USING VHDL SYNTHESIS AND VLSI LAYOUT TOOLS FOR COST ESTIMATION OF SUPERSCALAR ISSUE UNITS

BY

MATTHEW TODD GAVIN

B.S, University of Iowa, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Professor Wen-mei Hwu, for his support and mentorship during my two years as a graduate student at the University of Illinois. I would also like to thank John Gyllenhaal for his guidance throughout this project. I express thanks to my colleague and fellow Master's student Dimitri Argyres for working closely with me throughout the project. I am grateful to Professor Rajesh Gupta for acquiring and helping to install the Synopsys toolset. Finally, I wish to thank my wife Ann who endured my grumpiness and long hours in the laboratory as the thesis deadline approached.

TABLE OF CONTENTS

Page

1. INTRODUCTION

This thesis describes the interfacing and use of VHSIC Hardware Design Language (VHDL) synthesis tools and Very Large Scale Integration (VLSI) layout tools to obtain and compare speed and area estimates for the issue unit of a superscalar processor. It also describes results obtained through the use of these tools.

When designing a processor and its issue unit, the designer fixes various parameters, such as the issue width, the numbers of source and destination operands per instruction, and the number of available register file ports. Each design decision plays an important role in the performance of the issue unit and each tradeoff must be thoroughly considered. This is in direct conflict with the increasing time-to-market pressures faced in the competitive industry of processor design.

Thus this thesis has two goals. The first is to develop a methodology which allows the designer, through the use of VHDL, to quickly investigate the effects of his/her decisions on the speed and area requirements of the design unit in question. The second is to use

this methodology to examine a specific design unit, namely the issue unit of a superscalar processor.

A current topic of great interest in the computer architecture research community is the tradeoffs involved in choosing between a VLIW processor design and a superscalar processor design. Both processor designs attempt to extract as much instruction-level parallelism (ILP) from a program as possible; they significantly differ in their approach to this task. Very Long Instruction Word (VLIW) processors rely heavily upon intelligent compilers to do static code scheduling, determining at compile time when instructions will be executed. Superscalar processors, on the other hand, dynamically schedule the code at run time through the addition of extra hardware. Thus an argument in favor of the VLIW processor is that this extra hardware will significantly increase the hardware and cycle time required for a superscalar processor compared to those for a VLIW processor. VLIW processors themselves have several drawbacks. Their static code scheduling may be unable to utilize information available at run time. Static code scheduling also leads to difficulties maintaining binary code compatibility across processors.

Thus a great debate has focused on the relative merits of each type of processor. The tools and techniques developed in this thesis may resolve some of the questions posed in this debate. To facilitate quick investigation of various designs, it was decided that VHDL code would be written that would describe various portions of a processor design. For this thesis, portions of a superscalar issue unit were described in VHDL. Then, synthesis and layout tools would be used to take this VHDL code to a gate-level

design and subsequent VLSI layout. This code would be highly parameterizable; the user could input values for various parameters, then re-synthesize the code and evaluate the effect of changing the parameter on the design's speed and area performance.

A long term goal of this work would be to interface the IMPACT group's MDES code [1] with this tool. The IMPACT group at the University of Illinois uses MDES language descriptions of processors, when doing compilation for VLIW and superscalar processors. These same descriptions, in addition to supporting sophisticated compilation techniques, could potentially be interfaced with the tools developed in this thesis to generate layouts and timings for various parts of the processor in question.

Figure 1.1 shows an overview of the path followed in converting VHDL code to a VLSI layout. The path can be thought of as a three-step process: first synthesizing the design in Synopsys, then converting the Synopsys design to Mentor, and finally obtaining a layout in Mentor.

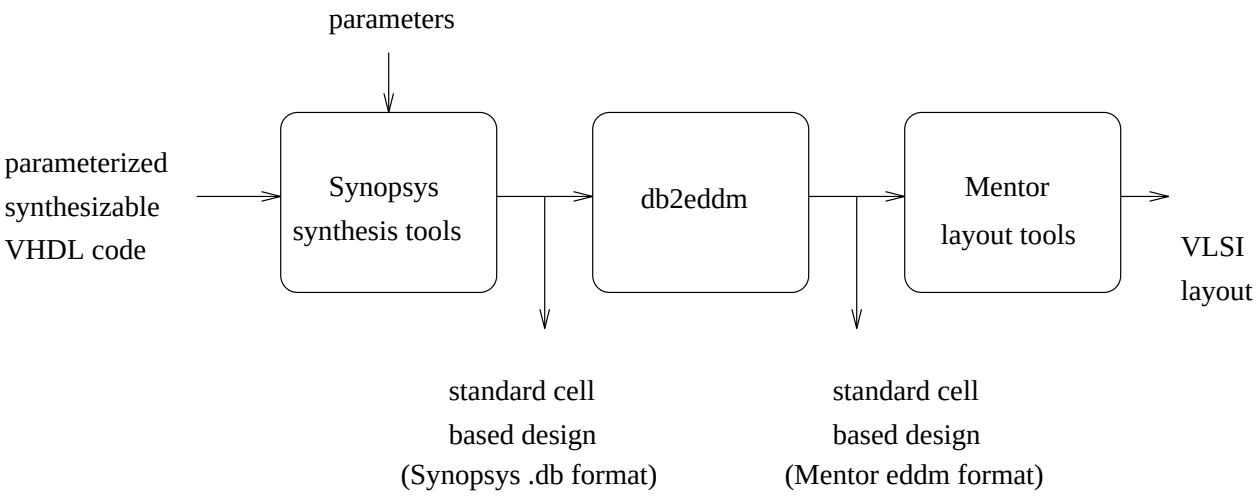

Figure 1.1: Overview of the Synthesis Process

This thesis is organized as follows. Chapter 2 describes the process shown in Figure 1.1, including the tools and the techniques used in interfacing them. Chapter 3 describes in detail the portions of the processor which were modeled in VHDL, synthesized, and brought to a VLSI layout. Chapter 4 describes the area and timing results subsequently obtained. Chapter 5 provides some concluding remarks. The appendices supply the VHDL code and descriptions of the scripts used to automate the methodology.

2. METHODOLOGY

2.1 Tools Used

For this project, it was necessary to first synthesize a VHDL design into a standard-cell based design and then a CMOS layout. No tool known to the author is able to carry VHDL from synthesis straight through to layout; thus different tools were needed for each task. Synopsys' *Design Compiler* (version 3.1b) was used to synthesize the VHDL to a standard-cell based design; Mentor Graphics' *IC Station* (version 8.2_5) was used to create a layout of that design.

These tools were used for two reasons. First, these were the best accessible tools for their respective tasks. Synopsys is recognized as a leading VHDL synthesis tool, and was fairly easy and quick to learn. The author had a great deal of prior experience with Mentor, which has been a leader in many areas of chip design, including simulation and layout. In addition, Dimitri Argyres had special knowledge of Mentor's layout tools; this knowledge was an invaluable aid to the author.

Second, Synopsys provides special support for transferring designs to and from Mentor, in the form of a toolset named SIFF (Synopsys Integrator for Falcon Framework). The SIFF program *db2eddm* is especially useful; it converts a Synopsys cell-based design (in db format), to a cell-based design in Mentor's eddm format. Thus the interface between Synopsys and Mentor was relatively easy to use and understand.

2.1.1 Synopsys' important features

Synopsys' Design Compiler was the tool used to generate standard cell-based designs. An important feature of the tool is its support of generics, allowing the VHDL code to be parameterizable. A component of a processor design may exhibit the same regular structure across different implementations, differing only in the amount of hardware needed. By using generics, the designer can write parameterized code modeling this component, then vary the parameters at compile time. This allows the compiler to generate the correct amount of hardware, in the same form specified by the code.

Synopsys allows the user some control over the hardware it generates; this was another important feature. Synopsys provides the designer control through two means: *optimization directives* and *optimization constraints*. Constraints instruct the synthesizer to attempt to produce hardware with a certain delay, or within a certain area. Directives such as flattening and structuring tell the synthesizer *how* to restructure the hardware. These two features give the user a measure of control over the hardware produced by the synthesizer; however, the synthesizer can not always meet the constraints the designer imposes. In all cases, Synopsys will generate detailed reports on the timing

and area requirements of the design it produces. The designer can use these reports to determine if the design obtained by Synopsys is acceptable. If not, the designer may try a new set of compilation options.

Both of these features offered by Synopsys were important. The goal of this thesis was to allow the designer to explore quickly a wide variety of designs. This can be done by writing parameterized, behavioral (as opposed to structural) VHDL code, and setting various compile time options appropriately.

2.2 Cell Library Used

In addition, it was necessary to choose the standard cell library. Synopsys would synthesize gate-level designs using these base cells; Mentor would then place and route these cells to provide a layout of the design. It was decided to use the CMOSN standard cell library that comes with Mentor's layout tools, since that cell library was readily accessible. It was also known that Mentor's layout tools were able to successfully route CMOSN cells; the author wished to avoid any difficulties involved with attempting to route other cells. This entire methodology, including layout, should work with any standard cell library that has cell layouts compatible with the layout tools.

The CMOSN library uses a 1.2 micron process. This feature size is large compared to those being used today; currently 0.5 micron processes are being used to fabricate processors. This should be kept in mind when results are presented in Chapter 4; area and delay both scale with smaller feature sizes.

2.3 Design Flow Overview

The flow is as follows: (see Figure 1.1):

1. Synthesize the VHDL using Synopsys
2. Convert the Synopsys cell-based design to Mentor format using db2eddm
3. Run a custom layout script on the Mentor cell-based design to generate the layout.

Scripts have been written to automate the process. One script handles the first item above. Another script, written in large part by Dimitri Argyres, handles the translation to Mentor's eddm format and subsequent layout generation. It may be necessary, however, to bypass the scripts and do the analysis by hand, in order to give the synthesis tools the proper feedback. The flow will now be described in further detail.

2.4 Generating a Standard Cell-Based Design in Synopsys

Synopsys' Design Compiler is the primary tool used to generate a standard cell-based design. The tool may be accessed through a command-line interface (dc_shell) or through an X window menu-based interface (Design Analyzer). Scripts may be written for batch processing by dc_shell. The end result is a cell-based design stored in Synopsys' db format. See Figure 2.1.

2.4.1 Analyzing the VHDL code into a VHDL design library

The *analyze* command does a syntax check on the VHDL code. Some logical errors may not be caught, most notably those arising from using legal VHDL code that is not supported by Synopsys. Analyzing the VHDL code is NOT included in the automation

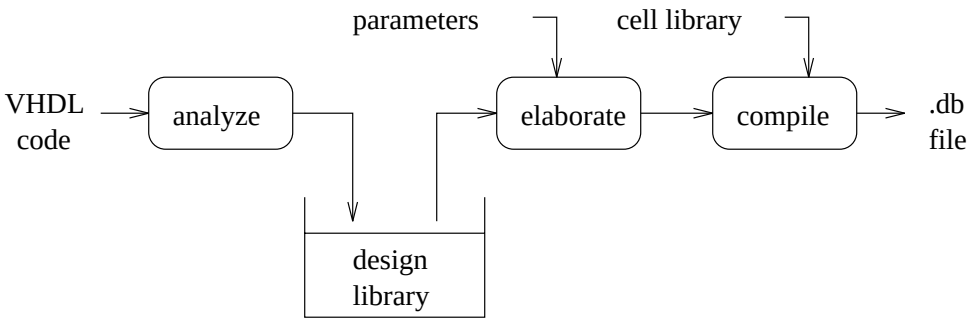

Figure 2.1: Synopsys Design Flow

scripts, because the VHDL code was intended to be behavioral code written by a human (as opposed to a completely structural code that may be written by a program). Thus the code may have syntax errors before this analysis/syntax check. Often multiple iterations of this step are necessary to remove all syntax errors from the VHDL code and successfully analyze the design units into a design library. After this has been done, the scripts automate the rest of the process.

