ABSTRACT

JANMEJAY, SATYANKAR. Preliminary Investigation of High Level Synthesis of a C++ Superscalar Processor Model. (Under the direction of Dr. Eric Rotenberg.)

Microprocessors are at the heart of most computing systems today. Their design involves a number of steps, two of which are coding a C++ simulator for its performance modelling and writing a Register Transfer Level (RTL) description. The RTL description is a hardware model of the design that goes through the steps of netlisting and physical design into a chip. RTL description is primarily done in a Hardware Description Language (HDL), the most popular of which is verilog. Of late, with the update of verilog standards in Verilog-2005 and System Verilog, a number of programming-language style features have been included in verilog. In this work, we create a source-to-source compiler that generates verilog code for the register renaming class of an in-house C++ superscalar processor. We verify the functionality of the generated code by writing a relevant test-bench.

Preliminary Investigation of High Level Synthesis of a C++ Superscalar Processor Model

by
Satyankar Janmejay

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina2015

APPROVED BY:

Dr. James Tuck	Dr. W. Rhett Davis

Dr. Eric Rotenberg
Chair of Advisory Committee

DEDICATION

To my parents.

BIOGRAPHY

Satyankar Janmejay was born in Spetember 1990 in the small town of Dumka, India. He received his Bachelor's in Technology from the Indian Institute of Technology , Banaras Hindu University (IIT-BHU), Varanasi, India in May 2011. During his undergraduate study, he worked as an intern at PMC-Sierra, Bangalore in the design verification group during the summer of 2010. After his graduation, he worked at Atrenta India Pvt. Ltd. an Electronic Design Automation company (now a part of Synopsys) in Natinal Capital Region, India. While at Atrenta, between July 2011 and July 2013, he worked in the VHDL/Verilog analyzers group and briefly in the Clock Domains Crossing group.

Currently, Satyankar is a graduate student in Computer Engineering at North Carolina State University(NCSU), under the guidance of Dr. Eric Rotenberg. His research interests lie in the area of Computer Architecure, VLSI design and Compilers. He is also affiliated with the Center for Efficient, Scalable and Reliable Computing(CESR) at NCSU.

ACKNOWLEDGEMENTS

I would like to thank my parents for providing me with the best resources at their disposal throughout my life and my brother for always motivating me. I am also thankful to my professors and peers at IIT-BHU for providing me a firm grounding in Engineering and helping me trace my career path. I would also like to thank Mr. K.C. Jha, a member of our extended family, who encouraged me to pursue a Masters degree.

Working with Dr. Eric Rotenberg has been one of the best opportunities I have ever had. It has truly been a privelege. He has always been supportive and a rich source of engineering wisdom, especially in dire straits.

I also thank my lab partners Anil Kannepalli, Vinesh Srinivasan and Arunesh Goswami for constant motivation and discussion at work. The list is incomplete without a mention of all my friends in Raleigh who have made this journey enjoyable.

This thesis is supported by NSF grant CCF-1218608. Any opinions, findings and conclusions or recommendations expressed herein are those of the author and do not necessarily reflect the views of the NSF.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

The design of a microprocessor is a complex process that involves a lot of engineers working in various teams in tandem. Moreover, this field is contiuously changing with a new technology available every few years due to the rapid improvement in the quality of Design Automation Tools. In this thesis, we strive to add to this catalogue, a tool to automatically convert a timing-exact performance model of the microprocessor into a synthesizable hardware model.

Following are the steps involved in a microprocessor design:

1. *Specification* - The chief figure of merit for a processor is the number of instructions executed per second, while meeting other constraints such as area and power[JLH12]. Over the last few decades, we have been achieving continuous improvements in this dimension. The reason for this being microarchitectural novelties like number of pipeline stages, in-order vs out-of-order,

superscalar, quality of branch prediction (aside from the improving technology nodes). The outlining of these microarchitectural details is the processor's specification.

2. *Writing a simulator* - A software simulator is written in a high level language, typically C/C++ because these, being relatively lower level, result in faster simulation as opposed to JAVA or Python. The simulator is typically a timing-accurate simulator, which means that it maintains all the states the actual hardware would at every clock cycle, the states here being not just the architecturally visible elements such as the register file, but also those such as pipeline registers, Rename Map Tables and everything else that would translate to a sequential circuit element if the processor were to be obtained from a manufacturing facility.

3. *RTL design and verification* - Once a theoretical model for the processor has been established, the design team moves on to coding an RTL description. The language used is typically one among Verilog and VHDL, although the former seems to have gained a lot more hold of late. The RTL is verified using OVM/UVM methodologies in a verification environment like System Verilog or Vera, with verilog taking a good hold here too.

4. *Physical design and verification* - This is the backend portion of the design cycle involving partitioning, floor-planning, placement, clock tree synthesis, signal routing and timing closure.

This thesis explores the relationship between 2 and 3 while attempting to question whether the latter is even necessary in contemporary times.

1.1 Register Transfer Level

The most common hardware description language in use today is *Verilog*. One of the differences between a hardware description language like verilog and a programming language like C/C++, at least syntactically, is the presence of sequential variables which only get assigned to at clock edges. Such variables respresent the state of the processor. Generally, programming in verilog is

a completely different paradigm because of the immense parallelism that hardware design offers. In general, programming a piece of hardware without due consideration to these nuances would result in a sub-optimal design.

That being said, modern Design Automation tools in tandem with modern hardware description laguages like Verilog-2005 and System Verilog offer the designer the ability to abstract away some details, in effect providing the ability to write HDL (Hardware Description Language) code that more closely resembles a procedural language. This is more true for the combinational logic part of the design where complex structures like loops and bypasses can be synthesized. To appreciate these advances, one should consider the fact that even increment/decrement operators and break statements have found their way into system-verilog and what more, those constructs also get synthesized by a synthesis tool like Design Compiler.

However, the optimality of a more rigorous, traditional hardware design is perhaps still unscalable, but what should not be lost is that we are on a trajectory of increasingly advanced tools that need to be fully leveraged.

1.2 Architecture Simulators

The class of simulators we focus on here are *cycle accurate timing simulators*. Such simulators maintain the state of a processor at each clock cycle. The pipeline stages of a microprocessor are necessarily well defined and maintained at each cycle. The cycle is simulated as an iteration in a while loop, while the state is represented by variables that could exist in between clock cycles. That would mean all variables except local variables : class members, global variables and static variables.

As a guideline, the simulators are designed with all the structures mimicking actual hardware. Therefore, it is unusual to find dynamic memory allocations(except for during initialization) and therefore, the use of pointers is usually restricted to arrays. Timing simulators also have another unique feature. The pipeline stages are called in reverse order. So, for a typical 5-stage pipeline, the

write-back stage would be called first and the fetch stage would be called last.

All the other features of programming languages are used with liberty.

1.3 Inferences

The fact that cycle exact timing simulators exactly represent the hardware structures means their equivalent verilog structures can be directly inferred. This means that pipeline registers and other memory elements are necessarily the eqivalents of non-local variables and can be inferred as such.

The building blocks of Verilog are modules, which get instantiated in other higher level modules. It is hard to miss the similarity with functions in C/C++. However, one aspect of difference is that while module instantiations in verilog are in parallel, those in C/C++ are sequential and their execution subject to the program control-flow. It presents a need to not only predicate the module instantiations, but also enforce covert sequentiality, something we will explore in later chapters. For now, we just note that all class members, globals and statics are interpreted as sequential variables and functions are interpreted as modules.

The other option would be to inline all functions into the while loop that calls the pipeline stages in reverse order. While this would offer some advantages in terms of automatic bypass creation, the software to create such a translation would perhaps be too complex. The former method (which we have used), results in a per-function analysis of the simulator code, although it involves a slightly complex manner of generating bypasses and sequentialization.

