Executive Summary

Compressed main memory system is a memory system intended to preserve some main memory space. The concept of compressed main memory system is to have the data and instructions stored in main memory as compressed state. Then, whenever data or instructions are requested by (read operation) CPU (Central Processing Unit), data or instructions are decompressed on-the-fly to its original form before being processed by CPU. Similarly, when CPU would like to store data into memory (write operation), data are compressed on-the-fly before entering main memory. The compression and decompression process are done by additional integrated hardware called hardware compressor-decompressor engine. The impact of having compressed main memory system is having an increase in memory bandwidth and cost saving due to increased memory storage capacity for the same memory size.

Our project's main purpose is to build a compressed main memory system that is suitable for ARM Cortex-M0 System-on-chip (SoC). To realize this system, an investigation regarding the feasibility of having a compressed main memory system on Cortex-M0 CPU is required during the literature review. Afterwards, a design of integration mechanism of Cortex-M0 CPU and a hardware compressor-decompressor engine into open source SoC is expected to be constructed in order to implement compressed main memory system. The designed integration mechanism is in form of module called "wrapper" that wrapped the CPU with its interfacing mechanism to the SoC bus. In addition, there is also requirement on evaluating how compressed main memory system improves a system which adopts off-chip RAM (Random Access Memory) as its main memory. This is because we expect positive performance improvement can be gained for implementing compressed main memory on off-chip main memory system.

Methodology used in this project is mainly by conducting behavioral simulations of our designs using Modelsim simulation software. Afterwards, we implemented the design using Xilinx Integrated Simulation Environment onto Xilinx Virtex-5 FPGA chip (in XUPV5 FPGA board) and performed numbers of experiments to analyze the performance of the system. This project involves intensive programming activity by using hardware description language (VHDL) to write the design codes and some portion of C-programming language to write Cortex-M0 test application program for testing the system.

List of achivements:

1. I identified and solved the fundamental problem that was not solved in the previous work regarding the integration of Cortex-M0 CPU into open source GRLIB IP Library SoC (refer to chapter 5 and sub-chapter 6.2)

2. I build my own working wrapper design called CM0_wrapper which integrates Cortex-M0 CPU into GRLIB IP Library SoC in purpose of building build basic transfer Cortex-M0 SoC. CM0_wrapper allows Cortex-M0 to perform stack operations which could not be performed by previous work. (refer to sub-chapter 6.3)

3. I build my own working compressed main memory system wrapper design called CM0X_wrapper which performs on-the-fly compression and decompression. (refer to chapter 6.4)

4. I have evaluated the benefit of having compressed main memory system on off-chip main memory based system. (refer to sub-chapter 7.2)

5. I have implemented and run all my wrapper designs including compressed main memory system in Xilinx Virtex 5 FPGA board. This proves that my design does not generate latches, in which this indicates a good design implementation on FPGA. (refer to sub-chapter 7.4).

Table of Contents

List of Abbreviations

AHB	Advanced High Performance Bus
AMBA	Advanced Microcontroller Bus Architecture
ARM	Advanced RISC Machines
ASIC	Application Specific Integrated Circuit
CAM	Content Addressable Memory
CBAT	Compressed Block Address Table
COMDEC	Compressor-Decompressor
CCRP	Code Compression RISC Processor
CCTP	Code Compression Thumb Processor
CLB^1	Cache Line Address LookAside Buffer
CLB^2	Configurable Logic Block
DBB	Decompression Block Buffer
DDR2	Dual Data Rate 2
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IOB	Input Output Buffer
IP	Intellectual Property
JTAG	Join Test Action Group
LED	Light Emitting Diode
LAT	Line Address Table
LUT	Look-Up-Table
MTF	Move-To-Front
MXT	Memory Expansion Technology
ODA	Out of Date Adaptation
RISC	Reduced Instructions Set Computing
SAMC	Semi Adaptive Markov Compression
SCMS	Selective Compressed Memory System
SOC	System-On-Chip
SODIMM	Small Outline Dual In-Line Memory Module
SRAM	Static Random-Access-Memory
STT	Sector Translation Table
UBSR	Uncompressed Block Size Register
VLIW	Very Long Instructions Word
VHDL	Very-High-Speed-Intergrated-Circuit Hardware Description Language

Chapter 1: Motivation

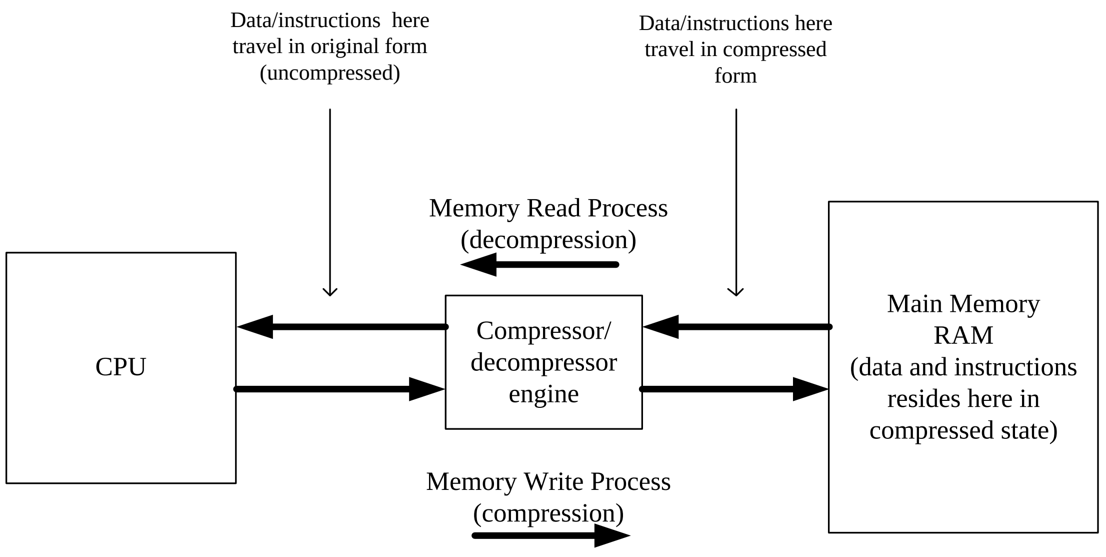
Computer technology throughout history has shown a remarkable evolution since the introduction of microprocessors in the late 1970s. As predicted by Gordon Moore, chip density has been increasing by factor of two in every two years and semiconductor industries have been using this prediction as a guiding principle to stay advanced in meeting market demand (or even inventing new one). While CPU (Central Processing Unit) capability keep improving annually (either by fabrication technology or parallelism), memory which is the second most important to overall computer performance is not keeping the same pace with CPU in term of its speed.

The pressure on the memory system has been getting more tremendous since the emergence of multi-core processors. Single core processor clock frequency has met its limit to 3.4GHz [1] and parallelism is the answer to overcome it. The invention of multi-core processor also means a further increase in the demand of more memory bandwidth with lower latency to avoid low level performance bottleneck. Multi-core processors also open more possibilities for software industry to develop highly sophisticated applications, which also resulting to the increase demands in memory size and speed. Nevertheless, memory speed technology is harder to be pushed than processor speed and resulting to unbearable higher price for same size but faster memory speed. Because of these reason, increasing number of computer applications are now limited by memory performance.

Computer architects come up with memory hierarchy as an economical solution to deal with memory bus limitations. Memory hierarchy is broken down into several levels, in which each level is faster, but smaller in size and more expensive per byte than the next lower level [2]. Memory hierarchy varies depending on the purpose of the system. The latest common hierarchy for general purpose system is known to be (from fastest to slowest): registers in CPU, two levels of caches, main memory, and mechanical disk [3].

Unlike the general purpose system, memory hierarchy for embedded system is known to have the most diverse design specifications with varying constraints. The constraints come from primary goal of embedded system which is to meet acceptable performance at minimum price for a specific target application [2]. To meet the goal, embedded system design is often focused to minimize power consumption and memory size requirement. In this case, implementing optimization in memory has possibility to give impacts in several areas including power consumption. The most sought memory optimization form is to improve code density that often not only reduces memory cost and power consumption but also increases speed of the system.

There are few approaches to deal with code density. First approach is to manually optimize assembly code by hand. This approach is time consuming and requires lots of effort from the programmers. In average, this approach only attains 10-20% denser code than that achieved by good compiler [4]. Second approach is to improve the compiler itself, in which the compaction result will always be lesser than hand coded approach. Third approach is to construct a new architecture that has compacted instruction sets. The most well-known and successful example is ARM processor with THUMB instruction set. Lastly, main memory compression is another approach that is able to be combined with previous approaches to further improving code density.



Data/instructions here travel in original form (uncompressed)
Data/instructions here travel in compressed form
Memory Read Process (decompression)
CPU
Compressor/ decompressor engine
Main Memory RAM (data and instructions resides here in compressed state)
Memory Write Process (compression)

Figure 1.0: General idea of compressed main memory system

In brief, the main idea of compressed main memory system is to allow content of the main memory (instructions and data) stored in compressed state, thus, preserving more space in main memory. While this condition provides several obvious advantages (that will be described throughout this research review), an additional mechanism to decompress and compress the memory content on-the-fly (during memory access by CPU) is required to be integrated in the system. This way makes the system to decompress the memory content only when it is needed by CPU and compress the data when CPU writes back to memory. This idea is well illustrated in the figure 1.0, in which we can observe that compressor-decompressor engine exists as additional mechanism between CPU and main memory.

Clearly, additional works done by compressor-decompressor engine may affect the system performance in term of memory access time. Therefore, on-the-fly compression/decompression mechanism is expected to be as fast and efficient as possible in order to minimize the overhead, otherwise, the memory access speed in this system will be inferior to the normal system.

One advantage of having compressed main memory system besides saving memory space is that it may increases memory bandwidth, which means there are more data travelling in the bus at a time since data is in compressed state. This advantage is more apparent when the system is having off-chip memory as its main memory. This is because off-chip memory such as is obviously slower than higher memory hierarchy like cache or on-chip RAM. However, most typical system uses off-chip RAM such as DDR SDRAM (Dual Data Rate Synchronous Dynamic RAM) for its main memory in purpose of saving cost per byte while sacrificing system performance. Hence, compressed main memory system is expected to show positive impact on improving system performance which utilizes off-chip RAM.

Chapter 2: Aims & Objectives

This project is categorized as type II research project. However, considerable amount of hardware implementation which resembles some portion of type I implementation project also exist.

Thereby***, this project has two aims as follows***:

1. To investigate the feasibility of introducing Compressed Main Memory System to ARM Cortex-M0 processor in System-on-Chip (SoC) environment.

2. To realize the hardware implementation of the investigated system for research purpose.

The first aim is mainly intended to gain an in-depth knowledge regarding compressed main memory system for embedded system processor (related to ARM processor) and study relevant projects that have been done before. The result of synthesizing pros and cons of related projects is useful for determining the suitable implementation for our proposed project.

The second aim is to deal with the implementation part whereby compressed main memory system and ARM Cortex-M0 processor are integrated into Aeroflex Gaisler GRLIB IP library SoC [5], which then be implemented and tested in Xilinx Virtex 5 XUPV5-LX110T FPGA board [6]. The compressed main memory system we are developing relies on state-of-the-art hardware compressor-decompressor engine called X-MatchPROVW[7] developed by researcher in University of Bristol.

Having our two aims well defined, we have separated the objectives for each aim that are described as below.

First aim-objectives:

1. To find the measurement parameter that defines a successful compressed main memory system.

2. To analyze the optimization done by relevant projects

3. To synthesize pros and cons of the related projects in order to come up with implementation method that suits our implementation environment.

Second aim-objectives:

1. To build basic transfer Cortex-M0 SoC based on GRLIB IP Library SoC.
   This SoC serves as reference of a basic Cortex-M0 SoC to evaluate our compressed main memory system. Modification of GRLIB SoC is needed such as replacing Leon3 CPU with Cortex-M0, memory modification etc.

2. To build compressed main memory system by integrating Cortex-M0 CPU and X-MatchPROVW compression-decompression engine into GRLIB SoC.

3. To add hardware optimization for improving the performance such as the addition of decompression buffer.

4. To investigate the benefit of having compressed main memory system on off-chip main memory based system.

Chapter 3: Literature Review

This section contains comprehensive literature reviews of hardware main memory compression mainly for embedded system. Materials taken for this literature reviews are mostly based on published paper and journals because there is no published text book in specific relation to hardware main memory compression yet.

The organization of the literature review is brought from the basic of main memory compression to specific reviews related to optimization of hardware memory compression. We are being slightly technically specific for presenting our literature review since it is the nature of the research in computer hardware area. Additionally, sometime we need to elaborate completely the whole system before we reach to the main point to support the understanding of the audience. This is because hardware implementations are often specific and proprietary.

3.1 Overview of Compressed Main Memory System

To begin with, this section aims to give the basic understanding of Compressed Main Memory System. Memory compression is one of many optimizations can be done in memory hierarchy that may require less effort and cost comparing to other optimizations depending on the approach taken. The basic idea of main memory compression is to save some main memory space (Random Access Memory, RAM) by compression, with expectation to increase main memory storage and decreases bus traffic [8]. Obviously, more memory space in RAM gives high chance for cost reduction, and reduced bus traffic may decrease power consumption if overhead from the modification is minimized. In addition, it may also improve memory access time since less data travels in the bus. Although the idea sounds trivial, the realizations of the system have been proved to face many challenges and design trade-off, which will be pointed out in this literature review.

The basic of compressing information in computer system is to find information redundancy, which is normally found in consecutive streams of address, predicted data values in program loops, and instructions with repeating sequences [9]. By having the awareness of this concept, there are three types of information redundancy derived from its source, which are address redundancy, instruction redundancy and data redundancy.

Address redundancy consists of instructions addresses and data addresses, whereby temporal and spatial locality exist [9]. Temporal locality means that the same memory location may be accessed again in the near future whereas spatial locality means that nearby memory location with respect to the current accessed memory location may be accessed in the near future [10]. For Instruction addresses, locality exist because they are highly to be sequential except when branch instructions are executed, albeit branch target address is typically near to the branch instruction address itself. For data addresses, data arrays scanned in the loop are the reasons for those two localities. Redundancy in instructions occurs by reason of instruction sequences, register operands, opcodes and immediate values are repeated. These repetitions are common due to basic characteristic of a program for having branches, loops, etc. Lastly, data redundancy frequently happens due to temporal and spatial locality when processor accesses data from memory [9].

Compressing the memory of a computer system can be done either software-based or hardware based. Software based compression would require intensive software modification in operating system level and has a broad flexibility in enhancing the compression algorithm. The first idea of software-based main memory compression emerged in early 1990s by Wilson who proposed a new level of memory hierarchy to store memory pages in compressed form [11]. It was later found that software based memory compression can be classified into adaptive and static [12] .The adaptive approach analyzes applications which memory can be most optimized to boost performance while static approach relies on mathematical model and less effective compared to adaptive one.

The hardware-based memory compression is more favored although it is a harder approach compared to software-based. Hardware based approach was found to excite more performance improvement up to several folds compared to software based which is only up to the factor of two[13]. Commonly, hardware memory compression requires hardware modification based on its memory hierarchy or its instructions set. By having the hardware modified, it surely opens wider range of implementation, which can even be applied to low cost embedded system in order to achieve low-cost and low-power embedded system.

For modification done on its instruction set, big chip companies such as ARM and MIPS comes up with compacted ISA (Instruction Set Architecture) [14]. ARM with its well-known THUMB instruction set, has claimed that there is 30% code density improvement from original ARM instructions [4]. In addition, THUMB instructions combined with utilization of L1 caches further increased the memory access speed by reason of cache line holds twice number of THUMB instructions, which then increases cache hit rate[14]. Albeit compaction of ISA proves to ensure maximum optimization, it requires construction of new architecture that contributes immense effort and cost to realize it. In addition, competition in this area is directly with giant company such as ARM.

For modification done in memory hierarchy, a common approach is to store instructions and data in compressed state inside RAM and decompresses them when they are needed to be accessed by CPU with or without cache. There are three working principles for compressed main memory system that we could synthesize from various papers:

1. Data and instructions are firstly compressed in off-the-fly manner (by compression tool) and stored in main memory.
2. When data or instructions are needed by CPU, they are decompressed on-the-fly just before being processed by CPU.
3. When CPU is going to write data to the main memory, data is compressed on-the-fly just before entering main memory.(please refer to figure 1.0 in section 1 to support the understanding of the concept)

There have been many attempts to realize this system, in which in the early phase, the implementation is developed for only compressing instructions (called “code compression")[14,15]. This way implies that only decompression process needs to be taken care on-the-fly (or sometimes we use term “online”). For the method which includes both data and instructions to be compressed, not only decompression but also compression process is required to be on-the-fly, and resulting increased complexity. The decision in choosing between instructions-only or both instructions-data compression certainly depends on the intent of the application. For application that frequently writes data to memory such as image or video processing, having both instructions and data compressed brings substantial performance increment. Additionally, the implementation of code compression can be combined with compacted ISA to achieve more optimal performance (provided that decompression/compression overhead is maintained to lowest level).

Many existing project related to memory compression only concern about getting more memory space by achieving high compression ratio. While high compression ratio results in greater space savings, it does not guarantee significant speed improvement. Otherwise, it would degrade the processing speed (in term of memory access time) caused by decompression/compression process’s overhead. This overhead is the main challenge that existing projects related to hardware memory compression system need to overcome. A key factor to overcome this problem is to employ on-the-fly lossless compression-decompression engine which employs compression algorithm as fast as possible. Howsoever, the integration of compression engine to the system is certainly not trivial and exhibited many challenges in order to integrate it seamlessly without additional overhead from the compression engine.

3.2 Compression-Decompression Engine and Algorithm

One of the main challenges in compressed main memory system is to reduce compression-decompression overhead to maximize the advantage of having the main memory compressed. A key factor (although not the only factor) to achieve it is to utilize compression engine with sophisticated algorithm that is fast enough (high throughput) while maintaining optimum compression ratio. In our project, we have chosen a fast decompression engine called X-MatchPROVWto be integrated in our SoC. Since X-MatchPROVWadopts dictionary compression algorithm, it is worth for us to understand the nature of this algorithm and compare it with other compression algorithms to see why dictionary algorithm fits in hardware main memory compression system.

This section is divided into two subsections. The first part discusses few compression algorithms commonly used by compressed main memory system and comparing them with dictionary compression algorithm. The second part discusses the compression engine we are using in this project regarding its algorithm and performance.

3.2.1 Dictionary Compression Algorithm

Compression algorithms are inherently based on encoding technique to convert portion of data into smaller forms. Compression in term of encoding technique falls into two main categories which are statistical compression and dictionary compression [16].

Statistical compression is based on probability of frequently occurring characters, in which these characters are encoded using shorter code while reduces the overall length of resulting compression [17]. Two types of statistical compression which are byte-based and arithmetic-based encoding techniques will be briefly reviewed on the following.

Huffman byte-based encoding is commonly considered to be the simplest one and it was adopted by CCRP (Compressed Code RISC Processor [15]) with slight modifications. The key idea in Huffman encoding is that the algorithm tries to generate the shortest codes that represent most often occurring elements [18]. These short codes are created by constructing histogram of frequently-occurred input data. CCRP modified Huffman algorithm (called Bounded Huffman) to meet reasonable decompression time since the nature of original Huffman encoding is only to achieve greater compression ratio with less effort. Consequently, 0.73 compression ratio was achieved by CCRP [15].

Arithmetic-based encoding algorithm is frequently used by state-of-the art data compression algorithm such as Prediction by Partial Match (PPM) and Dynamic Markov Compression (DMC) [16]. The key idea of arithmetic encoding is to subdivide the current interval into subintervals by referencing to probability of the next input [19]. This encoding technique emphasizes its high compression ratio and flexibility. It is flexible because it can be combined with other suitable model such as Semi Adaptive Markov Compression (SAMC), which combined arithmetic encoding with pre-calculated Markov model [16]. Lekatsas et al knew that arithmetic encoding suffers to slowness. Therefore, he uses arithmetic encoding as look-up table technique “only” for starting point, which then he modified it to allow random access and multi-bit decoding by creating decoding table. With this approach, SAMC managed to achieve better compression ratio up to 17% compared to Huffman encoding. However, SAMC still did not make substantial improvement in term of execution speed due to the nature of arithmetic encoding.

Dictionary compression relies on the dictionary-based encoding that simply searches for common instructions sequence, assign them with an index that points to dictionary entry and put them into an entry in dictionary as illustrated in figure 3.2-1 .The dictionary itself contains frequently occurring instructions and the indexes are chosen to be smaller in size compared to the encoded instructions [20].

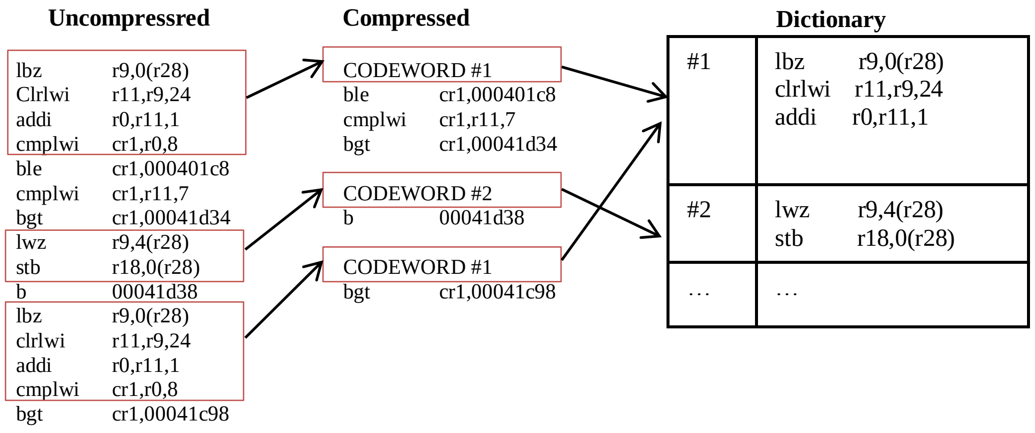


Figure 3.2-1: Illustration of dictionary compression [17]

Dictionary-based algorithm is widely adopted by state-of-art compression technique such as Ziv-Lempel LZ77 [20] and X-Match algorithm [7]. In the field of hardware memory compression, projects adopting dictionary based compression engine always come up to be the most optimal main memory compression system such as SCMS [21] (uses modified X-Match) and MXT by IBM[22] (uses Lempel-Ziv). SCMS provides 0.5 compression ratio with 20% increased memory access time while MXT also provides 0.5 compression ratio with only 1.3% speed improvement [21, 23].

Based on the facts we have delivered, clearly dictionary based algorithm outperforms statistical approach especially in hardware main memory compression area. It provides the best of both worlds between optimum compression ratio and fast execution. Therefore, we are confident that using dictionary based compression engine such as X-MatchPROVW is a key advantage for us to achieve highest possible result.

3.2.2 X-MatchPROVW Compression-Decompression Engine

This section elaborates the compression-decompression engine used as the key hardware in our proposed project. X-MatchPROVWis a lossless dictionary based hardware compression-decompression engine capable of delivering 1.092 GByte/s throughput when it is clocked in 273 MHz. Additionally, this compression engine is able to achieve compression ratio better than 0.5 for block size over 4 kilo bytes. The overwhelming performance can be achieved because X-MatchPROVWuses parallel architecture and 5-stage pipelines. These two optimizations resulting to quadruple compression throughput for same clock frequency and minimum latencies. By the reason of those facts, X-MatchPROVWhas overcome the main challenge of compressed main memory system, in which to employ compression-decompression engine that maintain optimum compression ratio with minimum latency [7,24].

X-MatchPROVW[7] is the new successor of previously made non-parsing X-MatchPRO[24]. “VW” as the last two letters stands for “variable width”, in which its dictionary algorithm is able to parse input to natural words in variable lengths (instead of fixed 4 bytes in previous X-MatchPRO). The newer version was created because the non-parsing X-MatchPRO does not perform well on compressing human readable data (ie. Text, html). Although the VW version is far superior to the non-parsing version, they are based on X-Match algorithm principle and thus similarities exist between both. Since our concern regarding compressor-decompressor engine is only to understand its main working principle, we are going to explain the non-parsing version of X-MatchPRO algorithm instead of X-MatchPROVWalgorithm as it is less complex and more practical to be presented in this research review.

The basic working principle of X-Match dictionary algorithm is to match current input tuple (fixed-width 4 bytes data element) from data stream with an entry in dictionary, wherein the dictionary contains previous tuples inputted to the engine. If current tuple and an entry happen to be fully matched, a 'full match' occurs. If at least two bytes in current tuple matched with those in an entry, a 'partial match' occurs with unmatched bytes transmitted as it is. If nothing is matched, miss bit of '1' followed by original tuple are transmitted. The dictionary employs Move to Front [MTF] strategy to enable locality for the input tuples. This strategy places current tuple on the front of the dictionary and other tuples are moved down by one location. When the dictionary is full, tuple at the last location is simply discarded [24,25].

Figure 3.2-2 shows compression ratio of varying block sizes on memory done by non-parsing X-MatchPRO, X-MatchPROVW, ALDC by IBM, LZS by HiFn and DCLZ by Hewlett-Packard. Since memory content is not human readable data, we can see non-parsing X-MatchPRO and parsing X-MatchPRO have little discrepancy in compression ratio on the memory. However, both versions of X-MatchPRO algorithms are capable to be compared with other commercial algorithms, which are deemed to be state-of-the-art compression technology available at the moment.

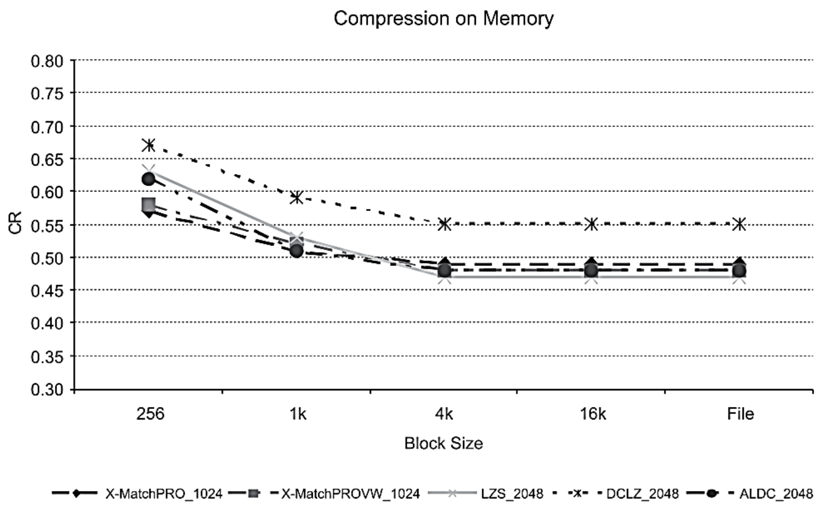


Figure 3.2-2: Compression ratio against range of compression block sizes on memory for various algorithms [7]

X-MatchPROVWhardware architecture is far more complex compared to non-parsing X-MatchPRO. To include complete explanation of X-MatchPROVWhardware operation is not practical in this research review. Therefore, since X-MatchPROVWis based on non-parsing X-MatchPRO and both are having some similarities, we will use non-parsing X-MatchPRO to elaborate its hardware operation principle.

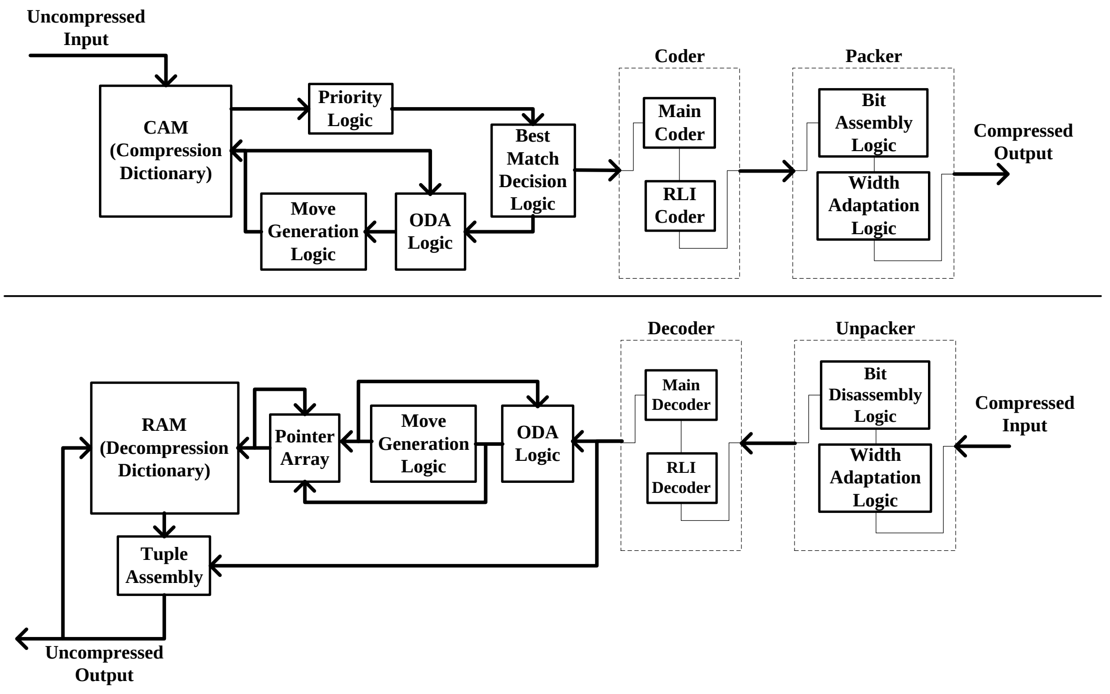


Figure 3.2-3: Non-Parsing X-MatchPRO Compression and Decompression hardware operation flow [23]

The hardware of non-parsing X-MatchPRO basically contains compressor and decompressor mechanism, in which overview of the architecture is described as follows (refer to figure 3.2-3):

1. Compression hardware architecture
   As can be observed from the top left hand corner of the graph, compression process starts from inputting the uncompressed tuple to the main dictionary engine. Dictionary in X-Match compression hardware adopts “Content Addressable Memory (CAM)”[28] array that input a tuple and output the match address of the tuple by having the CAM to be selectively shifted. As a result, MTF strategy can be executed in a single clock cycle. In order to find a match in CAM, “Best Match Decision Logic” is used to find the best match location and match type. Match location is binary code related to location of the matched bytes, and match type indicates which bytes of input tuple have matched. These location and type are essential to produce shift pattern (adaptation process) for modifying CAM (with “Move Generation Logic” and “Out of Date Adaptation (ODA) Logic”) and to be used for coding process in coder. Coder contains “Main coder” and “RLI coder”. Main coder consists of Phased Binary Coder (to process match location) and Huffman-based Coder (to process match type). RLI coder is to detect 32-bit repeating pattern. Lastly, necessary bytes resulted from those coder are packed in a “Packer” consisting “Bit Assembly Logic” and “Width Adaptation Logic”[24].

