Describing a System Using Connection Specification Language (CSL)

(CHAIN®works 2.1.1)

License

Disclaimer

Table of Contents

DESCRIBING A SYSTEM USING CONNECTION SPECIFICATION LANGUAGE (CSL™) 5

Introduction 5
Include Target Technology Library 5
System-Level Description 6
Endpoints 7
Adapter Interface Protocol 7
Port and Endpoint Basics 9
`initiator` Declaration 10
`target` Declaration 10
Address and Data Widths 10
peak Bandwidth 11
Burst Size 11
`outstanding` Transactions 11
`response` Delays 12
Address Spaces 12
Declaring an Address Map and Target Ranges 12
Referencing an Address Map in an Initiator Declaration 12
Referencing an Address Range in a Target Declaration 13
Defining Connections between Endpoints 13
Modes of Operation 13
Different Directions, Different Requirements 13
Switching Latency 16
Network Roundtrip Latency 16
System Roundtrip Latency 17
Setting Optimization Priorities 17
Over-Provisioning and Dealing with Uncertainty 18
Area and Power Statements 18
`area` Statement 18
`power` Statement 19
Example CSL File 20
First Placement Estimator (FPE) Constructs 24
Describing Physical Attributes 25
`area` 25
`pad_ring` 26
`aspect_ratio` 26
`block_type` 27
`footprint` 28

endpoint_area_utilization 28
halo 28
elasticity_threshold 29
Controlling Block Placement and Floor Planning 30
Forming Relationships 30
pin 30
pad 30
net 31
Pin, Pad, Net Connection Examples 31
Relative Coordinates 34
Absolute Coordinates 35
Example CSL Design (with FPE constructs) 35
General Design Methodology 39
Naming/Identifier Conventions 40
Number Conventions 41
Revision History 42
Feedback 42
Disclaimers 42

Silistix

Describing a System Using Connection Specification Language (CSL™)

Introduction

This guide describes how to use the Silistix Connection Specification Language (CSL) to define communication paths between major hardware blocks in your system. The simple, top-down design example introduces key CSL language constructs and concepts along the way. The CSL syntax is relatively easy to learn, especially for those with any prior programming experience.

If you are already familiar with the general CSL language concepts, see General Design Methodology on page 39 for a convenient reference summary.

After describing your system in CSL, you can quickly evaluate your system, analyze design trade-offs, and accurately estimate area, performance, and power using the easy-to-use Silistix CHAINarchitect™ software.

Include Target Technology Library

A key component in the Silistix technology is a library of targeted, pre-designed, pre-characterized, and pre-verified hard macro functions that implement the proven CHAIN Network-on-Chip (NoC). These macros, packaged as a library, are specific to a semiconductor fabrication vendor and a process technology node.

Because these library functions are already mapped to the targeted process technology, the Silistix CHAINworks software provides predictable timing, power, and area estimates.

Table 1 lists the "generic" technology libraries available as of July 2008. Other libraries have been specifically developed for proprietary "captive" semiconductor facilities at major semiconductor and systems companies.

Table 1: **Silistix Target Technology Libraries (July 2008)**

Fabrication Vendor	Process Node	Process	Library Vendor/ IP Library	Library Name
—	90 nm	—	Silistix Generic	`Silistix_90nm_Generic`
TSMC	130 nm	130G	Artisan Sage	`TSMC130G_Artisan_Sage`
	90 nm	90G	Artisan Sage	`TSMC90G_Artisan_Sage`
	65 nm	65G	Artisan Advantage	`TSMC65G_Artisan_Advantage`
UMC	130 nm	130E	Artisan Metro	`UMC130E_Artisan_Metro`

System-Level Description

Figure 1 represents a top-level block diagram of a typical system-on-a-chip (SoC) application. The `system` consists of multiple clock domains, each highlighted in a different color. Each `domain` is controlled by an independent clock input, each operating at its own inherent frequency.

Embedded within each domain is one or more hardware block such as CPUs, DSPs, memory, etc. In the CSL language, each of these blocks represents a potential `endpoint`, when connected together at the chip level using the CHAIN network.

Figure 1: **System-level block diagram**

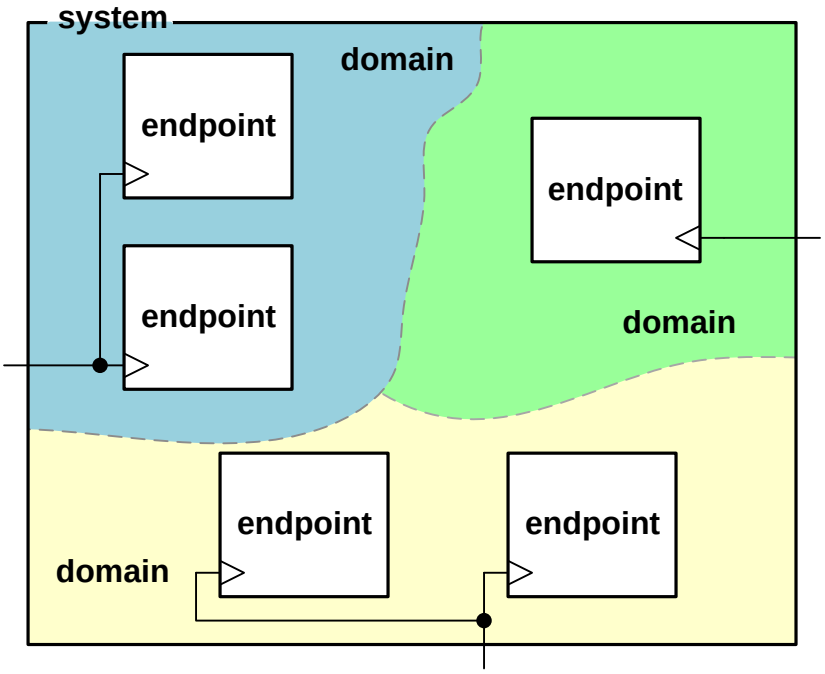


Figure 2 demonstrates these CSL system-level language concepts.

- The `library` statement declares that the target technology; in this case, the Silistix generic library. Table 1 lists the other available options.
- The `system` statement declares the name of the system.
- Each `domain` statement declares the name of the individual clock domain and specifies the native operating frequency in MHz.

Figure 2: **CSL Code Snippet Demonstrating `#include`, `system`, and `domain`**

```
/*************************************************************
**  CSL keywords and concepts demonstrated below:
**     library
**     system
**     domain
**     comments, both single- and multiple-line
*************************************************************/

// Target technology library
library Silistix_90nm_Generic ;

system my_system
{
     domain cpu_domain (400 MHz)
     {
          ... (see Figure 3)
     } // end cpu_domain

     domain memory_domain (133 MHz)
     {
          ...
     } // end memory_domain

     domain communications_domain (155.5 MHz)
     {
          ...
     } // communications_domain
}
```

Endpoints

An **`endpoint`** is a functional block within a clock domain that communicates to other endpoints in the system, either within the same clock domain or to endpoints in other clock domains. For each endpoint, describe the fundamental characteristics of the connection. Figure 3 provides an example.

Adapter Interface Protocol

Within a domain, each connection endpoint uses a local or native interface protocol, as illustrated in Figure 4. For example, a wide variety of IP functions use one of the ARM AMBA interface protocols (AXI, AHB, APB) while others might employ the Open Core Protocol (OCP).

The Silistix library contains adapter blocks that connect directly to the Silistix network. The IP block converses directly with other endpoints in the system using its native protocol. The adapter layer converts the protocol to Silistix network packets via a gateway.

Similarly, at other endpoints on the network, other adapters connect the Silistix network to other IP blocks, each supporting their own native protocol. In Figure 4, one IP block, using the ARM AXI bus protocol communicates via the Silistix network to another IP block that uses the Open Core Protocol (OCP).

The Silistix protocol adapter both connects an IP block to the Silistix network and provides protocol translation between endpoints on the network.

Figure 3: **Endpoint Code Snippet (expands concepts in Figure 2)**

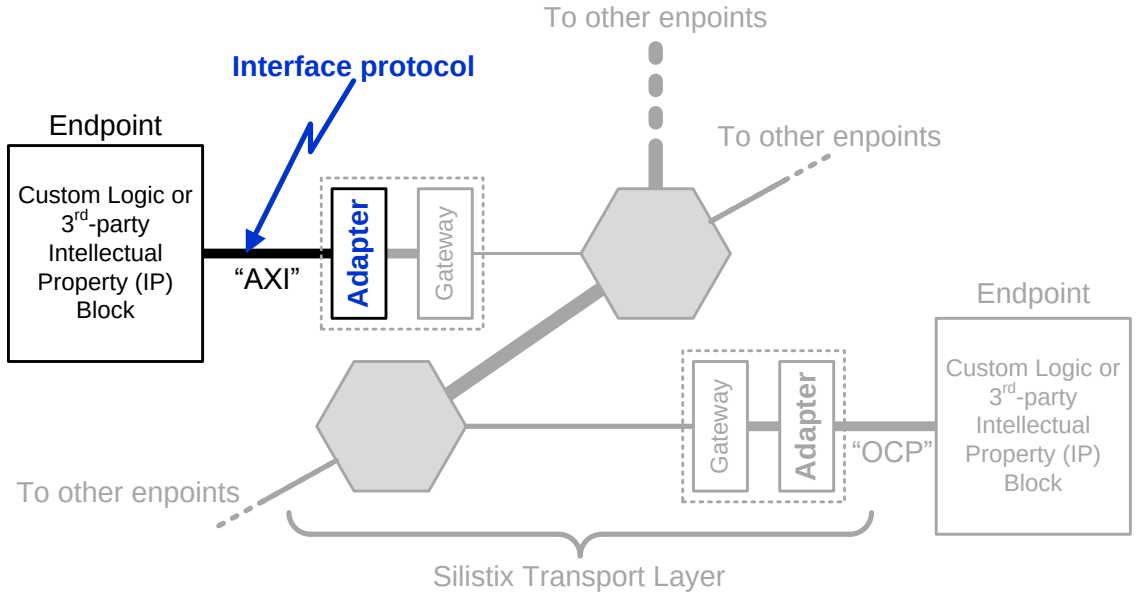
```
/*************************************************************
**  CSL keywords and concepts demonstrated below:
**      protocol, initiator, target
**      address, data, bits, bytes
**      peak, nominal, burstsize, MBs, Mbs, GBs, Gbs
**      address map, address range
**    ..outstanding, write response, read response
**      comments, both single- and multiple-line
*************************************************************/
domain cpu_domain (400 MHz) {  // from Figure 2
   CPU_endpoint {
      protocol = "AXI";  // See Table 2 for options

      initiator CPU_initiator {
         address = 32 bits;
         data = 32 bits;
         peak = 1600 MBs;  // only if less than theoretical
         burstsize = 32 bytes;
         address_map = CPU_address;
         outstanding = 8;
         write_response = 15 ns;
         read_response = 30 ns;
      } //end CPU_initiator

      target CPU_mailbox {
         address = 4 bits;
         data = 16 bits;
         burstsize = 16 bytes;
         address_range = COMM_address.CPU_mailbox;
      } // end CPU_mailbox target

   } // end CPU_endpoint
} // end domain
```

Figure 4: **Protocol Adaptor Connects an Endpoint to the Silistix Network**



For each endpoint, declare the native bus protocol used locally. The currently-supported protocols and adapters are listed in Table 2. Figure 3 provides a syntax example. In the example, the CPU endpoint includes both an initiator and a target. The endpoint uses an AMBA AHB interface protocol. The initiator sends addresses using its own address map. The CPU's target interface responds to addresses from a different endpoint.

