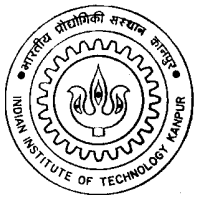
High Level Synthesis from Sim-nML Processor Specifications

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by
Souvik Basu



to the
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

Aug, 2001

Certificate

This is to certify that the work contained in the thesis entitled "*High Level Synthesis from Sim-nML Processor Specifications*", by *Souvik Basu*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Aug, 2001

(Dr. Rajat Moona)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Abstract

The design of modern embedded systems require automated modeling tools for faster design and for the study of various design trade-offs. These tools put together constitutes an integrated environment, where the designer can write the high level design specification and the tools will automatically generate the required hardware and software for the embedded system. Sim-nML is one of these types of specification based development system, encircling which several tools have been developed earlier.

In this thesis, we have developed a high level synthesis system based on Sim-nML processor ISA specification language. High level synthesis or behavioral synthesis deals with the problem of transforming a behavioral specification of a digital system to register-transfer level (RTL) implementation. Tools have been developed for behavioral and structural high level synthesis. Behavioral high level synthesis transforms Sim-nML specifications of processors to the corresponding behavioral Verilog models. These behavioral Verilog models are suitable for fast functional simulation using standard Verilog simulators. Structural high level synthesis generates structural synthesizable Verilog processor models from the corresponding Sim-nML specifications. The structural model is suitable for both functional simulation and synthesis to low level Verilog netlist. Architecture of the structural design is non-pipelined and takes multiple clock cycles to execute an instruction. The generated behavioral and structural Verilog models are compliant with the current industry standard simulation and synthesis tools.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

High level synthesis or behavioral synthesis deals with the problem of transforming a behavioral specification of a digital system to register-transfer level (RTL) implementation. The behavioral specification of the digital system may be the system architecture, or the algorithmic behavior of the system specified in some high level language. As the Very Large Scale Integrated circuits (VLSI) technology is currently providing space for multiple million gates of random logic per chip, it is getting difficult to design such large circuits using traditional *capture-and-simulate* design methodology. Also with the fast moving technology, time to market is one prime concern for the design along with chip area, power and speed. To shorten the product development cycle, newer *describe-and-synthesize* methodology is gaining more importance.

Describe-and-synthesize methodology [10] essentially moves design automation on higher levels of abstraction which makes design cycle shorter and allows more design alternatives to explore. The efficient synthesis tools can even out-perform average human designers in meeting the design constraints. Synthesis process is similar to compiling programs written in high level languages like C or Pascal to assembly language. Each component in the generated lower level of abstraction translates to components that provide a more detailed description of the system. Thus, each stage in synthesis is a design refinement process.

Sim-nML [34] is a language that can be used to specify a programmable processor at the level of its Instruction Set Architecture (ISA). The ISA specification of the processor includes the assembly language syntax, binary image and semantic behavior of the instructions . A resource usage model is currently under development, which captures some of structural constructs of the processor including complex pipelining etc. The *Sim-nML* processor specification is behavioral in nature. An integrated

development environment encircling the *Sim-nML* processor specification language is in the process of development, which includes the generation of retargetable assemblers [25], disassemblers [21], compiler back-ends [33], functional simulators [4], retargetable cache simulators [34], profilers [34] etc. In this thesis, a technique is developed to generate the structural and behavioral model of a programmable processor in Verilog HDL from its *Sim-nML* specification. The structural model is suitable for both synthesis and simulation. Similarly, the behavioral model is suitable for fast simulation. As *Sim-nML* specifies a programmable processor at its ISA level, the hardware generation methodology is suitable for Application Specific Instruction Processor (ASIP) generation.

1.2 Goals Achieved

The goals achieved in the thesis are as follows.

- From *Sim-nML* specification, behavioral Verilog model of the processor is generated. The generated Verilog description is a collection of Verilog statements suitable for fast functional simulation.
- A technique is developed to generate the structural synthesizable Verilog model of processors from *Sim-nML* specifications to a particular target architecture. The Verilog description can be synthesized to get the netlist of hardware structures.

1.3 High Level Synthesis

High Level Synthesis (HLS) is the transition from the algorithmic level specification of behavior of a digital system to a Register-Transfer Level (RTL) structure that implements the behavior. The input to HLS can be description of ISA or an algorithm written in some high level language. The output from a HLS is a connection of data path elements and a Finite State Machine (FSM) that implements the control path. The RTL level data path for the processor is composed of three types of components - functional units (e.g. ALUs, multipliers, and shifters), storage units (e.g. registers and memories) and interconnection units (e.g. buses and multiplexors). The FSM of the control path can be realized by a hardwired logic or by a microprogrammed control unit. The control path triggers the appropriate data path elements in synchronization with clock thereby implementing the functionality of the processor.

1.3.1 Input of High Level Synthesis System

The input description of a HLS can be behavioral specification of a digital system written in Verilog, VHDL, C or any suitable procedural language, processor architecture description languages like *Sim-nML* [34], LISA [50], MIMOLA [27], ISDL [14], nML [9] or any other form. There are several other languages proposed for describing hardware at varied level of abstractions, including some declarative languages and some higher level of system level behavioral languages. Examples of these languages include SystemC [45] and Esterel [48].

1.3.2 High Level Synthesis Stages

The HLS is typically carried out in five stages - compilation, transformation, scheduling, resource allocation and binding.

Input Description Compilation

The input behavioral description is compiled into an internal representation suitable for several high level transformations and optimizations employed in the subsequent HLS stages. The most used internal format is graph based representation where the data and control flow of the input (assuming procedural style of description) is stored, preserving the dependency and sequentiality of the input. The representation can be made in Data Flow Graph (DFG) or in Control Flow Graph (CFG) or in combined Control-Data Flow Graph (CDFG). Other intermediate representations include petri net and extensions of petri net[49] etc.

Transformation of Internal Representation

The transformation phase is one of the important stages, in which several optimizations are performed on the given input. The hardware performs operations in parallel. If the behavioral specification do not express the parallelism, it should be extracted from the specification. The objective of transformation is generally to minimize the silicon area of the generated chip and maximize the speed. Some other objectives could also be to optimize transformations for testability, low power consumption and reliability. The transformations are similar to the regular compiler optimizations along with some hardware specific optimizations. The general transformations include the following.

- **Temporary Variable Elimination :** For ease of description, the input behavioral description contains several temporary variables. These variables result in hardware registers, which would mean that the resultant hardware occupies

extra area. To be noted that, all temporary variables can't be removed due to underlying architectural constraints as explained in the later chapters.

- **Common Subexpression Elimination :** The parts of the code that are repeated are factored out. Thus, the hardware operations are needed to be performed only once, which reduces area. However, this transformation can reduce the speed of operation because otherwise the operations can be done in parallel with additional resources.

- **Dead Code elimination :** Removing dead code, i.e. code that serves no computational purpose, thus resulting in reduction of unnecessary hardware.

- **Expression Simplification :** Expressions are evaluated so that the operations may be done in a smaller number of steps. This transformation is done in compliance to the underlying architecture.

- **Constant Propagation :** If some value of constant is known then the values can be used to simplify the flow of the description, there by reducing the number of operations.

- **Loop Unrolling :** This is an important transformation, as hardware structure can't support loop directly. For input languages that support loops [1], the loop body can be replicated and if possible, some optimizations can be performed in the unrolled loop.

- **Hardware Specific Transformations :** These are optimizations not commonly used in compilers. These are however necessary in the hardware designs. One such optimization is to change the description such that it uses functions, which may be performed directly by the hardware. An example is given below. The expression 0.3333 + 2.6664*X could be simplified to 0.33*(1 + 8*X). In the simplified case, the multiplication by eight can be done by shift and the addition of one can be done using an increment.

As described above all the transformations are not target architecture independent. For example if the target architecture does not support *multiply and accumulate* as a functional unit then the *multiply and accumulate* operations must be divided in two sub operations, incorporating a temporary variable to hold the intermediate value. Contrary to this if the target architecture supports *multiply and accumulate* functional unit, there is no need to incorporate temporary variables to hold these values.

[1] In our high level synthesis *Sim-nML* language does not support loop construct

Scheduling

Scheduling assigns the operations in behavioral description into control steps. A control step usually corresponds to a cycle in the system clock, the basic time unit of the synchronous digital systems. The scheduling is constrained by the user according to the available resources or the maximum delay (i.e the speed of the digital system) or both. If no constraints are specified then it is possible to get the fastest hardware implementation exploiting maximum parallelism and using as many numbers of functional, storage and interconnection units. Such an implementation however requires the maximum area with the constraints it would have been possible to generate hardware with small area, using minimum number of functional, storage and/or interconnection units. The hardware implementation in this way may or may not have any parallelism in the operations. The generated hardware may or may not be the minimum area implementation, depending upon the area ratio of the functional and interconnection units.

In scheduling the total number of control steps necessary, is dependent on the constraints. If higher speed is required, less number of control steps are used and more operations are scheduled in each control step. This results in large functional units and silicon area of the hardware. If less area in the resultant hardware is required then less number of functional units are available in the generated hardware. Thus, less numbers of operations are performed in each control steps and low speed hardware is generated. In this way, scheduling determines the tread off between the design cost and performance. One important thing to remember that all the pre-defined scheduling constraints in HLS may not satisfied in scheduling stages due the architectural properties, available hardware components (will be allocated in next stage) etc. In that case user has to provide new set of constraints or objective function in the HLS and has to perform scheduling again to check the suitability of output with the objective function.

There are several approaches to solve all or particular class of scheduling problems in HLS. In general, Integer Linear Programming (ILP) formulation is correct for resource-constrained and time-constrained scheduling problems [20]. But as optimum scheduling problems are NP complete, the execution time of algorithms grow exponentially with the number of variables and number of inequalities in the formulated ILP. Thus for large practical problems heuristics have been developed that run efficiently maintaining the scheduling goals. Heuristic scheduling algorithms are of two classes - constructive approach and iterative refinement approach. There are several algorithms of each classes, where each of them differs in the input criteria and the next heuristic stage selection. The simplest constructive approach is the *As Soon As Possible* (ASAP) or eager scheduling. First the operations are stored in a list according to their topological order. Then operations are taken from the list one at a time and placed in the earliest possible control step. Similar to this another constructive heuristic approach is *As Last As Possible* (ALAP) or lazy scheduling. In

this scheduling, the operations are stored in the list, but scheduler tries to schedule the operation at the latest control stage. To make the hardware faster the delay in the *critical path* (the longest path in terms of control steps) is to be minimized. ASAP and ALAP scheduling scheme do not consider the *critical path* in choosing the next step. *List scheduling* which is another constructive approach solves the *critical path* problem by keeping a list of each operation that has not yet been selected at each of the control step. The list ordering is maintained by a priority function, which forms the global scheduling criteria. The priority function in the list scheduling can be chosen in several ways. Some examples of choosing priority functions are mobility [31], which is defined as the difference between the ASAP and ALAP scheduled values of an operation. Another example of priority function is urgency [11], which is defined as the minimum number of control steps from the bottom at which an operation can be scheduled before the timing constraint is violated etc. There are other examples of scheduling such as Force Directed Scheduling (FDS) etc.

- *Resource Allocation and Binding*

After scheduling, the next operation is to allocate resources from the component database. The allocation is done according to the scheduling while maintaining the scheduling order and preserving the constraints. The component database library may contain several types of functional units with different area, speed, power consumption, architectural variations such as pipelining or non-pipelining, storage units with different area, speed, power consumption and interconnection units. Resource allocator searches the component database and allocates suitable resource from it. The resource binding is final assignment of hardware resources to the scheduled operations, from the allocated set of resource. At this level variables are assigned to storage units. During this, variable lifetime is analyzed and resource binding is done. In optimized allocation and binding two variables may share the same storage resource if they are not accessed or altered in the same control step, i.e. , the variables are mutually exclusive. Operations are assigned to allocated functional units. Each functional unit can only execute one operation in one control step. Interconnection binding binds interconnections between storage and functional units. Typical interconnection units used are buses and multiplexers.

1.4 ASIC/ASIP Design Flow

The design of an Application Specific Integrated Circuit (ASIC) or Application Specific Instruction Processor (ASIP) starts from the behavioral description of the digital system, which includes the algorithm for the ASIC or the instruction set architecture of the ASIP. Our HLS methodology is suitable for the ASIP generation. In our

methodology, the instruction set of the ASIP can be specified in *Sim-nML* language.

The broad stages in the ASIC/ASIP design flow include the followings.

