Massachusetts Institute of Technology

6.375 Complex Digital Systems

2010 Spring

Advanced Processor Design

Group III

Michael Eskowitz

James Haupt

Advisor: Muralidaran Vijayaraghavan

Table of Contents

1. Abstract 3
2. Introduction 4
3. Basic CPU Design 4
4. Advanced Processor Implementation 6
   4.1. PCGen 8
   4.2. InstCounter 8
   4.3. Branch Predictor 9
   4.4. Decode 13
   4.5. Scoreboard 14
   4.6. Exec 15
   4.7. BExec 17
   4.8. MExec 18
   4.9. Counter4 18
   4.10. Writeback Stage 19
   4.11. Multi-Write Register File Implementation 20
5. Results 21
6. Conclusion 23
7. Future Work 24
   7.1. Scheduling Algorithm Details 24
   7.2. Scheduling Algorithm Analysis 27
   7.3. Application of Scheduling System for 3 Exec Modules 28
   7.4. Hardware Details 30
8. Bibliography 32

Table of Figures

Figure 3-1 Lab5 SMIPS Processor 5

Figure 3-2 Lab 5 Performance Metrics 5

Figure 4-1 Superscalar Processor Implementation 6

Figure 4-2 Branch Instruction Life-cycle 10

Figure 4-3 1-bit Branch Predictor Accuracy 10

Figure 4-4 1-bit Branch Predictor State Transition Diagram 11

Figure 4-5 2-bit Branch Predictor State Transition Diagram 11

Figure 4-6 2-bit Branch Predictor Accuracy 12

Figure 4-7 Tournament 2-bit Branch Predictor State Transition Diagram 12

Figure 4-8 Tournament 2-bit Branch Predictor Accuracy 13

Figure 4-9 Conceptual Representation of Decode Permutation Stage 13

Figure 4-10 Intra-Module Result Sharing Architecture 14

Figure 5-1 Summary of Benchmark IPCs 21

Figure 5-2 Comparison between issuing 2 or 1 instructions at a time 22

Figure 5-3 Resource Utilization 22

Figure 7-1 Expanding Execution Network 25

Figure 7-2 Probability of being able to schedule an instruction with two registers current in system 27

Figure 7-3 Potential Future Design Revision 30

Figure 7-4 Long-Term Architectural Goal 31

1. Abstract

In this project, we designed and implemented an in-order superscalar SMIPSv2 processor on a Xilinx Virtex-5 FPGA using Bluespec System Verilog. The somewhat novel approach we used to improve performance over a traditional superscalar processor is to include a Forward Map Buffer. This approach allows us to bypass the write-back stage entirely given instructions that depend on the previous instructions' results. By using the previous arithmetic instruction's result for the current instruction, we were able to show an improvement in performance over a traditional superscalar processor by over 20%.

2. Introduction

Microprocessors are the fundamental building block of our digital age. These processing systems are found in everything from cars to televisions to, certainly, our home computer. Today's high-end processors are capable of achieving a throughput of billions of operations per second (GOPS), a massive improvement over the systems available even as little as 5 years in the past. Despite the performance gains that are regularly obtained through the evolution of the microprocessor industry there are still many tasks which stress the processor to its limit and can even exceed the available system resources.

In order to rise and meet the challenge of these advanced computing tasks microprocessor designers employ a variety of methods and architectures to boost system performance. One particular technique that is often employed is known as a superscalar architecture. In superscalar processors, the central processing unit attempts to accelerate program execution by issuing multiple instructions simultaneously. It is the goal of this project to implement a basic superscalar processor which implements the SMIPS instruction set.

This paper will be organized as follows: We will describe basic processor design and functionality as related to the pipelined SMIPS processor that was implemented in lab 5 of this course. Having established a foundation for understanding microprocessor architecture, we will then present our advanced implementation of a superscalar SMIPS processor. This architecture will first be described at a high level in order to illustrate the functional role of the basic system components implemented during the course of this project. Once a high level understanding of superscalar system design is accomplished we will attempt to achieve a low-level understanding of the implementation details related to each module and functional unit within the system. We will then present our results and conclude with a summary of the future direction of this work.

3. Basic CPU Design

At the very basic level, microprocessors perform three simple steps in order operate. The microprocessor must first request an instruction from system memory (Fetch) in order to be assigned a task to perform. Once the instruction has been received, the processor will execute the desired operation using available data values (Decode/Execute). The results of this operation will then be written to local registers or system memory for future use (Writeback). There are many variations on this basic architecture, but all processors contain some implementation of this simple functionality.

The diagram below shows the pipelined SMIPS processor that was developed in lab 5 of this course. This processor architecture consists of three separate hardware blocks ("rules" in Bluespec System Verilog) that execute independent of one another. The FIFOs pcQ and wbQ located between the processor stages allow a decoupling of the stages and enable simultaneous firing of each stage. The stage pcGen is responsible for fetching the next instruction to be executed; the exec stage is responsible for decoding and executing the instruction that is received; and, finally, the writeback stage performs the expected functionality of committing execution results to memory.

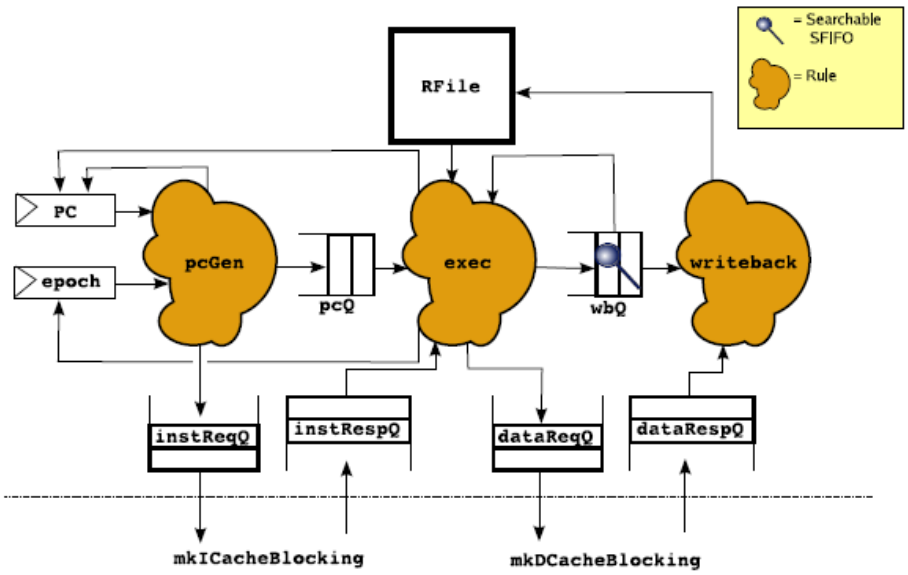

Figure 3-1 Lab5 SMIPS Processor

In order to test the processer, lab5 contained a set of benchmarks each of which performed a different task and exercising a subset of the processor's instruction set. The table below summarizes the instruction per cycle (IPC) metrics that were achieved for each of these benchmarks in both the original un-pipelined version and the modified, pipelined version. The initial version achieved an average IPC of approximately 0.65 while the pipelined version achieved an average IPC of approximately 0.79. It is the goal of this project to improve upon this design by implementing a super-scalar SMIPS processor.

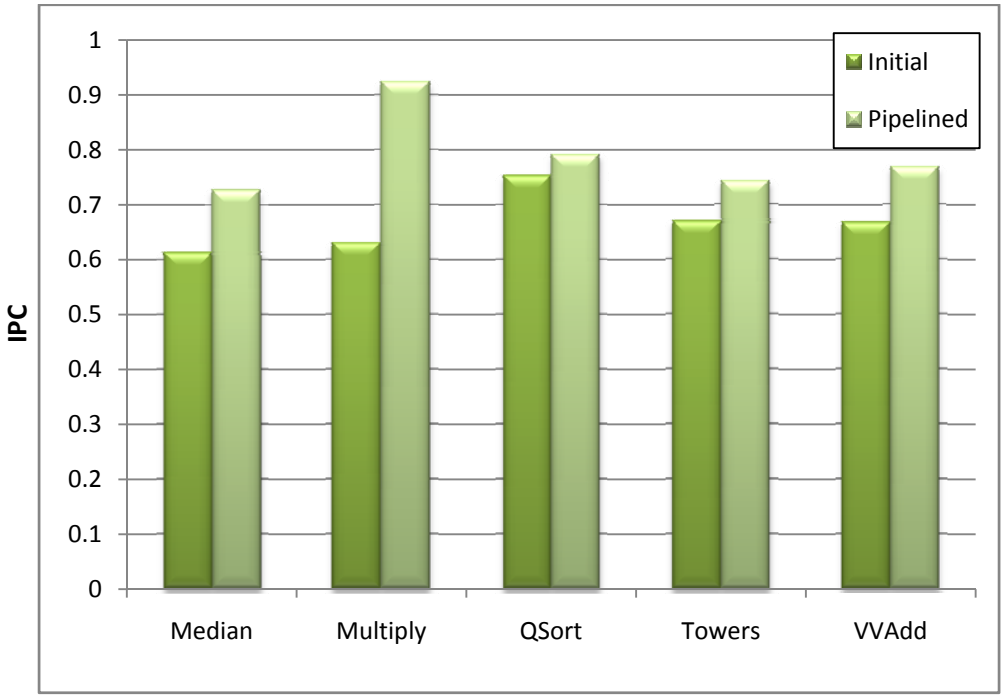

Figure 3-2 Lab 5 Performance Metrics

4. Advanced Processor Implementation

A superscalar processor attempts to achieve higher performance levels by issuing multiple instructions per clock cycle. The processor accomplishes this task by having redundant functional units that can each execute simultaneously. As we will see, the amount of performance increase depends largely upon the degree of instruction level parallelism within the code. This section of the report will provide a high-level overview of the functional implementation of our superscalar architecture which is shown in the diagram below.