2.4.2 Elaborating the analyzed design

The *elaborate* command takes an analyzed design from a VHDL design library and converts it to an intermediate equation-based format. At this time, the necessary parameters must be supplied so that all bit field widths, register sizes, etc. can be completely resolved before the next compilation step.

2.4.3 Compiling the elaborated design

Compilation performs the important task of mapping the equation-based design to a standard cell library. The end result of compilation is a synthesized standard-cell based

design file in db format. The cell library must be provided by the user. Synopsys in particular needs the ASCII file describing the function of each cell in the library, such as `Z <= NOT(A)` for an inverter. Although layouts were originally available for the CMOSN library, such a description file did NOT exist. Thus one was purchased from an external vendor to avoid the difficulty and delay of creating it.

When generating a file in db format, it is useful to have a symbol in Synopsys sdb format for each standard cell in the design. This collection of symbols in Synopsys format, called a symbol library, is especially useful when using Design Analyzer to view the design. SIFF's *eddm2slib* program was used to convert the CMOSN symbols from Mentor format to a Synopsys symbol library.

Different compilation directives, such as flattening and structuring, may be specified during the compile. These directives are very useful and often necessary when synthesizing behavioral code.

Flattening uses boolean algebra to remove all parentheses. This removes all logic structure, producing a design with a two-level sum of products form. This directive tends to decrease the critical path delay, but at the expense of greatly increased hardware.

Structuring adds intermediate variables (structure) to the equations in the design. *Timing-driven* structuring does so while considering required timing constraints. In Synopsys it is possible to set the maximum allowed delay to an output; the synthesizer will iteratively structure until the constraint is met, or until it cannot continue. For example,

it is possible to set the maximum delay to the output to 0; Synopsys will iterate until some point, then stop.

An *incremental compile* may also be specified, which does local optimizations on a circuit that has already been mapped to standard cells. This option is also very useful, and was used extensively to decrease the critical path on Synopsys' original designs.

2.5 Converting the Synopsys Design to a Mentor Design

As stated earlier, the SIFF program db2eddm was used to convert the Synopsys .db file to a Mentor component in eddm format. db2eddm requires a *map* file. This map file acts like a lookup table; for each standard cell in the Synopsys design, the map file specifies the name and location on disk of Mentor's corresponding standard cell.

db2eddm creates more than one component if the design is hierarchical. If a single component is instantiated multiple times in the VHDL, multiple versions of that component in Mentor will be created. This is unfortunate; if 1000 instances of the same design are explicitly instantiated in the VHDL code, 1000 Mentor components will be created, each identical to the rest. It would be optimal if only one of the designs were created, and all instances in the top-level design referred to it.

2.6 Generating a Layout from the Mentor Schematic

A layout script is run to obtain the VLSI layout corresponding to a Mentor standard cell-based design. This script does the following:

1. identifies the mentor component whose layout is to be created and the layout library to be used (CMOSN),
2. loads a design rules file,
3. autofloorplans the layout,
4. autoplaces standard cells, or blocks if the design is hierarchical,
5. autoroutes the layout,
6. minimizes vias,
7. compacts the design downward, then left,
8. checks to make sure design rules are not violated,
9. if specified, runs LVS to verify that the layout matches the cell-based schematic, and
10. saves the layout to disk.

The process of generating the Mentor layout is different for hierarchical schematics than for flat ones. If the schematic has hierarchy, the layout script must first be used to generate the layouts for the components instantiated by the top-level design. These layouts must then be added to the CMOSN library. After this has been done, the layout script may be run on the top-level Mentor schematic. The script will then take from the CMOSN library the layouts for the instantiated components. To avoid these difficulties, the code in this thesis was written and synthesized to produce flat schematics.

3. VHDL IMPLEMENTATIONS

This thesis had two goals: to develop a methodology allowing a designer to obtain, from VHDL, gate- and circuit-level schematics for a design, and to use these tools to measure characteristics of a superscalar processor's issue unit. In theory, VHDL code would be written once, and different parameters (corresponding to a different design) would be input to the synthesizer. I specifically focused on implementing in VHDL the following aspects of a superscalar processor's issue unit:

- dependency checking
- register file port arbitration
- instruction queue shifting.

The parameters referred to in various designs are as follows:

- w - issue width
- s - number of source registers per instruction
- d - number of destination registers per instruction
- r - number of bits in register identifier
- p - number of ports on the register file.

3.1 Dependence Checker

In a superscalar processor, it is determined at run time which of the previously fetched instructions are allowed to execute. Dependence checking enforces certain restrictions on the instructions which are allowed to proceed. Instructions precluded from executing are those instructions reading from, or writing to, a register being written to by a previous instruction. Comparators are used to pairwise check if any instruction's source or destination register ids match a destination id of a previous one. Comparators are also used to pairwise check if instructions use the same register file. (No dependencies exist between instructions using different register files, no matter what register ids they specify.)

The following VHDL data type `instr_t` was used for the dependency checker. It models an instruction as a record having an array of source operands and an array of data operands. A register file id is also included, which specifies which register file the instruction reads from and writes to.

```
type instr_t is record
   opcode: opcode_t;
   regfile_id: regfile_id_t;
   srcs: src_array;
   dests: dest_array;
end record;
```

Figure 3.1 shows dependence arcs for an example dependency checker with $w = 4$, $s = 2, d = 1$, where w is the issue width and $s(d)$ = number of source (destination) operands per instruction. Note that a total of nine dependence checks must be satisfied for the last instruction to be dependence free.

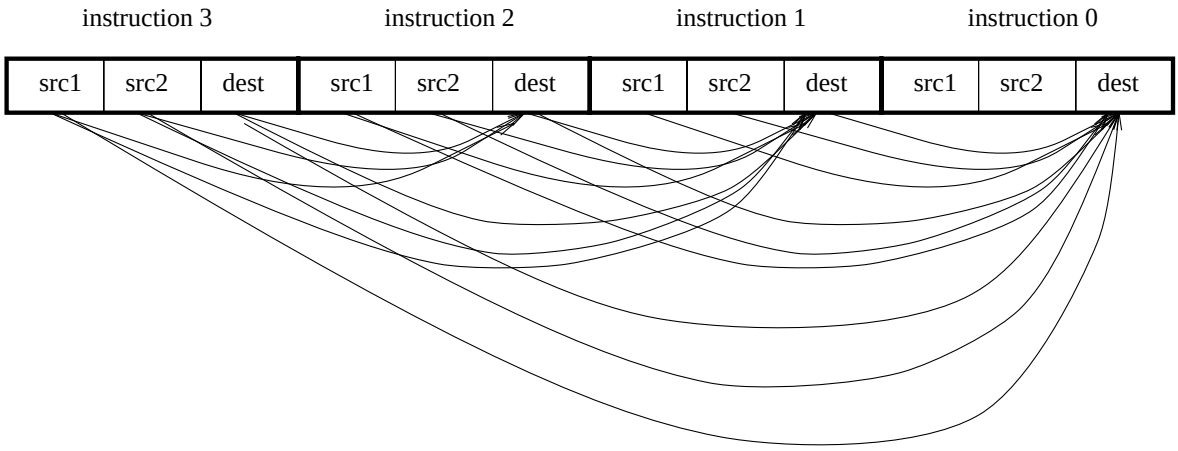

Figure 3.1: Dependence Arcs for $w = 4, s = 2, d = 1$

In general, the number of register id comparators required is $(s + d) * d * \binom{w}{2}$. The first and second term come from comparing all of an instruction's source and destination ids to the destination id of a previous instruction. The last term is a result of performing this comparison for each pair of instructions. Similarly, the number of register file id comparators needed is $\binom{w}{2}$. Thus the area required for dependency checking hardware may be expected to be proportional to $\binom{w}{2}$, which may be expressed as $(w^2 - w)/2$. This hypothesis will be tested in Section 4.

3.2 Register File Port Arbiter

In a multiple-issue processor architecture, many register file ports are needed to satisfy the possible operand requirements of issuing instructions. The maximum number of ports required is $w*s$. However, in practice, that many ports are often not required, since many instructions in a real instruction set require fewer than s operands, and dependencies limit the number of instructions that can simultaneously issue. Since register file ports

require much hardware, chip savings may be made by reducing the number of register file ports and adding hardware allowing the instructions to arbitrate for them. The obvious tradeoff is weighing the cost of that hardware against the hardware savings from reducing the number of ports.

My VHDL implementation of the arbiter follows that of Johnson [2]. This method uses a great deal of hardware to do arbitration in the least amount of time. Johnson's method for a p-ported register file is as follows:

- If the first register operand requires a register access, it is always enabled on the first port.
- If the second register operand requires a register access, it is enabled on the first port if the first register did NOT require a register access, and is otherwise enabled on the second port.
- In general, the register identifier for a required access is enabled on the first port if no other previous operand requires a port, on the second port if one previous operand uses a port, on the third port if two previous operands use ports, ..., on port p if $p-1$ previous operands use ports, and on no port if p previous operands use ports.

Johnson's method amounts to instantiating a one-counter for each {register id, port} pair. See Figure 3.2.

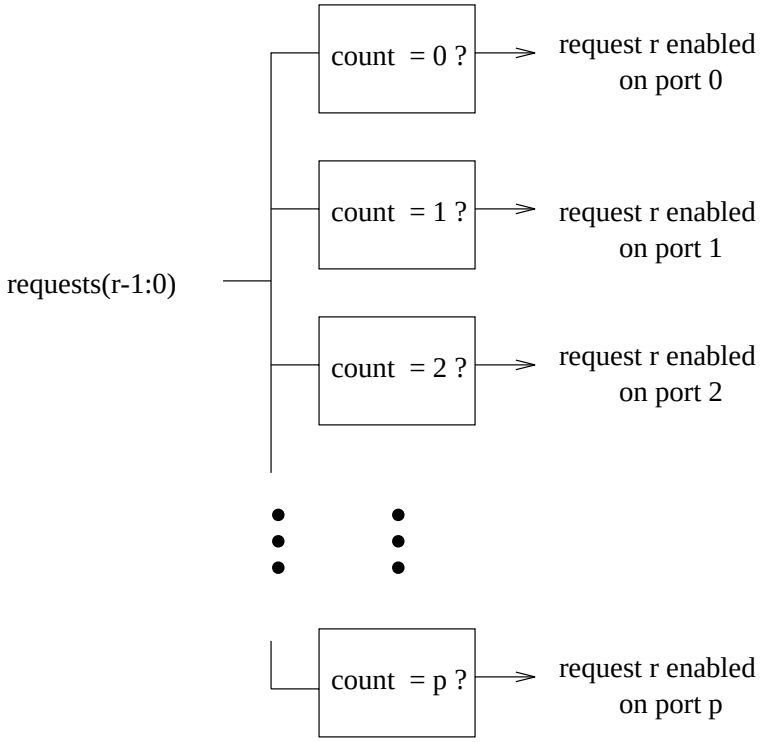

Figure 3.2: Request r Arbitrates for Port p

Johnson suggests implementing each counter as a two-level AND-OR network. Consider a four-issue processor with two source registers per instruction, for a total of eight requesters, all contending for four ports. Request 3 is enabled on port 1 if

request(0) and not request(1) and not request(2) or

not request(0) and request(1) and not request(2) or

not request(0) and not request(1) and request(2).