- **Behavioral Specification :** The behavior of the digital system is described in a suitable language.
- **High Level Synthesis :** Transformation of behavior to suitable hardware architecture while performing Design Space Exploration (DSE). In HLS, there can be one or multiple target architectures in which the input behaviors can be targeted to generate the structure. This is also known as behavioral synthesis.
- **Simulation of HLS Generated RTL Netlist :** The HLS generated netlist is simulated to verify the functional correctness for several test cases.
- **Logic Synthesis :** This is the next stage of synthesis where the architecture is more elaborated and several logic synthesis optimizations are performed. The design is finally mapped to a particular technology library provided by the semiconductor vendor. The output gives accurate measures of area, speed, power requirements etc.
- **Static Timing Analysis :** This is performed after inserting clock tree and clock buffers. The accurate timing analysis is done to verify the timing requirements.
- **Simulation of RTL Netlist :** The netlist is simulated after the logic synthesis to verify the functional equivalence with the high level synthesis generated netlist.
- **Floorplanning, Place & Route :** The chip floorplan is designed and chip modules are placed with proper routing, maintaining the timing and other functional constraints. Chip input/output guard rings etc. are also prepared for external interfacing.
- **Masking and Prototyping :** From the place & route data, layout masks are prepared and the chip is taped out.

Each of the above mentioned stages are collections of several sub-stages and each of them are quite complex in nature. Our methodology of ASIP generation is important as this can be extended to suitable co-design methodology of both hardware and software generation. This infers, from the instruction set description in *Sim-nML*, several system software can be automatically generated. For ASIP development there are methodologies to automatically generate ISA after analyzing the particular application domain requirements [6] [12]. These methodologies can be integrated with our *Sim-nML* based methodologies for rapid co-design of ASIP and related software developments.

1.5 Organization of The Thesis

Organization of the rest of the thesis is as follows. A survey of the related researches in this problem area is provided in *Chapter 2*. The design of our hardware generation system is given in *Chapter 3* and the implementation is discussed in *Chapter 4*. Finally results of the work are shown in *Chapter 5* where we have also drawn the conclusion.

The setup of Synopsys Design Compiler environment is provided in *Appendix A*. *Appendix B* gives the example of the structural data path of a hypothetical processor. *Appendix C* gives a section of behavioral synthesis generated Verilog code of Motorola 68HC11 processor. *Appendix D* gives an example of the simulation top-level module.

Chapter 2

Related Works

2.1 Introduction

Several research projects have been carried out in the area of High Level Synthesis (HLS) in the past and several projects are ongoing. While all HLS systems generate the hardware from high level specifications, the objective of two different projects may be different. Some projects aim at the minimization of area, while some other aim at the maximization of speed or minimization of power or a mix of these.

In this chapter, the research projects in HLS are broadly distinguished in two groups, based on the types of input specifications. The first group comprises of HLS systems that synthesize the hardware from specifications of Instruction Set Architecture (ISA) of a programmable processor. The second group comprises of the HLS systems that synthesize the hardware from algorithmic specifications of a digital system. Our approach of HLS from *Sim-nML* processor specifications falls in the first group. The *Sim-nML* processor specifications are also suitable for generating other system software as mentioned in *Chapter 1*.

ASIC generation from the algorithmic specifications of functionality falls in the second group. The second approach is capable of synthesizing ASICs, ASIPs, DSPs and general purpose processors. In that sense, the second approach supports a broad range of digital system HLS. However, it is not suitable for generating system software etc. from the input specifications. Thus, this approach can not provide an integrated methodology like the first one.

2.2 High Level Synthesis from Processor ISA Specifications

Some of the HLS systems that take processor ISA specifications as input are described below.

MIMOLA [27] hardware specification language, developed at University of Dortmund, Germany can be used to write structural specifications of a programmable processor at low level, exposing several hardware details. Hardware is then synthesized from MIMOLA specifications. MIMOLA being a low level specification, the hardware generation method is easier. MSS [28] is a MIMOLA based hardware synthesis system that can also take behavioral VHDL specifications as input.

ISDL [14], developed at MIT LCS is another programmable processor instruction set architecture specification language, which describes the behavior of a processor in attribute grammar notation. The language is suitable for general purpose programmable processor, but special emphasis has been given for VLIW architecture based processor specifications. In ISDL, the parallelism is explicitly specified using illegal instruction grouping and it is used for the generation of the parallel hardware. A synthesis tool HGEN has been developed that generates synthesizable Verilog for the underlying VLIW architecture from ISDL specifications.

nML [9] processor instruction set specification language, developed at TU Berlin has been used for hardware generation [8]. From the attribute grammar based representation, hardware elements have been generated. The nML language is similar to the *Sim-nML*, but the design of the system to generate hardware is very different from our work. In our work we have produced the intermediate flattened *Sim-nML* description and mapped it to fixed data path architecture. In nML the hardware modules 'HME's and 'HMC's are generated from the non flattened representation of the processor specifications.

LISA [50] processor specification language, developed at Aachen University of Technology, Germany is used to specify programmable processors. The processor specifications capture the instruction behavior along with several structural information, like pipelining etc. Structural information is specified using reservation tables and used in the hardware synthesis. VHDL hardware models have been synthesized from LISA for four stage pipelined ICORE architecture.

There are many other languages to specify processor instruction set architectures, like SLED [35], EXPRESSION [15] etc. Till date, no work has been published on HLS from these languages.

2.3 Other High Level Synthesis Systems

There are other types of high level synthesis systems that take behavioral description of a hardware (programmable or non-programmable) in some description language and generate hardware models. The HLS systems perform several types of optimizations and generate structural hardware netlist according to the objective function. Some of these types of high level synthesis systems are described below.

CMUDA [41], developed at Carnegie-Mellon University takes the description written in ISPS [1] language and generates hardware from it. The System Architect's Workbench [39] is a later extension of the CMUDA HLS system.

IMPACT [23], developed at Princeton University is a high level synthesis system specially designed for minimizing power consumption in control flow intensive circuits.

TRS [18], developed at MIT LCS describes hardware at micro-architecture level. TARC, Term Rewriting Architecture Compiler takes concurrent TRS specifications and generates synthesizable Verilog code.

Bedrock [26], developed at University of Cornell takes input behavioral specifications in a language similar to Pascal and generates FPGA synthesizable hardware model. The input specification language is named HardwarePal.

MAHA [32], developed at University of Southern California is a data path allocation system, which uses the critical path information for hardware synthesis. Several heuristics are developed to get the optimized solutions.

Olympus [5], developed at Stanford University uses HardwareC, a C like hardware specification language for the design specifications. The synthesis system has two toolsets, Hercules and Hebe. Hercules takes HardwareC input and passes result to Hebe, for scheduling and binding.

SPARK [47], under development at University of California, Irvine uses parallelizing compiler techniques to synthesize behavioral ANSI-C functionality specifications to generate synthesizable register-transfer level VHDL code.

CATHEDRAL-III [37], developed at IMEC and ESAT, Belgium is a HLS tool for high throughput DSP applications. The input specifications are written in SILAGE and the system generates both behavioral and structural synthesizable hardware models.

MMAlpha [7], developed at Irisa, France is a HLS tool used for synthesizing hardware for regular architectures like systolic arrays, from Alpha language processor specification. Alpha is a functional language for describing regular algorithms at behavioral level.

AMICAL [22], developed at TIMA Laboratory, France is a VHDL behavioral synthesis system that reads VHDL behavioral specifications and generates VHDL output.

CADDY-II [13], developed at FZI Research Center, University of Karlsruhe Germany is a high level synthesis system that takes behavioral description in VHDL or DSL and generates structural VHDL netlist. It supports different architectural alternatives like multiplexers and buses, single phase or two phase clock etc.

BSS [19], developed at Technical University of Braunschweig, Germany takes behavioral description written in C as input and generates synthesizable Verilog netlist. The tool is a part of COSYMA hardware-software co-design tool.

NESCIO [17] and NEAT [16], developed at Eindhoven University of Technology, Netherlands provides a framework for high level synthesis. NEAT is an object oriented high level synthesis interface and it is used by NESCIO HLS system.

CAMAD [49], developed at Linkoping University, Sweden is a HLS system that takes behavioral specifications written in Pascal like ADDL language, convert them to internal petri net structures and generate VHDL RTL netlist.

Rodin [46], developed at AITEC, Japan takes LSI behavioral specifications as input and generates logical circuits at RT level.

PICO-N system [36], developed at HP Labs automatically synthesizes embedded non-programmable accelerators from the nested loops described in C. The loops, which are the most time consuming part of program execution are converted to synthesizable VHDL RTL level structure. The output is synthesized as co-processor. The underlying architecture of the PICO-N HLS system is VLIW in nature.

DAA [24], developed at AT & T Bell Labs takes an expert system based approach to synthesize data path of general purpose processors. Other high level synthesis systems developed at AT & T Bell Labs are BRIDGE [40], BECOME [43], Cherm [44] and CONES [38].

Phideo [29], acronym for PHIlips viDEO compiler is developed at Philips research center for high-throughput digital applications, specially for video processing. It generates parallel architectures from the behavioral specifications of the digital systems.

CALLAS [3], developed at Siemens, Germany is a behavioral and logic synthesis tool.

Cyber [42], developed at NEC research lab is a high level synthesis tool that takes the specifications written in C as its input.

Chapter 3

Design of the High Level Synthesis System

3.1 Introduction

In this thesis work, the high level synthesis system is developed that generates both behavioral and structural HDL models of processors from *Sim-nML* processor specifications. The outputs are Verilog processor models, in which the behavioral models are suitable for fast functional simulation and the structural models are suitable for both functional simulation and hardware synthesis. The generated Verilog structural descriptions are fully compliant with Synopsys Inc.'s industry standard Design Compiler synthesis tool [2].

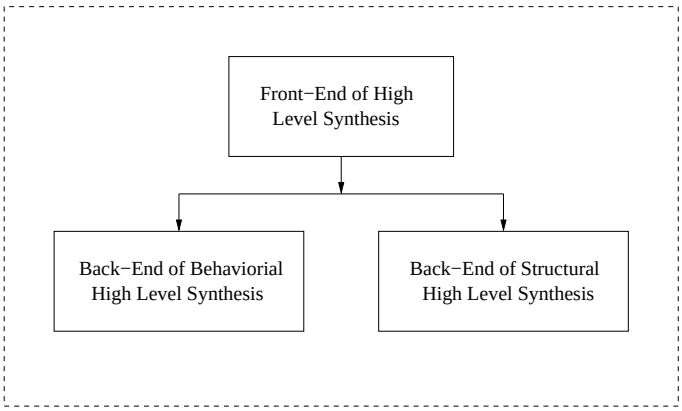


Figure 3.1: Overall Design of High Level Synthesis System

The overall design of the high level synthesis system (figure 3.1) consists of two parts, the front-end and the back-end. The front-end is same in both behavioral and structural high level synthesis systems. The back-end for the structural synthesis is more complex than the back-end for the behavioral synthesis system.

3.2 Design of the Front-end of High Level Synthesis System

The design of the front-end of the high level synthesis system is shown in the figure 3.2. It takes the *Sim-nML* processor specifications as input and produces their

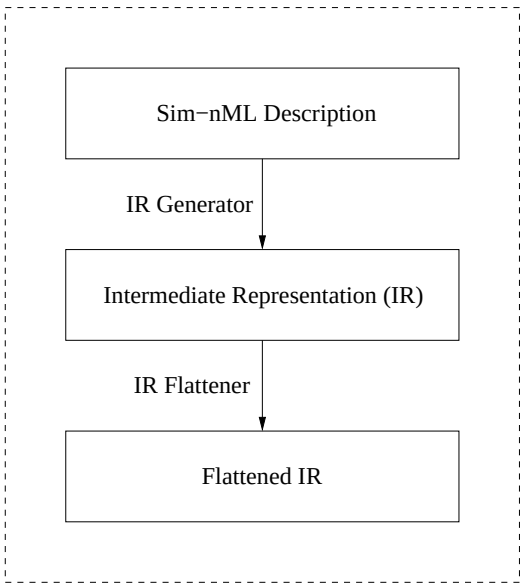


Figure 3.2: Front End of High Level Sysnthesis

flattened representation. In the first step, input *Sim-nML* processor specifications are converted to binary intermediate representations (IR) by an existing tool called *'irg'* [34]. IR is suitable for subsequent analysis such as flattening etc. *Sim-nML* specifies programmable processors in *attribute grammar* form, where the information of each machine level instruction is fragmented over an *attribute grammar* specification tree. The root node of the tree is named *'instruction'*. To get information about a particular instruction of the processor, the path from the root node to the corresponding leaf node is traversed, with proper parameter substitution at all levels. While flattening the internal representation, all such paths from root to the various leaf nodes are traversed and information about all possible machine instructions is obtained.

