Spring 2017

Survey of Branch Prediction, Pipelining, Memory Systems as Related to Computer Architecture

Kristina Landen
Embry-Riddle Aeronautical University

Survey of Branch Prediction, Pipelining, Memory Systems as Related to Computer Architecture

Honors Directed Study Research Project

Kristina Landen
EE/CE/SE Department
College of Engineering
landenk@my.erau.edu

Dr. Brian Davis
EE/CE/SE Department
College of Engineering
davisb22@erau.edu

Abstract

This paper is a survey of topics introduced in Computer Engineering Course CEC470: Computer Architecture (CEC470). The topics covered in this paper provide much more depth than what was provided in CEC470, in addition to exploring new concepts not touched on in the course. Topics presented include branch prediction, pipelining, registers, memory, and the operating system, as well as some general design considerations for computer architecture as a whole.

The design considerations explored include a discussion on different types of instruction types specific to the ARM Instruction Set Architecture, known as ARM and Thumb, as well as an exploration of the differences between heterogeneous and homogeneous multi-processors.

Further sections explain the interoperability of various portions of the computer architecture with a focus on performance optimizations. Branch prediction is introduced, and the quality improvement which branch prediction provides is detailed. An explanation of pipelining is given followed by how pipelining on different types of processors may be difficult. Registers, one of the fundamental parts of a computer, are explained in detail, as well as their importance to computer systems as a whole.

The memory and operating systems sections tie this paper together by delving deeper into the architecture of computers, then resurfacing with how the software and hardware interact through the operating system.

This paper concludes by tying each section discussed together and presenting the importance of computer architecture.

1 Introduction

The purpose of this paper is to present the research conducted in parallel with the Computer Engineering course CEC470: Computer Architecture (CEC470) in order to fulfill the requirements of an Honors Directed Study. This Honors Directed Study acts as a survey of the topics introduced in CEC470, and then goes beyond the content which was presented in

the lecture of this course. If more information is desired on any of the topics covered in this survey, see Section 9: References or Computer Architecture: A Quantitative Approach 5th Ed. by David Patterson and John L. Hennessy[1], the textbook for CEC470.

This paper will present the information from this survey in the most effective way possible. Topical coverage starts at a high level to provide general information regarding processor design, then moves into the specifics of processor implementation, and finally moves away from the physical aspects of processors to explore the interaction of software with processors.

2 Design Considerations

When any new system is being designed, there are a number of engineering decisions that must be made. However, before these decisions can be made, possible outcomes to each decision must be evaluated to ensure the decision is as accurate as possible. This section provides some analysis of design considerations that may be made in the design of computer systems.

Section 2.1: ARM vs. Thumb provides analysis on when each instruction type is used. ARM and Thumb are different kinds of instructions used in the ARM Instruction Set Architecture (ISA)[1]. Section 2.2: Heterogeneous vs. Homogeneous discusses the difference between heterogeneous processors and homogenous processors as well as which type may provide a performance increase in given situations.

2.1 ARM vs. Thumb

This section will discuss the performance potential of ARM and Thumb instructions, and determine whether or not there is ever a situation where both ARM and Thumb instructions may be used in conjunction. First, background information regarding ARM and Thumb will be provided in order to effectively convey the differences between ARM and Thumb. Next, the differences between ARM and Thumb which result in a change in performance between the two will be highlighted. Finally, this section will discuss whether or not the use of both ARM and Thumb instructions is desired in any situation.

ARM and Thumb are both types of instruction that are used in the ARM Instruction Set Architecture (ISA)[1]. The primary difference between the two instruction types is size; ARM instructions are 32 bits versus Thumb instructions are half the size of ARM instructions at 16 bits. In some cases, ARM instructions are composed of Thumb instructions. This will decrease the number of total instructions in a program because the Thumb instructions will be executed as if they were one ARM instruction. However, this translation is not always possible because there is not a direct conversion between all ARM and Thumb instructions.

Cost, performance and power are just a few parameters that designers focus on. Being able to use a combination of ARM and Thumb code within an application enables designers to balance the cost, performance and power characteristics of the overall system. Where performance is the primary concern, generally the fewer instructions required the better, therefore, using ARM instructions alone will usually give the best results[2]. Writing applications using Thumb instructions will enable more of the most frequently used instructions to be stored in on-chip memory. This translates to a higher level of code density and results in a lower level of power use because fewer memory accesses are made.

In order to achieve the desired balance of power, performance and code density to produce an optimized design, designers tend to use a mixture of both ARM and Thumb instructions[2]. One way designers may do this is by identifying performance critical code and using ARM instructions for this portion of the code. Where possible, Thumb instructions are used for the remainder of the code to minimize memory footprint.

2.2 Heterogeneous vs. Homogeneous

This section will discuss the differences between heterogeneous multiprocessors and homogeneous multiprocessors, then provide analysis on the amount of generalized performance improvement is required for a heterogeneous multiprocessor system to be used over a homogeneous multiprocessor system. First, an explanation of the differences between a heterogeneous multiprocessor and a homogeneous multiprocessor will be presented. Through this explanation, facets of each design will show how the performance of each system differs. Finally, this section will discuss how much performance improvement a heterogeneous multiprocessor system must provide the system overall in order justify the additional cost of implementation.

A homogeneous multiprocessor is a processor which has multiple processor cores on the same chip where each core is of the same type[1]. A heterogeneous multiprocessor is a processor which has multiple cores on the same chip where at least one of the processors on the chip is of a different type than the rest[1]. An application which may use a homogeneous multiprocessor is one which does not have much variety in its computations, such as a desktop computer. In contrast, heterogeneous multiprocessors may be used in applications where many different kinds of computations are being performed, or where the different computations each have a specialized processor for that task which is optimized for performance, power consumption, or both.

