CHALMERS

Implementation and Verification of a 7-Stage Pipeline Processor

Master's Thesis in Embedded Electronic System Design

KARTHIK MANCHANAHALLI RAJENDRA PRASAD

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden 2015
Master's Thesis 2015:3

Implementation and Verification of a 7-stage Pipeline Processor

Karthik Manchanahalli Rajendra Prasad

Examiner: Per Larsson-Edefors

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden March 2015

Abstract

This report details the implementation of a 7-stage processor pipeline using VHDL. The report excludes discussion on instruction and data caches. The pipeline stages are balanced with respect to timing. The pipeline design is verified using embedded microprocessor benchmarks. The synthesized pipeline design is evaluated in terms of timing, area, and power. Implementation and evaluation of the branch predictors are emphasized, as they are vital part of a processor pipeline. Prior to branch predictor implementation, several branch predictor configurations have been evaluated, for their performance, through simulations using SimpleScalar tool. Based on the simulation results, few best performing predictors have been implemented and verified by integrating them into a processor pipeline.

The pipeline design is synthesized using Cadence Encounter RTL compiler in order to extract area and power estimates. Based on the synthesis results, evaluation of the pipeline and its function units has been carried out. The multiplier and the branch predictor unit are identified as the most critical with respect timing. Solutions have been suggested to improve the timing balance between the pipeline stages. It has also been evaluated that even though the branch predictors contribute for a significant improvement in the performance of a pipeline, they also account for $\approx 45\,\%$ of total area and $\approx 65\,\%$ of total power of the pipeline. The effect of caches on the pipeline timing, area, and power is not considered for evaluations.

Acknowledgments

I would like to thank the following people for their contribution and support.

- My examiner and supervisor, Per Larsson-Edefors, at Chalmers University of Technology for his constant support throughout my thesis.
- My supervisor, Alen Bardizbanyan, at Chalmers University of Technology for his technical support throughout my thesis.
- My classmates, Fredrik Brosser, Christoffer Fougstedt, and Jesper Johansson, for their contribution towards implementing certain parts of the pipeline design.

Karthik Manchanahalli Rajendra Prasad, Gothenburg, 2015/03/25

Contents

List of Figures

List of Tables

1

Introduction

Power dissipation in a processor has been a concern for computer architects in recent times. Since the power is directly proportional to the frequency of operation, increasing the speed of a processor will in turn result in increased power dissipation. The generated heat will have a negative impact on the electrical characteristics of a semiconductor. Increasing temperature will decrease the operating speed and increase the leakage in a transistor. For instance, a super computer consumes an average power of $6-10\,\mathrm{MW}$ [1]. A lot of heat is generated when several of these computers are operating simultaneously. This scenario is very common in data centers. The heat generated needs to be disposed of, because it can adversely affect the functionality and the lifetime of the circuit components. Money is spent not only to power these devices, but also to keep them cool. Hence power dissipation is one of the major limiting factors in improving the performance of a processor.

In recent years, computer architects have moved towards multi-core processor architecture to combat the increasing power dissipation and still improve the performance of a processor at the same time. Multi-core processors have achieved this goal by limiting the clock speed and executing the instructions in parallel. Even though the multi-core architecture has contained the power dissipation to an extent, it poses new challenges of its own such as power dissipation in inter-processor communication. Currently, a lot of research is going on in the field of multi-core architecture in order to make future processors energy-efficient. The power efficiency of a multi-core architecture is mainly determined by the power efficiency of a individual cores [2].

1.1 Motivation

Efficiency is an important factor when designing a processor pipeline. A pipeline is said to be efficient with respect to timing when each of its stages takes almost the same time

to execute. A classic example of an in-order processor pipeline is a 5-stage pipeline [3]. One of the major drawbacks of this pipeline is that its pipeline stages are not evenly balanced with respect to timing. For instance, since the instruction fetch stage needs to access the memory to fetch the instructions, this stage takes twice as much as time the decode stage or the write back stage takes to execute. This drawback limits the clock speed of the pipeline because the clock speed of a pipeline is determined by the stage with the longest delay. Hence, a 5-stage pipeline is not efficient with respect to timing. This issue can be overcome by further pipelining the stages with longest delays such as the instruction fetch and data memory access stage. This will result in increased clock speed and a uniform time balance between the pipeline stages. Hence, in order to achieve a timing balance between the pipeline stages, a pipeline with additional number of stages as compared to the 5-stage pipeline is necessary.

1.2 Problem Statement

Currently, the VLSI research group at Computer Science and Engineering department of Chalmers University of Technology uses a 5-stage processor pipeline to make evaluations on implementation aspect such as area and power consumption. As described before, the 5-stage pipeline has its own limitations. In addition to these limitations, the 5-stage pipeline doesn't include a branch predictor. A branch predictor is necessary in order to eliminate the branch misprediction penalties and to improve the performance of the pipeline. In order to carry out the evaluations a pipeline with an additional number of stages such as a 7-stage pipeline is necessary. This pipeline design will allow instruction and data cache accesses to happen across two cycles, which in turn will lead to higher clock rates.

Since there is no publicly available design of a 7-stage processor pipeline, the pipeline needs to be designed form the ground up. Several design issues need to be addressed while building a pipeline from scratch. By doing so, a better understanding on the internal working of the pipeline is gained. This insight can be used to fine tune the pipeline to achieve better performance. Customization is another reason why the pipeline needs to be built from scratch. The pipeline can be designed, implemented, and optimized for the required purpose. This report will present the implementation of a 7-stage processor pipeline, which will be used in future research studies.

Project Background

The implementation of the 7-stage processor pipeline started with the implementation of the branch predictors, as part of a project course in Embedded Electronic System Design (MPEES) program. Prior to my master's thesis, I and Fredrik Brosser were part of this project course. We had simulated different branch predictor configurations and implemented a configurable VHDL model of a branch predictor. In the same project course, Christoffer Fougstedt and Jesper Johansson were involved in implementing the

backend of a 7-stage processor pipeline. This thesis work will include some of their work to implement a 7-stage processor pipeline.

1.3 Thesis Objective

The goals for this masters thesis work are as follows

- Integration and verification of the branch predictor module in an existing 5-stage processor pipeline.
- Implementation and verification of a 7-stage processor pipeline.
- Balancing the 7-stage pipeline with respect to timing.
- Evaluation of the pipeline in terms of timing, area, and power.

1.4 Related Work

The implementation of processor pipelines has been dealt with since long time. One well known MIPS processor is R4000 [4], based on a 64-bit architecture and eight pipeline stages. The R4000 has three pipeline stages dedicated to data memory access as compared two pipeline stages in a 7-stage pipeline. OpenRISC [5] project is an open source community providing implementations of RISC instruction set architecture. The processor architectures provided by this community are advanced and used for commercial (derivatives of OpenRISC architecture) and academic research purposes. For instance, OpenRISC 1000 is a full 32/64-bit architecture with support for DSP and floating-point instructions.

The branch predictors have a huge impact on the performance of the pipeline. The design of the branch predictors is a well explored area in the processor community, with notable research being done in the early 1990's. Trade-offs related to the BTB (Branch Target Buffer) design and implementations were explored early on [6]. Yeh et al. [7, 8] focused on describing and investigating two-level adaptive branch predictors and their implementation, discussing the possible variations of two-level branch predictors and defining the different types according to the manner in which the table content is kept (Global-Private-Set). This classification yielded variations of the two-level adaptive predictor, e.g., GAg (Global-And-global) or PAg (Private-And-global). These variations will be discussed more in detail in Chapter 2. McFarling looked at more complex predictors and the possibility of combining global and local predictors for improved accuracy [9]. Later, Jimenez concluded that the hardware cost of many complex predictors do not show a proportionate performance increase [10]. Instead Jimenez presented a simple branch predictor design approach.

1.5 Report Outline

The remaining of the report is organized in the following manner.

Chapter 2 details some of the basic concepts in the field of computer architecture. It also describes the terminology used in this report.

Chapter 3 gives a brief description of the method used to carry out the branch predictor simulations at an architectural level using SimpleScalar tool. The results of the simulation are also presented and discussed in this chapter.

Chapter 4 describes the implementation of the 7-stage pipeline. Implementation of each stage of the pipeline is described in detail.

Chapter 5 describes the verification process involved in verifying the implementation of the Branch predictor and the 7-stage pipeline. It details the testbench setup, which is the 5-stage pipeline, used to initially verify the implementation of the branch predictor. It also describes the method and tools used in the verification process.

Chapter 6 presents the post-synthesis results of the 7-stage pipeline. The evaluation of the pipeline in terms of timing, area, and power is described.

Chapter 7 presents the conclusion of the thesis work. It also presents few future work on the pipeline that can be carried out in order to improve the performance of the pipeline.

2

Technical Background

Pipeline is a vital part of a processor that performs data processing. A pipeline consists of instruction and data memories, registers, and functional units such as ALU. The pipeline implemented here is based on a MIPS 32-bit Instruction Set Architecture (ISA). A pipeline is basically a hardware implementation of an ISA.

An ISA can be classified into two main categories, Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC) [2]. A RISC architecture is built with an intention to simplify the decoding process of the instructions, thereby reducing its cycle time. In RISC architecture, the instruction size is kept constant, with regular formats, which makes the pipelining of the instruction easier. All instructions are executed in a single cycle. In contrast to RISC architecture, the instructions in CISC architecture are designed for variable length and formats. The intention is to make the instructions more compact and use fewer instructions to execute a fragment of code. This results in better utilization of the caches. The MIPS ISA used here is a RISC architecture developed by MIPS Technologies [11]. There are multiple versions of MIPS ISA available, but MIPS32 version is used.

2.1 MIPS Instruction Set Architecture

According MIPS ISA, the length of a instruction is constant, which is 32-bit in this case. Based on functionality, the instructions can be classified into following groups [2].

- Arithmetic/Logic Instructions
- Memory Access Instructions
- Control Instructions

Arithmetic/Logic Instructions

Arithmetic instructions include addition, subtraction, and multiplication operations. Depending on the operands the arithmetic operation can be either signed or unsigned. Currently, the pipeline is designed to handle only integer arithmetic operations. Logic operations include bit-wise AND, OR, NOR, and XOR. All arithmetic and logic instructions, except for multiplication, take one clock cycle to execute. Multiplication instructions, depending on the design of the multiplier, take two or more clock cycles to execute.

Memory Access Instructions

Memory access instructions include load and store operations. A load operation will load a word/byte of a data from the memory. A store operation will store a word/byte of data to the memory. Depending on the availability of the requested data in the cache, the memory instructions take several clock cycles to execute.

Control Instructions

Control instructions include the conditional branch and jump operations. These instructions break the sequential execution of the instructions and jump to a target address and starts executing instructions from there. In case of branch instructions, the breaking of the sequence is based on a condition. For instance, if the value in a register is equal to (less than, greater than) zero then break the sequence. Whereas jump instructions break the sequence unconditionally. The target address is encoded in the instruction.

