PERFORMANCE AND COST ANALYSIS OF THE EXECUTION STAGE OF SUPERSCALAR MICROPROCESSORS

BY

DIMITRI CARL ARGYRES

B.S., University of California, Davis, 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 like to thank my advisor, Professor Wen-mei Hwu, for his guidance, support and belief in me. I would also like to thank John Gyllenhaal for his useful suggestions, guidance and help in the project. Thanks go to my colleague Matt Gavin who did a great deal of work in figuring out the Synopsys tools and how to convert designs to Mentor Graphics. I would like to thank Derek Cho and Roger Bringmann, who helped with the preliminary performance work, as well as the entire IMPACT group. Finally, I would like to thank my parents, Andreas and Joanne Argyres, for their encouragement, support, and love throughout this endeavor.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

1. INTRODUCTION

The goal of this thesis is to analyze performance and cost tradeoffs of the execution stage in superscalar or VLIW microprocessors. In such wide-issue processors, certain resources are seldom used. An example of this is the eighth branch unit of an eight-issue processor. This resource is used only if eight branch instructions are in the issue window, which is a rare occurrence. Relatively few functional units can thus achieve very good performance. The IMPACT C compiler, developed at the University of Illinois at Urbana-Champaign, is a versatile tool that generates code and does scheduling for several different processor types with a varying number of resources such as registers, functional units, and issue widths. With the scheduled code, a close approximation of the cycle count and, hence, relative performance of several different processor configurations is obtained. IMPACT can be used to find the point at which adding resources does not increase performance significantly.

In designing a microprocessor, it is necessary to balance the increasing hardware cost with the benefits of obtaining a higher performance. The performance/cost ratio should

be maximized, and a minimum performance goal should be set as a specification. Because the cost of a chip is directly proportional to its area, one way to determine the cost is to multiply the number of each component by its area, and then add the resulting areas. This cost method, which will be referred to as the simplex method, can be generated very quickly. The best resource configuration in terms of functional units for both four- and eight-issue superscalar processors was found using the simplex cost method and performance numbers generated from IMPACT.

The simplex cost method does not take into account the cost of routing block components together or the cost of extra multiplexors needed to choose the set of functional units that the instruction flows through. An attempt was made to improve the simplex cost method by including these factors. Instead of a pure mathematical formulation, the execution stage was isolated from the rest of the chip, and a tool was built which took a processor description and generated a layout for it. This layout includes the multiplexors and the routing cost and gives the area cost of the execution stage. This cost is then put in a mathematical formulation estimating the size of the entire chip. This tool, the *H*ardware *GEN*erator for the *EX*ecution unit (HGENEX), gives a nonfunctioning layout because it does not implement the functional units with working logic. It was designed to give an idea of the relative cost of various functional unit configurations in the execution stage.

Experiments using the HGENEX method were performed on a few resource configurations, and the results were compared to those for the simplex method and to a chip

micrograph of the PowerPC 604 to see if the tool obtained a realistic layout. One goal of these comparisons was to determine whether HGENEX has the potential to be a useful cost analysis tool. Another goal was to determine the problems associated with HGENEX to see if they could be eliminated in future versions of the tool. If such a tool could give a good cost estimate of an implementation, it would eventually be incorporated into the IMPACT framework to provide the architect the ability to determine the performance and cost benefits of various decisions. After careful analysis, it was discovered that more work is needed to expand and improve HGENEX to make this a reality.

This thesis is divided into ten chapters. Much of Chapter 2 was taken from [4] and gives an overview of the IMPACT compiler used to generate the performance numbers. Chapter 3 describes the parameters for the performance experiments and the methodology required to find the best configuration. In Chapter 4, the simplex cost method is described in detail, and the experimental results finding the best resource configuration are given in Chapter 5. Chapter 6 describes the HGENEX tool in detail, while Chapter 7 shows how to use it. In Chapter 8, the simplex method cost numbers are compared to HGENEX's numbers, and problems with the HGENEX method are evaluated. Chapter 9 describes future changes that would improve HGENEX, and Chapter 10 contains the conclusions.

2. OVERVIEW OF IMPACT

The IMPACT compiler is a generalized C and FORTRAN compiler that can generate optimized code for various architectures and machine resource configurations. It also implements new research code optimizations in order to analyze their effect on performance. Figure 2.1 shows a block diagram of the compiler. It is divided into three distinct parts, based on the intermediate code representation used in translation from C code to machine-specific assembly code. The highest level of intermediate representation is called *PCode* and is parallel-C code with intact loop constructs. Memory system optimizations, loop transformations, and general dependence analysis take place at this level. The next level of intermediate representation is called *Hcode*; it is a simplified form of C with only simple *if-then-else* and *goto* control flow constructs. Statement-level profiling, profile-guided code layout, and function inline expansion are performed at this level.

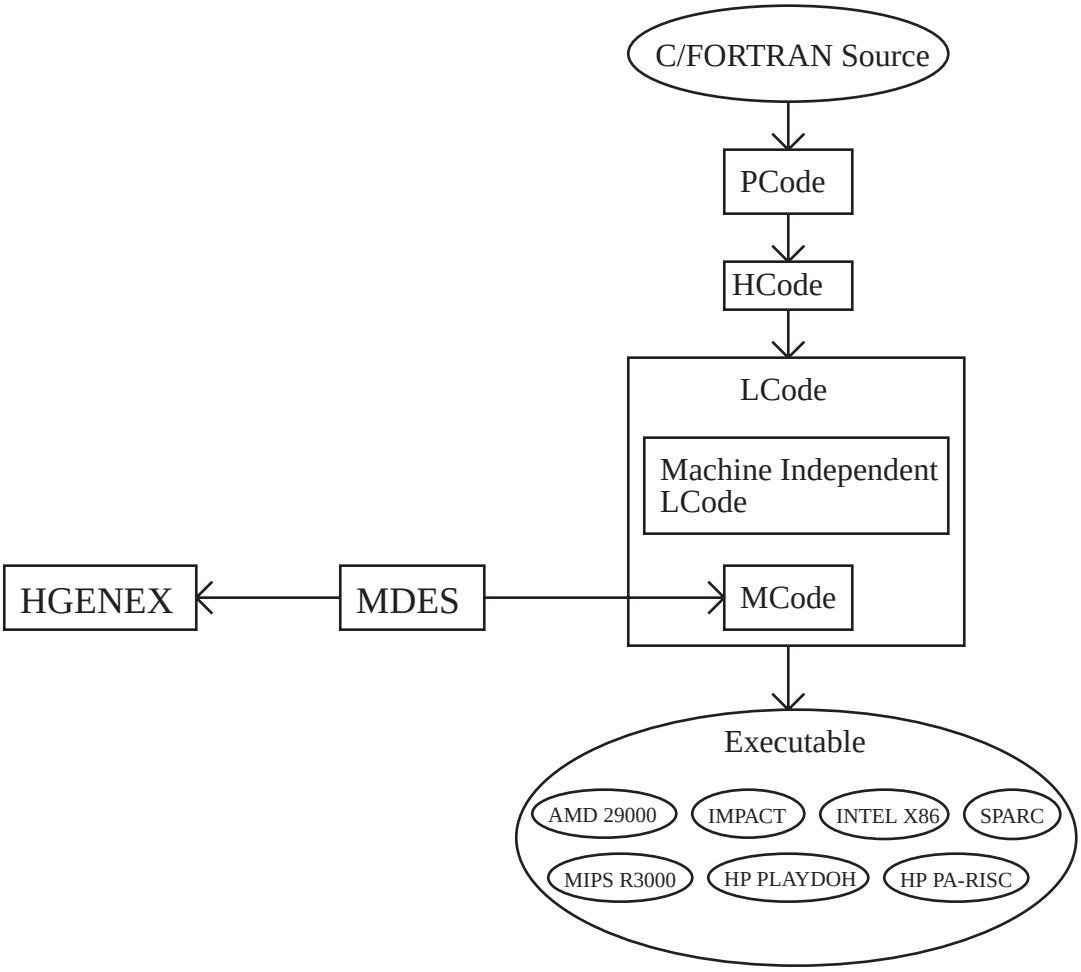


Figure 2.1: The IMPACT Compiler

The final and lowest code representation is called *Lcode*. Lcode is a generalized register transfer language. It is similar to most RISC instruction-based assembly language. Lcode is divided into two parts: machine independent Lcode and machine specific code known as *Mcode*. Machine independent Lcode instructions must eventually be mapped to Mcode before the compilation process is finished. At the machine independent Lcode level, several machine independent classic optimizations are performed. These include constant propagation, forward copy propagation, backward copy propagation, common subexpression elimination, redundant load elimination, redundant store elimination, strength reduction, constant folding, constant combining, operation folding, operation cancellation, code reordering, dead code removal, jump optimization, unreachable code elimination, loop invariant code removal, loop global variable migration, loop induction variable strength reduction, loop induction variable elimination, and loop induction variable reassociation. Additionly, interprocedural safety analysis is performed, which includes identifying safe instructions for speculation and function calls that do not modify memory.

Superblock and hyperblock compilation techniques are also performed at the Lcode level. Superblock support includes superblock formation using execution profile information, superblock classical optimization, and superblock instruction-level parallelism (ILP) optimization. If predicated execution support is available, then hyperblocks are used instead of superblocks. All superblock optimizations have been modified to operate

on hyperblocks. Furthermore, optimizations specific to hyperblocks exploiting predicated execution are available.

The major components of Mcode generation are the scheduler and the register allocator. Scheduling is performed using either acyclic global scheduling or software pipelining with modulo scheduling. Acyclic global scheduling is applied before register allocation (prepass scheduling) and after register allocation (postpass scheduling) to generate an efficient schedule for the code. Software pipelining is applied on certain loops identified at the Pcode level. Graph coloring based register allocation is utilized for all target architectures. For each target architecture, a set of specially tailored peephole optimizations are performed. These peephole optimizations are designed to remove inefficiencies during Lcode to Mcode conversion, take advantage of specialized opcodes available in the architecture, and remove inefficient code inserted by the register allocator. Execution profile information is also used if available.

IMPACT must compile code for a specific processor, which is described in great detail by the *M*achine *DES*cription (MDES) [1]. The MDES contains information on available functional units, size of the register files, instruction latencies, instruction input and output constraints, addressing modes, and pipeline constraints. The MDES is used by the optimization phases to determine if such transformations would be beneficial. The scheduler and register allocator rely on the MDES to generate correct and efficient code. The MDES is also used as an input to HGENEX, linking it with the rest of the IMPACT framework.

3. PERFORMANCE EXPERIMENTS

3.1 Introduction

In a VLIW or superscalar microprocessor, design of the execution stage is of utmost importance, because it consumes a large portion of the IC area. For most application programs, having as many of each type of functional unit as the issue width of the processor is wasteful, because it is very unlikely there will be so many instructions demanding a specific type of resource and being able to issue simultaneously. For this reason, one must decide how many of each type of functional unit are needed. Having more functional units reduces resource constraints and improves performance, but will also raise the chip's cost by increasing the number of transistors needed for implementation. At some point, there are diminishing returns when it is too expensive to increase performance by adding functional units. The goal of the following experiments was to find this point and maximize the performance/cost ratio in four- and eight-issue superscalar microprocessors.

3.2 Experiment

To obtain performance numbers, the IMPACT compiler framework was used to compile and schedule several integer and floating point benchmarks. The processor model used for these compilations was an in-order issue superscalar processor with one delay slot. The HP PA-RISC instruction set was assumed by the compiler and scheduler. Also included in the processor model was the assumption of 64 integer registers, 64 floating point registers, 32 double-precision floating point registers, and perfect instruction and data caches. Hence, cache and memory effects are ignored by these simulations. All instruction throughput is dependent only upon availability of resources to the scheduler and the parallelism in the instruction stream. The performance numbers were generated with scheduled code. Although more accurate and precise results could have been obtained with a simulation of the program, this would have been too time consuming.

Four- and eight-issue machines were examined. The machines had four different types of functional units: integer units (IALUs), floating point units (FALUs), load/store memory units, and branch units. Integer units executed all integer operations including adds, subtracts, multiplies, divides, shifts, rotates, compares, and logical operations; floating point units executed all floating point instructions; branch units executed all control flow instructions; and load/store units executed all memory operations. The machines also had fully pipelined units so that any unit could process a new instruction on each clock cycle. Instruction latencies of the HP PA-RISC 7100 processor were assumed in determining dependencies. These latencies are given in Table 3.1. All units were

completely independent of each other and each instruction used only one functional unit. Therefore, any n instructions that use different functional units and are not dependent on each other may execute simultaneously. Likewise, any n instructions that use the same resource and are not dependent on each other may execute simultaneously if there are n or more of that type of resource available. This is assuming a processor with an issue width greater than or equal to n.