When looking to decide whether to use a heterogeneous or homogeneous multiprocessor system, much analysis for the specific application is required in order to make this decision. Heterogeneous multiprocessor systems require vastly more simulation than homogeneous multiprocessor systems because of the varied

levels of heat dissipation, heat generation, and power consumption[3]. Further considerations include the physical layout of the processors on the board. In a homogeneous multiprocessor system, each processor is the same, so placement is not as large of an issue. However, in a heterogeneous multiprocessor system, the different processors may need access to different memory banks and require different configurations than what may have been designed for a homogeneous multiprocessor system.

To determine the performance increase, analysis should be performed to determine if the projected profits from the performance boost will cover the extra cost of implementing a heterogeneous system. The issue of support for the product once released arises in these considerations as well since heterogeneous systems are more difficult to work with.

3 Branch Prediction

Regardless of whether a heterogeneous or homogeneous processor is used, all modern, high performance processors use branch prediction to improve performance[1]. Branch prediction is the process of predicting the outcome of a branch. A branch is a type of control logic which allows code to move from one block to another. The decoding and interpretation of a branch instruction by the processor takes a long time. Branch prediction allows the processor to predict what the outcome of the branch will be based on some prediction strategy which is set by the ISA. In the case which the prediction is correct, computation time is utilized effectively with the potential for minimal wasted computation cycles. In the case where the prediction is incorrect, performance is not improved by the branch prediction. However, depending on the prediction strategy in place, the branch predictor may learn from this misprediction in order to be more accurate in the future.

Section 3.1: Overall Quality Improvement provides further information regarding branch prediction and its effectiveness over code which does not implement branch prediction. Section 3.2: Algorithms Used discusses a number of different branch prediction strategies and whether or not they are used in practice. Section 3.3: Branch Prediction on Heterogeneous Multi-Core Systems ties in some operating systems concepts and explores the effect of a context switch on branch prediction when using a heterogeneous multi-core processor.

3.1 Overall Quality Improvement

This section will provide a definition of branch prediction and detail the performance improvement which it provides. First, branch prediction will be explained. Following this definition, some analysis of why branch prediction provides a performance increase over code which does not use branch prediction will be explored.

The purpose of branch prediction is to reduce the total number of stalls that are caused by branch statements. When a branch is encountered in code, it must be executed before the code knows what instructions to run next as there are multiple options. Branch prediction tells the code to assume the outcome of the branch is taken or not taken and to continue executing instructions based on that assumption. When the branch finishes executing, the final state of the branch is communicated to the rest of the code. If the prediction was correct, the code continues executing as it was. However, if the prediction was incorrect, the code must squash the instructions that were executed from the wrong branch target, then fetch all new instructions from the correct target. Without branch prediction, the code would not be able to fetch instructions while the branch finishes executing, wasting these computation cycles.

Branch prediction is the process of making an educated guess as to whether a branch will be taken or not taken based on a preset algorithm[1]. A branch is a category of instruction which causes the code to move to another block to continue execution. Branch prediction has the ability to be static or dynamic[4]. Static branch prediction means that a given branch will always be predicted as taken or not taken without possibility of change throughout the duration of the program. Dynamic branch prediction means that the predicted outcome of a branch is dependent on an algorithm, and the prediction may change throughout the course of the program. Code is able to use a combination of both static and dynamic branch predictors based on the type of branch.

The improvement branch prediction provides is dependent on the number of branches in the code, as well as the type of prediction being used as different prediction methods have varied rates of success. Overall, branch prediction provides an increase in performance for code containing branches. This improvement is based on the number of computational cycles which are able to be used for computation rather than wasted on a system which does not use branch prediction.

3.2 Algorithms Used

This section will explore various techniques for branch prediction, to identify which strategies are quality, and finally to state which are used in practice. First, each of the different branch prediction strategies that were identified in the research for this document will be presented. Of note, due to the limited nature of this paper, the algorithms presented here are by no means a complete listing of all branch prediction algorithms. Next, of the strategies listed here, the quality of each will be assessed with both positive and negative aspects of each discussed. Finally, whether or not a given algorithm is used in practice or not will be disclosed.

In order to fully explain the different branch prediction strategies, some more background information must be given. There are three different kinds of

branches: forward conditional, backward conditional, and unconditional branches[5]. Forward conditional branches are when a branch evaluates to a target that is somewhere forward in the instruction stream. Backward conditional branches are when a branch evaluates to a target that is somewhere backwards in the instruction stream. Common instances of backward conditional branches are loops. Unconditional branches are branches which will always occur.

A static or dynamic prediction strategy will determine which different algorithms or methods are available for use. For static branch prediction, the strategy may either be predict taken, predict not taken, or some combination that specifies the branch type such as backward branch predict taken, forward branch predict not taken[5]. The third strategy is advantageous for programs with loops because it will have a higher percentage of correctly predicted branches for backward branches.

Dynamic branch prediction is able to use one-level prediction, two-level adaptive prediction, or a tournament predictor. One-level prediction uses a counter based on a specific branch to use said branch's history to predict its future outcomes[5]. The address of the branch is used as an index into a table where these counters are stored. When a branch is correctly predicted taken, a counter is incremented. When a branch is correctly predicted not taken, the same counter is decremented. In the case where the prediction was incorrect, the opposite occurs. For instance, if a branch was incorrectly predicted taken, the counter would be decremented, or if a branch was incorrectly predicted not taken, the counter would be incremented. The status of this counter is used to make the prediction for the branch's next iteration. If the counter holds a value of zero or one the prediction is not taken, and if the value of the counter is two or three the prediction is taken.

The two-level adaptive branch prediction is very similar to the one-level branch prediction strategy. The two-level strategy uses the same counter concept as the one-level, except the two-level implements this counter while taking input from other branches. This strategy may also be used to predict the direction of the branch based on the direction and outcomes of other branches in the program. This strategy is also called a global history counter[5].

