Lakehead University

Knowledge Commons,http://knowledgecommons.lakeheadu.ca

Electronic Theses and Dissertations | Electronic Theses and Dissertations from 2009

2017

Mitigating the impact of decompression latency in L1 compressed data caches via prefetching

Rea, Sean

https://knowledgecommons.lakeheadu.ca/handle/2453/4134

Downloaded from Lakehead University, KnowledgeCommons

Mitigating the Impact of Decompression Latency in L1 Compressed Data Caches via Prefetching

by

Sean Rea

A thesis

presented to Lakehead University

in partial fulfillment of the requirement for the degree of

Master of Science

In

Electrical and Computer Engineering

Thunder Bay, Ontario, Canada

September 1, 2017

Copyright 2017 Sean Rea

Abstract

Expanding cache size is a common approach for reducing cache miss rates and increasing performance in processors. This approach, however, comes at a cost of increased static and dynamic power consumption by the cache. Static power scales with the number of transistors in the design, while dynamic power increases with the number of transistors being switched and the effective operating frequency of the cache.

Cache compression is a technique that can increase the effective capacity of cache memory without experiencing the same gains in static and dynamic power consumption. Alternatively, this technique can reduce the physical size and therefore the static and dynamic energy usage of the cache while maintaining reasonable effective cache capacity. A drawback of compression is that a delay, or decompression latency, is experienced when accessing the compressed data, which affects the critical execution path of the processor. This latency can have a noticeable impact on processor performance, especially when implemented in first level caches.

Cache prefetching techniques have been used to hide the latency of lower level memory accesses. This work aims to investigate the combination of current prefetching techniques and cache compression techniques to reduce the effect of decompression latency and therefore improve the feasibility of power reduction via compression in high level caches.

We propose an architecture that combines L1 data cache compression with table-based prefetching to predict which cache lines will require decompression. The architecture then performs decompression in parallel, moving the delay due to decompression off the critical path of the processor. The architecture is verified using 90nm CMOS technology simulations in a new branch of SimpleScalar, using Wattch as a baseline, and cache model inputs from CACTI. Compression and decompression hardware are synthesized using the 90nm Cadence GPDK and verified at the register-transfer level.

The results of our verifications demonstrate that using Base-Delta-Immediate (BΔI) compression, in combination with Last Outcome (LO), Stride (S), and Two-Level (2L) prefetch methods, or hybrid combinations of these methods (S/LO or 2L/S), provides performance improvement over Base-Delta-Immediate (BΔI) compression alone in L1 data cache. On average, across the SPEC CPU 2000 benchmarks tested, Base-Delta-Immediate (BΔI) compression results in a slowdown of 3.6%. Implementing a 1K-Set Last Outcome prefetch mechanism improves slowdown to 2.1% and reduces the energy consumption of the L1 Data Cache by 21% versus a baseline scheme with no compression.

Acknowledgements

For introducing me to computer architecture and providing me with an engaging topic to study while at Lakehead, thank you Dr. Atoofian. I have learned a lot under your supervision.

Thank you to Dr. Mansour and Dr. Tayebi for endorsing my application for part-time studies back in 2012 and to Dr. Natarajan for keeping an eye on me for the first few years. Thank you Dr. Christoffersen for your active role in enabling students at Lakehead to have access to Cadence tools.

Thank you to all my family who have given their support and made every accommodation possible for me to pursue this degree. Most importantly, thank you Marcelle and Adam for supporting my decision to continue with my studies.

Sean Rea
rea@ieee.org

Contents

List of Figures

List of Tables

List of Symbols

c_i	carry-in
c_{i+1}	carryout
E	energy
g_i	generate function
G_i	block generate function
M	maximum number of SimPoint clusters
p_i	propagate function
P_i	block propagate function
t	time
$t_{VCD,\ ps}$	value change dump timestamp (in picoseconds)

List of Abbreviations

2L	two-level prefetcher
BΔI	base-delta-immediate compression
BBV	basic block vector
CMC	Canadian Microelectronics Corporation
CMOS	complementary metal-oxide-semiconductor
CPI	cycles per instruction
CPU	central processing unit
EDP	energy-delay product
FIFO	first-in, first-out
FLAC	free lossless audio codec file format
FPC	frequent pattern compression
GHB	global history buffer
GPDK	generic process design kit
IPC	instructions per cycle
L1	level-1 cache
L2	level-2 cache
L3	level-3 cache
LO	last outcome prefetcher
LRU	least recently used
PC	program counter
PDK	process design kit
PNG	portable network graphics file format
RISC	reduced instruction set computer
simpoint	simulation point
S	stride prefetcher
S/LO	hybrid stride/last outcome prefetcher
SPEC	Standard Performance Evaluation Corporation
VCD	value change dump
ZCA	zero-content augmented cache
ZIP	ZIP file format

Chapter 1

Introduction

In 2016, it has been estimated that the world creates 2.5 quintillion bytes of data per day [1]. At the time, that estimate suggested that 90% of the world's data had been created in the previous two years alone. In as early as 2013, it was approximated that Information and Communication Technologies were consuming nearly 10% of the world's electricity generation [2]. With this rate of data growth, and the current impact computing has on the world's energy consumption, there is a need to investigate ways to improve the way we store and process data.

Linked with the growth of our data generation is the emergence of a rapidly growing mobile device market. This market relies heavily on low power, battery supplied devices. Unavoidably, the best way to provide a longer battery life for these devices is for the devices themselves to consume less power. While improvements can be made to the devices themselves (e.g. supply voltage and device size), in many applications, the best way to reduce power is to find efficiencies at the architectural level [3]. Redefining the architecture can result in orders of magnitude in reduction of power depending on the specific application.

When we look at the data we are creating, it is clear that patterns exist that create inefficiencies in the way it is stored and processed [4]. Data patterns may consist of values that are repeated over-and-over again, values that are very close to each other, and even large sets of null data. Some of the greatest sources of data in the world today are the cameras on our mobile devices. Images are a great example of data that consists of patterns. Pixel data contains sequences of values, which can be identical or very close in magnitude (when looking at color value, brightness, etc.). Programs that manipulate this image data generally handle large data arrays. These arrays are frequently initialized to some repeated value, often zero. And frequently enough, the developers of those programs may over-provision data types to hold that data such that most of it goes unused (narrow data). These inefficiencies in our data contribute to unnecessary storage and processing.

In computing architecture, we utilize a memory hierarchy to ensure that the data we use most frequently is closest to the CPU and therefore accessible as fast as possible. The closest memory spaces, L1 and L2 cache, take up large areas on-chip and consume large amounts of power. Depending on the architecture, cache memory in a processor can account for upwards of 40% of the total power budget [5]. Because the cache must handle our data, which is full of inefficiencies, a significant portion of this energy consumption could be avoided via compression.

Compression is possible by replacing the most inefficient patterns with a set of smaller representations, known as encoding. Encoding can be done with a fixed dictionary or a compression scheme can dynamically and iteratively assign code words to patterns. Most of our data is compressible to some extent, whether it is text, image, or audio data. This is why, in main memory, it is common to store and transfer large files in a compressed format (e.g. ZIP, PNG, FLAC).

Cache-level compression in a processor is a technique that can increase the effective capacity of the cache, and therefore improve performance of the processor, by compressing cache lines before they are stored in the cache. Alternatively, this same method can be used to reduce the physical size of the cache and therefore reduce the power consumption.

Typically, cache lines consist of several bytes of data. In set-associative caches, multiple lines, or ways, may be stored at a given cache index. Each of these ways store a complete uncompressed cache line. If we can compress the size of these cache lines, we can store more data in each set. Alternatively, we could reduce the physical size of the cache and store the same, or similar, amounts of data.

The power of a cache depends on its size and the frequency of accesses. By reducing the total size of the cache, we can significantly reduce the power consumption. In addition, the size of the data lines we read from the cache are potentially reduced in size as well. Therefore, we can model the savings in dynamic power consumption in the cache by considering the size ratio of compressed data vs an uncompressed cache line.

In the past, researchers have avoided implementing compression in high-level caches such as L1 data cache because of the impact decompression latency has on the overall performance of the processor [4, 6, 7]. Access times at this level are in the order of a few clock cycles. To add even a few clock cycles to this access time will cause significant performance delays and defeat the intent of the high-level cache.

However, it is possible to implement additional techniques, such as prefetching, to remove the some of the burden of decompression from the processor's critical path. If this approach is successful, it could lead to improving the feasibility of implementing compression in high-level caches (specifically L1 data cache).

Our work focusses on the combination of data cache compression and table-based prefetching to explore the feasibility of implementing compression in L1 data cache. We evaluate this new architecture by examining what impact it has on the performance and power consumption of a CPU during the execution of standard benchmarks. Specially, this work makes three key contributions:

(1) An architecture is proposed that combines compression, specifically Base-Delta-Immediate compression, in L1 data cache with table-based prefetching methods, such as Last Outcome, Stride, and Two-Level, to predict which cache lines will require decompression. The architecture then performs decompression in parallel, therefore moving the delay due to decompression off the critical path of the processor.

(2) Modifications are made to Wattch [5], a branch of SimpleScalar [8] that is an open-source processor modelling tool for analyzing and optimizing power consumption at the architectural level. This tool is extended to model Base-Delta-Immediate compression in combination with table-based prefetching to show the benefit of performing decompression as a parallel activity to execution and increase the feasibility of implementing compression in L1 caches.

(3) 64-byte compressor and decompressor hardware is designed in 90nm CMOS and tested for implementation with Base-Delta compression. Static and dynamic power analysis is performed on the new hardware, reinforcing its suitability for use in a power-reducing compressed cache scheme.

The remainder of this thesis is organized into five chapters. **Chapter 2** provides an overview of cache compression and prefetching and recent research that has been done in these areas. In **Chapter 3**, we define the proposed compression and prefetching architecture, discuss what changes are necessary to accommodate the new architecture in a conventional superscalar processor, and provide the details of the hardware design for the compressor and decompressor units. **Chapter 4** discusses the tools used and modified to model the compression and prefetching architecture. In **Chapter 5**, the results of simulation are provided and discussed. Finally, **Chapter 6** provides a summary of the work done, the significance of the results, and future work that could be done to advance this research.

Chapter 2

Background and Related Work

In this chapter, we review the concepts of memory hierarchy, compression, data value prediction, and prefetching. We review existing work in the areas of cache compression, data value prediction, and prefetching and go into detail of the operation of one compression scheme, Base-Delta-Immediate, three prediction schemes: Last Outcome, Stride, and Two-Level as well as hybrid combinations of these schemes. Finally, the motivation behind this work is presented.

2.1 Memory Hierarchy

The speed in which a processor can read information from memory has a great impact on the performance of that processor. Fast memory, however, is expensive. For this reason, memory is organized into levels that exploit small amounts of fast memory close to the processor, and larger, slower levels of memory farther away. This organization is referred to as the Memory Hierarchy. In this hierarchy, between the central processing unit (CPU) and "main memory" are various levels of cache memory.

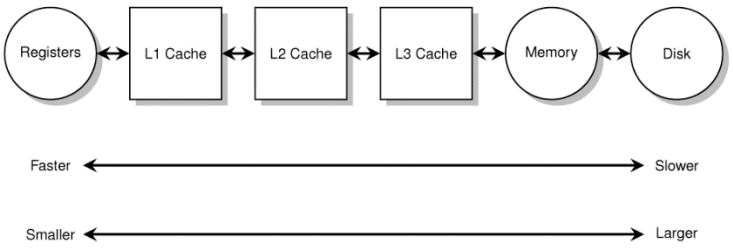

Figure 2.1 – Memory Hierarchy

When the data for a given address is stored in the high-level cache (i.e. L1 cache), then the processor has fast access to this information. If the data is not there, the processor must retrieve it from lower levels. This is referred to as a cache miss. Cache misses are classified by three types: compulsory misses, capacity misses, and conflict misses [9]. Compulsory misses occur during start-up, when no information exists in

the cache. Capacity misses occur if the cache is not large enough for all the blocks required during the execution of a program and blocks are discarded. If these discarded blocks must be read again, they must be fetched from lower levels. Conflict misses occur if the cache is not fully associative. In this case, blocks may be discarded even before the cache is full.

Increasing the capacity of the cache can reduce the number of capacity and conflict misses and therefore increase performance. This, however, comes at a cost of increased power consumption.

2.2 Cache Compression

Similar to data files in main memory, the data within cache memory consists of patterns that can be exploited by compression techniques to save space. Cache compression is a method that can be used to increase the capacity of the cache without experiencing the same increase in power consumption.

Because the intent of cache memory is to provide low latency access to data, compression and decompression must be performed at the hardware level in the processor rather than at the software level, as is commonly performed on files in main memory. This same requirement for low latency cache access is why most of the previous work done on the topic of cache compression focusses on low-level cache (i.e. L2 cache and L3 cache). Because L1 caches typically have access latencies in the order of a few clock cycles, adding a decompression latency on top of that can degrade performance beyond acceptable levels.

The ideal compression scheme for implementation in L1 cache is one that is, of course, fast, but is also capable of encoding the most common patterns that exist within data stored in memory. These most common patterns can be grouped into four main categories: zeros, repeated values, narrow values, and other patterns [4].

Zeros

Zero values are widely used throughout programs, primarily in variable initializations, null pointers, false boolean values, and sparse matrices [4].

Repeated Values

Similar to zero values, repeated values may appear in the form of variable initializations. Another cause for repeated values is image data. Adjacent pixels tend to contain similar information such as colour data [4].

Narrow Values

It is common for developers to over-allocate space to variables to protect from overflow during execution of a program. In some cases, these variables never come close to their maximum values. A small value stored as a large data type is considered a narrow value [4].

Other Patterns

This group is not meant to include all other patterns, but rather patterns that specifically have low dynamic range. For example, an array of pointers that all point to the same region of memory [4].

2.2.1 Related Work in Cache Compression

Research has been done to evaluate hardware-based data compression in CPU caches [4, 6, 7, 10, 11, 12] as well as in GPGPU [13]. The following papers have explored which methods exploit the most opportunities in data patterns and at which levels in cache they are most beneficial. A common understanding among this work is that decompression latency is a problem when implementing in fast caches and is cited as the reason for avoiding L1 compression in some works [4, 6, 7].

In [7], the authors present Frequent Pattern Compression (FPC), which compresses data that fits into one of seven patterns. Each 32-bit word is evaluated separately so data is not compressed spanning multiple words. The scheme is evaluated in L2 cache with L1 cache being left uncompressed. The design is evaluated against the Wisconsin Commercial Workload Suite and six benchmarks from the SPEC CPU 2000 suite. The scheme provides compression ratios ranging from 1.0 to 2.4 over all their benchmarks. This scheme captures the main three groups of patterns that exist in data. However, this scheme does not address the behaviour of low dynamic range that data exhibits.

In [11], the authors exploit the common scenario of storing null data in caches by augmenting the uncompressed cache with an additional cache that is only required to store the addresses of zero-content cache lines. The authors evaluate this zero-content augmented (ZCA) cache in every combination of cache level from L1 to L3. They found that implementing ZCA in L3 alone was sufficient to experience most of the benefit and found up to a 22% speedup when run against SPEC CPU 2000 benchmarks. Because this scheme only looks at zeros, decompression latency is not an issue, and the authors are able to explore this technique in all levels of cache without affecting the read latency. This scheme, however, does not address most of the patterns that exist within cache data.

In [12], the authors compress the cache line by encoding 32-bit words that appear in a predefined list of "frequent values." The scheme requires that a cache line be compressible to 50% in size or less or it is not compressed at all. Encoding bits are required for each word in the cache line. The authors determine that their scheme can improve the miss rate for six integer benchmarks from SPEC CPU 95 as much as 36.4%. Due to the table-lookup nature of the scheme, it cannot capture all repeating values efficiently. As well, it misses the important narrow values that occur within the data.

In [4], the authors present a new compression scheme that this work builds upon, called Base-Delta-Immediate. In addition to null data and repeating values, this work exploits two trends in data called narrow values and low dynamic range. The scheme compresses cache lines that can be represented as a single base and an array of small deltas. The authors evaluated their scheme against the SPEC CPU 2006 benchmark suite, among other benchmarks. The authors achieve an average compression ratio of 1.53 across all benchmarks when compressing L2 cache. Due to the impact decompression latency would have on L1 cache, the authors focus on L2. Because this scheme addresses all the patterns discussed in the other works, and more, it is the compression scheme we use. We will address the issue of implementing this scheme in L1 cache by combining common prefetching techniques to mask the effect of the decompression latency.

2.2.2 **Base-Delta**

Back to [4], the authors first describe the foundation of their scheme, called Base-Delta. Base-Delta is a cache compression scheme that stores a cache line as one large base value along with an array of smaller deltas. The concept behind the scheme is that, for many cache lines, the data values have a low dynamic range (the difference between values is small). For example, Figure 2.2 shows an example of how a 32-byte cache line may be compressed using a **Base 4 Delta 1** compression scheme.

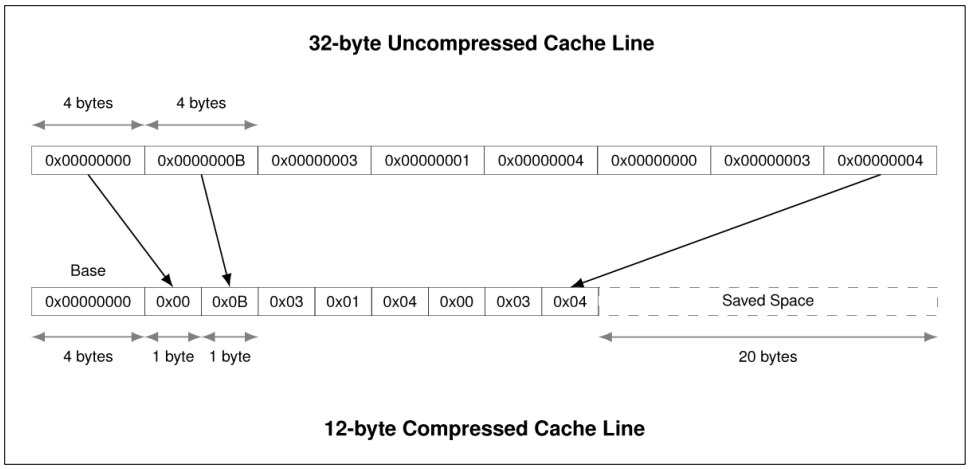

Figure 2.2 – 32-Byte Cache Line Compressed with Base-Delta

From the figure, you can see how this cache line benefits from low dynamic range. In this example, the **Base 4 Delta 1** scheme is used. This means the chosen size of the base is 4 bytes, and the size of the deltas is 1 byte.

In some cases, compression may benefit from having multiple bases. For example, the cache line in Figure 2.3 clearly shows patterns with low dynamic range around two bases.

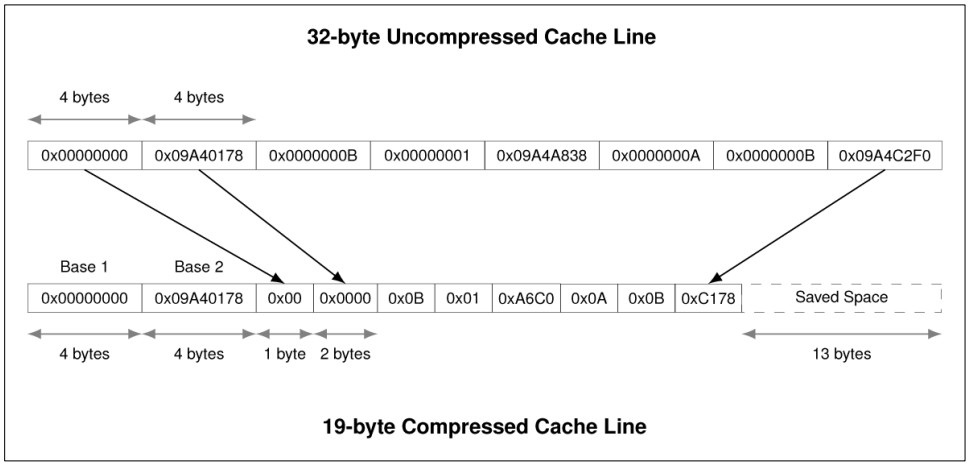

Figure 2.3 – 32-Byte Cache Line Compressed with Base-Delta (2 Bases)

Determining two optimized bases is a high latency task that is not feasible during execution. The authors resolve this issue by implementing the **immediate** portion of their compression making Base-Delta-Immediate [4].

2.2.3 **Base-Delta-Immediate**

Base-Delta-Immediate (BΔI) compression implements a 2-base Base-Delta scheme where one base is always zero. This method sees much of the benefit of a 2-base system, without adding the need to store a second base. To implement this immediate base, an array of flag bits called the **immediate mask** is included in the tag to identify which deltas refer to the base and which refer to zero.

To change a conventional cache into a Base-Delta-Immediate cache, we double the number of tags. This allows us to utilize the vacant space in the cache created during compression. Next, we modify how the tags point to the data stored in the cache. The data array is divided into 8-byte segments rather than 64-byte blocks. Rather than pointing to a constant 64-byte block, the tag now points to a variable-size compressed block. The location of the compressed block at a given cache index is determined by summing the size of cache blocks stored in front of it. Lastly, we add the encoding bits (and immediate mask, as mentioned above) to the tag. This allows us to define the type of compression applied to the data for a given way. These changes to the architecture are shown in Figure 2.4.

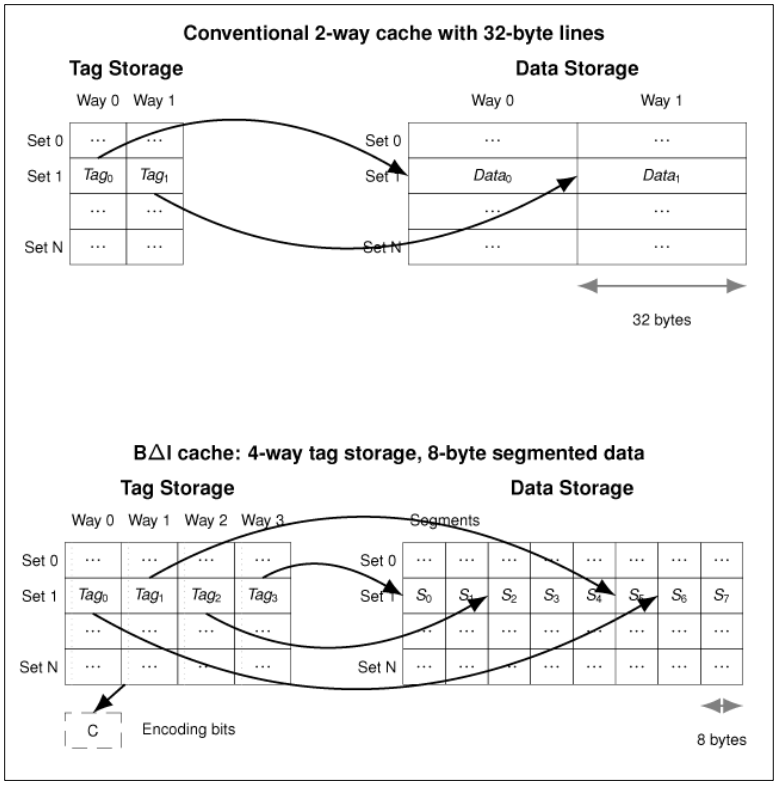

Figure 2.4 – Changes to Tag and Data Architecture for BΔI Compression

These changes are implemented functionally in SimpleScalar and have their power and access time impact modeled using CACTI (see Chapter 4, Simulation Methodology). The additional hardware required for compressing and decompressing the BΔI cache lines is discussed separately in Chapter 3.

2.3 Data Prefetching and Data Value Prediction

Prefetching is a method that can be used to reduce the miss rate of all three types of cache misses. Data can be prefetched (read in parallel to execution, before it is requested), either directly into the next-level-up cache or into a custom buffer that can be accessed faster than main memory [9]. While successful prefetches can reduce memory latency and improve overall processor performance, unused prefetches will have a negative impact on power consumption while having no positive impact on performance.

Data value prediction is similar to prefetching such that it employs a table-based predictor to improve performance of the processor. Unlike prefetching, data value predictor tables do not store the address in memory where the data exists. Rather, it stores the data itself – specifically the result of single-register producing instructions. The processor then continues execution using this predicted result. If an incorrect prediction is made, the processor pipeline must be flushed of any instructions that depend on this data.

The approach we take concerning prefetching is neither a direct application of prefetching nor of data value prediction. Rather, our prefetcher predicts which address in the L1 data cache will be accessed based on the program counter of each load instruction. Then, in parallel, the processor decompresses this data in the cache, if it is compressed, and inserts it into an external buffer.

The key similarities between prefetching and data value prediction are the prediction table methods used. These methods have been explored extensively for both applications. We look at this past research to determine which tables are best suited for our application.

Reviewing the prediction schemes presented in literature, five types stand out as candidates for this work, as discussed in [14]. The simplest, Last Outcome, is evaluated first to determine how quickly we can recover the cost of decompression with the lowest possible complexity. Next, Stride and the hybrid Stride/Last Outcome predictors are used and evaluated. Lastly, the Two-Level and hybrid Two-Level/Stride predictors are looked at to capture more complex patterns. Global History Buffer is discussed as a potential extension by evaluating the benefit of predictor tables with a depth greater than one.

2.3.1 **Related Work in Prefetching**

In [14], data value predictors are discussed using Last Outcome, Stride, 2-Level Pattern History methods, and two hybrids of these methods. Data Value Prediction uses prediction tables in the same way as prefetching. Data Value Predictors predict the data value rather than the address of the data in memory. Accuracy is critical for Data Value Predictors because if an incorrect prediction is made, any progressed instructions dependant on this value much be flushed out of the pipeline. The authors found that the Last Outcome scheme was correct 28-62% of the time and incorrect up to 72% of the time. Stride was correct 35-77% and 3-6% incorrect. Two-Level makes minor improvements in prediction accuracy over Last Outcome (1-3%). However, the two-level table scheme greatly improves the misprediction rate to 1-13%. The first hybrid scheme implements Last Outcome when Stride is not in steady state. This results in a correct predictions rate of 49-80% and incorrect predictions 20-51%. The second hybrid predictor combines the 2-Level scheme with Stride. This scheme made correct predictions 50-82% of the time and mispredictions only 5-18% of the time. This scheme is, however, the most complex to implement.

An important takeaway from this research are the incorrect prediction rates. In the Data Value Predictor method, if we make an incorrect prediction, we must flush the processor of any instructions that depend on the incorrect value. In our compression-prefetching method, we do not have to purge any information. However, incorrect predictions will cause unnecessary cache accesses which will increase power and may evict useful data from the decompression buffer (depending on the depth of the buffer).

In [6], the authors present a two-table prefetching scheme called Global History Buffer (GHB). GHB itself, as with the above research on Data Value Predictors, explores multiple prediction schemes: Address Correlation, Distance Correlation and Constant Stride. The key benefit of GHB is the two independently sized tables. The first is the Index Table which only stores the tag and a pointer to the head of a list stored in the GHB Table. The GHB acts as a circular buffer, keeping only the latest information. The authors investigate different table configurations with a degree of four (values prefetched each access). They found that GHB Distance prefetching resulted in a 20% speedup over conventional table Distance prefetching when indexed by the miss address and 6% when indexed by the program counter (PC).

The key enabler of the Global History Buffer is to minimize space and hold the latest information about cache misses. The same technique can be applied to our prediction table. However, this approach will only be useful if tables with a depth greater than 1 prove to be valuable. The Global History Buffer approach

will not be explored directly in this work, but this work can easily be extended to explore this possibility in a later study.

2.3.2 **Last Outcome**

For this thesis, prefetching techniques are used to predict which cache lines may require decompression from L1 data cache before the instruction is decoded using the PC of the instruction. The intent is to load the data from a compressed cache, decompress it, and make it available to the CPU in parallel with other stages to remove the bottleneck that is decompression latency.

The simplest scheme that will be implemented is Last Outcome. Figure 2.5 shows the traditional implementation of this scheme.

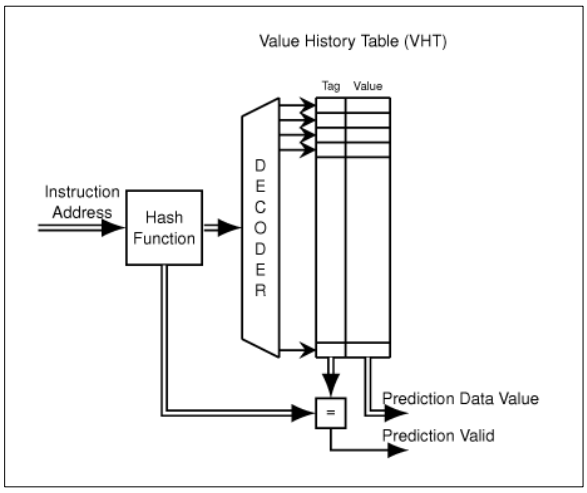

Figure 2.5 – Last Outcome Prefetching

In the figure, you see that a table exists to store two values for each entry: tag and value. **Tag** identifies the load instruction address and **Value** identifies the memory address loaded by that instruction. The configuration of the table can be varied similar to that of cache memory: associativity, depth (multiple values per tag), etc. Unlike the traditional architecture, in this work it is not necessary to verify if the prediction is correct in order to validate instructions with dependencies. If it is incorrect, the processor will merely suffer the full latency of decompression.

2.3.3 **Stride and Hybrid Stride / Last Outcome**

As mentioned earlier, the authors in [14] propose a hybrid prediction table that implements a Constant Stride prefetcher, and uses the Last Outcome result when the stride prefetcher is not in a steady state. We have already reviewed the behaviour of the Last Outcome table. So, let us review the functionality of a stride prefetcher.

Stride Prefetching

Similar to Last Outcome, we will be indexing the Stride table by Program Counter (PC) of each load instruction. When an entry is updated in the table, the value of the stride is calculated as the difference between the current and last memory addresses that are loaded. The state of the prefetcher can be Init, Transient, or Steady. Therefore, as shown in Figure 2.6, the table requires four columns: Tag, State, Value, and Stride.

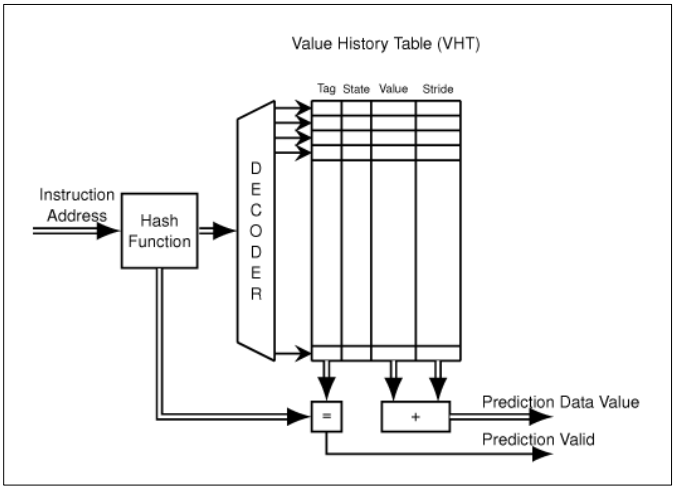

Figure 2.6 – Stride Prefetching

When a line in the table is first entered, there is no previous data from which to calculate the stride. The entry is in an initialized state. After this line is updated at least once, a stride can be calculated and the entry is in the transient state. The line will remain in this transient state until an update occurs that produces the same stride value as is currently stored in the table. When this occurs, the table is updated to steady state and this value for the stride is used. Figure 2.7 shows an overview of the state machine for this prefetcher.

