CHAPTER 8 CSL Memory Map

TABLE 8.1 Chapter Outline

8.1 Definitions
8.2 CSL Memory Map Overview
8.3 CSL Memory Map Concepts
8.4 CSL Memory Map Command Summary
8.5 CSL Memory Map Commands

8.1 Definitions

TABLE 8.2 Definitions

cslc	CSL Compiler
SE	State Element
AS	Aggregate Structure

State Element is a memory element.
Aggregate Structure is an object which is composed of multiple elements.State element

8.1.1 The memory map methodology

The memory map serves as a specification for both the software and the hardware team.
For both hardware and software designers, it is good practice to use the names of the variables instead of hard-coding addresses and bit fields in their application code. Application code which refers to variable names is much clearer and much less sensitive to changes than code containing hard-coded numbers.

NOTE: Typically, a software engineer will encapsulate the mappings from names to addresses in a C header file, and a hardware engineer will encapsulate them in a RTL file. Header files and packages may also contain supporting functions that implement the access mechanism of a variable.

8.1.2 Address Definitions

TABLE 8.3 Address terminology

Term	Meaning
memory map	A set of registers/memory words, associated numeric addresses and symbolic names
base address	Memory map's starting address range
address range(s)	Memory map's starting and ending addresses
absolute address	Globally unique address (global address)
relative address	Address relative to another address (local address)
address increment amount	Amount to increment between adjacent addresses

TABLE 8.4 Determining register widths

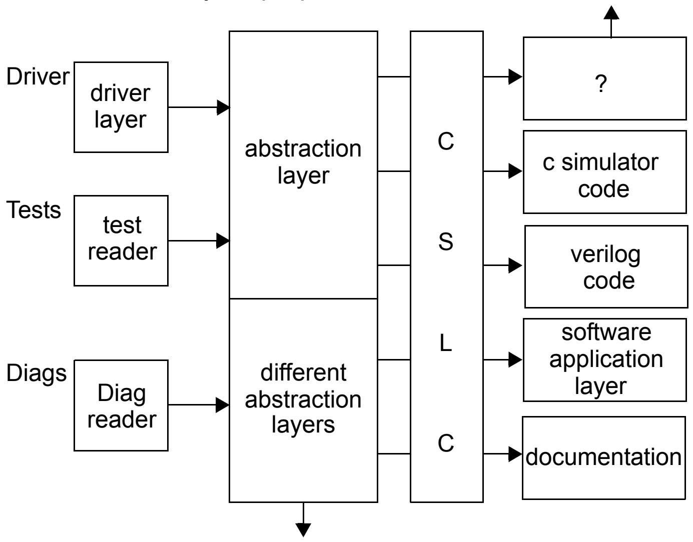
word width	register or memory word width in bits
fields	named sub-vectors in register or memory words
byte write enables	byte enables in the register or memory word

8.2 CSL Memory Map Overview

The CSL creates the memory map for entire chips or sub-hierarchies of chips. The program can declare memory mapped locations' registers, assign addresses to the registers, and create the state element code in the target language. The memory mapped elements can be register file, seperate flip-flops, registers, FIFO's, SRAM's, and/or any combination of the above.
The CSL memory map specification is used to specify the format to store data. The format includes the name and size of each data item in a memory element, the relative distance between memory elements, the unit that the memory element belongs to and an initialisation value for the memory element. Also can be specified an attribute for the memory map: r -can be read, w -can be written.
The user specifies a memory map for a chip design using the CSL memory map file (contains CSL memory map commands). CSL memory map commands can be specified in different files. The CSL memory map specification is used to assign addresses to sequential elements (e.g. register, register file, sram) in a design. The memory address space which is specified by the CSL memory map specification refers to structures which are generated from both the memory map specification and other CSL hardware specifications such as a register file specification. The CSL Compiler (cslc) will compile the CSL memory map specification. The cslc will then create memory mapped structures which are addressable by other hardware units or by software through bus acceses to the unit containing the

memory mapped structure.

FIGURE 8.1 Memory map Specification



8.2.1 Intro

The CSL lanaguage is used to declare registers, assign addresses to the registers, and create the state element code in the target language. In addition, the associated structures (e.g. register) are also generated with different types of attributes.
Why is a memory map generator required for chip design projects?
Memory maps are specified by the following groups on chip design projects team:

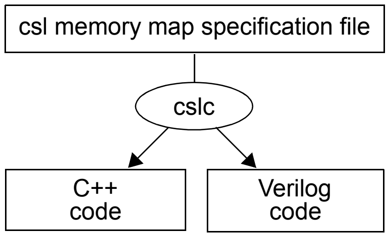
- RTL team
- C++ chip simulator team
- test team
- software driver team:
  - sw application team
  - documentation team

Each member of each team must have the correct address specification implemented in their code at all times. Using one source to generate the RTL and C++ constants and functions associated with the memory map ensures that all team members are using the same address constants at all times.

In addition, addresses in the memory map are made visible to different layers in the hardware and software stacks.

8.2.1.0.1 Flow

FIGURE 8.2 CSL memory map generation flow



8.2.1.1 Abstraction layer

Abstraction layers for device families make each subsequent device generation backwards compatible with device drivers by using an abstract layer. A hw abstraction layer uses a layer of indirection between the hw registers and the sw API.

TABLE 8.5 HAL

sw API
hw abstraction layer
hw registers/memory map

8.3 CSL Memory Map Concepts

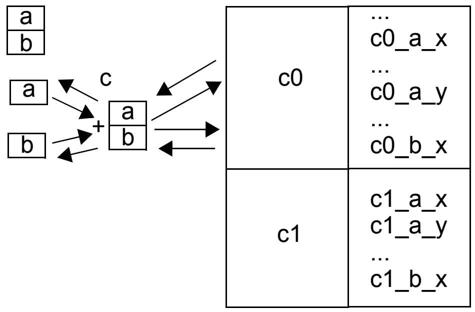
A memory map contains a starting address and an ending address. We define an address range as a pair of addresses (start_addr, end_addr). The address range may or may not be contigous. Each memory location can have a symbolic name. The symbolic name can be used to reference the memory location by Verilog and C++ code. Memory locations can contain fields (see Link to Register definition of field).

8.3.1 Address modes

There are two addressing modes (relative and absolute addressing):

- Relative addressing uses a base address and an offset to access a memory location. The base address is the start of a memory page. The base address plus the offset is the address of the memory location to access.
- Absolute addressing uses one specific address to access a specific memory location.

FIGURE 8.3

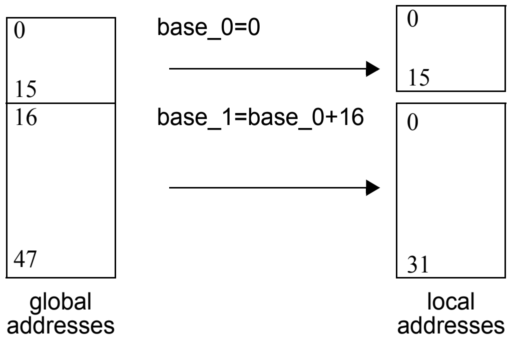


- partial registers can be accessed using regular addressing (e.g byte enables) in contiguous structures
- partial registers can be accessed using irregular field access in irregular registers outside of contiguous structures
- entire registers can be accessed in contiguous structures
- entire irregular registers can be accessed outside of contiguous structures

8.3.1.1 Global and local address maps

Local address maps are specified for Aggregate Structures (AS). The local address maps base address is 0 (zero). Addresses within the address maps may be declared either illegal (will never be used) or reserved for future use.
AS address maps can be added to other local address maps or the global address map. When the AS is added to another address map, the AS memory elements can be accesed with either the local or global addresses. The cslc will generate functions which convert global addresses to local adresses.

