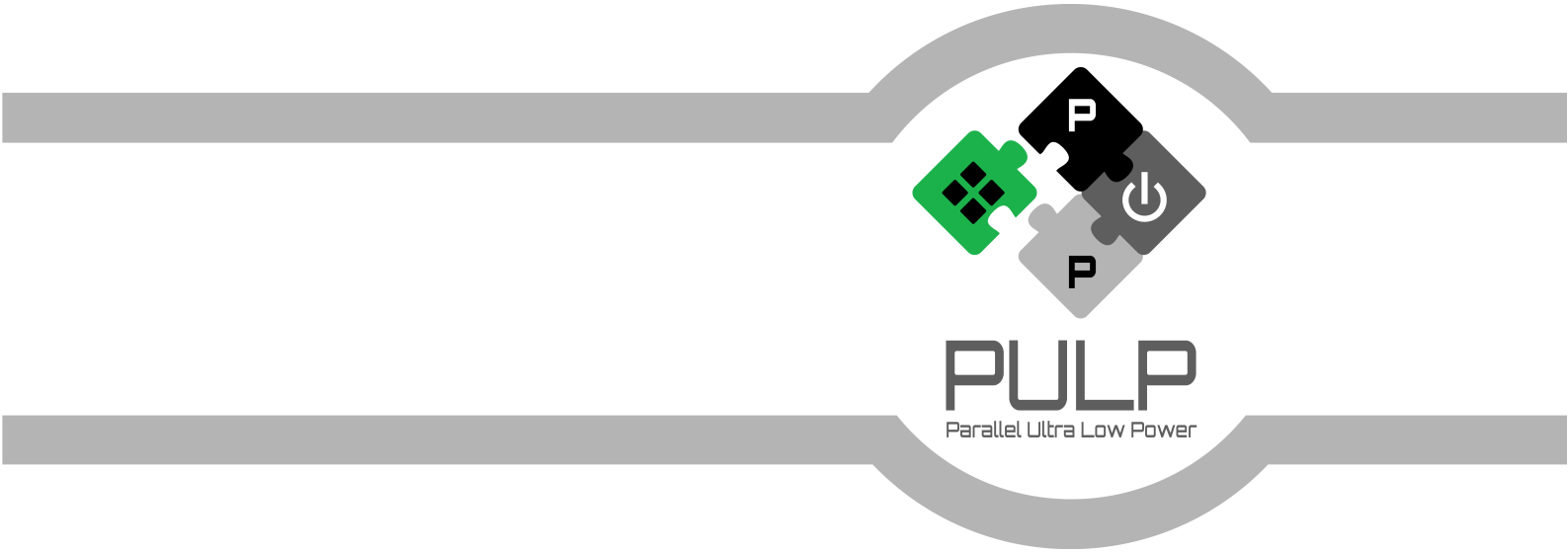


Hyper Bus Module: User Manual

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

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

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.