Hybrid or tournament prediction strategies use a combination of two or more other prediction strategies[5]. For example, any static prediction used in conjunction with a dynamic prediction strategy would be considered a hybrid strategy.

All of the strategies listed here are used in practice. The two-bit counter presented in the one-level branch prediction strategy is used in a number of other branch prediction strategies, including a predictor for choosing which predictor to use. One disadvantage to each of these strategies is that their level of improvement for a given code will vary depending on what is written into the code.

3.3 Branch Prediction on Heterogeneous Multi-Core Systems

This section will explain how branch prediction methods function on a heterogeneous multi-core system when a context switch occurs. First, a brief definition of context switch will be provided along with its relevance in this discussion, and why this topic was addressed in the first place. Following this brief background will be the discussion of how the branch prediction algorithms and the associated hardware are affected by context switches.

For the purposes of this section, a context switch occurs when multiple threads or processes are running on a system and they must share computation time. The context switch is the process of saving one thread or process' context so it may resume execution when the computer is returned to it. A context consists of the code and registers and anything else the process or thread requires for execution.

The relevance of a context switch in this discussion is that the assumption is made that the code in question is running on a heterogeneous multicore system. This means that a process may be assigned to either of the core types. If another process is admitted, the current process running on a given core may be forced to save its context and stop running. This section discusses what happens to the branch predictors associated with a given process upon the occurrence of a context switch.

When analyzing a big.LITTLE system, a heterogeneous multi-core system with a common ISA of ARM, the system utilizes global task scheduling[6]. This scheduling mechanism allows the operating system to be able to accurately assess which core type and specifically which core a new process or thread should be placed on based on expected performance. This also means that the operating system may specifically target either big or LITTLE cores on the system, and potentially move a thread or process from a big core to a LITTLE core, or vise versa. However, the context of a thread or process running on a big.LITTLE system is able to be transferred between the two different kinds of processor[6]. This means that all memory within the thread or process is saved, including the branch prediction data.

4 Pipelining

Any modern processor will be using pipelining in order to optimize performance. Without pipelining in a processor, each instruction must wait until the previous instruction has completed before the next may begin in a true sequential manner. On a pipelined system, each clock cycle an instruction may begin regardless of whether or not the previous instruction has completed or not[1].

Section 4.1: Hazards provides a number of problems that are introduced when pipelining is implemented in a processor. Section 4.2: Schedule to Avoid Hazards and Dependencies discusses how a pipeline may be scheduled to avoid the hazards discussed in Section 4.1: Hazards. Section 4.3: Stages Related to Performance provides an analysis of how many

stages a pipeline may have and how the number of stages is directly related to the performance of the pipeline. Section 4.4: Dynamic Frequency Scaling and Performance explains the effect of dynamic frequency scaling on pipelining and the resulting effect on performance overall. Section 4.5: Pipelining on a Heterogeneous System discusses how pipelining functions on a heterogeneous multi-core system. Section 4.6: Pipelining on a Common ISA Heterogeneous System continues the analysis begun in Section 4.5: Pipelining on a Heterogeneous System but focuses on the situation of a common ISA.

4.1 Hazards

This section will provide an explanation as to what hazards are and why they matter. This section will also further introduce pipelining and how hazards affect pipelining. First, definitions of hazards, including the different types, will be provided. Next, an explanation of pipelining will be provided. Finally, this section will tie hazards and pipelining together to explain the impact that hazards have on pipelining.

Pipelining is the process of executing more than one instruction in a given computational cycle. Consider a single instruction. For the purposes of academia, there are five main stages to completely execute an instruction: instruction fetch, decode, execute, memory, and write back[7]. At any given time in the overall execution the instruction will only be in one of these five stages at a time. Pipelining takes advantage of this and begins executing other instructions once the initial instruction finishes a given stage. In order to ensure that data from different instructions do not become intermingled, based on the length of the longest stage, each instruction is not allowed to move to the next stage of execution until a set amount of time has passed. This set amount of time is the inverse of the clock frequency.

By changing the way that instructions are executed, a number of issues are introduced. These issues are called hazards, and are the purpose of this section. There are three types of hazards: structural, data, and control[7]. Structural hazards occur when an instruction requires some functional unit in order to complete its execution but a functional unit of that type is unavailable due to pipelining. Data hazards occur when the output for one instruction is an input for a subsequent instruction, and the data is not available when the second instruction goes to execute because the first instruction has not yet produced the data. Control hazards occur when branches enter the pipeline and change the order of instructions to be executed.

Each of these hazards have the potential to cause serious delays in the pipeline. In most situations, when one or more of these hazards occur, a bubble, or stall, must be inserted into the pipeline in order to preserve instruction order. Because of this, the benefits of the addition of pipelining must be contrasted with their potential

cost. Section 4.2: Schedule to Avoid Hazards and Dependencies discusses how the costs associated with pipelining may be minimized.

4.2 Schedule to Avoid Hazards and Dependencies

This section will further discuss pipelining and hazards with an emphasis on how the pipeline may be scheduled to avoid data dependencies by both the compiler, the hardware, or some combination thereof. First this section will explain data dependencies. Next, this section will detail how the pipeline is capable of being scheduled from perspectives of both the hardware and the compiler. Finally, an explanation of how the pipeline may be scheduled by the hardware, compiler, or some combination in order to avoid data dependencies and hazards will be provided.

Name dependencies occur when multiple instructions refer to the same variable. These dependencies do not always cause issues, but they can. If two sequential instructions exist such that the destination of the first instruction is an operand for the second instruction, this is a data dependency which will cause a delay in the pipeline.