2.2 Instruction Format

The size of the instructions in MIPS architecture is constant. All instructions are encoded into a 32-bit word format. The format of the instructions can be classified into three groups, Register (R-type), Immediate (I-type), and Jump (J-type) instructions formats. These formats are shown in Figure 2.1, 2.2, and 2.3. The description of the fields in these instruction formats are given in Table 2.1. All ALU instructions operate on the registers, results written to the registers. Memory instructions such as Load and Store transfer data between the memory and the register.

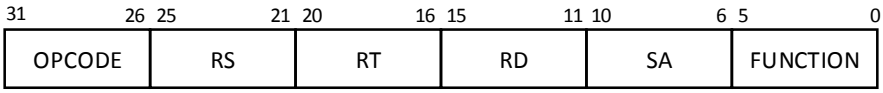

Figure 2.1: Register (R-type) instruction format

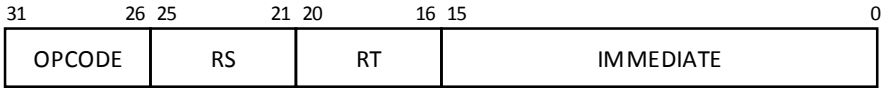

Figure 2.2: Immediate (I-type) instruction format

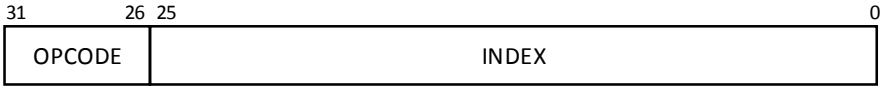

Figure 2.3: Jump (J-type) instruction format

2.3 7-Stage Pipeline

Figure 2.4 shows the block diagram of a 7-stage processor pipeline. The operations such as branch prediction, instruction/data cache access, and multiplication, are pipelined for two-cycle access. Since these operations consume more time, pipelining them into more number of stages will reduce the delay in each stage. The following paragraphs will briefly describe each stage of the pipeline.

Instruction Fetch (IF)

This stage mainly includes the branch prediction and the instruction cache access. The branch predictor consists of a Branch Target Buffer (BTB) and Branch Direction Predictor (BDP). The BTB stores the address of the branch target. The structure of the BTB is similar to a cache (set-associative) and can be implemented using SRAM cells. The operation of the BTB is pipelined for two clock cycles, starting with indexing the BTB in the first clock cycle followed by the tag comparison in the second clock cycle. The branch direction predictor operates in the second clock cycle and predicts whether the branch should be taken or not. The results from both the BTB and the direction

Table 2.1: Description of the fields in the instruction format

Field	Description
Opcode	6-bit code, which defines the operation of the instruction
RS	5-bit index used to access the source register
RD	5-bit index used to access the destination register
RT	5-bit index used to access the target (source/destination) register
SA	5-bit code, which specifies the shift amount
Function	6-bit code, which defines the functionality of the operation
Immediate	16-bit signed operand used for immediate operations
Index	26-bit index used to form the target address for jump instructions

predictor is used to determine the direction of the branch. The branch misprediction penalty is 2 clock cycles, considering that the branch instruction is followed by the branch delay slot instruction as per MIPS. The instruction cache will be implemented for two cycle access with way prediction. In the first clock cycle, one of the ways, predicted by a way-predictor, of a set-associative cache is indexed. In the second clock cycle, tag comparison is carried out to determine the data hit or miss. In case of way misprediction, the pipeline will be stalled and all other remaining ways of the instruction cache is accessed.

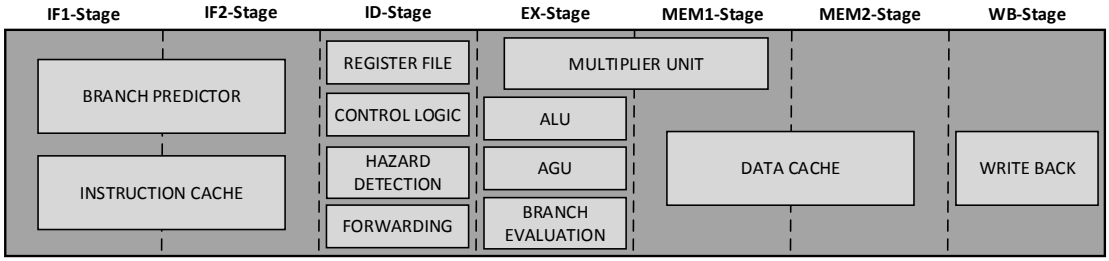

Figure 2.4: Block diagram of a 7-stage processor pipeline

Instruction Decode (ID)

This stage mainly deals with decoding the instructions and handling the data dependencies between the instructions. The control unit decodes the instruction and generates the necessary control signals. The hazard detection unit detects the data hazards in the pipeline and stalls the pipeline if necessary. For instance, an instruction following a load instruction and using the result of the load instruction can cause a data hazard, because the result of a load instruction is obtained only in MEM2-stage. So the instruction in the decode stage waiting for this result needs to be stalled. The forwarding unit detects data dependencies and controls the forwarding multiplexers. The register file contains 32 general purpose registers.

Execute (EX)

This stage mainly consists of an ALU, an Address Generation Unit (AGU), a multiplier unit and a branch evaluation unit. All arithmetic instructions are executed in this stage. The memory address for the load and store instructions is calculated by the AGU. The multiplier unit is one of the units with longest delay, so it is pipelined into two stages. All branch instructions are evaluated in this stage. In case of a branch misprediction, the speculative instructions in the pipeline are flushed and the program counter is loaded with the correct target address.

Data Memory Access (MEM)

In this stage the data cache is accessed to read/write data. The access to the data cache is pipelined for two cycle access.

Write Back (WB)

This is the final stage of the pipeline and it handles the writing to the registers.

2.4 Hazards

Hazard is a situation when an instruction cannot be executed in a pipeline. Hazards in a pipeline can be classified into three categories.

Structural Hazard It occurs in case of a single instruction and data L-1 cache, when two pipeline stages are trying to access the same cache in the same clock cycle. As per the MIPS ISA there is a separate instruction and data cache, hence there will be no structural hazard.

Data Hazard It occurs when an instruction cannot be executed because of its dependency on previous instruction result.

Control Hazard It occurs when the instruction fetched cannot be executed because of the change in flow of the instructions.

Data Hazard

There is always a data dependencies on register or memory operands. Since in the 7-stage pipeline the load and the store instructions are executed in the memory stage and they arrive at the memory stage in the process order, data dependencies on the memory operand don't cause data hazard. In case of register operands the data is written to the register in the write back stage and read from the registers in the decode stage, hence in some cases it might lead to data hazards. Data hazards on the register operands can be classified into following four types.

Read After Write (RAW) A current instruction to be executed needs the result of the previously executed instruction. If the current instruction doesn't wait until the result of the previous instruction is written into the register file then the value read by the current instruction will be wrong.

Write After Read (WAR) A current instruction writes a value to a register location before the previous instruction reads the value from this register location. This scenario is not possible in the 7-stage pipeline as the instructions flow in the process order.

Write After Write (WAW) A current instruction writes a value to a register location before the previous instruction writes a value to the same register location. This scenario is similar to WAR hazard scenario. Since the instructions are executed in the process order, a current instruction can't write to a register location before its previous instruction does.

RAW data hazard can be solved by forwarding the results from the intermediate stages. Forwarding will be discussed in detail in Chapter 4.

Control Hazard

Control hazard is the caused by branch or jump instructions. Branch instructions change the instruction flow based on a condition. In the 7-stage pipeline all the branch instructions are evaluated in the execute stage of the pipeline. Evaluating a branch instruction includes determining the branch direction and calculating the branch target address. Until the branch instruction is evaluated, the instructions are fetched assuming that the branch is not taken. In case the branch is taken, then the instruction in the previous stages will be flushed. Branch predictors are used to overcome the control hazard. Branch predictors predict the branch direction and target address. Branch predictors will be dealt in detail in Chapter 3 and Chapter 4.

3

Branch Predictor Simulation

Branch predictors are used to predict the direction and target address of a branch instruction. Branch predictors are a vital part of a processor pipeline. Figure 3.1 shows the block diagram of a part of a processor pipeline with a branch predictor integrated into the front end of the pipeline. When a branch instruction is mispredicted the instructions in the front end (IF and ID stages) of the pipeline is discarded. This results in increased execution time and wastage of power. For instance, in a classic 5-stage pipeline two instructions are discarded when a branch is mispredicted. Hence mispredicting branch instructions will give rise to severe processor performance penalties.

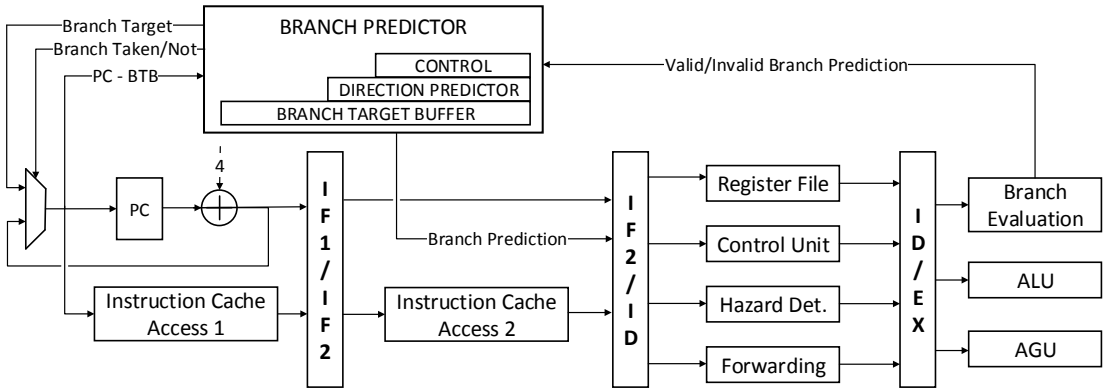

Figure 3.1: Branch predictor integrated into the front end of a processor pipeline

On the bright side, branch instructions, such as loops, often display patterns in the branch direction, which can be detected and used by the branch predictors. Furthermore, branch instructions often branch to the same target address repeatedly. By storing the target address in a buffer, it is possible to predict the target address. By doing this it is possible to design branch predictors with a high prediction accuracy. A typical branch predictor consists of a Branch Direction Predictor (BDP) and a Branch Target Buffer

(BTB). BDP predicts the direction of the branch instruction and BTB is used to store the previous target address of the branch instruction.