1.1 Scope and Purpose

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

2 Architecture

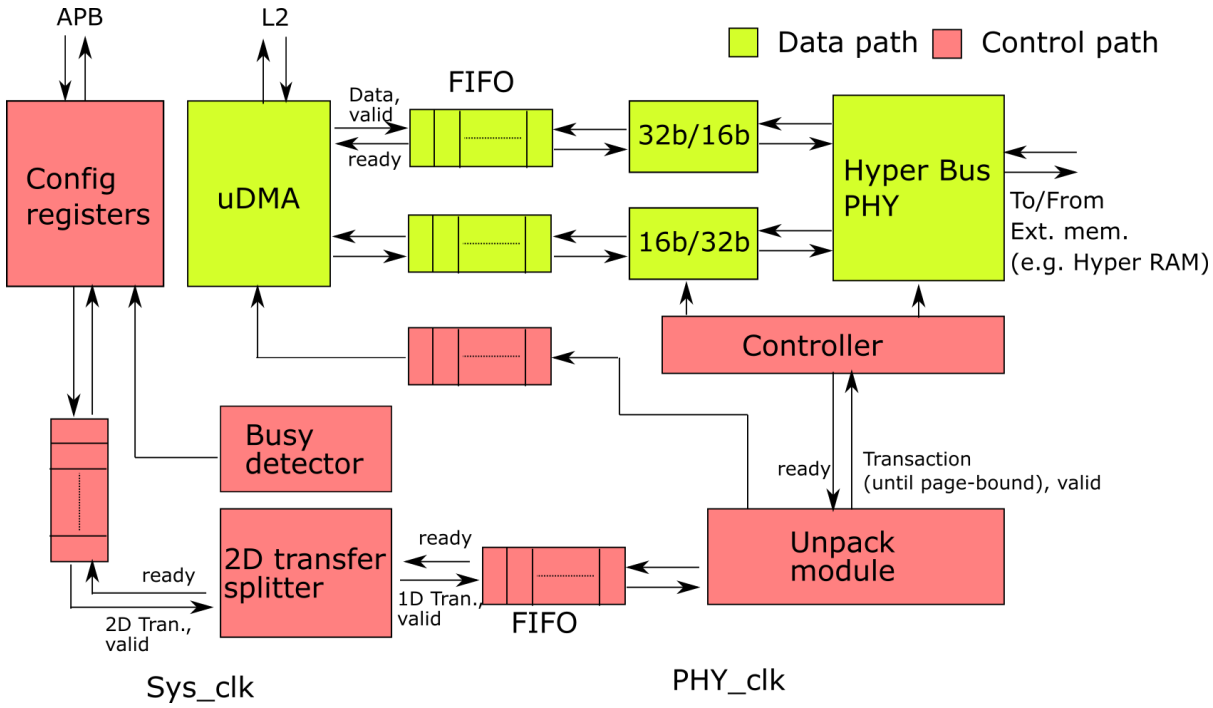


Figure 1 Architectural diagram of the Hyper bus module

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.

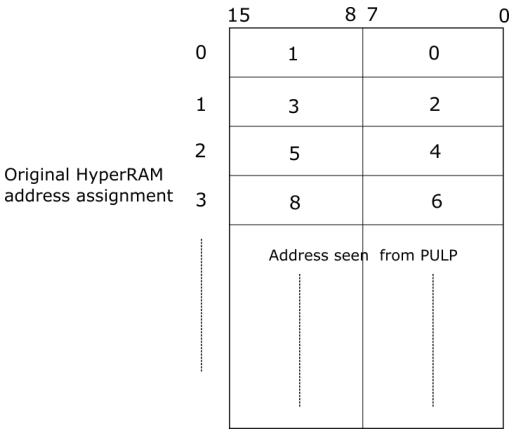


Figure 2 Memory address assignment

2.1 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 (L_{2D}), start address of the memory (SA), the stride length (L_{st}), and the length of each 1D transfer (L_{1D}). From the start address, a 1D transfer is conducted with the length of L_{1D} and then, the start address of the next 1D transfer is shifted by L_{st}. The entire 2D transfer is finished when all the data are sent.