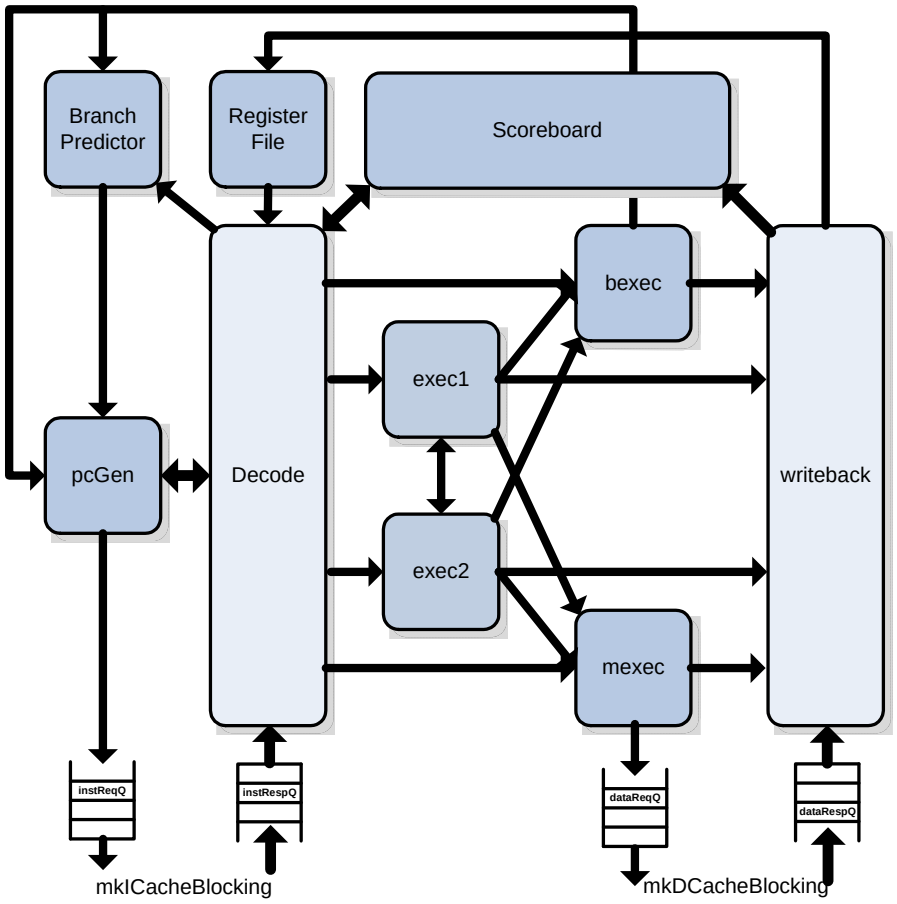

Figure 4-1 Superscalar Processor Implementation

This diagram is very similar in functionality to that of the pipelined SMIPS architecture discussed previously. Our implementation now consists of four pipeline stages: pcgen, decode, execute and writeback. The difference being that decode and execute have been separated with execute becoming a series of modules designed to handle the different instruction types. Additionally a number of supporting functional units have been developed in order to support the infrastructure necessary to accomplish superscalar execution.

The program counter implemented in our design is responsible for determining the flow of execution through a given code. Our module accomplished this by requesting a single address from memory. The

specific address that is requested is determined by the program counter (PC) register which tracks the most recently executed instruction address. Although pcgen requests only a single address we have modified the interface to cache memory such that it returns a total of two addresses in a single clock cycle which is necessary for our processor to execute multiple instructions per cycle. Pcgen must then employ a branch predictor in order to determine if either instruction is a branch and if it is taken.

The branch predictor in this system plays a crucial role in maintaining system performance. Should the processor fail to identify branches before executing them, it will need to invalidate all memory requests that are pending and fetch the correct address again. The latency between memory request and memory response can be quite lengthy and as such any faulty requests represent an unacceptable hit to system performance.

Once a memory request has been issued, the pcgen module will provide a record of the requested address as well as branch status (if any) to the decode rule. This rule is responsible for determining the instruction types as well as if the instructions can fire. The criteria for determining the allowability of an instruction is related to the number and type of instructions as well as if instruction source and destinations conflict. The decode rule will consult the system scoreboard in order to determine if the data required by the system is available. Any instructions that are not fired will be stored for future execution. Those that are fired, however, will be directed to the appropriate execution module: BExec, MExec, Exec1 or Exec2.

The processing system designed in the course of this project contains separate branch and memory execution units. The benefit for maintaining a separate branch unit is that it restricts the number of locations at which the program counter is changed. Further, a single memory execution unit exists as we are provided with a single interface to system memory. By separating these two from the main arithmetic execution modules it permits simultaneous branch and memory instructions to execute within the system.

Once execution has been completed, the writeback stage is responsible for performing any commits to the register file. The writeback stage connects to a multi-write register file which allows for the commit of multiple results in a single cycle. As these commits occur, the register addresses will be removed from the scoreboard in order to permit further execution using those addresses. This multi-write register file was necessary for superscalar execution.

As we have seen, at a high level, there are many components necessary for the correct functioning of a superscalar processor. The following sections will discuss the low level implementation details of each of the modules and rules as implemented in our design. Further we will discuss all performance enhancements that were made in order to optimize system performance as well as any pitfalls encountered along the way.

4.1. PCGen

As we have seen, a microprocessor operates by obtaining instructions from system memory, performing the specified operation and storing the results in system registers. The functional block which specifies the instruction addresses to be retrieved from memory is the module PCGen in our system. This module tracks the current address and essentially performs an educated guesses at what the next instruction will be. This module is accessed by BExec, Exec1 and Exec2 when the program counter needs to be changed. In order to accomplish this task the module provides a setPC() interface which allows the caller to override the program counter value being used for request in the current clock cycle. The complete interface definition for the PCGen module is shown below.

```
interface PCGen;
      method   Action           setPC(Addr newPC);
      method   Bit#(TagSz)      getEpoch;
      method   Action           incEpoch;
      method   PCQData          getPCQ;
      method   Action           decPC;
      interface Get#(MemReq#(AddrSz,TagSz,0)) getInstReq;
endinterface
```

The program counter is used in multiple locations. It is fed into the decode rule by means of a PCQData structure that is read out of the getPCQ() method and stored in a FIFO internal to the processor. Additionally, the PCGen module provides an interface to the instruction memory cache to request future instructions.

In addition to maintaining the program counter the PCGen module is responsible for maintaining an epoch register that is used to kill out-of-date instructions that occur when the program branches. The PCGen module provides a getEpoch() method to retrieve the current epoch as well as an incEpoch() method to increment the epoch register in the event of a branch.

4.2. InstCounter

In addition to using the epoch register to kill instructions that are no longer valid we employ an instruction counter to track the proper ordering of instructions within the system. The instruction counter module provides the method get() to obtain the current instruction count as well as the methods incr(), to increment the count, and reset(), to reset the counter to 0. Further, the module provides the methods oldValue() and oldValue2() which return the value prior to reset and the value before that (which is only available during the cycle when reset() is called). The complete interface definition is summarized below.

```
interface   InstCounter;
      method   Bit#(32)   get();
      method   Action     incr();
      method   Action     reset(Bit#(32) old);
      method   Bit#(32)   oldValue();
      method   Bit#(32)   oldValue2();
endinterface
```

4.3. Branch Predictor

The "educated" part of the "educated guess" that PCGen performs involves predicting if one of the next addresses corresponds with a branch instruction. The branch predictor module allows the system to more accurately determine when branches happen and adjust the program counter accordingly. In order to accomplish this feat the branch predictor module provides the methods search1() and search2() which allows the system to query two addresses in order to determine if they are known branches. In the event that a new branch is discovered, in the normal course of program operation, it can be added to the records that the branch predictor maintains using the update() method. Further, a falsely predicted branch can be removed from the branch predictor's records using the remove() method. A complete summary of the branch predictor's interface is presented below.

```
interface   BPredict;
      method   Action   update(Addr srcAddr, Addr destAddr, Bool taken);
      method   Action   remove(Addr srcAddr);
      method   Maybe#(Addr)   search1(Addr srcAddr);
      method   Maybe#(Addr)   search2(Addr srcAddr);
endinterface
```

The processor developed in lab 5 implemented what is known as a branch target buffer. These buffers typically are indexed by the lower few bits of the program counter (PC) address and maintain a small record of both branch addresses and the result of executing that branch (the address of the next PC). In our pipelined implementation this buffer is accessed in the pcgen module before the instruction is decoded or even received from memory. By accessing the buffer early in the pipeline we attempt to identify and handle branches as soon as possible. Identification occurs by determining if the instruction is in the buffer. If a matching PC is found in the buffer then the address is likely a branch. In the event of a branch, a memory request is immediately made for the predicted address contained in the buffer and the PC is updated with that address. If, however, a matching address is not found then we assume that the instruction is not a branch and we proceed with the normal flow of execution; the PC is incremented in the standard manner and all memory requests are issued according to this value. The following diagram, modified from [1] shows the mapping of this algorithm to the stages of the lab 5 processor. It should be noted that this diagram neglects a path by which incorrectly predicted branches are removed from the system.

