CHAPTER 1 CSL FIFO

TABLE 1.1 Chapter Overview

1.1 CSL FIFO Overview
1.2 CSL FIFO Concepts
1.3 CSL FIFO Examples
1.4 CSL FIFO Checker
1.5 CSL FIFO Reports
1.6 CSL FIFO Miscellaneous

1.1 CSL FIFO Overview

1.1.1 FIFO Description

A FIFO or queue is a first in first out hardware circuit. The CSL supports the specfication of fifos clocked by either single clocks or two clocks (synchronous fifo). The FIFO generates empty and full signals to indicate whether data can be pushed or poped respectively. Optionally low and almost full signals can be generated to indicate to the downstream and upstream logic that the FIFO has a certain number of entries. Typcially watermarks are used as "early warning" signals by downstream and upstream logic to stall the consumer or producer logic prior to reaching an empty or full condition respectively. The reason that watermarks are used is to prevent a FIFO underflow or overflow condition in the case of either distributed control or a clock domain crossing FIFO where several cycles are required to communicate the full or empty condition. Instead the producer or consumer logic is notified that the FIFO has reached a watermark and that the producer or consumer should stop or start an activity. For example, the consumer may want to know when there are a certain number of entries in the FIFO which can be used to perform an operation. Once the operation is started it expects a certain number of data elements to be available and the operation cannot be stalled. Or a FIFO can be connected to a centralized memory which uses a DMA engine

to pull data out of the FIFO using a burst mode over a system on chip (SOC) bus which does not support split transactions. Once the burst operation is started it cannot be stopped until all data is transmitted. FIFOs are configured using the CSL. The CSL FIFO generator will use SRAM components to build the SRAM portion of the FIFOs. Either synchronous or asynchronous SRAM memories components will be used depending on the CSL FIFO specification. SRAM components which exactly match the width and depth of the FIFO will not be created; instead FIFOs which are implemented using SRAM components which are greater than or equal to the size of the specified width and depth. The SRAM component library will range from <smallest_width> x <smallest_depth> to <largest_width> x <largest_depth>. The SRAM components increase in size by powers of 2. The unused portions of the SRAM will have the corresponding input and output signals tied to ground.

1.1.2 Implementation:

FIFOs are implemented using either flip-flops/register or SRAMs for their memory arrays.
FIFO size determines whether to use flip-flops/register or SRAM's for the actual implementaion.
FIFOs generate full and empty signals for the reader and writer respectively.
FIFOs use read and write counters to point to the entries to read and write respectively.
FIFOs can be modified to have behaviors that do not follow the strict definition of a FIFO. For example the FIFO could be first in/random read. In this case it is not strictly a FIFO but we include it in this discussion. The FIFO can be physically located with its control logic or the control logic can be separated from the FIFO and located in different units. The FIFO's memory array can be one logical unit and many physical units which are located together or distributed.

1.2 CSL FIFO Concepts

1.2.1 FIFO Parameters

1. A FIFO can be written (with a push operation) or read (with a pop operation). There can be a clock signal for read, write or for both.

A FIFO has 1 or 2 clock domains.
i) If the read and the write signals are using the same clock then the FIFO is a single clock domain device.
ii) If the read and the write signals are using different clocks then the FIFO is a 2 clock domain device.

2. reset: The FIFO control is used with an asynchronus or synchronus reset signal. Reset signals should be avoided for power consumption reasons. The contents of the data words should not be reseted in order to save power.

3. depth: A FIFO has N words (levels). This is the depth of the FIFO. The physical implementation of the FIFO may have more words than is required in order to use pre-built RAM macros or to take advantage of a particular soft RAM macro.

4. word size: The FIFO word size is the width of the data word in the FIFO.

5. valid bit: Optionally each data word in the FIFO can have an optional valid bit (valid == 1). The valid bit is reset to 0 (invalid) when the FIFO is reset.

6. full condition: FIFOs generate a full signal which indicates when the FIFO has no more entries available to write. The full condition occurs in a standard FIFO when the write pointer equals the read pointer. In practice the FIFO is full when there are still entries left in the FIFO.

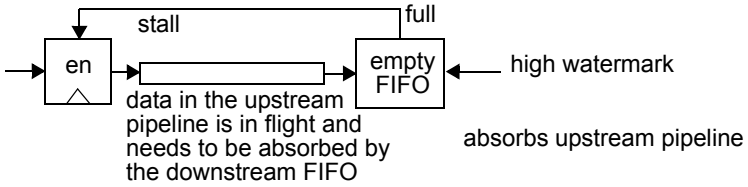
7. almost full: The FIFO full condition can be set when a certain value which is less than the total number of entries in the fifo is encountered. After the almost full is asserted by the fifo the transactions in flight in the upstream pipeline can be written to the remaining empty words in the FIFO. The full signal calculation is delayed by the read address synchronization delay of 2 or more cycles. The read address is synchronized with the write clock and the write clock synchronized read address is subtracted from the write clock to generate the full signal. There is still data in the upstream pipeline which is feeding the FIFO and the data needs to be written to the FIFO.

There are n cycles between the actual time that the FIFO hits the almost full due to the address synchronization delay. During the synchronization delay data in the upstream and downstream pipelines continues to flow. Since data continues to flow for n cycles n push operations can occur. To accomodate the additional n push operations n words are reserved in the FIFO above the almost full. The real full condition is n plus the number of words reserved.
almost_full_external_full = internal_full or *num_words > almost_full*.

FIGURE 1.1 FIFO absorbtion

The almost_full signal is set to a value below the real FIFO full condition to allow for the upstream pipeline data to be written to the FIFO. For example, if the FIFO has 10 entries the upstream pipe will write 2 entries to the FIFO after receiving the full signal then the almost full is 8. After receiving the full signal there will be 2 entries in the upstream pipeline. The two entries will be pushed onto the FIFO.
Almost full (almost_full) and almost empty (almost_empty) are subtracted from the total number of entries which have been writen in the FIFO. Full = 0 or a negative result.

FIGURE 1.2 FIFO flow control

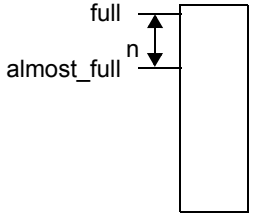


A watermark is a level in the FIFO that is used to detect a certain condition. The condition can be a low point or a high point in the FIFO. The watermark tells the adjacent units that the FIFO is getting close to full or getting close to empty.