FIGURE 8.4 Global and local address maps



8.3.2 CSL memory map specifications

The CSL(Chip SpecificationLanguage) is used to create a memory map specifications. A memory location in the memory map is associated with hardware state elements in the design. Scalar elements (single elements) or aggregate structures or groups are added to the memory map. The objects are inserted into the memory map at the next address (current_addr + auto_inc_amount - structure ?) or at an absolute address specified by the user. Aggregate structures (containing more than one state element) are added by specifying their base address and the number of elements in the aggregate structure.

8.3.3 Uses

Memory map is used by software to:

- control the machine
- write data to the machine
- read data from the machine
- checkpoint the machine state
- restore the machine state

8.3.4 Memory map elements

The following is a list of CSL elements that can be added to a CSL memory:
Each address corresponds to a state element.

TABLE 8.6 Memory Map Hardware elements

Element	Mnemonic
Latch	LA
Flip-flop	FF
Register	Reg
Register File	RF
SRAM	SRAM
FIFO	FIFO

8.3.5 Register write

8.3.5.1 Writing to an element in the memory map

All memory elements may be written using functions which write the individual fields or

which write a value to all fields.
If the raw value is greater than 32-bits then an array of 32-bit values is written.

8.3.6 Access

Software or hardware can “access“ memory mapped elements.

8.3.7 Generated code and docs

The cslc compiles CSL memory map specifications into address constants in different output languages.
The cslc converts memory map addresses into absolute and relative addresses, ranges, masks which are created in C/C++ and RTL defines. Generated output from CSL register description:

- documentations
- C/C++ defines
- RTL defines
- C/C++ code
- Verilog code
- VHDL code

Constants are generated in the following forms: C++ const int, C #define and Verilog ‘define macros; C/C++ shift and mask, Verilog part selects and bit ranges. The following functions are also generated:

- Base address decoders
- Unit address decoders
- Global to local address convertors
- Address range checkers
- Wrapper logic to write/read registers, FF, FIFO, stack, RegFile, SRAM, memories

8.3.7.1 Defines types

There are 4 different kinds of defines in the spec file:

```
csl_define is used in the CSL specification file
csl_define_v_c used to generate the C/C++ #define and verilog ‘define
csl_define_c used to generate the C/C++ #define
csl_define_v used to generate the verilog ‘define
```

8.3.8 How to create memory map specification

- Create the memory map name
- Create the memory map base address
- Choose to autoincrement addresses (optional)

Next the user writes a memory element specification. The memory element specification includes the following:

- name of the memory element,
- the address of the memory element (either relative to the previous memory element or an absolute address)
- the width of the register
- the field declarations

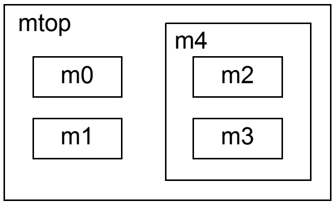
Field declarations

- name of the field
- attribute bits for the field
- width of the field

8.3.9 Creating hierarchical memory maps

Declare the memory objects. Create a hierarchy of declared memory objects. In the Figure 8.5 is an example with a hierarchical memory map which is created as follows: the m2 and m3 memory maps are enclosed within m4 memory map. m4, m0 and m1 memory maps are contained inside of the mtop memory map.
A memory map can be included in another memory map.

FIGURE 8.5 Hierarchical memory map



CSL CODE

```
//the memory map objects are declared
csl_memory_map m0,m1,m2,m3,m4,mtop;

//the memory map hierarchy is created
m4.add_element(m2);
m4.add_element(m3);
```

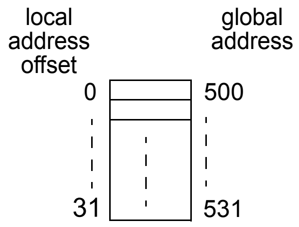
```
mtop.add_element(m0);
mtop.add_element(m1);
mtop.add_element(m4);
```

The memory hierarchy will be converted into a set of hierarchical memories which can be written and read with decoders that create the write_enable's and mux trees to select the data.

8.3.10 Memory Map Generator

- assign unique names to unique memory locations unless otherwise specified
- memory location is separated into fields (fields in CSL)
- memory map is made out of registers, register files, fifos, other state elements
- a cheker will check if there are no duplicates
- fields in language
- every object have address

FIGURE 8.6 Local address(relative offset) and global address(absolute)



8.3.11 Memory map checkers

The cslc will check the memory map specification for correctness using the following checks:

- an address is in the global address space.
- an address is in the local address space.
- an address is not in a reserved or illegal address range.
- there are no state elements without address.
- there are no state elements with the same address.
- all memory elements have addresses which fall inside of the address space.
- no memory elements have addresses which fall outside of the address space.
- no memory elements have adresses which are unaligned.
- the width of the register does not exceed the word width for the memory range.
- no duplicate names are allowed.
- no illegal names in terms of the target programming languages are allowed.

8.3.11.1 HW unit address bounds check

An address range checker is used to check for valid addresses. If the address is in the object's memory range (addr >= unit_base_addr && addr <= unit_limit_addr) then the address is a valid object address, else the address is either an illegal address if the object has to respond to address operations or a don't care address if the unit only responds to addresses which are valid object addresses.
The memory map specification can generate a bad address checker which sets an error bit, captures the bad address in a register and generates an interrupt.

PSEUDO CODE

```
if((ADDR_BASE <= address) && (ADDR_MAX >= address))
  address in range
else
  address_out_of_range
```

8.3.11.2 Analysis

If there is an address collision, an address checker will generate an error. An address collision occurs when 2 or more memory elements map to the same address and an alias wrapping has not been defined.

8.3.12 Tester

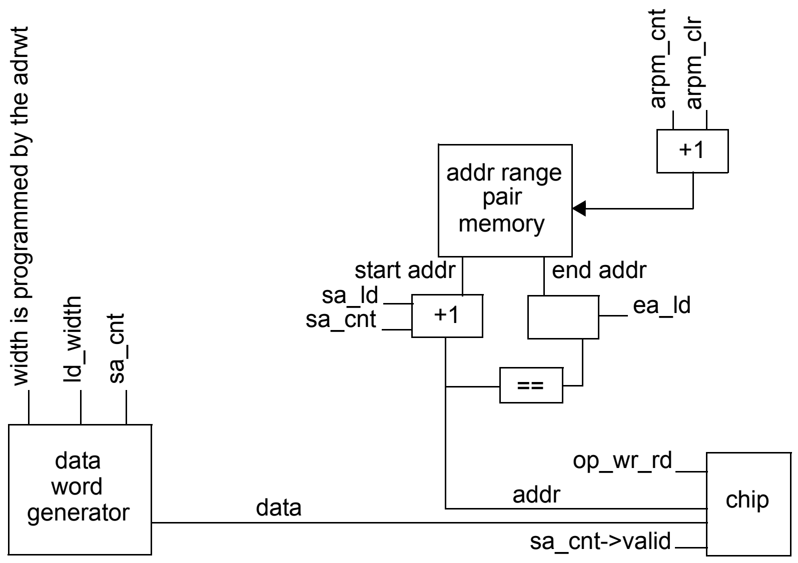
cslc generates memory map tester(C++ and RTL code) which is used to verify that each memory address can be written and read correctly. The automatic memory map checker performs a write, read and compare on each memory location in the memory map. The automatic memory map tester generates a report which contains the results of the evaluation. The tester can be used in a testbench or synthesized into hardware to include in the design.
Control implements the state machine.

FIGURE 8.7 State machine to control data path

address
range
width
?

	start order	end order	width
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
...			

