CHAPTER 7 CSL Register

TABLE 7.1 Chapter Outline

7.1 Definitions

TABLE 7.2 Definitions

CSLC	CSL compiler
register write architecture	registers can be written and read with data or control values. The control registers detect when they are written and cause some action in the unit or chip to occur
shadow register	a shadow register is a register which contains a copy of the values in another register in the same simulator or a different simulator
ASD	Application Specific Device

7.2 CSL Register Overview

What's a register? Where do you use it?
Register specifications evolve over the lifetime of a chip project. We will describe the register memory element specification development steps in the following paragraphs.

A designer can write a memory element specification for each new chip design. The memory element specification includes the following:

- name of the register,
- the width of the register

- the cell declarations
- the address of the register (either relative to the previous memory element or an absolute address)
- attribute bits
- control pins

Cell declarations:

- name of the cell
- attribute bits for the cell
- width of the cell What's this? A register cell has only one bit. It is a memory cell?
- optional enum

The memory element specfication is then checked for correctness.

The implementation of the memory mapped structure can be a register file, separate flip flops, registers, FIFO's, SRAM's, and/or any combination of the above. The program can declare registers, assign addresses to the registers, and create the state element code in the target .
The cslc can create signals with addresses in different output languages and the associated structures (e.g. register). The CSL Register specifications include the following:

- clock name
- reset signal name and reset value
- set signal name and set value
- enable signal name

CSL register specifications are compiled by the CSLC to create registers in the target language (i.e. Verlog, C++, etc.). The CSL register specfication is used to create memory mapped state elements that are addressable by other hardware units or by software through writes or reads.
Fields within register files are called cells. Cells can be named. Registers with special cells can be declared and added to a composite object such as a register file.
Cells can be associated with an ennumerated type.

7.3 CSL Register Concepts

7.3.0.1 Register declaration

A register is declared with the name of the register and the bit range. A read/write permission is specified for each cell in the reg. The address of the reg is calculated either by incrementing the previous address or is explicitly specified using a number or a constant. Note that where numbers are allowed constants can be used instead. The width of the cells cannot exceed the width of the register. The cells are optional. Each cell may define its own rws(**read/write/shadow**) bits which override the register's rws bits. If the cell does not define the rws bits then the cell uses the register's rws bits.

TABLE 7.3 Cells, registers and tables

cell	bitrange or part of a register or a cell
register	bitrange or cells and/or bitranges
table	rows, columns, registers, cells, tables

Registers can contain cells. The cells can have a bit range. The cells can have defined values. The defined values can be specified with an enumerated type. The cells can be ordered in the register through a concatenation operation, some like this: {[6:4],[1]}.

7.3.1 Register state types

The following is a list of CSL register state types:

TABLE 7.4 State types

Elment	STATE_TYPE
latch	LA
flip flop	FF
SRAM	SRAM
queue	circular FIFO

7.3.1.1 Register pins

The input and output signal names are based on the register name by default and can be overridden.
The register pins are configured using the pin names listed in Table 7.5.

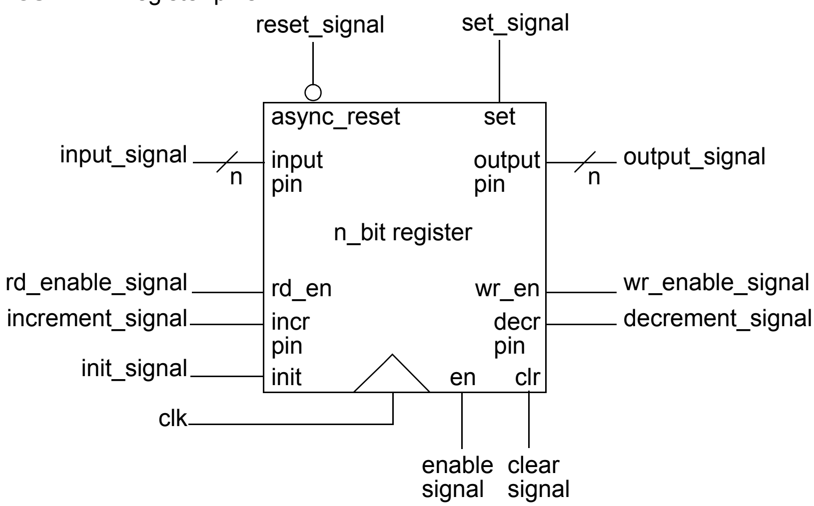
TABLE 7.5 Register pins

No	Pin Name	Description	Connected to
0	clk	clock signal	clock tree
1	reset	signal which resets the register to *reset_value*	reset tree
2	set	signal which sets the register to *set_value*	logic control signal
3	clr	signal which sets the register to *clear_value*	logic control signal
4	en	enable signal	logic control signal
5	in	data input	data signal
6	out	data output	data signal
7	inc	increment control	control signal

TABLE 7.5 Register pins

No	Pin Name	Description	Connected to
8	dec	decrement control	control signal
9	init	initialize the register	control signal
10	rd_en	read enable	control signal
11	wr_en	write enable	control signal

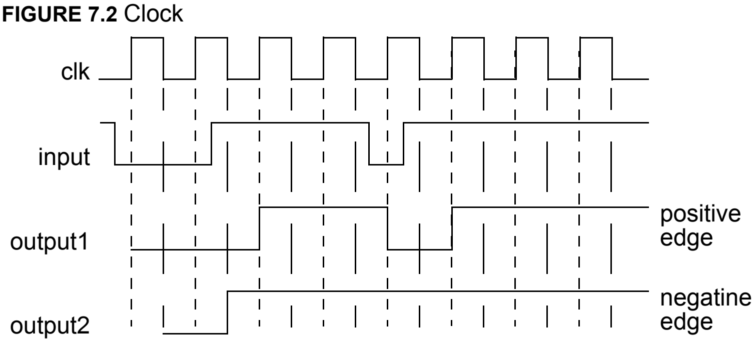
FIGURE 7.1 Register pins



7.3.1.1.1 Clock

The pin that connects the clock signal that can be from a clock generator with the register. All transitions can occur when the valid clock transition takes place.

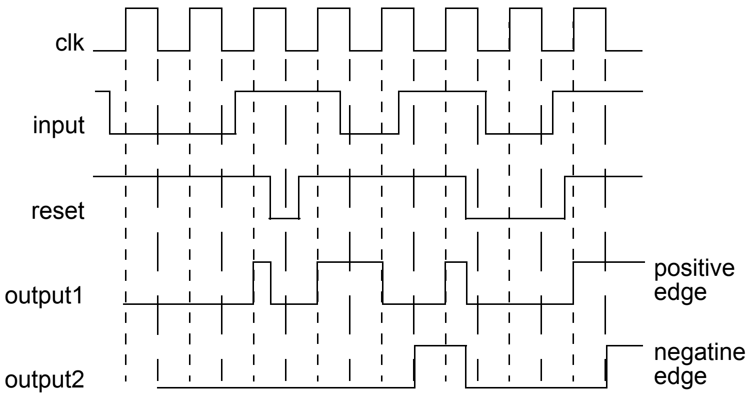
FIGURE 7.2 Clock



7.3.1.1.2 Reset

The pin that connect the asynchronous reset signal with register. The reset signal will reset the register when is valid(on 0 level) and the register don't wait for clock transition.
The asyncronous reset signal has the highest priority.

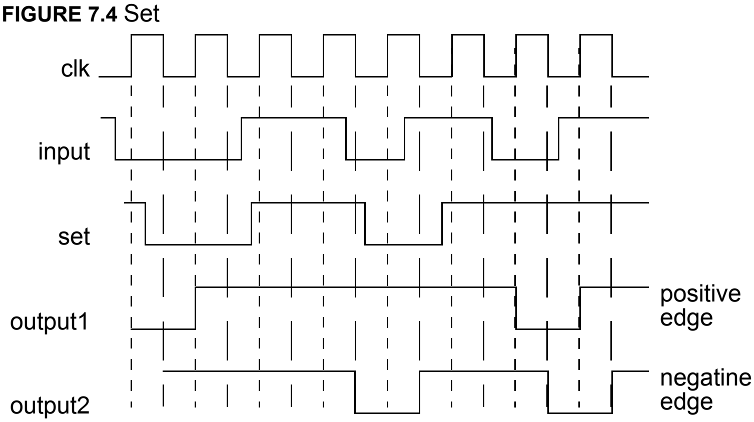
FIGURE 7.3 Reset



7.3.1.1.3 Set

Is the pin where the synchronous set signal is connected with register. When set_signal is valid (on 0 level) the register is loaded with the set value. The default set value is 1.
Set signal has a higher priority than data signal.

FIGURE 7.4 Set



7.3.1.1.4 Clear

This pin connects the clear signal to the register. The register will be loaded with the clear value when the clear signal is valid

7.3.1.1.5 Enable

Connect the enable signal to the register. When the enable signal is valid the register can be read/ wite.

7.3.1.1.6 Increment

This pin connect the signal that will increment the value to a counter register.

TABLE 7.6 A register counter with 2 bits

Increment	Output value
0	0
1	1
1	2
1	3
0	3
1	0
1	1

7.3.1.1.7 Decrement

This pin connect the signal that will decrement the value to a counter register.

TABLE 7.7 A register counter with 2 bits

Decrement	Output value
0	0
1	3
1	2
1	1
0	1
1	0
1	3

7.3.1.1.8 Initialzation

This pin connect the init signal to the register. The register will be initialized with the specified value.

7.3.1.1.9 Read enable

Connect the read enable signal to register. When this signal is valid the output value can be read from the output pin.

7.3.1.1.10 Write enable

Connect the write enable signal to register. When this signal is valid the input value can be write.

7.3.1.1.11 Input and output pins

Registers can have paralel input, paralel output, serial input and serial output.
Ce se intampla cu tabelele astea?

TABLE 7.8 Valid combinations

I/O pin							
serial input		X		X	X	X	
serial output			X	X	X		X
paralel input	X	X	X	X			X
paralel output	X	X	X	X		X	

7.3.2 Register types

Register types are mutually exclusive. Only one type may be set for a register. The register types are listed in Table 7.9.

TABLE 7.9 Register types

Mnemonic	flip flop type	description
	Action Sequence fifo	action commands are loaded into a fifo;a go signal is sent and the commands are executed in sequence; when all commands are executed a done signal is asserted
ATOM	***AtomicRegister***	***hardware semaphore bit plus a data register***
CNT	CounterRegister	up/down counter register
CTL	Control register	used to control other hardware elements
DFF	D flip-flop Register	register with D flip-flops
EVNT	Event Register	captures events
INT	Interrupt Register	intrerrupt mask, interrupt enable, and interrupt registers

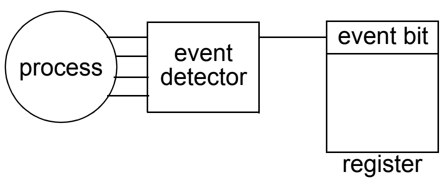
TABLE 7.9 Register types

Mnemonic	flip flop type	description
LFSR	Linear Feedback Shift Register	a shift register whose input bit is a linear function of its previous state
SFT	Shift Segister	register that shift the bits
SEMA	***Semaphore Register***	***hardware semaphore register***
STATISTIC	Statistics Register	capture statistics based on events
STATUS	Status Register	records status

7.3.2.1 Action Sequence Register

The Action Sequence Register specifies the sequence of expected actions. An action is a memory rd/wr or an event.
An action has to complete before the next action can occur. Memory reads/writes which are back to back can occur without waiting but must complete before the next event.
!!need to figure out a better looking way for the following content
wait for read address 10
wait for read address 20
send event edge detectors
Trebuie o reprezentare grafica
Event detectors
Figure 7.5 shows a process which generates events. An event detector captures specific events using either level sensitive or edge triggered logic and could then set an event bit in a register and/or perform any other specific tasks. Event detectors are used for interrupts

FIGURE 7.5 Event detector in use



Action type:

- level sensitive: 0 or 1
- edge sensitive: 0 to 1 transition or 1 to 0 transition

7.3.2.2 Atomic registers -> Move to the next release

Registers can be protected with semaphore bits. When the semaphore bit is set the register cannot be modified by memory address writes. The semaphore bits can be set by either hardware or software. The atomic register can only be written by software or hardware if the software routine address or the hardware unit is first able to set the atomic register lock bit. The sequence of operations to control the atomic register is as follows:

1.test and set the lock bit

2.read modify write (rmw) the atomic register

3.clear the lock bit

An atomic register is writable by the owner processor and readable by each neighbour of the owner. The lock bit is also used when the owner processor writes and in this case the lock bit is set to avoid the access of the rest of proccessors to read the data.
The atomic register has an address for lock bit and and another address for data fields.

