UC Irvine
ICS Technical Reports

Title
System-on-chip modeling using objects and their relationships

Permalink
https://escholarship.org/uc/item/6qm2c97n

Authors
Doucet, Frederic
Sinha, Vivek
Siska, Chuck
et al.

Publication Date
1999-11-11

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Z
699
C3
no.99-53

ICS

TECHNICAL REPORT

System-on-Chip Modeling Using Objects and Their Relationships

Frederic Doucet, Vivek Sinha, Chuck Siska, Rajesh Gupta

Technical Report #99-53

Information and Computer Science
University of California
Irvine, CA 92697-3245
http://www.ics.uci.edu/~iesag

November 11, 1999

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Information and Computer Science
University of California, Irvine

SLBAR
Z
699
C3
no. 99-53

Abstract

System conceptualization and modeling requires support for both hardware and software components specification, and the ability to do complete system simulation rapidly and accurately. In this paper, we present a system engineering methodology that is built upon the definition and implementation of object relationship. Our approach allows a designer to build an executable, simulatable as well as syntesizable system model using the same language platform. We show how class interfaces can be used for structural representation throughout all design abstraction levels. The object oriented methodology allows IP-Reuse through object libraries and design patterns, and automatic documentation generation. The design process employs an extended UML (Unified Modeling Language) notation and classes from Scenic and ICSP class libraries. We describe our implementation of a prototype that supports our design methodology.

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Contents

1 Introduction 5

2 Our Approach to System Conceptualization 6
2.1 Relationships 6
2.2 Hardware Class Interfaces 7
2.2.1 Scenic 7
2.2.2 ICSP 8
2.3 Design Flow 10

3 Case Study: A DLX Compatible Pipeline 11
3.1 Step 1: Identify synchronous processes 11
3.2 Step 2: Identify polymorphic behavior 12
3.3 Step 3: Identify asynchronous processes 13
3.4 Step 4: Identify control logic 13
3.5 Step 5: Object Model 14
3.6 Simulation 14

4 Summary and Conclusion 16

A C++ Source Code for the DLX Pipeline Implementation 19
A.1 dlx_pipeline_c.hh 19
A.2 if_stage_c.hh 21
A.3 if_stage_c.cc 22
A.4 id_stage_c.hh 23
A.5 id_stage_c.cc 24
A.6 ex_stage_c.hh 25
A.7 ex_stage_c.cc 26
A.8 mem_stage_c.hh 27
A.9 mem_stage_c.cc 28
A.10 wb_stage_c.hh 29
A.11 wb_stage_c.cc 30
A.12 if_id_format_c.hh 31
A.13 if_id_format_c.cc 32
A.14 id_ex_format_c.hh 33
A.15 id_ex_format_c.cc 34
A.16 ex_mem_format_c.hh 35
A.17 ex_mem_format_c.cc 36
A.18 mem_wb_format_c.hh 37
A.19 mem_wb_format_c.cc 38

A.20 pc_setting_logic_c.hh 39
A.21 pc_setting_logic_c.cc 40
A.22 forwarding_unit_c.hh 41
A.23 forwarding_unit_c.cc 42
A.24 fixed_fields_c.hh 45
A.25 fixed_fields_c.cc 46
A.26 instruction_c.hh 47
A.27 instruction_c.cc 49
A.28 load_instruction_c.hh 51
A.29 load_instruction_c.cc 52
A.30 store_instruction_c.hh 53
A.31 store_instruction_c.cc 54
A.32 alu_instruction_c.hh 55
A.33 alu_instruction_c.cc 56
A.34 branch_instruction_c.hh 58
A.35 branch_instruction_c.cc 59
A.36 nop_instruction_c.hh 60
A.37 nop_instruction_c.cc 61
A.38 halt_instruction_c.hh 62
A.39 halt_instruction_c.cc 63
A.40 main.cc 64

List of Figures

1 UML examples for relationships between classes: (a) generalization (b) composition (c) aggregation (d) association 7
2 Scenic classes: (a) behavioral hierarchy (b) structural hierarchy (c) Scenic driver (d) module communications (e) communication hierarchy 8
3 Floorplan layout example (a) congested (b) reshaped (c) attributes reflected in ICSP (port & block placement) 9
4 ICSP classes diagram 10
5 Basic class diagram for the DLX pipeline 12
6 Polymorphic DLX instruction hierarchy 12
7 Asynchronous processes in DLX pipeline 13
8 Control logic example in DLX pipeline 13
9 Simplified object diagram for the DLX pipeline 14
10 Simple DLX pipeline test program 15
11 DLX pipeline simulation output for the small loop program 15

1 Introduction

The continuing increase of the integration density of microelectronic circuits is leading to incredible amounts of gates available for designs. However, the evolution of EDA tools is much slower. This design gap continues to grow, leading to a productivity crisis that have been identified by many researchers [1, 2].

There are several problems in design and verification of systems on chip (SOC) that can be traced to inadequate support for conceptualization and modeling of these systems. For instance design iterations are long, with little and *ad hoc* design reuse. Many SOCs using processor cores are not merely ASICs, but platforms for running embedded software, a large fraction of the value is in the software. But, we still have fragmented hardware and software design flows, despite many notable efforts made to close the gap [3, 5].

We believe that there is a need to push the design process and notation to a higher level of abstraction to encapsulate these problems, or take different views on them. Higher abstraction implies automation, adherence to "rules", and well defined interfaces. Doing this, we can structure the design flow, and address the SOC design problems in a systematic way. By increasing the hierarchical abstraction, we will be able to better manage complexity and reach an IP abstraction [4] that will add to the composability of SOC designs.

Mirroring their growth in the design of complex software systems, object oriented methods have been adapted to microelectronic IC and system design. See for instance [5, 6]. These techniques were applied in successful chip designs in [7, 8].

This work builds upon these earlier efforts by focusing on the design process, and explores changes needed to enhance the designers productivity. We will go back to the specification level (of the design methodology), and focus on how we should specify, conceptualize and model a system. We present a systematic approach to design modeling that is centered on object relationships in object oriented methods. Our work specifically builds on class libraries developed under the Scenic project [6]. We have augmented these interfaces to improve support for design conceptualization. Our approach to system modeling addresses two key aspects: (a) graphical design entry; (b) modeling of structure, relationships and behavior.

The graphical design entry is built using a UML [9] based front end. While it provides only rudimentary support for concurrency in hardware modeling, UML is useful as a design and documentation language. In addition to design visualization, our focus in using UML is on developing a visual formalism to enable the automatic generation of the underlying C++ code.

In the next section, we will present our approach to system conceptualization, and what design process should be implemented with it. In the third section, we will do a case study of the implementation of a DLX processor pipeline, and illustrate each step of the proposed flow with the example. The fourth section will discuss our methodology and its implications for SOC design and validation.

2 Our Approach to System Conceptualization

The goal of system conceptualization is to enable the designer to commit a design into a formal or semi-formal description that can be later used as documentation as well as input to verification tools for system correctness. This form of system conceptualization is an integral part of an important system design methodology.

We start with system level design exploration using the specify-explore-refine paradigm [10]. This methodology enables the writing of precise specifications and manual design space exploration and refinement. However, to close the gap between conceptual exploration and architectural exploration [11], we seek the capability to capture structural information at the highest levels of abstraction. We explain the notion of structure later in this paper.

We know that object oriented design is useful for modeling microelectronic systems by supporting structure as a key concept even at the conceptual level. The application of the object oriented development methodology for software engineering has been well studied [12] and successful in the industry. This methodology is used to describe the system in terms of class and object structures. This collection of information begins at an early stage in the process, at the behavioral level. It is very easy to go from one level of abstraction to the other, since components and behaviors are encapsulated behind precise interfaces.

It is possible to add more detailed structural information as the architectural and technological decisions are made. This process is repeated until designers reach a good mix between structure and behavior at the proper level of abstraction, which can produce satisfactory implementation results.

To enable this process, we need a notation to capture the specification. We chose UML which is a language for specifying, designing, visualizing and documenting complex systems. It provides a convenient notation that allows the developer to capture structural and behavioral details which can cover both software and hardware characteristics.

In this section, we will look at the interfaces available for the specification of the hardware components, and their relations to each other. We will then take a look at interfaces to express the physical layout information that we want to have throughout the design process. Once we know theses interfaces, we will elaborate about design flow changes to specify hardware components.

2.1 Relationships

One of the most important aspect of the classes modeling is the characterization of the relationships between the entities in a design. Let us look at the major kinds of relationships we usually find in a model, and talk about their meaning for a hardware implementation. Figure 1 presents the UML graphical notation for them.

1. *Generalization:* From a base class, we can derive specialized classes, through inheritance. These specialized classes inherit all interface elements from the superclass: the

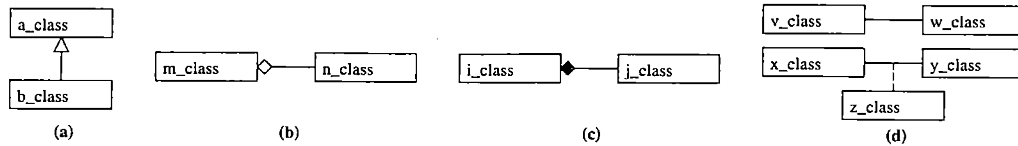

Figure 1: UML examples for relationships between classes: (a) generalization (b) composition (c) aggregation (d) association

data structures, and member functions. The subclass's interface is said to conform to the superclass's interface, i.e. an object from the subclass is able to substitute an object from the superclass. We also say that a superclass is a generalization of its subclasses. If we look at Figure 1(a), the superclass is *a_class*, and the subclass is *b_class*. For a hardware implementation, this means that a component could be replaced by another component which has the same port interfaces.

2. *Aggregation:* one object owns another one. The owner has logical control on the ownee. This is logical encapsulation, and expresses behavior hierarchy. For a hardware implementation, for instance, it may correspond to communication between two entities in a master-slave style. On Figure 1(b), *m_class* aggregates *n_class.*

3. *Composition:* one object owns and contains another one. A stronger form of aggregation. The owner has the control over the invocation and liveliness of the ownee. Hard encapsulation, expresses structural component hierarchy. For the hardware implementation, it may correspond to component hierarchy in a design. On figure 1(c), *j_class* is a component of *i_class.* This may represent, for instance a CPU datapath that physically contains an ALU block.

4. *Association:* represents conceptual relationships between classes. In the Figure 1(d), *v_class* has an association with *w_class* and *z_class* is an association class, which allows to add attributes, operations to the association. From an hardware perspective, it corresponds to communications, through either signals or channels.

2.2 Hardware Class Interfaces

Our work builds upon Scenic and ICSP class libraries for hardware modeling. In this section, we briefly describe important elements of these libraries relevant to this work.