1.2.2 Almost full

The user can specify the almost_full. Or the almost full can be programmatically computed by the cslc based on the analysis of the upstream logic and the number of cycles required to synchronise the read address and the compute signal.

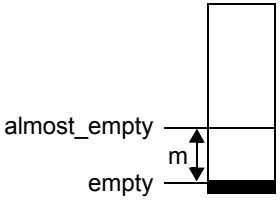
FIGURE 1.3 Almost full trigger



1.2.3 Almost empty

Almost emptys alert the design that a lower bound (LB) has been reached. The LB tells the down-stream logic not to pull data from the FIFO.

FIGURE 1.4 Emptying the FIFO



1. physical queue : n queues to avoid the head of line blocking(HOLB) problem. If there are n units writing to the logical FIFO then n physical FIFOs can be used to handle each of the writes. Then an arbiter can be used to select each of the input FIFO's in turn to avoid the case where one

unit's input will not be able to be popped for some reason and therefore none of the other units can make forward progress because their FIFO's will fill up.

2. counter type (normal or grey coded) : If the FIFO uses one clock domains then a normal counter is used. If the FIFO uses two clock domains then a grey coded counter is used to avoid glitches on the synchronizers that are connected to the counter.

3. scheduler/arbiter : If there are n units which write to the FIFO then an arbiter is used. The arbitration scheme may have to support high priority interrupts.

4. random access write : The FIFO entries can be addressed and written randomly.

5. random access read : The FIFO entries can be addressed and read randomly.

6. credit based control : The read and the write side use credits to read and write the FIFOs. The FIFO does not contain any control logic. The write side is initialized with all of the credits. The read side is initialized with zero credits.

NOTE:NOTE: Explain this system how it works. What is a credit?

7. credit based write : When the producer writes to the FIFO the producer subtracts 1 from its credits and the consumer adds 1 to its credits.

8. credit based read : When the consumer reads from the FIFO the consumer subtracts 1 from its credits and the producer adds 1 to its credits.

9. input bandwidth specification : The input bandwith is the amount of data written per cycle specified in the width of the data word.

10. output bandwidth specification : The output bandwith is the amount of data read per cycle specified in the width of the data word.

11. bandwidth sizing : The FIFO depth should be sized based on the ratio of the output bandwidth to the input bandwidth. It is a balance of the latency of the system, data bandwidth of the system bus, I/O protocol overhead and data bandwidth for the I/O protocol bus the user is connecting to, with the IP FIFO buffering in-between. Also, in every FIFO size decision the user needs to understand the trade off of size (gate count) compared to performance (throughput).

NOTE:NOTE: Need an equation here.

12. asynchronous or synchronous reset : Specify whether the FIFO has an asynchronous or synchronous reset.

13. arbitrated control using a credit based scheme : The FIFO is a single physical memory and resides in unit X. Units A,B,C write to the FIFO. There is an input mux. There is an arbiter which handles requests from the different units.

14. **distributed control using a credit based scheme** : The central FIFO is a single physical memory and resides in unit X. The central FIFO has an input mux. Units A,B,C write to the central FIFO from their local FIFOs. There is an arbitration scheme which is distributed to the driving units. The reason for this is so that the units can control when they pop the data out of their local FIFOs to load into the central FIFO.

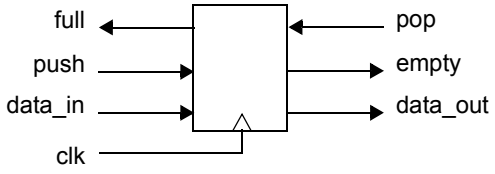
15. **distributed FIFO's which drive a multiplexer bus.** NOTE: Write something about this.

16. **split FIFO to drive different clock domains** : the problem here is that FIFO(clka) receives data in the clka domain and needs to drive data out to two different units which are in different clock domains. The two different units pull the data from the two output FIFOs (FIFO(clkb) and FIFO(clkc)).

1.2.4 CSL FIFO configuration functions

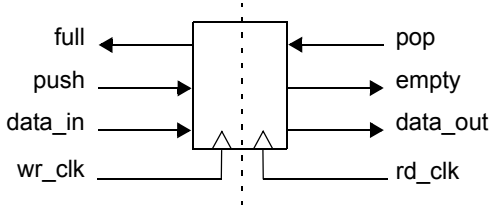
A synchronous FIFO(see Figure 1.5) uses the same clk for write and read operations.

FIGURE 1.5 Synchronous FIFO



A asynchronous FIFO(see Figure 1.6) uses the wr clk for write operations and rd clk for read operations.

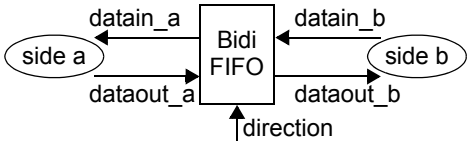
FIGURE 1.6 Asynchronous FIFO



1.2.5 Bidirectional FIFO

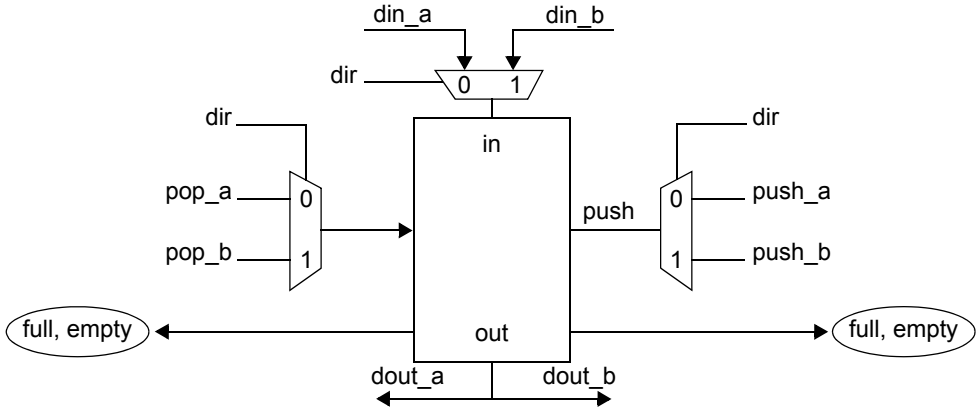
Units which switch data flow directions can save area by using a bidirectional FIFO.

FIGURE 1.7 Bidirectional FIFO



The bidirectional FIFO can be written by side a and side b. The direction flow is controlled by a direction bit.The direction bit controlls the input and is ANDed with the FIFO's valid bit to create the qualified output bit for the output.

