Public

ASAM

Grant agreement no. 100265
Artemis Project

ASAM

Automatic Architecture Synthesis and Application Mapping

D4.1: Implementation and prototyping infrastructure: organization and interfaces

Due Date of Deliverable	October 31, 2010
Completion Date of Deliverable	October 31, 2010
Start Date of Project	May 1, 2010 – Duration 36 months
Lead partner for Deliverable	UNICA
Author(s)	P. Meloni(UNICA), L. Raffo (UNICA), M. Lindwer (SH), L. Jozwiak (TUE), M. Berekovic (TUBS)

Revision: 2.0

Project co-funded by the Artemis Joint Undertaking Call 2009		
Dissemination Level		
PU	Public	x
PP	Restricted to other program participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	
COA	Confidential appendices	

Contents

1 Introduction

This document is aimed at defining a preliminary set of specifications to be taken into account within the creation of the technology-dependent hardware generation/prototyping/implementation/optimization infrastructure that is the main object of WP4.

The main aim of such framework will be to provide support for all the levels of the architectural analysis, reducing the gap between the estimation of the performances considered during the early steps of the design flow and those really measurable after the implementation.

Firstly, the framework will be capable of creating an FPGA implementation of a candidate system configuration. The target application will be executed on it for emulation, in order to obtain switching activity figures and performance metrics to be considered during the micro- and macro-architectural optimization processes.

Secondly, the infrastructure will be capable of creating all the necessary input for a prospective VLSI back-end implementation process of the candidate architecture.

The document is organized as follows: a first section describes the general organization of the infrastructure, briefly depicting its subcomponents (tools) and their functions. Accordingly, the second section describes more in detail the interfaces between the infrastructure and the external environment, discussing more in detail the information that must be provided to feed the different tools and the results that will be obtained in output from their execution.

All the assumptions included in the present document have to be considered preliminary, their validation will be an outcome of the planned research activities, the proposed specifications may be changed during the project to increase the efficiency and the usefulness of the design flow and its particular tools.