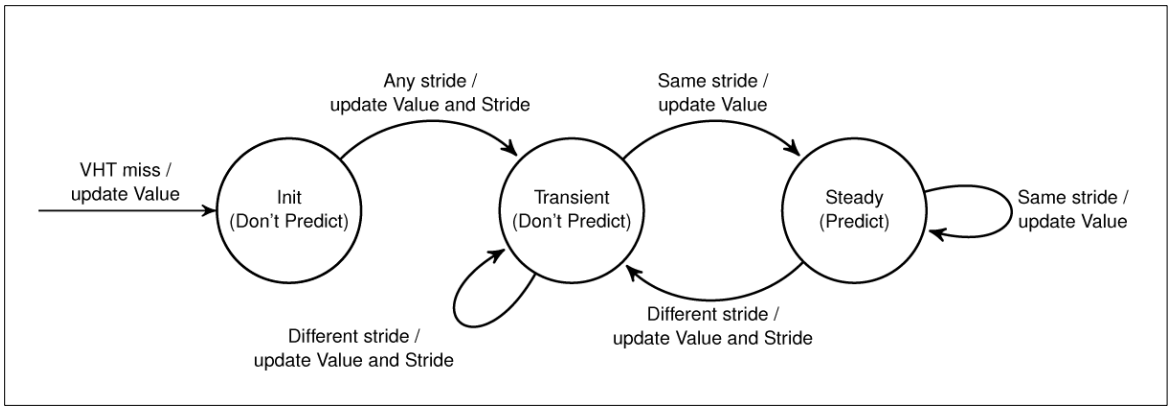

Figure 2.7 – Stride State Machine

2.3.4 **Two-Level and Hybrid Two-Level / Stride**

In addition to a hybrid S/LO prefetch table, the authors in [14] propose a hybrid prediction table that implements a Two-Level prefetcher, and uses the Stride result when the Two-Level prefetcher does not make a prediction. We have already reviewed the behaviour of the Stride table. So, let us review the functionality of a two-level prefetcher.

Two-Level Prefetching

Similar to the previous methods, we index the two-level table by Program Counter (PC) of each load instruction. When an entry is updated in the table, the LRU and pattern information are updated. If the address does not already exist in the table at this location, then the least-recently-used address is replaced. As shown in Figure 2.8, the table requires four columns: Tag, LRU, Value History Pattern, and Data Values. The data values in our case are load addresses.

A second table exists called the Pattern History Table. This table is indexed by the value history pattern and ranks the addresses stored as values in the value history table. During the FETCH stage, if we hit the prefetch table for a given load instruction PC, we index the PHT at the resultant pattern. If there exists a rank greater than a pre-set threshold, then we predict that value from the prefetch table. During the MEM stage, if a value in the table is the target of a load instruction, the rank is increased. The other values in the table are decreased such that there is a net zero ranking.

As with the other tables, we are not concerned if a misprediction is made as the result will simply be a full decompression latency seen by the MEM stage and an unused value eventually being evicted from the decompression buffer.

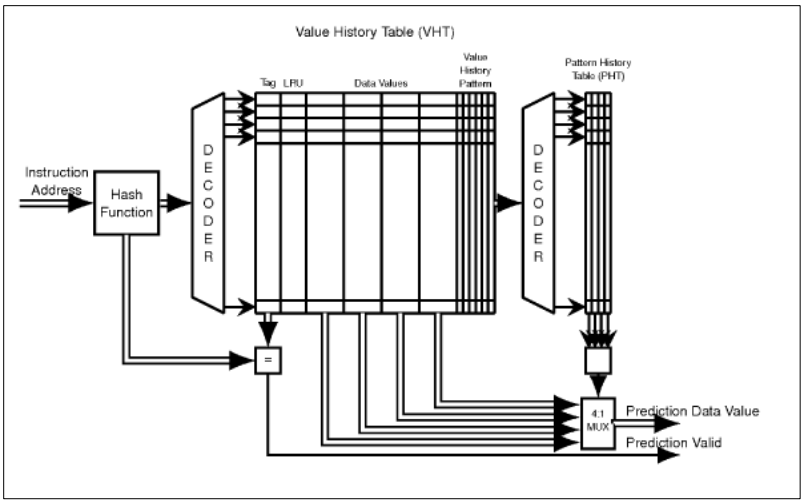

Figure 2.8 – Two-Level Prefetch Table and Pattern History Table

2.4 Thesis Motivation

Among all the works mentioned so far relating to cache compression, there exists two common gaps in the research. First, due to the impact of the decompression latency, apart from ZCA compression in [11], previous work has not used compression in L1 cache. We hope to address this issue by introducing prefetching of the decompressed information to side step this decompression latency in our architecture.

Second, all the works on compression that were mentioned above have focussed on using their compression schemes to improve performance of the cache. This work intends on reviewing the benefit of reducing the size of the cache to save power, and implementing prefetching as a means of maintaining performance.

Chapter 3

Cache Compression and Prefetching

In this chapter, we present a new architecture that combines cache compression with a prefetching mechanism to predict which memory addresses might require decompression based on the program counter of the instruction. The design of the compressor and decompressor hardware are discussed including the selection of the hierarchical carry-lookahead adder and the theory behind it.

3.1 Compression Architecture

The compression architecture discussed in this work is a detailed implementation of the high-level design presented by the authors in [4]. To implement this architecture for L1 data cache, we must consider when data would be compressed and when it would be decompressed in a superscalar processor. In this architecture, data compression takes place when data is written into the cache. That is, on any write operation or a read miss. Decompression takes place on a read hit. These events are summarized in Table 3.1 and represent the major insertion points for this new architecture in a superscalar processor.

Table 3.1 – Compression Events

L1 Data Cache Event	Action
Read Hit	Decompress
Read Miss	Compress
Write Hit	Compress
Write Miss	Compress

Decompression

On a **Read Hit**, we check the encoding bits for the hit cache line. If the line is encoded as compressed, we put this cache line to the decompression hardware. After a number of cycles equal to the decompression latency, the uncompressed result is available at the output of the decompressor hardware. If the line is not

compressed, the line is read as usual from the cache and is available in a number of cycles equivalent to the access time of the L1 data cache.

The additional logic required to check the encoding bits to determine if the line is compressed or not is included in the design of the decompressor. The CPU can read the result from the output of the decompressor regardless of compression. If the data is uncompressed, the result will available significantly faster as it is merely passing the input data through a single multiplexer.

Compression

On a **Read Miss** or **Write Miss**, we check the compressibility of the data to be written into the cache line. Based on the best possible compression scheme that this data fits into, the size of the cache line is determined. Then, we check the size of the cache set at the miss address. If there is room for the new cache line, compressible or not, then it is written to the cache. If there is not enough room, we evict data in the cache at that index in an LRU fashion until there is enough room.

On a **Write Hit**, we treat compressibility in the same manner as any miss with one minor change. When determining the space remaining in the set at the write address, we do not consider the space currently occupied by the hit address. This space will be overwritten by the new write data. This is critical as the new data may consume more space and may not even be compressible. If this is the case, we can expect one or more segments to be thrown from the cache to accommodate the new data.

Because updating the cache, and therefore compression, occurs off the critical execution path, this is not a time-critical task. Therefore, each instance of writing data to the cache goes through the compressor hardware. The compressor hardware itself, as you will see later in this chapter, checks the 64-byte data for compressibility, selects the optimum compression scheme (or no compression scheme), and outputs the encoding bits and data to be written (compressed or not). This means that no additional logic is required to be added to the CPU to accommodate compression.

3.1.1 Power Considerations

When modelling the power consumption for this compression architecture, we can take into consideration that fact that we are reading and writing smaller sets of data from and to the cache. Static power and tag dynamic power remain the same. However, we can represent the data array dynamic energy calculation as:

$$E_{dynamic,compressed} = \frac{size\ of\ compressed\ cache\ line}{size\ of\ uncompressed\ cache\ line} \times E_{dynamic,per\ access} \tag{3.1}$$

Relating to the cache events mentioned previously in Table 3.1, we can model power with respect to these events as well. The power impact is shown in Table 3.2.

Table 3.2 – Power Events

L1 Data Cache Event	Power Impact
Read Hit	Tag Read, Data Read
Read Miss	Tag Read, Tag Write, Data Write
Write Hit	Tag Read, Tag Write, Data Write
Write Miss	Tag Read, Tag Write, Data Write

3.2 Prefetching Architecture

From the compression architecture described earlier, you can see that we add decompression clock cycles for a read hit if the data is compressed in the cache. These additional cycles are necessary to allow the decompression hardware enough time to decompress the line. The purpose of the prefetching architecture is to avoid having this decompression of L1 data cache lines on the critical path of the processor. To accomplish this, we look for a way to perform decompression in parallel to other stages in the CPU. Consider the classic RISC architecture shown in Figure 3.1 [9].

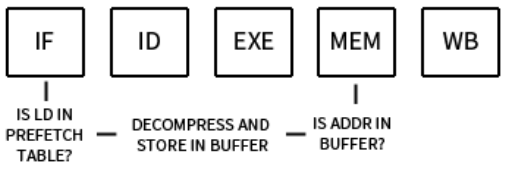

Figure 3.1 – Prefetching Applied to Classic RISC Architecture

In the Instruction Fetch (IF) stage, the program counter (PC) is used to access the next instruction from memory. At this point, it is important to know if the next instruction is a load instruction, if the data to be loaded is currently compressed in L1 data cache, and most importantly, what is the address of this data in memory. If we have this information, we can then read the compressed data from L1 data cache and decompress it in parallel with the Instruction Decode (ID) and Execution (EXE) stages.

In the Memory Access (MEM) stage, data is read from memory at the address determined during the ID and EXE stages. If we have a buffer containing cache lines that have been decompressed already, we will read from here rather than from the cache.

We actually do not need to know much about the instruction to accomplish this. Similar to [14] and [15], we index our prefetch table using only the PC of the instruction. We do not populate the table every time we generate a single register result, as done in [14], nor do we populate the table on a cache miss. Rather, in our architecture we add entries to our prefetch table each time we suffer the full decompression latency on a compressed cache hit. This means that each entry represents a load instruction. At a minimum, we store the address of the compressed cache line in the cache. Depending on the prefetch table scheme, we store other information to aid in making a correct prediction of the next compressed address that is read by this load instruction.

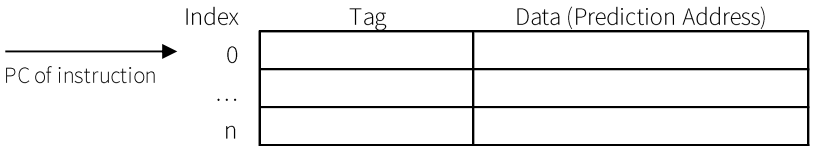

Figure 3.2 – Prefetch Table Structure

This architecture requires updating the behavior of the CPU in two key areas: FETCH stage and MEM stage.

3.2.1 **FETCH**

After we fetch an instruction, we want to know if we should begin reading from the L1 data cache. We check our prefetch table for an entry at the index of our program counter. If we return an address prediction from the table, then we populate another table called the decompression buffer. The power considerations for this table are shown in Table 3.3.

Table 3.3 – Prefetch Table Power Events

Prefetch Table Event	Power Impact
Read Hit	Tag Read, Data Read, Decompression Buffer Tag / Data Write
Read Miss	Tag Read

If we are using one of the Two-Level prefetching schemes, we will require a second table access. This table is referred to as the Pattern History Table (PHT). In this case, if the request hits the prefetch table, a pattern is returned. We then read the PHT at the pattern index, and return a reference to a value that is stored in the prefetch table. This value is the prediction address. The power considerations for the prefetch table change as well, as we only read the table data if the PHT hits over the threshold.

Table 3.4 – Two-Level Table Power Events

Two-Level Table Event	Power Impact
Prefetch Read Hit	Prefetch Tag Read
Prefetch Read Miss	Prefetch Tag Read
PHT Read Hit	PHT Read, Prefetch Data Read Decompression Buffer Tag / Data Write
PHT Read Miss	PHT Read

The decompression buffer contains the complete decompressed 64-byte cache lines. It is implemented as a FIFO cache. This buffer should be large enough that data is not being evicted before it is required in the MEM stage. However, as the table gets larger, the power consumption and access time rise. Therefore, we need to determine the best value for this table experimentally.

3.2.2 **MEM**

In the MEM stage, for a load instruction, we will now know the actual address of the data to be read from the cache. At this point in the new architecture, we read the decompression buffer to see if our data exists there, decompressed. We will use the data if the PC and address of data in the buffer match the instruction that is now in the MEM stage. If the data is there, we can read it as fast as the access time for the table. The access time and power of the table depend on the size of the table.

Table 3.5 – Decompression Buffer Power Events

Decompression Buffer Event	Power Impact
Read Hit	Buffer Tag Read, Buffer Data Read
Read Miss	Buffer Tag Read

If we are using a stride or two-level prefetcher, we use this opportunity to update the stride and stride state or the pattern history of the entry in the prefetch table.

If we must access the cache directly in the MEM stage, this is where we add entries into our prefetch table. However, we only do this if we are on the critical path. For example, we do not update our prefetch table if we are decompressing into the decompression buffer.

3.3 Hardware Design

Because read latency is such an important aspect of cache memory, especially in L1 cache, this compression scheme must be implemented at the architectural level (rather than at the software / compiler level). Therefore, it requires additional hardware to implement compression / decompression. In [4], the authors provide a high-level concept of the compression and decompression schemes. However, no design is presented or evaluated. It is important to verify that the new hardware required for this proposed architecture does not have power requirements that exceed the benefit of the architecture itself. Furthermore, it is important to define the delay requirements for decompression, as this has a direct impact on the performance of the CPU in the proposed architecture. For these reasons, we designed 64-byte compressor and decompressor units in Verilog to confirm the power consumption penalty as well as the hardware delay.

Compressor

The compressor unit contains separate hardware to evaluate the cache line for each type of compression scheme in parallel. This method prioritizes speed over resource usage. Because much of the hardware required to evaluate the different compression schemes is the same (largely based on adders / subtractors), a more resource-optimized approach would be to evaluate each method serially using the same hardware. In the future, it would be interesting to evaluate this approach for compression, as this task does not fall on the critical execution path. In the current design, we evaluate each compression scheme in parallel with the design shown in Figure 3.3.

To perform compression, the 64-byte cache line is divided into 2, 4, or 8-byte segments. The first segment is chosen as the **base**. Then, this base is subtracted from each of the remaining segments. The result of this subtraction is the array of **deltas**. A delta is stored as either a 1, 2, or 4-byte value, depending on the compression scheme being used. If all deltas can be stored without overflow, then the compression is valid.

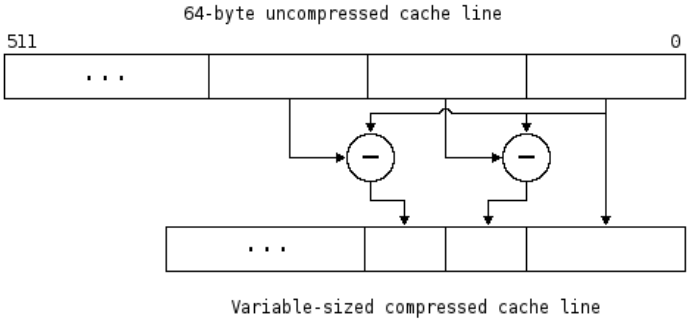

Figure 3.3 – Compressor Design

Decompressor

The decompressor unit follows the same design, except the subtraction operation is replaced by simple addition. Unlike the compressor, it is important that we prioritize speed over resource usage in the decompressor because our intent is to minimize the decompression latency. Figure 3.4 shows the design.

To perform decompression, the compressed cache line is divided into segments depending on the encoding of the data. The first 2, 4, or 8 bytes is the base. The base is carried to the decompressed line as-is. The remaining bytes are divided into 1, 2, or 4-byte deltas. These deltas are added to the base to create the decompressed segment. As a redundancy, the first delta is always zero (representing the delta of the first segment which is the base).

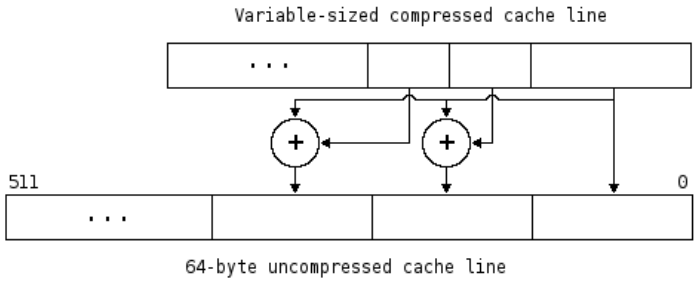

Figure 3.4 – Decompressor Design

The basis of the compressor and decompressor designs used for this project are 64-bit, 32-bit, and 16-bit adders. Compression requires a subtraction operation between 8, 4, and 2-byte blocks within a single cache line depending on the compression scheme. Decompression works in the opposite manner. Addition of 8, 4, and 2-byte "bases" with 1, 2, and 4-byte "deltas" restores the data to an uncompressed state. Large adders are discussed in depth in [16] and, as with the overall design approach, provide the opportunity to prioritize

speed over resource allocation. Ultimately, we selected the hierarchical carry-lookahead adder as the basis of the design due to its balance between speed and resource usage.

3.3.1 **Hierarchical Carry-Lookahead Adder**

The primary goal of this work is to avoid the latency of decompression on the critical execution path by using prefetching to perform decompression in parallel. However, when prefetching fails (i.e. compulsory misses during start-up, or when the predicted load address is incorrect), the processor will see the full penalty of decompression. Therefore, it is important to minimize this delay as much as possible. The delay of the decompressor depends on the design of the adders used in the new hardware.

Simple adders implement a "full adder" block for each bit and propagate carry bits serially through the circuit. While these circuits use a small number of gates, and therefore consume less power, they are very slow. Each bit requires the previous bits to be evaluated first causing many gate delays.

Alternatively, we can consider a full 64-bit carry lookahead adder. Because none of the stages execute serially, this is one of the fastest adders we can implement here. However, because each bit requires the same information as all previous bits, the complexity and size of this hardware would become excessive.

Nesbit and Smith describe a hierarchical carry-lookahead adder that divides the carry-lookahead function into 8-bit blocks, which are each evaluated serially by propagating the carry bit through the circuit [15]. This approach is a trade-off between good speed and moderate resource usage. To describe this adder, we must look at the definition for the full adder. The truth table of the full adder is shown in Figure 3.5 and Karnaugh map in Figure 3.6.

c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 3.5 – Truth Table for Full Adder

c_i \ x_iy_i	00	01	11	10
0			1	
1		1	1	1

Figure 3.6 – Karnaugh Map for Full Adder

From the truth table and k-map, one can see that the carryout bit for a given stage can be determined as:

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i \tag{3.2}$$

And the sum bit is the XOR of the three input signals:

$$s_i = x_i \overline{y}_i \overline{c}_i + \overline{x}_i y_i \overline{c}_i + \overline{x}_i \overline{y}_i c_i + x_i y_i c_i \tag{3.3}$$

Factoring the carry-in provides:

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i \tag{3.4}$$

From this equation, two important functions are defined, *generate* and *propagate*. The *generate* function is defined as:

$$g_i = x_i y_i \tag{3.5}$$

The *propagate* function is defined as:

$$p_i = x_i + y_i \tag{3.6}$$

Leaving the relationship between the carryout and these functions as being:

$$c_{i+1} = g_i + p_i c_i \tag{3.7}$$

So, let's look at the carryout of our first 8-bit block, c_8:

$$c_8 = g_7 + p_7 c_7 \tag{3.8}$$

Expanding this formula provides:

$$\begin{aligned} c_8 = {} & g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 + p_7 p_6 p_5 p_4 p_3 g_2 \\ & + p_7 p_6 p_5 p_4 p_3 p_2 g_1 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 c_0 \end{aligned} \tag{3.9}$$

From this expanded view, we can define the *generate* and *propagate* signals for the entire block:

$$\begin{aligned} G_0 = {} & g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 + p_7 p_6 p_5 p_4 p_3 g_2 \\ & + p_7 p_6 p_5 p_4 p_3 p_2 g_1 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0 \end{aligned} \tag{3.10}$$

And,

$$P_0 = p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 \tag{3.11}$$

Which results in,

$$c_8 = G_0 + P_0 c_0 \tag{3.12}$$

Later stages are calculated in the same way:

$$\begin{aligned} c_{16} &= G_1 + P_1 c_8 \\ &= G_1 + P_1 G_0 + P_1 P_0 c_0 \end{aligned} \tag{3.13}$$

In the modules that handle 2-byte bases, 16-bit adder/subtractors are used. In the modules that handle 4-byte bases, 32-bit adder/subtractors are used. Finally, in the modules that handle 8-byte bases, 64-bit adder/subtractors are used. Figure 3.7 shows the generic design of a 16-bit adder using smaller 8-bit lookahead adders.

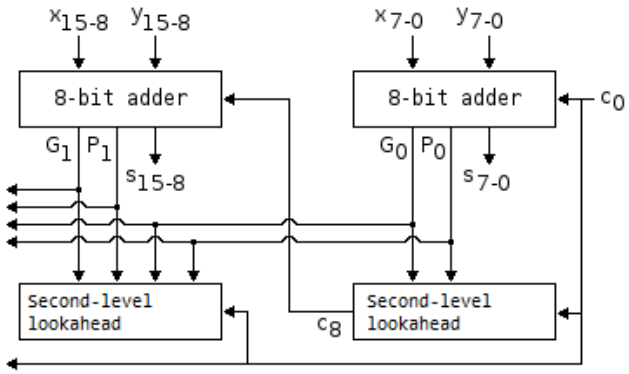

Figure 3.7 – Adder Design

3.3.2 **Implementation in Verilog**

The above derivations are the basis of the compressor and decompressor designs in Verilog. Source files for these designs are included in Appendix A. The general structures of the designs are highlighted here to show the modularity of the designs and the significance of the adders.

Compressor

The structure of the compressor design in Verilog, including the test bench used to verify the design, is as follows:

```
testbench (compressor_testbench.v)
  compressor (compressor.v)
    bdi (bdi.v)
      hadder8 (hadder8.v)
    bdi32 (bdi32.v)
      hadder8 (hadder8.v)
    bdi16 (bdi16.v)
      hadder8 (hadder8.v)
```

Figure 3.8 – HDL Structure of Compressor

The module **hadder8**, implements the 8-bit adder block from a hierarchical carry-lookahead adder. That is, it outputs the block generate (G_i) and block propagate (P_i) functions rather than the carryout (c_{i+1}) as does a typical ripple-carry adder.

The next module up implements as many of these 8-bit blocks as are necessary to perform the subtraction function. These modules are also responsible for inverting the input to turn **hadder8** into a subtractor.

- **bdi** implements a 64-bit adder, so 8 instances of **hadder8** for "base 8" compression
- **bdi32** implements a 32-bit adder, so 4 instances of **hadder8** for "base 4" compression
- **bdi16** implements a 16-bit adder, so 2 instances of **hadder8** for "base 2" compression

These modules evaluate all delta sizes in parallel. For example, **bdi** outputs three valid bits: one for base 8 delta 4, one for base 8 delta 2, and one for base 8 delta 1.

The top module, **compressor**, is responsible for instantiating blocks of **bdi**, **bdi32**, and **bdi16** on the cache line to check for all three base sizes in parallel.

- 8 instances of **bdi** for "base 8" compression on a 512-bit cache line
- 16 instances of **bdi32** for "base 4" compression on a 512-bit cache line
- 32 instances of **bdi16** for "base 2" compression on a 512-bit cache line

Module **compressor** then takes all valid bits and determines which compression scheme will be used, if any.

Testbench Strategy

To test the functionality of the compressor, testing was performed using Xilinx ISE WebPACK [17]. Input stimulus to the compressor module is the 512-bit uncompressed cache line. Test points were chosen as the boundary conditions for each of the six base/delta combinations as well as a simple zeros and repeating values lines. A similar approach was taken to first test the Base-Delta-Immediate model in SimpleScalar. Test cases are shown in Table 3.6 and the compressor output is shown functioning in Figure 3.9.

Table 3.6 – Compressor Test Cases

Test Case	Base (S_0=S_3=…=S_n)	Delta (S_1=S_2)	Expected result
Zeros	0 (0x0…0)	0 (0x0…0)	Zeros pass.
Repeating Values	-1 (0xF…F)	-1 (0xF…F)	Repeating values pass.
Base 8 Delta 1 Lower Fail	0 (0x0…0)	-129 (0xFFFFFFFFFFFFFF7F)	B8D1 fail. B8D2 pass.
Base 8 Delta 1 Lower Pass	0 (0x0…0)	-128 (0xFFFFFFFFFFFFFF80)	B8D1 pass.
Base 8 Delta 1 Upper Fail	0 (0x0…0)	128 (0x0000000000000080)	B8D1 fail. B8D2 pass.
Base 8 Delta 1 Upper Pass	0 (0x0…0)	127 (0x000000000000007F)	B8D1 pass.
Base 8 Delta 2 Lower Fail	0 (0x0…0)	-32,769 (0xFFFFFFFFFFFF7FFF)	B8D2 fail. B8D4 pass.
Base 8 Delta 2 Lower Pass	0 (0x0…0)	-32,768 (0xFFFFFFFFFFFF8000)	B8D2 pass.
Base 8 Delta 2 Upper Fail	0 (0x0…0)	32,768 (0x0000000000008000)	B8D2 fail. B8D4 pass.
Base 8 Delta 2 Upper Pass	0 (0x0…0)	32,767 (0x0000000000007FFF)	B8D2 pass.
Base 8 Delta 4 Lower Fail	0 (0x0…0)	-2,147,483,649 (0xFFFFFFFF7FFFFFFF)	Not compressible.
Base 8 Delta 4 Lower Pass	0 (0x0…0)	-2,147,483,648 (0xFFFFFFFF80000000)	B8D4 pass.
Base 8 Delta 4 Upper Fail	0 (0x0…0)	2,147,483,648 (0x0000000080000000)	Not compressible.
Base 8 Delta 4 Upper Pass	0 (0x0…0)	2,147,483,647 (0x000000007FFFFFFF)	B8D4 pass.
Base 4 Delta 1 Lower Fail	0 (0x0…0)	-129 (0xFFFFFF7F)	B4D1 fail. B4D2 pass.
Base 4 Delta 1 Lower Pass	0 (0x0…0)	-128 (0xFFFFFF80)	B4D1 pass.
Base 4 Delta 1 Upper Fail	0 (0x0…0)	128 (0x00000080)	B4D1 fail. B4D2 pass.
Base 4 Delta 1 Upper Pass	0 (0x0…0)	127 (0x0000007F)	B4D1 pass.
Base 4 Delta 2 Lower Fail	0 (0x0…0)	-32,769 (0xFFFF7FFF)	Not compressible.
Base 4 Delta 2 Lower Pass	0 (0x0…0)	-32,768 (0xFFFF8000)	B4D2 pass.
Base 4 Delta 2 Upper Fail	0 (0x0…0)	32,768 (0x00008000)	Not compressible.
Base 4 Delta 2 Upper Pass	0 (0x0…0)	32,767 (0x00007FFF)	B4D2 pass.
Base 2 Delta 1 Lower Fail	0 (0x0…0)	-129 (0xFF7F)	Not compressible.
Base 2 Delta 1 Lower Pass	0 (0x0…0)	-128 (0xFF80)	B2D1 pass.
Base 2 Delta 1 Upper Fail	0 (0x0…0)	128 (0x0080)	Not compressible.
Base 2 Delta 1 Upper Pass	0 (0x0…0)	127 (0x007F)	B2D1 pass.

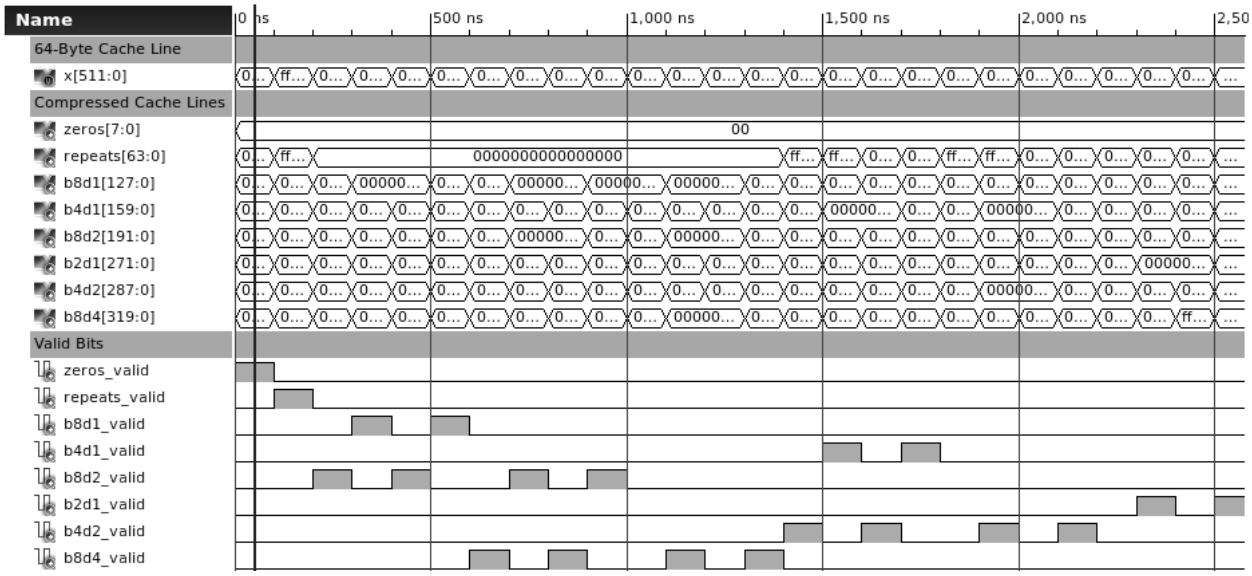

Figure 3.9 – Testbench Waveforms for Compressor in Xilinx ISE

Decompressor

The structure of the decompressor design in Verilog, including the test bench used to verify the design, is as follows:

```
testbench (decompressor_testbench.v)
  decompressor (decompressor.v)
    hadd (hadd.v)
      hadder8 (hadder8.v)
    hadd32 (hadd32.v)
      hadder8 (hadder8.v)
    hadd16 (hadd16.v)
      hadder8 (hadder8.v)
```

Figure 3.10 – HDL Structure of Decompressor

The **hadder8** module is identical to that of the compressor. The key differences between the decompressor and compressor are that the second level modules (**hadd**, **hadd32**, and **hadd16**) do not convert **hadder8** into a subtractor and they do not have to evaluate delta overflow. These modules strictly build the 64-bit, 32-bit, and 16-bit hierarchical carry-lookahead adders.