Simulations were carried out in order to determine the best branch predictor configuration for the 7-stage pipeline. The SimpleScalar simulator [12] is an open source pipeline simulator written in C. SimpleScalar is able to simulate the PISA and Alpha instruction set architectures and produces simulation statistics and results used to evaluate the pipeline configuration. SimpleScalar with the PISA instruction set along with the MiBench [13] and SPEC [14] benchmarks were used. The MiBench benchmark suite comprises 35 benchmarks written in C, representing typical embedded applications. The SPEC benchmark suite comprises 21 benchmarks. In addition to MiBench benchmarks SPEC benchmarks are used for evaluation because the SPEC benchmarks are bigger than the MiBench benchmarks. Evaluation of the branch predictor configurations using bigger benchmark is done in order to make sure that the branch predictors performance is independent of benchmark size.

Parameters of BDP, such as type and table size, and BTB, such as associativity and table size, are varied to determine their impact on the performance of the processor. In total 74 branch predictor configurations were simulated and evaluated. The results were compared to a theoretically perfect predictor with 100 % accuracy. To extract and compile the results perl scripts were used for automation. In these simulations we have chosen to limit our exploration to predictor table sizes in the range 64-4,096 branch entries. It is a tedious process to simulate all the branch predictor configurations using SPEC benchmarks. Hence a few of the best performing branch predictor configurations are selected by evaluating them using the MiBench benchmark suite and then the selected configurations are reevaluated using the SPEC benchmark suite.

Performance metrics used in the evaluation of the branch predictors are the Execution Time Ratio (ETR) and the prediction accuracy. Execution time is defined as the time taken to execute a benchmark in terms of clock cycles. To normalize the execution time a perfect branch predictor with 100 % accuracy is considered. Hence the ETR is defined as the ratio of the execution time using a particular branch predictor to the execution time using a perfect predictor. A branch predictor with a prediction accuracy of 100 % will have an ETR of 1. Hence in practice a branch predictor can have an ETR of $\geq$1. Besides ETR and prediction accuracy, standard deviation of the ETR and power is also considered as a performance metric. The standard deviation is considered in order to determine whether the branch predictor performance is consistent throughout the benchmarks. The power is obtained by using Wattch tool [15], which is tightly integrated with SimpleScalar. The power values given by the Wattch tool are not accurate, hence the power values are only used for elimination purpose.

Branch Direction Predictor

BDP predicts the direction of a branch instruction, whether the branch is taken or not. Two different kinds of BDP are evaluated, Bimodal and Two-Level predictor. Figure 3.2 shows the algorithm of a bimodal predictor. A bimodal predictor is a 2-bit counter - the Most Significant Bit (MSB) is used to predict the direction of the branch instruction, branch is taken if the MSB is 1 else not taken. The prediction is evaluated in the EX stage. Based on the evaluation result the counter is incremented if the branch is taken else decremented. An array of such 2-bit counters is implemented and indexed using the Program Counter (PC).

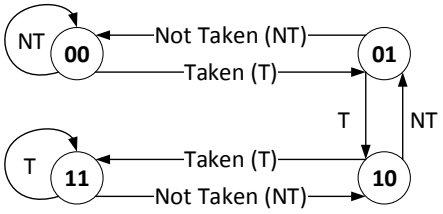

Figure 3.2: Algorithm of a bimodal predictor

A two-level predictor is similar to a bimodal predictor except for an additional level. Figure 3.3 shows a block diagram of a two-level predictor. The first level is an array of shift registers and the second level is an array of 2-bit counters. Two-level predictors will be discussed in detail in Chapter 4. Based on the size of the level-1 table (Branch History Table (BHT)) and level-2 table (Pattern History Table (PHT)), two-level predictors can be classified in several subtypes, as given in Table 3.1.

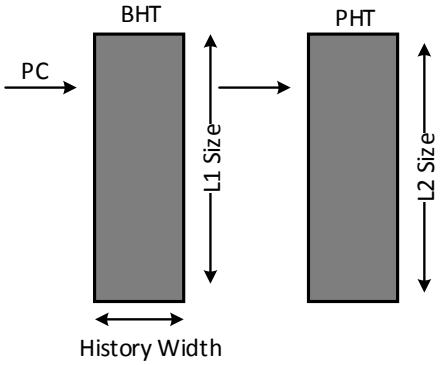

Figure 3.3: Two-Level predictor

The naming convention used in the plots are as follows, {Predictor Type}/{Level-1 Table Size}/{Level-2 Table Size}. Figure 3.4 shows the plot of ETR and the prediction accuracy of the selected branch predictor configurations using MiBench benchmarks. It

Table 3.1: Two-Level Predictor Configurations

Predictor	L1 Size	History Width	L2 Size	XOR
Global-And-global (GAg)	1	W	2^W	0
Global-And-private (GAp)	1	W	$> 2^W$	0
Private-And-global (PAg)	N	W	2^W	0
Private-And-private (PAp)	N	W	2^{N+W}	0
Gshare	1	W	2^W	1

is clear from the plot that the two-level predictors perform better than the bimodal predictors in terms of ETR and prediction accuracy. Figure 3.5 shows the plot of the standard deviation and estimated power of the selected branch predictor configurations. The standard deviation seems to be lower, hence the selected branch predictor configurations perform evenly over all the benchmarks. Even though the performance of the two-level predictors is better than the bimodal predictors, the estimated power of the two-level predictors is higher than the bimodal predictors.

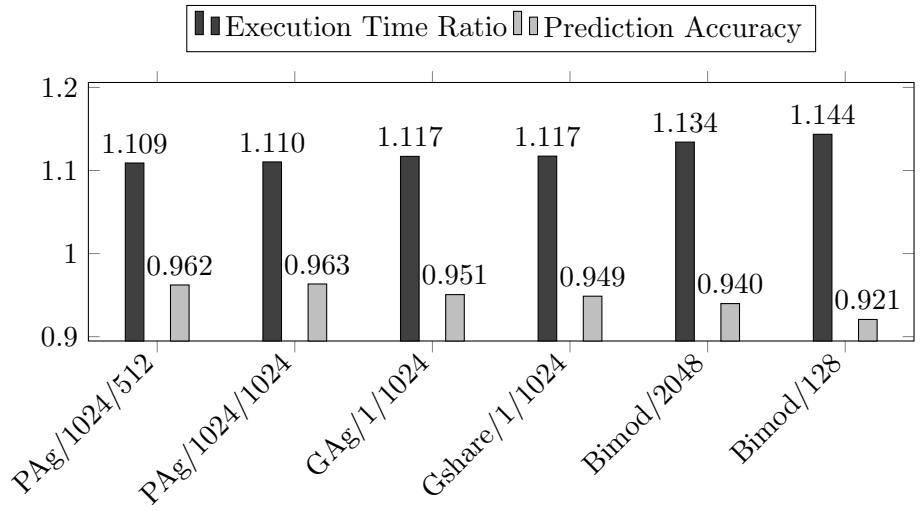

Figure 3.4: Execution time ratio and prediction accuracy of selected branch predictors obtained using MiBench benchmark suite

Figure 3.6 shows the plot of the ETR of selected branch predictor, which are reevaluated using SPEC benchmark suit. Comparing Figure 3.4 and Figure 3.6 it can be observed that the selected branch predictor performance is independent of size of the application. The two-level predictors still perform better than the bimodal predictors. Note that the BTB size and associativity are kept constant while only the BDP parameters are varied.

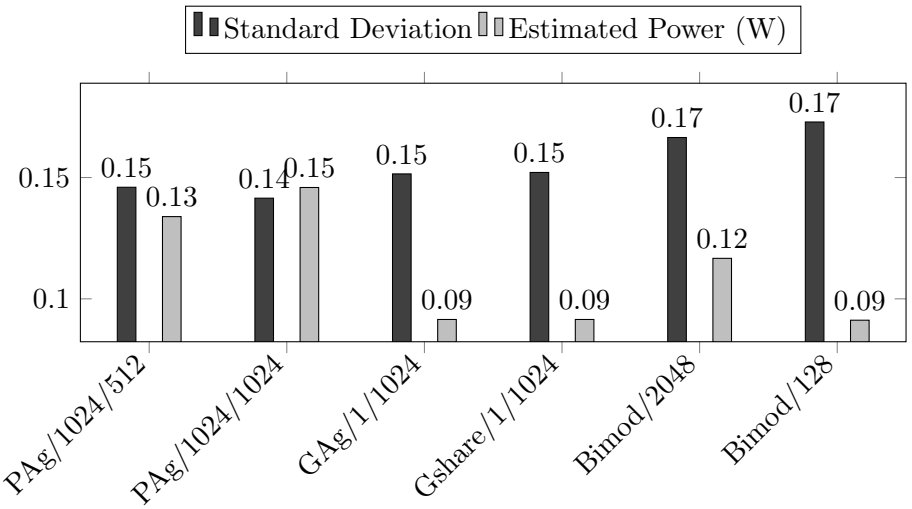

Figure 3.5: Standard deviation and estimated power (Wattch) of selected branch predictors obtained using MiBench benchmark suite

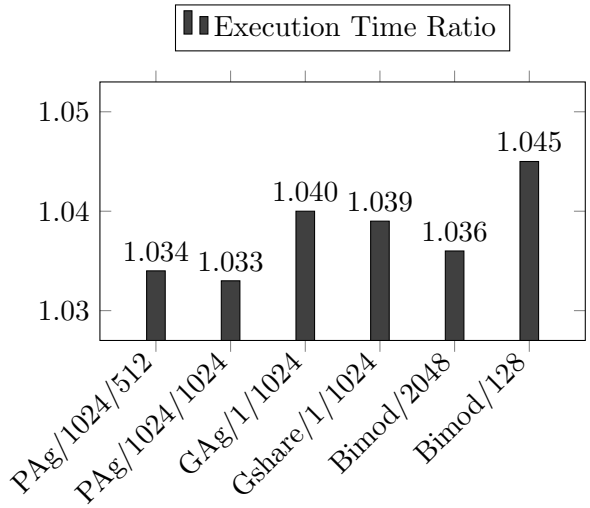

Figure 3.6: Execution time ratio of different branch predictors obtained using SPEC benchmark suite.

Branch Target Buffer

A BTB stores the target addresses of the branch instruction. The implementation of a BTB is similar to a cache. The table size and associativity of the BTB is varied in order to determine its influence on the performance of a processor. Figure 3.7 shows the variation in ETR with respect to different BTB table size and associativity. The Least Recently Used (LRU) is used as a replacement policy. Note that the BDP parameters

are kept constant while the BTB parameters are varied.

From Figure 3.7 it can be observed that the large BTB table size and high BTB associativity will result in the best performance in terms of ETR. On the other hand, increased table size and associativity will also results in significant increase in circuit complexity, area, and power. The predictor group that performed best in terms of accuracy and ETR in simulations are the large PAp-type predictors. However, it comes at a significant hardware cost compared to simpler predictors and the performance increase is not proportional to the increase in hardware cost. Bimodal, GAg, and Gshare predictors perform well in relation to the estimated hardware cost and power values. Based on the simulations results, large bimodal predictors (table size 2,048-4,096 entries) saturate at around 94 % direction prediction accuracy.