2. Decompression hardware architecture
   Contrast to the compression, decompression starts from “Unpacker”, in which compressed data are unpacked by “Width Adaptation Logic” and “Bit Disassembly Logic. Afterwards, compressed data is decoded in “Decoder” to construct uncompressed data. Then, data is sent to Adaptation process that is maintained in the same way as for compression but with different dictionary.
   Dictionary in X-Match decompression hardware is based on RAM which stores previously seen data. Content of RAM during decompression and CAM during compression must be the same. This is achieved by Pointer Array Logic that maps CAM’s content addresses to RAM for decompression [24].

3.3 Managing Compressed Block

The most problematic part in integrating online compressor-decompressor engine to a memory system is when dealing with control transfer functions such as conditional or unconditional branches. The problem comes when branch instruction is executed to jump to instruction that is in compressed form because the starting address of the block containing target instruction is altered due to compression. The altered target address is caused by the nature of compression, in which the resulting compressed blocks are in variable length.

Before we further elaborate on how to tackle branching problem in compressed main memory system. It is necessary to introduce the terminology of "Block" (which is not to be confused with basic block). A block here is defined as stream of bytes that can be a word (or group of words) of instructions or data. Hence, when instructions or data are to be compressed, they are taken as a word (or group of words) in form of uncompressed block, which then be compressed and stored in main memory as compressed block (containing single or multiple instructions)[9,16,17]. Compressing data and instructions in form of block is preferred because:

1. Blocks are capable to be accessed randomly in RAM. Hence, blocks are easy to be read or written and it does not need sequential access.
2. Blocks can be configured to be byte aligned.
3. Blocks are relatively easier to be managed for solving branch problem since we can assign address to each block.

Point number three is exactly what we are going to discuss in the rest of this section. Because each block can be assigned an address, it has become a typical approach for compressed main memory system to take advantage of block's address in solving branching problem. The typical approach is to have some kind of mapping table (some also call it translation table) which maps the addresses of compressed block to addresses of original uncompressed block. In other words, this mapping table performs address translation of compressed block so that those block addresses can be understood like the addresses belong to original block. This method is adopted by several implementation designs, in which one of them is called "Line Address Table" (LAT) by Wolfe and Chanin [15]. Another different approach is by Lefurgy et al which avoids the use of mapping table and use fixed size compressed block instead.

In order to further understand how block's address can be utilized to solve branching problem, we are going to discuss three different implementations of mapping table and one implementation of fixed size block through the next four subsections.

3.3.1 LAT in Compressed Code RISC Processor

Compressed main memory for embedded system has long been existed since early 1990s. Wolfe and Chanin came up with Compressed Code RISC Processor (CCRP), in which compressed instructions in instruction memory are decompressed for cache refill when cache misses occur. The main challenge faced by this system is to maintain execution of control transfer functions. To overcome this problem, Wolfe and Chanin developed mapping table called Line address Table (LAT)[15].

LAT contains entries to map compressed instruction block addresses to uncompressed instruction block addresses. In other words, when cache misses occur, cache refill engine uses LAT to locate instructions block that is going to be decompressed. LAT is stored along with program instructions in instruction memory (as can be seen in figure 3.3-1).

As can be observed from figure 3.3-2, each entry in LAT is eight bytes long holding information of eight compressed adjacent instruction blocks. The base address is 3 bytes pointer to physical address of the first compressed instruction block L0. Following to that, the subsequent each 5 bits represents the length of the "next" compressed instruction block. This length value ranges from 1 to 31. If the length is 0, it represents the length of the next instruction block that is uncompressed. Thus, each of compressed address can be found by adding the length to the base address. Computing the compressed block address in LAT is done by Cache Line Address LookAside Buffer (CLB) to retrieve the starting address of any compressed block. Since starting address of compressed address is known, then branch instruction problem is solved.

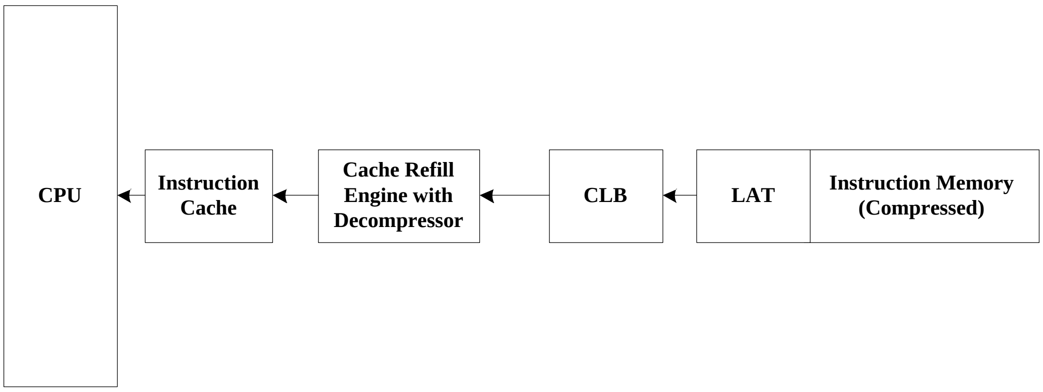

CPU
Instruction Cache
Cache Refill Engine with Decompressor
CLB
LAT
Instruction Memory (Compressed)

Figure 3.3-1: CCRP Memory Organization [4]

LAT Entry

Base Address	L0	L1	L2	L3	L4	L5	L6	L7
	5 bits	5 bits	5 bits	5 bits	5 bits	5 bits	5 bits	5 bits

Figure 3.3-2: LAT Entry in CCRP [4]

CLB actually contains Address Computation Unit and Content Addressable Memory (CAM). Thus, CLB is not only used for computing the compressed address but also act as a storage for storing some computed addresses from LAT. This makes CLB acts like a cache for LAT entries, thus, it provides another memory hierarchy to the cache refill engine. To illustrate this, consider when cache miss occurs, CLB is searched by cache refill engine, if the requested address found in CLB (as a result of storing computed address from LAT) then there is not increased cache penalty imposed. However, if requested address by cache refill engine could not be found in CLB, then only CLB perform address computation from LAT and additional cache miss penalty occurs. With CLB optimization, cache miss penalty is lesser than if LAT has direct connection with cache refill engine. However, the total cache miss penalty is still higher than that without LAT and CLB.

The addition of LAT and CLB does not come without additional cost. Besides the increased cache miss penalty as described previously, it also contains overhead due to the increased size of compressed code. Wolfe and Chanin stated that it takes eight LAT bytes for every 256 original compressed instructions bytes, which resulting additional overhead of 3.125% from original compressed code size [15].

Since the advent of CCRP, stream lines of main memory compression projects emerged to improve the use of LAT for their application. Lin et al modified LAT to only contain branch target addresses and results to much reduced size version of LAT in their VLIW code compression system [30].

3.3.2 CBAT in Compressed Code THUMB Processor

Another project that utilizes LAT-like table is done by Xu and Jones with project called Code Compression THUMB Processor (CCTP). CCTP is worth to mention in our review since CCTP is simply a modified CCRP with goal to support ARM processor employing THUMB instruction set. To accommodate ARM/THUMB instruction set, instruction block size to be compressed must be increased from 32 KB (as in original CCRP) to block size that gives optimum compression ratio for THUMB code, which is 256 KB in this case[14].

CCTP actually employs the same LAT which was renamed as Compressed Block Address Table (CBAT). Nevertheless, CCTP replaces CLB with Decompression Block Buffer (DBB). The difference between CLB and DBB is that DBB is located in Data Memory (RAM) with intention to reduce complexity previously caused by CLB.

Albeit the complexity is reduced, CCTP suffers from slow timing performance. Firstly, we think the main reason causing slow DBB access time is because of the utilization of RAM as a place to put DBB. Since RAM is inherently slower than cache used previously for CLB in CCRP, DBB access time is clearly higher than CLB access time. Secondly, ARM processor architecture uses virtual addresses instead of physical addresses to fetch instruction to L1 cache [14]. Thus, DBB must be able to contain mapped virtual address space that holds compressed instructions. Because this system has DBB that is much smaller in size compared to virtual address space, DBB has to possess many memory pages. These page tables are frequently accessed to map DBB pages correctly which causes performance speed degradation.

In addition, CCTP also faced issue related to decompression of instruction memory, in which ARM THUMB processor also stores global variables and literal data in instruction memory. In our point of view, this resulting L1 Data Cache also needs to be connected to decompression engine as shown in Figure 3.3-3. Through CCTP, we have learned some important things that need to be taken care when we implement our compressed main memory system with ARM processor.

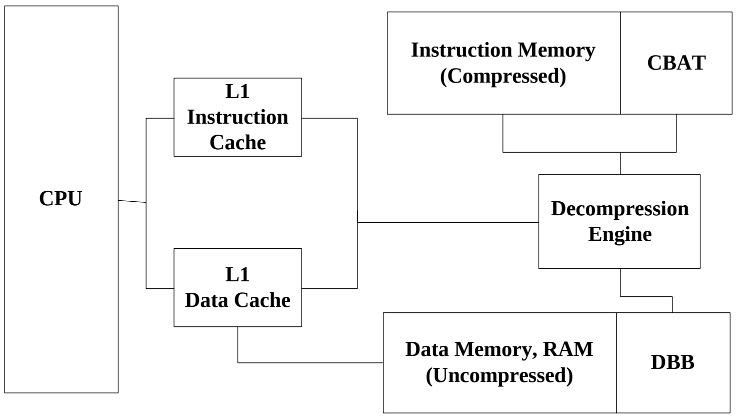


Figure 3.3-3: CCTP Memory Organization [3]

3.3.3 STT in Memory eXpansion Technology

The most well-known commercial server-class main memory compression system is IBM’s creation called Memory eXpansion Technology (MXT). MXT’s first application debuted in “Pinnacle”, a single chip memory controller compatible with Intel Pentium III/Xeon [22]. Examining mapping table of MXT in this review is essential since it is a robust hardware compressed main memory system that has been commercially launched. Before we explain the mapping table used in MXT, it is useful to first understand its memory organization and working principle of MXT.

MXT’s memory hierarchy consist of main memory L4, shared L3 cache with compressed memory controller and L1/L2 caches. Data is stored in Main memory L4 (8 GB available space and up to 16GB due to compression) in compressed form. The L3 cache is shared by all processor and IO devices, in which it stores uncompressed data and hides latency for main memory access. Controller performs compression and decompression between main memory and L3 caches. L1/L2 cache performs normal cache operations to each processor.

The controller employs modified version of Lempel-Ziv “Adaptive Dictionary” algorithm that compresses blocks and sub blocks in parallel manner [27]. The working principle of this compression engine is that the decoded data block is divided into number of equal sub blocks, in which each of it is handled by single compression engine. Therefore, MXT controller contains four compression engines each operates on 256 Bytes at 1Byte/cycle [22]. The controller also performs address conversions (real address for the main memory to physical address for caches immediate access) by referencing to Sector Translation Table (STT).

STT is a form of mapping table located in main memory. The unique part of STT is that STT is not only mapping addresses but also perform address size compression. Each entry in STT contains real address of 1 kilo bytes uncompressed block from L3 cache line. This real address is mapped to four physical sector addresses, in which each of them points to 256 bytes sector in main memory (a quarter of 1 kilo bytes). This 256 bytes sector consist of 16 entries each containing 16 bytes. Hence, this resulting to possible compression ratio of 64 to 1 because 1 kilo bytes is essentially mapped to 16 bytes. This 64 to 1 compressibility only occurs when block of data is compressible to be less than 121 bits, which then be stored entirely in STT entry as “trivial line” format and replaces the four address pointer (256 bytes sector) instead of being stored in physical memory.

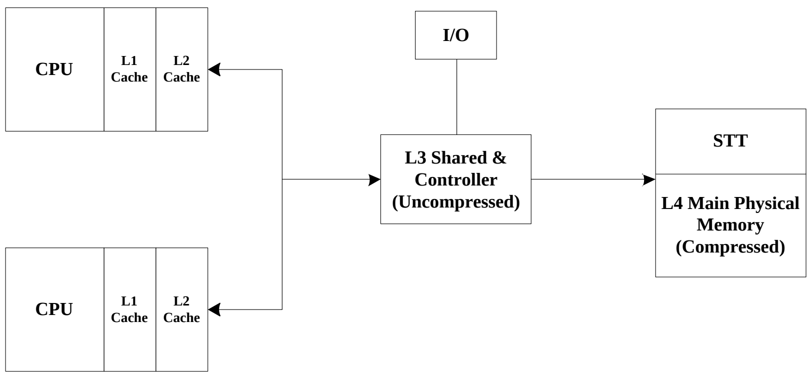

I/O
CPU
L1 Cache
L2 Cache
STT
L3 Shared & Controller (Uncompressed)
L4 Main Physical Memory (Compressed)
CPU
L1 Cache
L2 Cache

Figure 3.3-4: MXT Memory Organization [22].

Among three previously discussed mapping table methods (LAT, CBAT and STT), clearly STT is the most sophisticated one as it reduces the overhead caused by having translation table in the main memory. Efficient implementation of translation table like STT resulting more main memory space can be preserved and may improves memory access time.

Having to know that MXT is deemed to be a successful implementation of compressed main memory system, one could expect its performance would be superior to standard memory system. Abali et al performed experiment for analyzing MXT performance using SPECint2000 benchmark designed by SPEC consortium [23]. The benchmark was run to find the difference between standard system and MXT. Result of the benchmark shows that MXT successfully compresses main memory content with 2:1 compression ratio in average. Unfortunately, MXT insignificantly improves the system speed only by 1.3% compared to standard system yet at least there is no performance degradation caused by overhead of compression engine.

3.3.4 Fixed-Length Compressed Block.

All previous related projects we have described typically aim for high compression ratio while sacrificing performance (except for MXT). This is due to the utilization of mapping table that increases memory access time caused by address computation to translate the compressed block's address. It is essential to stress again that mapping table is required since the resulting compression block is in variable length.

There are two other problems may occur due to variable-length compressed block as pointed by Lee et al [21]. Those problems occur when block of data is decompressed to be read by CPU and then compressed back to be written to memory again. First problem is that, the resulting compressed block sometime can be longer than original block. This is called as "Data Expansion". Second, the new expanded block is no longer can be stored in the same memory location and this is called "Fat write" problem.

In contrast, if we fix the length of the resulting compressed block, there will be less effort to translate the compressed address and two major problems pointed previously are alleviated. This approach is called Fixed-Length Compressed Block.

Lefurgy et al [28] implemented fixed-length compressed block in his dictionary compression engine in order to reduce address translation time. He fixed the size of compressed block to be only half of the original size (as can be observed from figure 3.3-5). Therefore, simpler address calculation can be done and mapping table is not required.

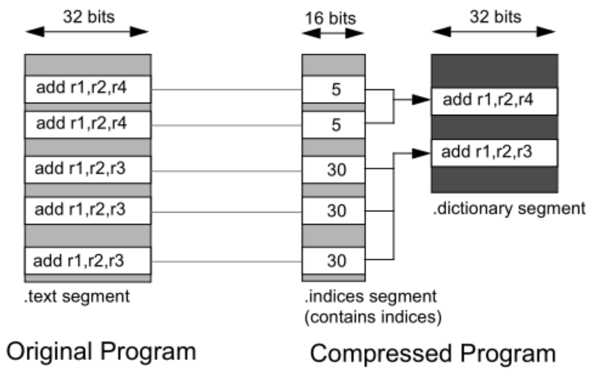


Figure 3.3-5: Fixed-length compressed block translation [28].

However, for us to attempt fixed-length compressed block method, the modification for compression engine might be required since Lefurgy et al [28] and Lee et al[21] developed their own compression engine. Modification in compression engine may require huge effort and it is not our main concern in this project to deal with the algorithm of the compression engine. In addition, although this method has advantage in reducing the compression-decompression overhead, it still has disadvantage of having less compression ratio than those which uses mapping table. It is also obvious to think that variable-size block resulting more compact size than fixed-size block.

3.4 Optimization with Decompression Buffer

We have understood that compression-decompression overhead is the main factor that suppresses the optimum potential of compressed main memory system. Having learnt the related projects, some projects implemented the compression engine between cache and memory [14, 15, 22]. Although we do not use cache in this project since ARM Cortex-M0 does not support cache, those projects have improved our knowledge in further understanding the performance and benefit of cache. By synthesizing our understanding about caches to our project, we consider that it would be useful if we add a buffer near CPU that acts like a cache in our SoC.

The buffer we mentioned just now has some entries, in which each stores a previously decompressed blocks. Therefore, when CPU requests the same instruction that has already been decompressed before, CPU can fetches directly from that buffer's entry and there is no decompression process executed. We assume that it would reduce significant overhead since we have seen how CLB reduces the decompression overhead from 12.5% to 3.125% [15].

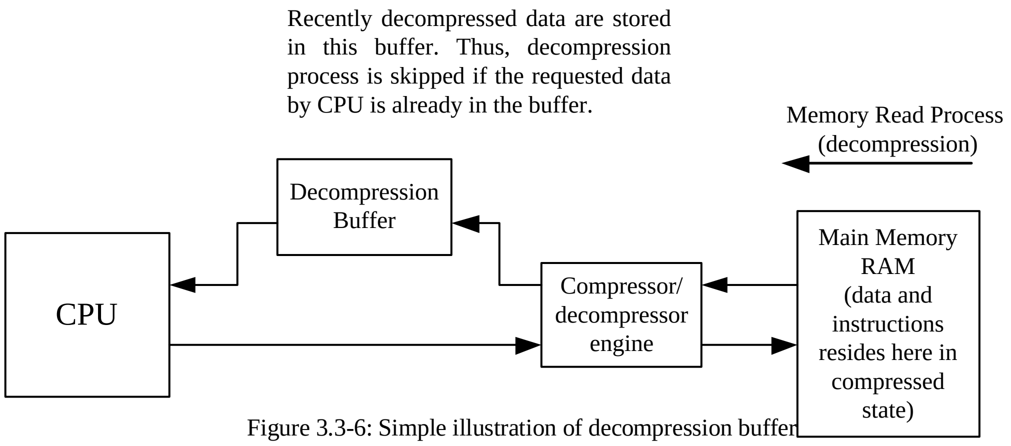

Recently decompressed data are stored in this buffer. Thus, decompression process is skipped if the requested data by CPU is already in the buffer.
Memory Read Process (decompression)
Decompression Buffer
CPU
Compressor/ decompressor engine
Main Memory RAM (data and instructions resides here in compressed state)

Figure 3.3-6: Simple illustration of decompression buffer

This buffer is actually has been implemented by Lee et al [21]. The buffer is called decompression buffer and does exactly like we mentioned previously. Lee et al integrated decompression buffer in their system called Selective Compressed Memory System (SCMS). There are plenty optimization implemented in SCMS such as selective compression, parallel decompression engine, fix-space allocation method (similar to fix-length compressed block), fetching decompressed data as early as possible and decompression buffer. SCMS is one of few compressed memory system that is capable to show the optimum potential of compressed main memory system. This system has 0.5 compression ratio with 53% decrease in data traffic as well as up to 20 % reduction in memory access time [21].

Lee et al performed simulation results analysis specific to measure the capability of decompression buffer. According to his analysis, decompression overhead is inversely proportional to number of the buffer entry. It was shown that decompression overhead can be reduced up to 80% when using 32 buffer entries. Howsoever, for buffer entries over 16, there is significant increase in implementation cost. Thus, it was concluded that the optimal number of buffer entries is 8 since it still gives 70% decompression overhead reduction with reasonable implementation cost [21].

3.5 Concluding Remarks

By examining our literature review, we have achieved all the objectives of the first aim of our research project. In this sub-chapter we would conclude what we have gained from the literature review to fulfill the objectives of the first aim listed in chapter two.

Firstly, we have identified the measurement parameters that determine a good compressed main memory system. A good compressed main memory system typically is required to have 0.5 compression ratio which means it should saves memory space at least by half of its original size. In addition, a good compressed main memory system should not produce significant compression-decompression overheads. In relation to our system, both compression ratio and overhead is achievable to be overcome since we are using X-MatchPROVW engine.

Secondly, we have analyzed the optimizations done by other projects related to our project. Optimization such as sophisticated translation table like one done by IBM MXT may enhance the compression ratio, further reduces overheads and improves the robustness of the system. In the other hand, SCMS aimed to simplify translation table by having fixed-length compressed block. Other way to perform optimization is to use decompression buffer that reduces further overheads.

Thirdly, we have synthesized pros and cons of each project we mentioned in the literature review by spotting which project that has better performance and optimization, in which we have mentioned that SCMS is by far the only system that reduced overhead significantly. And therefore, we decided to implement one of its optimization which is to use decompression buffer.

In our project, we avoided the use of translation table and decided to handle compressed block in straightforward manner. The reason behind this is that translation table requires enormous additional time to construct it properly because translation table requires some modification on the compression-decompression engine itself. Therefore, it is not feasible to be implemented in our project within time constraints we have (especially when we have to build the Cortex-M0 reference system from the beginning too). Additionally, it is not our objective to modify the compression and decompression engine (X-MatchPROVW) and not having translation table still allow us to achieve our objective to realize compressed main memory system for our research purpose.

What is more important in our research is that we need to investigate how compressed main memory system we develop will improve off-chip memory based system performance that was not emphasized by related projects we covered in literature review. The findings based on this intention might attract more attention to compressed main memory system research area since we all know that typical commercial system uses off-chip memory to save memory cost.

Chapter 4: Hardware Specifications

As mentioned in section 2 (Aims & Objectives), our project includes the implementation of the compressed main memory system for ARM Cortex-M0 in Aeroflex Gaisler GRLIB IP Core SoC environment. The whole system will be tested using Xilinx Virtex 5 XUPV5-LX110T FPGA board [6].

In this section we are going to elaborate each of the hardware specifications that are fundamental for building compressed main memory system according to our objectives.

4.1 ARM Cortex-M0 DesignStart Processor

As can be seen from the title of the project, we are designing compressed main memory system for ARM processor, which is Cortex-M0 processor. Cortex-M0 is a single bus interface 32-bit RISC (Reduced Instruction Set Computing) processor. This processor has 3-stage pipelines design containing fetch, decode and execution stage. However, it is academically categorized as Load-Store architecture because it has separate instructions for memory read/write and instructions for logical/arithmetic operations although there is no dedicated pipeline stage for memory load and store.[31].

Cortex-M0 only possesses 56 base instructions and adopts Thumb-2 instructions set. Thumb-2 is the successor of previous Thumb instruction set, in which Thumb-2 allows all operations to be done in single CPU state. Thumb-2 includes both 16-bit and 32-bit instructions, in which 32-bit is used only at the time when 16-bit instructions are not able to carry out the required operation. As a consequence, higher code density result [31].

Cortex-M0 has attained its great renown in embedded system industry due to its main advantages in term of its energy efficiency while still having remarkable performance than other embedded processor in its class. Size of Cortex-M0 is practically comparable to other 16-bit processors, and just several times bigger than 8 bit processor. This small size of M0 is caused by low gate counts (12k to 25k depending on features implemented) inside the chip that leads to lower power consumption. In addition, Cortex-M0 has a sleep-mode that reduces the power to minimum when sleep-mode is activated. However, since Cortex-M0 is 32-bit processor, it is significantly faster than popular 16-bit processor and even 8 times faster than the fastest 8051 implementations [31].

It is important to note that we are prototyping our system implementation using free version of Cortex-M0 namely Cortex-M0 DesignStart (Cortex-M0DS). Cortex-M0DS is delivered by ARM as preconfigured and obfuscated but synthesizable Verilog codes. There are some limited features in Cortex-M0DS (since it is free) despite being a fully working Cortex-M0 version. Cortex-M0DS lacks on having wake-up interrupt controller, hardware debug support and its AHB-Lite only support one master port. In addition, its Nested Vector Interrupt Controller (NVIC) only supports up to 16 interrupt inputs and the timer clock only has processor clock as its clocking source option [29]. Despite all missing features we mentioned, Cortex-M0DS still appropriate to be used in this project since we just intend to utilize the basic functionality of Cortex-M0.

Cortex-M0DS uses AHB-Lite (Advanced High-Performance Bus) on-chip bus interface with 32-bits bus width. AHB-Lite is part of AMBA (Advanced Microcontroller Bus Architecture) specification developed by ARM and it is considered as “simplified” form of AMBA AHB since it lacks of several protocols such as multiple bus masters protocol, arbiter protocol etc. AMBA AHB is intended to be a high performance system backbone bus. It allows low power efficient connection between processor, on-chip memory and off-chip memory [32, 33].

4.1.1 Basic CPU Transfer of ARM Cortex-M0DS

In order to integrate Cortex-M0DS into our desired SoC, the most important fundamental to understand is its basic CPU transaction which is also called "single transfer". A transfer is basically an operation of CPU receiving data from the bus ("read transfer") or sending data to the bus ("write transfer"). Because of ARM Cortex-M0DS adopts AHB-Lite master interface (as shown in figure 4.1-1), its single transfer operation is similar to AHB-Lite master's single transfer operation.

As we observed figure 4.1-2, for Cortex-M0DS to be able to start a transfer, signal HREADY must be driven high. Afterwards Cortex-M0DS initiates a transfer by its own as indicated by HTRANS = "10" alongside an assertion of an address to the bus and control signal to indicate whether it is read or write transfer (via HWRITE). In every transfer (whether it is read or write transfer), there are always two phases to be completed which are "Address Phase" and "Data Phase". Address phase is inherently a clock cycle of CPU asserting address to the bus and only happens when HTRANS = "10". Data phase is a clock cycle of CPU receiving data (via HRDATA ports) from the bus or sending data (via HWDATA ports) to the bus. Unlike the address phase, data phase can be extended by driving HREADY signal to low in purpose of waiting the data arriving to the CPU.

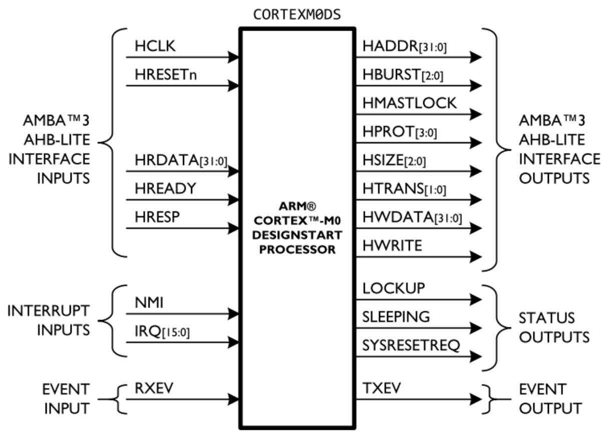


Figure 4.1-1: Cortex-M0DS Top-Level interface which adopts AHB-Lite Master interface since it acts as a master. [29]

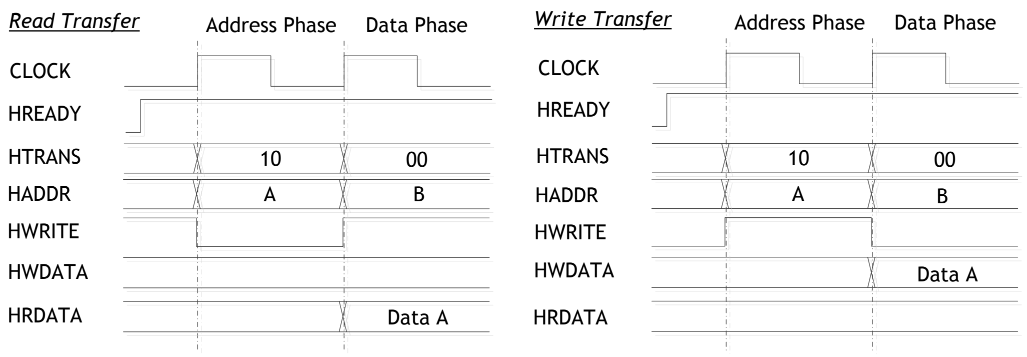


Figure 4.1-2: Basic ARM Cortex-M0DS transfer [32]

For more details regarding the function of each signal in the figure, please refer to AHB-Lite Protocol Specification [32].

From this point, it is important to know that we use the terminology Cortex-M0, Cortex-M0DS and CPU interchangeably.

4.2 Aeroflex Gaisler GRLIB IP Core Library System-on-Chip (SoC)

The system on chip environment we are using in this project is based on Aeroflex Gaisler GRLIB IP Core. IP core is a semiconductor intellectual property building block (core) that can be used within ASIC (Application Specific Integrated Circuit) and FPGA design. We chose this SoC environment because it is free to use although it is protected by GNU General Public Licenses.

GRLIB IP Library contains reusable IP Cores that are designed specifically for SoC development. It is written in form of VHDL (Very-High-Speed-Intergrated-Circuit Hardware Description Language) code that is broken down into many separated VHDL libraries, which mainly centered around common on chip bus. Therefore, GRLIB is claimed to be bus centric in which connects most IP cores through on-chip bus. The on-chip bus adopted by GRLIB is AMBA-2.0 AHB/APB bus by ARM due to the fact that it is free, market dominance and well documented.

The unique feature adopted by GRLIB is 'plug&play' feature which uses software for detecting system hardware configuration. This make possible to use applications or operating systems that have the capability for automatically configuring themselves against the underlying hardware. Thus, software applications used in this core need not to be specifically created solely for this system. Portability is also another advantage of using GRLIB [5]. Since GRLIB is technology independent, it can be easily implemented in FPGA technology as we intend to do in our project.

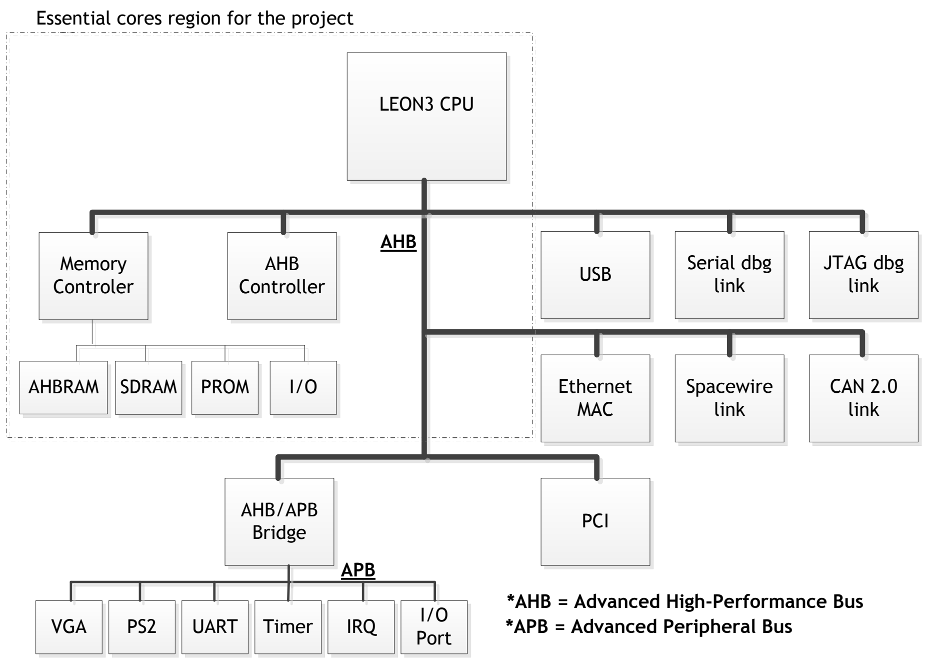


Figure 4.2-1: GRLIB SoC diagram with Leon3 processor [5]

Figure 4.2-1 shows the SoC implementation using GRLIB IP Library, in which Leon3 processor is already configured in the system by default. Other components shown in the picture are actually IP cores available inside GRLIB IP Library.