2.2.1 Scenic

Scenic is a library of C++ classes that extends the C++ language to provide interfaces and mechanisms to express concurrency, reactivity and data types that are required to model hardware systems [6]. The designer can compile a description into a simulator, and it should be possible in the near future to synthesize C++/Scenic code [13]. Figure 2 illustrates the

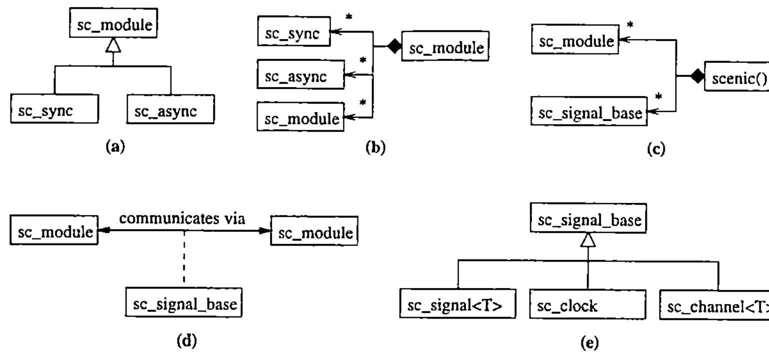

Figure 2: Scenic classes: (a) behavioral hierarchy (b) structural hierarchy (c) Scenic driver (d) module communications (e) communication hierarchy

main interfaces available, for a design using the Scenic class library. The functionality is very similar to the VHDL composition structure.

Processes: the interfaces for synchronous and asynchronous processes are *sc_sync* and *sc_async* respectively. A synchronous process is clocked through the interface, and is invoked on each desired edge. An asynchronous process is invoked according to events on the signal from which the process is sensitive. The behavior is similar as VHDL processes. The processes have a entry() methods which is invoked when an event triggers the process.

Signals: implemented with class *sc_signal.* Similar behavior as in VHDL modeling signals. A clock mechanism implements the time step calculation, and the event generation.

Channels: implemented in class *sc_channel.* Enables handshake communication between synchronous processes. Blocking buffered mechanism.

Modules: Modules can aggregate multiple modules, processes and signals. This forms the structure of a hardware entity.

Scenic Driver: This binds it all together. Encapsulate the library features, and invokes the simulation engine. The driver takes care of the concurrency and reactivity mechanisms internally.

2.2.2 ICSP

The Incidence Composition Structure Project (ICSP) library [15] is an extension to Scenic to express structural layout, routing and floorplan information. It allows the designer to start with a design partition and allocation of cores to functions, and iteratively determine what relative shapes and placement works best for all his modules and ports. The goal is to reduce wiring congestion, minimize routing overhead and keep blocks with tight requirements near

one another. This process is possible on the multiple levels of floorplanning activity: overall chip layout, optimization of individual cores and critical path optimization.

The Scenic component interfaces have been specialized to contain the following information:

1. *Physical Block Size & Placement:* as illustrated in Figure 3(c), by the size of the rectangle, and the placement of the modules, in relation to each other.

2. *Ports Locations:* indicates on which side of the rectangle a port is situated. The ports are contained in totally ordered sets, one set for each side of the rectangle.

The primary use of the ICSP library is in definition of structure to aid the design and synthesis of structural blocks, e.g. datapath blocks through regularity extraction [14].

Figure 4 shows the class diagram for the ICSP library. The placement of the components is specified by sets of incidence rules, which are managed internally by the library. Component layout at this level requires planarity testing to detect and warn about impossible layouts, provided by the library. Also, the user has the potential to see non-intuitive results. This can happen on adding or removing an edge constraint or changing a constraint's priority (if such a thing is included in the description semantics). For example, too many "next to" constraints might be impossible to support. Introducing priorities in the ordering may also produce a cascade of placement adjustments. Figure 3(a) shows a floorplan that has congestion highlighted in the circle. Figure 3(b) shows the same floorplan, with modification, to place the blocks in a structured way. Figure 3(c) shows the representation of the information using ICSP.

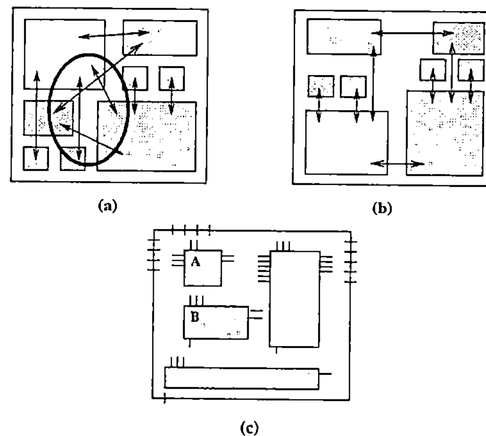

Figure 3: Floorplan layout example (a) congested (b) reshaped (c) attributes reflected in ICSP (port & block placement)

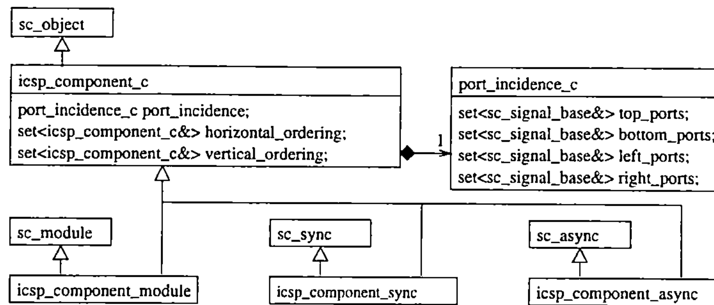

Figure 4: ICSP classes diagram

2.3 Design Flow

Now that we know the available design interfaces, let us look at the proposed design flow. We will focus on the hardware side, since the software design flow is the same as in a software engineering process.

The object oriented design is a relatively small part in the entire process. The designer has to have in mind the main blocks he or she wants to see in the design, and be able to do a structural decomposition, followed by a behavioral decomposition. The result of the second decomposition will partly be mapped on the result of the first one, and partly encapsulated into a generalized interface which will be specialized as needed. This is a elegant way to design a datapath, from a software engineer point of view. Note that these main blocks will be identified in an iterative fashion. But still, they form the base of the structure. The reader may notice an analogy with the task-component binding, which is here a behavior-structure binding. The steps to follow are:

1. *Identify synchronous processes.* For a synchronous design, these will form the major blocks of the design. By connecting theses blocks together, the identification of the data format for the interfaces will be needed. This is the first step to establish the structure of the system. These components will have *sc_sync* interfaces and will be on the floorplan. The behavior may be general, and dependent on what data is fed to the blocks

2. *Identify polymorphic behavior propagation.* Object-oriented design provides powerful behavior and structure decomposition methods. Inheritance can be used to minimize duplication of descriptions by promoting transparent shared behavior through the superclass interface. This step is an important part of the process; the identification of a generalized interface for the operations of the datapath of the system. The behavior is invoked and propagated throughout the synchronous stages by propagating a reference to the superclass from process to process. The superclass has several virtual methods that are designed to be called by each stage it will reach. We call this mechanism datapath design and functionality propagation through polymorphism.

3. *Identify asynchronous processes.* For a synchronous design, these processes consist of the control signals between the synchronous stages. These components have the *sc_async* interfaces, and will be on the floorplan. Note that a full asynchronous design is also possible, with this polymorphic behavior propagation.

4. *Identify control logic.* These components are glue logic between the processes. They are signal format, and control words. They do not have a Scenic interface, since they will be flattened in some process.

5. *Build object diagram.* Once the designer is satisfied with his class models, he can build the object diagram to express the object instances, their location on the floorplan, and the ports and wire information. He will change the Scenic interfaces to ICSP interfaces to do so.

This process is iterative, and the order of the steps is important in the beginning of the process. But once the initial models are built, the designer can easily do further iterations on a design, as it is refined for functionality. In this following section, we illustrate this methodology using an example.

We are in the process of developing a front end [16] which allows the user to go through the elaborated steps and to conceptualize and model the design using a tool with a graphical interface and the UML notation. The front end allows automatic generation of C++ code from classes and object diagrams.

3 Case Study: A DLX Compatible Pipeline

In this section, we present a case study for our design methodology with the conceptualization of a DLX compatible processor pipeline [17]. The model needs to be cycle accurate in the execution of the instructions, and byte code compatible with the DLX encoding. We will describe the process of the implementation step by step. Once the requirements are clear, we can proceed with the building of the class diagram.

3.1 Step 1: Identify synchronous processes

We have first done a class model of the stages of the pipeline. The stages of the pipeline, are shown on Figure 5. The stages are specialized from the *sc_sync* interface. All stages are synchronous processes, which get activated on each clock high tick. We also have specified classes for the format of the data for the communication between classes. Let us not commit now to the choice of the communication mechanism, but rather focus on behavior encapsulation.

Note that the *ex_stage_c* class is composed of one *alu_c* class. This ALU has no Scenic interface, which means that its methods are directly called by its owner, the EX stage. This

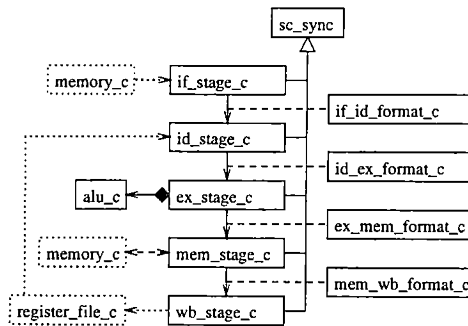

Figure 5: Basic class diagram for the DLX pipeline

means that the ALU is combinational logic; its input and output are registered by the EX stage.

Next step is to add the external structural component to the design. In this case, we have data and program memory and a register file. The rectangle for theses classes is dotted in the figure, because they are not in the DLX pipeline classes. The reader can also note that we added arrows to the association lines, which correspond to the direction of the data in the communication. These are slight changes from the standard UML conventions. Usually, an arrow indicates navigability or responsibility. In some extent, the arrow has a data or control responsibility.

3.2 Step 2: Identify polymorphic behavior

We have been through the major structural blocks of the pipeline, let us now look at the needed behavior structuring to be able to execute an instruction. In this design, we will propagate a reference to the generalized instruction class; but it will refer a specialized instruction class.

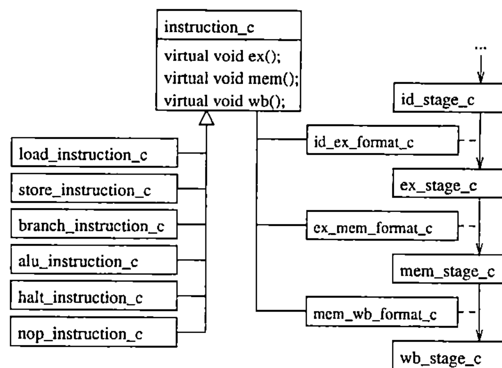

Figure 6: Polymorphic DLX instruction hierarchy

Looking at Figure 6 we see the *instruction_c* class interface, with the virtual methods

ex(), *mem()* and *wb()*. These are invoked by the EX, MEM and WB stages respectively. The specialized instruction classes (such as an *alu_instruction_c*) re-implement the methods, to have the right behavior.

In this approach, we used inheritance to handle the division of the EX, MEM, and WB stages to invoke the polymorphic behavior of an instruction. At these stages, a virtual method is called at the instruction level interface, and the call is "forwarded" to the specialized class. The overloaded virtual methods of the implemented instruction behaves appropriately given its type, for example, an *alu_instruction_c* will perform a call to the ALU at the EX stage, while a *load_instruction_c* would not call to the ALU.

3.3 Step 3: Identify asynchronous processes

Now, the pipeline is capable of propagating the instruction behavior through its stages. However, it needs some control processes to orchestrate data movement through the datapath.