FIGURE 7.6 Atomic register

lock bit	atomic register

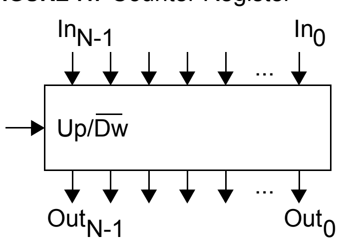
Software must use a spinlock[1] to acces the register and hardware must implement acces block to access the atomic register. Note that the atomic register access can be a local unit or remote unit access.

7.3.2.3 Counter Registrer

A binary counter register has an input to select the counting direction U/_D (U/_D=1 -> count UP, U/_D=0 -> count DOWN), a n-bit input for the number witch is loaded into register (start value) and a n-bit output. Counter Register needs a count signal and a direction control signal. The register type counter has optional args. The counter defaults to a start value of 1, and up direction counter.

NOTE: It is illegal to use counter add logic options without setting the register type to counter.

FIGURE 7.7 Counter Register

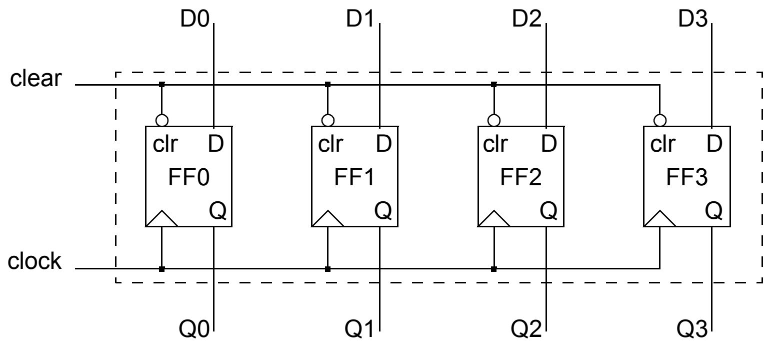


7.3.2.4 DFF register

This register contains n D-type flip-flops that are connected to the same clock signal.

1.In software engineering, a spinlock is used to put a thread into a wait loop.

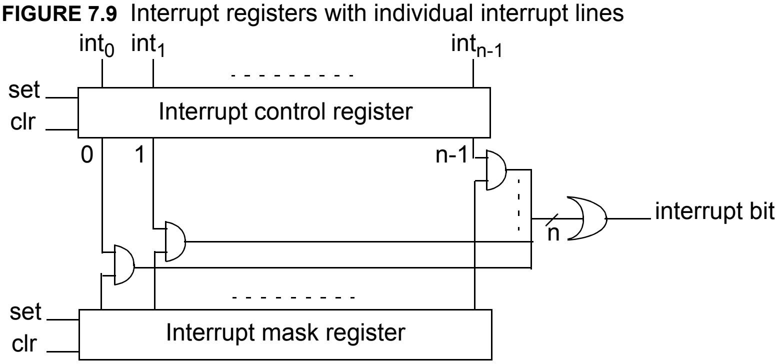
FIGURE 7.8 A DFF Register with 4 D-type flip-flops



7.3.2.5 Interrupt registers

The interrupt register is updated by events in the chip. The event detectors can be level or edge sensitive. The interrupt mask register is used to enable the bits driven by the interrupt register. Interrupt register can be cleared by hardware, software or both. A mask can be used to select parts of the register which are ORed together to form a single interrupt bit. When the interrupt bit is asserted a different unit is notifieded that there are one or more pending interrupts. The interrupt mask is used to queries the interrupt register and obtains the interrupt register's value. Another unit will resolve the intrerrupt.

FIGURE 7.9 Interrupt registers with individual interrupt lines



Set or clr signals can be created for individual bits or for the entire register

7.3.2.6 Linear Feedback Shift Register

LFSR is a shift register whose input bit is driven by the exclusive-or (xor) of some bits of the overall shift register value. Because the operation of the register is deterministic, the sequence of values produced by the register is completely determined by its current (or previous) state. The register has a finite number of possible states. A LFSR with a well-chosen feedback function can produce a sequence of bits which appears random and which has a very long cycle. The initial value of the

LFSR is called the seed and can't be 0 (all bits in 0) in which case it will never change.
The outputs that influence the input are called taps. The tap sequence of an LFSR can be represented as a polynomial mod 2. This is called the feedback polynomial or characteristic polynomial.

- If (and only if) this polynomial is a primitive, then the LFSR is maximal
- The LFSR will only be maximal if the number of taps is even
- The tap values in a maximal LFSR will be relatively prime
- There can be more than one maximal tap sequence for a given LFSR length

FIGURE 7.10 A LFSR with 3 bits width

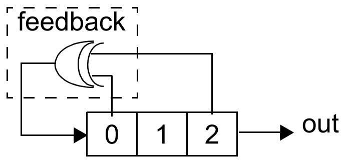


TABLE 7.10 The states for the cells from the LFSR with 3 bits

Input	Cell 0	Cell 1	Cell 2	State
1	1	0	0	0
1	1	1	0	1
0	1	1	1	2
1	0	1	1	3
0	1	0	1	4
0	0	1	0	5
1	0	0	1	6
1	1	0	0	0

7.3.2.7 Shift Register

Serial-in, serial-out shift registers delay data by one clock time for each stage. They will store a bit of data for each register.
At a logic shifter the vacated bits are 0-filled and for a arithmetic shifter fill by a copy of the sign bit.

FIGURE 7.11 A left shift register

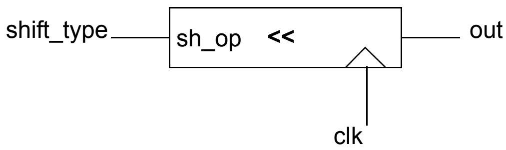


TABLE 7.11 Right Logic Shift Register and Right Arithmetic Shift Register

Cycle	Logic	Arithmetic
0	10110001	10110001
1	01011000	11011000
2	00101100	11101100
3	00010110	11110110
4	00001011	11111011
5	00000101	11111101
6	00000010	11111110

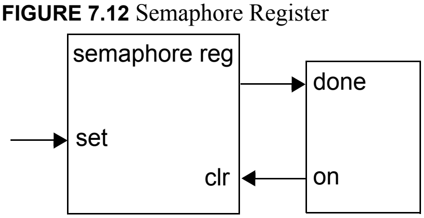
TABLE 7.12 Shift type

Shift type	Description
sal	Shift Arithmetic Left
sar	Shift Arithmetic Right
shl	Shift Logical Left
shr	Shift Logical Right
rol	Rotate Left
ror	Rotate Right

7.3.2.8 Semaphore register -> Move to the next release

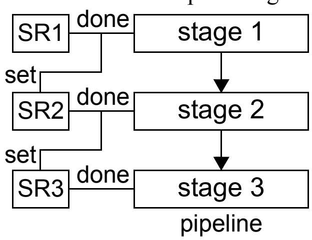
We will now describe a semaphore register operation. This register can only be written when certain conditions are met. This is true during certain time intervals or when certain equations are true or false.
When the register is set, then some operation has to complete before the semaphore register is cleared at the completion of the operation. Hardware or software can set the register.

FIGURE 7.12 Semaphore Register



One usage of semaphore registers can be seen in the pipeline scenario from Figure 7.13. The semaphore register on top (SR1) is cleared when the pipestage finishes to perform the associated operations by the 'done' signal. The same signal sets the next semaphore register (SR2) which corresponds to the following pipestage in the pipeline. A similar path is followed for the last pipestage and its associated semaphore register (SR3).

FIGURE 7.13 Semaphore registers used in a pipeline



There are situations where software can change a hardware register when they are not supposed to.
The hardware can prevent the software from updating registers at the wrong time.
If chips have registers which are not supposed to be updated during certain operations or time intervals then the register enable can be conditioned with a flag that prevents the register from being updated. The write to the register can be queued (stored in a flip flop) and released to write to the register when the operation is terminated.

Some parallel systems have a synchronization barrier. A sycnhronization barrier is a point which says that all processes/processors have stopped (think fork and join). When all processors/processes reach the synchronization barrier then the out of band signals can poll the hardware or change the state of the machine. The state of the machine cannot be changed during the normal operating mode. Hardware enforces this rule. This is similar to a semaphore but it applies to entire blocks or entire chips.

FIGURE 7.14 Semaphore register operation

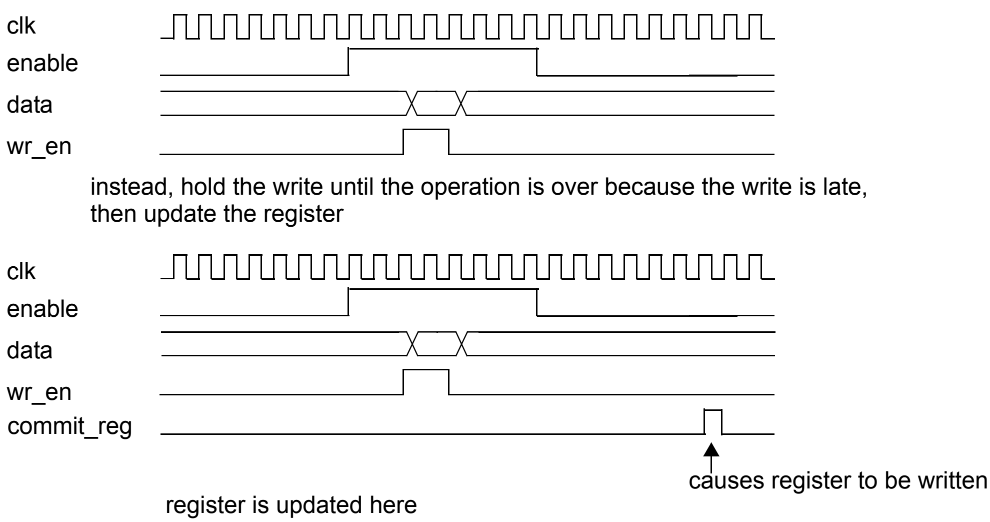


TABLE 7.13 Semaphore

Request	transition (s)
Wait	wait for event
Continue	after

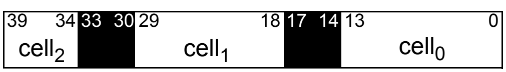
7.3.2.9 Status Register

A register in most CPUs which stores additional information about the results of machine instructions, e.g. comparisons. It usually consists of several independent flags such as carry, overflow and zero that indicates the status of various mathematical operations. The Status Register is chiefly used to determine the outcome of conditional branch instructions or other forms of conditional execution. is a collection of flag bits for a processor . These flags are commonly used during conditional testing and program branching.

7.3.3 Register cells

A cell (1 dimensional object) is a named bit range inside of a state element where the state element can be a flip flop, latch, register, register file or SRAM word. If a register contains cells some bits in the register can be unused. Cells can be used in any addressable memory element or any state element.Cells have names, and cells are used to assign a symbolic name to a bitrange. The bitrange can then be accesed using this symbolic name.
Registers can contain cells. A cell is a sub-range within a register. Cells can be read individually or part of the entire register. The csl register command can create individual cells in each register. Cells are named subranges in a register.

FIGURE 7.15 Cells are named subranges in a register



Cells are used to create memory words. Memory words can be grouped together to create table entries. For example a 64-bit memory word has the following cells {Cell F_{12}, Cell E_{10}, Cell D_{10}, Cell C_{16}, Cell B_8, Cell A_8}. The 64-bit word is divided into the upper and the lower 32-bit words.

There are several ways to create registers and adjacent cells (packed cells).

7.3.3.1 Cell names

In a memory map each register can contain cells.The default for the cslc is to allow different registers to contain cells with same name.We can specify that each cell in each register has to be unique by using the :

```
scope_name.unique_cell_names(true);
```

This specifies that cell names have to be unique.This can be applied to any in the entire scope (scope can be a CSL unit, an instance, a memory map name or a namespace), so every cell name in every register has to be unique. No duplicate cell names in different registers are allowed.

```
scope_name.unique_cell_names(false);
```

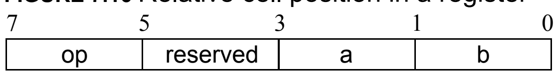
specifies that the cell names are not required to be unique.