Table 2: **Supported Adapter Interface Protocols**

Protocol Name	Description
CGP	Chain Gateway Protocol. Direct access onto the CHAIN network.
AHB	AMBA High-speed Bus
APB	AMBA Peripheral Bus. Only target endpoints are supported.
AXI	AMBA AXI Bus
OCP	Open Core Protocol

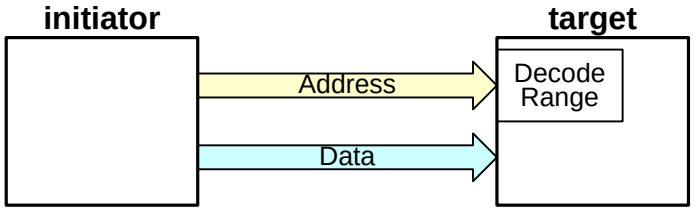
Port and Endpoint Basics

An endpoint supports one or more communication ports. Each port is an `initiator`, a `target`, or both. Figure 5 illustrates the role of both initiator and target within a transaction.

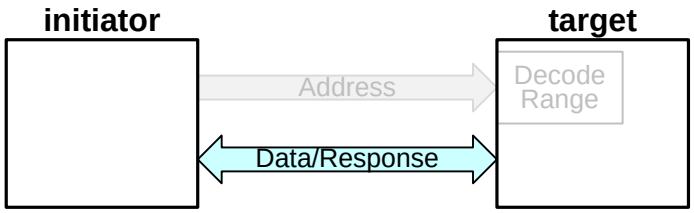
An `initiator`, as the name implies, begins and controls a transaction with a target endpoint. Typically, a transaction involves sending addresses or commands to the target endpoint. A transaction's command propagates from the initiator to the target over the ***command path***.

A `target` responds to the transaction by decoding the address and command presented by the initiator and then either sends data back to the initiator or accepts data from the initiator. The communication transfer from the target back to the initiator, over the ***response path***, completes the transaction. The target responds to an initiator's command either by returning read data or acknowledging a write operation.

Figure 5: **Initiator and Target in a Transaction**



a) An `initiator` begins a transaction by sending a command on the Address and Data bus.



b) If the transaction is to its decoded address range, the `target` responds by sending or accepting Data to/from the initiator.

`initiator` Declaration

An initiator declaration, depending on the adaptor protocol, may use the following fields. Specific protocol adapters support additional options.

- A name or identifier
- The **`address`** port width
- The **`data`** port width
- The **`address_map`** that the initiator uses to reference the attached targets
- Optionally, if peak transfer rate supported by the IP port is lower than the theoretical peak bandwidth (frequency × data width), then specify the port's **`peak`** transfer rate.
- Optionally, the maximum data **`burstsize`**
- Optionally, the maximum number of **`outstanding`** transactions allowed between operations
- Optionally, the **`response`** delay of the initiator between transactions
- Specify any protocol-specific attributes required for the interface. Consult the ***CHAIN Network Adapter User Guide*** for more information.

`target` Declaration

- A target declaration, depending on the adaptor protocol, may use the following fields. Specific protocol adapters support additional options.
- A name or identifier
- The **`address`** port width
- The **`data`** port width
- The address **`address_range`** within an address map, to which the target responds
- Optionally, if peak transfer rate supported by the IP interface is lower than the theoretical peak bandwidth (frequency × data width), then specify the interface **`peak`** transfer rate.
- Optionally, the maximum data **`burstsize`**
- Optionally, the **`response`** time of the target to an initiator command
- Specify any protocol-specific attributes required for the interface. Consult the ***CHAIN Network Adapter User Guide*** for more information.

Address and Data Widths

For each endpoint, define the width of the **`address`** port and the **`data`** port. This requirement applies to both initiators and targets.

Define the **`address`** and **`data`** port width as **`bits`**. Figure 3 provides an example. The address port requires enough bits to cover the specified address range, not the maximum address range supported by IP block contained within the endpoint.

The data port width is the maximum width of any individual data transfer.

Depending on the interface protocol used in the endpoint, there may be additional limitations on the data and address width. For example, the data width for an AMBA APB interface must be 8, 16, or 32 bits wide. Similarly, the APB address bus width must be a multiple of 8. The CHAINarchitect

software reports any mismatch between the requested address or data width and those supported by the interface protocol.

peak Bandwidth

The optional **peak** bandwidth for an endpoint is specified using a number of bits or bytes per second. Table 3 lists the units required when specified bandwidth. If the **peak** bandwidth is not specified, then the CSL compiler calculates the peak bandwidth from the connections to the port (frequency × data width). Consequently, the peak transfer rate need only be specified if the endpoint is not capable of carrying the theoretical peak bandwidth to and from the port (frequency × data width).

The bandwidth units are case sensitive. Lower-case '**b**' represents bit; upper-case '**B**' represents byte.

If the peak transfer rate for the endpoint port is specified in the CSL, CHAINarchitect ensures that the combined network transfer rate is less than the specified limit. If the calculated network traffic is greater than the specified peak transfer rate, then CHAINarchitect issues an error.

Table 3: **CSL Bandwidth Units**

Unit	Description
Mbs	Megabits per second. 1M = 1024K = 1,048,576 bits per second.
MBs	MegaBytes per second. 1M = 1024K = 1,048,576 bytes per second.
Gbs	Gigabits per second. 1G = 1024M = 1,048,576K = 1,073,741,824 bits per second.
GBs	GigaBytes per second. 1G = 1024M = 1,048,576K = 1,073,741,824 bytes per second.

Burst Size

The **burstsize** specifies the size of the longest transaction to be supported by the endpoint, measured in either **bits** or **bytes**.

outstanding Transactions

The **outstanding** declaration defines the maximum number of transactions that can be in flight between operations. Essentially, this defines how many transactions can be in the network before requiring acknowledgement. By default, this set to 1. The AHB Adapter Interface Protocol, for example, only supports a single outstanding transaction. Consult the adapter documentation for additional information.

response Delays

The **response** declaration defines how the initiator or target endpoint responds to read and write transactions. The CSL Compiler software uses the defined response value to calculate the system roundtrip latency and bandwidth.

The definitions for write and read responses vary between initiator and target, as shown in Table 4.

Table 4: **Write and Read Response Definitions for Initiator and Target Endpoints**

Declaration	Initiator	Target
write_response	Turnaround time between write transactions	The delay to return a write response
read_response	Turnaround time between read transactions	The delay to return read data

Address Spaces

Each initiator has a defined **address_map**. The address map describes the address decoding to the various targets connected to the initiator. Each decoded target address is specified as an **address_range** within the address map, as shown in Figure 6.

Each address range has a unique name within the address map, plus a starting and ending byte address. Ranges within an address map cannot overlap, although a target can respond to multiple address ranges, as shown in Figure 7.

Declaring an Address Map and Target Ranges

The address map and range must be declared before it is referenced in an endpoint declaration, as shown in Figure 6.

Figure 6: **CSL Code Snippet Demonstrating address map and range**

```
/*********************************************************
**  CSL keywords and concepts demonstrated below:
**    address map
**    address range
*********************************************************/
system my_system
{
  // address map for the CPU (see Figure 14)
  address_map CPU_address
  {
     range comm_mailbox    0x0000000 .. 0x00001ff;
     range external_sdram  0x1000000 .. 0x1ffffff;
  }
  domain cpu_domain (400 MHz)
     {
       ...
```

Referencing an Address Map in an Initiator Declaration

An **address_map** is required for each initiator. Essentially, the address map represents the initiator's view of the system. All initiators in a system can share a single, common address map or each initiator can have its own independent, locally-referenced address map. However, an initiator can only have one address map.

For an example, see the **address_map** declaration for the CPU_endpoint initiator in Figure 3.

Referencing an Address Range in a Target Declaration

As highlighted in Figure 7, a target can respond to a single address range, multiple address ranges, and even multiple address ranges from different address maps. If declaring multiple target address ranges, enclose the comma-separated list with braces or curly brackets ({, }). Each **address_range** reference has the following form.

<address_map_name>.<address_range_name>

Figure 7: **Example Target Address Map Reference**

```
// example of single target address range
          target CPU_mailbox {
              ...
              address_range = COMM_address.CPU_mailbox;
          } // end CPU_mailbox target definition

// example of multiple target address ranges
          target SDRAM_target {
              ...
              address_range = {CPU_address.external_sdram,
                               COMM_address.external_sdram};
          } // end SDRAM_target definition
```

Defining Connections between Endpoints

After defining the system, clock domains, endpoints, and initiator and target endpoints, then specify the connections between initiators and targets. To specify the connections, answer the following questions about your system.

- Which initiators and targets actually communicate in the system?
- How do the communications requirements change during different modes of operation?
- For each initiator-target pair, what are the roundtrip bandwidth and latency requirements when the initiator sends traffic to the target (initiator → target)?
- For each initiator-target pair, what are the bandwidth and latency requirements when the initiator receives traffic from the target (initiator ← target)?

Modes of Operation

Virtually all SoCs have inherent modes of operation. For example, many SoCs have a reset mode, a low power mode, and a high performance mode. Some designs may have dozens of operating modes. Within a CSL file, use a separate **mode** statement to specify the bandwidth and latency requirements for each exclusive operating mode. CHAINarchitect and CSL Compiler leverage this information to create a properly-provisioned network.

Figure 8 shows an example of how to declare a mode and the connections within each mode.

Different Directions, Different Requirements

The bandwidth and latency requirements for a connection are typically directional.

In CSL descriptions, the path from initiator → target is typically a completely separate network path than the initiator ← target path. This split network architecture offers several advantages; each path has different topologies and consequently different bandwidth and latency characteristics.