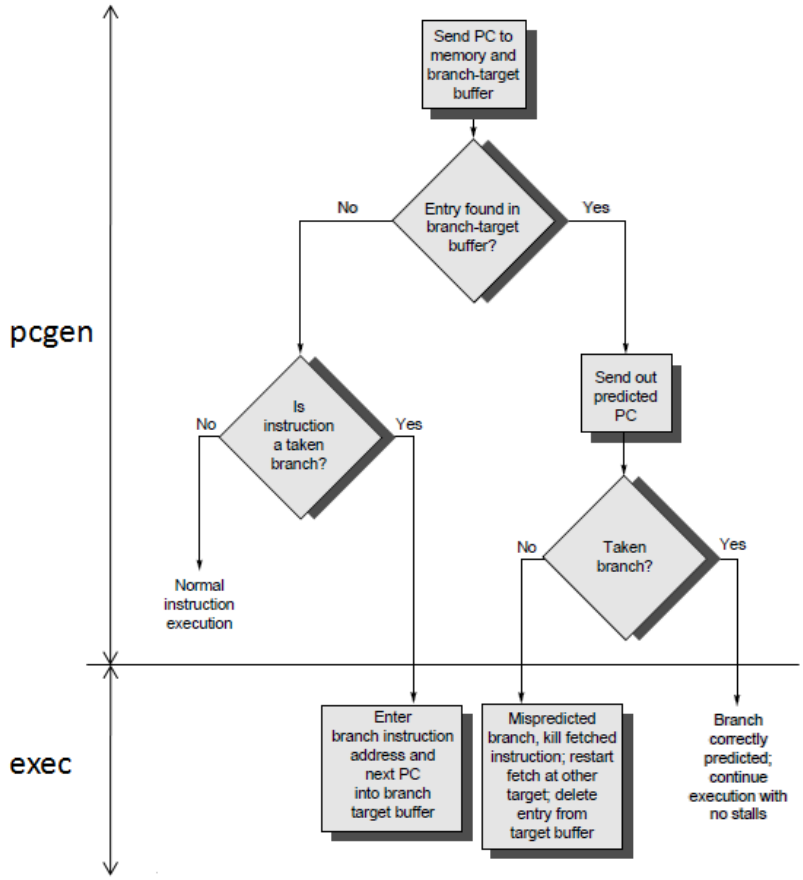

Figure 4-2 Branch Instruction Life-cycle

In evaluating our current branch predictor it is important to determine its hit rate. The following chart summarizes the percentage of correct branch predictions for the benchmarks used in lab 5.

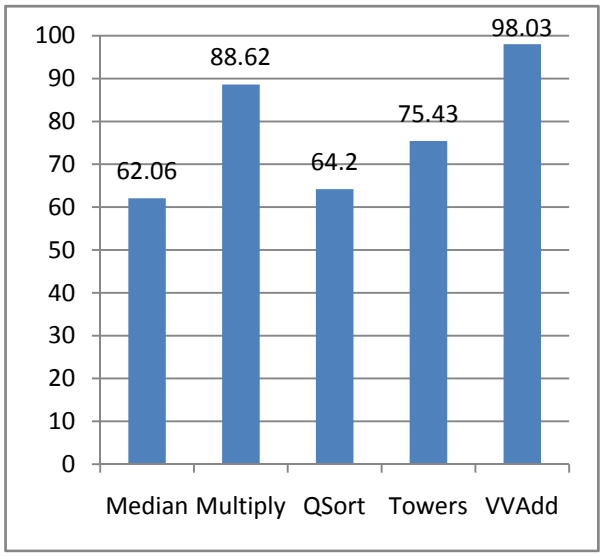

Figure 4-3 1-bit Branch Predictor Accuracy

We see from these numbers that the average prediction accuracy for our implementation is 77.6%. In order to access this performance we must rank it against other branch prediction algorithms. A search of the literature reveals [2] which contains a summary of branch predictor performance levels. According to this paper the Always Taken algorithm has a prediction accuracy of 62.5%, Last-Time achieves 89% and Two-Level Adaptive schemes can achieve from 90 to 95% accuracy depending upon the implementation type. Our implementation is a last-time algorithm and, as such, we can conclude that the performance metrics achieved in benchmarking agree with the numbers from this paper.

Although Two-Level Adaptive schemes offer the potential of up to 10% greater prediction accuracy we do not feel that it is feasible to implement such a scheme during the course of this project. In order to achieve greater branch prediction performance we will evaluate two potential modifications to our branch predictor. The first change possibility is to modify our implementation such that it is a 2-bit prediction scheme. The second option is to follow both paths at a branch and speculatively execute the resulting instructions until the true path is identified.

The branch predictor implemented in lab is a one-bit predictor, which is to say that we only know the state of its last execution. This attribute has profound implications for loop execution. In code that loops upon itself we expect to branch repeatedly until the execution path finally diverges. This implementation only remembers the last execution result for this branch which will always be a divergence. As a result, we expect that the branch predictor will always mispredict the first iteration through a loop. The transition diagram for this type of branch predictor is shown below.

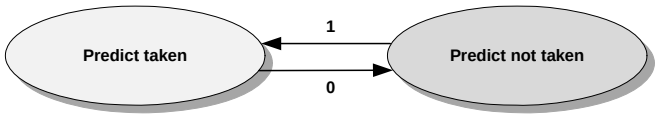

Figure 4-4 1-bit Branch Predictor State Transition Diagram

We can fix this prediction error by implementing a simple threshold which will change the prediction result only when the path has been followed twice in a row. The following diagram from [1] shows the states in a two-bit branch predictor. According to [3], a two-bit branch target buffer will have an average accuracy of 87%; a small gain for minimal effort.

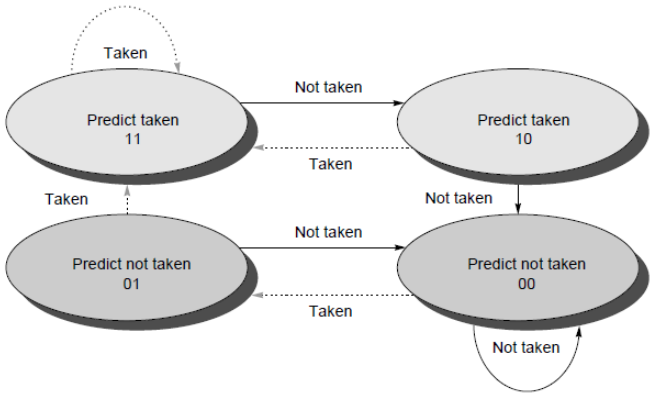

Figure 4-5 2-bit Branch Predictor State Transition Diagram

Our implementation of the 2-bit branch predictor achieved the following accuracy metrics. We see that these results are a slight improvement upon the 1-bit design discussed previously

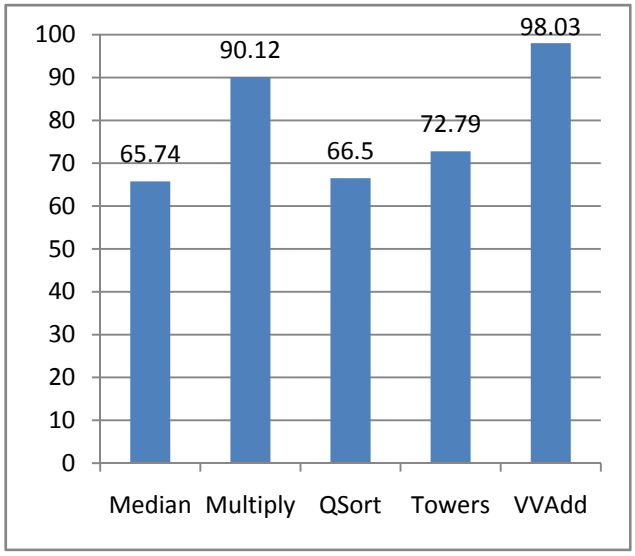

Figure 4-6 2-bit Branch Predictor Accuracy

In order to conduct a thorough survey of prediction algorithms for our design exploration phase we also considered a tournament predictor. The state diagram for a tournament predictor is shown below. This implementation achieved slightly lower average prediction accuracy than the 2-bit predictor. As a result of the lower performance metrics our final implementation contained a 2-bit predictor.