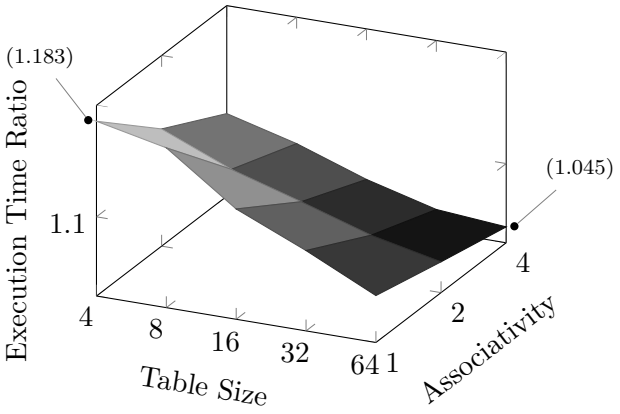

Figure 3.7: Performance for different BTB table sizes and associativities.

4

Pipeline Implementation

A VHDL model of a 7-stage in-order processor pipeline is implemented. The pipeline implementation is based on a MIPS32 ISA. It should be noted that the implementation of the instruction and data L-1 cache is not included in this report. This chapter will give a detailed description on the implementation aspect of each stage of the pipeline. Each stage is separated by pipeline registers, which are usually implemented using flip-flops. All pipeline registers are updated on the positive rising edge of the clock. There is no feedback within a single clock cycle in any of the pipeline stages.

Figure 4.1 shows the VHDL model of a 7-stage pipeline. Every clock cycle the PC is incremented and a new instruction is fetched from the instruction cache. The instruction and data caches used here are ideal, hence there will be no cache misses. The instruction is decoded, operands are fetched from the register file and appropriate control signals are generated. Generated control signals and fetched operands are propagated through the following stages. The control signals will determine the operation to be performed on the operands. Finally, the result of the operation will be written back to the register file if necessary. All instructions will go through all the stages of the pipeline even though some instructions will not include any specific operation related to that stage. For instance, an add instruction will not have anything to do with memory transfer, but still it will go through that stage. Even though it is a waste of clock cycles, the design of the pipeline will be simple and predictable.

4.1 Instruction Fetch

The instruction fetch (IF) stage is implemented as two stages in the 7-stage pipeline. Figure 4.2 shows the block diagram of the IF stage. It mainly includes an instruction cache and a branch predictor module. In a 5-stage pipeline, an instruction is fetched from the instruction cache (I-cache) in a single clock cycle. The operation includes in-

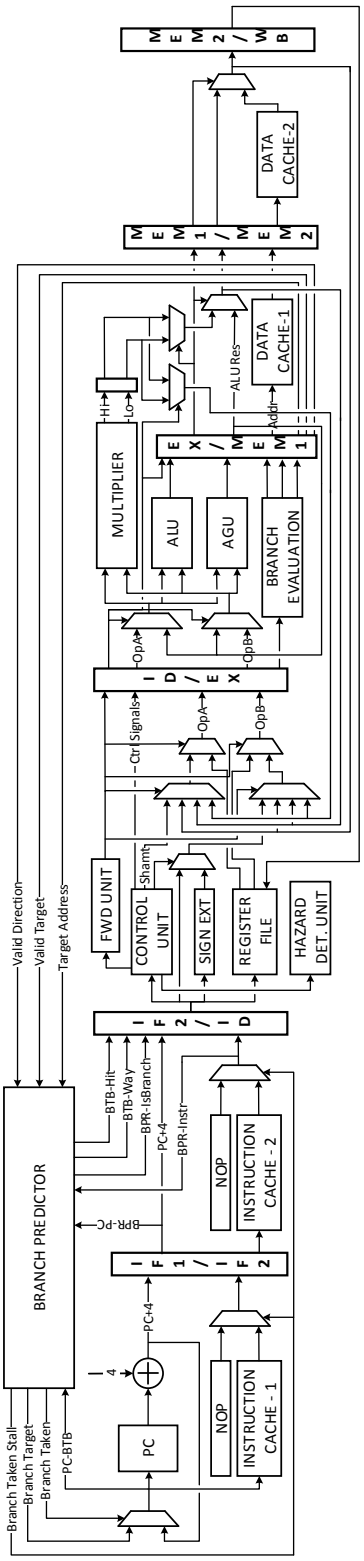

Figure 4.1: VHDL model of a 7-stage in-order processor pipeline

dexing the cache line, comparing tags, and generating hit signals. Since this operation is purely combinatorial, fetching instruction from the I-cache will have a long path delay. This path delay is almost twice the path delay of instruction decoding or writing to registers. Since the longest path delay of the pipeline determines the maximum clock speed at which the pipeline can be operated, the path delay of IF stage in the 5-stage pipeline limits the clock speed. Hence by dividing the instruction fetch operation into two stages, the path delay can be reduced and the clock speed of the pipeline can be increased.

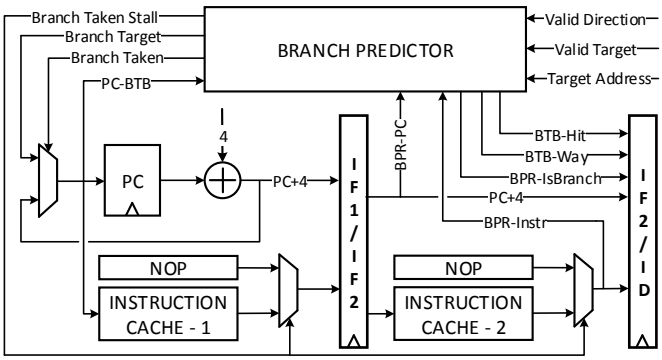

Figure 4.2: Block diagram of instruction fetch stages

The implementation of a BTB in the branch predictor module is similar to the implementation of a cache. Hence even the BTB will also have a long combinatorial path. By adding a pipeline register in the path, the path delay can be reduced. Since the implementation of caches is not discussed in this report, the main emphasis is given to the implementation of branch predictor.

The input to the program counter is either a target address from a branch instruction or an incremented value of the previous program counter value. Since the implementation is based on a 32-bit architecture and byte addressing method is used, the program counter is incremented by 4 bytes to address the next instruction. The multiplexer which feeds the program counter is controlled by the branch predictor. If the branch predictor encounters a branch instruction and predicts that the branch instruction should be taken, it switches the multiplexer to select the target address predicted by the branch predictor. The branch predictor also forwards its prediction results such as direction and target address, which will be used by the branch evaluation unit in the EX stage to determine the correctness of the prediction.

The BTB and the I-cache are implemented using SRAM cells. SRAM operates synchronously with the rising edge of the clock, hence all input data to the SRAM should be available (stable) before the rising edge of the clock. The rising edge of the clock to the SRAM latches the input data. In order to meet this requirement and respect

the setup time, the combinatorial value of the program counter is fed as a input to the branch predictor (BTB) and to the instruction cache. All branch instructions are evaluate in the EX stage. In case of a branch mis-prediction, the instruction in IF1, IF2, and ID stage, need to be flushed. Hence a multiplexer, controlled by the branch predictor, is implemented to flush the instruction. The following subsection will give a brief description on the implementation aspect of the branch predictor.

Branch Predictor

A configurable VHDL model of a branch predictor module is implemented. The module is designed with modularity and reconfigurability in mind. Configuration is done by setting generics at design time. Figure 4.3 shows the block diagram of the branch predictor module. The branch predictor module uses information from the IF stage to make a prediction in every clock cycle. It includes a Branch Direction Predictor (BDP), a Branch Target Buffer (BTB), a control, and a comparator module.

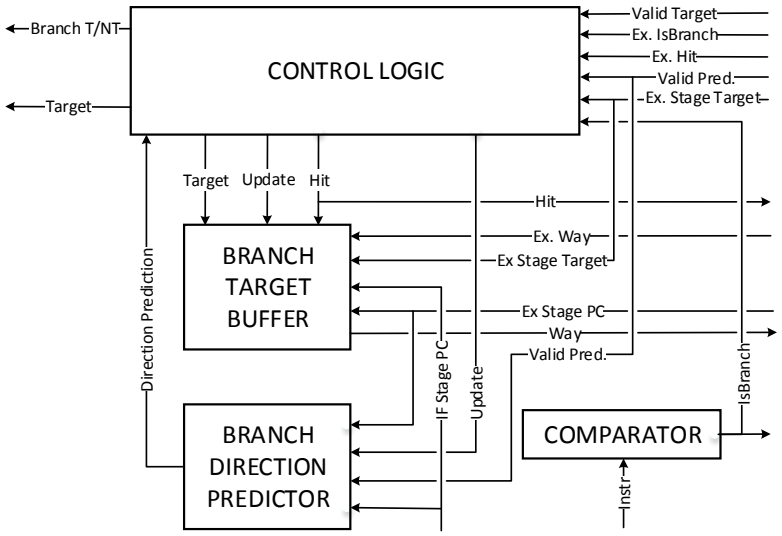

Figure 4.3: Branch predictor module block diagram

Control Logic

The control logic module is purely combinatorial. It takes the input from the BDP, BTB, and the comparator to make a prediction. The BDP predicts the direction of the branch. The BTB stores the target address. The comparator determines whether the instruction is a branch instruction or not. The input to the comparator is the instruction from the IF2 stage. The branch is taken only if all the following conditions are met.

- If the instruction is a branch instruction

- If BDP predicts that the branch should be taken
- If the target address is available in the BTB

The control module also receives the feedback information from the branch evaluation module in the EX stage. The feedback includes information such as whether the prediction is correct in terms of direction and target address and the correct target address, which is calculated in the EX stage. Based on this information the control module determines validity of the previous prediction. In case of invalid prediction, it flushes the instructions in the IF1, IF2, and ID stage, and loads the correct target address into the program counter register.

Branch Direction Predictor

Two types of branch predictors are considered in this work, Bimodal and Two-Level. Figure 4.4 and Figure 4.5 shows the implementation of a bimodal and a two-level branch predictor. A bimodal branch predictor is implemented as an array of 2-bit saturating counters. The working of a 2-bit counter is described in Chapter 3. The array is indexed using the program counter from the IF2 stage. Hence each branch instruction will have its own 2-bit counter. The MSB of the 2-bit counter is used for prediction.