Table 3.1: Instruction Latencies

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
memory load	2	FP multiply	2
memory store	1	FP divide (single-precision)	8
branch	1 / 1 slot	FP divide (double-precision)	15

The scheduler used in the experiments assumes the general speculation model, which is the most aggressive available. In this model, a potentially excepting instruction is allowed to be moved before a conditional branch, in which case it is replaced by a nonexcepting form of the instruction. An example of such a case is a division by zero, which would be avoided by taking a branch. The disadvantage of this model is that errors are hidden by the nonexcepting form of the instruction. If the branch were not taken, the code would not execute correctly, since the exception would not be handled. The restricted speculation model prevents such instructions from being moved before conditional branches so that such errors are avoided. The HP Precision Architecture is one of several architectures that provides nonexcepting instructions in support of the general speculation model, because

it gives an instruction schedule with the most parallelism possible. For this reason, the general speculation model was used.

The entire SPEC92 benchmark suite, as well as nine other commonly used non-numeric programs, was compiled and scheduled to obtain performance numbers for the experiment. This resulted in 29 total benchmarks, providing a realistic mix of actual processor applications. These benchmark programs are broken into integer and floating point benchmarks and are described in Tables 3.2 and 3.3. The performance and cost numbers obtained were put into a spreadsheet, which in turn calculated speedups and performance/cost ratios. Speedups were found by dividing the number of clocks cycles of the given configuration by the number of clock cycles of a machine with a minimum resource configuration. Such a machine has only one of each type of functional unit and has the minimum absolute cost and performance obtainable for a working superscalar machine of the given issue width. Cost numbers were also normalized to this minimum machine configuration.

3.3 Searching for the Best Configuration

For a four-issue processor with four types of functional units, there are $4^4 = 256$ resource configurations, and for an eight-issue processor with four types of functional units, there are $8^4 = 4096$ resource configurations. It would be infeasible to run IMPACT and generate a schedule for each permutation of resource configurations for all 29 benchmarks. Likewise, it would be infeasible to run a cost analysis tool for each possibility. Therefore,

Table 3.2: Floating Point Benchmarks

Floating Point Benchmarks	Benchmark Description
spice	analog circuit simulation
doduc	nuclear engineering monte carlo simulation
mdljdp2	quantum chemistry studying atom motion (double precision)
wave5	relativistic electromagnetic particle-in-cell simulation
tomcatv	vectorized mesh generation
ora	wave tracing through optical system
alvinn	autonomous land vehicle in a neural net
ear	medical simulation
mdljsp2	quantum chemistry studying atom motion (single precision)
swm256	shallow water model
su2cor	elementary particle masses using quark-gluon theory
hydro2d	galactic jets using hydrodynamical Navier Stokes equations
nasa7	NASA kernels
fpppp	performance simulation on the 2 electron integral derivative

to eliminate simulations that had low performance and high cost, a branch and bound algorithm was used to reduce the simulations to a reasonable number. For these experiments, the simplex cost method was used (see Chapter 4), and the spreadsheet directly calculated the cost numbers.

The first way to reduce the number of experiments was to find the performance of the complete configuration (i.e., n of each type of resource for an n-issue machine). This configuration has the best performance because it is never resource constrained. By applying this maximum possible performance number to all permutations of resource configurations, and given the cost numbers, an upper bound on the performance/cost ratio is achieved. For each actual simulation run, a true performance/cost ratio is found. All configurations whose upper bound on performance/cost is less than the largest true ratio may be discounted.

Table 3.3: Integer Benchmarks

Integer Benchmarks	Benchmark Description
espresso	truth table minimization
li	lisp interpreter
eqntott	boolean equation minimization
compress	compress files
sc	spreadsheet
cccp	GNU C preprocessor
cmp	compare files
eqn	format math formulas for troff
grep	string search
lex	lexical analyzer generator
qsort	quick sort
tbl	format tables for troff
wc	word count
yacc	parser generator

The second way to reduce the number of experiments is to place a new upper bound on those permutations with the same or fewer functional units of each type than that of an actual simulation. For example, if the {1234} configuration (this notation means 1 branch unit, 2 memory units, 3 IALUs, and 4 FALUs) was simulated, then all permutations with 1 branch unit, 2 or fewer memory units, 3 or fewer IALUs, and 4 or fewer FALUs have an upper bound on performance equal to that of the {1234} configuration. This would also mean that all such permutations have a new upper bound of performance/cost. If this new bound fell below that of some prior simulation, then the permutation could be eliminated. By proper choice of simulation runs, the number of required simulations was reduced from 256 to 32 for a four-issue machine and from 4096 to 52 for an eight-issue machine.

4. SIMPLE COST ANALYSIS

The simplest way of doing a cost analysis for the execution stage of a superscalar or VLIW architecture is to find an equation in which various parameters are input and the total area of the unit is output. This method, referred to as the simplex method in this thesis, may not be extremely accurate, but it may be close enough without going into the detail of building the entire chip. A possibly better way of performing cost analysis would be to build a very simple version of the chip, which would not have to be fully functional. This method, HGENEX, is described in detail in Chapter 6 and is not yet at the point of generating good cost numbers. Also, because of the great number of permutations possible for four- and eight-issue processors and the large amount of time required to run simulations using a better cost analysis model, it would not have been feasible to use HGENEX even if it proved highly superior to the simplex method. Thus, the simplex method was used to determine the best resource configurations with highest performance/cost ratios.

The basic model to determine the cost of an arbitrary resource configuration of a superscalar microprocessor is found by multiplying the number of each resource type with the chip area required by the resource. The area for the fetch, decode, and issue stages, the register files, the control logic, and the instruction and data caches is also added to obtain the total chip area. All areas for nonexecution stages are assumed to be constant with issue width, except for the fetch, decode, and issue stages, which are assumed to increase in area with the square of the issue width. The reason for this is that dependency checking is an $O(n^2)$ algorithm, so it is assumed that it will take $O(n^2)$ space on the chip, where n represents the issue width of the processor. Therefore, going from a four-issue machine to an eight-issue machine increases the chip area for the issue unit by a factor of four. The actual equations used for such cost calculation are given in Figure 4.1. The numbers generated are normalized with respect to a machine with a minimum configuration (one of each resource). Therefore, the final areas are relative to such a machine and are not the actual physical area in mm^2.

Ideally, since the HP PA-RISC instruction set was used, HP functional units would be used to determine the appropriate chip area for an implementation. This would give a more accurate idea of the actual size of such units, since these units are designed for the Precision Architecture's instruction set. Unfortunately, the HP functional units are not completely independent of each other. For example, the integer units perform branch address calculation [2]. Hence, there is no true branch unit. Also, the PA-7100 is a two-issue machine that can issue two instructions only if one is an integer and the other

Four-Issue:

$$Area = \frac{(1 - Ex_{\%})(Chip_{Area}^{PC604}) + B_i B_{Area} + M_i M_{Area} + I_i I_{Area} + F_i F_{Area}}{Chip_{Area}^{PC604}}$$

Eight-Issue:

$$Area = \frac{(1 - Ex_{\%})(Chip_{Area}^{PC604}) + 3 \times Issue_{Area} + B_i B_{Area} + M_i M_{Area} + I_i I_{Area} + F_i F_{Area}}{Chip_{Area}^{PC604} + 3 \times Issue_{Area}}$$

$Chip_{Area}^{PC604}$ = Area of the PowerPC 604 in mm^2
B_i = Number of Branch Units
M_i = Number of Memory Units
I_i = Number of Integer Units
F_i = Number of Floating Point Units
B_{Area} = Area for a Branch Unit in mm^2
M_{Area} = Area for a Memory Unit in mm^2
I_{Area} = Area for a Integer Unit in mm^2
F_{Area} = Area for a Floating Point Unit in mm^2
$Issue_{Area}$ = Area for an Issue Unit in mm^2 for a Four-Issue Machine
$Ex_{\%}$ = Percent of Chip Area for Execution Unit

Figure 4.1: Equations for Simple Cost Analysis

is a floating point instruction. These abnormalities make the PA-7100 a poor choice for finding areas for various functional unit areas.

Several architectures were examined to find independent branch, memory, integer, and floating point units. The architecture that most closely resembled the processor description used in the performance analysis was the PowerPC 604 [3]. Using the chip micrograph, each of these units was measured with a ruler. Although this method introduces some inaccuracies, it was the best given the limited information on the processor implementaion. The sizes found are given in Table 4.1. These sizes were used in both the simplex and HGENEX methods for cost analysis.

Table 4.1: PowerPC 604 Parameters

- $B_{Area} = 12.51mm^2 = 12,510,000\lambda^2$
- $M_{Area} = 11.26mm^2 = 11,260,000\lambda^2$
- $I_{Area} = 14.95mm^2 = 14,950,000\lambda^2$
- $F_{Area} = 20.50mm^2 = 20,500,000\lambda^2$
- $Chip_{Area}^{PC604} = 195.92mm^2 = 195,920,000\lambda^2$
- $Issue_{Area} = 24.03mm^2 = 24,030,000\lambda^2$
- $Ex_{\%} = 30.2\%$

5. EXPERIMENTAL RESULTS

5.1 Introduction

Using the search methods described in Chapter 3, the entire set of processor configurations was examined and the best performance/cost configurations were found. There were 29 cost/performance numbers for each resource configuration simulated, one for each benchmark run. Instead of averaging the performance of the entire set of benchmarks and obtaining a single performance/cost ratio, the benchmarks were broken into two different groups: integer and floating point benchmarks. The integer benchmarks use almost no floating point instructions. Hence, these benchmarks would tend to eliminate all floating point units, because they would not help performance and would increase cost. Thus, the results would be skewed based on the ratio of the number of integer to floating point benchmarks used in the experiment. To eliminate this problem, the benchmarks were grouped so that performance for integer and floating point code would not be compared. Two resource configurations with the best performance/cost ratios were obtained. The

first was for integer applications and the second for scientific floating point intensive applications. All cost numbers were obtained using the simplex cost method.

Also included in the next sections are the percentages of maximum possible performance achieved with the given configurations. The maximum performance is obtained with the {4444} configuration. Both the performance/cost ratio and the percentage of maximum performance are very important to a designer. A configuration may give an excellent ratio, but if its maximum performance percentage is low, then the inexpensive machine will have a poor performance. Thus, the percentage of maximum performance should be bounded when choosing a configuration, so that a minimum performance could be achieved. In a commercial system, this bound would be enforced by a minimum target performance given in the specification.

5.2 Four-Issue Performance

Table 5.1 shows the simulations with the best performance/cost cost ratios for a four-issue machine. The configuration with the best ratio for the integer benchmarks is {2221}, and the configuration with the best ratio for the floating point benchmarks is {1211}. The notation {4321} means 4 branch units, 3 memory units, 2 integer ALUs, and 1 floating point ALU. Both configurations are highlighted in bold in the table. While these two cases are indeed the best ratios for a four-issue machine, the other configurations included in Table 5.1 do not necessarily have the next best performance/cost ratios of all possible configurations because an exhaustive search was not done. Other

configurations may have similar performance/cost ratios, but they were not simulated. The configurations in Table 5.1 were the best of the small number of simulations done. Since their performance/cost ratios are close to that of the optimal configuration, these ratios give an idea of the best configurations possible. Most configurations with high ratios for integer benchmarks had a low floating point performance/cost ratio, but had good absolute floating point performance numbers. The configurations with high ratios for floating point benchmarks had low integer performance/cost ratios and poor absolute integer performance numbers.

Table 5.1: Performance and Cost for a Four-Issue Machine

Resource Configuration				Percent of Maximum Performance		Performance/Cost Ratio	
Branch	Memory	IALU	FALU	Integer	Floating Point	Integer	Floating Point
2	**2**	**2**	**1**	**85.1%**	**87.7%**	**1.38**	**1.01**
3	2	2	1	89.2%	87.7%	1.37	0.96
3	3	2	1	92.3%	89.1%	1.36	0.93
3	2	3	1	92.8%	89.1%	1.35	0.92
4	2	2	1	91.8%	87.7%	1.34	0.91

Best Configurations for Integer Benchmarks

1	**2**	**1**	**1**	**56.7%**	**81.9%**	**1.04**	**1.07**
1	2	2	1	69.6%	87.0%	1.19	1.06
1	2	2	2	69.6%	94.9%	1.09	1.06
2	2	2	1	85.1%	87.7%	1.38	1.01

Best Configurations for Floating Point Benchmarks

While the {1211} resource configuration achieved the best performance/cost ratio for floating point benchmarks, it had a very poor integer performance and a mediocre floating point performance. Hence, it would not be a good choice in designing a high-performance microprocessor. With a minimum bound of 85% on the maximum performance percentage, the {2221} configuration meets performance goals, has the best performance/cost ratio for integer benchmarks, and has a fairly high ratio for floating point benchmarks.