It is worth to know at this point that the first implementation phase of this project is to replace Leon3 Processor shown in the figure above with Cortex-M0DS. Elaboration on how it is done is further discussed in section 4.4.

4.3 X-MatchPROVW VHDL Module

We have discussed X-MatchPROVW engine's theory, its capabilities and its hardware architecture quite detail in literature review (sub-chapter 3.2.2). In this sub-chapter, we do not repeat what has already mentioned previously rather we describes its ports interface and how to control X-MatchPROVW to perform compression and decompression function.

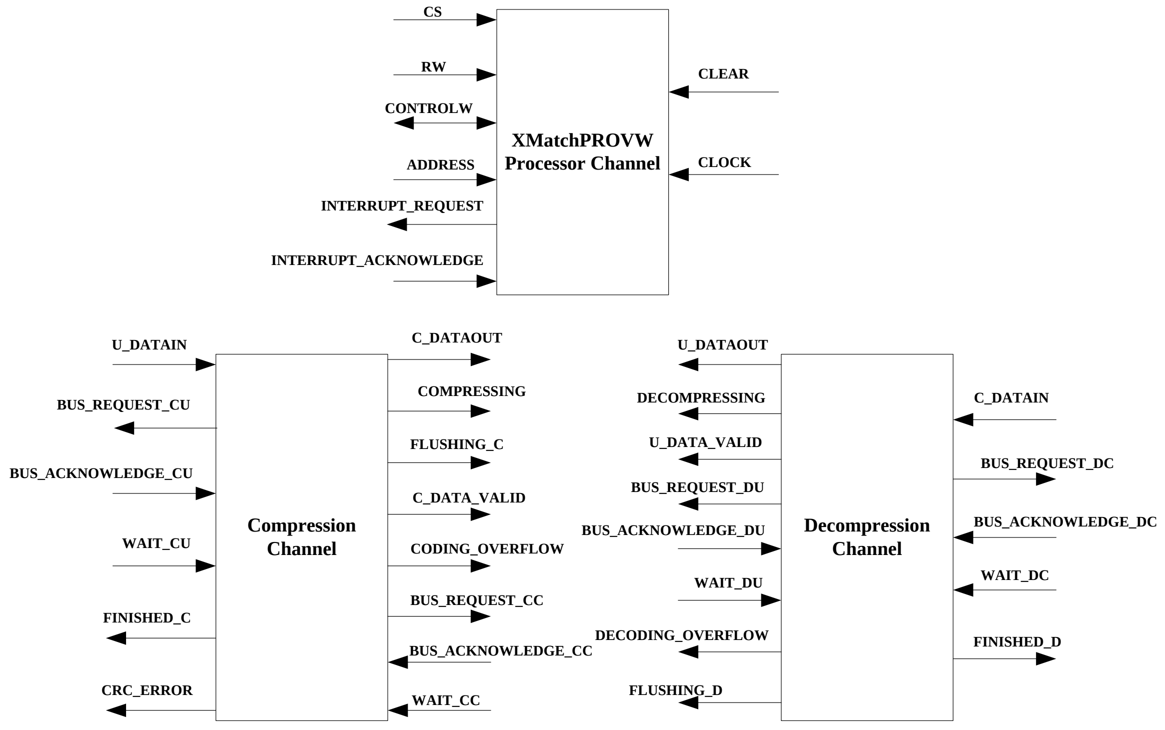


Figure 4.3-1: X-MatchPROVW Interface [41]

Figure 4.3-1 shows the port interface of X-MatchPROVW module written in VHDL. Its interface inherently is divided into three parts which are processor channel, compression channel and decompression channel. The details for each port's function can be referred from [41].

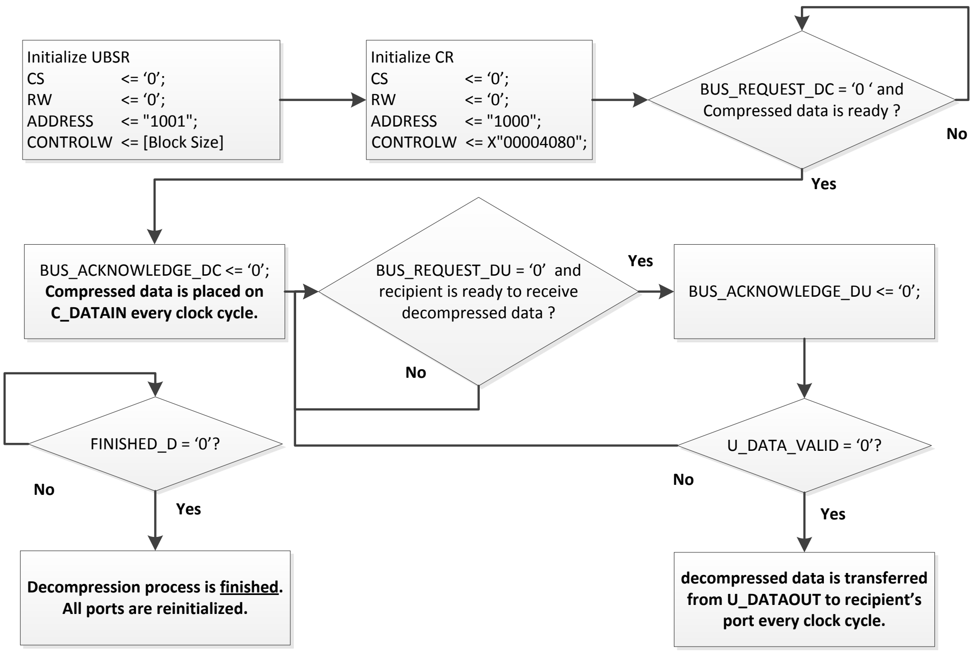

Initialize UBSR
CS <= '0';
RW <= '0';
ADDRESS <= "1001";
CONTROLW <= [Block Size]
Initialize CR
CS <= '0';
RW <= '0';
ADDRESS <= "1000";
CONTROLW <= X"00004080";
BUS_REQUEST_DC = '0 ' and Compressed data is ready ?
No
Yes
BUS_ACKNOWLEDGE_DC <= '0'; Compressed data is placed on C_DATAIN every clock cycle.
BUS_REQUEST_DU = '0' and recipient is ready to receive decompressed data ?
Yes
No
BUS_ACKNOWLEDGE_DU <= '0';
FINISHED_D = '0'?
No
Yes
U_DATA_VALID = '0'?
No
Yes
Decompression process is finished. All ports are reinitialized.
decompressed data is transferred from U_DATAOUT to recipient's port every clock cycle.

Figure 4.3-2: Flow chart of XMatchPROVW decompression procedure

Figure 4.3-2 shows the procedure to perform decompression function of X-MatchPROVW. We explain the decompression process first because it is the process that always executed first once system starts. To start decompression, two register which are Uncompressed Block Size Register (UBSR) and Control Register (CR) must be initialized. In order to access these register, we need to input address related to each register and write the value of CONTROLW port as shown in the figure. When initializing UBSR, CONTROLW value acts as the size of decompression block we would like the engine to set. When initializing CR, CONTROLW acts as configuration setup of the engine. Value of 0x00004080 in this case means that decompression is started and threshold value is set to 8. Notice that ports CS and RW are driven to '0' during these initializations and also we need to mention that all ports in X-MatchPROVW module is active low.

Since now decompression is started, X-MatchPROVW at some point will drive BUS_REQUEST_DC to 0 to tell that the engine is ready for accepting compressed data. Once this happens and compressed data provider is ready to supply compressed data, BUS_ACKNOWLEDGE_DC is driven low to tell the engine that compressed data is being supplied to the engine and C_DATAIN port starts receiving compressed data every clock cycle.

While the engine is receiving compressed data, the engine is also outputting decompressed data once the decompressed data is ready. To do this, the engine will drive BUS_REQUEST_DU to 0 when engine is ready to output decompressed data. Once this happens and recipient is ready, then BUS_ACKNOWLEDGE_DU is driven low to indicate that recipient is ready to receive decompressed data. Afterwards, engine will drive U_DATA_VALID to 0 and decompressed the data is transferred to recipient every clock cycle. When the decompression process and outputting decompressed data is all done, engine drives port FINISHED_D to 0 to indicate that decompression process is finished and all ports is reinitialized.

The compression procedure is somewhat similar to decompression procedure except ports that are used by compression process are those from compression channel. Once we understand the decompression process, it will be easy for us to understand the compression part as shown by figure 4.3-3.

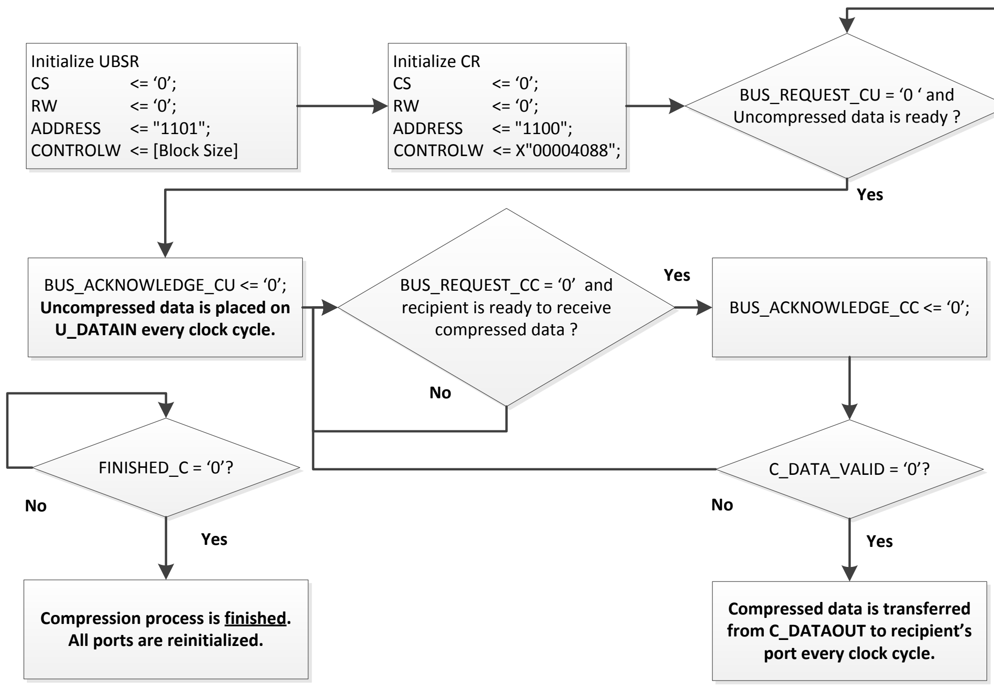

Initialize UBSR
CS <= '0';
RW <= '0';
ADDRESS <= "1101";
CONTROLW <= [Block Size]
Initialize CR
CS <= '0';
RW <= '0';
ADDRESS <= "1100";
CONTROLW <= X"00004088";
BUS_REQUEST_CU = '0 ' and Uncompressed data is ready ?
No
Yes
BUS_ACKNOWLEDGE_CU <= '0'; Uncompressed data is placed on U_DATAIN every clock cycle.
BUS_REQUEST_CC = '0' and recipient is ready to receive compressed data ?
Yes
No
BUS_ACKNOWLEDGE_CC <= '0';
FINISHED_C = '0'?
No
Yes
C_DATA_VALID = '0'?
No
Yes
Compression process is finished. All ports are reinitialized.
Compressed data is transferred from C_DATAOUT to recipient's port every clock cycle.

Figure 4.3-3: Flow chart of XMatchPROVW compression procedure

4.4 Xilinx Virtex 5 XUPV5-LX110T Board & FPGA Implementation Flow

We are implementing our system design in this project using Xilinx Virtex 5 XUPV5-LX110T FPGA board (from now we can call it XUPV5 board). We use XUPV5 (XUP stands for Xilinx University Program) board because it is a high-end development and evaluation platform board for research areas such as embedded systems, computer architecture, and operating systems. Key features important to our projects provided by the board include Xilinx Virtex-5 XC5VLX110T FPGA[35] 256-MB DDR2 small outline DIMM (SODIMM), JTAG interface, RS-232 serial port, 2 line LCD (each line accommodate 16 characters) and etcetera.

FPGA is a type of integrated circuit formed by numbers of reconfigurable logic blocks wired together with reprogrammable interconnects. FPGA is well-known to provide a low cost solution for low volume productions and allows faster time to market compared to ASIC. Basic building block (logic block) in Xilinx FPGA is called CLB (configurable logic blocks) that normally contains SRAM-based LUT (look up table) where users can implement their logic. In particular, Xilinx Virtex 5 CLB contains a pair of slices, in which each slice contains four LUT, four storage elements and carry logic [35, 36].

In order to implement our system design to Virtex-5 FPGA, we use Xilinx Integrated Synthesis Environment (ISE) version 13.2 and we follow typical Xilinx FPGA implementation flow as shown in figure 4.4-1. According to the figure, we firstly verify our design by performing behavioral simulation with our designed test bench code. Afterwards, we synthesize the HDL code to produce gate-level netlist of the design (list of interconnects at gate level) in which this netlist is saved into file called NGC (contains logical design data and constraints). Then, the resulting netlist enters three stages ISE implementation process.

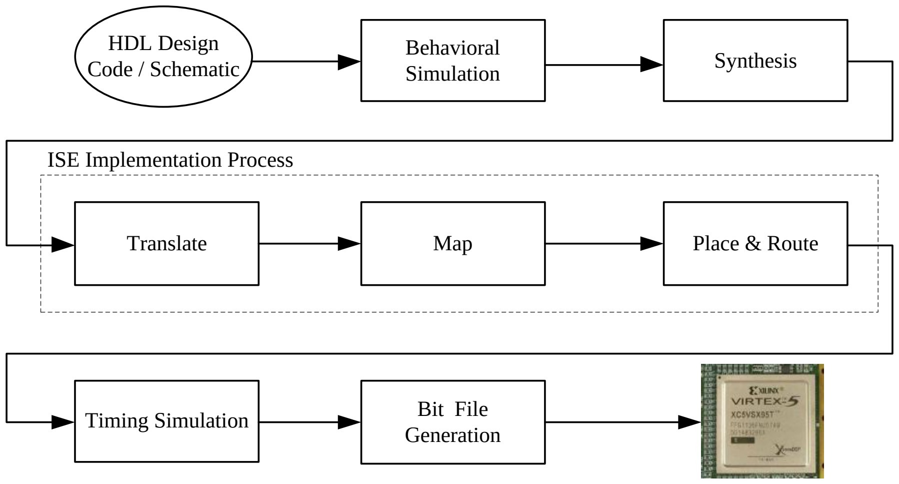


Figure 4.4-1: Typical Implementation flow in Xilinx ISE [38, 39]

ISE Implementation process consists of translate (merges multiple netlists and constraints into Xilinx design files), map (grouping gates to slices and IOBs), and place & route (placing and routing the component to the chip according to the design and obtain its timing data). Translate process is performed by NGDBUILD program which converts NGC to NGD netlist file (netlist which is dependent upon specific simulation library called SIMPRIM[36]). Map process is driven by MAP program which maps specific device such as LUTs, flip-flops etcetera. The resulting output is saved in NCD format which contains detailed information about switching delays. Place and route is driven by PAR program which defines locations of components and interconnections inside FPGA. Place and route is the most time consuming process and its output is also saved in NCD format [38, 39]

After ISE implementation process is done, timing simulation is conducted to produce timing report of the design and lastly, the design is converted to Bitstream file (.Bit file) so that it can be downloaded to Virtex-5 FPGA via USB JTAG cable or converted to PROM file.

Chapter 5: Understanding Previous Work

5.1 Overview of the Previous Work

To begin with, a project similar to our title was attempted last year by Mateusz Piekarek and intended for MEng final year project dissertation in University of Bristol [40]. In his project, he attempted to build Code Compression System (refer to section 3.1) in which the system only initially having compressed instruction in the off-chip RAM (DDR SDRAM, from now on we just call it SDRAM) and decompresses them on-the-fly. Data whether it resides on stack memory region or heap memory region is left as it is. In other words, on-the-fly compression functionality was not enabled and only on-the-fly decompression functionality was integrated.

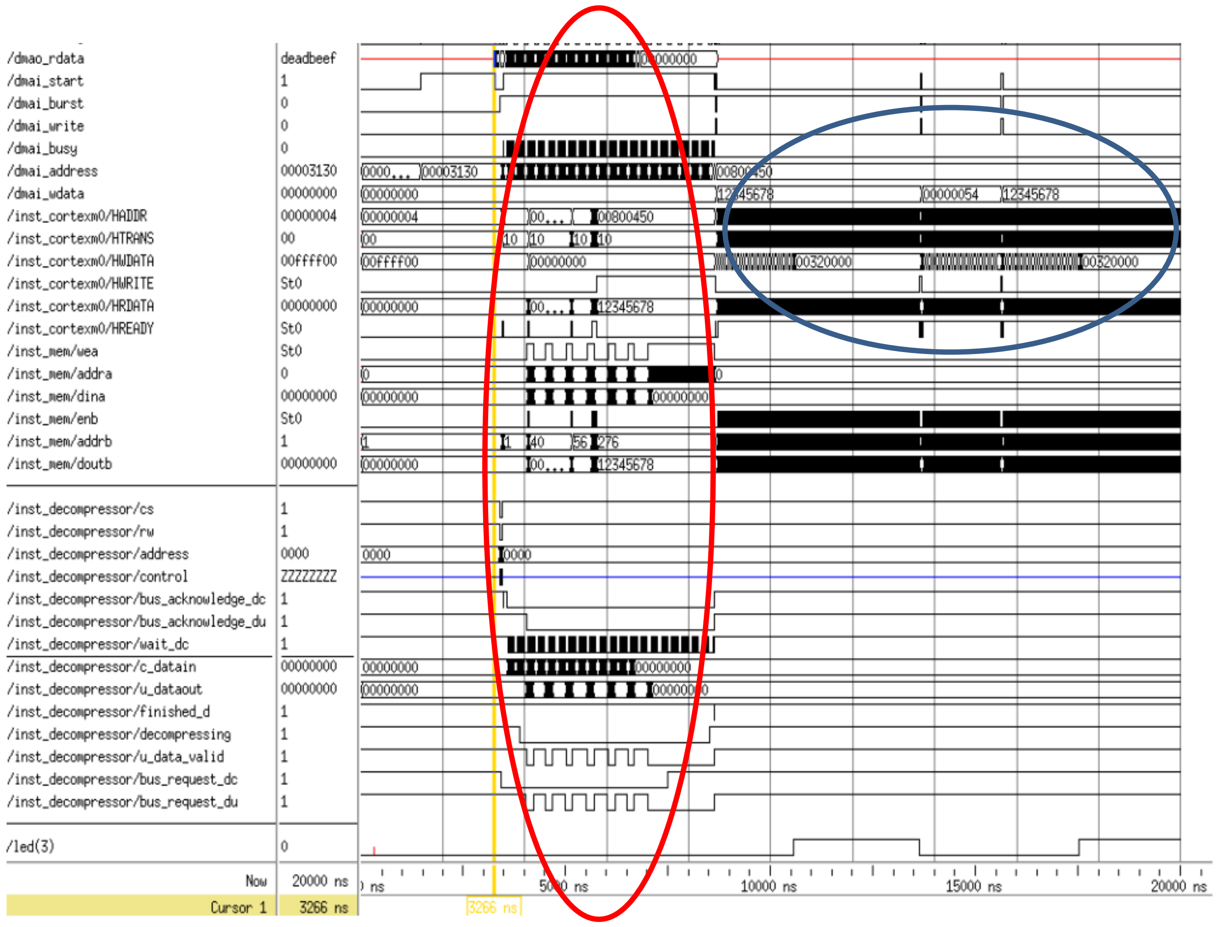

/dmao_rdata deadbeef
/dmai_start 1
/dmai_burst 0
/dmai_write 0
/dmai_busy 0
/dmai_address 00003130
/dmai_wdata 00000000
/inst_cortexm0/HADDR 00000004
/inst_cortexm0/HTRANS 00
/inst_cortexm0/HWDATA 00ffff00
/inst_cortexm0/HWRITE St0
/inst_cortexm0/HRDATA 00000000
/inst_cortexm0/HREADY St0
/inst_mem/wea St0
/inst_mem/addra 0
/inst_mem/dina 00000000
/inst_mem/enb St0
/inst_mem/addrb 1
/inst_mem/doutb 00000000
/inst_decompressor/cs 1
/inst_decompressor/rw 1
/inst_decompressor/address 0000
/inst_decompressor/control ZZZZZZZZ
/inst_decompressor/bus_acknowledge_dc 1
/inst_decompressor/bus_acknowledge_du 1
/inst_decompressor/wait_dc 1
/inst_decompressor/c_datain 00000000
/inst_decompressor/u_dataout 00000000
/inst_decompressor/finished_d 1
/inst_decompressor/decompressing 1
/inst_decompressor/u_data_valid 1
/inst_decompressor/bus_request_dc 1
/inst_decompressor/bus_request_du 1
/led(3) 0
Now 20000 ns
Cursor 1 3266 ns
0 ns
5000 ns
10000 ns
15000 ns
20000 ns
3266 ns

Figure 5.1-1: LED blinking application with decompression simulation.[40]

The above figure is one of his simulation screenshot showing that his system is able to decompress instruction on-the-fly and run LED blinking application. The area marked by red circle shows the decompression activity.

5.2 Unresolved Problem on Previous Work

Although Piekarek was able to integrate the on-the-fly decompression function, his system is only able to run “properly” one ARM Cortex-M0 application which is LED blinking program. This LED blinking program only performs read instructions to blinks LED without performing any data operation. In addition, he stated in his dissertation that there was unresolved problem regarding stack memory operation, in which the source of the problem was unknown.

The main critical issue faced by Piekarek in the previous project was that his system could not perform Cortex-M0 test program containing functions (which of course contains branch instructions), in which this is out of his expectation. As a result, this problem hampered the progress of the project since further test and development of the system was not possible to be performed without having the system gracefully performs basic control function instructions (branch instructions). Unfortunately, the exact cause of this problem was remained unknown at that time, however, the suspicion was addressed on the behavior of the system in performing stack memory operation.

In typical systems, when program jumps into functions or loops, some registers contents are temporarily stored (normally called as “Push” operation) into the memory storage called stack so that there are still enough number of registers to perform routines defined in the functions or loops. After execution leaves the function, the temporary stored register contents in the stack are loaded back to the register (normally called as “Pop” operation) to restore the execution state back to the point before entering functions, yet now with some updated register contents(depending on what the function routine does). In Cortex-M0, the push and pop of the stack memory operation requires a pointer (called stack pointer) to track the last filled data in the stack and the pointer pre-decrements for each push [31].

According to observation done by Piekarek to track the cause of the main issue he faced, there were unwanted differences in term of address of memory where the stack points between debug operation he conducted in ARM Keil MDK and system behavioral simulation he conducted in Modelsim (where the hardware functionality of the system was simulated). By knowing this fact, we concluded that the system at his project state could not handle stack operation as proper as what can be expected from a normal Cortex-M0 system.

After further time spent and many experiments done in analyzing his project, we finally found the main cause of his system’s inability to do stack operation. The stack operation was not able to be performed simply because the system could not perform “write operation” to the off-chip memory (SDRAM). Write operation is a fundamental operation which is as important as read operation between CPU and memory because a working system must be able to do both read and write operation to the memory. If one of these two functionalities is missing, the system simply does not work.

Although it can be noticed in figure 5.1-1 that there were “write” activities performed by CPU as marked by blue circle above, this write operation was just a “CPU attempt” to write to the bus and not to the memory itself. This is clearly seen from the address shown (0x008000450) during write operation above which was not pointing to any memory region. Also, there was no evidence provided in his dissertation that the CPU really writes to the memory (SDRAM) by showing that SDRAM ports receive and store data being sent by CPU through the bus.

To prove that the write operation did not work properly, we tried to run a very simple ARM application program, in which it writes “0xAAAA5555” to the writeable address region in the memory (in this case we write to address “0x00000404”) and then read it back again. Data 0xAAAA5555 is a data pattern detectable by LED detector mechanism to drive LED signal. Therefore, we expect from the program that when 0xAAAA5555 is being written to the memory, LED turns on, and when data is being read from the memory, LED also turns on. Between write and read of 0xAAAA5555, we deliberately turn off the LED. This simple program can be found in submitted zip file: “Sourcecodes_forassessment_ms1718\CompletelyDesigned by AGHA\ARM_Cortex-M0 C program.”

This program which is called Write_Read.c also can be understood in figure 5.2-1.

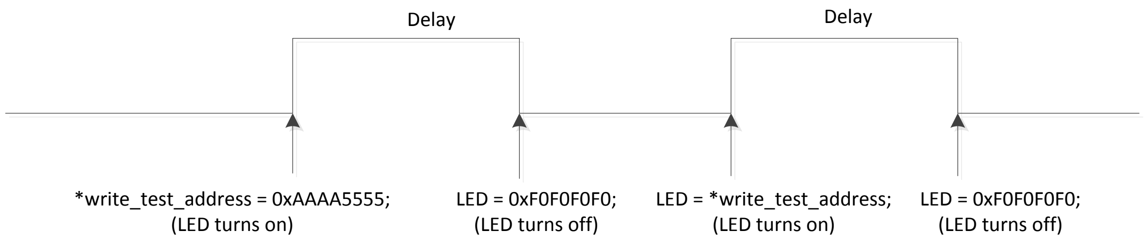
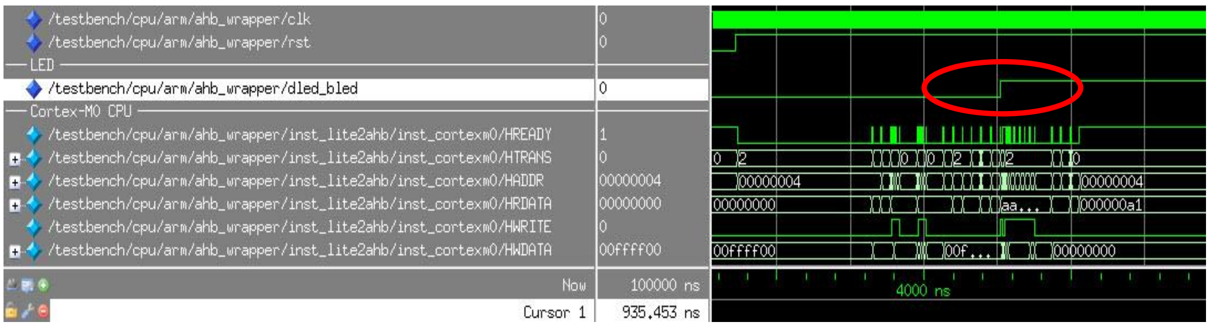


Figure 5.2-1: Expected LED signal behavior for Write_Read.c program.
*write_test_address points to location 0x00000404.



```
/testbench/cpu/arm/ahb_wrapper/clk                                   0
/testbench/cpu/arm/ahb_wrapper/rst                                   0
LED
/testbench/cpu/arm/ahb_wrapper/dled_bled                             0
Cortex-M0 CPU
/testbench/cpu/arm/ahb_wrapper/inst_lite2ahb/inst_cortexm0/HREADY    1
/testbench/cpu/arm/ahb_wrapper/inst_lite2ahb/inst_cortexm0/HTRANS    0
/testbench/cpu/arm/ahb_wrapper/inst_lite2ahb/inst_cortexm0/HADDR     00000004
/testbench/cpu/arm/ahb_wrapper/inst_lite2ahb/inst_cortexm0/HRDATA    00000000
/testbench/cpu/arm/ahb_wrapper/inst_lite2ahb/inst_cortexm0/HWRITE    0
/testbench/cpu/arm/ahb_wrapper/inst_lite2ahb/inst_cortexm0/HWDATA    00ffff00
                                                          Now        100000 ns
                                                          Cursor 1   935.453 ns
```

Figure 5.2-2: Simulation of Write_Read.c program on previous work’s basic single transfer system.

Piekarek developed his system step-by-step from building basic single transfer system to system which integrated on-the-fly decompression function and burst-parallel instruction fetch mechanism in his wrapper. However, since we were trying to solve a problem that seemed to appear from the early phase of the system development, we tested Write_Read.c program only on his basic single transfer system (called Lite2ahb in Piekarek’s dissertation [40]) that excludes the decompression function to see if the system performs write operation properly on its basic form.

From figure 5.2-2, we can observe from simulation graph that behavior of LED is not the same as can be expected from figure 5.2-1. This proves that there is something wrong with the design and the explanation would be delivered in later sub-chapter 6.2.

5.3 Conclusion of Analyzing Previous Work

To conclude this section, we have narrowed down the problem, in which we found that write operation did not function properly. Because write operation is fundamental but was found not working even in the basic form of previous work’s system design, we decided to not using any of the previous work’s design and start everything from beginning by firstly fixing the write operation problem. Fortunately, we have identified the cause of write problem and provide alternative solution to fix it, which would be explained in the next chapter.

Chapter 6: System Design & Solutions

6.1 Overview of the Design Phases and Methodology.

As can be understood from “Chapter 2: Aims & Objectives”, we are expected to come up with design and working implementation of compressed main memory system for Cortex-M0 SoC backed by X-MatchPROVW engine. We have mentioned that we are using GRLIB IP Core Library SoC as the SoC environment for Cortex-M0. However, since GRLIB IP Core Library has Leon3 processor attached by default as its CPU, we need to remove Leon3 and replace it with Cortex-M0.

Placing Cortex-M0 into the SoC is not as simple as connecting a module to another. We need to design a top-level module called “wrapper” containing the Cortex-M0 CPU and mechanism for dealing the interface between the Cortex-M0 CPU and the SoC bus (in this case AMBA AHB*). In short, the wrapper which interfaces the CPU and SoC bus is the main component we are designing in this project.* It is also the same case when we integrate X-MatchPROVW into the SoC, in which everything else we need to make compressed main memory system are designed and integrated in this wrapper.