FIGURE 8.8



8.3.13 Multiple instance cases for hierarchical memory maps

FIGURE 8.9

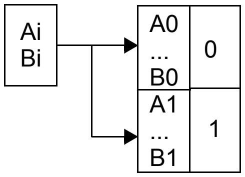


FIGURE 8.10

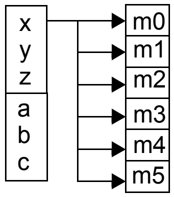
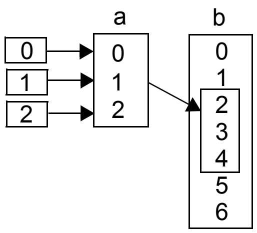


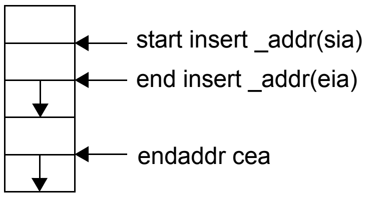
FIGURE 8.11



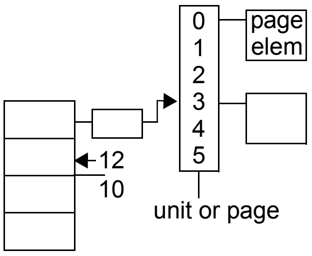
a - add objects to the mem map page

b - add mem map page to the mem map

FIGURE 8.12 Flat



1 more addresses
if success go to 2
else
2 insert addresses



psa = page start addr
pea = page end addr

P0 SA 0
P0 EA 4
P0 contains 0-2

8.3.14 Endianess

8.3.14.1 Address endianess

Address Endianes refers to the way the address is stored in memory and can be in **big-endian** or **little-endian** format. Little-endian order is when the smallest component address (the least significant bit (LSB)) is stored first in memory, while big-endian is when the largest component address (the most significant byte (MSB)) is stored first. For example, in a system with 4bit addressing if a component resides at address 0x7 in the larger address block 0xA, then the global address of the particular component would be stored as 0x7A in little endian format and 0xA7 in big endian format. Endianness does not specify what the value ends when stored in memory, but rather which end it begins with.

TABLE 8.7

little endian	b3	b2	b1	b0
big endian	b0	b1	b2	b3

8.3.15 Visibility

8.3.15.1 Address visibility

Many different users access the chip address space. Portions of the address space are made visible to specific types of users by means of address visibility. The following types of address visibility can be set:

- complete address map (all inclusive)
- architectually visible state – state visible to the software
- context switch state
- hardware resetable register must be reset in order for design to work
- software resetable register must be reset in order for design to work
- hardware abstraction layer
- test mode visible - visible during test mode

Figure 8.13 shows an example with the address visibility map

FIGURE 8.13 Example address visibility memory map

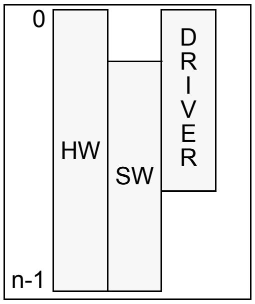
	N	AVS	CSS	HWR	SWR	HAL	TMV
0	X					X	X
1	X		X	X	X		X
2	X	X	X				X
	X	X	X				X
	X	X	X	X	X		X
	X						X
n-1	X		X	X			X

address

Only the portion of the address map that is required to carry out certain operations is made visible to the particular type of user.

- internal address space available to software driver
- external address space exposed to other applications

FIGURE 8.14 Address visibility map



8.3.15.2 Software visible memory map

The software team will write software which directly accesses the elements in the software visible memory map.

8.3.16 Address assignment

8.3.16.1 Assigning addresses to memory elements/registers

Both the absolute addressing method and the relative addressing method can be used to assign an address to a memory element. Both (the absolute addressing method and the relative addressing method) can be used to construct a memory map.

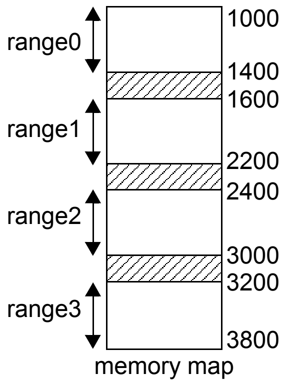
- The absolute addressing refers to the case when the exact address of the element is specified using a numeric expression: **base_addr** = *number*

- The relative addressing is obtained by adding a certain amount (increment) to a base address. The new address is thus relative to the base address from which it was computed: **relative** = *offset* + base_address

8.3.17 Address ranges

Address ranges can be contiguous or non-contiguous. The address range may not be contiguous if there are unused addresses. A concatentation of ranges/addresses is allowed. Address ranges can be associated with FIFO's or Register Files.

FIGURE 8.15 Create a memory map with a concatenation of ranges



8.3.17.1 Address alignment

A memory can be viewed as a sequence of bytes. A sequence of bytes can be partitioned into groups of bytes. Word alignment refers to the number of bytes in a memory word that are addresable. Individual memory words are n bits wide. Each individual memory word is addresable, the width of the memory word determining the width of the memory alignment.

Word alignment refers to the way words are stored and addressed in memory. Word-aligned means that the contents is stored, for example, at an address that is divisible by 4 (word size is 4 bytes or 32 bits). The same principle can be applied to byte quantities (8 bits) which can be stored at any address in memory, halfword quantities (16 bits) stored at addresses divisible by 2, doubleword quantities (64 bits) stored at addresses divisible by 8 etc.
The alignment of the memory words can be optionally specified in terms of the number of

bits or of the following:

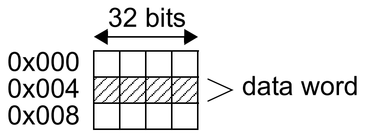
TABLE 8.8 Memory variable byte alignement and suffix

byte alignement	suffix added to variables which use this alignment
1 byte	_a8
2 byte	_a16
4 byte	_a32
8 byte	_a64
16 byte	_a128

One can increment the addresses considering the 128-bit boundary.

Example: a 32 bit data word which is byte aligned has 4 addressable bytes (it is byte addressable)

FIGURE 8.16 Memory map



8.3.18 Address increment

The address increment is a function of the memory element size and the data word size. Increment by 1, 2, 4, 8 bytes from the previous word in file. Default: *addr_inc = alignment_size / word_size;*

The next address may be computed using the current address and the increment address amount or the next address may be set using the **csl_set_addr** command.
Commands to jump to a new address:
memory_map_page_name.set_next_address(numeric_expression);
set_endianess(endianess_type);

8.3.18.1 Incrementing the memory map's current insertion address

The current address counter contains the address to insert the next scalar or aggregate memory element. The current address is set in the relative addressing mode by adding the size of the last memory element inserted in the memory address space to the current address. In the absolute addresing mode the current address is set by one of the following methods:

1. current address is set to a new address location

2. an offset is added to current address

Elements may also be added to a memory map by using the autoincrement function which automatically calculates the current address to insert the next State Element into the memory map.

8.3.18.2 Relative address for a specific memory location.

Addresses can be incremented relative to the previous memory element address using the **inc_addr** *amount* operation. If the **CSL incr_addr** amount is specified then the amount is added to the current address to get the next address. Else the size of the word and the address word alignment are used to determine the amount to add to the current address to get the next address.
increment_amount=data_word_width/alignment.

8.3.18.3 Address auto increment

The address increment amount is the amount to automatically add to an element during relative address insertion operations. The default auto increment amount to add to the current address during relative address insertion operations is one. The auto increment amount can be overridden by setting the auto increment amount explicitly in a constructor or using the *set_endianess(endianess_type);* function or by using the address alignment functions.

For each new register which is added to *memory_map_name*(mmn) is assigned a new address equal to last address + auto_address_increment amount, where aligment amount is set with the *set_alignment(numeric_expression);*

The amount to increment the address counter is computed by dividing the memory map aligment width in bits by the memory map data word width. For example, if the address word width is 32 and the addressable word width is 8 the address is incremented by 4 (8/32) for each new word added to the memory map (32 bits or 4 words).