Therefore, it would be a much better choice for a high-performance processor. If the machine's purpose was for floating point applications only, then the best choice would be the {1222} configuration because it has a high floating point performance and achieves a near-optimal performance/cost ratio.

5.3 Eight-Issue Performance

Table 5.2 shows the simulations with the best performance/cost ratios for an eight-issue machine. The {4431} configuration achieved the best ratio for integer benchmarks and is highlighted in bold in the table. This configuration had 88.5% of the maximum performance for integer benchmarks and 81.6% for floating point benchmarks. Several other resource configurations achieved near-optimal performance/cost ratios for integer benchmarks. The {5431} configuration, for example, had a slightly lower ratio, but had a higher percentage of maximum performance for integer benchmarks. Both configurations would be good choices for designing a high-performance microprocessor. While the absolute floating point performance is around 80% for all high ratio configurations running integer code, the performance/cost ratio is very low for floating point benchmarks. The results also show that branch units are more important than other kinds of functional units for integer code, because more of them are needed on average in the optimal configurations.

The {2322} configuration had the best performance/cost ratio for floating point benchmarks and is also highlighted in bold in Table 5.2. This configuration had an

Table 5.2: Performance and Cost for an Eight-Issue Machine

Resource Configuration				Percent of Maximum Performance		Performance/Cost Ratio	
Branch	Memory	IALU	FALU	Integer	Floating Point	Integer	Floating Point
4	**4**	**3**	**1**	**88.5%**	**81.6%**	**1.63**	**0.97**
4	3	3	1	85.8%	80.4%	1.62	0.98
5	3	3	1	87.7%	80.4%	1.61	0.95
5	4	3	1	93.7%	81.6%	1.61	0.93
4	3	4	1	87.7%	81.0%	1.60	0.95
4	4	4	1	90.9%	81.6%	1.60	0.93
Best Configurations for Integer Benchmarks							
2	**3**	**2**	**2**	**70.0%**	**92.0%**	**1.40**	**1.19**
1	2	2	2	53.8%	85.9%	1.16	1.19
1	3	2	2	55.3%	88.3%	1.15	1.18
2	2	2	2	66.8%	87.1%	1.38	1.16
1	4	2	2	56.1%	89.6%	1.13	1.16
2	3	3	2	75.5%	92.6%	1.45	1.15
Best Configurations for Floating Point Benchmarks							

absolute performance of 92.0% of the maximum possible performance for floating point benchmarks and 70.0% for integer benchmarks. The {2332} configuration had a slightly lower performance/cost ratio for floating point benchmarks, but it had a higher ratio for integer code and had better absolute performance than the {2322} configuration. The remaining configurations had very poor absolute integer performance. Since configurations {2332} and {2322} had poor integer performance and poor performance/cost ratios for integer benchmarks, they would not be good choices for a general purpose microprocessor. They would, however, be good choices for a processor running primarily scientific code.

In summary, for an eight-issue machine, the {4431} and {5431} resource configurations had the best performance/cost ratios and absolute performance for integer benchmarks. The {2322} and {2332} configurations had the best absolute performance and performance/cost ratios for floating point benchmarks. Unlike a four-issue machine,

finding a resource configuration for an eight-issue machine that gives good integer and floating point performance/cost ratios and absolute performance numbers is impossible. Thus, in choosing the resource configuration of a processor the applications being run must be considered. Processors built for integer code will not run floating point code efficiently because the code will need more floating point units, and the other units will not be fully utilized. Processors built for floating point code will not fully utilize the floating point units and will be lacking other functional units when running integer code.

6. THE HGENEX TOOL

6.1 Introduction

Chapter 4 describes a simple way to estimate the cost of a superscalar or VLIW machine. The areas of the functional units are multiplied by the number of such units, and these areas are then summed with the area for the fetch, issue, and decode stages, and the control and miscellaneous periphery logic. This final area in mm^2 is then divided by the area of a minimum configuration (one for each functional unit) to achieve a final area ratio. Such a method does not include the cost of interconnection or the extra components required to switch between functional units used.

To take these considerations into account, the HGENEX tool was created to estimate chip area by creating a layout instead of using a purely mathematical formulation. Working logic was not implemented for the functional units and only the execution stage was considered for layout. To obtain total chip area, the layout area was included in a mathematical formula that summed areas for all top-level components. HGENEX starts

with a *M*achine *DES*cription (MDES) containing enough information for a compiler, such as IMPACT, to create an instruction schedule. From this MDES, HGENEX determines how functional units are connected and creates structural VHDL code. Then, either Synopsys Design Analyzer or Mentor Graphics Autologic takes the VHDL files to a schematic. Using Mentor Graphics DVE, a design viewpoint is created to be used in Mentor Graphics layout tool, IC Station. Three phases are needed to achieve the translation from MDES to layout: VHDL synthesis, schematic synthesis, and layout synthesis. Figure 6.1 illustrates the complete process of this translation.

6.2 VHDL Synthesis

The first stage of synthesis is to generate synthesizable VHDL files to be used by other commercial CAD tool systems to create a schematic and then a layout of the execution stage. To begin, the high-level MDES (HMDES) is translated into a more useful form, the low-level MDES (LMDES). This form can be used by other systems, such as IMPACT. Information on such a translation and on using an MDES in general is found in [1]. HGENEX calls a function that will load the LMDES information into a data structure. HGENEX then takes this information, processes it, and builds its own data structure to specify how certain resources are connected to each other. Two VHDL files are generated: one that defines various entities or resources used, and another that defines how they are connected. These VHDL files synthesize into a schematic using either the Mentor Graphics Autologic or the Synopsys Design Analyzer synthesis tool.

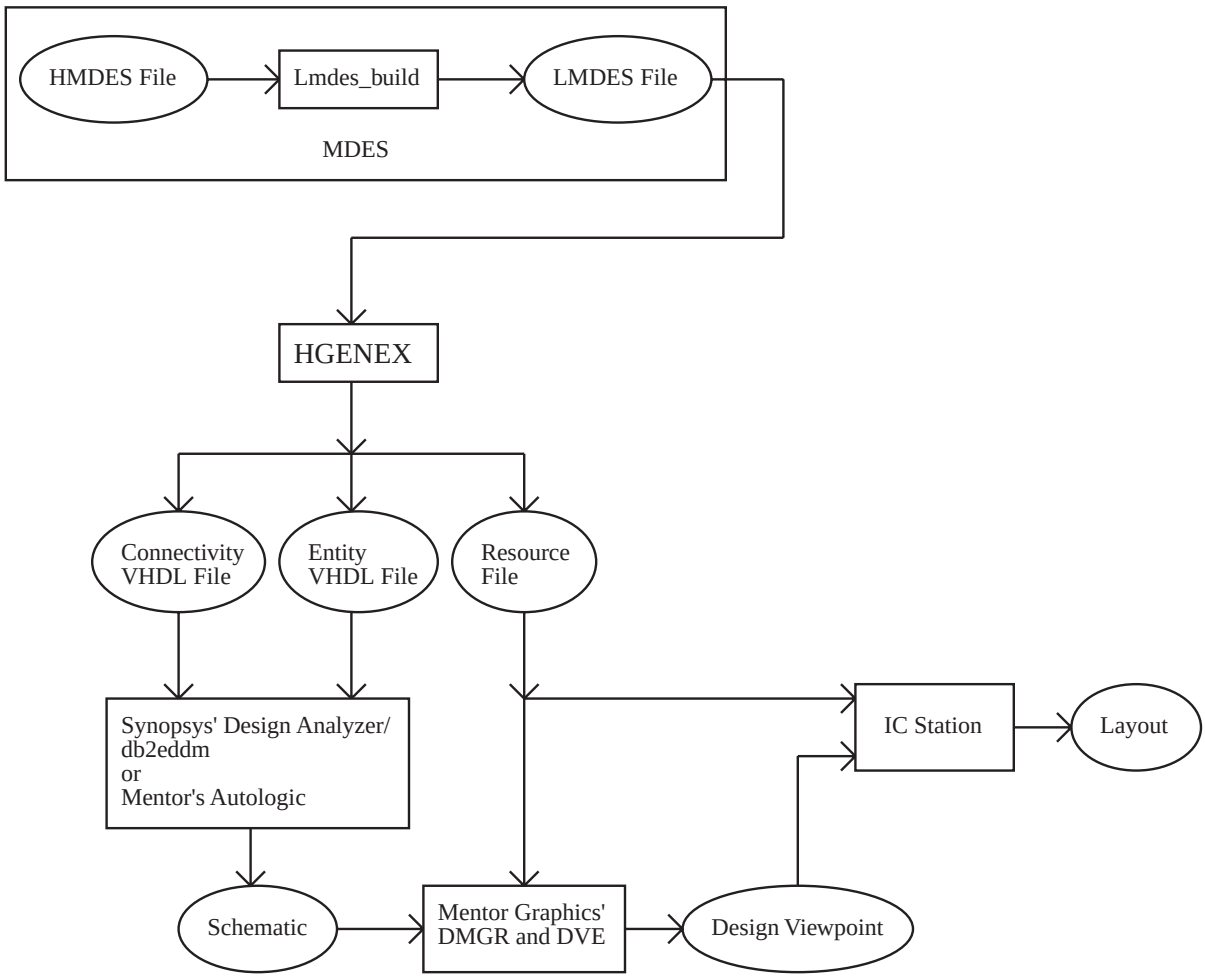


Figure 6.1: Complete Process: MDES to Layout

A third file, called the *resource file*, is also generated for layout synthesis. In addition to generating these files, HGENEX also outputs the number and type of extra multiplexors needed, as well as the number of wire interconnections required in the design. This gives an idea of the relative costs involved between different designs and also gives an idea of the run time requirement of the schematic and layout phases. Figure 6.2 shows an example output of the HGENEX tool when running the {1111} configuration for a four-issue machine.

```
Done building Data Structures

Finished Extracting Resource Sizes from Database

Wire Count:  896

Multiplexor Count:
64 Bit Mux 2 X 4:  5

Finished Building VHDL Files
```

Figure 6.2: Example of HGENEX Output

The HGENEX program uses the *Resources* and *ResTables* sections of the MDES. The Resources section defines every resource used by IMPACT when the MDES is used for scheduling and simulation. The number of each type of resource is also defined. HGENEX can ignore any resource that the user decides is not an actual piece of hardware and is used only for scheduling purposes, such as modeling interlocks. The ResTables section defines the clock cycle at which a resource will be used in the execution stage of the

pipeline. This information helps determine how resources are connected. An example of the HMDES Resources and ResTables sections is shown in Figure 6.3. This HMDES will be used to demonstrate what the HGENEX tool does and how a layout is finally generated. A more complicated example is given in the appendices.

The MDES ResTables section is broken into *instruction groupings*. Each instruction grouping is simply a group of several different instructions that use the same resources at the same clock cycle when they execute. Therefore, they flow through the same path in the execution stage. HGENEX looks at each instruction grouping separately; it uses the simple algorithm that any resource being used on clock cycle n connects to all resources being used on clock cycle $n + 1$. All connections are stored, and the final resource connections are the union of all of the connections for each resource for all instruction groupings.

Every resource does not have to be used in a single clock cycle. Resources can take several clock cycles, and different instruction groupings may use resources for different amounts of time. To handle these cases, every resource is broken into resource stages. See Figure 6.4 for an illustration of this idea. Each of these stages takes a single clock cycle for the operands to traverse through it. For example, in Figure 6.4, resource A is used for four clock cycles and resource B is used on the third cycle by an instruction. There should be a connection from resource A to resource B so that the instruction's operands can go to both resources A and B on the third cycle. Without breaking the resources into stages, the connectivity cannot be defined because resource A is a complete unit that has

```
# Enumerate resources
(Resources declaration
    slot[0..3]
    Hialu_0
    Hfalu_0
    Hfalu_0
    Hmem_0
    Hbranch
    out
end)

(ResTables declaration
    RL_IAlu    ( # integer alu op
                        (slot 0)
                        (Hialu_0 0)
                        (out 1)
               )
    RL_IBr     ( # control transfer instr
                        (slot 0)
                        (Hbranch 0)
                        (out 1)
               )
    RL_FPAlu   ( # single/double precision fp alu op
                        (slot 0)
                        (Hfalu_0 0)
                        (out 1)
               )
    RL_Load    ( # load instruction
                        (slot 0)
                        (Hmem_0 0)
                        (out 1)
               )
    RL_Store   ( # store instruction
                        (slot 0)
                        (Hmem_0  0)
                        (out 1)
               )
end)
```

Figure 6.3: Example HMDES Resources and Reservation Table Declarations

only one input and output. With stages, the connectivity can easily be specified. Some instructions may use a resource for several cycles and simply flow multiple times through the same hardware. A division unit is such an example. While breaking the unit into stages allows for fully pipelined resources, it will greatly replicate hardware, thus giving a very pessimistic area estimation. This problem can be overcome by specifying smaller area sizes for each stage (see Section 6.4) so that when all of the areas for these stages are summed, the final area will correspond to the actual area of the resource in question.