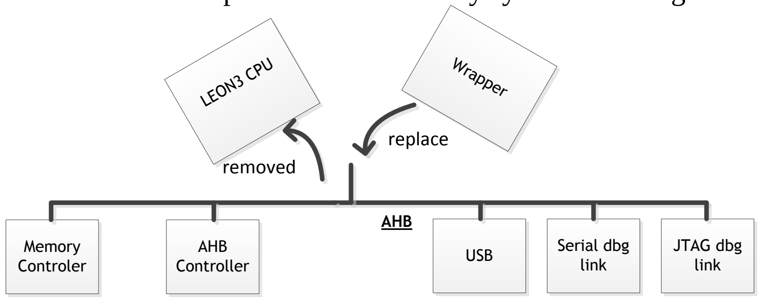


Figure 6.1-1: GRLIB SoC partial diagram having Leon3 being replaced by our wrapper design.

There are several design phases we need to conduct in this project, in which they are listed as follows:

1. Overcome the write problem occurred from the previous work

2. Design a wrapper which integrates only Cortex-M0 into the SoC in purpose of building basic transfer Cortex-M0 SoC. We call this wrapper design as “CM0_wrapper”.

3. Design a wrapper which integrates X-MatchPROVW engine, Cortex-M0 CPU and other necessary components in order to modify the SoC to become a compressed main memory system. We call this wrapper design as “CM0X_wrapper”.

4. Compare the performance of CM0X_wrapper with CM0_wrapper to evaluate performance improvement given by compressed main memory system.

After planning the design phases, we also decided to use several different software tools to realize the design. There are three fundamental softwares we are using which are: Modelsim, Xilinx ISE 13.2 and ARM Keil Microcontroller Development Kit (Keil MDK).

With the list of software tools above, the methodology in designing the system is as follows: firstly, Modelsim is used to compile and simulate our design coded in VHDL. After the design has been verified by simulation, we use Xilinx ISE 13.2 to synthesize the design code into gate-level netlist so that we can get the implementation report of the design and also to generate the bit file to be downloaded into the FPGA board. Keil MDK is used to design Cortex-M0 application programs written in C-programming language, in which these programs are to be executed by our system to check if Cortex-M0 works properly as CPU.

6.2 Solution to Previously Unresolved Problem

6.2.1 The Cause of Write Operation Problem

As discussed in chapter 5, we have identified further that inability of the system to do write operation in the previous work causes the system to be unable to perform stack operation and other sophisticated functions. Nevertheless, knowing that write operation has problem also means another work for us to identify what causes it.

After further experiments and analysis done to find the cause of the write problem, we concluded that there were two main causes:

1. Data to be written to memory was not converted to big-endian.
2. Incorrect method of wrapper handles “write operation”.

For the first point, it is important to know that when SDRAM is used, data and instructions stored in SDRAM are in form of big-endian. This is because the process of creating memory initialization for SDRAM (creating .srec file) is done in big-endian form by default. However, Cortex-M0 CPU operates with little-endian form of data (please refer to [44] regarding endianness). In previous work’s basic wrapper system, there is a mechanism of converting big-endian data/instructions to little-endian data/instruction before data/instruction enters Cortex-M0. Nevertheless, there was no mechanism of converting back little-endian data/instruction to big-endian form when Cortex-M0 intends to write to memory.

For the second point, after we examined closely the previous work’s basic transfer system wrapper codes, we concluded that the method of the previous work’s wrapper in handling write operation was not proper. If we observe the design of basic transfer wrapper called “Lite2Ahb” in Piekarek’s dissertation page 24 [40], we could see that the design used four states of finite state machine, in which according to our findings, such state machine is not necessary for the basic transfer Cortex-M0 system and may ruin the timing of the data arriving to CPU or memory. In addition, additional bus master interfacing mechanism provided by GRLIB SoC was used (called AHBMST), in which also not necessary because it increased the complexity of the system and takes up more logic resources inside the chip.

6.2.2 Solution to Write Operation Problem

The solution to overcome write operation problem is simply by completely redesign the Cortex-M0 basic transfer wrapper by taking care the points mentioned in the previous sub-chapter.

In our wrapper design, we added simple codes to convert back the little-endian data to big-endian when CPU writes to memory. We also removed AHBMST module so that we interfaced CPU directly to the AHB interface. The most apparent difference is that we do not use state-machine to control the interface between Cortex-M0 and AHB.

To prove that the fix we implemented in our new wrapper design works, we tried to run the “Write_Read.c” program which is the same program we ran on the previous work’s design in chapter 5. The simulation graph regarding this program running in our design is depicted in figure 6.2-1.

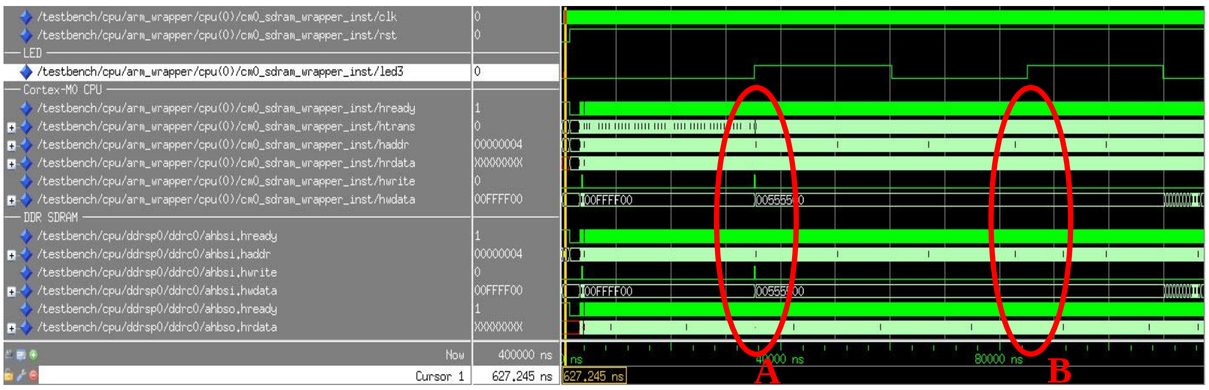


Figure 6.2-1: Simulation of CM0_SDRAM_wrapper design running Write_Read.c

From figure 6.2-1, it can be seen that there are two LED blinks, in which the LED behavior matches with LED behavior expectation diagram from figure 5.2-1. In addition, we provide two other simulation graphs which further prove the writing and reading operation, in which depicted by figure 6.2-2 and 6.2-3. Figure 6.2-2 zooms the area marked by circle A which indicates writing operation and figure 6.2-3 zooms the area marked by circle B which indicates reading back what was written in that particular address.

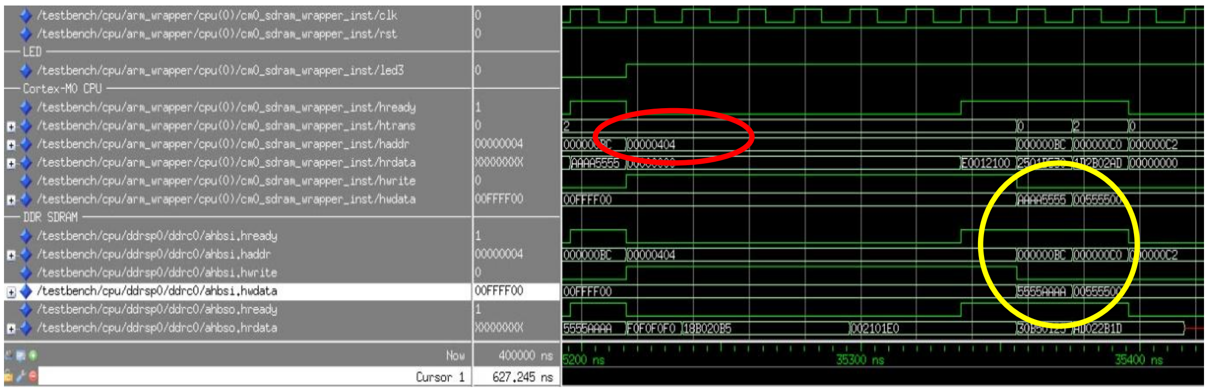


Figure 6.2-2: Evidence of writing 0xAAAA5555 data to address 0x00000404 in SDRAM from running Write_Read.c

In figure 6.2-2, we can observe the red circle shows haddr = 0x00000404 and we see that hwrite is high along that particular address. This means that CPU is performing address phase of write transfer. Normally, before CPU try to write a data, it reads the data first from memory. Therefore, we see hrdata contains 0xAAAA5555 before writing (near red circle) and this triggered LED to turn on. After that, when htrans = 0, we can see data phase is progressing as marked by yellow circle. We see that hwdata from CPU contain 0xAAAA5555 and ahbsi.hwdata in SDRAM contains the big-endian version of 0xAAAA5555 which means that the data is being written to SDRAM.

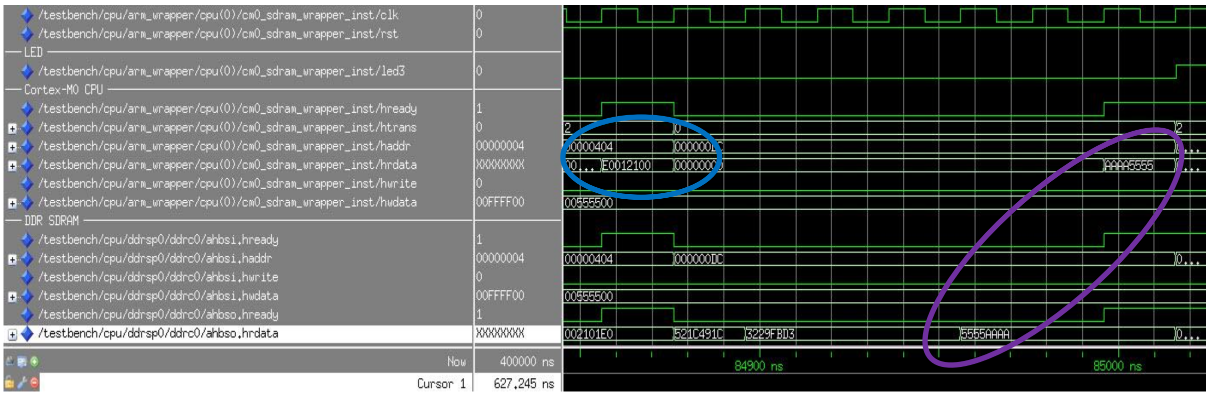


Figure 6.2-3: Evidence of reading 0xAAAA5555 data from address 0x00000404 in SDRAM from running Write_Read.c

In figure 6.2-3, blue circle marks haddr = 0x00000404 and hwrite = 0 which indicate address phase for read transfer for that particular address. Afterwards, we can observe in purple circle that the big-endian version of 0xAAAA5555 comes from SDRAM in ahbso.hrdata and transferred to CPU hrdata. This confirms that the write operation to address 0x00000404 was successful and thus, it could be read back and triggered the LED again.

It is important to know that this wrapper is not the main wrapper we are using in this project due to other problem in SDRAM that is to be explained in the later sub-chapter. However, the basic concept of this wrapper is very similar to the one we are going to explain in sub-chapter 6.3 which works with on-chip AHBRAM instead of off-chip SDRAM. The detail of the design process of the wrapper that works with AHBRAM is going to be explained in sub-chapter 6.3.

6.2.3 Stack & Heap Operation Problem and Solution

Having to know that write operation was finally overcome, we further realized that it still could not perform stack operation and heap operation. This problem still limits the capability of the system to perform more sophisticated program. This explains us about why the previous work's system can only perform LED blinking program which just read instructions to blinks LED without performing any data operation.

After further efforts were put to analyze the problem, we realized that the Cortex-M0 test program used by previous work was not developed to correctly map the read-write memory region in Keil MDK. The previous work's Cortex-M0 application program only contained "main.c" and "vector.c" file. Main.c described the functionality of the program and vector.c contained vector table to initialize address of exception types. Vector table is very important to be included because it also initializes the starting address of main stack pointer for stack operation.

What was wrong in the previous work is that the file vector.c was made very simple even it did not contain any means of defining the range of stack memory region and heap memory region. There was also no scatter file created that is normally used to create the layout of the memory region during compilation of the program. The absence of defining range of stack and heap memory region in vector.c file and creation of scatter file resulting read-write region was not configured during compiling the program, thus, stack and heap operation would not function properly. To fix this, we followed an alternative standard made by Keil MDK to create an assembly based file vector table called "startup.s".

This startup.s file provides a proper way of mapping the memory in the hardware according to the memory layout we design. Therefore, before we explain the important things about startup.s file, we explain first our memory layout design which represents the AHBRAM on-chip main RAM we are going to implement in SoC. (from this point, we call on-chip main RAM as AHBRAM).

Figure 6.2-4 shows the layout of AHBRAM we designed in our system. We decided to have a total of 32 KB (Kilo Bytes) AHBRAM because of two reasons: 1. We do not intend to create large size of Cortex-M0 test program, 2. The bigger the size of the AHBRAM, the longer the process to create memory initialization file in Xilinx ISE, which involves the creation of AHBRAM itself every time we would like implement new Cortex-M0 program in the simulation.

From that figure, we can observe that there are two separate regions which are Read-Only region and Read-Write region. We divide equally 32 KB total memory size into 16 KB for Read-Only region and 16 KB for Read-Write region. From 16 KB Read-Write region, we divide it again equally into 8 KB stack memory region and 8 KB heap memory region.

It is important to understand that we intend to implement 32 KB or accurately 32768 bytes AHBRAM size which has 32 bits data width for each address. In other words, each address contains 4 bytes of binary information and therefore there are 8192 address slots to construct 32768 bytes memory size. Please do not confuse about the difference between “address number in hex” and “address slot” (also called memory depth) in the figure below. Address slot is simply 32768 bytes divided by 4 bytes (32 bits) equal to 8192 slots. Address number in hex is simply address number corresponds to each byte, in which the last address number (0x8000) corresponds to address of byte 32768 (the last byte).

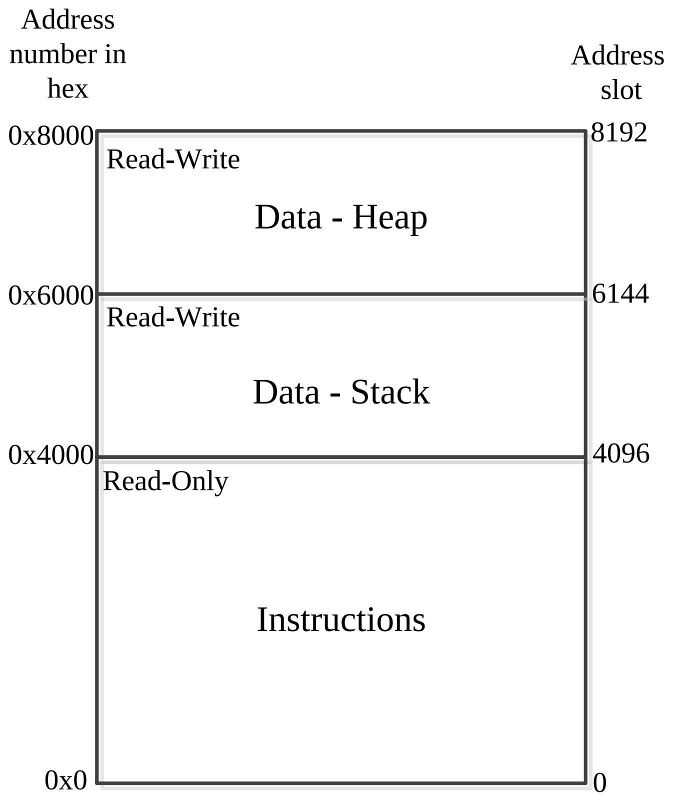
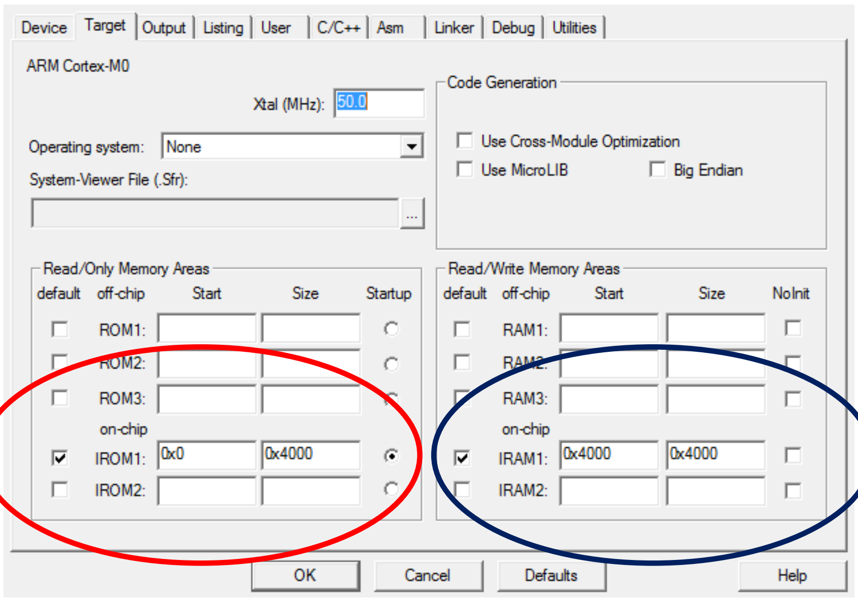


Figure 6.2-4: AHBRAM or main RAM layout design.

After we understand the layout of memory we are going to map in the hardware, we would describe the correct method of configuring memory map in Keil MDK since this part is considered as our effort that is fundamental in fixing this problem. To begin with, we need to open the “option for target” window in Keil MDK and select the “Target” tab as shown in figure 6.2-5. The area marked by red circle shows the value to configure the range of Read-Only region for instruction, in which it starts from 0x0 to 0x3FFF. The area marked by blue circle shows the value to configure the range of Read-Write region, in which it starts from 0x4000 to 0x7FFF.



ARM Cortex-M0

Xtal (MHz): 50.0

Operating system: None

System-Viewer File (.Sfr):

Code Generation

- [ ] Use Cross-Module Optimization
- [ ] Use MicroLIB
- [ ] Big Endian

Read/Only Memory Areas

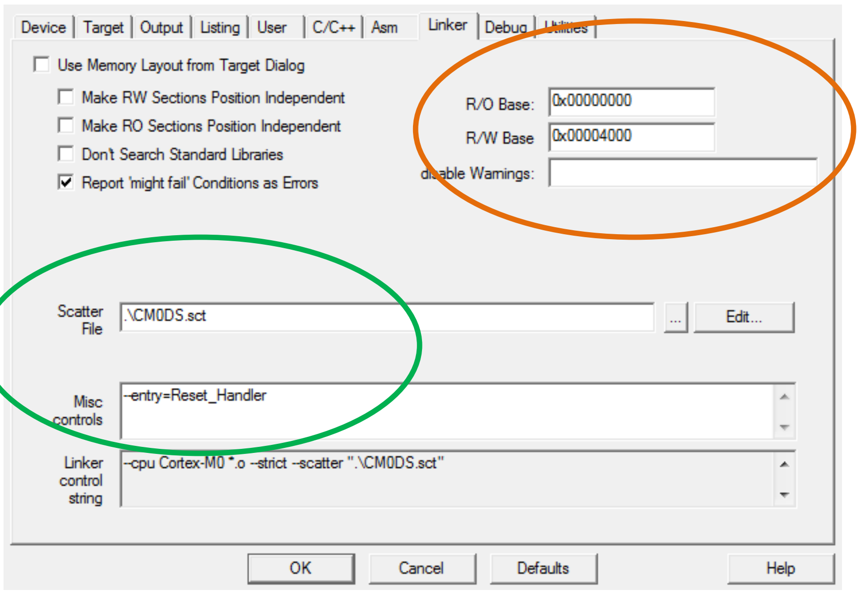
default	off-chip	Start	Size	Startup
☐	ROM1:			○
☐	ROM2:			○
☐	ROM3:			○
	on-chip			
☑	IROM1:	0x0	0x4000	●
☐	IROM2:			○

Read/Write Memory Areas

default	off-chip	Start	Size	NoInit
☐	RAM1:			☐
☐	RAM2:			☐
☐	RAM3:			☐
	on-chip			
☑	IRAM1:	0x4000	0x4000	☐
☐	IRAM2:			☐

Figure 6.2-5: “Target” tab in “Option for target” window of Keil MDK

Afterwards, we switch to “Linker” tab as shown by figure 6.2-6. In this tab, we set Read-Write base as shown by the orange circle. We should remember that 0x4000 refers to 16384 in decimal number which is half of the size of AHBRAM we designed. After that, we input scatter file and define the entry of the program as shown by green circle below. The entry of the program points to “Reset_Handler” label, in which the label is defined in startup.s file.



- [ ] Use Memory Layout from Target Dialog
- [ ] Make RW Sections Position Independent
- [ ] Make RO Sections Position Independent
- [ ] Dont Search Standard Libraries
- [x] Report 'might fail' Conditions as Errors

R/O Base: 0x00000000

R/W Base 0x00004000

disable Warnings:

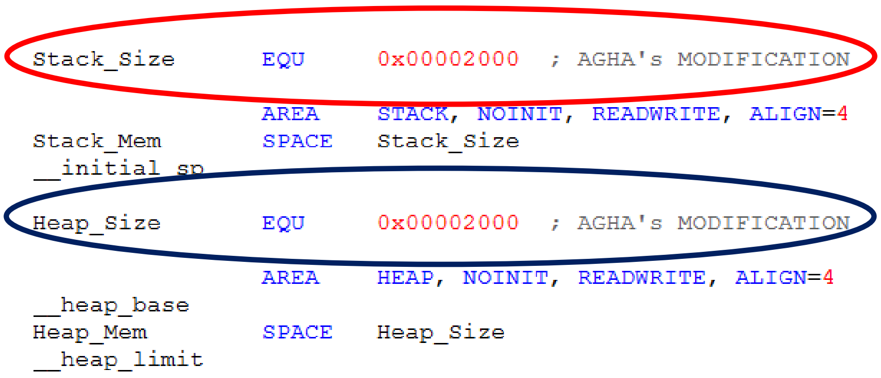
Scatter File: .\CM0DS.sct

Misc controls: --entry=Reset_Handler

Linker control string: --cpu Cortex-M0 *.o --strict --scatter ".\CM0DS.sct"

Figure 6.2-6: “Linker” tab in “Option for target” window of Keil MDK

For startup.s and scatter file, these files are provided in Sourcecodes_forassessment_ms1718\PartiallyModified by AGHA\ARM Keil files . However, we would like to highlight the crucial part of startup.s code as shown in figure 6.2-7 below. Area marked by red circle indicates the stack memory size and area marked by blue circle indicates heap memory size, in which we allocated *8192* bytes (8KB) (in hex: 0x00002000) for each of them.



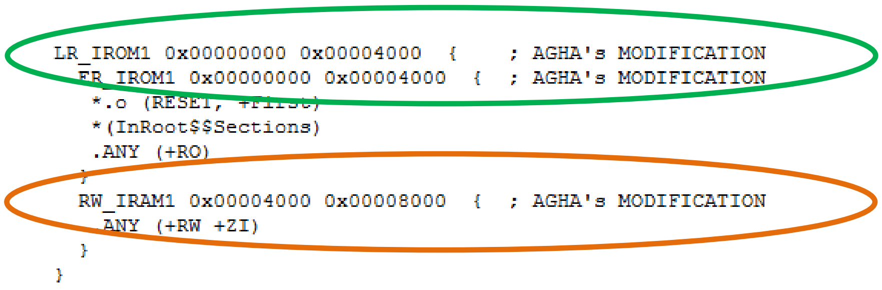
```
Stack_Size      EQU     0x00002000  ; AGHA's MODIFICATION

                AREA    STACK, NOINIT, READWRITE, ALIGN=4
Stack_Mem       SPACE   Stack_Size
__initial_sp
Heap_Size       EQU     0x00002000  ; AGHA's MODIFICATION

                AREA    HEAP, NOINIT, READWRITE, ALIGN=4
__heap_base
Heap_Mem        SPACE   Heap_Size
__heap_limit
```

Figure 6.2-7: Part of code in startup.s file which defines stack and heap size.

Regarding the scatter file as shown in figure 6.2-8 below, we must configure them to map the same address as shown in figure 6.2-5. Keep in mind that size for Read-Only region is 16384 bytes (16KB) which is the same as Read-Write region size.



```
LR_IROM1 0x00000000 0x00004000  {    ; AGHA's MODIFICATION
  ER_IROM1 0x00000000 0x00004000  {  ; AGHA's MODIFICATION
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
  RW_IRAM1 0x00004000 0x00008000  {  ; AGHA's MODIFICATION
   .ANY (+RW +ZI)
  }
}
```

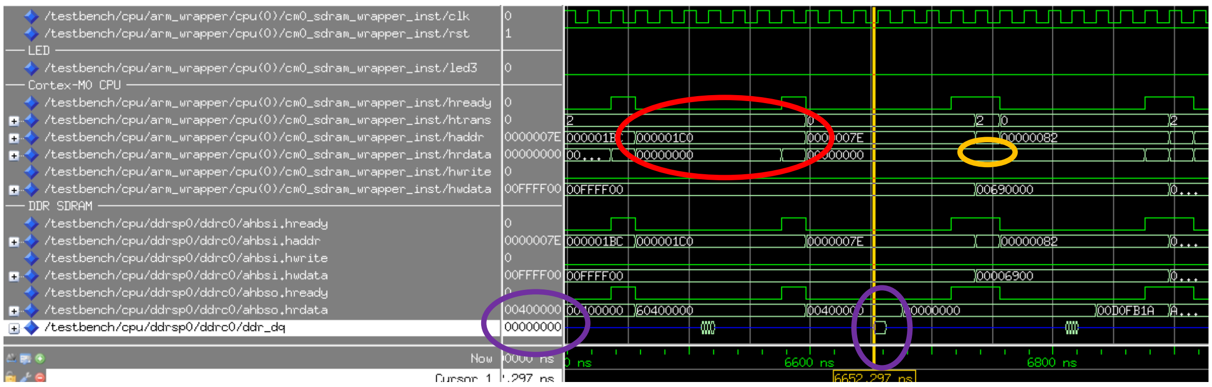
Figure 6.2-8: Content of scatter file (CMODS.sct) which defines the memory map.

With all these setups we have shown, we have made sure that the memory map is defined properly according to our specifications.

6.2.4 SDRAM Problem

After we successfully managed to fix stack and heap operation problem, we still find problem that hampers us to run proper program (that can read and write), in which the problem lies on the SDRAM itself. In this sub-chapter we would describe the detail of SDRAM problem.

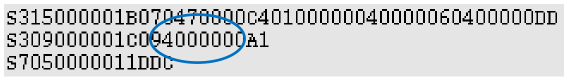
To begin with, it is important to remember that one of the reasons to build compressed main memory system is to increase memory bandwidth (more bytes/second) so that it may increase system processing speed when off-chip RAM is used because Off-Chip RAM is slower than on-chip RAM. In the previous work, SDRAM was used as main ram to store initial program and data. Before the system runs, SDRAM must firstly be loaded (initialized) with memory initialization file (.srec extension) which contains our desired Cortex-M0 program.



/testbench/cpu/arm_wrapper/cpu(0)/cm0_sdram_wrapper_inst/clk 0
/testbench/cpu/arm_wrapper/cpu(0)/cm0_sdram_wrapper_inst/rst 1
LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0_sdram_wrapper_inst/led3 0
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0_sdram_wrapper_inst/hready 0
/testbench/cpu/arm_wrapper/cpu(0)/cm0_sdram_wrapper_inst/htrans 0
/testbench/cpu/arm_wrapper/cpu(0)/cm0_sdram_wrapper_inst/haddr 0000007E
/testbench/cpu/arm_wrapper/cpu(0)/cm0_sdram_wrapper_inst/hrdata 00000000
/testbench/cpu/arm_wrapper/cpu(0)/cm0_sdram_wrapper_inst/hwrite 0
/testbench/cpu/arm_wrapper/cpu(0)/cm0_sdram_wrapper_inst/hwdata 00FFFF00
DDR SDRAM
/testbench/cpu/ddrsp0/ddrc0/ahbsi.hready 0
/testbench/cpu/ddrsp0/ddrc0/ahbsi.haddr 0000007E
/testbench/cpu/ddrsp0/ddrc0/ahbsi.hwrite 0
/testbench/cpu/ddrsp0/ddrc0/ahbsi.hwdata 00FFFF00
/testbench/cpu/ddrsp0/ddrc0/ahbso.hready 0
/testbench/cpu/ddrsp0/ddrc0/ahbso.hrdata 00400000
/testbench/cpu/ddrsp0/ddrc0/ddr_dq 00000000
Now
Cursor 1
000001C0
00000000
0000007E
00000082
00690000
00006900
60400000
00D0FB1A
6600 ns
6800 ns
6652.297 ns

Figure 6.2-9: Simulation of program executing startup file when SDRAM was used.

However, we came to know that when we initializes SDRAM with program containing startup.s file, not all desired SDRAM memory addresses are initialized properly. For instance, in figure 6.2-9, we can observe at red circle that CPU requested data from address 0x000001C0. Few clock cycles after that, we can see that the incoming data from aforementioned address was 0 as marked by yellow circle. As we traced back to the SDRAM ports shown by purple circle, the data associated to the address sent out by memory is really 0. This is a surprise to us because if we open the memory initialization file (.srec) of that particular program using text editor and search the content of the corresponding address as shown in figure 6.2-10, we see that the data that was supposed to reside on that address was 0x00000094 instead of 0 (note that figure 6.2-10 shows data in big-endian form). This clearly shows that there was a mismatch between what was intended to be put into the memory with what was actually placed in the memory.



```
S315000001B070470000C4010000004000006040000000DD
S309000001C094000000A1
S7050000011DDC
```

Figure 6.2-10: Few last lines of SDRAM initialization file (sdram.srec) for simulation figure 6.2-9.

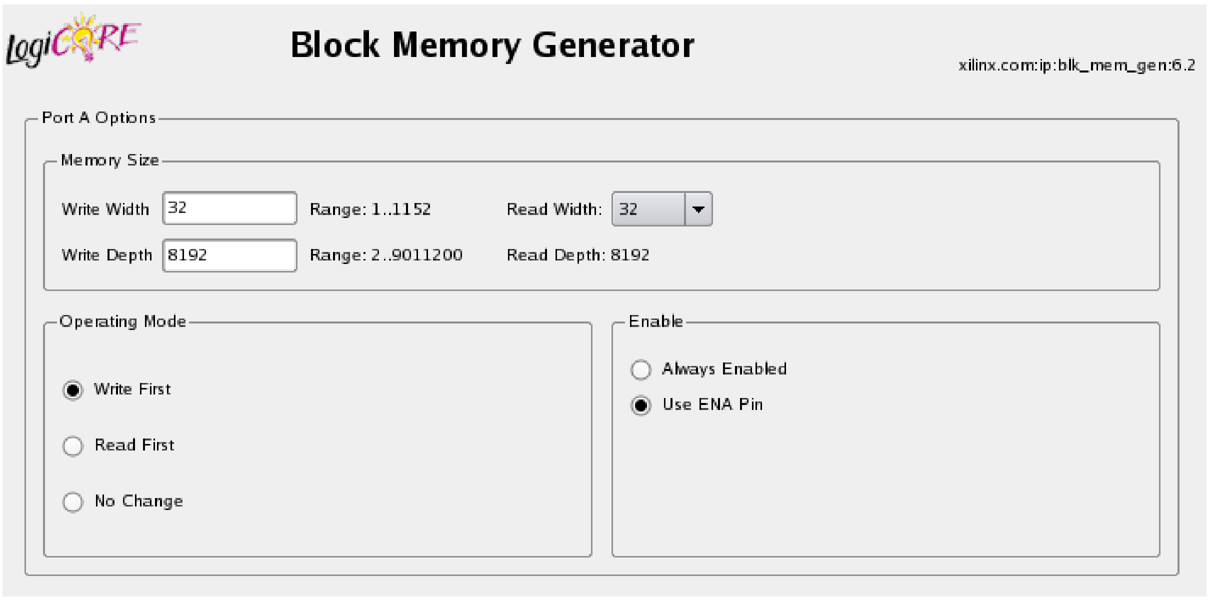
The cause of this problem is still unclear, however, we suspect that the problem may be originated from the GRLIB IP Library SoC itself in translating .srec extension file or the process of creating .srec file in Keil MDK itself is flawed. The first suspicion seems to be more accurate since we tried to create the SDRAM initialization file with different software other than Keil MDK yet it still did not solve the problem.