1.4 Prior Work

High Level Synthesis tools have been around for a long time[GGly], the popular ones among these today being Catapult-C, Vivado and Bluespe[Arvrc]. While these tools do convert a given C code to verilog, their objective is to create pipeline stages on their own following an analysis of the funtional

code that they attempt to churn. This is a different domain from microarchitecture simulators, where the pipeline stages need to be inferred as opposed to being created. However, we do intend to learn from the capabilities of the available HLS tools.

The other variety of tools available are of the likes of Chisel[Bacne], which generate C++ models and verilog models for a given piece of Chisel code(which is a hardware description code). Since most performance models are written in C/C++, Chisel is not useful in RTL generation for them.

CHAPTER

2

SPECIFICATION

In Chapter 1, we looked at how processor simulators are a special kind of procedural code and identified some features. In Chapter 1, we also mentioned that we will be following a per-function based analysis model. Some of these functions are called by other functions. We therefore expect quite a few instances in the generated RTL. With that in perspective, in this chapter, we will go into further detail on how several processor-simulator features can be converted to an eqivalent verilog.

2.1 Local Variables

In Chapter 1, we also mentioned that we will be following a per-function based analysis model. Some of these functions are called by other functions. We therefore expect quite a few instances in the generated RTL. We also mentioned in Chapter 1 that we make a distinction between local and

other variables.

Local variables exist within the scope of the function and get destroyed when the function returns. In a similar vein, *regs* and *wires* declared in modules, and not assigned to in an *always @ (posedge clock)* block, represent the output of various combintional elements. Therefore, all local variables are converted to *regs*. The reason they are not *wire* is that assignments in *always @ (*)* blocks cannot occur to *wires*.

2.2 Class members

The other kinds of variables are static variables, global variables and class members. In *721sim*(which is an in-house superscalar architecture simulator), there are no statics and globals. So, our focal point is class members, but the same analyses apply to the others. Unlike local variables, the class members do not exist in the functions as such, but can be accessed there nonetheless. We have two options on how to handle them:

1. Declare them as *regs* in the module and assign to them in *always @ (clock)* blocks. The variables will then be assigned to at the edge of clock and will retain their values in between the cycles.

2. Have them passed in as input ports to the module and return their values as output ports, in case they are being written to in the module.

Which option we select depends upon how the class member is being used across the program as we will find out in the next section.

2.2.1 Ports

All member functions of the class are expected to access a subset of all the class members. While passing in all the members to every member function module would work, it would result in a number of unnecessary ports being passed around. In addition, consider the case where all state is

represented by global variables. All the generated modules would have all the variables passed in as input ports.

The alternative, which we have used, is to analyze all the functions in a preliminary pass to identify what member variables are being read from and written to. The member variables that are simply being read are input ports, while the member variables that are being written to(or more generally both read from and written to) are declared as output ports in addition to being input ports. To avoid ambiguity, the output port is renamed.

The renamed output port is passed up to the instantiated module, making its value look like the output of a combinational logic block. However, the member function is really a sequential element(representing what would be a flip-flop in hardware). The sequential element would need to reside in exactly one module where it will be assigned to in an *always @ (clock)* block and initialized on reset. We need to find out which of the many modules it is. We arrive at this result using Closest Common Parent(CCP) analysis.

2.2.2 CCP Analysis

The module where the class member resides has the following properties:

1. All references to the class member occur in this function or functions that are called through this function. In terms of verilog, the modules that access this sequential element are instantiated by this module or by modules that are instatiated by this module and so on or this module itself.

2. No module that calls it accesses the class member

Determining such a module inevitably requires us to analyze the program call graph. It is essential that there are no recursive functions. Such a program structure would be unsynthesizable to start with as it would involve a combinational loop of sorts, besides complicating our CCP analysis.

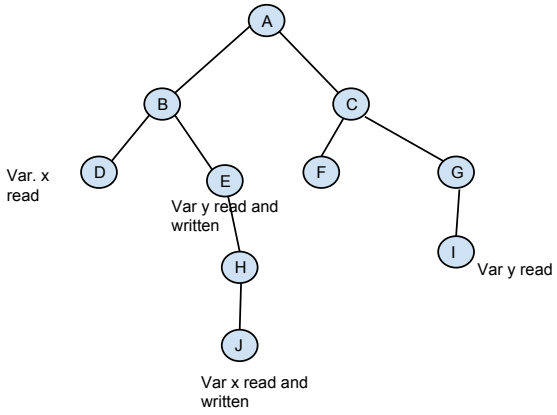

Figure 2.1 CCP analysis graph

Effectively, the call graph becomes a tree. The modules are nodes on the tree. There exists a node in the tree that is parent to all these nodes. Such a node is the Closest Common Parent since it is the node closest to all the nodes while also being a parent to them. Note that this analysis is done with respect to read variables and not read/write variables. If it was done with respect to write variables, the variable would reside in an instantiated module and its value would be required to be propogated up(as output port) and possibly also to other modules down a different path on the call graph. While it is certainly possible to create such a design, it appears unclean and cumbersome.

The CCP analysis is performed for every class member. All the modules on the call graph from the CCP to its final reader/writer also accept the member as input. They also pass it back as output to their instantiating module if the concerned module was a writer. In 2.1 the read CCP for *x* is the function B. It resides in module B and is an input port for modules D, E, H and J. It is an output port for E, H and J. Variable *y*'s read CCP is A. It is an input port for B, E, C, G and I and an output port for B and E.

CCP analysis is, however, not without a few complications:

1. If a function is called more than once in its caller, and that function is determined to be the CCP, then the CCP is actually the caller. Ideally, the two calls should represent two different nodes on the call graph as they are really two instances. With our ROSE compiler framework, that is not the case and we need to treat this case specially. This is also true for a function called inside a loop as it also corresponds to multiple instances. If we do not enforce this, we will end up having multiple, unsynchronized sequential elements and wrong results.

2. If the function is called from two different objects, it effectively means that there are multiple sequential variables, each corresponding its object. While it does not affect the instantiated modules in any way, if the member variable was being passed down from an instantiator and/or it was being passed up to an instantiator, we now create as many copies of the variable as the objects and append then to our list of input/output ports and also those of our parents in the call graph all the way up to the CCP.

2.2.3 Return Values

Functions in C can return values to their caller if they are not *void* functions. However, verilog modules do not return any values, except through ports. Therefore, for functions that return a value, an additional *reg* port called **outValue** is added to the list of output ports. The type of this port is derived from the function's return type.

2.2.4 Pass by reference

C allows variables to be passed by reference. For a C programmer, this means that even though the variable has been passed to the function, if its value gets modified during the call, it will be reflected in the caller. A similar functionality needs to be enforced in verilog. We therefore create an output port for every variable passed by reference.

2.2.5 Pointers being written to

The idea of pass by reference is not very different from that of a pointer in terms of what changes at the end of the function call. It would not be incorrect to create an output port for every argument passed as a pointer. However, the programmer might intend to pass a pointer simply because it is the easiest thing to do without much overhead of copying. In those cases, creating output ports would be a waste. Therefore, we analyze the contents of the function for any writes to variables passed in as pointers. If there are any writes to this variable, we clasify them as output ports and add them to the list of output ports for the module.

2.2.6 Other Ports

Clock, reset and enable also need to be passed down to all the modules. In our work, all sequential assignments happen at the positive edge of the clock. The sequential variables in the module are intialized when reset is high. Enable bit controls whether the combinational logic block will be executed. This will be discussed in detail later.

2.2.7 RAMs

A number of member variables are large data structures like Rename Map Table, Architecture Map Table, Load Queue, Store Queue etc. In actual hardware, these structures are typically RAMs/CAMs. Our model so far is designed to pass entire structures down to instantiated modules. Passing these structures down the heirarchy would imply passing a gamut of wires everywhere, when most of the contents of those wires would remain unaccessed.