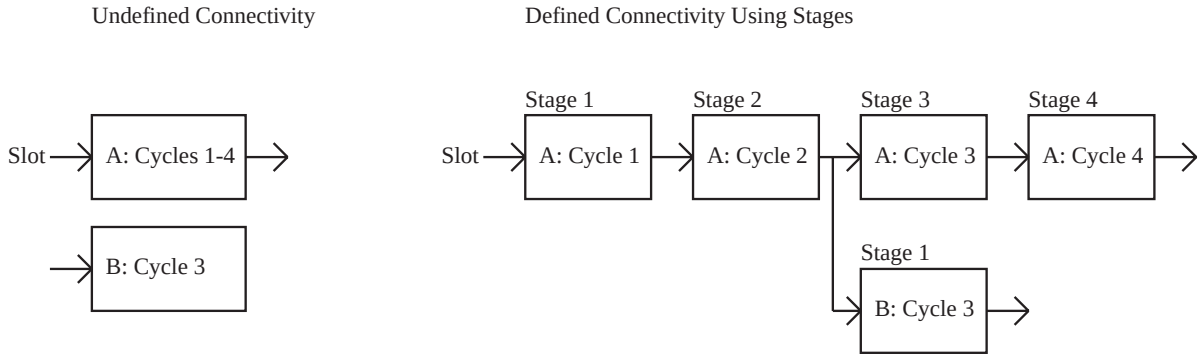


Figure 6.4: Breaking Resources into Stages

The simple algorithm of connecting all resources used on cycle n to each resource used on cycle $n + 1$ is slightly pessimistic, since there could be an instruction that uses two different paths through the execution stage simultaneously. Presently, each stage between these two paths would be cross connected. For example, for two given resources A and B, each with two stages, HGENEX will connect stage 1 of resource A to stage 2 of both resources A and B, as well as stage 1 of resource B to stage 2 of both resources A and B. This is illustrated in Figure 6.5. These two paths may not ever have to exchange information with each other, and hence, the extra connections are unnecessary

and wasteful. Unfortunately, the present MDES is limiting in that exact connectivity cannot be specified and a general algorithm must be used. Since some resources may exchange information, it is better to be pessimistic and assume complete connectivity, even though some connections might not be needed. At present, HGENEX also does this for multiple resources of the same type. This allows instructions to change from resource to resource of the same type within the execution stage. Although this could be useful for extremely complex datapath design, it is generally not needed and gives an overly pessimistic view of the connections required. Future versions of HGENEX should remove this feature.

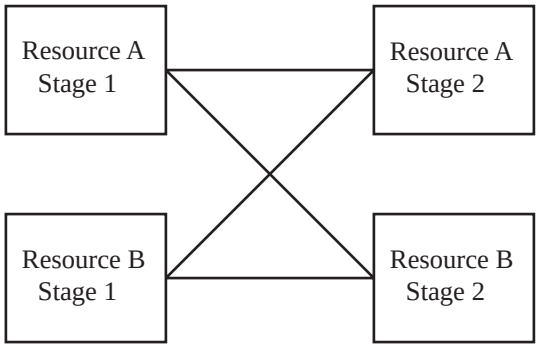


Figure 6.5: Cross Connecting for Multiple Paths Through the Datapath

All resources have only one input and one output and thus only process the data given to them. Switching between inputs is done with the use of external multiplexors. If there is only one resource to connect, it is connected directly with a bus. If there are several resources needing to connect, an $m \times n$ multiplexor is placed at the resource's input. m represents the number of select lines of the multiplexor and n the number inputs, so that $n = 2^m$. Each of the inputs are really buses of the appropriate size. The smallest possible

multiplexor is used, and the excess lines are tied to ground. Multiplexor control lines are defined and are present in the schematic.

All operands are assumed to have been decoded. Each bus line contains all of the operands that a resource will need. Hence, a machine that is 32 bits wide and whose arithmetic logic units will operate on two 32-bit operands should have 64-bit bus widths. This is specified in a separate parameter file (see Chapter 7 for details). The execution stage has global inputs called *slots* that connect it to the issue unit in the previous pipeline stage. The slots contain all of the operands needed for execution. Every path through the datapath connects to a slot. Also, there are global outputs called *out* buses that contain the processed data. All of the possible paths of the datapath connect to one or more out buses. HGENEX can handle multiple input slots or out buses.

HGENEX generates three files from scratch. Two of these files contain VHDL code, while the third assists the layout tool. The first VHDL file, called the *entity file*, puts a dummy behavioral description for each of the functional units, defining each of these components for the synthesis tool. Also, this file contains a synthesizable behavioral description of the needed multiplexors. The second VHDL file, called the *connectivity file*, is a structural description of how the basic components defined by the entity file are connected. Creating this file is the main purpose of HGENEX. The third file created is a resource file used by IC Station and other Mentor Graphics tools in preparation for layout. This file contains all of the cells to be created and the sizes of all of the functional units to be used in the design. These sizes are obtained from a *database file* that must

be created by the user. All sizes are specified in mm^2. Chapter 7 contains details of the actual use of HGENEX. Figure 6.6 shows the entity file; Figure 6.7 shows the VHDL file; and Figure 6.8 shows the resource file for the example HMDES given in Figure 6.3.

6.3 Schematic Synthesis

The VHDL files generated by HGENEX can be used in a synthesis tool to obtain a schematic. Two tools that HGENEX supports are Synopsys Design Analyzer and Mentor Graphics Autologic. However, the VHDL file that each tool will need is slightly different. Hence, HGENEX needs to know the tool for which it is creating VHDL code. This is specified along with other parameters in the parameter file. If Autologic is used to synthesize schematics, a schematic ready for layout will be produced. A script can be written to automate this part of the design. Unfortunately, Autologic does not synthesize as well as the Synopsys tools do, nor does it support various VHDL constructs that would be necessary if the functional units were to be implemented with working logic. Hence, a script automating the schematic synthesis using Autologic was not written.

The Synopsys Design Analyzer provides a better logic design for the multiplexors and allows for more control of the logic being created. Unfortunately, the schematics generated are not compatible with the Mentor Graphics layout tool. Therefore, a translation tool, called db2eddm, is used to translate the schematics to the Mentor Graphics format, eddm. A script was written to automate this process. This script used default options in Synopsys to generate the schematics. These options include *medium effort* by the design

```
ENTITY mux2S_64B IS
   PORT (i0, i1, i2, i3: IN BIT_VECTOR(63 DOWNTO 0);
        sel: IN BIT_VECTOR(1 DOWNTO 0); o: OUT BIT_VECTOR(63 DOWNTO 0));
END mux2S_64B;
ARCHITECTURE rtl OF mux2S_64B IS BEGIN
   WITH sel SELECT o <=
                        i0 WHEN "00",
                        i1 WHEN "01",
                        i2 WHEN "10",
                        i3 WHEN OTHERS;
END rtl;

ENTITY Hialu_0_0_S1 IS
  PORT (i: IN BIT_VECTOR(63 DOWNTO 0); o: OUT BIT_VECTOR(63 DOWNTO 0));
END Hialu_0_0_S1;
ARCHITECTURE behavior OF Hialu_0_0_S1 IS BEGIN
o <= NOT i;
END behavior;

ENTITY Hbranch_0_S1 IS
  PORT (i: IN BIT_VECTOR(63 DOWNTO 0); o: OUT BIT_VECTOR(63 DOWNTO 0));
END Hbranch_0_S1;
ARCHITECTURE behavior OF Hbranch_0_S1 IS BEGIN
o <= NOT i;
END behavior;

ENTITY Hfalu_0_0_S1 IS
  PORT (i: IN BIT_VECTOR(63 DOWNTO 0); o: OUT BIT_VECTOR(63 DOWNTO 0));
END Hfalu_0_0_S1;
ARCHITECTURE behavior OF Hfalu_0_0_S1 IS BEGIN
o <= NOT i;
END behavior;

ENTITY Hmem_0_0_S1 IS
  PORT (i: IN BIT_VECTOR(63 DOWNTO 0); o: OUT BIT_VECTOR(63 DOWNTO 0));
END Hmem_0_0_S1;
ARCHITECTURE behavior OF Hmem_0_0_S1 IS BEGIN
o <= NOT i;
END behavior;
```

Figure 6.6: Example Entity VHDL File

```
ENTITY execute IS
  PORT(slot_in0,slot_in1,slot_in2,slot_in3: IN BIT_VECTOR(63 DOWNTO 0);
      icp0, icp1, icp2, icp3, icp4, icp5, icp6, icp7, icp8, icp9: IN BIT;
      output0: OUT BIT_VECTOR(63 DOWNTO 0));
END execute;
ARCHITECTURE structure OF execute IS
COMPONENT Hialu_0_0_S1
   PORT(i: IN BIT_VECTOR(63 DOWNTO 0); o: OUT BIT_VECTOR(63 DOWNTO 0));
END COMPONENT;
COMPONENT Hbranch_0_S1
   PORT(i: IN BIT_VECTOR(63 DOWNTO 0); o: OUT BIT_VECTOR(63 DOWNTO 0));
END COMPONENT;
COMPONENT Hfalu_0_0_S1
   PORT(i: IN BIT_VECTOR(63 DOWNTO 0); o: OUT BIT_VECTOR(63 DOWNTO 0));
END COMPONENT;
COMPONENT Hmem_0_0_S1
   PORT(i: IN BIT_VECTOR(63 DOWNTO 0); o: OUT BIT_VECTOR(63 DOWNTO 0));
END COMPONENT;
COMPONENT mux2S_64B
   PORT(i0, i1, i2, i3: IN BIT_VECTOR(63 DOWNTO 0);
       sel: IN BIT_VECTOR(1 DOWNTO 0); o: OUT BIT_VECTOR(63 DOWNTO 0));
END COMPONENT;
SIGNAL GROUND, sig4, sig5, sig6, sig7, sig8, sig9, sig10, sig11:
       BIT_VECTOR(63 DOWNTO 0);
BEGIN
M0 : mux2S_64B PORT MAP (i0 => slot_in0, i1 => slot_in1, i2 => slot_in2,
     i3 => slot_in3, sel(1) => icp0, sel(0) => icp1,o => sig4);
C0 : Hialu_0_0_S1 PORT MAP (i => sig4, o => sig5);
M1 : mux2S_64B PORT MAP (i0 => slot_in0, i1 => slot_in1, i2 => slot_in2,
     i3 => slot_in3, sel(1) => icp2, sel(0) => icp3, o => sig6);
C1 : Hbranch_0_S1 PORT MAP (i => sig6, o => sig7);
M2 : mux2S_64B PORT MAP (i0 => slot_in0, i1 => slot_in1, i2 => slot_in2,
     i3 => slot_in3, sel(1) => icp4, sel(0) => icp5, o => sig8);
C2 : Hfalu_0_0_S1 PORT MAP (i => sig8, o => sig9);
M3 : mux2S_64B PORT MAP (i0 => slot_in0, i1 => slot_in1, i2 => slot_in2,
     i3 => slot_in3, sel(1) => icp6, sel(0) => icp7, o => sig10);
C3 : Hmem_0_0_S1 PORT MAP (i => sig10, o => sig11);
M4 : mux2S_64B PORT MAP (i0 => sig5, i1 => sig7, i2 => sig9, i3 => sig11,
     sel(1) => icp8, sel(0) => icp9, o => output0);
END structure;
```

Figure 6.7: Example Connectivity VHDL File

```
"buswidth"      64

"Hialu_0_0_S1"  14.950000
"Hbranch_0_S1"  12.510000
"Hfalu_0_0_S1"  20.500000
"Hmem_0_0_S1"   11.260000
"end"
"mux2S_64B_0"
"mux2S_64B_1"
"mux2S_64B_2"
"mux2S_64B_3"
"mux2S_64B_4"
"final_end"
```

Figure 6.8: Example Resource File

compiler and a *balanced logic depth*. Once the schematics are generated, the design is ready for layout. Figure 6.9 shows the schematic of the synthesized VHDL code given in Figures 6.6 and 6.7

6.4 Layout Synthesis

The layout phase uses the schematics generated in the previous phases and the resource file generated by HGENEX and produces a complete layout of the design. Before actual layout, some preliminary steps are taken to ensure proper functioning of the layout tool. Here, any previous cells or design viewpoints are deleted using Mentor Graphics Design Manager (DMGR). Next, a design viewpoint is created for the components used in the design. The Mentor tool Design Viewpoint Editor (DVE) is used for this purpose. The design viewpoint contains the technology information that the design will be mapped to and also a pointer to the logic schematic. All layouts were created using the MOSIS