FIGURE 1.8 Bidirectional FIFO



1.2.6 Tansaction checking

The FIFO data words can have a tag field associated with them when required. An example of a use of the tag field is when a split transaction bus is used, the requested transaction is sent with a value and the reply is sent with the same value. Then the FIFO entry with the tag can be accessed using a Content Addressable Memory(CAM) lookup scheme and the value associated with the tag can be popped. When an entry is written to the FIFO a tag is generated and pushed into the FIFO along with the data and a valid bit. The unit which pushed the data onto the FIFO is sent the tag associated with the pushed data. If the unit wants to cancel the transaction in the FIFO the unit can send a cancellation request with the tag and the FIFO will invalidate the word in the FIFO. See figure Figure 1.10

FIGURE 1.9 Transaction checkings

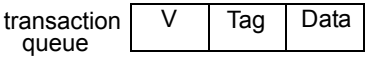
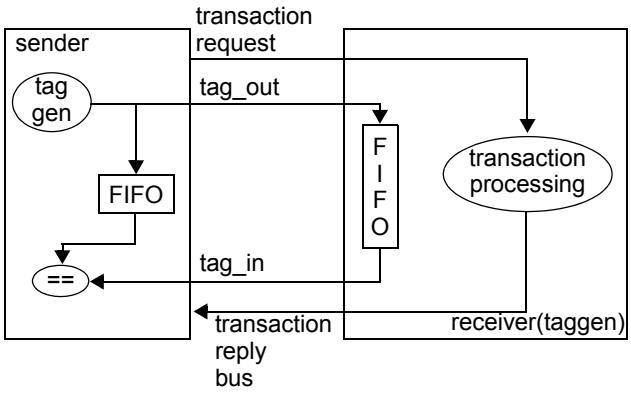


FIGURE 1.10 Transaction tag



Tags are generated by the tag generator and they are associated with specific transactions. Tags are saved in a tag FIFO.
The transactions consisting of the data and the tag is sent to another unit where the transaction is processed. The CPU sends a transaction reply containing the tag back to the requesting unit. The requesting unit extracts the transaction tag and pops the transaction tag from the transaction FIFO and compaires it to the expected transaction value. A CAM is used for this.
Transactions are transmitted with their corresponding tags. An inorder transaction stream returns the tags in the same order that the tags were sent. An out of order transaction stream returns the tags in any order.
FIFO entries are removed based on a tag number and the tag number is pushed on the tag queue. If the reply transaction has a tag which does not match a tag in the transaction queue then an error is generated.

1.2.7 FIFO signal qualification

FIGURE 1.11 Full signal qualified with stall signal

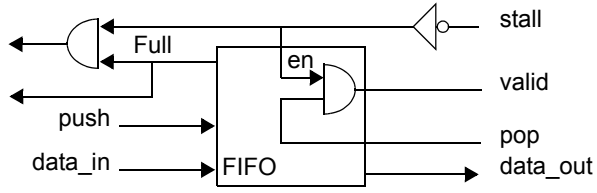
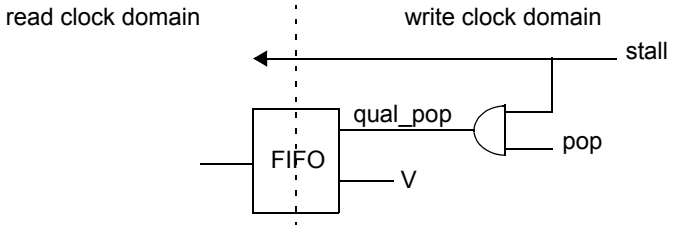


FIGURE 1.12 Pop signal qualified with the stall signal



These are the pop signals.

1.2.7.1 Data synchronization across clock domains

- gray coded
- metastability
- Mean time between failure (MTBF)

TABLE 1.2 Full and empty signal generation

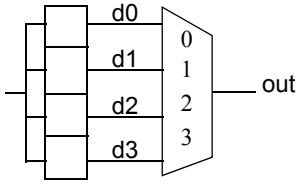
Delay	almost_full	almost_empty	used in
upstream pipeline absorbtion			full
delay trough read synch addr		X	full
delay trough write synch addr	X		empty

1.2.8 Asynchronous FIFO Synchronizer

Using a FIFO to synchronize an asynchronous data stream moves the synchronization out of the data path and facilitates flow control. This method involves shifting data into FIFO using the transmit clock and shifting data out of the FIFO using the local clock. Both clocks may be aperiodic. Synchronization is performed on the transmit and receive pointers to generate an “empty“ signal in the local clock domain and a “full“ signal in the transmit clock domain.

The FIFO in the center of the Figure 1.12 can be realized as a set of flip flops with clock enables and an output multiplexer as in Figure 1.13. In practice, a register file or transparent latches are often used to save area. The read and write pointers arrange that sequence data into and out of the FIFO. The write pointer is advanced synchronously with the transmit clock, xclk when the shiftIn signal is asserted. Similarly the read pointer is advanced synchronously with clk when shiftOut is asserted. Data is shifted into the FIFO in the receive clock domain. No synchronization is required unless the receiver attempts to read past the last word in the FIFO or the transmitter attempts to write beyond the FIFO capacity.

FIGURE 1.13 FIFO implemented using FF arrays



Synchronization is required to detect the “full“ and “empty“ conditions of the FIFO because this requires comparing the two pointers that are in different clock domains. In the figure, the read (write) pointer is synchronized with the transmit (local) clock, and the comparison tp check for full (empty) is performed in the transmit (local) domain. This method is preffered to the alternative of performing the comparison directly on the two pointers without synchronization and then synchronizing the resulting asynchronous full and empty signals because the alternative delays the response of the empty (full) signal of the ShiftOut (shiftIn) signal. The method illustrated immediately sets empty (full) when the last word is shifted out (in) , avoiding underruns (overruns). The synchronization delay is only incurred when information must be passed between the two clock domains. For example, when the transmitter shifts the first word into an empty FIFO, a synchronizer delay is required before the receiver detects empty as low.

