MODELLING THE DESIGN METHODOLOGY FOR THE RISCO MICROPROCESSOR

por

Flávio Rech Wagner

RP-174 Março/92

Trabalho realizado com o apoio do CNPq.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
Av. Bento Gonçalves, 9500 - Agronomia
91501 - Porto Alegre - RS - BRASIL
Telefone: (051) 336-8399/339-1355 - Ramal 6161
Telex: (051) 2680 - CCUF BR
FAX: (051) 336-5576
E-MAIL: CPGCC@INF.UFRGS.BR

Correspondência: UFRGS-CPGCC
Caixa Postal 15064
91501 - Porto Alegre - RS - BRASIL

Editor: Ricardo Augusto da Luz Reis (interino)

UFRGS

CNPq 3.04.03.00-6

Abstract

This report presents the modelling of the methodology employed in designing the physical layout of the operational block of the RISCO microprocessor IC, under development at the UFRGS. The model uses the concepts of design methodology management supported by the STAR design framework. The design process is organized as an hierarchy of methodologies, according to the design alternatives and the architecture of the microprocessor component modules. Each methodology specializes a basic structure (a conceptual model), which organizes the various representations created for the design objects during the design process. The model also specifies all design tasks, in a condition-driven approach which relates tasks and design data qualities.

Keywords
VLSI design automation. Design methodology management. Design frameworks.

Resumo

Este relatório apresenta a modelagem da metodologia empregada no projeto do *layout* físico do bloco operacional do circuito integrado microprocessador RISCO, em desenvolvimento na UFRGS. O modelo usa conceitos de gerência de metodologias de projeto suportados pelo ambiente de projeto STAR. O processo de projeto é organizado como uma hierarquia de metodologias, em função das alternativas de projeto e da arquitetura dos módulos componentes do microprocessador. Cada metodologia especializa uma estrutura básica (um modelo conceitual), que organiza as várias representações criadas para os objetos de projeto ao longo do processo de projeto. O modelo também especifica todas as tarefas de projeto, através de uma abordagem dirigida por condições que relaciona tarefas e qualidades dos dados de projeto.

Palavras-chave
Automação do projeto de circuitos VLSI. Gerência de metodologias de projeto. Ambientes de projeto.

Contents

List of Figures

1 Introduction

This report presents the modelling of the methodology employed in designing the physical layout of the operational block of the RISCO microprocessor IC, under development at the UFRGS. The model uses the concepts of design methodology management supported by the STAR design framework. The design process is organized as an hierarchy of methodologies, according to the design alternatives and to the architecture of the microprocessor component modules. Each methodology specializes a basic structure (a conceptual model), which organizes the various representations created for the design objects during the design process. The model also specifies all design tasks, in a condition-driven approach which relates tasks and desired design properties.

This report does not intend to extensively present the STAR data model neither the STAR design methodology management mechanisms. The reader should refer to other reports for this purpose [1, 2], and a knowledge of these concepts will be presumed in this report. It intends only to illustrate the principles of the STAR design methodology management model, not the application of a concrete tool which implements this model. A complete description of the RISCO microprocessor is neither in the scope of this report. This can be found in [3].

In this initial section, we first introduce the main features of design frameworks and design methodology management. We then introduce the STAR framework and the RISCO microprocessor.

The remaining of this report is organized as follows. Section 2 briefly presents the overall behavioral, structural and floorplan design of the whole microprocessor, as well as the structural design of the operational block. The design methodology of the operational block of the RISCO microprocessor is then formally described in Section 3. Section 4 gives an overview of the whole design methodology, explaining it as an hierarchy of partial methodologies. Finally, Section 5 gives concluding remarks. The Appendix introduces new constructs of the Plasma language, a semi-formal system used in the specification of the STAR data model. These additional constructs are needed for the definition of particular conceptual models.

1.1 Design frameworks

Design frameworks aim at the integration of tools so as to guarantee the overall consistency of the process of designing circuits and systems and to provide a uniform interaction between the designers and the tools. Examples of frameworks that partially or totally support these goals are Oct [4], from Berkeley, Cadweld [5], from Carnegie-Mellon, and CWS [6], from the Cadlab in Germany, as well as comercial products, such as the Open Framework from Cadence and the ValidFrame from Valid.

The main feature of a design framework is the provision of a uniform data model for design data representation [7], which supports the representation of circuits and systems as complex objects, taking into account aspects like composition of subobjects, hierarchy, and instantiation of objects.

A design framework data model must provide facilities for multiple representations for a design object. In the scope of this report, we designate the organization of these multiple representations as the object *control structure*. Different representations for a design object can correspond to

- design alternatives (e.g. a standard-cell or a gate-array approach);
- design views, i.e. representations of the same object at different abstraction levels (algorithmic, RT, logic, layout, etc);
- design revisions, i.e. consecutive refinements or improvements of the same object.

A design methodology [8] is a set of design rules that either enforce or guide the design activities performed by the user, so as to obtain design objects with desired properties. Rules can express:

- tasks that must be executed when the design process arrives at a given point (this point can be for instance expressed in terms of some design object properties);
- alternative design approaches to be followed from a given design point, as well as criteria for deciding between the possible design paths (again, these crieria can involve design object properties);
- design representations that must be created under given conditions (e.g. a representation at a more detailed design level or alternatives that must be compared according to some trade-offs).

Design methodology management is the control of the creation of the design objects and of the execution of the design tasks so that they conform to the established rules.

1.2 The STAR framework

The development of the STAR framework is a joint effort of the UFRGS and the IBM Rio Scientific Center which is based on previous experience of these groups in the field of design frameworks (the AMPLO environment [9], at the UFRGS, and the GARDEN data model [10], at IBM). The STAR framework supports the most important features expected from systems effectively open to the integration of tools aimed at various applications, architectures, and technologies [11]. STAR is based on a data model [1] which is derived from the GARDEN model. It has been shown that the GARDEN model supports more flexible and powerful concepts than other frameworks [12]. The STAR framework also offers special facilities for data and design methodology management and for cooperation between designers.

The definition of a design methodology in the STAR framework is based on three main principles: the task flow, the control structures of the design objects, and the

hierarchization of the design methodologies. A detailed description of the STAR design methodology management model can be found in [2].

Task flow is expressed through a condition-driven model, where input-output relationships between tools and design data are specified. These relationships can involve "qualities" of the design objects (such as the values of certain object attributes). A tool is eligible for execution when its input data, with the desired qualities, is available. The choice among many executable tools is left to the user.

A design methodology is strongly related to control structures that organize all representations that can be created for the design objects of a given application according to a given management strategy. These control structures explicitly contain all object "qualities" that are needed for the task sequencing. Each design object can have a different control structure, depending on the particular design methodology to be applied to it. We call the *conceptual scheme* of the application the set of all object control structures defined for this application.

Design methodologies can be organized in a hierarchical way. A new design methodology can be derived from a previous one by specializing (either by extending or restricting) the control structures of the previous methodology, and by adding new tasks to the task flow specification.

Any user can also be an "application manager", that defines a new application as a specialization of the application that he/she is authorized to execute. The application manager must define, prior to the execution of a design methodology, both its conceptual scheme and the associated tasks. While defining the conceptual scheme, all objects and attributes that he/she already knows as necessary are specified (although attribute values may be assigned during the design process), specially those needed for specifying the task flow control. However, new objects and attributes may be still defined and created during the design process.

1.3 The RISCO microprocessor

RISCO [3] is a 32-bit microprocessor which is being developed at the UFRGS. It is being designed using 1.5 um CMOS design rules with 2 metal levels. It has a three-address architecture and executes one instruction per cycle at normal flow, except for memory references. Its main architectural features are:

- data, instructions, and addresses are 32-bit words;
- a 32-bit word is the basic addressable unit, thus giving access to 4 Gwords;
- communication with the main memory is done through a 32-bit multiplexed bus for data and address;
- it has 32 32-bit registers, including the Program Counter, the Stack Pointer, the Processor Status Word, and R0 (constant zero);
- it has a 3-stage pipeline, achieving a peak instruction rate of one instruction per machine cycle;

- branch instructions have their execution delayed by one cycle.

The RISCO microprocessor has four main architectural blocks: the Operational Part, which is composed of 32 almost identical 1-bit wide slices, the Control Part, a Validation Interface, which is responsible for synchronizing and buffering the control signals from the Control Part to the Operational Part, and the Clock Generator.

The overall layout contains the Clock Generator, to the left, and the Control Part, to the right, in a top slice. Below them are located first the Validation Interface and then the Operational Part, both as wide as the top slice. The Operational Part is divided into 32 1-bit wide horizontal slices. It is implemented in a bus-based architecture. Two data busses run horizontaly through all cells of each slice. Control lines run vertically, while the power lines also run horizontally.