It is easy to see that the length of the OR-reduction done in each counter grows combinatorially. Determining if c ones of a vector of length v are logic-1 requires $\binom{v}{c}$ AND gates.

The VHDL code written for this thesis implements this algorithm equivalently:

```
for each request r in the instruction queue
    for each port p
        count if exactly p of last r requests are logic-1;
        if so, enable the request r on port p
    end
end
```

It was desired to implement the one counters as two-level AND-OR networks as Johnson suggests, in order to do the arbitration quickly. Instead, behavioral code was written that simply looped over a bit vector and incremented a variable whenever logic-1 was encountered. This behavioral code could give very undesirable results when synthesized, for instance, if the synthesizer decided to instantiate adders and counters. However, through optimization directives, it is still possible to obtain a two-level AND-OR format for the counters. This is possible in the Synopsys compiler by turning the flattening option on and turning the structuring option off. (The compiler still may not be able to generate a true two-level and-or format, since it typically only has available standard cells with less than five inputs. Thus multiple levels are synthesized; however, as the number of inputs grows, two levels of logic, each with few inputs, are often faster than one level of logic with many inputs.)

3.3 Instruction Queue

In a superscalar processor, between 0 and w instructions are issued to execution units each cycle. In many current implementations, an in-order issue model is used. This means that instructions must be issued in program sequence. Each cycle, instructions in the

queue are analyzed to determine if they may issue. Once it has been determined that an instruction can *not* issue, analysis conceptually stops, and only the previous instructions are issued. Thus the instruction queue may be implemented as a barrel shifter, which can shift from 0 to $w - 1$ positions every cycle, depending on how many instructions issue. If all instructions issue, a full load of the queue is performed. See Figure 3.3.

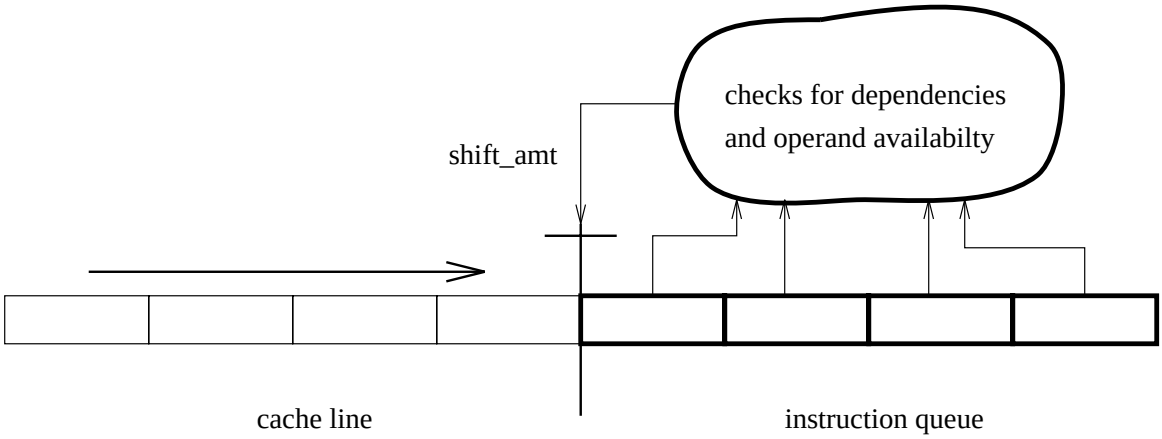

Figure 3.3: Barrel Shifting Instruction Queue, $w = 4$

The issue analysis determines how much of the incoming cache line the queue shifts in. In the model for this thesis, an instruction may *not* issue if a dependency exists between it and a previous instruction, or if a value the instruction requires is unavailable. In a real processor, the following checks may be made to determine if a register value is available:

- checking the register file's "valid bits" to determine if the value in the register is up to date;
- determining if the value must be forwarded by bypass logic; and
- determining if the register id successfully obtained a register port (if not forwarded by bypass logic).

The VHDL code assumes this logic is elsewhere and lumps the resulting information into an input vector. This input vector, `stream_avail`, tells for each source register in the instruction queue, if that value is currently available for use. The outputs of the dependency checker are also input to the design, as opposed to embedded within it. This allowed separate analysis of dependency checking and barrel shifting hardware. The critical path of the fetch unit would be determined by the sum of the times required to do dependence checking and subsequent barrel shifting.

Combinational logic within the shifter examines the dependencies and availability of registers and forms an *issue vector*. For each word in the instruction queue, the issue vector contains a corresponding bit telling if that instruction can issue. Due to the in-order nature of the issue model, the issue vector is a series of logic-0's followed by consecutive logic-1's. This issue vector is then priority encoded; this encoded value directly controls how many positions the queue should shift by. The barrel shift may be implemented by placing a mux in front of each bit in the queue; thus, for an instruction queue of length w, conceptually, a mux with $w + 1$ inputs sits in front of each bit.

The code uses the `instr_t` data type used by the dependency checker, with a slight change. A "filler" field is added which will take up any extra bits in the instruction word, not taken by the opcode or register specifiers. The user can specify the constant `instr_size`; the filler field will pad the opcode and register identifiers with an extra field to get this required length.

Not included in the VHDL code were the flip-flops needed in the instruction queue to store the instructions. This was done because these flip-flops are necessary in all multiple issue processors. However, the shifting logic is NOT necessary in a VLIW processor, where either all instructions in the queue issue, or none. It was desired to isolate the extra logic required by the issue unit of a superscalar processor, in order to measure the area and timing overhead it incurs.

3.4 Notes on Parameterization of VHDL Code

The design flow in Figure 2.1 shows parameters input during elaboration, after the code has already been compiled. This is unfortunately not possible with all parameters, because Synopsys disallows data types with more than one unknown. Typically it is desired to vary the issue width w; thus `instr_stream` is defined as an array of `instr_t` of length w. Parameters such as the number of source and destination operands within `instr_t` must be hard-coded in the VHDL package which defines the `instr_t` data type. These may be changed, but the package will have to be recompiled.

4. RESULTS

4.1 Dependence Checker

Dependence checkers were synthesized for various values of w and r, for $s = 2$ and $d = 1$. See Table 4.1. Throughout this thesis, all Mentor layout sizes are reported in units of square lambdas.

Table 4.1: Dependence Checker Synthesis Results

	delay (ns)		layout (10^6)
w	r=6	r=7	r=6
2	3.94	3.95	0.506
3	4.77	5.48	1.649
4	7.08	7.80	3.292
5	7.15	7.86	5.534
6	8.16	8.87	8.826
7	9.53	10.24	12.790
8	9.53	10.24	17.561
9	10.52	11.23	23.520
10	10.52	11.23	29.760

Figure 4.1 shows the gate-level schematic for a dependency checker with $w = 4$, $s = 2, d = 1$, and $r = 6$. Figure 4.2 shows the corresponding layout.

4.1.1 Analysis of delays

Theoretically, for constant s, d,and r, the delay should be logarithmic in w. The delay can be broken into two serial components: time needed for register id comparisons, and time needed to perform an OR-reduction of the comparator outputs. The time required for the comparators grows with r but is constant with w. The time required for the or-reduction grows with w, since the number of comparator outputs increases. When using standard cells with a maximum of n inputs, the number of bits b that can be reduced in l levels of logic is $b = n^l$. Thus the number of levels required to reduce b bits is $l = \frac{\log b}{\log n}$. Since n is constant, the number of levels of logic required to do an or-reduction is logarithmic with the number of bits to reduce. As seen in Section 3.1, the number of comparators needed, and thus the number of bits to reduce, increases with w^2. The logarithm of w^2 is $2 * \log w$. Thus, it has been shown that the theoretical time required to do dependence checking is logarithmic in w.

Figure 4.3 plots the delays given in Table 4.1. It is evident that the delay increases with issue width. The delay does appear to increase logarithmically.