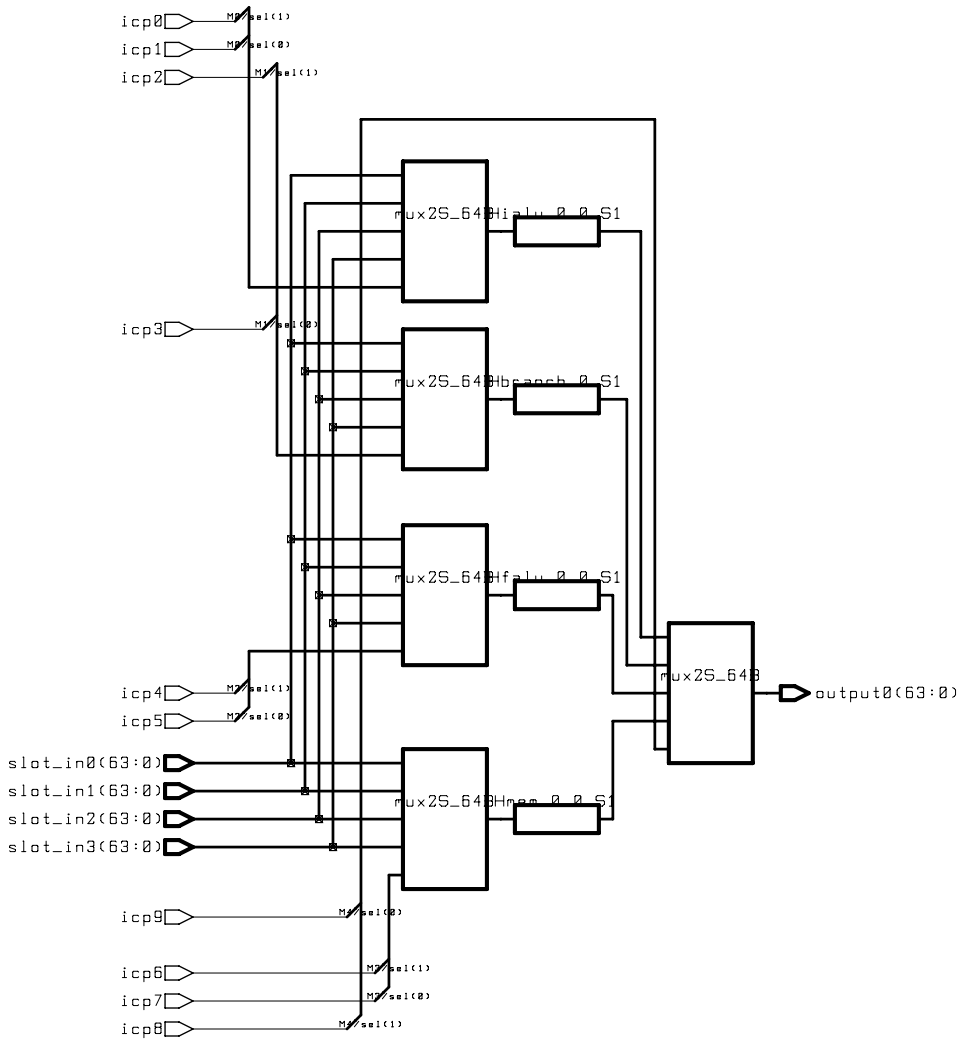


Figure 6.9: Schematic of Example Design

1.2 micron, two-level metal process. Areas obtained from HGENEX used units of λ^2 to be process independent.

The Mentor Graphics tool IC Station does the actual layout. Each multiplexor has a full logic schematic. This schematic goes through the layout tools, and a layout representing the schematic's logic is created from the MOSIS cmosn standard cell library. The following steps are used to make the layout for each multiplexor and the final execute stage of the pipeline: Load Design Rules, Autofloorplan, Autoplace Standard Cells, Autoroute All, Minimize Vias, Left Compaction, and Downward Compaction. Each of these functions is executed with the default options except for the compaction. Only routing channels were compacted, so that the routing and the instantiated cells would not overlap, thereby shorting wires.

The functional units have specified sizes and are not implemented in logic. Hence, a rectangle of the appropriate size is generated and placed in both the first level of metal (*metal1*) and the polysilicon (*poly*) layers. Ports are then placed on the left and right sides for the in and out ports, respectively. They are placed in the second level of metal (*metal2*) layer and are evenly spaced across the entire height of the cell. Power and ground ports are also included and are placed in the metal1 layer. Hence, a dummy layout cell is used for the functional units. However, the size of the cell corresponds to the area specified in the database file. Thus, despite the fact that the layout is not a functioning chip, a good estimation of area is generated. A script automates this

entire process. Figure 6.10 shows the layout for the example. The area turns out to be $296,811,392\lambda^2$.

Figure 6.10: Layout of Example Design

7. USING HGENEX

7.1 Introduction

This chapter describes how to use the HGENEX tool. The HMDES must be slightly modified to make the tool function properly. These modifications are relatively minor. A parameter file must be present to give HGENEX needed information. A database file must also be created to specify sizes of the various functional units. Finally, certain environment variables, files, and directory structures must be set up so that the scripts in the schematic and layout phases will run properly.

7.2 MDES Alterations

The LMDES file must be built from its corresponding HMDES file before HGENEX can run. This file must also be placed in the same directory as the HGENEX executable

file. The name of the LMDES file is specified in the HGEN_PARMS parameter file described in Section 7.3. To build the LMDES file, the function *Lmdes_build* is executed at the UNIX cshell prompt: *Lmdes_build* {*HMDES filename*} {*LMDES filename to build*}.

The first change needed for the MDES is that a resource called "out" must be specified in the Resource section of the HMDES file. The name is fixed and must be *out* just like the *slot* resource. The out resource is the output or result bus of the execution stage. The HGENEX tool uses this information to create output ports in the schematic and layout. Having one simple output bus or several output buses in which different paths through the execution stage go to different output buses is possible. The same path may also go to several different output buses. The out resource is similar in analogy to the slot resource that defines the input ports of the schematic and layout.

The present MDES requires that slots be the first resource used by an instruction grouping in the MDES ResTables section. The HGENEX tool has the additional requirement that for each instruction grouping in the ResTables section, the out resource(s) is specified. The clock cycle when this resource is used must be one greater than the last clock cycle of the last functional unit used. Figure 7.1 shows an improperly specified HMDES ResTables section with out bus problems. In the RL_ALU instruction grouping, the out resource should be used on clock cycle 4. The out resource is missing in the RL_IBr instruction grouping. Figure 7.2 shows a corrected form of the MDES.

Certain resources in the ResTables section of the HMDES are not real physical hardware resources. These resources model certain constraints that the hardware places on

```
RL_IAlu    (
              (slot 0)
              (Hialu_0 0..3)
              (out 3)
           )
RL_IBr     (
              (slot 0)
              (Hbranch 0)
           )
```

Figure 7.1: Example of an Improperly Specified HMDES ResTables Section: Out Bus Problems

```
RL_IAlu    (
              (slot 0)
              (Hialu_0 0..3)
              (out 4)
           )
RL_IBr     (
              (slot 0)
              (Hbranch 0)
              (out 1)
           )
```

Figure 7.2: Corrected HMDES ResTables Section

```
RL_IAlu    (
               (slot 0)
               (Hialu_0 0..2)
               (idummy_0 3)
               (Hialu_1 4..5)
               (idummy_1 5)
               (out 6)
           )
```

Figure 7.3: Example of an Improperly Specified HMDES ResTables Section: Hardware Use Problems

the compiler and are needed for compilation, but should not be included in the layout that HGENEX eventually produces. To solve this problem, HGENEX ignores all resources that do not start with a capital *H*, except the slot and out resources. An integer unit, *ialu*, is thus called *Hialu* in Figures 7.1 and 7.2, with the *H* standing for hardware.

Also, all clock cycles from 0 to the cycle used by the out resource(s) must be used by at least one hardware resource. There cannot be a clock cycle in which only nonhardware resources are used. This will cause HGENEX to give an error. Figure 7.3 shows an improperly specified ResTables section with hardware usage problems. Here clock cycle 3 is being used only by the *idummy* resource, which is not a physical hardware resource. Figure 7.4 shows one way of correcting the HMDES.

7.3 HGEN_PARMS

The file *HGEN_PARMS* contains various parameters that must be specified before HGENEX can run. A sample HGEN_PARMS is given in Figure 7.5. All seven parameters are mandatory. The format of the parameters are as follows: *parameter* = *value*. There

```
RL_IAlu    (
               (slot 0)
               (Hialu_0 0..3)
               (idummy_0 3)
               (Hialu_1 4..5)
               (idummy_1 5)
               (out 6)
           )
```

Figure 7.4: Corrected HMDES ResTables Section

must be a space after *parameter* and before *value*. There should also be an end of line character after *value*. Unlike other IMPACT parameter files, there is no semicolon after each parameter.

The first parameter, *bus_width*, is the width of the operand buses in bits that connect all of the resources together. Only a single value can be specified for all buses. Typical values would be 32, 64, or 128 bits. The parameter *cad_tool* is needed to tell the VHDL generator which CAD tool to generate VHDL for. Use *mentor* for Mentor Graphics Autologic or *synopsys* for the Synopsys Design Analyzer. The *lmdes* parameter states which LMDES file to use. The parameter *vhdl_file* is the name of the file that will contain the structural VHDL connecting the multiplexors and functional units after HGENEX is run (the connectivity file). The next parameter, *entity_file*, is the name of the new file HGENEX will create containing the definitions of the new hardware resources or entities (the entity file). To avoid confusion, it is recommended that *entity_file* and *vhdl_file* have very similar names. For example, an *E* could be used in the name to designate the *entity_file*. The *resource_file* parameter specifies the name of the resource file to

be generated by HGENEX. Each of these files is generated in the directory where the HGENEX executable is located.

```
bus_width      = 64
cad_tool       = synopsys
lmdes          = HP_4i1111.lmdes
vhdl_file      = HP_4i1111.vhdl
entity_file    = HP_4i1111E.vhdl
resource_file  = HP_4i1111.res
database_file  = Database
```

Figure 7.5: Example HGEN_PARMS File

Another file that is needed in addition to HGEN_PARMS is the database file whose name is specified by the parameter *database_file*. This file must be in the same directory as the HGENEX executable. Sizes of all of the hardware resources (given in mm^2) are specified in the database file needed by HGENEX to generate the resource file used by IC Station. The database file could contain the sizes of hundreds of different resources. HGENEX will search through the file and extract the sizes of only those functional units used in the particular design. Each resource should have a size for every stage that will be created by HGENEX, since every functional unit will be broken down into stages. A stage is specified with an S and then a number for the stage. Therefore, Hialu_0 stage 5 would become Hialu_0S5 in the database file.

The format of the file is *resource* = *size*. There must be a space after *resource* and before *size*. There must be an end of line character after *size*. An example database is given in Figure 7.6. After the MDES has been altered, the database file updated to contain the appropriate sizes of all of the resources used, and the HGEN_PARMS file set

to the desired values, the HGENEX executable file is ready to run. This executable is named *hgenex*.

```
Hialu_0_0_S1    14.95
Hialu_0_0_S2    14.95
Hialu_0_1_S1    14.95
Hialu_0_1_S2    14.95
Hbranch_0_S1    12.51
Hfalu_0_0_S1    20.5
Hfalu_0_1_S1    20.5
Hmem_0_0_S1     11.26
Hmem_0_1_S1     11.26
Hmem_0_2_S1     11.26
```