- **hadd** implements a 64-bit adder, so 8 instances of **hadder8** for "base 8" decompression
- **hadd32** implements a 32-bit adder, so 4 instances of **hadder8** for "base 4" decompression
- **hadd16** implements a 16-bit adder, so 2 instances of **hadder8** for "base 2" decompression

The top module, **decompressor**, has much more work to do than that of the compressor. This module must instantiate adders for each compression scheme, not just for each base.

- 8 instances of **hadd** for “base 8 delta 1” decompression on a 128-bit cache line
- 8 instances of **hadd** for “base 8 delta 2” decompression on a 192-bit cache line
- 8 instances of **hadd** for “base 8 delta 4” decompression on a 320-bit cache line
- 16 instances of **hadd32** for “base 4 delta 1” decompression on a 160-byte cache line
- 16 instances of **hadd32** for “base 4 delta 2” decompression on a 288-byte cache line
- 32 instances of **hadd16** for “base 2 delta 1” decompression on a 272-byte cache line

With all these instances, module **decompressor** attempts decompress an input cache line using all 8 methods at once and even outputs a 512-bit decompressed cache line for each. Only the line with an associated valid bit contains the correct data. Module **decompressor** sets this valid bit based on the input encoding bits.

Chapter 4

Simulation Methodology

In this chapter, we discuss the method for evaluating the performance of the new compression and prefetching architecture. The tools required to perform this analysis are discussed as well as the environment used to perform testing.

4.1 Methodology

In this section, we describe four key tools used in performing this work: SimPoint, CACTI, SimpleScalar, and Wattch.

SimPoint [18] is used to determine the intervals that can be executed to represent the full execution of a given program. We use SimPoint as a means of reducing the simulation time and size of outputs from the simulator without sacrificing the behavior of the benchmarks used. The decided simulation points are tabulated and the percent error of each is determined based on a comparison of CPI between the weighted simulation points and the full execution of the benchmark.

CACTI [19] is used to generate the static and dynamic power models for the various cache configurations used for this thesis. In addition, we use CACTI to model the prefetch tables and the new decompression buffer that is required for the proposed architecture. Configurations and power results are presented as they are used as inputs into the simulator.

SimpleScalar (specifically a branch called Wattch), and the changes introduced in this work, are used to model the behaviour of the cache compression and prefetching architectures.

A summary of the simulation approach is shown in Figure 4.1 and discussed in detail in the following sections.

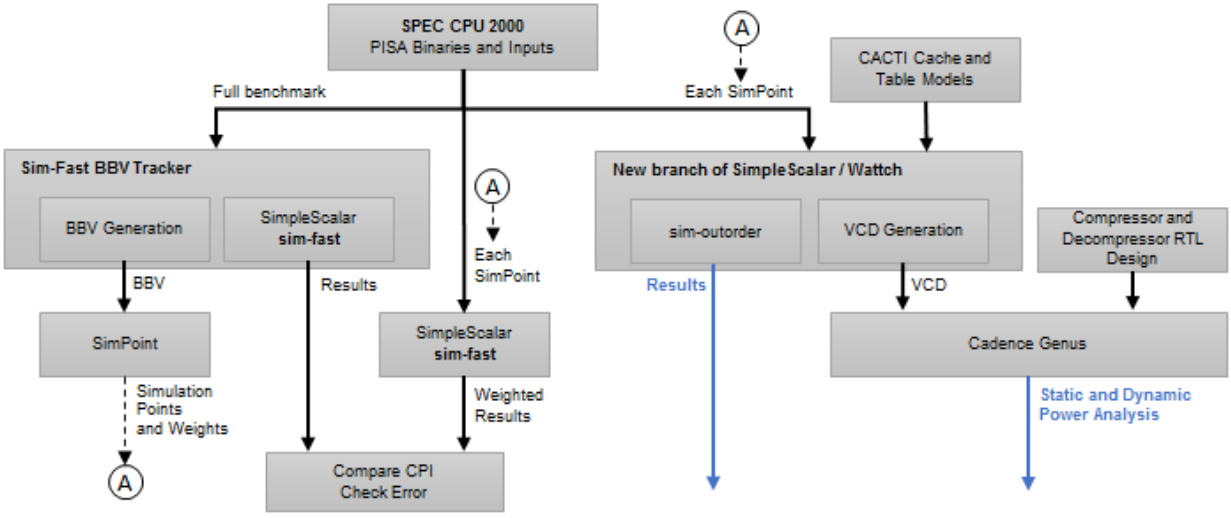

Figure 4.1 – Simulation Flow Diagram

4.1.1 **Simpoint**

For this thesis, parts of the SPEC CPU 2000 benchmark suite are used. Full runs of these benchmarks can take days to run even in a simple performance simulator (e.g. sim-fast). Running these in a detailed simulator such as sim-outorder, and especially in the modified version that we have developed, can take much longer. Therefore, it was necessary to identify smaller intervals of these benchmarks that could be executed. SimPoint is a tool that was created to choose simulation intervals that best represent the full program execution. SimPoint does this in four steps: Basic Block Vector (BBV) Analysis, Random Projection, Phase Classification, and Simulation Point Selection [18].

BBV Analysis

The Basic Block Vector (BBV) contains information about the behaviour of the program with respect to basic blocks. A basic block is a section of the program with one entry point and one exit point that executes from start to finish. The BBV itself is an array of elements representing the frequency each basic block is entered for a given execution interval (weighted by the number of instructions in that block).

For this thesis, BBV information for the SPEC CPU 2000 benchmarks is created using the tool **Sim-Fast BBV Tracker**. This tool is provided by the creators of SimPoint and is a modified version of SimpleScalar that generates the BBV files during execution of **sim-fast**.

Random Projection, Phase Classification, and Simulation Point Selection

SimPoint analyses the BBV file generated by the previous step and chooses a representation of each phase by finding the interval closest to the centre of the phase. Then, SimPoint determines the weight of that simulation point based on the number of intervals in that phase of the program's execution.

SimPoint Results

For this work, an interval of 100 million instructions is chosen for determining simulation points. The selection of 100 million instruction intervals is a balance between 1 billion, which generates very large output data, and 10 million, which is too small to run without performing a "warmup" routine. The authors state that 100 million is an appropriately sized interval to avoid the need to bring the simulations to a "warmup" state [16]. The maximum number of clusters in the k-means algorithm [20] executed in SimPoint was chosen based on the error produced by the resultant simulation points.

Choosing a single cluster would result in the simplest implementation. That is, no weighing or combination of results would be necessary. This method, however, does not yield good results, as most programs will contain multiple phases. The authors use the percent error in CPI between the full execution and the weighted simulation points as a means of evaluating the accuracy of the method. Rather than arbitrarily choosing a maximum number of clusters, we use this same method to evaluate the error as the authors do in [21].

Given a set of simulation points and weights, the following is the correct method of calculating the weighted CPI [22],

$$CPI = Weight_1 CPI_1 + Weight_2 CPI_2 + ... + Weight_n CPI_n \quad (4.1)$$

Using 164.gzip as an example, for M = 3:

Table 4.1 – 164.gzip CPI Values by Simulation Point

Simulation Point	Weight	CPI
6	0.296296	0.5884
15	0.222222	0.5289
25	0.481481	0.5790

$$CPI = (0.296296)(0.5884) + (0.222222)(0.5289) + (0.481481)(0.5790) \quad (4.2)$$
$$= 0.5707$$

Comparing this with the CPI result from the full execution, we can calculate the % error:

$$\% \; Error = \frac{\left|CPI_{Full} - CPI_{Simpoint}\right|}{CPI_{Full}}(100) \quad (4.3)$$
$$= \frac{|0.5728 - 0.5707|}{0.5728}(100)$$
$$= \frac{0.0021}{0.5728}(100)$$
$$= 0.37\%$$

Table 4.2 contains this error calculation for 18 of the SPEC CPU 2000 benchmarks for maximum number of clusters (M) from 1 to 3.

Table 4.2 – Simpoint Error by Maximum Number of Clusters

Benchmark	M=1	M=2	M=3
164.gzip	0.68%	0.19%	0.37%
168.wupwise	2.69%	2.79%	0.45%
171.swim	**49.28%**	8.16%	0.02%
172.mgrid	2.43%	0.57%	0.10%
173.applu	7.71%	5.72%	0.93%
175.vpr	2.84%	1.56%	3.07%
176.gcc	3.56%	6.62%	2.53%
177.mesa	0.29%	0.07%	0.03%
179.art	1.65%	0.09%	0.05%
181.mcf	**23.20%**	2.12%	4.39%
183.equake	1.48%	0.14%	0.48%
188.ammp	4.85%	0.00%	1.77%
197.parser	6.84%	**14.20%**	5.14%
253.perlbmk	0.15%	0.04%	0.29%
255.vortex	5.69%	4.44%	1.48%
256.bzip2	**17.60%**	**13.80%**	**14.64%**
300.twolf	4.28%	0.69%	0.00%
301.apsi	**11.60%**	**11.64%**	**12.25%**

As can be seen from the data, the amount of error is significant in the benchmarks 171.swim, 181.mcf, 197.parser, 256.bzip2, and 301.apsi. The authors of SimPoint evaluated the SPEC CPU 2000 benchmarks using a maximum number of clusters equal to 10 for intervals of 100 million instructions. In their data, the maximum percent error was 5.47%. So, we then evaluated the 18 benchmarks with these same parameters. The results are shown in Table 4.3.

Table 4.3 – SimPoint Error

Benchmark	Instructions (Full)	CPI (Full)	CPI (Simpoint)	% Error
164.gzip	2702173004	0.5728	0.5744	0.27%
168.wupwise	607580800644	0.6840	0.6831	0.13%
171.swim	440458734007	1.0914	1.1025	1.02%
172.mgrid	900584206345	0.5846	0.5849	0.06%
173.applu	827051421403	0.6924	0.6929	0.07%
175.vpr	86587310713	0.9186	0.9293	1.17%
176.gcc	84506842637	0.5707	0.5719	0.20%
177.mesa	304718816959	0.5779	0.5771	0.14%
179.art	10917697312	1.4175	1.4189	0.10%
181.mcf	49073257000	2.4408	2.5668	5.16%
183.equake	175021725999	1.0339	1.0333	0.05%
188.ammp	350015586932	0.9809	0.9834	0.25%
197.parser	9628364671	1.0034	1.0550	5.14%
253.perlbmk	1389497618	0.8167	0.8190	0.29%
255.vortex	114074663283	0.5291	0.5313	0.41%
256.bzip2	113183466499	0.5165	0.5208	0.84%
300.twolf	1394388332	0.8594	0.8586	0.09%
301.apsi	816009733414	0.6981	0.6820	2.31%

The maximum percent error from this method is 5.16% which is less than that of the 5.47% in the author's results, but quite similar. Therefore, for the purposes of this thesis, all benchmarks are executed for a maximum of 10 intervals of 100 million instructions as generated by SimPoint in the method discussed above. The resulting simulation points and their weights are shown in Table 4.4.

Table 4.4 – 100M SimPoint Results

Benchmark	Simulation Point	Weight
164.gzip	6	0.296296
164.gzip	14	0.111111
164.gzip	15	0.185185
164.gzip	23	0.407407
168.wupwise	59	0.00510288
168.wupwise	100	0.0454321
168.wupwise	389	0.510288
168.wupwise	825	0.38963
168.wupwise	3487	0.0138272
168.wupwise	5418	0.0357202
171.swim	1201	0.0817439
171.swim	1362	0.0610808
171.swim	2397	0.157584
171.swim	2729	0.0569936
171.swim	3008	0.156222
171.swim	3083	0.136921
171.swim	3125	0.236149
171.swim	4016	0.0560854
171.swim	4072	0.0572207
172.mgrid	1093	0.0715158
172.mgrid	2407	0.388895
172.mgrid	4844	0.0599667
172.mgrid	6231	0.179789
172.mgrid	6309	0.0896169
172.mgrid	6347	0.0579678
172.mgrid	7271	0.118712
172.mgrid	8395	0.0335369
173.applu	139	0.272551
173.applu	942	0.0962515
173.applu	1076	0.1052
173.applu	1872	0.0518742

Benchmark	Simulation Point	Weight
173.applu	3690	0.0613059
173.applu	3833	0.0669891
173.applu	6512	0.273398
173.applu	6542	0.0322854
173.applu	7986	0.0401451
175.vpr	2	0.0254335
175.vpr	354	0.439306
175.vpr	456	0.323699
175.vpr	582	0.211561
176.gcc	3	0.392899
176.gcc	22	0.0461538
176.gcc	214	0.0485207
176.gcc	229	0.0639053
176.gcc	305	0.127811
176.gcc	321	0.0556213
176.gcc	561	0.126627
176.gcc	694	0.138462
177.mesa	252	0.0203479
177.mesa	1271	0.13423
177.mesa	1276	0.225796
177.mesa	1417	0.328848
177.mesa	1845	0.0994421
177.mesa	2962	0.185756
177.mesa	3034	0.00557926
179.art	0	0.00917431
179.art	13	0.155963
179.art	40	0.247706
179.art	45	0.00917431
179.art	47	0.577982
181.mcf	17	0.0938776
181.mcf	169	0.281633

Table 4.4 – 100M SimPoint Results (continued)

Benchmark	Simulation Point	Weight
181.mcf	200	0.126531
181.mcf	247	0.0836735
181.mcf	277	0.328571
181.mcf	350	0.044898
181.mcf	378	0.0408163
183.equake	15	0.0142857
183.equake	60	0.0782857
183.equake	147	0.0794286
183.equake	931	0.204571
183.equake	961	0.202857
183.equake	1210	0.217714
183.equake	1551	0.202857
188.ammp	14	0.00942857
188.ammp	271	0.128571
188.ammp	568	0.195143
188.ammp	661	0.132857
188.ammp	1822	0.0148571
188.ammp	1896	0.130286
188.ammp	1970	0.0865714
188.ammp	2171	0.0611429
188.ammp	2251	0.0114286
188.ammp	2912	0.229714
197.parser	27	0.15625
197.parser	43	0.604167
197.parser	66	0.239583
253.perlbmk	0	0.0769231
253.perlbmk	1	0.230769
253.perlbmk	10	0.692308
255.vortex	55	0.0464912
255.vortex	104	0.0596491

Benchmark	Simulation Point	Weight
255.vortex	159	0.0710526
255.vortex	359	0.134211
255.vortex	387	0.148246
255.vortex	510	0.00877193
255.vortex	526	0.0763158
255.vortex	710	0.455263
256.bzip2	9	0.10168
256.bzip2	52	0.102564
256.bzip2	94	0.129973
256.bzip2	212	0.116711
256.bzip2	254	0.161804
256.bzip2	272	0.0565871
256.bzip2	486	0.0742706
256.bzip2	497	0.114943
256.bzip2	539	0.0884173
256.bzip2	587	0.0530504
300.twolf	0	0.0769231
300.twolf	1	0.0769231
300.twolf	2	0.0769231
300.twolf	4	0.307692
300.twolf	10	0.461538
301.apsi	167	0.101471
301.apsi	653	0.603554
301.apsi	2083	0.0463235
301.apsi	2453	0.0253676
301.apsi	2865	0.135294
301.apsi	2923	0.0205882
301.apsi	5422	0.0101716
301.apsi	5428	0.0448529
301.apsi	5986	0.0123775

4.1.2 **CACTI**

CACTI is a cache and memory access time, cycle time, area, leakage power, and dynamic energy modelling tool [23]. For the purposes of this thesis, the access time, leakage power, and dynamic energy calculations performed by CACTI are the focus. For specific cache configurations, the access time, leakage power, and dynamic energy parameters are determined and used as input into the simulator.

CACTI 6.5 was built from source and used for this thesis. One modification is made to CACTI to output the dynamic energy (tag, data, and total) for the write operation. The details of this change, building, and using CACTI are not included in this report.

CACTI Results

From the output of CACTI, the following lines are particularly relevant for this thesis and are used as input into the simulator:

```
Access time (ns): ...

Data array: Total dynamic read energy/access (nJ): ...
Data array: Total dynamic write energy/access (nJ): ...
Total leakage read/write power of a bank (mW): ...

Tag array:  Total dynamic read energy/access (nJ): ...
Tag array:  Total dynamic write energy/access (nJ): ...
Total leakage read/write power of a bank (mW): ...
```

Figure 4.2 – CACTI Output

Table 4.5 shows all the L1 cache configurations used for this thesis. The first configuration in the table represents the baseline scheme with no compression. The next two represents a compressed cache of half the size of the baseline. The tag and data banks for the compressed scheme are modelled in separate runs in CACTI.

Table 4.5 – CACTI L1 Cache Configurations and Power Results

Configuration	Data (bytes)	Assoc.	Tag (bits)	Data Read (nJ)	Data Write (nJ)	Data Static (mW)	Tag Read (nJ)	Tag Write (nJ)	Tag Static (mW)	Access Time (ns)	Cycles @ 3GHz
BASELINE	65536	2	17	0.254468	0.29159	25.0286	0.00642276	0.00698272	1.22089	1.65339	5 (4.96017)
COMPRESSED DATA	32768	1	17	0.149461	0.164224	13.6143	-	-	-	1.24155	4 (3.72465)
COMPRESSED TAG	65536	2	53	-	-	-	0.0113552	0.0261866	3.29114	1.80514	6 (5.41542)

Table 4.5 shows all the L2 cache configurations used for this thesis.

Table 4.6 – CACTI L2 Cache Timing

Configuration	Size (bytes)	Assoc.	Tag (bits)	Access Time (ns)	Cycles @ 3GHz
BASELINE	1048576	4	default	3.4286	11 (10.2858)

CACTI was also used to model the energy consumption of the prefetch tables. Data size is assumed to be 4 bytes per address in this model to store the target address of the load instruction.

Table 4.7 – CACTI Prefetch Table Configurations and Power Results

Configuration	Size (bytes)	Tag (bits)	Data Read (nJ)	Data Write (nJ)	Data Static (mW)	Tag Read (nJ)	Tag Write (nJ)	Tag Static (mW)	Access Time (ns)	Cycles @ 3GHz
LO 128	512	25	0.00549938	0.00612263	0.229478	0.00219443	0.00273977	0.209592	0.932003	3 (2.796009)
LO 1024	4096	22	0.0146796	0.016525	1.89212	0.00687723	0.00530333	1.20125	1.1291	4 (3.3873)
LO 2048	8192	21	0.0196109	0.0182605	3.50766	0.00937441	0.00836156	2.70726	1.28211	4 (3.84633)
STRIDE 128	512	43 (25+16+2)	0.00558349	0.00620673	0.240491	0.00320262	0.00405959	0.320843	0.924437	3 (2.773311)
STRIDE 1024	4096	40 (22+16+2)	0.0146796	0.016525	1.89212	0.0083401	0.0106468	2.31192	1.25809	4 (3.77427)
STRIDE 2048	8192	39 (21+16+2)	0.0196109	0.0182605	3.50766	0.0136775	0.0125793	4.30645	1.29731	4 (3.89193)
HYBRID S/LO 128	512	43 (25+16+2)	0.00558349	0.00620673	0.240491	0.00320262	0.00405959	0.320843	0.924437	3 (2.773311)
HYBRID S/LO 1024	4096	40 (22+16+2)	0.0146796	0.016525	1.89212	0.0083401	0.0106468	2.31192	1.25809	4 (3.77427)
HYBRID S/LO 2048	8192	39 (21+16+2)	0.0196109	0.0182605	3.50766	0.0136775	0.0125793	4.30645	1.25809	4 (3.77427)
2LEVEL 128	1024	30 (25+1+4)	0.00702388	0.00677314	0.493251	0.00243052	0.00305377	0.234909	0.959845	3 (2.879535)
2LEVEL 1024	8192	27 (22+1+4)	0.0196109	0.0182605	3.50766	0.00751555	0.00650857	1.38357	1.14894	4 (3.44682)
2LEVEL 2048	16384	26 (21+1+4)	0.0322663	0.0309884	6.95555	0.0184487	0.0115735	6.12193	1.28954	4 (3.86862)
HYBRID 2L/S 128	1024	48 (25+1+4+16+2)	0.00720647	0.00695573	0.431373	0.0034364	0.00437127	0.345173	0.94776	3 (2.84328)
HYBRID 2L/S 1024	8192	45 (22+1+4+16+2)	0.0196109	0.0182605	3.50766	0.00968252	0.0124506	2.72231	1.27512	4 (3.82536)
HYBRID 2L/S 2048	16384	44 (21+1+4+16+2)	0.0322663	0.0309884	6.95555	0.0244858	0.0227288	9.34134	1.60081	5 (4.80243)

For two-level prefetching, we require a second table called the Pattern History Table (PHT). This table is indexed by the access pattern and stores an integer value for each of the data values stored in the prefetch table.

Table 4.8 – CACTI Pattern History Table Power Results

Configuration	Size (bytes)	Tag (bits)	Data Read (nJ)	Data Write (nJ)	Data Static (mW)	Tag Read (nJ)	Tag Write (nJ)	Tag Static (mW)	Access Time (ns)	Cycles @ 3GHz
PHT 2D4P	64	4	0.0029709	0.00305794	0.0334625	0.00390386	0.000401266	0.0120452	0.575184	2 (1.725552)

Lastly, a decompression buffer is considered with 64-byte data and 1K sets. In CACTI, this buffer is modeled as L1 cache.

Table 4.9 – CACTI Decompression Buffer Power Results

Configuration	Size (bytes)	Tag (bits)	Data Read (nJ)	Data Write (nJ)	Data Static (mW)	Tag Read (nJ)	Tag Write (nJ)	Tag Static (mW)	Access Time (ns)	Cycles @ 3GHz
BUFFER 1K	1024	64 (32 + 32)	0.00702388	0.00677314	0.493251	0.00186356	0.00203764	0.0708125	0.835837	3 (2.507511)

4.1.3 **SimpleScalar**

To be able to measure the benefit of implementing cache compression with a prefetching mechanism, we use SimpleScalar to model the performance of the CPU.

SimpleScalar is an open-source processor modelling tool that is meant to be built upon for specific applications such as this work. SimpleScalar is written in C. The tool can emulate different instruction sets, including Alpha, ARM, x86, but most importantly PISA [24]. The binaries for the SPEC CPU 2000 benchmarks used for this work are compiled to PISA.

Wattch is a specific branch of SimpleScalar for analyzing and optimizing power consumption in the architecture of a CPU [5]. Wattch provides us with a mechanism to compare our power consumption in the cache and new hardware with the overall power consumption of the CPU.

4.1.3.1 Compression

To confirm the feasibility of this work, we check how many cache lines within the 18 benchmarks are compressible using the Base-Delta-Immediate compression scheme. To do this, we model Base-Delta-Immediate in SimpleScalar.

In this compression scheme, cache lines are compressed before they are written to the cache. Cache lines are written when they miss the cache or on a write hit. Therefore, we must add functionality to the simulator when we these events occur, as mentioned previously in Table 3.1.

Zeros

The check for zeros compressibility is straightforward. We iterate through all elements of the cache line array and flag zeros compressibility as not possible if any element does not equal zero.

Repeating Values

In the scheme proposed by the authors in [4], repeating 8-byte values are considered. Therefore, we check compressibility for this while checking for other "base 8" schemes. Starting at element zero, we concatenate the values of the next seven bytes to the current byte, then iterate through the cache line array by a stride of 8. At each iteration through the array, we check if the new 8-byte value equals the 8-byte value at element 0. If any element does not equal element zero, we flag repeating values compressibility as not possible.

Base-Delta-Immediate

Separate arrays and separate loops handle the compressibility check for each size of base. Base 8 behaves as described above. Base 4 iterates though the array in stride of 4, Base 2 in strides of 2.

To verify the compressibility of a Base-Delta-Immediate scheme, we check that each delta does not overflow its datatype referenced either from the base or from zero (immediate). If the second option is taken (immediate), then the immediate flag is set for that iteration. Table 4.10 shows the overflow parameters of each delta.

Table 4.10 – Delta Datatype and Overflow Information

Delta	Data Type	Floor	Ceiling
1	signed char	-128	127
2	signed short	-32768	32767
4	sighed int	-2147483648	2147483647

Validation of the Compression Model

To confirm that we have correctly modeled Base-Delta-Immediate compression in SimpleScalar, we write a program to exercise the boundary condition of each of the compression schemes, cross-compile that program to the PISA instruction set, and run this program through our model and verify the results.

This program consists of 26 arrays containing 64 1-byte elements. Those arrays contain the cache line values that exercise the boundaries of the model. Table 4.11 shows these values.

Table 4.11 – Boundary Conditions for Compression

array	description	significant value
char z[64]	Zeros	0 (0x0…0)
char r[64]	Repeating Values	-1 (0xF…F)
char b8d1lf[64]	Base 8 Delta 1 Lower Fail	-129 (0xFFFFFFFFFFFFFF7F)
char b8d1lp[64]	Base 8 Delta 1 Lower Pass	-128 (0xFFFFFFFFFFFFFF80)
char b8d1uf[64]	Base 8 Delta 1 Upper Fail	128 (0x0000000000000080)
char b8d1up[64]	Base 8 Delta 1 Upper Pass	127 (0x000000000000007F)
char b8d2lf[64]	Base 8 Delta 2 Lower Fail	-32,769 (0xFFFFFFFFFFFF7FFF)
char b8d2lp[64]	Base 8 Delta 2 Lower Pass	-32,768 (0xFFFFFFFFFFFF8000)
char b8d2uf[64]	Base 8 Delta 2 Upper Fail	32,768 (0x0000000000008000)
char b8d2up[64]	Base 8 Delta 2 Upper Pass	32,767 (0x0000000000007FFF)
char b8d4lf[64]	Base 8 Delta 4 Lower Fail	-2,147,483,649 (0xFFFFFFFF7FFFFFFF)
char b8d4lp[64]	Base 8 Delta 4 Lower Pass	-2,147,483,648 (0xFFFFFFFF80000000)
char b8d4uf[64]	Base 8 Delta 4 Upper Fail	2,147,483,648 (0x0000000080000000)
char b8d4up[64]	Base 8 Delta 4 Upper Pass	2,147,483,647 (0x000000007FFFFFFF)
char b4d1lf[64]	Base 4 Delta 1 Lower Fail	-129 (0xFFFFFF7F)
char b4d1lp[64]	Base 4 Delta 1 Lower Pass	-128 (0xFFFFFF80)
char b4d1uf[64]	Base 4 Delta 1 Upper Fail	128 (0x00000080)
char b4d1up[64]	Base 4 Delta 1 Upper Pass	127 (0x0000007F)
char b4d2lf[64]	Base 4 Delta 2 Lower Fail	-32,769 (0xFFFF7FFF)
char b4d2lp[64]	Base 4 Delta 2 Lower Pass	-32,768 (0xFFFF8000)
char b4d2uf[64]	Base 4 Delta 2 Upper Fail	32,768 (0x00008000)
char b4d2up[64]	Base 4 Delta 2 Upper Pass	32,767 (0x00007FFF)
char b2d1lf[64]	Base 2 Delta 1 Lower Fail	-129 (0xFF7F)
char b2d1lp[64]	Base 2 Delta 1 Lower Pass	-128 (0xFF80)
char b2d1uf[64]	Base 2 Delta 1 Upper Fail	128 (0x0080)
char b2d1up[64]	Base 2 Delta 1 Upper Pass	127 (0x007F)

Running our benchmark through our compression model in SimpleScalar, we get the following result, which matches the expected behaviour of the compressor hardware.

```
sim_num_byte_reads      187 # total number of byte reads
sim_num_zero_blocks       1 # total number of zero block reads
sim_num_repeats_blocks    1 # total number of repeats block reads
sim_num_del81_blocks      2 # total number of base 8 delta 1 reads
sim_num_del41_blocks      2 # total number of base 4 delta 1 reads
sim_num_del82_blocks      4 # total number of base 8 delta 2 reads
sim_num_del21_blocks      2 # total number of base 2 delta 1 reads
sim_num_del42_blocks      4 # total number of base 4 delta 2 reads
sim_num_del84_blocks      4 # total number of base 8 delta 4 reads
sim_num_uncompr_blocks  167 # total number of uncompressed reads
```

Figure 4.3 – Compression Model Verification Results

4.1.3.2 Compression with Prefetching Model

To implement the compression and prefetching model, we implement two key behaviours to the simulator:

(1) Read the prefetch table during fetch of a load instruction and, if we hit the table and return a prediction address, add the address to the decompression buffer along with a ready time equal to the current cycle plus all delays that block that data. Specifically affecting the decompression buffer are the prefetch table access time, the L1 data cache access time, and the decompression latency.

(2) Read decompression buffer before accessing L1 data cache to confirm if the correct address was there. Before running the `cache_access()` function for L1 data cache, we check if we have a correct PC and address in the decompression buffer. If we do, then our prefetch function will have correctly predicted the load address. If the PC and address are not correct in the decompression buffer, then we experience the full decompression latency and update our prefetch table information.

4.1.3.3 Stage Delays

Baseline SimpleScalar and Wattch implement single cycle pipeline stages. This is not a realistic model for many processors. Therefore, we implement a mechanism to include options for additional delays in each stage. The delays used in this work are based on [25]. To implement this, a new queue is added to store instructions delayed in the pipeline. This queue is monitored at the end of each stage and submits operations each cycle as they are ready.

4.1.3.4 VCD Output

A critical part of the power analysis in this work is to compare the power consumption of the new hardware to that of the processor and the cache. For this, we use Cadence Genus. To achieve an accurate dynamic power model in Cadence Genus, we need to set the actual input characteristics of the cache. To do this, we generate what is called a Value Change Dump (VCD) stimulus for the hardware for each of the simulation points run in the simulator. The header information of the compressor VCD file is shown in Figure 4.4.

```
$date
2017-05-31 19:06:21 EDT
$end

$version
VCD version 0.1
$end

$timescale
1 ps
$end

$scope
module compressor
$end

$var
wire 512 ! x
$end

$upscope $end
$enddefinitions $end

#0

$dumpvars
b0 !
$end
```

Figure 4.4 – Compressor VCD Header

Then, following the header, for each instance of compression (each L1 data cache miss or write hit), a timestamped update is written into the VCD file. The timestamp is calculated using the frequency option for the CPU and the number of cycles:

$$t_{VCD,ps} = \frac{1000}{f_{GHz}} sim\_cycle \tag{4.4}$$

The timestamp is written to the VCD file followed by the value of the compressor input as a 512-character ASCII string. Figure 4.5 shows an example of this.

```
#34632
b00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000101000000000000000000000000000001000
00000000000000000000010001000001000000000000000000000000000000000000000000000000
00010100011000001000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000100000000000000000
00000000000000000000000000000000000000000000010000000100101010000000000000000000
00000000000000000000000000000000 !
```

Figure 4.5 – Compressor VCD ASCII Value

For the decompressor, a carry bit is initialized to zero and the encoding bits are updated every time. The decompressor header must declare different variables as shown in Figure 4.6. An example of a timestamped update to the decompressor VCD file is shown in Figure 4.7.

```
$var wire 512 ! x $end
$var wire 1 # carry $end
$var wire 4 $ encoding $end
```

Figure 4.6 – Decompressor VCD Header Variables

```
#864801
b00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000101000010101000010100000101100001010000110010000101000011101000010100001010
10000101000011101000010010001101100001001001001000000100110000100000010011000001
00000100110011111000001001100101110000100010110000000010001101000000001000110111 0
00000011100000000000000010011101 !
b0111 $
```

Figure 4.7 – Decompressor VCD ASCII Value

4.1.3.5 SimPoint Implementation

Baseline SimpleScalar and Wattch allow for fast forwarding through a benchmark and running a certain block of instructions. This is implemented through the runtime options `-fastfwd` and `-max:inst`. The first is implemented as a signed integer and the second as an unsigned integer. This means, the deepest interval that can be run for any program is from instruction 2,147,483,648 to 6,442,450,942. Looking back at the results of our SimPoint analysis in Table 4.4, we see that our largest simpoint is 8395 for benchmark 172.mgrid. This means we are required to execute from instruction 839,500,000,001 to 839,600,000,000.