3.3 Design of the Back-end of Behavioral High Level Synthesis System

The design of the back-end of the behavioral high level synthesis system is shown in the figure 3.3. The back-end for behavioral synthesis system employs no opti-

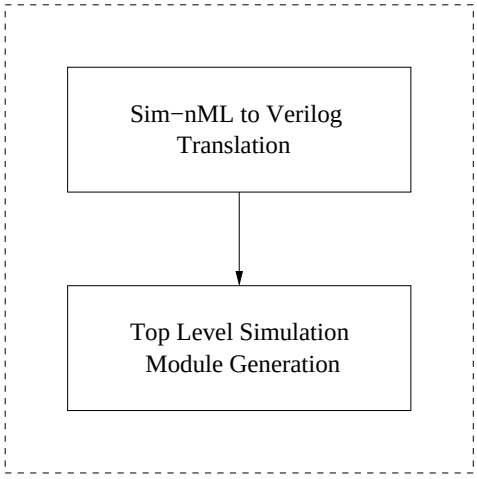


Figure 3.3: Behavioral Design Back-End

mizations for the input specifications. The back-end of behavioral synthesis system takes the flattened IR as input and for each machine instruction action, generates the Verilog behavioral processor model. The Verilog model is obtained as a simple translation from the IR. After generation of processor Verilog functional model, a top level simulation module is generated to facilitate the functional simulation process.

3.4 Design of the Back-end of Structural High Level Synthesis System

3.4.1 Overview of the Back-end Design

The design flow of the back-end of our structural high level synthesis system is shown in the figure 3.4. It has four major steps, optimizations of the flattened intermediate representations; scheduling of the optimized specifications; resource allocation and interconnection of resources; and control path generation. The data path is generated in scheduling and resource allocation steps. In the first step, optimizations are performed to improve the quality of the design (area minimization, speed enhancement

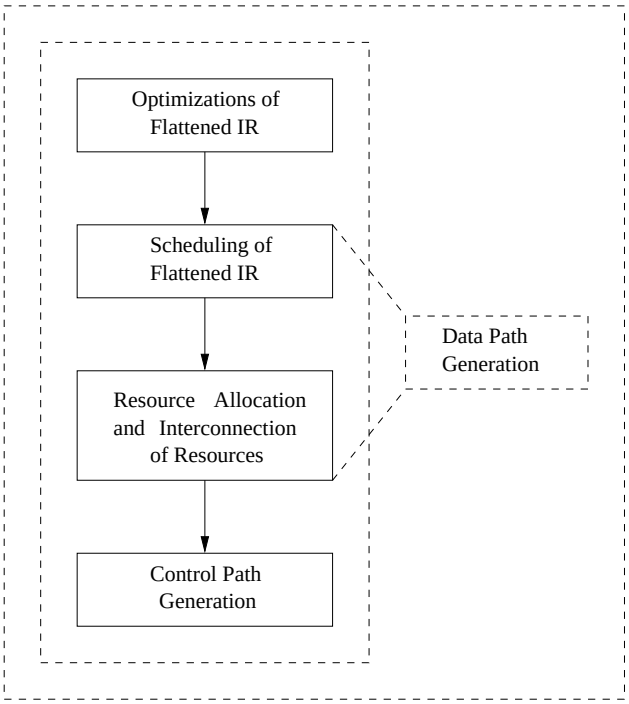


Figure 3.4: Back-End Design Flow

etc.), while maintaining the functionality or the semantic meaning of the *Sim-nML* processor specifications.

In the second step, the operations used in the optimized specifications are scheduled into control steps. Scheduling is performed under several constraints (like the types of resources available, maximum numbers of resources of each types, resource architectures, speed and power consumption of the generated hardware etc.) keeping one or multi-objective scheduling goal. In our implementation, the single objective chosen for optimization is area minimization.

After scheduling, the hardware resources are allocated. This step instantiates the hardware modules according to the scheduling. The scheduling and resource allocation are both architecture specific and are performed with a target architecture in mind. The target architecture for our approach is shown in the figure 3.5. It is a non-pipelined architecture, which takes multiple clock cycles to execute an instruction. The architectural features put extra constraints in the process of scheduling, resource allocation and interconnection of resource. For example, the number of ports on a register file will determine how many arguments can be read for an instruction at the same time. After instantiating the data path elements, interconnection elements including the multiplexers, de-multiplexers and wires are instantiated.

After generation of data path by means of scheduling and resource allocation, the

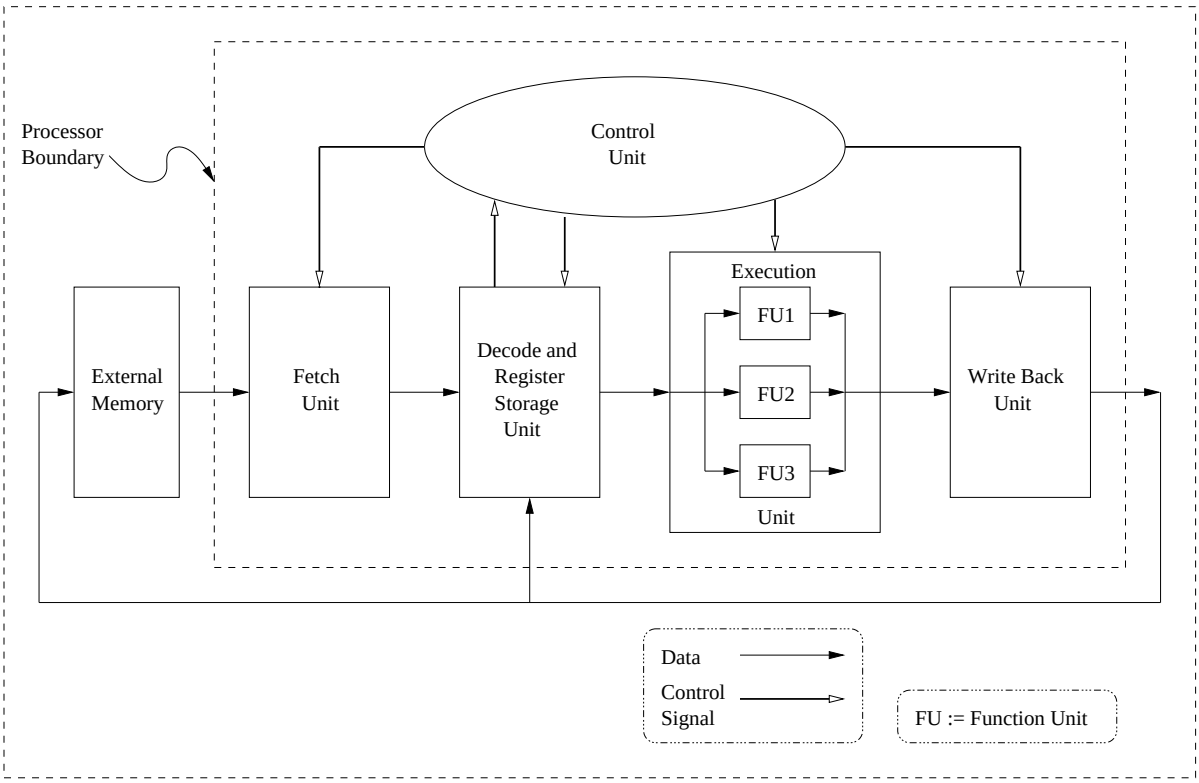


Figure 3.5: Processor Block Diagram

controller is generated. The controller generates control signals for the data path elements as per the decoded instructions and the schedule of the instructions.

The back-end of structural high level synthesis is parameterized. Some of the parameters that are used are the width of the input output ports of the functional and storage units; and corresponding wire widths. The parameters are specified using Verilog *'parameter'* construction.

3.4.2 Processor Architecture of the Structural Design

The synthesized processor model is expected to work with an external memory. The overall system block diagram is shown in the figure 3.6. Processor sends address and read/write control signal to memory and the data is exchanged between memory and processor. The processor contains data path and control path elements. The data path elements execute the instructions under the control of the signals generated by the control path elements. Control signals are generated according to the decoded instructions and the corresponding instruction scheduling (figure 3.6).

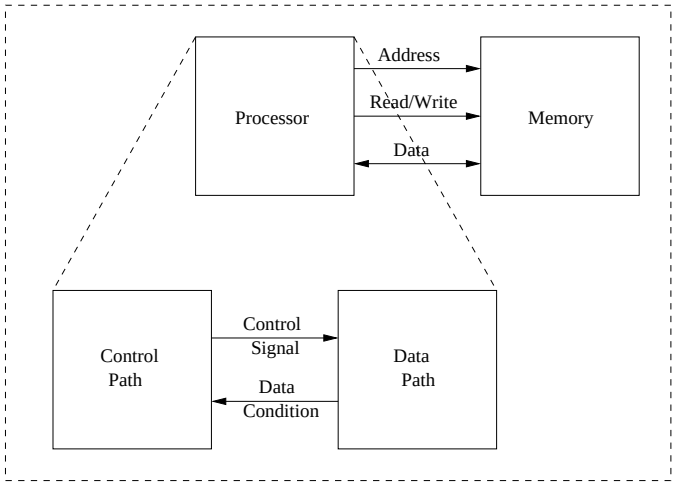


Figure 3.6: System Block Diagram

The different units of the architecture as shown in the figure 3.5 are, fetch unit; decode and register storage unit; execution unit and write-back unit. Fetch unit gets data and instructions from the memory. The write-back unit writes the data back into the memory and registers, after completion of execution in the execution unit. Decode and register storage unit, decodes the instructions and put operands to the execution unit from registers. Execution unit is a collection of several functional units as shown in the figure 3.5.

The target architecture of our design is simpler than pipelined architectures. This architecture is selected to show the feasibility of high level synthesis from *Sim-nML* specifications. Our aim was not to generate the architecture for efficiency but just to show the feasibility of the synthesis from *Sim-nML* specifications. More advanced high level synthesis systems can support one or more complex architectures from where the user can select the architecture of his choice. Further there can be advanced incremental *'design space exploration'* approaches in which the design is transformed automatically to one of the several alternative architectures to meet the desired objectives.

3.4.3 Optimizations of the Flattened Intermediate Representations

The *Sim-nML* specifications for a processor can be written in an unoptimized way, which makes the specification writing easier and elegant looking. It is necessary to optimize the specifications for area minimization and speed maximization. The optimizations on the flattened input are similar to the ones used by the compilers. The

optimizations that are suitable for synthesizing hardware are for example, temporary variable elimination, dead code elimination, common-subexpression elimination, expression simplification, constant propagation and some hardware-specific transformations etc. In our implementation, we perform only the temporary variable elimination and the dead code removal arising because of temporary variable elimination.

- **Temporary Variable Elimination :** Temporary variables used in *Sim-nML* specifications are defined using *'var'* data types. There are other types of variables in *Sim-nML* like *'reg'* and *'mem'* that specify registers and (external or internal) memory in the processor. The *'var'* data types do not correspond to any physical storage units. Systematic optimizations performed around these temporary variables do not destroy the semantics of the instruction specifications. As the two other data types corresponds to physical storage units, removal of any definition of these variables through optimizations can violate the overall processor ISA semantics. The removal of temporary variables eliminates the need of the storage registers. However, some temporary variables can not be removed from the design automatically. These remaining temporary variables are instantiated as temporary registers, keeping the functional consistency of the specifications.

- **Dead Code Elimination :** The dead code, i.e the code that serves no computational purpose is eliminated. Only the dead codes that modify the *'var'* type temporary variables are removed. This is because, for *'reg'* and *'mem'* data types, the seemingly dead code for one instruction can have semantic meaning associated with some other instructions.

3.4.4 Scheduling of the Optimized Instructions

The optimized instructions require a series of operations to be performed. Depending upon the dependency, these operations are scheduled one after another in time. Further depending upon the availability of the resources, some of these operations may be done in parallel.

Scheduling Constraints

- **Architecture of the Processor :** The non-pipelined multi-cycle architecture of the processor permits only one instruction to be executed at a time. After optimization, one instruction of the processor can use one or multiple functional units to execute the instruction. For each instruction, accesses to multiple functional units are performed in non-pipelined manner.

- **Number of Functional Units :** Generally in a processor, one or more than one functional units of the corresponding type are available. However, in our implementation, only one functional unit of any type is instantiated. This constraint is added to minimize area, which is kept as a scheduling goal. Thus if a *Sim-nML* specification contains a maximum of 10 additions over all instructions, only one adder will be instantiated in the design.