2 Behavioral, structural, and floorplan design

This section only intends to introduce the early design phases of the RISCO microprocessor design, including the behavioral, structural, and floorplan design. The object control structures are only introduced when this is relevant for the physical layout design, to be detailed later. The tasks invoked during these design stages are not specified.

2.1 Behavioral design

The RISCO microprocessor is initially specified through a behavioral description, by using some hardware description language. This description states the behavior of the RISCO instruction set: how each instruction affects the microprocessor registers and flags and the memory contents. It does not contain any references to possible implementations for the microprocessor.

The methodology for the microprocessor behavioral design specifies the RISCO Design object and its initial control structure, including Views that contain behavioral related information. As a result of the design process, ViewStates for these Views are created. The RISCO object has three initial attributes, corresponding to design requirements to be set by the application manager: maximum area, maximum power dissipation, and minimum clock frequency. These attributes are defined at the root of the control structure. Ports that are visible in the initial specification of the microprocessor, such as data and address busses and external control lines, are created during the design process and defined at the root. The design methodology specifies that all Ports have attributes PortDirection and BitWidth.

2.2 Structural design

2.2.1 Structural decomposition of the microprocessor

A structural representation for the RISCO microprocessor is manually generated from the behavioral one. The circuit is partitioned into its four main structural blocks: the operational block OP, the control block CP, the clock generator ClockGen, and the validation interface ValidatInterf between CP and OP. The methodology for the structural design extends the RISCO control structure by adding a ViewGroup VG-RISCO-Struct, which gathers all Views of the RISCO object that correspond to the structural design. One of these Views (V-RISCO-Struct-Obj) is of type MHD and contains four DesignInstances that make reference to other Designs (OP, CP, ClockGen, and ValidatInterf).

The user which is responsible for the structural design extends the conceptual scheme, by adding these objects. Figure 1 shows the definition for object OP (the operational block whose design is the subject of this report) in the new scheme. [1]

[1] This definition is given in the Plasma specification language, extended as presented in the Appendix.

```
OP :- Design
    has A_Bus :- Port
        has BitWidth = { 31 .. 0 }
        has PortDirection = inout
    has BOUT_BB :- Port
        has PortDirection = in
    is generalization of { V-OP-Struct (View MHD) ;
                           V-OP-SlicedStruct (View MHD) ;
                           V-OP-FullCustom-ExpandedStruct (View MHD) ;
                           VG-OP-Layout (ViewGroup) }
```

Figure 1: The OP object

The design methodology previously specifies which attributes do the OP Ports have, but actual Ports are created only during the design process. Only the Ports A_Bus (a data bus) and BOUT_BB (a control line from CP) are shown as example, since OP has some dozen Ports. These Ports have attribute PortDirection, while A_Bus has an additional attribute BitWidth. A_Bus is a bundle of 32 lines, and each of them can be referred to as A_Bus [i]. The object OP does not have any attributes (UserFields) defined at the Design level (the root of the hierarchy of object representations).

The methodology for the structural design already specifies a View V-OP-Struct, of type MHD, which will contain the structural representation of OP. The ViewGroup VG-OP-Layout will be added later by the methodology for the floorplanning design (see subsection 2.3), while the MHD Views V-OP-FullCustom-ExpandedStruct and V-OP-SlicedStruct will be needed by the methodology for the full-custom layout design of OP.

It must be noted that Ports, in the STAR data model, are automatically inherited by all representations below the node of the control structure where they are defined. The inheritance of Parameters and UserFields is not automatic and must be explicitly defined.

Ports, Parameters, and UserFields can be changed during the design process, when the change is not forbidden by the design methodology. They can be created or removed, and their attributes can also be created or removed. Also the attribute values can be changed. Any of these changes is automatically stored by the framework in a new "version" of the control structure node where the Port, Parameter, or UserField is defined.

2.2.2 Structural decomposition of the operational block

The operational block OP is manually designed as an interconnection of modules, such as an ALU, a register file, other registers, and some logic. Each of these modules

processes information that is 32-bit wide. The structural partition of OP is stored in a MHD View V-OP-Struct, as shown in Figure 2.

```
V-OP-Struct :- View (MHD)
  contains ALU-Part :- Component
    has ... (same Ports as ALU)
    has reference to Design : ALU
  contains BReg-Part :- Component
    has ... (same Ports as BReg)
    has reference to Design : BReg
  ...
  contains ALU-PartA :- DesignInstance of ALU-Part
  contains BReg-PartA :- DesignInstance of BReg-Part
  ...
  contains ... (Nets)
```

Figure 2: The structural View of the operational part

This View has DesignInstances that make reference to new Designs ALU, BReg (the register file), PC (the Program Counter), UD (the barrel shifter), etc. These Designs were not known when the methodology for the structural design has been defined. The user which is responsible for the structural design of the operational block dynamically extends the conceptual scheme by adding the new objects ALU, BReg, PC, UD, etc. and by defining the control structures needed for them during this design stage.

Figure 3 shows the definition of object ALU (we will restrict this report to the design of this module inside OP). We show only the Ports A_Bus and RUA_BA, since the ALU has a large number of Ports.

```
ALU :- Design
  has RUA_BA :- Port
    has PortDirection = in
  has A_Bus :- Port
    has BitWidth = { 31 .. 0 }
    has PortDirection = inout
  is generalization of { V-ALU-BehavSrc (View HDL) ;
                         V-ALU-BehavObj (View HDL) ;
                         V-ALU-Struct (View MHD) }
```

Figure 3: The ALU object

HDL Views V-ALU-BehavSrc and V-ALU-BehavObj are needed for the simulation

of the structural representation of the microprocessor, while the MHD View V-ALU-Struct is used in the structural decomposition of the ALU, as shown in the next subsection.

Supposing that the designer wants to simulate the structural decomposition of the RISCO microprocessor achieved until this point, behavioral descriptions must be created for all leaves of the structural hierarchy under the RISCO View V-RISCO-Struct. This must be done for the Designs ALU, BReg, etc (modules within OP), CP, ClockGen, and ValidatInterf (other modules within RISCO). The behavioral descriptions are split into two Views, both of type HDL, one containing the source description and the other the result of the compilation process (see Figure 3 for the ALU).

Now a configuration can be built for the structural View of OP, selecting View-States containing behavioral descriptions for the modules within OP. After that, a configuration can also be established for the RISCO structural view, with a selection of ViewStates containing behavioral descriptions for CP, ClockGen, and ValidatInterf, and using the already built configuration for OP.

As a result of the simulation of this configuration, the designer may wish to create new ViewStates for all design objects involved in the configuration, in order to achieve the best result. New configurations do not need to be created for RISCO and OP, if the above mentioned ones are dynamic and specify the most recent ViewState of each of these design objects.

2.2.3 Structural decomposition of the ALU

Also the modules within the operational block must be designed as a structural decomposition of more primitive cells. In the following, we illustrate this by showing the main steps in the design of the ALU.

As a first step, a ViewState for the ALU View V-ALU-Struct is created, as shown in Figure 4. V-ALU-Struct, of type MHD, is an interconnection of 32 slices of 1-bit wide ALUs.

```
V-ALU-Struct :- View (MHD)
  contains ALUSlice-Part :- Component
           has ... (same Ports as ALUSlice)
           has reference to View : V-ALUSlice-Struct
  contains ALUSlice1 :- DesignInstance of ALUSlice-Part
  contains ALUSlice2 :- DesignInstance of ALUSlice-Part
  ...
  contains ALUSlice32 :- DesignInstance of ALUSlice-Part
  contains ... (Nets)
```

Figure 4: The ALU structural View

V-ALU-Struct refers to a MHD View V-ALUSlice-Struct of another Design ALU-Slice, shown in Figure 5. This Design must have the same Ports as the ALU object, except that the data bus lines are now 1-bit wide. Since it is previously known that this object will be needed, its initial control structure, containing only V-ALU-Slice-Struct, can be already specified by the design methodology. The ViewGroup VG-ALUSlice-LO and the MHD View V-ALUSlice-Extracted will be later added by the methodology for the ALU physical design.

```
ALUSlice :- Design
  has RUA_BA :- Port
    has PortDirection = in
  has A_Bus :- Port
    has PortDirection = inout
  has ... (same other Ports as ALU)
  has MaxDelay :- UserField : integer
  is generalization of { V-ALUSlice-Struct (View MHD) ;
                         V-ALUSlice-Extracted (View MHD) ;
                         VG-ALUSlice-LO (ViewGroup) }
```