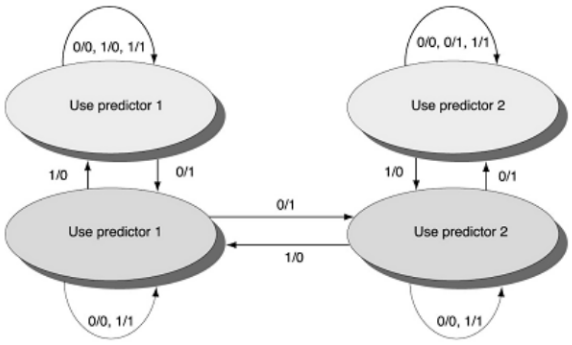

Figure 4-7 Tournament 2-bit Branch Predictor State Transition Diagram

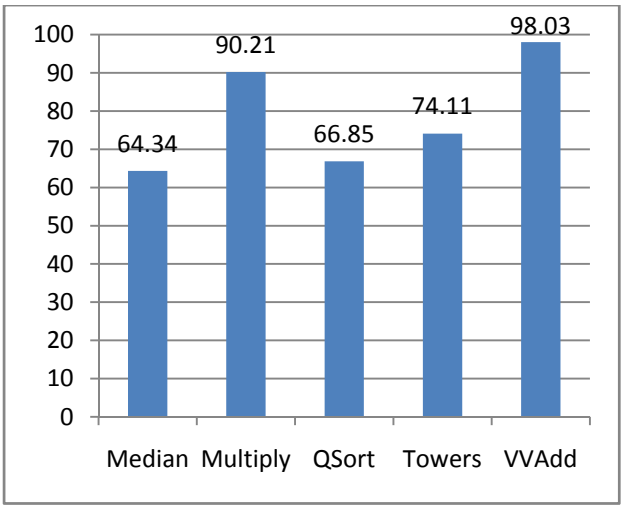

Figure 4-8 Tournament 2-bit Branch Predictor Accuracy

4.4. Decode

Receiving instructions from memory, the Decode stage performs the crucial operation of determining which instructions can be issued, and which modules should receive those instructions. This is accomplished through a series of three stages, which we refer to as Permutation, Blocking, and Issuance.

In the first stage, we examine first a set two registers that contain instructions we were not able to fire from the last cycle, as well as the two instructions we receive from the Instruction Memory Cache. We pick two of these instructions in order, with preference given to the held instructions, to send to the next stage. The distribution of these instructions is shown in the figure below.

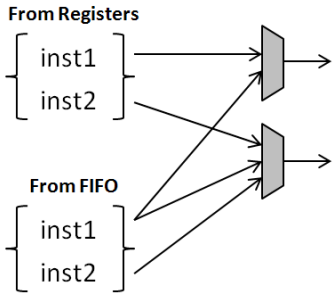

Figure 4-9 Conceptual Representation of Decode Permutation Stage

In the second stage, “Blocking”, we determine if we can issue both instructions, or only the first, through a series of checks. We block an instruction if one of the registers it is attempting to read has an outstanding write as determined by the scoreboard, which could happen on either instruction. Next, we check to make sure that the second instruction is not reading the same register the first writes to, or they are both writing to the same location, in which case we would let the first instruction be issued, and hold the second. Lastly, we check for double-memory or double-branch instructions, in which case

we have to hold the second since we can only handle one memory and one branch instruction at a time. A blocked first instruction also blocks the second instruction, maintaining the in-order execution of our processor.

Given a blocked instruction, we allow exceptions to occur by using a "Forward Map Buffer", which keeps track of the previous instructions issued to the two arithmetic execution units: Exec1 and Exec2. Each of the execution units have a register which contains the result of the previous cycle's execution of an arithmetic instruction, and permit reading of that register by the other modules. This allows utilize the evaluation of a previous cycle's instruction, without having to wait for the result to be written back to the register file, thus providing a form of bypass. The diagram below illustrates the data movement between the modules.

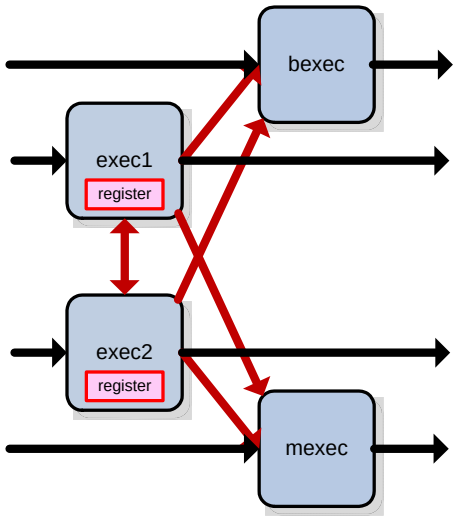

Figure 4-10 Intra-Module Result Sharing Architecture

Lastly, in the final stage, we issue the instructions to their respective modules. So a branch instruction would go to BExec, memory instruction to MExec, and an arithmetic instruction to one of the Exec modules. It is possible to issue two of any combination of 1 branch instruction, 1 memory instruction, and 2 arithmetic instructions.

4.5. Scoreboard

The processor that we have designed is capable of executing multiple instructions simultaneously. As we have seen, there are numerous ramifications from having such an architecture as it is possible to execute instructions that conflict with each other in parallel or use addresses that contain values that are no longer current. Issues involving stale data are known as data hazards and a common method of resolving them is through the use of a scoreboard.

The scoreboard implemented in our design provides several methods for tracking the number of writes outstanding for a given register. The inc() method is used in decode to increment the count stored in

the scoreboard indicating if the register is valid or not. Further, the interface contains a total of 4 read methods (rd1() through rd4()) which are used to query the availability of a total of 4 different registers. Similarly, there are an additional 4 decrement methods (dec1() through dec4()) which are used to remove registers from the scoreboard when write back occurs. The complete interface definition of the scoreboard is shown below.

```
interface   SB;
       method   Action    inc( Rindx rindx1, Rindx rindx2 );
       method   Action    dec( Rindx rindx );
       method   Action   dec1( Rindx rindx );
       method   Action   dec2( Rindx rindx );
       method   Action   dec3( Rindx rindx );
       method   Bit#(4)   rd1( Rindx rindx );
       method   Bit#(4)   rd2( Rindx rindx );
       method   Bit#(4)   rd3( Rindx rindx );
       method   Bit#(4)   rd4( Rindx rindx );
endinterface
```

4.6. Exec

The Execution modules in our processor are simple blocks that perform a single operation that is determined by the instruction that is passed to it through its put() method. This method stores the incoming instruction of type EMData in an internal FIFO until it is ready to be processed. Each module contains a process rule which reads from this FIFO, executes the first enqueued instruction and stores the result in a FIFO associated with its output method. The interface definition associated with the Exec module is shown below. The get() and deq() methods return the calculated result and dequeue the output FIFO respectively.

```
interface   Exec;
       method   Action     put(EMData dataInstIn);
       method   Data       getLastResult();
       method   MResult    get();
       method   Action     deq();
endinterface
```

Among the elements contained within the EMData type is an Instruction Count (IC) associated with the operation to be performed and either the data to be processed or a flag to let the module know that it is to use data from a neighboring processor (and which neighbor's data to use). The idea behind this concept is to allow each execution module to send the results of the most recent operation to its neighbors without requiring a write to the RFile first. The result of the instruction and the instruction's IC are extracted from the module by a rule within the processor which is responsible for writing to the register file.

The interface to the Exec module is responsible for handling two separate data types: the EQData and MResult structures. The EQData structure contains all information necessary to track and kill instructions within the processor (epoch and instruction count) as well as the instruction opcode, all associated data and the destination address. The instruction definition of EQData is shown below below.

```
typedef struct   {
       Bit#(TagSz)  epoch;
       Bit#(32)     icount;
       Bit#(6)      opcode;
       ExecData     dat1;
       ExecData     dat2;
       Rindx        rdst;
} EQData deriving(Bits, Eq);
```

We see that this data structure has elements of the type ExecData, which is a tagged union containing the elements listed below. As described in previous sections of this report we accomplish bypassing from writeback to decode through the Exec modules themselves. By tracking the destinations being written to by previous instructions we are able to use the old results in future instructions. The tag “Mine” instructs the Exec module to use the value it has stored in its register. If, however, the tag “Current” is passed in, the module knows to extract the data present in the structure. Further, the tags “North” and “South” direct the module to take the data value from a specific adjacent module.

```
typedef union tagged   {
       Data   Current;
       void   Mine;
       void   North;
       void   South;
}   ExecData deriving (Eq, Bits);
```

The result of execution of all three processing modules (Exec, BExec and MExec) is the MResult type. This data structure encapsulates all information necessary to track and kill instructions within the writeback stage as well as the result of executing the operation.

```
typedef struct   {
       Bit#(TagSz)  epoch;
       Bit#(32)     icount;
       WBResult     result;
}   MResult deriving(Bits, Eq);
```

4.7. BExec

In addition to the execution module, previously described, we have split off all functionality relating to branch and jump instructions into a separate module. The benefit of localizing these instructions into a single block is that the control logic required to track and change the program counter is minimized. As with all processing modules in the system, BExec receives instructions via a put() method and its output is accessed via get() and deq() methods. The interface definition for the BEXec module is listed below.

```
interface   BExec;
      method   Action     put(BQData  branchInstIn);
      method   MResult    get();
      method   Action     deq();
      method   Bool       no2ndInstruction();
      method   Bool       nextInstBad();
      method   Bool       regInstBad();
endinteface;
```

The methods no2ndInstruction(), nextInstBad() and regInstBad() are used in the decode rule in order to appropriately schedule execution. For all mispredicted branches, either taken or not, the method nextInstBad() will return True. In this case, if the instruction is also the first in the set of instructions to be executed then the method no2ndInstruction() will return True. In the case where the first instruction is a branch and it is taken regInstBad() will return True in order to cause the system to kill the next instruction in the sequence.