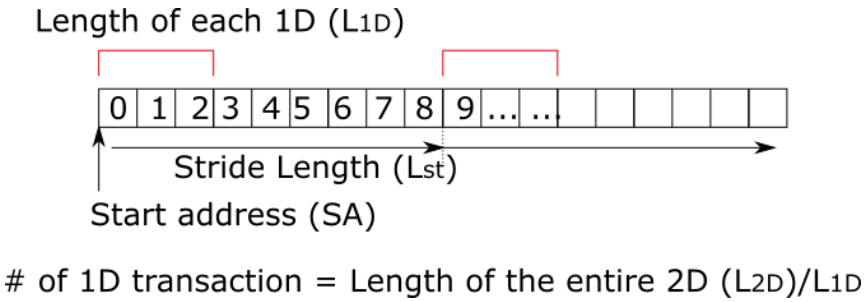


Figure 3 concept of 2D transfer

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

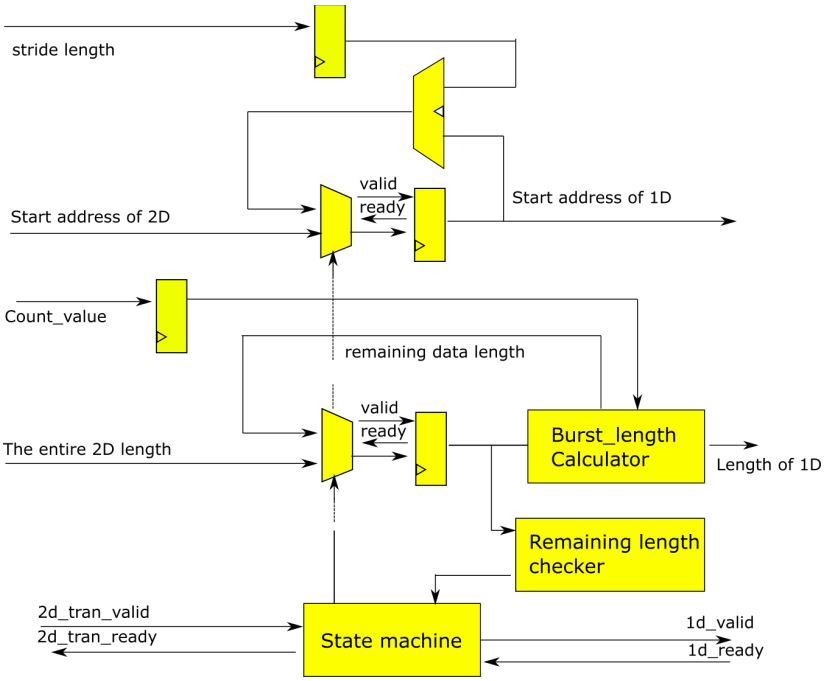


Figure 4 architectural block diagram of 2D transfer splitter

Firstly, the 2D transfer is captured by the registers. Based on this, the burst length calculator generates the 1D burst length with count_value (L_{1D}) 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 L_{1D}, 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.

2.2 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.

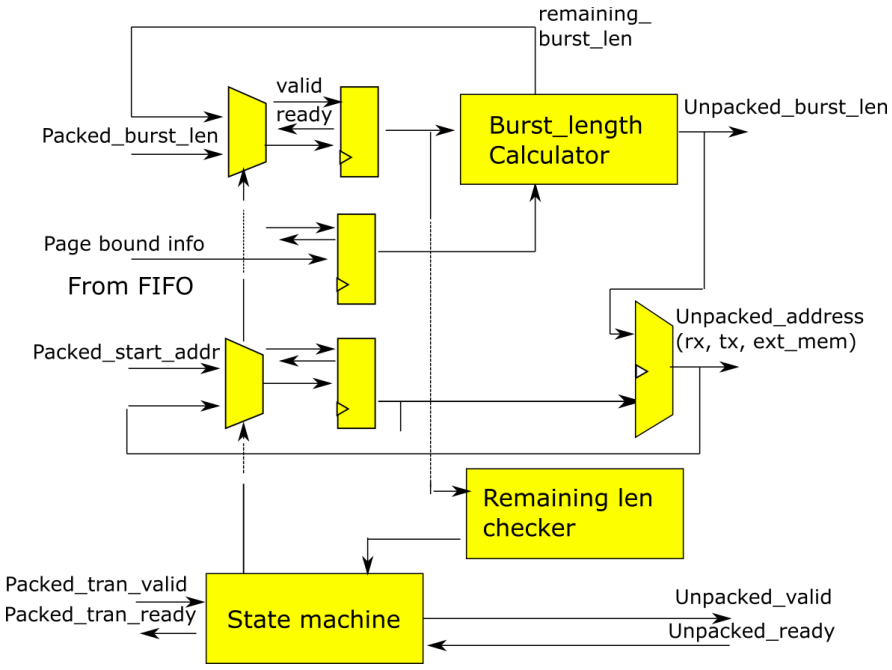


Figure 5 architectural block diagram of the unpack module

2.3 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.

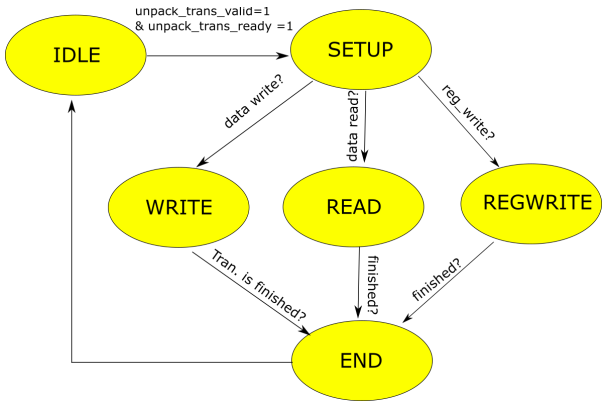


Figure 6 State machine of the controller module

2.4 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.

