POLITECNICO DI TORINO

Dipartimento di Elettronica e Telecomunicazioni

Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea Magistrale

Design of FPGA IP for modular architectures on VirtLAB board

Relatore: prof. Massimo Ruo Roch

Laureando: Andrea Bononi

ACKNOWLEDGEMENTS

CONTENTS

ACRONYMS

CR0 Configuration Register 0

CR1 Configuration Register 1

DDR Double Data Rate

DUT Device Under Test

FPGA Field Programmable Gate Array

FSM Finite State Machine

IP Intellectual Property

MCU MicroController Unit

RAM Random Access Memory

SDR Single Data Rate

VCR Virtual Configuration Register

INTRODUCTION

The recent SARS-CoV2 pandemic put a great strain on university courses. Despite the access to physical infrastructures was prohibited, videoconferencing and recorded videos allowed to proceed with the lectures without too many troubles. However, engineering teaching should also involve real laboratory experiences to provide students fundamental skills. When it comes to electronic lessons, it was usually not possible to provide the students the majority of the required instruments (such as digital oscilloscopes, signal generators and spectrum analyzers) given their high cost.

In this scenario, a low-cost experimental printed circuit board, namely the VirtLAB board, was developed at Politecnico di Torino to provide electronic students access to physical devices. Its architecture can be divided in two main sections [1]:

- **User section**: it contains an MCU (STM32L496) and an FPGA (Intel Cyclone 10 LP), which can be easily programmed by the students for educational purposes, together with some LEDs and some switches.
- **Master section**: it contains an MCU (STM32L496), an FPGA (Intel Cyclone 10 LP) and two external memories (a HyperRAM and a QSPI flash) to be used as generic data storage. From the point of view of a student, this side comes already programmed to provide a virtual replacement of the bench equipment.

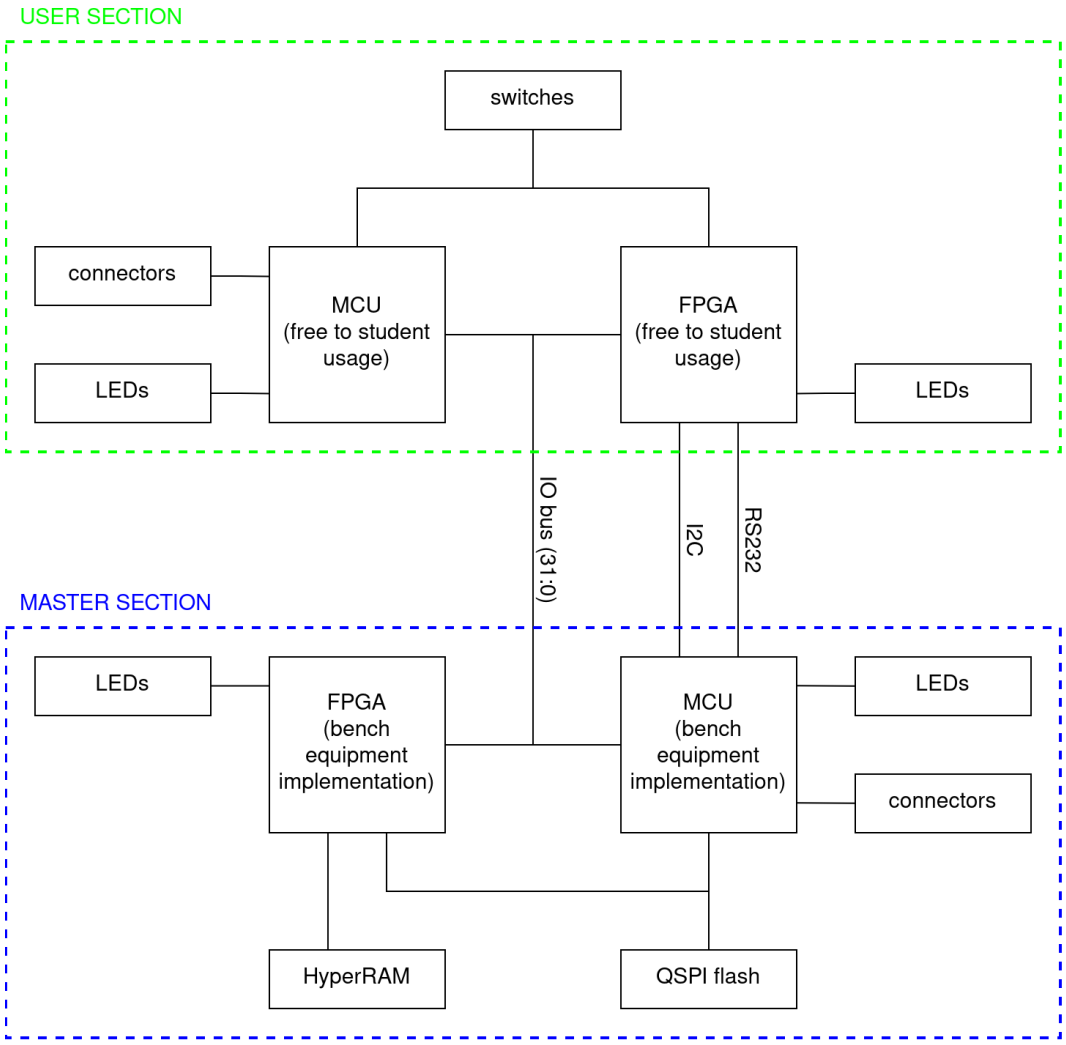

Figure 1.1: VirtLAB board block diagram

Currently, it is still not possible to exploit the master section to its full potential. In particular, the FPGA might used to implement several useful applications it may realize. In this regard, the best approach would be to create modular architectures using generic IP cores that share a common communication protocol, namely the Intel Avalon interface. In this way, it is possible to make full use of the features provided by the Intel CAD software:

- Several general-purpose IP cores with an Intel Avalon interface are already provided by Intel, such as on-chip memories, processors and so on.
- The Intel CAD software is able to automatically create the interconnection logic among IP cores that use an Intel Avalon interface.

At the moment, it is not possible to create an Avalon-based modular architecture able to communicate with any of the external memory storage devices. Indeed, both the HyperRAM interface and the QSPI interface are quite different from the Intel Avalon interface. This paper deals with the design, developement and testing of a custom IP core able to convert the HyperRAM interface into an Intel Avalon interface, so that it can be easily managed by any modular architecture. The whole document refers to a HyperRAM model S27KL0641DA, i.e. the exact model employed in the VirtLAB board.

SPECIFICATIONS

2.1 Avalon Memory Mapped Interface

The Avalon interface family defines different interfaces for different applications. What really matters for our purposes is the Avalon Memory Mapped interface, an address-based read/write interface typical of Host-Agent connections.

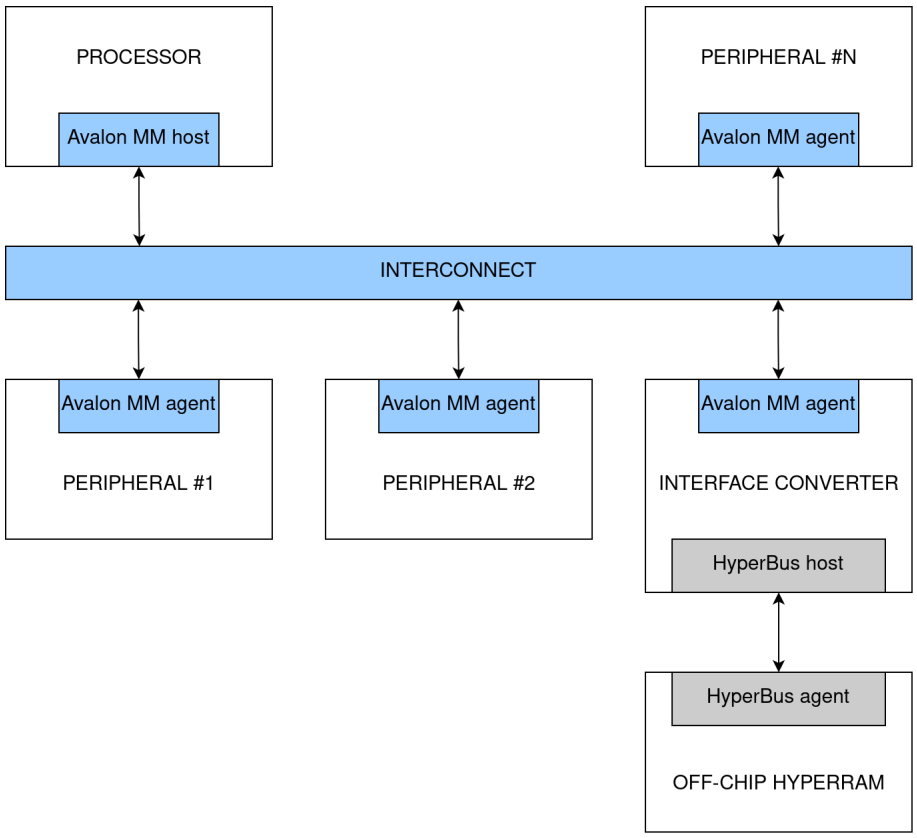

Figure 2.1: Typical Host-Agent system using components with an Avalon Memory Mapped interface (highlighted in light blue). The HyperRAM can be connected to the system only by using a suitable interface converter.

The Avalon Memory Mapped interface includes some always-required signals and several optional signals that might be useful depending on the peripheral. In our case, some specific considerations have to be taken into account:

- In general, the number of clock cycles required to read/write the HyperRAM is variable.
- The system must support burst operations.

Consequently, the Avalon Memory Mapped interface must include the following signals:

- *address*: the address to work with.
- *read*: it is asserted to indicate a read transfer.
- *write*: it is asserted to indicate a write transfer.
- *readdata*: the data read from the agent as a result of a a read transfer.

- *writedata*: the data to be written during a write transfer.
- *readdatavalid*: when asserted, it indicates that the readdata signal contains a valid data.
- *burstcount*: it indicates the number of transfers of a burst operation.
- *waitrequest*: it is asserted by the agent when it is unable to respond to a read/write request.

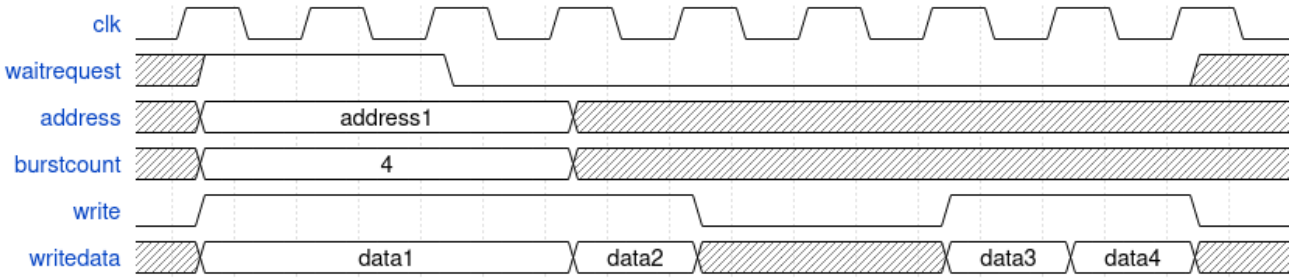

Figure 2.2: Avalon Memory Mapped interface - write operation timing diagram

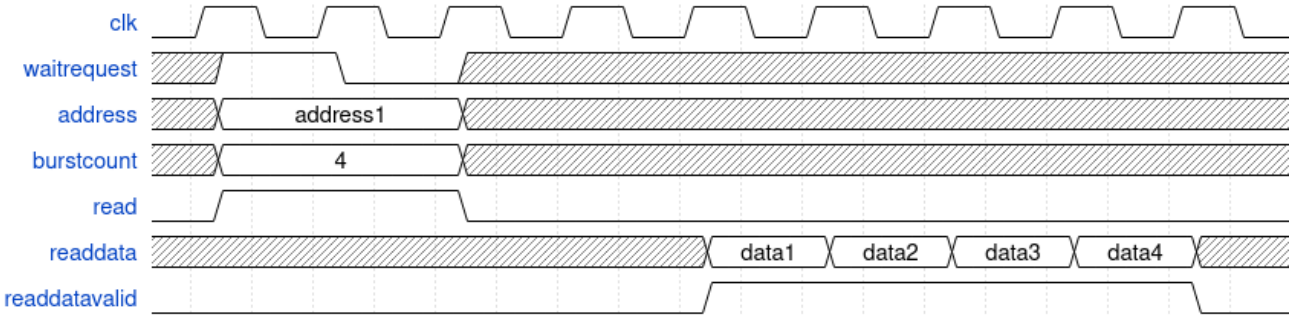

Figure 2.3: Avalon Memory Mapped interface, read operation timing diagram

2.2 HyperRAM Interface Specifications

The HyperRAM interface is based on an 8-bit DDR data bus used to transfer data, addresses and commands. The memory contains a couple of configuration registers that can be written in the same way as the memory locations, but using dedicated addresses. The interface includes the following signals:

- *CK*, *CK#* : differential clock.
- *RESET#* : active-low hardware reset.
- *CS#* : active-low chip select.
- *DQ*: 8-bit IO bus for data, addresses and commands.
- *RWDS*: read/write data strobe with the following functionality:
  - During a read data transfer it is edge-aligned with DQ and it can be used to sample it.
  - During a write data transfer it works as data masking signal.
  - During a command transfer it indicates if additional latency is required.

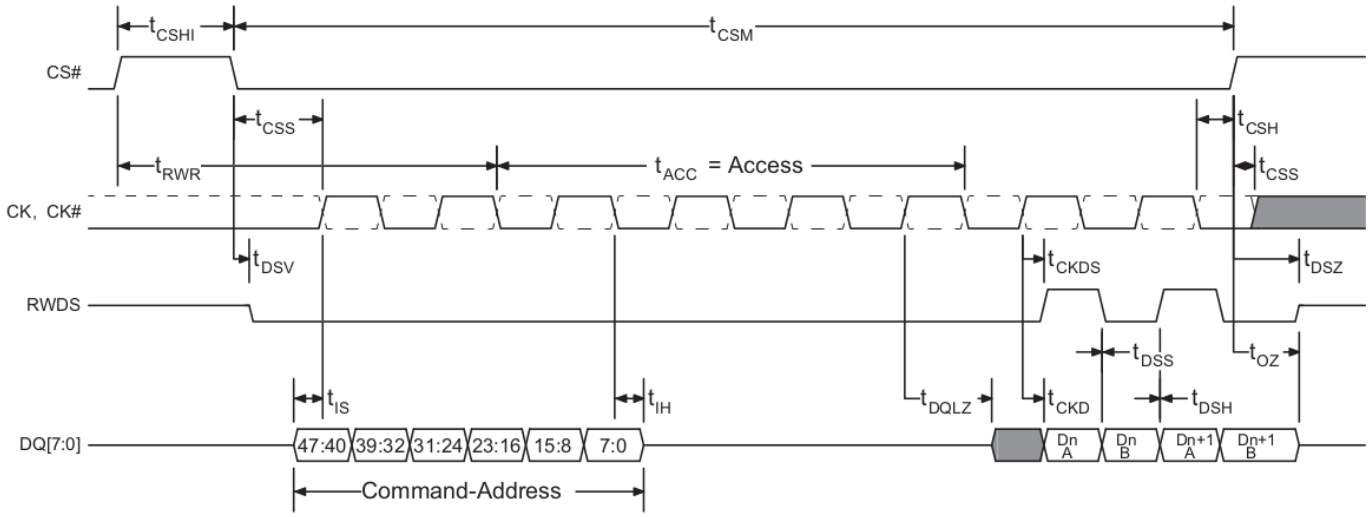

Figure 2.4: HyperRAM interface, read operation timing diagram. During the data transfer, the memory drives both DQ and RWDS. During the command transfer, the host drives DQ and the memory drives RWDS: if RWDS is driven low, the access time is equal to t_{acc} as shown, otherwise the access time is doubled.

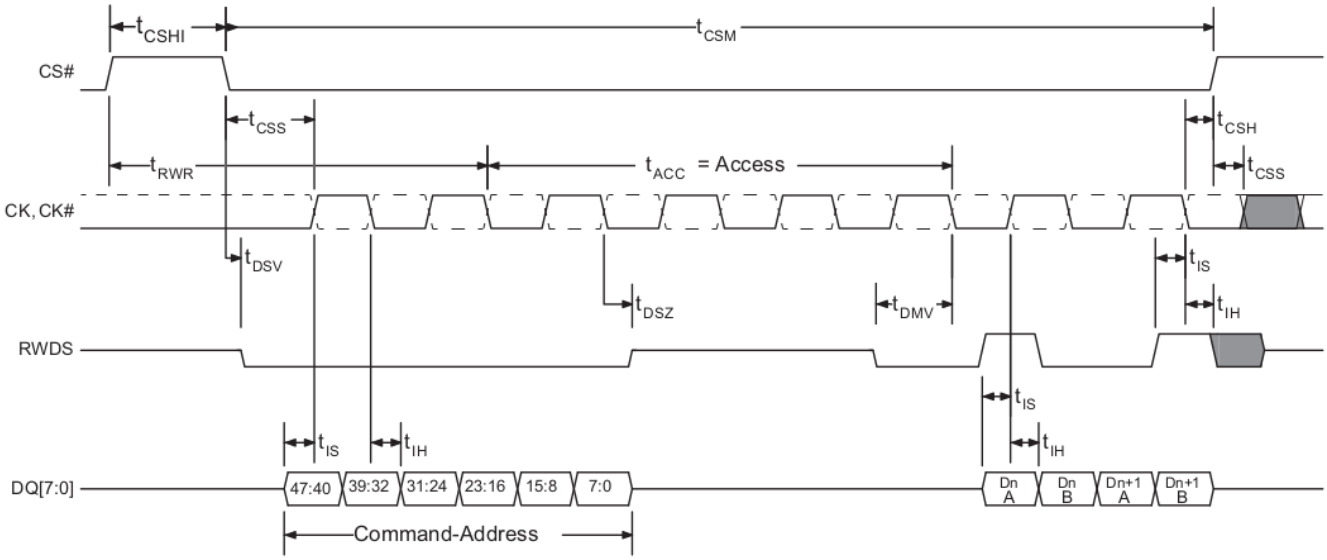

Figure 2.5: HyperRAM interface, write operation timing diagram. During the data transfer, the memory drives both DQ and RWDS. During the command transfer, the host drives DQ and the memory drives RWDS: if RWDS is driven low, the access time is equal to t_{acc} as shown, otherwise the access time is doubled.

