

Hyper Bus Module: User Manual

March 2021   
Version 1.1  
Hayate Okuhara (hayate.okuhara@unibo.it)

Micrel Lab and Multitherman Lab  
University of Bologna, Italy

Integrated Systems Lab  
ETH Zürich, Switzerland

Copyright 2021 ETH Zurich and University of Bologna.

Copyright and related rights are licensed under the Solderpad Hardware License, Version 0.51 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://solderpad.org/licenses/SHL-0.51. Unless required by applicable law or agreed to in writing, software, hardware and materials distributed under this License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Document Revisions

| **Rev.** | **Date** | **Author** | **Description** |
| --- | --- | --- | --- |
| 1.0 | 22/02/20 | Hayate Okuhara | First Draft |
| 1.1 | 01/03/21 | Hayate Okuhara | Public release |

Table of Contents

1 Introduction 5

1.1 Scope and Purpose 5

2 Architecture 6

2.1 2D transfer splitter 7

2.2 Unpack module 8

2.3 Controller 9

2.4 32b/16b and 16b/32b modules 10

2.5 Hyper bus phy 11

3 Register Map 12

4 Reg Field 13

5 APIs 18

# Introduction

This IP enables data transfer between the PULP system installing µDMA and off-chip memory modules compatible with the Hyper-Bus protocol. The source codes are written in System Verilog. It supports, the linear transfer, the page boundary consideration, byte addressing, and 2D data transfer. This design was tested mainly with Hyper RAM and these functionality are for DRAM memory products not for Flash one. For Flash products such as Hyper Flash, just read and write path are supported.

## Scope and Purpose

* Architecture of the module
* Memory map of the configuration registers
* APIs for c code
* Example source code to activate the module

# Architecture

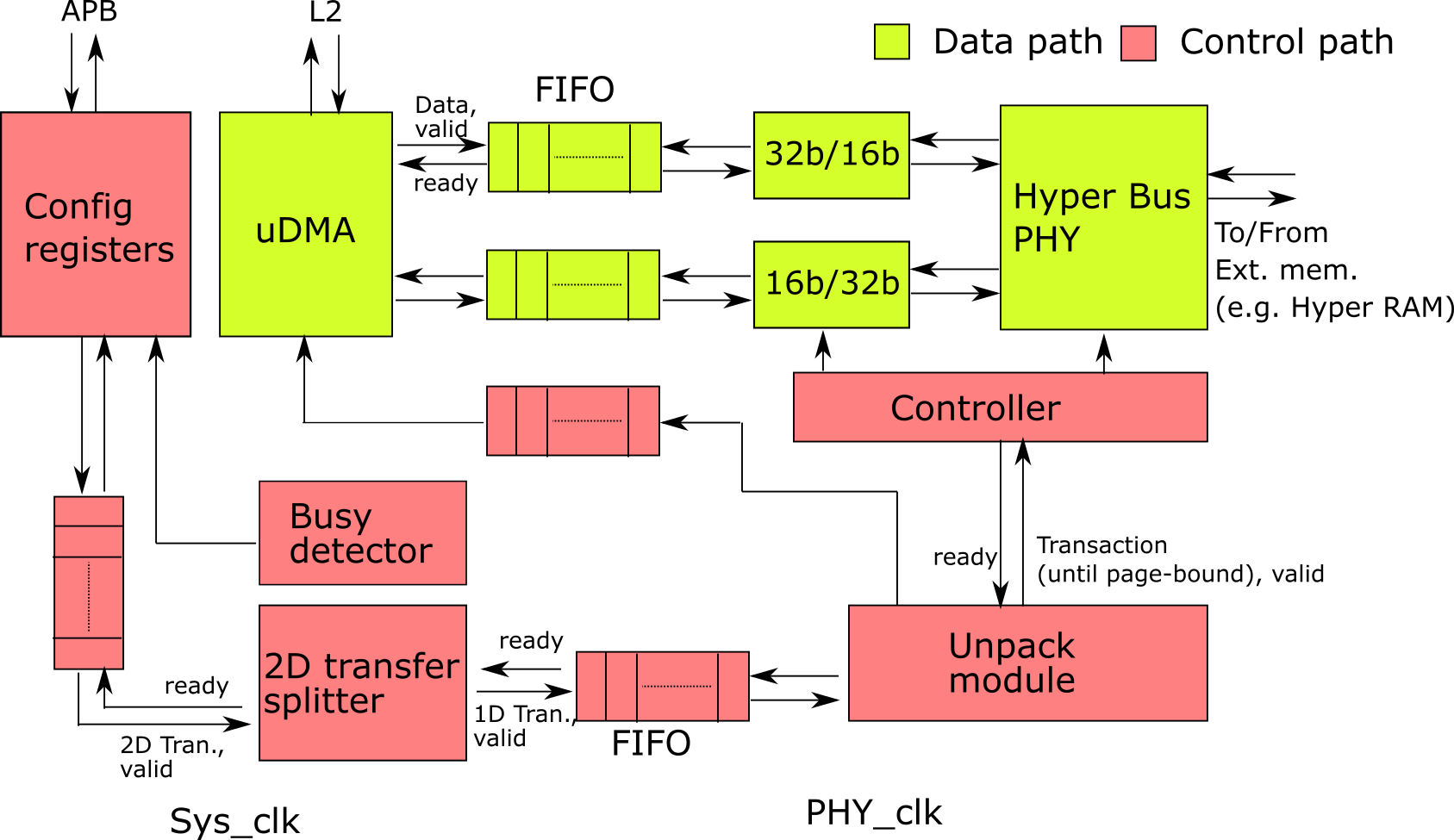


Figure 1 Architectural diagram of the Hyper bus module

Fig. 1 shows an architectural diagram of the Hyper bus module. It is composed of Config registers, the 2D transfer splitter, the unpack module, the controller, the data width modulators (32b/16b, 16b/32b), the FIFOs and the Hyper bus PHY. This module receives/sends data from/to the µDMA which is connected to the L2 memory. The config registers are accessed via the APB bus.

The config registers store the information required by data transfer such as the data size, start address. These information goes to µDMA, which starts communication between the PULP system and external memory modules such as Hyper RAM. Data from/to L2 passes through FIFO, 32b/16b(16b/32b), and the Hyper bus PHY. As the hyper bus protocol conducts its data transfer in 8bit DDR policy, these data path converts the data width between 8bit DDR to 32 bit SDR.

This module supports a 2D transfer capability as shown in the 2D transfer splitter module. Also, the unpack module considers page boundaries of external memory devices. The configuration register also stores whether these modules are activated or not.