2 Infrastructure general organization

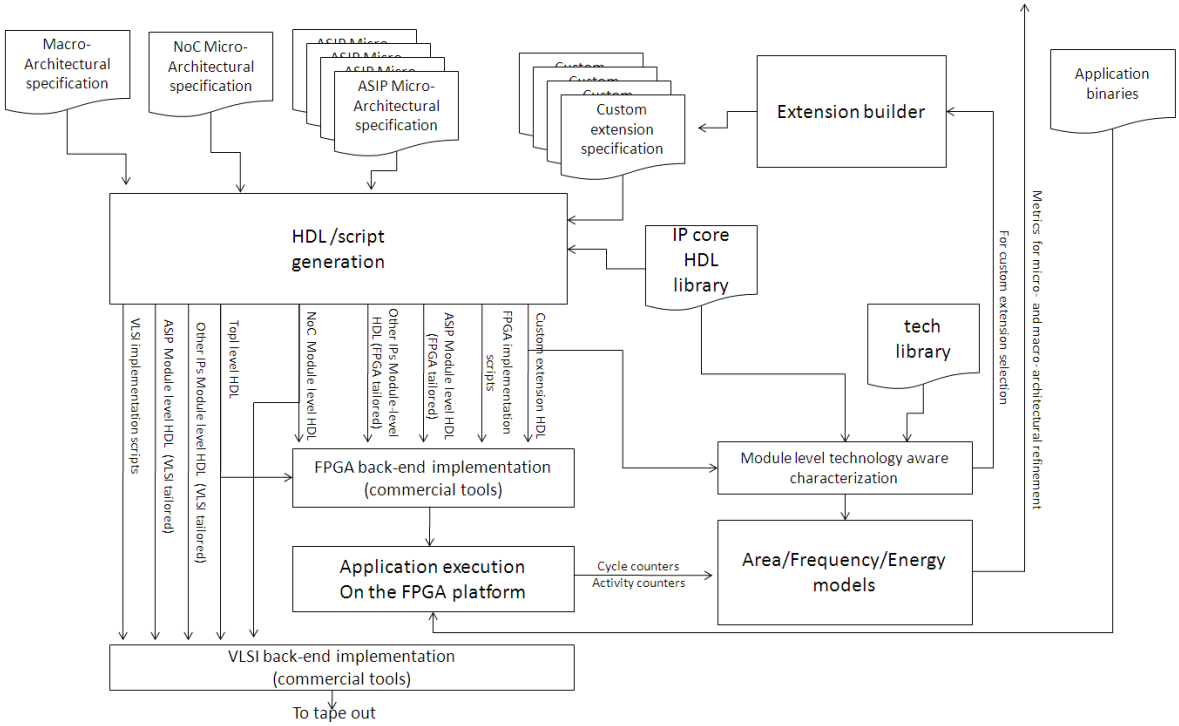

Figure 2.1: Infrastructure general organization

Figure 2.1 depicts the general overview of the infrastructure. It is composed by several sub-tools that interact with each other and with commercial tools. The inputs for the framework are placed at the top of the flow-chart. All the inputs are needed to describe the system configuration under prototyping.

In particular, as envisioned by the general approach of the ASAM project and discussed in Deliverable D1.1, the description of the system is provided by means of a macro-architectural specification (involving the system-level description), together with a set of micro-architectural specifications (involving the module-level description of the configurable modules to be instantiated in the system). The specific format of this input will be described more in detail in chapter 3.

In the following we will preliminarily describe the functionalities implemented by every tool, to create a set of development directives to be taken as reference during the research activity aimed at the creation of these tools.

2.1 The HDL/script generation

Figure 2.2 shows a detailed view of the sub-flow in charge of converting the system- and module-level specifications into the inputs needed for the implementation of the candidate architectural configuration. The sub-flow will produce all the HDL code to be synthesized and all the scripts needed to run the implementation flow on commercial tools.

As may be noticed in the figure, not all the information is generated by a single tool, in particular three different tools are used, each one in charge of creating the HDL description and the implementation commands related to a different kind of IP-core.