If no cell is given then the default cell name is the name of the register. A default name can be defined by setting the default cell name equal to a string:

```
scope_name.default_cell_name(string);
This cannot be used with unique cell names.
```

7.3.3.2 Relative cell position in a register

FIGURE 7.16 Relative cell position in a register

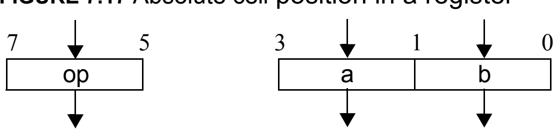


Relative cells information

```
csl_reg reg_name;
//creates a cell with the width of 2
csl_cell op(2);
csl_cell reserved(2);
csl_cell a(2);
csl_cell b(1);
reg_name.add_cell(op);
//added reserved cell to the right of op cell
reg_name.add_cell(reserved,RIGHT);
reg_name.add_cell(a,RIGHT);
reg_name.add_cell(b,RIGHT);
```

7.3.3.3 Absolute cell position in a register

FIGURE 7.17 Absolute cell position in a register



3 separate inputs and 3 separate outputs

```
//declares csl_reg
csl_reg reg_name;
// declares csl_cell with the range (7:5)
csl_cell op(7,5);
//adds the cell op to the register reg_name
```

```
reg_name.add_cell(op);

csl_cell a(3,1);
reg_name.add_cell(a);
csl_cell b(1,0);
reg_name.add_cell(b);
```

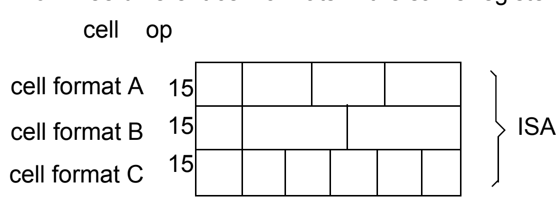
7.3.3.4 Creating Cells in registers

Registers can be constructed out of cells. Cells are effectively register fields. The same cell can be used in many different registers. A cell can have a different name (cellname) in different registers. Cells are specified using any bit range:

```
csl_reg reg_name;
csl_cell a1(15,10);
csl_cell a2(10,0);
reg_name.set_cells(csl_list(a1,a2));
csl_cell a3(15,9);
csl_cell a4(9,7);
csl_cell a5(7,0);
reg_name.set_cells(csl_list(a3,a4,a5));
```

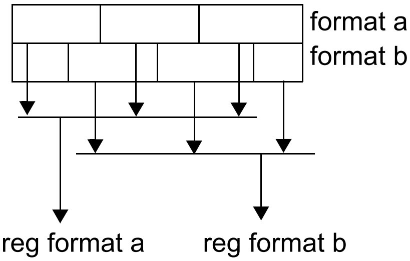
Cells may overlap if a register has more than one cell layout.
A format is a combination of cells that make up a register. A cell format is a group of cells

FIGURE 7.18 Three different cell formats in the same register



The three formats have the same cell (cell op) in the first position

FIGURE 7.19 two different formats in the same register



7.3.3.5 Register bitrange

Represent the width between MSB(Most Significant Bit) and LSB(Least Significant Bit). The bit range may not be contiguous if there are unused bits , a concatentation of ranges is allowed (i.e: {[6:4],[1]}).

7.3.3.6 Register write architectures

Some hardware architectures use a register write architecture where writes to registers and elements of registers trigger actions in the hardware. The CSLC will support the generation of event signals which will trigger these actions. Everytime a register is written an event occurs (something happens).

7.3.3.7 Register attributes

The CSL register declaration specifies the software read(r), write(w) and shadow(s) - rws attributes on a register and/or cell basis.

TABLE 7.14 Register attributes

Attribute	Operation
read (r) only	the register can be only read
write (w) only	the register can be only write
shadow(s)	the register can be shadow

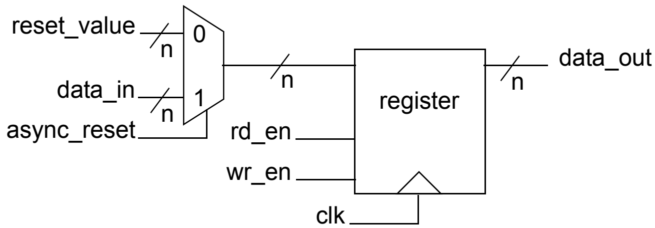
7.3.3.8 Register values

Registers can be set to either constant or the values driven by signals on the assertion of reset, init or clear:

7.3.3.8.1 Reset

After a reset operation the memory element is set to this value.Default is zero.

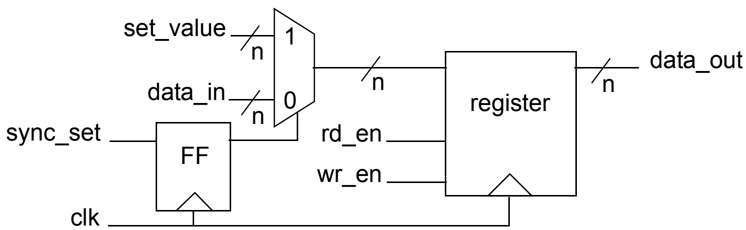
FIGURE 7.20 Register with asynchronous reset signal



7.3.3.8.2 Set

After a set operation the memory element is set to this value. Default is one.

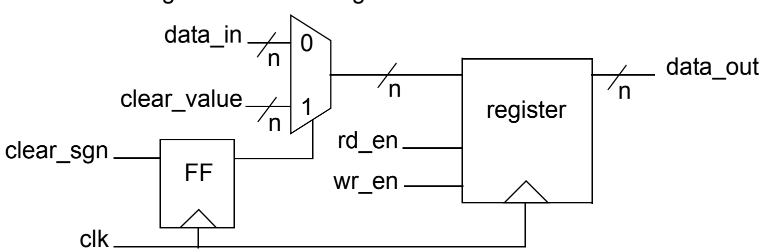
FIGURE 7.21 Register with synchronous set signal



7.3.3.8.3 Clear

After a clear operation the memory element is set to this value.

FIGURE 7.22 Register with clear signal



7.3.3.8.4 Initialzation

When the init signal is active the memory element is initalized with this value.

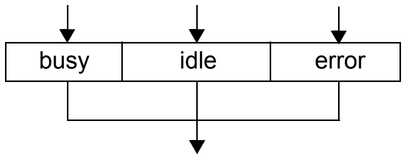
7.3.3.9 Register read/write operations

Registers which have bit access or irregular cell accesses (e.g. all registers in register file not byte aligned) are not inserted into contiguous physical structures such as SRAM's. Instead registers with irregular cells are separate structures which are accessed using muxes.

- individual bits can be read or written
- individual cells can be read or written
- the entire register can be read or written

The three bits busy, idle and error in the register shown in Figure 7.23 are updated independently by 3 different signals.

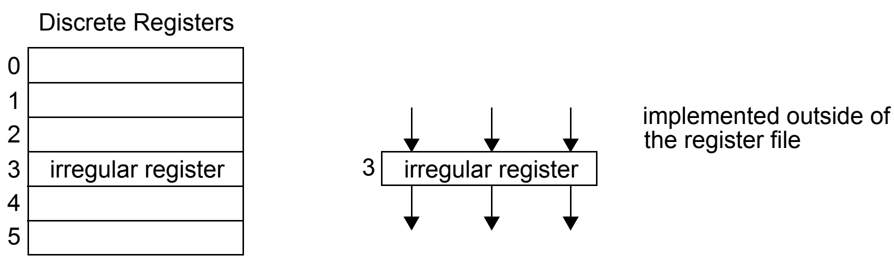
FIGURE 7.23 Bits in a register



7.3.3.9.1 Irregular Access Registers

If the register address falls in the middle of a contiguous address range associated with an aggregate memory element then the register is implemented separately from the contiguous address range and the corresponding element in the physical structure is unused.

FIGURE 7.24 Registers with bit irregular access in the middle of a contigous address range

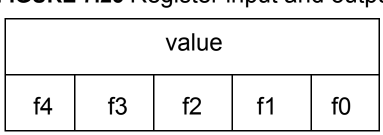


7.3.3.9.2 Writing and reading cells/registers

- input fields can be written separately or together
- output fields can be read separately or together
- an entire register can be written/read

Enable fields or the entire register can have write/read enables.

FIGURE 7.25 Register input and output fields



input registers 1 bit each (input fiels), are read as a whole range of 5 bits in the output field (abstract)

7.3.3.10 Register prefix

Specify a prefix name for all registers in a CSL specification file which will be prepended to each register name.

7.3.4 CSL Tables

Tables are two dimensional objects which store state. Tables can be declared as individual objects and glued together or the entire table can be declared in one operation. Tables can be specified in two different ways:

- the width of the table and the the number of rows in the table can be specified
- the user can specify column and row objects and add them individually or in groups to the table
- the user can create row objects
- the user can create col objects

Tables are implemented as memories in the actual design. Tables have a hierarchy.
The Table's hierarchical structure is based on the elements based on the hierarchical structure

Create the individual columns which are used to construct the table. Individual elements in the hierarchy can be accessed using hids.
Note that accesses of two or more rows simultaneously is more expensive in terms of the hardware implementation.

FIGURE 7.26 Create the individual columns

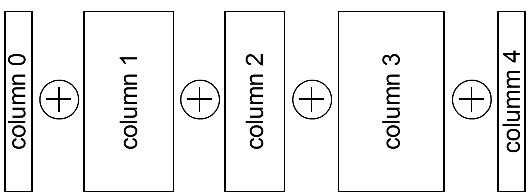
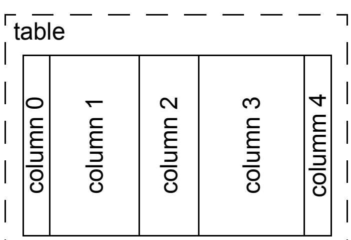


FIGURE 7.27 Concatenate the columns together to create the table



Create the row objects and concatenate then togever.

FIGURE 7.28 Create the individual rows

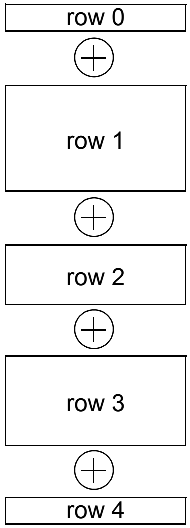
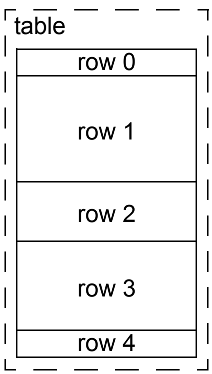
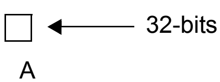


FIGURE 7.29 Concatenate the rows together to create the table



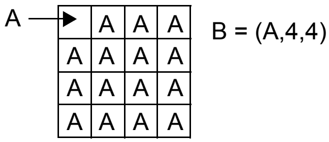
Create table out of three smaller tables

FIGURE 7.30 Create one cell



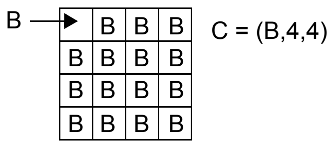
Create a 4X4 matrix of A's

FIGURE 7.31 Create a table of A cells



Create a 4X4 matrix of B's

FIGURE 7.32 Create a table of B cells



Create a 40X50 matrix of C's

FIGURE 7.33 Create a table of C cells

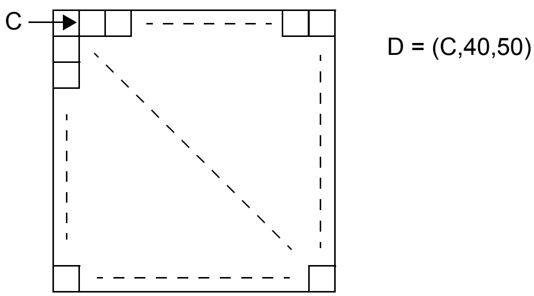


FIGURE 7.34 Using cells to create memory words

	31		0
DW0	cell F_{12}	cell E_{10}	cell D_{10}
DW1	cell C_{16}	cell B_8	cell A_8

```
csl_cell f (12);
csl_cell e (10);
csl_cell d (10);
csl_cell c (16);
csl_cell b (8);
```

```
csl_cell a (8);
csl_cell dw0;
csl_cell dw1;
dw0.set(f,e,d);
dw1.set(c,b,a);
csl_table t;
t.row(dw0);
t.row(dw1);
csl_row r0;
// set the contents
r0.set(dw0);
csl_row r1;
r1.set(dw1);
//row | concat(rows)
t.row(r0);
t.row(r1);
t.rows(ro,r1);
csl_column c;
c.columns(dw0, dw1);
t.column(c);
csl_column c0;
csl_column c1;
c0.column(dw0);
c1.column(dw1);
t.columns(c0, c1);
```

Cells can then be used to create tables. The table can be prefixed with type or can generate memory map addresses.
Example:

```
csl_define TABLEMAX 16;
csl_define WORD_WIDTH 32;
csl_table constant_table;
constant_table.word_type(WORD_WIDTH);
constant_table.num_words(TABLEMAX);
```

The code section above creates a 16 entry table and the word width is 32 bits
//adapt following text and then remove -> The name of this table is *constant_table*. The entry type is const32, which means that the table contains 32 bit words.