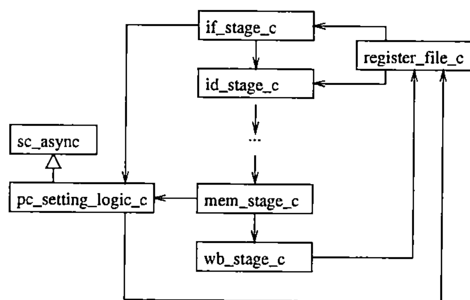

Figure 7: Asynchronous processes in DLX pipeline

Figure 7 shows the addition of an asynchronous process to update the program counter. It is generalized from the *sc_async* Scenic class, and triggered by changes on its input. This module should have the same physical properties as a synchronous process.

3.4 Step 4: Identify control logic

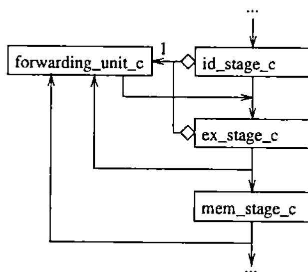

Figure 8: Control logic example in DLX pipeline

Figure 8 presents the mechanism for data forwarding in case of data dependencies between the instructions. Looking at the diagram, we see that the *forwarding_unit_c* class is aggregated in both the ID and EX stages, and that to perform its functionality, it receives the registered values from the inter stages registers. In this case, there should not be two entities of the forwarder because it is aggregated twice. The needed methods of this entity should be flattened in the stages. It is not the same method that is called by the ID and the EX stages. So, this should become asynchronous logic inside the entities. Then, it does not need an ICSP interface to have floorplanning information.

3.5 Step 5: Object Model

Once we reach a good compromise between structure and behavior, we can proceed to the object diagram, to express physical floorplanning information. At this time, non- ICSP classes with structural view should be converted to ICSP classes to be able to use the object view.

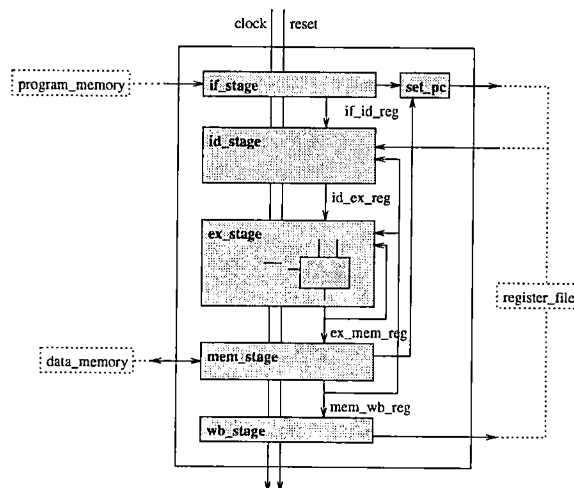

Figure 9: Simplified object diagram for the DLX pipeline

Figure 9 presents the object specification format. On it, we specify the port placement, and it shall be reflected automatically in the partially ordered sets for the ICSP port specification by our front end. The designer, with the help of the tool, should be able to estimate the area and study the feasibility of a design. It should also be possible to force these values, and add constraint and priorities.

3.6 Simulation

To validate the design, we built our own console- mode debugging tracing, and also used Scenic's ability to watch signals and emit wave data. Figure 10 presents a test program,

```
load        5, r1
andi     0, r2,r2
add      r1,r2,r2
subi     r1, 1,r1
bnez        r1, 2
nop
nop
nop
halt
nop
nop
nop
```

Figure 10: Simple DLX pipeline test program

consisting of a simple loop with a counter. Since the pipeline flush is not yet implemented, we have to put no operation instruction after the branch instruction.

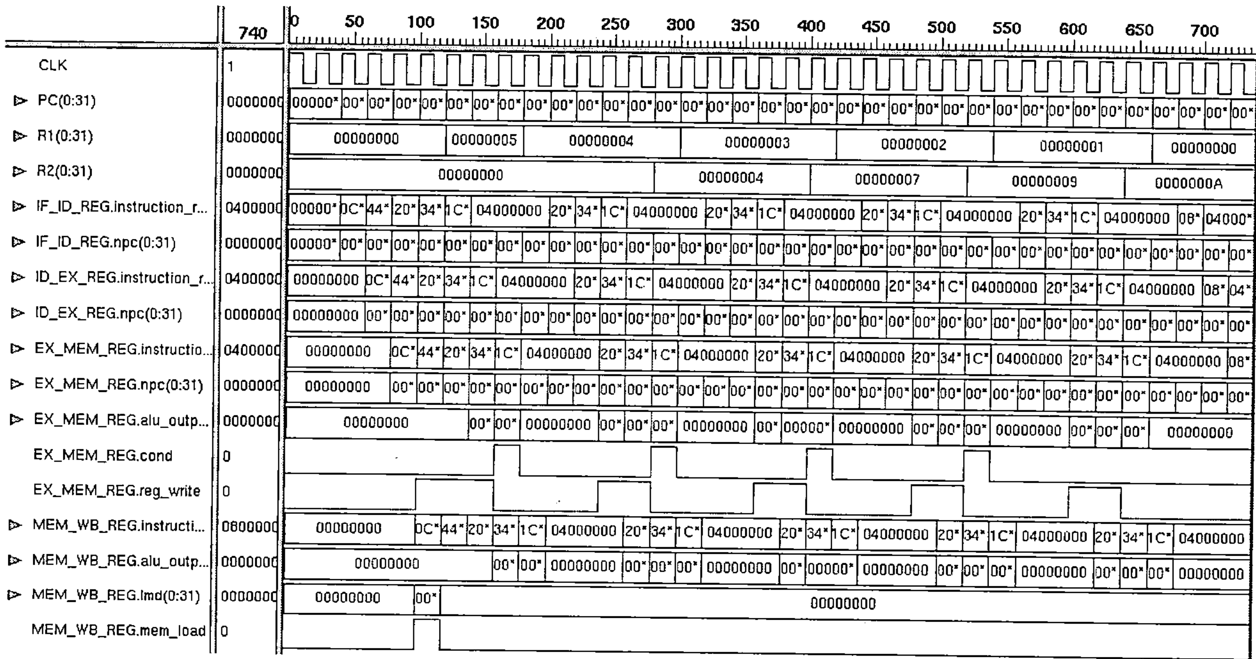

Figure 11: DLX pipeline simulation output for the small loop program

The DLX pipeline code consists of approximately 2000 lines of C++ code much of which can be automatically generated from the UML front end [16]. The actual body of the user code, which are the entry functions, is around 200 lines of code.

Simulation results from the DLX code are produced simply by compiling and executing the model. The simulation results are shown in Figure 11. Notice the decreasing register R1 as loop execution proceeds, and the instruction register being propagated down the pipeline.

This shows that complete cycle-accurate models can be constructed using our class library. They are also synthesizable using existing synthesis tools.

4 Summary and Conclusion

Design conceptualization and accurate behavioral and structural modeling are key to successful microelectronic system-on-chip design and validation. Object-oriented languages provide a good starting point to think about composability of such systems as collection of robust and reusable components. However, object oriented language extension (either as a library support or syntactical additions) is not enough, design methodology and corresponding tool support are critical to making a successful transition from software engineering to microelectronic system design. In this paper, we have presented a systematic approach to design modeling based on class interfaces and object relationships. Structural decomposition, that is specification of structural information at the conceptualization level, is critical to SOC designs. It also leads to an innovative way of behavior decomposition relying on superclass specialization over the structure.

Object-oriented task structuring is not immediately obvious to most microelectronic designers except those intimately familiar with the object oriented software engineering concepts. Our support for structure and subsystem decomposition using object oriented structuring criteria helps the designer effectively utilize the versatility of object-oriented software methods in constructing reusable system blocks through the shared composition of specialized objects. We present the notion of an object model based on similar notion from UML specialized for SOC design. Conceptual exploration is performed by refining the object model. Architectural exploration takes place as the abstraction levels is lowered, and the granularity of the specification gets smaller.

In summary, we present a compromise between functional decomposition and system structuring, which comprises of object oriented design methodologies, and C++ synthesis technology. This is possible because of growing need and support for C++ based hardware synthesis technology. The usage of the object oriented methodologies and the UML front end will enable the designer to build libraries, automatically document them, and use them as IP Core. Our future work includes design of test interfaces, and analysis of SOC timing behavior at the conceptual level while incorporating potential technology details.

Acknowledgments

The authors would like to acknowledge the contributions of Olivier Hebert for insightful discussions and for kindly sending the slides from [2], and of Sumit Gupta for his help. The authors would also like to acknowledge support from National Science Foundation award number NSF CCR-9806898, from DARPA/ITO DABT63-98-C-004, and from Synopsys Corporation (under UC Micro program) and for providing the Scenic class library.

References

[1] WEIL, P. EDA Roadmap Task Force Report: Design of Microprocessors. Electronic Design Automation Industry Council, 1999.

[2] AITKEN, R. C. Design of Testable Systems-on-Chip, a one day tutorial. GRIAO, Montreal, Qc, October 1999.

[3] WOLF, W. Object-oriented co-synthesis of distributed embedded systems. in *ACM Transactions on Design Automation of Electronic Systems*, July 1996.

[4] KEATING, M., BRICAUD, P. Reuse Methodology Manual for System-on-a-Chip Designs. Kluwer Academic Publishers, 1998.

[5] KUMAR, S., AYLOR, J. H., JOHNSON, B. W., WULF, W. A. Object-Oriented Techniques in Hardware Design. in *Computer*, June 1994.

[6] LIAO, S. Y., TJIANG, S., AND GUPTA, R. K. An Efficient Implementation of Reactivity in Modeling Hardware in the Scenic Synthesis and Simulation Environment. In *Proceedings of the Design Automation Conference*, 1997.

[7] VERKEST, D., COCKX, J., POTARGENT, F., JONG, G., DE MAN, H. On the use of C++ for system-on-chip design. In *IEEE Computer Society Workshop on VLSI*, April 1999.

[8] VERNALDE, S., SCHAUMONT, P., BOLSENS, I. An Object Oriented Programming Approach for Hardware Design. In *IEEE Computer Society Workshop on VLSI*, April 1999.

[9] FOWLER, M., KENDALL, S. UML Distilled: Applying the Standard Object Modeling Language. Addison-Wesley, 1997.

[10] GAJSKI, D. D., VAHID, F., NARAYAN, S., GONG, J. Specification and Design of Embedded Systems. PTR Prentice Hall, 1994.

[11] GUPTA, R. K. Timing-Driven System Design. In *IEEE Computer Society Workshop on VLSI*, April 1999.

[12] PRESSMAN, R. S. Software Engineering: a Practitioner's Approach, fourth edition. McGraw-Hill, 1997.

[13] SYSTEMC HOME PAGE http://www.systemc.org. 1999.

[14] CHOWDHARY, A., KALE, S., SARIPELLA, P. K., SEHGAL, N. K., GUPTA, R. K. Extraction of Functional Regularity in Datapath Circuits. in *IEEE Trans. on CAD*. Vol 19, no 9, Sept 1999.

[15] SISKA, C. Incidence Structure-Composition Project - Technical Report. CECS ICS, UC Irvine June 1999.