The branch execution module is the only component in the system to receive the BQData type. Similar to the EQData type, this structure encapsulates all the information necessary to identify and kill the instruction during processing. Additionally, this type includes the program counter, the predicted next value for this counter as well as a sequence number indicating the position of the branch within the set of instructions presently being considered. The remainder of the structure includes the actual instruction being issued as well as all required data words.

```
typedef struct   {
      Addr         pc;
      Bit#(TagSz)  epoch;
      Addr         nextpc;
      Bool         branched;
      Bool         inst2isabranch;
      Bit#(2)      instCount;
      Bit#(32)     icount;
      Instr        inst;
      ExecData     dat1;
      ExecData     dat2;
}   BQData deriving(Bits,Eq);
```

4.8. MExec

The final module in our collection of execution blocks is the memory execution module, MExec. This module is responsible for handling all load word, store word and MTC/MFC instructions. This module follows the standard put(), get() and deq() interface set forth within our design. Further, the cp0_tohost, cp0_fromhost and cp0_statsEn registers that are implemented within the processor have been encapsulated into this module. All writes to these registers are through the put() method and all reads are done using the get_cp0_tohost(), get_cp0_fromhost() or get_cp0_statsEn() accessor methods. The MExec module interface is shown below.

```
interface   MExec;
      method   Action     put(MQData memInstIn);
      method   MResult    get();
      method   DataReq    getMemReq();
      method   Action     deq();
      method   Action     deqMemReq();
      method   Bit#(32)   get_cp0_tohost();
      method   Bit#(32)   get_cp0_fromhost();
      method   Bool       get_cp0_statsEn();
endinterface
```

The MExec module receives an input of type MQData which contains tracking information, in the form of the epoch and instruction count, as well as the instruction and associated data values. The complete type definition is shown below.

```
typedef struct {
      Addr         pc;
      Bit#(TagSz)  epoch;
      Addr         nextpc;
      Bit#(32)     icount;
      Instr        inst;
      Data         dat1;
      Data         dat2;
}   MQData deriving(Bits,Eq);
```

4.9. Counter4

The implementation of the Counter4 module is relatively simple in comparison to the other modules in the system. Each of its increment methods is called by a separate module whenever an instruction is executed. The effect of calling one of these methods is to set a PulseWire as active in order to indicate that the method has been performed. Internal to the module is a single rule that fires on each clock cycle when enabled. This rule counts the number of PulseWires that have been set as active and adds that value to the running count of instruction that have been performed. The following code snipet illustrates how this is performed.

```
module mkCounter4(Counter4#(t_type))
       provisos (Bits#(t_type,asz),Arith#(t_type),Literal#(t_type));
       …
       rule   update(go_count);
              t_type inc_1   =   ( inc_numinst[0] )   ?   1   :   0;
              t_type inc_2   =   ( inc_numinst[1] )   ?   1   :   0;
              t_type inc_3   =   ( inc_numinst[2] )   ?   1   :   0;
              t_type inc_4   =   ( inc_numinst[3] )   ?   1   :   0;
              t_type temp_val=   counter_val + inc_1 + inc_2 + inc_3 + inc_4;
              count_val   <=   temp_val;
       endrule
       …
endmodule
```

The Counter4 module is designed to track the number of instructions being executed simultaneously. The module contains four inc#() methods which are accessed by the four system modules responsible for executing instructions: BExec, MExec, Exec1 and Exec2. The number of modules executing instructions at one time can be obtained using the method getValue(). Further, execution tracking can be enabled or disabled using the enable() method. The module interface is shown below.

```
interface Counter4#(type t_type);
       method   Action   inc0;
       method   Action   inc1;
       method   Action   inc2;
       method   Action   inc3;
       method   Action   enable(Bool onoff);
       method   t_type   getValue;
endinterface
```

4.10. Writeback Stage

The writeback stage of our processor has changed slightly compared to the one from Lab 5. In lab 5, we only needed to support 1 Exec module, however, for our superscalar architecture, we must support parallel writes to the register file. In order to achieve this functionality, we implemented a series of rules that are similar to the one used in lab 5. Each of these rules handle the particular writeback for each module.

4.11. Multi-Write Register File Implementation

The register file used in lab 5 was a wrapper around Bluespec's built in register file. The limitation with this particular design, however, is that it supports only a single write per clock cycle. Given that the goal of this project is to issue multiple instructions per cycle it is a requirement that our processor be able to write back multiple results at a time. This requirement necessitated that we develop our own multi-write register file. The task of developing a multi-write register file was accomplished by constructing a module containing a vector of registers.

The interface to our register file, shown below, contains multiple read and write methods. It is important to note that the hardware cannot write to the same register multiple times during a single clock cycle. The possibility of this event occurring is eliminated by the scheduling algorithm implemented in the decode stage. Two instructions containing the same destination register address are never issued simultaneously.

```
interface   RegFileMW;
      method   Action     wr0(Rindx addr, Bit#(32) d);
      method   Action     wr1(Rindx addr, Bit#(32) d);
      method   Action     wr2(Rindx addr, Bit#(32) d);
      method   Action     wr3(Rindx addr, Bit#(32) d);
      method   Bit#(32)   rd1(Rindx addr);
      method   Bit#(32)   rd2(Rindx addr);
      method   Bit#(32)   rd3(Rindx addr);
      method   Bit#(32)   rd4(Rindx addr);
endinterface
```

5. Results

During the course of this project we successfully implemented a superscalar version of the SMIPS processor used in lab 5. One of the resources provided with that lab which we have used extensively was a set of benchmarks. These benchmarks tested the functional correctness of our processor by performing a number of simple algorithms to exercise all elements of the available instruction set. Furthermore, as our implementation was a superscalar design, these benchmarks served to exercise all permutations of instruction issuance and delay as described in the prior sections.

The chart below summarizes the instruction per cycle (IPC) results for the execution of each of the benchmark programs. The table contains these results for 4 different implementations of our processor: the original SISD non-pipelined SMIPS processor, the SISD pipelined processor, our superscalar implementation without feedback between modules and the final superscalar architecture with feedback between modules. What we see from this chart is that on average we achieved a performance increase of 37% over the IPC of pipelined processor.

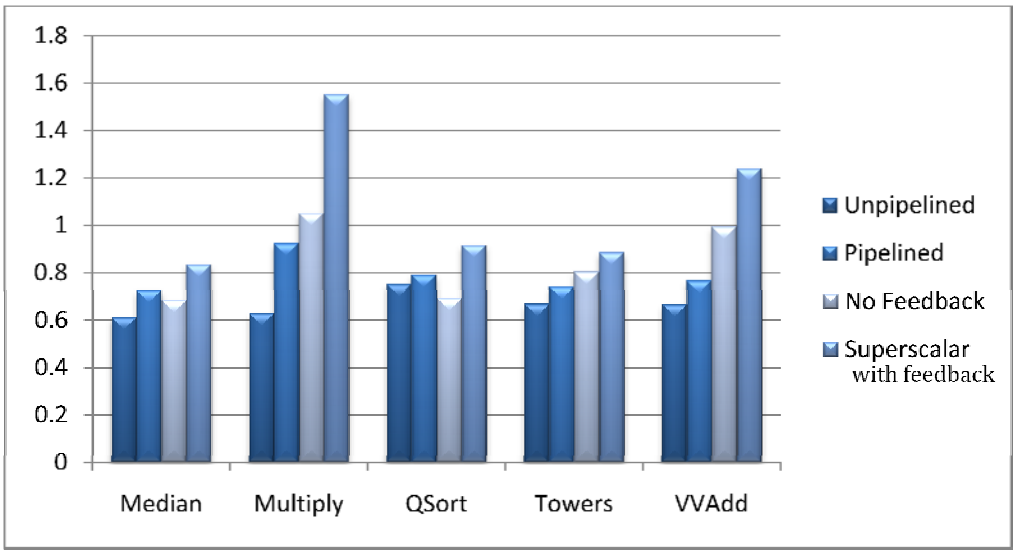

Figure 5-1 Summary of Benchmark IPCs

Looking at these results, however, it quickly becomes clear that the degree of acceleration is not constant but rather depends on the instruction level parallelism inherent within the code. This becomes even more readily apparent from the next chart which shows the number of times only 1 instruction is fired versus when 2 instructions are fired for each benchmark. The huge gain seen in the multiply benchmark is due to the fact that the system is able to issue 2 instructions simultaneously for the vast majority of the program.