7.3.4.1 CSL Row

Create a csl row that can be added to a table.

7.3.4.2 CSL Column

Create a csl column that can be added to a table.

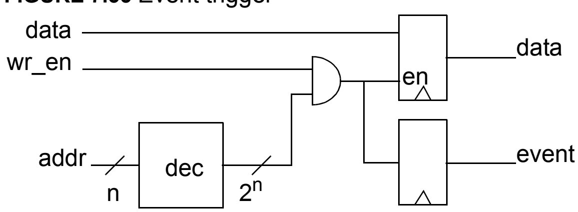
7.3.4.3 Event register

A write and/or read to a memory location can trigger an event on an event line. The event can be pipelined.

TABLE 7.15 Events

Event Types	Description
edge triggered	rising/falling
level sensitive	high/low

FIGURE 7.35 Event trigger



Examples :
The event can be a valid bit or a single pulse to start or stop a process or a write to a particular address range in a memory or a write to a reg or a write in a register file.

FIGURE 7.36 Complete an event

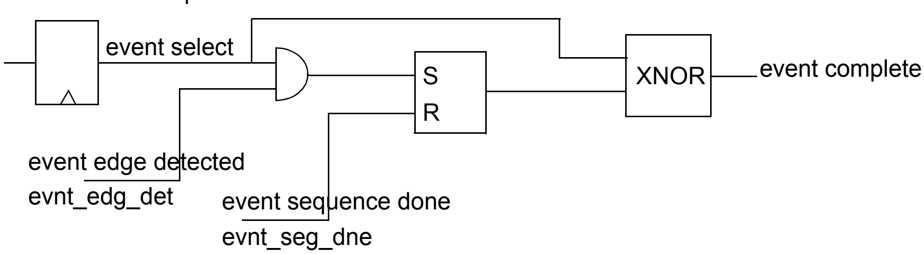
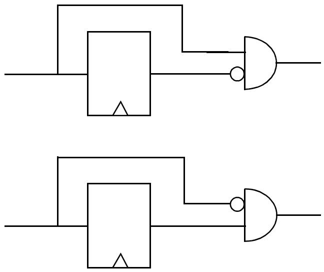


FIGURE 7.37 Edge detectors

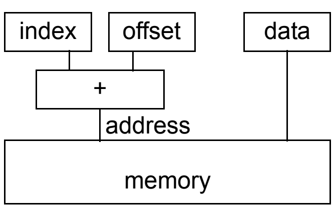


7.3.5 Memory map

7.3.5.1 Index/data pairs used to indirectly address memories

The index/data pair memory is used by first writing the index/data pair register write architectures to indirectly address a memory. A write operation to a memory location is done by first writing the register which contains the index/offset into the memory and then writing the data to the data register. The write to the data register triggers an event the next cycle which generates a memory write enable. A read to the memory is triggered by a read operation to the index/offset register. The read operation writes the index/offset register. The next cycle an event is triggered which generates a memory read enable. The memory is not in the address map of the machine. This allows the memory to be relocate in the software view in the sense that the memory is only addressed indirectly. Checkpointing and restoring the contents of the memory can either be done using a software routine or a built in engine which will sequence through the memory space and output the contents of the memory to the host initiating the checkpoint/restore operation..

FIGURE 7.38 Index/data pair used in memory addressing



7.3.5.2 Hardware register exclusion

Registers may be excluded from the generated memory mapped structure if the **noaddr** keyword is used in the register declaration.

7.3.5.3 Register addressing

Addresses are asigned to registers. See the CSL memory map section for instructions on how to assign an address to a cell, register or table.

7.3.6 Document generator

Ce e asta?

7.3.7 Code generator

Code/Documentation Generator
Now Verilog, VHDL, and C++ code can be generated.
Documentation describing the memory element set can be generated.
Next the relationship between the memory elements that are written is described

7.3.7.1 Individual register classes in C++ and verilog

The individual registers can be represented in C++ code. The C++ code will implement the following functions:

- real and shadow variables will be present in the private variable section
- cells will be stored in variables in the private variable section (using bit cell insertion/ extraction is far too heavy weight and the savings in terms of memory space in a C++ program are incidental compared to the run time cost of bit insertion and extraction using shift and mask).
- write/read entire register value (with optional mask to protect or override bits in register)
- assign an absolute and relative address to the register

csl_cell declaration:

- name only :
```
csl_cell bar;
```
- name with width :
```
csl_cell bar (10); //width of the cell is 10
```
- name with bit range :
```
csl_cell bar (bitrange); // the absolute indices of the cell are specified
```

7.3.8 Diagnostic Test generator

Test Generator
Tests to write/read/compare the memory elements which are generated. A diagnostic test writes a value to a register, reads a value from the register and compares the read value against the write value.

```
write_mem_elem(<addr>, <value>);
ret_val = read_mem_elem(<addr>);
if (<value> == ret_val)
else <print error message>
```

7.3.9 Register ports and logic

When creating a register, 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 7.16 Ports for a simple register

Port Name	Dir	W	Generated by	Description
clock	i	1	Automatically	Port for clock signal
reset	i	1	Automatically	Port for reset signal
set	i	1	Automatically	Port for set signal
clear	i	1	Automatically	Port for clear signal
init	i	1	Automatically	Port for init signal
enable	i	1	Automatically	Port for enable signal
parallel_input	i	ud	Automatically	Port for paralel input sigal
parallel_output	o	ud	Automatically	Port for paralel output signal
neg_output	o	ud	add_logic(neg_output);	Port for negative output signal
serial_input	i	1	add_logic(serial_input);	Port for serial input signal
serial_output	o	1	add_logic(serial_output);	Port for serial output signal
rd_en	i	1	add_logic(rd_en);	Port for read enable signal
wr_en	i	1	add_logic(wr_en);	Port for write enable signal

TABLE 7.16 Ports for a simple register

Port Name	Dir	W	Generated by	Description
field_name_input	i	?	add_logic(connect_input_to_field, field_name);	Port for connection with an input field
field_name_output	o	?	add_logic(connect_output_to_field, field_name);	Port for connection with an output field

TABLE 7.17 Extra ports for an action register

Port Name	Dir	W	Generated by	Description
clock	i	1	Automatically	Port for clock signal
reset	i	1	Automatically	Port for reset signal
set	i	1	Automatically	Port for set signal
clear	i	1	Automatically	Port for clear signal
init	i	1	Automatically	Port for init signal
enable	i	1	Automatically	Port for enable signal
event_signal	io?	1	Automatically	Port event signal

TABLE 7.18 Extra ports for an atomic register

Port Name	Dir	W	Generated by	Description
clock	i	1	Automatically	Port for clock signal
reset	i	1	Automatically	Port for reset signal
set	i	1	Automatically	Port for set signal
clear	i	1	Automatically	Port for clear signal
init	i	1	Automatically	Port for init signal
enable	i	1	Automatically	Port for enable signal
lock_signal	o	1	Automatically	Port for lock signal

TABLE 7.19 Extra ports for a counter register

Port Name	Dir	W	Generated by	Description
clock	i	1	Automatically	Port for clock signal
reset	i	1	Automatically	Port for reset signal

TABLE 7.19 Extra ports for a counter register

Port Name	Dir	W	Generated by	Description
set	i	1	Automatically	Port for set signal
clear	i	1	Automatically	Port for clear signal
init	i	1	Automatically	Port for init signal
enable	i	1	Automatically	Port for enable signal
cnt_output	o	ud	Automatically	Port for count output signal
gray_output	o	ud	add_logic(gray_output);	Port for gray count output signal
cnt_dir_signal	i	1	add_logic(cnt_dir_signal);	Port for count direction signal
inc_signal	i	1	add_logic(inc_signal);	Port for count-up signal
dec_signal	i	1	add_logic(dec_signal);	Port for count-down signal

TABLE 7.20 Extra ports for an interrupt register

Port Name	Dir	W	Generated by	Description
clock	i	1	Automatically	Port for clock signal
reset	i	1	Automatically	Port for reset signal
set	i	1	Automatically	Port for set signal
clear	i	1	Automatically	Port for clear signal
init	i	1	Automatically	Port for init signal
enable	i	1	Automatically	Port for enable signal
interrupt_signal	o	1	Automatically	Port for interrupt signal

TABLE 7.21 Extra ports for a shift register

Port Name	Dir	W	Generated by	Description
clock	i	1	Automatically	Port for clock signal
reset	i	1	Automatically	Port for reset signal
set	i	1	Automatically	Port for set signal

TABLE 7.21 Extra ports for a shift register

Port Name	Dir	W	Generated by	Description
clear	i	1	Automatically	Port for clear signal
init	i	1	Automatically	Port for init signal
enable	i	1	Automatically	Port for enable signal
sh_op_signal	i	1	add_logic(sh_op_signal);	Port for shift operation signal

TABLE 7.22 Extra ports for a semaphore register

Port Name	Dir	W	Generated by	Description
clock	i	1	Automatically	Port for clock signal
reset	i	1	Automatically	Port for reset signal
set	i	1	Automatically	Port for set signal
clear	i	1	Automatically	Port for clear signal
init	i	1	Automatically	Port for init signal
enable	i	1	Automatically	Port for enable signal
done_signal	o	1	Automatically	Port for done signal

Ports automatically generated by cslc for a all registers:

```
port: input - clock
```

DESCRIPTION :

Automatical create the clock port with the name *clock* for the *reg_object_name* register. All the operations with the register will be executed on the specified clock edge

```
port: input - reset
```

DESCRIPTION :

Automatical create the asynchronous reset port with the name *reset* for the *reg_object_name* register. An asynchronous signal affects any circuit without considering the clock signal transitions.
open

```
port: input - set
```

DESCRIPTION :

Automatical create the synchronous set port with the name *set* for the *reg_object_name* register.

```
port: input - clear
```

DESCRIPTION :

Automatical create the clear port with the name *clear* for the *reg_object_name* register. When the clear signal is active the register will be loaded with the clear value.

```
port: input - enable
```

DESCRIPTION :

Automatical create the enable port with the name *clear* for the *reg_object_name* register. Write and read operation will be made with respect for enable and clock signal.

```
port: input - init
```

DESCRIPTION :

Automatical create the initialization port with the name *init_signal* for the *reg_object_name* register. When the initialization signal is active the register will be initialized with the *init_value*.

```
port: input - parallel_input
```

DESCRIPTION :

Automatical create the parallel input port with the name *reg_input* for the *reg_object_name* register. The input port is n-bit width (user defined).

```
port: output - parallel_output
```

DESCRIPTION :

Automatical create the parallel output port with the name *reg_output* for the *reg_object_name* register. The output port is n-bit width (user defined).

Extra ports automatically generated by cslc for an action register:

```
port: input - clock
```

DESCRIPTION :

Automatical create the clock port with the name *clock* for the *reg_object_name* register. All the operations with the register will be executed on the specified clock edge

```
port: input - reset
```

DESCRIPTION :

Automatical create the asynchronous reset port with the name *reset* for the *reg_object_name* register. An asynchronous signal affects any circuit without considering the clock signal transitions.
open

```
port: input - set
```

DESCRIPTION :

Automatical create the synchronous set port with the name *set* for the *reg_object_name* register.

```
port: input - clear
```

DESCRIPTION :

Automatical create the clear port with the name *clear* for the *reg_object_name* register. When the clear signal is active the register will be loaded with the clear value.

```
port: input - enable
```

DESCRIPTION :

Automatical create the enable port with the name *clear* for the *reg_object_name* register. Write and read operation will be made with respect for enable and clock signal.

port: input - init

DESCRIPTION :

Automatical create the initialization port with the name *init_signal* for the *reg_object_name* register. When the initialization signal is active the register will be initialized with the *init_value*.

port: output - event_signal

DESCRIPTION :

Automatical create the event port with the name *event_signal* for the *action_reg_name* action register.

Extra ports automatically generated by cslc for an atomic register:

port: input - clock

DESCRIPTION :

Automatical create the clock port with the name *clock* for the *reg_object_name* register. All the operations with the register will be executed on the specified clock edge

port: input - reset

DESCRIPTION :

Automatical create the asynchronous reset port with the name *reset* for the *reg_object_name* register. An asynchronous signal affects any circuit without considering the clock signal transitions.
open

port: input - set

DESCRIPTION :

Automatical create the synchronous set port with the name *set* for the *reg_object_name* register.

port: input - clear

DESCRIPTION :

Automatical create the clear port with the name *clear* for the *reg_object_name* register. When the clear signal is active the register will be loaded with the clear value.

port: input - enable

DESCRIPTION :

Automatical create the enable port with the name *clear* for the *reg_object_name* register. Write and read operation will be made with respect for enable and clock signal.

port: input - init

DESCRIPTION :

Automatical create the initialization port with the name *init_signal* for the *reg_object_name* register. When the initialization signal is active the register will be initialized with the *init_value*.

port: output - lock_signal

DESCRIPTION :

Automatical create the lock port with the name *lock_signal* for the *atomic_reg_name* atomic register.