The solution lies in noting that these structures are indexed, possibly more than once in the clock period, while being read or written. The modules attempting to read into these structures need to pass the indices all the way up to the top level module where the RAM/CAM is instantiated

as output ports. The values get read at the top level and are passed down to the reader as input ports. For writes, both the index and value are to be passed up to the top level module through output ports. The bypassing is done by the RAM/CAM module itself.

2.3 Program flow

Since most C constructs have their corresponding verilog constructs, control flow need not be re-ordered. In fact, in this work, they are exactly the same as the C code (except for function calls which we will discuss later). The control flow happens in an *always @(*)* block, of which there is one in every module. All the assignments in this block are blocking. The use of blocking assignments enforces sequetiality between the statements, just like a C program. Non-blocking assignments happen in parallel and therefore are not suitable for procedural code.

Therefore, the process of translation simply involves reading every line of the function and sticking it into an always @ (*) block. Most translations occur in this manner, with a few tweaks that we will soon discuss. The result of such a translation is that the function body becomes a blob of combinational logic. Some variables do, however, need to be handled a little differently:

1. *Sequential variables in the module* - Sequential variables, if they do exist in the module, need to be assigned to in an *always @(clock)* block. The value that the variable will be assigned will have to be computed in the preceding *always @(*)* block. We now have two instances of this variable. Since they are both assigned to in *always* blocks, they will have to be *regs* of the same type. Also, one of them will need to be renamed. In this work, every successive renaming of a variable is suffixed with *_renamed_<i>*, where *i* is the number of times the variable has been renamed. We rename the variable that will be used in the *always @ (*)* block and allow the sequential variable to have the original name.

2. *Inout ports* - We refer to the input ports that are modified and passed back to the instantiator

here as inout ports, although we do not use the inout ports provided by verilog. These are mostly member variables that are being passed down the call graph (and then back up) starting from the CCP. As an obvious first step, we need to rename the output variable, since both the input and output port corresponding to the class member have the same names. The output port is renamed with the prefix *out* before the variable name. Therefore, *RMT* will be renamed as *outRMT*.

All references to the variable in the function are those without the appended *out*. These happen to be references to the input port. Since they are input ports, verilog does not allow them to be assigned to in any manner. Therefore, in the event that thy are being assigned to, there would be syntactical errors.

However, the output ports can be assigned to within an *always @ (*)*, block if they are declared as regs and with *assign* statements if they are declared as wires. Therefore, we replace all references of the member variable with the corresponding *out* appended counterpart, regardless of whether the variable is actually being written to. This offers a cleaner and easier to implement alternative to allowing the input port refernce to be used until the first write to the member, and using the output reference thereafter.

3. *Initialization* - Both the renamed sequential variables and the output ports need to be initialized before the procedural assignments begin. If this guideline is not followed, the variables could end up being unknowns in case the control flow does not reach the point of assignment. Moreover, the instantiating module would more likely want the unmodified value of the member variable it passed down in the event it was not tampered with as opposed to wanting unknowns.

As an example of the above mentioned specifications,

Listing 2.1 Example function showing the translation

```
void renamer::myFunction(bool &condition) {
    if((p>2) && A[i][0]){
      condition = true;
      copy(B, C);
      A[i] = p;
      p = 2;
   }
}
```

The generated verilog is:

Listing 2.2 Generated Verilog for the example function

```
module renamer_myFunction(condition, p, i, B, C, n, outp,
                outB, outcondition, enable, clock, reset);
   input [31:0] condition;
   input [31:0] p;
   input [31:0] i;
   input [63:0] [31:0] B;
   input [63:0] [31:0] C;
   input [31:0] n;
   input enable;
   input clock;
   input reset;
   output reg  [31:0] outp;
   output reg [63:0] [31:0] outB;
   output reg [31:0] outcondition;
   reg [63:0] [31:0] A;
```

```
reg [63:0] [31:0] A_renamed_1;
reg enable0;
reg [63:0] [31:0] renamer_copy_arg_0;
reg [63:0] [31:0] renamer_copy_arg_1;
reg [63:0] [31:0] renamer_copy_arg_3;


renamer_copy (renamer_copy_arg_0, renamer_copy_arg_1,
                n, renamer_copy_arg_3, enable0, clock, reset);


always @ (*)
begin
   outp = p;
   outB = B;
   outcondition = condition;
   enable0 = 1'b0;
   A_renamed_1 = A;
   if((outp > 2) && A[i][0])
   begin
      outcondition = 1'b1;
      enable0 = 1'b1;
      renamer_copy_arg_0 = outB;
      renamer_copy_arg_1 = C;
      outB = renamer_copy_arg_3;
      A_renamed_1[i] = outp;
```

```
        outp = 2;
    end
end

always @ (posedge clock)
begin
    if(reset)
    begin
        for(int j = 0; j < 32; j++)
        begin
            A[j] <= 0;
        end
    end
    else
    begin
        A <= A_renamed_1;
    end

end
endmodule
```

Note how the output variables have all been renamed in the *always @ (*)* block of the generated verilog. Also note their initialization before the *if(enable)* statement.

2.4 If-else

If-else statements in verilog are exactly the same as they are in C. The only difference is in the replacement of braces as defining a basic block with *begin* and *end* statements. The test condition and the body remain the same. The only point of disharmony occurs when the if statement in C was only a single statement and was hence not enclosed in braces as a basic block, but that did not happen in verilog owing to some extra statement/s that needed to be added. In this work, we identify the cases where such a digression happens and promptly insert *begin* and *end* markers where appropriate.

2.5 Loops

Both *for* and *while* loops are synthesizable in Verilog-2005 and System Verilog. While their consistent use might not be the finest exponent of circuit design, synthesis tools do a reasonably good job of generating RTL with them. The only restriction is that the number of iterations of the loop should somehow be inferred, if not specified explicitly. For example, the code snippet below gets synthesized on account of the number of iterations being identified as *n_branches*.

```
while((i < n_branches) &&(b > 0))
begin
    if ((  GBM &(1 << i)) == 0)
        b--;

    i++;
end
```

This loop is used in the function *renamer::stall_branch* and was successfully synthesized by Design Compiler. Also worth noting is the fact that most programmers use the post-increment/decrement

and pre-increment/decrement operators while writing the update expression in a for loop. Latest System Verilog standards support these operators and therefore they can be reproduced from the C code into the generated RTL verbatim. As was the case for *if* statements, the only modification necessary is creation of *begin* and *end* blocks, if necessary.

We are not providing support for loops implemented with *goto* statements since they are generally considered poor programming practice. Not surprisingly, they do not have System Verilog equialents.

2.6 Breaks

Break statements are a feature in system-verilog, much like they exist in C. They do not need to be implemented as such.

2.7 Returns

Since modules do not explicitly return any values, return statements do not exist in verilog. They need to be specially handled. Two things need to be done when we see a return statement:

1. Stop all further processing in the function
2. If the module returns a value, set the output port *outValue*

To accomplish 2, the return-value is assigned to *outValue* within the *always @(*)* block in a typical blocking assignment at the point of return statement in the C code. Therefore, the statement

```
return x;
```

will be replaced by

```
outValue = x;
```

1 is a little tricky. We need to make sure that once the control flow reaches the return statment, no other statement is executed (in a behavioral verilog model). Some of the statements will not be executed by virtue of being exclusive to the return statement in the control flow. For example, the else clause of if statements where the return statement was in the then clause will be exclusive to the return statement's execution. Every other statment should be predicated on whether or not the return was executed. Statements in sequential order are lumped into a single block and the block itself is predicated on a return bit. All return statements also correspond to a *return_<i>* bit getting set, where *i* is the number of returns in the function, that predicates the block.

Follwing is a translation of the function *renamer::find_free_bit* to the verilog module *renamer_-find_free_bit* which uses returns that cover both 1 and 2.