To accommodate this, we implement two new options at runtime `-interval` and `-simpoint`. The first is simply a renaming of `-max:inst`. The second, `simpoint`, is the multiple of `interval` that `fastfwd` should be set to. By declaring `fastfwd` as a 64-bit unsigned integer, we can accommodate all of our simpoints.

4.1.3.6 Technology Scaling

Out-of-the-box Wattch is based on 180nm technology parameters provided in an early technical report for CACTI. In **power.h**, Wattch is set up to allow for configuration and scalability of the CMOS feature size, as well as the CPU frequency used for calculations. To accommodate our chosen frequency of 3 GHz and the 90nm CMOS used to create the hardware for this thesis, the following updates were made to **power.h**:

Macro `TECH_POINT10` contains scaling definitions to bring the 180nm parameters down to a 100nm equivalent. Scaling exists for wire capacitance, wire resistance, feature length, feature area, voltage, threshold voltage, sense voltage, and overall power scaling.

Macro `FUDGEFACTOR` is used to scale results, further beyond that of the chosen tech point, from the CACTI function `calculate_time` that is built into Wattch. `FUDGEFACTOR` is given by dividing the defined technology size by the desired value:

$$\begin{aligned} FUDGEFACTOR &= \frac{TECH_{DEFINED}}{TECH_{DESIRED}} \\ &= \frac{100nm}{90nm} \\ &= 1.1111 \end{aligned} \tag{4.5}$$

Macro `Mhz` is used to define the frequency used throughout the power calculations.

4.1.4 Environment

Across the 18 benchmarks used, with a maximum SimPoint cluster size of 10, there are 122 total simulation points to be executed per configuration. With 12 configurations, there are 1464 instances of the simulator to be executed per simulation batch.

Benchmarking is performed on Lakehead University's 240 core Linux Cluster, Wesley [26]. Jobs are queued to the cluster using Torque. The output of the batch on Wesley are 1464 simulation result reports and 2928 Value Change Dump (VCD) files.

These 4392 files are moved from Wesley to Lakehead University's CMC server. Executed via scripting in Tcl, the hardware is synthesized and mapped for the compressor in Cadence Genus, then the compressor VCD file for each simulation point is input into Genus and the associated report file is appended with the dynamic power analysis results. This process is then repeated for the decompressor.

4.2 Synthesis and Static Power Analysis

In this section, the synthesis of this design using Cadence Genus is discussed as well as timing and power results and the selection of the 90nm Cadence Generic PDK. Place and route is presented for this design using Cadence Innovus.

Initial Analysis and PDK Selection

To evaluate the speed and power consumption of the hardware, the Verilog design files are synthesized using Cadence Genus Synthesis Solution.

Two process design kits were considered for this thesis, Cadence Generic 90nm PDK and FreePDK 45nm. Although power consumption is the focus of this work, the selection of the PDK was based on the delay for the decompressor, which lies on the critical execution path. Table 4.12 shows the results of this initial analysis.

Table 4.12 – Initial Static Power Analysis of Decompressor by PDK

Library	Delay (ps)	Static Power (nW)
Cadence 90nm Generic PDK v3.3 (fast.lib)	655	900,979.421
Cadence 90nm Generic PDK v3.3 (typical.lib)	1026	408,689.271
Cadence 90nm Generic PDK v3.3 (slow.lib)	2463	408,202.239
Cadence 90nm Generic PDK v3.3 (ss.0v75.lib)	3339	225,913.359
Cadence 90nm Generic PDK v3.3 (ss.0v67.lib)	4086	162,813.881
FreePDK 45nm v1.4 (gscl45nm.lib)	971	425,171.716

As can be seen from the results, the **fast** library from the Cadence 90nm GPDK is the fastest. At 655ps, even with any overhead that has not been accounted for, it is reasonable to assume that this hardware can provide decompression within 4 cycles at 3GHz. This assumption carries into the simulator discussed in the following section of this report.

Static Power

Static Power Analysis in Cadence Genus is straightforward. The Liberty Timing File (.lib) from the Process Design Kit (PDK) defines a parameter, **cell_leakage_power**, which is static power on a per-cell basis. After synthesizing the design, we can run the gates report to determine how many instances of each cell is used in the synthesized design.

```
Gate       Instances       Area    Library
----------------------------------------
AND2X1           631   2865.623    fast

...

XNOR2X1          139   1157.300    fast
----------------------------------------
total          16366  67124.920
```

Figure 4.8 – Genus Gates Report for Compressor (Condensed)

From this, we can validate Genus' static power calculation. Table 4.13 shows this validation for the compressor, Table 4.14 for the decompressor.

Table 4.13 – Compressor Static Power Determination

gate	cell_leakage_power	instances	total static power (nW)
AND2X1	44.6239	631	28157.6809
AND4X1	42.646	138	5885.148
AND4XL	40.714	2	81.428
AO21X1	84.9143	72	6113.8296
AO22X1	71.7681	75	5382.6075
AOI211XL	44.7636	81	3625.8516
AOI21XL	35.0591	553	19387.6823
AOI221XL	43.561	65	2831.465
AOI22XL	34.6297	160	5540.752
AOI2BB1XL	56.1234	57	3199.0338
AOI31XL	34.8351	80	2786.808
AOI32XL	34.4304	19	654.1776
AOI33XL	33.8609	75	2539.5675
CLKINVX1	29.4952	1148	33860.4896
CLKXOR2X1	128.734	167	21498.578
INVXL	20.9723	1496	31374.5608
MX2X1	86.8591	5	434.2955
MXI2XL	55.5867	408	22679.3736
NAND2BXL	59.9008	207	12399.4656
NAND2XL	20.9738	3363	70534.8894
NAND3BXL	58.5736	51	2987.2536
NAND3XL	21.0253	83	1745.0999
NAND4BXL	57.8584	173	10009.5032
NAND4XL	21.0281	162	3406.5522
NOR2BXL	35.5704	353	12556.3512
NOR2XL	35.3175	1349	47643.3075
NOR3BXL	45.2739	13	588.5607
NOR3XL	45.0716	193	8698.8188
NOR4BXL	51.7506	253	13092.9018
NOR4XL	51.5363	105	5411.3115
OA21X1	57.6361	150	8645.415
OAI211XL	20.8698	191	3986.1318
OAI21XL	20.8533	3030	63185.499
OAI221XL	20.7042	4	82.8168
OAI22XL	34.8023	11	382.8253
OAI2BB1XL	29.114	486	14149.404
OAI31XL	22.019	14	308.266
OR2X1	85.4618	653	55806.5554
OR2XL	76.9368	54	4154.5872
OR3X1	113.269	7	792.883
OR4X1	140.922	16	2254.752
OR4XL	132.377	74	9795.898
XNOR2X1	142.706	139	19836.134
Total Compressor Static Power			568488.51 nW

Table 4.14 – Decompressor Static Power Determination

gate	cell leakage power	instances	total static power (nW)
ADDHXL	156.633	8	1253.064
AND2X1	44.6239	731	32620.0709
AND4X1	42.646	94	4008.724
AO21X1	84.9143	294	24964.8042
AO22X1	71.7681	97	6961.5057
AOI211XL	44.7636	305	13652.898
AOI21XL	35.0591	932	32675.0812
AOI221XL	43.561	85	3702.685
AOI222XL	42.6974	8	341.5792
AOI22XL	34.6297	129	4467.2313
AOI2BB1XL	56.1234	63	3535.7742
AOI31XL	34.8351	74	2577.7974
AOI32XL	34.4304	98	3374.1792
CLKINVX1	29.4952	2393	70582.0136
CLKXOR2X1	128.734	118	15190.612
INVXL	20.9723	175	3670.1525
MXI2XL	55.5867	1812	100723.1004
NAND2BX1	72.5604	74	5369.4696
NAND2BXL	59.9008	561	33604.3488
NAND2XL	20.9738	4576	95976.1088
NAND3BX1	70.7386	74	5234.6564
NAND3BXL	58.5736	193	11304.7048
NAND3XL	21.0253	387	8136.7911
NAND4BBXL	113.985	17	1937.745
NAND4BXL	57.8584	97	5612.2648
NAND4XL	21.0281	140	2943.934
NOR2BX1	49.8456	193	9620.2008
NOR2BXL	35.5704	474	16860.3696
NOR2XL	35.3175	1849	65302.0575
NOR3BXL	45.2739	55	2490.0645
NOR3XL	45.0716	225	10141.11
NOR4BXL	51.7506	296	15318.1776
NOR4XL	51.5363	191	9843.4333
OA21X1	57.6361	280	16138.108
OAI211XL	20.8698	573	11958.3954
OAI21XL	20.8533	4210	87792.393
OAI221XL	20.7042	148	3064.2216
OAI22XL	34.8023	15	522.0345
OAI2BB1XL	29.114	929	27046.906
OR2X1	85.4618	819	69993.2142
OR4X1	140.922	88	12401.136
TLATXL	188.149	8	1505.192
XNOR2X1	142.706	62	8847.772
XNOR2XL	126.045	88	11091.96
XOR2XL	127.375	209	26621.375
Total Decompressor Static Power			900979.42 nW

We can then compare these calculated values to Genus' results for the simulation runs in SimpleScalar, described in detail later in this report. Using 164.gzip as an example, we can observe the output behaviour of Cadence Genus with regards to static power consumption of the compressor hardware. The results are shown in Table 4.15.

Table 4.15 – 164.gzip Compressor Static Power Values by Simulation Point

Simulation Point	Weight	P_{Static} (nW)
6	0.296296	568488.514
14	0.111111	568488.514
15	0.185185	568488.514
23	0.407407	568488.514

Notice from the results that that static power analysis is not affected when changing the input, which is an expected behaviour. Therefore, the following values are considered constant and valid and will be used throughout the remainder of this thesis:

Table 4.16 – Static Power for Compressor and Decompressor

Device	P_{Static} (nW)
Compressor	568488.514
Decompressor	900979.421

4.3 Dynamic Power Analysis

Dynamic power consists of three components: switching power, short-circuit power, and glitching power [27]. Genus groups these components into net power and internal power. Dynamic power is generally calculated as the following [27, 16]:

$$P_D = fCV_{DD}^2 \tag{4.6}$$

Net Power

Net power is the power consumption in a gate when charging the output load voltage from low to high. Therefore, Genus calculates net power as the following:

$$P_{Net} = 0.5 f_{toggle} C_L V_{DD}^2 \tag{4.7}$$

where f_{toggle} is the toggle rate calculated by Genus and C_L is the sum of load capacitances connected to the net.

Internal Power

Internal power is the product of frequency and "arc" energy for each input/output arc. Genus calculates internal power as the following:

$$P_{Internal} = \alpha_{A\to Y} E_{A\to Y} + \alpha_{B\to Y} E_{B\to Y} + ... + \alpha_{n\to Y} E_{n\to Y} \tag{4.8}$$

where $\propto_{A\to Y}$ is the arc activity calculated by Genus between input A and output Y and $E_{A\to Y}$ is the energy of the arc determined by Genus, based on the Liberty Timing File (.lib) for the chosen PDK.

Because dynamic power depends on the input stimulus to the compressor and decompressor modules, the most accurate way of modelling the dynamic power of these units is to use actual cache lines from the chosen benchmarks. Each time the compressor and decompressor must be accessed in the simulator, data is written to a file in Value Change Dump (VCD) format. This data is then input into Cadence Genus.

Table 4.17 through Table 4.20 provide an overview of the dynamic power results for the compressed configuration. The weighed dynamic power calculation for the compressor, using 164.gzip as an example, is shown in Table 4.17.

Table 4.17 – 164.gzip Compressor Dynamic Power Values by Simulation Point

Simulation Point	Weight	$P_{Internal}$ (nW)	P_{Net} (nW)	$P_{Dynamic}$ (nW)
6	0.296296	2,502,382.963	1,509,486.713	4,011,869.676
14	0.111111	2,376,079.125	1,425,466.881	3,801,546.007
15	0.185185	1,958,550.388	1,191,461.730	3,150,012.118
23	0.407407	2,642,784.746	1,573,089.284	4,215,874.030

$$\begin{aligned} P_{Dynamic} &= (0.296296)(4011869.676) + (0.111111)(3801546.007) \\ &\quad + (0.185185)(3150012.118) + (0.407407)(4215874.030) \\ &= 3{,}912{,}006.101\, nW \end{aligned} \tag{4.9}$$

The dynamic power results for the compressor, for each benchmark, are shown in Table 4.18.

Table 4.18 – Compressor Dynamic Power Results from Cadence Genus

Benchmark	P_{Static} (nW)	$P_{Internal}$ (nW)	P_{Net} (nW)	$P_{Dynamic}$ (nW)
164.gzip	568488.514	2444837.749	1467168.352	3912006.101
168.wupwise	568488.514	4733924.790	2825887.865	7559812.655
171.swim	568488.514	3406421.546	2142197.331	5548618.877
172.mgrid	568488.514	1503686.844	880473.875	2384160.718
173.applu	568488.514	2842913.555	1755334.081	4598247.636
175.vpr	568488.514	2687651.314	1589116.052	4276767.367
176.gcc	568488.514	1157472.704	654253.529	1811726.233
177.mesa	568488.514	2551430.640	1473686.003	4025116.642
179.art	568488.514	4479839.925	2742657.118	7222497.044
181.mcf	568488.514	3014341.404	1781392.370	4795733.774
183.equake	568488.514	2723545.966	1692572.412	4416118.377
188.ammp	568488.514	4531733.291	2794457.831	7326191.122
197.parser	568488.514	2125239.397	1209725.263	3334964.660
253.perlbmk	568488.514	2276058.199	1311922.307	3587980.505
255.vortex	568488.514	4255223.279	2478268.604	6733491.883
256.bzip2	568488.514	2000817.191	1158199.131	3159016.322
300.twolf	568488.514	1742468.002	979882.525	2722350.527
301.apsi	568488.514	4034095.505	2489728.965	6523824.469

The weighed dynamic power calculation for the decompressor, using 164.gzip as an example, is shown below.

Table 4.19 – 164.gzip Decompressor Power Values by Simulation Point

Simulation Point	Weight	$P_{Internal}$ (nW)	P_{Net} (nW)	$P_{Dynamic}$ (nW)
6	0.296296	512,909.725	328,296.277	841,206.002
14	0.111111	391,509.553	247,945.915	639,455.468
15	0.185185	265,728.129	137,641.278	403,369.407
23	0.407407	530,895.121	341,427.060	872,322.180

$$
\begin{aligned}
P_{Dynamic} &= (0.296296)(841206.002) + (0.111111)(639455.468) \\
&\quad + (0.185185)(403369.407) + (0.407407)(872322.180) \\
&= 750{,}384.636\ nW
\end{aligned}
\tag{4.10}
$$

The dynamic power results for the decompressor, for each benchmark, are shown in Table 4.20.

Table 4.20 – Decompressor Power Results from Cadence Genus

Benchmark	P_{Static} (nW)	$P_{Internal}$ (nW)	P_{Net} (nW)	$P_{Dynamic}$ (nW)
164.gzip	900979.421	460973.370	289411.266	750384.636
168.wupwise	900979.421	60397.361	33253.897	93651.257
171.swim	900979.421	1209495.482	749667.496	1959162.977
172.mgrid	900979.421	125365.641	55615.549	180981.190
173.applu	900979.421	16764.489	6941.406	23705.895
175.vpr	900979.421	202578.101	98126.196	300704.297
176.gcc	900979.421	484593.456	302914.825	787508.281
177.mesa	900979.421	567175.771	364661.884	931837.654
179.art	900979.421	241533.166	109215.950	350749.116
181.mcf	900979.421	217916.366	106922.446	324838.811
183.equake	900979.421	2549559.613	1590259.751	4139819.364
188.ammp	900979.421	238724.030	120220.861	358944.890
197.parser	900979.421	312999.100	152302.691	465301.791
253.perlbmk	900979.421	987344.724	635322.774	1622667.498
255.vortex	900979.421	369097.837	224798.216	593896.054
256.bzip2	900979.421	492697.294	304721.316	797418.610
300.twolf	900979.421	426706.339	274554.711	701261.051
301.apsi	900979.421	251257.052	129447.883	380704.935

4.4 Place and Route in Cadence Innovus

To confirm that the complexity of the hardware is not beyond implementation, Cadence Innovus is used to automatically place and route the design to chip. Figure 4.9 shows the final routing of the compressor in a 1mm by 1mm die.

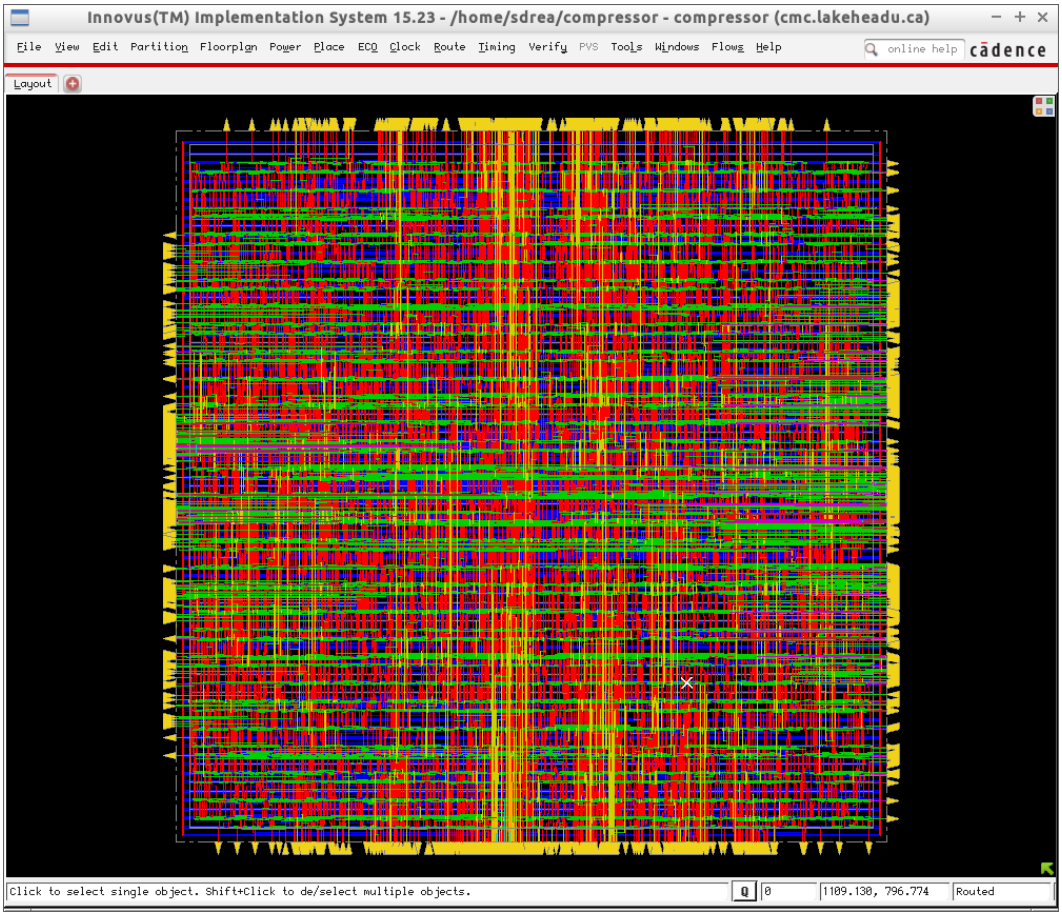

Figure 4.9 – Compressor Routing in Innovus

Chapter 5

Results

In this chapter, we look at the results of performance simulation. We start by reviewing the performance of the compression architecture, followed by the performance of the prefetch tables. Finally, we look at the overall performance of the new combined architecture.

5.1 Compression

As part of the compressibility check discussed in Chapter 4, we output in SimpleScalar a count of cache lines that are compressed by each of the schemes. Figure 5.1 shows the percentage of L1 data cache lines compressed by each compression scheme.

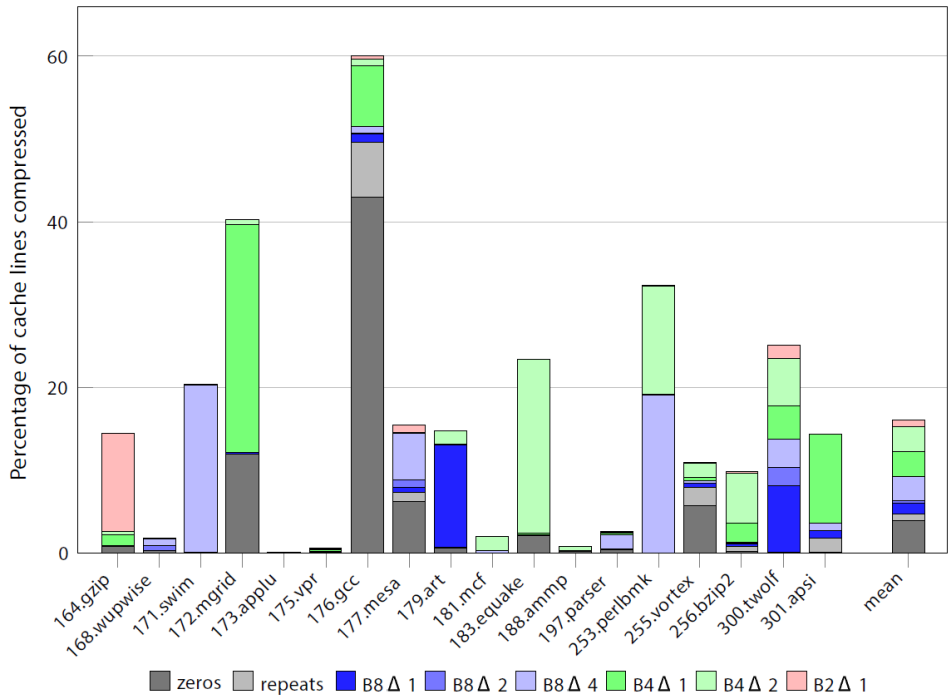

Figure 5.1 – Percentage of L1 Data Cache Lines Compressed by Each Scheme

As can be seen from the data, and as a verification of the results presented in [4], there is significant opportunity to apply Base-Delta-Immediate compression in L1 data cache. The best compression is achieved through zeros compression (64 bytes down to 8 bytes), so from Figure 5.1 we would expect a high compression ratio from 176.gcc because more than 40% of the cache lines are compressible using zeros compression. We also see that each of the compression schemes are well represented within the benchmarks. 300.twolf, for example, implements a nice balance of each of the Base-Delta-Immediate schemes.

Compression Ratio

We want to know what kind of impact this compression would have on the amount of data we are able to store in the cache. To do this, we can look at the compression ratio of each of the benchmarks. The compression ratio achieved by running each of the benchmarks through the simluator is shown in Figure 5.2. To calculate compression ratio, we compare the compressed size of the data with the uncompressed size:

$$\text{compression ratio} = \frac{\text{size}_{\text{uncompressed}}}{\text{size}_{\text{compressed}}} \tag{5.1}$$

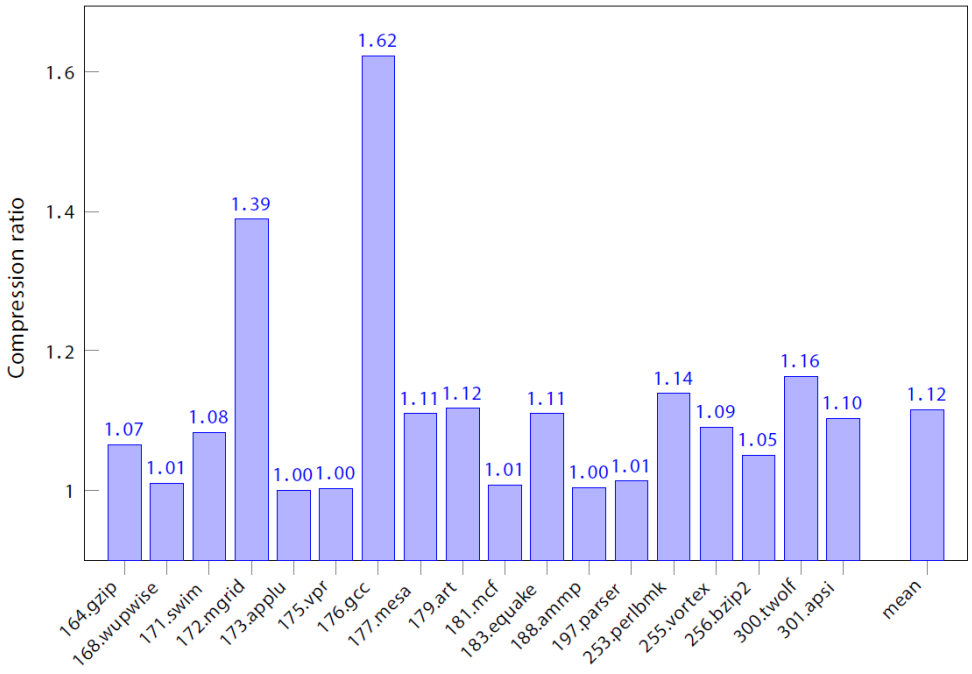

Figure 5.2 – Compression Ratio of L1 Data Cache

As expected from the compression rates shown previsouly in Figure 5.1, 176.gcc achieves the best compression mostly due to the 40% zeros compression. This is because **zeros** compression has the highest compression ratio among the schemes at a rate of 64/8. If we consider only this 40% zeros compression, and no other compressed cache lines, we would see the following compression ratio:

$$\text{compression ratio} = \frac{\text{size}_{\text{uncompressed}}}{\text{size}_{\text{compressed}}} = \frac{64}{.4(8) + .6(64)} = 1.54 \tag{5.2}$$

176.gcc does not achieve much more than this, with a ratio of 1.62. A compression ratio of 1.62 means that, on average, a 64-byte cache line is taking up 40 bytes of space. This is significant because it means, on average, each cache index holding 128 bytes now has room for 3 cache lines instead of 2.

Slowdown

We know, due to the decompression latency, that we will suffer a performance deterioration when we implement Base-Delta-Immediate compression – especially in L1 cache. To determine the slowdown, we compare the IPC of the compressed scheme versus the baseline scheme for each of the runs. The calculation for speedup and slowdown are shown below.

$$\text{speedup} = \frac{IPC_{compressed}}{IPC_{baseline}} \tag{5.3}$$

$$\text{slowdown} = 1 - speedup$$

Figure 5.3 shows the IPC of each of the benchmarks for the baseline configuration. Figure 5.4 shows the compressed scheme. The resultant speedup is shown in Figure 5.5.

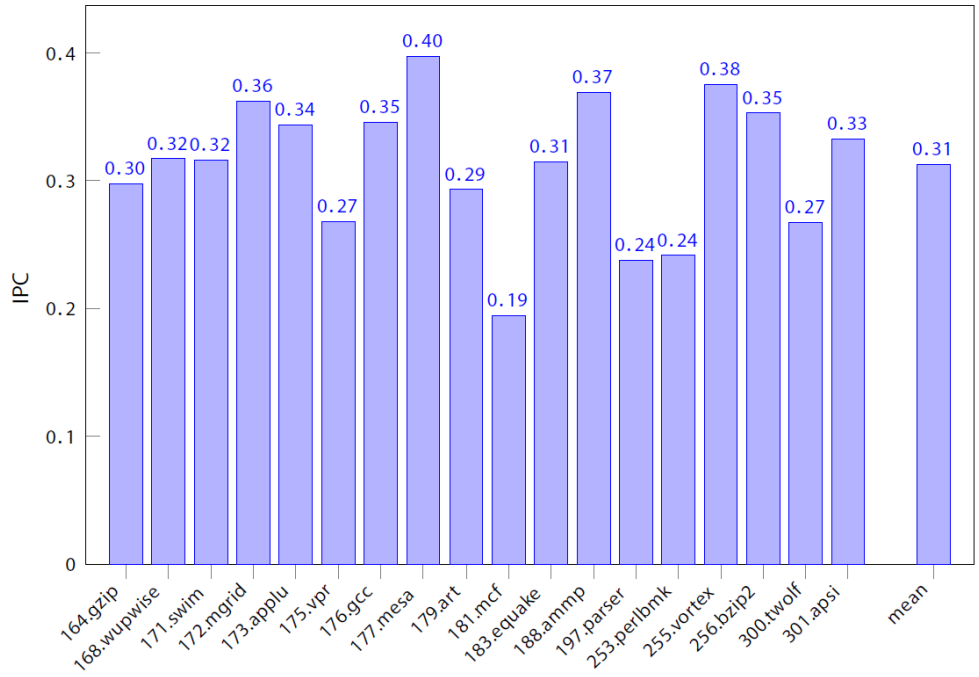

Figure 5.3 – IPC of Baseline Scheme

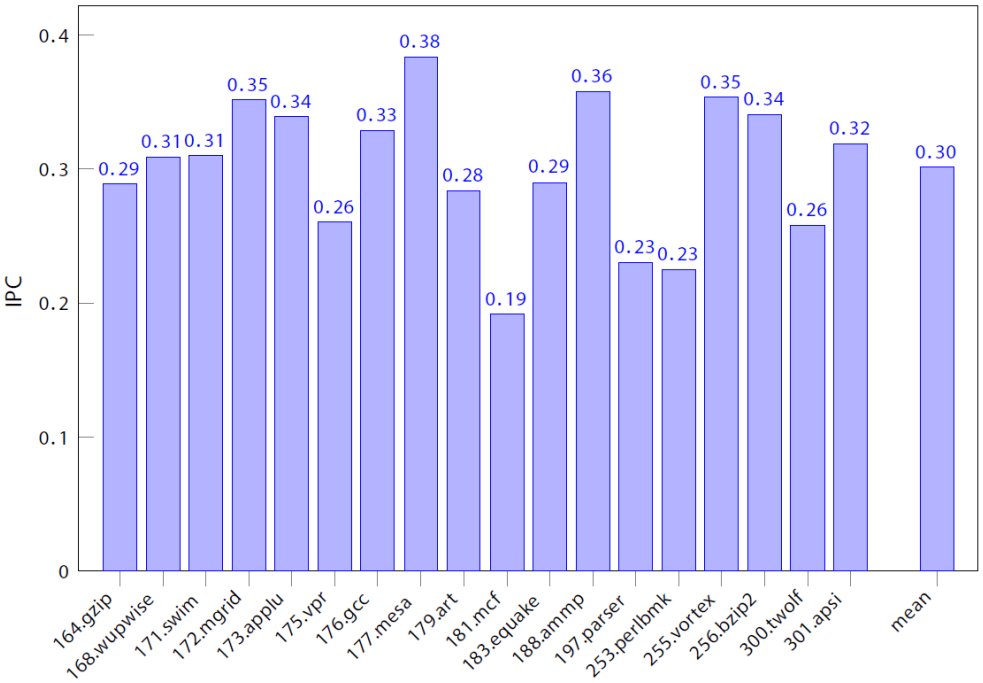

Figure 5.4 – IPC of Compressed Scheme

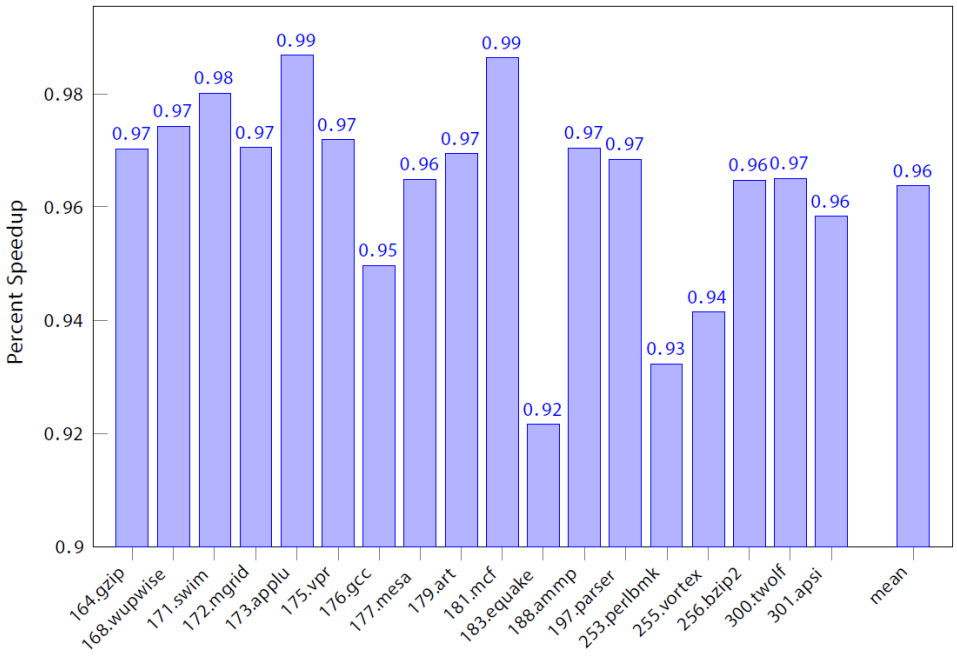

Figure 5.5 – Speedup of Compressed Scheme vs Baseline

What is siginificant here is that 183.equake has the most slowdown due to compression, yet 176.gcc has the highest compression ratio. This would likely be due to 183.equake experiencing more compressed cache hits and therefore experiencing more of the impact of the decompression latency.