Figure 5: The ALUSlice object

V-ALUSlice-Struct, shown in Figure 6, is an interconnection of instances of the most recent ViewState of structural representations of transistors and logic gates, which are supposed to exist in a cell library.

```
V-ALUSlice-Struct :- View (MHD)
  contains AND-Part :- Component
           has ... (Ports)
           has reference to most recent ViewState : V-ANDGate-Struct
  contains Transistor-Part :- Component
  ...
  contains AND1 :- DesignInstance of AND-Part
  ...
  contains ANDm :- DesignInstance of AND-Part
  contains Tr1 :- DesignInstance of Transistor-Part
  ...
  contains Trn :- DesignInstance of Transistor-Part
  ...
  contains ... (Nets)
```

Figure 6: The structural View of ALUSlice

2.2.4 The sliced structural representation of OP

A sliced representation of the operational block structure will be later needed for the layout design methodology. The operational block is seen as an array of 32 1-bit wide instances of an object OPSlice, as shown in Figure 7.

```
V-OP-SlicedStruct :- View (MHD)
  contains OPSlice-Part :- Component
             has ... (same Ports as OPSlice)
             has reference to Design : OPSlice
  contains OPSlice1 :- DesignInstance of OPSlice-Part
  contains OPSlice2 :- DesignInstance of OPSlice-Part
  ...
  contains OPSlice32 :- DesignInstance of OPSlice-Part
  contains ... (Nets)
```

Figure 7: The sliced structural representation of OP

Since the need for this decomposition of OP is known before the structural design, the object OPSlice, with its initial control structure (only the View V-OPSlice-Struct), is already defined in the design methodology for the structural decomposition, as shown in Figure 8. OPSlice has the same Ports as OP, except that the data bus lines are now 1-bit wide.

```
OPSlice :- Design
  has BOUT_BB :- Port
    has PortDirection = in
  has A_Bus :- Port
    has PortDirection = inout
  is generalization of { V-OPSlice-Struct (View MHD) ;
                         V-OPSlice-ExpandedStruct (View MHD) ;
                         V-OPSlice-LO (View LO) }
```

Figure 8: The OPSlice object

The MHD View V-OPSlice-Struct, shown in Figure 9, contains instances of the already existing structural representations of ALUSlice, BRegSlice, etc. The additional Views V-OPSlice-LO and V-OPSlice-ExpandedStruct will be added by the methodology for the operational block layout design.

```
V-OPSlice-Struct :- View (MHD)
  contains ALUSlice-Part :- Component
           has ... (same Ports as ALUSlice)
           has reference to View : V-ALUSlice-Struct
  contains BRegSlice-Part :- Component
           has ... (same Ports as BRegSlice)
           has reference to View : V-BRegSlice-Struct
  ...
  contains ALUSlice1 :- DesignInstance of ALUSlice-Part
  contains BRegSlice1 :- DesignInstance of BRegSlice-Part
  ...
  contains (Nets)
```

Figure 9: The structural representation of OPSlice

2.3 Floorplanning

The design methodology for the floorplanning process extends the control structures for the objects OP, CP, ClockGen, and ValidatInterf, already specified during the structural design, and defines an initial control structure for a new object Pads.

In the floorplanning process, the designer assigns initial values to the dimensions and positioning of the four main structural blocks of the RISCO microprocessor. The goal of the floorplanning process is to obtain the minimum possible overall area for the RISCO object. The further detailed layout design of the structural blocks may lead the designer to change these dimensions and positions. In order to accomplish the floorplanning task, a ViewState for a View V-RISCO-Layout, of type Layout, must be created for the RISCO Design (see Figure 10).

```
V-RISCO-Layout :- View (LO)
  has Width, Height :- UserField : integer
  contains OP-Part :- Component
    has ... (same Ports as OP)
  ... (same for CP, ClockGen, ValidatInterf)
  contains OP-Part1 :- DesignInstance of OP-Part
    has Width, Height :- UserField : integer
    has Coordinates :- UserField : { integer; integer }
  ... (same for CP, ClockGen, ValidatInterf)
```

Figure 10: The Layout View for the RISCO object

V-RISCO-Layout has attributes whose values are the width and the height of the overall microprocessor layout. These attributes are specified by the design method-

ology, but their values are assigned during the design process. The DesignInstances of V-RISCO-Layout correspond to the objects OP, CP, ClockGen, ValidatInterf, and Pads, and have three UserFields: width, height, and coordinates of the left bottom corner of the block. Since the partitioning of RISCO into these four modules is previously known, the design methodology could already enforce the existence of the respective DesignInstances. As a result of the floorplanning process, various ViewStates for V-RISCO-Layout can be created, with different values for width, height, and coordinates of its DesignInstances.

This design step also involves ViewGroups for the Designs OP, CP, ClockGen, ValidatInterf, and Pads, that will gather their layout Views. Figure 11 shows the ViewGroup VG-OP-Layout for the Design OP. It has two UserFields whose values are defined during the floorplanning process: width and height. The Ports now have additional methodology-defined attributes Width (the width of the layout track), Coordinates and ImplemLayer (the layout layer where the Port is implemented). At the layout level, each data bus (A_Bus and B_Bus) is implemented by two separate lines that carry complementary values. For the A_Bus, an additional Port A_Bus_not is thus created. Furthermore, since at the layout level each bit of the data busses must be separately routed, each bit of A_Bus has its own attributes. VG-OP-Layout also has additional Ports defined at this level, corresponding to the power supply lines. The existence of these Ports could be already enforced by the design methodology. The ViewGroups VG-OP-FullCustom and VG-OP-CellBased are added to the control structure by the methodologies for the OP layout design.

At the end of the floorplanning process, the attributes Width and Height of VG-OP-Layout must receive the values assigned to the width and height of the corresponding DesignInstance in the ViewState of V-RISCO-Layout which was considered to be the optimum one. One could, for instance, choose a ViewState with minimum overall area, but with a height for the operational block that is greater than a certain lower limit that is necessary for a good layout design. The values obtained for Width and Height will act as design constraints during the physical design of the operational block. Similar constraints are obtained for the design of the other blocks (CP, ClockGen, and ValidatInterf).

```
VG-OP-Layout :- ViewGroup
  has VDD :- Port
    has Width :- UserField : integer
    has Coordinates :- UserField : { integer; integer }
    has ImplemLayer :- UserField : Layer
  has GND :- Port
    has ... (same attributes as VDD)
  has BOUT_BB :- Port
    has Width :- UserField : integer
    has Coordinates :- UserField : { integer; integer }
    has ImplemLayer :- UserField : Layer
  has A_Bus [1] :- Port
    has Width :- UserField : integer
    has Coordinates :- UserField : { integer; integer }
    has ImplemLayer :- UserField : Layer
  has A_Bus_not [1] :- Port
    has Width :- UserField : integer
    has Coordinates :- UserField : { integer; integer }
    has ImplemLayer :- UserField : Layer
  has A_Bus [2] :- Port
    has ... (same attributes as A_Bus [1] )
  ... (same for A_Bus [3] until A_Bus [32] and
       for A_Bus_not [2] until A_Bus_not [32])
  has Width, Height :- UserField : integer
  is generalization of { VG-OP-FullCustom (ViewGroup) ;
                         VG-OP-CellBased (ViewGroup) }

Layer : string = { poly / diffusion / well / metal1 / metal2 / ...}
```

Figure 11: ViewGroup for the floorplanning of OP

3 Physical design of the operational block

Although two different design methodologies – full-custom and cell-based – can be tried in the search for an optimal solution for the operational block layout, this report will be restricted to the discussion of the full-custom design.

3.1 Splitting the operational block into slices

All representations for the full-custom design of the operational block are stored under VG-OP-FullCustom, shown in Figure 12. In the full-custom methodology, the operational block layout is designed as an array of 32 identical horizontal slices. This representation is stored in a Layout View V-OP-FullCustom-LO.

```
VG-OP-FullCustom :- ViewGroup
  has Width, Height :- UserField : integer
  has PowerBusWidth :- UserField : integer = value1
  has PowerBusDirection, DataBusDirection :- UserField :
                                          Direction = horizontal
  has ControlLinesDirection :- UserField : Direction = vertical
  is generalization of { V-OP-FullCustom-LO (View LO) ;
                         V-OP-FullCustom-ENL (View MHD) }
    {{ strict inherited Width, Height, PowerBusWidth,
       PowerBusDirection, DataBusDirection, ControlLinesDirection }}
```

Figure 12: ViewGroup for the full-custom design of the operational block

The full-custom design follows some guidelines that are established in attributes of VG-OP-FullCustom: the power supply and data bus lines must run in the horizontal direction, the control lines must run in the vertical direction, and the power supply lines must have a given width. This width receives an initial value which is used as a design constraint, but after the layout design, and knowing the real power consumption of the whole OP slices, the designer may need to change this value.