- The initiator → target command path need not be idle while waiting on the response of a previous transaction.

- The topology of the two networks can be very different, each optimized for its bandwidth and latency requirements. Typically, the initiator → target path and the initiator ← target paths are very different from one another. Optimizing each separately often saves silicon area, power, and likely wire congestion as well.

The CSL language uses five connection operators, shown in Table 5. Two operators define roundtrip network paths. The remaining three operators specify unidirectional connections. The source and destination of the connection are specified in the form <*endpoint_name*>.<*initiator_name*> and <*endpoint_name*>.<*target_name*>.

Table 5: **CSL Connection Operators**

Connection Type	Connection Operator	Specified Path	Function
Roundtrip	**=>**	Figure 10	Specifies roundtrip connection from an initiator, to a target, and write or read response from the target back to the initiator, including the write or read response delay of the target.
	<=		Specifies roundtrip connection from a target, to an initiator, and write or read response from the target back to the initiator, any specified separation between initiator transactions.
Uni-directional *(not recommended)*	**->**	Figure 9	Specifies connection from an initiator to a target
	<-		Specifies connection from a target back to an initiator
	<->		Specifies symmetric connections between initiator and target.

Roundtrip Connection Operators

In most applications, connections are specified using System Roundtrip Latency and bandwidth.

The **=>** operator specifies roundtrip connections originating from the initiator to the target, including target response time. This operation is typically a write operation. The **<=** operator specifies roundtrip connections from the target to the initiator, including any initiator delays between transactions. This operation is typically a read operation.

Unidirectional Connection Operators

Unidirectional connections are also allowed, but are not recommended except for specific advanced application cases. A unidirectional connection operator specifies the Switching Latency and bandwidth in a single direction between the two endpoints.

A third operator, not generally recommended, defines symmetric connections between the initiator and the target.

Figure 8: **Mode and Connections Code Snippet (expands concepts in Figure 2 and Figure 3)**

```
/*******************************************************
**  CSL keywords and concepts demonstrated below:
**    ..mode statement
**      endpoint.initiator and endpoint.target syntax
**      connections from initiator to target =>)
**      connections back to initiator from target (<=)
**      bandwidth and latency specifications
*******************************************************/
domain cpu_domain (400 MHz) {
      ...
}

mode high_speed {
   // Define path from CPU to SDRAM
   CPU.CPU_initiator => SDRAM.SDRAM_target
      (bandwidth=133 MBs, latency=90 ns);

   // Define path to CPU from SDRAM
   CPU.CPU_initiator <= SDRAM.SDRAM_target
      (bandwidth=333 MBs);
      ...
}  // end high_speed mode

mode low_power {
   // Define path from CPU to SDRAM
   CPU.CPU_initiator => SDRAM.SDRAM_target
      (bandwidth=10 MBs);

   // Define path to CPU from SDRAM
   CPU.CPU_initiator <= SDRAM.SDRAM_target
      (bandwidth=10 MBs);
      ...
}  // end low_power mode
```

CSL Example: Roundtrip Connections

In the example system illustrated in Figure 14, the CPU fetches code from an external SDRAM memory. Consequently, the CPU ← SDRAM path has significantly higher bandwidth requirements than the CPU → SDRAM path. Figure 8 provides a code snippet demonstrating how these connections are specified.

In CSL, bandwidth has the usual definition – the amount of data transferred over a defined time period. The CSL bandwidth units are listed in Table 3 on page 11.

Latency is a critical architectural parameter that can be very difficult to manage with legacy bus architectures. In the CSL description, the specified latency is the maximum acceptable System Roundtrip Latency. The system roundtrip latency includes the Silistix Network Roundtrip Latency plus any specified **response** Delays. The CSL connection specification defines the maximum acceptable system roundtrip latency. The CHAINarchitect software reports both the network and system roundtrip latency values as applicable.

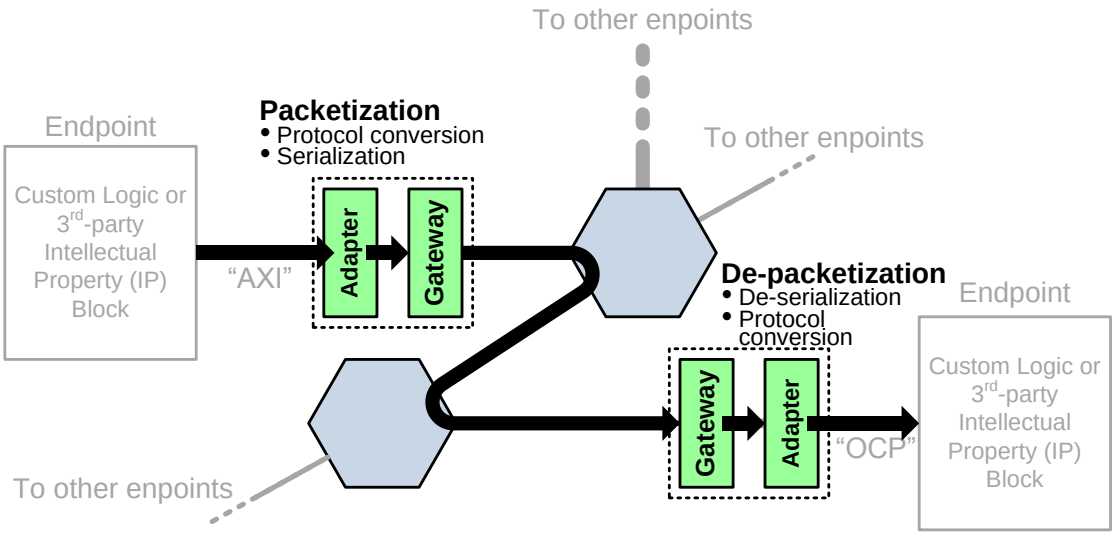
Switching Latency

As shown in Figure 9, ***switching latency*** is the total network flight time in a single direction. Switching latency is a fundamental component of the Network Roundtrip Latency and the System Roundtrip Latency. The switching latency includes the time to …

- convert from the interface protocol of the initiator IP block
- serialize the packet
- transport the package over the Silistix network
- de-serialize the packet
- convert the packet to the interface protocol used on the target IP block.

The switching latency depends on the traffic direction; it is not the total roundtrip time.

Figure 9: **Switching Latency is Total Network Flight Time in a Direction**



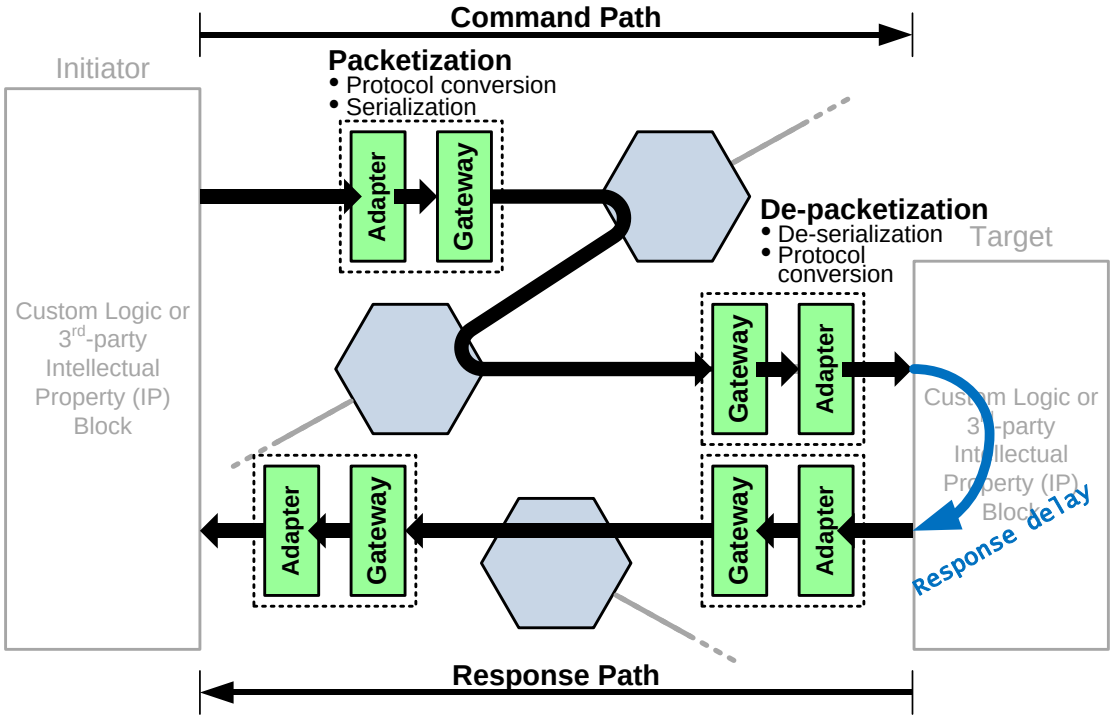
Network Roundtrip Latency

Network roundtrip latency is the sum of the command path Switching Latency and the response path Switching Latency over the Silistix network, as shown in Figure 10. Because the connection from the initiator to the target is specified separately from the connection from the target back to the initiator, both paths are optimized independently and may have very different network paths.

The network roundtrip latency is the total of the command and response paths, but does not include the unpredictable delays incurred within the initiator or target IP. The network roundtrip latency is highly predictable using the Silistix CHAINarchitect software. The resulting Silistix network is pre-implemented, with portions built using pre-characterized hard IP blocks. Consequently, the Silistix CHAINarchitect software provides highly-accurate timing information.

However, these IP delays can be specified by defining the `write_response` and `read_response` delays for the initiator and target ports.

Figure 10: **Network and System Roundtrip Latency**



System Roundtrip Latency

As shown in Figure 10, system roundtrip latency includes the Network Roundtrip Latency plus any specified `write_response` or `read_response` delay. The CHAINarchitect software easily predicts the Silistix network roundtrip latency. However, any delays incurred within the initiator or target IP blocks must be specified in the CSL file because CHAINarchitect cannot predict these delays.

Setting Optimization Priorities

CSL allows you to set relative latency, power, and area priorities for later processing by the CHAINworks software tools. This capability helps achieve a satisfactory balance between these often-conflicting design requirements. As shown in Figure 11, set the optimization level to '1' for the highest-priority requirement and '3' for the lowest-priority requirement.

Optimization can be set for the entire system or for specific connections.