Figure 7.6: Example of a Database File

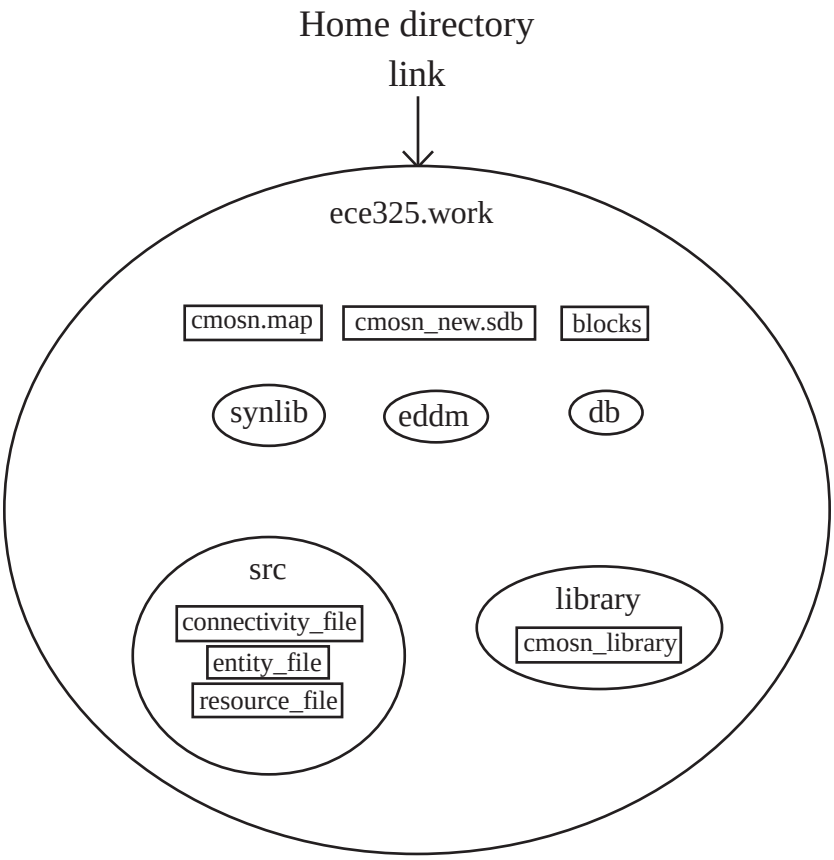
7.4 Schematic and Layout Scripts

To run the next phases of the HGENEX tool, an extensive setup procedure must be performed. These two phases must be run on machines installed with Mentor Graphics. Hence, files created by HGENEX may have to be transferred to different machines. Several environment variables must be set in order to use the Mentor tools for layout and the Synopsys tools for synthesis. Mentor variables are set up by running the VLSI class setup script *ece325*. The .cshrc must be modified to add certain environment variables for Synopsys. In addition, a file called *.synopsys_dc.setup* must be added to set up Synopsys specific parameters. To map the schematics from Synopsys tools to Mentor tools, a cmosn toolkit was purchased. This caused Synopsys to use the MOSIS cmosn library and technology to transfer the design to Mentor Graphics. An executable called

cmosn_new.sdb and a map file for transferring symbols to the cmosn library must be placed in a working directory where the tools will perform their tasks. Finally, the AMPLE userware file *blocks* should also be placed in the working directory alongside the cmosn setup file.

A symbolic link should be made from the home directory to the working directory, which is probably called *ece325.work* if the *ece325* script was used. Several directories should be made in this working directory. The first is the *synlib* directory, which will contain the Synopsys' analysis of the VHDL files. A *src* directory should be created to contain the files created from the VHDL generation phase of HGENEX. Hence, *src* should be loaded with the appropriate entity and connectivity VHDL files and resource files. A *db* and a *eddm* directory should be created to contain the db file created by Synopsys and the eddm Mentor schematics. Finally a *library* directory should be made which will contain a copy of the cmosn library. This library should be copied from the *$CMOSN_LIB/physical_lib* directory in Mentor using Design Manager. This is needed to add the block cells created by the layout script to the library so that the final hierarchical design may be created. Figure 7.7 shows the directory structure and other files needed to run the schematic and layout scripts.

To run the schematic phase of the tool, one simply invokes the script *synthesize* with certain specified arguments. Two arguments are necessary: the design and the directory. The design argument must be first and is the name of the design to be synthesized. By default, the design name is the name of the VHDL file without the *.vhdl* ending. The


Home directory
link
ece325.work
cmosn.map
cmosn_new.sdb
blocks
synlib
eddm
db
src
connectivity_file
entity_file
resource_file
library
cmosn_library

entity file is assumed to be the design name with an *E.vhdl* ending. The rest of the arguments must have a specifier. The directory argument has a *-dir* specifier and is the name of the symbolic link from the directory containing the *synthesize* script to the working directory. Therefore, to invoke the schematic script on the design HP_4i1111, using the run1 link to the working directory containing all of the setup directories, one executes the command *synthesize HP_4i1111 -dir run1* at the UNIX cshell prompt.

Other parameters that can be specified are the name of the db file to be generated using the *-db* specifier, the name of the connectivity VHDL file using the *-connect* specifier, the name of the entity VHDL file using the *-entity* specifier, and the name of the resource VHDL file using the *-resource* specifier. Once invoked, the script will synthesize the VHDL files and place the corresponding design in the db file. The script will also call a translation program, Synopsys db2eddm, which will create Mentor Graphics components and schematics. The design is ready for layout.

To run the layout phase of the tool, one invokes the script *make_layout*. The two arguments needed for this script are the name of the resource file and the name of the link to the working directory. Both arguments must have specifiers with *-resource* specifying the name of the resource file and *-dir* specifying the link to the working directory. Thus, to make a layout of the HP_4i1111 design with run1 as a link to the working directory, one would execute the command: *make_layout -resource HP_4i1111.res -dir run1* at the UNIX cshell prompt. The layout script will invoke the Mentor tools needed for layout. Block cells that had sizes specified in the database file are created first, and then multiplexors

are created using standard cells. All of these components are placed in the final layout and are routed together. This final layout is in a cell called *execute*. The entire process can take several hours to finish. Right now the HGENEX tool and the schematic/layout scripts are run on different file systems. The scripts must be modified if all needed tools are on one system.

8. COMPARISON OF HGENEX TO SIMPLEX COST ANALYSIS

8.1 Introduction

This chapter compares the simplex cost analysis method to the HGENEX cost analysis method. If the simplex method and the HGENEX tools achieve similar cost ratios, then generating layouts to estimate cost is not worth the trouble. However, if HGENEX produces different but reasonable results, it may be useful to have a tool that can estimate cost by generating a layout. Such a tool can also be much expanded and improved to become more useful to the architect.

8.2 HGENEX vs. Simplex Method

The HGENEX tool creates a layout for the functional units used in the processor. It does not generate a layout for other portions of the chip, such as the fetch, decode, and issue stages. Therefore, these other sizes must be added to the area HGENEX produces to determine a final layout size for HGENEX. Figure 8.1 gives equations determining total

chip area based on the equations in Chapter 4. Table 8.1 shows the comparison between the simplex method and HGENEX. This table shows the relative costs compared to a minimum configuration as well as the performance/cost ratios of each method. Two of the resource configurations have the best performance/cost ratios for four-issue machines and are highlighted in bold in the table. The {4431} and {2322} configurations were the optimal resource configurations for an eight-issue processor and were not able to be created because of hardware constraints set by the machines generating the layouts.

Four-Issue:

$$Area = \frac{(1 - Ex_{\%})(Chip_{Area}^{PC604}) + HGNEX_{Area}}{(1 - Ex_{\%})(Chip_{Area}^{PC604}) + HGNEX_{Area}^{Base}}$$

Eight-Issue:

$$Area = \frac{(1 - Ex_{\%})(Chip_{Area}^{PC604}) + 3 \times Issue_{Area} + HGENEX_{Area}}{(1 - Ex_{\%})(Chip_{Area}^{PC604}) + 3 \times Issue_{Area} + HGENEX_{Area}^{Base}}$$

$Chip_{Area}^{PC604}$ = Area of the PowerPC 604 in λ^2
$Issue_{Area}$ = Area for an Issue Unit in λ^2 for a Four-Issue Machine
$Ex_{\%}$ = Percent of Chip Area for Execution Unit
$HGENEX_{Area}^{Base}$ = HGENEX Area in λ^2 with One of Each Functional Unit
$HGENEX_{Area}$ = HGENEX Area in λ^2 for the Design

Figure 8.1: Equations for HGENEX Cost Analysis

As shown in Table 8.1, the areas generated from HGENEX are very high compared to those for the simplex method. The performance/cost ratios are also very different. The simplex method predicts that the {1222} configuration will cost more than the {2221}

configuration because the area of a floating point unit is more than that of a branch unit. However, Table 8.1 shows that HGENEX predicts the {2221} configuration will require more area. From the layout in Figure 6.10, it can be seen that routing and multiplexors make up the majority of the area. In addition, there is a great deal of space that is not used for anything. However, the layout of the PowerPC chip micrograph shows that very little space is wasted and the routing, while expensive, does not take up more area than the functional units. This seems to indicate that the results generated by HGENEX need more work to become valid. The improvements necessary for this are described in the next section.

Table 8.1: Comparison of the HGENEX and Simplex Cost Models

Resource Configuration						HGENEX Performance/Cost		Simplex Performance/Cost	
Branch	Memory	IALU	FALU	HGENEX Cost	Simplex Cost	Integer	Floating Point	Integer	Floating Point
1	1	1	1	1.00	1.00	1.00	1.00	1.00	1.00
2	**2**	**2**	**1**	1.85	1.20	0.89	0.65	1.38	1.01
1	**2**	**1**	**1**	1.67	1.06	0.66	0.68	1.04	1.07
1	2	2	2	1.76	1.24	0.77	0.74	1.09	1.06

8.3 Limitations and Problems with HGENEX

One major problem apparent from the layout in Figure 6.10 is the large amount of area in which there are no components and very little routing. In the full chip implementation, more effort would go into floorplanning to eliminate such empty space. Any remaining unused areas would be filled by other components not part of the execution stage. Eliminating this wasted area would greatly improve HGENEX. This could be

achieved by better floorplanning algorithms in the layout tool or by subtracting the unused space from the total area. Another problem is that a two-level metal interconnect was used for the routing. State of the art processes use a four-level metal interconnect. Routing in these processes is much less expensive because wires can be placed on top of each other.

Finally, the areas for the functional units are obtained by a real and very aggressive implementation. The multiplexor areas, however, are obtained from a standard cell design. In a real design most of the datapath would be implemented in a full custom design style. Hence, the sizes of the multiplexors relative to the functional units are not accurate. A more accurate layout for the execution stage would be generated if the same design style was used throughout this part of the chip. The problem with the full custom implementation is that automatic layout tools route the interconnect and give a larger layout than one done by hand. Hence, a fully standard cell design or functional unit areas based on such a design would cause HGENEX to give more accurate results. Such areas would only be accurate relative to others generated by HGENEX. Only the database file would need to be changed, and the present HGENEX could provide a better cost analysis than the simplex method. Whether this is truly the case is still unknown at this time.

9. FUTURE WORK

Chapter 8 showed that it is not clear whether the HGENEX tool presently provides a better cost model than does the simplex method for a superscalar microprocessor. While HGENEX includes the cost of peripheral multiplexor components, as well as the interconnection cost, the routing takes up too much area and the multiplexors and functional units were implemented in two different design styles, making their relative areas incorrect. More experiments need to be done using sizes of functional unit implemented with standard cells. Assuming that such new experiments yield encouraging results for HGENEX, further improvements could be made so that the tool would provide an even better cost estimate and could be expanded to eliminate all mathematical formulations for cost. Ideally, an HMDES file would be input, and a complete chip layout using a standard cell design style and working logic would be output. A path toward this eventual goal is presented.

First, the execution stage, as presently defined, does not have any input control signals. The first change would be to add an input control field specifying the control

for each of the functional units and would be quite straightforward. This could be added to HGEN_PARMS and would tell the schematic and synthesis tools to add extra ports and interconnections for the signals. This would probably not impact the size of the chip greatly, but the routing cost would increase because there would be more wires to interconnect and hence more constraints.

Another improvement would be to alter HGENEX so that multiple components with more than one stage would not be cross-connected, as illustrated in Figure 6.5. This could be an option in HGEN_PARMS. An even better, but more difficult solution, would be to change the MDES allowing for more precise connectivity information. The simple connectivity algorithm could be eliminated because exact connections would be specified. However, modifying the MDES would be very challenging since it is presently used by IMPACT and such changes could cause problems for the compiler. Another change would be to compress the stages so that a 15-stage floating point division unit with only one input and output for each stage, for example, would be compressed to a single component. This would reduce the number of components and the wiring required for the design, greatly speeding up the run time of the synthesis.

The functional units are currently specified in a database file containing sizes of each stage and resource. Instead of simple size information, a complete and working layout could be constructed out of standard cells if the appropriate specifications were input. A file would specify the exact functionality of an integer arithmetic unit and HGENEX could output a standard cell design that would be fully functional. While this would not

give accurate cost information in absolute terms because the standard cell design would take up more area than an aggressive implementation, it would give an accurate cost relative to other standard cell-based designs. This feature could also help the designer to achieve proper functionality and testing of new architectural features.

In Chapter 8, it was seen that a two-level metal process makes the routing of the components very expensive. State-of-the-art processes used in the PowerPC architecture use a four-level metal process [3]. Going to a process with more metal layers will reduce the interconnect area required because more levels can be used to do horizontal and vertical routing. Thus, if a state-of-the-art process using more interconnect levels were used with HGENEX, the routing costs would be much less pessimistic. To implement a new process with HGENEX, another standard cell library and process technology would have to be used in IC Station.

Finally, more areas of the microprocessor could be incorporated into the HGENEX tool. The fetch, issue, and decode stages would be generated along with the control section of the processor given the instruction set and available resources. All of these macro cells could then be put together so that an estimate of total chip area could be achieved, eliminating the need for a mathematical formulation to add in the areas of the other units. This would aid the architect in weighing the tradeoffs of various architectural features.