8.3.18.4 Word aligned and byte aligned address increment

The amount to increment addresses can be specified with the **address_alignment** keyword.
Table 8.9 shows the different combinations of memory address alignment, the byte align-

ment sizes and the corresponding address increment amounts.

TABLE 8.9 Address types. memory word alignement, and address increment amounts

word size	alignment bit size	address increment amount (word to forward)
8 bits	8 bits	1
16 bits	8 bits	2
32 bits	8 bits	4
64 bits	8 bits	8
32 bits	32 bits	1
64 bits	32 bits	2

8.3.19 Address insert

8.3.19.1 Relative address insertion operations

8.3.20 Memory access rights

The CSL memory map specfication file sets the software rws attributes on a register and/or element basis.
Addresable memory elements can be accessed in several different ways. The default memory access attribute assignment is read/write. We can assign custom memory access attributes to individual state elements:

- read only
- write only
- read/write
- shadow

8.3.21 CSL code examples

Address Range is specified using **set_address_range** method for specifying addresses:

```
set_address_range(range|[concatenation_of_ranges]);
```

In the example in Figure 8.15 address ranges are allocated in a memory map. Using the **address_range** method address ranges can be custom specified according to the needs of the design:

CSL CODE

```
csl_memory_map mmap;
```

```
mmap.set_address_range({(1000,1400),(1600,2200),(2400,3000),(3200,3800
)});
```

Note that some ranges may remain unallocated. These will be unused until otherwise specified.

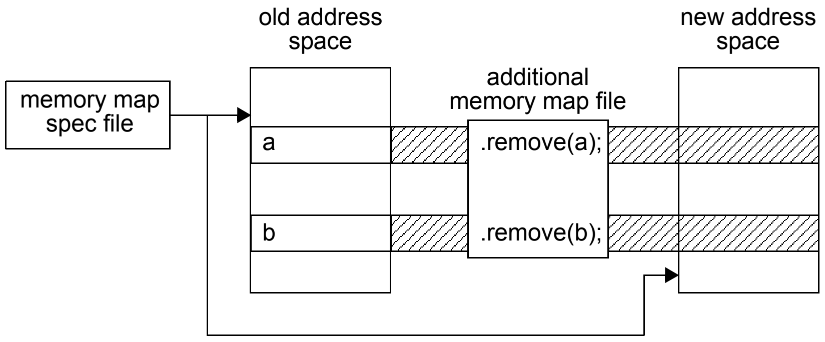
8.3.21.1 Hardware register exclusion

Registers or bit ranges may be excluded from the generated memory mapped structure using **remove()** methods:

```
memory_map_name0.append(memory_map_name1); //removes the memory ele-
ment from the memory map
memory_map_name0.append(memory_map_name1); //removes the range delim-
ited by the lower_limit and upper_limit from the current scope
```

The example in Figure 8.17 illustrates an address space to which a memory map specification file is applied. The memory map contains 2 registers (a and b) which are not needed in the new model yet the architecture must be kept intact. The **remove_element()** method simply removes the address space occupied by the two registers in the memory map and leaves it unallocated (now the two registers can no longer be addresed).

FIGURE 8.17 Register removal example



8.3.22 Generated Code

8.3.22.1 Generated C++ Code !!turn this to H2

```
#ifndef __csl_I_<NAME>_VH_
#define __csl_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
// generated from toolname : <toolname>
// path to tool:           : <path>
// tool version:           : <version>
// time stamp for tool:    : <tool time stamp>
// generated from filename : <filename>
// source filename:        : <filename>
// source file timestamp:  : <source file time stamp>
// generated file timestamp: <current file time stamp>

// Register register_name
#define register_name_REGISTER_ADDRESS        0x<address>

// value to reset the entire register to

// the following two fields can be defined using the field reset and
set values.
#define register_name_REGISTER_RESET_VAL       0x<reset_value>
#define register_name_REGISTER_SET_VAL         0x<set_value>

// the shift value is equal to the LSB bit position of the field
#define register_name_field_name_SHIFT_AMOUNT   <shift_value>
#define register_name_field_name_MASK           <mask>

// use the following define to set the value of the field
#define register_name_field_name_SET_SHIFT_AND_MASK <mask> << <shift>

// use the following define to get the value of the field
#define register_name_field_name_GET_SHIFT_AND_MASK <mask> >> <shift>

#define register_name_field_name_BITRANGE
<msb_bit_position>:<lsb_bit_position>
#define register_name_field_name_INIT_VAL  0x<field_init_value>
#define register_name_field_name_SET_VAL   0x<field_set_value>
#define memory_map_name_END_ADDRESS          <address>
```

8.3.22.2 Generated Verilog Code

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

//
// Generated by <toolname>
// DO NOT MODIFY
// -----------------------------------------------------------------------
-------
//
// 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
// generated from toolname : <toolname>
// path to tool:             <path>
// tool version:             <version>
// time stamp for tool:      <tool time stamp>
// generated from filename:  <filename>
// source filename:          <filename>
// source file timestamp:    <source file time stamp>
// generated file timestamp: <current file time stamp>

#define <name>_WIDTH16
#define <name>_RANGE15:0
#define <name>_ADDR0

// Register <reg_name>
```

```
#define <reg_name>_WIDTH8
#define <reg_name>_RANGE7:0
#define <reg_name> 32'h0
#define <reg_name>_RESET_NUM8'bxxxxxxxx
#define <reg_name>_INIT_NUM8'h0

//fields 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>_field_name_WIDTH<field_width>
#define <reg_name>_field_name_RANGE<field_range>
#define <reg_name>_field_name_RW<r=2, rw=3> // 10 and 11
#define <reg_name>_field_name_NUM  <field_width>'h<value>  // devulat
for
//<value> is 0

Example:

// Register register_name
#define register_name_ADDRESS 32'h<address>
#define register_name_RESET_VALUE 2'b<value>
#define register_name_SET_VALUE 3'h<value>
#define register_name_BITRANGE [<msb_bit_position>:<lsb_bit_position>]
#define register_name_REGISTER_WIDTH <width>
#define register_name_field_name_BITRANGE
#define register_name_field_name_field_WIDTH 1
#define register_name_field_name_ATTR
#define register_name_field_name_DEFAULT 1'h0
#define BASE_ADDRESS_<module_name>              <address>


class register_name : public register {
  register_name_ADDRESS 32'h<address>
  register_name_RESET_VALUE 2'b<value>
  register_name_SET_VALUE 3'h<value>
  register_name_BITRANGE [<msb_bit_position>:<lsb_bit_position>]
 field_name register_name_REGISTER_WIDTH <width>
  register_name_field_name_BITRANGE
  register_name_field_name_field_WIDTH 1
  register_name_field_name_ATTR
  register_name_field_name_DEFAULT 1'h0
  BASE_ADDRESS_<module_name>              <address>
}



#endif __csl_I_<NAME>_VH_
```

8.3.23 Generated code

8.3.23.1 Generated C++ Code

```
#ifndef_csl_1_<NAME>_VH_
#define_csl_1_<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, cop-
ied or duplicated in any form, in whole or in part, without the prior
written permission of <company name>
//
//RESTRICTED RIGHTS LEGEND:
// Use, dulpication or disclosure by the Government is subject to
restrictions as se
//forth in subdivision (c)(1)(ii) of the Rights in Technical Data and
Computer Soft
//ware clause at DFARS 252.227-7013, and/or in similar or succesor
clauses in the
//FAR, DOD or NASA FAR Supplement. Unpublished  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 timestamp>
#generated timestamp <current file time stamp>
//Register STATUS_0
#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 CEATA0_LAST_REG STATUS_0//0x000d
```

8.3.23.2 Generated Verilog Code

```
#ifndef_csl_|_<NAME>_VH_
#define_csl_|_<NAME>_VH
//DO  NOT EDIT -automatically generated by <toolname>!
// --------------------------------------------------------------------
--
///
//Copyright (c) <year><company name>
//All Rights Reserved
//
//this is UNPUBLISHED PROPRIETARY SOURCE CODE pf <company name>;
//the contents of this file may not be disclosed to third parties, cop-
ied or duplicated in any form, in whole or in part, without the prior
written permission of <company name>
```