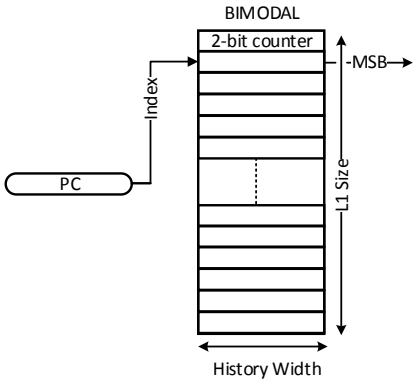

Figure 4.4: Block diagram of a bimodal branch predictor

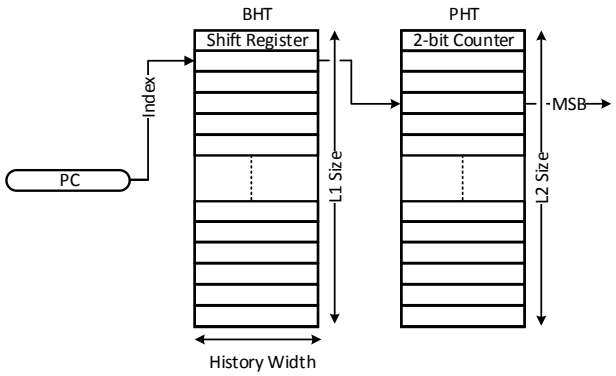

Figure 4.5: Generic two-level adaptive branch predictor

As the name suggests, a two-level branch predictor has two levels (tables), Branch History Table (BHT) and Pattern History Table (PHT). The implementation of a two-level branch predictor is similar to a bimodal branch predictor except for an additional level. A BHT is an array of shift registers. The BHT is indexed using the program counter from the IF2 stage. The shift register entry is left shifted every time a branch instruction is encountered. A '1' is inserted if the branch is taken else a '0' is inserted. The PHT is an array of 2-bit saturating counter and it is similar to bimodal branch predictor in functionality. The PHT is indexed using the content of the shift register. The branch

predictor design can be configured for type (two-level or bimodal branch predictor) and table sizes at design time. The branch predictor implementation uses register-based tables.

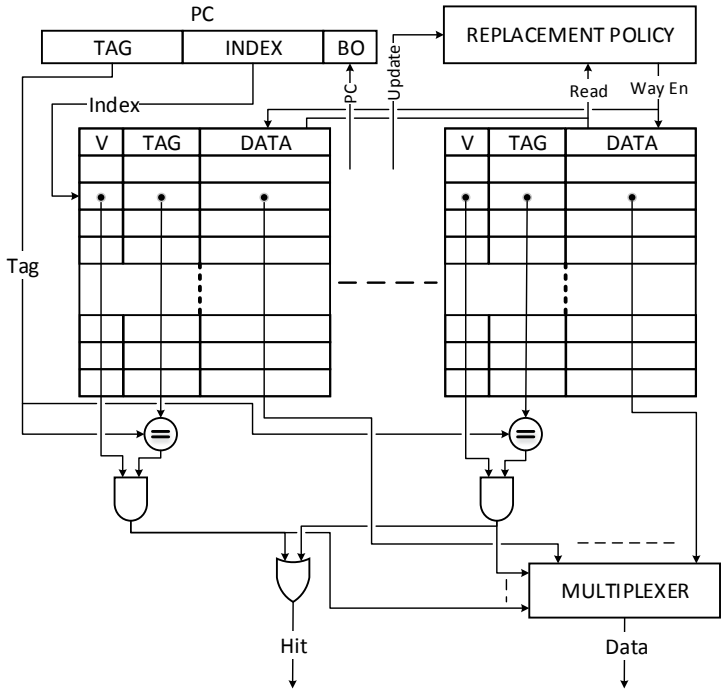

Figure 4.6: BTB block diagram.

Branch Target Buffer

The Branch Target Buffer (BTB) is used to store the target address of the branch instruction. As previously mentioned, the implementation of a BTB is similar to the implementation of a cache. Figure 4.6 shows the block diagram of a BTB. The BTB is implemented using SRAM cells and can be configured at design time. The configuration parameters are the number of sets and the set associativity. There are a number of fixed size SRAM blocks available, in sizes 32x32b, 128x32b, 512x32b, 1024x32b and 2048x32b. When synthesized, the BTB implementation will select the smallest SRAM block possible with the given configuration. For instance, both the BTB size 16 and 32 will use the 32x32b SRAM configuration. There is also a configuration variable for changing the replacement policy when using a set-associative BTB; selecting between LRU and Random replacement policy.

The BTB is indexed using the program counter from the IF1 stage. Each entry contains three fields, a Valid (V) bit field, Tag field, and a Data field. The valid bit field identifies the validity of the data present in the entry. Initially all valid bits will be cleared, after

which the valid bit field is set when a new data and tag is written into the entry. In case of multiple data stored within a data block then the Block Offset (BO) is used to select individual data from the block. Here data is the target address of the branch instruction. Since extracting data from the BTB will result in a long combinatorial path, BTB is implemented to operate in two stages. In the first stage the BTB is indexed using the program counter and in the second stage the data and the hit signals are generated.

4.2 Instruction Decode

The instructions fetched in the IF stage are decoded in the Instruction Decode (ID) stage. Figure 4.7 shows the block diagram of the ID stage. The inputs to the ID stage are the instruction, program counter and the prediction values from the branch predictor, which needs to be forwarded to the EX stage. The ID stage includes a control unit, forwarding unit, hazard detection unit, and register file. Besides these units, there is also a unit to sign extend the immediate value. The functionality of each of these units are described in the following subsections.

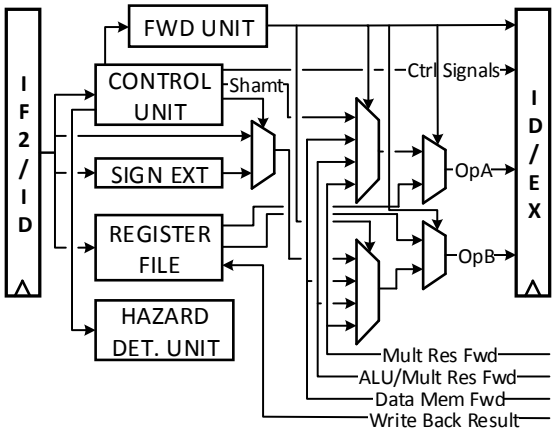

Figure 4.7: Block diagram of instruction decode stage

Control Unit

The control unit generates the control signals based on the instruction opcode. These generated control signals determine the operation to be performed by the different functional units. One such control signal is the ALU opcode as shown in Table 4.1. Based on the opcode and functional field in the instruction word the control unit encodes ALU operations into opcodes. The control unit also generates control signals such as register write, memory read, memory write, etc.

Table 4.1: Encoding ALU Operation

ALU Opcode	ALU Operation
0000	Addition (Signed)
0001	Addition (Unsigned)
0010	Subtraction (Signed)
0011	Subtraction (Unsigned)
0100	Bitwise AND
0101	Bitwise OR
0110	Bitwise XOR
0111	Bitwise NOR
1000	Shift Left
1001	Shift Right (Logical)
1010	Shift Right (Arithmetic)
1011	Set on less than

Forwarding Unit

Forwarding is used to overcome the stalling of the pipeline because of data dependencies between instructions. Figure 4.7 shows the two four-input multiplexers controlled by the forwarding unit. There are three forwarding paths from the different stages of the pipeline connected to the three inputs of these multiplexers. The first path (*Mult Res Fwd*) is the output of the multiplier. The second path (*ALU/Mult Res Fwd*) is the output of the multiplexer in the MEM1 stage, which outputs either the result from the ALU or the multiplier. The third path (*Data Mem Fwd*) is the output of the multiplexer in the MEM2 stage, which outputs either the result from the MEM1 or the data loaded from the data cache. The forth input to one of the multiplexers is the shift amount and to the other multiplexer is the sign extended immediate value. Even though forwarding paths reduce pipeline stalls due data hazards, they introduce long path delays which might limit the clock speed of the pipeline.

Hazard Detection

Adding forwarding paths in the pipeline will not eliminate all data dependencies. For instance, a load instruction, whose destination register is R3, followed by an addition instruction, whose source register is R3, will lead to a pipeline stall because the data is fetched from the data cache in MEM2 stage. Hence the pipeline needs to be stalled for at least 2 clock cycles in this scenario. If a load instruction is followed by a load delay slot instruction then the stalling of the pipeline can be reduced to a single cycle. It is a similar case when a multiplication instruction is followed by an instruction that uses

the result of the multiplication, then the pipeline needs to be stalled. The multiplication operation is usually pipelined for 2 stages. Hence the pipeline should be stalled for at least 2 clock cycles. It needs to be noted that when a pipeline is stalled only the instructions in the front end of the pipeline (IF1,IF2,ID) are stalled, but the instructions in back end of the pipeline (EX, MEM1, MEM2) will continue to execute.

Register File

The register file contains 32 General Purpose Registers (GPR), where each register is 32-bit wide. Figure 4.8 shows the block diagram of the register file. It also includes two read ports and one write port. The data is written to the register file at the rising edge of the clock and if the write enable is set. Writing to the register happens when an instruction reaches the write-back stage of the pipeline. As per MIPS ISA, GPR-0 is hard-wired to zero.

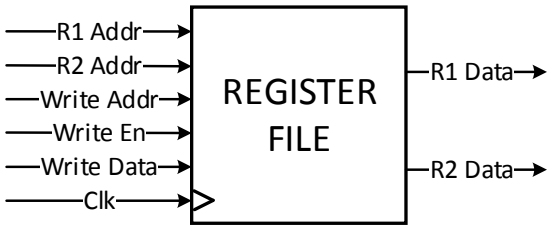

Figure 4.8: Block diagram of register file

4.3 Execute

All instructions are executed in the Execute (EX) stage of the pipeline. Figure 4.9 shows the block diagram of the EX stage of the pipeline. It includes a multiplier unit, an Address Generation Unit (AGU), an Arithmetic and Logic Unit (ALU), and a branch evaluation unit. The inputs to the EX stage are the two operands, the forwarding signals, and the control signals, which are used to enable the multiplier and to identify the specific operation that needs to be carried by the ALU.

Besides these functional units, there are two multiplexers which are used to select either the operands from the ID stage or the forwarding path, which is the previous result of the ALU. This forwarding path is necessary in case the current instruction needs the result of the previous instruction.

Multiplier

Several multiplier designs with different pipelined stages such as 2-stage and 3-stage pipelined multipliers have been evaluated for timing constraint. Multipliers are not implemented in this thesis work. Existing multiplier designs are integrated into the

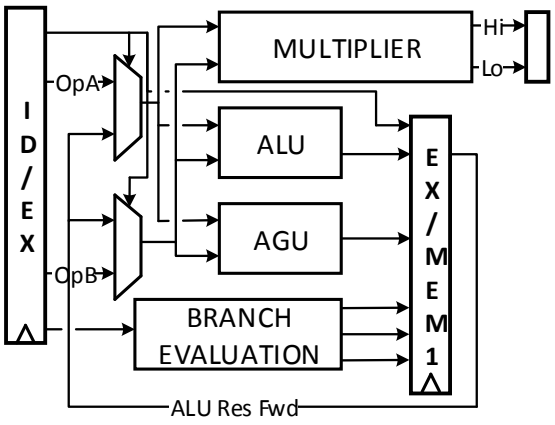

Figure 4.9: Block diagram of an execution stage

pipeline. The multiplier designs evaluated also include the DesignWare IP blocks from Synopsys [16]. Even though the preferred timing constraint can be achieved by using multipliers with more number of pipeline stages, adding more stages to the multiplier will result in more number of stall cycles due to data dependencies. Hence a trade-off is necessary between the preferred timing constraint and the performance penalty due to stall cycles. The final multiplier design chosen for the 7-stage pipeline will be based on timing constraint.