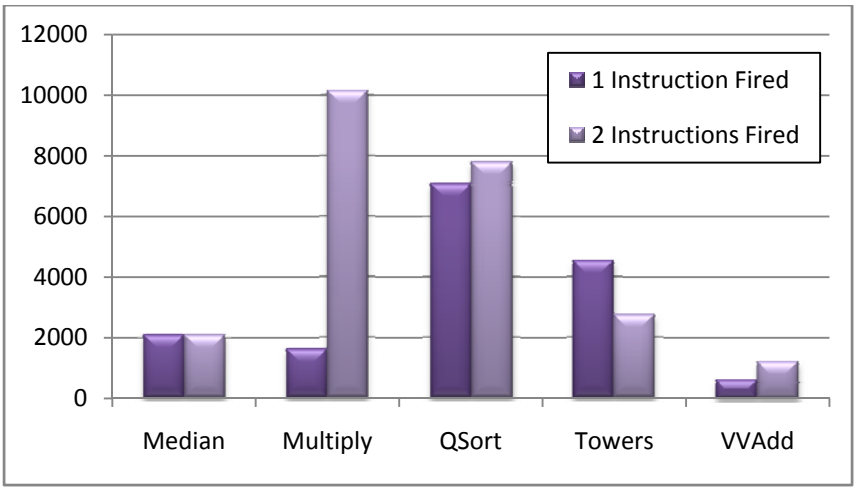

Figure 5-2 Comparison between issuing 2 or 1 instructions at a time

These performance gains are not without consequences, however, as the FPGA resource utilization has more than doubled from the implementations in lab 5. The table below summarizes the registers and LUTs required to implement the superscalar processor with feedback between modules. The additional resource requirements were largely due to the expanded decode stage as well as the addition of a scoreboard and additional writeback rules to read from each of the processing modules.

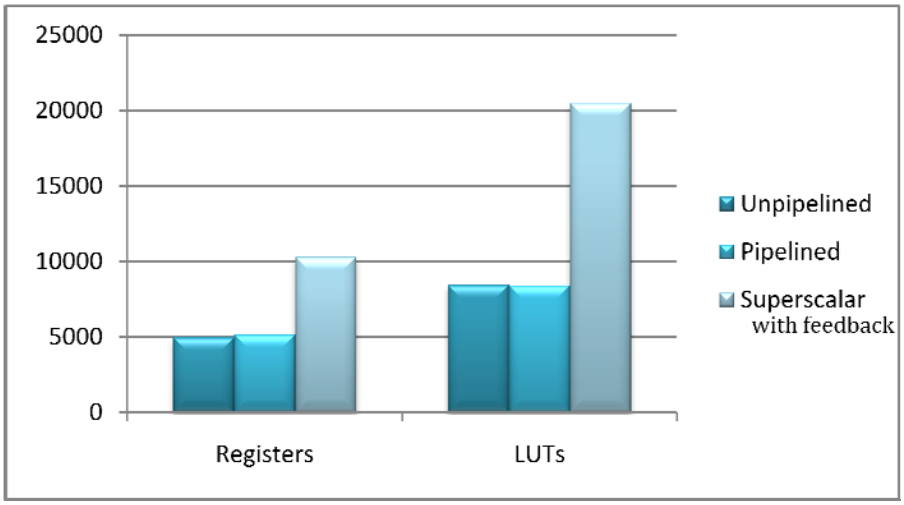

Figure 5-3 Resource Utilization

6. Conclusion

In the course of this 6-week project we successfully implemented a superscalar version of the SMIPS processor architecture. In order to accomplish this feat we modularized our original processor design by breaking off the logic to handle different instruction types into specialized modules. Branch and jump instructions all went to a BExec module in order to restrict the number of locations where the program counter could be changed. Memory load and store instructions were directed to a single MExec module which connected to our single memory interface. All arithmatic instructions were directed to two execution modules: Exec1 and Exec2. By having a redundant, segmented architecture we were able execute multiple instructions per clock cycle.

In addition to the various processing modules described here-in we accelerated our design through the use of a forward map buffer implemented in the decode stage of our processor. This buffer tracked the destination register of any issued instruction that required a register write upon completion. This table allowed us to associate the register with a specific execution unit within the system and allowed us to bypass writeback in some cases by exchanging values within and between the modules. The various stages involved in our implementation showed that the there was a great deal of performance improvement to be gained by handling operating in this manner.

Further, we have seen that the degree of acceleration achieved by the implementation of a superscalar architecture is largely dependent upon the instruction level parallelism inherent in the code being executed. We saw enormous gains in the Multiply benchmark as that code had very few conflicts between instructions. The tables presented in our results section show that this benchmark issued 2 instructions approximately 80% of the time in order to effect a near doubling of IPC. It is clear from these results that inherent code parallelism and efficient compiler optimizations are just as important in high performance computing as efficient hardware design. The accelerations we achieved would not be possible with poorly written code that had a low degree of parallelism.

7. Future Work

In the course of this project we implemented a superscalar architecture containing a total of two execution units. The long term goal of these development efforts is to construct a scalable framework in order to implement a dynamically reconfigurable data path processor. Towards this end we will need to extend our implementation such that it can include arbitrary number of execution modules. This process will involve developing a network infrastructure as well as extending our design to accommodate instruction reordering. The following pages will detail the algorithmic details of our intended implementation as well as all hardware revisions that will be necessary.

7.1. Scheduling Algorithm Details

To date we have implemented a method for efficiently parallelizing a serial instruction stream and demonstrated the benefit of module level bypassing. The scheduling algorithm implemented relied on the execution of at most two instructions. In order to make effective use of the interconnection between execution units in our processing system we will employ a forward write buffer to assist in scheduling instructions in the processing fabric. Each execution unit in the system will have a slot in the table. The table contains the destination register of the most recently executed instruction. Each execution unit is assigned a unique orthogonal id. Orthogonality is guaranteed through the use of one hot encoding. For instance, the ID assignments for a 3 execution unit system would be

Execution Unit	Execution ID
e1	001
e2	010
e3	100

The benefit of having orthogonality at this level is that we can easily apply set operations with simple binary arithmetic. For instance, if we are given a set of processing resources

```
P = { e1, e2, e5 }
```

we can represent this configuration by ORing the execution IDs of each of the set members.

```
P = 00001 || 00010 || 10000 = 10011
```

The resulting 5-bit word uniquely identifies a set containing only the execution units e1, e2 and e5. From this we see that the union (∪) of orthogonal set elements is the binary OR operation. It easily follows that the intersection (∩) is the binary AND.

As we have seen, the long term goal of this design path is a system which can contain an arbitrary number of execution units. The diagram below abstracts away the majority of interconnects required for book keeping and shows the primary data path through the system. The interconnection between execution units is of central importance to this design.

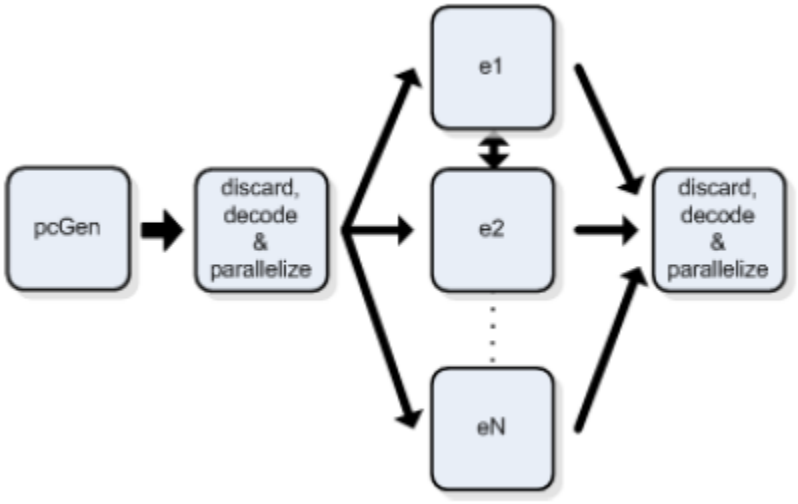

Figure 7-1 Expanding Execution Network

With a potentially increasing number of execution units the task of routing feeds from a bypass register file to each unit becomes increasingly cumbersome. An FPGA contains a finite number of layers and, as such, we foresee the interconnection between execution units and the register file to eventually become unrouteable. This obstacle is overcome by bypassing the register file within the execution module itself. Our execution modules will exchange results between their neighbors as well as maintain the results of the previous execution. It is the goal of our scheduling algorithm to exploit this bypass as effectively as possible.

As we have shown, we can uniquely identify an execution resource with a multi-bit orthogonal identification number. This number can also be taken to represent the location of data within system. The following pseudo code illustrates the process by which we can locate two registers (r1 and r2) within the system by checking the write history table.

```
r1loc = ((writehistory[1]  ==  r1) && 3'b001)
     || ((writehistory[2]  ==  r1) && 3'b010)
     || ((writehistory[3]  ==  r1) && 3'b100);
r2loc = ((writehistory[1]  ==  r2) && 3'b001)
     || ((writehistory[2]  ==  r2) && 3'b010)
     || ((writehistory[3]  ==  r2) && 3'b100);
```

The result of these operations will be unique identifiers showing the location of these registers. If the result is the binary sequence “000” we can conclude that the specific register is not in process anywhere.