[16] SIHNA, V. Relationship Aware UML Front End - Technical Report. CECS ICS, UC Irvine October 1999.

[17] HENNESSY, J. L., PATTERSON, D. A. Computer Organization and Design: The hardware/software interface. Morgan-Kaufman 1994.

A C++ Source Code for the DLX Pipeline Implementation

This appendix will list the main C++ files for the simple DLX pipeline implementation. The complete and most up to date code is available upon request, please send email to doucet@ics.uci.edu.

A.1 dlx_pipeline_c.hh

```
#include <scenic.h>

#include "if_stage_c.hh"
#include "id_stage_c.hh"
#include "ex_stage_c.hh"
#include "mem_stage_c.hh"
#include "wb_stage_c.hh"
#include "pc_setting_logic_c.hh"
#include "hazard_detection_unit_c.hh"

#ifndef DLX_PIPELINE_C
#define DLX_PIPELINE_C

class dlx_pipeline_c : public sc_module {
private:
public: // Should be able to put variable in private
       // to express data encapsulation.
  register_file_c&             register_file;
  memory_unit_base_c<unsigned>& program_memory;
  memory_unit_base_c<unsigned>& data_memory;

  // the stages entities
  if_stage_c  if_stage;
  id_stage_c  id_stage;
  ex_stage_c  ex_stage;
  mem_stage_c mem_stage;
  wb_stage_c  wb_stage;
  // the registers between the stages
  sc_signal<if_id_format_c>  if_id_reg;
  sc_signal<id_ex_format_c>  id_ex_reg;
  sc_signal<ex_mem_format_c> ex_mem_reg;
  sc_signal<mem_wb_format_c> mem_wb_reg;

  // control signals
  sc_signal<bool>           branch;
  sc_signal<unsigned>       branch_target_pc;
  sc_signal<unsigned>       npc;
  sc_signal<bool>           stall;

  // control process
  pc_setting_logic_c        pc_controler;
  hazard_detection_unit_c   hazard_detection_unit;

  dlx_pipeline_c(const char*                    NAME,
sc_clock_edge&                 TICK,
register_file_c&               REGISTER_FILE,
memory_unit_base_c<unsigned>& PROGRAM_MEMORY,
memory_unit_base_c<unsigned>& DATA_MEMORY,
sc_signal<bool>&               RESET
): sc_module(NAME),
   register_file(REGISTER_FILE),
   program_memory(PROGRAM_MEMORY),
   data_memory(DATA_MEMORY),
```

```
    if_stage("IF_STAGE",   TICK,
     RESET,
     if_id_reg,
     npc,
     REGISTER_FILE.pc,
     PROGRAM_MEMORY),
    id_stage("ID_STAGE",   TICK,
     if_id_reg,
     mem_wb_reg, // forwarded value
     id_ex_reg,
     REGISTER_FILE),
    ex_stage("EX_STAGE",   TICK,
     id_ex_reg,
     ex_mem_reg, // forwarded value
     mem_wb_reg, // forwarded value
     ex_mem_reg),
    mem_stage("MEM_STAGE", TICK,
      ex_mem_reg,
      mem_wb_reg,
      branch,
      branch_target_pc,
      DATA_MEMORY),
    wb_stage("WB_STAGE",   TICK,
     mem_wb_reg,
     REGISTER_FILE),

    pc_controler("PC_CONTROLER",
branch_target_pc,
npc,
branch, REGISTER_FILE.pc),
    hazard_detection_unit("HAZARD_DETECTION_UNIT",
 if_id_reg,
 id_ex_reg,
 stall) {
    end_module();
  }

};

#endif /* DLX_PIPELINE_C */
```

A.2 if_stage_c.hh

```
#include <scenic.h>
#include "if_id_format_c.hh"
#include "memory_unit_base_c.hh"
#include "register_file_c.hh"

#ifndef IF_STAGE_C
#define IF_STAGE_C

#pragma interface

class if_stage_c : public sc_sync {
private:
public:
  const sc_signal<bool>&        reset;
  sc_signal<unsigned>&          npc;
  const sc_signal<unsigned>&    pc;
  memory_unit_base_c<unsigned>& program_memory;
  sc_signal<if_id_format_c>&    if_id_reg;

  if_stage_c(const char*                      NAME,
     sc_clock_edge&                 TICK,
     const sc_signal<bool>&         RESET,
     sc_signal<if_id_format_c>&     IF_ID_REG,
     sc_signal<unsigned>&           NPC,
     const sc_signal<unsigned>&     PC,
     memory_unit_base_c<unsigned>&  PROGRAM_MEMORY);

  void entry(void);
};


#endif /* IF_STAGE_C */
```

A.3 if_stage_c.cc

```
#pragma implementation "if_stage_c.hh"

#include "if_stage_c.hh"
#include <iostream.h>

/////////////////////////////////////////////////////////
if_stage_c::if_stage_c(const char*                      NAME,
       sc_clock_edge&                TICK,
       const sc_signal<bool>&        RESET,
       sc_signal<if_id_format_c>&   IF_ID_REG,
       sc_signal<unsigned>&          NPC,
       const sc_signal<unsigned>&    PC,
       memory_unit_base_c<unsigned>& PROGRAM_MEMORY
         ) : sc_sync(NAME, TICK),
 reset(RESET),
 npc(NPC),
 pc(PC),
 program_memory(PROGRAM_MEMORY),
 if_id_reg(IF_ID_REG){
 // program_memory(PROGRAM_MEMORY) {
}

/////////////////////////////////////////////////////////
void if_stage_c::entry(void) {

  npc.write(0);
  wait(); // Wait for the first up tick.

  unsigned ir, pc_value;
  while (true) {
    cout << "| (1) IF: IR= " << hex << ir << " [";
    cout << sc_clock::time_stamp() << "] " << endl;

    pc_value= pc.read();
    ir=       program_memory.read(pc_value);

    if_id_format_c if_id_value(ir, ++pc_value);
    npc.write(pc_value);
    if_id_reg.write(if_id_value);

    wait();
  }
}
```

A.4 id_stage_c.hh

```
#include <scenic.h>
#include "if_id_format_c.hh"
#include "id_ex_format_c.hh"
#include "fixed_fields_c.hh"
#include "mem_wb_format_c.hh"
#include "forwarding_unit_c.hh"

#ifndef ID_STAGE_C
#define ID_STAGE_C

#pragma interface

class id_stage_c : public sc_sync {
private:

public:
  const sc_signal<if_id_format_c>&  if_id_reg;
  const sc_signal<mem_wb_format_c>& last_mem_wb_reg;
  register_file_c&                  register_file;
  forwarding_unit_c                 forwarding_unit;
  sc_signal<id_ex_format_c>&        id_ex_reg;

  id_stage_c(const char*                          NAME,
     sc_clock_edge&                    TICK,
     const sc_signal<if_id_format_c>&  IF_ID_REG,
     const sc_signal<mem_wb_format_c>& LAST_MEM_WB_REG,
     sc_signal<id_ex_format_c>&        ID_EX_REG,
     register_file_c&                  REGISTER_FILE
     );

  void entry(void);

};

#endif /* ID_STAGE_C */
```

A.5 id_stage_c.cc

```
#pragma implementation "id_stage_c.hh"

#include "id_stage_c.hh"
#include "instruction_c.hh"
#include "register_file_c.hh"
#include <assert.h>
#include <iostream.h>

//////////////////////////////////////////////////////////
id_stage_c::id_stage_c(const char*                        NAME,
       sc_clock_edge&                       TICK,
       const sc_signal<if_id_format_c>&  IF_ID_REG,
       const sc_signal<mem_wb_format_c>& LAST_MEM_WB_REG,
       sc_signal<id_ex_format_c>&         ID_EX_REG,
       register_file_c&                     REGISTER_FILE
       ) : sc_sync(NAME, TICK),
 if_id_reg(IF_ID_REG),
 last_mem_wb_reg(LAST_MEM_WB_REG),
 register_file(REGISTER_FILE),
                        id_ex_reg(ID_EX_REG) {
  register_file= REGISTER_FILE;
}

//////////////////////////////////////////////////////////
void id_stage_c::entry(void) {
  wait();
  wait();

  while (true) {
    if_id_format_c if_id_value= if_id_reg.read();

    fixed_fields_c ff=
      instruction_c::decode_fixed_fields(if_id_value.instruction_register);

    // fetch fixed fields register values //////////////////////////////////
    assert( ff.src_reg1< NUM_DATA_REGISTERS );
    unsigned op_a= register_file.d[ ff.src_reg1 ];
    assert( ff.src_reg2< NUM_DATA_REGISTERS );
    unsigned op_b= register_file.d[ ff.src_reg2 ];
    ////////////////////////////////////////////////////////////////////////

    instruction_c* new_instruction=
      instruction_c::get_new_instruction(ff, op_a, op_b);
    assert(new_instruction!= NULL);

    mem_wb_format_c mem_wb_value= last_mem_wb_reg.read();
    forwarding_unit.id_forward(new_instruction, mem_wb_value);

    id_ex_format_c id_ex_value(if_id_value.instruction_register,
       new_instruction,
       if_id_value.npc);
    id_ex_reg.write(id_ex_value);

    cout << "| (2) ID: "; new_instruction->print_tag();
    cout << "[" << sc_clock::time_stamp() << "]"<<endl;
    wait();
  }
}
```

A.6 ex_stage_c.hh

```
#include <scenic.h>
#include "alu_c.hh"
#include "id_ex_format_c.hh"
#include "ex_mem_format_c.hh"
#include "forwarding_unit_c.hh"

#ifndef EX_STAGE_C
#define EX_STAGE_C

#pragma interface

class ex_stage_c : public sc_sync {
private:

public:
  const sc_signal<id_ex_format_c>&  id_ex_reg;
  const sc_signal<ex_mem_format_c>& last_ex_mem_reg; //loop for data forwarding
  const sc_signal<mem_wb_format_c>& last_mem_wb_reg; //loop for data forwarding
  forwarding_unit_c                 forwarding_unit;
  alu_c<unsigned>                   alu;
  sc_signal<ex_mem_format_c>&       ex_mem_reg;

  ex_stage_c(const char*                          NAME,
     sc_clock_edge&                    TICK,
     const sc_signal<id_ex_format_c>&  ID_EX_REG,
     const sc_signal<ex_mem_format_c>& LAST_EX_MEM_REG,
     const sc_signal<mem_wb_format_c>& LAST_MEM_WB_REG,
     sc_signal<ex_mem_format_c>&       EX_MEM_REG);

  void entry(void);
};

#endif /* EX_STAGE_C */
```