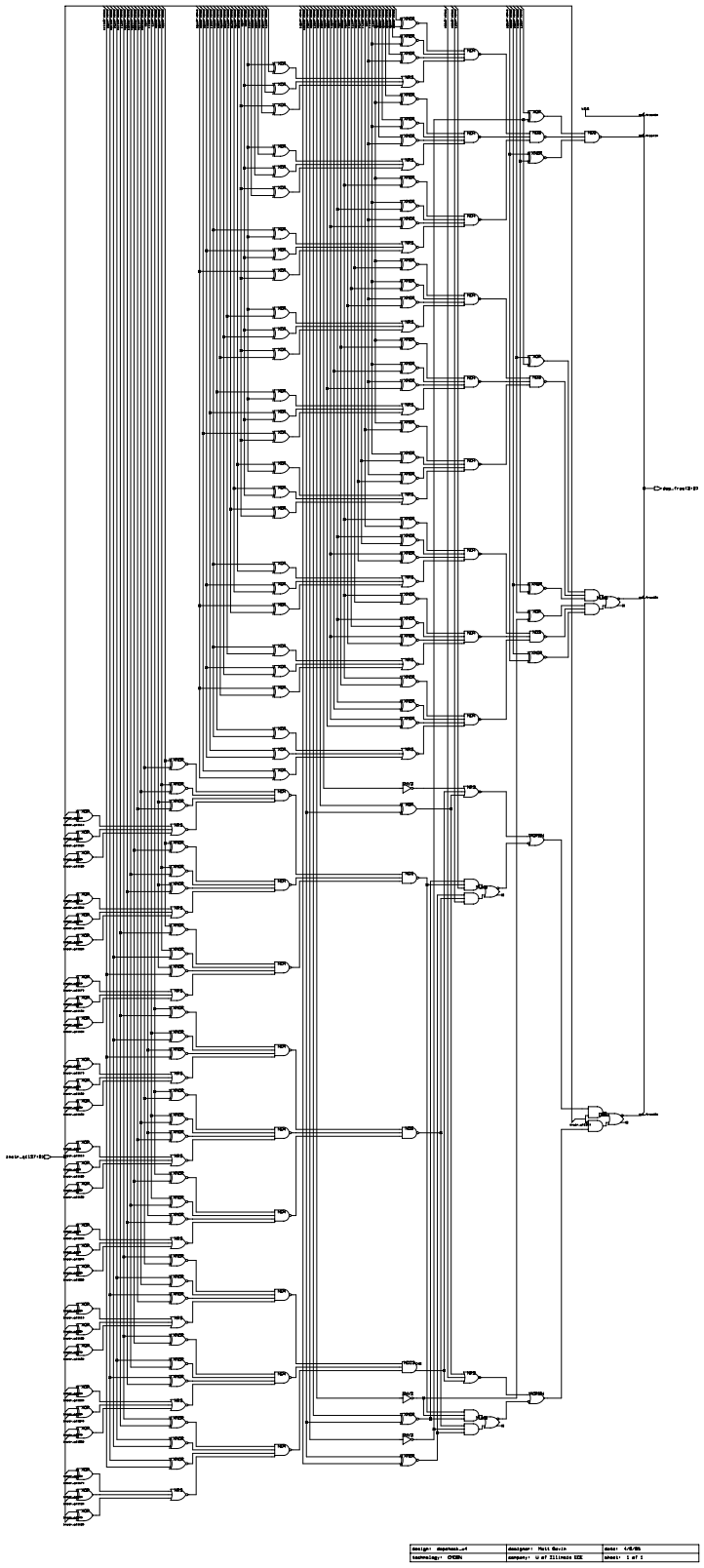

Figure 4.1: Gate-level Dependence Checker, $w = 4, s = 2, d = 1, r = 6$

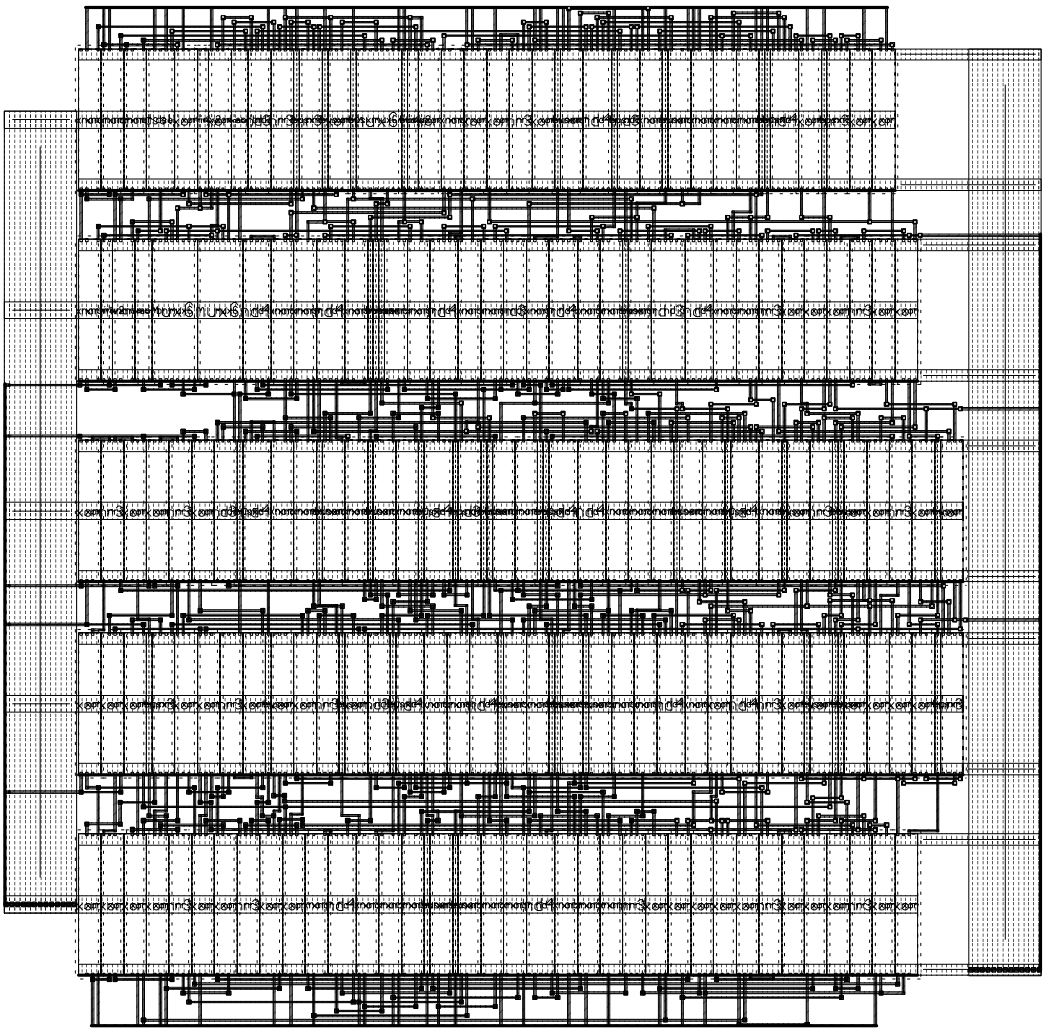

Figure 4.2: Layout for Dependence Checker

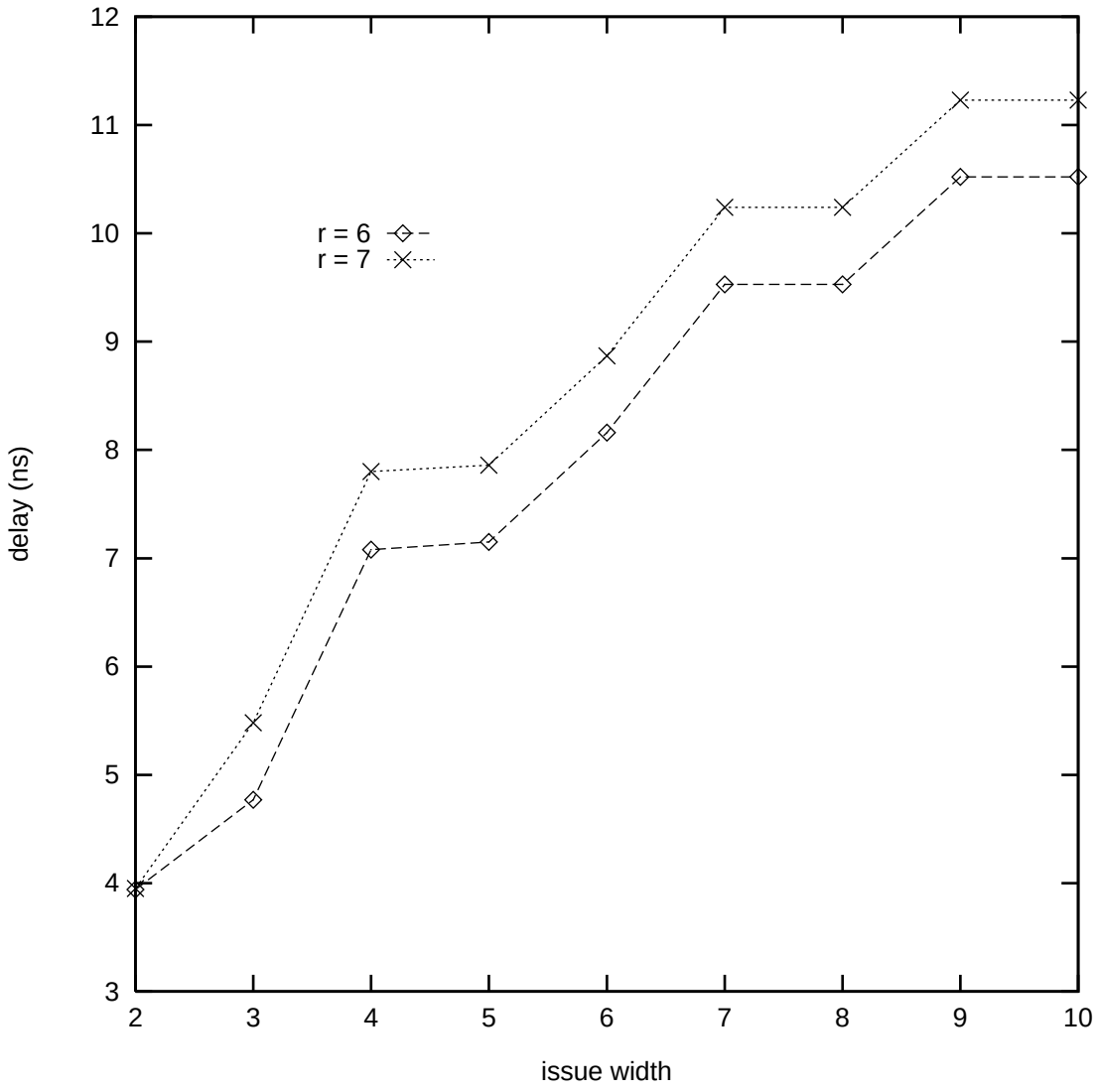

Figure 4.3: Dependence Checker Delay Times

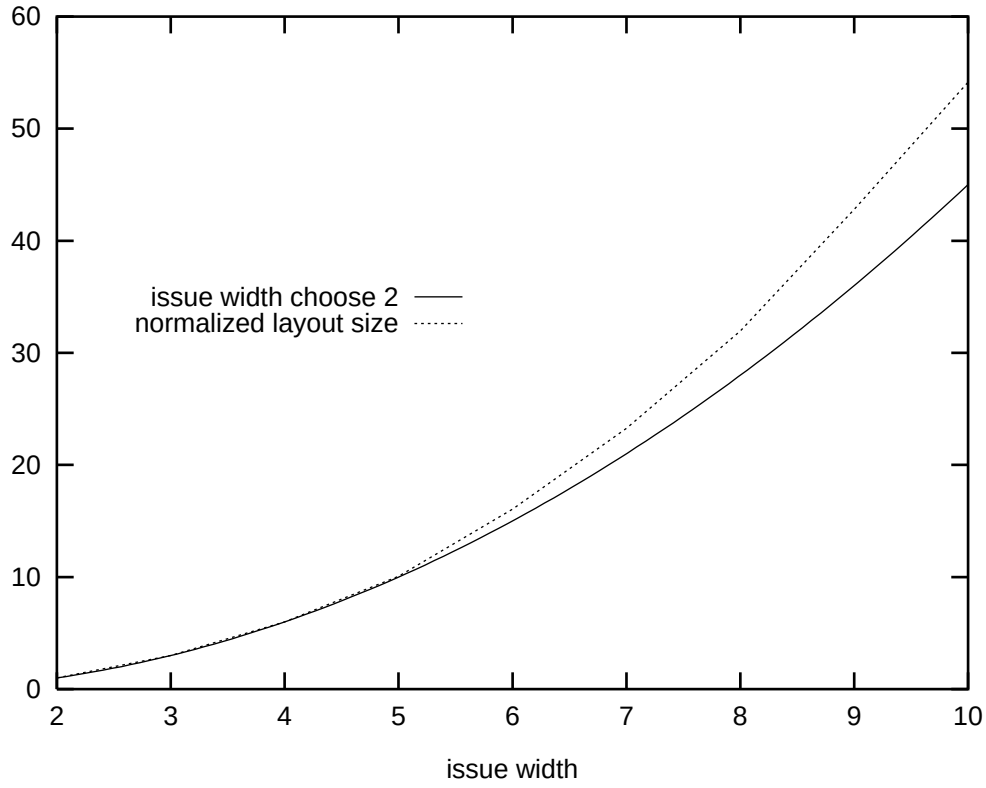

Figure 4.4: Dependence Checker Normalized Layout, $r = 6$

4.1.2 Analysis of areas

Figure 4.4 shows the increase in layout size for increasing issue width, for $r = 6$. Section 3.1 showed that the number of comparators required rises with $\binom{w}{2}$. The layout increases only slightly greater than this amount.