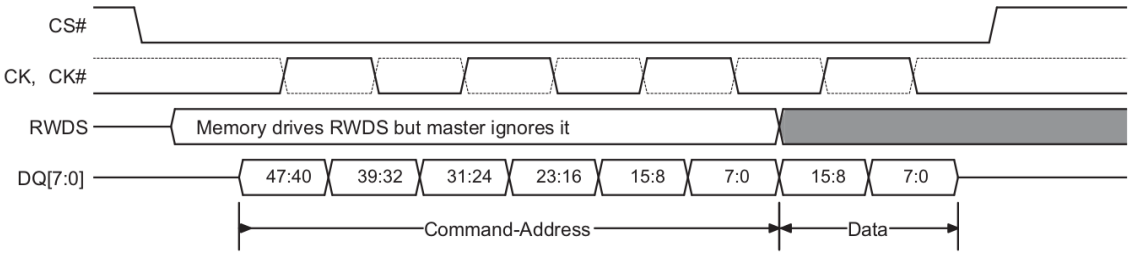

Figure 2.6: HyperRAM interface, register write operation timing diagram. DQ is always driven by the host.

Parameter	Symbol	100 MHz		Unit
		Min	Max	
Chip Select HIGH Between Transactions	t_{CSHI}	10.0	–	ns
HyperRAM Read-Write Recovery Time	t_{RWR}	40	-	ns
Chip Select Setup to next CK Rising Edge	t_{CSS}	3	–	ns
Data Strobe Valid	t_{DSV}	–	12	ns
Input Setup	t_{IS}	1.0	–	ns
Input Hold	t_{IH}	1.0	–	ns
Access Time	t_{ACC}	40	–	ns
Clock to DQs Low Z	t_{DQLZ}	0	–	ns
HyperRAM CK transition to DQ Valid (64 Mb)	t_{CKD}	1	7	ns
HyperRAM CK transition to DQ Valid (128 Mb)			8	
HyperRAM CK transition to DQ Invalid (64 Mb)	t_{CKDI}	0.5	5.2	ns
HyperRAM CK transition to DQ Invalid (128 Mb)			6.2	
CK transition to RWDS valid (64 Mb)	t_{CKDS}	1	7	ns
CK transition to RWDS valid (128 Mb)			8	
RWDS transition to DQ Valid	t_{DSS}	-0.8	+0.8	ns
RWDS transition to DQ Invalid	t_{DSH}	-0.8	+0.8	ns
Chip Select Hold After CK Falling Edge	t_{CSH}	0	–	ns
Chip Select Inactive to RWDS HI-Z	t_{DSZ}	–	7	ns
Chip Select Inactive to DQ HI-Z	t_{OZ}	–	7	ns
HyperRAM Chip Select Maximum LOW Time - Industrial Temperature	t_{CSM}	-	4.0	us
HyperRAM Chip Select Maximum LOW Time - Industrial Plus Temperature		-	1.0	us

Figure 2.7: HyperRAM interface timing parameters.

2.2.1 Command-Address

As we saw in section 2.2, the operation command and the addres are grouped in a 48-bit block, which is sent to the memory by the host one byte for clock level. Every bit of this block has its own meaning:

CA Bit#	Bit Name	Bit Function
47	R/W#	Identifies the transaction as a read or write. R/W# = 1 indicates a Read transaction R/W# = 0 indicates a Write transaction
46	Address Space (AS)	Indicates whether the read or write transaction accesses the memory or register space. AS = 0 indicates memory space AS = 1 indicates the register space The register space is used to access device ID and Configuration registers.
45	Burst Type	Indicates whether the burst will be linear or wrapped. Burst Type = 0 indicates wrapped burst Burst Type = 1 indicates linear burst
44-16	Row & Upper Column Address	Row & Upper Column component of the target address: System word address bits A31-A3 Any upper Row address bits not used by a particular device density should be set to 0 by the host controller master interface. The size of Rows and therefore the address bit boundary between Row and Column address is slave device dependent.
15-3	Reserved	Reserved for future column address expansion. Reserved bits are don't care in current HyperBus devices but should be set to 0 by the host controller master interface for future compatibility.
2-0	Lower Column Address	Lower Column component of the target address: System word address bits A2-0 selecting the starting word within a half-page.

Figure 2.8: Command-Address (CA) bit assignment

2.2.2 Configuration Registers

The S27KL0641DA HyperRAM contains two configuration registers that allow the user to set up different parameters.

Register	CA Bits	47	46	45	44-40	39-32	**31-24**	23-16	15-8	7-0
Configuration Register 0 Read		C0h or E0h				00h	01h	00h	00h	00h
Configuration Register 0 Write		60h				00h	01h	00h	00h	00h
Configuration Register 1 Read		C0h or E0h				00h	01h	00h	00h	01h
Configuration Register 1 Write		60h				00h	01h	00h	00h	01h

Figure 2.9: Command-Address configuration to access the configuration registers

CR1 Bit	Function	Settings (Binary)
15-2	Reserved	000000h — Reserved (default) Reserved for Future Use. When writing this register, these bits should be cleared to 0 for future compatibility.
1-0	Distributed Refresh Interval	10b — default 4 µs for Industrial temperature range devices 1 µs for Industrial Plus temperature range devices 11b — 1.5 times default 00b — 2 times default 01b — 4 times default

Figure 2.10: Configuration Register 1 (CR1) bit assignment

CR0 Bit	Function	Settings (Binary)
15	Deep Power Down Enable (64 Mb)	1 - Normal operation (default) 0 - Writing 0 to CR[15] causes the device to enter Deep Power Down
	Reserved (128 Mb)	Reserved for 128 Mb dual-die stack
14-12	Drive Strength	000 - 34 ohms (default) 001 - 115 ohms 010 - 67 ohms 011 - 46 ohms 100 - 34 ohms 101 - 27 ohms 110 - 22 ohms 111 - 19 ohms
11-8	Reserved	1 - Reserved (default) Reserved for Future Use. When writing this register, these bits should be set to 1 for future compatibility.
7-4	Initial Latency	0000 - 5 Clock Latency - 133 MHz 0001 - 6 Clock Latency - 166 MHz (default) 0010 - Reserved 0011 - Reserved 0100 - Reserved ... 1101 - Reserved 1110 - 3 Clock Latency - 83 MHz 1111 - 4 Clock Latency - 100 MHz
3	Fixed Latency Enable (64 Mb)	0 - Variable Latency - 1 or 2 times Initial Latency depending on RWDS during CA cycles. 1 - Fixed 2 times Initial Latency (default)
	Reserved (128 Mb)	1 - Fixed 2 times Initial Latency (default)
2	Hybrid Burst Enable	0: Wrapped burst sequences to follow hybrid burst sequencing 1: Wrapped burst sequences in legacy wrapped burst manner (default)
1-0	Burst Length	00 - 128 bytes 01 - 64 bytes 10- 16 bytes 11 - 32 bytes (default)

Figure 2.11: Configuration Register 0 (CR0) bit assignment

2.2.3 Deep Power Down Mode

The HyperRAM can enter a special mode, called Deep Power Down (DPD) mode, in which the current consumption is driven to the lowest possible level. This mode is entered setting the *Deep Power Down Enable* bit in CR0. The next access to the device, driving *CS#* low then high (dummy transaction), will cause the device to exit the DPD mode, as well as a hardware reset. A certain time is required to enter or exit the DPD mode.

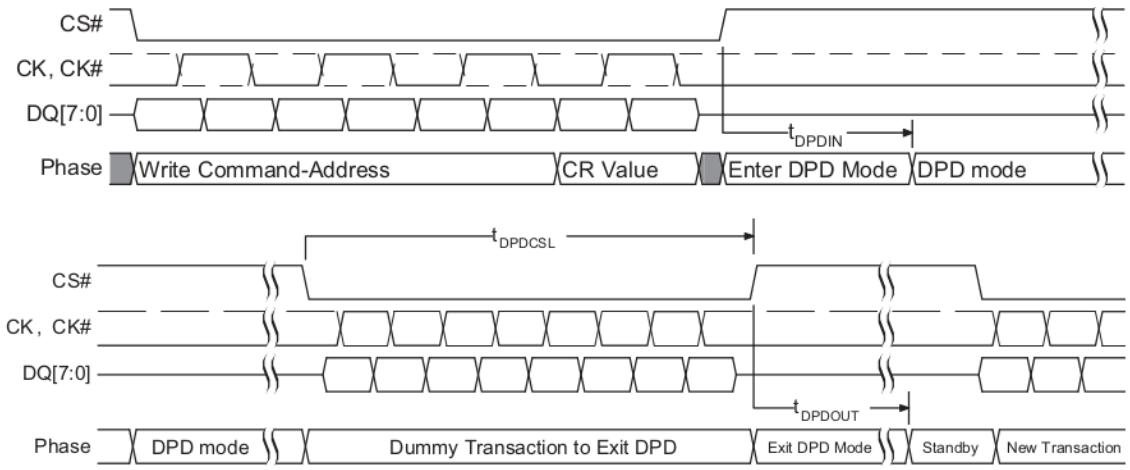

Figure 2.12: DPD timing diagram

2.2.4 Power-Up

The device must not be selected during the power-up, *CS#* has to remain high for a certain time. If *RESET#* is low during the power-up, the time counting does not start until *RESET#* goes high.

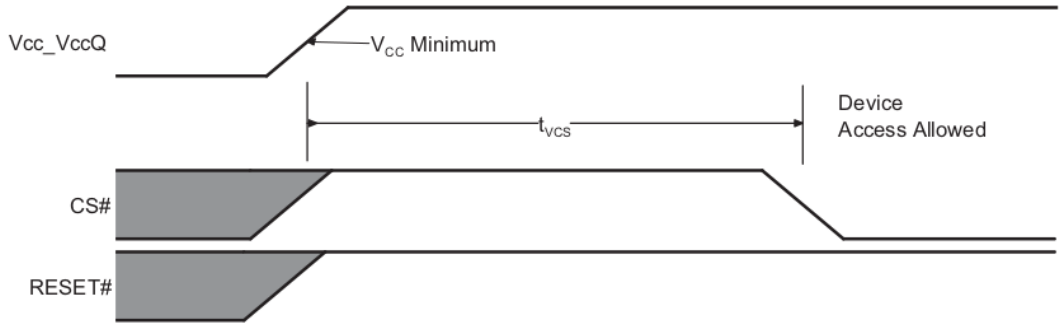

Figure 2.13: Power-up with *RESET#* high

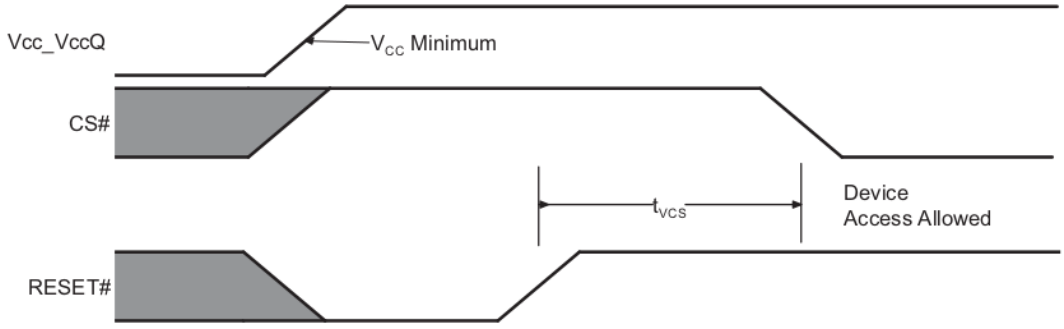

Figure 2.14: Power-up with *RESET#* low

2.3 Converter Design Specifications

Every HyperRAM interface uses a 32-bit addressing. However, the S27KL0641DA device is a 64 Mb memory partitioned in 16-bit words, therefore its addressing takes only 22 bits. On the other hand, the register space access requires dedicated addresses that are not in conflict with the ones related to the memory locations.

In this design, it was decided to virtualize the memory access. To be more precise, the Avalon Memory Mapped interface refers to a 23-bit virtual address, that is translated by the interface converter in the corresponding physical address of the HyperRAM:

- The virtual addresses from 0 to $2^{22}-1$ correspond to the physical memory location addresses from 0 to $2^{22}-1$.
- The virtual address 2^{22} refers to a virtual configuration register that allows the user to set up different parameters. From the point of view of the Avalon interface, the host can only access the virtual configuration register, whereas the physical configuration registers of the HyperRAM cannot be accessed. In this way, it is possible to decide which parameters can be dynamically configured and which cannot.
- The virtual addresses from $2^{22}+1$ to $2^{23}-1$ are reserved for future expansions.

The 16-bit virtual configuration register (VCR) is organized in the following way:

VCR BIT	FUNCTION	SETTINGS
0	Deep Power Down Enable	0: normal operation, exit DPD mode (default) 1: enter DPD mode
1	Fixed Latency Enable	0: variable latency (default) 1: fixed latency
2-15	Reserved	Reserved for future expansions.

By default, the memory works in normal mode. The host can force it to enter the DPD mode setting the *Deep Power Down Enble* bit in VCR and then to go back to normal mode resetting that same bit. The interface converter shall detect any update of the *Deep Power Down Enble* bit and consequently drive the memory according to figure 2.12. Moreover, since a DPD exit request corresponds to a VCR update, it is possible for the host to update multiple parameters at the same time. For this reason, the interface converter must automatically drive a CR1 update after exiting the DPD mode.

The default configuration of VCR does not match the default configuration of CR1 (figure 2.10). For this reason, the interface converter must automatically update CR1 after the power-up before allowing the host to start a new operation.

As we can see from figures 2.13 and 2.14, the interface converter must wait for the memory to power-up before allowing the host to start a new operation. Every time the memory is resetted, it must be powered-up again.

As far as the frequency is concerned, the S27KL0641DA HyperRAM can work up to 100 MHz. However, the interface converter is designed to work at 50 MHz, sending to the memory a clock at that same frequency (as described in section 4.1).

TEST ENVIRONMENT

Before starting the interface converter design, it is necessary to define a test environment for it. We can exploit some of the IP cores provided by the CAD software (which are of course well-functioning) to create an Avalon Memory Mapped system suitable for the test. In particular, the test environment is an processor-based system that read some inputs and change the status of some LEDs according to it. The HyperRAM is employed as data memory for the processor.

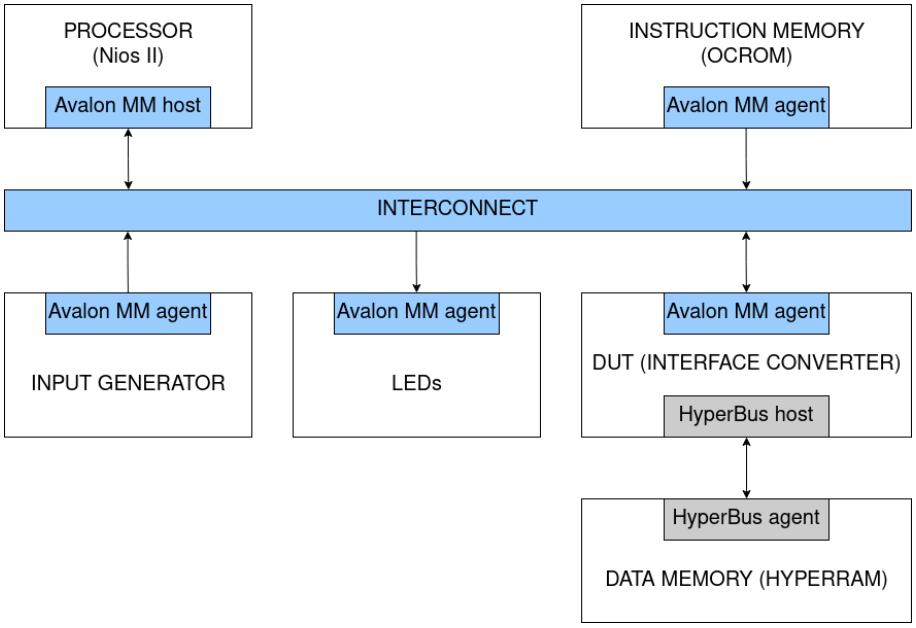

Figure 3.1: Test environment for the interface converter

To ensure that the test environment is well-functioning, the easiest way is to replace the DUT and the HyperRAM with an on-chip RAM, which can be obtained simply by using an IP core provided by the CAD software.

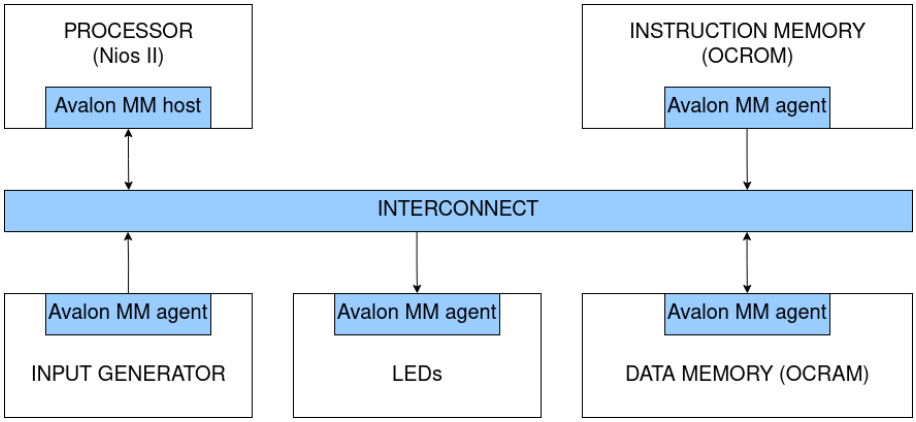

Figure 3.2: Test environment verification

DESIGN PARTITIONING

From now on, the interface converter is referred as *avs_hram_converter*, i.e. the name of the custom IP implementing it. The top-level view of this IP is shown in figure 4.1. On the Avalon side, the address line is on 23 bits and the data line is on 16 bits, as described in chapter 2, section 2.3. The burstcount signal is on 11 bits, i.e the maximum possible parallelism, corresponding to a theoretical maximum burst lenght equal to 2^{10} as stated in the Avalon documentation.