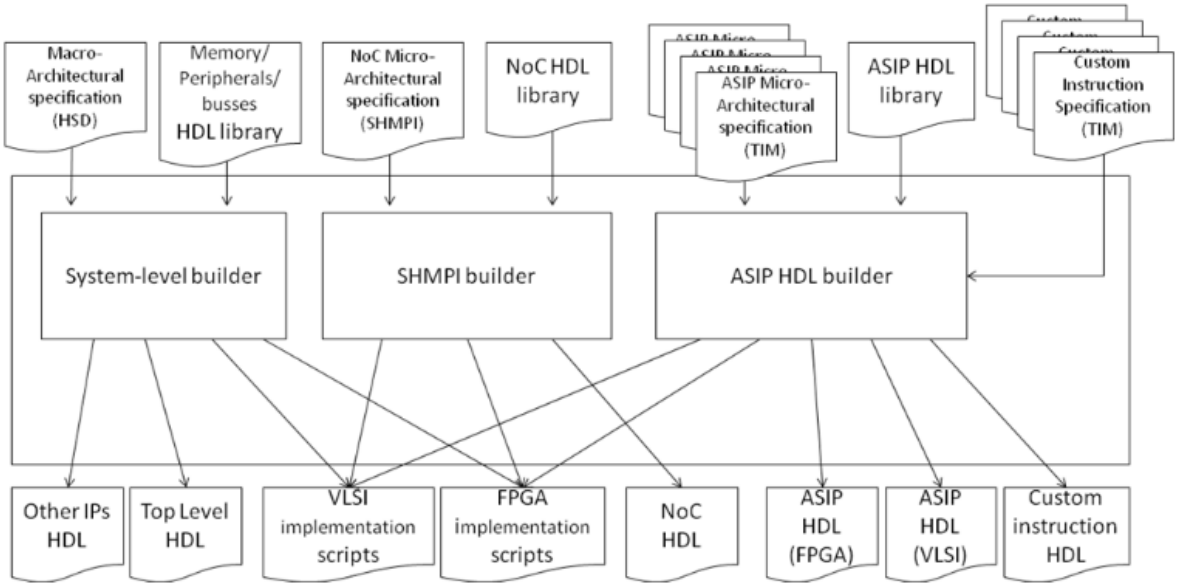

Figure 2.2: Detailed view of the HDL/script generation

2.1.1 System-Level Builder

Figure 2.2 we call Topology Builder a tool capable of analyzing the system-level specification describing the macro-architecture under prototyping/implementation. It will be capable of generating an HDL description of the upper hierarchy-level of the system, including instantiation of the micro-architectural sub-modules (the HDL description of the sub-modules will be produced by the other tools, see in the following).

2.1.2 ASIP HDL builder

The box called ASIP HDL builder in figure 2.2, represents a toolset that is part of the HiveLogic suite by Silicon Hive. The toolset, referring to a library of building blocks, allows the creation of the RTL description of a customized processor. The toolset, as mentioned, receives in input a specification of the micro-architectural features of the customized processor to be produced, expressed using the TIM language, a Silicon Hives proprietary language whose capabilities will

be described more in detail in the following sections and whose syntax is explained in depth in Deliverable D1.1. The TIM description, provided by the upper layers of the design flow, is elaborated by a tool named TIM-compiler. Beside generating some input for the software compilation toolchain, the TIM-compiler produces an intermediate file that is an input for Genesys, the tool actually taking care of generating the RTL code for the ASIP.

Genesys makes use of shared libraries containing the basic building blocks in a parametric form, ready to be assembled and customized based on desired processor requirements.

The RTL module representing the customized processor will be instantiated by the top-level RTL description created by the previously mentioned topology compiler. By means of the toolset, moreover, a testbench and the stimuli needed for functional verification are created.

Genesys provides also the capability of generating the HDL description of single functional units implementing custom instructions. This capability will be used inside the framework to evaluate the possibility of adding candidate custom instructions to a particular ASIP. An HDL module will be generated for each candidate, and passed in output for technology aware characterization.

All the HDL generations performed by Genesys can be targeted for FPGA (including the use of special primitives to minimize the resource utilization inside the programmable device) or for VLSI implementation (to create the HDL needed for a prospective implementation of the selected optimal architecture, or for feeding the technology-aware characterization process).

The TIM-based description of the ASIP micro-architecture, is also an input for the software compilation toolchain, that offers a compiler, a scheduler and a linker to translate an ANSI-C application into a binary program, with the same customized processor as a target. However, being the software infrastructure an objective of research activities planned within other work packages, the compilation toolchain is not considered part of this prototyping infrastructure. The binaries to be executed by the processors in the system are considered an input for the prototyping framework.

2.1.3 SHMPI builder

The SHMPI builder is a tool that generates a complete HDL Network-on-Chip topology, creating, connecting and configuring a set of network component instances. The produced topology is custom-tailored to the NoC micro-architectural specification file provided in input to the framework.

This file contains the network description, composed by a list of all the access points, to be connected to the cores instantiated in the system, along with the specific interconnection topology. Further detail about the format of the input file can be found in section .

The HDL description of the topology will also include all the structures, adequately configured and programmed, that allow the correct routing of the packets through the network according to the address maps specified in the topology file. The SHMPI builder will also provide as an output all the scripts and commands needed for the FPGA and VLSI implementation of the network modules.

2.2 FPGA back-end implementation

The FPGA-tailored HDL code and the scripts produced by the previously mentioned tools will be adequate to be taken as input by commercial tools. All the different implementation steps will be performed in order to produce the a programming file for the target FPGA device. The script generation will be kept as "vendor independent" as possible in order to allow the user to refer to his legacy implementation toolchain.

2.3 VLSI back-end implementation

The VLSI-tailored HDL code and the scripts produced by the previously mentioned tools will be taken as input by commercial tools to generate a pre-tape-out description of the system, suitable to be used for actual production or for be characterized at the highest precision level. The script generation will be kept as vendor independent as possible in order to allow the user to refer to his legacy implementation toolchain.