A.7 ex_stage_c.cc

```
#pragma implementation "ex_stage_c.hh"

#include "ex_stage_c.hh"
#include <iostream.h>

/////////////////////////////////////////////////////////
ex_stage_c::ex_stage_c(const char*                          NAME,
       sc_clock_edge&                    TICK,
       const sc_signal<id_ex_format_c>&  ID_EX_REG,
       const sc_signal<ex_mem_format_c>& LAST_EX_MEM_REG,
       const sc_signal<mem_wb_format_c>& LAST_MEM_WB_REG,
       sc_signal<ex_mem_format_c>&       EX_MEM_REG
       ): sc_sync(NAME, TICK),
 id_ex_reg(ID_EX_REG),
 last_ex_mem_reg(LAST_EX_MEM_REG),
 last_mem_wb_reg(LAST_MEM_WB_REG),
 ex_mem_reg(EX_MEM_REG) {
}

/////////////////////////////////////////////////////////
void ex_stage_c::entry(void) {

  // initialize the feedback loop...
  ex_mem_reg.write(ex_mem_format_c());
  wait();
  wait();
  wait();

  while (true) {
    cout << "| (3) EX " << " [";
    cout << sc_clock::time_stamp() << "] " << endl;
    id_ex_format_c id_ex_value= id_ex_reg.read();
    // data forwarding //////////////////////////////////////////////
    ex_mem_format_c last_ex_mem_value= last_ex_mem_reg.read();
    mem_wb_format_c last_mem_wb_value= last_mem_wb_reg.read();

    forwarding_unit.ex_forward(id_ex_value,
       last_ex_mem_value,
       last_mem_wb_value);
    ////////////////////////////////////////////////////////////////

    ex_mem_format_c ex_mem_value(id_ex_value.instruction_register,
 id_ex_value.the_instruction,
 id_ex_value.npc);

    // execute the instruction by polymorphism
    ex_mem_value.alu_output=
      ex_mem_value.the_instruction->execute_alu_output(alu);
    ex_mem_value.cond=
      ex_mem_value.the_instruction->execute_cond_output(alu);
    ex_mem_value.reg_write=
      ex_mem_value.the_instruction->reg_write();

    ex_mem_reg.write(ex_mem_value);
    wait();
  }
}
```

A.8 mem_stage_c.hh

```
#include <scenic.h>
#include "alu_c.hh"
#include "ex_mem_format_c.hh"
#include "mem_wb_format_c.hh"

#ifndef MEM_STAGE_C
#define MEM_STAGE_C

#pragma interface

class mem_stage_c : public sc_sync {
private:

public:
  const sc_signal<ex_mem_format_c>& ex_mem_reg;
  memory_unit_base_c<unsigned>&     data_memory;
  sc_signal<bool>&                  branch;
  sc_signal<unsigned>&              branch_target_pc;
  sc_signal<mem_wb_format_c>&       mem_wb_reg;

  mem_stage_c(const char*                          NAME,
      sc_clock_edge&                      TICK,
      const sc_signal<ex_mem_format_c>& EX_MEM_REG,
      sc_signal<mem_wb_format_c>&       MEM_WB_REG,
      sc_signal<bool>&                  BRANCH,
      sc_signal<unsigned>&              BRANCH_TARGET_PC,
      memory_unit_base_c<unsigned>&     DATA_MEMORY
      );

  void entry(void);

};

#endif /* MEM_STAGE_C */
```

A.9 mem_stage_c.cc

```
#pragma implementation "mem_stage_c.hh"

#include "mem_stage_c.hh"
#include <iostream.h>

mem_stage_c::mem_stage_c(const char*                          NAME,
 sc_clock_edge&                     TICK,
 const sc_signal<ex_mem_format_c>& EX_MEM_REG,
 sc_signal<mem_wb_format_c>&        MEM_WB_REG,
 sc_signal<bool>&                   BRANCH,
 sc_signal<unsigned>&               BRANCH_TARGET_PC,
 memory_unit_base_c<unsigned>&      DATA_MEMORY
 ): sc_sync(NAME, TICK),
   ex_mem_reg(EX_MEM_REG),
   data_memory(DATA_MEMORY),
   branch(BRANCH),
   branch_target_pc(BRANCH_TARGET_PC),
   mem_wb_reg(MEM_WB_REG) {
}

/////////////////////////////////////////////////////////
void mem_stage_c::entry(void) {
  branch.write(false);
  mem_wb_reg.write(mem_wb_format_c());

  wait();
  wait();
  wait();
  wait();

  while (true) {
    ex_mem_format_c ex_mem_value= ex_mem_reg.read();
    mem_wb_format_c mem_wb_value(ex_mem_value.instruction_register,
 ex_mem_value.the_instruction,
 ex_mem_value.alu_output);
    cout << "| (4) MEM: access " << " [";
    cout << sc_clock::time_stamp() << "] " << endl;

    if (ex_mem_value.cond== true) {
      cout << "branching to "<< ex_mem_value.alu_output << endl;
      branch.write(true);
      branch_target_pc.write(ex_mem_value.alu_output);
    }
    else {
      branch.write(false);
    }
    mem_wb_value.lmd=
      mem_wb_value.the_instruction->mem_access(mem_wb_value.alu_output,
       data_memory);
    mem_wb_value.mem_load= mem_wb_value.the_instruction->mem_load();
    mem_wb_reg.write(mem_wb_value);
    wait();
  }

}
```

A.10 wb_stage_c.hh

```
#include <scenic.h>
#include "register_file_c.hh"
#include "mem_wb_format_c.hh"

#ifndef WB_STAGE_C
#define WB_STAGE_C


class wb_stage_c : public sc_sync {
private:
public:
  const sc_signal<mem_wb_format_c>& mem_wb_reg;
  register_file_c&                  register_file;

  wb_stage_c(const char*                       NAME,
     sc_clock_edge&                     TICK,
     const sc_signal<mem_wb_format_c>& MEM_WB_REG,
     register_file_c&                   REGISTER_FILE);

  void entry(void);
};


#endif /* WB_STAGE_C */
```

A.11 wb_stage_c.cc

```
#pragma implementation "wb_stage_c.hh"

#include "wb_stage_c.hh"
#include <iostream.h>

/////////////////////////////////////////////////////////
wb_stage_c::wb_stage_c(const char*                              NAME,
       sc_clock_edge&                       TICK,
       const sc_signal<mem_wb_format_c>& MEM_WB_REG,
       register_file_c&                     REGISTER_FILE
       ): sc_sync(NAME, TICK),
 mem_wb_reg(MEM_WB_REG),
 register_file(REGISTER_FILE) {
}

/////////////////////////////////////////////////////////
void wb_stage_c::entry(void) {
  wait();
  wait();
  wait();
  wait();
  wait();

  while (true) {
    // this process will block on the read
    mem_wb_format_c mem_wb_value= mem_wb_reg.read();
    cout << "| (5) WB: access " << " [";
    cout << sc_clock::time_stamp() << "] " << endl;

    mem_wb_value.the_instruction->write_back(mem_wb_value.instruction_register,
     mem_wb_value.alu_output,
     mem_wb_value.lmd,
     register_file);
    wait();
  }
}
```

A.12 if_id_format_c.hh

```
#include <scenic.h>

#ifndef IF_ID_FORMAT_C
#define IF_ID_FORMAT_C

class if_id_format_c {
public:
  unsigned instruction_register;
  unsigned npc;
  if_id_format_c(void) {};
  if_id_format_c(unsigned u, unsigned n) { instruction_register= u; npc=n; }
  bool operator==(const if_id_format_c&) const ;

};

void sc_trace(sc_trace_file *tf,
      const if_id_format_c& m,
      const sc_string& NAME);

#endif /* IF_ID_FORMAT_C */
```

A.13 if_id_format_c.cc

```
#include "if_id_format_c.hh"


void sc_trace(sc_trace_file *tf,
      const if_id_format_c& m,
      const sc_string& NAME) {
  sc_trace(tf, m.instruction_register, NAME+ ".instruction_register");
  sc_trace(tf, m.npc,                  NAME+ ".npc");
};


bool if_id_format_c::operator==(const if_id_format_c& i) const {

  if ( (instruction_register != i.instruction_register) || (npc != i.npc) )
    return false;
  else
    return true;

};
```

A.14 id_ex_format_c.hh

```
#include <scenic.h>
#include "instruction_c.hh"

#ifndef ID_EX_FORMAT_C
#define ID_EX_FORMAT_C

class id_ex_format_c {
public:
  unsigned       instruction_register;
  instruction_c* the_instruction;
  unsigned       npc;


  id_ex_format_c(void) {
    instruction_register=0; the_instruction= NULL; npc=0;
  };
  id_ex_format_c(unsigned ir,
 instruction_c* i,
 unsigned n) {
    instruction_register= ir; the_instruction= i; npc=n;
  };
  bool operator==(const id_ex_format_c&) const;

};

void sc_trace(sc_trace_file *tf,
      const id_ex_format_c& m,
      const sc_string& NAME);

#endif /* ID_EX_FORMAT_C */
```

A.15 id_ex_format_c.cc

```
#include "id_ex_format_c.hh"

void sc_trace(sc_trace_file *tf,
      const id_ex_format_c& m,
      const sc_string& NAME) {

  sc_trace(tf, m.instruction_register, NAME+ ".instruction_register");
  sc_trace(tf, m.the_instruction, NAME+ ".the_instruction");
  sc_trace(tf, m.npc,             NAME+ ".npc");
};

bool id_ex_format_c::operator==(const id_ex_format_c& i) const {

  if ( (instruction_register != i.instruction_register) ||
       (the_instruction != i.the_instruction) ||
       (npc != i.npc) )
    return false;
  else
    return true;

};
```

A.16 ex_mem_format_c.hh

```
#include <scenic.h>
#include "instruction_c.hh"

#ifndef EX_MEM_FORMAT_C
#define EX_MEM_FORMAT_C

class ex_mem_format_c {
public:
  unsigned       instruction_register;
  instruction_c* the_instruction;
  unsigned       npc;
  unsigned       alu_output;
  bool           cond;
  bool           reg_write;

  ex_mem_format_c(void) {
    instruction_register= 0;
    the_instruction=      NULL;
    npc=                  0;
    alu_output=           0;
    cond=                 false;
    reg_write=            false;
  }

  ex_mem_format_c(unsigned ir, instruction_c* i, unsigned n) {
    instruction_register= ir;
    the_instruction=      i;
    npc=                  n;
    alu_output=           0;
    cond=                 false;
    reg_write=            false;
  }

  bool operator==(const ex_mem_format_c&) const;
};

void sc_trace(sc_trace_file *tf,
      const ex_mem_format_c& m,
      const sc_string& NAME);


#endif /* EX_MEM_FORMAT_C */
```

A.17 ex_mem_format_c.cc

```
#include "ex_mem_format_c.hh"

void sc_trace(sc_trace_file *tf,
      const ex_mem_format_c& m,
      const sc_string& NAME) {
  sc_trace(tf, m.instruction_register, NAME+ ".instruction_register");
  sc_trace(tf, m.the_instruction,      NAME+ ".the_instruction");
  sc_trace(tf, m.npc,                  NAME+ ".npc");
  sc_trace(tf, m.alu_output,           NAME+ ".alu_output");
  sc_trace(tf, m.cond,                 NAME+ ".cond");
  sc_trace(tf, m.reg_write,            NAME+ ".reg_write");
};


bool ex_mem_format_c::operator==(const ex_mem_format_c& i) const {

  if ( (instruction_register != i.instruction_register) ||
       (the_instruction != i.the_instruction)           ||
       (npc != i.npc)                                   ||
       (alu_output != i.alu_output)                     ||
       (cond != i.cond)                                 ||
       (reg_write!= i.reg_write)
       )
    return false;
  else
    return true;

};
```