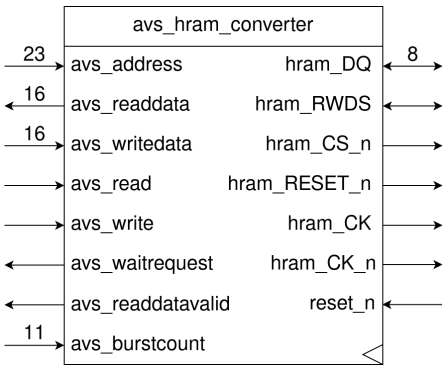

Figure 4.1: Top-level view of the interface converter

Unfortunately, it is usually not possible to push the burst lenght up to its theoretical maximum, since the duration of any memory operation is upper bounded. Considering that the latency of the converter depends on its implementation, it is not possible to estimate the actual upper bound of the burst lenght in advance. For this reason, the maximum parallelism is employed and the effective maximum of the burst lenght will be estimated after completing the design (section 4.8.1).

The design follows a top-down approach. At first, it is important to point out the main features to be implemented:

- **Command-Address building**: the Avalon input signals must be re-organized arranging the CA.
- **Configuration registers building**: every time the virtual configuration register is written, it is necessary to convert its content so that the physical configuration registers of the memory can be properly updated.
- **SDR to DDR conversion**: the 16-bit SDR data provided at the Avalon interface must be converted in an 8-bit DDR data to put it on the memory data bus.
- **DDR to SDR conversion**: the 8-bit DDR data provided by the memory (which is synchronous with RWDS and not with the internal clock) must be converted in a 16-bit SDR data and synchronized with the clock.
- **Clock shifting and clock gating**: the internal clock must by shifted by 90 degrees and properly gated before being sent to the memory.
- **Timer**: the system must be aware the passage of time to satisfy all the timing requirements.
- **Address reconstruction**. As we can see in figure 2.2, the host can interrupt a write operation at any time. However, the HyperRAM does not support this feature. For this reason, the interface converter must end the operation and start a new one when the burst is resumed. The new operation shall begin at the right address, i.e. the one immediately after the last written location.

The CAD software provides an IP implementing a clock controller. Indeed, we can just create a custom IP implementing all the features except the clock gating (*avs_hram_mainconv*) and combine it with the clock controller IP (*clkctrl*) to create the interface converter (*avs_hram_converter*), as shown in figure 4.2. The *avs_hram_mainconv* IP is composed by an FSM-based control unit (section 4.8) and an execution unit (figure 4.3):

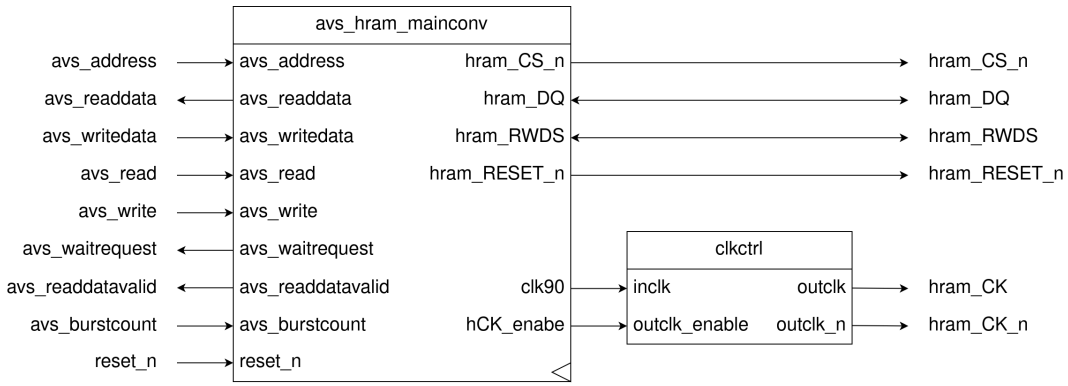

Figure 4.2: Architecture of the *avs_hram_converter* IP

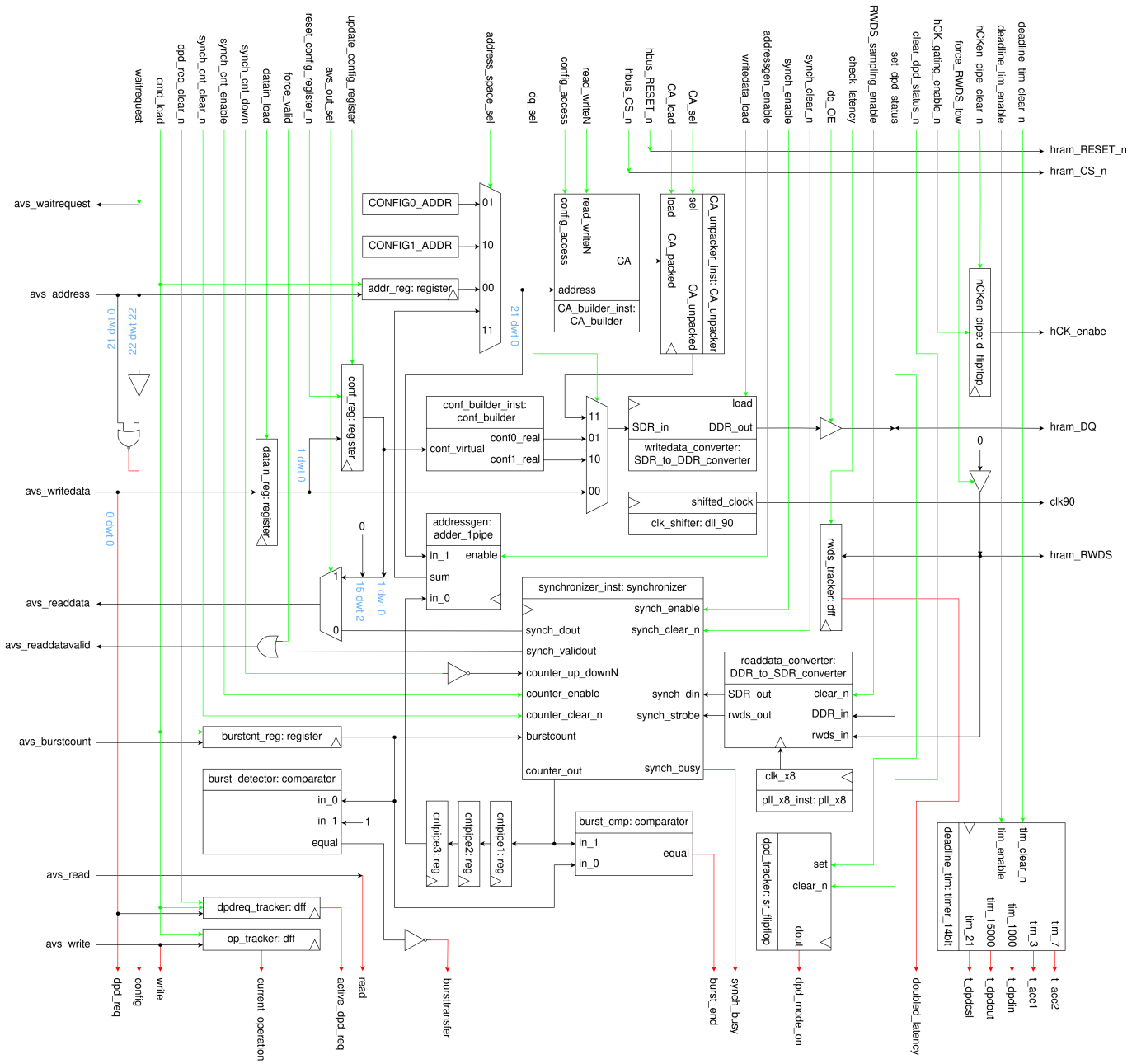

Figure 4.3: Execution unit of the *avs_hram_mainconv* IP. Avalon signals are placed on the left. HyperRAM signals are placed on the right. The top green signals represent the control signals received from the control unit. The bottom red signals represent the status signals sent to the control unit.

The role of each block represented in figure 4.3 is described in the table below:

NAME	TYPE	NICKNAME	DESCRIPTION
addr_reg	*register*	address register	It stores the virtual address provided on the Avalon side.
datain_reg	*register*	data register	It stores the input data provided on the Avalon side.
conf_reg	*register*	configuration register	It implements the virtual configuration register as described in section 2.3.
burstcnt_reg	*register*	burst lenght register	It stores the burst lenght value provided on the Avalon side.
CA_builder_inst	*CA_builder*	CA builder	It builds the 48-bit Command-Address starting from the address value and the operation type. Refer to section 4.3 for a detailed description.
CA_unpacker_inst	*CA_unpacker*	CA unpacker	It separates the Command-Address in 3 different 16-bit data. Each of them can be picked out depending on the value of a selector. Refer to section 4.6 for a detailed description.
conf_builder_inst	*conf_builder*	configuration builder	It generates the content of the physical configuration registers of the HyperRAM starting from the content of the virtual configuration register. Refer to section 4.5 for a detailed description.
writedata_converter	*SDR_to_DDR_converter*	writedata converter	It converts a 16-bit SDR data in a 8-bit DDR data, so that it can be channeled into the HyperRAM *DQ* bus. Refer to section 4.4 for a detailed description.
RWDS_tracker	*dff*	RWDS tracker	It samples the value of RWDS using the internal clock. It is not suitable for sampling it while receiving a DDR data, it is intended to be used during the CA transmission.
dpdreq_tracker	*dff*	DPD request tracker	It is employed to keep track of a DPD mode entry request.
op_tracker	*dff*	operation tracker	It is employed to keep track of the type of operation currently in execution.
dpd_tracker	*sr_flipflop*	DPD tracker	It is employed to keep track of the current memory mode (NORMAL, DPD).
readdata_converter	*DDR_to_SDR_converter*	readdata converter	It samples *RWDS* and *DQ* using a 400 MHz clock, converting the 8-bit DDR data in a 16-bit SDR data. It also provides a version of RWDS with a rising-edge approximately center-aligned with the SDR data. Refer to section 4.1 for a detailed description.
pll_x8_inst	*pll_x8*	pll x8	It generates a 400 MHz clock starting from the internal 50 MHz clock.
CONTINUED...			

NAME	TYPE	NICKNAME	DESCRIPTION
synchronizer_inst	*synchronizer*	synchronizer	It samples the 16-bit SRD at the output of readdata converter (using the center-aligned version of RWDS it provides) to synchronize it with the internal clock. It contains an 11-bit up-counter that can be accessed from outside.
burst_detector	*comparator*	burst detector	It notifies the system when the burst lenght is greater than 1.
burst_cmp	*comparator*	burst comparator	It is employed to notice when the burst transmission has reached its end (i.e. the burst lenght).
addressgen	*adder_1pipe*	address generator	It is employed when a write burst operation is resumed (after being stopped) to generate the new starting address.
clk_shifter	*ddl_90*	clock shifter	It introduces a 90 degree delay on the internal clock.

4.1 Readdata Converter

As we can see in figure 2.4, during the data transfer of a read operation the memory drives both *DQ* and *RWDS*, the former being edge-aligned to the latter. The effect of board traces can be neglected considering that the *DQ* trace and the *RWDS* trace have a similar lenght on the PCB. The main goal of *readdata converter* is to introduce a proper delay on *RWDS* so that it can be used to sample *DQ*; in particular, referring to figure 4.4, *DQ_out* and *RWDS_out* shall be approximately center-aligned.

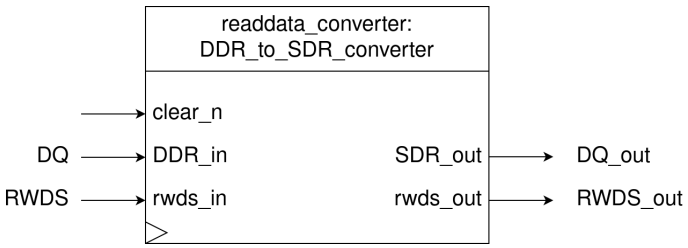

Figure 4.4: Top-level view of *readdata converter*.

Considering that *RWDS* oscillates at the clock frequency, a first possibility would be to introduce a combinational delay. In this way, it would be possible to push the memory frequency up to 100 MHz (assuming that all the other blocks in the system are fast enough). Theoretically, the Intel Cyclone 10 LP FPGA allows to introduce a controlled combinational delay on its input pins; however, the documentation is really poor, especially when it comes to explaining how to do it in the CAD software. Due to the many troubles encountered, it was decided to follow a different approach.

Instead of forcing a combinational delay, a very high frequency clock (oversampling clock) can be employed. In this way, *RWDS* can be shifted using just a flip-flop chain. To do that, the system need first to oversample both *RWDS* and *DQ* to synchronize them with the high-frequency clock. During the oversampling, some samples are collected violating the setup-hold constraints and they must be considered invalid. Indeed, the oversampling frequency must be high enough to collect more valid samples than invalid samples within a semi-period of *DQ* and *RWDS*, even in the worst case scenario.

To compute the minimum sampling frequency, a setup-hold time lower than 3/4 of the clock period has been considered (assumption that shall be verified during the synthesis). Considering a rising-edge sampling, in the worst case

both the first and the last samples are not valid, therefore the oversampling frequency must be at least 10 times the memory frequency:

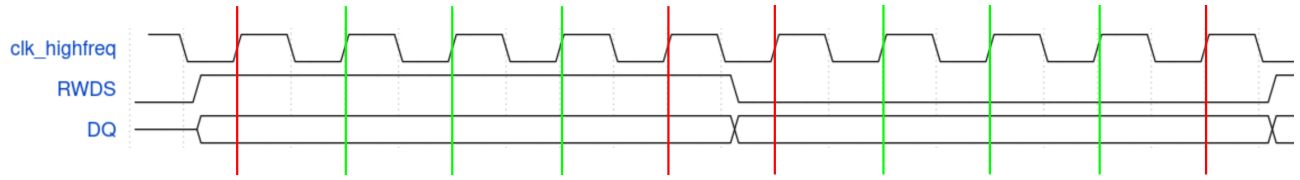

Figure 4.5: Oversampling frequency 10 times the memory frequency, single-edge sampling. The green samples are valid, the red samples are invalid.

Switching to a double-edge sampling, in the worst case three samples are not valid, therefore the oversampling frequency must be at least 8 times the memory frequency:

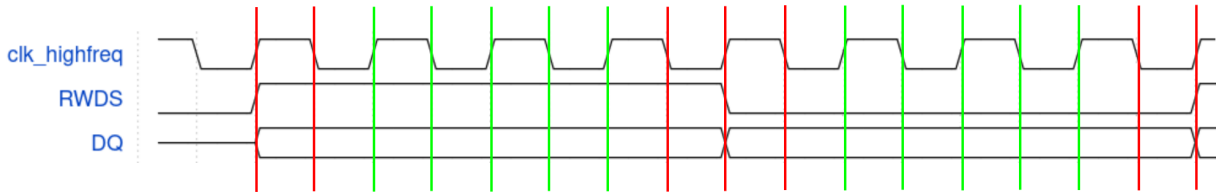

Figure 4.6: Oversampling frequency 8 times the memory frequency, double-edge sampling. The green samples are valid, the red samples are invalid.

The oversampling frequency is generated using a PLL available in the FPGA. As written in the Intel Cyclone 10 LP datasheet, the maximum output frequency of the PLL is equal to 472.5 MHz, therefore the memory frequency is upper bounded at 59 MHz (considering a double-edge sampling). In other words, to make the oversampling technique work it is not possible to push the memory frequency up to its maximum (100 MHz). Indeed, it was decided to lower the memory frequency to 50 MHz and to use a 400 MHz sampling frequency (generated by means of a PLL with a multiplication factor equal to 8).

Another important parameter is the value of the shift to be introduced on *RWDS*. Theoretically, it should be delayed by 1/4 of the clock period (5 ns, i.e. two sampling clock periods) to make it center-aligned with *DQ*. However, in the real case the setup-hold violations must be taken into account. As we can see in figure 4.7, a delay of 5 ns may lead to an oscillation of *RWDS_out* too close to the next variation of *DQ_out*. Indeed, the goal of *readdata converter* is to generate *RWDS_out* such that it can be used to sample *DQ_out* without introducing any setup-hold violation.

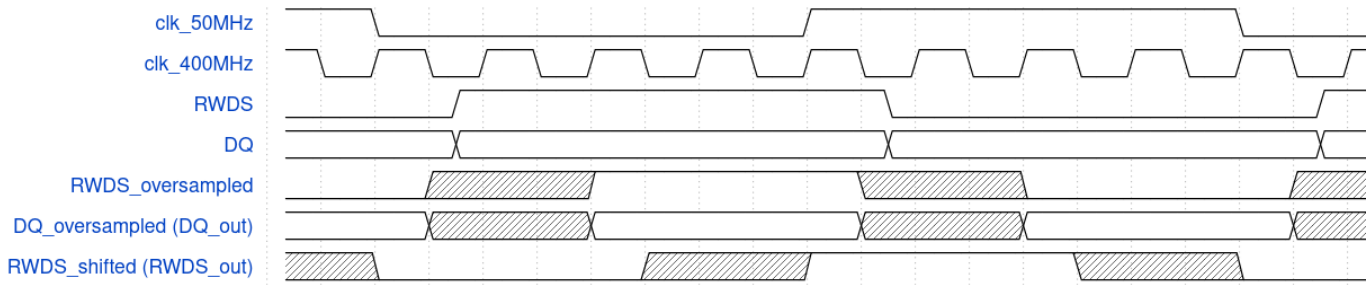

Figure 4.7: Example of possible setup-hold violation during the oversampling. In this case, *RWDS_out* is shifted by two sampling clock cycles with respect to *DQ_out*.

To make sure that no setup-hold violations are present, it is necessary to reduce the shift to a single sampling clock cycle (2.5 ns) as shown in figure 4.8.

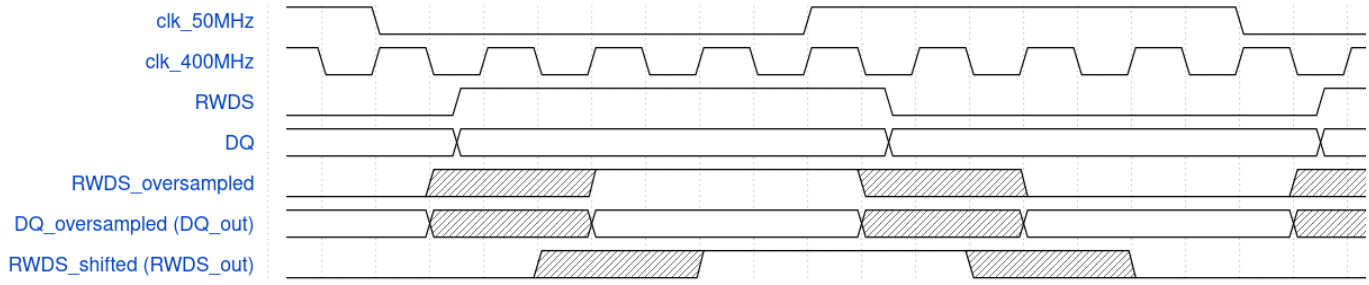

Figure 4.8: Timing digram with a single clock cycle shift between *RWDS_out* and *DQ_out*.