Our current system does not simultaneously execute multiple instructions that write to the same destination register. As a result, the binary sequence contained in r1loc or r2loc will be orthogonal to each other each representing a single execution unit. Given an instruction (i1) containing two source

registers (r1 and r2) we are not yet be able to schedule this instruction within the pool of resources as the intersection of r1loc and r2loc will be uniquely equal to “000”.

In order to use binary set operations to locate an instruction within the pool of resources we must instead consider the location of the register on the following clock cycle. As each execution unit has a unique orthogonal ID we can compute the subsequent locations of data as the union of adjacent IDs. The following table summarizes the results of this process for the 3 execution unit case.

Current Register Locations	Register Locations Next Cycle
001	011
010	111
100	110

Given two source registers (r1 and r2) we can now locate them in the system on the following clock cycle using the pseudo code

```
r1loc =   ((writehistory*1+ == r1) && 3’b011)
       || ((writehistory*2+ == r1) && 3’b111)
       || ((writehistory*3+ == r1) && 3’b110);
r2loc =   ((writehistory*1+ == r2) && 3’b011)
       || ((writehistory*2+ == r2) && 3’b111)
       || ((writehistory*3+ == r2) && 3’b110);
```

Again, the resultant sequence “000” indicates that the register is not in process.

We can now consider the case of an instruction (i1) containing two registers (r1 and r2) that may or may not both be in process within the system. A set of allowable execution units containing valid results for r1 and r2 can be obtains through the intersection of the results from the above equation.

```
i1loc = ((r1loc == 3’b000) ? 3’b111 : r1loc) &&
        ((r2loc == 3’b000) ? 3’b111 : r2loc);
```

The ternary operation is included in the above expression because a “000” sequence indicates that the specific register is not in process within the system and thus it can be scheduled without restriction. The only restrictions in that case would be for the other non-zero register location.

The result from this operation will be some combination of execution units for which the instruction is permitted. In the case we have been considering, a network of three execution units with connections between adjacent neighbors, the result will never be “000” as the center execution unit will contain all results for the previous set of instructions.

7.2. Scheduling Algorithm Analysis

Given a single instruction that has two source registers that are both within the system currently, our ability to locate and use them both in a single execution unit during the next clock cycle depends entirely upon their location within the processor fabric. In the current topology, our execution units only receive the results of adjacent processing elements. This interconnection scheme has the obvious consequence that the further apart the registers are physically, the less likely they are to be available. The maximum distance between execution units is a direct function of the number of units available and thus we expect the probability of scheduling an instruction to decrease with the number of execution units.

The plot below shows the results of combinatorial analysis to determine the probability of scheduling a single instruction in the processor fabric. Issuance of multiple instructions will obviously lead to scenarios where issuance is only permitted to conflicting resources. Look ahead scheduling of instruction locations could be used to maximize the probability of issuance.

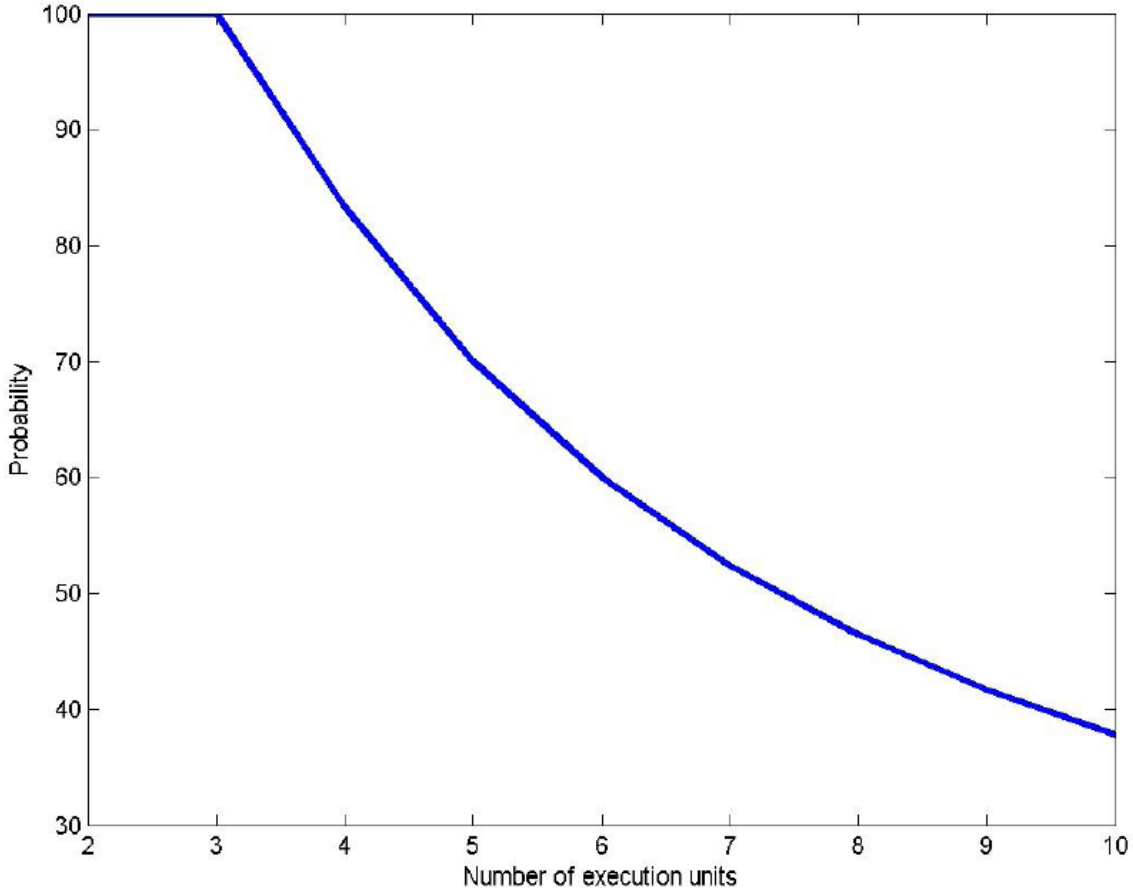

Figure 7-2 Probability of being able to schedule an instruction with two registers current in system

7.3. Application of Scheduling System for 3 Exec Modules

As we saw in the graph above, the system with three execute modules has a topology such that all modules will share register values at play in the system (which also holds true for the 2 module case). The three module configuration is the simplest and most compact configuration and, as such, we will delve into the process of scheduling instructions in this system.

As we saw previously, the location of a register within the system can be found using simple binary arithmetic:

```
r1loc = ((writehistory[1] == r1) && 3'b011)
     || ((writehistory[2] == r1) && 3'b111)
     || ((writehistory[3] == r1) && 3'b110);
r2loc = ((writehistory[1] == r2) && 3'b011)
     || ((writehistory[2] == r2) && 3'b111)
     || ((writehistory[3] == r2) && 3'b110);
```

These registers may not be present and as such we can account for that in the following manner:

```
i1loc_temp = ((r1loc == 3'b000) ? 3'b111 : r1loc)
          && ((r2loc == 3'b000) ? 3'b111 : r2loc);
```

This will tell us the locations at which an execute instruction can be scheduled within the system. For the case where we wish to execute three instructions we will have a total of three of these values

```
i1loc_temp = ((r1loc == 3'b000) ? 3'b111 : r1loc)
          && ((r2loc == 3'b000) ? 3'b111 : r2loc);
i2loc_temp = ((r3loc == 3'b000) ? 3'b111 : r3loc)
          && ((r4loc == 3'b000) ? 3'b111 : r4loc);
i3loc_temp = ((r5loc == 3'b000) ? 3'b111 : r5loc)
          && ((r6loc == 3'b000) ? 3'b111 : r6loc);
```

Our processor is of heterogeneous design with both execution, branch and memory modules. It is likely that the above instructions will not be targeted for execution. In that case, we will need an additional layer of logic to account for the possibility. To account for this possibility we have created functions which look at the opcode in the instruction in order to determine its type.

```
e1 = isExecInstr( instr1 );
e2 = isExecInstr( instr2 );
e3 = isExecInstr( instr3 );
```

Things are further complicated in that sequential instructions may be dependent upon one another. For instance, in the case where the result of instr1 is used as an argument in instr2 we are not allowed to execute in parallel. To account for this we have defined the functions conflictFree2 and conflictFree3.