A.18 mem_wb_format_c.hh

```
#include <scenic.h>
#include "instruction_c.hh"

#ifndef MEM_WB_FORMAT_C
#define MEM_WB_FORMAT_C

class mem_wb_format_c {
public:
  unsigned       instruction_register;
  instruction_c* the_instruction;
  unsigned       alu_output;
  unsigned       lmd;
  bool           mem_load;

  mem_wb_format_c(void) {
    instruction_register= 0;
    the_instruction=      NULL;
    alu_output=           0;
    lmd=                  0;
    mem_load=             false;
  }

  mem_wb_format_c(unsigned ir, instruction_c* i, unsigned a) {
    instruction_register= ir;
    the_instruction=      i;
    alu_output=           a;
    lmd=                  0;
    mem_load=             false;
  }

  bool operator==(const mem_wb_format_c&) const;
};

void sc_trace(sc_trace_file *tf,
      const mem_wb_format_c& m,
      const sc_string& NAME);

#endif /* MEM_WB_FORMAT_C */
```

A.19 mem_wb_format_c.cc

```
#include "mem_wb_format_c.hh"

void sc_trace(sc_trace_file *tf,
      const mem_wb_format_c& m,
      const sc_string& NAME) {
  sc_trace(tf, m.instruction_register, NAME+ ".instruction_register");
  sc_trace(tf, m.the_instruction, NAME+ ".the_instruction");
  sc_trace(tf, m.alu_output,      NAME+ ".alu_output");
  sc_trace(tf, m.lmd,             NAME+ ".lmd");
  sc_trace(tf, m.mem_load,        NAME+ ".mem_load");
};

bool mem_wb_format_c::operator==(const mem_wb_format_c& i) const {
  if ( (instruction_register != i.instruction_register) ||
       (the_instruction != i.the_instruction)           ||
       (alu_output != i.alu_output)                     ||
       (lmd != i.lmd)                                   ||
       (mem_load!= i.mem_load)
       )
    return false;
  else
    return true;
};
```

A.21 pc_setting_logic_c.cc

```
#pragma implementation "pc_setting_logic_c.hh"

#include "pc_setting_logic_c.hh"
#include <iostream.h>



/////////////////////////////////////////////////////////
pc_setting_logic_c::pc_setting_logic_c(const char*                  NAME,
  const sc_signal<unsigned>& BRANCH_TARGET_PC,
       const sc_signal<unsigned>& NPC,
       const sc_signal<bool>&     BRANCH,
       sc_signal<unsigned>&       PC
       ): sc_async(NAME),
 branch_target_pc(BRANCH_TARGET_PC),
 npc(NPC),
 branch(BRANCH),
 pc(PC) {
  sensitive(branch_target_pc);
  sensitive(npc);
  sensitive(branch);
}

/////////////////////////////////////////////////////////
void pc_setting_logic_c::entry(void) {

  //  cout << "pc_setting_logic_c::entry() TRIGERRED!!!" << flush;
  bool c= branch.read();

  if (c== false) {
    unsigned npc_val= npc.read();
    pc.write(npc_val);
    cout << "--------------------------------------------------" << endl;
    cout << "pc= " << npc_val << endl;
  }
  else {
    unsigned npc_val= branch_target_pc.read();
    pc.write(npc_val);
    cout << "pc= " << npc_val << endl;
  }
}
```

A.22 forwarding_unit_c.hh

```
#include <scenic.h>
#include "id_ex_format_c.hh"
#include "ex_mem_format_c.hh"
#include "mem_wb_format_c.hh"

#ifndef FORWARDING_UNIT_C
#define FORWARDING_UNIT_C

#pragma interface

class forwarding_unit_c {
private:
public:
  forwarding_unit_c(void);

  void id_forward(instruction_c*,
  const mem_wb_format_c&);

  void ex_forward(id_ex_format_c&,
  const ex_mem_format_c&,
  const mem_wb_format_c&);
};


#endif /* FORWARDING_UNIT_C */
```

A.23 forwarding_unit_c.cc

```
#pragma implementation "forwarding_unit_c.hh"

#include "forwarding_unit_c.hh"
#include "nop_instruction_c.hh"
#include <iostream.h>

/////////////////////////////////////////////////////////
forwarding_unit_c::forwarding_unit_c(void) {
}

/////////////////////////////////////////////////////////
void forwarding_unit_c::id_forward(instruction_c* the_instruction,
   const mem_wb_format_c& last_mem_wb_value) {
  fixed_fields_c    ff;
  nop_instruction_c nop(ff);
  instruction_c* mem_wb_instruction= last_mem_wb_value.the_instruction;

  unsigned src_reg1= the_instruction->fixed_fields.src_reg1;
  unsigned src_reg2= the_instruction->fixed_fields.src_reg2;

  if (mem_wb_instruction== NULL) mem_wb_instruction= (instruction_c*) &nop;

  //////////////////////////////////////////////////
  if (mem_wb_instruction->reg_write()== true) {
    unsigned dest_reg= mem_wb_instruction->fixed_fields.dest_reg;

    cout<<"src1="<<src_reg1<<" src2="<<src_reg2;
    cout<<" dest_reg="<<dest_reg<<endl;

    if (dest_reg!= 0) {
      if (dest_reg== src_reg1) {
the_instruction->fixed_fields.operand_a=
  last_mem_wb_value.alu_output;
cout << "((id) reg from wb stage)------> forward a (";
cout << last_mem_wb_value.alu_output << ")" << endl;
      }
      else if (dest_reg== src_reg2) {
the_instruction->fixed_fields.operand_b=
  last_mem_wb_value.alu_output;
cout << "((id) reg from wb stage)------> forward b (";
cout << last_mem_wb_value.alu_output << ")" << endl;
      }
    }
  }
  //////////////////////////////////////////////////
  else if (mem_wb_instruction->mem_load()== true) {
    unsigned dest_reg= mem_wb_instruction->fixed_fields.dest_reg;

    cout << "src1=" << src_reg1 << " src2=" << src_reg2;
    cout << " dest_reg=" << dest_reg << endl;
    if (src_reg1== dest_reg) {

      the_instruction->fixed_fields.operand_a= last_mem_wb_value.lmd;
      cout<< "((id) mem)--------------------> forward a (";
      cout<< last_mem_wb_value.lmd<<")" << endl;
    }
    else if (src_reg2== dest_reg) {
      the_instruction->fixed_fields.operand_b= last_mem_wb_value.lmd;
      cout<< "((id) mem)--------------------> forward b (";
      cout<< last_mem_wb_value.lmd<<")" << endl;
    }
  }
}

/////////////////////////////////////////////////////////
```

```
void forwarding_unit_c::ex_forward(id_ex_format_c& id_ex_reg,
    const ex_mem_format_c& last_ex_mem_value,
    const mem_wb_format_c& last_mem_wb_value) {
   fixed_fields_c    ff;
   nop_instruction_c nop(ff);
   instruction_c* id_ex_instruction=  id_ex_reg.the_instruction;
   instruction_c* ex_mem_instruction= last_ex_mem_value.the_instruction;
   instruction_c* mem_wb_instruction= last_mem_wb_value.the_instruction;
   if (id_ex_instruction== NULL) id_ex_instruction=  (instruction_c*) &nop;
   if (ex_mem_instruction== NULL) ex_mem_instruction= (instruction_c*) &nop;
   if (mem_wb_instruction== NULL) mem_wb_instruction= (instruction_c*) &nop;

   unsigned src_reg1= id_ex_reg.the_instruction->fixed_fields.src_reg1;
   unsigned src_reg2= id_ex_reg.the_instruction->fixed_fields.src_reg2;

   //////////////////////////////////////////////////
   if (ex_mem_instruction->reg_write()== true) {

     unsigned dest_reg= ex_mem_instruction->fixed_fields.dest_reg;
     cout<<"src1="<<src_reg1<<" src2="<<src_reg2;
     cout<<" dest_reg="<<dest_reg<<endl;

     if (dest_reg!= 0) {
       if (dest_reg== src_reg1) {
id_ex_instruction->fixed_fields.operand_a=
  last_ex_mem_value.alu_output;
cout << "(reg from mem stage)=----> forward a (";
cout << last_ex_mem_value.alu_output << ")" << endl;

       }
       else if (dest_reg== src_reg2) {
id_ex_instruction->fixed_fields.operand_b=
  last_ex_mem_value.alu_output;
cout << "(reg from mem stage)-----> forward b (";
cout << last_ex_mem_value.alu_output << ")" << endl;
       }
     }
   }
   //////////////////////////////////////////////////
   else if (mem_wb_instruction->reg_write()== true) {

     unsigned dest_reg= mem_wb_instruction->fixed_fields.dest_reg;

     cout<<"src1="<<src_reg1<<" src2="<<src_reg2;
     cout<<" dest_reg="<<dest_reg<<endl;

     if (dest_reg!= 0) {
       if (dest_reg== src_reg1) {
id_ex_instruction->fixed_fields.operand_a=
  last_mem_wb_value.alu_output;
cout << "(reg from wb stage)------> forward a (";
cout << last_mem_wb_value.alu_output << ")" << endl;

       }
       else if (dest_reg== src_reg2) {
id_ex_instruction->fixed_fields.operand_b=
  last_mem_wb_value.alu_output;
cout << "(reg from wb stage)------> forward b (";
cout << last_mem_wb_value.alu_output << ")" << endl;
       }
     }
   }
   //////////////////////////////////////////////////
   else if (mem_wb_instruction->mem_load()== true) {

     unsigned dest_reg= mem_wb_instruction->fixed_fields.dest_reg;
```

```
    cout << "src1=" << src_reg1 << " src2=" << src_reg2;
    cout << " dest_reg=" << dest_reg << endl;

    if (src_reg1== dest_reg) {

      id_ex_instruction->fixed_fields.operand_a= last_mem_wb_value.lmd;
      id_ex_reg.the_instruction->print_tag();
      cout<< "(mem)--------------------> forward a (";
      cout<< last_mem_wb_value.lmd<<")" << endl;
    }
    else if (src_reg2== dest_reg) {
      id_ex_instruction->fixed_fields.operand_b= last_mem_wb_value.lmd;
      cout<< "(mem)--------------------> forward b (";
      cout<< last_mem_wb_value.lmd<<")" << endl;
    }
  }
}
```

A.24 fixed_fields_c.hh

```
#ifndef FIXED_FIELDS_C
#define FIXED_FIELDS_C

#pragma interface

class fixed_fields_c {
private: // put everything public for now
public:
  unsigned src_reg1;
  unsigned src_reg2;
  unsigned dest_reg;

  unsigned instruction_register;
  unsigned operand_a;
  unsigned operand_b;

  fixed_fields_c(void) {};
  fixed_fields_c(unsigned ir, unsigned a, unsigned b);
};

#endif /* FIXED_FIELDS_C */
```

A.25 fixed_fields_c.cc

```
#pragma implementation "fixed_fields_c.hh"

#include "fixed_fields_c.hh"

fixed_fields_c::fixed_fields_c(unsigned ir, unsigned a, unsigned b) {

  src_reg1= 0;;
  src_reg2= 0;;
  dest_reg= 0;;

  instruction_register= ir;
  operand_a= a;
  operand_b= b;
}
```