4.2 Register File Port Arbiter

Register file arbiters were synthesized for various values of w and p. The number of sources s was left constant at 2. See Table 4.2 for a summary. Due to the huge amounts

of hardware, arbiters for $w > 5$ were not synthesized. Figure 4.5 shows the gate-level schematic produced by Synopsys for an arbiter for $w = 3, s = 2$, and $p = 3$.

Table 4.2: Arbitration Unit Area and Delays

w	p	layout (10^6)	delay (ns)
2	2	0.243	5.52
	3	0.400	5.77
3	3	1.420	6.62
	4	1.988	7.10
4	4	6.767	9.94
	5	9.252	11.07
5	5	34.106	19.12
	6	47.261	19.20

For constant w, as ports are added, the delay slightly increases. More importantly, the delay, and especially the layout size, increase rapidly with w. These are plotted in Figures 4.6 and 4.7. For each value of w, the delay or area for the smallest investigated value of p is plotted. A major increase in both measurements is seen for $w = 5$. It is evident that for processors with $w > 4$, this method of arbitration may be prohibitively expensive, due to area and timing problems. Hardware costs could be reduced by combining the logic for some of the one-counters; however, that would negatively impact the delay through the arbiter.

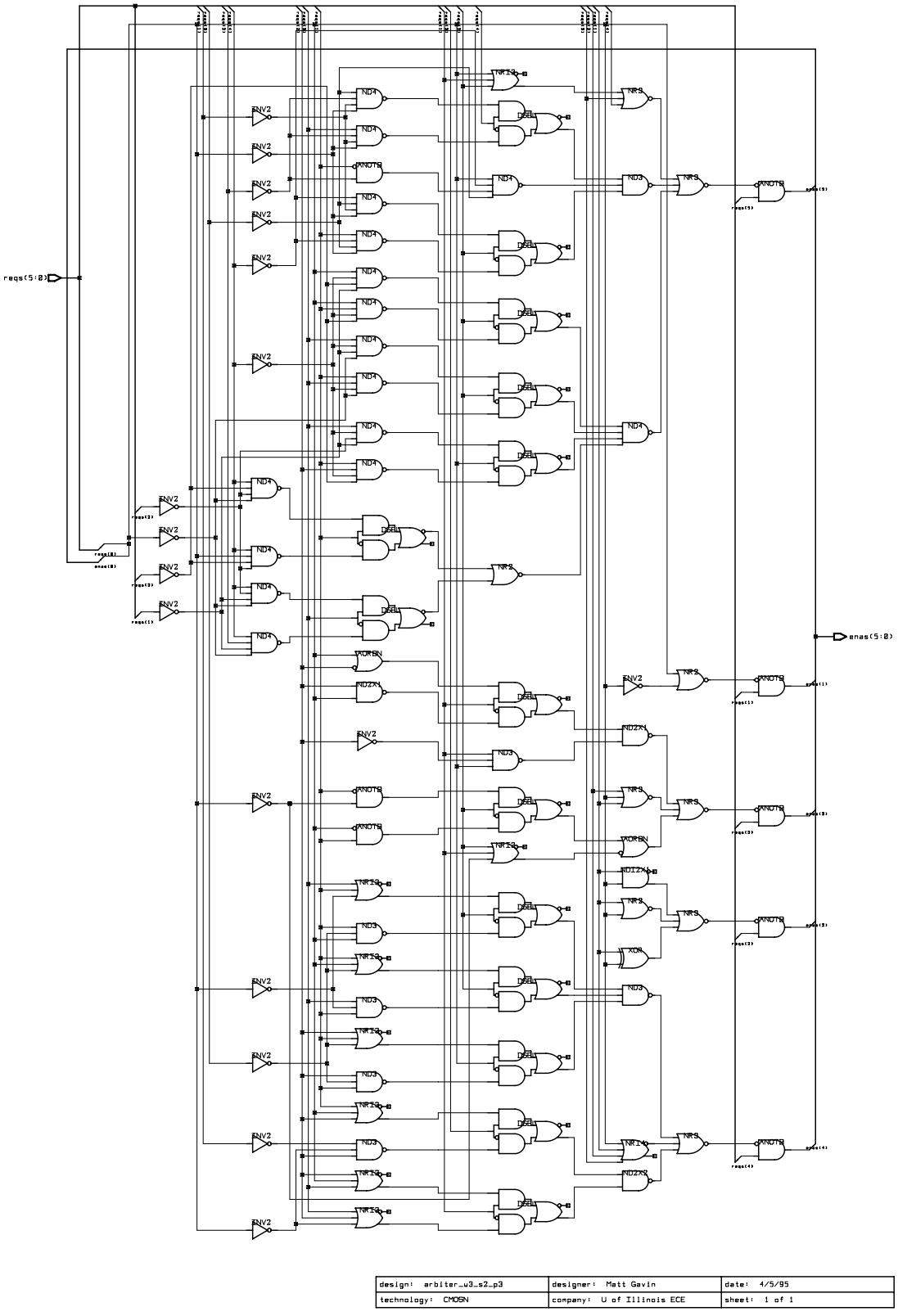

Figure 4.5: Schematic for Arbiter, $w = 3, s = 2, p = 3$

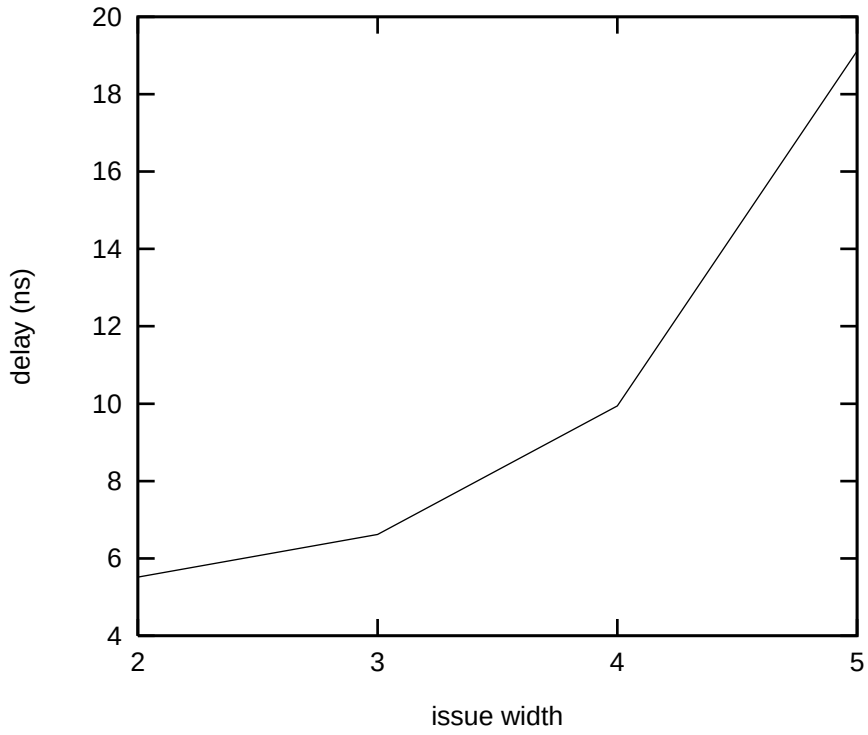

Figure 4.6: Arbitration Unit Delay

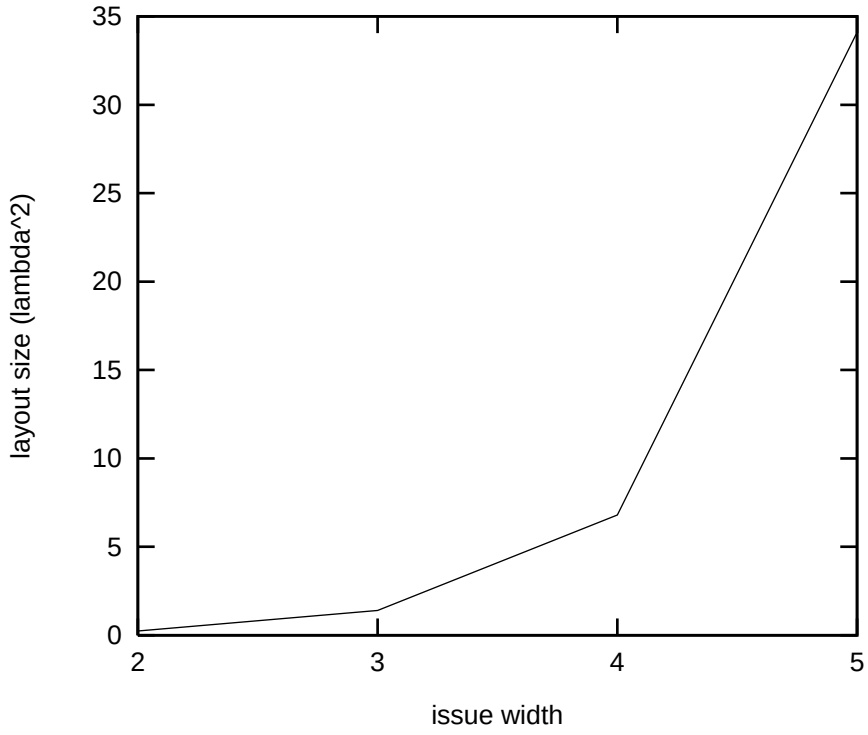

Figure 4.7: Arbitration Unit Area

4.3 Instruction Queue

4.3.1 Using incremental mapping to synthesize better designs

The default options in Synopsys produced shifters with very long latencies. Synopsys uses the linear delay calculation model, which takes into account not only the intrinsic delays through gates, but also the capacitive loads on their outputs. Thus its delay calculations are fairly accurate. See Figure 4.8 for the schematic of the shifting logic for a queue of $w = 2$. See Figure 4.9 for the critical path report.

Recall from Section 3.3 that a small amount of combinational logic drives multiplexers which determine how much the barrel shifter shifts. This can also be seen from Figure 4.8. When synthesizing this logic, Synopsys generated a few cells (ND3 and ANOTB) which had to drive very large capacitive loads. This tremendously increased the delay through the shifter. Synopsys did not try to remedy this, since the delay to the output was unconstrained by the designer. Coercing Synopsys to generate a few more cells for this logic would increase its driving capacity and significantly reduce the delay through the shifter without significantly increasing the chip area.

Five trials were done in an attempt to remedy this problem.