Static Power

The primary intent of implementing compression in L1 data cache is to reduce the size and therefore the power consumption of the cache. The amount of power savings here is important because this savings should outweigh any penalties introduced in the prefetching architecture or in the slowdown of performance.

Figure 5.6 shows the static energy consumption of the L1 data cache for the baseline scheme. Figure 5.7 shows the static energy for the compressed scheme, including the compression and decompression hardware energy. Figure 5.8 shows the ratio of compressed to baseline to highlight the reduction.

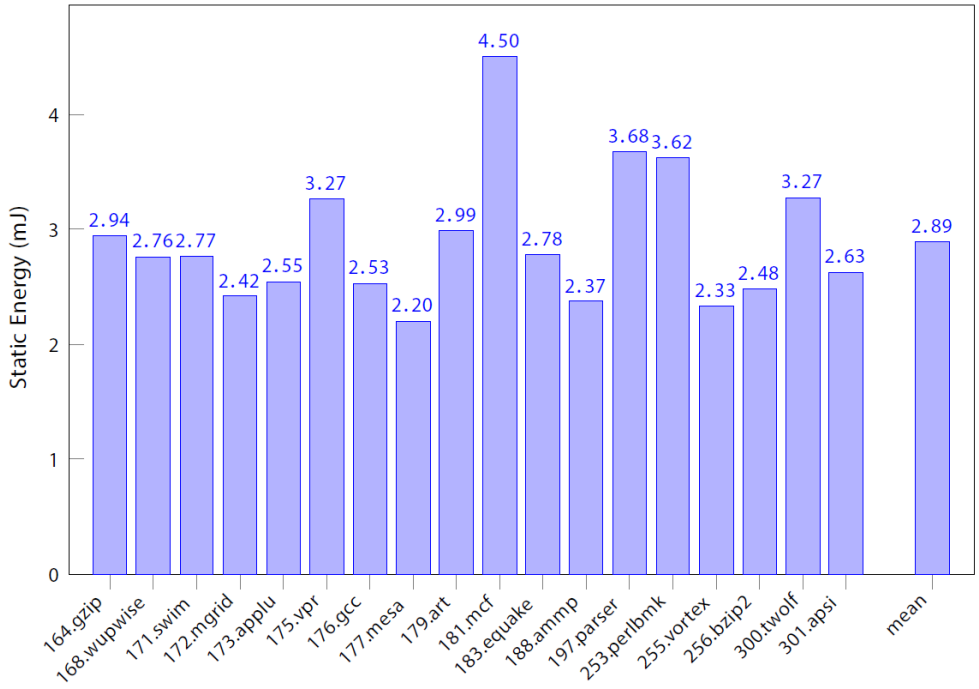

Figure 5.6 – L1 Data Cache Static Energy (Baseline Scheme)

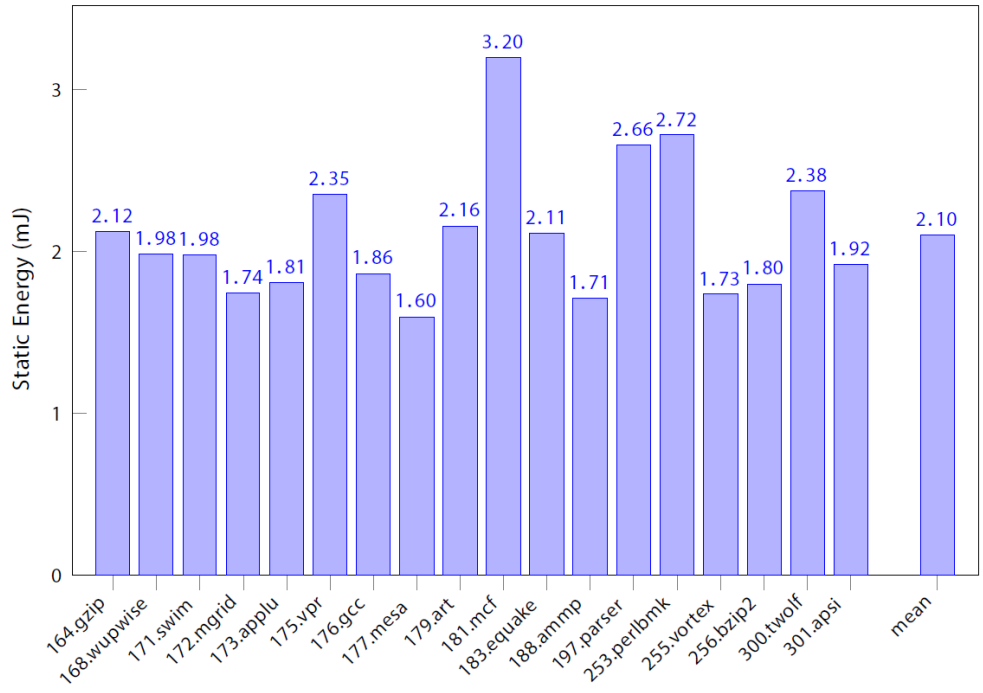

Figure 5.7 – L1 Data Cache Static Energy (Compressed Scheme)

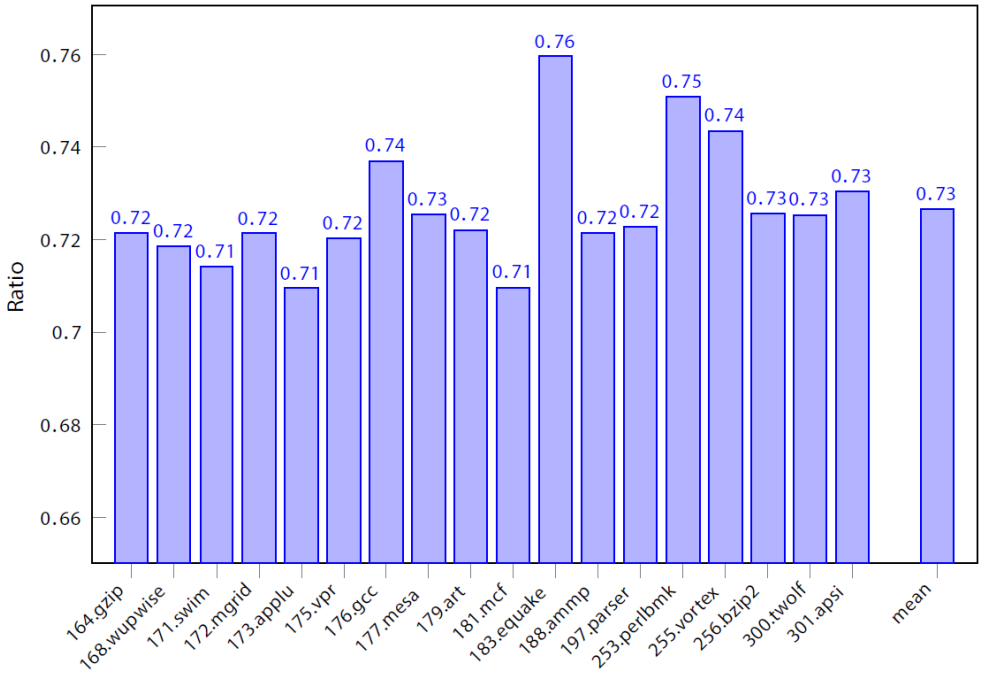

Figure 5.8 – L1 Data Cache Static Energy Ratio – Compressed vs Baseline

We see a significant static energy reduction in the cache itself due to its decrease in size. Looking at the static power from the CACTI model in Table 4.5, we would expect to see a reduction equal to:

$$\frac{P_{compressed}}{P_{baseline}} = \frac{13.6143 + 3.29114}{25.0286 + 1.22089} = 0.64 \tag{5.4}$$

However, 0.64 is not achieved due to the CPU slowdown caused by introducing compression and the power overhead of the compression and decompression hardware. In fact, you can correlate the balance of static power to the percent slowdown of the CPU due to compression. Comparing Figure 5.8 with the slowdown in Figure 5.5, you see that they complement eachother in this regard.

Dynamic Power

Because we change the cache performance as discussed above, the switching characteristics will change. In addition, the overall reduction in cache area will impact the configuration of the cache and therefore the energy required to read and write the cache.

Figure 5.9 shows the dynamic energy consumption of the L1 data cache for the baseline scheme. Figure 5.10 shows the dynamic energy for the compressed scheme, including the energy consumed in the compressor and decompressor. Figure 5.11 ashows the ratio of compressed to baseline to highlight the reduction.

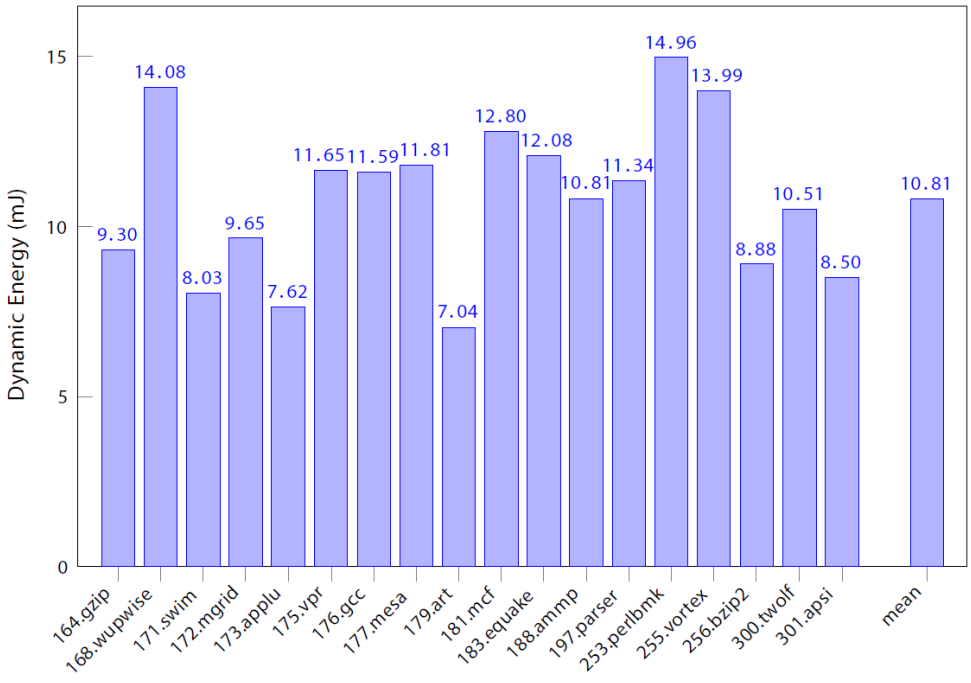

Figure 5.9 – L1 Data Cache Dynamic Energy (Baseline Scheme)

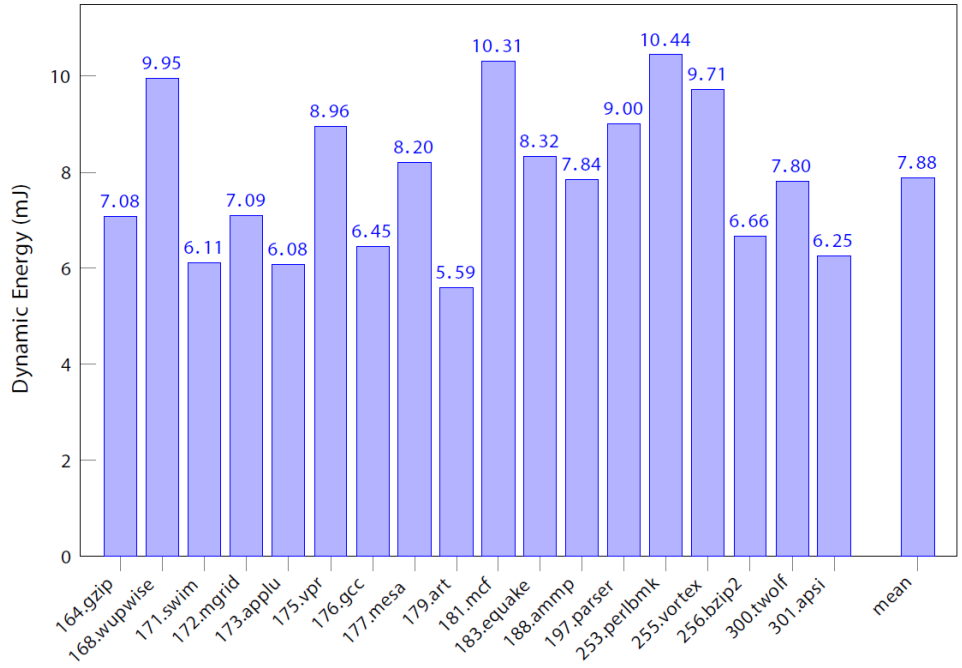

Figure 5.10 – L1 Data Cache Dynamic Energy (Compressed Scheme)

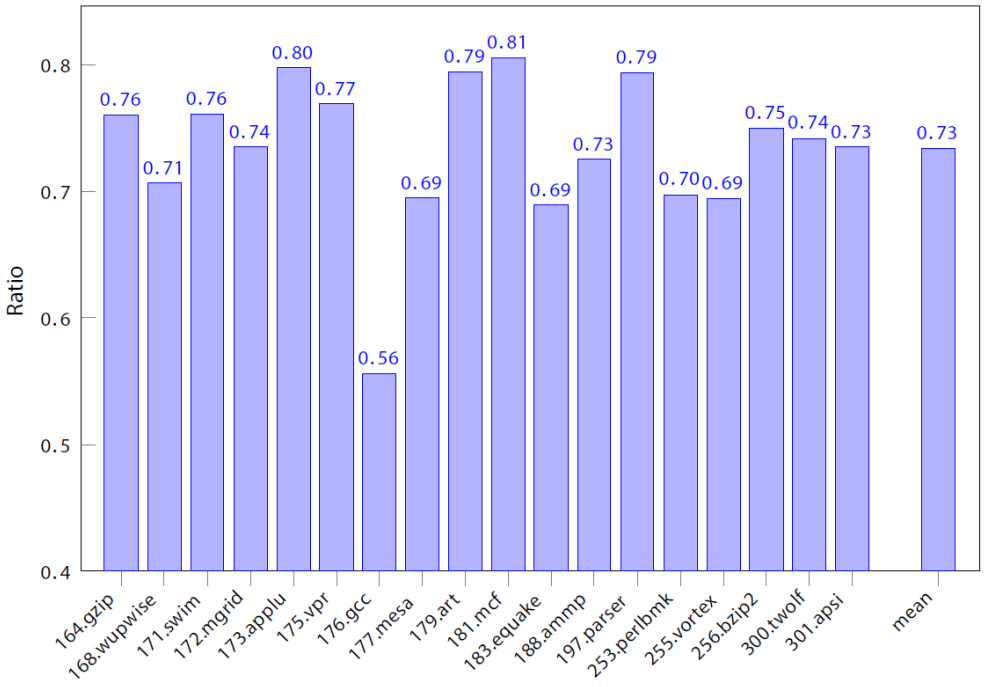

Figure 5.11 – L1 Data Cache Dynamic Energy Ratio – Compressed vs Baseline

We cannot compare the dynamic behaviour as we did with static and the slowdown. However, we do know that the compressed data size impacts the dynamic energy consumption of the cache. Therefore, we can determine how much of this energy reduction is due to the reduced data size by looking at the compression

ratio. For example, 176.gcc has a compression ratio of 1.62. This represents an average data size reduction of:

$$avg\ data\ reduction = \frac{1}{1.62} = 0.62 \tag{5.5}$$

The remaining energy reduction or gains in the CPU are due to the change in energy per access as well as the overall change in performance of the CPU.

5.2 Prefetching

We first look at the overall performance of all of the prefetching configurations used. To evaluate the performance of the prefetch tables, we consider two key elements. First, we look at what the hit percentage is for the table during the instruction fetch stage in the processor. That is, what percentage of load instructions successfully acquire a prediction address from the prefetch table based on the program counter only. Second, we look at how accurate those predictions are. By reviewing the state of decompressed lines in the decompression buffer when they are evicted, we can better understand how the prefetch tables are affecting the performance of the new architecture. In addition, this metric sheds some light on where improvements can be made to this architecture, as we will see in the data to follow.

Hit Percentage

A 128-Set and 1K-Set table were simulated for each of the prefetch table types (Last Outcome, Stride, Hybrid S/LO, Two-Level, and Hybrid 2L/S). To understand this selection, consider the static energy savings of compression presented in Figure 5.8. On average, we see a savings ratio of 0.27, which represents 0.79mJ in static energy, or 7.12mW in static power across the executed benchmarks. Reviewing the CACTI results in Table 4.7, the only 2K-Set table that keeps its static power within this range is the Last Outcome table. Therefore, we did not exceed 1K table sizes as we did not want to consume our static power savings entirely within the prefetch table.

For each of the 10 configurations, Figure 5.12 shows the percentage of load instructions that successfully receive a prediction address from the 128-Set prefetch tables. Figure 5.13 shows the percentage of instructions that hit the 1K prefetch tables.

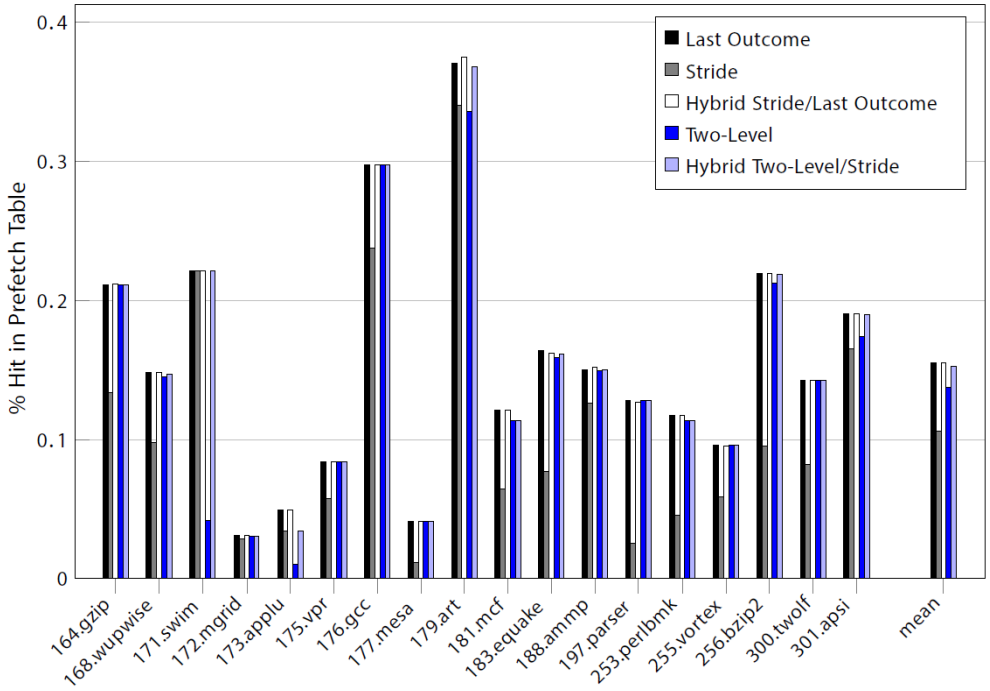

Figure 5.12 – Hit Percentage of Load Instructions by Prefetch Table (128 Set)

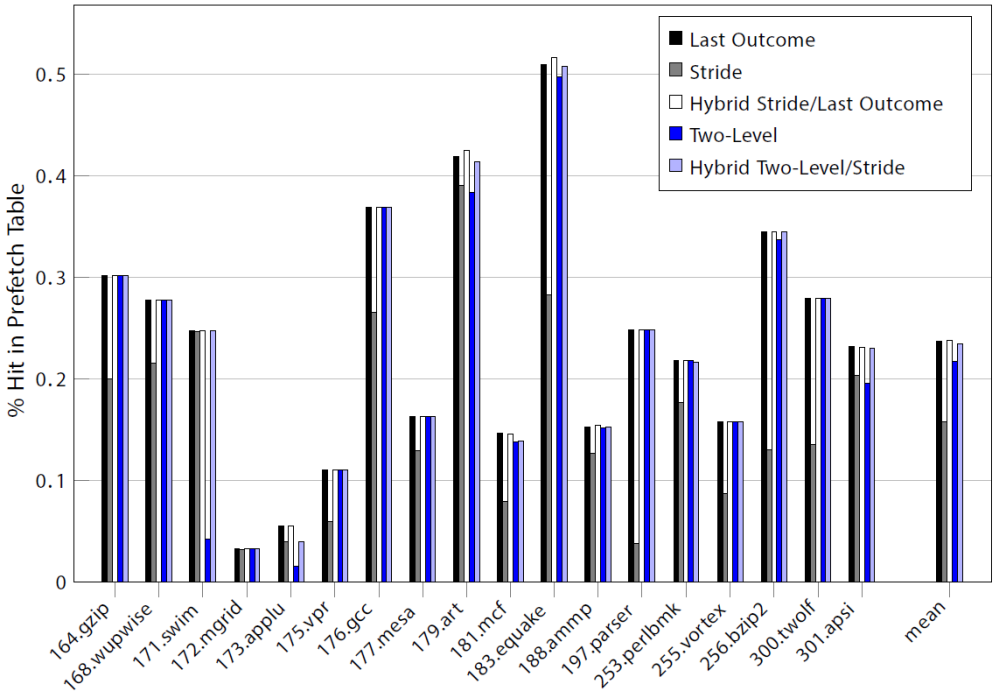

Figure 5.13 – Hit Percentage of Load Instructions by Prefetch Table (1K Set)

Figure 5.13 shows that the 1K-Set variation of each table out performs the 128-Set variant of the table. This is due to the reduction of conflict misses in the table. We also notice that overall, the prefetch table hit percentage is quite low, averaging 10% to 15% for 128-Set and 16% to 24% for 1K-Set.

Prediction Accuracy

When a prediction is made, data is decompressed from the cache and then entered into the decompression buffer. There are five possible results for entries in this buffer. If the buffer is not large enough, entries are evicted before they can be used. Used entries can be correct or incorrect. In addition, entries may be tossed out due to cache replacement or branch misprediction. Figure 5.14 show the results for the 10 prefetch configurations.

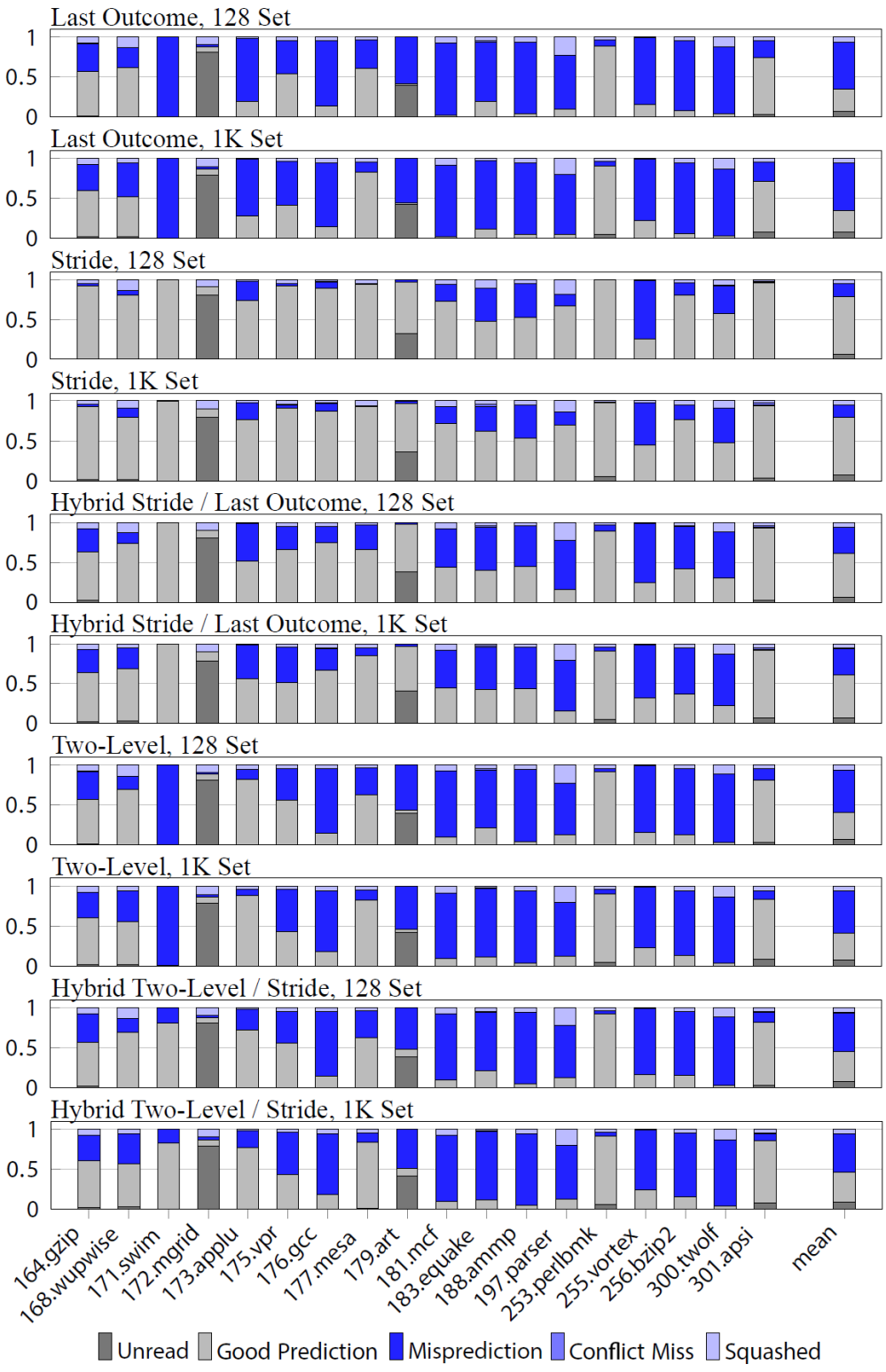

Figure 5.14 – Prediction Accuracy of 10 Prefetch Table Configurations

Some major factors stand out here. First, we notice that Stride prefetching is easily the most accurate. Second, we see that Last Outcome results in a large number of incorrect predictions. While these mispredictions do not directly impact the performance of the CPU, they do require an additional cache access which consumes unnecessary energy.

In addition, we notice that benchmarks 172.mgrid and 179.art are dumping many of the decompressed results from the decompression buffer before ever using them. This means that the normal number of load instructions between the instruction fetch stage and the mem stage is larger than our 1K buffer design which has 16 entries.

Static Power

The static energy consumption of all of the combined prefetch tables is shown in Figure 5.15 for 128-Set, Figure 5.16 for 1K-Set. This data includes the prefetch table, decompression buffer, and pattern history table in the case of two-level prefetching.