FIGURE 1.14 Clock domain crossing fifo

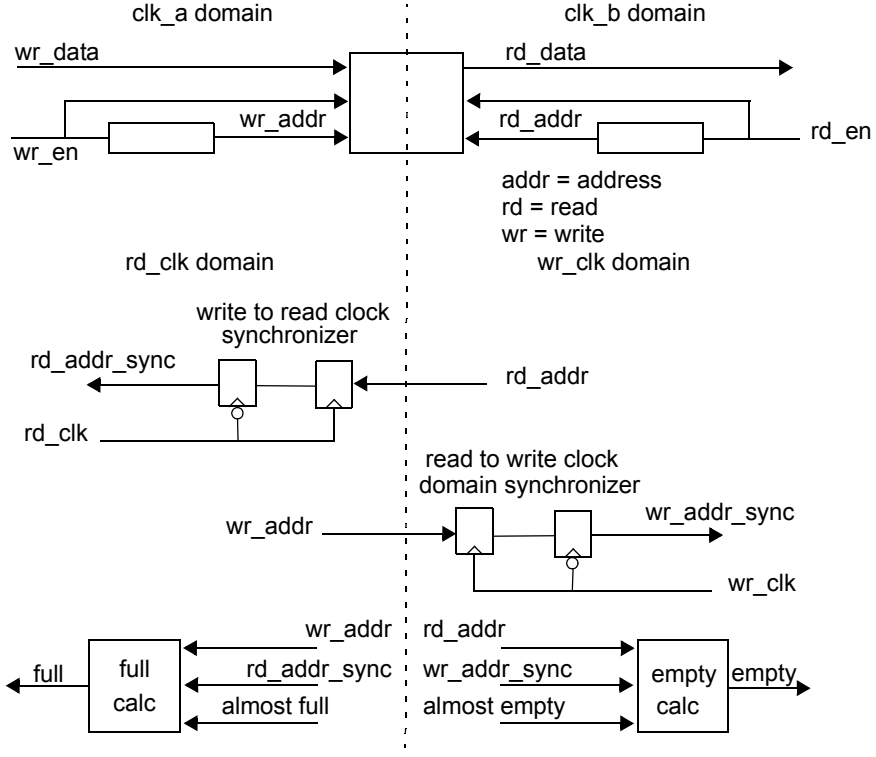
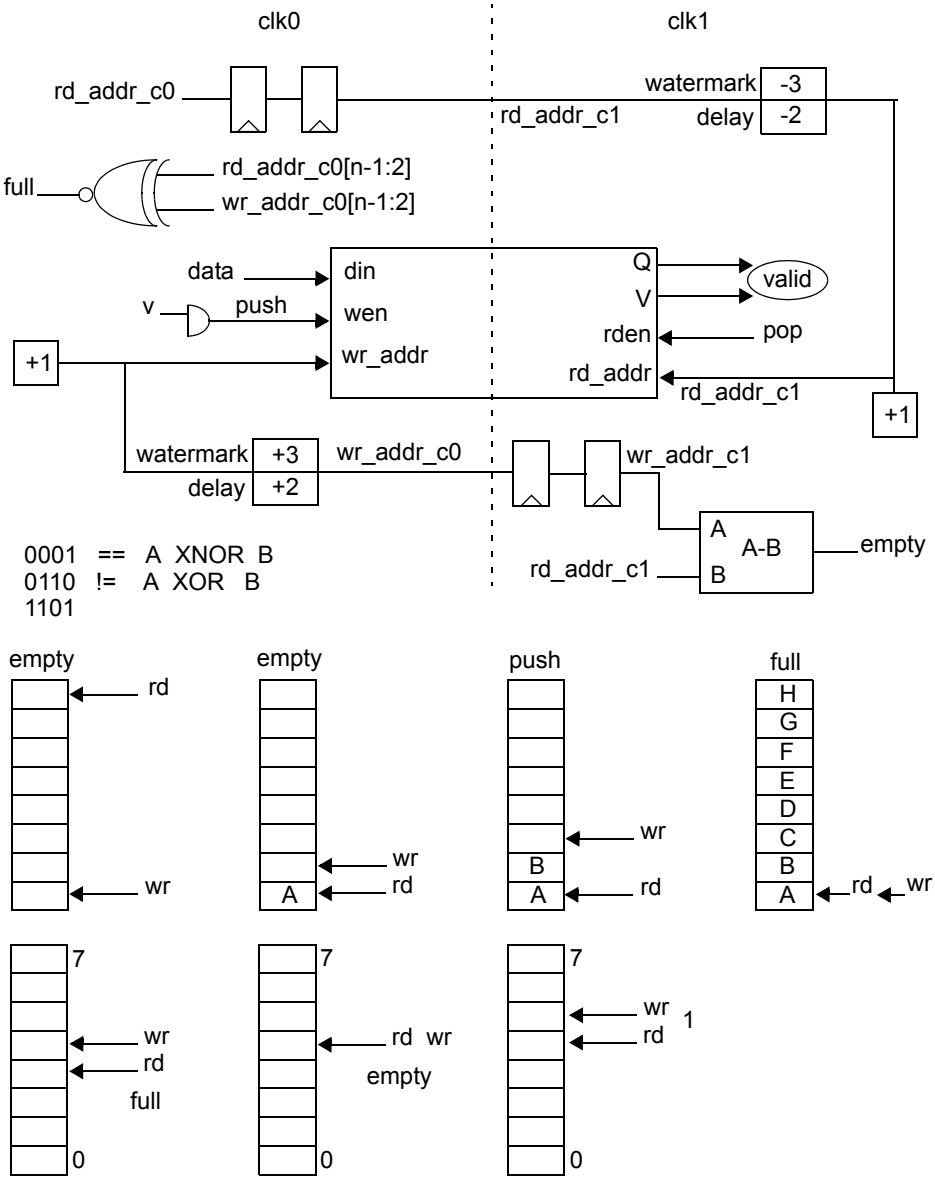


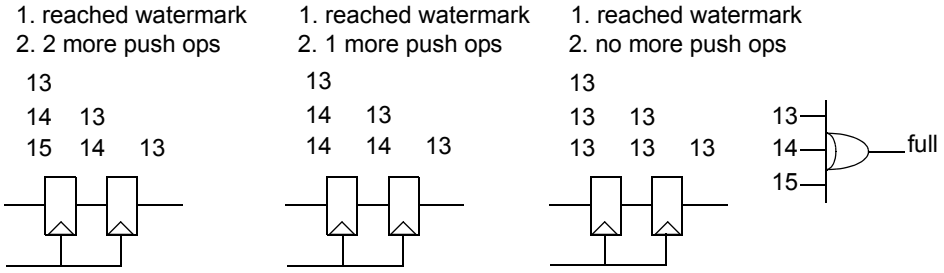
FIGURE 1.15 Clock domain transaction sequence



The diagram with the dashed line uses a suffix or either c0 or c1 to denote a signal in the clock 0 (write clock domain) or a signal in the clock 1(read clock domain).
Metastability (a state which exist between either "valid" digital logic state -an undefined voltage) leads to synchronization error.
Full signal is asserted when almost_full is reached - 2 empty slots in FIFO but because of 2 clock

delay trough synchronizers and possible cycle delay to synch 3 more pushes could occur. The full flag is asserted when the fifo counter is greater than the almost_full watermark.