Listing 2.3 C-code for find_free_bit

```
unsigned int renamer::find_free_bit() {
    unsigned int i;
    for (i = 0; i < n_branches; i++) {
        if (BIT_IS_ZERO(GBM, i))
            return(i);
    }
    return(NO_FREE_BIT);
}
```

Listing 2.4 Generated Verilog for find_free_bit

```
module renamer_find_free_bit (n_branches, GBM, outValue, enable, clock);
input [31:0]  n_branches;
input [63:0]  GBM;
input enable;
input clock;
```

```
reg return0;
reg return1;
output reg [31:0]  outValue;
reg [31:0]  i;

always @(*)
begin
    return0 = 1'b0;
    return1 = 1'b0;
    if(enable)
    begin
        for(i = 0;i <  n_branches;i++)
        begin
        if ((  GBM &(1 << i)) == 0)
            begin
            return0 = 1'b1;
            outValue = i;
            break;
            end

        end
        if (!return0 & 1)
            begin
            return1 = 1'b1;
            outValue = 3735928559;
```

```
            end
        end
    end
```

The macros BIT_IS_ZERO and NO_FREE_BIT in the given C code have been expanded in the generated verilog.

2.7.1 Returns in loops

When returns occur in a loop, the loop needs to be broken out of first and then the remaining code needs to be predicated. We use break statements to do so. So, if a return is found to occur in a loop, the return is replaced with two statements, a break and setting a return bit. The remaining code in the control flow is then predicated on this return bit.

2.8 Function Calls

Function calls present one of the key implementation challenges in a C to verilog translation. Since all functions are interpreted as modules, they will need to be instantiated. The problem with an instantiation is that it breaks the sequentiality of the otherwise procedural code, as it is parallel to the *always @ (*)* block. Therefore, all arguments will be supplied to the instance, regardless of whether or not the function was actually executed in the control flow.

2.8.1 Renaming

To enforce sequentiality despite the concommitant parallelism, we use renaming. If we did not employ renaming and *x* was to be passed to two different instances, both of which happen to execute in the control flow, then the last instance in the control flow will use a stale value of x, when it should actually use the value obtained as output from the first instance. The solution is to rename the

output from such an instance and use the new name in all further references to that variable.

One can imagine that the scheme mentioned above leads to the problem of static single assignment. Two instances that were originally functions along two separate control flow paths rename the same variable. But at the point where the control flows meet, we would need to select one of those values based on a predicate. This is a fairly complicated scheme, very similar to the classic static single assignment. Instead, we use a prologue and epilogue approach.

We do not rename the variable itself. Instead, we rename both the input and output ports into the instance. At the point of the function call in the control flow, we assign the variable into the input port and the output port to the variable. Therefore, the variable name will always reflect the true value at every point.

2.8.2 Predication

At the point of the function call in the *always @ (*)* block, we set the predicate, which we call *enable_<i>* where *i* is the number of instances predicated, at the point of the function call. Thus, the instances only get enabled when control flow reaches the point of function call. If we were to not implement predication for instances, and the instances had sequential variables which get assigned to on every clock edge, then those sequential variables would get updated with some compuation from the *always @ (*)* block in the instance. That would be incorrect behaviour.

In 2.2, the following lines demonstrate 2.8.1 and 2.8.2 in the *always @ (*)* block.

```
enable0 = 1'b1;
renamer_copy_arg_0 = outB;
renamer_copy_arg_1 = C;
outB = renamer_copy_arg_3;
```

The instance is the following:

```
renamer_copy(renamer_copy_arg_0, renamer_copy_arg_1, n,
```

```
renamer_copy_arg_3, enable0, clock, reset);
```

for the C function call:

```
copy(B, C);
```

2.8.3 Looped instaces

A special case occurs when an function is called within a for loop. In such a case, multiple instances of the function exist, as many as there are iterations in the loop.

We employ the verilog construct *generate for* to handle such instances. A complication arising here is that if a member variable is an output port from the instance, its updated value must be fed into the next instance of the module within the for loop. Furthermore, if the member variable is also an input to some other instance within the same for-generate block, while also being used in some procedural code, we need to be careful about using a stale value somewhere.

Our renaming strategy mentioned in 2.8.1 works perfectly here too, the only difference being that the newly created regs(for input arguments) and wires(for output arguments) are arrays and the input and output ports to the instances are an index into those arrays. Going by the example in 2.8.1, if the variable x is being passed through a function in a for-loop (which iterates 4 times), we create the *reg x_in[3:0]* and the *wire x_out[3:0]*. The arguments into the instance are *x_in[i]* for the input port and *x_out[i]* for the output port. Within the procedural *always @ (*)* block, at the point of the function call, we make the assignments $x\_in[i] = x$ and $x = x\_out[i]$.

Between successive iterations of the for loop, the value of x keeps getting updated to the latest value and that value gets used for the next iteration, which in turn means that the corresponding for-generate instance also gets the most recent value of x.

2.9 Payload

Processors require an instruction to be passed through pipeline stages. In hardware, this translates to a set of bits being copied from one pipeline register to the next at the edge of a clock. If something similar was to be done in a C++ simulator, it would involve copying a large structure from one pipeline stage to another across all stages. To have that happen on every iteration of the outer *while(1)* loop would slow down the simulator. In hardware, since all copy happens in parallel, this is not a problem.

As a standard practice, the information for every instruction in the window, that would otherwise be passed across stages, is kept in a sufficiently wide **payload buffer**. Every fetched instruction goes into the buffer and has an **index** associated with it. Instead of the entire instruction, the **index** gets passed through the pipeline stages.

When generating hardware, this nuance needs to be identified and replaced with the appropriate hardware model. The point to note here is that each pipeline stage in the simulator does have an **index**. We simply replace the **index** with **PAY**, which is a structure with all the payload information. In the verilog, we identify the point where the index copy happens and replace it with a payload copy.

2.10 High Level Overview

The result of the cross-compilation is a verilog model of the C/C++ simulator, where every function is an independent module and gets instantiated in other modules. Each of the modules has an enable bit that is an indicator of whether or not that function was called in a particular clock cycle. On every clock cycle, the top level module enables its instances which in turn conditionally enable their instances and so on. Therefore, there is a propogation of the enable bits (with the other input ports) down the heirarchy and output values from each of the instances up the heirarchy, which

may result in significant propogation delays.

CHAPTER

3

IMPLEMENTATION

The objective of the thesis is to take as input a given set of input C/C++ files and generate verilog files as output with the specifications mentioned in 2. The input file/s need to be read, parsed, interpreted and as a final step, the output file/s need to be created. In this chapter we discuss the details of how each of those steps are executed.

Instead of starting from scratch with a flex-bison parser to parse the C code, we pick a front-end compilation tool called ROSE that does the parsing for us and we use its interface to analyze the C code. This chapter contains details of the components from ROSE that we put to use. We also describe the data structures used during our passes and how we manage them so that the final write to the output file is seamless.

3.1 ROSE Infrastructure

A compiler front-end creates an intermediate representation for the source code it parses. Those intermediate representations can be tree based or linear or a mixture of both. The ROSE intermediate representation is an Abstract Syntax Tree (AST)[Lab].

ROSE compiler also has a number of other features for source-code analysis including those such as pre and post-order traversal of its AST, inherited and synthesized attributes, data-flow and control flow analysis, call-graph analysis and many more. In this work so far, we only need the AST and call-graph analysis tools.

3.1.1 AST structure

The ROSE AST is populated on a per-project basis. An *SgProject* node is the result of compiling a set of input files provided in the command line. This node is also the parent node of entire parse tree. The top level structures in each of the files are the children of the parent. Therefore, if the input files contain declarations for classes and functions, the respective *SgClassDefinition* and *SgFunctionDefinition* nodes are the children of *SgProject* node.

The parsing and AST creation is done by ROSE when the SgProject constructor is called.

```
SgProject* project = new SgProject(argc, argv);
```

where *argv* has all the input files to be parsed.

Further down the tree, every C construct is also represented as an AST node. For example, an *if* statement is represented in the AST by an *SgIfStmt*. *SgIfStmt* has three children, representing its condition, true and false(if it exists) bodies.