As we already said, the goal of *readdata converter* is to introduce a shift between *RWDS_out* and *DQ_out* so that the former can be employed to sample the latter. The sampling is implemented outside *readdata converter* by means of a register having the clock pin connected to *RWDS_out* and the input data pin connected to *DQ_out* (referring to figure 4.3, this register is located inside the *synchronizer*). In this regard, it is important to highlight that the timing behavior described in figure 4.8 works correctly only assuming that there are no other delay contributions between *readdata converter* and the sampling register (i.e. that the delay between the clock pin and the input pin of the sampling register is exactly equal to the delay between *DQ_out* and *RWDS_out*). Unfortunately, this assumption cannot be considered true. In general, we must consider the circuit in figure 4.9.

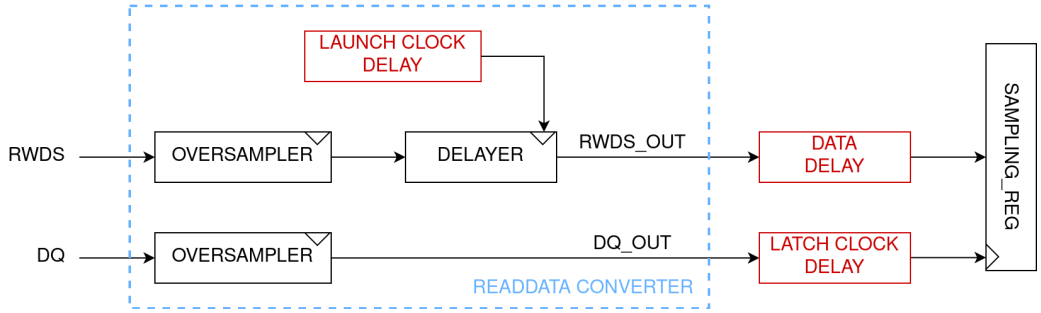

Figure 4.9: Basic scheme of *readdata converter*. All the delay terms are considered.

To make *readdata converter* work, it is necessary to estimate the value of the delay terms and compensate them. A basic timing analysis in which *readdata converter* behaves as described in figure 4.8 is reported in figure 4.10.

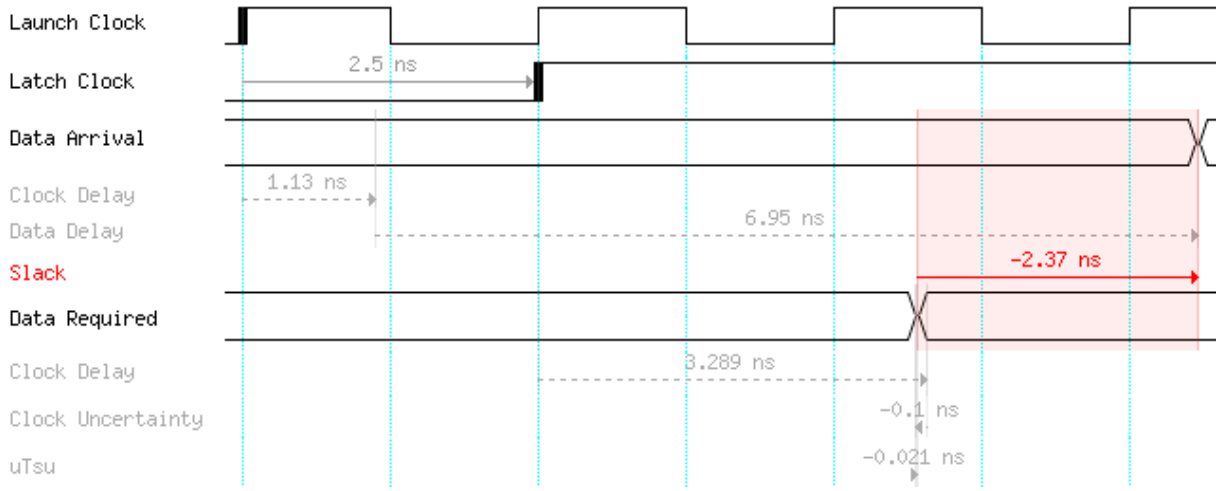

Figure 4.10: Timing analysis according to figure 4.8.

As we can see in figure 4.10, *RWDS_out* (the latch clock) is delayed by 2.5 ns with respect to *DQ_out* as expected. However, the value of the input data pin of the sampling register (data arrival) changes 2.37 ns later than required (i.e. slightly less than a period of the high-frequency clock). To compensate this timing violation, *readdata converter* can be designed to introduce a further shift of 2.5 ns (thus, a total delay of 5 ns), as shown in figure 4.11.

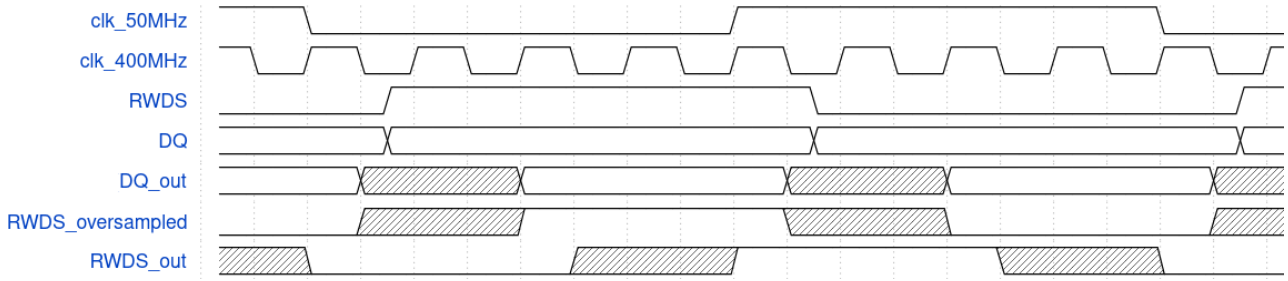

Figure 4.11: *readdata converter* timing diagram able to prevent setup/hold violations.

The timing diagram represented in figure 4.11 is able to prevent setup/hold violation. However, it is not able to make the system in figure 4.9 work. Indeed, *RWDS_out* cannot just shift *RWDS_oversampled*, since this signal may have spurious oscillations in correspondence of the invalid samples (which may assume whatever logic value). For this reason, the system oversamples *RWDS_in* and then implements a majority decision: when at least 5 samples out of 8 are different from the current decision about the value of *RWDS_in*, the current decision is toggled. Figure 4.12 represents some different scenarios, showing that the current decision does not display spurious oscillations regardless of the value assumed by the invalid samples. *DDR_in* is considered valid the clock period corresponding to the decision change; in this way, no setup/hold violations occour. *RWDS_out* is generated toggling a signal with 2 clock periods delay with respect to the variation of *DDR_out*, therefore setup/hold violations are prevented and no spurious oscillations are present. Moreover, its value is inverted with respect to the current decision, so that its rising edges correspond to the valid valued of *SDR_out*.

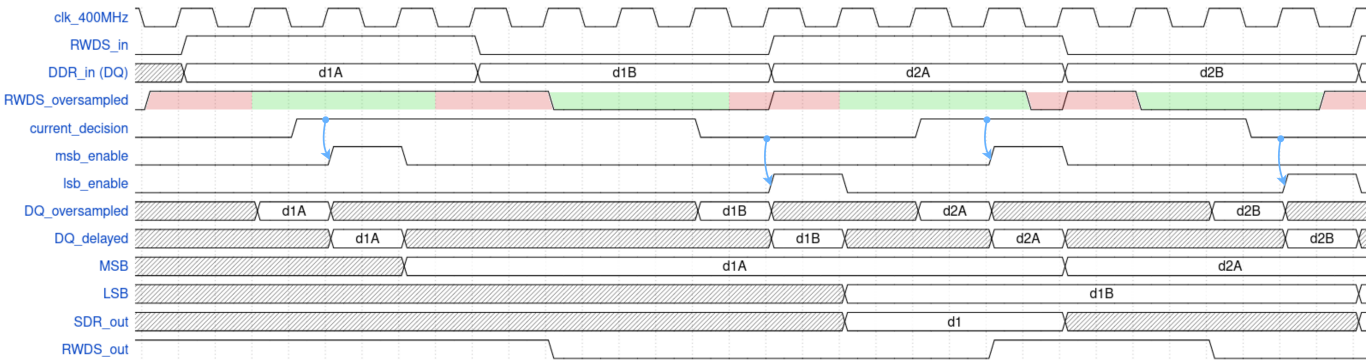

Figure 4.12: Timing diagram of different scenarios during the oversampling. Three samples per *RWDS_in* period are not valid (worst-case). The values of *RWDS_oversampled* highlighted in red are invalid (they corresponds to a setup/hold violation, therefore they may assume whatever logic value). The values of *RWDS_oversampled* highlighted in green are valid. The current decision is obtained through a combinational circuit and it is sampled by the rising edge of the high-frequency clock.

The architecture of *readdata converter* implementing the timing diagram in figure 4.12 is divided in an execution unit (described in subsection 4.1.1) and a control unit (described in subsection 4.1.3) implemented as a finite-state machine. As we can see in figure 4.13, the execution unit is strongly pipelined to be able to work with a 400 MHz clock frequency. Indeed, the circuit generating the current decision from the collected samples is not combinational, differently from what is assumed in figure 4.12, and the control signals generated by the control unit are pipelined too. Consequently, the actual timing diagram is slightly different from the one shown in figure 4.12, since a certain latency is introduced. However, the working principle remains the same.

4.1.1 Execution Unit

Figure 4.13 represents the architecture of the execution unit of *readdata converter*. The current decision about the value of *RWDS_in* is stored in a dedicated type-T flip-flop, namely the *tracker*. Two different flip-flop chains are used to oversample *RWDS_in* on both rising and falling edges of the high-frequency clock. The majority decision is implemented using a *voter* and some logic gates. In particular, the *voter* is able to notice if the sum of all the collected samples is equal (*eq4*) or greater (*gt4*) than four. If the sum is greater than four (which means that there are more samples equal to logic 1 than to logic 0) and the current decision (stored in the *tracker*) is logic 0, the decision is toggled. If the sum is lower than four (neither *eq4* nor *gt4* are set, thus there are more samples equal to logic 0 than to logic 1) and the current decision is logic 1, the decision is toggled.

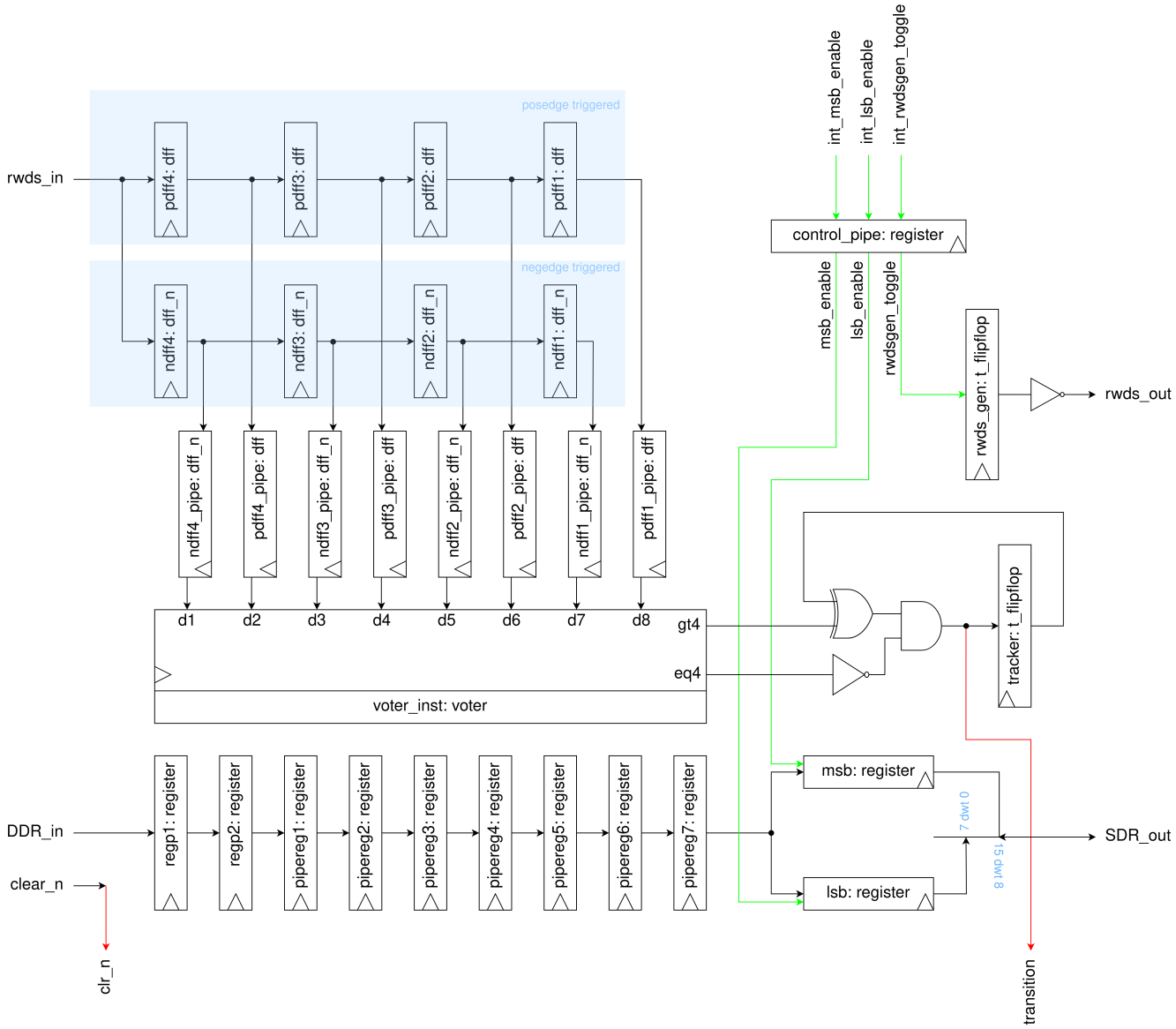

Figure 4.13: Execution unit of *readdata converter*. The top green signals are the control signals received from the control unit, whereas the bottom red signals are the status signals sent to the control unit.

4.1.2 Voter

The *voter* is able to compute the sum of its inputs and to notice if the result is equal (*eq4*) or greater (*gt4*) than four. As we can see in figure 4.14, it is diveded in five different pipeline stages. The sum is computed by means of a set of half-adders and full-adders. Starting from the sum, *eq4* and *gt4* are generated using some logic gates.

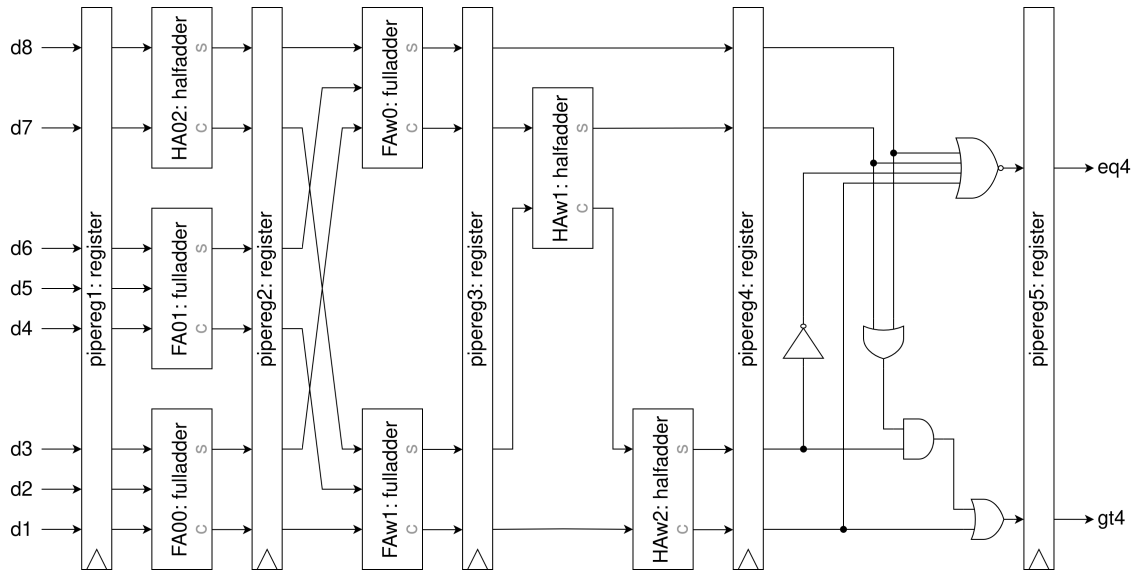

Figure 4.14: Architecture of the *voter*.

4.1.3 Control Unit

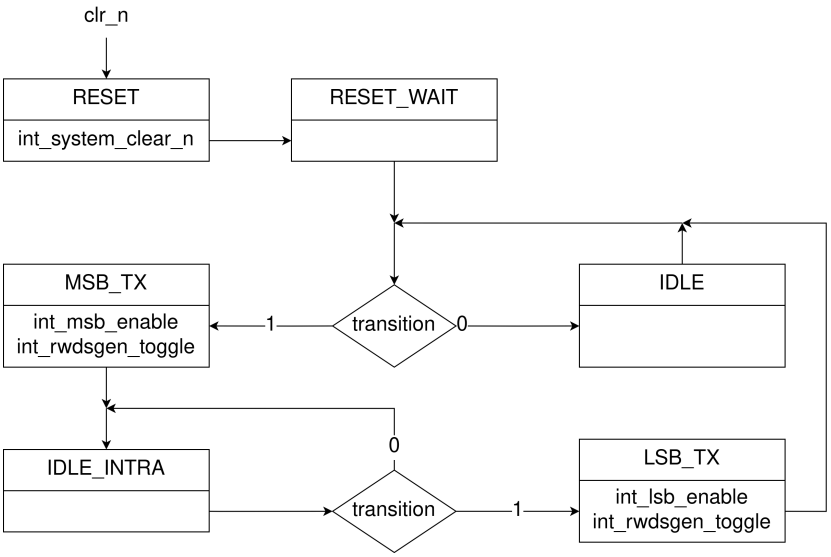

Figure 4.15: Control unit of *readdata converter*. It lets the data pass through the *lsb* register or the *msb* when a variation of decision is detected (refer to figure 4.1.1 for the execution unit). The register (*msb* or *lsb*) are selected alternatively to reconstruct a 16-bit SDR data.