Extra ports automatically generated by cslc for a counter register:

port: input - clock

DESCRIPTION :

Automatical create the clock port with the name *clock* for the *reg_object_name* register. All the operations with the register will be executed on the specified clock edge

port: input - reset

DESCRIPTION :

Automatical create the asynchronous reset port with the name *reset* for the *reg_object_name* register. An asynchronous signal affects any circuit without considering the clock signal transitions.
open

port: input - set

DESCRIPTION :

Automatical create the synchronous set port with the name *set* for the *reg_object_name* register.

port: input - clear

DESCRIPTION :

Automatical create the clear port with the name *clear* for the *reg_object_name* register. When the clear signal is active the register will be loaded with the clear value.

port: input - enable

DESCRIPTION :

Automatical create the enable port with the name *clear* for the *reg_object_name* register. Write and read operation will be made with respect for enable and clock signal.

port: input - init

DESCRIPTION :

Automatical create the initialization port with the name *init_signal* for the *reg_object_name* register. When the initialization signal is active the register will be initialized with the *init_value*.

port: output - cnt_output

DESCRIPTION :

Automatical create the counter output port with the name *cnt_output* for the *counter_reg_name.* The width of the *cnt_output* signal is equal to the width of counter register.

Extra ports automatically generated by cslc for an interrupt register:

port: input - clock

DESCRIPTION :
Automatical create the clock port with the name *clock* for the *reg_object_name* register. All the operations with the register will be executed on the specified clock edge

```
port: input - reset
```

DESCRIPTION :
Automatical create the asynchronous reset port with the name *reset* for the *reg_object_name* register. An asynchronous signal affects any circuit without considering the clock signal transitions.
open

```
port: input - set
```

DESCRIPTION :
Automatical create the synchronous set port with the name *set* for the *reg_object_name* register.

```
port: input - clear
```

DESCRIPTION :
Automatical create the clear port with the name *clear* for the *reg_object_name* register. When the clear signal is active the register will be loaded with the clear value.

```
port: input - enable
```

DESCRIPTION :
Automatical create the enable port with the name *clear* for the *reg_object_name* register. Write and read operation will be made with respect for enable and clock signal.

```
port: input - init
```

DESCRIPTION :
Automatical create the initialization port with the name *init_signal* for the *reg_object_name* register. When the initialization signal is active the register will be initialized with the *init_value*.

```
port: output - interrupt_signal
```

DESCRIPTION :
Automatical create the interrupt port with the name *interrupt_signal* for the *interrupt_reg_name* interrupt register. When an interrupt request will be validated the interrupt signal state will be changed

Extra ports automatically generated by cslc for a semaphore register:

```
port: input - clock
```

DESCRIPTION :
Automatical create the clock port with the name *clock* for the *reg_object_name* register. All the operations with the register will be executed on the specified clock edge

```
port: input - reset
```

DESCRIPTION :
Automatical create the asynchronous reset port with the name *reset* for the *reg_object_name* register. An asynchronous signal affects any circuit without considering the clock signal transitions.

open

`port: input - set`

DESCRIPTION :

Automatical create the synchronous set port with the name *set* for the *reg_object_name* register.

`port: input - clear`

DESCRIPTION :

Automatical create the clear port with the name *clear* for the *reg_object_name* register. When the clear signal is active the register will be loaded with the clear value.

`port: input - enable`

DESCRIPTION :

Automatical create the enable port with the name *clear* for the *reg_object_name* register. Write and read operation will be made with respect for enable and clock signal.

`port: input - init`

DESCRIPTION :

Automatical create the initialization port with the name *init_signal* for the *reg_object_name* register. When the initialization signal is active the register will be initialized with the *init_value*.

`port: output - done_signal`

DESCRIPTION :

Automatical create the done output port with the name *done_signal* for the *semaphore_reg_name* semaphore register. This port is 1bit width.

7.4 CSL Register Examples

The following sections contain CSL register examples.

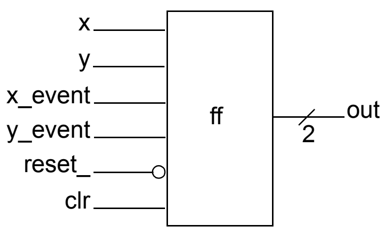
Examples section and replace with text explaining cells can have relative and absolute addresse
The cell address is relative to another cell's ending address

```
cell_name.mem_addr_abs(cell_addr_expr);
cell_name.mem_addr_rel(cell_addr_expr, addr_inc_amount);
```

!! Must add a counter up to these examples (isn't a counter up already present?)

7.4.1 FF with event bit update

FIGURE 7.39 FF with event bit update



CSL CODE

```
csl_reg ff;
ff.clock_name(clk);
ff.reset_val(0);
ff.reset_name(reset_);
ff.clr_val(0);
ff.width(2);
ff.event(x, 0);
ff.event(y, 1);
```

VERILOG CODE

```
module ff(reset_, clr, x_event, x, y_event, y, clk, out);
  input reset_;
  input clr;
  input x_event;
  input x;
  input y_event;
```

```
  input y;
  input clk;
  output reg[1:0] out;
  always @(posedge clk or negedge reset_) begin
    if (~reset_)
      out <= 2'b00;
    else if (x_event)
      out[0] <= x;
    else if (y_event)
      out[1] <= y;
    else if (clr)
      out <= 2'b00;
  end
endmodule
```

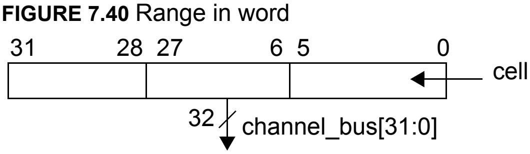
Certain bits in the register ff change based on an event. When the event occurs then the register is updated: when x_event happens the first bit of the register changes to x and when y_event happens the second bit changes to y.
The same bit can be modified by different conditions.

```
...
else if (x_event)
  out[0] <= x;
else if (y_event)
  out[0] <= y;
...
```

In the example above whatever event happens, the first bit is modified.

7.4.2 Using cells in registers

FIGURE 7.40 Range in word



specify the ranges for the word

```
rang0=[5:0]
rang1=[27:6]
rang2=[31:28]
```

construct the word by concatting the cells

```
cls_reg word0={r2,r1,r0};
```

The width of the register and cell names is inferred from the concatenation.

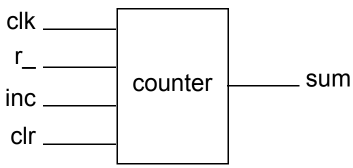
7.4.3 Counter register

Every time clk is on positive edge and inc is active (1), the output sum is incremented by INC_AMT(as long as r_ is not 0 and clr is not 1).

CSL CODE

```
create_counter(.width(1), .clk(clk), .reset(r_), .inc (inc), .clr
(clr), .setval(???), .cnt_out(sum));
```

FIGURE 7.41 Counter register



VERILOG CODE

```
//If a define is used then no define is created. If a number is used
//then a define is created.
`define SEND_BYTE_CNT_DW 10
wire        [`SEND_BYTE_CNT_DW-1:0] send_byte_cnt    ;
wire                                send_byte_cnt_inc;
wire                                send_byte_cnt_clr;

inc #(`SEND_BYTE_CNT_DW) inc_send_byte_cnt
  (.clk (clk),
   .r_  (reset_),
   .inc (cnt_inc),
   .clr (cnt_clr),
   .sum (cnt));

module inc(clk, r_, inc, clr, sum);
parameter W = 1;
parameter INC_AMT = 1;
parameter INIT_VAL = 0;
input clk;
input r_;
input inc;
```

```
input clr;
output reg [W-1:0] sum;
always @(posedge clk or negedge r_) begin
  if (~r_)
    sum <= INIT_VAL;
  else if (clr)
    sum <= INIT_VAL;
  else if (inc)
    sum <= sum + INC_AMT;
end
endmodule
```

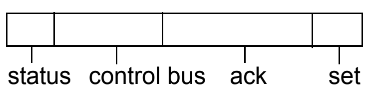
7.4.4 Register cells

Cells in the register can be defined using the name of the register as a prefix to the cell name

CSL CODE

```
    !!!!!1convert it to a register example
cell status.FOO             [7]   rw;
    field status.control_bus     [6:3] rw;
    field status.control.ack     [1]   rw;
    field status.control.set     [0]   rw;
```

FIGURE 7.42



GENERATED C++ CODE

```
#ifndef __GEN_I_<NAME>_VH_
#define __GEN_I_<NAME>_VH_

//
// DO NOT EDIT - automatically generated by <toolname>!
//
// --------------------------------------------------------------------
-------
//
// Copyright (c) <year>, <company name>
// All Rights Reserved.
```

```
//
// This is UNPUBLISHED PROPRIETARY SOURCE CODE of <company name>;
// the contents of this file may not be disclosed to third parties,
copied or
// duplicated in any form, in whole or in part, without the prior writ-
ten
// permission of <company name>.
//
// RESTRICTED RIGHTS LEGEND:
// Use, duplication or disclosure by the Government is subject to
restrictions
// as set forth in subdivision (c)(1)(ii) of the Rights in Technical
Data
// and Computer Software clause at DFARS 252.227-7013, and/or in simi-
lar or
// successor clauses in the FAR, DOD or NASA FAR Supplement. Unpub-
lished -
// rights reserved under the Copyright Laws of the United States.
//

# Generated C++ section
# toolname :                      <toolname>
# path to tool:                    <path>
# tool version:                    <version>
#time stamp for tool:          <tool time stamp>
# generated from filename : <filename>
# file timestamp                 <source file time stamp>
# generated timestamp       <current file time stamp>
// Register STATUS_0
!!change all the register names and change the cell names STATUS etc
#define STATUS_0                     0xc
#define STATUS_0_RESET_NUM                   0x0
#define STATUS_0_BSY_SHIFT                   7
#define STATUS_0_BSY_FIELD                   (0x1<<STATUS_0_BSY_SHIFT)
#define STATUS_0_BSY_RANGE                   7:7
#define STATUS_0_BSY_DEFAULT                 0x0
#define STATUS_0_DRDY_SHIFT                  6
#define STATUS_0_DRDY_FIELD                 (0x1<<STATUS_0_DRDY_SHIFT)
#define STATUS_0_DRDY_RANGE                  6:6
#define STATUS_0_DRDY_DEFAULT                        0x0
#define STATUS_0_DRQ_SHIFT                   3
```

```
#define STATUS_0_DRQ_FIELD                  (0x1<<STATUS_0_DRQ_SHIFT)
#define STATUS_0_DRQ_RANGE                  3:3
#define STATUS_0_DRQ_DEFAULT                0x0
#define STATUS_0_ERR_SHIFT                  0
#define STATUS_0_ERR_FIELD                  (0x1<<STATUS_0_ERR_SHIFT)
#define STATUS_0_ERR_RANGE                  0:0
#define STATUS_0_ERR_DEFAULT                0x0
#define STATUS_LAST_REG STATUS_0 // 0x000d
```

GENERATED VERILOG CODE

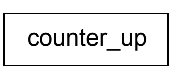
```
`define SEND_BYTE_CNT_DW 10
wire        [`SEND_BYTE_CNT_DW-1:0] send_byte_cnt    ;
wire                                send_byte_cnt_inc;
wire                                send_byte_cnt_clr;

inc #(`SEND_BYTE_CNT_DW) inc_send_byte_cnt
  (.clk (clk          ),
   .r_  (reset_      ),
   .inc (cnt_inc),
   .clr (cnt_clr),
   .sum (cnt    ));
```

7.4.5

CSL CODE

```
<CSL code>
```

FIGURE 7.43 counter up



GENERATED VERILOG CODE

```
module cnt_up (clk   ,
               reset_,
               inc   ,
               clr   ,
               sum);

parameter WIDTH    = 1;
parameter INIT_VAL = 0;
```

```
parameter INC_AMT  = 1;

input              clk   ;
input              reset_;
input              inc   ;
input              clr   ;
output [WIDTH-1:0] sum   ;
reg    [WIDTH-1:0] sum   ;

always @(negedge reset_ or posedge clk) begin
   if (~reset_)  sum <= INIT_VAL     ;
   else if (clr) sum <= INIT_VAL     ;
   else if (inc) sum <= sum + INC_AMT;
end

endmodule
```