FIGURE 1.16 Full generation



The FIFO has n entries. Full is generated at n-4 because it takes 3 cycles for the full signal to be generated in our example. So three cycles of data can be written into the FIFO before the upstream enable signal is turned off. The upstream enable signal is qualified with the full signal.

1.2.8.1 Synchronizer clock ratios

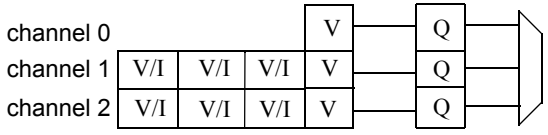
TABLE 1.3 Producer/Consumer clock Relation

Producer/Consumer Clock Relation	Phase Relation	Notes
prod_clk == consum_clk	in phase	
prod_clk == consum_clk	out of phase	
prod_clk < consum_clk	in phase	almost_empty and syncronizer needed
prod_clk < consum_clk	out of phase	almost_empty and syncronizer needed
prod_clk > consum_clk	in phase	almost_full and syncronizer needed
prod_clk > consum_clk	out of phase	almost_full and syncronizer needed
prod_clk << consum_clk	in phase	almost_empty and syncronizer needed
prod_clk << consum_clk	out of phase	almost_empty and syncronizer needed
prod_clk >> consum_clk	in phase	almost_full and syncronizer needed
prod_clk >> consum_clk	out of phase	almost_full and syncronizer needed
prod_clk multiple of consum_clk	in phase	almost_full and syncronizer needed
prod_clk multiple of consum_clk	out of phase	almost_full and syncronizer used
no relation		

The asynchronous FIFO synchronizer has a lower probability of synchronization failure and inherent flow-control. The probability of synchronization failure is reduced because synchronizers. The FIFO read and write address counters are implemented using gray code counters, which are effectivly

static machines. The gray code counter is implemented using flip-flops. The read and write address counters are synchronized to the other clock domain. A single synchronization is performed per word and only when the FIFO toggles between empty and non-empty states.
The FIFO synchronizer provides inherent flow control between the upstream and downstream logic, via the full and empty signals. Every symbol that is shifted into the FIFO will be shifted out in order. Data can be dropped or duplicated because of differences between the write and read clocks.

FIGURE 1.17 Channels with different bandwidths



1.2.8.2 Asynchronous FIFO Design CLK Ratios

If the sending clock is greater than twice the receiver clock then the sender clock may appear to skip a count in the count sequence in the receiver clk domain:
sender count 0 1 2 3 4 5 6 7
sender count 0 0 2 2 4 4 6 6
If a synchronizer is not used 2 and 3 are skiped.

The problem is that almost_empty, full and empty signals need to account for the extra skip so register leak will not occur. A solution can be to add extra empty 3/4 in FIFO to absorb up stream flow.

1.2.8.2.1 FIFO sizing examples

FIGURE 1.18 FIFO usage

FIGURE 1.19 Receive batch send single transactions

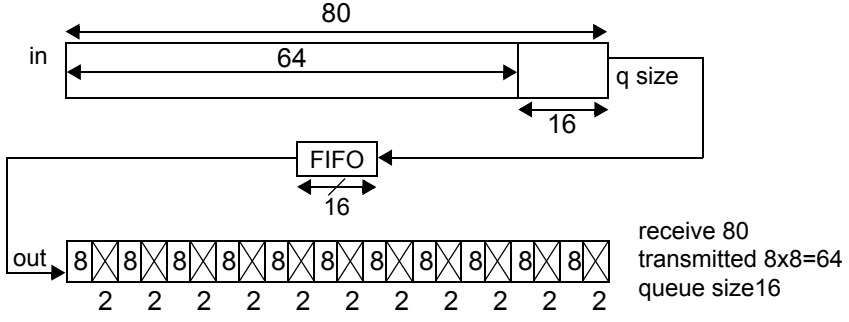


FIGURE 1.20 FIFO size 10

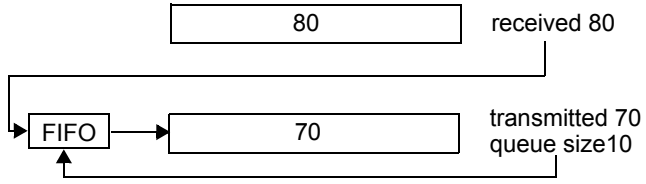
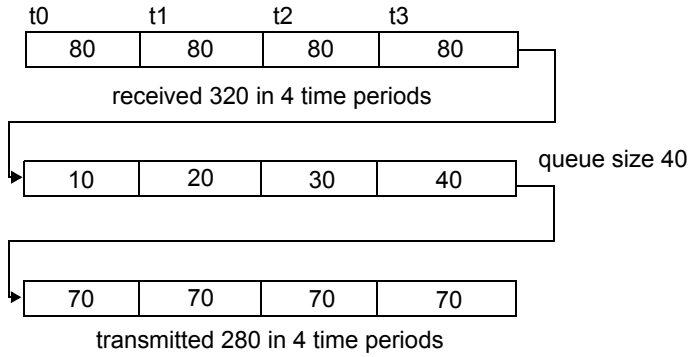


FIGURE 1.21 FIFO size 40



1.2.8.3 Gray Coded Read and Write Counters

The read and write counters are gray coded. Only one counter output bit changes per clock cycle. There are no "glitches" on the output ports of gray coded counters because there are no longest and shortest paths through the combinational logic in the gray coded counter. Instead a sequence of numbers is generated by the counter which have the property that only one bit changes per cycle. The read counter is incremented by a push. The write counter is incremented by a pop. The counters are synchronized with the opposite clk domain in order to compare the read and the write values.

1.2.9 FIFO/queue characteristics

1. Synchronization

- counter synchronization
- data synchronization
- control synchronization
- being captured by the clk domain synchronizers.
- gray coded counters instead of normal counters to avoid glitches
- time in cycles to synchronize->low watermark
- number of cycles in pipe to absorb->almost full

2. Flow Control: Absorbing upstream data before the pipeline can be stopped automatically popping the fifo to supply data to the datapath.

3.Bandwidth

- producer bandwidth
- consumer bandwidth
- min/max bandwidth per client

4. Control Type

- local control
- distributed control using a credit based scheme

5. Asynchronous Circuits //detalis

6. Metastability: the ability of a non-equlibrium electronic state to persist for a long (and theoretically unboundable) period of time.