10. CONCLUSIONS

This thesis described two methods of determining cost for superscalar microprocessors: the simplex cost method (Chapter 4) and the HGENEX method (Chapter 6). The simplex method was based solely on a mathematical formula, while the HGENEX method tried to estimate cost by generating a layout for the execution stage, a crucial part of the IC. HGENEX produced layouts that were much larger than areas predicted by the simplex formula. The layout was not realistic because a two-level metal process was used, thereby overestimating routing costs, and the sizes of the components used were based on different design styles. More experiments are needed to determine whether HGENEX can produce more accurate cost numbers than the simplex method.

Performances of different superscalar microprocessors were also evaluated using the IMPACT compiler. The clock cycle count was estimated from the instruction schedule the compiler produced. Performance data were combined with cost data obtained from the simplex method to generate performance/cost ratios. The highest ratio was found for all possible configurations of four-issue and eight-issue superscalar processors assuming

branch, memory, integer ALU, and floating point ALU functional units. The configuration with the highest ratio for a four-issue machine running integer benchmarks was {2 branch, 2 memory, 2 IALU, 1 FALU}. When running floating point benchmarks, the best configuration was {1211}. For an eight-issue machine using the same functional units, the configuration with the highest performance/cost ratio for integer benchmarks was {4431}. For floating point benchmarks, it was {2322}. All such configurations achieved at least 80% of the maximum possible performance (n-issue width, with n of each functional unit) when running one particular type of benchmark.

REFERENCES

[1] J. C. Gyllenhaal, "A machine description language for compilation," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[2] E. DeLano et al., "A high speed superscalar PA-RISC processor," in COMPCON Spring 1992, pp. 116-121, 1992.

[3] S. P. Song and M. Denman, "The PowerPC 604 RISC microprocessor," Motorola Inc. and International Business Machines Corp., 1994

[4] S. A. Mahlke, "Exploiting instruction level parallelism in the presence of conditional branches," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

REFERENCES NOT CITED

[1] B. Burgess et al., "The PowerPC 603 microprocessor. A high performance, low power, superscalar RISC microprocessor," in COMPCON Spring 1994, pp. 300-306, 1994.

[2] C. R. Moore, "The PowerPC 601 microprocessor," in COMPCON Spring 1993, pp. 109-116, 1993.

[3] R. A. Bringmann, "Enhancing instruction level parallelism through compiler-controlled execution," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[4] P. J. Ashenden, *The VHDL Cookbook*. South Australia: Department of Computer Science, University of Adelaide, 1990.

[5] Synopsys Corp., *dc_shell Scripts for Synthesis v3.1b*. Mountain View, CA: Synopsys, 1994.

[6] Synopsys Corp., *Command Reference for Synthesis v3.1b*. Mountain View, CA: Synopsys, 1994.

[7] Mentor Graphics Corp., *Introduction to AMPLE Training Workbook, Software Version 8.2*. Wilsonville, OR: Mentor Graphics, 1993.

[8] Mentor Graphics Corp., *Advanced AMPLE Instructor's Notes, Software Version 8.2*. Wilsonville, OR: Mentor Graphics, 1993.

[9] Mentor Graphics Corp., *CMOSN Design Kit User's Manual, Software Version 8.2_5*. Wilsonville, OR: Mentor Graphics, 1993.

[10] Mentor Graphics Corp., *IC Station Reference Manual, Software Version 8.2_5*. Wilsonville, OR: Mentor Graphics, 1993.

APPENDIX A. COMPLEX EXAMPLE: HMDES

To illustrate HGENEX's ability to be versatile and also to handle more complicated examples, a complete design from MDES to layout is shown. All intermediate steps such as the VHDL files, resource file, and schematic are also included. Here the Resource and ResTables sections of the HMDES file are shown.

```
(Resources declaration
    slot[0..1]
    Hialu_0[0..1]
    Hfalu_0
    Hmem_0[0..1]
    Hbranch
    out[0..1]
end)

(ResTables declaration
    RL_INOP    (
                  (slot 0)
                  (out 1)
               )
    RL_IAlu    (
                  (slot 0)
                  (Hialu_0 0..3)
                  (out 4)
```

```
            )
RL_ICmp     (
               (slot 0)
               (Hialu_0 0)
               (Hbranch 0..1)
               (out 2)
            )
RL_IBr      (
               (slot 0)
               (Hbranch 0)
               (out 1)
            )
RL_IBr2     (
               (slot 0)
               (Hbranch 0)
               (Hmem_0 1)
               (out 2)
            )
RL_FPAlu    (
               (slot 0)
               (Hfalu_0 0)
               (out 1)
            )
RL_FPMul2   (
               (slot 0)
               (Hfalu_0 0..1)
               (out 2)
            )
RL_FPAluMul  (
                  (slot 0)
                  (Hfalu_0 0)
                  (Hialu_0 0)
                  (out 2)
              )
RL_FPMul3   (
               (slot 0)
               (Hfalu_0 0)
               (out 1)
            )
RL_FPDivS   (
               (slot 0)
               (Hfalu_0 0)
```

```
                (out 1)
            )
    RL_FPDivD  (
                (slot 0)
                (Hfalu_0 0..4)
                (out 5)
            )
    RL_Load    (
                (slot 0)
                (Hmem_0 0..1)
                (out 2)
            )
    RL_Store   (
                (slot 0)
                (Hmem_0  0)
                (Hfalu_0 1)
                (out 2)
            )
end)
```

APPENDIX B. COMPLEX EXAMPLE: ENTITY FILE

This is the entity VHDL file that HGENEX creates from the MDES file given in Appendix A. The entity file defines all of the components to be used in the design.

```
ENTITY mux1S_4B IS
   PORT (i0, i1  : IN BIT_VECTOR (3 DOWNTO 0);
         sel   : IN BIT;
         o     : OUT BIT_VECTOR (3 DOWNTO 0));
END mux1S_4B;

ARCHITECTURE rtl OF mux1S_4B IS
BEGIN
     WITH sel SELECT
           o <=
          i0 WHEN '0',
          i1 WHEN OTHERS;
END rtl;


ENTITY mux2S_4B IS
   PORT (i0, i1, i2, i3  : IN BIT_VECTOR (3 DOWNTO 0);
         sel   : IN BIT_VECTOR (1 DOWNTO 0);
         o     : OUT BIT_VECTOR (3 DOWNTO 0));
END mux2S_4B;
```

```
ARCHITECTURE rtl OF mux2S_4B IS
BEGIN
     WITH sel SELECT
           o <=
          i0 WHEN "00",
          i1 WHEN "01",
          i2 WHEN "10",
          i3 WHEN OTHERS;
END rtl;


ENTITY mux3S_4B IS
   PORT (i0, i1, i2, i3, i4, i5, i6, i7  : IN BIT_VECTOR (3 DOWNTO 0);
         sel   : IN BIT_VECTOR (2 DOWNTO 0);
         o     : OUT BIT_VECTOR (3 DOWNTO 0));
END mux3S_4B;

ARCHITECTURE rtl OF mux3S_4B IS
BEGIN
     WITH sel SELECT
           o <=
          i0 WHEN "000",
          i1 WHEN "001",
          i2 WHEN "010",
          i3 WHEN "011",
          i4 WHEN "100",
          i5 WHEN "101",
          i6 WHEN "110",
          i7 WHEN OTHERS;
END rtl;


ENTITY mux4S_4B IS
   PORT (i0, i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, i12,
         i13, i14, i15  : IN BIT_VECTOR (3 DOWNTO 0);
         sel   : IN BIT_VECTOR (3 DOWNTO 0);
         o     : OUT BIT_VECTOR (3 DOWNTO 0));
END mux4S_4B;

ARCHITECTURE rtl OF mux4S_4B IS
BEGIN
     WITH sel SELECT
```

```
          o <=
         i0 WHEN "0000",
         i1 WHEN "0001",
         i2 WHEN "0010",
         i3 WHEN "0011",
         i4 WHEN "0100",
         i5 WHEN "0101",
         i6 WHEN "0110",
         i7 WHEN "0111",
         i8 WHEN "1000",
         i9 WHEN "1001",
         i10 WHEN "1010",
         i11 WHEN "1011",
         i12 WHEN "1100",
         i13 WHEN "1101",
         i14 WHEN "1110",
         i15 WHEN OTHERS;
END rtl;


ENTITY Hialu_0_0_S1 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hialu_0_0_S1;

ARCHITECTURE behavior OF Hialu_0_0_S1 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hialu_0_0_S2 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hialu_0_0_S2;

ARCHITECTURE behavior OF Hialu_0_0_S2 IS
BEGIN
o <= NOT i;
END behavior;
```

```
ENTITY Hialu_0_0_S3 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hialu_0_0_S3;

ARCHITECTURE behavior OF Hialu_0_0_S3 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hialu_0_0_S4 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hialu_0_0_S4;

ARCHITECTURE behavior OF Hialu_0_0_S4 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hialu_0_1_S1 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hialu_0_1_S1;

ARCHITECTURE behavior OF Hialu_0_1_S1 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hialu_0_1_S2 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hialu_0_1_S2;

ARCHITECTURE behavior OF Hialu_0_1_S2 IS
BEGIN
o <= NOT i;
END behavior;
```

```
ENTITY Hialu_0_1_S3 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hialu_0_1_S3;

ARCHITECTURE behavior OF Hialu_0_1_S3 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hialu_0_1_S4 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hialu_0_1_S4;

ARCHITECTURE behavior OF Hialu_0_1_S4 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hbranch_0_S1 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hbranch_0_S1;

ARCHITECTURE behavior OF Hbranch_0_S1 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hbranch_0_S2 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hbranch_0_S2;

ARCHITECTURE behavior OF Hbranch_0_S2 IS
BEGIN
```

```
o <= NOT i;
END behavior;


ENTITY Hmem_0_0_S1 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hmem_0_0_S1;

ARCHITECTURE behavior OF Hmem_0_0_S1 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hmem_0_0_S2 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hmem_0_0_S2;

ARCHITECTURE behavior OF Hmem_0_0_S2 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hmem_0_1_S1 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hmem_0_1_S1;

ARCHITECTURE behavior OF Hmem_0_1_S1 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hmem_0_1_S2 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hmem_0_1_S2;
```

```
ARCHITECTURE behavior OF Hmem_0_1_S2 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hfalu_0_0_S1 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hfalu_0_0_S1;

ARCHITECTURE behavior OF Hfalu_0_0_S1 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hfalu_0_0_S2 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hfalu_0_0_S2;

ARCHITECTURE behavior OF Hfalu_0_0_S2 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hfalu_0_0_S3 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hfalu_0_0_S3;

ARCHITECTURE behavior OF Hfalu_0_0_S3 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hfalu_0_0_S4 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
```

```
END Hfalu_0_0_S4;

ARCHITECTURE behavior OF Hfalu_0_0_S4 IS
BEGIN
o <= NOT i;
END behavior;


ENTITY Hfalu_0_0_S5 IS
  PORT (i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END Hfalu_0_0_S5;

ARCHITECTURE behavior OF Hfalu_0_0_S5 IS
BEGIN
o <= NOT i;
END behavior;
```