- **Number of Data Port Resources :** Similar to the most real designs, the storage units (registers, register files and memory) have one input data port, one output data port and one multiplexed read/write address port. Thus in one clock cycle either one read or one write operation can be performed in the storage units. The functional units in our implementation contain two input data ports and one output data port. In addition, the functional units are encapsulated inside one execution unit. The input and output data ports of the functional units are mapped to the input and output ports of the execution unit respectively. Thus in a clock cycle only the ports corresponding to one functional unit are available.

- **Types of Functional Units :** In our implementation, one functional unit can perform only one type of operation. Thus, no shared functional units like *'adder-subtractor'*, *'multiplier-adder'* etc. are instantiated. This makes one to one correspondence between the operations and functional units in the scheduling.

Scheduling Goal

- **Minimization of Processor Area :** The primary goal for our work in high level structural synthesis is to minimize area. A simple rule to achieve this is to minimize the number of each types of functional unit instantiated. However because of the minimization of functional units there is a need of extra multiplexers or multiplexers with large number of inputs. It may increase the overall size of the processor.

To generate scheduled operation sequences of every instruction, the instruction actions are converted to a sequence of *three address* operations. The *three address* form is like *'A = B + C'*, which reads from the storage units B and C, performs operation addition and writes back in the storage unit A. In our design, this *three address* form takes four clock cycles to complete the execution. Two cycles are needed to read from the storage units, one clock cycle to perform the operation and one clock cycle to write back in the storage unit.

We have shown some scheduling examples in the figure 3.7. The example in the figure 3.7A shows execution of one *three address* operation *R3 = R1 + R2* in four cycles. The reduction of clock cycle requirement to three clock cycles is shown in

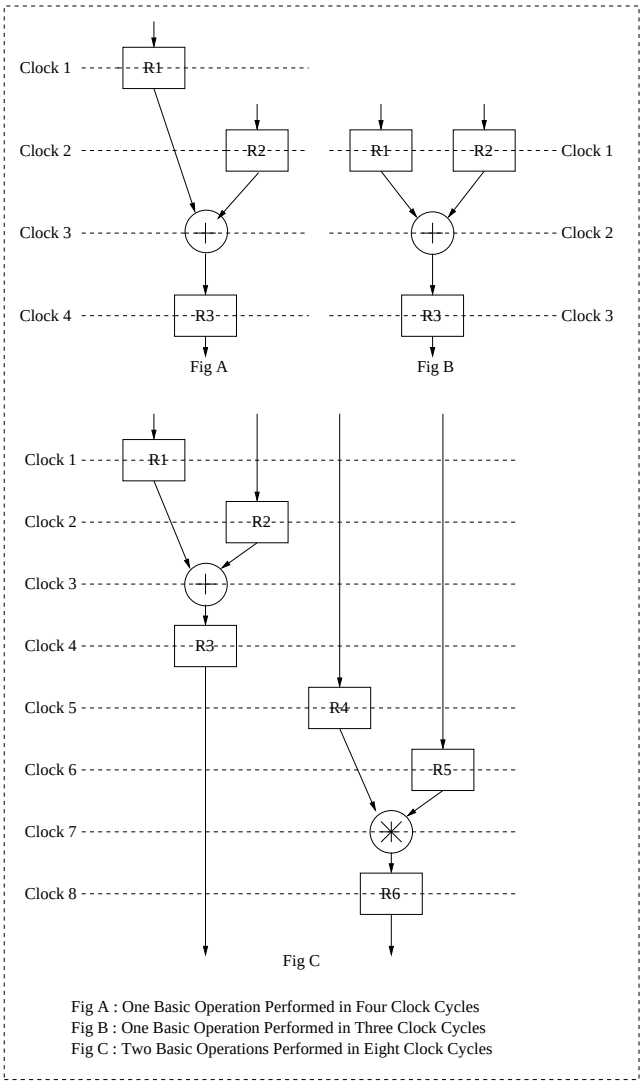


Figure 3.7: Scheduling Example Diagram

the figure 3.7B. If *R1* and *R2* are in a single register file, with only one read port, then scheduling can not be performed in this way. In a more conservative way, we have chosen scheduling of operations similar to one shown in the figure 3.7A. The scheduling of two *three address code* operations are shown in the figure 3.7C. It takes eight cycles to complete.

An example of the scheduling is given below. Here an expression involving four operand is first converted into two *three address* operations. A schedule of eight clock cycles is then drawn as shown in the example.

```
Sim-nML Instruction Action
A = B + C * D

Three Address Code
T = C * D
A = B + T

Scheduled Operations
Clock1        Read C
Clock2        Read D
Clock3        Multiply C, D
Clock4        Write T
Clock5        Read T
Clock6        Read B
Clock7        Add B, T
Clock8        Write A
```

Our scheduling is conservative and does not employ hardware parallelism. Thus there is a lot of scope for improvement in this approach. In this scheduling step, new temporary variables are generated because of translation to the *three address* code.

3.4.5 Resource Allocation and Interconnection Generation

After scheduling, all operations are mapped to functional units and all operands are mapped to storage units in the hardware model. During instantiation of the resources, there can be architectural variations in the instantiated units. For example, an adder can be carry look ahead adder or ripple carry adder based on the scheduling goal. As the functional and storage unit resources are shared across the instructions, these resources have multiple sources and destinations. From amongst these multiple sources, one is selected, depending upon the instruction and the clock cycle. For this multiplexers are used with appropriate controls. Similarly for the multiple destinations, appropriate de-multiplexers are instantiated.

3.4.6 Control Path Generation

After designing the data path, control path elements are instantiated to design the controller. Sequences of control signals are specific to the instructions as per the operations within the instruction and the scheduling of these operations. An instruction decoding unit is needed that decodes the instructions from its binary pattern. The control signals are then generated according to the scheduling of the operations in the corresponding instruction. To generate control signal for a scheduled *'three address'* operation, two read; one operation selection and one write control signals are generated sequentially. Total number of clock cycles needed to execute an instruction is the sum of the clock cycles needed to perform all scheduled *three address* operations.

In our implementation, the control path is not synthesized completely. Some part of the control path design is manually added after the synthesis.

Chapter 4

Implementation of High Level Synthesis System

4.1 Introduction

The high level synthesis system is implemented in C that runs on the Linux platforms. The tool takes *Sim-nML* processor specifications in intermediate format and generates behavioral and/or structural Verilog description of the programmable processor.

4.2 Implementation of Front-end of High Level Synthesis System

As explained earlier we used an intermediate representation of the *Sim-nML* processor specification as starting point of our approach. For this, we have used the intermediate representation generator (*irg*) developed by Rajiv A. R. [34]. It takes the input *Sim-nML* specifications and converts them to internal binary tabular format.

The intermediate representation is hierarchical in nature and is flattened using a tool. The flattening tool is an extension of the earlier work of disassembler generator [21] and functional simulator generator [4]. After flattening, all the mode and op rules are merged and all possible machine instructions (with all possible variations in the addressing modes) are retrieved. At this moment, the internal data structures hold the actions of all possible machine instructions with the expansion of appropriate mode rules.

4.3 Implementation of Back-end of Behavioral Synthesis System

The back-end of the behavioral synthesis system takes the flattened intermediate representations of the specification and translates each instruction action into the corresponding Verilog code. In the behavioral high level synthesis no optimization is performed. All *Sim-nML* variables in the input specifications defined using *'reg'*, *'mem'* and *'var'* data types, are converted to Verilog variables. The scalar variables of *Sim-nML* are translated to Verilog reg data types while the *Sim-nML* arrays are translated to Verilog register arrays. Examples of variables translations are given below.

```
Scalar Variables Translation
Sim-nML :        reg       A[1,card(32)]
Verilog :        reg[0:31] A

Vector Variables Translation
Sim-nML :        reg       B[100,card(32)]
Verilog :        reg[0:31] B[0:99]
```

Decoder for the instructions is implemented as Verilog *'casex'* statements. The image string to be decoded is stored in an intermediate register named *'IR'*. An example translation of a single Motorola 68HC11 microprocessor *Sim-nML* instruction action to behavioral Verilog is given below.

```
Sim-nML Action Sequence
op LDAA_Imm(Src : Imm8)
syntax  = format("ldaa %s", Src.syntax)
image   = format("10000110%s", Src.image)
action  = {
    R = Src;
    CCR<3..3> = R<7..7>;
    if R == 0 then
        CCR<2..2> = 1;
    else
```

```
        CCR<2..2> = 0;
    endif;
    CCR<1..1> = 0;
    A = R;
}
Behavioral Verilog Code
always @(posedge clock) begin
casex (IR[0:31]) 32'b10000110XXXXXXXX :
begin
    R = IR[8:15];
    CCR[3:3] = R[7:7];
    if( (R == 0) ) begin
        CCR[2:2] = 1;
    end
    else begin
        CCR[2:2] = 0;
    end
    CCR[1:1] = 0;
    A = R;
end
```

In the behavioral module, a simulation clock is added. All instructions actions are executed in the single simulation clock, irrespective of the number of basic operations in the instruction action. As shown in the earlier example, if the decode image string length is less than the specified bit width, the Verilog simulator extends the string by padding the zeros to the left. The array selection, control flow statements etc. are similar for both *Sim-nML* and Verilog language.

Sim-nML array selection can take variables as their array selection parameters, but Verilog array selection does not support variables in array selection. Thus, *Sim-nML* specifications with variables used for array selection can not be synthesized using the current behavioral synthesis tool. The following example of *Sim-nML* specification can not be synthesized in our implementation.

```
Part of Sim-nML Specification
reg   A[1,card(32)]
reg   T[1,card(32)]
```

```
reg   X[1,card(16)]
X  =  A<(31-T)..(16-T)>
```

After generation of the behavioral Verilog processor model, the Verilog simulation monitor module is added. The system tasks and functions added in the monitor module are like

```
$monitor($time, " Clk=%b, IR=%h, A=%d, B=%d",Clk,IR,A,B);
$readmemb("rom.mem",rom);
$display($time, " Clk=%b, IR=%h, A=%d, B=%d",Clk,IR,A,B);
$dumpfile("Processor.vcd");
$dumpvars(0,Processor);
$dumpflush;
```

The simulator monitor module continuously probes the various Verilog variables that represent the external signals on the pins or the internal signals (figure 4.1).

The monitor module also generates a VCD (Value Change Dump) file and dumps the information about simulation time, scope, signal definition and signal value changes in that text file [30]. This file is used for the post-processing to observe the signals. We tested our Verilog module using Cadence Verilog-XL simulator [51]. Simulation post processing was performed using tools like Cadence SignalScan [51].

In *Appendix C*, a part of generated Motorola 68HC11 Verilog description is presented. The corresponding top-level monitor module is shown in the *Appendix D*.

4.4 Implementation of Back-end of Structural Synthesis System

4.4.1 Introduction

The back-end of the structural synthesis system generates structural synthesizable Verilog code for the given *Sim-nML* processor specifications. Our implementation generates the Verilog code, which is compliant with the Synopsys Design Compiler. The generated Verilog code is built upon the Design Ware Library [2] components thus saving effort in rebuilding our own library. The DesignWare Library components used in the structural synthesis are *'DW03_reg_s_pl'*, *'DW01_decode'*, *'DW01_mux_any'*, *'DW01_add'* and *'DW01_sub'* etc.