The function conflictFree2 checks to ensure that instr1.rdst is contained no where within instr2. Similarly, the function conflictFree3 checks to ensure that instr1.rdst and instr2.rdst are contained no where within instr3. For our purposes we always give priority to instr1 and as such we do not check to see if it is allowed at this point.

```
valid1 = True;
valid2 = conflictFree2( instr1, instr2 );
valid3 = conflictFree3( instr1, instr2, instr3 );
```

We can apply these results to our scheduling algorithm as follows:

```
i1loc = ( valid1 && e1 ) ? 3'b111 : i1loc_temp;
i2loc = ( valid1 && e1 ) ? 3'b111 : i1loc_temp;
i3loc = ( valid1 && e1 ) ? 3'b111 : i1loc_temp;
```

As before, the binary sequence "111" is employed to allow for scheduling without restriction. Once all this logic is evaluated we can schedule instructions as follows:

```
if      ( (i1loc[0] == 1'b1) && (i2loc[1] == 1'b1) && (i3loc[2] == 1'b1) )
       exec1.put( instr1 ); exec2.put( instr2 ); exec3.put( instr3 );
else if ( (i1loc[0] == 1'b1) && (i3loc[1] == 1'b1) && (i2loc[2] == 1'b1) )
       exec1.put( instr1 ); exec3.put( instr2 ); exec2.put( instr3 );
else if ( (i2loc[0] == 1'b1) && (i1loc[1] == 1'b1) && (i3loc[2] == 1'b1) )
       exec2.put( instr1 ); exec1.put( instr2 ); exec3.put( instr3 );
else if ( (i2loc[0] == 1'b1) && (i3loc[1] == 1'b1) && (i1loc[2] == 1'b1) )
       exec2.put( instr1 ); exec3.put( instr2 ); exec1.put( instr3 );
else if ( (i3loc[0] == 1'b1) && (i1loc[1] == 1'b1) && (i2loc[2] == 1'b1) )
       exec3.put( instr1 ); exec1.put( instr2 ); exec2.put( instr3 );
else if ( (i3loc[0] == 1'b1) && (i2loc[1] == 1'b1) && (i1loc[2] == 1'b1) )
       exec3.put( instr1 ); exec2.put( instr2 ); exec1.put( instr3 );
else if ( (i1loc[0] == 1'b1) && (i2loc[1] == 1'b1) )
       …. // omitted for brevity
```

This portion of the issuance stage is essentially a rote enumeration of all possible combinations. There is likely a more efficient way to schedule the instructions within the processor fabric, but we have yet to devise such a solution. The scheduling task is quite tedious to implement with single cycle operations in combinatorial logic. Additionally, we expect that the vast majority of FPGA resources utilized by our processor will be in the scheduling stage and that the resource requirements will grow substantially with the number of execution units in use. (It is worth noting that this logic almost entirely evaporates for the two execution unit case or any topology where each exec unit is connected to all other exec units within the fabric.)

7.4. Hardware Details

The following diagram is an abstraction of our design as it does not show the scoreboard that we have implemented. Further, this diagram is an enhancement our implementation as the decode and parallelize stage feed into a network of processing elements. These processing elements are responsible for routing the results of the computation out of the network to the write back stage.

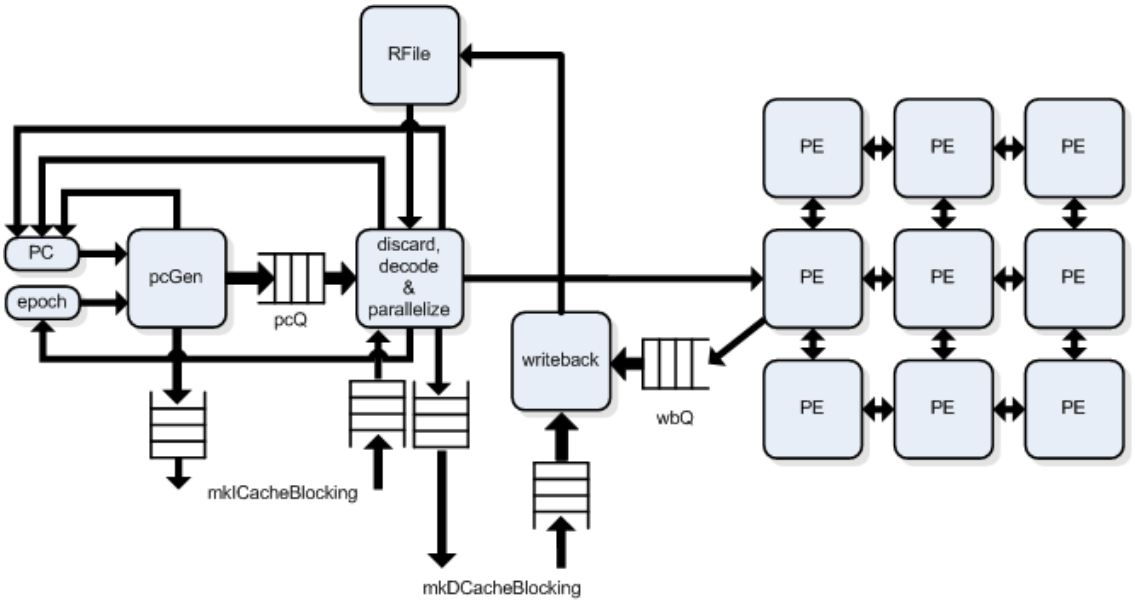

Figure 7-3 Potential Future Design Revision

We can extend this implementation by sharing the network of resources among multiple processor elements. The diagram below shows an abstraction of our redesigned processor which contains multiple processing elements (ALUs) that are fed instructions from an Execution Control Hub (XCH or ECH). This control unit will be responsible for fetching instructions from a Memory Management Hub (MMH) and relaying them to a Process Control Hub (PCH). As we have seen, the PCH will distribute the instructions to a network of compute resources which are effectively abstracted away from the ECH. This abstraction provides an opportunity for design exploration as well as a mechanism for potentially scaling the processor to an arbitrary degree.

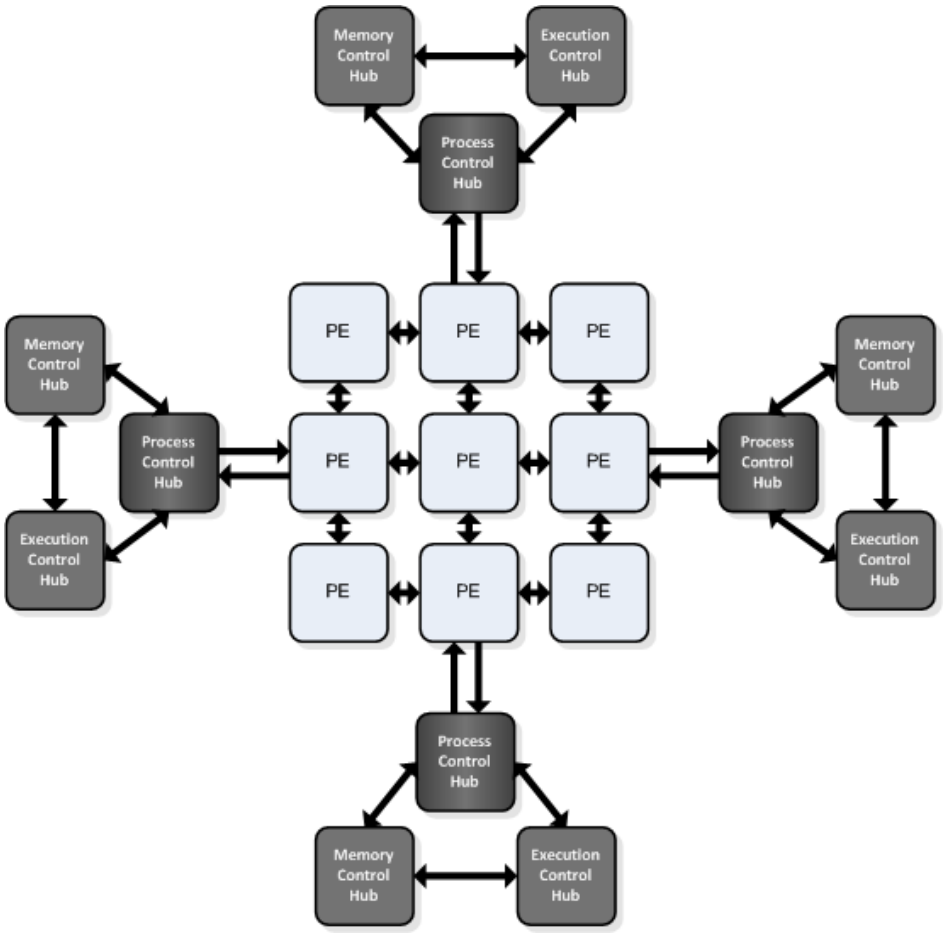

Figure 7-4 Long-Term Architectural Goal

8. Bibliography

1. **Hennessy, John L., Patterson, David A.** *Computer Architecture: A Quantitative Approach Third Edition.* San Francisco : Morgan Kaufmann Publishers, 2003.

2. *Alternative Implementations of Two-Level Adaptive Branch Prediction.* **Yeh, Tse-Yu and Patt, Yale N.** s.l. : The 19th Annual International Symposium on Computer Architecture, 1992.

3. **Lam, Normal, Si-En Chang, et al.** Evaluating the Performance of Dynamic Branch Prediction Schemes with BPSim. [Online] [Cited: 4 9, 2010.] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.9302&rep=rep1&type=pdf.