Because there was nothing much we could do with the problem related to SDRAM, we use alternative solution which is to use on-chip RAM that is called as AHBRAM (on-chip RAM with AHB interface). Nonetheless, AHBRAM is obviously very fast and using it as it is deflects one of the purposes of conducting our project, in which to see how much improvement gain we can obtain between normal system and our system when off-chip RAM is used. Therefore, we need to add a mechanism in our system to make the AHBRAM behaving like SDRAM. This SDRAM-based system behavior modeling on AHBRAM would be carefully explained in the later sub-chapter.

6.2.5 AHBRAM as Main Memory

Because of the SDRAM problem we are having, we decided for our project to utilize AHBRAM instead of SDRAM as main memory for the basic transfer Cortex-M0 system and compressed main memory system we are developing. AHBRAM is inherently an on-chip RAM having AHB slave interface provided by GRLIB IP Library SoC written in VHDL code (under the file name ahbram.vhd). The default AHBRAM module is just a top-level module having a mechanism to automatically generate on-chip RAM (called "syncram") according to user input configuration. However, this automatically generated syncram is not easy to be initialized to contain Cortex-M0 program, thus, we modified the AHBRAM module by replacing syncram with our own on-chip RAM that we called as "main_ram".

Main_ram (under the file name of main_ram.vhd) is a single port RAM module having 32768 bytes size we created using Xilinx Block Memory Generator. It is important to notice that the process of creating RAM in Xilinx also involves process of generating the memory initialization file (.mif extension) for the RAM itself. This .mif file which is generated during RAM creation process may contain Cortex-M0 program we designed using Keil MDK. This also implies that every time we would like to generate .mif file for different Cortex-M0 program, we have to undergo the process of creating main_ram using Xilinx Block Memory Generator, which is time consuming. Our specifications for creating main_ram are described by figure 6.2-11 below.



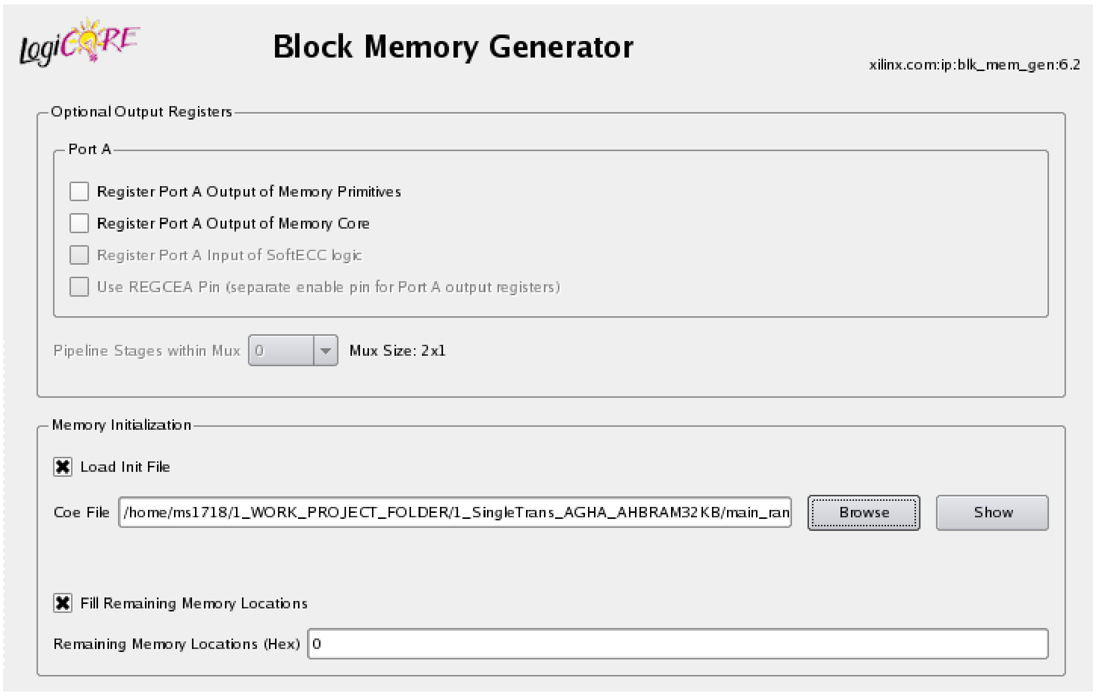
Write Width 32 Range: 1..1152 Read Width: 32
Write Depth 8192 Range: 2..9011200 Read Depth: 8192

Operating Mode: [✓ Write First] Read First No Change

Enable: Always Enabled [✓ Use ENA Pin]

Figure 6.2-11: Single port main_ram configuration in Xilinx Block Memory Generator

From figure above, notice that Write Depth = 8192. This is because write depth refers to number of address slots that we need because 8192 x 4 bytes = 32768 bytes. Write and read width is 32 because we CPU is working with 32 bits wide of data. Write first operating mode means that whatever being written to memory simultaneously appears in the output port of memory to be read. We use this mode because we found that there were glitches if we use other modes.


Block Memory Generator

xilinx.com:ip:blk_mem_gen:6.2

Optional Output Registers

Port A

- [ ] Register Port A Output of Memory Primitives
- [ ] Register Port A Output of Memory Core
- [ ] Register Port A Input of SoftECC logic
- [ ] Use REGCEA Pin (separate enable pin for Port A output registers)

Pipeline Stages within Mux: 0 Mux Size: 2x1

Memory Initialization

- [x] Load Init File

Coe File: /home/ms1718/1_WORK_PROJECT_FOLDER/1_SingleTrans_AGHA_AHBRAM32KB/main_ran

- [x] Fill Remaining Memory Locations

Remaining Memory Locations (Hex): 0

Figure 6.2-12: Loading .coe file to to create .mif file

Figure 6.2-12 shows the block memory generator page where we load the file to create .mif file. The file used to create .mif file is called .coe file. This .coe file is obtained by converting binary file of Cortex-M0 program obtained from Keil MDK.

After process of creating main_memory is finished, we will find three important files which are main_memory.vhd, main_memory.mif and main_memory.ngc. main_memory.vhd and main_memory.mif are put in the modelsim project simulation folder. Main_memory.ngc is used for Xilinx ISE implementation.

Afterwards, we need to compile main_memory.vhd in the modelsim so that it can be used for simulation. Nevertheless, because main_memory.vhd is used under ahbram.vhd which is belong to GRLIB library, we need to compile it uniquely by entering command (vcom) in modelsim transcripts which is:

vcom –work gaisler main_ram.vhd

However, before we compile main_ram, we need to make sure that we have added XilinxCoreLib to the modelsim. One way to add it easily is also by entering modelsim command (vmap) with the library path. For example:

vmap XilinxCoreLib /app/ise132/ISE_DS/ISE/vhdl/mti_se/6.6d/lin64/xilinxcorelib

(example used for modelsim in University of Bristol een5021 linux server)

In addition, we also have to modify config.vhd file to change parameter AHBRSZ = 32 so that AHBRAM would work with our 32KB size main_ram.

6.2.6 Modeling SDRAM-based System performance on AHBRAM-based System

We have described that AHBRAM is used to replace SDRAM because of the problem occurred in SDRAM. However, AHBRAM as on-chip RAM is too fast and therefore we need to provide a mechanism so that AHBRAM would perform as slow as SDRAM.

As we discussed in chapter 4.1.1 regarding basic CPU transfer of ARM Cortex-M0DS, we know that every transfer consist of address phase and data phase. In the case of using AHBRAM, data phase would last as fast as only a single clock cycle, which is clearly depicted by figure 4.1-2 in that sub-chapter. In the other hand, as can be seen from figure 6.2-13 below, data phase of SDRAM lasts at least 7 clock cycles, which is 7 times slower than AHBRAM.

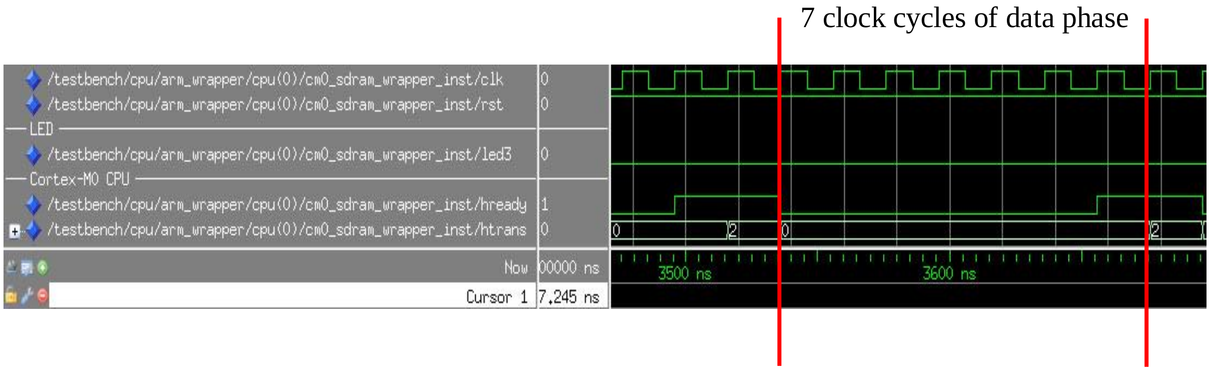

7 clock cycles of data phase
/testbench/cpu/arm_wrapper/cpu(0)/cm0_sdram_wrapper_inst/clk 0
/testbench/cpu/arm_wrapper/cpu(0)/cm0_sdram_wrapper_inst/rst 0
LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0_sdram_wrapper_inst/led3 0
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0_sdram_wrapper_inst/hready 1
/testbench/cpu/arm_wrapper/cpu(0)/cm0_sdram_wrapper_inst/htrans 0
0
2
0
2
Now 00000 ns
Cursor 1 7.245 ns
3500 ns
3600 ns

Figure 6.2-13: Simulation of SDRAM-based system, which highlights number of clock cycles during data phase.

Therefore, now we know that we need to extend the data phase of AHBRAM-based system to be as long as SDRAM-based system so that our system performs as it was using SDRAM. The mechanism to extend the data phase in order to mimic SDRAM-based system performance requires additional module to be designed called “bus_delay_controller”.

The addition of bus_delay_controller opens wider possibility for us to model not only SDRAM performance but also any other memory performance since we can control the data phase delay of a transaction to be as long as we want.

Bus_delay_controller module would work with our wrapper design which performs basic transfer and integrates only Cortex-M0 CPU (CM0_wrapper) but not with wrapper that integrates compressed main memory system (CM0X_wrapper). This means that, while CM0_wrapper would perform as it was using SDRAM, CM0X_wrapper still perform as fast as it is using AHBRAM. Hence, performance comparison between two wrappers may not be possible to be conducted.

Fortunately we have another alternative to model SDRAM system performance for CM0X_wrapper, in which we ran simple and predictable Cortex-M0 program in CM0X_wrapper, then, we estimated the CM0X_wrapper performance as if it was using SDRAM by means of mathematical calculation.

Because the method of modeling SDRAM system performance are different between CM0_wrapper and CM0X_wrapper, we would discuss each of the method more detail in the later sub-chapters.

6.3 CM0_wrapper Design

6.3.1 CM0_wrapper Design Process

This sub-chapter explains the second phase of designing compressed main memory system. In this phase, we designed a wrapper which integrated Cortex-M0 CPU into GRLIB SoC. We call this particular wrapper as CM0_wrapper. The purpose of this wrapper is to build basic transfer Cortex-M0 SoC as a reference comparison to the compressed main memory system we developed in this project.

As explained in sub-chapter 4.2, GRLIB SoC was chosen for our project because it utilizes AHB (Advanced High-Performance Bus) as its main bus system which is compatible for Cortex-M0. Thus, the key of integrating Cortex-M0 to the SoC resides on the interfacing between Cortex-M0 to AHB of the SoC.

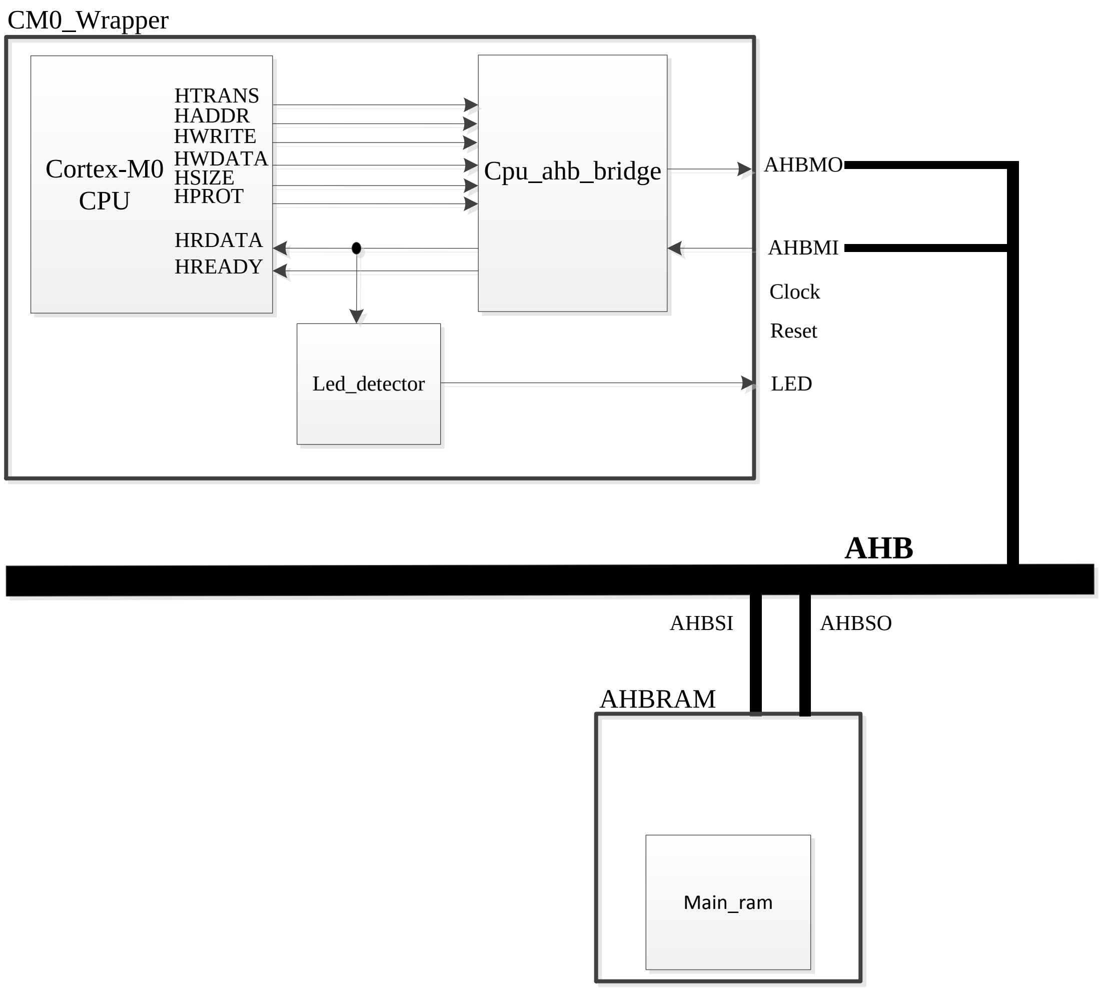


Figure 6.3-1: Simplified schematic of GRLIB SoC with CM0_wrapper

As can be seen from figure 6.3-1, CM0_wrapper is connected as master to AHB through AHBMI (MI stands for "Master Input") and AHBMO (MO stands for Master Output). In the other hand, AHBRAM as a slave is connected to AHB through AHBSI (slave input) and AHBSO (slave output). Notice that AHBRAM as top-level module contains main_ram module we created in previous sub-chapter. Not forget to mention that every module in the wrapper is connected to clock and reset port, however, the connections are not shown to provide better visibility of the schematic.

AHBMO and AHBMI are both declared as record types in VHDL code which means each of them contains one or more ports (think like structures in C code). AHBMO contains all necessary ports to interface Cortex-M0 output ports listed in Cortex_M0 box in the figure such as ahbmo.htrans, ahbmo.haddr, ahbmo.hwrite, ahbmo.hwdata, ahbmo.hsize, ahbmo.hprot. Similarly, AHBMI contains ports to connect to HRDATA and HREADY ports of Cortex-M0 which are ahbmi.hready and ahbmi.hrdata. All of these ports basically have been explained in sub-chapter 4.1.1 except HPROT and HSIZE. HPROT is protection port that indicates whether a transfer contain data or instruction. HSIZE indicates size of transfer whether it is byte, byte, half word or word.

In addition, AHBMO also contains very important request port namely ahbmo.hbusreq that needs to be triggered by CPU in order for CPU to request transaction to the bus. As can be seen from the figure 6.3-1, Cortex-M0 CPU does not have interface for ahbmo.hbusreq. This is why we need to build a mechanism in triggering ahbmo.hbusreq according to CPU actions.

Cpu_ahb_bridge was designed to accommodate the mechanism of triggering ahbmo.hbusreq port. The design of the module is originally very much simpler than one designed in the previous work, in which previously the wrapper contained four states of state machine and an interfacing module provided by GRLIB SoC called AHBMST[40]. We removed all of those because we properly understand the AHB interface and it results to reduced complexity since we have only cpu_ahb_bridge module. In addition, we coded cpu_ahb_bridge using “two processes” VHDL coding style [45], in which we registered all necessary ports so that those ports are synchronized with clock and latches are easily avoided.

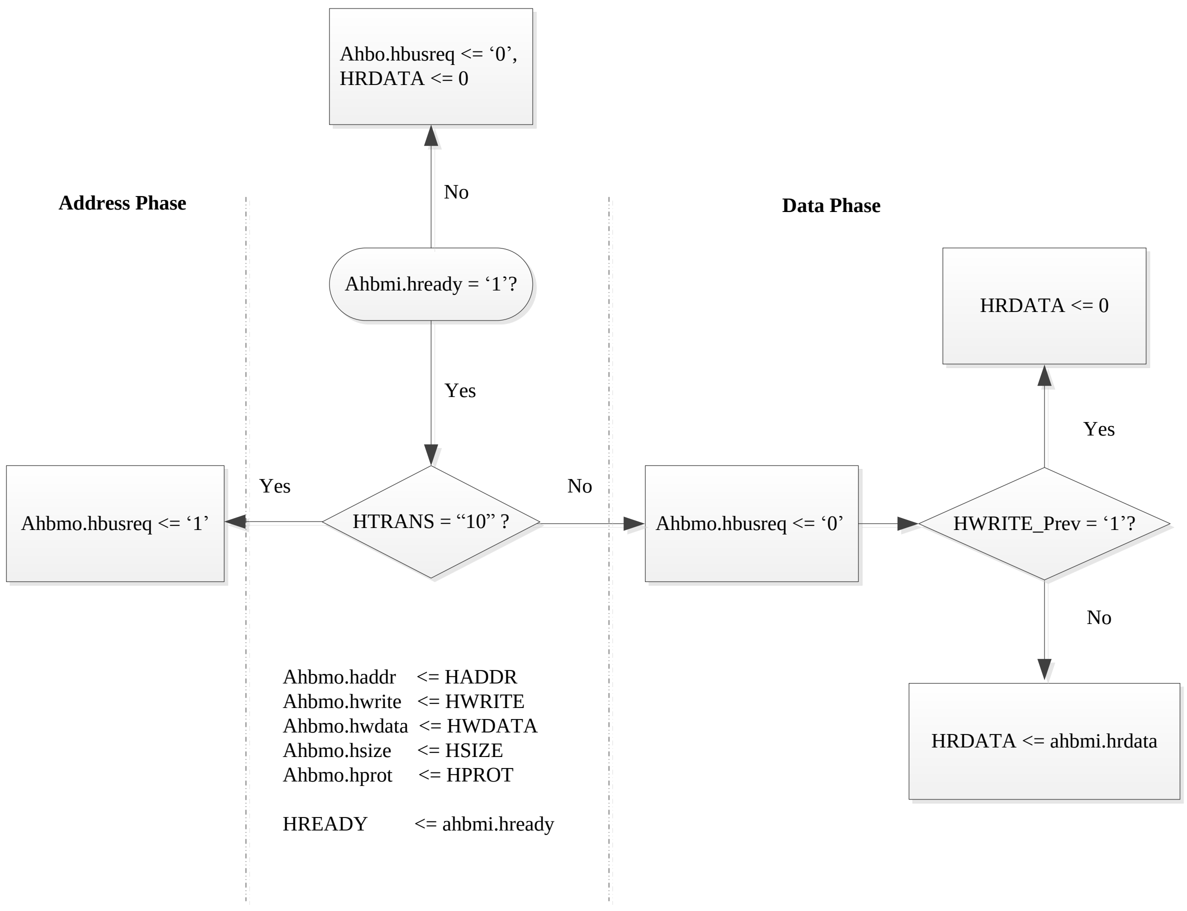


Figure 6.3-2: Flow chart of cpu_ahb_bridge module.

The flow chart in figure 6.3-2 shows the interfacing mechanism inside cpu_ahb_bridge module we designed. It can be seen that we only control ahbmo.hbusreq and HRDATA ports. Other Cortex-M0 ports such as HADDR, HREADY etc. are directly connected to AHB ports as shown in the figure 6.3-2. Ahbmi.hready is a port which indicates if AHB is ready to be used or not. If it is ready (value is '1'), it automatically activates the Cortex-M0 since it is connected to HREADY.

As we have discussed previously, every Cortex-M0 transfer consist of address phase and data phase. Once Cortex-M0 is ready, it is initially in idle state as indicated by HTRANS = "00", in which also means that it is performing data phase. After sometime, Cortex-M0 eventually starts to initiate a transaction (performing address phase) by driving HTRANS = "10" and Cortex-M0 automatically transfers address (and also other address phase parameters) to AHB since HADDR is already connected to ahbmo.haddr. At this time, ahbmo.hbusreq is driven high to indicate that the bus is being requested for CPU transaction.

When HTRANS turns back to "00", Cortex-M0 performs data phase and therefore removes the bus request. At this data phase, Cortex-M0 receives data from bus depending on signal HWRITE_Prev which detects whether signal HWRITE during address phase is 0 or 1. It also indicates that the current data phase is actually for read transfer of write transfer. If HWRITE_Prev is 0 then data phase is read transfer and therefore data from the bus is allowed to enter HRDATA of Cortex-M0. If HWRITE_Prev is 1 then data phase is write transfer and therefore HRDATA is supplied with 0. This HWRITE_Prev mechanism is important to be added as it prevents glitches when we are using very fast memory.

In addition, led_detector is a module developed by Martos from University of Buenos Aires [34] to detect data pattern 0xAAAA5555 (to turn on LED) or 0xF0F0F0F0 (to turn off LED) passing through HRDATA, which triggers LED on the FPGA board in order to verify the functionality of the CPU.

CM0_SDRAM_wrapper we briefly discussed in previous sub-chapter is basically the same with CM0_wrapper. The difference only is that there is no mechanism on converting big-endian to little-endian or vice versa in CM0_wrapper because instructions and data stored in main_ram are already in form of little-endian.

The VHDL code of this wrapper design can be observed in submitted zip file *:* "Sourcecodes_forassessment_ms1718\CompletelyDesigned by AGHA\CM0_wrapper files"

6.3.2 Modeling SDRAM-based System Performance with CM0_busdelaycont_wrapper

In sub-chapter 6.2.6, we have briefly discussed about the motivation of modifying our system to perform as it was using SDRAM. We also have briefly mentioned that we need to create an additional module called bus_delay_controller to delay the data phase of AHBRAM-based system.

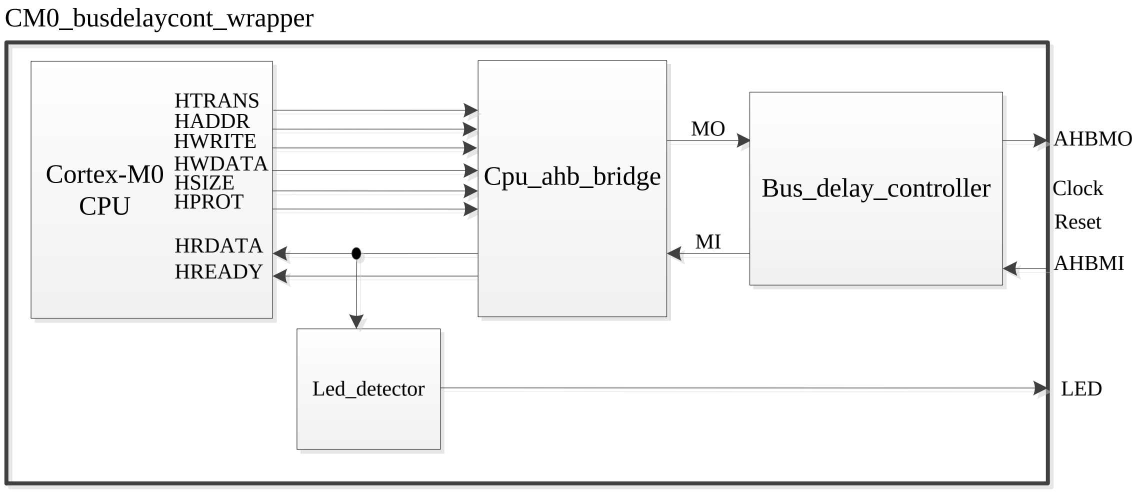


Figure 6.3-3: Simplified schematic of CM0_busdelaycont_wrapper.

As shown in figure 6.3-3, we designed a new wrapper for basic transfer Cortex-M0 system that also integrated bus_delay_controller inside. This wrapper is used only for simulation in order to conduct research of our project. It is important to know that nothing has been changed inside cpu_ahb_bridge. From the figure above, we notice that cpu_ahb_bridge is connected to bus_delay_controller via MI (Master In) and MO (Master Out) signals. These signals are record type signals which have exactly same structure as AHBMI and AHBMO.

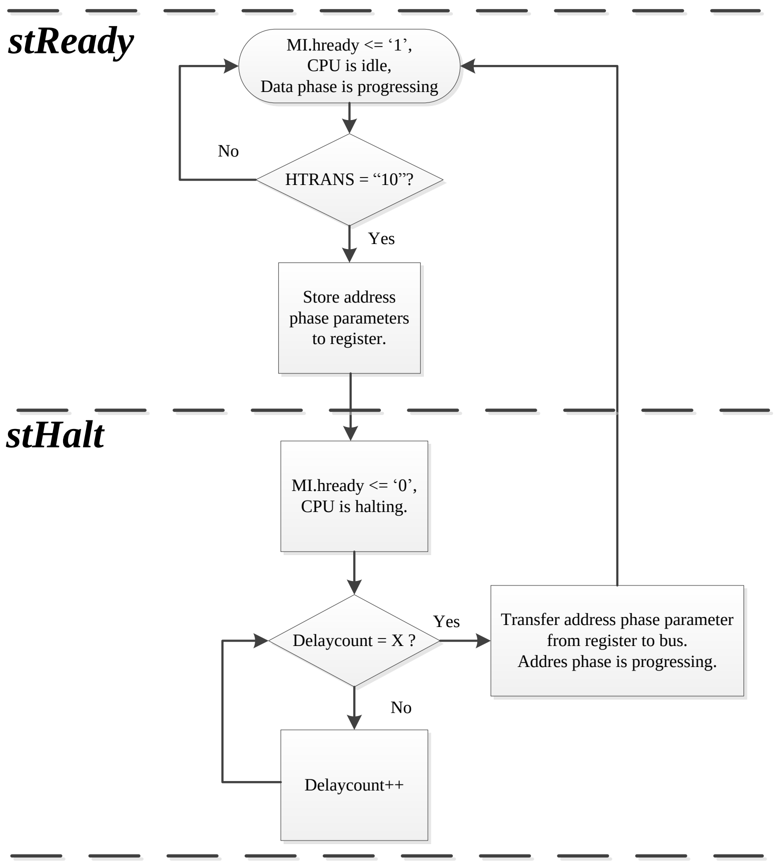

stReady
MI.hready <= '1', CPU is idle, Data phase is progressing
No
HTRANS = "10"?
Yes
Store address phase parameters to register.
stHalt
MI.hready <= '0', CPU is halting.
Delaycount = X ?
Yes
Transfer address phase parameter from register to bus. Addres phase is progressing.
No
Delaycount++

Figure 6.3-4: Flow chart of bus_delay_controller state machine.

As can be seen from figure 6.3-4, the state machine in bus_delay_controller inherently has two states which are stReady and stHalt. StReady is the state whereby Cortex-M0 is idle and accepting data phase while at some point it will perform address phase. The state changes to stHalt after Cortex-M0 has initiated a transaction by performing address phase indicated by HTRANS = “10”, in which also storing address phase parameter like HBUSREQ, HADDR, HWRITE, HPROT, etc. to the register.

In stHalt, MI.hready is driven to zero, thus, Cortex-M0 is halting and not performing address phase neither data phase. In this state, there is a counter (delaycount) counting every clock cycle until limit X is reached. *X is determined by desired number of clock cycles of data phase duration minus 2.* After X is reached by delaycount, address phase parameter which were previously stored into the register, are transferred to the bus (AHBMO), in which a clock cycle after that the state changes to stReady and data from bus arrives at CPU (or bus accepting data to be written) performing data phase.

By understanding the flow chart, we understand that what actually happens to delay the data phase of the CPU is that we delay the address phase parameter issuance to the bus. This is because when we are using AHBRAM, the data phase will be executed just right after the address phase execution since AHBRAM is an on-chip memory (very fast).

The bus_delay_controller module is also considered a very useful innovation to us since now we can model various memory performances by just configuring the delay of the data phase (delaycount). To prove that our bus_delay_controller works, we provide figure 6.3-5 and 6.3-6 that show our wrapper performs in two different memory performances.