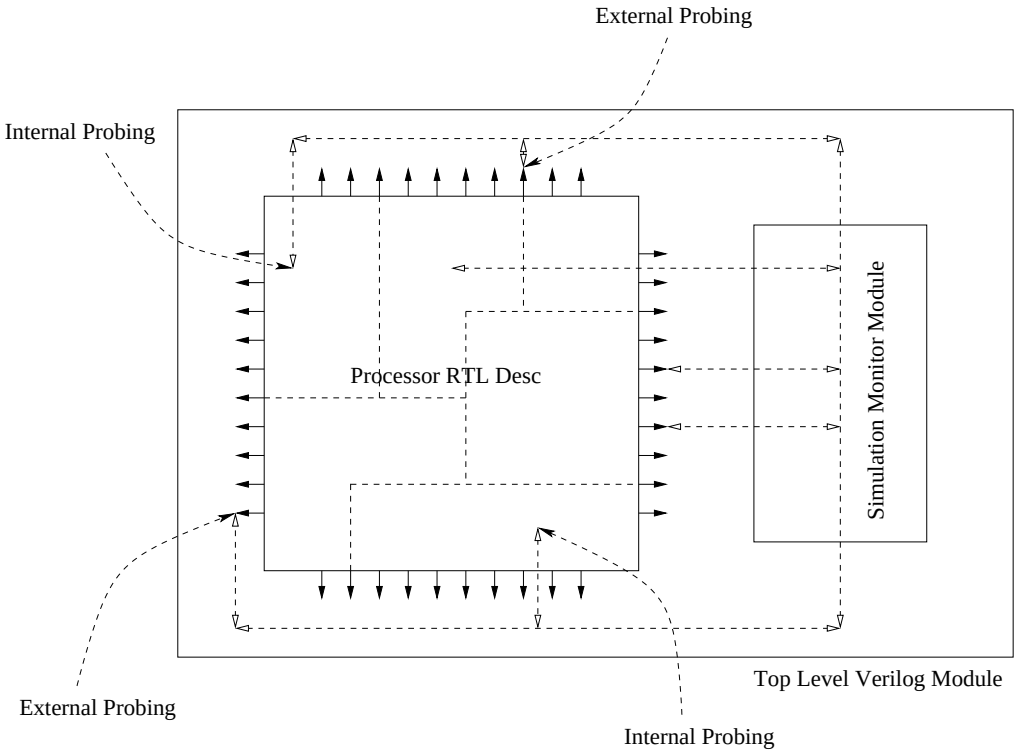


Figure 4.1: Simulation Monitor Module

4.4.2 Optimization of Flattened IR and Three Address Code Generation

On the flattened IR, we perform two optimizations - temporary variable elimination and dead code elimination. The temporary variable elimination algorithm is built over two passes on the flattened input. It removes *Sim-nML 'var'* type temporary variables. However, some temporary variables can not be eliminated automatically as explained later. The two pass algorithm for temporary variable elimination is given below.

```
Sim-nML Temporary Variable Elimination Algorithm
Pass 1 : Find The Basic Blocks for each Instruction Actions

Pass 2 : Removal of Temporary Uses
2.1    :    For each Instruction Action
2.2    :       For each Basic Block find definition of Temporary Variables.
2.3    :       Replace the use of a Temporary Variable with the last
```

```
                corresponding definition of the Temporary Variable.
2.4     :   Remove the definitions of Replaced Temporary Variables.
```

Following the temporary variable removal, we perform the dead code elimination. A code that is not reached, and only updates the temporary variable is removed. In addition to this, the last assignment statements to the temporary variables, which are not used subsequently, are also removed. This is possible, as the temporary variables are not assigned any storage. However, other codes such as assignments to registers etc. are retained as these are treated as side effects of the instructions.

After performing the optimizations, the instruction actions are converted to *'three address'* form. This form is suitable for scheduling as explained earlier. The *'three address'* form is sequence of operations that involve upto three operands, such as *'A = B + C'* or *'A = B'*. The algorithm to convert *'three address'* form is simple and is carried out over two passes on the optimized flattened code.

An example of temporary variable removal and three address code generation is given below.

```
After Temporary Variable Removal and
Before Three Address Code Generation
A = B + C * D

After Three Address Code Generation
(New Temporary Variables Generated)
X = C * D
A = B + X
```

It may be noted that during the *three address* code generation, new temporary variables may be introduced that are not removed. For these newly generated temporary variables storage, registers and register files are instantiated later.

4.4.3 Data Path Element Instantiation

The data path generation essentially comprises of instantiation of the functional and storage units and the interconnection between them. The data path implementation is done in four steps - instantiation of functional units; instantiation of storage units; placement of multiplexers and de-multiplexers and interconnection of components. In the data path generation process, one file is generated for each Verilog module.

In the implementation, it is assumed that all the functional units have two input and one output ports. This establishes a one to one correspondence to the *'three*

address' code and functional unit operations. Functional units input and output ports are connected to the input and output ports of the execution unit.

The *Sim-nML 'reg'* and *'mem'* types of scalar variables are realized using registers. Similarly, the arrays of *Sim-nML* variables are realized using register files. For the temporary variables that are not removed, registers and register files are instantiated. The storage for temporary variables is however shared across the instructions. Thus, if two instructions use two and four temporary variables of same type, then four temporary variable storage units will be instantiated in the design. In that case, the first instruction will use the two out of the four storage units.

An example of the generated Verilog structural register module is given below. Register files are instantiated in similar way, with an extra multiplexed read/write address port.

```
module EA_Reg(Clk,WD, WE, Reset, RD);
parameter            width=32;
parameter            reset_value=0;
input                Clk;
input                Reset;
input                WE;
input   [width-1:0]  WD;
output  [width-1:0]  RD;
DW03_reg_s_pl        #(width , reset_value)  R1( .d(WD), .clk(Clk),
                     .reset_N(Reset), .enable(WE), .q(RD) );
endmodule
```

The Verilog module *'EA_Reg'* instantiates *'R1'* module of *'DW03_reg_s_pl'* type. *'DW03_reg_s_pl'* is DesignWare library module which implements register with synchronous enable reset [52]. The inputs to the *'EA_Reg'* module are *'Clk'*, *'WD'*, *'WE'*, *'Reset'* and *'RD'*. Among these inputs, the *'WD'* and *'RD'* are write data input and read data output respectively. *'Reset'* is the reset control signal for the *'EA_Reg'* module, which is passed to the *'R1'* module. Upon reset, the value stored in the register is set to *'reset_value'*, which is equal to 0 in our case. The *'reset_value'* and *'width'* are constants declared as Verilog parameters. Names of these parameters are predefined in the Design Compiler synthesis tool. The *'width'* parameter defines the widths of the *'d'* and *'q'* ports of the *'DW03_reg_s_pl'* library module. In the implemented *'EA_Reg'* register module, *'WE'* is the control signal for write enable. *'Clk'* signal is added to pass the clock across the module.

An example of the generated Verilog structural execution unit module is given below. In our implementation, the execution unit contains single instantiation of

several necessary functional units each corresponding to the operation used by the instructions.

```
module Execution_Unit(ExIn0_Mux_O__Execution_Unit_In0_I,ExIn1_Mux_O_
  _Execution_Unit_In1_I,Clk,Sel,Execution_Unit_O__Execution_Unit_Dmux_I);
parameter               width=32;
input  [width-1:0]      ExIn0_Mux_O__Execution_Unit_In0_I;
input  [width-1:0]      ExIn1_Mux_O__Execution_Unit_In1_I;
input                   Clk;
input  [sel_width-1:0]  Sel;
output [width-1:0]      Execution_Unit_O__Execution_Unit_Dmux_I;
reg    [width-1:0]      Reg_In0;
reg    [width-1:0]      Reg_In1;
reg    [width-1:0]      Reg_Out;
wire   [width-1:0]      Out_1;
wire   [width-1:0]      Out_2;
always @(posedge Clk) begin
       Reg_In0=ExIn0_Mux_O__Execution_Unit_In0_I;
       Reg_In1=ExIn1_Mux_O__Execution_Unit_In1_I;
end
DW01_add #(width) Add1(.A(Reg_In0),.B(Reg_In1),.CI(),.SUM(Out_1),.CO());
DW01_sub #(width) Sub1(.A(Reg_In0),.B(Reg_In1),.CI(),.DIFF(Out_2),.CO());
case(Sel)
     0 : Reg_Out <= Out_1;
     1 : Reg_Out <= Out_2;
endcase
assign Execution_Unit_O__Execution_Unit_Dmux_I = Reg_Out;
endmodule
```

The above example of execution unit has two functional units - *'Add1'* and *'Sub1'* of *'DW01_add'* and *'DW01_sub'* types respectively. The execution unit contains two input data ports *'ExIn0_Mux_O__Execution_Unit_In0_I'* and *'ExIn1_Mux_O__Execution_Unit_In1_I'* of widths equal to parameter *'width'*. These data ports are connected to the multiplexers at the inputs of execution unit. The output data port is *'Execution_Unit_O__Execution_Unit_Dmux_I'* of width equal to *'width'*. The *'Sel'* control signal selects the output ports of the *'Add1'* or *'Sub1'* functional units. The operation of the execution unit is as follows. At the positive clock cycle, the data

inputs are read to internal registers *'Reg_In0'* and *'Reg_In1'*. The register values are passed to the functional units and the outputs of the functional units are stored in *'Out_1'* and *'Out_2* wires during the clock cycle. The wire values are put into the output internal register *'Reg_Out'* based on the *'Sel'* control signal. At the end of the clock cycle, the value of *'Reg_Out'* is assigned to the output *'Execution_Unit_O_Execution_Unit_Dmux_I'*.

After generating the functional and storage units, for each data (read/write/address) port, the corresponding associated instructions are identified. This gives the information about the necessary multiplexing and de-multiplexing units needed in the design.

For the functional units, the association between instructions, input and output data ports in find out. Accordingly for selecting, the multiplexers and de-multiplexers are generated. At the same time the wire interconnections are identified and instantiated.

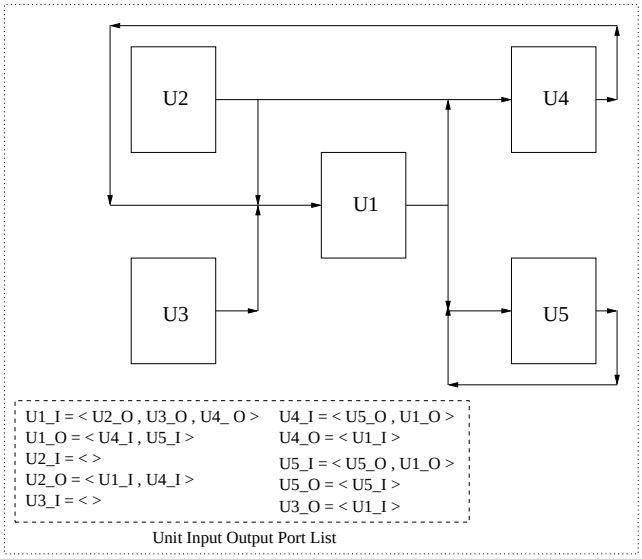


Figure 4.2: Unit Connection before Mux/Dmux

An example of the interconnection is shown in the figure 4.2 and 4.3. **U1** to **U5** in the figure 4.2 and figure 4.3 are functional or storage units. In figure 4.2 the functional unit wiring is shown with the collisions. Foe example, input to **U1** can be from one of the three outputs, namely that of **U2**, **U3** or **U4**. Accordingly the multiplexers are placed as shown in the figure 4.3 and re-wiring is done.

An example of generated data path of a hypothetical small processor with two input registers, one output register and an execution unit is given in *Appendix D*.

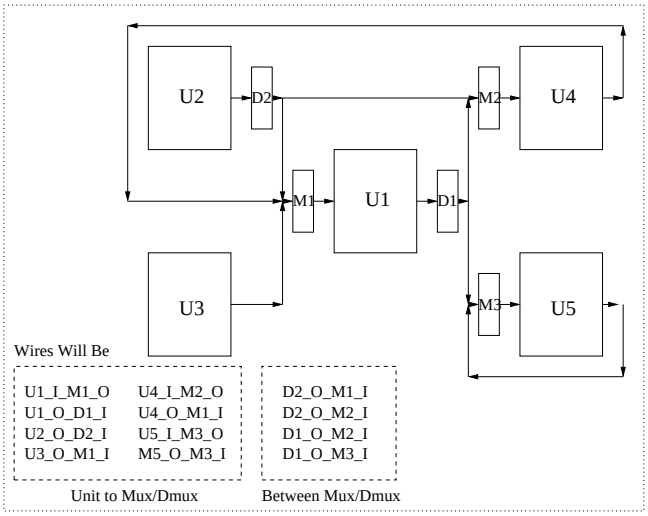


Figure 4.3: Unit Connection after Mux/Dmux

4.4.4 Control Unit Generation

Control unit takes the flattened image stored in the instruction register and according to the bit pattern of the instructions and the schedule of operations within the instruction, generates the control signals for the functional and storage units. Control path generation process is shown in an example below.

```
op  add(A:reg, B:reg, C:reg)
    syntax = format("add %s %s %s", A.syntax, B.syntax, C.syntax)
    image  = format("1011%s10%s11%s00", A.image, B.image, C.image)
    action = {
             A = B + C;
    }

Image String with 0, 1 and Unknown bits

    1011xxxx10xxxx11xxxx00

After decoding, the binary image segment selects the data path add operation.
The unknown bits, after decoding selects the registers for input and output.
```

In our implementation, only a small part of the control circuit is generated, while the remaining is added manually.

4.4.5 Top-Level and Simulation Module Generation

After generating the data paths and control paths, the toplevel module is generated, which instantiates the registers, register files, multiplexers, de-multiplexers and interconnects them according to the generated wires. The toplevel module is wrapped with simulation module, which is used for simulating the structural synthesizable Verilog processor model. The simulation module is similar as the simulation module described in the implementation of behavioral synthesis back-end.

Chapter 5

Results and Conclusion

5.1 Results

The result of the high level synthesis system is tested on the *Sim-nML* specifications of Motorola 68HC11, a subset of PowerPC 603 and another small hypothetical processor.