The module is operating at various clock domains. The configuration registers and 2D transfer splitter operates at the system clock domain. On the other hand, the unpack module, controller, 16b/32b (32b/16b), PHY operate at PHY\_clk (and a 90 degree shifted clock from PHY\_clk). Asynchronous FIFOs are inserted between the PHY\_clk and Sys\_clk domains for appropriate clock domain crossing.

The Hyperbus compatible memory modules such as the Hyper RAM assign its address per 2 bytes. Nevertheless, this module enables a byte addressing mode as shown in Fig.2. The address seen from the PULP system is assigned for every 8-bit data.

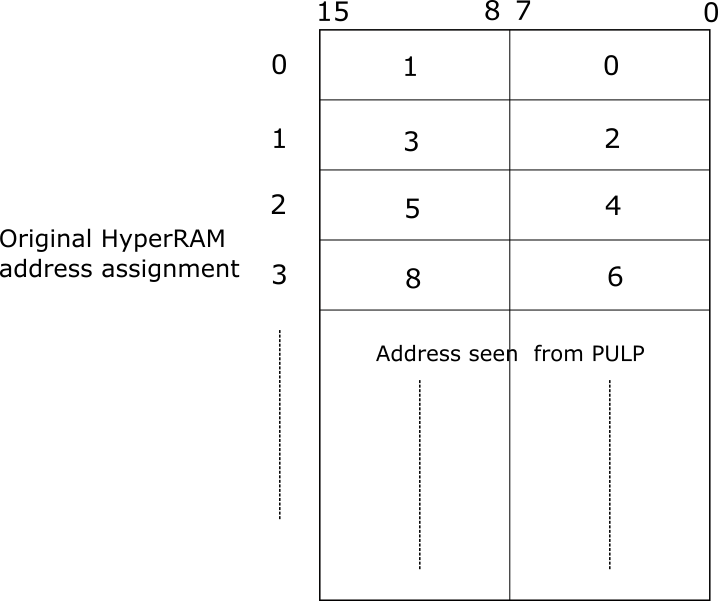


Figure 2 Memory address assignment

## 2D transfer splitter

Roughly speaking, this module divides a 2D data transfer into some pieces of 1D dada transfer. As shown in Fig.3, the 2D transfer requires the entire transfer length (L2D), start address of the memory (SA), the stride length (Lst), and the length of each 1D transfer (L1D). From the start address, a 1D transfer is conducted with the length of L1D and then, the start address of the next 1D transfer is shifted by Lst. The entire 2D transfer is finished when all the data are sent.

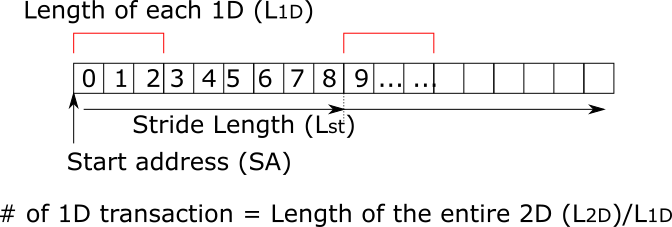


Figure 3 concept of 2D transfer

An architectural diagram of the actual 2D transfer splitter is shown in Fig.3.