VG-OP-FullCustom also has UserFields Width and Height. As a design goal, their values must be less than the values of the corresponding attributes in VG-OP-Layout, that were established during the floorplanning process. Since the ALU design is the critical task in the overall OP design, the ALU layout will be designed first. The designer will try to adjust the width and height of the ALU slices to the constraints established in the floorplanning. He/she will try to design the ALU slice with a minimal height, but without resulting in a too much wide module. The height of the ALU slice will serve as a design constraint for all other modules within the OP slice.

3.2 Designing the ALU slices

The ViewGroup VG-ALUSlice-LO gathers all layout related information about the ALU slices and presents attributes, shown in Figure 14, that are needed in order to guarantee the overall consistency of the layout design of the operational block slices:

- height – its value must be as aproximate as possible to 1/32 of the height of the operational block (attribute of VG-OPSlice-LO);
- power supply, data bus, and control lines attributes related to the track width and direction – their values must be the same of the corresponding attributes of VG-OP-FullCustom.

For each of the ALU slice interface signals the following attributes are defined: coordinates, implementation layer, and width of the channel layout.

Furthermore, the data busses traverses the operational block horizontally, so that coordinates must be defined for their left and right interface points, while some control lines, such as RUA_BA, traverse the ALU slices vertically, so that coordinates must be defined for their top and bottom interface points.

The values of these interface signal attributes will be determined by the ALUSlice layout design task.

There are two possible methodologies for the design of the ALU slices: manual or through a module generator. Representations for each of them are gathered in a ViewGroup (either VG-ALUSlice-Manual or VG-ALUSlice-ModGener) defined under VG-ALUSlice-LO, as shown in Figure 14. We will detail only the methodology for the manual design. Figure 13 shows the definition of VG-ALUSlice-Manual.

```
VG-ALUSlice-Manual :- ViewGroup
  is generalization of { V-ALUSlice-Manual-LO (View LO) ;
                         V-ALUSlice-Manual-ENL (View MHD) }
```

Figure 13: The ViewGroup for the manual design of the ALU slice

The manual design proceeds through six steps: layout generation, design rule checking, netlist extraction, netlist comparison, capacitance extraction, and timing evaluation. They are described next.

During the design of the ALU slice layout, many Auxiliary Objects are used: DesignRules-1.5 (file with design rules for the 1.5 micra technology), IdentifTrans-1.5 (file with knowledge about the identification of transistors and connections for the 1.5 micra technology), ComparTrans-1.5 (file with information about the netlist comparison for the 1.5 micra technology), StimuliFile and ResultFile (both containing waveforms for signals in the ALU slice).

```
VG-ALUSlice-LO :- ViewGroup
  has Height, Width :- UserField : integer
  has PowerBusWidth :- UserField : integer { value .. value }
  has PowerBusDirection, DataBusDirection, ControlLinesDirection
    :- UserField : Direction = { horizontal, vertical }
  has RUA_BA :- Port
    has ImplemLayer :- UserField : Layer
    has Width :- UserField : integer
    has TopInterface :- UserField
      has Coordinates :- UserField : { integer; integer }
    has BottomInterface :- UserField
      has Coordinates :- UserField : { integer; integer }
  has A_Bus :- Port
    has ImplemLayer :- UserField : Layer
    has Width :- UserField : integer
    has LeftInterface :- UserField
      has Coordinates :- UserField : { integer; integer }
    has RightInterface :- UserField
      has Coordinates :- UserField : { integer; integer }
  has A_Bus_not :- Port
    has ImplemLayer :- UserField : Layer
    has Width :- UserField : integer
    has LeftInterface :- UserField
      has Coordinates :- UserField : { integer; integer }
    has RightInterface :- UserField
      has Coordinates :- UserField : { integer; integer }
  has VDD :- Port
    ... (same attributes as A_Bus)
  has GND :- Port
    ... (same attributes as A_Bus)
  is generalization of { VG-ALUSlice-Manual (ViewGroup) ;
                         VG-ALUSlice-ModGener (ViewGroup) }
    {{ strict inherited Height, Width,
                        PowerBusWidth, PowerBusDirection,
                        DataBusDirection, ControlLinesDirection }}
```

Figure 14: The attributes of the ALU slices

Layout generation
With the mask editor configurated for a 1.5 micra grid, a ViewState for a Layout View V-ALUSlice-Manual-LO is created, as shown in Figure 15. This View has an attribute LayoutOK, and inherits attributes relative to the power, data, and control lines from VG-ALUSlice-LO.

```
task: ALU layout generation
tool: mask-editor [ configuration = 1.5 micra ]
input: ViewState for V-ALUSlice-Struct
output: ViewState for V-ALUSlice-Manual-LO
        Correlation: ViewState of V-ALUSlice-Manual-LO
                     manually generated from
                     ViewState of V-ALUSlice-Struct
goals: Height =< 1/32 V-RISCO-Layout . OP-Part1 . Height
       (OP-Part1 is a DesignInstance within V-RISCO-Layout
       corresponding to the operational block)
```

Figure 15: Layout generation for the ALU slice

This task defines the width and height of the ALU slice (attributes of V-ALUSlice-Manual-LO whose existence is inherited from VG-ALUSlice-LO), as well as the coordinates and the implementation layer of its interface signals. The layout must follow the design guidelines, relative to the power, data, and control lines, defined in a previous task (see Figure 16). [2]

```
task: ALU slice layout generation guidelines assignment
tool: data-manager
input: VG-OP-FullCustom
output: VG-ALUSlice-LO . PowerBusWidth :=
                         VG-OP-FullCustom . PowerBusWidth
        ... (same for PowerBusDirection, DataBusDirection,
             ControlLinesDirection)
goals: none
```

Figure 16: ALU slice layout generation guidelines

As a design goal, the height of the ALU slice must be smaller than 1/32 of the height of the operational block, established during the floorplanning. If this is not achieved, either the ALU slice layout must be redesigned or the floorplanning must

[2]The "data manager" used in the task specification of Figure 16 is a tool which allows the designer (or the design methodology manager) to directly operate upon the objects in the STAR data base.

be re-avaluated. The task also establishes a correlation between ViewStates of V-ALUSlice-Manual-LO and V-ALUSlice-Struct.

The control structure for VG-ALUSlice-LO already enforces that both interface points of A_Bus have the same implementation layer and the same track width. The same occurs for RUA_BA. The desired cell simmetry is however not completely enforced, since each interface point of A_Bus (and of RUA_BA) has its own coordinates. Only during the composition of the OP slice the complete simmetry will be thus verified.

Design rule checking

With the design rule checker also configurated for the 1.5 micra technology, the Layout View is checked. This task assigns a value (either *true* or *false*) to the attribute LayoutOK of V-ALUSlice-Manual-LO (see Figure 17). Since V-ALUSlice-Manual-LO is a primitive cell, with no references to other design objects, we do not need to specify a configuration for its processing.

```
task: ALU slice design rule checking
tool: design-rule-checker [ configuration = 1.5 micra ]
input: ViewState for V-ALUSlice-Manual-LO
       DesignRules-1.5 (file with design rules for the technology)
output: none
goals: V-ALUSlice-Manual-LO . LayoutOK = true
```

Figure 17: Design rule checking for the ALU slice

Netlist extraction

An electrical netlist extractor, configurated for the 1.5 micra technology, extracts a netlist from a good layout, generating a ViewState for a MHD View V-ALUSlice-Manual-ENL (see Figure 18).

```
V-ALUSlice-Manual-ENL :- View (MHD)
  has NetListOK :- UserField : boolean
  contains ... (Components)
    has reference to most recent ViewState : ...
  contains ... (DesignInstances)
  contains ... (Nets)
```

Figure 18: The netlist representation for the ALU slice

This View contains instances of objects defined in a cell library, making reference to their most recent ViewStates, and has an attribute NetListOK. This task, shown in Figure 19, also establishes a correlation between ViewStates of V-ALUSlice-Manual-LO and V-ALUSlice-Manual-ENL.

```
task: netlist extraction for the ALU slice
tool: ENL-extractor [ configuration = 1.5 micra ]
input: ViewState for V-ALUSlice-Manual-LO [ LayoutOK = true ]
       IdentifTrans-1.5 (technology file with the knowledge about
                         identification of transistors and connections)
output: ViewState for V-ALUSlice-Manual-ENL
        Correlation: ViewState of V-ALUSlice-Manual-ENL
                     extracted from
                     ViewState of V-ALUSlice-Manual-LO
goals: none
```