5.1.1 Result of Behavioral Synthesis System

The behavioral high level system tool is tested for Motorola 68HC11 microprocessor specifications (table 5.1). The size of the generated Verilog code is about the same

Sim-nML Description Lines of Code	2947
Total Number of Machine Instruction	210
Generated Behavioral Verilog Lines of Code	3708

Table 5.1: Behavioral Synthesis Run Statistics for Motorola 68HC11

order as that of the input *Sim-nML* specification. An example of the simulation run using the generated Verilog code is shown in figure 5.1.

The simulation of the Verilog code is performed using Cadence Inc.'s Verilog-XL simulator [51]. In the generated Verilog behavioral code, the machine instructions are executed in one simulation clock cycle, irrespective of the number of basic operations in the instruction. An example of the generated Motorola 68HC11 Verilog behavioral code is given in *Appendix C*.

```
  0 Clk=0, IR=xxxxxxxx, rom[i]=xxxxxxxx, A=  x, M=  x
 50 Clk=1, IR=xxxxxxxx, rom[i]=000086f0, A=  x, M=  x
100 Clk=0, IR=xxxxxxxx, rom[i]=000086f0, A=  x, M=  x
150 Clk=1, IR=000086f0, rom[i]=0000eaf0, A=  0, M=169
200 Clk=0, IR=000086f0, rom[i]=0000eaf0, A=  0, M=169
250 Clk=1, IR=0000eaf0, rom[i]=00008af0, A=  0, M=234
300 Clk=0, IR=0000eaf0, rom[i]=00008af0, A=  0, M=234
350 Clk=1, IR=00008af0, rom[i]=0000aaf0, A=  4, M=138
400 Clk=0, IR=00008af0, rom[i]=0000aaf0, A=  4, M=138
450 Clk=1, IR=0000aaf0, rom[i]=0000baf0, A=  4, M=170
500 Clk=0, IR=0000aaf0, rom[i]=0000baf0, A=  4, M=170
550 Clk=1, IR=0000baf0, rom[i]=0000aaf0, A=  4, M=186
600 Clk=0, IR=0000baf0, rom[i]=0000aaf0, A=  4, M=186
650 Clk=1, IR=0000aaf0, rom[i]=0000aaf0, A=  0, M=169
700 Clk=0, IR=0000aaf0, rom[i]=0000aaf0, A=  0, M=169
750 Clk=1, IR=0000aaf0, rom[i]=0000aef0, A=  0, M=170
800 Clk=0, IR=0000aaf0, rom[i]=0000aef0, A=  0, M=170
850 Clk=1, IR=0000aef0, rom[i]=0000aa70, A=  0, M=174
900 Clk=0, IR=0000aef0, rom[i]=0000aa70, A=  0, M=174
950 Clk=1, IR=0000aa70, rom[i]=xxxxxxxx, A=  0, M=170
```

Figure 5.1: Simulation of Behavioral Verilog Code

5.1.2 Result of Structural Synthesis System

The structural high level synthesis system is tested on a subset of *Sim-nML* PowerPC 603 processor specification and on a small hypothetical processor specification. The subset of PowerPC 603 processor specification includes general ALU instructions, branch instructions and memory load-store instructions. The generated Verilog code consists of several Verilog files, each instantiating the storage, functional and multiplexing/de-multiplexing units. The code that provides the interconnections among all units is kept in a single file. The control signal ports are generated automatically. However, the scheduled control signal sequences are added manually to get the complete Verilog code.

The figure 5.2 shows different levels of synthesis flow. We have also synthesized the generated Verilog code using logic synthesis tools, the Synopsys Design Compiler [52] and Cadence Silicon Ensemble [51]. We used the Design Ware Library for DesignCompiler synthesis.

From the synthesized netlist, the area and power requirements are estimated for the processor. The results are shown for the subset of PowerPC 603 specification

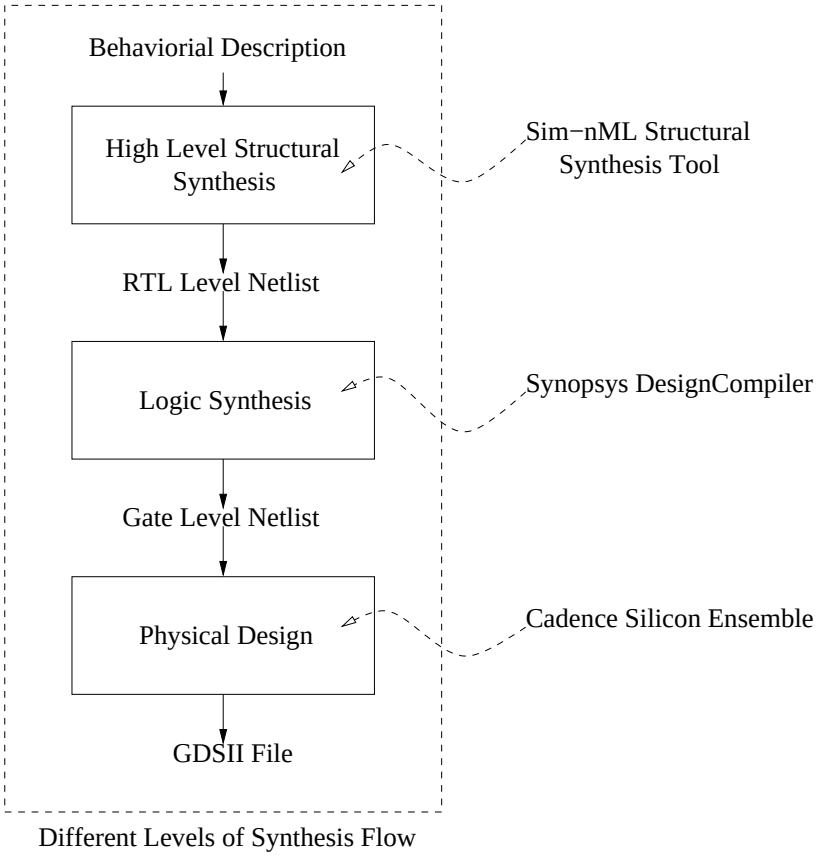


Figure 5.2: Different Levels of Synthesis

in the table 5.2. In the result statistics, there is no major difference in the number of lines of codes between the *Sim-nML* specification and the HLS generated Verilog code.

Number of Lines in Sim-nML Specification	508
Number of Lines in HLS generated Verilog Code	656
Number of Lines in Design Compiler generated Verilog Code	8478
Time for Synthesis (without Clock Tree insertion)	220 Sec
Time for Synthesis (with Clock Tree insertion)	780 Sec

Table 5.2: Structural Synthesis Run Statistics for PowerPC 603 subset

The total cell area for synthesized PowerPC 603 subset is given in the table 5.3 and 5.4. The smallest cell area is taken as of one unit and corresponding unit values

are shown in the tables. The smallest cell area depends on the target technology library based on which the absolute values can be calculated. The total cell area of the processor core includes combinatorial and non-combinatorial areas. After synthesizing

Area	Units
Combinatorial Area	6102.00
Non-combinatorial Area	10878.00
Total Cell Area	16980.00

Table 5.3: Total Cell Area for Synthesized PowerPC 603 subset

the structural Verilog model, clock tree is inserted in the model to generate the clock signals. The total cell area increases by nearly 5% after clock tree insertion.

Area	Units
Combinatorial Area	6684.00
Non-combinatorial Area	10878.00
Total Cell Area	17562.00

Table 5.4: Total Cell Area for Synthesized PowerPC 603 subset after Clock Tree Insertion

The processor core is a collection of several instantiated Verilog modules like registers, execution units etc. The cell area for the instantiated modules and for the gates used is shown in the tables 5.5 and 5.6.

Similar results for a hypothetical processor are shown in the tables 5.7 and 5.8. The processor contains only two input and one output registers and one execution unit. The design does not contain multiplexers and de-multiplexers.

5.2 Conclusion and Future Works

In this thesis, we have developed techniques to generate behavioral and structural synthesizable Verilog processor model from the *Sim-nML* processor specification language. The method is suitable for ASIP and/or other programmable processor generation where the instruction set of the processor is specified in *Sim-nML* language. The simulation and synthesis process of *Sim-nML* high level synthesis generated netlist is compliant with the current industry standard tools.

Cell	Area before Clock Tree insertion	Area after Clock Tree insertion
CIA_Reg	321	321
EA_Dmux	224	224
EA_WD_Mux	87	90
EA_Reg	321	321
ExIn0_Mux	250	260
ExIn1_Mux	469	497
ExecutionUnit_Dmux	224	224
ExecutionUnit	1527	1879
GPR_Dmux	224	224
GPR_RA_Mux	83	84
GPR_WA_Mux	83	84
GPR_WD_Mux	42	42
GPR_RegFile	11511	11702
IR_Dmux	224	224
IR_Reg	321	321
LR_Reg	321	321
NIA_WD_Mux	42	42
NIA_Reg	321	321
Temp0_Reg	321	321

Table 5.5: Total Cell Area by instantiated modules for Synthesized PowerPC 603 subset

The current design can be improved in several ways to support complex architectures. Support for VLIW, SuperScalar architectures, simple and complex pipelined architectures can be added. Overall better semi-automatic design space exploration mechanisms can be incorporated. The full *resource usage model* of *Sim-nML* language can be utilized to generate better quality hardware. At hardware synthesis more number of optimizations can be performed to generate more optimized hardware structure. The scheduling of the processor instructions can be improved in a major way. The total flow from *Sim-nML* to lowest level physical synthesis work can be more explored to get the complete flow of ASIP generation.

Before Clock Tree Insertion			**After Clock Tree Insertion**		
Gate	Count	Area	Gate	Count	Area
AN2	89	2	AN2I	90	2
NR8	32	6	FD1	1554	7
AO2	70	2	IVDA	27	1
FD1	1554	7	OR3	20	2
AO5	20	3	NR3	18	2
ND4	256	2	MUX21H	137	4
IV	121	1	EON1	1216	3
EON1	1222	3	IVI	359	1
AO4	4	2	NR5	4	4
MUX21H	160	4	OR2I	8	2
NR5	4	4	IVDAP	3	2
AN3	32	2	AOIP	6	2
OR3	3	2	EN	8	3
NR3	1	2	ND3	1	2
NR2	93	1	B4IP	3	4
EN	23	3	ND2	32	11
ND3	19	2	NR16	717	1
AO6	6	2	ND2I	106	1
ND2	53	1	NR2I	240	3
MUX31L	22	4	EO	31	3
EO1	8	3	ENI	4	2
EO	38	3	MUX21L	87	3
NR4	3	2	MUX31L	3	4
			MUX21LP	34	4
Total Area		16980	**Total Area**		17562

Table 5.6: Total Cell Area by instantiated gates for Synthesized PowerPC 603 subset

Area	Units
Combinatorial Area	452.00
Non-combinatorial Area	168.00
Total Cell Area	620.00

Table 5.7: Total Cell Area for Hypothetical Processor Data Path

Call	Area
EA_Reg	85
ExecutionUnit	369
IR_Reg	85
NIA_Reg	81
Total Area	620

Table 5.8: Total Cell Area by instantiated cells for Hypothetical Processor Data Path

Bibliography

[1] Barbacci, M. "Instruction Set Processor Specifications (ISPS): The Notion and its Applications". *IEEE Transactions on Computer-Aided Design* (Jan 1981).

[2] Bhatnagar, H. *"Advanced ASIC Chip Synthesis : Using Synopsys Design Compiler and Primetime"*. Kluwer Academic Publishers, 1999.

[3] Biesenack, J. "The Siemens High Level Synthesis System: CALLAS". *Sixth International Workshop on High Level Synthesis* (November 1992).

[4] Chandra, Y. S. "Retargetable Functional Simulator". Master's thesis, June 1999. http://www.cse.iitk.ac.in/research/mtech1997/9711121.html.

[5] DeMicheli, G., Ku, D., Mailhot, F., and Truong, T. "The Olympus Synthesis System for Digital Design". *IEEE Design and Test* (October 1990), 37–53.

[6] Despian, M. A., and Huang, I. J. "Synthesis of Application Specific Instruction Sets". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (June 1995), 663–675.

[7] Doran, K. W., and Oumarou, S. "Regular Array Synthesis Using Alpha". *Rapport de Recherche Irisa, No829* (May 1994). http://www.irisa.fr/cosi/ALPHA/.

[8] Fauth, A., Freericks, M., and Knoll, A. "Generation of Hardware Machine Models from Instruction Set Descriptions". *Proc. IEEE Workshop VLSI Signal Processing, Veldhoven (Netherlands)* (Oct 1993), 242–250. http://www.techfak.uni-bielefeld.de/techfak/ags/ti/forschung/publikationen/vlsi-93.ps .