GENERATED C++ CODE

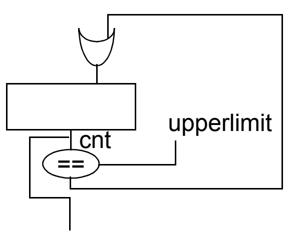
```
<C++ code>
```

7.4.6 Counter up to limit

CSL CODE

```
<CSL code>
```

FIGURE 7.44



GENERATED VERILOG CODE

```
module cnt_up_to_limit (clk   ,
                        reset_,
                        inc   ,
                        clr   ,
```

```
                         sum);

parameter WIDTH       = 1;
parameter INIT_VAL    = 0;
parameter UPPER_LIMIT = 1;
parameter INC_AMT     = 1;

input              clk   ;
input              reset_;
input              inc   ;
input              clr   ;
output [WIDTH-1:0] sum   ;
reg    [WIDTH-1:0] sum   ;

// stop the counter from incrementing once the limit is reached
wire stop_cnt = UPPER_LIMIT = sum;

wire qual_inc = inc && ~stop_cnt;
wire qual_clr=clr || stop_cnt;

always @(negedge reset_ or posedge clk) begin
   if (~reset_)  sum <= INIT_VAL     ;
   else if (qual) sum <= INIT_VAL     ;
   else if (qual_inc) sum <= sum + INC_AMT;
end

endmodule
```

GENERATED C++ CODE

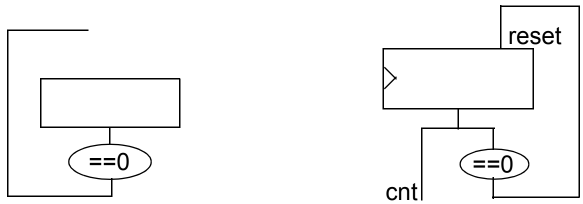
```
<C++ code>
```

7.4.7 Counter down to zero

CSL CODE

```
<CSL code>
```

FIGURE 7.45



GENERATED VERILOG CODE

```
module cnt_down_to_zero (clk   ,
                  reset_,
                  dec   ,
                  clr   ,
                  zero  , Count down to zero
                  sum);

parameter WIDTH    = 1;
parameter INIT_VAL = 0;
parameter DEC_AMT  = 1;

input              clk   ;
input              reset_;
input              inc   ;
input              clr   ;
output [WIDTH-1:0] sum   ;
output             zero  ;
reg    [WIDTH-1:0] sum     ;

always @(negedge reset_ or posedge clk) begin
   if (~reset_)  sum <= INIT_VAL     ;
   else if (clr) sum <= INIT_VAL     ;
   else if (zero)sum <= INIT_VAL     ;
   else if (inc) sum <= sum - DEC_AMT;
end

assign zero = ~|sum; // all bits shoulb be zero, OR all bits (should be
0) and then invert the result

endmodule
```

GENERATED C++ CODE

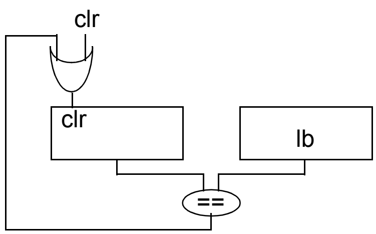
```
<C++ code>
```

7.4.8 Counter down to lower bound

CSL CODE

```
<CSL code>
```

FIGURE 7.46



GENERATED VERILOG CODE

```
// count down to lower bound

module cnt_down_to_lb (clk   ,
                       reset_,
                       dec   ,
                       clr   ,
                       lbe   , // equal to lower bound
                       sum);

parameter WIDTH       = 1;
parameter INIT_VAL    = 0;
parameter LOWER_BOUND = 0;
parameter DEC_AMT     = 1;

assign  lbe = LOWER_BOUND == sum;

input              clk   ;
input              reset_;
input              inc   ;
input              clr   ;
```

```
output [WIDTH-1:0] sum   ;
output             zero  ;
reg    [WIDTH-1:0] sum     ;

always @(negedge reset_ or posedge clk) begin
   if (~reset_)  sum <= INIT_VAL     ;
   else if (lbe) sum <= INIT_VAL     ;
   else if (clr) sum <= INIT_VAL     ;
   else if (inc) sum <= sum - DEC_AMT;
end

// width mismatch here because of parameter.
assign zero = LOWER_BOUND == sum; // all bits shoulb be zero, OR all
bits (should be 0) and then invert the result

endmodule
```

GENERATED C++ CODE

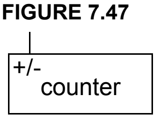
```
<C++ code>
```

7.4.9 Counter up/down

CSL CODE

```
<CSL code>
```

FIGURE 7.47



GENERATED VERILOG CODE

```
module cnt_up_down (clk     ,
                    reset_ ,
                    cnt    ,
                    clr    ,
                    up_down, // 0 down, 1 up
                    sum);

parameter WIDTH    = 1;
```

```
parameter INIT_VAL = 0;
parameter CNT_AMT  = 1;

input              clk   ;
input              reset_;
input              inc   ;
input              clr   ;
output [WIDTH-1:0] sum   ;
reg    [WIDTH-1:0] sum   ;

// take the two's complement of the CNT_AMOUNT if the down count is
enabled and
// we want to subtract CNT_AMOUNT.
// otherwise add the CNT_AMOUNT

wire    [WIDTH-1:0] inc_dec_amount = up_down ? ~CNT_AMT + 1 : CNT_AMT;

always @(negedge reset_ or posedge clk) begin
   if (~reset_)  sum <= INIT_VAL;
   else if (clr) sum <= INIT_VAL;
   else if (cnt) sum <= sum + inc_dec_amount ;
end

endmodule
```

GENERATED C++ CODE

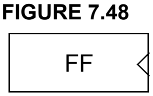
```
<C++ code>
```

7.4.10 Flip flop

CSL CODE

```
<CSL code>
```

FIGURE 7.48



GENERATED VERILOG CODE

```
module ff (clk,
```

```
            reset_,
            d,
            q);

parameter WIDTH = 1;

input              clk   ;
input              reset_;
input  [WIDTH-1:0] d     ;
output [WIDTH-1:0] q     ;
reg    [WIDTH-1:0] q     ;

always @(negedge reset_ or posedge clk) begin
   if (~reset_) q <= {WIDTH{1'b0}};
   else  q <= d;
end

endmodule
```

GENERATED C++ CODE

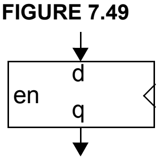
```
<C++ code>
```

7.4.11 Flip flop with enable

CSL CODE

```
<CSL code>
```

FIGURE 7.49



GENERATED VERILOG CODE

```
module ff_en (clk   ,
              reset_,
              d     ,
              en    ,
              q);
```

```
parameter WIDTH = 1;

input              clk   ;
input              reset_;
input [WIDTH-1:0]  d     ;
input              en    ;
output [WIDTH-1:0] q     ;
reg [WIDTH-1:0]    q     ;

always @(negedge reset_ or posedge clk) begin
   if (~reset_) q <= {WIDTH{1'b0}};
   else if (e)  q <= d;
end

endmodule
```

GENERATED C++ CODE

```
<C++ code>
```

7.4.12 Flip flop with enable and set

CSL CODE

```
<CSL code>
```

FIGURE 7.50

GENERATED VERILOG CODE

```
module ff_en_set (clk   ,
                  reset_,
                  set   ,
                  d     ,
                  en    ,
                  q     );
```

```
parameter WIDTH       = 1;
parameter SET_VALUE   = 0;
parameter RESET_VALUE = 0;

input               clk   ;
input               reset_;
input               set   ;
input [WIDTH-1:0]   d     ;
input               en    ;
output [WIDTH-1:0] q      ;
reg [WIDTH-1:0]    q      ;

always @(negedge reset_ or posedge clk) begin
   if (~reset_)  q <= RESET_VALUE;
   else if (set) q <= SET_VALUE  ;
   else if (en)  q <= d          ;
end

endmodule
```

GENERATED C++ CODE

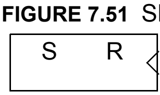
```
<C++ code>
```

7.4.13 SR flip flop

CSL CODE

```
<CSL code>
```

FIGURE 7.51 SRFF



GENERATED VERILOG CODE

```
module srff (clk   ,
             reset_,
             set   ,
             reset ,
             q);
```

```
parameter WIDTH = 1;

input clk   ;
input reset_;
input set   ;
input reset ;
output q    ;
reg    q    ;

wire    val_mux = reset ? 1'b0 ( set ? 1'b1 : q) ;

always @(negedge reset_ or posedge clk) begin
   if (~reset_) q <= 1'b0;
   else q <= val_mux;
end

endmodule
```

GENERATED VERILOG CODE

```
#ifndef __GEN_I_<NAME>_VH_
#define __GEN_I_<NAME>_VH_

//
// DO NOT EDIT - automatically generated by <toolname>!
//

// ----------------------------------------------------------------------
-------
//
// Copyright (c) <year>, <company name>
// All Rights Reserved.
//
// This is UNPUBLISHED PROPRIETARY SOURCE CODE of <company name>;
// the contents of this file may not be disclosed to third parties,
copied or
// duplicated in any form, in whole or in part, without the prior writ-
ten
// permission of <company name>.
//
// RESTRICTED RIGHTS LEGEND:
```

```
// Use, duplication or disclosure by the Government is subject to
restrictions
// as set forth in subdivision (c)(1)(ii) of the Rights in Technical
Data
// and Computer Software clause at DFARS 252.227-7013, and/or in simi-
lar or
// successor clauses in the FAR, DOD or NASA FAR Supplement. Unpub-
lished -
// rights reserved under the Copyright Laws of the United States.
//

# Generated verilog section
# toolname :                       <toolname>
# path to tool:                     <path>
# tool version:                     <version>
#time stamp for tool:           <tool time stamp>
# generated from filename : <filename>
# file timestamp                  <source file time stamp>
# generated timestamp       <current file time stamp>

#define <name>_WIDTH 16
#define <name>_RANGE 15:0
#define <name>_ADDR 0

// Register <reg_name>_0
#define <reg_name>_0_WIDTH 8
#define <reg_name>_0_RANGE 7:0
#define <reg_name>_0 32'h0
#define <reg_name>_0_RESET_NUM 8'bxxxxxxxx
#define <reg_name>_0_INIT_NUM 8'h0

// cells belonging to the above register
// there are n fields which in total width can equal but not exceed the
width of the above register definition
#define <reg_name>_0_<field_name>_WIDTH <field_width>
#define <reg_name>_0_<field_name>_RANGE <field_range>
#define <reg_name>_0_<field_name>_RW <r=2, rw=3> // 10 and 11
#define <reg_name>_0_<field_name>_NUM  <field_width>'h<value>  //
devulat for <value> is 0

Example:
```

```
// Register <register_name>_0
#define <register_name>_0 32'h5
#define <register_name>_0_RESET_NUM 2'bxx
#define <register_name>_0_INIT_NUM 3'h0
#define <register_name>_0_RANGE 2:1
#define <register_name>_0_WIDTH 2
#define <register_name>_0_<field_name>_RANGE 2
#define <register_name>_0_<field_name>_WIDTH 1
#define <register_name>_0_<field_name>_RW 3
#define <register_name>_0_<field_name>_DEFAULT 1'h0

#define <register_name>_0_<field_name>_RANGE 1
#define <register_name>_0_<field_name>_WIDTH 1
#define <register_name>_0_<field_name>_RW 3
#define <register_name>_0_<field_name>_DEFAULT 1'h0

#define BASE_ADDRESS_MODULE 32'h00000000

#endif __GEN_I_<NAME>_VH_
```