2.4 On FPGA platform execution

The actual prototyping will consist in the execution of the target application on the complete system implemented on the target FPGA platform, or in a partial evaluation, such as the execution of a part of the application code on a given ASIP implemented on the FPGA platform, or the test of the hardware of a proposed custom extension. The application binaries, provided as input to the prototyping infrastructure by the software compilation toolchain (see Section), will be loaded in the program memory of the processors included in the system. Dedicated hardware probes/counters related with the metrics to evaluate, will be connected to the system signals, and will be made accessible by the processors in the system as memory mapped resources. Their value will be sampled either or by a dedicated external processor implemented on the FPGA (in this case the access could be done with an arbitrary frequency selected by the user, in order to evaluate peak power consumption, for example), or by one of the processors under prototyping (to save hardware resources, but limiting the sampling at the end of the execution). The performance numbers and the activity traces obtained in this way will be interpreted using the models described in Section and provided as feedback to the upper layers of the design flow to drive the architecture refinement process.

2.5 Technology aware characterization and area/energy/frequency models

Within the proposed framework, the use of analytic models will be coupled to FPGA fast emulation and feedback from hardware synthesis tools, in order to obtain early power, energy, timing and area figures related to a prospective ASIC implementation, without the need to perform long-lasting cycle-accurate simulations.

The metrics extracted with the FPGA based emulation of the system or from hardware synthesis tools are passed as input to the models for the estimation of the physical figures of interest.

The timing figures, expressed in terms of clock cycles, will be translated in time units annotating them with the frequency numbers included in the models. In the same way, the switching activity counts extracted from the execution of the target application on the FPGA, will be translated in energy numbers referring to the energy models. The area models will be used to directly pre-estimate the area obstruction of each module in the system-level description, according to the chosen configuration for its parameters.

Such kind of models will be produced for every IP core prospectively included in the system, after a module-level design exploration phase that we call here technology aware characterization. The RTL description of the IP components will be synthesized, implemented at layout level and characterized under multiple architectural configurations and activity conditions, referring to a target VLSI technologic library, using commercial implementation tools. In this way, a database of experiments, each one with an associated power/area/energy number will be produced.

Using this "training set" of experiments, the models will be derived for each module, alternatively in three different forms:

- A look up table. In this case area obstruction/working frequency/energy consumption values are evaluated during the "training" phase and provided for every significant configuration of the parameters of the considered IP core (and to a given activity event, when energy is considered). The framework accesses the look up table as needed to evaluate the system configuration under prototyping. This form of modeling will be suitable for those IP cores that allow a limited number of significant configurations.

- A set of analytical expressions. In this case the model is an expression that computes area, timing and energy figures. The variables inside the expression will be the parameters and the activity counts. The coefficient will be derived using the training set by interpolation or regression. This form, prospectively less accurate, will be however suitable in those cases where the evaluation (during the "training phase") and the utilization (during the actual work of the framework) of a complete table of values would be unfeasible due to the unmanageable number of possible configurations.

- A hybrid form, where different analytical expressions are provided for different sub-ranges of variables.

A "test phase" is envisioned for the models, involving the verification of their accuracy. Figures obtained using the models will be compared with real values to estimate the error introduced by the modeling activity.

Preliminary tests assess the achievable accuracy of such models to be acceptable, when complete system topologies are considered, with respect to post layout analysis of real ASIC implementations.

2.6 Extension builder and extension characterization

ASIP IP cores considered in the ASAM project are configurable, but also extendable. This last feature means that custom operations can be added to basic processors together with the corresponding hardware implementing them, as well as, any other hardware extensions can be added to increase the processor effectiveness or efficiency. Also, the heterogeneous SoC system may contain hardwired accelerators, which are specified such that the flow will be able to match their functionality with parts of the application code. All these ASIP or SoC extensions are related to the micro-architecture and hardware design, and therefore, they will be constructed and selected by an autonomous sub-task in the scope of the micro-architecture customization task. In figure 2.1 this autonomous sub-task of the micro-architecture customization is represented as Extension Builder.