In addition, now we know that we have two wrapper designs for basic transfer Cortex-M0 system which are CM0_wrapper and CM0_busdelaycont_wrapper. The reason of having these two separate wrappers is that CM0_wrapper is used for FPGA implementation since it represents the ideal system of basic transfer Cortex-M0 system regardless any memory system we use in the SoC. If the SDRAM problem is fixed in the future work, CM0_wrapper can be used directly for both simulation and implementation. In the other hand, CM0_busdelaycont_wrapper is just a wrapper only for simulation purpose to support our research and temporarily overcome the problem of the SDRAM. We cannot use CM0_busdelaycont_wrapper for FPGA implementation (although it still works) because it introduces additional overhead in the FPGA caused by bus_delay_controller module which should not exist in the real system (since this module is just to temporarily overcome SDRAM problem for research purpose only). Having to use CM0_busdelaycont_wrapper for implementation may result to inaccurate reference comparison to the compressed main memory system we are developing in term of power consumption, logic utilization in FPGA and etc.

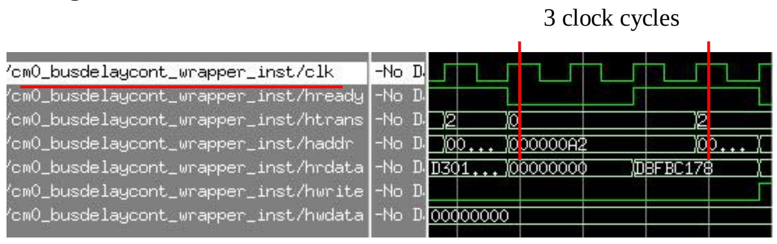

3 clock cycles
/cm0_busdelaycont_wrapper_inst/clk
/cm0_busdelaycont_wrapper_inst/hready
/cm0_busdelaycont_wrapper_inst/htrans
/cm0_busdelaycont_wrapper_inst/haddr
/cm0_busdelaycont_wrapper_inst/hrdata
/cm0_busdelaycont_wrapper_inst/hwrite
/cm0_busdelaycont_wrapper_inst/hwdata
2
0
00...
000000A2
D301...
00000000
DBFBC178
00000000

Figure 6.3-5: CM0_busdelaycont_wrapper having 3 clock cycles of data phase. Limit X = 1.

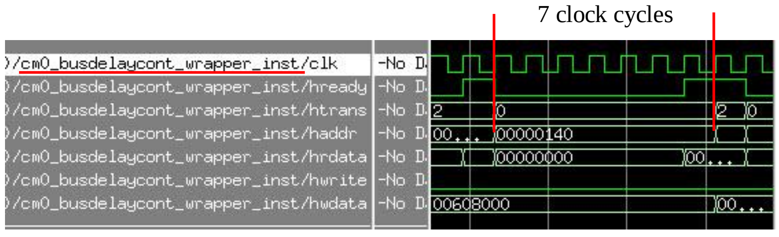

7 clock cycles
/cm0_busdelaycont_wrapper_inst/clk
/cm0_busdelaycont_wrapper_inst/hready
/cm0_busdelaycont_wrapper_inst/htrans
/cm0_busdelaycont_wrapper_inst/haddr
/cm0_busdelaycont_wrapper_inst/hrdata
/cm0_busdelaycont_wrapper_inst/hwrite
/cm0_busdelaycont_wrapper_inst/hwdata
2
0
00...
00000140
00000000
00608000

Figure 6.3-6: CM0_busdelaycont_wrapper having 7 clock cycles data phase. Limit X = 5.

6.4 CM0X_wrapper Design

6.4.1 Overview of CM0X_wrapper Design

Finally we have come into the third phase of the project which is the main design we have been developing in this project. The wrapper presented in this sub-chapter represents the design of compressed main memory system, in which we call the wrapper as CM0X_wrapper. The 'X' in "CM0X" refers to X-MatchPROVW compression-decompression engine that we use in this project.

Before we begin to discuss the technical part of the wrapper, it is important to understand the design concept of our compressed main memory system. In our system, types of information we decide to be compressed and decompressed are instructions and data in heap memory region. Stack data is not compressed or decompressed because stack data traffic is very dynamic. In other words, since data in stack regions changes frequently, large overhead may occur if we compress or decompress stack data on-the-fly because the compression-decompression engine might be overworked unnecessarily.

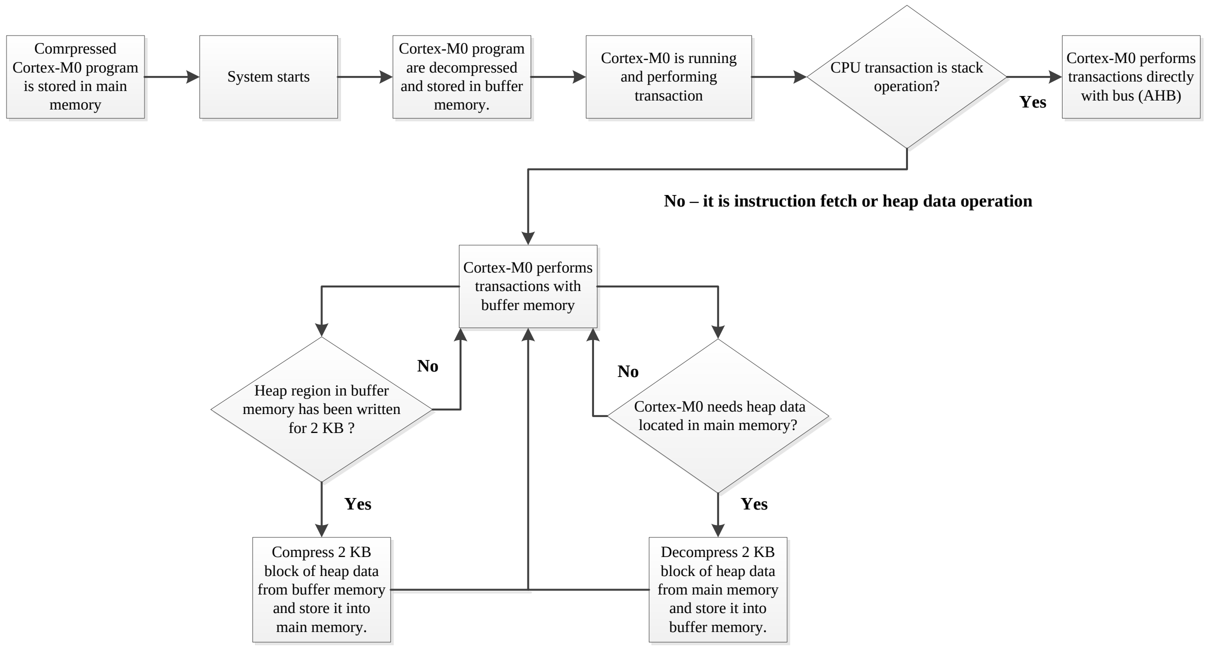

Comrpressed Cortex-M0 program is stored in main memory
System starts
Cortex-M0 program are decompressed and stored in buffer memory.
Cortex-M0 is running and performing transaction
CPU transaction is stack operation?
Yes
Cortex-M0 performs transactions directly with bus (AHB)
No – it is instruction fetch or heap data operation
Cortex-M0 performs transactions with buffer memory
No
Heap region in buffer memory has been written for 2 KB ?
No
Cortex-M0 needs heap data located in main memory?
Yes
Yes
Compress 2 KB block of heap data from buffer memory and store it into main memory.
Decompress 2 KB block of heap data from main memory and store it into buffer memory.

Figure 6.4-1: Simplified flow chart of our compressed main memory system working principle.

As can be seen from figure 6.4-1, the working principle of our compressed main memory system is that instructions (Cortex-M0 program) are initially stored in main memory (AHBRAM) in compressed state. When system is just started, compressed instructions are fetched from the main memory block by block (in which in our research we set the compression block to be two kilo bytes) and decompressed on-the-fly by X-MatchPROVW engine.

After on-the-fly decompression is finished, the decompressed instructions are then stored into buffer memory. Afterwards, Cortex-M0 just performs transactions with buffer memory for instructions fetch and data heap operations. Stack operation in which the data is left uncompressed, bypasses the X-MatchPROVW engine so that the operation only involves Cortex-M0 and AHB.

The on-the-fly compression takes part when heap region in buffer memory has been written for 2 KB. Afterwards, when Cortex-M0 requests heap data that is located in main memory, 2 KB block of heap data is decompressed and placed into buffer memory so that Cortex-M0 can fetch the heap data.

Since our focus in this project is to build working compressed main memory system for research purpose only, we designed the system in a simple manner in which the process of on-the-fly compression-decompression and managing the compressed blocks are handled in straightforward manner. This means that, we dictate the system of when it should perform on-the-fly compression and when it should perform on-the-fly decompression. We avoided the use of sophisticated translation table due to time constraints and our objective is still achievable without it (as clarified in sub-chapter 3.5). In addition, there was huge portion of time spent to "locate" and "fix" technical problem occurred in the previous work, which then we developed working basic transfer Cortex-M0 SoC that is already considered as a breakthrough in this project.

As seen in figure 6.4-1, initially program is stored in AHBRAM in compressed state, this implies that we need to find a way to compress the Cortex-M0 program outside the system and place it inside main_ram of AHBRAM. To do this, we have developed a wrapper that contains only X-MatchPROVW called "comp_xmatchstandalone" in purpose of compressing the Cortex-M0 program to be put into main_ram (or we call it offline compressing). This wrapper is not going to be explained but can be found in the submitted zip file: "\Sourcecodes forassessment ms1718\CompletelyDesigned by AGHA\XMatchPROVW standaloneCompression wrapper files"

6.4.2 Schematic of CM0X_wrapper Design

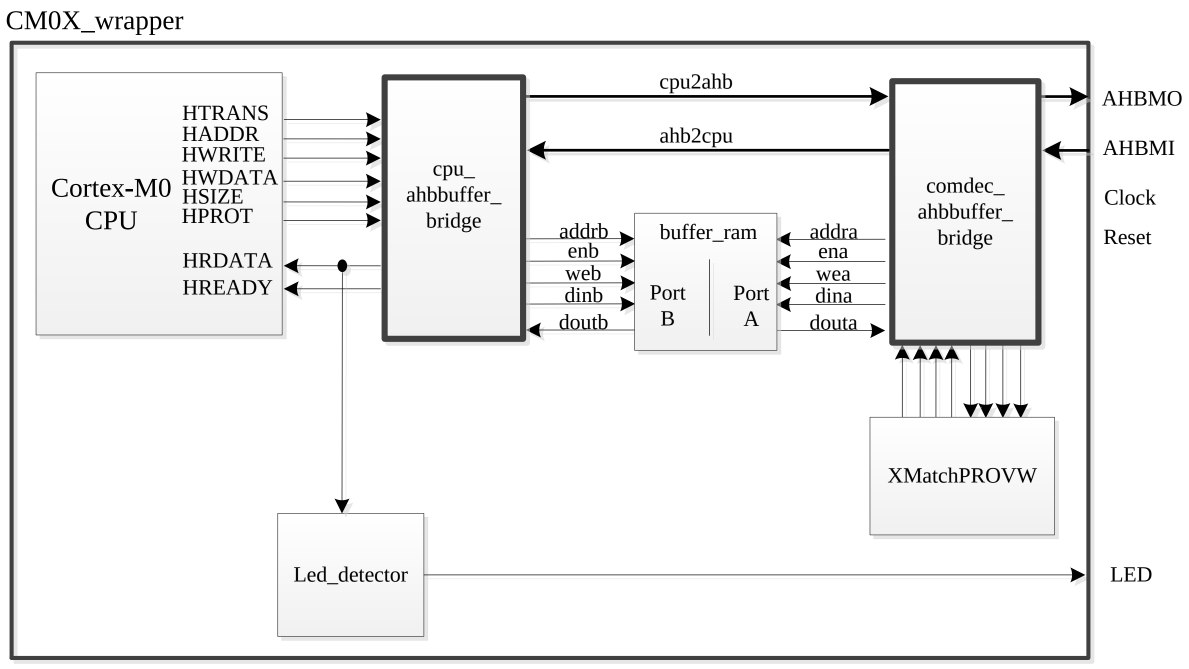


Figure 6.4-2: Simplified schematics of CM0X_wrapper

Figure 6.4-2 above illustrates CM0X_wrapper schematic, in which its signals description is simplified. Cortex-M0 CPU and Led_detector are still inside the wrapper while there are four new modules exist in the wrapper. X-MatchPROVW module seen in the figure is the compression-decompression engine that is readily provided by Dr. Jose L. Nunez-Yanez. We do not show the ports on X-MatchPROVW to simplify the schematic since its ports are too many to be displayed. Modules we designed in this wrapper are comdec_ahbbuffer_bridge, buffer_ram and cpu_ahbbuffer_bridge.

The source code for CM0_wrapper can be found in submitted zip file : **"**Sourcecodes_forassessment_ms1718\CompletelyDesigned by AGHA\CM0_wrapper files"

6.4.3 Comdec_ahbbuffer_bridge Module Design

Comdec_ahbbuffer_bridge is a module we designed to bridge AHB, X-MatchPROVW, buffer_ram and cpu_ahbbuffer_bridge. The bridge's responsibilities are listed as follows:

1. To control X-MatchPROVW ports for performing on-the-fly compression and decompression
2. To interface AHB with X-MatchPROVW
3. To interface AHB with cpu_ahbbuffer_bridge
4. To control port A of buffer_ram and interface it with X-MatchPROVW ports.

In order to perform all the functionality above, we designed a state machine inside the module which has 9 states. Among these 9 states, 4 states are used to perform on-the-fly decompression, 4 states are used to perform on-the-fly compression, and there is one idle state called st_idle. The full diagram of the state machine is shown by figure 6.4-3. Before we discuss further about the mechanism of the state machine, it is very important to make sure if we already understand how compression and decompression function in X-MatchPROVW module works as explained in sub-chapter 4.3.

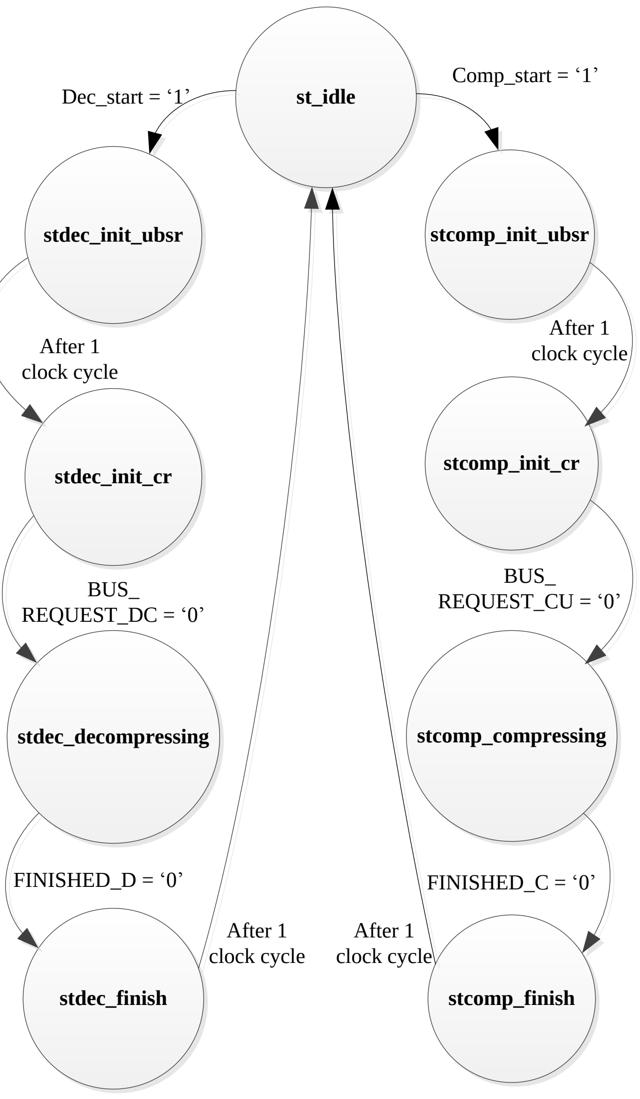


Figure 6.4-3: Comdec_ahbbuffer_bridge state machine diagram.

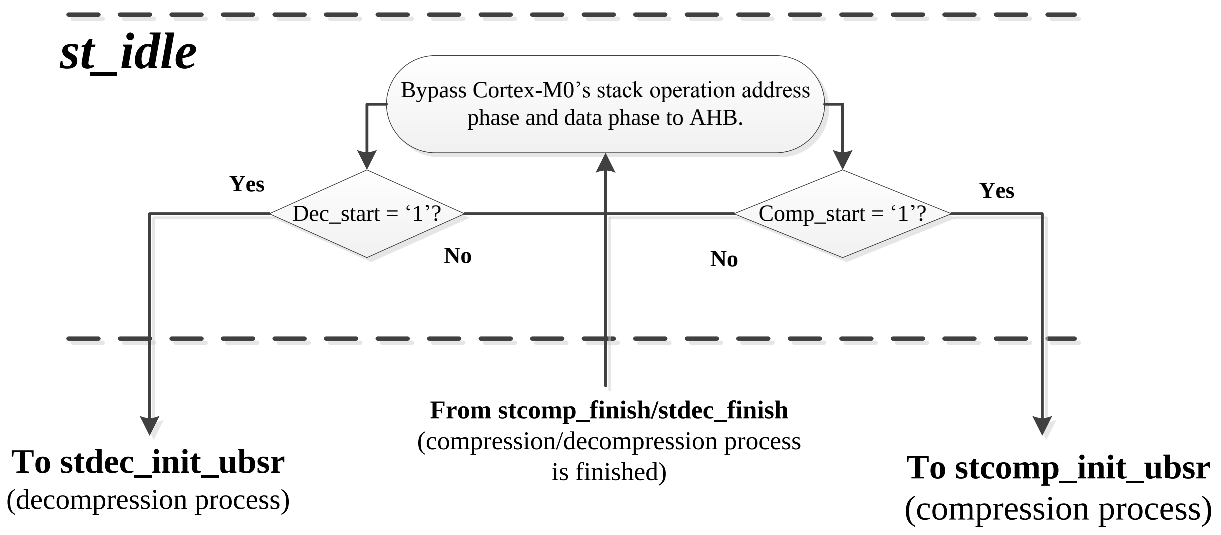

st_idle
Bypass Cortex-M0's stack operation address phase and data phase to AHB.
Yes
Dec_start = '1'?
No
No
Comp_start = '1'?
Yes
From stcomp_finish/stdec_finish
(compression/decompression process is finished)
To stdec_init_ubsr
(decompression process)
To stcomp_init_ubsr
(compression process)

Figure 6.4-4 Initial state in comdec_ahbbuffer_bridge state machine.

Figure 6.4-4 shows the initial state of the state machine in this module called st_idle. In this idle state, the module bypasses Cortex-M0 transaction for stack operation directly to AHB. As seen in figure 6.4-2, there are two signals bridging cpu_ahbbuffer_bridge and comdec_ahbbuffer_bridge which are cpu2ahb and ahb2cpu. These signals are interfaced directly to AHB during idle state to perform bypassing mechanism.

Other important function in this idle state is that it monitors two ports which triggers compression and decompression function. As shown in figure 6.4-4, dec_start is a port to indicate if there is a request for on-the-fly decompression from outside the module. If it is high, then state changes to perform decompression process. Similarly, comp_start is a port to monitor compression request.

The flow chart in figure 6.4-5 shows the on-the-fly decompression mechanism controlled in the state machine. The first two states shown above are the states to setup decompression function of X-MatchPROVW (to understand the engine's initialization process, please refer to sub-chapter 4.3.). In st_dec_init_cr, we initialize starting address to read from AHBRAM since we are going to fetch compressed data stored in AHBRAM. Also, we need to initialize starting address to write to buffer_ram because the resulting uncompressed data is written to port A of buffer_ram. The value of address initialization comes from outside of the module to support modularity of the design.

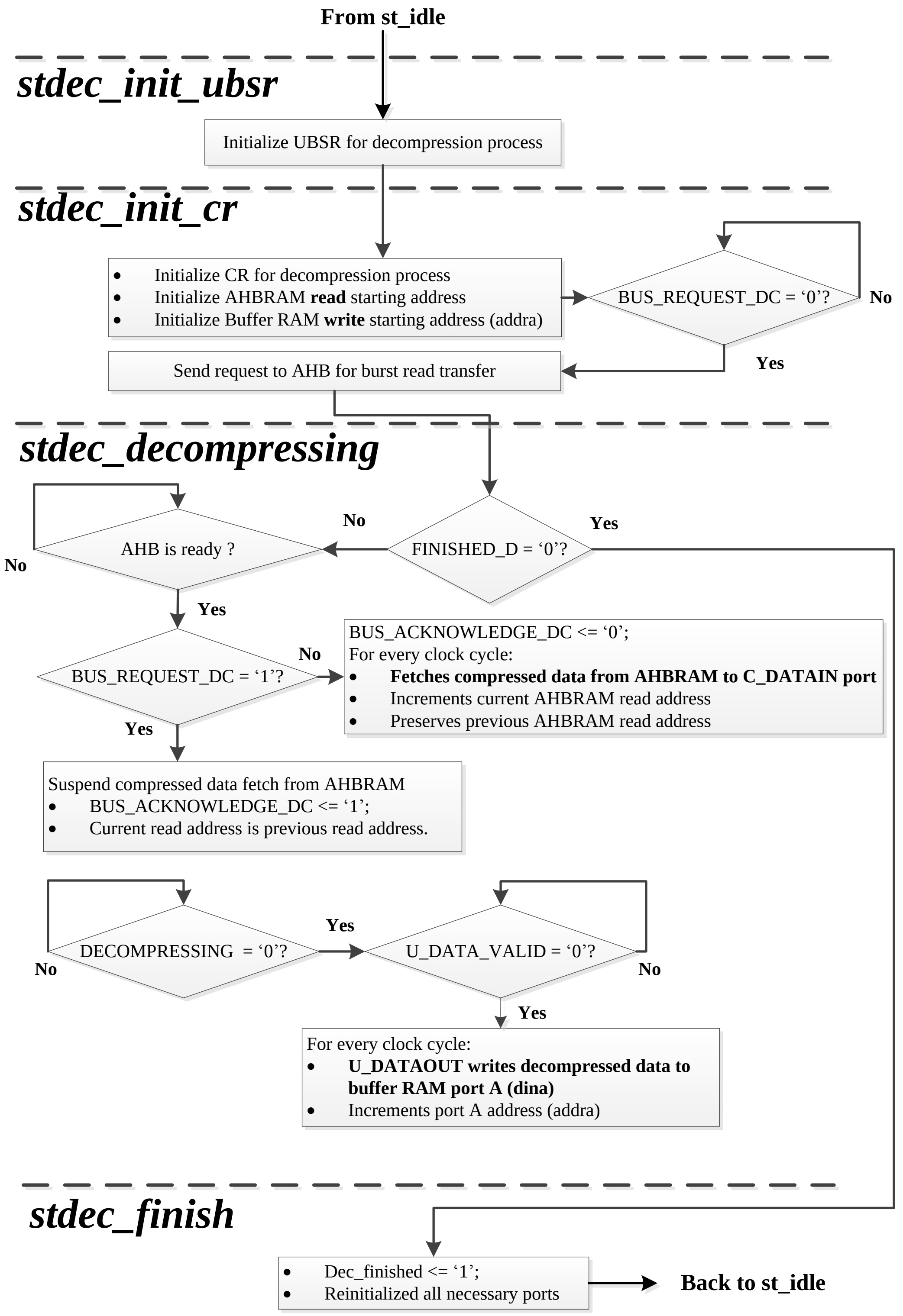

From st_idle
stdec_init_ubsr
Initialize UBSR for decompression process
stdec_init_cr
Initialize CR for decompression process
Initialize AHBRAM read starting address
Initialize Buffer RAM write starting address (addra)
BUS_REQUEST_DC = '0'?
No
Yes
Send request to AHB for burst read transfer
stdec_decompressing
AHB is ready ?
No
No
FINISHED_D = '0'?
Yes
Yes
BUS_REQUEST_DC = '1'?
No
BUS_ACKNOWLEDGE_DC <= '0';
For every clock cycle:
Fetches compressed data from AHBRAM to C_DATAIN port
Increments current AHBRAM read address
Preserves previous AHBRAM read address
Yes
Suspend compressed data fetch from AHBRAM
BUS_ACKNOWLEDGE_DC <= '1';
Current read address is previous read address.
DECOMPRESSING = '0'?
No
Yes
U_DATA_VALID = '0'?
No
Yes
For every clock cycle:
U_DATAOUT writes decompressed data to buffer RAM port A (dina)
Increments port A address (addra)
stdec_finish
Dec_finished <= '1';
Reinitialized all necessary ports
Back to st_idle

Figure 6.4-5: Flow chart of on-the-fly decompression states in comdec_ahbbuffer_bridge.

Decompression process starts when engine is ready to accept data from AHBRAM (indicated by BUS_REQUEST_DC = '0') and state changes to stdec_decompressing. In stdec_decompressing, there are two mechanisms that work in parallel. The first mechanism is to fetch compressed data from AHBRAM to C_DATAIN port every clock cycle once condition is satisfied. The second mechanism is to write uncompressed data from U_DATAOUT port of the engine to port A of buffer_ram every clock cycle. Once decompression process is finished, engine drives FINISHED_D port to zero and state changes to stdec_finish which then go back to st_idle.

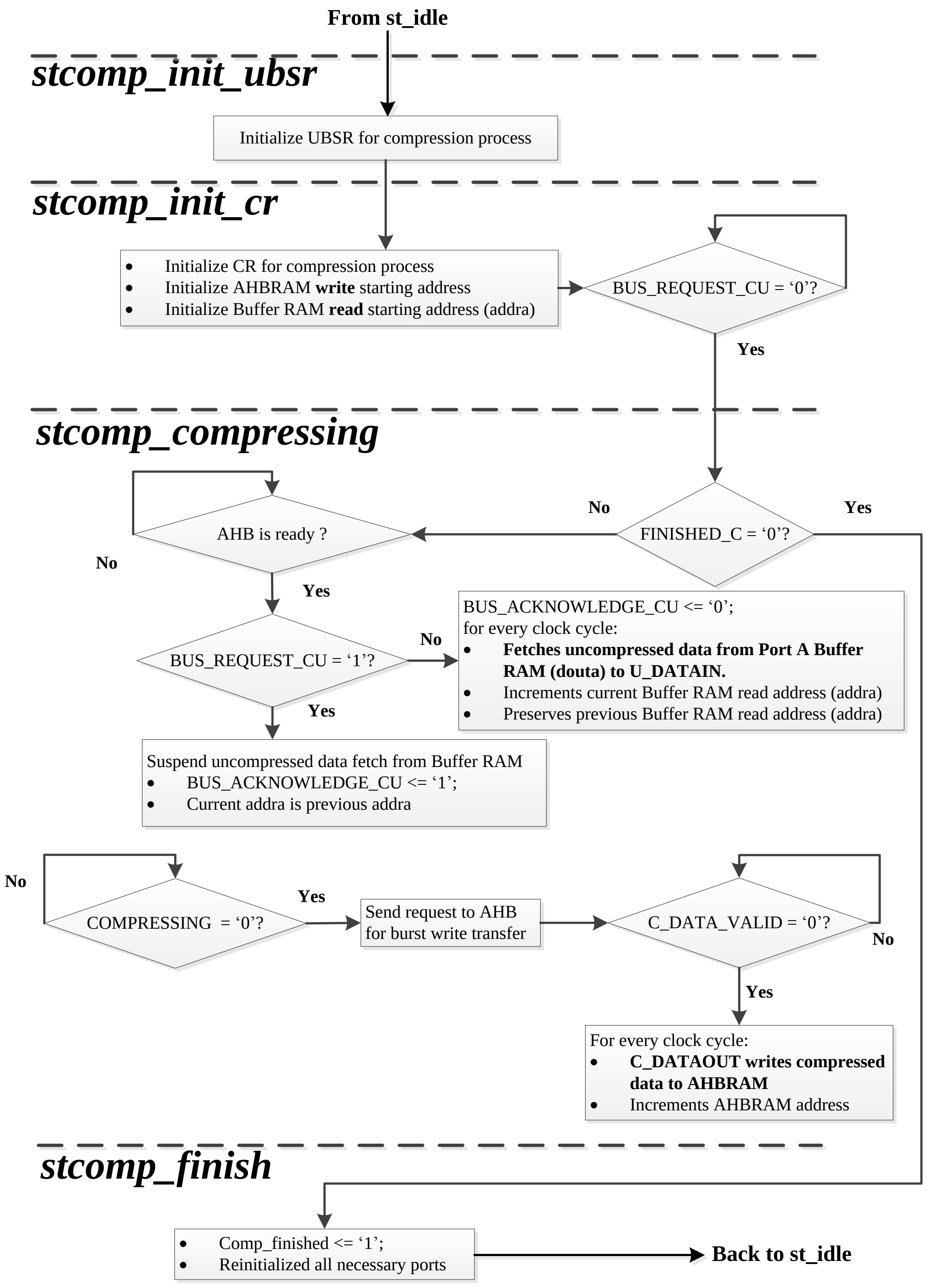

From st_idle
stcomp_init_ubsr
Initialize UBSR for compression process
stcomp_init_cr
Initialize CR for compression process
Initialize AHBRAM write starting address
Initialize Buffer RAM read starting address (addra)
BUS_REQUEST_CU = '0'?
Yes
stcomp_compressing
AHB is ready ?
No
No
FINISHED_C = '0'?
Yes
Yes
BUS_REQUEST_CU = '1'?
No
BUS_ACKNOWLEDGE_CU <= '0'; for every clock cycle: Fetches uncompressed data from Port A Buffer RAM (douta) to U_DATAIN. Increments current Buffer RAM read address (addra) Preserves previous Buffer RAM read address (addra)
Yes
Suspend uncompressed data fetch from Buffer RAM BUS_ACKNOWLEDGE_CU <= '1'; Current addra is previous addra
No
COMPRESSING = '0'?
Yes
Send request to AHB for burst write transfer
C_DATA_VALID = '0'?
No
Yes
For every clock cycle: C_DATAOUT writes compressed data to AHBRAM Increments AHBRAM address
stcomp_finish
Comp_finished <= '1'; Reinitialized all necessary ports
Back to st_idle

Figure 6.4-6: Flow chart of on-the-fly compression process in comdec_ahbbuffer_bridge.

On-the-fly compression procedure done in comdec_ahbbuffer_bridge has some similarities with the decompression procedure as can be seen in figure 6.4-6. The major difference we should notice here beside compression channel port being used is that the data is now fetched from buffer_ram which then compressed and stored into AHBRAM.