4.2 Synchronizer

As described in section 4.1, *readdata converter* generates a 16-bit SDR output and a properly shifted strobe to sample it. The SDR output can be connected to the input data pin of a sampling register having the clock pin connected to the strobe, as shown in figure 4.9. However, in this way the data is not synchronized to the 50 MHz clock.

The main goal of the *synchronizer*, apart from providing the sampling register, is to synchronize the data with the 50 MHz clock. The main idea is to alternate multiple sampling registers, so that the sampled data remains available for a longer time. The architecture is divided in an execution unit (figure 4.17) and a control unit (figure 4.19).

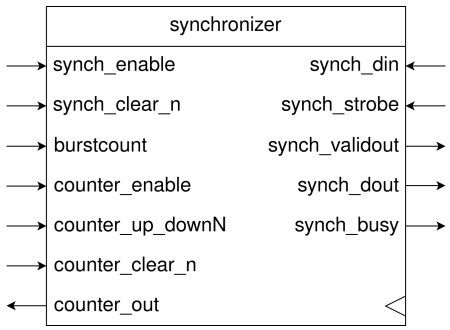

Figure 4.16: Top-level view of the *synchronizer*.

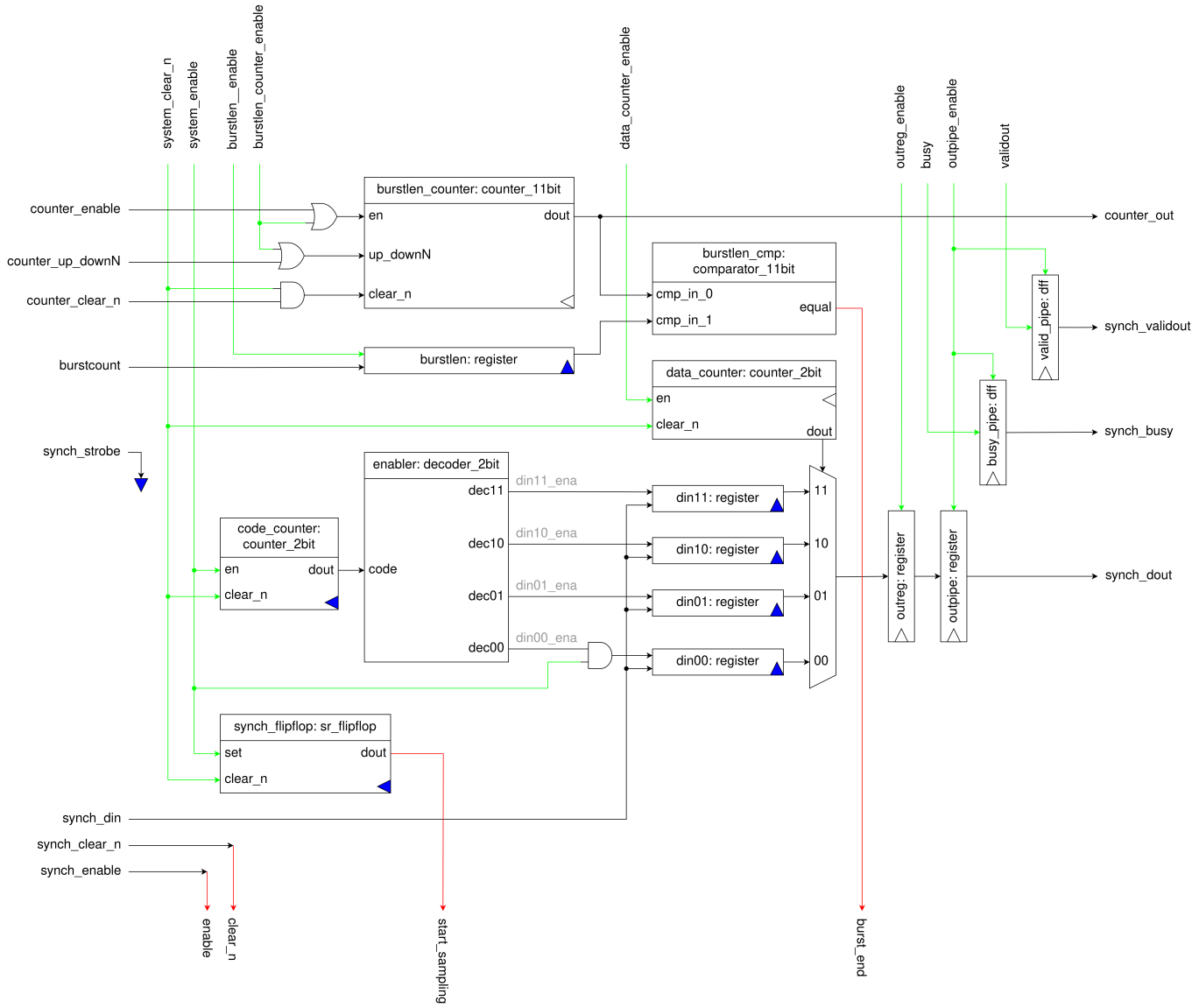

Figure 4.17: Execution unit of the *synchronizer*.

The actual synchronization is implemented by means of a set-reset flip-flop, namely the *synch_flipflop*, having the clock pin connected to the strobe, the set pin always set to logic 1 (assuming that the system is enabled) and the output pin connected to the FSM as a status signal called *start_sampling*. The strobe is asynchronous to the clock, therefore a variation of *start_sampling* may lead to setup/hold violations; in other words, the control unit may take more than one clock cycle to detect a variation of *start_sampling*. However, since the sampled data is available for

multiple clock cycles, this does not represent a problem. The timing diagram in figure 4.18 refers to a condition in which the activation of *start_sampling* is detected with a delay of 1 clock cycle.

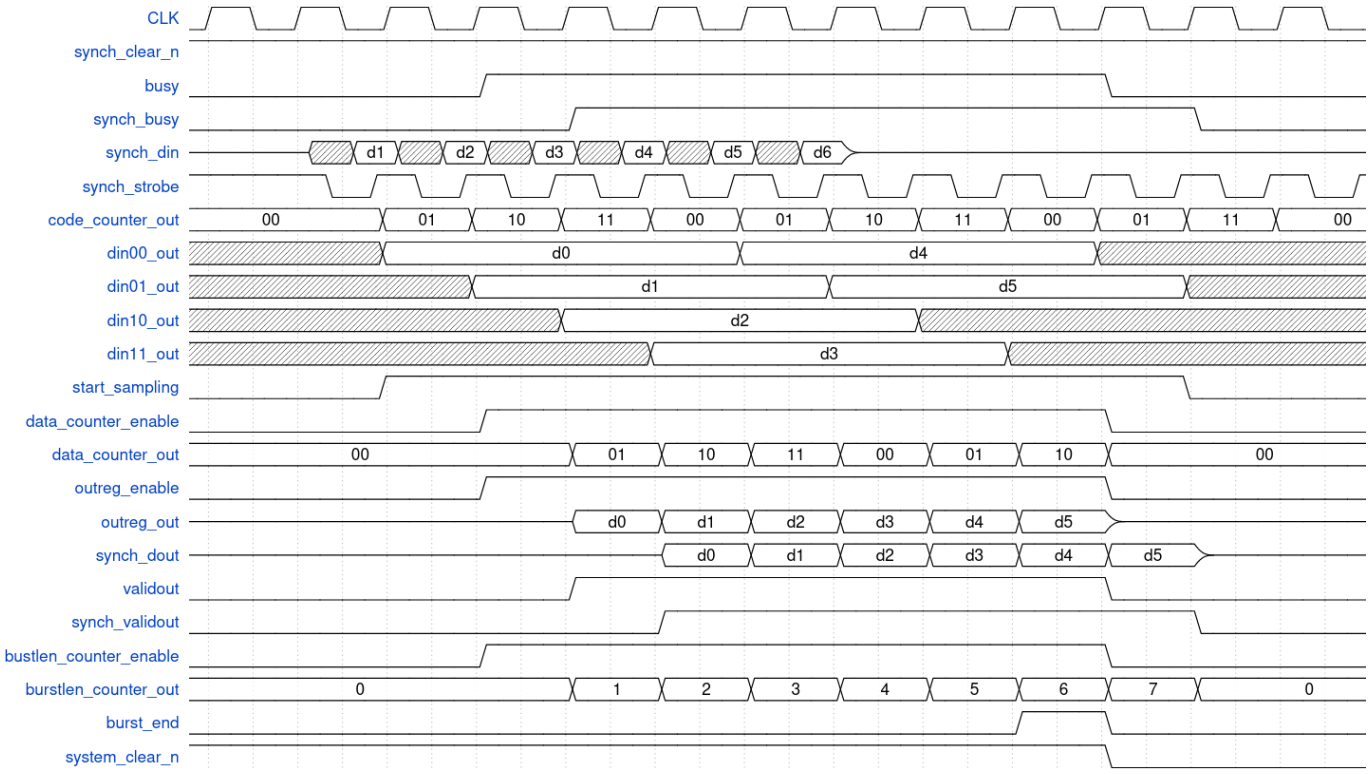

Figure 4.18: Timing diagram of the *synchronizer*.

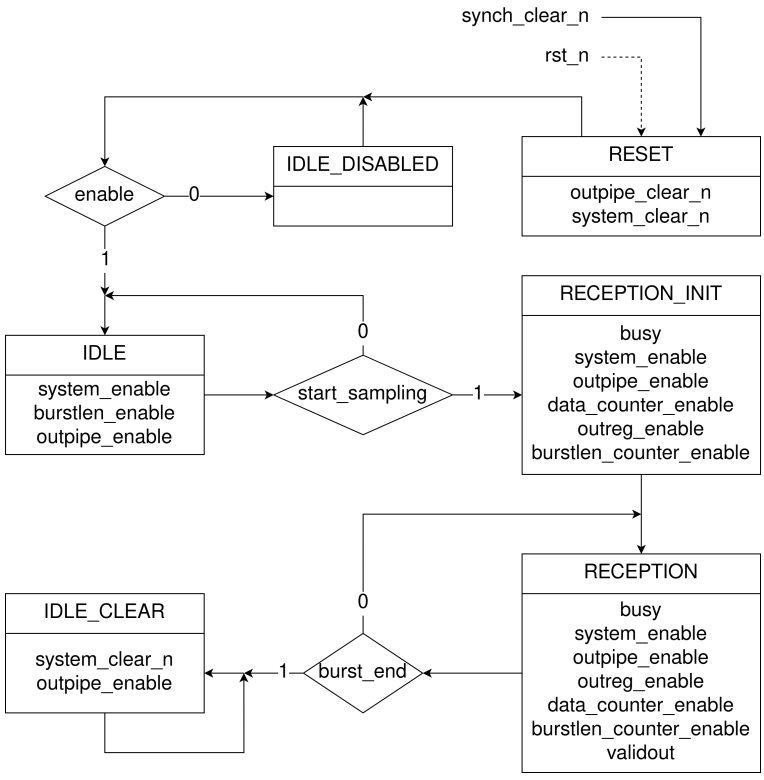

Figure 4.19: Control unit (finite-state machine) of the *synchronizer*.

The *synchronizer* is able to notice when the burst to be transferred is completed. The burst lenght is stored in a dedicated register called *burstlen*. The counter employed to keep track of the burst progress, namely the *burstlen_counter*, is made available outside, so that it can be employed independently when the *synchronizer* is not enabled.

Once the burst transfer is completed, the *synchronizer* remains in a state in which the FSM constantly reset all the components by means of the *system_clear_n* control signal, therefore it is not able to receive a new data burst. The activation of *system_clear_n* may be detected with a certain delay by the componend that are clocked by the strobe, since it is synchronous to the clock. However, *system_clear_n* remains active for many clock cycles, therefore this does not represent a problem. To restart the *synchronizer* and make it able to reveive a new burst, it is necessary to manually set the *synch_clear_n* signal after detecting *synch_busy* low, as shown in figure 4.18.

The *synchronizer* comes with an enable signal, namely *synch_enable*. As we can see in figure 4.19, this signal must be set before beginning a burst transfer (i.e. before the strobe starts oscillating), otherwise the strobe edges are ignored. Setting *synch_enable* after beginning a burst transfer causes an unknowk behavior. If *synch_enable* is deactivated before completing a burst trasfer, the transfer is completed anyway.

4.3 CA Builder

The 48-bit Command-Address is generated rearranging the bits of the address and combining them with the information about the operation type, as shown in figure 4.20.

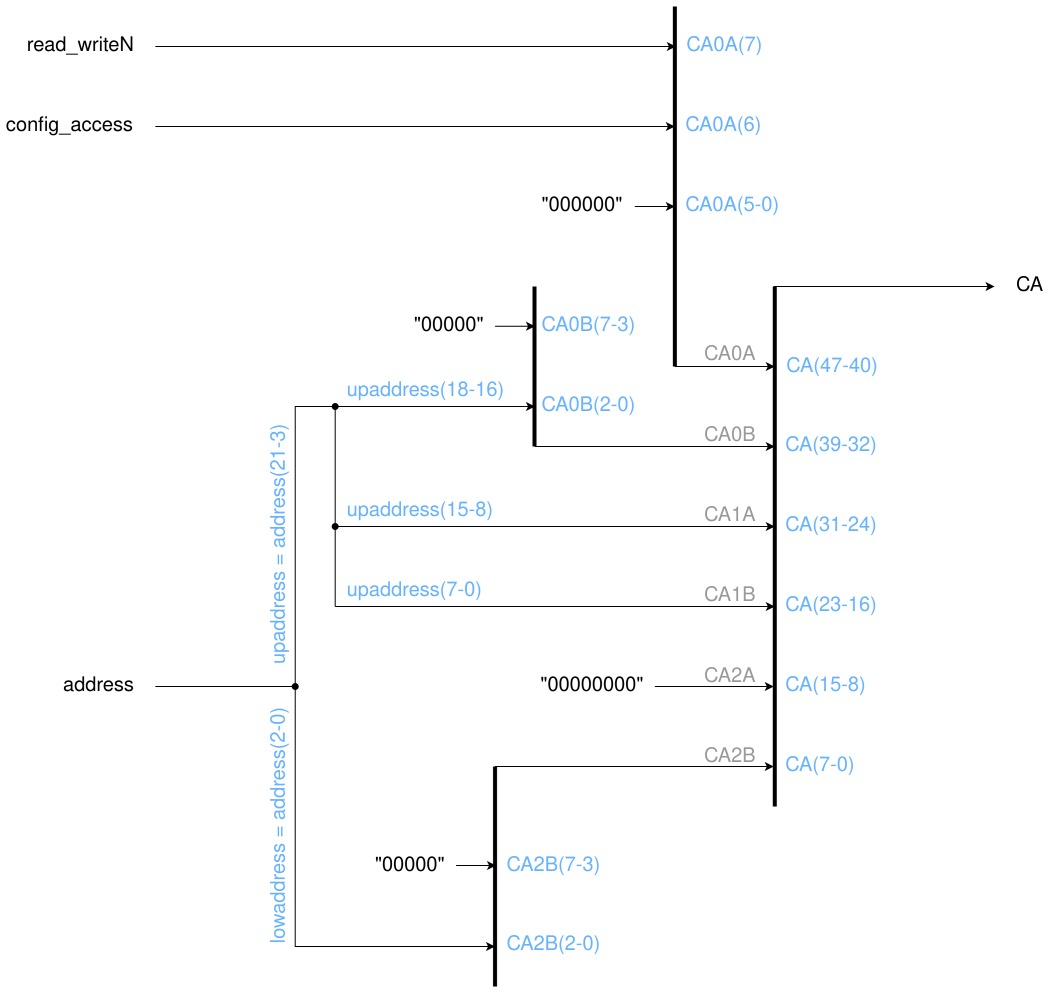

Figure 4.20: Architecture of the *CA builder*.

4.4 Writedata Converter

The Avalon side of the system works with a 16-bit SDR data, which must be converted in a 8-bit DDR data to be channeled in the HyperRAM DQ bus. This conversion is implemented in the *writedata converter*. A multiplexer having the selector connected to the system clock is exploited to alternate the more-significant byte and the less-significant byte of the SDR data, as represented in figure 4.21.

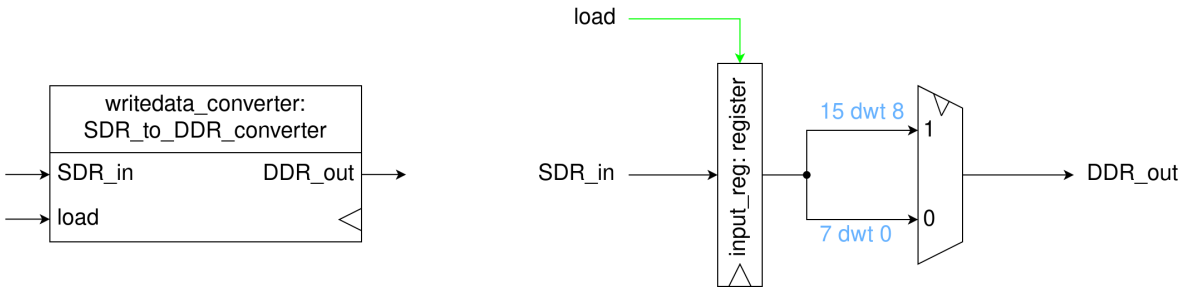

Figure 4.21: Top view (on the left) and internal architecture (on the right) of the *writedata converter*.

4.5 Configuration Builder

From the Avalon side, the host can only refer to the virtual configuration register. However, every time the virtual configuration register is written, the content of the physical configuration registers of the memory shall be updated. The *configuration builder* generates the content of the physical configuration registers starting from the content of the virtual configuration register, as shown in figure 4.22.