There are two special purpose register, LO and HI, which are specific to Multipliers. After multiplication, the higher word of the product is written to HO and the lower word of the product to LO register.

Arithmetic and Logic Unit

Table 4.1 shows the operations performed by the Arithmetic and Logic Unit (ALU) on the operands. The operation to be performed is selected by the control signals from the ID stage. The output of the ALU is fed back in the next clock cycle as a forwarding path and can be used as one of its operands.

The AGU unit calculates the memory address for the load and store operations. The AGU also receives the previously calculated ALU results as one of its operands. This forwarding path is controlled by the forwarding signals from the ID stage.

Branch Evaluation

All branch instructions are evaluated in EX stage. The branch evaluation unit receives the predicted signals such as predicted direction and target from the branch predictor. It evaluates the branch instruction and determines the correct branch direction and branch target address. This information is sent to branch predictor as a feedback. This feedback information is used by the branch predictor to determine the validity of its previous

prediction. In case the previous prediction by the branch predictor was an invalid prediction, the branch predictor reroutes the direction of the instruction flow to the correct direction. In case of valid prediction, the pipeline execution continues without any interruption.

The feedback information can be either fed back from the EX stage or it can be delayed and fed back in the next cycle, that is from the MEM1 stage. If the fed back path is from the EX stage then the path from the output of the ID/EX pipeline register to the input of the program counter, through the branch predictor, will be completely combinatorial. It will affect the timing of the pipeline because this path will have a long path delay. If the feedback path is from the MEM1 stage then the path from the output of the EX/MEM1 pipeline register to the input of the program counter, through the branch predictor, will be completely combinatorial. But, the path delay in case of feedback path from MEM1 stage will be less than the path delay in case of feedback path from EX stage. On the other hand, if the feedback path is set from the MEM1 stage, this will result in an addition of one clock cycle to the branch misprediction penalty.

The branch evaluation unit evaluates both the conditional and unconditional branch instruction, including unconditional jump, jump and link, and jump register instructions [17]. In case of conditional branch instruction, the effective target address is calculated by adding the 16-bit sign extended immediate value to the program counter value pointing to the instruction following the branch instruction. In case of unconditional jump instructions, the effective target address is calculated by left shifting the 26-bit index value by 2 bits and concatenating the resulting 28-bit value with the upper 4 bits of the program counter value pointing to the instruction following the jump instruction. In case of jump and link instruction, the effective target address is calculated in a similar way to the unconditional instruction and the return address, which is the address of the second instruction following the branch instruction, is written to GPR-31. In case of jump register instruction, the target address is in the register pointed out by the source register (RS).

4.4 Memory Access

The data memory access stage is implemented for two-cycle access. Since the implementation of the data caches is not discussed in this report, the implementation part of the memory access mainly includes the Load-Store (LS) logic. According to MIPS ISA the *load* instruction includes loading a byte, half-word, or a word. LS logic mainly handles the addressing errors when accessing the data memory. For instance, if a word from the memory needs to be loaded and the calculated address is pointing to an odd address, it means that the address calculated is wrong.

Besides LS logic, the memory stage also includes two two-input multiplexers, one in MEM1 and the other in MEM2 stage. Both the multipliers are controlled by the control

signals from the ID stage. The multiplexer in the MEM1 stage selects either the result from ALU or the multiplier. The multiplexer in the MEM2 stage selects either the result from the previous stage or the data loaded from the data memory. The output from both multiplexers are fed back as forwarding paths to the ID stage.

4.5 Write Back

All register writes happen in the write back stage of the pipeline. The register write signal is generated by the control unit in the ID stage and propagated to the write back stage through the pipeline registers. In case of register data dependencies, the register is first written and then the result is the register is accessed to be used as operands. Writing to registers is synchronous to clock, that is the result is written to the register at the rising edge of the clock.

5

Pipeline Verification

Verification is an important part of a design process. Verification is carried out to ensure that required functionality is implemented and bug free. This chapter describes the verification of a 7-stage processor pipeline. The verification process usually involves verifying the functionality of a design against its specification. In this case the specification is a set of predetermined data, for a respective benchmark, written to the memory during the MEM2-stage of the pipeline.

This chapter is divided into three sections. Section 5.1 describes the process and steps involved in the verification of the pipeline. Section 5.2 describes the testbench setup used for verifying the implementation of the pipeline. Section 5.3 concludes the result of the verification process.

5.1 Verification Process

The verification of the pipeline is divided into two steps. In the first step the branch predictor design is verified by integrating it into the 5-stage pipeline, because the branch predictor was implemented prior to the implementation of the 7-stage pipeline - target architecture was not available for verification. In the second step the branch predictor is integrated into the 7-stage pipeline and the verification of the pipeline is carried out.

Cadence Incisive (ncsim) simulator along with embedded microprocessor benchmarks (EEMBC) [18] are used to verify the functionality of the pipeline. Five EEMBC benchmarks were used for verification purpose. Each benchmark has a predetermined checksum of the data written to the memory during the MEM2-stage of the pipeline and this is used as a reference for verification of the pipeline.

The process involved in verifying the branch predictor and the 7-stage pipeline is similar.

The steps involved in the verification process are as follows.

1. The design is compiled and simulated using the ncsim simulator for a particular benchmark.
2. The data written to the memory in the MEM2-stage of the pipeline is stored into a file.
3. A python script is used to parse the file and calculate the checksum.
4. If the calculated checksum matches the expected checksum, then the verification is successful.

5.2 Testbench Setup

Testbenches are set up in order to verify the functionality of a design. A 5-stage pipeline is used as a testbench to verify the functionality of the branch predictor. To verify the functionality of the 7-stage pipeline a testbench is set up with ideal instruction and data caches. These testbenches are further described in detail in the following subsections.

Branch Predictor

Figure 5.1 shows the branch predictor module integrated into the 5-stage pipeline. Since the branch predictor is designed for two-cycle access, the existing 5-stage pipeline had to modified to accommodate this feature. The instruction fetch (IF) stage is divided into two stages, IF1 and IF2. Since the BTB is designed for two-cycle access, the program counter from the IF1 stage is used to index the BTB. In the IF2 stage the tag comparison in the BTB is carried out and the hit signal is generated. The BDP is indexed using the program counter from the IF2 stage. The instruction from the IF2 stage is fed into the predictors to determine whether it is a branch instruction or not. The result from branch predictor is obtained during the IF2 stage.

All the branch instructions are evaluated in instruction decode stage of the 5-stage pipeline. The feedback from the evaluation is sent to the branch predictor. The branch predictor uses this feedback to determine the correctness of the prediction, both direction and target. In case of valid prediction there will be no disruption in the instruction flow. In case of invalid prediction the program counter is loaded with the new value and the instructions in the IF1 and IF2 stage are cleared. According to MIPS ISA all branch instructions are followed by a branch delay slot instruction, which needs to be executed. Hence, in case of misprediction only the instruction in the IF1 stage is cleared. The branch misprediction penalty is only one clock cycle. Once the integration of the branch predictor is successful then the verification is carried out. The verification process is similar to the one described in the section 5.1.

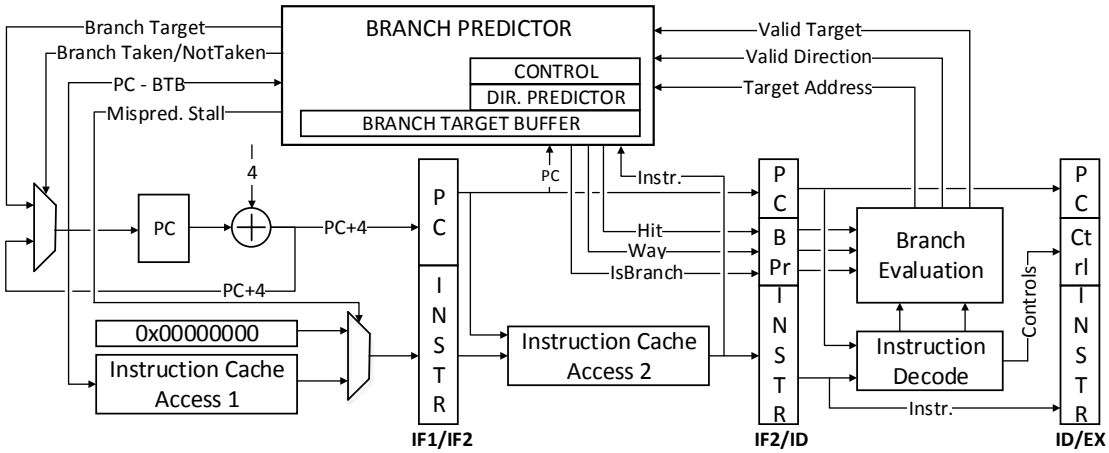

Figure 5.1: Branch predictor module integrated into a 5-stage pipeline, which has been modified to accommodate two-cycle access to the predictor.

7-Stage Pipeline

Figure 5.2 shows the schematic of the testbench setup for the verification of the 7-stage pipeline. The branch predictor module which is successfully verified in the 5-stage pipeline setup is integrated into the 7-stage pipeline. Since the instruction and data caches are not implemented as part of the pipeline, ideal caches are used for verification purpose. The signals such as program counter and data are wired for instruction cache. The signals such as data, address, read and write signals are wired for the data cache.

The verification process is similar to the one described in the section 5.1. At the start of the simulation the PC is updated with address pointing to the selected benchmark. The instructions are read from the benchmark and data is written to the memory, which is stored in a text file. There were situations when it was difficult to identify the problem in the pipeline, because of which the verification failed. In that case the 5-stage pipeline was used as a reference to pinpoint the problem - comparing the register values at each clock cycle.

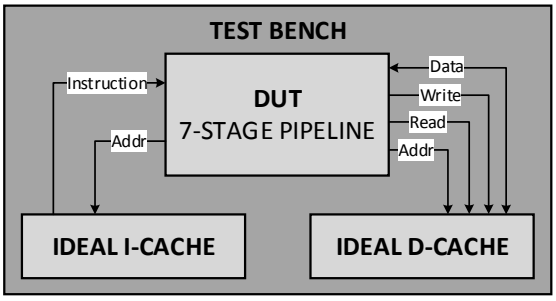

Figure 5.2: Schematic of the test bench used to verify the 7-stage pipeline

5.3 Result

The verification of the 7-stage pipeline design, including the branch predictor, is successful. The pipeline design is verified using the following five EEMBC benchmarks.

