD card. Make sure it is named `zcu104.bit`.

When the SD card is ready, insert it into the microSD card slot on your ZCU104 board and power on the board.

Onboard LEDs are the first indication that the bitstream is loaded successfully. When the board is powered up, the LED will be red and then turn green if the bitstream is loaded successfully. The ZCU104 bitstream will also make the four LEDs near the user buttons turn on and off in a circular pattern. The serial console over USB can be used to further check if the board is working correctly.

8.4 ZCU104 microUSB

ZCU104 has a microUSB port connected to the FTDI chip. It provides 4 channels connected as follows:

- Channel A is configured to support the JTAG chain.
- Channel B implements UART0 MIO18/19 connections.
- Channel C implements UART1 MIO20/21 connections.
- Channel D implements an UART2 PL-side bank 28 4-wire interface.

The channels should show up as subsequent `/dev/ttyUSBx` devices (0-3 if no others were present). Channel B is connected to the console in the PS Linux system.

To log in to the board, connect the microUSB cable to the PC and find Channel B among the `ttyUSB` devices in your system. If only `ttyUSB0` through `ttyUSB3` are visible, then Channel B will be `ttyUSB1`.

Open the serial console using e.g. `picocom` or `minicom` (you may need to install one). With `picocom`, use the following command (may require `sudo`):

```
picocom -b 115200 /dev/ttyUSB1
```

Press enter. When you see the following prompt:

```
Welcome to Buildroot  
buildroot login:
```

Use `root` as login and leave password empty. You can set up a password if needed.

8.5 Network setup

Connect the ZCU104 board to your local network (or directly to a PC) using an Ethernet cable.

The board uses a static IP address - 192.168.100.50 by default. If it does not conflict with your local network configuration, you can skip this section. You can find the default configuration [here](#).

To verify connectivity, use `ping 192.168.100.50`. You should see data being transmitted, e.g.:

```
$ ping 192.168.100.50
PING 192.168.100.50 (192.168.100.50) 56(84) bytes of data.
64 bytes from 192.168.100.50: icmp_seq=1 ttl=64 time=0.332 ms
64 bytes from 192.168.100.50: icmp_seq=2 ttl=64 time=0.072 ms
64 bytes from 192.168.100.50: icmp_seq=3 ttl=64 time=0.081 ms
```

8.5.1 Modifying the network configuration

If you need to modify the configuration, edit the `/etc/network/interfaces` file. The Linux rootfs on the SD card is fairly minimal and `vi` is the only editor available. You can also mount the card on your PC and edit the file.

After changing the configuration, reboot the board (type `reboot` in the serial console) and test if you can ping it with `ping <NEW_IP_ADDRESS>`.

8.6 SSH access

These instructions are optional but can be useful for more convenient updates of the bitstream (no need to remove the SD card from the slot on ZCU104).

Note

SSH on the board is configured to allow passwordless root access for simplicity but if that poses a security risk, modify `/etc/ssh/sshd_config` according to your needs and add a password for root.

You can log in over SSH using (replace the IP address if you modified board network configuration):

```
ssh root@192.168.100.50
```

To access the boot partition, first mount it with:

```
mount /dev/mmcblk0p1 /boot
```

This can be automated by adding the following entry in `/etc/fstab`:

<code>/dev/mmcblk0p1</code>	<code>/boot</code>	<code>vfat</code>	<code>rw</code>	<code>0</code>	<code>2</code>
-----------------------------	--------------------	-------------------	-----------------	----------------	----------------

When the boot partition is mounted, you can use `scp` to load the new bitstream, e.g.

```
scp build/zcu104/gateware/zcu104.bit root@192.168.100.50:/boot/zcu104.bit
```

Then use the `reboot` command to restart the board.

8.7 Controlling the board

When the setup has been finished the board can be controlled as any other board. Make sure to use `export TARGET=zcu104` before using the scripts (and `export IP_ADDRESS=...` if you modified the network configuration).

8.8 ZCU104 SD card image

The easiest way is to use the prebuilt SD card image. It is also possible to build the image from source if needed.

The SD card image consists of a boot partition and a rootfs. Currently, only rootfs is built using buildroot. The boot partition contents have to be built manually.

8.8.1 Bootloaders & kernel

Currently, we are using Xilinx FSBL, but it should be possible to use U-Boot SPL ([link1](#), [link2](#)).

FSBL and PMU firmware can be built with following the steps:

- <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842462/Build+PMU+Firmware>
- <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841798/Build+FSBL>

Create a project from the Vivado example project “Base Zynq UltraScale+ MPSoC” for ZCU104 eval board. Open the PS IP configurator and add the following:

- PS-PL Interfaces -> AXI HPM0 FPD (32-bit), AXI HPM1 FPD (32-bit)
- disable Carrier Detect in Memory Interfaces -> SD -> SD 0

The following script can be used to generate FSBL, PMU firmware and Device Tree:

```
#!/usr/bin/tclsh

set hwsgn [open_hw_design PATH/T0/Base_Zynq_MPSoC_wrapper.hdf]

generate_app -hw $hwsgn -os standalone -proc psu_cortexa53_0 -app zynqmp_fsbl \
    ↴compile -sw fsbl -dir ./fsbl/
generate_app -hw $hwsgn -os standalone -proc psu_pmu_0 -app zynqmp_pmufw \
    ↴compile -sw pmufw -dir ./pmufw

set_repo_path PATH/T0/device-tree-xlnx
create_sw_design device-tree -os device_tree -proc psu_cortexa53_0
generate_target -dir dts

close_hw_design [current_hw_design]
```

The Device Tree generated by Vivado is missing the ethernet-phy node. Modify `pcw.dtsi` as follows:

```
&gem3 {
    phy-mode = "rgmii-id";
    status = "okay";
```

(continues on next page)

(continued from previous page)

```
xlnx,ptp-enet-clock = <0x0>;
phy0: phy@c {
    reg = <0xc>;
    ti,rx-internal-delay = <0x8>;
    ti,tx-internal-delay = <0xa>;
    ti,fifo-depth = <0x1>;
    ti,rxctrl-strap-worka;
};
};
```

Then generate the Device Tree Blob in the dts directory:

```
gcc -I include -I . -E -nostdinc -undef -D__DTS__ -x assembler-with-cpp -o system.
↳dts system-top.dts
dtc -I dts -O dtb -o system.dtb system.dts
```

Build the rest of the required components:

- ARM trusted firmware: (<https://github.com/Xilinx/arm-trusted-firmware.git>) e6eea88b14aaaf456c49f9c7e6747584224648cb9 (tag: xlnx_rebase_v2.2)
- U-Boot: (<https://github.com/Xilinx/u-boot-xlnx.git>) d8fc4b3b70bccf1577dab69f6ddfd4ada9a93bac (tag: xilinx-v2018.3)
- Linux kernel: (<https://github.com/Xilinx/linux-xlnx.git>) 22b71b41620dac13c69267d2b7898ebfb14c954e (tag: xlnx_rebase_v5.4_2020.1)

Note

It may be necessary to apply the patches from `firmware/zcu104/buildroot/board/zynqmp/patches` when building U-Boot/Linux.

When building U-Boot, make sure to update its configuration (`u-boot-xlnx/.config`) with the following options:

```
CONFIG_USE_BOOTARGS=y
CONFIG_BOOTARGS="earlycon clk_ignore_unused console=ttyPS0,115200 root=/dev/
↳mmcblk0p2 rootwait rw earlyprintk rootfstype=ext4"
CONFIG_USE_BOOTCOMMAND=y
CONFIG_BOOTCOMMAND="load mmc 0:1 0x2000000 zcu104.bit; fpga load 0 0x2000000
↳$filesize; load mmc 0:1 0x2000000 system.dtb; load mmc 0:1 0x3000000 Image;_
↳booti 0x3000000 - 0x2000000"
```

These configure U-Boot to load the bitstream from the SD card and then start the system. When unfolding `CONFIG_BOOTCOMMAND`, we can see:

```
load mmc 0:1 0x2000000 zcu104.bit
fpga load 0 0x2000000 $filesize
load mmc 0:1 0x2000000 system.dtb
load mmc 0:1 0x3000000 Image
booti 0x3000000 - 0x2000000
```

Example of building ARM Trusted firmware:

```
make distclean
make -j`nproc` PLAT=zynqmp RESET_TO_BL31=1
```

Example of building U-Boot:

```
make -j`nproc` distclean
make xilinx_zynqmp_zcu104_revC_defconfig
# now modify .config directly or using `make menuconfig` as described earlier
make -j`nproc`
```

Example of building Linux:

```
make -j`nproc` ARCH=arm64 distclean
make ARCH=arm64 xilinx_zynqmp_defconfig
# optional `make menuconfig`
make -j`nproc` ARCH=arm64 dtbs
make -j`nproc` ARCH=arm64
```

Then download [zynq-mkbootimage](#) and prepare the following boot.bif file:

```
image:
{
    [fsbl_config] a53_x64
    [bootloader] fsbl.elf
    [pmufw_image] pmufw.elf
    [, destination_cpu=a53-0, exception_level=el-2] bl31.elf
    [, destination_cpu=a53-0, exception_level=el-2] u-boot.elf
}
```

Open a terminal and make sure that the filepaths specified in boot.bif are correct. Then use ``mkbootimage -zynqmp boot.bif boot.bin`` to create the boot.bin file.

8.8.2 Root filesystem

Download buildroot:

```
git clone git://git.buildroot.net/buildroot
git checkout 2020.08.2
```

Note

As of time of writing git checkout f45925a951318e9e53bead80b363e004301adc6f was required to avoid fakeroot errors when building.

Then prepare configuration using external sources and build everything:

```
make BR2_EXTERNAL=/PATH/TO/REPO/rowhammer-tester/firmware/zcu104/buildroot zynqmp_
↳zcu104_defconfig
make -j`nproc`
```

8.8.3 Flashing SD card

You can use `fdisk` to directly partition the SD card `/dev/xxx` or use the provided `genimage` configuration to create an SD card image that can be then directly copied to the SD card. The second method is usually more convenient.

Formatting SD card manually

Use `fdisk` or other tool to partition the SD card. The recommended partitioning scheme is as follows:

- Partition 1, FAT32, 128M
- Partition 2, ext4, 128M

Then create the filesystems:

```
sudo mkfs.fat -F 32 -n BOOT /dev/OUR_SD_CARD_PARTITION_1
sudo mkfs.ext4 -L rootfs /dev/OUR_SD_CARD_PARTITION_2
```

Write the rootfs:

```
sudo dd status=progress oflag=sync bs=4M if=/PATH/T0/BUILDROOT/output/images/
↪rootfs.ext4 of=/dev/OUR_SD_CARD_PARTITION_2
```

Mount the boot partition and copy the boot files and kernel image created earlier as well as the ZCU104 bitstream:

```
cp boot.bin /MOUNT/POINT/BOOT/
cp /PATH/T0/rowhammer-tester/build/zcu104/gateware/zcu104.bit /MOUNT/POINT/BOOT/
cp /PATH/T0/linux-xlnx/arch/arm64/boot/Image /MOUNT/POINT/BOOT/
cp /PATH/T0/linux-xlnx/arch/arm64/boot/dts/xilinx/zynqmp-zcu104-revA.dtb /MOUNT/
↪POINT/BOOT/system.dtb
```

Note: make sure to name the device tree blob `system.dtb` for the U-Boot to be able to find it (as shown in above commands).

Using genimage

The ZCU104 buildroot configuration will also build the `genimage` tool for the host system by default. The image configuration is described in the `firmware/zcu104/image.cfg` file. A script named `firmware/zcu104/genimage.sh` is also provided for convenience. Run it without arguments to get help. Then run it, providing correct paths to all the required files, to generate the `zcu104.img` file.

The image can be then copied to the SD card device (not partition! so e.g. `/dev/sdb`, not `/dev/sdb1`) using `dd`:

```
sudo dd status=progress oflag=sync bs=4M if=/PATH/T0/zcu104.img of=/dev/OUR_SD_
↪CARD
```

ARTY-A7 BOARD

The Arty-A7 board allows testing its on-board DDR3 module. The board is designed around the Artix-7 Field Programmable Gate Array (FPGA) from AMD(Xilinx).

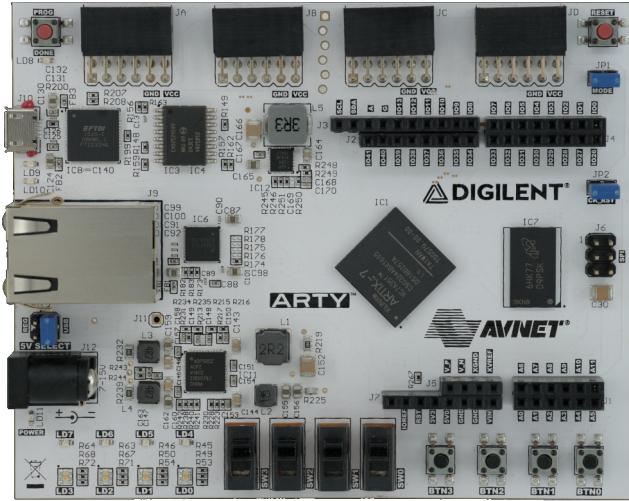


Fig. 9.1: Arty-A7 board

The following instructions explain how to set up the board. For FPGA digital design documentation for this board, refer to the *Digital design* chapter.

9.1 Board configuration

Connect the board USB and Ethernet cables to your computer and configure the network. The bitstream will be loaded from flash memory upon device power-on or after pressing the PROG button.

LPDDR4 TEST BED



Fig. 10.1: LPDDR4 Test Bed

This accessory allows interfacing with a single LPDDR4 ICs using the [LPDDR4 Test Board](#). The hardware design of the LPDDR4 Test Bed is released to GitHub (<https://github.com/antmicro/lpddr4-testbed>) as open source hardware. The hardware design includes Micron MT53E1G32D2NP-046. The LPDDR4 Test Bed exposes only one memory channel (channel A) to the FPGA located on a matching test board.

Warning

The LPDDR4 Test Bed has a form factor that is mechanically compatible with SO-DIMM DDR4 sockets. The pinout of the LPDDR4 Test Bed does not match SO-DIMM DDR4 specification so it cannot be used in systems supporting off-the-shelf SO-DIMM DDR4 memories.

DDR5 TEST BED

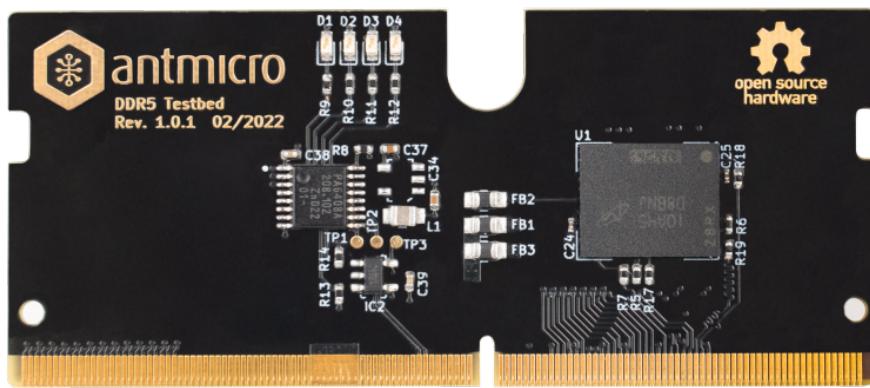


Fig. 11.1: DDR5 Test Bed

This accessory allows interfacing with a single DDR5 ICs using the [LPDDR4 Test Board](#). The hardware design of the DDR5 Test Bed is released to GitHub (<https://github.com/antmicro/ddr5-testbed>) as open source hardware. The hardware design of the DDR5 Test Bed includes a single Micron [MT60B2G8HB-48B:A](#) 16Gb DRAM. The DDR5 Test Bed exposes only one memory channel (channel A) to the FPGA located on a matching test board.

The DDR5 IC needs to be soldered to a custom [DDR5 Testbed](#) PCB which is available on GitHub as open source hardware.

Since the LPDDR4 Test Board offers a limited number of IO pins used for interfacing with the testbed, only one channel of a single DDR5 memory IC located on the Test Bed is accessible for testing.

⚠️ Warning

The DDR5 Test Bed has a form factor that is mechanically compatible with SO-DIMM DDR4 sockets. The pinout of the DDR5 Test Bed does not match SO-DIMM DDR4 specification so it cannot be used in systems supporting off-the-shelf SO-DIMM DDR4 memories.

CHAPTER TWELVE

LPDDR5 TEST BED



Fig. 12.1: LPDDR5 Test Bed

This accessory allows interfacing with a single LPDDR5 ICs using the *SO-DIMM DDR5 Tester*. The hardware design of the LPDDR5 Test Bed is released to GitHub (<https://github.com/antmicro/lpddr5-testbed>) as open source hardware. The hardware design of the LPDDR5 Test Bed includes a single Micron MT62F1G32D4DR-031 LPDDR5 IC.

 **Warning**

The DDR5 Test Bed has a form factor that is mechanically compatible with SO-DIMM DDR5 sockets. The pinout of the DDR5 Test Bed does not match SO-DIMM DDR5 specification so it cannot be used in systems supporting off-the-shelf SO-DIMM DDR5 memories.

CHAPTER
THIRTEEN

PERFORMING ATTACKS (HAMMERING)

Rowhammer attacks can be run against a DRAM module. The results can be then used for measuring cell retention. For the complete list of script modifiers, see `--help`.

There are two versions of the rowhammer script:

- `rowhammer.py` - uses regular memory access via EtherBone to fill/check the memory (slower).
- `hw_rowhammer.py` - BIST blocks will be used to fill/check the memory (much faster, but with some limitations regarding fill pattern).

BIST blocks are faster and are the intended way of running Rowhammer tester.

Hammering of a row is done by reading it. There are two ways to specify a number of reads:

- `--read_count N` - one pass of N reads
- `--read_count_range K M N` - multiple passes of reads, as generated by `range(K, M, N)`

Regardless of which one is used, the number of reads in one pass is divided equally between the hammered rows. If a user specifies `--read_count 1000`, then each row will be hammered 500 times.

By default, hammering is performed via DMA, but there is an alternative way with `--payload-executor` which bypasses the DMA and talks directly with the PHY. That allows the user to issue specific activation, refresh and precharge commands.

13.1 Attack modes

Several attack and row selection modes are available, but only one mode can be specified at a time.

- `--hammer-only`

Hammers rows without error checks or reports. When run with `rowhammer.py`, the attack is limited to one row pair. `hw_rowhammer.py` can attack up to 32 rows. With `--payload-executor` enabled, the row limit is dictated by the payload memory size.

For example, the following command will hammer rows 4 and 6 1000 times total (so 500 times each):

```
(venv) $ python hw_rowhammer.py --hammer-only 4 6 --read_count 1000
```

- `--all-rows`

Row pairs generated from the range(start-row, nrows - row-pair-distance, row-jump) expression will be hammered.

The generated pairs come in the format of (i, i + row-pair-distance). [Table 13.1](#) shows default values for arguments:

Table 13.1: Default values for arguments

argument	default
--start-row	0
--row-jump	1
--row-pair-distance	2

For instance, to hammer rows (0, 2), (1, 3), (2, 4), run the following command:

```
(venv) $ python hw_rowhammer.py --all-rows --nrows 5
```

And to hammer rows (10, 13), (12, 15), run:

```
(venv) $ python hw_rowhammer.py --all-rows --start-row 10 --nrows 16 --row-jump 2
      ↪--row-distance 3
```

Setting --row-pair-distance to 0 lets you check how hammering a single row affects other rows. Normally, activations and deactivations are achieved with row reads using the DMA, but in this case this is not possible. Since a single row is being read all the time, no deactivation command would be sent by the DMA. In this case, the --payload-executor argument is required as it bypasses the DMA and sends deactivation commands on its own:

```
(venv) $ python hw_rowhammer.py --all-rows --nrows 5 --row-pair-distance 0 --
      ↪payload-executor
```

- --row-pairs sequential

Hammers pairs of (start-row, start-row + n), where n is a value from 0 to nrows, e.g.:

```
(venv) $ python hw_rowhammer.py --row-pairs sequential --start-row 4 --nrows_
      ↪10
```

The command above will hammer the following set of row pairs:

```
(4, 4 + 0)
(4, 4 + 1)
...
(4, 4 + 9)
(4, 4 + 10)
```

- --row-pairs const

Two rows specified with the const-rows-pair parameter will be hammered:

```
(venv) $ python hw_rowhammer.py --row-pairs const --const-rows-pair 4 6
```

- --row-pairs random

nrows pairs of random rows will be hammered. Row numbers will be between start-row and start-row + nrows.

```
(venv) $ python hw_rowhammer.py --row-pairs random --start-row 4 --nrows 10
```

13.2 Patterns

You can choose a pattern that memory will be initially filled with:

- all_0 - all bits set to 0
- all_1 - all bits set to 1
- 01_in_row - alternating 0s and 1s in a row (0aaaaaaaaa in hex)
- 01_per_row - all 0s in odd-numbered rows, all 1s in even rows
- rand_per_row - random values for all rows

13.3 Example output

```
(venv) $ python hw_rowhammer.py --nrows 512 --read_count 10e6 --pattern 01_in_row
--row-pairs const --const-rows-pair 54 133 --no-refresh
Preparing ...
WARNING: only single word patterns supported, using: 0aaaaaaaaa
Filling memory with data ...
Progress: [=====] 16777216 / 16777216
Verifying written memory ...
Progress: [=====] 16777216 / 16777216 (Errors: 0)
OK
Disabling refresh ...
Running Rowhammer attacks ...
read_count: 10000000
Iter 0 / 1 Rows = (54, 133), Count = 10.00M / 10.00M
Reenabling refresh ...
Verifying attacked memory ...
Progress: [=====] 16777216 / 16777216 (Errors: 30)
Bit-flips for row 53: 5
Bit-flips for row 55: 11
Bit-flips for row 132: 12
Bit-flips for row 134: 3
```

13.4 Row selection examples

Warning

Attacks are performed on a single bank - bank 0 by default. To change the bank that is being attacked use the --bank flag.

- Select row pairs from row 3 (--start-row) to row 59 (--nrows) where the next pair is 5 rows over (--row-jump) from the previous one:

```
(venv) $ python hw_rowhammer.py --pattern 01_in_row --all-rows --start-row 3
      ↵--nrows 60 --row-jump 5 --no-refresh --read_count 10e4
```

- Select row pairs from row 3 to row 59 without a distance between subsequent pairs (no --row-jump), which means that rows pairs are incremented by 1:

```
(venv) $ python hw_rowhammer.py --pattern 01_in_row --all-rows --start-row 3
      ↵--nrows 60 --no-refresh --read_count 10e4
```

- Select all row pairs (from 0 to nrows - 1):

```
(venv) $ python hw_rowhammer.py --pattern 01_in_row --all-rows --nrows 512 --
      ↵no-refresh --read_count 10e4
```

- Select all row pairs (from 0 to nrows - 1) and save the error summary in JSON format to the test directory:

```
(venv) $ python hw_rowhammer.py --pattern 01_in_row --all-rows --nrows 512 --
      ↵no-refresh --read_count 10e4 --log-dir ./test
```

- Select a single row (42 in this case) and save the error summary in JSON format to the test directory:

```
(venv) $ python hw_rowhammer.py --pattern all_1 --row-pairs const --const-
      ↵rows-pair 42 42 --no-refresh --read_count 10e4 --log-dir ./test
```

- Select all rows (from 0 to nrows - 1) and hammer them one by one 1M times each.

```
(venv) $ python hw_rowhammer.py --all-rows --nrows 100 --row-pair-distance 0
      ↵--payload-executor --no-refresh --read_count 1e6
```

Note

Since for a single ended attack row activation needs to be triggered the --payload-executor switch is required. The size of the payload memory is set by default to 1024 bytes and can be changed using the --payload-size switch.

13.5 Cell retention measurement

The following parameter is used to conduct DRAM cell retention experiments.

- --no-attack-time <time>

Instead of performing a rowhammer attack, the script will load the RAM with selected pattern and sleep for time nanoseconds. After this time, it will check for any bitflips that could have happened. This option does not imply --no-refresh.

```
(venv) $ python hw_rowhammer.py --no-attack-time 10e9 --no-refresh
```

By executing this command with varying sleep intervals, it is possible to characterize the frequency and occurrence of bit flips over time. Fig. 13.1 presents the relationship between the detected bit flips and the elapsed time without a memory refresh. The results were collected from SK hynix HMCG84MEBRA112NBB off-the-shelf RDIMM DDR5 module.

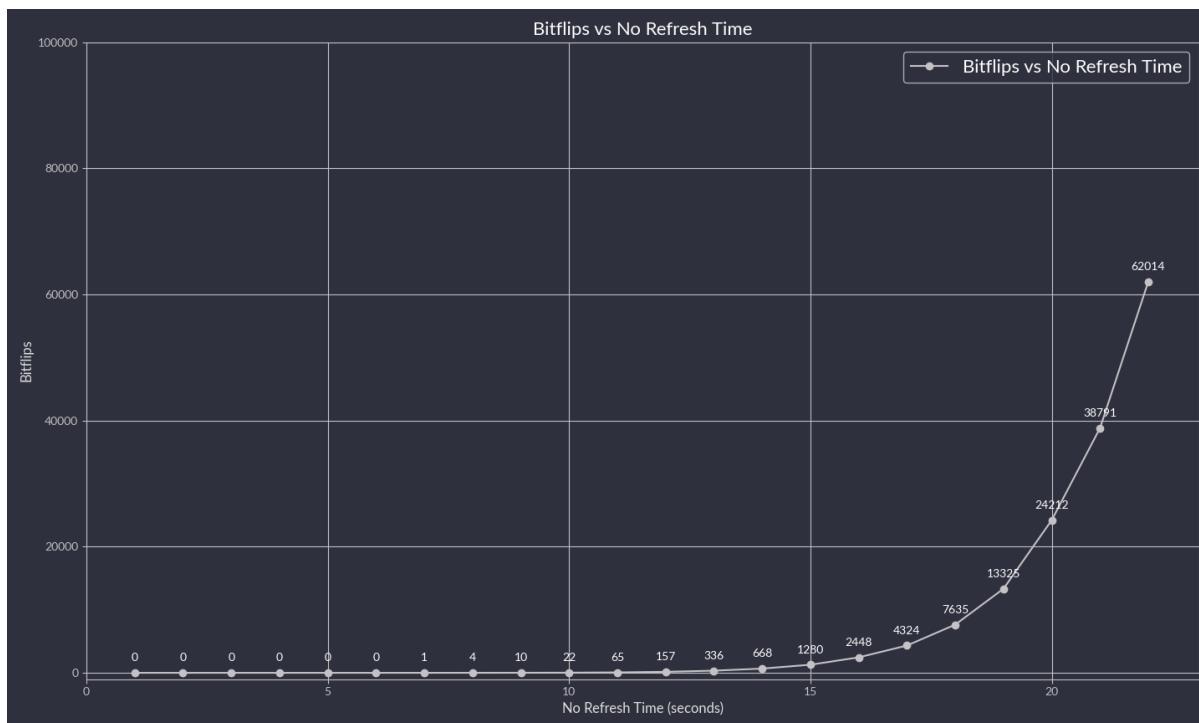


Fig. 13.1: Sample plot summarizing DRAM cell retention testing

13.6 Read count parameter optimization

The following set of commands can be used to analyze the relationship between the number of bit flips and the number of reads. Additionally, they help identify the optimal number of reads required to trigger a bit flip.

- Select all row pairs (from 0 to nrows - 1) and perform a set of tests for different read count values, starting from 10e4 and ending at 10e5 with a step of 20e4 (`--read_count_range [start stop step]`):

```
(venv) $ python hw_rowhammer.py --pattern 01_in_row --all-rows --nrows 512 --
    ↵no-refresh --read_count_range 10e4 10e5 20e4
```

- Perform a set of tests for different read count values in a given range for one row pair (50, 100):

```
(venv) $ python hw_rowhammer.py --pattern 01_in_row --row-pairs const --
    ↵const-rows-pair 50 100 --no-refresh --read_count_range 10e4 10e5 20e4
```

- Perform a set of tests for different read count values in a given range for one row pair (50, 100) and stop the test execution as soon as a bitflip is found:

```
(venv) $ python hw_rowhammer.py --pattern 01_in_row --row-pairs const --
→const-rows-pair 50 100 --no-refresh --read_count_range 10e4 10e5 20e4 --
→exit-on-bit-flip
```

- Perform a set of tests for different read count values in a given range for one row pair (50, 100) and save the error summary in JSON format to the test directory:

```
(venv) $ python hw_rowhammer.py --pattern 01_in_row --row-pairs const --
→const-rows-pair 50 100 --no-refresh --read_count_range 10e4 10e5 20e4 --
→log-dir ./test
```

- Perform a set of tests for different read count values in a given range for a sequence of attacks for different pairs, where the first row of a pair is 40 and the second one is a row of a number from range (40, nrows - 1):

```
(venv) $ python hw_rowhammer.py --pattern 01_in_row --row-pairs sequential --
→start-row 40 --nrows 512 --no-refresh --read_count_range 10e4 10e5 20e4
```

13.7 DRAM modules

When building one of the targets available in [rowhammer_tester/targets](#), you can specify a custom DRAM module using the `--module` argument. To find the default modules for each target, check the output of `--help`.

Note

Specifying different DRAM module makes most sense on boards that allow to easily replace the DRAM module, such as on ZCU104. On other boards it would be necessary to desolder the DRAM chip and solder a new one.

13.7.1 Adding new modules

To make development more convenient, modules can be added in the rowhammer-tester repository directly in file [rowhammer_tester/targets/modules.py](#). These definitions will be used before definitions in LiteDRAM.

To add a new module definition, use the existing ones as a reference. A new module class should derive from `SDRAMModule` (or the helper classes, e.g. `DDR4Module`). Timing/geometry values for a module have to be obtained from the relevant DRAM module's datasheet. The timings in classes deriving from `SDRAMModule` are specified in nanoseconds. The timing value can also be specified as a 2-element tuple `(ck, ns)`, in which case `ck` is the number of clock cycles and `ns` is the number of nanoseconds (and can be `None`). The highest of the resulting timing values will be used.

13.7.2 SPD EEPROM

For boards that use DIMM/SO-DIMM modules (e.g. ZCU104), it is possible to read the contents of DRAM module [SPD EEPROM memory](#). SPD contains several essential module parameters that the memory controller needs in order to use the DRAM module. SPD EEPROM can be read over an I²C bus.

Reading SPD EEPROM

To read the SPD memory, use the `rowhammer_tester/scripts/spd_eeprom.py` script. First, prepare the environment as described in the [Building Rowhammer designs](#) chapter. Then, use the following command to read the contents of SPD EEPROM and save it to a file, for example:

```
python rowhammer_tester/scripts/spd_eeprom.py read MTA4ATF51264HZ-3G2J1.bin
```

The contents of the file can then be used to get DRAM module parameters. Use the following command to examine the parameters:

```
python rowhammer_tester/scripts/spd_eeprom.py show MTA4ATF51264HZ-3G2J1.bin 125e6
```

Note that system clock frequency must be passed as an argument to determine timing values in controller clock cycles.

Using SPD data

The memory controller is able to set the timings read from an SPD EEPROM during system boot. The only requirement here is that the SoC is built with an I2C controller and the I2C pins are routed to the (R)DIMM module. There is no additional action required and the timings will be set automatically.

13.8 Utilities

Some scripts are simple and do not take command line arguments, others will provide help via `<script_name>.py --help` or `<script_name>.py -h`.

Few of the scripts accept a `--srv` option. With this option enabled, a program will start its own instance of `liteX_server` (the user doesn't need to run `make srv` to control the board).

13.8.1 Run LEDs demo - `leds.py`

Displays a simple “bouncing” animation using the LEDs on Arty-A7 board, with the light moving from side to side.

`-t TIME_MS` or `--time-ms TIME_MS` option can be used to adjust LED switching interval.

13.8.2 Check version - `version.py`

Prints the data stored in the LiteX identification memory:

- hardware platform identifier
- source code git hash
- build date

Example output:

```
(venv) python version.py
Rowhammer tester SoC on xc7k160tffg676-1, git:_
→e7854fdd16d5f958e616bbb4976a97962ee9197d 2022-07-24 15:46:52
```

13.8.3 Check CSRs - dump_regs.py

Dumps values of all CSRs. Example output of dump_regs.py:

```
0x82000000: 0x00000000 ctrl_reset
0x82000004: 0x12345678 ctrl_scratch
0x82000008: 0x00000000 ctrl_bus_errors
0x82002000: 0x00000000 uart_rxtx
0x82002004: 0x00000001 uart_txfull
0x82002008: 0x00000001 uart_rxempty
0x8200200c: 0x00000003 uart_ev_status
0x82002010: 0x00000000 uart_ev_pending
...
...
```

 Note

Note that the `ctrl_scratch` value is `0x12345678`. This is the reset value of this register. If you are getting a different value, it may indicate a problem.

13.8.4 Initialize memory - mem.py

Before the DRAM memory can be used, perform initialization and leveling using the `mem.py` script.

Expected output:

```
(venv) $ python mem.py
(LiteX output)
=====
Initialization =====
Initializing SDRAM @0x40000000...
Switching SDRAM to software control.
Read leveling:
m0, b0: |11111111111100000000000000000000| delays: 06+-06
m0, b1: |000000000000001111111111111000| delays: 21+-08
m0, b2: |00000000000000000000000000000011| delays: 31+-01
m0, b3: |00000000000000000000000000000000| delays: -
m0, b4: |00000000000000000000000000000000| delays: -
m0, b5: |00000000000000000000000000000000| delays: -
m0, b6: |00000000000000000000000000000000| delays: -
m0, b7: |00000000000000000000000000000000| delays: -
best: m0, b01 delays: 21+-07
m1, b0: |11111111111100000000000000000000| delays: 07+-07
m1, b1: |000000000000001111111111111000| delays: 22+-08
m1, b2: |00000000000000000000000000000001| delays: 31+-00
m1, b3: |00000000000000000000000000000000| delays: -
m1, b4: |00000000000000000000000000000000| delays: -
m1, b5: |00000000000000000000000000000000| delays: -
m1, b6: |00000000000000000000000000000000| delays: -
m1, b7: |00000000000000000000000000000000| delays: -
best: m1, b01 delays: 22+-08
Switching SDRAM to hardware control.
```

(continues on next page)

(continued from previous page)

```
Memtest at 0x40000000 (2MiB)...
  Write: 0x40000000-0x40200000 2MiB
  Read: 0x40000000-0x40200000 2MiB
Memtest OK
Memspeed at 0x40000000 (2MiB)...
  Write speed: 12MiB/s
  === Initialization succeeded. ===
Proceeding ...

Memtest (basic)
OK

Memtest (random)
OK
```

13.8.5 Enter BIOS - bios_console.py

It may happen that memory initialization fails when running the `mem.py` script. This is most likely due to using boards that allow to swap memory modules, such as the ZCU104.

The memory initialization procedure is performed by the CPU instantiated inside the FPGA fabric. The CPU runs the LiteX BIOS. In case of memory training failure, it may be helpful to access the LiteX BIOS console.

If the script cannot find a serial terminal emulator program on the host system, it will fall back to `litex_term` which ships with LiteX. It is however advised to install `picocom/minicom` as `litex_term` has worse performance.

In the BIOS console, use the `help` command to get information about other available commands. To re-run memory initialization and training, type `reboot`.

Note

To close `picocom/minicom`, use the `CTRL+A+X` key combination.

Example:

```
(venv) $ python bios_console.py
LiteX Crossover UART created: /dev/pts/4
Using serial backend: auto
picocom v3.1

port is      : /dev/pts/4
flowcontrol : none
baudrate is  : 1000000
parity is    : none
databits are : 8
stopbits are : 1
escape is    : C-a
local echo is: no
```

(continues on next page)

(continued from previous page)

```

noinit is      : no
noreset is     : no
hangup is      : no
nolock is      : no
send_cmd is    : sz -vv
receive_cmd is : rz -vv -E
imap is        :
omap is        :
emap is        : crcrlf,delbs,
logfile is     : none
initstring     : none
exit_after is  : not set
exit is        : no

Type [C-a] [C-h] to see available commands
Terminal ready
ad speed: 9MiB/s

===== Boot =====
Booting from serial...
Press Q or ESC to abort boot completely.
sL5DdSMmkekro
      Timeout
No boot medium found

===== Console =====

litex>

```

Perform memory tests from the BIOS

After entering the BIOS, you may want to perform a memory test using utilities built into the BIOS. There are several ways to do it:

- `mem_test` - performs a series of writes and reads to check if values read back are the same as those previously written. It is limited by a 32-bit address bus, so only 4 GiB of address space can be tested. You can get the origin of the RAM space using `mem_list` command.
- `sdram_test` - essentially `mem_test` but predefined for the first 1/32 of the defined RAM region size.
- `sdram_hw_test` - similar to `mem_test`, but accesses the SDRAM directly using DMAs, so it is not limited to 4 GiB.

It requires passing 2 arguments (`origin` and `size`) with a 3rd optional argument, `burst_length`. When using `sdram_hw_test` you don't have to offset the `origin` like in the case of `mem_test`. `size` is a number of bytes to test and `burst_length` is a number of full transfer writes to the SDRAM before reading and checking the written content. The default value for `burst_length` is 1, which means that after every write, a check is performed. Generally, higher `burst_length` values mean faster operation.

13.8.6 Test with BIST - `mem_bist.py`

A script written to test the BIST block functionality. Two tests are available:

- `test-modules` - the memory is initialized and then a series of errors is introduced. Then BIST is used to check the contents of the memory. If the number of errors detected is equal to the number of errors introduced, the test is passed.
- `test-memory` - a simple test that writes a pattern in the memory, reads it, and checks whether the content is correct. Both write and read operations are done via BIST.

13.8.7 Run benchmarks - `benchmark.py`

Benchmarks memory access performance. There are two subcommands available:

- `etherbone` - measures performance of the EtherBone bridge.
- `bist` - measures performance of DMA DRAM access using the BIST modules.

Example output:

```
(venv) $ python benchmark.py etherbone read 0x10000 --burst 255
Using generated target files in: build/lpddr4_test_board
Running measurement ...
Elapsed = 4.189 sec
Size    = 256.000 KiB
Speed   = 61.114 KiBps

(venv) $ python benchmark.py bist read
Using generated target files in: build/lpddr4_test_board
Filling memory before reading measurements ...
Progress: [=====] 16777216 / 16777216
Running measurement ...
Progress: [=====] 16777216 / 16777216 (Errors: 0)
Elapsed = 1.591 sec
Size    = 512.000 MiB
Speed   = 321.797 MiBps
```

13.8.8 Use logic analyzer - `analyzer.py`

This script utilizes the Litescope functionality to gather debug information about signals in the LiteX system. In-depth Litescope documentation is available on [GitHub](#).

Litescope analyzer needs to be instantiated in your design. A sample design with the analyzer added is provided as the `arty_litescope` TARGET and can be run using the Arty-A7 board. You can use `rowhammer_tester/targets/arty_litescope.py` as a reference for your own Litescope-enabled targets.

To build the `arty_litescope` sample and upload it to device, follow the instructions below:

1. In the root directory, run:

```
export TARGET=arty_litescope
make build
make upload
```

the analyzer.csv file will be created in the root directory.

2. Copy it to the target's build directory before using analyzer.py.

```
cp analyzer.csv build/arty_litescope/
```

3. Start litex-server with:

```
make srv
```

4. Execute the analyzer script in a separate shell:

```
export TARGET=arty_litescope
python rowhammer_tester/scripts/analyzer.py
```

Results will be stored in the dump.vcd file and can be viewed using gtkwave:

```
gtkwave dump.vcd
```

CHAPTER FOURTEEN

RESULT VISUALIZATION

After executing attacks on your board, you can use the results to draw a plot or visualize them with Python and matplotlib or with the [F4PGA Database Visualizer](#). This chapter describes scripts used for visualizing the rowhammer attacks. The script accept JSON files that can be generated as a result of hammering with commands described in the [Performing attacks \(hammering\)](#) chapter.

14.1 Plot bitflips - logs2plot.py

This script can plot graphs based on generated logs. It can generate two different types of graphs:

1. Distribution of bitflips across rows and columns. For example, you can generate graphs by calling:

```
(venv) $ python logs2plot.py your_error_summary.json
```

One graph will be generated for every attack. So if you attacked two row pairs (A, B), (C, D) with two different read counts each (X, Y), for a total of 4 attacks, you will get 4 plots:

- read count: X and pair: (A, B)
- read count: X and pair: (C, D)
- read count: Y and pair: (A, B)
- read count: Y and pair: (C, D)

You can control the number of displayed columns with `--plot-columns`. For example if your module has 1024 columns and you provide `--plot-columns 16`, then the DRAM columns will be displayed in groups of 64.

2. Distribution of rows affected by bitflips when targeting single rows. For example, you can generate a graph by calling:

```
(venv) $ python logs2plot.py --aggressors-vs-victims your_error_summary.  
→json
```

A graph will be generated with victims (affected rows) on the Y axis and aggressors (attacking rows) on the X axis. The row under attack is marked with the X symbol. The affected rows (victims) above and below the row under attack (aggressor) include cells along with the number of bitflips detected. The colors of the tiles indicate how many bitflips occurred for each victim.

You can enable additional annotation with `--annotate bitflips` so that the number of occurred bitflips will be explicitly labeled on top of each victim tile. Fig. 14.1 presents a set of bitflips recorded while hammering rows 1-50 of bank 0 in SK hynix HMCG84MEBRA112NBB from off-the-shelf RDIMM DDR5 memory under test.

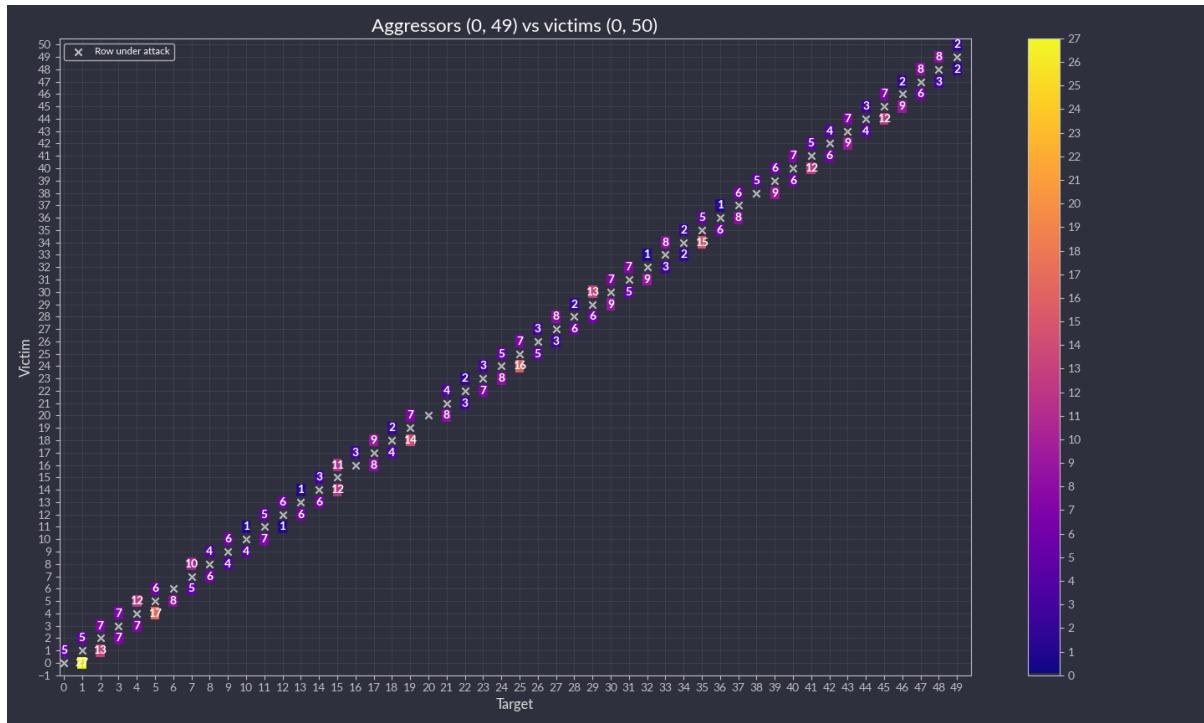




Fig. 14.2: Zooming in on the plot

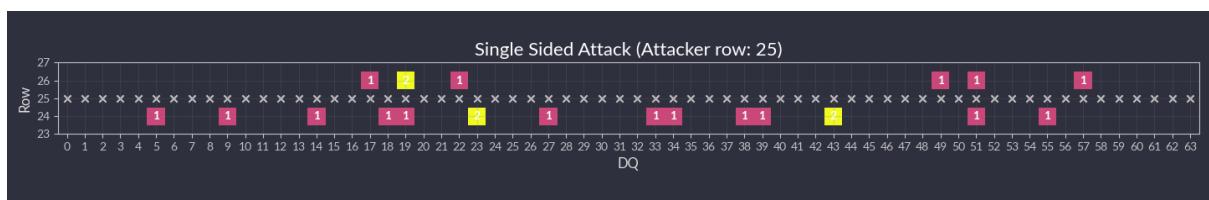


Fig. 14.3: Per pad statistics

14.3 Use F4PGA Visualizer - logs2vis.py

Similarly as in logs2plot.py, you can generate visualizations using the [F4PGA Database Visualizer](#).

To view results using the visualizer you need to:

1. Clone and build the visualizer:

```
git clone https://github.com/chipsalliance/f4pga-database-visualizer
cd f4pga-database-visualizer
npm run build
```

2. Run rowhammer.py or hw_rowhammer.py with --log-dir log_directory.

3. Generate JSON files for the visualizer:

```
python3 logs2vis.py log_directory/your_error_summary.json vis_directory
```

4. Copy generated JSON files from vis_directory to /path/to/f4pga-database-visualizer/dist/production/.

5. Start a simple HTTP server inside the production directory:

```
python -m http.server 8080
```

An example output generated with the --aggressors-vs-victims flag looks like so:

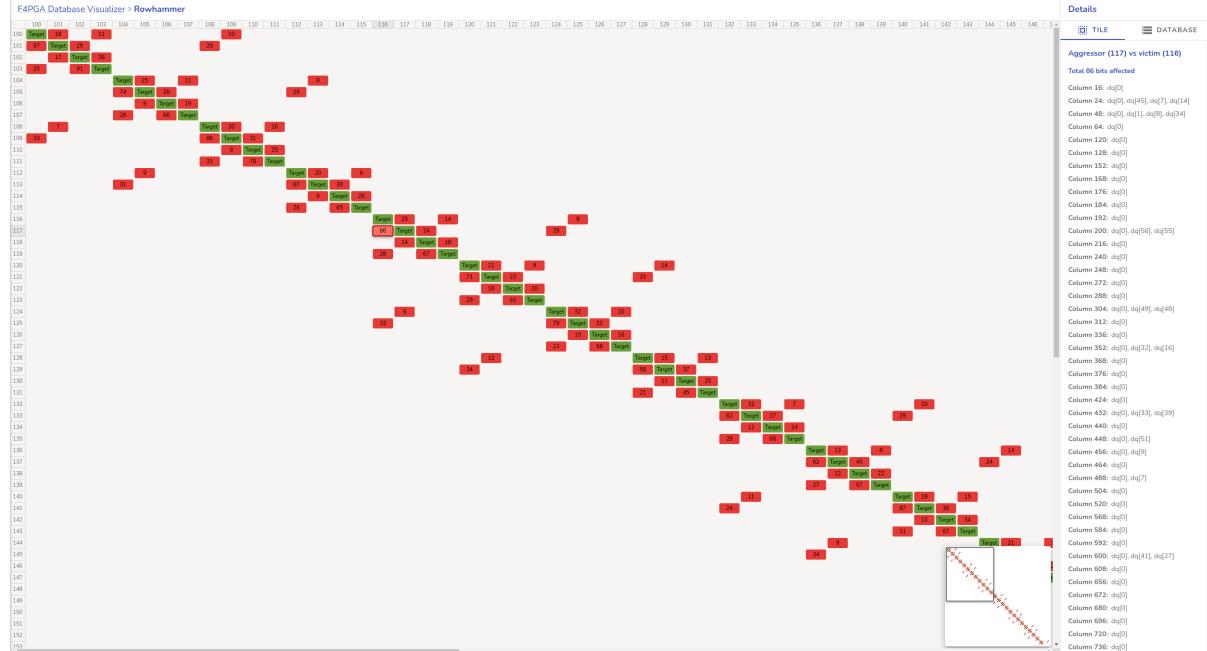


Fig. 14.4: Aggressors vs. victims flag output example

TEST-WRITING PLAYBOOK

The [Playbook directory](#) contains a group of Python classes and scripts designed to simplify the process of writing various rowhammer-related tests. These tests can be executed against a hardware platform.

15.1 Payload

Tests are generated as payload data. After generation, this data is transferred to a memory area in the device reserved for this purpose called payload memory. The payload contains an instruction list that can be interpreted by the payload executor module on hardware. The payload executor translates these instructions into DRAM commands. The payload executor connects directly to the DRAM PHY, bypassing the DRAM controller, as explained in the [Architecture section](#).

15.1.1 Changing payload memory size

Payload memory size can be changed up to the limit of memory available on the hardware platform used. The payload memory size is defined in [common.py](#), as an argument to LiteX:

```
add_argument("--payload-size", default="1024", help="Payload memory size in bytes  
→")
```

The examples shown in this chapter don't require any changes. When writing your own [configurations](#), you may need to change the default value.

15.2 Row mapping

In the case of DRAM modules, the physical layout of the memory rows in hardware can be different from the logical numbers assigned to them. The nature of a rowhammer attack is such that only the physically adjacent rows are affected by the aggressor row. There are several mapping strategies that can be implemented to deal with the problem of disparity between physical location and logical enumeration:

- TrivialRowMapping - logical address is the same as physical address
- TypeARowMapping - a more complex mapping method, reverse-engineered as a part of the [Defeating Software Mitigations against Rowhammer: a Surgical Precision Hammer paper](#)
- TypeBRowMapping - logical address is the physical one multiplied by 2, taken from [Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques](#)

15.3 Row Generator class

The Row Generator class is responsible for creating a list of row numbers used in a rowhammer attack. Currently, one instance of this class is available.

15.3.1 EvenRowGenerator

Generates a list of even numbered rows. Uses the row mapping specified by the *payload generator class* used. Two configuration parameters are needed for EvenRowGenerator:

- `nr_rows` - number of rows to be generated
- `max_row` - maximal number to be used. The chosen numbers will be *modulo max_row*

15.3.2 HalfDoubleRowGenerator

Generates a list of rows for a **Half-Double** attack. The list will repeat rows to create weight difference between attacks at different distances. Used by `HalfDoubleAnalysisPayloadGenerator`.

- `nr_rows` - number of rows in the attack pattern
- `distance_one` - indicates whether the attack has a distance one component
- `double_sided` - is the attack double-sided?
- `distance_two` - indicates whether the attack has a distance two component
- `attack_rows_start` - the index of the first attack row
- `max_attack_row_idx` - the position of the last attack row relative to `attack_rows_start`
- `decoy_rows_start` - the index of the first decoy row; there are 3 decoy rows that are used as placebos in various situations: to hammer away from the victim but to keep the number of hammers and timing constant

15.4 Payload generator class

The purpose of the payload generator is to prepare a payload and process the test outcome. It is a class that can be reused in different tests *configurations*. Payload generators are located in the `payload_generators` directory.

15.4.1 Available payload generators

Row mapping and row generator settings are combined into a payload generator class. Currently, two payload generators are available.

RowListPayloadGenerator

RowListPayloadGenerator is a simple payload generator that can use a RowGenerator class to generate rows, and then generate a payload that hammers the rows in this list. (hammering is a term used to describe multiple read operations on the same row). RowListPayloadGenerator can also issue refresh commands to the DRAM module. The configs that can be used in `payload_generator_config` for this payload generator are listed below:

- `row_mapping` - the *row mapping* used
- `row_generator` - the *row generator class* used to generate the rows

- `row_generator_config` - parameters for the row generator
- `verbose` - should verbose output be generated (true or false)
- `fill_local` - when enabled, permits shrinking the filled memory area to aggressor and victim rows
- `read_count` - number of hammers (reads) per row
- `refresh` - should refresh be enabled (true or false)

Examples of `configurations` for this test are provided as `configs/example_row_list_*.json` files. Some of them require a significant amount of memory declared as payload memory. To execute a minimalistic example from within the `rowhammer-tester` repo, execute:

```
source venv/bin/activate
export TARGET=arty # change accordingly
cd rowhammer_tester/scripts/playbook/
python playbook.py configs/example_row_list_minimal.json
```

Expected output:

```
Progress: [=====] 65536 / 65536
Row sequence:
[0, 2, 4, 6, 14, 12, 10, 8, 16, 18]
Generating payload:
  tRAS = 5
  tRP = 3
  tREFI = 782
  tRFC = 32
  Repeatable unit: 930
  Repetitions: 93
  Payload size = 0.10KB / 1.00KB
  Payload per-row toggle count = 0.010K x10 rows
  Payload refreshes (if enabled) = 10 (disabled)
  Expected execution time = 1903 cycles = 0.019 ms

Transferring the payload ...

Executing ...
Time taken: 0.738 ms

Progress: [==                               ] 3338 / 65536 (Errors: 1287)
...
```

HammerTolerancePayloadGenerator

`HammerTolerancePayloadGenerator` is a payload generator for measuring and characterizing rowhammer tolerance. It can provide information about how many rows and bits are susceptible to the rowhammer attack. It can also provide information about the location of susceptible bits.

A series of double-sided hammers against the available group of victim rows is performed. The double-sided hammers increase in intensity based on `read_count_step` parameter. Here are the parameters that can be specified in `payload_generator_config` for this payload generator:

- `row_mapping` - this is the *row mapping* used
- `row_generator` - this is the *row generator class* used to generate the rows
- `row_generator_config` - parameters for the row generator
- `verbose` - should verbose output be generated (true or false)
- `fill_local` - when enabled, permits shrinking the filled memory area to just the aggressors and the victim
- `nr_rows` - number of rows to conduct the experiment over. This is the number of aggressor rows
 - The number of victim rows will be lower by 2; for example, to perform hammering for 32 victim rows, use 34 as the parameter value
- `read_count_step` - this is how much to increment the hammer count between multiple tests for the same row
 - This is the number of hammers on a single side (total number of hammers on both sides is 2x this value)
- `initial_read_count` - hammer count for the first test for a given row. Defaults to `read_count_step` if unspecified
- `distance` - distance between aggressors and victim. Defaults to 1
- `baseline` - when enabled, a retention effect baseline is collected by hammering distant rows for the same amount of time as the aggressor rows
- `first_dummy_row` - location of the first of two dummy rows used for baselining
- `iters_per_row` - number of times the hammer count is incremented for each row

The results are a series of histograms with appropriate labeling.

Example *configurations* for this test are provided as `configs/example_hammer_*.json` files. Some of them require a significant amount of memory declared as payload memory. To execute a minimalistic example from within the `rowhammer-tester` repo, execute:

```
source venv/bin/activate
export TARGET=arty # change accordingly
cd rowhammer-tester/scripts/playbook/
python playbook.py configs/example_hammer_minimal.json
```

Expected output:

```
Progress: [=====] 3072 / 3072
Generating payload:
  tRAS = 5
  tRP = 3
  tREFI = 782
  tRFC = 32
Repeatable unit: 186
Repetitions: 93
Payload size = 0.04KB / 1.00KB
Payload per-row toggle count = 0.010K x2 rows
```

(continues on next page)

(continued from previous page)

```
Payload refreshes (if enabled) = 10 (disabled)
Expected execution time = 1263 cycles = 0.013 ms
```

Transferring the payload ...

Executing ...

Time taken: 0.647 ms

Progress: [=====] 323 / 1024 (Errors: 320)

...

HalfDoubleAnalysisPayloadGenerator

Half-Double is a Rowhammer phenomenon where accesses to both distance-one and distance-two neighbors of a victim row are used to generate bit flips. This payload generator allows us to characterize the Half-Double effect on a memory part.

For each candidate victim row, the analysis starts out with the maximum number of hammers and minimum dilution level. Then, we proceed as follows:

1. Dilution is increased until pure distance-one attacks stop working.
2. Verify that pure distance-two attack doesn't work.
3. Increase dilution level and record the number of bit flips in the victim until either the bit flips stop or maximum dilution level is reached.
4. Once maximum dilution level is reached or bit flips stop, reduce hammer count by a step and reset dilution to initial level and retry step 3. Repeat until the lowest hammer count is reached.

Note

The hammer count changes on a linear scale and dilution changes on an exponential scale.

Results are presented as a table of values with columns representing the hammer count and the rows representing dilution levels. See Tables 2 and 3 in the [Half-Double white paper](#) as examples.

- `max_total_read_count` - maximum number of hammers issued to any given row during an iteration
- `read_count_steps` - the amount to decrement the number of hammers for each iteration of the outer loop
- `initial_dilution` - initial value for dilution. Dilution resets to this value at the beginning of the inner loop
- `dilution_multiplier` - dilution is multiplied by this value for each iteration of the inner loop
- `verbose` - generates more output
- `row_mapping` - specifies the style in which the rows are mapped on the chip

- attack_rows_start - starting row number for rows used to attack the victim
- max_attack_row_idx - index measured from attack_rows_start for the last attack row
- decoy_rows_start - the position of the first decoy row. There are three decoy rows. They are used as placebos during pure distance-one portions of the experiment to make the number of hammers and their timing comparable
- max_dilution - maximum value for dilution
- fill_local - only reinitialize affected rows between experiments, as an optimization

15.5 Configurations

Test configuration files are represented as JSON files. An example:

```
{
    "inversion_divisor" : 2,
    "inversion_mask" : "0b10",
    "payload_generator" : "RowListPayloadGenerator",
    "payload_generator_config" : {
        "row_mapping" : "TypeARowMapping",
        "row_generator" : "EvenRowGenerator",
        "read_count" : 27,
        "max_iteration" : 10,
        "verbose" : true,
        "refresh" : false,
        "fill_local" : true,
        "row_generator_config" : {
            "nr_rows" : 10,
            "max_row" : 64
        }
    }
}
```

The following parameters are supported:

- payload_generator - name of the *payload generator class* to use
- row_pattern - pattern that will be stored in rows
- inversion_divisor and inversion_mask - controls which rows get the inverted pattern described in the *inversion section*
- payload_generator_config - these parameters are specific for the *payload generator class* used

15.5.1 Inversion

If needed, use the bitwise-inverted data pattern for selected tested rows. Two parameters are used to specify which rows are to be inverted:

- inversion_divisor
- inversion_mask

Example: `inversion_divisor = 8, inversion_mask = 0b10010010` (bits 1, 4 and 7 are “on”). We iterate through all row numbers $0,1,2,3,4,\dots,8,9,10,\dots$. First, a modulo `inversion_divisor` operation is performed on a row number. In this case, it’s $\text{mod } 8$. Next, we check if the bit in `inversion_mask` in the position corresponding to our row number (after modulo) is “on” or “off”. If it’s “on”, this whole row will be inverted. The results for our example are visible in [Table 15.1](#) below.

Table 15.1: Inversion example table

Row num- ber	Row num- ber	Value
mod- ulo divisor (8)		
0	0	pattern
1	1	inverted pattern
2	2	pattern
3	3	pattern
4	4	inverted pattern
5	5	pattern
6	6	pattern
7	7	inverted pattern
8	0	pattern
9	1	inverted pattern
10	2	pattern
11	3	pattern
12	4	inverted pattern

CHAPTER SIXTEEN

MEMORY TESTING

This chapter provides a brief overview of memory validation and testing methodologies used for ensuring that the Rowhammer tester remains usable with various memory targets.

16.1 CI driven testing

The digital *design targets* supported by the Rowhammer tester are periodically synthesized with a Continuous Integration system within Antmicro's internal infrastructure. The synthesized designs (bitstreams) are then uploaded to physical testers orchestrated with CI runners. You can read more about the hardware-in-the-loop testing methodology and Antmicro's Scalerunner project in this [blog note](#).

The hardware platforms currently used in the CI-based testing are:

- *RDIMM DDR5 Tester Rev. 1.0* with off-the-shelf Micron MTC10F1084S1RC48BA1 NGCC memory module installed.
- *SODIMM DDR5 Tester* with off-the-shelf Micron MTC8C1084S1SC48BA1 memory module installed.
- *SODIMM DDR5 Tester* with *LPDD5 Testbed* with Micron MT62F1G32D4DR-031 WT:B memory IC soldered.
- *LPDDR4 Tester* with *DDR5 Testbed* with Micron MT60B2G8HB-48B memory IC soldered.

16.2 Manual testing

In order to increase test coverage with respect to a number of different off-the-shelf memory modules, semi-automated testing is performed.

In this scenario the memory module under test is installed in one of the testers and verified with a Memtest routine. The logs from the testing process are collected in a common storage bucket. Based on the collected logs, a historical test coverage is evaluated.

The unified testing procedure consists of three short memory tests followed by one extended test:

1. Short Test Run each of the following commands:

```
python3 rowhammer_tester/scripts/mem.py --srv --size 0x200000
```

Repeat this command three times to ensure consistency.

2. Extended Test Run the following command:

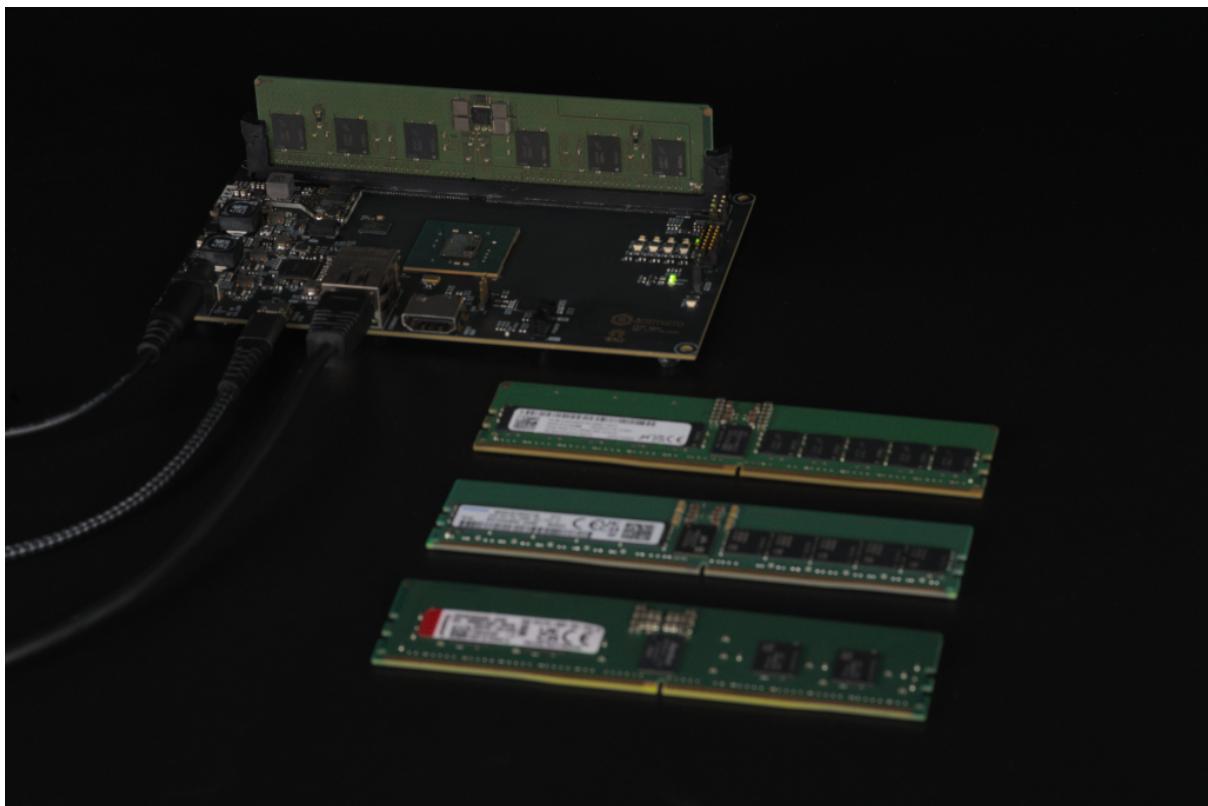


Fig. 16.1: Memory testing setup for manual tests

```
python3 rowhammer_tester/scripts/mem.py --srv --size 0x800000
```

16.3 RDIMM DDR5 test coverage

The following RDIMM DDR5 coverage table outlines the DDR5 RDIMM modules that have passed all of the above tests. The table matches the memory module and the SHA of the latest Rowhammer tester commit that was used during the experimentation.

Table 16.1: RDIMM DDR5 test coverage

	Memory MPN	Manu-facturer	RCD Manu-facturer	Short Memtest	Extended Memtest	Latest SHA
1	HMCG84MEBRA112NB	SK hynix	Rambus	6/6	2/2	fd2d59c
2	HMCG88AGBRA188NN	SK hynix	Montage	6/6	2/2	bd05e52
3	KF548R36RB-32	Kingston	Montage	6/6	2/2	35bfd92
4	KF560R32RB-16	Kingston	REA Renesas	6/6	2/2	35bfd92
5	M321R4GA0BB0-CQKMS	Samsung	Montage	6/6	2/2	bd05e52
6	M321R4GA3BB6-CQKET	Samsung	IDT Renesas	6/6	2/2	bd05e52
7	M321R4GA3BB6-CQKMG	Samsung	Montage	6/6	2/2	bd05e52
8	M321R4GA3PB0-CWMCJ	Samsung	IDT Renesas	6/6	2/2	35bfd92
9	M329R8GA0BB0-CQKVG	Samsung	Rambus	6/6	2/2	35bfd92
10	MTC10F1084S1RC48B_	Micron	Rambus	6/6	2/2	bd05e52
11	MTC10F1084S1RC48B_	Micron	IDT Renesas	6/6	2/2	35bfd92
12	MTC10F1084S1RC48B_	Micron	Rambus	6/6	2/2	bd05e52
13	MTC18F104S1PC48BA_	Micron	IDT Renesas	6/6	2/2	bd05e52
14	MTC20F2085S1RC48B_	Micron	IDT Renesas	6/6	2/2	35bfd92
15	MTC20F2085S1RC48B_	Micron	Rambus	6/6	2/2	bd05e52
16	MTC20F2085S1RC48B_	Micron	IDT Renesas	6/6	2/2	bd05e52
17	MTC10F1084S1RC48B_	Micron	IDT Renesas	6/6	2/2	fd2d59c

CHAPTER
SEVENTEEN

BUILDING LINUX TARGET

The memory controllers synthesized for Rowhammer testing can be utilized as parts of a regular digital design that is capable of booting an operating system. In such scenario the memory controller is used by the operating system for interacting with a DRAM memory. This chapter describes a separate target configuration that has been created in order to synthesize a Linux-capable system that you can run on Antmicro's *RDIMM DDR5 Tester*.

17.1 Base DDR5 Tester Linux Options

The ddr5_tester_linux target is configured via specifying the TARGET_ARGS variable and requires the following arguments:

Option	Documentation
--build	When specified will invoke synthesis and hardware analysis tool (Vivado by default). Will produce programmable bitstream.
--l2-size	Specifies the L2 cache size.
--iodelay-clk-	IODELAY clock frequency.
--module	The DDR5 module to be used.
--with-wishbone	VexRiscV SMP specific option. Disables native LiteDRAM interface.
--wishbone-for	VexRiscV SMP specific option. Forces the wishbone bus to be 32 bits wide.

Additionally, you can set up EtherBone or Ethernet to communicate with the system as described below.

17.1.1 Ethernet Options

Option	Documentation
--with-ethernet	Sets up Ethernet for the DDR5 Tester board.
--remote-ip-address	The IP address of the remote machine connected to DDR5 Tester.
--local-ip-address	Local (DDR5 Tester's) IP address.

17.1.2 Etherbone Options

Option	Documentation
--with-etherbone	Sets up Ethernet for DDR5 Tester board.
--ip-address	IP address to be used for the EtherBone.
--mac-address	MAC address to be used for the EtherBone.

17.2 Building the RDIMM DDR5 Tester Linux Target

After configuring the RDIMM DDR5 Tester Linux, the target can be build with `make build`. Below you can see a use example of a DDR5 Tester Linux Target with Ethernet configured:

```
make build TARGET=ddr5_tester_linux TARGET_ARGS="--build --l2-size 256 --iodelay-  
→clk-freq 400e6 --module MTC10F1084S1RC --with-wishbone-memory --wishbone-force-  
→32b --with-ethernet --remote-ip-address 192.168.100.100 --local-ip-address 192.  
→168.100.50"
```

17.3 Interacting with RDIMM DDR5 Tester Linux Target

First, load the bitstream onto the RDIMM DDR5 Tester with the help of OpenFPGALoader:

```
openFPGALoader --board antmicro_ddr5_tester build/ddr5_tester_linux/gateware/  
→antmicro_ddr5_tester.bit --freq 3e6
```

In order to connect to the board, assign the 192.168.100.100 IP Address to the Ethernet interface that is plugged to the DDR5 Tester board and set up the device if needed, e.g. by running:

```
ip addr add 192.168.100.100/24 dev $ETH  
ip link set dev $ETH up
```

Where ETH is the name of your Ethernet interface. When the Ethernet interface has been set up correctly, you may access the BIOS console on the DDR5 Tester with:

```
picocom -b 115200 /dev/ttyUSB2
```

17.4 Setting up a TFTP Server

Several Linux boot methods can be invoked here but booting via Ethernet is recommended. In order to enable netboot, you need to set up a TFTP server first.

Note

Running a TFTP server varies between distributions in terms of TFTP implementation names and locations of the configuration file.

As an example, below is a quick guide on how to configure a TFTP server for Arch Linux. Firstly, if not equipped already, get an implementation of a TFTP server, for example:

```
pacman -S tftp-hpa
```

The TFTP server is configured via a /etc/conf.d/tftpd file. Here's a suggested configuration for the DDR5 Tester Linux boot process:

```
TFTP_USERNAME="tftp"  
TFTPD_OPTIONS="--secure"  
TFTP_DIRECTORY="/srv/tftp"  
TFTP_ADDRESS="192.168.100.100:69"
```

TFTP_ADDRESS is specified with the --remote-ip-address option whilst building the target and the port is the default one for the TFTP server. The TFTP_DIRECTORY is the TFTP's root directory.

To start the TFTP service, run:

```
systemctl start tftpd
```

To check whether the TFTP sever is set up properly, run:

```
cd /srv/tftp/ && echo "TEST TFTP SERVER" > test  
cd ~/ && tftp 192.168.100.100 -c get test
```

The test file should appear in the home directory with “TEST TFTP SERVER” as its content.

17.5 Booting Linux on RDIMM DDR5 Tester Linux Target

You will need the following binaries:

- Linux kernel Image
- Compiled devicetree
- Opensbi's fw_jump.bin
- rootfs.cpio

All of these can be obtained with the use of provided firmware/ddr5_tester/buildroot buildroot external configuration. To build binaries with buildroot, run:

```
git clone --single-branch -b 2023.05.x https://github.com/buildroot/buildroot.git  
pushd buildroot  
make BR2_EXTERNAL="$(pwd)/../firmware/ddr5_tester/buildroot" ddr5_vexriscv_  
→defconfig
```

Then, transfer the binaries to the TFTP root directory:

```
mv buildroot/output/images/* /srv/tftp/  
mv /srv/tftp/fw_jump.bin /srv/tftp/opensbi.bin
```

The address map of the binaries alongside boot arguments can be contained within the boot.json file, for example:

```
{  
    "/srv/tftp/Image": "0x40000000",  
    "/srv/tftp/rv32.dtb": "0x40ef0000",  
    "/srv/tftp/rootfs.cpio": "0x42000000",  
    "/srv/tftp/opensbi.bin": "0x40f00000",  
    "bootargs": {  
        "r1": "0x00000000",  
        "r2": "0x40ef0000",  
        "r3": "0x00000000",  
        "addr": "0x40f00000"  
    }  
}
```

With Linux boot binaries in the TFTP's root directory with boot.json, netboot can be invoked from the BIOS console with:

```
netboot /srv/tftp/boot.json
```

Upon successful execution a similar log will be printed:

```
litex> netboot /srv/tftp/boot.json
Booting from network...
Local IP: 192.168.100.50
Remote IP: 192.168.100.100
Booting from /srv/tftp/boot.json (JSON)...
Copying /srv/tftp/Image to 0x40000000... (7395804 bytes)
Copying /srv/tftp/rv32.dtb to 0x40ef0000... (2463 bytes)
Copying /srv/tftp/rootfs.cpio to 0x42000000... (22128128 bytes)
Copying /srv/tftp/opensbi.bin to 0x40f00000... (1007056 bytes)
Executing booted program at 0x40f00000

===== Liftoff! =====
```

Then, the OpenSBI and Linux boot log should follow:

```
OpenSBI v1.3-24-g84c6dc1

----- / ____ \ ----- / ____| | \ \ \ \ |
| | | | | - -- | _ _ _ | (_____| |_) || | | | | |
| | | | | '_ \| / _ \ | '_ \| \ \ \ \ \| | _ < | |
| | __| | |_) | | _/ | | |_) ) | |_) || | |
\_\_/_/ | .__/ \_\_/_/ | | | | _/_/ | _/_/ | _/_/ |
          | |
          |_-|
```

Platform Name : LiteX / VexRiscv-SMP
Platform Features : medeleg
Platform HART Count : 8
Platform IPI Device : aclint-mswi
Platform Timer Device : aclint-mtimer @ 100000000Hz
Platform Console Device : litex_uart
(...)

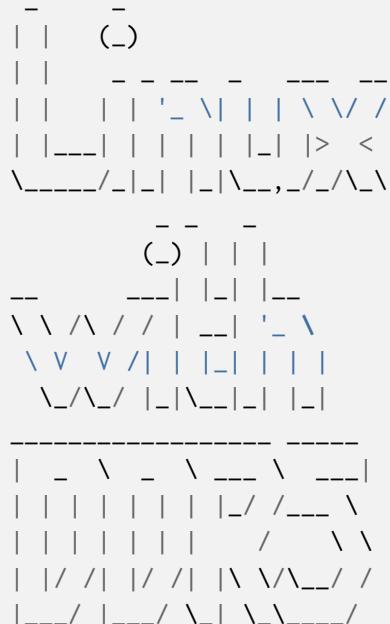
(continues on next page)

(continued from previous page)

```
[    0.000000] Linux version 5.11.0 (riscv32-buildroot-linux-gnu-gcc.br_real_
↳(Buildroot 2023.05.2-154-g787a633711) 11.4.0, GNU ld (GNU Binutils) 2.38) #2_
↳SMP Mon Sep 25 10:52:22 CEST 2023
[    0.000000] earlycon: sbi0 at I/O port 0x0 (options '')
[    0.000000] printk: bootconsole [sbi0] enabled
(...)
```

And then:

```
Welcome to Buildroot
buildroot login: root
```



32-bit RISC-V Linux running on DDR5 Tester.

```
login[65]: root login on 'console'
```

CHAPTER EIGHTEEN

PCIE SUPPORT

The selected hardware platforms used for DRAM testing include an optional PCIe interface break-routed from the on-board FPGA. These platforms are *SO-DIMM DDR5 Testers* supporting PCIe x4 and *RDIMM DDR5 Tester* in Revision 2.0 supporting PCIe x8. The platforms were designed in the form factor of PCIe cards and are mechanically compliant with host platforms with a PCIe root complex.



Fig. 18.1: RDIMM DDR5 Tester connected to Intel NUC-series host PC over PCIe x8.

Enabling a PCIe interface in the digital design allows for fast data exchange between the host PC, the FPGA and the memory.

Note

Enabling PCIe connectivity currently requires extending the existing Rowhammer tester codebase with setup-specific features. In particular, PCIe requires mapping the API/commands into a Rowhammer-specific API.

18.1 DRAM Bender integration

PCIe interface allows to integrate the RDIMM DDR5 Tester with third party DRAM testers. One of them is *DRAM Bender* project. In order to make the RDIMM DDR5 Tester compliant with DRAM Bender API it is required to transpile *Bender API* into Rowhammer Payload Executor

command set via an intermediate software layer. The software layer executed on the PCIe host side can map the 3rd-party command set into Rowhammer Tester format with respect to the mapping specified below.

The table below presents encodings for the supported Payload Executor instructions. The NOOP, LOOP and STOP instructions are used to control the flow of the Payload Executor module (are not propagated to the memory). The DFI instruction is a sequence of the DDR commands and its length is determined by the number of preconfigured DDR phases. The example below presents single Rowhammer Tester instruction for the device working in 4 phases.

This mapping is common for all DRAM variants, the LSB distinguishes the Payload Executor control commands from the DFI commands.

After having transpiled the program, it is expected for the driver to transfer the payload over the PCIe interface onto the Rowhammer Tester platform.

The table below represents the DDR-specific instructions encoding for the Rowhammer Tester platform. A row of the table represents the DDR command (if one-cycle) or a DFI PHASE of the command.

Each of the DDR command in the DFI sequence can be prefixed with a TIMESLICE argument that determines the delay with which the next command in sequence is issued.

CHAPTER
NINETEEN

DOCUMENTATION FOR ROW HAMMER TESTER ARTY-A7

19.1 Modules

19.1.1 Interrupt Controller

This device has an EventManager-based interrupt system. Individual modules generate *events* which are wired into a central interrupt controller.

When an interrupt occurs, you should look the interrupt number up in the CPU- specific interrupt table and then call the relevant module.

Assigned Interrupts

The following interrupts are assigned on this system:

Interrupt	Module
1	<i>TIMER0</i>
0	<i>UART</i>

19.2 Register Groups

19.2.1 LEDs

Register Listing for LEDs

Register	Address
<i>LEDS_OUT</i>	<i>0xf0000000</i>

LEDS_OUT

Address: *0xf0000000* + *0x0* = *0xf0000000*

Led Output(s) Control.

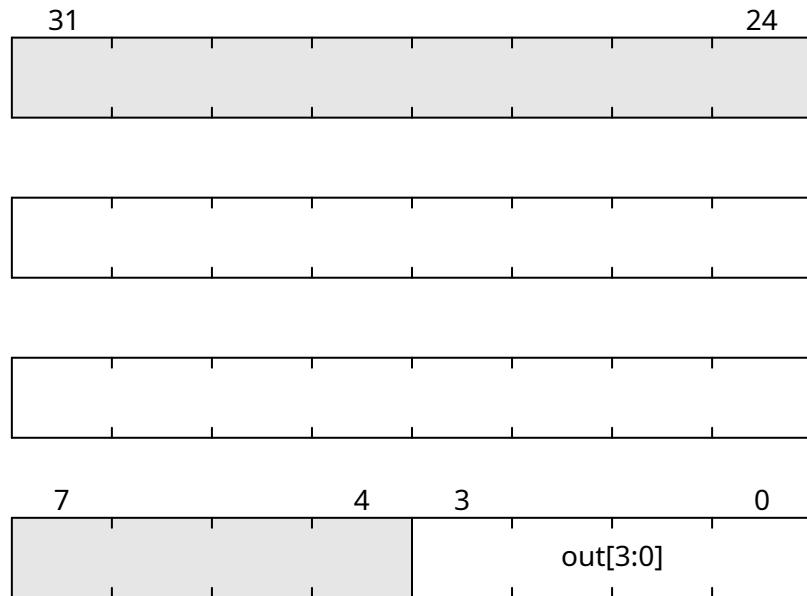


Fig. 19.1: LEDS_OUT

19.2.2 DDRPHY

Register Listing for DDRPHY

Register	Address
<i>DDRPHY_RST</i>	0xf0000800
<i>DDRPHY_DLW_SEL</i>	0xf0000804
<i>DDRPHY_HALF_SYS8X_TAPS</i>	0xf0000808
<i>DDRPHY_WLEVEL_EN</i>	0xf000080c
<i>DDRPHY_WLEVEL_STROBE</i>	0xf0000810
<i>DDRPHY_RDLY_DQ_RST</i>	0xf0000814
<i>DDRPHY_RDLY_DQ_INC</i>	0xf0000818
<i>DDRPHY_RDLY_DQ_BITSLIP_RST</i>	0xf000081c
<i>DDRPHY_RDLY_DQ_BITSLIP</i>	0xf0000820
<i>DDRPHY_WDLY_DQ_BITSLIP_RST</i>	0xf0000824
<i>DDRPHY_WDLY_DQ_BITSLIP</i>	0xf0000828
<i>DDRPHY_RDPHASE</i>	0xf000082c
<i>DDRPHY_WRPHASE</i>	0xf0000830

DDRPHY_RST

Address: $0xf0000800 + 0x0 = 0xf0000800$



Fig. 19.2: DDRPHY_RST

DDRPHY_DLY_SEL

Address: 0xf0000800 + 0x4 = 0xf0000804

DDRPHY_HALF_SYS8X_TAPS

Address: 0xf0000800 + 0x8 = 0xf0000808

DDRPHY_WLEVEL_EN

Address: 0xf0000800 + 0xc = 0xf000080c

DDRPHY_WLEVEL_STROBE

Address: 0xf0000800 + 0x10 = 0xf0000810

DDRPHY_RDLY_DQ_RST

Address: 0xf0000800 + 0x14 = 0xf0000814

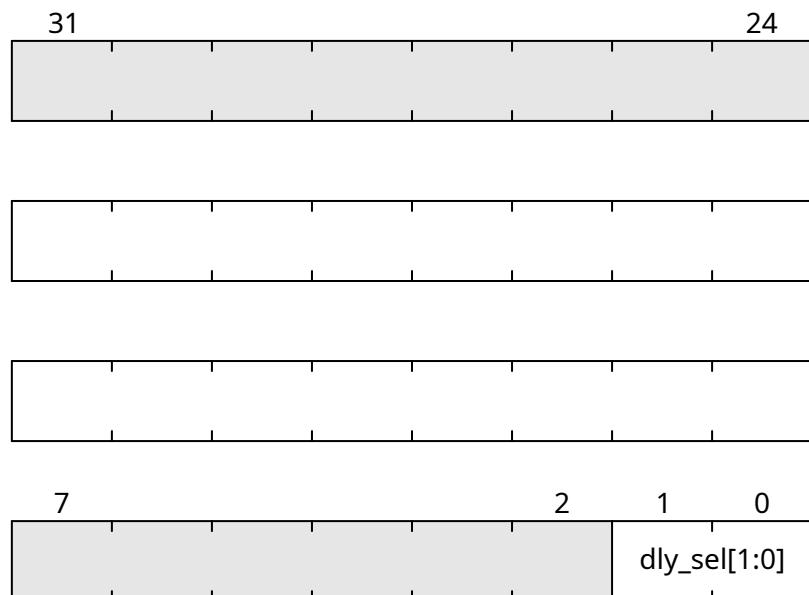


Fig. 19.3: `DDRPHY_DLY_SEL`

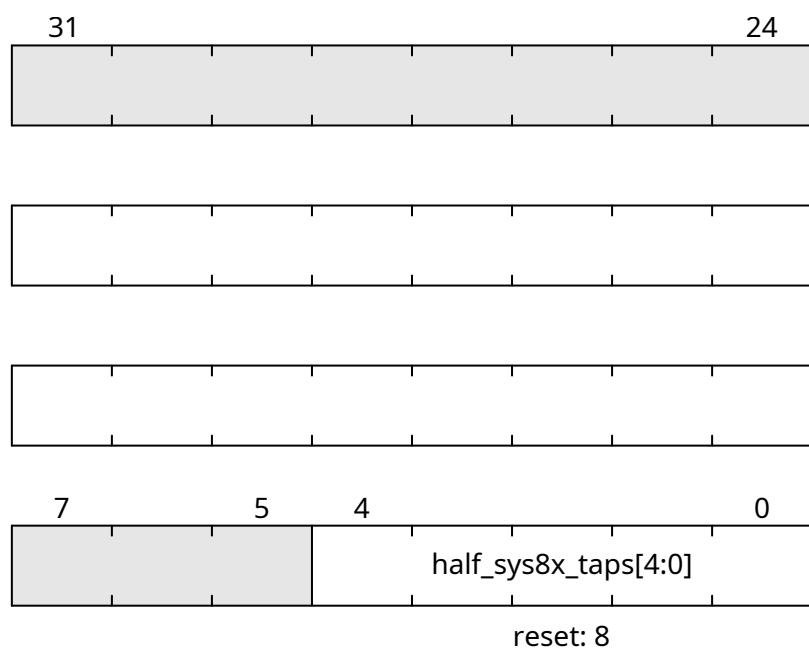


Fig. 19.4: `DDRPHY_HALF_SYS8X_TAPS`



Fig. 19.5: `DDRPHY_WLEVEL_EN`

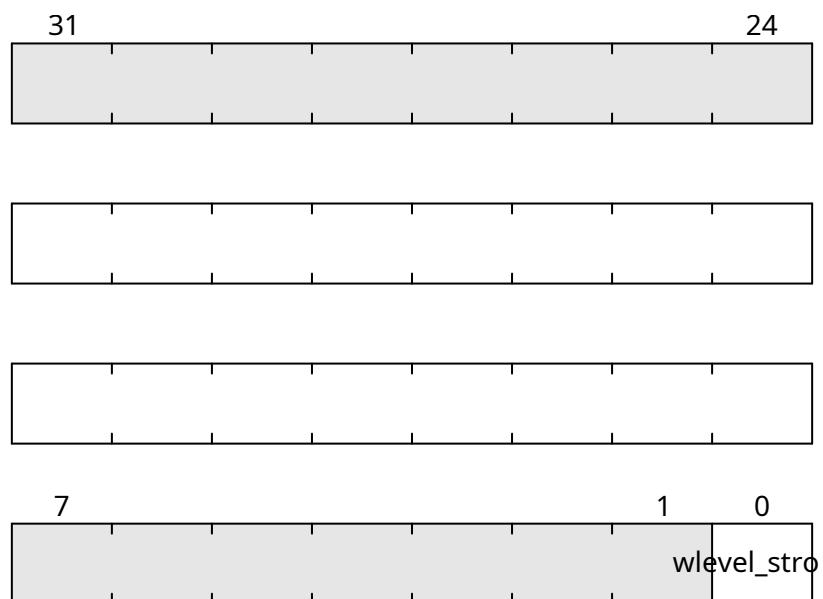


Fig. 19.6: `DDRPHY_WLEVEL_STROBE`

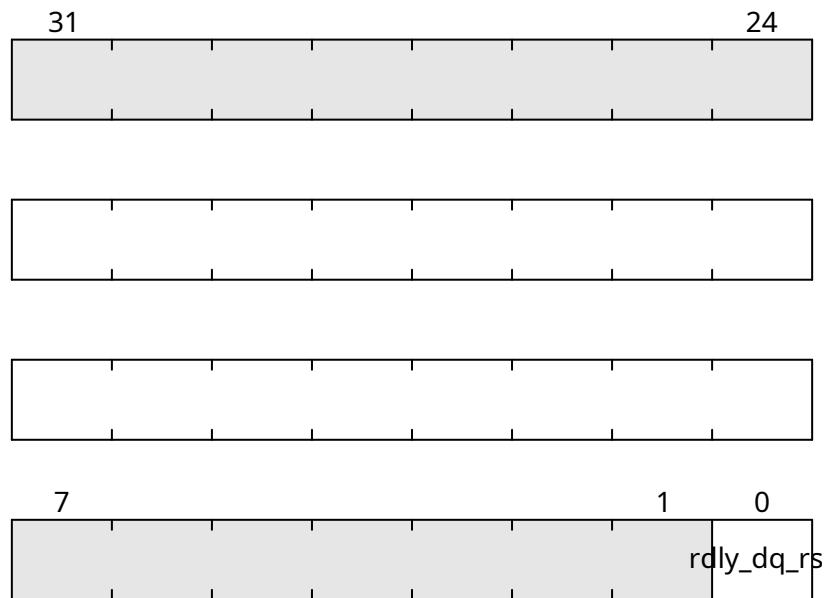


Fig. 19.7: DDRPHY_RDLY_DQ_RST

DDRPHY_RDLY_DQ_INC

Address: 0xf0000800 + 0x18 = 0xf0000818

DDRPHY_RDLY_DQ_BITSLIP_RST

Address: 0xf0000800 + 0x1c = 0xf000081c

DDRPHY_RDLY_DQ_BITSLIP

Address: 0xf0000800 + 0x20 = 0xf0000820

DDRPHY_WDLY_DQ_BITSLIP_RST

Address: 0xf0000800 + 0x24 = 0xf0000824

DDRPHY_WDLY_DQ_BITSLIP

Address: 0xf0000800 + 0x28 = 0xf0000828



Fig. 19.8: DDRPHY_RDLY_DQ_INC

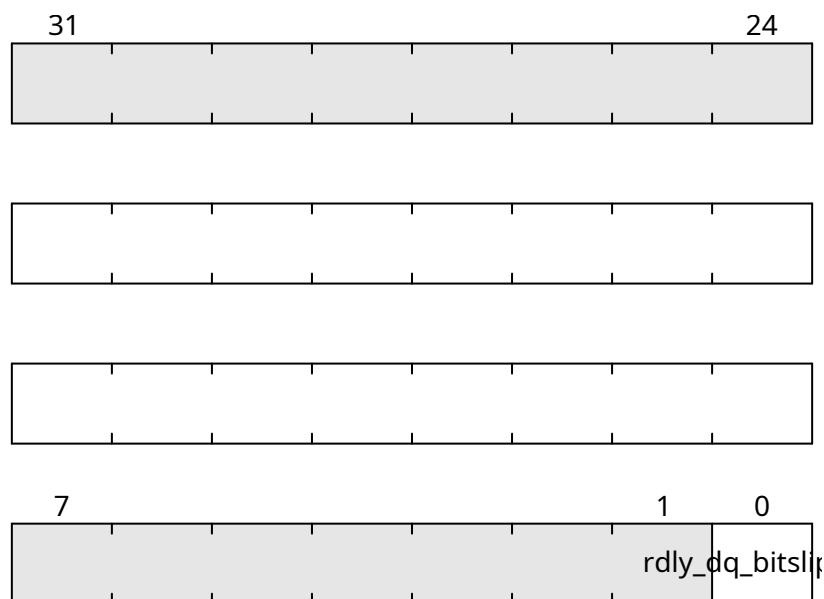


Fig. 19.9: DDRPHY_RDLY_DQ_BITSLIP_RST



Fig. 19.10: DDRPHY_RDLY_DQ_BITSLIP

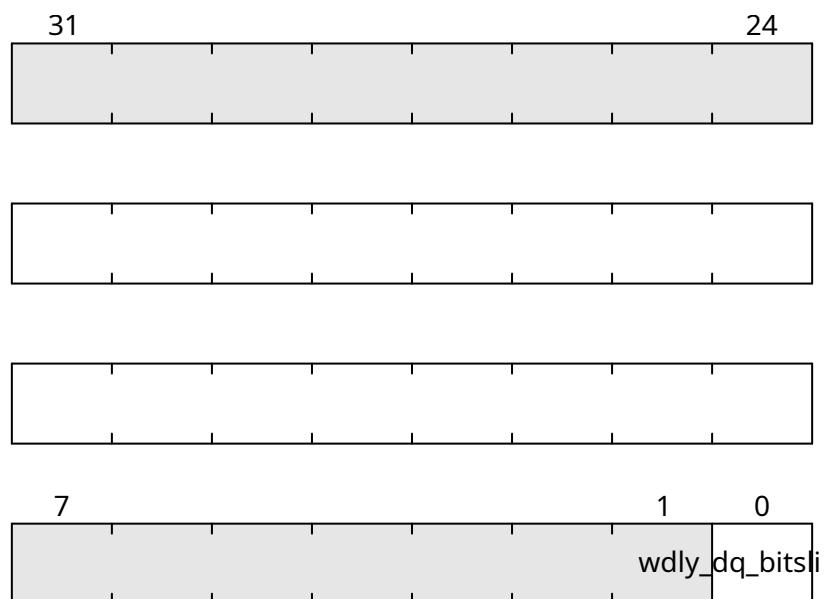


Fig. 19.11: DDRPHY_WDLY_DQ_BITSLIP_RST



Fig. 19.12: DDRPHY_WDLY_DQ_BITSLIP

DDRPHY_RDPHASE

Address: $0xf0000800 + 0x2c = 0xf000082c$

DDRPHY_WRPHASE

Address: $0xf0000800 + 0x30 = 0xf0000830$

19.2.3 CONTROLLER_SETTINGS

Allows to change LiteDRAMController behaviour at runtime

Register Listing for CONTROLLER_SETTINGS

Register	Address
CONTROLLER_SETTINGS_REFRESH	0xf0001000

CONTROLLER_SETTINGS_REFRESH

Address: $0xf0001000 + 0x0 = 0xf0001000$

Enable/disable Refresh commands sending

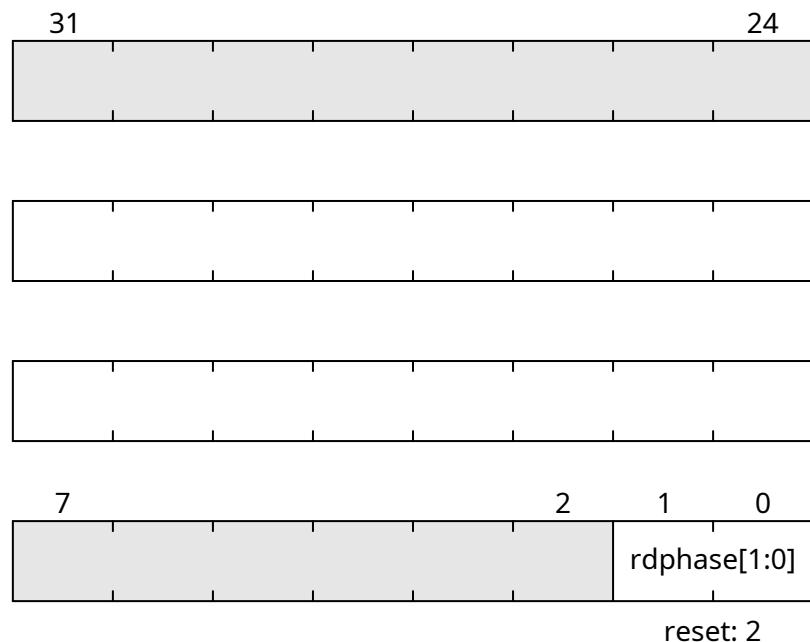


Fig. 19.13: DDRPHY_RDPHASE

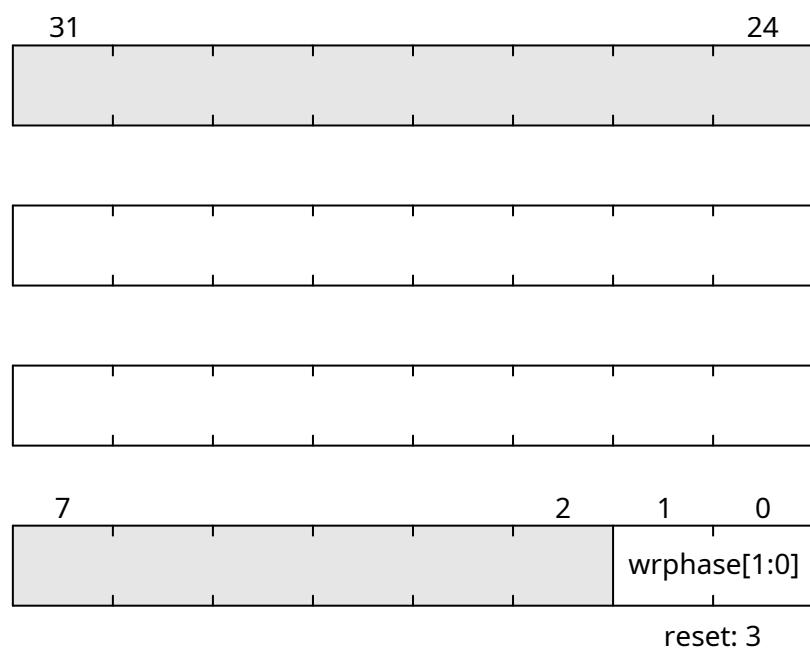


Fig. 19.14: DDRPHY_WRPAGE

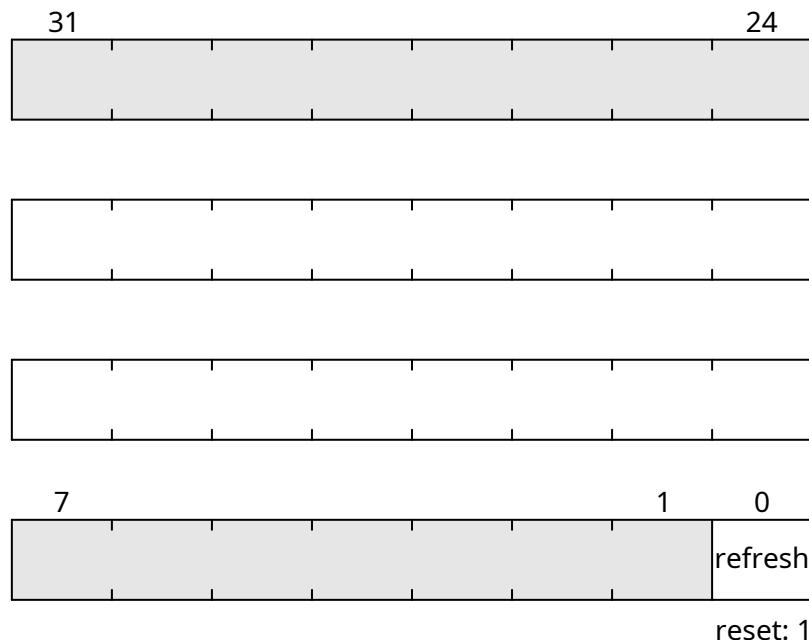


Fig. 19.15: CONTROLLER_SETTINGS_REFRESH

19.2.4 DDRCTRL

Register Listing for DDRCTRL

Register	Address
<i>DDRCTRL_INIT_DONE</i>	<i>0xf0001800</i>
<i>DDRCTRL_INIT_ERROR</i>	<i>0xf0001804</i>

DDRCTRL_INIT_DONE

Address: $0xf0001800 + 0x0 = 0xf0001800$

DDRCTRL_INIT_ERROR

Address: $0xf0001800 + 0x4 = 0xf0001804$

19.2.5 ETHPHY

Register Listing for ETHPHY

Register	Address
<i>ETHPHY_CRG_RESET</i>	<i>0xf0002000</i>
<i>ETHPHY_MDIO_W</i>	<i>0xf0002004</i>
<i>ETHPHY_MDIO_R</i>	<i>0xf0002008</i>



Fig. 19.16: DDRCTRL_INIT_DONE

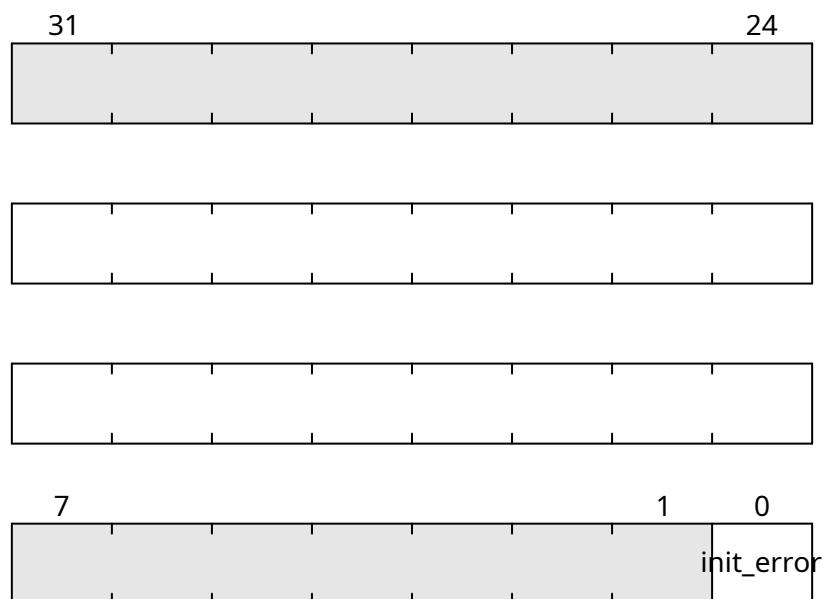


Fig. 19.17: DDRCTRL_INIT_ERROR

ETHPHY_CRG_RESET

Address: $0xf0002000 + 0x0 = 0xf0002000$

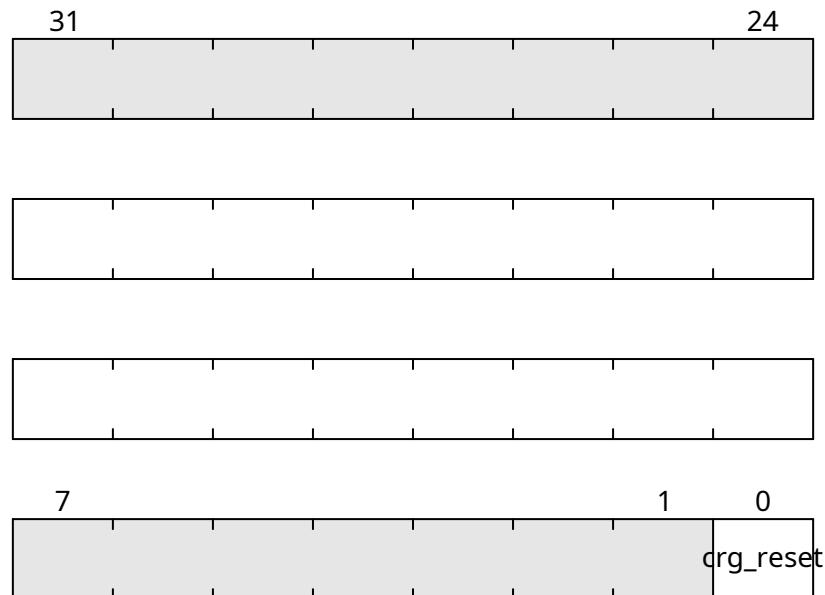


Fig. 19.18: ETHPHY_CRG_RESET

ETHPHY_MDIO_W

Address: $0xf0002000 + 0x4 = 0xf0002004$

Field	Name	Description

ETHPHY_MDIO_R

Address: $0xf0002000 + 0x8 = 0xf0002008$

Field	Name	Description

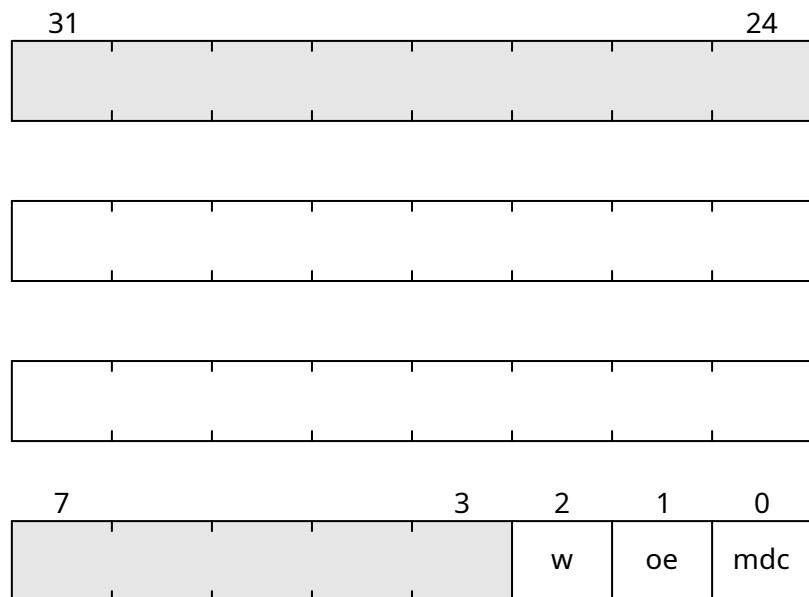


Fig. 19.19: ETHPHY_MDIO_W



Fig. 19.20: ETHPHY_MDIO_R

19.2.6 ROWHAMMER

Row Hammer DMA attacker

This module allows to perform a Row Hammer attack by configuring it with two addresses that map to different rows of a single bank. When enabled, it will perform alternating DMA reads from the given locations, which will result in the DRAM controller having to repeatedly open/close rows at each read access.

Register Listing for ROWHAMMER

Register	Address
<i>ROWHAMMER_ENABLED</i>	0xf0002800
<i>ROWHAMMER_ADDRESS1</i>	0xf0002804
<i>ROWHAMMER_ADDRESS2</i>	0xf0002808
<i>ROWHAMMER_COUNT</i>	0xf000280c

ROWHAMMER_ENABLED

Address: $0xf0002800 + 0x0 = 0xf0002800$

Used to start/stop the operation of the module



Fig. 19.21: ROWHAMMER_ENABLED

ROWHAMMER_ADDRESS1

Address: $0xf0002800 + 0x4 = 0xf0002804$

First attacked address

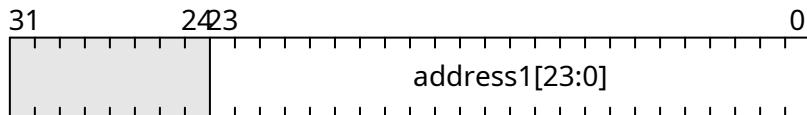


Fig. 19.22: ROWHAMMER_ADDRESS1

ROWHAMMER_ADDRESS2

Address: $0xf0002800 + 0x8 = 0xf0002808$

Second attacked address

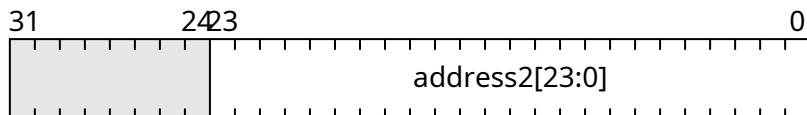


Fig. 19.23: ROWHAMMER_ADDRESS2

ROWHAMMER_COUNT

Address: $0xf0002800 + 0xc = 0xf000280c$

This is the number of DMA accesses performed. When the module is enabled, the value can be freely read. When the module is disabled, the register is clear-on-write and has to be read before the next attack.

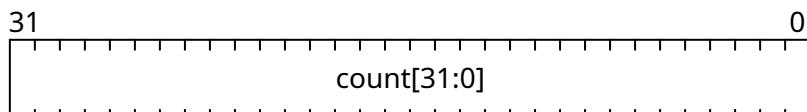


Fig. 19.24: ROWHAMMER_COUNT

19.2.7 WRITER

DMA DRAM writer.

Allows to fill DRAM with a predefined pattern using DMA.

Pattern

Provides access to RAM to store access pattern: *mem_addr* and *mem_data*. The pattern address space can be limited using the *data_mask*.

For example, having *mem_addr* filled with $[0x04, 0x02, 0x03, \dots]$ and *mem_data* filled with $[0xff, 0xaa, 0x55, \dots]$ and setting *data_mask* = *0b01*, the pattern $[(address, data), \dots]$ written will be: $[(0x04, 0xff), (0x02, 0xaa), (0x04, 0xff), \dots]$ (wraps due to masking).

DRAM memory range that is being accessed can be configured using *mem_mask*.

To use this module, make sure that *ready* is 1, then write the desired number of transfers to *count*. Writing to the *start* CSR will initialize the operation. When the operation is ongoing *ready* will be 0.

Register Listing for WRITER

Register	Address
<i>WRITER_START</i>	0xf0003000
<i>WRITER_READY</i>	0xf0003004
<i>WRITER_MODULO</i>	0xf0003008
<i>WRITER_COUNT</i>	0xf000300c
<i>WRITER_DONE</i>	0xf0003010
<i>WRITER_MEM_MASK</i>	0xf0003014
<i>WRITER_DATA_MASK</i>	0xf0003018
<i>WRITER_DATA_DIV</i>	0xf000301c
<i>WRITER_INVERTER_DIVISOR_MASK</i>	0xf0003020
<i>WRITER_INVERTER_SELECTION_MASK</i>	0xf0003024
<i>WRITER_LAST_ADDRESS</i>	0xf0003028

WRITER_START

Address: $0xf0003000 + 0x0 = 0xf0003000$

Write to the register starts the transfer (if ready=1)



Fig. 19.25: WRITER_START

WRITER_READY

Address: $0xf0003000 + 0x4 = 0xf0003004$

Indicates that the transfer is not ongoing



Fig. 19.26: WRITER_READY

WRITER_MODULO

Address: $0xf0003000 + 0x8 = 0xf0003008$

When set use modulo to calculate DMA transfers address rather than bit masking

WRITER_COUNT

Address: $0xf0003000 + 0xc = 0xf000300c$

Desired number of DMA transfers

WRITER_DONE

Address: $0xf0003000 + 0x10 = 0xf0003010$

Number of completed DMA transfers

WRITER_MEM_MASK

Address: $0xf0003000 + 0x14 = 0xf0003014$

DRAM address mask for DMA transfers



Fig. 19.27: WRITER_MODULO

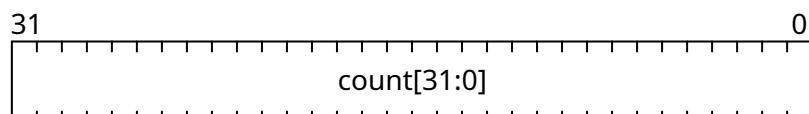


Fig. 19.28: WRITER_COUNT

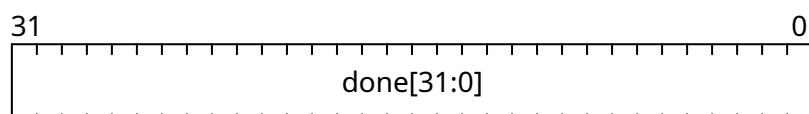


Fig. 19.29: WRITER_DONE

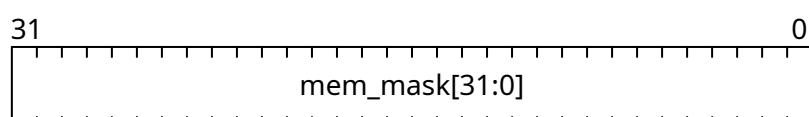


Fig. 19.30: WRITER_MEM_MASK

WRITER_DATA_MASK

Address: $0xf0003000 + 0x18 = 0xf0003018$

Pattern memory address mask

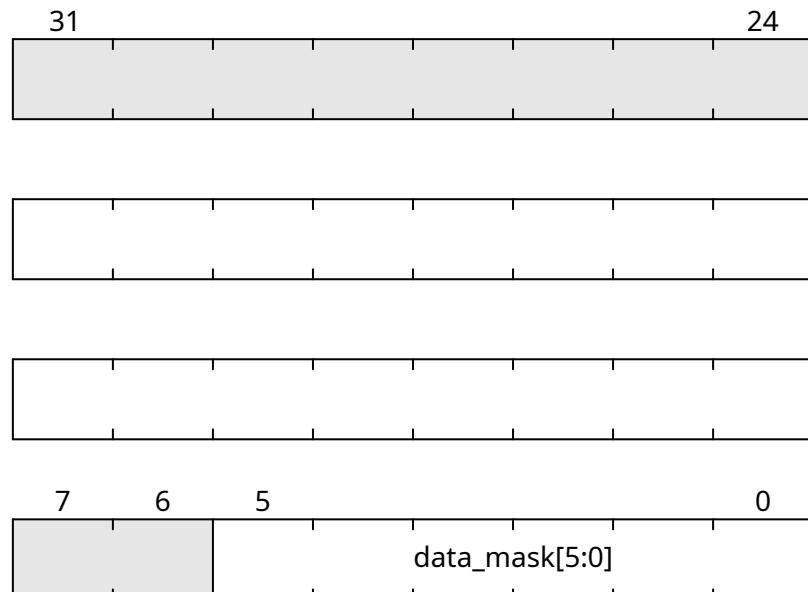


Fig. 19.31: WRITER_DATA_MASK

WRITER_DATA_DIV

Address: $0xf0003000 + 0x1c = 0xf000301c$

Pattern memory address divisor-1

WRITER_INVERTER_DIVISOR_MASK

Address: $0xf0003000 + 0x20 = 0xf0003020$

Divisor mask for selecting rows for which pattern data gets inverted

WRITER_INVERTER_SELECTION_MASK

Address: $0xf0003000 + 0x24 = 0xf0003024$

Selection mask for selecting rows for which pattern data gets inverted

WRITER_LAST_ADDRESS

Address: $0xf0003000 + 0x28 = 0xf0003028$

Number of completed DMA transfers

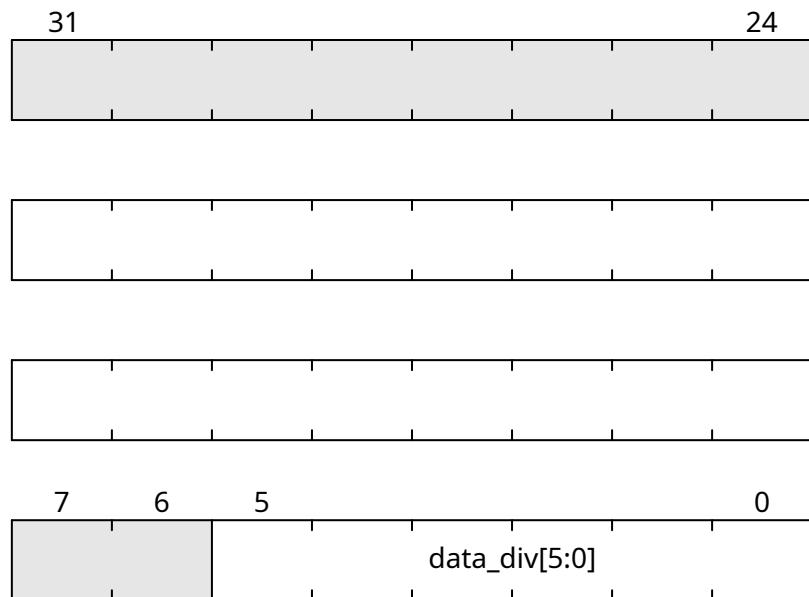


Fig. 19.32: WRITER_DATA_DIV

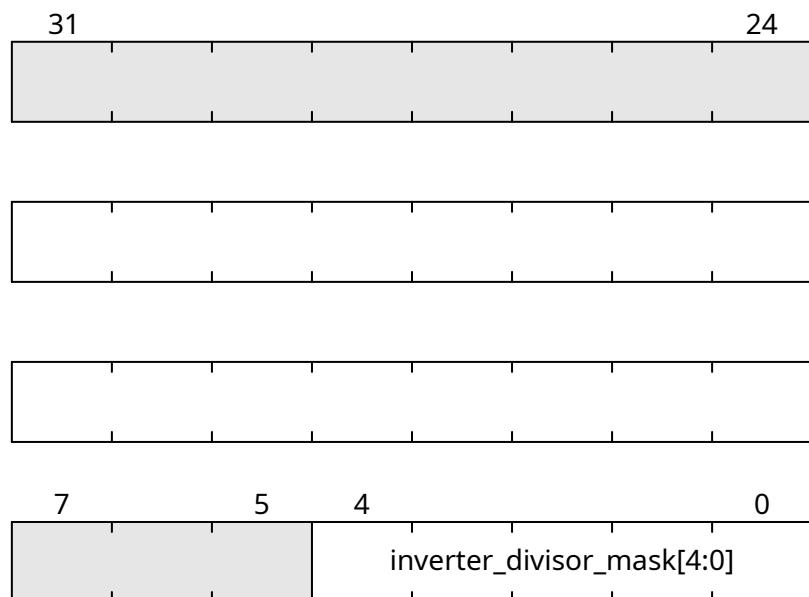


Fig. 19.33: WRITER_INVERTER_DIVISOR_MASK

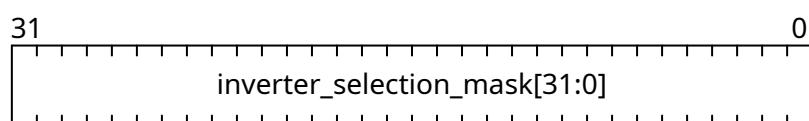


Fig. 19.34: WRITER_INVERTER_SELECTION_MASK

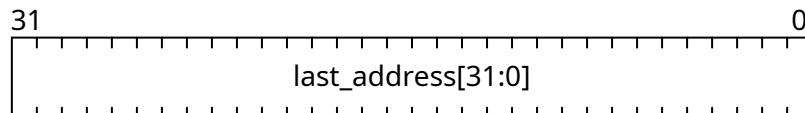


Fig. 19.35: WRITER_LAST_ADDRESS

19.2.8 READER

DMA DRAM reader.

Allows to check DRAM contents against a predefined pattern using DMA.

Pattern

Provides access to RAM to store access pattern: *mem_addr* and *mem_data*. The pattern address space can be limited using the *data_mask*.

For example, having *mem_addr* filled with [0x04, 0x02, 0x03, ...] and *mem_data* filled with [0xff, 0xaa, 0x55, ...] and setting *data_mask* = 0b01, the pattern [(address, data), ...] written will be: [(0x04, 0xff), (0x02, 0xaa), (0x04, 0xff), ...] (wraps due to masking).

DRAM memory range that is being accessed can be configured using *mem_mask*.

To use this module, make sure that *ready* is 1, then write the desired number of transfers to *count*. Writing to the *start* CSR will initialize the operation. When the operation is ongoing *ready* will be 0.

Reading errors

This module allows to check the locations of errors in the memory. It scans the configured memory area and compares the values read to the predefined pattern. If *skip_fifo* is 0, this module will stop after each error encountered, so that it can be examined. Wait until the *error_ready* CSR is 1. Then use the CSRs *error_offset*, *error_data* and *error_expected* to examine the errors in the current transfer. To continue reading, write 1 to *error_continue* CSR. Setting *skip_fifo* to 1 will disable this behavior entirely.

The final number of errors can be read from *error_count*. NOTE: This value represents the number of erroneous DMA transfers.

The current progress can be read from the *done* CSR.

Register Listing for READER

Register	Address
<i>READER_START</i>	0xf0003800
<i>READER_READY</i>	0xf0003804
<i>READER_MODULO</i>	0xf0003808
<i>READER_COUNT</i>	0xf000380c
<i>READER_DONE</i>	0xf0003810
<i>READER_MEM_MASK</i>	0xf0003814
<i>READER_DATA_MASK</i>	0xf0003818
<i>READER_DATA_DIV</i>	0xf000381c
<i>READER_INVERTER_DIVISOR_MASK</i>	0xf0003820
<i>READER_INVERTER_SELECTION_MASK</i>	0xf0003824
<i>READER_ERROR_COUNT</i>	0xf0003828
<i>READER_SKIP_FIFO</i>	0xf000382c
<i>READER_ERROR_OFFSET</i>	0xf0003830
<i>READER_ERROR_DATA3</i>	0xf0003834
<i>READER_ERROR_DATA2</i>	0xf0003838
<i>READER_ERROR_DATA1</i>	0xf000383c
<i>READER_ERROR_DATA0</i>	0xf0003840
<i>READER_ERROR_EXPECTED3</i>	0xf0003844
<i>READER_ERROR_EXPECTED2</i>	0xf0003848
<i>READER_ERROR_EXPECTED1</i>	0xf000384c
<i>READER_ERROR_EXPECTED0</i>	0xf0003850
<i>READER_ERROR_READY</i>	0xf0003854
<i>READER_ERROR_CONTINUE</i>	0xf0003858

READER_START

Address: $0xf0003800 + 0x0 = 0xf0003800$

Write to the register starts the transfer (if ready=1)

READER_READY

Address: $0xf0003800 + 0x4 = 0xf0003804$

Indicates that the transfer is not ongoing

READER_MODULO

Address: $0xf0003800 + 0x8 = 0xf0003808$

When set use modulo to calculate DMA transfers address rather than bit masking

READER_COUNT

Address: $0xf0003800 + 0xc = 0xf000380c$

Desired number of DMA transfers

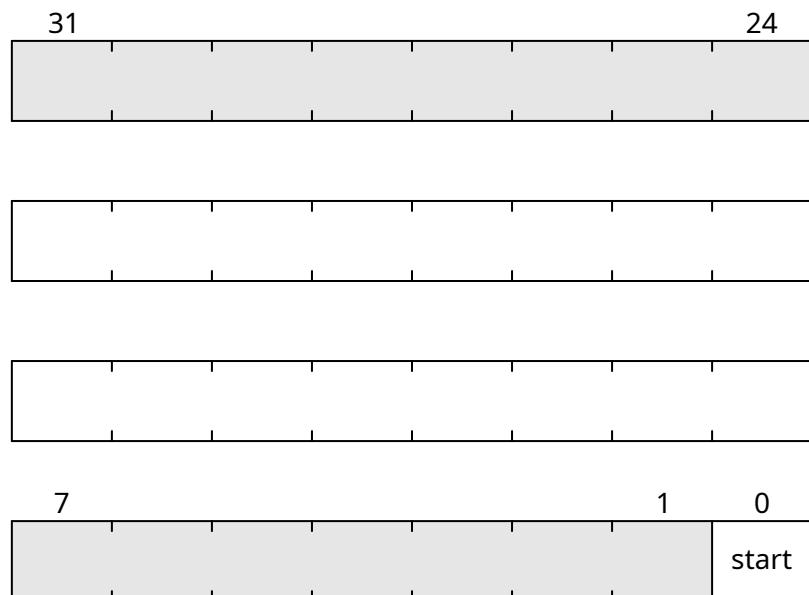


Fig. 19.36: READER_START



Fig. 19.37: READER_READY



Fig. 19.38: READER_MODULO

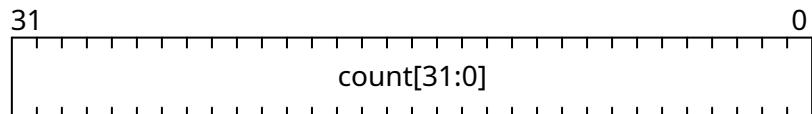


Fig. 19.39: READER_COUNT

READER_DONE

Address: $0xf0003800 + 0x10 = 0xf0003810$

Number of completed DMA transfers

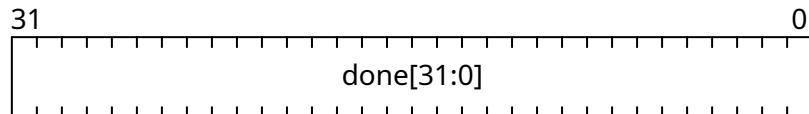


Fig. 19.40: READER_DONE

READER_MEM_MASK

Address: $0xf0003800 + 0x14 = 0xf0003814$

DRAM address mask for DMA transfers

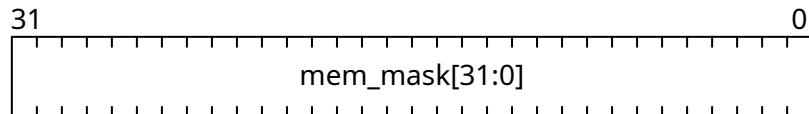


Fig. 19.41: READER_MEM_MASK

READER_DATA_MASK

Address: $0xf0003800 + 0x18 = 0xf0003818$

Pattern memory address mask

READER_DATA_DIV

Address: $0xf0003800 + 0x1c = 0xf000381c$

Pattern memory address divisor-1

READER_INVERTER_DIVISOR_MASK

Address: $0xf0003800 + 0x20 = 0xf0003820$

Divisor mask for selecting rows for which pattern data gets inverted

READER_INVERTER_SELECTION_MASK

Address: $0xf0003800 + 0x24 = 0xf0003824$

Selection mask for selecting rows for which pattern data gets inverted

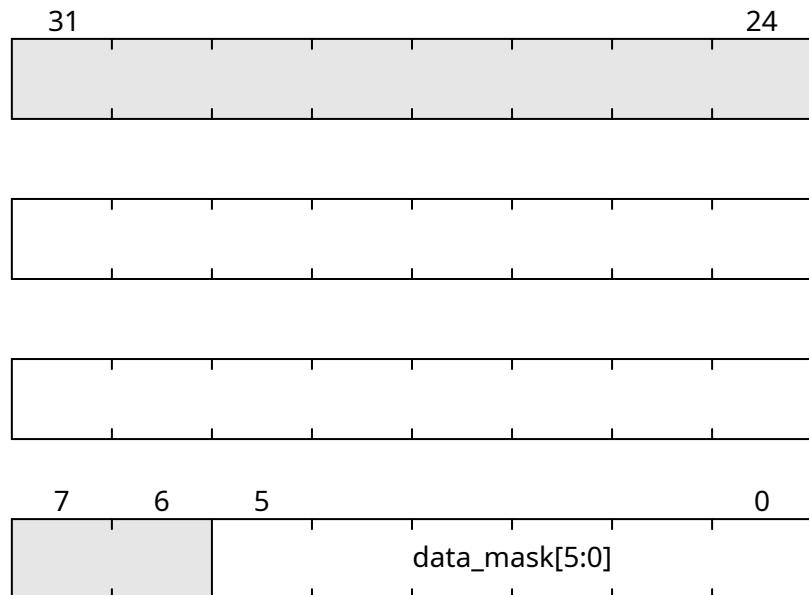


Fig. 19.42: READER_DATA_MASK

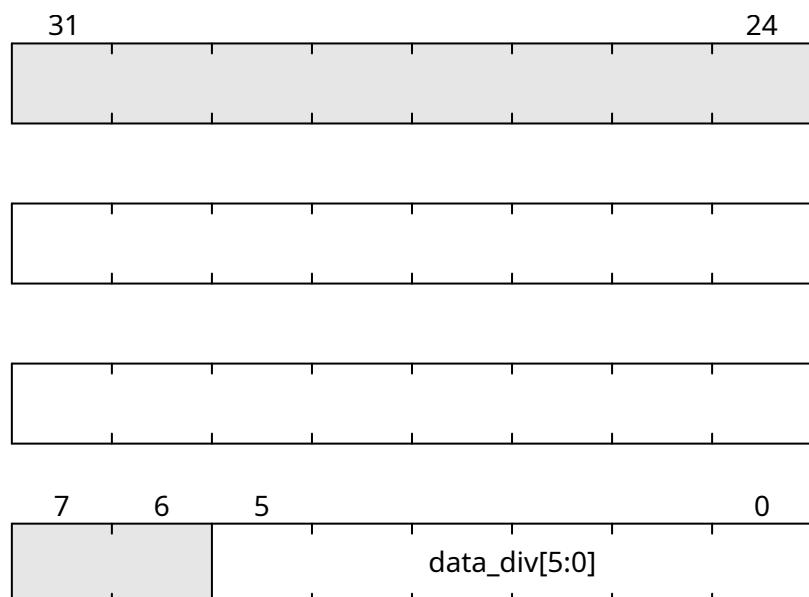


Fig. 19.43: READER_DATA_DIV

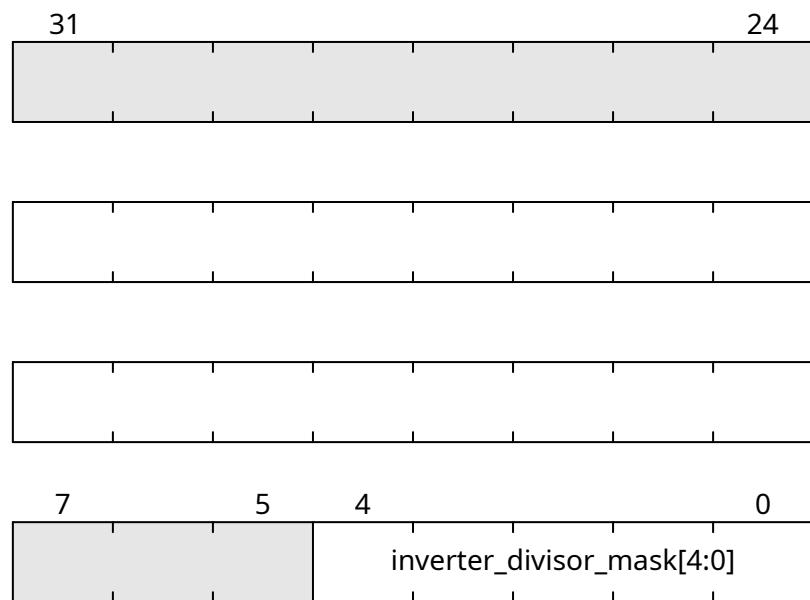


Fig. 19.44: READER_INVERTER_DIVISOR_MASK

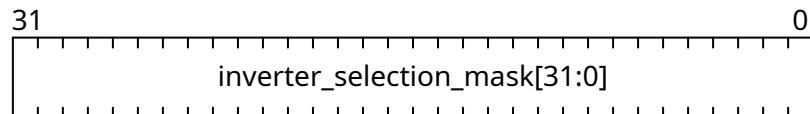


Fig. 19.45: READER_INVERTER_SELECTION_MASK

READER_ERROR_COUNT

Address: $0xf0003800 + 0x28 = 0xf0003828$

Number of errors detected

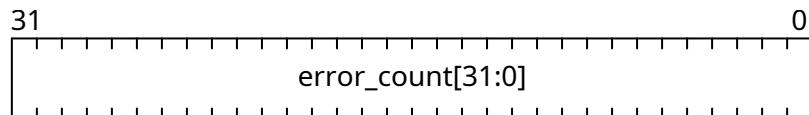


Fig. 19.46: READER_ERROR_COUNT

READER_SKIP_FIFO

Address: $0xf0003800 + 0x2c = 0xf000382c$

Skip waiting for user to read the errors FIFO



Fig. 19.47: READER_SKIP_FIFO

READER_ERROR_OFFSET

Address: $0xf0003800 + 0x30 = 0xf0003830$

Current offset of the error

READER_ERROR_DATA3

Address: $0xf0003800 + 0x34 = 0xf0003834$

Bits 96-127 of READER_ERROR_DATA. Erroneous value read from DRAM memory

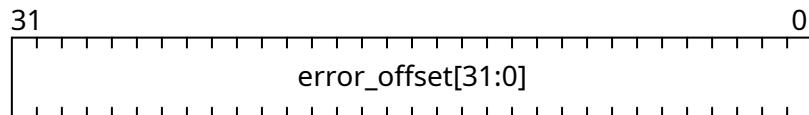


Fig. 19.48: READER_ERROR_OFFSET

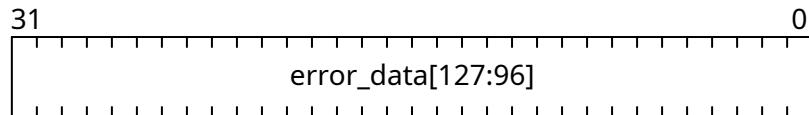


Fig. 19.49: READER_ERROR_DATA3

READER_ERROR_DATA2

Address: $0xf0003800 + 0x38 = 0xf0003838$

Bits 64-95 of *READER_ERROR_DATA*.

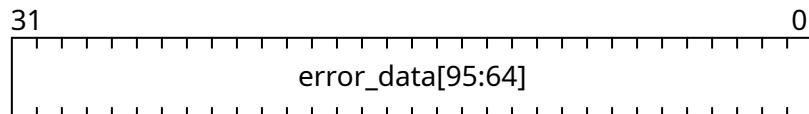


Fig. 19.50: READER_ERROR_DATA2

READER_ERROR_DATA1

Address: $0xf0003800 + 0x3c = 0xf000383c$

Bits 32-63 of *READER_ERROR_DATA*.

READER_ERROR_DATA0

Address: $0xf0003800 + 0x40 = 0xf0003840$

Bits 0-31 of *READER_ERROR_DATA*.

READER_ERROR_EXPECTED3

Address: $0xf0003800 + 0x44 = 0xf0003844$

Bits 96-127 of *READER_ERROR_EXPECTED*. Value expected to be read from DRAM memory

READER_ERROR_EXPECTED2

Address: $0xf0003800 + 0x48 = 0xf0003848$

Bits 64-95 of *READER_ERROR_EXPECTED*.

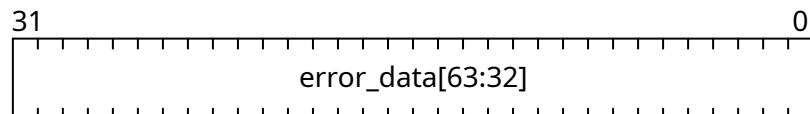


Fig. 19.51: READER_ERROR_DATA1

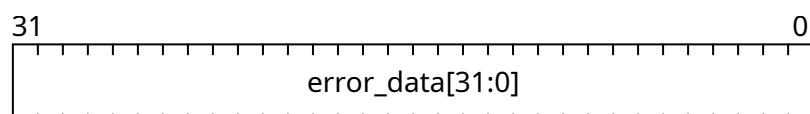


Fig. 19.52: READER_ERROR_DATA0

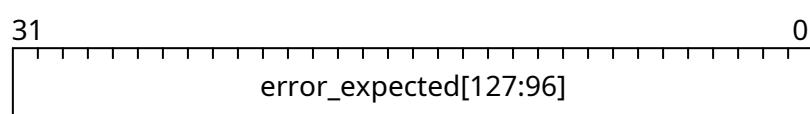


Fig. 19.53: READER_ERROR_EXPECTED3

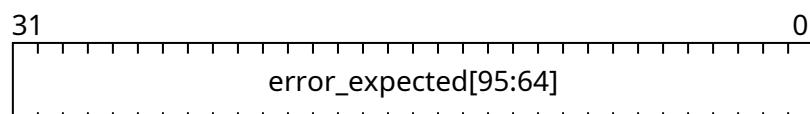


Fig. 19.54: READER_ERROR_EXPECTED2

READER_ERROR_EXPECTED1

Address: $0xf0003800 + 0x4c = 0xf000384c$

Bits 32-63 of *READER_ERROR_EXPECTED*.

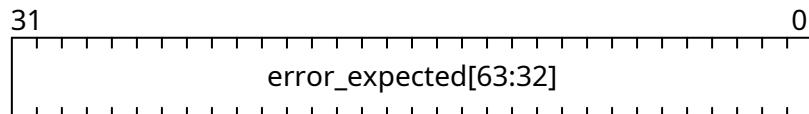


Fig. 19.55: READER_ERROR_EXPECTED1

READER_ERROR_EXPECTED0

Address: $0xf0003800 + 0x50 = 0xf0003850$

Bits 0-31 of *READER_ERROR_EXPECTED*.

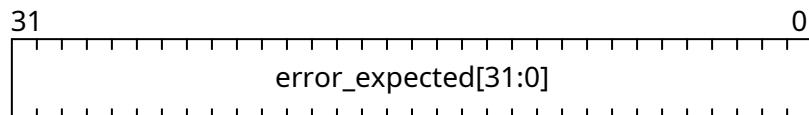


Fig. 19.56: READER_ERROR_EXPECTED0

READER_ERROR_READY

Address: $0xf0003800 + 0x54 = 0xf0003854$

Error detected and ready to read

READER_ERROR_CONTINUE

Address: $0xf0003800 + 0x58 = 0xf0003858$

Continue reading until the next error

19.2.9 DFI_SWITCH

Register Listing for DFI_SWITCH

Register	Address
<i>DFI_SWITCH_REFRESH_COUNT1</i>	<i>0xf0004000</i>
<i>DFI_SWITCH_REFRESH_COUNT0</i>	<i>0xf0004004</i>
<i>DFI_SWITCH_AT_REFRESH1</i>	<i>0xf0004008</i>
<i>DFI_SWITCH_AT_REFRESH0</i>	<i>0xf000400c</i>
<i>DFI_SWITCH_REFRESH_UPDATE</i>	<i>0xf0004010</i>

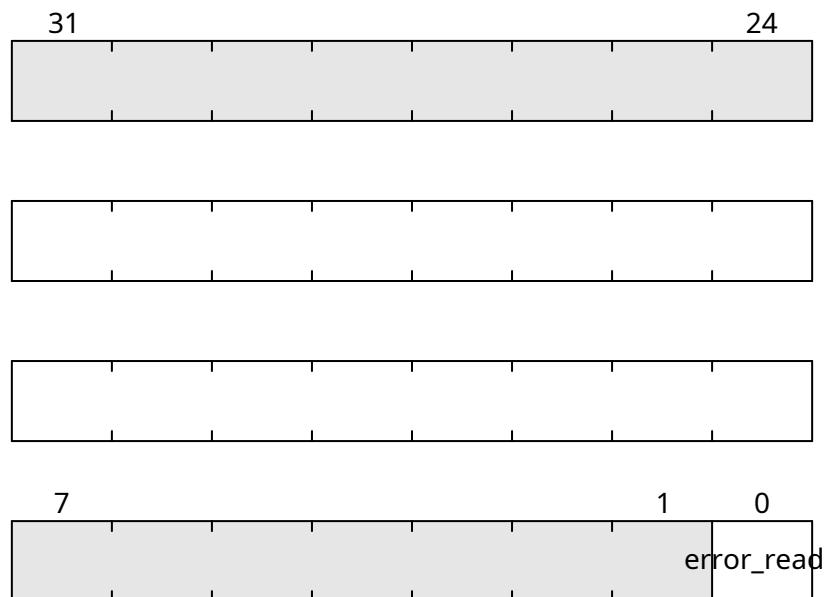


Fig. 19.57: READER_ERROR_READY

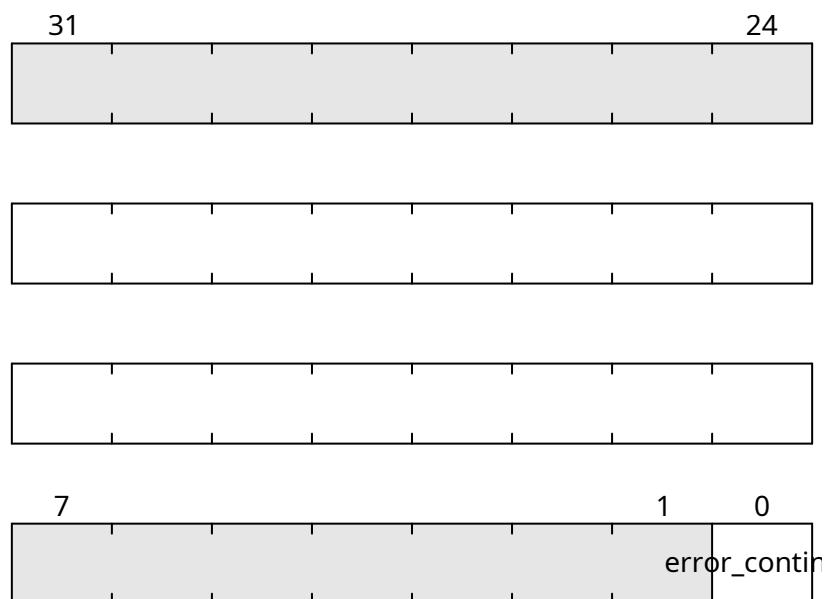


Fig. 19.58: READER_ERROR_CONTINUE

DFI_SWITCH_REFRESH_COUNT1

Address: $0xf0004000 + 0x0 = 0xf0004000$

Bits 32-63 of *DFI_SWITCH_REFRESH_COUNT*. Count of all refresh commands issued (both by Memory Controller and the Payload Executor). Value is latched from internal counter on mode transition: MC -> PE or by writing to the *refresh_update* CSR.

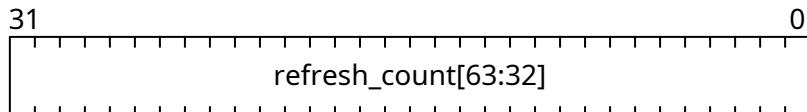


Fig. 19.59: DFI_SWITCH_REFRESH_COUNT1

DFI_SWITCH_REFRESH_COUNT0

Address: $0xf0004000 + 0x4 = 0xf0004004$

Bits 0-31 of *DFI_SWITCH_REFRESH_COUNT*.

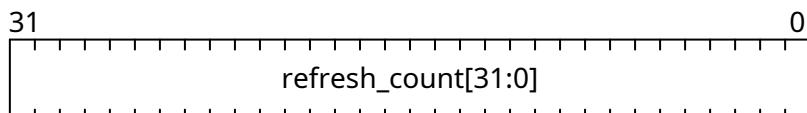


Fig. 19.60: DFI_SWITCH_REFRESH_COUNT0

DFI_SWITCH_AT_REFRESH1

Address: $0xf0004000 + 0x8 = 0xf0004008$

Bits 32-63 of *DFI_SWITCH_AT_REFRESH*. If set to a value different than 0 the mode transition MC -> PE will be performed only when the value of this register matches the current refresh commands count.

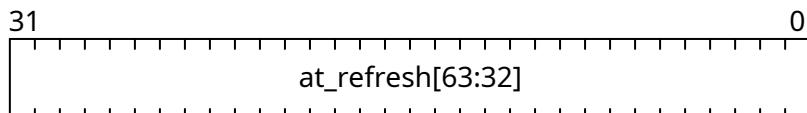


Fig. 19.61: DFI_SWITCH_AT_REFRESH1

DFI_SWITCH_AT_REFRESH0

Address: $0xf0004000 + 0xc = 0xf000400c$

Bits 0-31 of *DFI_SWITCH_AT_REFRESH*.

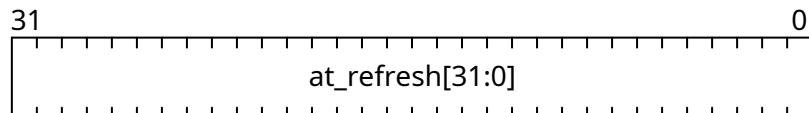


Fig. 19.62: DFI_SWITCH_AT_REFRESH0

DFI_SWITCH_REFRESH_UPDATE

Address: $0xf0004000 + 0x10 = 0xf0004010$

Force an update of the *refresh_count* CSR.



Fig. 19.63: DFI_SWITCH_REFRESH_UPDATE

19.2.10 PAYLOAD_EXECUTOR

Executes the DRAM payload from memory

Instruction decoder

All instructions are 32-bit. The format of most instructions is the same, except for the LOOP instruction, which has a constant TIMESLICE of 1.

NOOP with a TIMESLICE of 0 is a special case which is interpreted as STOP instruction. When this instruction is encountered execution gets finished immediately.

NOTE: TIMESLICE is the number of cycles the instruction will take. This means that instructions other than NOOP that use TIMESLICE=0 are illegal (although will silently be executed as having TIMESLICE=1).

NOTE2: LOOP instruction will *jump* COUNT times, meaning that the “code” inside the loop will effectively be executed COUNT+1 times.

Op codes:

Op	Value
NOOP	0b000
LOOP	0b111
ACT	0b100
PRE	0b101
REF	0b110
ZQC	0b001
READ	0b010

Instruction format:

LSB	MSB
dfi: OP_CODE TIMESLICE ADDRESS	
noop: OP_CODE TIMESLICE_NOOP	
loop: OP_CODE COUNT JUMP	
stop: <NOOP> 0	

Where ADDRESS depends on the DFI command and is one of:

LSB	MSB
RANK BANK COLUMN	
RANK BANK ROW	

Register Listing for PAYLOAD_EXECUTOR

Register	Address
PAYLOAD_EXECUTOR_START	0xf0004800
PAYLOAD_EXECUTOR_STATUS	0xf0004804
PAYLOAD_EXECUTOR_READ_COUNT	0xf0004808
PAYLOAD_EXECUTOR_EXEC_START1	0xf000480c
PAYLOAD_EXECUTOR_EXEC_START0	0xf0004810
PAYLOAD_EXECUTOR_EXEC_STOP1	0xf0004814
PAYLOAD_EXECUTOR_EXEC_STOP0	0xf0004818

PAYLOAD_EXECUTOR_START

Address: $0xf0004800 + 0x0 = 0xf0004800$

Writing to this register initializes payload execution



Fig. 19.64: PAYLOAD_EXECUTOR_START

PAYLOAD_EXECUTOR_STATUS

Address: $0xf0004800 + 0x4 = 0xf0004804$

Payload executor status register

Field	Name	Description
[0]	READY	Indicates that the executor is not running
[1]	OVERFLOW	Indicates the scratchpad memory address counter has overflowed due to the number of READ commands sent during execution

PAYLOAD_EXECUTOR_READ_COUNT

Address: $0xf0004800 + 0x8 = 0xf0004808$

Number of data from READ commands that is stored in the scratchpad memory

PAYLOAD_EXECUTOR_EXEC_START1

Address: $0xf0004800 + 0xc = 0xf000480c$

Bits 32-63 of PAYLOAD_EXECUTOR_EXEC_START. Number of cycles elapsed until the start of the payload execution.

PAYLOAD_EXECUTOR_EXEC_START0

Address: $0xf0004800 + 0x10 = 0xf0004810$

Bits 0-31 of PAYLOAD_EXECUTOR_EXEC_START.

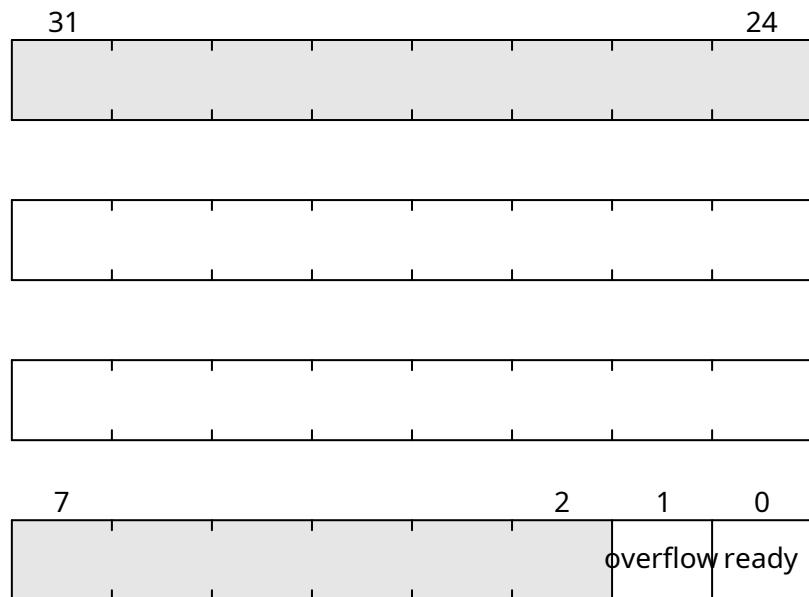


Fig. 19.65: PAYLOAD_EXECUTOR_STATUS

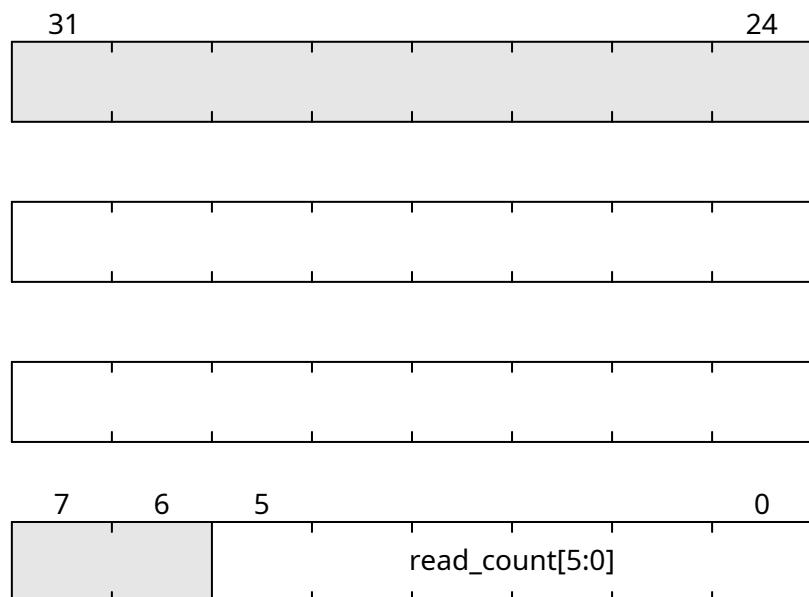


Fig. 19.66: PAYLOAD_EXECUTOR_READ_COUNT

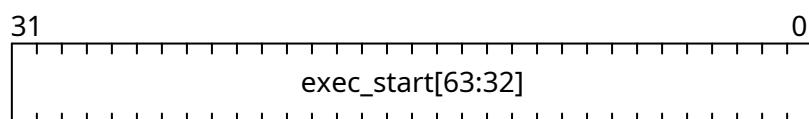


Fig. 19.67: PAYLOAD_EXECUTOR_EXEC_START1

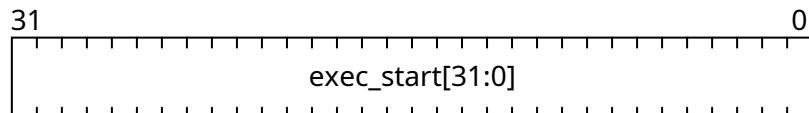


Fig. 19.68: PAYLOAD_EXECUTOR_EXEC_START0

PAYLOAD_EXECUTOR_EXEC_STOP1

Address: $0xf0004800 + 0x14 = 0xf0004814$

Bits 32-63 of PAYLOAD_EXECUTOR_EXEC_STOP. Number of cycles elapsed until the end of the payload execution.

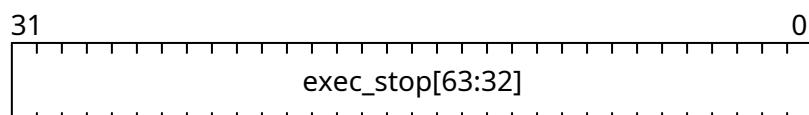


Fig. 19.69: PAYLOAD_EXECUTOR_EXEC_STOP1

PAYLOAD_EXECUTOR_EXEC_STOP0

Address: $0xf0004800 + 0x18 = 0xf0004818$

Bits 0-31 of PAYLOAD_EXECUTOR_EXEC_STOP.

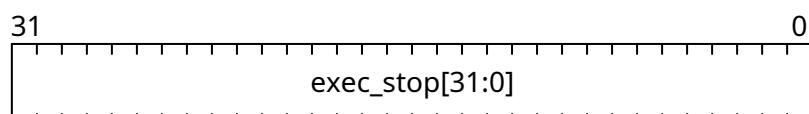


Fig. 19.70: PAYLOAD_EXECUTOR_EXEC_STOP0

19.2.11 CTRL

Register Listing for CTRL

Register	Address
<i>CTRL_RESET</i>	<i>0xf0005000</i>
<i>CTRL_SCRATCH</i>	<i>0xf0005004</i>
<i>CTRL_BUS_ERRORS</i>	<i>0xf0005008</i>

CTRL__RESET

Address: $0xf0005000 + 0x0 = 0xf0005000$

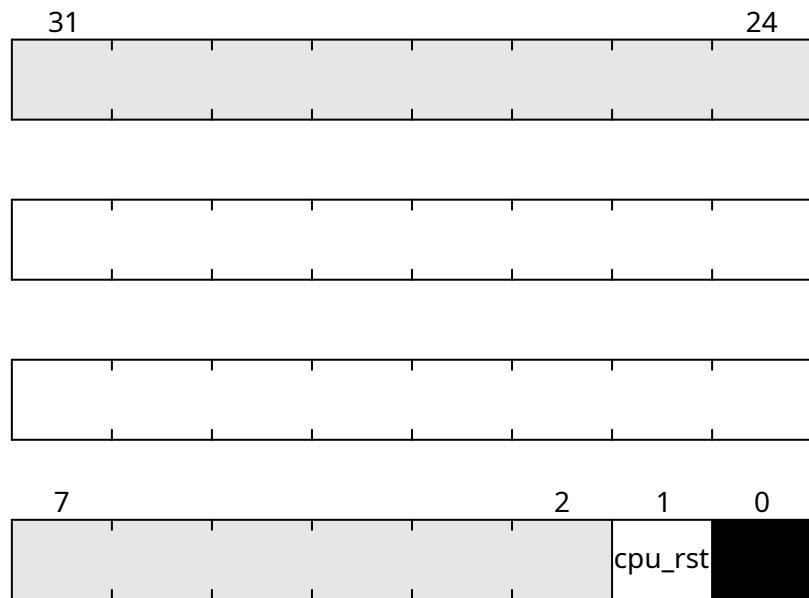


Fig. 19.71: CTRL_RESET

Field	Name	Description
[0]	SOC_RST	Write 1 to this register to reset the full SoC (Pulse Reset)
[1]	CPU_RST	Write 1 to this register to reset the CPU(s) of the SoC (Hold Reset)

CTRL_SCRATCH

Address: $0xf0005000 + 0x4 = 0xf0005004$

Use this register as a scratch space to verify that software read/write accesses to the Wishbone/CSR bus are working correctly. The initial reset value of 0x1234578 can be used to verify endianness.

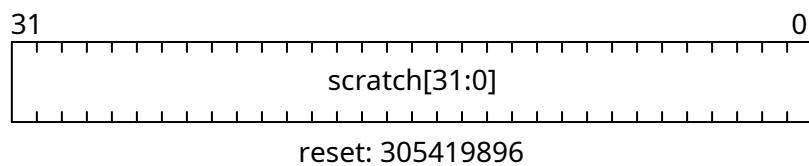


Fig. 19.72: CTRL_SCRATCH

CTRL_BUS_ERRORS

Address: $0xf0005000 + 0x8 = 0xf0005008$

Total number of Wishbone bus errors (timeouts) since start.

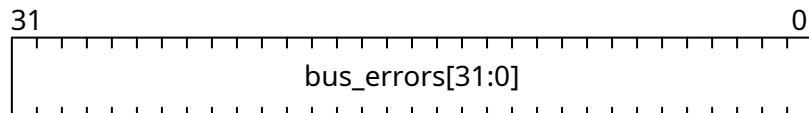


Fig. 19.73: CTRL_BUS_ERRORS

19.2.12 IDENTIFIER_MEM

Register Listing for IDENTIFIER_MEM

Register	Address
<i>IDENTIFIER_MEM</i>	<i>0xf0005800</i>

IDENTIFIER_MEM

Address: $0xf0005800 + 0x0 = 0xf0005800$

8 x 110-bit memory

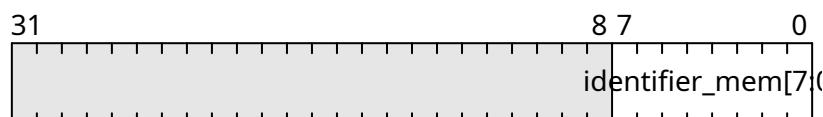


Fig. 19.74: IDENTIFIER_MEM

19.2.13 SDRAM

Register Listing for SDRAM

Register	Address
<i>SDRAM_DFII_CONTROL</i>	<i>0xf0006000</i>
<i>SDRAM_DFII_PIO_COMMAND</i>	<i>0xf0006004</i>
<i>SDRAM_DFII_PIO_COMMAND_ISSUE</i>	<i>0xf0006008</i>
<i>SDRAM_DFII_PIO_ADDRESS</i>	<i>0xf000600c</i>
<i>SDRAM_DFII_PIO_BADDRESS</i>	<i>0xf0006010</i>
<i>SDRAM_DFII_PIO_WRDATA</i>	<i>0xf0006014</i>
<i>SDRAM_DFII_PIO_RDDATA</i>	<i>0xf0006018</i>
<i>SDRAM_DFII_PI1_COMMAND</i>	<i>0xf000601c</i>
<i>SDRAM_DFII_PI1_COMMAND_ISSUE</i>	<i>0xf0006020</i>
<i>SDRAM_DFII_PI1_ADDRESS</i>	<i>0xf0006024</i>
<i>SDRAM_DFII_PI1_BADDRESS</i>	<i>0xf0006028</i>
<i>SDRAM_DFII_PI1_WRDATA</i>	<i>0xf000602c</i>
<i>SDRAM_DFII_PI1_RDDATA</i>	<i>0xf0006030</i>
<i>SDRAM_DFII_PI2_COMMAND</i>	<i>0xf0006034</i>
<i>SDRAM_DFII_PI2_COMMAND_ISSUE</i>	<i>0xf0006038</i>

continues on next page

Table 19.1 – continued from previous page

Register	Address
<i>SDRAM_DFII_PI2_ADDRESS</i>	0xf000603c
<i>SDRAM_DFII_PI2_BADDRESS</i>	0xf0006040
<i>SDRAM_DFII_PI2_WRDATA</i>	0xf0006044
<i>SDRAM_DFII_PI2_RDDATA</i>	0xf0006048
<i>SDRAM_DFII_PI3_COMMAND</i>	0xf000604c
<i>SDRAM_DFII_PI3_COMMAND_ISSUE</i>	0xf0006050
<i>SDRAM_DFII_PI3_ADDRESS</i>	0xf0006054
<i>SDRAM_DFII_PI3_BADDRESS</i>	0xf0006058
<i>SDRAM_DFII_PI3_WRDATA</i>	0xf000605c
<i>SDRAM_DFII_PI3_RDDATA</i>	0xf0006060
<i>SDRAM_CONTROLLER_TRP</i>	0xf0006064
<i>SDRAM_CONTROLLER_TRCD</i>	0xf0006068
<i>SDRAM_CONTROLLER_TWR</i>	0xf000606c
<i>SDRAM_CONTROLLER_TWTR</i>	0xf0006070
<i>SDRAM_CONTROLLER_TREFI</i>	0xf0006074
<i>SDRAM_CONTROLLER_TRFC</i>	0xf0006078
<i>SDRAM_CONTROLLER_TFAW</i>	0xf000607c
<i>SDRAM_CONTROLLER_TCCD</i>	0xf0006080
<i>SDRAM_CONTROLLER_TCCD_WR</i>	0xf0006084
<i>SDRAM_CONTROLLER_TRTP</i>	0xf0006088
<i>SDRAM_CONTROLLER_TRRD</i>	0xf000608c
<i>SDRAM_CONTROLLER_TRC</i>	0xf0006090
<i>SDRAM_CONTROLLER_TRAS</i>	0xf0006094
<i>SDRAM_CONTROLLER_TZQCS</i>	0xf0006098
<i>SDRAM_CONTROLLER_LAST_ADDR_0</i>	0xf000609c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0</i>	0xf00060a0
<i>SDRAM_CONTROLLER_LAST_ADDR_1</i>	0xf00060a4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1</i>	0xf00060a8
<i>SDRAM_CONTROLLER_LAST_ADDR_2</i>	0xf00060ac
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2</i>	0xf00060b0
<i>SDRAM_CONTROLLER_LAST_ADDR_3</i>	0xf00060b4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3</i>	0xf00060b8
<i>SDRAM_CONTROLLER_LAST_ADDR_4</i>	0xf00060bc
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4</i>	0xf00060c0
<i>SDRAM_CONTROLLER_LAST_ADDR_5</i>	0xf00060c4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5</i>	0xf00060c8
<i>SDRAM_CONTROLLER_LAST_ADDR_6</i>	0xf00060cc
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6</i>	0xf00060d0
<i>SDRAM_CONTROLLER_LAST_ADDR_7</i>	0xf00060d4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7</i>	0xf00060d8

SDRAM_DFII_CONTROL

Address: 0xf0006000 + 0x0 = 0xf0006000

Control DFI signals common to all phases

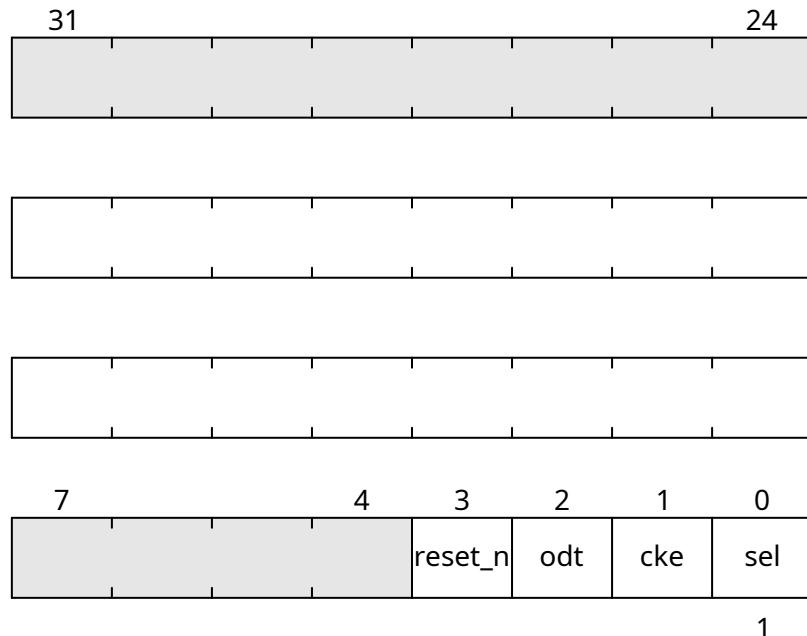


Fig. 19.75: SDRAM_DFII_CONTROL

Field	Name	Description						
[0]	SEL	<table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0b0</td><td>Software control. (CPU)</td></tr> <tr> <td>0b1</td><td>Hardware control (default).</td></tr> </tbody> </table>	Value	Description	0b0	Software control. (CPU)	0b1	Hardware control (default).
Value	Description							
0b0	Software control. (CPU)							
0b1	Hardware control (default).							
[1]	CKE	DFI clock enable bus						
[2]	ODT	DFI on-die termination bus						
[3]	RESET_N	DFI clock reset bus						

SDRAM_DFII_PIO_COMMAND

Address: $0xf0006000 + 0x4 = 0xf0006004$

Control DFI signals on a single phase

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

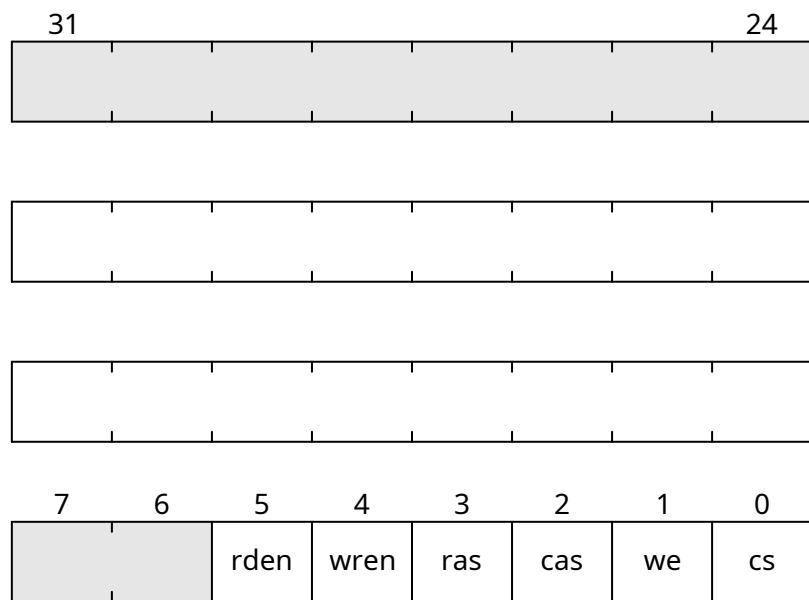


Fig. 19.76: SDRAM DFII PIO COMMAND

SDRAM_DFIIPIO_COMMAND_ISSUE

Address: $0xf0006000 + 0x8 = 0xf0006008$

SDRAM_DFIIPIO_ADDRESS

Address: $0xf0006000 + 0xc = 0xf000600c$

DFI address bus

SDRAM DFII PIO BADDRESS

Address: $0xf0006000 + 0x10 = 0xf0006010$

DFI bank address bus

SDRAM DFII PIO WRDATA

Address: $0xf0006000 + 0x14 = 0xf0006014$

DFI write data bus

SDRAM DFII PIO RDDATA

Address: $0xf0006000 + 0x18 \equiv 0xf0006018$

DFI read data bus

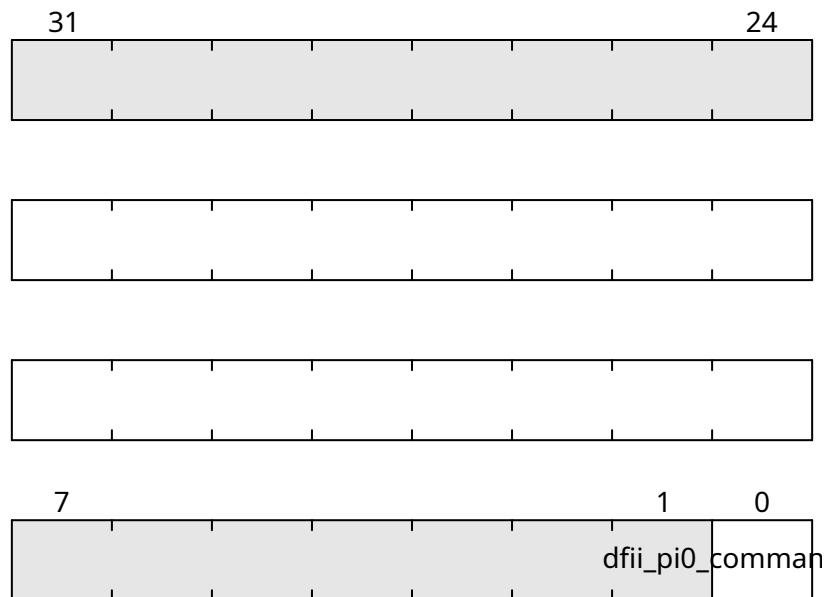


Fig. 19.77: SDRAM_DFII_PI0_COMMAND_ISSUE

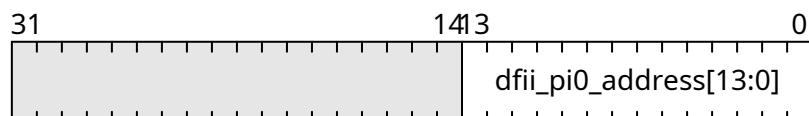


Fig. 19.78: SDRAM_DFII_PI0_ADDRESS

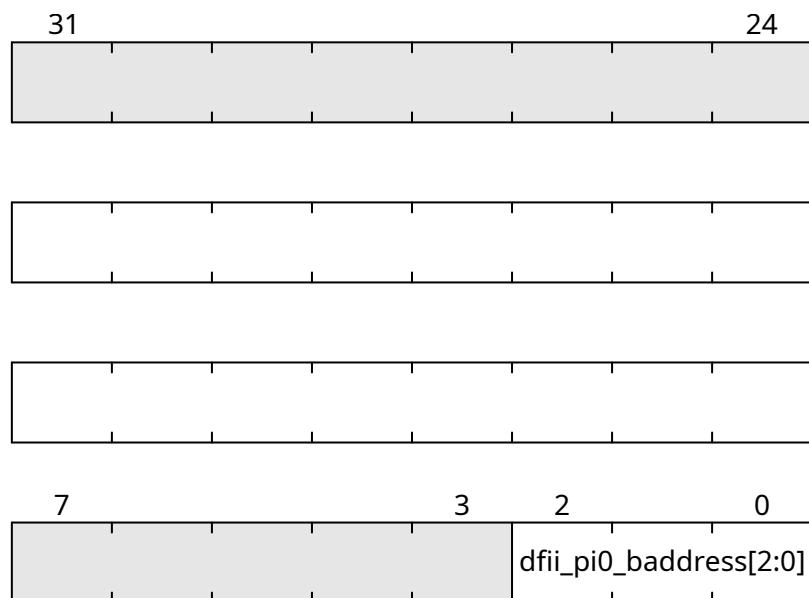


Fig. 19.79: SDRAM_DFII_PI0_BADDRESS

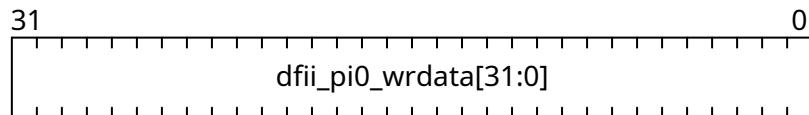


Fig. 19.80: SDRAM_DFII_PI0_WRDATA

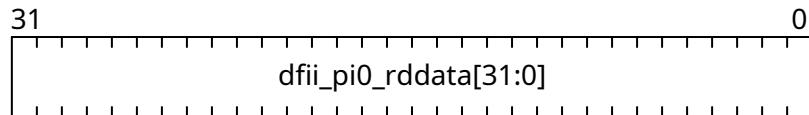


Fig. 19.81: SDRAM_DFII_PI0_RDDATA

SDRAM_DFII_PI1_COMMAND

Address: 0xf0006000 + 0x1c = 0xf000601c

Control DFI signals on a single phase

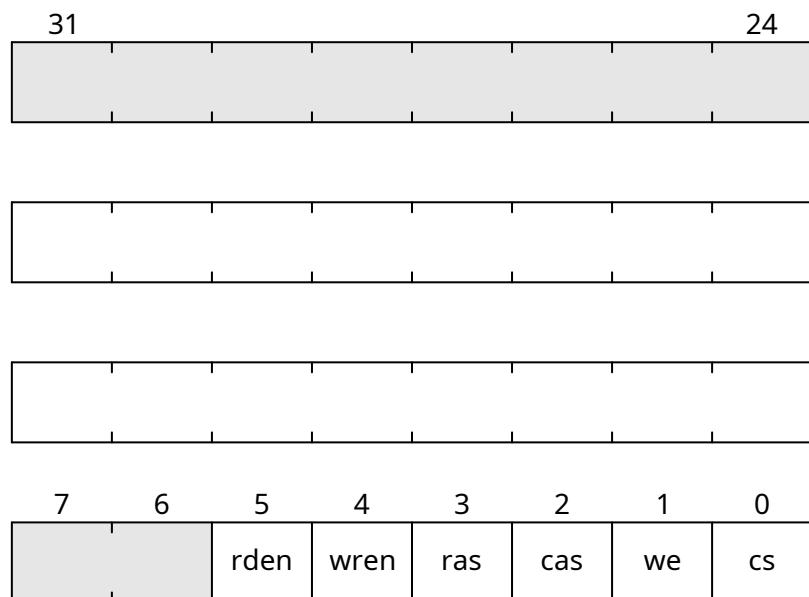


Fig. 19.82: SDRAM_DFII_PI1_COMMAND

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFII_PI1_COMMAND_ISSUE

Address: $0xf0006000 + 0x20 = 0xf0006020$

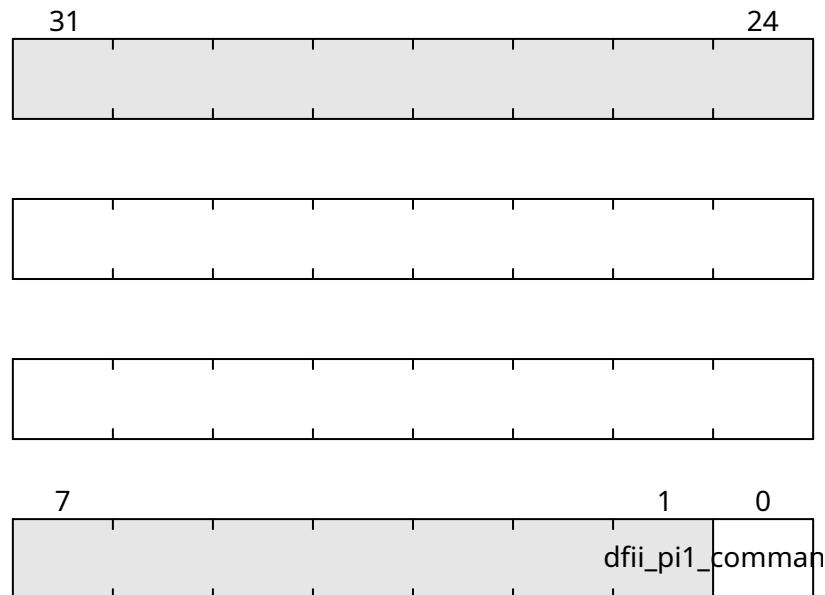


Fig. 19.83: SDRAM_DFII_PI1_COMMAND_ISSUE

SDRAM_DFII_PI1_ADDRESS

Address: $0xf0006000 + 0x24 = 0xf0006024$

DFI address bus

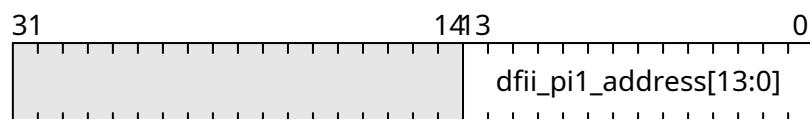


Fig. 19.84: SDRAM_DFII_PI1_ADDRESS

SDRAM_DFII_PI1_BADDRESS

Address: $0xf0006000 + 0x28 = 0xf0006028$

DFI bank address bus

SDRAM_DFII_PI1_WRDATA

Address: $0xf0006000 + 0x2c = 0xf000602c$

DFI write data bus

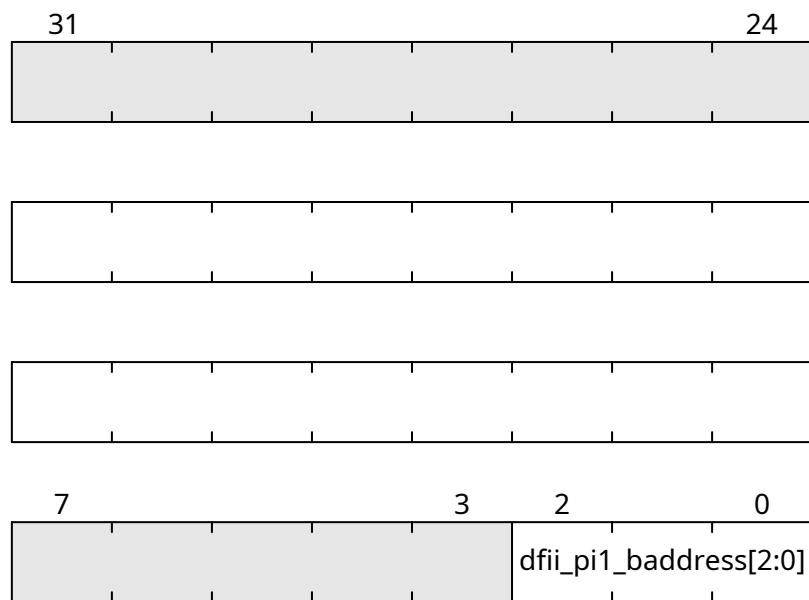


Fig. 19.85: `SDRAM_DFII_PI1_BADDRESS`

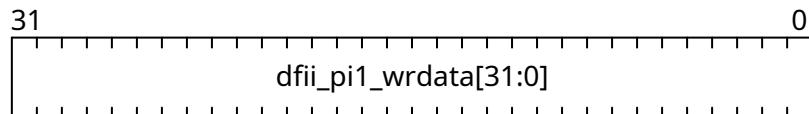


Fig. 19.86: `SDRAM_DFII_PI1_WRDATA`

SDRAM_DFII_PI1_RDDATA

Address: $0xf0006000 + 0x30 = 0xf0006030$

DFI read data bus

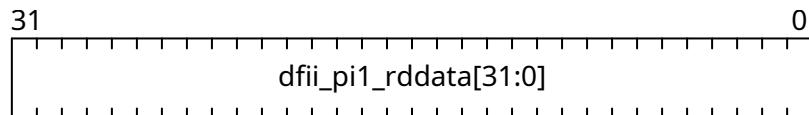


Fig. 19.87: SDRAM_DFII_PI1_RDDATA

SDRAM_DFII_PI2_COMMAND

Address: $0xf0006000 + 0x34 = 0xf0006034$

Control DFI signals on a single phase

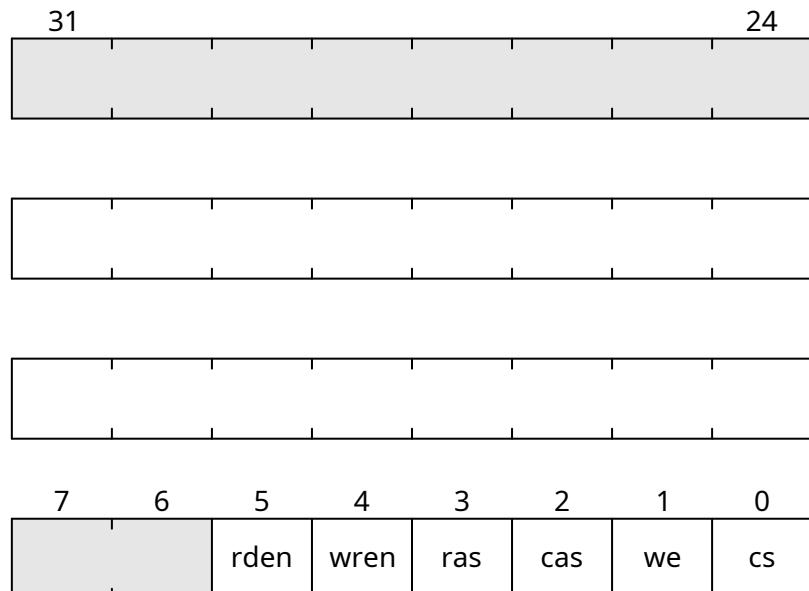


Fig. 19.88: SDRAM_DFII_PI2_COMMAND

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFII_PI2_COMMAND_ISSUE

Address: $0xf0006000 + 0x38 = 0xf0006038$

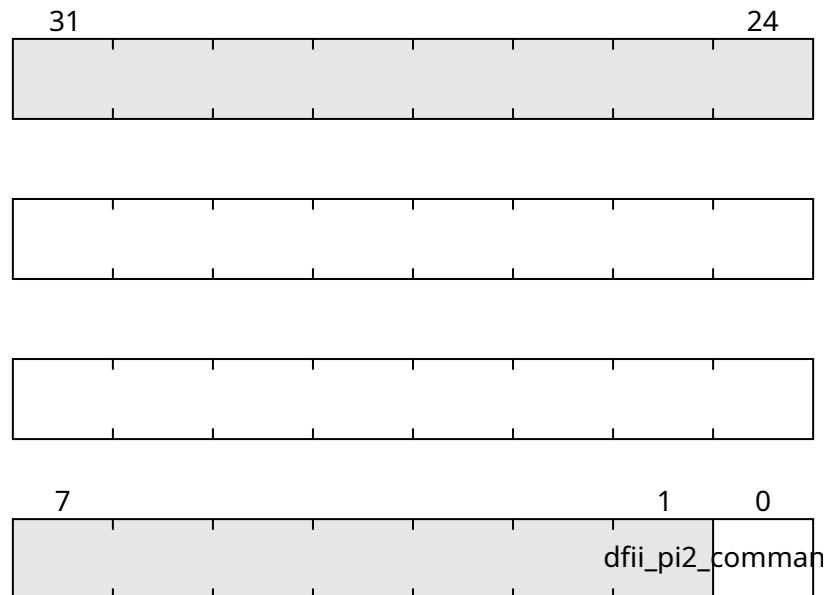


Fig. 19.89: SDRAM_DFII_PI2_COMMAND_ISSUE

SDRAM_DFII_PI2_ADDRESS

Address: $0xf0006000 + 0x3c = 0xf000603c$

DFI address bus

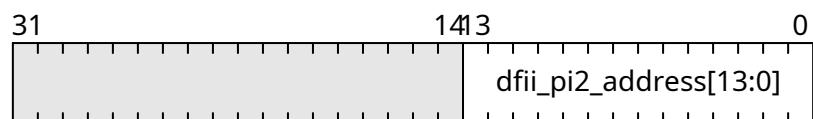


Fig. 19.90: SDRAM_DFII_PI2_ADDRESS

SDRAM_DFII_PI2_BADDRESS

Address: $0xf0006000 + 0x40 = 0xf0006040$

DFI bank address bus

SDRAM_DFII_PI2_WRDATA

Address: $0xf0006000 + 0x44 = 0xf0006044$

DFI write data bus



Fig. 19.91: `SDRAM_DFII_PI2_BADDRESS`

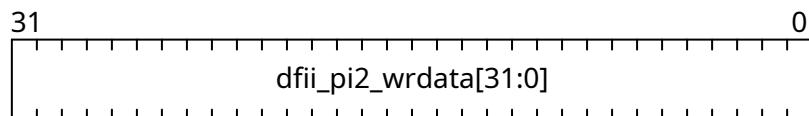


Fig. 19.92: `SDRAM_DFII_PI2_WRDATA`

SDRAM_DFII_PI2_RDDATA

Address: $0xf0006000 + 0x48 = 0xf0006048$

DFI read data bus

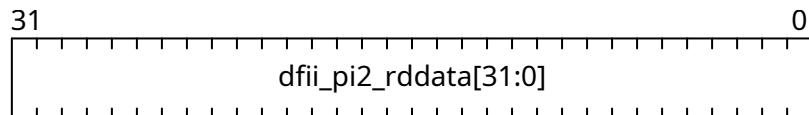


Fig. 19.93: SDRAM_DFII_PI2_RDDATA

SDRAM_DFII_PI3_COMMAND

Address: $0xf0006000 + 0x4c = 0xf000604c$

Control DFI signals on a single phase

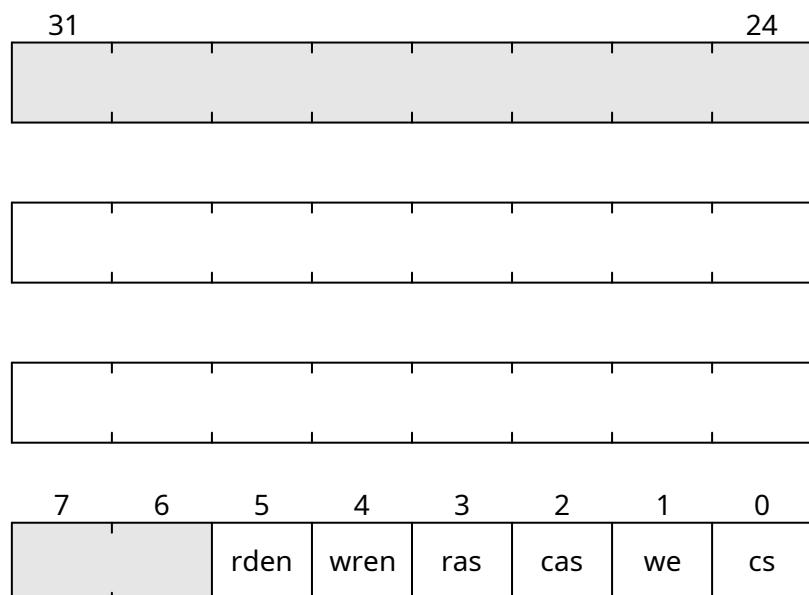


Fig. 19.94: SDRAM_DFII_PI3_COMMAND

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFII_PI3_COMMAND_ISSUE

Address: $0xf0006000 + 0x50 = 0xf0006050$

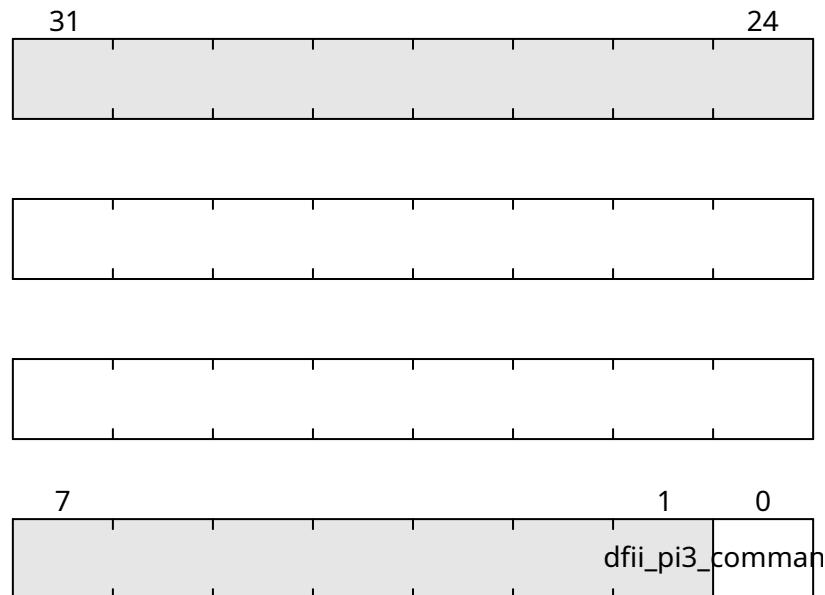


Fig. 19.95: SDRAM_DFII_PI3_COMMAND_ISSUE

SDRAM_DFII_PI3_ADDRESS

Address: $0xf0006000 + 0x54 = 0xf0006054$

DFI address bus

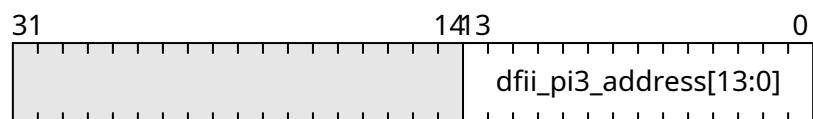


Fig. 19.96: SDRAM_DFII_PI3_ADDRESS

SDRAM_DFII_PI3_BADDRESS

Address: $0xf0006000 + 0x58 = 0xf0006058$

DFI bank address bus

SDRAM_DFII_PI3_WRDATA

Address: $0xf0006000 + 0x5c = 0xf000605c$

DFI write data bus

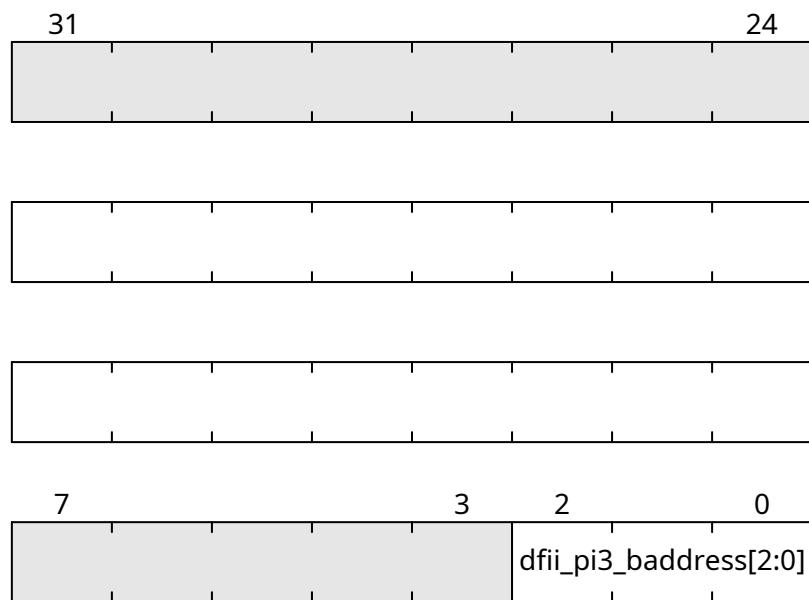


Fig. 19.97: `SDRAM_DFII_PI3_BADDRESS`

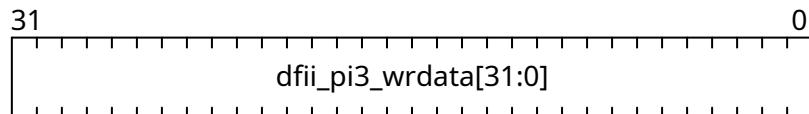


Fig. 19.98: `SDRAM_DFII_PI3_WRDATA`

SDRAM_DFII_PI3_RDDATA

Address: $0xf0006000 + 0x60 = 0xf0006060$

DFI read data bus

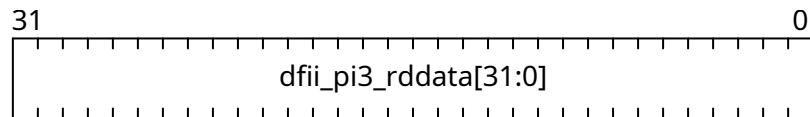


Fig. 19.99: SDRAM_DFII_PI3_RDDATA

SDRAM_CONTROLLER_TRP

Address: $0xf0006000 + 0x64 = 0xf0006064$

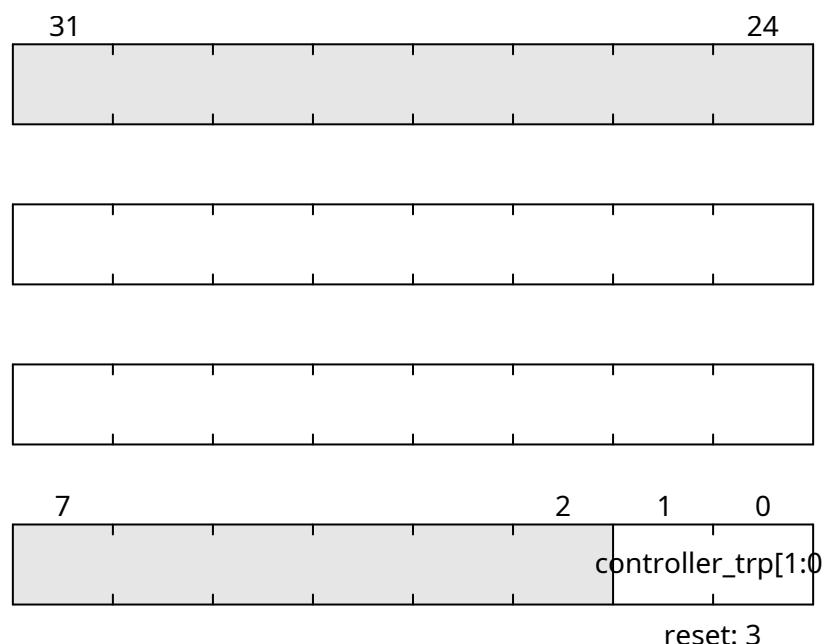


Fig. 19.100: SDRAM_CONTROLLER_TRP

SDRAM_CONTROLLER_TRCD

Address: $0xf0006000 + 0x68 = 0xf0006068$

SDRAM_CONTROLLER_TWR

Address: $0xf0006000 + 0x6c = 0xf000606c$

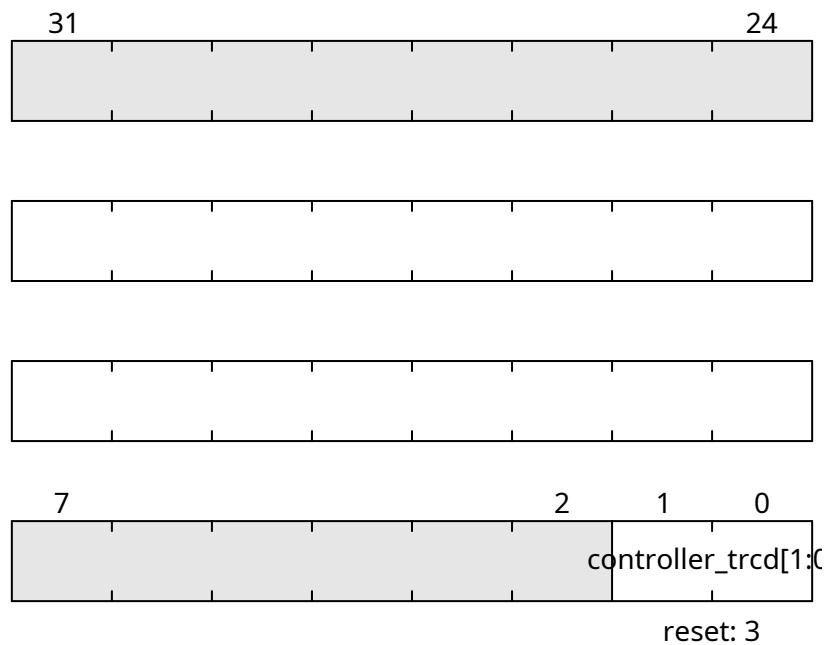


Fig. 19.101: SDRAM_CONTROLLER_TRCD

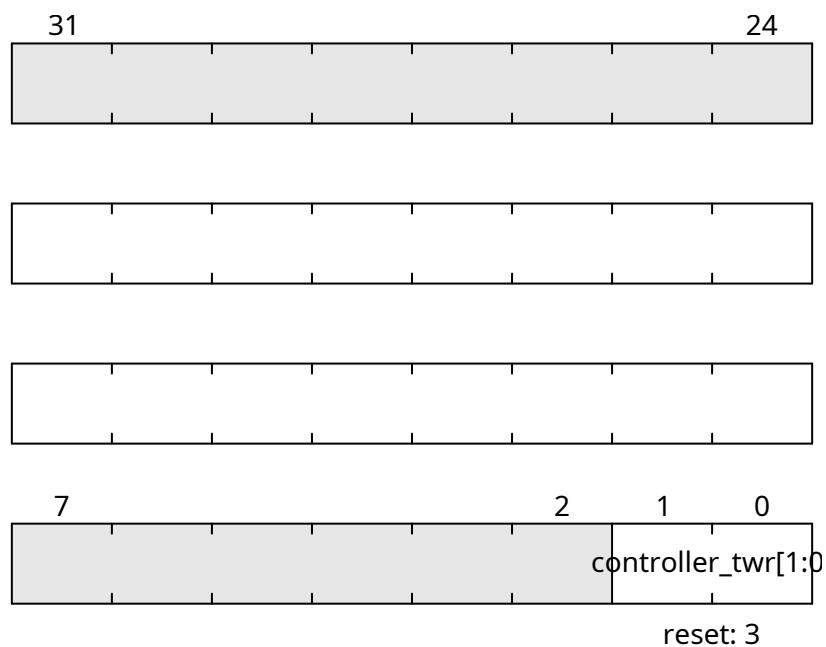


Fig. 19.102: SDRAM_CONTROLLER_TWR

SDRAM_CONTROLLER_TWTR

Address: $0xf0006000 + 0x70 = 0xf0006070$

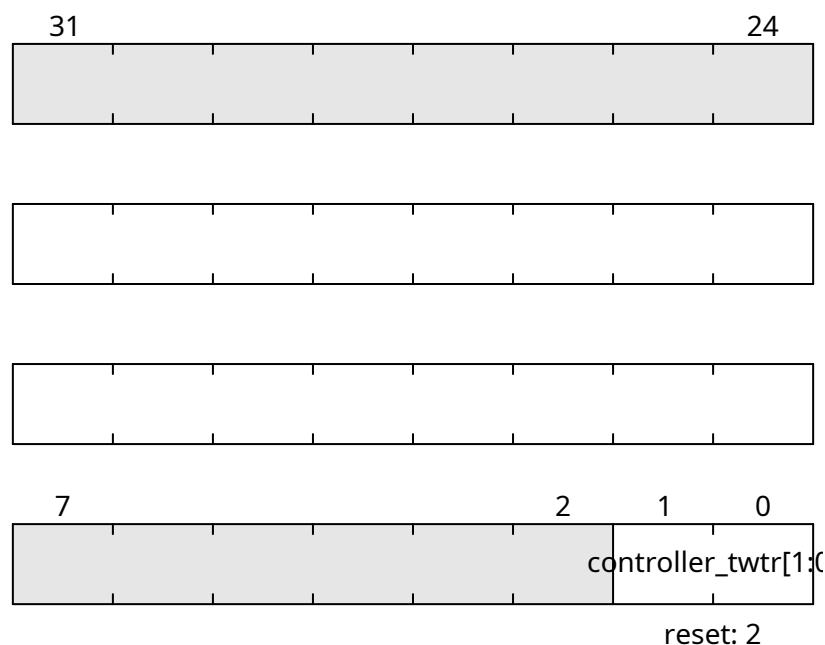


Fig. 19.103: SDRAM_CONTROLLER_TWTR

SDRAM_CONTROLLER_TREFI

Address: $0xf0006000 + 0x74 = 0xf0006074$

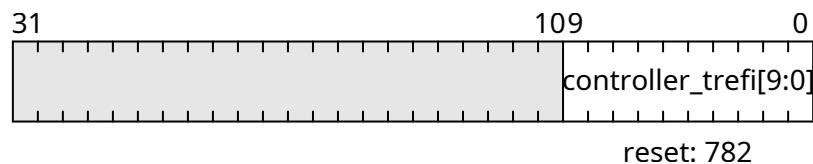


Fig. 19.104: SDRAM_CONTROLLER_TREFI

SDRAM_CONTROLLER_TRFC

Address: $0xf0006000 + 0x78 = 0xf0006078$

SDRAM_CONTROLLER_TFAW

Address: $0xf0006000 + 0x7c = 0xf000607c$

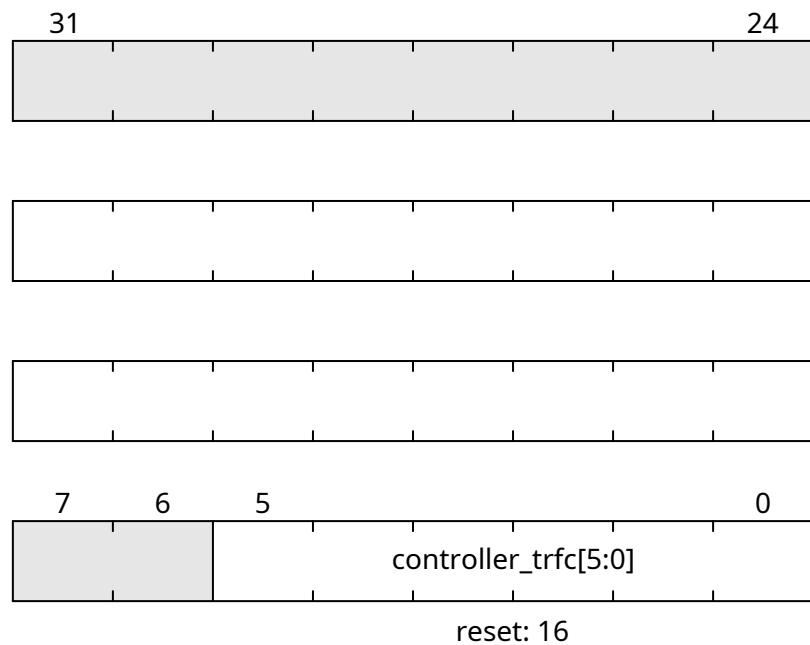


Fig. 19.105: SDRAM_CONTROLLER_TRFC

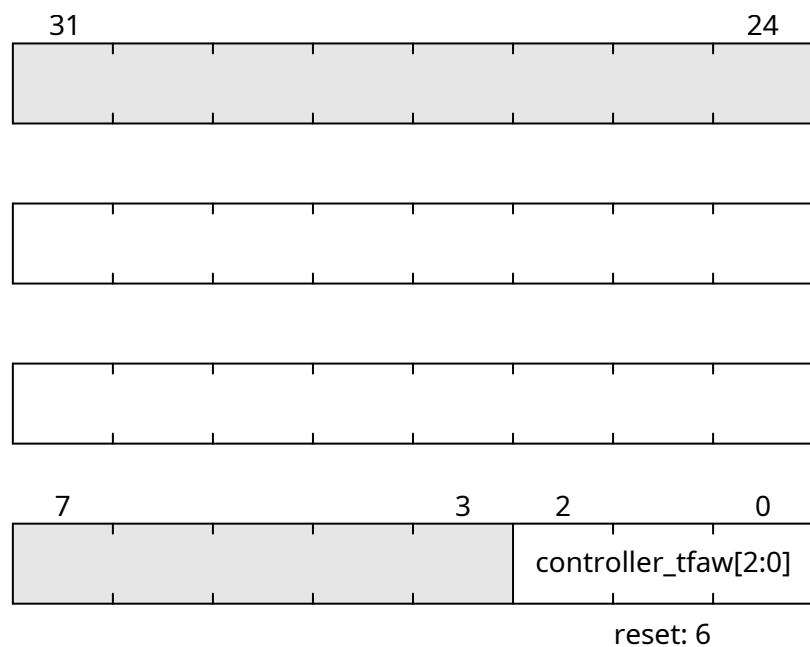


Fig. 19.106: SDRAM_CONTROLLER_TFAW

SDRAM_CONTROLLER_TCCD

Address: $0xf0006000 + 0x80 = 0xf0006080$

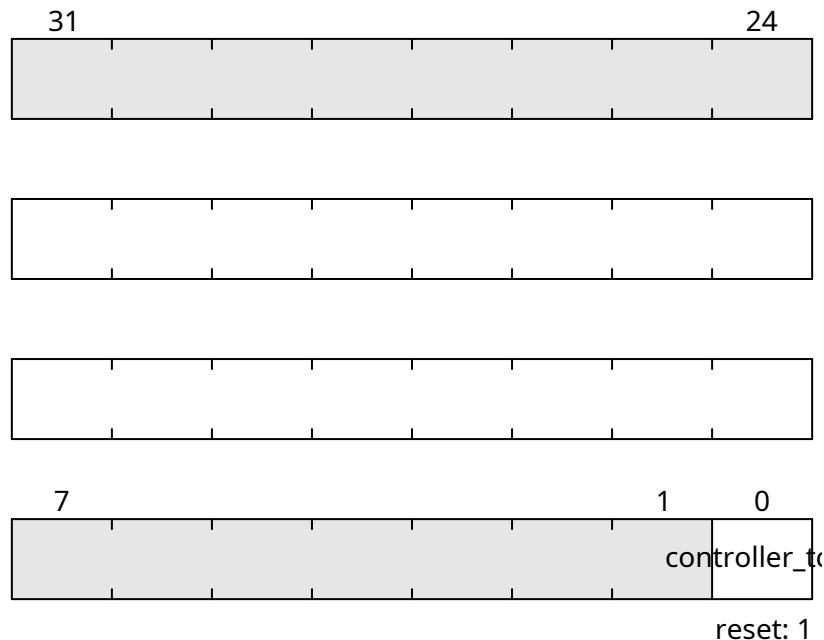


Fig. 19.107: SDRAM_CONTROLLER_TCCD

SDRAM_CONTROLLER_TCCD_WR

Address: $0xf0006000 + 0x84 = 0xf0006084$

SDRAM_CONTROLLER_TRTP

Address: $0xf0006000 + 0x88 = 0xf0006088$

SDRAM_CONTROLLER_TRRD

Address: $0xf0006000 + 0x8c = 0xf000608c$

SDRAM_CONTROLLER_TRC

Address: $0xf0006000 + 0x90 = 0xf0006090$

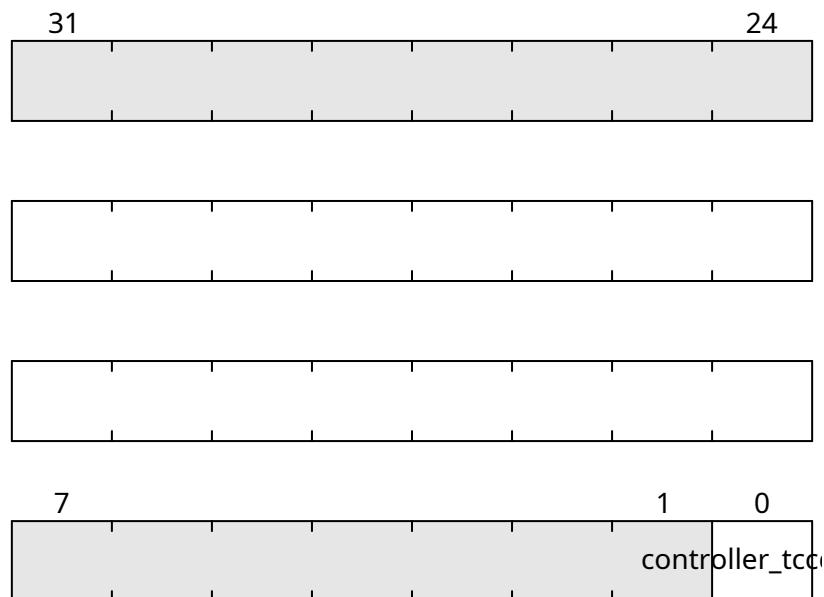


Fig. 19.108: SDRAM_CONTROLLER_TCCD_WR

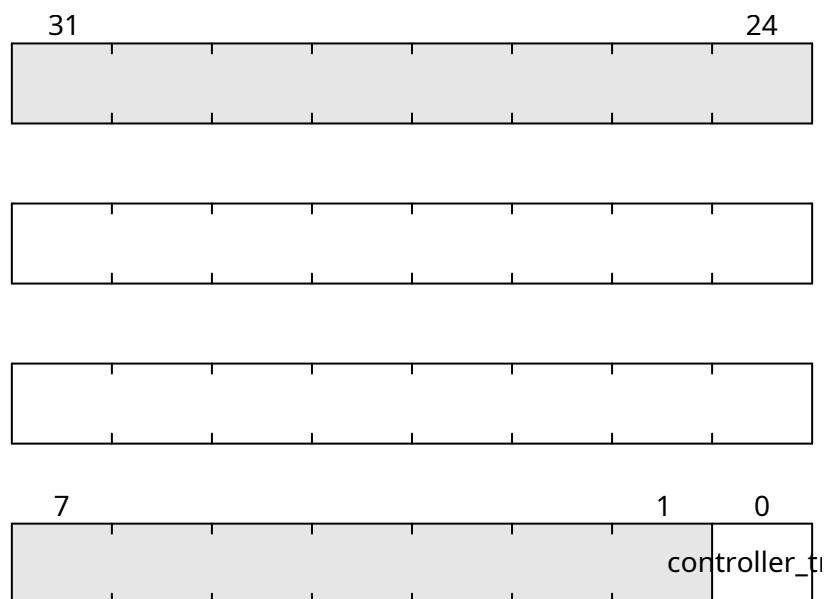


Fig. 19.109: SDRAM_CONTROLLER_TRTP

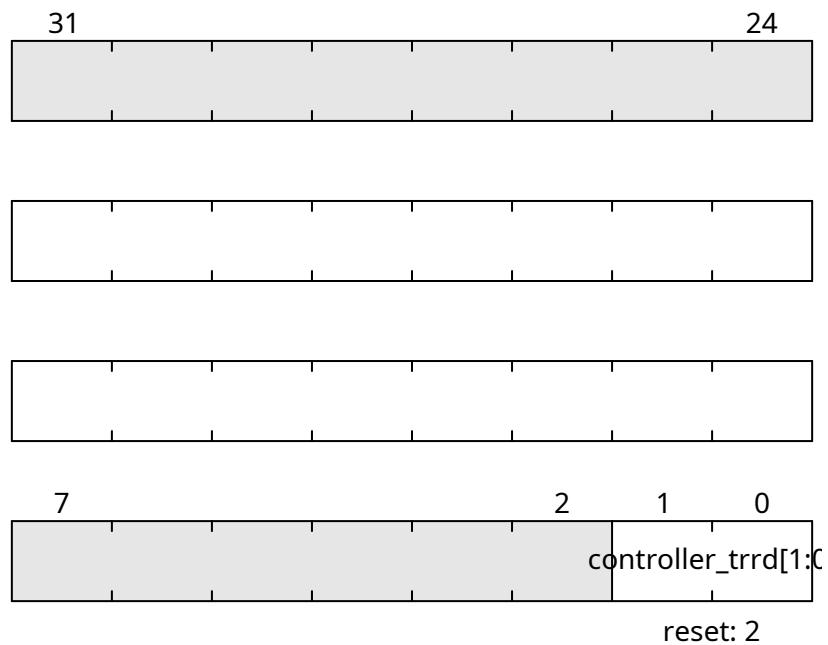


Fig. 19.110: SDRAM_CONTROLLER_TRRD

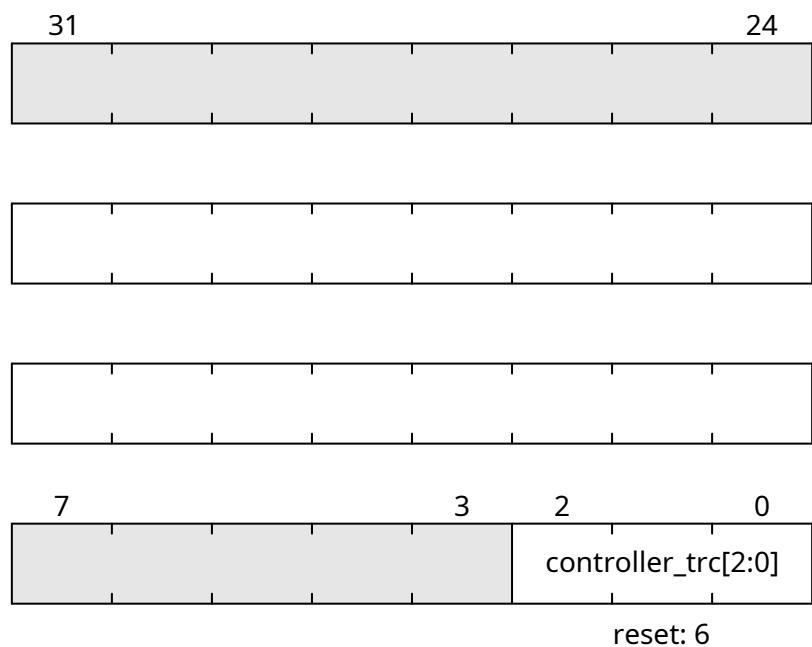


Fig. 19.111: SDRAM_CONTROLLER_TRC

SDRAM_CONTROLLER_TRAS

Address: $0xf0006000 + 0x94 = 0xf0006094$

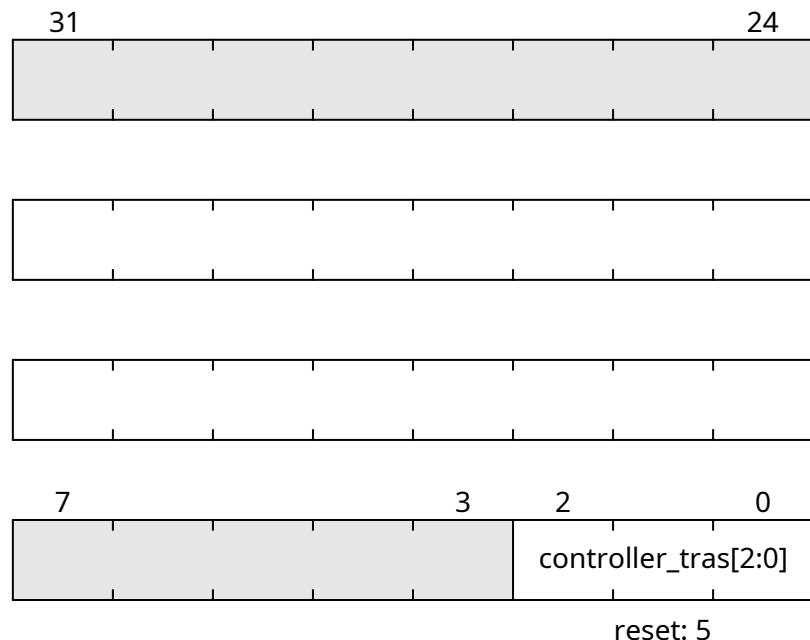


Fig. 19.112: SDRAM_CONTROLLER_TRAS

SDRAM_CONTROLLER_TZQCS

Address: $0xf0006000 + 0x98 = 0xf0006098$

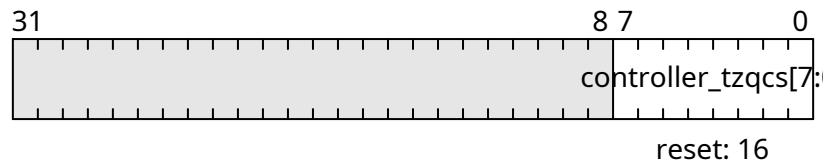


Fig. 19.113: SDRAM_CONTROLLER_TZQCS

SDRAM_CONTROLLER_LAST_ADDR_0

Address: $0xf0006000 + 0x9c = 0xf000609c$

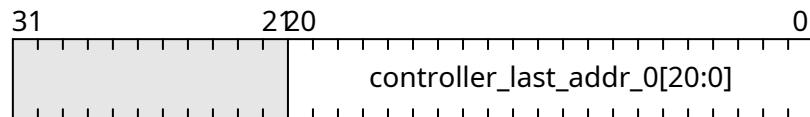


Fig. 19.114: SDRAM_CONTROLLER_LAST_ADDR_0

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0

Address: $0xf0006000 + 0xa0 = 0xf00060a0$

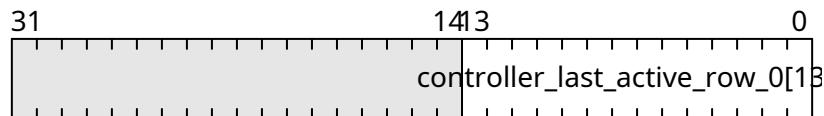


Fig. 19.115: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0

SDRAM_CONTROLLER_LAST_ADDR_1

Address: $0xf0006000 + 0xa4 = 0xf00060a4$

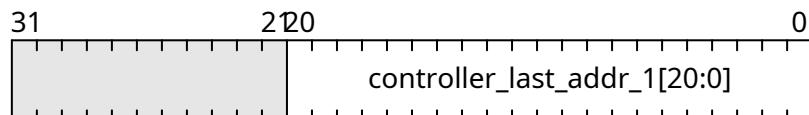


Fig. 19.116: SDRAM_CONTROLLER_LAST_ADDR_1

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1

Address: $0xf0006000 + 0xa8 = 0xf00060a8$

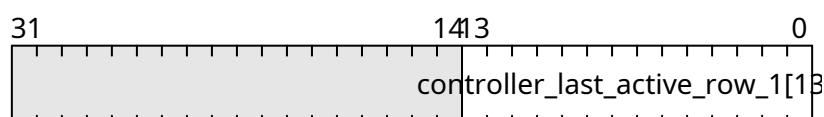


Fig. 19.117: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1

SDRAM_CONTROLLER_LAST_ADDR_2

Address: $0xf0006000 + 0xac = 0xf00060ac$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2