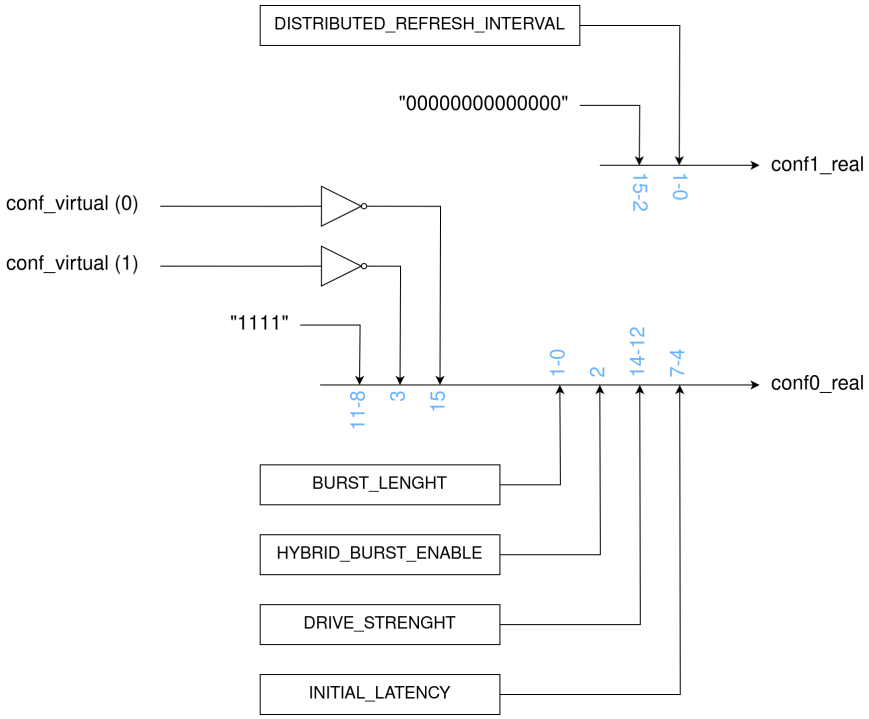

Figure 4.22: Architecture of the *configuration builder*.

4.6 CA Unpacker

The 48-bit Command-Address generated by the *CA builder* must be separated in three different 16-bit segments (namely *CA0*, *CA1* and *CA2*), so that it can be fed to the *writedata converter* and channeled in the HyperRAM DQ bus. This separation is implemented by the *CA unpacker*, represented in figure 4.23. The selection between *CA0*, *CA1* and *CA2* is achieved through a dedicated selector.

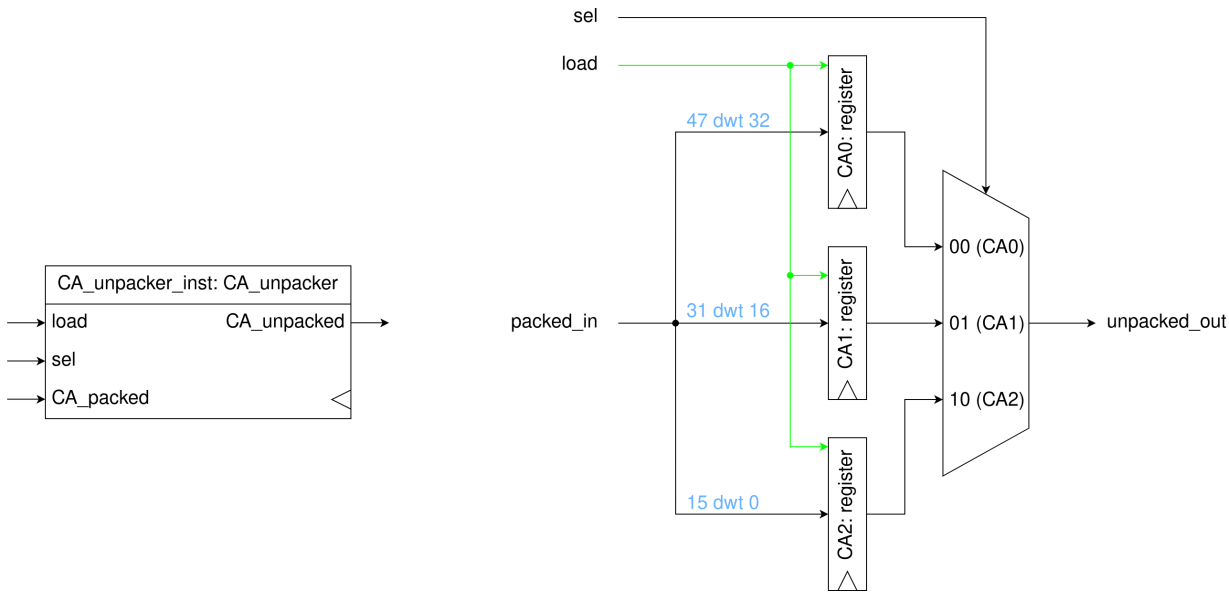

Figure 4.23: Top view (on the left) and internal architecture (on the right) of the *CA unpacker*.

4.7 Address Generator

When it comes to the Avalon communication protocol, the host can interrupt the burst transfer at any time during a write operation. However, the HyperRAM protocol does not support this feature. For this reason, the interface converter must terminate the operation toward the HyperRAM when the host interrupts the transfer and begin a new operation when the transfer is resumed. Consequently, any interruption of the transfer implies to reconstruct the start address (i.e. the address from which to start the operation) and to send again the command-address to the memory. For this reason, a burst interruption is really expensive in terms of latency and it would be better to avoid it.

During a write operation, the burst progress is tracked by means of the 11-bit counter provided by the *synchronizer* (*burstlen counter* referring to figure 4.17). When the host interrupts and resumes the transfer, the new start address can be generated from the output of this counter. In this regard, it is important to make some considerations:

- During a burst transfer the counter output is always two steps ahead with respect to the actual burst progress, so that the control unit can detect in advance when the burst is about to terminate (figure 4.24).
- When a burst transfer is interrupted (i.e. when the control unit switches from the *WRITEBURST* status to the *STOP_BURST_1* status) the counter out increases by one more step, although a new data has not been transferred (figure 4.24).
- The initial address provided by the host (namely the base address) remains stored in *addr_reg* (figure 4.3) until the trasfer is completed.

All this considered, the start address can be reconstructed as *new start address* = *base address* + *counter out* - 3. Three pipeline registers are placed on the counter output (*cntpipe1*, *cntpipe2* and *cntpipe3* in figure 4.3) to obtain *counter out* - 3, whereas the sum with the base address is implemented using an adder (*addressgen* in figure 4.3).

4.8 Control Unit and Timing Diagrams

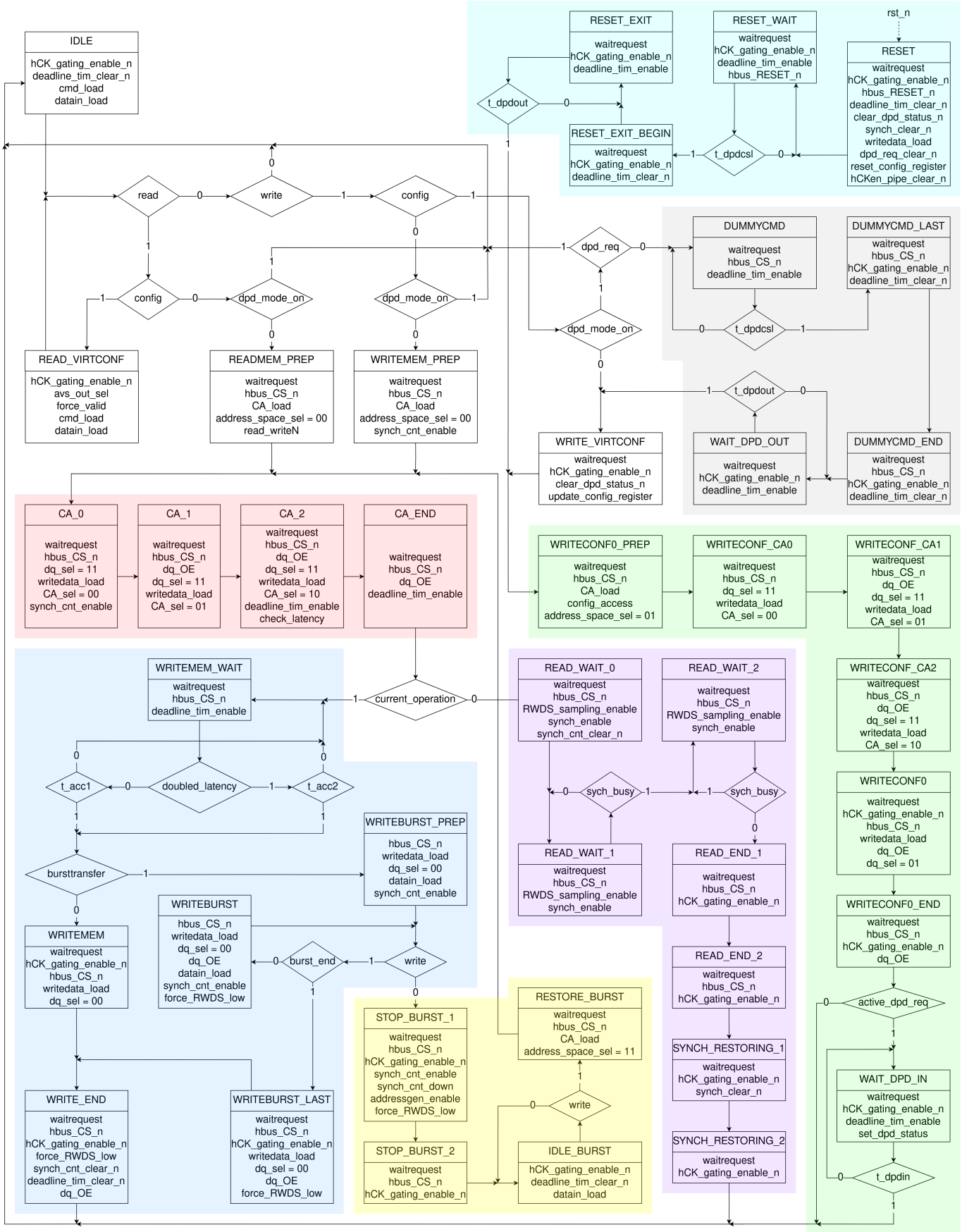

Figure 4.24: Control unit of the *avs_hram_mainconv* IP. Turquoise block: reset and power-on process. Grey block: DPD mode exit process. Green block: configuration registers update. Purple block: read operation. Blue block: write operation.

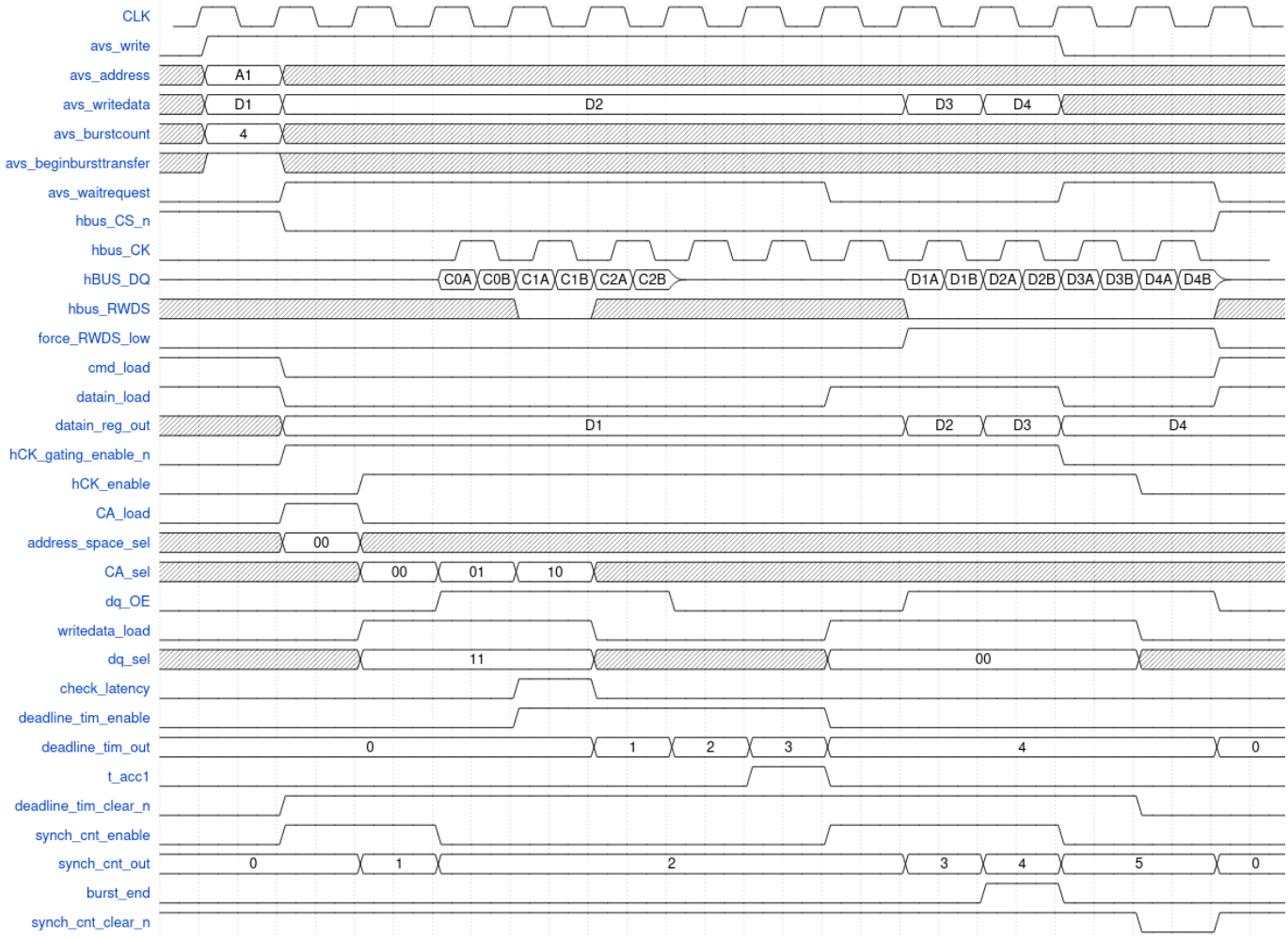

Figure 4.25: Write burst operation. The Avalon signals vary according to figure 2.2. The HyperRAM signals match the timing in figure 2.5.

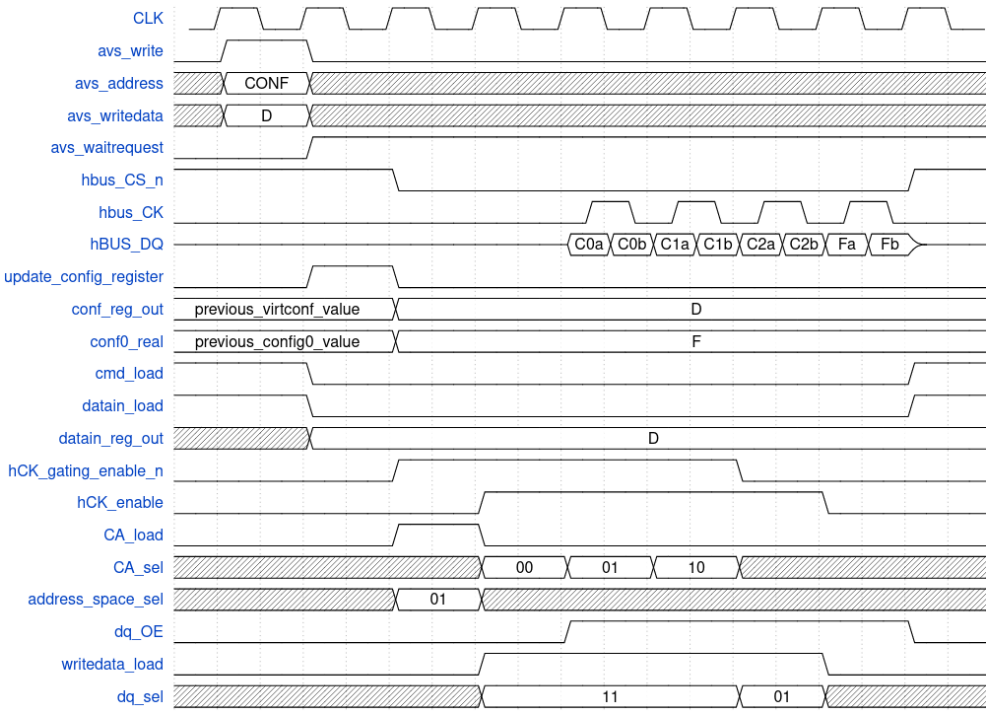

Figure 4.26: Configuration register update operation. The Avalon signals vary according to figure 2.2. The HyperRAM signals match the timing in figure 2.6.