Figure 11: **CSL Code Snippet Demonstrating `optimize`**

```
/*************************************************************
**  CSL keywords and concepts demonstrated below:
**    optimize.latency, area, and power syntax
*************************************************************/
  // set optimization priority (1=highest, 3=lowest)
  optimize_area=1;
  optimize_latency=2;
  optimize_power=3;
```

Over-Provisioning and Dealing with Uncertainty

In some systems, it may be difficult to accurately specify bandwidth and latency requirements. The CSL language allows you to easily over provision requirements using a `utilization_threshold` setting.

By default, threshold is set to 1, meaning that the CHAINworks software will build networks that can approach the full specified limits. For example, if a particular latency setting is set to 7 ns, then CHAINworks will build connections with latency right up to the limit. Setting threshold to a lower number over-provisions the actual bandwidth and latency limits as shown in Equation 1 and Equation 2. For example, setting `utilization_threshold = 0.8` reduces a specified 7 ns latency to 0.8 × 7 ns = 5.6 ns.

Equation 1

$$Effective\ Bandwidth = \frac{Specified\ Bandwidth}{\texttt{utilization\_threshold}}$$

Equation 2

$$Effective\ Latency = Specified\ Latency \times \texttt{utilization\_threshold}$$

The CSL code snippet in Figure 12 shows how to set the `utilization_threshold`.

Figure 12: **CSL Code Snippet Demonstrating `threshold`**

```
/*************************************************************
**  CSL keywords and concepts demonstrated below:
**    utilization_threshold
*************************************************************/
  utilization_threshold = 0.6;      // low confidence in limits
  CPU.CPU_initiator -> SDRAM.SDRAM_target
        (bandwidth=266 MBs, latency=8.5 ns);
    ...
  utilization_threshold = 1.0;      // high confidence in limits
  COMM.communications_initiator -> CPU.CPU_mailbox
       (bandwidth=2 MBs, latency=50 ns);
```

An alternate interpretation of the threshold specification is to view it as a confidence level. If you are confident of the system's bandwidth and latency limits, set **`utilization_threshold`** high (0.8 to 1.0). If confidence is lower, set **`utilization_threshold`** to 0.6 to 0.8.

The **`utilization_threshold`** can be set for the entire system or for specific connections.

Area and Power Statements

The **`area`** and **`power`** statements provide the total area and power budgets provided in the system design. These values are used by the Silistix CHAINarchitect and CSL Compiler software to report percentages of total die area utilized by the interconnect logic and the miniscule amount of additional power required to implement the CHAIN network.

`area` Statement

Use the **`area`** statement to specify the total die area target of your design or for an individual endpoint. It is often more accurate to represent area at the endpoint level whenever possible. CHAINarchitect and the CSL Compiler use this value to determine the percentage of total die area utilized by the interconnect logic. It is also used for floor plan estimation purposes. If the **`area`** statement is specified, then the value given takes precedence over any area values set for endpoints.

See the area section on page 25 on how to use this command with the First Placement Estimator (FPE) tool.

Figure 13 provides an example of how to use the **area** statement to specify total silicon area occupied by the system, including the area occupied by the CHAIN network.

Figure 13: **CSL Code Snippet Demonstrating area and power Statements**

```
/*************************************************************
**  CSL keywords and concepts demonstrated below:
**    area and power syntax
*************************************************************/
  area = 48.8 mm2;    // Total silicon area, including CHAIN network
  power = 250.0 mW;   // Total system power, without CHAIN network
```

The area statement has two possible unit values, as shown in Table 6. The specified technology library contains the conversion information between these two unit values. See **endpoint_area_utilization** on page 28 for more information on how these two unit systems interrelate.

Table 6: **Area Statement Units**

Unit	Description
mm2	Square millimeters of silicon.
kgates	Thousands of gates.

power Statement

Use the **power** statement to specify the total power consumed by your design or for a specific portion of the system. CHAINarchitect and the CSL Compiler use this value is used to report the total system power, including power consumed by the interconnect logic. If the **power** statement is specified, then the value given takes precedence over any power values set for endpoints.

Figure 13 above provides an example of how to use the **power** statement to specify total system power consumed by the system, excluding the power consumed by the CHAIN network.

The power statement supports the unit values shown in Table 7. The "power per megahertz" units are primarily used to specify sub-domains within the system.

Table 7: **Power Statement Units**

Unit	Description
uW	microwatts
mW	milliwatts
Watts	Watts
uWpMHz	microwatts per megahertz using the clock frequency defined for the associated domain
mWpMHz	milliwatts per megahertz using the clock frequency defined for the associated domain
WattpMHz	Watts per megahertz using the clock frequency defined for the associated domain

It is typically more accurate to represent power at the endpoint level whenever possible.

Example CSL File

Figure 14 presents a top-level block diagram of an example system. The system includes a CPU, a communications controller, and a shared SDRAM memory serving both the CPU and communications controller. The CPU and the communications controller swap data via mailbox registers.

Figure 14: **Block Diagram of Example System**

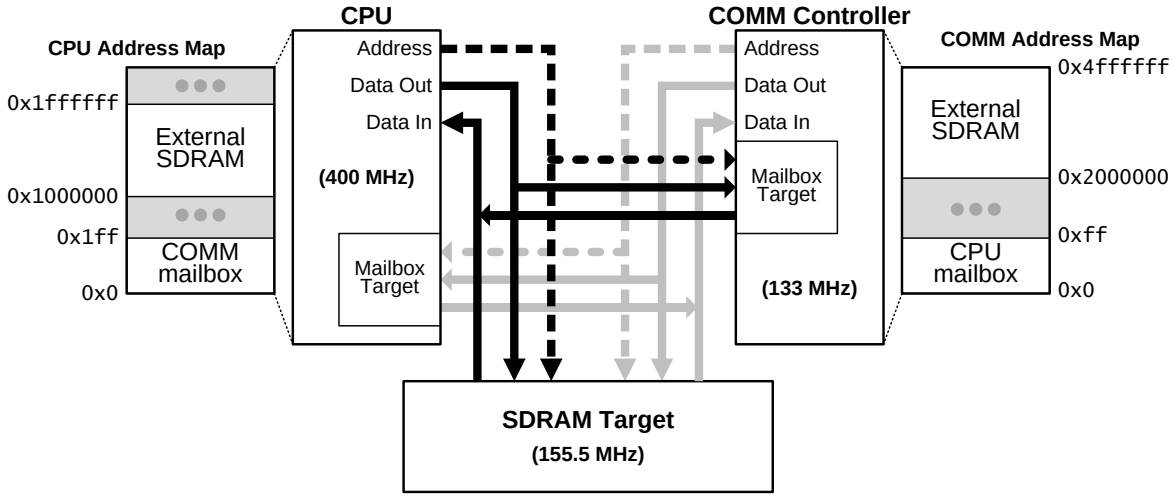


Figure 15 provides a complete CSL description for the example system shown in Figure 14.

Figure 15: **Complete Example CSL File**

```
// Target technology library
library Silistix_90nm_Generic ;

system my_system {

  // set optimization priority (1=highest, 3=lowest)
  optimize_area=1;
  optimize_latency=2;
  optimize_power=3;

  area = 48.8 mm2;    // Total silicon area, including CHAIN network
  power = 250.0 mW;   // Total system power, without CHAIN network

  // set threshold on how close the actual values approach the
  //    specified limits
  // (alternate view):  what is the confidence in the
  //     specified limits?
  utilization_threshold = 90% ;

  // address map for the CPU
  address_map CPU_address {
     range comm_mailbox    0x0000000 .. 0x00001ff;
     range external_sdram  0x1000000 .. 0x4fffffff;
  }
```

```
// address map for the COMM controller
address_map COMM_address {
   range CPU_mailbox     0x0000000 .. 0x00000ff;
   range external_sdram  0x2000000 .. 0x4ffffff;
}

// define each of the various clock domains and endpoints
domain cpu_domain (400 MHz) {
   CPU {
        protocol = "AXI";  // "AHB", "APB", "AXI", "OCP"

        initiator CPU_initiator {
           address = 32 bits ;
           data = 32 bits ;
           peak = 1600 MBs ;
           burstsize = 32 bytes ;
           address_map = CPU_address ;
           outstanding = 8;
           write_response = 2.5 ns ;
           read_response = 2.5 ns ;
           // AXI protocol-specific options
           axi_id_bits = 4 ;
        } // end CPU_initiator

        target CPU_mailbox {
           address = 8 bits ;
           data = 16 bits ;
           burstsize = 16 bytes ;
           address_range = COMM_address.CPU_mailbox ;
           // AXI protocol-specific options
           axi_id_bits = 1 ;
           axi_write_interleave_depth = 1 ;
        } // end CPU_mailbox target
  } // end CPU endpoint
} // end cpu_domain

domain memory_domain (333 MHz) {
  SDRAM {
        protocol = "AHB"; // "AHB", "APB", "AXI", "OCP"

        target SDRAM_target {
           address = 32 bits ;
           data = 32 bits ;
           burstsize = 128 bytes ;
           address_range = {CPU_address.external_sdram,
                            COMM_address.external_sdram} ;
           write_response = 25 ns ;
           read_response = 50 ns ;
         } // end SDRAM_target
  } // end SDRAM endpoint
} // end memory_domain
```

```
domain communications_domain (180 MHz) {
   COMM {
        protocol = "AHB"; // "AHB", "APB", "AXI", "OCP"

        initiator communications_initiator {
           address = 32 bits ;
           data = 32 bits ;
           burstsize = 32 bytes ;
           address_map = COMM_address ;
           ahb_version = lite ;
        } // end communications_initiator

        target comm_mailbox {
           address = 16 bits ;
           data =16 bits ;
           burstsize = 64 bytes ;
           address_range = CPU_address.comm_mailbox ;
           write_response = 10 ns ;
           read_response = 5 ns ;
           ahb_version = lite ;
        } // end comm_mailbox
  } // end COMM endpoint
} // end communications_domain

// Define operating modes and connections between endpoints

utilization_threshold = 75% ;  // lower confidence in limits

mode high_speed {
  // Connections between CPU and SDRAM
  CPU.CPU_initiator => SDRAM.SDRAM_target
    (bandwidth=133 MBs, latency=90 ns) ;

  CPU.CPU_initiator <= SDRAM.SDRAM_target
     (bandwidth=333 MBs) ;

  // Connections between COMM controller and SDRAM
  COMM.communications_initiator => SDRAM.SDRAM_target
     (bandwidth=50 MBs) ;

  COMM.communications_initiator <= SDRAM.SDRAM_target
     (bandwidth=133 MBs) ;

  // Connection between CPU and COMM controller mailbox
  CPU.CPU_initiator => COMM.comm_mailbox
     (bandwidth=0 MBs) ;

  CPU.CPU_initiator <= COMM.comm_mailbox
     (bandwidth=0 MBs) ;

  // Connection between COMM and CPU mailbox
  COMM.communications_initiator => CPU.CPU_mailbox
     (bandwidth=0 MBs);

  COMM.communications_initiator <= CPU.CPU_mailbox
     (bandwidth=0 MBs);
}  // end high_speed mode
```

```
utilization_threshold = 85% ;  // confidence greater for
                               // following modes

mode low_power {
  // Connections between CPU and SDRAM
  CPU.CPU_initiator => SDRAM.SDRAM_target
    (bandwidth=10 MBs) ;

  CPU.CPU_initiator <= SDRAM.SDRAM_target
     (bandwidth=10 MBs) ;

  // Connections between COMM controller and SDRAM
  COMM.communications_initiator => SDRAM.SDRAM_target
     (bandwidth=0 MBs) ;

  COMM.communications_initiator <= SDRAM.SDRAM_target
     (bandwidth=0 MBs) ;

  // Connection between CPU and COMM controller mailbox
  CPU.CPU_initiator => COMM.comm_mailbox
     (bandwidth=0 MBs) ;

  CPU.CPU_initiator <= COMM.comm_mailbox
     (bandwidth=0 MBs) ;

  // Connection between COMM and CPU mailbox
  COMM.communications_initiator => CPU.CPU_mailbox
     (bandwidth=0 MBs);

  COMM.communications_initiator <= CPU.CPU_mailbox
     (bandwidth=0 MBs);

} // end low_power mode

// Connection between CPU and COMM controller mailbox
utilization_threshold = 95%;     // high confidence in limits

mode reset {
  // Connections between CPU and SDRAM
  CPU.CPU_initiator => SDRAM.SDRAM_target
    (bandwidth=33 MBs) ;

  CPU.CPU_initiator <= SDRAM.SDRAM_target
     (bandwidth=33 MBs) ;

  // Connections between COMM controller and SDRAM
  COMM.communications_initiator => SDRAM.SDRAM_target
     (bandwidth=0 MBs) ;

  COMM.communications_initiator <= SDRAM.SDRAM_target
     (bandwidth=0 MBs) ;
```

```
    optimize_area=1;
    optimize_power=2;
    optimize_latency=3;

    // Connection between CPU and COMM controller mailbox
    CPU.CPU_initiator => COMM.comm_mailbox
       (bandwidth=5 MBs) ;

    CPU.CPU_initiator <= COMM.comm_mailbox
       (bandwidth=5 MBs) ;

    // Connection between COMM and CPU mailbox
    COMM.communications_initiator => CPU.CPU_mailbox
       (bandwidth=2 MBs, latency=150 ns);

    COMM.communications_initiator <= CPU.CPU_mailbox
       (bandwidth=2 MBs, latency=110 ns);

  } // end reset mode

  // Add "--define:INCLUDE_SCSL" to CSL Compiler options to
  // include structural CSL file during NPV validation
    #if defined(INCLUDE_SCSL)
    #include "struct_csl.csl"
    #endif

} // end system
```