7.4.14 Example register code

```
module cnt_up (clk   ,
               reset_,
               inc   ,
               clr   ,
               sum);

parameter WIDTH    = 1;
parameter INIT_VAL = 0;
parameter INC_AMT  = 1;

input              clk   ;
input              reset_;
input              inc   ;
```

```
input              clr   ;
output [WIDTH-1:0] sum   ;
reg    [WIDTH-1:0] sum   ;

always @(negedge reset_ or posedge clk) begin
   if (~reset_)  sum <= INIT_VAL     ;
   else if (clr) sum <= INIT_VAL     ;
   else if (inc) sum <= sum + INC_AMT;
end

endmodule



module cnt_up_to_limit (clk   ,
                        reset_,
                        inc   ,
                        clr   ,
                        sum);

parameter WIDTH       = 1;
parameter INIT_VAL    = 0;
parameter UPPER_LIMIT = 1;
parameter INC_AMT     = 1;

input              clk   ;
input              reset_;
input              inc   ;
input              clr   ;
output [WIDTH-1:0] sum   ;
reg    [WIDTH-1:0] sum   ;

// stop the counter from incrementing once the limit is reached
wire stop_cnt = UPPER_LIMIT = sum;

wire qual_inc = inc && ~stop_cnt;

always @(negedge reset_ or posedge clk) begin
   if (~reset_)  sum <= INIT_VAL     ;
   else if (clr) sum <= INIT_VAL     ;
```

```
   else if (qual_inc) sum <= sum + INC_AMT;
end

endmodule


module cnt_down_to_zero (clk   ,
                 reset_,
                 dec   ,
                 clr   ,
                 zero  , Count down to zero
                 sum);

parameter WIDTH    = 1;
parameter INIT_VAL = 0;
parameter DEC_AMT  = 1;

input              clk   ;
input              reset_;
input              inc   ;
input              clr   ;
output [WIDTH-1:0] sum   ;
output             zero  ;
reg    [WIDTH-1:0] sum   ;

always @(negedge reset_ or posedge clk) begin
   if (~reset_)  sum <= INIT_VAL     ;
   else if (clr) sum <= INIT_VAL     ;
   else if (inc) sum <= sum - DEC_AMT;
end

assign zero = ~|sum; // all bits shoulb be zero, OR all bits (should be
0) and then invert the result

endmodule


// count down to lower bound

module cnt_down_to_lb (clk   ,
```

```
                  reset_,
                  dec   ,
                  clr   ,
                  zero  , Count down to zero
                  sum);

parameter WIDTH       = 1;
parameter INIT_VAL    = 0;
parameter LOWER_BOUND = 0;
parameter DEC_AMT     = 1;

input                clk   ;
input                reset_;
input                inc   ;
input                clr   ;
output [WIDTH-1:0] sum   ;
output               zero  ;
reg    [WIDTH-1:0] sum     ;

always @(negedge reset_ or posedge clk) begin
   if (~reset_)  sum <= INIT_VAL     ;
   else if (clr) sum <= INIT_VAL     ;
   else if (inc) sum <= sum - DEC_AMT;
end

// width mismatch here because of parameter.
assign zero = LOWER_BOUND == sum; // all bits shoulb be zero, OR all
bits (should be 0) and then invert the result

endmodule

module cnt_up_down (clk    ,
                    reset_ ,
                    cnt    ,
                    clr    ,
                    up_down, // 0 down, 1 up
                    sum);

parameter WIDTH    = 1;
parameter INIT_VAL = 0;
```

```
parameter CNT_AMT  = 1;

input              clk   ;
input              reset_;
input              inc   ;
input              clr   ;
output [WIDTH-1:0] sum   ;
reg    [WIDTH-1:0] sum   ;

// take the two's complement of the CNT_AMOUNT if the down count is
enabled and
// we want to subtract CNT_AMOUNT.
// otherwise add the CNT_AMOUNT

wire    [WIDTH-1:0] inc_dec_amount = up_down ? ~CNT_AMT + 1 : CNT_AMT;

always @(negedge reset_ or posedge clk) begin
   if (~reset_)  sum <= INIT_VAL;
   else if (clr) sum <= INIT_VAL;
   else if (cnt) sum <= sum + inc_dec_amount ;
end

endmodule




module ff (clk,
           reset_,
           d,
           q);

parameter WIDTH = 1;

input              clk   ;
input              reset_;
input  [WIDTH-1:0] d     ;
output [WIDTH-1:0] q     ;
reg    [WIDTH-1:0] q     ;
```

```
always @(negedge reset_ or posedge clk) begin
   if (~reset_) q <= {WIDTH{1'b0}};
   else  q <= d;
end

endmodule

module ff_en (clk   ,
              reset_,
              d     ,
              en    ,
              q);

parameter WIDTH = 1;

input              clk   ;
input              reset_;
input [WIDTH-1:0]  d     ;
input              en    ;
output [WIDTH-1:0] q     ;
reg [WIDTH-1:0]    q     ;

always @(negedge reset_ or posedge clk) begin
   if (~reset_) q <= {WIDTH{1'b0}};
   else if (e)  q <= d;
end

endmodule

module ff_en_set (clk   ,
                  reset_,
                  set   ,
                  d     ,
                  en    ,
                  q     );

parameter WIDTH       = 1;
parameter SET_VALUE   = 0;
parameter RESET_VALUE = 0;
```

```
input              clk   ;
input              reset_;
input              set   ;
input [WIDTH-1:0]  d     ;
input              en    ;
output [WIDTH-1:0] q     ;
reg [WIDTH-1:0]    q     ;

always @(negedge reset_ or posedge clk) begin
   if (~reset_)  q <= RESET_VALUE;
   else if (set) q <= SET_VALUE  ;
   else if (en)  q <= d          ;
end

endmodule

module srff (clk   ,
             reset_,
             set   ,
             reset ,
             q);

parameter WIDTH = 1;

input clk   ;
input reset_;
input set   ;
input reset ;
output q    ;
reg    q    ;

wire    val_mux = reset ? 1'b0 ( set ? 1'b1 : q) ;

always @(negedge reset_ or posedge clk) begin
   if (~reset_) q <= 1'b0;
   else q <= val_mux;
end

endmodule
```

<BEGIN OF MOVED HERE CONTENTS>

- adding state elements to the design
  - the leaf level modules contain state elements(SE) such as flip-flops, latches, FIFOs, regfiles and memories

the elements in the leaf level modules are defined with the CSL register specification.

</END OF MOVED HERE CONTENTS>

Garbage or to be REMOVED

reg_name.**counter(**[width][,initial][,final][,step][,direction]**)**;

DESCRIPTION :

Define the register type as counter. Following is a detailed description of the optional parameters which can be passed to the **counter()** method. Width is used to specify the bit size of the output. Initial is the starting value of the counter and can be a numeric expression or a constant. Final is the ending value of the counter (the value to which the counter will increment/decrement) and can be a numeric expression or a constant.Step defines the increment/decrement amount of the counter and can be a numeric expression or a constant. Direction specifies the counting direction: if the **up** keyword is used the counter will increment using the step size, else if the **down** keyword is used the counter will decrement using the step size

Default values:If the width parameter is not specified, default value is 32bits. If no intial value is specified, this variable will default to zero. If no final value is declared the default will be the maximum according to the bit range of the counter output signal. If the counter output signal is not declared, this will default to 32 bits. The default, if no step variable is declared will be 1.

EXAMPLE :

In this example a register of type counter is declared. The **counter()** method can take different parameters. In this case a 3bit counter is created: it starts counting at 1(001 in binary), ends at 7(111 in binary) and increments with 1 unit at each iteration.

CSL CODE

```
csl_unit Top;
csl_reg reg_cnt;
reg_cnt.counter(3’b001,3’b111,1,up);
```

SEE ALSO :

n/a

VERILOG CODE

```
module Top;
  wire [2:0] out;
  reg_cnt reg_cnt_0(out,clk);
endmodule


//the following module contains a behavioral model of a counter up
module reg_cnt(out,clk);
  output [2:0] out;
  reg output = 3’b001;
  always@ (posedge clk) begin
    if(out<=3’b111)
      out <= out + 1’b001;
    else
      out <= 1’b001;
  end
```

```
endmodule
```

----

reg_name.**interrupt(**[width],[mask]**)**;

DESCRIPTION :

Define the register type as interrupt. This type of register is updated by event detectors that may be level sensitive or edge sensitive.The **interrupt()** method may receive two parameters. The width sets the bit size of the register (it can be a numeric expression or a constant). The mask specifies if the interrupt register uses a mask (the mask is loaded into another register) and it can be a numeric expression, a constant, or a variable if the mask is loaded dynamically
Default values, if only one or none of the parameters are specified are: 32bits for the width, and no mask register.

EXAMPLE :

CSL CODE

```
csl_define IRQ_MASK 8'b00001111;
csl_unit Top;
csl_reg reg_irq;
reg_irq.interrupt(8,IRQ_MASK);
```

SEE ALSO :

n/a

VERILOG CODE

```
'define IRQ_MASK 8'b00001111;
module Top;
endmodule
module interrupt_subreg(out,in);
  input [7:0] in;
  output [7:0] out;
  reg out;
  always@(in) begin
    out<=in;
  end
endmodule
module mask_subreg(out,set);
  input [7:0] set;
  output [7:0] out;
```

----

reg_name.**cell(***cell_name*, *bitrange*, *attributes***);**

DESCRIPTION :

n/a

EXAMPLE :

CSL CODE

```
//csl code goes here
```

```
status.cell(unitx, bitrange, attributes);
```

DESCRIPTION :

n/a

EXAMPLE :

CSL CODE

```
//csl code goes here
```

```
cell_name.enum(enum_name = value);
```

DESCRIPTION :

Create an enumerated value which is associated with the cell. Create as many enums as needed:

EXAMPLE :

CSL CODE

```
//csl code goes here
```

cell_name.**range(***range***)**;

DESCRIPTION :

Create a range for the cell.

EXAMPLE :

CSL CODE

```
//csl code goes here
```

```
memory_map_name.cell(cell_name);
```

DESCRIPTION :

Create a new cell.

EXAMPLE :

CSL CODE

```
//csl code goes here
```

memory_map_name.cell_name.**range(***bit_range***);**

DESCRIPTION :

Create a range for the cell.

EXAMPLE :

CSL CODE

```
//csl code goes here
```

memory_map_name.cell_name.**attributes(***attrribute_list***);**

DESCRIPTION :

Apply attributes to the cell.

EXAMPLE :

CSL CODE

```
//csl code goes here
```

memory_map_name.cell_name.**cell(***cell_name***);**

DESCRIPTION :

Create a new cell within a cell.

EXAMPLE :

CSL CODE

```
//csl code goes here
```

```
object_name.get_msb(constant_numeric_expression); //add this for every object that has a width
object_name.set_msb(constant_numeric_expression);
```

DESCRIPTION :

The bit position of the msb bit in the memory element range.

EXAMPLE :

CSL CODE

```
csl_reg reg_x;
reg_x.width(32);
int xw = reg_x.lsb();
int yw = reg_x.msb();
```

```
object_name.get_lsb(constant_numeric_expression);   //add this for
every object that has a widt
```

object_name.**set_lsb(***constant_numeric_expression***);**

DESCRIPTION :

The bit position of the lsb bit in the memory element full range.

EXAMPLE :

CSL CODE

```
//csl code goes here
```

----

reg_name.**interrupt(***mask***)**;

DESCRIPTION :

Define the register type as interrupt *width* is a *numeric_expression* which is the mask of the register

EXAMPLE :

CSL CODE

```
//csl code goes here
```

reg_name.**semaphore(***signal_name***);**

DESCRIPTION :

Define the register type as semaphore.

EXAMPLE :

CSL CODE

```
//csl code goes here
```

----

reg_name.**event();**

DESCRIPTION :

Define the register type as event. Each bit in the event register is associated with an event signal and is set whenever a *signal_name* is asserted; the signal_name can have an ~ associated with it, showing that it has a low assertion level

EXAMPLE :

CSL CODE

```
//csl code goes here
```

register_name.**status()**;

DESCRIPTION :

creates a status register

EXAMPLE :

CSL CODE

```
//csl code goes here
```

<ADD>

Action Sequence Register

The Action Sequence Register specifies the sequence of expected actions.

An action is a memory rd/wr or an event.

An action has to complete before the next action can occur. Memory rds/wrs which are back to back can occur without waiting but must complete before the next event.

rd x

rd y

event q

FIGURE 7.52 Edge detectors

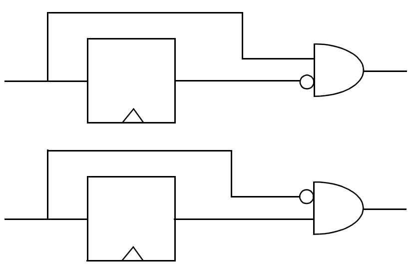
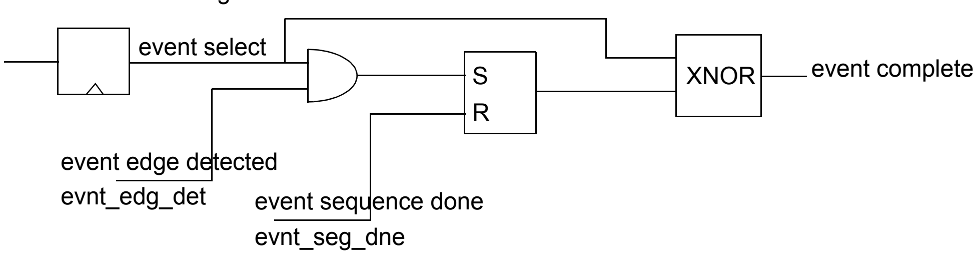


FIGURE 7.53 Event register



</ADD>

<ADD>
Mechanisms to reduce the bandwidth between the host processor and the application specific device (ASD).

TABLE 7.23 Semaphore

Request	transition (s)
Wait	wait for event
Continue	after

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.