A.26 instruction_c.hh

```
#include "fixed_fields_c.hh"
#include "alu_c.hh"
#include "memory_unit_base_c.hh"
#include "register_file_c.hh"

#include <iostream.h>

#ifndef INSTRUCTION_C
#define INSTRUCTION_C

//#pragma interface

class instruction_c {
protected:

public:
  ////////////////////////////////////////////////////////////////////////////
  // decode table should be read from a file for a kind of semi-"custom"
  // dlx instruction set. This is the order that the opcode will be
  // decoded for now. (should build a minicompiler...)
  // NEED TO MODIFY THE .CC FILE WHEN CHANGING THIS ENUM!!!
  enum instruction_kind_e { ILLEGAL, //     0
    NOP,   //.......1
    HALT,  //       2
    LOAD,  //       3
    STORE, //       4
    J,     //.......5
    BEQZ,  //       6
    BNEZ,  //       7
    ADD,   //       8
    ADDI,  //       9
    ADDU,  //......10
    ADDUI, //      11
    SUB,   //      12
    SUBI,  //      13
    SUBU,  //      14
    SUBUI, //......15
    AND,   //      16
    ANDI,  //      17
    OR,    //      18
    ORI,   //      19
    XOR,   //......20
    XORI   //      21
  };
  /////////////////////////////////////////////////////////////////////

  ////////////////
  fixed_fields_c fixed_fields;

  instruction_c(void) {};
  instruction_c(fixed_fields_c ff) {fixed_fields= ff;}

  void print_tag(void);

  ////////////////////////////////////////////////////////////////////////////
  static fixed_fields_c   decode_fixed_fields(unsigned ir);
  static instruction_c*   get_new_instruction(fixed_fields_c ff,
      unsigned op_a,
      unsigned op_b);

  ////////////////////////////////////////////////////////////////////////////
  // To handle the forwarding, these two methods
  virtual bool      reg_write(void)                       {return false;};
  virtual bool      mem_load(void)                        {return false;};
  ////////////////////////////////////////////////////////////////////////////
```

```
  ///////////////////////////////////////////////////////////////////////////
  // The following three methods are executed in the ex, mem and wb
  // dlx stages respectively.
  virtual bool     execute_cond_output(alu_c<unsigned>&) {assert(false);};
  virtual unsigned execute_alu_output(alu_c<unsigned>&)  {assert(false);};
  virtual unsigned mem_access(unsigned alu_output,
      memory_unit_base_c<unsigned>& data_memory) {
    assert(false);};
  virtual void write_back(unsigned instruction_register,
  unsigned alu_output,
  unsigned lmd,
  register_file_c&)              {assert(false);};
  ///////////////////////////////////////////////////////////////////////////
};

#endif /* INSTRUCTION_C */
```

A.27 instruction_c.cc

```
#include "instruction_c.hh"
#include <iostream.h>

#include "fixed_fields_c.hh"

#include "nop_instruction_c.hh"
#include "halt_instruction_c.hh"
#include "load_instruction_c.hh"
#include "store_instruction_c.hh"
#include "branch_instruction_c.hh"
#include "alu_instruction_c.hh"

// Modify this method when adding new instructions...
void instruction_c::print_tag(void) {
  const char* instruction_kind_string[22]= { "ILLEGAL", //.....0
      "NOP",   //       1
      "HALT",  //       2
      "LOAD",  //       3
      "STORE", //       4
      "J",     //.......5
      "BEQZ",  //       6
      "BNEZ",  //       7
      "ADD",   //       8
      "ADDI",  //       9
      "ADDU",  //......10
      "ADDUI", //      11
      "SUB",   //      12
      "SUBI",  //      13
      "SUBU",  //      14
      "SUBUI", //......15
      "AND",   //      16
      "ANDI",  //      17
      "OR",    //      18
      "ORI",   //      19
      "XOR",   //......20
      "XORI"   //      21
  };
  unsigned opcode= fixed_fields.instruction_register >> 26;
  cout << instruction_kind_string[opcode] << " " <<flush;
}

/////////////////////////////////////////////////////////
fixed_fields_c instruction_c::decode_fixed_fields(unsigned ir){
  // See H&P book for DLX instruction format
  fixed_fields_c ff;

  ff.instruction_register= ir;
  ff.src_reg1=             (ir & 0x03e00000) >> 21;
  ff.src_reg2=             (ir & 0x001f0000) >> 16;

  return ff;
}

/////////////////////////////////////////////////////////
instruction_c* instruction_c::get_new_instruction(fixed_fields_c ff,
  unsigned op_a,
  unsigned op_b) {

  switch (ff.instruction_register >> 26) {
    case instruction_c::ILLEGAL:
    cout << "decode illegal instruction? (will assert(false));" << endl;
    assert(false);
    break;
  case instruction_c::NOP:
    return (instruction_c*) (new nop_instruction_c(ff));
```

```
      break;
    case instruction_c::HALT:
      return (instruction_c*) (new halt_instruction_c(ff));
      break;
    case instruction_c::LOAD:
      return (instruction_c*) (new load_instruction_c(ff));
      break;
    case instruction_c::STORE:
      return (instruction_c*) (new store_instruction_c(ff, op_a));
      break;
    case instruction_c::J:
    case instruction_c::BEQZ:
    case instruction_c::BNEZ:
      return (instruction_c*) (new branch_instruction_c(ff, op_a));
      break;
    case instruction_c::ADD:
    case instruction_c::ADDI:
    case instruction_c::SUB:
    case instruction_c::SUBI:
    case instruction_c::AND:
    case instruction_c::ANDI:
    case instruction_c::OR:
    case instruction_c::ORI:
    case instruction_c::XOR:
    case instruction_c::XORI:
      return (instruction_c*) (new alu_instruction_c(ff, op_a, op_b));
      break;
      // add new instruction decoding here...
      // match with enum in instruction_c.hh
    default:
      cout << "| ID: Unimplemented instruction, will assert false..." << flush;
      assert(false);
      break;

  }
  return (instruction_c*) NULL; // dummy
}
```

A.28 load_instruction_c.hh

```
#include <scenic.h>
#include "instruction_c.hh"

#ifndef LOAD_INSTRUCTION_C
#define LOAD_INSTRUCTION_C

#pragma interface

class load_instruction_c: public instruction_c {
private:
public:

  load_instruction_c(fixed_fields_c ff);

  virtual bool     mem_load(void) {return true;};

  virtual bool     execute_cond_output(alu_c<unsigned>&);
  virtual unsigned execute_alu_output(alu_c<unsigned>&);
  virtual unsigned mem_access(unsigned alu_output,
  memory_unit_base_c<unsigned>& data_memory);
  virtual void write_back(unsigned instruction_register,
  unsigned alu_output,
  unsigned lmd,
  register_file_c&);
};

#endif /* LOAD_INSTRUCTION_C */
```

A.29 load_instruction_c.cc

```
#pragma implementation "load_instruction_c.hh"

#include "load_instruction_c.hh"


///////////////////////////////////////////////////////////
load_instruction_c::load_instruction_c(fixed_fields_c ff): instruction_c(ff) {

  fixed_fields.src_reg1=
    (fixed_fields.instruction_register & 0x03e00000) >> 21;
  fixed_fields.dest_reg=
    (fixed_fields.instruction_register & 0x001f0000) >> 16;

};

///////////////////////////////////////////////////////////
bool load_instruction_c::execute_cond_output(alu_c<unsigned>&) {
  return false;
}

///////////////////////////////////////////////////////////
unsigned load_instruction_c::execute_alu_output(alu_c<unsigned>& alu) {
  // calculate memory address
  unsigned immediate_value= (fixed_fields.instruction_register & 0x0000ffff);
  return alu.add(fixed_fields.operand_a, immediate_value);
}


///////////////////////////////////////////////////////////
unsigned load_instruction_c::mem_access(unsigned alu_output,
  memory_unit_base_c<unsigned>& data_memory) {
  return data_memory.read(alu_output);
}


///////////////////////////////////////////////////////////
void load_instruction_c::write_back(unsigned instruction_register,
    unsigned /*alu_output*/,
    unsigned lmd,
    register_file_c& regfile) {

  cout << " reg#"<< fixed_fields.dest_reg <<" = "<< lmd << " " << endl;
  regfile.d[ fixed_fields.dest_reg ].write(lmd);
}
```

A.30 store_instruction_c.hh

```
#include <scenic.h>
#include "instruction_c.hh"

#ifndef STORE_INSTRUCTION_C
#define STORE_INSTRUCTION_C

#pragma interface

class store_instruction_c: public instruction_c {
private:
public:

  store_instruction_c(fixed_fields_c ff, unsigned op_a);

  virtual bool     execute_cond_output(alu_c<unsigned>&);
  virtual unsigned execute_alu_output(alu_c<unsigned>&);
  virtual unsigned mem_access(unsigned alu_output,
      memory_unit_base_c<unsigned>& data_memory);
  virtual void write_back(unsigned instruction_register,
  unsigned alu_output,
  unsigned lmd,
  register_file_c&);

};

#endif /* STORE_INSTRUCTION_C */
```

A.31 store_instruction_c.cc

```
#pragma implementation "store_instruction_c.hh"

#include "store_instruction_c.hh"

store_instruction_c::store_instruction_c(fixed_fields_c ff,
 unsigned op_a):
  instruction_c(ff) {

  fixed_fields.src_reg1=
    (fixed_fields.instruction_register & 0x03e00000) >> 21;
  fixed_fields.operand_a= op_a;

};

/////////////////////////////////////////////////////////////
bool store_instruction_c::execute_cond_output(alu_c<unsigned>&) {
  return false;
}

/////////////////////////////////////////////////////////////
unsigned store_instruction_c::execute_alu_output(alu_c<unsigned>& alu) {
  // calculate memory address
  unsigned immediate_value= (fixed_fields.instruction_register & 0x0000ffff);
  return alu.add(fixed_fields.operand_a, immediate_value);
}


/////////////////////////////////////////////////////////////
unsigned store_instruction_c::mem_access(unsigned alu_output,
 memory_unit_base_c<unsigned>& data_memory) {
  data_memory.write(alu_output, fixed_fields.operand_b);
  return 0;
}


/////////////////////////////////////////////////////////////
void store_instruction_c::write_back(unsigned /*instruction_register*/,
     unsigned /*alu_output*/,
     unsigned /*lmd*/,
     register_file_c& /*regfile*/) {
  // do nothing for store inst...
}
```

A.32 alu_instruction_c.hh

```
#include <scenic.h>
#include "instruction_c.hh"

#ifndef ALU_INSTRUCTION_C
#define ALU_INSTRUCTION_C

#pragma interface

class alu_instruction_c: public instruction_c {
private:
public:

  alu_instruction_c(fixed_fields_c ff, unsigned op_a, unsigned op_b);

  virtual bool     reg_write(void) {return true;};
  virtual bool     execute_cond_output(alu_c<unsigned>&);
  virtual unsigned execute_alu_output(alu_c<unsigned>&);
  virtual unsigned mem_access(unsigned alu_output,
      memory_unit_base_c<unsigned>& data_memory);
  virtual void write_back(unsigned instruction_register,
  unsigned alu_output,
  unsigned lmd,
  register_file_c&);
};


#endif /* ALU_INSTRUCTION_C */
```