In order to reduce the amount of stall time caused by data dependencies, both the compiler and the hardware are able to assist in the scheduling of instructions[1]. The compiler is able to view all of the instructions in a given program and insert appropriate "no-op" instructions into the instruction stream to sufficiently spread out dependent instructions. A "no-op" instruction is an instruction where nothing happens. However, in order to do this successfully, the compiler must have sufficient knowledge about the hardware which it is running on to know the amount of time in cycles that an instruction will take to produce a value. From the hardware side of things, additional hardware called bypass paths are able to be added to try and avoid no-ops. These bypass paths move data from the end of the execute stage to the beginning of the execute phase so that subsequent instructions may have access to the needed data as soon as possible rather than having to fetch the recently produced data from memory or architected registers. In the case that the data is still not available, the hardware is able to insert stalls into the instruction stream similar to how the compiler may insert no-ops.

It is easier for the compiler to perform the scheduling because the compiler has access to all of the code in the overall program as well as essentially infinite time in terms of computation. Conversely, the hardware only has access to the set of instructions that are inflight. Because of this, the compiler is better at scheduling instructions. A caveat to this is that the compiler must have information about the hardware which it is running on in order to effectively schedule the instructions.

The compiler could schedule instructions any way that it sees fit based on the data from the ISA, however this may not be the most effective. The compiler is able to

analyze the content of each instruction and create a directed graph of all of the instructions with the instructions as the nodes, and connect the nodes based on the dependencies between each instruction[8]. The ideal usage of this directed graph is to select the path which has the least amount of stall cycles. However, the issue with this is the analysis and final selection of the path with the least stall cycles is a problem which is NP-Complete. The combination of static methods with the dynamic path traversal is one way to reduce the amount of time required to find a potentially optimal path[8].

Even though the compiler will perform its own optimizations, these optimizations, as previously stated may not be entirely optimal, so the hardware may add its own optimizations in order to further better the execution of the instructions. It is beneficial for both the hardware and compiler to provide optimizations for the scheduling of the instructions because of the limitations of the compiler and the hardware on their own.

4.3 Stages related to performance

This section explains how the number of stages within a pipeline impact the performance of the pipeline. This section will also explain how the type of instructions being executed on a pipeline will also impact the overall performance of the pipeline.

As stated in Section 4.1: Hazards, academia teaches a five-stage pipeline. This five-stage pipeline can be expanded or compressed to have more or less pipeline stages. Both of these implementations have advantages and disadvantages related to a variety of performance metrics. By increasing the number of stages, the pipeline is forcing the instructions to take more time overall to complete their execution. This allows the system to execute larger instructions with increased efficiency. However, smaller instructions may finish their execution early and waste computation cycles. The amount of performance improvement will also be variable based on the types of instructions and programs that are being run on the system because different instructions will have different lengths, and different programs will have different quantities of different kinds of instructions.

4.4 Dynamic frequency scaling and performance

This section will further discuss the performance of pipelining with relation to dynamic frequency scaling. First, this section will provide an explanation as to what dynamic frequency scaling is. Once dynamic frequency scaling is explained, this section will relate dynamic frequency scaling to the performance of a pipeline.

Dynamic frequency scaling is the process which a processor goes through to change its operating frequency in order to increase performance, or reduce power consumption. In general, this adaptation of the system would just cause rate at

which instructions move through the pipeline to vary. However, if the system supports a variable-length pipeline[9], then other changes may take place.

A variable-length pipeline is a pipeline which is capable of changing the number of stages it contains based on the operating frequency[9]. Variable-length pipelines are atypical in industry, and mentioned here for theoretical completeness. In Koppanalil's article[9], it is stated that in the two operating modes, deep and shallow mode, the number of pipeline cycles in deep mode is double that of the cycles in shallow mode. The paper states that deep mode is to be executed when the processor is operating at high frequencies, and shallow mode when the processor is operating at low frequencies. The transition between deep and shallow mode is done by enabling and disabling the circuitry required for the separation of the pipeline stages as needed for each specific mode.

The combination of dynamic frequency scaling with variable-length pipeline stages allows for performance increase in processors which support both. This performance increase is based in the amount of power saved as well as the speed of processing when in deep and shallow modes respectively[9]. On systems which do not support variable-length pipeline stages, dynamic frequency scaling increases performance through reduced power consumption and increased instruction throughput when the frequency is reduced and increased respectively.

4.5 Pipelining on a heterogeneous system

The purpose of this section is to explain how pipelining works on a heterogeneous multi-core system, specifically when a process is moved from one core to another, either of a different core type or of the same core type. First, potential differences between different core types which are relevant to the pipelining process will be discussed. This section will then explain what happens to the contents of the pipeline when a process is swapped from one core to another.

Generally, when a heterogeneous multi-core processor is implemented, the design goal is to improve performance related to the application of the system. This performance improvement focus could be heat conservation, power consumption, or overall instruction throughput. In order to do this, the different processors selected to be included in the processor are vastly different, but the overall purpose of each will be application dependent. Because of this, the number of pipeline stages will likely be different across the different core types, but does not mean they must be. The same logic applies to the order of the pipeline stages: they may be in the same order, but may not be.

Since the primary goal of a heterogeneous multi-core processor is to improve performance, if moving a process from one core type to another core type provides a performance increase, that is what the processor will do. When this occurs, because the pipeline is not part of the context of the process, the instructions

inflight will either finish and commit or be flushed from the pipeline in order to allow the rest of the context to save and the process to be moved.

4.6 Pipelining on a common ISA heterogeneous system

This section will explore the differences in the pipelines of the different cores involved in a common ISA heterogeneous multiprocessor, such as big.LITTLE, and how these differences impact the performance of the system overall. This section will first explain the possible differences between processors on a big.LITTLE system and why these differences matter. This section will then address these differences and provide an explanation as to why the differences provide an overall improvement with regards to the pipeline of the system.