Address: $0xf0006000 + 0xb0 = 0xf00060b0$

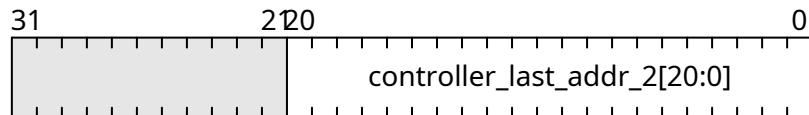


Fig. 19.118: SDRAM_CONTROLLER_LAST_ADDR_2

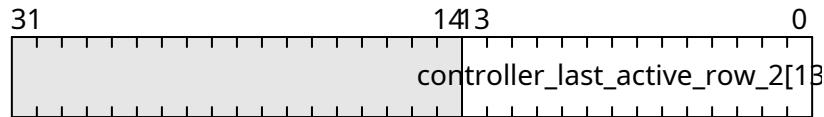


Fig. 19.119: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2

SDRAM_CONTROLLER_LAST_ADDR_3

Address: $0xf0006000 + 0xb4 = 0xf00060b4$

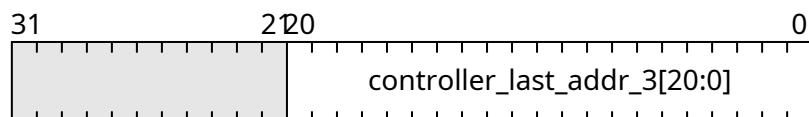


Fig. 19.120: SDRAM_CONTROLLER_LAST_ADDR_3

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3

Address: $0xf0006000 + 0xb8 = 0xf00060b8$

SDRAM_CONTROLLER_LAST_ADDR_4

Address: $0xf0006000 + 0xbc = 0xf00060bc$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4

Address: $0xf0006000 + 0xc0 = 0xf00060c0$

SDRAM_CONTROLLER_LAST_ADDR_5

Address: $0xf0006000 + 0xc4 = 0xf00060c4$

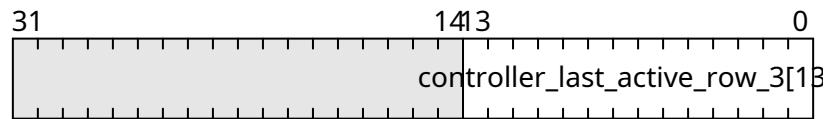


Fig. 19.121: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3

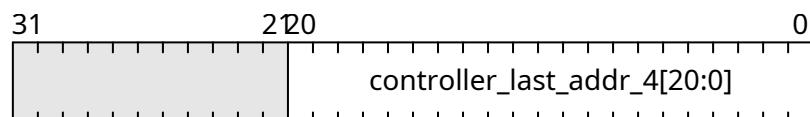


Fig. 19.122: SDRAM_CONTROLLER_LAST_ADDR_4

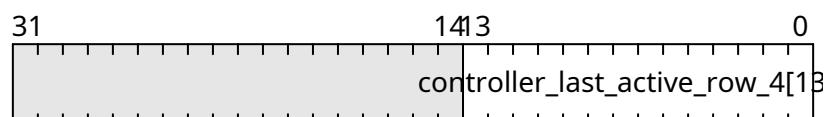


Fig. 19.123: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4

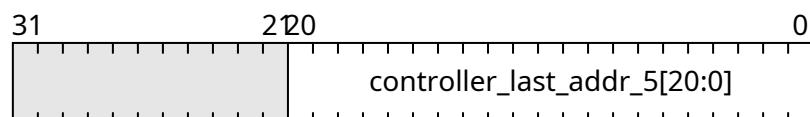


Fig. 19.124: SDRAM_CONTROLLER_LAST_ADDR_5

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5

Address: $0xf0006000 + 0xc8 = 0xf00060c8$

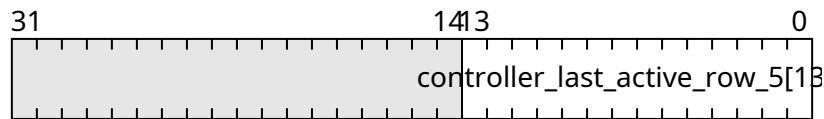


Fig. 19.125: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5

SDRAM_CONTROLLER_LAST_ADDR_6

Address: $0xf0006000 + 0xcc = 0xf00060cc$

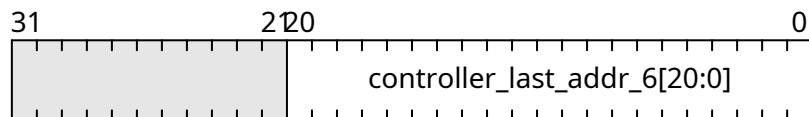


Fig. 19.126: SDRAM_CONTROLLER_LAST_ADDR_6

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6

Address: $0xf0006000 + 0xd0 = 0xf00060d0$

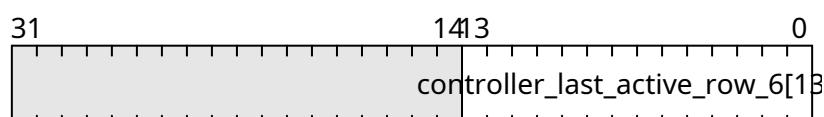


Fig. 19.127: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6

SDRAM_CONTROLLER_LAST_ADDR_7

Address: $0xf0006000 + 0xd4 = 0xf00060d4$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7

Address: $0xf0006000 + 0xd8 = 0xf00060d8$

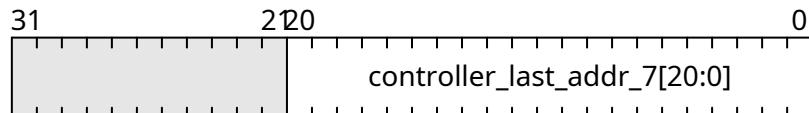


Fig. 19.128: SDRAM_CONTROLLER_LAST_ADDR_7

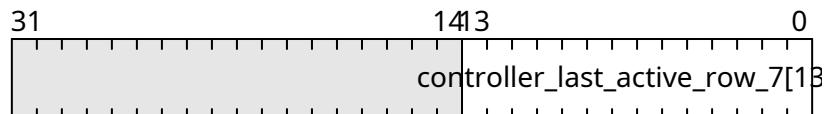


Fig. 19.129: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7

19.2.14 SDRAM_CHECKER

Register Listing for SDRAM_CHECKER

Register	Address
<i>SDRAM_CHECKER_RESET</i>	0xf0006800
<i>SDRAM_CHECKER_START</i>	0xf0006804
<i>SDRAM_CHECKER_DONE</i>	0xf0006808
<i>SDRAM_CHECKER_BASE</i>	0xf000680c
<i>SDRAM_CHECKER_END</i>	0xf0006810
<i>SDRAM_CHECKER_LENGTH</i>	0xf0006814
<i>SDRAM_CHECKER_RANDOM</i>	0xf0006818
<i>SDRAM_CHECKER_TICKS</i>	0xf000681c
<i>SDRAM_CHECKER_ERRORS</i>	0xf0006820

SDRAM_CHECKER_RESET

Address: $0xf0006800 + 0x0 = 0xf0006800$

SDRAM_CHECKER_START

Address: $0xf0006800 + 0x4 = 0xf0006804$

SDRAM_CHECKER_DONE

Address: $0xf0006800 + 0x8 = 0xf0006808$



Fig. 19.130: SDRAM_CHECKER_RESET



Fig. 19.131: SDRAM_CHECKER_START



Fig. 19.132: SDRAM_CHECKER_DONE

SDRAM_CHECKER_BASE

Address: $0xf0006800 + 0xc = 0xf000680c$

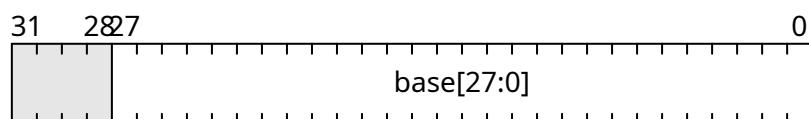


Fig. 19.133: SDRAM_CHECKER_BASE

SDRAM_CHECKER_END

Address: $0xf0006800 + 0x10 = 0xf0006810$

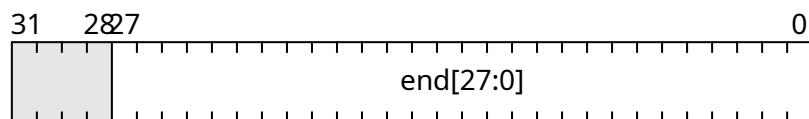


Fig. 19.134: SDRAM_CHECKER_END

SDRAM_CHECKER_LENGTH

Address: $0xf0006800 + 0x14 = 0xf0006814$

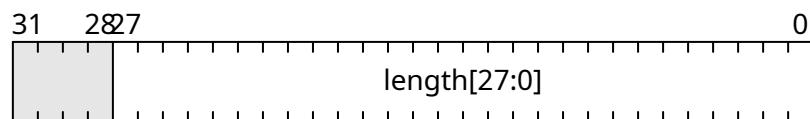


Fig. 19.135: SDRAM_CHECKER_LENGTH

SDRAM_CHECKER_RANDOM

Address: $0xf0006800 + 0x18 = 0xf0006818$

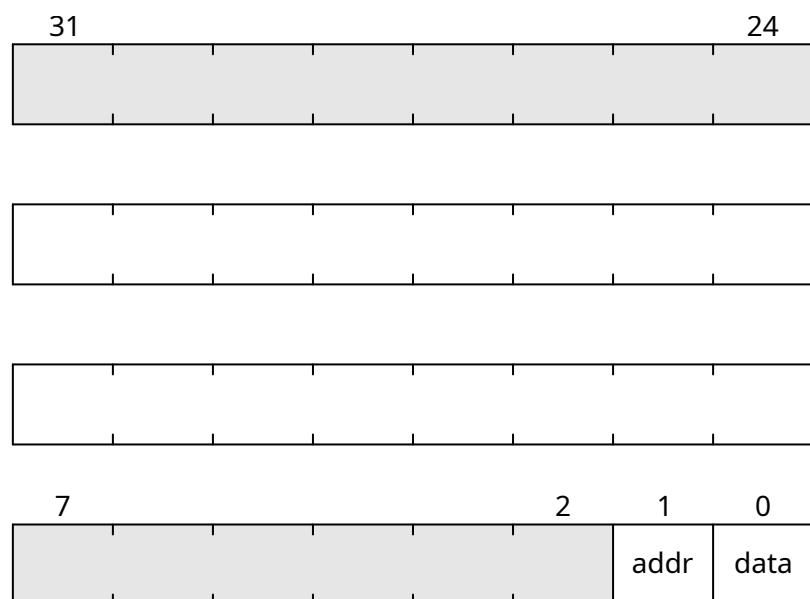


Fig. 19.136: SDRAM_CHECKER_RANDOM

Field	Name	Description

SDRAM_CHECKER_TICKS

Address: $0xf0006800 + 0x1c = 0xf000681c$

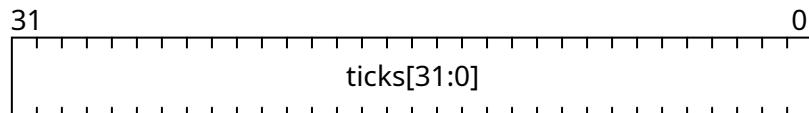


Fig. 19.137: SDRAM_CHECKER_TICKS

SDRAM_CHECKER_ERRORS

Address: $0xf0006800 + 0x20 = 0xf0006820$

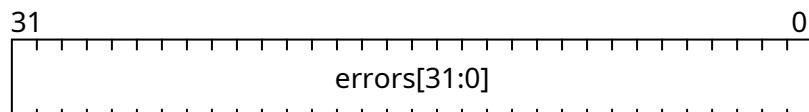


Fig. 19.138: SDRAM_CHECKER_ERRORS

19.2.15 SDRAM_GENERATOR

Register Listing for SDRAM_GENERATOR

Register	Address
<i>SDRAM_GENERATOR_RESET</i>	<i>0xf0007000</i>
<i>SDRAM_GENERATOR_START</i>	<i>0xf0007004</i>
<i>SDRAM_GENERATOR_DONE</i>	<i>0xf0007008</i>
<i>SDRAM_GENERATOR_BASE</i>	<i>0xf000700c</i>
<i>SDRAM_GENERATOR_END</i>	<i>0xf0007010</i>
<i>SDRAM_GENERATOR_LENGTH</i>	<i>0xf0007014</i>
<i>SDRAM_GENERATOR_RANDOM</i>	<i>0xf0007018</i>
<i>SDRAM_GENERATOR_TICKS</i>	<i>0xf000701c</i>

SDRAM_GENERATOR_RESET

Address: $0xf0007000 + 0x0 = 0xf0007000$

SDRAM_GENERATOR_START

Address: $0xf0007000 + 0x4 = 0xf0007004$

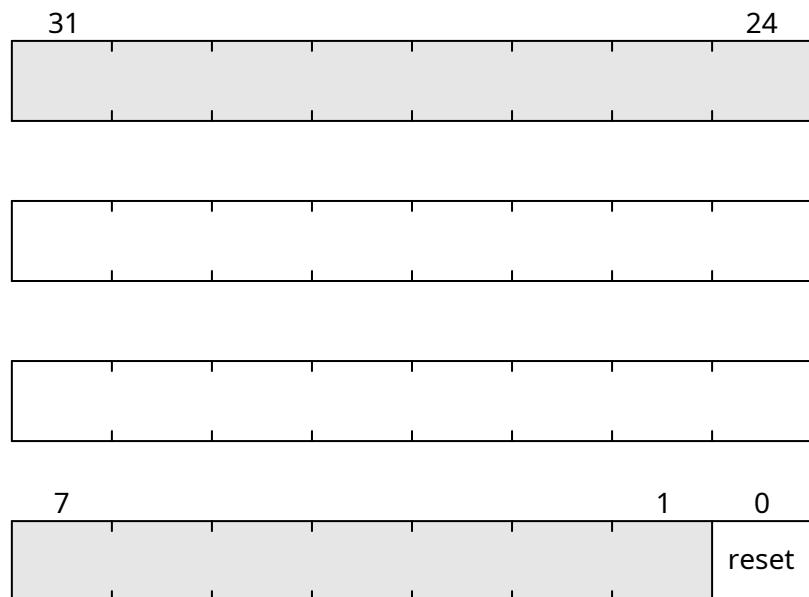


Fig. 19.139: SDRAM_GENERATOR_RESET



Fig. 19.140: SDRAM_GENERATOR_START

SDRAM_GENERATOR_DONE

Address: $0xf0007000 + 0x8 = 0xf0007008$



Fig. 19.141: SDRAM_GENERATOR_DONE

SDRAM_GENERATOR_BASE

Address: $0xf0007000 + 0xc = 0xf000700c$

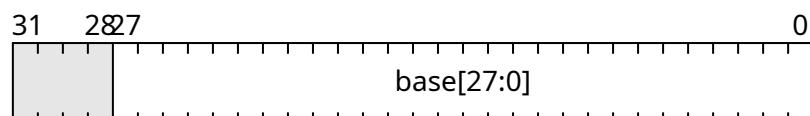


Fig. 19.142: SDRAM GENERATOR BASE

SDRAM GENERATOR END

Address: $0xf0007000 + 0x10 = 0xf0007010$

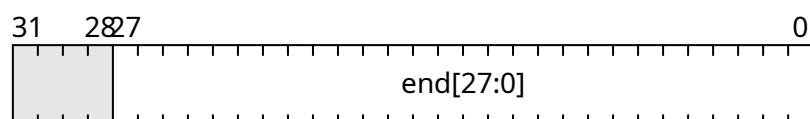


Fig. 19.143: SDRAM GENERATOR END

SDRAM_GENERATOR_LENGTH

Address: $0xf0007000 + 0x14 = 0xf0007014$

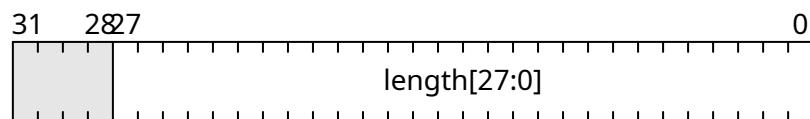


Fig. 19.144: SDRAM_GENERATOR_LENGTH

SDRAM_GENERATOR_RANDOM

Address: $0xf0007000 + 0x18 = 0xf0007018$

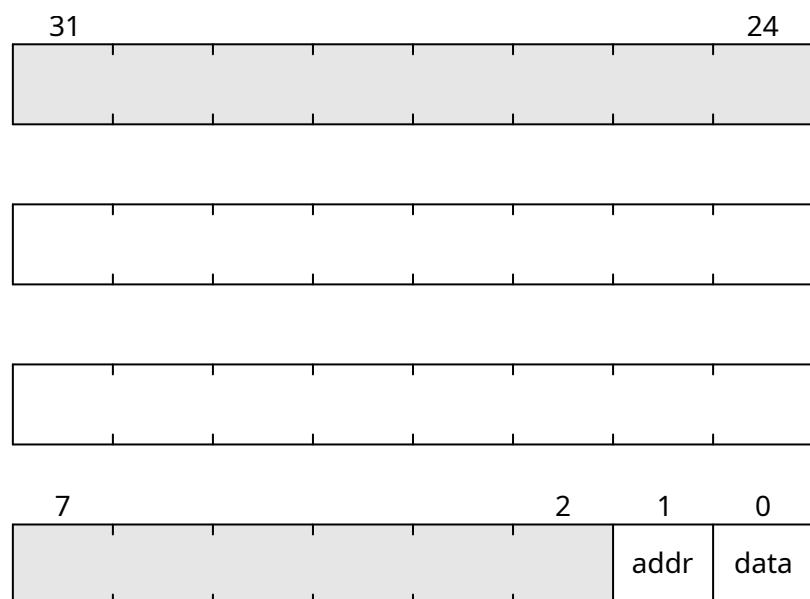


Fig. 19.145: SDRAM_GENERATOR_RANDOM

Field	Name	Description

SDRAM_GENERATOR_TICKS

Address: $0xf0007000 + 0x1c = 0xf000701c$

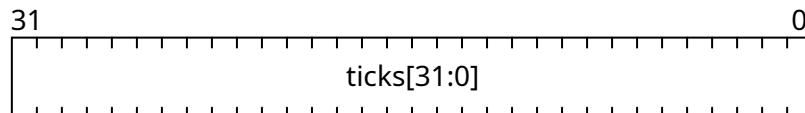


Fig. 19.146: SDRAM_GENERATOR_TICKS

19.2.16 TIMER0

Timer

Provides a generic Timer core.

The Timer is implemented as a countdown timer that can be used in various modes:

- Polling : Returns current countdown value to software
- One-Shot: Loads itself and stops when value reaches 0
- Periodic: (Re-)Loads itself when value reaches 0

en register allows the user to enable/disable the Timer. When the Timer is enabled, it is automatically loaded with the value of *load* register.

When the Timer reaches 0, it is automatically reloaded with value of *reload* register.

The user can latch the current countdown value by writing to *update_value* register, it will update *value* register with current countdown value.

To use the Timer in One-Shot mode, the user needs to:

- Disable the timer
- Set the load register to the expected duration
- (Re-)Enable the Timer

To use the Timer in Periodic mode, the user needs to:

- Disable the Timer
- Set the load register to 0
- Set the reload register to the expected period
- Enable the Timer

For both modes, the CPU can be advertised by an IRQ that the duration/period has elapsed. (The CPU can also do software polling with *update_value* and *value* to know the elapsed duration)

Register Listing for TIMERO

Register	Address
<i>TIMERO_LOAD</i>	0xf0007800
<i>TIMERO_RELOAD</i>	0xf0007804
<i>TIMERO_EN</i>	0xf0007808
<i>TIMERO_UPDATE_VALUE</i>	0xf000780c
<i>TIMERO_VALUE</i>	0xf0007810
<i>TIMERO_EV_STATUS</i>	0xf0007814
<i>TIMERO_EV_PENDING</i>	0xf0007818
<i>TIMERO_EV_ENABLE</i>	0xf000781c

TIMERO_LOAD

Address: $0xf0007800 + 0x0 = 0xf0007800$

Load value when Timer is (re-)enabled. In One-Shot mode, the value written to this register specifies the Timer's duration in clock cycles.



Fig. 19.147: TIMERO_LOAD

TIMERO_RELOAD

Address: $0xf0007800 + 0x4 = 0xf0007804$

Reload value when Timer reaches 0. In Periodic mode, the value written to this register specify the Timer's period in clock cycles.

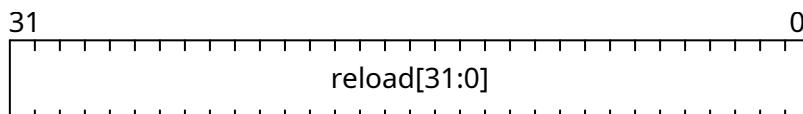


Fig. 19.148: TIMERO_RELOAD

TIMERO_EN

Address: $0xf0007800 + 0x8 = 0xf0007808$

Enable flag of the Timer. Set this flag to 1 to enable/start the Timer. Set to 0 to disable the Timer.

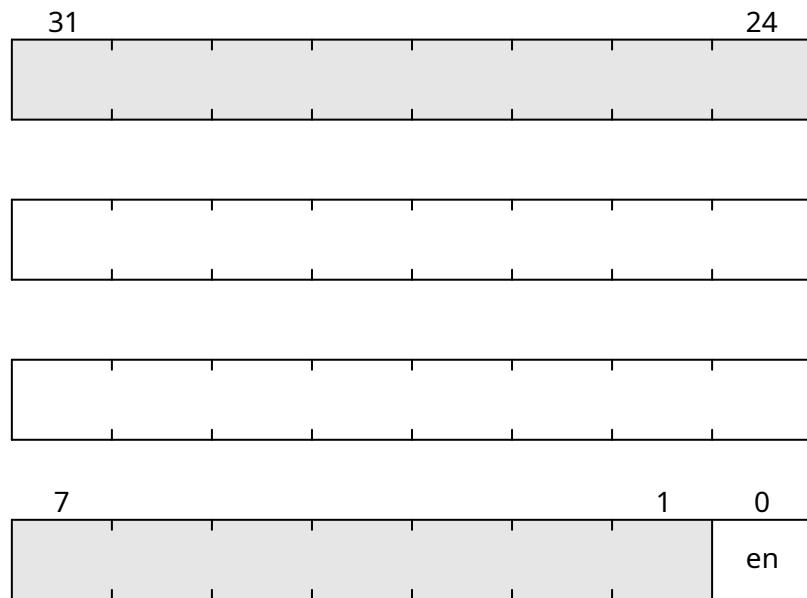


Fig. 19.149: TIMERO_EN

TIMERO_UPDATE_VALUE

Address: $0xf0007800 + 0xc = 0xf000780c$

Update trigger for the current countdown value. A write to this register latches the current countdown value to value register.

TIMERO_VALUE

Address: $0xf0007800 + 0x10 = 0xf0007810$

Latched countdown value. This value is updated by writing to update_value.

TIMERO_EV_STATUS

Address: $0xf0007800 + 0x14 = 0xf0007814$

This register contains the current raw level of the zero event trigger. Writes to this register have no effect.

Field	Name	Description
[0]	ZERO	Level of the zero event

TIMERO_EV_PENDING

Address: $0xf0007800 + 0x18 = 0xf0007818$

When a zero event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.



Fig. 19.150: TIMERO_UPDATE_VALUE

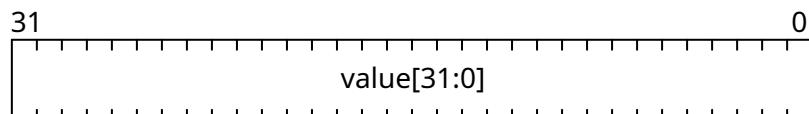


Fig. 19.151: TIMERO_VALUE



Fig. 19.152: TIMERO_EV_STATUS



Fig. 19.153: TIMERO_EV_PENDING

Field	Name	Description
[0]	ZERO	1 if a zero event occurred. This Event is triggered on a falling edge.

TIMERO_EV_ENABLE

Address: $0xf0007800 + 0x1c = 0xf000781c$

This register enables the corresponding zero events. Write a 0 to this register to disable individual events.

Field	Name	Description
[0]	ZERO	Write a 1 to enable the zero Event

19.2.17 UART



Fig. 19.154: TIMERO_EV_ENABLE

Register Listing for UART

Register	Address
UART_RXTX	0xf0008000
UART_TXFULL	0xf0008004
UART_RXEMPTY	0xf0008008
UART_EV_STATUS	0xf000800c
UART_EV_PENDING	0xf0008010
UART_EV_ENABLE	0xf0008014
UART_TXEMPTY	0xf0008018
UART_RXFULL	0xf000801c
UART_XOVER_RXTX	0xf0008020
UART_XOVER_TXFULL	0xf0008024
UART_XOVER_RXEMPTY	0xf0008028
UART_XOVER_EV_STATUS	0xf000802c
UART_XOVER_EV_PENDING	0xf0008030
UART_XOVER_EV_ENABLE	0xf0008034
UART_XOVER_TXEMPTY	0xf0008038
UART_XOVER_RXFULL	0xf000803c

UART_RXTX

Address: $0xf0008000 + 0x0 = 0xf0008000$

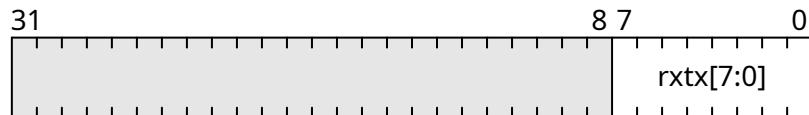


Fig. 19.155: UART_RXTX

UART_TXFULL

Address: $0xf0008000 + 0x4 = 0xf0008004$

TX FIFO Full.



Fig. 19.156: UART_TXFULL

UART_RXEMPTY

Address: $0xf0008000 + 0x8 = 0xf0008008$

RX FIFO Empty.

UART_EV_STATUS

Address: $0xf0008000 + 0xc = 0xf000800c$

This register contains the current raw level of the rx event trigger. Writes to this register have no effect.

Field	Name	Description
[0]	TX	Level of the tx event
[1]	RX	Level of the rx event



Fig. 19.157: `UART_RXEMPTY`

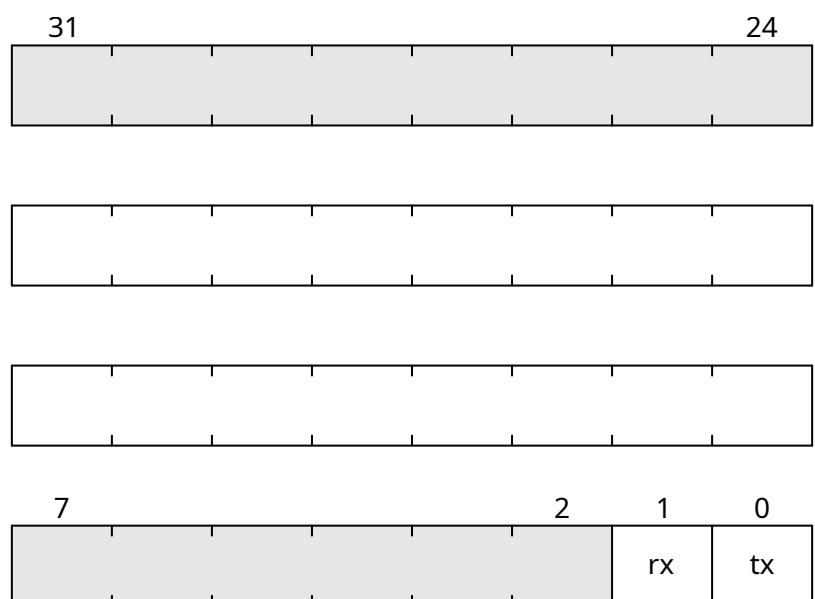


Fig. 19.158: `UART_EV_STATUS`

UART_EV_PENDING

Address: $0xf0008000 + 0x10 = 0xf0008010$

When a rx event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.

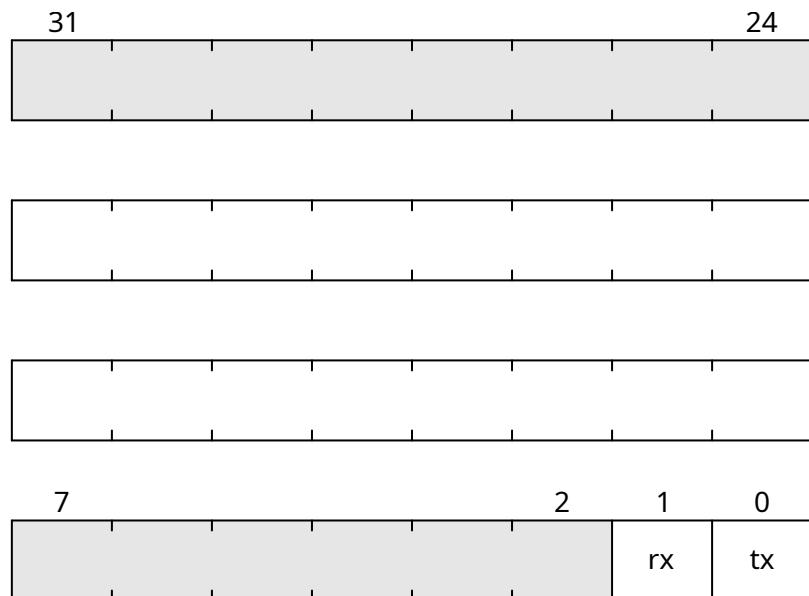


Fig. 19.159: UART_EV_PENDING

Field	Name	Description
[0]	TX	1 if a tx event occurred. This Event is triggered on a falling edge.
[1]	RX	1 if a rx event occurred. This Event is triggered on a falling edge.

UART_EV_ENABLE

Address: $0xf0008000 + 0x14 = 0xf0008014$

This register enables the corresponding rx events. Write a 0 to this register to disable individual events.

Field	Name	Description
[0]	TX	Write a 1 to enable the tx Event
[1]	RX	Write a 1 to enable the rx Event

UART_TXEMPTY

Address: $0xf0008000 + 0x18 = 0xf0008018$

TX FIFO Empty.

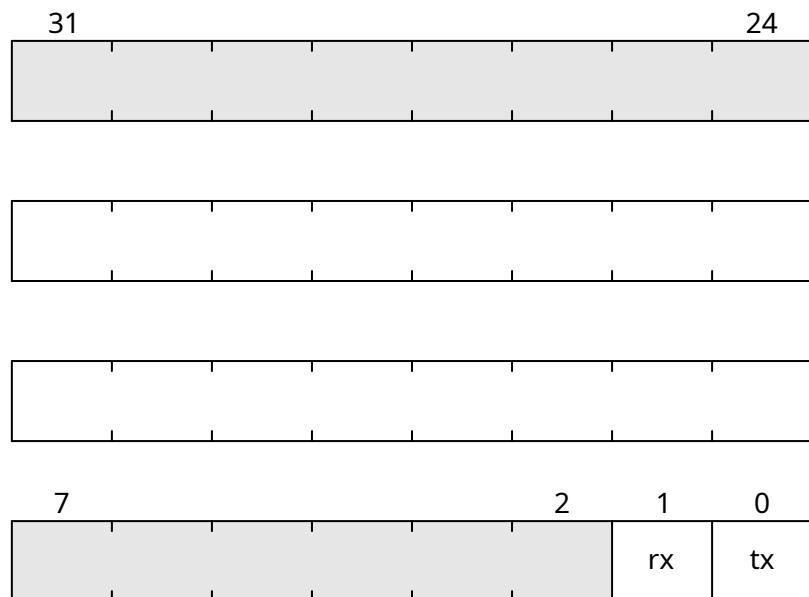


Fig. 19.160: `UART_EV_ENABLE`



Fig. 19.161: `UART_TXEMPTY`

UART_RXFULL

Address: $0xf0008000 + 0x1c = 0xf000801c$

RX FIFO Full.



Fig. 19.162: UART_RXFULL

UART_XOVER_RXTX

Address: $0xf0008000 + 0x20 = 0xf0008020$

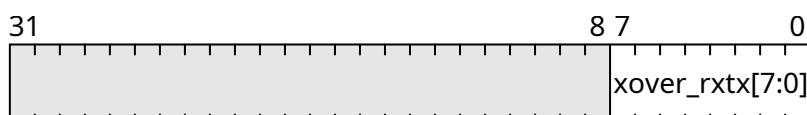


Fig. 19.163: UART_XOVER_RXTX

UART_XOVER_TXFULL

Address: $0xf0008000 + 0x24 = 0xf0008024$

TX FIFO Full.

UART_XOVER_RXEMPTY

Address: $0xf0008000 + 0x28 = 0xf0008028$

RX FIFO Empty.

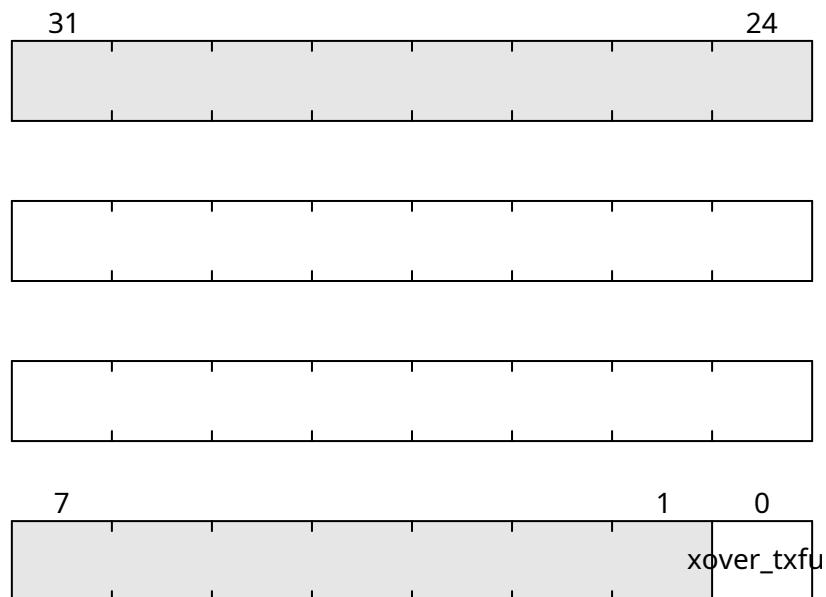


Fig. 19.164: `UART_XOVER_TXFULL`

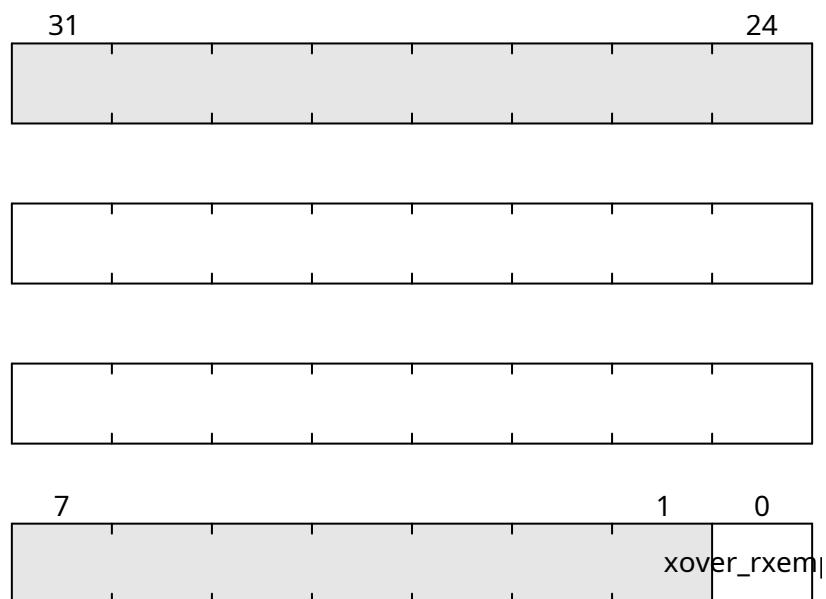


Fig. 19.165: `UART_XOVER_RXEMPTY`

UART_XOVER_EV_STATUS

Address: $0xf0008000 + 0x2c = 0xf000802c$

This register contains the current raw level of the rx event trigger. Writes to this register have no effect.

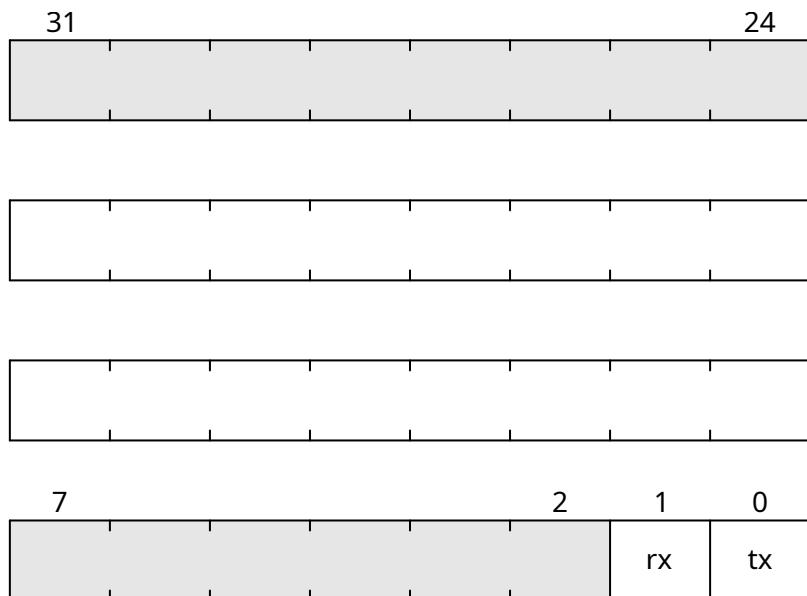


Fig. 19.166: UART_XOVER_EV_STATUS

Field	Name	Description
[0]	TX	Level of the tx event
[1]	RX	Level of the rx event

UART_XOVER_EV_PENDING

Address: $0xf0008000 + 0x30 = 0xf0008030$

When a rx event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.

Field	Name	Description
[0]	TX	1 if a tx event occurred. This Event is triggered on a falling edge .
[1]	RX	1 if a rx event occurred. This Event is triggered on a falling edge .

UART_XOVER_EV_ENABLE

Address: $0xf0008000 + 0x34 = 0xf0008034$

This register enables the corresponding rx events. Write a 0 to this register to disable individual events.

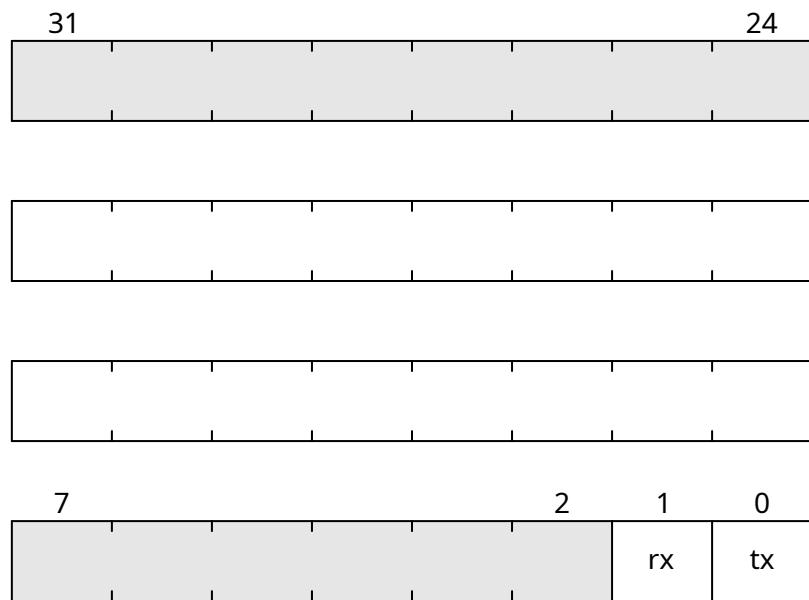


Fig. 19.167: `UART_XOVER_EV_PENDING`

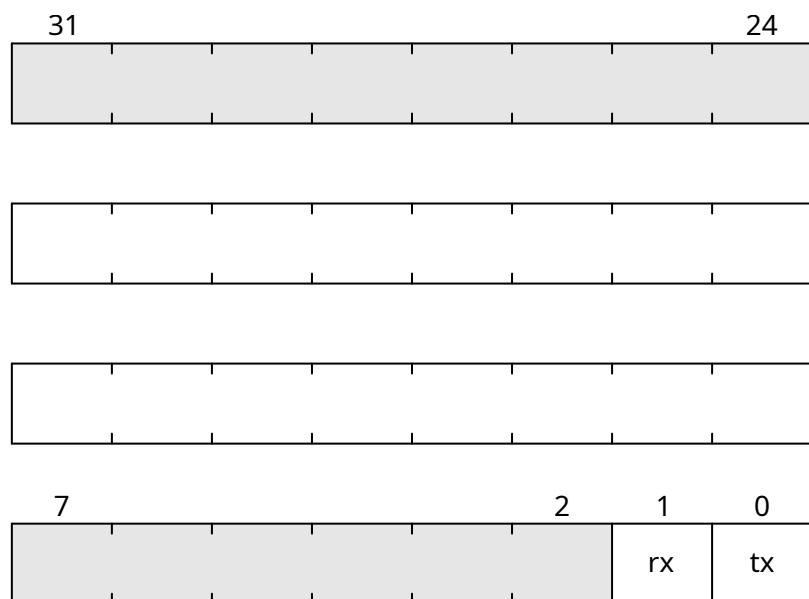


Fig. 19.168: `UART_XOVER_EV_ENABLE`

Field	Name	Description
[0]	TX	Write a 1 to enable the tx Event
[1]	RX	Write a 1 to enable the rx Event

UART_XOVER_TXEMPTY

Address: $0xf0008000 + 0x38 = 0xf0008038$

TX FIFO Empty.

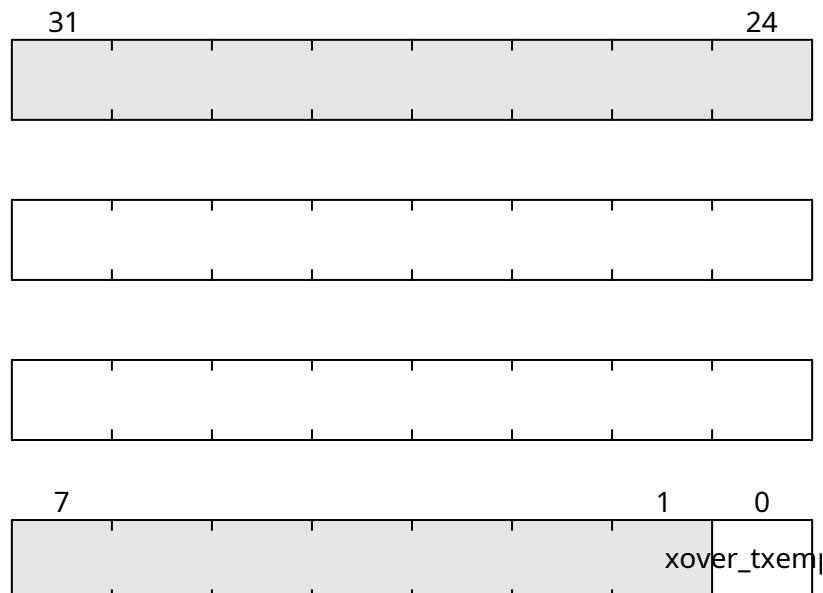


Fig. 19.169: `UART_XOVER_TXEMPTY`

UART_XOVER_RXFULL

Address: $0xf0008000 + 0x3c = 0xf000803c$

RX FIFO Full.

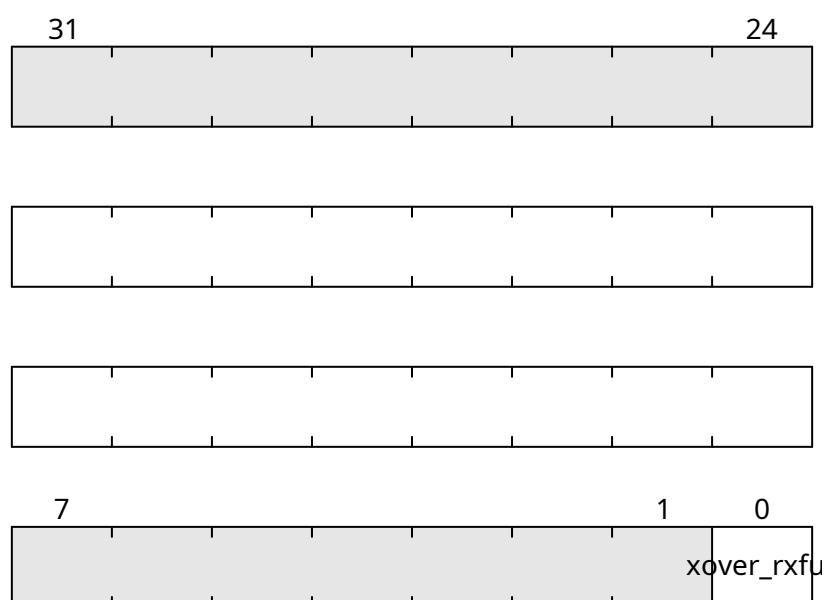


Fig. 19.170: `UART_XOVER_RXFULL`

DOCUMENTATION FOR ROW HAMMER TESTER ZCU104

20.1 Modules

20.1.1 Interrupt Controller

This device has an EventManager-based interrupt system. Individual modules generate *events* which are wired into a central interrupt controller.

When an interrupt occurs, you should look the interrupt number up in the CPU- specific interrupt table and then call the relevant module.

Assigned Interrupts

The following interrupts are assigned on this system:

Interrupt	Module
1	<i>TIMER0</i>
0	<i>UART</i>

20.2 Register Groups

20.2.1 LEDs

Register Listing for LEDs

Register	Address
<i>LEDS_OUT</i>	<i>0xf0000000</i>

LEDS_OUT

Address: *0xf0000000* + *0x0* = *0xf0000000*

Led Output(s) Control.

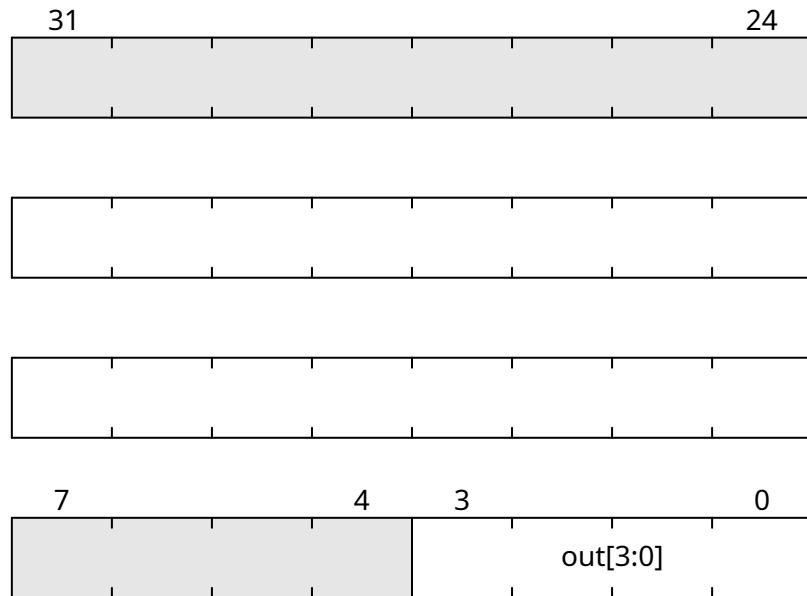


Fig. 20.1: LEDS_OUT

20.2.2 DDRPHY

Register Listing for DDRPHY

Register	Address
<i>DDRPHY_RST</i>	0xf0000800
<i>DDRPHY_EN_VTC</i>	0xf0000804
<i>DDRPHY_HALF_SYS8X_TAPS</i>	0xf0000808
<i>DDRPHY_WLEVEL_EN</i>	0xf000080c
<i>DDRPHY_WLEVEL_STROBE</i>	0xf0000810
<i>DDRPHY_CDLY_RST</i>	0xf0000814
<i>DDRPHY_CDLY_INC</i>	0xf0000818
<i>DDRPHY_CDLY_VALUE</i>	0xf000081c
<i>DDRPHY_DLY_SEL</i>	0xf0000820
<i>DDRPHY_RDLY_DQ_RST</i>	0xf0000824
<i>DDRPHY_RDLY_DQ_INC</i>	0xf0000828
<i>DDRPHY_RDLY_DQ_BITSLIP_RST</i>	0xf000082c
<i>DDRPHY_RDLY_DQ_BITSLIP</i>	0xf0000830
<i>DDRPHY_WDLY_DQ_RST</i>	0xf0000834
<i>DDRPHY_WDLY_DQ_INC</i>	0xf0000838
<i>DDRPHY_WDLY_DQS_RST</i>	0xf000083c
<i>DDRPHY_WDLY_DQS_INC</i>	0xf0000840
<i>DDRPHY_WDLY_DQS_INC_COUNT</i>	0xf0000844
<i>DDRPHY_WDLY_DQ_BITSLIP_RST</i>	0xf0000848
<i>DDRPHY_WDLY_DQ_BITSLIP</i>	0xf000084c
<i>DDRPHY_RDPHASE</i>	0xf0000850
<i>DDRPHY_WRPHASE</i>	0xf0000854

DDRPHY_RST

Address: $0xf0000800 + 0x0 = 0xf0000800$

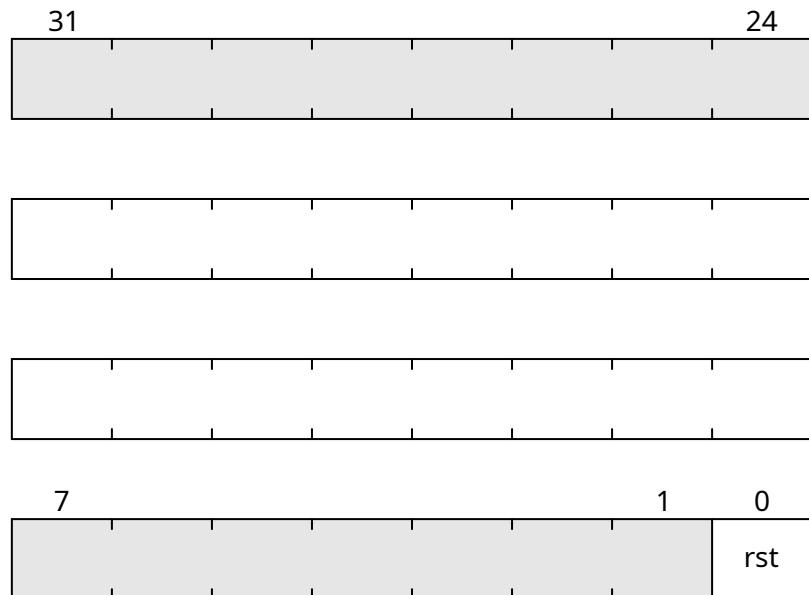


Fig. 20.2: DDRPHY_RST

DDRPHY_EN_VTC

Address: $0xf0000800 + 0x4 = 0xf0000804$

DDRPHY_HALF_SYS8X_TAPS

Address: $0xf0000800 + 0x8 = 0xf0000808$

DDRPHY_WLEVEL_EN

Address: $0xf0000800 + 0xc = 0xf000080c$

DDRPHY_WLEVEL_STROBE

Address: $0xf0000800 + 0x10 = 0xf0000810$

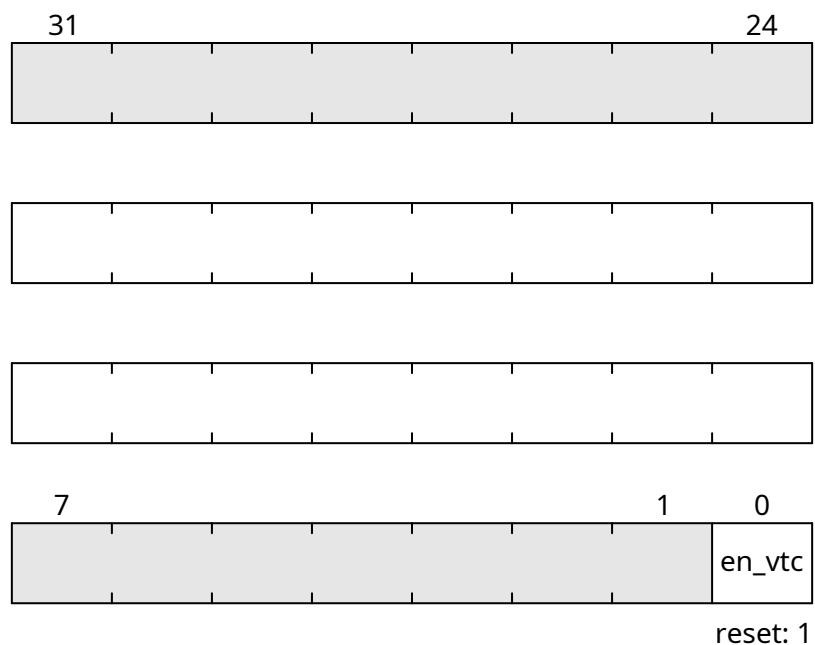


Fig. 20.3: DDRPHY_EN_VTC

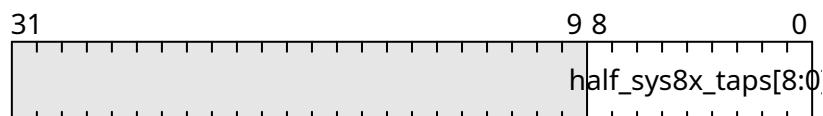


Fig. 20.4: DDRPHY_HALF_SYS8X_TAPS

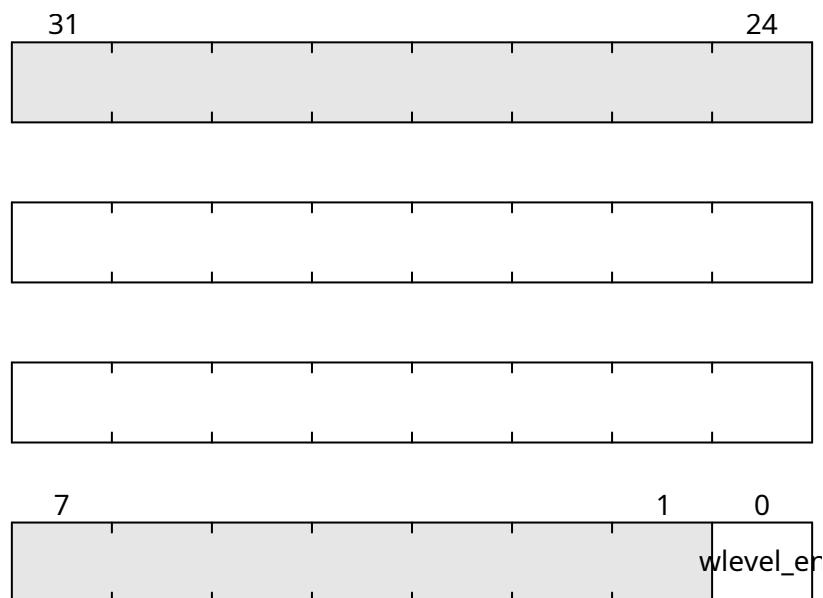


Fig. 20.5: DDRPHY_WLEVEL_EN

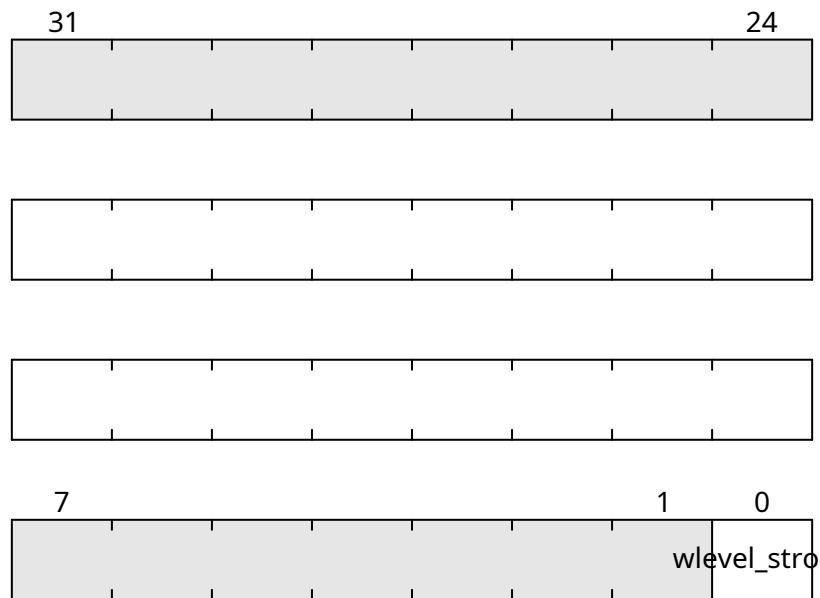


Fig. 20.6: DDRPHY_WLEVEL_STROBE

DDRPHY_CDLY_RST

Address: 0xf0000800 + 0x14 = 0xf0000814

DDRPHY_CDLY_INC

Address: 0xf0000800 + 0x18 = 0xf0000818

DDRPHY_CDLY_VALUE

Address: 0xf0000800 + 0x1c = 0xf000081c

DDRPHY_DLY_SEL

Address: 0xf0000800 + 0x20 = 0xf0000820

DDRPHY_RDLY_DQ_RST

Address: 0xf0000800 + 0x24 = 0xf0000824



Fig. 20.7: DDRPHY_CDLY_RST



Fig. 20.8: DDRPHY_CDLY_INC

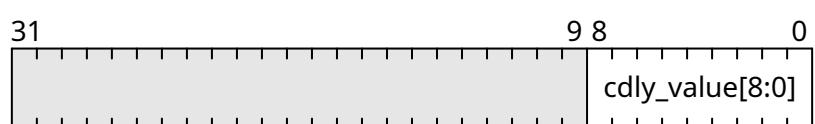


Fig. 20.9: DDRPHY_CDLY_VALUE

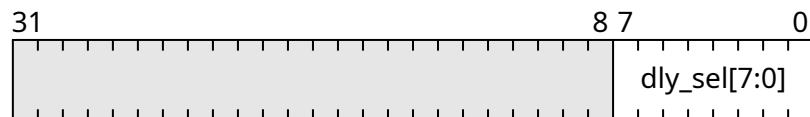


Fig. 20.10: DDRPHY_DLY_SEL



Fig. 20.11: DDRPHY_RDLY_DQ_RST

DDRPHY_RDLY_DQ_INC

Address: $0xf0000800 + 0x28 = 0xf0000828$

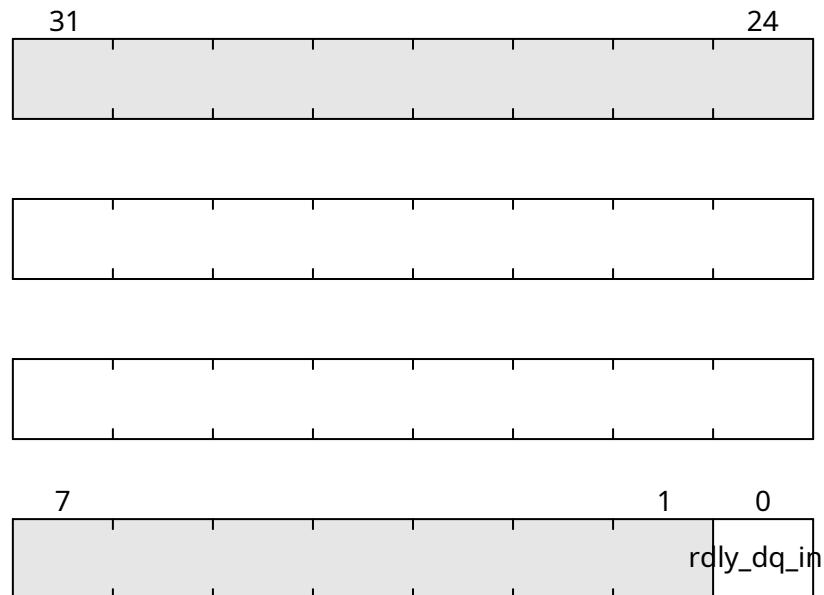


Fig. 20.12: DDRPHY_RDLY_DQ_INC

DDRPHY_RDLY_DQ_BITSLIP_RST

Address: $0xf0000800 + 0x2c = 0xf000082c$

DDRPHY_RDLY_DQ_BITSLIP

Address: $0xf0000800 + 0x30 = 0xf0000830$

DDRPHY_WDLY_DQ_RST

Address: $0xf0000800 + 0x34 = 0xf0000834$

DDRPHY_WDLY_DQ_INC

Address: $0xf0000800 + 0x38 = 0xf0000838$

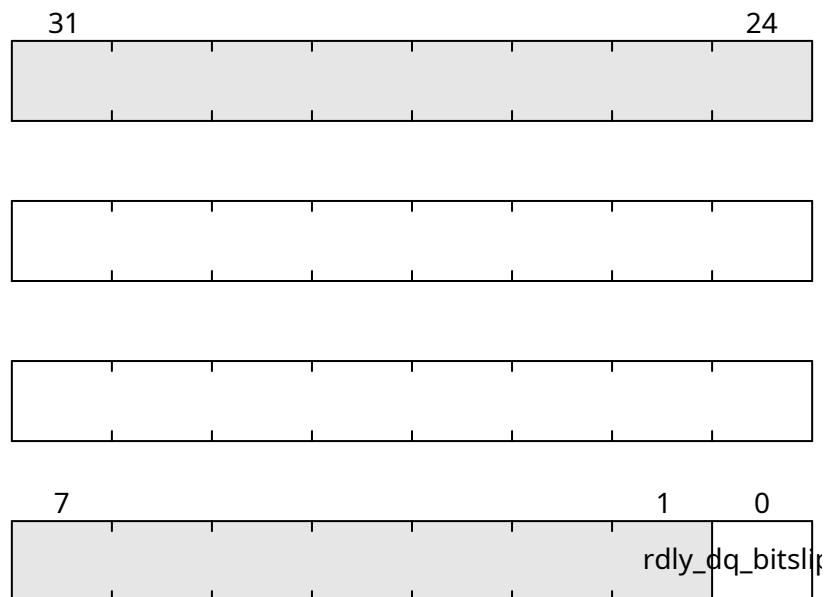


Fig. 20.13: `DDRPHY_RDLY_DQ_BITSLIP_RST`

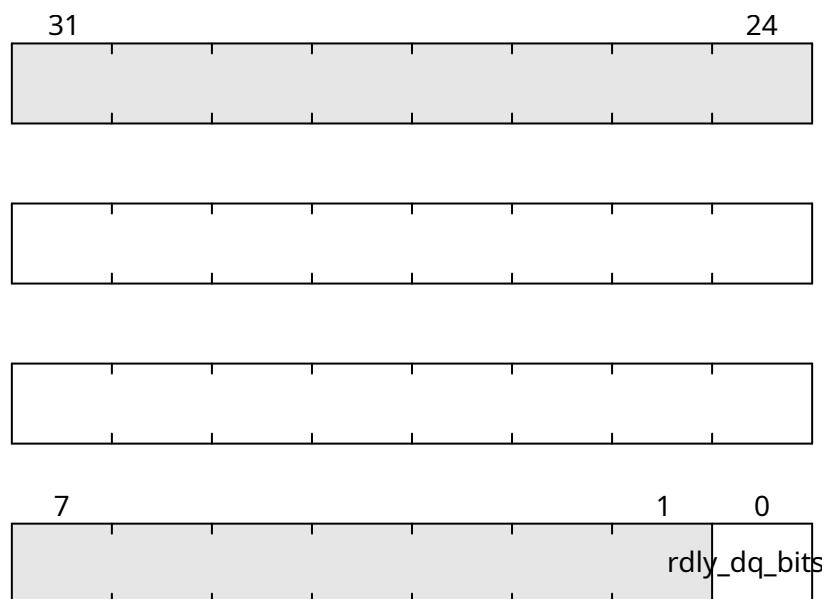


Fig. 20.14: `DDRPHY_RDLY_DQ_BITSLIP`

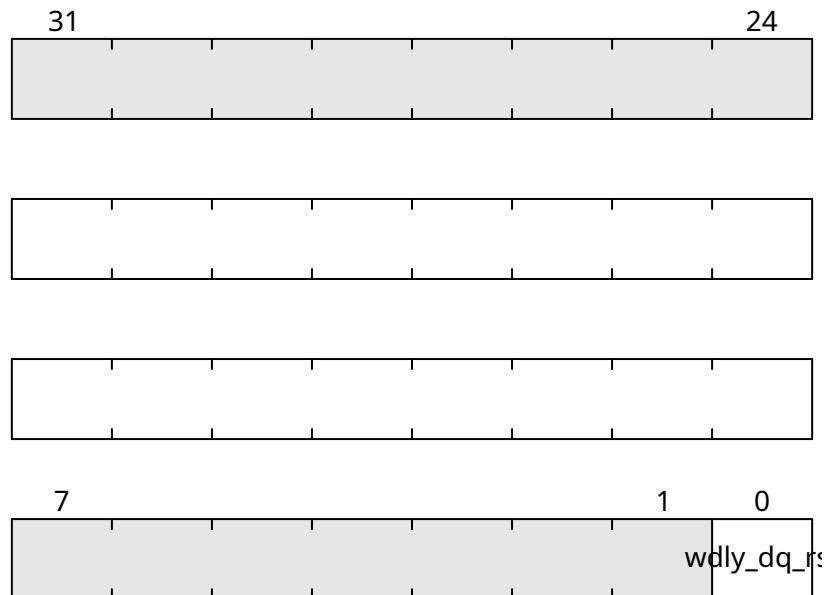


Fig. 20.15: `DDRPHY_WDLY_DQ_RST`

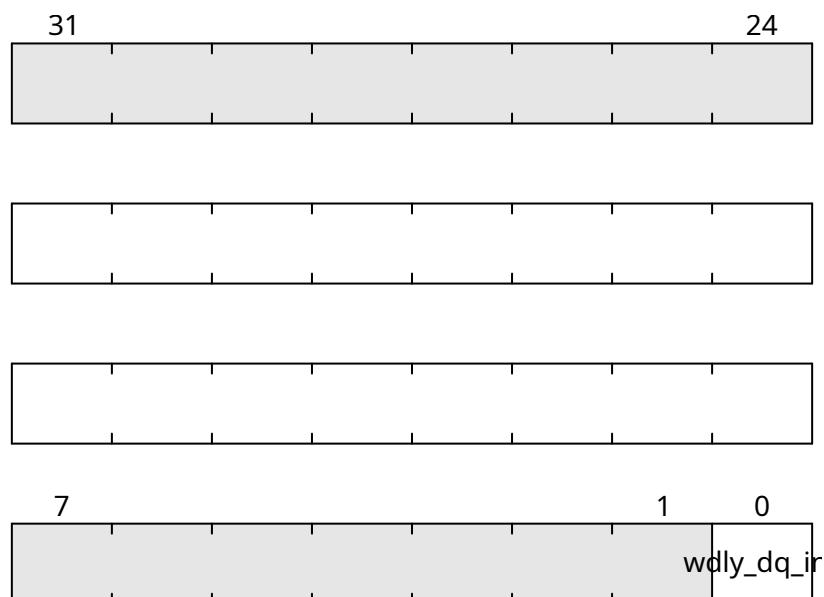


Fig. 20.16: `DDRPHY_WDLY_DQ_INC`

DDRPHY_WDLY_DQS_RST

Address: $0xf0000800 + 0x3c = 0xf000083c$

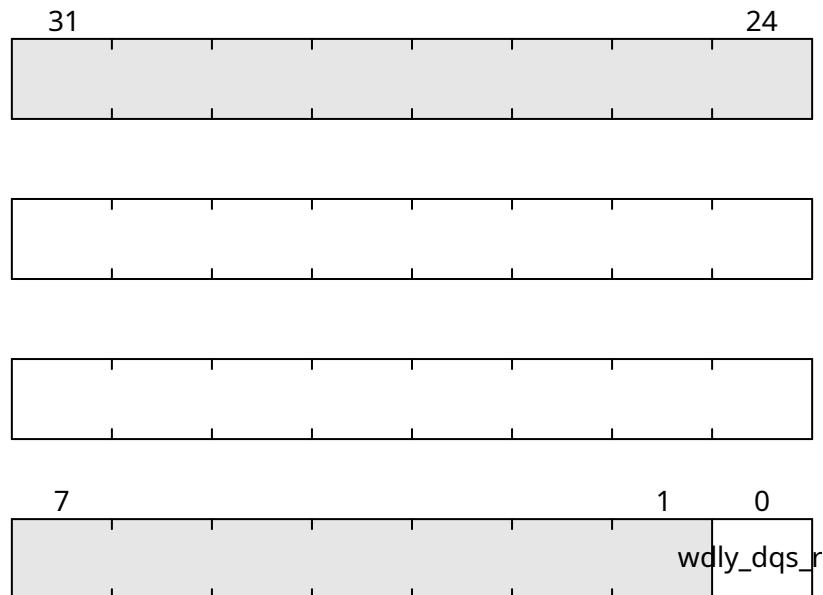


Fig. 20.17: DDRPHY_WDLY_DQS_RST

DDRPHY_WDLY_DQS_INC

Address: $0xf0000800 + 0x40 = 0xf0000840$

DDRPHY_WDLY_DQS_INC_COUNT

Address: $0xf0000800 + 0x44 = 0xf0000844$

DDRPHY_WDLY_DQ_BITSLIP_RST

Address: $0xf0000800 + 0x48 = 0xf0000848$

DDRPHY_WDLY_DQ_BITSLIP

Address: $0xf0000800 + 0x4c = 0xf000084c$

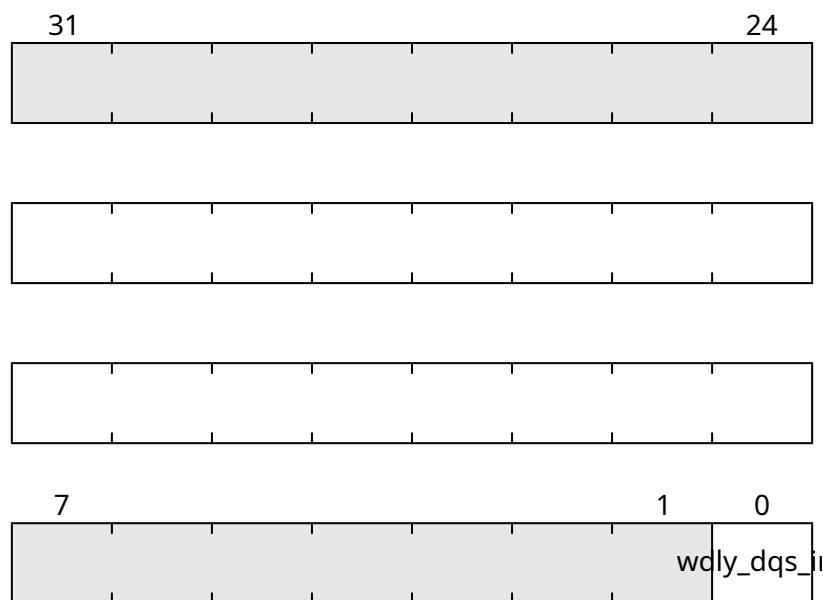


Fig. 20.18: DDRPHY_WDLY_DQS_INC

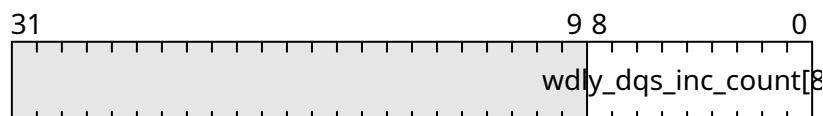


Fig. 20.19: DDRPHY_WDLY_DQS_INC_COUNT

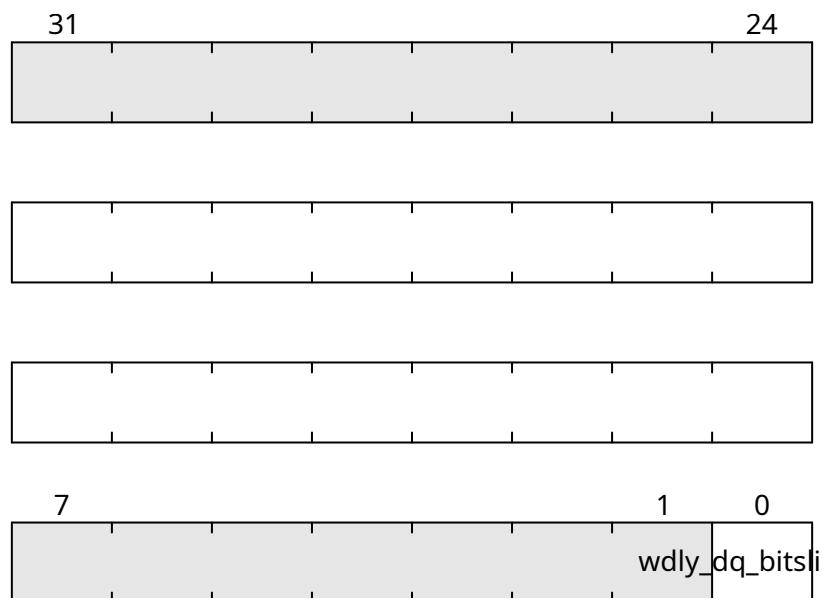


Fig. 20.20: DDRPHY_WDLY_DQ_BITSLIP_RST

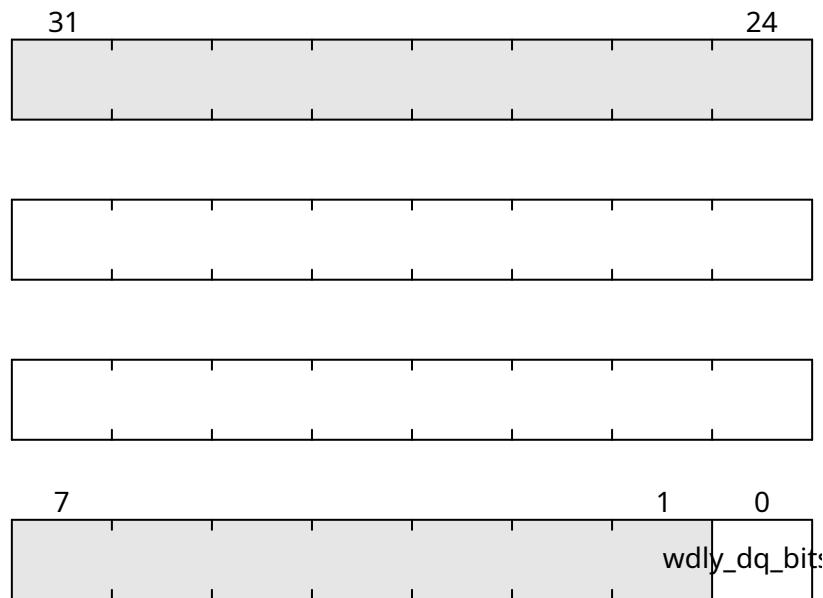


Fig. 20.21: DDRPHY_WDLY_DQ_BITSLIP

DDRPHY_RDPHASE

Address: 0xf0000800 + 0x50 = 0xf0000850

DDRPHY_WRPHASE

Address: 0xf0000800 + 0x54 = 0xf0000854

20.2.3 CONTROLLER_SETTINGS

Allows to change LiteDRAMController behaviour at runtime

Register Listing for CONTROLLER_SETTINGS

Register	Address
<i>CONTROLLER_SETTINGS_REFRESH</i>	<i>0xf0001000</i>

CONTROLLER_SETTINGS_REFRESH

Address: 0xf0001000 + 0x0 = 0xf0001000

Enable/disable Refresh commands sending

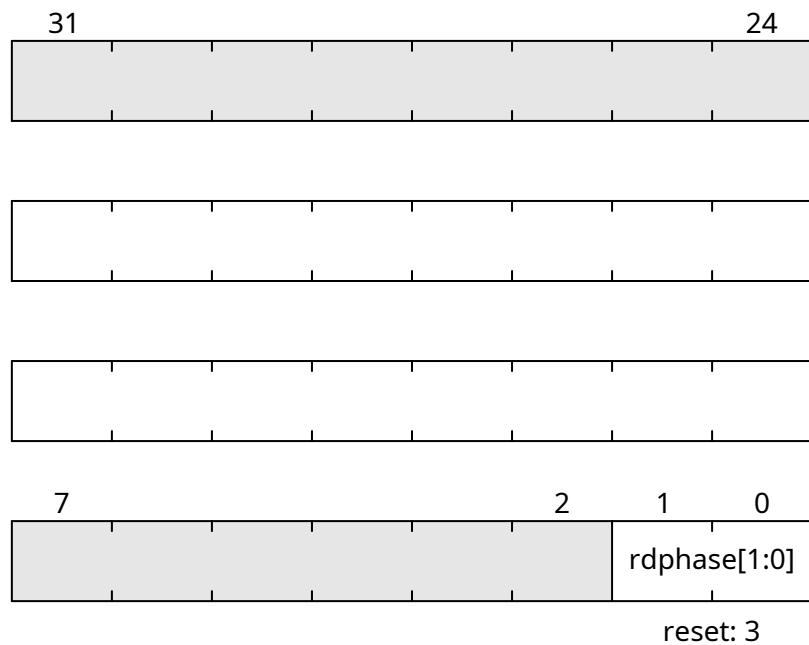


Fig. 20.22: DDRPHY_RDPHASE

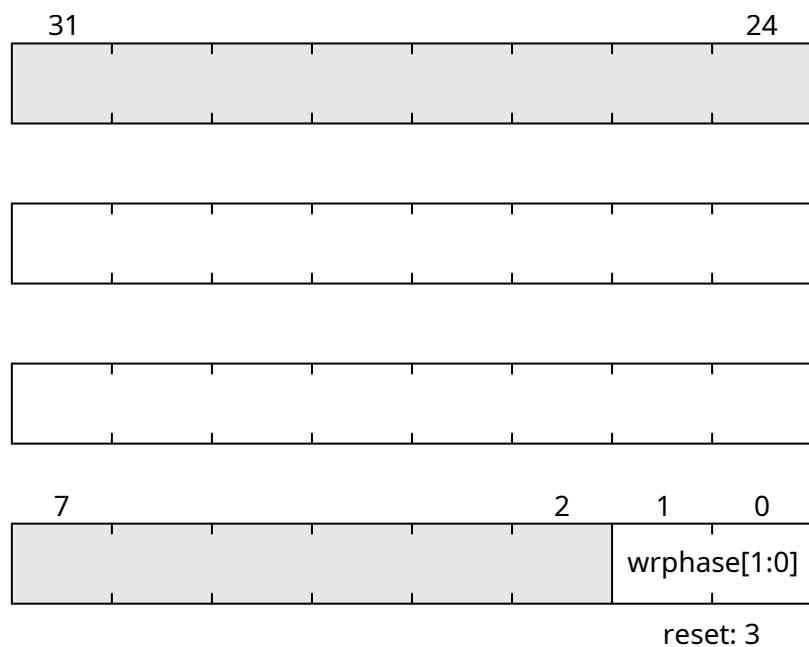


Fig. 20.23: DDRPHY_WRPAGE

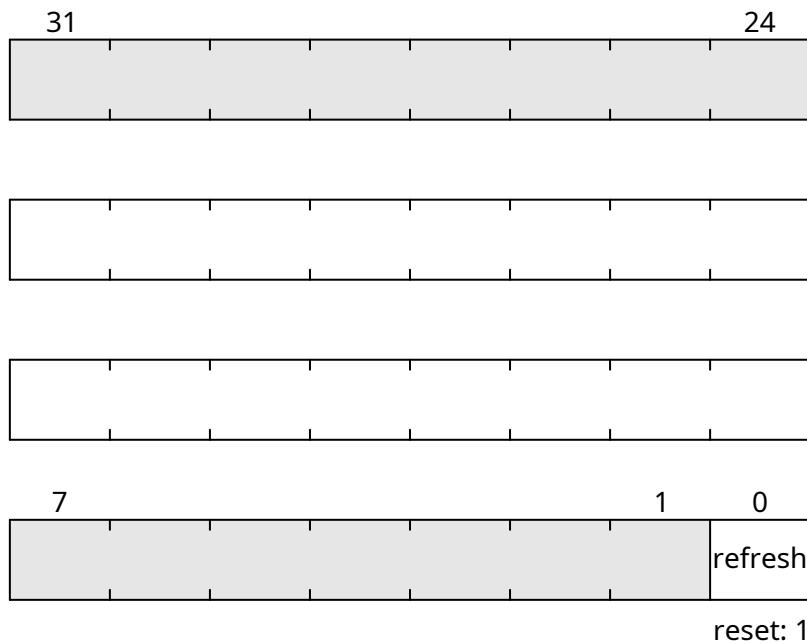


Fig. 20.24: CONTROLLER_SETTINGS_REFRESH

20.2.4 DDRCTRL

Register Listing for DDRCTRL

Register	Address
<i>DDRCTRL_INIT_DONE</i>	<i>0xf0001800</i>
<i>DDRCTRL_INIT_ERROR</i>	<i>0xf0001804</i>

DDRCTRL_INIT_DONE

Address: $0xf0001800 + 0x0 = 0xf0001800$

DDRCTRL_INIT_ERROR

Address: $0xf0001800 + 0x4 = 0xf0001804$

20.2.5 ROWHAMMER

Row Hammer DMA attacker

This module allows to perform a Row Hammer attack by configuring it with two addresses that map to different rows of a single bank. When enabled, it will perform alternating DMA reads from the given locations, which will result in the DRAM controller having to repeatedly open/close rows at each read access.



Fig. 20.25: DDRCTRL_INIT_DONE

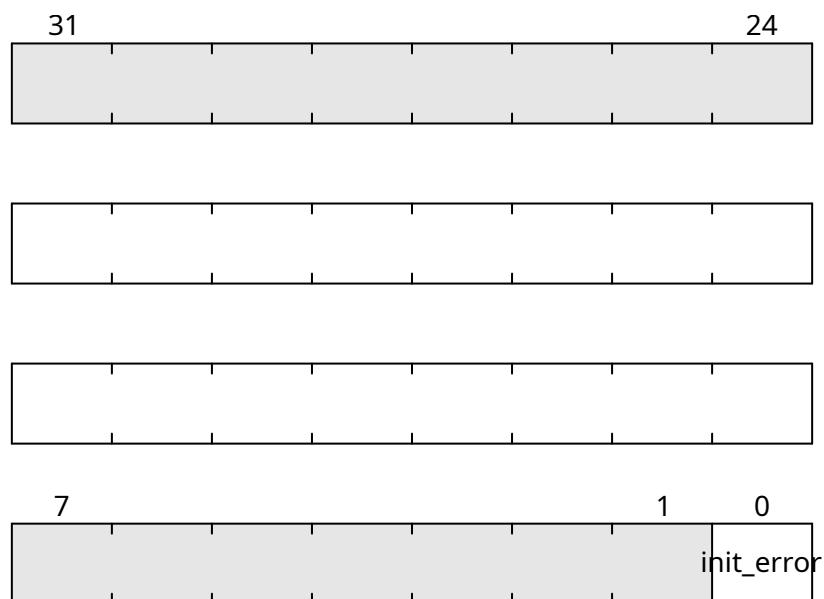


Fig. 20.26: DDRCTRL_INIT_ERROR

Register Listing for ROWHAMMER

Register	Address
<i>ROWHAMMER_ENABLED</i>	<i>0xf0002000</i>
<i>ROWHAMMER_ADDRESS1</i>	<i>0xf0002004</i>
<i>ROWHAMMER_ADDRESS2</i>	<i>0xf0002008</i>
<i>ROWHAMMER_COUNT</i>	<i>0xf000200c</i>

ROWHAMMER_ENABLED

Address: $0xf0002000 + 0x0 = 0xf0002000$

Used to start/stop the operation of the module



Fig. 20.27: ROWHAMMER_ENABLED

ROWHAMMER_ADDRESS1

Address: $0xf0002000 + 0x4 = 0xf0002004$

First attacked address

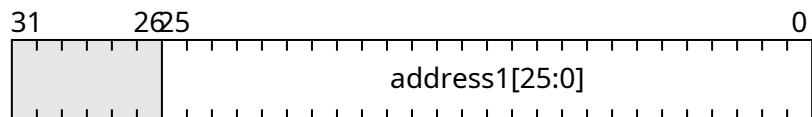


Fig. 20.28: ROWHAMMER_ADDRESS1

ROWHAMMER_ADDRESS2

Address: $0xf0002000 + 0x8 = 0xf0002008$

Second attacked address

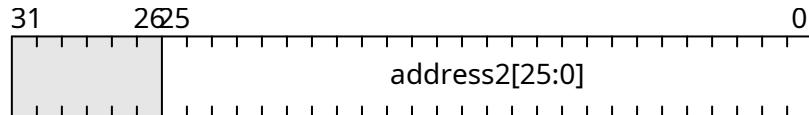


Fig. 20.29: ROWHAMMER_ADDRESS2

ROWHAMMER_COUNT

Address: $0xf0002000 + 0xc = 0xf000200c$

This is the number of DMA accesses performed. When the module is enabled, the value can be freely read. When the module is disabled, the register is clear-on-write and has to be read before the next attack.

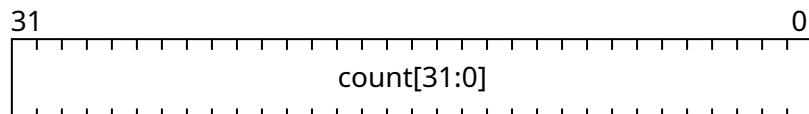


Fig. 20.30: ROWHAMMER_COUNT

20.2.6 WRITER

DMA DRAM writer.

Allows to fill DRAM with a predefined pattern using DMA.

Pattern

Provides access to RAM to store access pattern: *mem_addr* and *mem_data*. The pattern address space can be limited using the *data_mask*.

For example, having *mem_addr* filled with [0x04, 0x02, 0x03, ...] and *mem_data* filled with [0xff, 0xaa, 0x55, ...] and setting *data_mask* = 0b01, the pattern [(address, data), ...] written will be: [(0x04, 0xff), (0x02, 0xaa), (0x04, 0xff), ...] (wraps due to masking).

DRAM memory range that is being accessed can be configured using *mem_mask*.

To use this module, make sure that *ready* is 1, then write the desired number of transfers to *count*. Writing to the *start* CSR will initialize the operation. When the operation is ongoing *ready* will be 0.

Register Listing for WRITER

Register	Address
WRITER_START	0xf0002800
WRITER_READY	0xf0002804
WRITER_MODULO	0xf0002808
WRITER_COUNT	0xf000280c
WRITER_DONE	0xf0002810
WRITER_MEM_MASK	0xf0002814
WRITER_DATA_MASK	0xf0002818
WRITER_DATA_DIV	0xf000281c
WRITER_INVERTER_DIVISOR_MASK	0xf0002820
WRITER_INVERTER_SELECTION_MASK	0xf0002824
WRITER_LAST_ADDRESS	0xf0002828

WRITER_START

Address: $0xf0002800 + 0x0 = 0xf0002800$

Write to the register starts the transfer (if ready=1)



Fig. 20.31: WRITER_START

WRITER_READY

Address: $0xf0002800 + 0x4 = 0xf0002804$

Indicates that the transfer is not ongoing



Fig. 20.32: WRITER_READY

WRITER_MODULO

Address: $0xf0002800 + 0x8 = 0xf0002808$

When set use modulo to calculate DMA transfers address rather than bit masking

WRITER_COUNT

Address: $0xf0002800 + 0xc = 0xf000280c$

Desired number of DMA transfers

WRITER_DONE

Address: $0xf0002800 + 0x10 = 0xf0002810$

Number of completed DMA transfers

WRITER_MEM_MASK

Address: $0xf0002800 + 0x14 = 0xf0002814$

DRAM address mask for DMA transfers

WRITER_DATA_MASK

Address: $0xf0002800 + 0x18 = 0xf0002818$

Pattern memory address mask



Fig. 20.33: WRITER_MODULO

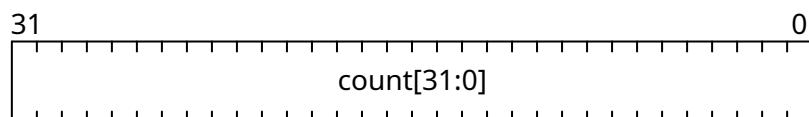


Fig. 20.34: WRITER_COUNT

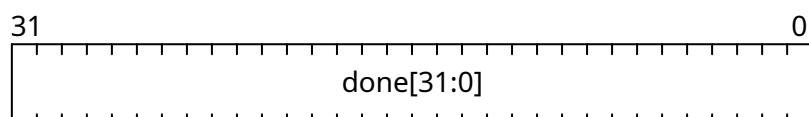


Fig. 20.35: WRITER_DONE

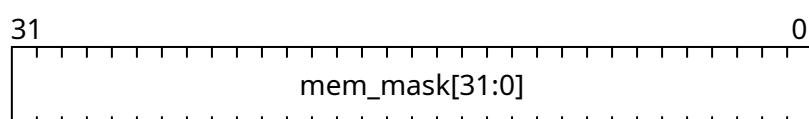


Fig. 20.36: WRITER_MEM_MASK



Fig. 20.37: WRITER_DATA_MASK

WRITER_DATA_DIV

Address: 0xf0002800 + 0x1c = 0xf000281c

Pattern memory address divisor-1

WRITER_INVERTER_DIVISOR_MASK

Address: 0xf0002800 + 0x20 = 0xf0002820

Divisor mask for selecting rows for which pattern data gets inverted

WRITER_INVERTER_SELECTION_MASK

Address: 0xf0002800 + 0x24 = 0xf0002824

Selection mask for selecting rows for which pattern data gets inverted

WRITER_LAST_ADDRESS

Address: 0xf0002800 + 0x28 = 0xf0002828

Number of completed DMA transfers

20.2.7 READER

DMA DRAM reader.

Allows to check DRAM contents against a predefined pattern using DMA.

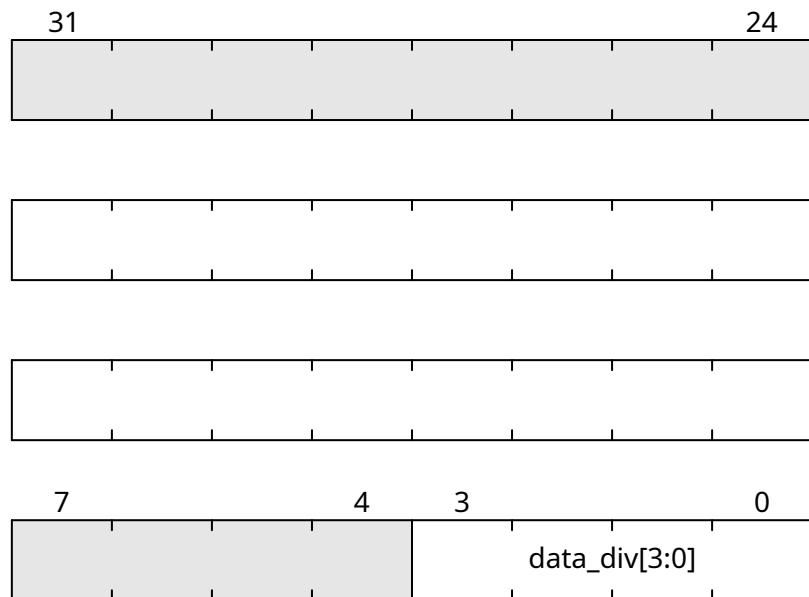


Fig. 20.38: WRITER_DATA_DIV

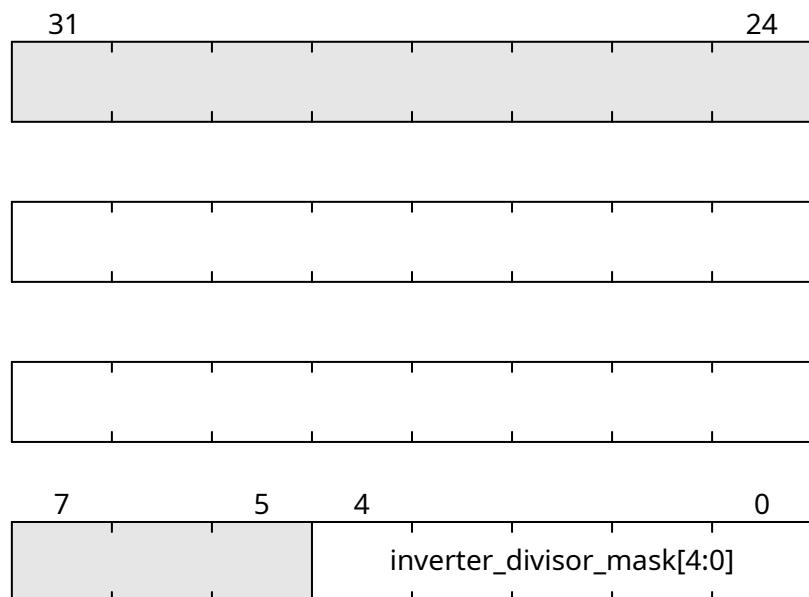


Fig. 20.39: WRITER_INVERTER_DIVISOR_MASK

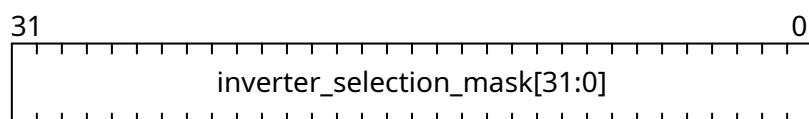


Fig. 20.40: WRITER_INVERTER_SELECTION_MASK

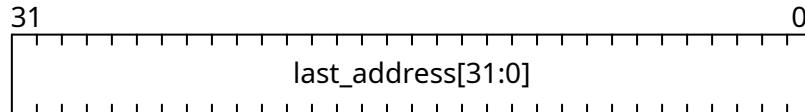


Fig. 20.41: WRITER_LAST_ADDRESS

Pattern

Provides access to RAM to store access pattern: *mem_addr* and *mem_data*. The pattern address space can be limited using the *data_mask*.

For example, having *mem_addr* filled with [0x04, 0x02, 0x03, ...] and *mem_data* filled with [0xff, 0xaa, 0x55, ...] and setting *data_mask* = 0b01, the pattern [(address, data), ...] written will be: [(0x04, 0xff), (0x02, 0xaa), (0x04, 0xff), ...] (wraps due to masking).

DRAM memory range that is being accessed can be configured using *mem_mask*.

To use this module, make sure that *ready* is 1, then write the desired number of transfers to *count*. Writing to the *start* CSR will initialize the operation. When the operation is ongoing *ready* will be 0.

Reading errors

This module allows to check the locations of errors in the memory. It scans the configured memory area and compares the values read to the predefined pattern. If *skip_fifo* is 0, this module will stop after each error encountered, so that it can be examined. Wait until the *error_ready* CSR is 1. Then use the CSRs *error_offset*, *error_data* and *error_expected* to examine the errors in the current transfer. To continue reading, write 1 to *error_continue* CSR. Setting *skip_fifo* to 1 will disable this behavior entirely.

The final number of errors can be read from *error_count*. NOTE: This value represents the number of erroneous DMA transfers.

The current progress can be read from the *done* CSR.

Register Listing for READER

Register	Address
<i>READER_START</i>	0xf0003000
<i>READER_READY</i>	0xf0003004
<i>READER_MODULO</i>	0xf0003008
<i>READER_COUNT</i>	0xf000300c
<i>READER_DONE</i>	0xf0003010
<i>READER_MEM_MASK</i>	0xf0003014
<i>READER_DATA_MASK</i>	0xf0003018
<i>READER_DATA_DIV</i>	0xf000301c
<i>READER_INVERTER_DIVISOR_MASK</i>	0xf0003020
<i>READER_INVERTER_SELECTION_MASK</i>	0xf0003024
<i>READER_ERROR_COUNT</i>	0xf0003028

continues on next page

Table 20.1 – continued from previous page

Register	Address
READER_SKIP_FIFO	0xf000302c
READER_ERROR_OFFSET	0xf0003030
READER_ERROR_DATA15	0xf0003034
READER_ERROR_DATA14	0xf0003038
READER_ERROR_DATA13	0xf000303c
READER_ERROR_DATA12	0xf0003040
READER_ERROR_DATA11	0xf0003044
READER_ERROR_DATA10	0xf0003048
READER_ERROR_DATA9	0xf000304c
READER_ERROR_DATA8	0xf0003050
READER_ERROR_DATA7	0xf0003054
READER_ERROR_DATA6	0xf0003058
READER_ERROR_DATA5	0xf000305c
READER_ERROR_DATA4	0xf0003060
READER_ERROR_DATA3	0xf0003064
READER_ERROR_DATA2	0xf0003068
READER_ERROR_DATA1	0xf000306c
READER_ERROR_DATA0	0xf0003070
READER_ERROR_EXPECTED15	0xf0003074
READER_ERROR_EXPECTED14	0xf0003078
READER_ERROR_EXPECTED13	0xf000307c
READER_ERROR_EXPECTED12	0xf0003080
READER_ERROR_EXPECTED11	0xf0003084
READER_ERROR_EXPECTED10	0xf0003088
READER_ERROR_EXPECTED9	0xf000308c
READER_ERROR_EXPECTED8	0xf0003090
READER_ERROR_EXPECTED7	0xf0003094
READER_ERROR_EXPECTED6	0xf0003098
READER_ERROR_EXPECTED5	0xf000309c
READER_ERROR_EXPECTED4	0xf00030a0
READER_ERROR_EXPECTED3	0xf00030a4
READER_ERROR_EXPECTED2	0xf00030a8
READER_ERROR_EXPECTED1	0xf00030ac
READER_ERROR_EXPECTED0	0xf00030b0
READER_ERROR_READY	0xf00030b4
READER_ERROR_CONTINUE	0xf00030b8

READER_START

Address: $0xf0003000 + 0x0 = 0xf0003000$

Write to the register starts the transfer (if ready=1)

READER_READY

Address: $0xf0003000 + 0x4 = 0xf0003004$

Indicates that the transfer is not ongoing



Fig. 20.42: READER_START



Fig. 20.43: READER_READY

READER_MODULO

Address: $0xf0003000 + 0x8 = 0xf0003008$

When set use modulo to calculate DMA transfers address rather than bit masking



Fig. 20.44: READER_MODULO

READER_COUNT

Address: $0xf0003000 + 0xc = 0xf000300c$

Desired number of DMA transfers

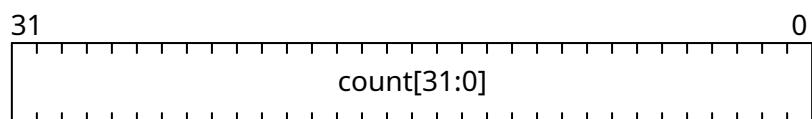


Fig. 20.45: READER_COUNT

READER_DONE

Address: $0xf0003000 + 0x10 = 0xf0003010$

Number of completed DMA transfers

READER_MEM_MASK

Address: $0xf0003000 + 0x14 = 0xf0003014$

DRAM address mask for DMA transfers

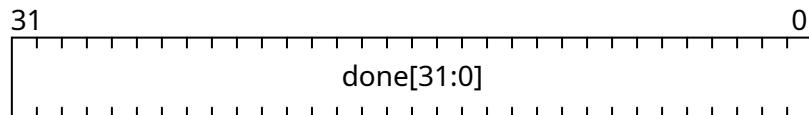


Fig. 20.46: READER_DONE

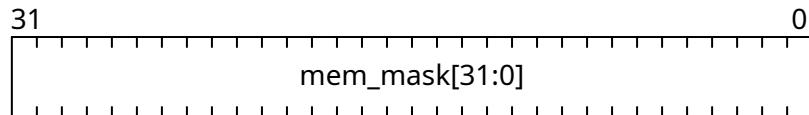


Fig. 20.47: READER_MEM_MASK

READER_DATA_MASK

Address: 0xf0003000 + 0x18 = 0xf0003018

Pattern memory address mask

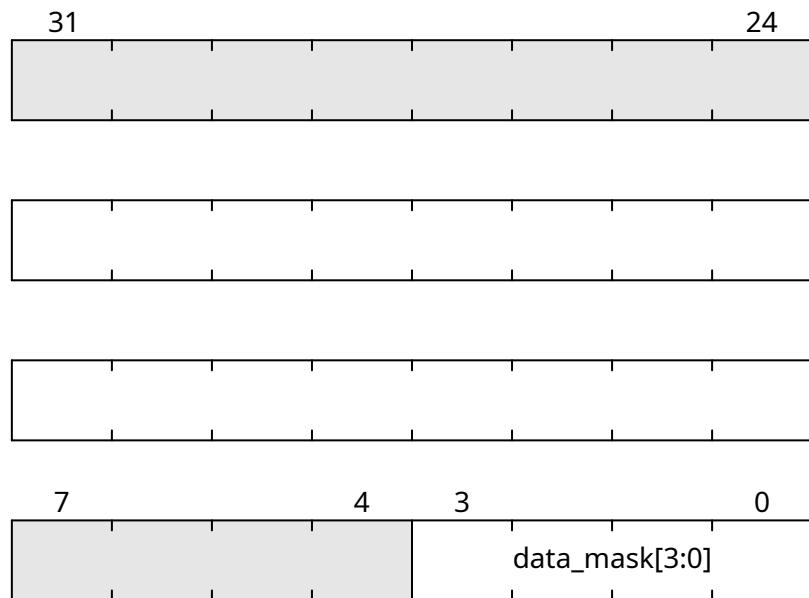


Fig. 20.48: READER_DATA_MASK

READER_DATA_DIV

Address: 0xf0003000 + 0x1c = 0xf000301c

Pattern memory address divisor-1

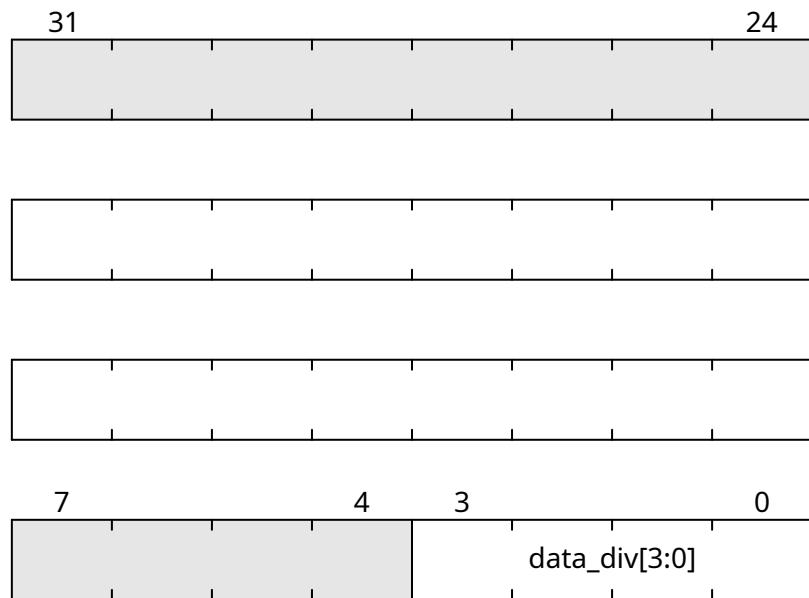


Fig. 20.49: READER_DATA_DIV

READER_INVERTER_DIVISOR_MASK

Address: 0xf0003000 + 0x20 = 0xf0003020

Divisor mask for selecting rows for which pattern data gets inverted

READER_INVERTER_SELECTION_MASK

Address: 0xf0003000 + 0x24 = 0xf0003024

Selection mask for selecting rows for which pattern data gets inverted

READER_ERROR_COUNT

Address: 0xf0003000 + 0x28 = 0xf0003028

Number of errors detected

READER_SKIP_FIFO

Address: 0xf0003000 + 0x2c = 0xf000302c

Skip waiting for user to read the errors FIFO

READER_ERROR_OFFSET

Address: 0xf0003000 + 0x30 = 0xf0003030

Current offset of the error

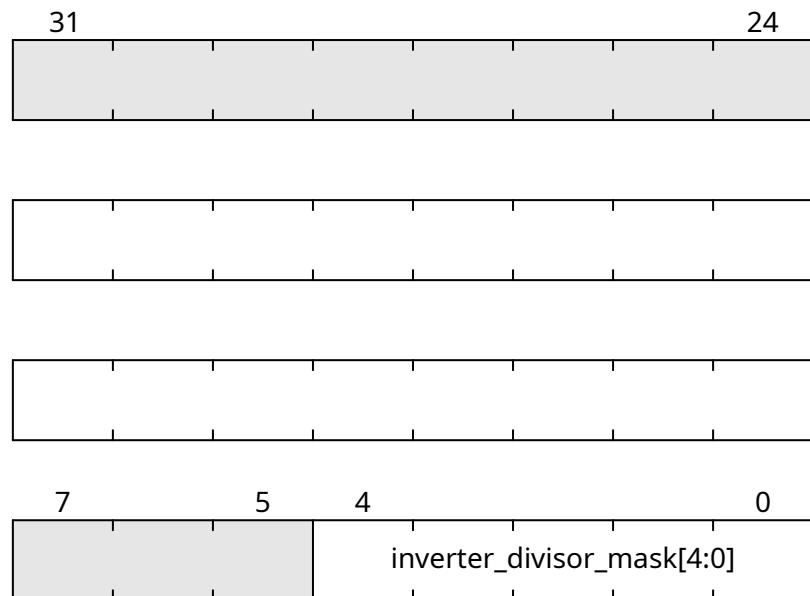


Fig. 20.50: READER_INVERTER_DIVISOR_MASK

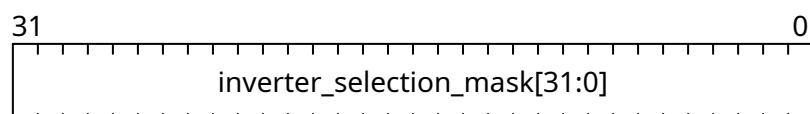


Fig. 20.51: READER_INVERTER_SELECTION_MASK

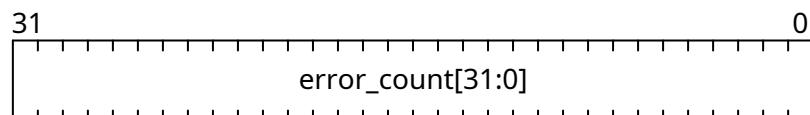


Fig. 20.52: READER_ERROR_COUNT



Fig. 20.53: READER_SKIP_FIFO

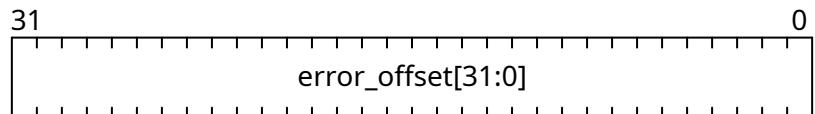


Fig. 20.54: READER_ERROR_OFFSET

READER_ERROR_DATA15

Address: $0xf0003000 + 0x34 = 0xf0003034$

Bits 480-511 of *READER_ERROR_DATA*. Erroneous value read from DRAM memory

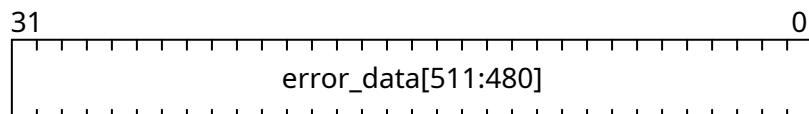


Fig. 20.55: READER_ERROR_DATA15

READER_ERROR_DATA14

Address: $0xf0003000 + 0x38 = 0xf0003038$

Bits 448-479 of *READER_ERROR_DATA*.

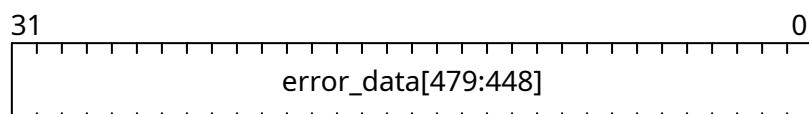


Fig. 20.56: READER_ERROR_DATA14

READER_ERROR_DATA13

Address: $0xf0003000 + 0x3c = 0xf000303c$

Bits 416-447 of *READER_ERROR_DATA*.

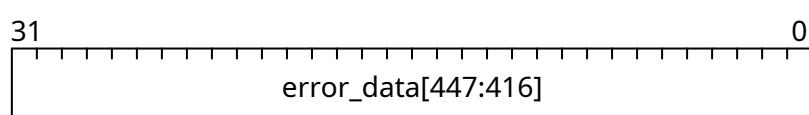


Fig. 20.57: READER_ERROR_DATA13

READER_ERROR_DATA12

Address: $0xf0003000 + 0x40 = 0xf0003040$

Bits 384-415 of *READER_ERROR_DATA*.

READER_ERROR_DATA11

Address: $0xf0003000 + 0x44 = 0xf0003044$

Bits 352-383 of *READER_ERROR_DATA*.

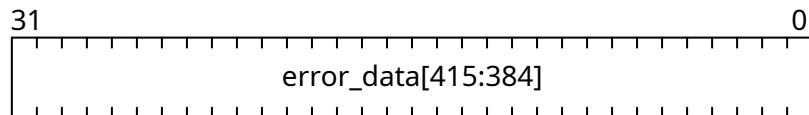


Fig. 20.58: READER_ERROR_DATA12

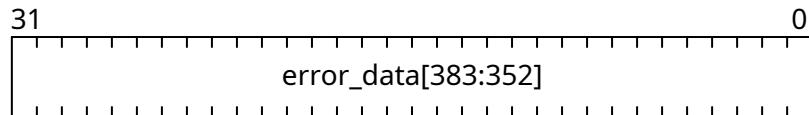


Fig. 20.59: READER_ERROR_DATA11

READER_ERROR_DATA10

Address: $0xf0003000 + 0x48 = 0xf0003048$

Bits 320-351 of *READER_ERROR_DATA*.

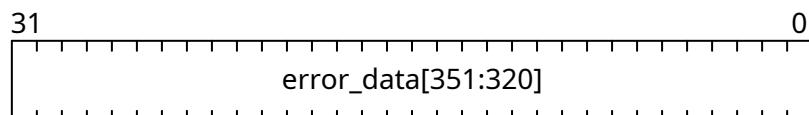


Fig. 20.60: READER_ERROR_DATA10

READER_ERROR_DATA9

Address: $0xf0003000 + 0x4c = 0xf000304c$

Bits 288-319 of *READER_ERROR_DATA*.

READER_ERROR_DATA8

Address: $0xf0003000 + 0x50 = 0xf0003050$

Bits 256-287 of *READER_ERROR_DATA*.

READER_ERROR_DATA7

Address: $0xf0003000 + 0x54 = 0xf0003054$

Bits 224-255 of *READER_ERROR_DATA*.

READER_ERROR_DATA6

Address: $0xf0003000 + 0x58 = 0xf0003058$

Bits 192-223 of *READER_ERROR_DATA*.

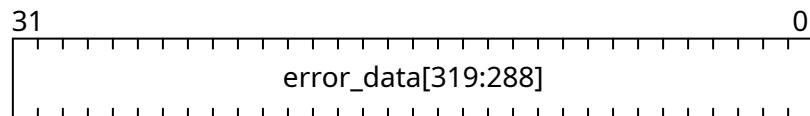


Fig. 20.61: READER_ERROR_DATA9

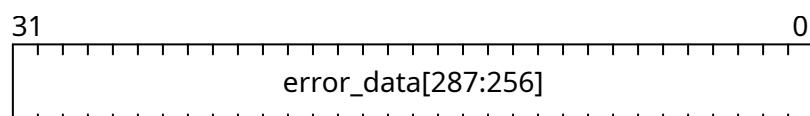


Fig. 20.62: READER_ERROR_DATA8

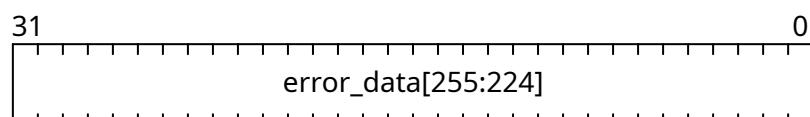


Fig. 20.63: READER_ERROR_DATA7

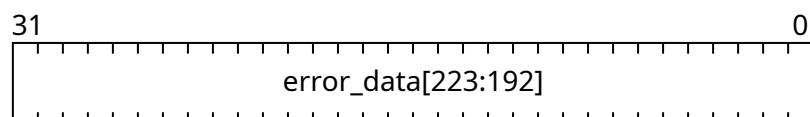


Fig. 20.64: READER_ERROR_DATA6

READER_ERROR_DATA5

Address: $0xf0003000 + 0x5c = 0xf000305c$

Bits 160-191 of *READER_ERROR_DATA*.

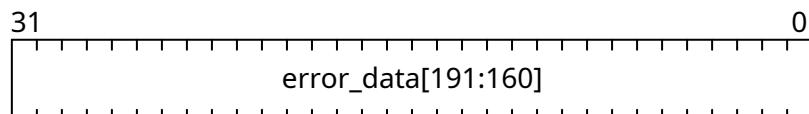


Fig. 20.65: *READER_ERROR_DATA5*

READER_ERROR_DATA4

Address: $0xf0003000 + 0x60 = 0xf0003060$

Bits 128-159 of *READER_ERROR_DATA*.

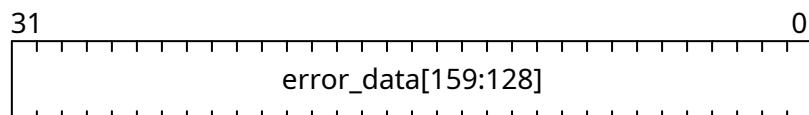


Fig. 20.66: *READER_ERROR_DATA4*

READER_ERROR_DATA3

Address: $0xf0003000 + 0x64 = 0xf0003064$

Bits 96-127 of *READER_ERROR_DATA*.

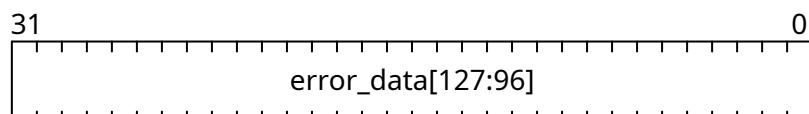


Fig. 20.67: *READER_ERROR_DATA3*

READER_ERROR_DATA2

Address: $0xf0003000 + 0x68 = 0xf0003068$

Bits 64-95 of *READER_ERROR_DATA*.

READER_ERROR_DATA1

Address: $0xf0003000 + 0x6c = 0xf000306c$

Bits 32-63 of *READER_ERROR_DATA*.

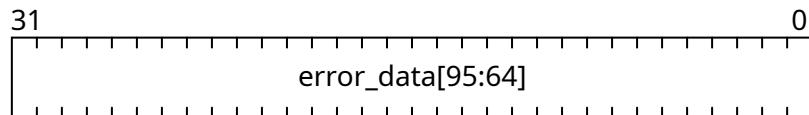


Fig. 20.68: READER_ERROR_DATA2

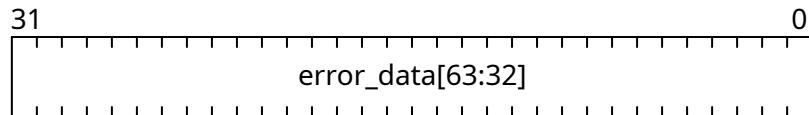


Fig. 20.69: READER_ERROR_DATA1

READER_ERROR_DATA0

Address: $0xf0003000 + 0x70 = 0xf0003070$

Bits 0-31 of *READER_ERROR_DATA*.

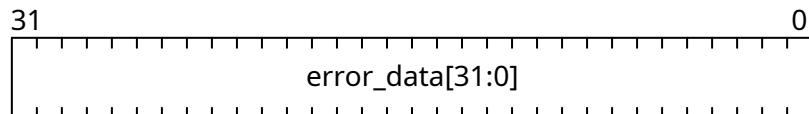


Fig. 20.70: READER_ERROR_DATA0

READER_ERROR_EXPECTED15

Address: $0xf0003000 + 0x74 = 0xf0003074$

Bits 480-511 of *READER_ERROR_EXPECTED*. Value expected to be read from DRAM memory

READER_ERROR_EXPECTED14

Address: $0xf0003000 + 0x78 = 0xf0003078$

Bits 448-479 of *READER_ERROR_EXPECTED*.

READER_ERROR_EXPECTED13

Address: $0xf0003000 + 0x7c = 0xf000307c$

Bits 416-447 of *READER_ERROR_EXPECTED*.

READER_ERROR_EXPECTED12

Address: $0xf0003000 + 0x80 = 0xf0003080$

Bits 384-415 of *READER_ERROR_EXPECTED*.

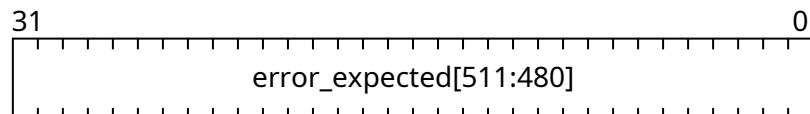


Fig. 20.71: READER_ERROR_EXPECTED15

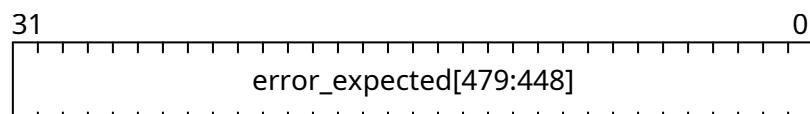


Fig. 20.72: READER_ERROR_EXPECTED14

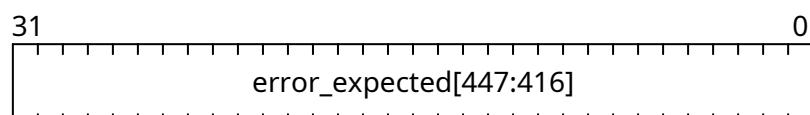


Fig. 20.73: READER_ERROR_EXPECTED13

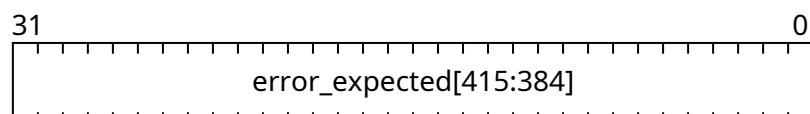


Fig. 20.74: READER_ERROR_EXPECTED12

READER_ERROR_EXPECTED11

Address: $0xf0003000 + 0x84 = 0xf0003084$

Bits 352-383 of *READER_ERROR_EXPECTED*.

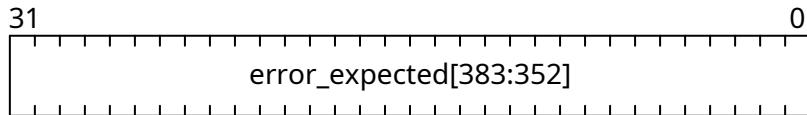


Fig. 20.75: READER_ERROR_EXPECTED11

READER_ERROR_EXPECTED10

Address: $0xf0003000 + 0x88 = 0xf0003088$

Bits 320-351 of *READER_ERROR_EXPECTED*.

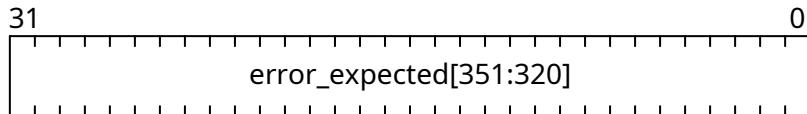


Fig. 20.76: READER_ERROR_EXPECTED10

READER_ERROR_EXPECTED9

Address: $0xf0003000 + 0x8c = 0xf000308c$

Bits 288-319 of *READER_ERROR_EXPECTED*.

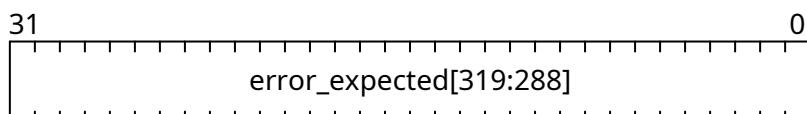


Fig. 20.77: READER_ERROR_EXPECTED9

READER_ERROR_EXPECTED8

Address: $0xf0003000 + 0x90 = 0xf0003090$

Bits 256-287 of *READER_ERROR_EXPECTED*.

READER_ERROR_EXPECTED7

Address: $0xf0003000 + 0x94 = 0xf0003094$

Bits 224-255 of *READER_ERROR_EXPECTED*.

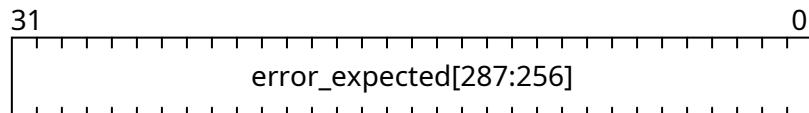


Fig. 20.78: READER_ERROR_EXPECTED8

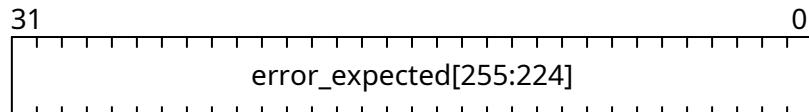


Fig. 20.79: READER_ERROR_EXPECTED7

READER_ERROR_EXPECTED6

Address: $0xf0003000 + 0x98 = 0xf0003098$

Bits 192-223 of *READER_ERROR_EXPECTED*.

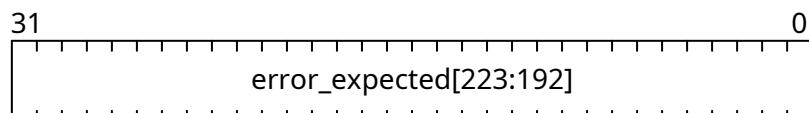


Fig. 20.80: READER_ERROR_EXPECTED6

READER_ERROR_EXPECTED5

Address: $0xf0003000 + 0x9c = 0xf000309c$

Bits 160-191 of *READER_ERROR_EXPECTED*.

READER_ERROR_EXPECTED4

Address: $0xf0003000 + 0xa0 = 0xf00030a0$

Bits 128-159 of *READER_ERROR_EXPECTED*.

READER_ERROR_EXPECTED3

Address: $0xf0003000 + 0xa4 = 0xf00030a4$

Bits 96-127 of *READER_ERROR_EXPECTED*.

READER_ERROR_EXPECTED2

Address: $0xf0003000 + 0xa8 = 0xf00030a8$

Bits 64-95 of *READER_ERROR_EXPECTED*.

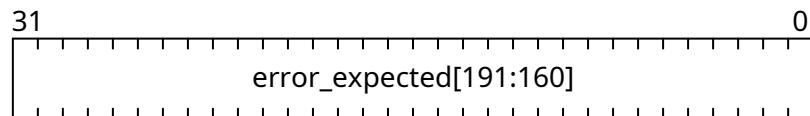


Fig. 20.81: READER_ERROR_EXPECTED5

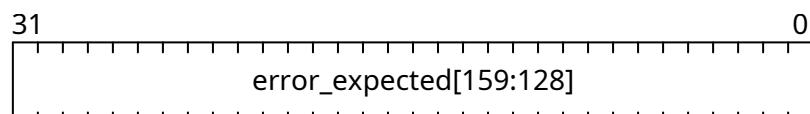


Fig. 20.82: READER_ERROR_EXPECTED4

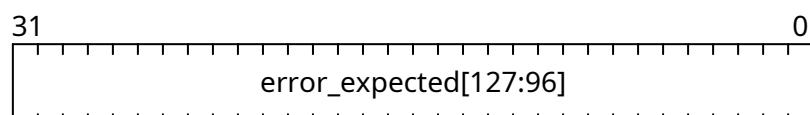


Fig. 20.83: READER_ERROR_EXPECTED3

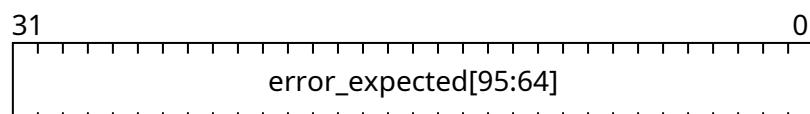


Fig. 20.84: READER_ERROR_EXPECTED2

READER_ERROR_EXPECTED1

Address: $0xf0003000 + 0xac = 0xf00030ac$

Bits 32-63 of *READER_ERROR_EXPECTED*.

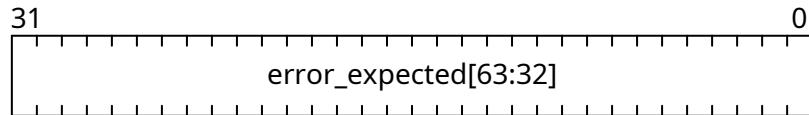


Fig. 20.85: READER_ERROR_EXPECTED1

READER_ERROR_EXPECTED0

Address: $0xf0003000 + 0xb0 = 0xf00030b0$

Bits 0-31 of *READER_ERROR_EXPECTED*.

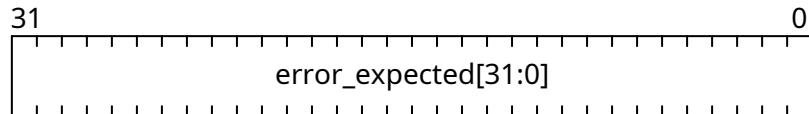


Fig. 20.86: READER_ERROR_EXPECTED0

READER_ERROR_READY

Address: $0xf0003000 + 0xb4 = 0xf00030b4$

Error detected and ready to read

READER_ERROR_CONTINUE

Address: $0xf0003000 + 0xb8 = 0xf00030b8$

Continue reading until the next error

20.2.8 DFI_SWITCH

Register Listing for DFI_SWITCH

Register	Address
<i>DFI_SWITCH_REFRESH_COUNT1</i>	<i>0xf0003800</i>
<i>DFI_SWITCH_REFRESH_COUNT0</i>	<i>0xf0003804</i>
<i>DFI_SWITCH_AT_REFRESH1</i>	<i>0xf0003808</i>
<i>DFI_SWITCH_AT_REFRESH0</i>	<i>0xf000380c</i>
<i>DFI_SWITCH_REFRESH_UPDATE</i>	<i>0xf0003810</i>

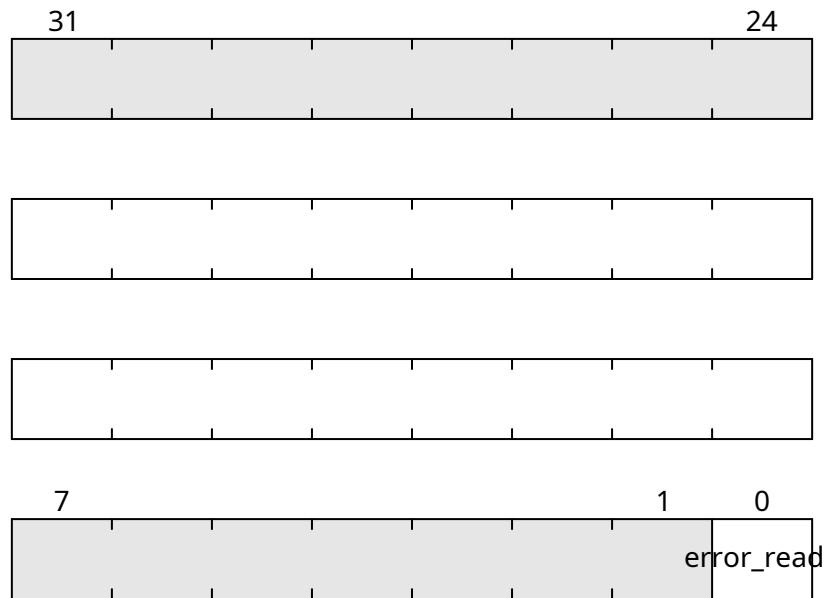


Fig. 20.87: READER_ERROR_READY

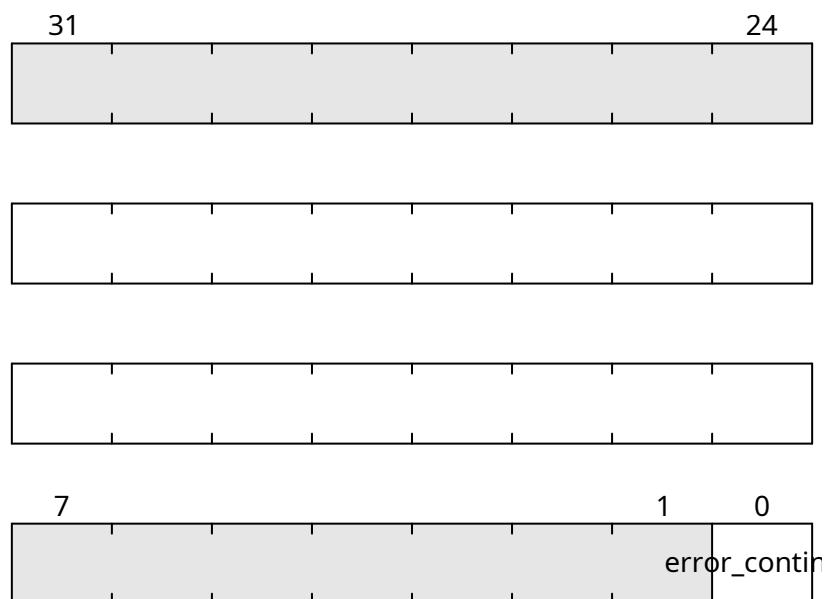


Fig. 20.88: READER_ERROR_CONTINUE

DFI_SWITCH_REFRESH_COUNT1

Address: $0xf0003800 + 0x0 = 0xf0003800$

Bits 32-63 of *DFI_SWITCH_REFRESH_COUNT*. Count of all refresh commands issued (both by Memory Controller and the Payload Executor). Value is latched from internal counter on mode transition: MC -> PE or by writing to the *refresh_update* CSR.

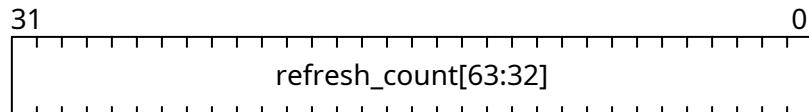


Fig. 20.89: DFI_SWITCH_REFRESH_COUNT1

DFI_SWITCH_REFRESH_COUNT0

Address: $0xf0003800 + 0x4 = 0xf0003804$

Bits 0-31 of *DFI_SWITCH_REFRESH_COUNT*.

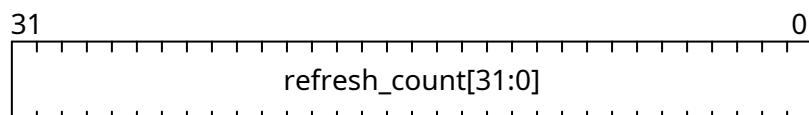


Fig. 20.90: DFI_SWITCH_REFRESH_COUNT0

DFI_SWITCH_AT_REFRESH1

Address: $0xf0003800 + 0x8 = 0xf0003808$

Bits 32-63 of *DFI_SWITCH_AT_REFRESH*. If set to a value different than 0 the mode transition MC -> PE will be performed only when the value of this register matches the current refresh commands count.

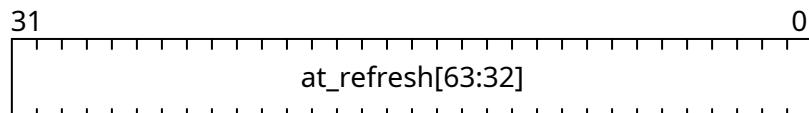


Fig. 20.91: DFI_SWITCH_AT_REFRESH1

DFI_SWITCH_AT_REFRESH0

Address: $0xf0003800 + 0xc = 0xf000380c$

Bits 0-31 of *DFI_SWITCH_AT_REFRESH*.

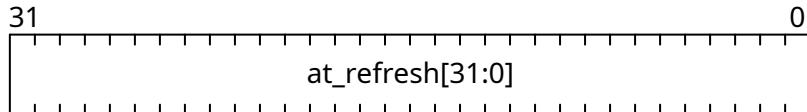


Fig. 20.92: DFI_SWITCH_AT_REFRESH0

DFI_SWITCH_REFRESH_UPDATE

Address: $0xf0003800 + 0x10 = 0xf0003810$

Force an update of the *refresh_count* CSR.

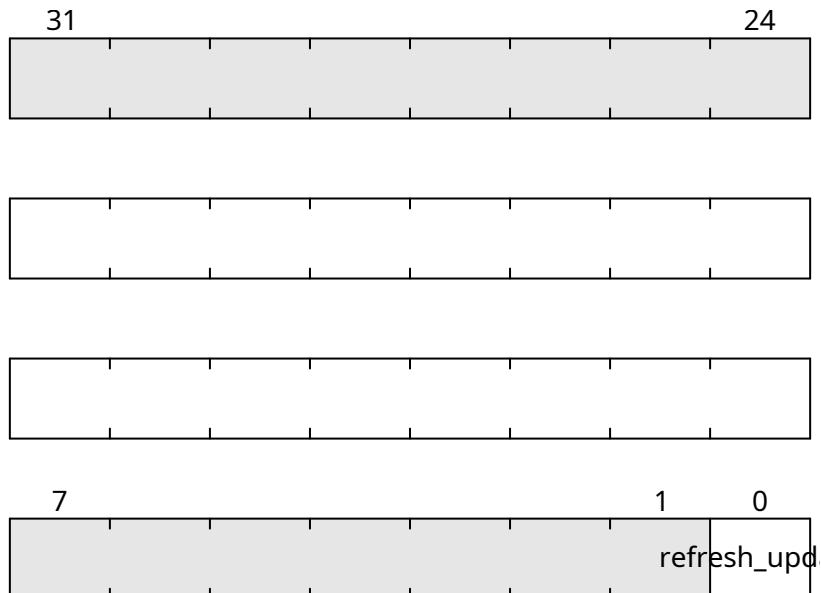


Fig. 20.93: DFI_SWITCH_REFRESH_UPDATE

20.2.9 PAYLOAD_EXECUTOR

Executes the DRAM payload from memory

Instruction decoder

All instructions are 32-bit. The format of most instructions is the same, except for the LOOP instruction, which has a constant TIMESLICE of 1.

NOOP with a TIMESLICE of 0 is a special case which is interpreted as STOP instruction. When this instruction is encountered execution gets finished immediately.

NOTE: TIMESLICE is the number of cycles the instruction will take. This means that instructions other than NOOP that use TIMESLICE=0 are illegal (although will silently be executed as having TIMESLICE=1).

NOTE2: LOOP instruction will *jump* COUNT times, meaning that the “code” inside the loop will effectively be executed COUNT+1 times.

Op codes:

Op	Value
NOOP	0b000
LOOP	0b111
ACT	0b100
PRE	0b101
REF	0b110
ZQC	0b001
READ	0b010

Instruction format:

LSB	MSB
dfi: OP_CODE TIMESLICE ADDRESS	
noop: OP_CODE TIMESLICE_NOOP	
loop: OP_CODE COUNT JUMP	
stop: <NOOP> 0	

Where ADDRESS depends on the DFI command and is one of:

LSB	MSB
RANK BANK COLUMN	
RANK BANK ROW	

Register Listing for PAYLOAD_EXECUTOR

Register	Address
PAYLOAD_EXECUTOR_START	0xf0004000
PAYLOAD_EXECUTOR_STATUS	0xf0004004
PAYLOAD_EXECUTOR_READ_COUNT	0xf0004008
PAYLOAD_EXECUTOR_EXEC_START1	0xf000400c
PAYLOAD_EXECUTOR_EXEC_START0	0xf0004010
PAYLOAD_EXECUTOR_EXEC_STOP1	0xf0004014
PAYLOAD_EXECUTOR_EXEC_STOP0	0xf0004018

PAYLOAD_EXECUTOR_START

Address: $0xf0004000 + 0x0 = 0xf0004000$

Writing to this register initializes payload execution



Fig. 20.94: PAYLOAD_EXECUTOR_START

PAYLOAD_EXECUTOR_STATUS

Address: $0xf0004000 + 0x4 = 0xf0004004$

Payload executor status register

Field	Name	Description
[0]	READY	Indicates that the executor is not running
[1]	OVERFLOW	Indicates the scratchpad memory address counter has overflowed due to the number of READ commands sent during execution

PAYLOAD_EXECUTOR_READ_COUNT

Address: $0xf0004000 + 0x8 = 0xf0004008$

Number of data from READ commands that is stored in the scratchpad memory

PAYLOAD_EXECUTOR_EXEC_START1

Address: $0xf0004000 + 0xc = 0xf000400c$

Bits 32-63 of PAYLOAD_EXECUTOR_EXEC_START. Number of cycles elapsed until the start of the payload execution.

PAYLOAD_EXECUTOR_EXEC_START0

Address: $0xf0004000 + 0x10 = 0xf0004010$

Bits 0-31 of PAYLOAD_EXECUTOR_EXEC_START.

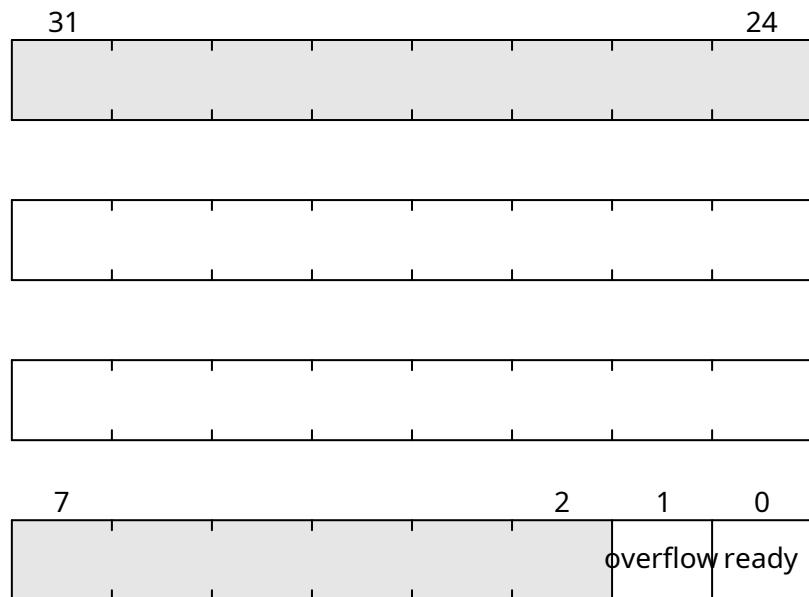


Fig. 20.95: PAYLOAD_EXECUTOR_STATUS

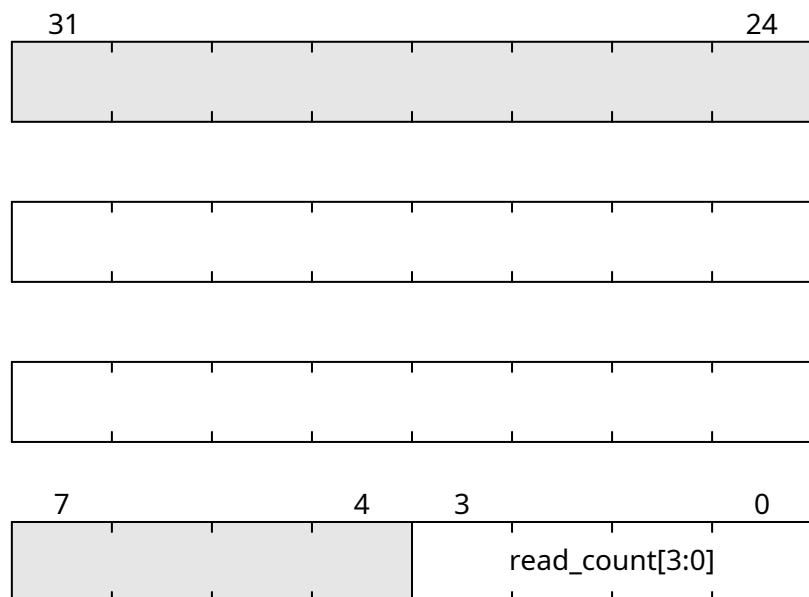


Fig. 20.96: PAYLOAD_EXECUTOR_READ_COUNT

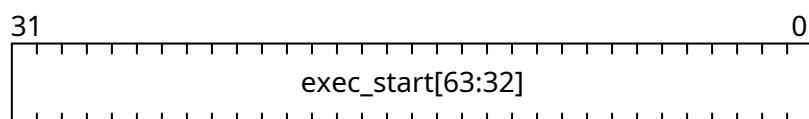


Fig. 20.97: PAYLOAD_EXECUTOR_EXEC_START1

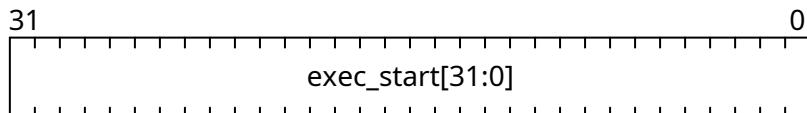


Fig. 20.98: PAYLOAD_EXECUTOR_EXEC_START0

PAYLOAD_EXECUTOR_EXEC_STOP1

Address: $0xf0004000 + 0x14 = 0xf0004014$

Bits 32-63 of PAYLOAD_EXECUTOR_EXEC_STOP. Number of cycles elapsed until the end of the payload execution.

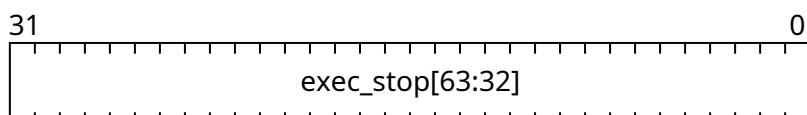


Fig. 20.99: PAYLOAD_EXECUTOR_EXEC_STOP1

PAYLOAD_EXECUTOR_EXEC_STOP0

Address: $0xf0004000 + 0x18 = 0xf0004018$

Bits 0-31 of PAYLOAD_EXECUTOR_EXEC_STOP.

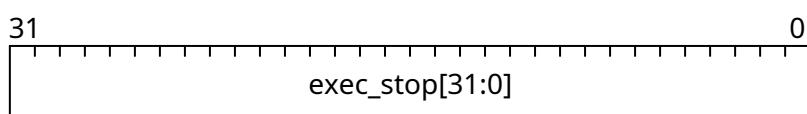


Fig. 20.100: PAYLOAD_EXECUTOR_EXEC_STOP0

20.2.10 I2C

Register Listing for I2C

Register	Address
I2C_W	0xf0004800
I2C_R	0xf0004804
I2C_WORKER	0xf0004808
I2C_I2C_WORKER_START	0xf000480c
I2C_I2C_WORKER_I2C_CTRL	0xf0004810
I2C_I2C_WORKER_I2C_STATE	0xf0004814
I2C_I2C_WORKER_FIFOS_ACCESS_PORT	0xf0004818
I2C_I2C_WORKER_WRITE_FIFO_STATE	0xf000481c
I2C_I2C_WORKER_READ_FIFO_STATE	0xf0004820

I2C_W

Address: $0xf0004800 + 0x0 = 0xf0004800$

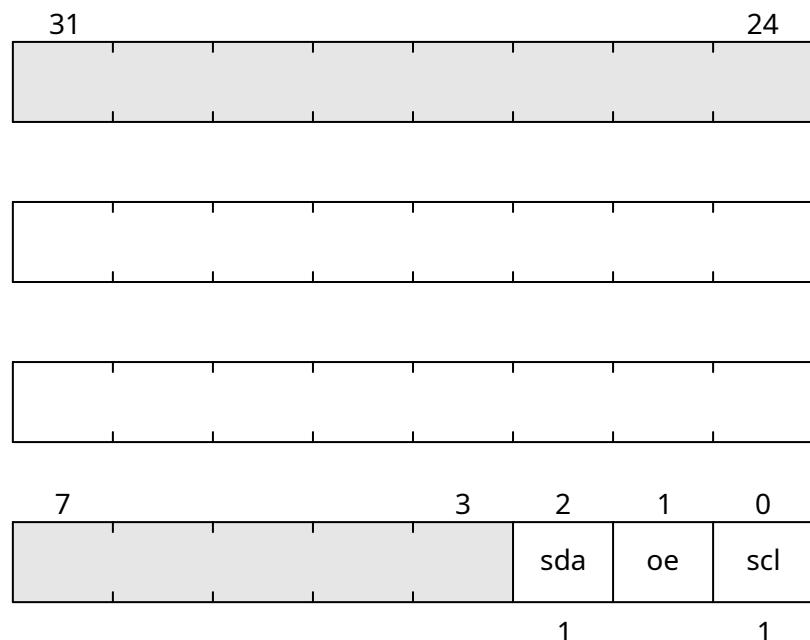


Fig. 20.101: I2C_W

Field	Name	Description

I2C_R

Address: $0xf0004800 + 0x4 = 0xf0004804$

Field	Name	Description

I2C_WORKER

Address: $0xf0004800 + 0x8 = 0xf0004808$

Field	Name	Description

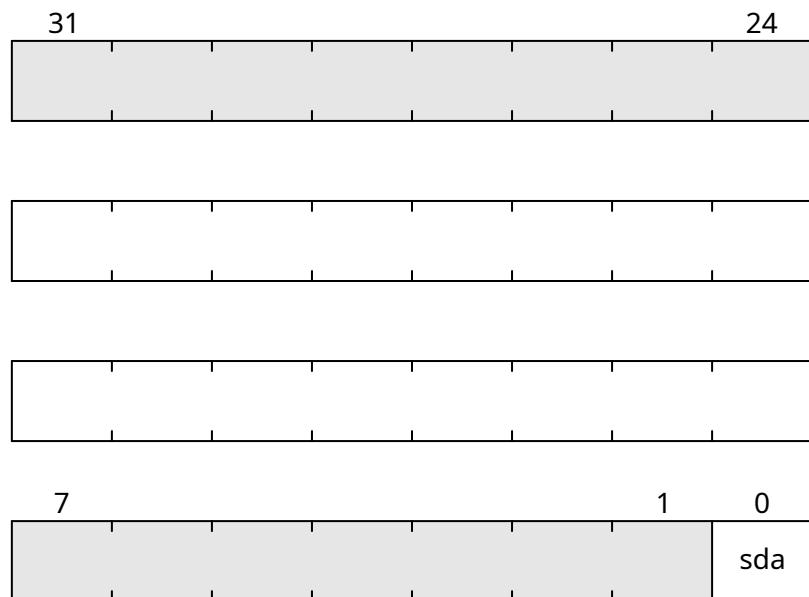


Fig. 20.102: I2C_R

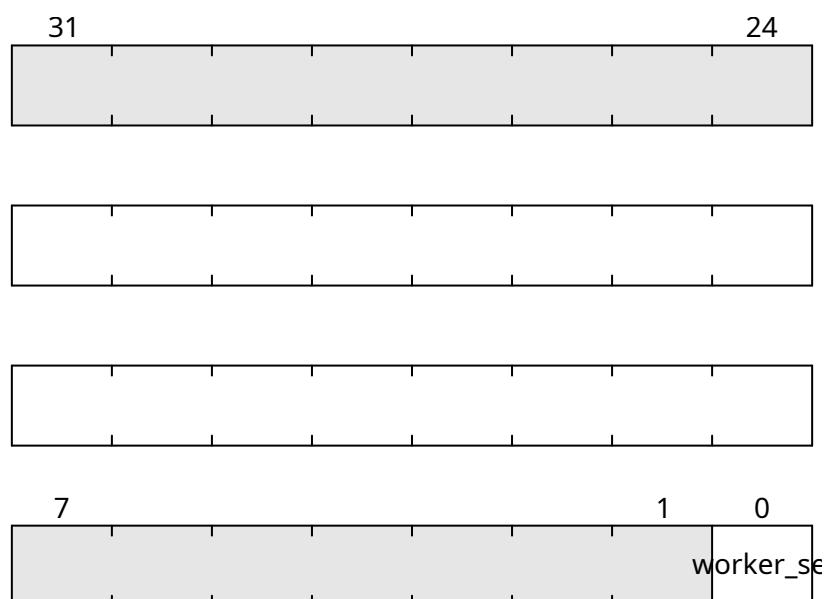


Fig. 20.103: I2C_WORKER

I2C_I2C_WORKER_START

Address: $0xf0004800 + 0xc = 0xf000480c$

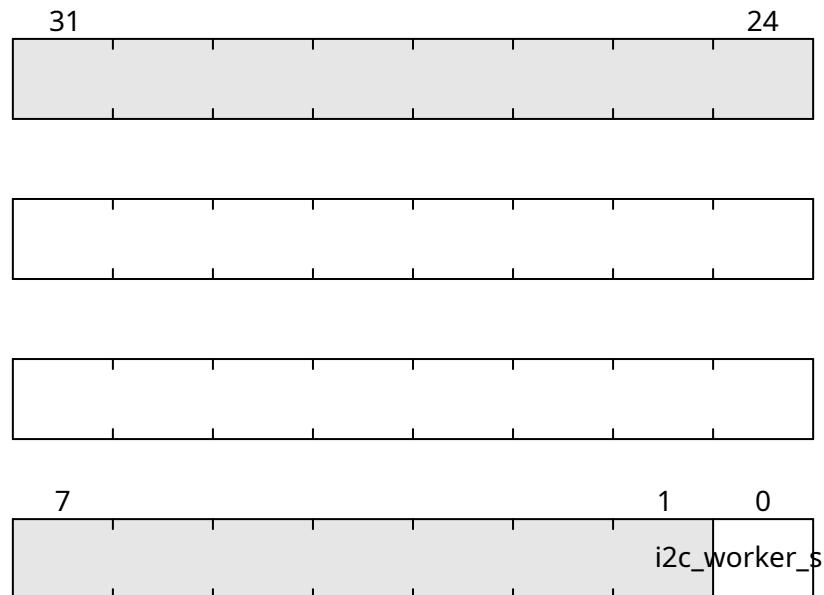


Fig. 20.104: I2C_I2C_WORKER_START

I2C_I2C_WORKER_I2C_CTRL

Address: $0xf0004800 + 0x10 = 0xf0004810$

Field	Name	Description

I2C_I2C_WORKER_I2C_STATE

Address: $0xf0004800 + 0x14 = 0xf0004814$

Field	Name	Description



Fig. 20.105: I2C_I2C_WORKER_I2C_CTRL

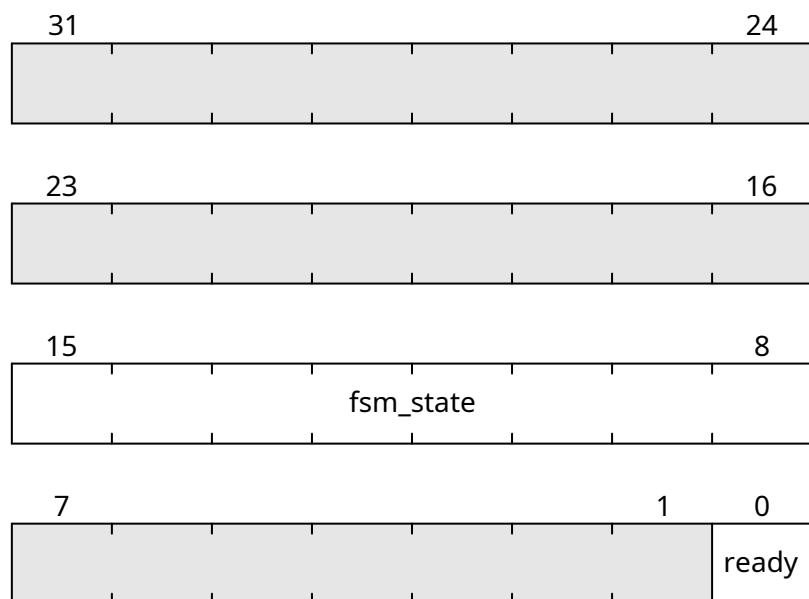


Fig. 20.106: I2C_I2C_WORKER_I2C_STATE

I2C_I2C_WORKER_FIFO_ACCESS_PORT

Address: $0xf0004800 + 0x18 = 0xf0004818$

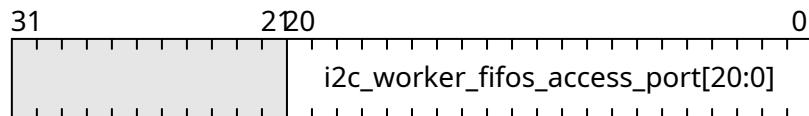


Fig. 20.107: I2C_I2C_WORKER_FIFO_ACCESS_PORT

I2C_I2C_WORKER_WRITE_FIFO_STATE

Address: $0xf0004800 + 0x1c = 0xf000481c$

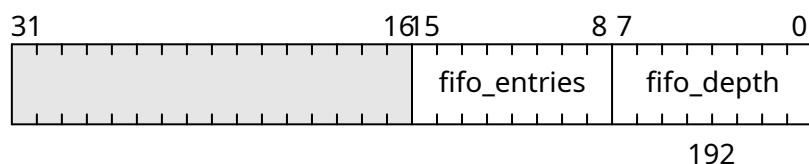


Fig. 20.108: I2C_I2C_WORKER_WRITE_FIFO_STATE

Field	Name	Description

I2C_I2C_WORKER_READ_FIFO_STATE

Address: $0xf0004800 + 0x20 = 0xf0004820$

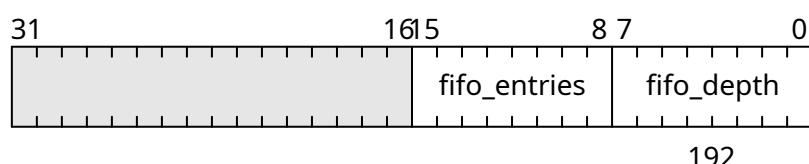


Fig. 20.109: I2C_I2C_WORKER_READ_FIFO_STATE

Field	Name	Description

20.2.11 CTRL

Register Listing for CTRL

Register	Address
<i>CTRL_RESET</i>	0xf0005000
<i>CTRL_SCRATCH</i>	0xf0005004
<i>CTRL_BUS_ERRORS</i>	0xf0005008

CTRL__RESET

Address: $0xf0005000 + 0x0 = 0xf0005000$

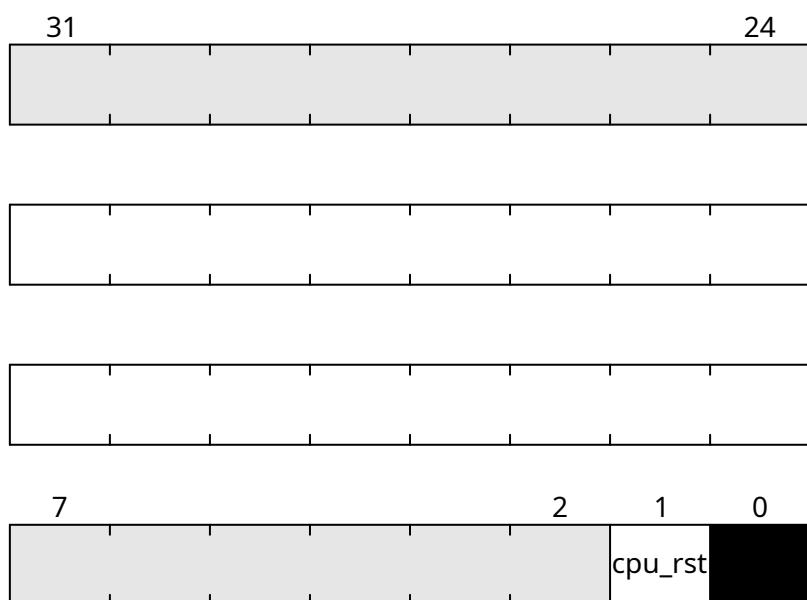


Fig. 20.110: CTRL_RESET

Field	Name	Description
[0]	SOC_RST	Write 1 to this register to reset the full SoC (Pulse Reset)
[1]	CPU_RST	Write 1 to this register to reset the CPU(s) of the SoC (Hold Reset)

CTRL_SCRATCH

Address: $0xf0005000 + 0x4 = 0xf0005004$

Use this register as a scratch space to verify that software read/write accesses to the Wishbone/CSR bus are working correctly. The initial reset value of 0x1234578 can be used to verify endianness.

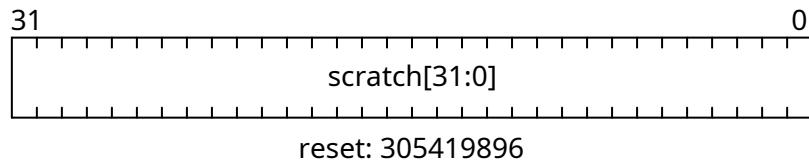


Fig. 20.111: CTRL_SCRATCH

CTRL_BUS_ERRORS

Address: $0xf0005000 + 0x8 = 0xf0005008$

Total number of Wishbone bus errors (timeouts) since start.

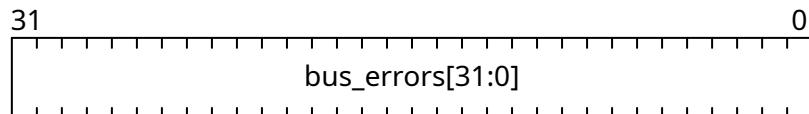


Fig. 20.112: CTRL_BUS_ERRORS

20.2.12 IDENTIFIER_MEM

Register Listing for IDENTIFIER_MEM

Register	Address
<i>IDENTIFIER_MEM</i>	<i>0xf0005800</i>

IDENTIFIER_MEM

Address: $0xf0005800 + 0x0 = 0xf0005800$

8 x 113-bit memory

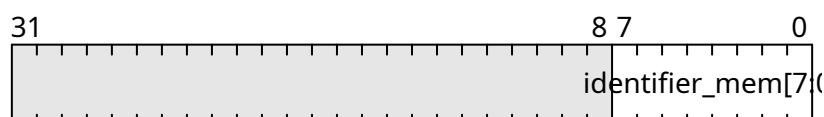


Fig. 20.113: IDENTIFIER_MEM

20.2.13 SDRAM

Register Listing for SDRAM

Register	Address
<i>SDRAM_DFII_CONTROL</i>	<i>0xf0006000</i>
<i>SDRAM_DFII_PIO_COMMAND</i>	<i>0xf0006004</i>

continues on next page

Table 20.2 – continued from previous page

Register	Address
<i>SDRAM_DFII_PIO_COMMAND_ISSUE</i>	0xf0006008
<i>SDRAM_DFII_PIO_ADDRESS</i>	0xf000600c
<i>SDRAM_DFII_PIO_BADDRESS</i>	0xf0006010
<i>SDRAM_DFII_PIO_WRDATA3</i>	0xf0006014
<i>SDRAM_DFII_PIO_WRDATA2</i>	0xf0006018
<i>SDRAM_DFII_PIO_WRDATA1</i>	0xf000601c
<i>SDRAM_DFII_PIO_WRDATA0</i>	0xf0006020
<i>SDRAM_DFII_PIO_RDDATA3</i>	0xf0006024
<i>SDRAM_DFII_PIO_RDDATA2</i>	0xf0006028
<i>SDRAM_DFII_PIO_RDDATA1</i>	0xf000602c
<i>SDRAM_DFII_PIO_RDDATA0</i>	0xf0006030
<i>SDRAM_DFII_PI1_COMMAND</i>	0xf0006034
<i>SDRAM_DFII_PI1_COMMAND_ISSUE</i>	0xf0006038
<i>SDRAM_DFII_PI1_ADDRESS</i>	0xf000603c
<i>SDRAM_DFII_PI1_BADDRESS</i>	0xf0006040
<i>SDRAM_DFII_PI1_WRDATA3</i>	0xf0006044
<i>SDRAM_DFII_PI1_WRDATA2</i>	0xf0006048
<i>SDRAM_DFII_PI1_WRDATA1</i>	0xf000604c
<i>SDRAM_DFII_PI1_WRDATA0</i>	0xf0006050
<i>SDRAM_DFII_PI1_RDDATA3</i>	0xf0006054
<i>SDRAM_DFII_PI1_RDDATA2</i>	0xf0006058
<i>SDRAM_DFII_PI1_RDDATA1</i>	0xf000605c
<i>SDRAM_DFII_PI1_RDDATA0</i>	0xf0006060
<i>SDRAM_DFII_PI2_COMMAND</i>	0xf0006064
<i>SDRAM_DFII_PI2_COMMAND_ISSUE</i>	0xf0006068
<i>SDRAM_DFII_PI2_ADDRESS</i>	0xf000606c
<i>SDRAM_DFII_PI2_BADDRESS</i>	0xf0006070
<i>SDRAM_DFII_PI2_WRDATA3</i>	0xf0006074
<i>SDRAM_DFII_PI2_WRDATA2</i>	0xf0006078
<i>SDRAM_DFII_PI2_WRDATA1</i>	0xf000607c
<i>SDRAM_DFII_PI2_WRDATA0</i>	0xf0006080
<i>SDRAM_DFII_PI2_RDDATA3</i>	0xf0006084
<i>SDRAM_DFII_PI2_RDDATA2</i>	0xf0006088
<i>SDRAM_DFII_PI2_RDDATA1</i>	0xf000608c
<i>SDRAM_DFII_PI2_RDDATA0</i>	0xf0006090
<i>SDRAM_DFII_PI3_COMMAND</i>	0xf0006094
<i>SDRAM_DFII_PI3_COMMAND_ISSUE</i>	0xf0006098
<i>SDRAM_DFII_PI3_ADDRESS</i>	0xf000609c
<i>SDRAM_DFII_PI3_BADDRESS</i>	0xf00060a0
<i>SDRAM_DFII_PI3_WRDATA3</i>	0xf00060a4
<i>SDRAM_DFII_PI3_WRDATA2</i>	0xf00060a8
<i>SDRAM_DFII_PI3_WRDATA1</i>	0xf00060ac
<i>SDRAM_DFII_PI3_WRDATA0</i>	0xf00060b0
<i>SDRAM_DFII_PI3_RDDATA3</i>	0xf00060b4
<i>SDRAM_DFII_PI3_RDDATA2</i>	0xf00060b8
<i>SDRAM_DFII_PI3_RDDATA1</i>	0xf00060bc
<i>SDRAM_DFII_PI3_RDDATA0</i>	0xf00060c0
<i>SDRAM_CONTROLLER_TRP</i>	0xf00060c4

continues on next page

Table 20.2 – continued from previous page

Register	Address
<i>SDRAM_CONTROLLER_TRCD</i>	0xf00060c8
<i>SDRAM_CONTROLLER_TWR</i>	0xf00060cc
<i>SDRAM_CONTROLLER_TWTR</i>	0xf00060d0
<i>SDRAM_CONTROLLER_TREFI</i>	0xf00060d4
<i>SDRAM_CONTROLLER_TRFC</i>	0xf00060d8
<i>SDRAM_CONTROLLER_TFAW</i>	0xf00060dc
<i>SDRAM_CONTROLLER_TCCD</i>	0xf00060e0
<i>SDRAM_CONTROLLER_TCCD_WR</i>	0xf00060e4
<i>SDRAM_CONTROLLER_TRTP</i>	0xf00060e8
<i>SDRAM_CONTROLLER_TRRD</i>	0xf00060ec
<i>SDRAM_CONTROLLER_TRC</i>	0xf00060f0
<i>SDRAM_CONTROLLER_TRAS</i>	0xf00060f4
<i>SDRAM_CONTROLLER_TZQCS</i>	0xf00060f8
<i>SDRAM_CONTROLLER_LAST_ADDR_0</i>	0xf00060fc
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0</i>	0xf0006100
<i>SDRAM_CONTROLLER_LAST_ADDR_1</i>	0xf0006104
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1</i>	0xf0006108
<i>SDRAM_CONTROLLER_LAST_ADDR_2</i>	0xf000610c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2</i>	0xf0006110
<i>SDRAM_CONTROLLER_LAST_ADDR_3</i>	0xf0006114
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3</i>	0xf0006118
<i>SDRAM_CONTROLLER_LAST_ADDR_4</i>	0xf000611c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4</i>	0xf0006120
<i>SDRAM_CONTROLLER_LAST_ADDR_5</i>	0xf0006124
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5</i>	0xf0006128
<i>SDRAM_CONTROLLER_LAST_ADDR_6</i>	0xf000612c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6</i>	0xf0006130
<i>SDRAM_CONTROLLER_LAST_ADDR_7</i>	0xf0006134
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7</i>	0xf0006138

SDRAM_DFII_CONTROL

Address: 0xf0006000 + 0x0 = 0xf0006000

Control DFI signals common to all phases

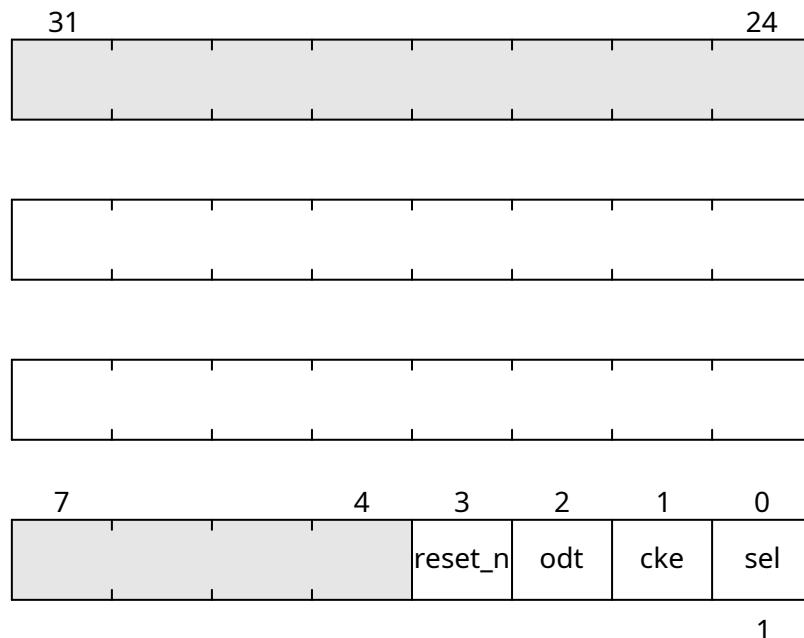


Fig. 20.114: SDRAM_DFII_CONTROL

Field	Name	Description						
[0]	SEL	<table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0b0</td><td>Software control. (CPU)</td></tr> <tr> <td>0b1</td><td>Hardware control (default).</td></tr> </tbody> </table>	Value	Description	0b0	Software control. (CPU)	0b1	Hardware control (default).
Value	Description							
0b0	Software control. (CPU)							
0b1	Hardware control (default).							
[1]	CKE	DFI clock enable bus						
[2]	ODT	DFI on-die termination bus						
[3]	RESET_N	DFI clock reset bus						

SDRAM_DFII_PIO_COMMAND

Address: $0xf0006000 + 0x4 = 0xf0006004$

Control DFI signals on a single phase

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

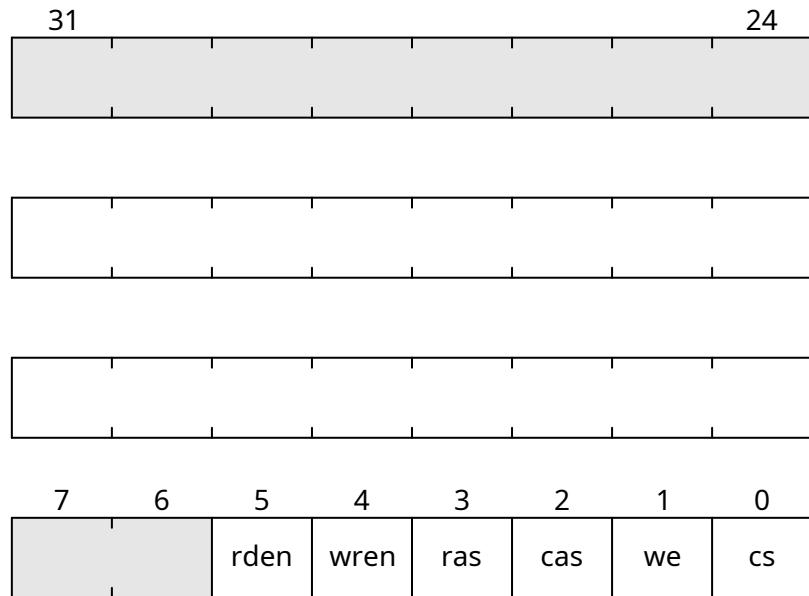


Fig. 20.115: SDRAM DFII PIO COMMAND

SDRAM_DFIIPIO_COMMAND_ISSUE

Address: 0xf0006000 + 0x8 = 0xf0006008

SDRAM_DFII_PIO_ADDRESS

Address: 0xf0006000 + 0xc = 0xf000600c

DFI address bus

SDRAM_DFII_PIO_BADDRESS

Address: $0xf0006000 + 0x10 = 0xf0006010$

DFI bank address bus

SDRAM_DFII_PIO_WRDATA3

Address: $0xf0006000 + 0x14 = 0xf0006014$

Bits 96-127 of *SDRAM_DFII_PIO_WRDATA*. DFI write data bus

SDRAM_DFII_PIO_WRDATA2

Address: $0xf0006000 + 0x18 = 0xf0006018$

Bits 64-95 of *SDRAM_DFII_PIO_WRDATA*.

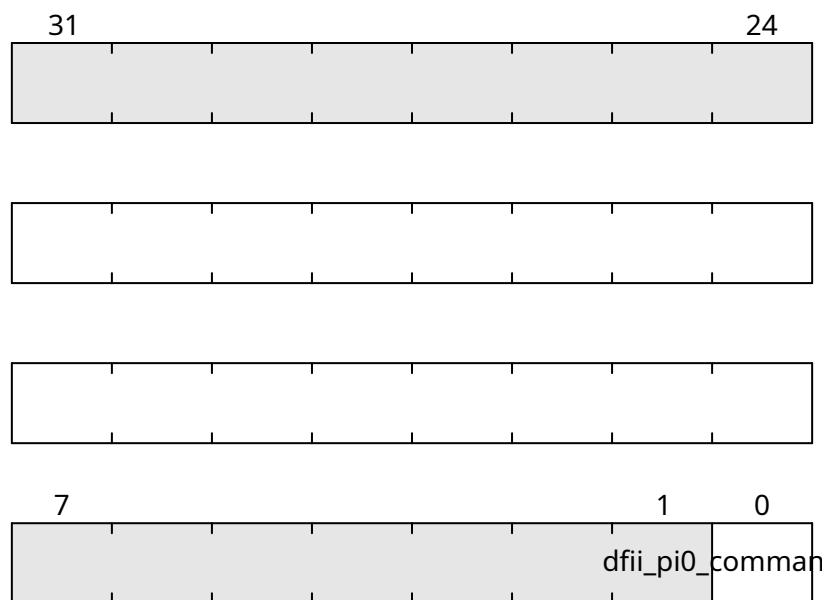


Fig. 20.116: SDRAM_DFII_PI0_COMMAND_ISSUE

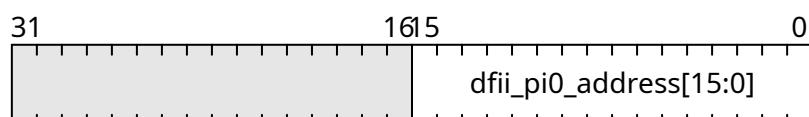


Fig. 20.117: SDRAM_DFII_PIO_ADDRESS

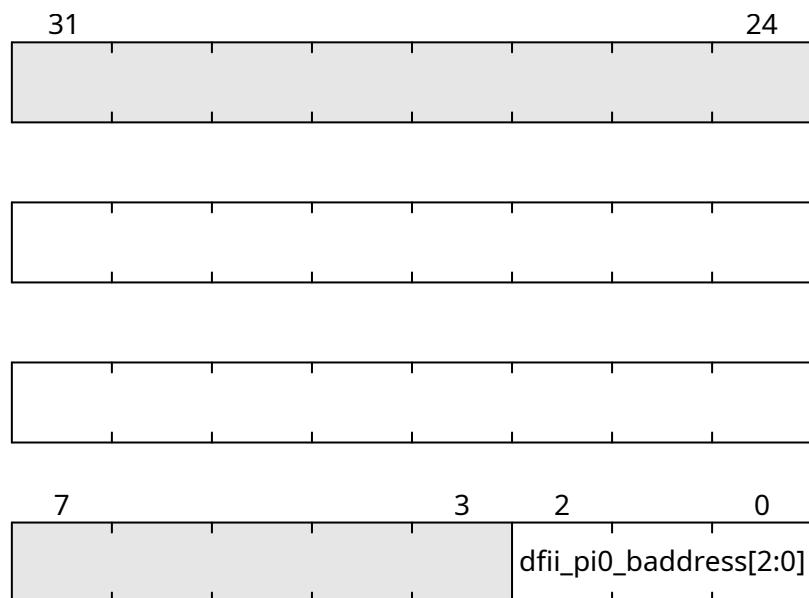


Fig. 20.118: SDRAM_DFII_PIO_BADDRESS

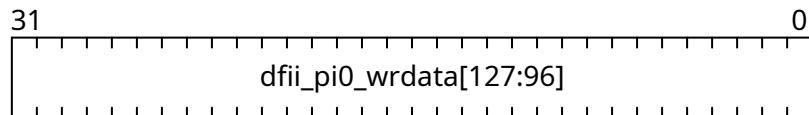


Fig. 20.119: SDRAM_DFII_PIO_WRDATA3

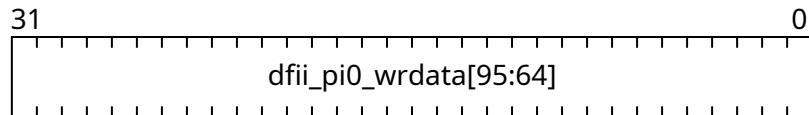


Fig. 20.120: SDRAM_DFII_PIO_WRDATA2

SDRAM_DFII_PIO_WRDATA1

Address: $0xf0006000 + 0x1c = 0xf000601c$

Bits 32-63 of *SDRAM_DFII_PIO_WRDATA*.

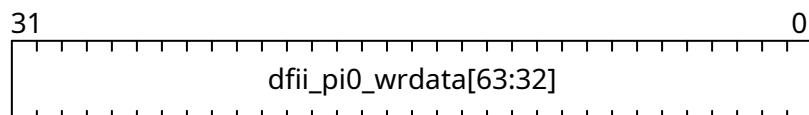


Fig. 20.121: SDRAM_DFII_PIO_WRDATA1

SDRAM_DFII_PIO_WRDATA0

Address: $0xf0006000 + 0x20 = 0xf0006020$

Bits 0-31 of *SDRAM_DFII_PIO_WRDATA*.

SDRAM_DFII_PIO_RDDATA3

Address: $0xf0006000 + 0x24 = 0xf0006024$

Bits 96-127 of *SDRAM_DFII_PIO_RDDATA*. DFI read data bus

SDRAM_DFII_PIO_RDDATA2

Address: $0xf0006000 + 0x28 = 0xf0006028$

Bits 64-95 of *SDRAM_DFII_PIO_RDDATA*.

SDRAM_DFII_PIO_RDDATA1

Address: $0xf0006000 + 0x2c = 0xf000602c$

Bits 32-63 of *SDRAM_DFII_PIO_RDDATA*.

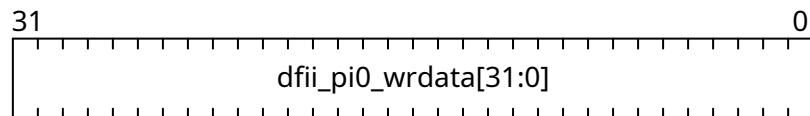


Fig. 20.122: SDRAM_DFII_PI0_WRDATA0

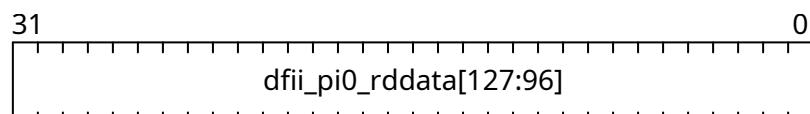


Fig. 20.123: SDRAM_DFII_PI0_RDDATA3

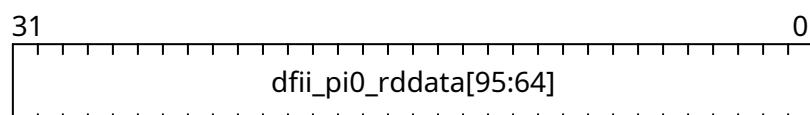


Fig. 20.124: SDRAM_DFII_PI0_RDDATA2

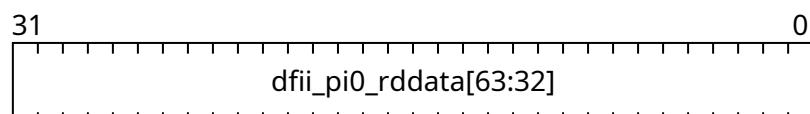


Fig. 20.125: SDRAM_DFII_PI0_RDDATA1

SDRAM_DFII_PIO_RDDATA0

Address: $0xf0006000 + 0x30 = 0xf0006030$

Bits 0-31 of SDRAM_DFII_PIO_RDDATA.

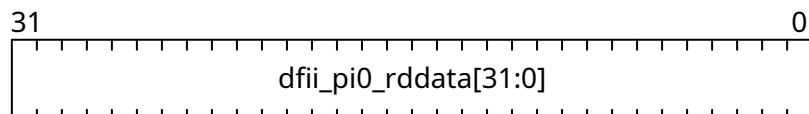


Fig. 20.126: SDRAM_DFII_PIO_RDDATA0

SDRAM_DFII_PI1_COMMAND

Address: $0xf0006000 + 0x34 = 0xf0006034$

Control DFI signals on a single phase

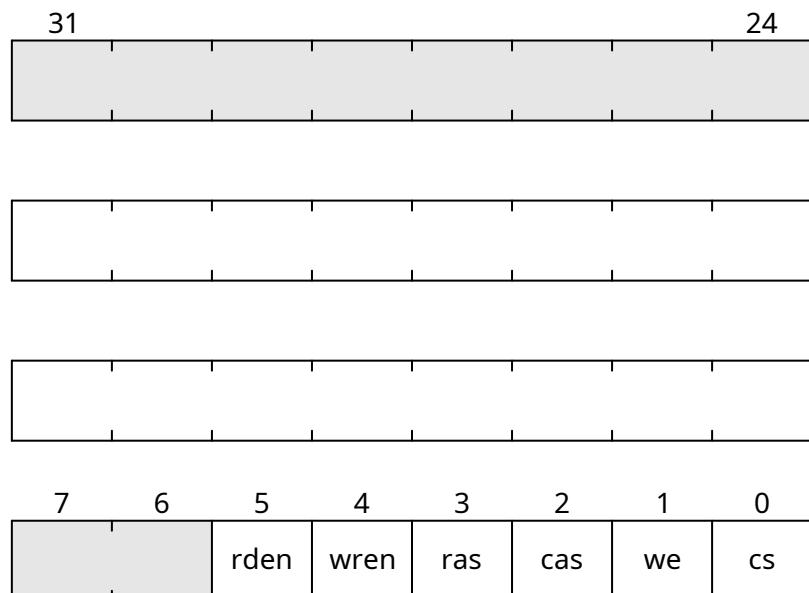


Fig. 20.127: SDRAM_DFII_PI1_COMMAND

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFII_PI1_COMMAND_ISSUE

Address: $0xf0006000 + 0x38 = 0xf0006038$

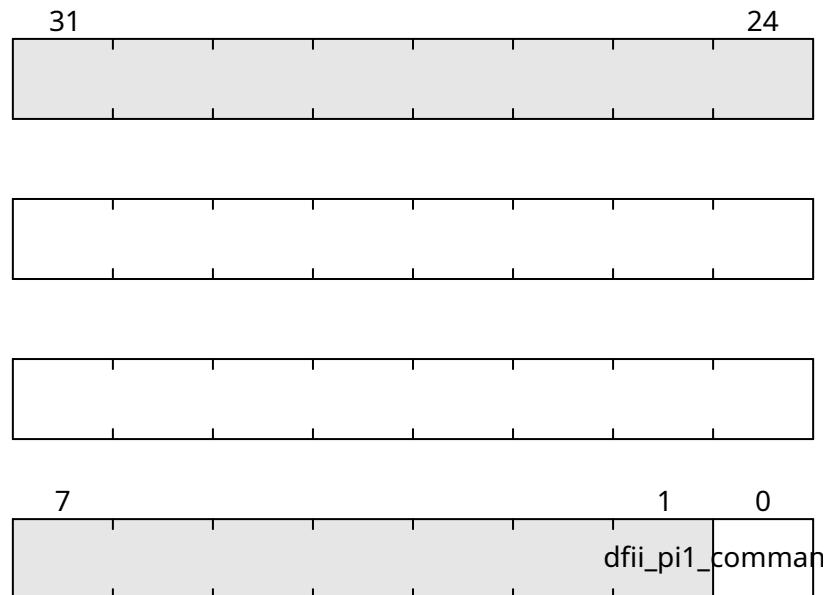


Fig. 20.128: SDRAM_DFII_PI1_COMMAND_ISSUE

SDRAM_DFII_PI1_ADDRESS

Address: $0xf0006000 + 0x3c = 0xf000603c$

DFI address bus

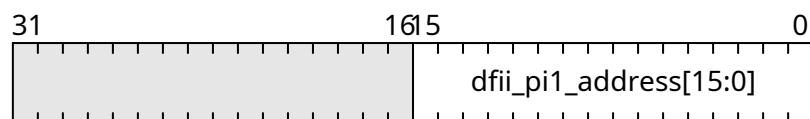


Fig. 20.129: SDRAM_DFII_PI1_ADDRESS

SDRAM_DFII_PI1_BADDRESS

Address: $0xf0006000 + 0x40 = 0xf0006040$

DFI bank address bus

SDRAM_DFII_PI1_WRDATA3

Address: $0xf0006000 + 0x44 = 0xf0006044$

Bits 96-127 of *SDRAM_DFII_PI1_WRDATA*. DFI write data bus

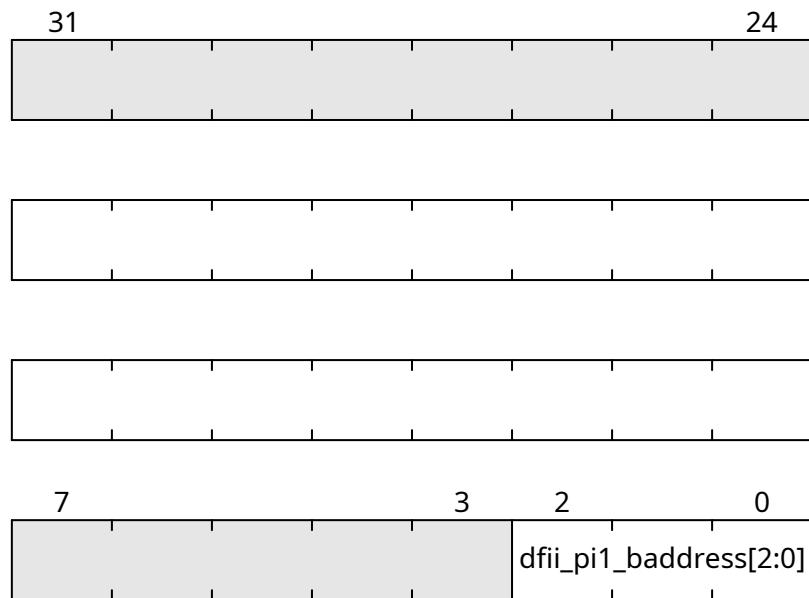


Fig. 20.130: SDRAM_DFII_PI1_BADDRESS

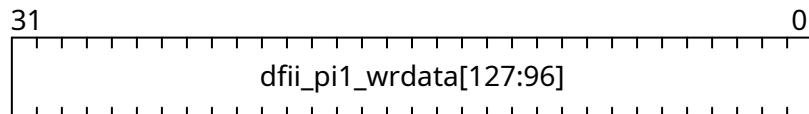


Fig. 20.131: SDRAM_DFII_PI1_WRDATA3

SDRAM_DFII_PI1_WRDATA2

Address: $0xf0006000 + 0x48 = 0xf0006048$

Bits 64-95 of *SDRAM_DFII_PI1_WRDATA*.

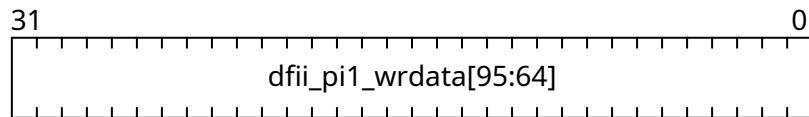


Fig. 20.132: SDRAM_DFII_PI1_WRDATA2

SDRAM_DFII_PI1_WRDATA1

Address: $0xf0006000 + 0x4c = 0xf000604c$

Bits 32-63 of *SDRAM_DFII_PI1_WRDATA*.

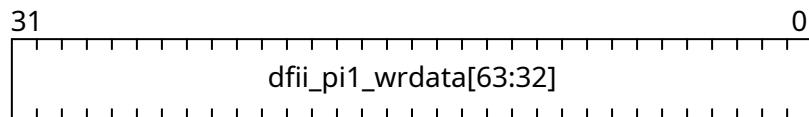


Fig. 20.133: SDRAM_DFII_PI1_WRDATA1

SDRAM_DFII_PI1_WRDATA0

Address: $0xf0006000 + 0x50 = 0xf0006050$

Bits 0-31 of *SDRAM_DFII_PI1_WRDATA*.

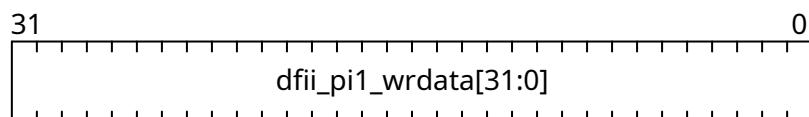


Fig. 20.134: SDRAM_DFII_PI1_WRDATA0

SDRAM_DFII_PI1_RDDATA3

Address: $0xf0006000 + 0x54 = 0xf0006054$

Bits 96-127 of *SDRAM_DFII_PI1_RDDATA*. DFI read data bus

SDRAM_DFII_PI1_RDDATA2

Address: $0xf0006000 + 0x58 = 0xf0006058$

Bits 64-95 of *SDRAM_DFII_PI1_RDDATA*.

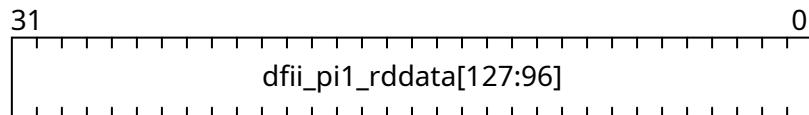


Fig. 20.135: SDRAM_DFII_PI1_RDDATA3

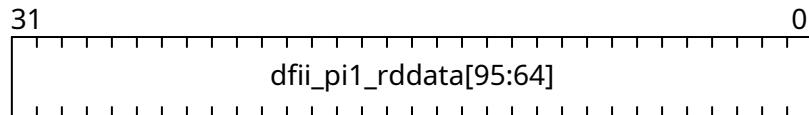


Fig. 20.136: SDRAM_DFII_PI1_RDDATA2

SDRAM_DFII_PI1_RDDATA1

Address: $0xf0006000 + 0x5c = 0xf000605c$

Bits 32-63 of *SDRAM_DFII_PI1_RDDATA*.