The source code can, in principle, be analyzed by performing a traversal of the AST. Every node in the tree (an *SgNode*) has the functions, *get_numberOfTraversalSuccessors()* and *get_traversalSuccessorByIndex()* which can be used to recursively navigate the AST. There is an easier

alternative to doing so, which we cover next. However, a recognition of the AST structure is handy in solving a lot of problems, especially those dealing with expressions.

3.1.2 ROSE object model

In addition to the AST, ROSE also populates each of the nodes in the AST. For example an *SgIfStmt*, in addition to being a node in the AST, is also an object of the *SgIfStmt* class. The members of the class are *p_conditional*, *p_true_body* and *p_false_body* and it also has the member functions *get_conditional()*, *get_true_body()* and *get_false_body()* which we use to access those members. Such member functions exist for all nodes and are more convenient to use than tree parsing.

We would like to mention a few of the important underlying node types that we frequently use in our analysis:

1. **SgNode** - This is the base class from which all other nodes are derived. Every node in the AST is therefore an *SgNode*.

2. **SgInitializedName** - Every variable in the program is associated with an *SgInitializedName*. It is created at the point of the variable declaration. All variables, including members can be uniquely identified with their *SgInitializedName*. We use the variable's *SgInitializedName* in its CCP analysis.

3. **SgVarRefExp** - This node is created for a variable reference. Its corresponding *SgInitialized-Name* can be found by calling the member functions *get_symbol()->get_declaration()*.

4. **SgPntrArrRefExp** - This node is created for array indexing or pointer dereferencing.

5. **SgStatement** - This is a base class for all statements such as *SgIfStmt*, *SgAssignStmt*, *SgClass-Declaration*, *SgFunctionDeclaration* and many other C statements.

6. **SgType** - The type of a variable or function is stored as an *SgType*. The type can be obtained by calling the member function *get_type()*.

3.1.3 Sage Interface

ROSE provides an interface for making API calls of convenience functions. These functions operate on the underlying AST and return the required values to the user. Some of the important functions that we have used are mentioned here.

1. **querySubtree** - This function takes two arguments, an *SgNode* and an *SgType*. It returns a vector of all the nodes in the subtree with the *SgNode* as parent which are of *SgType* type. It does a traversal of the subtree and checks if the child nodes are of *SgType* type.

2. **getEnclosingNode** - This function also takes as input an *SgNode* and an *SgType*. It returns the closest parent of the *SgNode* that is of type *SgType*.

There are a number of other utility functions the Sage Interface provides. A complete list can be found in *sageInterface.h*.

3.1.4 Call Graph Analysis

ROSE provides the **CallGraphBuilder** class for call-graph analysis. The class takes as input the *SgProject* node and populates the class *SgIncidenceDirectedGraph* where each of the nodes is an *SgGraphNode* corresponding to an *SgFunctionDeclaration*. A function could have multiple declarations depending on how many times it has been *externed*. In such a case, ROSE selects the first non-defining declaration. For a member function, that we mostly poupulate the Call Graph with, this is its declaration in the class definition.

SgIncidenceDirectedGraph is a directed graph derived from the class *SgGraph*. We use the methods for getting predecessors (*getPredecessors()*) and successors (*getSuccessors()*) for our purposes. The *SgIncidenceDirectedGraph* can be obtained from a *CallGraphBuilder* object using the method *getGraphNodesMapping()*.

An examination of the include file *CxxGrammar.h* in the ROSE build directory is a good reference for further understanding of the call graph. It is also worth noting that ROSE was unable to generate a complete call graph unless all of the input files were clubbed together into a single file. ROSE provides an API *generateDotGraph()* to print out an *SgIncidenceDirectedGraph* in DOT format, which can be converted to a visual graph format and read as pdf file using *dot* command from the command-line.

3.2 Verilog Object Model

During our analysis pass over the Rose Object Model, we populate an object model(OM) of verilog that we decompile in our final step. In essence, this means that at the end of our analysis pass, we have an object for every module that we created that contains other objects for each of the statements in the *always@(*)* block. We will describe the OM in further detail next, but the verilog OM is in principle not very different from the OM ROSE creates for C code except that it is geared towards verilog.

The reason we do not decompile on the fly as we analyze the C code is that we discover the need for new registers and wires as we go through the simulator code. As we found out in the case of instance argument population in 2.8.1, we also feel the need to print some statements before the current statement being analyzed. If we were to print the verilog code on the fly, we would need to take the file pointer back by a few lines (the number of which we would have to calculate), print the code and take the file pointer back. Such a cumbersome tactic can be entirely circumvented with an OM population-decompile model.

We next describe various elements of our Verilog OM.

3.2.1 VeModule

This is the class representing a module. The analysis of a function results in the creation of this object. So, this is the top level object for a module. It has the following members:

1. _name : The name of the module. Typically, this is the name of the function. For member functions, the class name gets appaended to the function name, with a separating underscore.
2. _inPorts : A vector of the input ports to the module.
3. _outPorts : A vector of the output ports of the module.
4. _seqStatements : Sequential statements that are within the *always @ (posedge clock* block.
5. _combStatements : Combinational statements that are within the *always @ (*)* block.
6. _regs : The reg variables in the module.
7. _instances : Instances in the module.
8. _forgens : For-generate loops in the module.
9. _assigns : Assign statments in the module. (So far none have been necessary in this work)
10. _nReturnBits : Return statements result in the creation of predicate bits(2.7). This variable keeps track of how many of those have been created for the module so far in the analysis. This is useful in creating a new predicate bit(the current value of the _nReturnBits variable) as well as determining how many have been created in the entire module, which is useful in declaration and initialization of those bits.
11. _returnType : This is an *SgType* representing the return type of the function that this module was translated from.

12. _nodeReturnsMap : As mentioned in 2.7, statements need to be predicated on return bits. This map keeps track of what return bits are conditioned on every *SgNode*.

13. _callEnableBits : In 2.8.2, we saw the need for enable bits in an instantiation. This variable keeps track of these predicate bits like 10.

14. _seqVars : The list of sequential variables, essentially the members obtained from 2.2.2.

3.2.2 VePort

This is the class representing a port to a module. Its member is an *SgInitializedName* corresponding to the member variable.

3.2.3 VeStatement

This is a superclass for other verilog statements. It has the following members:

1. _printStatementTerminator : Some statements, like an if-conditional do not need to be terminated with a semicolon. By default, this boolean is set to *true*. If, however, it is set to *false*, the statement is printed without a terminator.

2. _statementsBefore : Some statements need to be preceded by other statements. Case in point instances whose input arguments need to be renamed (2.8.1)

3. _statementsAfter : Some statements need to be followed by other statements, like instances(2.8.1).

The following classes are derived from *VeStatement* class:

1. VeBasicBlock : It containes a *vector* of *VeStatement* and represents the set of verilog statements between *begin* and *end*. Each of those statements could be of any kind.

2. VeAssignStatement : It represents a blocking verilog assignment statement. Its members are an *SgAssignOp* - the ROSE AST node for the C assignment and *_functionData*.

   _functionData is a data structure that holds the location of a function call within an expression. Since a function call finally has to be moved out of the procedural assignment block as an instance, it needs to be replaced with the function's output port that holds the return value. In general, this data structure becomes relevant wherever an expression is present since a function call can occur in any expression.

3. VeReturnAssignStatement : A return statement that returns a value needs an assignment into and output port from an expression. Additionally, the return bit also needs to be set. Since two statements could potentially be printed, this is treated as a separate class from *VeAssignStatement*. Its members are a string for the return expression (*_returnExpr*), a string for the assignment to the predicate return bit(*_str*) as discussed in 2.7, a boolean *_break* to manage the case where the return happens in a for-loop (2.7.1) and *_functionData* described 2.