- Autocorrelation
- Convolutional Encoder
- Fast Fourier Transform
- Viterbi Decoder
- RGB to CMYK Conversion

Once the verification is successful, the design is synthesized and evaluated for area, power, and timing. The evaluation of the pipeline is discussed in detail in Chapter 6.

6

Pipeline Evaluation

Area, power, and timing are considered as metrics in order to evaluate the 7-stage pipeline. The VHDL implementation of the pipeline is synthesized using Cadence Encounter RTL compiler (Cadence RC). Synopsys Design Compiler (Synopsys DC) is used to evaluate the pipeline in terms of timing. Even though both the tools, Cadence RC and Synopsys DC, are similar in functionality, Synopsys DC provides DesignWare multiplier IP blocks with different pipelined stages that can be used for timing evaluation. The performance impact of the instruction and data cache on the pipeline is not considered in this evaluation.

6.1 Timing

The longest combinatorial path delay between the stages in a pipeline determines the operating clock frequency of a processor. Hence it is important to balance the pipeline stages with respect to path delay. In order to identify the longest combinatorial paths, the pipeline is synthesized for different timing constraints using Synopsys DC with 65 nm Low-Power, Low Vt cell library. Based on the synthesis result two main combinatorial paths were identified which are critical with respect to delay. The identified critical paths are as follows.

Path 1 Path from the output of the ID/EX to the internal pipeline register of the multiplier, assuming that the multiplier is pipelined.

Path 2 Path from the output of the MEM1/EX, through the branch predictor, to the input of the program counter.

Multiplier

Two multiplier designs, Booth-recoded and Synopsys DesignWare block IP, with different pipeline stages are evaluated for the timing constraint. Table 6.1 gives the time slack for

these multiplier designs synthesized with different timing constraints. Both the multiplier designs pipelined for 2-stages meet the 1.5 ns timing constraint with zero time slack. But when the timing constraint is reduced to 1.25 ns, both the 2-stage pipelined multiplier designs fail to meet this timing constraint, with negative time slack as given in Table 6.1. In order to meet 1.25 ns timing constraint, a 3-stage pipelined DesignWare multiplier IP is integrated. Even though the 3-stage design meets the timing constraint, it affects the pipeline performance by introducing an additional clock cycle to the stall penalty due to data dependencies. The timing constraint can still be reduced and met by using multipliers with higher pipelined stage, but it will increase the stall penalty and the gain might not be worth it, as other paths might get critical in the meanwhile.

Table 6.1: Multiplier time slack

Design	**Timing Constraint**	
	1.5 ns	1.25 ns
Booth-recoded (2-stage)	0	-0.16
DesignWare IP (2-stage)	0	-0.13
DesignWare IP (3-stage)	0	0

Branch Predictor

Two branch direction predictor designs, bimodal and two-level predictor, are evaluated for timing. For the evaluation of bimodal predictor a table size of 128 is used. For the evaluation of two-level predictor a table size 128 is used for both the levels, L1 and L2. Table 6.2 gives the time slack for these branch predictor designs synthesized with different timing constraints. The bimodal predictor meets the 1.5 ns timing constraint, but fails to meet the 1.25 ns timing constraint, whereas the two-level predictor fails to meet both the 1.5 ns and 1.25 ns timing constraint. Initially the feedback path from the branch evaluation unit was provided from the EX stage, because of the long path delay the feedback path has been moved to MEM1 stage. Hence it can concluded that the bimodal predictor is better than the two-level predictor in terms of timing.

Table 6.2: Branch predictor time slack

Design	**Timing Constraint**	
	1.5 ns	1.25 ns
Bimodal	0	-0.06
Two-Level	-0.40	-

6.2 Area

Area is one of the important metrics in determining the performance of an integrated circuit. In this section, the 7-stage pipeline design is synthesized using Cadence in order to extract the area occupied by different functionals units in the pipeline. Table 6.3 gives the setting used during the synthesis of the pipeline.

Table 6.3: Synthesis Settings

Compiler	Cadence Encounter RTL Compiler
Clock period	1400 ps
Toggle rate (on primary inputs)	0.1
Cell library	65 nm LP, 1.2 V, Standard Vt Library
SRAM library	65 nm LP, 1.2 V
Optimization Effort	Medium

7-Stage Pipeline

Figure 6.1 shows the plot of the area occupied by the 7-stage pipeline and the functional units in the pipeline. From the plot it can be deduced that the branch predictor design occupies ≈ 45 % of the total area of the pipeline. A bimodal predictor with table size of 128 and a BTB with table size of 32 and 2-way set associativity are used in the synthesis. In a branch predictor with this configuration, BTB accounts for ≈ 86 % of the total area of the branch predictor. The table size of the BTB used for the synthesis is the minimum available SRAM configuration. If the size of the BTB is increased further, the branch predictor will account for most of the silicon area of the pipeline. Hence care should be taken while choosing the branch predictor configuration.

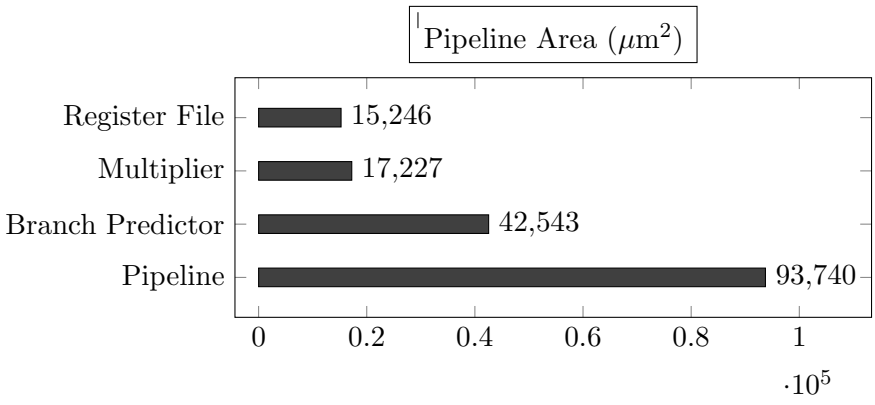

Figure 6.1: Post-synthesis area estimate of the pipeline and its functional units

Branch Predictor

From the plot in Figure 6.1, it is clear that the branch predictor plays a significant role when it comes to silicon area of a pipeline. In order to determine the effect of different branch predictor configurations on the area, several branch predictor configurations are synthesized and evaluated. Figure 6.2 shows the plot of the area occupied by different BDP configurations. It needs to be noted that the size and the configuration of the BTB is kept constant. From the plot it can be deduced that the large two-level (PAp) predictors occupy huge area as compared to bimodal predictors, whereas the smaller two-level predictors such as GAg and Gshare are in close range to bimodal predictors. The prediction accuracy of these two-level predictors is $\approx 3\,\%$ higher than the bimodal predictors.

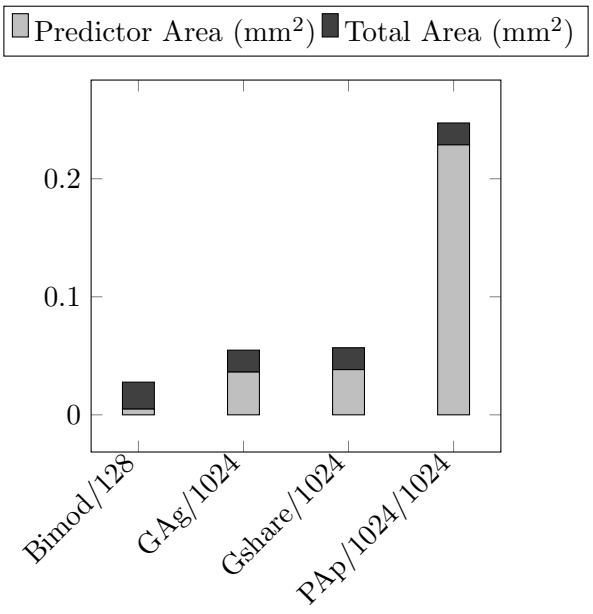

Figure 6.2: Post-synthesis results with respect to area for different direction predictors.

Figure 6.3 shows the plot of the area occupied by different BTB configuration. It needs to be noted that the size and the configuration of the BDP are kept constant. The BTB is implemented using SRAM cells. Based on the simulation results obtained, Chapter 3, it is clear that higher the BTB size and associativity, better the performance of the branch predictor. However, from the plot it is clear that a high cost in terms of area needs to be paid for better performance. Hence a trade-off is necessary between the area and the performance of the BTB.

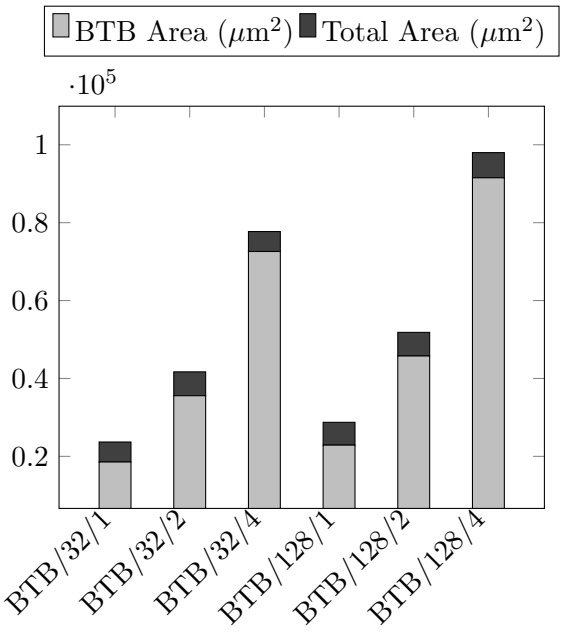

Figure 6.3: Post-synthesis results with respect to area for different BTB configurations

6.3 Power

Similar to evaluation in terms of area, the evaluations are carried out in terms of power as well. The pipeline is synthesized with the settings as given in Table 6.3. The power values are obtained by using static switching probability on all the primary inputs, hence the power values are not accurate.

7-Stage Pipeline

Figure 6.4 shows the power consumption of the pipeline and its functional units. From the plot is can be deduced that the branch predictors account for the maximum power consumption in a pipeline, with a consumption rate of $\approx 65\,\%$ of the total power consumed by the pipeline. From the plots in the Figure 6.1 and Figure 6.4, it is clear that the branch predictor has a huge impact on the pipeline area and power consumption.