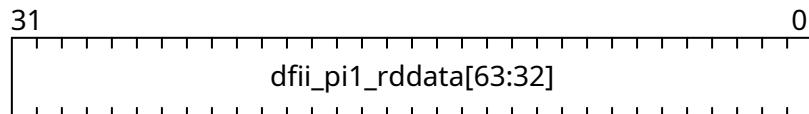


Fig. 20.137: SDRAM_DFII_PI1_RDDATA1

SDRAM_DFII_PI1_RDDATA0

Address: $0xf0006000 + 0x60 = 0xf0006060$

Bits 0-31 of *SDRAM_DFII_PI1_RDDATA*.

SDRAM_DFII_PI2_COMMAND

Address: $0xf0006000 + 0x64 = 0xf0006064$

Control DFI signals on a single phase

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

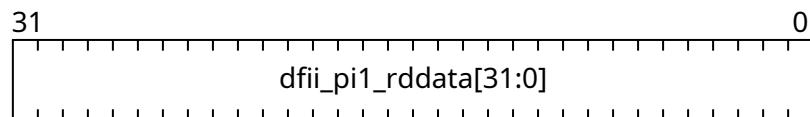


Fig. 20.138: SDRAM_DFII_PI1_RDDATA0

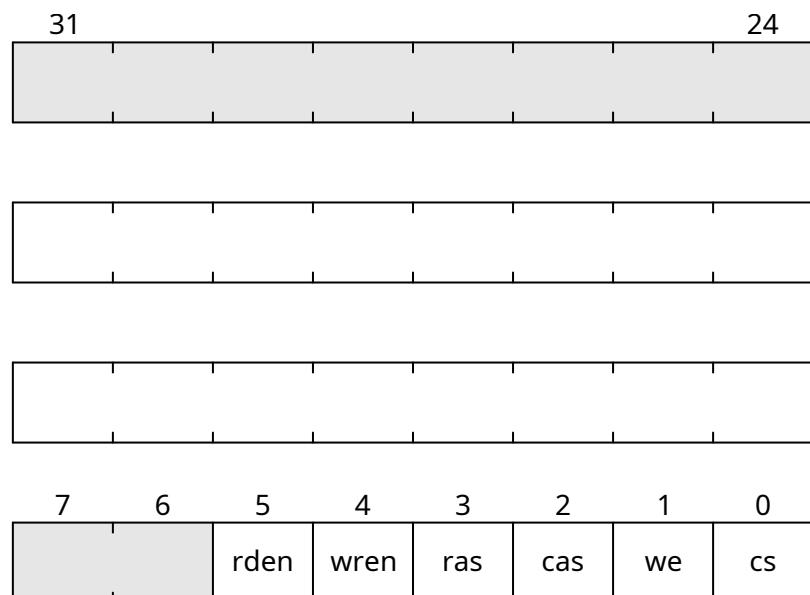


Fig. 20.139: SDRAM_DFII_PI2_COMMAND

SDRAM_DFII_PI2_COMMAND_ISSUE

Address: $0xf0006000 + 0x68 = 0xf0006068$

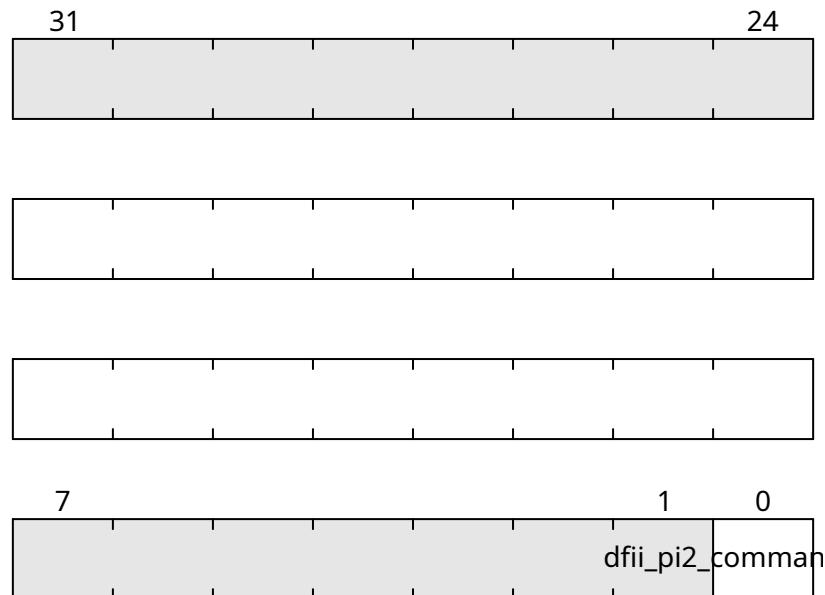


Fig. 20.140: SDRAM_DFII_PI2_COMMAND_ISSUE

SDRAM_DFII_PI2_ADDRESS

Address: $0xf0006000 + 0x6c = 0xf000606c$

DFI address bus

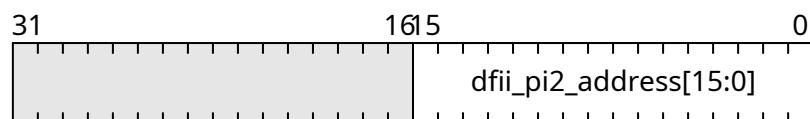


Fig. 20.141: SDRAM_DFII_PI2_ADDRESS

SDRAM_DFII_PI2_BADDRESS

Address: $0xf0006000 + 0x70 = 0xf0006070$

DFI bank address bus

SDRAM_DFII_PI2_WRDATA3

Address: $0xf0006000 + 0x74 = 0xf0006074$

Bits 96-127 of *SDRAM_DFII_PI2_WRDATA*. DFI write data bus



Fig. 20.142: SDRAM_DFII_PI2_BADDRESS

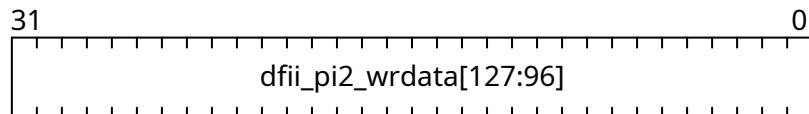


Fig. 20.143: SDRAM_DFII_PI2_WRDATA3

SDRAM_DFII_PI2_WRDATA2

Address: $0xf0006000 + 0x78 = 0xf0006078$

Bits 64-95 of *SDRAM_DFII_PI2_WRDATA*.

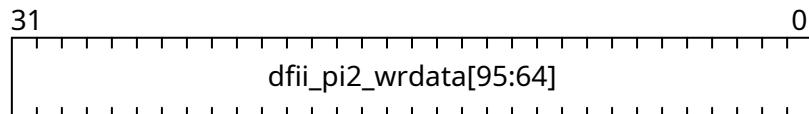


Fig. 20.144: SDRAM_DFII_PI2_WRDATA2

SDRAM_DFII_PI2_WRDATA1

Address: $0xf0006000 + 0x7c = 0xf000607c$

Bits 32-63 of *SDRAM_DFII_PI2_WRDATA*.

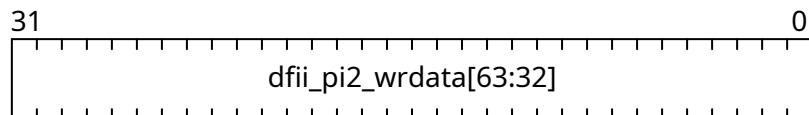


Fig. 20.145: SDRAM_DFII_PI2_WRDATA1

SDRAM_DFII_PI2_WRDATA0

Address: $0xf0006000 + 0x80 = 0xf0006080$

Bits 0-31 of *SDRAM_DFII_PI2_WRDATA*.

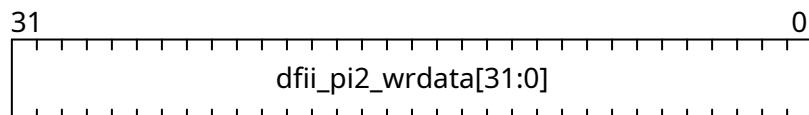


Fig. 20.146: SDRAM_DFII_PI2_WRDATA0

SDRAM_DFII_PI2_RDDATA3

Address: $0xf0006000 + 0x84 = 0xf0006084$

Bits 96-127 of *SDRAM_DFII_PI2_RDDATA*. DFI read data bus

SDRAM_DFII_PI2_RDDATA2

Address: $0xf0006000 + 0x88 = 0xf0006088$

Bits 64-95 of *SDRAM_DFII_PI2_RDDATA*.

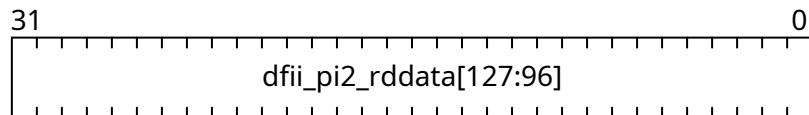


Fig. 20.147: SDRAM_DFII_PI2_RDDATA3

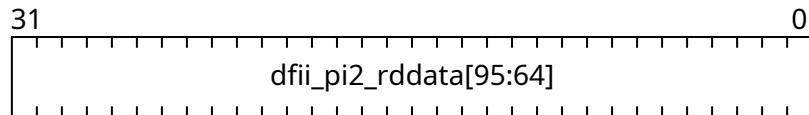


Fig. 20.148: SDRAM_DFII_PI2_RDDATA2

SDRAM_DFII_PI2_RDDATA1

Address: $0xf0006000 + 0x8c = 0xf000608c$

Bits 32-63 of *SDRAM_DFII_PI2_RDDATA*.

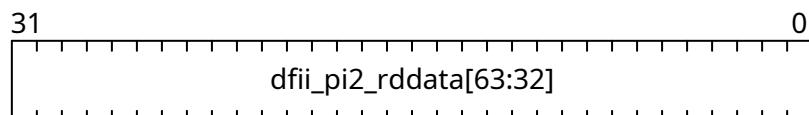


Fig. 20.149: SDRAM_DFII_PI2_RDDATA1

SDRAM_DFII_PI2_RDDATA0

Address: $0xf0006000 + 0x90 = 0xf0006090$

Bits 0-31 of *SDRAM_DFII_PI2_RDDATA*.

SDRAM_DFII_PI3_COMMAND

Address: $0xf0006000 + 0x94 = 0xf0006094$

Control DFI signals on a single phase

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

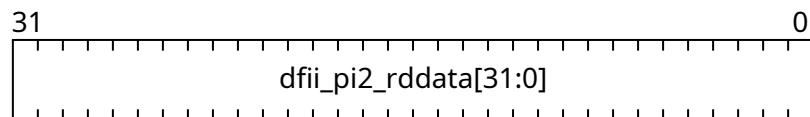


Fig. 20.150: SDRAM_DFII_PI2_RDDATA0

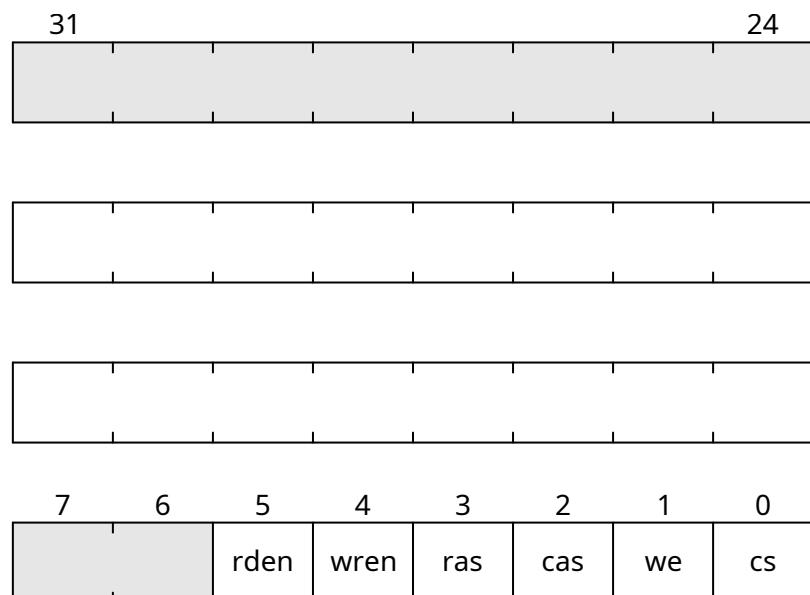


Fig. 20.151: SDRAM_DFII_PI3_COMMAND

SDRAM_DFII_PI3_COMMAND_ISSUE

Address: $0xf0006000 + 0x98 = 0xf0006098$



Fig. 20.152: SDRAM_DFII_PI3_COMMAND_ISSUE

SDRAM_DFII_PI3_ADDRESS

Address: $0xf0006000 + 0x9c = 0xf000609c$

DFI address bus

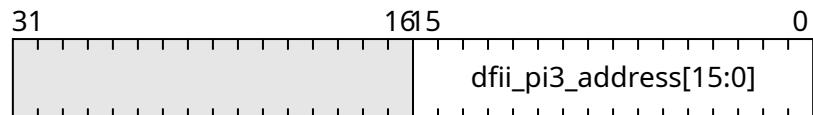


Fig. 20.153: SDRAM DFII PI3 ADDRESS

SDRAM DFII PI3 BADDRESS

Address: 0xf0006000 + 0xa0 = 0xf00060a0

DFI bank address bus

SDRAM DFII PI3 WRDATA3

Address: $0xf0006000 + 0xa4 = 0xf00060a4$

Bits 96-127 of *SDRAM DFII PI3 WRDATA*. DFI write data bus

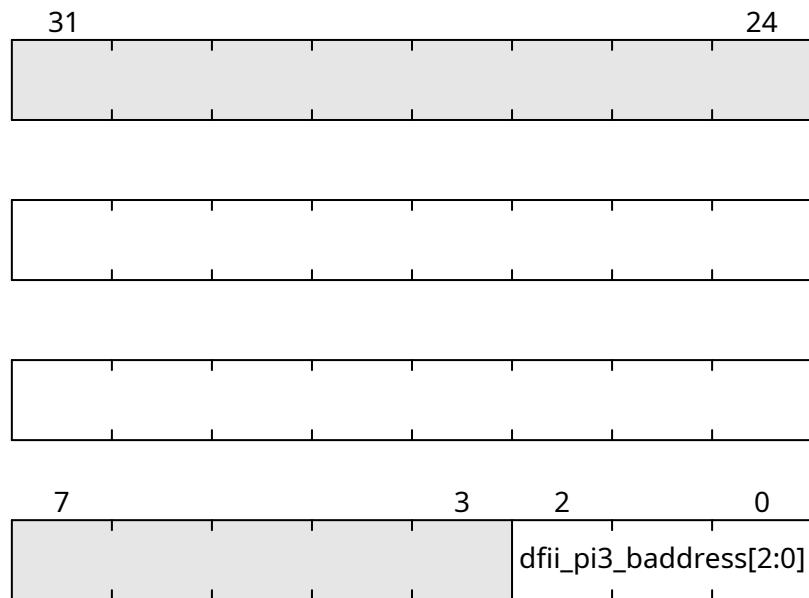


Fig. 20.154: SDRAM_DFII_PI3_BADDRESS

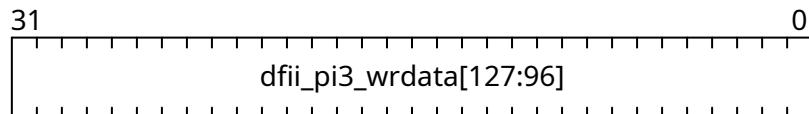


Fig. 20.155: SDRAM_DFII_PI3_WRDATA3

SDRAM_DFII_PI3_WRDATA2

Address: $0xf0006000 + 0xa8 = 0xf00060a8$

Bits 64-95 of *SDRAM_DFII_PI3_WRDATA*.

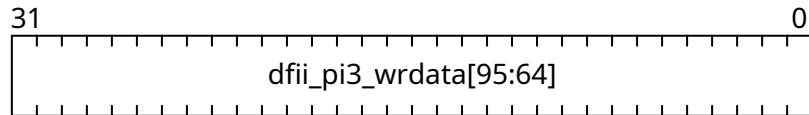


Fig. 20.156: SDRAM_DFII_PI3_WRDATA2

SDRAM_DFII_PI3_WRDATA1

Address: $0xf0006000 + 0xac = 0xf00060ac$

Bits 32-63 of *SDRAM_DFII_PI3_WRDATA*.

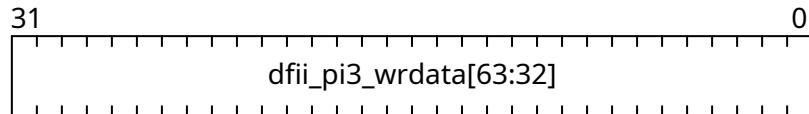


Fig. 20.157: SDRAM_DFII_PI3_WRDATA1

SDRAM_DFII_PI3_WRDATA0

Address: $0xf0006000 + 0xb0 = 0xf00060b0$

Bits 0-31 of *SDRAM_DFII_PI3_WRDATA*.

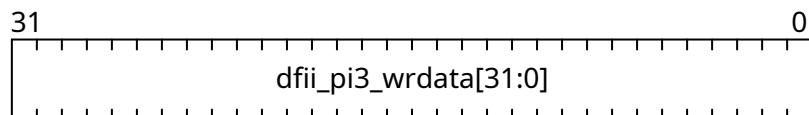


Fig. 20.158: SDRAM_DFII_PI3_WRDATA0

SDRAM_DFII_PI3_RDDATA3

Address: $0xf0006000 + 0xb4 = 0xf00060b4$

Bits 96-127 of *SDRAM_DFII_PI3_RDDATA*. DFI read data bus

SDRAM_DFII_PI3_RDDATA2

Address: $0xf0006000 + 0xb8 = 0xf00060b8$

Bits 64-95 of *SDRAM_DFII_PI3_RDDATA*.

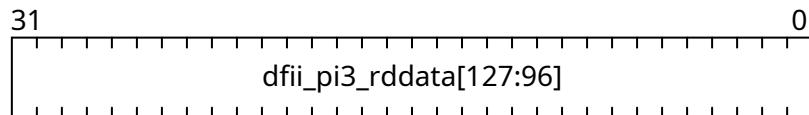


Fig. 20.159: SDRAM_DFII_PI3_RDDATA3

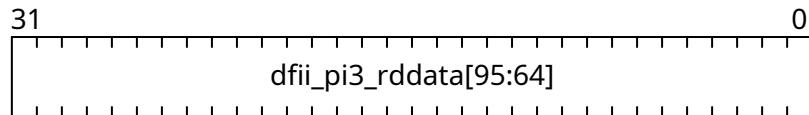


Fig. 20.160: SDRAM_DFII_PI3_RDDATA2

SDRAM_DFII_PI3_RDDATA1

Address: $0xf0006000 + 0xbc = 0xf00060bc$

Bits 32-63 of *SDRAM_DFII_PI3_RDDATA*.

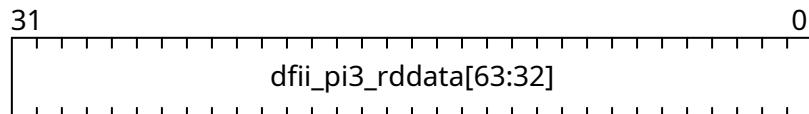


Fig. 20.161: SDRAM_DFII_PI3_RDDATA1

SDRAM_DFII_PI3_RDDATA0

Address: $0xf0006000 + 0xc0 = 0xf00060c0$

Bits 0-31 of *SDRAM_DFII_PI3_RDDATA*.

SDRAM_CONTROLLER_TRP

Address: $0xf0006000 + 0xc4 = 0xf00060c4$

SDRAM_CONTROLLER_TRCD

Address: $0xf0006000 + 0xc8 = 0xf00060c8$

SDRAM_CONTROLLER_TWR

Address: $0xf0006000 + 0xcc = 0xf00060cc$

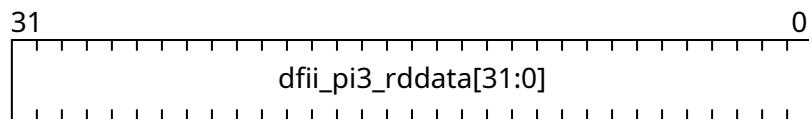


Fig. 20.162: SDRAM_DFII_PI3_RDDATA0

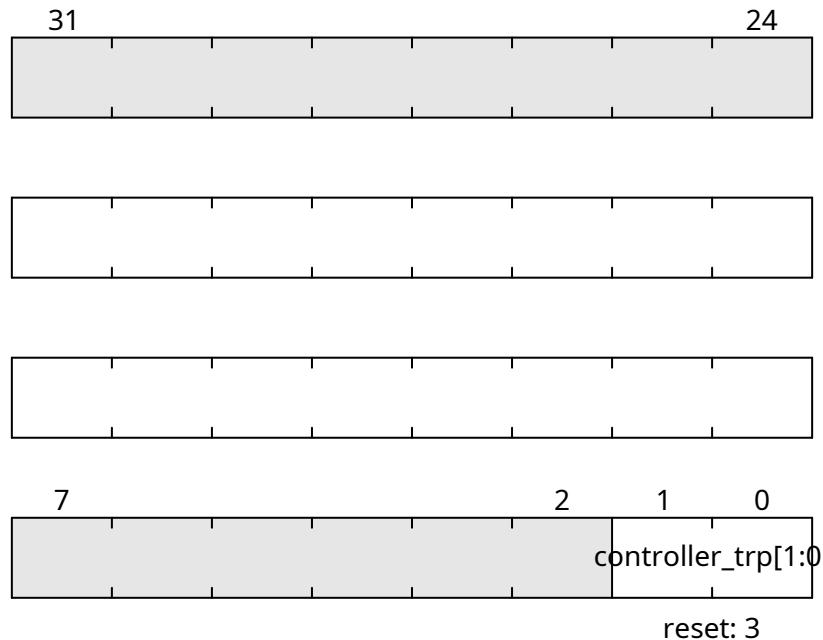


Fig. 20.163: SDRAM_CONTROLLER_TRP

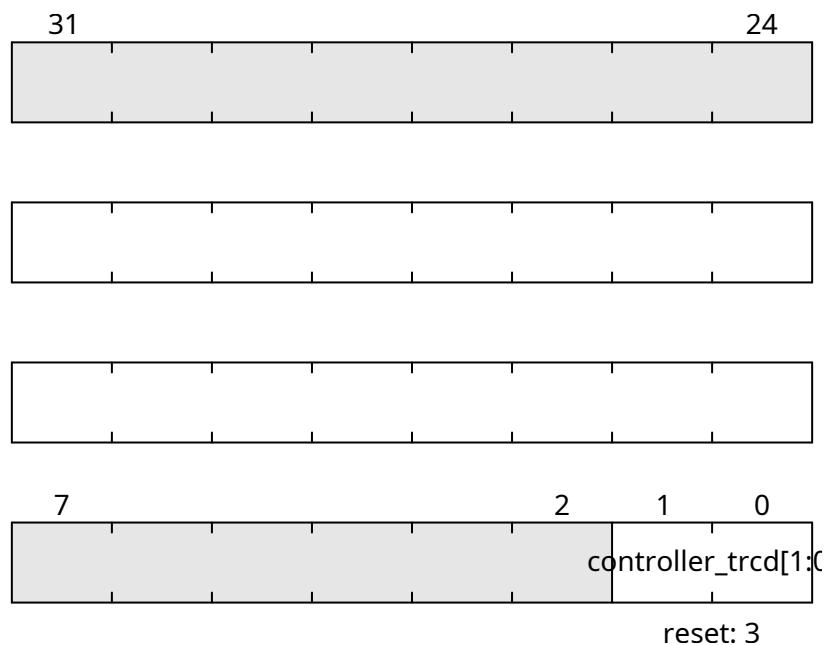


Fig. 20.164: SDRAM_CONTROLLER_TRCD

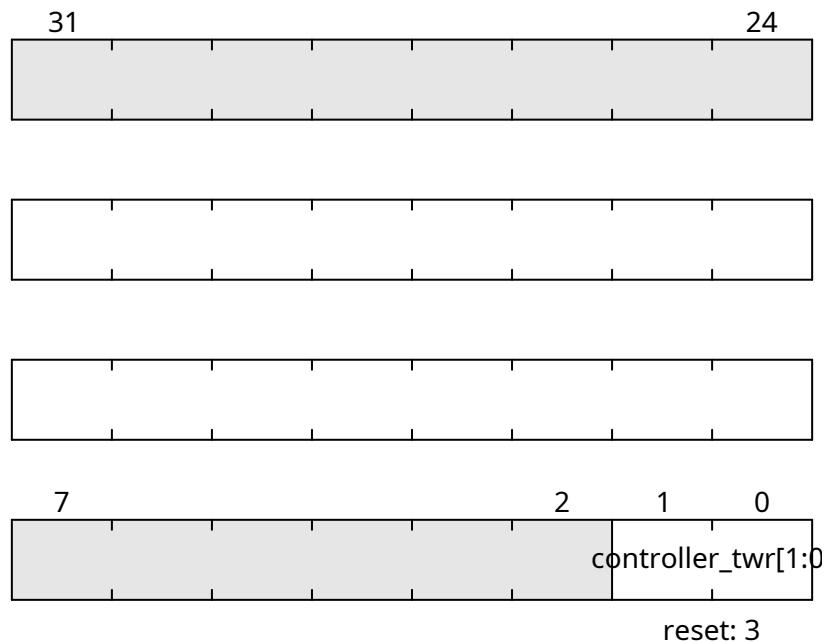


Fig. 20.165: SDRAM_CONTROLLER_TWR

SDRAM_CONTROLLER_TWTR

Address: 0xf0006000 + 0xd0 = 0xf00060d0

SDRAM_CONTROLLER_TREFI

Address: 0xf0006000 + 0xd4 = 0xf00060d4

SDRAM_CONTROLLER_TRFC

Address: 0xf0006000 + 0xd8 = 0xf00060d8

SDRAM_CONTROLLER_TFAW

Address: 0xf0006000 + 0xdc = 0xf00060dc

SDRAM_CONTROLLER_TCCD

Address: 0xf0006000 + 0xe0 = 0xf00060e0

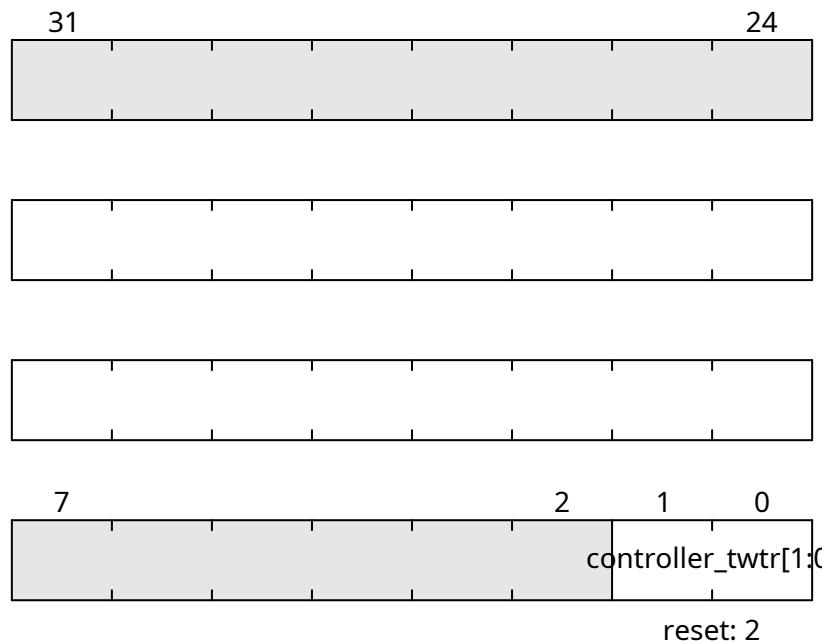


Fig. 20.166: SDRAM_CONTROLLER_TWTR

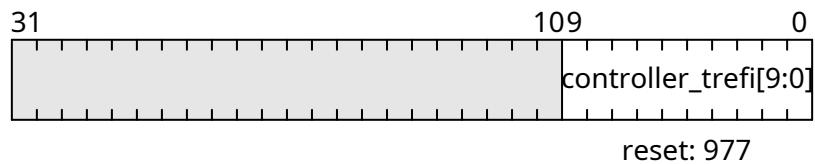


Fig. 20.167: SDRAM_CONTROLLER_TREFI

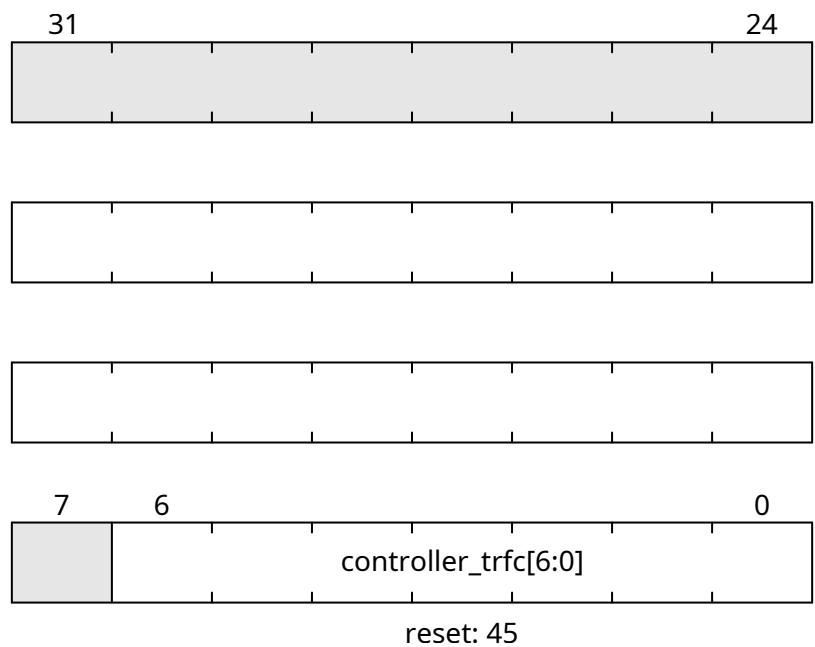


Fig. 20.168: SDRAM_CONTROLLER_TRFC

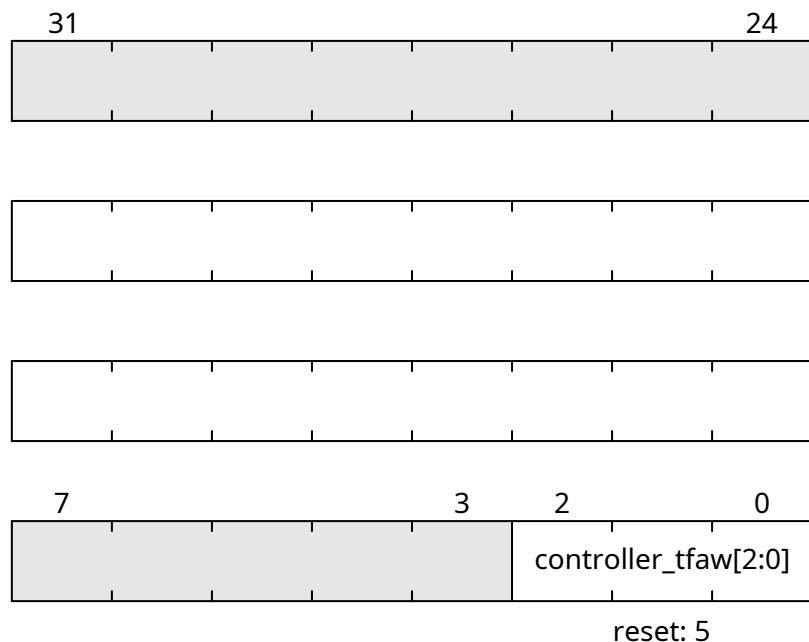


Fig. 20.169: SDRAM_CONTROLLER_TFAW

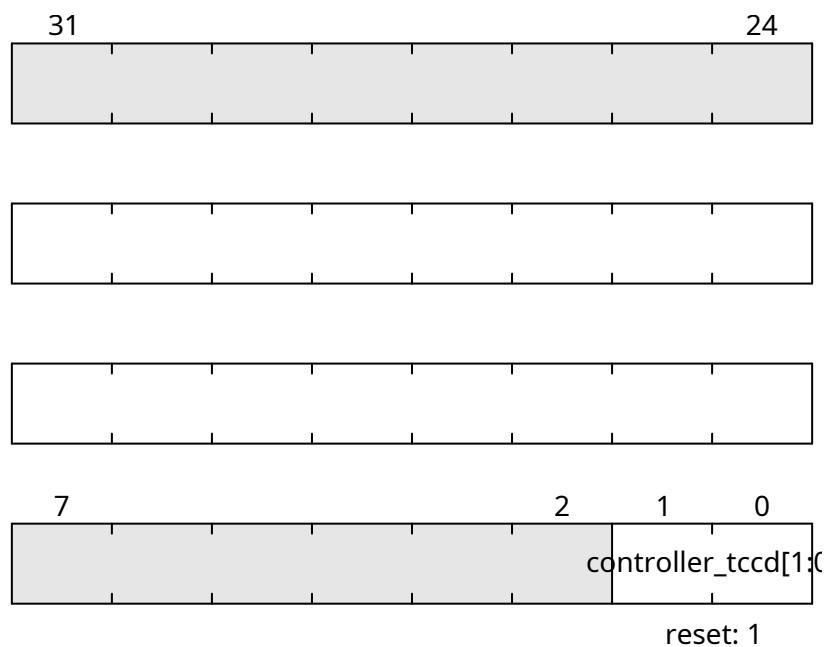


Fig. 20.170: SDRAM_CONTROLLER_TCCD

SDRAM_CONTROLLER_TCCD_WR

Address: $0xf0006000 + 0xe4 = 0xf00060e4$

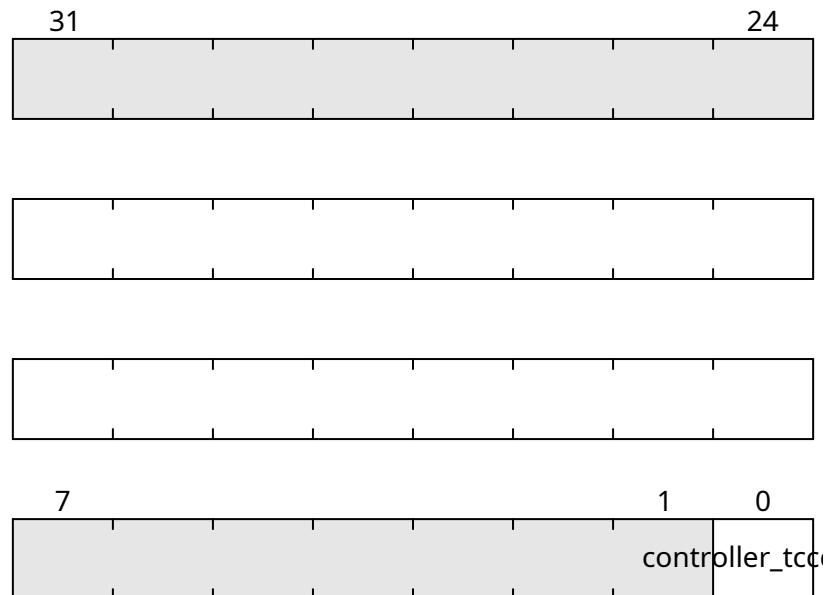


Fig. 20.171: SDRAM_CONTROLLER_TCCD_WR

SDRAM_CONTROLLER_TRTP

Address: $0xf0006000 + 0xe8 = 0xf00060e8$

SDRAM_CONTROLLER_TRRD

Address: $0xf0006000 + 0xec = 0xf00060ec$

SDRAM_CONTROLLER_TRC

Address: $0xf0006000 + 0xf0 = 0xf00060f0$

SDRAM_CONTROLLER_TRAS

Address: $0xf0006000 + 0xf4 = 0xf00060f4$

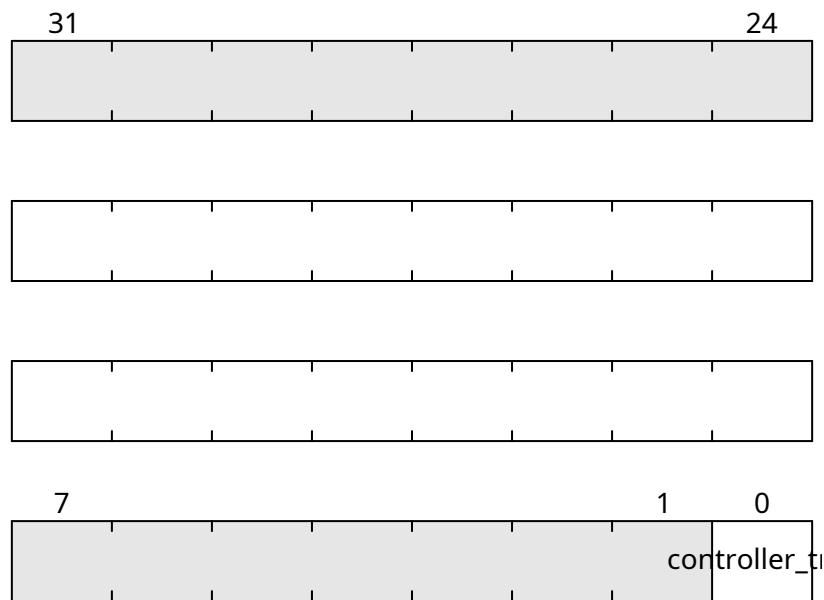


Fig. 20.172: SDRAM_CONTROLLER_TRTP

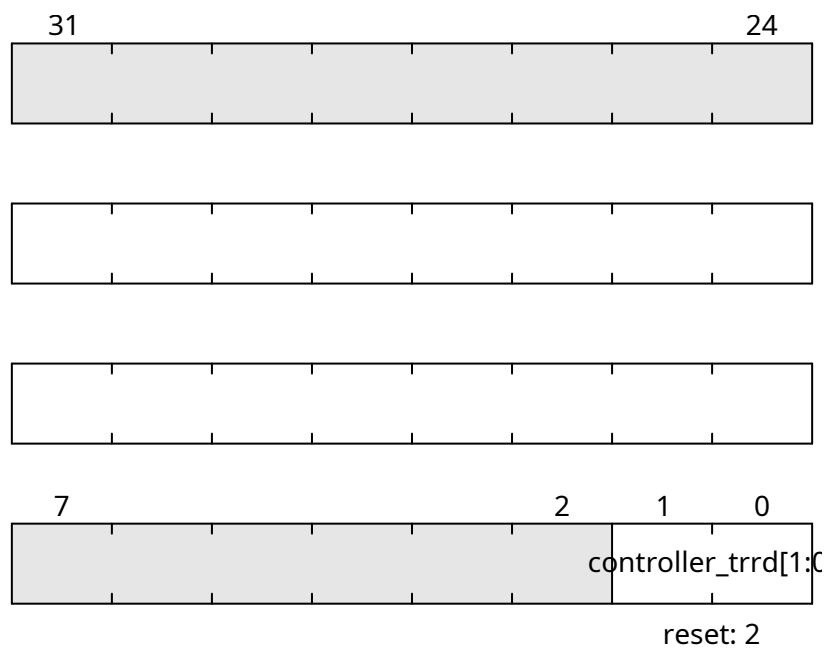


Fig. 20.173: SDRAM_CONTROLLER_TRRD

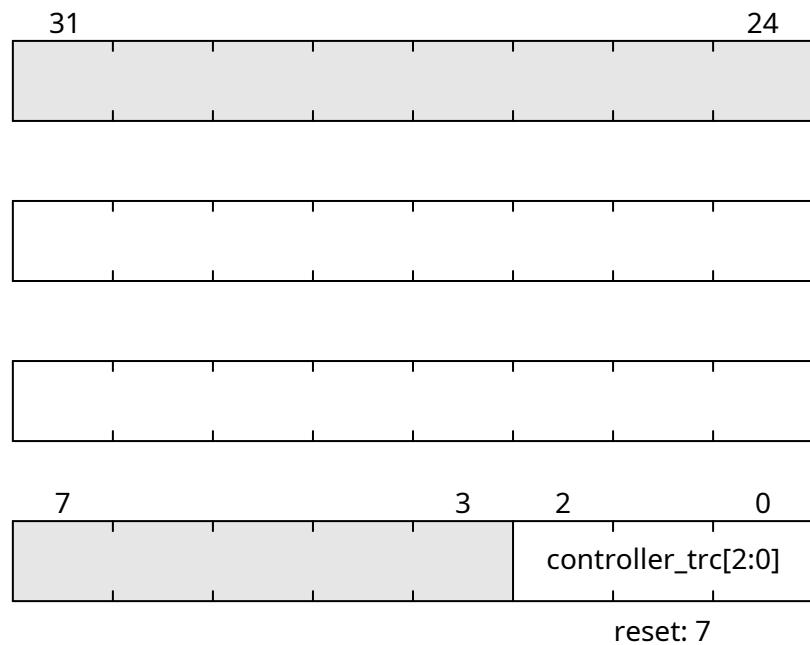


Fig. 20.174: SDRAM_CONTROLLER_TRC

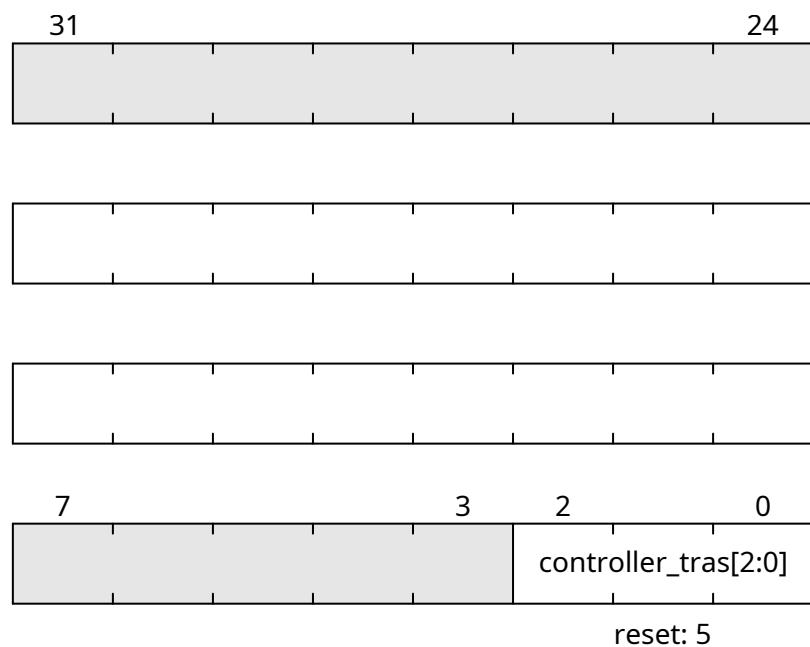


Fig. 20.175: SDRAM_CONTROLLER_TRAS

SDRAM_CONTROLLER_TZQCS

Address: $0xf0006000 + 0xf8 = 0xf00060f8$

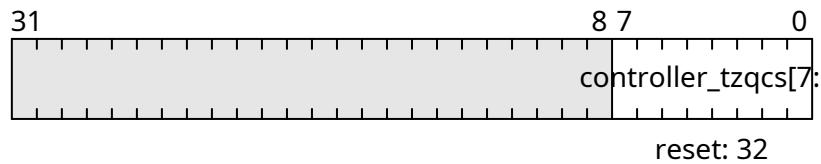


Fig. 20.176: SDRAM_CONTROLLER_TZQCS

SDRAM_CONTROLLER_LAST_ADDR_0

Address: $0xf0006000 + 0xfc = 0xf00060fc$

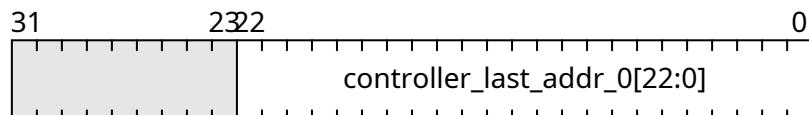


Fig. 20.177: SDRAM_CONTROLLER_LAST_ADDR_0

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0

Address: $0xf0006000 + 0x100 = 0xf0006100$

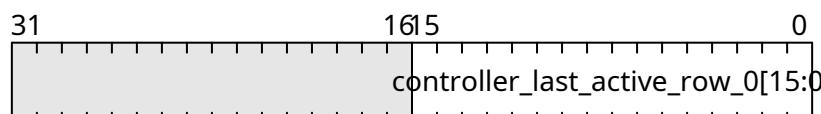


Fig. 20.178: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0

SDRAM_CONTROLLER_LAST_ADDR_1

Address: $0xf0006000 + 0x104 = 0xf0006104$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1

Address: $0xf0006000 + 0x108 = 0xf0006108$

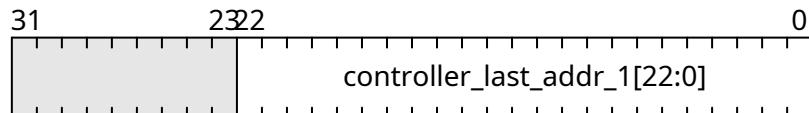


Fig. 20.179: SDRAM_CONTROLLER_LAST_ADDR_1

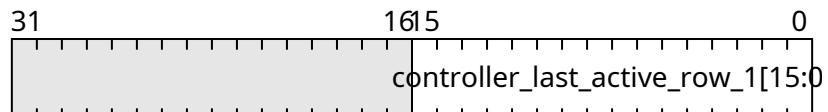


Fig. 20.180: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1

SDRAM_CONTROLLER_LAST_ADDR_2

Address: $0xf0006000 + 0x10c = 0xf000610c$

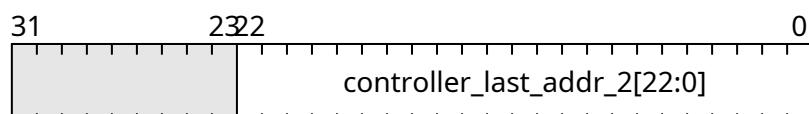


Fig. 20.181: SDRAM_CONTROLLER_LAST_ADDR_2

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2

Address: $0xf0006000 + 0x110 = 0xf0006110$

SDRAM_CONTROLLER_LAST_ADDR_3

Address: $0xf0006000 + 0x114 = 0xf0006114$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3

Address: $0xf0006000 + 0x118 = 0xf0006118$

SDRAM_CONTROLLER_LAST_ADDR_4

Address: $0xf0006000 + 0x11c = 0xf000611c$

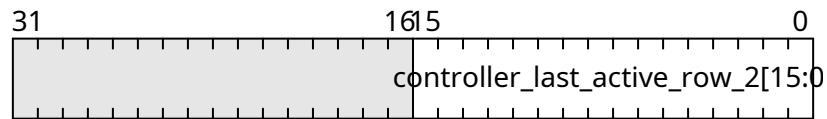


Fig. 20.182: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2

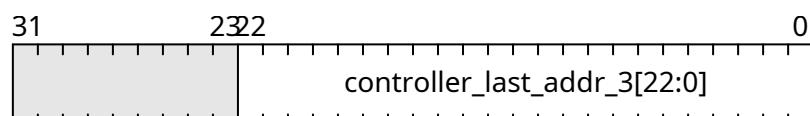


Fig. 20.183: SDRAM_CONTROLLER_LAST_ADDR_3

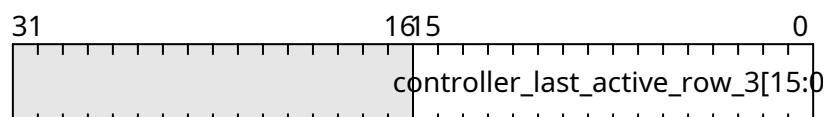


Fig. 20.184: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3

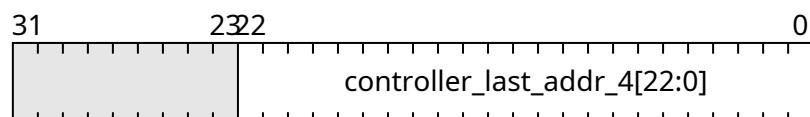


Fig. 20.185: SDRAM_CONTROLLER_LAST_ADDR_4

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4

Address: $0xf0006000 + 0x120 = 0xf0006120$

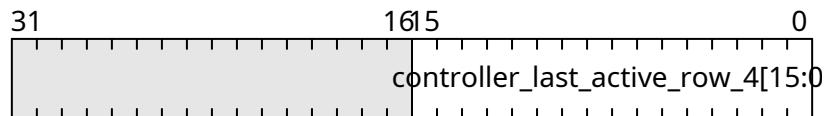


Fig. 20.186: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4

SDRAM_CONTROLLER_LAST_ADDR_5

Address: $0xf0006000 + 0x124 = 0xf0006124$

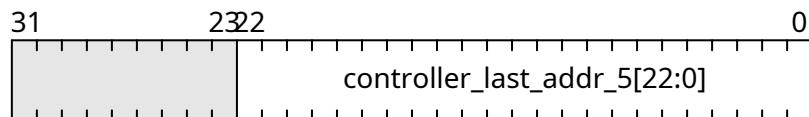


Fig. 20.187: SDRAM_CONTROLLER_LAST_ADDR_5

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5

Address: $0xf0006000 + 0x128 = 0xf0006128$

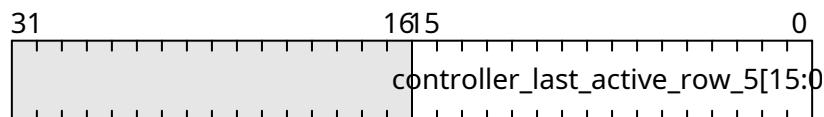


Fig. 20.188: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5

SDRAM_CONTROLLER_LAST_ADDR_6

Address: $0xf0006000 + 0x12c = 0xf000612c$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6

Address: $0xf0006000 + 0x130 = 0xf0006130$

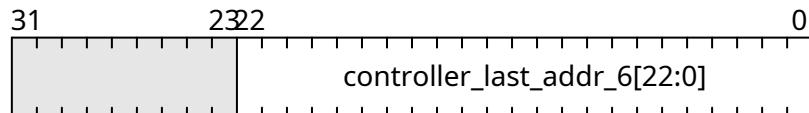


Fig. 20.189: SDRAM_CONTROLLER_LAST_ADDR_6

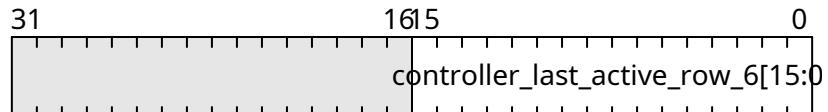


Fig. 20.190: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6

SDRAM_CONTROLLER_LAST_ADDR_7

Address: $0xf0006000 + 0x134 = 0xf0006134$

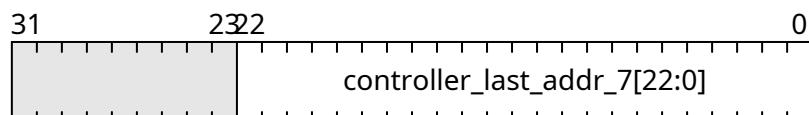


Fig. 20.191: SDRAM_CONTROLLER_LAST_ADDR_7

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7

Address: $0xf0006000 + 0x138 = 0xf0006138$

20.2.14 SDRAM_CHECKER

Register Listing for SDRAM_CHECKER

Register	Address
<i>SDRAM_CHECKER_RESET</i>	<i>0xf0006800</i>
<i>SDRAM_CHECKER_START</i>	<i>0xf0006804</i>
<i>SDRAM_CHECKER_DONE</i>	<i>0xf0006808</i>
<i>SDRAM_CHECKER_BASE</i>	<i>0xf000680c</i>
<i>SDRAM_CHECKER_END</i>	<i>0xf0006810</i>
<i>SDRAM_CHECKER_LENGTH</i>	<i>0xf0006814</i>
<i>SDRAM_CHECKER_RANDOM</i>	<i>0xf0006818</i>
<i>SDRAM_CHECKER_TICKS</i>	<i>0xf000681c</i>
<i>SDRAM_CHECKER_ERRORS</i>	<i>0xf0006820</i>

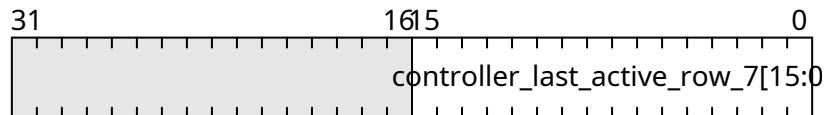


Fig. 20.192: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7

SDRAM_CHECKER_RESET

Address: $0xf0006800 + 0x0 = 0xf0006800$



Fig. 20.193: SDRAM_CHECKER_RESET

SDRAM_CHECKER_START

Address: $0xf0006800 + 0x4 = 0xf0006804$

SDRAM_CHECKER_DONE

Address: $0xf0006800 + 0x8 = 0xf0006808$

SDRAM_CHECKER_BASE

Address: $0xf0006800 + 0xc = 0xf000680c$



Fig. 20.194: SDRAM_CHECKER_START



Fig. 20.195: SDRAM_CHECKER_DONE

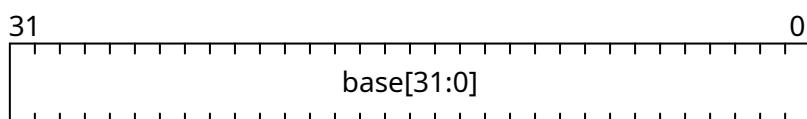


Fig. 20.196: SDRAM_CHECKER_BASE

SDRAM_CHECKER_END

Address: $0xf0006800 + 0x10 = 0xf0006810$

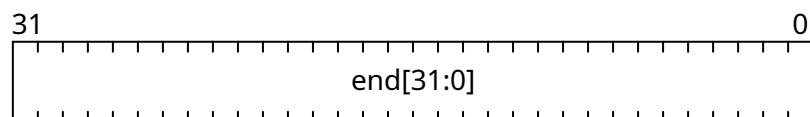


Fig. 20.197: SDRAM_CHECKER_END

SDRAM_CHECKER_LENGTH

Address: $0xf0006800 + 0x14 = 0xf0006814$

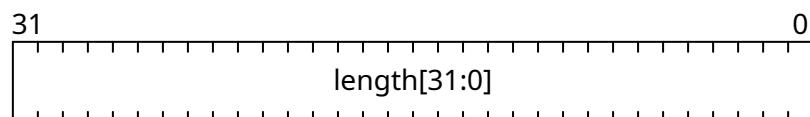


Fig. 20.198: SDRAM_CHECKER_LENGTH

SDRAM_CHECKER_RANDOM

Address: $0xf0006800 + 0x18 = 0xf0006818$



Fig. 20.199: SDRAM_CHECKER_RANDOM

Field	Name	Description

SDRAM_CHECKER_TICKS

Address: $0xf0006800 + 0x1c = 0xf000681c$

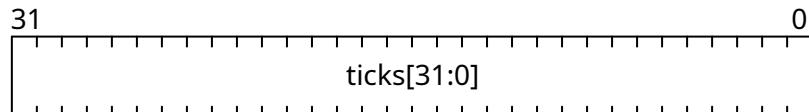


Fig. 20.200: SDRAM_CHECKER_TICKS

SDRAM_CHECKER_ERRORS

Address: $0xf0006800 + 0x20 = 0xf0006820$

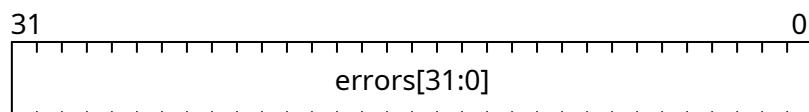


Fig. 20.201: SDRAM_CHECKER_ERRORS

20.2.15 SDRAM_GENERATOR

Register Listing for SDRAM_GENERATOR

Register	Address
<i>SDRAM_GENERATOR_RESET</i>	<i>0xf0007000</i>
<i>SDRAM_GENERATOR_START</i>	<i>0xf0007004</i>
<i>SDRAM_GENERATOR_DONE</i>	<i>0xf0007008</i>
<i>SDRAM_GENERATOR_BASE</i>	<i>0xf000700c</i>
<i>SDRAM_GENERATOR_END</i>	<i>0xf0007010</i>
<i>SDRAM_GENERATOR_LENGTH</i>	<i>0xf0007014</i>
<i>SDRAM_GENERATOR_RANDOM</i>	<i>0xf0007018</i>
<i>SDRAM_GENERATOR_TICKS</i>	<i>0xf000701c</i>

SDRAM_GENERATOR_RESET

Address: $0xf0007000 + 0x0 = 0xf0007000$



Fig. 20.202: SDRAM_GENERATOR_RESET

SDRAM_GENERATOR_START

Address: $0xf0007000 + 0x4 = 0xf0007004$

SDRAM_GENERATOR_DONE

Address: $0xf0007000 + 0x8 = 0xf0007008$

SDRAM_GENERATOR_BASE

Address: $0xf0007000 + 0xc = 0xf000700c$

SDRAM_GENERATOR_END

Address: $0xf0007000 + 0x10 = 0xf0007010$



Fig. 20.203: SDRAM_GENERATOR_START



Fig. 20.204: SDRAM_GENERATOR_DONE

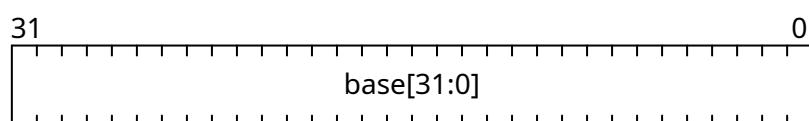


Fig. 20.205: SDRAM_GENERATOR_BASE

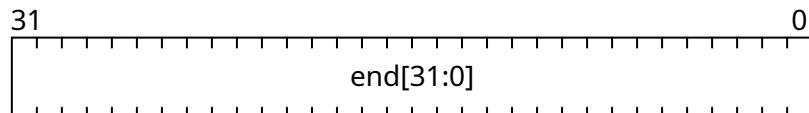


Fig. 20.206: SDRAM_GENERATOR_END

SDRAM_GENERATOR_LENGTH

Address: $0xf0007000 + 0x14 = 0xf0007014$

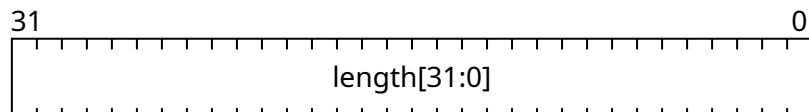


Fig. 20.207: SDRAM_GENERATOR_LENGTH

SDRAM_GENERATOR_RANDOM

Address: $0xf0007000 + 0x18 = 0xf0007018$

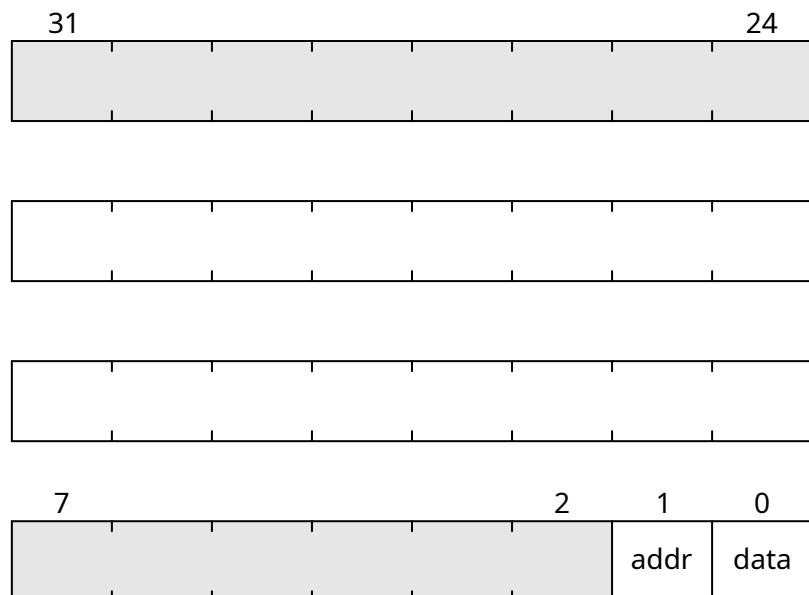


Fig. 20.208: SDRAM_GENERATOR_RANDOM

Field	Name	Description

SDRAM_GENERATOR_TICKS

Address: $0xf0007000 + 0x1c = 0xf000701c$

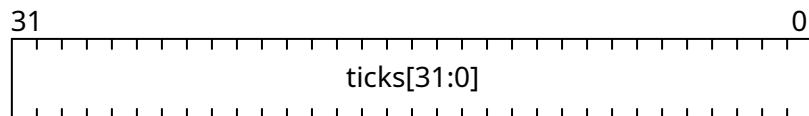


Fig. 20.209: SDRAM_GENERATOR_TICKS

20.2.16 TIMERO

Timer

Provides a generic Timer core.

The Timer is implemented as a countdown timer that can be used in various modes:

- Polling : Returns current countdown value to software
- One-Shot: Loads itself and stops when value reaches 0
- Periodic: (Re-)Loads itself when value reaches 0

en register allows the user to enable/disable the Timer. When the Timer is enabled, it is automatically loaded with the value of *load* register.

When the Timer reaches 0, it is automatically reloaded with value of *reload* register.

The user can latch the current countdown value by writing to *update_value* register, it will update *value* register with current countdown value.

To use the Timer in One-Shot mode, the user needs to:

- Disable the timer
- Set the load register to the expected duration
- (Re-)Enable the Timer

To use the Timer in Periodic mode, the user needs to:

- Disable the Timer
- Set the load register to 0
- Set the reload register to the expected period
- Enable the Timer

For both modes, the CPU can be advertised by an IRQ that the duration/period has elapsed. (The CPU can also do software polling with *update_value* and *value* to know the elapsed duration)

Register Listing for TIMERO

Register	Address
<i>TIMERO_LOAD</i>	0xf0007800
<i>TIMERO_RELOAD</i>	0xf0007804
<i>TIMERO_EN</i>	0xf0007808
<i>TIMERO_UPDATE_VALUE</i>	0xf000780c
<i>TIMERO_VALUE</i>	0xf0007810
<i>TIMERO_EV_STATUS</i>	0xf0007814
<i>TIMERO_EV_PENDING</i>	0xf0007818
<i>TIMERO_EV_ENABLE</i>	0xf000781c

TIMERO_LOAD

Address: $0xf0007800 + 0x0 = 0xf0007800$

Load value when Timer is (re-)enabled. In One-Shot mode, the value written to this register specifies the Timer's duration in clock cycles.

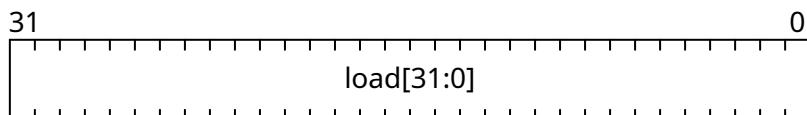


Fig. 20.210: TIMERO_LOAD

TIMERO_RELOAD

Address: $0xf0007800 + 0x4 = 0xf0007804$

Reload value when Timer reaches 0. In Periodic mode, the value written to this register specify the Timer's period in clock cycles.

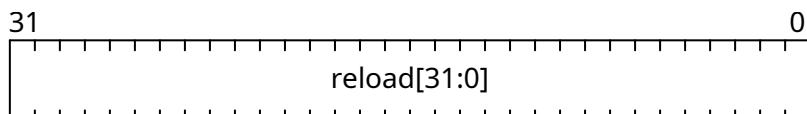


Fig. 20.211: TIMERO_RELOAD

TIMERO_EN

Address: $0xf0007800 + 0x8 = 0xf0007808$

Enable flag of the Timer. Set this flag to 1 to enable/start the Timer. Set to 0 to disable the Timer.



Fig. 20.212: TIMERO_EN

TIMERO_UPDATE_VALUE

Address: $0xf0007800 + 0xc = 0xf000780c$

Update trigger for the current countdown value. A write to this register latches the current countdown value to value register.

TIMERO_VALUE

Address: $0xf0007800 + 0x10 = 0xf0007810$

Latched countdown value. This value is updated by writing to update_value.

TIMERO_EV_STATUS

Address: $0xf0007800 + 0x14 = 0xf0007814$

This register contains the current raw level of the zero event trigger. Writes to this register have no effect.

Field	Name	Description
[0]	ZERO	Level of the zero event

TIMERO_EV_PENDING

Address: $0xf0007800 + 0x18 = 0xf0007818$

When a zero event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.



Fig. 20.213: TIMERO_UPDATE_VALUE

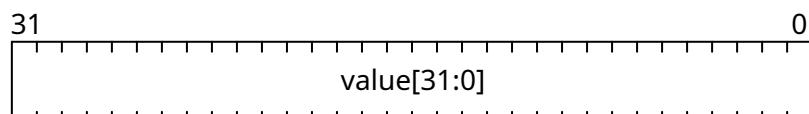


Fig. 20.214: TIMERO_VALUE



Fig. 20.215: TIMERO_EV_STATUS



Fig. 20.216: TIMERO_EV_PENDING

Field	Name	Description
[0]	ZERO	1 if a zero event occurred. This Event is triggered on a falling edge.

TIMERO_EV_ENABLE

Address: $0xf0007800 + 0x1c = 0xf000781c$

This register enables the corresponding zero events. Write a 0 to this register to disable individual events.

Field	Name	Description
[0]	ZERO	Write a 1 to enable the zero Event

20.2.17 UART



Fig. 20.217: TIMERO_EV_ENABLE

Register Listing for UART

Register	Address
UART_RXTX	0xf0008000
UART_TXFULL	0xf0008004
UART_RXEMPTY	0xf0008008
UART_EV_STATUS	0xf000800c
UART_EV_PENDING	0xf0008010
UART_EV_ENABLE	0xf0008014
UART_TXEMPTY	0xf0008018
UART_RXFULL	0xf000801c
UART_XOVER_RXTX	0xf0008020
UART_XOVER_TXFULL	0xf0008024
UART_XOVER_RXEMPTY	0xf0008028
UART_XOVER_EV_STATUS	0xf000802c
UART_XOVER_EV_PENDING	0xf0008030
UART_XOVER_EV_ENABLE	0xf0008034
UART_XOVER_TXEMPTY	0xf0008038
UART_XOVER_RXFULL	0xf000803c

UART_RXTX

Address: $0xf0008000 + 0x0 = 0xf0008000$

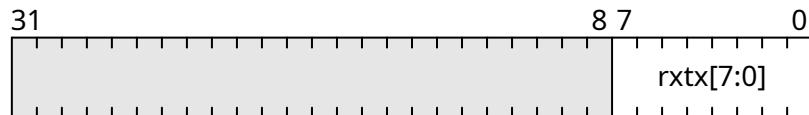


Fig. 20.218: UART_RXTX

UART_TXFULL

Address: $0xf0008000 + 0x4 = 0xf0008004$

TX FIFO Full.



Fig. 20.219: UART_TXFULL

UART_RXEMPTY

Address: $0xf0008000 + 0x8 = 0xf0008008$

RX FIFO Empty.

UART_EV_STATUS

Address: $0xf0008000 + 0xc = 0xf000800c$

This register contains the current raw level of the rx event trigger. Writes to this register have no effect.

Field	Name	Description
[0]	TX	Level of the tx event
[1]	RX	Level of the rx event



Fig. 20.220: UART_RXEMPTY

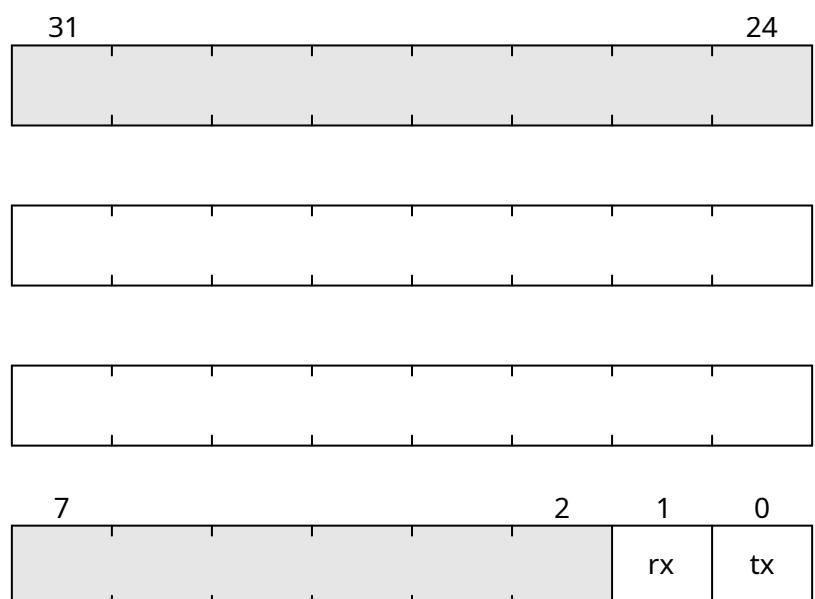


Fig. 20.221: UART_EV_STATUS

UART_EV_PENDING

Address: $0xf0008000 + 0x10 = 0xf0008010$

When a rx event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.

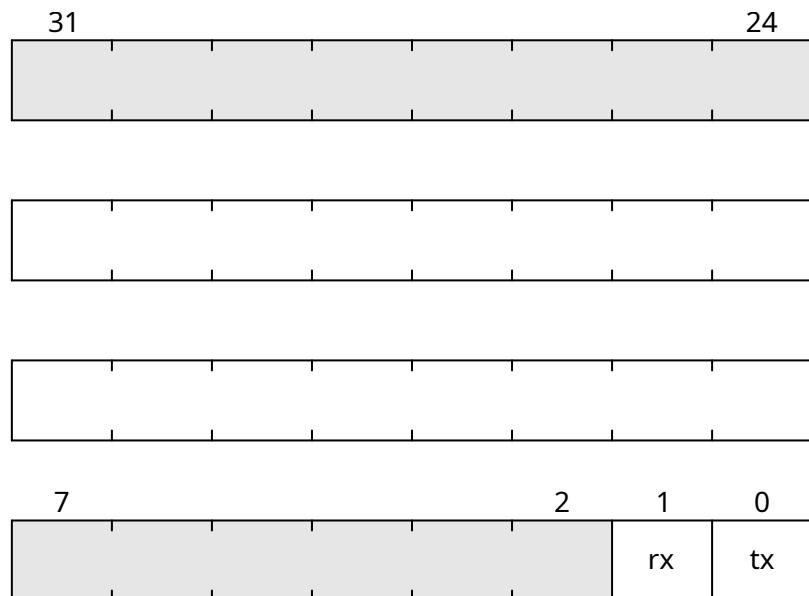


Fig. 20.222: UART_EV_PENDING

Field	Name	Description
[0]	TX	1 if a tx event occurred. This Event is triggered on a falling edge.
[1]	RX	1 if a rx event occurred. This Event is triggered on a falling edge.

UART_EV_ENABLE

Address: $0xf0008000 + 0x14 = 0xf0008014$

This register enables the corresponding rx events. Write a 0 to this register to disable individual events.

Field	Name	Description
[0]	TX	Write a 1 to enable the tx Event
[1]	RX	Write a 1 to enable the rx Event

UART_TXEMPTY

Address: $0xf0008000 + 0x18 = 0xf0008018$

TX FIFO Empty.

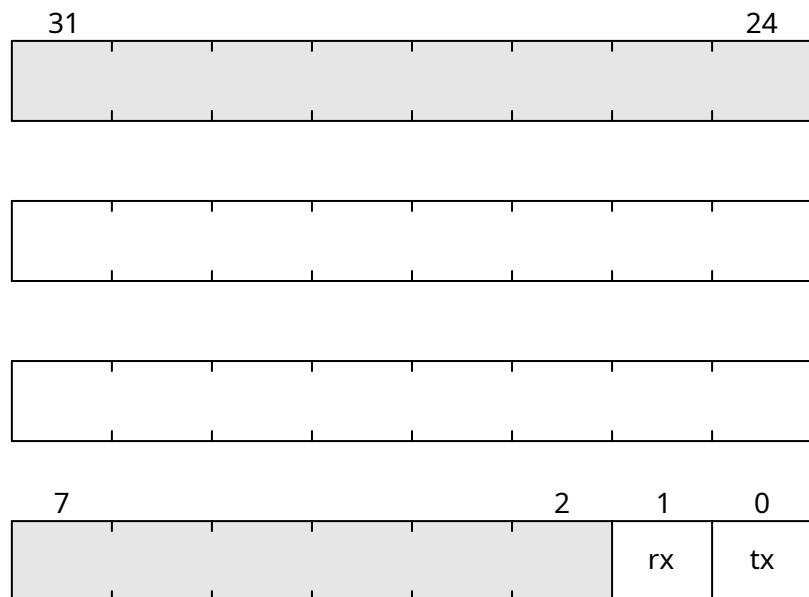


Fig. 20.223: `UART_EV_ENABLE`



Fig. 20.224: `UART_TXEMPTY`

UART_RXFULL

Address: $0xf0008000 + 0x1c = 0xf000801c$

RX FIFO Full.

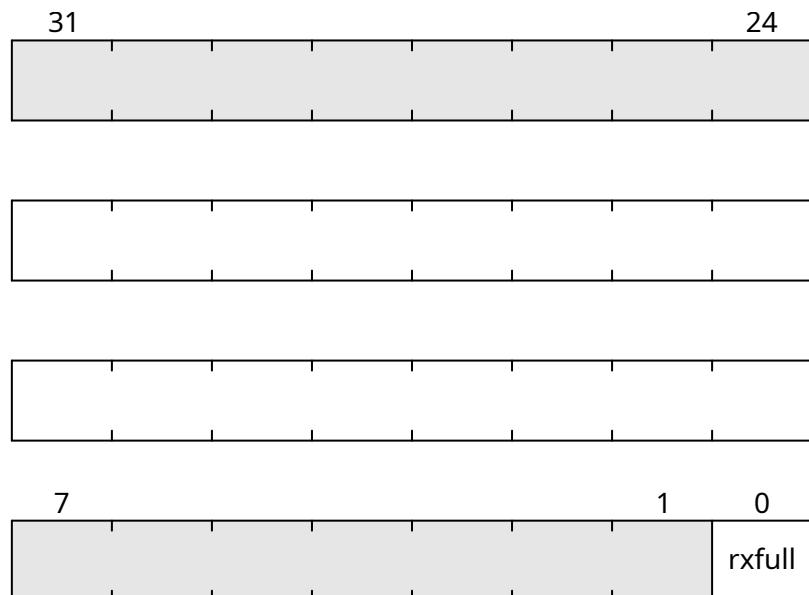


Fig. 20.225: UART_RXFULL

UART_XOVER_RXTX

Address: $0xf0008000 + 0x20 = 0xf0008020$

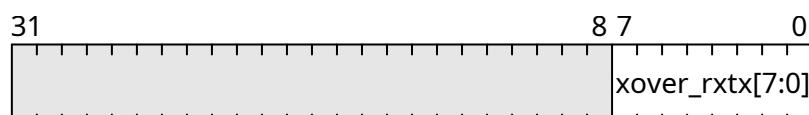


Fig. 20.226: UART_XOVER_RXTX

UART_XOVER_TXFULL

Address: $0xf0008000 + 0x24 = 0xf0008024$

TX FIFO Full.

UART_XOVER_RXEMPTY

Address: $0xf0008000 + 0x28 = 0xf0008028$

RX FIFO Empty.

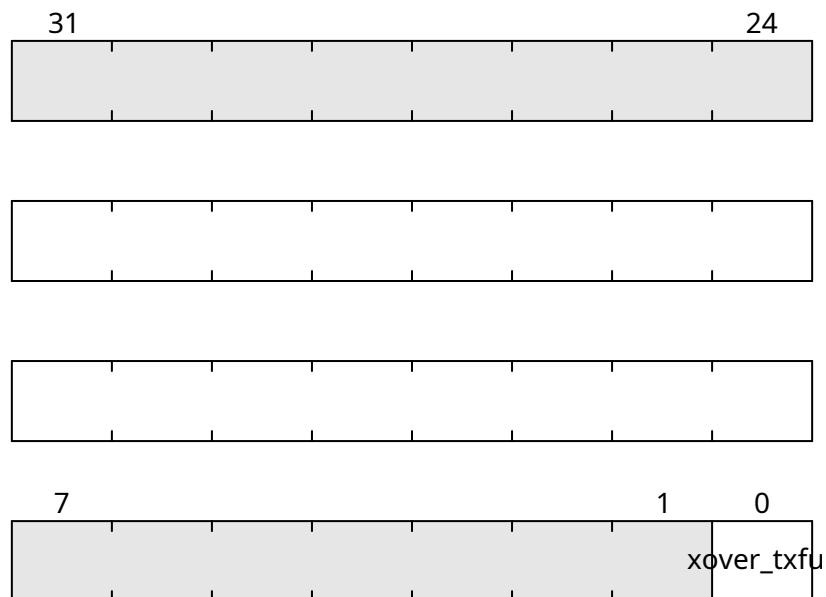


Fig. 20.227: `UART_XOVER_TXFULL`

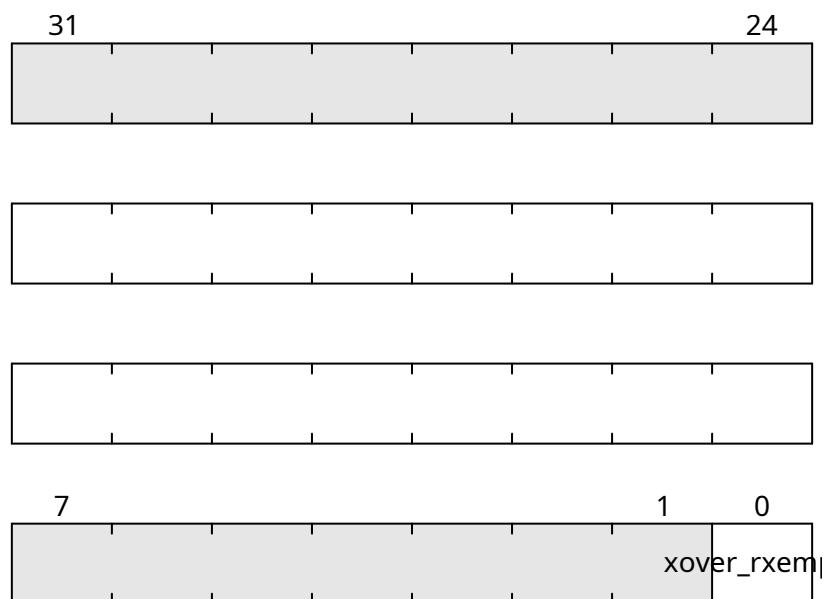


Fig. 20.228: `UART_XOVER_RXEMPTY`

UART_XOVER_EV_STATUS

Address: $0xf0008000 + 0x2c = 0xf000802c$

This register contains the current raw level of the rx event trigger. Writes to this register have no effect.

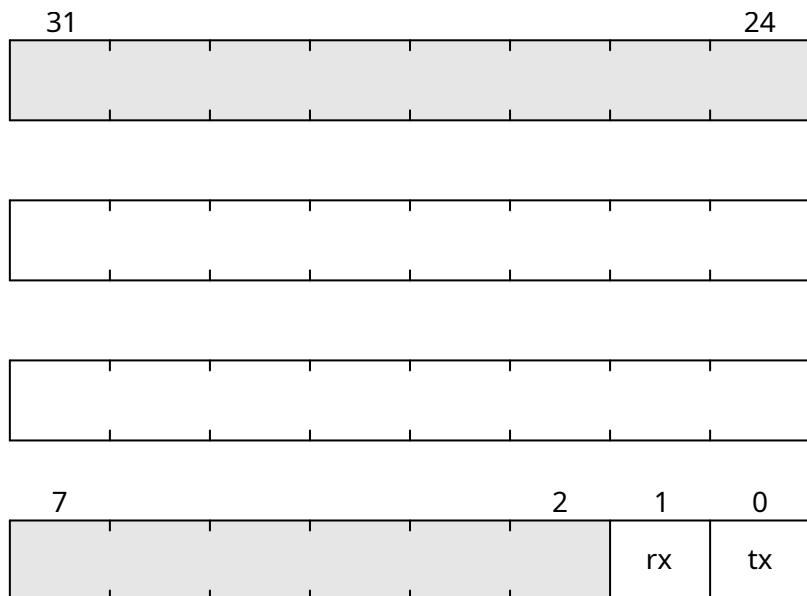


Fig. 20.229: UART_XOVER_EV_STATUS

Field	Name	Description
[0]	TX	Level of the tx event
[1]	RX	Level of the rx event

UART_XOVER_EV_PENDING

Address: $0xf0008000 + 0x30 = 0xf0008030$

When a rx event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.

Field	Name	Description
[0]	TX	1 if a tx event occurred. This Event is triggered on a falling edge .
[1]	RX	1 if a rx event occurred. This Event is triggered on a falling edge .

UART_XOVER_EV_ENABLE

Address: $0xf0008000 + 0x34 = 0xf0008034$

This register enables the corresponding rx events. Write a 0 to this register to disable individual events.

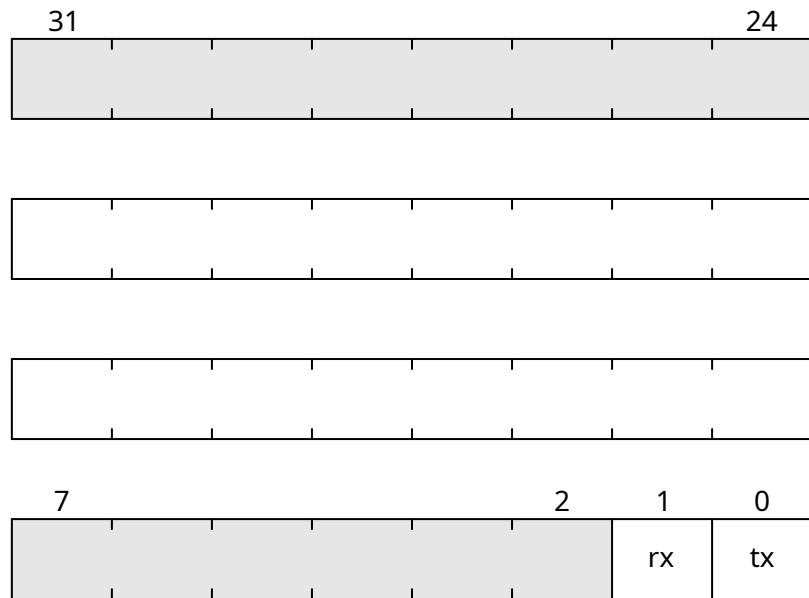


Fig. 20.230: `UART_XOVER_EV_PENDING`



Fig. 20.231: `UART_XOVER_EV_ENABLE`

Field	Name	Description
[0]	TX	Write a 1 to enable the tx Event
[1]	RX	Write a 1 to enable the rx Event

UART_XOVER_TXEMPTY

Address: $0xf0008000 + 0x38 = 0xf0008038$

TX FIFO Empty.

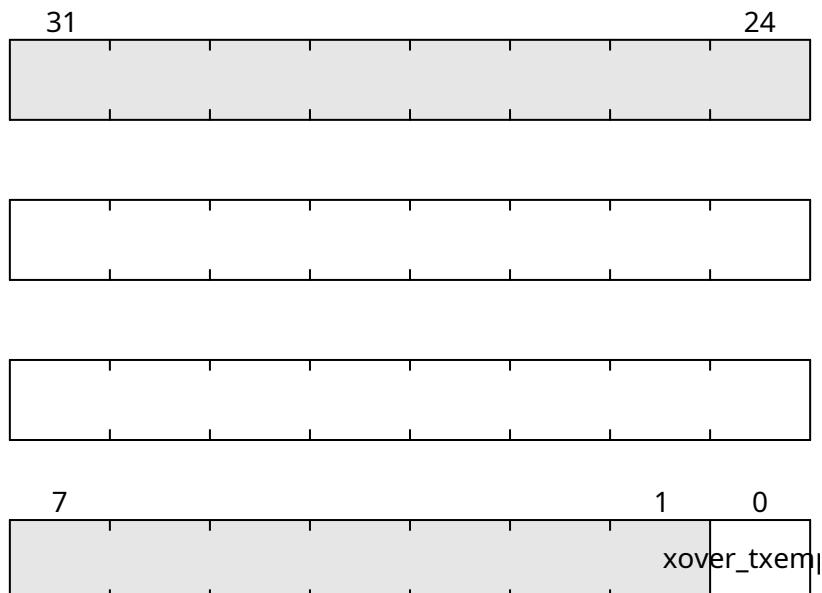


Fig. 20.232: UART_XOVER_TXEMPTY

UART_XOVER_RXFULL

Address: $0xf0008000 + 0x3c = 0xf000803c$

RX FIFO Full.

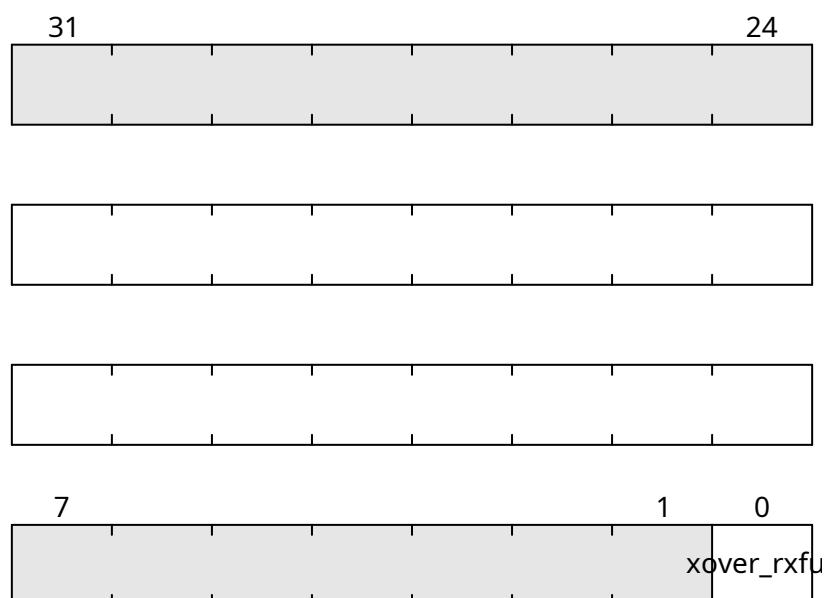


Fig. 20.233: UART_XOVER_RXFULL

CHAPTER
TWENTYONE

DOCUMENTATION FOR ROW HAMMER TESTER DATA CENTER
DRAM TESTER

21.1 Modules

21.1.1 Interrupt Controller

This device has an EventManager-based interrupt system. Individual modules generate *events* which are wired into a central interrupt controller.

When an interrupt occurs, you should look the interrupt number up in the CPU- specific interrupt table and then call the relevant module.

Assigned Interrupts

The following interrupts are assigned on this system:

Interrupt	Module
1	<i>TIMER0</i>
0	<i>UART</i>

21.2 Register Groups

21.2.1 LEDS

Register Listing for LEDS

Register	Address
<i>LEDS_OUT</i>	<i>0xf0000000</i>

LEDS_OUT

Address: *0xf0000000* + *0x0* = *0xf0000000*

Led Output(s) Control.

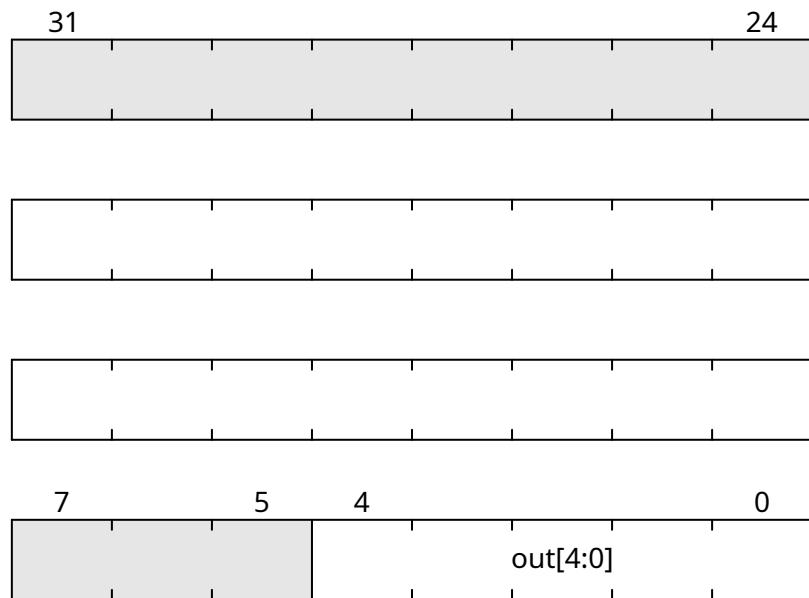


Fig. 21.1: LEDS_OUT

21.2.2 DDRPHY

Register Listing for DDRPHY

Register	Address
<i>DDRPHY_RST</i>	0xf0000800
<i>DDRPHY_DLX_SEL</i>	0xf0000804
<i>DDRPHY_HALF_SYS8X_TAPS</i>	0xf0000808
<i>DDRPHY_WLEVEL_EN</i>	0xf000080c
<i>DDRPHY_WLEVEL_STROBE</i>	0xf0000810
<i>DDRPHY_CDLY_RST</i>	0xf0000814
<i>DDRPHY_CDLY_INC</i>	0xf0000818
<i>DDRPHY_RDLY_DQ_RST</i>	0xf000081c
<i>DDRPHY_RDLY_DQ_INC</i>	0xf0000820
<i>DDRPHY_RDLY_DQ_BITSLIP_RST</i>	0xf0000824
<i>DDRPHY_RDLY_DQ_BITSLIP</i>	0xf0000828
<i>DDRPHY_WDLY_DQ_RST</i>	0xf000082c
<i>DDRPHY_WDLY_DQ_INC</i>	0xf0000830
<i>DDRPHY_WDLY_DQS_RST</i>	0xf0000834
<i>DDRPHY_WDLY_DQS_INC</i>	0xf0000838
<i>DDRPHY_WDLY_DQ_BITSLIP_RST</i>	0xf000083c
<i>DDRPHY_WDLY_DQ_BITSLIP</i>	0xf0000840
<i>DDRPHY_RDPHASE</i>	0xf0000844
<i>DDRPHY_WRPHASE</i>	0xf0000848
<i>DDRPHY_ALERT</i>	0xf000084c
<i>DDRPHY_RST_ALERT</i>	0xf0000850

DDRPHY_RST

Address: $0xf0000800 + 0x0 = 0xf0000800$



Fig. 21.2: DDRPHY_RST

DDRPHY_DLY_SEL

Address: $0xf0000800 + 0x4 = 0xf0000804$

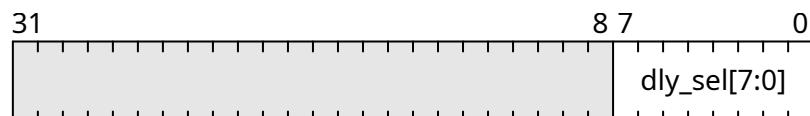


Fig. 21.3: DDRPHY_DLY_SEL

DDRPHY_HALF_SYS8X_TAPS

Address: $0xf0000800 + 0x8 = 0xf0000808$

DDRPHY_WLEVEL_EN

Address: $0xf0000800 + 0xc = 0xf000080c$

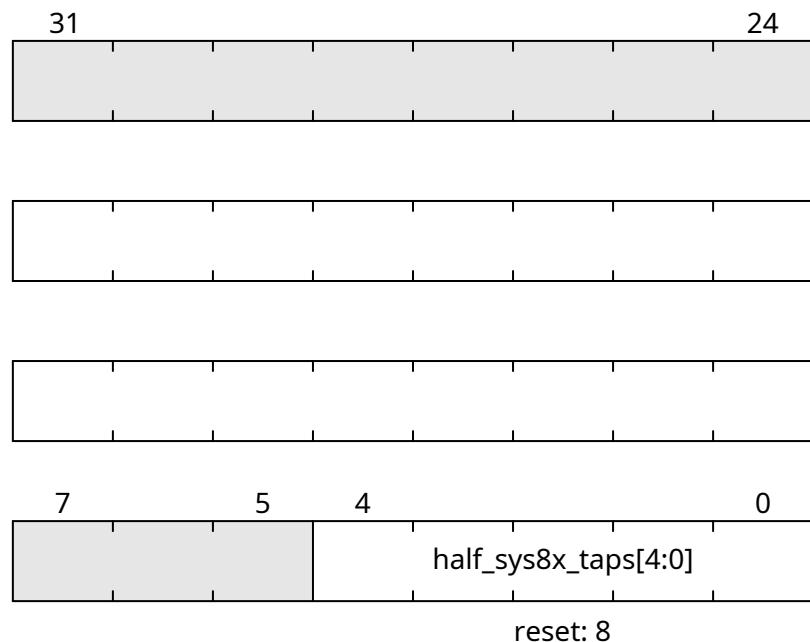


Fig. 21.4: `DDRPHY_HALF_SYS8X_TAPS`

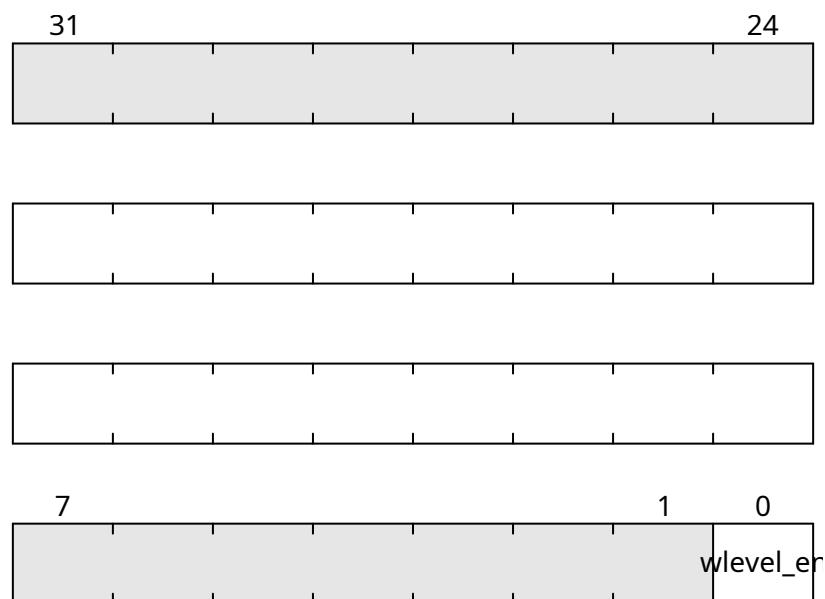


Fig. 21.5: `DDRPHY_WLEVEL_EN`

DDRPHY_WLEVEL_STROBE

Address: $0xf0000800 + 0x10 = 0xf0000810$



Fig. 21.6: DDRPHY_WLEVEL_STROBE

DDRPHY_CDLY_RST

Address: $0xf0000800 + 0x14 = 0xf0000814$

DDRPHY_CDLY_INC

Address: $0xf0000800 + 0x18 = 0xf0000818$

DDRPHY_RDLY_DQ_RST

Address: $0xf0000800 + 0x1c = 0xf000081c$

DDRPHY_RDLY_DQ_INC

Address: $0xf0000800 + 0x20 = 0xf0000820$



Fig. 21.7: DDRPHY_CDLY_RST



Fig. 21.8: DDRPHY_CDLY_INC

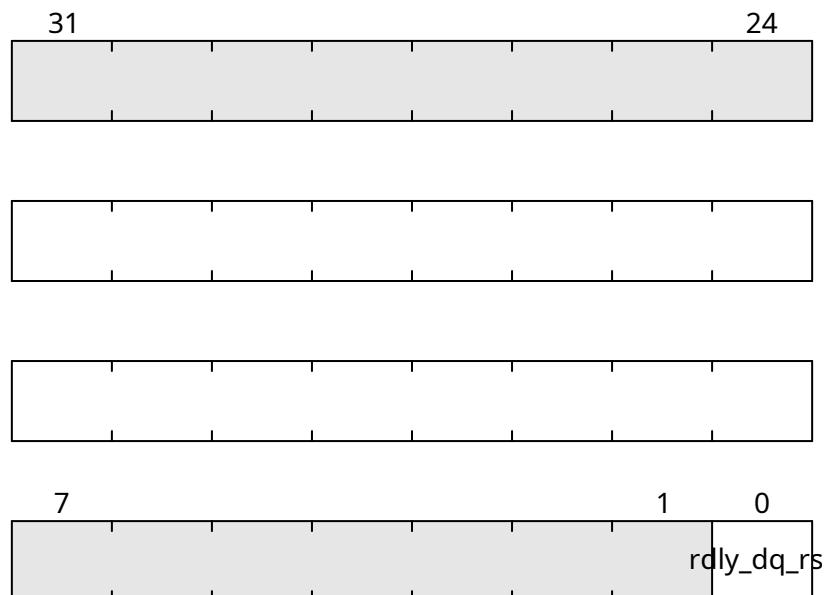


Fig. 21.9: DDRPHY_RDLY_DQ_RST



Fig. 21.10: DDRPHY_RDLY_DQ_INC

DDRPHY_RDLY_DQ_BITSLIP_RST

Address: $0xf0000800 + 0x24 = 0xf0000824$

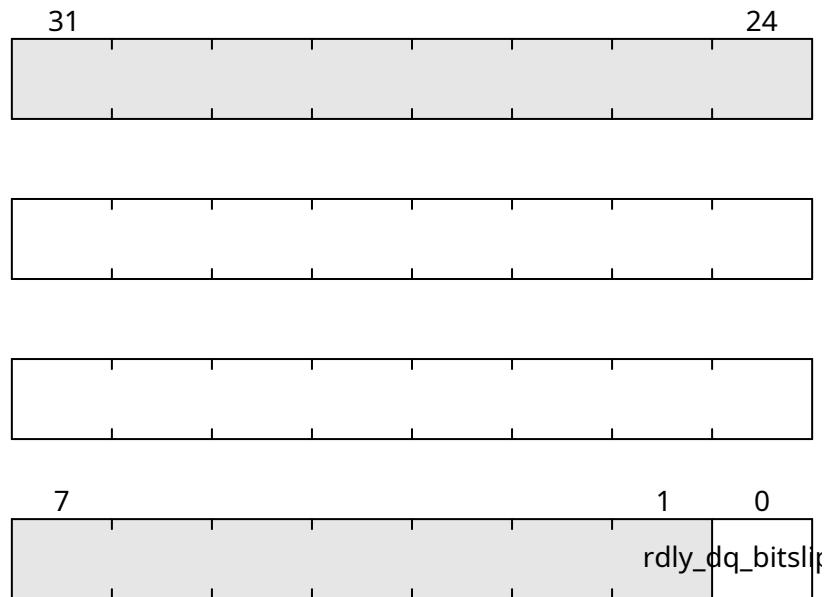


Fig. 21.11: DDRPHY_RDLY_DQ_BITSLIP_RST

DDRPHY_RDLY_DQ_BITSLIP

Address: $0xf0000800 + 0x28 = 0xf0000828$

DDRPHY_WDLY_DQ_RST

Address: $0xf0000800 + 0x2c = 0xf000082c$

DDRPHY_WDLY_DQ_INC

Address: $0xf0000800 + 0x30 = 0xf0000830$

DDRPHY_WDLY_DQS_RST

Address: $0xf0000800 + 0x34 = 0xf0000834$



Fig. 21.12: DDRPHY_RDLY_DQ_BITSLIP

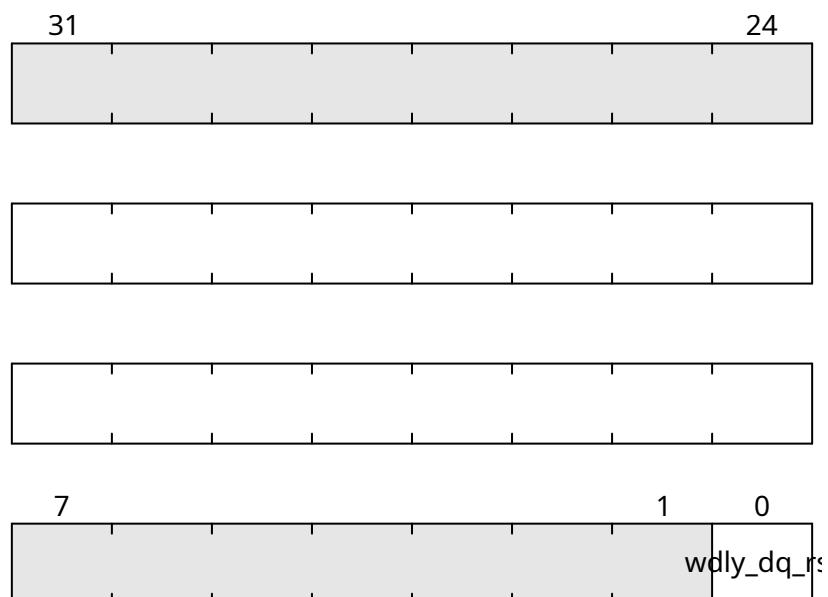


Fig. 21.13: DDRPHY_WDLY_DQ_RST

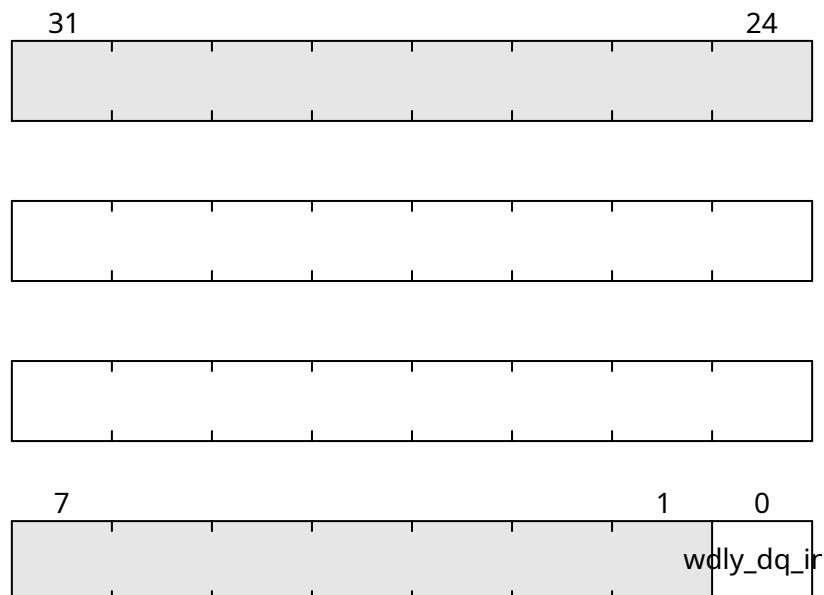


Fig. 21.14: DDRPHY_WDLY_DQ_INC

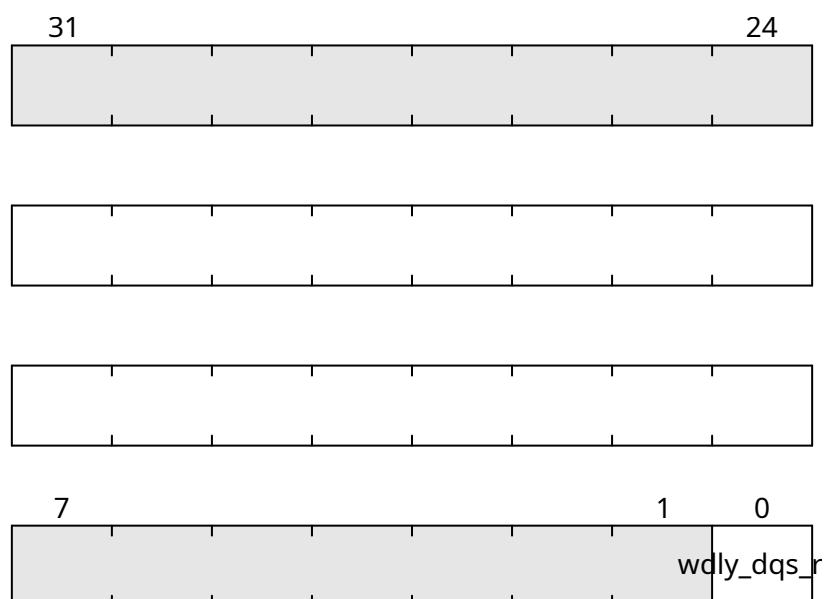


Fig. 21.15: DDRPHY_WDLY_DQS_RST

DDRPHY_WDLY_DQS_INC

Address: $0xf0000800 + 0x38 = 0xf0000838$



Fig. 21.16: DDRPHY_WDLY_DQS_INC

DDRPHY_WDLY_DQ_BITSLIP_RST

Address: $0xf0000800 + 0x3c = 0xf000083c$

DDRPHY_WDLY_DQ_BITSLIP

Address: $0xf0000800 + 0x40 = 0xf0000840$

DDRPHY_RDPHASE

Address: $0xf0000800 + 0x44 = 0xf0000844$

DDRPHY_WRPHASE

Address: $0xf0000800 + 0x48 = 0xf0000848$

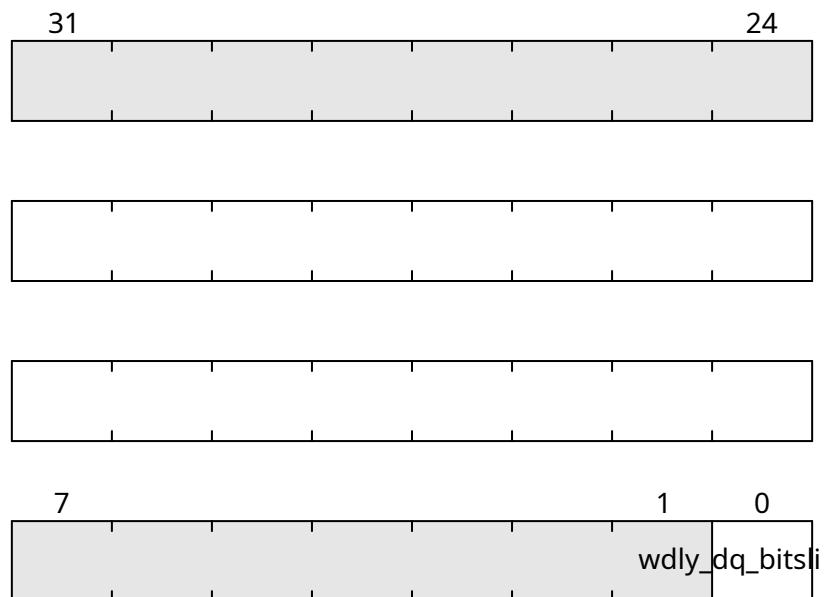


Fig. 21.17: DDRPHY_WDLY_DQ_BITSLIP_RST

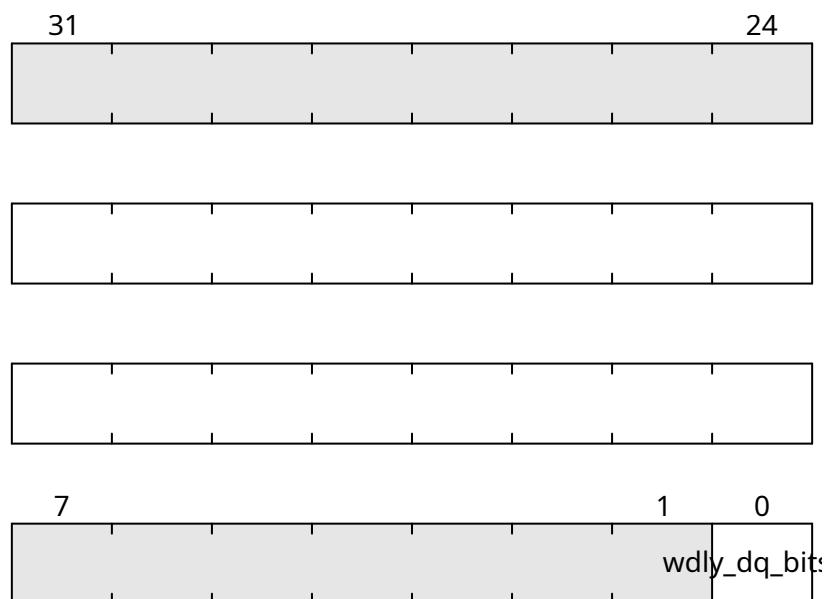


Fig. 21.18: DDRPHY_WDLY_DQ_BITSLIP

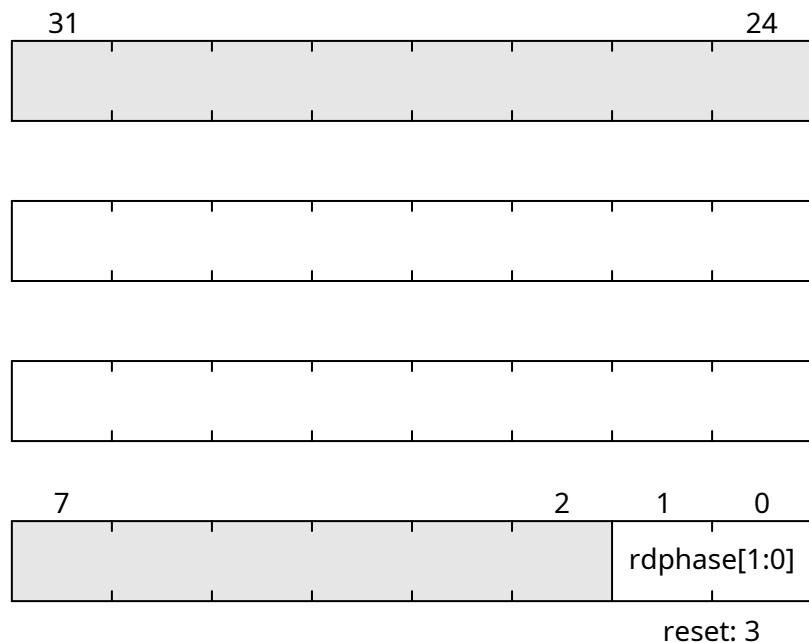


Fig. 21.19: DDRPHY_RDPHASE

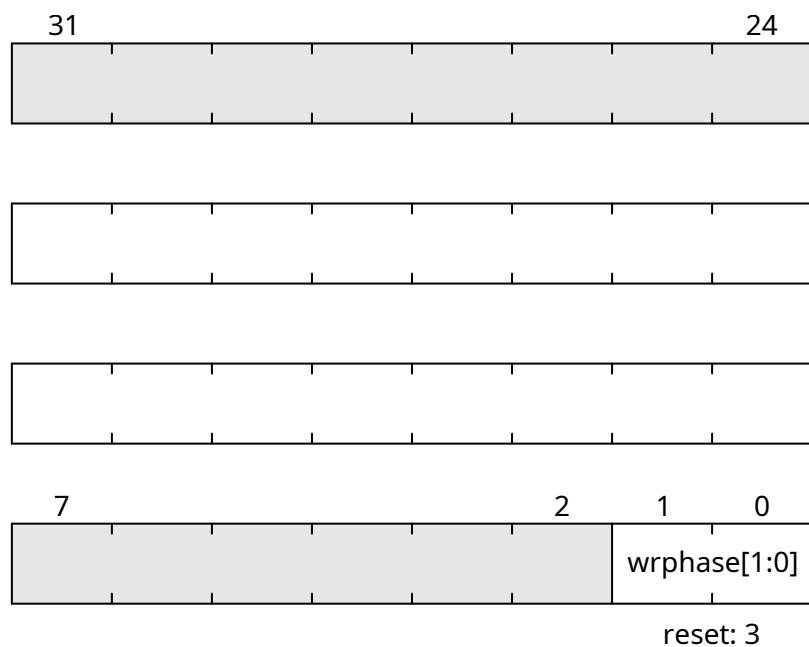


Fig. 21.20: DDRPHY_WRPAGE

DDRPHY_ALERT

Address: $0xf0000800 + 0x4c = 0xf000084c$

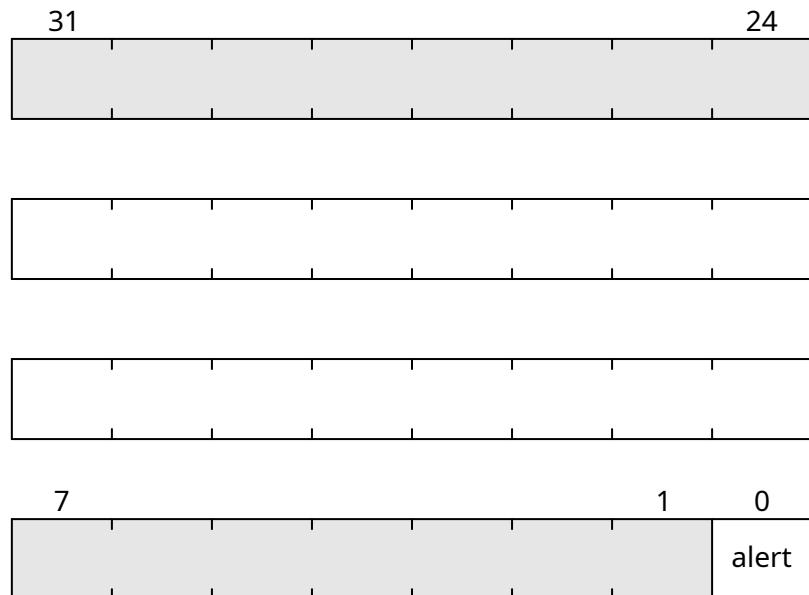


Fig. 21.21: DDRPHY_ALERT

DDRPHY_RST_ALERT

Address: $0xf0000800 + 0x50 = 0xf0000850$

21.2.3 CONTROLLER_SETTINGS

Allows to change LiteDRAMController behaviour at runtime

Register Listing for CONTROLLER_SETTINGS

Register	Address
<i>CONTROLLER_SETTINGS_REFRESH</i>	<i>0xf0001000</i>

CONTROLLER_SETTINGS_REFRESH

Address: $0xf0001000 + 0x0 = 0xf0001000$

Enable/disable Refresh commands sending

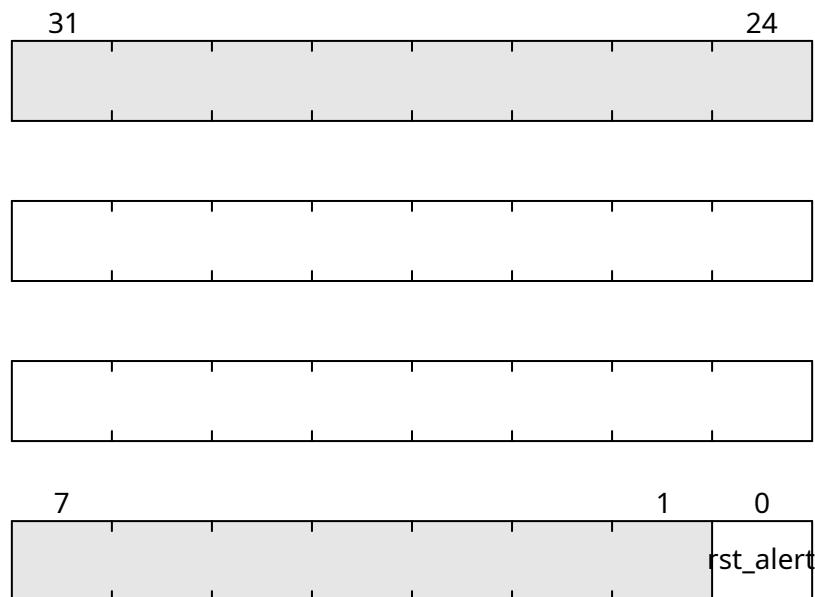


Fig. 21.22: DDRPHY_RST_ALERT

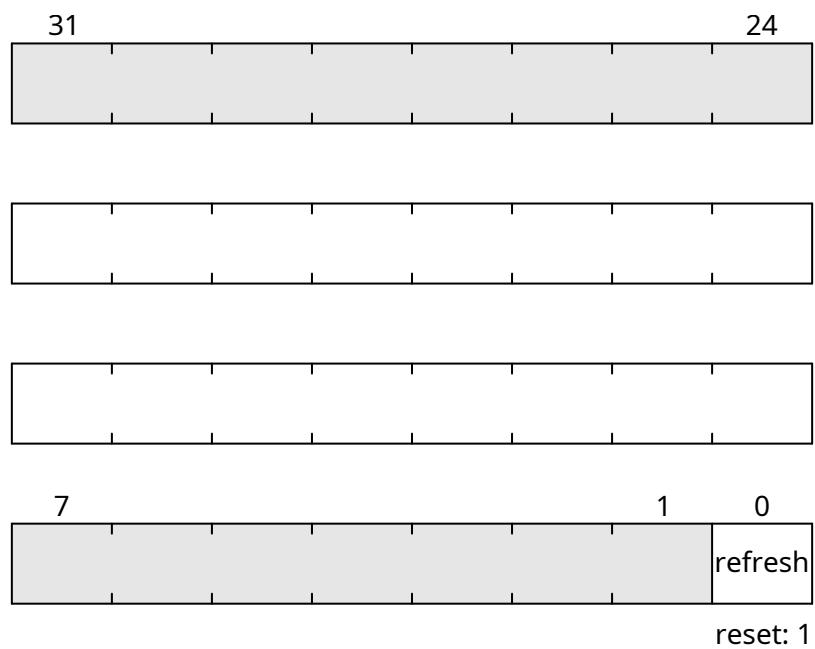


Fig. 21.23: CONTROLLER_SETTINGS_REFRESH

21.2.4 DDRCTRL

Register Listing for DDRCTRL

Register	Address
<i>DDRCTRL_INIT_DONE</i>	<i>0xf0001800</i>
<i>DDRCTRL_INIT_ERROR</i>	<i>0xf0001804</i>

DDRCTRL_INIT_DONE

Address: $0xf0001800 + 0x0 = 0xf0001800$

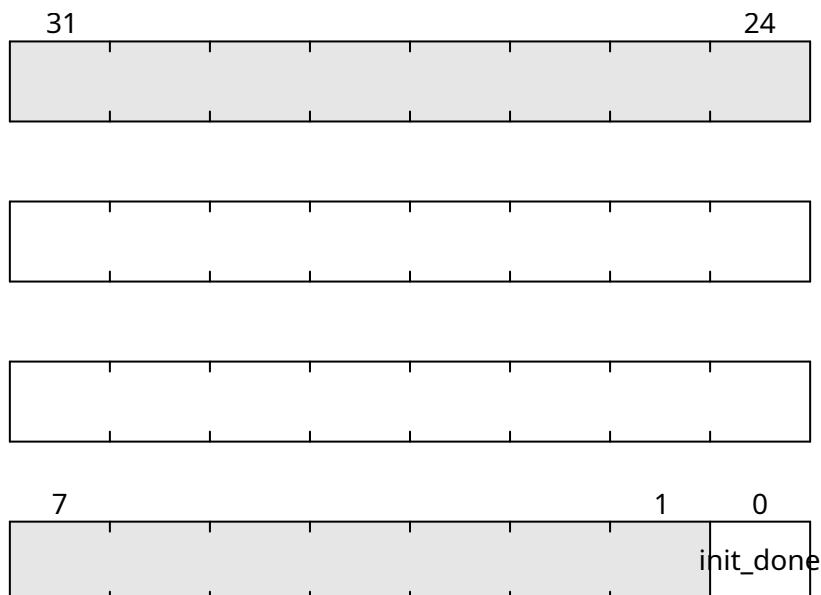


Fig. 21.24: DDRCTRL_INIT_DONE

DDRCTRL_INIT_ERROR

Address: $0xf0001800 + 0x4 = 0xf0001804$

21.2.5 ROWHAMMER

Row Hammer DMA attacker

This module allows to perform a Row Hammer attack by configuring it with two addresses that map to different rows of a single bank. When enabled, it will perform alternating DMA reads from the given locations, which will result in the DRAM controller having to repeatedly open/close rows at each read access.

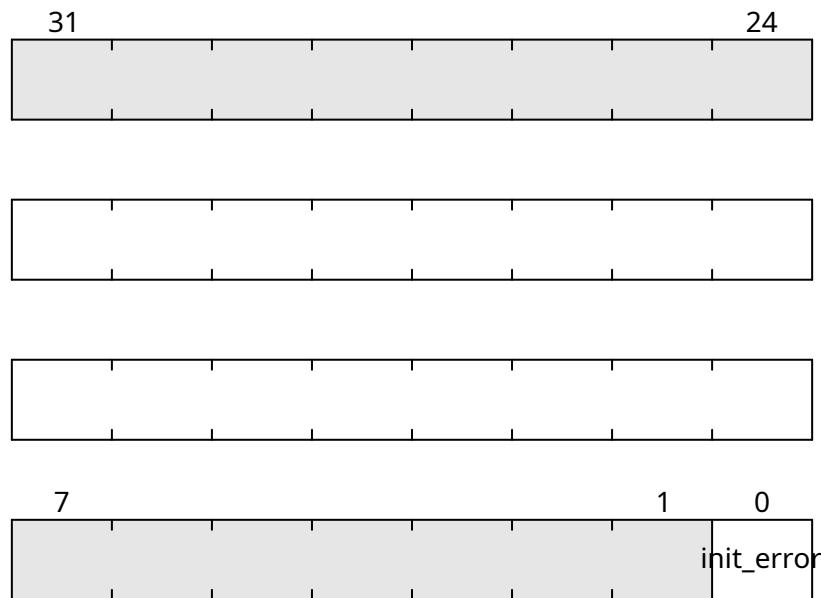


Fig. 21.25: DDRCTRL_INIT_ERROR

Register Listing for ROWHAMMER

Register	Address
<i>ROWHAMMER_ENABLED</i>	<i>0xf0002000</i>
<i>ROWHAMMER_ADDRESS1</i>	<i>0xf0002004</i>
<i>ROWHAMMER_ADDRESS2</i>	<i>0xf0002008</i>
<i>ROWHAMMER_COUNT</i>	<i>0xf000200c</i>

ROWHAMMER_ENABLED

Address: $0xf0002000 + 0x0 = 0xf0002000$

Used to start/stop the operation of the module

ROWHAMMER_ADDRESS1

Address: $0xf0002000 + 0x4 = 0xf0002004$

First attacked address

ROWHAMMER_ADDRESS2

Address: $0xf0002000 + 0x8 = 0xf0002008$

Second attacked address



Fig. 21.26: `ROWHAMMER_ENABLED`

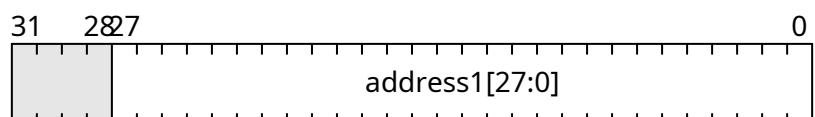


Fig. 21.27: `ROWHAMMER_ADDRESS1`

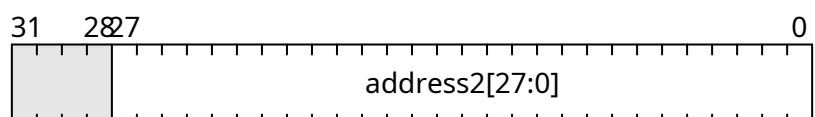


Fig. 21.28: `ROWHAMMER_ADDRESS2`

ROWHAMMER_COUNT

Address: $0xf0002000 + 0xc = 0xf000200c$

This is the number of DMA accesses performed. When the module is enabled, the value can be freely read. When the module is disabled, the register is clear-on-write and has to be read before the next attack.

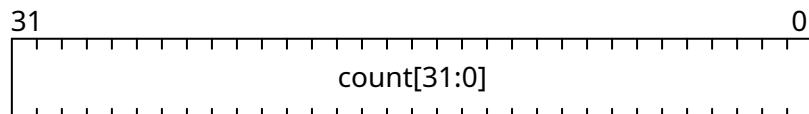


Fig. 21.29: ROWHAMMER_COUNT

21.2.6 WRITER

DMA DRAM writer.

Allows to fill DRAM with a predefined pattern using DMA.

Pattern

Provides access to RAM to store access pattern: *mem_addr* and *mem_data*. The pattern address space can be limited using the *data_mask*.

For example, having *mem_addr* filled with $[0x04, 0x02, 0x03, \dots]$ and *mem_data* filled with $[0xff, 0xaa, 0x55, \dots]$ and setting *data_mask* = *0b01*, the pattern $[(address, data), \dots]$ written will be: $[(0x04, 0xff), (0x02, 0xaa), (0x04, 0xff), \dots]$ (wraps due to masking).

DRAM memory range that is being accessed can be configured using *mem_mask*.

To use this module, make sure that *ready* is 1, then write the desired number of transfers to *count*. Writing to the *start* CSR will initialize the operation. When the operation is ongoing *ready* will be 0.

Register Listing for WRITER

Register	Address
WRITER_START	0xf0002800
WRITER_READY	0xf0002804
WRITER_MODULO	0xf0002808
WRITER_COUNT	0xf000280c
WRITER_DONE	0xf0002810
WRITER_MEM_MASK	0xf0002814
WRITER_DATA_MASK	0xf0002818
WRITER_DATA_DIV	0xf000281c
WRITER_INVERTER_DIVISOR_MASK	0xf0002820
WRITER_INVERTER_SELECTION_MASK	0xf0002824
WRITER_LAST_ADDRESS	0xf0002828

WRITER_START

Address: $0xf0002800 + 0x0 = 0xf0002800$

Write to the register starts the transfer (if ready=1)



Fig. 21.30: WRITER_START

WRITER_READY

Address: $0xf0002800 + 0x4 = 0xf0002804$

Indicates that the transfer is not ongoing

WRITER_MODULO

Address: $0xf0002800 + 0x8 = 0xf0002808$

When set use modulo to calculate DMA transfers address rather than bit masking

WRITER_COUNT

Address: $0xf0002800 + 0xc = 0xf000280c$

Desired number of DMA transfers

WRITER_DONE

Address: $0xf0002800 + 0x10 = 0xf0002810$

Number of completed DMA transfers



Fig. 21.31: WRITER_READY



Fig. 21.32: WRITER_MODULO

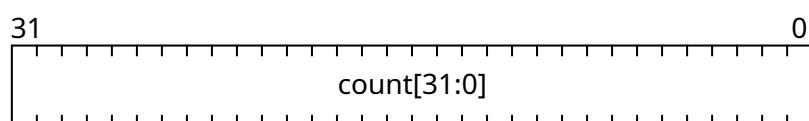


Fig. 21.33: WRITER_COUNT

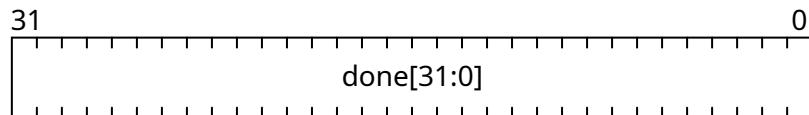


Fig. 21.34: WRITER_DONE

WRITER_MEM_MASK

Address: $0xf0002800 + 0x14 = 0xf0002814$

DRAM address mask for DMA transfers

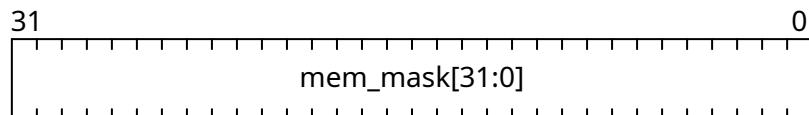


Fig. 21.35: WRITER_MEM_MASK

WRITER_DATA_MASK

Address: $0xf0002800 + 0x18 = 0xf0002818$

Pattern memory address mask

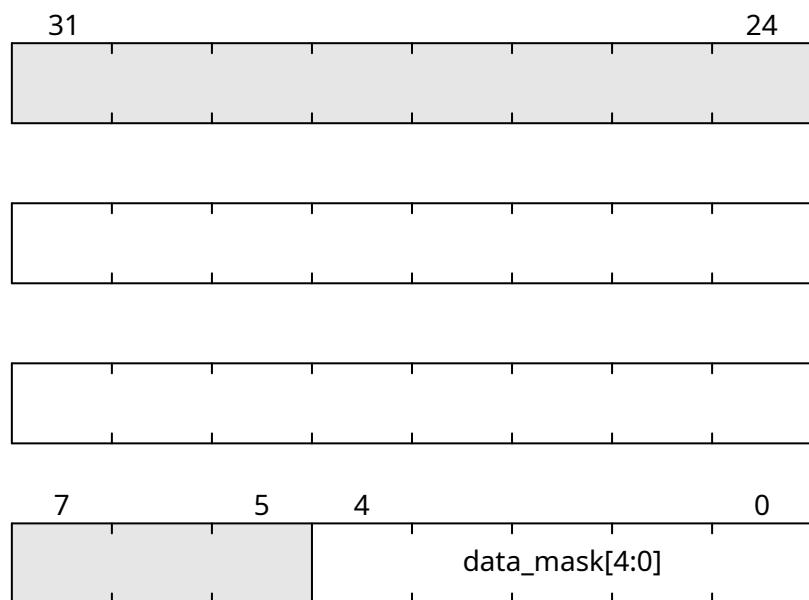


Fig. 21.36: WRITER_DATA_MASK

WRITER_DATA_DIV

Address: $0xf0002800 + 0x1c = 0xf000281c$

Pattern memory address divisor-1

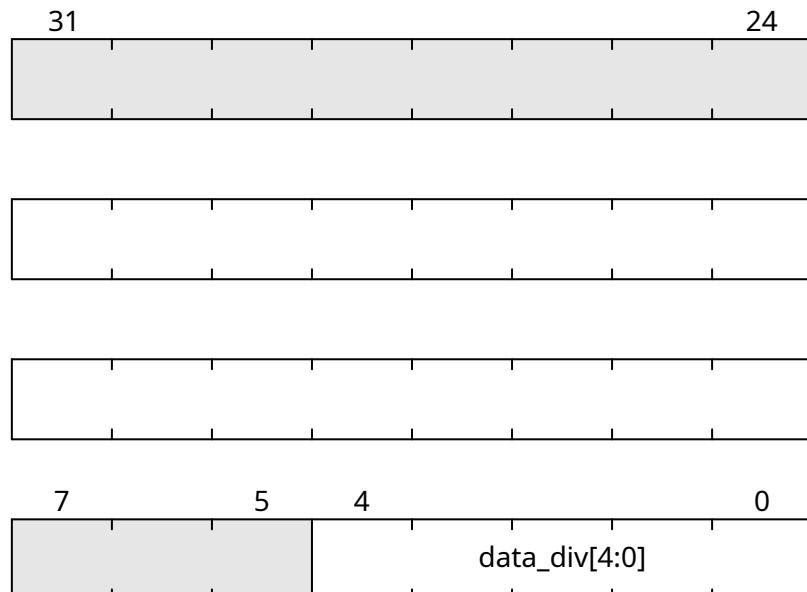


Fig. 21.37: WRITER_DATA_DIV

WRITER_INVERTER_DIVISOR_MASK

Address: $0xf0002800 + 0x20 = 0xf0002820$

Divisor mask for selecting rows for which pattern data gets inverted

WRITER_INVERTER_SELECTION_MASK

Address: $0xf0002800 + 0x24 = 0xf0002824$

Selection mask for selecting rows for which pattern data gets inverted

WRITER_LAST_ADDRESS

Address: $0xf0002800 + 0x28 = 0xf0002828$

Number of completed DMA transfers

21.2.7 READER

DMA DRAM reader.

Allows to check DRAM contents against a predefined pattern using DMA.

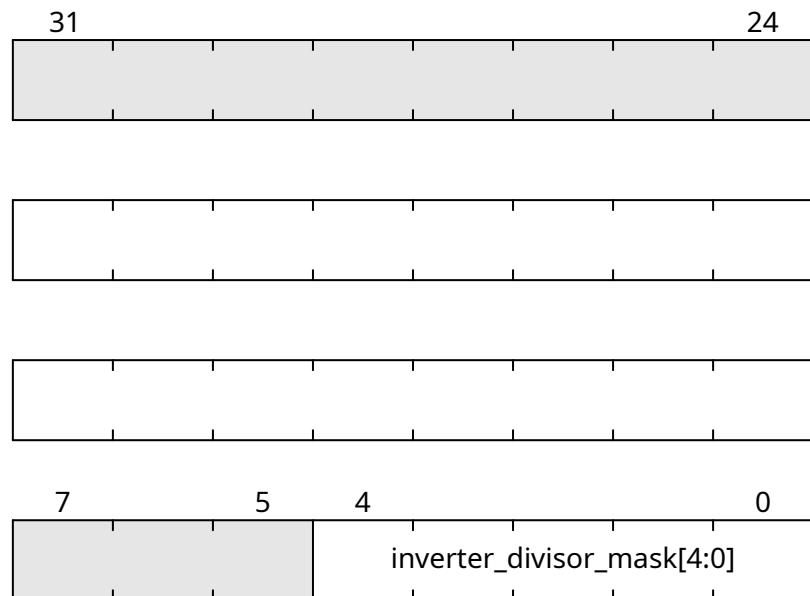


Fig. 21.38: WRITER_INVERTER_DIVISOR_MASK

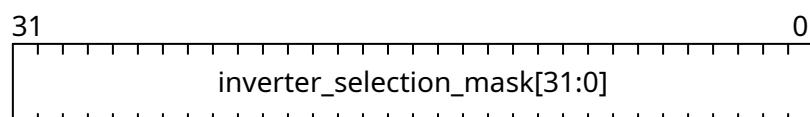


Fig. 21.39: WRITER_INVERTER_SELECTION_MASK

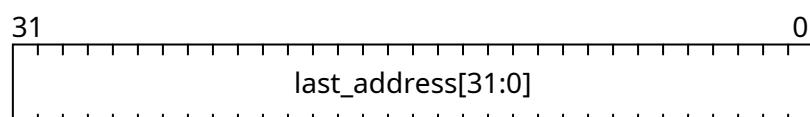


Fig. 21.40: WRITER_LAST_ADDRESS

Pattern

Provides access to RAM to store access pattern: *mem_addr* and *mem_data*. The pattern address space can be limited using the *data_mask*.

For example, having *mem_addr* filled with [0x04, 0x02, 0x03, ...] and *mem_data* filled with [0xff, 0xaa, 0x55, ...] and setting *data_mask* = 0b01, the pattern [(address, data), ...] written will be: [(0x04, 0xff), (0x02, 0xaa), (0x04, 0xff), ...] (wraps due to masking).

DRAM memory range that is being accessed can be configured using *mem_mask*.

To use this module, make sure that *ready* is 1, then write the desired number of transfers to *count*. Writing to the *start* CSR will initialize the operation. When the operation is ongoing *ready* will be 0.

Reading errors

This module allows to check the locations of errors in the memory. It scans the configured memory area and compares the values read to the predefined pattern. If *skip_fifo* is 0, this module will stop after each error encountered, so that it can be examined. Wait until the *error_ready* CSR is 1. Then use the CSRs *error_offset*, *error_data* and *error_expected* to examine the errors in the current transfer. To continue reading, write 1 to *error_continue* CSR. Setting *skip_fifo* to 1 will disable this behavior entirely.

The final number of errors can be read from *error_count*. NOTE: This value represents the number of erroneous *DMA transfers*.

The current progress can be read from the *done* CSR.

Register Listing for READER

Register	Address
<i>READER_START</i>	0xf0003000
<i>READER_READY</i>	0xf0003004
<i>READER_MODULO</i>	0xf0003008
<i>READER_COUNT</i>	0xf000300c
<i>READER_DONE</i>	0xf0003010
<i>READER_MEM_MASK</i>	0xf0003014
<i>READER_DATA_MASK</i>	0xf0003018
<i>READER_DATA_DIV</i>	0xf000301c
<i>READER_INVERTER_DIVISOR_MASK</i>	0xf0003020
<i>READER_INVERTER_SELECTION_MASK</i>	0xf0003024
<i>READER_ERROR_COUNT</i>	0xf0003028
<i>READER_SKIP_FIFO</i>	0xf000302c
<i>READER_ERROR_OFFSET</i>	0xf0003030
<i>READER_ERROR_DATA7</i>	0xf0003034
<i>READER_ERROR_DATA6</i>	0xf0003038
<i>READER_ERROR_DATA5</i>	0xf000303c
<i>READER_ERROR_DATA4</i>	0xf0003040
<i>READER_ERROR_DATA3</i>	0xf0003044
<i>READER_ERROR_DATA2</i>	0xf0003048
<i>READER_ERROR_DATA1</i>	0xf000304c

continues on next page

Table 21.1 – continued from previous page

Register	Address
<i>READER_ERROR_DATA0</i>	0xf0003050
<i>READER_ERROR_EXPECTED7</i>	0xf0003054
<i>READER_ERROR_EXPECTED6</i>	0xf0003058
<i>READER_ERROR_EXPECTED5</i>	0xf000305c
<i>READER_ERROR_EXPECTED4</i>	0xf0003060
<i>READER_ERROR_EXPECTED3</i>	0xf0003064
<i>READER_ERROR_EXPECTED2</i>	0xf0003068
<i>READER_ERROR_EXPECTED1</i>	0xf000306c
<i>READER_ERROR_EXPECTED0</i>	0xf0003070
<i>READER_ERROR_READY</i>	0xf0003074
<i>READER_ERROR_CONTINUE</i>	0xf0003078

READER_START

Address: $0xf0003000 + 0x0 = 0xf0003000$

Write to the register starts the transfer (if ready=1)



Fig. 21.41: READER_START

READER_READY

Address: $0xf0003000 + 0x4 = 0xf0003004$

Indicates that the transfer is not ongoing



Fig. 21.42: READER_READY

READER_MODULO

Address: $0xf0003000 + 0x8 = 0xf0003008$

When set use modulo to calculate DMA transfers address rather than bit masking

READER_COUNT

Address: $0xf0003000 + 0xc = 0xf000300c$

Desired number of DMA transfers

READER_DONE

Address: $0xf0003000 + 0x10 = 0xf0003010$

Number of completed DMA transfers

READER_MEM_MASK

Address: $0xf0003000 + 0x14 = 0xf0003014$

DRAM address mask for DMA transfers

READER_DATA_MASK

Address: $0xf0003000 + 0x18 = 0xf0003018$

Pattern memory address mask



Fig. 21.43: READER_MODULO

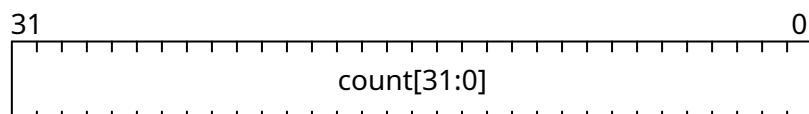


Fig. 21.44: READER_COUNT

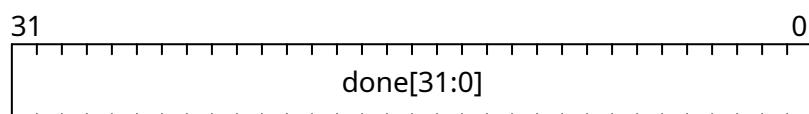


Fig. 21.45: READER_DONE

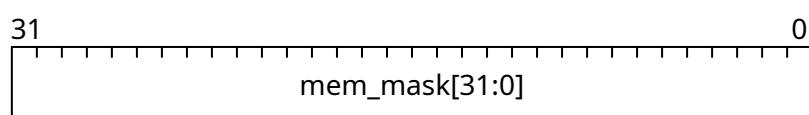


Fig. 21.46: READER_MEM_MASK

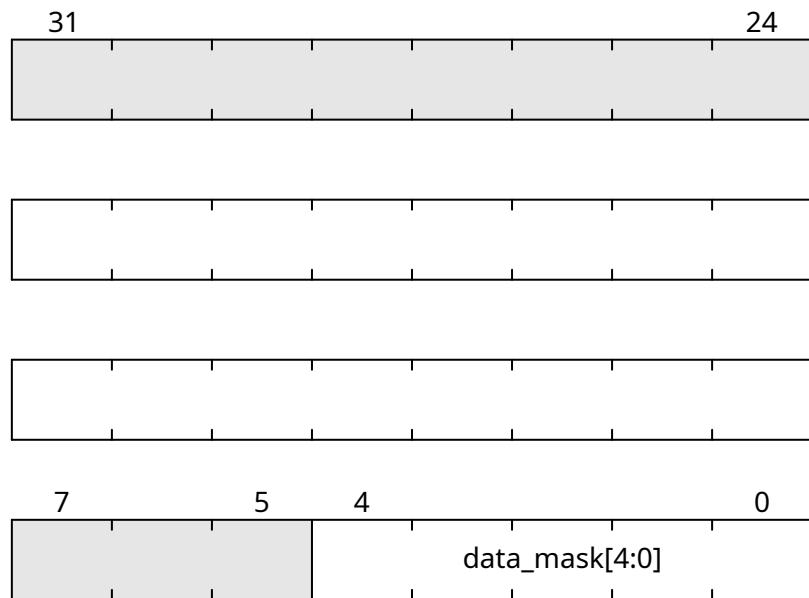


Fig. 21.47: READER_DATA_MASK

READER_DATA_DIV

Address: 0xf0003000 + 0x1c = 0xf000301c

Pattern memory address divisor-1

READER_INVERTER_DIVISOR_MASK

Address: 0xf0003000 + 0x20 = 0xf0003020

Divisor mask for selecting rows for which pattern data gets inverted

READER_INVERTER_SELECTION_MASK

Address: 0xf0003000 + 0x24 = 0xf0003024

Selection mask for selecting rows for which pattern data gets inverted

READER_ERROR_COUNT

Address: 0xf0003000 + 0x28 = 0xf0003028

Number of errors detected

READER_SKIP_FIFO

Address: 0xf0003000 + 0x2c = 0xf000302c

Skip waiting for user to read the errors FIFO

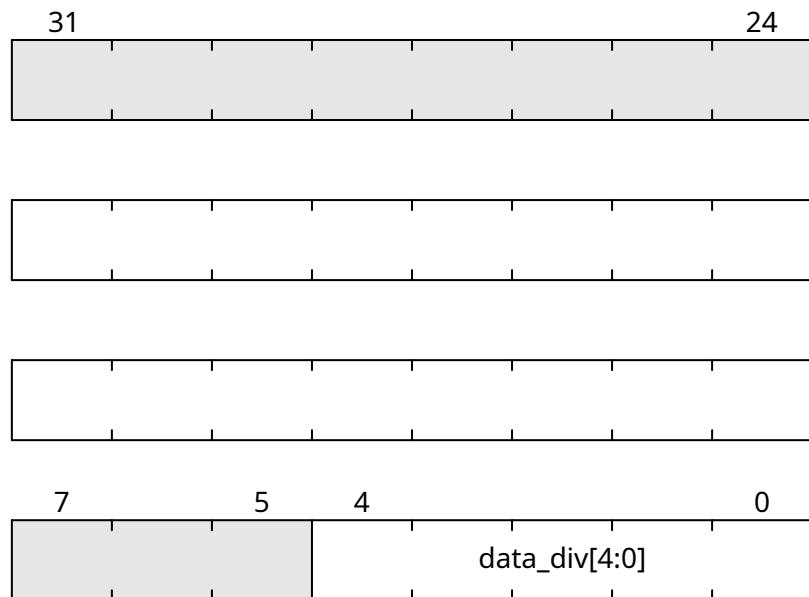


Fig. 21.48: READER_DATA_DIV

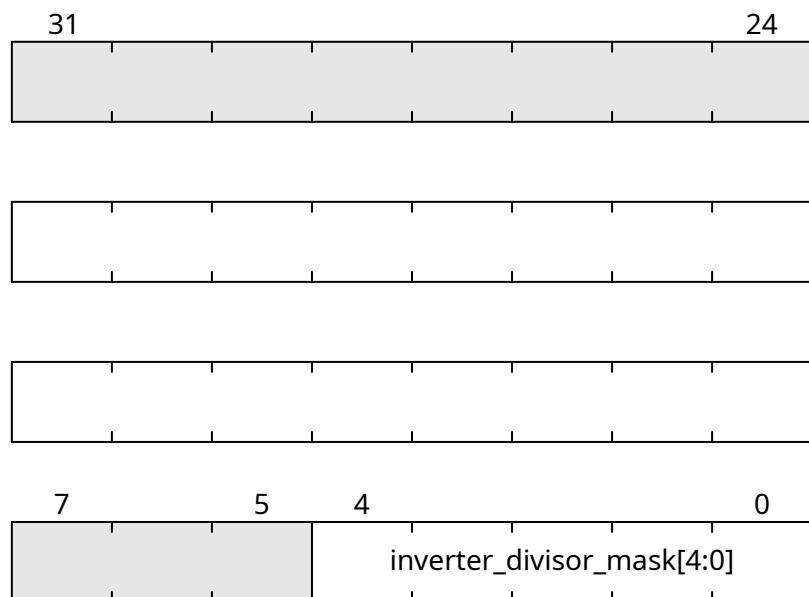


Fig. 21.49: READER_INVERTER_DIVISOR_MASK

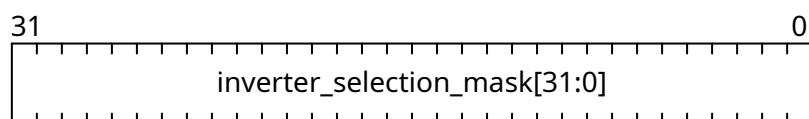


Fig. 21.50: READER_INVERTER_SELECTION_MASK

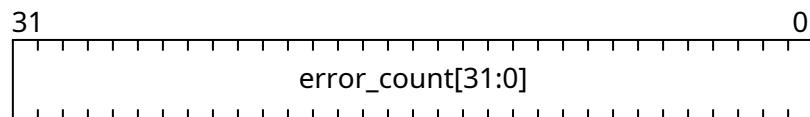


Fig. 21.51: READER_ERROR_COUNT



Fig. 21.52: READER_SKIP_FIFO

READER_ERROR_OFFSET

Address: $0xf0003000 + 0x30 = 0xf0003030$

Current offset of the error

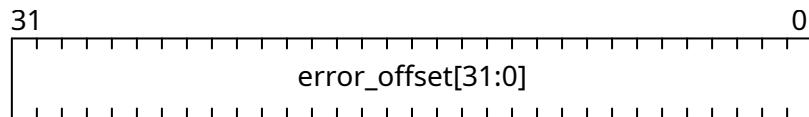


Fig. 21.53: READER_ERROR_OFFSET

READER_ERROR_DATA7

Address: $0xf0003000 + 0x34 = 0xf0003034$

Bits 224-255 of READER_ERROR_DATA. Erroneous value read from DRAM memory

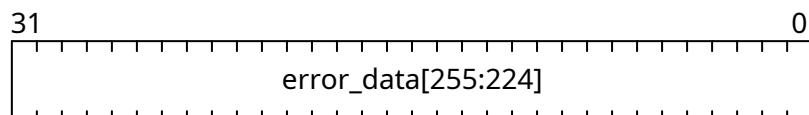


Fig. 21.54: READER_ERROR_DATA7

READER_ERROR_DATA6

Address: $0xf0003000 + 0x38 = 0xf0003038$

Bits 192-223 of READER_ERROR_DATA.

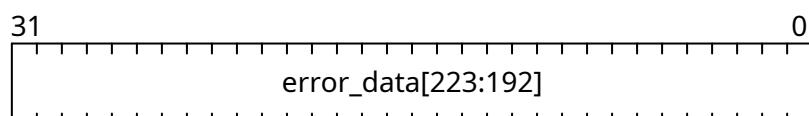


Fig. 21.55: READER_ERROR_DATA6

READER_ERROR_DATA5

Address: $0xf0003000 + 0x3c = 0xf000303c$

Bits 160-191 of READER_ERROR_DATA.

READER_ERROR_DATA4

Address: $0xf0003000 + 0x40 = 0xf0003040$

Bits 128-159 of READER_ERROR_DATA.

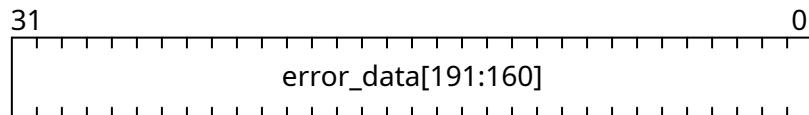


Fig. 21.56: READER_ERROR_DATA5

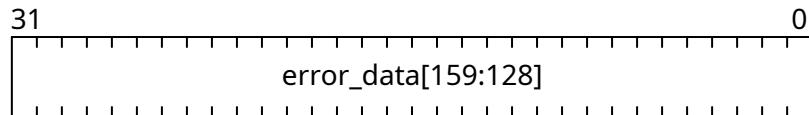


Fig. 21.57: READER_ERROR_DATA4

READER_ERROR_DATA3

Address: 0xf0003000 + 0x44 = 0xf0003044

Bits 96-127 of *READER_ERROR_DATA*.

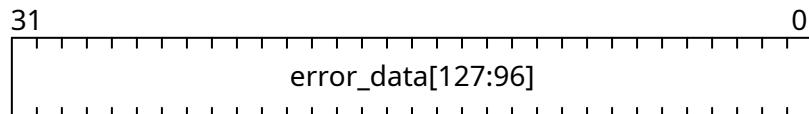


Fig. 21.58: READER_ERROR_DATA3

READER_ERROR_DATA2

Address: 0xf0003000 + 0x48 = 0xf0003048

Bits 64-95 of *READER_ERROR_DATA*.

READER_ERROR_DATA1

Address: 0xf0003000 + 0x4c = 0xf000304c

Bits 32-63 of *READER_ERROR_DATA*.

READER_ERROR_DATA0

Address: 0xf0003000 + 0x50 = 0xf0003050

Bits 0-31 of *READER_ERROR_DATA*.

READER_ERROR_EXPECTED7

Address: 0xf0003000 + 0x54 = 0xf0003054

Bits 224-255 of *READER_ERROR_EXPECTED*. Value expected to be read from DRAM memory

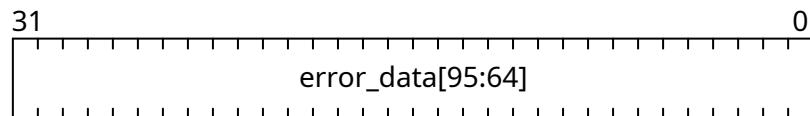


Fig. 21.59: READER_ERROR_DATA2

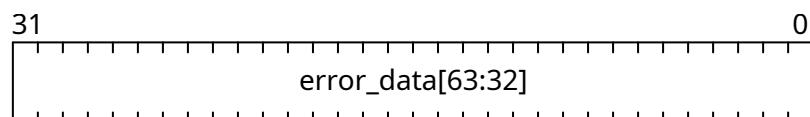


Fig. 21.60: READER_ERROR_DATA1

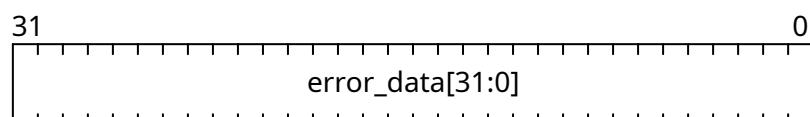


Fig. 21.61: READER_ERROR_DATA0

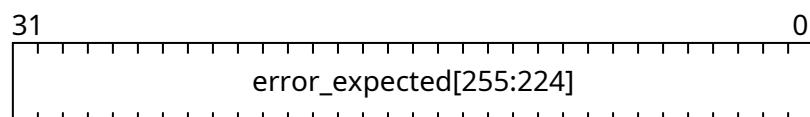


Fig. 21.62: READER_ERROR_EXPECTED7

READER_ERROR_EXPECTED6

Address: $0xf0003000 + 0x58 = 0xf0003058$

Bits 192-223 of *READER_ERROR_EXPECTED*.

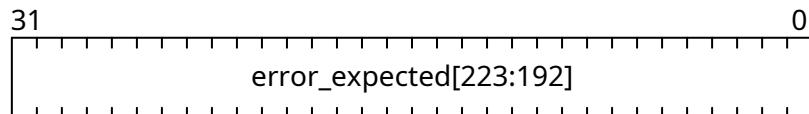


Fig. 21.63: READER_ERROR_EXPECTED6

READER_ERROR_EXPECTED5

Address: $0xf0003000 + 0x5c = 0xf000305c$

Bits 160-191 of *READER_ERROR_EXPECTED*.

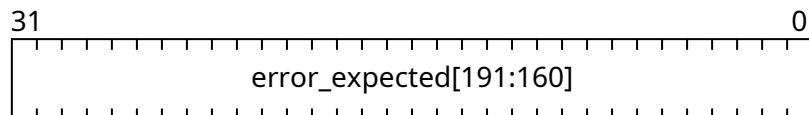


Fig. 21.64: READER_ERROR_EXPECTED5

READER_ERROR_EXPECTED4

Address: $0xf0003000 + 0x60 = 0xf0003060$

Bits 128-159 of *READER_ERROR_EXPECTED*.

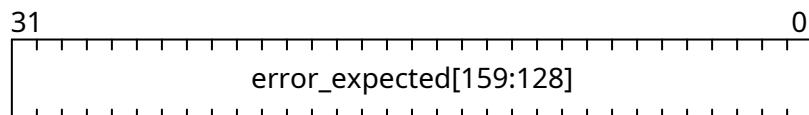


Fig. 21.65: READER_ERROR_EXPECTED4

READER_ERROR_EXPECTED3

Address: $0xf0003000 + 0x64 = 0xf0003064$

Bits 96-127 of *READER_ERROR_EXPECTED*.

READER_ERROR_EXPECTED2

Address: $0xf0003000 + 0x68 = 0xf0003068$

Bits 64-95 of *READER_ERROR_EXPECTED*.

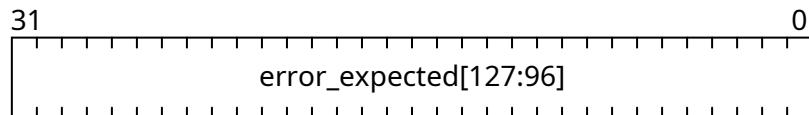


Fig. 21.66: READER_ERROR_EXPECTED3

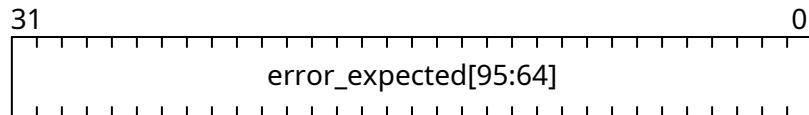


Fig. 21.67: READER_ERROR_EXPECTED2

READER_ERROR_EXPECTED1

Address: $0xf0003000 + 0x6c = 0xf000306c$

Bits 32-63 of *READER_ERROR_EXPECTED*.

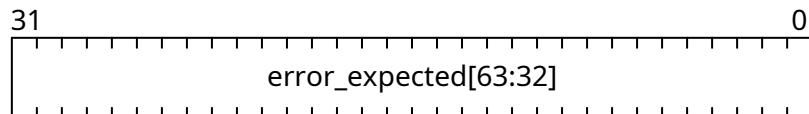


Fig. 21.68: READER_ERROR_EXPECTED1

READER_ERROR_EXPECTED0

Address: $0xf0003000 + 0x70 = 0xf0003070$

Bits 0-31 of *READER_ERROR_EXPECTED*.

READER_ERROR_READY

Address: $0xf0003000 + 0x74 = 0xf0003074$

Error detected and ready to read

READER_ERROR_CONTINUE

Address: $0xf0003000 + 0x78 = 0xf0003078$

Continue reading until the next error

21.2.8 DFI_SWITCH

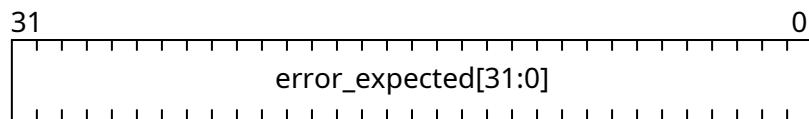


Fig. 21.69: READER_ERROR_EXPECTED0

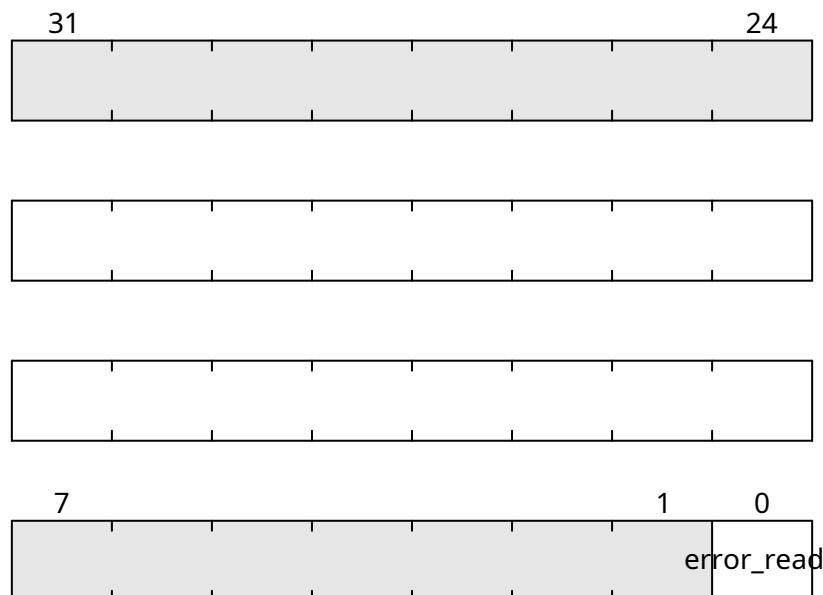


Fig. 21.70: READER_ERROR_READY

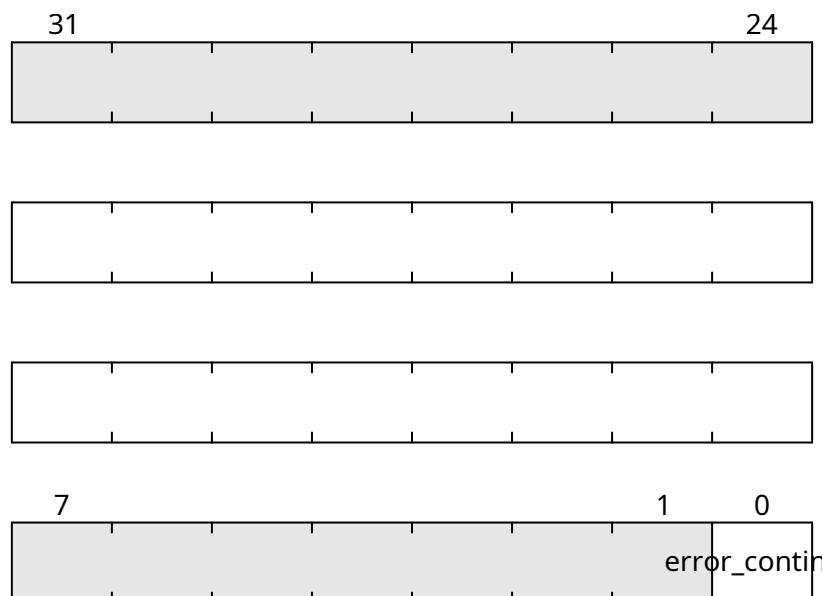


Fig. 21.71: READER_ERROR_CONTINUE

Register Listing for DFI_SWITCH

Register	Address
<i>DFI_SWITCH_REFRESH_COUNT1</i>	0xf0003800
<i>DFI_SWITCH_REFRESH_COUNT0</i>	0xf0003804
<i>DFI_SWITCH_AT_REFRESH1</i>	0xf0003808
<i>DFI_SWITCH_AT_REFRESH0</i>	0xf000380c
<i>DFI_SWITCH_REFRESH_UPDATE</i>	0xf0003810

DFI_SWITCH_REFRESH_COUNT1

Address: $0xf0003800 + 0x0 = 0xf0003800$

Bits 32-63 of *DFI_SWITCH_REFRESH_COUNT*. Count of all refresh commands issued (both by Memory Controller and the Payload Executor). Value is latched from internal counter on mode transition: MC -> PE or by writing to the *refresh_update* CSR.

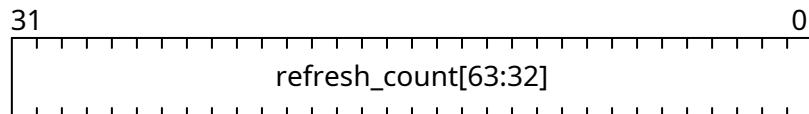


Fig. 21.72: DFI_SWITCH_REFRESH_COUNT1

DFI_SWITCH_REFRESH_COUNT0

Address: $0xf0003800 + 0x4 = 0xf0003804$

Bits 0-31 of *DFI_SWITCH_REFRESH_COUNT*.

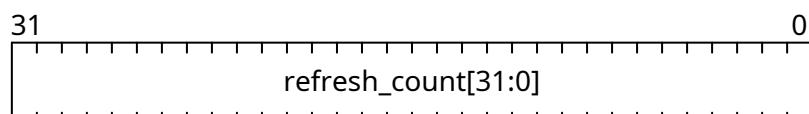


Fig. 21.73: DFI_SWITCH_REFRESH_COUNT0

DFI_SWITCH_AT_REFRESH1

Address: $0xf0003800 + 0x8 = 0xf0003808$

Bits 32-63 of *DFI_SWITCH_AT_REFRESH*. If set to a value different than 0 the mode transition MC -> PE will be performed only when the value of this register matches the current refresh commands count.

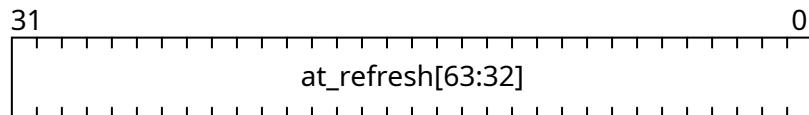


Fig. 21.74: DFI_SWITCH_AT_REFRESH1

DFI_SWITCH_AT_REFRESH0

Address: $0xf0003800 + 0xc = 0xf000380c$

Bits 0-31 of *DFI_SWITCH_AT_REFRESH*.

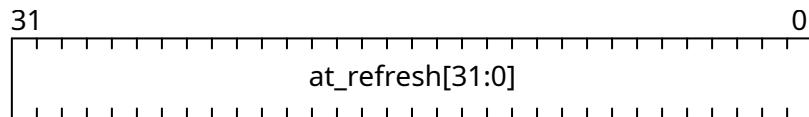


Fig. 21.75: DFI_SWITCH_AT_REFRESH0

DFI_SWITCH_REFRESH_UPDATE

Address: $0xf0003800 + 0x10 = 0xf0003810$

Force an update of the *refresh_count* CSR.

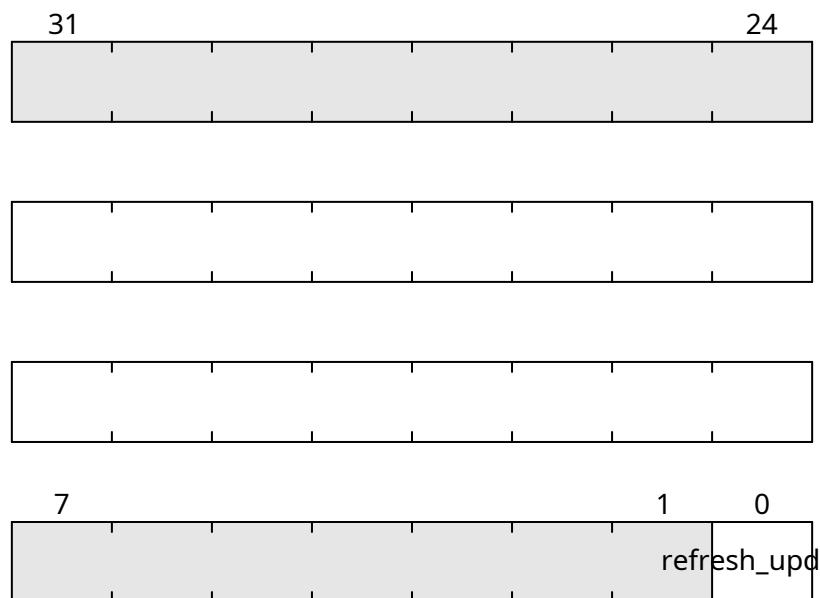


Fig. 21.76: DFI_SWITCH_REFRESH_UPDATE

21.2.9 PAYLOAD_EXECUTOR

Executes the DRAM payload from memory

Instruction decoder

All instructions are 32-bit. The format of most instructions is the same, except for the LOOP instruction, which has a constant TIMESLICE of 1.

NOOP with a TIMESLICE of 0 is a special case which is interpreted as STOP instruction. When this instruction is encountered execution gets finished immediately.

NOTE: TIMESLICE is the number of cycles the instruction will take. This means that instructions other than NOOP that use TIMESLICE=0 are illegal (although will silently be executed as having TIMESLICE=1).

NOTE2: LOOP instruction will *jump* COUNT times, meaning that the “code” inside the loop will effectively be executed COUNT+1 times.

Op codes:

Op	Value
NOOP	0b000
LOOP	0b111
ACT	0b100
PRE	0b101
REF	0b110
ZQC	0b001
READ	0b010

Instruction format:

LSB	MSB
dfi:	OP_CODE TIMESLICE ADDRESS
noop:	OP_CODE TIMESLICE_NOOP
loop:	OP_CODE COUNT JUMP
stop:	<NOOP> 0

Where ADDRESS depends on the DFI command and is one of:

LSB	MSB
RANK BANK COLUMN	
RANK BANK ROW	

Register Listing for PAYLOAD_EXECUTOR

Register	Address
PAYLOAD_EXECUTOR_START	0xf0004000
PAYLOAD_EXECUTOR_STATUS	0xf0004004
PAYLOAD_EXECUTOR_READ_COUNT	0xf0004008
PAYLOAD_EXECUTOR_EXEC_START1	0xf000400c
PAYLOAD_EXECUTOR_EXEC_START0	0xf0004010
PAYLOAD_EXECUTOR_EXEC_STOP1	0xf0004014
PAYLOAD_EXECUTOR_EXEC_STOP0	0xf0004018

PAYLOAD_EXECUTOR_START

Address: $0xf0004000 + 0x0 = 0xf0004000$

Writing to this register initializes payload execution



Fig. 21.77: PAYLOAD_EXECUTOR_START

PAYLOAD_EXECUTOR_STATUS

Address: $0xf0004000 + 0x4 = 0xf0004004$

Payload executor status register

Field	Name	Description
[0]	READY	Indicates that the executor is not running
[1]	OVERFLOW	Indicates the scratchpad memory address counter has overflowed due to the number of READ commands sent during execution

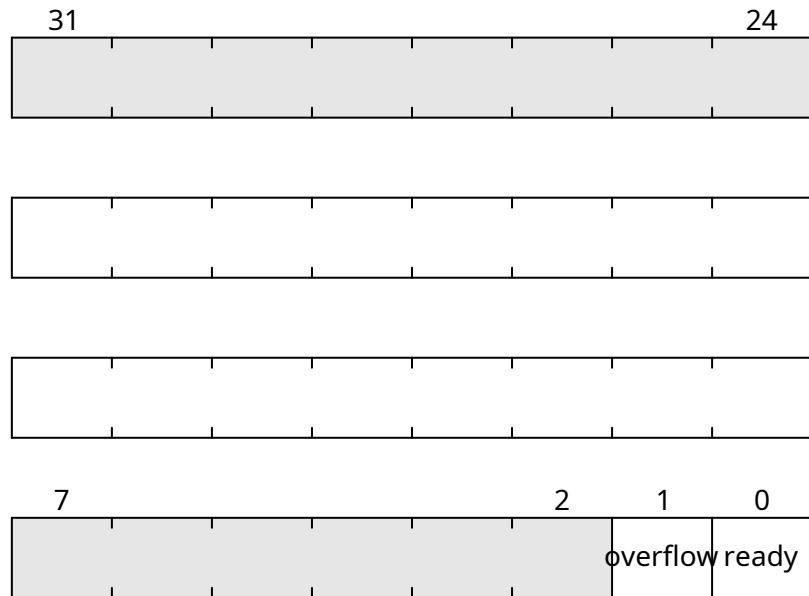


Fig. 21.78: PAYLOAD_EXECUTOR_STATUS

PAYLOAD_EXECUTOR_READ_COUNT

Address: $0xf0004000 + 0x8 = 0xf0004008$

Number of data from READ commands that is stored in the scratchpad memory

PAYLOAD_EXECUTOR_EXEC_START1

Address: $0xf0004000 + 0xc = 0xf000400c$

Bits 32-63 of PAYLOAD_EXECUTOR_EXEC_START. Number of cycles elapsed until the start of the payload execution.

PAYLOAD_EXECUTOR_EXEC_START0

Address: $0xf0004000 + 0x10 = 0xf0004010$

Bits 0-31 of PAYLOAD_EXECUTOR_EXEC_START.

PAYLOAD_EXECUTOR_EXEC_STOP1

Address: $0xf0004000 + 0x14 = 0xf0004014$

Bits 32-63 of PAYLOAD_EXECUTOR_EXEC_STOP. Number of cycles elapsed until the end of the payload execution.

PAYLOAD_EXECUTOR_EXEC_STOP0

Address: $0xf0004000 + 0x18 = 0xf0004018$

Bits 0-31 of PAYLOAD_EXECUTOR_EXEC_STOP.

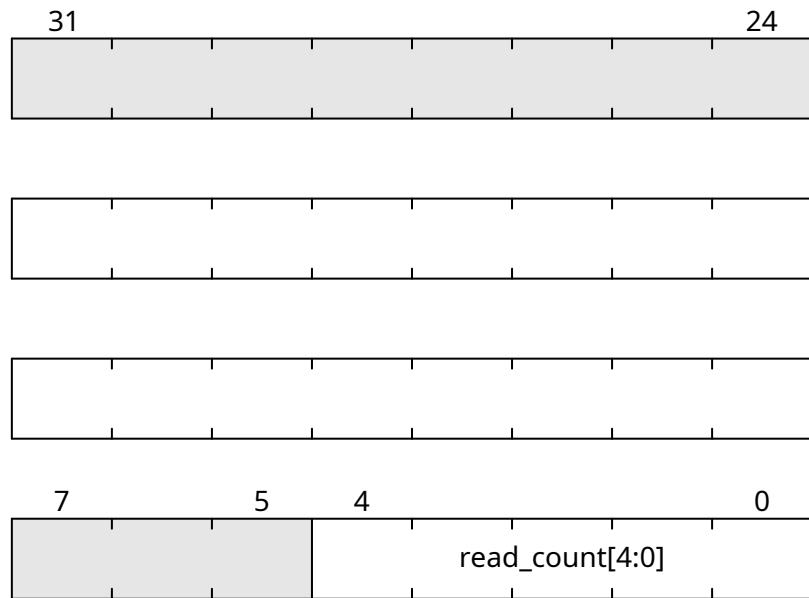


Fig. 21.79: PAYLOAD_EXECUTOR_READ_COUNT

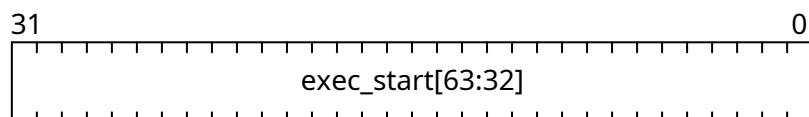


Fig. 21.80: PAYLOAD_EXECUTOR_EXEC_START1

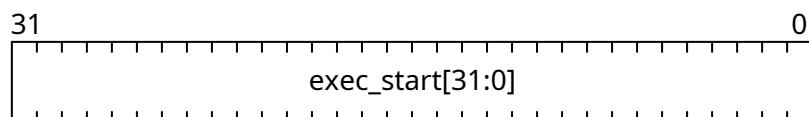


Fig. 21.81: PAYLOAD_EXECUTOR_EXEC_START0

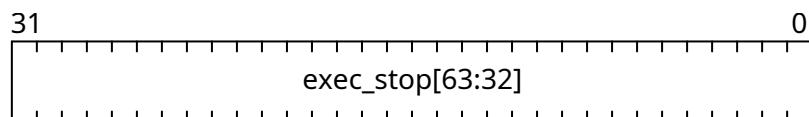


Fig. 21.82: PAYLOAD_EXECUTOR_EXEC_STOP1

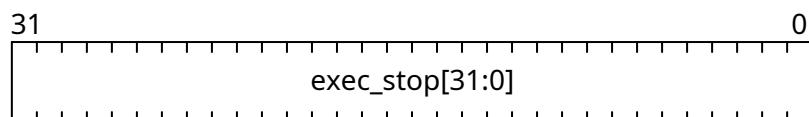


Fig. 21.83: PAYLOAD_EXECUTOR_EXEC_STOP0

21.2.10 I2C

Register Listing for I2C

Register	Address
<i>I2C_W</i>	0xf0004800
<i>I2C_R</i>	0xf0004804
<i>I2C_WORKER</i>	0xf0004808
<i>I2C_I2C_WORKER_START</i>	0xf000480c
<i>I2C_I2C_WORKER_I2C_CTRL</i>	0xf0004810
<i>I2C_I2C_WORKER_I2C_STATE</i>	0xf0004814
<i>I2C_I2C_WORKER_FIFOS_ACCESS_PORT</i>	0xf0004818
<i>I2C_I2C_WORKER_WRITE_FIFO_STATE</i>	0xf000481c
<i>I2C_I2C_WORKER_READ_FIFO_STATE</i>	0xf0004820

I2C_W

Address: $0xf0004800 + 0x0 = 0xf0004800$

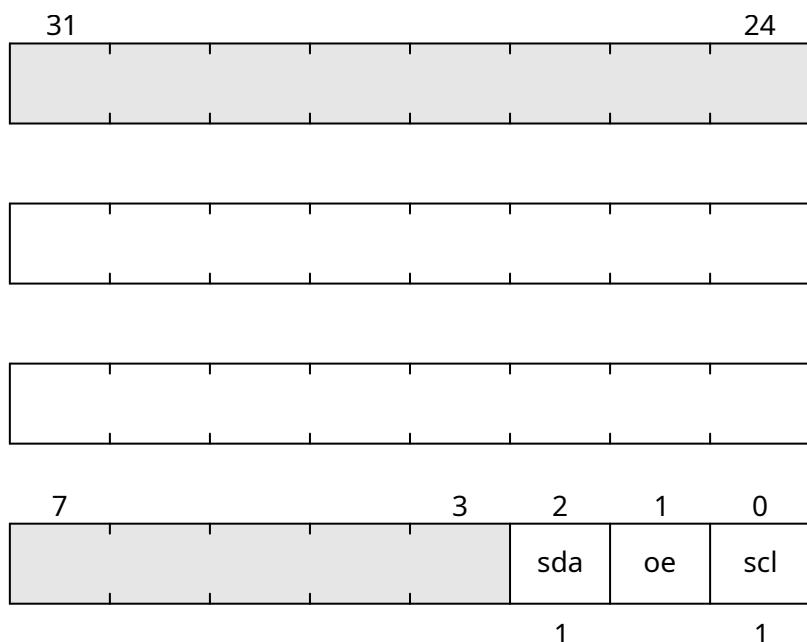


Fig. 21.84: I2C_W

Field	Name	Description

I2C_R

Address: $0xf0004800 + 0x4 = 0xf0004804$



Fig. 21.85: I2C_R

Field	Name	Description

I2C_WORKER

Address: $0xf0004800 + 0x8 = 0xf0004808$

Field	Name	Description

I2C_I2C_WORKER_START

Address: $0xf0004800 + 0xc = 0xf000480c$

I2C_I2C_WORKER_I2C_CTRL

Address: $0xf0004800 + 0x10 = 0xf0004810$

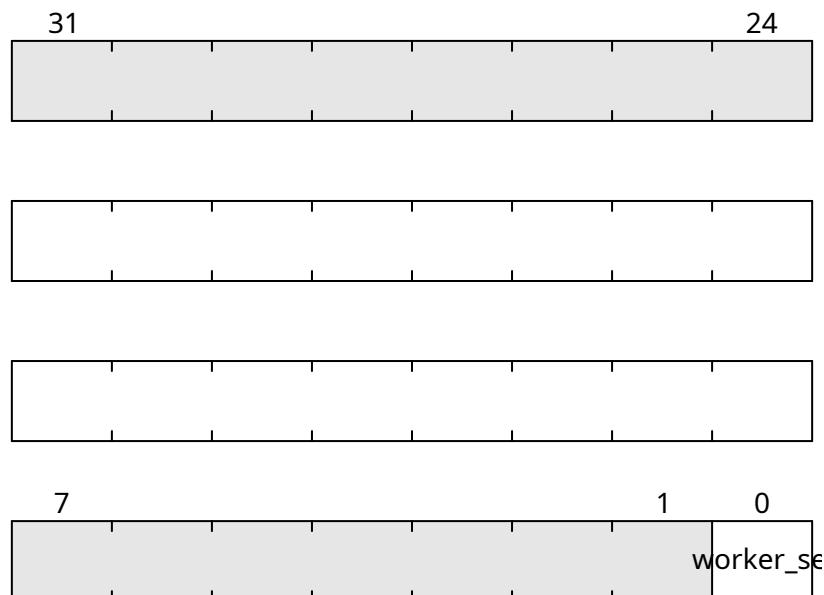


Fig. 21.86: I2C_WORKER

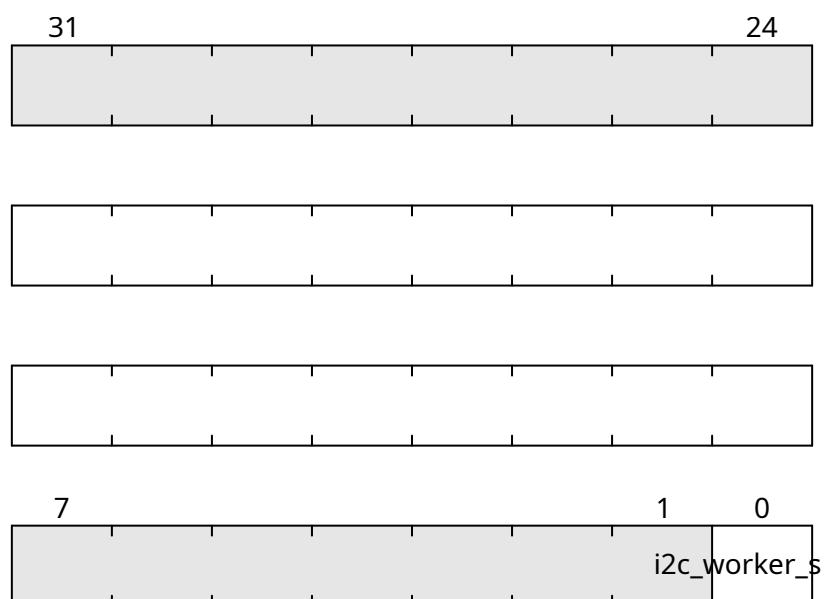


Fig. 21.87: I2C_I2C_WORKER_START



Fig. 21.88: I2C_I2C_WORKER_I2C_CTRL

Field	Name	Description

I2C_I2C_WORKER_I2C_STATE

Address: $0xf0004800 + 0x14 = 0xf0004814$

Field	Name	Description

I2C_I2C_WORKER_FIFOS_ACCESS_PORT

Address: $0xf0004800 + 0x18 = 0xf0004818$

I2C_I2C_WORKER_WRITE_FIFO_STATE

Address: $0xf0004800 + 0x1c = 0xf000481c$

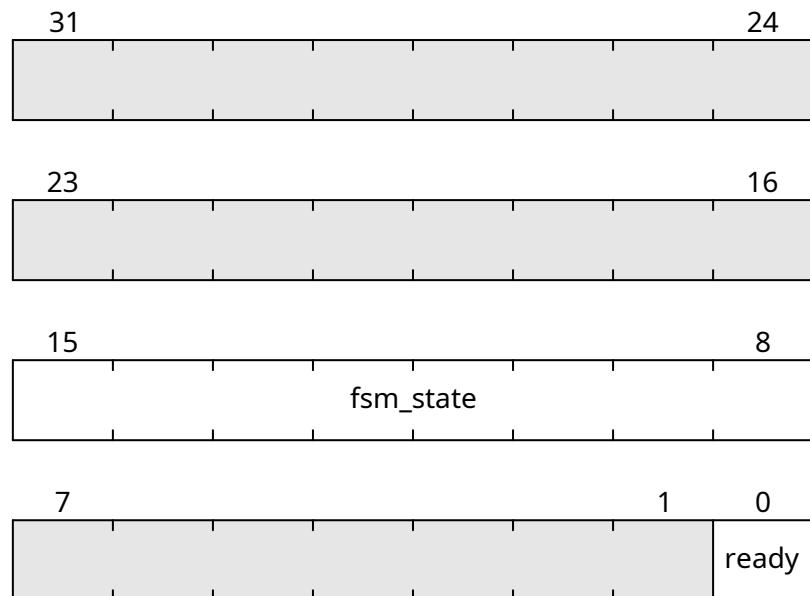


Fig. 21.89: `I2C_I2C_WORKER_I2C_STATE`

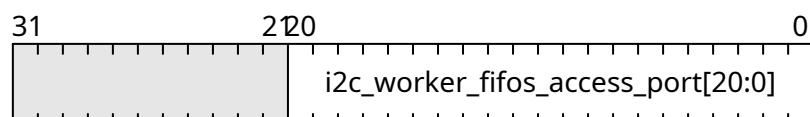


Fig. 21.90: `I2C_I2C_WORKER_FIFOS_ACCESS_PORT`

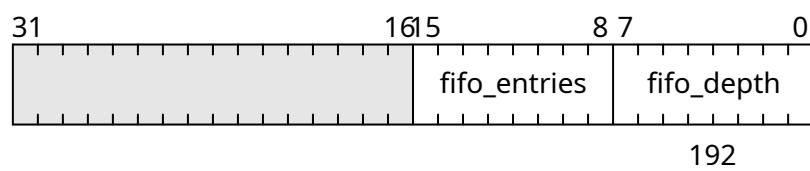


Fig. 21.91: `I2C_I2C_WORKER_WRITE_FIFO_STATE`

Field	Name	Description

I2C_I2C_WORKER_READ_FIFO_STATE

Address: $0xf0004800 + 0x20 = 0xf0004820$

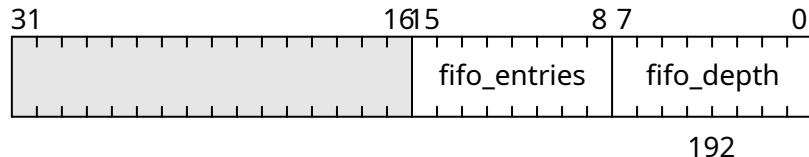


Fig. 21.92: I2C_I2C_WORKER_READ_FIFO_STATE

Field	Name	Description

21.2.11 CTRL

Register Listing for CTRL

Register	Address
<i>CTRL_RESET</i>	<i>0xf0005000</i>
<i>CTRL_SCRATCH</i>	<i>0xf0005004</i>
<i>CTRL_BUS_ERRORS</i>	<i>0xf0005008</i>

CTRL_RESET

Address: $0xf0005000 + 0x0 = 0xf0005000$

Field	Name	Description
[0]	SOC_RST	Write 1 to this register to reset the full SoC (Pulse Reset)
[1]	CPU_RST	Write 1 to this register to reset the CPU(s) of the SoC (Hold Reset)

CTRL_SCRATCH

Address: $0xf0005000 + 0x4 = 0xf0005004$

Use this register as a scratch space to verify that software read/write accesses to the Wishbone/CSR bus are working correctly. The initial reset value of 0x1234578 can be used to verify endianness.

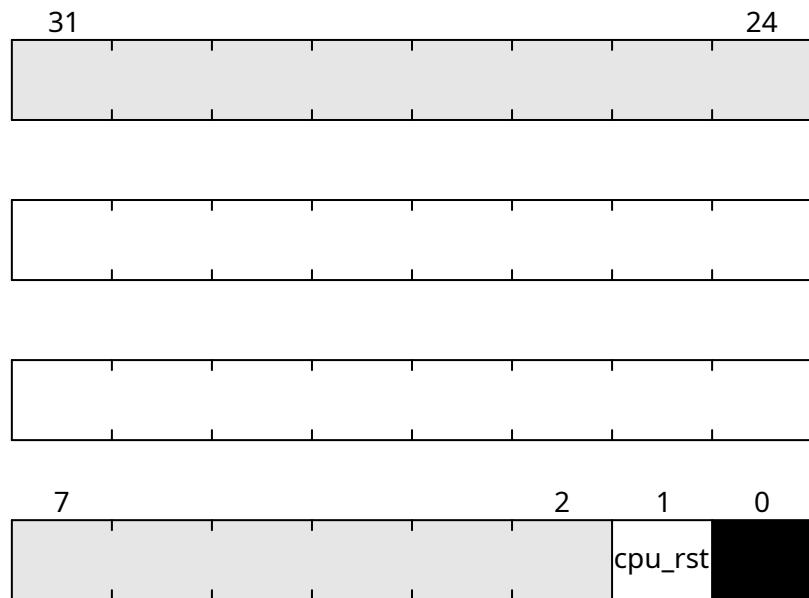


Fig. 21.93: CTRL_RESET

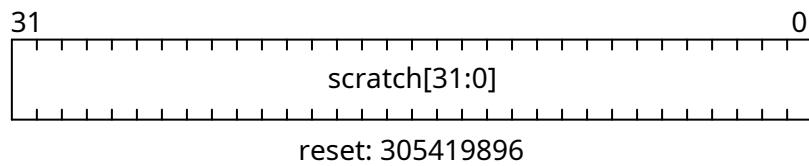


Fig. 21.94: CTRL_SCRATCH

CTRL_BUS_ERRORS

Address: $0xf0005000 + 0x8 = 0xf0005008$

Total number of Wishbone bus errors (timeouts) since start.

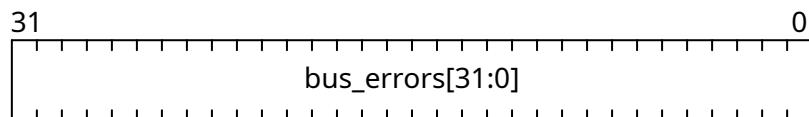


Fig. 21.95: CTRL_BUS_ERRORS

21.2.12 ETPHY

Register Listing for ETPHY

Register	Address
<i>ETPHY_CRG_RESET</i>	<i>0xf0005800</i>
<i>ETPHY_MDIO_W</i>	<i>0xf0005804</i>
<i>ETPHY_MDIO_R</i>	<i>0xf0005808</i>

ETPHY_CRG_RESET

Address: $0xf0005800 + 0x0 = 0xf0005800$



Fig. 21.96: ETPHY_CRG_RESET

ETHPHY_MDIO_W

Address: $0xf0005800 + 0x4 = 0xf0005804$

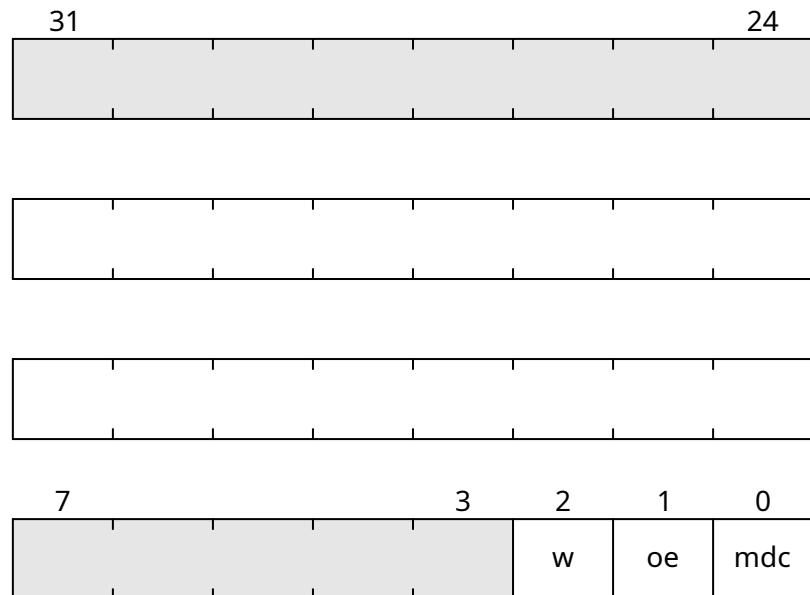


Fig. 21.97: ETHPHY_MDIO_W

Field	Name	Description

ETHPHY_MDIO_R

Address: $0xf0005800 + 0x8 = 0xf0005808$

Field	Name	Description

21.2.13 IDENTIFIER_MEM

Register Listing for IDENTIFIER_MEM

Register	Address
<i>IDENTIFIER_MEM</i>	<i>0xf0006000</i>



Fig. 21.98: ETHPHY_MDIO_R

IDENTIFIER_MEM

Address: $0xf0006000 + 0x0 = 0xf0006000$

8 x 109-bit memory

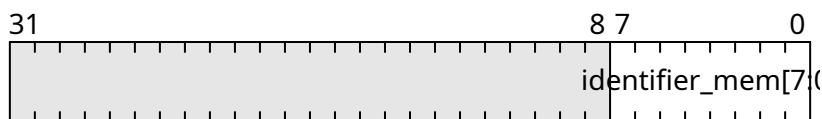


Fig. 21.99: IDENTIFIER_MEM

21.2.14 SDRAM

Register Listing for SDRAM

Register	Address
<i>SDRAM_DFII_CONTROL</i>	0xf0006800
<i>SDRAM_DFII_PIO_COMMAND</i>	0xf0006804
<i>SDRAM_DFII_PIO_COMMAND_ISSUE</i>	0xf0006808
<i>SDRAM_DFII_PIO_ADDRESS</i>	0xf000680c
<i>SDRAM_DFII_PIO_BADDRESS</i>	0xf0006810
<i>SDRAM_DFII_PIO_WRDAT A1</i>	0xf0006814
<i>SDRAM_DFII_PIO_WRDAT A0</i>	0xf0006818
<i>SDRAM_DFII_PIO_RDDAT A1</i>	0xf000681c
<i>SDRAM_DFII_PIO_RDDAT A0</i>	0xf0006820
<i>SDRAM_DFII_PI1_COMMAND</i>	0xf0006824

continues on next page

Table 21.2 – continued from previous page

Register	Address
<i>SDRAM_DFII_PI1_COMMAND_ISSUE</i>	0xf0006828
<i>SDRAM_DFII_PI1_ADDRESS</i>	0xf000682c
<i>SDRAM_DFII_PI1_BADDRESS</i>	0xf0006830
<i>SDRAM_DFII_PI1_WRDATA1</i>	0xf0006834
<i>SDRAM_DFII_PI1_WRDATA0</i>	0xf0006838
<i>SDRAM_DFII_PI1_RDDATA1</i>	0xf000683c
<i>SDRAM_DFII_PI1_RDDATA0</i>	0xf0006840
<i>SDRAM_DFII_PI2_COMMAND</i>	0xf0006844
<i>SDRAM_DFII_PI2_COMMAND_ISSUE</i>	0xf0006848
<i>SDRAM_DFII_PI2_ADDRESS</i>	0xf000684c
<i>SDRAM_DFII_PI2_BADDRESS</i>	0xf0006850
<i>SDRAM_DFII_PI2_WRDATA1</i>	0xf0006854
<i>SDRAM_DFII_PI2_WRDATA0</i>	0xf0006858
<i>SDRAM_DFII_PI2_RDDATA1</i>	0xf000685c
<i>SDRAM_DFII_PI2_RDDATA0</i>	0xf0006860
<i>SDRAM_DFII_PI3_COMMAND</i>	0xf0006864
<i>SDRAM_DFII_PI3_COMMAND_ISSUE</i>	0xf0006868
<i>SDRAM_DFII_PI3_ADDRESS</i>	0xf000686c
<i>SDRAM_DFII_PI3_BADDRESS</i>	0xf0006870
<i>SDRAM_DFII_PI3_WRDATA1</i>	0xf0006874
<i>SDRAM_DFII_PI3_WRDATA0</i>	0xf0006878
<i>SDRAM_DFII_PI3_RDDATA1</i>	0xf000687c
<i>SDRAM_DFII_PI3_RDDATA0</i>	0xf0006880
<i>SDRAM_CONTROLLER_TRP</i>	0xf0006884
<i>SDRAM_CONTROLLER_TRCD</i>	0xf0006888
<i>SDRAM_CONTROLLER_TWR</i>	0xf000688c
<i>SDRAM_CONTROLLER_TWTR</i>	0xf0006890
<i>SDRAM_CONTROLLER_TREFI</i>	0xf0006894
<i>SDRAM_CONTROLLER_TRFC</i>	0xf0006898
<i>SDRAM_CONTROLLER_TFAW</i>	0xf000689c
<i>SDRAM_CONTROLLER_TCCD</i>	0xf00068a0
<i>SDRAM_CONTROLLER_TCCD_WR</i>	0xf00068a4
<i>SDRAM_CONTROLLER_TRTP</i>	0xf00068a8
<i>SDRAM_CONTROLLER_TRRD</i>	0xf00068ac
<i>SDRAM_CONTROLLER_TRC</i>	0xf00068b0
<i>SDRAM_CONTROLLER_TRAS</i>	0xf00068b4
<i>SDRAM_CONTROLLER_TZQCS</i>	0xf00068b8
<i>SDRAM_CONTROLLER_LAST_ADDR_0</i>	0xf00068bc
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0</i>	0xf00068c0
<i>SDRAM_CONTROLLER_LAST_ADDR_1</i>	0xf00068c4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1</i>	0xf00068c8
<i>SDRAM_CONTROLLER_LAST_ADDR_2</i>	0xf00068cc
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2</i>	0xf00068d0
<i>SDRAM_CONTROLLER_LAST_ADDR_3</i>	0xf00068d4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3</i>	0xf00068d8
<i>SDRAM_CONTROLLER_LAST_ADDR_4</i>	0xf00068dc
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4</i>	0xf00068e0
<i>SDRAM_CONTROLLER_LAST_ADDR_5</i>	0xf00068e4

continues on next page

Table 21.2 – continued from previous page

Register	Address
SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5	0xf00068e8
SDRAM_CONTROLLER_LAST_ADDR_6	0xf00068ec
SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6	0xf00068f0
SDRAM_CONTROLLER_LAST_ADDR_7	0xf00068f4
SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7	0xf00068f8
SDRAM_CONTROLLER_LAST_ADDR_8	0xf00068fc
SDRAM_CONTROLLER_LAST_ACTIVE_ROW_8	0xf0006900
SDRAM_CONTROLLER_LAST_ADDR_9	0xf0006904
SDRAM_CONTROLLER_LAST_ACTIVE_ROW_9	0xf0006908
SDRAM_CONTROLLER_LAST_ADDR_10	0xf000690c
SDRAM_CONTROLLER_LAST_ACTIVE_ROW_10	0xf0006910
SDRAM_CONTROLLER_LAST_ADDR_11	0xf0006914
SDRAM_CONTROLLER_LAST_ACTIVE_ROW_11	0xf0006918
SDRAM_CONTROLLER_LAST_ADDR_12	0xf000691c
SDRAM_CONTROLLER_LAST_ACTIVE_ROW_12	0xf0006920
SDRAM_CONTROLLER_LAST_ADDR_13	0xf0006924
SDRAM_CONTROLLER_LAST_ACTIVE_ROW_13	0xf0006928
SDRAM_CONTROLLER_LAST_ADDR_14	0xf000692c
SDRAM_CONTROLLER_LAST_ACTIVE_ROW_14	0xf0006930
SDRAM_CONTROLLER_LAST_ADDR_15	0xf0006934
SDRAM_CONTROLLER_LAST_ACTIVE_ROW_15	0xf0006938

SDRAM_DFII_CONTROL

Address: $0xf0006800 + 0x0 = 0xf0006800$

Control DFI signals common to all phases

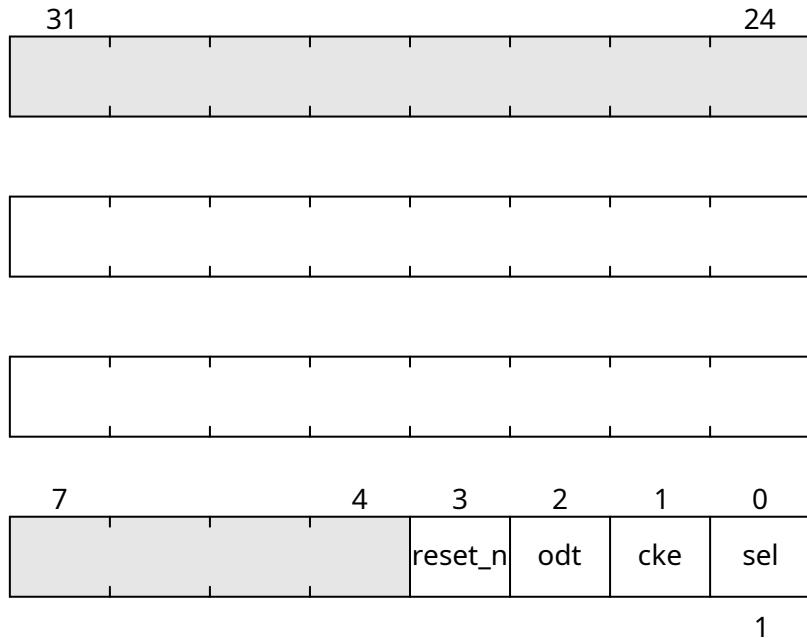


Fig. 21.100: SDRAM DFII CONTROL

Field	Name	Description						
[0]	SEL	<table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0b0</td><td>Software (CPU) control.</td></tr> <tr> <td>0b1</td><td>Hardware control (default).</td></tr> </tbody> </table>	Value	Description	0b0	Software (CPU) control.	0b1	Hardware control (default).
Value	Description							
0b0	Software (CPU) control.							
0b1	Hardware control (default).							
[1]	CKE	DFI clock enable bus						
[2]	ODT	DFI on-die termination bus						
[3]	RESET_N	DFI clock reset bus						

SDRAM_DFII_PIO_COMMAND

Address: $0xf0006800 + 0x4 = 0xf0006804$

Control DFI signals on a single phase

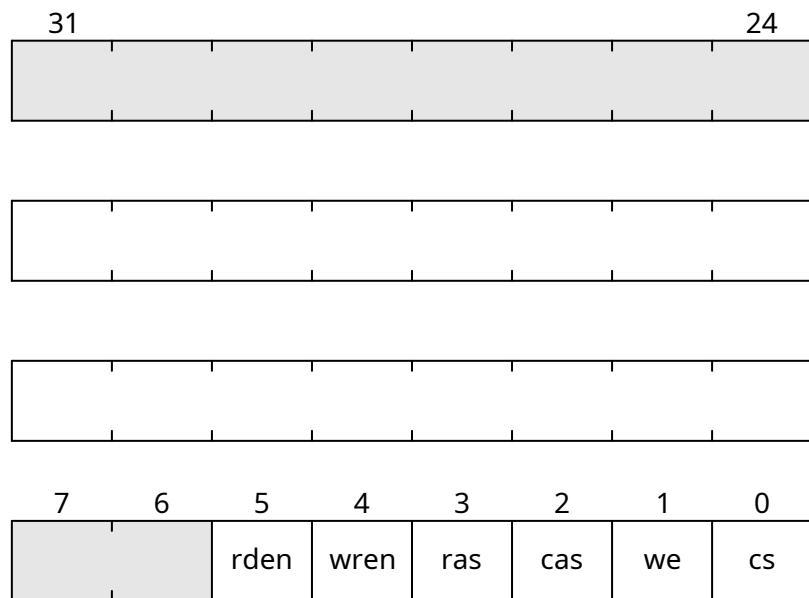


Fig. 21.101: SDRAM_DFII_PIO_COMMAND

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFIIPIO_COMMAND_ISSUE

Address: $0xf0006800 + 0x8 = 0xf0006808$

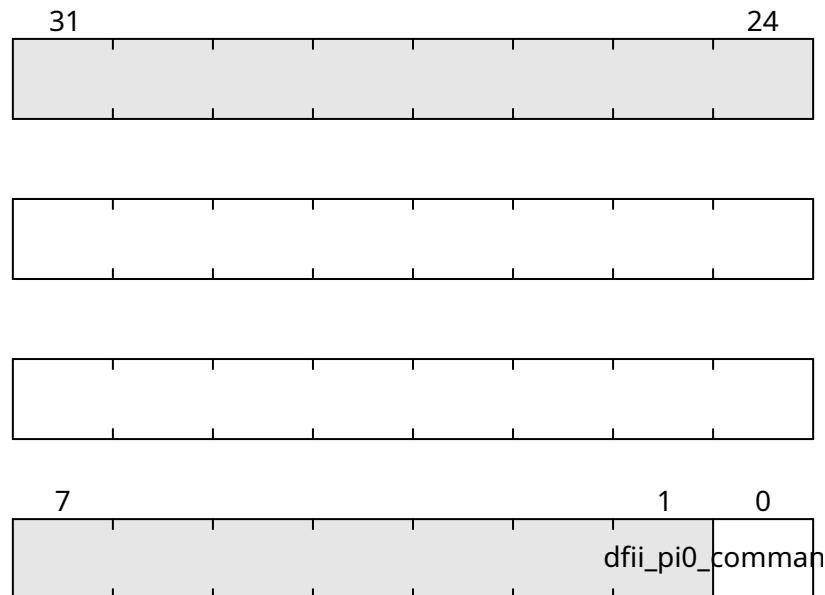


Fig. 21.102: SDRAM_DFIIPIO_COMMAND_ISSUE

SDRAM_DFIIPIO_ADDRESS

Address: $0xf0006800 + 0xc = 0xf000680c$

DFI address bus

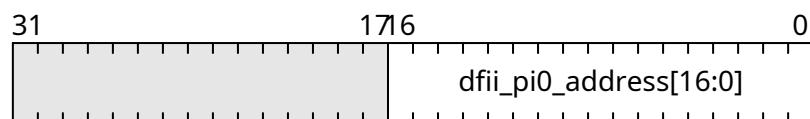


Fig. 21.103: SDRAM_DFIIPIO_ADDRESS

SDRAM_DFIIPIO_BADDRESS

Address: $0xf0006800 + 0x10 = 0xf0006810$

DFI bank address bus

SDRAM_DFIIPIO_WRDATA1

Address: $0xf0006800 + 0x14 = 0xf0006814$

Bits 32-63 of *SDRAM_DFIIPIO_WRDATA*. DFI write data bus

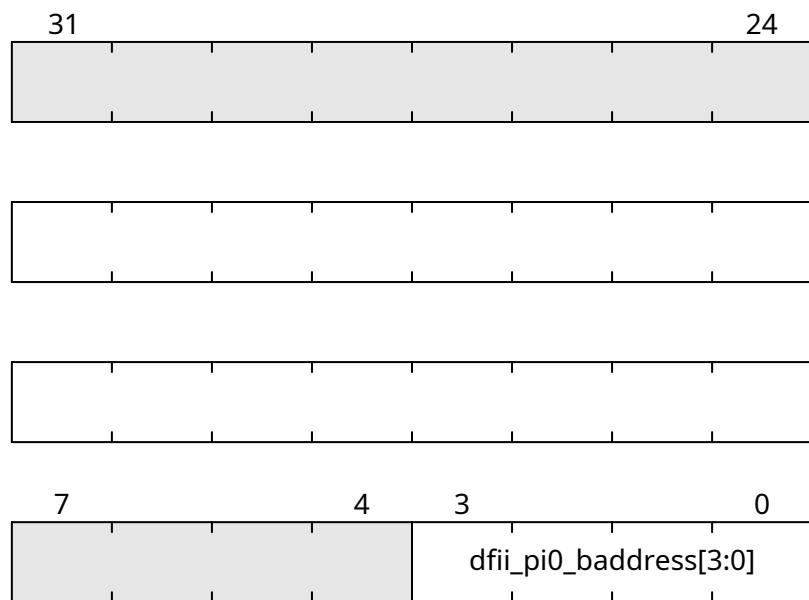


Fig. 21.104: SDRAM_DFII_PI0_BADDRESS

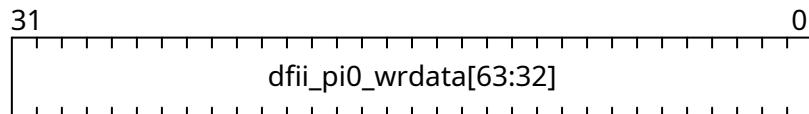


Fig. 21.105: SDRAM_DFII_PI0_WRDATA1

SDRAM_DFII_PIO_WRDATA0

Address: $0xf0006800 + 0x18 = 0xf0006818$

Bits 0-31 of *SDRAM_DFII_PIO_WRDATA*.