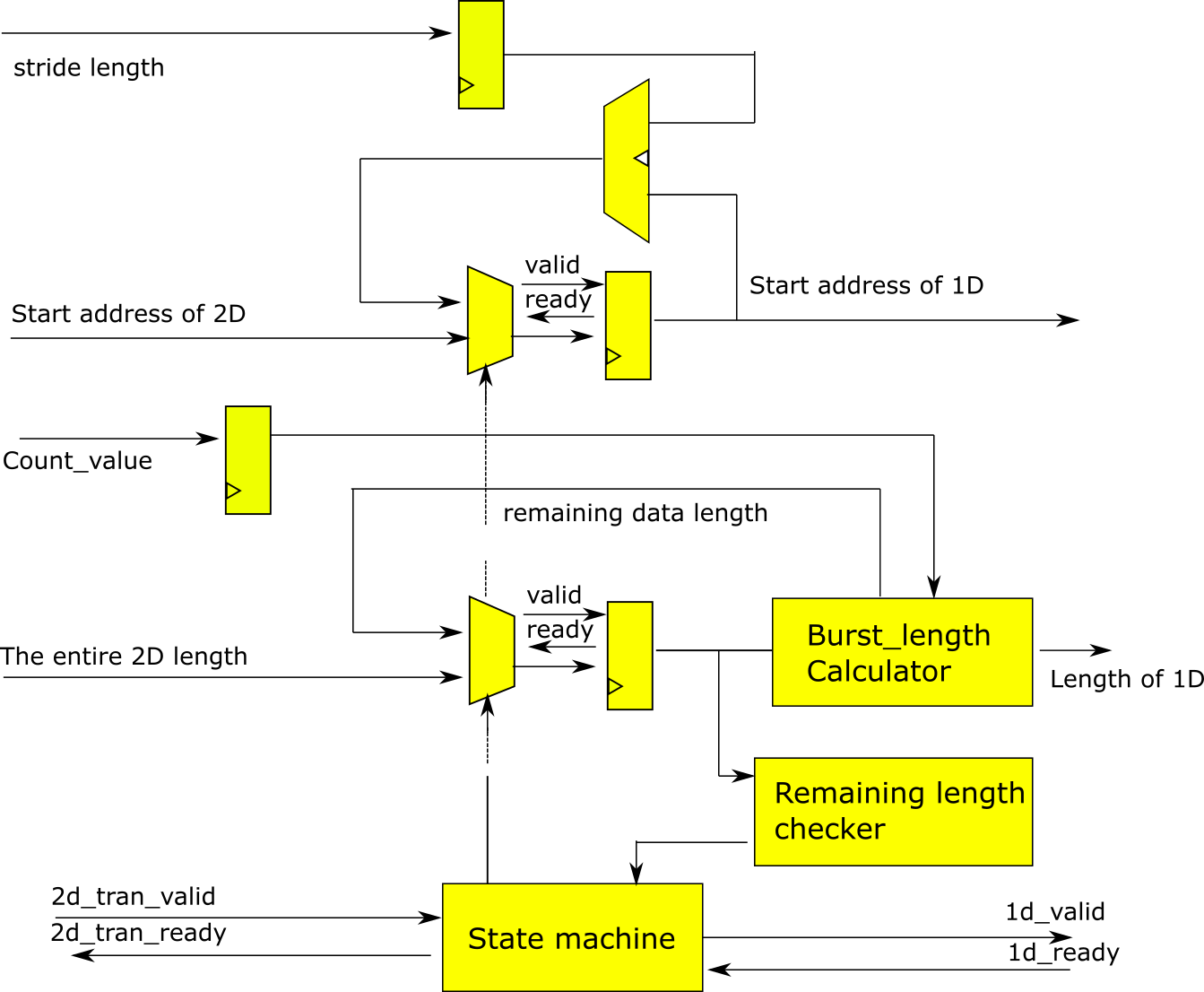


Figure 4 architectural block diagram of 2D transfer splitter

Firstly, the 2D transfer is captured by the registers. Based on this, the burst length calculator generates the 1D burst length with count\_value (L1D) coming from the configuration register and the remaining data length is fed back to the multiplexer. In case that the remaining data length is less than L1D, that remaining value is directly output to the next module. Also, the adder generates the start address of each 1D transaction. Using the previous 1D start address and stride length, the new start address is obtained. State machine generates the valid signal for each 1D transfer, also, control the multiplexer for the start address and length.

## Unpack module

Fig.5 shows a block diagram of the unpack module. This module considers the page boundary in an external memory. For example, if an external memory has page boundaries, this module divides the input transaction to not cross them. Firstly, the input transaction is stored by the registers, then, the burst length is calculated to not cross the boundary. Based on this length, the start address for each transaction is generated. These small pieces of the transactions continue to be generated until the remaining data becomes 0. The state machine generates the valid signal for each transaction.

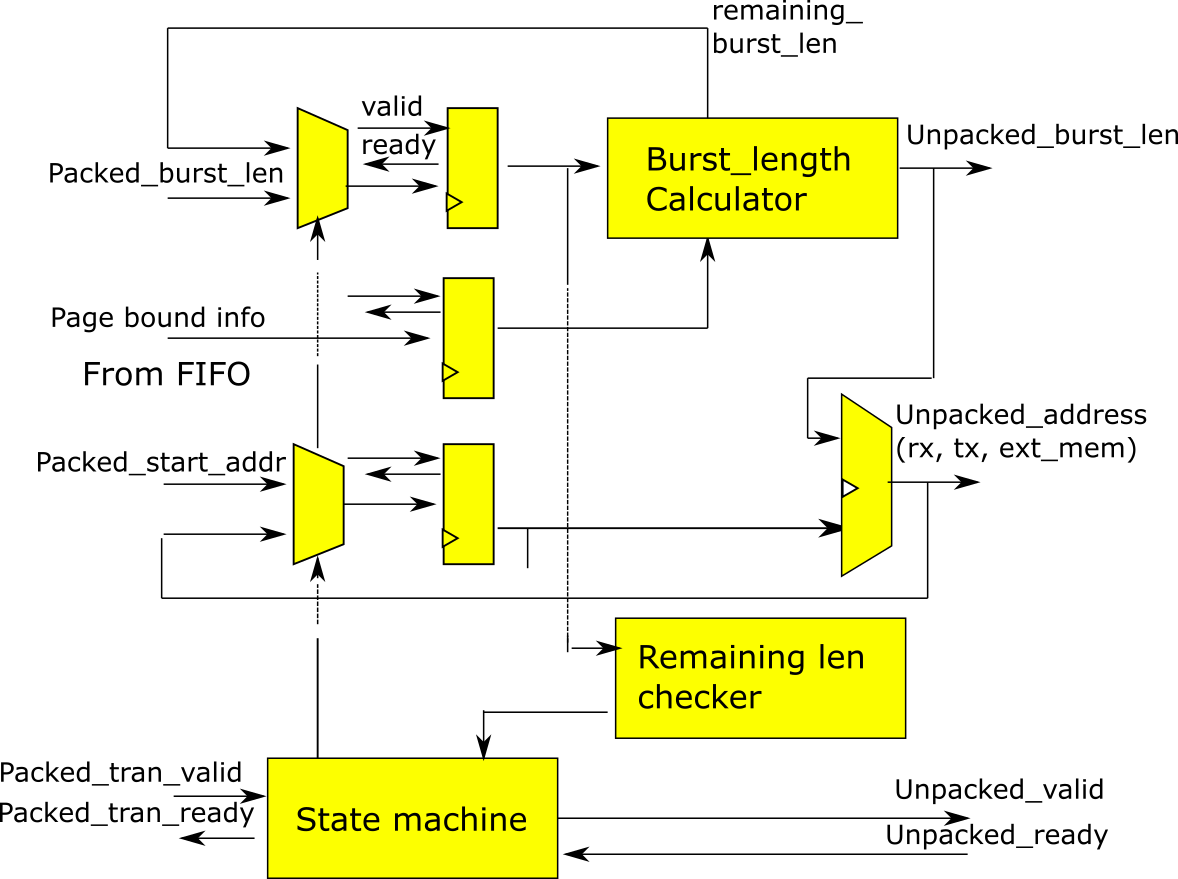


Figure 5 architectural block diagram of the unpack module

## Controller

The controller is a state machine as shown in Fig. 6. According to the input transaction information, its state is changed. The start state accepts the transaction provided by the unpack module. Then, the state becomes SETUP. During the setup, the input transaction is decoded and necessary information for the PHY and the data path (16b/32b 32b/16b modules) are generated. It includes, the number of un-transmitted data length, valid signal for each transaction, and data mask information. After that, the state is changed to one corresponding to the transaction operation. When all the data transfer is finished, the state is changed to end and back to the initial state.