```
//RESTRICTED RIGHTS LEGEND:
// Use, dulpication or disclosure by the Government is subject to
restrictions as se
//forth in subdivision (c)(1)(ii) of the Rights in Technical Data and
Computer Soft
//ware clause at DFARS 252.227-7013, and/or in similar or succesor
clauses in the
//FAR, DOD or NASA FAR Supplement. Unpublished  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>_WIDTH16
#define <name>_RANGE 15:0
#define <name>_ADDR0
//register <reg_name>_0
#define <reg_name>_0_WIDTH8
#define <reg_name>_0_RANGE 7:0
#define <reg_name>_0 32'h0
#define <reg_name>_0_RESET_NUM8'bxxxxxxxx
#define <reg_name>_0_INIT_NUM8'h0
//fields belonging to the above register
//there are n fields which in total can equal ubt not exceeded 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>
```

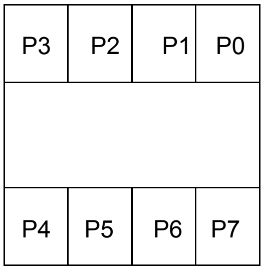
Example:

```
//register register_name_0
#define register_name_032'h5
#define register_name_0_RESET_NUM2'bxx
#define register_name_0_INIT_NUM3'h0
```

```
#define register_name_0_RANGE2:1
#define register_name_0_WIDTH2
#define register_name_0_field_name_RANGE2
#define register_name_0_field_name_WIDTH1
#define register_name_0_field_name_RW3
#define register_name_0_field_name_DEFAULT'h0
#define register_name_0_field_name_RANGE1
#define register_name_0_field_name_WIDTH1
#define register_name_0_field_name_RW3
#define register_name_0_field_name_DEFAULT1'h0
#deine BASE_ADDRESS_MODULE32'h00000000
#endif_csl_I_<NAME>_VH_
```

NOTE:<Move this to commands examples>

FIGURE 8.18

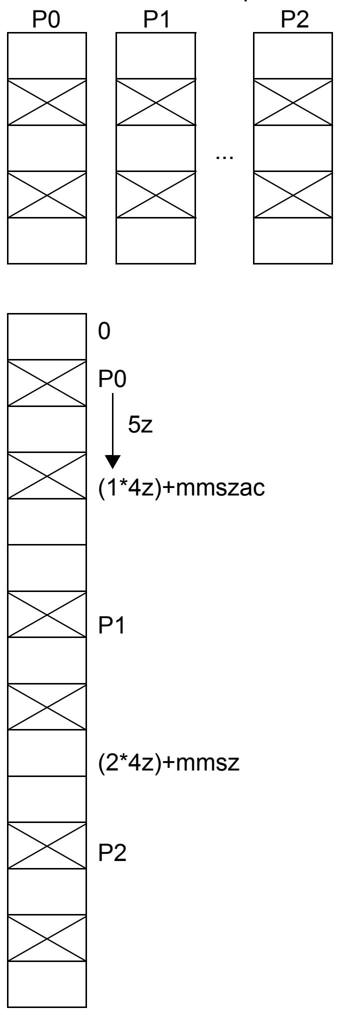


```
csl_memory_map mmn;
csl_unit p[0-7];
p[0-7].set_range(mmn, 0, 29);
set_address(mmn, \1.getadde_size()*\2);
```

- \1=p[0-7]
- \2=[0-7]

```
default address= lastobject.baseaddress()+lastob-
ject.addr_size()+mmn.inc_amounts
p[0-7].add_to_memory_map(mmn);
user next address which is equal to mmn.inc_amounts
```

FIGURE 8.19 Individual processors memory spaces and the combined memory map shrink figure



each processor address space starts at an offset
</Move this to commands examples>
<ADD>
move this ADD to sw components
Consumer Electronic Chips

FIGURE 8.20

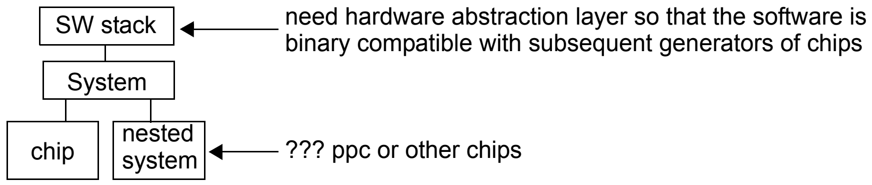
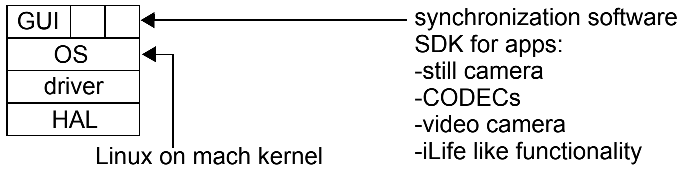

SW stack
System
chip
nested system
need hardware abstraction layer so that the software is binary compatible with subsequent generators of chips
??? ppc or other chips

FIGURE 8.21



</ADD>

<ADD>

NOTE:Code generation - move to code gen doc

!also move the generated code abvoe

<memory_map_class_name>::enum<register_name>_<field_name_in_register>_<enum _name_in_field>

All letters are capitilized except the enum.

Register output abreviations

mme = memory map element

```
fld  = field in a register
enum = enumerated type value for a given field
```

```
C/C++ classes will be generated with a prefix letter "C".
C/C++ enumerated types will be generated with a prefix letter "c"
(i.e. enum cenum<enumerated_type_name> {...}).
```

Verilog code will be generated with a prefix letter "v"
Verilog defines which are equivalent to the C/C++ enumerated will be generated with a prefix letter "cv" (i.e. `define venum<enumerated_type_name> <value>).

```
enumerated types should have an illegal field which can be returned
from C/C++ switch default cass and from Verilog cas statement
default cases. The illegal field can be "caught" by the "down-
stream" logic and can flag problems with switch and case statement
selector inputs.
```
</ADD>

Virtual Memory

SW address map is global.
HW address map is local.
Upper bits are the page ID.
Upper bits map to a unit ID.

TABLE 8.10 Virtual memory table

upper bits	global	local
0	m	0
0	n	(n-m)
1	p	0
1	q	(q-p)
2	b	0
2	c	(c-b)
3	d	0
3	e	e-d

FIGURE 8.22

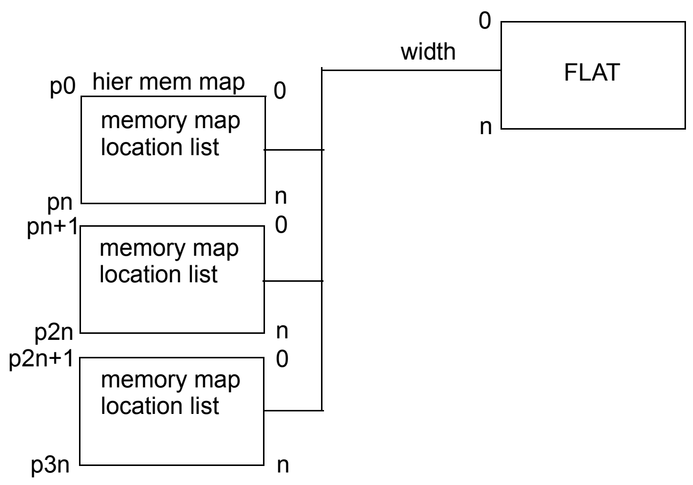


TABLE 8.11

		local
flat	x,y	x, y
	psa+m, psa+n address	m, n
hier	x, y	x, y
v m	sa=(pno<<amount) ea=(pno<<amount)	1.x 1.y x,y

VM base	VMPN in Addr
000000	0000000 = (0<<20) | 0
100000	0100000 = (1<<20) | 0000
200000	0200000 = (2<<20) | 0000
20 bits global	28 bit global

<ADDED_2007.05.12>

```
Different methods used for  programming chip registers
```

Chips contain registers which need to be configured with values
The register values also need to be read out to a different unit ont he chip or outside of the chip.

CSL provides a way to write a set of registers on a chip using one or 4 different
physical bus/network topologies. The

All buses contain essentially the same set of commands.