During the construction of the promising extensions and selection of the best of them, the proposed extensions have to be analyzed and evaluated in separation, before deciding to include them into the ASIP IP cores or SoCs, and before actually including them. The prototyping infrastructure has to enable the separate analysis and evaluation of the hardware extensions proposed. Fortunately, Genesys provides among others the capability of generating the HDL description of single functional units implementing custom instructions. This capability will be used inside the prototyping infrastructure to evaluate the possibility of adding candidate custom instructions and related hardware to a particular ASIP. Using Genesys an HDL module will be generated for hardware of each candidate custom instruction, and used as input for the technology aware characterization. In a similar way all other ASIP or SoC hardware extensions will be analyzed and evaluated, i.e. their corresponding HDL modules will be generated and characterized subsequently.

3 Implementation and prototyping infrastructure interfaces

In this section we define how the infrastructure will be interfaced with the external environment. We preliminarily outline the information that must be provided as input to the framework, the results that will be expected as outputs and the format that will be taken as reference.

3.1 Input information

3.1.1 System-level specification (Macro-architectural specification)

The macro-architectural specification of the system will be produced by the upper levels of the design flow. It will include directives related with:

- System population (number and kind of Hive cores to be included by the system)
- Memory mapping
- External memories
- FIFO adapters
- Bus interconnects
- Network On Chip interconnects
- Host Processor

To ensure better compatibility with the whole SHs toolset, the adoption of SHs proprietary HSD format is envisioned.

The HSD (Hive System Description) describes the components, connections and properties of a system. With the word "system" is intended as a number of Silicon Hive processors, a host processor, memories (also called system memories) and/or custom devices.

The HSD language is a declarative language, in which objects can be declared and connected by connecting the ports on the objects. HSD syntax is completely identical to TIM language syntax, that is the language used by Silicon Hive to fully describe its own cores and functionality.

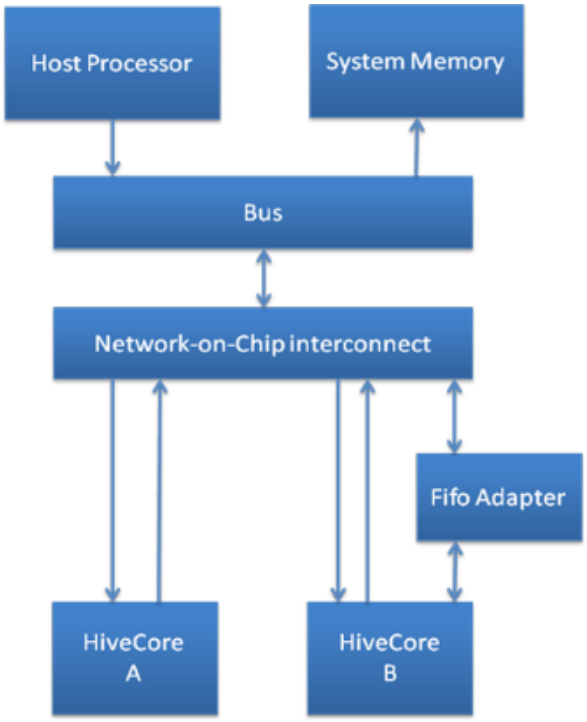

Figure 3.1: Example of System Configuration

All of these components are connected through a bus, a Network-on-Chip or directly. In HSD, there is a full set of basic building blocks that can be instantiated and connected to describe and provide the desired functionality: this set includes buses, protocol converters, dma units, fifo adapters, external memories among other devices.

Listing 3.1: Basic system description

```
1  #include "my_cell/sdk/include/my_cell_processor.hsd"
2
3  Device SystemBus (mmio_port<32,32> ip0) -> (mmio_port<32,32> op0)
4  {};
5
6  System my_system
7  {
8    my_cell proc(bus.op0);
9    HostProcessor host<32,32>;
10   SystemBus bus(host.op0);
11 }
```

In listing 3.1 a basic system description is presented: firstly, the HSD file regarding the Hive core is included (since theres no need of editing it and is always provided with a core), then a System bus is instantiated with one input and one output port (of Memory Mapped I/O type); these blocks are then used to create a basic System, with the addition of a virtual container for an Host Processor that takes care of interact with lower level cores.