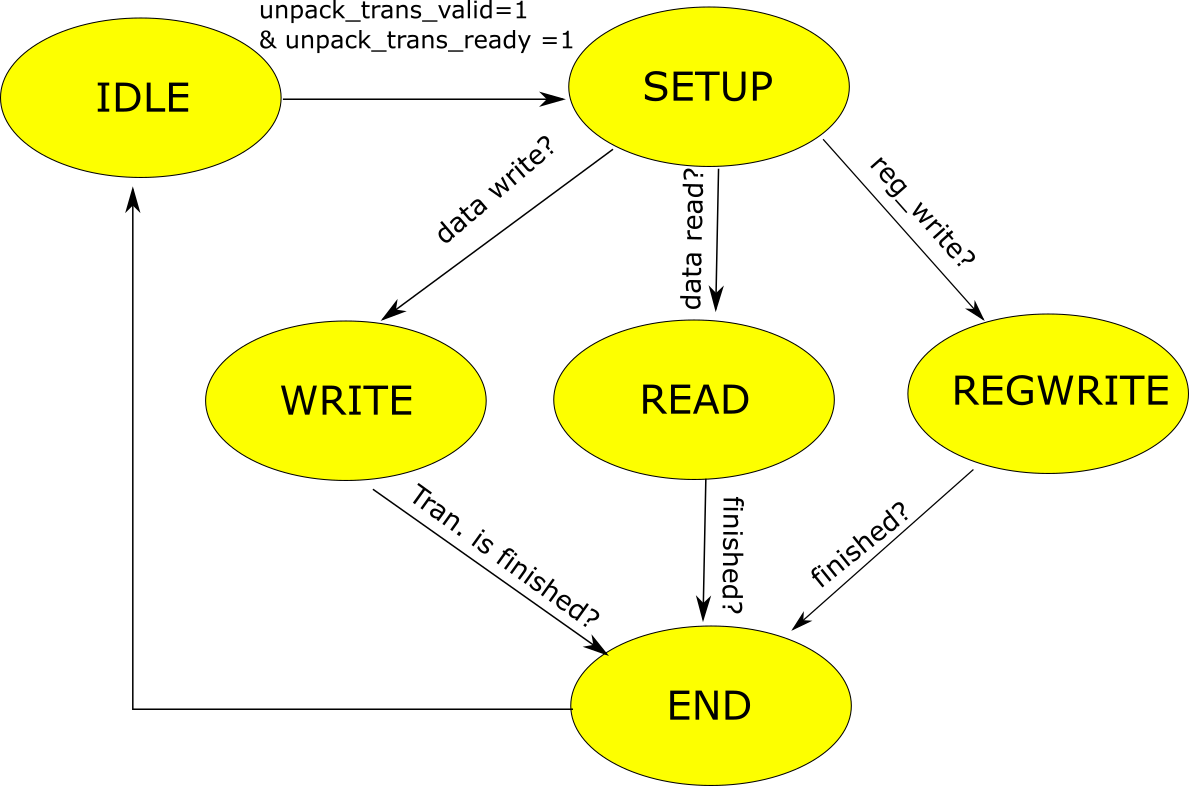


Figure 6 State machine of the controller module

## 32b/16b (tx\_buffer) and 16b/32b (rx\_buffer) modules

The uDMA sends data in 32-bit width SDR while the Hyperbus protocol is 8bit data width DDR. Hence, a data width modulator is necessary. The 32bit width is shorten to 16bit first, then, the PHY converts the 16bit SDR into 8bit DDR. The implemented 32b/16b module is described in Fig. 7. Roughly speaking, the multiplexer selects the16 bit between the upper and lower part of the 32-bit data.

As the external memory originally assign the data address for each 16-bits and data write/read is only conducted for every 16 bits, we need to implement a way that stores/read data starting from the mid 8-bit positions. The basic idea to do this is that, although the PHY sends 16bit data, the lower 8bit of the first 16bit is just masked. Hence, 1 byte shift is conducted by the data rotator.

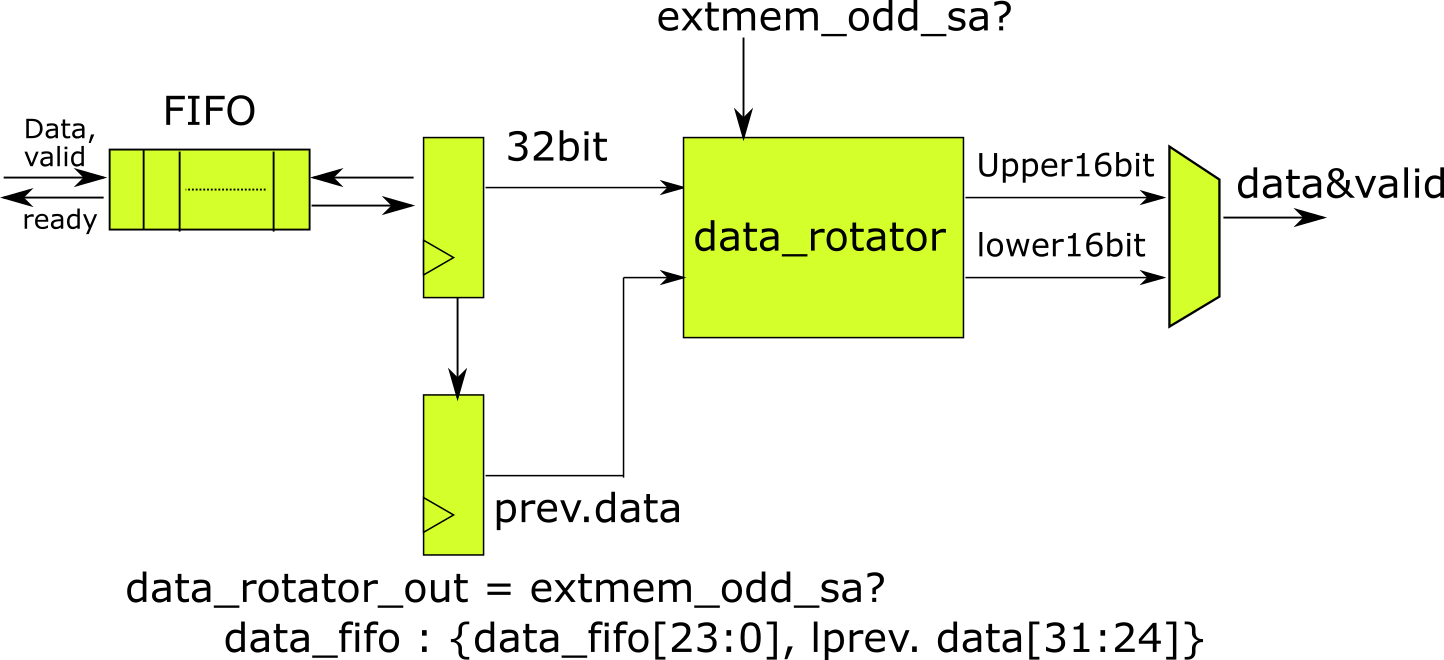


Figure 7 32b/16b module

Similarly to the 32b16b module 16b/32b module generates the 32bit data by accumulating two 16bit data (Fig. 8). Also, if we want to start a read operation with an odd start address seen from the PULP system, the lower 8bit of the first 16bit from the Hyper RAM is meaning less data. Hence data shift mechanism is implemented also in the 16b/32b module.