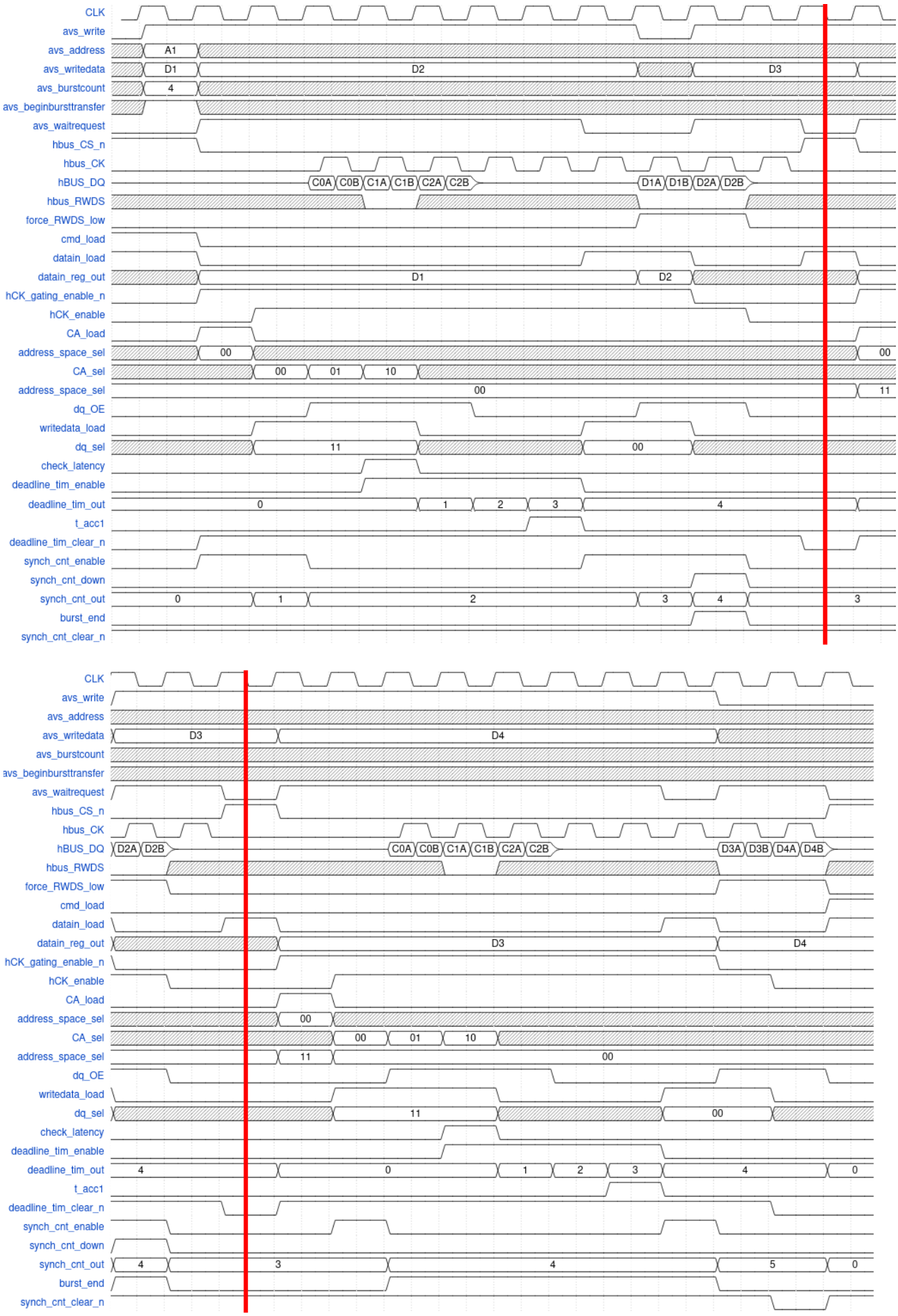

Figure 4.27: Write burst operation with burst interruption.

4.8.1 Memory Timing Constraints

As we can see in figures 2.4 and 2.5, several timing contraints (whose value is listed in figure 2.7) must be respected to ensure the correct functioning of the memory:

- $\mathbf{T_{CSHI}}$: always ensured by the FSM
- $\mathbf{T_{CSS}}$: always ensured by the FSM
- $\mathbf{T_{RWR}}$: always ensured by the FSM
- $\mathbf{T_{ACC}}$: always ensured by the FSM
- $\mathbf{T_{CSH}}$: always ensured being null
- $\mathbf{T_{DSV}}$: always ensured by the FSM
- $\mathbf{T_{CKDS}}$, $\mathbf{T_{CKD}}$: irrelevant, the trace delay is not considered
- $\mathbf{T_{DSZ}}$ (read operation) : irrelevant, only the edges of RWDS are significant
- $\mathbf{T_{DSZ}}$ (write operation) : irrelevant, RWDS is acquired early enough
- $\mathbf{T_{DSS}}$, $\mathbf{T_{DSH}}$: irrelevant, they do not affect the oversampling
- $\mathbf{T_{OZ}}$: irrelevant, the value of DQ is relevant only in correspondence of the edges of RWDS
- $\mathbf{T_{DQLZ}}$: irrelevant, the value of DQ is relevant only in correspondence of the edges of RWDS
- $\mathbf{T_{CSM}}$: it limits the maximum burst lenght
- $\mathbf{T_{IS}}$, $\mathbf{T_{IH}}$: forced during synthesis

Ideally, the *clock shifter* introduces a delay (T_{CKSH}) between the internal clock and the HyperRAM clock equal to 90° (5 ns). However, in the real case this delay is characterized by an uncertainty of ±50 ps. Moreover, the *clock controller* introduces an additional delay term ($T_{CLKCTRL}$). Similarly, a certain combinational delay (T_{CLKDQ}) is present between the main clock and DQ. All this considered, the actual delay between DQ and the HyperRAM clock (T_{IS}) is definitely different from 90°. To ensure the constraint on the minimum value of T_{IS} it is necessary to consider the worst case on T_{CKSH} and to force a constraint on T_{CLKDQ} during the synthesis:

$$T_{IS} = T_{CKSH} + T_{CLKCTRL} - T_{CLKDQ} \qquad \min\{T_{CKSH}\} = 4.95 \text{ ns} \qquad \min\{T_{CLKCTRL}\} = 0 \text{ ns}$$

$$T_{IS} > 1 \text{ ns} \quad \Rightarrow \quad T_{CLKDQ} < 3.95 \text{ ns}$$

The condition $T_{CLKDQ} < 3.95$ ns can be easily satisfied during the synthesis. Actually, this condition has been extended to $T_{CLKDQ} < 1$ ns to have some margin, since the synthesizer does not have any issue to satisfy it.

When it comes to the constraint on T_{IH}:

$$10 \text{ ns} - T_{IH} = T_{CKSH} + T_{CLKCTRL} - T_{CLKDQ} \qquad \max\{T_{CKSH}\} = 5.05 \text{ ns} \qquad \min\{T_{CLKDQ}\} = 0 \text{ ns}$$

$$T_{IH} > 1 \text{ ns} \quad \Rightarrow \quad T_{CLKCTRL} < 3.95 \text{ ns}$$

The *clock controller* is a particular IP already available in the Intel catalog. Unfortunately, its documentation does not specify its delay. However, the condition $T_{CLKCTRL} < 3.95$ ns is not that strict, therefore we can assume that it is always satisfied.

T_{CSM} limits the maximum burst lenght. In the worst case (doubled access time, read operation), a latency of 12 clock periods must be considered between the assertion of *hram_CS_n* and the beginning of the burst transfer (as we can see in figure 5.3). Moreover, at the end of the burst trasfer 7 additional clock cycles are required before de-asserting *hram_CS_n* (as shown in figure 5.4). As a result:

$$12\ T_{CLK} + 7\ T_{CLK} + N_{BURST}\ T_{CLK} < \max\{T_{CSM}\} = 4\ \mu\text{s} \quad \Rightarrow \quad N_{BURST} < 181$$

As we can see, even though the burst lenght is expressed on 11 bits, only 9 bits are actually necessary.

TEST RESULTS

To test the interface converter against a large number of different stimuli it can be inserted in a processor-based system, as represented in figure 3.1. However, even if this approach allows to test the DUT under several different condition, it is possible that some particular operations of our interest are never tested. For instance, the processor may never try to put the memory in DPD mode. For this reason, it was decided to perform different analysis:

- **Preliminary simulation**: a custom testbench is created to directly drive the interface converter in order to verify its behavior under specific conditions.
- **Final simulation**: the interface converter is inserted in the processor-based system (figure 3.1) to simulate it against a large number of data.
- **Test on VirtLAB**: the processor-based system containing the interface converter is synthesized in the FPGA.

5.1 Preliminary Simulation

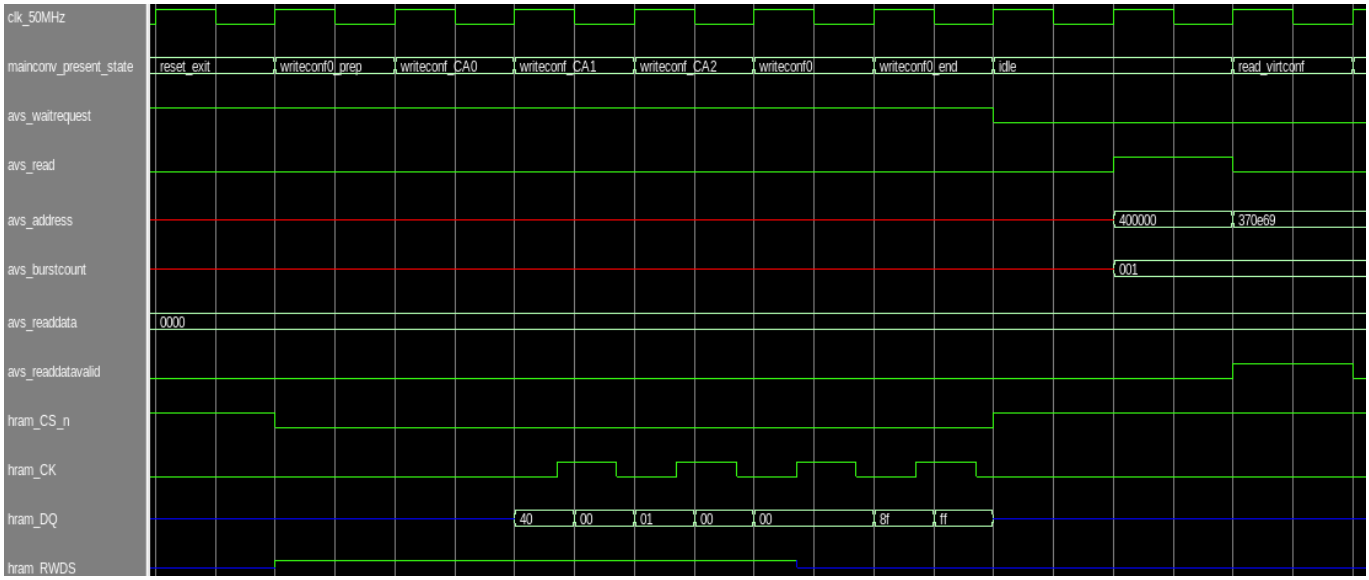

Figure 5.1: Automatic configuration register update after power-on.

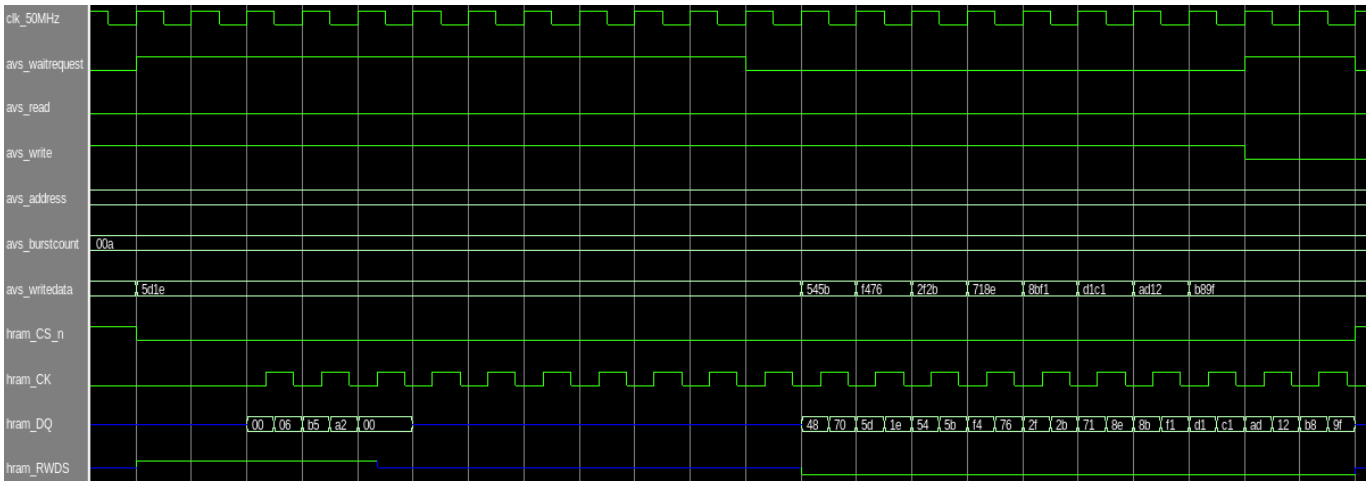

Figure 5.2: Write operation.

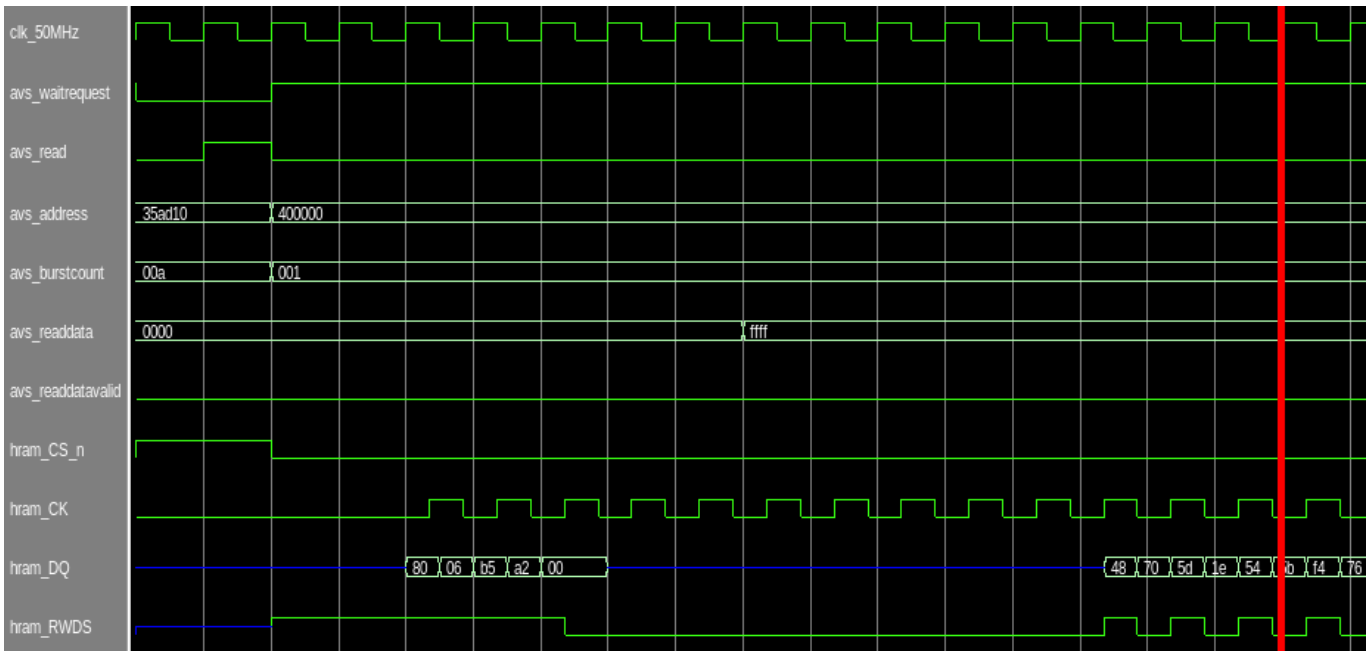

Figure 5.3: Read operation - part 1.

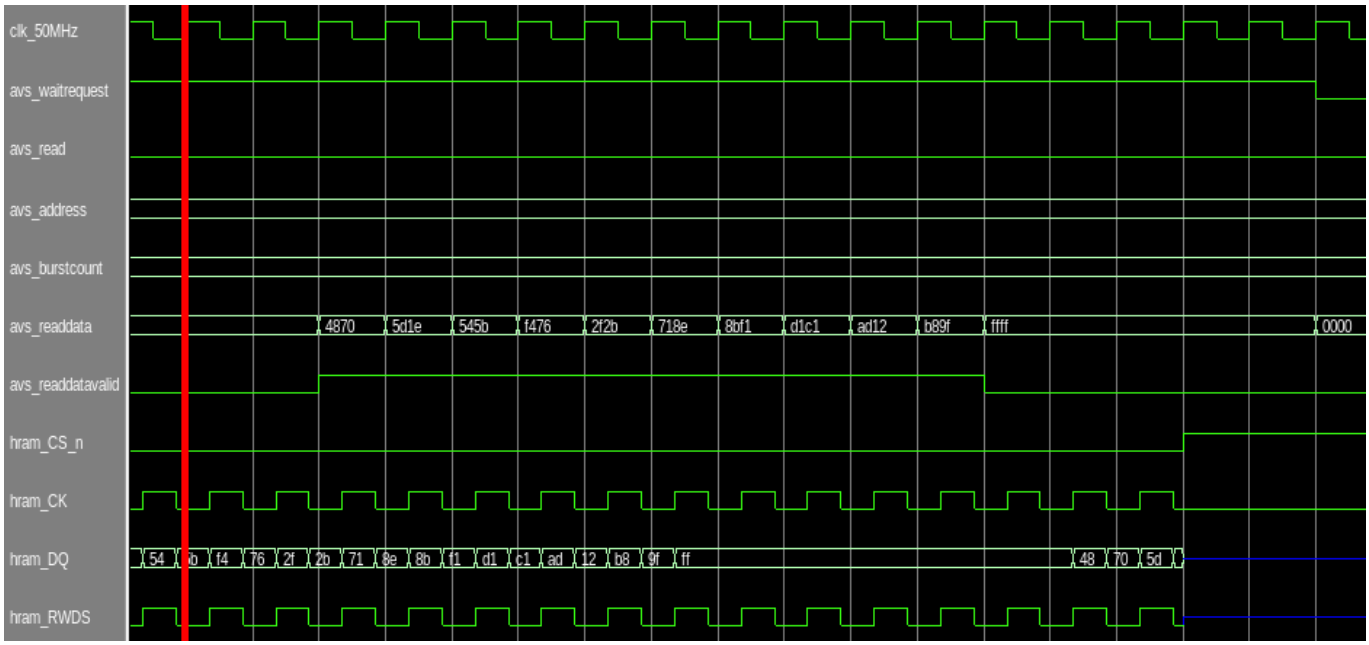

Figure 5.4: Read operation - part 2.

As we can see in figure 5.1, the system automatically initialize CR0 after the power-up as expected (refer to section 4.8). Moreover, a virtual configuration register read operation is succesfully completed.