First Placement Estimator (FPE) Constructs

The previous sections described how to construct an example system using a Silistix network. All the CSL constructs used so far describe various communication and connectivity aspects of the design. The following sections use additional CSL constructs to describe the physical attributes of the design. These constructs are necessary to use the First Placement Estimator (FPE) tool, which is part of CHAINarchitect.

These additional CSL constructs perform one of the two functions, as further described in the sections below.

- Describing Physical Attributes
- Controlling Block Placement and Floor Planning

With these language constructs and the FPE tool, the CHAINarchitect software generates a more accurate model of the entire system, including any placement-induced affects. For example, without FPE, the CHAINarchitect software is unaware of actual physical placement and evaluates network performance assuming that all the network components are physically placed at their maximum distance or hop length. However, due to real physical placement constraints, some of the network connections may exceed this maximum hop length. With FPE, the CHAINarchitect software automatically inserts the required number of “pipelatch” components to re-buffer network connections, shortening the hop length between components. These pipelatch components maintain bandwidth regardless of physical separation while introducing only minor increases in latency for the connection. With FPE, the CHAINarchitect software generates a report file that accounts for these placement-induced effects, including the pipelatch components. The FPE tool also generates an initial placement of the design that is used with synthesis or physical placement. Each initial placement can be evaluated and optimized by adjusting the CSL description and changing various options available in CHAINarchitect.

For additional information on running the First Placement Estimator (FPE) tool, see the "Generate First Placement Estimation" section in ***Building and Analyzing On-Chip Networks using CHAINarchitect*** user guide.

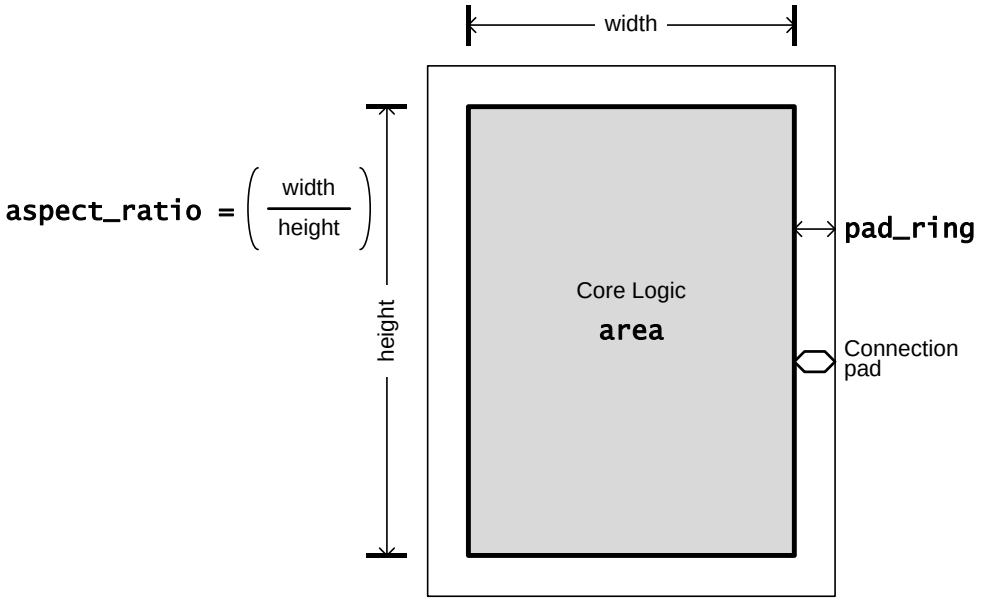
Describing Physical Attributes

The CSL language constructs in this section describe the physical attributes of the system or individual endpoints.

area

The `area` statement defines the area of an individual IP block or for the entire system. When used within an endpoint, the `area` statement specifies the silicon area consumed by the IP block. When describing the area of the entire system, the `area` statement defines just the core logic area and not the pad ring that surrounds the core logic, as shown in Figure 16.

Figure 16: **Area, pad_ring, and aspect_ratio**



The Silistix software uses the values specified by the designer when calculating the percentage of total die area utilized by the interconnect logic. These values are used by the First Placement Estimator (FPE) tool for floor planning purposes. The area specified for the entire system takes precedence over the calculated value from the area values set for each endpoint when determining total area. However, the area values set for the endpoints are still used by the FPE tool.

The `area` specification uses two possible units. The `mm2` unit system measures area in post-layout square millimeters of silicon for the target process technology. The `kgates` unit system describes area indirectly, measured in raw synthesized thousands of gates, pre-layout and without any consideration for clock trees, reset distribution, etc. This value is most commonly derived by synthesizing the endpoint with a logic synthesis package and using the reported gate-equivalence. The specified Silistix target library includes information on how many thousands of equivalent gates are in a square millimeter of silicon for the target process technology. The `endpoint_area_utilization` specification further describes the fraction of the post-layout area that is occupied by gates, the remainder being wiring.

The **area** statement has the following forms.

```
area = <value> mm2 ;
```

or

```
area = <value> kgates ;
```

Example

```
area = 144 mm2; // Die is 12 mm on a side, assuming a square die
```

Example

```
area = 57.7 kgates; // The IP block is equivalent to 57,700
                    // gates according to logic synthesis
endpoint_area_utilization = 80%; // gate area is 80% of total
// Total post-layout area is 125% of gate area,
//   which includes area for wiring
```

Example

```
#define CORE_WIDTH_MM  3.2
#define CORE_HEIGHT_MM 4.2
area = ( CORE_WIDTH_MM * CORE_HEIGHT_MM ) ;
```

pad_ring

The pad ring surrounds the core logic area as illustrated in Figure 16.

The **pad_ring** statement specifies the width of the pad ring surrounding the die, which is equal width around. Any connection pads defined by **pad** statements reside within this pad ring. The **pad_ring** statement has the following form.

```
pad_ring = <expression> um; // Specified in microns
```

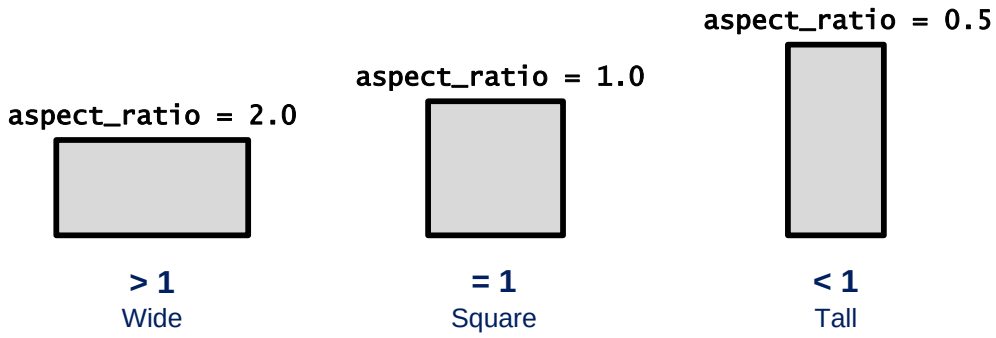
If no **pad_ring** is specified, then the Silistix software assumes a 200 micron wide pad ring.

aspect_ratio

The aspect ratio specification describes the shape of the die or IP block. The aspect ratio value is the width divided by the height, as illustrated in Figure 16. Consequently, a value greater than one specifies a block that is wider than it is tall as pictured in Figure 17. Conversely, a value less than one specifies a block that is taller than it is wide. A value of one specifies a perfect square.

The **aspect_ratio** and the **area** statements completely specify the shape of the die, endpoint, or object, which is used by the First Placement Estimator (FPE).

Figure 17: **Aspect Ratio Examples**



The **aspect_ratio** statement has the following form.

```
aspect_ratio = <value> ; // ratio of ( width / height )
```

If no **aspect_ratio** is defined, then the Silistix software assumes a ratio of 1.0, which represents a perfectly square die or IP block.