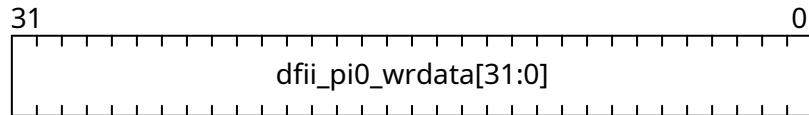


Fig. 21.106: SDRAM_DFII_PIO_WRDATA0

SDRAM_DFII_PIO_RDDATA1

Address: $0xf0006800 + 0x1c = 0xf000681c$

Bits 32-63 of *SDRAM_DFII_PIO_RDDATA*. DFI read data bus

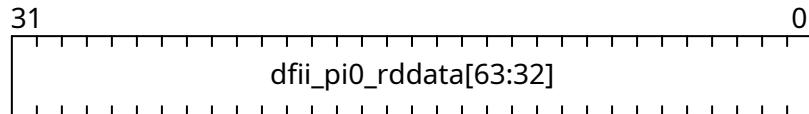


Fig. 21.107: SDRAM_DFII_PIO_RDDATA1

SDRAM_DFII_PIO_RDDATA0

Address: $0xf0006800 + 0x20 = 0xf0006820$

Bits 0-31 of *SDRAM_DFII_PIO_RDDATA*.

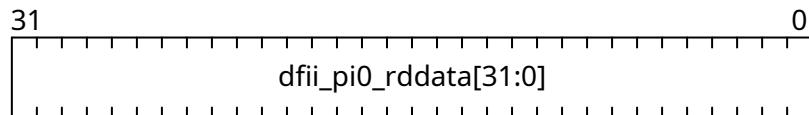


Fig. 21.108: SDRAM_DFII_PIO_RDDATA0

SDRAM_DFII_PI1_COMMAND

Address: $0xf0006800 + 0x24 = 0xf0006824$

Control DFI signals on a single phase

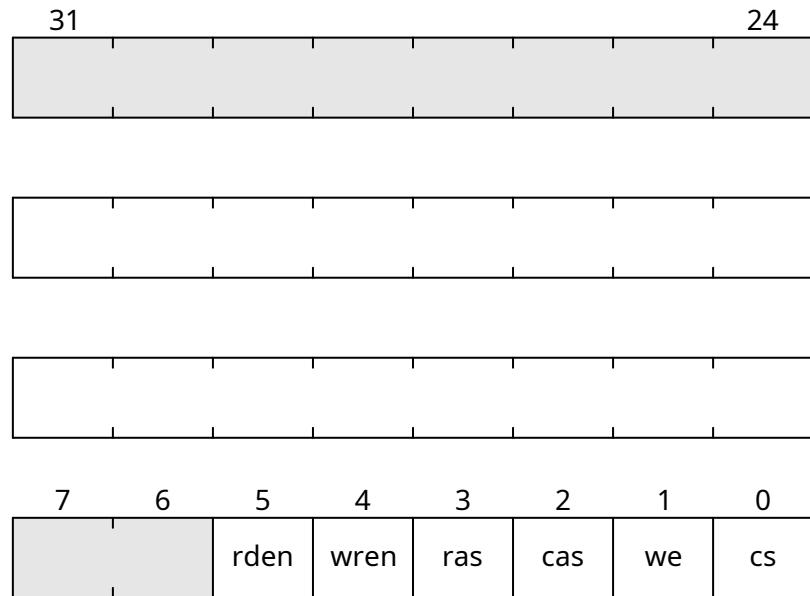


Fig. 21.109: SDRAM_DFII_PI1_COMMAND

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFII_PI1_COMMAND_ISSUE

Address: $0xf0006800 + 0x28 = 0xf0006828$

SDRAM_DFII_PI1_ADDRESS

Address: $0xf0006800 + 0x2c = 0xf000682c$

DFI address bus

SDRAM_DFII_PI1_BADDRESS

Address: $0xf0006800 + 0x30 = 0xf0006830$

DFI bank address bus

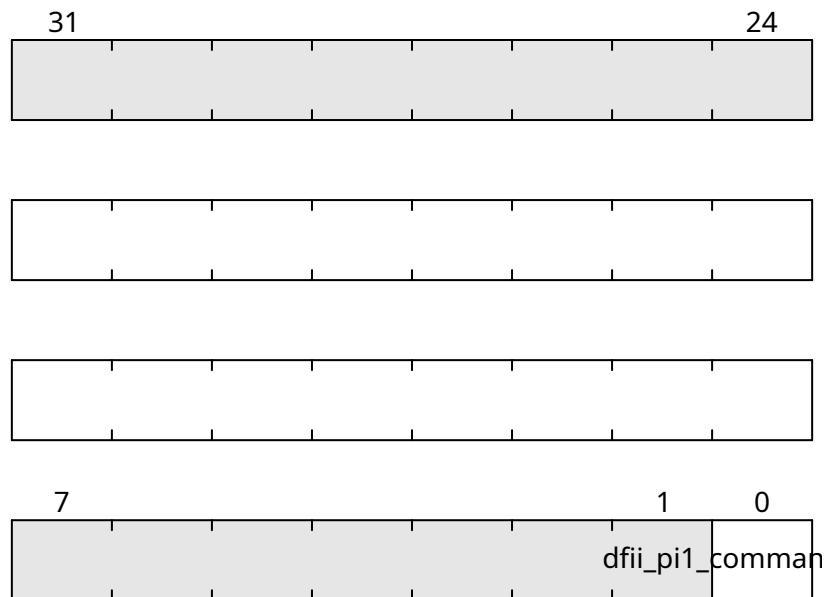


Fig. 21.110: SDRAM_DFII_PI1_COMMAND_ISSUE

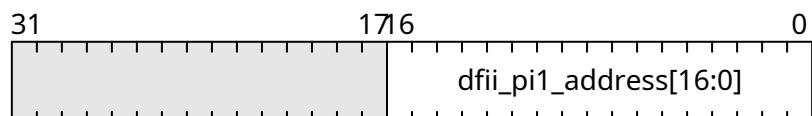


Fig. 21.111: SDRAM_DFII_PI1_ADDRESS

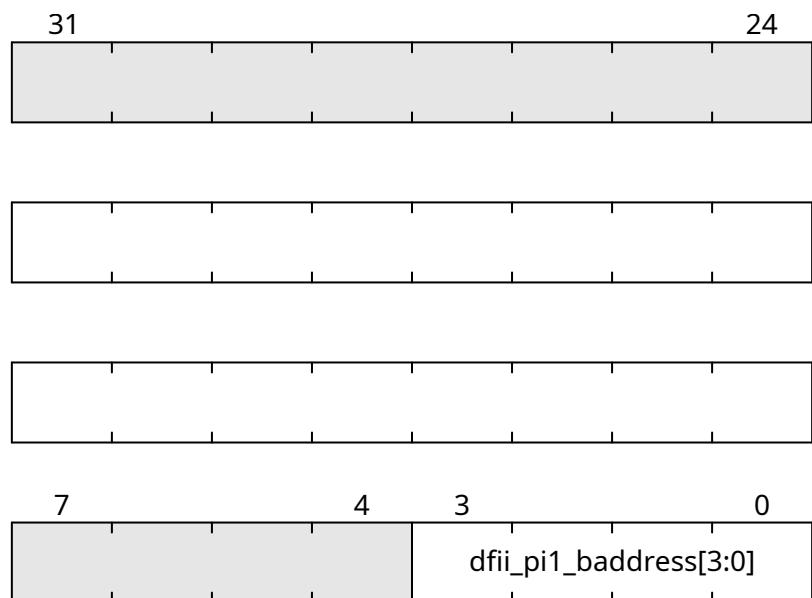


Fig. 21.112: SDRAM_DFII_PI1_BADDRESS

SDRAM_DFII_PI1_WRDATA1

Address: $0xf0006800 + 0x34 = 0xf0006834$

Bits 32-63 of *SDRAM_DFII_PI1_WRDATA*. DFI write data bus

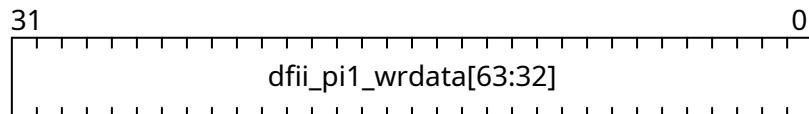


Fig. 21.113: SDRAM_DFII_PI1_WRDATA1

SDRAM_DFII_PI1_WRDATA0

Address: $0xf0006800 + 0x38 = 0xf0006838$

Bits 0-31 of *SDRAM_DFII_PI1_WRDATA*.

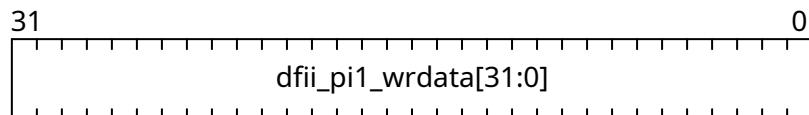


Fig. 21.114: SDRAM_DFII_PI1_WRDATA0

SDRAM_DFII_PI1_RDDATA1

Address: $0xf0006800 + 0x3c = 0xf000683c$

Bits 32-63 of *SDRAM_DFII_PI1_RDDATA*. DFI read data bus

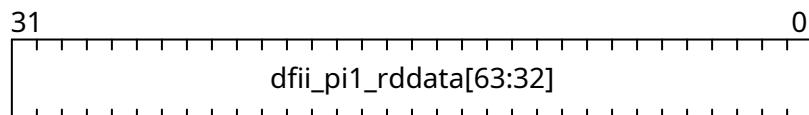


Fig. 21.115: SDRAM_DFII_PI1_RDDATA1

SDRAM_DFII_PI1_RDDATA0

Address: $0xf0006800 + 0x40 = 0xf0006840$

Bits 0-31 of *SDRAM_DFII_PI1_RDDATA*.

SDRAM_DFII_PI2_COMMAND

Address: $0xf0006800 + 0x44 = 0xf0006844$

Control DFI signals on a single phase

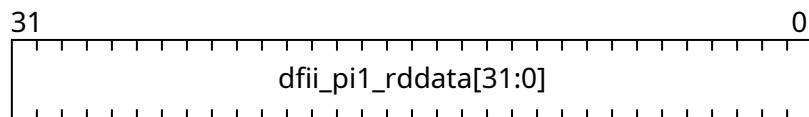


Fig. 21.116: SDRAM_DFII_PI1_RDDATA0

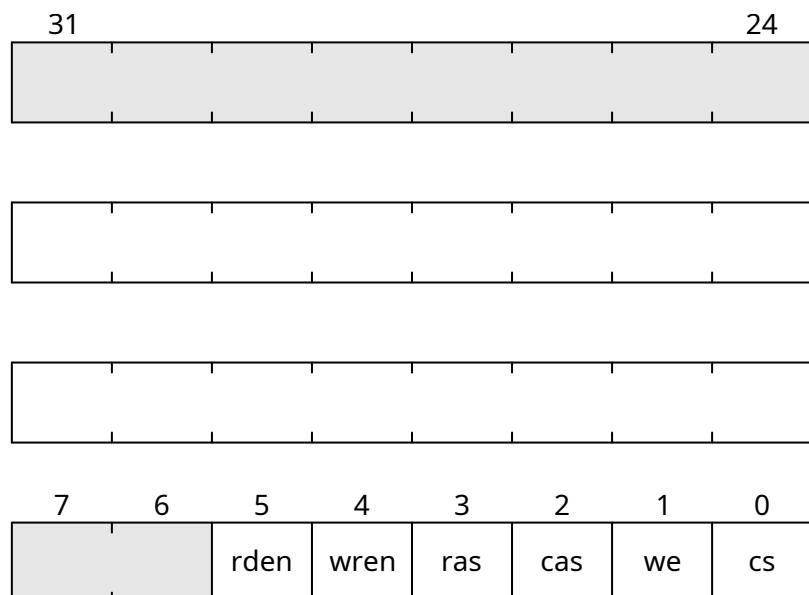


Fig. 21.117: SDRAM_DFII_PI2_COMMAND

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFII_PI2_COMMAND_ISSUE

Address: $0xf0006800 + 0x48 = 0xf0006848$

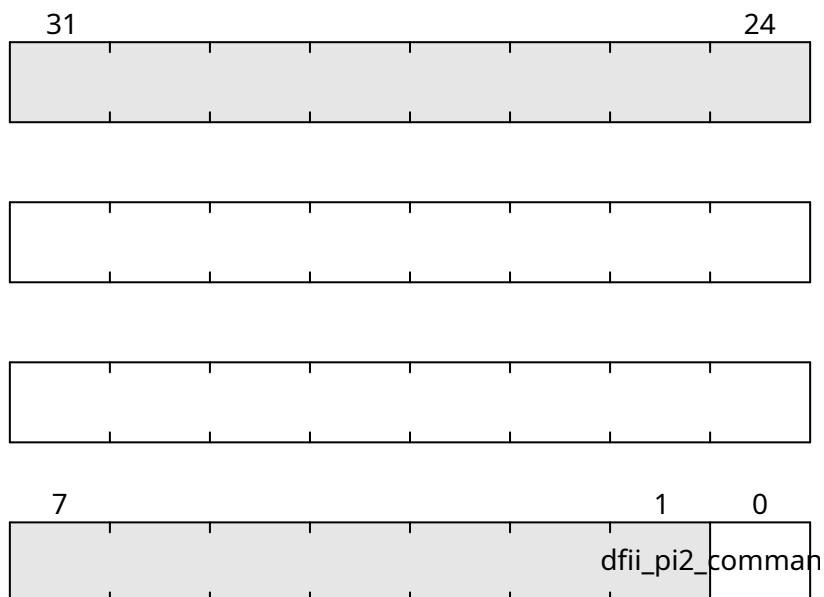


Fig. 21.118: SDRAM_DFII_PI2_COMMAND_ISSUE

SDRAM_DFII_PI2_ADDRESS

Address: $0xf0006800 + 0x4c = 0xf000684c$

DFI address bus

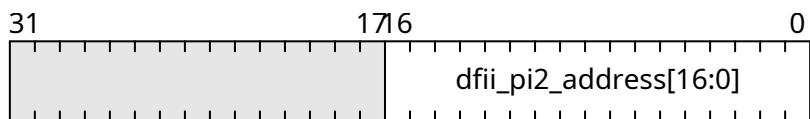


Fig. 21.119: SDRAM_DFII_PI2_ADDRESS

SDRAM_DFII_PI2_BADDRESS

Address: $0xf0006800 + 0x50 = 0xf0006850$

DFI bank address bus

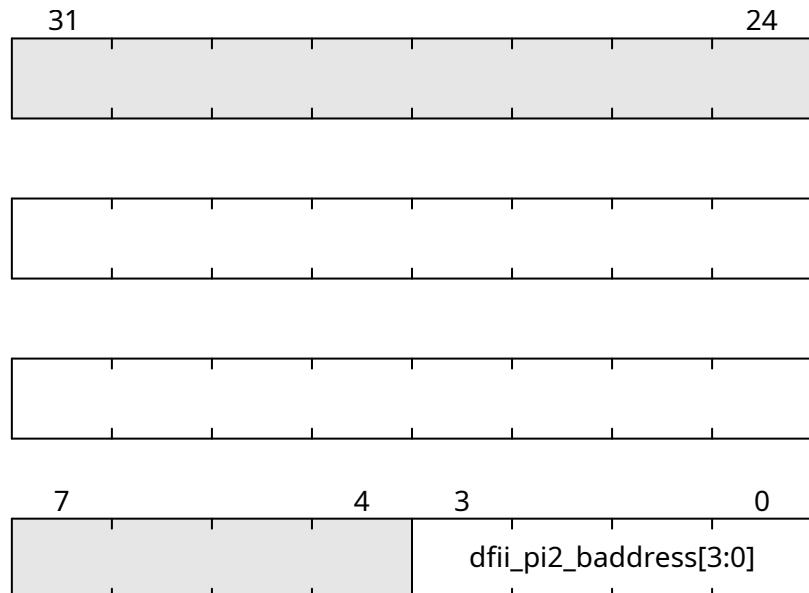


Fig. 21.120: SDRAM_DFII_PI2_BADDRESS

SDRAM_DFII_PI2_WRDATA1

Address: $0xf0006800 + 0x54 = 0xf0006854$

Bits 32-63 of *SDRAM_DFII_PI2_WRDATA*. DFI write data bus

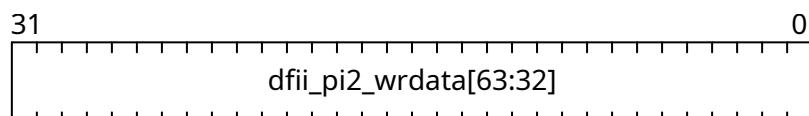


Fig. 21.121: SDRAM_DFII_PI2_WRDATA1

SDRAM_DFII_PI2_WRDATA0

Address: $0xf0006800 + 0x58 = 0xf0006858$

Bits 0-31 of *SDRAM_DFII_PI2_WRDATA*.

SDRAM_DFII_PI2_RDDATA1

Address: $0xf0006800 + 0x5c = 0xf000685c$

Bits 32-63 of *SDRAM_DFII_PI2_RDDATA*. DFI read data bus

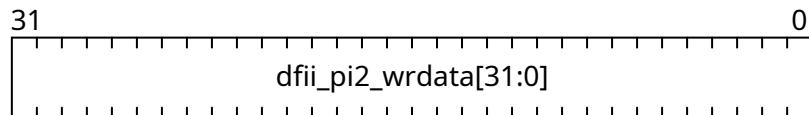


Fig. 21.122: SDRAM_DFII_PI2_WRDATA0

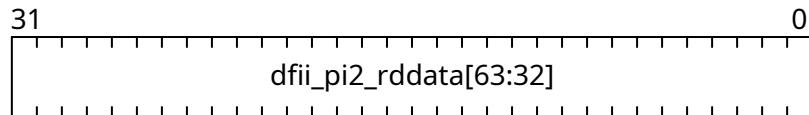


Fig. 21.123: SDRAM_DFII_PI2_RDDATA1

SDRAM_DFII_PI2_RDDATA0

Address: 0xf0006800 + 0x60 = 0xf0006860

Bits 0-31 of *SDRAM_DFII_PI2_RDDATA*.

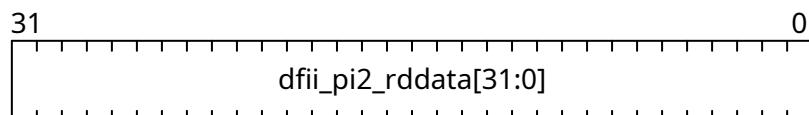


Fig. 21.124: SDRAM_DFII_PI2_RDDATA0

SDRAM_DFII_PI3_COMMAND

Address: 0xf0006800 + 0x64 = 0xf0006864

Control DFI signals on a single phase

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFII_PI3_COMMAND_ISSUE

Address: 0xf0006800 + 0x68 = 0xf0006868

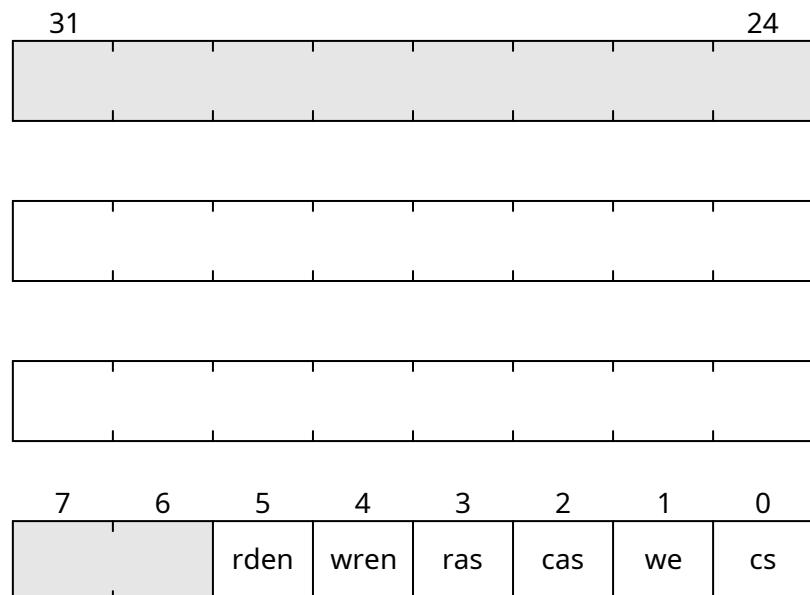


Fig. 21.125: SDRAM_DFII_PI3_COMMAND

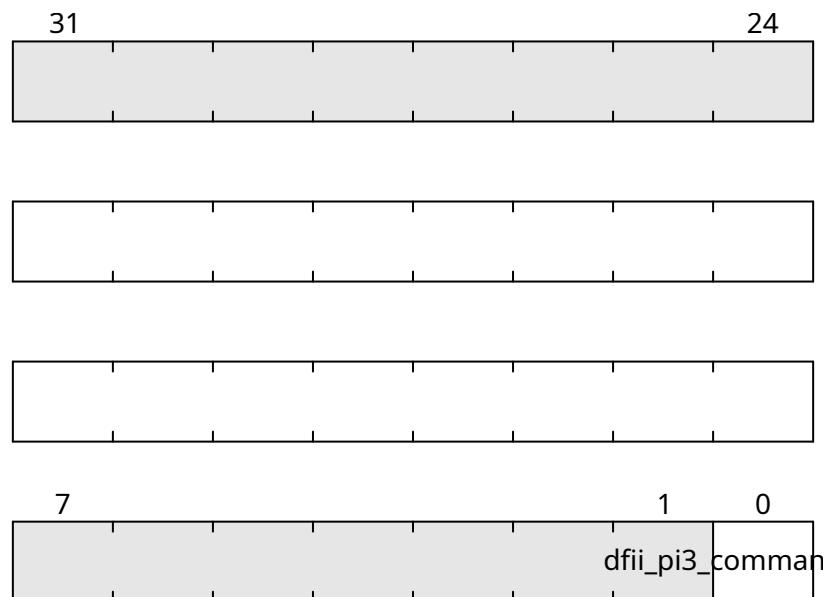


Fig. 21.126: SDRAM_DFII_PI3_COMMAND_ISSUE

SDRAM_DFII_PI3_ADDRESS

Address: $0xf0006800 + 0x6c = 0xf000686c$

DFI address bus

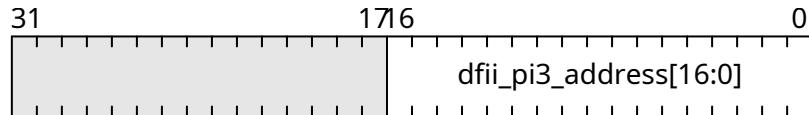


Fig. 21.127: SDRAM_DFII_PI3_ADDRESS

SDRAM_DFII_PI3_BADDRESS

Address: $0xf0006800 + 0x70 = 0xf0006870$

DFI bank address bus



Fig. 21.128: SDRAM_DFII_PI3_BADDRESS

SDRAM_DFII_PI3_WRDATA1

Address: $0xf0006800 + 0x74 = 0xf0006874$

Bits 32-63 of *SDRAM_DFII_PI3_WRDATA*. DFI write data bus

SDRAM_DFII_PI3_WRDATA0

Address: $0xf0006800 + 0x78 = 0xf0006878$

Bits 0-31 of *SDRAM_DFII_PI3_WRDATA*.

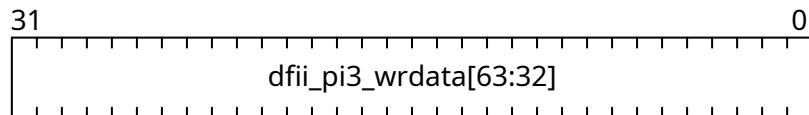


Fig. 21.129: SDRAM_DFII_PI3_WRDATA1

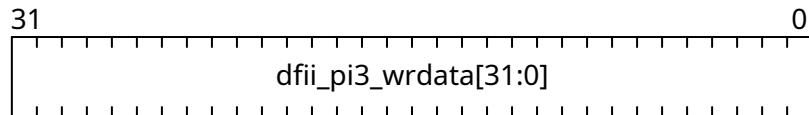


Fig. 21.130: SDRAM_DFII_PI3_WRDATA0

SDRAM_DFII_PI3_RDDATA1

Address: $0xf0006800 + 0x7c = 0xf000687c$

Bits 32-63 of *SDRAM_DFII_PI3_RDDATA*. DFI read data bus

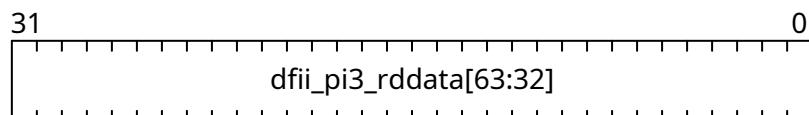


Fig. 21.131: SDRAM_DFII_PI3_RDDATA1

SDRAM_DFII_PI3_RDDATA0

Address: $0xf0006800 + 0x80 = 0xf0006880$

Bits 0-31 of *SDRAM_DFII_PI3_RDDATA*.

SDRAM_CONTROLLER_TRP

Address: $0xf0006800 + 0x84 = 0xf0006884$

SDRAM_CONTROLLER_TRCD

Address: $0xf0006800 + 0x88 = 0xf0006888$

SDRAM_CONTROLLER_TWR

Address: $0xf0006800 + 0x8c = 0xf000688c$

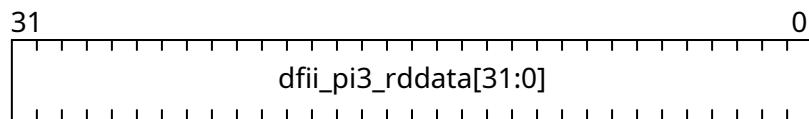


Fig. 21.132: SDRAM_DFII_PI3_RDDATA0

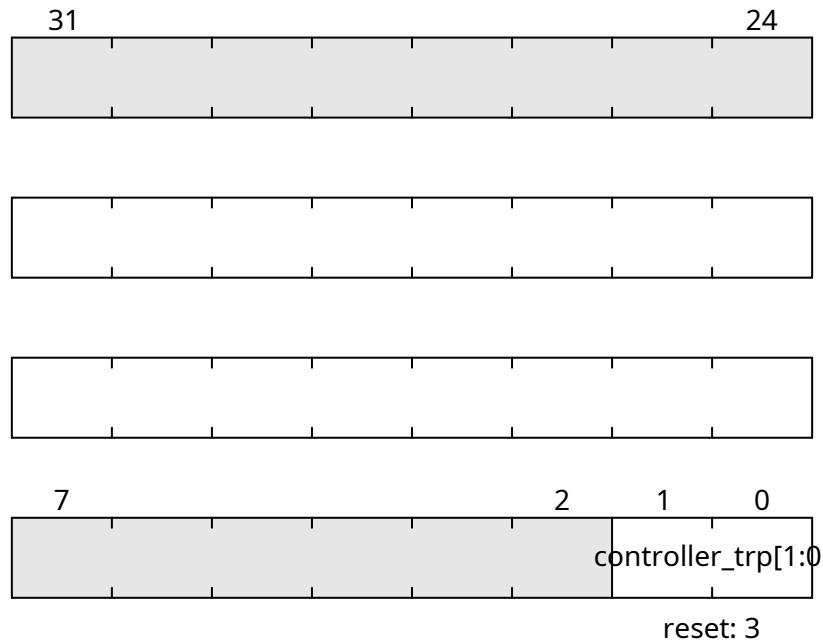


Fig. 21.133: SDRAM_CONTROLLER_TRP

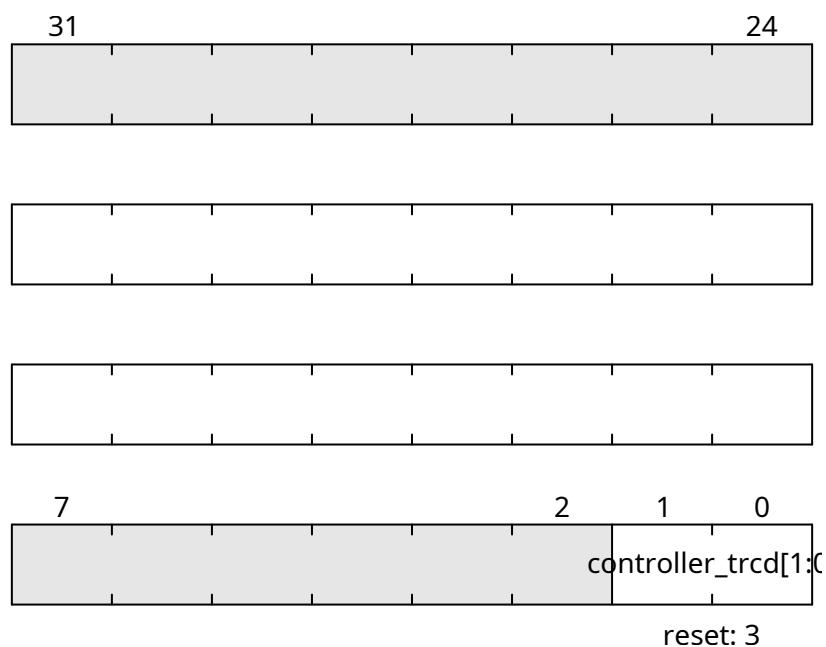


Fig. 21.134: SDRAM_CONTROLLER_TRCD

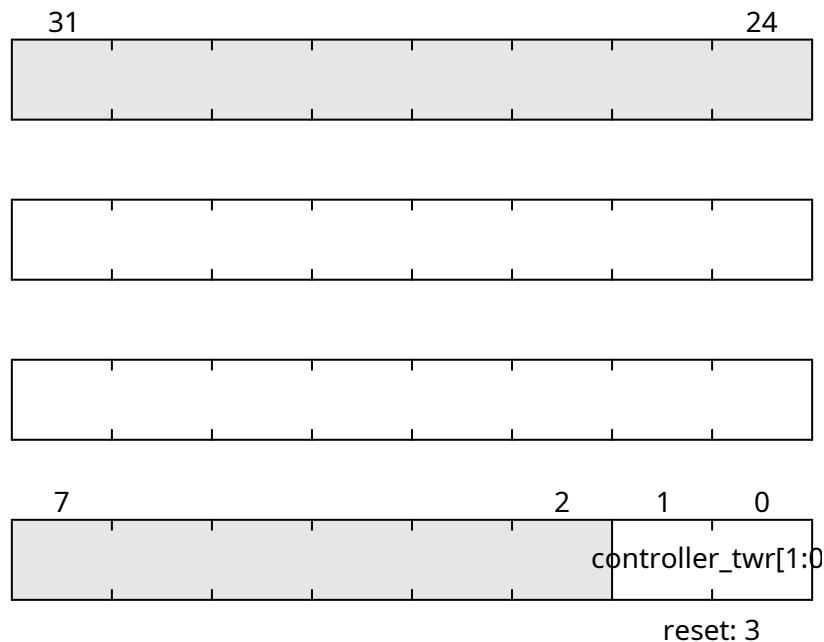


Fig. 21.135: SDRAM_CONTROLLER_TWR

SDRAM_CONTROLLER_TWTR

Address: 0xf0006800 + 0x90 = 0xf0006890

SDRAM_CONTROLLER_TREFI

Address: 0xf0006800 + 0x94 = 0xf0006894

SDRAM_CONTROLLER_TRFC

Address: 0xf0006800 + 0x98 = 0xf0006898

SDRAM_CONTROLLER_TFAW

Address: 0xf0006800 + 0x9c = 0xf000689c

SDRAM_CONTROLLER_TCCD

Address: 0xf0006800 + 0xa0 = 0xf00068a0

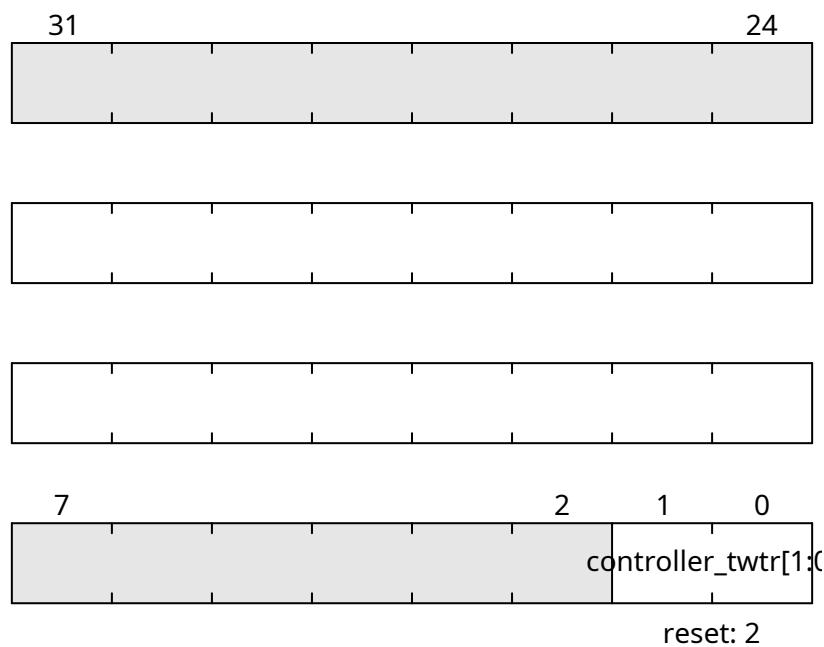


Fig. 21.136: SDRAM_CONTROLLER_TWTR

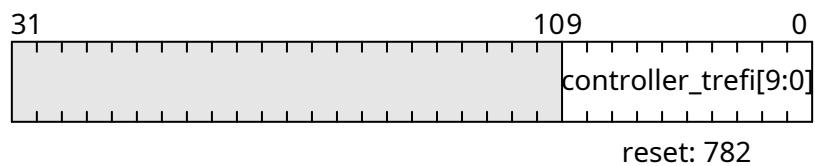


Fig. 21.137: SDRAM_CONTROLLER_TREFI

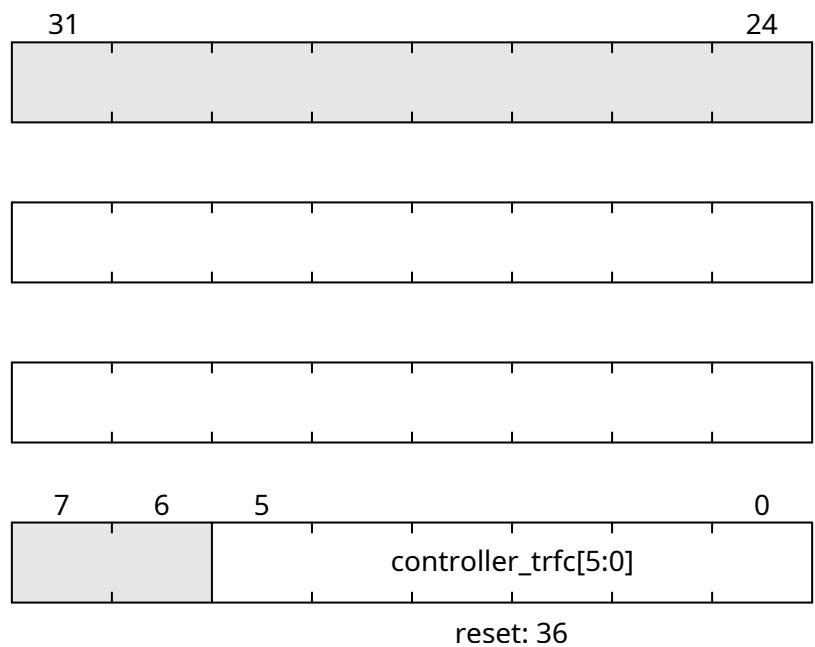


Fig. 21.138: SDRAM_CONTROLLER_TRFC

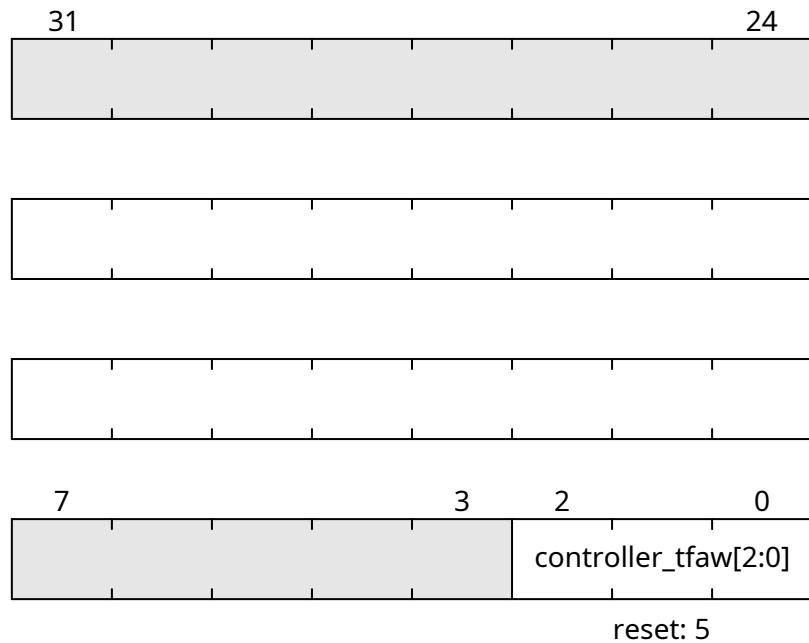


Fig. 21.139: SDRAM_CONTROLLER_TFAW

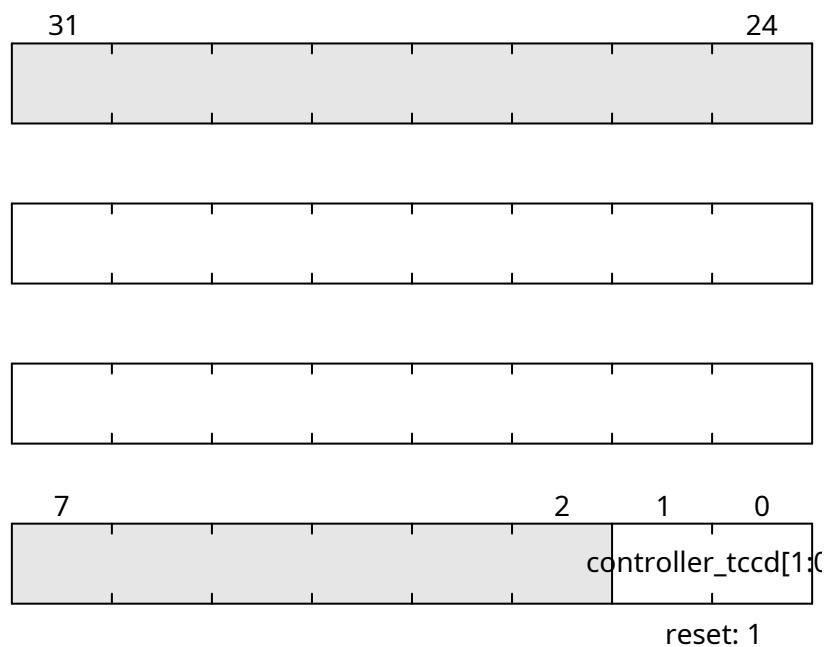


Fig. 21.140: SDRAM_CONTROLLER_TCCD

SDRAM_CONTROLLER_TCCD_WR

Address: $0xf0006800 + 0xa4 = 0xf00068a4$

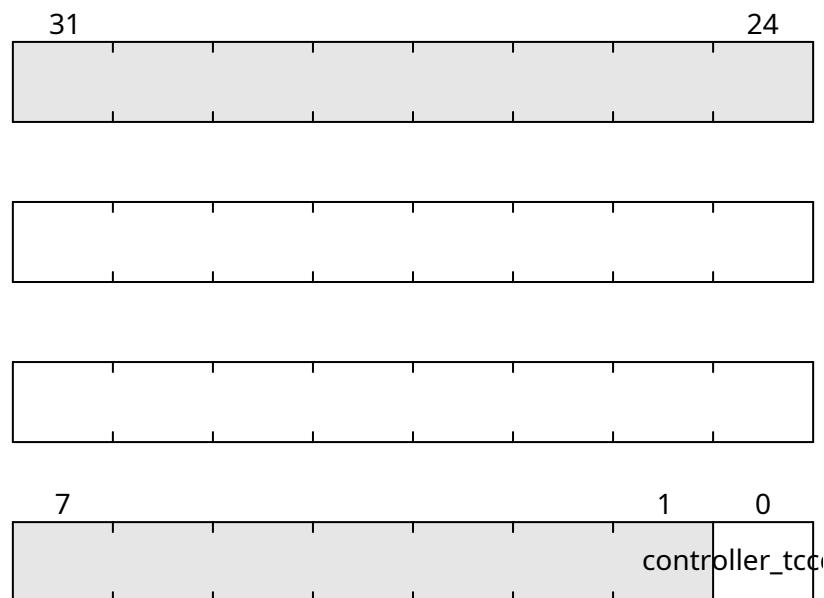


Fig. 21.141: SDRAM_CONTROLLER_TCCD_WR

SDRAM_CONTROLLER_TRTP

Address: $0xf0006800 + 0xa8 = 0xf00068a8$

SDRAM_CONTROLLER_TRRD

Address: $0xf0006800 + 0xac = 0xf00068ac$

SDRAM_CONTROLLER_TRC

Address: $0xf0006800 + 0xb0 = 0xf00068b0$

SDRAM_CONTROLLER_TRAS

Address: $0xf0006800 + 0xb4 = 0xf00068b4$

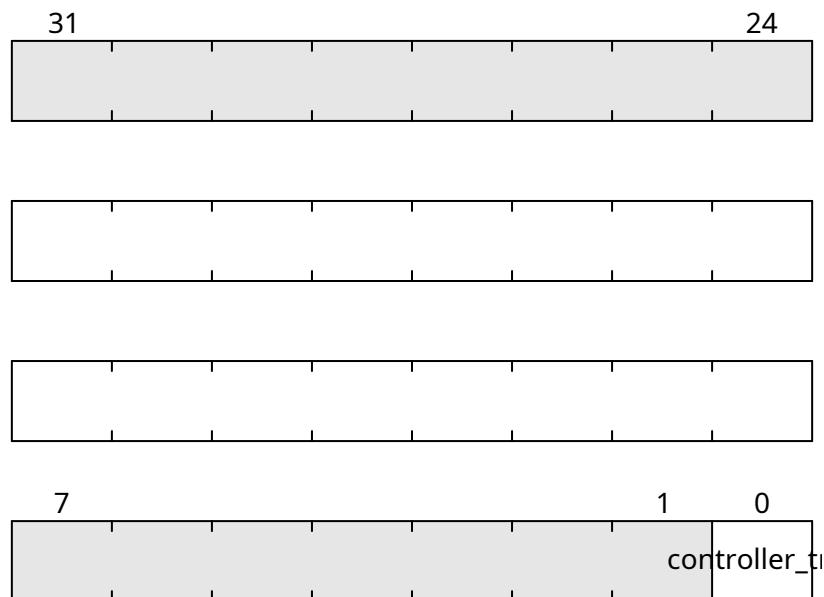


Fig. 21.142: SDRAM_CONTROLLER_TRTP

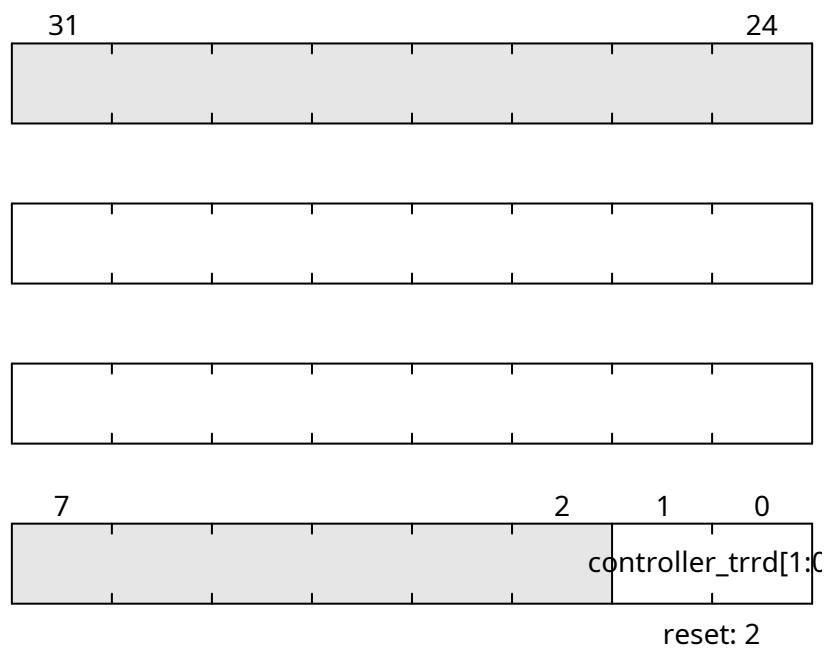


Fig. 21.143: SDRAM_CONTROLLER_TRRD

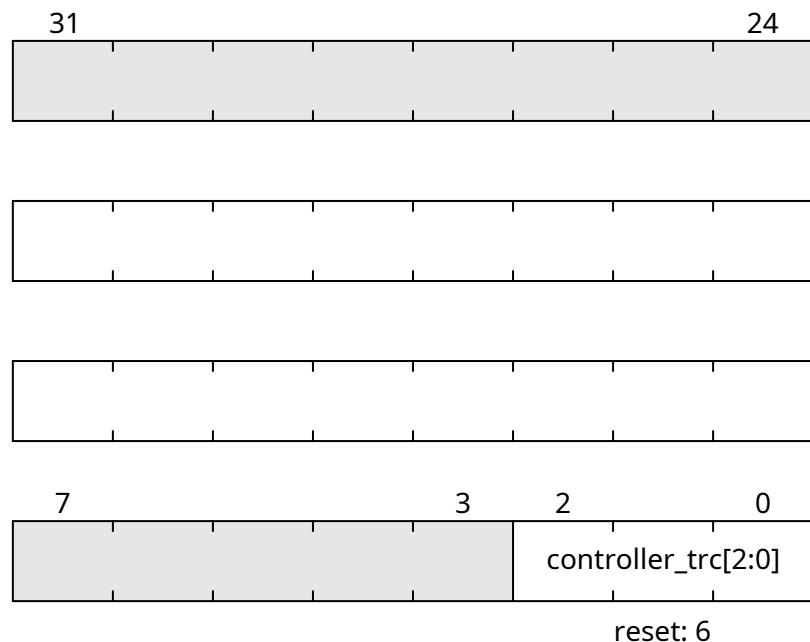


Fig. 21.144: SDRAM_CONTROLLER_TRC

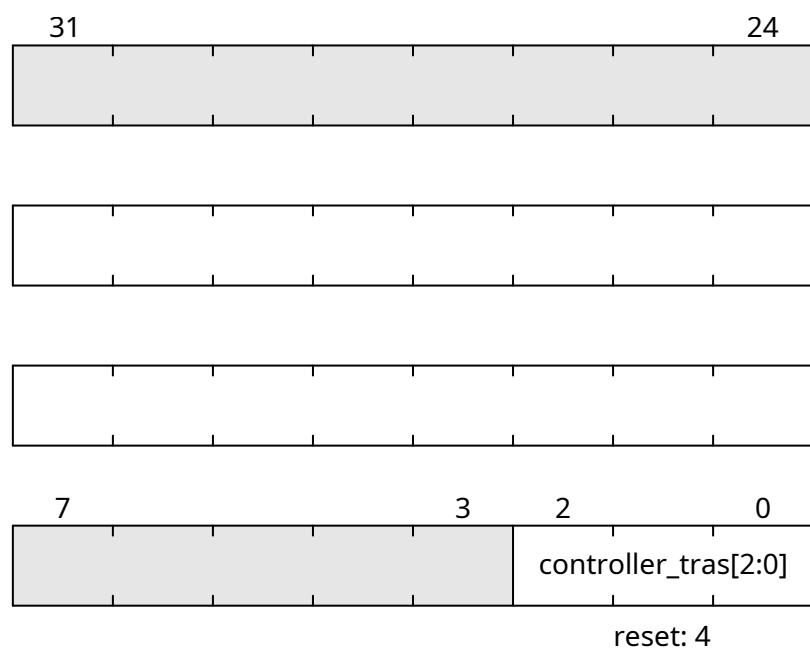


Fig. 21.145: SDRAM_CONTROLLER_TRAS

SDRAM_CONTROLLER_TZQCS

Address: $0xf0006800 + 0xb8 = 0xf00068b8$

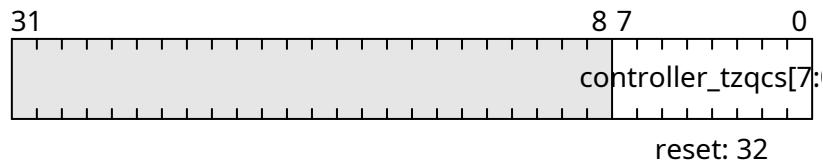


Fig. 21.146: SDRAM_CONTROLLER_TZQCS

SDRAM_CONTROLLER_LAST_ADDR_0

Address: $0xf0006800 + 0xbc = 0xf00068bc$

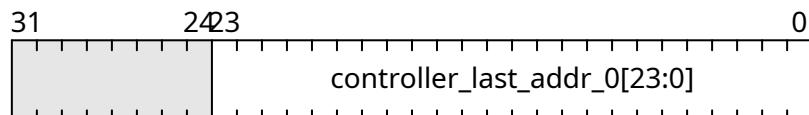


Fig. 21.147: SDRAM_CONTROLLER_LAST_ADDR_0

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0

Address: $0xf0006800 + 0xc0 = 0xf00068c0$

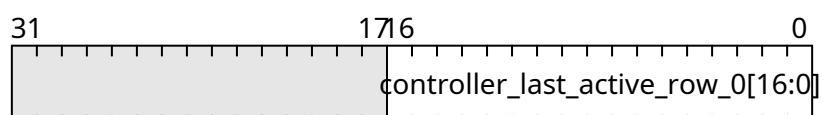


Fig. 21.148: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0

SDRAM_CONTROLLER_LAST_ADDR_1

Address: $0xf0006800 + 0xc4 = 0xf00068c4$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1

Address: $0xf0006800 + 0xc8 = 0xf00068c8$

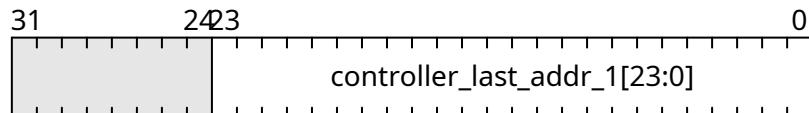


Fig. 21.149: SDRAM_CONTROLLER_LAST_ADDR_1

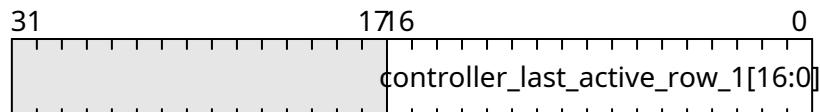


Fig. 21.150: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1

SDRAM_CONTROLLER_LAST_ADDR_2

Address: $0xf0006800 + 0xcc = 0xf00068cc$

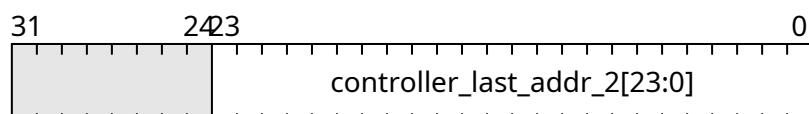


Fig. 21.151: SDRAM_CONTROLLER_LAST_ADDR_2

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2

Address: $0xf0006800 + 0xd0 = 0xf00068d0$

SDRAM_CONTROLLER_LAST_ADDR_3

Address: $0xf0006800 + 0xd4 = 0xf00068d4$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3

Address: $0xf0006800 + 0xd8 = 0xf00068d8$

SDRAM_CONTROLLER_LAST_ADDR_4

Address: $0xf0006800 + 0xdc = 0xf00068dc$

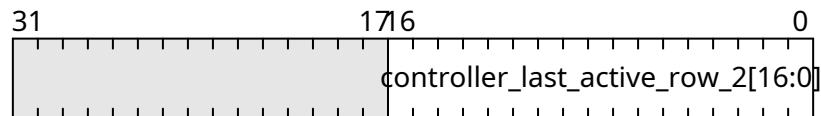


Fig. 21.152: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2

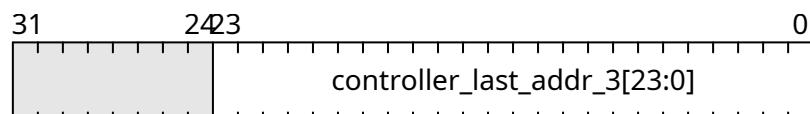


Fig. 21.153: SDRAM_CONTROLLER_LAST_ADDR_3

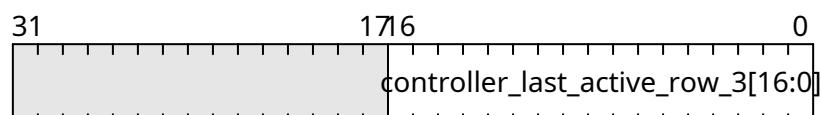


Fig. 21.154: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3

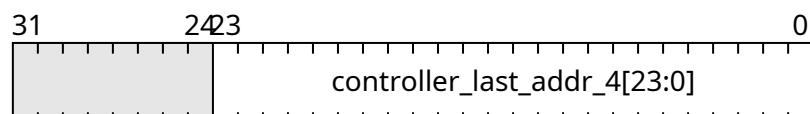


Fig. 21.155: SDRAM_CONTROLLER_LAST_ADDR_4

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4

Address: $0xf0006800 + 0xe0 = 0xf00068e0$

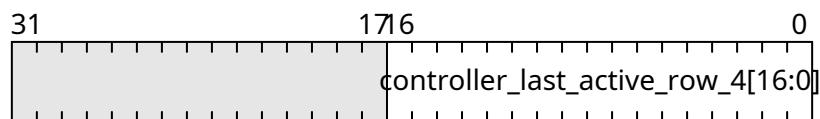


Fig. 21.156: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4

SDRAM_CONTROLLER_LAST_ADDR_5

Address: $0xf0006800 + 0xe4 = 0xf00068e4$

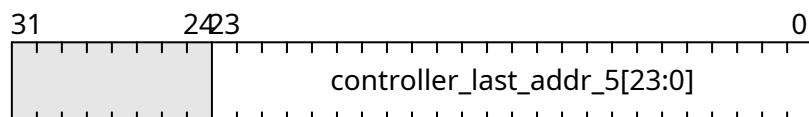


Fig. 21.157: SDRAM_CONTROLLER_LAST_ADDR_5

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5

Address: $0xf0006800 + 0xe8 = 0xf00068e8$

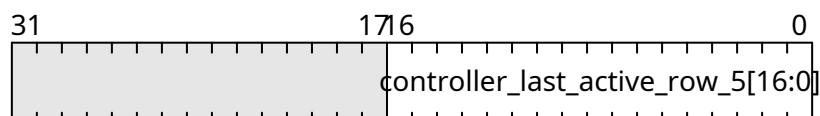


Fig. 21.158: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5

SDRAM_CONTROLLER_LAST_ADDR_6

Address: $0xf0006800 + 0xec = 0xf00068ec$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6

Address: $0xf0006800 + 0xf0 = 0xf00068f0$

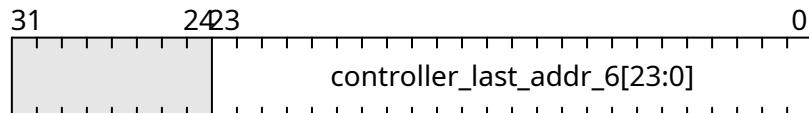


Fig. 21.159: SDRAM_CONTROLLER_LAST_ADDR_6

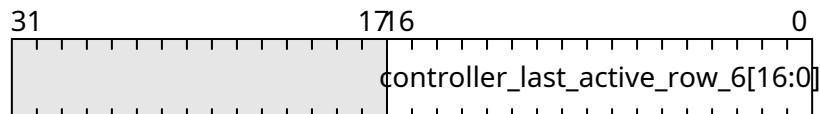


Fig. 21.160: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6

SDRAM_CONTROLLER_LAST_ADDR_7

Address: $0xf0006800 + 0xf4 = 0xf00068f4$

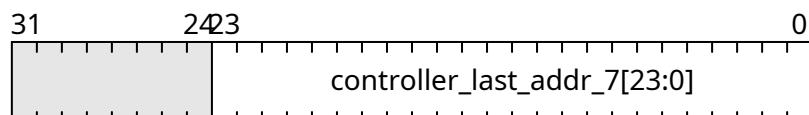


Fig. 21.161: SDRAM_CONTROLLER_LAST_ADDR_7

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7

Address: $0xf0006800 + 0xf8 = 0xf00068f8$

SDRAM_CONTROLLER_LAST_ADDR_8

Address: $0xf0006800 + 0xfc = 0xf00068fc$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_8

Address: $0xf0006800 + 0x100 = 0xf0006900$

SDRAM_CONTROLLER_LAST_ADDR_9

Address: $0xf0006800 + 0x104 = 0xf0006904$

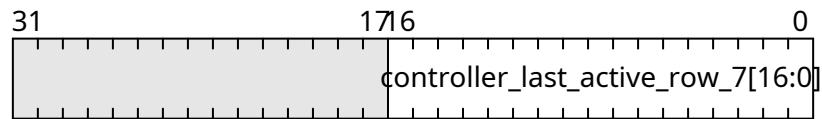


Fig. 21.162: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7

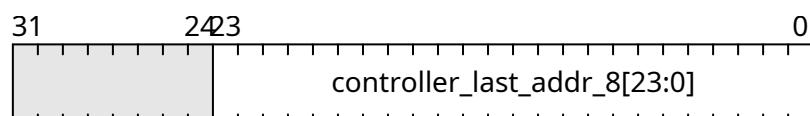


Fig. 21.163: SDRAM_CONTROLLER_LAST_ADDR_8

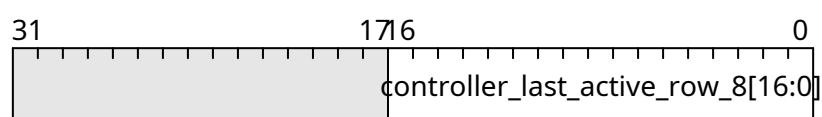


Fig. 21.164: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_8

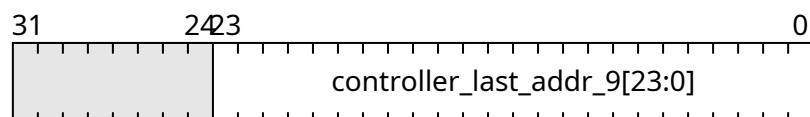


Fig. 21.165: SDRAM_CONTROLLER_LAST_ADDR_9

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_9

Address: $0xf0006800 + 0x108 = 0xf0006908$



Fig. 21.166: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_9

SDRAM_CONTROLLER_LAST_ADDR_10

Address: $0xf0006800 + 0x10c = 0xf000690c$

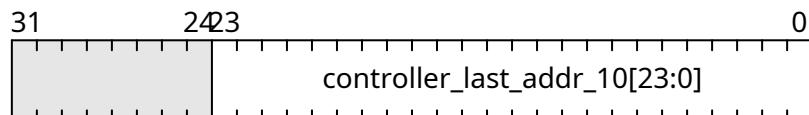


Fig. 21.167: SDRAM_CONTROLLER_LAST_ADDR_10

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_10

Address: $0xf0006800 + 0x110 = 0xf0006910$

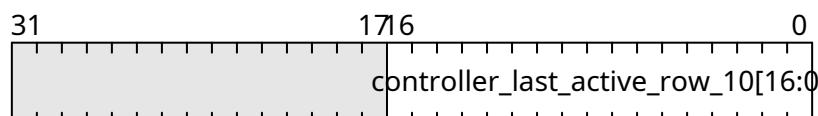


Fig. 21.168: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_10

SDRAM_CONTROLLER_LAST_ADDR_11

Address: $0xf0006800 + 0x114 = 0xf0006914$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_11

Address: $0xf0006800 + 0x118 = 0xf0006918$

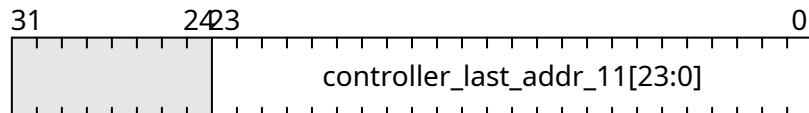


Fig. 21.169: SDRAM_CONTROLLER_LAST_ADDR_11

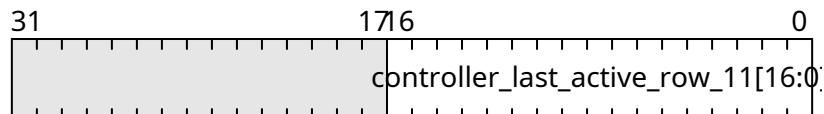


Fig. 21.170: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_11

SDRAM_CONTROLLER_LAST_ADDR_12

Address: $0xf0006800 + 0x11c = 0xf000691c$

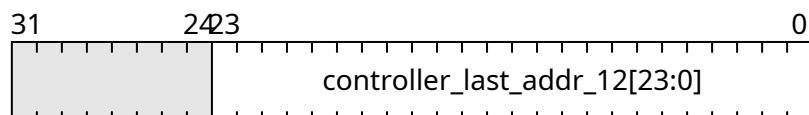


Fig. 21.171: SDRAM_CONTROLLER_LAST_ADDR_12

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_12

Address: $0xf0006800 + 0x120 = 0xf0006920$

SDRAM_CONTROLLER_LAST_ADDR_13

Address: $0xf0006800 + 0x124 = 0xf0006924$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_13

Address: $0xf0006800 + 0x128 = 0xf0006928$

SDRAM_CONTROLLER_LAST_ADDR_14

Address: $0xf0006800 + 0x12c = 0xf000692c$

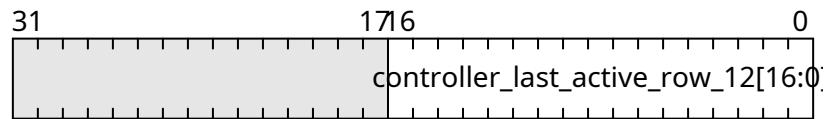


Fig. 21.172: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_12

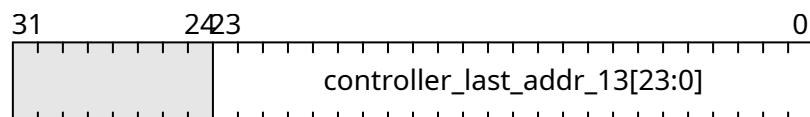


Fig. 21.173: SDRAM_CONTROLLER_LAST_ADDR_13

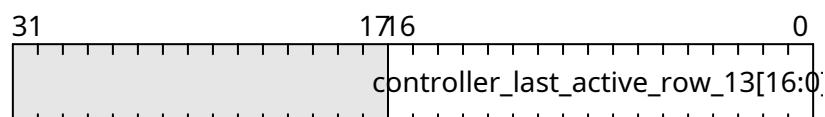


Fig. 21.174: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_13

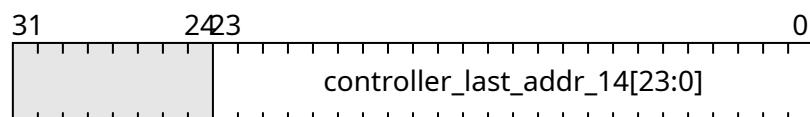


Fig. 21.175: SDRAM_CONTROLLER_LAST_ADDR_14

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_14

Address: $0xf0006800 + 0x130 = 0xf0006930$

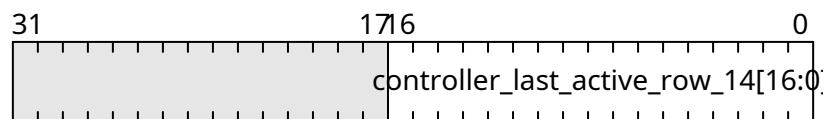


Fig. 21.176: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_14

SDRAM_CONTROLLER_LAST_ADDR_15

Address: $0xf0006800 + 0x134 = 0xf0006934$

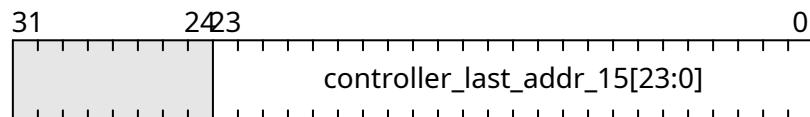


Fig. 21.177: SDRAM_CONTROLLER_LAST_ADDR_15

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_15

Address: $0xf0006800 + 0x138 = 0xf0006938$

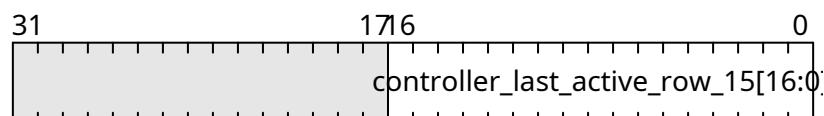


Fig. 21.178: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_15

21.2.15 SDRAM_CHECKER

Register Listing for SDRAM_CHECKER

Register	Address
<i>SDRAM_CHECKER_RESET</i>	0xf0007000
<i>SDRAM_CHECKER_START</i>	0xf0007004
<i>SDRAM_CHECKER_DONE</i>	0xf0007008
<i>SDRAM_CHECKER_BASE1</i>	0xf000700c
<i>SDRAM_CHECKER_BASE0</i>	0xf0007010
<i>SDRAM_CHECKER_END1</i>	0xf0007014
<i>SDRAM_CHECKER_END0</i>	0xf0007018
<i>SDRAM_CHECKER_LENGTH1</i>	0xf000701c
<i>SDRAM_CHECKER_LENGTH0</i>	0xf0007020
<i>SDRAM_CHECKER_RANDOM</i>	0xf0007024
<i>SDRAM_CHECKER_TICKS</i>	0xf0007028
<i>SDRAM_CHECKER_ERRORS</i>	0xf000702c

SDRAM_CHECKER_RESET

Address: $0xf0007000 + 0x0 = 0xf0007000$



Fig. 21.179: SDRAM_CHECKER_RESET

SDRAM_CHECKER_START

Address: $0xf0007000 + 0x4 = 0xf0007004$



Fig. 21.180: SDRAM_CHECKER_START

SDRAM_CHECKER_DONE

Address: $0xf0007000 + 0x8 = 0xf0007008$

SDRAM_CHECKER_BASE1

Address: $0xf0007000 + 0xc = 0xf000700c$

Bits 32-32 of *SDRAM_CHECKER_BASE*.

SDRAM_CHECKER_BASE0

Address: $0xf0007000 + 0x10 = 0xf0007010$

Bits 0-31 of *SDRAM_CHECKER_BASE*.

SDRAM_CHECKER_END1

Address: $0xf0007000 + 0x14 = 0xf0007014$

Bits 32-32 of *SDRAM_CHECKER_END*.

SDRAM_CHECKER_END0

Address: $0xf0007000 + 0x18 = 0xf0007018$

Bits 0-31 of *SDRAM_CHECKER_END*.

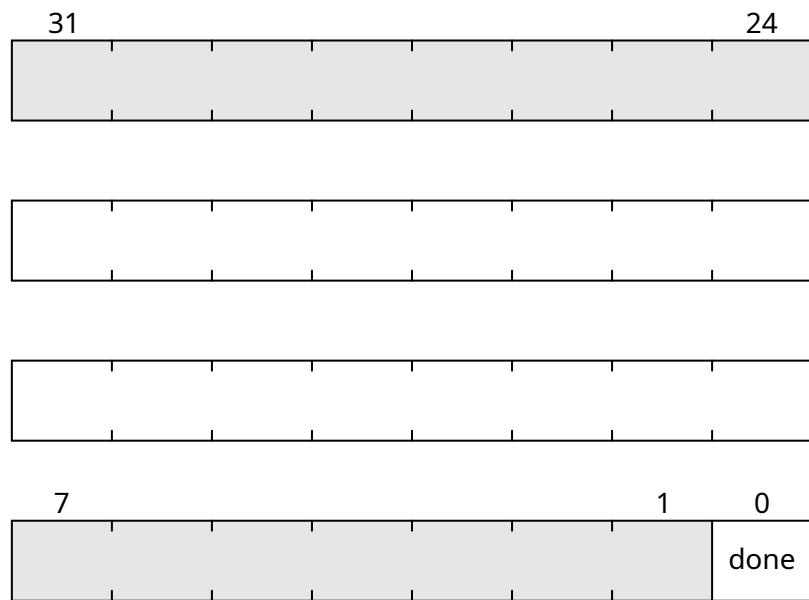


Fig. 21.181: SDRAM_CHECKER_DONE

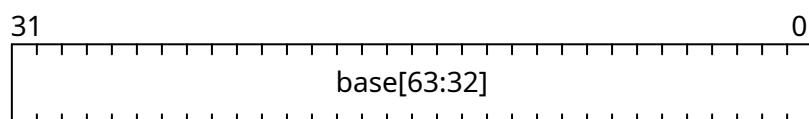


Fig. 21.182: SDRAM_CHECKER_BASE1

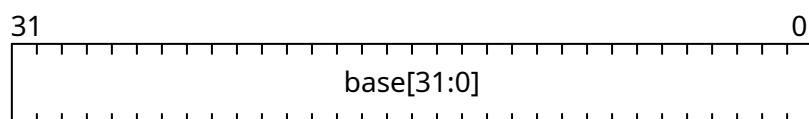


Fig. 21.183: SDRAM_CHECKER_BASE0

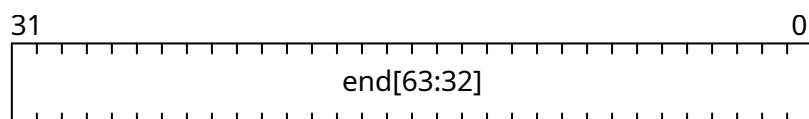


Fig. 21.184: SDRAM_CHECKER_END1

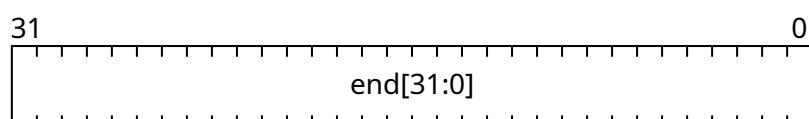


Fig. 21.185: SDRAM_CHECKER_END0

SDRAM_CHECKER_LENGTH1

Address: $0xf0007000 + 0x1c = 0xf000701c$

Bits 32-32 of *SDRAM_CHECKER_LENGTH*.

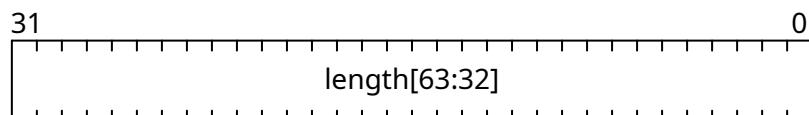


Fig. 21.186: SDRAM_CHECKER_LENGTH1

SDRAM_CHECKER_LENGTH0

Address: $0xf0007000 + 0x20 = 0xf0007020$

Bits 0-31 of *SDRAM_CHECKER_LENGTH*.

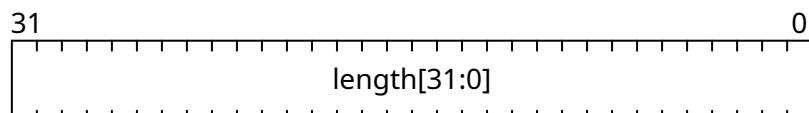


Fig. 21.187: SDRAM_CHECKER_LENGTH0

SDRAM_CHECKER_RANDOM

Address: $0xf0007000 + 0x24 = 0xf0007024$

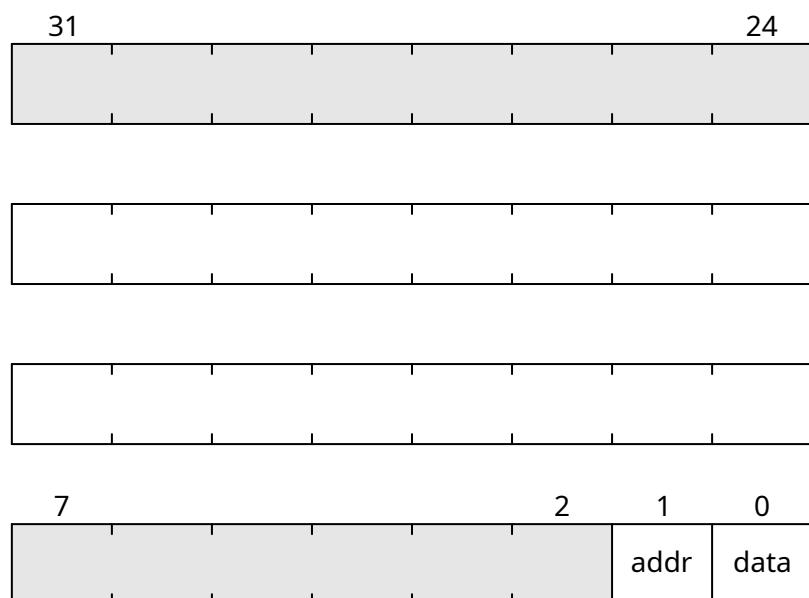


Fig. 21.188: SDRAM_CHECKER_RANDOM

Field	Name	Description

SDRAM_CHECKER_TICKS

Address: $0xf0007000 + 0x28 = 0xf0007028$

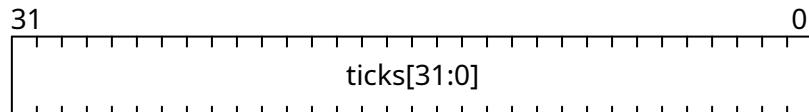


Fig. 21.189: SDRAM_CHECKER_TICKS

SDRAM_CHECKER_ERRORS

Address: $0xf0007000 + 0x2c = 0xf000702c$

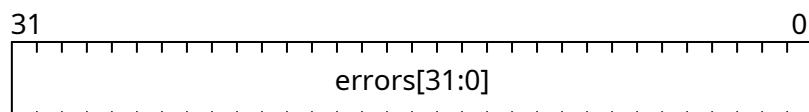


Fig. 21.190: SDRAM_CHECKER_ERRORS

21.2.16 SDRAM_GENERATOR

Register Listing for SDRAM_GENERATOR

Register	Address
<i>SDRAM_GENERATOR_RESET</i>	<i>0xf0007800</i>
<i>SDRAM_GENERATOR_START</i>	<i>0xf0007804</i>
<i>SDRAM_GENERATOR_DONE</i>	<i>0xf0007808</i>
<i>SDRAM_GENERATOR_BASE1</i>	<i>0xf000780c</i>
<i>SDRAM_GENERATOR_BASE0</i>	<i>0xf0007810</i>
<i>SDRAM_GENERATOR_END1</i>	<i>0xf0007814</i>
<i>SDRAM_GENERATOR_END0</i>	<i>0xf0007818</i>
<i>SDRAM_GENERATOR_LENGTH1</i>	<i>0xf000781c</i>
<i>SDRAM_GENERATOR_LENGTH0</i>	<i>0xf0007820</i>
<i>SDRAM_GENERATOR_RANDOM</i>	<i>0xf0007824</i>
<i>SDRAM_GENERATOR_TICKS</i>	<i>0xf0007828</i>

SDRAM_GENERATOR_RESET

Address: $0xf0007800 + 0x0 = 0xf0007800$

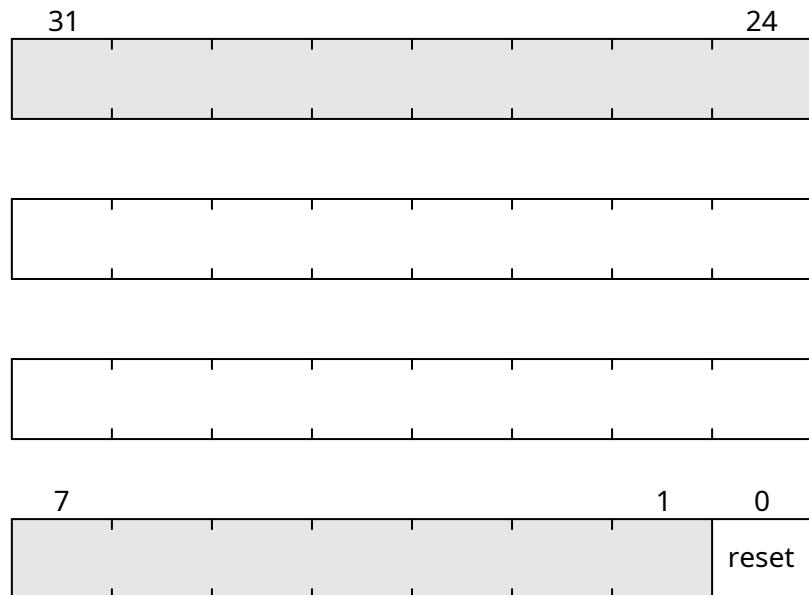


Fig. 21.191: SDRAM_GENERATOR_RESET

SDRAM_GENERATOR_START

Address: $0xf0007800 + 0x4 = 0xf0007804$

SDRAM_GENERATOR_DONE

Address: $0xf0007800 + 0x8 = 0xf0007808$

SDRAM_GENERATOR_BASE1

Address: $0xf0007800 + 0xc = 0xf000780c$

Bits 32-32 of *SDRAM_GENERATOR_BASE*.

SDRAM_GENERATOR_BASE0

Address: $0xf0007800 + 0x10 = 0xf0007810$

Bits 0-31 of *SDRAM_GENERATOR_BASE*.



Fig. 21.192: SDRAM_GENERATOR_START



Fig. 21.193: SDRAM_GENERATOR_DONE

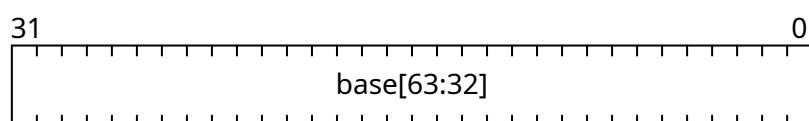


Fig. 21.194: SDRAM_GENERATOR_BASE1

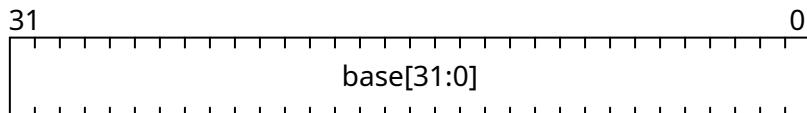


Fig. 21.195: SDRAM_GENERATOR_BASE0

SDRAM_GENERATOR_END1

Address: $0xf0007800 + 0x14 = 0xf0007814$

Bits 32-32 of *SDRAM_GENERATOR-END*.

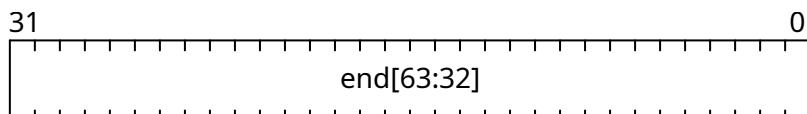


Fig. 21.196: SDRAM_GENERATOR_END1

SDRAM_GENERATOR_END0

Address: $0xf0007800 + 0x18 = 0xf0007818$

Bits 0-31 of *SDRAM_GENERATOR-END*.

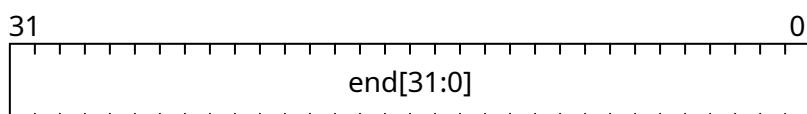


Fig. 21.197: SDRAM_GENERATOR_END0

SDRAM_GENERATOR_LENGTH1

Address: $0xf0007800 + 0x1c = 0xf000781c$

Bits 32-32 of *SDRAM_GENERATOR_LENGTH*.

SDRAM_GENERATOR_LENGTH0

Address: $0xf0007800 + 0x20 = 0xf0007820$

Bits 0-31 of *SDRAM_GENERATOR_LENGTH*.

SDRAM_GENERATOR_RANDOM

Address: $0xf0007800 + 0x24 = 0xf0007824$

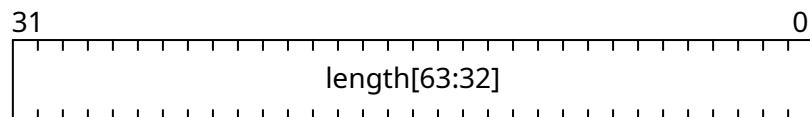


Fig. 21.198: SDRAM_GENERATOR_LENGTH1

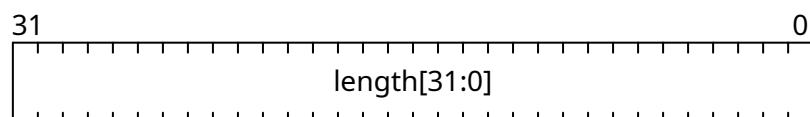


Fig. 21.199: SDRAM_GENERATOR_LENGTH0



Fig. 21.200: SDRAM_GENERATOR_RANDOM

Field	Name	Description

SDRAM_GENERATOR_TICKS

Address: $0xf0007800 + 0x28 = 0xf0007828$

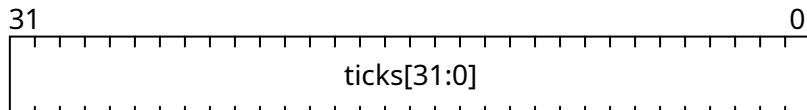


Fig. 21.201: SDRAM_GENERATOR_TICKS

21.2.17 TIMERO

Timer

Provides a generic Timer core.

The Timer is implemented as a countdown timer that can be used in various modes:

- Polling : Returns current countdown value to software
- One-Shot: Loads itself and stops when value reaches 0
- Periodic: (Re-)Loads itself when value reaches 0

en register allows the user to enable/disable the Timer. When the Timer is enabled, it is automatically loaded with the value of *load* register.

When the Timer reaches 0, it is automatically reloaded with value of *reload* register.

The user can latch the current countdown value by writing to *update_value* register, it will update *value* register with current countdown value.

To use the Timer in One-Shot mode, the user needs to:

- Disable the timer
- Set the load register to the expected duration
- (Re-)Enable the Timer

To use the Timer in Periodic mode, the user needs to:

- Disable the Timer
- Set the load register to 0
- Set the reload register to the expected period
- Enable the Timer

For both modes, the CPU can be advertised by an IRQ that the duration/period has elapsed. (The CPU can also do software polling with `update_value` and `value` to know the elapsed duration)

Register Listing for TIMERO

Register	Address
<code>TIMERO_LOAD</code>	<code>0xf0008000</code>
<code>TIMERO_RELOAD</code>	<code>0xf0008004</code>
<code>TIMERO_EN</code>	<code>0xf0008008</code>
<code>TIMERO_UPDATE_VALUE</code>	<code>0xf000800c</code>
<code>TIMERO_VALUE</code>	<code>0xf0008010</code>
<code>TIMERO_EV_STATUS</code>	<code>0xf0008014</code>
<code>TIMERO_EV_PENDING</code>	<code>0xf0008018</code>
<code>TIMERO_EV_ENABLE</code>	<code>0xf000801c</code>

TIMERO_LOAD

Address: $0xf0008000 + 0x0 = 0xf0008000$

Load value when Timer is (re-)enabled. In One-Shot mode, the value written to this register specifies the Timer's duration in clock cycles.

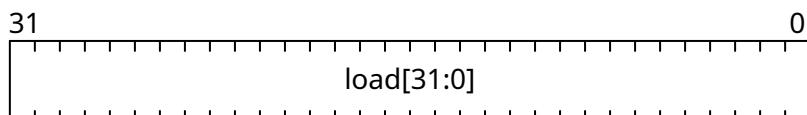


Fig. 21.202: TIMERO_LOAD

TIMERO_RELOAD

Address: $0xf0008000 + 0x4 = 0xf0008004$

Reload value when Timer reaches 0. In Periodic mode, the value written to this register specify the Timer's period in clock cycles.

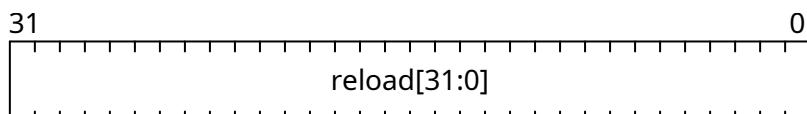


Fig. 21.203: TIMERO_RELOAD

TIMERO_EN

Address: $0xf0008000 + 0x8 = 0xf0008008$

Enable flag of the Timer. Set this flag to 1 to enable/start the Timer. Set to 0 to disable the Timer.

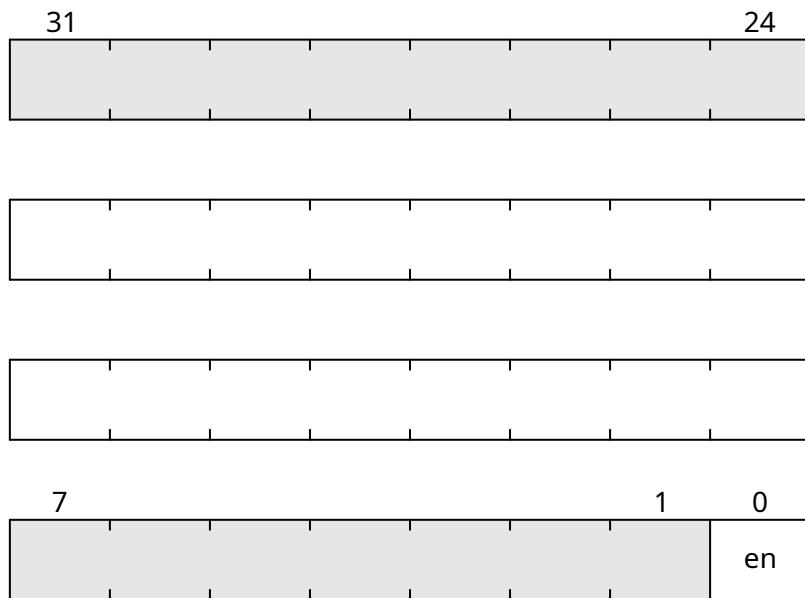


Fig. 21.204: TIMERO_EN

TIMERO_UPDATE_VALUE

Address: $0xf0008000 + 0xc = 0xf000800c$

Update trigger for the current countdown value. A write to this register latches the current countdown value to value register.

TIMERO_VALUE

Address: $0xf0008000 + 0x10 = 0xf0008010$

Latched countdown value. This value is updated by writing to update_value.

TIMERO_EV_STATUS

Address: $0xf0008000 + 0x14 = 0xf0008014$

This register contains the current raw level of the zero event trigger. Writes to this register have no effect.

Field	Name	Description
[0]	ZERO	Level of the zero event

TIMERO_EV_PENDING

Address: $0xf0008000 + 0x18 = 0xf0008018$

When a zero event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.



Fig. 21.205: TIMERO_UPDATE_VALUE

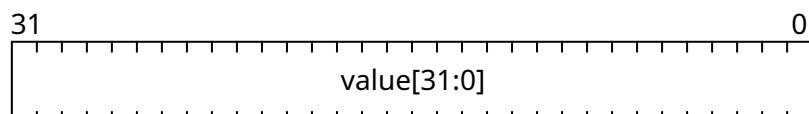


Fig. 21.206: TIMERO_VALUE



Fig. 21.207: TIMERO_EV_STATUS



Fig. 21.208: TIMERO_EV_PENDING

Field	Name	Description
[0]	ZERO	1 if a zero event occurred. This Event is triggered on a falling edge.

TIMERO_EV_ENABLE

Address: $0xf0008000 + 0x1c = 0xf000801c$

This register enables the corresponding zero events. Write a 0 to this register to disable individual events.

Field	Name	Description
[0]	ZERO	Write a 1 to enable the zero Event

21.2.18 UART



Fig. 21.209: TIMERO_EV_ENABLE

Register Listing for UART

Register	Address
UART_RXTX	0xf0008800
UART_TXFULL	0xf0008804
UART_RXEMPTY	0xf0008808
UART_EV_STATUS	0xf000880c
UART_EV_PENDING	0xf0008810
UART_EV_ENABLE	0xf0008814
UART_TXEMPTY	0xf0008818
UART_RXFULL	0xf000881c
UART_XOVER_RXTX	0xf0008820
UART_XOVER_TXFULL	0xf0008824
UART_XOVER_RXEMPTY	0xf0008828
UART_XOVER_EV_STATUS	0xf000882c
UART_XOVER_EV_PENDING	0xf0008830
UART_XOVER_EV_ENABLE	0xf0008834
UART_XOVER_TXEMPTY	0xf0008838
UART_XOVER_RXFULL	0xf000883c

UART_RXTX

Address: $0xf0008800 + 0x0 = 0xf0008800$



Fig. 21.210: UART_RXTX

UART_TXFULL

Address: $0xf0008800 + 0x4 = 0xf0008804$

TX FIFO Full.



Fig. 21.211: UART_TXFULL

UART_RXEMPTY

Address: $0xf0008800 + 0x8 = 0xf0008808$

RX FIFO Empty.

UART_EV_STATUS

Address: $0xf0008800 + 0xc = 0xf000880c$

This register contains the current raw level of the rx event trigger. Writes to this register have no effect.

Field	Name	Description
[0]	TX	Level of the tx event
[1]	RX	Level of the rx event



Fig. 21.212: UART_RXEMPTY

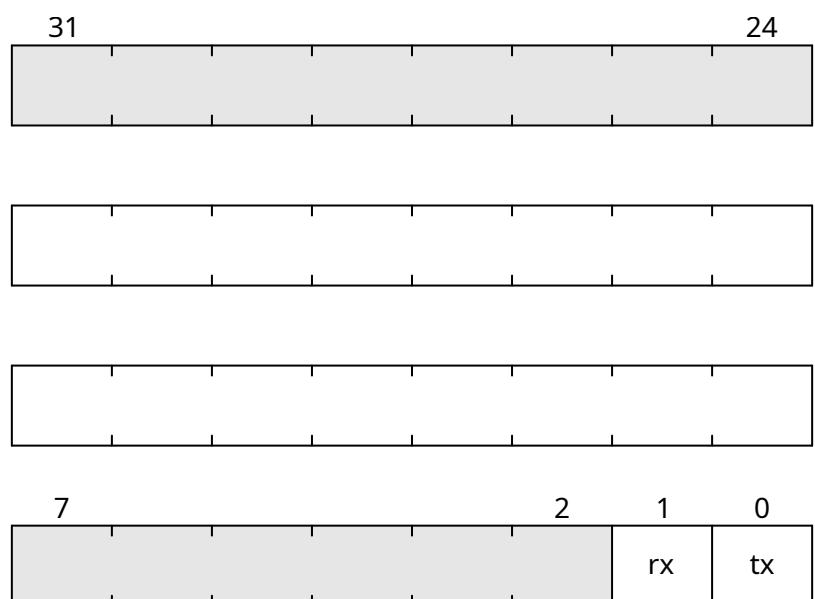


Fig. 21.213: UART_EV_STATUS

UART_EV_PENDING

Address: $0xf0008800 + 0x10 = 0xf0008810$

When a rx event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.

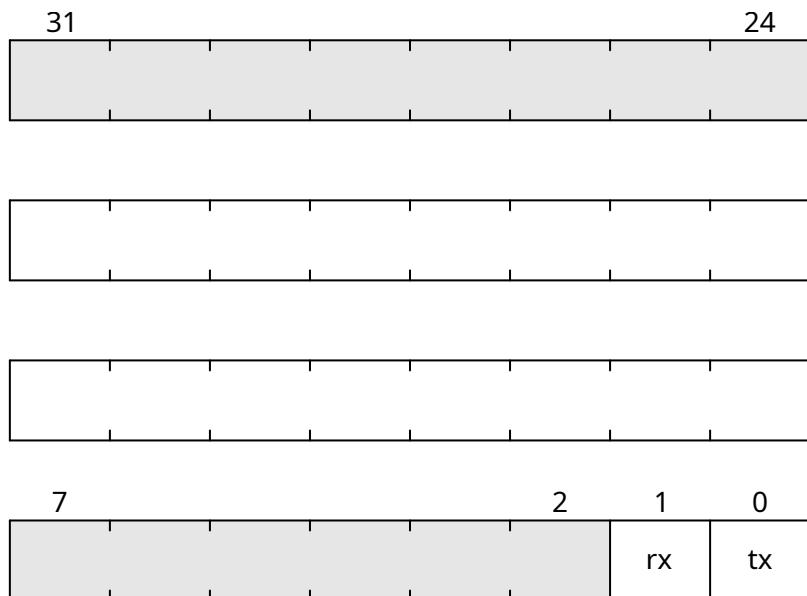


Fig. 21.214: UART_EV_PENDING

Field	Name	Description
[0]	TX	1 if a tx event occurred. This Event is triggered on a falling edge.
[1]	RX	1 if a rx event occurred. This Event is triggered on a falling edge.

UART_EV_ENABLE

Address: $0xf0008800 + 0x14 = 0xf0008814$

This register enables the corresponding rx events. Write a 0 to this register to disable individual events.

Field	Name	Description
[0]	TX	Write a 1 to enable the tx Event
[1]	RX	Write a 1 to enable the rx Event

UART_TXEMPTY

Address: $0xf0008800 + 0x18 = 0xf0008818$

TX FIFO Empty.

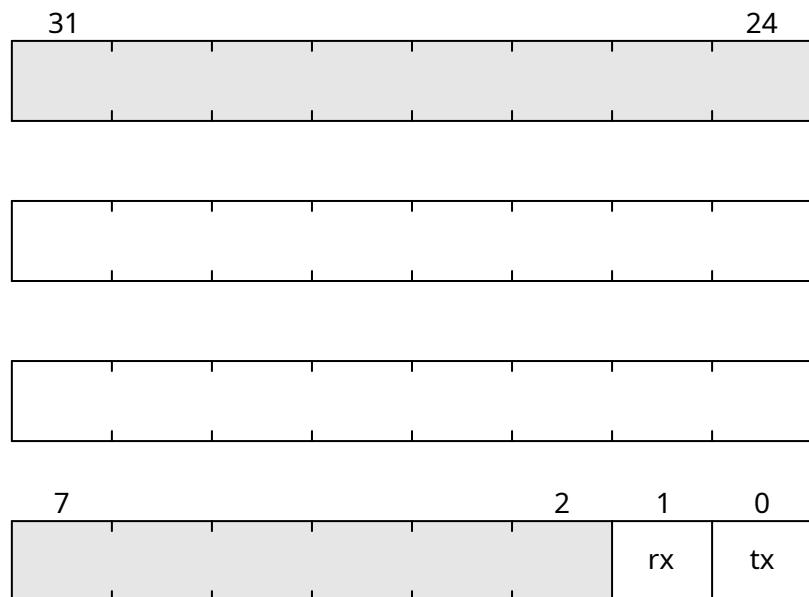


Fig. 21.215: **UART_EV_ENABLE**



Fig. 21.216: **UART_TXEMPTY**

UART_RXFULL

Address: $0xf0008800 + 0x1c = 0xf000881c$

RX FIFO Full.

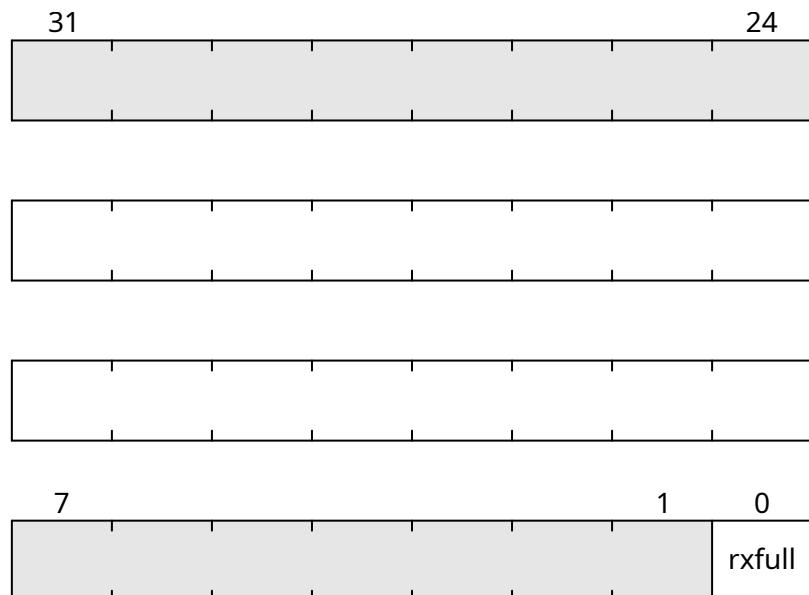


Fig. 21.217: UART_RXFULL

UART_XOVER_RXTX

Address: $0xf0008800 + 0x20 = 0xf0008820$

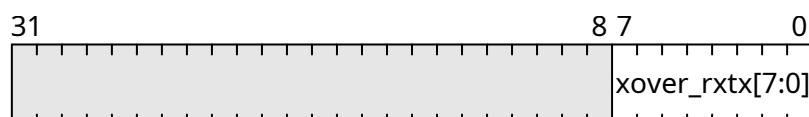


Fig. 21.218: UART_XOVER_RXTX

UART_XOVER_TXFULL

Address: $0xf0008800 + 0x24 = 0xf0008824$

TX FIFO Full.

UART_XOVER_RXEMPTY

Address: $0xf0008800 + 0x28 = 0xf0008828$

RX FIFO Empty.



Fig. 21.219: `UART_XOVER_TXFULL`

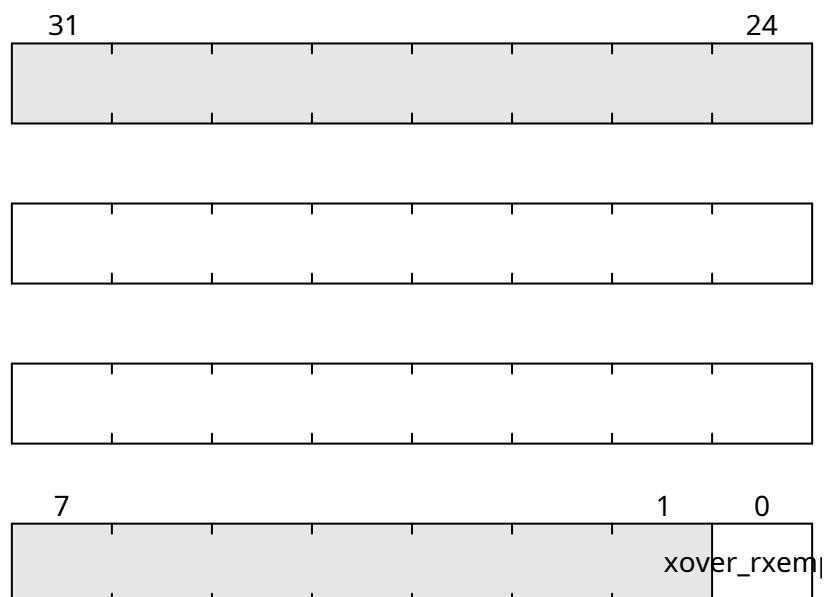


Fig. 21.220: `UART_XOVER_RXEMPTY`

UART_XOVER_EV_STATUS

Address: $0xf0008800 + 0x2c = 0xf000882c$

This register contains the current raw level of the rx event trigger. Writes to this register have no effect.

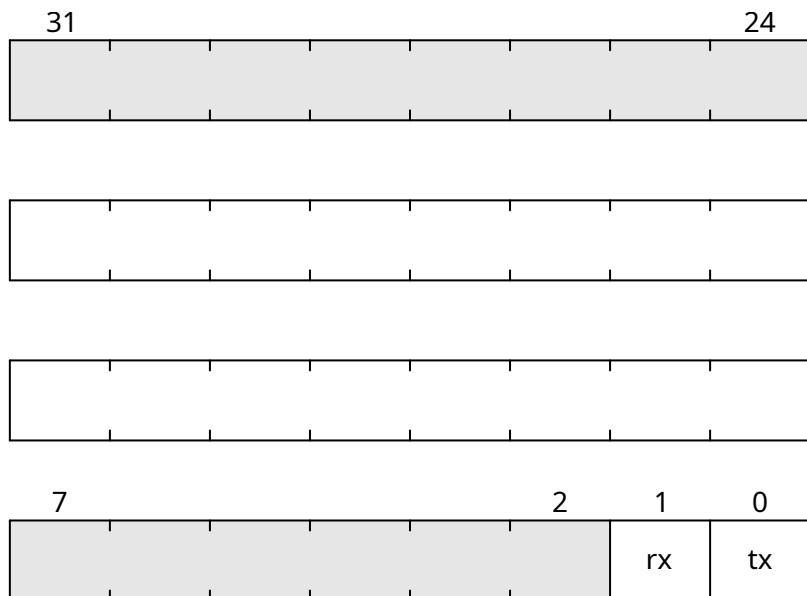


Fig. 21.221: UART_XOVER_EV_STATUS

Field	Name	Description
[0]	TX	Level of the tx event
[1]	RX	Level of the rx event

UART_XOVER_EV_PENDING

Address: $0xf0008800 + 0x30 = 0xf0008830$

When a rx event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.

Field	Name	Description
[0]	TX	1 if a tx event occurred. This Event is triggered on a falling edge .
[1]	RX	1 if a rx event occurred. This Event is triggered on a falling edge .

UART_XOVER_EV_ENABLE

Address: $0xf0008800 + 0x34 = 0xf0008834$

This register enables the corresponding rx events. Write a 0 to this register to disable individual events.

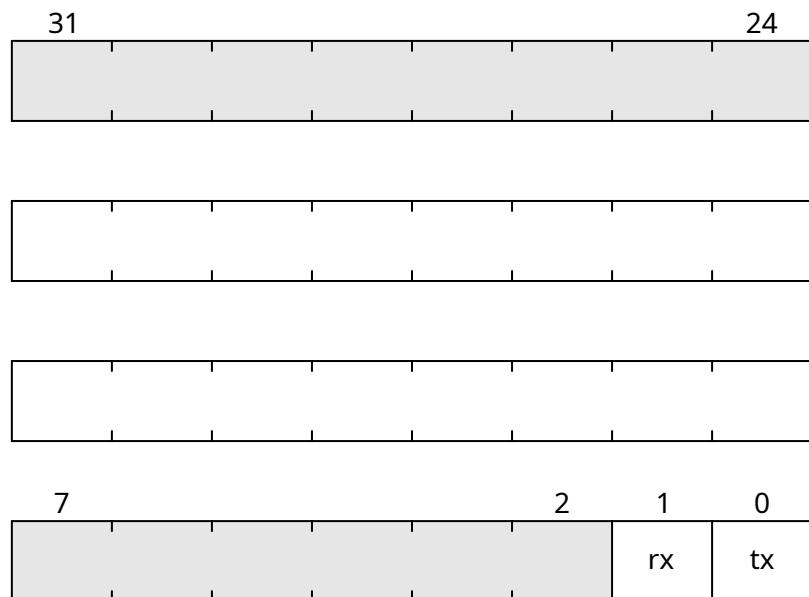


Fig. 21.222: `UART_XOVER_EV_PENDING`

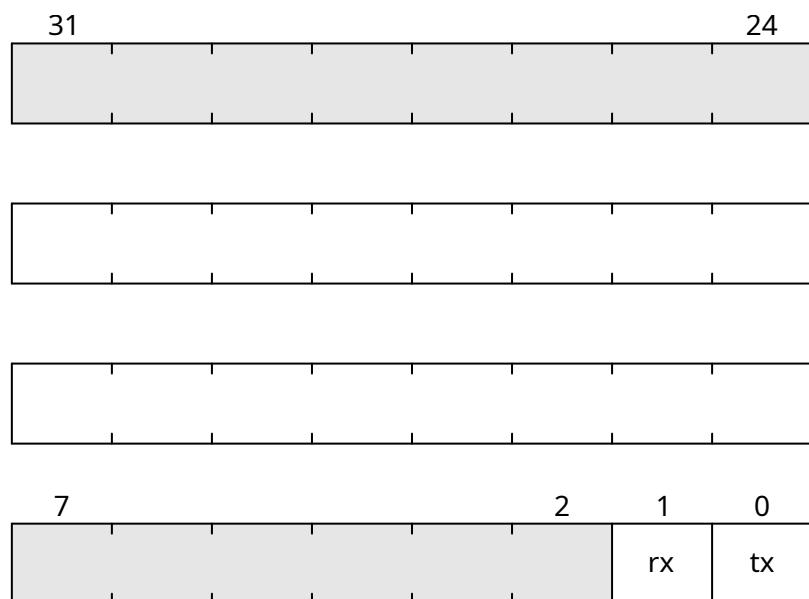


Fig. 21.223: `UART_XOVER_EV_ENABLE`

Field	Name	Description
[0]	TX	Write a 1 to enable the tx Event
[1]	RX	Write a 1 to enable the rx Event

UART_XOVER_TXEMPTY

Address: $0xf0008800 + 0x38 = 0xf0008838$

TX FIFO Empty.

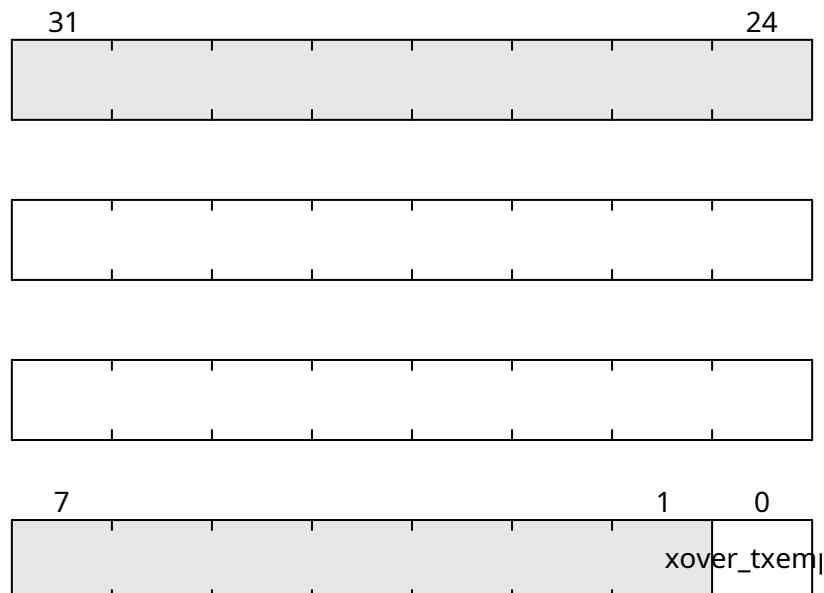


Fig. 21.224: UART_XOVER_TXEMPTY

UART_XOVER_RXFULL

Address: $0xf0008800 + 0x3c = 0xf000883c$

RX FIFO Full.

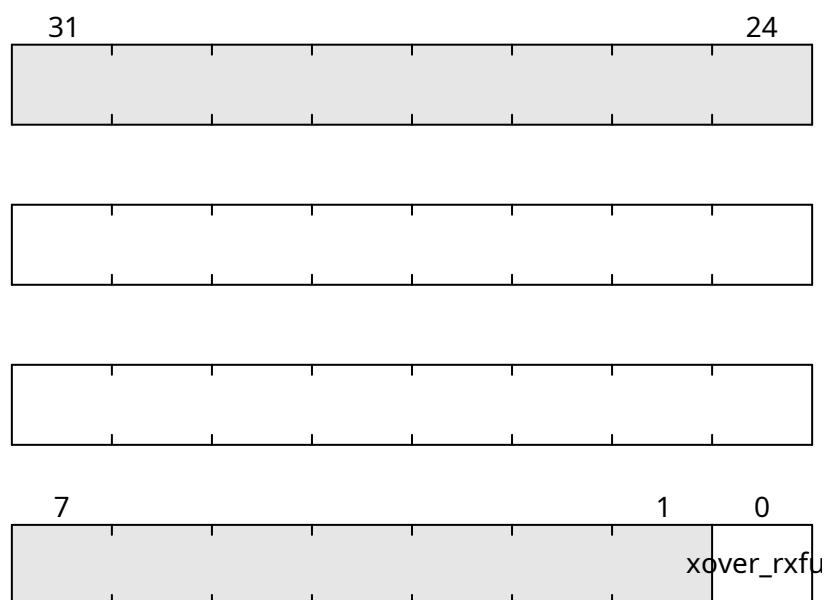


Fig. 21.225: UART_XOVER_RXFULL

CHAPTER
TWENTYTWO

DOCUMENTATION FOR ROW HAMMER TESTER LPDDR4 TEST BOARD

22.1 Modules

22.1.1 Interrupt Controller

This device has an EventManager-based interrupt system. Individual modules generate *events* which are wired into a central interrupt controller.

When an interrupt occurs, you should look the interrupt number up in the CPU- specific interrupt table and then call the relevant module.

Assigned Interrupts

The following interrupts are assigned on this system:

Interrupt	Module
1	<i>TIMER0</i>
0	<i>UART</i>

22.2 Register Groups

22.2.1 LEDS

Register Listing for LEDS

Register	Address
<i>LEDS_OUT</i>	<i>0xf0000000</i>

LEDS_OUT

Address: 0xf0000000 + 0x0 = 0xf0000000

Led Output(s) Control.

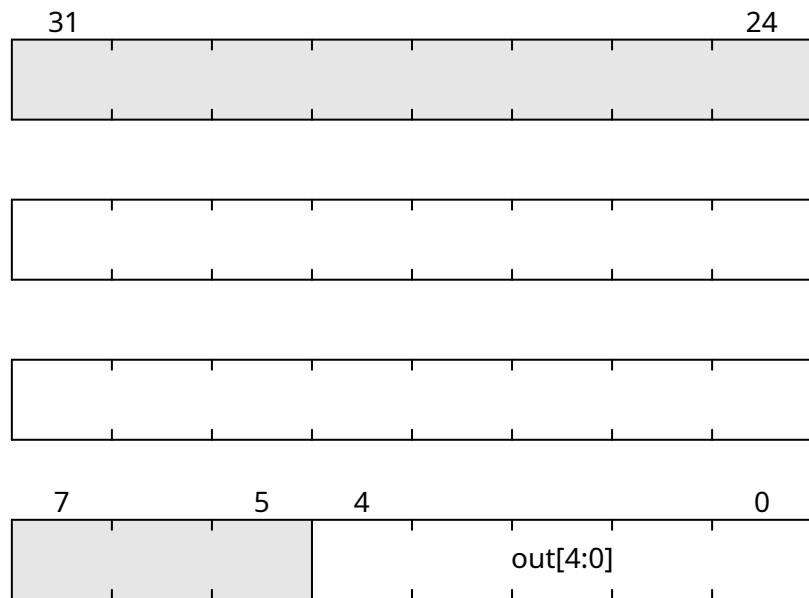


Fig. 22.1: LEDS_OUT

22.2.2 DDRPHY

Register Listing for DDRPHY

Register	Address
<i>DDRPHY_RST</i>	0xf0000800
<i>DDRPHY_WLEVEL_EN</i>	0xf0000804
<i>DDRPHY_WLEVEL_STROBE</i>	0xf0000808
<i>DDRPHY_DLW_SEL</i>	0xf000080c
<i>DDRPHY_RDLY_DQ_BITSLIP_RST</i>	0xf0000810
<i>DDRPHY_RDLY_DQ_BITSLIP</i>	0xf0000814
<i>DDRPHY_WDLY_DQ_BITSLIP_RST</i>	0xf0000818
<i>DDRPHY_WDLY_DQ_BITSLIP</i>	0xf000081c
<i>DDRPHY_RDPHASE</i>	0xf0000820
<i>DDRPHY_WRPHASE</i>	0xf0000824
<i>DDRPHY_HALF_SYS8X_TAPS</i>	0xf0000828
<i>DDRPHY_RDLY_DQ_RST</i>	0xf000082c
<i>DDRPHY_RDLY_DQ_INC</i>	0xf0000830
<i>DDRPHY_RDLY_DQS_RST</i>	0xf0000834
<i>DDRPHY_RDLY_DQS_INC</i>	0xf0000838
<i>DDRPHY_CDLY_RST</i>	0xf000083c
<i>DDRPHY_CDLY_INC</i>	0xf0000840
<i>DDRPHY_WDLY_DQ_RST</i>	0xf0000844
<i>DDRPHY_WDLY_DQ_INC</i>	0xf0000848
<i>DDRPHY_WDLY_DQS_RST</i>	0xf000084c
<i>DDRPHY_WDLY_DQS_INC</i>	0xf0000850

DDRPHY_RST

Address: $0xf0000800 + 0x0 = 0xf0000800$



Fig. 22.2: DDRPHY_RST

DDRPHY_WLEVEL_EN

Address: $0xf0000800 + 0x4 = 0xf0000804$

DDRPHY_WLEVEL_STROBE

Address: $0xf0000800 + 0x8 = 0xf0000808$

DDRPHY_DLY_SEL

Address: $0xf0000800 + 0xc = 0xf000080c$

DDRPHY_RDLY_DQ_BITSLIP_RST

Address: $0xf0000800 + 0x10 = 0xf0000810$



Fig. 22.3: DDRPHY_WLEVEL_EN

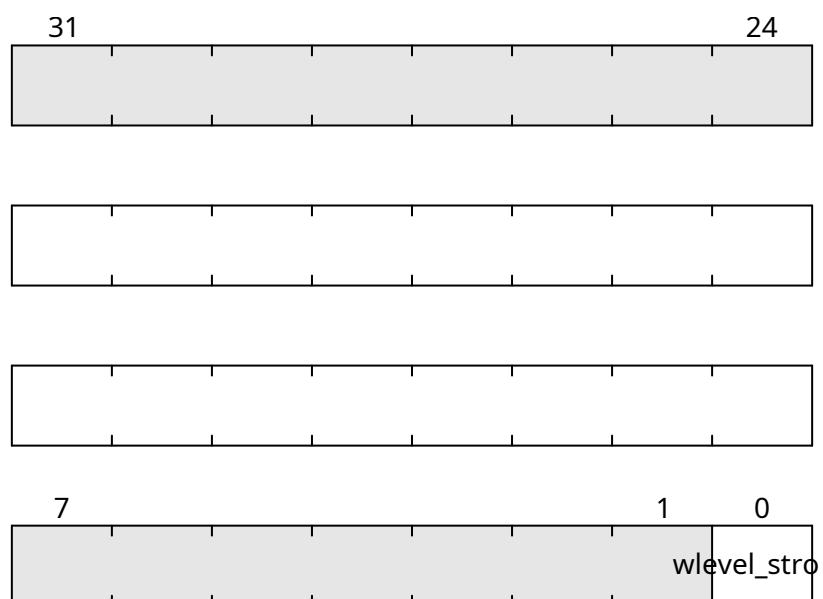


Fig. 22.4: DDRPHY_WLEVEL_STROBE

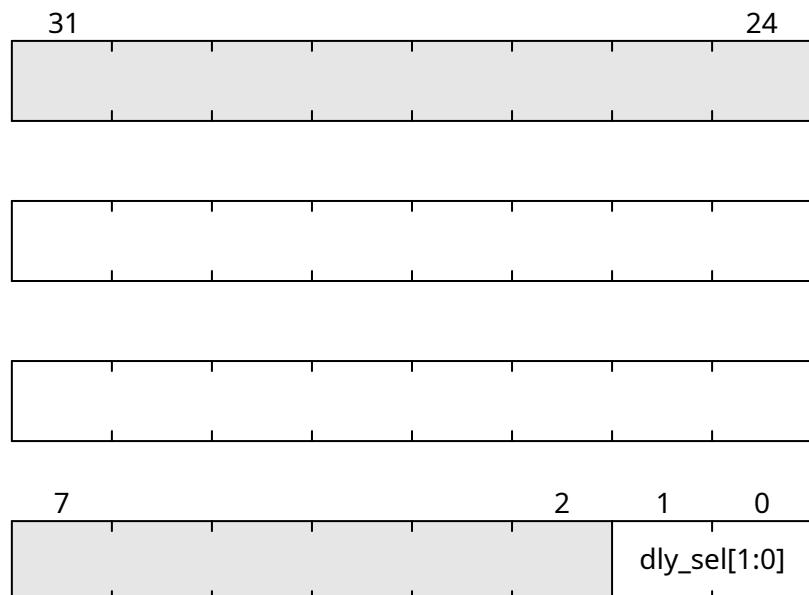


Fig. 22.5: DDRPHY_DLY_SEL

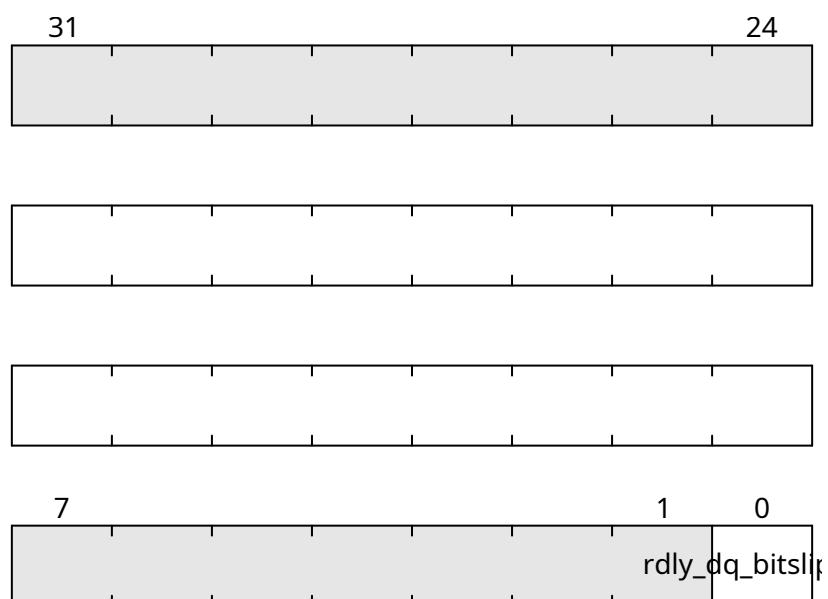


Fig. 22.6: DDRPHY_RDLY_DQ_BITSLIP_RST

DDRPHY_RDLY_DQ_BITSLIP

Address: $0xf0000800 + 0x14 = 0xf0000814$

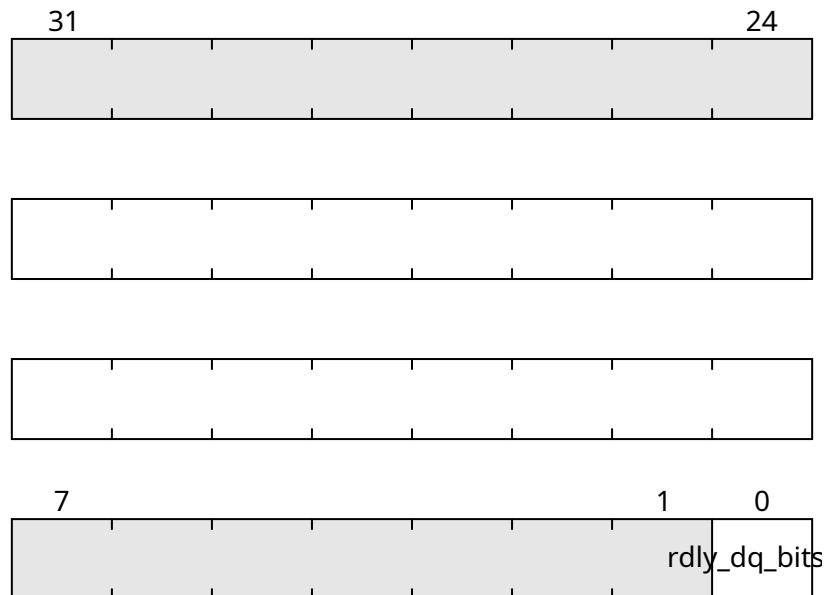


Fig. 22.7: DDRPHY_RDLY_DQ_BITSLIP

DDRPHY_WDLY_DQ_BITSLIP_RST

Address: $0xf0000800 + 0x18 = 0xf0000818$

DDRPHY_WDLY_DQ_BITSLIP

Address: $0xf0000800 + 0x1c = 0xf000081c$

DDRPHY_RDPHASE

Address: $0xf0000800 + 0x20 = 0xf0000820$

DDRPHY_WRPHASE

Address: $0xf0000800 + 0x24 = 0xf0000824$

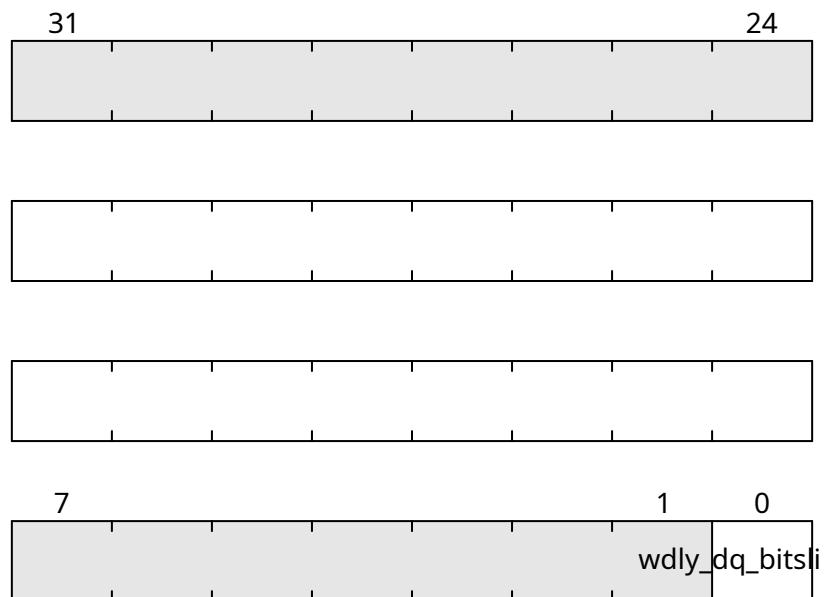


Fig. 22.8: DDRPHY_WDLY_DQ_BITSLIP_RST

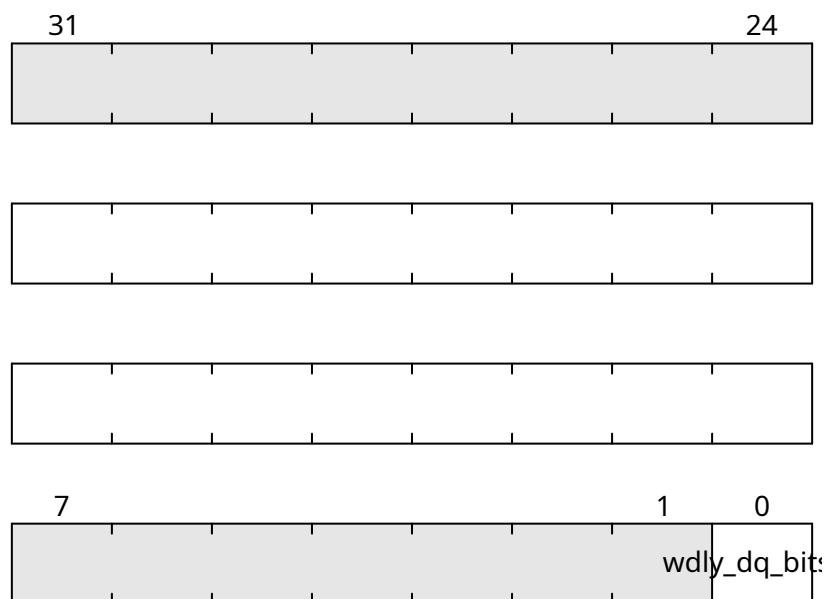


Fig. 22.9: DDRPHY_WDLY_DQ_BITSLIP

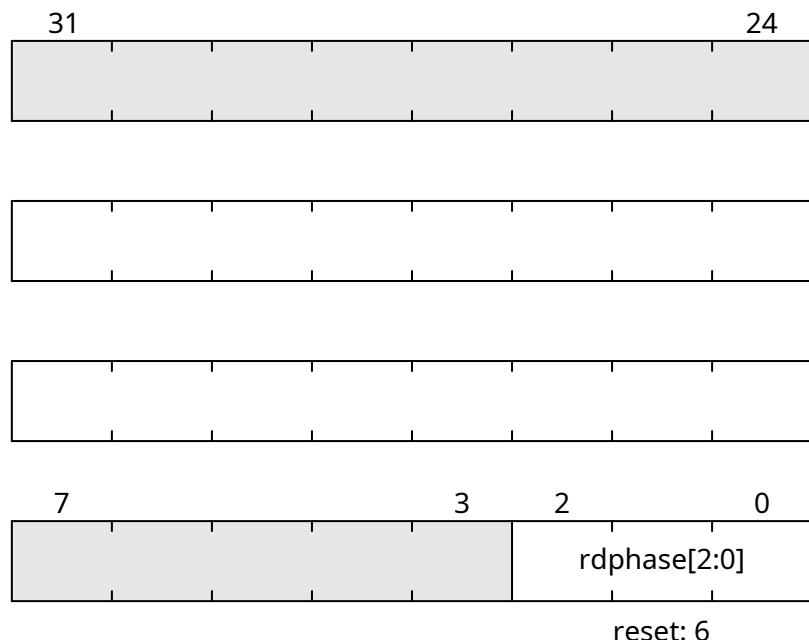


Fig. 22.10: DDRPHY_RDPHASE

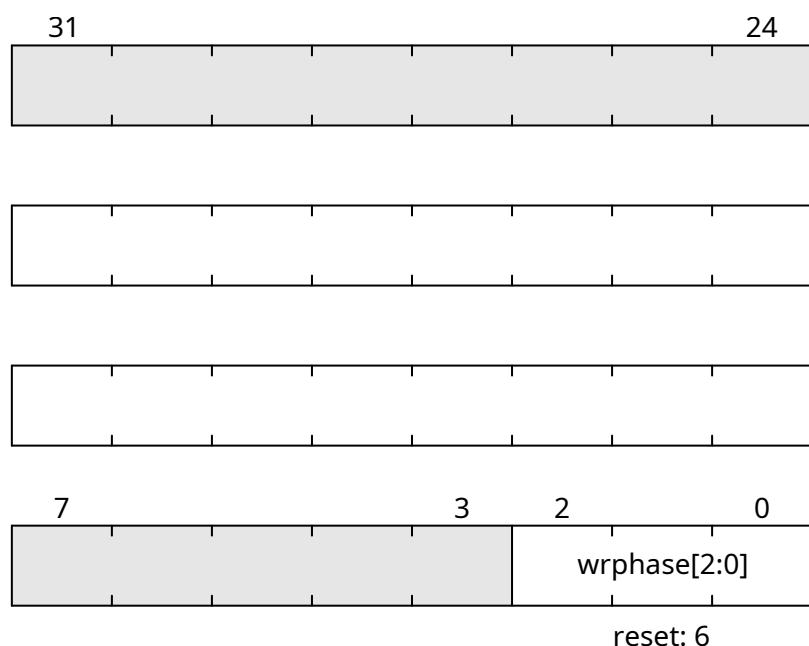


Fig. 22.11: DDRPHY_WRPAGE

DDRPHY_HALF_SYS8X_TAPS

Address: $0xf0000800 + 0x28 = 0xf0000828$

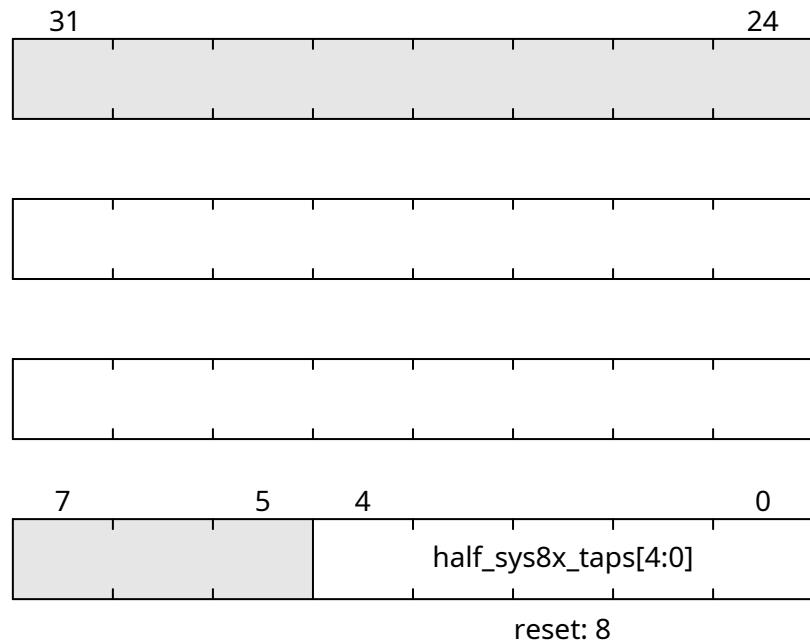


Fig. 22.12: DDRPHY_HALF_SYS8X_TAPS

DDRPHY_RDLY_DQ_RST

Address: $0xf0000800 + 0x2c = 0xf000082c$

DDRPHY_RDLY_DQ_INC

Address: $0xf0000800 + 0x30 = 0xf0000830$

DDRPHY_RDLY_DQS_RST

Address: $0xf0000800 + 0x34 = 0xf0000834$

DDRPHY_RDLY_DQS_INC

Address: $0xf0000800 + 0x38 = 0xf0000838$



Fig. 22.13: DDRPHY_RDLY_DQ_RST

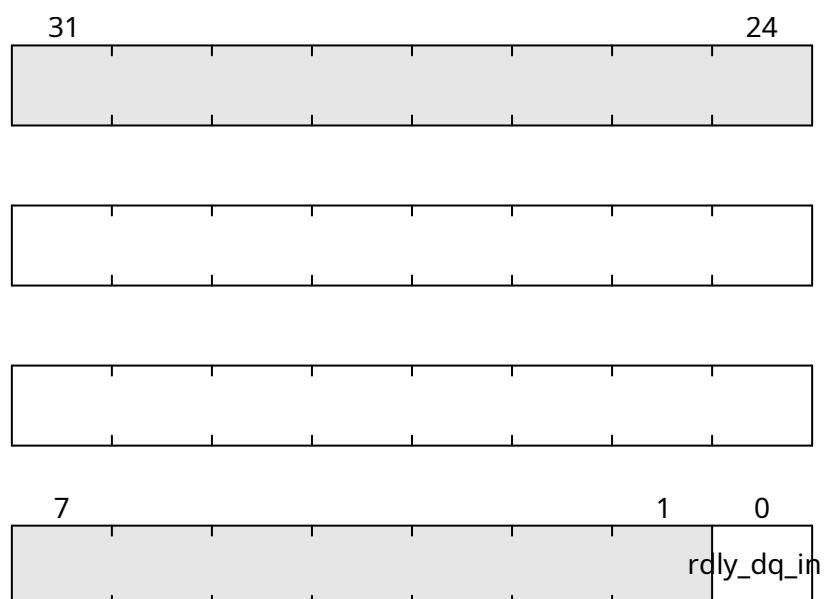


Fig. 22.14: DDRPHY_RDLY_DQ_INC

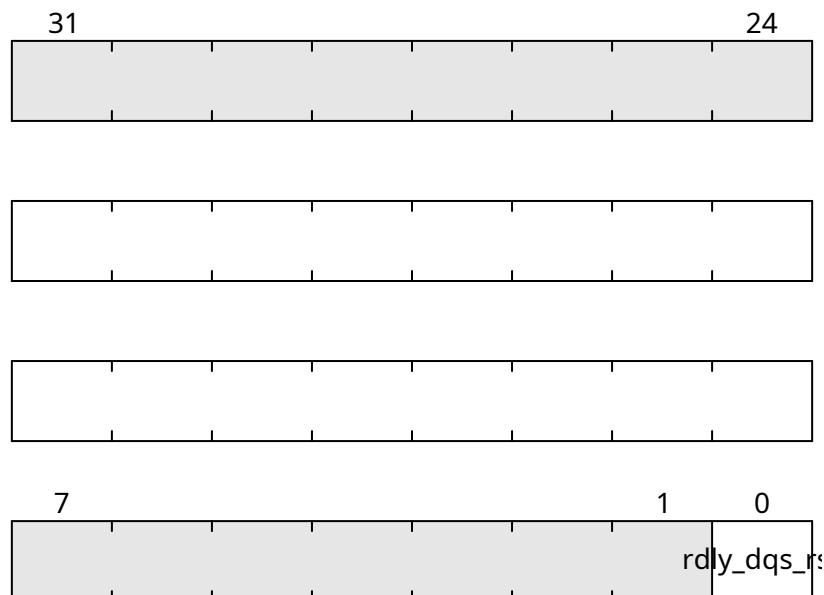


Fig. 22.15: `DDRPHY_RDLY_DQS_RST`

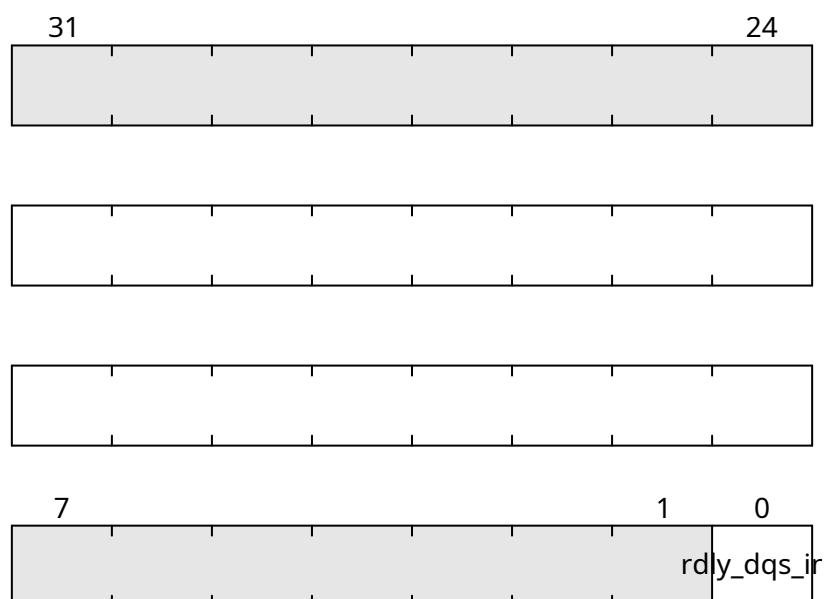


Fig. 22.16: `DDRPHY_RDLY_DQS_INC`

DDRPHY_CDLY_RST

Address: $0xf0000800 + 0x3c = 0xf000083c$



Fig. 22.17: DDRPHY_CDLY_RST

DDRPHY_CDLY_INC

Address: $0xf0000800 + 0x40 = 0xf0000840$

DDRPHY_WDLY_DQ_RST

Address: $0xf0000800 + 0x44 = 0xf0000844$

DDRPHY_WDLY_DQ_INC

Address: $0xf0000800 + 0x48 = 0xf0000848$

DDRPHY_WDLY_DQS_RST

Address: $0xf0000800 + 0x4c = 0xf000084c$

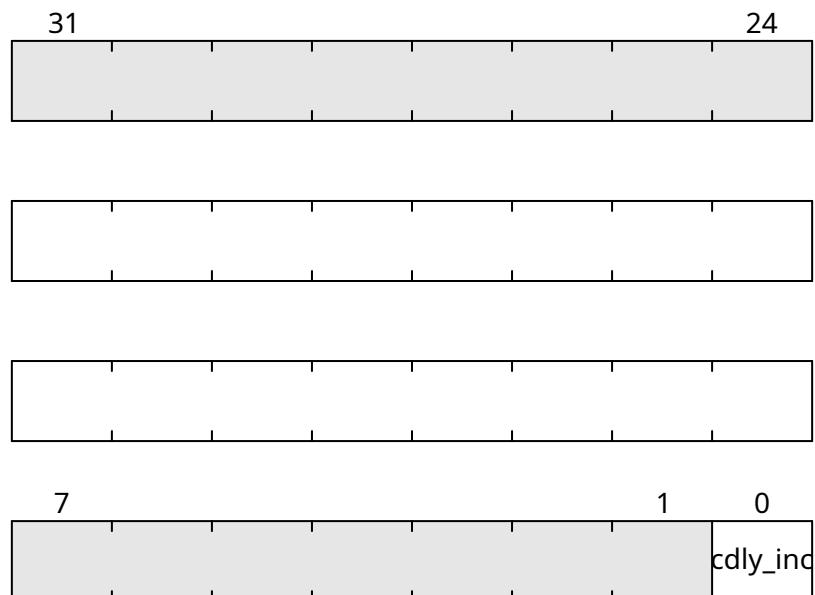


Fig. 22.18: DDRPHY_CDLY_INC

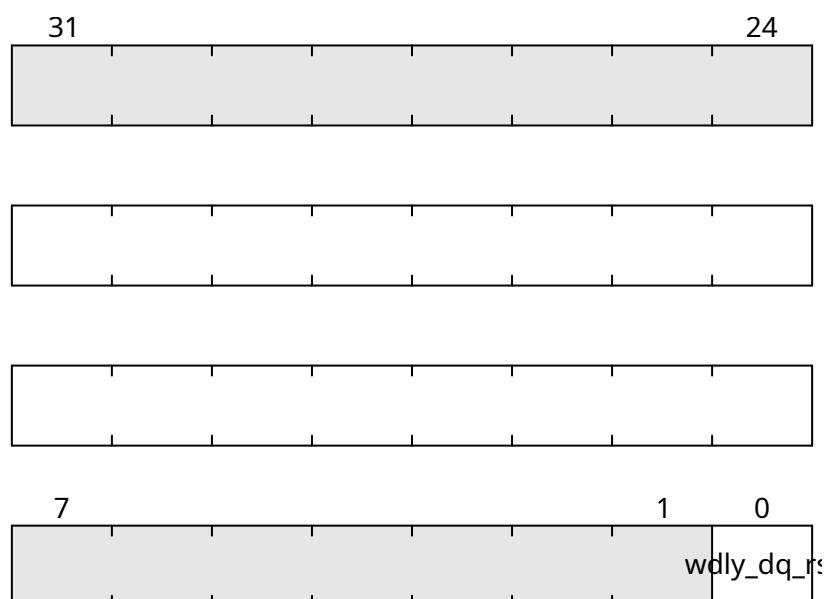


Fig. 22.19: DDRPHY_WDLY_DQ_RST

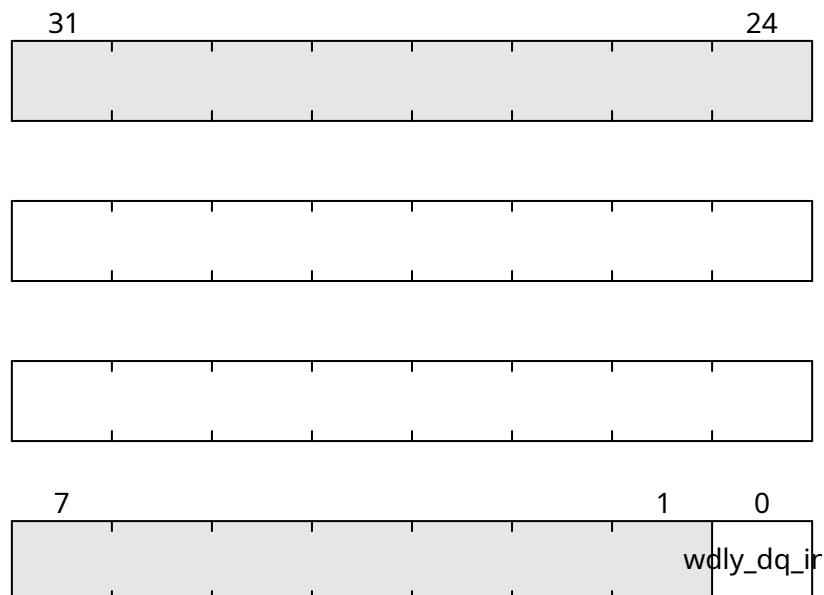


Fig. 22.20: DDRPHY_WDLY_DQ_INC

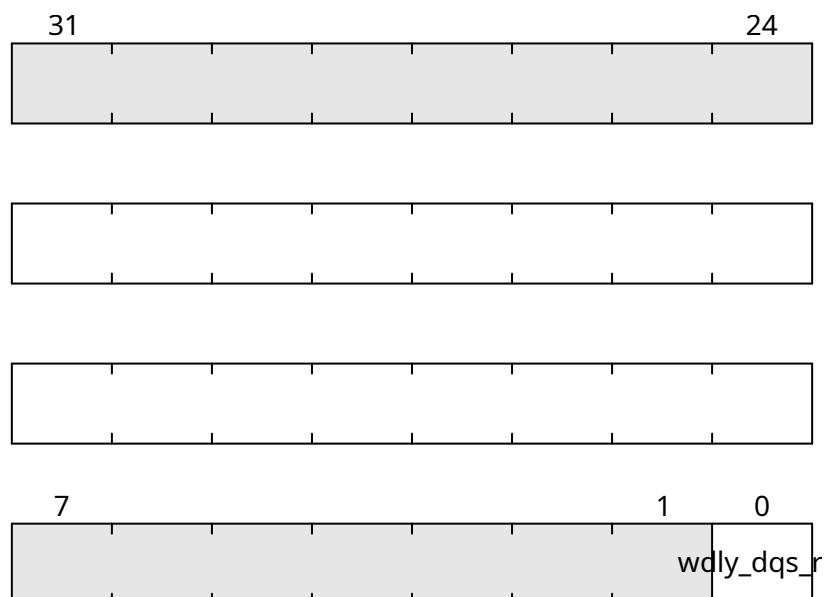


Fig. 22.21: DDRPHY_WDLY_DQS_RST

DDRPHY_WDLY_DQS_INC

Address: $0xf0000800 + 0x50 = 0xf0000850$



Fig. 22.22: DDRPHY_WDLY_DQS_INC

22.2.3 CONTROLLER_SETTINGS

Allows to change LiteDRAMController behaviour at runtime

Register Listing for CONTROLLER_SETTINGS

Register	Address
<i>CONTROLLER_SETTINGS_REFRESH</i>	<i>0xf0001000</i>

CONTROLLER_SETTINGS_REFRESH

Address: $0xf0001000 + 0x0 = 0xf0001000$

Enable/disable Refresh commands sending

22.2.4 DDRCTRL

Register Listing for DDRCTRL

Register	Address
<i>DDRCTRL_INIT_DONE</i>	<i>0xf0001800</i>
<i>DDRCTRL_INIT_ERROR</i>	<i>0xf0001804</i>

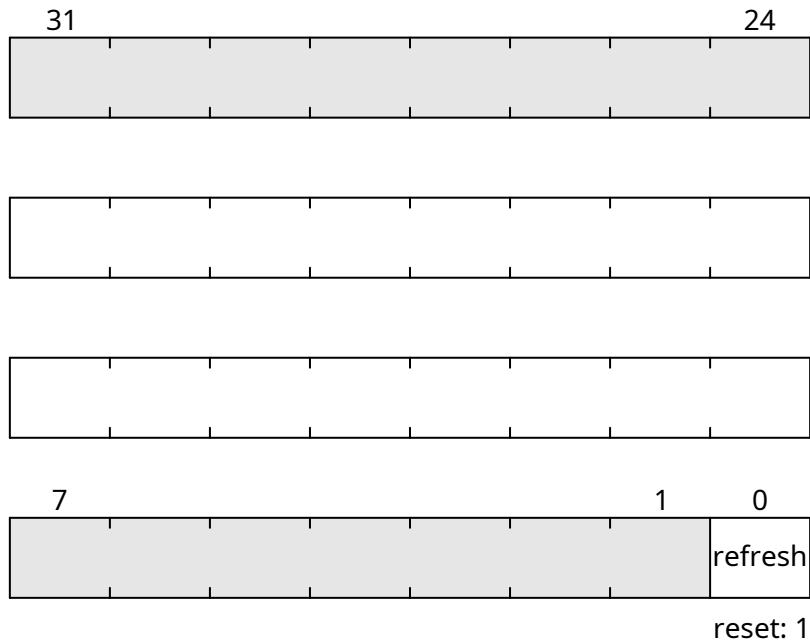


Fig. 22.23: CONTROLLER_SETTINGS_REFRESH

DDRCTRL_INIT_DONE

Address: $0xf0001800 + 0x0 = 0xf0001800$

DDRCTRL_INIT_ERROR

Address: $0xf0001800 + 0x4 = 0xf0001804$

22.2.5 ROWHAMMER

Row Hammer DMA attacker

This module allows to perform a Row Hammer attack by configuring it with two addresses that map to different rows of a single bank. When enabled, it will perform alternating DMA reads from the given locations, which will result in the DRAM controller having to repeatedly open/close rows at each read access.

Register Listing for ROWHAMMER

Register	Address
<i>ROWHAMMER_ENABLED</i>	<i>0xf0002000</i>
<i>ROWHAMMER_ADDRESS1</i>	<i>0xf0002004</i>
<i>ROWHAMMER_ADDRESS2</i>	<i>0xf0002008</i>
<i>ROWHAMMER_COUNT</i>	<i>0xf000200c</i>



Fig. 22.24: DDRCTRL_INIT_DONE

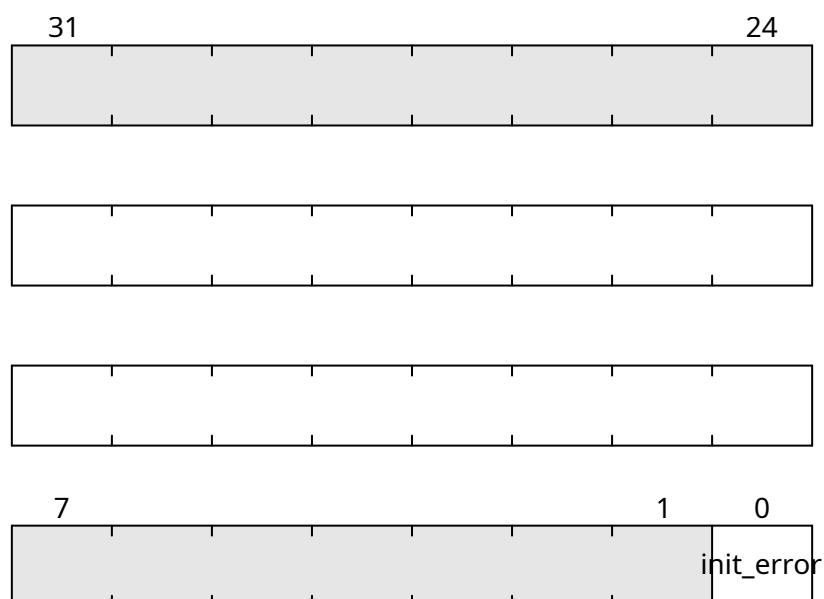


Fig. 22.25: DDRCTRL_INIT_ERROR

ROWHAMMER_ENABLED

Address: $0xf0002000 + 0x0 = 0xf0002000$

Used to start/stop the operation of the module



Fig. 22.26: ROWHAMMER_ENABLED

ROWHAMMER_ADDRESS1

Address: $0xf0002000 + 0x4 = 0xf0002004$

First attacked address

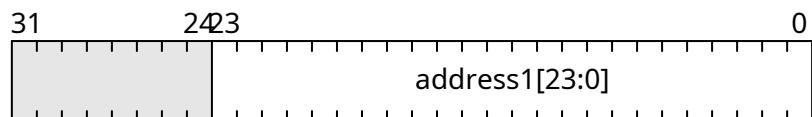


Fig. 22.27: ROWHAMMER_ADDRESS1

ROWHAMMER_ADDRESS2

Address: $0xf0002000 + 0x8 = 0xf0002008$

Second attacked address

ROWHAMMER_COUNT

Address: $0xf0002000 + 0xc = 0xf000200c$

This is the number of DMA accesses performed. When the module is enabled, the value can be freely read. When the module is disabled, the register is clear-on-write and has to be read before the next attack.

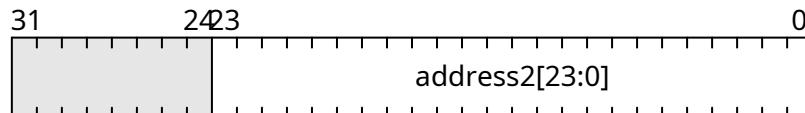


Fig. 22.28: ROWHAMMER_ADDRESS2

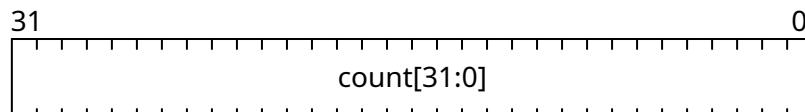


Fig. 22.29: ROWHAMMER_COUNT

22.2.6 WRITER

DMA DRAM writer.

Allows to fill DRAM with a predefined pattern using DMA.

Pattern

Provides access to RAM to store access pattern: *mem_addr* and *mem_data*. The pattern address space can be limited using the *data_mask*.

For example, having *mem_addr* filled with [0x04, 0x02, 0x03, ...] and *mem_data* filled with [0xff, 0xaa, 0x55, ...] and setting *data_mask* = 0b01, the pattern [(address, data), ...] written will be: [(0x04, 0xff), (0x02, 0xaa), (0x04, 0xff), ...] (wraps due to masking).

DRAM memory range that is being accessed can be configured using *mem_mask*.

To use this module, make sure that *ready* is 1, then write the desired number of transfers to *count*. Writing to the *start* CSR will initialize the operation. When the operation is ongoing *ready* will be 0.

Register Listing for WRITER

Register	Address
WRITER_START	0xf0002800
WRITER_READY	0xf0002804
WRITER_MODULO	0xf0002808
WRITER_COUNT	0xf000280c
WRITER_DONE	0xf0002810
WRITER_MEM_MASK	0xf0002814
WRITER_DATA_MASK	0xf0002818
WRITER_DATA_DIV	0xf000281c
WRITER_INVERTER_DIVISOR_MASK	0xf0002820
WRITER_INVERTER_SELECTION_MASK	0xf0002824
WRITER_LAST_ADDRESS	0xf0002828

WRITER_START

Address: $0xf0002800 + 0x0 = 0xf0002800$

Write to the register starts the transfer (if ready=1)



Fig. 22.30: WRITER_START

WRITER_READY

Address: $0xf0002800 + 0x4 = 0xf0002804$

Indicates that the transfer is not ongoing

WRITER_MODULO

Address: $0xf0002800 + 0x8 = 0xf0002808$

When set use modulo to calculate DMA transfers address rather than bit masking

WRITER_COUNT

Address: $0xf0002800 + 0xc = 0xf000280c$

Desired number of DMA transfers

WRITER_DONE

Address: $0xf0002800 + 0x10 = 0xf0002810$

Number of completed DMA transfers



Fig. 22.31: WRITER_READY



Fig. 22.32: WRITER_MODULO

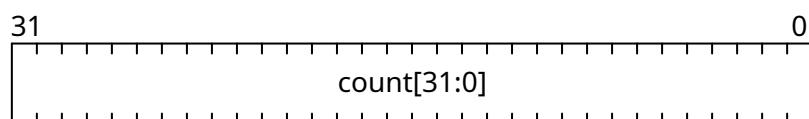


Fig. 22.33: WRITER_COUNT

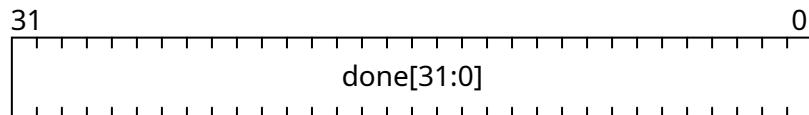


Fig. 22.34: WRITER_DONE

WRITER_MEM_MASK

Address: $0xf0002800 + 0x14 = 0xf0002814$

DRAM address mask for DMA transfers

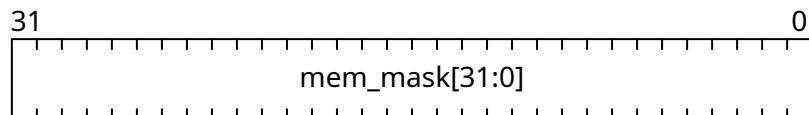


Fig. 22.35: WRITER_MEM_MASK

WRITER_DATA_MASK

Address: $0xf0002800 + 0x18 = 0xf0002818$

Pattern memory address mask

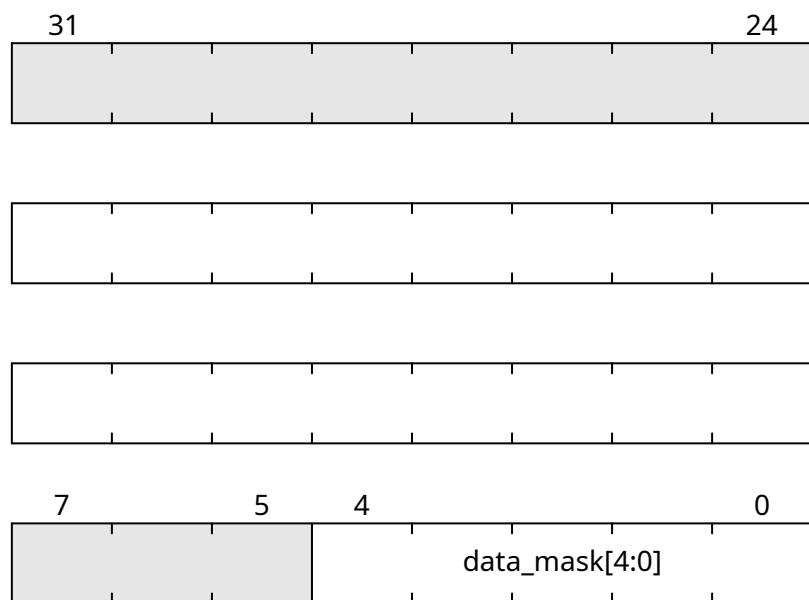


Fig. 22.36: WRITER_DATA_MASK

WRITER_DATA_DIV

Address: $0xf0002800 + 0x1c = 0xf000281c$

Pattern memory address divisor-1

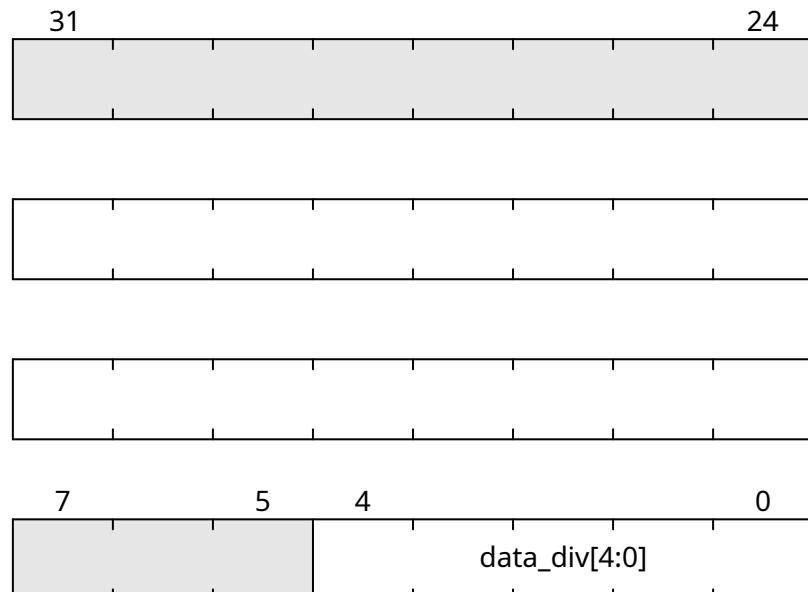


Fig. 22.37: WRITER_DATA_DIV

WRITER_INVERTER_DIVISOR_MASK

Address: $0xf0002800 + 0x20 = 0xf0002820$

Divisor mask for selecting rows for which pattern data gets inverted

WRITER_INVERTER_SELECTION_MASK

Address: $0xf0002800 + 0x24 = 0xf0002824$

Selection mask for selecting rows for which pattern data gets inverted

WRITER_LAST_ADDRESS

Address: $0xf0002800 + 0x28 = 0xf0002828$

Number of completed DMA transfers

22.2.7 READER

DMA DRAM reader.

Allows to check DRAM contents against a predefined pattern using DMA.

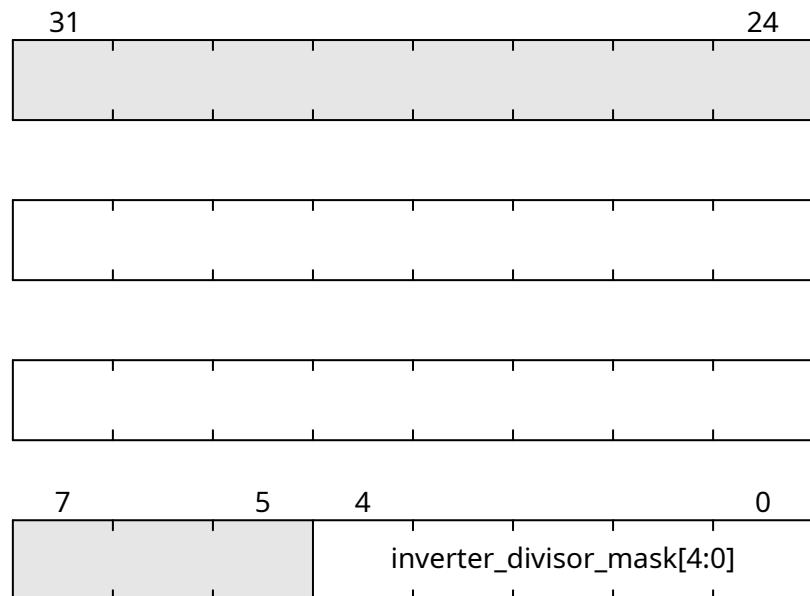


Fig. 22.38: WRITER_INVERTER_DIVISOR_MASK

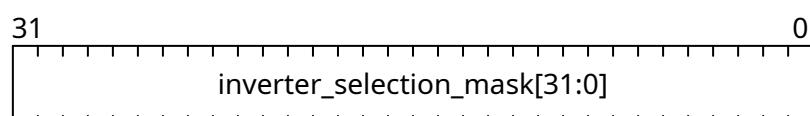


Fig. 22.39: WRITER_INVERTER_SELECTION_MASK

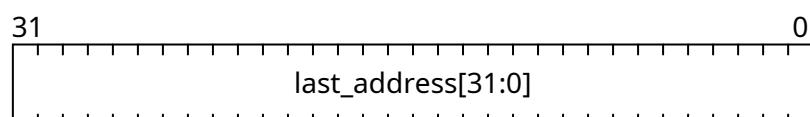


Fig. 22.40: WRITER_LAST_ADDRESS

Pattern

Provides access to RAM to store access pattern: *mem_addr* and *mem_data*. The pattern address space can be limited using the *data_mask*.

For example, having *mem_addr* filled with [0x04, 0x02, 0x03, ...] and *mem_data* filled with [0xff, 0xaa, 0x55, ...] and setting *data_mask* = 0b01, the pattern [(address, data), ...] written will be: [(0x04, 0xff), (0x02, 0xaa), (0x04, 0xff), ...] (wraps due to masking).

DRAM memory range that is being accessed can be configured using *mem_mask*.

To use this module, make sure that *ready* is 1, then write the desired number of transfers to *count*. Writing to the *start* CSR will initialize the operation. When the operation is ongoing *ready* will be 0.

Reading errors

This module allows to check the locations of errors in the memory. It scans the configured memory area and compares the values read to the predefined pattern. If *skip_fifo* is 0, this module will stop after each error encountered, so that it can be examined. Wait until the *error_ready* CSR is 1. Then use the CSRs *error_offset*, *error_data* and *error_expected* to examine the errors in the current transfer. To continue reading, write 1 to *error_continue* CSR. Setting *skip_fifo* to 1 will disable this behavior entirely.

The final number of errors can be read from *error_count*. NOTE: This value represents the number of erroneous *DMA transfers*.

The current progress can be read from the *done* CSR.

Register Listing for READER

Register	Address
<i>READER_START</i>	0xf0003000
<i>READER_READY</i>	0xf0003004
<i>READER_MODULO</i>	0xf0003008
<i>READER_COUNT</i>	0xf000300c
<i>READER_DONE</i>	0xf0003010
<i>READER_MEM_MASK</i>	0xf0003014
<i>READER_DATA_MASK</i>	0xf0003018
<i>READER_DATA_DIV</i>	0xf000301c
<i>READER_INVERTER_DIVISOR_MASK</i>	0xf0003020
<i>READER_INVERTER_SELECTION_MASK</i>	0xf0003024
<i>READER_ERROR_COUNT</i>	0xf0003028
<i>READER_SKIP_FIFO</i>	0xf000302c
<i>READER_ERROR_OFFSET</i>	0xf0003030
<i>READER_ERROR_DATA7</i>	0xf0003034
<i>READER_ERROR_DATA6</i>	0xf0003038
<i>READER_ERROR_DATA5</i>	0xf000303c
<i>READER_ERROR_DATA4</i>	0xf0003040
<i>READER_ERROR_DATA3</i>	0xf0003044
<i>READER_ERROR_DATA2</i>	0xf0003048
<i>READER_ERROR_DATA1</i>	0xf000304c

continues on next page

Table 22.1 – continued from previous page

Register	Address
<i>READER_ERROR_DATA0</i>	0xf0003050
<i>READER_ERROR_EXPECTED7</i>	0xf0003054
<i>READER_ERROR_EXPECTED6</i>	0xf0003058
<i>READER_ERROR_EXPECTED5</i>	0xf000305c
<i>READER_ERROR_EXPECTED4</i>	0xf0003060
<i>READER_ERROR_EXPECTED3</i>	0xf0003064
<i>READER_ERROR_EXPECTED2</i>	0xf0003068
<i>READER_ERROR_EXPECTED1</i>	0xf000306c
<i>READER_ERROR_EXPECTED0</i>	0xf0003070
<i>READER_ERROR_READY</i>	0xf0003074
<i>READER_ERROR_CONTINUE</i>	0xf0003078

READER_START

Address: $0xf0003000 + 0x0 = 0xf0003000$

Write to the register starts the transfer (if ready=1)



Fig. 22.41: READER_START

READER_READY

Address: $0xf0003000 + 0x4 = 0xf0003004$

Indicates that the transfer is not ongoing



Fig. 22.42: READER_READY

READER_MODULO

Address: $0xf0003000 + 0x8 = 0xf0003008$

When set use modulo to calculate DMA transfers address rather than bit masking

READER_COUNT

Address: $0xf0003000 + 0xc = 0xf000300c$

Desired number of DMA transfers

READER_DONE

Address: $0xf0003000 + 0x10 = 0xf0003010$

Number of completed DMA transfers

READER_MEM_MASK

Address: $0xf0003000 + 0x14 = 0xf0003014$

DRAM address mask for DMA transfers

READER_DATA_MASK

Address: $0xf0003000 + 0x18 = 0xf0003018$

Pattern memory address mask

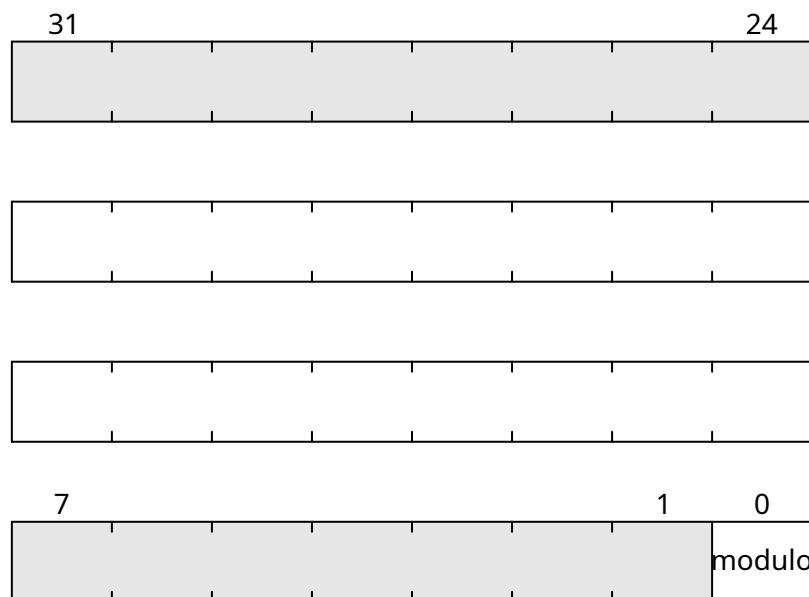


Fig. 22.43: READER_MODULO

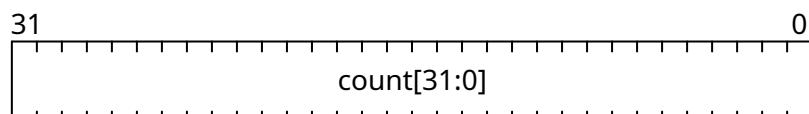


Fig. 22.44: READER_COUNT

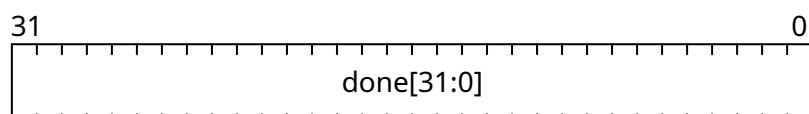


Fig. 22.45: READER_DONE

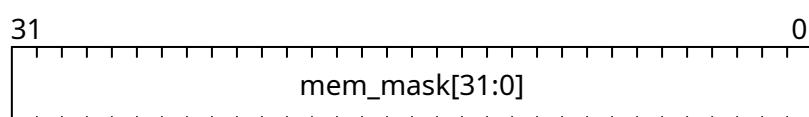


Fig. 22.46: READER_MEM_MASK

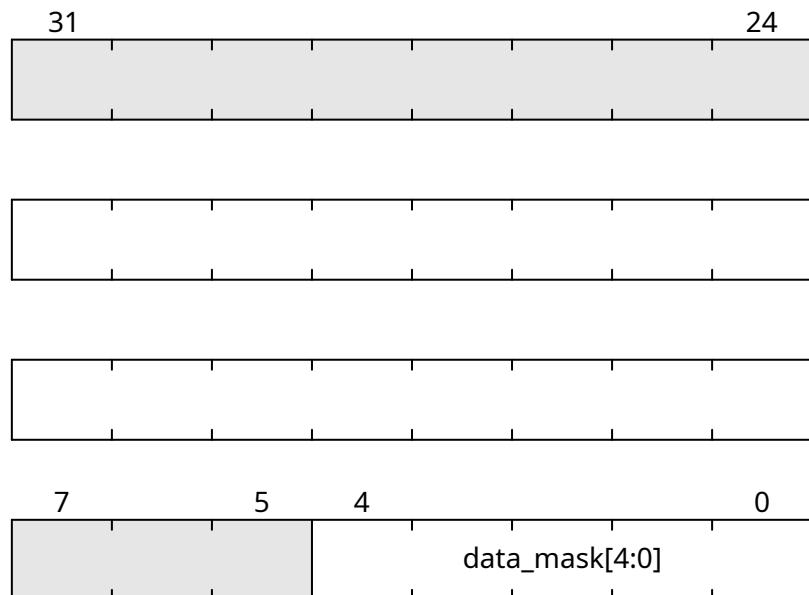


Fig. 22.47: READER_DATA_MASK

READER_DATA_DIV

Address: 0xf0003000 + 0x1c = 0xf000301c

Pattern memory address divisor-1

READER_INVERTER_DIVISOR_MASK

Address: 0xf0003000 + 0x20 = 0xf0003020

Divisor mask for selecting rows for which pattern data gets inverted

READER_INVERTER_SELECTION_MASK

Address: 0xf0003000 + 0x24 = 0xf0003024

Selection mask for selecting rows for which pattern data gets inverted

READER_ERROR_COUNT

Address: 0xf0003000 + 0x28 = 0xf0003028

Number of errors detected

READER_SKIP_FIFO

Address: 0xf0003000 + 0x2c = 0xf000302c

Skip waiting for user to read the errors FIFO

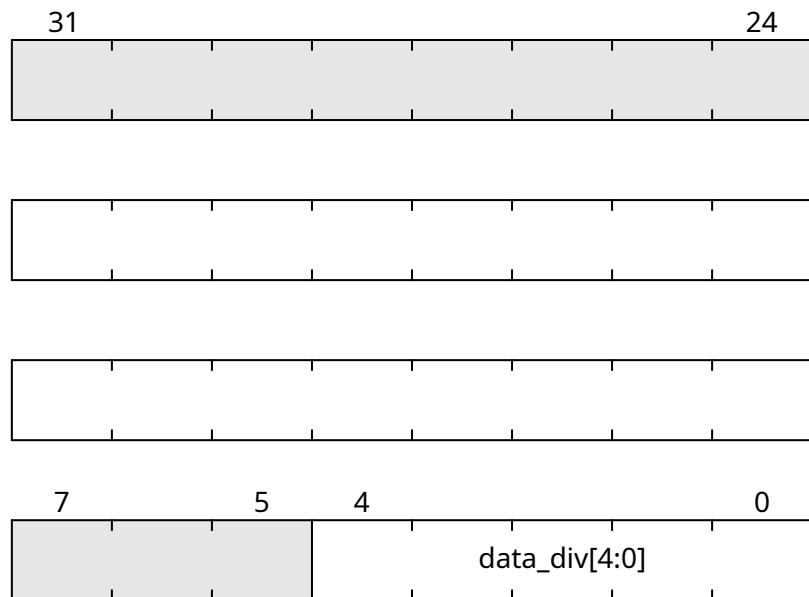


Fig. 22.48: READER_DATA_DIV

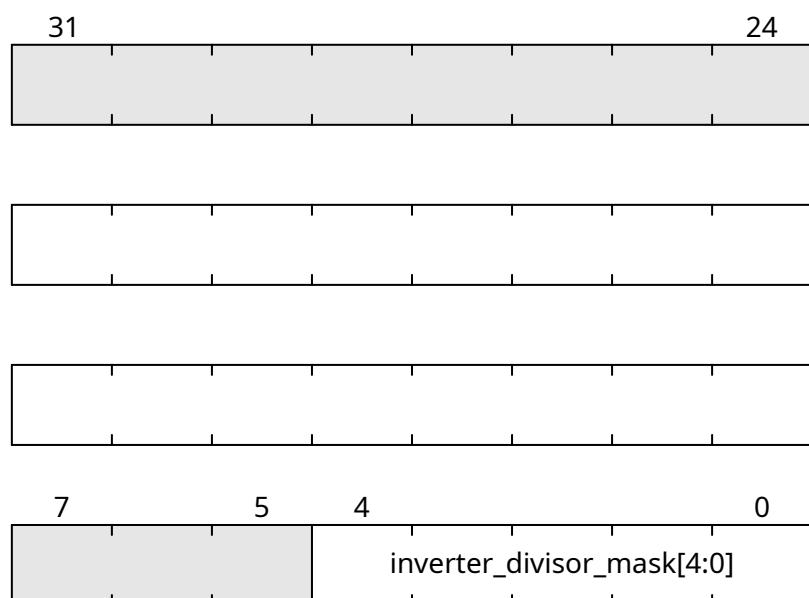


Fig. 22.49: READER_INVERTER_DIVISOR_MASK

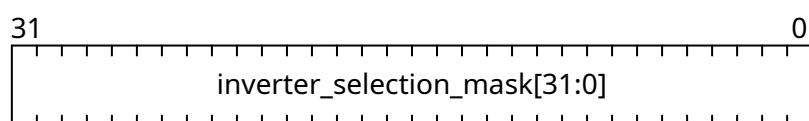


Fig. 22.50: READER_INVERTER_SELECTION_MASK

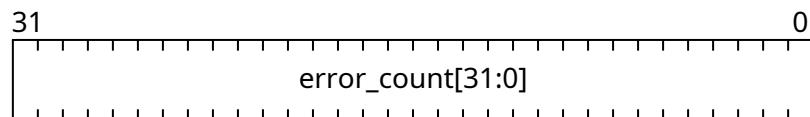


Fig. 22.51: READER_ERROR_COUNT



Fig. 22.52: READER_SKIP_FIFO

READER_ERROR_OFFSET

Address: $0xf0003000 + 0x30 = 0xf0003030$

Current offset of the error

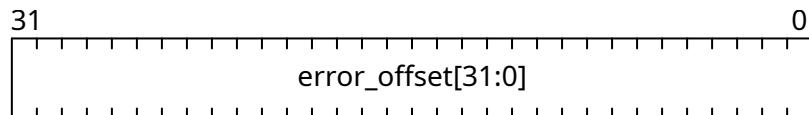


Fig. 22.53: READER_ERROR_OFFSET

READER_ERROR_DATA7

Address: $0xf0003000 + 0x34 = 0xf0003034$

Bits 224-255 of READER_ERROR_DATA. Erroneous value read from DRAM memory

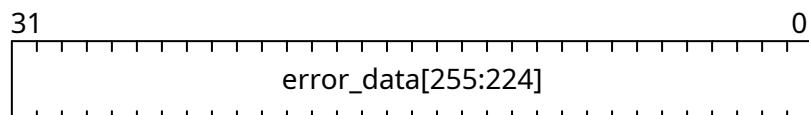


Fig. 22.54: READER_ERROR_DATA7

READER_ERROR_DATA6

Address: $0xf0003000 + 0x38 = 0xf0003038$

Bits 192-223 of READER_ERROR_DATA.

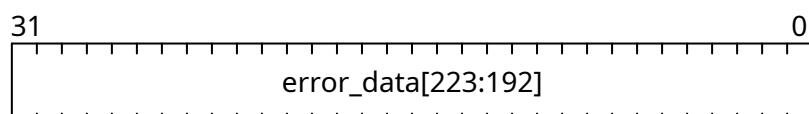


Fig. 22.55: READER_ERROR_DATA6

READER_ERROR_DATA5

Address: $0xf0003000 + 0x3c = 0xf000303c$

Bits 160-191 of READER_ERROR_DATA.

READER_ERROR_DATA4

Address: $0xf0003000 + 0x40 = 0xf0003040$

Bits 128-159 of READER_ERROR_DATA.

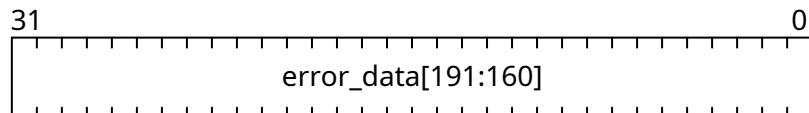


Fig. 22.56: READER_ERROR_DATA5

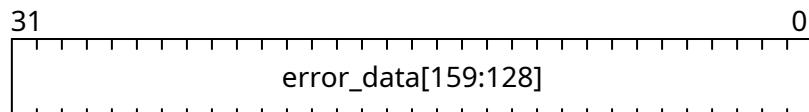


Fig. 22.57: READER_ERROR_DATA4

READER_ERROR_DATA3

Address: 0xf0003000 + 0x44 = 0xf0003044

Bits 96-127 of *READER_ERROR_DATA*.

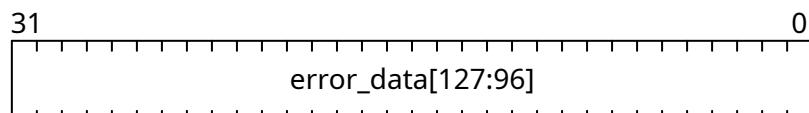


Fig. 22.58: READER_ERROR_DATA3

READER_ERROR_DATA2

Address: 0xf0003000 + 0x48 = 0xf0003048

Bits 64-95 of *READER_ERROR_DATA*.

READER_ERROR_DATA1

Address: 0xf0003000 + 0x4c = 0xf000304c

Bits 32-63 of *READER_ERROR_DATA*.

READER_ERROR_DATA0

Address: 0xf0003000 + 0x50 = 0xf0003050

Bits 0-31 of *READER_ERROR_DATA*.

READER_ERROR_EXPECTED7

Address: 0xf0003000 + 0x54 = 0xf0003054

Bits 224-255 of *READER_ERROR_EXPECTED*. Value expected to be read from DRAM memory

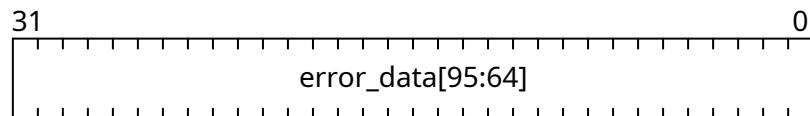


Fig. 22.59: READER_ERROR_DATA2

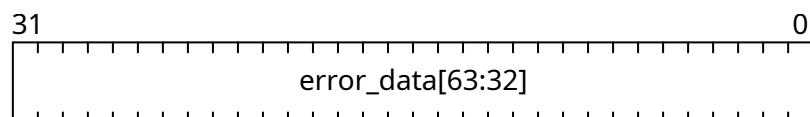


Fig. 22.60: READER_ERROR_DATA1

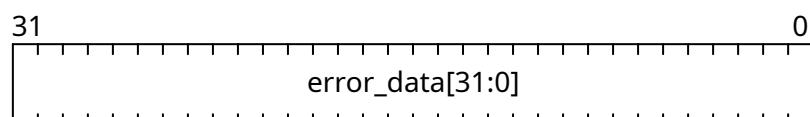


Fig. 22.61: READER_ERROR_DATA0

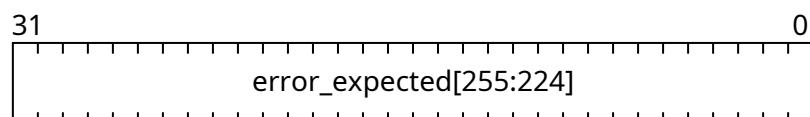


Fig. 22.62: READER_ERROR_EXPECTED7

READER_ERROR_EXPECTED6

Address: $0xf0003000 + 0x58 = 0xf0003058$

Bits 192-223 of *READER_ERROR_EXPECTED*.

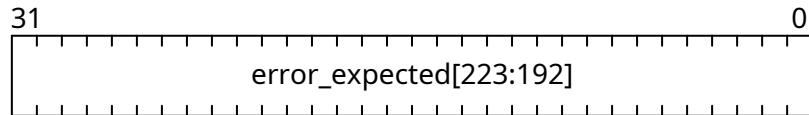


Fig. 22.63: READER_ERROR_EXPECTED6

READER_ERROR_EXPECTED5

Address: $0xf0003000 + 0x5c = 0xf000305c$

Bits 160-191 of *READER_ERROR_EXPECTED*.

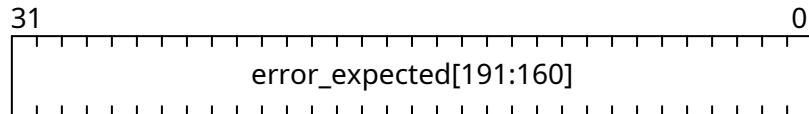


Fig. 22.64: READER_ERROR_EXPECTED5

READER_ERROR_EXPECTED4

Address: $0xf0003000 + 0x60 = 0xf0003060$

Bits 128-159 of *READER_ERROR_EXPECTED*.

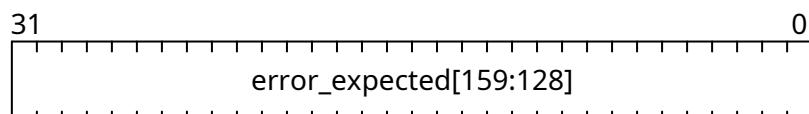


Fig. 22.65: READER_ERROR_EXPECTED4

READER_ERROR_EXPECTED3

Address: $0xf0003000 + 0x64 = 0xf0003064$

Bits 96-127 of *READER_ERROR_EXPECTED*.

READER_ERROR_EXPECTED2

Address: $0xf0003000 + 0x68 = 0xf0003068$

Bits 64-95 of *READER_ERROR_EXPECTED*.

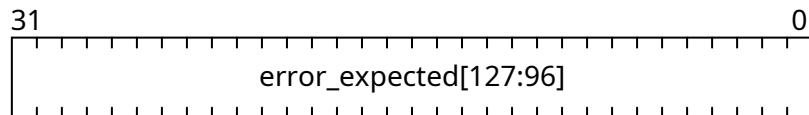


Fig. 22.66: READER_ERROR_EXPECTED3

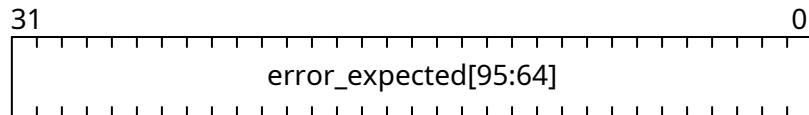


Fig. 22.67: READER_ERROR_EXPECTED2

READER_ERROR_EXPECTED1

Address: $0xf0003000 + 0x6c = 0xf000306c$

Bits 32-63 of *READER_ERROR_EXPECTED*.

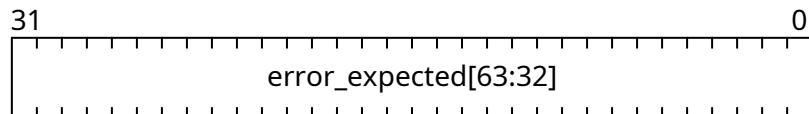


Fig. 22.68: READER_ERROR_EXPECTED1

READER_ERROR_EXPECTED0

Address: $0xf0003000 + 0x70 = 0xf0003070$

Bits 0-31 of *READER_ERROR_EXPECTED*.

READER_ERROR_READY

Address: $0xf0003000 + 0x74 = 0xf0003074$

Error detected and ready to read

READER_ERROR_CONTINUE

Address: $0xf0003000 + 0x78 = 0xf0003078$

Continue reading until the next error

22.2.8 DFI_SWITCH

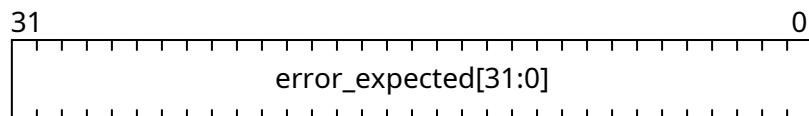


Fig. 22.69: READER_ERROR_EXPECTED0

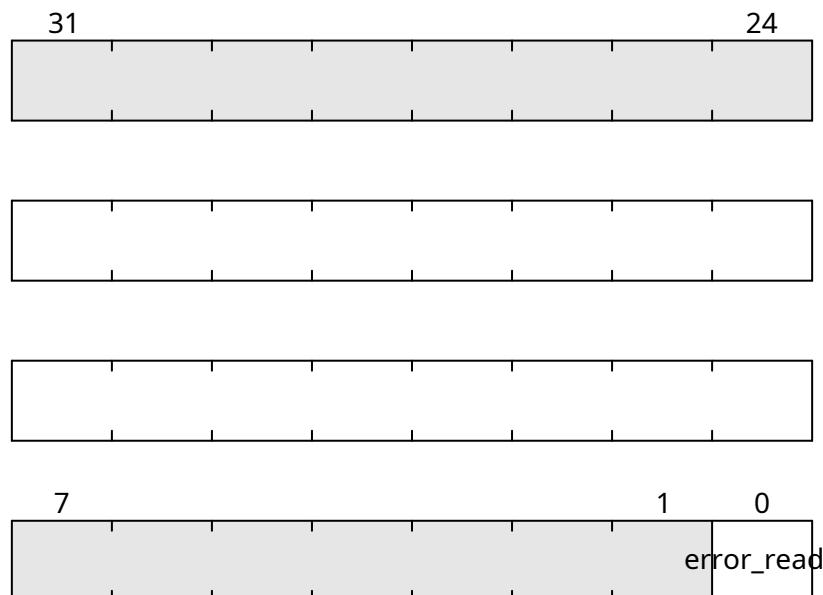


Fig. 22.70: READER_ERROR_READY

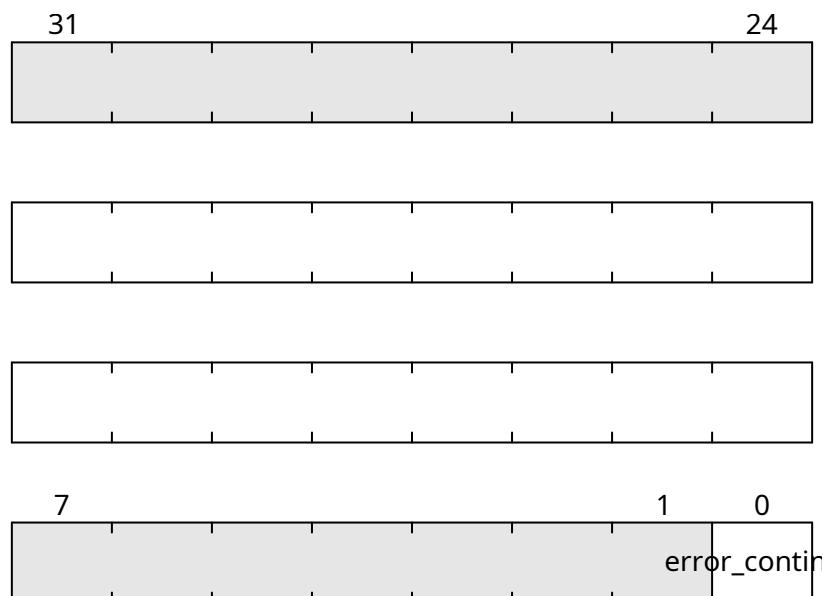


Fig. 22.71: READER_ERROR_CONTINUE

Register Listing for DFI_SWITCH

Register	Address
<i>DFI_SWITCH_REFRESH_COUNT1</i>	0xf0003800
<i>DFI_SWITCH_REFRESH_COUNT0</i>	0xf0003804
<i>DFI_SWITCH_AT_REFRESH1</i>	0xf0003808
<i>DFI_SWITCH_AT_REFRESH0</i>	0xf000380c
<i>DFI_SWITCH_REFRESH_UPDATE</i>	0xf0003810

DFI_SWITCH_REFRESH_COUNT1

Address: $0xf0003800 + 0x0 = 0xf0003800$

Bits 32-63 of *DFI_SWITCH_REFRESH_COUNT*. Count of all refresh commands issued (both by Memory Controller and the Payload Executor). Value is latched from internal counter on mode transition: MC -> PE or by writing to the *refresh_update* CSR.

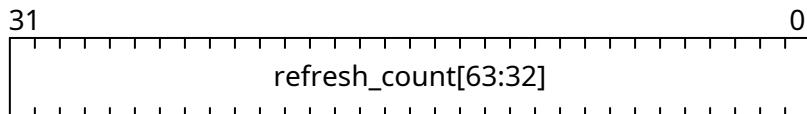


Fig. 22.72: DFI_SWITCH_REFRESH_COUNT1

DFI_SWITCH_REFRESH_COUNT0

Address: $0xf0003800 + 0x4 = 0xf0003804$

Bits 0-31 of *DFI_SWITCH_REFRESH_COUNT*.

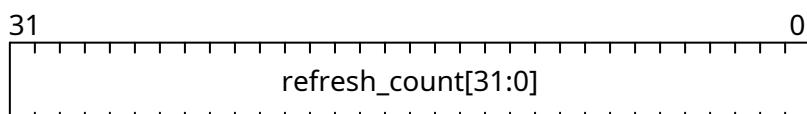


Fig. 22.73: DFI_SWITCH_REFRESH_COUNT0

DFI_SWITCH_AT_REFRESH1

Address: $0xf0003800 + 0x8 = 0xf0003808$

Bits 32-63 of *DFI_SWITCH_AT_REFRESH*. If set to a value different than 0 the mode transition MC -> PE will be performed only when the value of this register matches the current refresh commands count.

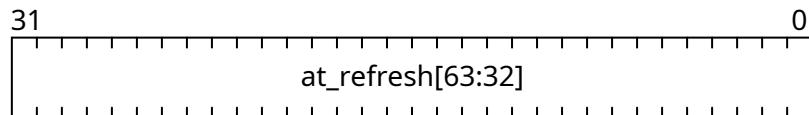


Fig. 22.74: DFI_SWITCH_AT_REFRESH1

DFI_SWITCH_AT_REFRESH0

Address: $0xf0003800 + 0xc = 0xf000380c$

Bits 0-31 of *DFI_SWITCH_AT_REFRESH*.

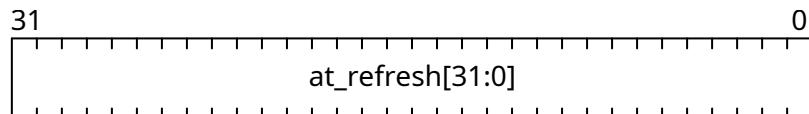


Fig. 22.75: DFI_SWITCH_AT_REFRESH0

DFI_SWITCH_REFRESH_UPDATE

Address: $0xf0003800 + 0x10 = 0xf0003810$

Force an update of the *refresh_count* CSR.

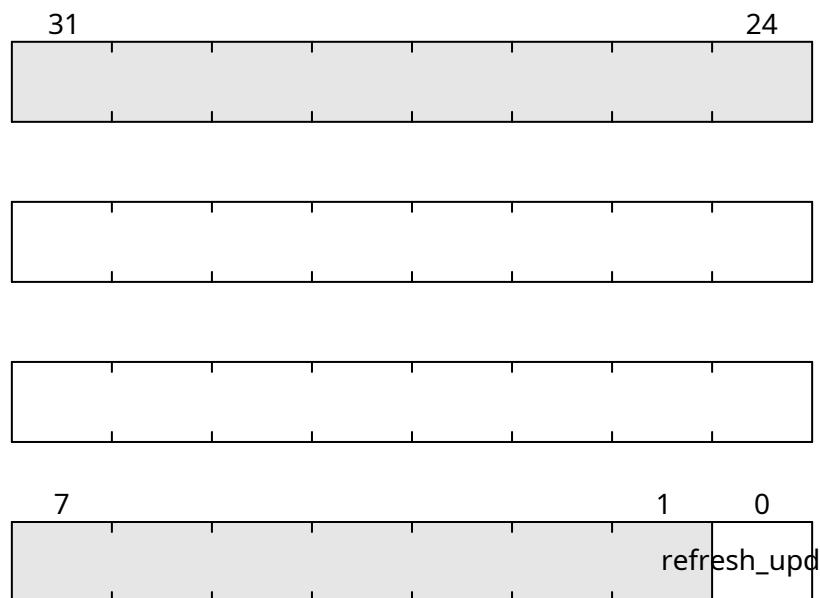


Fig. 22.76: DFI_SWITCH_REFRESH_UPDATE

22.2.9 PAYLOAD_EXECUTOR

Executes the DRAM payload from memory

Instruction decoder

All instructions are 32-bit. The format of most instructions is the same, except for the LOOP instruction, which has a constant TIMESLICE of 1.

NOOP with a TIMESLICE of 0 is a special case which is interpreted as STOP instruction. When this instruction is encountered execution gets finished immediately.

NOTE: TIMESLICE is the number of cycles the instruction will take. This means that instructions other than NOOP that use TIMESLICE=0 are illegal (although will silently be executed as having TIMESLICE=1).

NOTE2: LOOP instruction will *jump* COUNT times, meaning that the “code” inside the loop will effectively be executed COUNT+1 times.

Op codes:

Op	Value
NOOP	0b000
LOOP	0b111
ACT	0b100
PRE	0b101
REF	0b110
ZQC	0b001
READ	0b010

Instruction format:

LSB	MSB
dfi:	OP_CODE TIMESLICE ADDRESS
noop:	OP_CODE TIMESLICE_NOOP
loop:	OP_CODE COUNT JUMP
stop:	<NOOP> 0

Where ADDRESS depends on the DFI command and is one of:

LSB	MSB
RANK BANK COLUMN	
RANK BANK ROW	

Register Listing for PAYLOAD_EXECUTOR

Register	Address
PAYLOAD_EXECUTOR_START	0xf0004000
PAYLOAD_EXECUTOR_STATUS	0xf0004004
PAYLOAD_EXECUTOR_READ_COUNT	0xf0004008
PAYLOAD_EXECUTOR_EXEC_START1	0xf000400c
PAYLOAD_EXECUTOR_EXEC_START0	0xf0004010
PAYLOAD_EXECUTOR_EXEC_STOP1	0xf0004014
PAYLOAD_EXECUTOR_EXEC_STOP0	0xf0004018

PAYLOAD_EXECUTOR_START

Address: $0xf0004000 + 0x0 = 0xf0004000$

Writing to this register initializes payload execution



Fig. 22.77: PAYLOAD_EXECUTOR_START

PAYLOAD_EXECUTOR_STATUS

Address: $0xf0004000 + 0x4 = 0xf0004004$

Payload executor status register

Field	Name	Description
[0]	READY	Indicates that the executor is not running
[1]	OVERFLOW	Indicates the scratchpad memory address counter has overflowed due to the number of READ commands sent during execution

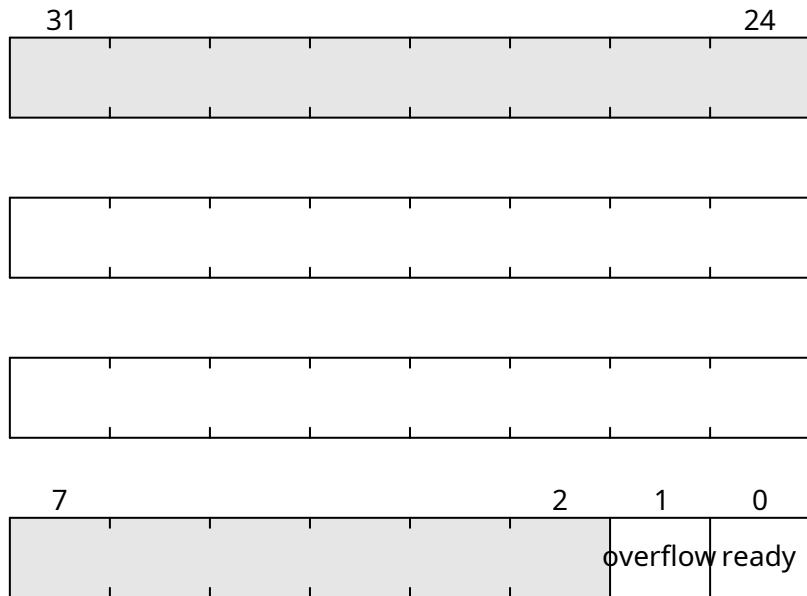


Fig. 22.78: PAYLOAD_EXECUTOR_STATUS

PAYLOAD_EXECUTOR_READ_COUNT

Address: $0xf0004000 + 0x8 = 0xf0004008$

Number of data from READ commands that is stored in the scratchpad memory

PAYLOAD_EXECUTOR_EXEC_START1

Address: $0xf0004000 + 0xc = 0xf000400c$

Bits 32-63 of PAYLOAD_EXECUTOR_EXEC_START. Number of cycles elapsed until the start of the payload execution.

PAYLOAD_EXECUTOR_EXEC_START0

Address: $0xf0004000 + 0x10 = 0xf0004010$

Bits 0-31 of PAYLOAD_EXECUTOR_EXEC_START.

PAYLOAD_EXECUTOR_EXEC_STOP1

Address: $0xf0004000 + 0x14 = 0xf0004014$

Bits 32-63 of PAYLOAD_EXECUTOR_EXEC_STOP. Number of cycles elapsed until the end of the payload execution.

PAYLOAD_EXECUTOR_EXEC_STOP0

Address: $0xf0004000 + 0x18 = 0xf0004018$

Bits 0-31 of PAYLOAD_EXECUTOR_EXEC_STOP.