```
addr (address)
data
v    (valid)
cmd  (command)
```

All buses/networks are connected to the controller and all leaf level units.
In the caes of the tree network there may be intermediate nodes which are used to
gather information from a cluster of units and for timing reasons.

1. In band SOC bus
2a. Out of band network tree
2b. Out of band network Ring
3. In band pipeline

==

1. In band SOC bus
Each bus master waits for a slot on the bus and then sends a bus command to
another unit on the bus. All units "listen" to the bus for bus commands
addressed to the unit.

2a. Out of band network tree
The Out of band network tree has both a send and a receive network
The send network contains the following of signals:

```
addr - data
data - address
v    - valid
cmd  - command
```

```
The send network is used to send data and commands to the leaf level
units.
The leaf level units execute the commands and if requested send a reply
via the
reply tree to the controller.
The controller broadcasts messages to all units which match the uid in
the message and then
execute the command.

2b. Out of band network Ring
The Out of band network ring connects all units in a ring topology.
The ring contains the following of signals:
addr - data
data - address
v    - valid
cmd  - command

3. In band pipeline
Each pipestage can contain one or more registers which can be read/
written via packets sent down the
command pipeline. The command pipelne packets contain the following
signals.
addr - data
data - address
v    - valid
cmd  - command
When the address in the pipestage address signal matches an address in
the pipestage and the valid is
'1' then the command is exectued and a register is either read or writ-
ten.

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

// note that in the memory map b elow we do not set the data word width
or the address word
width. The clsc will determine the address word width based on the
address range (start and end
addresses) for the memory map.

csl_memory_map mem_map {
  csl_memory_map_page unit_a;
  mem_map () {
    set_type(VM_WITH_ADDRESS);
    unit_a.set_range(0, 65767);
  }
}

csl_enum bus_cmd {
  BUS_CMD_RD,
  BUS_CMD_WR,
  BUS_CMD_PING,
  BUS_CMD_NOP
};

// create a bus with signal names that match the pin names on the reg-
isters that the bus
// is logically connected to. There are intermediate units whiuch the
bus is connected to
// for timing and distribution reasons. The units that the bus is con-
nected to have a bus
// interface unit (BIU) that the bus is connected to. The BIU detects
commands that are
// intended for the unit and converts the commands into local control,
address, and data
//
//
//
//

csl_interface reply_bus {
  csl_port data(input,32),
           addr(input, mem_map.get_address_word_width()), // 9 bits
since log2(512) = 9
           v   (input  ) ;
};

csl_interface cmd_bus : reply_bus {
  csl port cmd(input,2);
  ifc(){
    cmd.add_enum(bus_cmd);
  }
};
```

```
csl_unit controller {
  cmd_bus bus_out;
  reply_bus bus_in;
  controller(){
    bus_out.reverse();
  }
};


csl_register_group unit_a_rg {
  csl_register r[[0-31]](32); // create 32 32-bit registers

  unit_a_rg() {

  }
};

csl_unit a{
  cmd_bus   bus_in;
  reply_bus bus_out;
  unit_a_rg unit_a_rg0;
  int unit_a_mem_map_base_addr;

  a() {
    unit_a_mem_map_base_addr = 2048;
    bus_out.reverse();
    mem_map.unit_a.add(unit_a_rg0, "unit_regs",
unit_a_mem_map_base_addr);
    unit_a_rg0.use_biu_to_write();

// unit_a_rg0 has the same interface as bus_in so they can be connected
// each register has a set of pins that match the signal names and
directions
// in the bus.
// however the bus_in and the bus_out are not directly connected to the
registers
// instead intermediate logic is created to write the registers.
//
// The cslc detects that each register in the register group unit_a_rg0
are in the memory
// map. Since all registers in the memmory map they need to be con-
nected to the the unit_a
// BIU (bus interface unit ) which listens to the bus as described
above and generates the
// write enable (wr_en) signals for each individual register.
//
// The interface bus_out is no directly connected to the register out-
puts. Instead the register
// outputs are connected to a mux and the bus_in_addr selects the reg-
ister to send back to the
// controller which sent the read command to unit_a.
//
// If not all registers are in the memory map generate a compiler
error.

    unit_a_rg0.connect(bus_in);
    bus_in.connect(unit_a_rg0);

    csl_signal a_en =
    reg_0.d = data;
    mem_map.set_unit_address_signal(a,addr);
  }
};


csl_unit top{
  a a0;
  controller cntl0;
  top(){
    a0.set_instance_id(3);

  }
};
```

OLD CSL CODE

```
csl_memory_map mem_map;

csl_enum bus_cmd {
  BUS_CMD_RD,
  BUS_CMD_WR,
  BUS_CMD_PING,
  BUS_CMD_NOP
};

csl_interface reply_bus {
  csl_port data(input,32),
           addr(input,5) ,
           v(input)      ;
};

csl_interface cmd_bus : reply_bus {
  csl port cmd(input,2);
  ifc(){
    cmd.add_enum(bus_cmd);
  }
};

csl_unit controller {
  cmd_bus bus_out;
  reply_bus bus_in;
  controller(){
    bus_out.reverse();
  }
};

csl_unit a{
  cmd_bus   bus_in;
  reply_bus bus_out;
  csl_register reg_0(32);
  a(){
   bus_out.reverse();
   reg_0.
   csl_signal a_en =
   reg_0.d = data;
  }
};
mem_map.add_logic(object_wr_en, address)
mem_map.add_object(a.reg_0)
mem_map.set_unit_address_signal(a,addr);
```