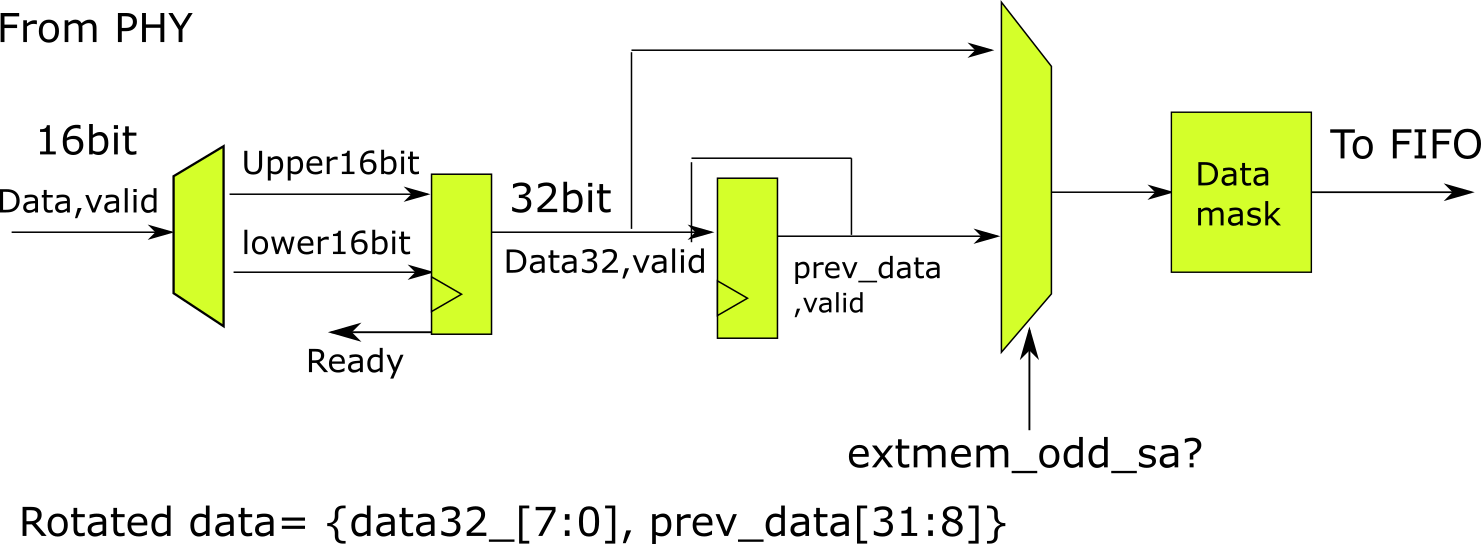


Figure 8 16b/32b module

## Hyper bus phy

Fig. 9 shows an architectural diagram of the Hyperbus PHY. This module sends/receives data to/from the external memory while complying with the Hyperbus protocol. The TX path converts the 16bit SDR data into 8bit DDR. The RX path recover 16bit SDR data from 8bit DDR. The state machine controls the timing when DDR data, the command (such as read/write), and address for the external memory are sent. Also, the chip select signal is generated by the state machine. The data\_mask is used for enabling the byte addressing mode in this module. This data\_mask is converted into rwds in the hyperbus protocol.

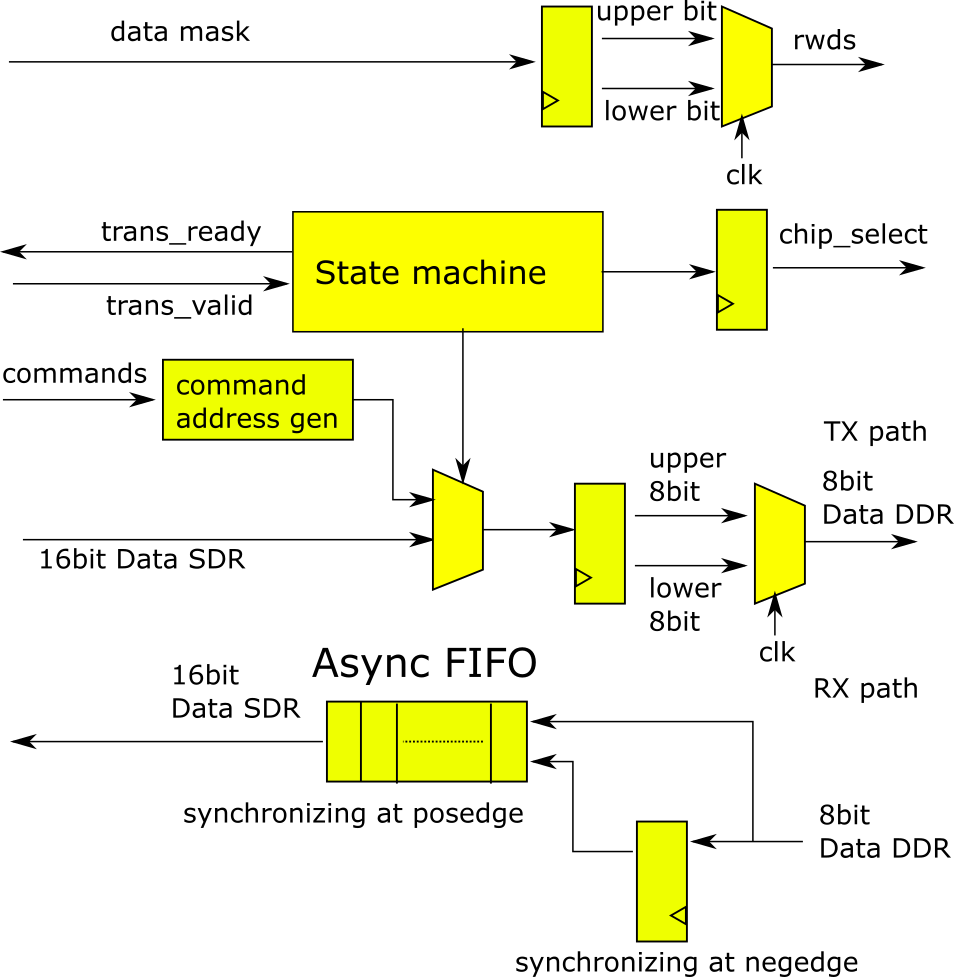
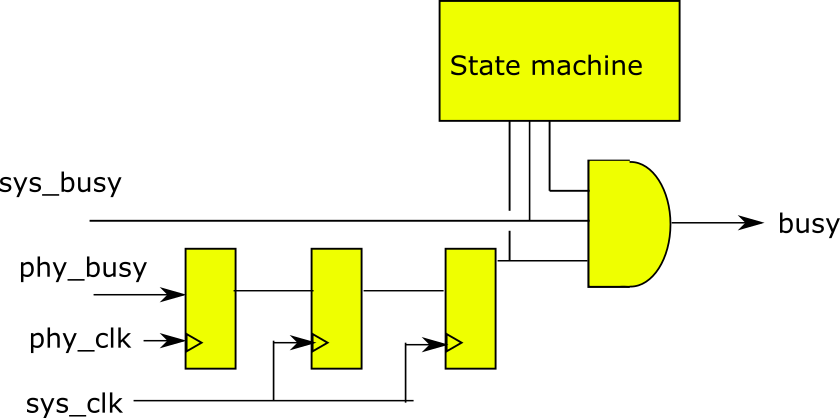