7. Probability of a Failure

- Scheduling/arbitrating writes to the FIFO from 2 or more sourcres.
- Scheduling/arbitrating reads from the FIFO from 2 or more destinations.
- Assuring fairness
- Head of line blocking problem (HOLB)

Each source has its own FIFO (the input FIFO is split). An arbitration unit controls the FIFO select . The output of each FIFO is connected to a MUX. When designing a circuit we can use a switch to connect n inputs to n outputs. The inputs and/or outputs can be connected to a FIFO.

1.2.10 Fifo status registers

CSLC offers programmable status generation registers for FIFO flow control. These registers counter the number of writes to:

- almost empty register: when the counter equals the almost empty value the almost_empty output signal is enabled
- almost full register: when the counter equals the almost full value the almost_full output signal is enabled
- empty register: when the counter equals the empty value then the empty output signal is asserted
- full register: when the counter equals the full value then the full output signal is asserted

1.2.11 Fifo ports and logic

When creating a fifo, the compiler automatically creates default ports and logic for the respective unit. Custom **add_logic()** commands add functionality and/or ports to the unit. The ports, their functionality and naming is detailed below:

TABLE 1.4

Port Name	Dir	W	Generated by	Descrition
sync_reset	input	1	Automatically	Port for synchronous reset signal
data_in	input	ud	Automatically	Port for input data signal
data_out	input	ud	Automatically	Port for output data signal
pop	input	1	Automatically	Port for pop signal
push	input	1	Automatically	Port for push signal
full	input	1	Automatically	Port for full signal
empty	input	1	Automatically	Port for empty signal
async_reset	input	1	add_logic(async_reset);	Port for asynchronous reset signal
pushback	input	1	add_logic(pushback);	Port for pushback signal
programmable_depth	input	ud	add_logic(programmable_depth, default_depth);	Port for programmable depth signal
priority_select	input	ud	add_logic(parallel_output, all | vector_of_addresses);	Port for priority select signal
stall	input	1	add_logic(stall);	Port for stall signal
stall_rd_side	input	1	add_logic(stall_rd_side);	Port for read side signal
stall_wr_side	input	1	add_logic(stall_wr_side);	Port for write side signal
wr_addr	output	ud	add_logic(output_wr_addr);	Port for write address signal
rd_addr	output	ud	add_logic(output_rd_addr);	Port for read address signal
priority_bypass	output	ud	add_logic(parallel_output, all | vector_of_addresses);	Port for priority bypass signal
<sram_rd>_data	output	ud	add_logic(sram_rd);	Port for sram_rd_data signal
<sram_rd>_en	input	1	add_logic(sram_rd);	Port for sram_rd_en signal
<sram_rd>_addr	input	ud	add_logic(sram_rd);	Port for sram_rd_addr signal
<sram_wr>_addr	input	ud	add_logic(sram_wr);	Port for sram_wr_addr signal
<sram_wr>_data	input	ud	add_logic(sram_wr);	Port for sram_wr_data signal
<sram_wr>_en	input	1	add_logic(sram_wr);	Port for sram_wr_en signal
wr_release	input	1	add_logic(wr_release);	Port for wr_release signal

TABLE 1.4

Port Name	Dir	W	Generated by	Descrition
almost_empty	output	1	add_logic(almost_empty,address);	Port for almost_empty signal
almost_full	output	1	add_logic(almost_full,address);	Port for almost_full signal
credit	output	ud	add_logic(credit);	Port for credit signal
rd_credit	output	ud	add_logic(rd_credit);	Port for read credit signal
wr_credit	output	ud	add_logic(wr_credit);	Port for write credit signal
overflow	output	1	add_logic(flow);	Port for overflow signal
underflow	output	1	add_logic(flow);	Port for underflow signal

TABLE 1.5

Port Name	Description
sync_reset	Set a synchronous reset signal. Reset is synchronous with clock
data_in	FIFO data input signal name.
data_out	FIFO data output signal name.
pop	FIFO data pop signal name.
push	FIFO data push signal name.
full	Generate a FIFO full signal with the name *signal_name*. if the FIFO is asynchronous then then the signal is generated in the write clock domain from the clock synchronized version of the read address and the write clock domain read address.
empty	Generate a FIFO empty signal with the name signal_name. If the FIFO is asynchronous then then the signal is generated in the read clock domain from the clock synchronized version of the write address and the read clock domain read address
async_reset	This is an asynchronous reset command. When the reset signal is on the low level (logic “0“) the fifo is filled up with 0 values, the clock signal edge doesn’t matter.
pushback	Push back the value popped off of the top of the fifo. Don't really push it back. Instead move the read pointer back one position. This feature requries that the full and empty signals are generated from logic that seperates the write and the read pointers by at least one postion to allow the push_back operation to change the read pointer.

TABLE 1.5

Port Name	Description
programmable_depth	The FIFO depth is controlled by an input to the fifo. The write and read pointers are reseted when their value equals the value of *signal_name.* The *default_depth* is the maximum depth for the fifo, and the maximum value that *signal_name* can have.
priority_select	This signal is asserted at the same time as a write to the fifo. When asserted the write to the FIFO is sent to the input of the high prioriy bypass unit.
stall	The entire FIFO is stalled and pop/push operations are not allowed. This signal is used for flow through fifos which pop themselves whenever the FIFO is not empty
stall_rd_side	The read side of the FIFO is stalled andand FIFO pop operations are not allowed.
stall_wr_side	The write side of the FIFO is stalled and FIFO push operations are not allowed
wr_addr	The FIFO write address is available at the FIFO interface
rd_addr	The FIFO read address is available at the FIFO interface
priority_bypass	This signal is asserted at the same time as a write to the fifo. When asserted the write to the FIFO is sent to the input of the high prioriy bypass unit
<sram_rd>_data	The FIFO read side is an SRAM interface. The FIFO can be read in any order.
<sram_rd>_en	The FIFO read side is an SRAM interface. The FIFO can be read in any order.
<sram_rd>_addr	The FIFO read side is an SRAM interface. The FIFO can be read in any order.
<sram_wr>_addr	The FIFO write side is an SRAM interface. The FIFO can be written in any order.
<sram_wr>_data	The FIFO write side is an SRAM interface. The FIFO can be written in any order.
<sram_wr>_en	The FIFO write side is an SRAM interface. The FIFO can be written in any order.
wr_release	This will set the wr_addr_release_limit register equal to the current wr_addr_hold_limit used in conjuction with the wr_hold_arch switch