The source code for comdec_ahbbuffer_bridge can be found in submitted zip file : "Sourcecodes_forassessment_ms1718\CompletelyDesigned by AGHA\CM0X_wrapper files"

6.4.4 Cpu_ahbbuffer_bridge Module Design

Cpu_ahbbuffer_bridge is a module we designed to bridge CPU, buffer_ram and comdec_ahbbuffer_bridge. The bridge has several responsibilities as follows:

1. To control Cortex-M0 CPU (whether it should be running or halting)
2. To interface Port B buffer_ram with Cortex-M0.
3. To classify CPU operations whether it is instruction fetch operation, stack operation or heap operation.
4. To manage address and data phase of CPU whether the phase is supposed to operate with port B of buffer_ram or be bypassed to AHB through comdec_ahbbuffer_bridge.

In this module, a state machine is also required to perform all the functionalities above. The state machine in this module has 5 states as shown in figure 6.4-7. Although this state machine looks simple, it was not easy to come up with this design and the implementation of this state machine in term of coding it was not trivial.

The working principle of the state machine shown in figure 6.4-7 is that state changes between st_idle, st_read_inst, and st_readwrite_data after an address phase of either instruction fetch or data operation has been executed. Data operation in this context means stack operation, heap operation or other operation (that may be related to peripheral of SoC). In order to further understand the function of each state, we provide flow charts that describe each states as shown by figure 6.4-8, 6.4-9 and 6.4-10.

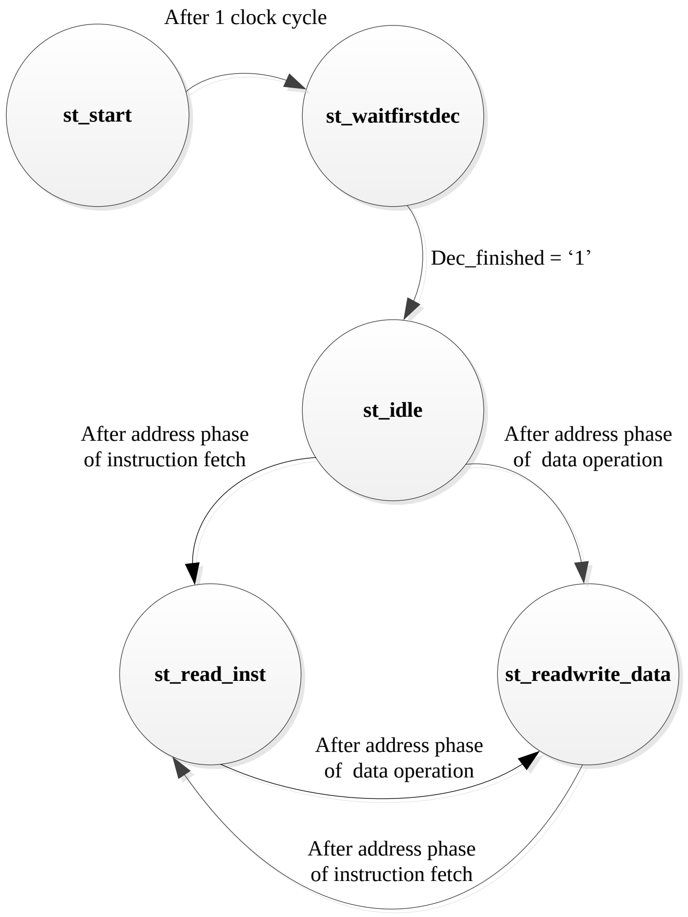

After 1 clock cycle
st_start
st_waitfirstdec
Dec_finished = ‘1’
st_idle
After address phase of instruction fetch
After address phase of data operation
st_read_inst
st_readwrite_data
After address phase of data operation
After address phase of instruction fetch

Figure 6.4-7: Cpu_ahbbuffer_bridge state machine diagram.

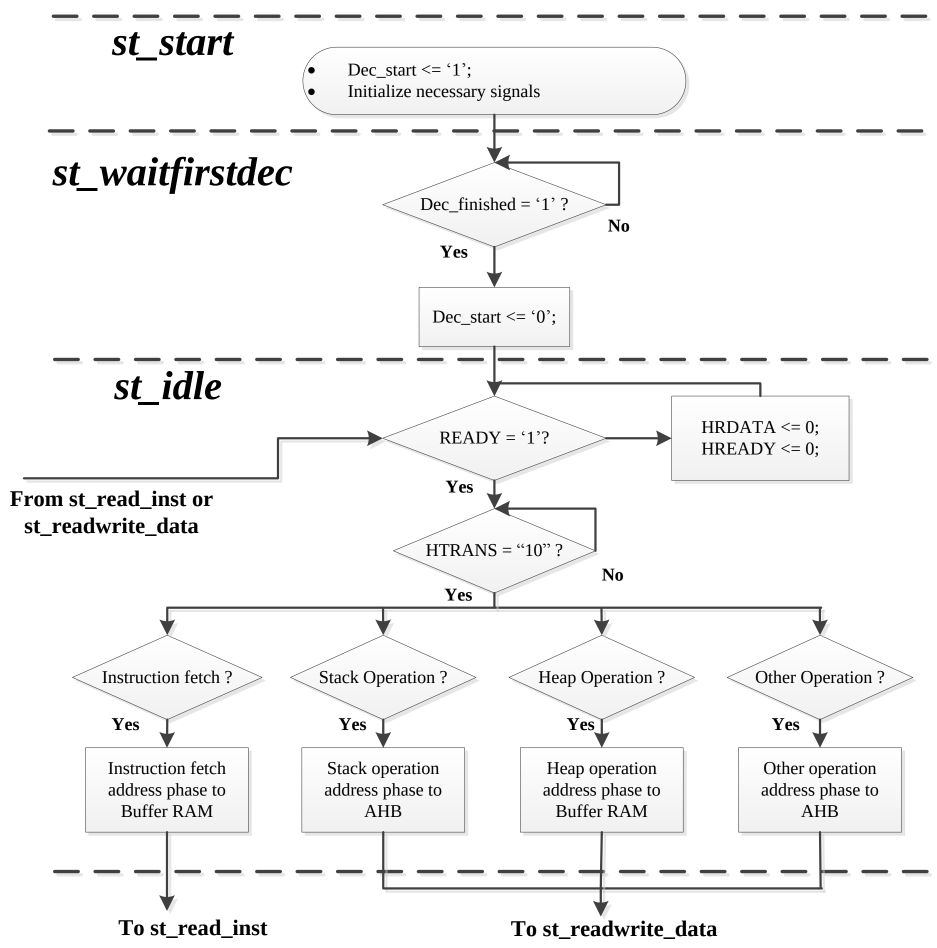


Figure 6.4-8: Flow chart of the first three states in cpu_ahbbuffer_bridge state machine.

When system has just started, this module has a role to request the first decompression process by driving dec_start port to '1'. Then in the next state, the module waits until dec_finished is '1' which indicates the first decompression process is done and then it removes dec_start request.

The next state which is st_idle, is an important state that initiates address phase to buffer_ram or AHB according to the operation requested by Cortex-M0. It is important to remember that HTRANS = "10" indicates that CPU wants to initiate an address phase of a transaction (refer to sub-chapter 4.1.1). In addition, the state recognizes the Cortex-M0 operation by comparing the address requested by CPU (HADDR) with the memory map layout we designed in sub-chapter 6.2.3.

In the case of CPU requests for read instruction (instruction fetch) while state is st_idle or st_readwrite_data, state will change to st_read_inst. St_read_inst (as shown in figure 6.4-9) is inherently a state to perform data phase of instruction fetch after address phase issuance of instruction fetch is done in st_idle or st_readwrite_data. This implies that in most cases, once state has just changed to st_read_inst, condition of HTRANS = "00" is immediately selected and it executes data phase of instruction fetch, which then the state changes back to st_idle.

Nevertheless, sometimes Cortex-M0 performs burst transfer in which Cortex-M0 may request new transaction while state is in st_read_inst. To accommodate this, we put mechanism to sense if a new transaction is initiated in this state (indicated by HTRANS = "10") and perform address phase according to the type of the transaction operation. For new transaction other than instruction fetch, it causes state to change to st_readwrite_data after the address phase issuance.

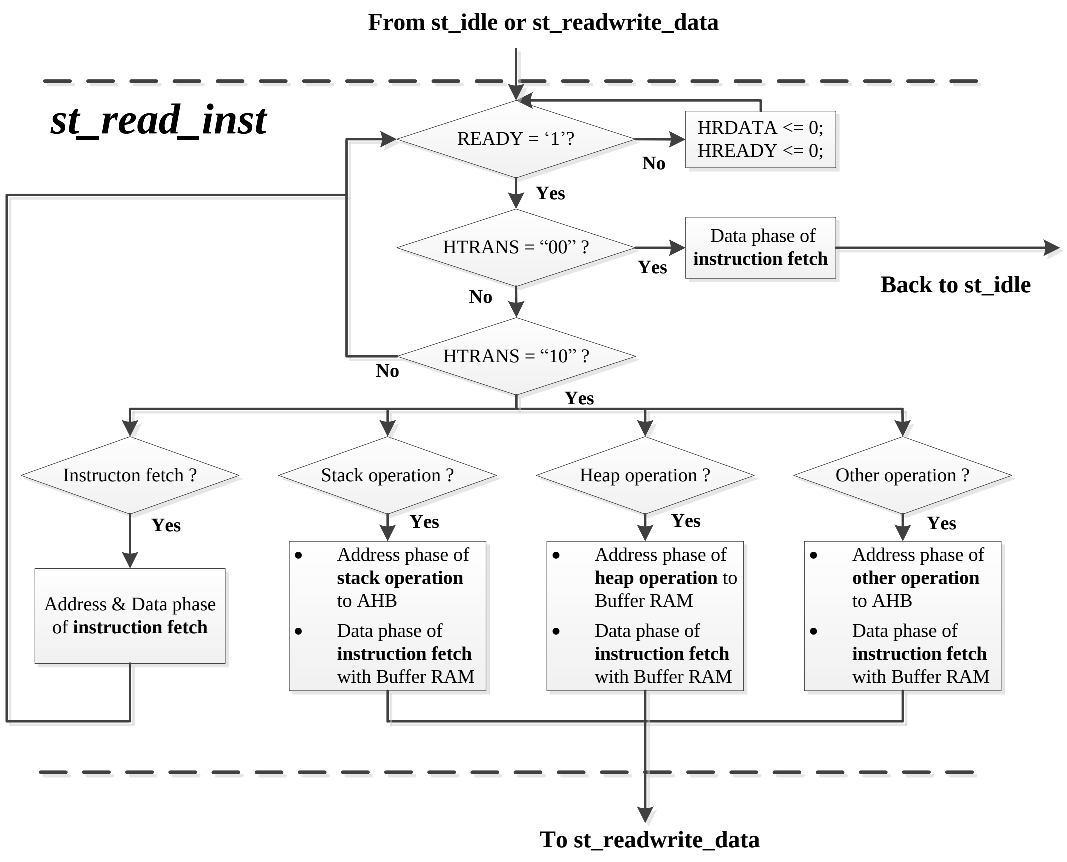

From st_idle or st_readwrite_data
st_read_inst
READY = '1'?
No
HRDATA <= 0;
HREADY <= 0;
Yes
HTRANS = "00" ?
Yes
Data phase of instruction fetch
Back to st_idle
No
HTRANS = "10" ?
No
Yes
Instructon fetch ?
Stack operation ?
Heap operation ?
Other operation ?
Yes
Address & Data phase of instruction fetch
Address phase of stack operation to AHB
Data phase of instruction fetch with Buffer RAM
Address phase of heap operation to Buffer RAM
Data phase of instruction fetch with Buffer RAM
Address phase of other operation to AHB
Data phase of instruction fetch with Buffer RAM
To st_readwrite_data

Figure 6.4-9: Flow chart of st_read_inst state.

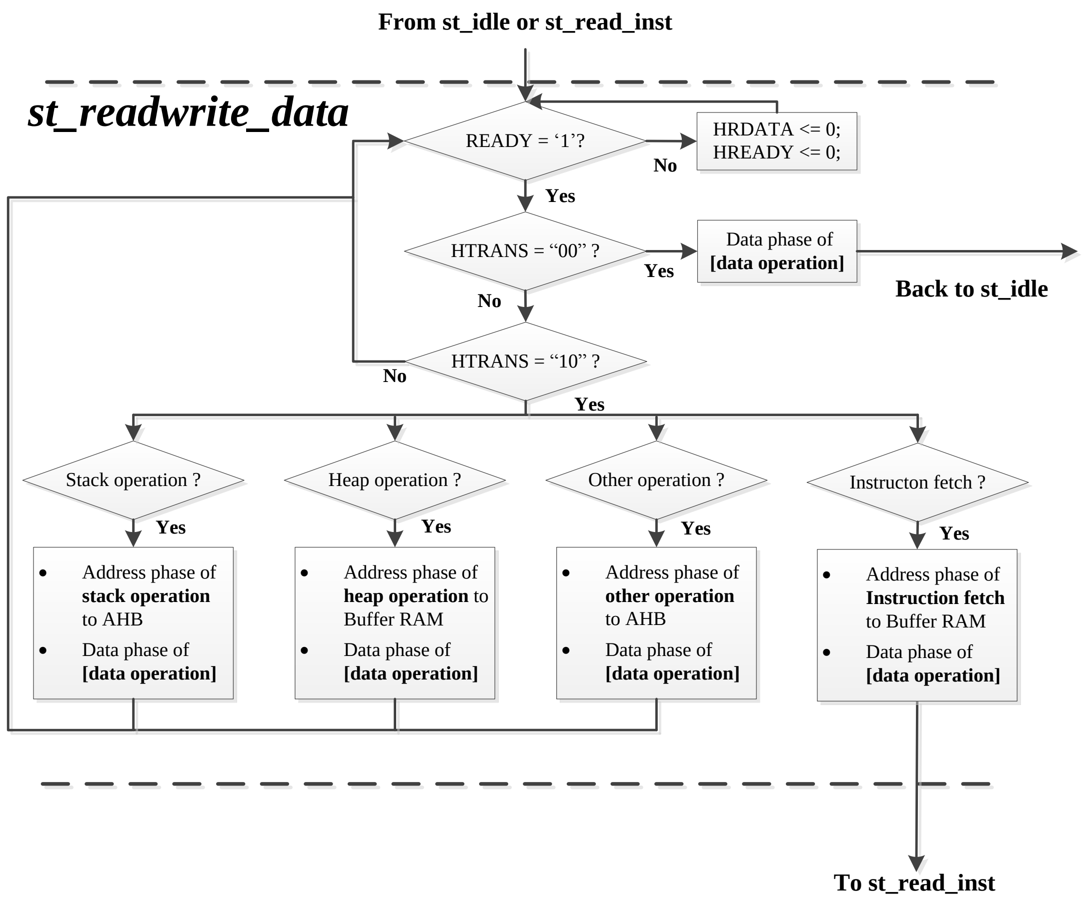

From st_idle or st_read_inst
st_readwrite_data
READY = '1'?
No
HRDATA <= 0;
HREADY <= 0;
Yes
HTRANS = "00" ?
Yes
Data phase of [data operation]
Back to st_idle
No
HTRANS = "10" ?
No
Yes
Stack operation ?
Heap operation ?
Other operation ?
Instructon fetch ?
Yes
Address phase of stack operation to AHB
Data phase of [data operation]
Address phase of heap operation to Buffer RAM
Data phase of [data operation]
Address phase of other operation to AHB
Data phase of [data operation]
Address phase of Instruction fetch to Buffer RAM
Data phase of [data operation]
To st_read_inst

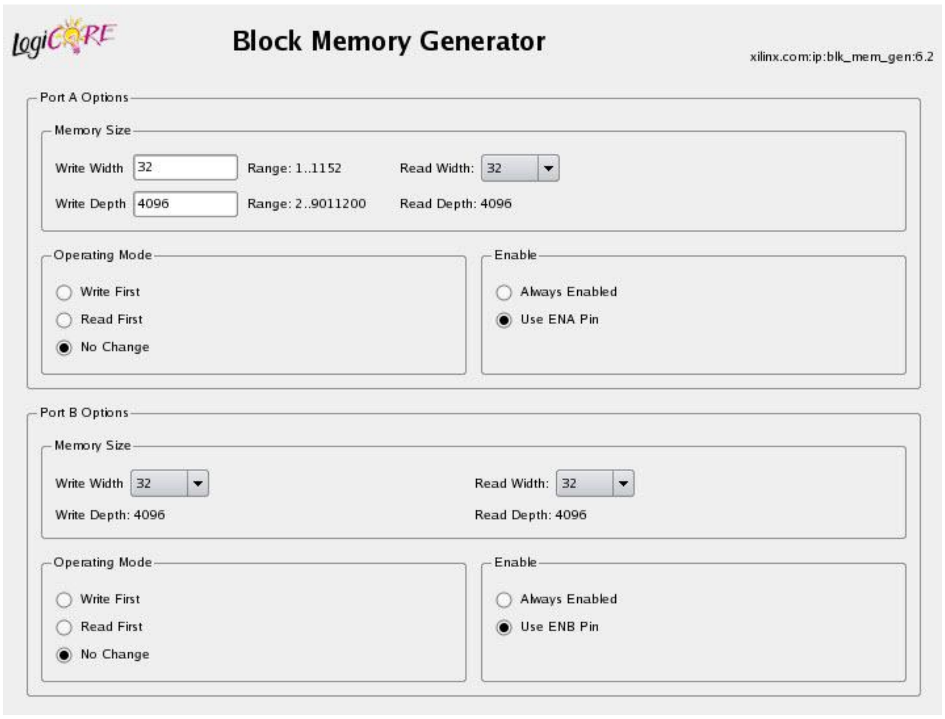
Figure 6.4-10: Flow chart of st_readwrite_data state.

St_readwrite_data as shown in figure 6.4-10 is a state designed to perform data phase of stack, heap and other operation (we group and call these operations as “data operation”). The concept inside st_readwrite_data is roughly similar to st_read_inst. However, clear difference can be easily seen on how the data phase of data operation is handled compared to the data phase of instruction operation. This difference can be observed in cpu_ahbbuffer_bridge source code provided in submitted zip file : Sourcecodes_forassessment_ms1718\CompletelyDesigned by AGHA\CM0X_wrapper files

6.4.5 Buffer_ram

Buffer_ram is a dual-port on-chip RAM we designed using Xilinx Block Memory Generator. The purpose of this buffer is to hold the decompressed data or instructions temporarily near CPU. It also serves as optimization implementation in our system since it may serve as cache. However, we do not implement sophisticated caching technique because the test program we ran on our system is pretty small in size, hence, the whole program is able to be contained in buffer_ram after first decompression process is finished.

In this project, we designed the buffer_ram to have total size of 16 KB. Then, we divided the total size into 8KB for instructions and another 8KB for heap data. The configuration of Xilinx Block Memory Generator to create our buffer_ram is shown in figure 6.4-11. It is important to notice that 4096 of write depth is number of address slots in the memory. Thus, 4096 x 4 bytes equals to 16384 bytes.


Block Memory Generator

xilinx.com:ip:blk_mem_gen:6.2

Port A Options

Memory Size

Write Width: 32 — Range: 1..1152 — Read Width: 32

Write Depth: 4096 — Range: 2..9011200 — Read Depth: 4096

Operating Mode: Write First / Read First / [✓ No Change]

Enable: Always Enabled / [✓ Use ENA Pin]

Port B Options

Memory Size

Write Width: 32 — Read Width: 32

Write Depth: 4096 — Read Depth: 4096

Operating Mode: Write First / Read First / [✓ No Change]

Enable: Always Enabled / [✓ Use ENB Pin]

Figure 6.4-11: Configuration to create buffer_ram in Xilinx Block Memory Generator

Chapter 7: Results and Analysis

7.1 Functional Verification Results.

In order to acknowledge that our designs work properly, we performed functional verification on all wrappers we designed. We verified our designs by using test bench provided by GRLIB SoC that is able to simulate the GRLIB SoC functionality in Modelsim. By simulating the test bench and having the memory (AHBRAM main_ram) injected or initialized with our Cortex-M0 program, we can observe the ports and signals of our wrappers designs through waveform viewer, which then we verify the functionality according to the behavior of Cortex-M0 program we injected.

7.1.1 CM0_wrapper Functional Simulation

Once the wrapper has been designed, we developed few test programs to test whether the basic transfer Cortex-M0 SoC we develop works properly.

The first program we tested is a very simple program called "LED_Blinking.c" program originally developed by Martos[34]. This program was also used by Piekarek to verify his system since this program is very simple and only require Cortex-M0 to do read operation.

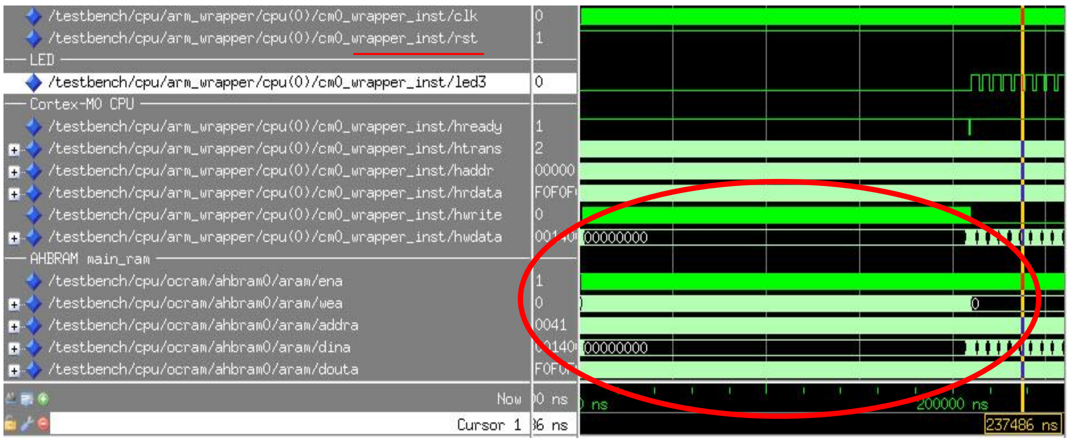

/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/clk
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/rst
LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/led3
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hready
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/htrans
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/haddr
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hrdata
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hwrite
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hwdata
AHBRAM main_ram
/testbench/cpu/ocram/ahbram0/aram/ena
/testbench/cpu/ocram/ahbram0/aram/wea
/testbench/cpu/ocram/ahbram0/aram/addra
/testbench/cpu/ocram/ahbram0/aram/dina
/testbench/cpu/ocram/ahbram0/aram/douta
Now
Cursor 1
200000 ns
237486 ns

Figure 7.1-1: LED_Blinking.c program runs in CM0_wrapper.
Time marker marks the end of 5th LED blink.

In figure 7.1-1, the red circle marks the impact of using startup.s file, in which it makes program to perform startup sequence that initializes stack and heap memory region to zero. After the startup sequence finishes, we can see the LED starts blinking infinitely because CPU reads 0xAAAA5555 (turn on LED) and 0xF0F0F0F0 (turn off LED) alternately.

The second program we tested was a more sophisticated program that performs stack operation intensively. The program is called "Recursive_Fibonacci.c", which is designed to generate 8 Fibonacci numbers starting from 0. Then, whenever number 2, 5, and 8 are generated, the program sends 0xAAAA5555 to CPU so that led_detector triggers LED for certain delay then turn it off again. Since the generation of Fibonacci numbers in this program is done by recursive function, there are many stack operations done during the program execution because recursive function involves a lot of function calls. Thus, by successfully running this program, it makes certain that our wrapper design can properly perform stack operation.

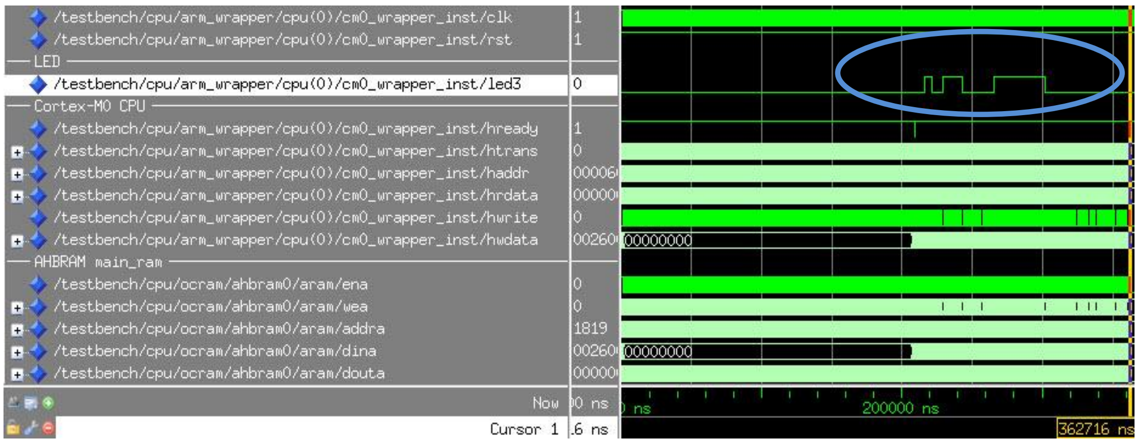

/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/clk
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/rst
LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/led3
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hready
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/htrans
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/haddr
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hrdata
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hwrite
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hwdata
AHBRAM main_ram
/testbench/cpu/ocram/ahbram0/aram/ena
/testbench/cpu/ocram/ahbram0/aram/wea
/testbench/cpu/ocram/ahbram0/aram/addra
/testbench/cpu/ocram/ahbram0/aram/dina
/testbench/cpu/ocram/ahbram0/aram/douta
Now
Cursor 1
200000 ns
362716 ns

Figure 7.1-2: Recursive_Fibonacci.c runs in CM0_wrapper.

The area marked by blue circle in figure 7.1-2 shows that LED blinked three times in different durations. The bigger the generated Fibonacci number the longer the LED is being turned on due to calculation process done by recursive function. We can also observe the activity of the main_ram in the region under the blinking LED that main_ram was active for being read and written, which indicates that stack operation was being performed. The source code of this program can be found in submitted zip file: "Sourcecodes_forassessment_ms1718\CompletelyDesigned by AGHA\ARM_Cortex-M0 C program"

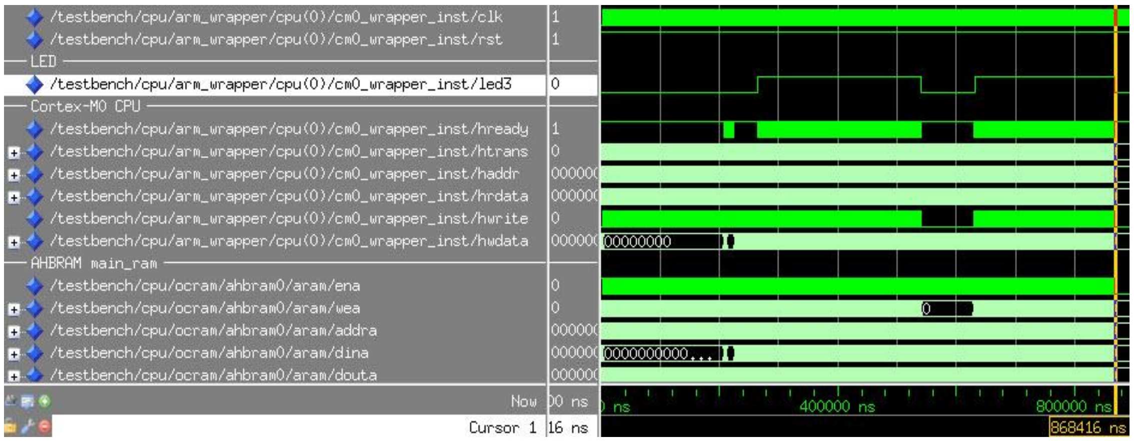

/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/clk
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/rst
LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/led3
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hready
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/htrans
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/haddr
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hrdata
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hwrite
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hwdata
AHBRAM main_ram
/testbench/cpu/ocram/ahbram0/aram/ena
/testbench/cpu/ocram/ahbram0/aram/wea
/testbench/cpu/ocram/ahbram0/aram/addra
/testbench/cpu/ocram/ahbram0/aram/dina
/testbench/cpu/ocram/ahbram0/aram/douta
Now
Cursor 1
400000 ns
800000 ns
868416 ns

Figure 7.1-3: Heap_write_read2KB_CR05.c runs in CM0_wrapper.

The last program we tested was a program which exercised heap memory operation. This program is called "Heap_write_read2KB_CR05.c ", in which it firstly writes 512 32-bits data in heap memory region starting from address 0x00006000 which begins with 0xAAAA5555. CR05 means the data (not the program) is able to be compressed by half (compression ratio 0.5) After that, delay was introduced and then data stored in 512 heap addresses (2KB size) are read back again, resulting LED to be turned on during heap writing process and heap reading process due to the data pattern.

As seen in figure 7.1-3, we can see two LED blinks, in which the first LED blink indicated write operation to heap region, and second LED blink indicated read operation on heap region. Actually, the algorithm of the program is based on to "Write_Read.c" that we tested on CM0_SDRAM_wrapper in previous sub-chapters. The difference is that Heap_write_read2KB_CR05.c uses startup.s file and perform more write operations by writing 2 KB of data to main memory. We can clearly compare figure 6.2-1 and 7.1-3 that there were not many HWRITE activity in figure 6.2-1 as compared to figure 7.1-3. This is because heap operation is always accompanied by stack operation and resulting

CPU to write frequently as shown by yellow and green circle in figure 7.1-3. Figure 7.1-4 and 7.1-5 show proofs that the system wrote and read to heap region (starting from address 0x00006000).

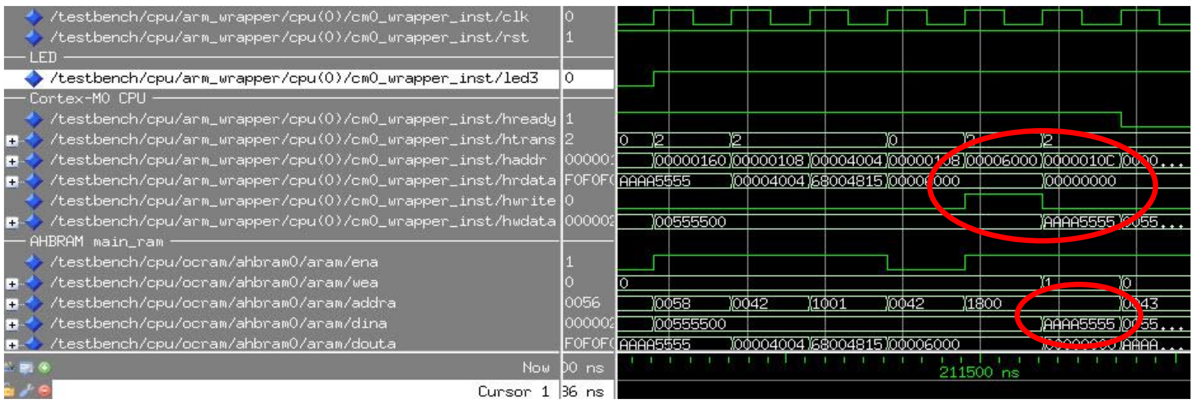

/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/clk
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/rst
LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/led3
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hready
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/htrans
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/haddr
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hrdata
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hwrite
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hwdata
AHBRAM main_ram
/testbench/cpu/ocram/ahbram0/aram/ena
/testbench/cpu/ocram/ahbram0/aram/wea
/testbench/cpu/ocram/ahbram0/aram/addra
/testbench/cpu/ocram/ahbram0/aram/dina
/testbench/cpu/ocram/ahbram0/aram/douta
Now
Cursor 1
211500 ns

Figure 7.1-4: Simulation of CM0_wrapper writing 0xAAAA5555 to heap address 0x00006000. LED turned on before writing because data has to be brought to CPU first (from instruction memory region) before being written by CPU.