[9] Fauth, A., Praet, J. V., and Freericks, M. "Describing Instruction Sets Using nML (Extended Version).". *Technical report, Technische University at Berlin and IMEC, Berlin (Germany)/Leuven (Belgium)* (1995). ftp://ftp.imec.be/pub/vsdm/reports/retargetable_code_generation/af-edtc95.ps.gz.

[10] Gajski, D. D., Dutt, N. D., and Wu, A. C.-H. *"High Level Synthesis Introduction to Chip and Systems Design"*. Kluwer Academic Publishers, 1992.

[11] Girczyc, E. F., Bhur, R. J. A., and Knight, J. P. "Application of a Subset of Ada as an Algorithmic Hardware Description Language for Graph Based Hardware Compilation.". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (April 1985).

[12] Gschwind, M. "Instruction Set Selection for ASIP Design.". *Proc. of the Seventh International Workshop on Hardware/Software Co-Design* (May 1999), 7–11.

[13] Gutberlet, P., Muller, J., Kramer, H., and Rosensite, W. "Scheduling Between Basic Blocks in the CADDY Synthesis System". *Proc. of the Europian Design Automotion Conference* (1992). http://www.fzi.de/sim/Caddy/ .

[14] Hadjiyiannis, G., Hanono, S., and Devadas, S. "ISDL: An Instruction Set Description Language for Retargetability". *In Proceedings of the 34th Design Automation Conference* (June 1997), 299–302.

[15] Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., and Nicolau, A. "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability". *Proc. of the Design, Automation and Test in Europe* (1999).

[16] Heijligers, M. "NEAT: an Object Oriented High Level Synthesis Interface". *Proc. IEEE ISCAS, 1994.* (1994). ftp://ftp.ics.ele.tue.nl/pub/papers/hls/ISCAS94.ps.gz.

[17] Hilderink, H. "NESCIO: An Interactive High Level Synthesis Framework". *Proc. of the Workshop on Circuits, Systems and Signal Processing* (March 1994). ftp://ftp.ics.ele.tue.nl/pub/papers/hls/NESCIO-94.ps.gz.

[18] Hoe, J. C., and Arvind. "Hardware Synthesis from Term Rewriting Systems". *Proc. of VLSI'99"* (December 1999). ftp://csg-ftp.lcs.mit.edu/pub/papers/csgmemo/memo-421a.ps.gz .

[19] Holtmann, U. "High-Level Synthesis System BSS". http://www.cs.tu-bs.de/eis/english/research/oldies/e9520BSS.htm.

[20] Hwang, C. T., Lee, J., and Hsu, Y. C. "A Formal Approch to the Scheduling Problem in High Level Synthesis". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (April 1991), 464–475.

[21] Jain, N. C. "Disassembler Using High-Level Processor Models". Master's thesis, January 1999. http://www.cse.iitk.ac.in/research/mtech1997/9711113.html.

[22] Jerraya, A. A., Park, I., and O'Brien, K. "AMICAL: An Interactive High Level Synthesis Environment". *Proc. of European CAD Conference* (Feb 1993). http://tima-cmp.imag.fr/tima/sls/amical/amical.html.

[23] Khouri, S. K., Lakshminarayana, G., and Jha, N. K. "IMPACT: A High-Level Synthesis System for Low Power Control-Flow Intensive Circuits". *Proc. of the 1998 Design Automation and Test in Europe (DATE '98)* (1998).

[24] Kowalski, T. J., and Thomas, D. E. "The VLSI Design Automotion Assistant : Prototype System". *Proc. 20th Design Automotion Conf* (June 1983), 479–489.

[25] Kumari, S. "An Automatic Assembler Generator for Sim-nML Description Language". Master's thesis, March 2000. http://www.cse.iitk.ac.in/research/mtech1998/9811119.html.

[26] Leeser, M., Chapman, R., Aagaard, M., Linderman, M., and Meier, S. "High Level Synthesis and Generating FPGAs with the BEDROC system". *Journal of VLSI Signal Processing* (1993), 191–214.

[27] Marwedel, P. "The MIMOLA Design system : Tools for the Design of Digital Processors". *Proc. of the 21th Design Automotion Conference* (1984), 53–58.

[28] Marwedel, P. "Matching System and Component Behaviour in MIMOLA Synthesis Tools". *Proc. of the European Design Automation Conference (EDAC)* (1990).

[29] Meerbergen, J., Lippens, P., Verhaegh, W., and Werf, A. V. D. "PHIDEO: High Level Synthesis for High Throughput Applications". *Journal of VLSI Signal Processing* (May 1995). http://www.research.philips.com/pressmedia/releases/e14.html.

[30] Palnitkar, S. *"Verilog HDL A Guide to Digital Design and Synthesis"*. Prentice Hall, Upper Saddle River, NJ, 1996.

[31] Pangrle, B. M., and Gajski, D. D. "Design Tools for Intelligent Silicon Compilation". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (Nov 1987), 1098–1112.

[32] Parker, A. C., Mlinar, M., and Pizarro, J. "MAHA: A Program for Data Path Synthesis". *Proc. of 23rd Design Automotion Conference* (June 1986), 461–466.

[33] Pogde, P. "Retargettable Code Generation using Sim-nML Machine Description". Master's thesis, May 2000. http://www.cse.iitk.ac.in/research/mtech1998/9811114.html.

[34] Rajiv, A. R. "Retargetable Profiling Tools and their Application in Cache Simulation and Code Instrumentation". Master's thesis, December 1999. http://www.cse.iitk.ac.in/research/mtech1998/9811117.html.

[35] Raksey, N., and Fernandez. "Specifying Representations of Machine Instructions". *ACM Transaction on Programming Langauges and Systems* (May 1997). http://www.cs.virginia.edu/ nr/pubs/specifying-abstract.html.

[36] Schreiber, R., Aditya, S., and Rau, B. e. "High-Level Synthesis of Nonprogrammable Hardware Accelerators". *HP Labs Technical Reports (HPL-2000-31)* (2000). http://www.hpl.hp.com/techreports/2000/HPL-2000-31.html.

[37] S.Note, W.Geurts, F.Catthoor, and Man, H. "Cathedral-III: Architecture Driven High-Level Synthesis for High Throughput DSP Applications". *Proc. 28th ACM/IEEE Design Automation Conf* (1991), 597–602.

[38] Stroud, C. "CONES: A System for Automated Synthesis of VLSI and Programmable Logic from Bbehavioral Models". *Proc. of IEEE ICCAD, Santa Clara* (Nov 1986).

[39] Thomas, D. E., Dirkes, E. M., Walker, R. A., Rajan, J. V., Nestor, J. A., and Blackburn, R. L. "The System Architect's Workbench". 337–343.

[40] Tseng. "Bridge: A Versatile Behavioral Synthesis System". *Proc. of 25th ACM/IEEE Design Automation Conference* (1988), 415–420.

[41] Tseng, C. J., and Siewiorek, D. P. "Automated Synthesis of Data Paths in Digital Systems". *IEEE Transactions on Computer Aided Design* (July 1986).

[42] Wakabayashi, K. "C-Based High-Level Synthesis System, Cyber-Design Experience". .

[43] Wei, R.-S. "BECOME: Behavior Level Circuit Synthesis Based on Structure Mapping". *Proc. of 25th ACM/IEEE Design Automation Conference* (1988), 409–414.

[44] Woo, N.-S. "A Global, Dynamic Register Allocation and Binding for a Data Path Synthesis System". *Proc. of the 27th Design Automotion Conference* (June 1990), 505–510.

[45] "An Introduction to System-Level Modelling in SystemC 2.0". http://www.systemc.org/papers/SystemC_WP20.pdf.

[46] "High Level Synthesis System: RODIN". *Proc. of Fifth Generation Computer Systems* (1992). http://www.icot.or.jp/ARCHIVE/Museum/IFS/abst/070.html.

[47] "SPARK: Synthesis using Parallelizing Compiler Techniques". http://www.cecs.uci.edu/ spark/index.shtml.

[48] "The Esterel Language Primer, version v5_91". ftp://ftp.esterel.org/esterel/pub/papers/primer.ps.

[49] Zebo, P., and Krzysztof, K. "Automated Transformation of Algorithms into Register-Transfer Level Implementations". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (Feb 1994), 150–166.

[50] Zivojnovic, V., Pees, S., and Meyr, H. "LISA - Machine Description Language and Generic Machine Model for HW/SW Co-Design". *In Proceedings of 1996 IEEE Workshop on VLSI Signal Processing* (1996). http://www.ert.rwth-aachen.de/Projekte/Tools/LISA/lisa.html.

[51] "OpenBook Reference Manual". *Cadence Inc.*.

[52] "DesignWare Library Manual". *Synopsys Inc.*.

Appendix A

Synopsys Design Compiler Configuration Setup

A.1 Design Compiler .synopsys_dc.setup File

Following is the Synopsys Design Compiler (DC) setup file .synopsys_dc.setup which is necessey to access all the DC Basic and Foundation Library components during the synthesis process.

```
company = "IIT Kanpur";
designer = "CARES";
technology = "0.35 micron";
search_path = search_path + { "." ,
                "/space/synopsys/syn_sim99.10/libraries/" };
target_library = {class.db};
synthetic_library = {dw01.sldb dw02.sldb,dw03.sldb,dw04.sldb,dw06.sldb};
link_library = target_library + synthetic_library;
symbol_library = {class.sdb};
```

A.2 Design Compiler Compilation Script

Folowing is the Synopsys DC Compilation Script. The script reads synthetic library 'synthesis.sl' and produces internal '.sldb' file. After that it compiles all sub_modules

and writes low level verilog netlist according to the target_library 'class.db' as defined in '.synopsys_de.setup'. At the next stage clock tree is inserted with period 50Hz in the port 'Clk' and the design is re-compiled to get the total area, power and other reports.

```
read_lib        synthesis.sl
write_lib        synthesis.sldb
sub_modules = {NIA_Reg, EA_Reg, IR_Reg, ExecutionUnit, TopLevel}
foreach(module,sub_modules){
        read -format verilog module + ".v"
        if(dc_shell_status != 1){
                sh echo 'error ' module
                quit;
        }
        compile
        write -format verilog -hierarchy module
}
report_area > area_before_CT
set_wire_load LARGE -mode enclosed
set_operating_condition WORST
create_clock -period 50 -waveform  { 0 25 } Clk
set_clock_skew -delay 2.0 -minus_uncertainty 3.0 Clk
set_input_delay 2.0 -clock Clk -max all_inputs()
set_max_area 0
compile
write -hierarchy -output netlist.db
report_area > area_after_CT
quit
```

A.3 Design Compiler Parameterized Library Specification

Following is the Synopsys DC Synthetic Library 'synthesis.sl' which is used for parameterized implementation of Design Ware Library components. Without this the DC

components like 'DW03_reg_s_pl' can't be used parametrically. This file also specifies the implementation architecture of the modules during synthesis process. If for any module implementation is not specified, DC will choose a particulat implementation from the internal DC database.

```
library("synthesis.sldb") {
        module(DW03_reg_s_pl) {
        design_library : "DW03_reg_s_pl.db";
        parameter(width) {
                hdl_parameter : TRUE;
        }
        parameter(reset_value) {
                hdl_parameter : TRUE;
        }
        implementation(sim){
        }
        pin(d) {
                direction : input;
                bit_width : "width";
        }
        pin(clk) {
                direction : input;
                bit_width : "1";
        }
        pin(reset_N) {
                direction : input;
                bit_width : "1";
        }
        pin(enable) {
                direction : input;
                bit_width : "1";
        }
        pin(q) {
                direction : output;
                bit_width : "width";
        }}
        module(DW01_decode) {
        design_library : "DW01_decode.db";
        parameter(width) {
                hdl_parameter : TRUE;
```

```
}
parameter(dec_width) {
        hdl_parameter : TRUE;
}
pin(A) {
        direction : input;
        bit_width : "width";
}
pin(B) {
        direction : output;
        bit_width : "dec_width";
}}
module(DW01_mux_any) {
design_library : "DW01_mux.db";
parameter(A_width) {
        hdl_parameter : TRUE;
}
parameter(SEL_width)  {
        hdl_parameter : TRUE;
}
parameter(MUX_width) {
        hdl_parameter : TRUE;
}
pin(A) {
        direction : "input";
        bit_width : "A_width";
}
pin(SEL) {
        direction : "input";
        bit_width : "SEL_width";
}
pin(MUX) {
        direction : "output";
        bit_width : "MUX_width";
}}
module(DW01_add) {
design_library : "DW01_add.db";
parameter(width) {
        hdl_parameter : TRUE;
```

```
}
pin(A) {
        direction : "input";
        bit_width : "width";
}
pin(B) {
        direction : "input";
        bit_width : "width";
}
pin(CI) {
        direction : "input";
        bit_width : "1";
}
pin(SUM) {
        direction : "output";
        bit_width : "width";
}
pin(CO) {
        direction : "output";
        bit_width : "1";
}}
module(DW01_sub) {
design_library : "DW01_sub.db";
parameter(width) {
        hdl_parameter : TRUE;
}
pin(A) {
        direction : "input";
        bit_width : "width";
}
pin(B) {
        direction : "input";
        bit_width : "width";
}
pin(CI) {
        direction : "input";
        bit_width : "1";
}
pin(DIFF) {
```

```
        direction : "output";
        bit_width : "width";
}
pin(CO) {
        direction : "output";
        bit_width : "1";
}}
module(DW01_absval) {
design_library : "DW01_absval";
parameter(width) {
        hdl_parameter : TRUE;
}
pin(A){
        direction : "input";
        bit_width : "width";
}
pin(ABSVAL) {
        direction : "output";
        bit_width : "width";
}}
module(DW01_ash) {
design_library : "DW01_ash";
parameter(A_width) {
        hdl_parameter : TRUE;
}
parameter(SH_width) {
        hdl_parameter : TRUE;
}
pin(A) {
        direction : "input";
        bit_width : "A_width";
}
pin(DATA_TC) {
        direction : "input";
        bit_width : "1";
}
pin(SH) {
        direction : "input";
        bit_width : "SH_width";
```

```
}
pin(SH_TC) {
        direction : "input";
        bit_width : "1";
}
pin(B) {
        direction : "output";
        bit_width : "A_width";
}}}
```