A.33 alu_instruction_c.cc

```
#pragma implementation "alu_instruction_c.hh"

#include "alu_instruction_c.hh"
#include <assert.h>
#include <iostream.h>

/////////////////////////////////////////////////////////////
alu_instruction_c::alu_instruction_c(fixed_fields_c ff,
     unsigned op_a,
     unsigned op_b) : instruction_c(ff) {

  fixed_fields.src_reg1=
    (fixed_fields.instruction_register & 0x03e00000) >> 21;
  fixed_fields.operand_a= op_a;

  // switch opcode (watch if the ff in the instruction is ok
  // because of the constructor invocation order)
  switch ( fixed_fields.instruction_register >> 26 ) {
  case instruction_c::ADDI:
  case instruction_c::SUBI:
  case instruction_c::ANDI:
  case instruction_c::ORI:
  case instruction_c::XORI:
    fixed_fields.src_reg2= 0;
    fixed_fields.operand_b= (fixed_fields.instruction_register & 0x0000ffff);
    fixed_fields.dest_reg=
      (fixed_fields.instruction_register & 0x001f0000) >> 16;
    break;
  case instruction_c::ADD:
  case instruction_c::SUB:
  case instruction_c::AND:
  case instruction_c::OR:
  case instruction_c::XOR:
    fixed_fields.src_reg2=
      (fixed_fields.instruction_register & 0x001f0000) >> 16;
    fixed_fields.operand_b= op_b;
    fixed_fields.dest_reg=
      (fixed_fields.instruction_register & 0x0000f800) >> 11;
    break;
  default:
    assert(false);
    break;
  }
};

/////////////////////////////////////////////////////////////
bool alu_instruction_c::execute_cond_output(alu_c<unsigned>&) {
  return false;
}

/////////////////////////////////////////////////////////////
unsigned alu_instruction_c::execute_alu_output(alu_c<unsigned>& alu) {
  unsigned alu_result;

  switch (fixed_fields.instruction_register >> 26 ) {
  case instruction_c::ADD:
  case instruction_c::ADDI:
    alu_result= alu.add(fixed_fields.operand_a, fixed_fields.operand_b); break;
  case instruction_c::SUB:
  case instruction_c::SUBI:
    alu_result= alu.sub(fixed_fields.operand_a, fixed_fields.operand_b); break;
  case instruction_c::AND:
  case instruction_c::ANDI:
    alu_result= alu.and(fixed_fields.operand_a, fixed_fields.operand_b); break;
  case instruction_c::OR:
```

```
   case instruction_c::ORI:
     alu_result= alu.or(fixed_fields.operand_a, fixed_fields.operand_b); break;
   case instruction_c::XOR:
   case instruction_c::XORI:
     alu_result= alu.or(fixed_fields.operand_a, fixed_fields.operand_b); break;
  default:
     assert(false);
     break;
   }
   print_tag();
   cout << " op a="<<fixed_fields.operand_a;
   cout << " op b="<<fixed_fields.operand_b;
   cout << " alu_result= "<<alu_result<<endl;
   return alu_result;

}

//////////////////////////////////////////////////////////
unsigned alu_instruction_c::mem_access(unsigned,
       memory_unit_base_c<unsigned>& ) {
  // do nothing...
  return 0;
}

//////////////////////////////////////////////////////////
void alu_instruction_c::write_back(unsigned instruction_register,
   unsigned alu_output,
   unsigned /*lmd*/,
   register_file_c& regfile) {
  print_tag();
  cout << " reg#"<< "=" << fixed_fields.dest_reg << endl;

  regfile.d[ fixed_fields.dest_reg ].write(alu_output);
}
```

A.34 branch_instruction_c.hh

```
#include <scenic.h>
#include "instruction_c.hh"

#ifndef BRANCH_INSTRUCTION_C
#define BRANCH_INSTRUCTION_C

#pragma interface

class branch_instruction_c: public instruction_c {
private:
public:

  branch_instruction_c(fixed_fields_c ff, unsigned op_a);

  virtual bool     execute_cond_output(alu_c<unsigned>&);
  virtual unsigned execute_alu_output(alu_c<unsigned>&);
  virtual unsigned mem_access(unsigned alu_output,
      memory_unit_base_c<unsigned>& data_memory);
  virtual void write_back(unsigned instruction_register,
  unsigned alu_output,
  unsigned lmd,
  register_file_c&);

};


#endif /* BRANCH_INSTRUCTION_C */
```

A.35 branch_instruction_c.cc

```
#pragma implementation "branch_instruction_c.hh"
#include "branch_instruction_c.hh"

/////////////////////////////////////////////////////////
branch_instruction_c::branch_instruction_c(fixed_fields_c ff, unsigned op_a):
  instruction_c(ff) {
  //  switch opcode
  switch ( fixed_fields.instruction_register >> 26 ) {
  case instruction_c::BEQZ:
  case instruction_c::BNEZ:
    fixed_fields.src_reg1=
      (fixed_fields.instruction_register & 0x03e00000) >> 21;
    fixed_fields.operand_a= op_a;
    break;
  case instruction_c::J:
  default:
    assert(false);
    break;
  }
};

/////////////////////////////////////////////////////////
bool branch_instruction_c::execute_cond_output(alu_c<unsigned>&) {
  // evaluate condition according to the opcode
  bool return_value;
  //  switch opcode
  switch ( fixed_fields.instruction_register >> 26 ) {
  case instruction_c::BEQZ:
    return_value = fixed_fields.operand_a == 0;
    break;
  case instruction_c::BNEZ:
    return_value = fixed_fields.operand_a != 0;
    break;
  case instruction_c::J:
  default:
    assert(false);
    return_value= 0;
    break;
  }
  return return_value;
}

/////////////////////////////////////////////////////////
unsigned branch_instruction_c::execute_alu_output(alu_c<unsigned>& alu) {
  // calculate memory address (check that carefully to be sure...)
  //return alu.add(fixed_fields.operand_a, fixed_fields.immediate_value);
  unsigned immediate_value= (fixed_fields.instruction_register & 0x0000ffff);
  return immediate_value;
}

/////////////////////////////////////////////////////////
unsigned branch_instruction_c::mem_access(unsigned alu_output,
 memory_unit_base_c<unsigned>& ) {
  return 0;
}

/////////////////////////////////////////////////////////
void branch_instruction_c::write_back(unsigned /*instruction_register*/,
      unsigned /*alu_output*/,
      unsigned /*lmd*/,
      register_file_c& /*regfile*/) {
}
```

A.36 nop_instruction_c.hh

```
#include <scenic.h>
#include "instruction_c.hh"

#ifndef NOP_INSTRUCTION_C
#define NOP_INSTRUCTION_C

#pragma interface

class nop_instruction_c: public instruction_c {
private:
public:

  nop_instruction_c(fixed_fields_c ff): instruction_c(ff) {};

  virtual bool     execute_cond_output(alu_c<unsigned>&);
  virtual unsigned execute_alu_output(alu_c<unsigned>&);
  virtual unsigned mem_access(unsigned alu_output,
      memory_unit_base_c<unsigned>& data_memory);
  virtual void write_back(unsigned instruction_register,
  unsigned alu_output,
  unsigned lmd,
  register_file_c&);
};
#endif /* NOP_INSTRUCTION_C */
```

A.37 nop_instruction_c.cc

```
#pragma implementation "nop_instruction_c.hh"

#include "nop_instruction_c.hh"
#include <assert.h>
#include <iostream.h>

/////////////////////////////////////////////////////////////
bool nop_instruction_c::execute_cond_output(alu_c<unsigned>&) {
  return false;
}

/////////////////////////////////////////////////////////////
unsigned nop_instruction_c::execute_alu_output(alu_c<unsigned>&) {
  return 0;
}

/////////////////////////////////////////////////////////////
unsigned nop_instruction_c::mem_access(unsigned,
       memory_unit_base_c<unsigned>& ) {
  return 0;
}

/////////////////////////////////////////////////////////////
void nop_instruction_c::write_back(unsigned /*instruction_register*/,
    unsigned  /*alu_output*/,
    unsigned  /*lmd*/,
    register_file_c& /*regfile*/) {
}
```

A.38 halt_instruction_c.hh

```
#include <scenic.h>
#include "instruction_c.hh"

#ifndef HALT_INSTRUCTION_C
#define HALT_INSTRUCTION_C

#pragma interface

class halt_instruction_c: public instruction_c {
private:
public:

  halt_instruction_c(fixed_fields_c ff): instruction_c(ff) {};

  virtual bool     execute_cond_output(alu_c<unsigned>&);
  virtual unsigned execute_alu_output(alu_c<unsigned>&);
  virtual unsigned mem_access(unsigned alu_output,
      memory_unit_base_c<unsigned>& data_memory);
  virtual void write_back(unsigned instruction_register,
  unsigned alu_output,
  unsigned lmd,
  register_file_c&);
};
#endif /* HALT_INSTRUCTION_C */
```

A.39 halt_instruction_c.cc

```
#pragma implementation "halt_instruction_c.hh"

#include "halt_instruction_c.hh"
#include <iostream.h>
#include <stdlib.h>
/////////////////////////////////////////////////////////////
bool halt_instruction_c::execute_cond_output(alu_c<unsigned>&) {
  return false;
}

/////////////////////////////////////////////////////////////
unsigned halt_instruction_c::execute_alu_output(alu_c<unsigned>&) {
  return 0;
}

/////////////////////////////////////////////////////////////
unsigned halt_instruction_c::mem_access(unsigned,
       memory_unit_base_c<unsigned>& ) {
  // do nothing...
  return 0;
}

/////////////////////////////////////////////////////////////
void halt_instruction_c::write_back(unsigned /*instruction_register*/,
    unsigned  /*alu_output*/,
    unsigned  /*lmd*/,
    register_file_c& /*regfile*/) {
  cout<<"halt instruction wb stage"<<endl;
  exit(1);
}
```

A.40 main.cc

```
#include <scenic.h>
#include <stdlib.h>

#include "dlx_pipeline_c.hh"

int scenic(int ac, char *av[])
{
  sc_clock clock("CLK", 20.0, 0.5, 0.0, true);

  sc_signal<bool> reset;
  register_file_c register_file;
  memory_unit_base_c<unsigned> program_memory;
  memory_unit_base_c<unsigned> data_memory;

  program_memory.initialize("progmem.dlx");
  data_memory.initialize("datamem.dlx");
  register_file.reset();

  dlx_pipeline_c dlx_pipeline("DLX_PIPELINE",
      clock.pos(),
      register_file,
      program_memory,
      data_memory,
      reset);

  sc_trace_file* trace_file= sc_create_wif_trace_file("trace");
  clock.trace(trace_file);
  sc_trace(trace_file, register_file.pc, "PC");
  sc_trace(trace_file, register_file.d[1], "R1");
  sc_trace(trace_file, register_file.d[2], "R2");
  sc_trace(trace_file, dlx_pipeline.if_id_reg, "IF_ID_REG");
  sc_trace(trace_file, dlx_pipeline.id_ex_reg, "ID_EX_REG");
  sc_trace(trace_file, dlx_pipeline.ex_mem_reg, "EX_MEM_REG");
  sc_trace(trace_file, dlx_pipeline.mem_wb_reg, "MEM_WB_REG");

  if (ac==1)
    sc_clock::start(-1);
  else
    sc_clock::start(atoi(av[1]));

  return 0;
}
```