Other than the write operation (figure 5.2) and the read configuration (figures 5.3 and 5.4), it is important to simulate the behavior of the system in DPD mode. In particular, it must be able to:

- enter the DPD mode when requested (figure 5.5)
- ignore any operation beside a DPD exit request (figure 5.6)
- exit the DPD mode when requested (figures 5.6 and 5.7)
- automatically update the value of CR0 after exiting the DPD mode (figure 5.8)

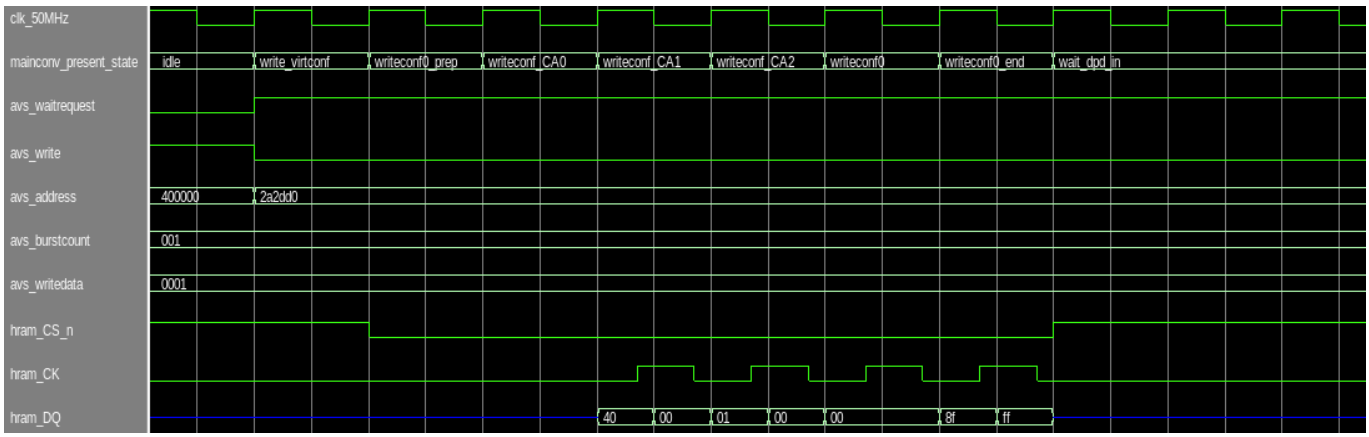

Figure 5.5: DPD mode entry request.

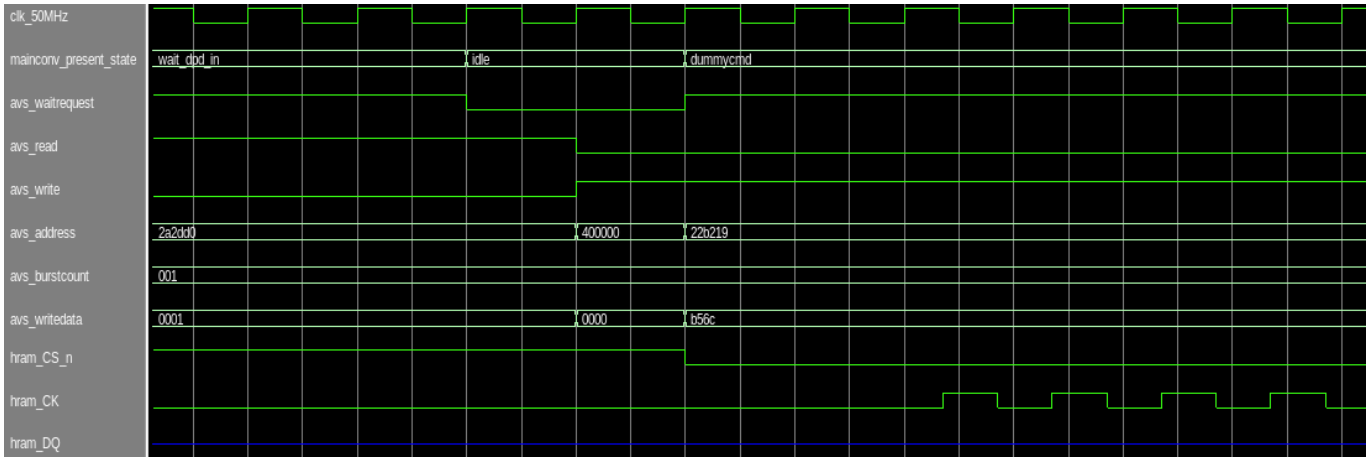

Figure 5.6: Ignored read operation and DPD mode exit request.

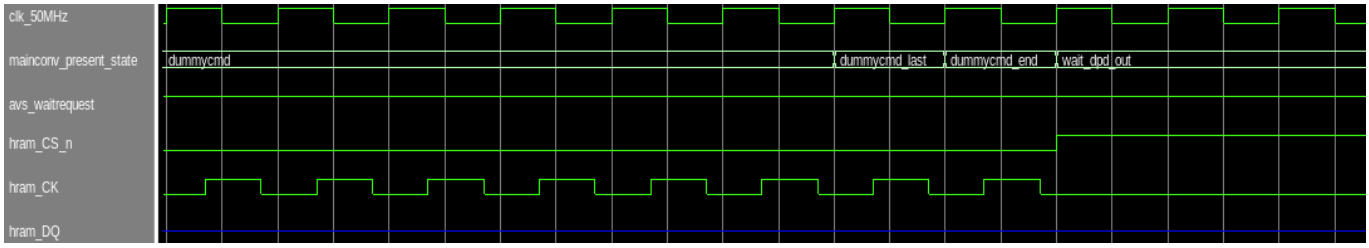

Figure 5.7: DPD mode exit.

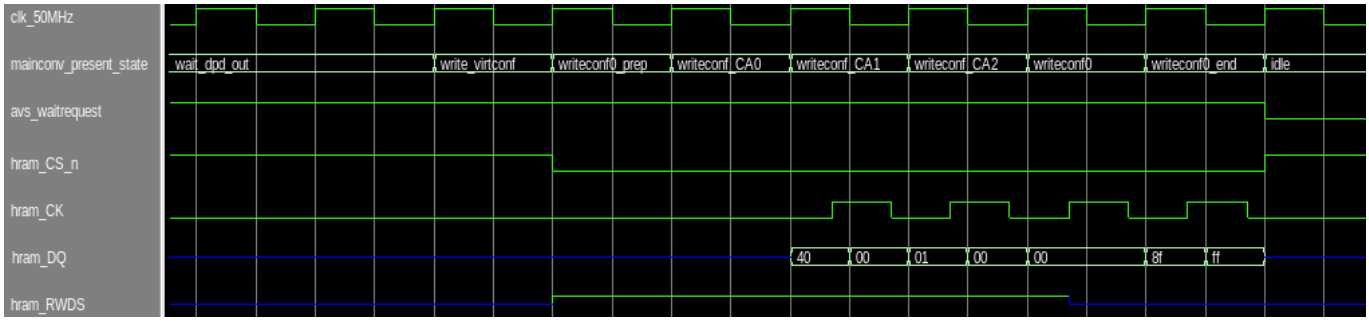

Figure 5.8: Automatic CR0 update after going back to normal mode.

5.2 Final Simulation

The interface converter is inserted in a processor-based system to simulate it against a large number of data, as we can see in figure 3.1. The *Nios II* processor is programmed to read four different signals provided by the *input generator* and to drive four different LEDs according to their value: when a status signal is set, the LED associated with it is turned on and viceversa. The HyperRAM (together with the interface converter) is employed as data memory for the *Nios II* processor.

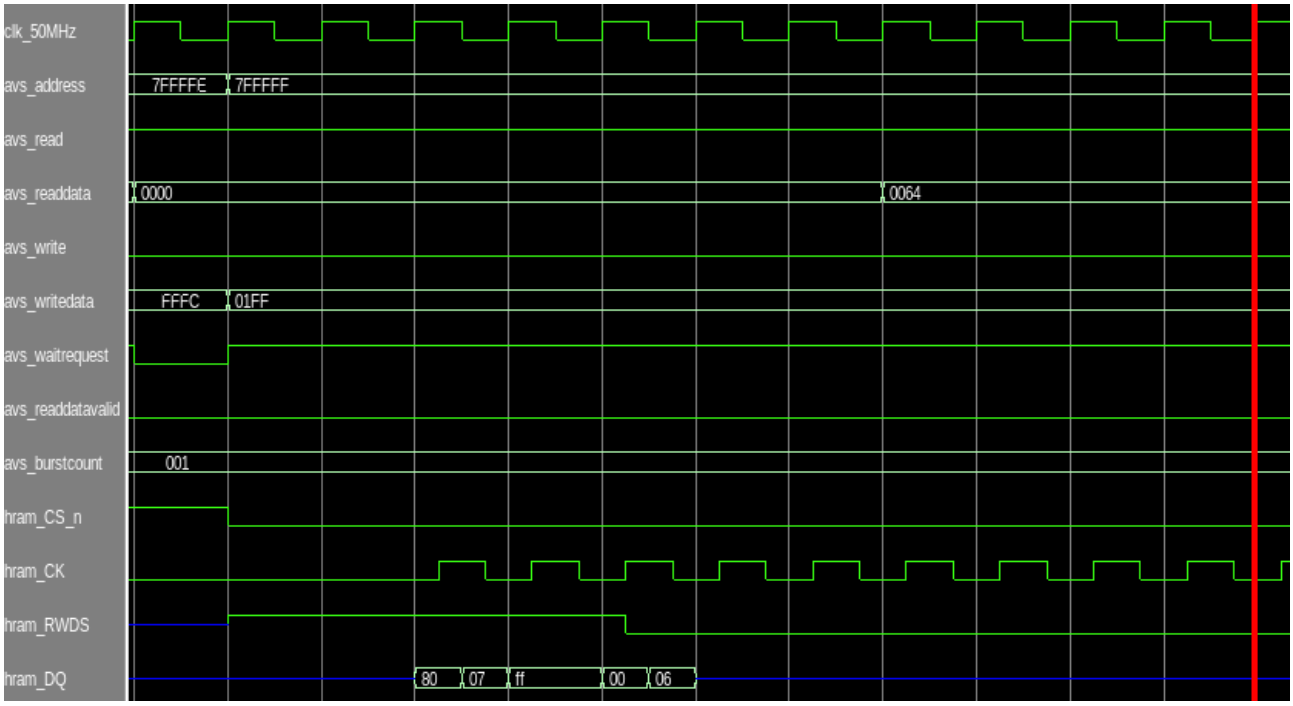

Figure 5.9: One of the many read operations driven by the processor - part 1.

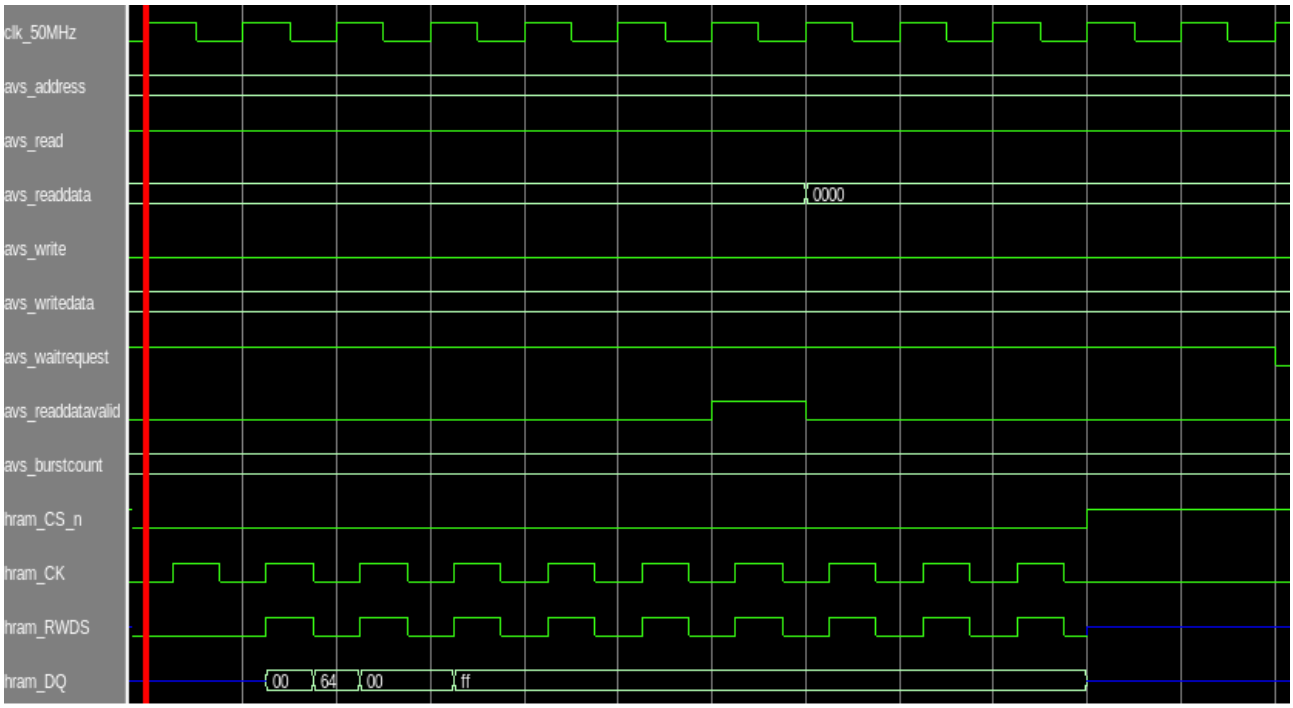

Figure 5.10: One of the many read operations driven by the processor - part 2.

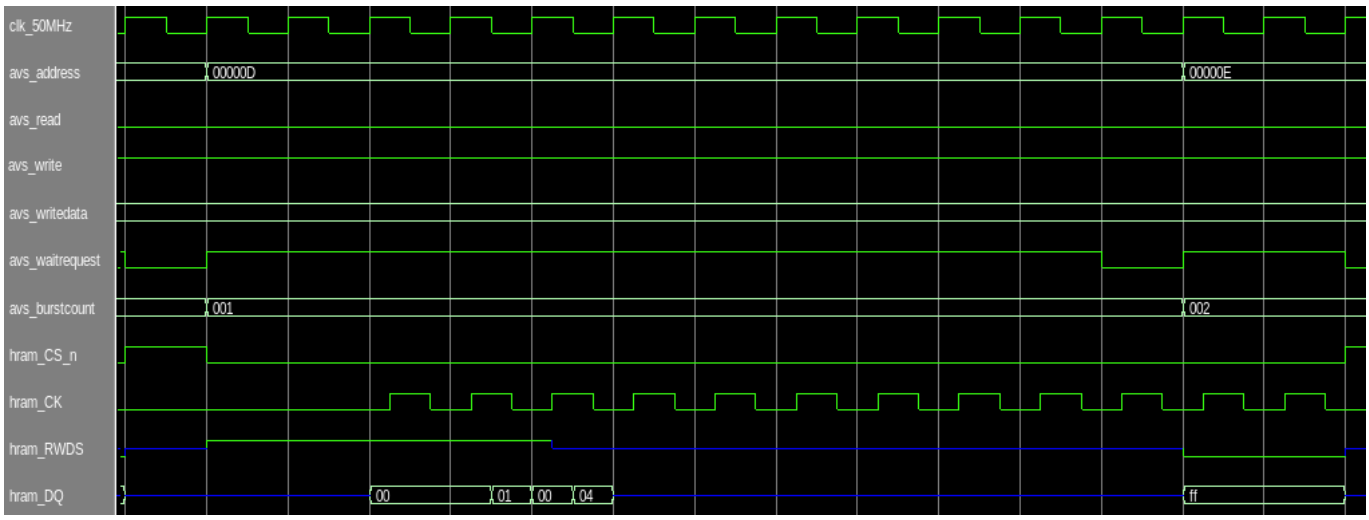

Figure 5.11: One of the many write operations driven by the processor.

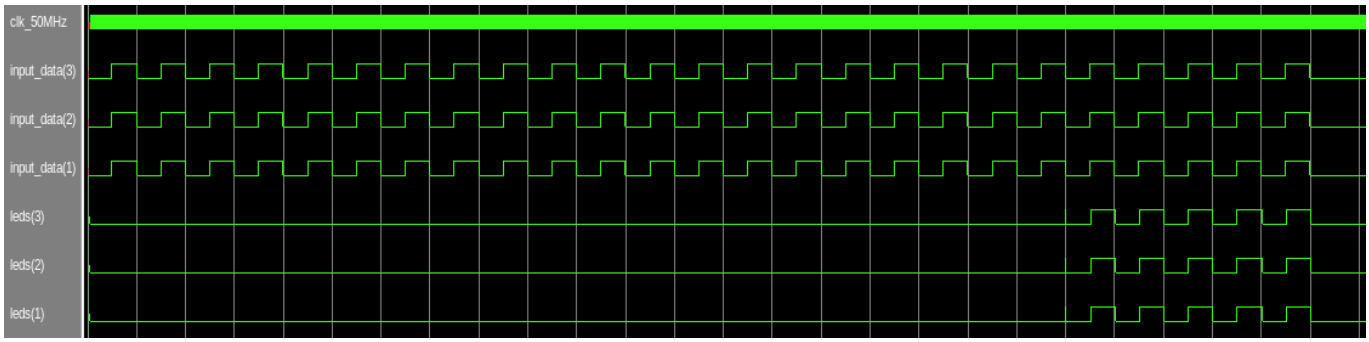

Figure 5.12: Processor-based system behavior.

At first, the processor-based system must wait for the memory to power-up, therefore it cannot process the input values, as we can see in figure 5.12. Indeed, during this initial time interval the LEDs are constantly turned off regardless of the value of the inputs. When the memory is powered-up, the processor drives the LEDs according to the input values as expected.

5.3 Test on VirtLAB

FUTURE EXTENSIONS

The designed system is capable of correctly interface the HyperRAM with the Intel Avalon bus. However, it is still possible to work on some details to improve its performance. In particular:

- As described in section 4.1, the working frequency of the memory has been reduced to 50 MHz due to the limitations related to the oversampling. However, all the other components of the system are able to work at 100 MHz. Indeed, exploiting a controlled combinational delay (instead of the oversampling) to implement the *readdata converter* would allow to push the working frequency up to 100 MHz.
- On the Avalon side, the burst lenght is characterized by a parallelism of 11 bits. However, the burst lenght value cannot be higher than 181, therefore only 9 bits are actually useful, as described in section 4.8.1. For this reason, the counter and the comparator inside the *synchronizer* (*burstlen_counter* and *burstlen_cmp* referring to figure 4.17) can be implemented with a reduced parallelism, saving power and resources.
- The virtual configuration register allows to dynamically modify only two parameters of the HyperRAM (working mode and memory access latency). However, the design can be easily modified to allow the host to access also other parameters.

BIBLIOGRAPHY

[1] Massimo Ruo Roch, Maurizio Martina, *VirtLAB: a Low-Cost Platform for Electronic Lab experiments*, Sensors