4. VeExprStatement : C allows the use of expressions as independent statements. The most common occurance is that of function calls that typically do not return a value. But others, such as increment and decrement operators could also occur. These are to be converted to verilog verbatim as *VeExprStatement*. The class contains a member *_str* that is a direct conversion to string of the C expression. Since it is an expression, the class also has the member *_functionData* (2).

5. VeInitializationStatement : These statements are, in principle, not very different from *VeAssign-Statement*, except that they represent C code created from an assigned variable declaration. Since C allows multiple variable declarations in the same statement, this class has a vector of strings, each representing the assignment to a different variable.

6. VeIfStatement : This statement represents a verilog if statement. It has the following members:
   (a) _condition : The condition(*VeStatement*) determining the direction of the if statement
   (b) _trueBody : A *VeStatement* for the condition being true. In the case that there are multiple statements, the *VeStatement* would be a *VeBasicBlock*.
   (c) _falseBody : A *VeStatement* for the condition being false. In the case that there are multiple statements, the *VeStatement* would be a *VeBasicBlock*.
   (d) _falseBodyExists : In case the false body does not exist, this bit is set to false;
7. VeReturnIfStatement : As discussed in 2.7, some statements need to be predicated on return bits. This class represents those statements. It contains a *_condition* string (which is a bitwise and of some return bits) and a *_trueBody* which is the *VeStatement* being predicated. It differs from the *VeIfStatement* in that the condition is a string.
8. VeForStatement : It represents a verilog for statement. It has the following members:
   (a) _initExpr : A list of initialization statements(*VeStatement*) for the loop.
   (b) _updateExpr : The update that happens at the end of every iteration of the loop.
   (c) _checkExpr : The condition for exiting the loop.
   (d) _body : A *VeStatement* representing the body of the loop. For multiple statements, this would still work, since *VeBasicBlock* derives from *VeBasicBlock*.
9. VeWhileStatement : It represents a verilog while statement. Its members are the exit condition of the while loop and its body. They are both of *VeStatement* type. For the condition, the *_printStatementTerminator* bit is set to false (1).
10. VeCaseStatement : It represents a verilog case statement. Its members are the *SgStatement* *_condition* and a vector of *VeCaseEntry* each of which have a *_matchCondition* and *_body*.

11. VeInstanceStmt : It is used as a place-holder for verilog instances in the procedural code. It is useful for printing _statementsBefore and _statementsAfter as well as predicating for returns.

3.2.4 VeInstance

This is a class for a verilog instance. Its members are:

1. _instanceId : A unique name string assigned to an instance.
2. _moduleName : The name of the verilog module for this instance.
3. _args : A vector of strings, each representing the port connections for the instance.

3.2.5 VeForGenerate

This is a class for the verilog generate-for construct. Its members are the exit condition of the for generate loop and a vector of intances within the for generate loop. The iteration variable of the loop is always *genvar_i* and the update expression is always *genvar_i = genvar_i+1*.

3.2.6 VeReg

This class represents a verilog *reg* or *wire* variable. It has the following members:

1. _name : A string for the reg/wire name.
2. _type : An *SgType** to infer the bit-width of the reg/wire.
3. _isWire : A boolean to distinguish if the variable is a reg or wire.

3.3 Algorithm and Data Structures

As mentioned before, the simulator files need to be read, parsed, analyzed and as a final step printed to verilog. ROSE compiler does the reading and parsing for us and presents an object model of the C

code that the user can traverse.

We analyze the ROSE Object Model in two passes. The first pass is used to identify the input and output ports for each of the modules and the second pass populates the verilog OM. The final step goes through the verilog OM and prints verilog code. In this section, we go through each of these steps in further detail.

3.3.1 Pass 1 - Ports and CCP Analysis

In 2.2.1, we outlined the need for a call graph and CCP analysis in determining the ports for every module. Here we further detail how we go about populating the ports for every module using our first pass through all the functions in the input files.

As a first step, we create the call graph for our input files using the *CallGraphBuilder* class. We need to be careful about which functions are a part of the call graph. Functions that are a part of the C headers like *stdio.h, stdlib.h, math.h* need to be ignored, since the source code for these are not be available and separate verilog libraries need to be created for them. The *buildGraph()* function takes as an argument a predicate function that returns true if the function should be a part of the call graph and false if not. We also identify our root function-node for our analysus, which is the function with the *while(1)* loop.

Next, we need to identify all the class member variables being read or written in a given function and create mappings from function declarations to a list of its read/write and write-only variables. The maps *funcToWriteOnlyVarsMap* and *funcToReadWriteVarsMap* are populated in this step. We also create a set of all class-members across the program called *allSeqVars*. All variable references (*SgVarRefExp*) in the function are iterated over. Their corresponding *SgInitializedName* are obtained and tested for being class members as opposed to local variables. Those variables that are being written to are pushed into *funcToWriteOnlyVarsMap* and those that are being referenced in any way

are pushed into *funcToReadWriteVarsMap* in the map entry for the current function being evaluated.

Result: How to populate the map variables

```
for All functions in project do
    for All variables in function do
        if variable is a class member then
            if variable is being written in function then
                add variable to funcToWriteOnlyVarsMap;
            end
            add variable to funcToReadWriteVarsMap;
            add variable to seqVars;
        end
    end
end
```

In order to find the CCP for every member variable, we iterate through all the member variables from *allSeqVars* and pass them into the function *getCCPForVariable* twice, for read/write and write CCP analysis. The *getCCPForVariable* function takes as input a member-variable and one of the maps generated in the first step. Using the map, it finds out all the functons that access the variable. Using the call graph, every function's call trace/s to the root can be calculated. This is done by the function *getPathToRoot* by recursively finding a predecessor and pushing it onto a vector that is returned once the call trace reaches the root node. In this manner we can obtain a vector of all possible call traces for this member variable. These vectors start with the root function and have the same contents until the CCP after which they may diverge. The last common function in a forward walk of these functions is identified as the CCP.

The *getCCPForVariable* function returns by reference the variable *functionToVarMapWithTrace* that is also map from variables to their accessing function like *funcToReadWriteVarsMap* or *funcToWriteOnlyVarsMap* except that it accounts for the fact that the members need to be input or

output ports for all functions between the CCP and point of access. On the vector of traces obtained above, we set all functions from the CCP down to the end of the trace as having the member-variable be an accessor.

```
Result: getCCPForVariable function
for each function in funcToVarMap for variable do
    get path to root for function;
    add path to pathsVector ;
end
sameNodes = true;
index = 0;
while sameNodes is true do
    for each path in pathsVector do
        if path[index] is not same for all then
            sameNodes = false;
        end
    end
end
return pathsVector[0][index];
```

There is a complication for class objects being member variables for other classes. This is true for the renamer module. As there can be more than one of these, we need to create multiple member variables of such a class. We do it in the following manner : We find out the class type of the member-variable we are currently processing. Next, we go through the traces that we generated previously in backward order, i.e. starting from the leaf in the call graph and check if that function is a member of the same class as the member variable we are processing. The first function for which this is not true is the one where the objects last exist. At this point we rename the variable by creating a new *SgInitializedName* for each object that the function used and put this renaming into a *renamedVariablesMap*. This renaming is also passed to every function up the trace till the CCP. Also the *renamedVariablesMap* variable is returned back by *getCCPForVariable*.

At the end of this analysis, we know the CCP function for every variable. We also know which functions(modules) need to have what variables as input and output ports based on the content of the variables *functionReadWriteVarMapWithTrace* and *functionWriteVarMapWithTrace*. There is one minor nuance that is still unaccounted for. Some variables are expected to have different functions as their *readCCP* and *writeCCP*. The *writeCCP* is lower on the call trace than the *readCCP*. A model where a module accesses a sequential variable from its instance seems suspiscious. Therefore, we make this variable an output port for all functions in the call graph between the writeCCP and readCCP for that variable by making the required changes to *functionWriteVarMapWithTrace* in the function *propogateWriteCCPtoReadCCP*.