Figure 9 the PHY module

## Busy detector



The busy detector monitors valid ready signals of all the modules. Hence, once a transaction is issued by the configuration register, busy signal is asserted until its all the data transfers are finished. The state machine is used for assuring once sys\_busy is asserted, the busy signal remains “1” until the entire transaction is finished.

Since the busy signal of the phy part is synchronizing at the phy clock, two flip flops are inserted in the clock domain boundary to avoid the metastable state of the phy\_busy signal.

## Clock tree diagram for physical implementation

Fig. 10 depicts a brief clock tree diagram of the hyperbus module. Its clock inputs are the SOC\_CLK and the HYPER\_CLK\_IN. The frequency of the latter is two times faster than the actual PHY clock. The module has two phases (HYPER\_CLK\_0, HYPER\_CLK\_90) of the phy clocks generated by the clk\_gen\_i module.

The configuration register, fifo for the 2-D transfers, 2D transfer splitter, and transaction are operating at the frequency domain of soc\_clk. These are driven by the positive edge of the clock.

Asynchronous FIFOs are used for clock domain crossing between the phy\_domain and soc\_clk domain. So these are synchronizing at positive edge of the two clocks. Also, the busy detector receives these clocks. It has cdc registers inside.

The unpack module, controller, rx\_buffer, tx\_buffer, and the phy are synchronizing at the phy clock domain. All of them except the phy operate with the in phase clock (HYPER\_CLK\_0), while the phy utilizes the two phases. More detailed diagram for the phy is drawn in Fig. 11.

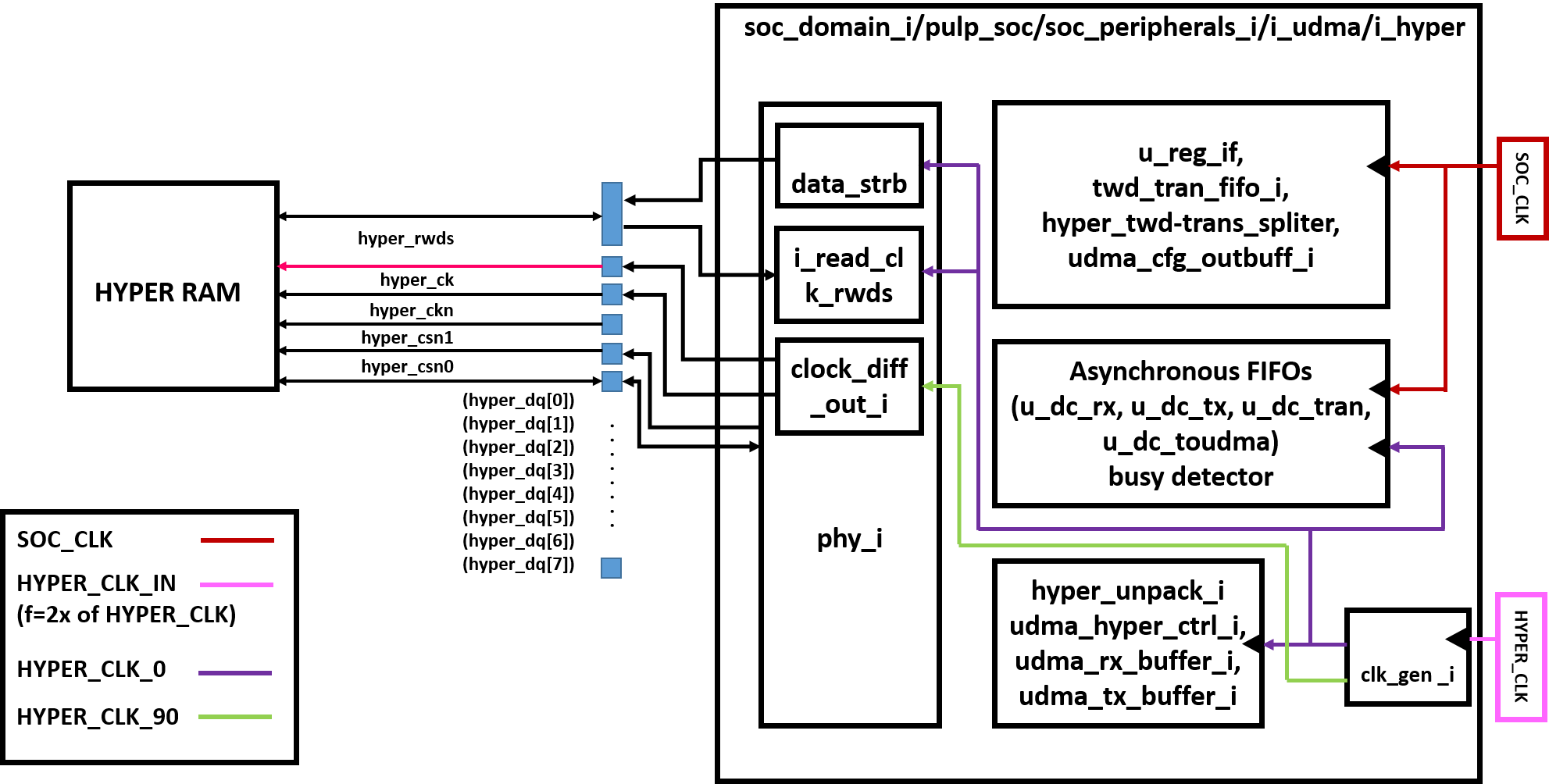


Figure 10 Clock tree diagram

As can be seen in the diagram, the submodules for the data strobe, state machine, and ddr data generator are synchronizing at HYPER\_CLK\_0. The read path, i\_read\_clk\_rwds, receives two clock sources because the hyperbus protocol utilizes rwds signal from the external memory as a clock and this phy operates at HYPER\_CLK\_0. Hence, an asynchronous FIFO is implemented. The rwds signal coming from an external memory is delayed and fed to the FIFO. Chip select and output clock (hyper\_ck) are synchronizing at HYPER\_CLK\_90 while their enable signals are generated by the in-phase domain.