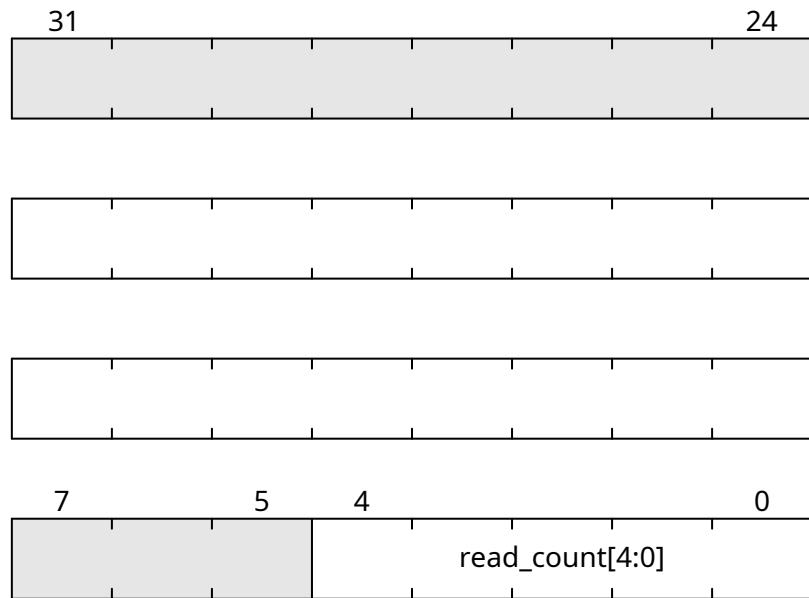


Fig. 22.79: PAYLOAD_EXECUTOR_READ_COUNT

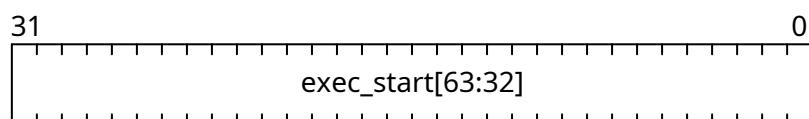


Fig. 22.80: PAYLOAD_EXECUTOR_EXEC_START1

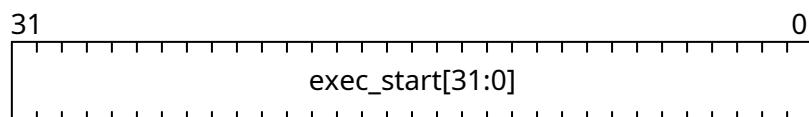


Fig. 22.81: PAYLOAD_EXECUTOR_EXEC_START0

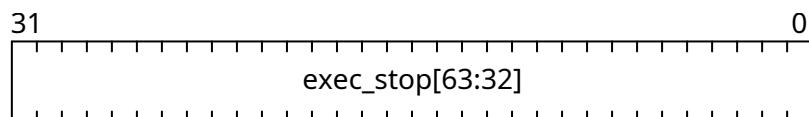


Fig. 22.82: PAYLOAD_EXECUTOR_EXEC_STOP1

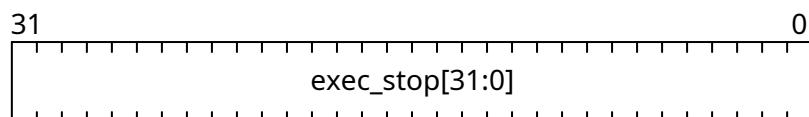


Fig. 22.83: PAYLOAD_EXECUTOR_EXEC_STOP0

22.2.10 CTRL

Register Listing for CTRL

Register	Address
<i>CTRL_RESET</i>	0xf0004800
<i>CTRL_SCRATCH</i>	0xf0004804
<i>CTRL_BUS_ERRORS</i>	0xf0004808

CTRL__RESET

Address: $0xf0004800 + 0x0 = 0xf0004800$

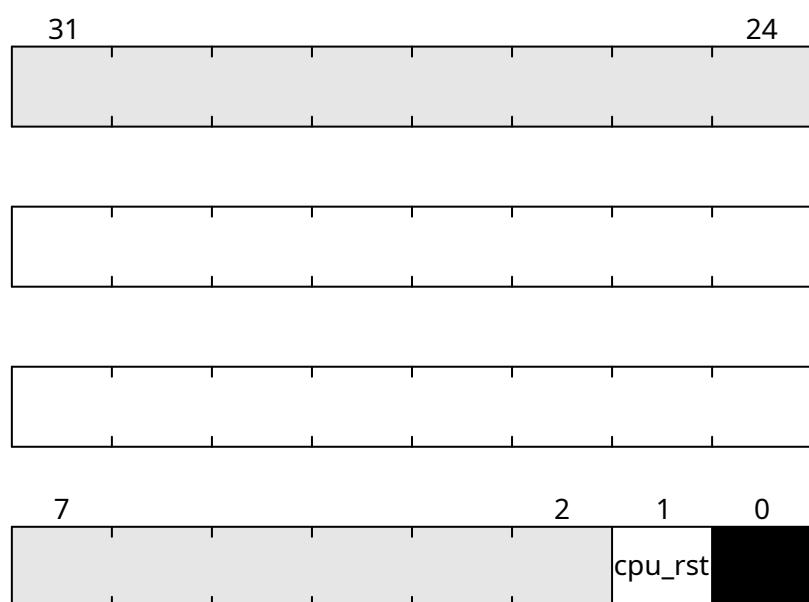


Fig. 22.84: CTRL__RESET

Field	Name	Description
[0]	SOC_RST	Write 1 to this register to reset the full SoC (Pulse Reset)
[1]	CPU_RST	Write 1 to this register to reset the CPU(s) of the SoC (Hold Reset)

CTRL_SCRATCH

Address: $0xf0004800 + 0x4 = 0xf0004804$

Use this register as a scratch space to verify that software read/write accesses to the Wishbone/CSR bus are working correctly. The initial reset value of 0x1234578 can be used to verify endianness.

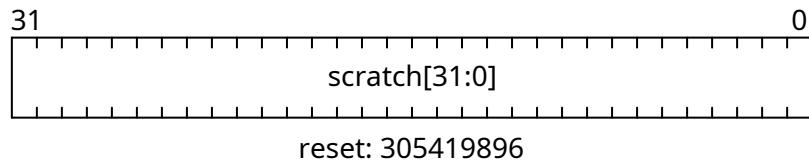


Fig. 22.85: CTRL_SCRATCH

CTRL_BUS_ERRORS

Address: $0xf0004800 + 0x8 = 0xf0004808$

Total number of Wishbone bus errors (timeouts) since start.

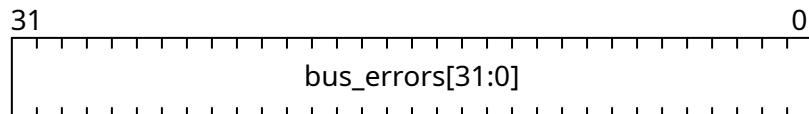


Fig. 22.86: CTRL_BUS_ERRORS

22.2.11 ETPHY

Register Listing for ETPHY

Register	Address
<i>ETPHY_CRG_RESET</i>	<i>0xf0005000</i>
<i>ETPHY_MDIO_W</i>	<i>0xf0005004</i>
<i>ETPHY_MDIO_R</i>	<i>0xf0005008</i>

ETPHY_CRG_RESET

Address: $0xf0005000 + 0x0 = 0xf0005000$

ETPHY_MDIO_W

Address: $0xf0005000 + 0x4 = 0xf0005004$

Field	Name	Description

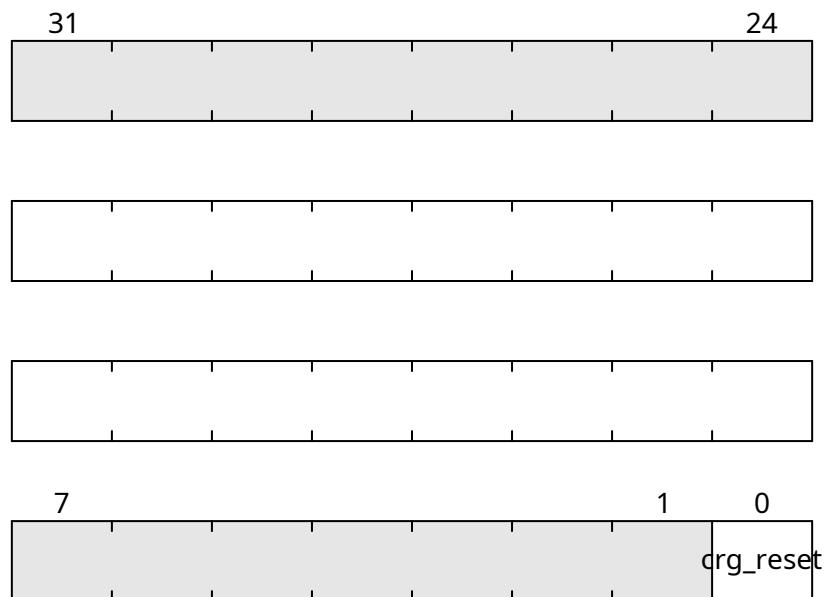


Fig. 22.87: ETHPHY_CRG_RESET

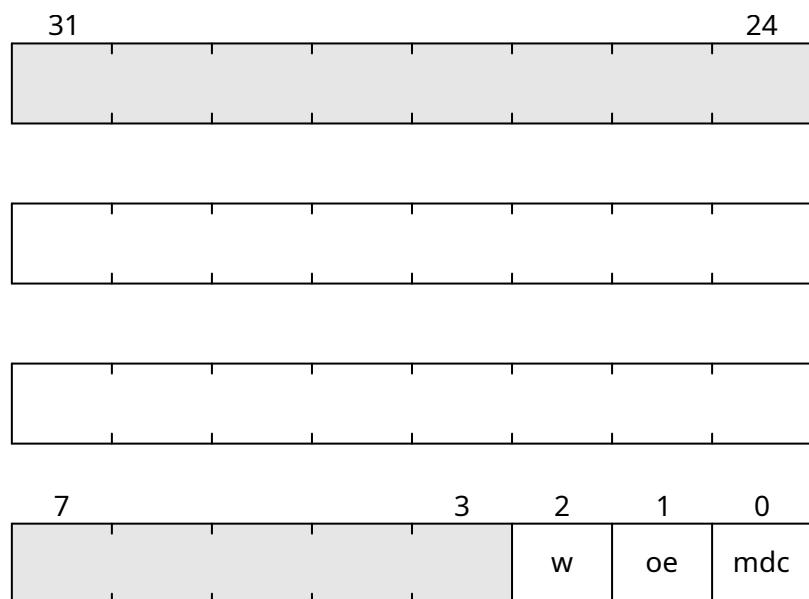


Fig. 22.88: ETHPHY_MDIO_W

ETHPHY_MDIO_R

Address: $0xf0005000 + 0x8 = 0xf0005008$

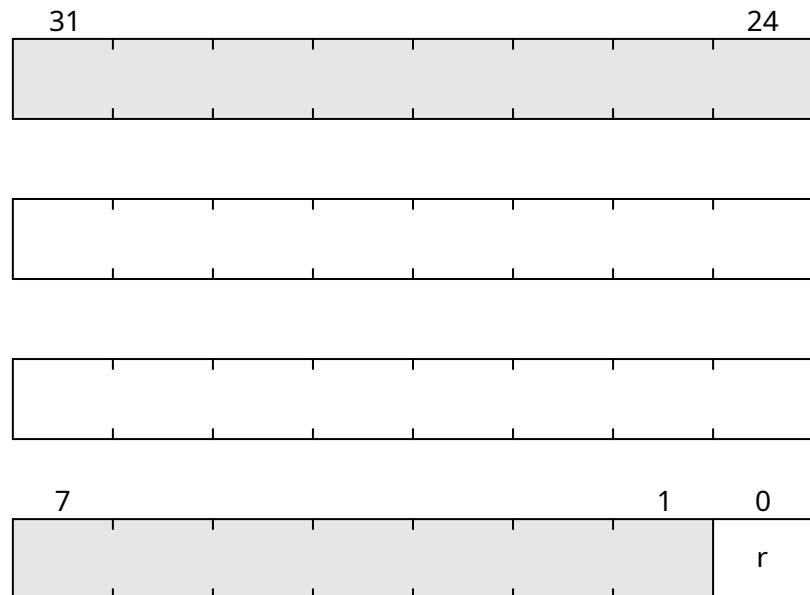


Fig. 22.89: ETHPHY_MDIO_R

Field	Name	Description

22.2.12 IDENTIFIER_MEM

Register Listing for IDENTIFIER_MEM

Register	Address
<i>IDENTIFIER_MEM</i>	<i>0xf0005800</i>

IDENTIFIER_MEM

Address: $0xf0005800 + 0x0 = 0xf0005800$

8 x 108-bit memory

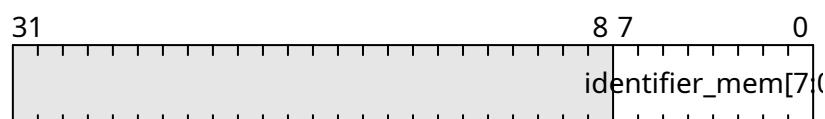


Fig. 22.90: IDENTIFIER_MEM

22.2.13 SDRAM

Register Listing for SDRAM

Register	Address
<i>SDRAM_DFII_CONTROL</i>	0xf0006000
<i>SDRAM_DFII_PIO_COMMAND</i>	0xf0006004
<i>SDRAM_DFII_PIO_COMMAND_ISSUE</i>	0xf0006008
<i>SDRAM_DFII_PIO_ADDRESS</i>	0xf000600c
<i>SDRAM_DFII_PIO_BADDRESS</i>	0xf0006010
<i>SDRAM_DFII_PIO_WRDATA</i>	0xf0006014
<i>SDRAM_DFII_PIO_RDDATA</i>	0xf0006018
<i>SDRAM_DFII_PI1_COMMAND</i>	0xf000601c
<i>SDRAM_DFII_PI1_COMMAND_ISSUE</i>	0xf0006020
<i>SDRAM_DFII_PI1_ADDRESS</i>	0xf0006024
<i>SDRAM_DFII_PI1_BADDRESS</i>	0xf0006028
<i>SDRAM_DFII_PI1_WRDATA</i>	0xf000602c
<i>SDRAM_DFII_PI1_RDDATA</i>	0xf0006030
<i>SDRAM_DFII_PI2_COMMAND</i>	0xf0006034
<i>SDRAM_DFII_PI2_COMMAND_ISSUE</i>	0xf0006038
<i>SDRAM_DFII_PI2_ADDRESS</i>	0xf000603c
<i>SDRAM_DFII_PI2_BADDRESS</i>	0xf0006040
<i>SDRAM_DFII_PI2_WRDATA</i>	0xf0006044
<i>SDRAM_DFII_PI2_RDDATA</i>	0xf0006048
<i>SDRAM_DFII_PI3_COMMAND</i>	0xf000604c
<i>SDRAM_DFII_PI3_COMMAND_ISSUE</i>	0xf0006050
<i>SDRAM_DFII_PI3_ADDRESS</i>	0xf0006054
<i>SDRAM_DFII_PI3_BADDRESS</i>	0xf0006058
<i>SDRAM_DFII_PI3_WRDATA</i>	0xf000605c
<i>SDRAM_DFII_PI3_RDDATA</i>	0xf0006060
<i>SDRAM_DFII_PI4_COMMAND</i>	0xf0006064
<i>SDRAM_DFII_PI4_COMMAND_ISSUE</i>	0xf0006068
<i>SDRAM_DFII_PI4_ADDRESS</i>	0xf000606c
<i>SDRAM_DFII_PI4_BADDRESS</i>	0xf0006070
<i>SDRAM_DFII_PI4_WRDATA</i>	0xf0006074
<i>SDRAM_DFII_PI4_RDDATA</i>	0xf0006078
<i>SDRAM_DFII_PI5_COMMAND</i>	0xf000607c
<i>SDRAM_DFII_PI5_COMMAND_ISSUE</i>	0xf0006080
<i>SDRAM_DFII_PI5_ADDRESS</i>	0xf0006084
<i>SDRAM_DFII_PI5_BADDRESS</i>	0xf0006088
<i>SDRAM_DFII_PI5_WRDATA</i>	0xf000608c
<i>SDRAM_DFII_PI5_RDDATA</i>	0xf0006090
<i>SDRAM_DFII_PI6_COMMAND</i>	0xf0006094
<i>SDRAM_DFII_PI6_COMMAND_ISSUE</i>	0xf0006098
<i>SDRAM_DFII_PI6_ADDRESS</i>	0xf000609c
<i>SDRAM_DFII_PI6_BADDRESS</i>	0xf00060a0
<i>SDRAM_DFII_PI6_WRDATA</i>	0xf00060a4
<i>SDRAM_DFII_PI6_RDDATA</i>	0xf00060a8
<i>SDRAM_DFII_PI7_COMMAND</i>	0xf00060ac
<i>SDRAM_DFII_PI7_COMMAND_ISSUE</i>	0xf00060b0

continues on next page

Table 22.2 – continued from previous page

Register	Address
<i>SDRAM_DFII_PI7_ADDRESS</i>	0xf00060b4
<i>SDRAM_DFII_PI7_BADDRESS</i>	0xf00060b8
<i>SDRAM_DFII_PI7_WRDATA</i>	0xf00060bc
<i>SDRAM_DFII_PI7_RDDATA</i>	0xf00060c0
<i>SDRAM_CONTROLLER_TRP</i>	0xf00060c4
<i>SDRAM_CONTROLLER_TRCD</i>	0xf00060c8
<i>SDRAM_CONTROLLER_TWR</i>	0xf00060cc
<i>SDRAM_CONTROLLER_TWTR</i>	0xf00060d0
<i>SDRAM_CONTROLLER_TREFI</i>	0xf00060d4
<i>SDRAM_CONTROLLER_TRFC</i>	0xf00060d8
<i>SDRAM_CONTROLLER_TFAW</i>	0xf00060dc
<i>SDRAM_CONTROLLER_TCCD</i>	0xf00060e0
<i>SDRAM_CONTROLLER_TCCD_WR</i>	0xf00060e4
<i>SDRAM_CONTROLLER_TRTP</i>	0xf00060e8
<i>SDRAM_CONTROLLER_TRRD</i>	0xf00060ec
<i>SDRAM_CONTROLLER_TRC</i>	0xf00060f0
<i>SDRAM_CONTROLLER_TRAS</i>	0xf00060f4
<i>SDRAM_CONTROLLER_LAST_ADDR_0</i>	0xf00060f8
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0</i>	0xf00060fc
<i>SDRAM_CONTROLLER_LAST_ADDR_1</i>	0xf0006100
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1</i>	0xf0006104
<i>SDRAM_CONTROLLER_LAST_ADDR_2</i>	0xf0006108
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2</i>	0xf000610c
<i>SDRAM_CONTROLLER_LAST_ADDR_3</i>	0xf0006110
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3</i>	0xf0006114
<i>SDRAM_CONTROLLER_LAST_ADDR_4</i>	0xf0006118
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4</i>	0xf000611c
<i>SDRAM_CONTROLLER_LAST_ADDR_5</i>	0xf0006120
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5</i>	0xf0006124
<i>SDRAM_CONTROLLER_LAST_ADDR_6</i>	0xf0006128
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6</i>	0xf000612c
<i>SDRAM_CONTROLLER_LAST_ADDR_7</i>	0xf0006130
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7</i>	0xf0006134

SDRAM_DFII_CONTROL

Address: 0xf0006000 + 0x0 = 0xf0006000

Control DFI signals common to all phases

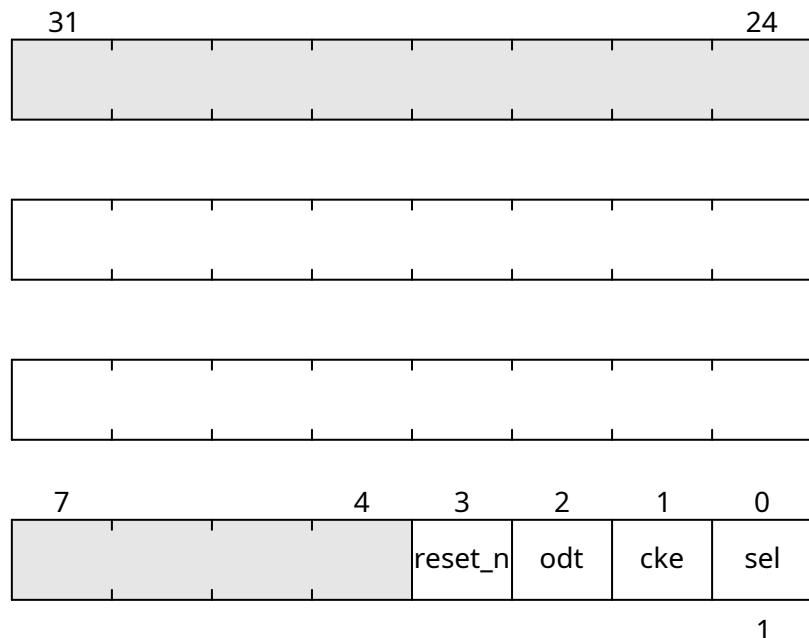


Fig. 22.91: SDRAM_DFII_CONTROL

Field	Name	Description						
[0]	SEL	<table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0b0</td><td>Software control. (CPU)</td></tr> <tr> <td>0b1</td><td>Hardware control (default).</td></tr> </tbody> </table>	Value	Description	0b0	Software control. (CPU)	0b1	Hardware control (default).
Value	Description							
0b0	Software control. (CPU)							
0b1	Hardware control (default).							
[1]	CKE	DFI clock enable bus						
[2]	ODT	DFI on-die termination bus						
[3]	RESET_N	DFI clock reset bus						

SDRAM_DFII_PIO_COMMAND

Address: $0xf0006000 + 0x4 = 0xf0006004$

Control DFI signals on a single phase

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

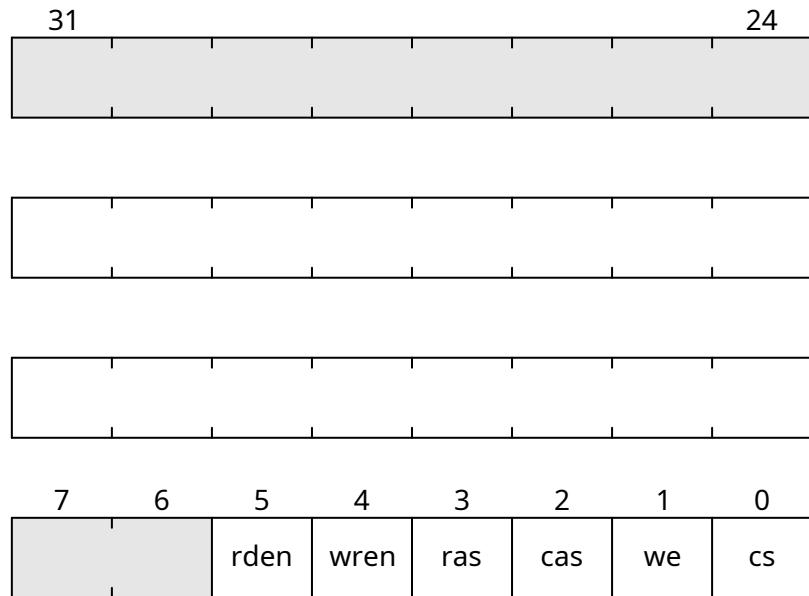


Fig. 22.92: SDRAM DFII PIO COMMAND

SDRAM_DFII_PIO_COMMAND_ISSUE

Address: $0xf0006000 + 0x8 = 0xf0006008$

SDRAM_DFIIPIO_ADDRESS

Address: $0xf0006000 + 0xc = 0xf000600c$

DFI address bus

SDRAM DFII PIO BADDRESS

Address: $0xf0006000 + 0x10 = 0xf0006010$

DFI bank address bus

SDRAM DFII PIO WRDATA

Address: $0xf0006000 + 0x14 = 0xf0006014$

DFI write data bus

SDRAM DFII PIO RDDATA

Address: $0xf0006000 + 0x18 \equiv 0xf0006018$

DFI read data bus

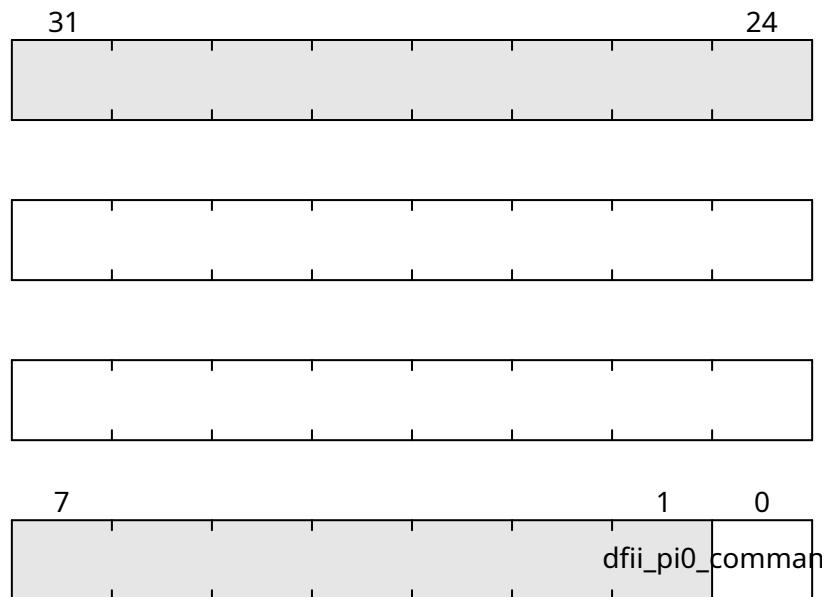


Fig. 22.93: SDRAM_DFII_PI0_COMMAND_ISSUE

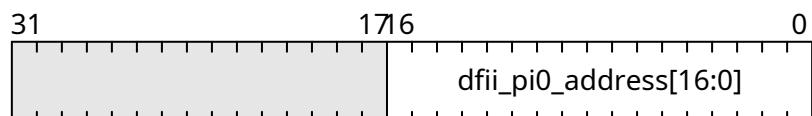


Fig. 22.94: SDRAM_DFII_PI0_ADDRESS

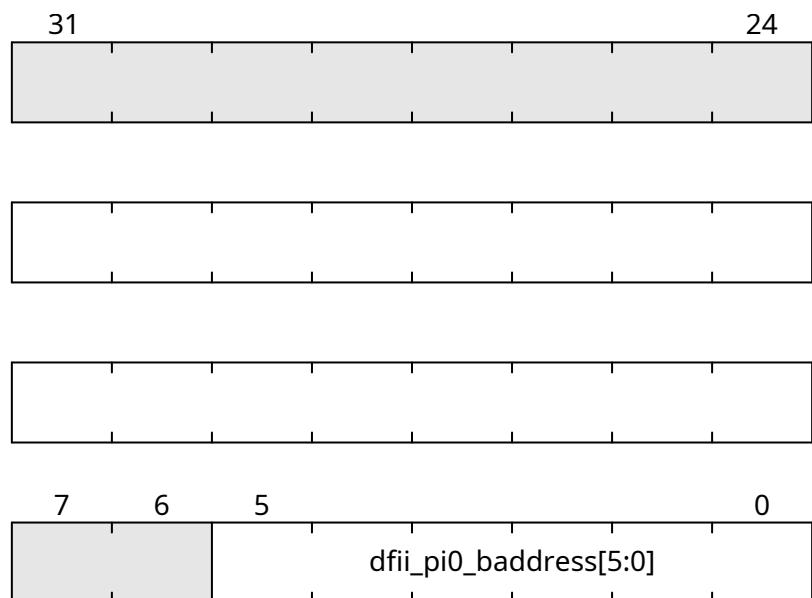


Fig. 22.95: SDRAM_DFII_PI0_BADDRESS

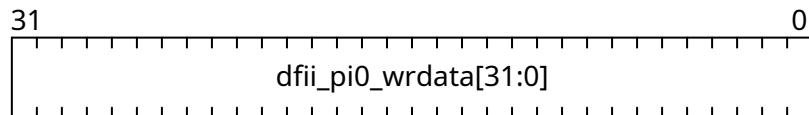


Fig. 22.96: SDRAM_DFII_PI0_WRDATA

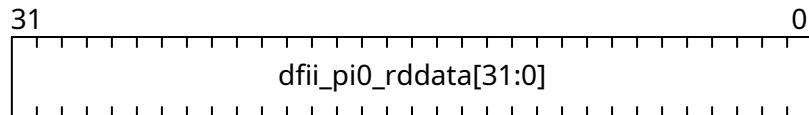


Fig. 22.97: SDRAM_DFII_PI0_RDDATA

SDRAM_DFII_PI1_COMMAND

Address: 0xf0006000 + 0x1c = 0xf000601c

Control DFI signals on a single phase

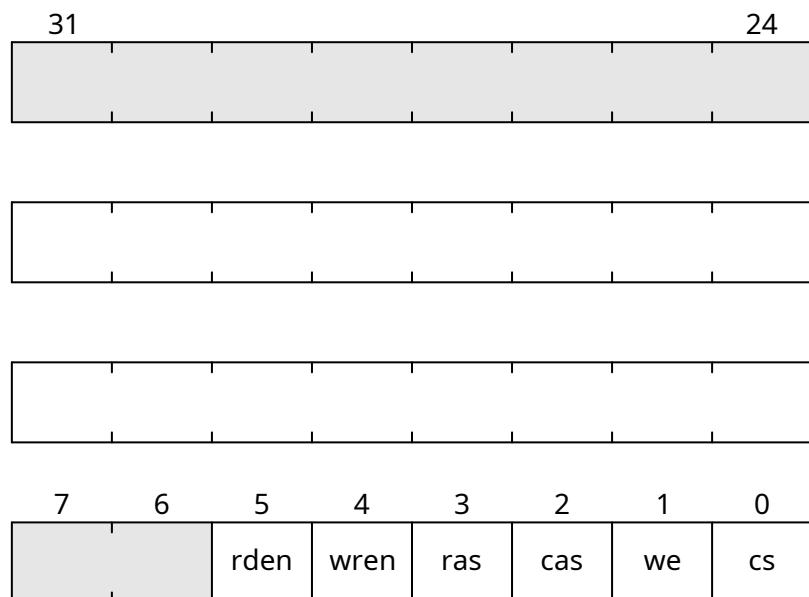


Fig. 22.98: SDRAM_DFII_PI1_COMMAND

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFII_PI1_COMMAND_ISSUE

Address: $0xf0006000 + 0x20 = 0xf0006020$

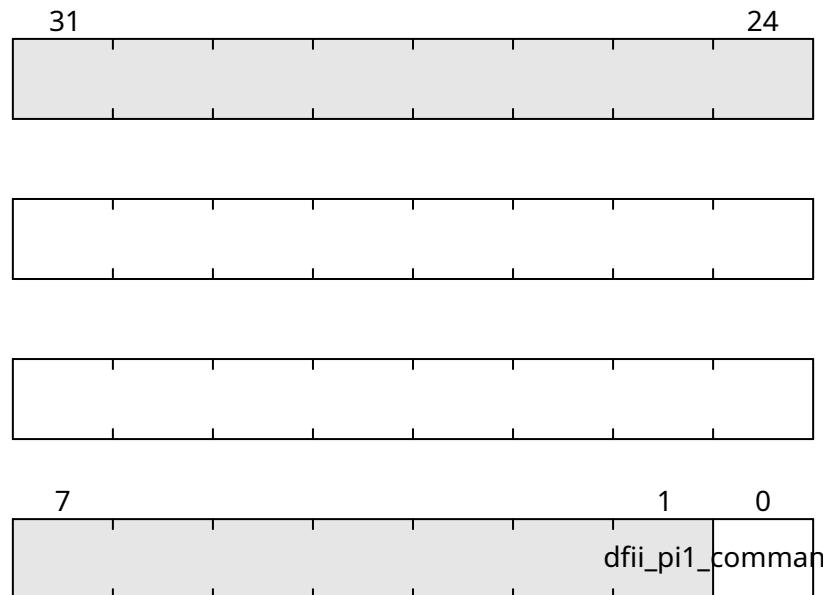


Fig. 22.99: SDRAM_DFII_PI1_COMMAND_ISSUE

SDRAM_DFII_PI1_ADDRESS

Address: $0xf0006000 + 0x24 = 0xf0006024$

DFI address bus

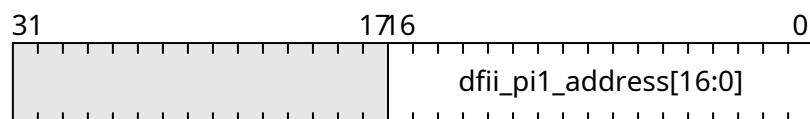


Fig. 22.100: SDRAM_DFII_PI1_ADDRESS

SDRAM_DFII_PI1_BADDRESS

Address: $0xf0006000 + 0x28 = 0xf0006028$

DFI bank address bus

SDRAM_DFII_PI1_WRDATA

Address: $0xf0006000 + 0x2c = 0xf000602c$

DFI write data bus

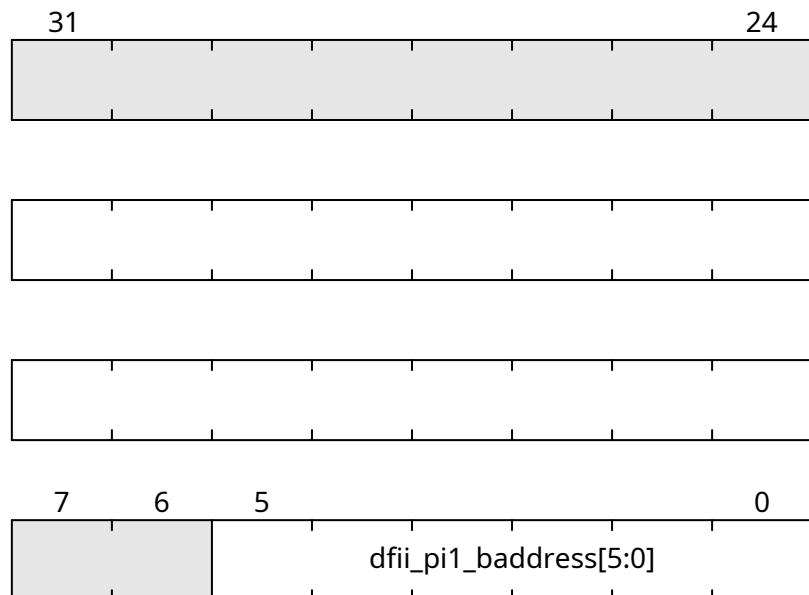


Fig. 22.101: SDRAM_DFII_PI1_BADDRESS

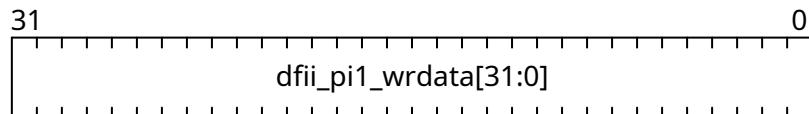


Fig. 22.102: SDRAM_DFII_PI1_WRDATA

SDRAM_DFII_PI1_RDDATA

Address: $0xf0006000 + 0x30 = 0xf0006030$

DFI read data bus

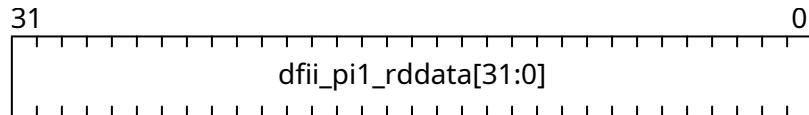


Fig. 22.103: SDRAM_DFII_PI1_RDDATA

SDRAM_DFII_PI2_COMMAND

Address: $0xf0006000 + 0x34 = 0xf0006034$

Control DFI signals on a single phase

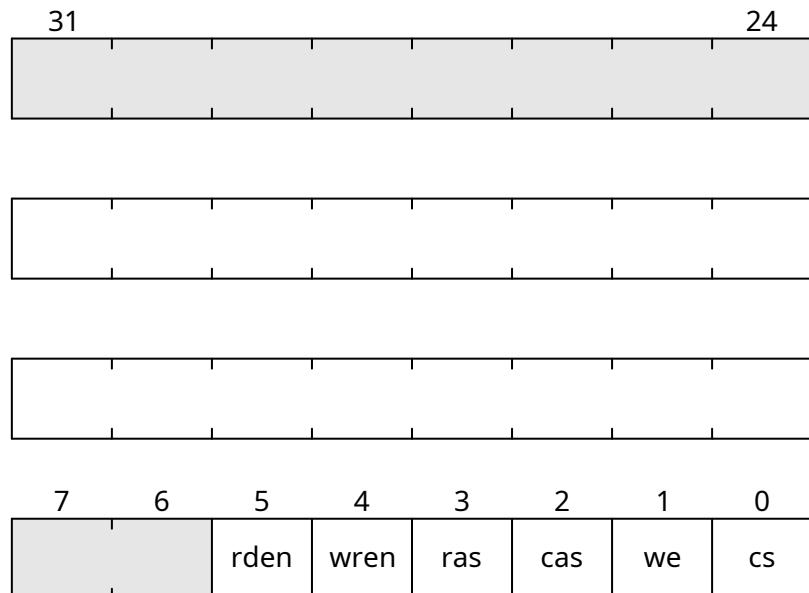


Fig. 22.104: SDRAM_DFII_PI2_COMMAND

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFII_PI2_COMMAND_ISSUE

Address: $0xf0006000 + 0x38 = 0xf0006038$

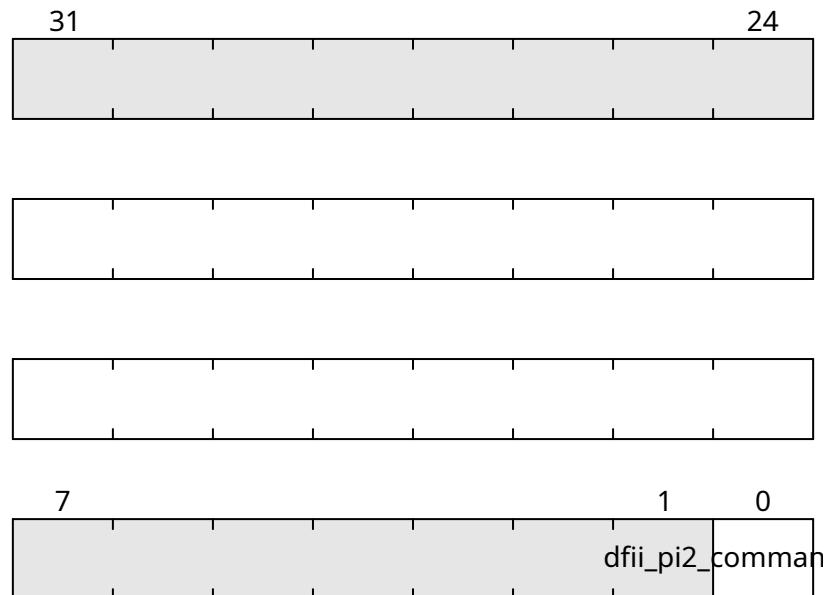


Fig. 22.105: SDRAM_DFII_PI2_COMMAND_ISSUE

SDRAM_DFII_PI2_ADDRESS

Address: $0xf0006000 + 0x3c = 0xf000603c$

DFI address bus

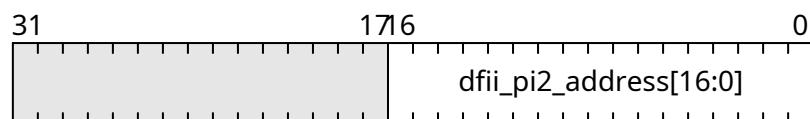


Fig. 22.106: SDRAM_DFII_PI2_ADDRESS

SDRAM_DFII_PI2_BADDRESS

Address: $0xf0006000 + 0x40 = 0xf0006040$

DFI bank address bus

SDRAM_DFII_PI2_WRDATA

Address: $0xf0006000 + 0x44 = 0xf0006044$

DFI write data bus

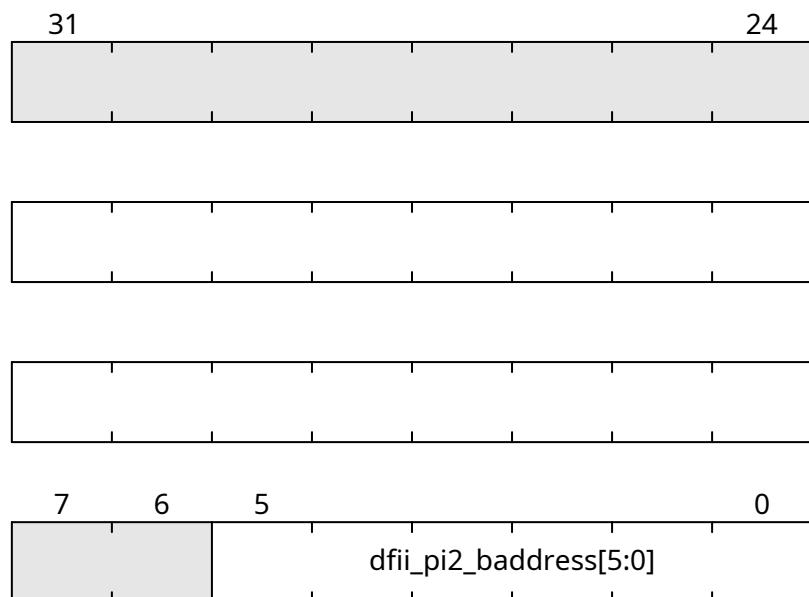


Fig. 22.107: SDRAM_DFII_PI2_BADDRESS

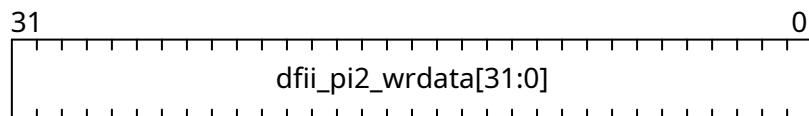


Fig. 22.108: SDRAM_DFII_PI2_WRDATA

SDRAM_DFII_PI2_RDDATA

Address: $0xf0006000 + 0x48 = 0xf0006048$

DFI read data bus

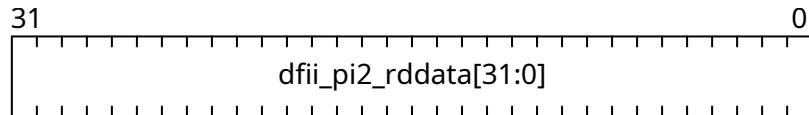


Fig. 22.109: SDRAM_DFII_PI2_RDDATA

SDRAM_DFII_PI3_COMMAND

Address: $0xf0006000 + 0x4c = 0xf000604c$

Control DFI signals on a single phase

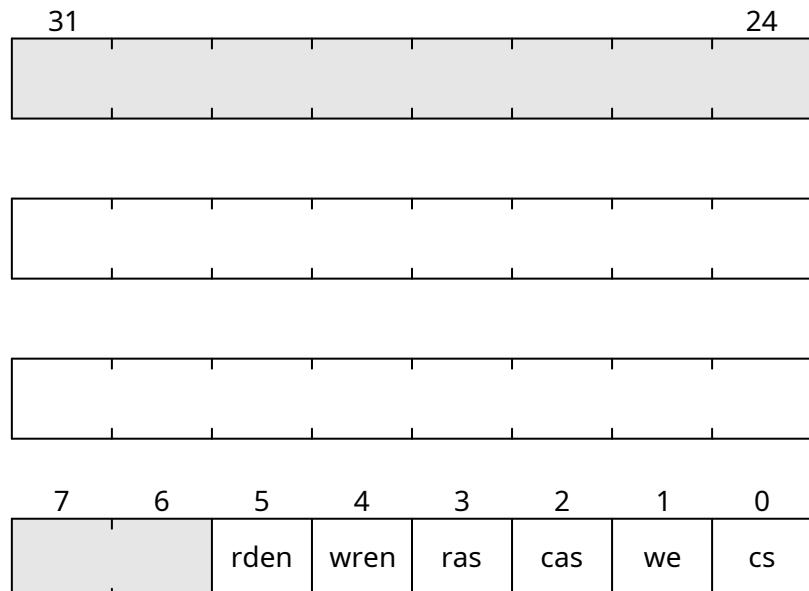


Fig. 22.110: SDRAM_DFII_PI3_COMMAND

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFII_PI3_COMMAND_ISSUE

Address: $0xf0006000 + 0x50 = 0xf0006050$

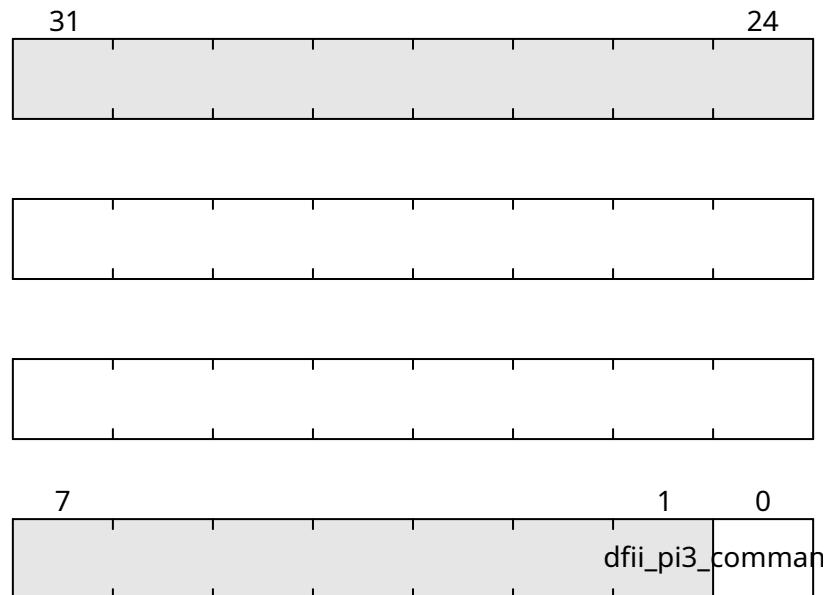


Fig. 22.111: SDRAM_DFII_PI3_COMMAND_ISSUE

SDRAM_DFII_PI3_ADDRESS

Address: $0xf0006000 + 0x54 = 0xf0006054$

DFI address bus

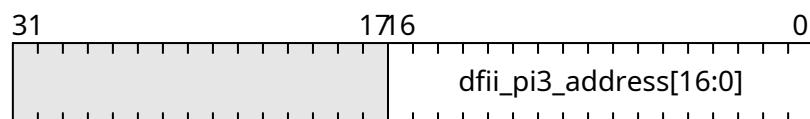


Fig. 22.112: SDRAM_DFII_PI3_ADDRESS

SDRAM_DFII_PI3_BADDRESS

Address: $0xf0006000 + 0x58 = 0xf0006058$

DFI bank address bus

SDRAM_DFII_PI3_WRDATA

Address: $0xf0006000 + 0x5c = 0xf000605c$

DFI write data bus

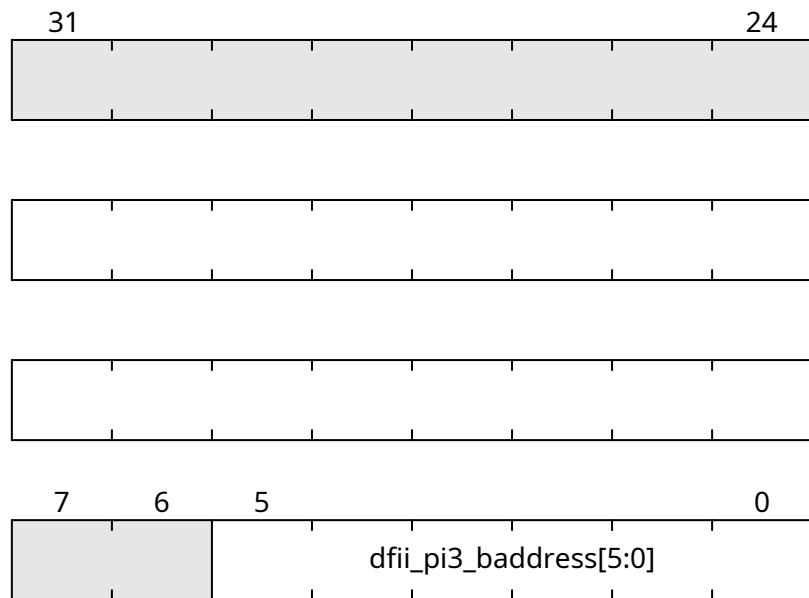


Fig. 22.113: SDRAM_DFII_PI3_BADDRESS

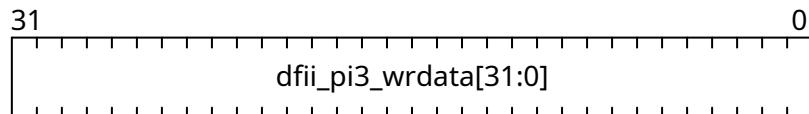


Fig. 22.114: SDRAM_DFII_PI3_WRDATA

SDRAM_DFII_PI3_RDDATA

Address: $0xf0006000 + 0x60 = 0xf0006060$

DFI read data bus

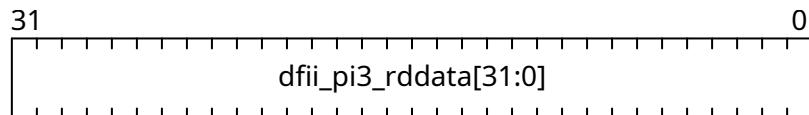


Fig. 22.115: SDRAM_DFII_PI3_RDDATA

SDRAM_DFII_PI4_COMMAND

Address: $0xf0006000 + 0x64 = 0xf0006064$

Control DFI signals on a single phase

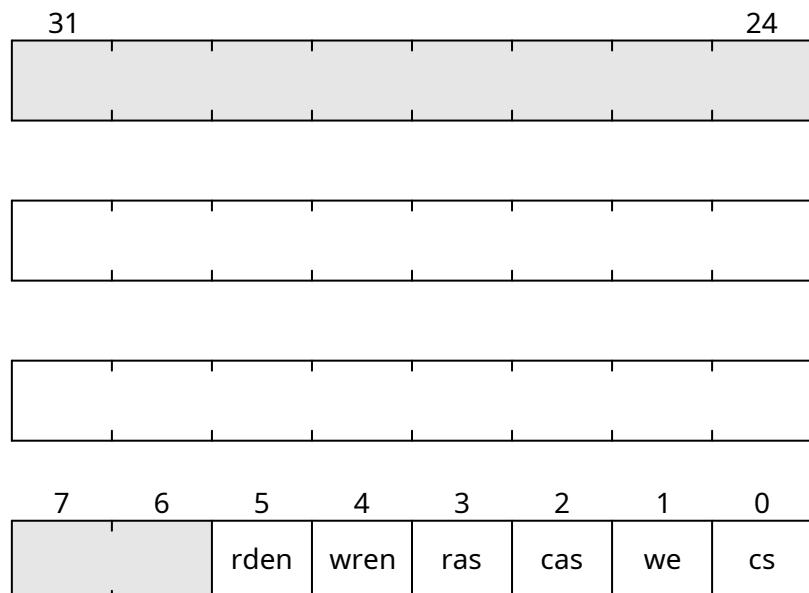


Fig. 22.116: SDRAM_DFII_PI4_COMMAND

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFII_PI4_COMMAND_ISSUE

Address: $0xf0006000 + 0x68 = 0xf0006068$

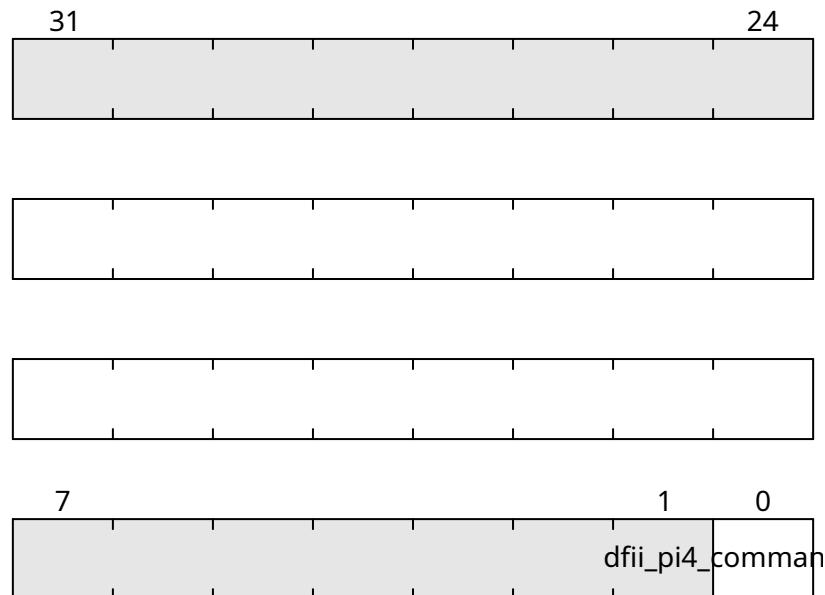


Fig. 22.117: SDRAM_DFII_PI4_COMMAND_ISSUE

SDRAM_DFII_PI4_ADDRESS

Address: $0xf0006000 + 0x6c = 0xf000606c$

DFI address bus

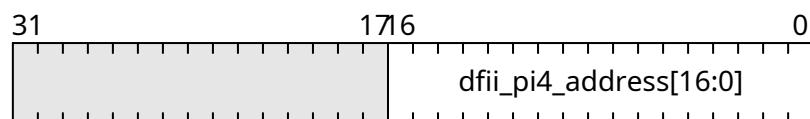


Fig. 22.118: SDRAM_DFII_PI4_ADDRESS

SDRAM_DFII_PI4_BADDRESS

Address: $0xf0006000 + 0x70 = 0xf0006070$

DFI bank address bus

SDRAM_DFII_PI4_WRDATA

Address: $0xf0006000 + 0x74 = 0xf0006074$

DFI write data bus

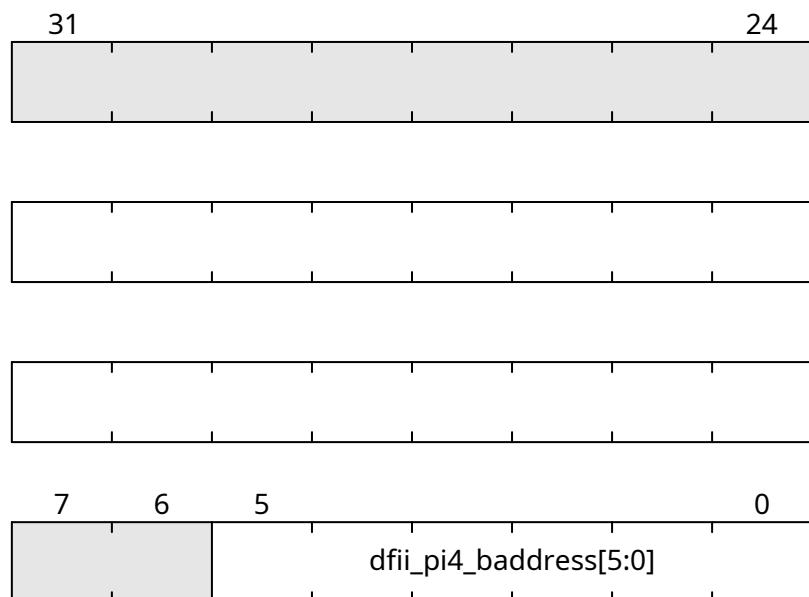


Fig. 22.119: SDRAM_DFII_PI4_BADDRESS

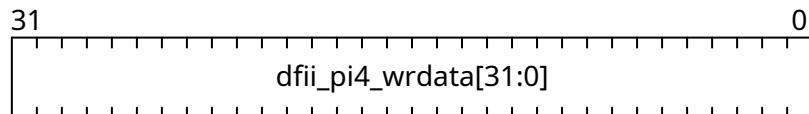


Fig. 22.120: SDRAM_DFII_PI4_WRDATA

SDRAM_DFII_PI4_RDDATA

Address: $0xf0006000 + 0x78 = 0xf0006078$

DFI read data bus

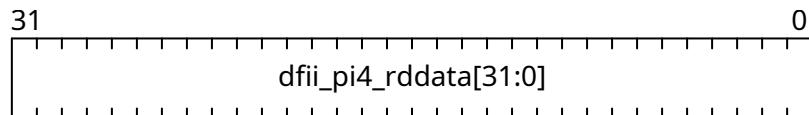


Fig. 22.121: SDRAM_DFII_PI4_RDDATA

SDRAM_DFII_PI5_COMMAND

Address: $0xf0006000 + 0x7c = 0xf000607c$

Control DFI signals on a single phase

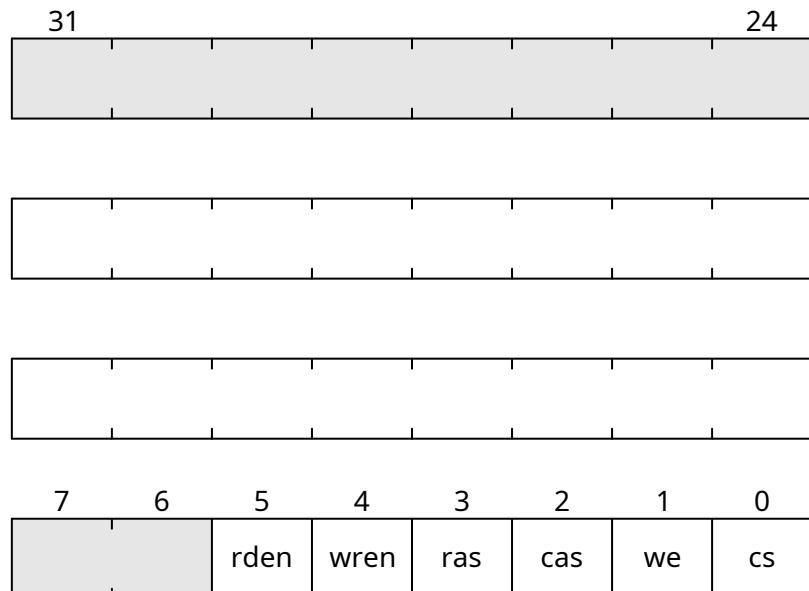


Fig. 22.122: SDRAM_DFII_PI5_COMMAND

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFII_PI5_COMMAND_ISSUE

Address: $0xf0006000 + 0x80 = 0xf0006080$

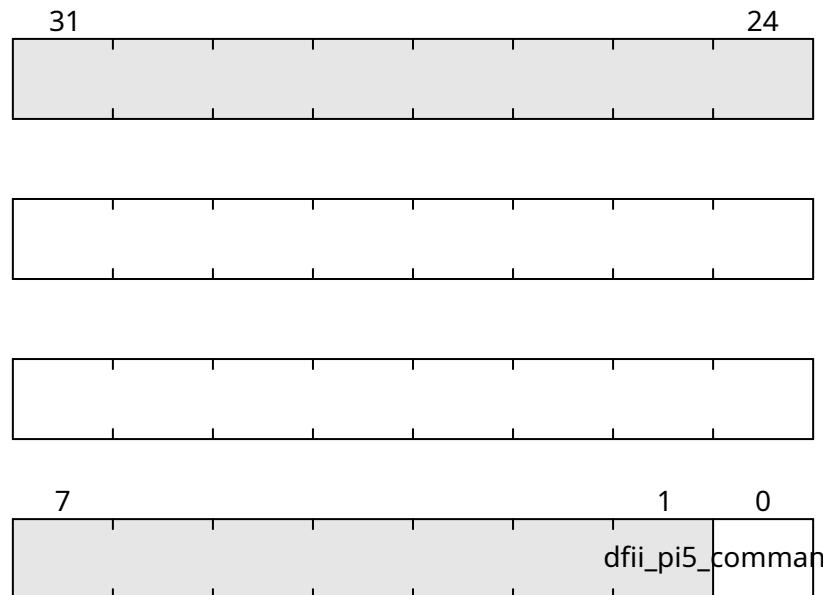


Fig. 22.123: SDRAM_DFII_PI5_COMMAND_ISSUE

SDRAM_DFII_PI5_ADDRESS

Address: $0xf0006000 + 0x84 = 0xf0006084$

DFI address bus

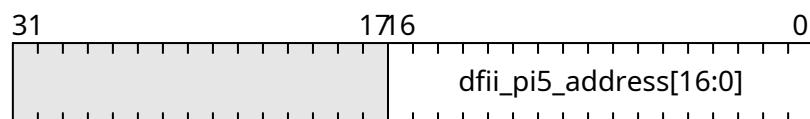


Fig. 22.124: SDRAM_DFII_PI5_ADDRESS

SDRAM_DFII_PI5_BADDRESS

Address: $0xf0006000 + 0x88 = 0xf0006088$

DFI bank address bus

SDRAM_DFII_PI5_WRDATA

Address: $0xf0006000 + 0x8c = 0xf000608c$

DFI write data bus

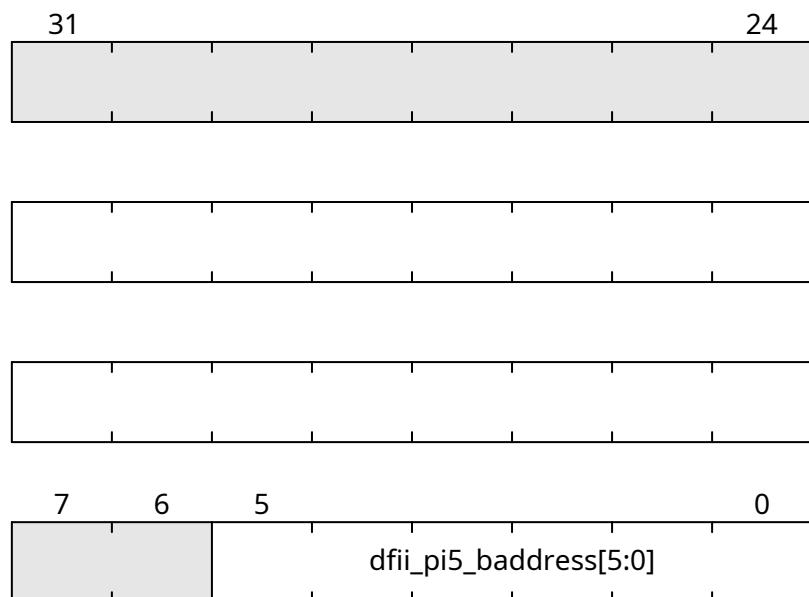


Fig. 22.125: `SDRAM_DFII_PI5_BADDRESS`

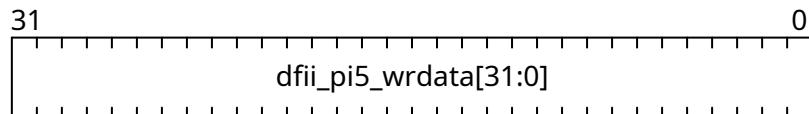


Fig. 22.126: `SDRAM_DFII_PI5_WRDATA`

SDRAM_DFII_PI5_RDDATA

Address: $0xf0006000 + 0x90 = 0xf0006090$

DFI read data bus

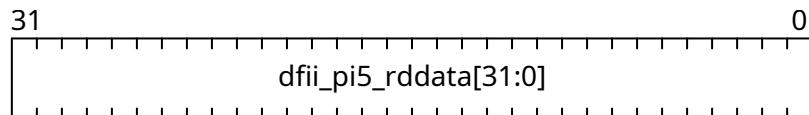


Fig. 22.127: SDRAM_DFII_PI5_RDDATA

SDRAM_DFII_PI6_COMMAND

Address: $0xf0006000 + 0x94 = 0xf0006094$

Control DFI signals on a single phase

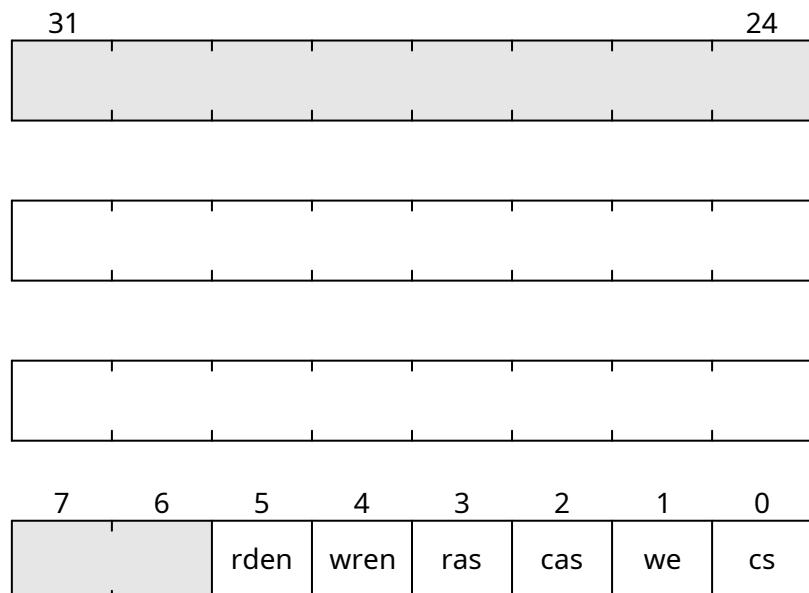


Fig. 22.128: SDRAM_DFII_PI6_COMMAND

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFII_PI6_COMMAND_ISSUE

Address: $0xf0006000 + 0x98 = 0xf0006098$

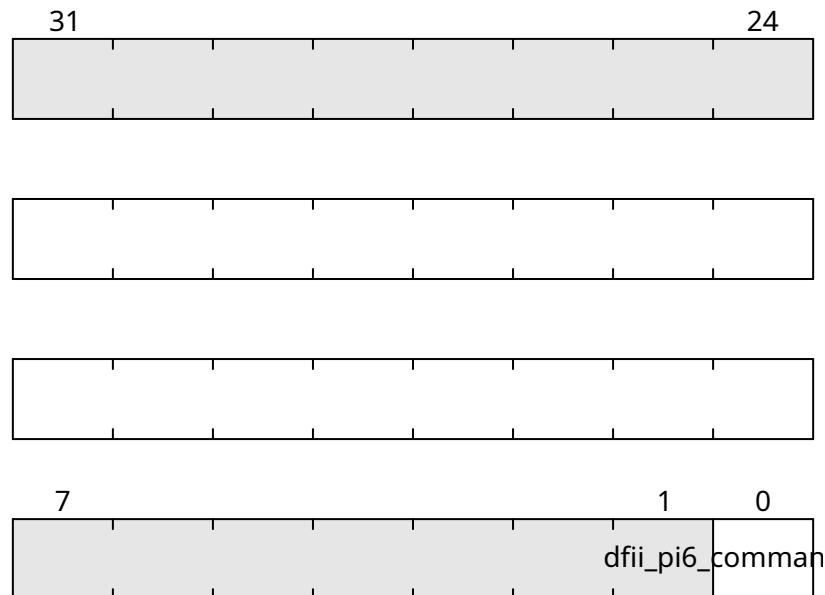


Fig. 22.129: SDRAM_DFII_PI6_COMMAND_ISSUE

SDRAM_DFII_PI6_ADDRESS

Address: $0xf0006000 + 0x9c = 0xf000609c$

DFI address bus

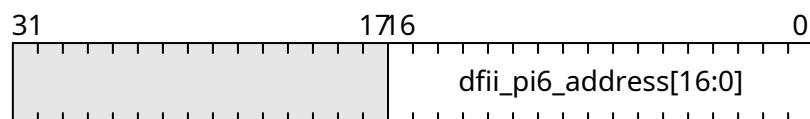


Fig. 22.130: SDRAM_DFII_PI6_ADDRESS

SDRAM_DFII_PI6_BADDRESS

Address: $0xf0006000 + 0xa0 = 0xf00060a0$

DFI bank address bus

SDRAM_DFII_PI6_WRDATA

Address: $0xf0006000 + 0xa4 = 0xf00060a4$

DFI write data bus

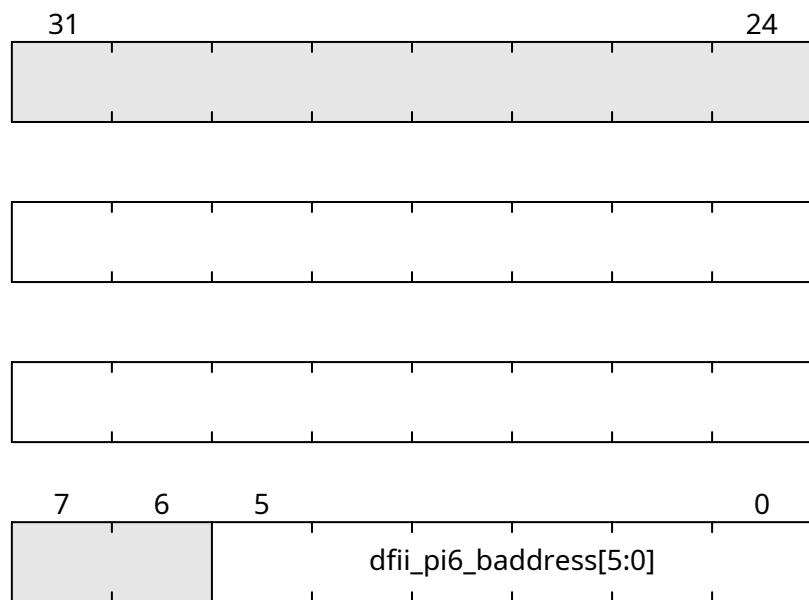


Fig. 22.131: `SDRAM_DFII_PI6_BADDRESS`

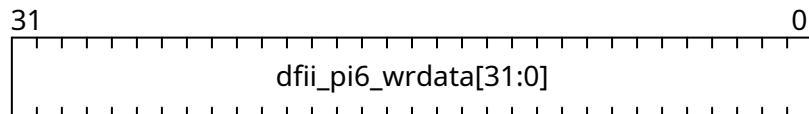


Fig. 22.132: `SDRAM_DFII_PI6_WRDATA`

SDRAM_DFII_PI6_RDDATA

Address: $0xf0006000 + 0xa8 = 0xf00060a8$

DFI read data bus

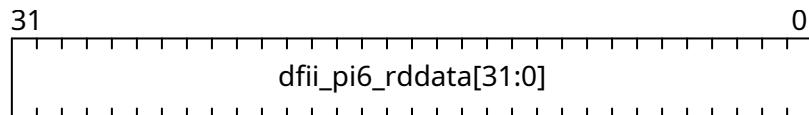


Fig. 22.133: SDRAM_DFII_PI6_RDDATA

SDRAM_DFII_PI7_COMMAND

Address: $0xf0006000 + 0xac = 0xf00060ac$

Control DFI signals on a single phase

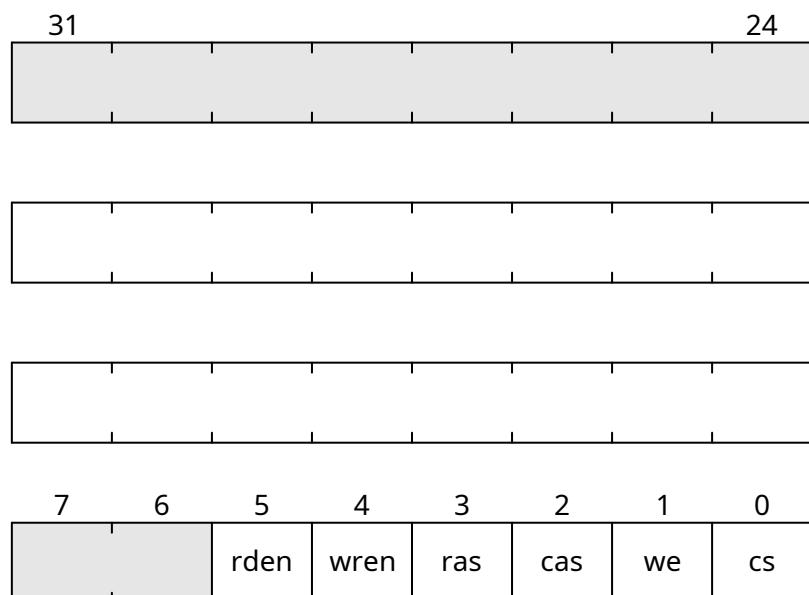


Fig. 22.134: SDRAM_DFII_PI7_COMMAND

Field	Name	Description
[0]	CS	DFI chip select bus
[1]	WE	DFI write enable bus
[2]	CAS	DFI column address strobe bus
[3]	RAS	DFI row address strobe bus
[4]	WREN	DFI write data enable bus
[5]	RDEN	DFI read data enable bus

SDRAM_DFII_PI7_COMMAND_ISSUE

Address: $0xf0006000 + 0xb0 = 0xf00060b0$

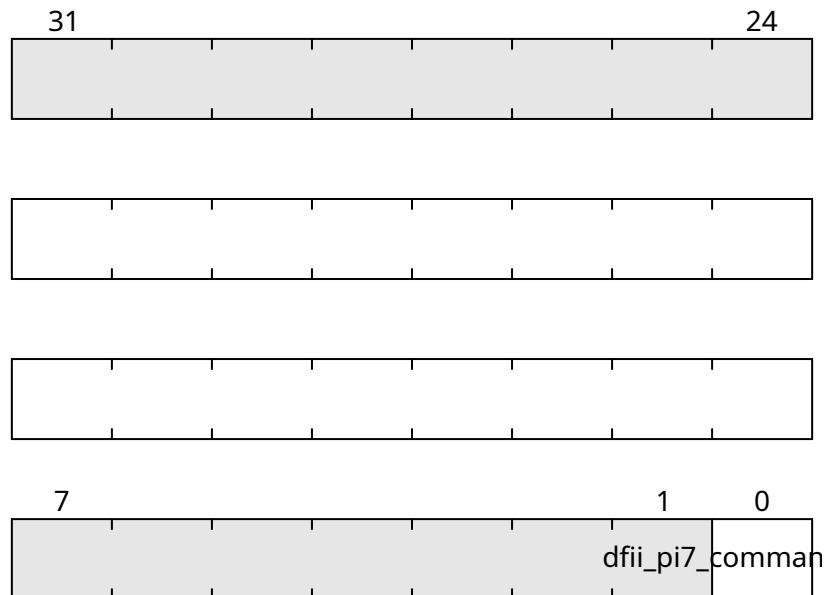


Fig. 22.135: SDRAM_DFII_PI7_COMMAND_ISSUE

SDRAM_DFII_PI7_ADDRESS

Address: $0xf0006000 + 0xb4 = 0xf00060b4$

DFI address bus

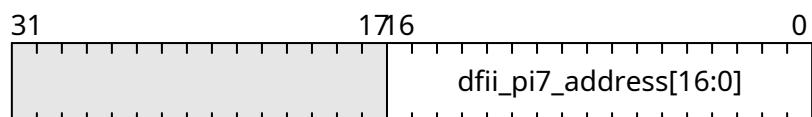


Fig. 22.136: SDRAM_DFII_PI7_ADDRESS

SDRAM_DFII_PI7_BADDRESS

Address: $0xf0006000 + 0xb8 = 0xf00060b8$

DFI bank address bus

SDRAM_DFII_PI7_WRDATA

Address: $0xf0006000 + 0xbc = 0xf00060bc$

DFI write data bus

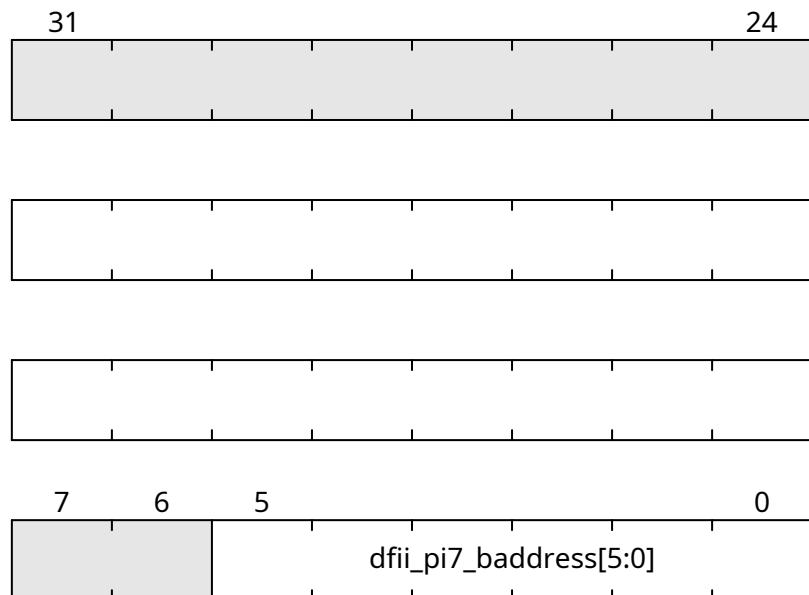


Fig. 22.137: SDRAM_DFII_PI7_BADDRESS

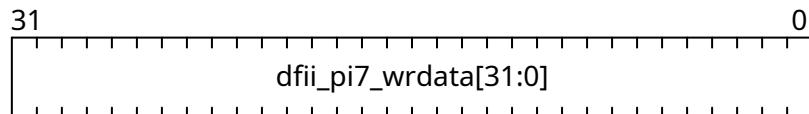


Fig. 22.138: SDRAM_DFII_PI7_WRDATA

SDRAM_DFII_PI7_RDDATA

Address: $0xf0006000 + 0xc0 = 0xf00060c0$

DFI read data bus

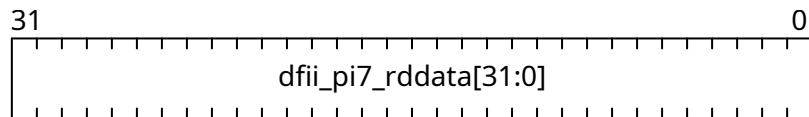


Fig. 22.139: SDRAM_DFII_PI7_RDDATA

SDRAM_CONTROLLER_TRP

Address: $0xf0006000 + 0xc4 = 0xf00060c4$

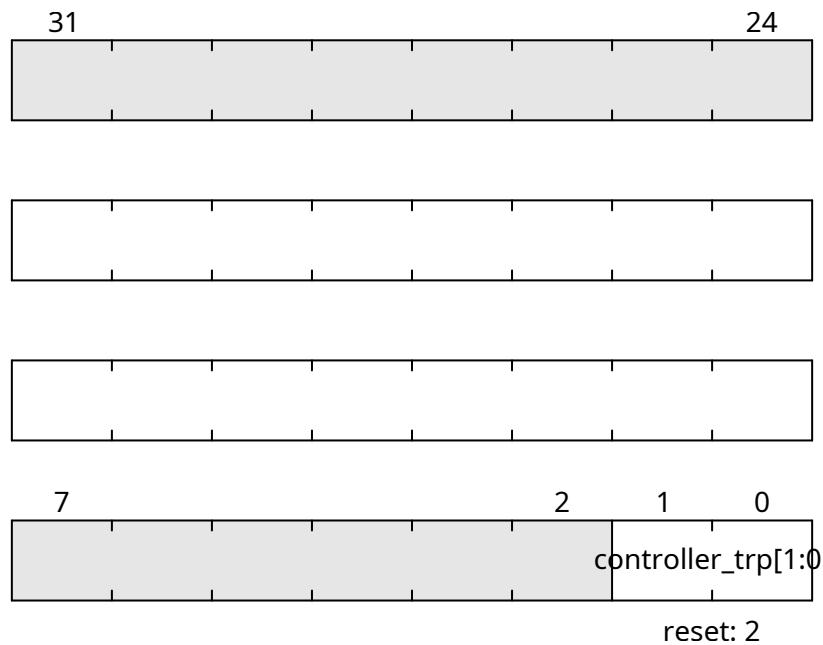


Fig. 22.140: SDRAM_CONTROLLER_TRP

SDRAM_CONTROLLER_TRCD

Address: $0xf0006000 + 0xc8 = 0xf00060c8$

SDRAM_CONTROLLER_TWR

Address: $0xf0006000 + 0xcc = 0xf00060cc$

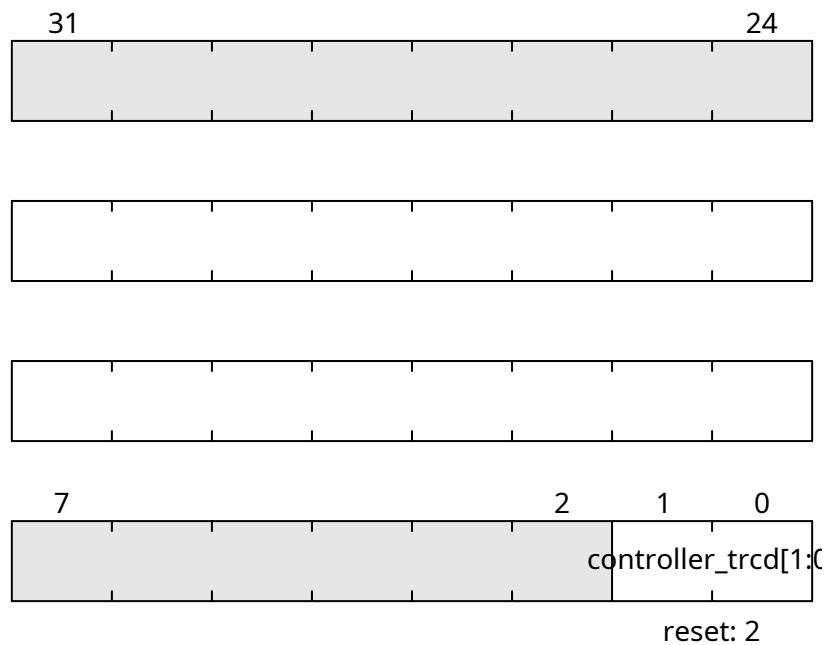


Fig. 22.141: SDRAM_CONTROLLER_TRCD

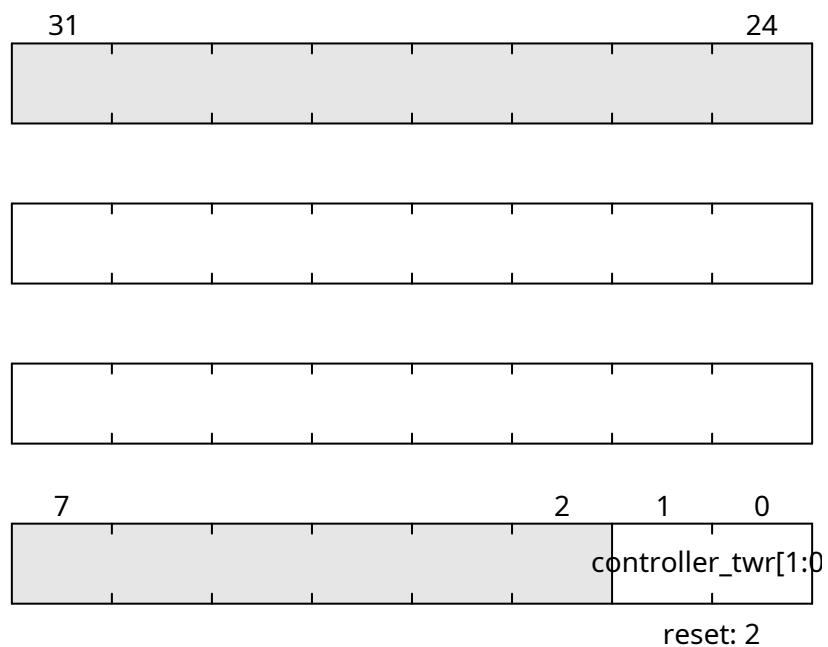


Fig. 22.142: SDRAM_CONTROLLER_TWR

SDRAM_CONTROLLER_TWTR

Address: $0xf0006000 + 0xd0 = 0xf00060d0$

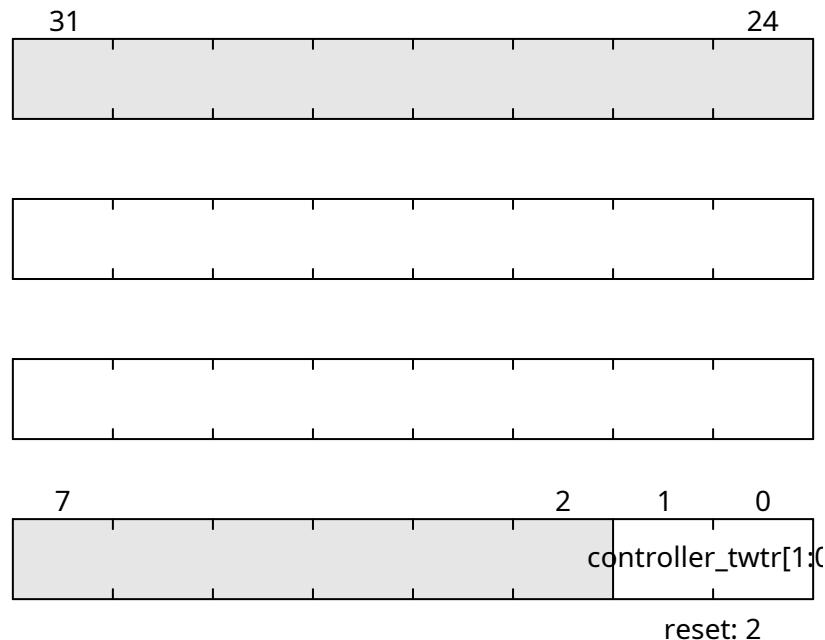


Fig. 22.143: SDRAM_CONTROLLER_TWTR

SDRAM_CONTROLLER_TREFI

Address: $0xf0006000 + 0xd4 = 0xf00060d4$

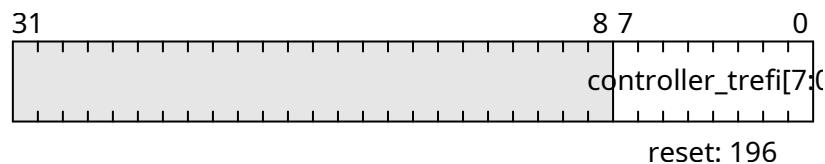


Fig. 22.144: SDRAM_CONTROLLER_TREFI

SDRAM_CONTROLLER_TRFC

Address: $0xf0006000 + 0xd8 = 0xf00060d8$

SDRAM_CONTROLLER_TFAW

Address: $0xf0006000 + 0xdc = 0xf00060dc$

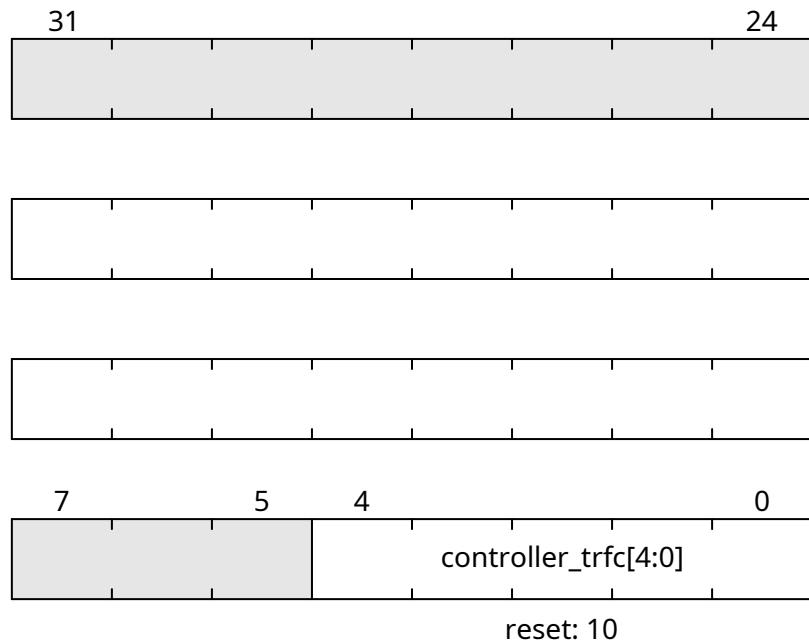


Fig. 22.145: SDRAM_CONTROLLER_TRFC

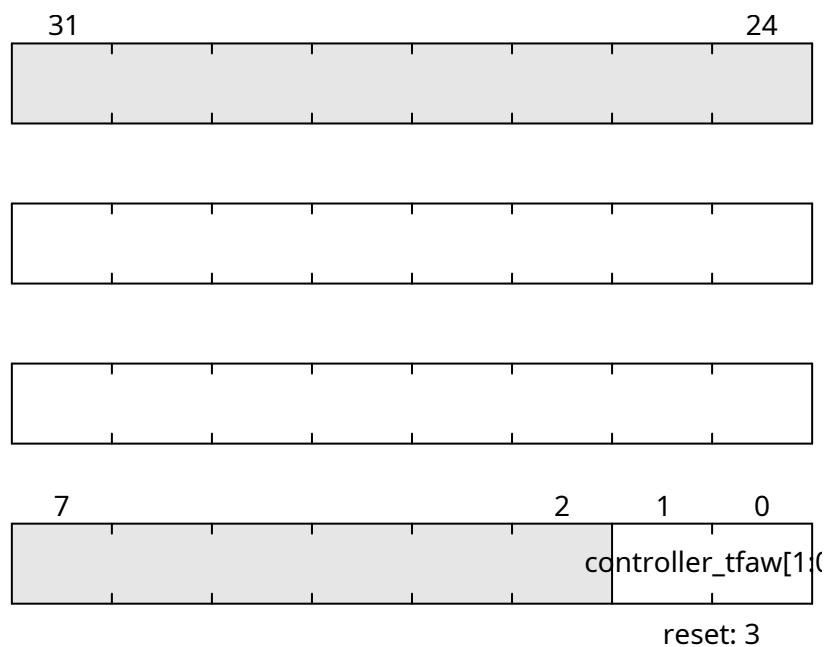


Fig. 22.146: SDRAM_CONTROLLER_TFAW

SDRAM_CONTROLLER_TCCD

Address: $0xf0006000 + 0xe0 = 0xf00060e0$

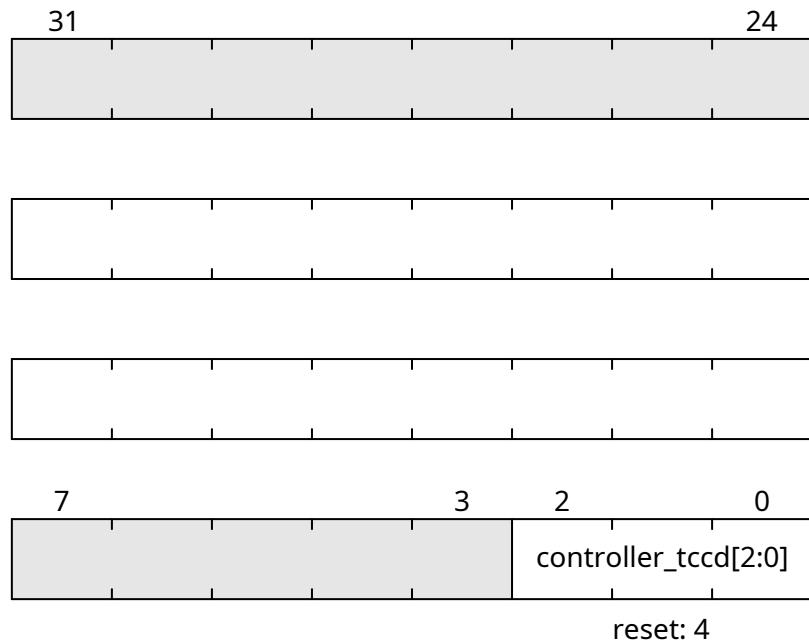


Fig. 22.147: SDRAM_CONTROLLER_TCCD

SDRAM_CONTROLLER_TCCD_WR

Address: $0xf0006000 + 0xe4 = 0xf00060e4$

SDRAM_CONTROLLER_TRTP

Address: $0xf0006000 + 0xe8 = 0xf00060e8$

SDRAM_CONTROLLER_TRRD

Address: $0xf0006000 + 0xec = 0xf00060ec$

SDRAM_CONTROLLER_TRC

Address: $0xf0006000 + 0xf0 = 0xf00060f0$

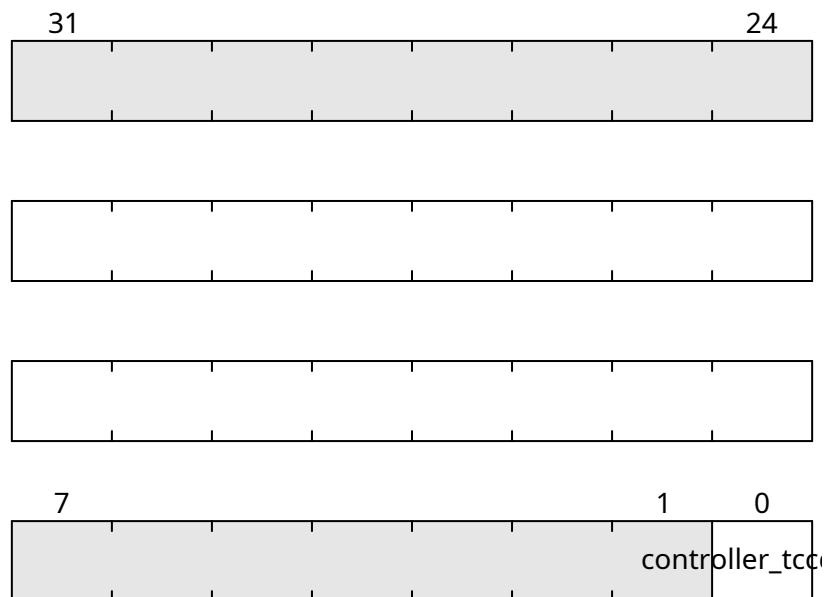


Fig. 22.148: SDRAM_CONTROLLER_TCCD_WR

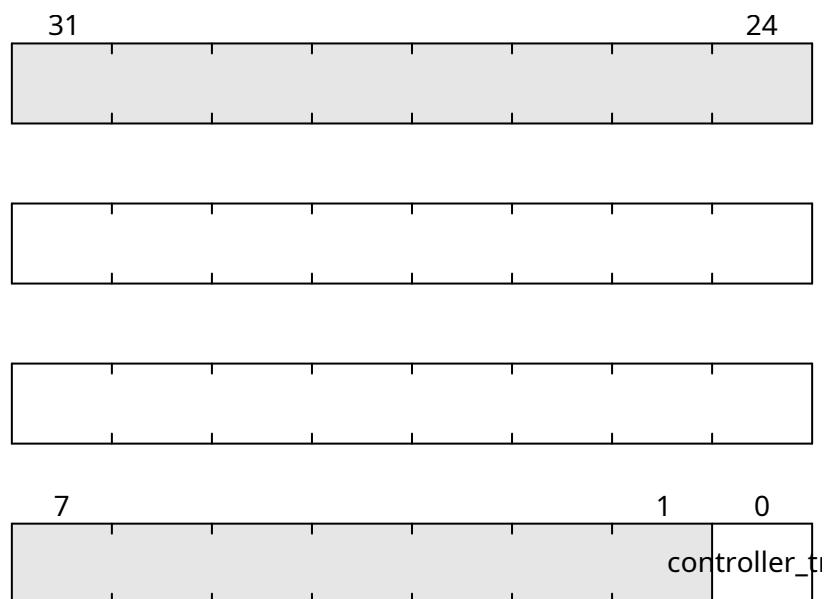


Fig. 22.149: SDRAM_CONTROLLER_TRTP

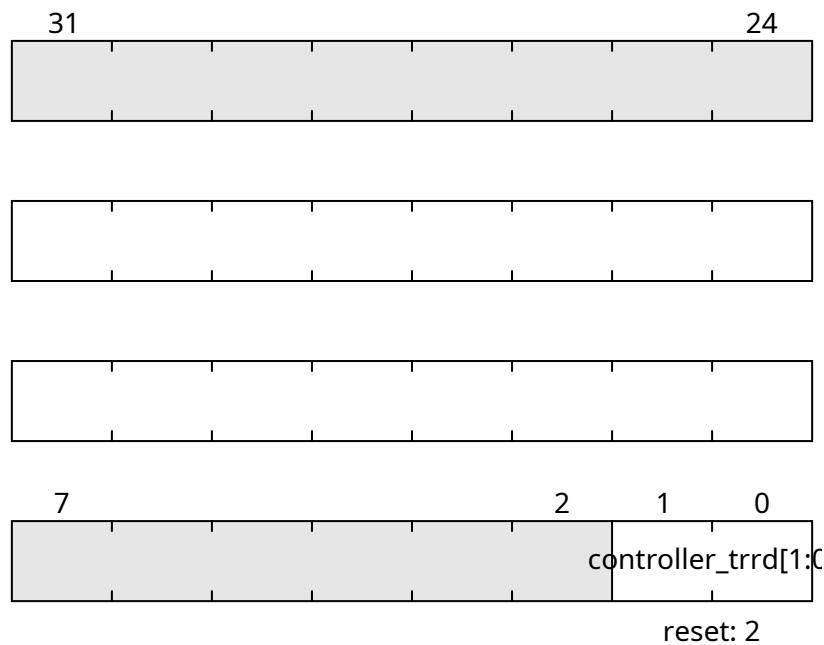


Fig. 22.150: SDRAM_CONTROLLER_TRRD

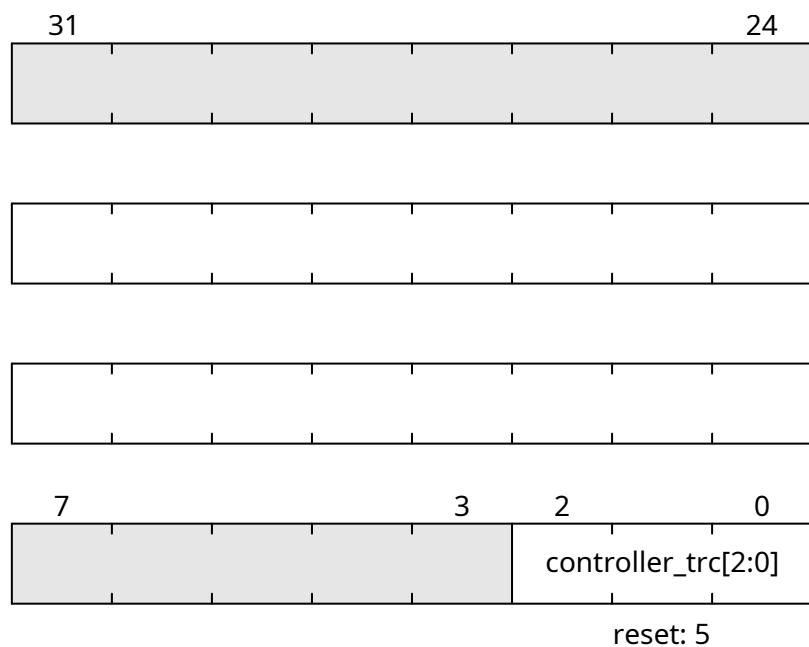


Fig. 22.151: SDRAM_CONTROLLER_TRC

SDRAM_CONTROLLER_TRAS

Address: $0xf0006000 + 0xf4 = 0xf00060f4$

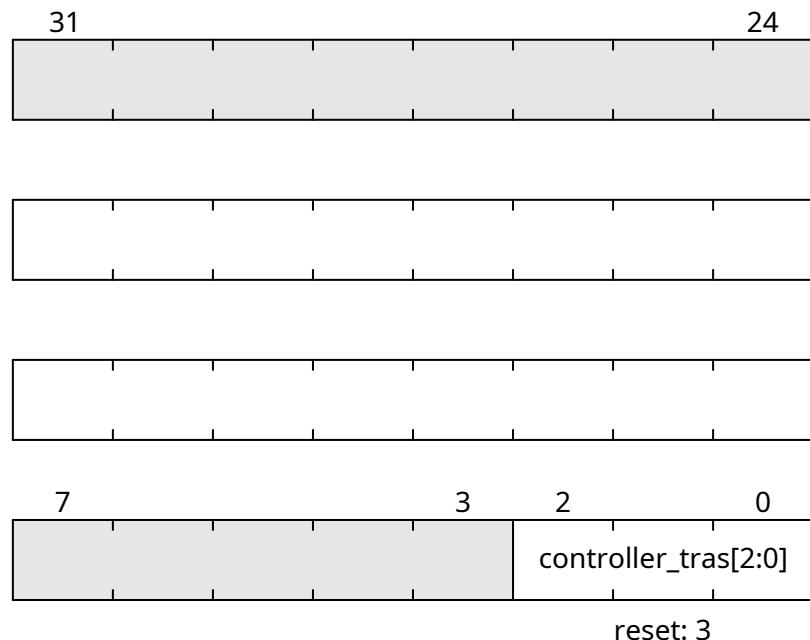


Fig. 22.152: SDRAM CONTROLLER TRAS

SDRAM CONTROLLER LAST ADDR 0

Address: 0xf0006000 + 0xf8 = 0xf00060f8

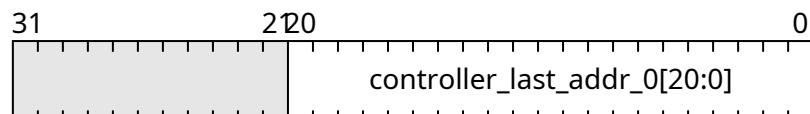


Fig. 22.153: SDRAM CONTROLLER LAST ADDR 0

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0

Address: $0xf0006000 + 0xfc = 0xf00060fc$

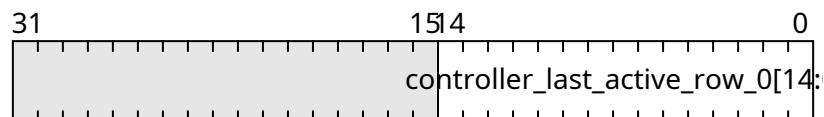


Fig. 22.154: SDRAM CONTROLLER LAST ACTIVE ROW 0

SDRAM_CONTROLLER_LAST_ADDR_1

Address: $0xf0006000 + 0x100 = 0xf0006100$

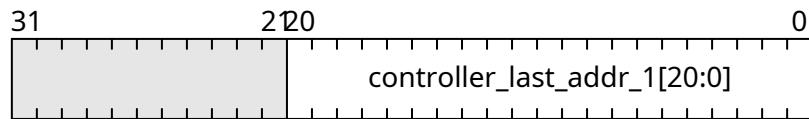


Fig. 22.155: SDRAM_CONTROLLER_LAST_ADDR_1

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1

Address: $0xf0006000 + 0x104 = 0xf0006104$

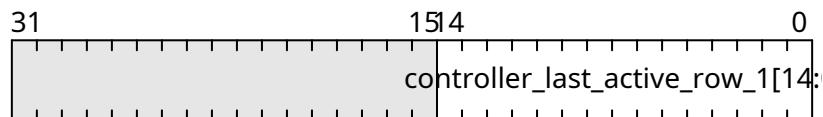


Fig. 22.156: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1

SDRAM_CONTROLLER_LAST_ADDR_2

Address: $0xf0006000 + 0x108 = 0xf0006108$

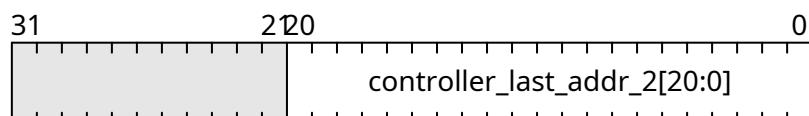


Fig. 22.157: SDRAM_CONTROLLER_LAST_ADDR_2

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2

Address: $0xf0006000 + 0x10c = 0xf000610c$

SDRAM_CONTROLLER_LAST_ADDR_3

Address: $0xf0006000 + 0x110 = 0xf0006110$

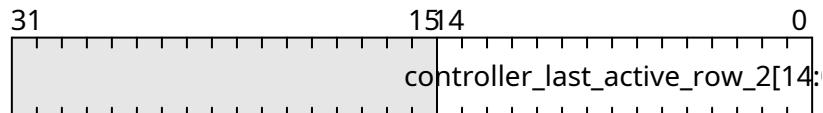


Fig. 22.158: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2

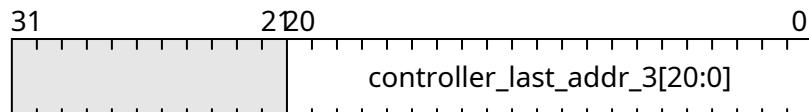


Fig. 22.159: SDRAM_CONTROLLER_LAST_ADDR_3

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3

Address: $0xf0006000 + 0x114 = 0xf0006114$

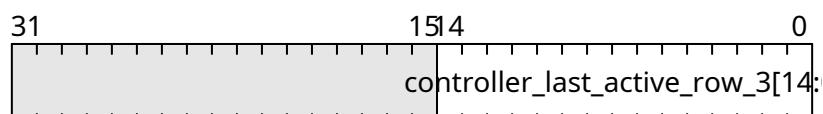


Fig. 22.160: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3

SDRAM_CONTROLLER_LAST_ADDR_4

Address: $0xf0006000 + 0x118 = 0xf0006118$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4

Address: $0xf0006000 + 0x11c = 0xf000611c$

SDRAM_CONTROLLER_LAST_ADDR_5

Address: $0xf0006000 + 0x120 = 0xf0006120$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5

Address: $0xf0006000 + 0x124 = 0xf0006124$

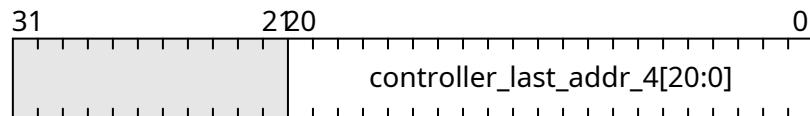


Fig. 22.161: SDRAM_CONTROLLER_LAST_ADDR_4

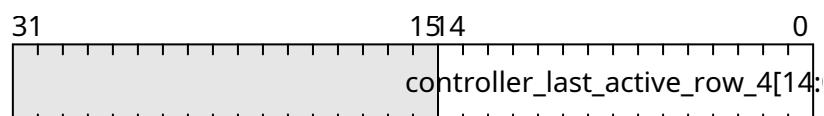


Fig. 22.162: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4

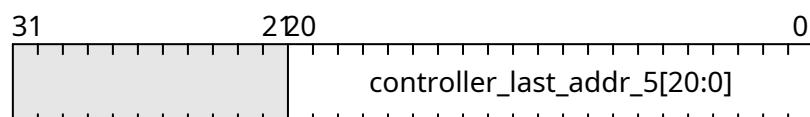


Fig. 22.163: SDRAM_CONTROLLER_LAST_ADDR_5

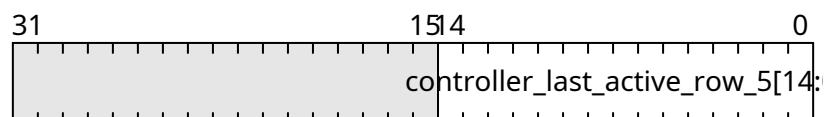


Fig. 22.164: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5

SDRAM_CONTROLLER_LAST_ADDR_6

Address: $0xf0006000 + 0x128 = 0xf0006128$

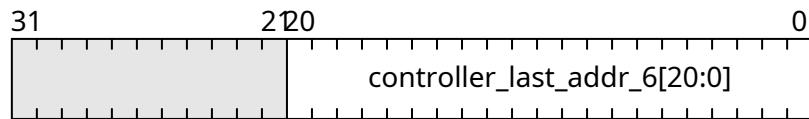


Fig. 22.165: SDRAM_CONTROLLER_LAST_ADDR_6

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6

Address: $0xf0006000 + 0x12c = 0xf000612c$

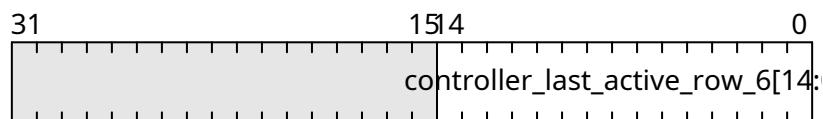


Fig. 22.166: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6

SDRAM_CONTROLLER_LAST_ADDR_7

Address: $0xf0006000 + 0x130 = 0xf0006130$

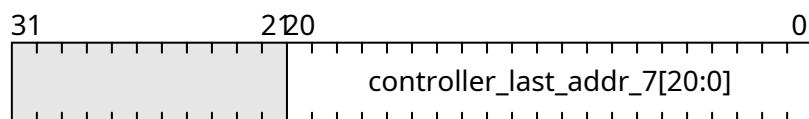


Fig. 22.167: SDRAM_CONTROLLER_LAST_ADDR_7

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7

Address: $0xf0006000 + 0x134 = 0xf0006134$

22.2.14 SDRAM_CHECKER

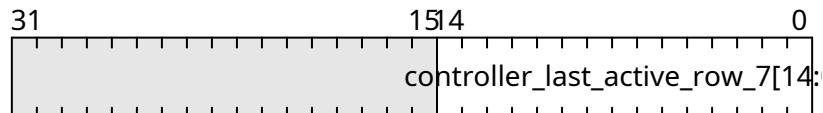


Fig. 22.168: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7

Register Listing for SDRAM_CHECKER

Register	Address
<i>SDRAM_CHECKER_RESET</i>	0xf0006800
<i>SDRAM_CHECKER_START</i>	0xf0006804
<i>SDRAM_CHECKER_DONE</i>	0xf0006808
<i>SDRAM_CHECKER_BASE</i>	0xf000680c
<i>SDRAM_CHECKER_END</i>	0xf0006810
<i>SDRAM_CHECKER_LENGTH</i>	0xf0006814
<i>SDRAM_CHECKER_RANDOM</i>	0xf0006818
<i>SDRAM_CHECKER_TICKS</i>	0xf000681c
<i>SDRAM_CHECKER_ERRORS</i>	0xf0006820

SDRAM_CHECKER_RESET

Address: $0xf0006800 + 0x0 = 0xf0006800$



Fig. 22.169: SDRAM_CHECKER_RESET

SDRAM_CHECKER_START

Address: $0xf0006800 + 0x4 = 0xf0006804$



Fig. 22.170: SDRAM_CHECKER_START

SDRAM_CHECKER_DONE

Address: $0xf0006800 + 0x8 = 0xf0006808$

SDRAM_CHECKER_BASE

Address: $0xf0006800 + 0xc = 0xf000680c$

SDRAM_CHECKER_END

Address: $0xf0006800 + 0x10 = 0xf0006810$

SDRAM_CHECKER_LENGTH

Address: $0xf0006800 + 0x14 = 0xf0006814$



Fig. 22.171: SDRAM_CHECKER_DONE

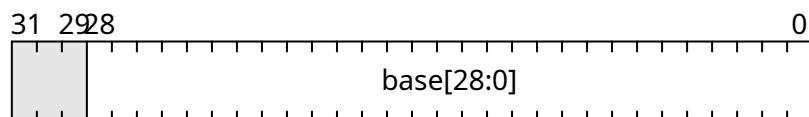


Fig. 22.172: SDRAM_CHECKER_BASE

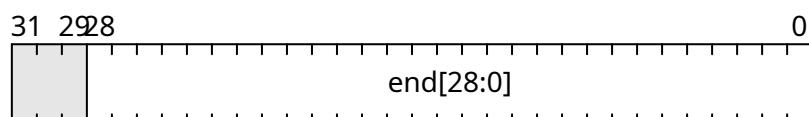


Fig. 22.173: SDRAM_CHECKER_END

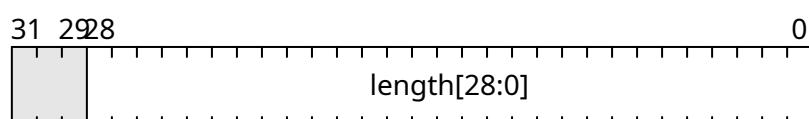


Fig. 22.174: SDRAM_CHECKER_LENGTH

SDRAM_CHECKER_RANDOM

Address: $0xf0006800 + 0x18 = 0xf0006818$



Fig. 22.175: SDRAM_CHECKER_RANDOM

Field	Name	Description

SDRAM_CHECKER_TICKS

Address: $0xf0006800 + 0x1c = 0xf000681c$

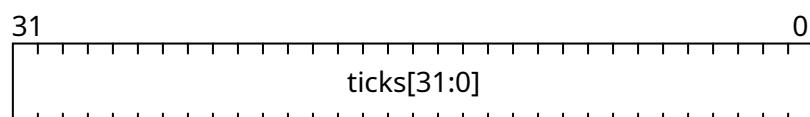


Fig. 22.176: SDRAM_CHECKER_TICKS

SDRAM_CHECKER_ERRORS

Address: $0xf0006800 + 0x20 = 0xf0006820$

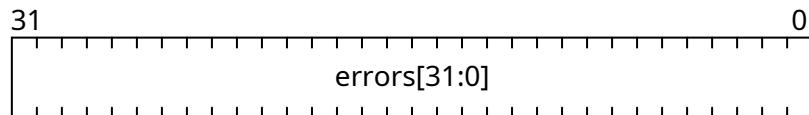


Fig. 22.177: SDRAM_CHECKER_ERRORS

22.2.15 SDRAM_GENERATOR

Register Listing for SDRAM_GENERATOR

Register	Address
<i>SDRAM_GENERATOR_RESET</i>	0xf0007000
<i>SDRAM_GENERATOR_START</i>	0xf0007004
<i>SDRAM_GENERATOR_DONE</i>	0xf0007008
<i>SDRAM_GENERATOR_BASE</i>	0xf000700c
<i>SDRAM_GENERATOR_END</i>	0xf0007010
<i>SDRAM_GENERATOR_LENGTH</i>	0xf0007014
<i>SDRAM_GENERATOR_RANDOM</i>	0xf0007018
<i>SDRAM_GENERATOR_TICKS</i>	0xf000701c

SDRAM_GENERATOR_RESET

Address: $0xf0007000 + 0x0 = 0xf0007000$



Fig. 22.178: SDRAM_GENERATOR_RESET

SDRAM_GENERATOR_START

Address: $0xf0007000 + 0x4 = 0xf0007004$

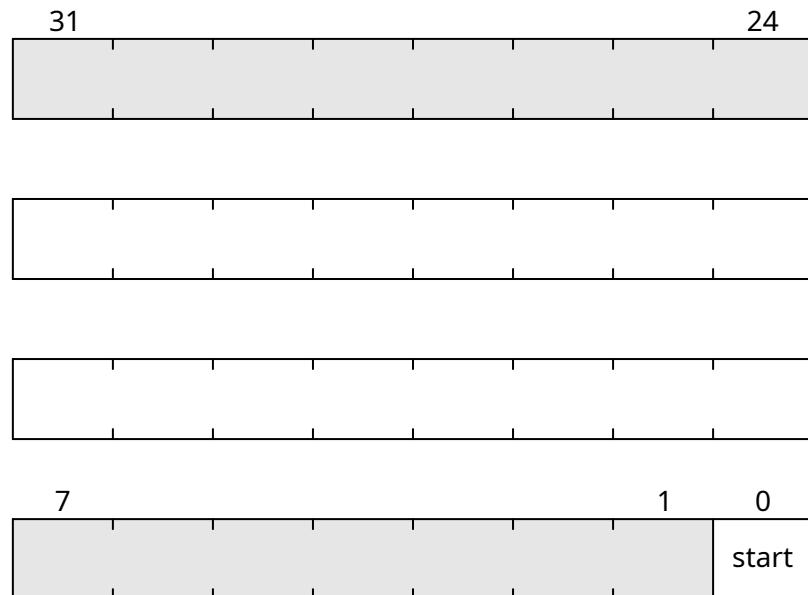


Fig. 22.179: SDRAM_GENERATOR_START

SDRAM_GENERATOR_DONE

Address: $0xf0007000 + 0x8 = 0xf0007008$

SDRAM_GENERATOR_BASE

Address: $0xf0007000 + 0xc = 0xf000700c$

SDRAM_GENERATOR_END

Address: $0xf0007000 + 0x10 = 0xf0007010$

SDRAM_GENERATOR_LENGTH

Address: $0xf0007000 + 0x14 = 0xf0007014$



Fig. 22.180: SDRAM_GENERATOR_DONE

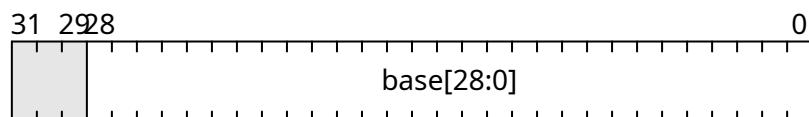


Fig. 22.181: SDRAM_GENERATOR_BASE

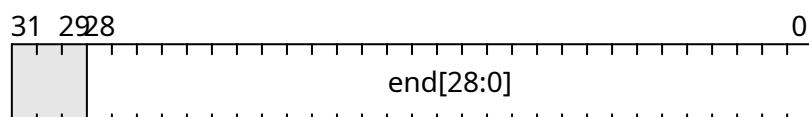


Fig. 22.182: SDRAM_GENERATOR_END

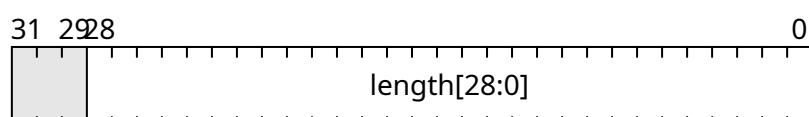


Fig. 22.183: SDRAM_GENERATOR_LENGTH

SDRAM_GENERATOR_RANDOM

Address: $0xf0007000 + 0x18 = 0xf0007018$



Fig. 22.184: SDRAM_GENERATOR_RANDOM

Field	Name	Description

SDRAM_GENERATOR_TICKS

Address: $0xf0007000 + 0x1c = 0xf000701c$

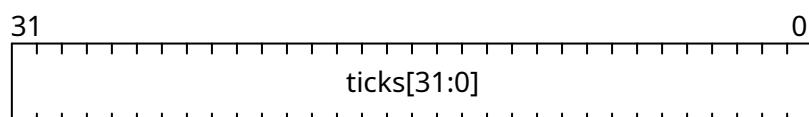


Fig. 22.185: SDRAM_GENERATOR_TICKS

22.2.16 TIMER0

Timer

Provides a generic Timer core.

The Timer is implemented as a countdown timer that can be used in various modes:

- Polling : Returns current countdown value to software
- One-Shot: Loads itself and stops when value reaches 0
- Periodic: (Re-)Loads itself when value reaches 0

`en` register allows the user to enable/disable the Timer. When the Timer is enabled, it is automatically loaded with the value of `load` register.

When the Timer reaches 0, it is automatically reloaded with value of `reload` register.

The user can latch the current countdown value by writing to `update_value` register, it will update `value` register with current countdown value.

To use the Timer in One-Shot mode, the user needs to:

- Disable the timer
- Set the load register to the expected duration
- (Re-)Enable the Timer

To use the Timer in Periodic mode, the user needs to:

- Disable the Timer
- Set the load register to 0
- Set the reload register to the expected period
- Enable the Timer

For both modes, the CPU can be advertised by an IRQ that the duration/period has elapsed. (The CPU can also do software polling with `update_value` and `value` to know the elapsed duration)

Register Listing for TIMERO

Register	Address
<code>TIMERO_LOAD</code>	<code>0xf0007800</code>
<code>TIMERO_RELOAD</code>	<code>0xf0007804</code>
<code>TIMERO_EN</code>	<code>0xf0007808</code>
<code>TIMERO_UPDATE_VALUE</code>	<code>0xf000780c</code>
<code>TIMERO_VALUE</code>	<code>0xf0007810</code>
<code>TIMERO_EV_STATUS</code>	<code>0xf0007814</code>
<code>TIMERO_EV_PENDING</code>	<code>0xf0007818</code>
<code>TIMERO_EV_ENABLE</code>	<code>0xf000781c</code>

`TIMERO_LOAD`

Address: `0xf0007800 + 0x0 = 0xf0007800`

Load value when Timer is (re-)enabled. In One-Shot mode, the value written to this register specifies the Timer's duration in clock cycles.

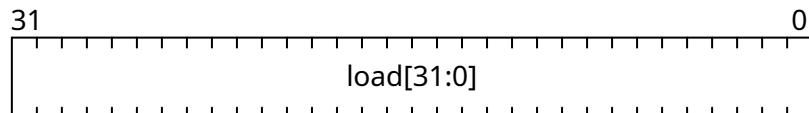


Fig. 22.186: TIMER0_LOAD

TIMER0_RELOAD

Address: $0xf0007800 + 0x4 = 0xf0007804$

Reload value when Timer reaches 0. In Periodic mode, the value written to this register specify the Timer's period in clock cycles.

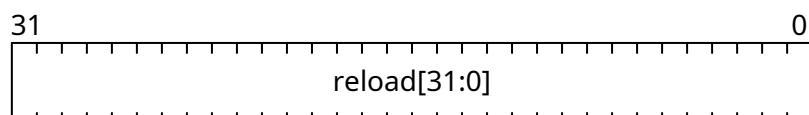


Fig. 22.187: TIMER0_RELOAD

TIMER0_EN

Address: $0xf0007800 + 0x8 = 0xf0007808$

Enable flag of the Timer. Set this flag to 1 to enable/start the Timer. Set to 0 to disable the Timer.



Fig. 22.188: TIMER0_EN

TIMER0_UPDATE_VALUE

Address: $0xf0007800 + 0xc = 0xf000780c$

Update trigger for the current countdown value. A write to this register latches the current countdown value to value register.



Fig. 22.189: TIMER0_UPDATE_VALUE

TIMER0_VALUE

Address: $0xf0007800 + 0x10 = 0xf0007810$

Latched countdown value. This value is updated by writing to update_value.

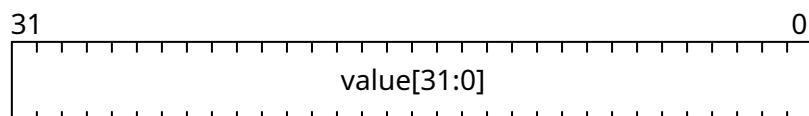


Fig. 22.190: TIMER0_VALUE

TIMER0_EV_STATUS

Address: $0xf0007800 + 0x14 = 0xf0007814$

This register contains the current raw level of the zero event trigger. Writes to this register have no effect.

Field	Name	Description
[0]	ZERO	Level of the zero event

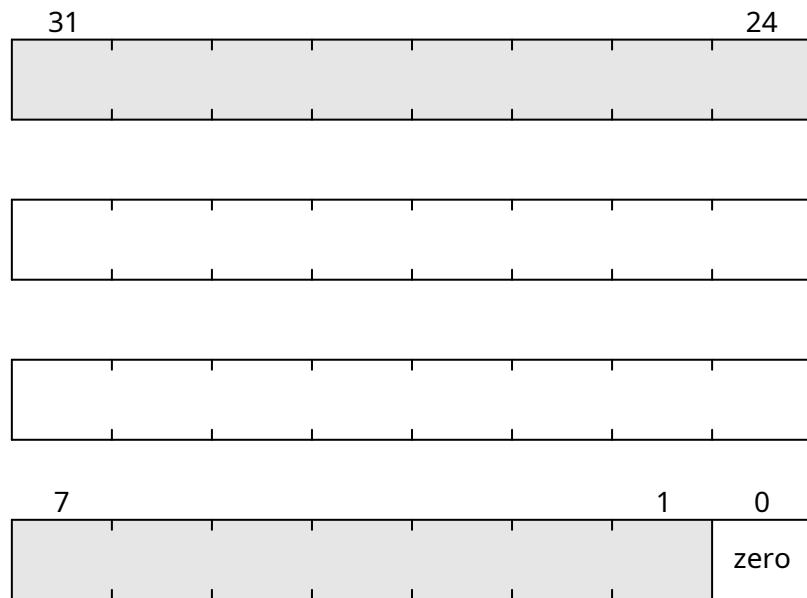


Fig. 22.191: TIMERO_EV_STATUS

TIMERO_EV_PENDING

Address: $0xf0007800 + 0x18 = 0xf0007818$

When a zero event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.

Field	Name	Description
[0]	ZERO	1 if a zero event occurred. This Event is triggered on a falling edge.

TIMERO_EV_ENABLE

Address: $0xf0007800 + 0x1c = 0xf000781c$

This register enables the corresponding zero events. Write a 0 to this register to disable individual events.

Field	Name	Description
[0]	ZERO	Write a 1 to enable the zero Event



Fig. 22.192: `TIMER0_EV_PENDING`



Fig. 22.193: `TIMER0_EV_ENABLE`

22.2.17 UART

Register Listing for UART

Register	Address
<i>UART_RXTX</i>	0xf0008000
<i>UART_TXFULL</i>	0xf0008004
<i>UART_RXEMPTY</i>	0xf0008008
<i>UART_EV_STATUS</i>	0xf000800c
<i>UART_EV_PENDING</i>	0xf0008010
<i>UART_EV_ENABLE</i>	0xf0008014
<i>UART_TXEMPTY</i>	0xf0008018
<i>UART_RXFULL</i>	0xf000801c
<i>UART_XOVER_RXTX</i>	0xf0008020
<i>UART_XOVER_TXFULL</i>	0xf0008024
<i>UART_XOVER_RXEMPTY</i>	0xf0008028
<i>UART_XOVER_EV_STATUS</i>	0xf000802c
<i>UART_XOVER_EV_PENDING</i>	0xf0008030
<i>UART_XOVER_EV_ENABLE</i>	0xf0008034
<i>UART_XOVER_TXEMPTY</i>	0xf0008038
<i>UART_XOVER_RXFULL</i>	0xf000803c

UART_RXTX

Address: $0xf0008000 + 0x0 = 0xf0008000$

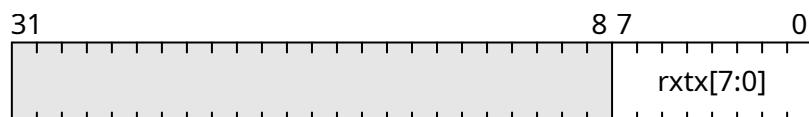


Fig. 22.194: *UART_RXTX*

UART_TXFULL

Address: $0xf0008000 + 0x4 = 0xf0008004$

TX FIFO Full.

UART_RXEMPTY

Address: $0xf0008000 + 0x8 = 0xf0008008$

RX FIFO Empty.



Fig. 22.195: `UART_TXFULL`



Fig. 22.196: `UART_RXEMPTY`

UART_EV_STATUS

Address: $0xf0008000 + 0xc = 0xf000800c$

This register contains the current raw level of the rx event trigger. Writes to this register have no effect.

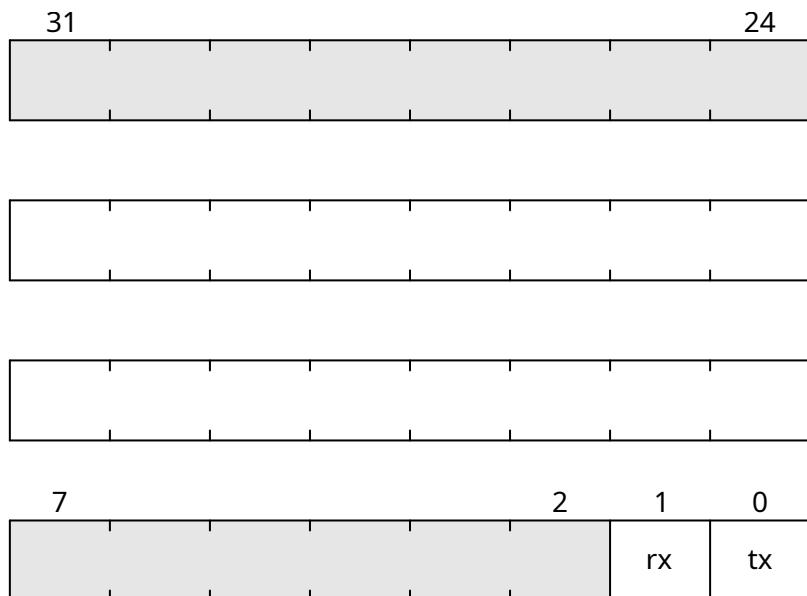


Fig. 22.197: UART_EV_STATUS

Field	Name	Description
[0]	TX	Level of the tx event
[1]	RX	Level of the rx event

UART_EV_PENDING

Address: $0xf0008000 + 0x10 = 0xf0008010$

When a rx event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.

Field	Name	Description
[0]	TX	1 if a tx event occurred. This Event is triggered on a falling edge .
[1]	RX	1 if a rx event occurred. This Event is triggered on a falling edge .

UART_EV_ENABLE

Address: $0xf0008000 + 0x14 = 0xf0008014$

This register enables the corresponding rx events. Write a 0 to this register to disable individual events.

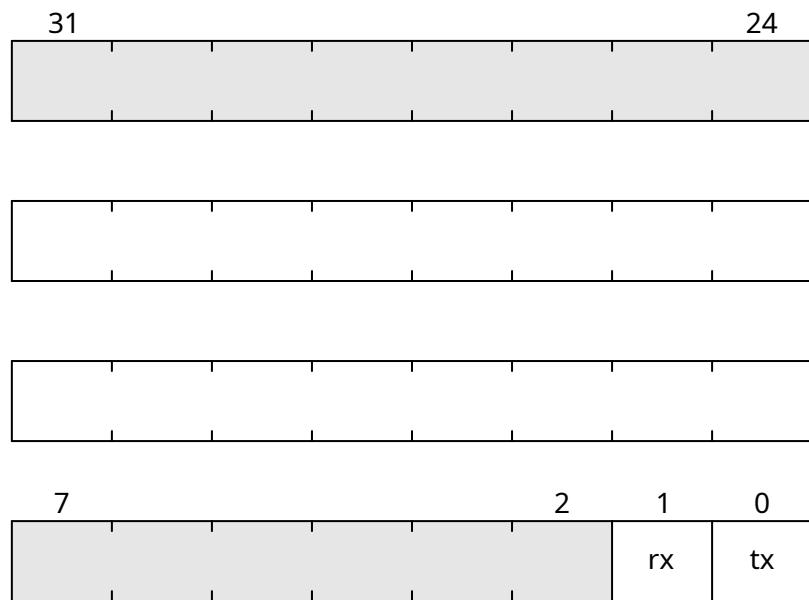


Fig. 22.198: `UART_EV_PENDING`

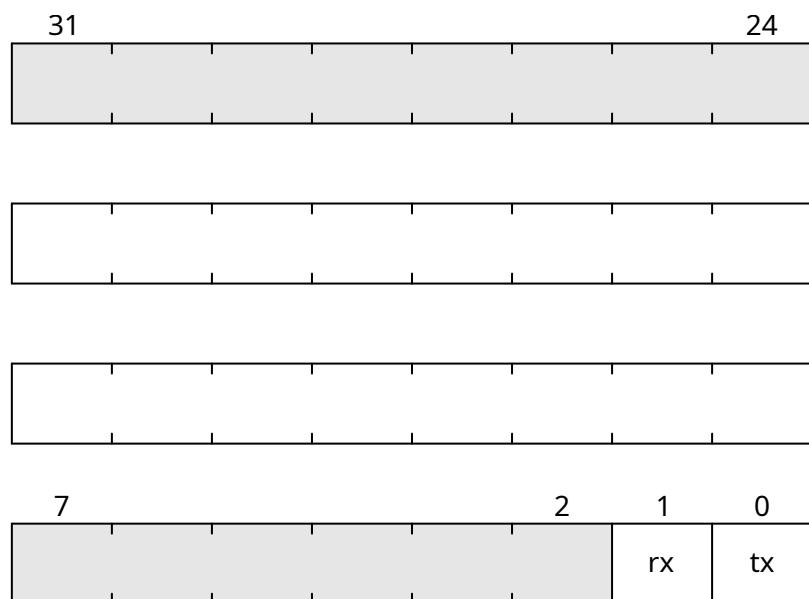


Fig. 22.199: `UART_EV_ENABLE`

Field	Name	Description
[0]	TX	Write a 1 to enable the tx Event
[1]	RX	Write a 1 to enable the rx Event

UART_TXEMPTY

Address: $0xf0008000 + 0x18 = 0xf0008018$

TX FIFO Empty.



Fig. 22.200: UART_TXEMPTY

UART_RXFULL

Address: $0xf0008000 + 0x1c = 0xf000801c$

RX FIFO Full.

UART_XOVER_RXTX

Address: $0xf0008000 + 0x20 = 0xf0008020$

UART_XOVER_TXFULL

Address: $0xf0008000 + 0x24 = 0xf0008024$

TX FIFO Full.



Fig. 22.201: `UART_RXFULL`

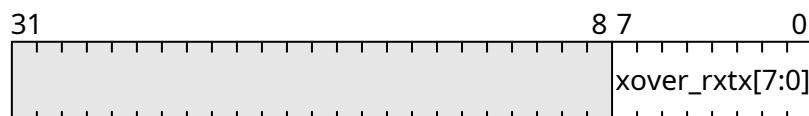


Fig. 22.202: `UART_XOVER_RXTX`

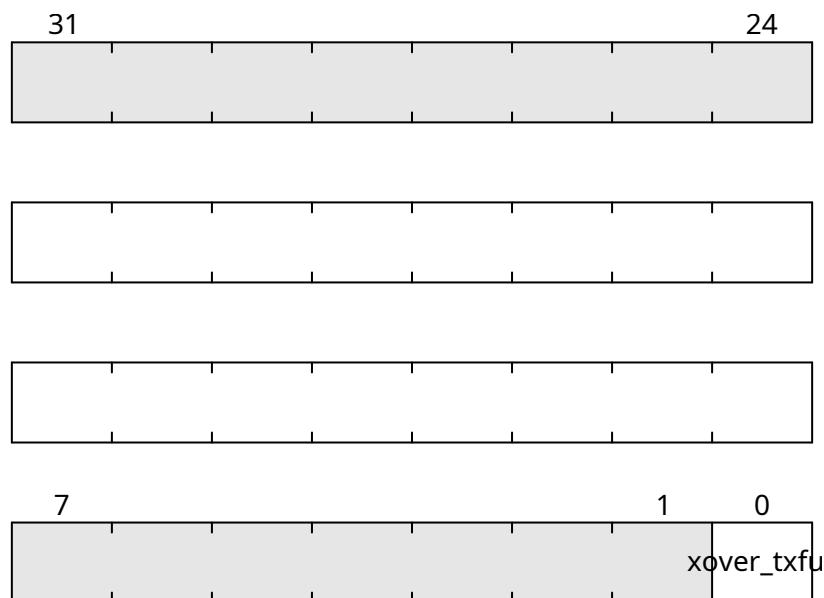


Fig. 22.203: `UART_XOVER_TXFULL`

UART_XOVER_RXEMPTY

Address: $0xf0008000 + 0x28 = 0xf0008028$

RX FIFO Empty.



Fig. 22.204: UART_XOVER_RXEMPTY

UART_XOVER_EV_STATUS

Address: $0xf0008000 + 0x2c = 0xf000802c$

This register contains the current raw level of the rx event trigger. Writes to this register have no effect.

Field	Name	Description
[0]	TX	Level of the tx event
[1]	RX	Level of the rx event

UART_XOVER_EV_PENDING

Address: $0xf0008000 + 0x30 = 0xf0008030$

When a rx event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.

Field	Name	Description
[0]	TX	1 if a tx event occurred. This Event is triggered on a falling edge .
[1]	RX	1 if a rx event occurred. This Event is triggered on a falling edge .

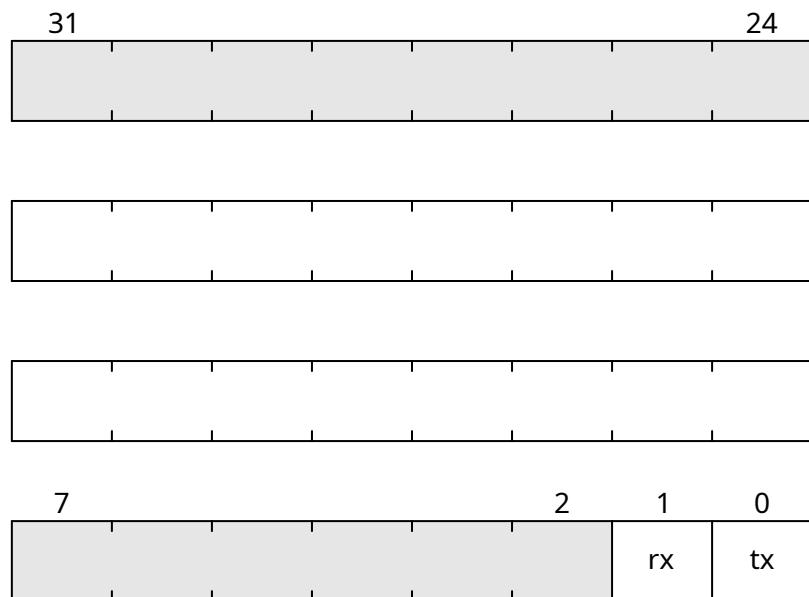


Fig. 22.205: `UART_XOVER_EV_STATUS`

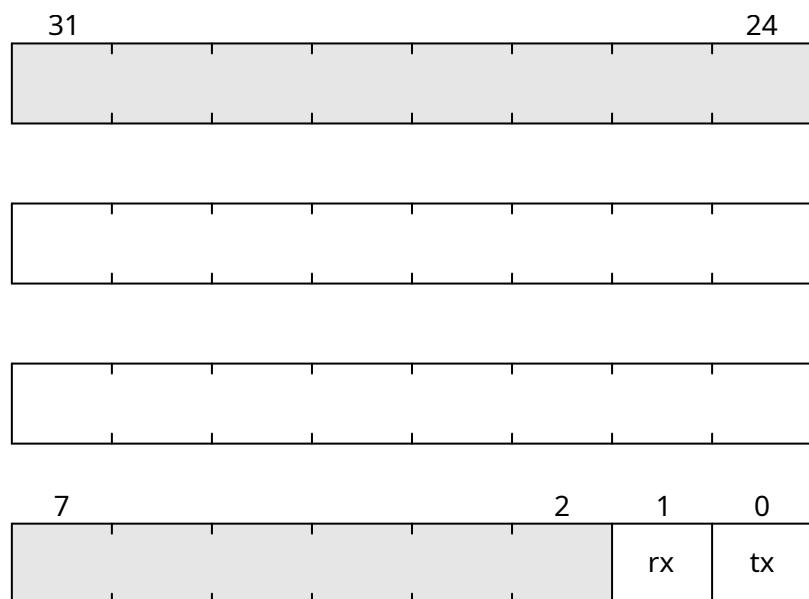


Fig. 22.206: `UART_XOVER_EV_PENDING`

UART_XOVER_EV_ENABLE

Address: $0xf0008000 + 0x34 = 0xf0008034$

This register enables the corresponding rx events. Write a 0 to this register to disable individual events.

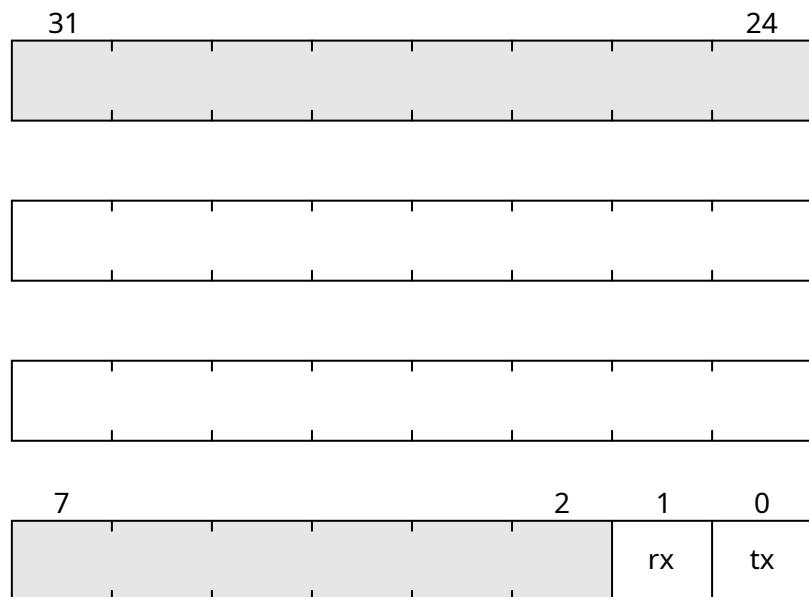


Fig. 22.207: UART_XOVER_EV_ENABLE

Field	Name	Description
[0]	TX	Write a 1 to enable the tx Event
[1]	RX	Write a 1 to enable the rx Event

UART_XOVER_TXEMPTY

Address: $0xf0008000 + 0x38 = 0xf0008038$

TX FIFO Empty.

UART_XOVER_RXFULL

Address: $0xf0008000 + 0x3c = 0xf000803c$

RX FIFO Full.

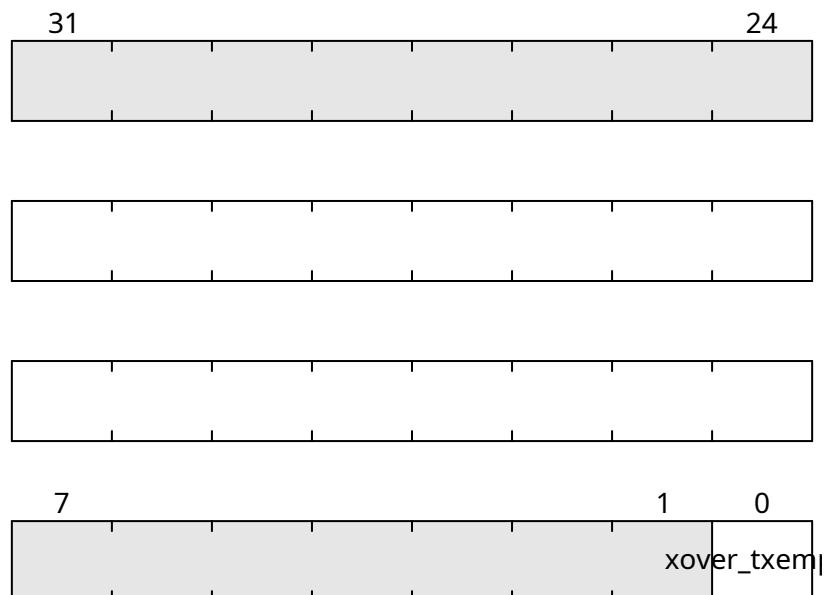


Fig. 22.208: `UART_XOVER_TXEMPTY`

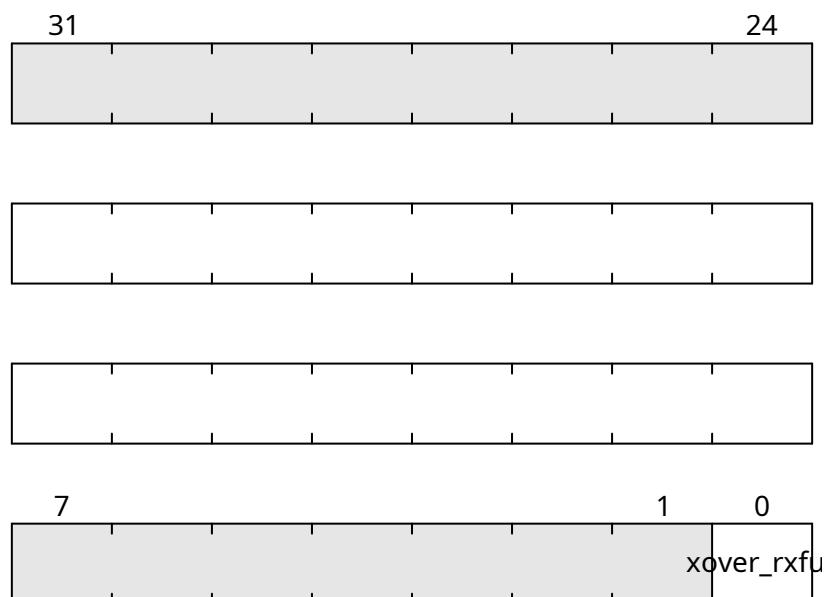


Fig. 22.209: `UART_XOVER_RXFULL`

CHAPTER
TWENTYTHREE

DOCUMENTATION FOR ROW HAMMER TESTER DDR5 TEST BOARD

23.1 Modules

23.1.1 Interrupt Controller

This device has an EventManager-based interrupt system. Individual modules generate *events* which are wired into a central interrupt controller.

When an interrupt occurs, you should look the interrupt number up in the CPU- specific interrupt table and then call the relevant module.

Assigned Interrupts

The following interrupts are assigned on this system:

Interrupt	Module
1	<i>TIMER0</i>
0	<i>UART</i>

23.2 Register Groups

23.2.1 LEDS

Register Listing for LEDS

Register	Address
<i>LEDS_OUT</i>	<i>0xf0000000</i>

LEDS_OUT

Address: *0xf0000000* + *0x0* = *0xf0000000*

Led Output(s) Control.

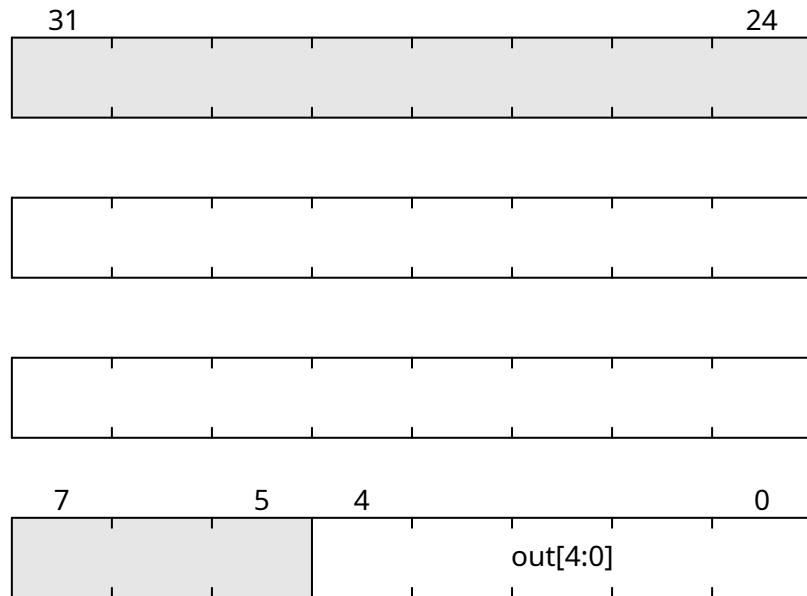


Fig. 23.1: LEDS_OUT

23.2.2 DDRPHY

Register Listing for DDRPHY

Register	Address
<i>DDRPHY_CSRMODULE_ENABLE_FIFOS</i>	0xf0000800
<i>DDRPHY_CSRMODULE_RST</i>	0xf0000804
<i>DDRPHY_CSRMODULE_RDIMM_MODE</i>	0xf0000808
<i>DDRPHY_CSRMODULE_RDPHASE</i>	0xf000080c
<i>DDRPHY_CSRMODULE_WRPHASE</i>	0xf0000810
<i>DDRPHY_CSRMODULE_ALERT</i>	0xf0000814
<i>DDRPHY_CSRMODULE_ALERT_REDUCE</i>	0xf0000818
<i>DDRPHY_CSRMODULE_SAMPLE_ALERT</i>	0xf000081c
<i>DDRPHY_CSRMODULE_RESET_ALERT</i>	0xf0000820
<i>DDRPHY_CSRMODULE_CKDLY_RST</i>	0xf0000824
<i>DDRPHY_CSRMODULE_CKDLY_INC</i>	0xf0000828
<i>DDRPHY_CSRMODULE_PREAMBLE</i>	0xf000082c
<i>DDRPHY_CSRMODULE_WLEVEL_EN</i>	0xf0000830
<i>DDRPHY_CSRMODULE_PAR_ENABLE</i>	0xf0000834
<i>DDRPHY_CSRMODULE_PAR_VALUE</i>	0xf0000838
<i>DDRPHY_CSRMODULE_DISCARD_RD_FIFO</i>	0xf000083c
<i>DDRPHY_CSRMODULE_DL_Y_SEL</i>	0xf0000840
<i>DDRPHY_CSRMODULE_DQ_DQS_RATIO</i>	0xf0000844
<i>DDRPHY_CSRMODULE_CK_RDLY_INC</i>	0xf0000848
<i>DDRPHY_CSRMODULE_CK_RDLY_RST</i>	0xf000084c
<i>DDRPHY_CSRMODULE_CK_RDDLY</i>	0xf0000850
<i>DDRPHY_CSRMODULE_CK_RDDLY_PREAMBLE</i>	0xf0000854
<i>DDRPHY_CSRMODULE_CK_WDLY_INC</i>	0xf0000858

continues on next page

Table 23.1 – continued from previous page

Register	Address
<i>DDRPHY_CSRMODULE_CK_WDLY_RST</i>	0xf000085c
<i>DDRPHY_CSRMODULE_CK_WDLY_DQS</i>	0xf0000860
<i>DDRPHY_CSRMODULE_CK_WDDLY_INC</i>	0xf0000864
<i>DDRPHY_CSRMODULE_CK_WDDLY_RST</i>	0xf0000868
<i>DDRPHY_CSRMODULE_CK_WDLY_DQ</i>	0xf000086c
<i>DDRPHY_CSRMODULE_DQ_DLY_SEL</i>	0xf0000870
<i>DDRPHY_CSRMODULE_CSDLY_RST</i>	0xf0000874
<i>DDRPHY_CSRMODULE_CSDLY_INC</i>	0xf0000878
<i>DDRPHY_CSRMODULE_CADLY_RST</i>	0xf000087c
<i>DDRPHY_CSRMODULE_CADLY_INC</i>	0xf0000880
<i>DDRPHY_CSRMODULE_PARDLY_RST</i>	0xf0000884
<i>DDRPHY_CSRMODULE_PARDLY_INC</i>	0xf0000888
<i>DDRPHY_CSRMODULE_CSDLY</i>	0xf000088c
<i>DDRPHY_CSRMODULE_CADLY</i>	0xf0000890
<i>DDRPHY_CSRMODULE_RDLY_DQ_RST</i>	0xf0000894
<i>DDRPHY_CSRMODULE_RDLY_DQ_INC</i>	0xf0000898
<i>DDRPHY_CSRMODULE_RDLY_DQS_RST</i>	0xf000089c
<i>DDRPHY_CSRMODULE_RDLY_DQS_INC</i>	0xf00008a0
<i>DDRPHY_CSRMODULE_RDLY_DQS</i>	0xf00008a4
<i>DDRPHY_CSRMODULE_RDLY_DQ</i>	0xf00008a8
<i>DDRPHY_CSRMODULE_WDLY_DQ_RST</i>	0xf00008ac
<i>DDRPHY_CSRMODULE_WDLY_DQ_INC</i>	0xf00008b0
<i>DDRPHY_CSRMODULE_WDLY_DM_RST</i>	0xf00008b4
<i>DDRPHY_CSRMODULE_WDLY_DM_INC</i>	0xf00008b8
<i>DDRPHY_CSRMODULE_WDLY_DQS_RST</i>	0xf00008bc
<i>DDRPHY_CSRMODULE_WDLY_DQS_INC</i>	0xf00008c0
<i>DDRPHY_CSRMODULE_WDLY_DQS</i>	0xf00008c4
<i>DDRPHY_CSRMODULE_WDLY_DQ</i>	0xf00008c8
<i>DDRPHY_CSRMODULE_WDLY_DM</i>	0xf00008cc

DDRPHY_CSRMODULE_ENABLE_FIFOS

Address: 0xf0000800 + 0x0 = 0xf0000800

DDRPHY_CSRMODULE_RST

Address: 0xf0000800 + 0x4 = 0xf0000804

DDRPHY_CSRMODULE_RDIMM_MODE

Address: 0xf0000800 + 0x8 = 0xf0000808

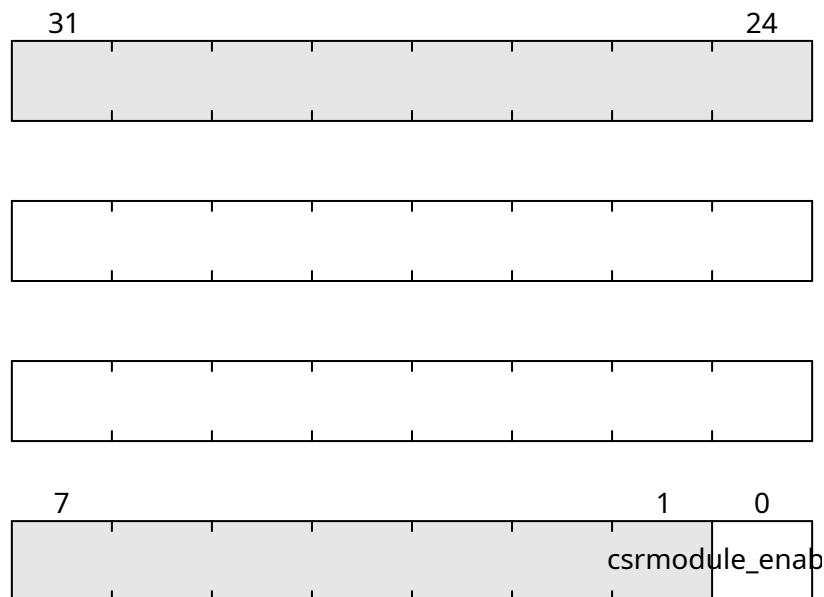


Fig. 23.2: DDRPHY_CSRMODULE_ENABLE_FIFOS

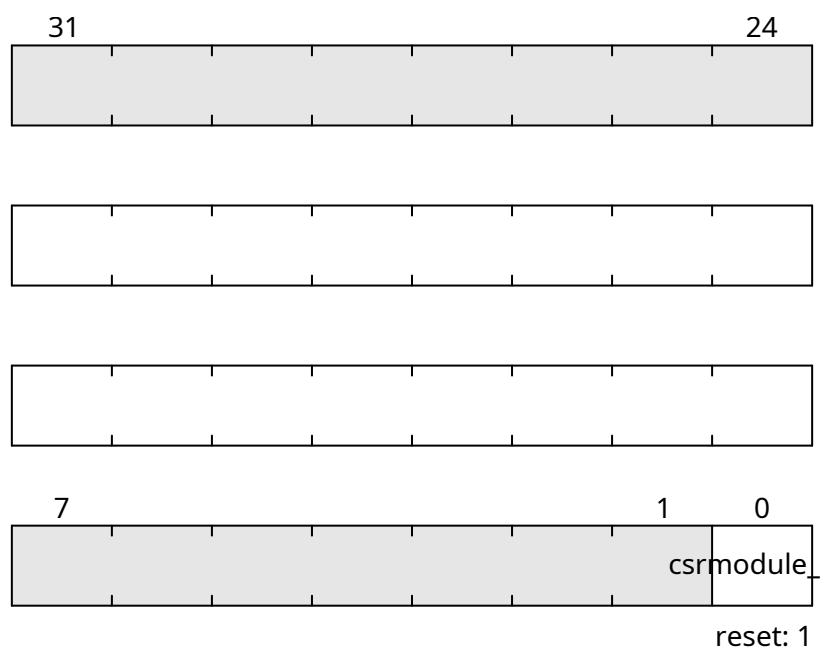


Fig. 23.3: DDRPHY_CSRMODULE_RST

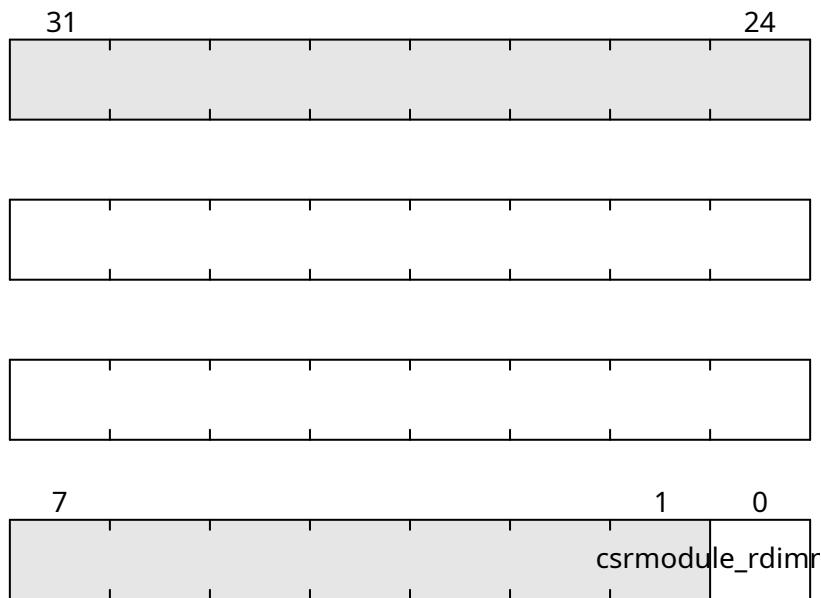


Fig. 23.4: DDRPHY_CSRMODULE_RDIMM_MODE

DDRPHY_CSRMODULE_RDPHASE

Address: 0xf0000800 + 0xc = 0xf000080c

DDRPHY_CSRMODULE_WRPHASE

Address: 0xf0000800 + 0x10 = 0xf0000810

DDRPHY_CSRMODULE_ALERT

Address: 0xf0000800 + 0x14 = 0xf0000814

DDRPHY_CSRMODULE_ALERT_REDUCE

Address: 0xf0000800 + 0x18 = 0xf0000818

Field	Name	Description
[0]	INITIAL_STATE	Initial value of all bits
[1]	OPERATION	0 - or (default), 1 -and

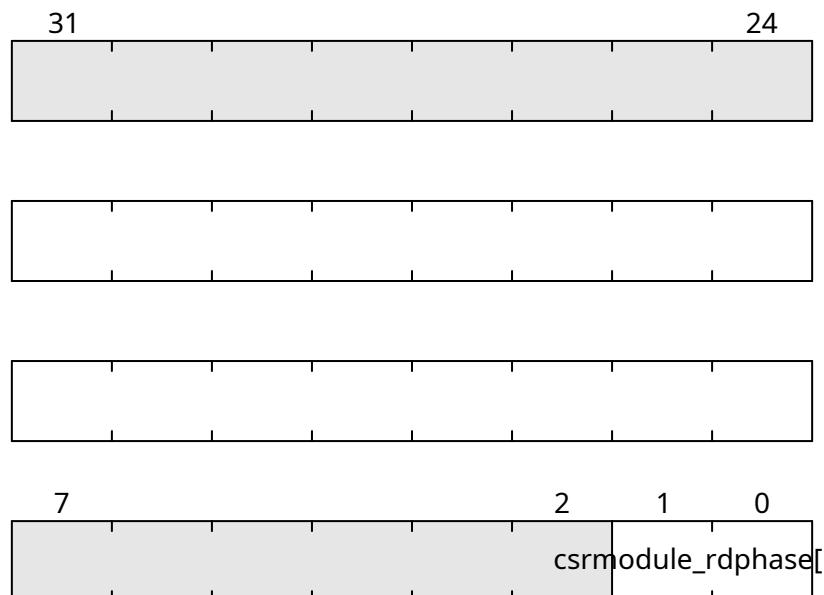


Fig. 23.5: DDRPHY_CSRMODULE_RDPHASE

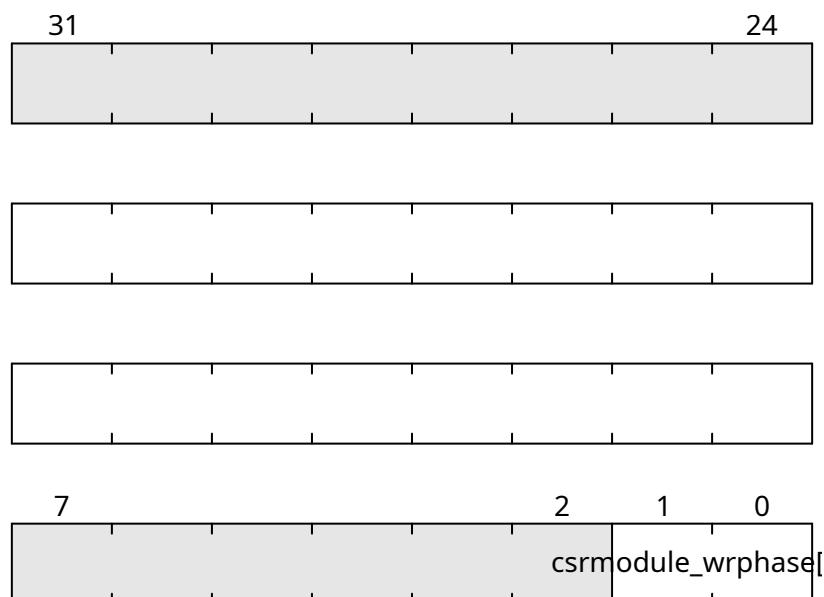


Fig. 23.6: DDRPHY_CSRMODULE_WRPAGE

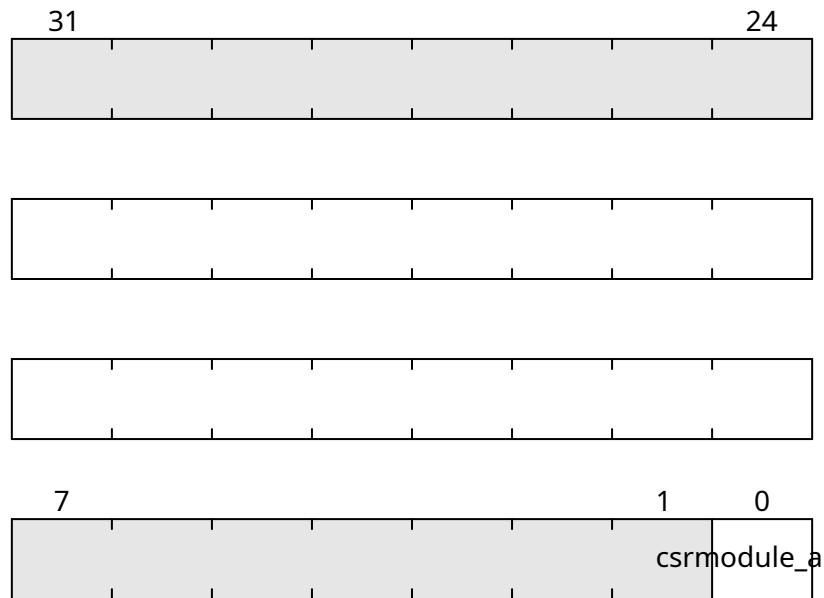


Fig. 23.7: DDRPHY_CSRMODULE_ALERT

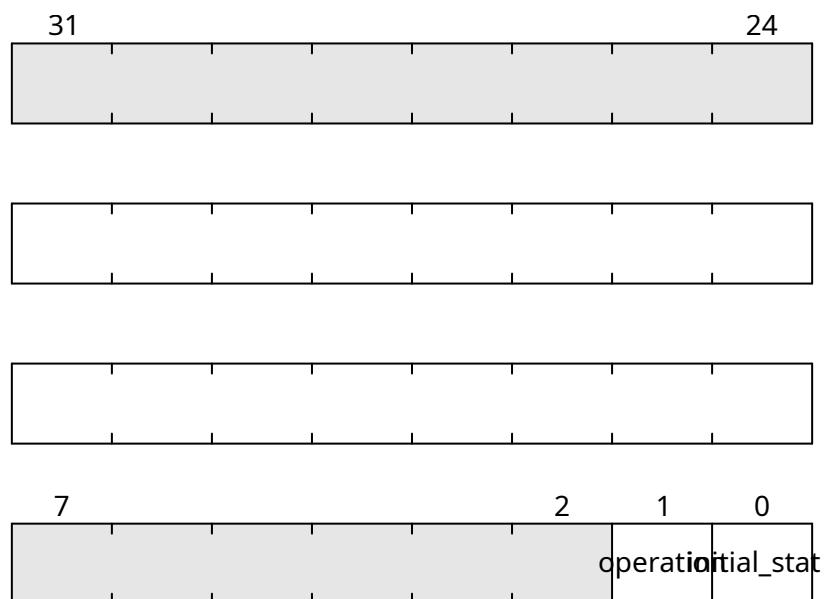


Fig. 23.8: DDRPHY_CSRMODULE_ALERT_REDUCE

DDRPHY_CSRMODULE_SAMPLE_ALERT

Address: $0xf0000800 + 0x1c = 0xf000081c$

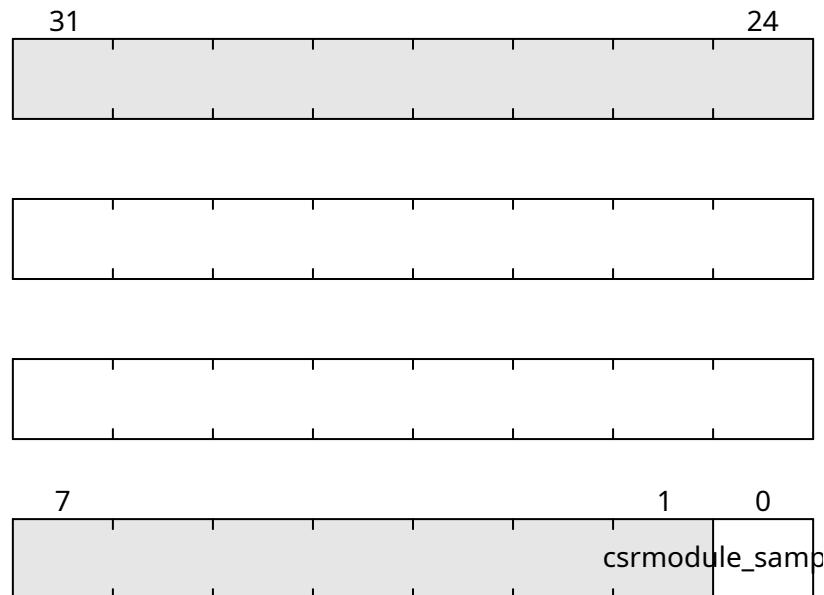


Fig. 23.9: DDRPHY_CSRMODULE_SAMPLE_ALERT

DDRPHY_CSRMODULE_RESET_ALERT

Address: $0xf0000800 + 0x20 = 0xf0000820$

DDRPHY_CSRMODULE_CKDLY_RST

Address: $0xf0000800 + 0x24 = 0xf0000824$

DDRPHY_CSRMODULE_CKDLY_INC

Address: $0xf0000800 + 0x28 = 0xf0000828$

DDRPHY_CSRMODULE_PREAMBLE

Address: $0xf0000800 + 0x2c = 0xf000082c$

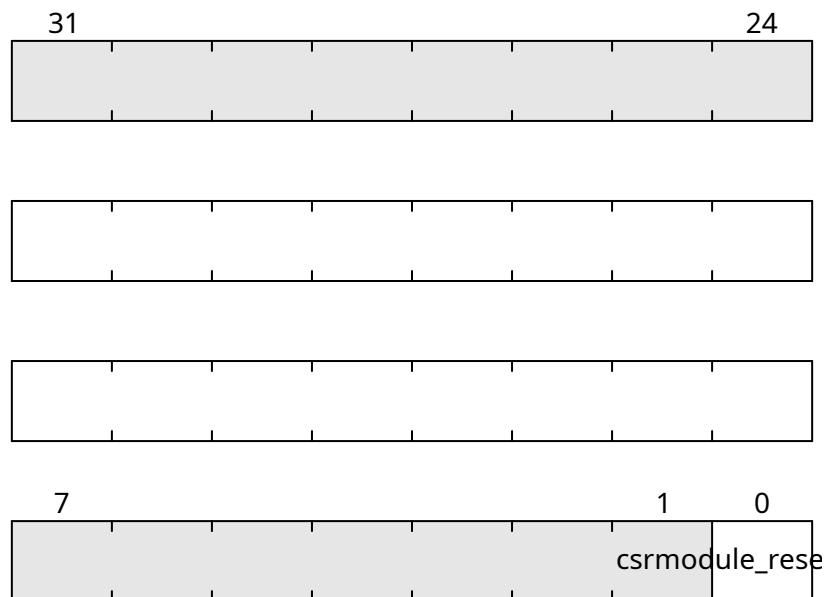


Fig. 23.10: DDRPHY_CSRMODULE_RESET_ALERT

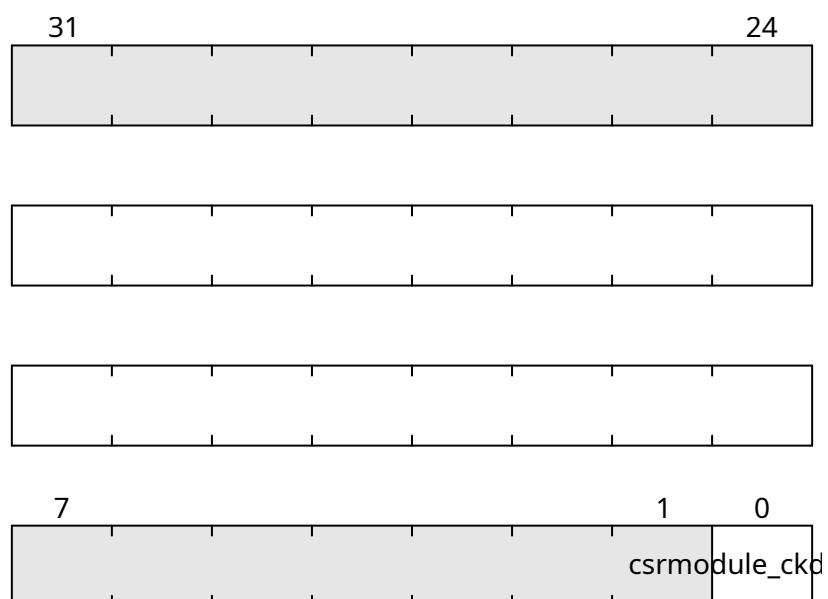


Fig. 23.11: DDRPHY_CSRMODULE_CKDLY_RST

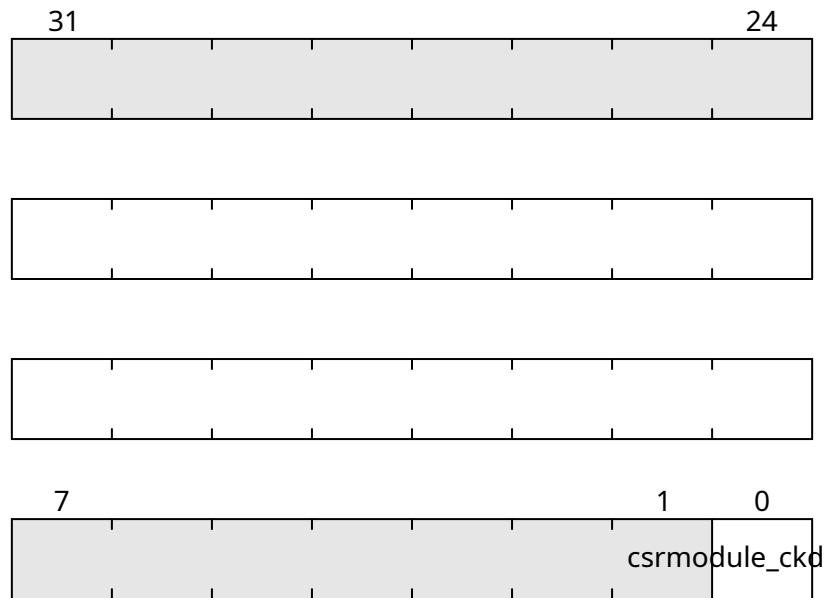


Fig. 23.12: DDRPHY_CSRMODULE_CKDLY_INC

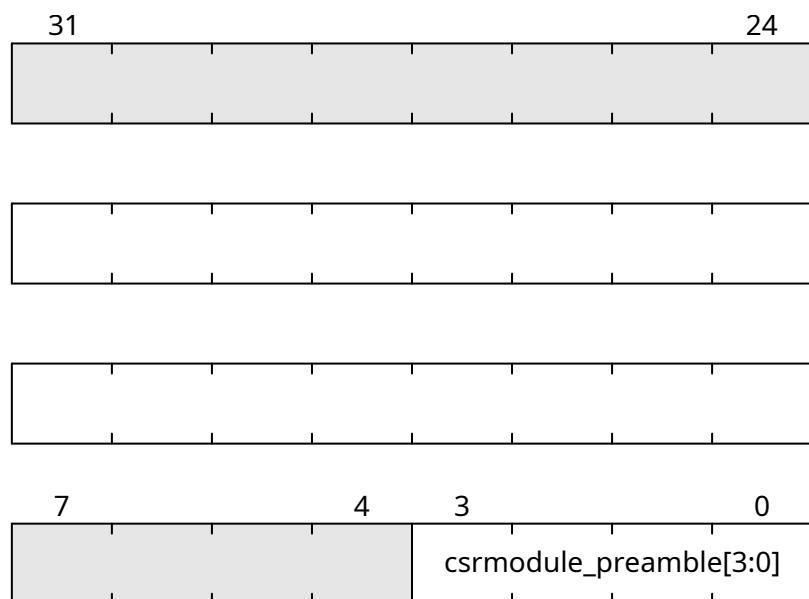


Fig. 23.13: DDRPHY_CSRMODULE_PREAMBLE

DDRPHY_CSRMODULE_WLEVEL_EN

Address: $0xf0000800 + 0x30 = 0xf0000830$

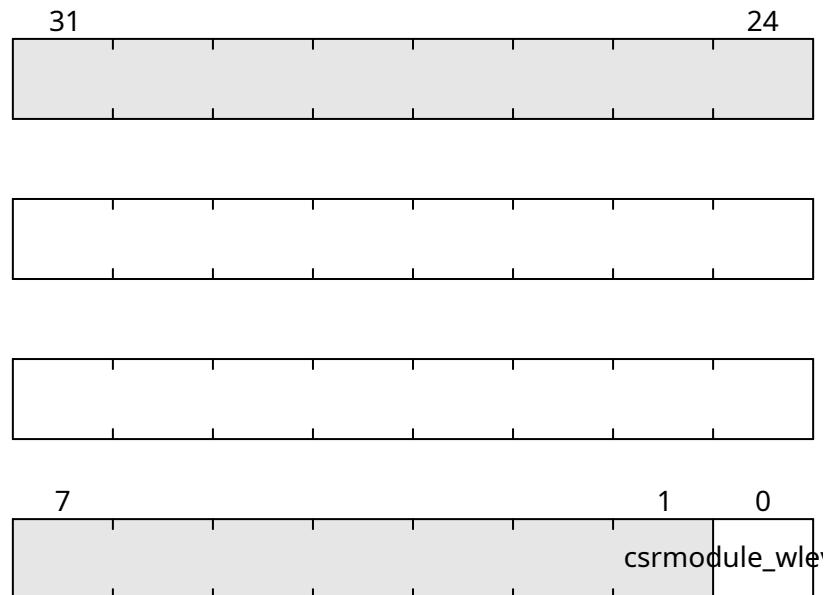


Fig. 23.14: DDRPHY_CSRMODULE_WLEVEL_EN

DDRPHY_CSRMODULE_PAR_ENABLE

Address: $0xf0000800 + 0x34 = 0xf0000834$

DDRPHY_CSRMODULE_PAR_VALUE

Address: $0xf0000800 + 0x38 = 0xf0000838$

DDRPHY_CSRMODULE_DISCARD_RD_FIFO

Address: $0xf0000800 + 0x3c = 0xf000083c$

DDRPHY_CSRMODULE_DLY_SEL

Address: $0xf0000800 + 0x40 = 0xf0000840$

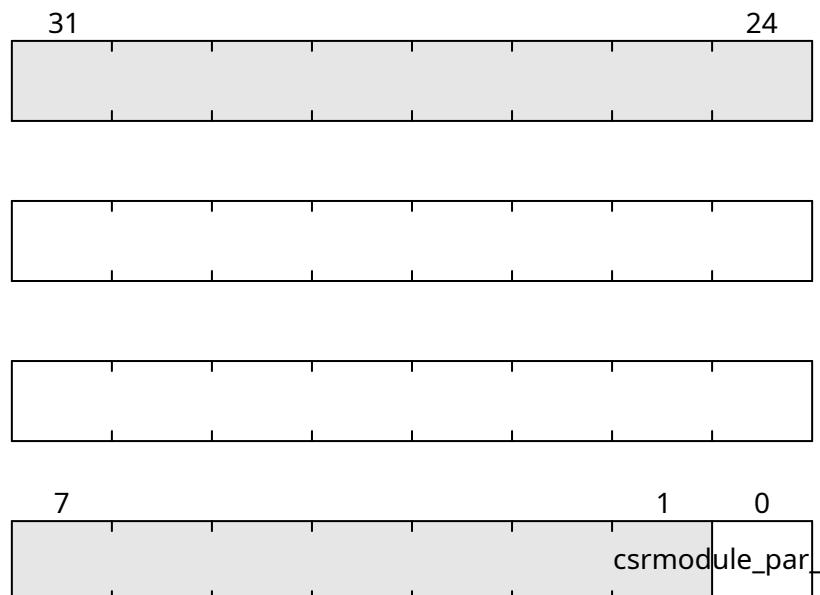


Fig. 23.15: DDRPHY_CSRMODULE_PAR_ENABLE

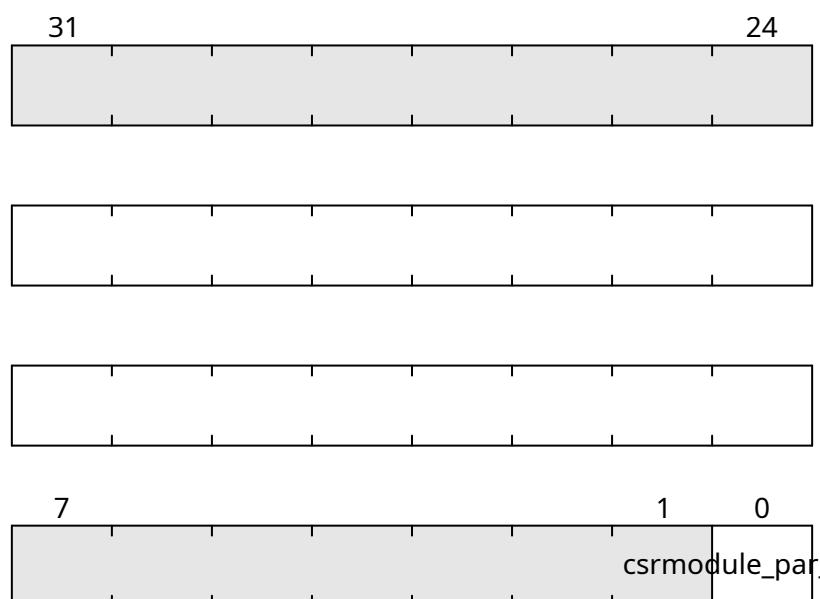


Fig. 23.16: DDRPHY_CSRMODULE_PAR_VALUE

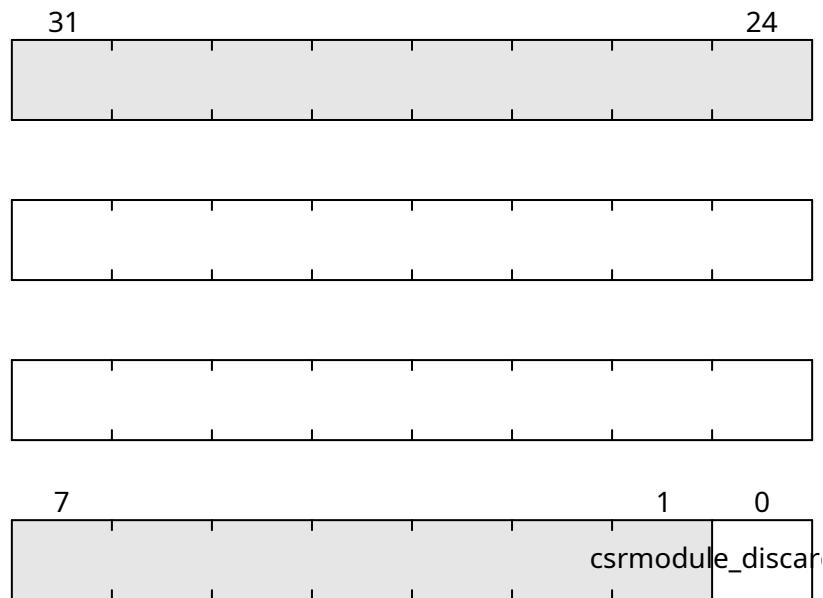


Fig. 23.17: DDRPHY_CSRMODULE_DISCARD_RD_FIFO

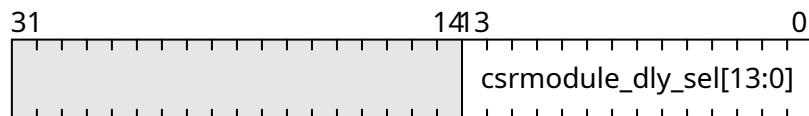


Fig. 23.18: DDRPHY_CSRMODULE_DLY_SEL

DDRPHY_CSRMODULE_DQ_DQS_RATIO

Address: 0xf0000800 + 0x44 = 0xf0000844

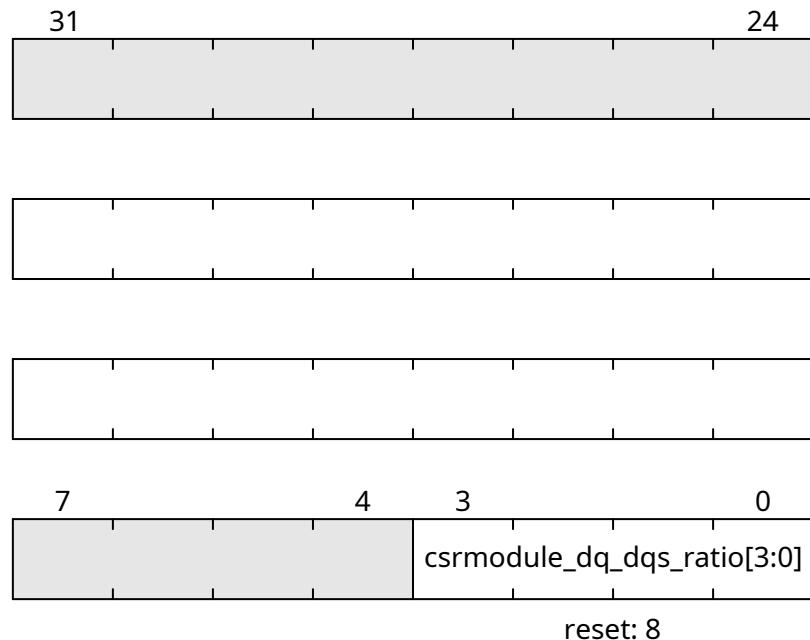


Fig. 23.19: DDRPHY_CSRMODULE_DQ_DQS_RATIO

DDRPHY_CSRMODULE_CK_RDLY_INC

Address: 0xf0000800 + 0x48 = 0xf0000848

DDRPHY_CSRMODULE_CK_RDLY_RST

Address: 0xf0000800 + 0x4c = 0xf000084c

DDRPHY_CSRMODULE_CK_RDDLY

Address: 0xf0000800 + 0x50 = 0xf0000850

DDRPHY_CSRMODULE_CK_RDDLY_PREAMBLE

Address: 0xf0000800 + 0x54 = 0xf0000854

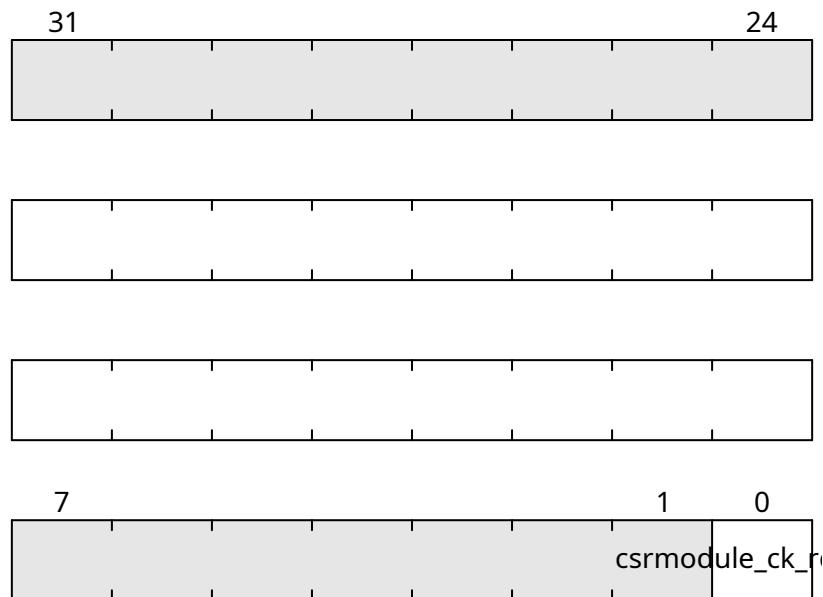


Fig. 23.20: `DDRPHY_CSRMODULE_CK_RDLY_INC`

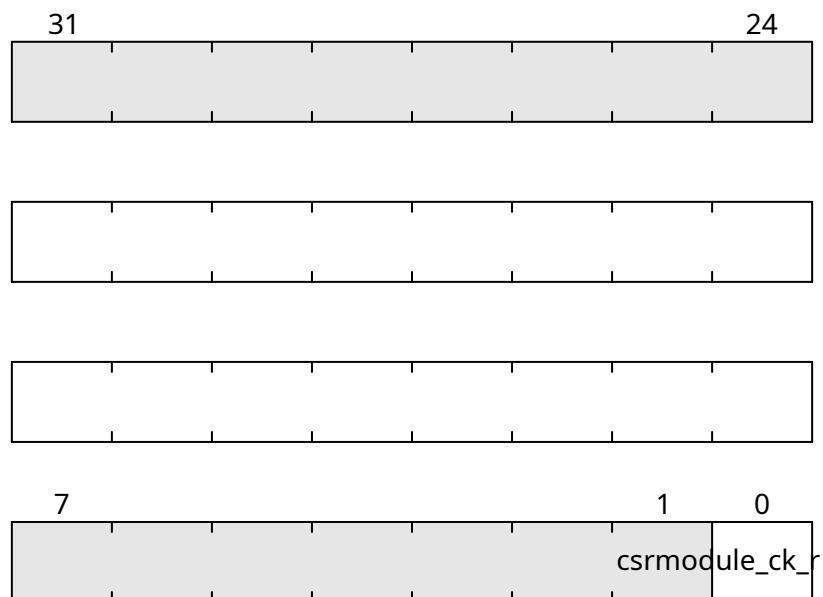


Fig. 23.21: `DDRPHY_CSRMODULE_CK_RDLY_RST`

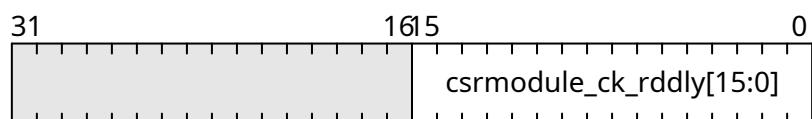


Fig. 23.22: `DDRPHY_CSRMODULE_CK_RDDLY`

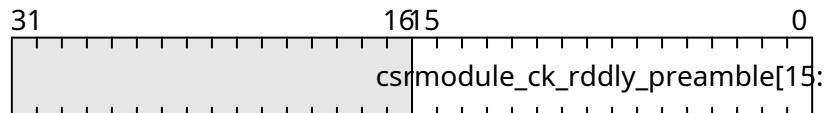


Fig. 23.23: DDRPHY_CSRMODULE_CK_RDDLY_PREAMBLE

DDRPHY_CSRMODULE_CK_WDLY_INC

Address: $0xf0000800 + 0x58 = 0xf0000858$

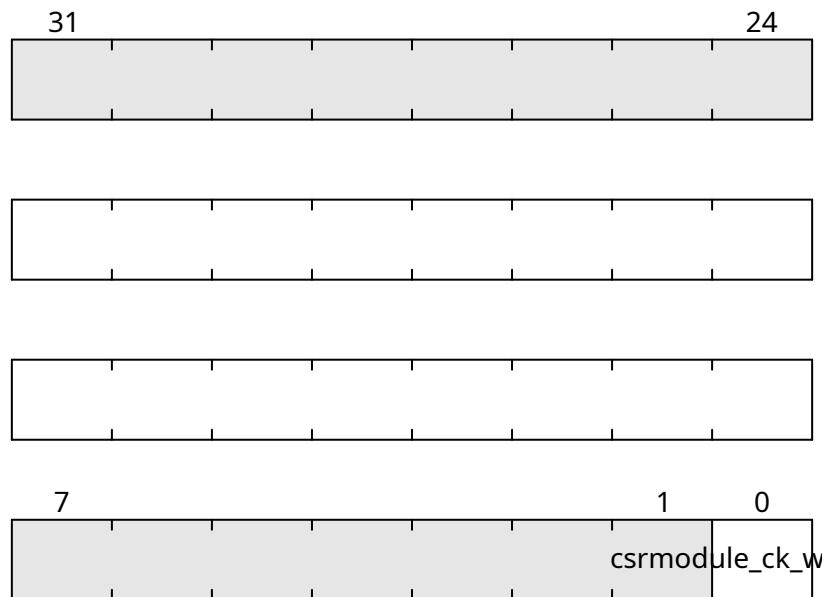


Fig. 23.24: DDRPHY_CSRMODULE_CK_WDLY_INC

DDRPHY_CSRMODULE_CK_WDLY_RST

Address: $0xf0000800 + 0x5c = 0xf000085c$

DDRPHY_CSRMODULE_CK_WDLY_DQS

Address: $0xf0000800 + 0x60 = 0xf0000860$

DDRPHY_CSRMODULE_CK_WDDLY_INC

Address: $0xf0000800 + 0x64 = 0xf0000864$

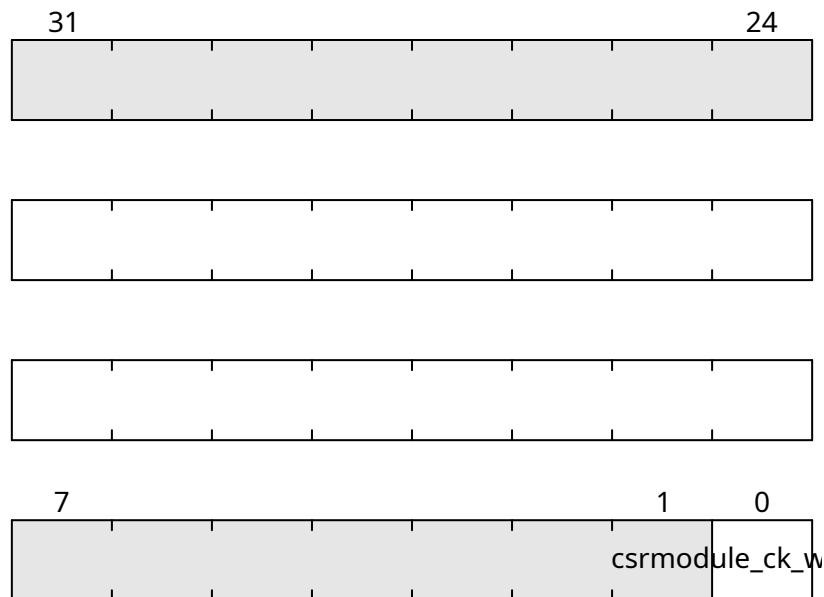


Fig. 23.25: `DDRPHY_CSRMODULE_CK_WDLY_RST`

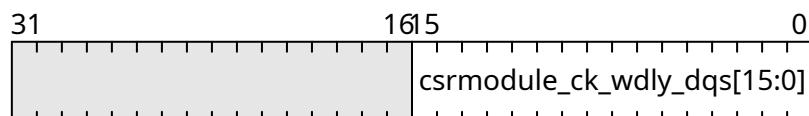


Fig. 23.26: `DDRPHY_CSRMODULE_CK_WDLY_DQS`

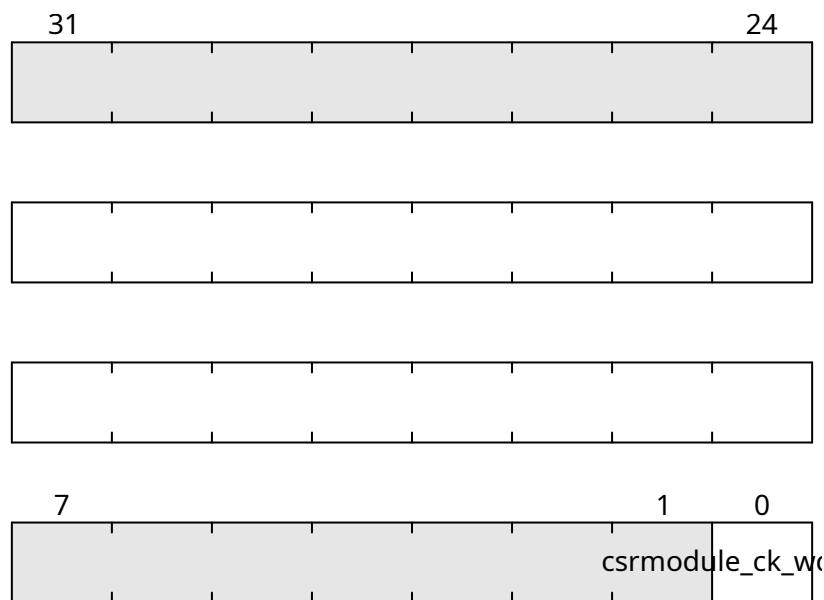


Fig. 23.27: `DDRPHY_CSRMODULE_CK_WDDLY_INC`

DDRPHY_CSRMODULE_CK_WDDLY_RST

Address: $0xf0000800 + 0x68 = 0xf0000868$

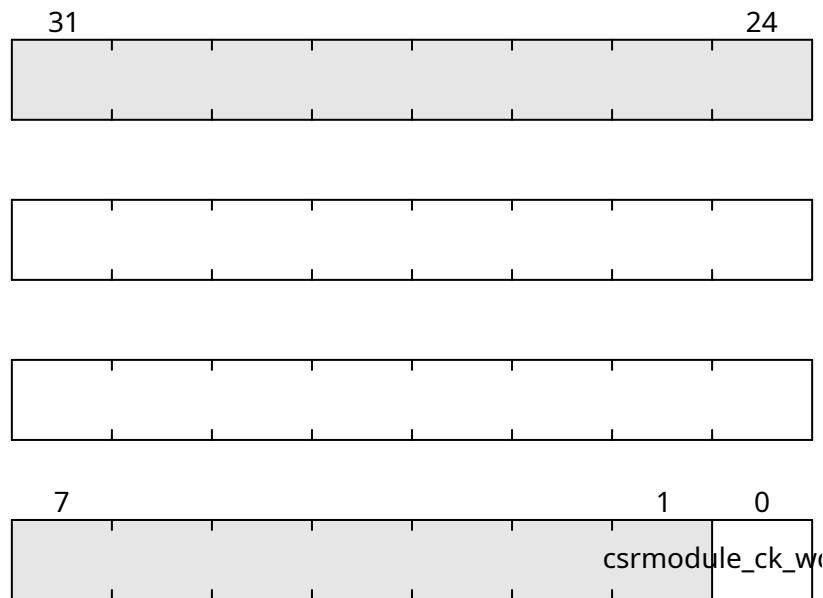


Fig. 23.28: DDRPHY_CSRMODULE_CK_WDDLY_RST

DDRPHY_CSRMODULE_CK_WDLY_DQ

Address: $0xf0000800 + 0x6c = 0xf000086c$

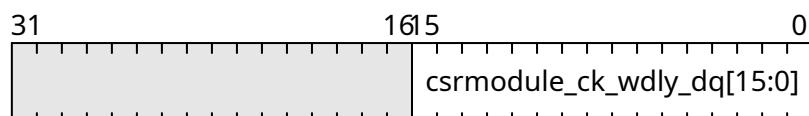


Fig. 23.29: DDRPHY_CSRMODULE_CK_WDLY_DQ

DDRPHY_CSRMODULE_DQ_DLY_SEL

Address: $0xf0000800 + 0x70 = 0xf0000870$

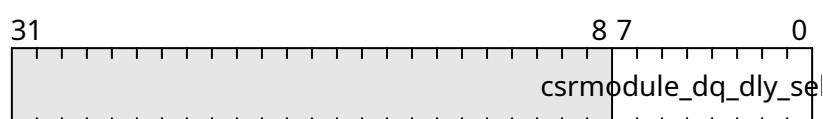


Fig. 23.30: DDRPHY_CSRMODULE_DQ_DLY_SEL

DDRPHY_CSRMODULE_CSDLY_RST

Address: $0xf0000800 + 0x74 = 0xf0000874$

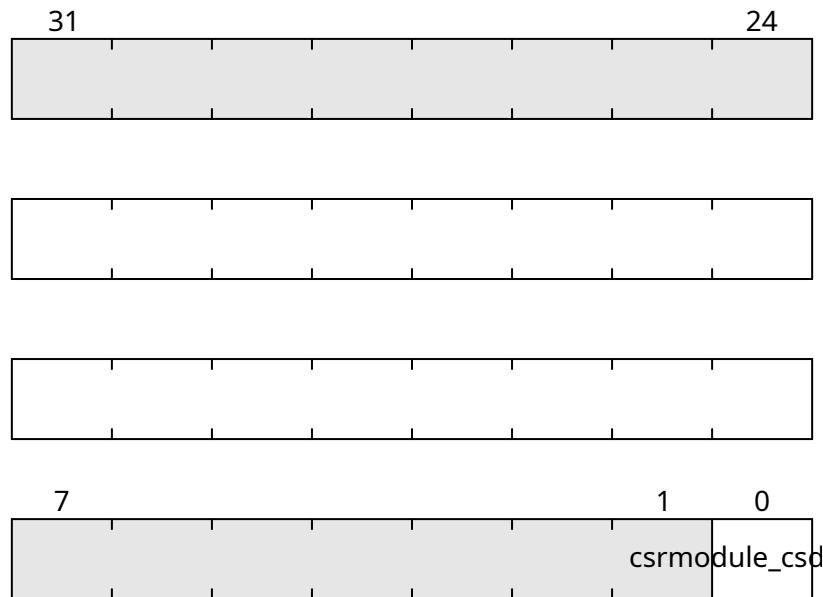


Fig. 23.31: DDRPHY_CSRMODULE_CSDLY_RST

DDRPHY_CSRMODULE_CSDLY_INC

Address: $0xf0000800 + 0x78 = 0xf0000878$

DDRPHY_CSRMODULE_CADLY_RST

Address: $0xf0000800 + 0x7c = 0xf000087c$

DDRPHY_CSRMODULE_CADLY_INC

Address: $0xf0000800 + 0x80 = 0xf0000880$

DDRPHY_CSRMODULE_PARDLY_RST

Address: $0xf0000800 + 0x84 = 0xf0000884$

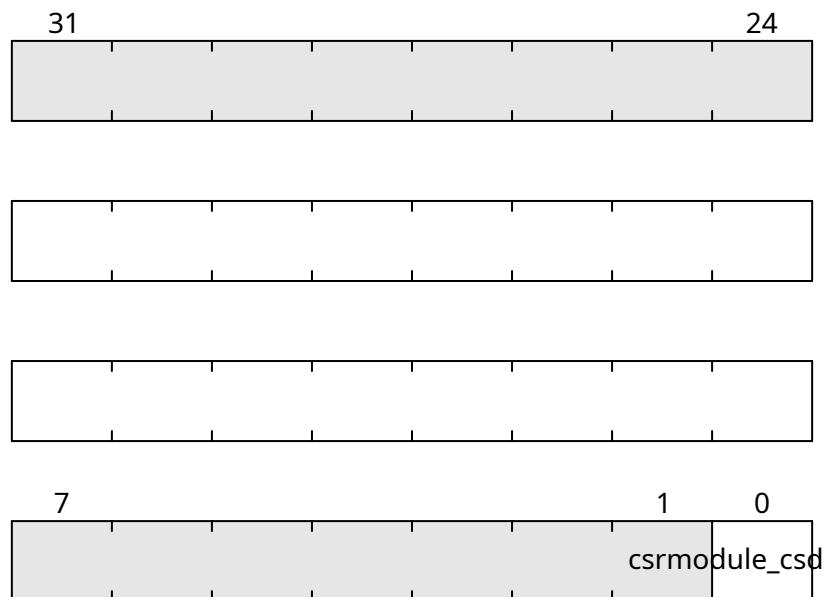


Fig. 23.32: `DDRPHY_CSRMODULE_CSDLY_INC`

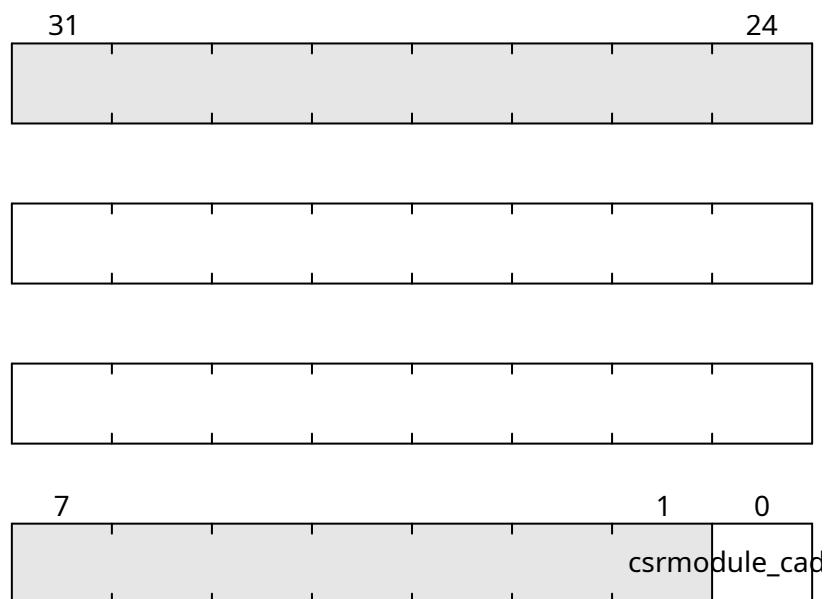


Fig. 23.33: `DDRPHY_CSRMODULE_CADLY_RST`

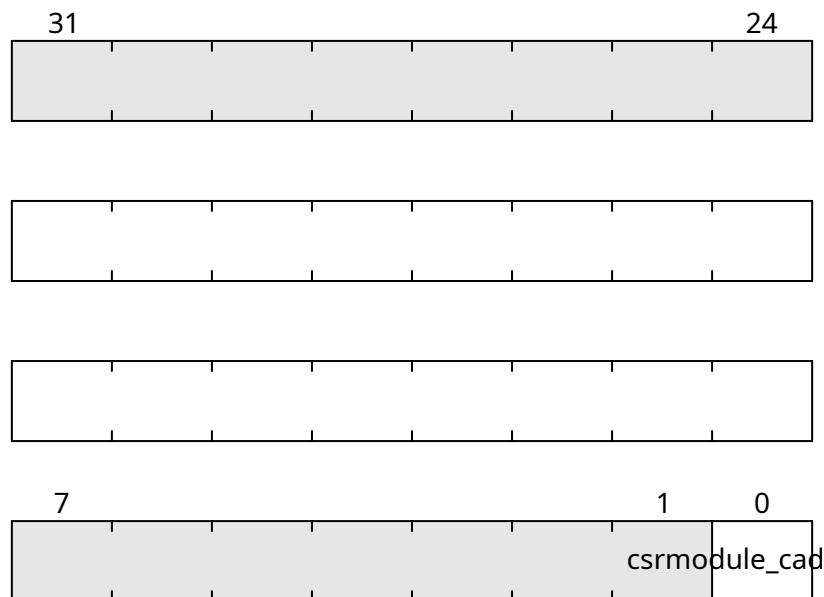


Fig. 23.34: `DDRPHY_CSRMODULE_CADLY_INC`

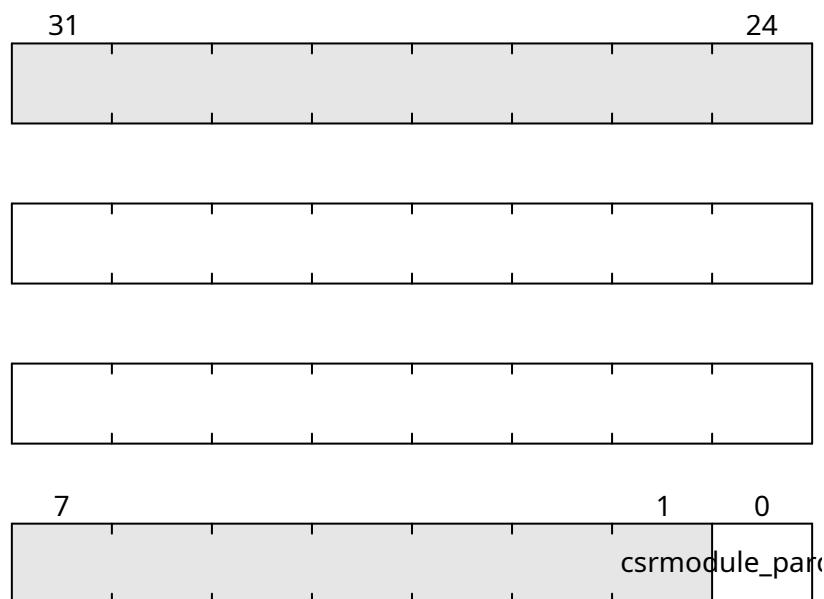


Fig. 23.35: `DDRPHY_CSRMODULE_PARDLY_RST`

DDRPHY_CSRMODULE_PARDLY_INC

Address: $0xf0000800 + 0x88 = 0xf0000888$

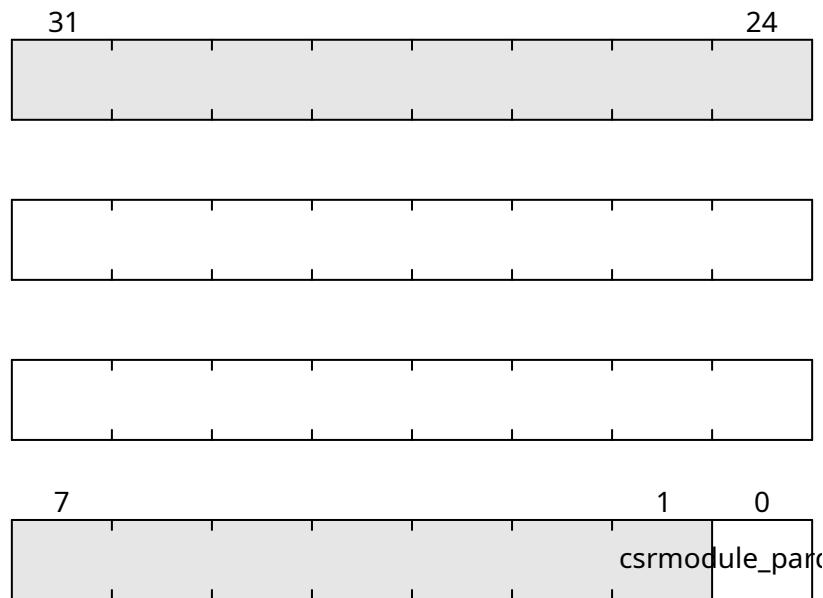


Fig. 23.36: DDRPHY_CSRMODULE_PARDLY_INC

DDRPHY_CSRMODULE_CSDLY

Address: $0xf0000800 + 0x8c = 0xf000088c$

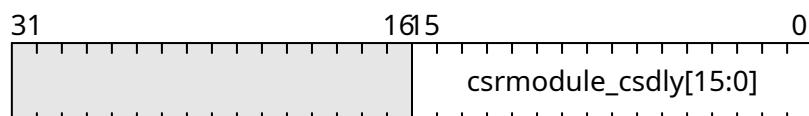


Fig. 23.37: DDRPHY_CSRMODULE_CSDLY

DDRPHY_CSRMODULE_CADLY

Address: $0xf0000800 + 0x90 = 0xf0000890$

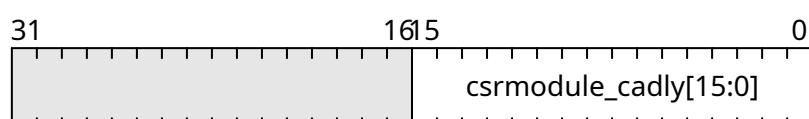


Fig. 23.38: DDRPHY_CSRMODULE_CADLY

DDRPHY_CSRMODULE_RDLY_DQ_RST

Address: $0xf0000800 + 0x94 = 0xf0000894$

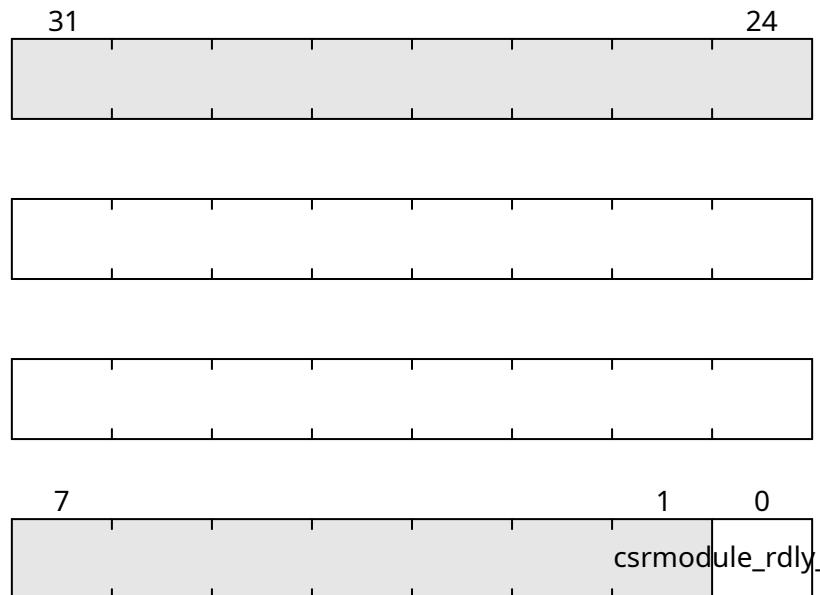


Fig. 23.39: DDRPHY_CSRMODULE_RDLY_DQ_RST

DDRPHY_CSRMODULE_RDLY_DQ_INC

Address: $0xf0000800 + 0x98 = 0xf0000898$

DDRPHY_CSRMODULE_RDLY_DQS_RST

Address: $0xf0000800 + 0x9c = 0xf000089c$

DDRPHY_CSRMODULE_RDLY_DQS_INC

Address: $0xf0000800 + 0xa0 = 0xf00008a0$

DDRPHY_CSRMODULE_RDLY_DQS

Address: $0xf0000800 + 0xa4 = 0xf00008a4$

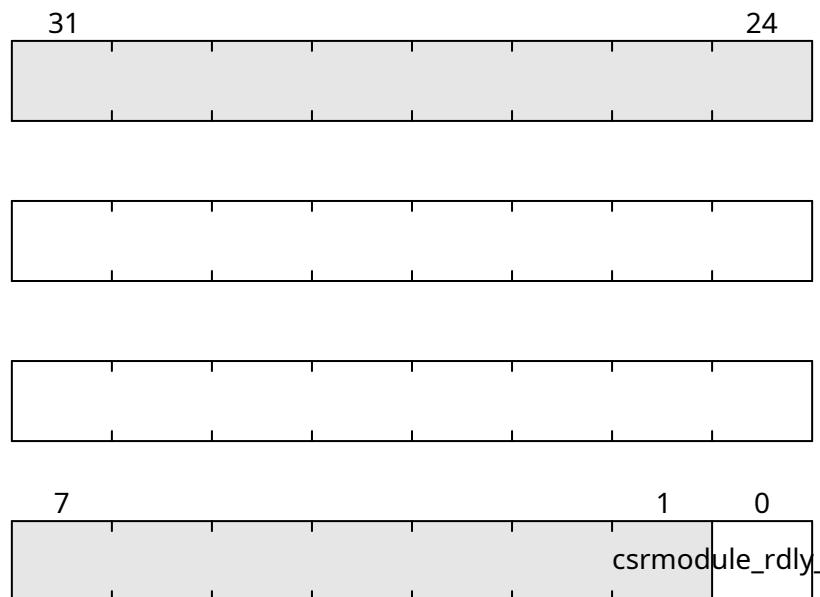


Fig. 23.40: DDRPHY_CSRMODULE_RDLY_DQ_INC

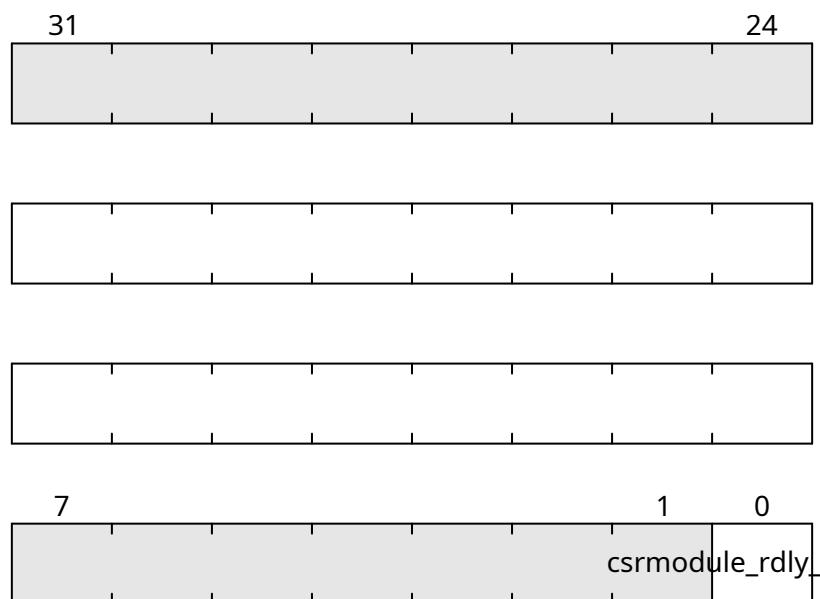


Fig. 23.41: DDRPHY_CSRMODULE_RDLY_DQS_RST

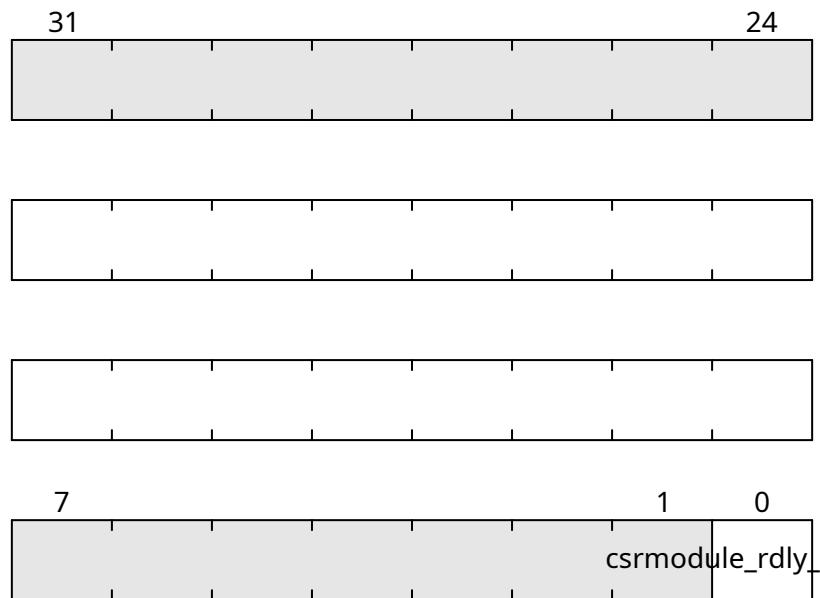


Fig. 23.42: `DDRPHY_CSRMODULE_RDLY_DQS_INC`

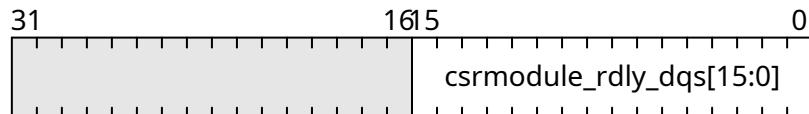


Fig. 23.43: `DDRPHY_CSRMODULE_RDLY_DQS`

DDRPHY_CSRMODULE_RDLY_DQ

Address: $0xf0000800 + 0xa8 = 0xf00008a8$

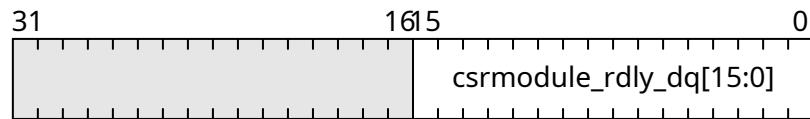


Fig. 23.44: DDRPHY_CSRMODULE_RDLY_DQ

DDRPHY_CSRMODULE_WDLY_DQ_RST

Address: $0xf0000800 + 0xac = 0xf00008ac$

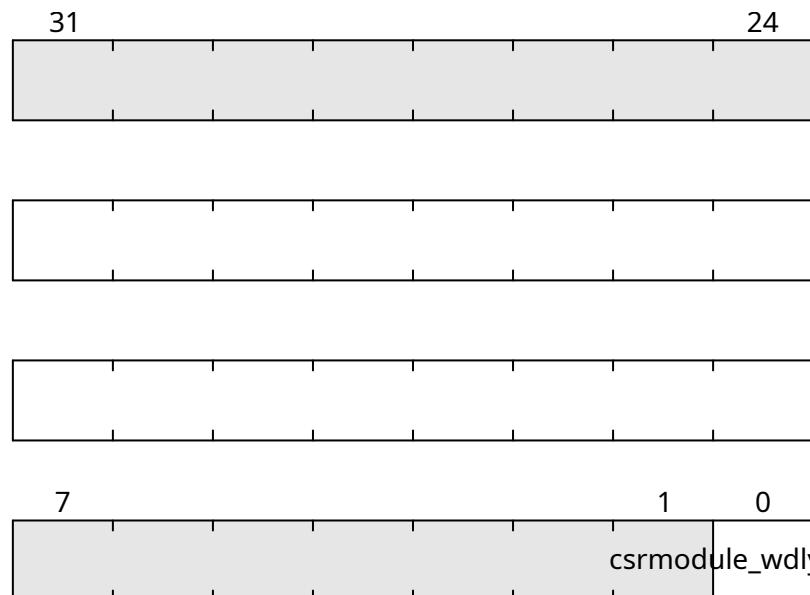


Fig. 23.45: DDRPHY_CSRMODULE_WDLY_DQ_RST

DDRPHY_CSRMODULE_WDLY_DQ_INC

Address: $0xf0000800 + 0xb0 = 0xf00008b0$

DDRPHY_CSRMODULE_WDLY_DM_RST

Address: $0xf0000800 + 0xb4 = 0xf00008b4$

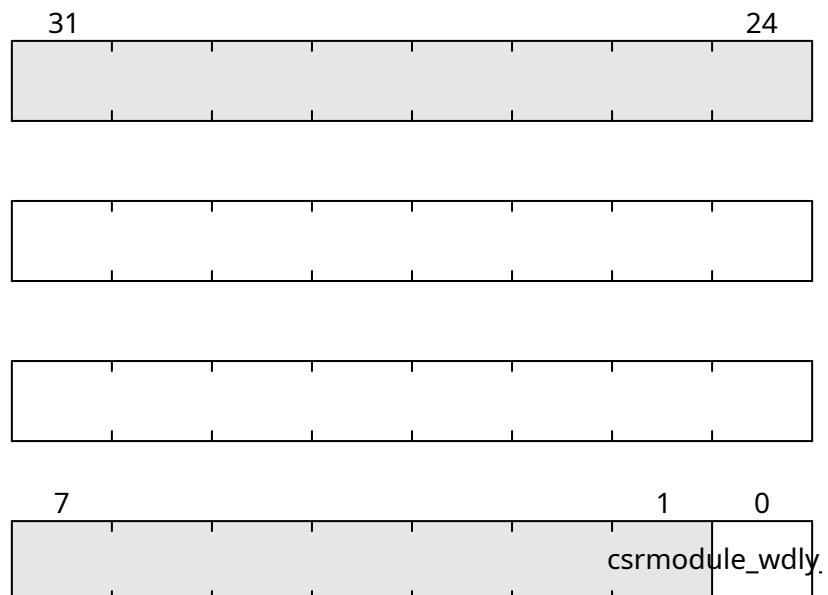


Fig. 23.46: `DDRPHY_CSRMODULE_WDLY_DQ_INC`

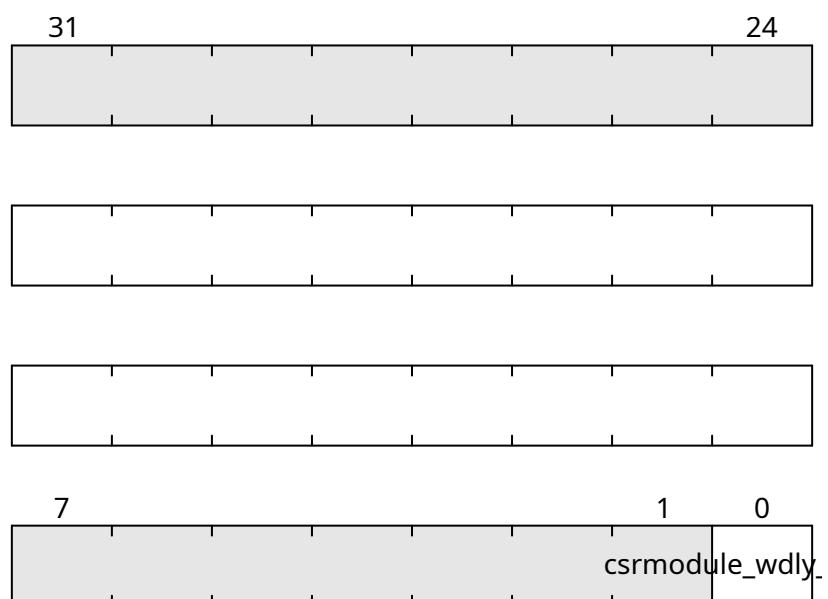


Fig. 23.47: `DDRPHY_CSRMODULE_WDLY_DM_RST`

DDRPHY_CSRMODULE_WDLY_DM_INC

Address: $0xf0000800 + 0xb8 = 0xf00008b8$

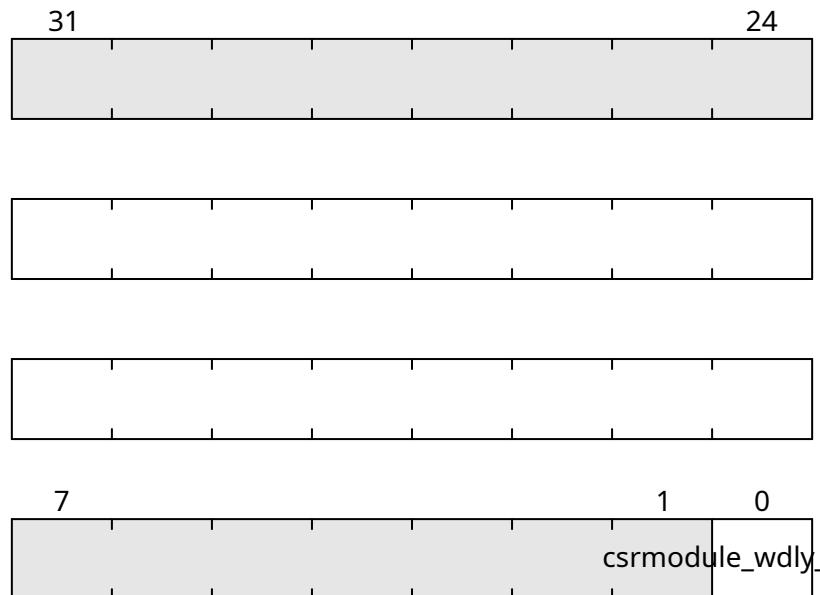


Fig. 23.48: DDRPHY_CSRMODULE_WDLY_DM_INC

DDRPHY_CSRMODULE_WDLY_DQS_RST

Address: $0xf0000800 + 0xbc = 0xf00008bc$

DDRPHY_CSRMODULE_WDLY_DQS_INC

Address: $0xf0000800 + 0xc0 = 0xf00008c0$

DDRPHY_CSRMODULE_WDLY_DQS

Address: $0xf0000800 + 0xc4 = 0xf00008c4$

DDRPHY_CSRMODULE_WDLY_DQ

Address: $0xf0000800 + 0xc8 = 0xf00008c8$

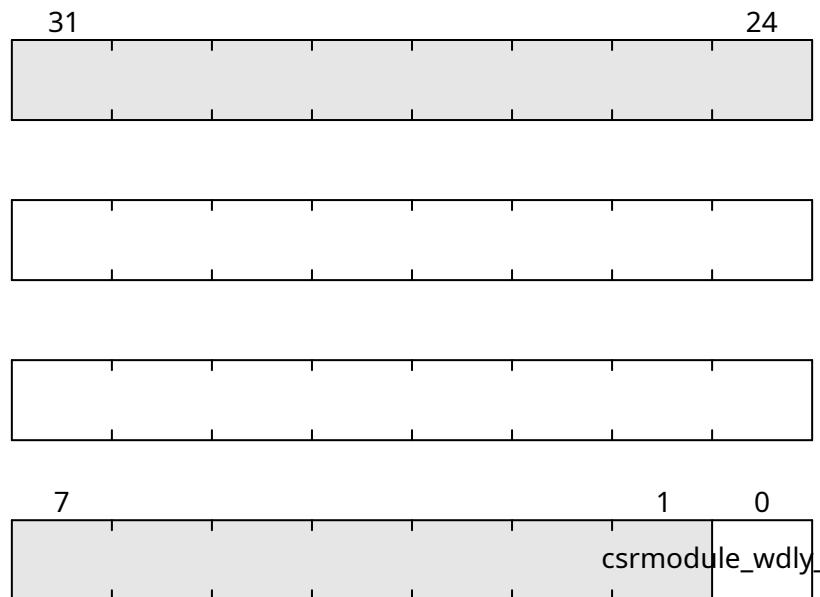


Fig. 23.49: `DDRPHY_CSRMODULE_WDLY_DQS_RST`

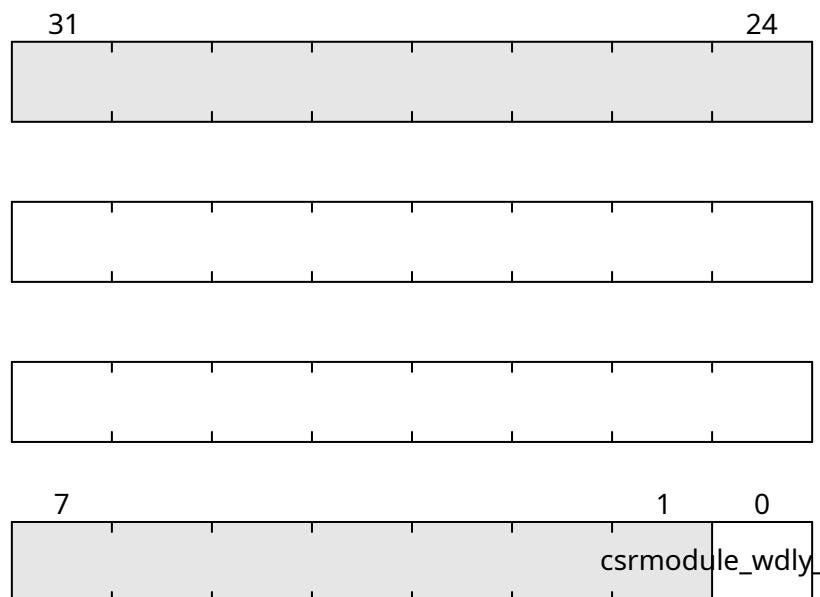


Fig. 23.50: `DDRPHY_CSRMODULE_WDLY_DQS_INC`

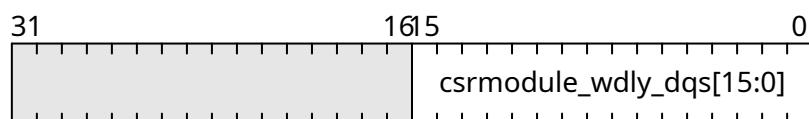


Fig. 23.51: `DDRPHY_CSRMODULE_WDLY_DQS`

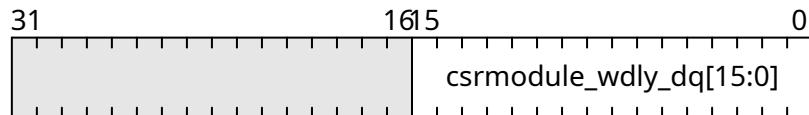


Fig. 23.52: DDRPHY_CSRMODULE_WDLY_DQ

DDRPHY_CSRMODULE_WDLY_DM

Address: 0xf0000800 + 0xcc = 0xf00008cc

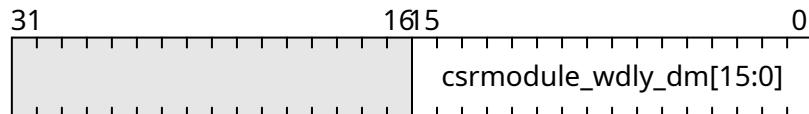


Fig. 23.53: DDRPHY_CSRMODULE_WDLY_DM

23.2.3 CONTROLLER_SETTINGS

Allows to change LiteDRAMController behaviour at runtime

Register Listing for CONTROLLER_SETTINGS

Register	Address
<i>CONTROLLER_SETTINGS_REFRESH</i>	<i>0xf0001000</i>

CONTROLLER_SETTINGS_REFRESH

Address: 0xf0001000 + 0x0 = 0xf0001000

Enable/disable Refresh commands sending

23.2.4 DDRCTRL

Register Listing for DDRCTRL

Register	Address
<i>DDRCTRL_INIT_DONE</i>	<i>0xf0001800</i>
<i>DDRCTRL_INIT_ERROR</i>	<i>0xf0001804</i>