Example

```
aspect_ratio = 1.2;
```

Example

```
#define CORE_WIDTH_MM  3.2
#define CORE_HEIGHT_MM 4.2
aspect_ratio = ( CORE_WIDTH_MM / CORE_HEIGHT_MM ) ;
```

block_type

For the First Placement Estimator (FPE) tool, each endpoint or functional block described in the CSL source file can be assigned one of four possible block types. The block type indicates how the block is delivered or described. The various options appear in Table 8. A **soft** IP block is described as RTL in Verilog; synthesized; and then placed and routed during physical implementation. A **hard** IP block is is a pre-implemented, technology-targeted function complete with physical layout. The block type also describes whether a block connects to Silistix network or whether is it a stand-alone terminal function, separate from the Silistix network. If the block type is not defined, then the FPE software assumes that the endpoint is a soft IP block connected to the Silistix network.

So why describe a terminal function that does not connect to the Silistix network? These terminal functions exist in the final design and they occupy space in the physical layout. If these terminal blocks are defined in the CSL, then the FPE software generates a more accurate initial placement and results in a faster back-end flow. Assign these non-connected blocks a **block_type** of **soft_terminal** or **hard_terminal**, depending on whether the block is a soft or hard IP block as described in Table 8. Examples might include a block of memory or a DMA engine driving one of the endpoints. These terminal blocks can be associated with other blocks as described in "Forming Relationships" starting on page 30.

Table 8: **Block Types**

Macro Type	Character	block_type=	
		Connects to Silistix Network	Does not connect to Silistix Network
Soft	Described in RTL, no physical placement information	**soft**	**soft_terminal**
Hard	Hard macro IP with physical layout	**hard**	**hard_terminal**

The **block_type** statement has the following form.

```
block_type = <type> ;
```

Where :

<type> is **soft**, **hard**, **soft_terminal**, or **hard_terminal** as described in Table 8.

A **hard** or **hard_terminal** block type also requires an associated **footprint** statement.

Example

```
Unrelated_soft_IP {
     block_type = soft_terminal ;
} // Other logic used in design, but not on Silistix network
```

footprint

The **footprint** statement specifies the actual name of a hard IP block to be used for placement and routing. Any IP blocks declared with a **block_type** of **hard** or **hard_terminal** must have an associated **footprint** statement. The specified name is used when writing out the placed design to the DEF file. Subsequently, your preferred ASIC/SoC place and route software will insert the correct GDSII view from the library based on this name.

The **footprint** statement has the following form.

footprint = "*<name>*" ;

Example

```
sram_block_cpu {
     block_type = hard_terminal ;
     footprint = "SRAM_DP_32x512" ;
          . . .
} // SRAM block for CPU
```

endpoint_area_utilization

The **endpoint_area_utilization** statement defines how an **area** that is specified in pre-layout **kgates** is then converted to post-layout estimates of total die area, measured in **mm2**. As shown in Equation 3, the endpoint area utilization is specified as the percentage of the final post-layout area that is occupied by gates, the remainder filled with wiring. The target technology library includes a conversion factor that specifies how many thousands of gates fit in a square millimeter of silicon for the target process technology. The **endpoint_area_utilization** statement modifies the values defined in the target technology library, allowing modifications for a particular tool flow, level of expertise, or fabrication contstraints.

The **endpoint_area_utilization** statement has the following form.

endpoint_area_utilization = *<expression>* **%**;

Example:

```
endpoint_area_utilization = 80%; // gate area is 80% of total
// Total post-layout area is 125% of gate area
```

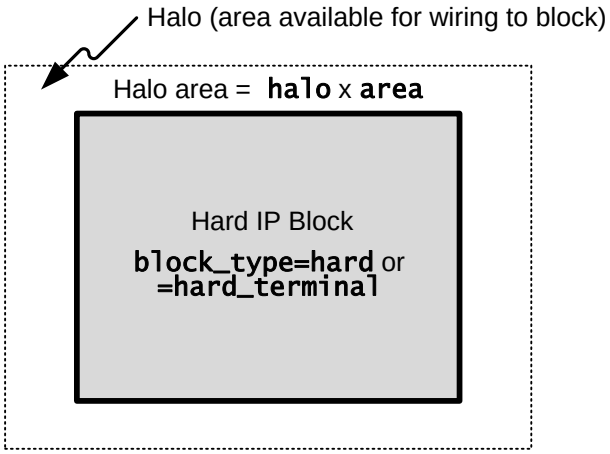
Equation 3

$$\text{Post-layout } \textbf{mm2} = \frac{\textbf{kgates} \times (\textit{Library Conversion Factor})}{\textbf{endpoint\_area\_utilization}}$$

halo

As shown in Figure 18, a "halo" is a region surrounding a hard IP block where other gates or blocks cannot be placed, thereby providing space for wire routing. The **halo** statement specifies the percentage of the hard macro area allocated to wiring to and from the hard IP block. For example, **halo = 5%** creates a region surrounding the block equal to 5% of the area of the block. This value can be specified for any endpoints with **block_type = hard** or **block_type = hard_terminal**. The halo value also applies to the Silistix networks elements that are hard IP blocks.

Figure 18: **Halo around Hard IP Block**



The `halo` statement has the following form.

```
halo = <expression> %;
```

If no halo value is specified, then the Silistix software uses a halo of 5%.

elasticity_threshold

Before First Placement Estimation, the Silistix software estimates timing based on the maximum network hop length, which is determined by the selected technology library. The maximum hop length is the maximum wire length allowed between network components. Connections longer than the maximum hop length are automatically re-buffered using pipelatch components. The `elasticity_threshold` statement scales the maximum network hop length as shown in Equation 4, allowing additional delay and flexibility during layout.

Equation 4

$$\text{Actual hop length} = (\text{Maximum hop length}) \times \texttt{elasticity\_threshold}$$

For example, if the maximum hop length for the specified technology library is 600 µm, and `elasticity_threshold = 80%`, then the Silistix software re-buffers network connections every 480 µm (80% of 600 µm). The extra 20% margin allows for additional flexibility during actual layout.

An `elasticity_threshold = 100%` means that network connections are at the maximum hop length, making it difficult to move the network blocks during physical implementation without perturbing their bandwidth capability. In other words, at 100%, there is little flexibility on where network components can be placed, either in FPE or in physical layout.

The `elasticity_threshold` has the following form.

```
elasticity_threshold = <value> %;
```

If no `elasticity_threshold` is specified, then the Silistix software uses 80%.

Controlling Block Placement and Floor Planning

The CSL language provides various constructs to describe the "virtual" connections and associations between blocks. The connections are not actual signal wires in the design but are a simple means to create relationships between IP blocks or pads on the device. These relationships provide important spatial information and constraints to the First Placement Estimator (FPE) tool.

Forming Relationships

A relationship is defined by first declaring two anchor points and then connecting the two points together with a **net**. An anchor point is either a **pin** on an IP block or a **pad** connected in the pad ring of the device. At least one of the anchor points must be a pin.

A **net** statement then connects the two anchor points together and specifies the relative importance or weight of this relationship.

pin

A pin statement names an anchor point, and specifies its location within a block using Relative Coordinates. This pin can then be referenced in **net** statements to define spatial relationships between blocks. The **pin** statement has the following form.

```
pin <name> <rel_x>%,<rel_y>% ;
```

Where :

<name> is the name of the pin, unique to the endpoint.

<rel_x> is the relative position from the center of the block, based on the width of the block.

<rel_y> is the relative position from the center of the block, based on the height of the block.

To define a relationship with a Silistix network gateway, declare the pin within the initiator or target port declaration. Alternatively, use the **--fpe-center-gateways** or set the **w4** weight using the **--fpe-weights** option for the CSL Compiler or from within the CHAINarchitect software.

pad

The **pad** statement defines the location of an I/O pad on the die. This statement provides a means to associate an IP block to a pad location die during First Placement Estimation (FPE).

A pad can then be connected to a pin specified for endpoint using a **net** statement. The **pad** statement has the following form.

```
pad <name> <rel_x> %,<rel_y> %;
```

Where :

<name> is a unique pad name.

<rel_x> specifies the horizontal position of the pad, relative to the center of the die based on the width of the die. For example, -50% represents the left edge of the die and +50% represents the right edge of the die.

<rel_y> specifies the vertical position of the pad, relative to the center of the die based on the height of the die. For example, -50% represents the bottom edge of the die and +50% represents the top edge of the die.

net

The **net** statement connects pins that are defined in endpoints to pads or to pins defined in other endpoints. The net statement creates "virtual" connections between pins and pads; these are not actual connections within the design but associates blocks to one another. This association provides important spatial constraints to the First Placement Estimator (FPE). Each net statement has a weight that represents the strength of the virtual connection. The higher the weight, the more important it is that the blocks be placed closer together.

The **net** statement has the following form.

```
net <pin_or_pad1> = <pin_or_pad2> (<weight>);
```

Where:

<pin_or_pad1> is the name of a pin defined in an endpoint or a previously-defined pad.

<pin_or_pad2> is the name of a different pin or a previously-defined pad.

<pin_or_pad1> and *<pin_or_pad2>* cannot both be pads. One connection must be to non-pad pin.

<weight> is a value between 1 and 1000. The higher the weight, the higher is the relative importance of the associated connection. Generally, a higher weight results in the connected IP blocks being placed closer together.

When referencing a pin, the following form may be used.

```
<clkdomain>.<endpoint>.<pin>
```

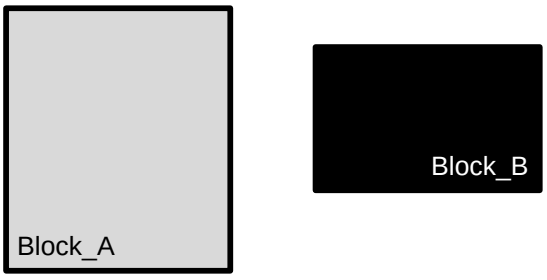
If *<endpoint>* is unique to the system, then *<clkdomain>* may be omitted.

A pad or pin name can only be connected once in the current release. To connect a pad or pin to multiple anchor points, duplicate the pad or pin but with a unique name.

Pin, Pad, Net Connection Examples

An example best illustrates how to create relationships between blocks. Assume that there are two IP blocks in the system, Block_A and Block_B as shown in Figure 19.

Figure 19: **Two IP Blocks in the System**



To form a relationship between these two IP blocks, first create an anchor point within each block, as shown in Figure 20. An anchor point within an IP block is called a **pin**. Creating a connection to a **pad** anchor point is slightly different as described later. In CSL, pins must be declared within the endpoint declaration as shown in Figure 21. The pin can be placed physically anywhere on the IP block using Relative Coordinates. For simplicity's sake, this example places the pins in the center of each block, at position 0%, 0%.