TABLE 1.5

Port Name	Description
almost_empty	A almost_empty signal is generated when the FIFO almost empty address is reached If the FIFO is asynchronous then then the signal is generated in the read clock domain. Note that there is a delay of n cycles, where n is the synchronization delay of the FIFO write address, until the almost_empty signal is generated. Optionally use *name* as the name of the almost_empty signal.
almost_full	A almost full signal is generated when the FIFO almost full address is reached If the FIFO is asynchronous then then the signal is generated in the read clock domain. Note that there is a delay of n cycles, where n is the synchronization delay of the FIFO write address, until the almost_full signal is generated. Optionally use *name* as the name of the almost_full signal.
credit	Use a distributed credit debit mechanism to control the fifo. No full or empty signals are generated. Instead FIFO write status/control is handled by the producer and the FIFO read status and control is handled by the consumer.
rd_credit	Use a distributed credit debit mechanism to cotnrol the FIFO read. No full or empty signals are generated. Instead fifo write status/control is handled by the producer and the fifo read status and control is handled by the consumer.
wr_credit	Use a distributed credit debit mechanism to cotnrol the FIFO write. No full or empty signals are generated. Instead fifo write status/control is handled by the producer and the fifo read status and control is handled by the consumer.

Ports automatically generated by cslc for fifo:

```
port: input - sync_reset
```

DESCRIPTION :

Create the port *sync_reset* for the synchronous reset signal. Reset is synchronous with clock

```
port: input - data_in
```

DESCRIPTION :

Create the port *data_in* for data input signal. The signal width depends on the with of the FIFO that is user specified.

```
port: input - data_out
```

DESCRIPTION :

Create the port *data_out* for data output signal. The signal width depends on the with of the FIFO that is user specified.

port: input - pop

DESCRIPTION :

Create the port *pop* for pop signal. The signal width is 1and is used get the last saved data in the fifo.

port: input - push

DESCRIPTION :

Create the port *push* for push signal. The signal width is 1and is used store data in top of the fifo.

port: input - full

DESCRIPTION :

Generate a FIFO full port with the name *full*. If the FIFO is asynchronous then then the signal is generated in the write clock domain from the clock synchronized version of the read address and the write clock domain read address.

port: input - empty

DESCRIPTION :

Generate a FIFO empty port with the name *empty*. If the FIFO is asynchronous then then the signal is generated in the read clock domain from the clock synchronized version of the write address and the read clock domain read address.

1.3 CSL FIFO Examples

A FIFO (first in first out) or queue is used to buffer data when the downstream pipeline stages are busy and/or FIFO are used to cross clock domains. CSL fifo can specify either a synchronous (one clock is used to clock the FIFO) or an asynchronous FIFO (2 clocks are used to clock the FIFO : one for the read domain and one for the write domain). The FIFO can be configured to absorb “upstream” pipeline data which is in flight after a stall is asserted from the write clock domain and is synchronized to the read clock domain.

```
csl_fifo <fifo_name>;
size(<#>); //# of entries in fifo
clk(<clk_name>); //synchronous clock name
rd(<clk_name>); //synchronous read clock name
wr(<clk_name>); //synchronous write clock name
almost_full(<clk_name>); //synchronous almost full clock name
almost_empty(<clk_name>); //synchronous almost empty clock name
full(<clk_name>); //synchronous full clock name
empty(<clk_name>); //synchronous empty clock name
```

1.4 CSL FIFO Checker

1.5 CSL FIFO Reports

1.5.1 Fifo status regs

programmable status generation registers for FIFO flow control
These registers count the number of writes to

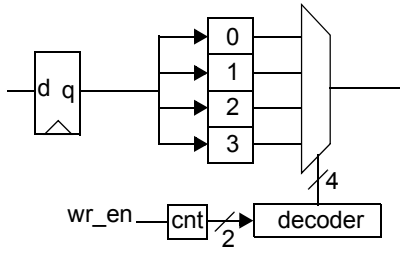
- almost empty register = when the counter equals the almost empty value the almost_empty output signal is enabled
- almost full register = when the counter equals the almost full value the almost_full output signal is enabled
- empty register - when the counter equals the empty value then the the empty output signal is asserted
- full register - when the counter equals the full value then the the full output signal is asserted

FIFO queue size shall be parameterizable.
The system will be tested with different FIFO sizes to emulate flow control issues.

1.6 CSL FIFO Miscellaneous

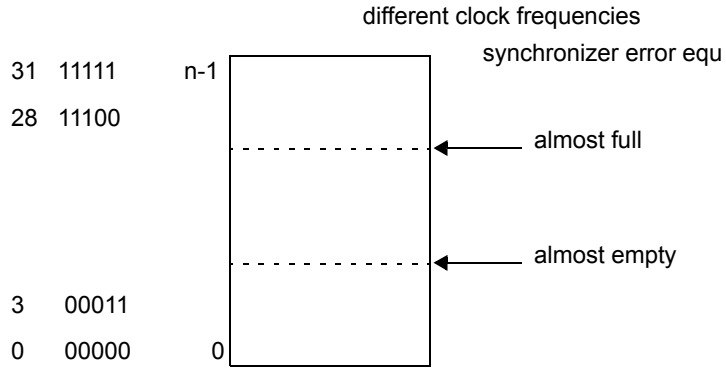
The read and write addresses are separated by a constant. The FIFO can be written and read every clock cycle. There is an <n> cycle delay across the FIFO depending on the clock frequency ratios and the phase relationship between the clocks. Difference between write and read counters (accounts for delay) need to guarantee min/max phase difference.
1, 2, 3 cycle delay across out of phase clocks.

FIGURE 1.22



The synchronizer in the figure above has 4 registers. Each register can be written by the input data line. Each register can be selected by a mux. The mux select line is driven by a counter. The counter is incremented by rd_en signal. The write enable is a 4-bit signal that is mutex which is the output of a decoder. The decoder is driven by a two bit counter. The two bit counter is incremented by a wr_en signal. The distance between the read and the write counters is set programmatically after reset. The distance between the two counters is a function of the phase difference between the two clock domains. The delay between writing an element to an empty FIFO and reading the element out of the FIFO is due to the synchronization delays between the read and write pointers which are used to update the full an empty signals and the high and almost emptys.