HSD also allows integration of custom devices inside a common system description. As long as language syntax is respected, only interface knowledge and interconnect is needed to achieve it.

Indeed, the instantiation of Network-on-Chip topologies, generated by the SHMPI builder, within HSD will be possible referring to it as to a so-called "black-box" device. Only its interfaces will be specified, specifying for each port (that will be an access point for the network) its type (streaming or memory-mapped), its data width and address width. Then, the black box can be normally connected to the other components in the system, leaving the internal implementation details inside the underneath custom RTL code.

The same mechanism will be used, within the project, to allow the inclusion of modules generated on the basis of their micro-architectural specification in the system, offering a feasible solution to the embedding heterogeneous suppliers' cores in a single, high-level description.

3.1.2 Module level specification (Micro-architectural specification)

Two main micro-architectural descriptions will be given in input to the system, specifying respectively the internal structure of each ASIP processor included in the system and the details of the interconnection platform. In the following we provide the description of such specifications, highlighting the information that they must provide to the previously mentioned HDL generators.

3.2 ASIP-processor specification (TIM language)

All Silicon Hive processors are built according to an architecture template. At the top-level the template defines a processor as consisting two different parts, VLIW core and a memory/system interface. Combined, these units form a VLIW machine capable of executing parallel software with a single thread of control.

The VLIW core performs computations under software program control, and consists of a VLIW data path and a sequencer controlling the data path under software control. The data path contains a number of functional units organized in a number of parallel issue slots, which are connected to registers organized in a number of register files via programmable interconnect. The functional units perform compute operations on intermediate data stored in the register files. On each issue slot, an operation can be started in every clock cycle.

The memory/system interface provides a memory and a I/O subsystem allowing the core to be easily integrated in any system.

The system in which a cell is integrated has access to the memories within the processor via slave interfaces. System access to program memory and status and control registers of the cell is provided through slave interfaces as well. These interfaces support commonly used standard protocol to provide easy integration of a processor in a wide variety of system architectures.

Using TIM language during design time, the following design choices can be made:

- number of issue slots
- functional units (number, type, distribution)
- register files (number, sizes, ports)

- interconnect within issue slots and between issue slots
- instruction set
- application-specific instruction set
- local memories (number, connection, ports)
- interfaces: number, type, protocol

Please refer to deliverable D1.1 for more details about the TIM language and micro-architectural specification contents.

3.3 Network-on-chip specification (SHMPI builder language)

The creation of a NoC topology to be instantiated as interconnect structure inside a system under prototyping is done automatically. This generation is done by means of a two-step procedure: first, the input file is parsed by a high-level topology compiler in to an internal data structure (the parsing engine is an extension of parsing engine of the Xpipes compiler, a network topology customizer well known in literature), then the output RTL files that describe the hardware components of the complete NoC platform are generated.

The builder automatically instantiates network components (routers, links, network interfaces, counters) for a specific NoC topology, using the xpipes library, which is designed for achieving high performances. All the components are highly parameterized, and they can be tailored to the communication needs of a specific architecture; also links can be pipelined to an arbitrary degree.

The main contents of this topology file are:

- Number and IDs of the access points to the interconnection platform.
- Interconnection architecture, expressed in terms of switches and links.
- The routing tables to be used by the network interfaces (the considered interconnection network will apply source routing).
- A set of parameters used to configure the cores and the interconnection modules. Examples of configurable parameters for the interconnection network are the number of I/O ports of the switches, the depth of the packet buffers, the number of virtual channels and the separation of clock domains.
- The address map for the memory cores and for the different memory mapped peripherals to be connected by in the platform. This information can be eventually taken from the system level description, and will be used to associate the memory addresses accessed by the access points to the different entries of the routing tables, thus identifying for every requested address a path to a specific destination.
- Specific directives for the inclusion and the connection of the performance/event counters needed for the metrics extraction.