Figure 20: **Creating Anchor Points**

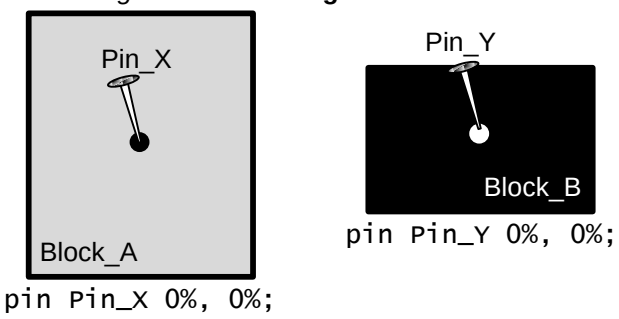


Figure 21: **Declaring Pins**

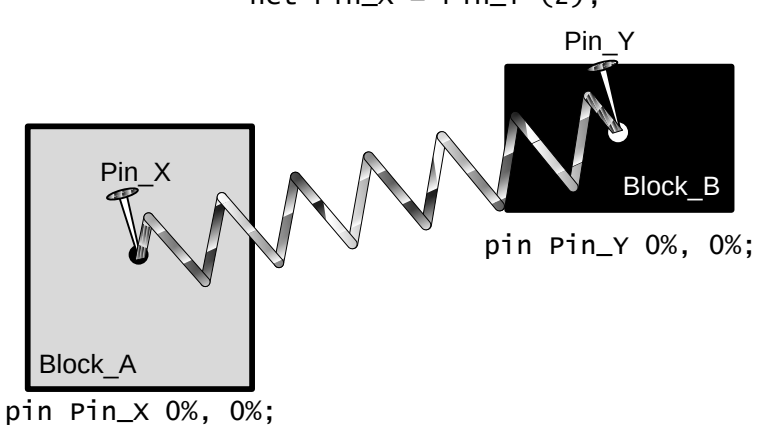
```
Block_A {
   . . .
   pin pin_X 0%, 0% ; // declare Pin_X
   . . .
}

   . . .

Block_B {
   . . .
   pin pin_Y 0%, 0% ; // declare Pin_Y
   . . .
}
```

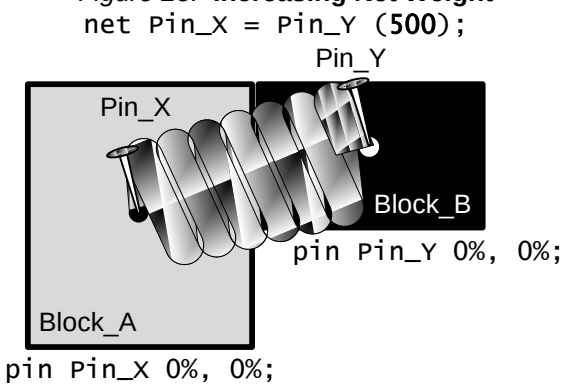
To associate the two blocks, connect them using a **net** declaration as shown in Figure 22. In CSL, the net statement can generally appear anywhere after the **pin** or **pad** anchor points are declared. The **net** statement equates the two pins and then specifies the relative strength or weight of the connection between them. Think of this **net** connection as a spring connecting the two anchor points.

Figure 22: **Declaring a Net**



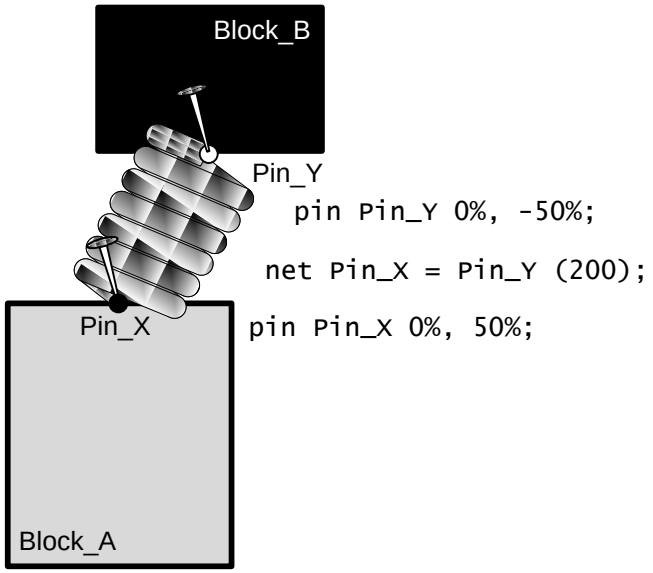
If the spring is relatively weak, as it is in Figure 22, the spring stretches easily. Consequently, Block_A and Block_B are placed in the general vicinity of each other, but not necessarily closely together. By increasing the strength or weight on the net, the blocks are pulled closer together, as illustrated in Figure 23. Increasing the net weight from 2 to 500 strengthens the connection between the blocks, pulling them closer together.

Figure 23: **Increasing Net Weight**



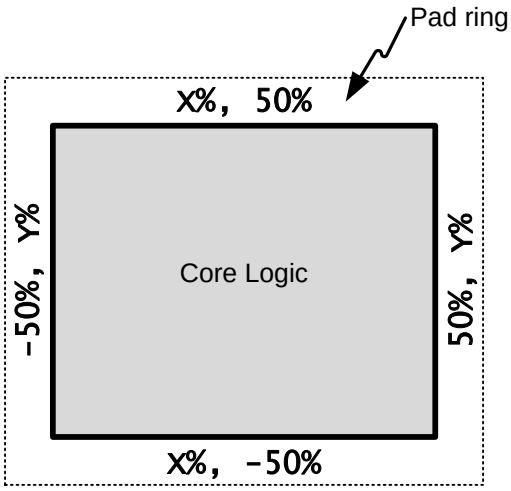
Changing the position of the pins within the IP blocks also impacts placement. In Figure 23, the pins are located in the center of the endpoints. Consequently, Block_A and Block_B fit equally well along any of the four block edges. However, changing the location of the pins, as shown in Figure 24, tends to place Block_B on top of Block_A. In Block_A, the pin moved from the center (0%, 0%) up to the top edge (0%, 50%). Similarly in Block_B, the pin moved from the center down to the bottom edge (0%, -50%).

Figure 24: **Changing Pin Location Impacts Placement**



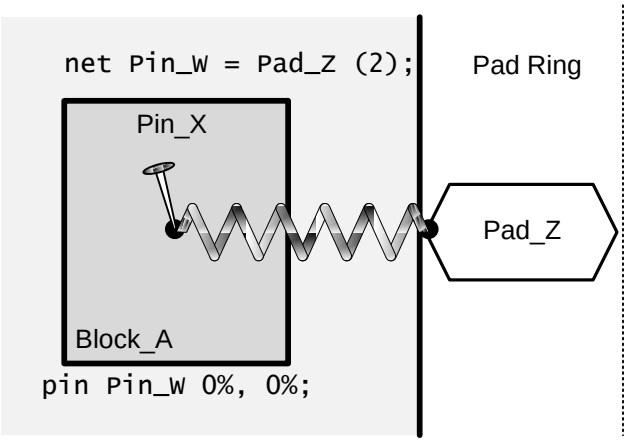
Using a **pad** as an anchor point is similar to using a pin. The difference is that the pad resides in the pad ring surrounding the core area. Again, the **pad** statement does not physically create a pad on the die. Instead, it defines an anchor point in the pad ring surrounding the die. The pad location is specified using Relative Coordinates except that either the X- or Y-coordinate is locked to a device edge. Pads along the top edge of the die always have a Y-coordinate fixed at 50%, as shown in Figure 25. The X-coordinate can vary, specifying any location along the top but the Y-coordinate is fixed at 50%. The other die edges are similar.

Figure 25: **Pad Coordinates**



The example in Figure 26 connects a pin in Block_A, called Pin_W, to a pad called Pad_Z, which is located in the pad ring along the right edge.

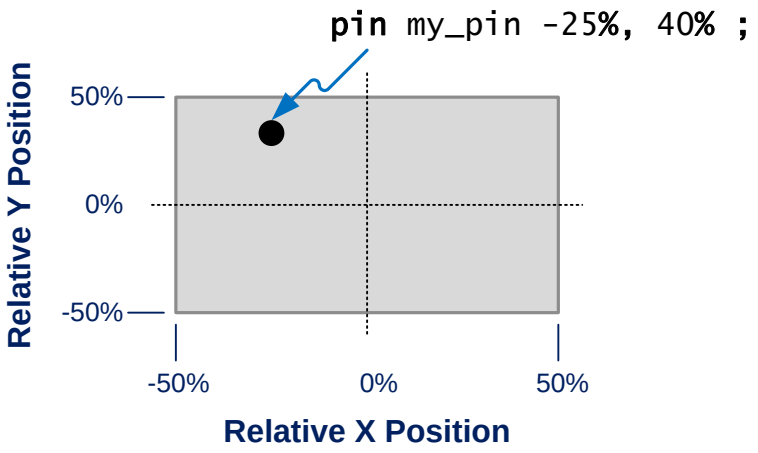
Figure 26: **Example Connection to Pad along Right Edge**



Relative Coordinates

Specify locations within the core area, within the endpoint, or within an IP block using the relative coordinate system shown in Figure 27. This relative coordinate system applies regardless of the actual size or aspect ratio of the block or die. The center of the block is at position 0%, 0%. The horizontal position, or X-coordinate, starts at -50% at the left edge, increases to 0 at the midpoint, and continues increasing to 50% at the right edge. The vertical position or Y-coordinate is similar, although -50% is at the bottom edge, 50% along the top edge.

Figure 27: **Relative Coordinate System**



The same relative coordinate system applies when defining a pad location, as shown in Figure 25.

Absolute Coordinates

Use the `locate` statement to specify an absolute location, relative to the lower left corner of the core area. The values specified are in microns. A `locate` statement applies to any `block_type`, although it is most often used to place `hard` or `hard_terminal` blocks because soft IP blocks typically have an irregular, free-form shape.

`locate` *<x_loc>*, *<y_loc>*;

Where :

<x_loc> and *<y_loc>* are specified in microns, relative to the lower left corner of the core area.

Example CSL Design (with FPE constructs)

The CSL file listed in Figure 28 is essentially the same as that shown in Figure 15 except that it includes various FPE constructions. For additional information on running the First Placement Estimator (FPE) tool, see the "Generate First Placement Estimation" section in ***Building and Analyzing On-Chip Networks using CHAINarchitect*** user guide.