FIGURE 1.23



1.6.1 Automatically test a FIFO

1. create a testbench and instantiate the FIFO

2. load the FIFO with a sequence of numbers (0,1,2,3,4...). Use a counter to create the numbers

3. pop the FIFO and check the sequence for skipped numbers or duplicate numbers

4. vary the relationship of the clocks

5. vary the order of push and pop operations

6. test burst push and pop

7. test single push and pop

8. test single push and burst pop

9. test burst push and single pop

//DEREK SAID TO ME TO MOVE THIS SECTION ,ORIGINATING FROM CCSL_REGISTER HERE
//MC SAT SEPT 2,6.48 PM

FIGURE 1.24 !!move this figure to event detector section!

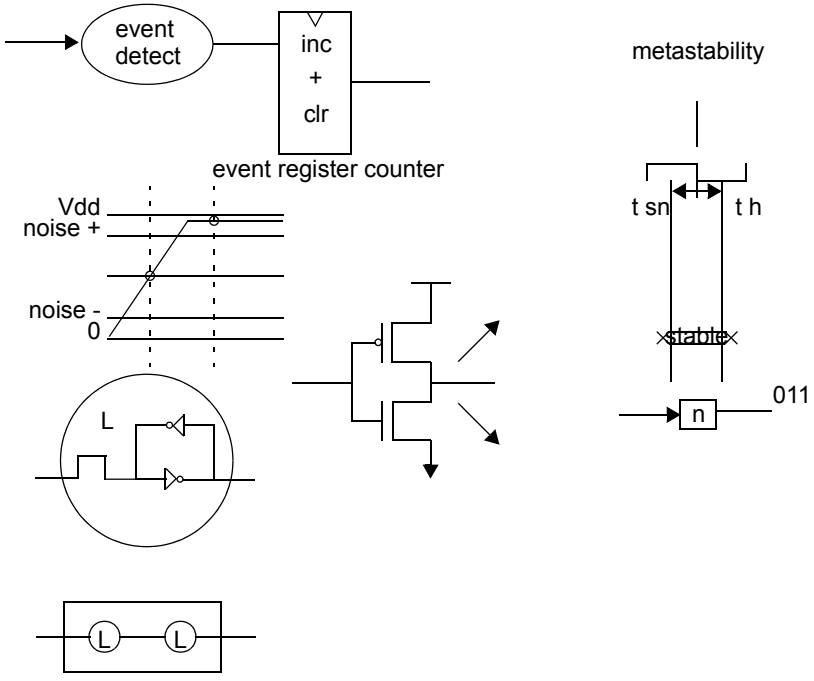
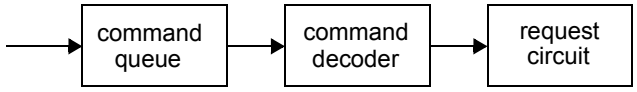


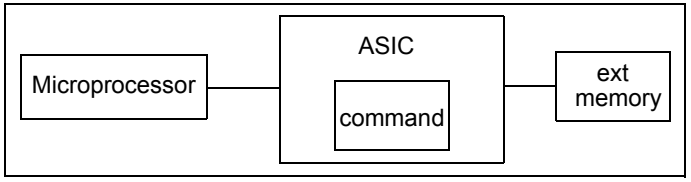
FIGURE 1.25 !!move this figure to event trigger section

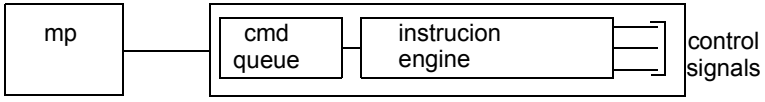


The idea behind the blocking semaphore is to minimize the latency to execute a sequence of commands. The host ASD interaction is minimized because the control of the sequencing is transferred from the host to the ASD. The host does not have to poll the ASD to find out when an event occured and then send a new command to the ASD. Instead the host sends the commands to the ASD. The ASD then executes the requests and waits for the request to complete before executing the next command(wait).
Select events based on

- rising/falling edge
- low/high level

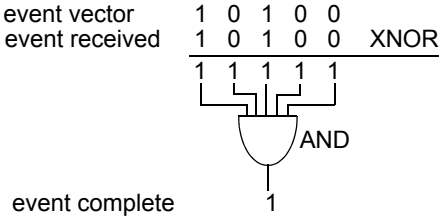
FIGURE 1.26





The microprocessor wants to read/write a sequence of words in the ASIC internal or external memory.
Up to 4 timeout bits can be set.
Wait for events to occur.
no overloading

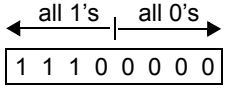
FIGURE 1.27



Ordered event tracking

A → B → C
envar (!A&B) | (!A&C) | (!B&C)

FIGURE 1.28 event enable register



8 possible events
3 events selected
Now interleave memory read/writes with events by spacing out events with 0's.

FIGURE 1.29

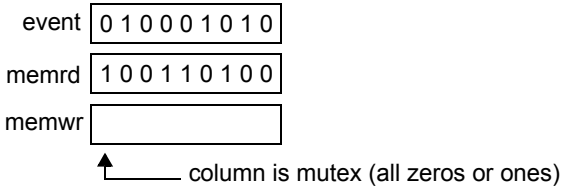
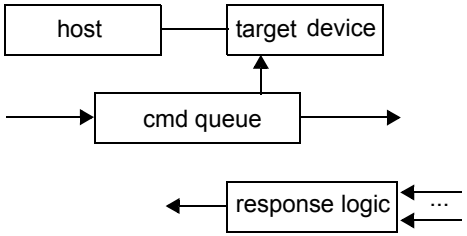


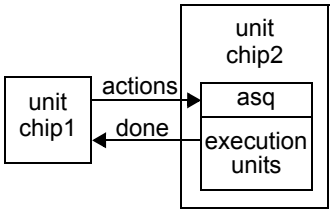
FIGURE 1.30 Mechanism for host software control



Commands are streamed from the host to the target device command queue.
Commands are issued from the command queue.
When responses are received the next command is executed.
</move to the new chapter Host Interface>

Mechanisms to reduce the bandwidth between the host processor and the application specific device (ASD).
<this was added afterwards>
AS queue
!!must figure out a better looking way for this
write actions to a queue
pop the queue
execute the action

FIGURE 1.31 Action sequence register



chip1 sends actions to chip2 in a batch, chip2 executes the actions, chip 2 sens a ‘done’ signal to chip1
</end of add>
The host sends a stream of request and wait commands to the Command Request.
Event completion
The ASD block queues up the stream of commands.
The ASD block has circuits which handle the commands. The ASD contains event detection circuits which assert when specific events occur and record the event.