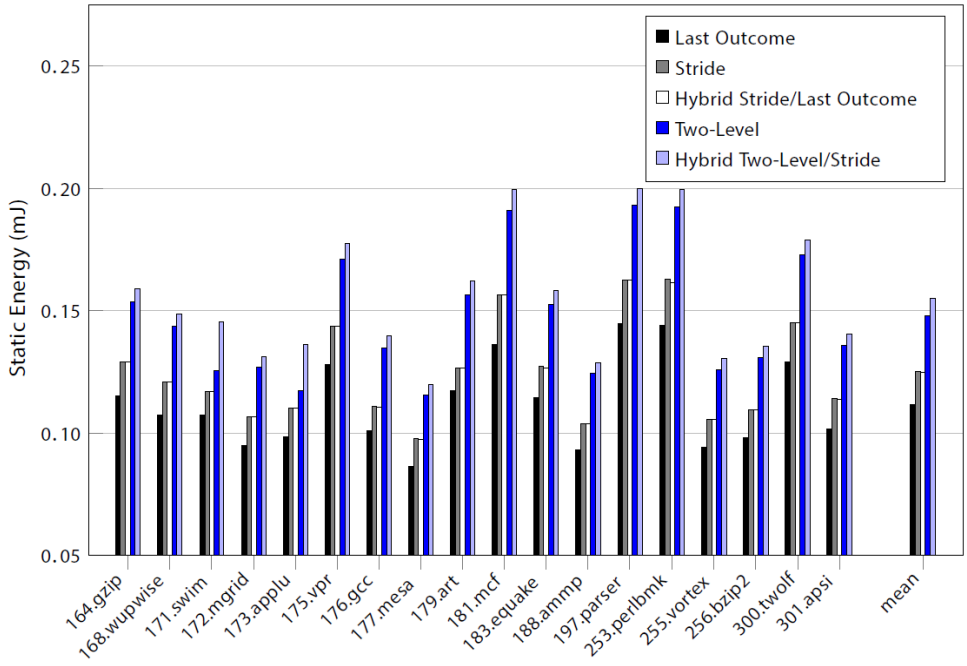

Figure 5.15 – Static Energy by Prefetch Table (128 Set)

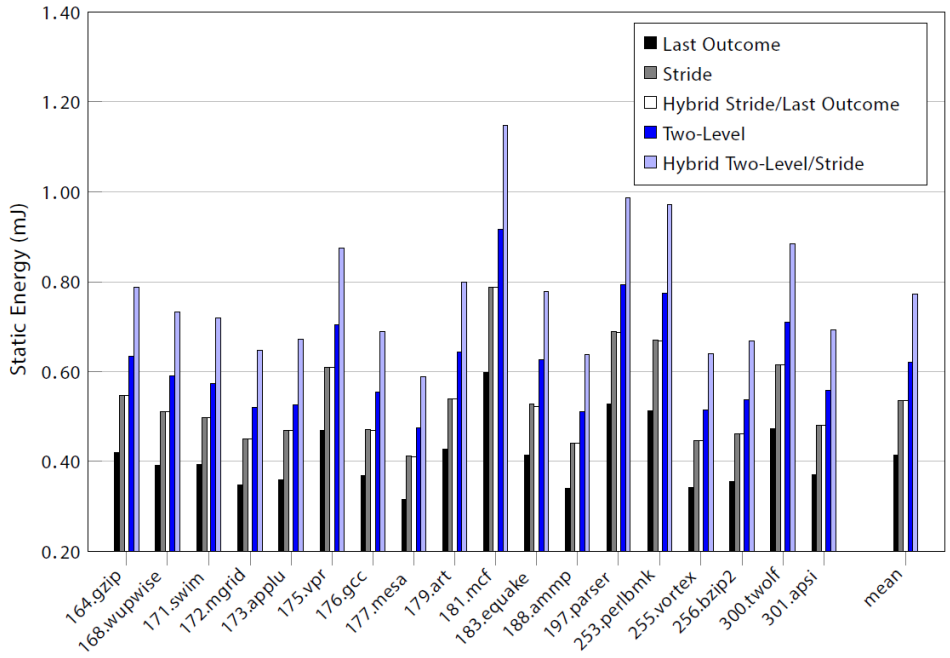

Figure 5.16 – Static Energy by Prefetch Table (1K Set)

Dynamic Power

The dynamic energy consumption of all of the combined prefetch tables is shown in Figure 5.17 for 128-Set, Figure 5.18 for 1K-Set. This data includes the prefetch table, decompression buffer, and pattern history table in the case of two-level prefetching.

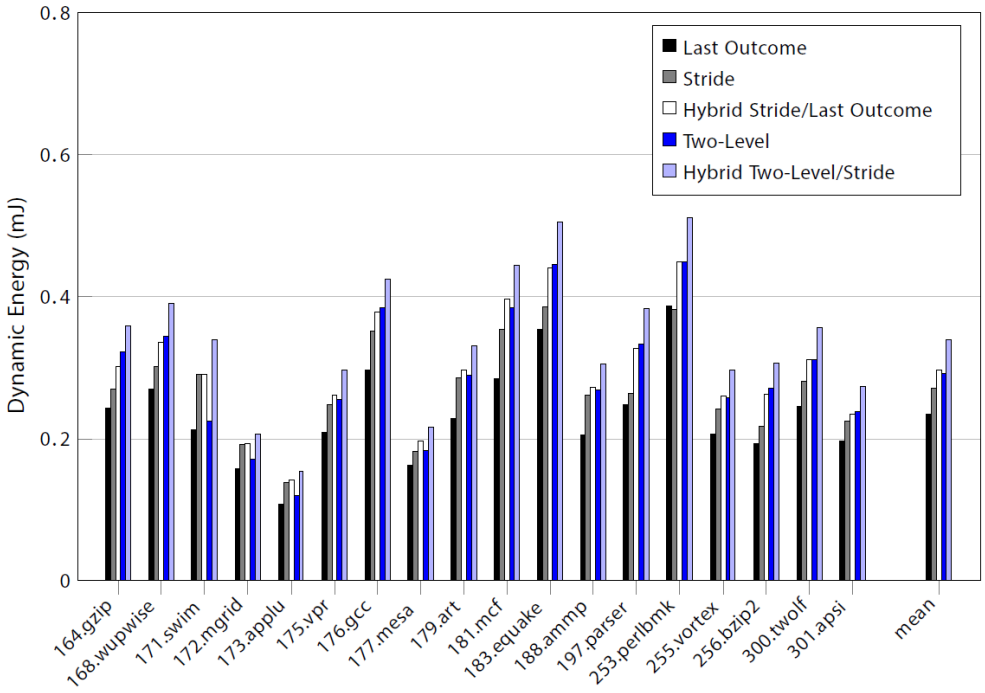

Figure 5.17 – Dynamic Energy by Prefetch Table (128 Set)

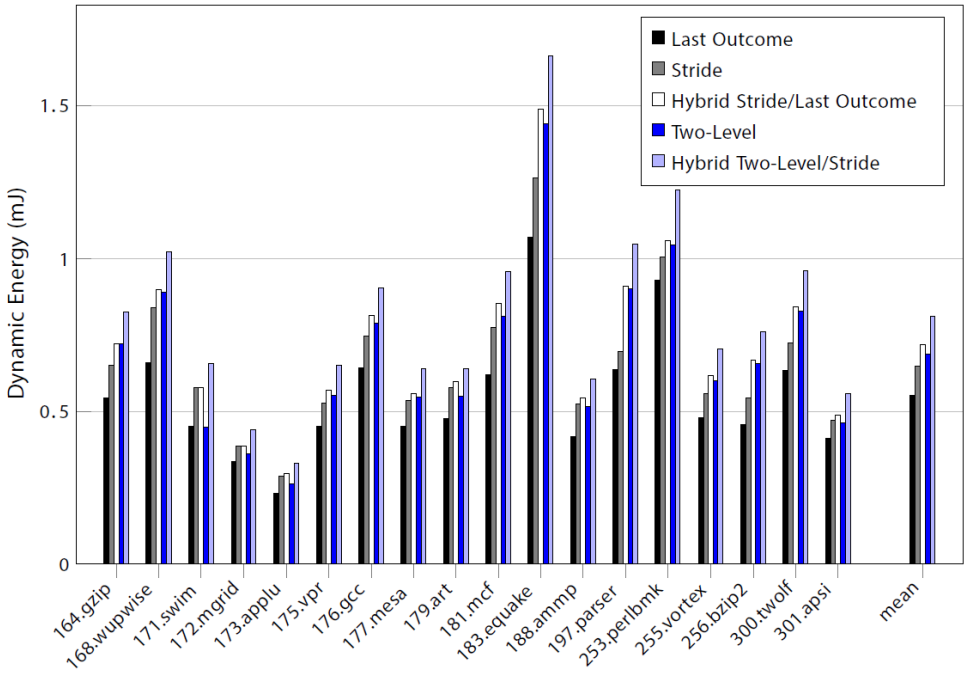

Figure 5.18 – Dynamic Energy by Prefetch Table (1K Set)

5.3 Compression and Prefetching

In this section, we look at the overall results of combining prefetching with cache compression and compare those results with the compression-only configuration.

Cache Energy vs. Performance

Figure 5.19 shows the slowdown versus the power consumed in L1 data cache. This figure identifies two important table configurations: Stride (128) and Hybrid Stride / Last Outcome (1K).

- **Stride (128)** provides the best speedup-to-dl1 energy relationship.

- **Hybrid Stride / Last Outcome (1K)** provides the best overall speedup, which we will see is the most important in the data to come.

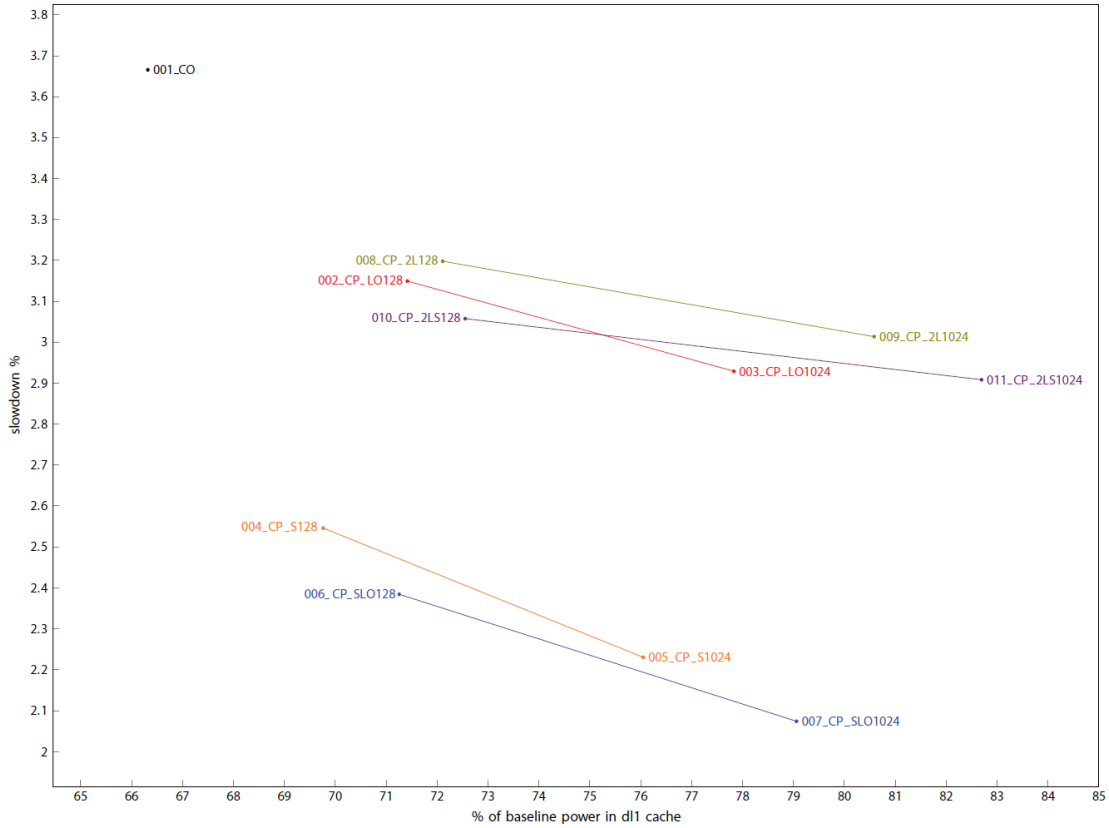

Figure 5.19 – L1 Data Cache Energy vs Performance

CPU Power vs. Performance

Figure 5.20 shows the overall energy savings in the CPU versus slowdown compared with the baseline configuration. **Stride (128)** and **Hybrid Stride / Last Outcome (128)** stand out here as well because they actually consume less energy than the compressed architecture alone. This is because the performance benefit of the prefetching actually results in a reduction in power.

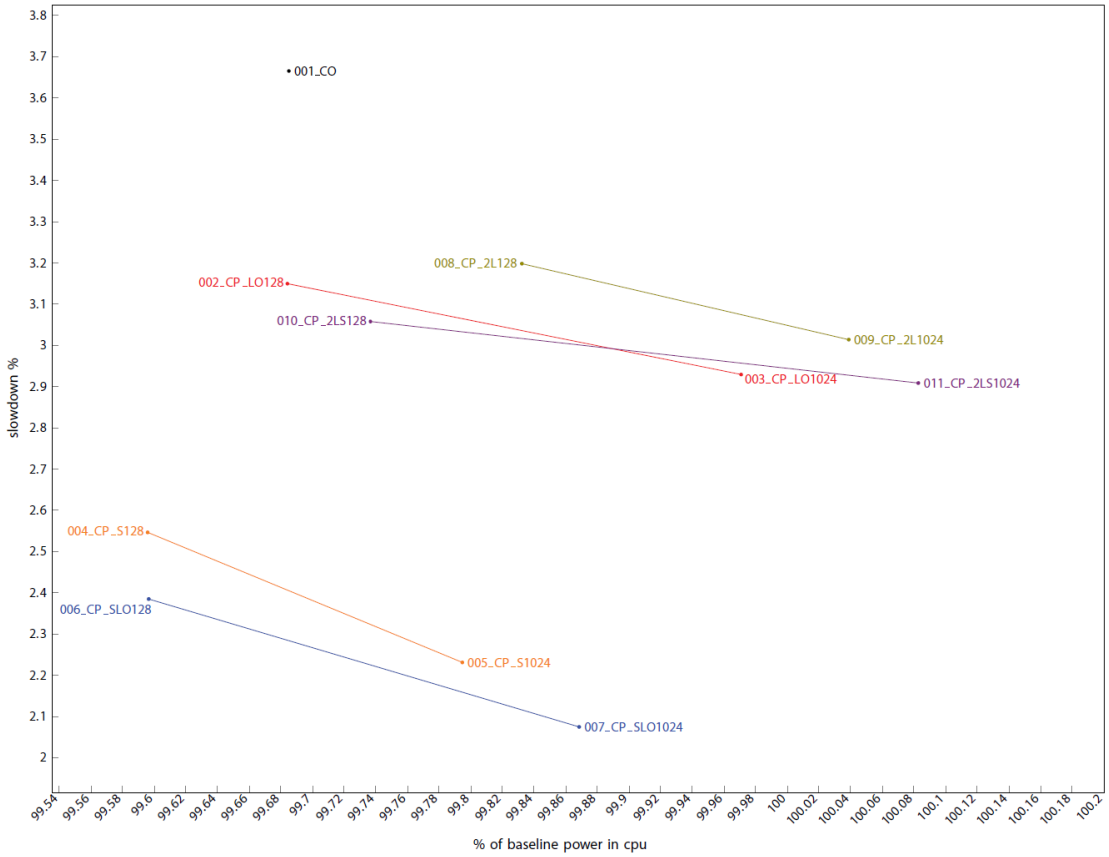

Figure 5.20 – CPU Energy vs. Performance

Speedup due to Prefetching

It is clear from the previous figure that performance plays an important role in the overall energy consumption of the CPU. In Figure 5.21, we compare the speedup of the different prefetching methods for each of the benchmarks for 128-Set tables, in Figure 5.22 for 1K-Set tables.

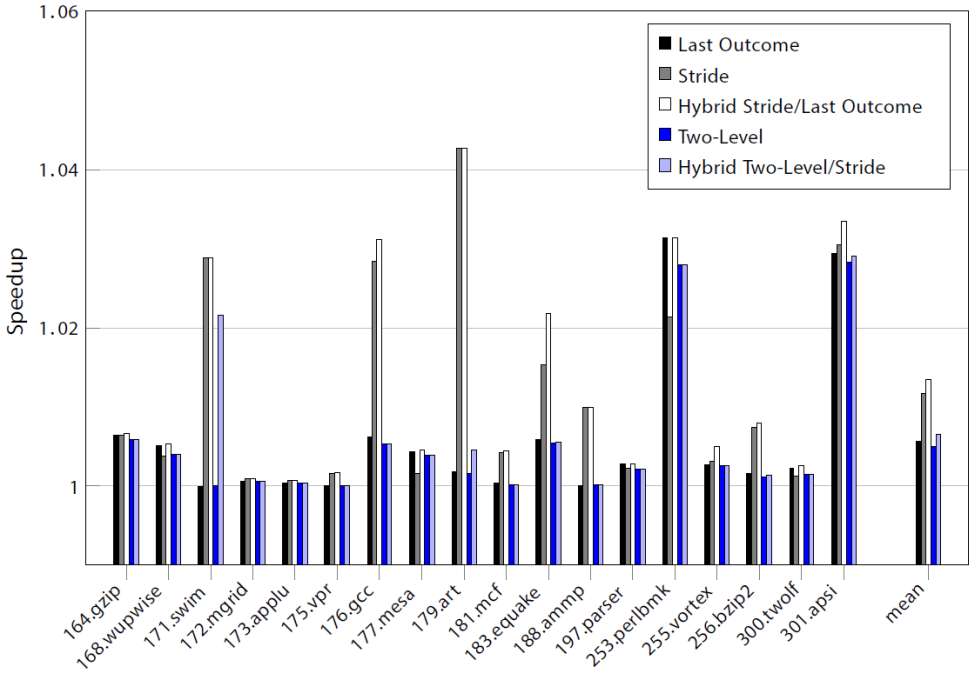

Figure 5.21 – Speedup Due to Prefetching (128 Set, vs. Compressed Only)

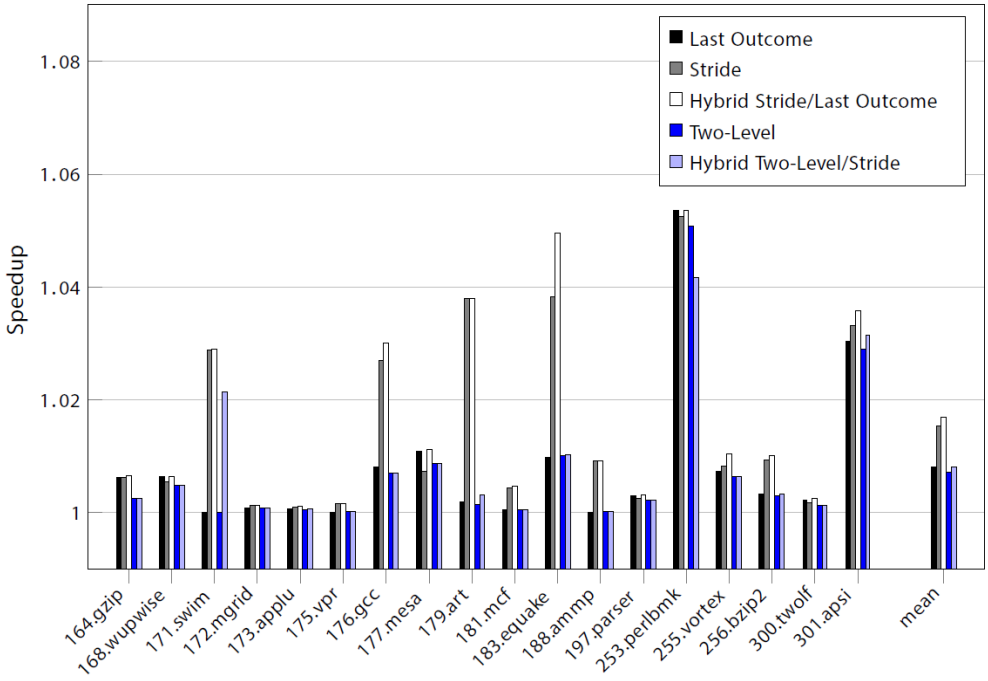

Figure 5.22 – Speedup Due to Prefetching (1K Set, vs. Compressed Only)

Energy-Delay Product

To determine which prefetching method stands out as the best, we need to consider both the overall performance of the CPU as well as the energy consumption. To do this, we use the product of the two metrics as follows:

$$EDP = Et \tag{5.6}$$

Where E is the total energy consumed by the CPU (in Joules) and t is the runtime of the program (in seconds). Figure 5.23 shows the energy-delay product, using values normalized to the baseline scheme, for each of the prefetch table configurations.

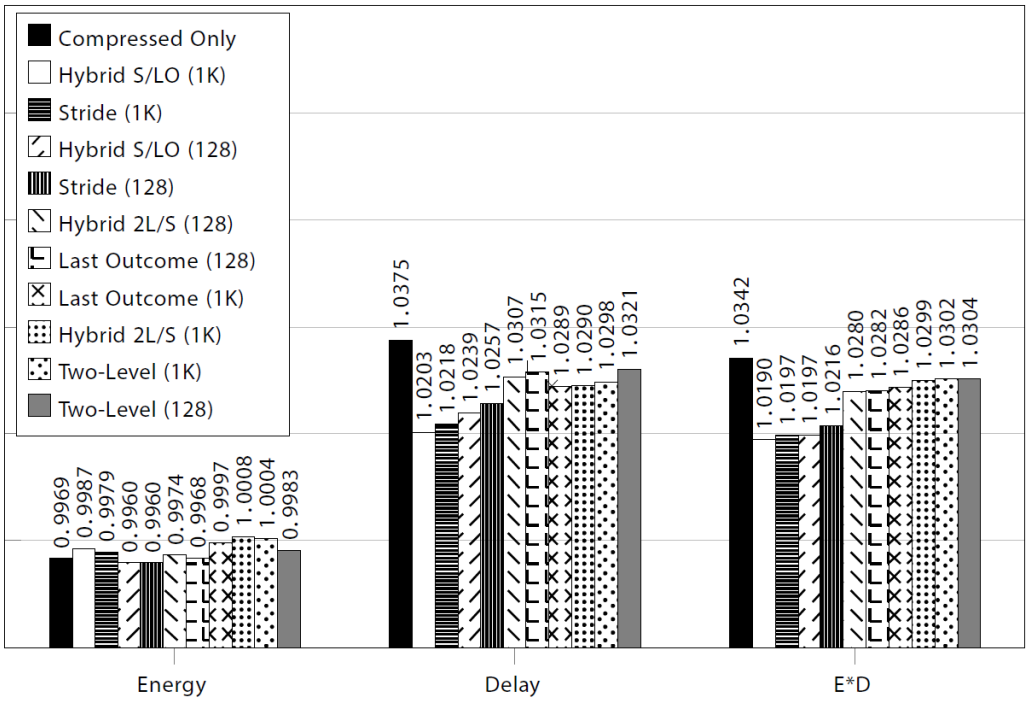

Figure 5.23 – Energy-Delay Product (CPU)

Figure 5.23 shows that all evaluated prefetch tables in combination with Base-Delta-Immediate (BΔI) compression outperform compression alone in L1 data cache. Based on the energy-delay product, the 1K Stride/Last Outcome table has the best overall performance for the SPEC CPU 2000 benchmarks used.

Chapter 6

Summary and Future Work

As hardware designers shift their priority to power-efficient architectures, compression research provides an opportunity to explore ways to handle smaller data in a processor. As we handle smaller data, dynamic energy reduces because we reduce the number of transistors that are being switched. As long as we can maintain the performance of the processor within reasonable slowdown constraints, we should be able to achieve better energy-delay configurations as we continue to research compression and prefetching architectures.

6.1 Contributions

Our work evaluates the potential for implementing compression in L1 data cache as a means of improving power efficiency. The proposed architecture combines prefetching with compression to move the decompression latency off the critical execution path. From the data provided by this work, we see that most prefetching tables implemented in this architecture provide an improvement over compression in L1 data cache alone. That is, the energy-delay product is improved by implementing prefetching versus no prefetching. In addition to having some residual slowdown which is impacting the energy-delay product, the slowdown itself causes an increase in static power that further amplifies the energy-delay product.

Also, as part of this research, we have developed a new branch of SimpleScalar specifically geared towards compression and prefetching. Even without further modification, this tool can be used in combination with CACTI to evaluate a wider range of cache configurations or prefetch table configurations. We have successfully implemented an interface between the tool and Cadence Genus for dynamic power analysis in the form of the Value Change Dump (VCD) output.

Lastly, we have successfully developed 64-byte compressor and decompressor units in 90nm CMOS that fall within acceptable power and timing constraints. This hardware is designed to work specifically with Base-Delta compression.

6.2 Future Work

Moving forward, there are opportunities to improve the configuration of prefetching tables within the proposed architecture. As can been seen from Figure 5.12 in the previous section, our prefetch tables suffer from a very low hit rate. In this data, a hit is considered when the table successfully makes a prediction (i.e. transient stride state counts as a miss). Even looking at 1K Last Outcome prefetching, which makes a prediction as long as the PC is indexed in the table, our average hit rate is less than 25%. Future work should look at improving the hit rate of the prefetch tables by experimenting with different table configurations, including deeper tables.

The decompression hardware produced during this work allows for the ability to decompress (in full or in part) multiple compressed cache lines during the Instruction Decode and Execution stages of processing. There is also opportunity to make predictions when the load instruction PC misses the prefetch table. The only penalties for making an incorrect prediction are an extra cache access and experiencing the full decompression latency. So, there may be opportunity to exchange some power savings for an improved energy-delay product.

In our branch of SimpleScalar, we have a number of areas to be worked on in the future. First and foremost, we need to address the gap caused by different versions of CACTI cache models being used. Wattch uses a cache model from an early release of CACTI. This creates a disconnect between the CPU power model and the compressed cache and prefetching table models developed in CACTI 6.5 for this thesis. There is an opportunity to revise the parts of Wattch used in our simulator to match the latest release of CACTI. In addition, rather than allowing for static and dynamic power input to the simulator, it would be ideal to instead input the cache configuration and use the CACTI functions directly in our simulator to extract the timing and power models automatically.

The Value Change Dump (VCD) output of the simulator is a nice feature when we want a detailed dynamic power model for specific hardware we have designed and synthesized to a PDK. Currently, the VCD output is quite large and requires compression separately from our simulator. Ideally, our simulator would

implement the **zlib** compression library in C and output the compressed VCD file for processing in Genus. This is ideal because Genus is already capable of reading in compressed VCD files.

Finally, in our compression and decompression hardware, more research could be done to determine faster and smaller designs that are compatible with this architecture. As mentioned previously, we implement the hierarchical carry-lookahead adder as a compromise between speed and resource utilization. In the case of the decompressor, certainly we must prioritize the speed of the hardware. However, for the compressor, we have taken the assumption that, because compression does not take place on the critical path of CPU execution, compressor delay is not necessarily a priority. Therefore, we may be able to develop a resource-optimized compressor to improve our power consumption.

This work has demonstrated that we can combine cache compression with prefetching to improve the performance of the CPU over implementing compression alone in L1 data caches. Future work in this area may identify better prefetching tables or more efficient decompression hardware that improves the feasibility of implementing compression in high-level caches.

Bibliography

[1] "10 Key Marketing Trends for 2017," IBM Watson Marketing.

[2] "The Cloud Begins with Coal," Digital Power Group, 2013.

[3] D. J. Frank, R. H. Dennard, E. Nowak, P. M. Solomon, Y. Taur and H.-S. P. Wong, "Device Scaling Limits of Si MOSFETs and Their Application Dependencies," *Proceedings of the IEEE,* vol. 89, no. 3, 2001.

[4] G. Pekhimenko, V. Seshadri, O. Mutlu, T. C. Mowry, P. B. Gibbons and M. A. Kozuch, "Base-Delta-Immediate Compression: A Practical Data Compression Mechanism for On-Chip Caches," Carnegie Mellon University , 2012.

[5] D. Brooks, V. Tiwari and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," in *Proceedings of the 27th International Symposium on Computer Architecture*, Vancouver, BC, 2000.

[6] A. R. Alameldeen and D. A. Wood, "Adaptive Cache Compression for High-Performance Processors," in *31st Annual International Symposium on Computer Architecture*, 2004.

[7] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for L2 caches," Computer Sciences Department, University of Wisconsin-Madison, 2004.

[8] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, 1997.

[9] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach," 5th ed., Waltham, MA, Morgan Kaufmann, 2012, pp. 71-144.

[10] X. Chen, L. Yang, R. P. Dick, L. Shang and H. Lekatsas, "C-Pack: A High-Performance Microprocessor Cache Compression Algorithm," *IEEE Transactions on VLSI Systems,* vol. 18, no. 8, 2010.

[11] J. Dusser, T. Piquet and A. Seznec, "Zero-content augmented caches," in *Proceedings of the 23rd international conference on Supercomputing (ICS '09)*, New York, 2009.

[12] J. Yang, Y. Zhang and R. Gupta, "Frequent value compression in data caches," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture (MICRO 33)*, New York, NY, 2000.

[13] E. Atoofian, "Many-Thread Aware Compression in GPGPUs," in *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress*, Toulouse, 2016.

[14] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (MICRO 30)*, Washington, DC, 1997.

[15] K. J. Nesbit and J. E. Smith, "Data Cache Prefetching Using a Global History Buffer," *IEE Proceedings Software,* 2004.

[16] S. Brown and Z. Vranesic, "Fundamentals of Digital Logic," 2nd ed., New York, NY, McGraw Hill, 2008.

[17] "ISE WebPACK Design Software," [Online]. Available: https://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.html.

[18] G. Hamerly, E. Perelman, J. Lau and B. Calder, "SimPoint 3.0: Faster and More Flexible Program Analysis," *Journal of Instruction Level Parallelism,* 2005.

[19] N. Muralimanohar, R. Balasubramonian and N. P. Jouppi, "CACTI 6.0: A Tool to Understand Large Caches," Hewlett-Packard Laboratories, 2009.

[20] J. MacQueen, "Some Methods for Classification and Analysis of Multivariable Observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[21] "SimPoint," [Online]. Available: http://cseweb.ucsd.edu/~calder/simpoint/index.htm.

[22] G. Hamerly, E. Perelman and B. Calder, "How to use SimPoint to pick simulation points," *ACM SIGMETRICS Performance Evaluation Review,* vol. 31, no. 4, pp. 25-30, 2004.

[23] "HP Labs : CACTI," [Online]. Available: http://www.hpl.hp.com/research/cacti/.

[24] "SimpleScalar LLC," [Online]. Available: http://www.simplescalar.com/.

[25] A. Moshovos, "Checkpointing alternatives for high performance, power-aware processors," in *Proceedings of the 2003 international symposium on Low power electronics and design (ISLPED '03)*, New York, NY, 2003.

[26] "Lakehead University High Performance Computing Centre (LUHPCC)," [Online]. Available: http://hpc.lakeheadu.ca/.

[27] M. W. Allam, "New Methodologies for Low-Power High-Performance Digital VLSI Design," Waterloo, ON, 2000.

[28] S. Thoziyoor, N. Muralimanohar, J. H. Ahn and N. P. Jouppi, "CACTI 5.1," Hewlett-Packard Laboratories, 2008.

[29] "Wattch Download," [Online]. Available: http://www.eecs.harvard.edu/~dbrooks/wattch-form.html.

[30] M. Hosseini, "A Survey of Data Compression Algorithms and their Applications," 2012.

[31] S. Mittal, "A Survey of Recent Prefetching Techniques for Processor Caches," *ACM Computing Surveys,* vol. 49, no. 2, 2016.

[32] J. A. Butts and G. S. Sohi, "A Static Power Model for Architects," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000.

[33] B. R. Rau and J. A. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective," *The Journal of Supercomputing,* vol. 7, no. 1, 1993.