On a big.LITTLE system, the differences between the two core types include number of pipeline stages, instruction types, order of instruction, and cache interfaces[10]. Each of these differences contribute to improved performance for each of the two core types. The big core is meant to be a high-performance core with a larger number of pipeline stages to handle more power intensive operations. The LITTLE core is meant to be a power saving core with a lower number of pipeline stages to handle smaller operations that would potentially be a waste of computation cycles on the big core.

By incorporating multiple big and multiple LITTLE cores on a single chip performance is improved. At any given time, the chip may shut off any core which is not being used in order to save power[10]. The chip is also able to choose which core to give any given process, potentially running multiple processes on a single core because that alternative was seen as more efficient than powering up another core and running the additional processes on it.

Specifically, looking at the number of pipeline stages on the big and LITTLE cores, based on the kind of processes that each core type is meant to run, the varied number of stages provide a performance increase in each core. The big core has a larger number of stages which allow more complicated instructions to be broken into smaller, more manageable pieces of executable code which will be executed with a fewer number of delay slots than if the same process had been executed on a LITTLE core. The LITTLE core has a smaller number of pipeline stages to allow the less computationally intensive processes to execute quicker than they would on a big core.

5 Registers

Moving even deeper into the architecture of processors, the next topic to be covered are registers. When data is being used by the program, it would be impossible to accomplish anything in a timely manner if each instruction had to go all the way to main memory implemented with DRAM on each reference. To avoid this, registers were created in the

instruction set architecures as a temporary storage for data[1]. This section discusses different types of registers and how each are used.

Section 5.1: Conventional Register File vs. Rename Register File explains the differences between a conventional register file and a rename register file, then goes into detail about how each is used. Section 5.2: Register File Relevance to Bits per Register and Ports describes the impact of the layout of the register file on registers and ports. Section 5.3: Architected Registers vs. Rename Registers vs. Inflight Instructions Across Different Processors explains what architected registers, rename registers, and inflight instructions are, then highlights the differences between each.

5.1 Conventional register file vs. rename register file

This section presents the differences between a conventional register file and a rename register file. This section will provide definitions of both types of register files. Followed by these definitions will be an emphasis on the differences between conventional and rename register files. Finally, uses of both kinds of register files will be explored in addition to how each are constructed.

A conventional register file is a series of registers laid out in a grid. These registers are accessed through bit and word lines which access the columns and rows of the grid of registers respectively[11]. Each register stores either a committed value or an intermediate value of a calculation. A rename register file is similar to a conventional register file, except a rename register file stores a mapping of physical registers to architected registers[12].

Conventional register files are used in every processor which contains registers. The format of the register file plays a key role in how each register is accessed. To access any register, both the corresponding bit and word lines must be activated[11]. This allows registers which have different bit and word lines to be accessed for reading and writing potentially simultaneously. This relationship will be expanded upon in the next section.

Rename register files are only located on systems which implement register renaming to help prevent hazards as a result of execution being performed out of order[12]. Rather than working directly with the architected registers, physical registers are mapped to the architected registers and used in the instruction sequence. When multiple instructions are expected to write to the same architected register, a different physical register is assigned to each instruction for the designated architected register. This process is what allows hazards to be avoided.

5.2 Register file relevance to bits per register and ports

This section will explain the relationship between the layout area of a CPU register file, the number of bits per register contained, and the number of ports in and out of the register file, as introduced in the previous section.

The number of read and write ports on a register file is directly related to the number of bit and word lines, along with the overall area of the register file[11]. For each port, there is one word line running horizontally across the register file. For each port which is designated for writing, there are two bit lines running vertically across the register file. For each port which is designated for reading, there is one bit line running vertically across the register file.

This number of bit and word lines impacts the overall size of the register file due to the size of the wire used for these lines[11]. This wire size will increase, allowing more data to be transmitted, as the number of ports into the register file increases. More specifically, the wire size will always be the square of the number of ports into the register file.

As the number of ports into the register file increases, so does the wire size into the register file. The register file is generally pitch-matched to the size of the datapath associated with the register file[11]. As the size of the register file wire increases, it forces the data path wire size to increase. The data path wire travels throughout the circuit, and if its size continuously increases, design issues regarding heat and size will be caused as the register file grows in size. To keep these issues at a minimum, circuit designers will use multiple register files on a single system.

5.3 Architected registers vs rename vs inflight instructions across different processors

This section will provide the similarities between architectural registers, rename registers, and inflight instructions over different processors. First the differences between architectural registers, rename registers, and inflight instructions will be explained. The similarities between each of these will be provided following their definitions.

Of the three, inflight instructions are the most different concerning physical characteristics. Inflight instructions are the total number of instructions which are currently being executed, which is an indicator of how deep the system pipeline is[1]. Architected registers are the registers which are designated by the instruction set architecture (ISA)[1]. Some architected registers may have specific purposes varying from ISA to ISA. Rename registers are implemented in systems which use register renaming techniques[11]. These registers are physical registers which are mapped to the architected registers in the system.

Inflight instructions carry no real similarities to architected or rename registers, aside from the total number of inflight instructions will vary with the ISA due to the varied depth of the pipeline on different machines. Architected and rename registers are fundamentally the same functional unit, just used differently by the ISA. Their uses will vary dependent on the ISA which is implemented.

Architected registers are the only independent functional unit on this list which will change based off of the ISA rather than a specific processor implementation. Here two ISAs, MIPS and ARM, will be analyzed for their implementations of architected registers. The MIPS ISA contains 32 architected registers[13]. Of these, registers 29, 30, and 31 are the stack pointer, frame pointer, and the return address of a function call respectively. Of note are registers 0 and 1 which are the constant 0 and reserved for the assembler respectively. These registers are unique to the MIPS ISA. The ARM ISA contains 16 architected registers[1]. Similar to the MIPS ISA, register 13 is the stack pointer. Different from the MIPS ISA, registers 14 and 15 are the link register and program counter respectively. The ARM ISA also has a current program status register which contains 32 bits which communicate various conditions of the system including whether Thumb mode is enabled or if an interrupt has occurred[1].