Figure 19: Extracting the netlist for the ALU slice

Netlist comparison

An electrical netlist comparator compares the extracted netlist with the structural design of the ALU slice (View V-ALUSlice-Struct, see Figure 6). This task, shown in Figure 20, assigns a value (either *true* or *false*) to the attribute NetListOK of V-ALUSlice-Manual-ENL. Since both V-ALUSlice-Struct and V-ALUSlice-Manual-ENL make reference to the most recent ViewState of objects assigned to their Components, ConfigurationDefinitions are not needed for their processing. If the task goal (NetListOK = true) is not achieved, the designer must create a new layout for the ALU slice and re-execute the previous tasks (DRC and netlist extraction).

```
task: netlist comparison for the ALU slice
tool: ENL-comparator
input: ViewState for V-ALUSlice-Manual-ENL,
       ViewState for V-ALUSlice-Struct,
       ComparTrans-1.5 (technology file with information
                        about the comparison)
output: none
goals: V-ALUSlice-Manual-ENL . NetListOK = true
```

Figure 20: Netlist comparison for the ALU slice

Capacitance extraction

The capacitance of some important ALU signals (those that drive the data bus lines, the carry line, the control lines) is calculated, as shown in Figure 21. These values are back-annotated to a structural representation V-ALUSlice-Extracted of the ALU slice. This View, besides the back-annotated capacitances, is identical to V-ALUSlice-Struct. The task establishes correlations between a ViewState of V-ALUSlice-Extracted and ViewStates of both V-ALUSlice-Manual-LO and V-ALUSlice-Struct.

```
task: capacitance extraction for the ALU slice
tool: ENL-extractor [ configuration = 1.5 micra,
                      extract parasitic capacitances ]
input: ViewState for V-ALUSlice-Manual-LO
       ViewState for V-ALUSlice-Struct
output: ViewState for V-ALUSlice-Extracted
        Correlation: ViewState of V-ALUSlice-Extracted created from
                     ViewState of V-ALUSlice-Struct
        Correlation: ViewState of V-ALUSlice-Extracted
                     contains parasitic capacitances from
                     ViewState of V-ALUSlice-Manual-LO
goals: none
```

Figure 21: Capacitance extraction for the ALU slice

Timing evaluation

By electrically simulating the structural representation V-ALUSlice-Extracted with the calculated capacitances, the maximum delay of the ALU slice (attribute MaxDelay of ALUSlice, see Figure 5) is evaluated, as shown in Figure 22, and compared to an expected value. Other timing estimates from the structural design are also confirmed, such as the delay in the carry line.

Many simulation runs are needed, each using a different stimuli file. The creation of these files is not shown in this report. We suppose that a certain number of such files has been created for exercizing the ALU slice. They are bound to this object through correlations. Result files generated by the simulation runs are bound through correlations to both the simulated ViewState of the ALU slice and the stimuli file used in the simulation run.

As for V-ALUSlice-Struct, from which it is derived, V-ALUSlice-Extracted makes reference to the most recent ViewState of the objects assigned to their Components, so that it can be processed without specifying a ConfigurationDefinition.

task: timing evaluation of the ALU slice
tool: electrical-simulator
input: ViewState for V-ALUSlice-Extracted
StimuliFile [correlated with ALUSlice]
output: ResultFile
Correlation : ResultFile x ViewState of V-ALUSlice-Extracted
Correlation : ResultFile x StimuliFile
goals: ALUSlice . MaxDelay < expected value
... (other timing estimates)

Figure 22: Timing evaluation of the ALU slice

3.3 Designing the other operational block modules

For all other modules of the operational block slice (the register file Breg, the barrel shifter UD, the program counter PC, etc) a layout is now generated, by using the height of the ALU slice as a constraint and by trying to get the less wider modules as possible. The coordinates and widths of the horizontal busses (A_Bus, A_Bus_not, B_Bus, B_Bus_not, VDD, and GND), that have been defined during the ALU slice layout design, must be identical in all other modules.

The design methodology must guarantee that the guidelines for power supply, data bus, and control lines, established in VG-OP-Layout, are followed in all these modules.

3.4 Composing the operational block

Now that the layouts of all modules of an operational block slice have been designed, they are composed to create the complete layout of a slice. The slices are then composed into the complete layout of the operational block.

Composing the operational block slice
The layout of each operational block slice is designed as an abutment of layouts for slices of the modules ALU, BReg, etc. The composition of the operational block slice is performed by two basic tasks.

In the first task, shown in Figure 24, a ViewState for a Layout View V-OPSlice-LO (see Figure 23) is created. This View describes the layout of OPSlice as a composition of the layouts of the modules ALUSlice, BRegSlice, etc. As already explained for the ALU slice design, new Ports appear at the layout representation (VDD, GND, A_Bus_not), and new Port UserFields are added (ImplemLayer, Width, and Coordinates).

```
V-OPSlice-LO :- View (LO)
  has Width, Height :- UserField : integer
  has PowerBusWidth :- UserField : integer = { value .. value }
  has BOUT_BB :- Port
    has ImplemLayer :- UserField : Layer
    has Width :- UserField : integer
    has Coordinates :- UserField : { integer; integer }
  has A_Bus :- Port
    has ImplemLayer :- UserField : Layer
    has Width :- UserField : integer
    has LeftInterface :- UserField
      has Coordinates :- UserField : { integer; integer }
    has RightInterface :- UserField
      has Coordinates :- UserField : { integer; integer }
  has A_Bus_not :- Port
    has ImplemLayer :- UserField : Layer
    has Width :- UserField : integer
    has LeftInterface :- UserField
      has Coordinates :- UserField : { integer; integer }
    has RightInterface :- UserField
      has Coordinates :- UserField : { integer; integer }
  has VDD :- Port
    ... (same attributes as A_Bus)
  has GND :- Port
    ... (same attributes as A_Bus)
  contains ALUSlice-Part :- Component
            has Width, Height :- UserField : integer
            has ... (same Ports as VG-ALUSlice-LO)
            has reference to View : V-ALUSlice-Manual
  contains BRegSlice-Part :- Component
            has Width, Height :- UserField : integer
            has ... (same Ports as VG-BRegSlice-LO)
            has reference to View : V-BRegSlice-Manual
           ...
  contains ALUSlice-PartA :- DesignInstance of ALUSlice-Part
            has Width, Height :- UserField : integer
            has Coordinates : UserField : { integer, integer }
  contains BRegSlice-PartA :- DesignInstance of BRegSlice-Part
            has Width, Height :- UserField : integer
            has Coordinates : UserField : { integer, integer }
```

Figure 23: The layout of the operational block slice

The designer has to decide about the relative positioning of the modules inside the operational block slice. There are two modules (an input and an output register) whose positions are important because of their connections to the pads. The other modules can be positioned anywhere, because of the bus architecture, which makes the modules independent from each other. There is no routing between the modules. With the mask editor, the designer thus assigns values to the attributes Coordinates of each DesignInstance of the OP slice layout.

Since Components in this View refer to the Views V-ALUSlice-Manual, V-BRegSlice-Manual, etc, and in these Views values have been assigned to the attributes Width, ImplemLayer, and Coordinates of each of their interface signals, these values are inherited by these Components and by the DesignInstances that instantiate them.

```
task: layout composition for the operational block slice
tool: mask-editor [ configuration = 1.5 micra ]
input: ViewState for V-ALUSlice-Manual-LO
       ... (the same for BRegSlice, PCSlice, UDSlice, etc)
output: ViewState for V-OPSlice-LO
        (results in V-OPSlice-LO . Width :=
          ALUSlice-PartA . Width + BRegSlice-PartA . Width + ...)
        (results in V-OPSlice-LO . Height := ALUSlice-PartA . Height)
goals: none
```

Figure 21: Composing an operational block slice

In the second task, a ConfigurationDefinition C-OPSlice-LO is created for a ViewState of the View V-OPSlice-LO, as shown in Figure 25. This configuration selects a Layout ViewState for each Component in the View. Since V-OPSlice-LO already statically selects Views for these Components (e.g. V-ALUSlice-Manual for the Component that corresponds to the ALU slice), the configuration must select a ViewState for this View (e.g. the narrowest ViewState).

```
task: configuration of the composition of the OP slice layout
tool: configurator
input: ViewState for V-OPSlice-LO
output: C-OPSlice-LO for ViewState of V-OPSlice-LO
configuration criterium: ViewState with minimum Width
goals: C-OPSlice-LO . Width =< V-RISCO-Layout . OP-Part1 . Width
       (OP-Part1 is a DesignInstance within V-RISCO-Layout
       corresponding to the operational block)
```

Figure 25: Configuration of the composition of the OP slice layout

It must be noted that the ConfigurationDefinitions inherit the attributes of the objects from which they are derived. In this case, C-OPSlice-LO has the attribute Width as V-OPSlice-LO. As a design goal, the designer has to obtain an OP slice with a Width smaller than the Width established as a design constraint during the floorplanning. If this is not achieved, either the floorplanning has to be re-evaluated or the layout of one or several modules of the OP slice have to be re-designed.