Appendix A

Verilog Source

```
module compressor(
        input wire [511:0] x,
        output wire [127:0] b8d1,
        output wire [191:0] b8d2,
        output wire [319:0] b8d4,
        output wire [159:0] b4d1,
        output wire [287:0] b4d2,
        output wire [271:0] b2d1,
        output wire [63:0] repeats,
        output wire [7:0] zeros,
        output wire b8d1_valid,
        output wire b8d2_valid,
        output wire b8d4_valid,
        output wire b4d1_valid,
        output wire b4d2_valid,
        output wire b2d1_valid,
        output wire repeats_valid,
        output wire zeros_valid
        );

        wire [63:0] eightbyte0;
        wire [63:0] eightbyte1;
        wire [63:0] eightbyte2;
        wire [63:0] eightbyte3;
        wire [63:0] eightbyte4;
        wire [63:0] eightbyte5;
        wire [63:0] eightbyte6;
        wire [63:0] eightbyte7;
        wire [7:0] eightoverflow;
        wire [7:0] eightd1_valid;
        wire [7:0] eightd2_valid;
        wire [7:0] eightd4_valid;

        bdi inst00(
                .x(x[63:0]),
                .y(x[63:0]),
                .s(eightbyte0[63:0]),
                .overflow(eightoverflow[0]),
                .d1_valid(eightd1_valid[0]),
                .d2_valid(eightd2_valid[0]),
                .d4_valid(eightd4_valid[0])
        );

        bdi inst01(
                .x(x[63:0]),
                .y(x[127:64]),
                .s(eightbyte1[63:0]),
                .overflow(eightoverflow[1]),
                .d1_valid(eightd1_valid[1]),
                .d2_valid(eightd2_valid[1]),
                .d4_valid(eightd4_valid[1])
        );
```

```
bdi inst02(
        .x(x[63:0]),
        .y(x[191:128]),
        .s(eightbyte2[63:0]),
        .overflow(eightoverflow[2]),
        .d1_valid(eightd1_valid[2]),
        .d2_valid(eightd2_valid[2]),
        .d4_valid(eightd4_valid[2])
);

bdi inst03(
        .x(x[63:0]),
        .y(x[255:192]),
        .s(eightbyte3[63:0]),
        .overflow(eightoverflow[3]),
        .d1_valid(eightd1_valid[3]),
        .d2_valid(eightd2_valid[3]),
        .d4_valid(eightd4_valid[3])
);

bdi inst04(
        .x(x[63:0]),
        .y(x[319:256]),
        .s(eightbyte4[63:0]),
        .overflow(eightoverflow[4]),
        .d1_valid(eightd1_valid[4]),
        .d2_valid(eightd2_valid[4]),
        .d4_valid(eightd4_valid[4])
);

bdi inst05(
        .x(x[63:0]),
        .y(x[383:320]),
        .s(eightbyte5[63:0]),
        .overflow(eightoverflow[5]),
        .d1_valid(eightd1_valid[5]),
        .d2_valid(eightd2_valid[5]),
        .d4_valid(eightd4_valid[5])
);

bdi inst06(
        .x(x[63:0]),
        .y(x[447:384]),
        .s(eightbyte6[63:0]),
        .overflow(eightoverflow[6]),
        .d1_valid(eightd1_valid[6]),
        .d2_valid(eightd2_valid[6]),
        .d4_valid(eightd4_valid[6])
);

bdi inst07(
        .x(x[63:0]),
        .y(x[511:448]),
        .s(eightbyte7[63:0]),
        .overflow(eightoverflow[7]),
        .d1_valid(eightd1_valid[7]),
        .d2_valid(eightd2_valid[7]),
        .d4_valid(eightd4_valid[7])
);

assign b8d1_valid = (&eightd1_valid) & (&(~eightoverflow));
assign b8d2_valid = (&eightd2_valid) & (&(~eightoverflow));
assign b8d4_valid = (&eightd4_valid) & (&(~eightoverflow));

assign b8d1[63:0] = x[63:0];
assign b8d1[71:64] = eightbyte0[7:0];
assign b8d1[79:72] = eightbyte1[7:0];
assign b8d1[87:80] = eightbyte2[7:0];
assign b8d1[95:88] = eightbyte3[7:0];
```

```
assign b8d1[103:96] = eightbyte4[7:0];
assign b8d1[111:104] = eightbyte5[7:0];
assign b8d1[119:112] = eightbyte6[7:0];
assign b8d1[127:120] = eightbyte7[7:0];

assign b8d2[63:0] = x[63:0];
assign b8d2[79:64] = eightbyte0[15:0];
assign b8d2[95:80] = eightbyte1[15:0];
assign b8d2[111:96] = eightbyte2[15:0];
assign b8d2[127:112] = eightbyte3[15:0];
assign b8d2[143:128] = eightbyte4[15:0];
assign b8d2[159:144] = eightbyte5[15:0];
assign b8d2[175:160] = eightbyte6[15:0];
assign b8d2[191:176] = eightbyte7[15:0];

assign b8d4[63:0] = x[63:0];
assign b8d4[95:64] = eightbyte0[31:0];
assign b8d4[127:96] = eightbyte1[31:0];
assign b8d4[159:128] = eightbyte2[31:0];
assign b8d4[191:160] = eightbyte3[31:0];
assign b8d4[223:192] = eightbyte4[31:0];
assign b8d4[255:224] = eightbyte5[31:0];
assign b8d4[287:256] = eightbyte6[31:0];
assign b8d4[319:288] = eightbyte7[31:0];

wire [31:0] fourbyte0;
wire [31:0] fourbyte1;
wire [31:0] fourbyte2;
wire [31:0] fourbyte3;
wire [31:0] fourbyte4;
wire [31:0] fourbyte5;
wire [31:0] fourbyte6;
wire [31:0] fourbyte7;
wire [31:0] fourbyte8;
wire [31:0] fourbyte9;
wire [31:0] fourbyte10;
wire [31:0] fourbyte11;
wire [31:0] fourbyte12;
wire [31:0] fourbyte13;
wire [31:0] fourbyte14;
wire [31:0] fourbyte15;

wire [15:0] fouroverflow;
wire [15:0] fourd1_valid;
wire [15:0] fourd2_valid;
wire [15:0] fourd4_valid;

bdi32 inst10(
        .x(x[31:0]),
        .y(x[31:0]),
        .s(fourbyte0[31:0]),
        .overflow(fouroverflow[0]),
        .d1_valid(fourd1_valid[0]),
        .d2_valid(fourd2_valid[0]),
        .d4_valid(fourd4_valid[0])
);

bdi32 inst11(
        .x(x[31:0]),
        .y(x[63:32]),
        .s(fourbyte1[31:0]),
        .overflow(fouroverflow[1]),
        .d1_valid(fourd1_valid[1]),
        .d2_valid(fourd2_valid[1]),
        .d4_valid(fourd4_valid[1])
);
```

```
bdi32 inst12(
        .x(x[31:0]),
        .y(x[95:64]),
        .s(fourbyte2[31:0]),
        .overflow(fouroverflow[2]),
        .d1_valid(fourd1_valid[2]),
        .d2_valid(fourd2_valid[2]),
        .d4_valid(fourd4_valid[2])
);

bdi32 inst13(
        .x(x[31:0]),
        .y(x[127:96]),
        .s(fourbyte3[31:0]),
        .overflow(fouroverflow[3]),
        .d1_valid(fourd1_valid[3]),
        .d2_valid(fourd2_valid[3]),
        .d4_valid(fourd4_valid[3])
);

bdi32 inst14(
        .x(x[31:0]),
        .y(x[159:128]),
        .s(fourbyte4[31:0]),
        .overflow(fouroverflow[4]),
        .d1_valid(fourd1_valid[4]),
        .d2_valid(fourd2_valid[4]),
        .d4_valid(fourd4_valid[4])
);

bdi32 inst15(
        .x(x[31:0]),
        .y(x[191:160]),
        .s(fourbyte5[31:0]),
        .overflow(fouroverflow[5]),
        .d1_valid(fourd1_valid[5]),
        .d2_valid(fourd2_valid[5]),
        .d4_valid(fourd4_valid[5])
);

bdi32 inst16(
        .x(x[31:0]),
        .y(x[223:192]),
        .s(fourbyte6[31:0]),
        .overflow(fouroverflow[6]),
        .d1_valid(fourd1_valid[6]),
        .d2_valid(fourd2_valid[6]),
        .d4_valid(fourd4_valid[6])
);

bdi32 inst17(
        .x(x[31:0]),
        .y(x[255:224]),
        .s(fourbyte7[31:0]),
        .overflow(fouroverflow[7]),
        .d1_valid(fourd1_valid[7]),
        .d2_valid(fourd2_valid[7]),
        .d4_valid(fourd4_valid[7])
);

bdi32 inst18(
        .x(x[31:0]),
        .y(x[287:256]),
        .s(fourbyte8[31:0]),
        .overflow(fouroverflow[8]),
        .d1_valid(fourd1_valid[8]),
        .d2_valid(fourd2_valid[8]),
        .d4_valid(fourd4_valid[8])
);
```

```
bdi32 inst19(
        .x(x[31:0]),
        .y(x[319:288]),
        .s(fourbyte9[31:0]),
        .overflow(fouroverflow[9]),
        .d1_valid(fourd1_valid[9]),
        .d2_valid(fourd2_valid[9]),
        .d4_valid(fourd4_valid[9])
);

bdi32 inst110(
        .x(x[31:0]),
        .y(x[351:320]),
        .s(fourbyte10[31:0]),
        .overflow(fouroverflow[10]),
        .d1_valid(fourd1_valid[10]),
        .d2_valid(fourd2_valid[10]),
        .d4_valid(fourd4_valid[10])
);

bdi32 inst111(
        .x(x[31:0]),
        .y(x[383:352]),
        .s(fourbyte11[31:0]),
        .overflow(fouroverflow[11]),
        .d1_valid(fourd1_valid[11]),
        .d2_valid(fourd2_valid[11]),
        .d4_valid(fourd4_valid[11])
);

bdi32 inst112(
        .x(x[31:0]),
        .y(x[415:384]),
        .s(fourbyte12[31:0]),
        .overflow(fouroverflow[12]),
        .d1_valid(fourd1_valid[12]),
        .d2_valid(fourd2_valid[12]),
        .d4_valid(fourd4_valid[12])
);

bdi32 inst113(
        .x(x[31:0]),
        .y(x[447:416]),
        .s(fourbyte13[31:0]),
        .overflow(fouroverflow[13]),
        .d1_valid(fourd1_valid[13]),
        .d2_valid(fourd2_valid[13]),
        .d4_valid(fourd4_valid[13])
);

bdi32 inst114(
        .x(x[31:0]),
        .y(x[479:448]),
        .s(fourbyte14[31:0]),
        .overflow(fouroverflow[14]),
        .d1_valid(fourd1_valid[14]),
        .d2_valid(fourd2_valid[14]),
        .d4_valid(fourd4_valid[14])
);

bdi32 inst115(
        .x(x[31:0]),
        .y(x[511:480]),
        .s(fourbyte15[31:0]),
        .overflow(fouroverflow[15]),
        .d1_valid(fourd1_valid[15]),
        .d2_valid(fourd2_valid[15]),
        .d4_valid(fourd4_valid[15])
);
```

```
assign b4d1_valid = (&fourd1_valid) & (&(~fouroverflow));
assign b4d2_valid = (&fourd2_valid) & (&(~fouroverflow));


assign b4d1[31:0] = x[31:0];
assign b4d1[39:32] = fourbyte0[7:0];
assign b4d1[47:40] = fourbyte1[7:0];
assign b4d1[55:48] = fourbyte2[7:0];
assign b4d1[63:56] = fourbyte3[7:0];
assign b4d1[71:64] = fourbyte4[7:0];
assign b4d1[79:72] = fourbyte5[7:0];
assign b4d1[87:80] = fourbyte6[7:0];
assign b4d1[95:88] = fourbyte7[7:0];
assign b4d1[103:96] = fourbyte8[7:0];
assign b4d1[111:104] = fourbyte9[7:0];
assign b4d1[119:112] = fourbyte10[7:0];
assign b4d1[127:120] = fourbyte11[7:0];
assign b4d1[135:128] = fourbyte12[7:0];
assign b4d1[143:136] = fourbyte13[7:0];
assign b4d1[151:144] = fourbyte14[7:0];
assign b4d1[159:152] = fourbyte15[7:0];

assign b4d2[31:0] = x[31:0];
assign b4d2[47:32] = fourbyte0[15:0];
assign b4d2[63:48] = fourbyte1[15:0];
assign b4d2[79:64] = fourbyte2[15:0];
assign b4d2[95:80] = fourbyte3[15:0];
assign b4d2[111:96] = fourbyte4[15:0];
assign b4d2[127:112] = fourbyte5[15:0];
assign b4d2[143:128] = fourbyte6[15:0];
assign b4d2[159:144] = fourbyte7[15:0];
assign b4d2[175:160] = fourbyte8[15:0];
assign b4d2[191:176] = fourbyte9[15:0];
assign b4d2[207:192] = fourbyte10[15:0];
assign b4d2[223:208] = fourbyte11[15:0];
assign b4d2[239:224] = fourbyte12[15:0];
assign b4d2[255:240] = fourbyte13[15:0];
assign b4d2[271:256] = fourbyte14[15:0];
assign b4d2[287:272] = fourbyte15[15:0];

wire [15:0] twobyte0;
wire [15:0] twobyte1;
wire [15:0] twobyte2;
wire [15:0] twobyte3;
wire [15:0] twobyte4;
wire [15:0] twobyte5;
wire [15:0] twobyte6;
wire [15:0] twobyte7;
wire [15:0] twobyte8;
wire [15:0] twobyte9;
wire [15:0] twobyte10;
wire [15:0] twobyte11;
wire [15:0] twobyte12;
wire [15:0] twobyte13;
wire [15:0] twobyte14;
wire [15:0] twobyte15;
wire [15:0] twobyte16;
wire [15:0] twobyte17;
wire [15:0] twobyte18;
wire [15:0] twobyte19;
wire [15:0] twobyte20;
wire [15:0] twobyte21;
wire [15:0] twobyte22;
wire [15:0] twobyte23;
wire [15:0] twobyte24;
wire [15:0] twobyte25;
wire [15:0] twobyte26;
wire [15:0] twobyte27;
```

```
wire [15:0] twobyte28;
wire [15:0] twobyte29;
wire [15:0] twobyte30;
wire [15:0] twobyte31;

wire [31:0] twooverflow;
wire [31:0] twod1_valid;
wire [31:0] twod2_valid;
wire [31:0] twod4_valid;

bdi16 inst20(
        .x(x[15:0]),
        .y(x[15:0]),
        .s(twobyte0[15:0]),
        .overflow(twooverflow[0]),
        .d1_valid(twod1_valid[0]),
        .d2_valid(twod2_valid[0]),
        .d4_valid(twod4_valid[0])
);

bdi16 inst21(
        .x(x[15:0]),
        .y(x[31:16]),
        .s(twobyte1[15:0]),
        .overflow(twooverflow[1]),
        .d1_valid(twod1_valid[1]),
        .d2_valid(twod2_valid[1]),
        .d4_valid(twod4_valid[1])
);

bdi16 inst22(
        .x(x[15:0]),
        .y(x[47:32]),
        .s(twobyte2[15:0]),
        .overflow(twooverflow[2]),
        .d1_valid(twod1_valid[2]),
        .d2_valid(twod2_valid[2]),
        .d4_valid(twod4_valid[2])
);

bdi16 inst23(
        .x(x[15:0]),
        .y(x[63:48]),
        .s(twobyte3[15:0]),
        .overflow(twooverflow[3]),
        .d1_valid(twod1_valid[3]),
        .d2_valid(twod2_valid[3]),
        .d4_valid(twod4_valid[3])
);

bdi16 inst24(
        .x(x[15:0]),
        .y(x[79:64]),
        .s(twobyte4[15:0]),
        .overflow(twooverflow[4]),
        .d1_valid(twod1_valid[4]),
        .d2_valid(twod2_valid[4]),
        .d4_valid(twod4_valid[4])
);

bdi16 inst25(
        .x(x[15:0]),
        .y(x[95:80]),
        .s(twobyte5[15:0]),
        .overflow(twooverflow[5]),
        .d1_valid(twod1_valid[5]),
        .d2_valid(twod2_valid[5]),
        .d4_valid(twod4_valid[5])
);
```

```
bdi16 inst26(
        .x(x[15:0]),
        .y(x[111:96]),
        .s(twobyte6[15:0]),
        .overflow(twooverflow[6]),
        .d1_valid(twod1_valid[6]),
        .d2_valid(twod2_valid[6]),
        .d4_valid(twod4_valid[6])
);

bdi16 inst27(
        .x(x[15:0]),
        .y(x[127:112]),
        .s(twobyte7[15:0]),
        .overflow(twooverflow[7]),
        .d1_valid(twod1_valid[7]),
        .d2_valid(twod2_valid[7]),
        .d4_valid(twod4_valid[7])
);

bdi16 inst28(
        .x(x[15:0]),
        .y(x[143:128]),
        .s(twobyte8[15:0]),
        .overflow(twooverflow[8]),
        .d1_valid(twod1_valid[8]),
        .d2_valid(twod2_valid[8]),
        .d4_valid(twod4_valid[8])
);

bdi16 inst29(
        .x(x[15:0]),
        .y(x[159:144]),
        .s(twobyte9[15:0]),
        .overflow(twooverflow[9]),
        .d1_valid(twod1_valid[9]),
        .d2_valid(twod2_valid[9]),
        .d4_valid(twod4_valid[9])
);

bdi16 inst210(
        .x(x[15:0]),
        .y(x[175:160]),
        .s(twobyte10[15:0]),
        .overflow(twooverflow[10]),
        .d1_valid(twod1_valid[10]),
        .d2_valid(twod2_valid[10]),
        .d4_valid(twod4_valid[10])
);

bdi16 inst211(
        .x(x[15:0]),
        .y(x[191:176]),
        .s(twobyte11[15:0]),
        .overflow(twooverflow[11]),
        .d1_valid(twod1_valid[11]),
        .d2_valid(twod2_valid[11]),
        .d4_valid(twod4_valid[11])
);

bdi16 inst212(
        .x(x[15:0]),
        .y(x[207:192]),
        .s(twobyte12[15:0]),
        .overflow(twooverflow[12]),
        .d1_valid(twod1_valid[12]),
        .d2_valid(twod2_valid[12]),
        .d4_valid(twod4_valid[12])
);
```

```
bdi16 inst213(
        .x(x[15:0]),
        .y(x[223:208]),
        .s(twobyte13[15:0]),
        .overflow(twooverflow[13]),
        .d1_valid(twod1_valid[13]),
        .d2_valid(twod2_valid[13]),
        .d4_valid(twod4_valid[13])
);

bdi16 inst214(
        .x(x[15:0]),
        .y(x[239:224]),
        .s(twobyte14[15:0]),
        .overflow(twooverflow[14]),
        .d1_valid(twod1_valid[14]),
        .d2_valid(twod2_valid[14]),
        .d4_valid(twod4_valid[14])
);

bdi16 inst215(
        .x(x[15:0]),
        .y(x[255:240]),
        .s(twobyte15[15:0]),
        .overflow(twooverflow[15]),
        .d1_valid(twod1_valid[15]),
        .d2_valid(twod2_valid[15]),
        .d4_valid(twod4_valid[15])
);

bdi16 inst216(
        .x(x[15:0]),
        .y(x[271:256]),
        .s(twobyte16[15:0]),
        .overflow(twooverflow[16]),
        .d1_valid(twod1_valid[16]),
        .d2_valid(twod2_valid[16]),
        .d4_valid(twod4_valid[16])
);

bdi16 inst217(
        .x(x[15:0]),
        .y(x[287:272]),
        .s(twobyte17[15:0]),
        .overflow(twooverflow[17]),
        .d1_valid(twod1_valid[17]),
        .d2_valid(twod2_valid[17]),
        .d4_valid(twod4_valid[17])
);

bdi16 inst218(
        .x(x[15:0]),
        .y(x[303:288]),
        .s(twobyte18[15:0]),
        .overflow(twooverflow[18]),
        .d1_valid(twod1_valid[18]),
        .d2_valid(twod2_valid[18]),
        .d4_valid(twod4_valid[18])
);

bdi16 inst219(
        .x(x[15:0]),
        .y(x[319:304]),
        .s(twobyte19[15:0]),
        .overflow(twooverflow[19]),
        .d1_valid(twod1_valid[19]),
        .d2_valid(twod2_valid[19]),
        .d4_valid(twod4_valid[19])
);
```

```
bdi16 inst220(
        .x(x[15:0]),
        .y(x[335:320]),
        .s(twobyte20[15:0]),
        .overflow(twooverflow[20]),
        .d1_valid(twod1_valid[20]),
        .d2_valid(twod2_valid[20]),
        .d4_valid(twod4_valid[20])
);

bdi16 inst221(
        .x(x[15:0]),
        .y(x[351:336]),
        .s(twobyte21[15:0]),
        .overflow(twooverflow[21]),
        .d1_valid(twod1_valid[21]),
        .d2_valid(twod2_valid[21]),
        .d4_valid(twod4_valid[21])
);

bdi16 inst222(
        .x(x[15:0]),
        .y(x[367:352]),
        .s(twobyte22[15:0]),
        .overflow(twooverflow[22]),
        .d1_valid(twod1_valid[22]),
        .d2_valid(twod2_valid[22]),
        .d4_valid(twod4_valid[22])
);

bdi16 inst223(
        .x(x[15:0]),
        .y(x[383:368]),
        .s(twobyte23[15:0]),
        .overflow(twooverflow[23]),
        .d1_valid(twod1_valid[23]),
        .d2_valid(twod2_valid[23]),
        .d4_valid(twod4_valid[23])
);

bdi16 inst224(
        .x(x[15:0]),
        .y(x[399:384]),
        .s(twobyte24[15:0]),
        .overflow(twooverflow[24]),
        .d1_valid(twod1_valid[24]),
        .d2_valid(twod2_valid[24]),
        .d4_valid(twod4_valid[24])
);

bdi16 inst225(
        .x(x[15:0]),
        .y(x[415:400]),
        .s(twobyte25[15:0]),
        .overflow(twooverflow[25]),
        .d1_valid(twod1_valid[25]),
        .d2_valid(twod2_valid[25]),
        .d4_valid(twod4_valid[25])
);

bdi16 inst226(
        .x(x[15:0]),
        .y(x[431:416]),
        .s(twobyte26[15:0]),
        .overflow(twooverflow[26]),
        .d1_valid(twod1_valid[26]),
        .d2_valid(twod2_valid[26]),
        .d4_valid(twod4_valid[26])
);
```

```
bdi16 inst227(
        .x(x[15:0]),
        .y(x[447:432]),
        .s(twobyte27[15:0]),
        .overflow(twooverflow[27]),
        .d1_valid(twod1_valid[27]),
        .d2_valid(twod2_valid[27]),
        .d4_valid(twod4_valid[27])
);

bdi16 inst228(
        .x(x[15:0]),
        .y(x[463:448]),
        .s(twobyte28[15:0]),
        .overflow(twooverflow[28]),
        .d1_valid(twod1_valid[28]),
        .d2_valid(twod2_valid[28]),
        .d4_valid(twod4_valid[28])
);

bdi16 inst229(
        .x(x[15:0]),
        .y(x[479:464]),
        .s(twobyte29[15:0]),
        .overflow(twooverflow[29]),
        .d1_valid(twod1_valid[29]),
        .d2_valid(twod2_valid[29]),
        .d4_valid(twod4_valid[29])
);

bdi16 inst230(
        .x(x[15:0]),
        .y(x[495:480]),
        .s(twobyte30[15:0]),
        .overflow(twooverflow[30]),
        .d1_valid(twod1_valid[30]),
        .d2_valid(twod2_valid[30]),
        .d4_valid(twod4_valid[30])
);

bdi16 inst231(
        .x(x[15:0]),
        .y(x[511:496]),
        .s(twobyte31[15:0]),
        .overflow(twooverflow[31]),
        .d1_valid(twod1_valid[31]),
        .d2_valid(twod2_valid[31]),
        .d4_valid(twod4_valid[31])
);

assign b2d1_valid = (&twod1_valid) & (&(~twooverflow));

assign b2d1[15:0] = x[15:0];
assign b2d1[23:16] = twobyte0[7:0];
assign b2d1[31:24] = twobyte1[7:0];
assign b2d1[39:32] = twobyte2[7:0];
assign b2d1[47:40] = twobyte3[7:0];
assign b2d1[55:48] = twobyte4[7:0];
assign b2d1[63:56] = twobyte5[7:0];
assign b2d1[71:64] = twobyte6[7:0];
assign b2d1[79:72] = twobyte7[7:0];
assign b2d1[87:80] = twobyte8[7:0];
assign b2d1[95:88] = twobyte9[7:0];
assign b2d1[103:96] = twobyte10[7:0];
assign b2d1[111:104] = twobyte11[7:0];
assign b2d1[119:112] = twobyte12[7:0];
assign b2d1[127:120] = twobyte13[7:0];
assign b2d1[135:128] = twobyte14[7:0];
assign b2d1[143:136] = twobyte15[7:0];
```

```
        assign b2d1[151:144] = twobyte16[7:0];
        assign b2d1[159:152] = twobyte17[7:0];
        assign b2d1[167:160] = twobyte18[7:0];
        assign b2d1[175:168] = twobyte19[7:0];
        assign b2d1[183:176] = twobyte20[7:0];
        assign b2d1[191:184] = twobyte21[7:0];
        assign b2d1[199:192] = twobyte22[7:0];
        assign b2d1[207:200] = twobyte23[7:0];
        assign b2d1[215:208] = twobyte24[7:0];
        assign b2d1[223:216] = twobyte25[7:0];
        assign b2d1[231:224] = twobyte26[7:0];
        assign b2d1[239:232] = twobyte27[7:0];
        assign b2d1[247:240] = twobyte28[7:0];
        assign b2d1[255:248] = twobyte29[7:0];
        assign b2d1[263:256] = twobyte30[7:0];
        assign b2d1[271:264] = twobyte31[7:0];

        assign repeats[63:0] = x[63:0];
        assign repeats_valid = & ((x[63:0] ^~ x[127:64]) & (x[63:0] ^~ x[191:128]) & (x[63:0] ^~
        x[255:192]) & (x[63:0] ^~ x[319:256]) & (x[63:0] ^~ x[383:320]) & (x[63:0] ^~ x[447:384]) &
        (x[63:0] ^~ x[511:448]));

        assign zeros = 0;
        assign zeros_valid = &(~x);

endmodule

module bdi(
        input wire [63:0] x,
        input wire [63:0] y,
        output wire [63:0] s,
        output wire overflow,
        output wire d1_valid,
        output wire d2_valid,
        output wire d4_valid
        );

        wire c8, c16, c24, c32, c40, c48, c56;
        wire i00, i01, i02, i03, i04, i05, i06, i07, i08, i09;
        wire i10, i11, i12, i13, i14, i15, i16, i17, i18, i19;
        wire i20, i21, i22, i23, i24, i25, i26, i27, i28, i29;
        wire i30, i31, i32, i33, i34;
        wire g[7:0];
        wire p[7:0];
        wire [63:0] xnot;

        //subtractor
        assign xnot = ~x;
        assign c0 = 1;

        wire c64;

        and prim00 (i00, p[0], c0);
        or prim01 (c8, g[0], i00);

        and prim02 (i01, p[1], g[0]);
        and prim03 (i02, p[1], p[0], c0);
        or prim04 (c16, g[1], i01, i02);

        and prim05 (i03, p[2], g[1]);
        and prim06 (i04, p[2], p[1], g[0]);
        and prim07 (i05, p[2], p[1], p[0], c0);
        or prim08 (c24, g[2], i03, i04, i05);

        and prim09 (i06, p[3], g[2]);
        and prim10 (i07, p[3], p[2], g[1]);
        and prim11 (i08, p[3], p[2], p[1], g[0]);
        and prim12 (i09, p[3], p[2], p[1], p[0], c0);
        or prim13 (c32, g[3], i06, i07, i08, i09);
```

```
and prim14 (i10, p[4], g[3]);
and prim15 (i11, p[4], p[3], g[2]);
and prim16 (i12, p[4], p[3], p[2], g[1]);
and prim17 (i13, p[4], p[3], p[2], p[1], g[0]);
and prim18 (i14, p[4], p[3], p[2], p[1], p[0], c0);
or prim19 (c40, g[4], i10, i11, i12, i13, i14);

and prim20 (i15, p[5], g[4]);
and prim21 (i16, p[5], p[4], g[3]);
and prim22 (i17, p[5], p[4], p[3], g[2]);
and prim23 (i18, p[5], p[4], p[3], p[2], g[1]);
and prim24 (i19, p[5], p[4], p[3], p[2], p[1], g[0]);
and prim25 (i20, p[5], p[4], p[3], p[2], p[1], p[0], c0);
or prim26 (c48, g[5], i15, i16, i17, i18, i19, i20);

and prim27 (i21, p[6], g[5]);
and prim28 (i22, p[6], p[5], g[4]);
and prim29 (i23, p[6], p[5], p[4], g[3]);
and prim30 (i24, p[6], p[5], p[4], p[3], g[2]);
and prim31 (i25, p[6], p[5], p[4], p[3], p[2], g[1]);
and prim32 (i26, p[6], p[5], p[4], p[3], p[2], p[1], g[0]);
and prim33 (i27, p[6], p[5], p[4], p[3], p[2], p[1], p[0], c0);
or prim34 (c56, g[6], i21, i22, i23, i24, i25, i26, i27);

and prim35 (i28, p[7], g[6]);
and prim36 (i29, p[7], p[6], g[5]);
and prim37 (i30, p[7], p[6], p[5], g[4]);
and prim38 (i31, p[7], p[6], p[5], p[4], g[3]);
and prim39 (i32, p[7], p[6], p[5], p[4], p[3], g[2]);
and prim40 (i33, p[7], p[6], p[5], p[4], p[3], p[2], g[1]);
and prim41 (i34, p[7], p[6], p[5], p[4], p[3], p[2], p[1], g[0]);
and prim42 (i35, p[7], p[6], p[5], p[4], p[3], p[2], p[1], p[0], c0);
or prim43 (c64, g[7], i28, i29, i30, i31, i32, i33, i34, i35);

xor prim44 (overflow, c64, xnot[63], y[63], s[63]);
assign d1_valid = (&(s[63:7])) | (&(~s[63:7]));
assign d2_valid = (&(s[63:15])) | (&(~s[63:15]));
assign d4_valid = (&(s[63:31])) | (&(~s[63:31]));

hadder8 block0(
        .c0(c0),
        .x(xnot[7:0]),
        .y(y[7:0]),
        .s(s[7:0]),
        .G(g[0]),
        .P(p[0])
);

hadder8 block1(
        .c0(c8),
        .x(xnot[15:8]),
        .y(y[15:8]),
        .s(s[15:8]),
        .G(g[1]),
        .P(p[1])
);

hadder8 block2(
        .c0(c16),
        .x(xnot[23:16]),
        .y(y[23:16]),
        .s(s[23:16]),
        .G(g[2]),
        .P(p[2])
);
```

```
        hadder8 block3(
                .c0(c24),
                .x(xnot[31:24]),
                .y(y[31:24]),
                .s(s[31:24]),
                .G(g[3]),
                .P(p[3])
        );

        hadder8 block4(
                .c0(c32),
                .x(xnot[39:32]),
                .y(y[39:32]),
                .s(s[39:32]),
                .G(g[4]),
                .P(p[4])
        );

        hadder8 block5(
                .c0(c40),
                .x(xnot[47:40]),
                .y(y[47:40]),
                .s(s[47:40]),
                .G(g[5]),
                .P(p[5])
        );

        hadder8 block6(
                .c0(c48),
                .x(xnot[55:48]),
                .y(y[55:48]),
                .s(s[55:48]),
                .G(g[6]),
                .P(p[6])
        );

        hadder8 block7(
                .c0(c56),
                .x(xnot[63:56]),
                .y(y[63:56]),
                .s(s[63:56]),
                .G(g[7]),
                .P(p[7])
        );

endmodule

module bdi32(
        input wire [31:0] x,
        input wire [31:0] y,
        output wire [31:0] s,
        output wire overflow,
        output wire d1_valid,
        output wire d2_valid,
        output wire d4_valid
        );

        wire c8, c16, c24, c32;
        wire i00, i01, i02, i03, i04, i05, i06, i07, i08, i09;
        wire g[3:0];
        wire p[3:0];
        wire [31:0] xnot;

        //subtractor
        assign xnot = ~x;
        assign c0 = 1;

        and prim00 (i00, p[0], c0);
        or prim01 (c8, g[0], i00);
```

```
        and prim02 (i01, p[1], g[0]);
        and prim03 (i02, p[1], p[0], c0);
        or prim04 (c16, g[1], i01, i02);

        and prim05 (i03, p[2], g[1]);
        and prim06 (i04, p[2], p[1], g[0]);
        and prim07 (i05, p[2], p[1], p[0], c0);
        or prim08 (c24, g[2], i03, i04, i05);

        and prim09 (i06, p[3], g[2]);
        and prim10 (i07, p[3], p[2], g[1]);