1. defaults (timing driven structuring only)
2. flattening (high effort) and TDS
3. flatten, then TDS
4. set constraint of 7 ns output delay
5. set constraint of 5 ns output delay

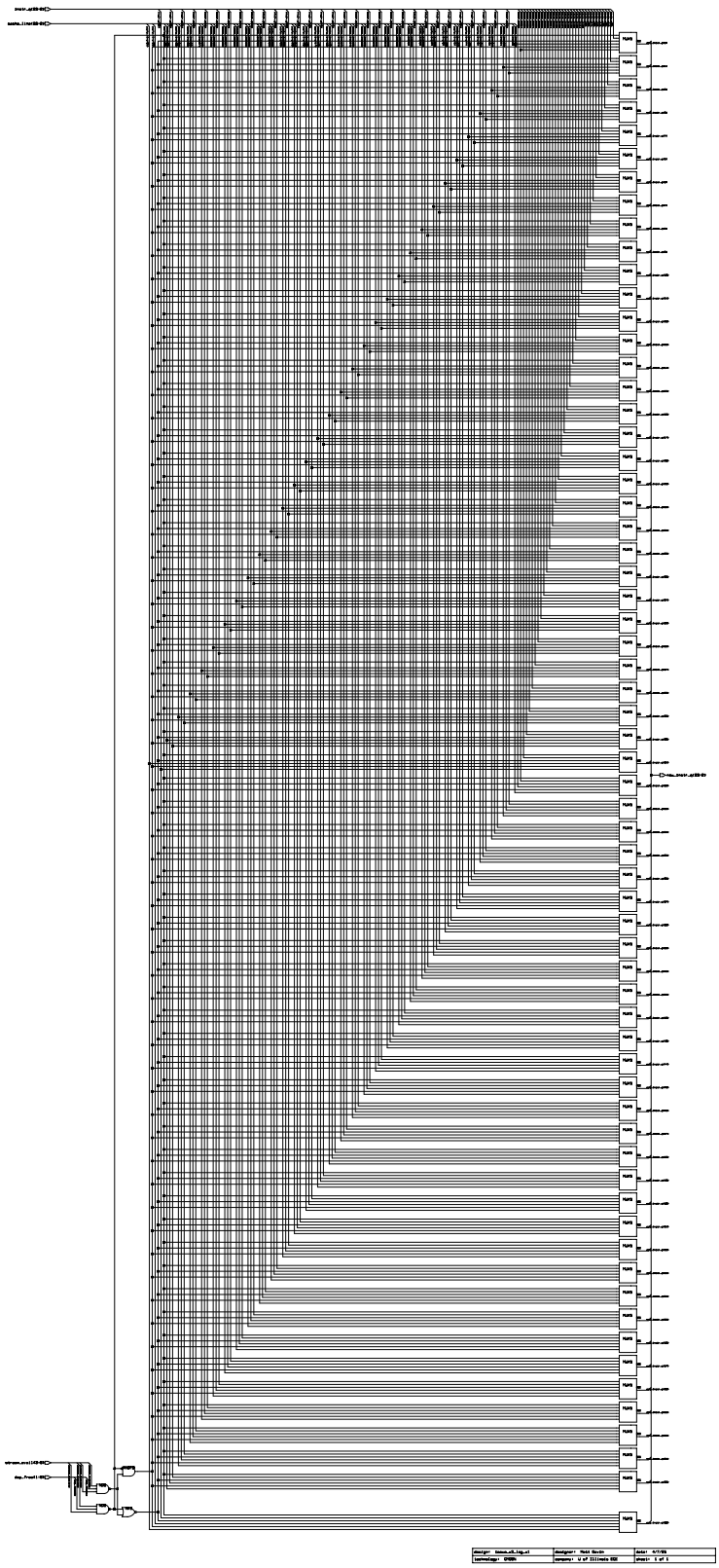

Figure 4.8: Original Schematic for Instruction Queue, $w = 2$

```
Point                                          Incr      Path
-----------------------------------------------------------
input external delay                           0.00      0.00 r
dep_free(0) (in)                               0.00      0.00 r
U1088/OUT (ND3)                               13.04     13.04 f
U1090/OUT (ANOTB)                             26.51     39.55 r
U1041/OUT1 (MUX5)                              1.29     40.84 f
U1041/OUT2 (MUX5)                              0.32     41.16 r
new_instr_q(1][OPCODE][5) (out)                0.00     41.16 r
data arrival time                                       41.16
-----------------------------------------------------------
(Path is unconstrained)
```

(a)

```
Point                                          Incr      Path
-----------------------------------------------------------
input external delay                           0.00      0.00 f
dep_free(0) (in)                               0.00      0.00 f
U107/OUT (ND3)                                 1.76      1.76 r
U102/OUT (NR2)                                 0.80      2.56 f
U113/OUT1 (BUFF)                               0.66      3.22 r
U113/OUT2 (BUFF)                               2.17      5.39 f
U96/OUT1 (MUX5)                                1.13      6.52 r
U96/OUT2 (MUX5)                                0.33      6.85 f
new_instr_q(0][DESTS][0][0) (out)              0.00      6.85 f
data arrival time                                        6.85
```

(b)

Figure 4.9: Timing Reports for (a) Original and (b) Optimized Instruction Queue, $w = 2$

Table 4.3 shows the results of these trials. Layouts were not done for these trials. Instead, the area measurement is one given by Synopsys. Synopsys measured area by summing cell areas, as given for each cell in the CMOSN technology library description file. Through wire load models, it is possible for Synopsys to take wiring into account when measuring area; this was not done, however, for lack of an accurate model.

Areas for various cells were: inverter, 5; 2 input NAND, 7; 2-TO-1 MUX, 15; 4-TO-1 MUX, 25. These areas and all areas reported by Synopsys are unitless and are intended only to measure relative size differences, not absolute layout sizes. In this thesis, any measurements labeled as "area (Syn)" refer to area measurements calculated by Synopsys, not Mentor.

Table 4.3: Instruction Queue Synthesis Results, $w = 2$

	1	2	3	4	5
area (Syn)	1317	1314	1335	2053	2226
delay (ns)	52	41	33	7	5

Table 4.3 clearly exhibits the classic area vs. delay tradeoff. However, most of these designs were suboptimal in that they attempted to completely reconstruct the whole circuit - not just the small but critical amount of logic driving the muxes.

A new approach was chosen: using incremental mapping in conjunction with setting constraints. Incremental mapping will look at small areas in the circuit where significant optimizations are possible. Using this, in addition to setting constraints on output delays,

yielded excellent savings. See Table 4.4 for the result of doing incremental mappings, for various output delay constraints, on the schematic originally produced for $w = 2$.

Table 4.4: Incremental Mappings of $w = 2$ Queue

	1	2	3	4	5
area (Syn)	1328	1349	1384	1457	1618
delay (ns)	20	10	7	6	5

Note that using incremental mapping obtained the same critical path savings as the previous methods, but at much less expense. The design produced for the 7 ns delay appears to be optimal for area and speed. See Figure 4.10 for the schematic. For this mapping, the critical path decreased by 83% while the area increased by only 5.3%.

See Figure 4.9 for the corresponding critical path report. Note that approximately the same number of cells is in the path, but the delay is much smaller since no cells drive large capacitive loads.

The above process was repeated for queues of lengths 3 through 7. See Table 4.5 for a summary of various incremental mappings. Again, large decreases in the critical path came at the expense of very modest increases in hardware. The optimal designs (as chosen at the author's discretion) are listed in Table 4.6.

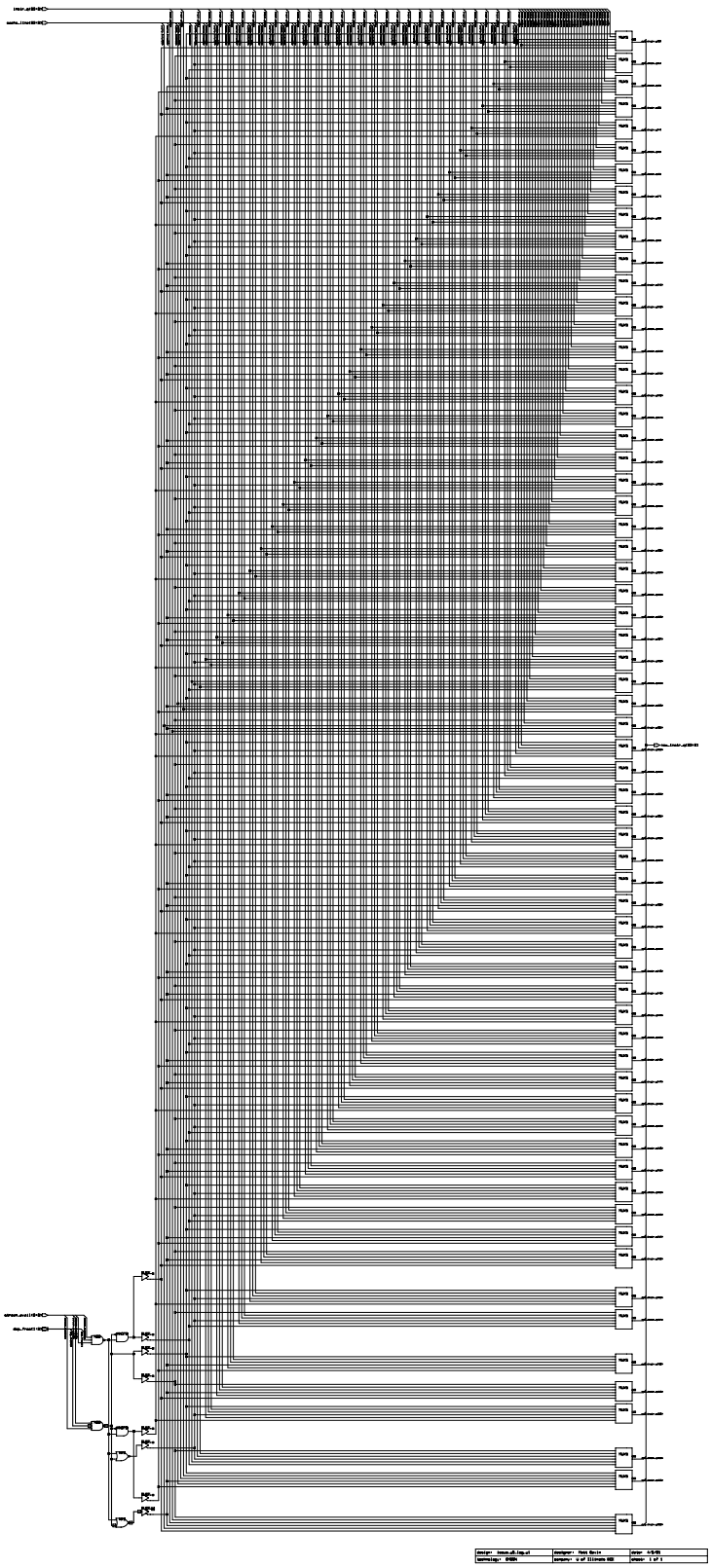

Figure 4.10: Optimized Schematic for Instruction Queue, $w = 2$

Table 4.5: Synopsys Areas and Delays for Incremental Mappings of $w = 3 - 7$ Queues

w		orig	iter 1	iter 2	iter3	iter 4
3	area (Syn)	2461	2504	2504	2549	
	delay (ns)	34	10	8.4	7	
4	area (Syn)	5465	5599	5695	5873	
	delay (ns)	69	15	11	9	
5	area (Syn)	7633	7875	7967	8000	8162
	delay (ns)	103	15	12	11	10
6	area (Syn)	10162	10333	10557	10733	
	delay (ns)	91	20	15	14	
7	area (Syn)	12996	13182	13221	13229	13371
	delay (ns)	67	20	18	17	16.3

Table 4.6: Optimal Designs for $w = 2 - 7$ Queues

w	area (Syn)	delay (ns)
2	1384	7
3	2549	7
4	5695	11
5	8000	11
6	10557	15
7	13371	16.3

4.3.2 Analysis of delays

As shown in Section 3.1, the levels of logic required to reduce b bits goes as $\log b$. Each mux in the queue may be considered a bit reducer. Thus one would expect the delay to increase logarithmically with w.