Composing the operational block

Now that the slice layout is composed, 32 slices are put together to create the layout of the whole operational block. We are simplifying the real procedure, assuming that the 32 slices are identical. In fact, the calculation of the ALU overflow asks for differences in the two topmost slices.

The View V-OP-FullCustom-LO defines a Component OPSlice-Part from which 32 DesignInstances are created (see Figure 26), and that makes reference to the most recent ViewState of V-OPSlice-LO. These DesignInstances have an attribute Height that is inherited from V-OPSlice-LO. As a result of the task, the height of the operational block, stored as an attribute of VG-OP-FullCustom, is calculated as the sum of the heights of the DesignInstances, as shown in Figure 27.

```
V-OP-FullCustom-LO :- View (LO)
  has LayoutOK :- UserField : boolean
  contains OPSlice-Part :- Component
              has Width, Height :- UserField : integer
              has ... (same Ports as OPSlice)
              has reference to most recent ViewState : V-OPSlice-LO
  contains OPSlice1 :- DesignInstance of OPSlice-Part
              has Width, Height :- UserField : integer
              has Coordinates :- UserField : { integer; integer }
  contains OPSlice2 :- DesignInstance of OPSlice-Part
              has Width, Height :- UserField : integer
              has Coordinates :- UserField : { integer; integer }
  ...
  contains OPSlice32 :- DesignInstance of OPSlice-Part
              has Width, Height :- UserField : integer
              has Coordinates :- UserField : { integer; integer }
```

Figure 26: The full-custom layout of the operational block

In the composition of the OP layout, we decided to use a dynamic configuration for V-OP-FullCustom-LO that automatically selects the most recent ViewState for the layout of OPSlice. However, since this ViewState is a composition of layouts of the modules ALUslice, BRegSlice, etc, many configurations may have been created for it, using different ViewStates for V-ALUSlice-Manual, V-BRegSlice-Manual, etc.

task: composition of the OP layout
tool: mask-editor [configuration = 1.5 micra]
input: VG-OP-FullCustom,
ViewState for V-OPSlice-LO
output: ViewState for V-OP-FullCustom-LO
(results in VG-OP-FullCustom . Height := OPSlice1 . Height +
OPSlice2 . Height + ... + OPSlice32 . Height)
goals: none

Figure 27: Composing the full-custom layout of the operational block

A ConfigurationDefinition must be thus defined for V-OP-FullCustom-LO, in order to verify it, selecting a particular configuration for the most recent ViewState of V-OPSlice-LO. Figure 28 shows that the configurator uses as criterion the selection of the most recent configuration for this ViewState.

task: configuration for the composition of the OP layout
tool: configurator
input: ViewState for V-OP-FullCustom-LO
output: C-OP-FullCustom-LO for ViewState of V-OP-FullCustom-LO
configuration criterium: most recent ConfigurationDefinition
for the Viewstate of V-OPSlice-LO
goals: none

Figure 28: Configuration for the composition of the OP layout

3.5 Verifying the layout composition of OP

Now the layout composed for the operational block in the configuration C-OP-Full-Custom-LO can be processed. This processing will check the overall layout, structure, and timing. Because of the unavailability of hierarchical tools, this checking will be done for a complete flattened structure. More powerful hierarchical tools could restrict the checking to the interconnections between the modules, disregarding their internal details. Furthermore, since the available electrical simulator would consume an unaffordable time for an overall simulation, the timing evaluation will be done only for the operational block slices.

Design rule checking

As shown in Figure 29, the design rule checker assigns a value (either *true* or *false*) to the attribute LayoutOK of C-OP-FullCustom-LO. It must be noted that C-OP-FullCustom-LO inherits the attribute LayoutOK from V-OP-FullCustom-LO. This task will verify if the modules ALUSlice, BRegSlice, etc. have been designed with vertical simmetry, i.e. if control lines have the same top and bottom x-coordinates. The control structure already enforces the same implementation layer and channel width for the top and bottom interface points.

```
task: design rule checking for the operational block
tool: design-rule-checker [ configuration = 1.5 micra ]
input: C-OP-FullCustom-LO
       DesignRules-1.5 (file with design rules for the technology)
output: none
goal: C-OP-FullCustom-LO . LayoutOK = true
```

Figure 29: Design rule checking for the operational block

Netlist extraction

As shown in Figure 30, the electrical netlist extractor, configurated for the 1.5 micra technology, creates, from C-OP-FullCustom-LO, a ViewState for the MHD View V-OP-FullCustom-ENL under VG-OP-FullCustom. The task also establishes a correlation between the ViewState of V-OP-FullCustom-ENL and the configuration C-OP-FullCustom-LO.

```
task: netlist extraction for the operational block
tool: ENL-extractor [ configuration = 1.5 micra ]
input: C-OP-FullCustom-LO [ LayoutOK = true ]
       IdentifTrans-1.5 (technology file with knowledge about
                         identification of transistors and connections)
output: ViewState for V-OP-FullCustom-ENL
        Correlation: ViewState of V-OP-FullCustom-ENL
                     extracted from C-OP-FullCustom-LO
goals: none
```

Figure 30: Netlist extraction for the operational block

Flattening the operational block

A flattened structural representation for the operational block, without hierarchy, is needed for the comparison with the extracted netlist. This representation is created from V-OP-Struct by an hierarchy expander and stored in a MHD View V-OP-FullCustom-ExpandedStruct under the OP Design, as shown in Figure 31. The hierarchy expander must perform configuration functions, since it must select descriptions for the DesignInstances within V-OP-Struct. The task also establishes a correlation between the ViewStates of V-OP-FullCustom-ExpandedStruct and V-OP-Struct.

task: flattening of the operational block
tool: hierarchy-expander
input: ViewState for V-OP-Struct
output: ViewState for V-OP-FullCustom-ExpandedStruct
Correlation: ViewState of V-OP-FullCustom-ExpandedStruct
expanded from ViewState of V-OP-Struct
goals: none

Figure 31: Flattening of the operational block

Netlist comparison

The netlist comparator compares V-OP-FullCustom-ENL with V-OP-FullCustom-ExpandedStruct, as shown in Figure 32, assigning a value (either *true* or *false*) to the attribute NetListOK of V-OP-FullCustom-ENL.

task: netlist comparison for the operational block
tool: ENL-comparator
input: ViewState for V-OP-FullCustom-ENL
ViewState for V-OP-FullCustom-ExpandedStruct
ComparTrans-1.5 (technology file with information
about the comparison)
output: none
goals: V-OP-FullCustom-ENL . NetListOK = true

Figure 32: Netlist comparison for the operational block

Flattening the operational block slice

Since the timing evaluation will be performed upon the operational block slice, and not upon the whole operational block, a flattened structural representation for the slice must be created. As shown in Figure 33, the hierarchy expander creates, from

V-OPSlice-Struct, a ViewState for V-OPSlice-ExpandedStruct, which seats directly under the OPSlice Design. The task establishes a correlation between ViewStates of V-OPSlice-ExpandedStruct and V-OPSlice-Struct.

task: flattening of the OP slice
tool: hierarchy-expander
input: ViewState for V-OPSlice-Struct
output: ViewState for V-OPSlice-ExpandedStruct
Correlation: ViewState of V-OPSlice-ExpandedStruct
expanded from
ViewState of V-OPSlice-Struct
goals: none

Figure 33: Flattening of the operational block slice

Capacitance extraction

The netlist extractor, configurated for the 1.5 micra technology, extracts parasitic capacitances from the layout of the operational block slices, and back-annotates this information to the flattened structural representation V-OPSlice-ExpandedStruct, as shown in Figure 34. This task establishes a correlation between the ViewStates of V-OPSlice-ExpandedStruct and V-OPSlice-LO.

task: capacitance extraction for the OP slice
tool: ENL-extractor [configuration = 1.5 micra,
extract parasitic capacitances]
input: ViewState for V-OPSlice-LO
output: ViewState for V-OPSlice-ExpandedStruct
Correlation: ViewState of V-OPSlice-ExpandedStruct
contains parasitic capacitances from
ViewState of V-OPSlice-LO
goals: none

Figure 34: Capacitance extraction for the operational block slice

Timing evaluation

The electrical simulator is run, in order to confirm the early timing estimates for the operational block slices, as shown in Figure 35. We do not need to specify a configuration for V-OPSlice-ExpandedStruct, since it is supposed that this description makes reference to the most recent ViewState of the basic Designs (gates and transistors).