3.3.2 OM population

With our knowledge of the input and output ports for a module, we can proceed into populating its other parts. As mentioned before, we perform our analysis on a per-function basis. We iterate over all valid functions in the program and pass them to the function *processFunctionBody*. This function analyzes the function's signature and the maps *functionToVarMapWithTrace* and *funcToReadWriteVarsMap* to populate the input and output ports of the module. It also takes into consideration factors like arguments being passed by reference and so on.

We first poupulate the non-blocking statements for our module. These are the variables whose readWriteCCP was this function. As mentioned in the previous chapter, these variables are renamed using the *renameVariable* method we have for every module. Hereafter, all references to the variable in the blocking-assignment code will have the renamed variable. Since *VeExprStatement* directly accepts a string form of a statement, we use this function. It is harder to create explicit statements in ROSE, since ROSE requires them to be linked to an AST.

Next, we iterate over all the statements in the C function and pass them to the function *convertToVerilog*. The function *convertToVerilog* is recursive. It accepts an *SgStatement* and converts it to

an equivalent *VeStatement* or *VeInstance*. Before it begins analyzing the statement, it checks if the statement has any return conditional bits (1) associated with it. We will get to how a statement got these bits a little later. For now, the return predicates for this statement are and-ed into a condition string for a *VeReturnIfStatement*.

The *convertToVerilog* function has a large switch-case statement to deal with all the possible C statement types. For most statements, the relationship is exact. For example, a *VeBasicBlock* is a vector of *VeStatement*s just as a *SgBasicBlock* is a vector of *SgStaement*s. In this case, *convertToVerilog* recursively calls itself to create a *VeStatement* for each of its *SgStatement*s. The *VeStatement*s are then pushed into a vector, a *VeBasicBlock* object is made from the vector and returned. The advantage of deriving every statement from an abstract *VeStatement* type allows us to make the recursive calls.

A similar straightforward recursive approach also works for *SgIfStmt*, *SgForStatement*, *SgWhileStmt* and *SgSwitchStatement*. The statements where the handling needs to be improvised are the following:

1. SgReturnStatement : In 2.7, we outlined the tasks to be performed when a return occurs. In 3.2.3, we presented the specialized *VeReturnAssignStatement*. An object of this class needs to be created.

   A unique predicate variable for this return statement is created and is set. The functionality for 1 is achieved by calling the function *markReturnConditionalOnOtherStmts*, which is a wrapper function for *markReturnConditinoalOnRightSiblings*. *markReturnConditinoalOnRightSiblings* is a recursive function that maps the recently created return prediate bit on every node to its right on the AST. What this means in terms of code is that all statements following the current statement get predicated. Children of a node on the AST are indexed from left to right. So, all nodes with a greater child-index are identified as being on the right of the current node and are predicated. The function is then called recursively on the parent. We skip the predication when the parent is an if-statement, since its children are mutually exclusive. The

return conditional bits mapped here are later used when those nodes are passed into the *convertToVerilog* function as the condition string for the *VeReturnIfStatement*.

As mentioned before, it is difficult to create authentic *SgStatement*s without them being a part of the AST. Therefore, the *return<i> = 1'b1* statement is created as a string and passed to the *VeReturnAssignStatement* constructor. Also passed to the constructor are the return value expression (if any) and whether this return was inside a loop(2.7).

2. SgVariableDeclaration : A C declaration may or may not be followed by an initialization. In either case, we create a new *VeReg* for every C variable declared. For every initialized variable, an assignment string is created and a vector of such strings is formed. The vector is used to create a *VeInitializationStatement* object and returned.

3. SgExprStatement : C expressions could exist as independent statements. Most commonly they are assign statements and function calls. An object of type *VeExprStatement* is created with a string representing the *SgExprStatement* and returned.

3.3.3 Expressions

Expressions are ubiquitous in C code. One of the reasons they cannot be directly translated into verilog is because they may include function calls. In our implementation, we create instances for function calls. The point of function call needs to be replaced with the correct output port of the corresponding instance and a *VeInstance* needs to be created for the module. The former is accomplished by populating the data structure functionData. Every member of this vector is a replacement string and the position and length of the sub-string to be replaced in the original expression string. This vector will be used during decompile.

The other part of handling function calls is creating a *VeInstance*. There are three parts to populating the port-connections of the *VeInstance*:

1. Arguments : The first port connections are the actual arguments in the C code. They were also a part of the function's signature in the C code and in our module creation, they were numerically the first. To meet the specifications in 2.8.1, *VeStatement* class has the members _statementsBefore and _statementsAfter (3.2.3). Regs are created for each of the input port-connections. Strings for assignment to these regs from function arguments are created and pushed into _statementsBefore vector for this *VeInstanceStmt*.

2. Read/write member variables : These variables are obtained from the functionReadWrite-VarMapWithTrace created in CCP analysis. If they are just read variables for the instantiating module, they are simply added as a port-connection. Otherwise, as in the previous case, a reg is created and assigned to with the member-variable's current value. The string for this assignment is pushed into the _statementsBefore vector and the newly created reg is made a port connection.

3. Write member variables : They are obtained from *functionWriteVarMapWithTrace* and are output ports for the instance. A wire variable is created for every such variable and made a port connection for the instance. A string representing an assignment from the newly created wire to the member variable is made and added to the _statementsAfter vector.

The same steps are followed for *VeForGenerate* statements with the difference that the newly created regs are an array with dimensions equal to $\log_2 nIterations$ and the assignments are made to and from indexed regs and wires respectively. Also, the port connections are indexed since the instances are going to be inside a generate-for loop and the created object is of *VeForGenerate* type.

Such a processing of expressions is done by the function *processExpressionForReplacements()*. The function creates the *VeForGenerate* and *VeInstance* objects and adds them to the module. It returns the variable *functionData* that we described earlier. While functions are the primary targets of the variable, we also use it to perform replacements for other constructs, like type casts , renamed

variables(3.3.2) and payload buffer (2.9). The function *processExpressionForReplacements()* is called whenever an expression needs to be added to the OM, for example the condition of if-statements.

3.4 Decompile

Once the verilog OM has been populated, it is possible to traverse it and print all the constructs in a sequential manner. We do so by creating a virtual function called *decompile()* in the *VeStatement* class. All the classes derived from *VeStatement* implement their *decompile()* routines. The *VeModule*, *VeInstance* and *VeForGenerate* class also have their *decompile()* member functions.

From the *main()* function, we call the *decompile()* routine for each module after its OM has been populated. Aside from printing the module signature, ports and regs, the decompile routine is called for every element *_combStatements*. Since this is a virtual function, the appropriate *VeStatement* class is identified by dynamic linkage and its decompile routine is called. The same sequence of events is repeated when the decompilation of any *VeStatement* is required in another *VeStatement*, say the *VeStatement*s in a *VeBasicBlock*.

3.4.1 Expressions

One of the intricacies involved in the decompilation of expressions is the presence of *this->* in the reference of any class member. We perform a pass over the expression to eliminate all occurances of *this->*. Also, the string replacements obtained from *_functionData* need to be made during decompilation.

We also perform the substitution of all input port names with their corresponding output port names (2) in the expression at this stage.

3.4.2 Initialization

The predicate bits that we created for returns and function enables need to be initialized to 0. The output ports(for member variables) also need to be initialized to their input values. These are made in the decompile routine of *VeModule* class before the *VeStatement*s in the *always @ (*)* block are decompiled.

3.4.3 Before and After Statements

Some *VeStatement*s have *_statementsBefore* and *_statementsAfter* attaced to them (because of instances). These are printed before and after the body of those statements.

CHAPTER

4

RESULTS

We test the tool on *721sim*, which is an in-house superscalar processor simulator. The features implemented so far are consistent with the functionality of the rename stage and the renamer module. Therefore, we translate the functions for these into verilog. For *721sim*, these functions are:

1. processor::rename2 - The function that does the renaming. This gets translated into the module processor_rename2.