Branch Predictor

Figure 6.5 shows the plot of the power consumption of different BDP configurations with respect to the power consumption of the branch predictor. It needs to be noted that the size and the configuration of the BTB is kept constant. Previously it has been shown that the larger two-level predictors such as PAp occupy more area. From the plot in Figure 6.5 it can be observed that they also consume more power when compared to smaller two-level (GAg, Gshare) and bimodal predictors. The prediction accuracy

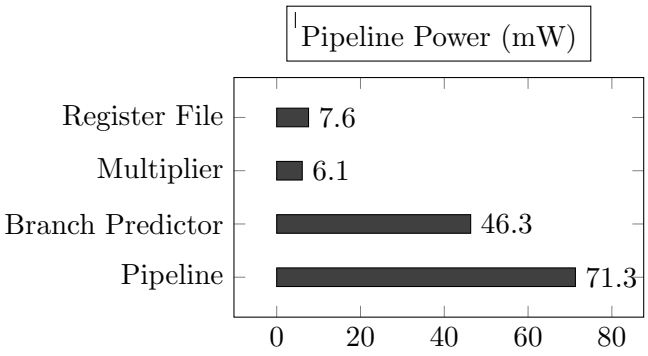

Figure 6.4: Post-synthesis power estimate of the pipeline and its functional units

of PAp predictor is $\approx 1\,\%$ higher than a GAg/Gshare predictor and $4\,\%$ higher than a bimodal predictor. Hence it can be deduced that the performance gain obtained from the bigger two-level predictors is not worth the cost that needs to be paid in terms of area and power.

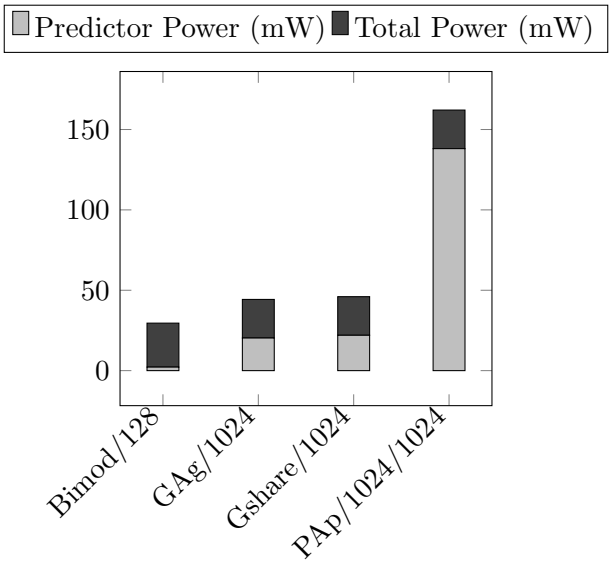

Figure 6.5: Post-synthesis results with respect to power for different direction predictors.

Figure 6.6 shows the plot of the power consumption of different BTB configurations with respect to the power consumption of the branch predictor (BTB and BDP). The size and the configuration of the BDP is kept constant in this case. From the plot it is clear that the BTB accounts for more than $90\,\%$ of the total power consumed by the branch predictor. In this case the table size of BDP is kept small. In case of bigger BDP table size the power consumed by the BTB still dominates. It is been shown from the branch predictor simulations that, besides BDP, a good BTB configuration can have

huge impact on the performance of a processor. Hence it is important to make a good trade-off between the size of a BDP and BTB. It needs to be noted that the difference in area and power between the BTB configurations with 32 and 128 table size is not proportional to the increase in the table size, because for the small SRAM blocks the peripheral circuitry is dominant. When the size of the SRAM is increased from 32 to 128, only the number of cells will increase. The area and power of the BTB scales linearly with respect to the table size only when the table size has increased more than $\approx$ 4k (words).

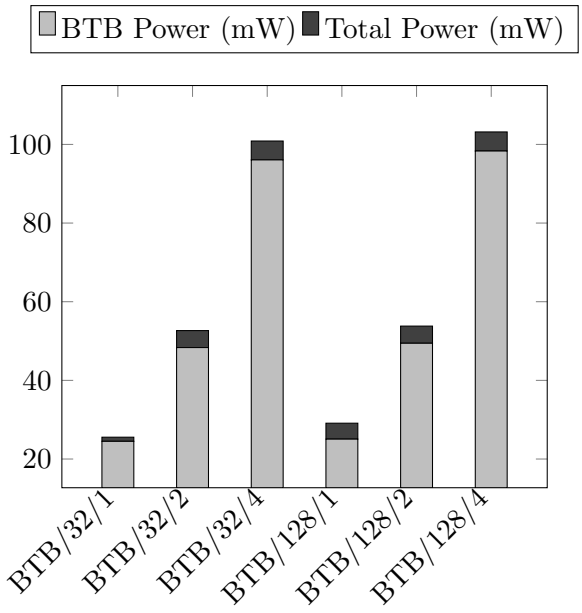

Figure 6.6: Post-synthesis results with respect to power for different BTB configurations.

7

Conclusion

Implementation and evaluation of a 7-stage processor pipeline has been carried out. Implementation of instruction and data cache are not discussed. More emphasis is given to the branch predictors than any other functional units, because they play a vital role in determining the performance of a pipeline. The pipeline is verified using EEMBC benchmarks and well balanced with respect to timing. Ideal caches were used during the verification of the pipeline. Implementation of multipliers are also not discussed. Existing multiplier designs are integrated into the pipeline and evaluated.

Several branch predictor configurations, of BDP and BTB, are simulated in order to identify a few good configurations in terms of execution time and prediction accuracy. Based on the simulation results, it can deduced that the larger two-level predictor such as PAp perform better than a bimodal or a smaller two-level predictors such as a GAg or a Gshare predictor. But, when the designs are synthesized the larger PAp predictors consume more area and power than the bimodal or smaller two-level predictors. The performance gain in using a large PAp predictor is not worth the cost to be paid for the area and power it consumes. Hence a smaller two-level or a bimodal predictor is preferable in terms of area and power. Furthermore, when it comes to timing constraint, two-level predictors fails to meet the required timing constraint. Two-level predictors have longer path delay than the bimodal predictors. Hence a bimodal predictor is preferable over the two-level predictors in terms of timing, area, and power.

Since the results from both the BDP and the BTB are used in order to predict a branch instruction, they are equally important in determining the performance of a branch predictor. Based on the simulation results, it can be deduced that the bigger table size and higher associativity of a BTB in turn leads to better performance of a branch predictor. But, the price to be paid in terms of area and power for better BTB performance is also higher. Based on the BTB area and power plots, it is clear that the increase in asso-

ciativity will have a huge impact on area on power than the increase in number of sets (table size). Hence a trade-off is required between the BTB table size and associativity. A good branch predictor performance can be achieved with a BTB table size of 128 and a 2-way associativity.

The multiplier and the branch predictor are the two main critical paths which affect the timing of the pipeline. The pipeline meets the timing constraint of 1.5 ns with the DesignWare 2-stage multiplier and a bimodal predictor. The pipeline can be pushed to meet the 1.25 ns timing constraint by integrating a DesignWare 3-stage multiplier, but the current branch predictor design will fail to meet the timing constraint. By further pipelining the feedback path of the branch predictor, 1.25 ns timing constraint can be achieved.

The impact of integrating an instruction and a data cache into the pipeline is not considered in this evaluation. Introduction of caches will have a huge impact on the timing, area, and power of the pipeline. Some of the conclusions, based on the results of the pipeline evaluation, made above might not be valid when the caches are integrated and evaluated as part of the pipeline. For instance, the area occupied by the caches will be so huge that either using a bimodal or a two-level predictor will have negligible impact on the total area of the pipeline. Hence these evaluations need to be reconsidered after integrating the caches into the pipeline.

7.1 Future Work

An instruction and a data cache with two-cycle access will be implemented and integrated in the 7-stage pipeline. The caches implemented will be way-predicting set-associative, because the way-predicting caches reduce the energy consumption by about 70 % as compared to conventional caches [19]. The pipeline will be verified and re-balanced with respect to timing. The verified pipeline will then be synthesized and re-evaluated with respect to area and power. The performance of the branch predictors can further be improved by optimizing the content of the BTB, that is to store the branch target instruction instead of branch target address. This technique will help in reducing fetch cycles in case of valid prediction.

Bibliography

[1] Power Consumption of Top 10 Super Computers.
URL http://www.top500.org/lists/2011/06/press-release/#.U5ALYRbA5BU

[2] M. Dubois, M. Annavaram, P. Stenström, Parallel Computer Organization and Design, Cambridge University Press, 2012.

[3] D. A. Patterson, J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, Fourth Edition, 4th Edition, Morgan Kaufmann Publishers Inc., 2008.

[4] S. Mirapuri, M. Woodacre, N. Vasseghi, The MIPS R4000 Processor, in: IEEE Micro, Volume 12, Issue 2, 1992, pp. 10–22.

[5] OpenRISC - Free and open RISC instruction set architecture.
URL http://opencores.org/or1k/Main_Page

[6] C. Perleberg, A. Smith, Branch target buffer design and optimization, IEEE Transactions on Computers 42 (4) (1993) 396–412.

[7] T.-Y. Yeh, Y. Patt, A comparison of dynamic branch predictors that use two levels of branch history, in: Proceedings of the 20th Annual International Symposium on Computer Architecture, 1993, pp. 257–266.

[8] T.-Y. Yeh, Y. Patt, Alternative implementations of two-level adaptive branch prediction, in: Proceedings of the 19th Annual Int'l Symposium on Computer Architecture, 1992, pp. 124–134.

[9] S. McFarling, Combining branch predictors, Western Research Laboratory Technical Note TN-36 (1993).

[10] D. Jimenez, Reconsidering complex branch predictors, in: Proceedings of the 9th International Symposium on High-Performance Computer Architecture, 2003, pp. 43–52.

[11] M. T. Inc., MIPS32 architecture for programmers volume i: Introduction to the MIPS32 architecture, `http://www.imgtec.com/mips/architectures/mips32.asp`, accessed: 2010-09-30.

[12] T. Austin, E. Larson, D. Ernst, SimpleScalar: An infrastructure for computer system modeling, Computer 35 (2) (2002) 59–67.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, MiBench: A free, commercially representative embedded benchmark suite, in: Proc. Int. Workshop on Workload Characterization, 2001, pp. 3–14.

[14] J. L. Henning, SPEC CPU2000: measuring CPU performance in the new millennium, Computer 33 (7) (2000) 28–35.

[15] D. Brooks, V. Tiwari, M. Martonosi, Wattch: a framework for architectural-level power analysis and optimizations, in: Proceedings of the 27th annual international symposium on Computer architecture, 2000, pp. 83–94.

[16] S. Inc., Designware building block IP dcumentation overview, `http://www.synopsys.com/dw/dwlibdocs.php`, accessed: 2010-09-30.

[17] M. T. Inc., MIPS32 architecture for programmers volume ii: The MIPS32 instruction set, `http://www.imgtec.com/mips/architectures/mips32.asp`, accessed: 2010-09-30.

[18] EEMBC, Embedded microprocessor benchmarks: System benchmarks, `http://www.eembc.org/benchmark/products.php`, accessed: 2010-09-30.

[19] K. Inoue, T. Ishihara, K. Murakami, Way-predicting set-associative cache for high performance and low energy consumption, in: Proceedings, 1999 International Symposium on Low Power Electronics and Design, 1999, pp. 273–275.