Many simulation runs are needed, each using a different stimuli file and generating a different result file.

task: timing evaluation of the OP slice
tool: electrical-simulator
input: ViewState for V-OPSlice-ExpandedStruct
StimuliFile [correlated with OPSlice]
output: ResultFile
Correlation : ResultFile x ViewState of V-OPSlice-ExpandedStruct
Correlation : ResultFile x StimuliFile
goals: none

Figure 35: Timing evaluation of the operational block slice

4 Hierarchical derivation of the design methodology

The complete design methodology for the microprocessor design may be defined in a hierarchical way. Starting with an initial design methodology, each new methodology is derived from a previous one by specializing already existing control structures, defining control structures for new objects, and adding new tasks.

Hierarchization of design methodologies may serve different purposes:

- A designer sees only the design object representations and tasks defined for the design methodology he/she is using. Objects and tasks defined for other methodologies, such as those derived from the current one, are hidden from him/her.
- The design can start as soon as an initial methodology is established for the first design steps. Design methodologies for certain sub-objects or for certain specialized design activities (test generation, for instance) can be derived later on, as the design proceeds.
- New tools can be integrated into the design environment without disturbing already existing design activities. These tools may handle new design representations and/or auxiliary objects to be defined.

Figure 36 shows a possible hierarchical derivation for the complete RISCO design methodology. In the following, we briefly summarize methodologies $M1$ thru $M4$, which have been covered by this report.

M0. RISCO behavior
  M1. RISCO structure
    M2. RISCO floorplanning
      M3. OP full-custom
        M4. ALU slice manual
        M5. ALU slice with module generator
      M6. OP cell-based
      M7. CP random logic
      M8. CP PLA-based

Figure 36: Hierarchy of design methodologies

M0: **Behavioral design** Methodology $M0$ is the initial methodology for the behavioral design of the RISCO microprocessor. It defines an initial control structure for the RISCO object, containing only a behavioral HDL View for it. Ports that

are visible in the initial specification of the microprocessor, such as data and address busses and control lines for external communication with memory and I/O devices, are defined at the control structure root. Tasks in $M0$ include behavior definition (with a text editor), compilation (with the compiler for a hardware description language), and simulation (with the simulator for this language).

$M1$: **Structural design** A methodology $M1$ for the microprocessor structural design is derived from $M0$. It adds a structural MHD View to the RISCO control structure and defines four new objects OP, CP, ValidatInterf, and ClockGen, with their respective initial control structures. These control structures contain behavioral HDL Views. Ports corresponding to the control signals between OP and CP are already defined. For the OP object, $M1$ defines a structural MHD View for representing it as an interconnection of objects ALU, BReg, PC, UD, etc, while for each of these additional objects structural representations (as interconnections of gates and/or transistors) must be also defined and created. The initial control structures for these additional objects must be thus defined during the execution of the methodology $M1$. Tasks include the structural decomposition and simulation of RISCO and OP.

$M2$: **Floorplanning design** For the floorplanning design of the microprocessor, established in the methodology $M2$, the already existing control structures must be refined. A Layout View is defined for RISCO. ViewGroups for gathering layout related information are defined for OP, CP, ValidatInterf, and ClockGen. For all these objects, the data and address busses are decomposed into 32 lines, additional Ports corresponding to the complementary bus lines and to the power supply lines are added, and Ports have new, layout related attributes. Tasks that are related to the floorplan edition and verification are included.

From $M2$, four different design methodologies can be derived, corresponding to the full-custom and cell-based design of OP and to the random logic and PLA-based design of CP.

$M3$: **Full-custom design of the operational block** Methodology $M3$, which is devoted to the full-custom design of the operational part, refines the OP control structure with new ViewGroups and Views, such as VG-OP-FullCustom, V-OP-FullCustom-LO, and V-OP-FullCustom-ENL. It also defines new objects OPSlice and ALUSlice and their control structures, and extends the already existing control structure for ALU. Tasks include the layout composition and verification of OPSlice as an abutment of layouts for ALUSlice, BRegSlice, etc, and the layout composition and verification of OP as an abutment of 32 instances of OPSlice. Auxiliary objects for these tasks (technology files, for instance) are defined for $M3$.

$M4$: **ALU slice manual design** Methodology $M3$ depends on the existence of either $M4$ or $M5$, which allow the layout design of ALUSlice. In the case of $M4$, for

instance, VG-ALUSlice-Manual, V-ALUSlice-Manual-LO, and V-ALUSlice-Manual-ENL are added to the control structure of ALUSlice. New tasks are defined for the layout generation, design rule checking, netlist extraction, netlist comparison, capacitance extraction, and timing evaluation of ALUSlice. Some Auxiliary Objects already defined for $M3$ are also used within $M4$, although new ones must be created, such as stimuli files for the electrical simulation of ALUSlice.

5 Concluding remarks

This report presented in detail the formalization of the design methodology for a microprocessor layout, according to the design methodology management model defined for the STAR framework.

The example illustrated all the main features of this model:

- a design methodology is defined by a conceptual scheme and a set of tasks;
- the conceptual scheme is a set of control structures for the various design objects – a control structure organizes the several representations to be created for an object along the design process and establishes relationships between them;
- tasks are defined through a condition-driven approach, where input conditions for the task execution and task goals are specified by assertions about design qualities;
- design methodologies can be hierarchically derived from each other, by defining new objects and their control structures, by extending already existing control structures, and by defining new tasks;
- the designer is constrained to use the design object representations and tasks defined for the current design methodology, but he/she may have the right to dinamically add new objects and representations as they are needed during the design process.

It has been shown [2, 8] that the combination of these features is not found on other systems. Most other systems base design methodology management on task flow control, which is oriented towards design guidance. Even though they offer sophisticated task control facilities, these are not coupled with a powerful design data representation model, which adds automatic design consistency to the design guidance. The STAR design data model allows the definition of control structures which are strongly related to design methodology management, since they organize all representations to be created during the design process according to methodology-specific strategies.

Future work include testing the proposed design methodology management model in other applications, such as formal hardware verification, and then designing and implementing concrete mechanisms on top of the already existing STAR resources.

Acknowledgments

I acknowledge the valuable help of Luigi Carro, who gave me a complete and detailed description of the methodology employed in the layout design of the RISCO microprocessor.

References

[1] F.R. Wagner. The data model of the STAR framework. Technical report, CPGCC / UFRGS, Porto Alegre, 1991.

[2] F.R. Wagner and A.H. Viegas de Lima. Design methodology management in the STAR framework. Technical report, CPGCC / UFRGS, Porto Alegre, 1991.

[3] A. Junqueira, A. Suzim, C. Marcon, M. Dossa, and L. Cleto. RISCO – microprocessador CMOS 32 bits. Technical report, CPGCC / UFRGS, Porto Alegre, 1990.

[4] D.S. Harrison et al. Data management and graphics editing in the Berkeley Design Environment. In *International Conference on Computer Aided Design.* IEEE, 1986.

[5] J. Daniell and S.W. Director. An object-oriented approach to CAD tool control within a design framework. In *26th Design Automation Conference.* ACM/IEEE, 1989.

[6] K. Gottheil et al. The CADLAB workstation CWS – an open, generic system for tool integration. In F.J. Rammig, editor, *IFIP Workshop on Tool Integration and Design Environments.* North-Holland, 1988.

[7] F.R. Wagner. Modelos de representação e gerência de dados em ambientes de projeto de sistemas digitais. Technical Report CCR-121, IBM Rio Scientific Center, Rio de Janeiro, 1991.

[8] F.R. Wagner. Design methodology management in design frameworks. Technical report, CPGCC / UFRGS, Porto Alegre, 1991.

[9] F.R. Wagner, C. Freitas, and L.G. Golendziner. The AMPLO system – an integrated environment for digital systems design. In F.J. Rammig, editor, *IFIP Workshop on Tool Integration and Design Environments.* North-Holland, 1988.

[10] E.B. de la Quintana, G.O. Annarumma, and P. Molinari Neto. GARDEN – the Design Data Interface. Technical Report CCR-107, IBM Rio Scientific Center, Rio de Janeiro, 1990.

[11] F.R. Wagner, A.H. Viegas de Lima, L.G. Golendziner, and C. Iochpe. STAR: Um ambiente para a integração de ferramentas de projeto de sistemas digitais. In *VI Congresso da Sociedade Brasileira de Microeletrônica*, Belo Horizonte, 1991. SBMICRO.