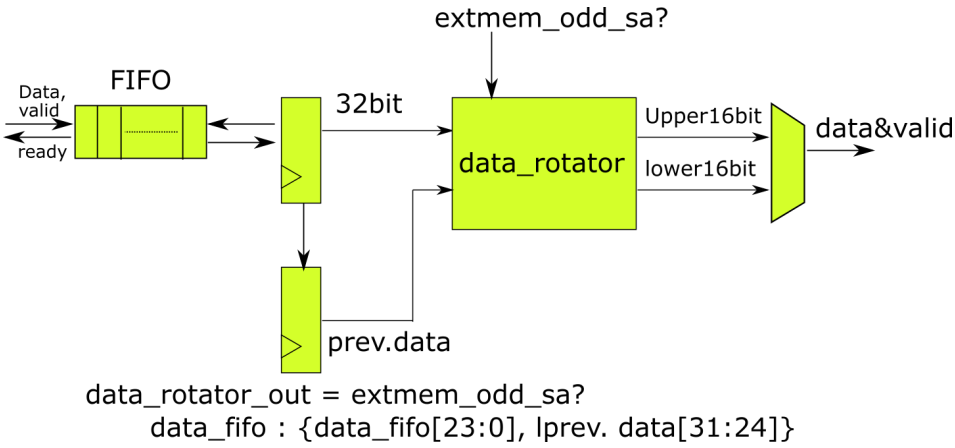


Figure 7 32b/16b module

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.

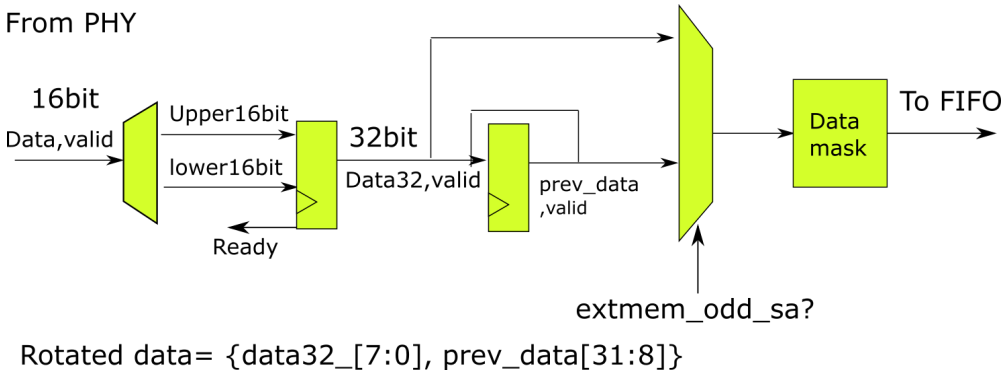


Figure 8 16b/32b module

2.5 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.

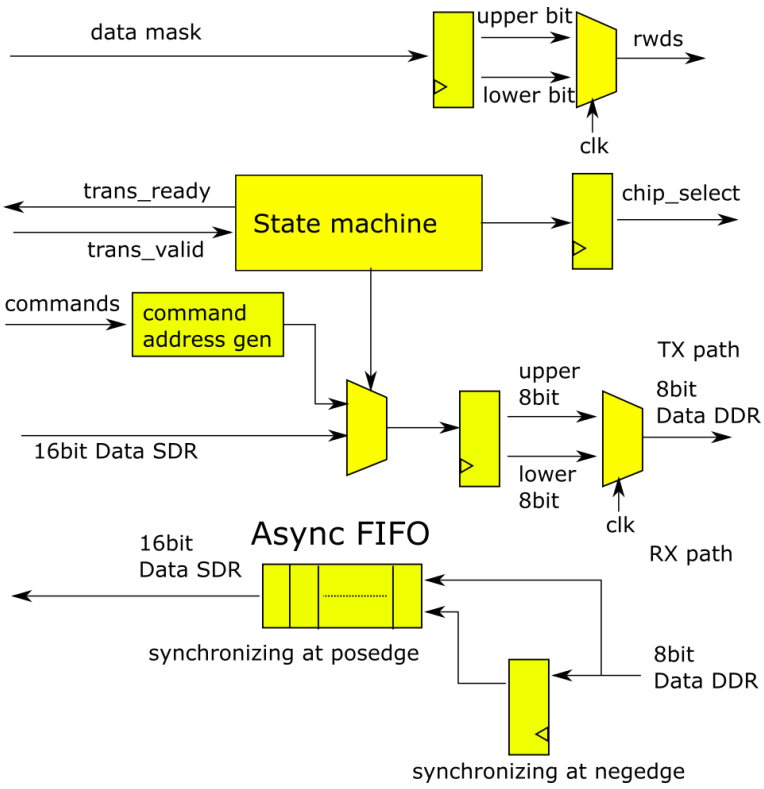
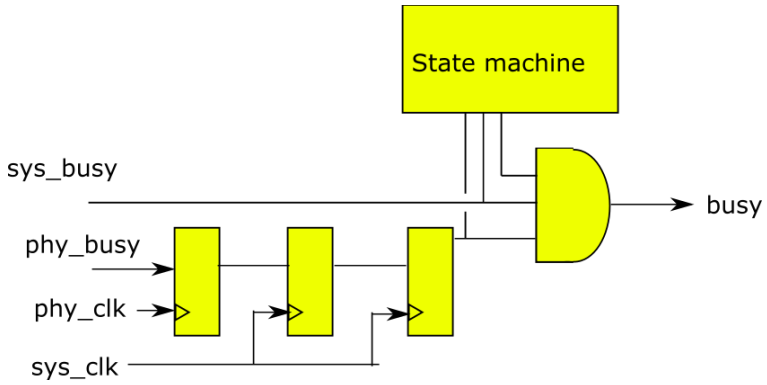