Figure 28: **Example CSL Design with FPE Constructs**

```
// Target technology library
library Silistix_90nm_Generic;

system my_system {

  // set optimization priority (1=highest, 3=lowest)
  optimize_area=1;
  optimize_latency=2;
  optimize_power=3;

  #define CORE_WIDTH_MM  3.5
  #define CORE_HEIGHT_MM 4.5

// Total silicon area, including CHAIN network
  area = CORE_WIDTH_MM * CORE_HEIGHT_MM mm2 ;

  aspect_ratio = ( CORE_WIDTH_MM / CORE_HEIGHT_MM );
```

```
power = 250.0 mW;   // Total system power

// Set exclusion region around IP block, 5% of total area
halo = 5% ;

// Set size of pad ring to 100 microns (um)
pad_ring = 100 um ;

elasticity_threshold = 80% ;

// set threshold on how close the actual values approach the
//    specified limits
// (alternate view):  what is the confidence in the
//     specified limits?
Utilization_threshold = 0.90% ;

// address map for the CPU
address_map CPU_address {
   range comm_mailbox    0x0000000 .. 0x00001ff;
   range external_sdram  0x1000000 .. 0x4ffffff;
}

// address map for the COMM controller
address_map COMM_address {
   range CPU_mailbox     0x0000000 .. 0x00000ff;
   range external_sdram  0x2000000 .. 0x4ffffff;
}

pad CPU_endpoint_pad  -50%,  50% ; // Top-left corner
pad SDRAM_endpoint_pad 50%,   0% ; // Right edge
pad COMM_endpoint_pad   0%, -50% ; // Bottom edge
pad interface_pad     -50%, -25% ; // Left edge

// define each of the various clock domains and endpoints
clock_domain cpu_domain (400 MHz) {
   CPU {
        protocol = "AXI";  // "AHB", "APB", "AXI", "OCP"
        pin CPU_anchor    -50%, 50% ;
        // anchor point to SRAM hard block
        pin CPU_SRAM_anchor 0%,  0% ;
        area = 700 kgates ;
        block_type = soft ;

        initiator CPU_initiator {
           address = 32 bits ;
           data = 32 bits ;
           peak = 1600 MBs ;
           burstsize = 32 bytes ;
           address_map = CPU_address ;
           outstanding = 8;
           write_response = 2.5 ns ;
           read_response = 2.5 ns ;
           axi_id_bits = 4 ;
        } // end CPU_initiator
```

```
        target CPU_mailbox {
           address = 8 bits ;
           data = 16 bits ;
           burstsize = 16 bytes ;
           address_range = COMM_address.CPU_mailbox ;
           axi_id_bits = 1 ;
           axi_write_interleave_depth = 1 ;
        } // end CPU_mailbox target
  } // end CPU endpoint

  // Declare SRAM block used with CPU, not on network
  sram_block_cpu {
       block_type = hard_terminal ;
       footprint = "SRAM_DP_32x512" ;
       // locate = 100,400;
       aspect_ratio = 1.7 ;
       area = 0.11 mm2 ; // 60k gates. Synopsys reported= 0.1
       pin SRAM_anchor 0%, 0% ;
  } // SRAM block for CPU, no connected to network

  // Declare soft interface logic block, not on network
  interface_logic {
      block_type = soft_terminal ;
      area = 0.08 mm2 ;
      pin interface_anchor 0%, 0% ;
  } // not connected to network

} // end cpu_domain


clock_domain memory_domain (333 MHz) {
  SDRAM {
        protocol = "AHB"; // "AHB", "APB", "AXI", "OCP"
        area = 2.5 mm2 ;
        aspect_ratio = 1.3 ;
        pin SDRAM_anchor      0%, 0% ;

        target SDRAM_target {
           address = 32 bits ;
           data = 32 bits ;
           burstsize = 128 bytes ;
           address_range = {CPU_address.external_sdram,
                            COMM_address.external_sdram} ;
           write_response = 25 ns ;
           read_response = 50 ns ;
        } // end SDRAM_target
  } // end SDRAM endpoint
} // end memory_domain
```

```
  clock_domain communications_domain (180 MHz) {
     COMM {
          protocol = "AHB"; // "AHB", "APB", "AXI", "OCP"
          area = 2.1 mm2 ;
          pin COMM_anchor 0%, 0% ;
          initiator communications_initiator {
             address = 32 bits ;
             data = 32 bits ;
             burstsize = 64 bytes ;
             address_map = COMM_address ;
             ahb_version = lite ;
          } // end communications_initiator

          target comm_mailbox {
             address = 16 bits ;
             data = 16 bits ;
             burstsize = 64 bytes ;
             address_range = CPU_address.comm_mailbox ;
             write_response = 10 ns ;
             read_response = 5 ns ;
             ahb_version = lite ;
          } // end comm_mailbox
      } // end COMM endpoint
  } // end communications_domain


  // Declare pins and pads before specifying nets
  // Net weights defined in parentheses.

  // Associate CPU block and SRAM block
  net CPU_SRAM_anchor = SRAM_anchor (500) ;

  // Define pin/pad connections for placement
  net CPU_endpoint_pad = CPU_anchor (500) ;
  net COMM_endpoint_pad = COMM_anchor (50) ;
  net SDRAM_endpoint_pad = SDRAM_anchor (250) ;
  net interface_pad = interface_anchor (10) ;

// Define connections between endpoints

    . . .

} // end system
```

General Design Methodology

The following steps provide an overall methodology to specify a system in CSL.

Include the target technology library.

For the `system` …

- Give it a name
- Declare each independent `address map`. For each address map …
  - Give it a name
  - Declare each `range` of target addresses within the address map. For each address range …
    - Give it a name
    - Provide a starting (low) byte address
    - Provide an ending (high) high address
- Declare every clock `domain`. For each domain …
  - Give it a name
  - Specify its operating frequency in `MHz` or `Mhz`
  - Within each clock domain, declare every `endpoint` that connects to other endpoints in the system. For each endpoint …
    - Give it a name
    - Declare the interface `protocol`
    - Declare every `initiator`, including the following characteristics:
      - ☐ Give it a name
      - ☐ Declare `address` width in `bits`
      - ☐ Declare a `data` width in `bits`
      - ☐ Optionally, declare a `peak` transfer rate in `Mbs`, `MBs`, `Gbs`, or `GBs`. Only required if the endpoint cannot support the theoretical peak bandwidth (frequency × data width).
      - ☐ Declare a maximum `burstsize` in `bits` or `bytes`
      - ☐ Declare an `address_map`
      - ☐ Optionally, declare the number of `outstanding` transactions allowed
      - ☐ For accurate roundtrip system timing predictions, specify the `write_response` and `read_response` delay between initiator write and read requests
    - Declare every `target`, including the following characteristics:
      - ☐ Give it a name
      - ☐ Declare `address` width in `bits`
      - ☐ Declare a `data` width in `bits`
      - ☐ Optionally, declare a `peak` transfer rate in `Mbs`, `MBs`, `Gbs`, or `GBs`
      - ☐ Declare a maximum `burstsize` in `bits` or `bytes`

- Declare an `address_range` or multiple ranges
- For accurate roundtrip system timing predictions, specify the `write_response` and `read_response` delay for the target

For each operating `mode`, declare the following connection information.

- For each connection from an initiator → target, describe the write operation characteristics.
  - Specify roundtrip connections using the `=>` operator. Alternatively, specify unidirectional connections using the `->` connection operator.
  - Bandwidth in `Mbs`, `MBs`, `Gbs`, or `GBs`
  - Optionally, specify latency in `ns`. Specify roundtrip system latency when using the `=>` operator and switching latency with using the `->` operator.
- For each connection from an initiator ← target, describe the read operation characteristics.
  - Specify roundtrip connections using the `=>` operator. Alternatively, specify unidirectional connections using the `->` connection operator.
  - Bandwidth in `Mbs`, `MBs`, `Gbs`, or `GBs`
  - Optionally, specify latency in `ns`. Specify roundtrip system latency when using the `=>` operator and switching latency with using the `->` operator.

Set the relative priorities for `optimize_latency`, `optimize_power`, and for `optimize_area`, with `1` being the highest, `3` being the lowest. The priorities can be set for the entire system or for individual connections.

Optionally, specify the total silicon `area` occupied by the system design, including the area used by the CHAIN network.

Optionally, specify the total `power` consumed by the system design, excluding the power consumed by the CHAIN network.

To overprovision latency or bandwidth, set the `utilization_threshold` level, either for the entire system or for individual connections.

Add First Placement Estimator (FPE) constructs.

Naming/Identifier Conventions

To make your CSL code more understandable to others and to ease debugging and analysis, use descriptive names.

In the CSL language, names or identifiers …

- are **case sensitive**
- must begin with one of the following …
  - an **uppercase letter** (A .. Z), or
  - a **lowercase letter** (a .. z), or
  - an **underscore character** (_)
- may include any further combination of …
  - **uppercase letters** (A .. Z),
  - l**owercase letters** (a .. z),

  - **digits** (0, 1, .., 9), and
  - **underscore characters** (_).
- can be **up to 32 characters** long. Any identifier longer than 32 characters is truncated to 32 characters.

Do not use a space, or a hyphen, dash, or minus sign (-) in names or identifiers. These characters will generate a syntax error.

Table 9 provides a few examples of invalid names along with corresponding valid names.

Table 9: **Invalid and Valid Names/Identifiers**

Invalid Names/Identifiers	Issue?	Valid Names/Identifiers
`CPU-System`	Hyphen in name	`CPU_System`
`32MHz`	Name begins with a number	`Clock_32MHz` or `_32MHz`

Number Conventions

In CSL, specify numbers as using one of the formats shown in Table 10.

Table 10: **CSL Number Formats**

Number Type	Examples
Decimal Integer	`12345`
Hexadecimal Integer	`0xabc123`

Decimal Floating Point	`1.2345`
	`0.1234`
	`1.23e-10`

Revision History

Revision	Date	Description/Revisions
1.3	31-OCT-2008	Minor corrections. Added information on using Modes of Operation when declaring connections.
1.2	5-AUG-2008	Added sections on First Placement Estimator (FPE) Constructs. Updated to CHAINworks 2.1 release.
1.1	22-MAY-2008	Updated with recent syntax changes.
1.0	21-DEC-2007	Initial release.

Feedback

Feedback on this Silistix document and all Silistix products is highly encouraged. If you have a comment, correction, or suggestion to improve this document, please send us an E-mail. Please include complete details including page numbers, section titles, or figure or table numbers where appropriate. Thank you in advance for helping us to improve our products and services.

feedback@silistix.com

Disclaimers