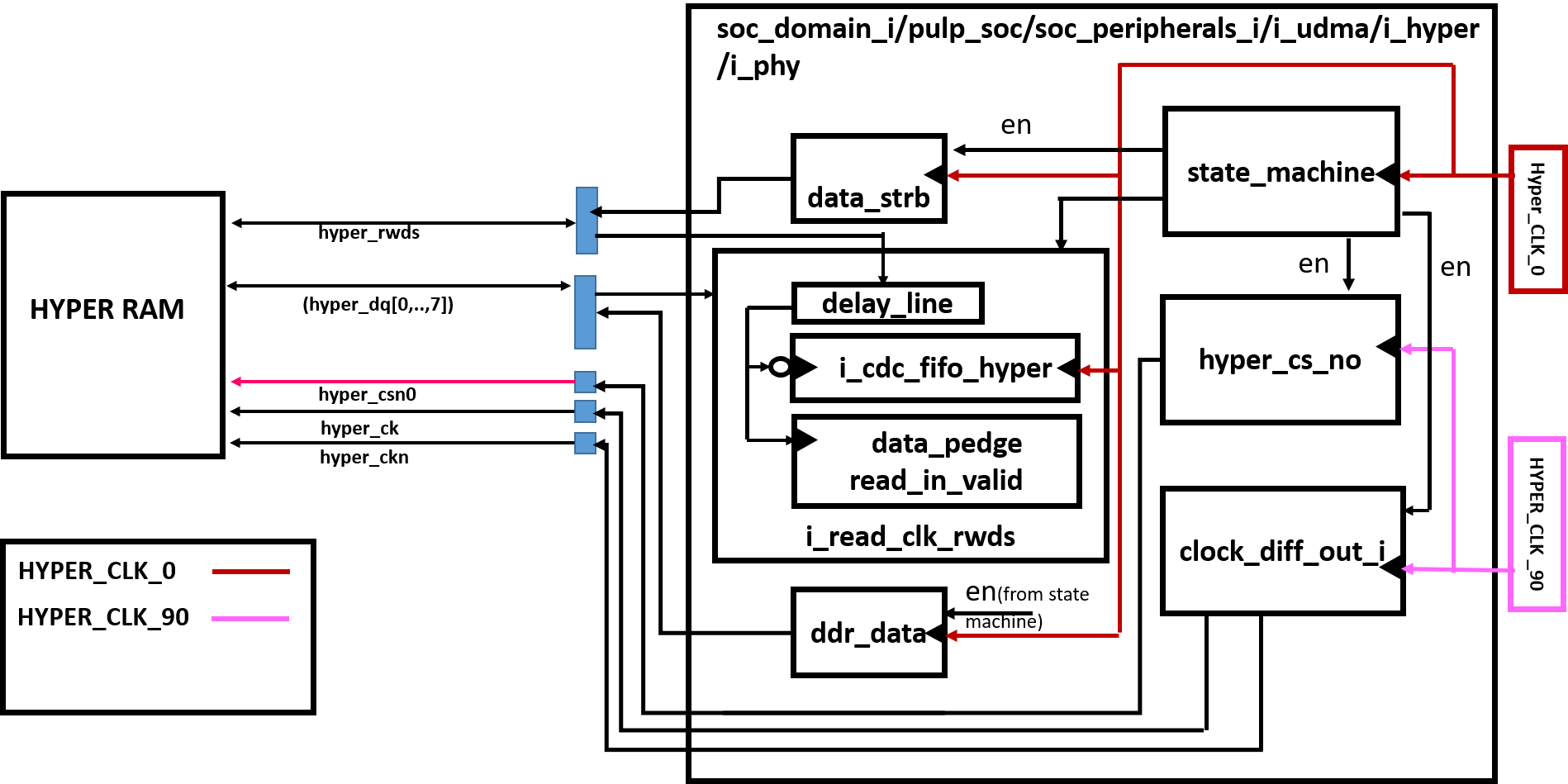


Figure 11 Detailed clock tree diagram of the PHY module

## Hyper Flash compatibility

For data read,

Since Hyper Flash does not have any data mask,

# Register Map

| **Register Name** | **Offset** | **Description** |
| --- | --- | --- |
| REG\_RX\_SADDR | 0x00 | L2 start address for RX region |
| REG\_RX\_SIZE | 0x04 | Size of the software buffer in L2 (in byte) |
| REG\_UDMA\_RXCFG | 0x08 | µDMA configuration data for the RX direction (clr, en, datasize, continuous) |
| REG\_TX\_SADDR | 0x0C | L2 start address for TX region |
| REG\_TX\_SIZE | 0x10 | Size of the total TX data to be sent (in byte) |
| REG\_UDMA\_TXCFG | 0x14 | µDMA configuration data for the TX direction (clr, en, datasize, continuous) |
| HYPER\_CA\_SETUP | 0x18 | Command/Address setup for Hyperbus protocol |
| REG\_HYPER\_ADDR | 0x1C | Start address of the external memory (e.g. Hyper RAM) |
| REG\_PAGE\_BOUND | 0x20 | Page boundary setting for the external memory |
| REG\_T\_LATENCY\_ACCESS | 0x24 | T\_latency\_access |
| REG\_EN\_LATENCY\_ADD | 0x28 | The additional latency of the Hyper bus protocol is activated |
| REG\_T\_CS\_MAX | 0x2C | Maximum cycle counts for negating the chip select |
| REG\_T\_RW\_RECOVERY | 0x30 | T\_ready\_write\_recovery |
| REG\_T\_RWDS\_DELAY\_LINE | 0x34 | Configuration for the delay line |
| REG\_T\_VARI\_LATENCY | 0x38 | Cycle counts for capturing the input rwds signal |
| N\_HYPER\_DEVICE | 0x3C | Reserved for the future extension |
| REG\_HYPER\_CFG | 0x40 | Data to be written in the Hyper RAM configuration registers |
| MEM\_SEL | 0x48 | Reserved for the future extension |
| TWD\_ACT | 0x4C | Enabling the 2D transfer functionality |
| TWD\_COUNT | 0x50 | Size of the 2D transfer count in byte |
| TWD\_STRIDE | 0x54 | Stride value of the 2D transfer |

Table 1: configuration register map

# Reg Field

**REG\_RX\_SAADDR**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Reserved | | | Start address(RX) | | | | | | | | | | | | | |

**REG\_RX\_SIZE**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| RX data size to be sent to external memory modules in bytes | | | | | | | | | | | | | | | |

**REG\_UDMA\_RX\_CFG**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|  | | | | | | | | | | CLR | EN |  | Data size | | continuous |

Data size is fixed to 32bit. Continuous mode is not supported.