DDRCTRL_INIT_DONE

Address: 0xf0001800 + 0x0 = 0xf0001800

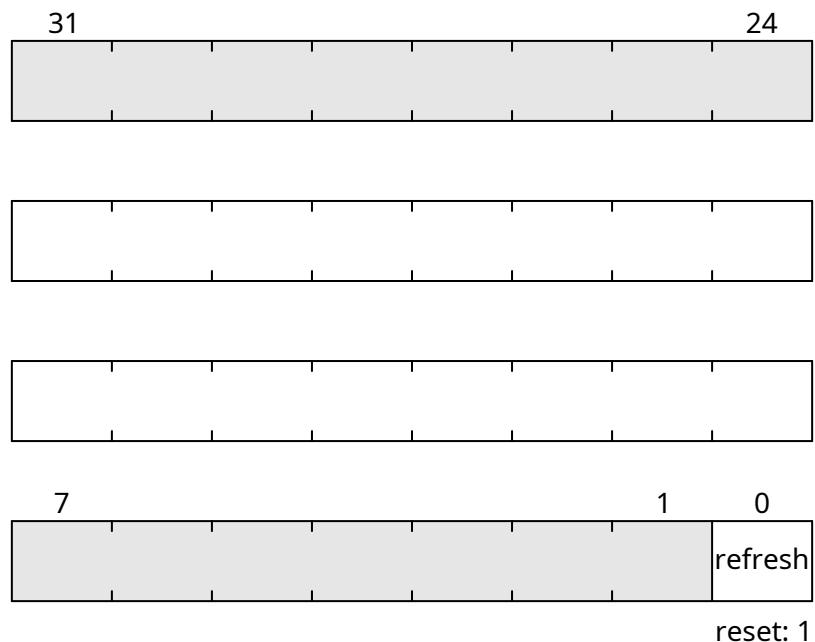


Fig. 23.54: CONTROLLER_SETTINGS_REFRESH

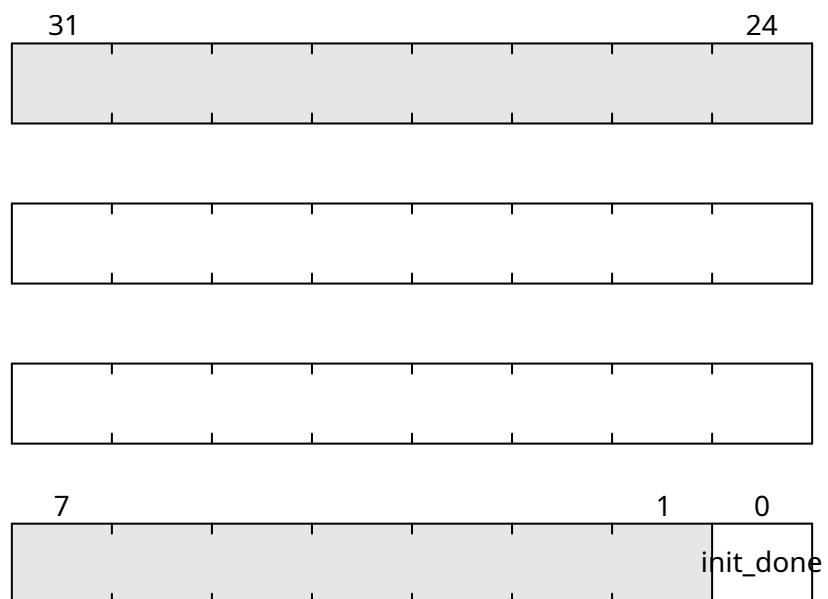


Fig. 23.55: DDRCTRL_INIT_DONE

DDRCTRL_INIT_ERROR

Address: $0xf0001800 + 0x4 = 0xf0001804$



Fig. 23.56: DDRCTRL_INIT_ERROR

23.2.5 ROWHAMMER

Row Hammer DMA attacker

This module allows to perform a Row Hammer attack by configuring it with two addresses that map to different rows of a single bank. When enabled, it will perform alternating DMA reads from the given locations, which will result in the DRAM controller having to repeatedly open/close rows at each read access.

Register Listing for ROWHAMMER

Register	Address
<i>ROWHAMMER_ENABLED</i>	<i>0xf0002000</i>
<i>ROWHAMMER_ADDRESS1</i>	<i>0xf0002004</i>
<i>ROWHAMMER_ADDRESS2</i>	<i>0xf0002008</i>
<i>ROWHAMMER_COUNT</i>	<i>0xf000200c</i>

ROWHAMMER_ENABLED

Address: $0xf0002000 + 0x0 = 0xf0002000$

Used to start/stop the operation of the module



Fig. 23.57: ROWHAMMER_ENABLED

ROWHAMMER_ADDRESS1

Address: 0xf0002000 + 0x4 = 0xf0002004

First attacked address

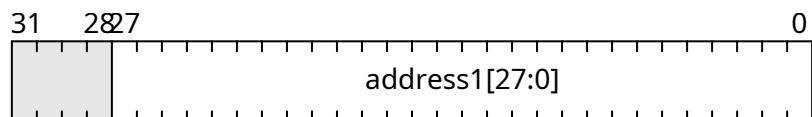


Fig. 23.58: ROWHAMMER_ADDRESS1

ROWHAMMER_ADDRESS2

Address: 0xf0002000 + 0x8 = 0xf0002008

Second attacked address

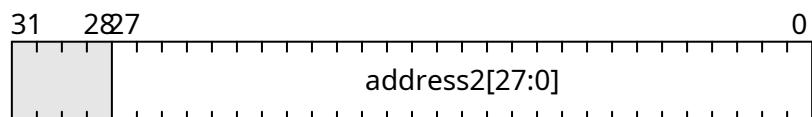


Fig. 23.59: ROWHAMMER_ADDRESS2

ROWHAMMER_COUNT

Address: $0xf0002000 + 0xc = 0xf000200c$

This is the number of DMA accesses performed. When the module is enabled, the value can be freely read. When the module is disabled, the register is clear-on-write and has to be read before the next attack.

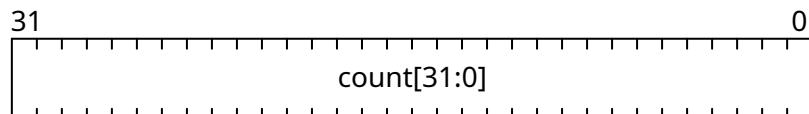


Fig. 23.60: ROWHAMMER_COUNT

23.2.6 WRITER

DMA DRAM writer.

Allows to fill DRAM with a predefined pattern using DMA.

Pattern

Provides access to RAM to store access pattern: *mem_addr* and *mem_data*. The pattern address space can be limited using the *data_mask*.

For example, having *mem_addr* filled with $[0x04, 0x02, 0x03, \dots]$ and *mem_data* filled with $[0xff, 0xaa, 0x55, \dots]$ and setting *data_mask* = *0b01*, the pattern $[(address, data), \dots]$ written will be: $[(0x04, 0xff), (0x02, 0xaa), (0x04, 0xff), \dots]$ (wraps due to masking).

DRAM memory range that is being accessed can be configured using *mem_mask*.

To use this module, make sure that *ready* is 1, then write the desired number of transfers to *count*. Writing to the *start* CSR will initialize the operation. When the operation is ongoing *ready* will be 0.

Register Listing for WRITER

Register	Address
WRITER_START	0xf0002800
WRITER_READY	0xf0002804
WRITER_MODULO	0xf0002808
WRITER_COUNT	0xf000280c
WRITER_DONE	0xf0002810
WRITER_MEM_MASK	0xf0002814
WRITER_DATA_MASK	0xf0002818
WRITER_DATA_DIV	0xf000281c
WRITER_INVERTER_DIVISOR_MASK	0xf0002820
WRITER_INVERTER_SELECTION_MASK	0xf0002824
WRITER_LAST_ADDRESS	0xf0002828

WRITER_START

Address: $0xf0002800 + 0x0 = 0xf0002800$

Write to the register starts the transfer (if ready=1)

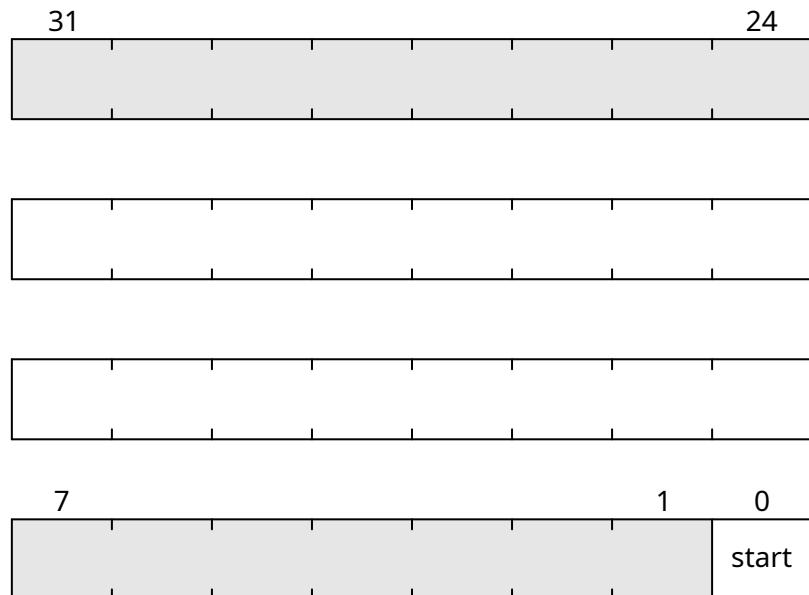


Fig. 23.61: WRITER_START

WRITER_READY

Address: $0xf0002800 + 0x4 = 0xf0002804$

Indicates that the transfer is not ongoing

WRITER_MODULO

Address: $0xf0002800 + 0x8 = 0xf0002808$

When set use modulo to calculate DMA transfers address rather than bit masking

WRITER_COUNT

Address: $0xf0002800 + 0xc = 0xf000280c$

Desired number of DMA transfers

WRITER_DONE

Address: $0xf0002800 + 0x10 = 0xf0002810$

Number of completed DMA transfers



Fig. 23.62: WRITER_READY



Fig. 23.63: WRITER_MODULO

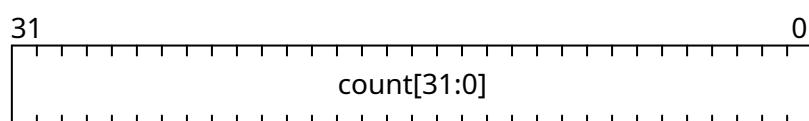


Fig. 23.64: WRITER_COUNT

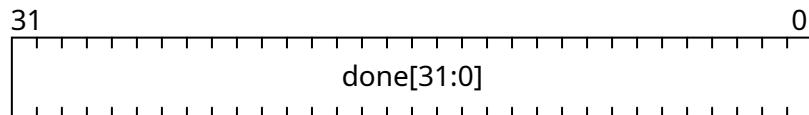


Fig. 23.65: WRITER_DONE

WRITER_MEM_MASK

Address: $0xf0002800 + 0x14 = 0xf0002814$

DRAM address mask for DMA transfers

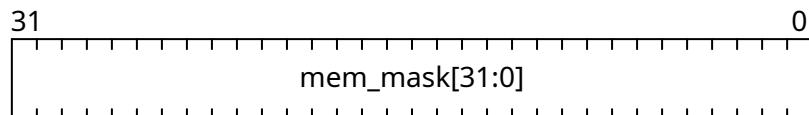


Fig. 23.66: WRITER_MEM_MASK

WRITER_DATA_MASK

Address: $0xf0002800 + 0x18 = 0xf0002818$

Pattern memory address mask

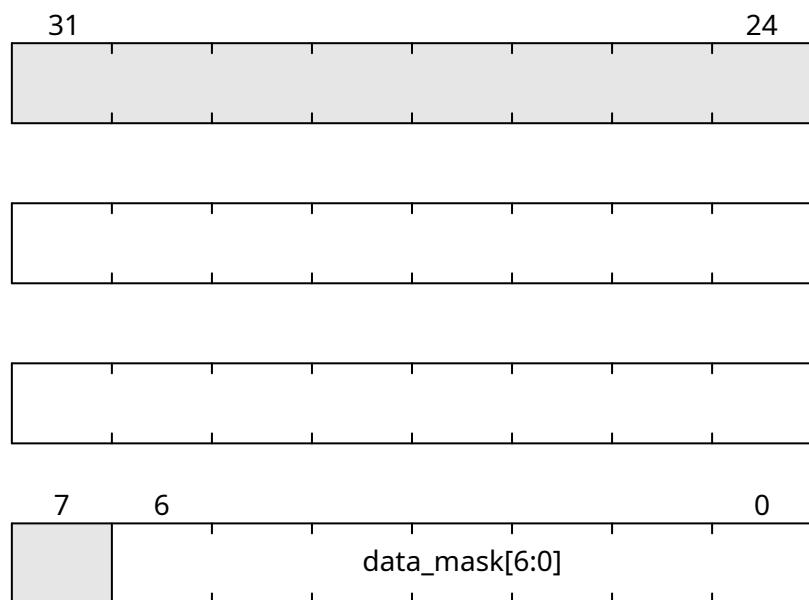


Fig. 23.67: WRITER_DATA_MASK

WRITER_DATA_DIV

Address: $0xf0002800 + 0x1c = 0xf000281c$

Pattern memory address divisor-1

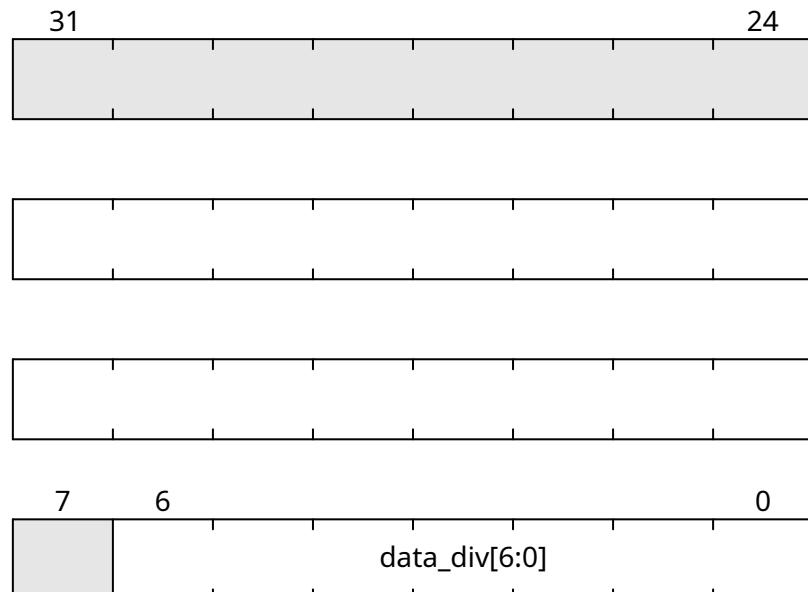


Fig. 23.68: WRITER_DATA_DIV

WRITER_INVERTER_DIVISOR_MASK

Address: $0xf0002800 + 0x20 = 0xf0002820$

Divisor mask for selecting rows for which pattern data gets inverted

WRITER_INVERTER_SELECTION_MASK

Address: $0xf0002800 + 0x24 = 0xf0002824$

Selection mask for selecting rows for which pattern data gets inverted

WRITER_LAST_ADDRESS

Address: $0xf0002800 + 0x28 = 0xf0002828$

Number of completed DMA transfers

23.2.7 READER

DMA DRAM reader.

Allows to check DRAM contents against a predefined pattern using DMA.

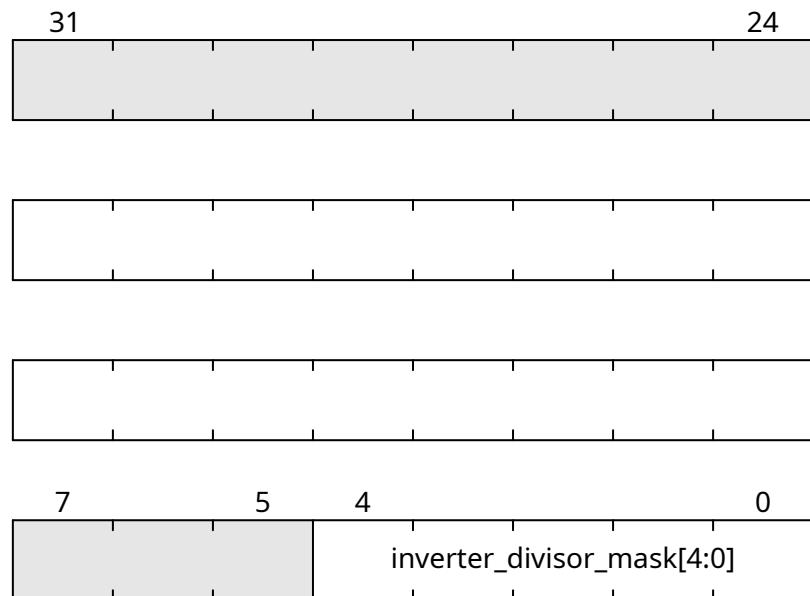


Fig. 23.69: WRITER_INVERTER_DIVISOR_MASK

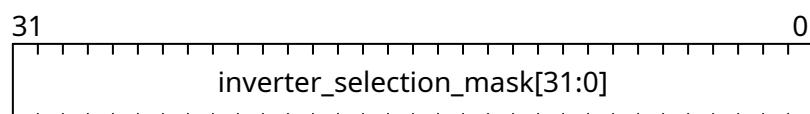


Fig. 23.70: WRITER_INVERTER_SELECTION_MASK

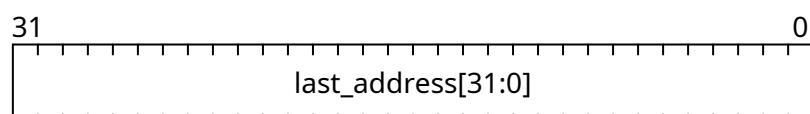


Fig. 23.71: WRITER_LAST_ADDRESS

Pattern

Provides access to RAM to store access pattern: *mem_addr* and *mem_data*. The pattern address space can be limited using the *data_mask*.

For example, having *mem_addr* filled with [0x04, 0x02, 0x03, ...] and *mem_data* filled with [0xff, 0xaa, 0x55, ...] and setting *data_mask* = 0b01, the pattern [(address, data), ...] written will be: [(0x04, 0xff), (0x02, 0xaa), (0x04, 0xff), ...] (wraps due to masking).

DRAM memory range that is being accessed can be configured using *mem_mask*.

To use this module, make sure that *ready* is 1, then write the desired number of transfers to *count*. Writing to the *start* CSR will initialize the operation. When the operation is ongoing *ready* will be 0.

Reading errors

This module allows to check the locations of errors in the memory. It scans the configured memory area and compares the values read to the predefined pattern. If *skip_fifo* is 0, this module will stop after each error encountered, so that it can be examined. Wait until the *error_ready* CSR is 1. Then use the CSRs *error_offset*, *error_data* and *error_expected* to examine the errors in the current transfer. To continue reading, write 1 to *error_continue* CSR. Setting *skip_fifo* to 1 will disable this behavior entirely.

The final number of errors can be read from *error_count*. NOTE: This value represents the number of erroneous *DMA transfers*.

The current progress can be read from the *done* CSR.

Register Listing for READER

Register	Address
<i>READER_START</i>	0xf0003000
<i>READER_READY</i>	0xf0003004
<i>READER_MODULO</i>	0xf0003008
<i>READER_COUNT</i>	0xf000300c
<i>READER_DONE</i>	0xf0003010
<i>READER_MEM_MASK</i>	0xf0003014
<i>READER_DATA_MASK</i>	0xf0003018
<i>READER_DATA_DIV</i>	0xf000301c
<i>READER_INVERTER_DIVISOR_MASK</i>	0xf0003020
<i>READER_INVERTER_SELECTION_MASK</i>	0xf0003024
<i>READER_ERROR_COUNT</i>	0xf0003028
<i>READER_SKIP_FIFO</i>	0xf000302c
<i>READER_ERROR_OFFSET</i>	0xf0003030
<i>READER_ERROR_DATA1</i>	0xf0003034
<i>READER_ERROR_DATA0</i>	0xf0003038
<i>READER_ERROR_EXPECTED1</i>	0xf000303c
<i>READER_ERROR_EXPECTED0</i>	0xf0003040
<i>READER_ERROR_READY</i>	0xf0003044
<i>READER_ERROR_CONTINUE</i>	0xf0003048

READER_START

Address: $0xf0003000 + 0x0 = 0xf0003000$

Write to the register starts the transfer (if ready=1)



Fig. 23.72: READER_START

READER_READY

Address: $0xf0003000 + 0x4 = 0xf0003004$

Indicates that the transfer is not ongoing

READER_MODULO

Address: $0xf0003000 + 0x8 = 0xf0003008$

When set use modulo to calculate DMA transfers address rather than bit masking

READER_COUNT

Address: $0xf0003000 + 0xc = 0xf000300c$

Desired number of DMA transfers

READER_DONE

Address: $0xf0003000 + 0x10 = 0xf0003010$

Number of completed DMA transfers



Fig. 23.73: READER_READY



Fig. 23.74: READER_MODULO

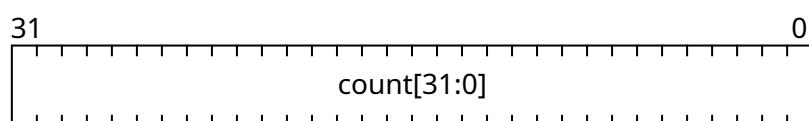


Fig. 23.75: READER_COUNT

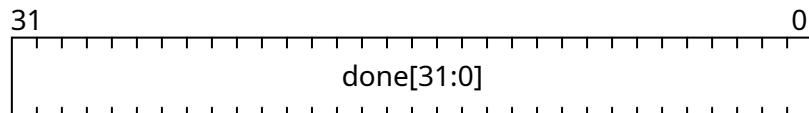


Fig. 23.76: READER_DONE

READER_MEM_MASK

Address: $0xf0003000 + 0x14 = 0xf0003014$

DRAM address mask for DMA transfers

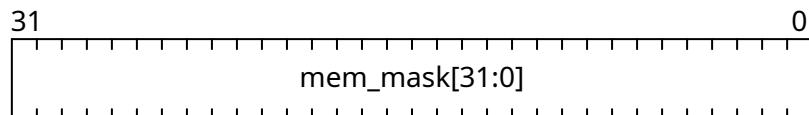


Fig. 23.77: READER_MEM_MASK

READER_DATA_MASK

Address: $0xf0003000 + 0x18 = 0xf0003018$

Pattern memory address mask

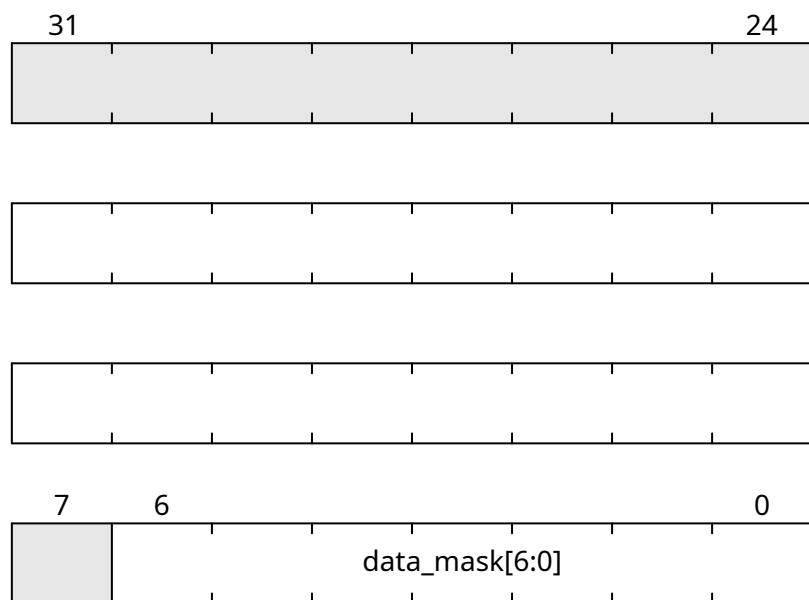


Fig. 23.78: READER_DATA_MASK

READER_DATA_DIV

Address: $0xf0003000 + 0x1c = 0xf000301c$

Pattern memory address divisor-1

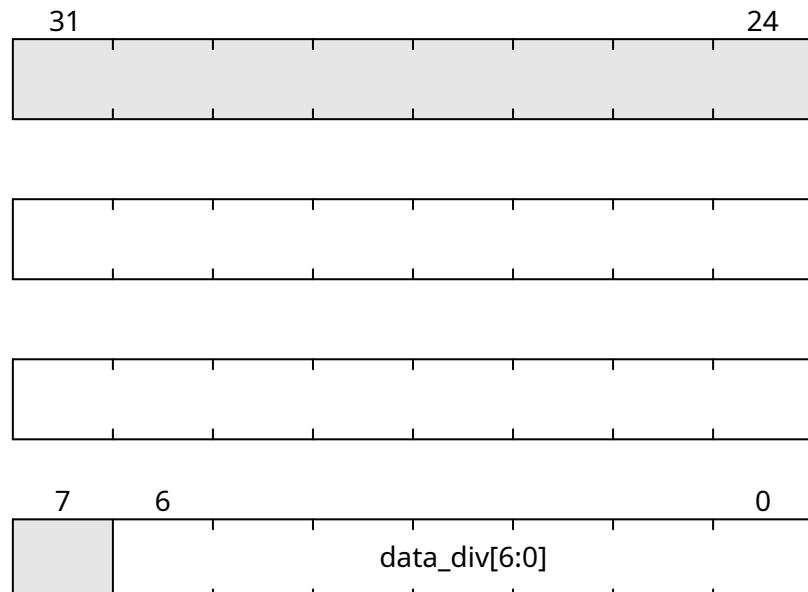


Fig. 23.79: READER_DATA_DIV

READER_INVERTER_DIVISOR_MASK

Address: $0xf0003000 + 0x20 = 0xf0003020$

Divisor mask for selecting rows for which pattern data gets inverted

READER_INVERTER_SELECTION_MASK

Address: $0xf0003000 + 0x24 = 0xf0003024$

Selection mask for selecting rows for which pattern data gets inverted

READER_ERROR_COUNT

Address: $0xf0003000 + 0x28 = 0xf0003028$

Number of errors detected

READER_SKIP_FIFO

Address: $0xf0003000 + 0x2c = 0xf000302c$

Skip waiting for user to read the errors FIFO

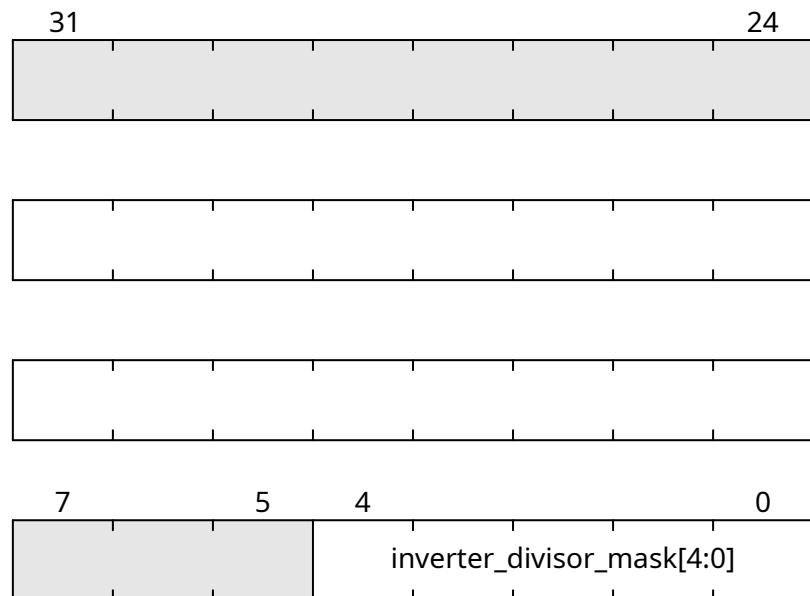


Fig. 23.80: READER_INVERTER_DIVISOR_MASK

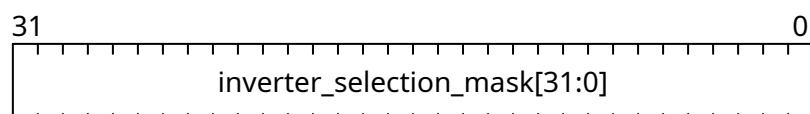


Fig. 23.81: READER_INVERTER_SELECTION_MASK

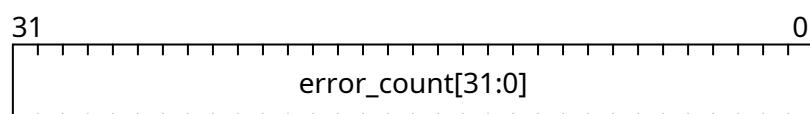


Fig. 23.82: READER_ERROR_COUNT



Fig. 23.83: READER_SKIP_FIFO

READER_ERROR_OFFSET

Address: $0xf0003000 + 0x30 = 0xf0003030$

Current offset of the error

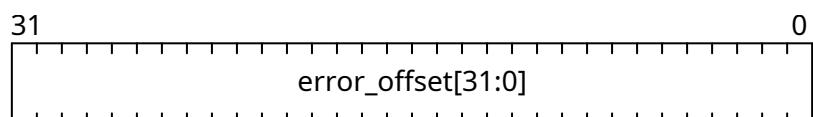


Fig. 23.84: READER_ERROR_OFFSET

READER_ERROR_DATA1

Address: $0xf0003000 + 0x34 = 0xf0003034$

Bits 32-63 of READER_ERROR_DATA. Erroneous value read from DRAM memory

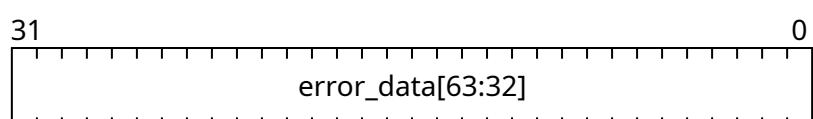


Fig. 23.85: READER_ERROR_DATA1

READER_ERROR_DATA0

Address: $0xf0003000 + 0x38 = 0xf0003038$

Bits 0-31 of *READER_ERROR_DATA*.

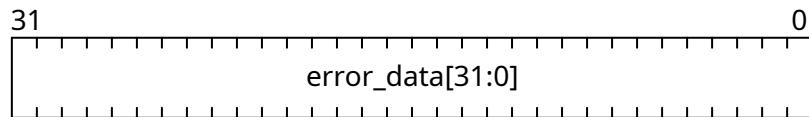


Fig. 23.86: READER_ERROR_DATA0

READER_ERROR_EXPECTED1

Address: $0xf0003000 + 0x3c = 0xf000303c$

Bits 32-63 of *READER_ERROR_EXPECTED*. Value expected to be read from DRAM memory

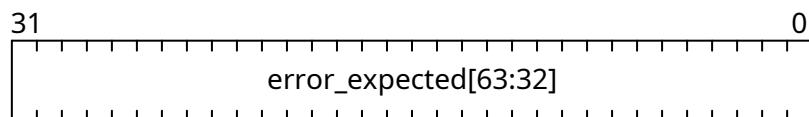


Fig. 23.87: READER_ERROR_EXPECTED1

READER_ERROR_EXPECTED0

Address: $0xf0003000 + 0x40 = 0xf0003040$

Bits 0-31 of *READER_ERROR_EXPECTED*.

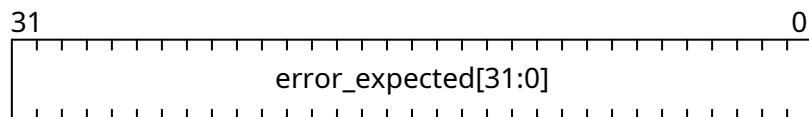


Fig. 23.88: READER_ERROR_EXPECTED0

READER_ERROR_READY

Address: $0xf0003000 + 0x44 = 0xf0003044$

Error detected and ready to read

READER_ERROR_CONTINUE

Address: $0xf0003000 + 0x48 = 0xf0003048$

Continue reading until the next error

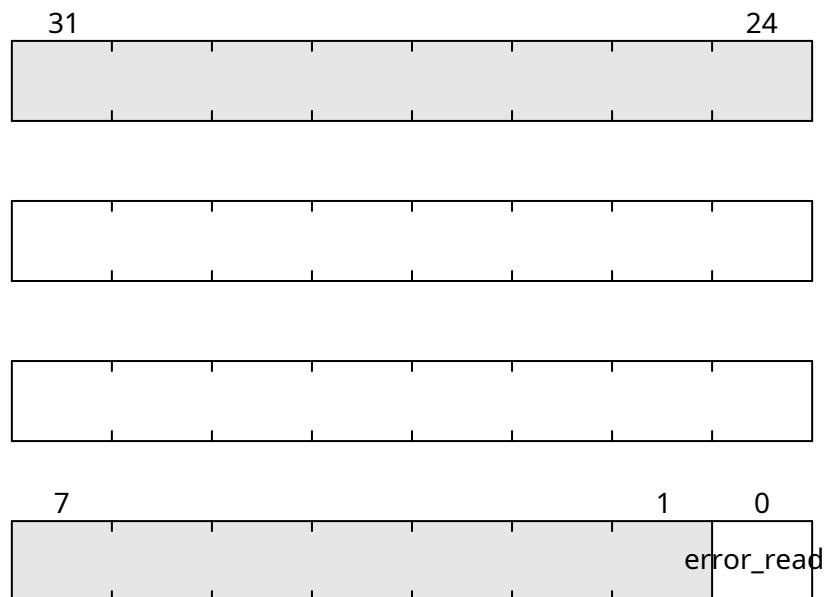


Fig. 23.89: READER_ERROR_READY

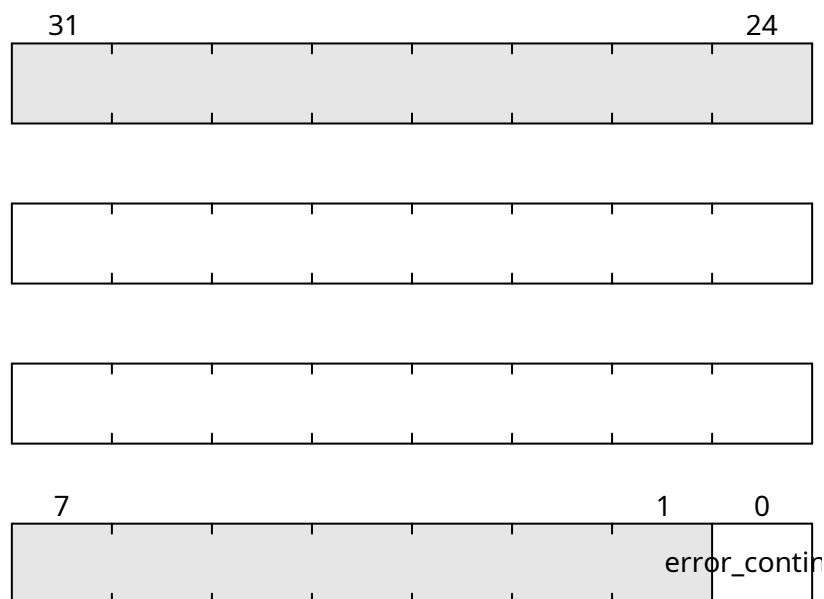


Fig. 23.90: READER_ERROR_CONTINUE

23.2.8 DFI_SWITCH

Register Listing for DFI_SWITCH

Register	Address
<i>DFI_SWITCH_REFRESH_COUNT1</i>	0xf0003800
<i>DFI_SWITCH_REFRESH_COUNT0</i>	0xf0003804
<i>DFI_SWITCH_AT_REFRESH1</i>	0xf0003808
<i>DFI_SWITCH_AT_REFRESH0</i>	0xf000380c
<i>DFI_SWITCH_REFRESH_UPDATE</i>	0xf0003810

DFI_SWITCH_REFRESH_COUNT1

Address: $0xf0003800 + 0x0 = 0xf0003800$

Bits 32-63 of *DFI_SWITCH_REFRESH_COUNT*. Count of all refresh commands issued (both by Memory Controller and the Payload Executor). Value is latched from internal counter on mode transition: MC -> PE or by writing to the *refresh_update* CSR.

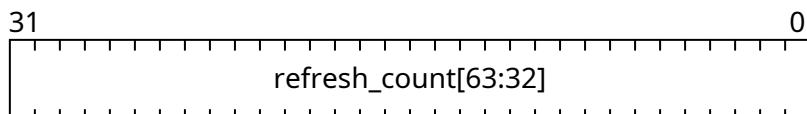


Fig. 23.91: DFI_SWITCH_REFRESH_COUNT1

DFI_SWITCH_REFRESH_COUNT0

Address: $0xf0003800 + 0x4 = 0xf0003804$

Bits 0-31 of *DFI_SWITCH_REFRESH_COUNT*.

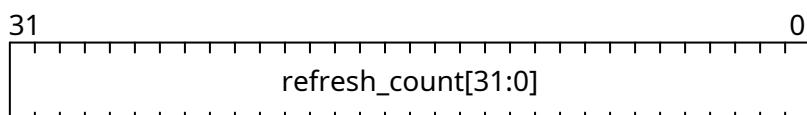


Fig. 23.92: DFI_SWITCH_REFRESH_COUNT0

DFI_SWITCH_AT_REFRESH1

Address: $0xf0003800 + 0x8 = 0xf0003808$

Bits 32-63 of *DFI_SWITCH_AT_REFRESH*. If set to a value different than 0 the mode transition MC -> PE will be performed only when the value of this register matches the current refresh commands count.

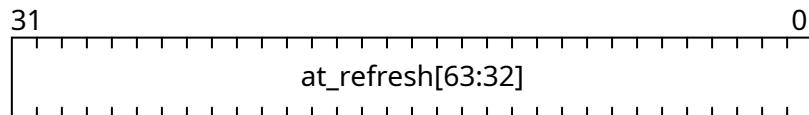


Fig. 23.93: DFI_SWITCH_AT_REFRESH1

DFI_SWITCH_AT_REFRESH0

Address: $0xf0003800 + 0xc = 0xf000380c$

Bits 0-31 of *DFI_SWITCH_AT_REFRESH*.

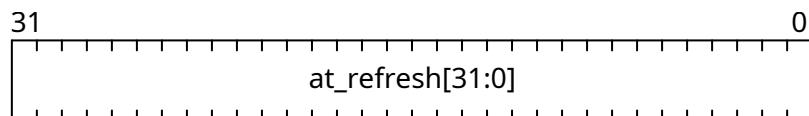


Fig. 23.94: DFI_SWITCH_AT_REFRESH0

DFI_SWITCH_REFRESH_UPDATE

Address: $0xf0003800 + 0x10 = 0xf0003810$

Force an update of the *refresh_count* CSR.



Fig. 23.95: DFI_SWITCH_REFRESH_UPDATE

23.2.9 PAYLOAD_EXECUTOR

Executes the DRAM payload from memory

Instruction decoder

All instructions are 32-bit. The format of most instructions is the same, except for the LOOP instruction, which has a constant TIMESLICE of 1.

NOOP with a TIMESLICE of 0 is a special case which is interpreted as STOP instruction. When this instruction is encountered execution gets finished immediately.

NOTE: TIMESLICE is the number of cycles the instruction will take. This means that instructions other than NOOP that use TIMESLICE=0 are illegal (although will silently be executed as having TIMESLICE=1).

NOTE2: LOOP instruction will *jump* COUNT times, meaning that the “code” inside the loop will effectively be executed COUNT+1 times.

Op codes:

Op	Value
NOOP	0b000
LOOP	0b111
ACT	0b100
PRE	0b101
REF	0b110
ZQC	0b001
READ	0b010

Instruction format:

LSB	MSB
dfi:	OP_CODE TIMESLICE ADDRESS
noop:	OP_CODE TIMESLICE_NOOP
loop:	OP_CODE COUNT JUMP
stop:	<NOOP> 0

Where ADDRESS depends on the DFI command and is one of:

LSB	MSB
RANK BANK COLUMN	
RANK BANK ROW	

Register Listing for PAYLOAD_EXECUTOR

Register	Address
PAYLOAD_EXECUTOR_START	0xf0004000
PAYLOAD_EXECUTOR_STATUS	0xf0004004
PAYLOAD_EXECUTOR_READ_COUNT	0xf0004008
PAYLOAD_EXECUTOR_EXEC_START1	0xf000400c
PAYLOAD_EXECUTOR_EXEC_START0	0xf0004010
PAYLOAD_EXECUTOR_EXEC_STOP1	0xf0004014
PAYLOAD_EXECUTOR_EXEC_STOP0	0xf0004018

PAYLOAD_EXECUTOR_START

Address: $0xf0004000 + 0x0 = 0xf0004000$

Writing to this register initializes payload execution



Fig. 23.96: PAYLOAD_EXECUTOR_START

PAYLOAD_EXECUTOR_STATUS

Address: $0xf0004000 + 0x4 = 0xf0004004$

Payload executor status register

Field	Name	Description
[0]	READY	Indicates that the executor is not running
[1]	OVERFLOW	Indicates the scratchpad memory address counter has overflowed due to the number of READ commands sent during execution

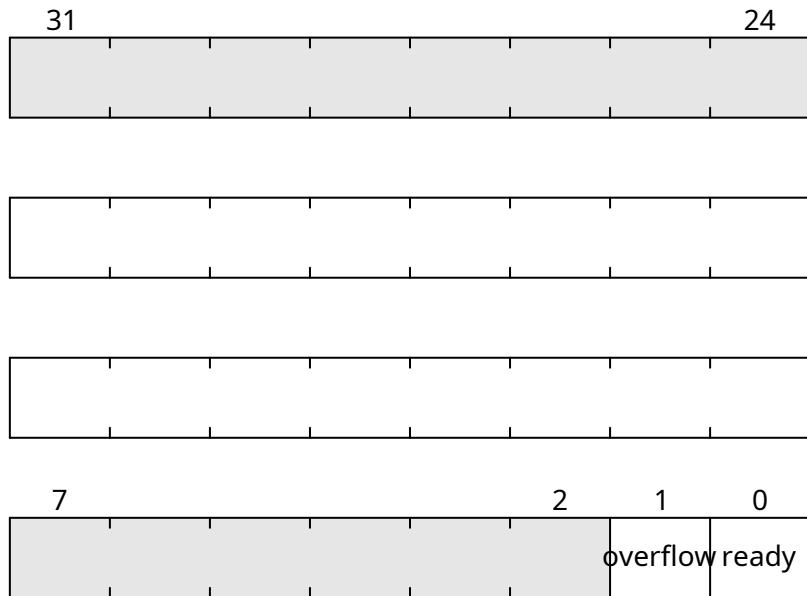


Fig. 23.97: PAYLOAD_EXECUTOR_STATUS

PAYLOAD_EXECUTOR_READ_COUNT

Address: $0xf0004000 + 0x8 = 0xf0004008$

Number of data from READ commands that is stored in the scratchpad memory

PAYLOAD_EXECUTOR_EXEC_START1

Address: $0xf0004000 + 0xc = 0xf000400c$

Bits 32-63 of PAYLOAD_EXECUTOR_EXEC_START. Number of cycles elapsed until the start of the payload execution.

PAYLOAD_EXECUTOR_EXEC_START0

Address: $0xf0004000 + 0x10 = 0xf0004010$

Bits 0-31 of PAYLOAD_EXECUTOR_EXEC_START.

PAYLOAD_EXECUTOR_EXEC_STOP1

Address: $0xf0004000 + 0x14 = 0xf0004014$

Bits 32-63 of PAYLOAD_EXECUTOR_EXEC_STOP. Number of cycles elapsed until the end of the payload execution.

PAYLOAD_EXECUTOR_EXEC_STOP0

Address: $0xf0004000 + 0x18 = 0xf0004018$

Bits 0-31 of PAYLOAD_EXECUTOR_EXEC_STOP.

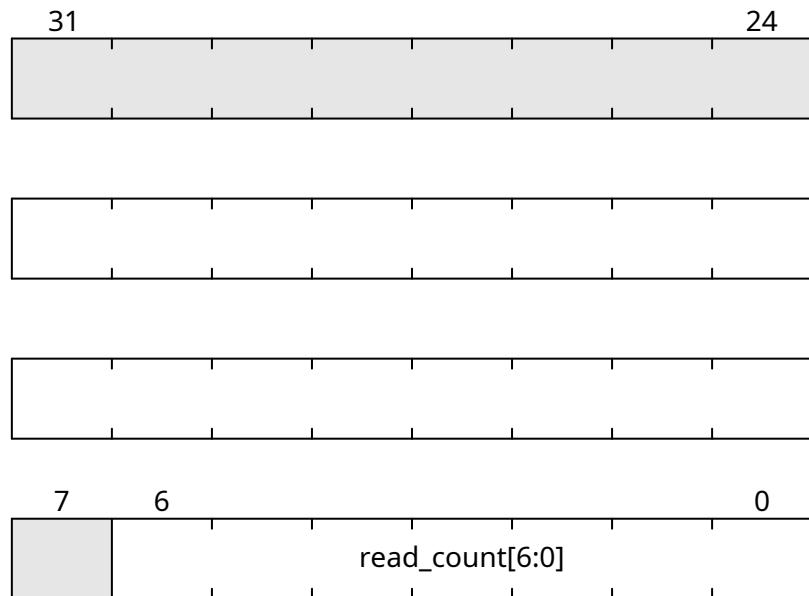


Fig. 23.98: PAYLOAD_EXECUTOR_READ_COUNT

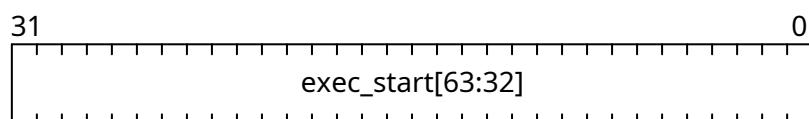


Fig. 23.99: PAYLOAD_EXECUTOR_EXEC_START1

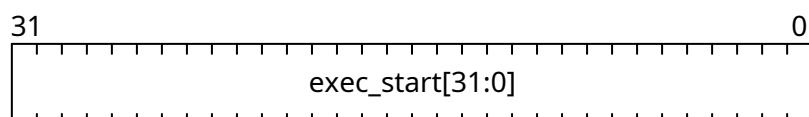


Fig. 23.100: PAYLOAD_EXECUTOR_EXEC_START0

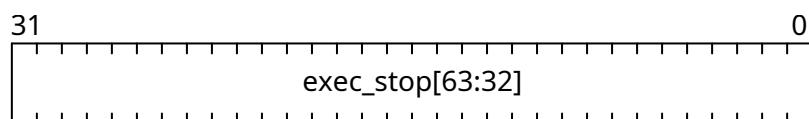


Fig. 23.101: PAYLOAD_EXECUTOR_EXEC_STOP1

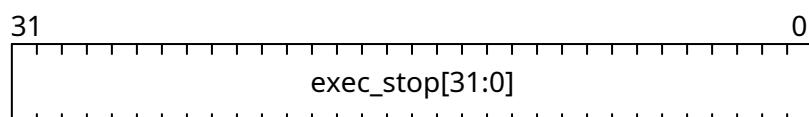


Fig. 23.102: PAYLOAD_EXECUTOR_EXEC_STOP0

23.2.10 CTRL

Register Listing for CTRL

Register	Address
<i>CTRL_RESET</i>	0xf0004800
<i>CTRL_SCRATCH</i>	0xf0004804
<i>CTRL_BUS_ERRORS</i>	0xf0004808

CTRL__RESET

Address: $0xf0004800 + 0x0 = 0xf0004800$

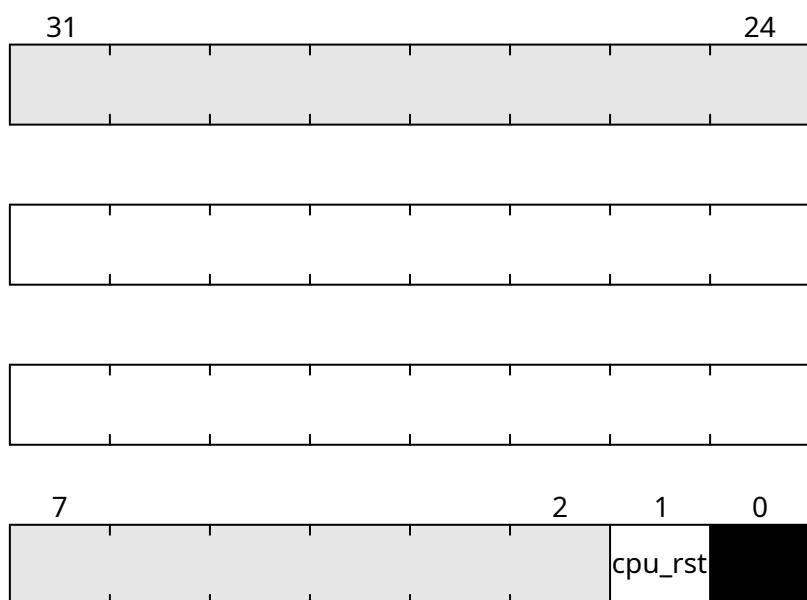


Fig. 23.103: CTRL__RESET

Field	Name	Description
[0]	SOC_RST	Write 1 to this register to reset the full SoC (Pulse Reset)
[1]	CPU_RST	Write 1 to this register to reset the CPU(s) of the SoC (Hold Reset)

CTRL_SCRATCH

Address: $0xf0004800 + 0x4 = 0xf0004804$

Use this register as a scratch space to verify that software read/write accesses to the Wishbone/CSR bus are working correctly. The initial reset value of 0x1234578 can be used to verify endianness.

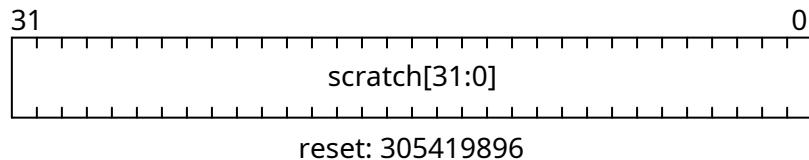


Fig. 23.104: CTRL_SCRATCH

CTRL_BUS_ERRORS

Address: $0xf0004800 + 0x8 = 0xf0004808$

Total number of Wishbone bus errors (timeouts) since start.

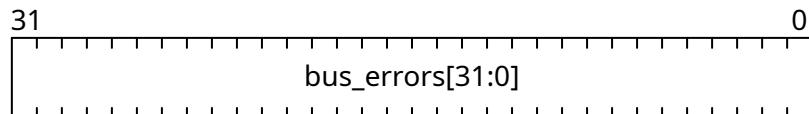


Fig. 23.105: CTRL_BUS_ERRORS

23.2.11 ETPHY

Register Listing for ETPHY

Register	Address
<i>ETPHY_CRG_RESET</i>	<i>0xf0005000</i>
<i>ETPHY_MDIO_W</i>	<i>0xf0005004</i>
<i>ETPHY_MDIO_R</i>	<i>0xf0005008</i>

ETPHY_CRG_RESET

Address: $0xf0005000 + 0x0 = 0xf0005000$

ETPHY_MDIO_W

Address: $0xf0005000 + 0x4 = 0xf0005004$

Field	Name	Description



Fig. 23.106: ETHPHY_CRG_RESET

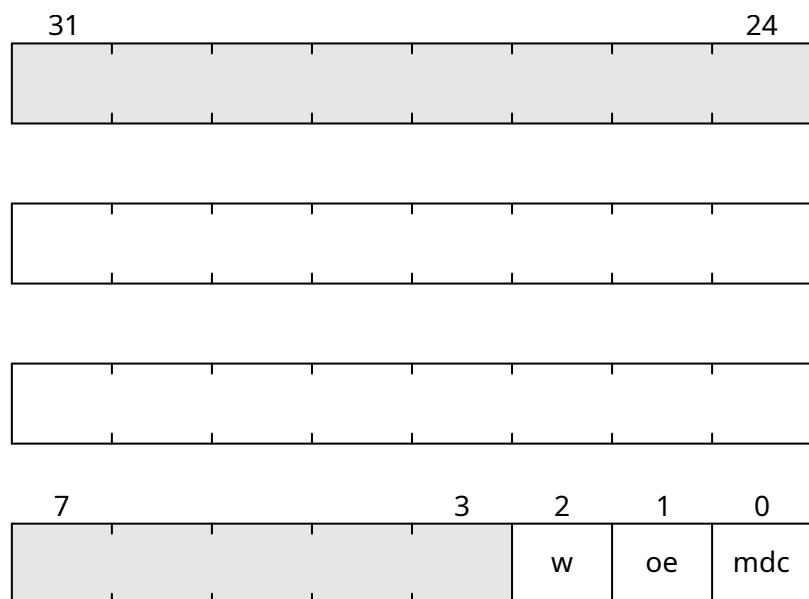


Fig. 23.107: ETHPHY_MDIO_W

ETHPHY_MDIO_R

Address: $0xf0005000 + 0x8 = 0xf0005008$



Fig. 23.108: ETHPHY_MDIO_R

Field	Name	Description

23.2.12 IDENTIFIER_MEM

Register Listing for IDENTIFIER_MEM

Register	Address
<i>IDENTIFIER_MEM</i>	<i>0xf0005800</i>

IDENTIFIER_MEM

Address: $0xf0005800 + 0x0 = 0xf0005800$

8 x 108-bit memory

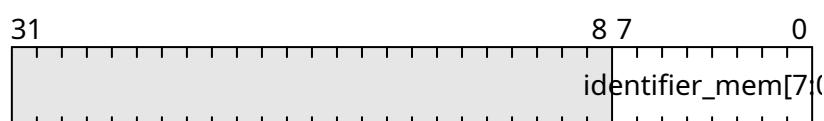


Fig. 23.109: IDENTIFIER_MEM

23.2.13 MAIN

Register Listing for MAIN

Register	Address
<i>MAIN_DQ_REMAPPING1</i>	0xf0006000
<i>MAIN_DQ_REMAPPING0</i>	0xf0006004

MAIN_DQ_REMAPPING1

Address: $0xf0006000 + 0x0 = 0xf0006000$

Bits 32-63 of *MAIN_DQ_REMAPPING*.

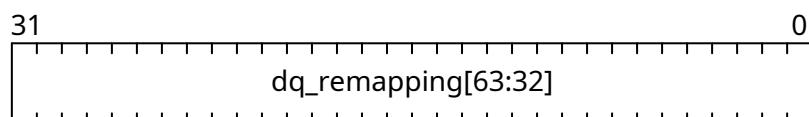


Fig. 23.110: MAIN_DQ_REMAPPING1

MAIN_DQ_REMAPPING0

Address: $0xf0006000 + 0x4 = 0xf0006004$

Bits 0-31 of *MAIN_DQ_REMAPPING*.

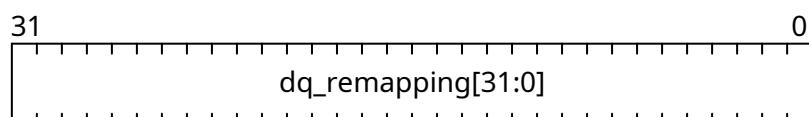


Fig. 23.111: MAIN_DQ_REMAPPING0

23.2.14 SDRAM

Register Listing for SDRAM

Register	Address
<i>SDRAM_DFII_CONTROL</i>	0xf0006800
<i>SDRAM_DFII_FORCE_ISSUE</i>	0xf0006804
<i>SDRAM_DFII_CMDINJECTOR_COMMAND_STORAGE</i>	0xf0006808
<i>SDRAM_DFII_CMDINJECTOR_COMMAND_STORAGE_WR_MASK</i>	0xf000680c
<i>SDRAM_DFII_CMDINJECTOR_PHASE_ADDR</i>	0xf0006810
<i>SDRAM_DFII_CMDINJECTOR_STORE_CONTINUOUS_CMD</i>	0xf0006814
<i>SDRAM_DFII_CMDINJECTOR_STORE_SINGLESOT_CMD</i>	0xf0006818
<i>SDRAM_DFII_CMDINJECTOR_SINGLE_SHOT</i>	0xf000681c
<i>SDRAM_DFII_CMDINJECTOR_ISSUE_COMMAND</i>	0xf0006820

continues on next page

Table 23.2 – continued from previous page

Register	Address
<i>SDRAM_DFII_CMDINJECTOR_WRDATA_SELECT</i>	0xf0006824
<i>SDRAM_DFII_CMDINJECTOR_WRDATA</i>	0xf0006828
<i>SDRAM_DFII_CMDINJECTOR_WRDATA_S</i>	0xf000682c
<i>SDRAM_DFII_CMDINJECTOR_WRDATA_STORE</i>	0xf0006830
<i>SDRAM_DFII_CMDINJECTOR_SETUP</i>	0xf0006834
<i>SDRAM_DFII_CMDINJECTOR_SAMPLE</i>	0xf0006838
<i>SDRAM_DFII_CMDINJECTOR_RESULT_ARRAY</i>	0xf000683c
<i>SDRAM_DFII_CMDINJECTOR_RESET</i>	0xf0006840
<i>SDRAM_DFII_CMDINJECTOR_RDDATA_SELECT</i>	0xf0006844
<i>SDRAM_DFII_CMDINJECTOR_RDDATA_CAPTURE_CNT</i>	0xf0006848
<i>SDRAM_DFII_CMDINJECTOR_RDDATA</i>	0xf000684c
<i>SDRAM_CONTROLLER_TRP</i>	0xf0006850
<i>SDRAM_CONTROLLER_TRCD</i>	0xf0006854
<i>SDRAM_CONTROLLER_TWR</i>	0xf0006858
<i>SDRAM_CONTROLLER_TWTR</i>	0xf000685c
<i>SDRAM_CONTROLLER_TREFI</i>	0xf0006860
<i>SDRAM_CONTROLLER_TRFC</i>	0xf0006864
<i>SDRAM_CONTROLLER_TFAW</i>	0xf0006868
<i>SDRAM_CONTROLLER_TCCD</i>	0xf000686c
<i>SDRAM_CONTROLLER_TCCD_WR</i>	0xf0006870
<i>SDRAM_CONTROLLER_TRTP</i>	0xf0006874
<i>SDRAM_CONTROLLER_TRRD</i>	0xf0006878
<i>SDRAM_CONTROLLER_TRC</i>	0xf000687c
<i>SDRAM_CONTROLLER_TRAS</i>	0xf0006880
<i>SDRAM_CONTROLLER_LAST_ADDR_0</i>	0xf0006884
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0</i>	0xf0006888
<i>SDRAM_CONTROLLER_LAST_ADDR_1</i>	0xf000688c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1</i>	0xf0006890
<i>SDRAM_CONTROLLER_LAST_ADDR_2</i>	0xf0006894
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2</i>	0xf0006898
<i>SDRAM_CONTROLLER_LAST_ADDR_3</i>	0xf000689c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3</i>	0xf00068a0
<i>SDRAM_CONTROLLER_LAST_ADDR_4</i>	0xf00068a4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4</i>	0xf00068a8
<i>SDRAM_CONTROLLER_LAST_ADDR_5</i>	0xf00068ac
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5</i>	0xf00068b0
<i>SDRAM_CONTROLLER_LAST_ADDR_6</i>	0xf00068b4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6</i>	0xf00068b8
<i>SDRAM_CONTROLLER_LAST_ADDR_7</i>	0xf00068bc
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7</i>	0xf00068c0
<i>SDRAM_CONTROLLER_LAST_ADDR_8</i>	0xf00068c4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_8</i>	0xf00068c8
<i>SDRAM_CONTROLLER_LAST_ADDR_9</i>	0xf00068cc
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_9</i>	0xf00068d0
<i>SDRAM_CONTROLLER_LAST_ADDR_10</i>	0xf00068d4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_10</i>	0xf00068d8
<i>SDRAM_CONTROLLER_LAST_ADDR_11</i>	0xf00068dc
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_11</i>	0xf00068e0

continues on next page

Table 23.2 – continued from previous page

Register	Address
<i>SDRAM_CONTROLLER_LAST_ADDR_12</i>	0xf00068e4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_12</i>	0xf00068e8
<i>SDRAM_CONTROLLER_LAST_ADDR_13</i>	0xf00068ec
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_13</i>	0xf00068f0
<i>SDRAM_CONTROLLER_LAST_ADDR_14</i>	0xf00068f4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_14</i>	0xf00068f8
<i>SDRAM_CONTROLLER_LAST_ADDR_15</i>	0xf00068fc
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_15</i>	0xf0006900
<i>SDRAM_CONTROLLER_LAST_ADDR_16</i>	0xf0006904
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_16</i>	0xf0006908
<i>SDRAM_CONTROLLER_LAST_ADDR_17</i>	0xf000690c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_17</i>	0xf0006910
<i>SDRAM_CONTROLLER_LAST_ADDR_18</i>	0xf0006914
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_18</i>	0xf0006918
<i>SDRAM_CONTROLLER_LAST_ADDR_19</i>	0xf000691c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_19</i>	0xf0006920
<i>SDRAM_CONTROLLER_LAST_ADDR_20</i>	0xf0006924
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_20</i>	0xf0006928
<i>SDRAM_CONTROLLER_LAST_ADDR_21</i>	0xf000692c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_21</i>	0xf0006930
<i>SDRAM_CONTROLLER_LAST_ADDR_22</i>	0xf0006934
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_22</i>	0xf0006938
<i>SDRAM_CONTROLLER_LAST_ADDR_23</i>	0xf000693c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_23</i>	0xf0006940
<i>SDRAM_CONTROLLER_LAST_ADDR_24</i>	0xf0006944
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_24</i>	0xf0006948
<i>SDRAM_CONTROLLER_LAST_ADDR_25</i>	0xf000694c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_25</i>	0xf0006950
<i>SDRAM_CONTROLLER_LAST_ADDR_26</i>	0xf0006954
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_26</i>	0xf0006958
<i>SDRAM_CONTROLLER_LAST_ADDR_27</i>	0xf000695c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_27</i>	0xf0006960
<i>SDRAM_CONTROLLER_LAST_ADDR_28</i>	0xf0006964
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_28</i>	0xf0006968
<i>SDRAM_CONTROLLER_LAST_ADDR_29</i>	0xf000696c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_29</i>	0xf0006970
<i>SDRAM_CONTROLLER_LAST_ADDR_30</i>	0xf0006974
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_30</i>	0xf0006978
<i>SDRAM_CONTROLLER_LAST_ADDR_31</i>	0xf000697c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_31</i>	0xf0006980

SDRAM_DFII_CONTROL

Address: 0xf0006800 + 0x0 = 0xf0006800

Control DFI signals common to all phases

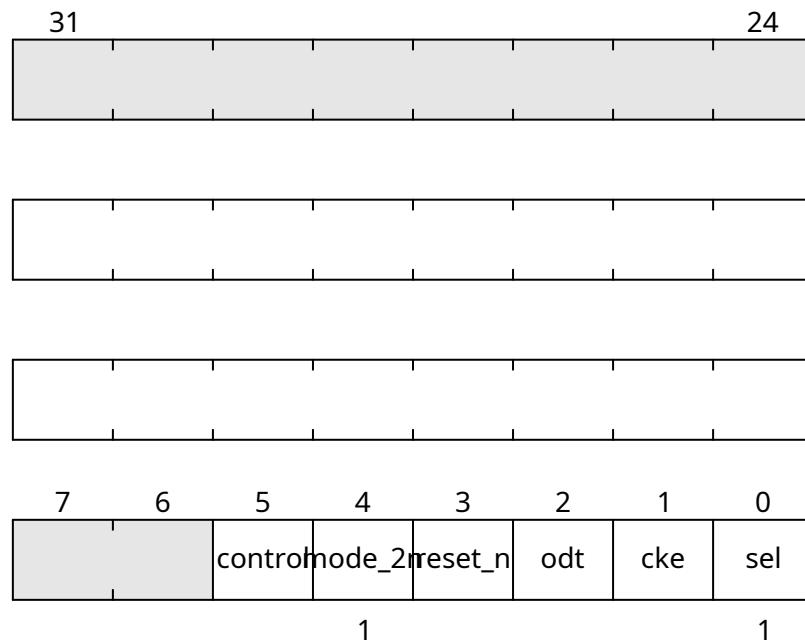


Fig. 23.112: SDRAM_DFII_CONTROL

Field	Name	Description						
[0]	SEL	<table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0b0</td><td>Software (CPU) control.</td></tr> <tr> <td>0b1</td><td>Hardware control (default).</td></tr> </tbody> </table>	Value	Description	0b0	Software (CPU) control.	0b1	Hardware control (default).
Value	Description							
0b0	Software (CPU) control.							
0b1	Hardware control (default).							
[1]	CKE	DFI clock enable bus						
[2]	ODT	DFI on-die termination bus						
[3]	RESET_N	DFI clock reset bus						
[4]	MODE_2N	<table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0b0</td><td>In 1N mode</td></tr> <tr> <td>0b1</td><td>In 2N mode (Default)</td></tr> </tbody> </table>	Value	Description	0b0	In 1N mode	0b1	In 2N mode (Default)
Value	Description							
0b0	In 1N mode							
0b1	In 2N mode (Default)							
[5]	CONTROL	<table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0b1</td><td>Cmd Injector</td></tr> </tbody> </table>	Value	Description	0b1	Cmd Injector		
Value	Description							
0b1	Cmd Injector							

SDRAM_DFII_FORCE_ISSUE

Address: $0xf0006800 + 0x4 = 0xf0006804$

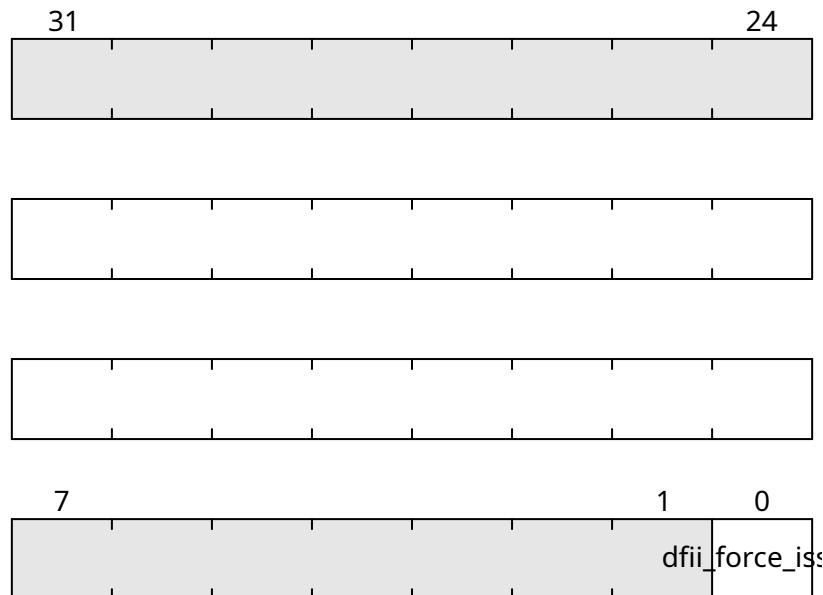


Fig. 23.113: SDRAM_DFII_FORCE_ISSUE

SDRAM_DFII_CMDINJECTOR_COMMAND_STORAGE

Address: $0xf0006800 + 0x8 = 0xf0006808$

DDR5 command and control signals

Field	Name	Description
[13:0]	CA	Command/Address bus
[14]	CS	DFI chip select bus

SDRAM_DFII_CMDINJECTOR_COMMAND_STORAGE_WR_MASK

Address: $0xf0006800 + 0xc = 0xf000680c$

DDR5 wrdata mask control signals

Field	Name	Description

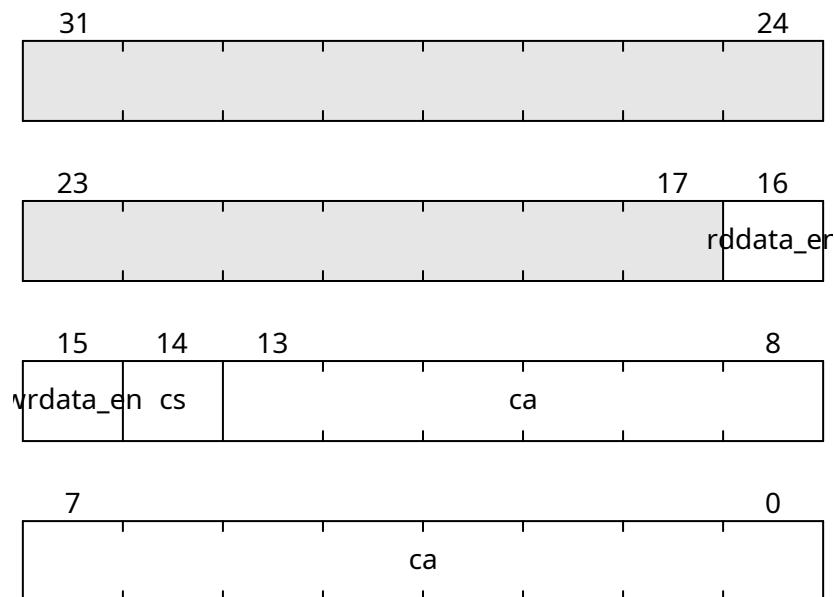


Fig. 23.114: SDRAM_DFII_CMDINJECTOR_COMMAND_STORAGE

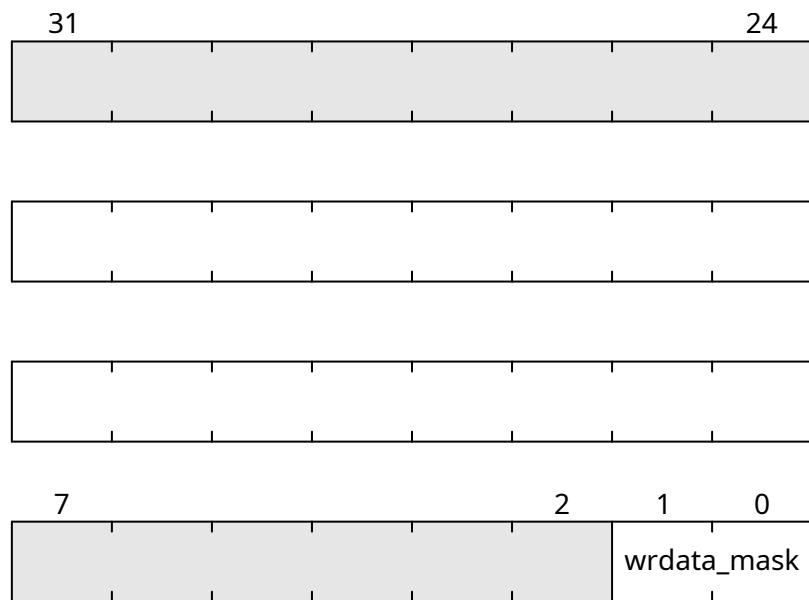


Fig. 23.115: SDRAM_DFII_CMDINJECTOR_COMMAND_STORAGE_WR_MASK

SDRAM_DFII_CMDINJECTOR_PHASE_ADDR

Address: $0xf0006800 + 0x10 = 0xf0006810$

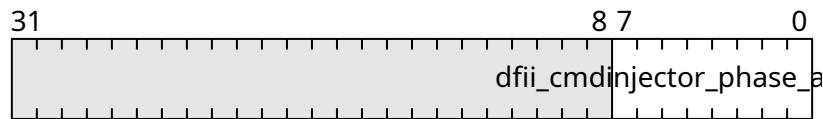


Fig. 23.116: SDRAM_DFII_CMDINJECTOR_PHASE_ADDR

SDRAM_DFII_CMDINJECTOR_STORE_CONTINUOUS_CMD

Address: $0xf0006800 + 0x14 = 0xf0006814$

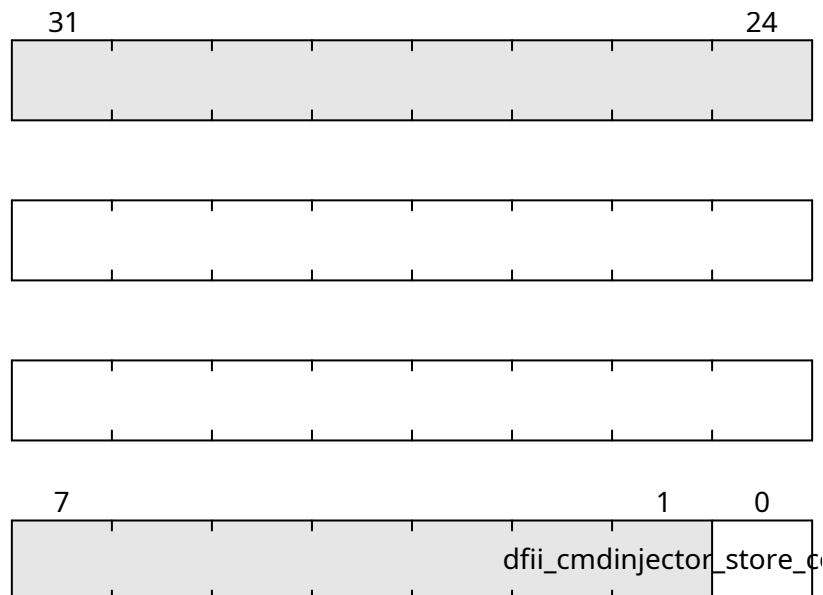


Fig. 23.117: SDRAM_DFII_CMDINJECTOR_STORE_CONTINUOUS_CMD

SDRAM_DFII_CMDINJECTOR_STORE_SINGLESHT_CMD

Address: $0xf0006800 + 0x18 = 0xf0006818$

SDRAM_DFII_CMDINJECTOR_SINGLE_SHOT

Address: $0xf0006800 + 0x1c = 0xf000681c$

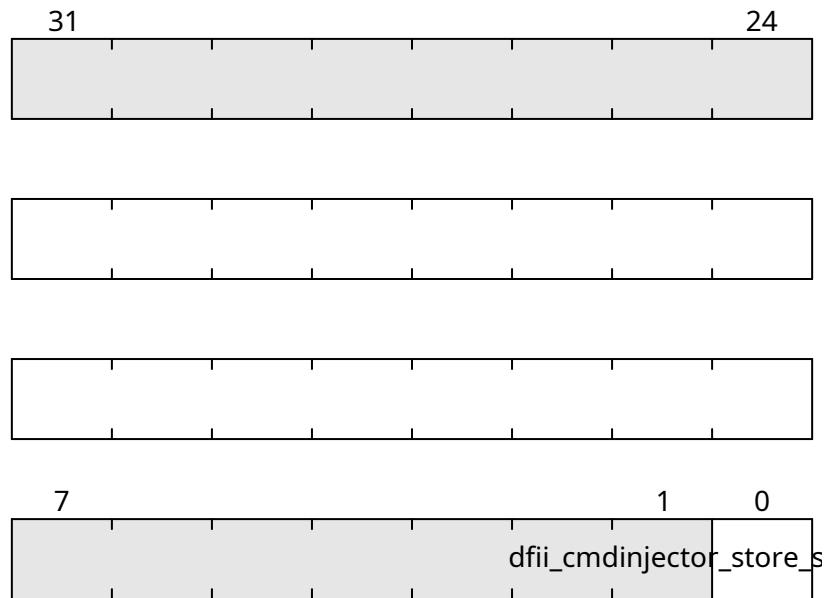


Fig. 23.118: SDRAM_DFII_CMDINJECTOR_STORE_SINGLESHTOT_CMD

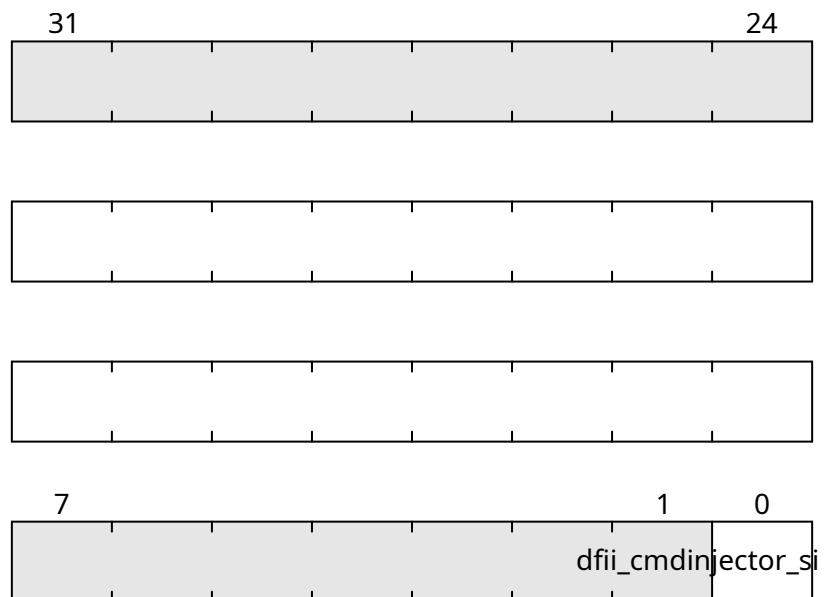


Fig. 23.119: SDRAM_DFII_CMDINJECTOR_SINGLE_SHOT

SDRAM_DFII_CMDINJECTOR_ISSUE_COMMAND

Address: $0xf0006800 + 0x20 = 0xf0006820$

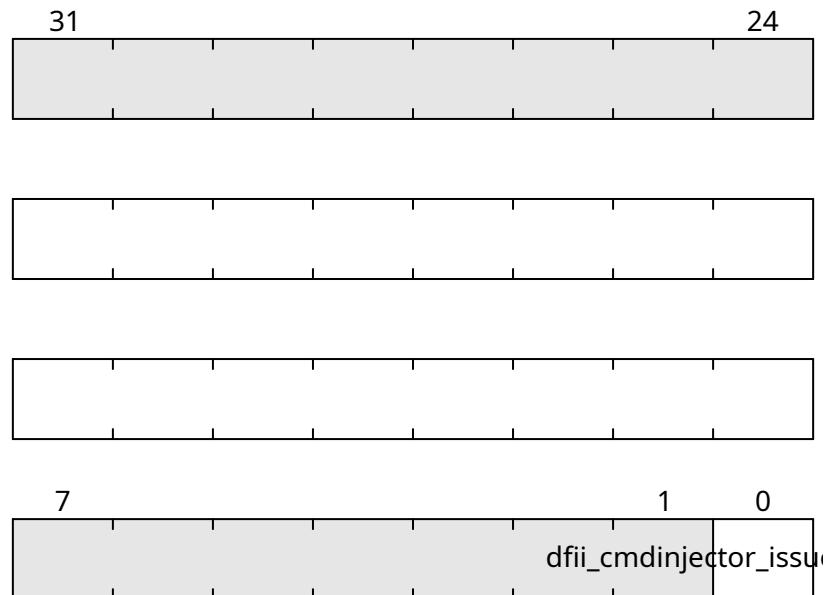


Fig. 23.120: SDRAM_DFII_CMDINJECTOR_ISSUE_COMMAND

SDRAM_DFII_CMDINJECTOR_WRDATASELECT

Address: $0xf0006800 + 0x24 = 0xf0006824$

SDRAM_DFII_CMDINJECTOR_WRDATASELECT

Address: $0xf0006800 + 0x28 = 0xf0006828$

SDRAM_DFII_CMDINJECTOR_WRDATAS

Address: $0xf0006800 + 0x2c = 0xf000682c$

SDRAM_DFII_CMDINJECTOR_WRDATASTORE

Address: $0xf0006800 + 0x30 = 0xf0006830$

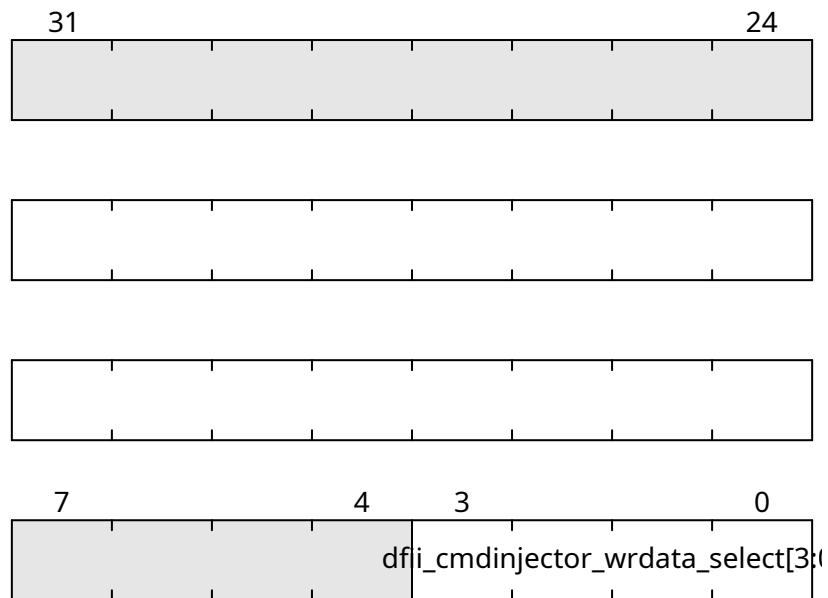


Fig. 23.121: SDRAM_DFII_CMDINJECTOR_WRDATA_SELECT

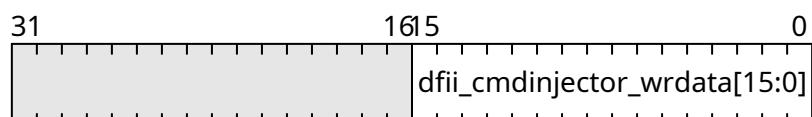


Fig. 23.122: SDRAM_DFII_CMDINJECTOR_WRDATA

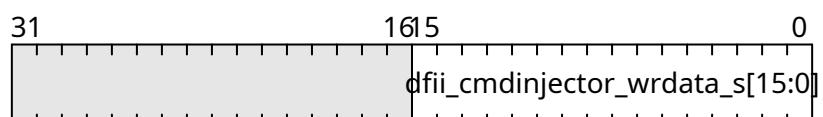


Fig. 23.123: SDRAM_DFII_CMDINJECTOR_WRDATA_S



Fig. 23.124: SDRAM_DFII_CMDINJECTOR_WRDATA_STORE

SDRAM_DFII_CMDINJECTOR_SETUP

Address: $0xf0006800 + 0x34 = 0xf0006834$

Field	Name	Description
[0]	INITIAL_STATE	Initial value of all bits
[1]	OPERATION	0 - or (default), 1 -and

SDRAM_DFII_CMDINJECTOR_SAMPLE

Address: $0xf0006800 + 0x38 = 0xf0006838$

SDRAM_DFII_CMDINJECTOR_RESULT_ARRAY

Address: $0xf0006800 + 0x3c = 0xf000683c$

SDRAM_DFII_CMDINJECTOR_RESET

Address: $0xf0006800 + 0x40 = 0xf0006840$

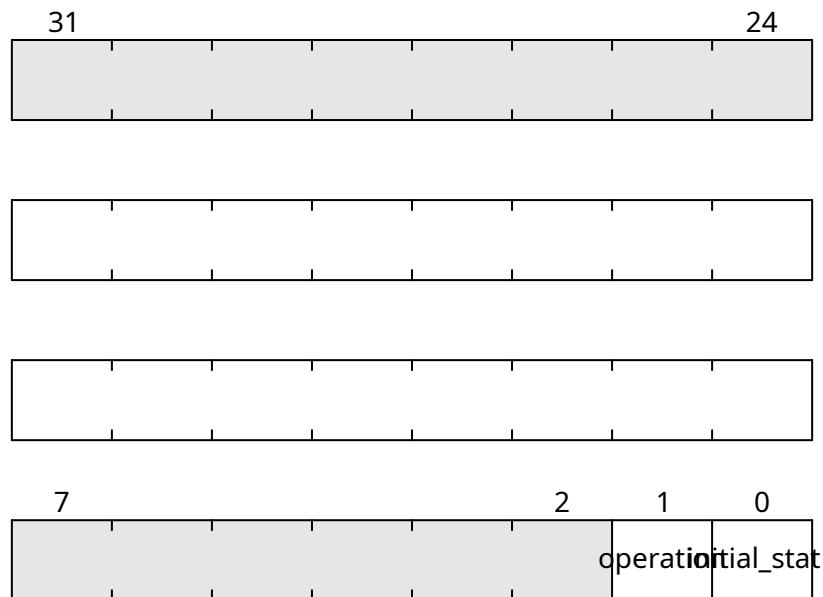


Fig. 23.125: SDRAM_DFII_CMDINJECTOR_SETUP

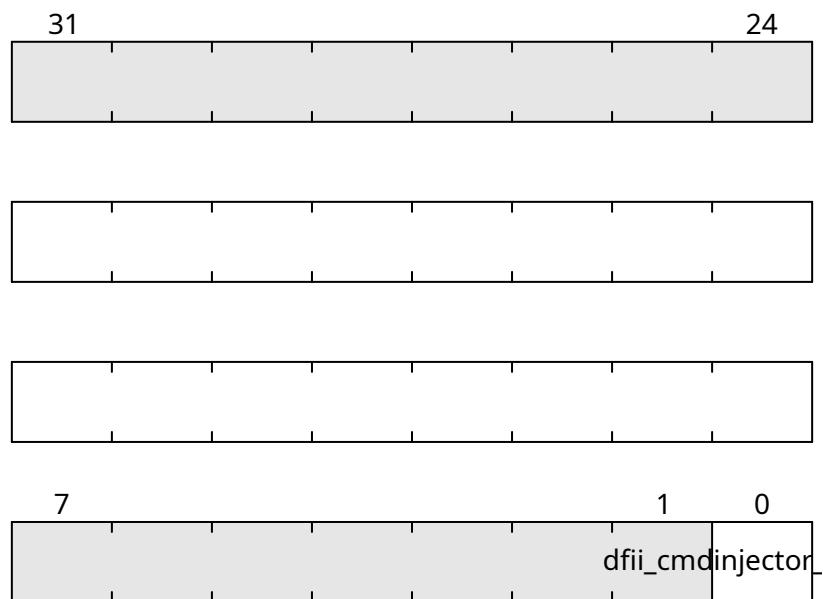


Fig. 23.126: SDRAM_DFII_CMDINJECTOR_SAMPLE

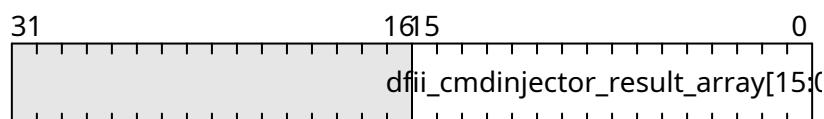


Fig. 23.127: SDRAM_DFII_CMDINJECTOR_RESULT_ARRAY

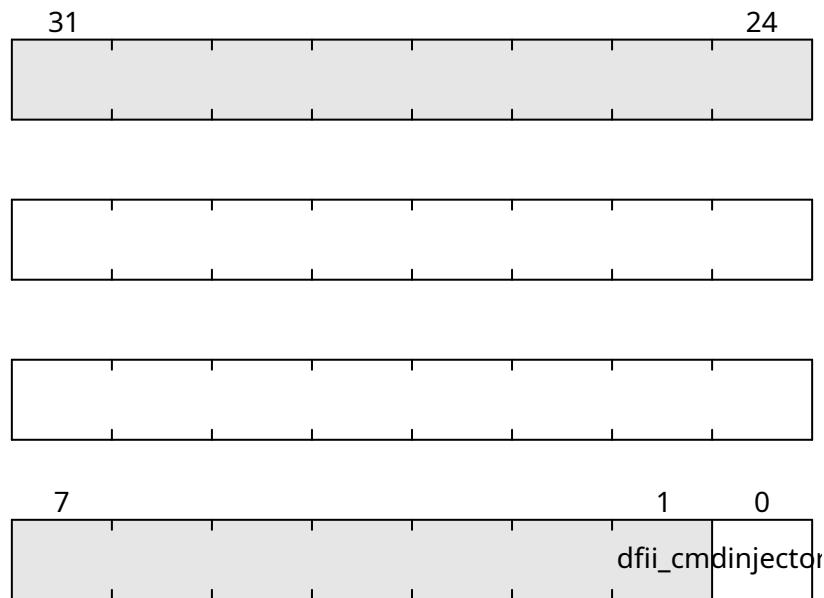


Fig. 23.128: SDRAM_DFII_CMDINJECTOR_RESET

SDRAM_DFII_CMDINJECTOR_RDDATA_SELECT

Address: 0xf0006800 + 0x44 = 0xf0006844

SDRAM_DFII_CMDINJECTOR_RDDATA_CAPTURE_CNT

Address: 0xf0006800 + 0x48 = 0xf0006848

SDRAM_DFII_CMDINJECTOR_RDDATA

Address: 0xf0006800 + 0x4c = 0xf000684c

SDRAM_CONTROLLER_TRP

Address: 0xf0006800 + 0x50 = 0xf0006850

SDRAM_CONTROLLER_TRCD

Address: 0xf0006800 + 0x54 = 0xf0006854

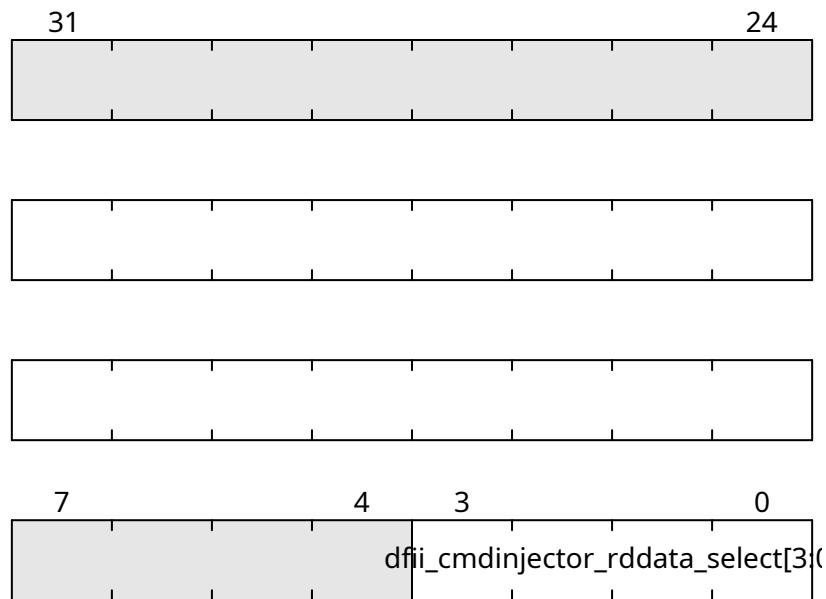


Fig. 23.129: SDRAM_DFII_CMDINJECTOR_RDDATA_SELECT

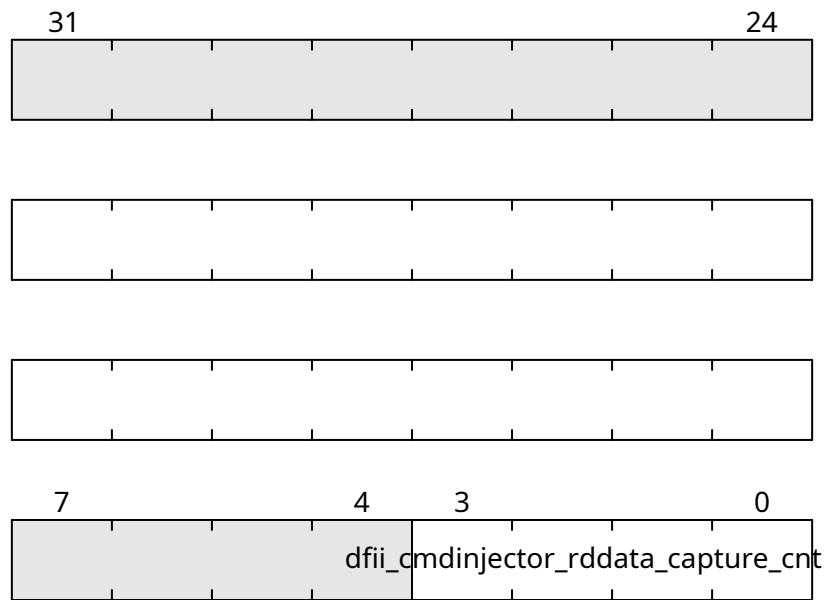


Fig. 23.130: SDRAM_DFII_CMDINJECTOR_RDDATA_CAPTURE_CNT



Fig. 23.131: SDRAM_DFII_CMDINJECTOR_RDDATA

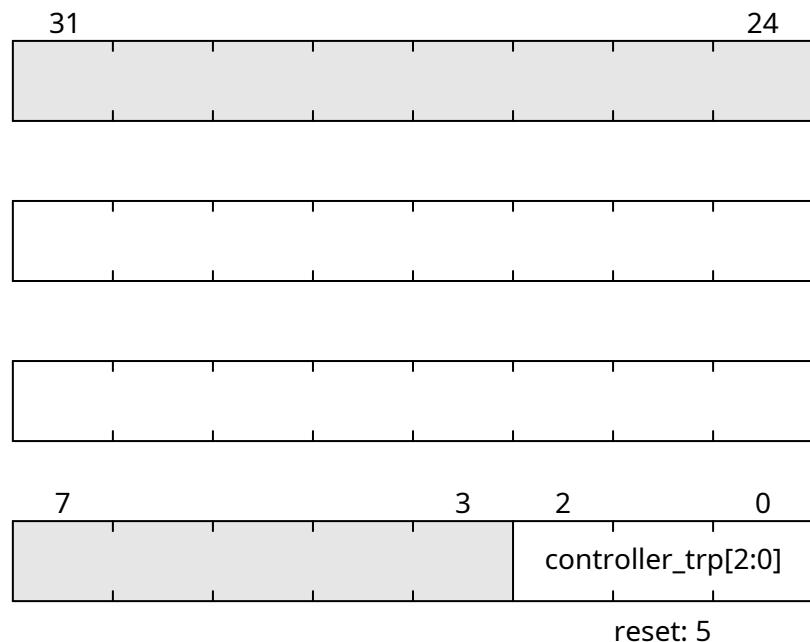


Fig. 23.132: SDRAM_CONTROLLER_TRP

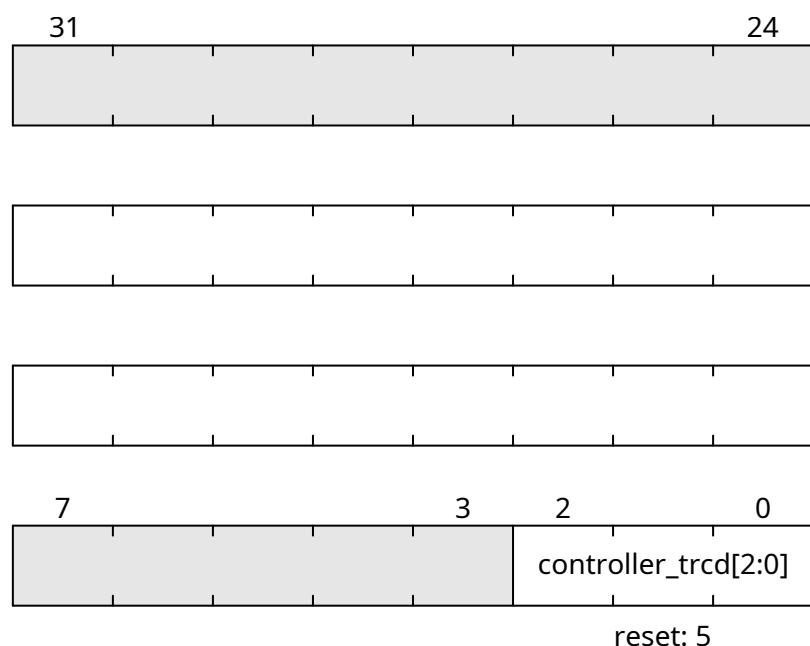


Fig. 23.133: SDRAM_CONTROLLER_TRCD

SDRAM_CONTROLLER_TWR

Address: 0xf0006800 + 0x58 = 0xf0006858

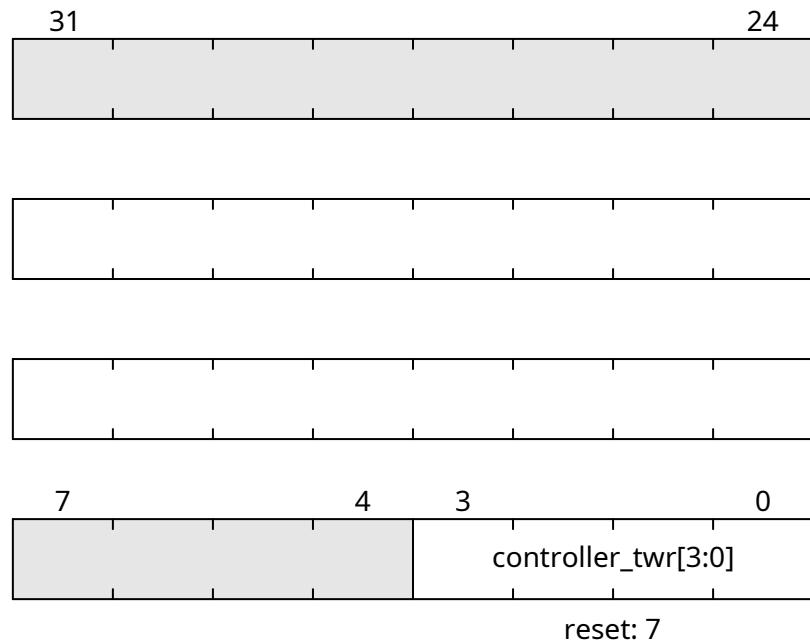


Fig. 23.134: SDRAM_CONTROLLER_TWR

SDRAM_CONTROLLER_TWTR

Address: 0xf0006800 + 0x5c = 0xf000685c

SDRAM_CONTROLLER_TREFI

Address: 0xf0006800 + 0x60 = 0xf0006860

SDRAM_CONTROLLER_TRFC

Address: 0xf0006800 + 0x64 = 0xf0006864

SDRAM_CONTROLLER_TFAW

Address: 0xf0006800 + 0x68 = 0xf0006868

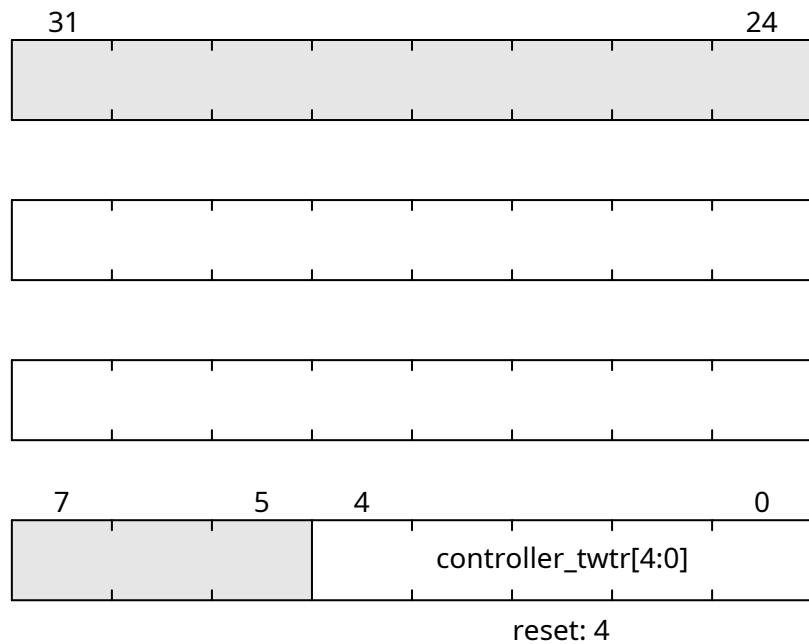


Fig. 23.135: SDRAM_CONTROLLER_TWTR

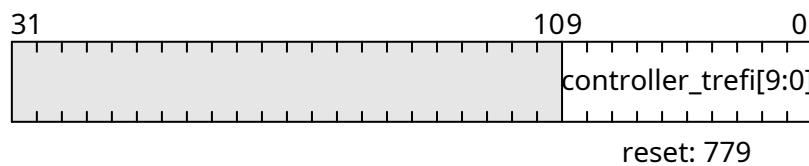


Fig. 23.136: SDRAM_CONTROLLER_TREFI

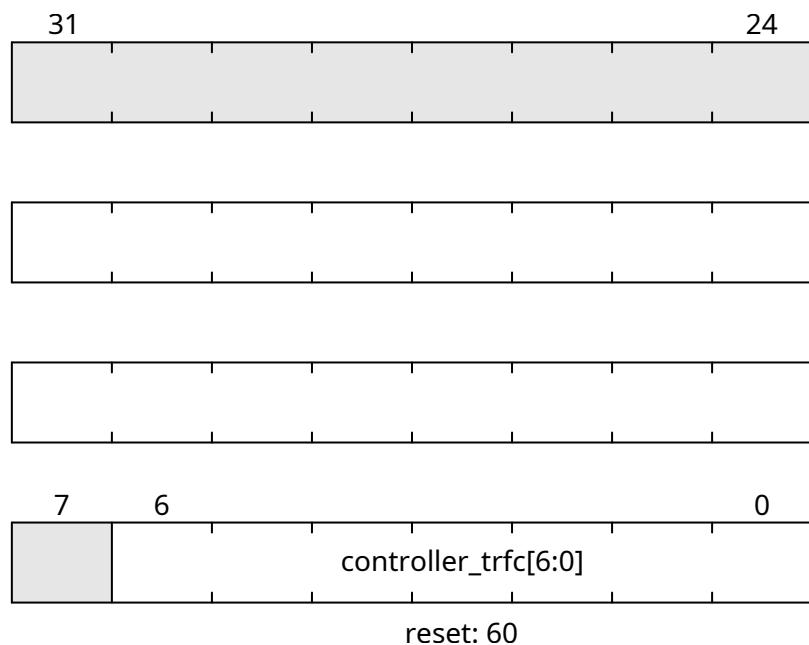


Fig. 23.137: SDRAM_CONTROLLER_TRFC

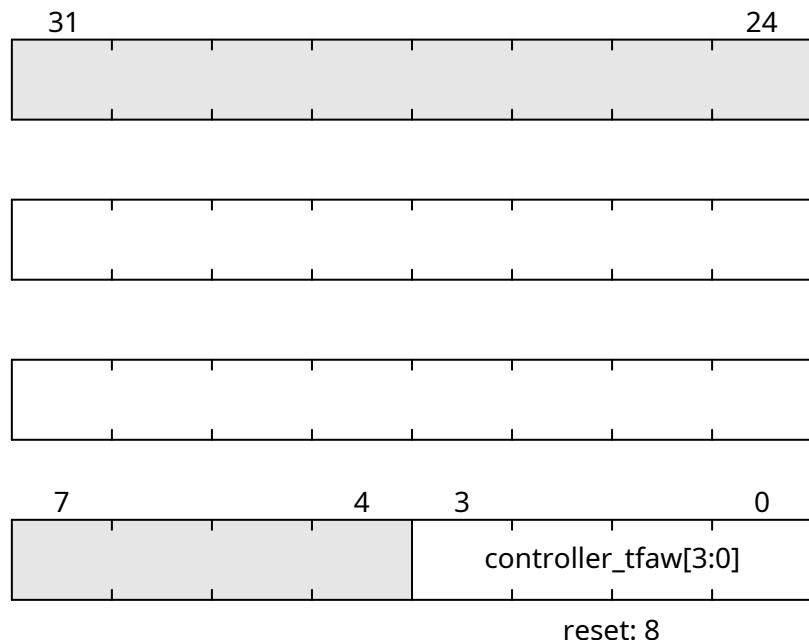


Fig. 23.138: SDRAM_CONTROLLER_TFAW

SDRAM_CONTROLLER_TCCD

Address: 0xf0006800 + 0x6c = 0xf000686c

SDRAM_CONTROLLER_TCCD_WR

Address: 0xf0006800 + 0x70 = 0xf0006870

SDRAM_CONTROLLER_TRTP

Address: 0xf0006800 + 0x74 = 0xf0006874

SDRAM_CONTROLLER_TRRD

Address: 0xf0006800 + 0x78 = 0xf0006878

SDRAM_CONTROLLER_TRC

Address: 0xf0006800 + 0x7c = 0xf000687c

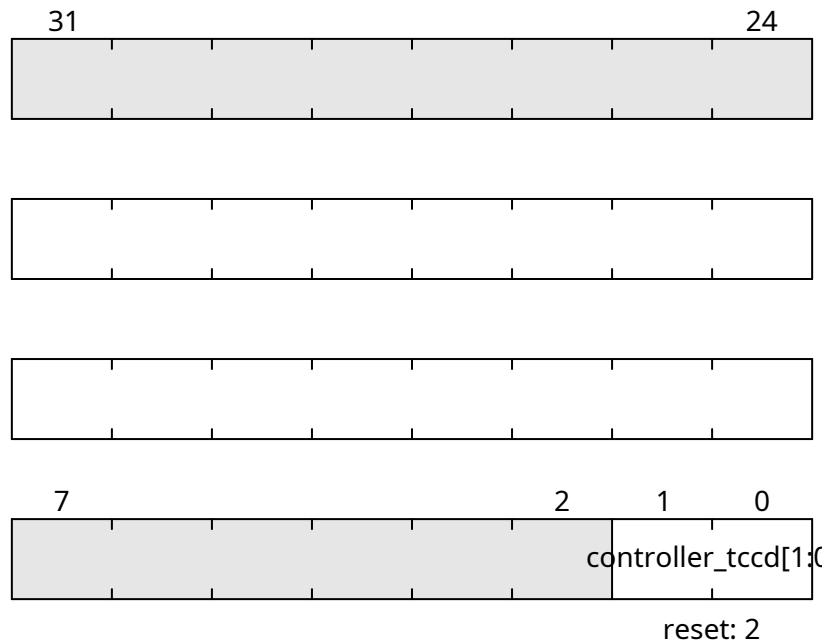


Fig. 23.139: SDRAM_CONTROLLER_TCCD

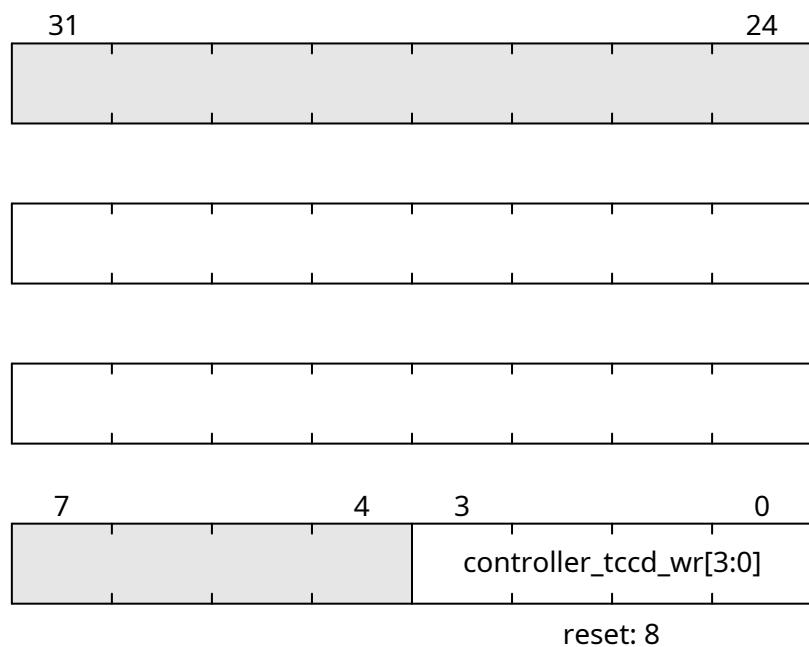


Fig. 23.140: SDRAM_CONTROLLER_TCCD_WR

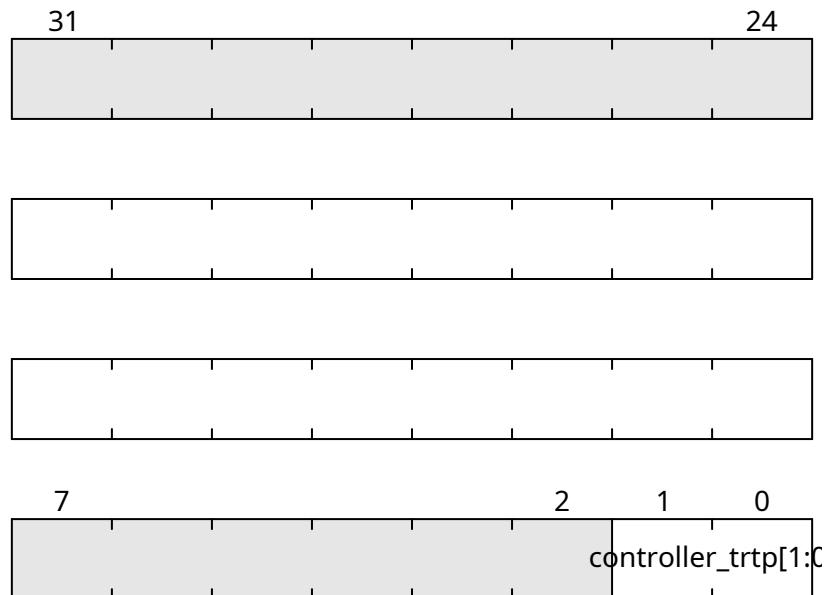


Fig. 23.141: SDRAM_CONTROLLER_TRTP

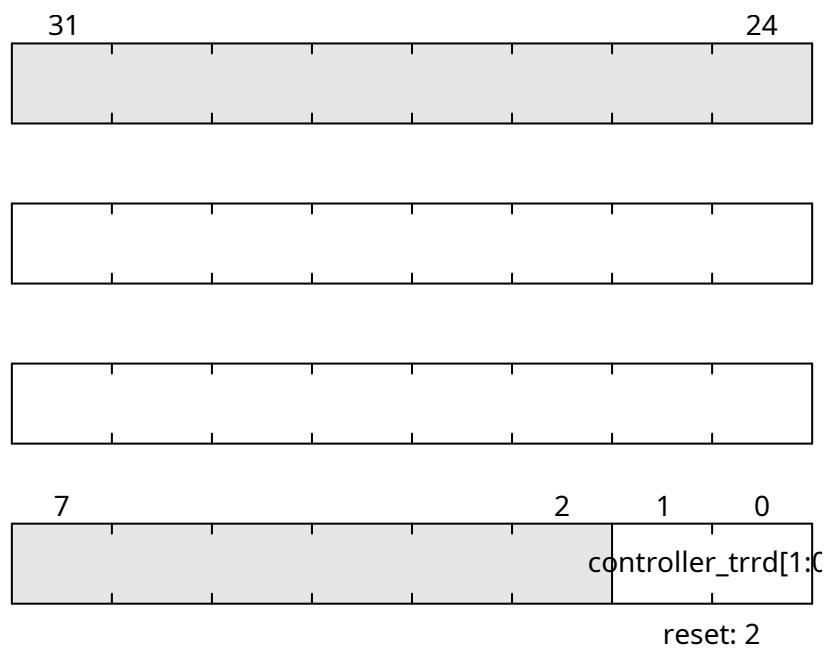


Fig. 23.142: SDRAM_CONTROLLER_TRRD

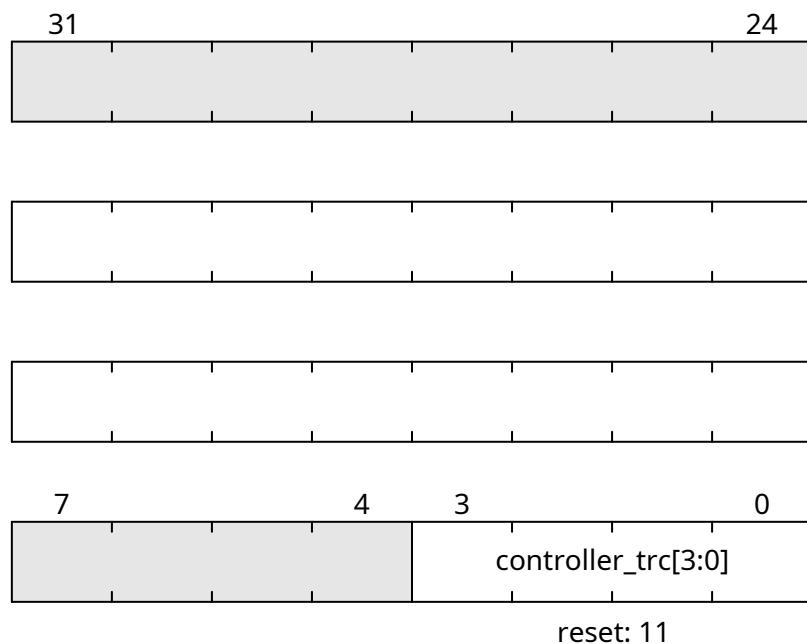


Fig. 23.143: SDRAM CONTROLLER TRC

SDRAM_CONTROLLER_TRAS

Address: $0xf0006800 + 0x80 = 0xf0006880$

SDRAM_CONTROLLER_LAST_ADDR_0

Address: $0xf0006800 + 0x84 = 0xf0006884$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0

Address: $0xf0006800 + 0x88 = 0xf0006888$

SDRAM CONTROLLER LAST ADDR 1

Address: $0xf0006800 + 0x8c = 0xf000688c$

SDRAM CONTROLLER LAST ACTIVE ROW 1

Address: $0xf0006800 + 0x90 = 0xf0006890$

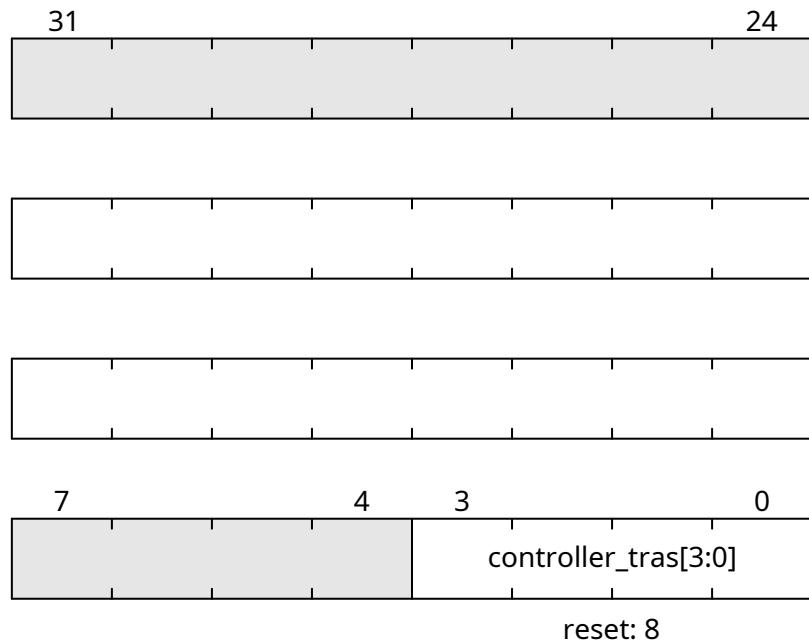


Fig. 23.144: SDRAM_CONTROLLER_TRAS

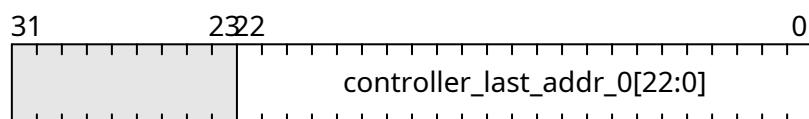


Fig. 23.145: SDRAM_CONTROLLER_LAST_ADDR_0

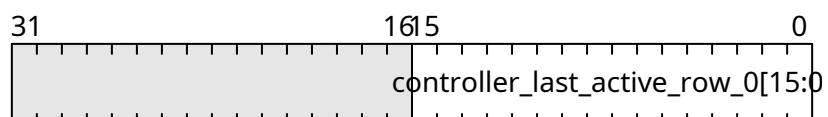


Fig. 23.146: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0

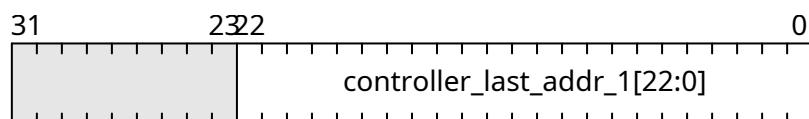


Fig. 23.147: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1

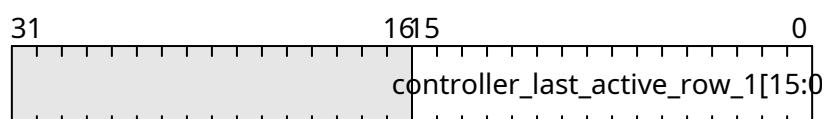


Fig. 23.148: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1

SDRAM_CONTROLLER_LAST_ADDR_2

Address: $0xf0006800 + 0x94 = 0xf0006894$

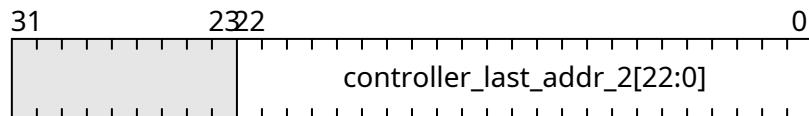


Fig. 23.149: SDRAM_CONTROLLER_LAST_ADDR_2

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2

Address: $0xf0006800 + 0x98 = 0xf0006898$

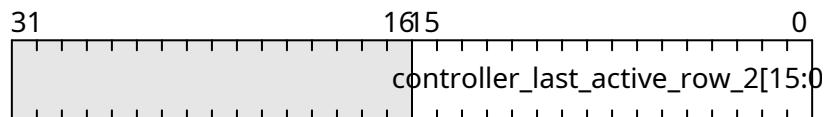


Fig. 23.150: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2

SDRAM_CONTROLLER_LAST_ADDR_3

Address: $0xf0006800 + 0x9c = 0xf000689c$

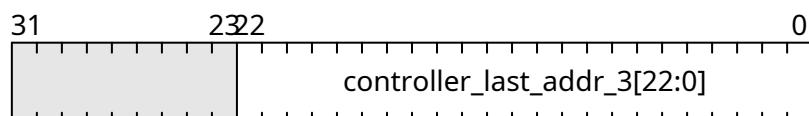


Fig. 23.151: SDRAM_CONTROLLER_LAST_ADDR_3

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3

Address: $0xf0006800 + 0xa0 = 0xf00068a0$

SDRAM_CONTROLLER_LAST_ADDR_4

Address: $0xf0006800 + 0xa4 = 0xf00068a4$

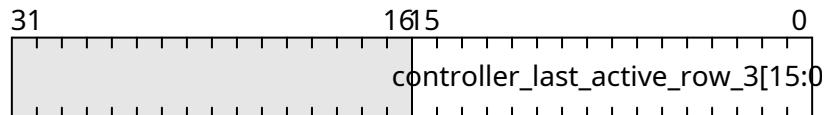


Fig. 23.152: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3

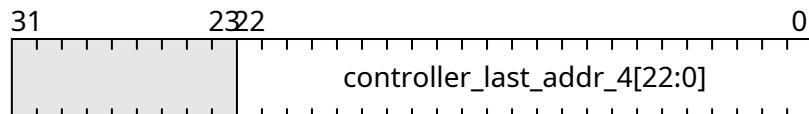


Fig. 23.153: SDRAM_CONTROLLER_LAST_ADDR_4

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4

Address: $0xf0006800 + 0xa8 = 0xf00068a8$

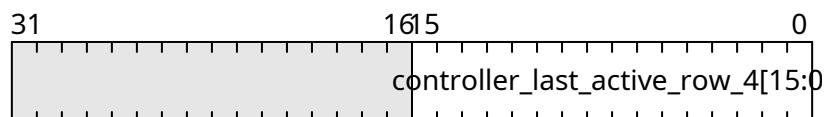


Fig. 23.154: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4

SDRAM_CONTROLLER_LAST_ADDR_5

Address: $0xf0006800 + 0xac = 0xf00068ac$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5

Address: $0xf0006800 + 0xb0 = 0xf00068b0$

SDRAM_CONTROLLER_LAST_ADDR_6

Address: $0xf0006800 + 0xb4 = 0xf00068b4$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6

Address: $0xf0006800 + 0xb8 = 0xf00068b8$

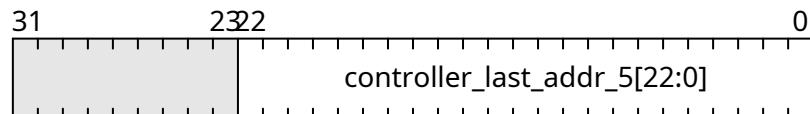


Fig. 23.155: SDRAM_CONTROLLER_LAST_ADDR_5

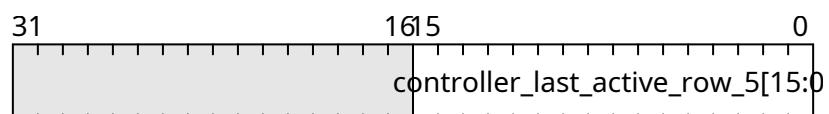


Fig. 23.156: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5

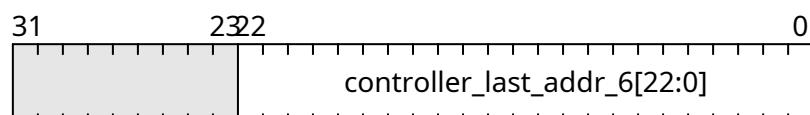


Fig. 23.157: SDRAM_CONTROLLER_LAST_ADDR_6

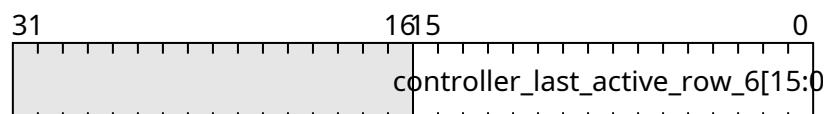


Fig. 23.158: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6

SDRAM_CONTROLLER_LAST_ADDR_7

Address: $0xf0006800 + 0xbc = 0xf00068bc$

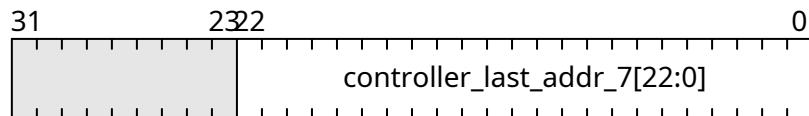


Fig. 23.159: SDRAM_CONTROLLER_LAST_ADDR_7

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7

Address: $0xf0006800 + 0xc0 = 0xf00068c0$

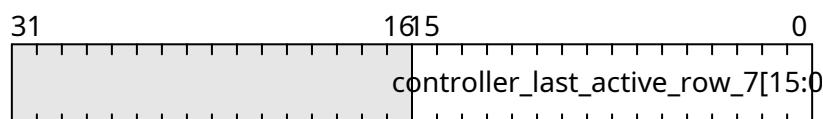


Fig. 23.160: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7

SDRAM_CONTROLLER_LAST_ADDR_8

Address: $0xf0006800 + 0xc4 = 0xf00068c4$

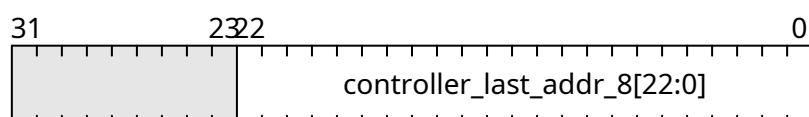


Fig. 23.161: SDRAM_CONTROLLER_LAST_ADDR_8

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_8

Address: $0xf0006800 + 0xc8 = 0xf00068c8$

SDRAM_CONTROLLER_LAST_ADDR_9

Address: $0xf0006800 + 0xcc = 0xf00068cc$

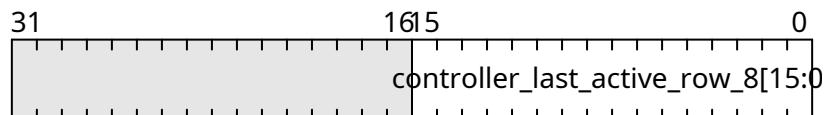


Fig. 23.162: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_8

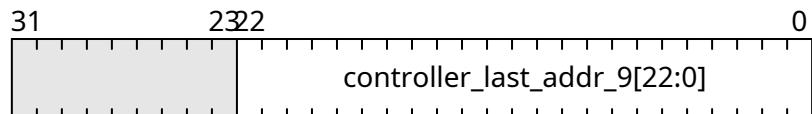


Fig. 23.163: SDRAM_CONTROLLER_LAST_ADDR_9

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_9

Address: $0xf0006800 + 0xd0 = 0xf00068d0$

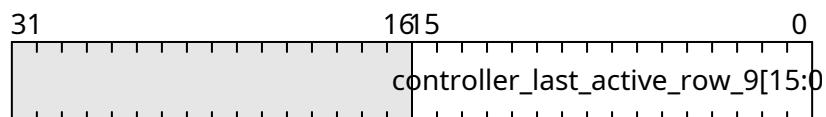


Fig. 23.164: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_9

SDRAM_CONTROLLER_LAST_ADDR_10

Address: $0xf0006800 + 0xd4 = 0xf00068d4$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_10

Address: $0xf0006800 + 0xd8 = 0xf00068d8$

SDRAM_CONTROLLER_LAST_ADDR_11

Address: $0xf0006800 + 0xdc = 0xf00068dc$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_11

Address: $0xf0006800 + 0xe0 = 0xf00068e0$

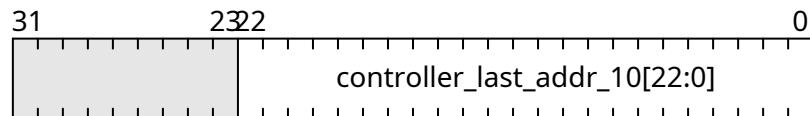


Fig. 23.165: SDRAM_CONTROLLER_LAST_ADDR_10

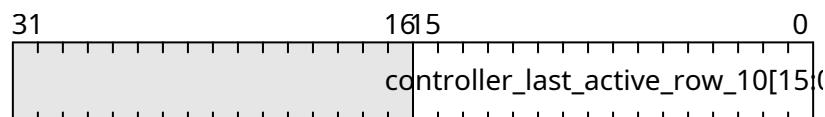


Fig. 23.166: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_10

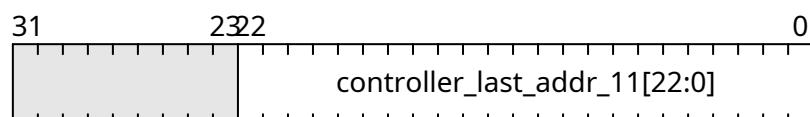


Fig. 23.167: SDRAM_CONTROLLER_LAST_ADDR_11

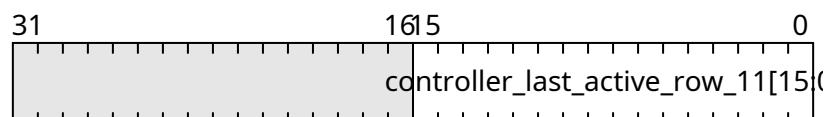


Fig. 23.168: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_11

SDRAM_CONTROLLER_LAST_ADDR_12

Address: $0xf0006800 + 0xe4 = 0xf00068e4$

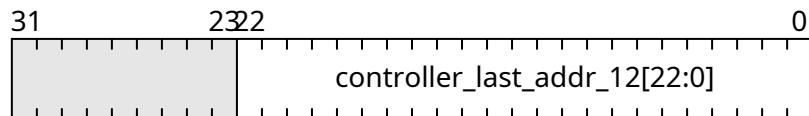


Fig. 23.169: SDRAM_CONTROLLER_LAST_ADDR_12

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_12

Address: $0xf0006800 + 0xe8 = 0xf00068e8$

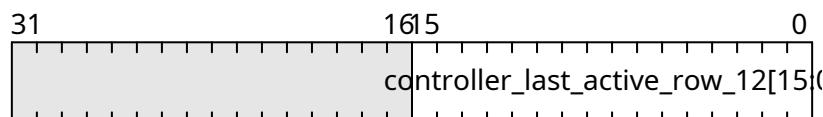


Fig. 23.170: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_12

SDRAM_CONTROLLER_LAST_ADDR_13

Address: $0xf0006800 + 0xec = 0xf00068ec$

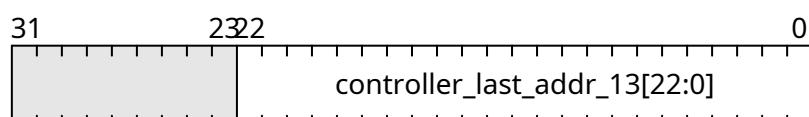


Fig. 23.171: SDRAM_CONTROLLER_LAST_ADDR_13

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_13

Address: $0xf0006800 + 0xf0 = 0xf00068f0$

SDRAM_CONTROLLER_LAST_ADDR_14

Address: $0xf0006800 + 0xf4 = 0xf00068f4$

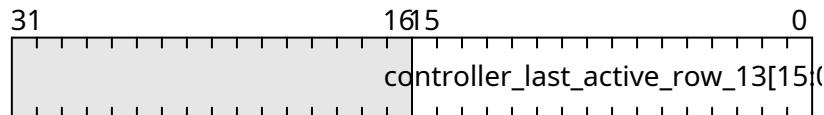


Fig. 23.172: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_13

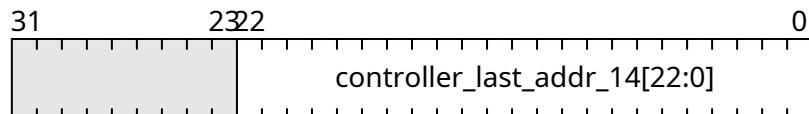


Fig. 23.173: SDRAM_CONTROLLER_LAST_ADDR_14

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_14

Address: $0xf0006800 + 0xf8 = 0xf00068f8$

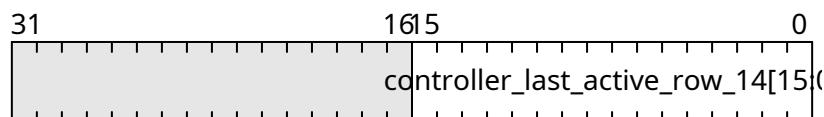


Fig. 23.174: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_14

SDRAM_CONTROLLER_LAST_ADDR_15

Address: $0xf0006800 + 0xfc = 0xf00068fc$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_15

Address: $0xf0006800 + 0x100 = 0xf0006900$

SDRAM_CONTROLLER_LAST_ADDR_16

Address: $0xf0006800 + 0x104 = 0xf0006904$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_16

Address: $0xf0006800 + 0x108 = 0xf0006908$

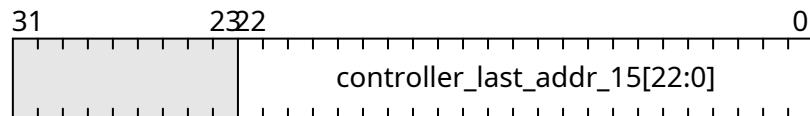


Fig. 23.175: SDRAM_CONTROLLER_LAST_ADDR_15

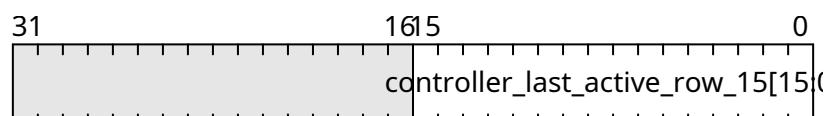


Fig. 23.176: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_15

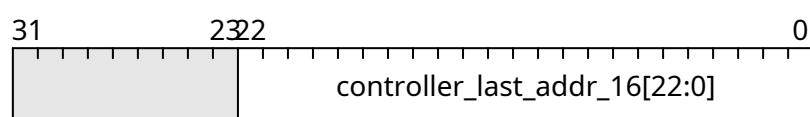


Fig. 23.177: SDRAM_CONTROLLER_LAST_ADDR_16

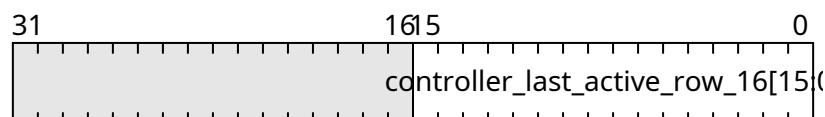


Fig. 23.178: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_16

SDRAM_CONTROLLER_LAST_ADDR_17

Address: $0xf0006800 + 0x10c = 0xf000690c$

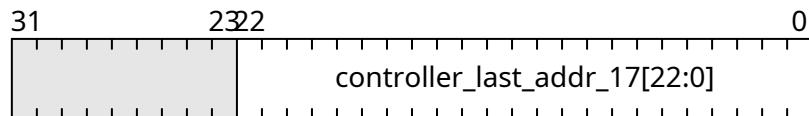


Fig. 23.179: SDRAM_CONTROLLER_LAST_ADDR_17

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_17

Address: $0xf0006800 + 0x110 = 0xf0006910$

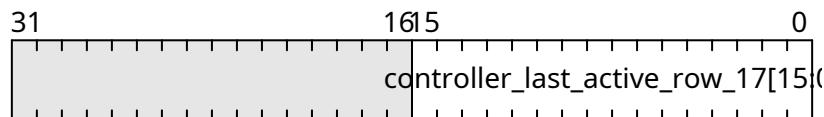


Fig. 23.180: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_17

SDRAM_CONTROLLER_LAST_ADDR_18

Address: $0xf0006800 + 0x114 = 0xf0006914$

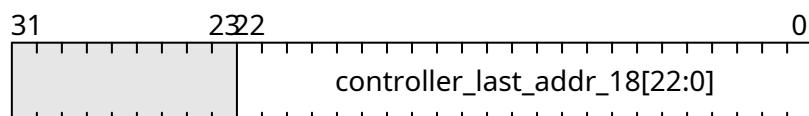


Fig. 23.181: SDRAM_CONTROLLER_LAST_ADDR_18

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_18

Address: $0xf0006800 + 0x118 = 0xf0006918$

SDRAM_CONTROLLER_LAST_ADDR_19

Address: $0xf0006800 + 0x11c = 0xf000691c$

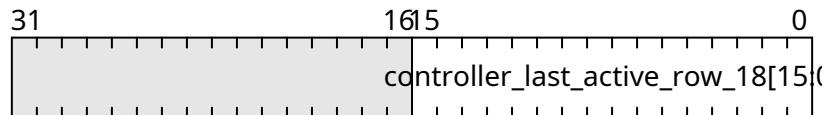


Fig. 23.182: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_18

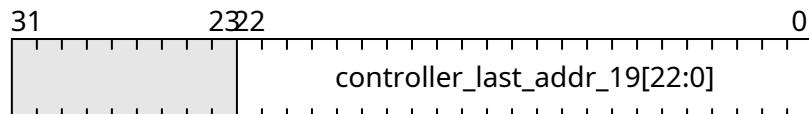


Fig. 23.183: SDRAM_CONTROLLER_LAST_ADDR_19

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_19

Address: $0xf0006800 + 0x120 = 0xf0006920$

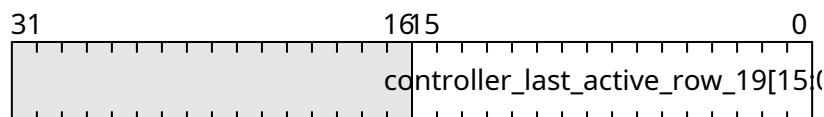


Fig. 23.184: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_19

SDRAM_CONTROLLER_LAST_ADDR_20

Address: $0xf0006800 + 0x124 = 0xf0006924$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_20

Address: $0xf0006800 + 0x128 = 0xf0006928$

SDRAM_CONTROLLER_LAST_ADDR_21

Address: $0xf0006800 + 0x12c = 0xf000692c$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_21

Address: $0xf0006800 + 0x130 = 0xf0006930$

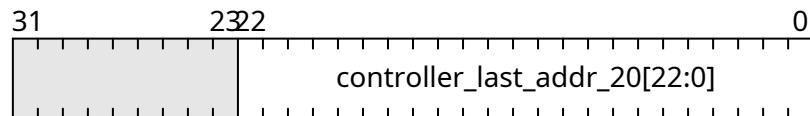


Fig. 23.185: SDRAM_CONTROLLER_LAST_ADDR_20

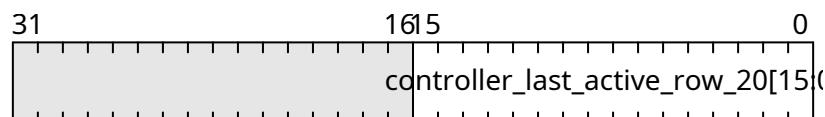


Fig. 23.186: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_20

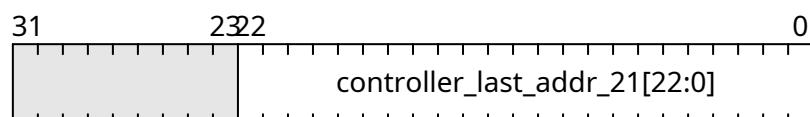


Fig. 23.187: SDRAM_CONTROLLER_LAST_ADDR_21

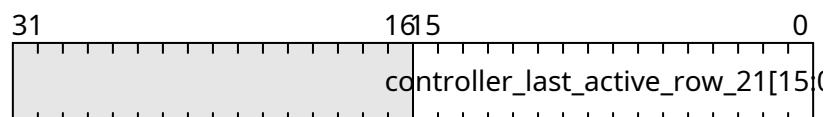


Fig. 23.188: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_21

SDRAM_CONTROLLER_LAST_ADDR_22

Address: $0xf0006800 + 0x134 = 0xf0006934$

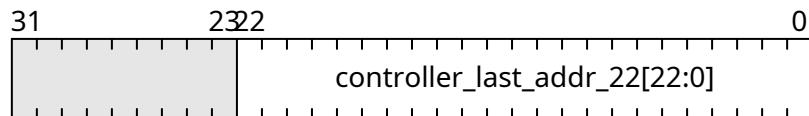


Fig. 23.189: SDRAM_CONTROLLER_LAST_ADDR_22

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_22

Address: $0xf0006800 + 0x138 = 0xf0006938$

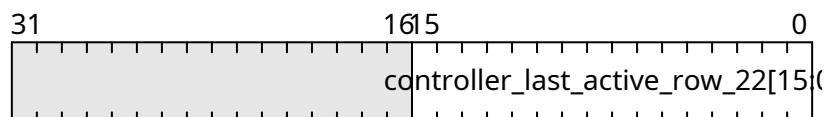


Fig. 23.190: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_22

SDRAM_CONTROLLER_LAST_ADDR_23

Address: $0xf0006800 + 0x13c = 0xf000693c$

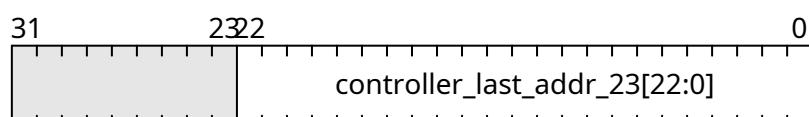


Fig. 23.191: SDRAM_CONTROLLER_LAST_ADDR_23

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_23

Address: $0xf0006800 + 0x140 = 0xf0006940$

SDRAM_CONTROLLER_LAST_ADDR_24

Address: $0xf0006800 + 0x144 = 0xf0006944$

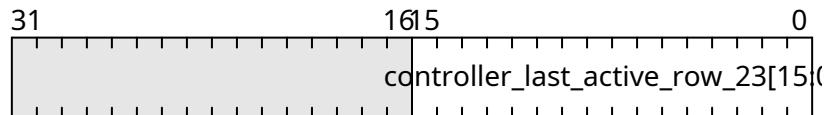


Fig. 23.192: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_23

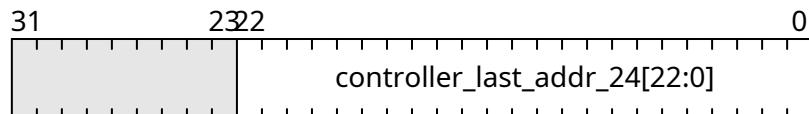


Fig. 23.193: SDRAM_CONTROLLER_LAST_ADDR_24

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_24

Address: $0xf0006800 + 0x148 = 0xf0006948$

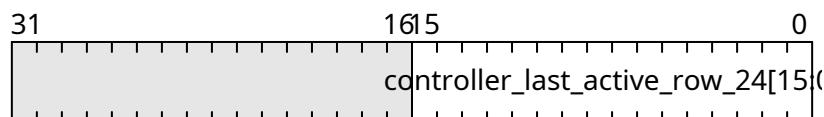


Fig. 23.194: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_24

SDRAM_CONTROLLER_LAST_ADDR_25

Address: $0xf0006800 + 0x14c = 0xf000694c$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_25

Address: $0xf0006800 + 0x150 = 0xf0006950$

SDRAM_CONTROLLER_LAST_ADDR_26

Address: $0xf0006800 + 0x154 = 0xf0006954$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_26

Address: $0xf0006800 + 0x158 = 0xf0006958$

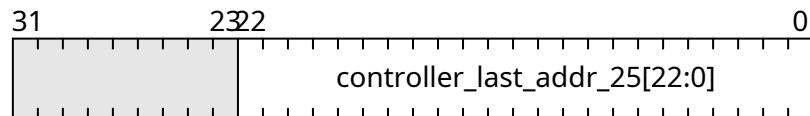


Fig. 23.195: SDRAM_CONTROLLER_LAST_ADDR_25

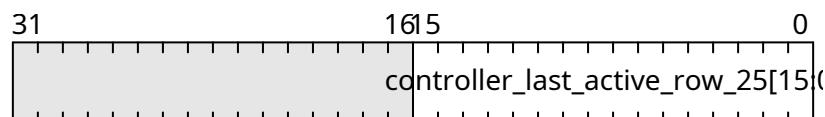


Fig. 23.196: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_25

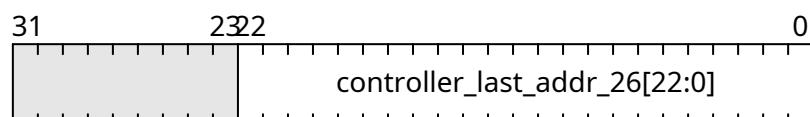


Fig. 23.197: SDRAM_CONTROLLER_LAST_ADDR_26

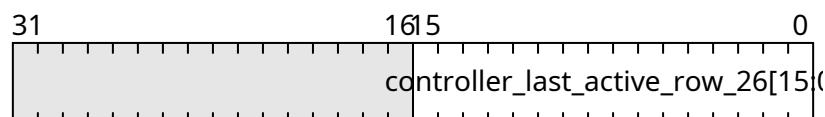


Fig. 23.198: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_26

SDRAM_CONTROLLER_LAST_ADDR_27

Address: $0xf0006800 + 0x15c = 0xf000695c$

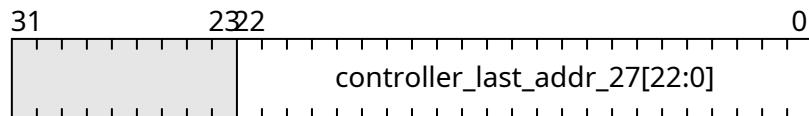


Fig. 23.199: SDRAM_CONTROLLER_LAST_ADDR_27

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_27

Address: $0xf0006800 + 0x160 = 0xf0006960$

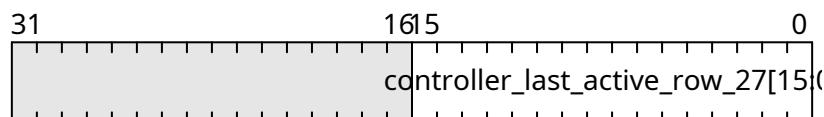


Fig. 23.200: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_27

SDRAM_CONTROLLER_LAST_ADDR_28

Address: $0xf0006800 + 0x164 = 0xf0006964$

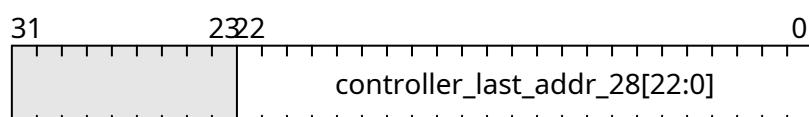


Fig. 23.201: SDRAM_CONTROLLER_LAST_ADDR_28

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_28

Address: $0xf0006800 + 0x168 = 0xf0006968$

SDRAM_CONTROLLER_LAST_ADDR_29

Address: $0xf0006800 + 0x16c = 0xf000696c$

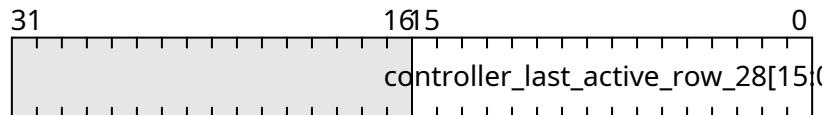


Fig. 23.202: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_28

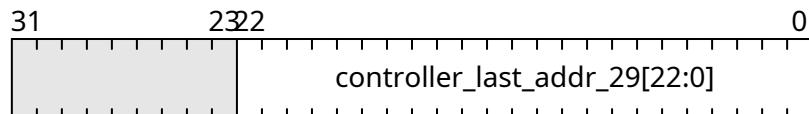


Fig. 23.203: SDRAM_CONTROLLER_LAST_ADDR_29

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_29

Address: $0xf0006800 + 0x170 = 0xf0006970$

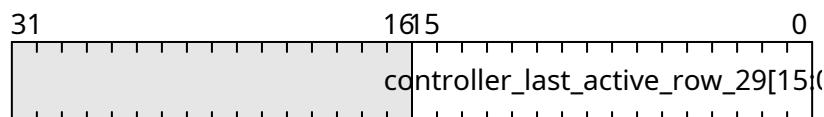


Fig. 23.204: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_29

SDRAM_CONTROLLER_LAST_ADDR_30

Address: $0xf0006800 + 0x174 = 0xf0006974$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_30

Address: $0xf0006800 + 0x178 = 0xf0006978$

SDRAM_CONTROLLER_LAST_ADDR_31

Address: $0xf0006800 + 0x17c = 0xf000697c$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_31

Address: $0xf0006800 + 0x180 = 0xf0006980$

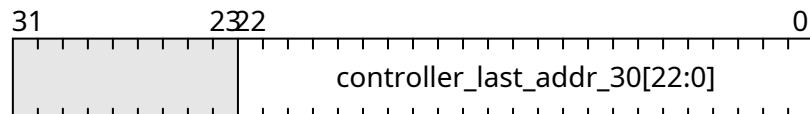


Fig. 23.205: SDRAM_CONTROLLER_LAST_ADDR_30

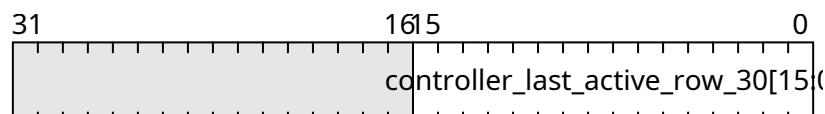


Fig. 23.206: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_30

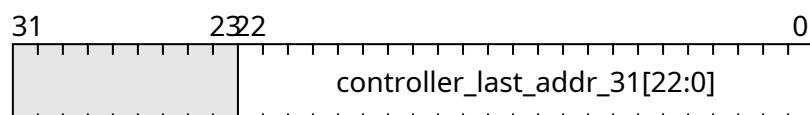


Fig. 23.207: SDRAM_CONTROLLER_LAST_ADDR_31

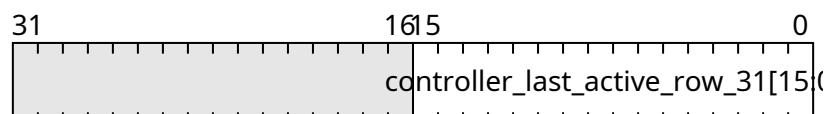


Fig. 23.208: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_31

23.2.15 SDRAM_CHECKER

Register Listing for SDRAM_CHECKER

Register	Address
<i>SDRAM_CHECKER_RESET</i>	0xf0007000
<i>SDRAM_CHECKER_START</i>	0xf0007004
<i>SDRAM_CHECKER_DONE</i>	0xf0007008
<i>SDRAM_CHECKER_BASE</i>	0xf000700c
<i>SDRAM_CHECKER_END</i>	0xf0007010
<i>SDRAM_CHECKER_LENGTH</i>	0xf0007014
<i>SDRAM_CHECKER_RANDOM</i>	0xf0007018
<i>SDRAM_CHECKER_TICKS</i>	0xf000701c
<i>SDRAM_CHECKER_ERRORS</i>	0xf0007020

SDRAM_CHECKER_RESET

Address: $0xf0007000 + 0x0 = 0xf0007000$



Fig. 23.209: SDRAM_CHECKER_RESET

SDRAM_CHECKER_START

Address: $0xf0007000 + 0x4 = 0xf0007004$



Fig. 23.210: SDRAM_CHECKER_START

SDRAM_CHECKER_DONE

Address: 0xf0007000 + 0x8 = 0xf0007008

SDRAM_CHECKER_BASE

Address: 0xf0007000 + 0xc = 0xf000700c

SDRAM_CHECKER_END

Address: 0xf0007000 + 0x10 = 0xf0007010

SDRAM_CHECKER_LENGTH

Address: 0xf0007000 + 0x14 = 0xf0007014

SDRAM_CHECKER_RANDOM

Address: 0xf0007000 + 0x18 = 0xf0007018



Fig. 23.211: SDRAM_CHECKER_DONE

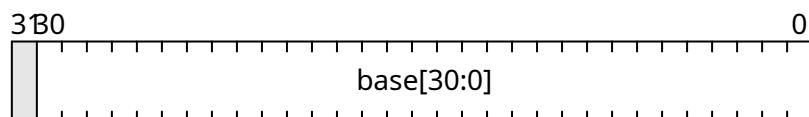


Fig. 23.212: SDRAM_CHECKER_BASE

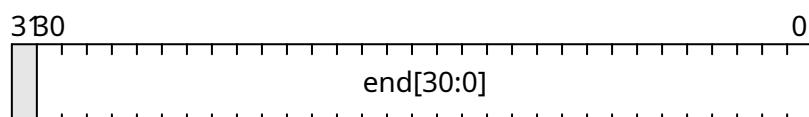


Fig. 23.213: SDRAM_CHECKER_END

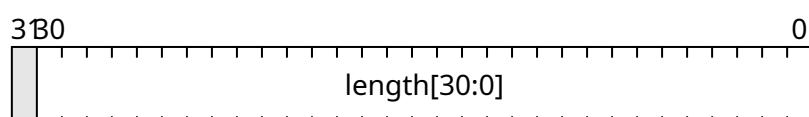


Fig. 23.214: SDRAM_CHECKER_LENGTH



Fig. 23.215: SDRAM_CHECKER_RANDOM

Field	Name	Description

SDRAM_CHECKER_TICKS

Address: $0xf0007000 + 0x1c = 0xf000701c$



Fig. 23.216: SDRAM_CHECKER_TICKS

SDRAM_CHECKER_ERRORS

Address: $0xf0007000 + 0x20 = 0xf0007020$

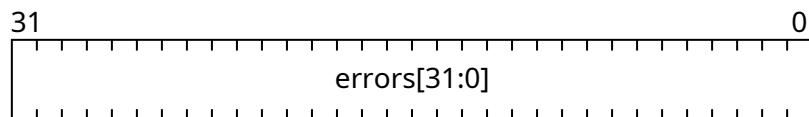


Fig. 23.217: SDRAM_CHECKER_ERRORS

23.2.16 SDRAM_GENERATOR

Register Listing for SDRAM_GENERATOR

Register	Address
<i>SDRAM_GENERATOR_RESET</i>	0xf0007800
<i>SDRAM_GENERATOR_START</i>	0xf0007804
<i>SDRAM_GENERATOR_DONE</i>	0xf0007808
<i>SDRAM_GENERATOR_BASE</i>	0xf000780c
<i>SDRAM_GENERATOR_END</i>	0xf0007810
<i>SDRAM_GENERATOR_LENGTH</i>	0xf0007814
<i>SDRAM_GENERATOR_RANDOM</i>	0xf0007818
<i>SDRAM_GENERATOR_TICKS</i>	0xf000781c

SDRAM_GENERATOR_RESET

Address: $0xf0007800 + 0x0 = 0xf0007800$



Fig. 23.218: SDRAM_GENERATOR_RESET

SDRAM_GENERATOR_START

Address: $0xf0007800 + 0x4 = 0xf0007804$



Fig. 23.219: SDRAM_GENERATOR_START

SDRAM_GENERATOR_DONE

Address: $0xf0007800 + 0x8 = 0xf0007808$

SDRAM_GENERATOR_BASE

Address: $0xf0007800 + 0xc = 0xf000780c$

SDRAM_GENERATOR_END

Address: $0xf0007800 + 0x10 = 0xf0007810$

SDRAM_GENERATOR_LENGTH

Address: $0xf0007800 + 0x14 = 0xf0007814$

SDRAM_GENERATOR_RANDOM

Address: $0xf0007800 + 0x18 = 0xf0007818$



Fig. 23.220: SDRAM_GENERATOR_DONE

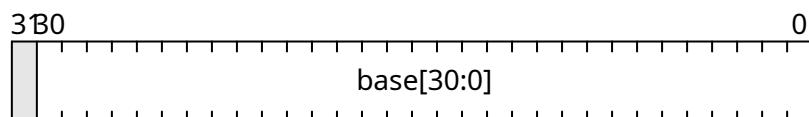


Fig. 23.221: SDRAM_GENERATOR_BASE

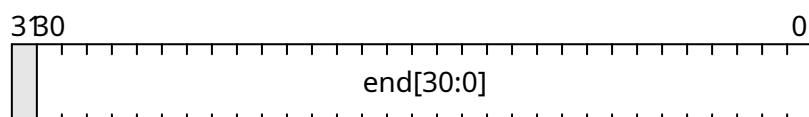


Fig. 23.222: SDRAM_GENERATOR_END

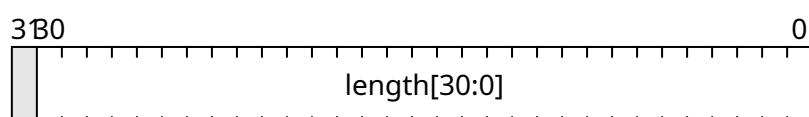


Fig. 23.223: SDRAM_GENERATOR_LENGTH



Fig. 23.224: SDRAM_GENERATOR_RANDOM

Field	Name	Description

SDRAM_GENERATOR_TICKS

Address: $0xf0007800 + 0x1c = 0xf000781c$

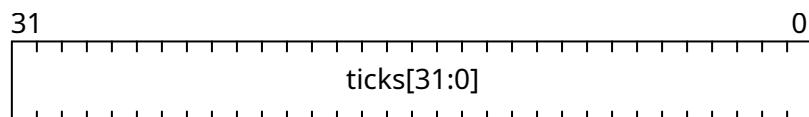


Fig. 23.225: SDRAM_GENERATOR_TICKS

23.2.17 TIMER0

Timer

Provides a generic Timer core.

The Timer is implemented as a countdown timer that can be used in various modes:

- Polling : Returns current countdown value to software
- One-Shot: Loads itself and stops when value reaches 0
- Periodic: (Re-)Loads itself when value reaches 0

en register allows the user to enable/disable the Timer. When the Timer is enabled, it is automatically loaded with the value of *load* register.

When the Timer reaches 0, it is automatically reloaded with value of *reload* register.

The user can latch the current countdown value by writing to *update_value* register, it will update *value* register with current countdown value.

To use the Timer in One-Shot mode, the user needs to:

- Disable the timer
- Set the *load* register to the expected duration
- (Re-)Enable the Timer

To use the Timer in Periodic mode, the user needs to:

- Disable the Timer
- Set the *load* register to 0
- Set the *reload* register to the expected period
- Enable the Timer

For both modes, the CPU can be advertised by an IRQ that the duration/period has elapsed. (The CPU can also do software polling with *update_value* and *value* to know the elapsed duration)

Register Listing for TIMERO

Register	Address
<i>TIMERO_LOAD</i>	0xf0008000
<i>TIMERO_RELOAD</i>	0xf0008004
<i>TIMERO_EN</i>	0xf0008008
<i>TIMERO_UPDATE_VALUE</i>	0xf000800c
<i>TIMERO_VALUE</i>	0xf0008010
<i>TIMERO_EV_STATUS</i>	0xf0008014
<i>TIMERO_EV_PENDING</i>	0xf0008018
<i>TIMERO_EV_ENABLE</i>	0xf000801c

TIMERO_LOAD

Address: $0xf0008000 + 0x0 = 0xf0008000$

Load value when Timer is (re-)enabled. In One-Shot mode, the value written to this register specifies the Timer's duration in clock cycles.

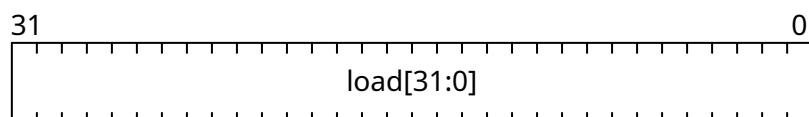


Fig. 23.226: TIMERO_LOAD

TIMER0_RELOAD

Address: $0xf0008000 + 0x4 = 0xf0008004$

Reload value when Timer reaches 0. In Periodic mode, the value written to this register specify the Timer's period in clock cycles.

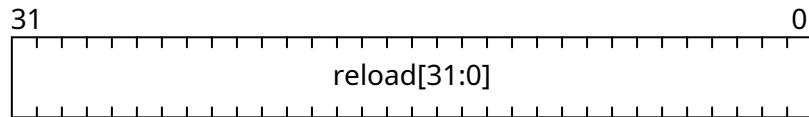


Fig. 23.227: TIMER0_RELOAD

TIMER0_EN

Address: $0xf0008000 + 0x8 = 0xf0008008$

Enable flag of the Timer. Set this flag to 1 to enable/start the Timer. Set to 0 to disable the Timer.



Fig. 23.228: TIMER0_EN

TIMER0_UPDATE_VALUE

Address: $0xf0008000 + 0xc = 0xf000800c$

Update trigger for the current countdown value. A write to this register latches the current countdown value to value register.



Fig. 23.229: TIMERO_UPDATE_VALUE

TIMERO_VALUE

Address: 0xf0008000 + 0x10 = 0xf0008010

Latched countdown value. This value is updated by writing to update_value.

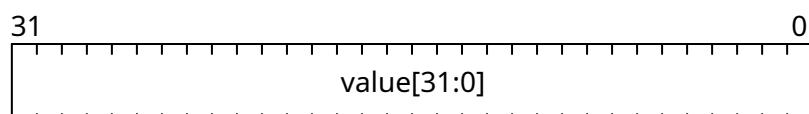


Fig. 23.230: TIMERO_VALUE

TIMERO_EV_STATUS

Address: 0xf0008000 + 0x14 = 0xf0008014

This register contains the current raw level of the zero event trigger. Writes to this register have no effect.

Field	Name	Description
[0]	ZERO	Level of the zero event

TIMERO_EV_PENDING

Address: 0xf0008000 + 0x18 = 0xf0008018

When a zero event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.



Fig. 23.231: TIMER0_EV_STATUS



Fig. 23.232: TIMER0_EV_PENDING

Field	Name	Description
[0]	ZERO	1 if a <i>zero</i> event occurred. This Event is triggered on a falling edge.

TIMER0_EV_ENABLE

Address: $0xf0008000 + 0x1c = 0xf000801c$

This register enables the corresponding zero events. Write a 0 to this register to disable individual events.

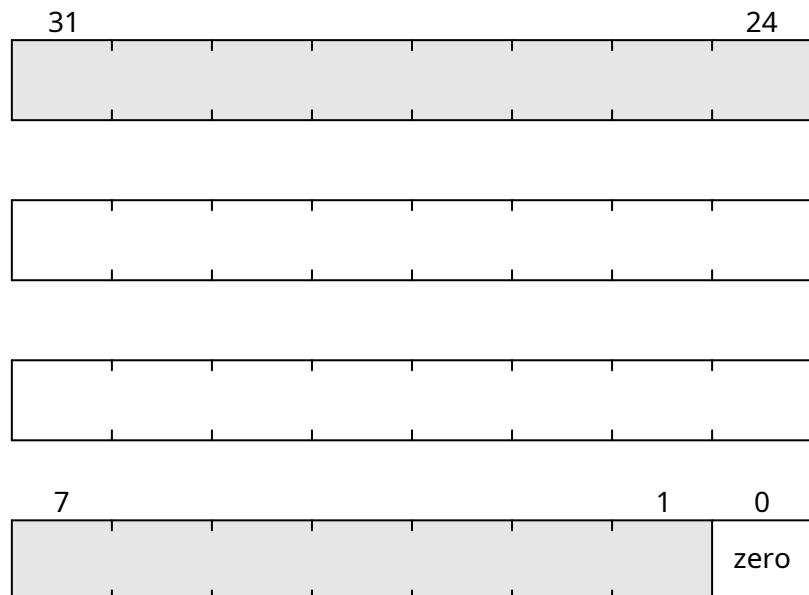


Fig. 23.233: TIMER0_EV_ENABLE

Field	Name	Description
[0]	ZERO	Write a 1 to enable the zero Event

23.2.18 UART

Register Listing for UART

Register	Address
<i>UART_RXTX</i>	0xf0008800
<i>UART_TXFULL</i>	0xf0008804
<i>UART_RXEMPTY</i>	0xf0008808
<i>UART_EV_STATUS</i>	0xf000880c
<i>UART_EV_PENDING</i>	0xf0008810
<i>UART_EV_ENABLE</i>	0xf0008814
<i>UART_RXEMPTY</i>	0xf0008818
<i>UART_RXFULL</i>	0xf000881c
<i>UART_XOVER_RXTX</i>	0xf0008820
<i>UART_XOVER_TXFULL</i>	0xf0008824
<i>UART_XOVER_RXEMPTY</i>	0xf0008828
<i>UART_XOVER_EV_STATUS</i>	0xf000882c
<i>UART_XOVER_EV_PENDING</i>	0xf0008830
<i>UART_XOVER_EV_ENABLE</i>	0xf0008834
<i>UART_XOVER_RXEMPTY</i>	0xf0008838
<i>UART_XOVER_RXFULL</i>	0xf000883c

UART_RXTX

Address: $0xf0008800 + 0x0 = 0xf0008800$

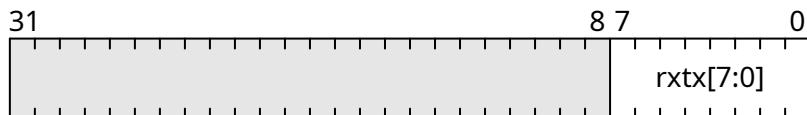


Fig. 23.234: UART_RXTX

UART_TXFULL

Address: $0xf0008800 + 0x4 = 0xf0008804$

TX FIFO Full.

UART_RXEMPTY

Address: $0xf0008800 + 0x8 = 0xf0008808$

RX FIFO Empty.

UART_EV_STATUS

Address: $0xf0008800 + 0xc = 0xf000880c$

This register contains the current raw level of the rx event trigger. Writes to this register have no effect.



Fig. 23.235: UART_TXFULL



Fig. 23.236: UART_RXEMPTY

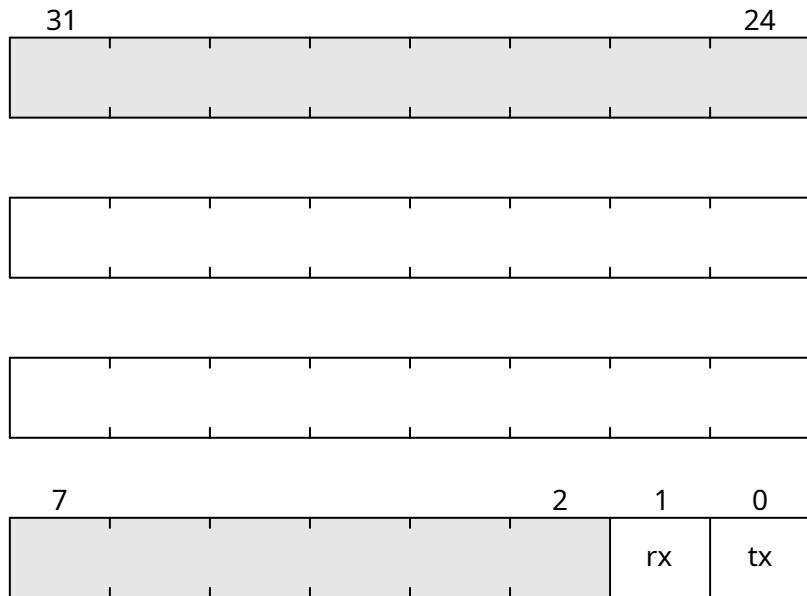


Fig. 23.237: `UART_EV_STATUS`

Field	Name	Description
[0]	TX	Level of the tx event
[1]	RX	Level of the rx event

UART_EV_PENDING

Address: $0xf0008800 + 0x10 = 0xf0008810$

When a rx event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.

Field	Name	Description
[0]	TX	1 if a tx event occurred. This Event is triggered on a falling edge .
[1]	RX	1 if a rx event occurred. This Event is triggered on a falling edge .

UART_EV_ENABLE

Address: $0xf0008800 + 0x14 = 0xf0008814$

This register enables the corresponding rx events. Write a 0 to this register to disable individual events.

Field	Name	Description
[0]	TX	Write a 1 to enable the tx Event
[1]	RX	Write a 1 to enable the rx Event

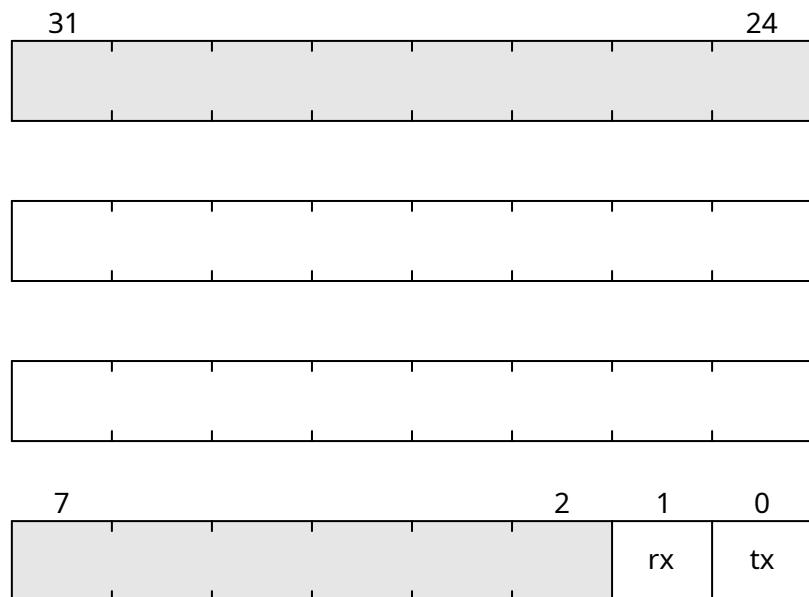


Fig. 23.238: `UART_EV_PENDING`

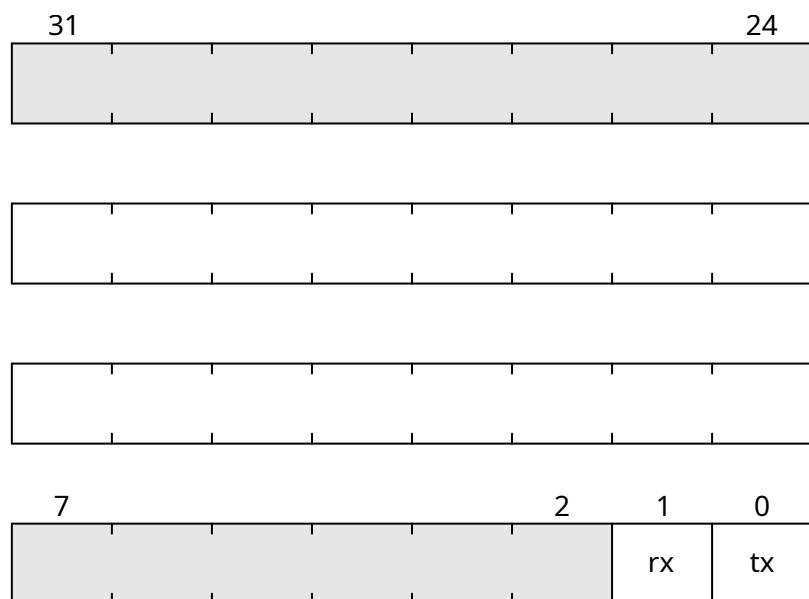


Fig. 23.239: `UART_EV_ENABLE`

UART TXEMPTY

Address: $0xf0008800 + 0x18 = 0xf0008818$

TX FIFO Empty.



Fig. 23.240: UART TXEMPTY

UART_RXFULL

Address: $0xf0008800 + 0x1c = 0xf000881c$

RX FIFO Full.

UART XOVER RXTX

Address: $0xf0008800 + 0x20 = 0xf0008820$

UART_XOVER_TXFULL

Address: $0xf0008800 + 0x24 = 0xf0008824$

TX FIFO Full.

UART_XOVER_RXEMPTY

Address: 0xf0008800 + 0x28 = 0xf0008828

RX FIFO Empty.



Fig. 23.241: `UART_RXFULL`

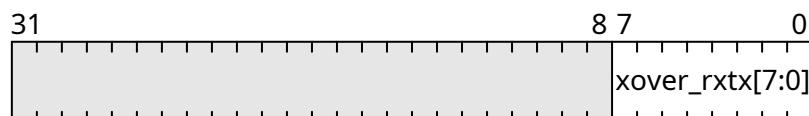


Fig. 23.242: `UART_XOVER_RXTX`

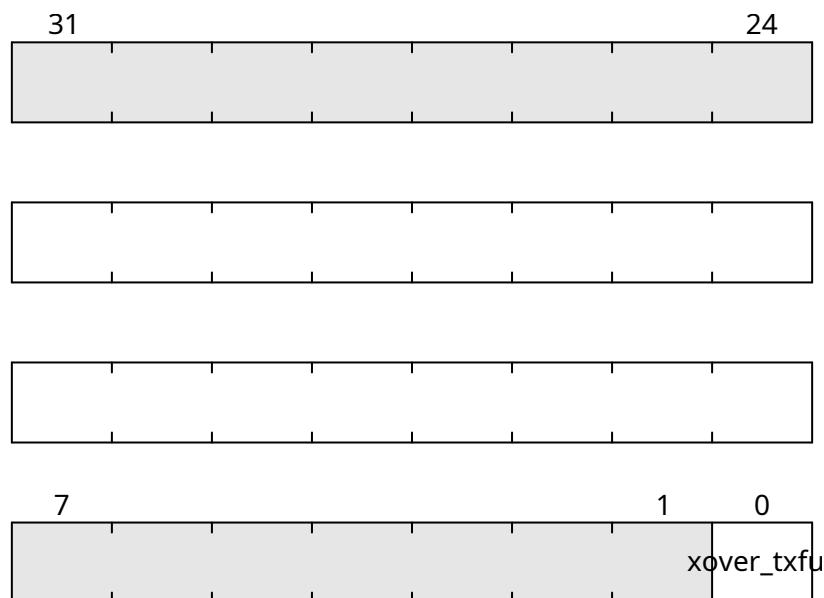


Fig. 23.243: `UART_XOVER_TXFULL`



Fig. 23.244: UART_XOVER_RXEMPTY

UART_XOVER_EV_STATUS

Address: 0xf0008800 + 0x2c = 0xf000882c

This register contains the current raw level of the rx event trigger. Writes to this register have no effect.