The following is an topology example file, interspersed with explanatory text. File format is standard txt. The line numbers shown are provided for reference only.

```
1  // Topology file example:
2  topology(topology_2x2);
```

Line 1 is a standard C-style line comment (block comments are not allowed).

Line 2 specifies topology name, and is used as an identifier discriminating among the different topologies under emulation.

Then, the access points are declared. An information about the kind of core connected to the access point is provided to be capable of fine-tuning the Nis connected to the access points accordingly. Cores can be processing units, private memories, shared memories, semaphores.

```
3  core(pu_0, switch_0, 1, 6);
4  core(pm_1, switch_0, 1, 6, target:0x10, high:0x1000ffff);
5  core(shm_2, switch_1, 1, 6, target:0x06, high:0x0600ffff);
6  core(ts_3, switch_1, 1, 6, Testandset, target:0xff, 16);
```

Line 3 defines an access point to be connected to the processing unit, assigns an ID (_0 in this case), specifies to which network switch the related NI has to be connected (switch_0 in this case) and specifies some parameters for the related NI (frequency ratio between core and network and number of buffer locations).

Line 4 and 5 define respectively a private memory and a shared memory. For such shared devices some additional information are needed, i.e. memory mapping and size (obtained from the memory high address).

Line 6 defines a Test-and-Set hardware semaphore. In this case the directive specifies the memory mapping, and the chosen value for an architectural parameter (allowed maximum number of contemporary locks).

We envision the possibility of deriving some of the directives provided in this first section directly from the system-level description, to avoid redundancy and possible incoherencies. Whatever else specified in the file is related with the topological shape of the network and with the routing strategy, thus is intrinsically related with the micro-architectural description of the NoC structure.

```
7  switch(switch_0, 3, 3, 6, 0, 0);
8  switch(switch_1, 3, 3, 6, 0, 0);
```

Lines 7 and 8 define switches. For every module parameters such as ID, number of input ports, number of output ports, input buffers depth are defined (respectively the first four fields in the expression). Moreover for every switch the directive defines some information related with a special support, implemented in the network modules for dynamic reconfiguration of the network. Every switch is reconfigurable at runtime from one of the NIs (the fifth and the sixth field in the directive represent respectively the ID of the NI in charge of reconfiguring the switch and the NI dedicated port through which the reconfiguration is transmitted).

```
9  link(link0,  switch_0,  switch_1);
10 link(link1,  switch_1,  switch_0);
```

Lines 9 and 10 define links. Every link instantiation establish an half-duplex connection between two switches. For every link the related directive specifies, in order, link name, source switch, destination switch. Only links between two switches are allowed. NIs are connected to the switches according to the directives related to the access points.

Finally, the routing strategy for the packets is derived,. For every source-destination pair in the system, a route is declared listing in order all the switches that must be traversed by the packets sent from the source to reach the destination.

```
11 route(ap_0, pm_1,   switches:0);
12 route(ap_0, ts_2,  switches:0,1);
13 route(ap_0, shm_3,  switches:0,1);
14 route(pm_1, core_0, switches:0);
15 route(ts_2, core_0: switches:1,0);
16 route(shm_3, core_0: switches:1,0);
```

A route directive has the following fields, source, target (both must be core types), switches that take part in the route (specified in the right order).

3.3.1 Compiled code for each core in the platform under prototyping/implementation

The software compilation toolchain is not considered part of the prototyping infrastructure. Its output, i.e. the binaries to be loaded in the system memories, are considered as an input for the toolset.

Typically, two kinds of binary files could be provided:

Binaries for the host processors, including instructions and data to be executed by the host processors, as well as trunks of code and data to be dispatched by the host processor towards the ASIPs in the system.

In case of global data in the applications, the generated control/host code will also initialize the local memories of the processors.

Binaries for the accelerators could be also provided, to be used for partial prototyping purposes, when the content of their private program and data memories will have to be initialized independently, at the beginning of the prototyping, without the intervention of the host processor.

In particular all the binaries will be generated after compilations according to the system-level address maps in the HSD. From those, adequate header files are generated and used to allow each processor's compiler to address each separately addressable block within each other element in the system address map.