**REG\_TX\_SAADDR**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Reserved | | | Start address (TX) | | | | | | | | | | | | | |

**REG\_TX\_SIZE**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| TX data size to be sent to external memory modules in bytes | | | | | | | | | | | | | | | |

**REG\_UDMA\_TX\_CFG**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Reserved | | | | | | | | | | CLR | EN | Reserved | | | continuous |

\*Data size is fixed to 32 bit.

**HYPER\_CA\_SETUP**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Reserved | | | | | | | | | | | | | rw | Addr\_space | Burst\_type |

Rw 0: write operation 1: read operation

Addr\_space 0: memory array 1: register space

Burst\_type 0: wrapped burst (Not supported) 1: Linear burst

**REG\_HYPER\_ADDR**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| SA of external memory | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| SA of external memory | | | | | | | | | | | | | | | |

**REG\_PAGE\_BOUND**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Reserved | | | | | | | | | | | | | Page\_bound\_length | | |

000: 128 bytes, 001: 256 bytes, 010: 512 bytes, 011: 1024 bytes, Others: no\_boundary

**REG\_T\_LATENCY\_ACCESS**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| T\_latency\_access | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| T\_latency\_access | | | | | | | | | | | | | | | |

The latency count of the Hyper Bus protocol is defined. Default value is 6 which allows the data transfer up to 166MHz of the operational frequency.

**REG\_EN\_LATENCY\_ADD**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Reserved | | | | | | | | | | | | | | | En |

En 0 : It depends on Hyper bus rwds signal whether or not the additional latency of the hyper bus protocol is added

1: force the module to have the additional latency

**REG\_T\_CS\_MAX**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Maximum cycle counts for negating the chip select signal | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Maximum cycle counts for negating the chip select signal | | | | | | | | | | | | | | | |

**REG\_T\_RW\_RECOVERY**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Cycle counts for T read write recovery of the Hyper bus protocol | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Cycle counts for T read write recovery of the Hyper bus protocol | | | | | | | | | | | | | | | |

**REG\_T\_RWDS\_DELAY\_LINE**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Configuration for the delay line | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Configuration for the delay line | | | | | | | | | | | | | | | |

This register is not supported in the FPGA emulation.

**REG\_T\_VARI\_LATENCY**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Cycle counts for capturing the input rwds signal | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Cycle counts for capturing the input rwds signal | | | | | | | | | | | | | | | |

When its variable latency functionality is activated, the Hyper bus protocol requires rwds inputs for checking whether the additional latency is required or not. This register adjusts the timing for capturing this signal.

**REG\_Hyper\_CFG**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Data to be sent to configuration registers of external memory modules | | | | | | | | | | | | | | | |

This register stores a configuration data which is sent to external memory modules. For example, in case of Hyper RAM, the configuration registers 0/1 are written with this 16 bit data.

**TWD\_ACT**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Reserved | | | | | | | | | | | | | | | En |

En 0: 2D transaction is disabled 1: 2D transaction is activated

**TWD\_COUNT**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 2D count length (in byte) | | | | | | | | | | | | | | | |

**TWD\_STRIDE**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 2D stride length (in byte) | | | | | | | | | | | | | | | |

# APIs

**Void udma\_hyper\_setup()**

Hyper bus module initialization such as activating the clock

**Void udma\_hyper\_sleep()**

Deactivating the hyper bus module clock

**Void udma\_hyper\_dwrite(unsigned int len, unsinged int ext\_addr, unsigned int l2\_addr, unsigned int page\_bound)**

Linear write is conducted, len is the entire burst length in bytes, ext\_addre is the start address of the external memory. l2\_addr is the start address of the L2 memory. page\_bound defines the page boundary of the external memory.

page\_bound can be selected from 128, 256, 512, 1024. Any other value will set the no boundary.

**Void udma\_hyper\_dread(unsigned int len, unsinged int ext\_addr, unsigned int l2\_addr, unsigned int page\_bound)**

Linear read is conducted, len is the entire burst length in bytes, ext\_addre is the start address of the external memory. l2\_addr is the start address of the L2 memory. page\_bound defines the page boundary of the external memory.

page\_bound can be selected from 128, 256, 512, 1024. Any other value will set the no boundary.

**int udma\_hyper\_busy ()**

This function can check whether or not any transactions are conducted in the hyper bus module. If it is busy, 1 is returned. Otherwise, the result is 0.

**Void udma\_hyper\_wait()**

This function suspends a program until udma\_hyper\_busy() becomes 0.

**A sample program with the APIs is shown below**

==================================================================

#include <stdio.h>

#include <rt/rt\_api.h>

#include </rt/rt\_freq.h>

#include <stdint.h>

#include </hyperbus\_test.h>

#define BUFFER\_SIZE 64

int main() {

int tx\_buffer[BUFFER\_SIZE],rx\_buffer[BUFFER\_SIZE];

int a;

int \*p;

int hyper\_addr;

udma\_hyper\_setup();

printf(" current frequency %d \n", \_\_rt\_freq\_periph\_get());

for (int i=0; i< (BUFFER\_SIZE); i++)

{

tx\_buffer[i] = 0xffff0000+i;

}

hyper\_addr = 1;

udma\_hyper\_dwrite((BUFFER\_SIZE\*4), hyper\_addr, (unsigned int)tx\_buffer, 0);

printf("BUSY: %d \n", udma\_hyper\_busy());

udma\_hyper\_wait();

printf("BUSY: %d \n", udma\_hyper\_busy());

udma\_hyper\_dread((BUFFER\_SIZE\*4), hyper\_addr, (unsigned int)rx\_buffer, 0);

udma\_hyper\_wait();

printf("BUSY: %d \n", udma\_hyper\_busy());

for (int i=0; i< BUFFER\_SIZE; i++)

{

printf("rx\_buffer[%d] = %x \n", i, rx\_buffer[i]);

}

return 0;

}

==================================================================