```
csl_unit top{
  a a0;
  top(){
    a0.set_instance_id();
  }
};
```

VERILOG CODE

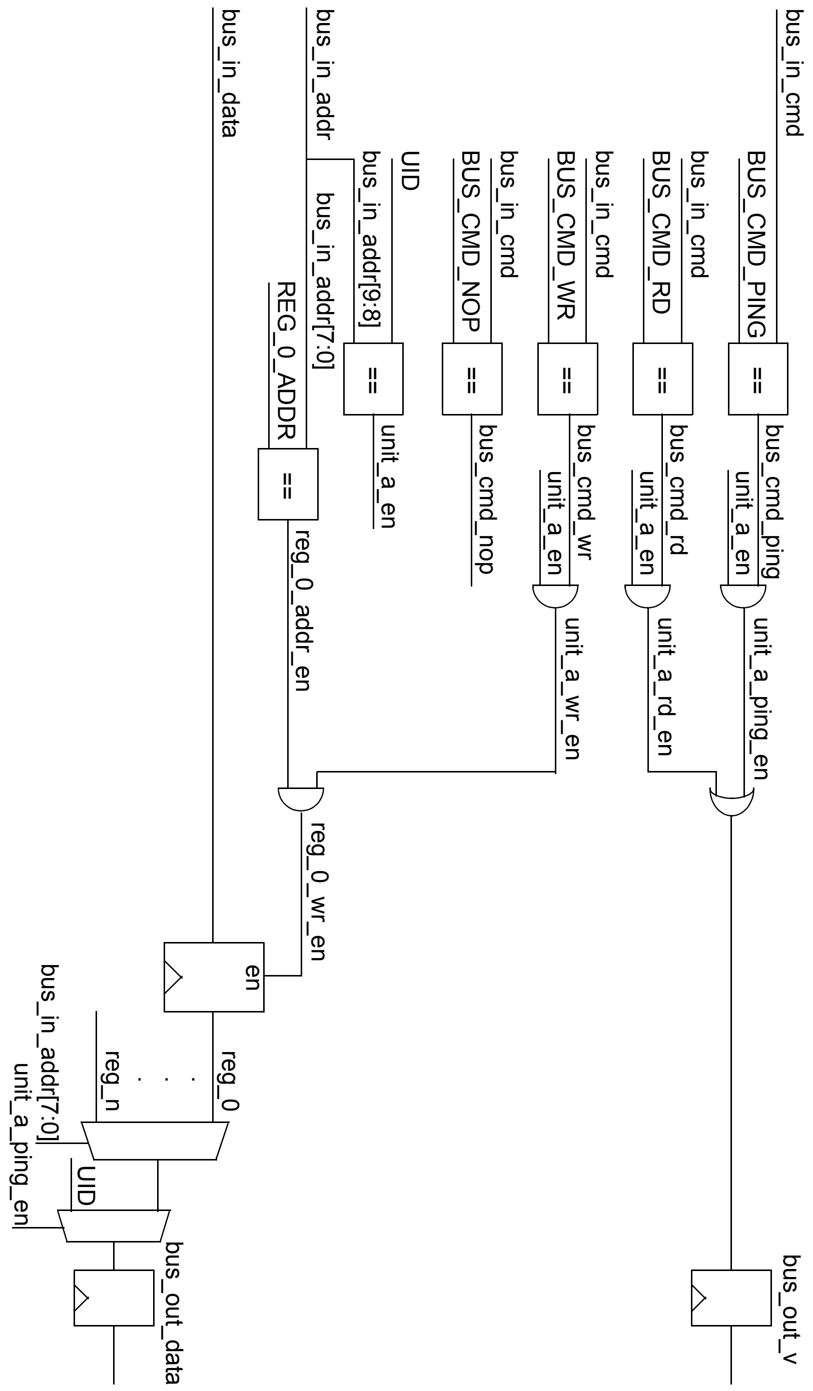
```
`define BUS_CMD_RD      0
`define BUS_CMD_WR      1
`define BUS_CMD_PING    2
`define BUS_CMD_NOP     3
`define UID             3 //unit id
`define REG_0_ADDR    128 //reg_0 address

module a(bus_in_data,
         bus_in_addr,
         bus_in_cmd ,
         clk,
         bus_out_data,
         bus_out_v
         );

 input [31:0] bus_in_data;
 input [9:0]  bus_in_addr;
 input [1:0]  bus_in_cmd;
 input clk;

 output bus_out_v;
 reg bus_out_v;
 output [31:0] bus_out_data;
 reg [31:0] bus_out_data;

 //local signals
 reg [31:0] reg_0;

 wire bus_cmd_rd   = (`BUS_CMD_RD   == bus_in_cmd)  ;
 wire bus_cmd_wr   = (`BUS_CMD_WR   == bus_in_cmd)  ;
 wire bus_cmd_ping = (`BUS_CMD_PING == bus_in_cmd);
 wire bus_cmd_nop  = (`BUS_CMD_NOP  == bus_in_cmd) ;
 wire unit_a_en     = bus_in_addr[9:8] == `UID;
 wire unit_a_wr_en  = unit_a_en && bus_cmd_wr;
 wire unit_a_rd_en  = unit_a_en && bus_cmd_rd;
 wire unit_a_ping_en= unit_a_en && bus_cmd_ping;

 wire reg_0_addr_en = bus_in_addr[7:0] == `REG_0_ADDR;
 wire reg_0_wr_en   = unit_a_wr_en && reg_0_addr_en;

 always @(posedge clk) begin
   if(reg_0_wr_en) begin
     reg_0 <= bus_in_data;
   end
 end

always @(posedge clk) begin
    bus_out_v <= unit_a_rd_en & unit_a_ping_en;
end


always @(posedge clk) begin
    if(unit_a_rd_en) begin
  case (bus_in_addr)
  `REG_0_ADDR: bus_out_data <= reg_0;
  endcase
  end
  else if(unit_a_ping_en) begin
      bus_out_data = `UID;
  end
end
endmodule
```

FIGURE 8.23 Bus Interface Unit Command Decoder

0

1

2


bus_in_cmd
BUS_CMD_PING
==
bus_cmd_ping
unit_a_en
unit_a_ping_en
bus_in_cmd
BUS_CMD_RD
==
bus_cmd_rd
unit_a_en
unit_a_rd_en
bus_in_cmd
BUS_CMD_WR
==
bus_cmd_wr
unit_a_en
unit_a_wr_en
bus_in_cmd
BUS_CMD_NOP
==
bus_cmd_nop
UID
bus_in_addr[9:8]
==
unit_a_en
bus_in_addr
bus_in_addr[7:0]
REG_0_ADDR
==
reg_0_addr_en
reg_0_wr_en
bus_in_data
en
reg_0
. . .
reg_n
bus_in_addr[7:0]
UID
unit_a_ping_en
bus_out_data
bus_out_v

</ADDED_2007.05.12>

<ADDED ON 2007.05.16>

NOTE:UPDATE COMMAND SUMMARY ACCORDING TO THIS

Note: There should be 8 examples from :

TABLE 8.12

	User defined mem map	automatic mem map
flat	x	x
hierarchical	x	x
virtual with page number and address	x	x
virtual with base address	x	x

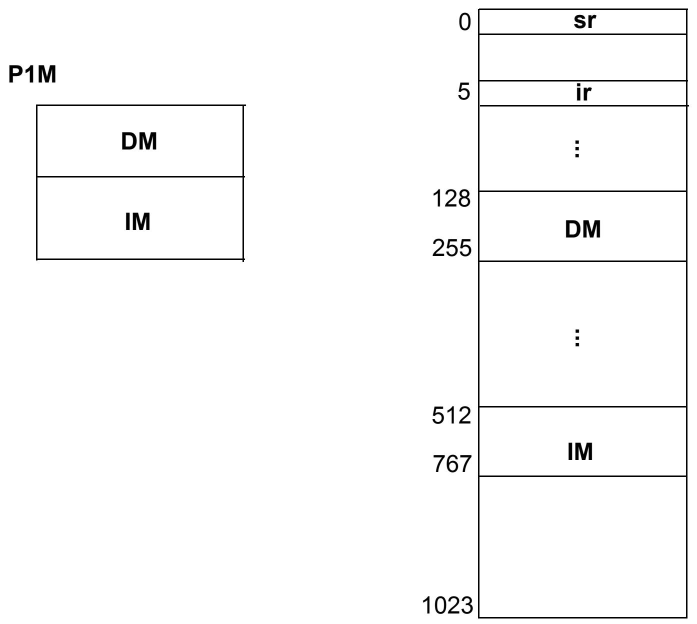
User defined example:

```
csl_unit processor {
   ...
};

csl_unit cluster {
  processor p[[0-7]];
};

csl_unit chip {
  cluster c[[0-7]];
};

csl_memory_map_page mproc {
  mproc(){
    set_unit(processor);
  }
};

csl_memory_map_page mcluster {
  mproc mp[[0-7]](p[[0-7]]); //user specified
  mcluster(){
    set_unit(cluster);
  }
};

csl_memory_map_page mchip {
  mcluster mc[0-7]](c[[0-7]]); //user specified
  mchip(){
    set_unit(chip);
  }
};

csl_memory_map mmap {
  mchip mchip;
  mmap(){
  set_type(hierarchical);
  }
}
```

Automatic example:

```
csl_unit processor {
   ...
};

csl_unit cluster {
  processor p[[0-7]];
};

csl_unit chip {
  cluster c[[0-7]];
};

csl_memory_map_page mproc {
  mproc(){
    set_unit(processor);
  }
};

csl_memory_map mmap {
   mmap(){
  set_top_unit(chip);
  set_type(hierarchical);
  use_instance_decl_order();
  }
}