Figure 9 the PHY module

2.6 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.

2.7 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.

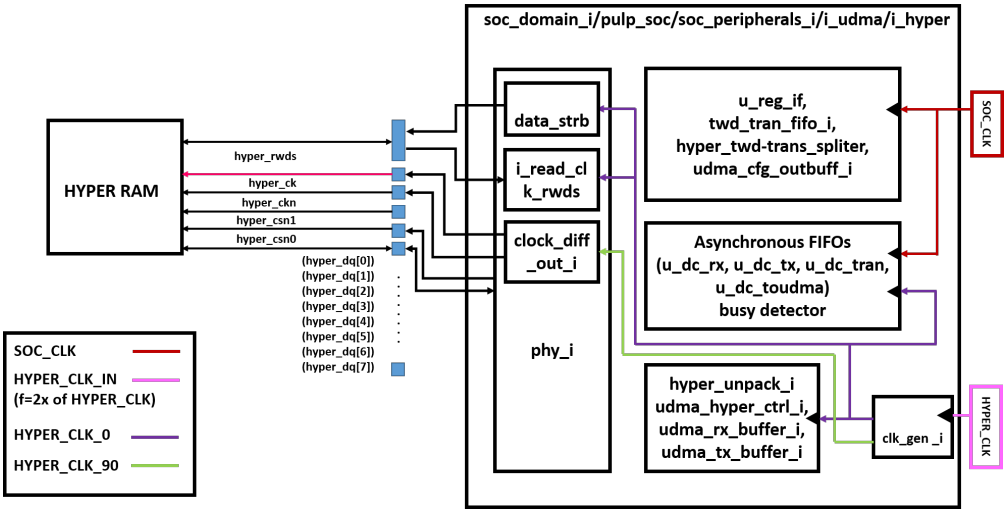


Figure 10 Clock tree diagram

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.

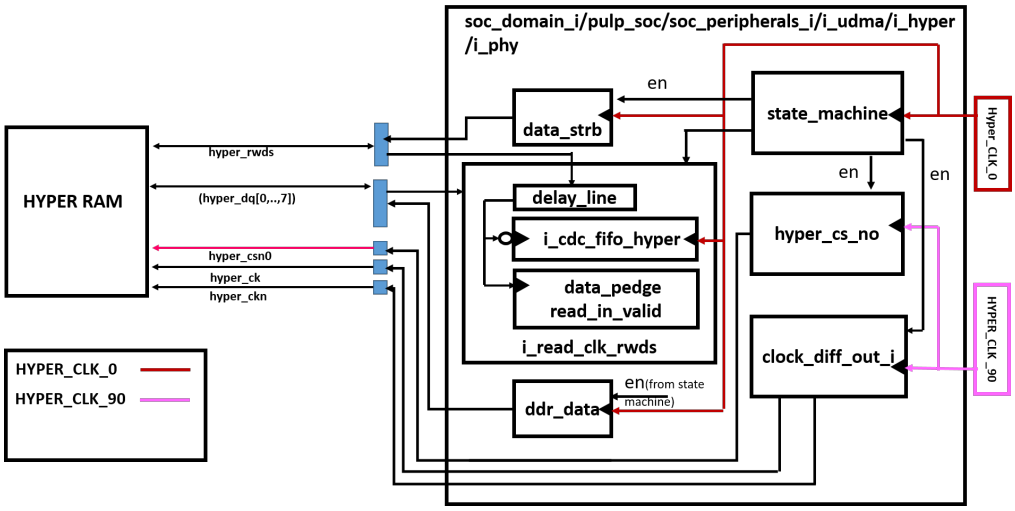


Figure 11 Detailed clock tree diagram of the PHY module

2.8 Hyper Flash compatibility

For data read,

Since Hyper Flash does not have any data mask,

3 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

4 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)															

5 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;
}
```

==========