FIGURE 7.54

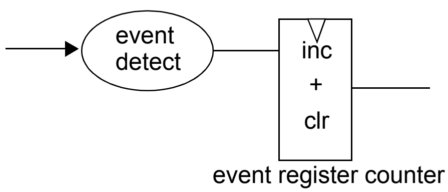


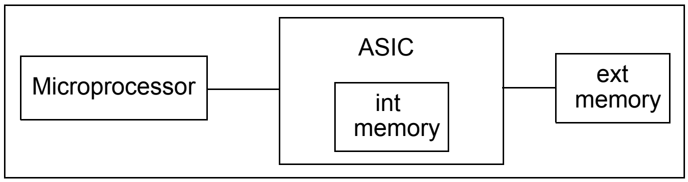
FIGURE 7.55

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

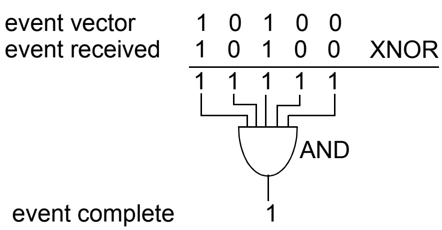
- rising/falling edge
- low/high level

FIGURE 7.56



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 7.57

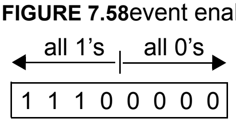


Ordered event tracking

A → B → C

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

FIGURE 7.58event enable register



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

FIGURE 7.59

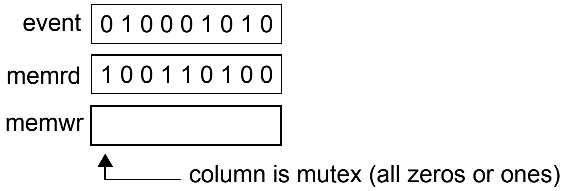
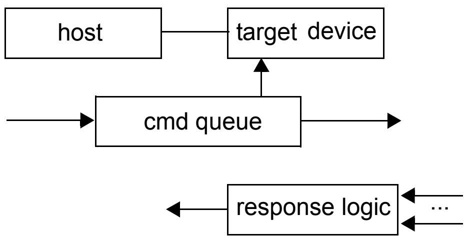


FIGURE 7.60 Mechanism for host software control



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.

</ADD>

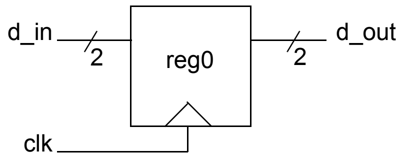
Removed Examples:

```
port:- input clock;
```

EXAMPLE :

Create a register with clock signal

FIGURE 7.61 A register with clock signal



CSL CODE

```
//AV
csl_reg reg_d(2);
reg_d.connect_clock(clk);
reg_d.connect_input(d_in,2);
reg_d.connect_output(d_out,2);
```

VERILOG CODE

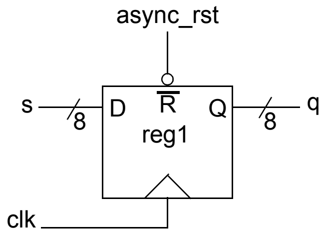
```
//AV
module reg_d(clk,d_in,d_out);
    input clk;
    input [1:0] d_in;
    output [1:0] d_out;
    reg [1:0] d_out;
    always @(posedge clk) begin
          d_out = d_in;
    end
endmodule
```

port: input **reset;**

EXAMPLE :

Create a register named *reg1* that will have an asynchronous reset pin.

FIGURE 7.62 A register with asynchronous reset signal



CSL CODE

```
//AV
csl_reg reg1(8);
reg1.set_type(dff);
reg1.connect_input(s,8);
reg1.connect_output(q);
reg1.reset_value(8'b0);
reg1.connect_reset(async_rst);
reg1.connect_clock(clk);
```

VERILOG CODE

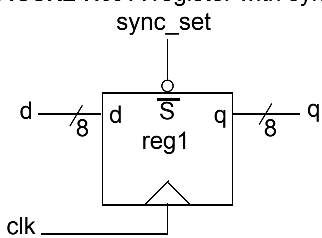
```
//AV
module reg1(async_rst,clk,s,q);
    input async_rst,clk;
    input [7:0] s;
    output [7:0] q;
    reg [7:0] q;
    always @(posedge clk or negedge async_rst) begin
        if (! async_rst) begin q = 8'b0; end
        else begin q = s; end
    end
endmodule
```

port: input **set;**

EXAMPLE :

Create a register named *reg1* that will have an synchronous set pin.

FIGURE 7.63 A register with synchronous set signal



CSL CODE

```
//AV
csl_reg reg1(8);
reg1.set_type(dff);
reg1.connect_input(d,8);
reg1.connect_output(q);
reg1.set_value(8'b1);
reg1.connect_set(sync_set);
reg1.connect_clock(clk);
```

VERILOG CODE

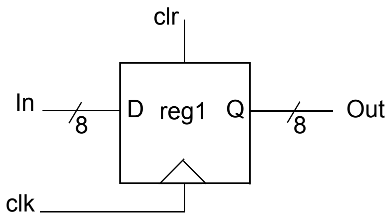
```
//AV
module reg1(sync_set,clk,d,q);
    input sync_set,clk;
    input [7:0] d;
    output [7:0] q;
    reg [7:0] q;
    always @(posedge clk ) begin
        if (! sync_set) begin q = 8'b1; end
        else begin q = d end
    end
endmodule
```

port: input **clear;**

EXAMPLE :

Create a registers. The registers will be cleared when the clear signal will be active.

FIGURE 7.64 A register with clear signal.



CSL CODE

```
//AV
csl_reg reg1;
reg1.set_range(7,0);
reg1.connect_clock(clk);
reg1.clear_value(8'b0);
reg1.connect_input(in,8);
reg1.connect_output(out,8);
reg1.connect_clear(clr);
```

VERILOG CODE

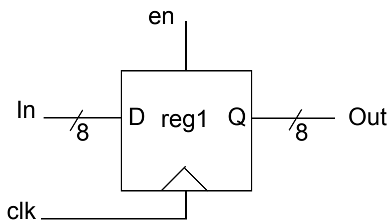
```
//AV
module reg1(clr,clk,in,out);
    parameter clr_val = 8'b0;
    input clr,clk;
    input [7:0] in;
    output [7:0] out;
    reg [7:0] out;
    always @(posedge clk) begin
        if(clr) begin out = clr_val; end
        else begin out = in; end
    end
endmodule
```

port: input enable

EXAMPLE :

Create a registers with enable pin

FIGURE 7.65 A register with enable signal.



CSL CODE

```
//AV
csl_reg reg1;
reg1.set_range(7,0);
reg1.connect_clock(clk);
reg1.connect_input(in,8);
reg1.connect_output(out,8);
reg1.connect_enable(en);
```

VERILOG CODE

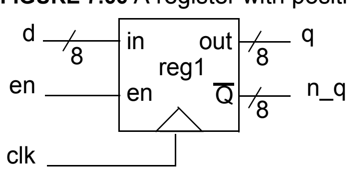
```
//AV
module reg1(clr,clk,in,out);
    input clr,clk;
    input [7:0] in;
    output [7:0] out;
    reg [7:0] out;
    always @(posedge clk) begin
        if(en) begin out = in; end
    end
endmodule
```

port: in neg_output

EXAMPLE :

Create a register with positive and negative output signals.

FIGURE 7.66 A register with positive and negative outputs



CSL CODE

```
//AV
csl_reg reg1(8);
```

```
reg1.connect_clock(clk);
reg1.connect_input(d,8);
reg1.connect_output(q,8);
reg1.connect_neg_output(n_q,8);
reg1.connect_enable(en);
```

VERILOG CODE

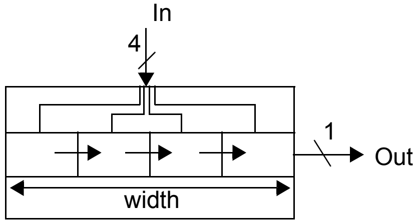
```
//AV
module rs_reg(en,clk,d,q,n_q);
    input en, clk;
    input [7:0] d;
    output [7:0] q, n_q;
    reg [7:0] q, n_q;
    always @(posedge clk) begin
       if(en) begin
           q = d;
           n_q = ~d;
       end
    end
endmodule
```

port: input serial_ouput

EXAMPLE :

Create a shift register that has serial output and paralel input.

FIGURE 7.67 A register with serial output and paralel input



CSL CODE

```
//AV
csl_reg shift_serial_output;
shift_serial_output.set_width(4);
shift_serial_output.set_type(sft);
shift_serial_output.connect_clock(clk);
shift_serial_output.set_shift_type(shr);
shift_serial_output.connect_serial_output(out);
```

```
shift_serial_output.connect_intput(in,4);
```

VERILOG CODE

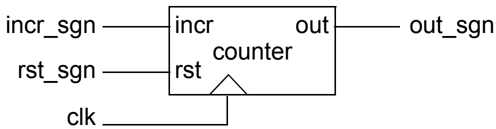
```
//AV
```

`Port: input inc_signal`

EXAMPLE :

Create a counter that will have a pin that connect the increment signal with this counter.

FIGURE 7.68 A counter with increment signal



CSL CODE

```
//AV
csl_reg cnt_inc(4);
cnt_inc.set_type(cnt);
cnt_inc.connect_clock(clk);
cnt_inc.connect_reset(rst_sgn);
cnt_inc.set_start_val(4’b0);
cnt_inc.connect_inc_signal(incr_sgn);
cnt_inc.connect_output(out_sgn);
```

VERILOG CODE

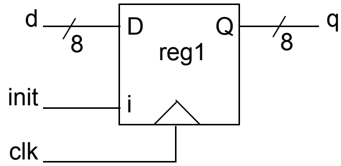
```
//AV
module cnt_inc(rst,clk,inc_sgn,out);
    parameter start_val = 4'b0;
    input rst, clk, inc_sgn;
    output [3:0] out;
    reg [3:0] out;
    always @(posedge clk or negedge rst) begin
        if(! rst) begin out = start_val; end
        else if(inc_sgn) begin out = out +1; end
    end
endmodule
```

port: input init_signal

EXAMPLE :

Creates a register that have an init signal.

FIGURE 7.69 A register with init signal



CSL CODE

```
//AV
csl_reg reg1(8);
reg1.connect_clock(clk);
reg1.init_value(8'b0);
reg1.connect_input(d,8);
reg1.connect_output(q,8);
reg1.connect_init_signal(init);
```

VERILOG CODE

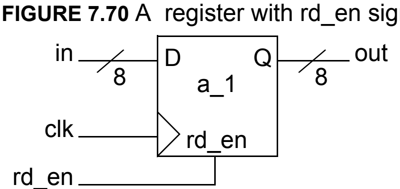
```
//AV
module reg1(clk,d,q,q_neg,init);
    parameter init_val = 8'b1;
    input en, clk;
    input [7:0] d;
    output [7:0] q;
    reg [7:0] q;
     always @(posedge clk) begin
       if(init_sgn) begin q = init_val; end
       else begin q = d; end
    end
endmodule
```

port: input rd_en

EXAMPLE :

Create a register with rd_en pin. The read operations are validated by rd_en signals.

FIGURE 7.70 A register with rd_en signal



CSL CODE

```
//AV
```

```
csl_reg a_1(8);
a_1.connect_clock(clk);
a_1.connect_input(d,8);
a_1.connect_ouput(q,8);
a_1.connect_rd_en(rd_en);
```

VERILOG CODE

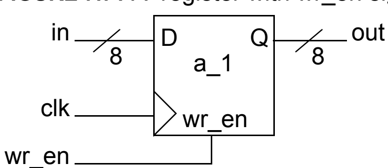
```
//AV
module rs_d(rd_en,clk,d,q);
    parameter init_val = 8'b0;
    input rd_en, clk;
    input [7:0] d;
    output [7:0] q;
    reg [7:0] q;
    reg [7:0] mem;
    always @(posedge clk) begin
       mem = d;
       if(rd_en) begin q = mem; end
    end
endmodule
```

port: input wr_en

EXAMPLE :

Create a register with wr_en pin. The write operations are validated by wr_en signals.

FIGURE 7.71 A register with wr_en signal



CSL CODE

```
//AV
csl_reg a_1(8);
a_1.connect_clock(clk);
a_1.connect_input(d,8);
a_1.connect_ouput(q,8);
a_1.connect_wr_en(wr_en);
```

VERILOG CODE

```
//AV
module rs_d(wr_en,clk,d,q);
```

```
    parameter init_val = 8'b0;
    input wr_en, clk;
    input [7:0] d;
    output [7:0] q;
    reg [7:0] q;
    reg [7:0] mem;
    always @(posedge clk) begin
       if(wr_en) begin q = d; end
    end
endmodule
```