Table 4.6 shows the optimal delay through the barrel shifter for various values of w. Inspection of the table reveals that the delay grows slower than w but faster than $\log w$. This can be accounted for by considering delays through the driving logic, which

will carry more weight for small values of w, as have been examined here. Thus it may be concluded that in general the delay rises with $\log w$.

4.3.3 Analysis of areas

For $w = 2 - 7$, layouts were completed for the original design and for the optimal mappings as shown in Table 4.6. See Table 4.7. The layout areas given by Mentor were compared with those predicted by Synopsys. Synopsys generally underestimated the percentage increase, and increasingly so with increasing w. This is explained by the failure to use Synopsys' ability to take into account area resulting from wiring.

Table 4.7: Various Shifter Layouts

	issue width					
layout	2	3	4	5	6	7
original (10^6)	2.99	6.29	13.07	19.81	26.31	35.95
best (10^6)	3.16	6.71	14.56	22.92	29.17	40.50
%increase (Synopsys)	5.3%	3.6%	4.2%	4.8%	3.9%	2.9%
%increase (Mentor)	5.7%	6.7%	11.3%	15.7%	10.8%	12.7%

Conceptually, for an instruction queue of width w, with instructions of length l, $w * l$ muxes are required. The area required by a w-to-1 mux follows w, as it requires w 2-input AND gates, as well as reduction logic to or-reduce those outputs. Then the area required for the barrel shifter is $l * w^2$. Thus the area required by a barrel shifter rises as w^2.

Figure 4.11 plots the Mentor layout size vs. w for the values given in Table 4.7. The graph of w^2 is also plotted. The plots match very well.

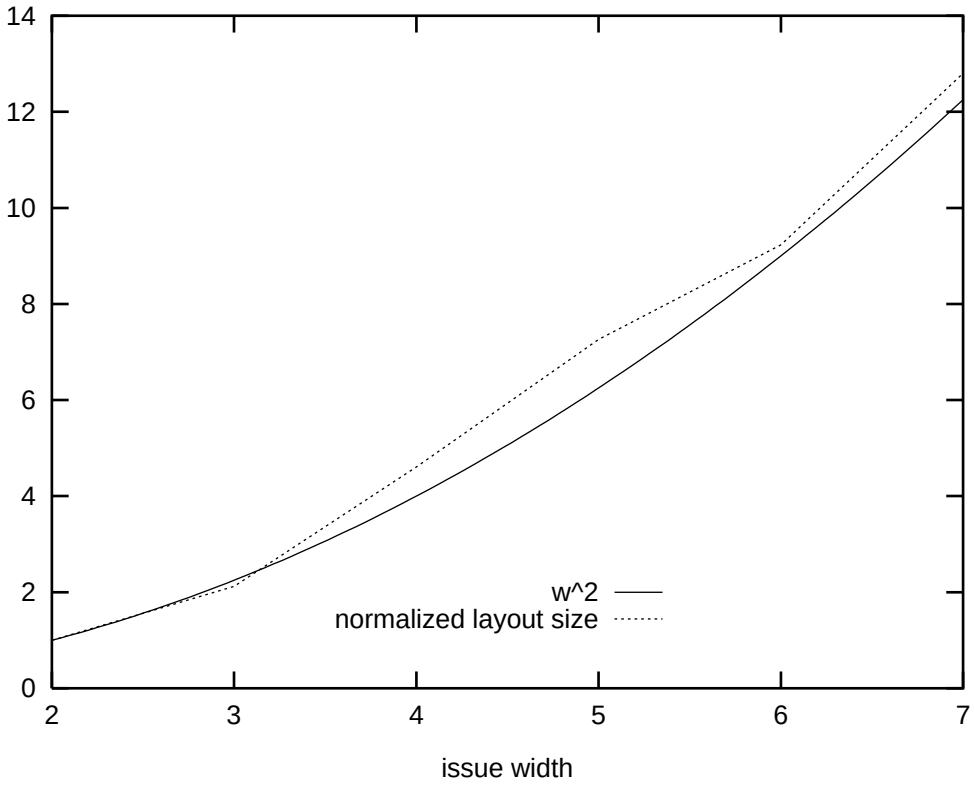

Figure 4.11: Instruction Queue Layouts

5. CONCLUSIONS

This document has described the methodology used to obtain schematics and layouts for VHDL designs (Chapter 2), the VHDL modeling of aspects of a superscalar issue unit (Chapter 3), and the results obtained by synthesizing these models (Chapter 4).

Table 5.1 summarizes the area and delay conclusions reached in Chapter 4:

Table 5.1: Summary of Conclusions

	dependence checker	Johnson's register port arbiter	instruction barrel shifter
area trend	$\binom{w}{2}$ (or w^2 for large w)	expensive for $w > 4$	w^2
delay trend	$\log w$	expensive for $w > 4$	$\log w$

From the results above, when examining chip cost, it may be concluded that neither the in-order instruction queue nor the dependence checker scale well when attempting to extract higher ILP. The register port arbiter took the most hardware, although methods could be used to reduce it at the expense of cycle time. The shifting instruction

queue and dependence checking hardware are required only by superscalar processors. Thus it appears that the complexity of extracting higher levels of ILP in the superscalar processors of the future will require much chip area.

Synopsys (or in general, any synthesis tool) will not yield optimal designs unless it receives feedback from the designer. Indeed, its output is heavily influenced by the user's choice of compile-time options. This greatly affects the results reached, especially those reached for the instruction shift unit. Thus the effort put forth by the user to reach an optimal design greatly affects the accuracy of the results generated by tools such as those in this thesis.

It is hoped that the time and work invested in developing these tools and models will be useful to the ongoing success of the IMPACT project at the University of Illinois.

REFERENCES

[1] J. Gyllenhaal, "A Machine Description Language for Compilation," M.S. thesis, Dept. Electrical and Computer Engineering, University of Illinois, 1994.

[2] M. Johnson, *Superscalar Processor Design*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991.

REFERENCES NOT CITED

[1] R. Lipsett, C. Schaefer and C. Ussery, *VHDL: Hardware Description and Design.* Norwell, MA: Kluwer Academic Publishers, 1989.

[2] *VHDL Compiler Reference Manual, v3.1b.* Synopsys Corporation, 1994.

[3] *Design Analyzer Reference Manual, v3.1b.* Synopsys Corporation, 1994.

[4] *Design Compiler Family Reference Manual, v3.1b.* Synopsys Corporation, 1994.

[5] *Synopsys Integrator for Falcon Framework User Guide, v3.1b.* Synopsys Corporation, 1994.

[6] *System Installation and Configuration Guide, v3.1b.* Synopsys Corporation, 1994.

[7] *AMPLE Reference Manual, v8.2.* Mentor Graphics Corporation, 1993.

[8] *Design Manager Reference Manual, v8.2.* Mentor Graphics Corporation, 1993.

[9] *Design Architect Reference Manual, v8.2_5.* Mentor Graphics Corporation, 1993.

[10] *IC Station Reference Manual, v8.2_5.* Mentor Graphics Corporation, 1993.

[11] *Getting Started with Falcon Framework, v8.2_5.* Mentor Graphics Corporation, 1993.

APPENDIX A. DEPENDENCE CHECKER

The following is the VHDL code used to synthesize the dependence checkers. The code instantiates as many comparators as needed and performs a NOR-reduction for each instruction to determine if it is dependence free.

```
library thesis,ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;  -- for reduction functions
use thesis.types.all, thesis.funcs.all;

entity depcheck is
   generic (
      w: integer
   );
   port(
      instr_q: in instr_stream(w-1 downto 0);
      dep_free: out std_logic_vector(w-1 downto 0)
   );
end depcheck;

architecture behav of depcheck is

   subtype dep_vec_t is std_logic_vector(w-1 downto 0);
   subtype comp_vec_t is
      std_logic_vector((num_srcs+num_dests)*num_dests-1 downto 0);
```

```
begin
   -- pairwise analyze instructions for dependencies
   -- for each instr in instr_q (except 0)
   --      look at previous instrs for deps
   -- check 1 against 0, 2 against 1 & 0, 3 against 2,1,0, etc...

   process(instr_q)
      variable comp_vec: comp_vec_t;
      variable dep_vec: dep_vec_t;
   begin

      dep_free(0) <= '1';

      for instr1 in 1 to w-1 loop
         for instr2 in 0 to instr1-1 loop

            for d1 in 0 to num_dests-1 loop
               for d2 in 0 to num_dests-1 loop
               comp_vec(d1*num_dests+d2) :=
                  compare_fn(instr_q(instr1).dests(d1),
                           instr_q(instr2).dests(d2));
               end loop;
            end loop;

            for s in 0 to num_srcs-1 loop
               for d in 0 to num_dests-1 loop
               comp_vec(num_dests*num_dests+s*num_dests+d) :=
                  compare_fn(instr_q(instr1).srcs(s),
                           instr_q(instr2).dests(d));
               end loop;
            end loop;

            dep_vec(instr2) := OR_REDUCE(comp_vec) and
            compare_fn(instr_q(instr1).regfile_id, instr_q(instr2).regfile_id);

         end loop;

         dep_free(instr1) <= NOR_REDUCE(dep_vec(instr1-1 downto 0));

      end loop;
   end process;
end behav;
```

APPENDIX B. REGISTER FILE PORT ARBITER

The following is the VHDL code used to synthesize the register file port arbiters. For each (request, port) pair it is determined if the request is enabled on the corresponding port.

```
library thesis,ieee,parts;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;  -- reduction function
use thesis.types.all, thesis.funcs.all;
use parts.components.all;

entity arbiter is
   generic( w, s, p : integer);
   port (
         reqs: in std_logic_vector(w*s-1 downto 0);
         enas: out std_logic_vector(w*s-1 downto 0)
   );
end arbiter;

architecture struct of arbiter is

   constant num_reqs: integer := w * s;
   subtype ena_vec_t is std_logic_vector(p-1 downto 0);
   type enable_matrix_t is array (natural range <>) of ena_vec_t;
   signal ena_vecs: enable_matrix_t(num_reqs-1 downto 0);
   signal one_cnt_signal: std_logic_vector((p*num_reqs)-1 downto 0);
```

```
begin

   requester: for r in 1 to num_reqs-1 generate
      read_port: for rp in 0 to my_min(r,p-1) generate
               ctr: one_cnt_lastn_eq_c
                  generic map(num_reqs, r, rp)
                  port map(reqs, one_cnt_signal(r*p+rp));
      end generate;
   end generate;

   enables: for i in 0 to num_reqs-1 generate
      enas(i) <= reqs(i) and or_reduce(ena_vecs(i));
   end generate;

   process(reqs, one_cnt_signal)
      variable req_ena_vec :  ena_vec_t;
   begin
      -- check if 'r_port' of last 'req' requests are 1's
      -- if so, enable the request on this port

      for req in 0 to num_reqs-1 loop   -- req = requester id
         for r_port in 0 to p-1 loop
            if r_port <= req then
               -- if request 0, don't need to examine request vector
               if req = 0 and r_port = 0 then
                  req_ena_vec(r_port) := reqs(req);
               else
                  req_ena_vec(r_port) :=
                     reqs(req) and one_cnt_signal(req*p+r_port);
               end if;
            else
               -- never enable request on port with higher id
               req_ena_vec(r_port) := '0';
            end if;

         end loop;
         ena_vecs(req) <= req_ena_vec;
      end loop;
   end process;
end struct;
```