Appendix B

Example of Structural Datapath of a Hypothetical Processor

The structural Verilog description of a hypothetical processor is given below. The hypothetical processor contains three registers and one execution unit, which contains four functional units - adder, subtractor, shifter and absolute value calculator.

```
module TopLevel(Clk,EA_Reset,IR_Reset,NIA_Reset,EA_WE,IR_WE,NIA_WE,Ex_Sel);
parameter Width = 8;
input Clk;
input EA_Reset;
input IR_Reset;
input NIA_Reset;
input EA_WE;
input IR_WE;
input NIA_WE;
input [1:0] Ex_Sel;
wire [Width-1 : 0] EA_RD_Out;
wire [Width-1 : 0] IR_RD_Out;
wire [Width-1 : 0] EX_Out;
EA_Reg EA_Reg_inst(.Clk(Clk), .WD(), .WE(EA_WE), .Reset(EA_Reset),
                .RD(EA_RD_Out) );
IR_Reg IR_Reg_inst(.Clk(Clk), .WD(), .WE(IR_WE), .Reset(IR_Reset),
                .RD(IR_RD_Out) );
```

```
ExecutionUnit ExecutionUnit_inst(.A(EA_RD_Out), .B(IR_RD_Out),
                .Clk(Clk), .Sel(Ex_Sel), .C(EX_Out) );
NIA_Reg NIA_Reg_inst(.Clk(Clk), .WD(EX_Out), .WE(NIA_WE),
                .Reset(NIA_Reset), .RD() );
endmodule

module ExecutionUnit(A,B,Clk,Sel,C);
parameter width = 8;
parameter SH_width = 3;
parameter A_width = 8;
parameter B_width = 8;
parameter Sel_width = 2;
parameter C_width = 8;
input [A_width - 1 : 0 ] A;
input [B_width - 1 : 0 ] B;
output [C_width - 1 : 0 ] C;
input Clk;
input [Sel_width - 1 : 0 ] Sel;
reg [C_width - 1 : 0 ] C;
wire [2 : 0] B_Sh;
wire [A_width-1 : 0] Out1;
wire [A_width-1 : 0] Out2;
wire [A_width-1 : 0] Out3;
wire [A_width-1 : 0] Out4;
wire CI_inst;
assign B_Sh = 3'b010;
assign CI_inst = 0;
DW01_add #(width) Add1(.A(A), .B(B), .CI(CI_inst), .SUM(Out1), .CO());
DW01_sub #(width) Sub1(.A(A), .B(B), .CI(CI_inst), .DIFF(Out2), .CO());
DW01_ash #(A_width, SH_width) Shift1(.A(A), .DATA_TC(CI_inst),
                        .SH(B_Sh), .SH_TC(CI_inst), .B(Out3));
DW01_absval #(width) Abs1(.A(A), .ABSVAL(Out4));
always @(Sel or Out1 or Out2 or Out3 or Out4) begin
case(Sel)        // synopsys full_case parallel_case
```

```
        2'b00 : C = Out1;
        2'b01 : C = Out2;
        2'b10 : C = Out3;
        2'b11 : C = Out4;
endcase
end
endmodule

module IR_Reg(Clk,WD, WE, Reset, RD);
parameter width = 8;
parameter reset_value = 5;
input Clk;
input Reset;
input WE;
input [ width-1 : 0 ] WD;
output [ width-1 : 0 ] RD;
reg Enable;
always @ (Clk) begin
        Enable = WE;
        $display($time,"IR_WE = %b, IR_Reset = %b, IR_RD = %b",WE,Reset, RD);
end
DW03_reg_s_pl #(width , reset_value) R1( .d(WD), .clk(Clk),
                .reset_N(Reset), .enable(Enable), .q(RD) );
endmodule

module        EA_Reg(Clk,WD, WE, Reset, RD);
parameter width = 8;
parameter reset_value = 10;
input Clk;
input Reset;
input WE;
input [ width - 1 : 0 ] WD;
output [ width - 1 : 0 ] RD;
reg Enable;
```

```
always @(Clk) begin
        Enable = WE;
        $display($time,"EA_WE = %b, EA_Reset = %b, EA_RD = %b",WE, Reset, RD);
end
DW03_reg_s_pl #(width , reset_value) R1( .d(WD), .clk(Clk),
                .reset_N(Reset), .enable(Enable), .q(RD) );
endmodule

module        NIA_Reg(Clk,WD, WE, Reset, RD);
parameter width = 8;
parameter reset_value = 0;
input Clk;
input Reset;
input WE;
input [ width - 1 : 0 ] WD;
output [ width - 1 : 0 ] RD;
reg Enable;
DW03_reg_s_pl #(width , reset_value) R1( .d(WD), .clk(Clk),
                .reset_N(Reset), .enable(Enable), .q(RD) );
endmodule
```

Appendix C

Section of Generated Verilog Behavioral Synthesis Code

Following is the section of generated Verilog behavioral synthesis code of Motorola MC68HC11 microprocessor. The lines of code in Sim-nML specification is 2947. The flattened description contains 210 machine instructions. The action section of the machine instructions are translated to generate behavioral Verilog code of the corresponding microcontroller. Total lines of generated Verilog code is 3708.

```
module     Processor(clock);
reg [0:31] IR;
reg [0:7]  M[0:1000];
reg [0:15] D;
reg [0:7]  A;
reg [0:7]  B;
reg [0:7]  CCR;
reg [0:15] IX;
reg [0:15] IY;
reg [0:31] SP;
reg [0:31] PC;
reg [0:31] NPC;
reg [0:7]  TmpSrc;
reg [0:7]  R;
reg [0:31] LR;
reg [0:0]  TmpBit;
input      clock;
always @(posedge clock) begin
```

```
casex (IR[0:31]) //synthesis parallel case
        32'h10000110XXXXXXXX   :
begin
      R = IR[8:15];
   CCR[3:3] = R[7:7];
   if( (R == 0) ) begin
      CCR[2:2] = 1;
   end
 else begin
      CCR[2:2] = 0;
   end
   CCR[1:1] = 0;
   A = R;
end
////////////// End of Instruction 0 //////////////
        32'h10010110XXXXXXXX   :
begin
      R = M[IR[8:23]];
   CCR[3:3] = R[7:7];
   if( (R == 0) ) begin
      CCR[2:2] = 1;
   end
 else begin
      CCR[2:2] = 0;
   end
   CCR[1:1] = 0;
   A = R;
end
////////////// End of Instruction 1 //////////////
        32'h10110110XXXXXXXXXXXXXXXX   :
begin
      R = M[IR[8:15]];
   CCR[3:3] = R[7:7];
   if( (R == 0) ) begin
      CCR[2:2] = 1;
   end
 else begin
      CCR[2:2] = 0;
   end
```

```
   CCR[1:1] = 0;
   A = R;
end
////////////// End of Instruction 2 //////////////
        32'h11111100XXXXXXXXXXXXXXXX   :
begin
      LR[0:7] = M[IR[8:15]];
   LR[8:15] = M[(IR[8:15] + 1)];
   CCR[3:3] = LR[15:15];
   if( (R == 0) ) begin
      CCR[2:2] = 1;
   end
 else begin
      CCR[2:2] = 0;
   end
   CCR[1:1] = 0;
   D = LR;
end
////////////// End of Instruction 12 //////////////
        32'h10111001XXXXXXXXXXXXXXXX   :
begin
      R = ((A + M[IR[8:15]]) + CCR[0:0]);
   CCR[5:5] = (((A[3:3] & TmpSrc[3:3]) | (TmpSrc[3:3] &
                        R[3:3])) | (R[3:3] & A[3:3]));
   CCR[3:3] = R[7:7];
   if( (R == 0) ) begin
      CCR[2:2] = 1;
   end
 else begin
      CCR[2:2] = 0;
   end
   CCR[1:1] = ((A[7:7] & TmpSrc[7:7]) &  (!(R[7:7] |
                (!(A[7:7] &  (!(TmpSrc[7:7] & R[7:7])))))));
   CCR[0:0] = ((A[7:7] & TmpSrc[7:7]) | (TmpSrc[7:7]
                &  (!(R[7:7] |  (!(R[7:7] & A[7:7]))))));
   A = R;
end
////////////// End of Instruction 45 //////////////
        32'h00100011XXXXXXXX   :
```

```
begin
      if( ((CCR[0:0] + CCR[2:2]) == 1) ) begin
      NPC = ((PC + IR[8:15]) + 2);
   end
 else begin
      NPC = (PC + 2);
   end
end
////////////// End of Instruction 194 //////////////
        32'h00101101XXXXXXXX   :
begin
      if( (CCR[1:1] ^ (CCR[3:3] == 0)) ) begin
      NPC = ((PC + IR[8:15]) + 2);
   end
 else begin
      NPC = (PC + 2);
   end
end
////////////// End of Instruction 195 //////////////
endcase
end
endmodule
```

Appendix D

Simulation Top Level Monitor File Sample

A sample code of simulation top level module which probes the module input/output pins and/or the internal reg/wire of the processor module. Simulation data is stored in 'Processor.vcd' file. The post suimulation data stored in 'Processor.vcd' file can be analyzed using post simulation data analysis tool as Cadence SignalScan etc.

```
module        Processor ;
reg         Clk;
reg         EA_Reset;
reg         IR_Reset;
reg         NIA_Reset;
reg         EA_WE;
reg         IR_WE;
reg         NIA_WE;
reg         [1:0] Ex_Sel;
reg         [6:0] rom[20:0];
integer         i;
initial
// Monitors several external and internal registers and wires.
          $monitor($time, " Clk = %b, EA_WE = %b, IR_WE = %b,
         NIA_WE = %b, Ex_Sel = %b C = %d", Clk, EA_WE, IR_WE,
         NIA_WE, Ex_Sel[1:0], TopLevel_inst.ExecutionUnit_inst.C);
TopLevel TopLevel_inst(.Clk(Clk), .EA_Reset(EA_Reset),
```

```
        .IR_Reset(IR_Reset),  .NIA_Reset(NIA_Reset),
        .EA_WE(EA_WE), .IR_WE(IR_WE), .NIA_WE(NIA_WE), .Ex_Sel(Ex_Sel) );
initial                 //Clock Generation Module
begin
        Clk = 1'b0;
        forever         #50 Clk = ~Clk;
end
initial
begin
        $readmemb("rom.mem",rom);
        i = 0;
end
always @(posedge Clk) begin
        assign {EA_Reset, IR_Reset, EA_WE, IR_WE, NIA_WE, Ex_Sel} = rom[i];
        $display($time, "EA_Reset = %b, IR_Reset = %b,
                EA_WE = %b, IR_WE = %b, NIA_WE = %b, Ex_Sel = %b",
                EA_Reset, IR_Reset, EA_WE, IR_WE, NIA_WE,Ex_Sel);
        i = i + 1;
end
initial
begin
        $dumpfile("Processor.vcd");
        $dumpvars(0,Processor);
        #550;
        $dumpflush;
end
initial
begin
        #550        $finish;
end
endmodule
```