6 Memory

Section 5: Registers explained the use of registers, a simple, but high speed form of memory. Section 6: Memory will go into much more detail on the implementation of memory hierarchy, and how memory can be protected.

Section 6.1: Implementation of Data Structures discusses how both the stack and heap are implemented in memory as well as what kind of information is stored in each, and why it matters what information is stored where. Section 6.2: Protection from Single Event Upsets explores a variety of techniques used to protect memory from errors. Section 6.3: Processing In Memory explains what processing in memory is and why processing in memory is useful. Section 6.4: Reduction of Average Memory Access Time provides a number of methods which are used to reduce the amount of time that it takes to access main memory. Section 6.5: Prefetching explains what prefetching is with regards to hardware and software, then gives examples of each. Section 6.6: Effects of Prefetching provides insight on the improvements which prefetching provides systems.

6.1 Implementation of Data Structures

This section provides an explanation of how different data structures are implemented in memory. First, the concept of a stack and a heap will be explained, and what information is stored in each data structure will be defined based on common practice. Next, the implementation of each of these data structures in memory will be detailed. Finally, an explanation of why it matters what kind of information is stored in each the stack and the heap will be provided.

In order for this section to be understood, background on the definitions of a stack and a heap must be provided. A stack is a data structure that follows "first in, last out" or FILO. This means that the first item that is placed onto the stack, or pushed onto the stack, will be the last item removed, or popped, from the stack. A heap is implemented as a tree in memory with either the highest or lowest priority item stored in the root of the tree. As items are removed from the tree, it is reordered in order to keep the highest or lowest priority item at the root of the tree[14].

Both the stack and the heap are stored in the random access memory (RAM) of the computer[15]. Within the code, variables which are written in to each block of code are stored on the stack because the stack has a limited amount of storage space which is determined at compile time. Space on the stack is allocated upon code block entry, and deallocated on code block exit. If there are dynamically allocated variables within the code, these variables are stored in the heap. These variables are only deallocated when the program calls a memory deallocation function.

Within memory, the stack and heap are stored in different ways, providing each with a different access speed. The stack is stored in sequential memory allowing for faster access times[15]. The heap is stored randomly throughout the available RAM in the system[15]. Because of this, the access time for any variable stored in the heap is lower since it takes longer to locate than it would if the variable were on the stack.

The location of the stack and the heap in memory is important due to its impact on memory access speeds. Memory access speeds impact the entire system due to their inherently high latencies. Anything that can be done to decrease the latency of a memory access should be done in order to increase the efficiency of the system overall.

6.2 Protection from Single Event Upsets

This section will explain how memory is able to be protected from errors such as single event upsets (SEUs). First, an overview of the various sources of error in memory will be provided. Next, the various techniques used to detect, and in some cases correct, data errors will be discussed. Of note, the methods discussed in this section are not a complete listing of all possible methods to correct and detect errors in code. This is just a subset of possible methods which were covered in the research conducted for this paper. Finally, a discussion of where each of these methods is used will be provided.

Errors in memory can be caused by a number of things, but are most commonly caused by forms of radiation or SEUs. SEUs occur when a single energetic particle passes through a chip, altering a small number of bits while leaving no permanent damage to the chip itself[16].

Common techniques used to detect, and in some cases correct, SEUs include parity, cyclic redundancy check (CRC), hamming code, Reed-Solomon error correction, and chipkill. Parity detects an odd number of errors in the code by summing all the bits in the code in question, then appends an additional bit based on whether the total number of one's counted was odd or even[1]. If an even number of bits are altered, the count of one's will remain the same and the error will go undetected. CRC acts as a non-secure hash function by appending a check value based on polynomial division[17]. The calculation is repeated after the data is transmitted and if the check values do not match, corrective action is taken. Hamming code is an expanded version of parity which breaks large pieces of data into chunks and calculates the parity of those chunks[18]. A parity is also taken of the parity bits. This allows hamming codes to detect multiple errors, and correct a single error. The Reed-Solomon error correction code is one of the more complicated error checking codes. By adding some number of check symbols to a set of data, a Reed-Solomon code may detect any number up to the total number of check symbols added worth of errors in the code[17]. Reed-Solomon is also able to correct up to half of the total number of check symbols worth of bit errors. This method is also able to detect erasures, or some combination of errors and erasures. Finally, chipkill is able to protect the integrated circuit as a whole rather than the specific data stored on it by using a hamming code and spreading the data across multiple chips[1].

In general, most of these methods are used before data is stored, when data is transmitted, or when data is received. CRC is generally good at detecting noise in transmitted data[17]. Parity works best when it is used in conjunction with other techniques, such as its use in hamming codes. Chipkill is effective on its own since it works on the integrated circuit level rather than the data level.

6.3 Processing In Memory

This section will define processing-in-memory (PIM) and to discuss some of the many advantages and disadvantages of PIM.

Processing in Memory (PIM) is when a processor is placed within the random access memory (RAM) on a chip[19]. The purpose of PIM is to reduce the latency by increasing the transfer rate between the processor and the memory system. PIM helps reduce the transfer rate because the performance of the processor is directly related to the stack performance since the majority of the active data and memory being accessed is within the stack. The issue of waiting for data to be fetched from memory is known as the Von Neumann bottleneck[20]. PIM is also able to decrease power consumption since the processor and memory are physically closer.

The advantages of PIM are the reduction in power and memory access latency which it provides[19]. The disadvantages of PIM are limitations of the amount of

memory which is available, which also causes chips with PIM implemented to be less customizable[19]. PIM requires a larger chip size than standard chips which causes it to have less modularity[19]. The heat emitted from chips which use PIM is an issue as well because of the layout of the chip with the processor and memory so close together[19].