APPENDIX C. INSTRUCTION QUEUE

The following is the VHDL code used to synthesize the instruction queues. An issue vector is formed; this is used to determine if the queue does a full load, shifts, or does not change. The shift is implemented through the CatShift() function.

```
library thesis,ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use thesis.types.all, thesis.funcs.all;
use ieee.std_logic_arith.all; -- for unsigned declaration
use ieee.std_logic_misc.all; -- for reductions

entity issue is
   generic(w: integer);
   port(
      instr_q, cache_line: in instr_stream(w-1 downto 0);
      dep_free: in std_logic_vector(w-1 downto 0);
      stream_avail: in stream_srcs_available(w-1 downto 0);
      new_instr_q: out instr_stream(w-1 downto 0)
      );

   constant log_w: integer := log_table(w);
end issue;
```

```
architecture behav of issue is

   signal all_valid, issue_vector: std_logic_vector(w-1 downto 0);
   signal passes_check: std_logic_vector(w-1 downto 0);
   signal shift_amt: std_logic_vector(log_w-1 downto 0);
   signal ld: std_logic;
   signal encoder_input: std_logic_vector(w-1 downto 0);
   signal encoder_output: unsigned(log_w-1 downto 0);

begin

   -- check available bits for all register id's in instruction
   avail: for instr in all_valid'range generate
      all_valid(instr) <= and_reduce(stream_avail(instr));
   end generate;

   -- check instrs for dependencies, invalid regs, port enables
   passes_check <= dep_free and all_valid;

   -- form issue vector (i.e. 0011111)
   -- this vector tells if corresponding instruction allowed to issue

   issue_vector(0) <= passes_check(0);
   issue_vector(w-1 downto 1) <=
      passes_check(w-1 downto 1) and issue_vector(w-2 downto 0);

   -- get shift_amt by priority encode issue vector (00111110 => 5(dec) = 100)
   -- append '0' so that bit position of highest '1' is same as shift_amt
   -- encoder will return 0 if issue vector is 0

   encoder_input <= issue_vector(w-2 downto 0) & '0';
   encoder_output <= prio_encode(unsigned(encoder_input), log_w);
   shift_amt <= conv_std_logic_vector(encoder_output,log_w);

   -- use high bit of issue_vector to indicate a full load
   ld <= issue_vector(w-1);

   new_instr_q <=
         cache_line when (ld = '1')
      else
         CatShift(instr_q, shift_amt, cache_line);
end behav;
```

APPENDIX D. ONE_CNT_LASTN_EQ_C

The following is the VHDL code used to synthesize the one-counters needed for the register file port arbiter. The code is behavioral; to obtain properly synthesized designs, compilation directives are embedded within the code.

```
library ieee;
use ieee.std_logic_1164.all;

entity one_cnt_lastn_eq_c is
   generic(v,n,c: integer);
   port(vin: in std_logic_vector(v-1 downto 0); c_ones: out std_logic);
end;

architecture behav of one_cnt_lastn_eq_c is

-- pragma dc_script_begin
-- set_flatten -effort high
-- set_flatten true
-- set_structure false
-- pragma dc_script_end

begin
   process(vin)
      variable count : integer range 0 to n;
```

```
   begin
      count := 0;
      for i in 0 to n-1 loop
         if (vin(i) = '1') then
            count := count + 1;
         end if;
      end loop;

      if count = c then
         c_ones <= '1';
      else
         c_ones <= '0';
      end if;

   end process;
end behav;
```

APPENDIX E. TYPES PACKAGE

The following is the VHDL package *types*, which is referenced by many of the code modules written for this thesis. It defines constants and data types, including `instr_t`.

```
library ieee;
use ieee.std_logic_1164.all;

package types is

   constant reg_addr_size : integer := 6;
   constant opcode_size: integer := 6;
   constant regfile_id_size: integer := 2;

   constant num_read_ports: integer := 4;
   constant num_regs : integer := 32;
   constant num_srcs: integer := 2;
   constant num_dests: integer := 1;

   constant instr_size: integer := 32;

   type int_table_t is array(integer range <>) of integer;

   constant log_table: int_table_t(32 downto 0)  := (
      -1,0,1,2,2,3,3,3,3,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5
   );

   subtype reg_addr_t is std_logic_vector(reg_addr_size-1 downto 0);
```

```
   subtype opcode_t is std_logic_vector(opcode_size-1 downto 0);
   subtype regfile_id_t is std_logic_vector(regfile_id_size-1 downto 0);
   type src_array is array(num_srcs-1 downto 0) of reg_addr_t;
   type dest_array is array(num_dests-1 downto 0) of reg_addr_t;
   constant filler_size : integer :=
      instr_size - opcode_size - regfile_id_size
            - reg_addr_size*(num_srcs + num_dests);

   type instr_t is record
      opcode: opcode_t;
      regfile_id: regfile_id_t;
      srcs: src_array;
      dests: dest_array;
      filler: std_logic_vector(filler_size-1 downto 0);
   end record;

   type instr_stream is array(integer range <>) of instr_t;

   subtype instr_srcs_available is std_logic_vector(num_srcs-1 downto 0);
   type stream_srcs_available is array(integer range <>) of
         instr_srcs_available;

end types;
```

APPENDIX F. FUNCS PACKAGE

The following is the VHDL package *funcs*, which is referenced by many of the code modules written for this thesis. It defines three functions. `compare_fn()` is used in the dependence checker code; the remaining two are referenced in the instruction queue code.

```
library thesis,ieee;
use ieee.std_logic_1164.all, ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all, ieee.std_logic_arith.all;
use thesis.types.all;

package funcs is
   function compare_fn(a,b: in std_logic_vector) return std_logic;
   function prio_encode(a: in unsigned; m: in integer) return unsigned;
   function CatShift (Value: instr_stream; ShiftCtrl: std_logic_vector;
            ShiftInVal: instr_stream) return instr_stream;
end funcs;

package body funcs is

function compare_fn(a,b: in std_logic_vector) return std_logic is
begin
   if (a=b) then return '1';
   else return '0';
   end if;
end;
```

```
function prio_encode(a: in unsigned; m: in integer) return unsigned is
   variable i: integer;
   variable output: unsigned(m-1 downto 0);
begin

   if (conv_integer(unsigned(a)) = 0) then
      output := (others => '0');
   else
      for i in a'length-1 downto 0 loop
         if a(i) = '1' then
            output := conv_unsigned(i,m);
            exit;
         end if;
      end loop;
   end if;

   return output;
end prio_encode;

function CatShift (Value: instr_stream; ShiftCtrl: std_logic_vector;
           ShiftInVal: instr_stream) return instr_stream is
   variable retval: instr_stream(Value'range);
   variable i: integer;
begin

   -- may shift from Value'high to 0 positions
   -- the zero shift must be treated separately to avoid syntax errors

   for i in Value'range loop
      if i = ShiftCtrl then
         if i = 0 then
            retval := Value;
         else
            retval := ShiftInVal(i-1 downto 0) & Value(Value'high downto i);
         end if;
      end if;
   end loop;

   return retval;
end CatShift;

end funcs;
```

APPENDIX G. SYNTHESIZE

The following is a description of the c-shell script *synthesize*. The script takes a previously analyzed design from a VHDL design library, and invokes design compiler to obtain a standard cell-based design.

```
usage:   synthesize <design> -lib <library> [-arch <architecture>]
                    [-area] [-param <string>] [-timing] [-ungroup]

description:

   synthesize takes code that has been previously analyzed into a VHDL
   design library, and synthesizes it into a standard cell based design
   in Synopsys format. The resulting .db file is written to disk.  The
   user must provide the necessary parameters if the VHDL code is
   parameterizable (contains generics).

filesystem configuration:

   The following directory organization must exist before running the
scripts:
                      logical_library_name/
                  /         |          |      \
                db/       src/       eddm/   lib/
```

The VHDL source code must reside in src. The files produced by analyzing these files (.sim, .syn) must reside in lib. synthesize will elaborate and compile the code in lib and save the resulting .db dile to the db directory. make_layout will write results to the eddm directory.

options:

<design>
Specifies the name of the VHDL entity to synthesize.

-arch <architecture>
Specifies the architecture to attach to the entity. In Synopsys, the entity, architecture, and any instantiated entities and architectures must all reside in the same directory. Multiple architectures may reside there; if so, this will tell the compiler which one to use. If multiple architectures exist and none is specified, the most recently analyzed architecture (.mra) is used.

-area
Instructs Synopsys to report area information on the chip.

-lib <library>
Specifies the logical library name where the previously analyzed entity and architecture files (.sim and .syn) must be located. The mapping from library name to the physical filesystem directory should be specified in the user's ~/.synopsys_dc.setup file. This script requires that this mapping be of the form <library>/lib.

-param <string>
Specifies the parameters to be used for the VHDL generics. The format is the same as that for passing generics in Synopsys' 'elaborate' command. The string must be enclosed in double-quotes, and consists of generic-integer pairs separated by the equals sign. For example:

```
-param "w=2,s=2,d=1"
```

-timing
Instructs Synopsys to report timing information on the critical path through the design. The report specifies the delay in nanoseconds, and the cells on the path.

-ungroup
   Instructs Synopsys to flatten the design it produces. The .db file Synopsys first produces will contain hierarchy if the VHDL code makes explicit component instantiations. If this option is specified, Synopsys will then flatten the design, and the resulting .db file will contain no hierarchy. Design flattening sometimes makes easier the process of converting the .db file to a schematic and layout.

APPENDIX H. MAKE_LAYOUT

The following is a description of the c-shell script *make_layout*. The script takes a design in Synopsys db format, invokes db2eddm to convert it to a Mentor schematic, and invokes ICStation to create a layout of that schematic.

```
usage:  make_layout <design> -lib <library> -hier -lvs
                             -no_db2eddm -no_lay
description:

   make_layout first converts the design in the .db file to a standard
   cell based design in Mentor's eddm format. It then calls ICStation to
   convert this cell-based design to a VLSI layout.

filesystem configuration:

   same as for synthesize.

options:

   <design>

      Specifies the name of the .db file to convert to Mentor, then
      layout. The .db extension is assumed. This name will NOT be the
      same as the name used in the synthesize script if generics were
      used. In that case, the generics are appended to the name of the
     .db file.
```

```
-lib <library>

   Specifies a logical library name which is mapped in
   ~/.synopsys_dc.setup to an actual directory.  The script looks in
   <library>/db for the Synopsys design file specified.  The
   schematic and layout produced are both stored in <library>/eddm.

-hier

   Instructs the layout script that the design is hierarchical.
   Default is flat.

-lvs

   Instructs the layout to check its layout against the cell-based
   schematic for correctness. Default is off.

-no_db2eddm

   Instructs the script that the Mentor schematic already exists and
   thus db2eddm need not be run to produce it. Proceed straight to layout.

-no_lay

   Instructs the script to only run db2eddm and not to perform the
   layout. Only the cell-based schematic is produced.
```