Field	Name	Description
[0]	TX	Level of the tx event
[1]	RX	Level of the rx event

UART_XOVER_EV_PENDING

Address: 0xf0008800 + 0x30 = 0xf0008830

When a rx event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.

Field	Name	Description
[0]	TX	1 if a tx event occurred. This Event is triggered on a falling edge.
[1]	RX	1 if a rx event occurred. This Event is triggered on a falling edge.

UART_XOVER_EV_ENABLE

Address: 0xf0008800 + 0x34 = 0xf0008834

This register enables the corresponding rx events. Write a 0 to this register to disable individual events.

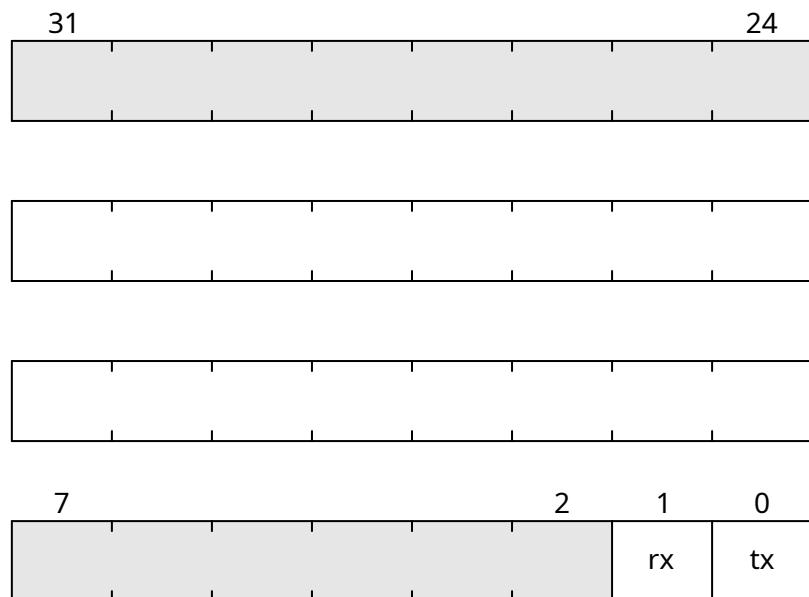


Fig. 23.245: `UART_XOVER_EV_STATUS`

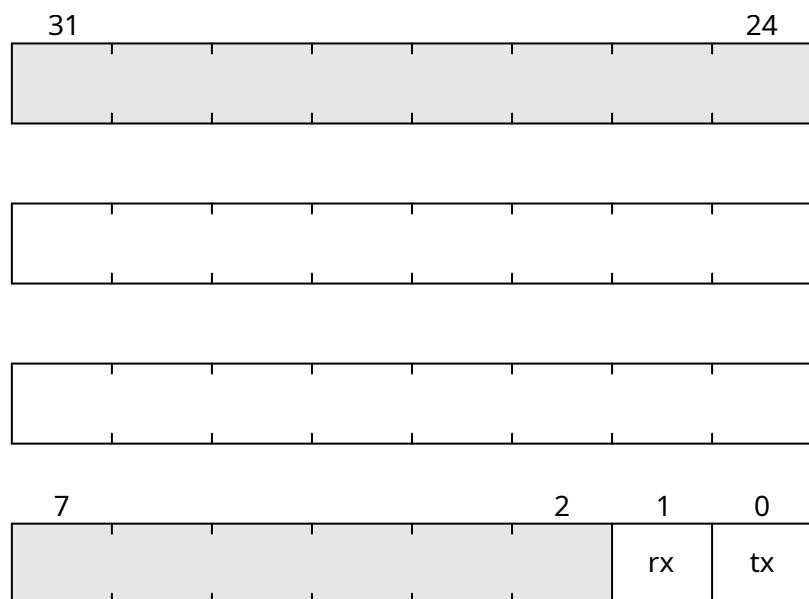


Fig. 23.246: `UART_XOVER_EV_PENDING`

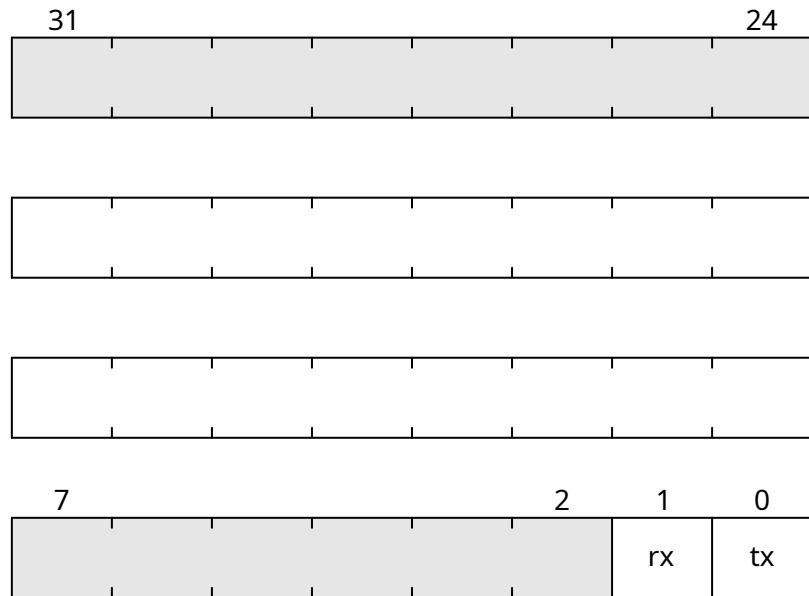


Fig. 23.247: **UART_XOVER_EV_ENABLE**

Field	Name	Description
[0]	TX	Write a 1 to enable the tx Event
[1]	RX	Write a 1 to enable the rx Event

UART_XOVER_TXEMPTY

Address: $0xf0008800 + 0x38 = 0xf0008838$

TX FIFO Empty.

UART_XOVER_RXFULL

Address: $0xf0008800 + 0x3c = 0xf000883c$

RX FIFO Full.



Fig. 23.248: `UART_XOVER_TXEMPTY`

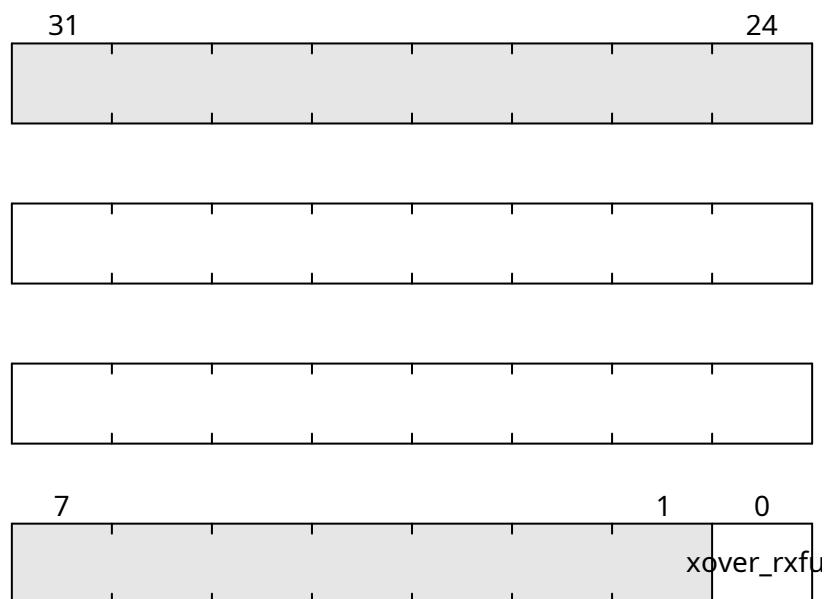


Fig. 23.249: `UART_XOVER_RXFULL`

CHAPTER
TWENTYFOUR

DOCUMENTATION FOR ROW HAMMER TESTER DDR5 TESTER

24.1 Modules

24.1.1 Interrupt Controller

This device has an EventManager-based interrupt system. Individual modules generate *events* which are wired into a central interrupt controller.

When an interrupt occurs, you should look the interrupt number up in the CPU- specific interrupt table and then call the relevant module.

Assigned Interrupts

The following interrupts are assigned on this system:

Interrupt	Module
1	<i>TIMER0</i>
0	<i>UART</i>

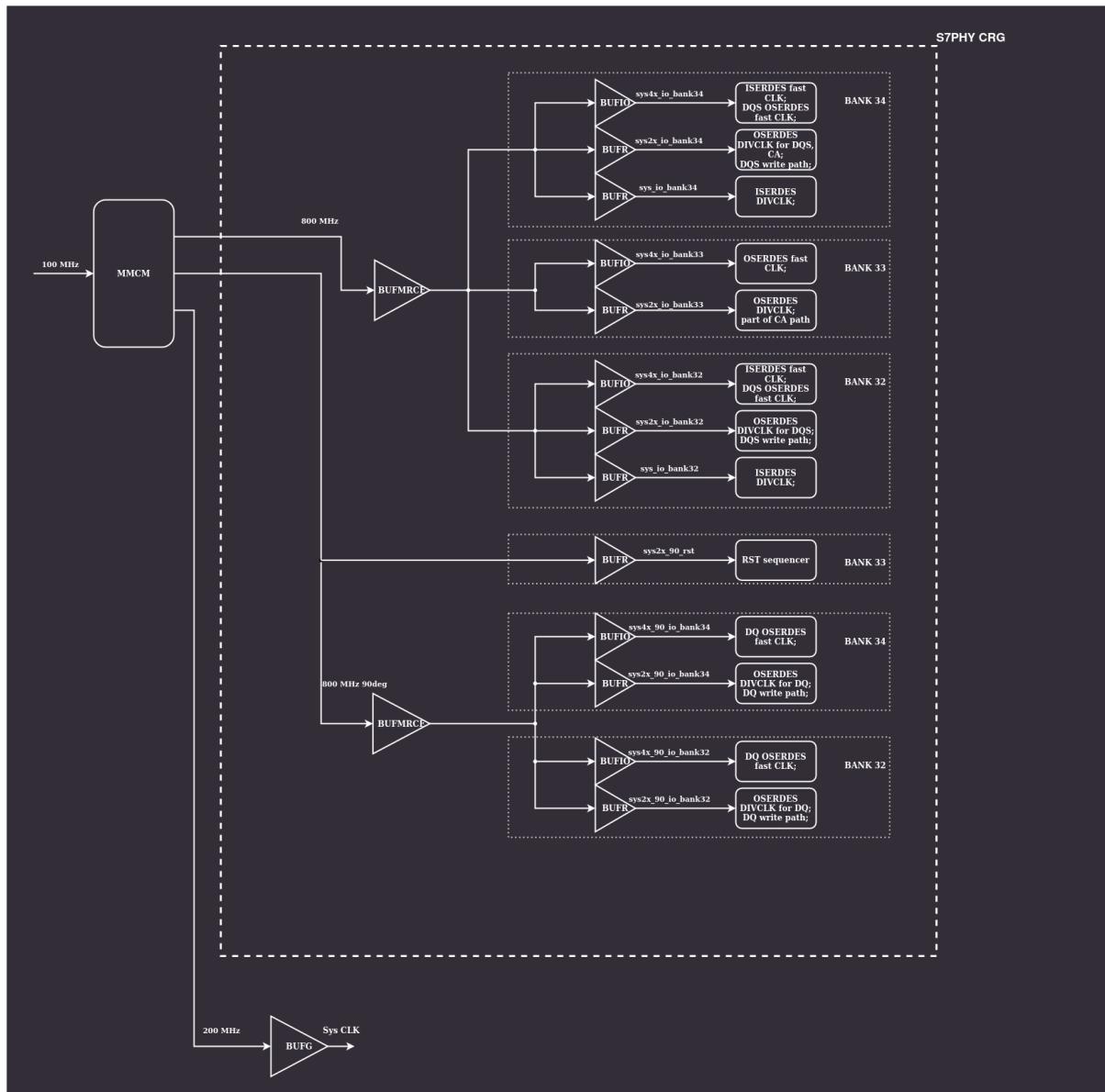
24.1.2 PHYCRG

S7CRGPHY

This module contains 7 series specific clock and reset generation for S7DDR5 PHY. It adds:

- BUFMRCE to control multi region PHYs (UDIMMs and/or RDIMMs),
- BUFMRCE/BUFRs reset sequence,
- ISERDES reset sequence correct with Xilinx documentation and design advisories,
- OSERDES reset sequence.

DDR5 Tester Clock tree



24.2 Register Groups

24.2.1 LEDs

Register Listing for LEDs

Register	Address
<i>LEDS_OUT</i>	<i>0xf0000000</i>

LEDS_OUT

Address: $0xf0000000 + 0x0 = 0xf0000000$

Led Output(s) Control.

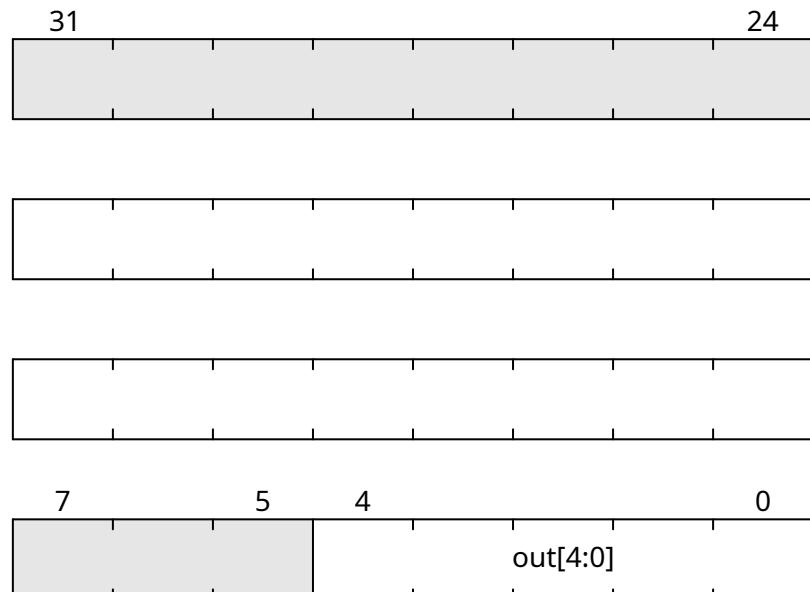


Fig. 24.1: LEDS_OUT

24.2.2 DDRPHY

Register Listing for DDRPHY

Register	Address
<i>DDRPHY_CSRMODULE_ENABLE_FIFOS</i>	<i>0xf0000800</i>
<i>DDRPHY_CSRMODULE_RST</i>	<i>0xf0000804</i>
<i>DDRPHY_CSRMODULE_RDIMM_MODE</i>	<i>0xf0000808</i>
<i>DDRPHY_CSRMODULE_RDPHASE</i>	<i>0xf000080c</i>
<i>DDRPHY_CSRMODULE_WRPHASE</i>	<i>0xf0000810</i>
<i>DDRPHY_CSRMODULE_ALERT</i>	<i>0xf0000814</i>
<i>DDRPHY_CSRMODULE_ALERT_REDUCE</i>	<i>0xf0000818</i>
<i>DDRPHY_CSRMODULE_SAMPLE_ALERT</i>	<i>0xf000081c</i>
<i>DDRPHY_CSRMODULE_RESET_ALERT</i>	<i>0xf0000820</i>
<i>DDRPHY_CSRMODULE_CKDLY_RST</i>	<i>0xf0000824</i>
<i>DDRPHY_CSRMODULE_CKDLY_INC</i>	<i>0xf0000828</i>
<i>DDRPHY_CSRMODULE_A_PREAMBLE</i>	<i>0xf000082c</i>
<i>DDRPHY_CSRMODULE_A_WLEVEL_EN</i>	<i>0xf0000830</i>
<i>DDRPHY_CSRMODULE_A_PAR_ENABLE</i>	<i>0xf0000834</i>
<i>DDRPHY_CSRMODULE_A_PAR_VALUE</i>	<i>0xf0000838</i>
<i>DDRPHY_CSRMODULE_A_DISCARD_RD_FIFO</i>	<i>0xf000083c</i>
<i>DDRPHY_CSRMODULE_A_DLY_SEL</i>	<i>0xf0000840</i>
<i>DDRPHY_CSRMODULE_A_DQ_DQS_RATIO</i>	<i>0xf0000844</i>

continues on next page

Table 24.1 – continued from previous page

Register	Address
<i>DDRPHY_CSRMODULE_A_CK_RDLY_INC</i>	0xf0000848
<i>DDRPHY_CSRMODULE_A_CK_RDLY_RST</i>	0xf000084c
<i>DDRPHY_CSRMODULE_A_CK_RDDLY</i>	0xf0000850
<i>DDRPHY_CSRMODULE_A_CK_RDDLY_PREAMBLE</i>	0xf0000854
<i>DDRPHY_CSRMODULE_A_CK_WDLY_INC</i>	0xf0000858
<i>DDRPHY_CSRMODULE_A_CK_WDLY_RST</i>	0xf000085c
<i>DDRPHY_CSRMODULE_A_CK_WDLY_DQS</i>	0xf0000860
<i>DDRPHY_CSRMODULE_A_CK_WDDLY_INC</i>	0xf0000864
<i>DDRPHY_CSRMODULE_A_CK_WDDLY_RST</i>	0xf0000868
<i>DDRPHY_CSRMODULE_A_CK_WDLY_DQ</i>	0xf000086c
<i>DDRPHY_CSRMODULE_A_DQ_DLY_SEL</i>	0xf0000870
<i>DDRPHY_CSRMODULE_A_CSDLY_RST</i>	0xf0000874
<i>DDRPHY_CSRMODULE_A_CSDLY_INC</i>	0xf0000878
<i>DDRPHY_CSRMODULE_A_CADLY_RST</i>	0xf000087c
<i>DDRPHY_CSRMODULE_A_CADLY_INC</i>	0xf0000880
<i>DDRPHY_CSRMODULE_A_PARDLY_RST</i>	0xf0000884
<i>DDRPHY_CSRMODULE_A_PARDLY_INC</i>	0xf0000888
<i>DDRPHY_CSRMODULE_A_CSDLY</i>	0xf000088c
<i>DDRPHY_CSRMODULE_A_CADLY</i>	0xf0000890
<i>DDRPHY_CSRMODULE_A_RDLY_DQ_RST</i>	0xf0000894
<i>DDRPHY_CSRMODULE_A_RDLY_DQ_INC</i>	0xf0000898
<i>DDRPHY_CSRMODULE_A_RDLY_DQS_RST</i>	0xf000089c
<i>DDRPHY_CSRMODULE_A_RDLY_DQS_INC</i>	0xf00008a0
<i>DDRPHY_CSRMODULE_A_RDLY_DQS</i>	0xf00008a4
<i>DDRPHY_CSRMODULE_A_RDLY_DQ</i>	0xf00008a8
<i>DDRPHY_CSRMODULE_A_WDLY_DQ_RST</i>	0xf00008ac
<i>DDRPHY_CSRMODULE_A_WDLY_DQ_INC</i>	0xf00008b0
<i>DDRPHY_CSRMODULE_A_WDLY_DM_RST</i>	0xf00008b4
<i>DDRPHY_CSRMODULE_A_WDLY_DM_INC</i>	0xf00008b8
<i>DDRPHY_CSRMODULE_A_WDLY_DQS_RST</i>	0xf00008bc
<i>DDRPHY_CSRMODULE_A_WDLY_DQS_INC</i>	0xf00008c0
<i>DDRPHY_CSRMODULE_A_WDLY_DQS</i>	0xf00008c4
<i>DDRPHY_CSRMODULE_A_WDLY_DQ</i>	0xf00008c8
<i>DDRPHY_CSRMODULE_A_WDLY_DM</i>	0xf00008cc
<i>DDRPHY_CSRMODULE_B_PREAMBLE</i>	0xf00008d0
<i>DDRPHY_CSRMODULE_B_WLEVEL_EN</i>	0xf00008d4
<i>DDRPHY_CSRMODULE_B_PAR_ENABLE</i>	0xf00008d8
<i>DDRPHY_CSRMODULE_B_PAR_VALUE</i>	0xf00008dc
<i>DDRPHY_CSRMODULE_B_DISCARD_RD_FIFO</i>	0xf00008e0
<i>DDRPHY_CSRMODULE_B_DLY_SEL</i>	0xf00008e4
<i>DDRPHY_CSRMODULE_B_DQ_DQS_RATIO</i>	0xf00008e8
<i>DDRPHY_CSRMODULE_B_CK_RDLY_INC</i>	0xf00008ec
<i>DDRPHY_CSRMODULE_B_CK_RDLY_RST</i>	0xf00008f0
<i>DDRPHY_CSRMODULE_B_CK_RDDLY</i>	0xf00008f4
<i>DDRPHY_CSRMODULE_B_CK_RDDLY_PREAMBLE</i>	0xf00008f8
<i>DDRPHY_CSRMODULE_B_CK_WDLY_INC</i>	0xf00008fc
<i>DDRPHY_CSRMODULE_B_CK_WDLY_RST</i>	0xf0000900
<i>DDRPHY_CSRMODULE_B_CK_WDLY_DQS</i>	0xf0000904

continues on next page

Table 24.1 – continued from previous page

Register	Address
<i>DDRPHY_CSRMODULE_B_CK_WDDLY_INC</i>	0xf0000908
<i>DDRPHY_CSRMODULE_B_CK_WDDLY_RST</i>	0xf000090c
<i>DDRPHY_CSRMODULE_B_CK_WDLY_DQ</i>	0xf0000910
<i>DDRPHY_CSRMODULE_B_DQ_DLY_SEL</i>	0xf0000914
<i>DDRPHY_CSRMODULE_B_CSDLY_RST</i>	0xf0000918
<i>DDRPHY_CSRMODULE_B_CSDLY_INC</i>	0xf000091c
<i>DDRPHY_CSRMODULE_B_CADLY_RST</i>	0xf0000920
<i>DDRPHY_CSRMODULE_B_CADLY_INC</i>	0xf0000924
<i>DDRPHY_CSRMODULE_B_PARDLY_RST</i>	0xf0000928
<i>DDRPHY_CSRMODULE_B_PARDLY_INC</i>	0xf000092c
<i>DDRPHY_CSRMODULE_B_CSDLY</i>	0xf0000930
<i>DDRPHY_CSRMODULE_B_CADLY</i>	0xf0000934
<i>DDRPHY_CSRMODULE_B_RDLY_DQ_RST</i>	0xf0000938
<i>DDRPHY_CSRMODULE_B_RDLY_DQ_INC</i>	0xf000093c
<i>DDRPHY_CSRMODULE_B_RDLY_DQS_RST</i>	0xf0000940
<i>DDRPHY_CSRMODULE_B_RDLY_DQS_INC</i>	0xf0000944
<i>DDRPHY_CSRMODULE_B_RDLY_DQS</i>	0xf0000948
<i>DDRPHY_CSRMODULE_B_RDLY_DQ</i>	0xf000094c
<i>DDRPHY_CSRMODULE_B_WDLY_DQ_RST</i>	0xf0000950
<i>DDRPHY_CSRMODULE_B_WDLY_DQ_INC</i>	0xf0000954
<i>DDRPHY_CSRMODULE_B_WDLY_DM_RST</i>	0xf0000958
<i>DDRPHY_CSRMODULE_B_WDLY_DM_INC</i>	0xf000095c
<i>DDRPHY_CSRMODULE_B_WDLY_DQS_RST</i>	0xf0000960
<i>DDRPHY_CSRMODULE_B_WDLY_DQS_INC</i>	0xf0000964
<i>DDRPHY_CSRMODULE_B_WDLY_DQS</i>	0xf0000968
<i>DDRPHY_CSRMODULE_B_WDLY_DQ</i>	0xf000096c
<i>DDRPHY_CSRMODULE_B_WDLY_DM</i>	0xf0000970

DDRPHY_CSRMODULE_ENABLE_FIFOS

Address: 0xf0000800 + 0x0 = 0xf0000800

DDRPHY_CSRMODULE_RST

Address: 0xf0000800 + 0x4 = 0xf0000804

DDRPHY_CSRMODULE_RDIMM_MODE

Address: 0xf0000800 + 0x8 = 0xf0000808

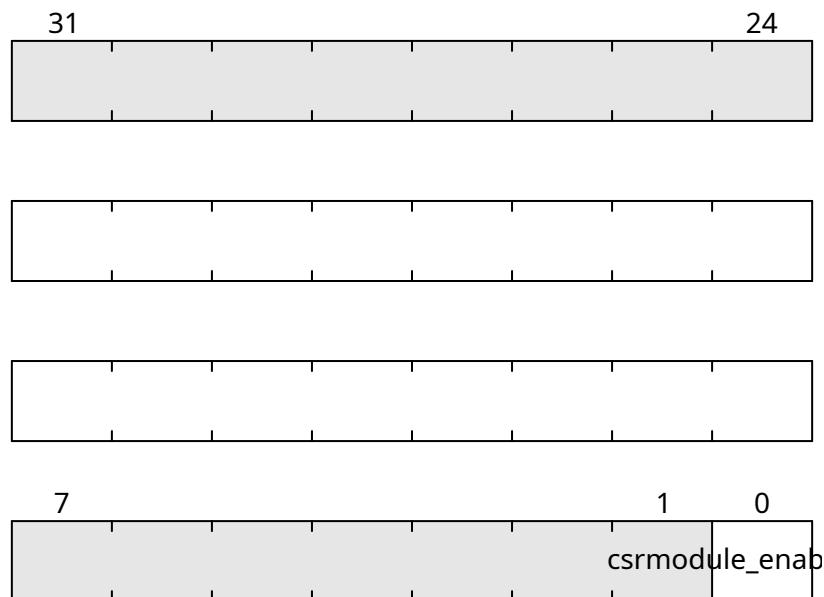


Fig. 24.2: DDRPHY_CSRMODULE_ENABLE_FIFOS

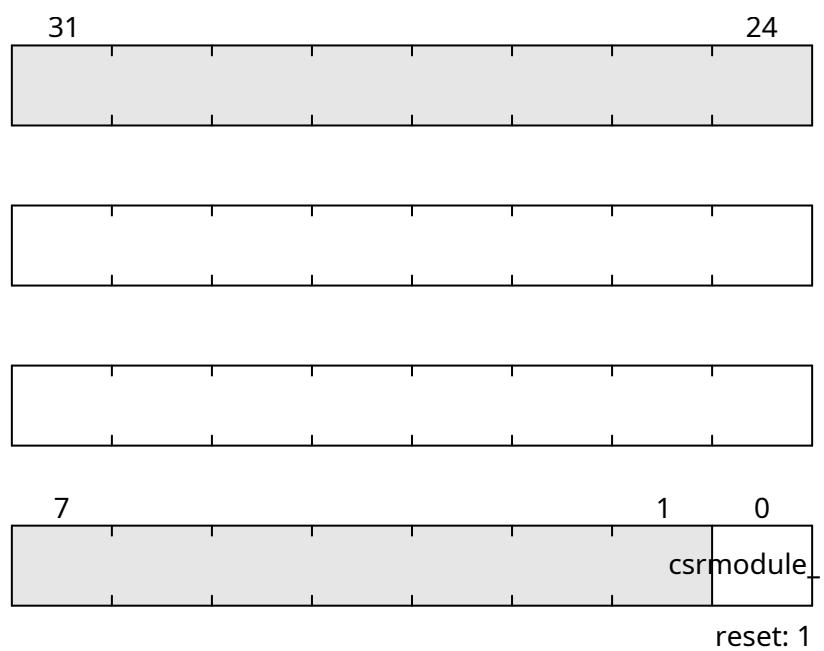


Fig. 24.3: DDRPHY_CSRMODULE_RST

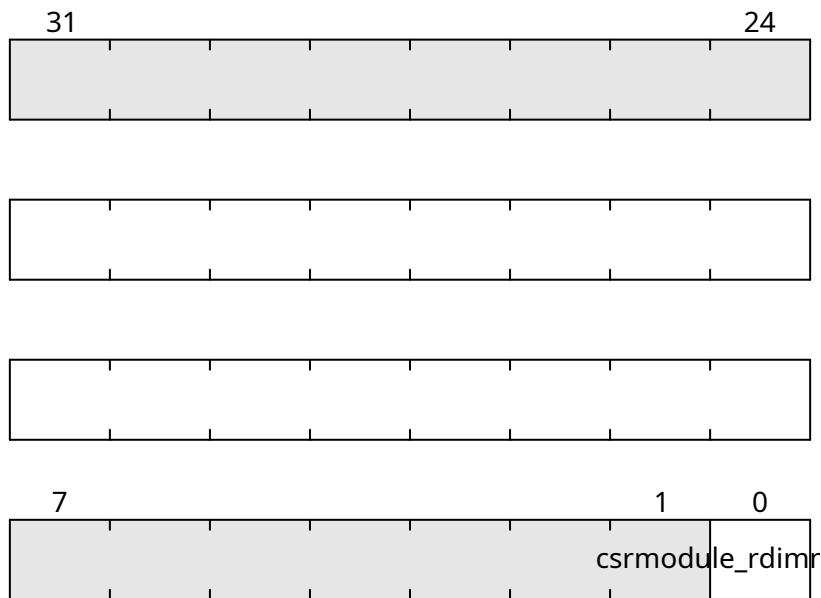


Fig. 24.4: DDRPHY_CSRMODULE_RDIMM_MODE

DDRPHY_CSRMODULE_RDPHASE

Address: $0xf0000800 + 0xc = 0xf000080c$

DDRPHY_CSRMODULE_WRPHASE

Address: $0xf0000800 + 0x10 = 0xf0000810$

DDRPHY_CSRMODULE_ALERT

Address: $0xf0000800 + 0x14 = 0xf0000814$

DDRPHY_CSRMODULE_ALERT_REDUCE

Address: $0xf0000800 + 0x18 = 0xf0000818$

Field	Name	Description
[0]	INITIAL_STATE	Initial value of all bits
[1]	OPERATION	0 - or (default), 1 -and

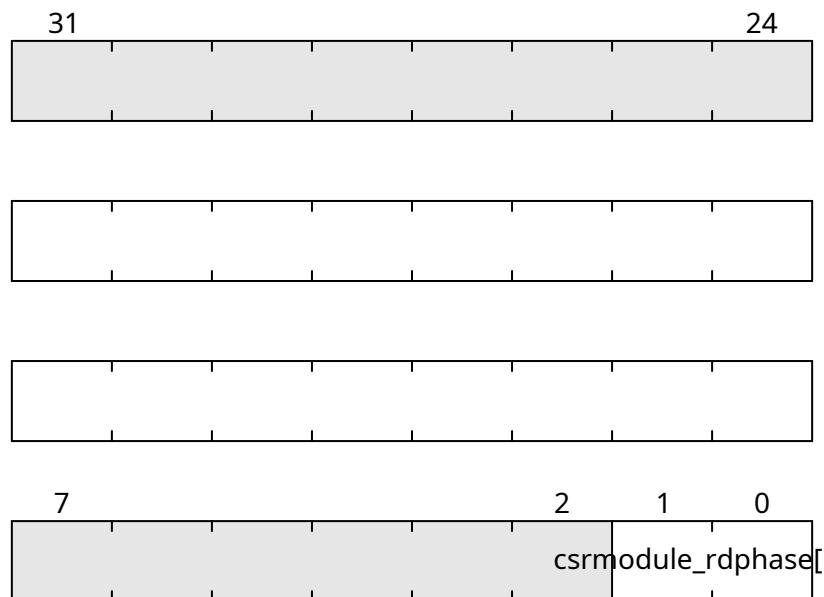


Fig. 24.5: DDRPHY_CSRMODULE_RDPHASE

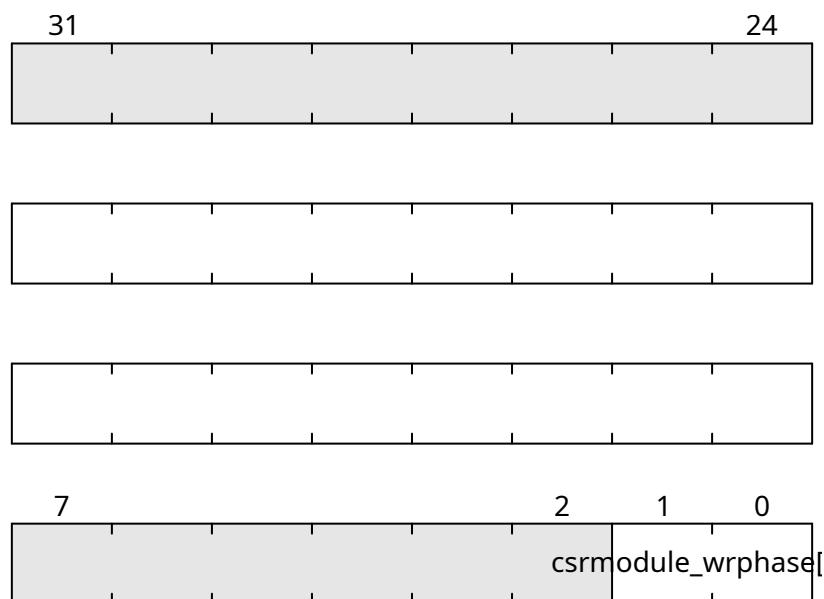


Fig. 24.6: DDRPHY_CSRMODULE_WRPAGE

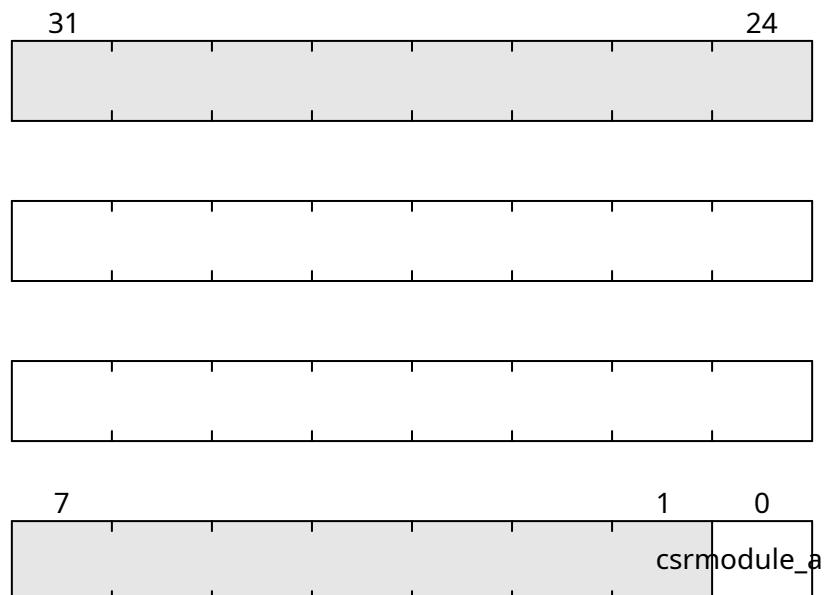


Fig. 24.7: DDRPHY_CSRMODULE_ALERT

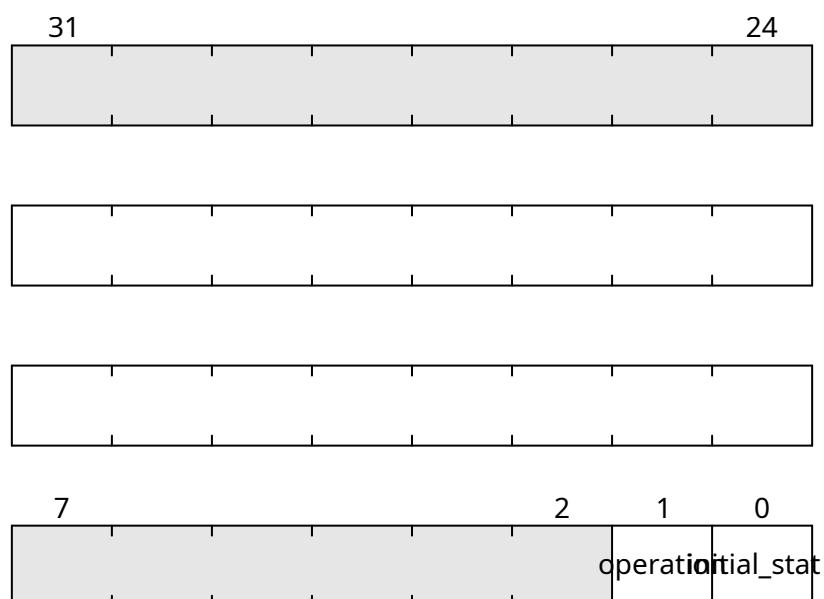


Fig. 24.8: DDRPHY_CSRMODULE_ALERT_REDUCE

DDRPHY_CSRMODULE_SAMPLE_ALERT

Address: $0xf0000800 + 0x1c = 0xf000081c$

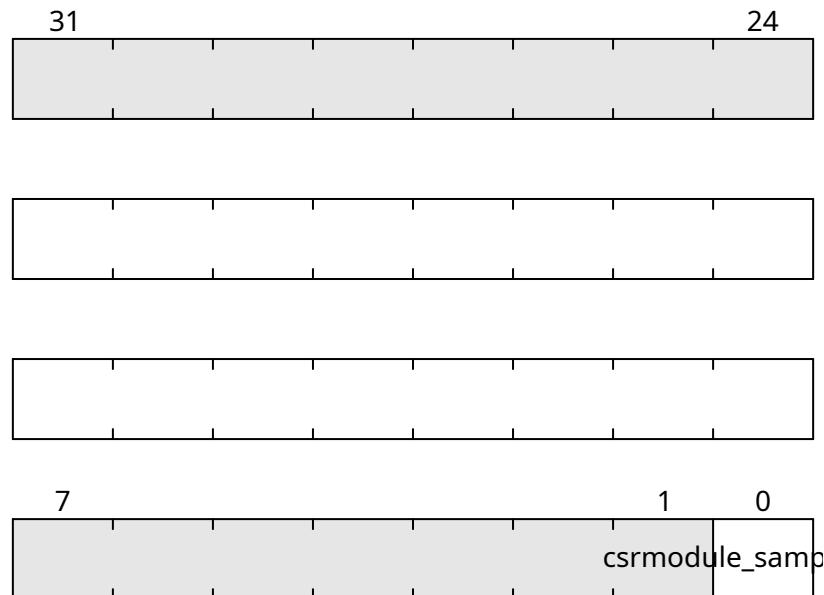


Fig. 24.9: DDRPHY_CSRMODULE_SAMPLE_ALERT

DDRPHY_CSRMODULE_RESET_ALERT

Address: $0xf0000800 + 0x20 = 0xf0000820$

DDRPHY_CSRMODULE_CKDLY_RST

Address: $0xf0000800 + 0x24 = 0xf0000824$

DDRPHY_CSRMODULE_CKDLY_INC

Address: $0xf0000800 + 0x28 = 0xf0000828$

DDRPHY_CSRMODULE_A_PREAMBLE

Address: $0xf0000800 + 0x2c = 0xf000082c$

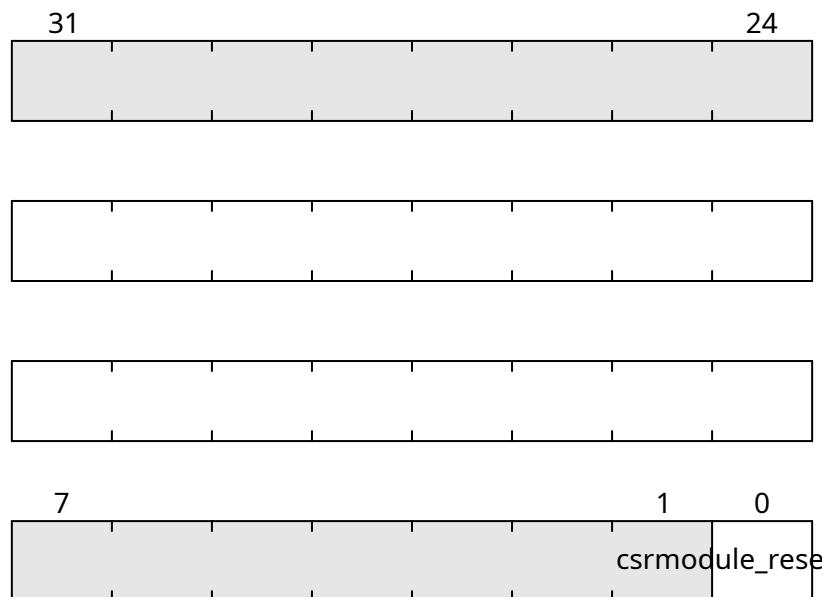


Fig. 24.10: DDRPHY_CSRMODULE_RESET_ALERT

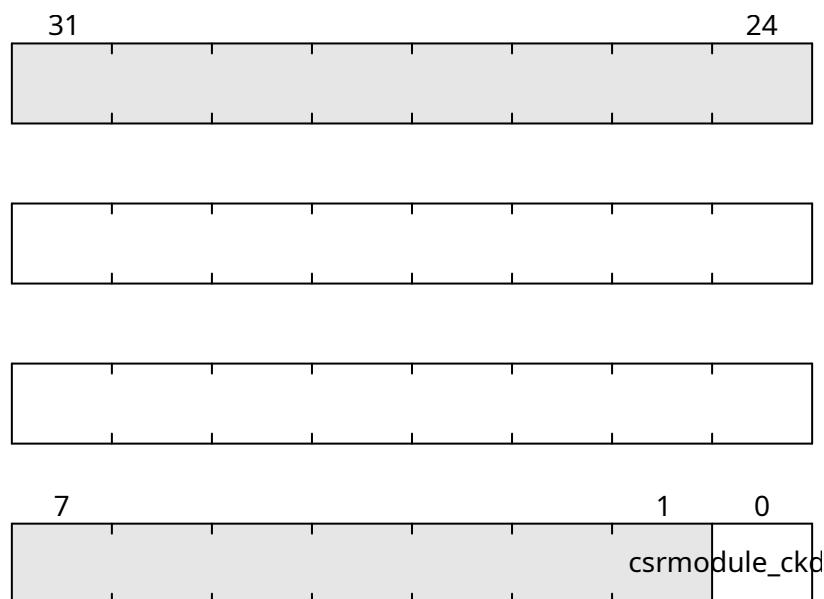


Fig. 24.11: DDRPHY_CSRMODULE_CKDLY_RST

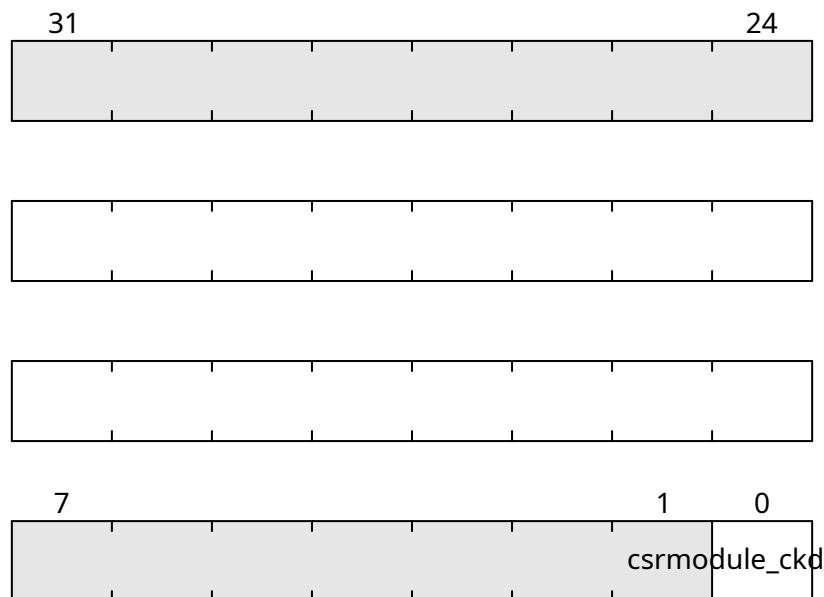


Fig. 24.12: `DDRPHY_CSRMODULE_CKDLY_INC`

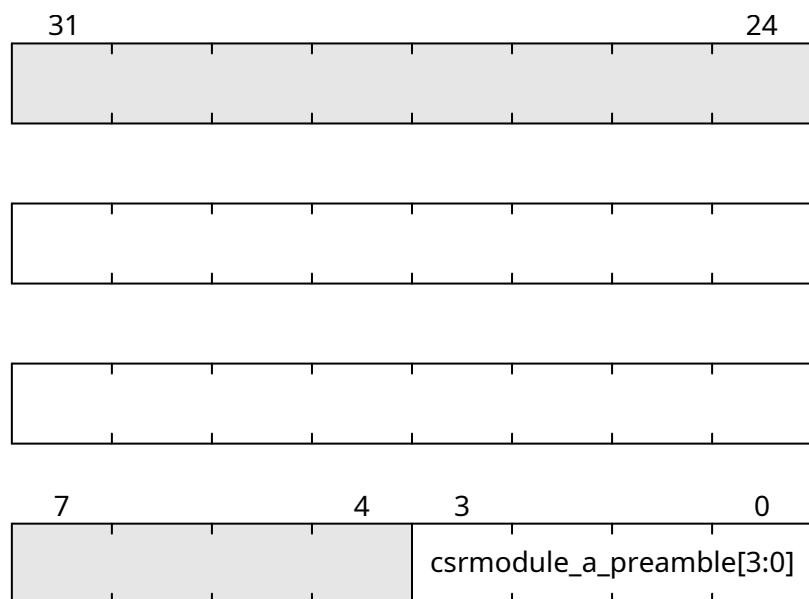


Fig. 24.13: `DDRPHY_CSRMODULE_A_PREAMBLE`

DDRPHY_CSRMODULE_A_WLEVEL_EN

Address: $0xf0000800 + 0x30 = 0xf0000830$

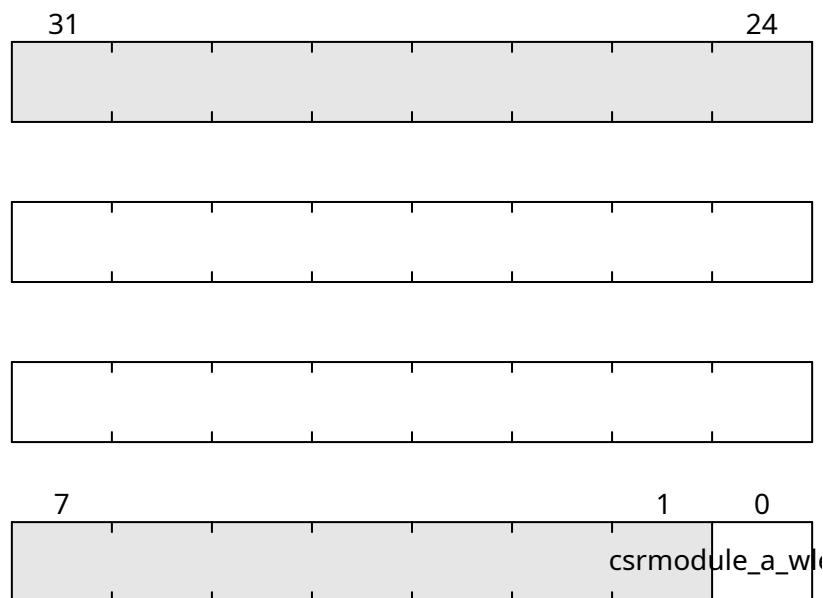


Fig. 24.14: DDRPHY_CSRMODULE_A_WLEVEL_EN

DDRPHY_CSRMODULE_A_PAR_ENABLE

Address: $0xf0000800 + 0x34 = 0xf0000834$

DDRPHY_CSRMODULE_A_PAR_VALUE

Address: $0xf0000800 + 0x38 = 0xf0000838$

DDRPHY_CSRMODULE_A_DISCARD_RD_FIFO

Address: $0xf0000800 + 0x3c = 0xf000083c$

DDRPHY_CSRMODULE_A_DLY_SEL

Address: $0xf0000800 + 0x40 = 0xf0000840$

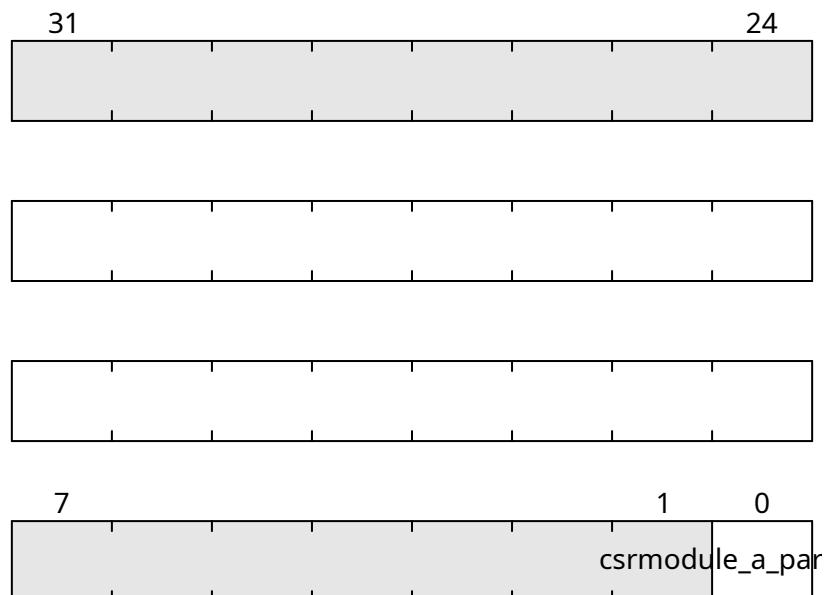


Fig. 24.15: `DDRPHY_CSRMODULE_A_PAR_ENABLE`

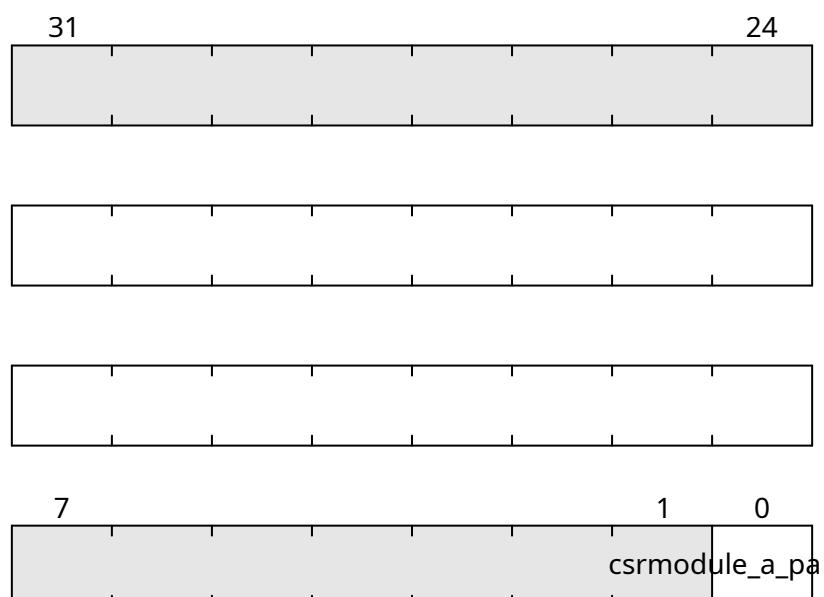


Fig. 24.16: `DDRPHY_CSRMODULE_A_PAR_VALUE`

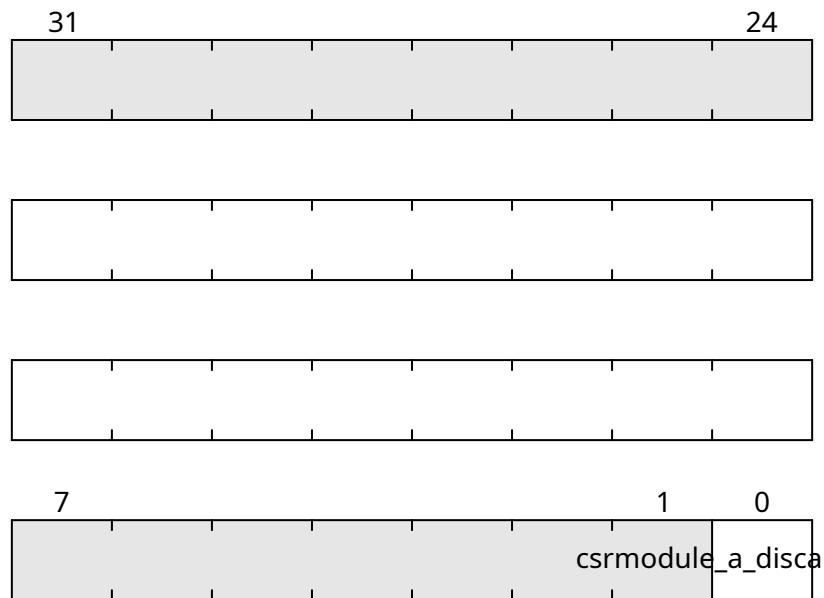


Fig. 24.17: DDRPHY_CSRMODULE_A_DISCARD_RD_FIFO

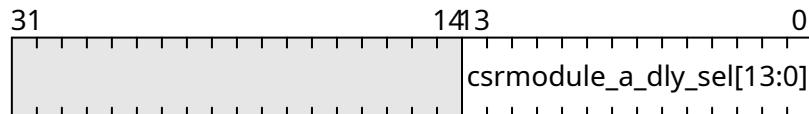


Fig. 24.18: DDRPHY_CSRMODULE_A_DLY_SEL

DDRPHY_CSRMODULE_A_DQ_DQS_RATIO

Address: $0xf0000800 + 0x44 = 0xf0000844$

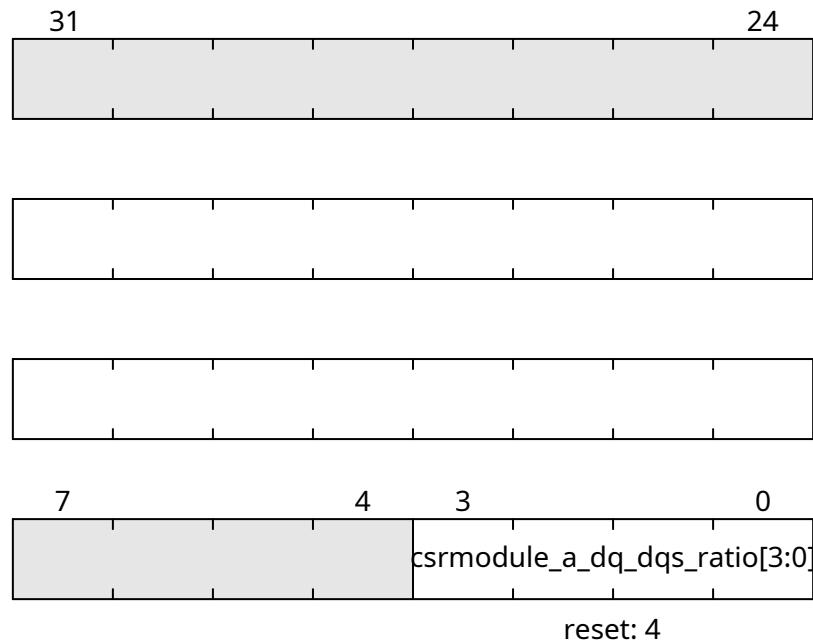


Fig. 24.19: DDRPHY_CSRMODULE_A_DQ_DQS_RATIO

DDRPHY_CSRMODULE_A_CK_RDLY_INC

Address: $0xf0000800 + 0x48 = 0xf0000848$

DDRPHY_CSRMODULE_A_CK_RDLY_RST

Address: $0xf0000800 + 0x4c = 0xf000084c$

DDRPHY_CSRMODULE_A_CK_RDDLY

Address: $0xf0000800 + 0x50 = 0xf0000850$

DDRPHY_CSRMODULE_A_CK_RDDLY_PREAMBLE

Address: $0xf0000800 + 0x54 = 0xf0000854$

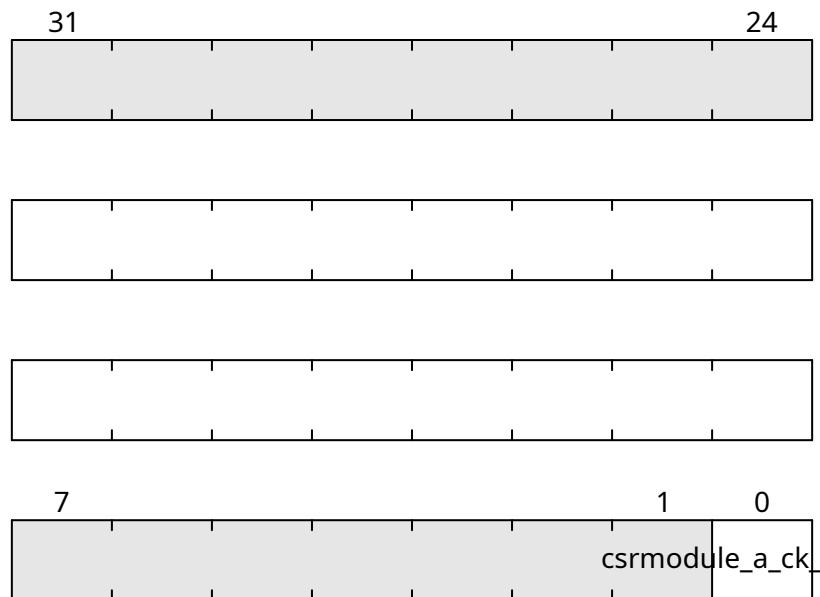


Fig. 24.20: `DDRPHY_CSRMODULE_A_CK_RDLY_INC`

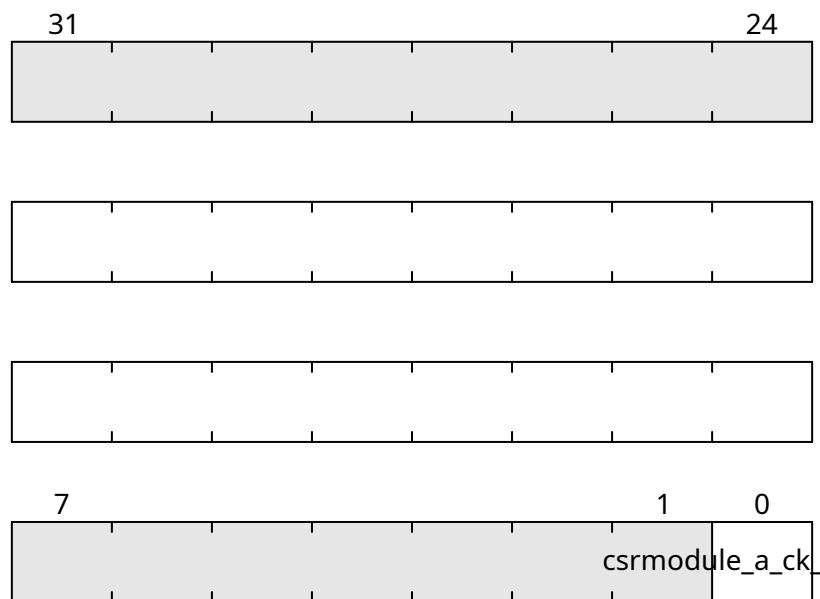


Fig. 24.21: `DDRPHY_CSRMODULE_A_CK_RDLY_RST`

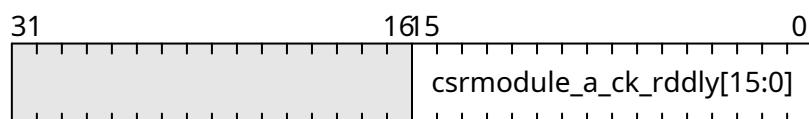


Fig. 24.22: `DDRPHY_CSRMODULE_A_CK_RDDLY`

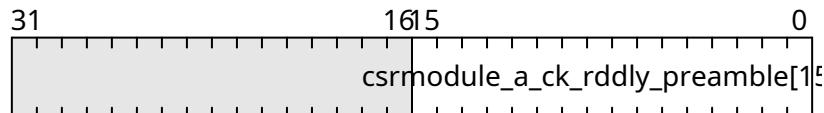


Fig. 24.23: DDRPHY_CSRMODULE_A_CK_RDDLY_PREAMBLE

DDRPHY_CSRMODULE_A_CK_WDLY_INC

Address: $0xf0000800 + 0x58 = 0xf0000858$

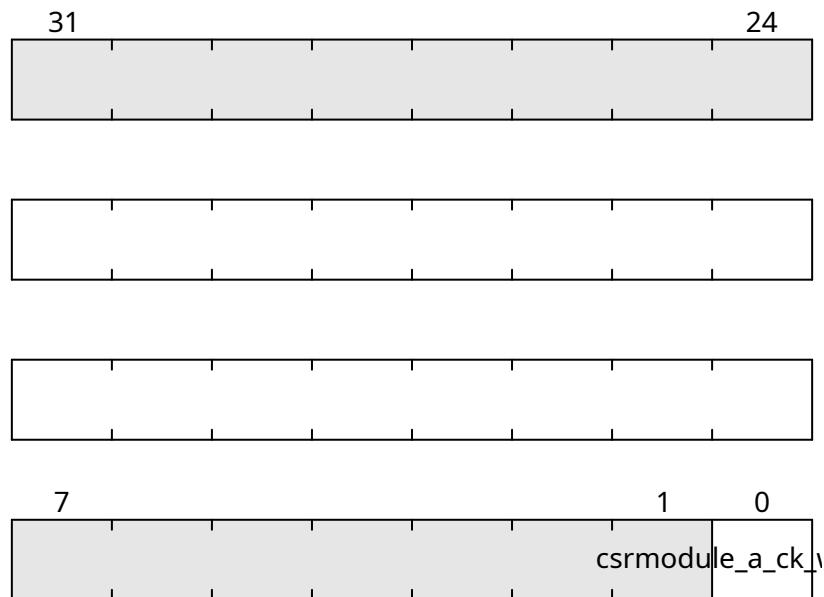


Fig. 24.24: DDRPHY_CSRMODULE_A_CK_WDLY_INC

DDRPHY_CSRMODULE_A_CK_WDLY_RST

Address: $0xf0000800 + 0x5c = 0xf000085c$

DDRPHY_CSRMODULE_A_CK_WDLY_DQS

Address: $0xf0000800 + 0x60 = 0xf0000860$

DDRPHY_CSRMODULE_A_CK_WDDLY_INC

Address: $0xf0000800 + 0x64 = 0xf0000864$

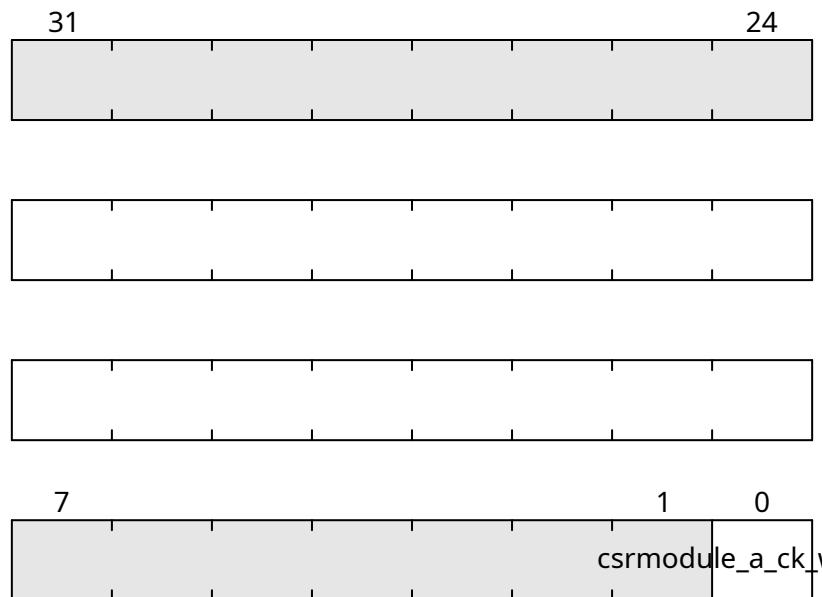


Fig. 24.25: DDRPHY_CSRMODULE_A_CK_WDLY_RST

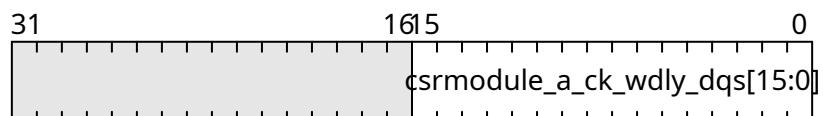


Fig. 24.26: DDRPHY_CSRMODULE_A_CK_WDLY_DQS

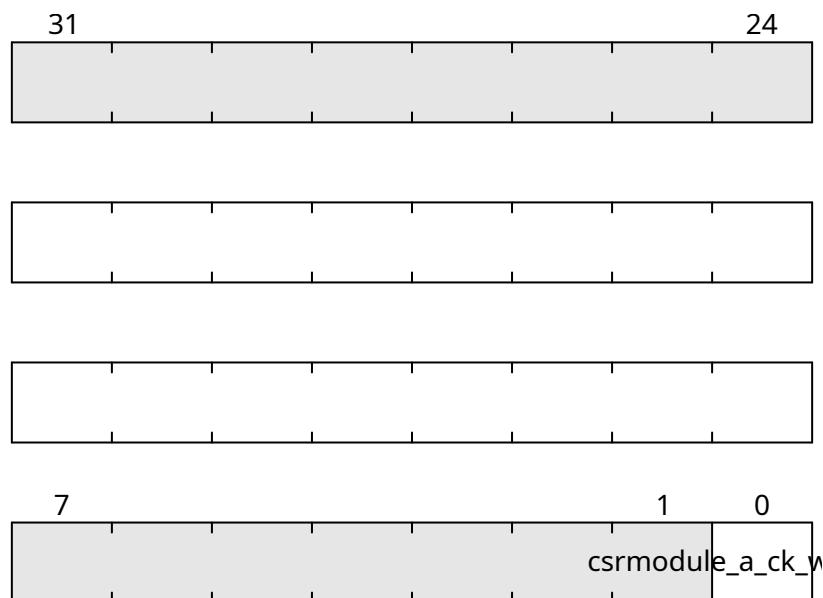


Fig. 24.27: DDRPHY_CSRMODULE_A_CK_WDDLY_INC

DDRPHY_CSRMODULE_A_CK_WDDLY_RST

Address: $0xf0000800 + 0x68 = 0xf0000868$

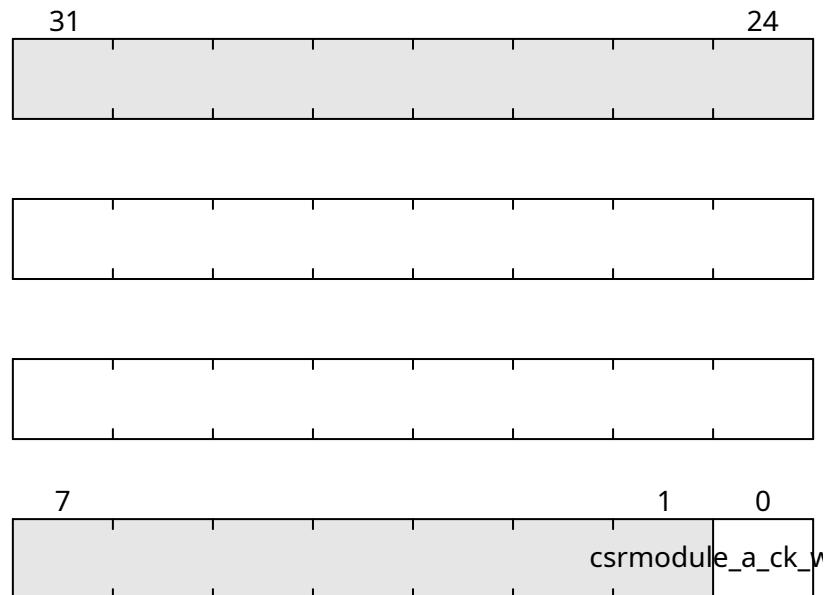


Fig. 24.28: DDRPHY_CSRMODULE_A_CK_WDDLY_RST

DDRPHY_CSRMODULE_A_CK_WDLY_DQ

Address: $0xf0000800 + 0x6c = 0xf000086c$

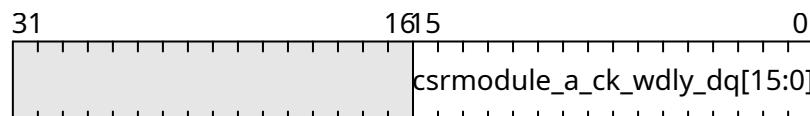


Fig. 24.29: DDRPHY_CSRMODULE_A_CK_WDLY_DQ

DDRPHY_CSRMODULE_A_DQ_DLY_SEL

Address: $0xf0000800 + 0x70 = 0xf0000870$

DDRPHY_CSRMODULE_A_CSDLY_RST

Address: $0xf0000800 + 0x74 = 0xf0000874$

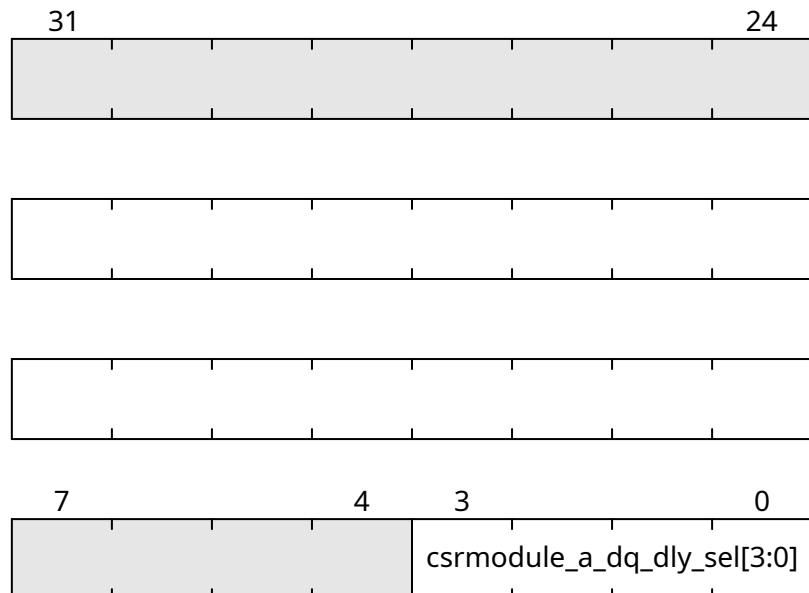


Fig. 24.30: DDRPHY_CSRMODULE_A_DQ_DLY_SEL

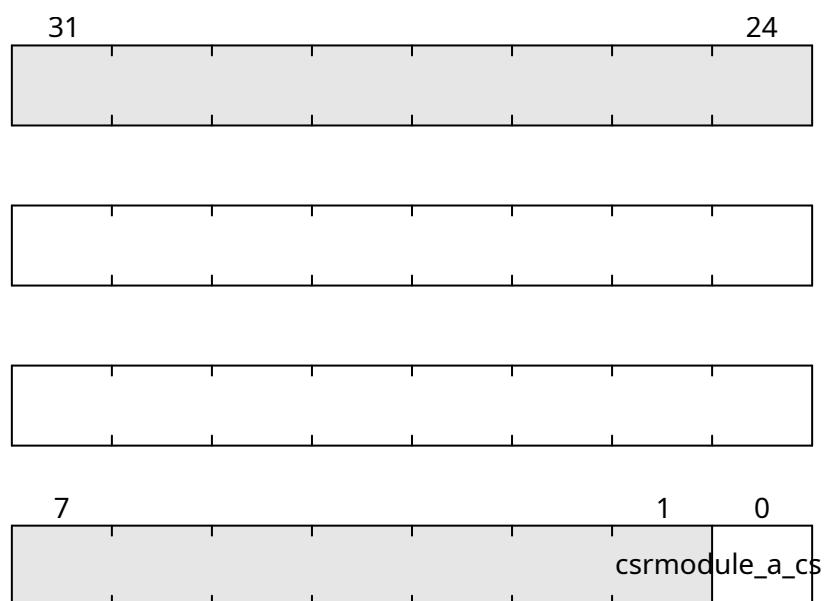


Fig. 24.31: DDRPHY_CSRMODULE_A_CSDLY_RST

DDRPHY_CSRMODULE_A_CSDLY_INC

Address: $0xf0000800 + 0x78 = 0xf0000878$

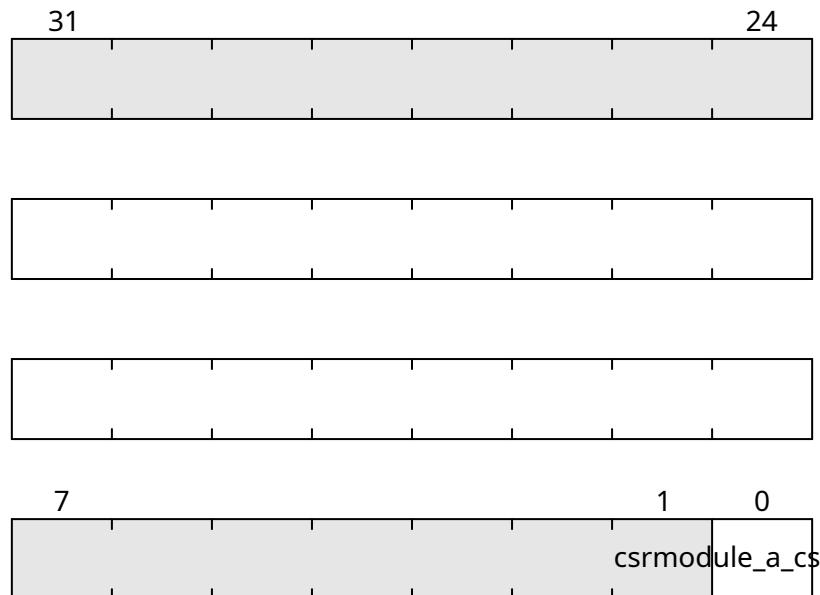


Fig. 24.32: DDRPHY_CSRMODULE_A_CSDLY_INC

DDRPHY_CSRMODULE_A_CADLY_RST

Address: $0xf0000800 + 0x7c = 0xf000087c$

DDRPHY_CSRMODULE_A_CADLY_INC

Address: $0xf0000800 + 0x80 = 0xf0000880$

DDRPHY_CSRMODULE_A_PARDLY_RST

Address: $0xf0000800 + 0x84 = 0xf0000884$

DDRPHY_CSRMODULE_A_PARDLY_INC

Address: $0xf0000800 + 0x88 = 0xf0000888$

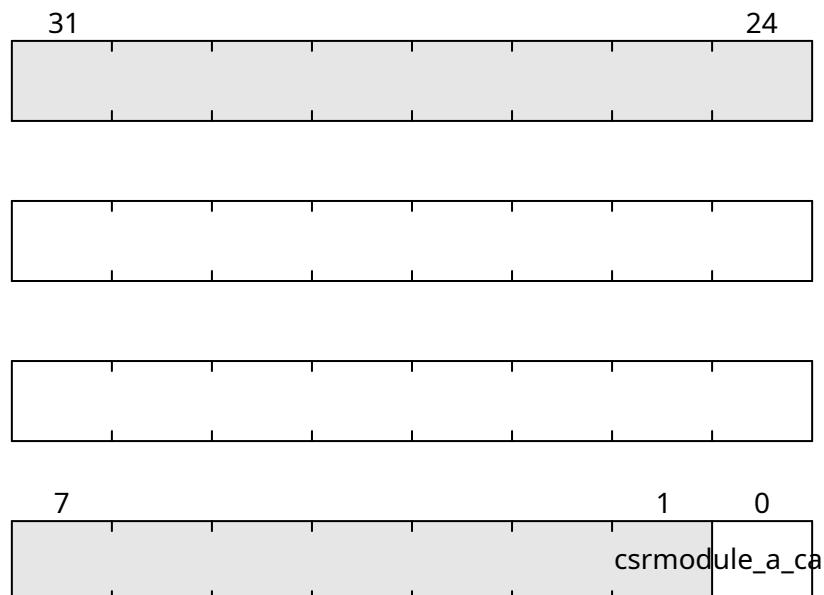


Fig. 24.33: DDRPHY_CSRMODULE_A_CADLY_RST

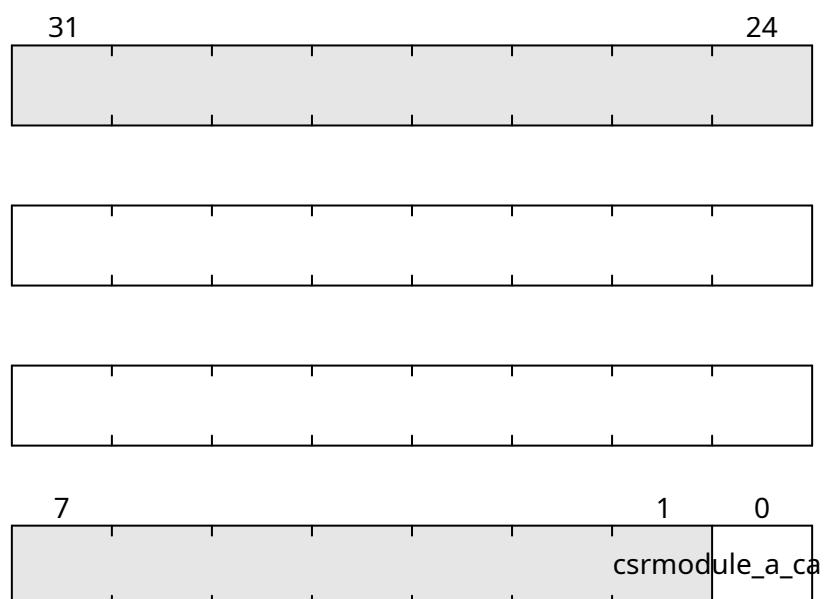


Fig. 24.34: DDRPHY_CSRMODULE_A_CADLY_INC

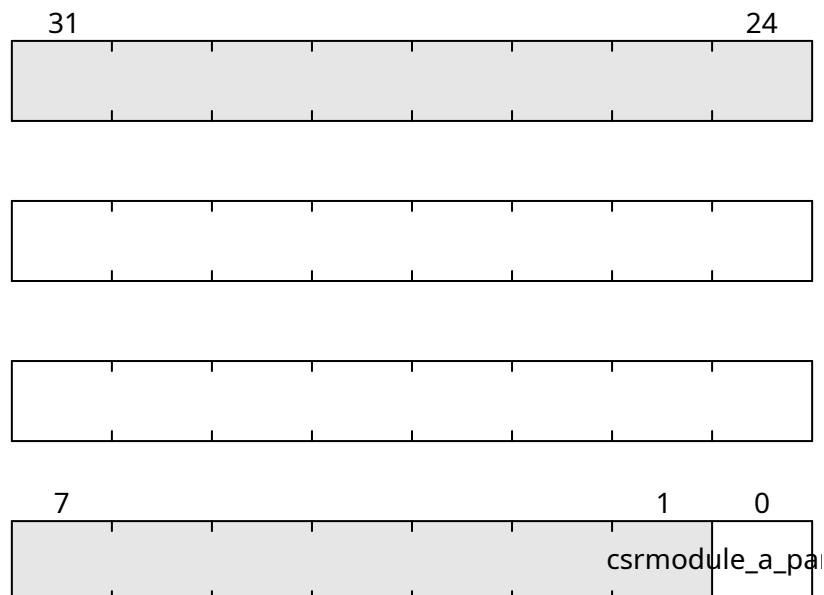


Fig. 24.35: DDRPHY_CSRMODULE_A_PARDLY_RST

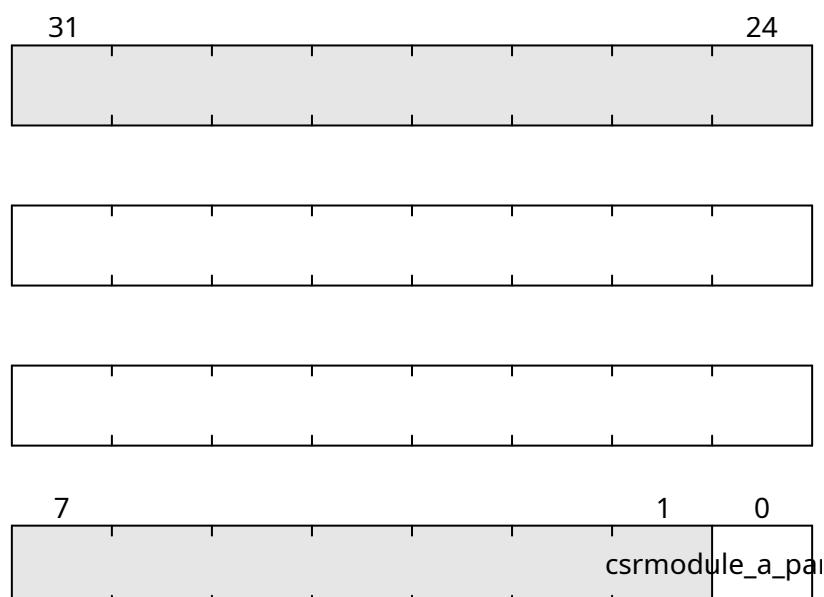


Fig. 24.36: DDRPHY_CSRMODULE_A_PARDLY_INC

DDRPHY_CSRMODULE_A_CSDLY

Address: $0xf0000800 + 0x8c = 0xf000088c$

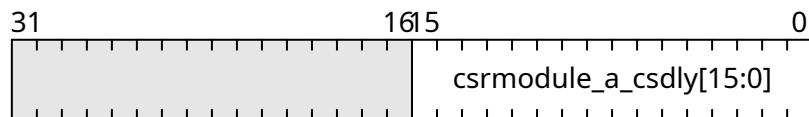


Fig. 24.37: DDRPHY_CSRMODULE_A_CSDLY

DDRPHY_CSRMODULE_A_CADLY

Address: $0xf0000800 + 0x90 = 0xf0000890$

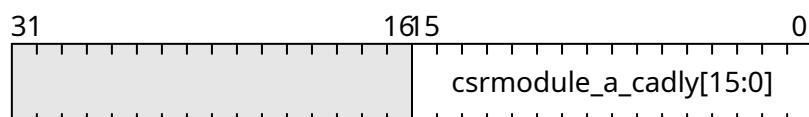


Fig. 24.38: DDRPHY_CSRMODULE_A_CADLY

DDRPHY_CSRMODULE_A_RDLY_DQ_RST

Address: $0xf0000800 + 0x94 = 0xf0000894$

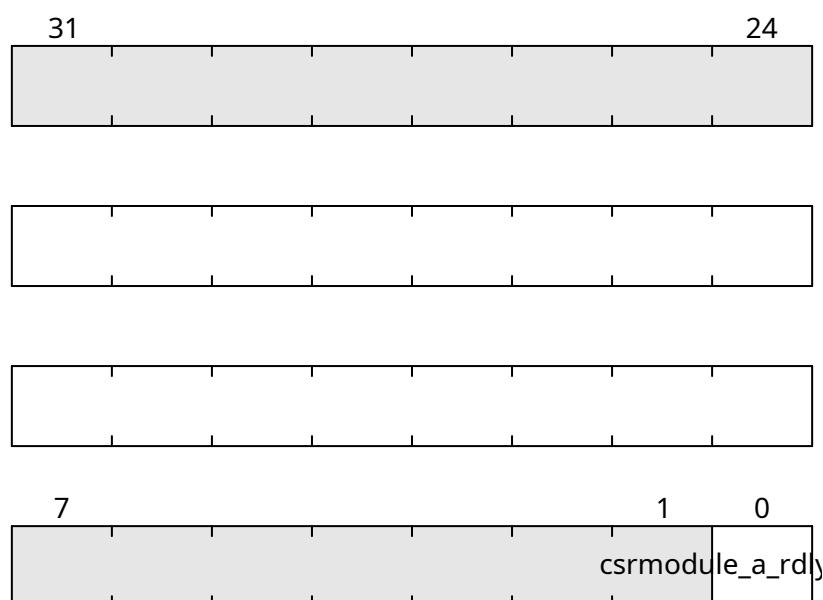


Fig. 24.39: DDRPHY_CSRMODULE_A_RDLY_DQ_RST

DDRPHY_CSRMODULE_A_RDLY_DQ_INC

Address: $0xf0000800 + 0x98 = 0xf0000898$

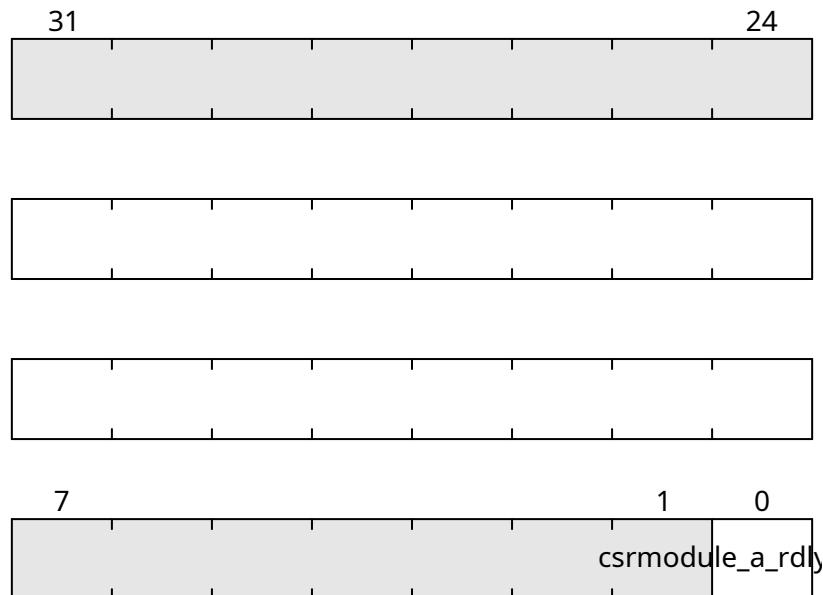


Fig. 24.40: DDRPHY_CSRMODULE_A_RDLY_DQ_INC

DDRPHY_CSRMODULE_A_RDLY_DQS_RST

Address: $0xf0000800 + 0x9c = 0xf000089c$

DDRPHY_CSRMODULE_A_RDLY_DQS_INC

Address: $0xf0000800 + 0xa0 = 0xf00008a0$

DDRPHY_CSRMODULE_A_RDLY_DQS

Address: $0xf0000800 + 0xa4 = 0xf00008a4$

DDRPHY_CSRMODULE_A_RDLY_DQ

Address: $0xf0000800 + 0xa8 = 0xf00008a8$

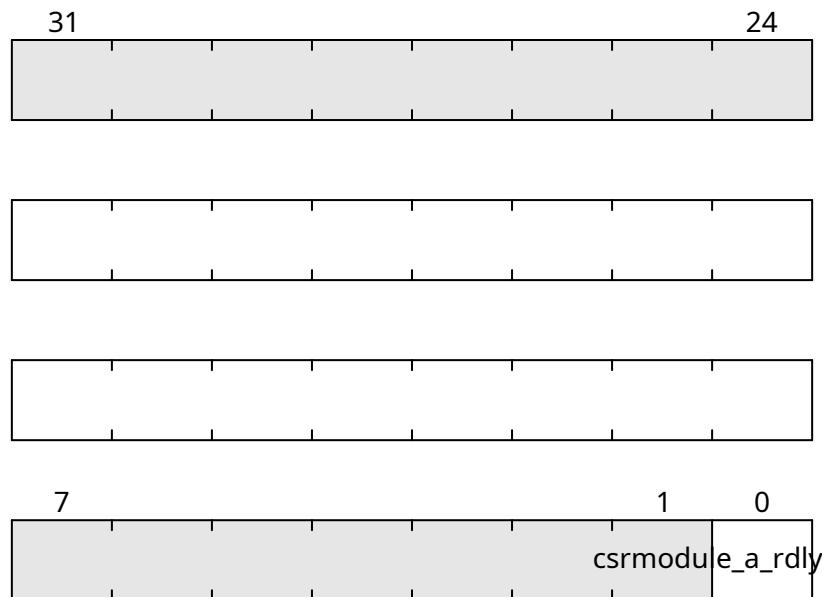


Fig. 24.41: `DDRPHY_CSRMODULE_A_RDLY_DQS_RST`

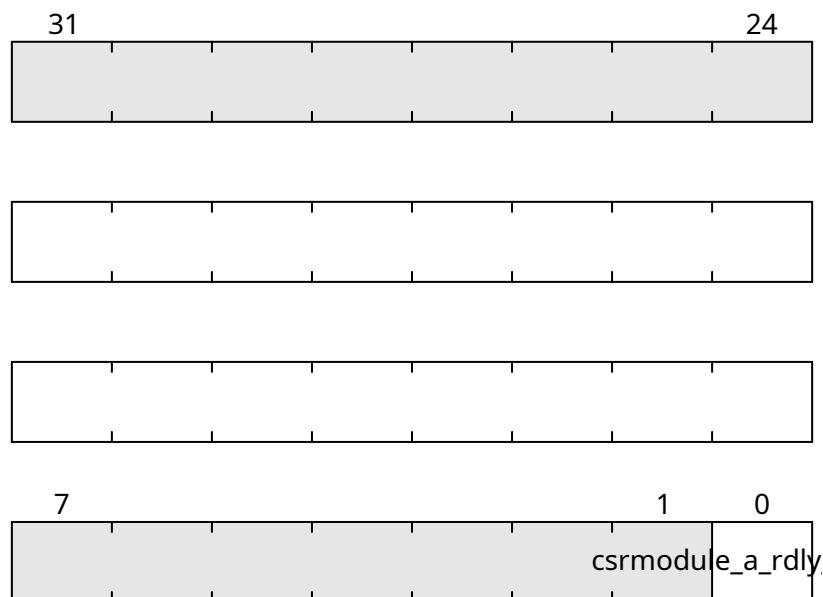


Fig. 24.42: `DDRPHY_CSRMODULE_A_RDLY_DQS_INC`

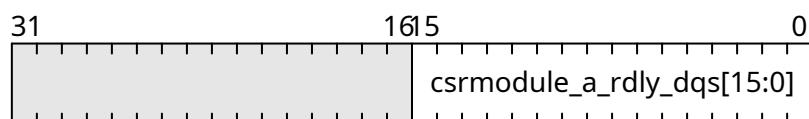


Fig. 24.43: `DDRPHY_CSRMODULE_A_RDLY_DQS`

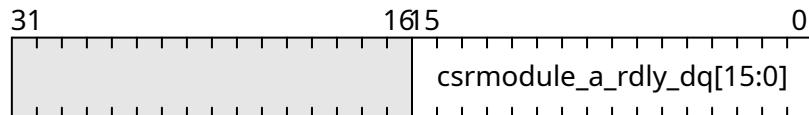


Fig. 24.44: DDRPHY_CSRMODULE_A_RDLY_DQ

DDRPHY_CSRMODULE_A_WDLY_DQ_RST

Address: $0xf0000800 + 0xac = 0xf00008ac$

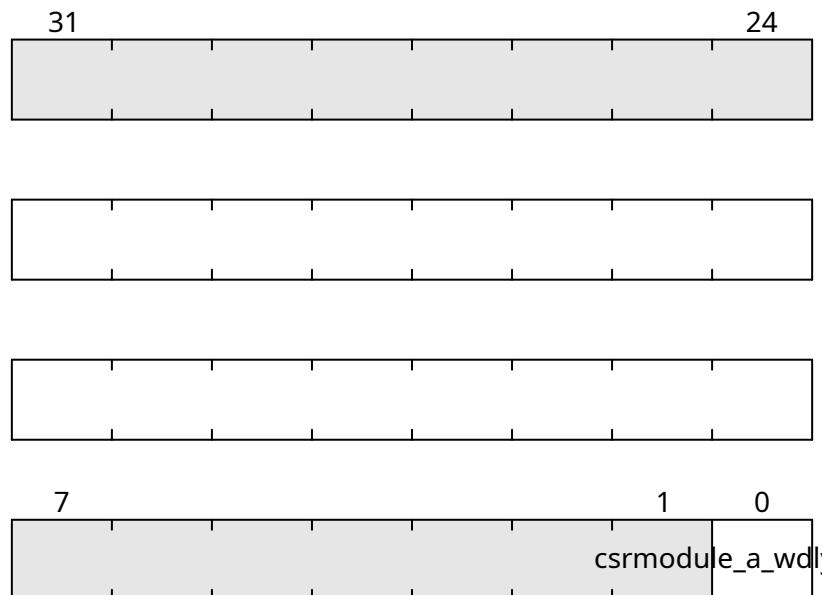


Fig. 24.45: DDRPHY_CSRMODULE_A_WDLY_DQ_RST

DDRPHY_CSRMODULE_A_WDLY_DQ_INC

Address: $0xf0000800 + 0xb0 = 0xf00008b0$

DDRPHY_CSRMODULE_A_WDLY_DM_RST

Address: $0xf0000800 + 0xb4 = 0xf00008b4$

DDRPHY_CSRMODULE_A_WDLY_DM_INC

Address: $0xf0000800 + 0xb8 = 0xf00008b8$

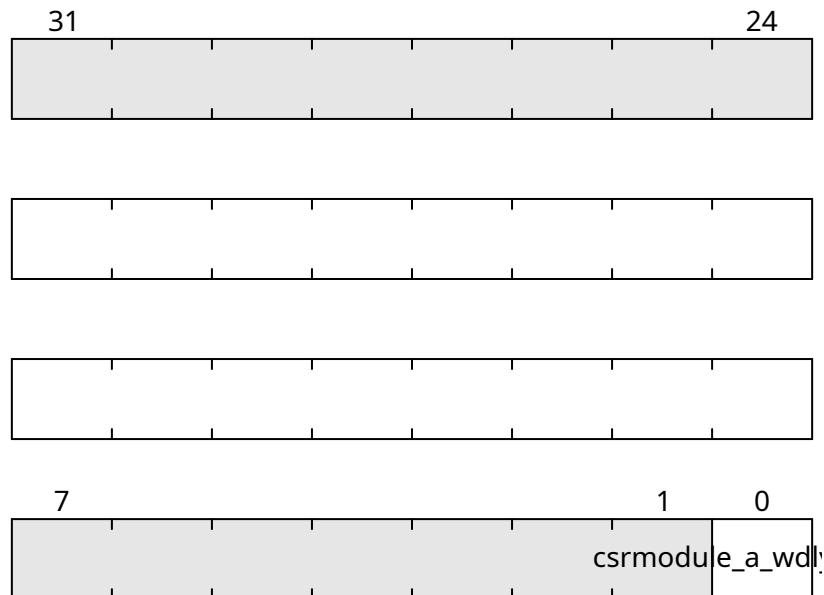


Fig. 24.46: `DDRPHY_CSRMODULE_A_WDLY_DQ_INC`

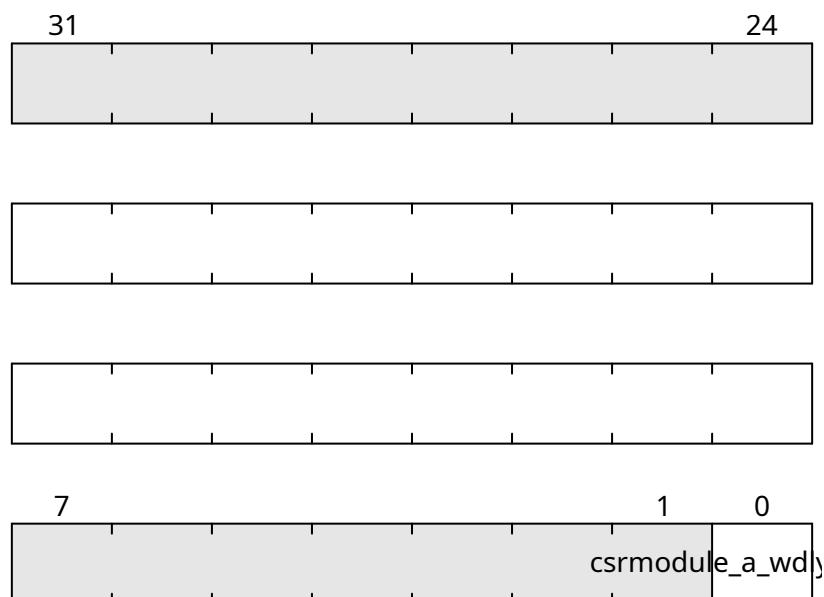


Fig. 24.47: `DDRPHY_CSRMODULE_A_WDLY_DM_RST`

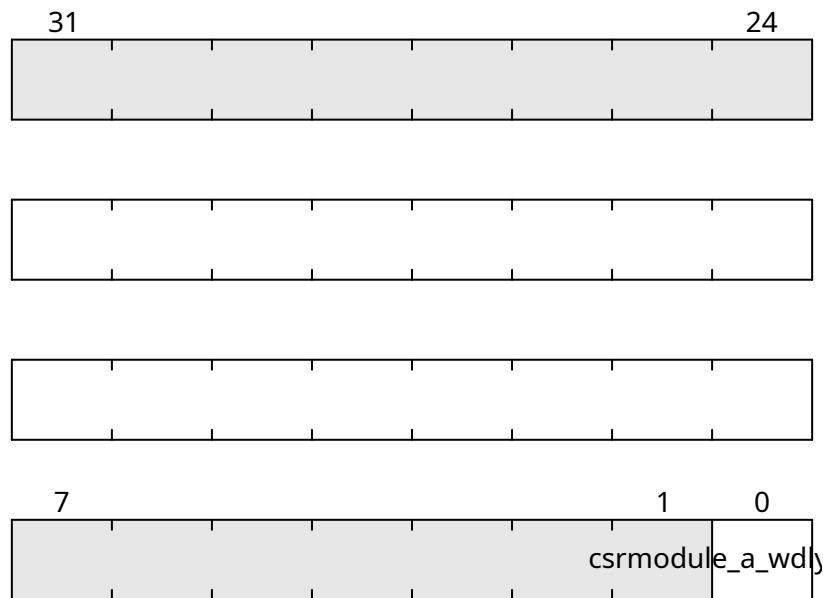


Fig. 24.48: DDRPHY_CSRMODULE_A_WDLY_DM_INC

DDRPHY_CSRMODULE_A_WDLY_DQS_RST

Address: $0xf0000800 + 0xbc = 0xf00008bc$

DDRPHY_CSRMODULE_A_WDLY_DQS_INC

Address: $0xf0000800 + 0xc0 = 0xf00008c0$

DDRPHY_CSRMODULE_A_WDLY_DQS

Address: $0xf0000800 + 0xc4 = 0xf00008c4$

DDRPHY_CSRMODULE_A_WDLY_DQ

Address: $0xf0000800 + 0xc8 = 0xf00008c8$

DDRPHY_CSRMODULE_A_WDLY_DM

Address: $0xf0000800 + 0xcc = 0xf00008cc$

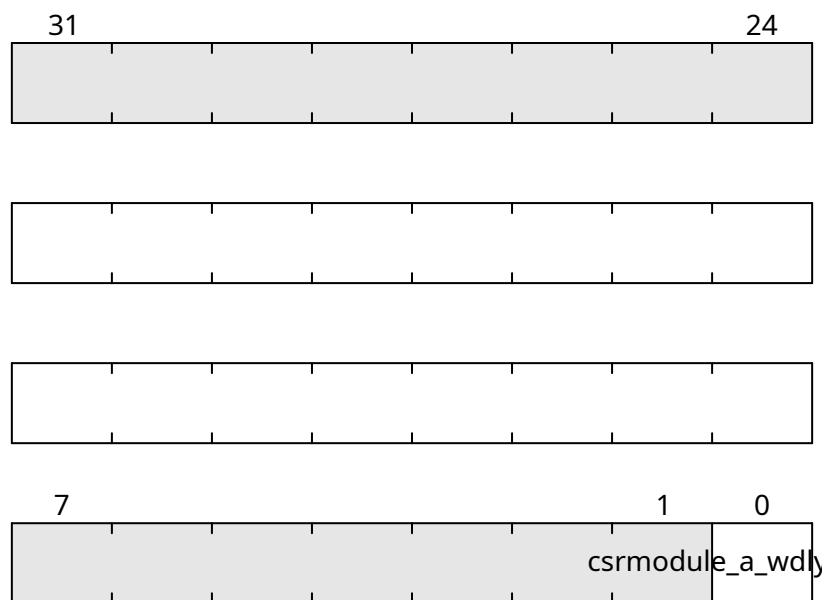


Fig. 24.49: `DDRPHY_CSRMODULE_A_WDLY_DQS_RST`

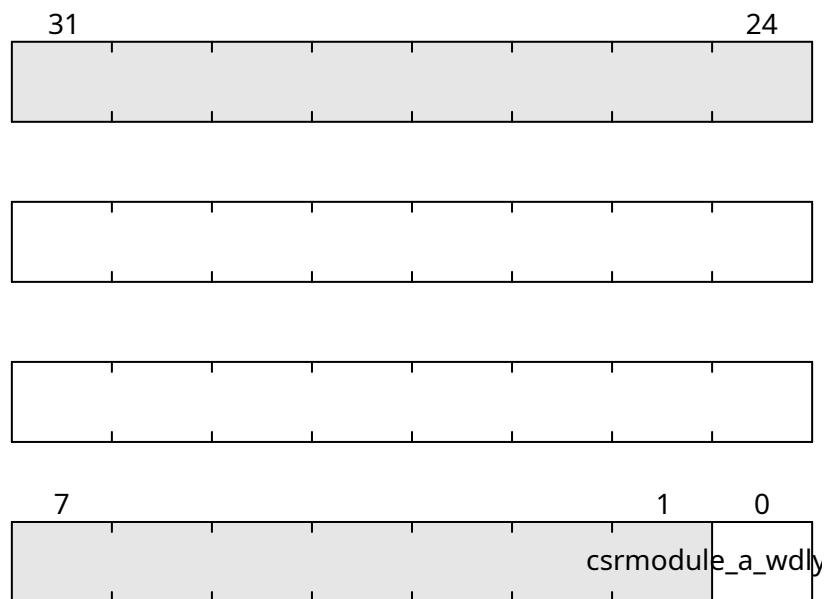


Fig. 24.50: `DDRPHY_CSRMODULE_A_WDLY_DQS_INC`

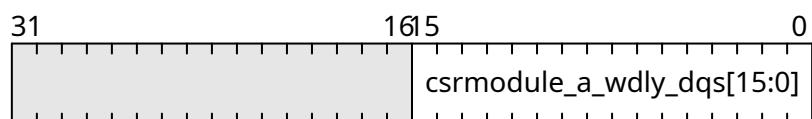


Fig. 24.51: `DDRPHY_CSRMODULE_A_WDLY_DQS`

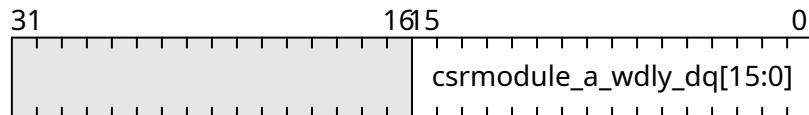


Fig. 24.52: DDRPHY_CSRMODULE_A_WDLY_DQ

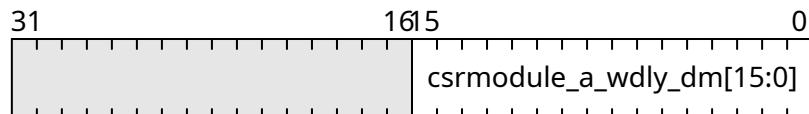


Fig. 24.53: DDRPHY_CSRMODULE_A_WDLY_DM

DDRPHY_CSRMODULE_B_PREAMBLE

Address: 0xf0000800 + 0xd0 = 0xf00008d0

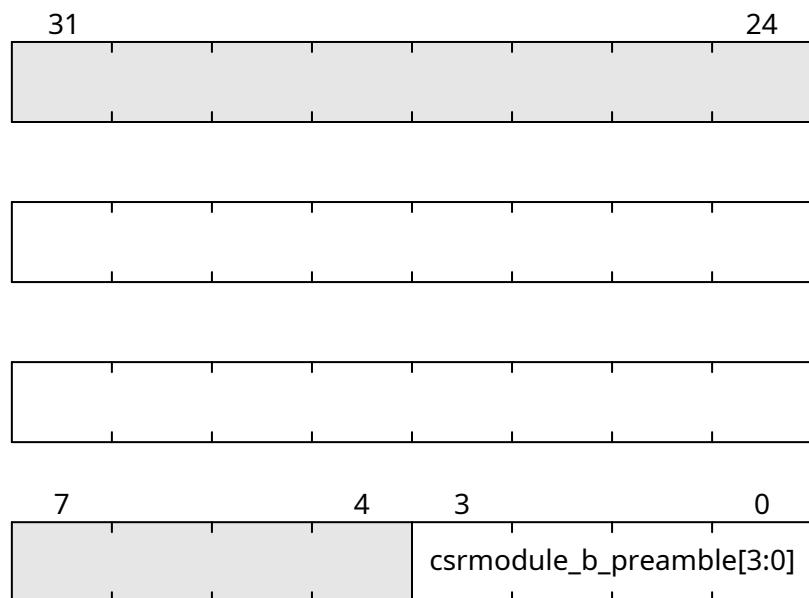


Fig. 24.54: DDRPHY_CSRMODULE_B_PREAMBLE

DDRPHY_CSRMODULE_B_WLEVEL_EN

Address: 0xf0000800 + 0xd4 = 0xf00008d4

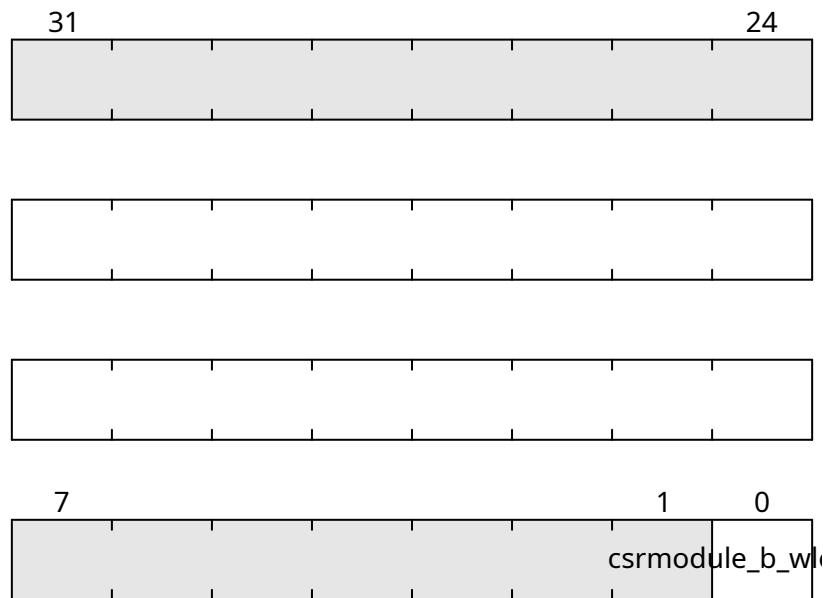


Fig. 24.55: DDRPHY_CSRMODULE_B_WLEVEL_EN

DDRPHY_CSRMODULE_B_PAR_ENABLE

Address: $0xf0000800 + 0xd8 = 0xf00008d8$

DDRPHY_CSRMODULE_B_PAR_VALUE

Address: $0xf0000800 + 0xdc = 0xf00008dc$

DDRPHY_CSRMODULE_B_DISCARD_RD_FIFO

Address: $0xf0000800 + 0xe0 = 0xf00008e0$

DDRPHY_CSRMODULE_B_DLY_SEL

Address: $0xf0000800 + 0xe4 = 0xf00008e4$

DDRPHY_CSRMODULE_B_DQ_DQS_RATIO

Address: $0xf0000800 + 0xe8 = 0xf00008e8$

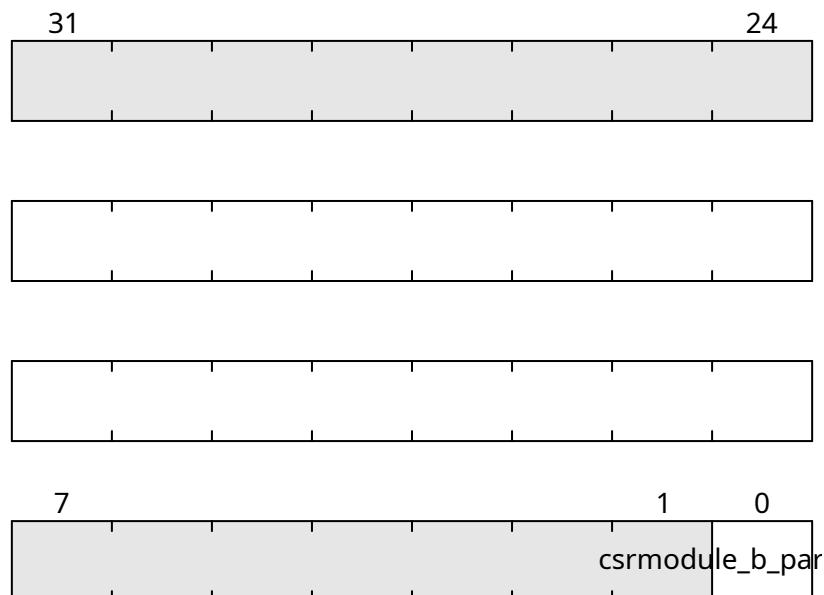


Fig. 24.56: `DDRPHY_CSRMODULE_B_PAR_ENABLE`

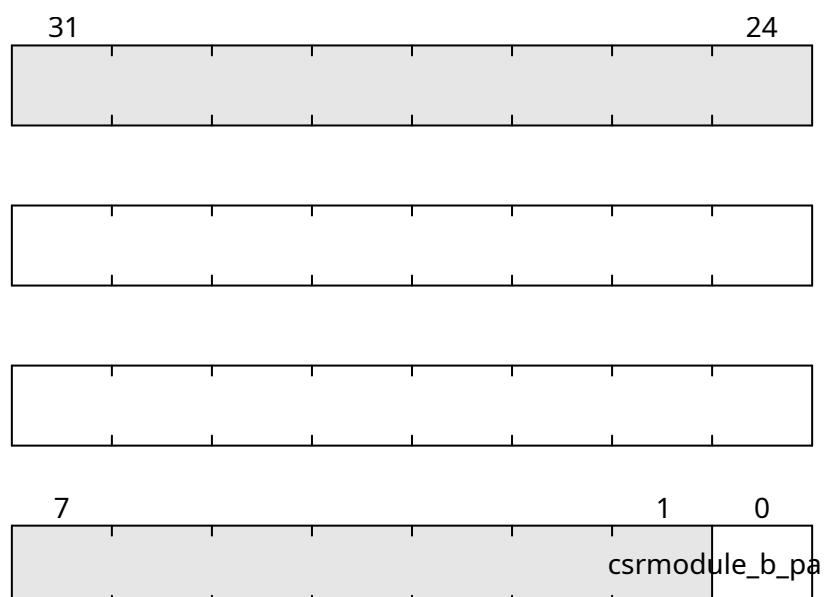


Fig. 24.57: `DDRPHY_CSRMODULE_B_PAR_VALUE`

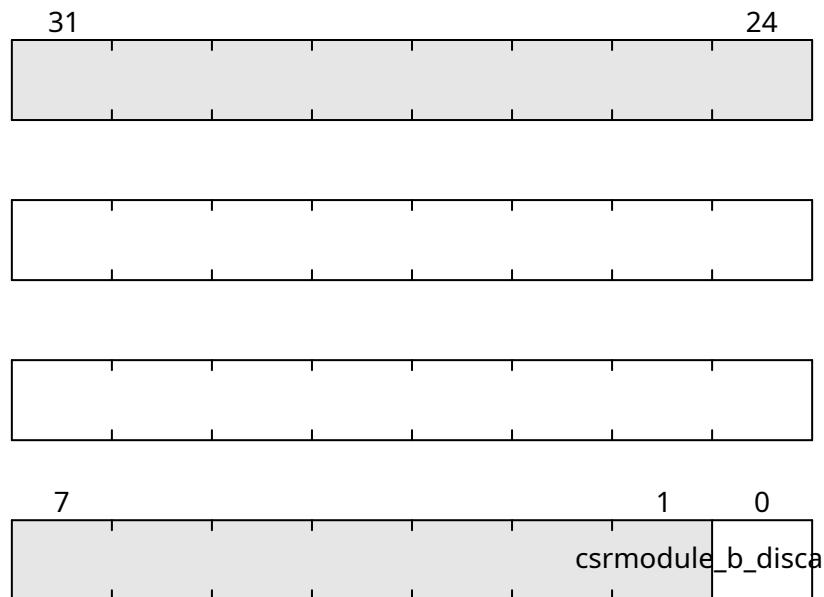


Fig. 24.58: DDRPHY_CSRMODULE_B_DISCARD_RD_FIFO

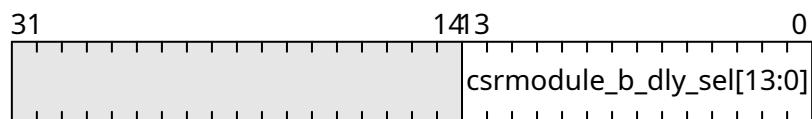


Fig. 24.59: DDRPHY_CSRMODULE_B_DLY_SEL

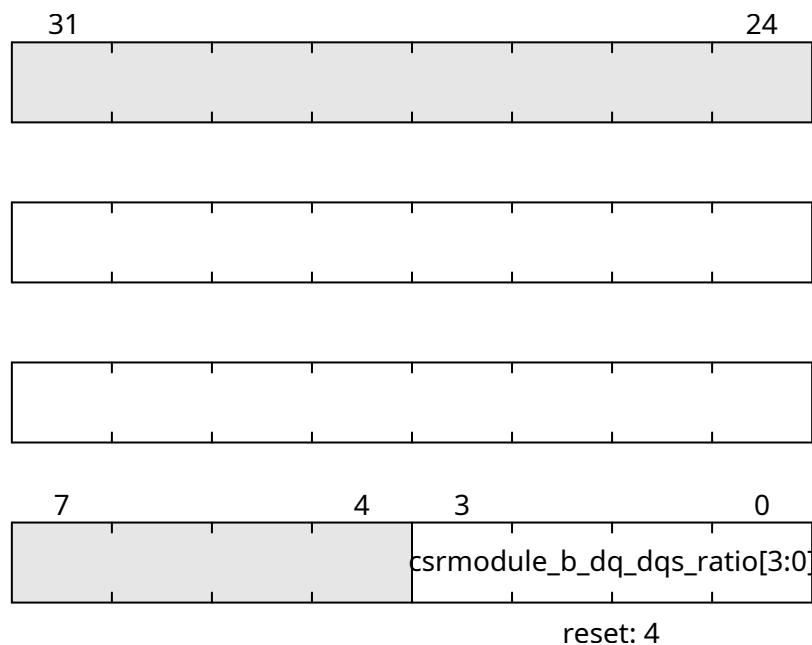


Fig. 24.60: DDRPHY_CSRMODULE_B_DQ_DQS_RATIO

DDRPHY_CSRMODULE_B_CK_RDLY_INC

Address: 0xf0000800 + 0xec = 0xf00008ec

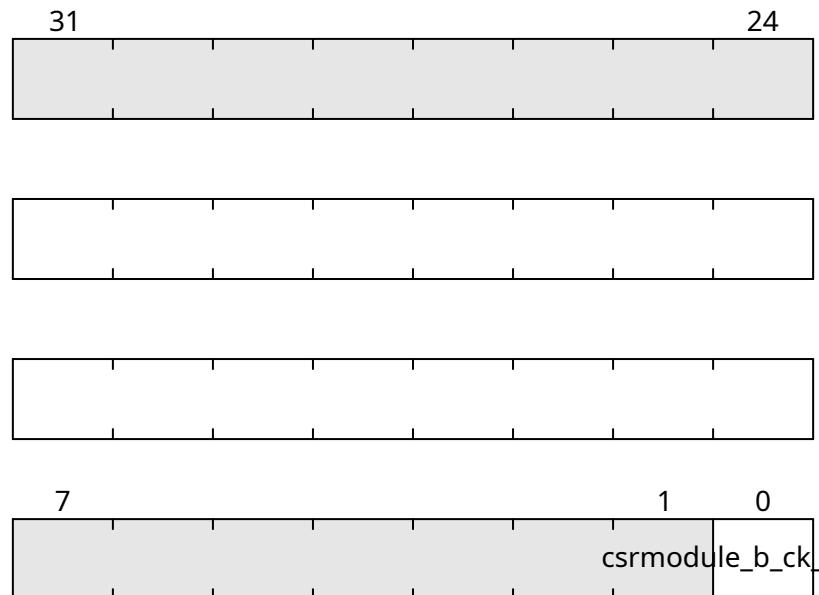


Fig. 24.61: DDRPHY_CSRMODULE_B_CK_RDLY_INC

DDRPHY_CSRMODULE_B_CK_RDLY_RST

Address: 0xf0000800 + 0xf0 = 0xf00008f0

DDRPHY_CSRMODULE_B_CK_RDDLY

Address: 0xf0000800 + 0xf4 = 0xf00008f4

DDRPHY_CSRMODULE_B_CK_RDDLY_PREAMBLE

Address: 0xf0000800 + 0xf8 = 0xf00008f8

DDRPHY_CSRMODULE_B_CK_WDLY_INC

Address: 0xf0000800 + 0xfc = 0xf00008fc

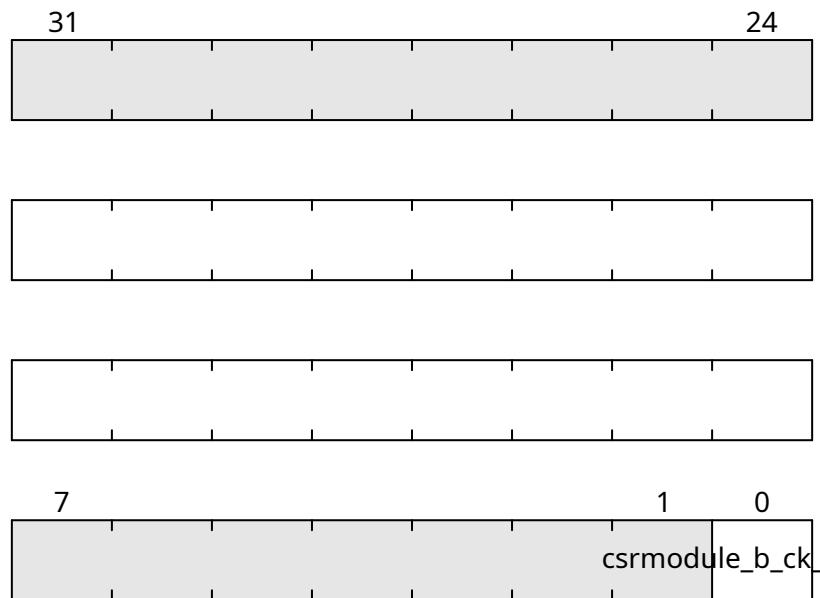


Fig. 24.62: `DDRPHY_CSRMODULE_B_CK_RDLY_RST`

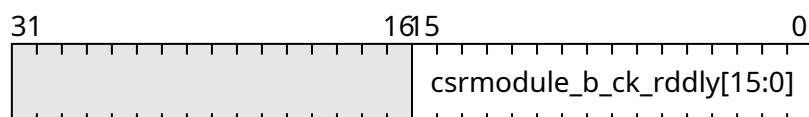


Fig. 24.63: `DDRPHY_CSRMODULE_B_CK_RDDLY`

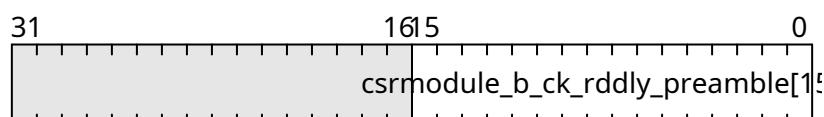


Fig. 24.64: `DDRPHY_CSRMODULE_B_CK_RDDLY_PREAMBLE`

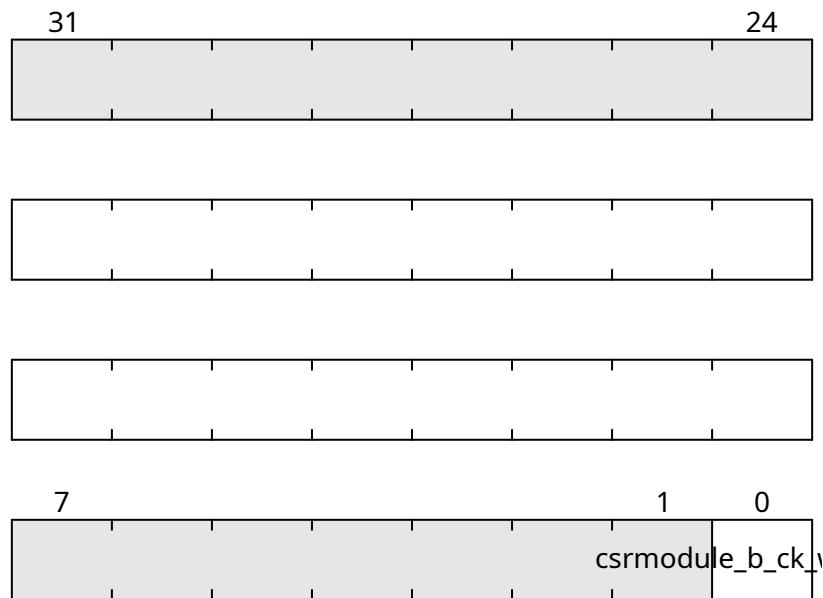


Fig. 24.65: DDRPHY_CSRMODULE_B_CK_WDLY_INC

DDRPHY_CSRMODULE_B_CK_WDLY_RST

Address: $0xf0000800 + 0x100 = 0xf0000900$

DDRPHY_CSRMODULE_B_CK_WDLY_DQS

Address: $0xf0000800 + 0x104 = 0xf0000904$

DDRPHY_CSRMODULE_B_CK_WDDLY_INC

Address: $0xf0000800 + 0x108 = 0xf0000908$

DDRPHY_CSRMODULE_B_CK_WDDLY_RST

Address: $0xf0000800 + 0x10c = 0xf000090c$

DDRPHY_CSRMODULE_B_CK_WDLY_DQ

Address: $0xf0000800 + 0x110 = 0xf0000910$

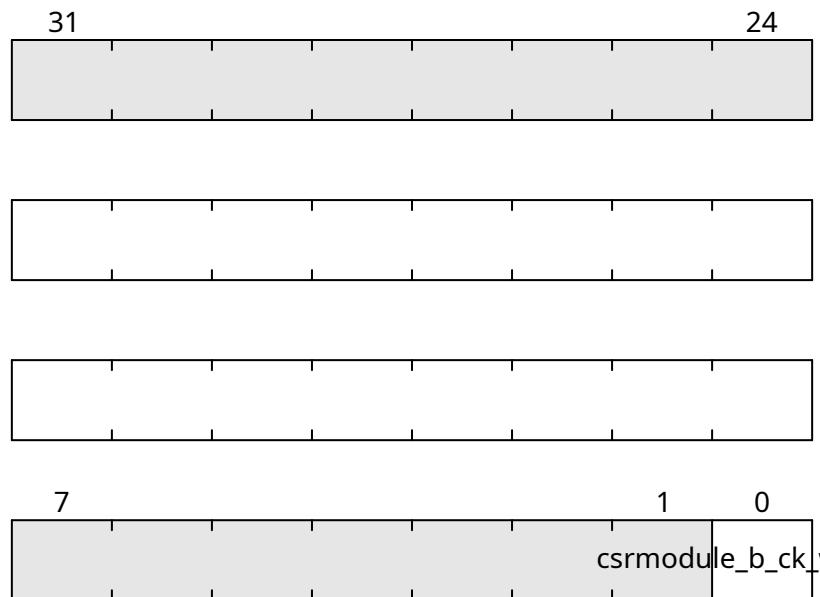


Fig. 24.66: `DDRPHY_CSRMODULE_B_CK_WDLY_RST`

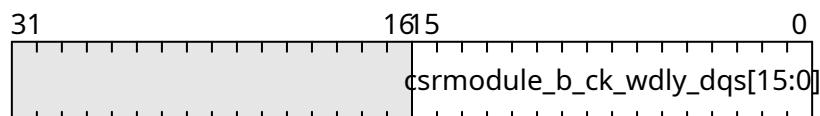


Fig. 24.67: `DDRPHY_CSRMODULE_B_CK_WDLY_DQS`

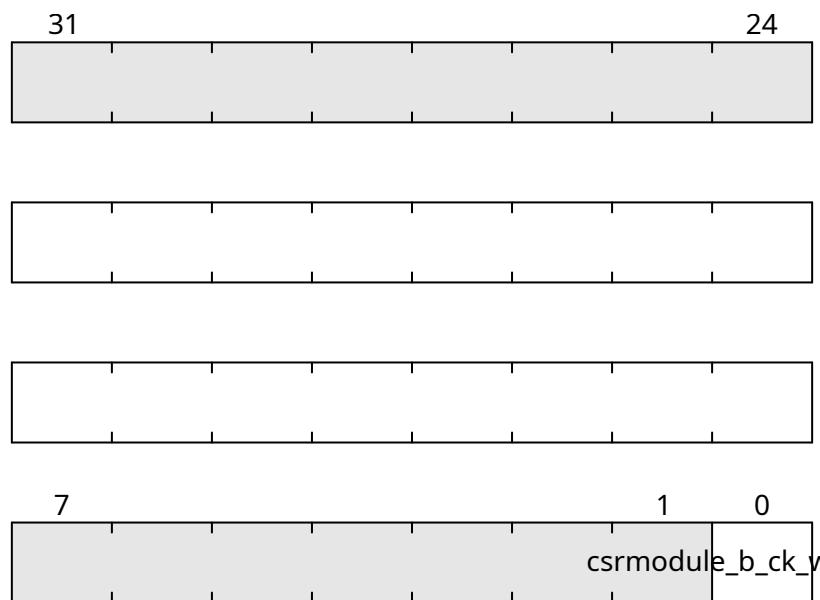


Fig. 24.68: `DDRPHY_CSRMODULE_B_CK_WDDLY_INC`

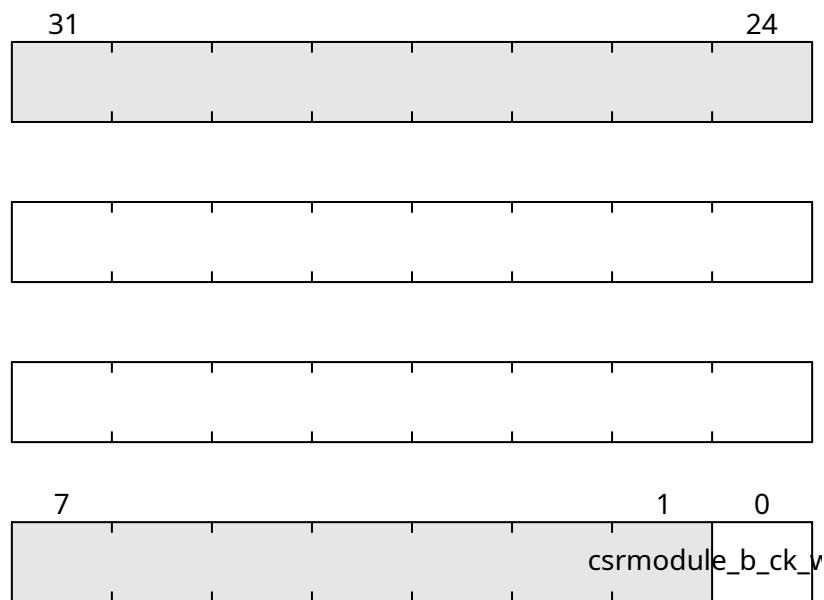


Fig. 24.69: DDRPHY_CSRMODULE_B_CK_WDDLY_RST

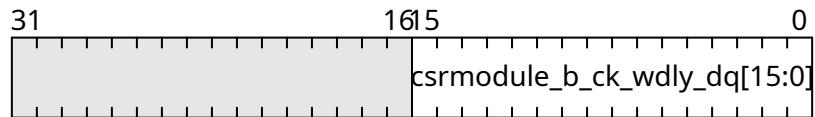


Fig. 24.70: DDRPHY_CSRMODULE_B_CK_WDLY_DQ

DDRPHY_CSRMODULE_B_DQ_DLY_SEL

Address: 0xf0000800 + 0x114 = 0xf0000914

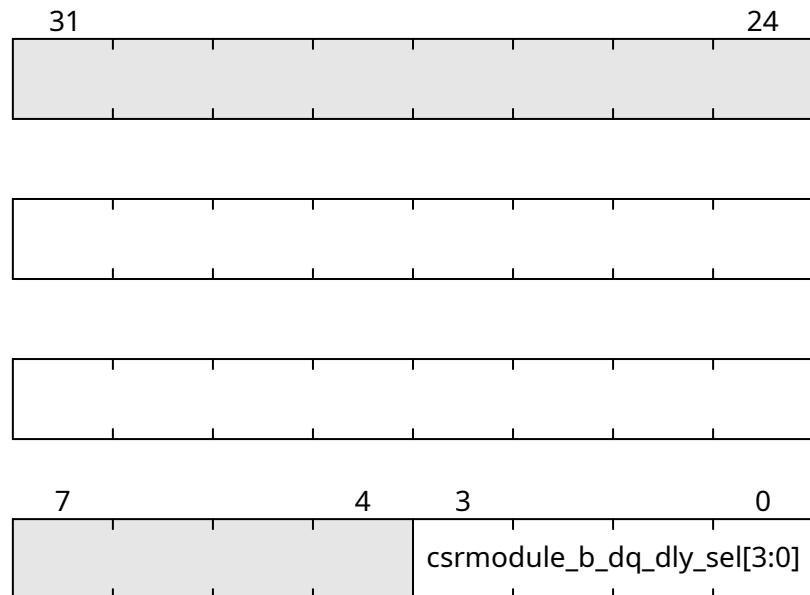


Fig. 24.71: DDRPHY_CSRMODULE_B_DQ_DLY_SEL

DDRPHY_CSRMODULE_B_CSDLY_RST

Address: 0xf0000800 + 0x118 = 0xf0000918

DDRPHY_CSRMODULE_B_CSDLY_INC

Address: 0xf0000800 + 0x11c = 0xf000091c

DDRPHY_CSRMODULE_B_CADLY_RST

Address: 0xf0000800 + 0x120 = 0xf0000920

DDRPHY_CSRMODULE_B_CADLY_INC

Address: 0xf0000800 + 0x124 = 0xf0000924

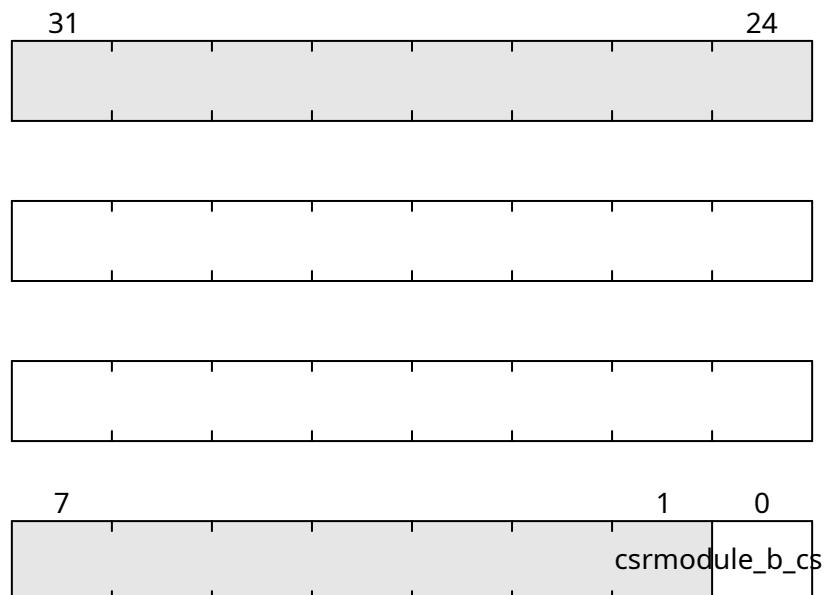


Fig. 24.72: DDRPHY_CSRMODULE_B_CSDLY_RST

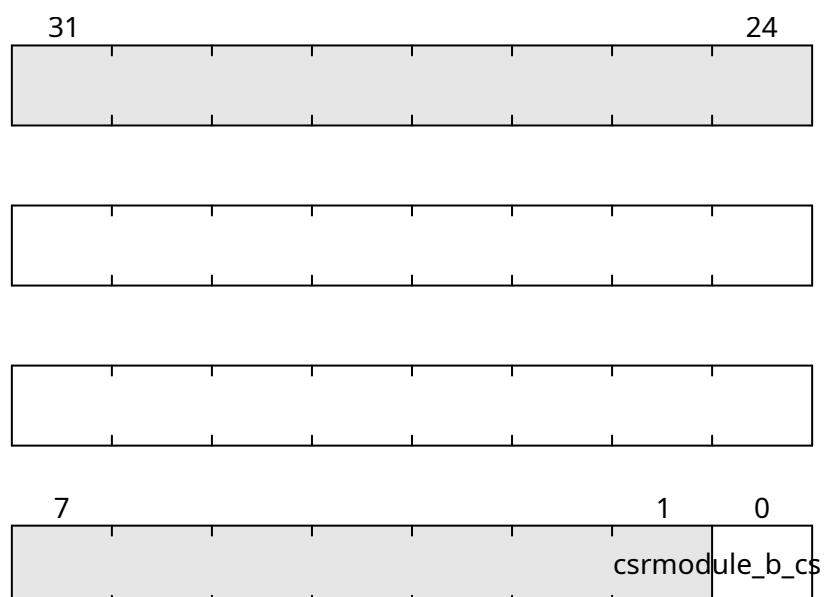


Fig. 24.73: DDRPHY_CSRMODULE_B_CSDLY_INC

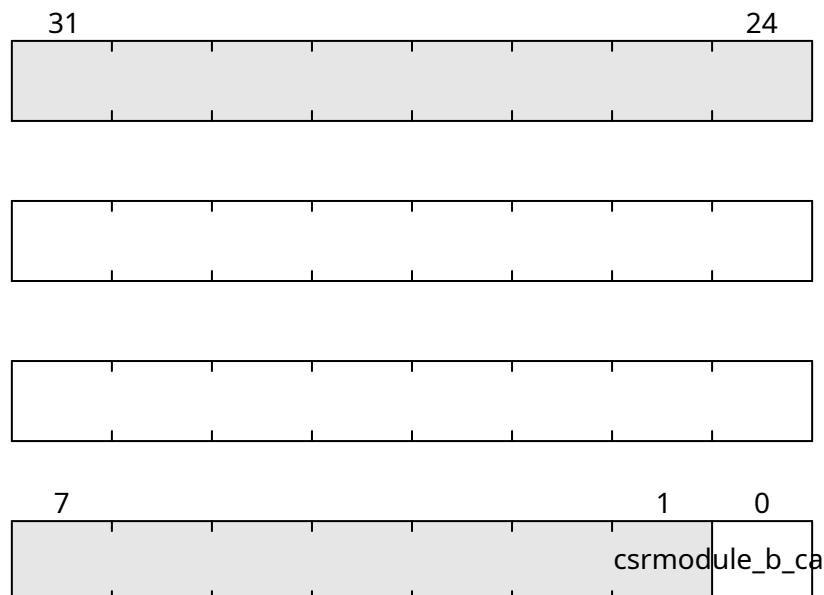


Fig. 24.74: DDRPHY_CSRMODULE_B_CADLY_RST

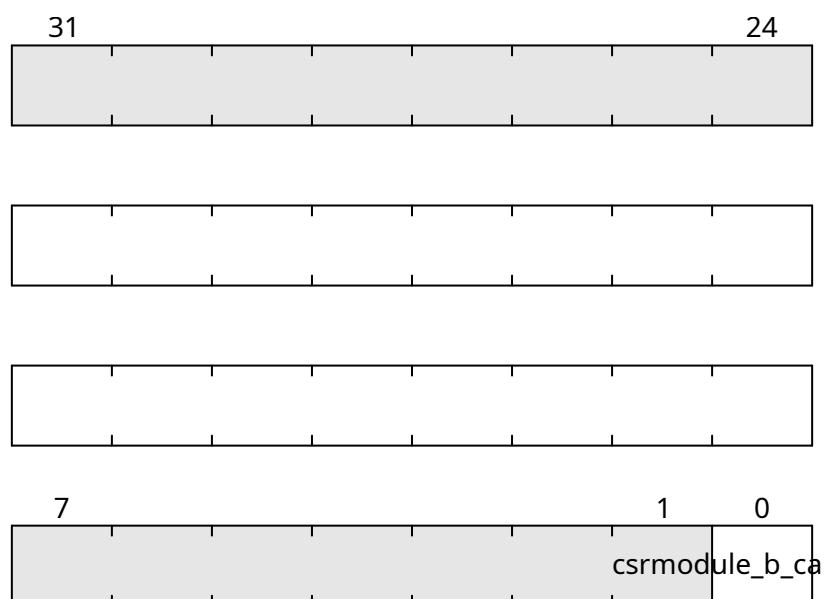


Fig. 24.75: DDRPHY_CSRMODULE_B_CADLY_INC

DDRPHY_CSRMODULE_B_PARDLY_RST

Address: $0xf0000800 + 0x128 = 0xf0000928$

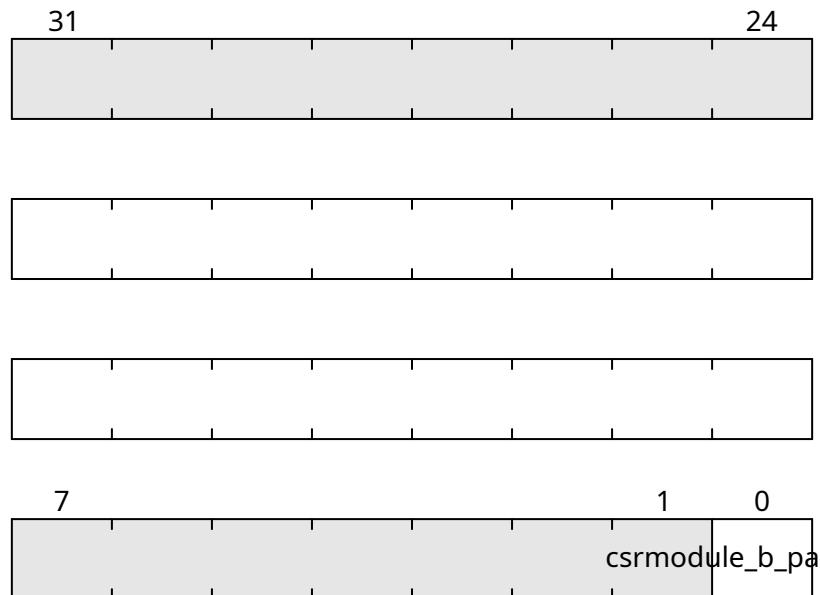


Fig. 24.76: DDRPHY_CSRMODULE_B_PARDLY_RST

DDRPHY_CSRMODULE_B_PARDLY_INC

Address: $0xf0000800 + 0x12c = 0xf000092c$

DDRPHY_CSRMODULE_B_CSDLY

Address: $0xf0000800 + 0x130 = 0xf0000930$

DDRPHY_CSRMODULE_B_CADLY

Address: $0xf0000800 + 0x134 = 0xf0000934$

DDRPHY_CSRMODULE_B_RDLY_DQ_RST

Address: $0xf0000800 + 0x138 = 0xf0000938$

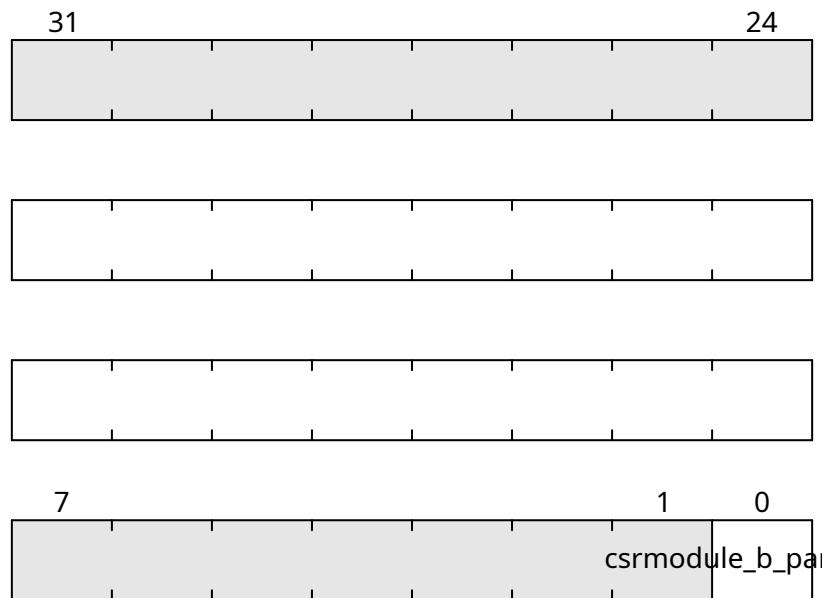


Fig. 24.77: DDRPHY_CSRMODULE_B_PARDLY_INC

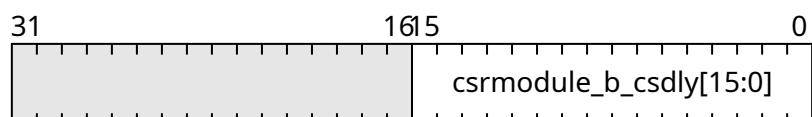


Fig. 24.78: DDRPHY_CSRMODULE_B_CSDLY

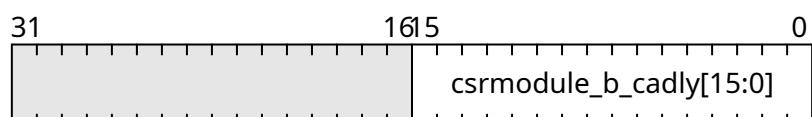


Fig. 24.79: DDRPHY_CSRMODULE_B_CADLY

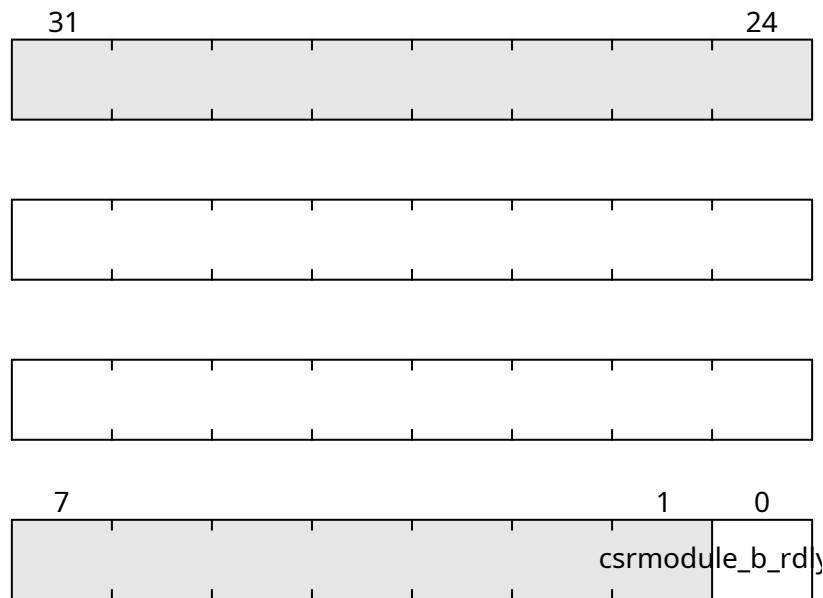


Fig. 24.80: DDRPHY_CSRMODULE_B_RDLY_DQ_RST

DDRPHY_CSRMODULE_B_RDLY_DQ_INC

Address: 0xf0000800 + 0x13c = 0xf000093c

DDRPHY_CSRMODULE_B_RDLY_DQS_RST

Address: 0xf0000800 + 0x140 = 0xf0000940

DDRPHY_CSRMODULE_B_RDLY_DQS_INC

Address: 0xf0000800 + 0x144 = 0xf0000944

DDRPHY_CSRMODULE_B_RDLY_DQS

Address: 0xf0000800 + 0x148 = 0xf0000948

DDRPHY_CSRMODULE_B_RDLY_DQ

Address: 0xf0000800 + 0x14c = 0xf000094c

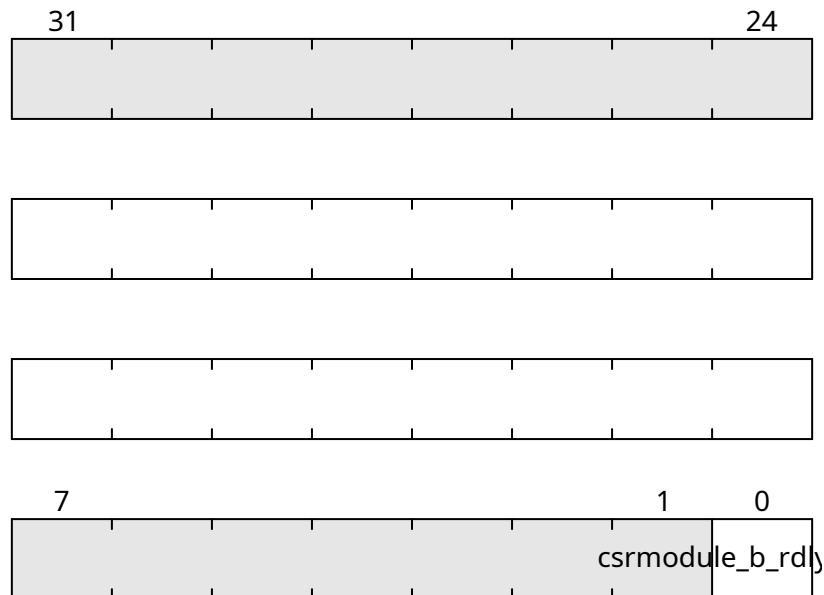


Fig. 24.81: DDRPHY_CSRMODULE_B_RDLY_DQ_INC

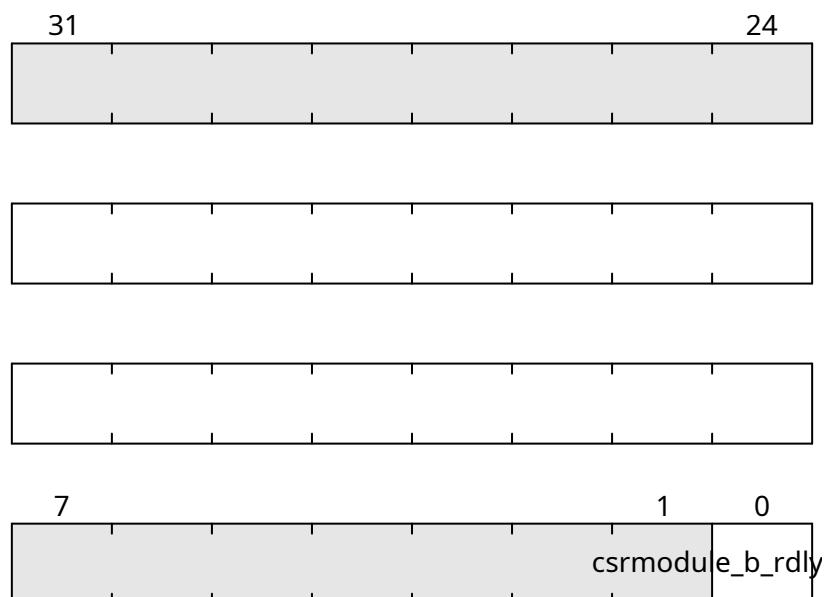


Fig. 24.82: DDRPHY_CSRMODULE_B_RDLY_DQS_RST

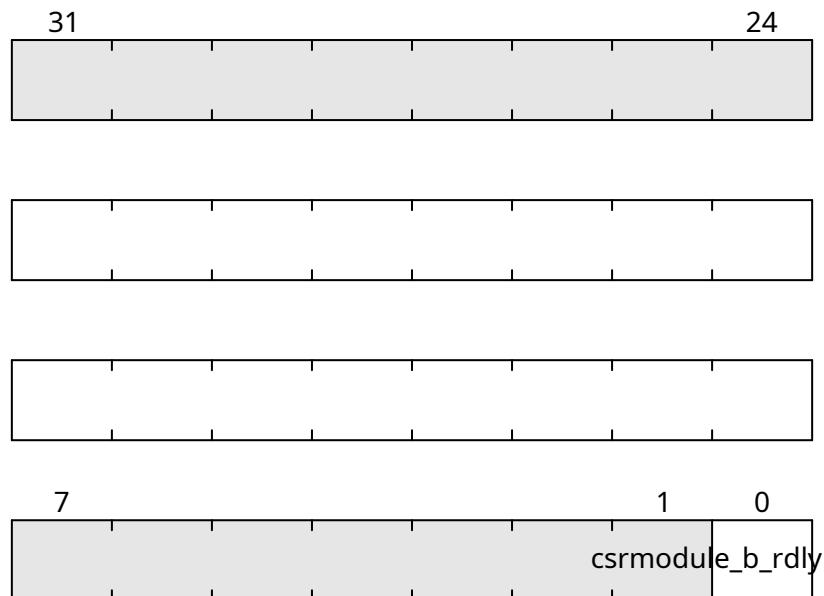


Fig. 24.83: DDRPHY_CSRMODULE_B_RDLY_DQS_INC

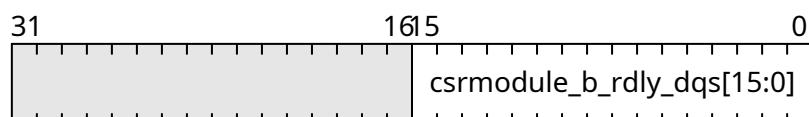


Fig. 24.84: DDRPHY_CSRMODULE_B_RDLY_DQS

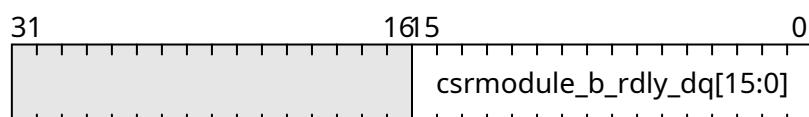


Fig. 24.85: DDRPHY_CSRMODULE_B_RDLY_DQ

DDRPHY_CSRMODULE_B_WDLY_DQ_RST

Address: $0xf0000800 + 0x150 = 0xf0000950$

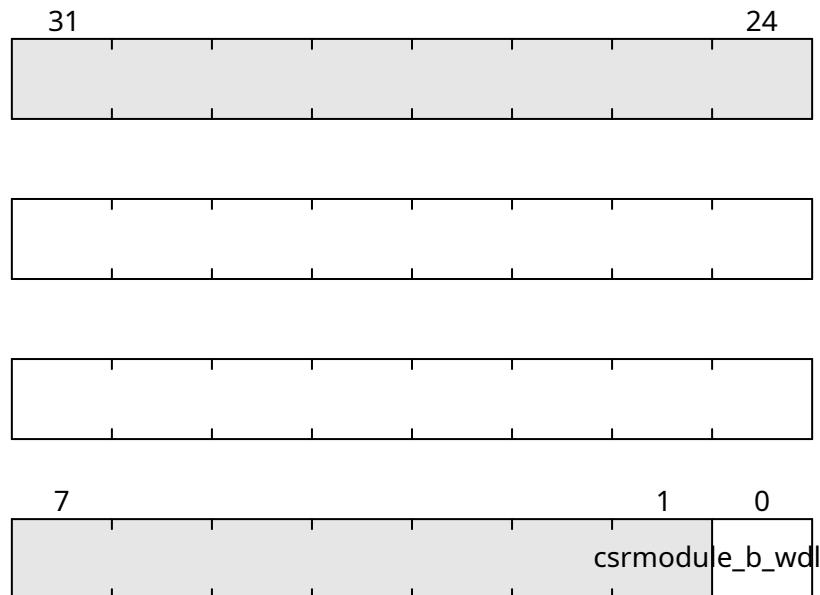


Fig. 24.86: DDRPHY_CSRMODULE_B_WDLY_DQ_RST

DDRPHY_CSRMODULE_B_WDLY_DQ_INC

Address: $0xf0000800 + 0x154 = 0xf0000954$

DDRPHY_CSRMODULE_B_WDLY_DM_RST

Address: $0xf0000800 + 0x158 = 0xf0000958$

DDRPHY_CSRMODULE_B_WDLY_DM_INC

Address: $0xf0000800 + 0x15c = 0xf000095c$

DDRPHY_CSRMODULE_B_WDLY_DQS_RST

Address: $0xf0000800 + 0x160 = 0xf0000960$

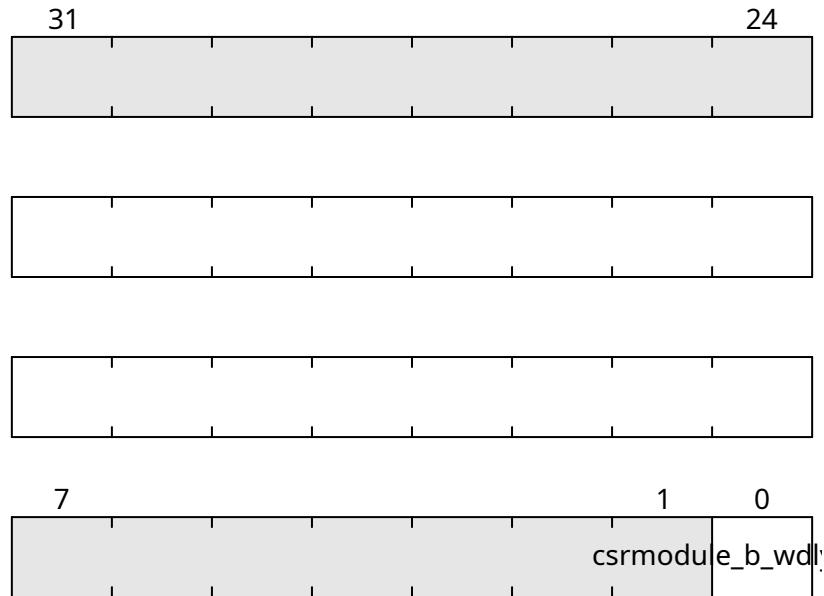


Fig. 24.87: `DDRPHY_CSRMODULE_B_WDLY_DQ_INC`

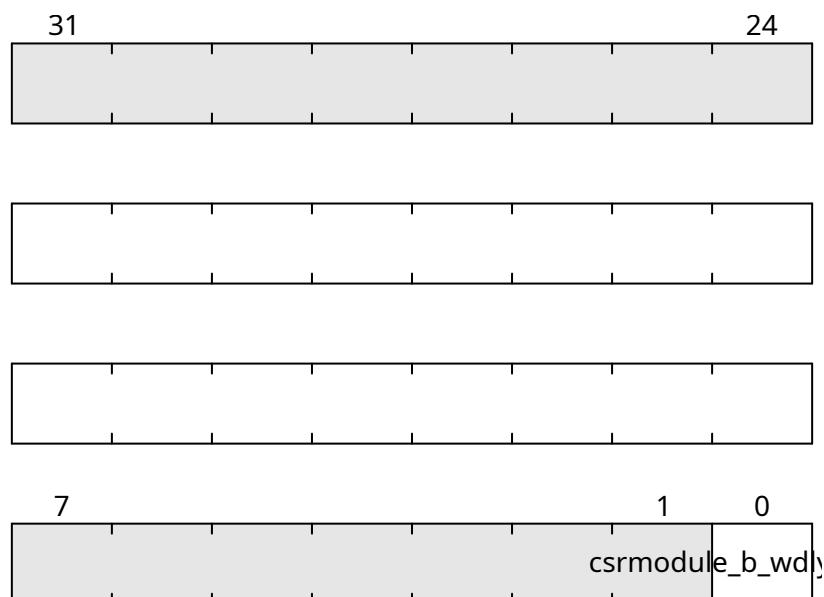


Fig. 24.88: `DDRPHY_CSRMODULE_B_WDLY_DM_RST`

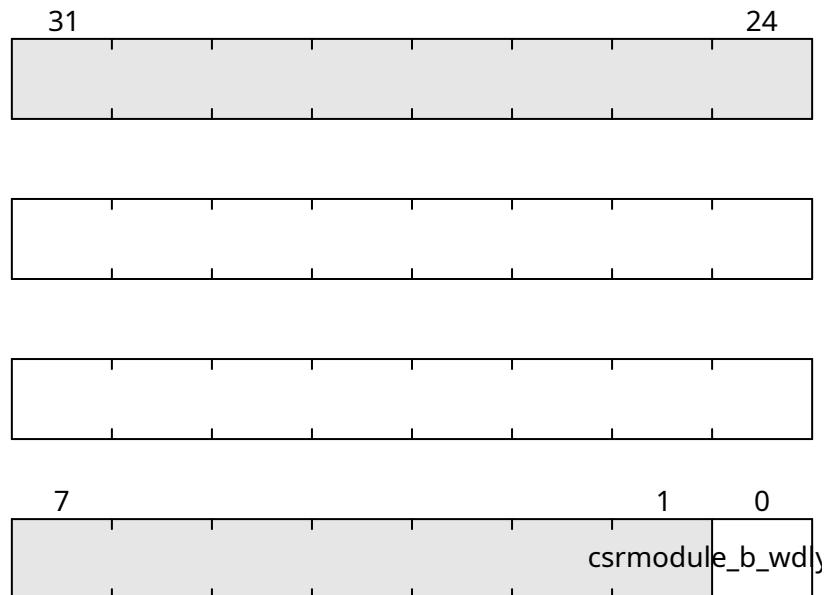


Fig. 24.89: DDRPHY_CSRMODULE_B_WDLY_DM_INC

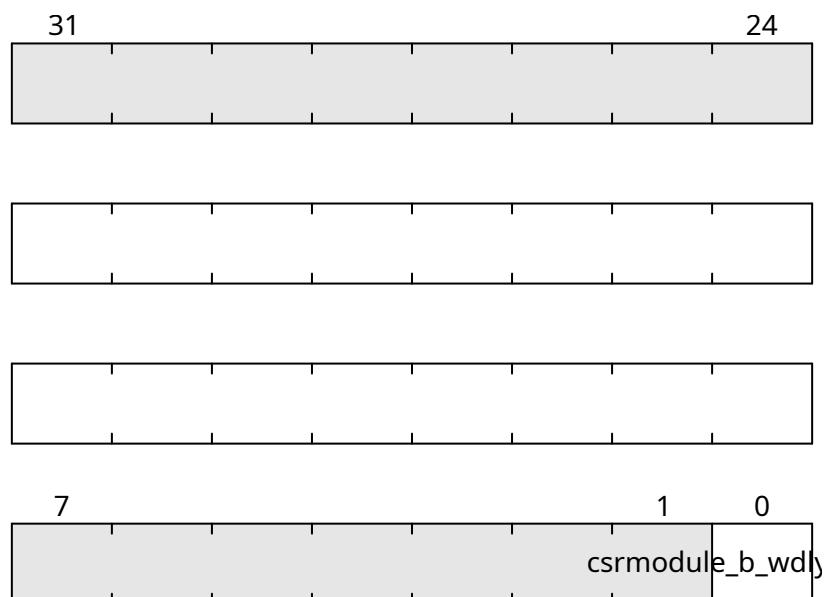


Fig. 24.90: DDRPHY_CSRMODULE_B_WDLY_DQS_RST

DDRPHY_CSRMODULE_B_WDLY_DQS_INC

Address: $0xf0000800 + 0x164 = 0xf0000964$

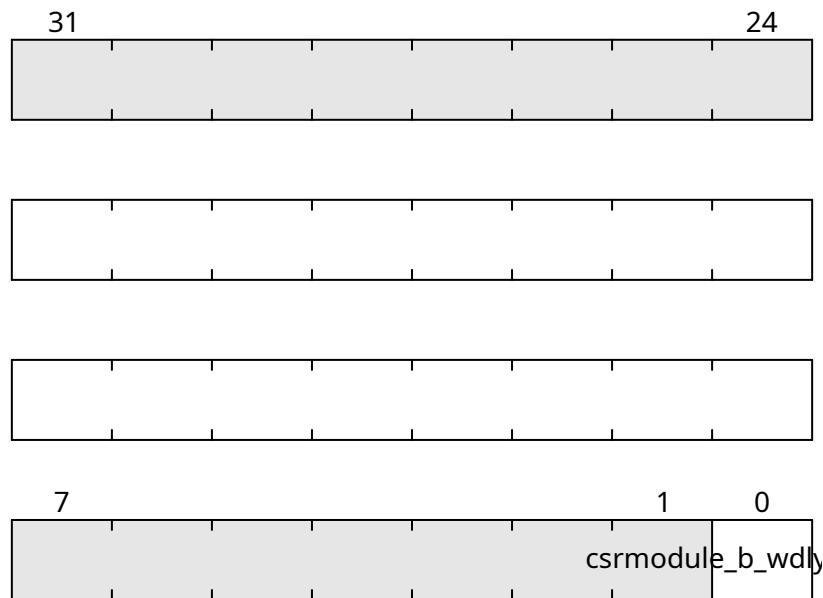


Fig. 24.91: DDRPHY_CSRMODULE_B_WDLY_DQS_INC

DDRPHY_CSRMODULE_B_WDLY_DQS

Address: $0xf0000800 + 0x168 = 0xf0000968$

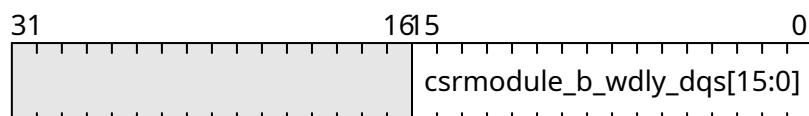


Fig. 24.92: DDRPHY_CSRMODULE_B_WDLY_DQS

DDRPHY_CSRMODULE_B_WDLY_DQ

Address: $0xf0000800 + 0x16c = 0xf000096c$

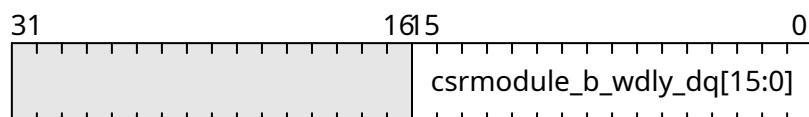


Fig. 24.93: DDRPHY_CSRMODULE_B_WDLY_DQ

DDRPHY_CSRMODULE_B_WDLY_DM

Address: $0xf0000800 + 0x170 = 0xf0000970$

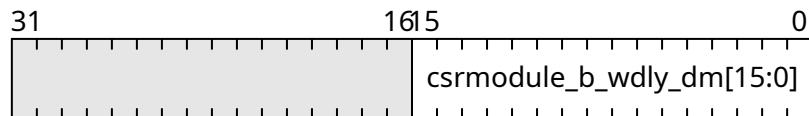


Fig. 24.94: DDRPHY_CSRMODULE_B_WDLY_DM

24.2.3 CONTROLLER_SETTINGS

Allows to change LiteDRAMController behaviour at runtime

Register Listing for CONTROLLER_SETTINGS

Register	Address
<i>CONTROLLER_SETTINGS_REFRESH</i>	<i>0xf0001000</i>

CONTROLLER_SETTINGS_REFRESH

Address: $0xf0001000 + 0x0 = 0xf0001000$

Enable/disable Refresh commands sending

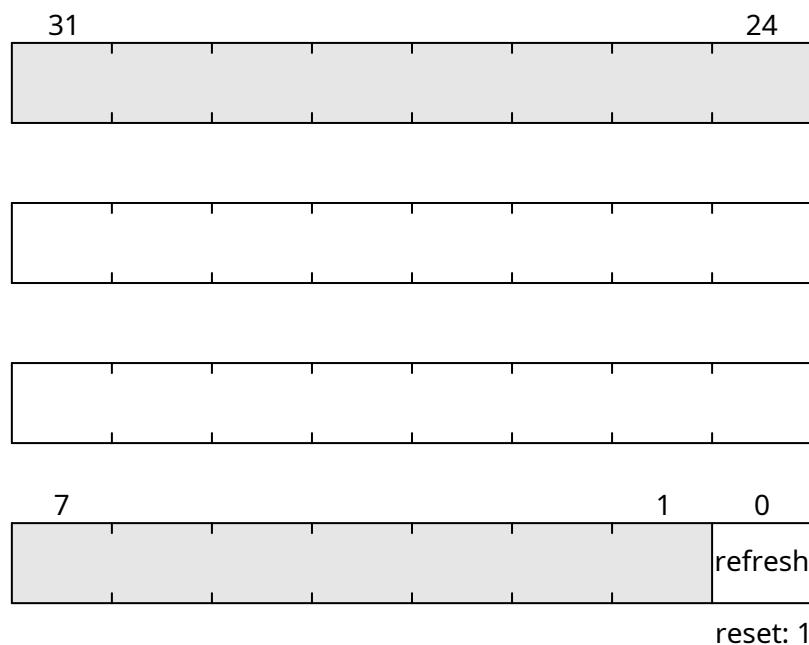


Fig. 24.95: CONTROLLER_SETTINGS_REFRESH

24.2.4 DDRCTRL

Register Listing for DDRCTRL

Register	Address
<i>DDRCTRL_INIT_DONE</i>	<i>0xf0001800</i>
<i>DDRCTRL_INIT_ERROR</i>	<i>0xf0001804</i>

DDRCTRL_INIT_DONE

Address: $0xf0001800 + 0x0 = 0xf0001800$

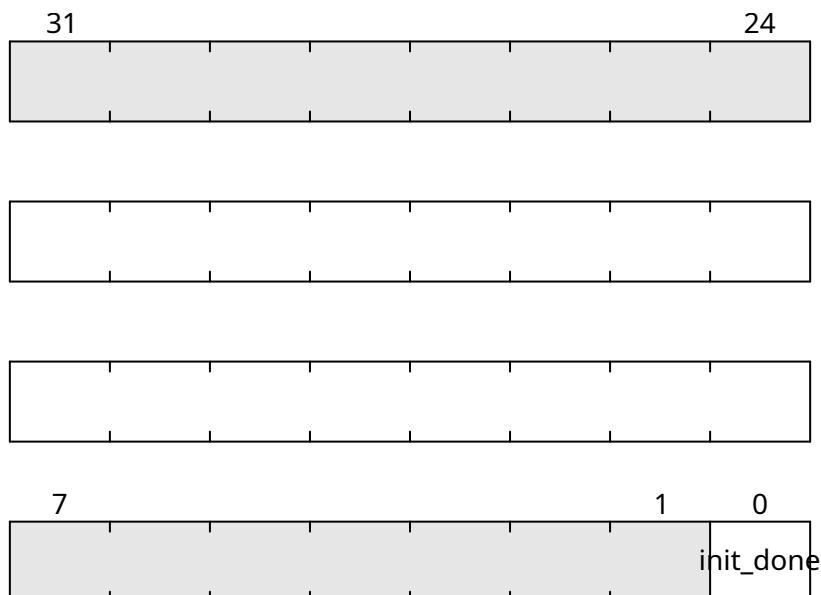


Fig. 24.96: DDRCTRL_INIT_DONE

DDRCTRL_INIT_ERROR

Address: $0xf0001800 + 0x4 = 0xf0001804$

24.2.5 ROWHAMMER

Row Hammer DMA attacker

This module allows to perform a Row Hammer attack by configuring it with two addresses that map to different rows of a single bank. When enabled, it will perform alternating DMA reads from the given locations, which will result in the DRAM controller having to repeatedly open/close rows at each read access.



Fig. 24.97: DDRCTRL_INIT_ERROR

Register Listing for ROWHAMMER

Register	Address
<i>ROWHAMMER_ENABLED</i>	<i>0xf0002000</i>
<i>ROWHAMMER_ADDRESS1</i>	<i>0xf0002004</i>
<i>ROWHAMMER_ADDRESS2</i>	<i>0xf0002008</i>
<i>ROWHAMMER_COUNT</i>	<i>0xf000200c</i>

ROWHAMMER_ENABLED

Address: $0xf0002000 + 0x0 = 0xf0002000$

Used to start/stop the operation of the module

ROWHAMMER_ADDRESS1

Address: $0xf0002000 + 0x4 = 0xf0002004$

First attacked address

ROWHAMMER_ADDRESS2

Address: $0xf0002000 + 0x8 = 0xf0002008$

Second attacked address



Fig. 24.98: ROWHAMMER_ENABLED

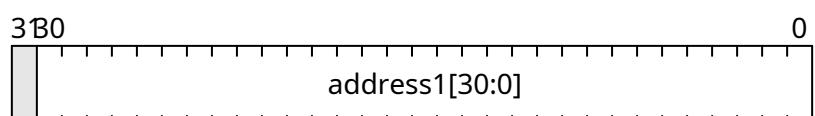


Fig. 24.99: ROWHAMMER_ADDRESS1

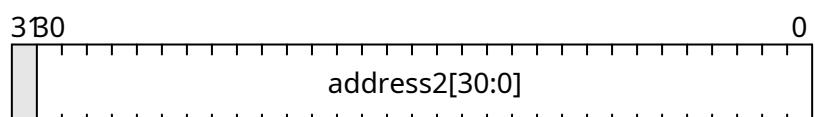


Fig. 24.100: ROWHAMMER_ADDRESS2

ROWHAMMER_COUNT

Address: $0xf0002000 + 0xc = 0xf000200c$

This is the number of DMA accesses performed. When the module is enabled, the value can be freely read. When the module is disabled, the register is clear-on-write and has to be read before the next attack.

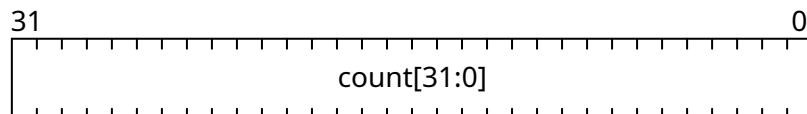


Fig. 24.101: ROWHAMMER_COUNT

24.2.6 WRITER

DMA DRAM writer.

Allows to fill DRAM with a predefined pattern using DMA.

Pattern

Provides access to RAM to store access pattern: *mem_addr* and *mem_data*. The pattern address space can be limited using the *data_mask*.

For example, having *mem_addr* filled with [0x04, 0x02, 0x03, ...] and *mem_data* filled with [0xff, 0xaa, 0x55, ...] and setting *data_mask* = 0b01, the pattern [(address, data), ...] written will be: [(0x04, 0xff), (0x02, 0xaa), (0x04, 0xff), ...] (wraps due to masking).

DRAM memory range that is being accessed can be configured using *mem_mask*.

To use this module, make sure that *ready* is 1, then write the desired number of transfers to *count*. Writing to the *start* CSR will initialize the operation. When the operation is ongoing *ready* will be 0.

Register Listing for WRITER

Register	Address
WRITER_START	0xf0002800
WRITER_READY	0xf0002804
WRITER_MODULO	0xf0002808
WRITER_COUNT	0xf000280c
WRITER_DONE	0xf0002810
WRITER_MEM_MASK	0xf0002814
WRITER_DATA_MASK	0xf0002818
WRITER_DATA_DIV	0xf000281c
WRITER_INVERTER_DIVISOR_MASK	0xf0002820
WRITER_INVERTER_SELECTION_MASK	0xf0002824
WRITER_LAST_ADDRESS	0xf0002828

WRITER START

Address: 0xf0002800 + 0x0 = 0xf0002800

Write to the register starts the transfer (if ready=1)



Fig. 24.102: WRITER_START

WRITER_READY

Address: $0xf0002800 + 0x4 = 0xf0002804$

Indicates that the transfer is not ongoing

WRITER MODULO

Address: 0xf0002800 + 0x8 = 0xf0002808

When set use modulo to calculate DMA transfers address rather than bit masking

WRITER COUNT

Address: $0xf0002800 + 0xc = 0xf000280c$

Desired number of DMA transfers

WRITER DONE

Address: $0xf0002800 + 0x10 = 0xf0002810$

Number of completed DMA transfers



Fig. 24.103: WRITER_READY



Fig. 24.104: WRITER_MODULO

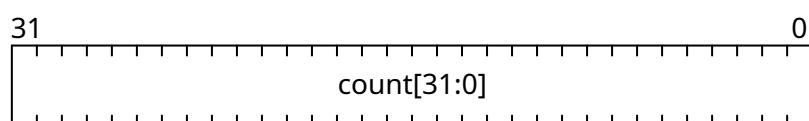


Fig. 24.105: WRITER_COUNT

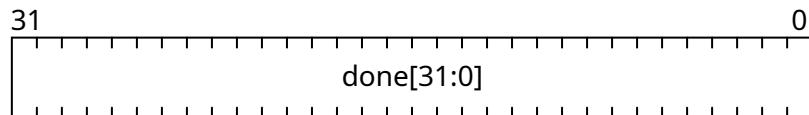


Fig. 24.106: WRITER_DONE

WRITER_MEM_MASK

Address: $0xf0002800 + 0x14 = 0xf0002814$

DRAM address mask for DMA transfers

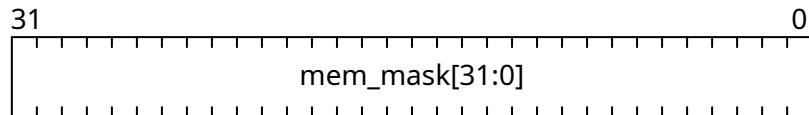


Fig. 24.107: WRITER_MEM_MASK

WRITER_DATA_MASK

Address: $0xf0002800 + 0x18 = 0xf0002818$

Pattern memory address mask

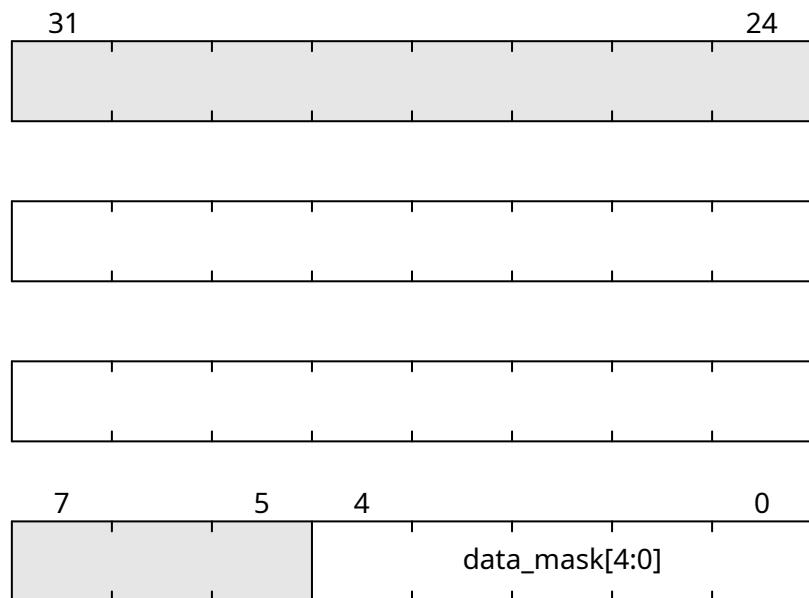


Fig. 24.108: WRITER_DATA_MASK

WRITER_DATA_DIV

Address: $0xf0002800 + 0x1c = 0xf000281c$

Pattern memory address divisor-1

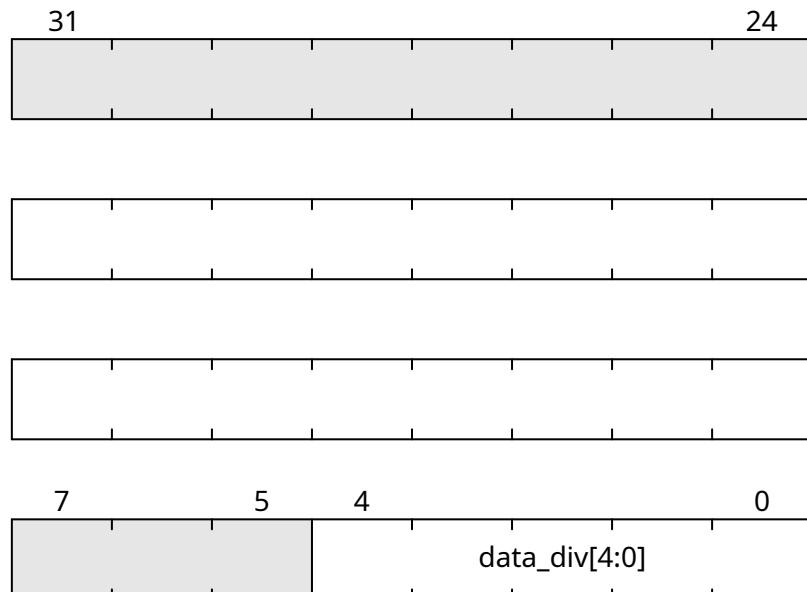


Fig. 24.109: WRITER_DATA_DIV

WRITER_INVERTER_DIVISOR_MASK

Address: $0xf0002800 + 0x20 = 0xf0002820$

Divisor mask for selecting rows for which pattern data gets inverted

WRITER_INVERTER_SELECTION_MASK

Address: $0xf0002800 + 0x24 = 0xf0002824$

Selection mask for selecting rows for which pattern data gets inverted

WRITER_LAST_ADDRESS

Address: $0xf0002800 + 0x28 = 0xf0002828$

Number of completed DMA transfers

24.2.7 READER

DMA DRAM reader.

Allows to check DRAM contents against a predefined pattern using DMA.

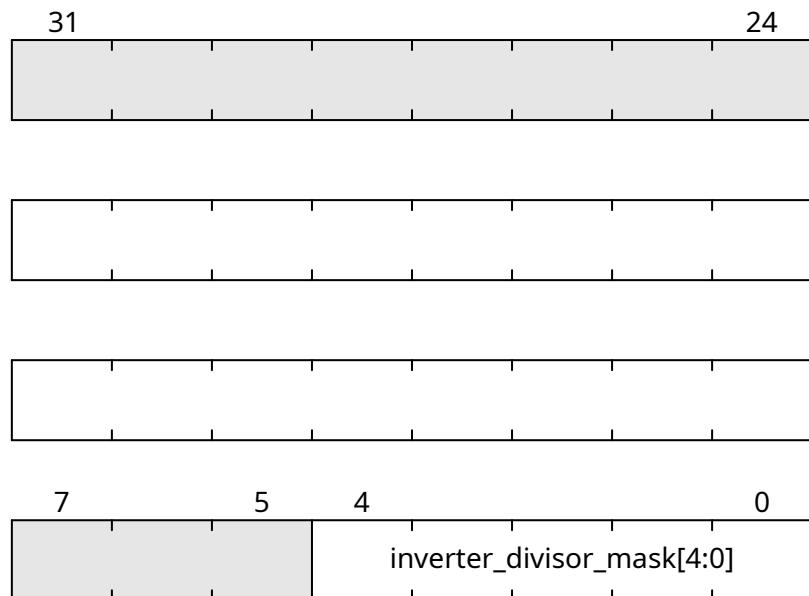


Fig. 24.110: WRITER_INVERTER_DIVISOR_MASK

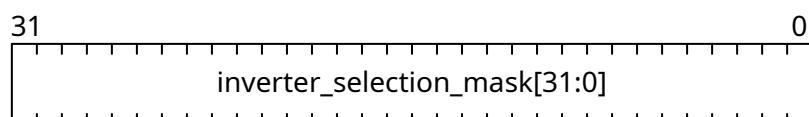


Fig. 24.111: WRITER_INVERTER_SELECTION_MASK

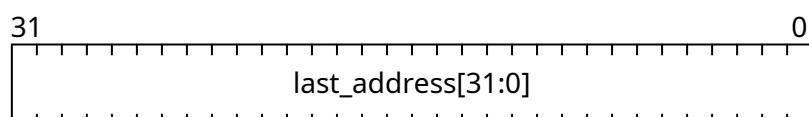


Fig. 24.112: WRITER_LAST_ADDRESS

Pattern

Provides access to RAM to store access pattern: *mem_addr* and *mem_data*. The pattern address space can be limited using the *data_mask*.

For example, having *mem_addr* filled with [0x04, 0x02, 0x03, ...] and *mem_data* filled with [0xff, 0xaa, 0x55, ...] and setting *data_mask* = 0b01, the pattern [(address, data), ...] written will be: [(0x04, 0xff), (0x02, 0xaa), (0x04, 0xff), ...] (wraps due to masking).

DRAM memory range that is being accessed can be configured using *mem_mask*.

To use this module, make sure that *ready* is 1, then write the desired number of transfers to *count*. Writing to the *start* CSR will initialize the operation. When the operation is ongoing *ready* will be 0.

Reading errors

This module allows to check the locations of errors in the memory. It scans the configured memory area and compares the values read to the predefined pattern. If *skip_fifo* is 0, this module will stop after each error encountered, so that it can be examined. Wait until the *error_ready* CSR is 1. Then use the CSRs *error_offset*, *error_data* and *error_expected* to examine the errors in the current transfer. To continue reading, write 1 to *error_continue* CSR. Setting *skip_fifo* to 1 will disable this behavior entirely.

The final number of errors can be read from *error_count*. NOTE: This value represents the number of erroneous *DMA transfers*.

The current progress can be read from the *done* CSR.

Register Listing for READER

Register	Address
<i>READER_START</i>	0xf0003000
<i>READER_READY</i>	0xf0003004
<i>READER_MODULO</i>	0xf0003008
<i>READER_COUNT</i>	0xf000300c
<i>READER_DONE</i>	0xf0003010
<i>READER_MEM_MASK</i>	0xf0003014
<i>READER_DATA_MASK</i>	0xf0003018
<i>READER_DATA_DIV</i>	0xf000301c
<i>READER_INVERTER_DIVISOR_MASK</i>	0xf0003020
<i>READER_INVERTER_SELECTION_MASK</i>	0xf0003024
<i>READER_ERROR_COUNT</i>	0xf0003028
<i>READER_SKIP_FIFO</i>	0xf000302c
<i>READER_ERROR_OFFSET</i>	0xf0003030
<i>READER_ERROR_DATA7</i>	0xf0003034
<i>READER_ERROR_DATA6</i>	0xf0003038
<i>READER_ERROR_DATA5</i>	0xf000303c
<i>READER_ERROR_DATA4</i>	0xf0003040
<i>READER_ERROR_DATA3</i>	0xf0003044
<i>READER_ERROR_DATA2</i>	0xf0003048
<i>READER_ERROR_DATA1</i>	0xf000304c

continues on next page

Table 24.2 – continued from previous page

Register	Address
<i>READER_ERROR_DATA0</i>	0xf0003050
<i>READER_ERROR_EXPECTED7</i>	0xf0003054
<i>READER_ERROR_EXPECTED6</i>	0xf0003058
<i>READER_ERROR_EXPECTED5</i>	0xf000305c
<i>READER_ERROR_EXPECTED4</i>	0xf0003060
<i>READER_ERROR_EXPECTED3</i>	0xf0003064
<i>READER_ERROR_EXPECTED2</i>	0xf0003068
<i>READER_ERROR_EXPECTED1</i>	0xf000306c
<i>READER_ERROR_EXPECTED0</i>	0xf0003070
<i>READER_ERROR_READY</i>	0xf0003074
<i>READER_ERROR_CONTINUE</i>	0xf0003078

READER_START

Address: $0xf0003000 + 0x0 = 0xf0003000$

Write to the register starts the transfer (if ready=1)



Fig. 24.113: READER_START

READER_READY

Address: $0xf0003000 + 0x4 = 0xf0003004$

Indicates that the transfer is not ongoing



Fig. 24.114: READER_READY

READER_MODULO

Address: $0xf0003000 + 0x8 = 0xf0003008$

When set use modulo to calculate DMA transfers address rather than bit masking

READER_COUNT

Address: $0xf0003000 + 0xc = 0xf000300c$

Desired number of DMA transfers

READER_DONE

Address: $0xf0003000 + 0x10 = 0xf0003010$

Number of completed DMA transfers

READER_MEM_MASK

Address: $0xf0003000 + 0x14 = 0xf0003014$

DRAM address mask for DMA transfers

READER_DATA_MASK

Address: $0xf0003000 + 0x18 = 0xf0003018$

Pattern memory address mask



Fig. 24.115: READER_MODULO

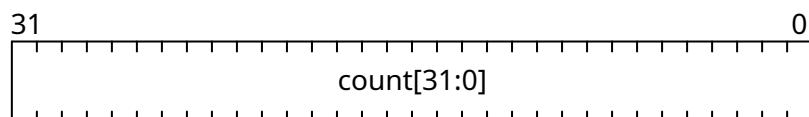


Fig. 24.116: READER_COUNT

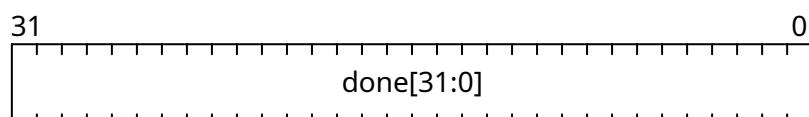


Fig. 24.117: READER_DONE

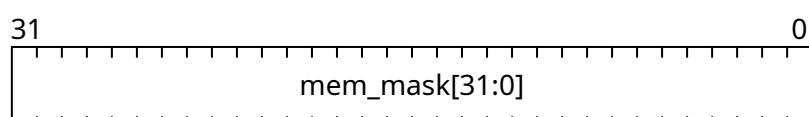


Fig. 24.118: READER_MEM_MASK

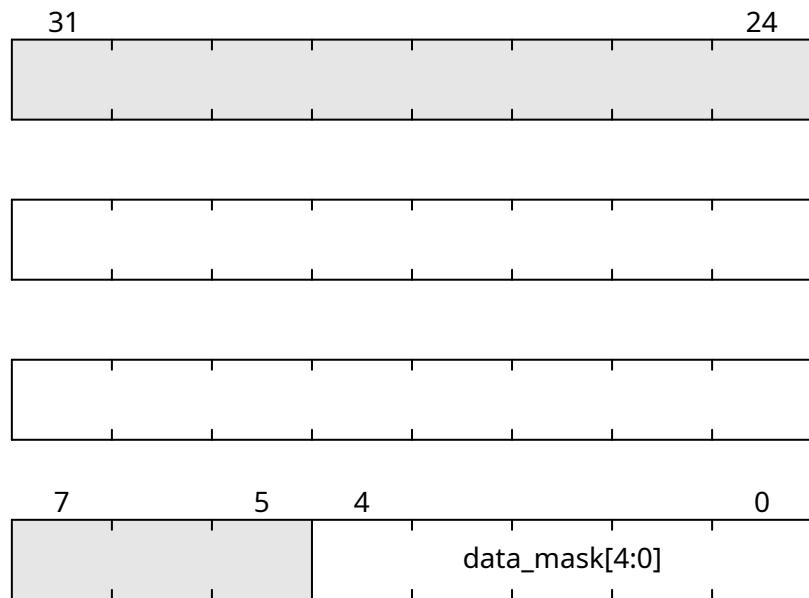


Fig. 24.119: READER_DATA_MASK

READER_DATA_DIV

Address: 0xf0003000 + 0x1c = 0xf000301c

Pattern memory address divisor-1

READER_INVERTER_DIVISOR_MASK

Address: 0xf0003000 + 0x20 = 0xf0003020

Divisor mask for selecting rows for which pattern data gets inverted

READER_INVERTER_SELECTION_MASK

Address: 0xf0003000 + 0x24 = 0xf0003024

Selection mask for selecting rows for which pattern data gets inverted

READER_ERROR_COUNT

Address: 0xf0003000 + 0x28 = 0xf0003028

Number of errors detected

READER_SKIP_FIFO

Address: 0xf0003000 + 0x2c = 0xf000302c

Skip waiting for user to read the errors FIFO

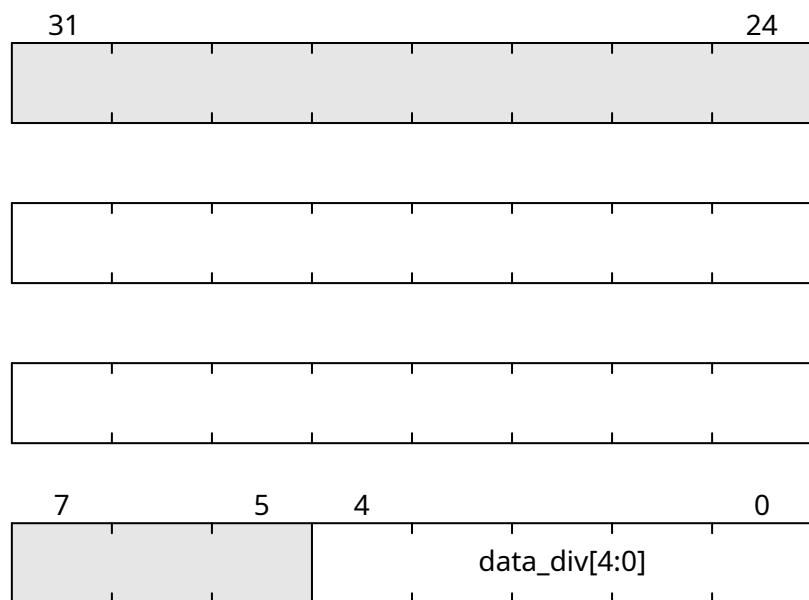


Fig. 24.120: READER_DATA_DIV

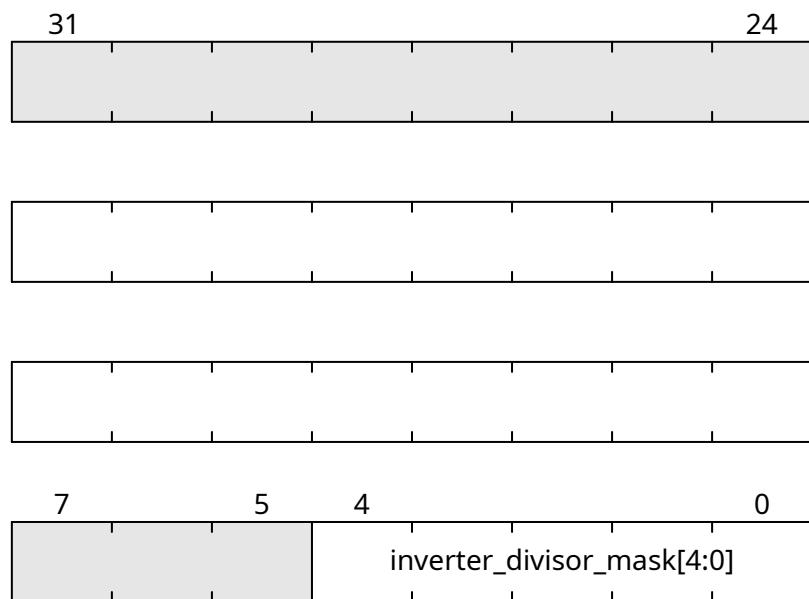


Fig. 24.121: READER_INVERTER_DIVISOR_MASK

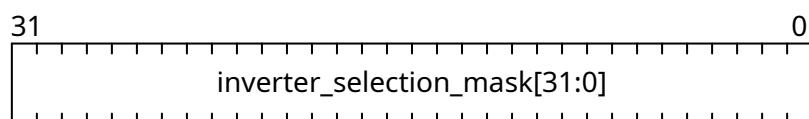


Fig. 24.122: READER_INVERTER_SELECTION_MASK

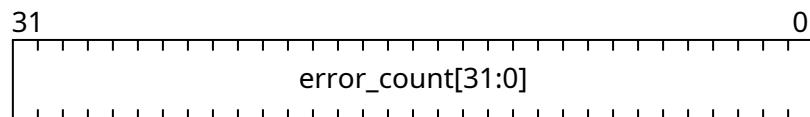


Fig. 24.123: READER_ERROR_COUNT



Fig. 24.124: READER_SKIP_FIFO

READER_ERROR_OFFSET

Address: $0xf0003000 + 0x30 = 0xf0003030$

Current offset of the error

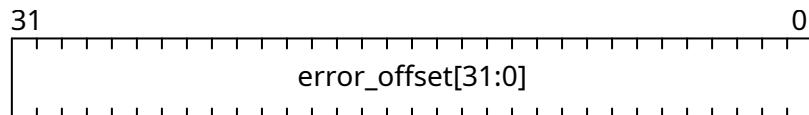


Fig. 24.125: READER_ERROR_OFFSET

READER_ERROR_DATA7

Address: $0xf0003000 + 0x34 = 0xf0003034$

Bits 224-255 of READER_ERROR_DATA. Erroneous value read from DRAM memory

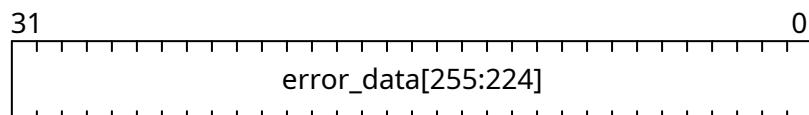


Fig. 24.126: READER_ERROR_DATA7

READER_ERROR_DATA6

Address: $0xf0003000 + 0x38 = 0xf0003038$

Bits 192-223 of READER_ERROR_DATA.

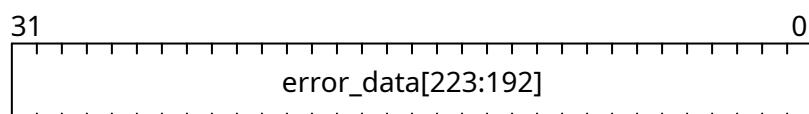


Fig. 24.127: READER_ERROR_DATA6

READER_ERROR_DATA5

Address: $0xf0003000 + 0x3c = 0xf000303c$

Bits 160-191 of READER_ERROR_DATA.

READER_ERROR_DATA4

Address: $0xf0003000 + 0x40 = 0xf0003040$

Bits 128-159 of READER_ERROR_DATA.

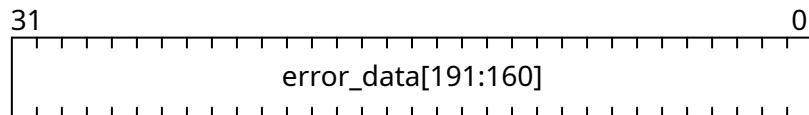


Fig. 24.128: READER_ERROR_DATA5

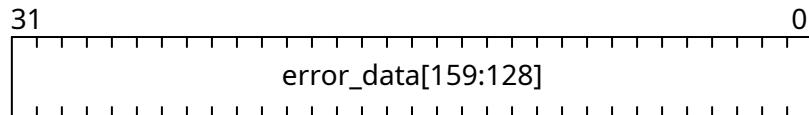


Fig. 24.129: READER_ERROR_DATA4

READER_ERROR_DATA3

Address: $0xf0003000 + 0x44 = 0xf0003044$

Bits 96-127 of *READER_ERROR_DATA*.

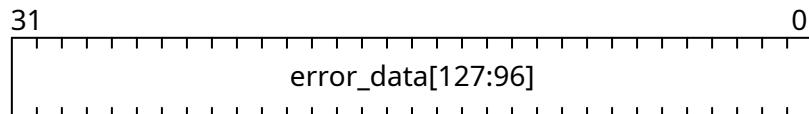


Fig. 24.130: READER_ERROR_DATA3

READER_ERROR_DATA2

Address: $0xf0003000 + 0x48 = 0xf0003048$

Bits 64-95 of *READER_ERROR_DATA*.

READER_ERROR_DATA1

Address: $0xf0003000 + 0x4c = 0xf000304c$

Bits 32-63 of *READER_ERROR_DATA*.

READER_ERROR_DATA0

Address: $0xf0003000 + 0x50 = 0xf0003050$

Bits 0-31 of *READER_ERROR_DATA*.

READER_ERROR_EXPECTED7

Address: $0xf0003000 + 0x54 = 0xf0003054$

Bits 224-255 of *READER_ERROR_EXPECTED*. Value expected to be read from DRAM memory

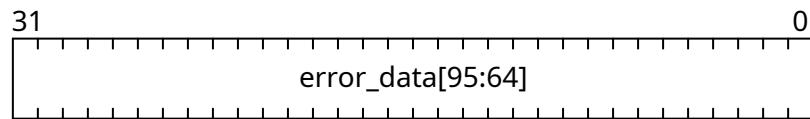


Fig. 24.131: READER_ERROR_DATA2

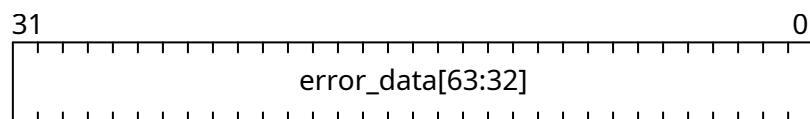


Fig. 24.132: READER_ERROR_DATA1

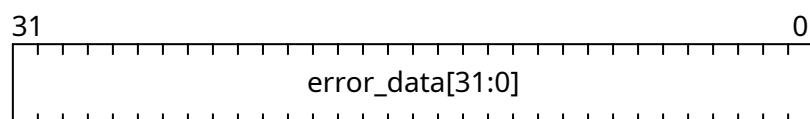


Fig. 24.133: READER_ERROR_DATA0

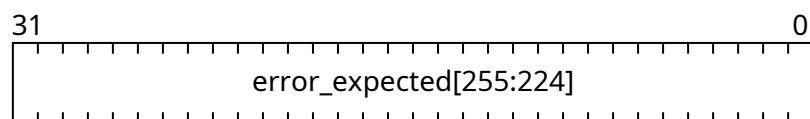


Fig. 24.134: READER_ERROR_EXPECTED7

READER_ERROR_EXPECTED6

Address: $0xf0003000 + 0x58 = 0xf0003058$

Bits 192-223 of *READER_ERROR_EXPECTED*.

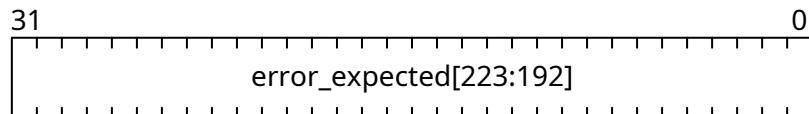


Fig. 24.135: READER_ERROR_EXPECTED6

READER_ERROR_EXPECTED5

Address: $0xf0003000 + 0x5c = 0xf000305c$

Bits 160-191 of *READER_ERROR_EXPECTED*.

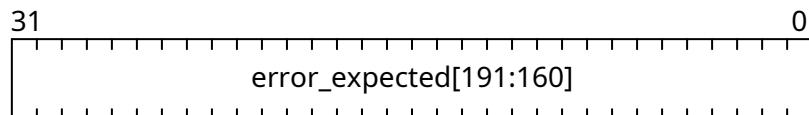


Fig. 24.136: READER_ERROR_EXPECTED5

READER_ERROR_EXPECTED4

Address: $0xf0003000 + 0x60 = 0xf0003060$

Bits 128-159 of *READER_ERROR_EXPECTED*.

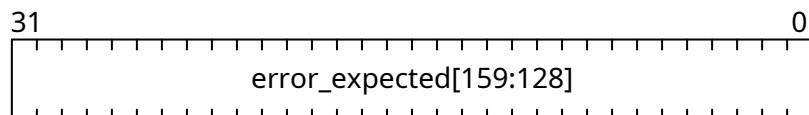


Fig. 24.137: READER_ERROR_EXPECTED4

READER_ERROR_EXPECTED3

Address: $0xf0003000 + 0x64 = 0xf0003064$

Bits 96-127 of *READER_ERROR_EXPECTED*.

READER_ERROR_EXPECTED2

Address: $0xf0003000 + 0x68 = 0xf0003068$

Bits 64-95 of *READER_ERROR_EXPECTED*.