[12] F.R. Wagner and A.H. Viegas de Lima. Design version management in the GARDEN framework. In *28th Design Automation Conference.* ACM/IEEE, 1991.

A Extending the Plasma language for conceptual scheme definition

Plasma [1] is a semi-formal specification language, specially conceived for describing the STAR data model in a clear way. It is not argued that it is a formal language, neither that its mechanisms are complete, orthogonal, or even consistent with each other. These same considerations are also valid for the extensions to the language introduced in this report, intended for the specification of particular conceptual schemes.

The presentation of the new language constructs will be done by exemplification.

Creating objects

The STAR data model defines various object types, such as Design, ViewState, and View.

ALU :- Design

creates a Design object with name ALU.

V-ALU-Struct :- View (MHD)

creates a View object with name V-ALU-Struct and indicates that this View is of type MHD.

Defining attributes

Object attributes (UserFields, in the STAR terminology) may be defined by the user. They have a user-defined name. They must be of some of the basic STAR data types (integer, real, string, etc) or of some user-defined data type derived from them. User-defined data types can be built using records, arrays, and sets, as well as by enumeration or subseting. An initial value may be assigned to an attribute in the conceptual scheme. This value may be modified at run-time.

ALU has Cell-Area :- UserField : integer

defines for object ALU a UserField with name Cell-Area and data type *integer*.

ALULogicView has ALUTechnology :- UserField : string = 2 micra

assigns to UserField ALUTechnology of object ALULogicView the value "2 micra".

```
ALU has ALUTechnology :- UserField : LayoutTechnology = { 2 micra / 1.5 micra }
```

defines for an object ALU a UserField with name ALUTechnology and user-defined data type LayoutTechnology, whose possible values are "2 micra" and "1.5 micra" (a data type created by subseting the type *string*).

UserFields may be structured and contain other UserFields, as in the example below.

```
ALU has ALUTechnology :- UserField
        has TechFile : file
        has TechVersion : integer
```

defines that an attribute ALUTechnology is composed of two sub-attributes, named TechFile (of basic data type *file*) and TechVersion (of basic data type *integer*).

Defining Ports and their attributes

The STAR data model specifies that Ports may be defined at different nodes of a control structure (Designs, ViewGroups, Views, and ViewStates). Ports created in a conceptual scheme must have a name. Furthermore, Ports may have user-defined attributes. As for the UserFields, these attributes have a name, a data type, and a possible initial value already defined in the conceptual scheme. They may be also composed of other sub-attributes.

```
OP :- Design
  has ABus :- Port
    has BitWidth = { 31 .. 0 }
    has Coordinates :- UserField : { integer; integer }
```

creates a Port with name ABus for the object OP. This Port has two attributes. BitWidth is a system-defined attribute (a pair of integers, indicating the identification of the MSB and LSB), while Coordinates is a user-defined attribute whose type is also a pair of integers.

Defining generalizations

The STAR data model defines that Designs are generalizations of ViewGroups and Views, and that ViewGroups are in turn generalizations of other ViewGroups and Views.

In a particular conceptual scheme, one has to define the names of the ViewGroups and Views which are generalized by a Design or a ViewGroup. It is also possible to define which UserFields of the generalization object are inherited by the generalized sub-objects.

```
ALU is generalization of {  ALULogicView (View MHD);
                            ALU-Layout-VG (ViewGroup) }
```

defines that ALU is a generalization of a View (of type MHD) with name ALULogic-View and of a ViewGroup with name ALU-Layout-VG.

```
ALU is generalization of ALULogicView
    {{ single inherited ALUTechnology }}
```

indicates that the UserField ALUTechnology of the generalization object ALU is inherited (in *strict* mode) by the generalized sub-objects (only ALULogicView, in this case).

Creating compositions and references

The STAR data model specifies that an object View may contain DesignInstances and/or Components, and that either Components or DesignInstances may reference other object representations. Furthermore, Components and DesignInstances have Ports and UserFields.

```
ALU contains Add1 :- Component
            has ... (Ports)
            has ... (UserFields)
```

defines that the object ALU contains a Component of name Add1 with given Ports and UserFields.

```
ALU contains AddA :- DesignInstance of Add1
```

defines that the object ALU contains a DesignInstance of name AddA which is an instance of Component Add1.

```
ALU contains Add1 :- Component
            has reference to Design : Register
```

defines that the object ALU has a Component Add1 which makes reference to the Design object of name Register.

References may also be done to particular ViewGroups, Views, or ViewStates of other objects. A particular dynamic configuration may be created with the construct

```
ALU contains Add1 :- Component
            has reference to most recent ViewState : RegisterView
```

which defines that the Component Add1 makes a reference to the most recent View-State of the View object of name RegisterView.

Relatórios de Pesquisa

RP-174: "Modelling the design methodology for the RISCO microprocessor", março, 1992.
F.R. WAGNER

RP-173: "BIBFRAC - Biblioteca para Geração de Fractais", fevereiro, 1992.
F.S. Montenegro; S.D. Olabarriaga

RP-172 "Introdução à Teoria dos Intervalos", fevereiro.
D.M. Claudio; T.A. Divério; H. Korzenowski; M. Leyser; L.C. Lamb

RP-171: "SMDB - Sistema de Manutenção de Banco de Dados - Manual de Operação do Sistema, janeiro, 1992.
G. GENSAS; B.R.T. FRANCIOSI; D.M. CLAUDIO

RP-170: "Design Methodology Management in the STAR Framework, dezembro, 1991.
F.R. WAGNER, A.H.V. DE LIMA

RP-169: "Lotos - Language of Temporal Ordering Specification, Novembro, 1991.
M.C. MÓRA; D.J. NUNES

RP-168: "Um Estudo Comparativo de Métodos Intervalares para Avaliação de Funções Racionais", setembro 1991.
A.R. COELHO; L.M. CADORE; G.P. DIMURO; D.M. CLAUDIO

RP-167: "The Data Model of the STAR Framework", novembro 1991.
F.R. WAGNER

RP-166: "Design Methodology Management in Design", novembro 1991.
F.R. WAGNER

RP-165: "Desenvolvimento de "Asa" e "Trama"", outubro, 1991.
S.D. OLABARRIAGA; E.M. CORRÊA; C. CALLIARI; A.L. POMPERMAYER

RP-164: "Estudo Topológico e Elétrico da Nova Célula de Base para CIs Gate Array - Tecnologia 1.2 um", outubro, 1991.
L.R. FROSI; G.V. PAIXÃO; J.L.G. CUNHA; D.A.C. BARONE

RP-163: "Representação de Conhecimento em Engenharia do Conhecimento", setembro, 1991.
N. EDELWEISS

RP-162: "Biblioteca de PADS Digitais CMOS 1.5 um - Versão 1", setembro 1991.
M.K. DOSSA

RP-161: "Versões Intervalares do Método de Newton", agosto 1991.
H. KORZENOWSKI; M. LEYSER; T.A. DIVERIO; D.M. CLAUDIO

RP-160: "Processamento Vetorial e Vetorização de Algoritmos na Máquina Convex C210", agosto 1991.
T.A. DIVERIO

RP-159: "Um estudo de técnicas de validação e de verificação de produtos de software", junho 1991.
N. EDELWEISS

RP-158: "Extensão das Ferramentas PIU/LINUS de especificação e controle de interfaces com o usuário", Junho 1991.
J.P. FIGUEIRÓ

RP-157: "The Domain of Nets and the Semantic Bases of a Notation for Nets", Abril 1991.
A.C.R. COSTA

RP-156: "Continuous Predicates and Logical Reflexivity", Abril 1991.
A.C.R. COSTA

RP-155: "TENTOS - Gerenciador de Software para Microeletrônica", abril 1991.
F.G. MORAES; R.A.L. REIS.

RP-154: "Sistemas Especialistas para a Engenharia de Software", Abril 1991.
H. AHLERT.

RP-153: "Estudo Comparativo e Taxonomia de Ferramentas de Suporte à Construção de Sistemas, Abril, 1991.
H. AHLERT.

RP-152: "IMP-MAC - Emulador de Impressora Padrão Apple", Abril, 1991.
C. DE ROSE; R.F. WEBER.

RP-151: "Em direção a um modelo para representação de aplicações de escritórios baseadas em documentos", Abril, 1991.
D.B.A. RUIZ.

RP-150: "Implementação de Sistemas de Gerência de Banco de Dados", Abril, 1991.
D.B.A. RUIZ.

RP-149: "Integração de Ferramentas no Sistema AMPLO: Crítica e Proposta de Extensões", março, 1991.
F.R. WAGNER.

RP-148: "Interface de Entrada para Teclado e Mouse", março, 1991.
J.M. DE SÁ.