        and prim11 (i08, p[3], p[2], p[1], g[0]);
        and prim12 (i09, p[3], p[2], p[1], p[0], c0);
        or prim13 (c32, g[3], i06, i07, i08, i09);

        xor prim44 (overflow, c32, xnot[31], y[31], s[31]);

        assign d1_valid = (&(s[31:7])) | (&(~s[31:7]));
        assign d2_valid = (&(s[31:15])) | (&(~s[31:15]));
        assign d4_valid = 0;

        hadder8 block0(
                .c0(c0),
                .x(xnot[7:0]),
                .y(y[7:0]),
                .s(s[7:0]),
                .G(g[0]),
                .P(p[0])
        );

        hadder8 block1(
                .c0(c8),
                .x(xnot[15:8]),
                .y(y[15:8]),
                .s(s[15:8]),
                .G(g[1]),
                .P(p[1])
        );

        hadder8 block2(
                .c0(c16),
                .x(xnot[23:16]),
                .y(y[23:16]),
                .s(s[23:16]),
                .G(g[2]),
                .P(p[2])
        );

        hadder8 block3(
                .c0(c24),
                .x(xnot[31:24]),
                .y(y[31:24]),
                .s(s[31:24]),
                .G(g[3]),
                .P(p[3])
        );

endmodule

module bdi16(
        input wire [15:0] x,
        input wire [15:0] y,
        output wire [15:0] s,
        output wire overflow,
        output wire d1_valid,
        output wire d2_valid,
        output wire d4_valid
        );
```

```
        wire c8, c16;
        wire i00, i01, i02;
        wire g[1:0];
        wire p[1:0];
        wire [15:0] xnot;

        //subtractor
        assign xnot = ~x;
        assign c0 = 1;

        and prim00 (i00, p[0], c0);
        or prim01 (c8, g[0], i00);

        and prim02 (i01, p[1], g[0]);
        and prim03 (i02, p[1], p[0], c0);
        or prim04 (c16, g[1], i01, i02);

        xor prim44 (overflow, c16, xnot[15], y[15], s[15]);

        assign d1_valid = (&(s[15:7])) | (&(~s[15:7]));
        assign d2_valid = 0;
        assign d4_valid = 0;

        hadder8 block0(
                .c0(c0),
                .x(xnot[7:0]),
                .y(y[7:0]),
                .s(s[7:0]),
                .G(g[0]),
                .P(p[0])
        );

        hadder8 block1(
                .c0(c8),
                .x(xnot[15:8]),
                .y(y[15:8]),
                .s(s[15:8]),
                .G(g[1]),
                .P(p[1])
        );

endmodule

module decompressor(
        input wire carry,
        input wire [3:0] encoding,
        input wire [511:0] x,
        output wire [511:0] b8d1,
        output wire [511:0] b8d2,
        output wire [511:0] b8d4,
        output wire [511:0] b4d1,
        output wire [511:0] b4d2,
        output wire [511:0] b2d1,
        output wire [511:0] repeats,
        output wire [511:0] zeros,
        output wire [511:0] uncompressed,
        output wire b8d1_valid,
        output wire b8d2_valid,
        output wire b8d4_valid,
        output wire b4d1_valid,
        output wire b4d2_valid,
        output wire b2d1_valid,
        output wire repeats_valid,
        output wire zeros_valid
        );

        reg [7:0] valids;
```

```
assign zeros_valid = valids[7];
assign repeats_valid = valids[6];
assign b8d1_valid = valids[5];
assign b8d2_valid = valids[4];
assign b8d4_valid = valids[3];
assign b4d1_valid = valids[2];
assign b4d2_valid = valids[1];
assign b2d1_valid = valids[0];

//uncomp
assign uncompressed = x;

//zeros
assign zeros = 0;

//repeats

assign repeats[63:0] = x[63:0];
assign repeats[127:64] = x[63:0];
assign repeats[191:128] = x[63:0];
assign repeats[255:192] = x[63:0];
assign repeats[319:256] = x[63:0];
assign repeats[383:320] = x[63:0];
assign repeats[447:384] = x[63:0];
assign repeats[511:448] = x[63:0];

//b8d1

hadd modb8d1b0(
        .c0(carry),
        .x(x[63:0]),
        .y({{56{x[71]}} , x[71:64]}),
        .s(b8d1[63:0])
);

hadd modb8d1b1(
        .c0(carry),
        .x(x[63:0]),
        .y({{56{x[79]}} , x[79:72]}),
        .s(b8d1[127:64])
);

hadd modb8d1b2(
        .c0(carry),
        .x(x[63:0]),
        .y({{56{x[87]}} , x[87:80]}),
        .s(b8d1[191:128])
);

hadd modb8d1b3(
        .c0(carry),
        .x(x[63:0]),
        .y({{56{x[95]}} , x[95:88]}),
        .s(b8d1[255:192])
);

hadd modb8d1b4(
        .c0(carry),
        .x(x[63:0]),
        .y({{56{x[103]}} , x[103:96]}),
        .s(b8d1[319:256])
);

hadd modb8d1b5(
        .c0(carry),
        .x(x[63:0]),
        .y({{56{x[111]}} , x[111:104]}),
        .s(b8d1[383:320])
);
```

```
hadd modb8d1b6(
        .c0(carry),
        .x(x[63:0]),
        .y({{56{x[119]}} , x[119:112]}),
        .s(b8d1[447:384])
);

hadd modb8d1b7(
        .c0(carry),
        .x(x[63:0]),
        .y({{56{x[127]}} , x[127:120]}),
        .s(b8d1[511:448])
);
//b8d2
hadd modb8d2b0(
        .c0(carry),
        .x(x[63:0]),
        .y({{48{x[79]}} , x[79:64]}),
        .s(b8d2[63:0])
);

hadd modb8d2b1(
        .c0(carry),
        .x(x[63:0]),
        .y({{48{x[95]}} , x[95:80]}),
        .s(b8d2[127:64])
);

hadd modb8d2b2(
        .c0(carry),
        .x(x[63:0]),
        .y({{48{x[111]}} , x[111:96]}),
        .s(b8d2[191:128])
);

hadd modb8d2b3(
        .c0(carry),
        .x(x[63:0]),
        .y({{48{x[127]}} , x[127:112]}),
        .s(b8d2[255:192])
);

hadd modb8d2b4(
        .c0(carry),
        .x(x[63:0]),
        .y({{48{x[143]}} , x[143:128]}),
        .s(b8d2[319:256])
);

hadd modb8d2b5(
        .c0(carry),
        .x(x[63:0]),
        .y({{48{x[159]}} , x[159:144]}),
        .s(b8d2[383:320])
);

hadd modb8d2b6(
        .c0(carry),
        .x(x[63:0]),
        .y({{48{x[175]}} , x[175:160]}),
        .s(b8d2[447:384])
);

hadd modb8d2b7(
        .c0(carry),
        .x(x[63:0]),
        .y({{48{x[191]}} , x[191:176]}),
        .s(b8d2[511:448])
);
```

```
//b8d4

hadd modb8d4b0(
        .c0(carry),
        .x(x[63:0]),
        .y({{32{x[95]}} , x[95:64]}),
        .s(b8d4[63:0])
);

hadd modb8d4b1(
        .c0(carry),
        .x(x[63:0]),
        .y({{32{x[127]}} , x[127:96]}),
        .s(b8d4[127:64])
);

hadd modb8d4b2(
        .c0(carry),
        .x(x[63:0]),
        .y({{32{x[159]}} , x[159:128]}),
        .s(b8d4[191:128])
);

hadd modb8d4b3(
        .c0(carry),
        .x(x[63:0]),
        .y({{32{x[191]}} , x[191:160]}),
        .s(b8d4[255:192])
);

hadd modb8d4b4(
        .c0(carry),
        .x(x[63:0]),
        .y({{32{x[223]}} , x[223:192]}),
        .s(b8d4[319:256])
);

hadd modb8d4b5(
        .c0(carry),
        .x(x[63:0]),
        .y({{32{x[255]}} , x[255:224]}),
        .s(b8d4[383:320])
);

hadd modb8d4b6(
        .c0(carry),
        .x(x[63:0]),
        .y({{32{x[287]}} , x[287:256]}),
        .s(b8d4[447:384])
);

hadd modb8d4b7(
        .c0(carry),
        .x(x[63:0]),
        .y({{32{x[319]}} , x[319:288]}),
        .s(b8d4[511:448])
);

//b4d1

hadd32 modb4d1b0(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[39]}} , x[39:32]}),
        .s(b4d1[31:0])
);
```

```
hadd32 modb4d1b1(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[47]}} , x[47:40]}),
        .s(b4d1[63:32])
);

hadd32 modb4d1b2(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[55]}} , x[55:48]}),
        .s(b4d1[95:64])
);

hadd32 modb4d1b3(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[63]}} , x[63:56]}),
        .s(b4d1[127:96])
);

hadd32 modb4d1b4(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[71]}} , x[71:64]}),
        .s(b4d1[159:128])
);

hadd32 modb4d1b5(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[79]}} , x[79:72]}),
        .s(b4d1[191:160])
);

hadd32 modb4d1b6(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[87]}} , x[87:80]}),
        .s(b4d1[223:192])
);

hadd32 modb4d1b7(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[95]}} , x[95:88]}),
        .s(b4d1[255:224])
);

hadd32 modb4d1b8(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[103]}} , x[103:96]}),
        .s(b4d1[287:256])
);

hadd32 modb4d1b9(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[111]}} , x[111:104]}),
        .s(b4d1[319:288])
);

hadd32 modb4d1b10(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[119]}} , x[119:112]}),
        .s(b4d1[351:320])
);
```

```
hadd32 modb4d1b11(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[127]}} , x[127:120]}),
        .s(b4d1[383:352])
);

hadd32 modb4d1b12(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[135]}} , x[135:128]}),
        .s(b4d1[415:384])
);

hadd32 modb4d1b13(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[143]}} , x[143:136]}),
        .s(b4d1[447:416])
);

hadd32 modb4d1b14(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[151]}} , x[151:144]}),
        .s(b4d1[479:448])
);

hadd32 modb4d1b15(
        .c0(carry),
        .x(x[31:0]),
        .y({{24{x[159]}} , x[159:152]}),
        .s(b4d1[511:480])
);

//b4d2

hadd32 modb4d2b0(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[47]}} , x[47:32]}),
        .s(b4d2[31:0])
);

hadd32 modb4d2b1(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[63]}} , x[63:48]}),
        .s(b4d2[63:32])
);

hadd32 modb4d2b2(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[79]}} , x[79:64]}),
        .s(b4d2[95:64])
);

hadd32 modb4d2b3(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[95]}} , x[95:80]}),
        .s(b4d2[127:96])
);
```

```
hadd32 modb4d2b4(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[111]}} , x[111:96]}),
        .s(b4d2[159:128])
);

hadd32 modb4d2b5(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[127]}} , x[127:112]}),
        .s(b4d2[191:160])
);

hadd32 modb4d2b6(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[143]}} , x[143:128]}),
        .s(b4d2[223:192])
);

hadd32 modb4d2b7(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[159]}} , x[159:144]}),
        .s(b4d2[255:224])
);

hadd32 modb4d2b8(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[175]}} , x[175:160]}),
        .s(b4d2[287:256])
);

hadd32 modb4d2b9(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[191]}} , x[191:176]}),
        .s(b4d2[319:288])
);

hadd32 modb4d2b10(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[207]}} , x[207:192]}),
        .s(b4d2[351:320])
);

hadd32 modb4d2b11(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[223]}} , x[223:208]}),
        .s(b4d2[383:352])
);

hadd32 modb4d2b12(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[239]}} , x[239:224]}),
        .s(b4d2[415:384])
);

hadd32 modb4d2b13(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[255]}} , x[255:240]}),
        .s(b4d2[447:416])
);
```

```
hadd32 modb4d2b14(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[271]}} , x[271:256]}),
        .s(b4d2[479:448])
);

hadd32 modb4d2b15(
        .c0(carry),
        .x(x[31:0]),
        .y({{16{x[287]}} , x[287:272]}),
        .s(b4d2[511:480])
);

//b2d1

hadd16 modb2d1b0(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[23]}} , x[23:16]}),
        .s(b2d1[15:0])
);

hadd16 modb2d1b1(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[31]}} , x[31:24]}),
        .s(b2d1[31:16])
);

hadd16 modb2d1b2(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[39]}} , x[39:32]}),
        .s(b2d1[47:32])
);

hadd16 modb2d1b3(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[47]}} , x[47:40]}),
        .s(b2d1[63:48])
);

hadd16 modb2d1b4(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[55]}} , x[55:48]}),
        .s(b2d1[79:64])
);

hadd16 modb2d1b5(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[63]}} , x[63:56]}),
        .s(b2d1[95:80])
);

hadd16 modb2d1b6(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[71]}} , x[71:64]}),
        .s(b2d1[111:96])
);
```

```
hadd16 modb2d1b7(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[79]}} , x[79:72]}),
        .s(b2d1[127:112])
);

hadd16 modb2d1b8(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[87]}} , x[87:80]}),
        .s(b2d1[143:128])
);

hadd16 modb2d1b9(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[95]}} , x[95:88]}),
        .s(b2d1[159:144])
);

hadd16 modb2d1b10(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[103]}} , x[103:96]}),
        .s(b2d1[175:160])
);

hadd16 modb2d1b11(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[111]}} , x[111:104]}),
        .s(b2d1[191:176])
);

hadd16 modb2d1b12(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[119]}} , x[119:112]}),
        .s(b2d1[207:192])
);

hadd16 modb2d1b13(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[127]}} , x[127:120]}),
        .s(b2d1[223:208])
);

hadd16 modb2d1b14(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[135]}} , x[135:128]}),
        .s(b2d1[239:224])
);

hadd16 modb2d1b15(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[143]}} , x[143:136]}),
        .s(b2d1[255:240])
);

hadd16 modb2d1b16(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[151]}} , x[151:144]}),
        .s(b2d1[271:256])
);
```

```
hadd16 modb2d1b17(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[159]}} , x[159:152]}),
        .s(b2d1[287:272])
);

hadd16 modb2d1b18(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[167]}} , x[167:160]}),
        .s(b2d1[303:288])
);

hadd16 modb2d1b19(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[175]}} , x[175:168]}),
        .s(b2d1[319:304])
);

hadd16 modb2d1b20(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[183]}} , x[183:176]}),
        .s(b2d1[335:320])
);

hadd16 modb2d1b21(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[191]}} , x[191:184]}),
        .s(b2d1[351:336])
);

hadd16 modb2d1b22(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[199]}} , x[199:192]}),
        .s(b2d1[367:352])
);

hadd16 modb2d1b23(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[207]}} , x[207:200]}),
        .s(b2d1[383:368])
);

hadd16 modb2d1b24(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[215]}} , x[215:208]}),
        .s(b2d1[399:384])
);

hadd16 modb2d1b25(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[223]}} , x[223:216]}),
        .s(b2d1[415:400])
);

hadd16 modb2d1b26(
        .c0(carry),
        .x(x[15:0]),
        .y({{8{x[231]}} , x[231:224]}),
        .s(b2d1[431:416])
);
```

```
        hadd16 modb2d1b27(
                .c0(carry),
                .x(x[15:0]),
                .y({{8{x[239]}} , x[239:232]}),
                .s(b2d1[447:432])
        );

        hadd16 modb2d1b28(
                .c0(carry),
                .x(x[15:0]),
                .y({{8{x[247]}} , x[247:240]}),
                .s(b2d1[463:448])
        );

        hadd16 modb2d1b29(
                .c0(carry),
                .x(x[15:0]),
                .y({{8{x[255]}} , x[255:248]}),
                .s(b2d1[479:464])
        );

        hadd16 modb2d1b30(
                .c0(carry),
                .x(x[15:0]),
                .y({{8{x[263]}} , x[263:256]}),
                .s(b2d1[495:480])
        );

        hadd16 modb2d1b31(
                .c0(carry),
                .x(x[15:0]),
                .y({{8{x[271]}} , x[271:264]}),
                .s(b2d1[511:496])
        );

        always @(*) begin
          case (encoding)
          4'b0000 :
            valids = 8'b10000000;

          4'b0001 :
            valids = 8'b01000000;

          4'b0010 :
            valids = 8'b00100000;

          4'b0011 :
            valids = 8'b00010000;

          4'b0100 :
            valids = 8'b00001000;

          4'b0101 :
            valids = 8'b00000100;

          4'b0110 :
            valids = 8'b00000010;

          4'b0111 :
            valids = 8'b00000001;

          4'b1111 :
            valids = 8'b00000000;

          endcase
          end

endmodule
```

```
module hadd(
        input wire c0,
        input wire [63:0] x,
        input wire [63:0] y,
        output wire [63:0] s,
        output wire c64
        );

        wire c8, c16, c24, c32, c40, c48, c56;
        wire i00, i01, i02, i03, i04, i05, i06, i07, i08, i09;
        wire i10, i11, i12, i13, i14, i15, i16, i17, i18, i19;
        wire i20, i21, i22, i23, i24, i25, i26, i27, i28, i29;
        wire i30, i31, i32, i33, i34;
        wire g[7:0];
        wire p[7:0];

        //assign c0 = 0;

        and prim00 (i00, p[0], c0);
        or prim01 (c8, g[0], i00);

        and prim02 (i01, p[1], g[0]);
        and prim03 (i02, p[1], p[0], c0);
        or prim04 (c16, g[1], i01, i02);

        and prim05 (i03, p[2], g[1]);
        and prim06 (i04, p[2], p[1], g[0]);
        and prim07 (i05, p[2], p[1], p[0], c0);
        or prim08 (c24, g[2], i03, i04, i05);

        and prim09 (i06, p[3], g[2]);
        and prim10 (i07, p[3], p[2], g[1]);
        and prim11 (i08, p[3], p[2], p[1], g[0]);
        and prim12 (i09, p[3], p[2], p[1], p[0], c0);
        or prim13 (c32, g[3], i06, i07, i08, i09);

        and prim14 (i10, p[4], g[3]);
        and prim15 (i11, p[4], p[3], g[2]);
        and prim16 (i12, p[4], p[3], p[2], g[1]);
        and prim17 (i13, p[4], p[3], p[2], p[1], g[0]);
        and prim18 (i14, p[4], p[3], p[2], p[1], p[0], c0);
        or prim19 (c40, g[4], i10, i11, i12, i13, i14);

        and prim20 (i15, p[5], g[4]);
        and prim21 (i16, p[5], p[4], g[3]);
        and prim22 (i17, p[5], p[4], p[3], g[2]);
        and prim23 (i18, p[5], p[4], p[3], p[2], g[1]);
        and prim24 (i19, p[5], p[4], p[3], p[2], p[1], g[0]);
        and prim25 (i20, p[5], p[4], p[3], p[2], p[1], p[0], c0);
        or prim26 (c48, g[5], i15, i16, i17, i18, i19, i20);

        and prim27 (i21, p[6], g[5]);
        and prim28 (i22, p[6], p[5], g[4]);
        and prim29 (i23, p[6], p[5], p[4], g[3]);
        and prim30 (i24, p[6], p[5], p[4], p[3], g[2]);
        and prim31 (i25, p[6], p[5], p[4], p[3], p[2], g[1]);
        and prim32 (i26, p[6], p[5], p[4], p[3], p[2], p[1], g[0]);
        and prim33 (i27, p[6], p[5], p[4], p[3], p[2], p[1], p[0], c0);
        or prim34 (c56, g[6], i21, i22, i23, i24, i25, i26, i27);

        and prim35 (i28, p[7], g[6]);
        and prim36 (i29, p[7], p[6], g[5]);
        and prim37 (i30, p[7], p[6], p[5], g[4]);
        and prim38 (i31, p[7], p[6], p[5], p[4], g[3]);
        and prim39 (i32, p[7], p[6], p[5], p[4], p[3], g[2]);
        and prim40 (i33, p[7], p[6], p[5], p[4], p[3], p[2], g[1]);
        and prim41 (i34, p[7], p[6], p[5], p[4], p[3], p[2], p[1], g[0]);
        and prim42 (i35, p[7], p[6], p[5], p[4], p[3], p[2], p[1], p[0], c0);
        or prim43 (c64, g[7], i28, i29, i30, i31, i32, i33, i34, i35);
```

```
hadder8 block0(
        .c0(c0),
        .x(x[7:0]),
        .y(y[7:0]),
        .s(s[7:0]),
        .G(g[0]),
        .P(p[0])
);

hadder8 block1(
        .c0(c8),
        .x(x[15:8]),
        .y(y[15:8]),
        .s(s[15:8]),
        .G(g[1]),
        .P(p[1])
);

hadder8 block2(
        .c0(c16),
        .x(x[23:16]),
        .y(y[23:16]),
        .s(s[23:16]),
        .G(g[2]),
        .P(p[2])
);

hadder8 block3(
        .c0(c24),
        .x(x[31:24]),
        .y(y[31:24]),
        .s(s[31:24]),
        .G(g[3]),
        .P(p[3])
);

hadder8 block4(
        .c0(c32),
        .x(x[39:32]),
        .y(y[39:32]),
        .s(s[39:32]),
        .G(g[4]),
        .P(p[4])
);

hadder8 block5(
        .c0(c40),
        .x(x[47:40]),
        .y(y[47:40]),
        .s(s[47:40]),
        .G(g[5]),
        .P(p[5])
);

hadder8 block6(
        .c0(c48),
        .x(x[55:48]),
        .y(y[55:48]),
        .s(s[55:48]),
        .G(g[6]),
        .P(p[6])
);
```

```
        hadder8 block7(
                .c0(c56),
                .x(x[63:56]),
                .y(y[63:56]),
                .s(s[63:56]),
                .G(g[7]),
                .P(p[7])
        );

endmodule

module hadd32(
        input wire c0,
        input wire [31:0] x,
        input wire [31:0] y,
        output wire [31:0] s
        );

        wire c8, c16, c24, c32;
        wire i00, i01, i02, i03, i04, i05, i06, i07, i08, i09;
        wire g[3:0];
        wire p[3:0];

        and prim00 (i00, p[0], c0);
        or prim01 (c8, g[0], i00);

        and prim02 (i01, p[1], g[0]);
        and prim03 (i02, p[1], p[0], c0);
        or prim04 (c16, g[1], i01, i02);

        and prim05 (i03, p[2], g[1]);
        and prim06 (i04, p[2], p[1], g[0]);
        and prim07 (i05, p[2], p[1], p[0], c0);
        or prim08 (c24, g[2], i03, i04, i05);

        and prim09 (i06, p[3], g[2]);
        and prim10 (i07, p[3], p[2], g[1]);
        and prim11 (i08, p[3], p[2], p[1], g[0]);
        and prim12 (i09, p[3], p[2], p[1], p[0], c0);
        or prim13 (c32, g[3], i06, i07, i08, i09);


        hadder8 block0(
                .c0(c0),
                .x(x[7:0]),
                .y(y[7:0]),
                .s(s[7:0]),
                .G(g[0]),
                .P(p[0])
        );

        hadder8 block1(
                .c0(c8),
                .x(x[15:8]),
                .y(y[15:8]),
                .s(s[15:8]),
                .G(g[1]),
                .P(p[1])
        );

        hadder8 block2(
                .c0(c16),
                .x(x[23:16]),
                .y(y[23:16]),
                .s(s[23:16]),
                .G(g[2]),
                .P(p[2])
        );
```

```
        hadder8 block3(
                .c0(c24),
                .x(x[31:24]),
                .y(y[31:24]),
                .s(s[31:24]),
                .G(g[3]),
                .P(p[3])
        );

endmodule

module hadd16(
        input c0,
        input wire [15:0] x,
        input wire [15:0] y,
        output wire [15:0] s
        );

        wire c8, c16;
        wire i00, i01, i02;
        wire g[1:0];
        wire p[1:0];

        and prim00 (i00, p[0], c0);
        or prim01 (c8, g[0], i00);

        and prim02 (i01, p[1], g[0]);
        and prim03 (i02, p[1], p[0], c0);
        or prim04 (c16, g[1], i01, i02);


        hadder8 block0(
                .c0(c0),
                .x(x[7:0]),
                .y(y[7:0]),
                .s(s[7:0]),
                .G(g[0]),
                .P(p[0])
        );

        hadder8 block1(
                .c0(c8),
                .x(x[15:8]),
                .y(y[15:8]),
                .s(s[15:8]),
                .G(g[1]),
                .P(p[1])
        );

endmodule

module hadder8(
        input wire c0,
        input wire [7:0] x,
        input wire [7:0] y,
        output wire [7:0] s,
        output wire G,
        output wire P
        );

        wire [7:0] g, p;
        wire [8:1] c;
        wire i00, i01, i02, i03, i04, i05, i06, i07, i08, i09;
        wire i10, i11, i12, i13, i14, i15, i16, i17, i18, i19;
        wire i20, i21, i22, i23, i24, i25, i26, i27, i28, i29;
        wire i30, i31, i32, i33, i34;

        xor prim00 (s[0], c0, x[0], y[0]);
        and prim01 (g[0], x[0], y[0]);
```

```
or prim02 (p[0], x[0], y[0]);

and prim03 (i00, p[0], c0);
or prim04 (c[1], g[0], i00);

xor prim05 (s[1], c[1], x[1], y[1]);
and prim06 (g[1], x[1], y[1]);
or prim07 (p[1], x[1], y[1]);

and prim08 (i01, p[1], g[0]);
and prim09 (i02, p[1], p[0], c0);
or prim10 (c[2], g[1], i01, i02);

xor prim11 (s[2], c[2], x[2], y[2]);
and prim12 (g[2], x[2], y[2]);
or prim13 (p[2], x[2], y[2]);

and prim14 (i03, p[2], g[1]);
and prim15 (i04, p[2], p[1], g[0]);
and prim16 (i05, p[2], p[1], p[0], c0);
or prim17 (c[3], g[2], i03, i04, i05);

xor prim18 (s[3], c[3], x[3], y[3]);
and prim19 (g[3], x[3], y[3]);
or prim20 (p[3], x[3], y[3]);

and prim21 (i06, p[3], g[2]);
and prim22 (i07, p[3], p[2], g[1]);
and prim23 (i08, p[3], p[2], p[1], g[0]);
and prim24 (i09, p[3], p[2], p[1], p[0], c0);
or prim25 (c[4], g[3], i06, i07, i08, i09);

xor prim26 (s[4], c[4], x[4], y[4]);
and prim27 (g[4], x[4], y[4]);
or prim28 (p[4], x[4], y[4]);

and prim29 (i10, p[4], g[3]);
and prim30 (i11, p[4], p[3], g[2]);
and prim31 (i12, p[4], p[3], p[2], g[1]);
and prim32 (i13, p[4], p[3], p[2], p[1], g[0]);
and prim33 (i14, p[4], p[3], p[2], p[1], p[0], c0);
or prim34 (c[5], g[4], i10, i11, i12, i13, i14);

xor prim35 (s[5], c[5], x[5], y[5]);
and prim36 (g[5], x[5], y[5]);
or prim37 (p[5], x[5], y[5]);

and prim38 (i15, p[5], g[4]);
and prim39 (i16, p[5], p[4], g[3]);
and prim40 (i17, p[5], p[4], p[3], g[2]);
and prim41 (i18, p[5], p[4], p[3], p[2], g[1]);
and prim42 (i19, p[5], p[4], p[3], p[2], p[1], g[0]);
and prim43 (i20, p[5], p[4], p[3], p[2], p[1], p[0], c0);
or prim44 (c[6], g[5], i15, i16, i17, i18, i19, i20);

xor prim45 (s[6], c[6], x[6], y[6]);
and prim46 (g[6], x[6], y[6]);
or prim47 (p[6], x[6], y[6]);

and prim48 (i21, p[6], g[5]);
and prim49 (i22, p[6], p[5], g[4]);
and prim50 (i23, p[6], p[5], p[4], g[3]);
and prim51 (i24, p[6], p[5], p[4], p[3], g[2]);
and prim52 (i25, p[6], p[5], p[4], p[3], p[2], g[1]);
and prim53 (i26, p[6], p[5], p[4], p[3], p[2], p[1], g[0]);
and prim54 (i27, p[6], p[5], p[4], p[3], p[2], p[1], p[0], c0);
or prim55 (c[7], g[6], i21, i22, i23, i24, i25, i26, i27);

xor prim56 (s[7], c[7], x[7], y[7]);
```

```
        and prim57 (g[7], x[7], y[7]);
        or prim58 (p[7], x[7], y[7]);

        and prim59 (i28, p[7], g[6]);
        and prim60 (i29, p[7], p[6], g[5]);
        and prim61 (i30, p[7], p[6], p[5], g[4]);
        and prim62 (i31, p[7], p[6], p[5], p[4], g[3]);
        and prim63 (i32, p[7], p[6], p[5], p[4], p[3], g[2]);
        and prim64 (i33, p[7], p[6], p[5], p[4], p[3], p[2], g[1]);
        and prim65 (i34, p[7], p[6], p[5], p[4], p[3], p[2], p[1], g[0]);

        and prim66 (P, p[7], p[6], p[5], p[4], p[3], p[2], p[1], p[0]);
        or prim67 (G, g[7], i28, i29, i30, i31, i32, i33, i34);

endmodule
```