6.4 Reduction of Average Memory Access Time

This section will discuss a number of ways in which the main memory access time may be reduced. This section will also detail which processes are used in practice, not just discussed in theory.

Memory interleaving is one method used by computer architects in order to reduce average memory access time (AMAT)[21]. Memory interleaving distributes sequential data across multiple chips rather than in order all on one chip allowing multiple sections of the sequential data to be accessed simultaneously through the same index[21]. This decreases AMAT by reducing the number of memory accesses and index calculations required for a memory access.

Cache memory is another method of reducing AMAT. Cache is a small amount of fast memory which stores data that is frequently accessed[1]. Multiple levels of cache may exist in order to provide more potential reduction. The access reduction is not achieved on the first access, rather on any later accesses once the data is in the cache. Higher levels of cache, those which are accessed more frequently, are smaller to allow higher speeds. Cache decreases AMAT by storing data closer to the processor allowing fewer memory accessed to propagate to main memory.

6.5 Prefetching

This section explains what prefetching is, then to provide a number of prefetching methods. This section will provide examples of both software and hardware prefetching algorithms.

Prefetching is used to reduce the average memory access time by fetching memory from main memory before it is needed[22]. Different prefetching schemes use different methods, but overall, prefetching occurs as a result of a cache miss. Instead of only fetching the requested memory, prefetching allows the system to fetch additional memory in an attempt to save time later.

In order for prefetching to be implemented in software, the programmer must have extensive knowledge of the hardware, and insert fetch instructions into the machine language manually, or through an educated compiler[22]. Due to this, software prefetching is more complicated than hardware prefetching, and is used less frequently since the compiler generally does not have the necessary information

about the hardware to make prefetching decisions, and manually adding fetch instructions to the machine language code is tedious.

Hardware prefetching is widely used as it is simpler than software prefetching as the hardware has all the information needed to appropriately fetch additional instructions. Five different hardware prefetching methods to be addressed in this section can be split into two categories: sequential prefetching and data structure prefetching.

The three sequential prefetching methods include prefetch on miss, tagged prefetch, and adaptive prefetch. The prefetch on miss strategy fetches the next sequential block of memory in addition to the requested block of memory as a result of a cache miss[22]. This strategy has approximately a 50% effectiveness rate. The tagged prefetch method assigns a tag bit to every block of memory. This bit is used to detect when a block is demand-fetched, or a prefetched block is referenced for the first time[22]. In both cases, when a block is fetched, the next sequential block in memory is also fetched. This strategy is slightly more effective than the prefetch on miss strategy due to the principles of spatial locality. The adaptive sequential prefetch method modifies the prefetch on miss strategy to not only fetch the next sequential block, but to also fetch as many sequential blocks as deemed appropriate by the degree of spatial locality of the system[22].

The two data structure prefetch methods are dependence based, and hardware based pointer data prefetch. The dependence based prefetch method identifies pointers in memory, then looks at the address of the pointer, as well as the address which it points to in memory. Based on this, when the pointer is loaded, the place in memory which it points to is prefetched. This method is not always effective because not all pointer loads are address loads[22]. The hardware based pointer data prefetch method identifies load instructions which are responsible for advancing a pointer through a linked list. This method prefetches all possible addresses for this operation and stores the data into a prefetch buffer which has a one computational cycle latency, similar to that of a first level cache[22].

6.6 Effects of Prefetching

This section will compare the performance of computers which use prefetching to the performance of computers which do not use prefetching. The primary metric to be the focus of this section is overall number of misses. This section will also discuss the circumstances in which both software and hardware prefetching are used.

In a study[23] conducted by Wei-Chung Hsu and James E. Smith it was shown that systems which use hardware prefetching out-perform systems without hardware prefetching. Systems without prefetching relied on line size to improve miss rate. The optimal line size varied from 64 word lines to 128 word lines depending on the overall size of the cache[23]. For the smaller caches, 64 word lines was more effective, and for larger caches, 128 word lines was more effective due to cache pollution. The concept of cache pollution is the ejection of potentially useful lines in the cache in order to insert a line which has fewer useful instructions[23].

In the same study[23], the benefits of implementing hardware prefetching were based on which prefetching strategy was used. This study observed fall-through and target prefetching. Fall-through prefetching most closely matches the adaptive sequential method previously discussed. Target prefetching most closely matches dependence based prefetch combined with prefetch on miss. The fall-through prefetch method provided a reduction in misses by one third as compared to the system with no prefetching. The longer line size does not provide an advantage with this method of prefetching. This is because longer lines simulate the same spatial locality which the fall-through prefetcher implements on its own[23]. Similarly, the target prefetching method provided approximately the same results as the fall-through prefetch method. Of note, the target prefetching method performed better with larger line sizes because it does not have as much built in spatial locality as the fall-through prefetching method[23].

7 Operating System

The operating system is an important part of a system which includes processors, especially with regards to performance metrics such as execution time. The operating system is responsible for the scheduling of processes on processors.

Section 7.1: Heterogeneous System Process Selection With Affinity discusses how the operating system uses affinity to select which processor a process may run on when there are multiple different processors that the process may run on. Section 7.2: Heterogeneous System Process Selection Without Affinity continues the discussion begun in Section 7.1: Heterogeneous System Process Selection With Affinity except ignoring process affinity to present other ways the operating system determines on which processor to let a process run.

7.1 Heterogeneous system process selection w/affinity

This section will explain how the operating system (OS) selects a specific core for a process to run on when there are multiple cores capable of running that process. This discussion will discuss the aforementioned process with regards to a heterogeneous system.