2. renamer::find_free_bit - Returns if a new branch can be checkpointed. Gets translated to renamer_find_free_bit.

3. renamer::map_copy - Copies the maps from one to another ex. Architectural Map Table to Rename Map Table. Gets translated to renamer_map_copy.

4. renamer::stall_reg - Checks for a stall in the rename stage. Translates to renamer_stall_reg.

5. renamer::stall_branch- Checks for a stall in the case of absence of unavailable checkpoints. Translates to renamer_stall_branch.

6. renamer::get_branch_mask - return the Global Bit Mask. Translates to renamer_get_branch_-mask.

7. renamer::rename_rsrc - Returns the current physical register mapping for a given logical register. Translates to renamer_rename_rsrc.

8. renamer::rename_rdst - Returns a free physical register and creates a new mapping for it. Translates to renamer_rename_rdst.

9. renamer::checkpoint - Checkpoints the RMT. Translates to the module renamer_checkpoint.

10. renamer::dispatch_inst - Creates an Active List entry for the instruction. Gets translated to renamer_dispatch_inst.

11. renamer::is_ready - Checks if the ready bit for a physical register is set. Gets translated to renamer_is_ready.

12. renamer::clear_ready - Clears the ready bit for a physical register. Gets translated to renamer_-clear_ready.

13. renamer::set_ready - Sets the ready bit for a physical register. Gets translated to renamer_set_-ready.

14. renamer::read - Reads the Register File (RF) for a value of a physical register. Gets translated to renamer_read.

15. renamer::write - Writes a value to the RF. Gets translated to renamer_write.

16. renamer::set_complete - Sets the completed bit on an Active List entry. Gets converted to renamer_set_complete.

17. renamer::resolve - Frees the GBM for a correctly predicted branch and resets various renamer structures for an incorrectly predicted branch. Gets translated to renamer_resolve.

18. renamer::commit - In essence, updates the AMT, but also a lot of other things. Gets translated to renamer_commit.

These functions cover a significant part of the supersclar processor, as is evident from 4.1. The data structures in the register renaming class (RMT, AMT, Shadow Map Table, Active List, Free List, Physical Register File and Physical Register Ready bits) are an integral part of the superscalar processor.

505 lines of C++ simulator code were translated to 1,319 lines of verilog.

In order to make sure that the generated RTL from our tool is working, we need to test the functionality of the generated verilog.

As a preliminary test, we perform syntax analysis by compiling the generated verilog with modelsim and design vision. We next proceed to verifying its correctness.

4.1 Testbench

With the C to verilog translations we performed, we are able to track the entire sequential state of the renamer. However, one problem is that some of the renamer functions (like commit, resolve) get called outside the rename2 stage. For example, commit is called in retire stage, which we aren't completely able to translate for now. We need a way to keep track of the state of the renamer members during the execution of those modules, even though we do not have their verilog code.

In a hypothetical scenario where all the pipeline stages were available in verilog, there would be a given number of instances of all those modules. By analysing the simulator code, this can be

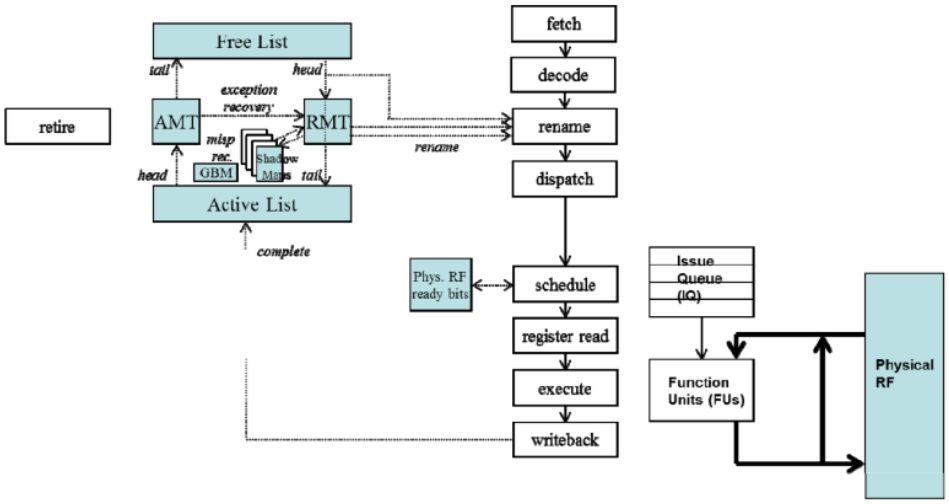

Figure 4.1 Shaded blocks correspond to data structures in the Register Renaming Class. Dotted lines/arrows correspond to referencing these data structures through methods of the class.

inferred. Also, there is be an order in which these functions are called. This is important because the renamer module's sequential elements will need to be modified in that order by the instances in order to obtain correct results.

In our testbench, we create instances for all the renamer modules that are instantiated outside of the rename2 module. These other instances accept one or more of the renamer members as input and may modify some of these members. Therefore, we perform the kind of renaming we did in 2.8.1. Some of these instances can be inside for-loops. We also handle this case by creating arrays of those wires and indexing them into the instances in the for-gen loops.

We must remember that there are two objects of the renamer class in *721sim*, REN_INT and REN_FP. Therefore, each of the renamer members has two copies which need to pass through and be updated by different instances.

We also create an instance of the processor_rename2 module. Its inputs are all the sequential elements of both REN_INT and REN_FP type and the contents of the RENAMER2 pipeline stage. Its output are the contents of of the DISPATCH pipeline stage.

4.1.1 Stimulus

The instantiated modules have enable inputs, like all modules that we generate in this work (2.8.2) that are necessary for proper functioning. Some of the modules also have inputs aside from the sequential elements of the renamer module. These inputs are arguments of their respective functions in the C simulator.

We run the simulator for 50000 cycles and obtain a dynamic trace of the enable values and the value of the arguments in a verilog format with the appropriate variables from the test bench. Every cycle in verilog is an iteration in the *while(1)* loop and therefore we print a time delay at the start of the new iteration.

We also print the contents of the RENAME2 register and DISPATCH[0].valid in verilog format for

the testbench. These prints are directed to stimulus.sv file that is ‘include-d in the testbench.

4.1.2 Verification

The output of the RENAME2 stage, which is the contents of the DISPATCH register such as the mappings for source and destination logical registers, are compared against those obtained from the run of *721sim*. We obtained an exact match for 575 cycles of gcc benchmark.

4.2 Modifications in *721sim*

One of the assumptions in our analysis is that a pointer necessarily means an array. In *721sim*, the renamers REN_INT and REN_FP are declared as pointers and need to be changed to objects.

4.3 Further Work

So far in this work, we are passing huge data structures like the RMT as a bundle of wires to some module as input and back up as output. This needs to be changed to an indexing mechanism, whereby the read and write indices (and write data) are passed to the top level module where the RAM is instantiated and the corresponding values are passed down wherever necessary.

The body of work also needs to be extended to cover the remaining pipeline stages so that we can have a fully functional RTL description of the processor.

BIBLIOGRAPHY

[Arvrc] Arvind. “Why chip design can’t be left to EE’s”. *UC. Irvine* (March 2004).

[Bacne] Bachrach, J. et al. “Chisel: Constructing hardware in a Scala embedded language”. *ACM/EDAC/IEEE Design Automation Conference(DAC)* **49** (June 2012), pp. 1212–1221.

[GGly] G.Martin & G.Smith. “High-Level Synthesis: Past, Present and Future”. *IEEE Design & Test of Computers* **26**.4 (July-August 2009), pp. 18–25.

[JLH12] John L. Hennessy, D. A. P. *Computer Architecture - A quntitative approach*. Morgan Kaufman, 2012.

[Lab] Laboratory, L. R. N. *Rose Compiler Framerwork*. URL: `rosecompiler.org/ROSE_HTML_Reference`.