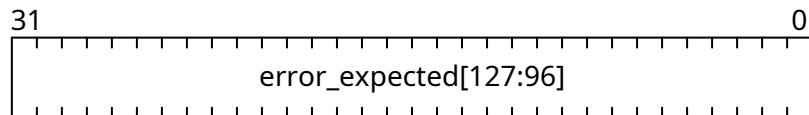


Fig. 24.138: READER_ERROR_EXPECTED3

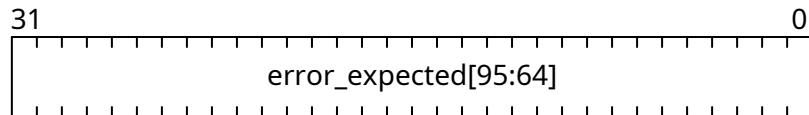


Fig. 24.139: READER_ERROR_EXPECTED2

READER_ERROR_EXPECTED1

Address: 0xf0003000 + 0x6c = 0xf000306c

Bits 32-63 of *READER_ERROR_EXPECTED*.

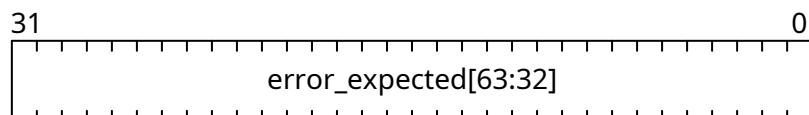


Fig. 24.140: READER_ERROR_EXPECTED1

READER_ERROR_EXPECTED0

Address: 0xf0003000 + 0x70 = 0xf0003070

Bits 0-31 of *READER_ERROR_EXPECTED*.

READER_ERROR_READY

Address: 0xf0003000 + 0x74 = 0xf0003074

Error detected and ready to read

READER_ERROR_CONTINUE

Address: 0xf0003000 + 0x78 = 0xf0003078

Continue reading until the next error

24.2.8 DFI_SWITCH

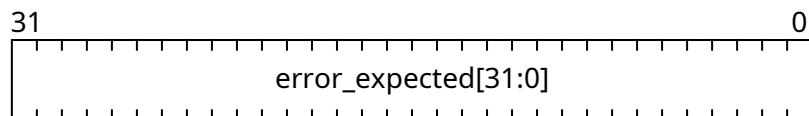


Fig. 24.141: READER_ERROR_EXPECTED0

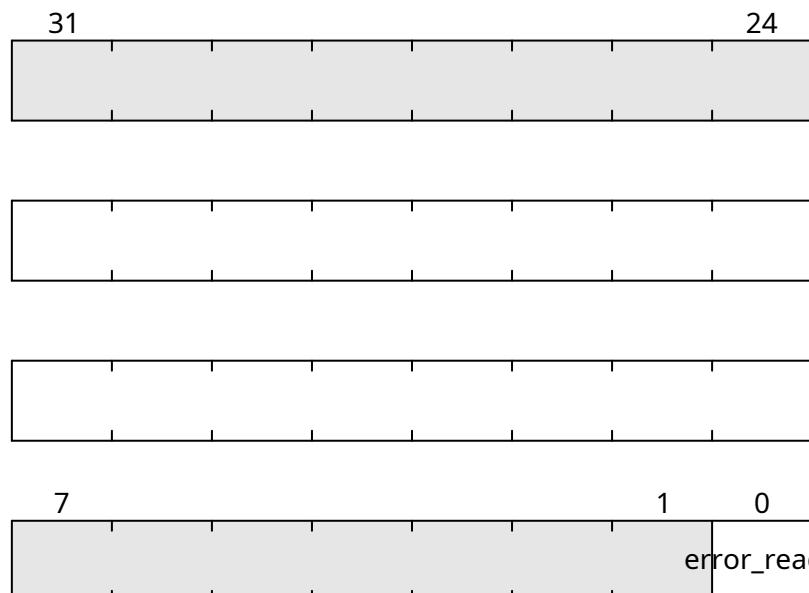


Fig. 24.142: READER_ERROR_READY

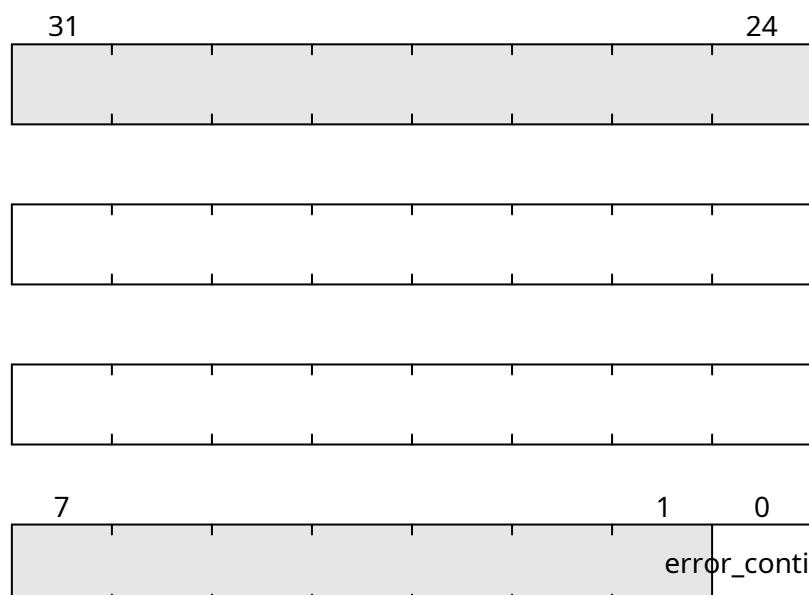


Fig. 24.143: READER_ERROR_CONTINUE

Register Listing for DFI_SWITCH

Register	Address
<i>DFI_SWITCH_REFRESH_COUNT1</i>	0xf0003800
<i>DFI_SWITCH_REFRESH_COUNT0</i>	0xf0003804
<i>DFI_SWITCH_AT_REFRESH1</i>	0xf0003808
<i>DFI_SWITCH_AT_REFRESH0</i>	0xf000380c
<i>DFI_SWITCH_REFRESH_UPDATE</i>	0xf0003810

DFI_SWITCH_REFRESH_COUNT1

Address: $0xf0003800 + 0x0 = 0xf0003800$

Bits 32-63 of *DFI_SWITCH_REFRESH_COUNT*. Count of all refresh commands issued (both by Memory Controller and the Payload Executor). Value is latched from internal counter on mode transition: MC -> PE or by writing to the *refresh_update* CSR.

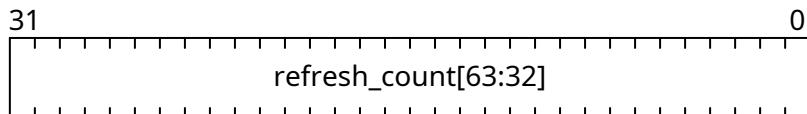


Fig. 24.144: DFI_SWITCH_REFRESH_COUNT1

DFI_SWITCH_REFRESH_COUNT0

Address: $0xf0003800 + 0x4 = 0xf0003804$

Bits 0-31 of *DFI_SWITCH_REFRESH_COUNT*.

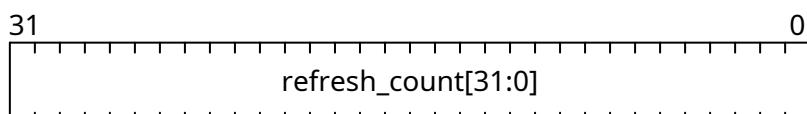


Fig. 24.145: DFI_SWITCH_REFRESH_COUNT0

DFI_SWITCH_AT_REFRESH1

Address: $0xf0003800 + 0x8 = 0xf0003808$

Bits 32-63 of *DFI_SWITCH_AT_REFRESH*. If set to a value different than 0 the mode transition MC -> PE will be performed only when the value of this register matches the current refresh commands count.

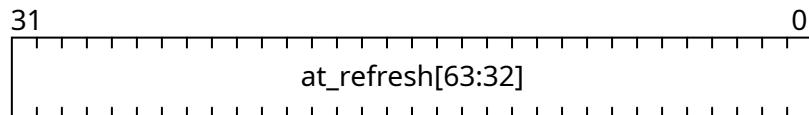


Fig. 24.146: DFI_SWITCH_AT_REFRESH1

DFI_SWITCH_AT_REFRESH0

Address: $0xf0003800 + 0xc = 0xf000380c$

Bits 0-31 of *DFI_SWITCH_AT_REFRESH*.

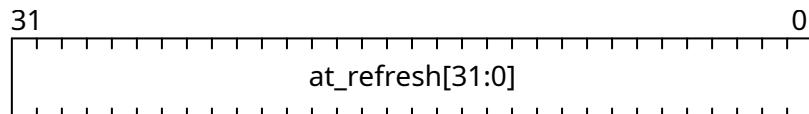


Fig. 24.147: DFI_SWITCH_AT_REFRESH0

DFI_SWITCH_REFRESH_UPDATE

Address: $0xf0003800 + 0x10 = 0xf0003810$

Force an update of the *refresh_count* CSR.

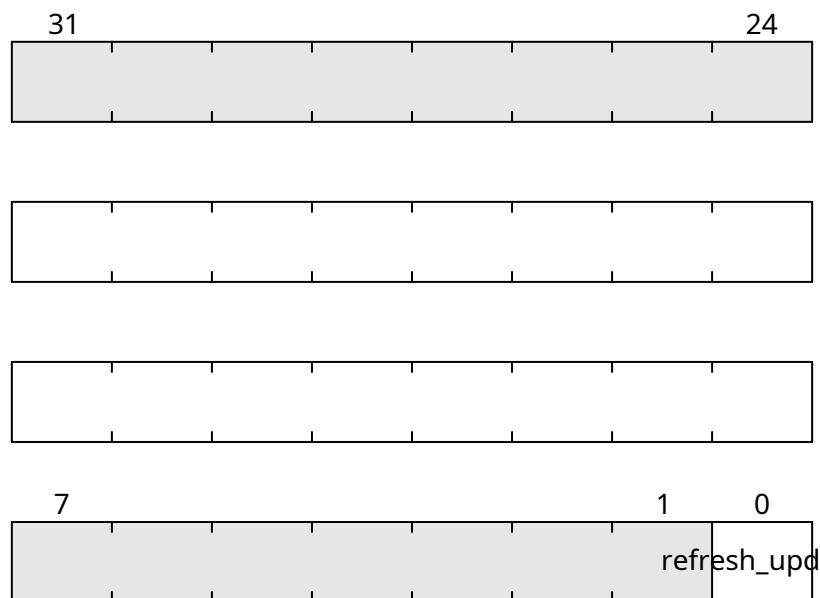


Fig. 24.148: DFI_SWITCH_REFRESH_UPDATE

24.2.9 PAYLOAD_EXECUTOR

Executes the DRAM payload from memory

Instruction decoder

All instructions are 32-bit. The format of most instructions is the same, except for the LOOP instruction, which has a constant TIMESLICE of 1.

NOOP with a TIMESLICE of 0 is a special case which is interpreted as STOP instruction. When this instruction is encountered execution gets finished immediately.

NOTE: TIMESLICE is the number of cycles the instruction will take. This means that instructions other than NOOP that use TIMESLICE=0 are illegal (although will silently be executed as having TIMESLICE=1).

NOTE2: LOOP instruction will *jump* COUNT times, meaning that the “code” inside the loop will effectively be executed COUNT+1 times.

Op codes:

Op	Value
NOOP	0b000
LOOP	0b111
ACT	0b100
PRE	0b101
REF	0b110
ZQC	0b001
READ	0b010

Instruction format:

LSB	MSB
dfi:	OP_CODE TIMESLICE ADDRESS
noop:	OP_CODE TIMESLICE_NOOP
loop:	OP_CODE COUNT JUMP
stop:	<NOOP> 0

Where ADDRESS depends on the DFI command and is one of:

LSB	MSB
RANK BANK COLUMN	
RANK BANK ROW	

Register Listing for PAYLOAD_EXECUTOR

Register	Address
PAYLOAD_EXECUTOR_START	0xf0004000
PAYLOAD_EXECUTOR_STATUS	0xf0004004
PAYLOAD_EXECUTOR_READ_COUNT	0xf0004008
PAYLOAD_EXECUTOR_EXEC_START1	0xf000400c
PAYLOAD_EXECUTOR_EXEC_START0	0xf0004010
PAYLOAD_EXECUTOR_EXEC_STOP1	0xf0004014
PAYLOAD_EXECUTOR_EXEC_STOP0	0xf0004018

PAYLOAD_EXECUTOR_START

Address: $0xf0004000 + 0x0 = 0xf0004000$

Writing to this register initializes payload execution



Fig. 24.149: PAYLOAD_EXECUTOR_START

PAYLOAD_EXECUTOR_STATUS

Address: $0xf0004000 + 0x4 = 0xf0004004$

Payload executor status register

Field	Name	Description
[0]	READY	Indicates that the executor is not running
[1]	OVERFLOW	Indicates the scratchpad memory address counter has overflowed due to the number of READ commands sent during execution

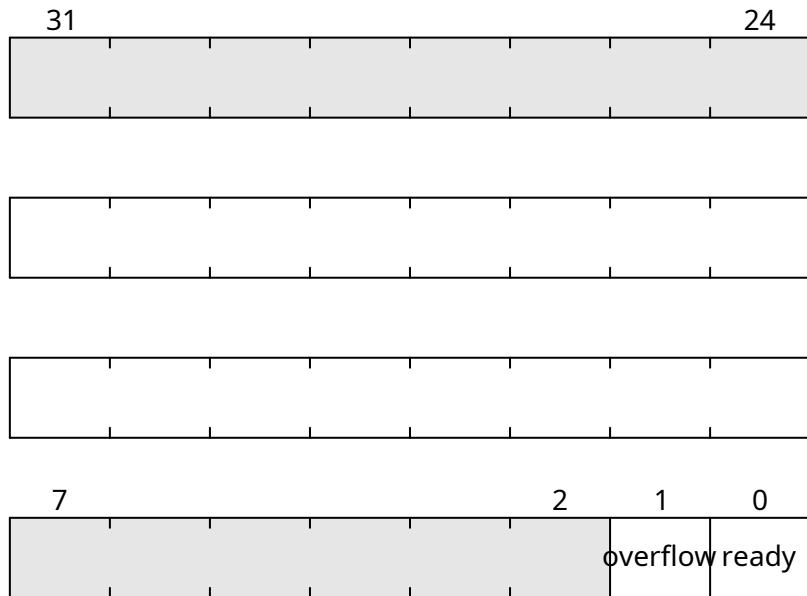


Fig. 24.150: PAYLOAD_EXECUTOR_STATUS

PAYLOAD_EXECUTOR_READ_COUNT

Address: $0xf0004000 + 0x8 = 0xf0004008$

Number of data from READ commands that is stored in the scratchpad memory

PAYLOAD_EXECUTOR_EXEC_START1

Address: $0xf0004000 + 0xc = 0xf000400c$

Bits 32-63 of PAYLOAD_EXECUTOR_EXEC_START. Number of cycles elapsed until the start of the payload execution.

PAYLOAD_EXECUTOR_EXEC_START0

Address: $0xf0004000 + 0x10 = 0xf0004010$

Bits 0-31 of PAYLOAD_EXECUTOR_EXEC_START.

PAYLOAD_EXECUTOR_EXEC_STOP1

Address: $0xf0004000 + 0x14 = 0xf0004014$

Bits 32-63 of PAYLOAD_EXECUTOR_EXEC_STOP. Number of cycles elapsed until the end of the payload execution.

PAYLOAD_EXECUTOR_EXEC_STOP0

Address: $0xf0004000 + 0x18 = 0xf0004018$

Bits 0-31 of PAYLOAD_EXECUTOR_EXEC_STOP.

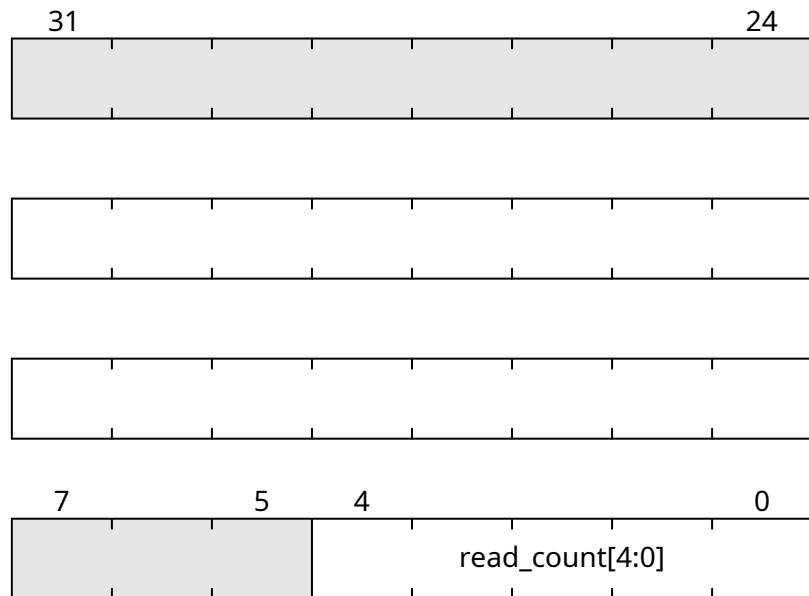


Fig. 24.151: PAYLOAD_EXECUTOR_READ_COUNT

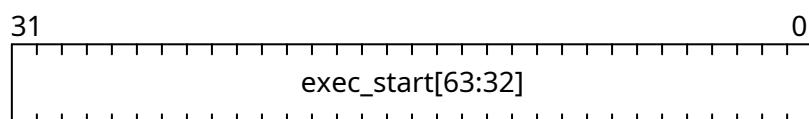


Fig. 24.152: PAYLOAD_EXECUTOR_EXEC_START1

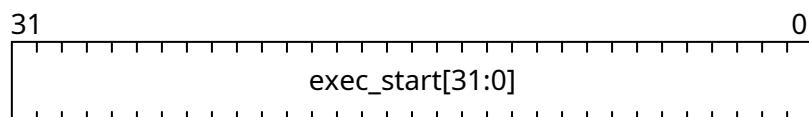


Fig. 24.153: PAYLOAD_EXECUTOR_EXEC_START0

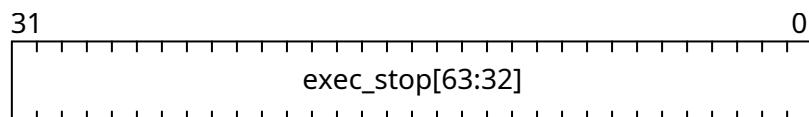


Fig. 24.154: PAYLOAD_EXECUTOR_EXEC_STOP1

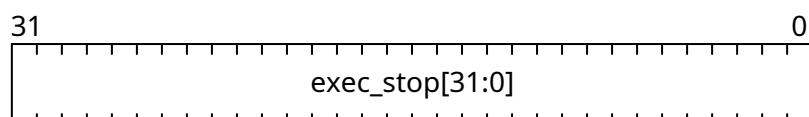


Fig. 24.155: PAYLOAD_EXECUTOR_EXEC_STOP0

24.2.10 I2C

Register Listing for I2C

Register	Address
<i>I2C_W</i>	0xf0004800
<i>I2C_R</i>	0xf0004804
<i>I2C_WORKER</i>	0xf0004808
<i>I2C_I2C_WORKER_START</i>	0xf000480c
<i>I2C_I2C_WORKER_I2C_CTRL</i>	0xf0004810
<i>I2C_I2C_WORKER_I2C_STATE</i>	0xf0004814
<i>I2C_I2C_WORKER_FIFOS_ACCESS_PORT</i>	0xf0004818
<i>I2C_I2C_WORKER_WRITE_FIFO_STATE</i>	0xf000481c
<i>I2C_I2C_WORKER_READ_FIFO_STATE</i>	0xf0004820

I2C_W

Address: $0xf0004800 + 0x0 = 0xf0004800$

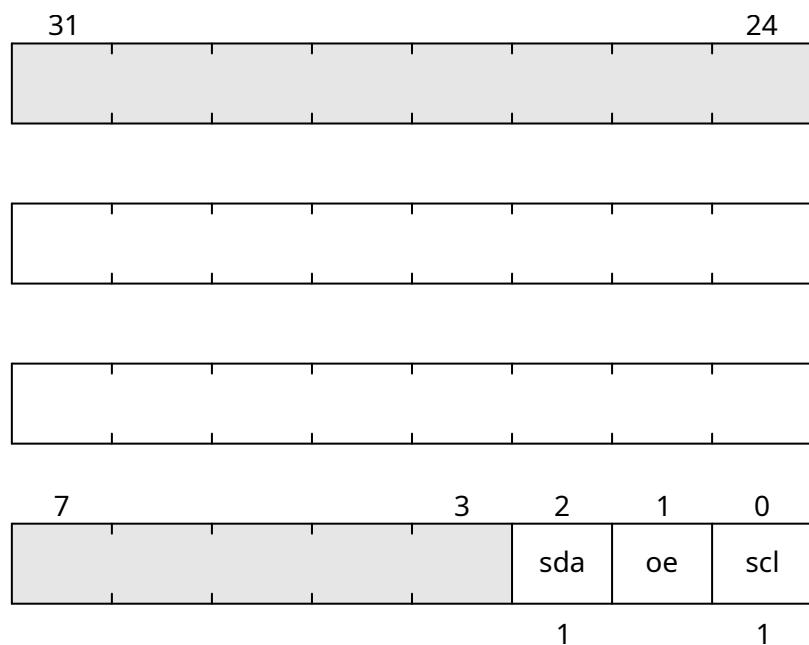


Fig. 24.156: I2C_W

Field	Name	Description

I2C_R

Address: $0xf0004800 + 0x4 = 0xf0004804$

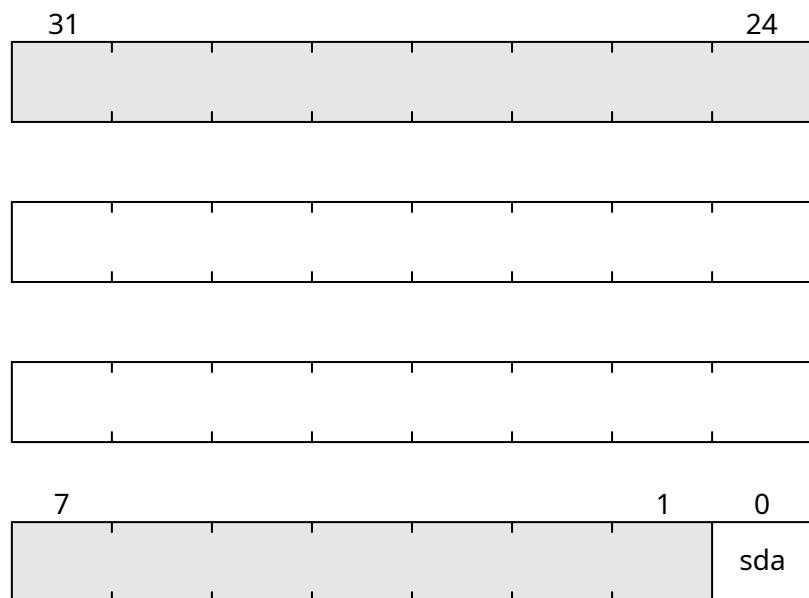


Fig. 24.157: I2C_R

Field	Name	Description

I2C_WORKER

Address: $0xf0004800 + 0x8 = 0xf0004808$

Field	Name	Description

I2C_I2C_WORKER_START

Address: $0xf0004800 + 0xc = 0xf000480c$

I2C_I2C_WORKER_I2C_CTRL

Address: $0xf0004800 + 0x10 = 0xf0004810$

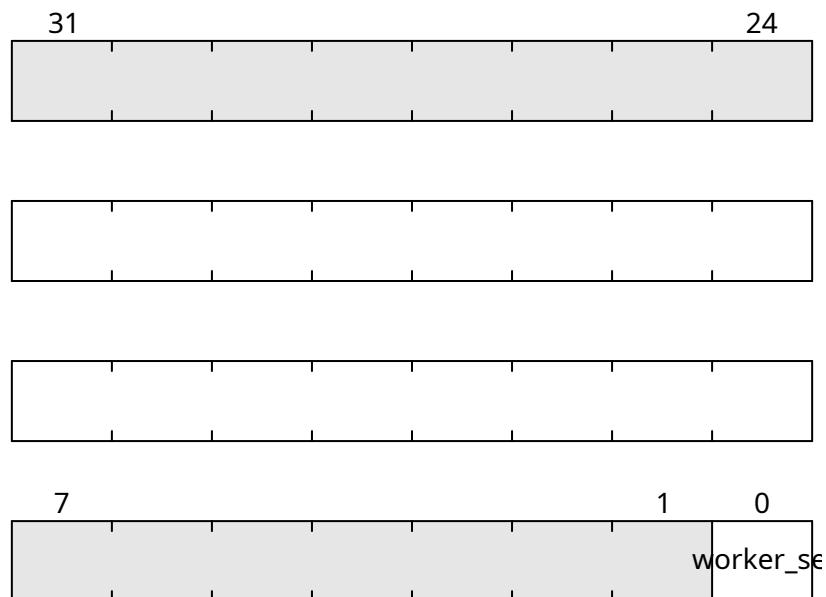


Fig. 24.158: I2C_WORKER

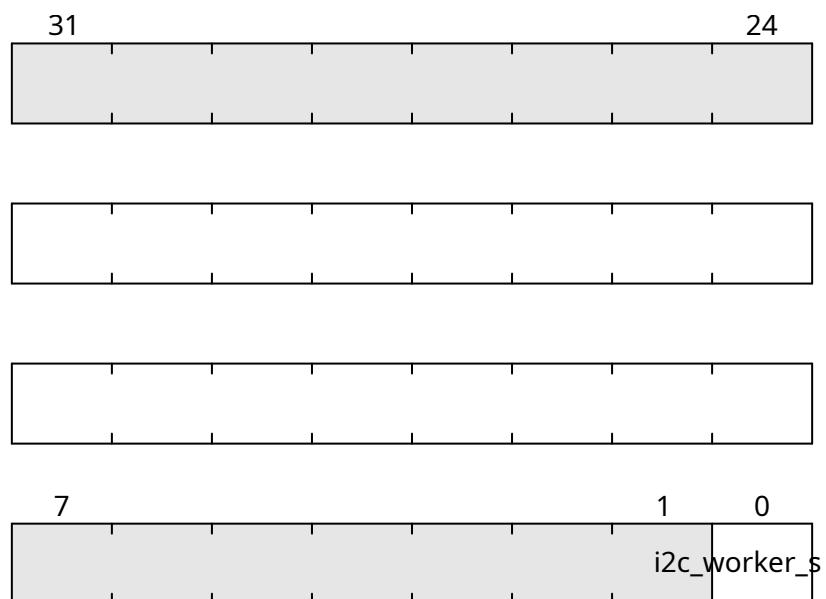


Fig. 24.159: I2C_I2C_WORKER_START



Fig. 24.160: I2C_I2C_WORKER_I2C_CTRL

Field	Name	Description

I2C_I2C_WORKER_I2C_STATE

Address: $0xf0004800 + 0x14 = 0xf0004814$

Field	Name	Description

I2C_I2C_WORKER_FIFOS_ACCESS_PORT

Address: $0xf0004800 + 0x18 = 0xf0004818$

I2C_I2C_WORKER_WRITE_FIFO_STATE

Address: $0xf0004800 + 0x1c = 0xf000481c$

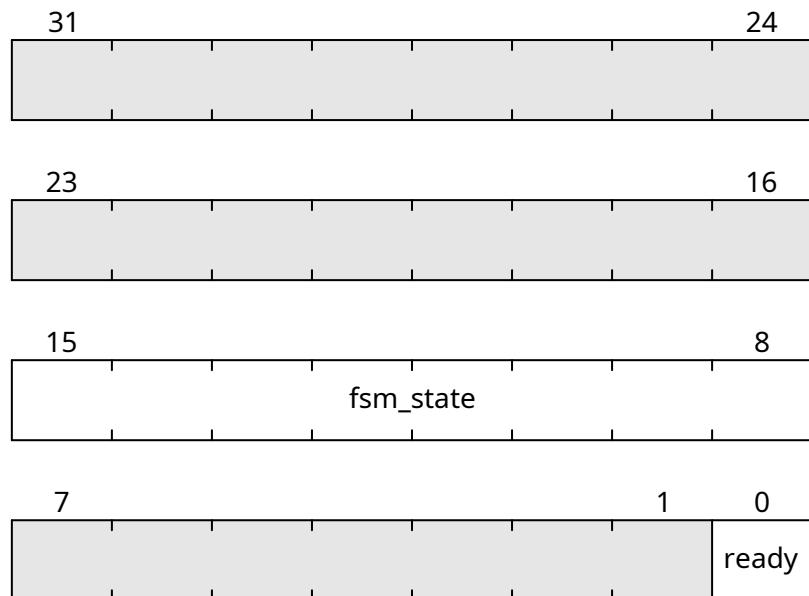


Fig. 24.161: `I2C_I2C_WORKER_I2C_STATE`

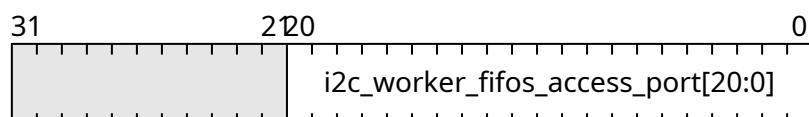


Fig. 24.162: `I2C_I2C_WORKER_FIFOS_ACCESS_PORT`

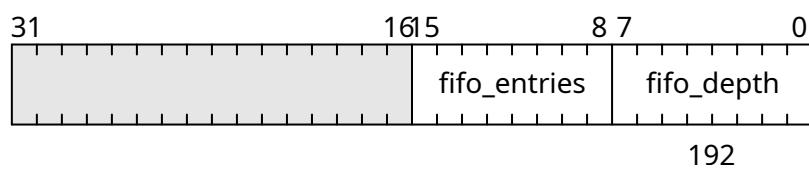


Fig. 24.163: `I2C_I2C_WORKER_WRITE_FIFO_STATE`

Field	Name	Description

I2C_I2C_WORKER_READ_FIFO_STATE

Address: $0xf0004800 + 0x20 = 0xf0004820$

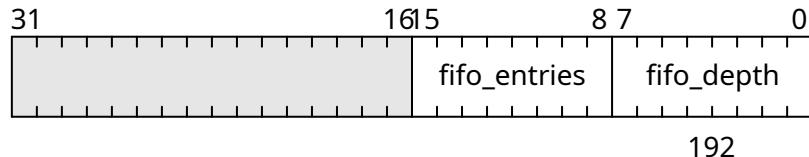


Fig. 24.164: I2C_I2C_WORKER_READ_FIFO_STATE

Field	Name	Description

24.2.11 CTRL

Register Listing for CTRL

Register	Address
<i>CTRL_RESET</i>	<i>0xf0005000</i>
<i>CTRL_SCRATCH</i>	<i>0xf0005004</i>
<i>CTRL_BUS_ERRORS</i>	<i>0xf0005008</i>

CTRL_RESET

Address: $0xf0005000 + 0x0 = 0xf0005000$

Field	Name	Description
[0]	SOC_RST	Write 1 to this register to reset the full SoC (Pulse Reset)
[1]	CPU_RST	Write 1 to this register to reset the CPU(s) of the SoC (Hold Reset)

CTRL_SCRATCH

Address: $0xf0005000 + 0x4 = 0xf0005004$

Use this register as a scratch space to verify that software read/write accesses to the Wishbone/CSR bus are working correctly. The initial reset value of 0x1234578 can be used to verify endianness.

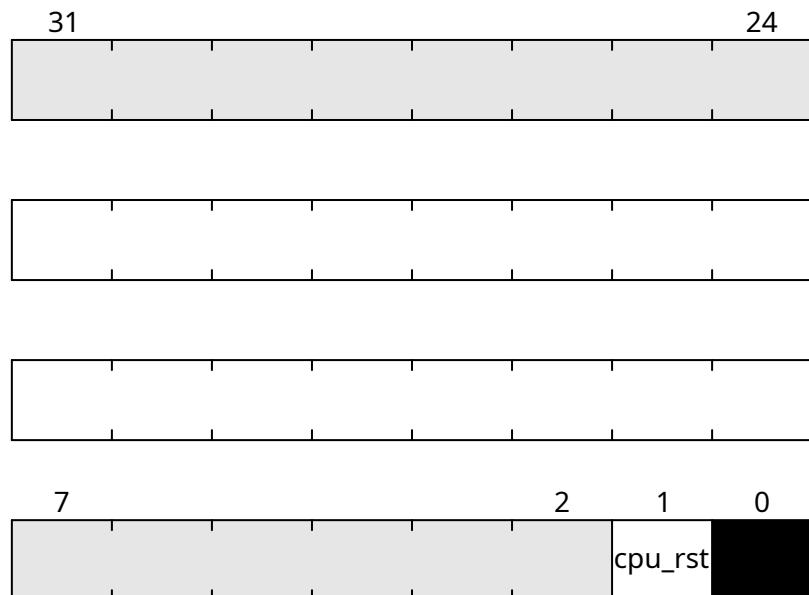


Fig. 24.165: CTRL_RESET

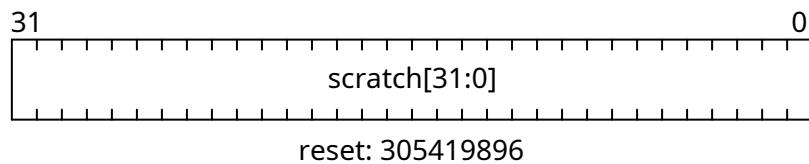


Fig. 24.166: CTRL_SCRATCH

CTRL_BUS_ERRORS

Address: $0xf0005000 + 0x8 = 0xf0005008$

Total number of Wishbone bus errors (timeouts) since start.

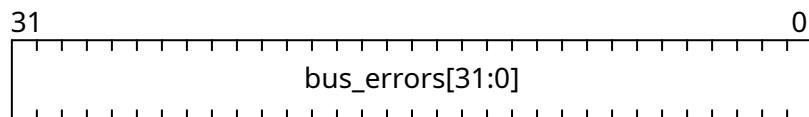


Fig. 24.167: CTRL_BUS_ERRORS

24.2.12 ETPHY

Register Listing for ETPHY

Register	Address
<i>ETPHY_CRG_RESET</i>	<i>0xf0005800</i>
<i>ETPHY_MDIO_W</i>	<i>0xf0005804</i>
<i>ETPHY_MDIO_R</i>	<i>0xf0005808</i>

ETPHY_CRG_RESET

Address: $0xf0005800 + 0x0 = 0xf0005800$

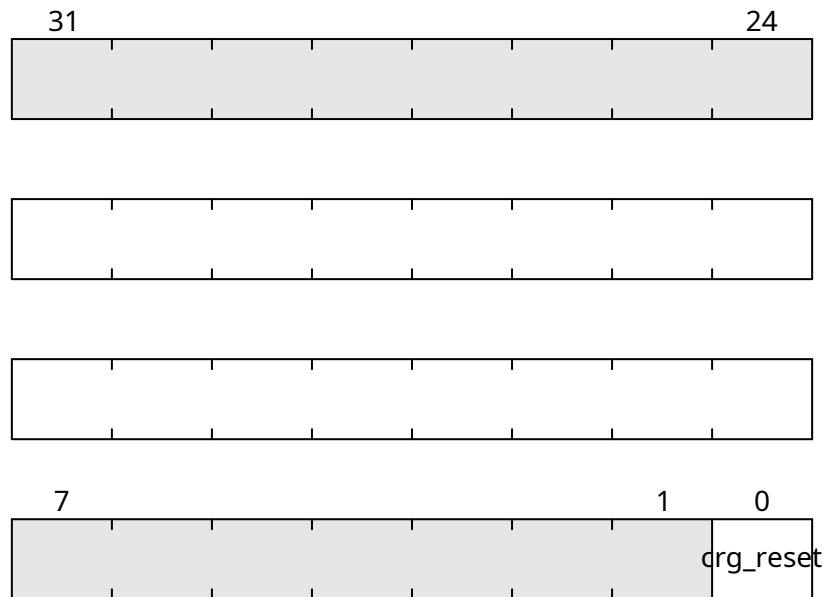


Fig. 24.168: ETPHY_CRG_RESET

ETHPHY_MDIO_W

Address: $0xf0005800 + 0x4 = 0xf0005804$

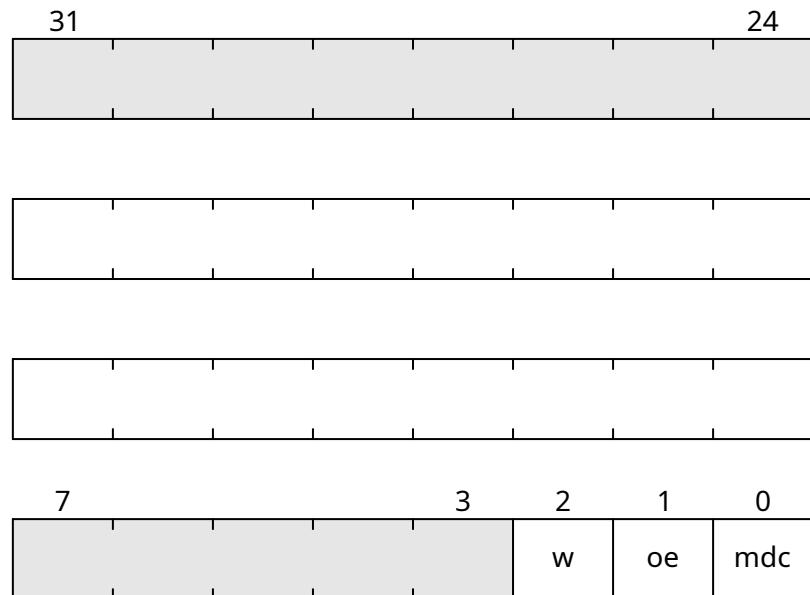


Fig. 24.169: ETHPHY_MDIO_W

Field	Name	Description

ETHPHY_MDIO_R

Address: $0xf0005800 + 0x8 = 0xf0005808$

Field	Name	Description

24.2.13 IDENTIFIER_MEM

Register Listing for IDENTIFIER_MEM

Register	Address
<i>IDENTIFIER_MEM</i>	<i>0xf0006000</i>



Fig. 24.170: ETHPHY_MDIO_R

IDENTIFIER_MEM

Address: 0xf0006000 + 0x0 = 0xf0006000

8 x 109-bit memory

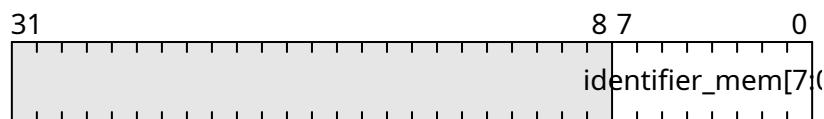


Fig. 24.171: IDENTIFIER_MEM

24.2.14 MAIN

Register Listing for MAIN

Register	Address
<i>MAIN_A_DQ_REMAPPING3</i>	<i>0xf0006800</i>
<i>MAIN_A_DQ_REMAPPING2</i>	<i>0xf0006804</i>
<i>MAIN_A_DQ_REMAPPING1</i>	<i>0xf0006808</i>
<i>MAIN_A_DQ_REMAPPING0</i>	<i>0xf000680c</i>
<i>MAIN_B_DQ_REMAPPING3</i>	<i>0xf0006810</i>
<i>MAIN_B_DQ_REMAPPING2</i>	<i>0xf0006814</i>
<i>MAIN_B_DQ_REMAPPING1</i>	<i>0xf0006818</i>
<i>MAIN_B_DQ_REMAPPING0</i>	<i>0xf000681c</i>

MAIN_A_DQ_REMAPPING3

Address: $0xf0006800 + 0x0 = 0xf0006800$

Bits 96-127 of *MAIN_A_DQ_REMAPPING*.

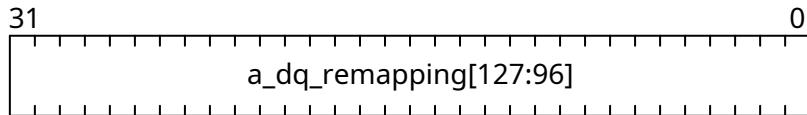


Fig. 24.172: MAIN_A_DQ_REMAPPING3

MAIN_A_DQ_REMAPPING2

Address: $0xf0006800 + 0x4 = 0xf0006804$

Bits 64-95 of *MAIN_A_DQ_REMAPPING*.

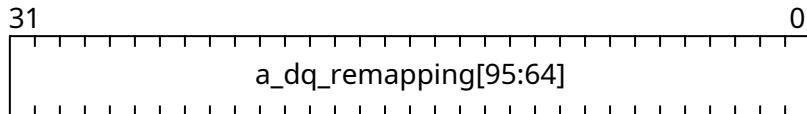


Fig. 24.173: MAIN_A_DQ_REMAPPING2

MAIN_A_DQ_REMAPPING1

Address: $0xf0006800 + 0x8 = 0xf0006808$

Bits 32-63 of *MAIN_A_DQ_REMAPPING*.

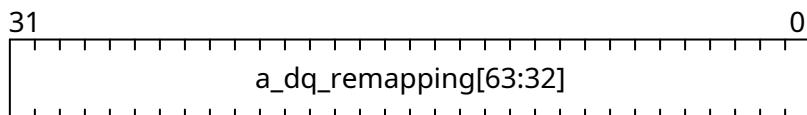


Fig. 24.174: MAIN_A_DQ_REMAPPING1

MAIN_A_DQ_REMAPPING0

Address: $0xf0006800 + 0xc = 0xf000680c$

Bits 0-31 of *MAIN_A_DQ_REMAPPING*.

MAIN_B_DQ_REMAPPING3

Address: $0xf0006800 + 0x10 = 0xf0006810$

Bits 96-127 of *MAIN_B_DQ_REMAPPING*.

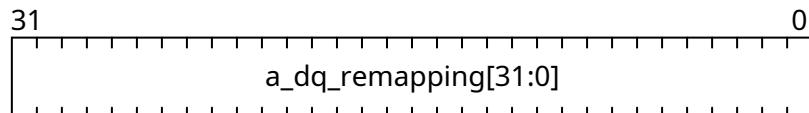


Fig. 24.175: MAIN_A_DQ_REMAPPING0

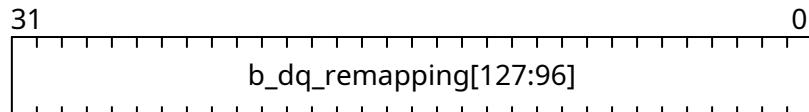


Fig. 24.176: MAIN_B_DQ_REMAPPING3

MAIN_B_DQ_REMAPPING2

Address: $0xf0006800 + 0x14 = 0xf0006814$

Bits 64-95 of *MAIN_B_DQ_REMAPPING*.

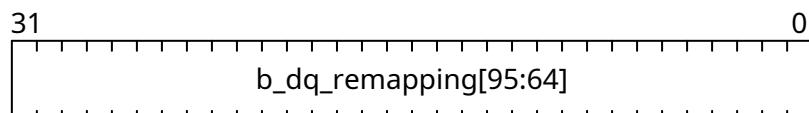


Fig. 24.177: MAIN_B_DQ_REMAPPING2

MAIN_B_DQ_REMAPPING1

Address: $0xf0006800 + 0x18 = 0xf0006818$

Bits 32-63 of *MAIN_B_DQ_REMAPPING*.

MAIN_B_DQ_REMAPPING0

Address: $0xf0006800 + 0x1c = 0xf000681c$

Bits 0-31 of *MAIN_B_DQ_REMAPPING*.

24.2.15 SDRAM

Register Listing for SDRAM

Register	Address
<i>SDRAM_DFII_CONTROL</i>	<i>0xf0007000</i>
<i>SDRAM_DFII_FORCE_ISSUE</i>	<i>0xf0007004</i>
<i>SDRAM_DFII_A_CMDINJECTOR_COMMAND_STORAGE</i>	<i>0xf0007008</i>
<i>SDRAM_DFII_A_CMDINJECTOR_COMMAND_STORAGE_WR_MASK</i>	<i>0xf000700c</i>
<i>SDRAM_DFII_A_CMDINJECTOR_PHASE_ADDR</i>	<i>0xf0007010</i>

continues on next page

Table 24.3 – continued from previous page

Register	Address
<i>SDRAM_DFII_A_CMDINJECTOR_STORE_CONTINUOUS_CMD</i>	0xf0007014
<i>SDRAM_DFII_A_CMDINJECTOR_STORE_SINGLESHTO_CMD</i>	0xf0007018
<i>SDRAM_DFII_A_CMDINJECTOR_SINGLE_SHOT</i>	0xf000701c
<i>SDRAM_DFII_A_CMDINJECTOR_ISSUE_COMMAND</i>	0xf0007020
<i>SDRAM_DFII_A_CMDINJECTOR_WRDATA_SELECT</i>	0xf0007024
<i>SDRAM_DFII_A_CMDINJECTOR_WRDATA</i>	0xf0007028
<i>SDRAM_DFII_A_CMDINJECTOR_WRDATA_S</i>	0xf000702c
<i>SDRAM_DFII_A_CMDINJECTOR_WRDATA_STORE</i>	0xf0007030
<i>SDRAM_DFII_A_CMDINJECTOR_SETUP</i>	0xf0007034
<i>SDRAM_DFII_A_CMDINJECTOR_SAMPLE</i>	0xf0007038
<i>SDRAM_DFII_A_CMDINJECTOR_RESULT_ARRAY</i>	0xf000703c
<i>SDRAM_DFII_A_CMDINJECTOR_RESET</i>	0xf0007040
<i>SDRAM_DFII_A_CMDINJECTOR_RDDATA_SELECT</i>	0xf0007044
<i>SDRAM_DFII_A_CMDINJECTOR_RDDATA_CAPTURE_CNT</i>	0xf0007048
<i>SDRAM_DFII_A_CMDINJECTOR_RDDATA</i>	0xf000704c
<i>SDRAM_DFII_B_CMDINJECTOR_COMMAND_STORAGE</i>	0xf0007050
<i>SDRAM_DFII_B_CMDINJECTOR_COMMAND_STORAGE_WR_MASK</i>	0xf0007054
<i>SDRAM_DFII_B_CMDINJECTOR_PHASE_ADDR</i>	0xf0007058
<i>SDRAM_DFII_B_CMDINJECTOR_STORE_CONTINUOUS_CMD</i>	0xf000705c
<i>SDRAM_DFII_B_CMDINJECTOR_STORE_SINGLESHTO_CMD</i>	0xf0007060
<i>SDRAM_DFII_B_CMDINJECTOR_SINGLE_SHOT</i>	0xf0007064
<i>SDRAM_DFII_B_CMDINJECTOR_ISSUE_COMMAND</i>	0xf0007068
<i>SDRAM_DFII_B_CMDINJECTOR_WRDATA_SELECT</i>	0xf000706c
<i>SDRAM_DFII_B_CMDINJECTOR_WRDATA</i>	0xf0007070
<i>SDRAM_DFII_B_CMDINJECTOR_WRDATA_S</i>	0xf0007074
<i>SDRAM_DFII_B_CMDINJECTOR_WRDATA_STORE</i>	0xf0007078
<i>SDRAM_DFII_B_CMDINJECTOR_SETUP</i>	0xf000707c
<i>SDRAM_DFII_B_CMDINJECTOR_SAMPLE</i>	0xf0007080
<i>SDRAM_DFII_B_CMDINJECTOR_RESULT_ARRAY</i>	0xf0007084
<i>SDRAM_DFII_B_CMDINJECTOR_RESET</i>	0xf0007088
<i>SDRAM_DFII_B_CMDINJECTOR_RDDATA_SELECT</i>	0xf000708c
<i>SDRAM_DFII_B_CMDINJECTOR_RDDATA_CAPTURE_CNT</i>	0xf0007090
<i>SDRAM_DFII_B_CMDINJECTOR_RDDATA</i>	0xf0007094
<i>SDRAM_CONTROLLER_TRP</i>	0xf0007098
<i>SDRAM_CONTROLLER_TRCD</i>	0xf000709c
<i>SDRAM_CONTROLLER_TWR</i>	0xf00070a0
<i>SDRAM_CONTROLLER_TWTR</i>	0xf00070a4
<i>SDRAM_CONTROLLER_TREFI</i>	0xf00070a8
<i>SDRAM_CONTROLLER_TRFC</i>	0xf00070ac
<i>SDRAM_CONTROLLER_TFAW</i>	0xf00070b0
<i>SDRAM_CONTROLLER_TCCD</i>	0xf00070b4
<i>SDRAM_CONTROLLER_TCCD_WR</i>	0xf00070b8
<i>SDRAM_CONTROLLER_TRTP</i>	0xf00070bc
<i>SDRAM_CONTROLLER_TRRD</i>	0xf00070c0
<i>SDRAM_CONTROLLER_TRC</i>	0xf00070c4
<i>SDRAM_CONTROLLER_TRAS</i>	0xf00070c8
<i>SDRAM_CONTROLLER_LAST_ADDR_0</i>	0xf00070cc
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0</i>	0xf00070d0

continues on next page

Table 24.3 – continued from previous page

Register	Address
<i>SDRAM_CONTROLLER_LAST_ADDR_1</i>	0xf00070d4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1</i>	0xf00070d8
<i>SDRAM_CONTROLLER_LAST_ADDR_2</i>	0xf00070dc
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2</i>	0xf00070e0
<i>SDRAM_CONTROLLER_LAST_ADDR_3</i>	0xf00070e4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3</i>	0xf00070e8
<i>SDRAM_CONTROLLER_LAST_ADDR_4</i>	0xf00070ec
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4</i>	0xf00070f0
<i>SDRAM_CONTROLLER_LAST_ADDR_5</i>	0xf00070f4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5</i>	0xf00070f8
<i>SDRAM_CONTROLLER_LAST_ADDR_6</i>	0xf00070fc
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6</i>	0xf0007100
<i>SDRAM_CONTROLLER_LAST_ADDR_7</i>	0xf0007104
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7</i>	0xf0007108
<i>SDRAM_CONTROLLER_LAST_ADDR_8</i>	0xf000710c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_8</i>	0xf0007110
<i>SDRAM_CONTROLLER_LAST_ADDR_9</i>	0xf0007114
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_9</i>	0xf0007118
<i>SDRAM_CONTROLLER_LAST_ADDR_10</i>	0xf000711c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_10</i>	0xf0007120
<i>SDRAM_CONTROLLER_LAST_ADDR_11</i>	0xf0007124
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_11</i>	0xf0007128
<i>SDRAM_CONTROLLER_LAST_ADDR_12</i>	0xf000712c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_12</i>	0xf0007130
<i>SDRAM_CONTROLLER_LAST_ADDR_13</i>	0xf0007134
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_13</i>	0xf0007138
<i>SDRAM_CONTROLLER_LAST_ADDR_14</i>	0xf000713c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_14</i>	0xf0007140
<i>SDRAM_CONTROLLER_LAST_ADDR_15</i>	0xf0007144
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_15</i>	0xf0007148
<i>SDRAM_CONTROLLER_LAST_ADDR_16</i>	0xf000714c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_16</i>	0xf0007150
<i>SDRAM_CONTROLLER_LAST_ADDR_17</i>	0xf0007154
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_17</i>	0xf0007158
<i>SDRAM_CONTROLLER_LAST_ADDR_18</i>	0xf000715c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_18</i>	0xf0007160
<i>SDRAM_CONTROLLER_LAST_ADDR_19</i>	0xf0007164
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_19</i>	0xf0007168
<i>SDRAM_CONTROLLER_LAST_ADDR_20</i>	0xf000716c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_20</i>	0xf0007170
<i>SDRAM_CONTROLLER_LAST_ADDR_21</i>	0xf0007174
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_21</i>	0xf0007178
<i>SDRAM_CONTROLLER_LAST_ADDR_22</i>	0xf000717c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_22</i>	0xf0007180
<i>SDRAM_CONTROLLER_LAST_ADDR_23</i>	0xf0007184
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_23</i>	0xf0007188
<i>SDRAM_CONTROLLER_LAST_ADDR_24</i>	0xf000718c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_24</i>	0xf0007190

continues on next page

Table 24.3 – continued from previous page

Register	Address
<i>SDRAM_CONTROLLER_LAST_ADDR_25</i>	0xf0007194
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_25</i>	0xf0007198
<i>SDRAM_CONTROLLER_LAST_ADDR_26</i>	0xf000719c
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_26</i>	0xf00071a0
<i>SDRAM_CONTROLLER_LAST_ADDR_27</i>	0xf00071a4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_27</i>	0xf00071a8
<i>SDRAM_CONTROLLER_LAST_ADDR_28</i>	0xf00071ac
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_28</i>	0xf00071b0
<i>SDRAM_CONTROLLER_LAST_ADDR_29</i>	0xf00071b4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_29</i>	0xf00071b8
<i>SDRAM_CONTROLLER_LAST_ADDR_30</i>	0xf00071bc
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_30</i>	0xf00071c0
<i>SDRAM_CONTROLLER_LAST_ADDR_31</i>	0xf00071c4
<i>SDRAM_CONTROLLER_LAST_ACTIVE_ROW_31</i>	0xf00071c8

SDRAM_DFII_CONTROL

Address: $0xf0007000 + 0x0 = 0xf0007000$

Control DFI signals common to all phases

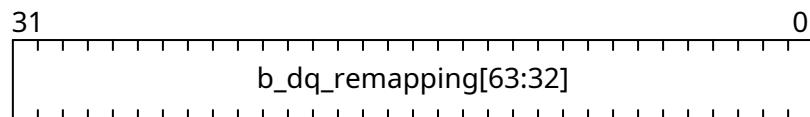


Fig. 24.178: MAIN_B_DQ_REMAPPING1

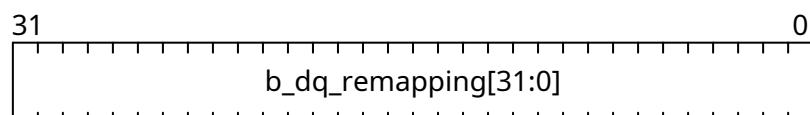


Fig. 24.179: MAIN_B_DQ_REMAPPING0

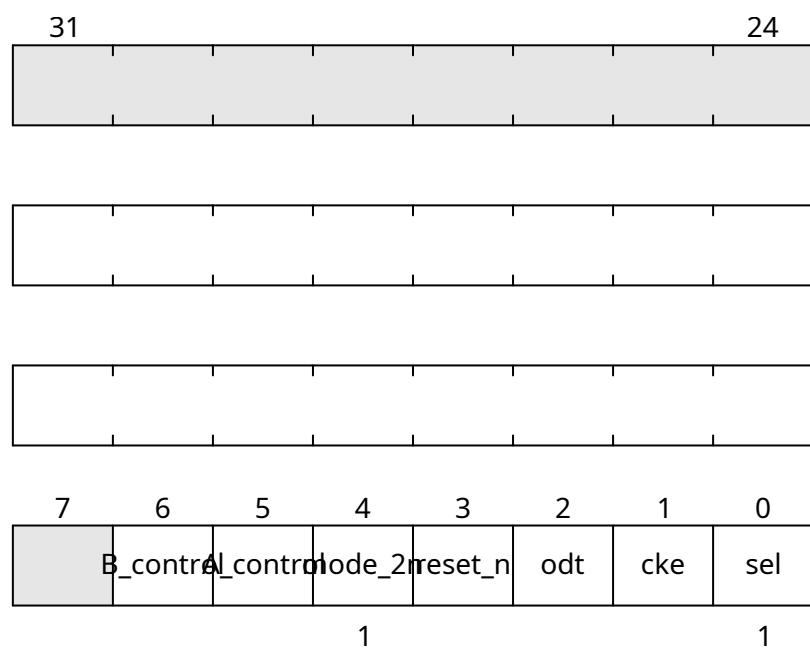


Fig. 24.180: SDRAM_DFII_CONTROL

Field	Name	Description						
[0]	SEL	<table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0b0</td><td>Software (CPU) control.</td></tr> <tr> <td>0b1</td><td>Hardware control (default).</td></tr> </tbody> </table>	Value	Description	0b0	Software (CPU) control.	0b1	Hardware control (default).
Value	Description							
0b0	Software (CPU) control.							
0b1	Hardware control (default).							
[1]	CKE	DFI clock enable bus						
[2]	ODT	DFI on-die termination bus						
[3]	RESET_N	DFI clock reset bus						
[4]	MODE_2N	<table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0b0</td><td>In 1N mode</td></tr> <tr> <td>0b1</td><td>In 2N mode (Default)</td></tr> </tbody> </table>	Value	Description	0b0	In 1N mode	0b1	In 2N mode (Default)
Value	Description							
0b0	In 1N mode							
0b1	In 2N mode (Default)							
[5]	A_CONTROL	<table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0b1</td><td>A_Cmd Injector</td></tr> </tbody> </table>	Value	Description	0b1	A_Cmd Injector		
Value	Description							
0b1	A_Cmd Injector							
[6]	B_CONTROL	<table border="1"> <thead> <tr> <th>Value</th><th>Description</th></tr> </thead> <tbody> <tr> <td>0b1</td><td>B_Cmd Injector</td></tr> </tbody> </table>	Value	Description	0b1	B_Cmd Injector		
Value	Description							
0b1	B_Cmd Injector							

SDRAM_DFII_FORCE_ISSUE

Address: $0xf0007000 + 0x4 = 0xf0007004$

SDRAM_DFII_A_CMDINJECTOR_COMMAND_STORAGE

Address: $0xf0007000 + 0x8 = 0xf0007008$

DDR5 command and control signals

Field	Name	Description
[13:0]	CA	Command/Address bus
[15:14]	CS	DFI chip select bus

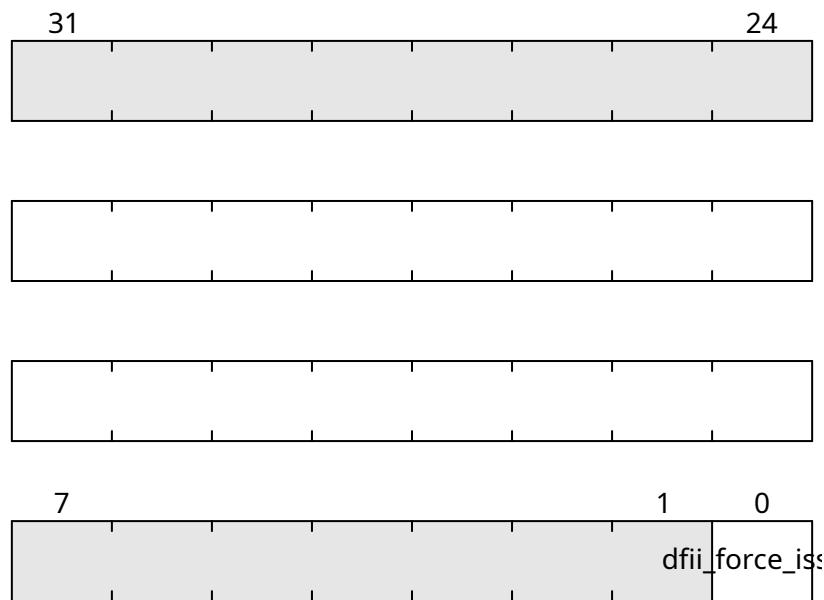


Fig. 24.181: `SDRAM_DFII_FORCE_ISSUE`

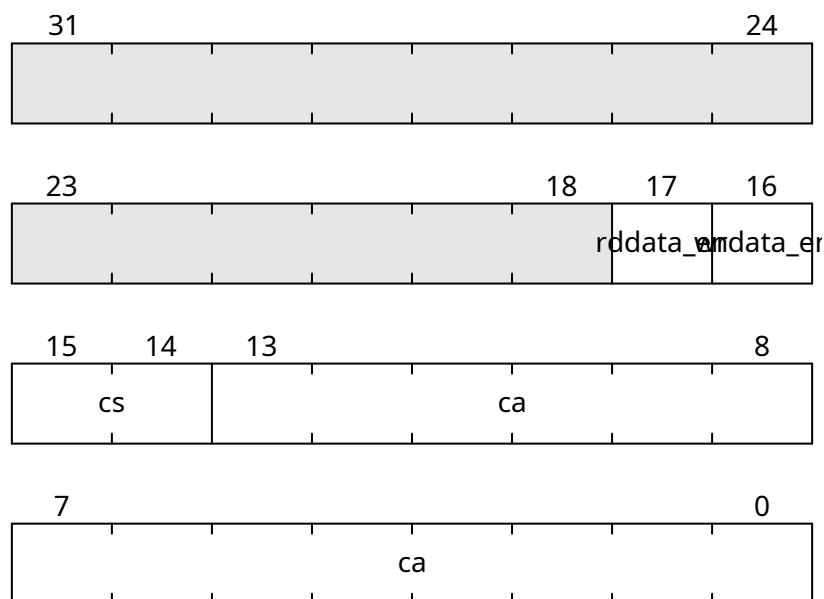


Fig. 24.182: `SDRAM_DFII_A_CMDINJECTOR_COMMAND_STORAGE`

SDRAM_DFII_A_CMDINJECTOR_COMMAND_STORAGE_WR_MASK

Address: $0xf0007000 + 0xc = 0xf000700c$

DDR5 wrdata mask control signals

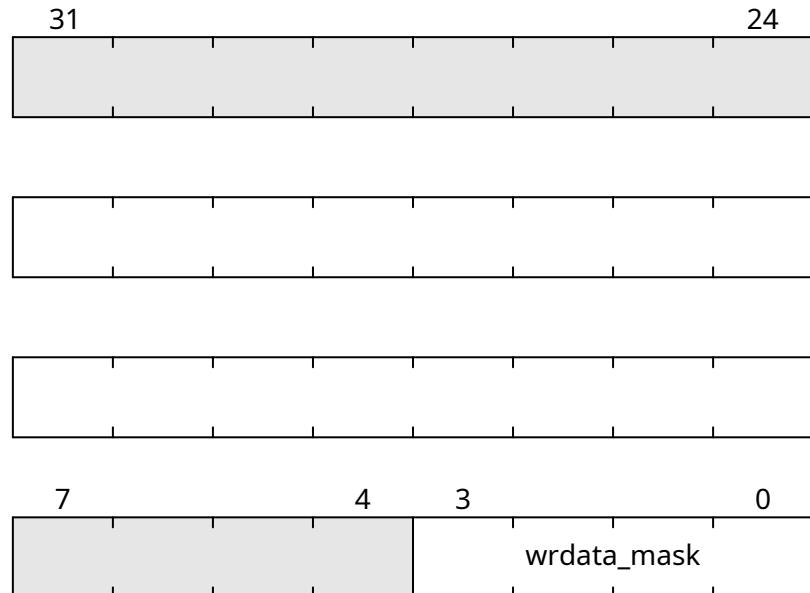


Fig. 24.183: SDRAM_DFII_A_CMDINJECTOR_COMMAND_STORAGE_WR_MASK

Field	Name	Description

SDRAM_DFII_A_CMDINJECTOR_PHASE_ADDR

Address: $0xf0007000 + 0x10 = 0xf0007010$

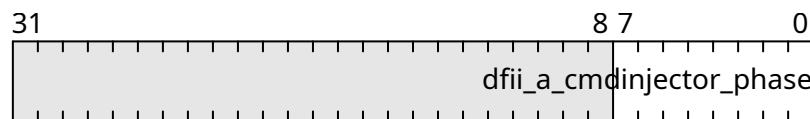


Fig. 24.184: SDRAM_DFII_A_CMDINJECTOR_PHASE_ADDR

SDRAM_DFII_A_CMDINJECTOR_STORE_CONTINUOUS_CMD

Address: $0xf0007000 + 0x14 = 0xf0007014$

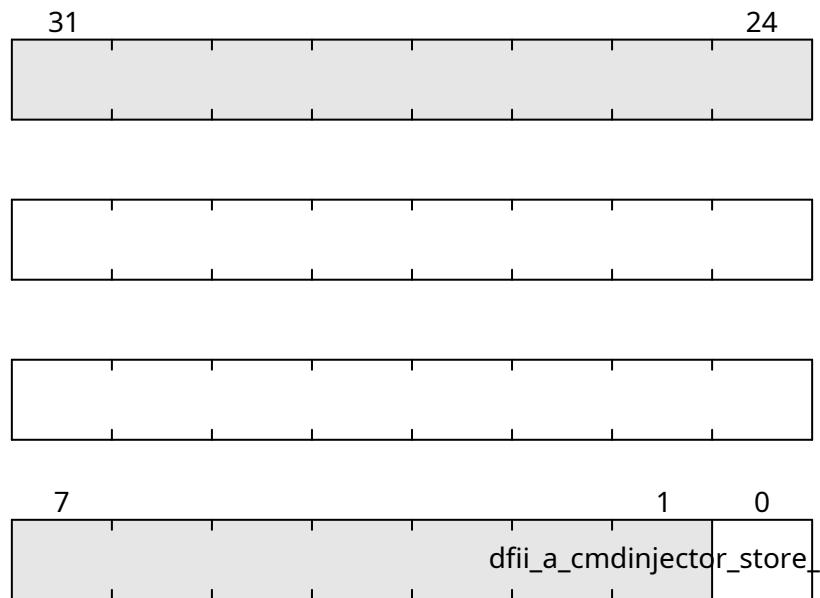


Fig. 24.185: SDRAM_DFII_A_CMDINJECTOR_STORE_CONTINUOUS_CMD

SDRAM_DFII_A_CMDINJECTOR_STORE_SINGLESHOT_CMD

Address: $0xf0007000 + 0x18 = 0xf0007018$

SDRAM_DFII_A_CMDINJECTOR_SINGLE_SHOT

Address: $0xf0007000 + 0x1c = 0xf000701c$

SDRAM_DFII_A_CMDINJECTOR_ISSUE_COMMAND

Address: $0xf0007000 + 0x20 = 0xf0007020$

SDRAM_DFII_A_CMDINJECTOR_WRDATA_SELECT

Address: $0xf0007000 + 0x24 = 0xf0007024$

SDRAM_DFII_A_CMDINJECTOR_WRDATA

Address: $0xf0007000 + 0x28 = 0xf0007028$

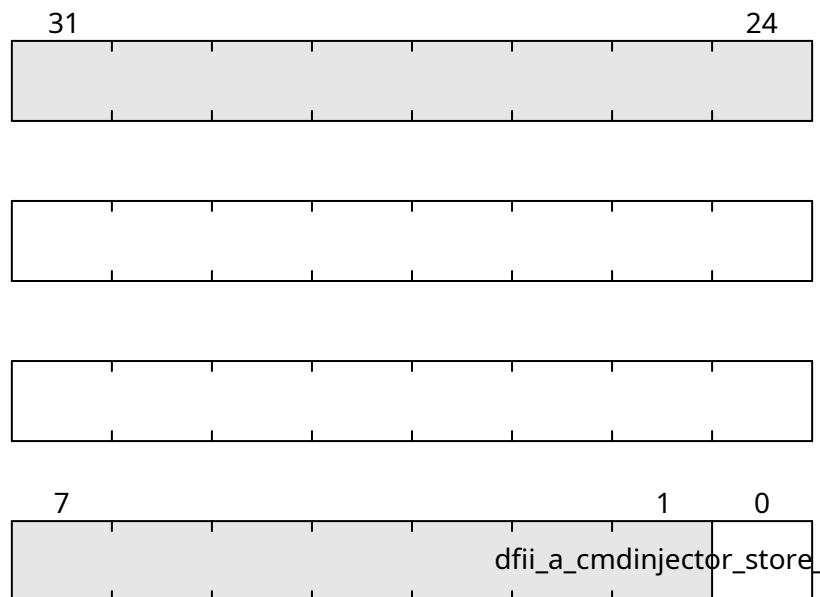


Fig. 24.186: SDRAM_DFII_A_CMDINJECTOR_STORE_SINGLESHT_CMD

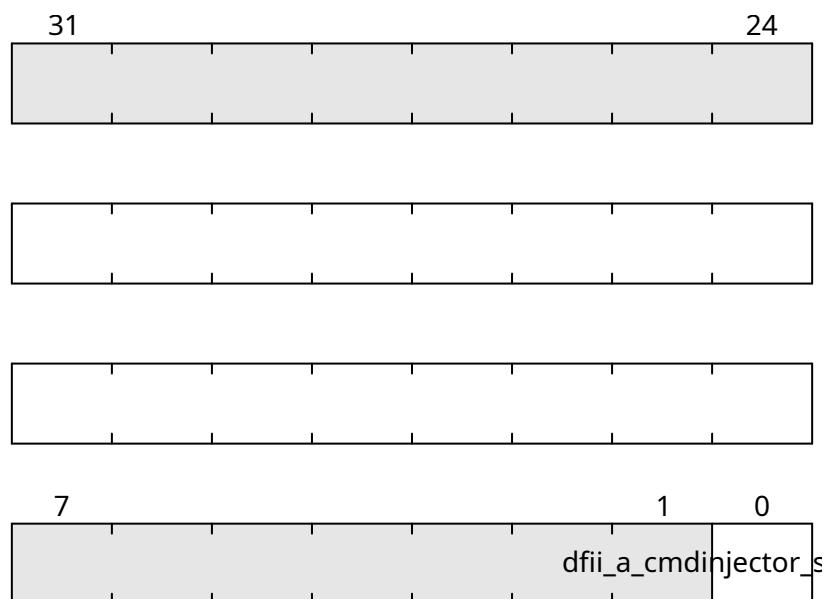


Fig. 24.187: SDRAM_DFII_A_CMDINJECTOR_SINGLE_SHOT

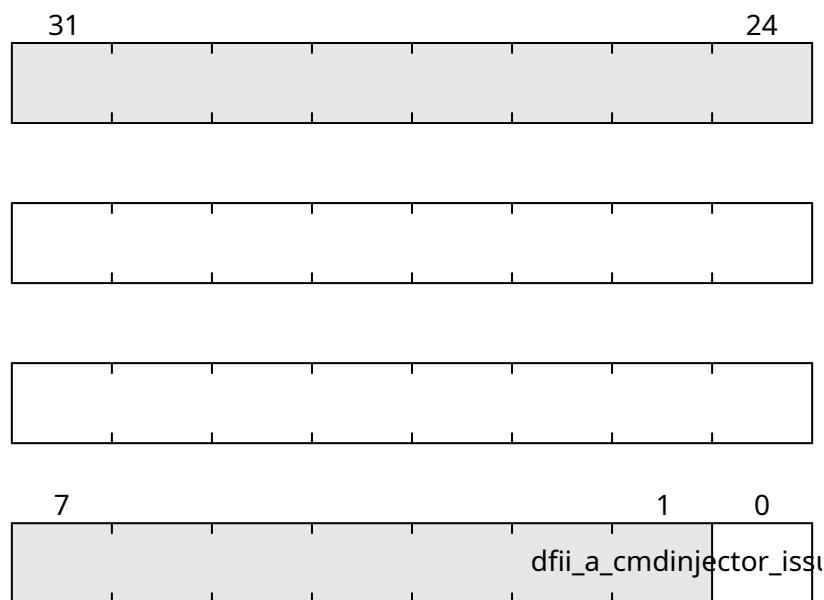


Fig. 24.188: SDRAM_DFII_A_CMDINJECTOR_ISSUE_COMMAND

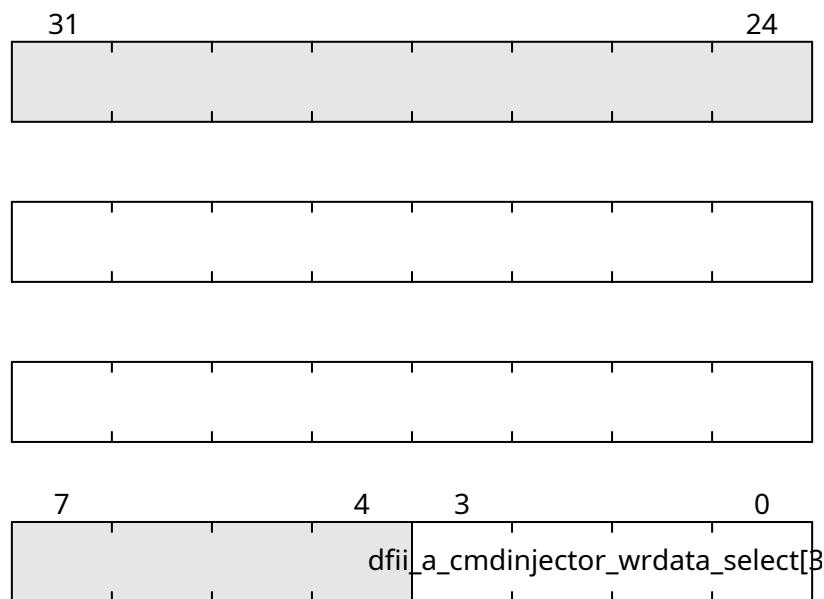


Fig. 24.189: SDRAM_DFII_A_CMDINJECTOR_WRDATA_SELECT

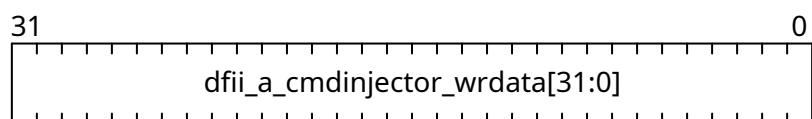


Fig. 24.190: SDRAM_DFII_A_CMDINJECTOR_WRDATA

SDRAM_DFII_A_CMDINJECTOR_WRDATA_S

Address: $0xf0007000 + 0x2c = 0xf000702c$

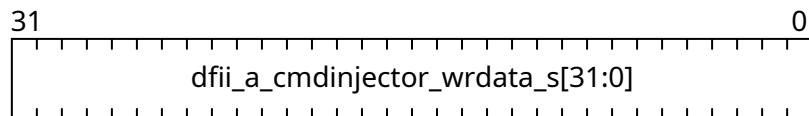


Fig. 24.191: SDRAM_DFII_A_CMDINJECTOR_WRDATA_S

SDRAM_DFII_A_CMDINJECTOR_WRDATA_STORE

Address: $0xf0007000 + 0x30 = 0xf0007030$

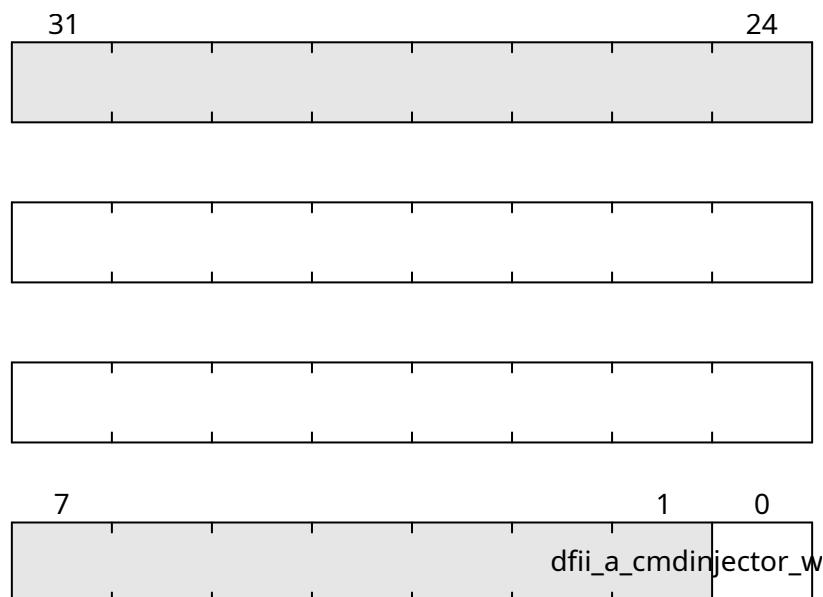


Fig. 24.192: SDRAM_DFII_A_CMDINJECTOR_WRDATA_STORE

SDRAM_DFII_A_CMDINJECTOR_SETUP

Address: $0xf0007000 + 0x34 = 0xf0007034$

Field	Name	Description
[0]	INITIAL_STATE	Initial value of all bits
[1]	OPERATION	0 - or (default), 1 -and



Fig. 24.193: SDRAM DFII A CMDINJECTOR SETUP

SDRAM_DFII_A_CMDINJECTOR_SAMPLE

Address: $0xf0007000 + 0x38 = 0xf0007038$

SDRAM_DFII_A_CMDINJECTOR_RESULT_ARRAY

Address: $0xf0007000 + 0x3c = 0xf000703c$

SDRAM_DFII_A_CMDINJECTOR_RESET

Address: $0xf0007000 + 0x40 = 0xf0007040$

SDRAM DFII A CMDINJECTOR RDDATA SELECT

Address: $0xf0007000 + 0x44 = 0xf0007044$

SDRAM_DFII_A_CMDINJECTOR_RDDATA_CAPTURE_CNT

Address: $0xf0007000 + 0x48 = 0xf0007048$

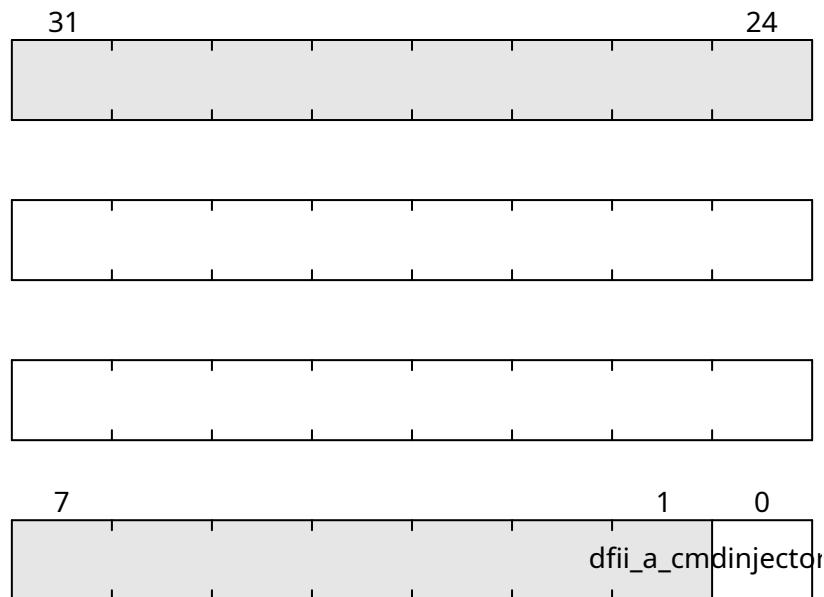


Fig. 24.194: SDRAM_DFII_A_CMDINJECTOR_SAMPLE

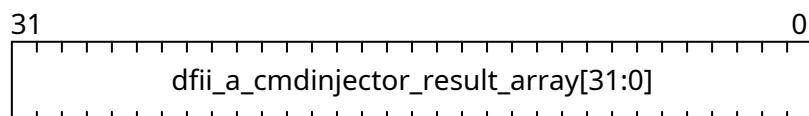


Fig. 24.195: SDRAM_DFII_A_CMDINJECTOR_RESULT_ARRAY

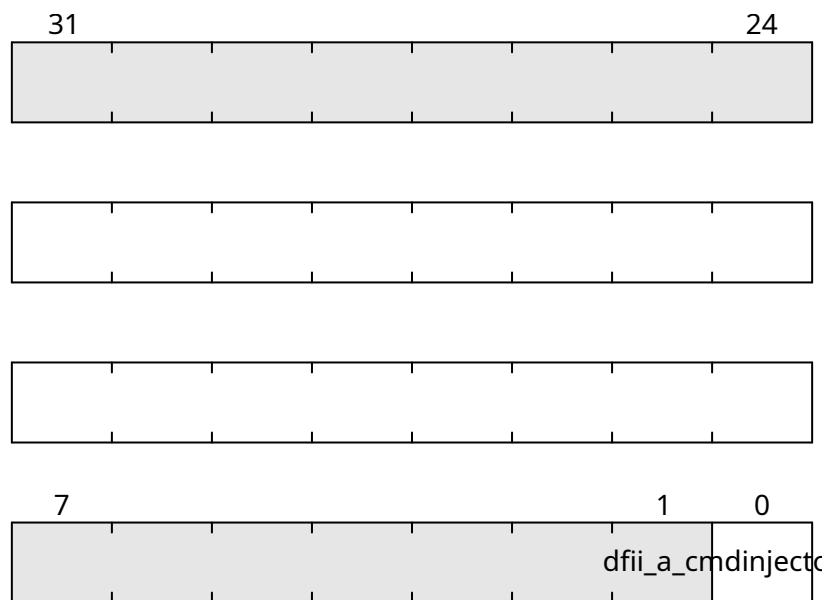


Fig. 24.196: SDRAM_DFII_A_CMDINJECTOR_RESET

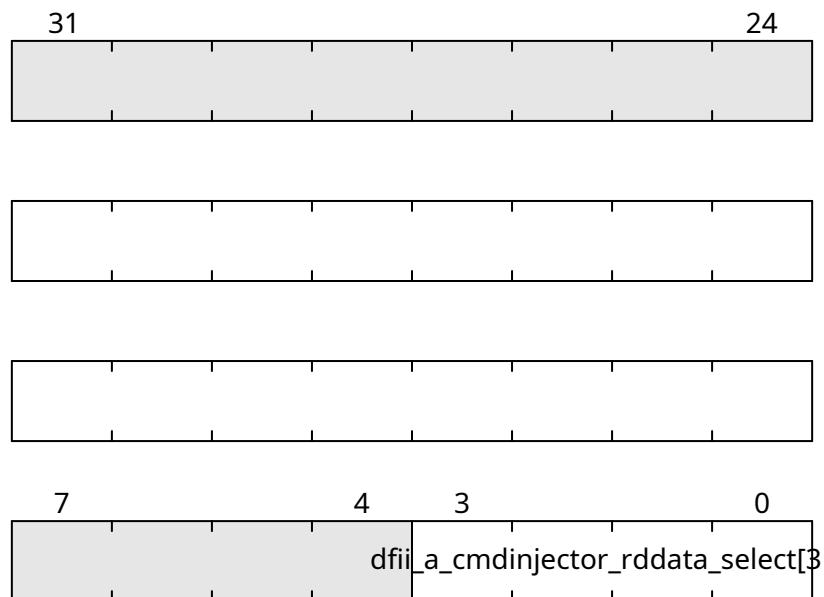


Fig. 24.197: SDRAM_DFII_A_CMDINJECTOR_RDDATA_SELECT

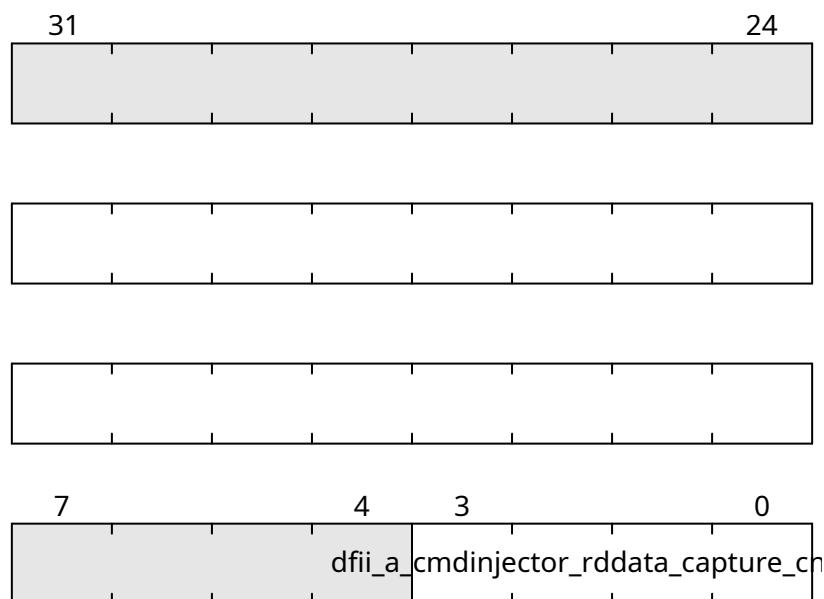


Fig. 24.198: SDRAM_DFII_A_CMDINJECTOR_RDDATA_CAPTURE_CNT

SDRAM_DFII_A_CMDINJECTOR_RDDATA

Address: $0xf0007000 + 0x4c = 0xf000704c$

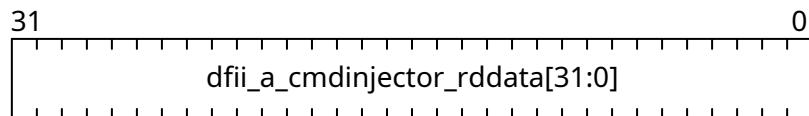


Fig. 24.199: SDRAM_DFII_A_CMDINJECTOR_RDDATA

SDRAM_DFII_B_CMDINJECTOR_COMMAND_STORAGE

Address: $0xf0007000 + 0x50 = 0xf0007050$

DDR5 command and control signals

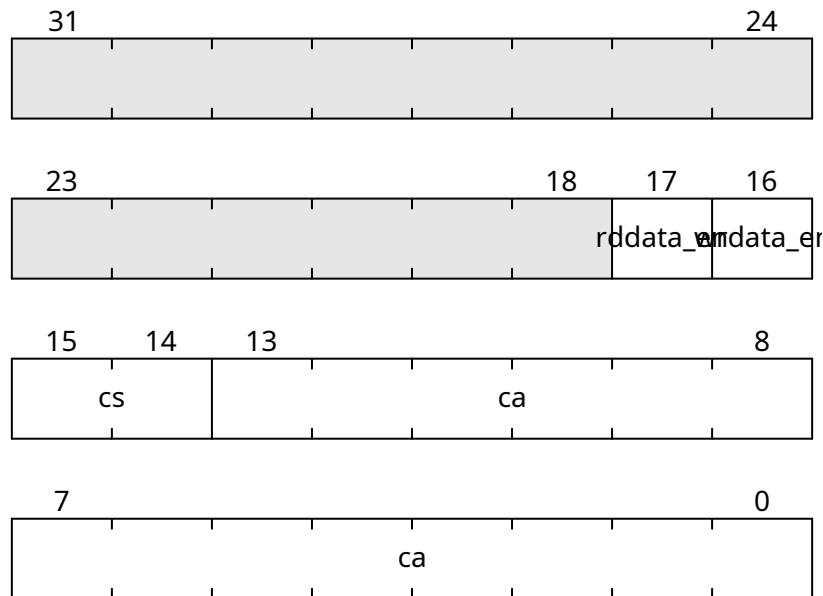


Fig. 24.200: SDRAM_DFII_B_CMDINJECTOR_COMMAND_STORAGE

Field	Name	Description
[13:0]	CA	Command/Address bus
[15:14]	CS	DFI chip select bus

SDRAM_DFII_B_CMDINJECTOR_COMMAND_STORAGE_WR_MASK

Address: $0xf0007000 + 0x54 = 0xf0007054$

DDR5 wrdata mask control signals

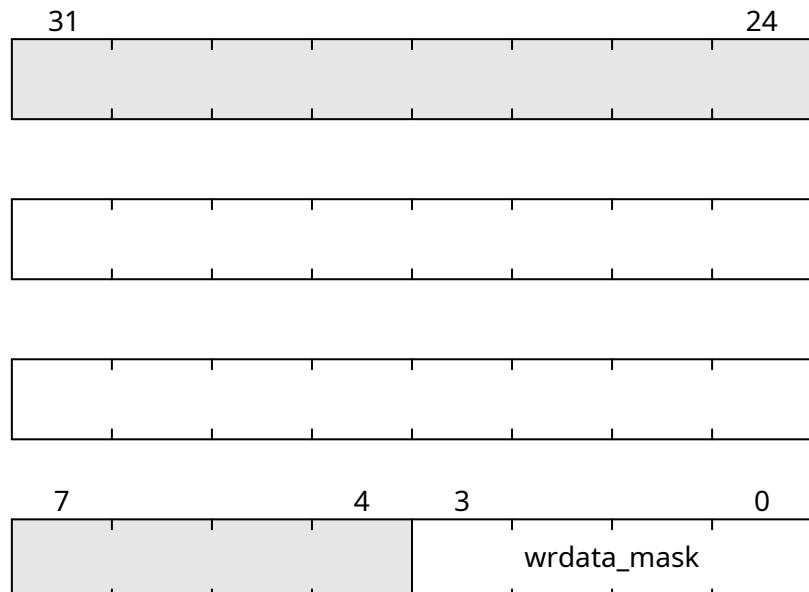


Fig. 24.201: SDRAM_DFII_B_CMDINJECTOR_COMMAND_STORAGE_WR_MASK

Field	Name	Description

SDRAM_DFII_B_CMDINJECTOR_PHASE_ADDR

Address: $0xf0007000 + 0x58 = 0xf0007058$

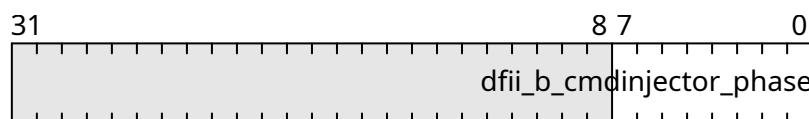


Fig. 24.202: SDRAM_DFII_B_CMDINJECTOR_PHASE_ADDR

SDRAM_DFII_B_CMDINJECTOR_STORE_CONTINUOUS_CMD

Address: $0xf0007000 + 0x5c = 0xf000705c$

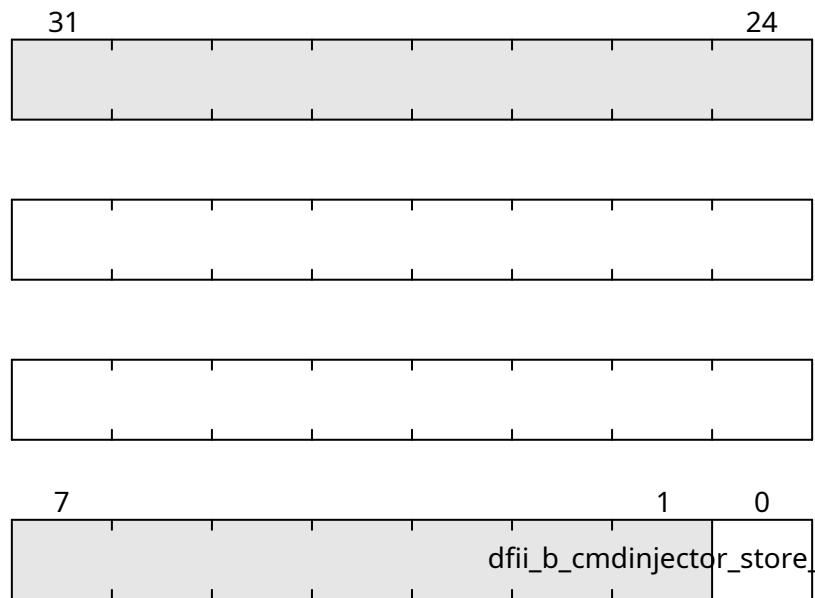


Fig. 24.203: SDRAM_DFII_B_CMDINJECTOR_STORE_CONTINUOUS_CMD

SDRAM_DFII_B_CMDINJECTOR_STORE_SINGLESHOT_CMD

Address: $0xf0007000 + 0x60 = 0xf0007060$

SDRAM_DFII_B_CMDINJECTOR_SINGLE_SHOT

Address: $0xf0007000 + 0x64 = 0xf0007064$

SDRAM_DFII_B_CMDINJECTOR_ISSUE_COMMAND

Address: $0xf0007000 + 0x68 = 0xf0007068$

SDRAM_DFII_B_CMDINJECTOR_WRDATA_SELECT

Address: $0xf0007000 + 0x6c = 0xf000706c$

SDRAM_DFII_B_CMDINJECTOR_WRDATA

Address: $0xf0007000 + 0x70 = 0xf0007070$

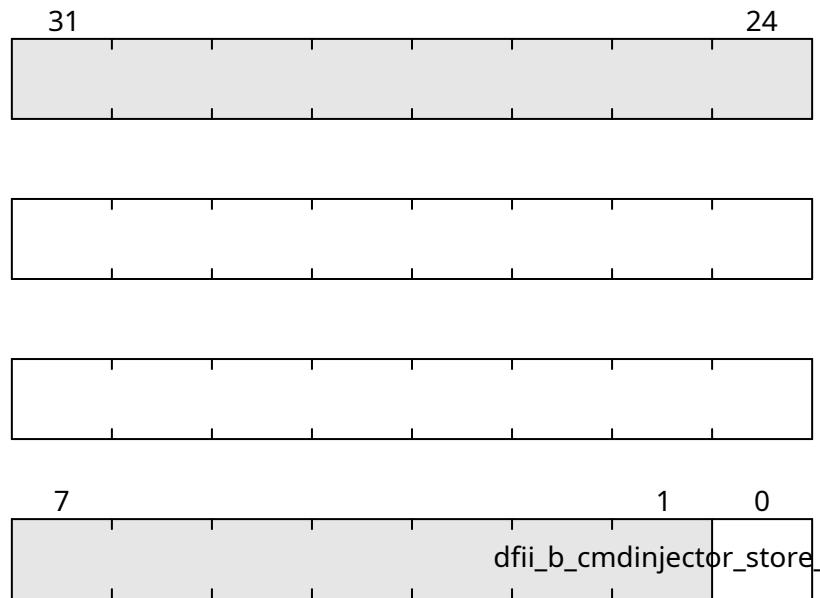


Fig. 24.204: SDRAM_DFII_B_CMDINJECTOR_STORE_SINGLESHTOT_CMD

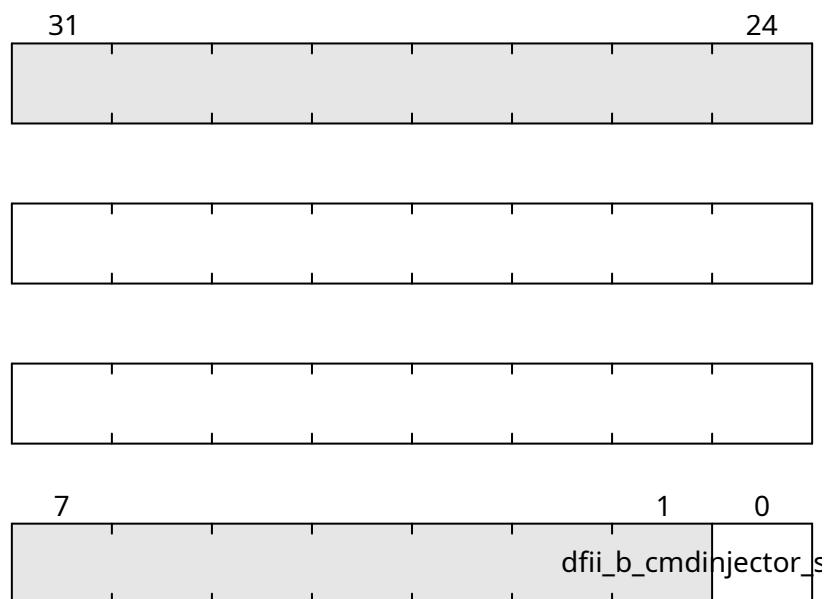


Fig. 24.205: SDRAM_DFII_B_CMDINJECTOR_SINGLE_SHOT



Fig. 24.206: SDRAM_DFII_B_CMDINJECTOR_ISSUE_COMMAND

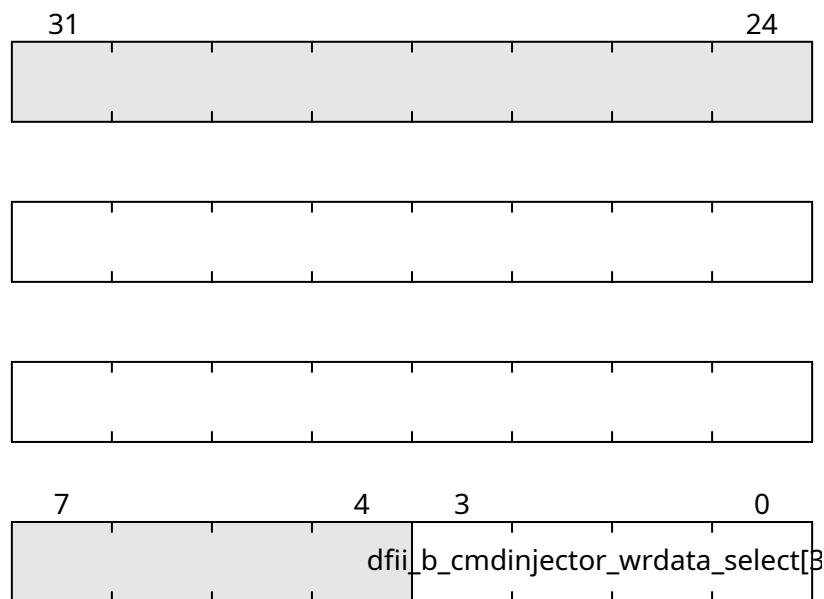


Fig. 24.207: SDRAM_DFII_B_CMDINJECTOR_WRDATA_SELECT

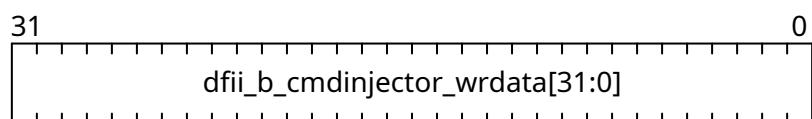


Fig. 24.208: SDRAM_DFII_B_CMDINJECTOR_WRDATA

SDRAM_DFII_B_CMDINJECTOR_WRDATA_S

Address: $0xf0007000 + 0x74 = 0xf0007074$

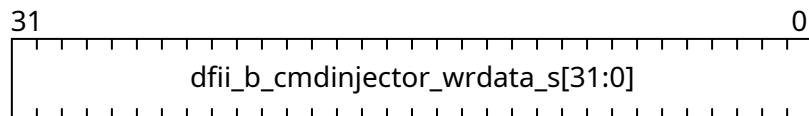


Fig. 24.209: SDRAM_DFII_B_CMDINJECTOR_WRDATA_S

SDRAM_DFII_B_CMDINJECTOR_WRDATA_STORE

Address: $0xf0007000 + 0x78 = 0xf0007078$

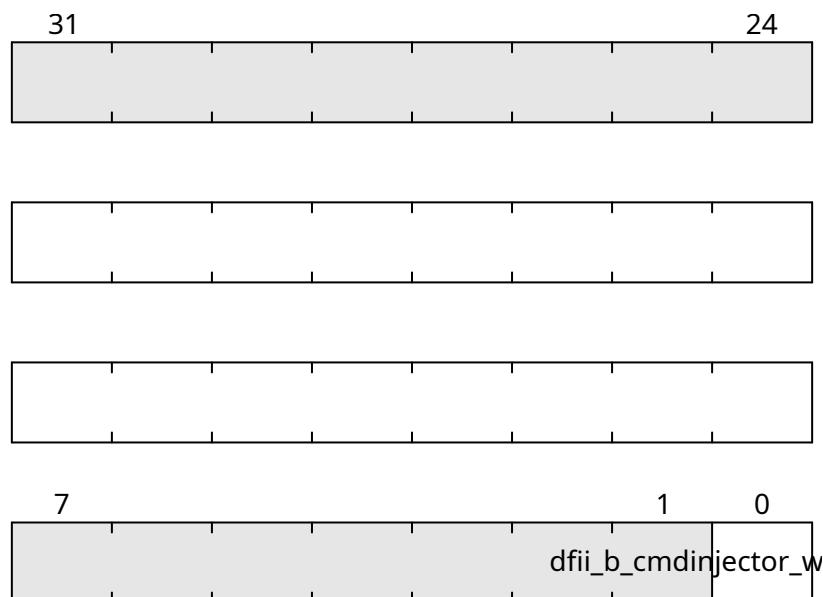


Fig. 24.210: SDRAM_DFII_B_CMDINJECTOR_WRDATA_STORE

SDRAM_DFII_B_CMDINJECTOR_SETUP

Address: $0xf0007000 + 0x7c = 0xf000707c$

Field	Name	Description
[0]	INITIAL_STATE	Initial value of all bits
[1]	OPERATION	0 - or (default), 1 -and



Fig. 24.211: SDRAM_DFII_B_CMDINJECTOR_SETUP

SDRAM_DFII_B_CMDINJECTOR_SAMPLE

Address: $0xf0007000 + 0x80 = 0xf0007080$

SDRAM_DFII_B_CMDINJECTOR_RESULT_ARRAY

Address: $0xf0007000 + 0x84 = 0xf0007084$

SDRAM_DFII_B_CMDINJECTOR_RESET

Address: $0xf0007000 + 0x88 = 0xf0007088$

SDRAM_DFII_B_CMDINJECTOR_RDDATA_SELECT

Address: $0xf0007000 + 0x8c = 0xf000708c$

SDRAM_DFII_B_CMDINJECTOR_RDDATA_CAPTURE_CNT

Address: $0xf0007000 + 0x90 = 0xf0007090$

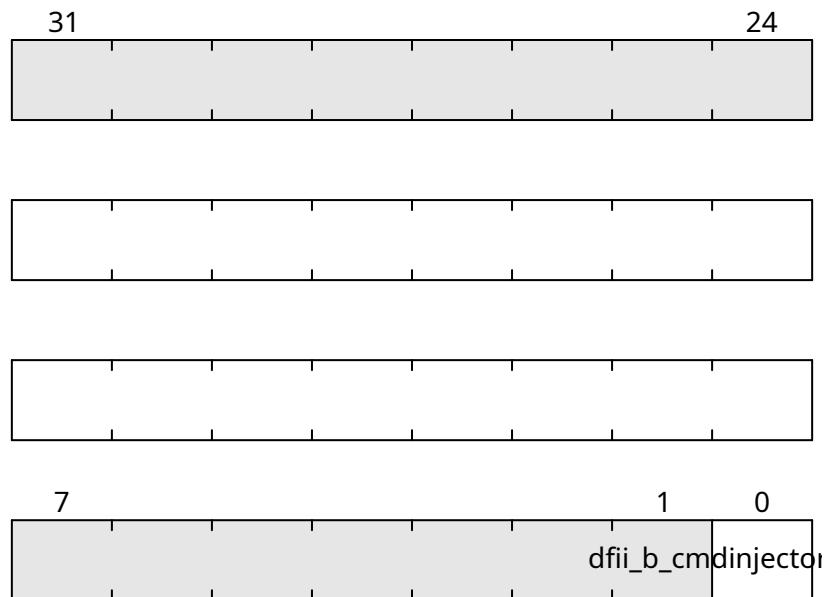


Fig. 24.212: SDRAM_DFII_B_CMDINJECTOR_SAMPLE

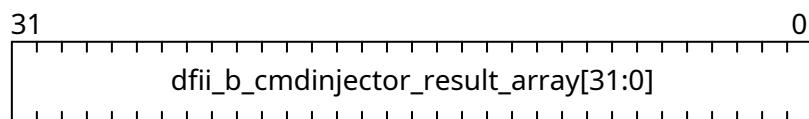


Fig. 24.213: SDRAM_DFII_B_CMDINJECTOR_RESULT_ARRAY

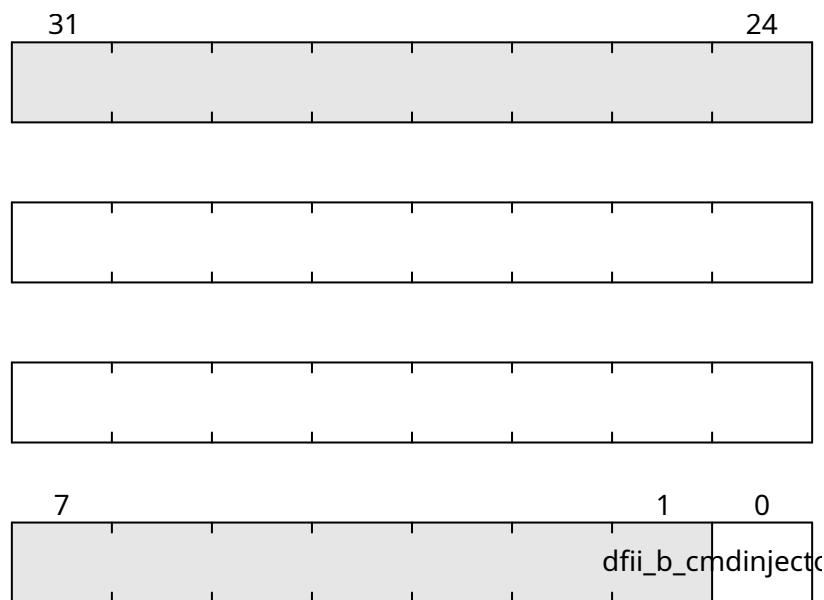


Fig. 24.214: SDRAM_DFII_B_CMDINJECTOR_RESET

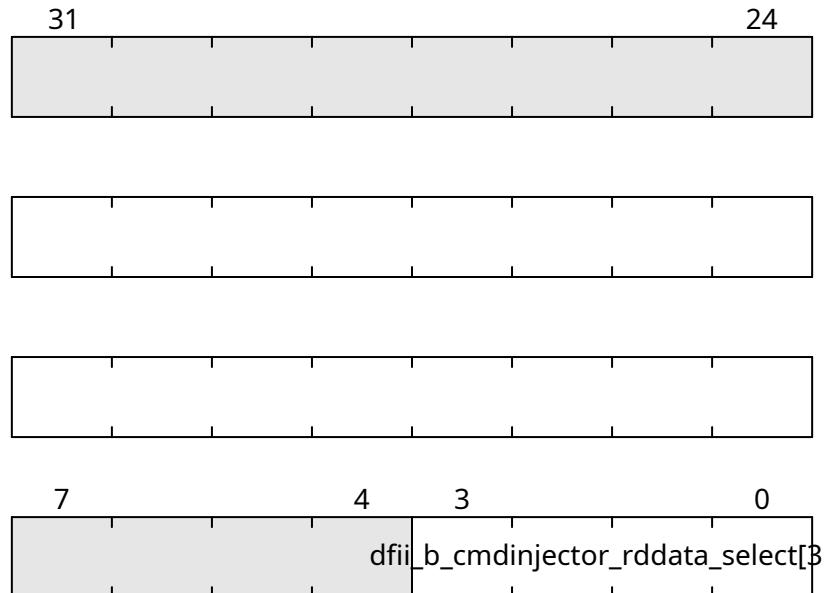


Fig. 24.215: SDRAM_DFII_B_CMDINJECTOR_RDDATA_SELECT

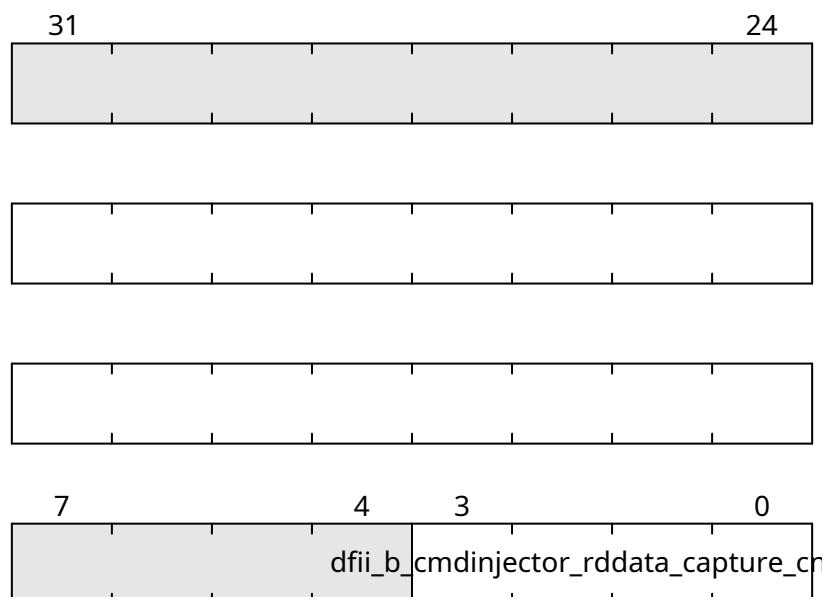


Fig. 24.216: SDRAM_DFII_B_CMDINJECTOR_RDDATA_CAPTURE_CNT

SDRAM_DFII_B_CMDINJECTOR_RDDATA

Address: $0xf0007000 + 0x94 = 0xf0007094$

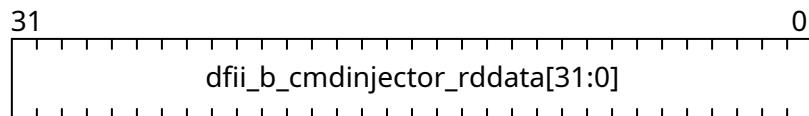


Fig. 24.217: SDRAM_DFII_B_CMDINJECTOR_RDDATA

SDRAM_CONTROLLER_TRP

Address: $0xf0007000 + 0x98 = 0xf0007098$

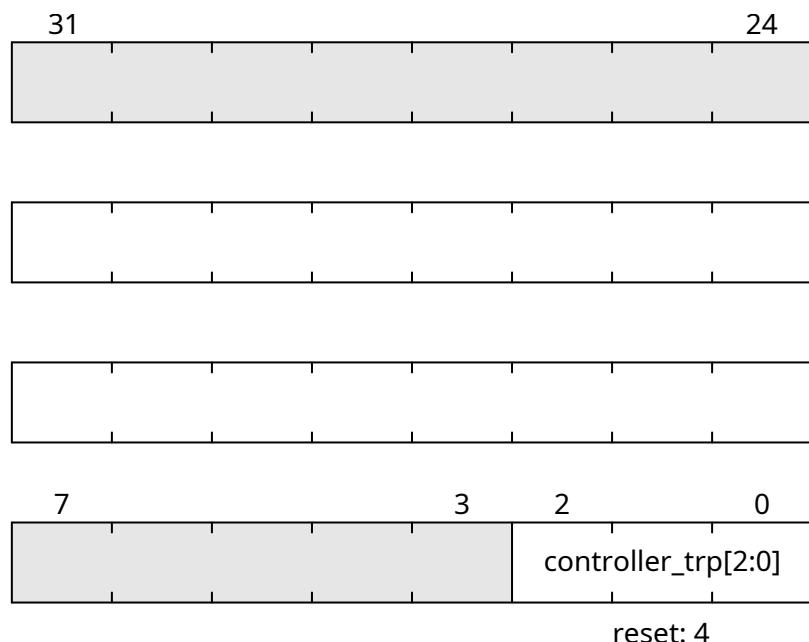


Fig. 24.218: SDRAM_CONTROLLER_TRP

SDRAM_CONTROLLER_TRCD

Address: $0xf0007000 + 0x9c = 0xf000709c$

SDRAM_CONTROLLER_TWR

Address: $0xf0007000 + 0xa0 = 0xf00070a0$

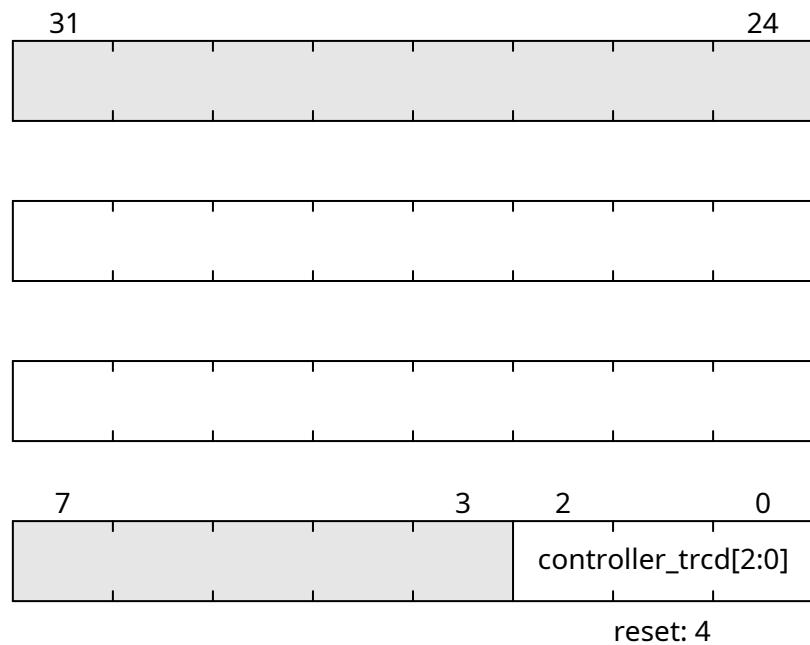


Fig. 24.219: SDRAM_CONTROLLER_TRCD

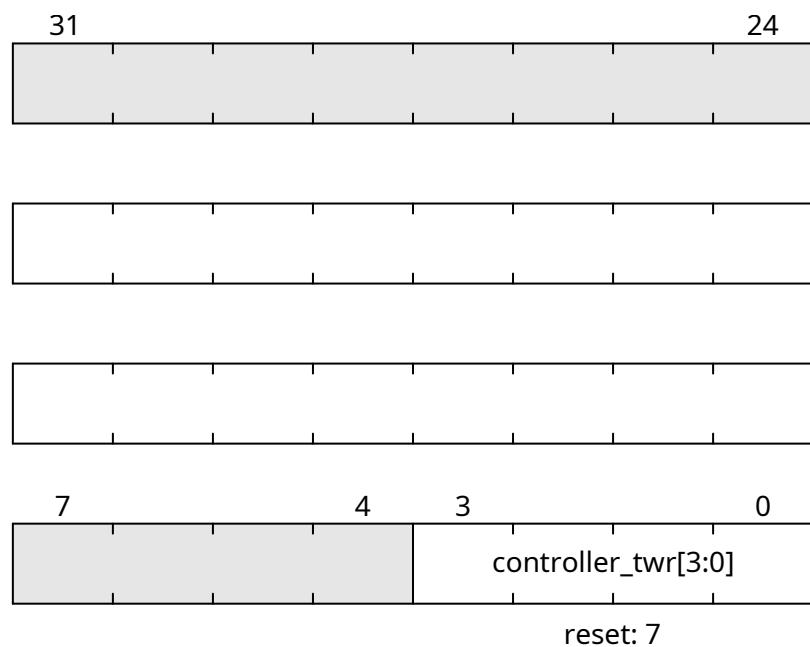


Fig. 24.220: SDRAM_CONTROLLER_TWR

SDRAM_CONTROLLER_TWTR

Address: $0xf0007000 + 0xa4 = 0xf00070a4$

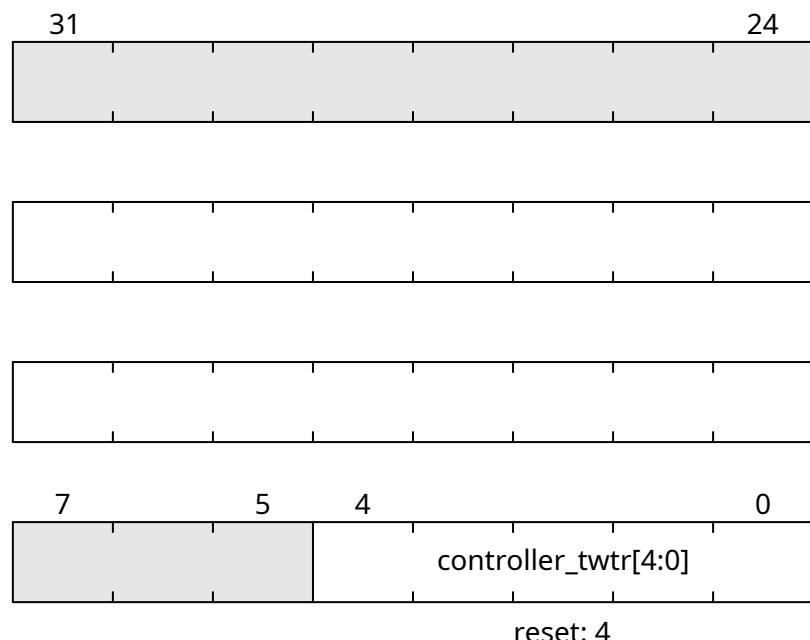


Fig. 24.221: SDRAM_CONTROLLER_TWTR

SDRAM_CONTROLLER_TREFI

Address: $0xf0007000 + 0xa8 = 0xf00070a8$

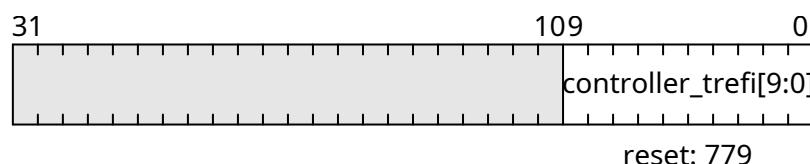


Fig. 24.222: SDRAM_CONTROLLER_TREFI

SDRAM_CONTROLLER_TRFC

Address: $0xf0007000 + 0xac = 0xf00070ac$

SDRAM_CONTROLLER_TFAW

Address: $0xf0007000 + 0xb0 = 0xf00070b0$

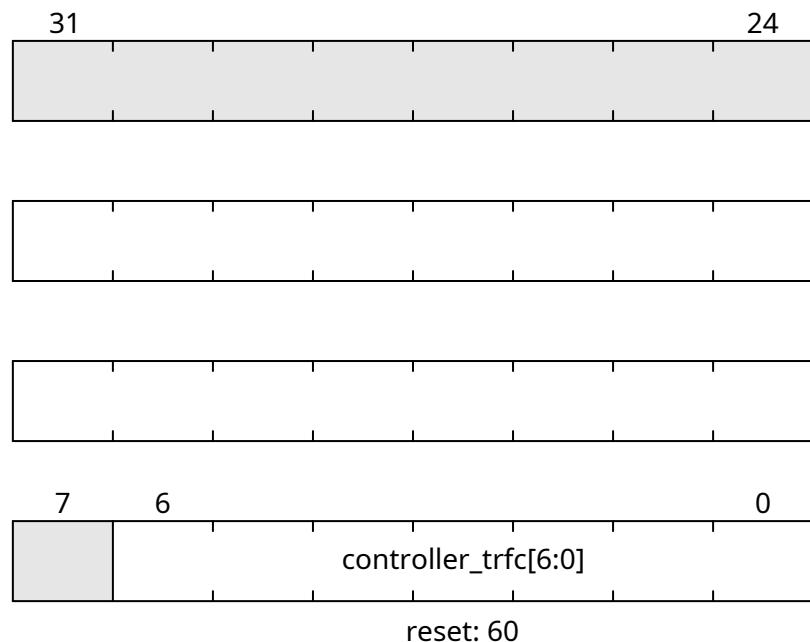


Fig. 24.223: SDRAM_CONTROLLER_TRFC

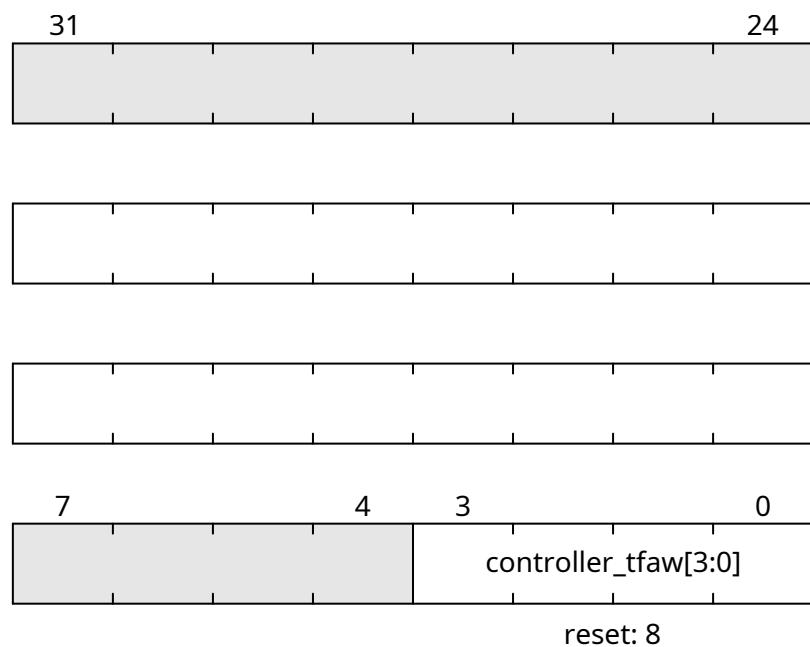


Fig. 24.224: SDRAM_CONTROLLER_TFAW

SDRAM_CONTROLLER_TCCD

Address: $0xf0007000 + 0xb4 = 0xf00070b4$

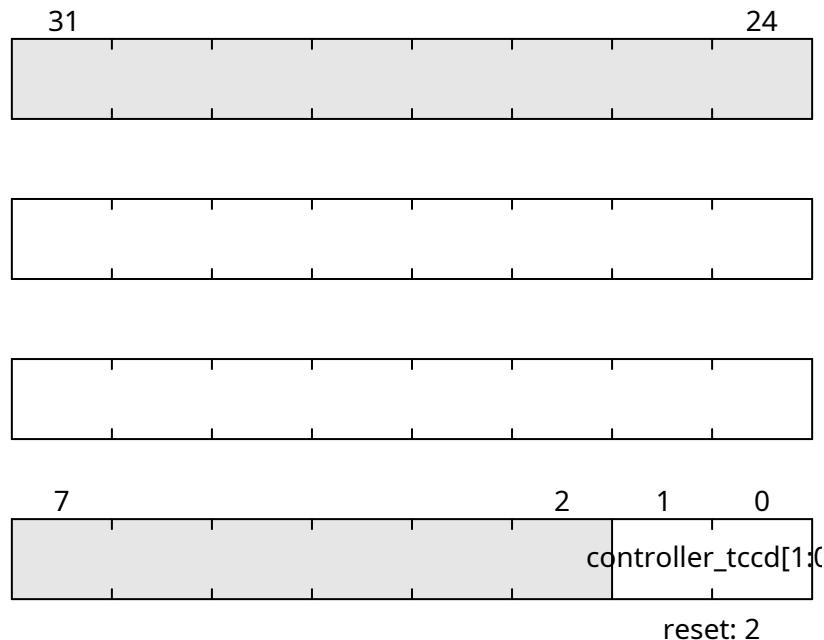


Fig. 24.225: SDRAM_CONTROLLER_TCCD

SDRAM_CONTROLLER_TCCD_WR

Address: $0xf0007000 + 0xb8 = 0xf00070b8$

SDRAM_CONTROLLER_TRTP

Address: $0xf0007000 + 0xbc = 0xf00070bc$

SDRAM_CONTROLLER_TRRD

Address: $0xf0007000 + 0xc0 = 0xf00070c0$

SDRAM_CONTROLLER_TRC

Address: $0xf0007000 + 0xc4 = 0xf00070c4$

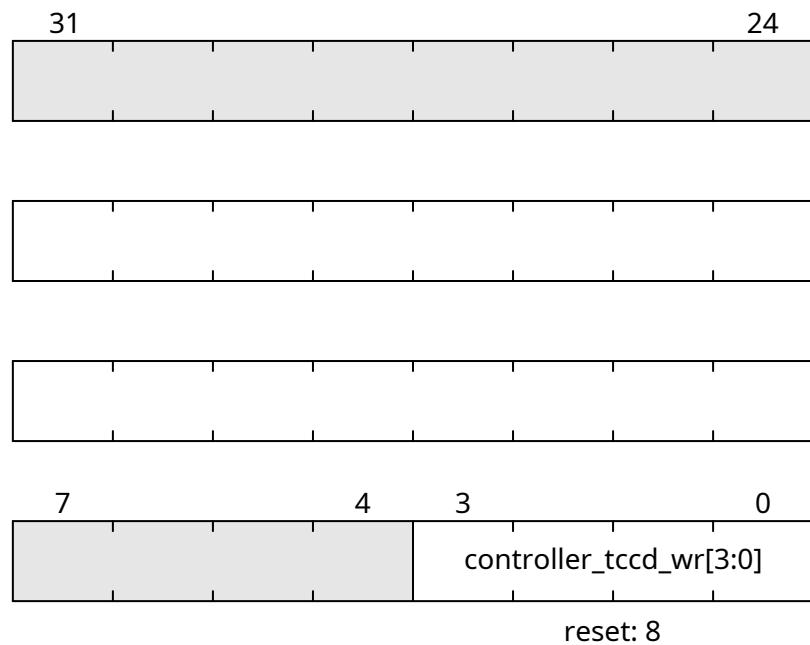


Fig. 24.226: SDRAM_CONTROLLER_TCCD_WR

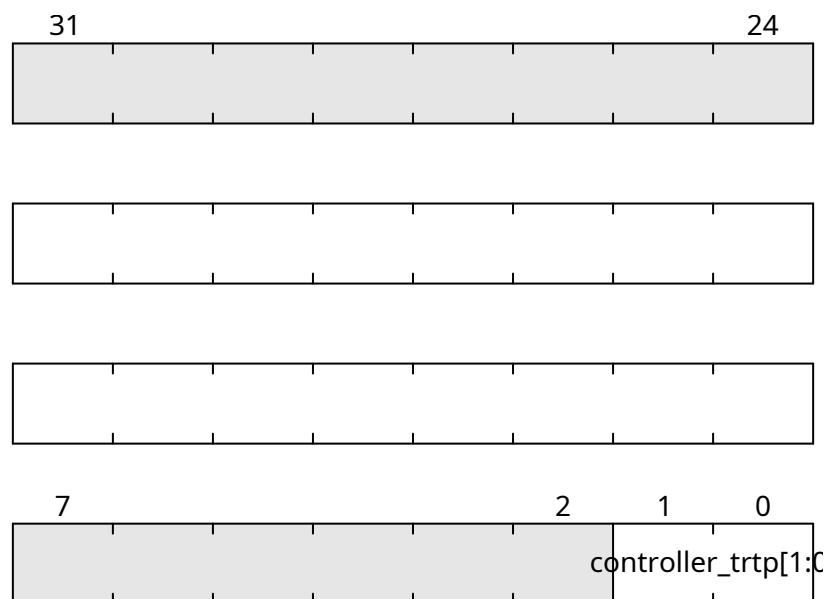


Fig. 24.227: SDRAM_CONTROLLER_TRTP

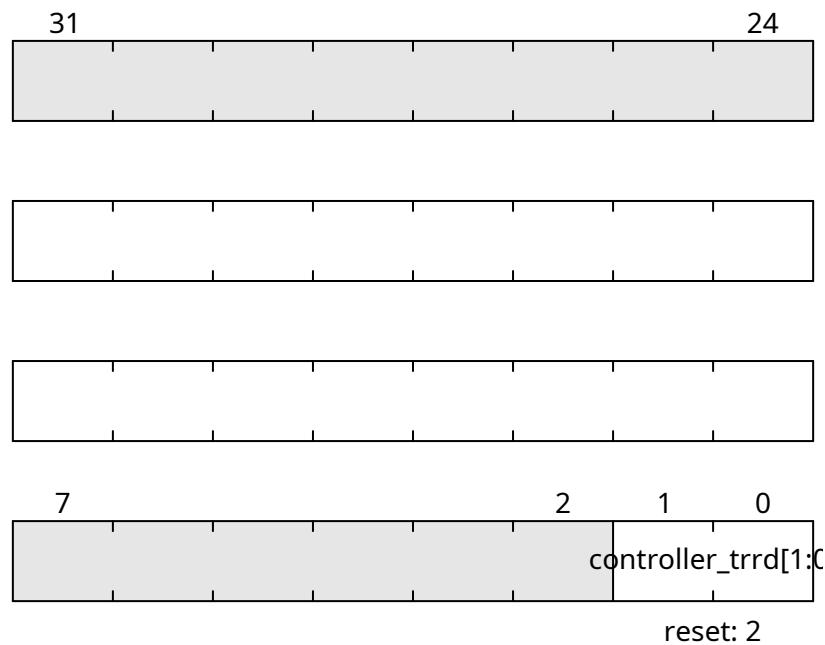


Fig. 24.228: SDRAM_CONTROLLER_TRRD

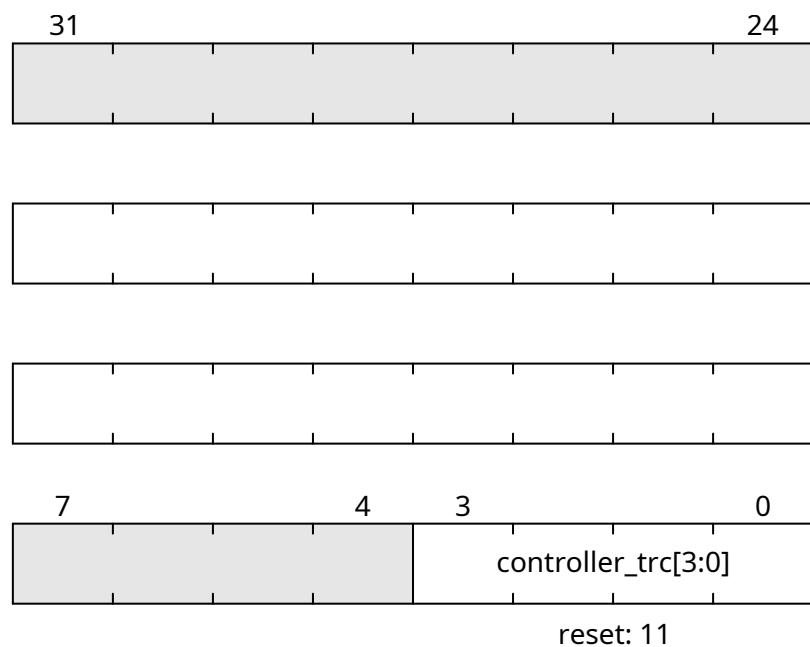


Fig. 24.229: SDRAM_CONTROLLER_TRC

SDRAM_CONTROLLER_TRAS

Address: $0xf0007000 + 0xc8 = 0xf00070c8$

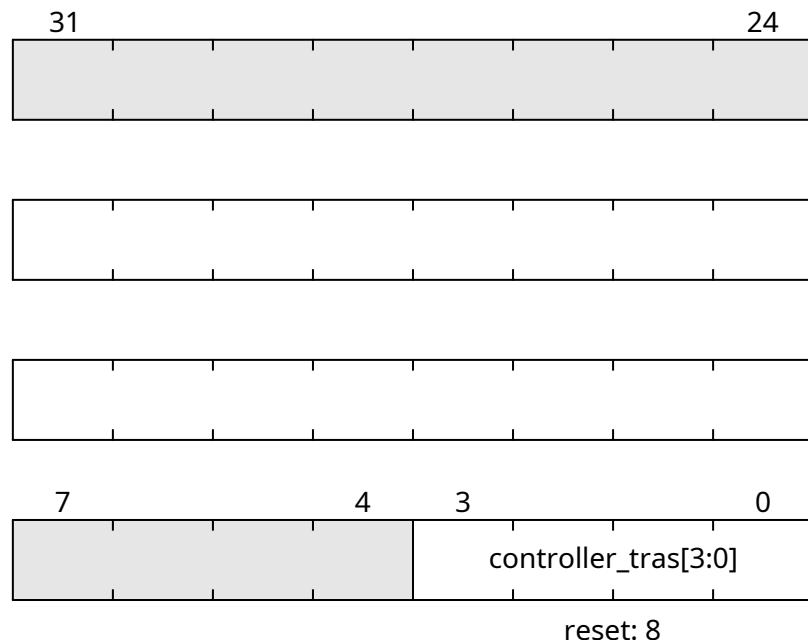


Fig. 24.230: SDRAM_CONTROLLER_TRAS

SDRAM_CONTROLLER_LAST_ADDR_0

Address: $0xf0007000 + 0xcc = 0xf00070cc$

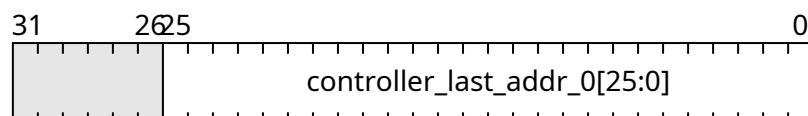


Fig. 24.231: SDRAM_CONTROLLER_LAST_ADDR_0

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0

Address: $0xf0007000 + 0xd0 = 0xf00070d0$

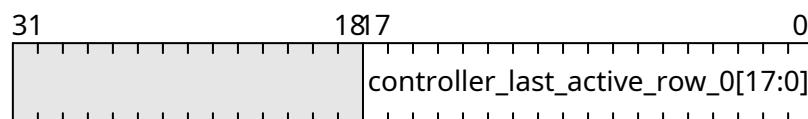


Fig. 24.232: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_0

SDRAM_CONTROLLER_LAST_ADDR_1

Address: $0xf0007000 + 0xd4 = 0xf00070d4$

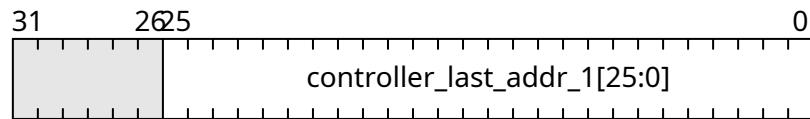


Fig. 24.233: SDRAM_CONTROLLER_LAST_ADDR_1

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1

Address: $0xf0007000 + 0xd8 = 0xf00070d8$

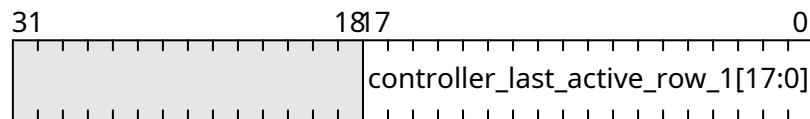


Fig. 24.234: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_1

SDRAM_CONTROLLER_LAST_ADDR_2

Address: $0xf0007000 + 0xdc = 0xf00070dc$

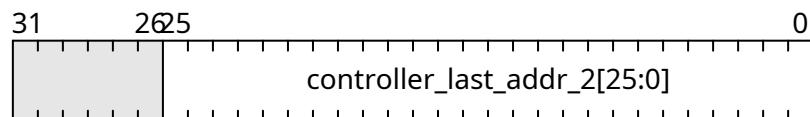


Fig. 24.235: SDRAM_CONTROLLER_LAST_ADDR_2

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2

Address: $0xf0007000 + 0xe0 = 0xf00070e0$

SDRAM_CONTROLLER_LAST_ADDR_3

Address: $0xf0007000 + 0xe4 = 0xf00070e4$

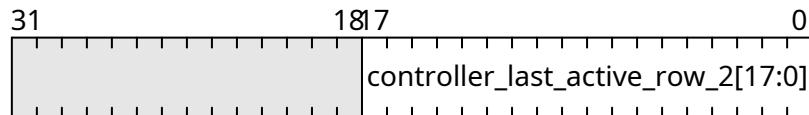


Fig. 24.236: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_2

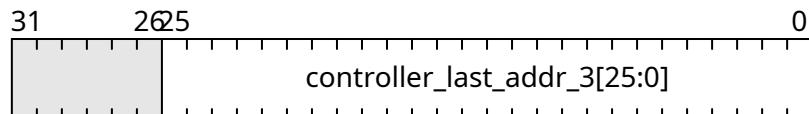


Fig. 24.237: SDRAM_CONTROLLER_LAST_ADDR_3

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3

Address: $0xf0007000 + 0xe8 = 0xf00070e8$

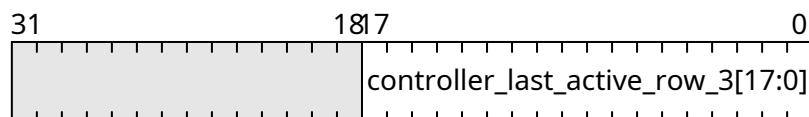


Fig. 24.238: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_3

SDRAM_CONTROLLER_LAST_ADDR_4

Address: $0xf0007000 + 0xec = 0xf00070ec$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4

Address: $0xf0007000 + 0xf0 = 0xf00070f0$

SDRAM_CONTROLLER_LAST_ADDR_5

Address: $0xf0007000 + 0xf4 = 0xf00070f4$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5

Address: $0xf0007000 + 0xf8 = 0xf00070f8$

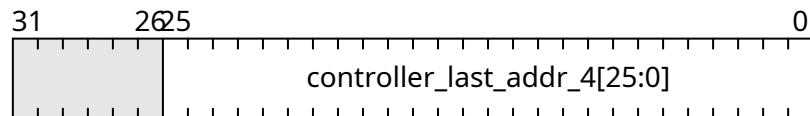


Fig. 24.239: SDRAM_CONTROLLER_LAST_ADDR_4

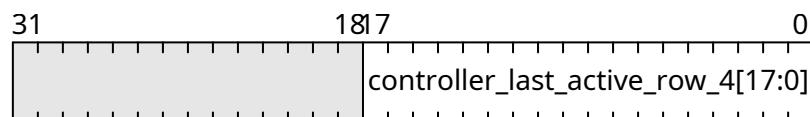


Fig. 24.240: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_4

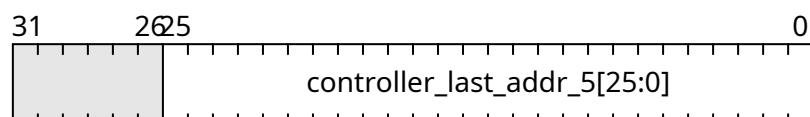


Fig. 24.241: SDRAM_CONTROLLER_LAST_ADDR_5

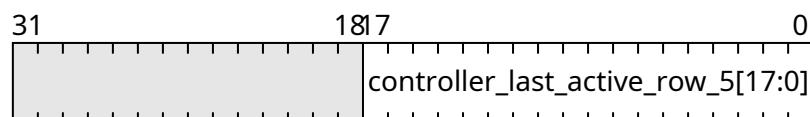


Fig. 24.242: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_5

SDRAM_CONTROLLER_LAST_ADDR_6

Address: $0xf0007000 + 0xfc = 0xf00070fc$

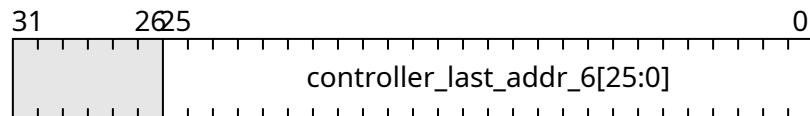


Fig. 24.243: SDRAM_CONTROLLER_LAST_ADDR_6

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6

Address: $0xf0007000 + 0x100 = 0xf0007100$

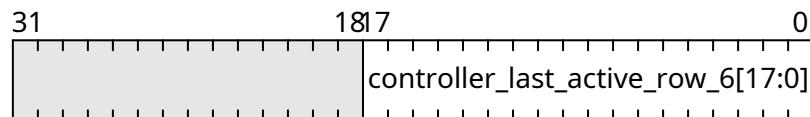


Fig. 24.244: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_6

SDRAM_CONTROLLER_LAST_ADDR_7

Address: $0xf0007000 + 0x104 = 0xf0007104$

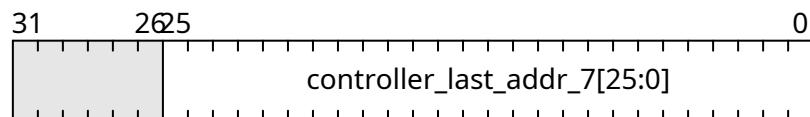


Fig. 24.245: SDRAM_CONTROLLER_LAST_ADDR_7

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7

Address: $0xf0007000 + 0x108 = 0xf0007108$

SDRAM_CONTROLLER_LAST_ADDR_8

Address: $0xf0007000 + 0x10c = 0xf000710c$

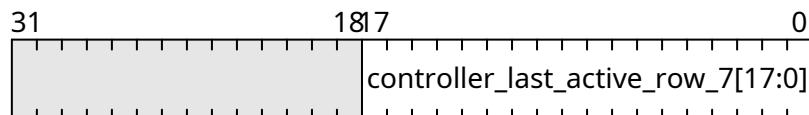


Fig. 24.246: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_7

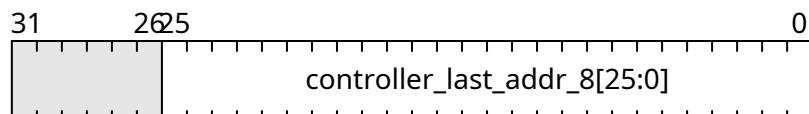


Fig. 24.247: SDRAM_CONTROLLER_LAST_ADDR_8

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_8

Address: $0xf0007000 + 0x110 = 0xf0007110$

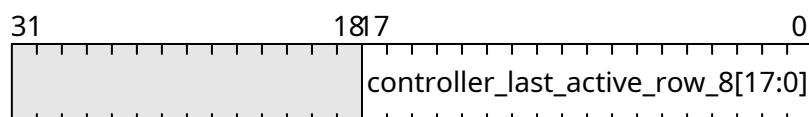


Fig. 24.248: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_8

SDRAM_CONTROLLER_LAST_ADDR_9

Address: $0xf0007000 + 0x114 = 0xf0007114$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_9

Address: $0xf0007000 + 0x118 = 0xf0007118$

SDRAM_CONTROLLER_LAST_ADDR_10

Address: $0xf0007000 + 0x11c = 0xf000711c$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_10

Address: $0xf0007000 + 0x120 = 0xf0007120$

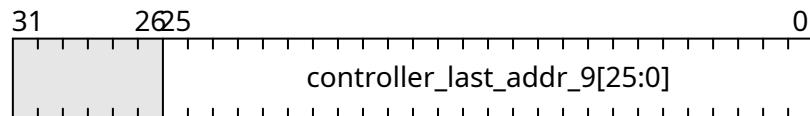


Fig. 24.249: SDRAM_CONTROLLER_LAST_ADDR_9

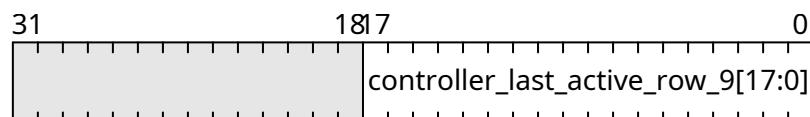


Fig. 24.250: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_9

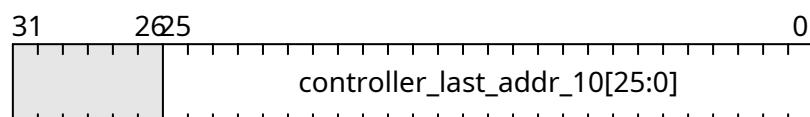


Fig. 24.251: SDRAM_CONTROLLER_LAST_ADDR_10

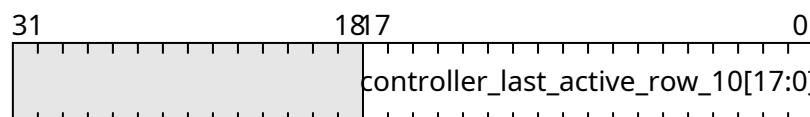


Fig. 24.252: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_10

SDRAM_CONTROLLER_LAST_ADDR_11

Address: $0xf0007000 + 0x124 = 0xf0007124$

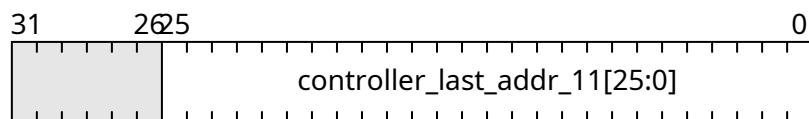


Fig. 24.253: SDRAM_CONTROLLER_LAST_ADDR_11

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_11

Address: $0xf0007000 + 0x128 = 0xf0007128$

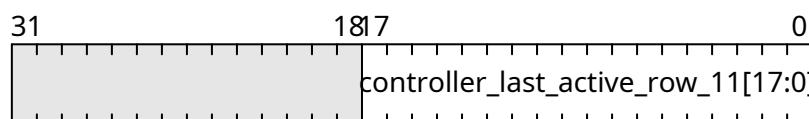


Fig. 24.254: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_11

SDRAM_CONTROLLER_LAST_ADDR_12

Address: $0xf0007000 + 0x12c = 0xf000712c$

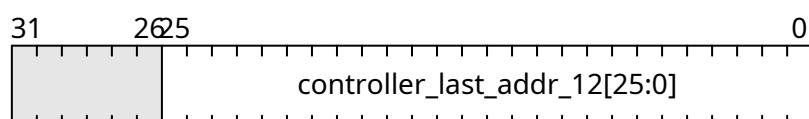


Fig. 24.255: SDRAM_CONTROLLER_LAST_ADDR_12

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_12

Address: $0xf0007000 + 0x130 = 0xf0007130$

SDRAM_CONTROLLER_LAST_ADDR_13

Address: $0xf0007000 + 0x134 = 0xf0007134$

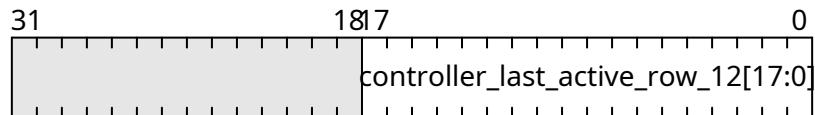


Fig. 24.256: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_12

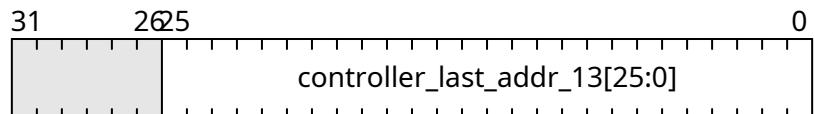


Fig. 24.257: SDRAM_CONTROLLER_LAST_ADDR_13

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_13

Address: 0xf0007000 + 0x138 = 0xf0007138

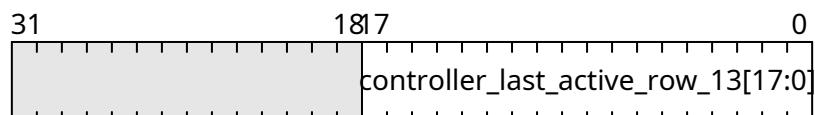


Fig. 24.258: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_13

SDRAM_CONTROLLER_LAST_ADDR_14

Address: 0xf0007000 + 0x13c = 0xf000713c

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_14

Address: 0xf0007000 + 0x140 = 0xf0007140

SDRAM_CONTROLLER_LAST_ADDR_15

Address: 0xf0007000 + 0x144 = 0xf0007144

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_15

Address: 0xf0007000 + 0x148 = 0xf0007148

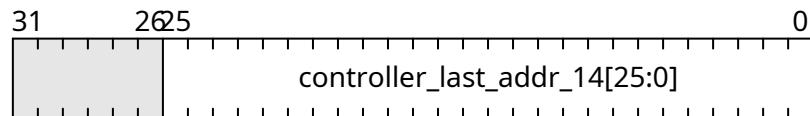


Fig. 24.259: SDRAM_CONTROLLER_LAST_ADDR_14

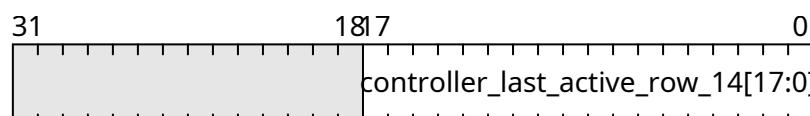


Fig. 24.260: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_14

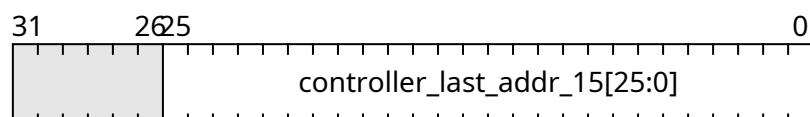


Fig. 24.261: SDRAM_CONTROLLER_LAST_ADDR_15

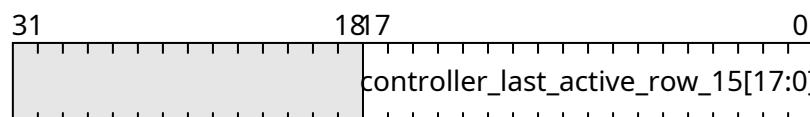


Fig. 24.262: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_15

SDRAM_CONTROLLER_LAST_ADDR_16

Address: $0xf0007000 + 0x14c = 0xf000714c$

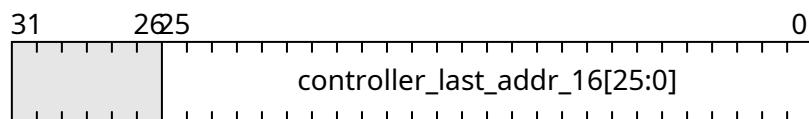


Fig. 24.263: SDRAM_CONTROLLER_LAST_ADDR_16

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_16

Address: $0xf0007000 + 0x150 = 0xf0007150$

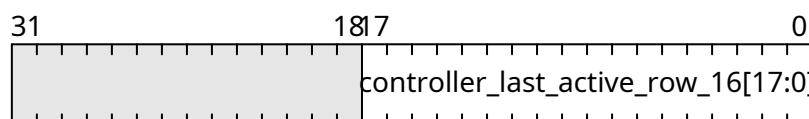


Fig. 24.264: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_16

SDRAM_CONTROLLER_LAST_ADDR_17

Address: $0xf0007000 + 0x154 = 0xf0007154$

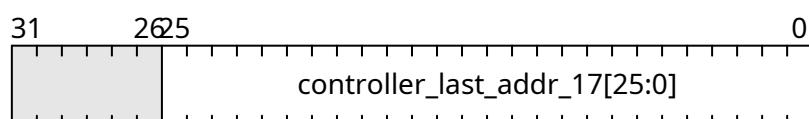


Fig. 24.265: SDRAM_CONTROLLER_LAST_ADDR_17

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_17

Address: $0xf0007000 + 0x158 = 0xf0007158$

SDRAM_CONTROLLER_LAST_ADDR_18

Address: $0xf0007000 + 0x15c = 0xf000715c$

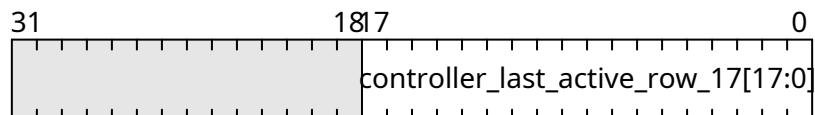


Fig. 24.266: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_17

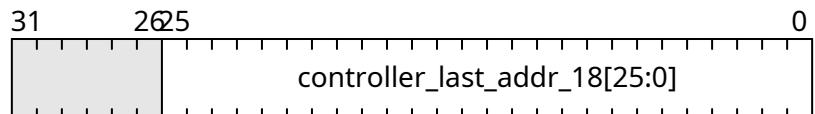


Fig. 24.267: SDRAM_CONTROLLER_LAST_ADDR_18

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_18

Address: $0xf0007000 + 0x160 = 0xf0007160$

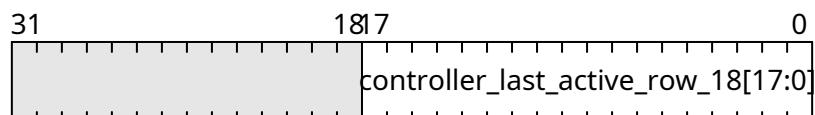


Fig. 24.268: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_18

SDRAM_CONTROLLER_LAST_ADDR_19

Address: $0xf0007000 + 0x164 = 0xf0007164$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_19

Address: $0xf0007000 + 0x168 = 0xf0007168$

SDRAM_CONTROLLER_LAST_ADDR_20

Address: $0xf0007000 + 0x16c = 0xf000716c$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_20

Address: $0xf0007000 + 0x170 = 0xf0007170$

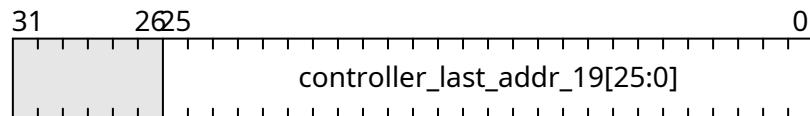


Fig. 24.269: SDRAM_CONTROLLER_LAST_ADDR_19

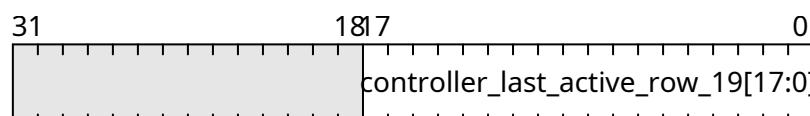


Fig. 24.270: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_19

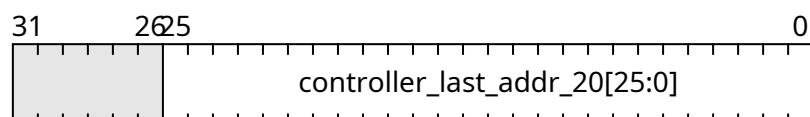


Fig. 24.271: SDRAM_CONTROLLER_LAST_ADDR_20

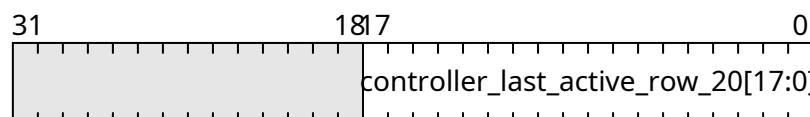


Fig. 24.272: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_20

SDRAM_CONTROLLER_LAST_ADDR_21

Address: $0xf0007000 + 0x174 = 0xf0007174$

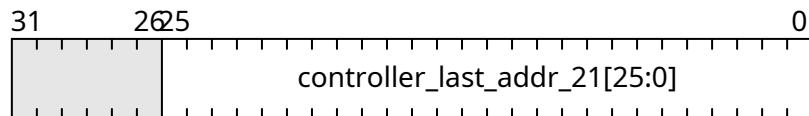


Fig. 24.273: SDRAM_CONTROLLER_LAST_ADDR_21

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_21

Address: $0xf0007000 + 0x178 = 0xf0007178$

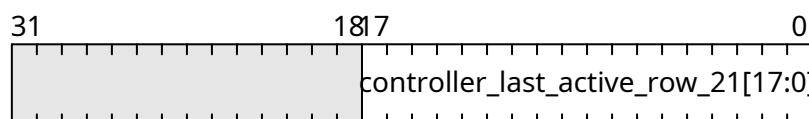


Fig. 24.274: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_21

SDRAM_CONTROLLER_LAST_ADDR_22

Address: $0xf0007000 + 0x17c = 0xf000717c$

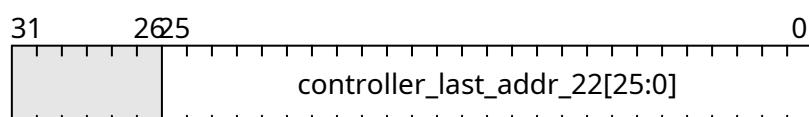


Fig. 24.275: SDRAM_CONTROLLER_LAST_ADDR_22

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_22

Address: $0xf0007000 + 0x180 = 0xf0007180$

SDRAM_CONTROLLER_LAST_ADDR_23

Address: $0xf0007000 + 0x184 = 0xf0007184$

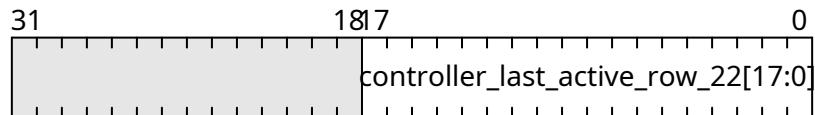


Fig. 24.276: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_22

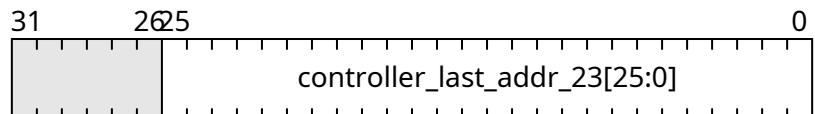


Fig. 24.277: SDRAM_CONTROLLER_LAST_ADDR_23

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_23

Address: $0xf0007000 + 0x188 = 0xf0007188$

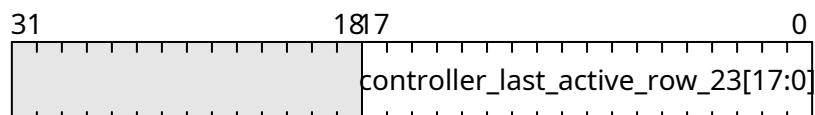


Fig. 24.278: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_23

SDRAM_CONTROLLER_LAST_ADDR_24

Address: $0xf0007000 + 0x18c = 0xf000718c$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_24

Address: $0xf0007000 + 0x190 = 0xf0007190$

SDRAM_CONTROLLER_LAST_ADDR_25

Address: $0xf0007000 + 0x194 = 0xf0007194$

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_25

Address: $0xf0007000 + 0x198 = 0xf0007198$

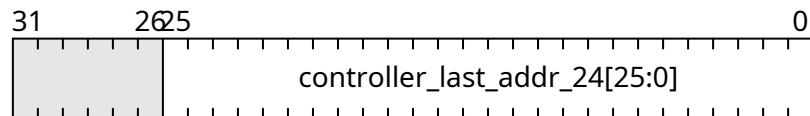


Fig. 24.279: SDRAM_CONTROLLER_LAST_ADDR_24

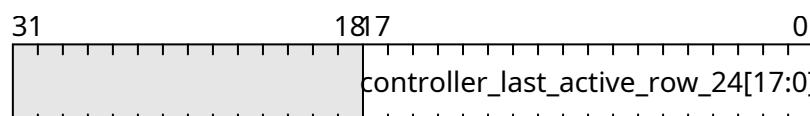


Fig. 24.280: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_24

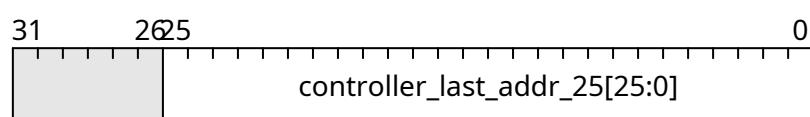


Fig. 24.281: SDRAM_CONTROLLER_LAST_ADDR_25

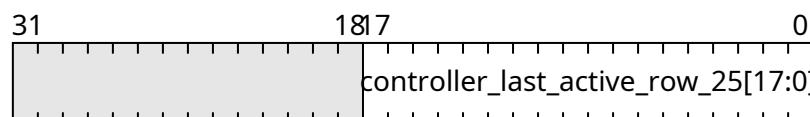


Fig. 24.282: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_25

SDRAM_CONTROLLER_LAST_ADDR_26

Address: $0xf0007000 + 0x19c = 0xf000719c$

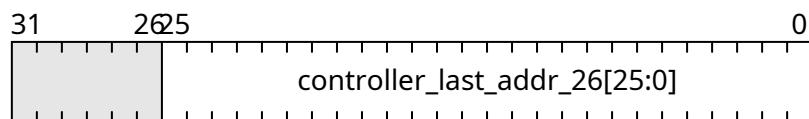


Fig. 24.283: SDRAM_CONTROLLER_LAST_ADDR_26

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_26

Address: $0xf0007000 + 0x1a0 = 0xf00071a0$

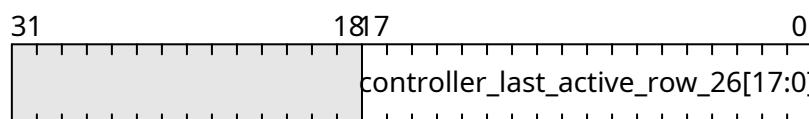


Fig. 24.284: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_26

SDRAM_CONTROLLER_LAST_ADDR_27

Address: $0xf0007000 + 0x1a4 = 0xf00071a4$

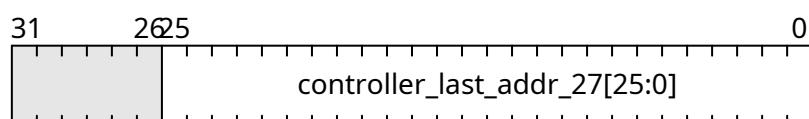


Fig. 24.285: SDRAM_CONTROLLER_LAST_ADDR_27

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_27

Address: $0xf0007000 + 0x1a8 = 0xf00071a8$

SDRAM_CONTROLLER_LAST_ADDR_28

Address: $0xf0007000 + 0x1ac = 0xf00071ac$

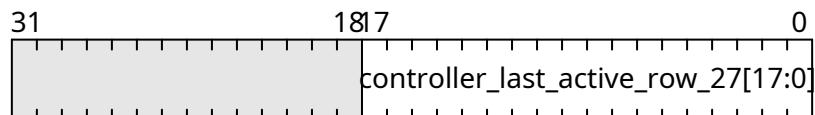


Fig. 24.286: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_27

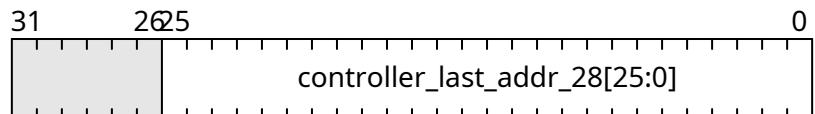


Fig. 24.287: SDRAM_CONTROLLER_LAST_ADDR_28

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_28

Address: 0xf0007000 + 0x1b0 = 0xf00071b0

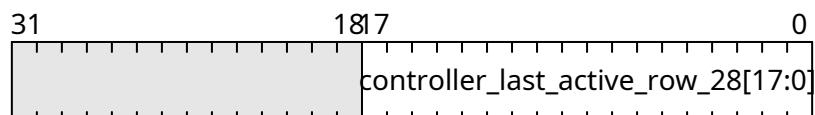


Fig. 24.288: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_28

SDRAM_CONTROLLER_LAST_ADDR_29

Address: 0xf0007000 + 0x1b4 = 0xf00071b4

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_29

Address: 0xf0007000 + 0x1b8 = 0xf00071b8

SDRAM_CONTROLLER_LAST_ADDR_30

Address: 0xf0007000 + 0x1bc = 0xf00071bc

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_30

Address: 0xf0007000 + 0x1c0 = 0xf00071c0

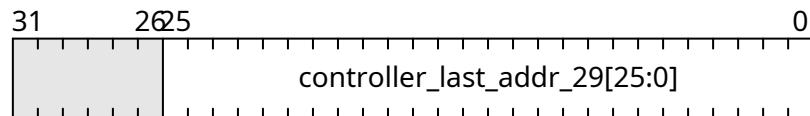


Fig. 24.289: SDRAM_CONTROLLER_LAST_ADDR_29

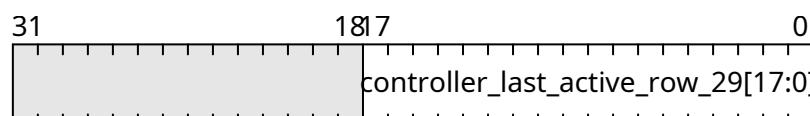


Fig. 24.290: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_29

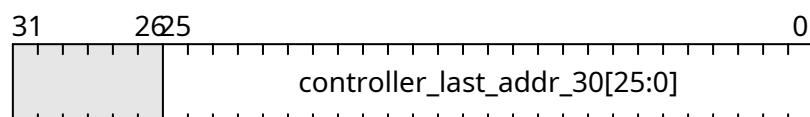


Fig. 24.291: SDRAM_CONTROLLER_LAST_ADDR_30

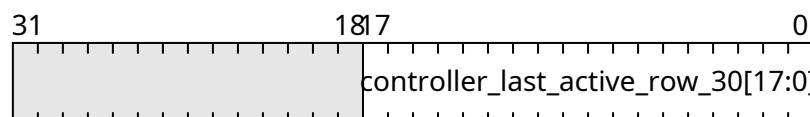


Fig. 24.292: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_30

SDRAM_CONTROLLER_LAST_ADDR_31

Address: $0xf0007000 + 0x1c4 = 0xf00071c4$

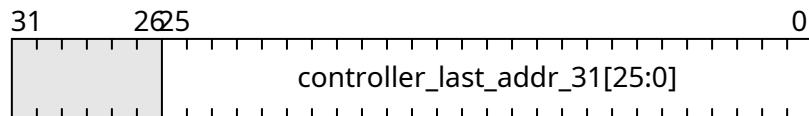


Fig. 24.293: SDRAM_CONTROLLER_LAST_ADDR_31

SDRAM_CONTROLLER_LAST_ACTIVE_ROW_31

Address: $0xf0007000 + 0x1c8 = 0xf00071c8$

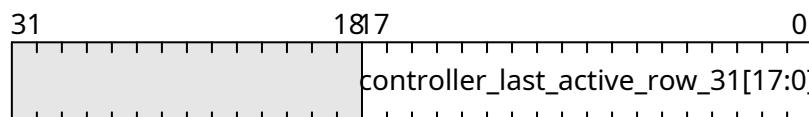


Fig. 24.294: SDRAM_CONTROLLER_LAST_ACTIVE_ROW_31

24.2.16 SDRAM_CHECKER

Register Listing for SDRAM_CHECKER

Register	Address
<i>SDRAM_CHECKER_RESET</i>	<i>0xf0007800</i>
<i>SDRAM_CHECKER_START</i>	<i>0xf0007804</i>
<i>SDRAM_CHECKER_DONE</i>	<i>0xf0007808</i>
<i>SDRAM_CHECKER_BASE1</i>	<i>0xf000780c</i>
<i>SDRAM_CHECKER_BASE0</i>	<i>0xf0007810</i>
<i>SDRAM_CHECKER_END1</i>	<i>0xf0007814</i>
<i>SDRAM_CHECKER_END0</i>	<i>0xf0007818</i>
<i>SDRAM_CHECKER_LENGTH1</i>	<i>0xf000781c</i>
<i>SDRAM_CHECKER_LENGTH0</i>	<i>0xf0007820</i>
<i>SDRAM_CHECKER_RANDOM</i>	<i>0xf0007824</i>
<i>SDRAM_CHECKER_TICKS</i>	<i>0xf0007828</i>
<i>SDRAM_CHECKER_ERRORS</i>	<i>0xf000782c</i>

SDRAM_CHECKER_RESET

Address: $0xf0007800 + 0x0 = 0xf0007800$



Fig. 24.295: SDRAM_CHECKER_RESET

SDRAM_CHECKER_START

Address: $0xf0007800 + 0x4 = 0xf0007804$

SDRAM_CHECKER_DONE

Address: $0xf0007800 + 0x8 = 0xf0007808$

SDRAM_CHECKER_BASE1

Address: $0xf0007800 + 0xc = 0xf000780c$

Bits 32-35 of *SDRAM_CHECKER_BASE*.

SDRAM_CHECKER_BASE0

Address: $0xf0007800 + 0x10 = 0xf0007810$

Bits 0-31 of *SDRAM_CHECKER_BASE*.

SDRAM_CHECKER_END1

Address: $0xf0007800 + 0x14 = 0xf0007814$

Bits 32-35 of *SDRAM_CHECKER_END*.



Fig. 24.296: SDRAM_CHECKER_START



Fig. 24.297: SDRAM_CHECKER_DONE

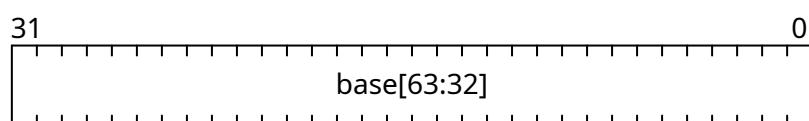


Fig. 24.298: SDRAM_CHECKER_BASE1

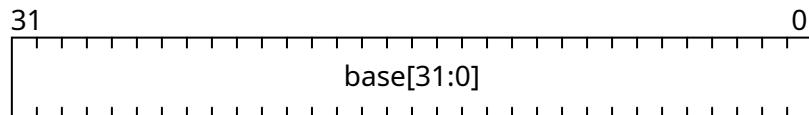


Fig. 24.299: SDRAM_CHECKER_BASE0

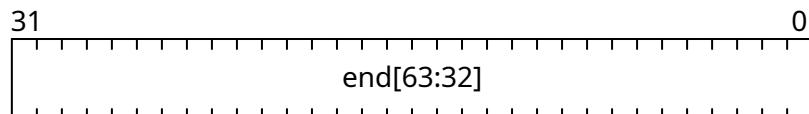


Fig. 24.300: SDRAM_CHECKER_END1

SDRAM_CHECKER_END0

Address: 0xf0007800 + 0x18 = 0xf0007818

Bits 0-31 of *SDRAM_CHECKER_END*.

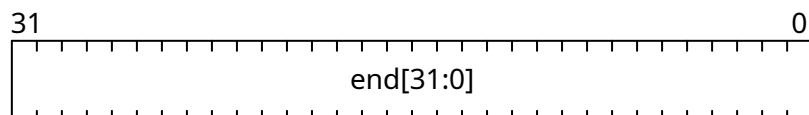


Fig. 24.301: SDRAM_CHECKER_END0

SDRAM_CHECKER_LENGTH1

Address: 0xf0007800 + 0x1c = 0xf000781c

Bits 32-35 of *SDRAM_CHECKER_LENGTH*.

SDRAM_CHECKER_LENGTH0

Address: 0xf0007800 + 0x20 = 0xf0007820

Bits 0-31 of *SDRAM_CHECKER_LENGTH*.

SDRAM_CHECKER_RANDOM

Address: 0xf0007800 + 0x24 = 0xf0007824

Field	Name	Description

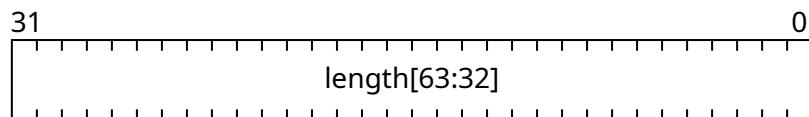


Fig. 24.302: SDRAM_CHECKER_LENGTH1

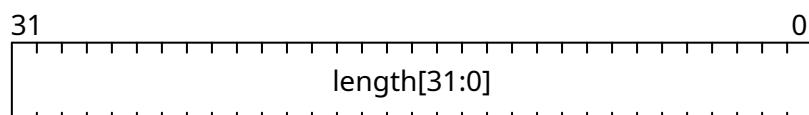


Fig. 24.303: SDRAM_CHECKER_LENGTH0



Fig. 24.304: SDRAM_CHECKER_RANDOM

SDRAM_CHECKER_TICKS

Address: $0xf0007800 + 0x28 = 0xf0007828$

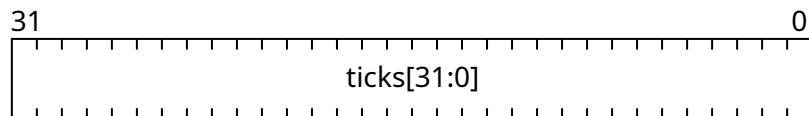


Fig. 24.305: SDRAM_CHECKER_TICKS

SDRAM_CHECKER_ERRORS

Address: $0xf0007800 + 0x2c = 0xf000782c$

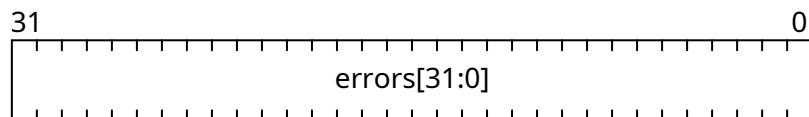


Fig. 24.306: SDRAM_CHECKER_ERRORS

24.2.17 SDRAM_GENERATOR

Register Listing for SDRAM_GENERATOR

Register	Address
<i>SDRAM_GENERATOR_RESET</i>	<i>0xf0008000</i>
<i>SDRAM_GENERATOR_START</i>	<i>0xf0008004</i>
<i>SDRAM_GENERATOR_DONE</i>	<i>0xf0008008</i>
<i>SDRAM_GENERATOR_BASE1</i>	<i>0xf000800c</i>
<i>SDRAM_GENERATOR_BASE0</i>	<i>0xf0008010</i>
<i>SDRAM_GENERATOR_END1</i>	<i>0xf0008014</i>
<i>SDRAM_GENERATOR_END0</i>	<i>0xf0008018</i>
<i>SDRAM_GENERATOR_LENGTH1</i>	<i>0xf000801c</i>
<i>SDRAM_GENERATOR_LENGTH0</i>	<i>0xf0008020</i>
<i>SDRAM_GENERATOR_RANDOM</i>	<i>0xf0008024</i>
<i>SDRAM_GENERATOR_TICKS</i>	<i>0xf0008028</i>

SDRAM_GENERATOR_RESET

Address: $0xf0008000 + 0x0 = 0xf0008000$



Fig. 24.307: SDRAM_GENERATOR_RESET

SDRAM_GENERATOR_START

Address: 0xf0008000 + 0x4 = 0xf0008004

SDRAM_GENERATOR_DONE

Address: 0xf0008000 + 0x8 = 0xf0008008

SDRAM_GENERATOR_BASE1

Address: 0xf0008000 + 0xc = 0xf000800c

Bits 32-35 of *SDRAM_GENERATOR_BASE*.

SDRAM_GENERATOR_BASE0

Address: 0xf0008000 + 0x10 = 0xf0008010

Bits 0-31 of *SDRAM_GENERATOR_BASE*.

SDRAM_GENERATOR_END1

Address: 0xf0008000 + 0x14 = 0xf0008014

Bits 32-35 of *SDRAM_GENERATOR_END*.



Fig. 24.308: SDRAM_GENERATOR_START



Fig. 24.309: SDRAM_GENERATOR_DONE

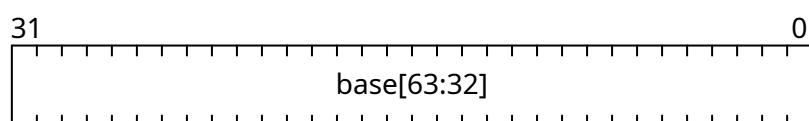


Fig. 24.310: SDRAM_GENERATOR_BASE1

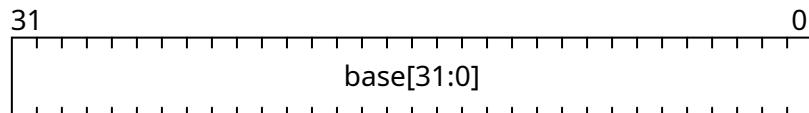


Fig. 24.311: SDRAM_GENERATOR_BASE0

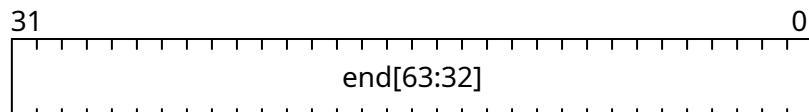


Fig. 24.312: SDRAM_GENERATOR_END1

SDRAM_GENERATOR_END0

Address: 0xf0008000 + 0x18 = 0xf0008018

Bits 0-31 of *SDRAM_GENERATOR_END*.

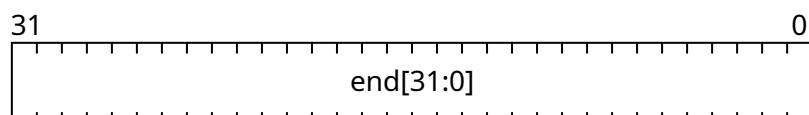


Fig. 24.313: SDRAM_GENERATOR_END0

SDRAM_GENERATOR_LENGTH1

Address: 0xf0008000 + 0x1c = 0xf000801c

Bits 32-35 of *SDRAM_GENERATOR_LENGTH*.

SDRAM_GENERATOR_LENGTH0

Address: 0xf0008000 + 0x20 = 0xf0008020

Bits 0-31 of *SDRAM_GENERATOR_LENGTH*.

SDRAM_GENERATOR_RANDOM

Address: 0xf0008000 + 0x24 = 0xf0008024

Field	Name	Description

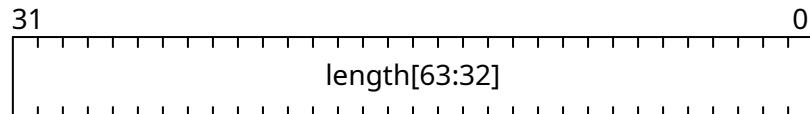


Fig. 24.314: SDRAM_GENERATOR_LENGTH1

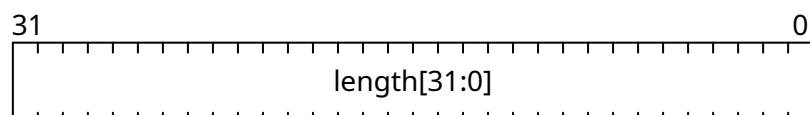


Fig. 24.315: SDRAM_GENERATOR_LENGTH0



Fig. 24.316: SDRAM_GENERATOR_RANDOM

SDRAM_GENERATOR_TICKS

Address: $0xf0008000 + 0x28 = 0xf0008028$

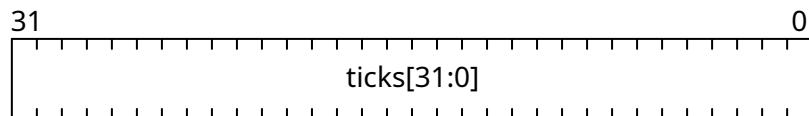


Fig. 24.317: SDRAM_GENERATOR_TICKS

24.2.18 TIMERO

Timer

Provides a generic Timer core.

The Timer is implemented as a countdown timer that can be used in various modes:

- Polling : Returns current countdown value to software
- One-Shot: Loads itself and stops when value reaches 0
- Periodic: (Re-)Loads itself when value reaches 0

en register allows the user to enable/disable the Timer. When the Timer is enabled, it is automatically loaded with the value of *load* register.

When the Timer reaches 0, it is automatically reloaded with value of *reload* register.

The user can latch the current countdown value by writing to *update_value* register, it will update *value* register with current countdown value.

To use the Timer in One-Shot mode, the user needs to:

- Disable the timer
- Set the load register to the expected duration
- (Re-)Enable the Timer

To use the Timer in Periodic mode, the user needs to:

- Disable the Timer
- Set the load register to 0
- Set the reload register to the expected period
- Enable the Timer

For both modes, the CPU can be advertised by an IRQ that the duration/period has elapsed. (The CPU can also do software polling with *update_value* and *value* to know the elapsed duration)

Register Listing for TIMERO

Register	Address
<i>TIMERO_LOAD</i>	0xf0008800
<i>TIMERO_RELOAD</i>	0xf0008804
<i>TIMERO_EN</i>	0xf0008808
<i>TIMERO_UPDATE_VALUE</i>	0xf000880c
<i>TIMERO_VALUE</i>	0xf0008810
<i>TIMERO_EV_STATUS</i>	0xf0008814
<i>TIMERO_EV_PENDING</i>	0xf0008818
<i>TIMERO_EV_ENABLE</i>	0xf000881c

TIMERO_LOAD

Address: $0xf0008800 + 0x0 = 0xf0008800$

Load value when Timer is (re-)enabled. In One-Shot mode, the value written to this register specifies the Timer's duration in clock cycles.

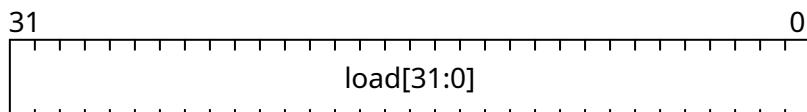


Fig. 24.318: TIMERO_LOAD

TIMERO_RELOAD

Address: $0xf0008800 + 0x4 = 0xf0008804$

Reload value when Timer reaches 0. In Periodic mode, the value written to this register specify the Timer's period in clock cycles.

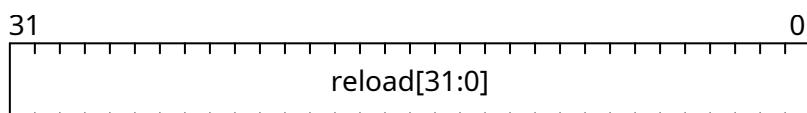


Fig. 24.319: TIMERO_RELOAD

TIMERO_EN

Address: $0xf0008800 + 0x8 = 0xf0008808$

Enable flag of the Timer. Set this flag to 1 to enable/start the Timer. Set to 0 to disable the Timer.



Fig. 24.320: TIMERO_EN

TIMERO_UPDATE_VALUE

Address: $0xf0008800 + 0xc = 0xf000880c$

Update trigger for the current countdown value. A write to this register latches the current countdown value to value register.

TIMERO_VALUE

Address: $0xf0008800 + 0x10 = 0xf0008810$

Latched countdown value. This value is updated by writing to update_value.

TIMERO_EV_STATUS

Address: $0xf0008800 + 0x14 = 0xf0008814$

This register contains the current raw level of the zero event trigger. Writes to this register have no effect.

Field	Name	Description
[0]	ZERO	Level of the zero event

TIMERO_EV_PENDING

Address: $0xf0008800 + 0x18 = 0xf0008818$

When a zero event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.



Fig. 24.321: TIMERO_UPDATE_VALUE

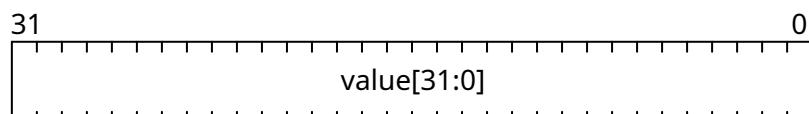


Fig. 24.322: TIMERO_VALUE



Fig. 24.323: TIMERO_EV_STATUS



Fig. 24.324: TIMERO_EV_PENDING

Field	Name	Description
[0]	ZERO	1 if a zero event occurred. This Event is triggered on a falling edge.

TIMERO_EV_ENABLE

Address: $0xf0008800 + 0x1c = 0xf000881c$

This register enables the corresponding zero events. Write a 0 to this register to disable individual events.

Field	Name	Description
[0]	ZERO	Write a 1 to enable the zero Event

24.2.19 UART



Fig. 24.325: TIMERO_EV_ENABLE

Register Listing for UART

Register	Address
UART_RXTX	0xf0009000
UART_TXFULL	0xf0009004
UART_RXEMPTY	0xf0009008
UART_EV_STATUS	0xf000900c
UART_EV_PENDING	0xf0009010
UART_EV_ENABLE	0xf0009014
UART_TXEMPTY	0xf0009018
UART_RXFULL	0xf000901c
UART_XOVER_RXTX	0xf0009020
UART_XOVER_TXFULL	0xf0009024
UART_XOVER_RXEMPTY	0xf0009028
UART_XOVER_EV_STATUS	0xf000902c
UART_XOVER_EV_PENDING	0xf0009030
UART_XOVER_EV_ENABLE	0xf0009034
UART_XOVER_TXEMPTY	0xf0009038
UART_XOVER_RXFULL	0xf000903c

UART_RXTX

Address: $0xf0009000 + 0x0 = 0xf0009000$



Fig. 24.326: UART_RXTX

UART_TXFULL

Address: $0xf0009000 + 0x4 = 0xf0009004$

TX FIFO Full.



Fig. 24.327: UART_TXFULL

UART_RXEMPTY

Address: $0xf0009000 + 0x8 = 0xf0009008$

RX FIFO Empty.

UART_EV_STATUS

Address: $0xf0009000 + 0xc = 0xf000900c$

This register contains the current raw level of the rx event trigger. Writes to this register have no effect.

Field	Name	Description
[0]	TX	Level of the tx event
[1]	RX	Level of the rx event



Fig. 24.328: UART_RXEMPTY

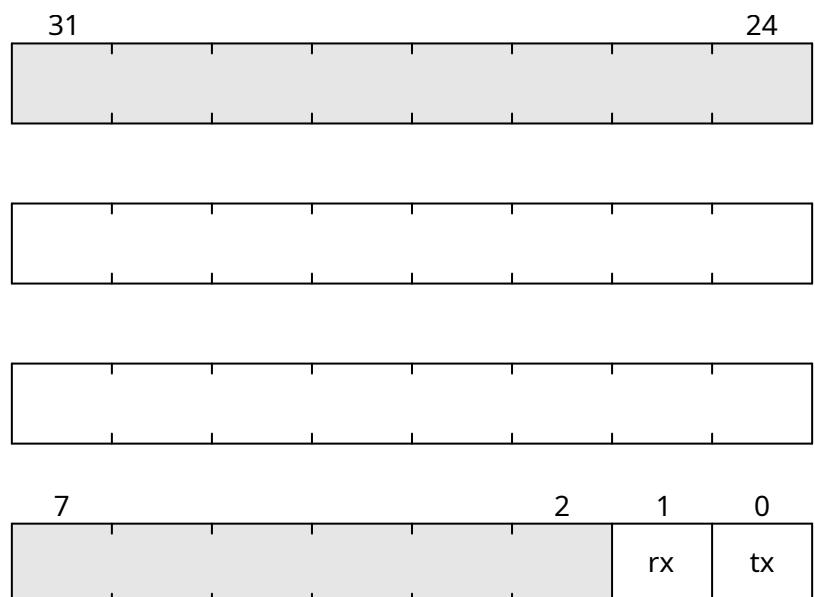


Fig. 24.329: UART_EV_STATUS

UART_EV_PENDING

Address: $0xf0009000 + 0x10 = 0xf0009010$

When a rx event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.

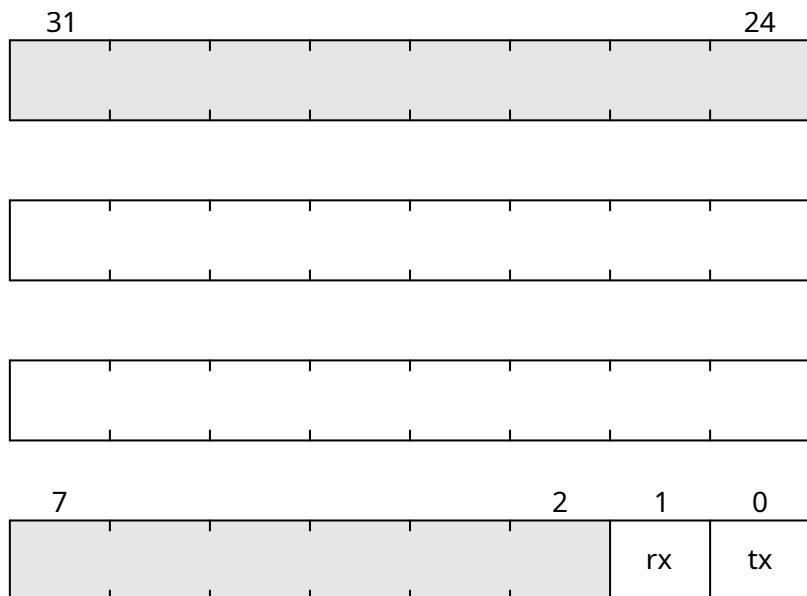


Fig. 24.330: UART_EV_PENDING

Field	Name	Description
[0]	TX	1 if a tx event occurred. This Event is triggered on a falling edge.
[1]	RX	1 if a rx event occurred. This Event is triggered on a falling edge.

UART_EV_ENABLE

Address: $0xf0009000 + 0x14 = 0xf0009014$

This register enables the corresponding rx events. Write a 0 to this register to disable individual events.

Field	Name	Description
[0]	TX	Write a 1 to enable the tx Event
[1]	RX	Write a 1 to enable the rx Event

UART_TXEMPTY

Address: $0xf0009000 + 0x18 = 0xf0009018$

TX FIFO Empty.

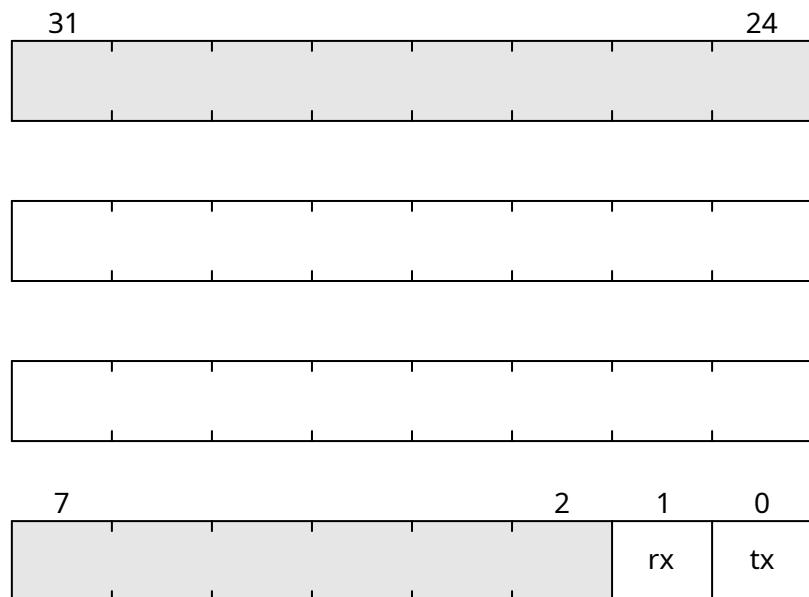


Fig. 24.331: **UART_EV_ENABLE**



Fig. 24.332: **UART_TXEMPTY**

UART_RXFULL

Address: $0xf0009000 + 0x1c = 0xf000901c$

RX FIFO Full.



Fig. 24.333: UART_RXFULL

UART_XOVER_RXTX

Address: $0xf0009000 + 0x20 = 0xf0009020$

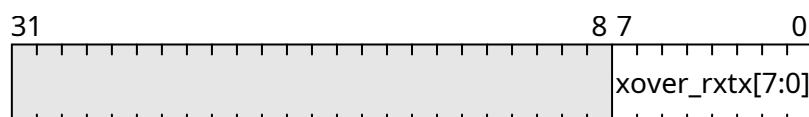


Fig. 24.334: UART_XOVER_RXTX

UART_XOVER_TXFULL

Address: $0xf0009000 + 0x24 = 0xf0009024$

TX FIFO Full.

UART_XOVER_RXEMPTY

Address: $0xf0009000 + 0x28 = 0xf0009028$

RX FIFO Empty.



Fig. 24.335: `UART_XOVER_TXFULL`

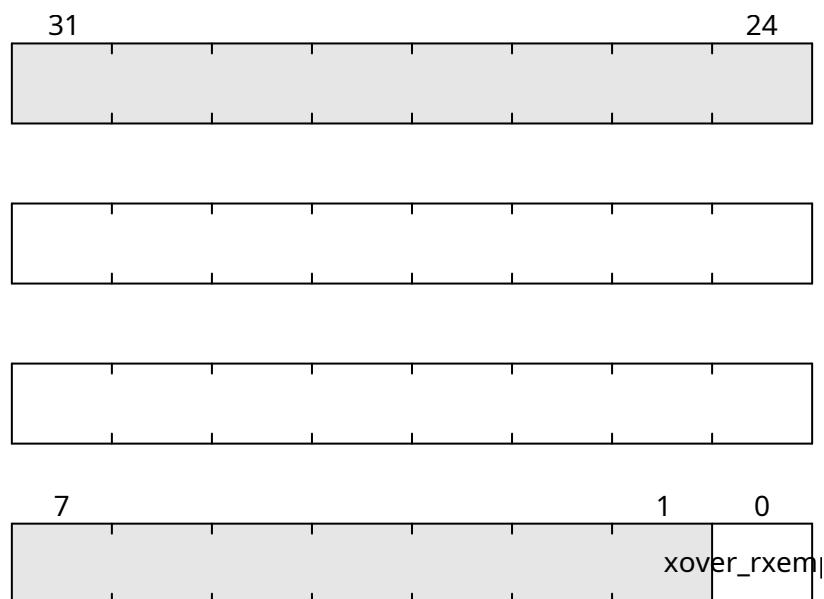


Fig. 24.336: `UART_XOVER_RXEMPTY`

UART_XOVER_EV_STATUS

Address: $0xf0009000 + 0x2c = 0xf000902c$

This register contains the current raw level of the rx event trigger. Writes to this register have no effect.

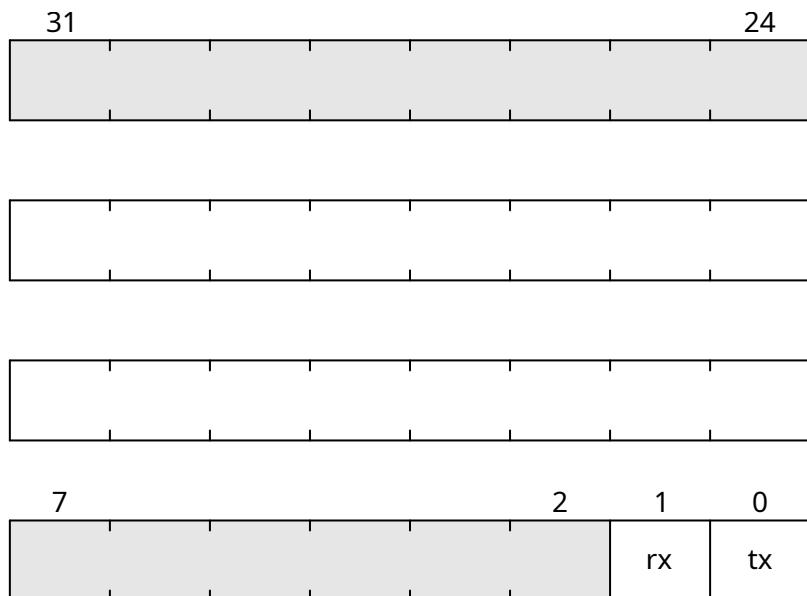


Fig. 24.337: UART_XOVER_EV_STATUS

Field	Name	Description
[0]	TX	Level of the tx event
[1]	RX	Level of the rx event

UART_XOVER_EV_PENDING

Address: $0xf0009000 + 0x30 = 0xf0009030$

When a rx event occurs, the corresponding bit will be set in this register. To clear the Event, set the corresponding bit in this register.

Field	Name	Description
[0]	TX	1 if a tx event occurred. This Event is triggered on a falling edge .
[1]	RX	1 if a rx event occurred. This Event is triggered on a falling edge .

UART_XOVER_EV_ENABLE

Address: $0xf0009000 + 0x34 = 0xf0009034$

This register enables the corresponding rx events. Write a 0 to this register to disable individual events.

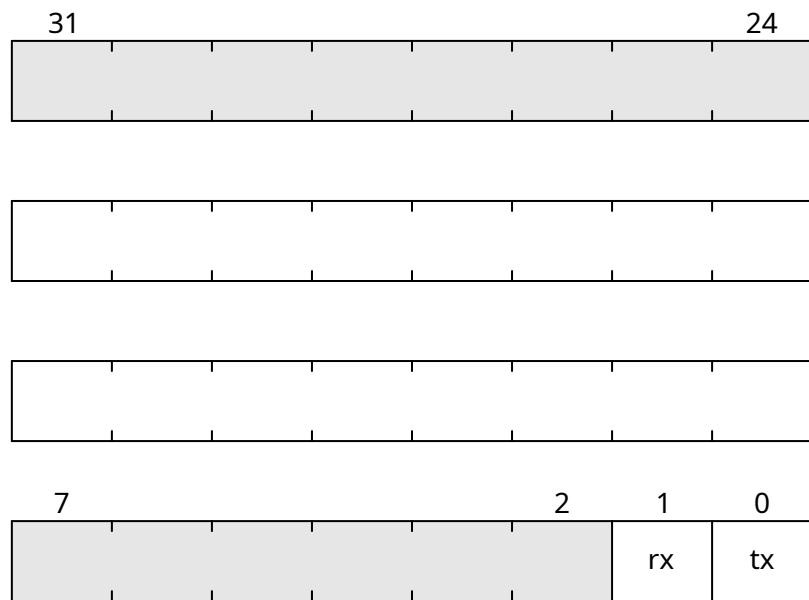


Fig. 24.338: `UART_XOVER_EV_PENDING`

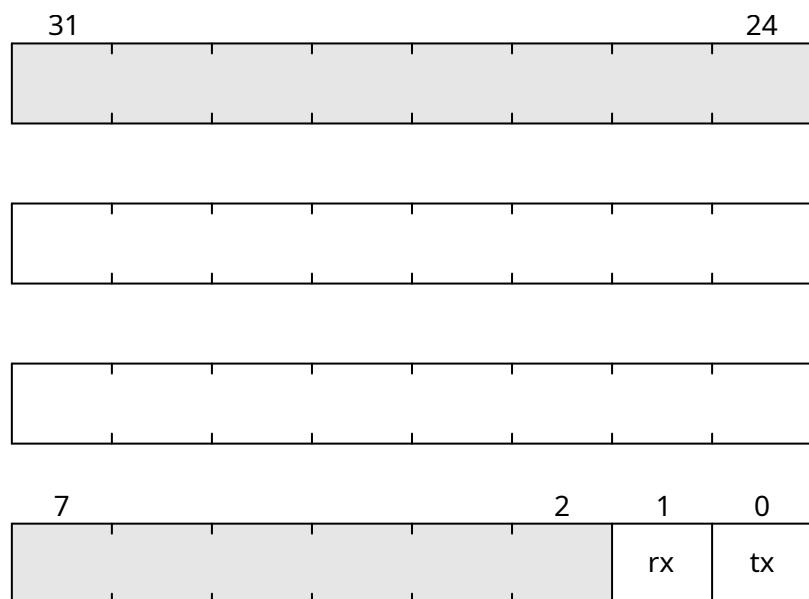


Fig. 24.339: `UART_XOVER_EV_ENABLE`

Field	Name	Description
[0]	TX	Write a 1 to enable the tx Event
[1]	RX	Write a 1 to enable the rx Event

UART_XOVER_TXEMPTY

Address: $0xf0009000 + 0x38 = 0xf0009038$

TX FIFO Empty.

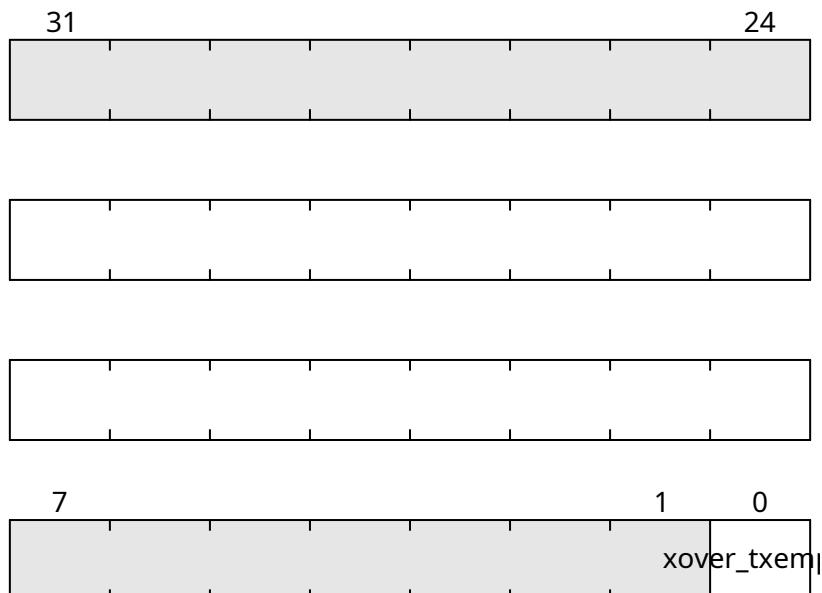


Fig. 24.340: UART_XOVER_TXEMPTY

UART_XOVER_RXFULL

Address: $0xf0009000 + 0x3c = 0xf000903c$

RX FIFO Full.

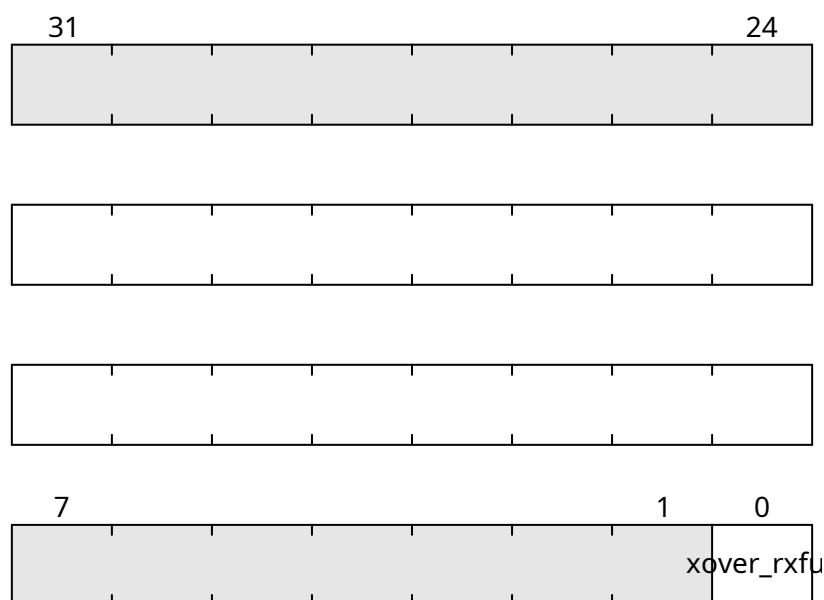


Fig. 24.341: UART_XOVER_RXFULL