The OS uses processor affinity to determine which processor core a process should run on. Processor affinity is a term used to describe the association between processes and processor cores[24]. One instance of processor affinity is preferred processor[24]. A preferred processor for a process is determined based on whether the process has executed on a processor before. The purpose of a preferred processor is to attempt to maximize the probability that data from a process may remain in the cache memory from a previous execution. Preferred processors may also relate to the performance capabilities of a particular processor. If all of the processors are the same regarding performance, and the process in question has never executed, the processor affinity for the process is the same for all available processors. The OS will try to schedule a process to run on the processor for which it has the highest affinity. However, the OS will not necessarily stop another process from executing to allow a different process to execute on the processor for which its affinity is highest, especially if other processors are available. Processor affinity acts as a guideline for the OS rather than a mandate.

7.2 Heterogeneous system process selection w/o affinity

This section expands on the previous section, Section 7.1, except to ignore the concept of affinity within the operating system (OS).

OSs are able to use heuristic models in order to determine the power consumption of a particular process when it is executed on a particular processor. Using this data, the OS is able to make a decision as to which process should run on which processor[25]. This heuristic model not only is able to account for power consumption, but for overall throughput as well. Once data is gathered on the currently running processes, calculations are performed to determine whether the current processes are running on processors which provide the system overall with the lowest power consumption with the highest throughput[25]. Since a heuristic is being used, it is impossible to achieve a perfect balance between throughput and power, but an optimized balance based on predetermined system preferences will be achieved.

Specifically, when operating on a big.LITTLE processor system with four different core types, this heuristic to dynamically map processes will continuously swap the executing processes in order to achieve optimal throughput to power balance, first focusing on throughput, then optimizing for power[25].

8 Conclusion

This paper presented information discovered through research on a number of topics which were introduced in CEC470. Starting at a high level, this paper gave information regarding high level design decisions which must be made before designing or selecting a processor. Next, this paper discussed branch prediction and pipelining, moving the content to a deeper level, with a focus on processor optimizations. Continuing to move deeper into processor

design, this paper discussed registers and their role in data storage. Related, this paper then explored memory and its role in a processor. Finally, moving away from the physical aspects of the processor, this paper discussed how operating systems interact with processors.

No new or unique information was presented in this paper, as it is a survey of computer architecture overall and meant as a learning tool in the context of an Honors Directed Study.

9 References

[1]Hennessy, John L, David A Patterson, and Andrea Arpaci-Dusseau. Computer Architecture. Waltham: Morgan Kaufmann, 2012. Print.

[2]Phelan, Richard. Improving ARM Code Density And Performance New Thumb Extensions To The ARM Architecture. 1st ed. ARM Limited, 2003. Web. 1 Feb. 2017.

[3]Hyari, Abeer. A Comparative Study On Heterogeneous And Homogeneous Multiprocessors. 1st ed. University of Jordan, Computer Engineering Department, 2009. Web. 1 Feb. 2017.

[4]Branch Prediction. 1st ed. 2006. Web. 1 Mar. 2017.

[5]Branch Prediction. 1st ed. New Jersey Institute of Technology. Web. 1 Mar. 2017.

[6]Big.LITTLE Technology: The Future Of Mobile Making Very High Performance Available In A Mobile Envelope Without Sacrificing Energy Efficiency. 1st ed. ARM Limited, 2013. Web. 22 Mar. 2017.

[7]Prabhu, Gurpur M. "Pipeline Hazards". Web.cs.iastate.edu. Web. 15 Feb. 2017.

[8]Instruction Scheduling. 1st ed. Web. 5 Apr. 2017.

[9]Koppanalil, Jinson, et all. A Case For Dynamic Pipeline Scaling. 1st ed. North Carolina State University, 2002. Web. 5 Apr. 2017.

[10]Big.LITTLE Technology: The Future Of Mobile. 1st ed. ARM Limited, 2013. Web. 25 Apr. 2017.

[11]"Register File". En.wikipedia.org. 2017. Web. 15 Feb. 2017.

[12]"Register Renaming". En.wikipedia.org. N.p., 2017. Web. 15 Feb. 2017.

[13]"CPU Registers". Doc.ic.ac.uk. Web. 1 Mar. 2017.

[14]"Heap (Data Structure)". En.wikipedia.org. 2017. Web. 3 May 2017.

[15]"Difference Between Stack And Heap - Programmer And Software Interview Questions And Answers". Programmer and Software Interview Questions and Answers. 2017. Web. 22 Mar. 2017.

[16]Mittal, Sparsh, and Jeffrey S. Vetter. "A Survey Of Techniques For Modeling And Improving Reliability Of Computing Systems". IEEE Transactions on Parallel and Distributed Systems 27.4 (2016): 1226-1238. Web. 3 May 2017.

[17]"Error Detection And Correction". En.wikipedia.org. 2017. Web. 22 Mar. 2017.

[18]"Hamming Code". En.wikipedia.org. 2017. Web. 22 Mar. 2017.

[19]Rouse, Margaret. "What Is Processing In Memory (PIM)? - Definition From Whatis.Com". SearchBusinessAnalytics. Web. 22 Mar. 2017.

[20]Rouse, Margaret. "What Is Von Neumann Bottleneck? - Definition From Whatis.Com". WhatIs.com. Web. 22 Mar. 2017.

[21]"Interleaved Memory". Cs.umd.edu. Web. 5 Apr. 2017.

[22]Pourdowlat, Pirouz et al. Hardware Prefetching Schemes. 1st ed. 2005. Web. 19 Apr. 2017.

[23]Hsu, W.-C., and J.E. Smith. "A Performance Study Of Instruction Cache Prefetching Methods". IEEE Transactions on Computers 47.5 (1998): 497-508. Web. 19 Apr. 2017.

[24]White Paper Processor Affinity Multiple CPU Scheduling. 1st ed. TMurgent Technologies, 2003. Web. 1 Mar. 2017.

[25]Dynamic Thread Mapping For High-Performance, Power-Efficient Heterogeneous Many-Core Systems. 1st ed. IEEE, 2013. Web. 22 Mar. 2017.