//This generates the same code as the user defined version above
```

EXAMPLE :



```
csl_register sr;
csl_register ir;

csl_memory im(16,128);
csl_memory dm(16,256);

csl_unit p{
  im im();
  dm dm();

  sr sr();
  ir ir();
  p(){
     sr.add_to_mem_map();
     ir.add_to_mem_map(5);          // insert at address 5
     dm.add_to_mem_map(128);        // insert at address 128
```

```
        im.add_to_mem_map(512);       // insert at address 512
      }

    csl_memory_page mp{
      mp(){
        set_unit(p);
      }

    csl_memory_map mm{
      mp mp;
      mm(){
        set_top_unit(chip);
        set_type(hierarchical);
        autogen_mem_map;
      }
    }

    csl_unit cl{
      p p[[0-7]];
    }


    csl_unit chip{
      cl cl[[0-7]];
      chip(){
      }
    }
```

Generated header file:

```
    #define sr              0x000
    #define ir              0x005
    #define mp_start_addr   0x000
    #define mp_end_addr     0x3FF
    #define dm_start_addr   0x080
    #define dm_end_addr     0x0FF
    #define im_start_addr   0x200
    #define im_end_addr     0x2FF
```

When generating the memory map acces rights and visibility will be specified as parameters to generate only those defines that correspond to that specific options.

The adaptor needs to know how to connect the pins objects in the memory map to the network which will read/write the objects in the memory map.
Flat memory will need - data, address, command (W/R) and valid.
All units will listen to the address bus and they will need an address range checker (optional).

For hierarchical memory maps there will be a tree of enable signals to select the unit .
ex: chip enable + cluster enable + processor enable

Virtual with unit ID and address
The upper bits of the address bus will be used to identify the unit (unit ID), the lower bits will be used to address local memory in the selected unit.

Virtual memory with base address ?

Example Memory Map and SOC bus

FIGURE 8.24 Unit internal registers select logic

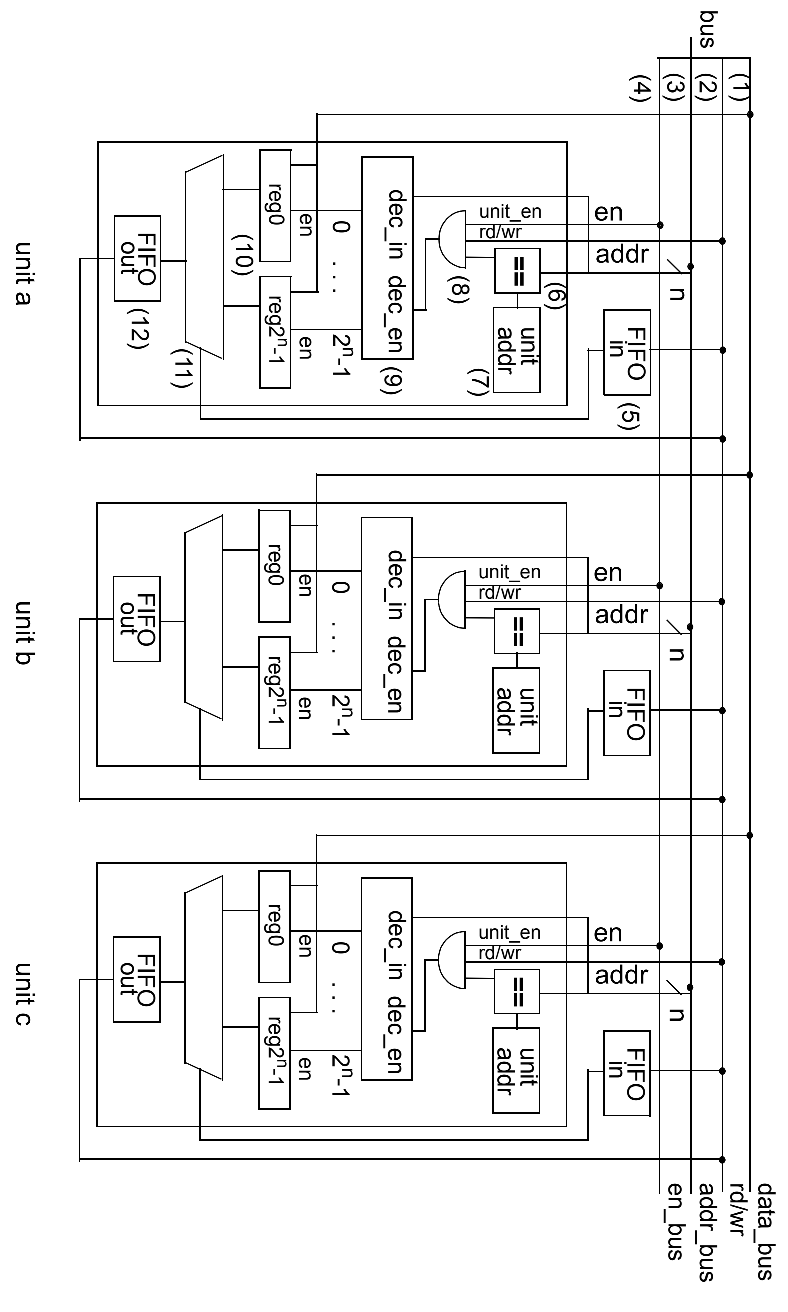
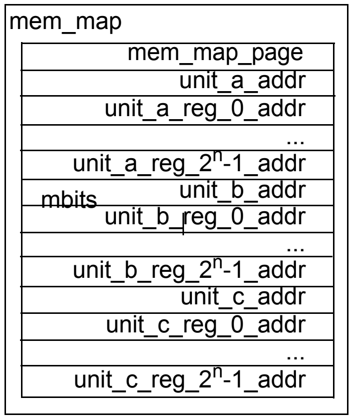

bus
(1)
(2)
(3)
(4)
unit a
unit b
unit c
en
addr
n
unit_en
rd/wr
==
unit addr
FIFO in
dec_in
dec_en
reg0
reg2^n-1
en
0
. . .
2^n-1
FIFO out
(5)
(6)
(7)
(8)
(9)
(10)
(11)
(12)
data_bus
rd/wr
addr_bus
en_bus

TABLE 8.13 Unit internal registers select logic elements

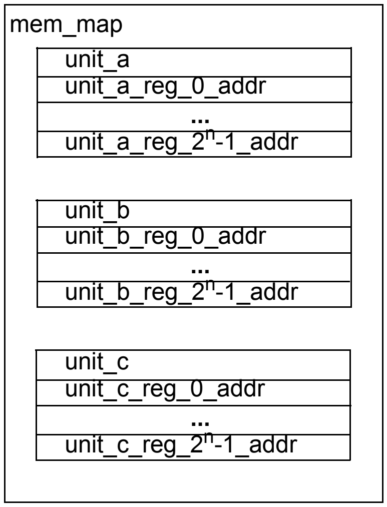
Nr.of components	Description
1	Data bus
2	rd/rw bus
3	Address bus (n bits)
4	Enable bus
5	FIFO in
6	Comparator
7	Unit address
8	And gate
9	Decoder
10	Register
11	Multiplexor
12	FIFO out

FIGURE 8.25 Flat Memory Map



- Send an address on the address bus;
- if the bus address equals the unit address and the unit enable is on, then enable the decoder;
- send another address on the address bus;

FIGURE 8.26 Hierarchical Memory Map



Same HW logic as flat.

Virtual Memory Map

Same SW structure as hierarchical

- send an address on the bus;

- only the first n bits are compared against the unit address;
- the last m bits are used for the decoder input;
- advantage: only one address is sent on the bus to select the register;
- disadvantage: for sequential access of registers from the same unit, bus space is wasted for the unit address (which, in this case, is redundant safe for the first access)