APPENDIX C. COMPLEX EXAMPLE: CONNECTIVITY FILE

This is the connectivity VHDL file that HGENEX creates from the MDES file given in Appendix A. The connectivity file specifies how all components are connected.

```
ENTITY execute IS
  PORT (slot_in0, slot_in1  : IN BIT_VECTOR (3 DOWNTO 0);
        icp0, icp1, icp2, icp3, icp4,
        icp5, icp6, icp7, icp8, icp9, icp10,
        icp11, icp12, icp13, icp14, icp15, icp16,
        icp17, icp18, icp19, icp20, icp21, icp22,
        icp23, icp24  : IN BIT;
        output0, output1  : OUT BIT_VECTOR (3 DOWNTO 0));
END execute;

ARCHITECTURE structure OF execute IS

COMPONENT Hialu_0_0_S1
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hialu_0_0_S2
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hialu_0_0_S3
```

```
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hialu_0_0_S4
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hialu_0_1_S1
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hialu_0_1_S2
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hialu_0_1_S3
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hialu_0_1_S4
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hbranch_0_S1
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hbranch_0_S2
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hmem_0_0_S1
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hmem_0_0_S2
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hmem_0_1_S1
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
```

```
END COMPONENT;
COMPONENT Hmem_0_1_S2
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hfalu_0_0_S1
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hfalu_0_0_S2
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hfalu_0_0_S3
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hfalu_0_0_S4
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT Hfalu_0_0_S5
   PORT(i: IN BIT_VECTOR (3 DOWNTO 0);
        o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT mux1S_4B
   PORT(i0: IN BIT_VECTOR (3 DOWNTO 0); i1: IN BIT_VECTOR (3 DOWNTO 0);
       sel: IN BIT;
       o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT mux2S_4B
   PORT(i0: IN BIT_VECTOR (3 DOWNTO 0); i1: IN BIT_VECTOR (3 DOWNTO 0);
       i2: IN BIT_VECTOR (3 DOWNTO 0); i3: IN BIT_VECTOR (3 DOWNTO 0);
       sel: IN BIT_VECTOR (1 DOWNTO 0);
       o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT mux3S_4B
   PORT(i0: IN BIT_VECTOR (3 DOWNTO 0); i1: IN BIT_VECTOR (3 DOWNTO 0);
       i2: IN BIT_VECTOR (3 DOWNTO 0); i3: IN BIT_VECTOR (3 DOWNTO 0);
       i4: IN BIT_VECTOR (3 DOWNTO 0); i5: IN BIT_VECTOR (3 DOWNTO 0);
       i6: IN BIT_VECTOR (3 DOWNTO 0); i7: IN BIT_VECTOR (3 DOWNTO 0);
       sel: IN BIT_VECTOR (2 DOWNTO 0);
```

```
       o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;
COMPONENT mux4S_4B
   PORT(i0: IN BIT_VECTOR (3 DOWNTO 0); i1: IN BIT_VECTOR (3 DOWNTO 0);
       i2: IN BIT_VECTOR (3 DOWNTO 0); i3: IN BIT_VECTOR (3 DOWNTO 0);
       i4: IN BIT_VECTOR (3 DOWNTO 0); i5: IN BIT_VECTOR (3 DOWNTO 0);
       i6: IN BIT_VECTOR (3 DOWNTO 0); i7: IN BIT_VECTOR (3 DOWNTO 0);
       i8: IN BIT_VECTOR (3 DOWNTO 0); i9: IN BIT_VECTOR (3 DOWNTO 0);
       i10: IN BIT_VECTOR (3 DOWNTO 0); i11: IN BIT_VECTOR (3 DOWNTO 0);
       i12: IN BIT_VECTOR (3 DOWNTO 0); i13: IN BIT_VECTOR (3 DOWNTO 0);
       i14: IN BIT_VECTOR (3 DOWNTO 0); i15: IN BIT_VECTOR (3 DOWNTO 0);
       sel: IN BIT_VECTOR (3 DOWNTO 0);
       o: OUT BIT_VECTOR (3 DOWNTO 0));
END COMPONENT;

SIGNAL GROUND, sig2, sig3, sig4, sig5, sig6, sig7, sig8, sig9, sig10,
       sig11, sig12, sig13, sig14, sig15, sig16, sig17, sig18, sig19,
       sig20, sig21, sig22, sig23, sig24, sig25, sig26, sig27, sig28,
       sig29, sig30, sig31 : BIT_VECTOR (3 DOWNTO 0);


BEGIN
M0 : mux2S_4B PORT MAP (i0 => slot_in0, i1 => slot_in1, i2 => sig27,
     i3 => GROUND, sel(1) => icp0, sel(0) => icp1, o => sig2);
C0 : Hialu_0_0_S1 PORT MAP (i => sig2, o => sig3);
C1 : Hialu_0_0_S2 PORT MAP (i => sig3, o => sig4);
C2 : Hialu_0_0_S3 PORT MAP (i => sig4, o => sig5);
C3 : Hialu_0_0_S4 PORT MAP (i => sig5, o => sig6);
M1 : mux2S_4B PORT MAP (i0 => slot_in0, i1 => slot_in1, i2 => sig27,
     i3 => GROUND, sel(1) => icp2, sel(0) => icp3, o => sig7);
C4 : Hialu_0_1_S1 PORT MAP (i => sig7, o => sig8);
M2 : mux1S_4B PORT MAP (i0 => sig3, i1 => sig8, sel =>icp4, o => sig9);
C5 : Hialu_0_1_S2 PORT MAP (i => sig9, o => sig10);
M3 : mux1S_4B PORT MAP (i0 => sig4, i1 => sig10, sel =>icp5, o => sig11);
C6 : Hialu_0_1_S3 PORT MAP (i => sig11, o => sig12);
M4 : mux1S_4B PORT MAP (i0 => sig5, i1 => sig12, sel =>icp6, o => sig13);
C7 : Hialu_0_1_S4 PORT MAP (i => sig13, o => sig14);
M5 : mux1S_4B PORT MAP (i0 => slot_in0, i1 => slot_in1, sel =>icp7,
     o => sig15);
C8 : Hbranch_0_S1 PORT MAP (i => sig15, o => sig16);
M6 : mux2S_4B PORT MAP (i0 => sig3, i1 => sig8, i2 => sig16, i3 =>
     GROUND, sel(1) => icp8, sel(0) => icp9, o => sig17);
```

```
C9 : Hbranch_0_S2 PORT MAP (i => sig17, o => sig18);
M7 : mux2S_4B PORT MAP (i0 => sig16, i1 => slot_in0, i2 => slot_in1,
     i3 => GROUND, sel(1) => icp10, sel(0) => icp11, o => sig19);
C10 : Hmem_0_0_S1 PORT MAP (i => sig19, o => sig20);
C11 : Hmem_0_0_S2 PORT MAP (i => sig20, o => sig21);
M8 : mux2S_4B PORT MAP (i0 => sig16, i1 => slot_in0, i2 => slot_in1,
     i3 => GROUND, sel(1) => icp12, sel(0) => icp13, o => sig22);
C12 : Hmem_0_1_S1 PORT MAP (i => sig22, o => sig23);
M9 : mux1S_4B PORT MAP (i0 => sig20, i1 => sig23, sel =>icp14,
     o => sig24);
C13 : Hmem_0_1_S2 PORT MAP (i => sig24, o => sig25);
M10 : mux2S_4B PORT MAP (i0 => slot_in0, i1 => slot_in1, i2 => sig20,
      i3 => sig23, sel(1) => icp15, sel(0) => icp16, o => sig26);
C14 : Hfalu_0_0_S1 PORT MAP (i => sig26, o => sig27);
C15 : Hfalu_0_0_S2 PORT MAP (i => sig27, o => sig28);
C16 : Hfalu_0_0_S3 PORT MAP (i => sig28, o => sig29);
C17 : Hfalu_0_0_S4 PORT MAP (i => sig29, o => sig30);
C18 : Hfalu_0_0_S5 PORT MAP (i => sig30, o => sig31);
M11 : mux4S_4B PORT MAP (i0 => sig6, i1 => sig14, i2 => sig18, i3 =>
      sig16, i4 => sig20, i5 => sig23, i6 => sig27, i7 => sig28,
      i8 => sig3, i9 => sig8, i10 => sig31, i11 => sig21, i12 =>
      sig25, i13 => GROUND, i14 => GROUND, i15 => GROUND, sel(3) =>
      icp17, sel(2) => icp18, sel(1) => icp19, sel(0) => icp20, o =>
      output0);
M12 : mux4S_4B PORT MAP (i0 => sig6, i1 => sig14, i2 => sig18, i3 =>
      sig16, i4 => sig20, i5 => sig23, i6 => sig27, i7 => sig28,
      i8 => sig3, i9 => sig8, i10 => sig31, i11 => sig21, i12 =>
      sig25, i13 => GROUND, i14 => GROUND, i15 => GROUND, sel(3) =>
      icp21, sel(2) => icp22, sel(1) => icp23, sel(0) => icp24, o =>
      output1);
END structure;


-- Wire Count:  144

-- Multiplexor Count:
-- 4 Bit Mux 2 X 2:  5
-- 4 Bit Mux 3 X 4:  6
-- 4 Bit Mux 5 X 16:  2
```

APPENDIX D. COMPLEX EXAMPLE: RESOURCE FILE

This is the resource VHDL file that HGENEX creates from the MDES file given in Appendix A. The resource file specifies the components and their sizes to the Mentor Graphics layout tool, IC Station.

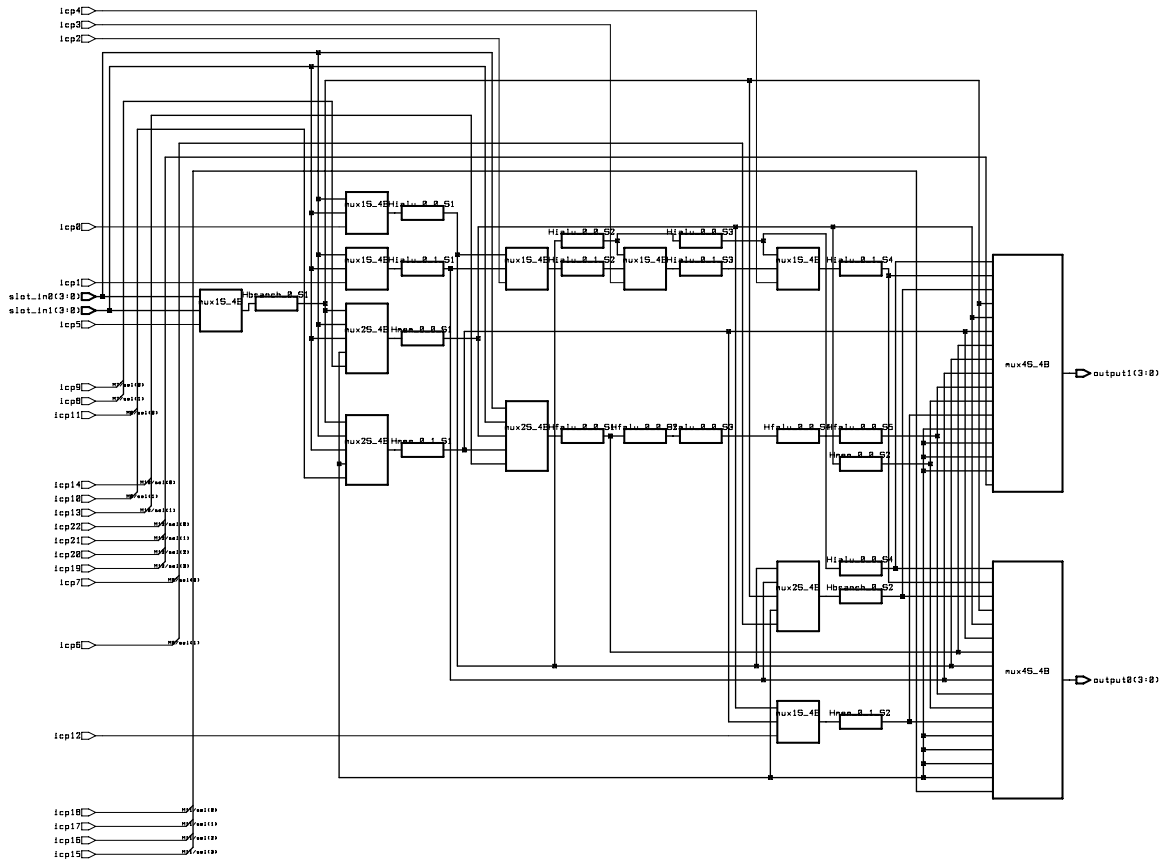
```
"buswidth"      4

"Hialu_0_0_S1"  4.950000
"Hialu_0_0_S2"  4.950000
"Hialu_0_0_S3"  4.950000
"Hialu_0_0_S4"  4.950000
"Hialu_0_1_S1"  4.950000
"Hialu_0_1_S2"  4.950000
"Hialu_0_1_S3"  4.950000
"Hialu_0_1_S4"  4.950000
"Hbranch_0_S1"  2.510000
"Hbranch_0_S2"  2.510000
"Hmem_0_0_S1"   1.260000
"Hmem_0_0_S2"   2.260000
"Hmem_0_1_S1"   3.260000
"Hmem_0_1_S2"   4.260000
"Hfalu_0_0_S1"  3.500000
"Hfalu_0_0_S2"  3.500000
"Hfalu_0_0_S3"  3.500000
```

```
"Hfalu_0_0_S4"  3.500000
"Hfalu_0_0_S5"  3.500000
"end"
"mux1S_4B_0"
"mux1S_4B_1"
"mux1S_4B_2"
"mux1S_4B_3"
"mux1S_4B_4"
"mux2S_4B_0"
"mux2S_4B_1"
"mux2S_4B_2"
"mux2S_4B_3"
"mux2S_4B_4"
"mux2S_4B_5"
"mux4S_4B_0"
"mux4S_4B_1"
"final_end"
```

APPENDIX E. COMPLEX EXAMPLE: SCHEMATIC

This is the schematic of the synthesized VHDL files given in Appendices B and C.



APPENDIX F. COMPLEX EXAMPLE: LAYOUT

This is the final layout generated from the schematic in Appendix E. The area of this layout is $155,602,192\lambda^2$.