The tools provided by the commercial implementation toolchain will be used to initialize the contents of the memories before the prototyping, see section for further details.

3.4 Output information

3.4.1 Metrics, cost functions

The main metrics that will be considered as objectives for the optimization of the system will be:

- Area occupation

  As an initial step, the area occupation of each module included in the system, estimated by means of the mentioned models, will be considered as an independent additive contribution to the total area of the system. In a second phase, we prospectively envision the use of a physical floorplanner to improve the accuracy of the area estimation. The introduction of such a tool in the toolset will allow the estimation of the total area of the SoC under optimization, considering the actual placement of the physical macros representing the modules on a prospective die.

- Power/Energy consumption

  Power and energy will be estimated in an activity-based manner, annotating the switching activity obtained from the counters included in the FPGA prototyping platform, with energy numbers derived from the previously mentioned models.

- Execution Time

  The number of cycles evaluated after the prototyping will be scaled according to the target working frequency of the system. Achievable working frequencies will be estimated, by means of the mentioned frequency models, for every candidate system-level configuration under prototyping, according to the minimum clock period estimate for every module included in the system. Emulation consistency issues might arise when interfacing with off-chip devices (e.g. memories, I/O ports), since the access latencies, expressed in terms of clock ticks, evaluated by the prototyping environment might be considerably different than those really measured in a prospective VLSI SoC. To avoid this problem and to ensure the accuracy of the emulation, the clock speed of all the off-chip devices that are intended to interface with the final system, will be scaled down proportionally to the emulated operating frequency.

Beside the objective functions to be optimized, several other metrics will be collected at each prototyping step, to drive the optimization algorithms in iteratively refining the architectural configuration.

Here follows a preliminary list, to be validated or extended/reduced during the project according to the results of the future research activities.

- Metrics related with ASIP micro-architecture:
  - Number of accesses to every issue slot
  - Number of accesses to every functional unit inside the issue slots
  - Number of accesses to every register file
  - Number of accesses to every internal bus from load-store-unit to memory

  - Number of accesses to every branch of the register selection network module

- Metrics related with the interconnect structure
  - Number of flits sent by every NI and by every switch port
  - Number of nacks at each switch port due to congestion
  - Number of idle cycles

The implementation of the hardware probes, needed for the collection of such metrics in the prototyping hardware, will be oriented to the minimization of the FPGA resources. As much resources as possible will be shared with the acquisition of the event counts needed for the annotation of the power models.

Obviously, the hardware needed for probing and counting, will not be included in the HDL generation, when targeting VLSI implementation of the system.

3.4.2 FPGA implementation scripts

Provided scripts cover the entire workflow to implement developed hardware/software application on a FPGA platform: the whole toolchain will be based on commercial synthesis and implementation tools. The whole process will be tailored according to the device used and the physical memory available on the FPGA platform: little user intervention will be required, prior to the usage of scripts, to setup vital information otherwise unknown to the toolchain (such as FPGA chip model, clock frequency available on the board etc.)

Scripts will also take care of loading the software application binaries, embedding them into the bitstream obtained from hardware synthesis. The loading of the binaries relies on the knowledge of the precise mapping of the memory blocks onto the FPGA B-RAM primitives available on the target FPGA. The description of such memory layout will be generated automatically, according to the target device and to the size of the memory blocks under prototyping.

A bitstream file containing both the hardware implementation and the software application will be generated and used to program the FPGA board.

3.4.3 VLSI implementation scripts

Provided scripts cover the entire workflow to implement developed hardware/software application on an VLSI platform: the whole toolchain will be based on commercial synthesis and implementation tools.

The tools will create all necessary files for a full synthesis, timing, area and power optimization, and layout. Constraints input files for delay, power, area and (eventually) floorplanning will be considered. Advanced power saving methodologies and techniques such as CPF/UPF will be supported. As little user intervention as possible will be required, mostly covered in the form of additional (text) constraints files.

As is the case of the FPGA flow, scripts will also take care of loading the software application binaries, embedding them into the HDL instantiations for simulation.