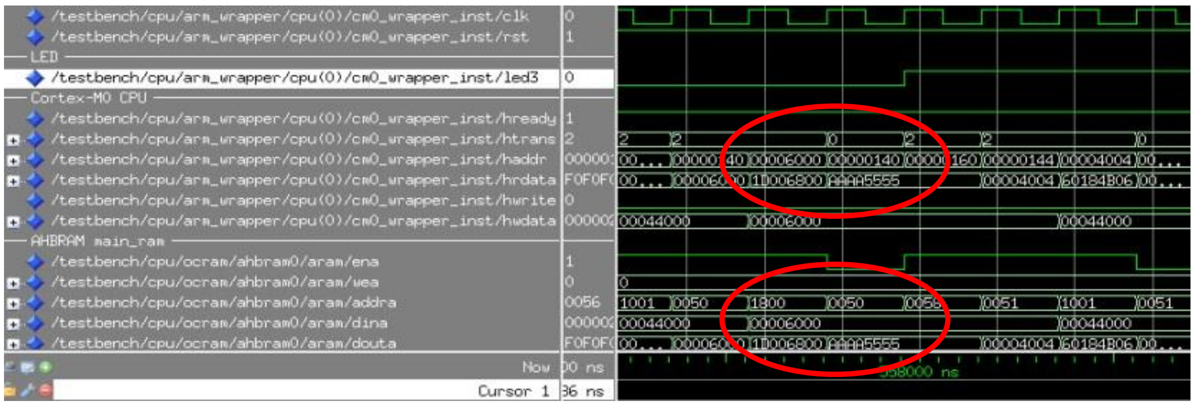

/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/clk
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/rst
LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/led3
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hready
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/htrans
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/haddr
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hrdata
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hwrite
/testbench/cpu/arm_wrapper/cpu(0)/cm0_wrapper_inst/hwdata
AHBRAM main_ram
/testbench/cpu/ocram/ahbram0/aram/ena
/testbench/cpu/ocram/ahbram0/aram/wea
/testbench/cpu/ocram/ahbram0/aram/addra
/testbench/cpu/ocram/ahbram0/aram/dina
/testbench/cpu/ocram/ahbram0/aram/douta
Now
Cursor 1
358000 ns

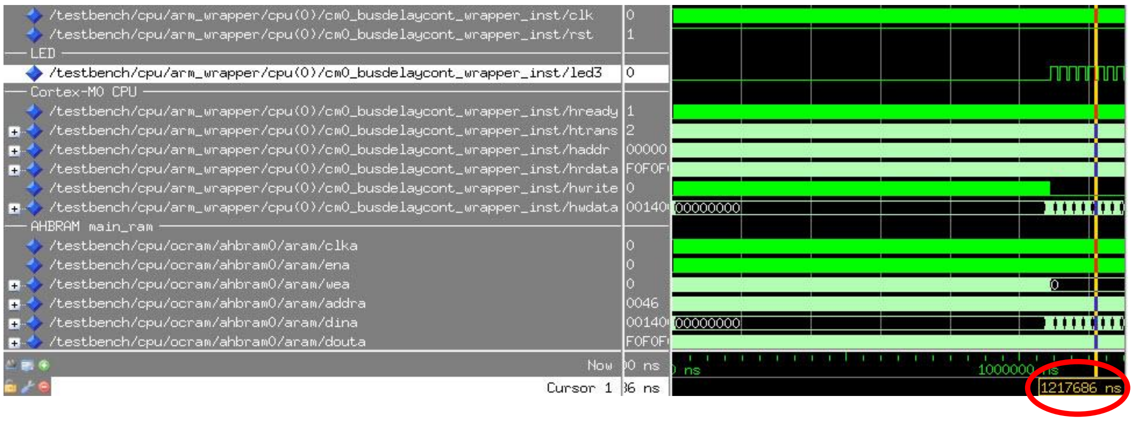
Figure 7.1-5: Simulation of CM0_wrapper reading from heap address 0x00006000 and turned on LED.

Now that we have verified that the basic transfer Cortex-M0 SoC we developed works properly to perform stack and heap operations. It is important to restate that both CM0_SDRAM_wrapper (sub-chapter 6.2.2) and CM0_wrapper are our original designs, in which CM0_SDRAM_wrapper is made based on CM0_wrapper. This is already a big leap to the previous work and this provides foundation of integrating Cortex-M0 to open source GRLIB SoC for everyone who is interested to conduct research in system-on-chip project with low budget.

7.1.2 CM0_busdelaycont_wrapper Functional Simulation

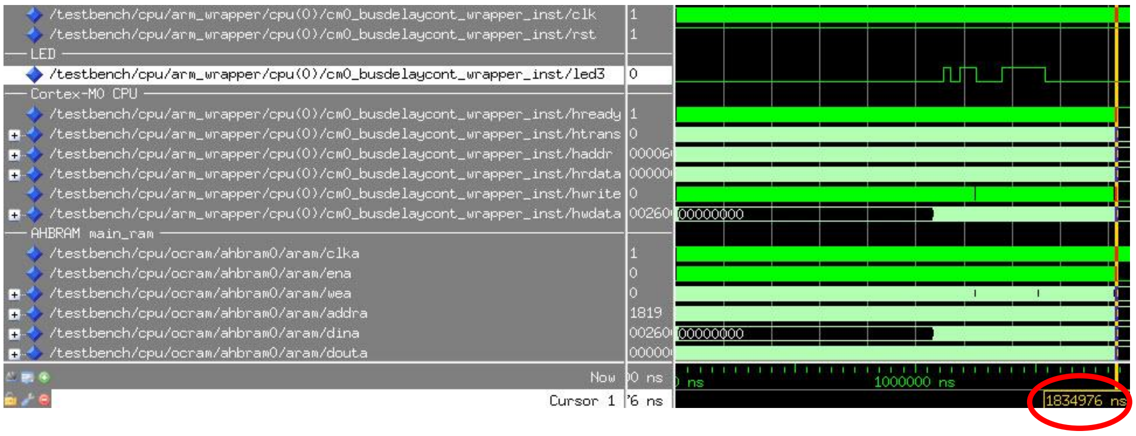
In this part, we show proofs that our CM0_busdelaycont_wrapper is able to perform the previous three Cortex-M0 programs shown in previous sub-chapter. We set the wrapper to perform as slow as SDRAM (7 clock cycles data phase), thus, the most important thing to notice in the following figures (figure 7.1-6, 7.1-7, 7.1-8) is the program execution time marked by red circle in each figure.

As we compare the execution time with figures in CM0_wrapper functional simulation (which also have execution time marked in the figures), we can notice CM0_busdelaycont_wrapper as significantly higher execution time. This is due to the virtue of our bus_delay_controller module.



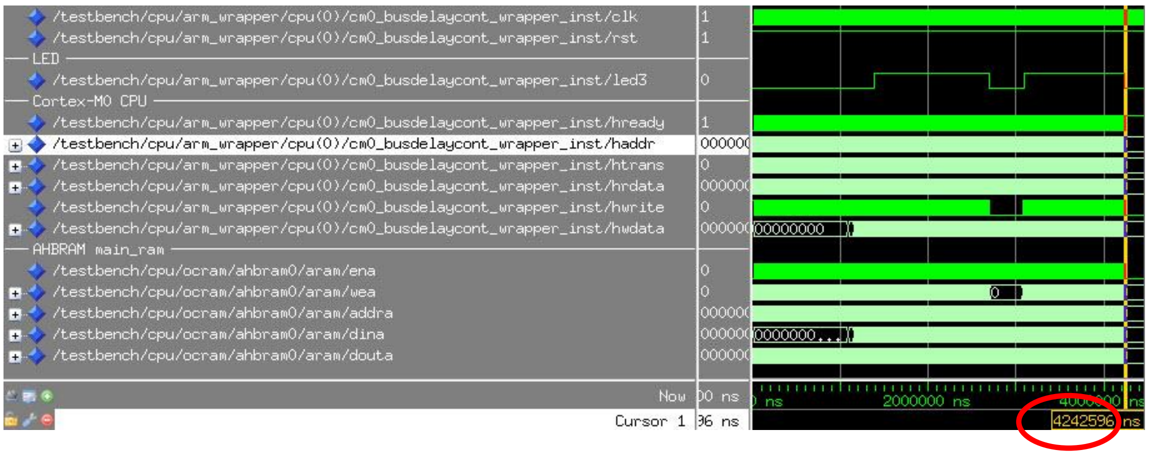
```
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/clk      0
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/rst      1
LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/led3     0
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/hready   1
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/htrans   2
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/haddr    00000
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/hrdata   F0F0F
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/hwrite   0
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/hwdata   00140   00000000
AHBRAM main_ram
/testbench/cpu/ocram/ahbram0/aram/clka    0
/testbench/cpu/ocram/ahbram0/aram/ena     0
/testbench/cpu/ocram/ahbram0/aram/wea     0
/testbench/cpu/ocram/ahbram0/aram/addra   0046
/testbench/cpu/ocram/ahbram0/aram/dina    00140   00000000
/testbench/cpu/ocram/ahbram0/aram/douta   F0F0F
Now        0 ns      ns      1000000 ns
Cursor 1   6 ns      1217686 ns
```

Figure 7.1-6: LED_Blinking.c program runs in CM0_busdelaycont_wrapper.
Time marker marks the end of 5th LED blink.



```
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/clk      1
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/rst      1
LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/led3     0
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/hready   1
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/htrans   0
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/haddr    00006
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/hrdata   00000
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/hwrite   0
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/hwdata   00260   00000000
AHBRAM main_ram
/testbench/cpu/ocram/ahbram0/aram/clka    1
/testbench/cpu/ocram/ahbram0/aram/ena     0
/testbench/cpu/ocram/ahbram0/aram/wea     0
/testbench/cpu/ocram/ahbram0/aram/addra   1819
/testbench/cpu/ocram/ahbram0/aram/dina    00260   00000000
/testbench/cpu/ocram/ahbram0/aram/douta   00000
Now        0 ns      ns      1000000 ns
Cursor 1   76 ns     1834976 ns
```

Figure 7.1-7: Recursive Fibonacci.c program runs in CM0_busdelaycont_wrapper.



```
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/clk      1
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/rst      1
LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/led3     0
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/hready   1
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/haddr    00000
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/htrans   0
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/hrdata   00000
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/hwrite   0
/testbench/cpu/arm_wrapper/cpu(0)/cm0_busdelaycont_wrapper_inst/hwdata   00000   00000000
AHBRAM main_ram
/testbench/cpu/ocram/ahbram0/aram/ena     0
/testbench/cpu/ocram/ahbram0/aram/wea     0
/testbench/cpu/ocram/ahbram0/aram/addra   00000
/testbench/cpu/ocram/ahbram0/aram/dina    00000   0000000...
/testbench/cpu/ocram/ahbram0/aram/douta   00000
Now        00 ns     ns      2000000 ns      4000000 ns
Cursor 1   36 ns     4242596 ns
```

Figure 7.1-8: Heap_write_read2KB_CR05.c program runs in
CM0_busdelaycont_wrapper.

7.1.3 CM0X_wrapper Functional Simulation

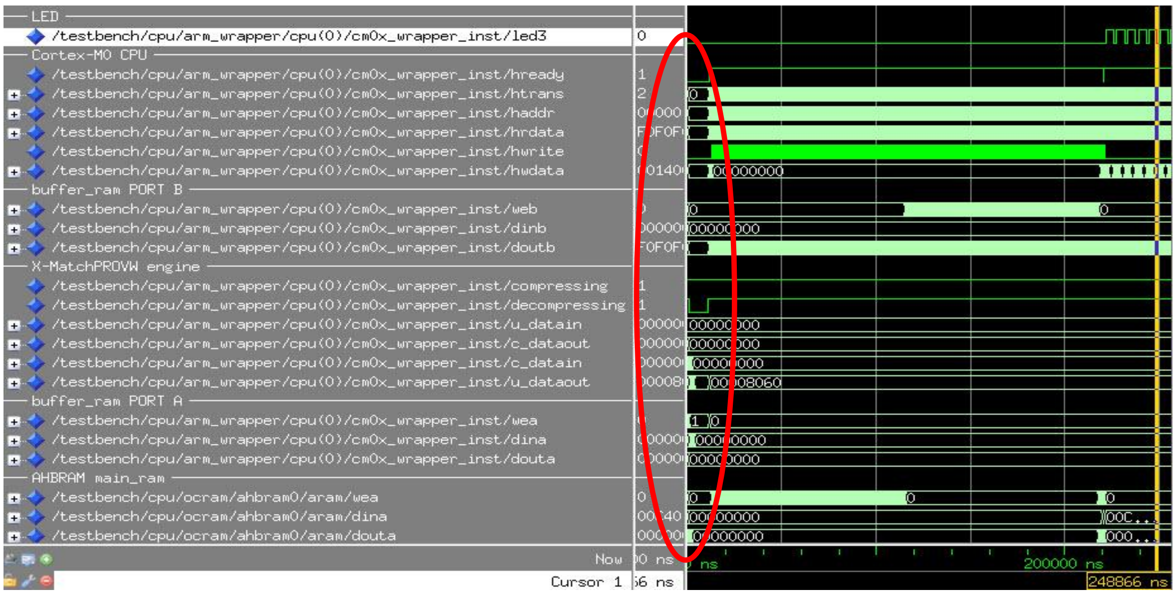

LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/led3
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hready
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/htrans
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/haddr
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hrdata
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hwrite
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hwdata
buffer_ram PORT B
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/web
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/dinb
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/doutb
X-MatchPROVW engine
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/compressing
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/decompressing
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/u_datain
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/c_dataout
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/c_datain
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/u_dataout
buffer_ram PORT A
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/wea
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/dina
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/douta
AHBRAM main_ram
/testbench/cpu/ocram/ahbram0/aram/wea
/testbench/cpu/ocram/ahbram0/aram/dina
/testbench/cpu/ocram/ahbram0/aram/douta
Now
Cursor 1
200000 ns
248866 ns

Figure 7.1-9: Compressed LED_Blinking.c runs in CM0X_wrapper

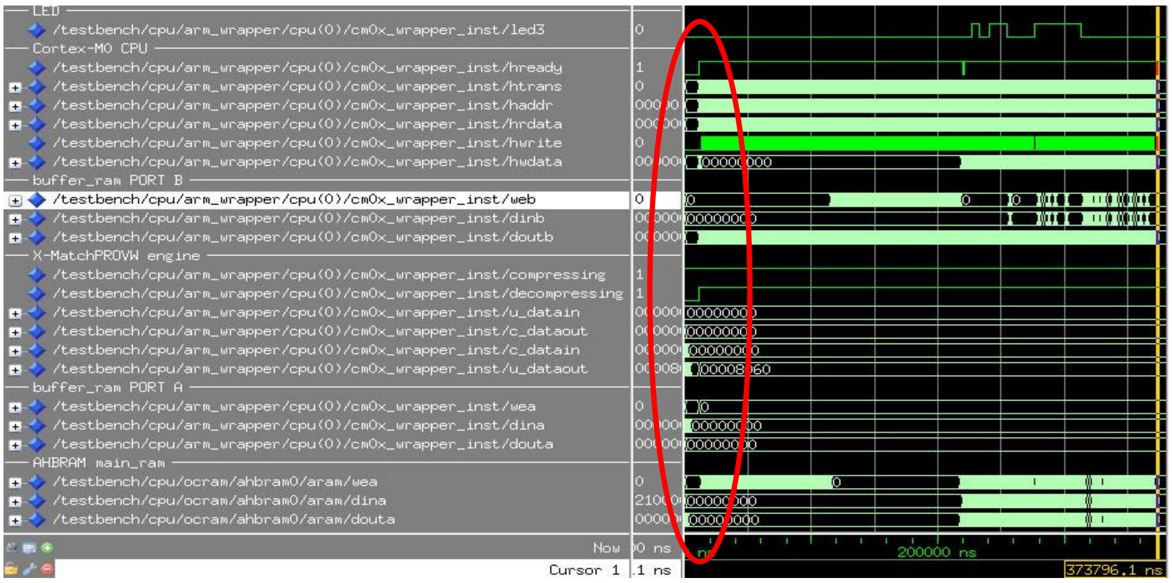

LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/led3
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hready
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/htrans
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/haddr
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hrdata
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hwrite
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hwdata
buffer_ram PORT B
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/web
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/dinb
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/doutb
X-MatchPROVW engine
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/compressing
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/decompressing
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/u_datain
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/c_dataout
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/c_datain
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/u_dataout
buffer_ram PORT A
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/wea
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/dina
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/douta
AHBRAM main_ram
/testbench/cpu/ocram/ahbram0/aram/wea
/testbench/cpu/ocram/ahbram0/aram/dina
/testbench/cpu/ocram/ahbram0/aram/douta
Now
Cursor 1
200000 ns
373796.1 ns

Figure 7.1-10: Compressed Recursive_Fibonacci.c runs in CM0X_Wrapper

From both compressed LED_Blinking.c and Recursive_Fibonacci.c simulation figure, we can observe that on-the-fly decompression worked at the beginning of each program as marked by red circle. The decompression activity indicated that the compressed instructions were decompressed and stored into buffer_ram. We can also observe the LED activity on each program was identical to those simulated in CM0_wrapper (sub-chapter 6.3.2) which means that the program ran successfully in our wrapper. However, there was no on-the-fly compression process involved in both program because there was no heap activity introduced in the program.

To test whether on-the-fly compression works in our CM0X_wrapper, we used compressed version of Heap_write_read2KB_CR05.c program as shown in figure 7.1-11 to 7.1-16. In this program, there are two on-the-fly decompression and one on-the-fly compression. The first decompression happened at the beginning of the program execution (marked by blue circle), which was to fetch compressed instructions from main memory, decompressed it and placed it to buffer_ram. After that, on-the-fly compression happened after 2KB of heap data has been written to buffer_ram (marked by red circle).

Afterwards, Cortex-M0 requested data from the first heap memory location which triggered the on-the-fly decompression to fetch compressed heap and placed it to buffer_ram (marked by white circle).

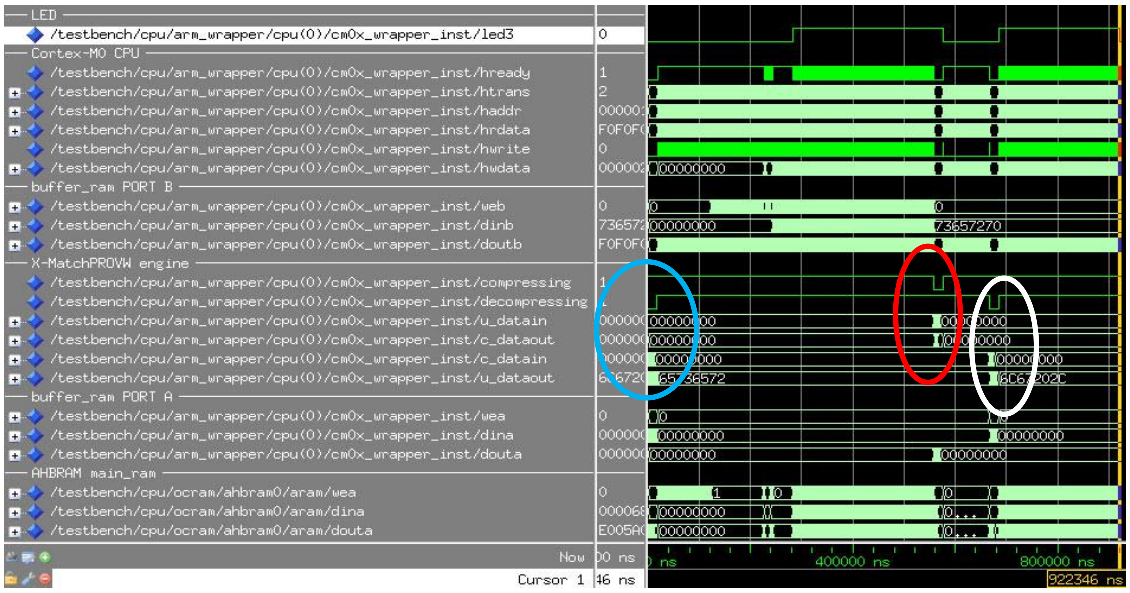

LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/led3
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hready
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/htrans
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/haddr
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hrdata
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hwrite
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hwdata
buffer_ram PORT B
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/web
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/dinb
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/doutb
X-MatchPROVW engine
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/compressing
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/decompressing
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/u_datain
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/c_dataout
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/c_datain
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/u_dataout
buffer_ram PORT A
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/wea
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/dina
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/douta
AHBRAM main_ram
/testbench/cpu/ocram/ahbram0/aram/wea
/testbench/cpu/ocram/ahbram0/aram/dina
/testbench/cpu/ocram/ahbram0/aram/douta
Now
Cursor 1
400000 ns
800000 ns
922346 ns

Figure 7.1-11: Compressed Heap_write_read2KB_CR05.c runs in CM0X_Wrapper

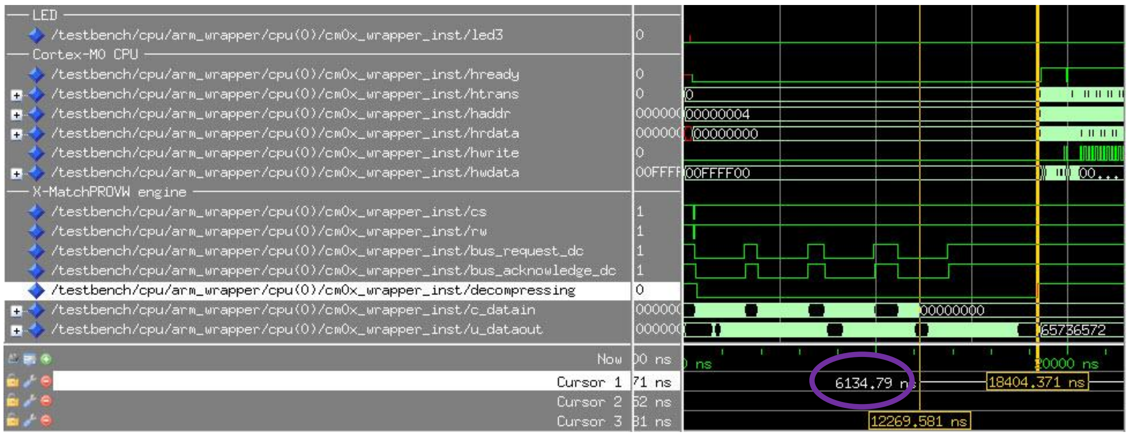

LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/led3
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hready
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/htrans
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/haddr
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hrdata
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hwrite
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hwdata
X-MatchPROVW engine
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/cs
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/rw
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/bus_request_dc
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/bus_acknowledge_dc
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/decompressing
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/c_datain
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/u_dataout
Now
Cursor 1
Cursor 2
Cursor 3
6134.79 ns
18404.371 ns
12269.581 ns

Figure 7.1-12: First on-the-fly decompression in Heap_write_read2KB_CR05.c simulation CM0X_Wrapper

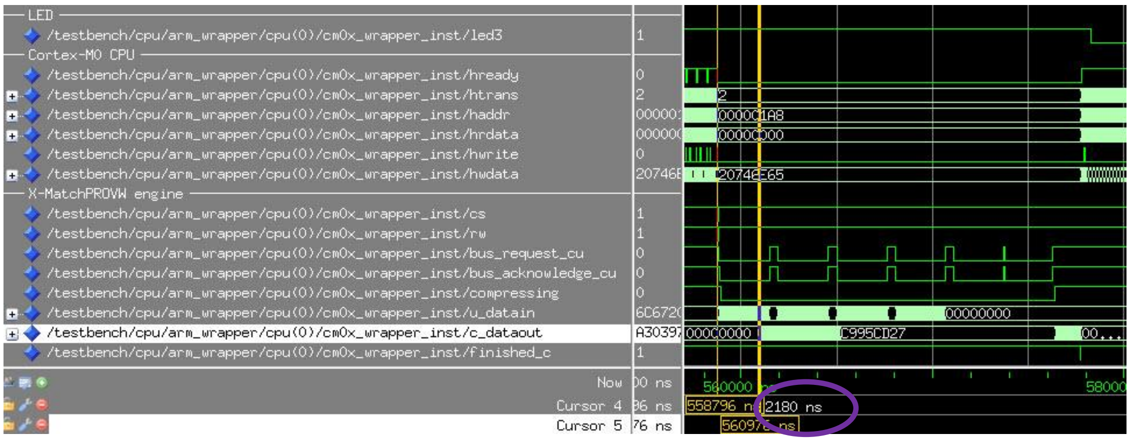

LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/led3
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hready
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/htrans
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/haddr
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hrdata
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hwrite
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hwdata
X-MatchPROVW engine
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/cs
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/rw
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/bus_request_cu
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/bus_acknowledge_cu
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/compressing
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/u_datain
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/c_dataout
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/finished_c
Now
Cursor 4
Cursor 5
2180 ns

Figure 7.1-13: On-the-fly compression in Heap_write_read2KB_CR05.c simulation CM0X_Wrapper.
Notice that compressed data (c_dataout) has approximately half data density than the origina data(u_datain) due to compression ratio 0.5

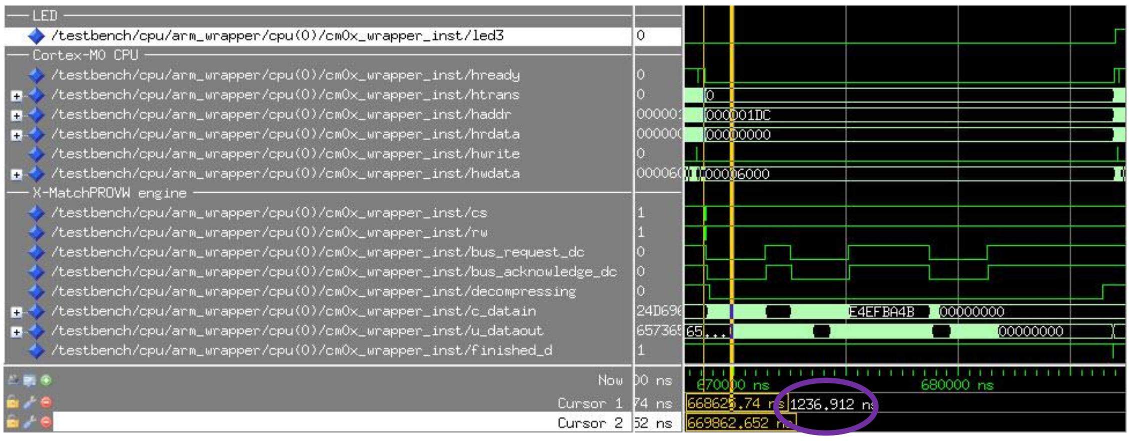

LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/led3
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hready
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/htrans
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/haddr
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hrdata
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hwrite
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hwdata
X-MatchPROVW engine
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/cs
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/rw
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/bus_request_dc
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/bus_acknowledge_dc
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/decompressing
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/c_datain
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/u_dataout
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/finished_d
000001DC
00000000
00006000
E4EFBA4B
00000000
Now
Cursor 1
Cursor 2
670000 ns
680000 ns
668625.74 ns
1236.912 ns
669862.652 ns

Figure 7.1-14: Second on-the-fly heap decompression in Heap_write_read2KB_CR05.c simulation CM0X_Wrapper

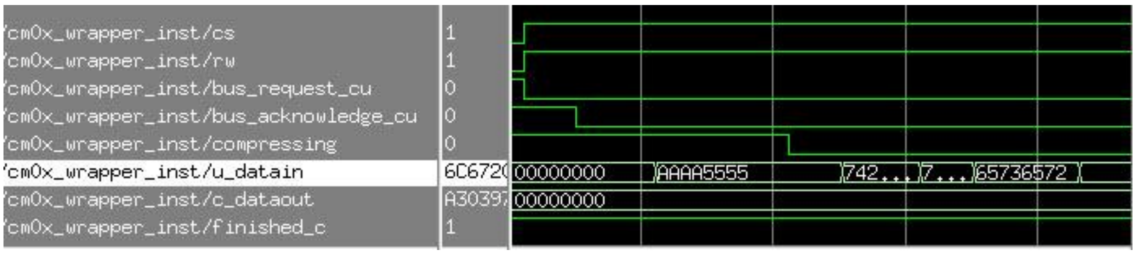

cm0x_wrapper_inst/cs
cm0x_wrapper_inst/rw
cm0x_wrapper_inst/bus_request_cu
cm0x_wrapper_inst/bus_acknowledge_cu
cm0x_wrapper_inst/compressing
cm0x_wrapper_inst/u_datain
cm0x_wrapper_inst/c_dataout
cm0x_wrapper_inst/finished_c
00000000
AAAA5555
65736572
00000000

Figure 7.1-14: Beginning of heap compression. Proof that 0xAAAA5555 was only the first few data.

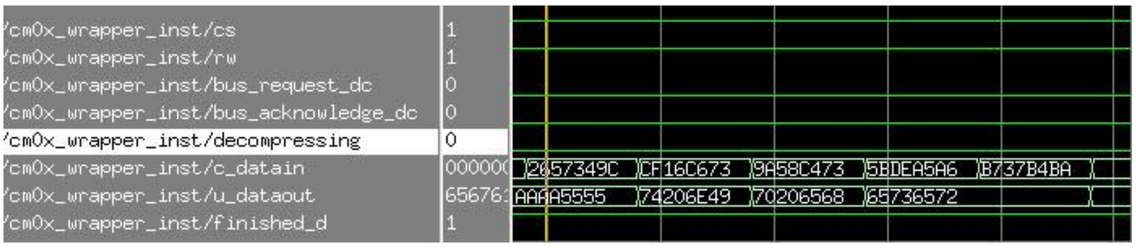

cm0x_wrapper_inst/cs
cm0x_wrapper_inst/rw
cm0x_wrapper_inst/bus_request_dc
cm0x_wrapper_inst/bus_acknowledge_dc
cm0x_wrapper_inst/decompressing
cm0x_wrapper_inst/c_datain
cm0x_wrapper_inst/u_dataout
cm0x_wrapper_inst/finished_d
2657349C
CF16C673
9A58C473
5BDEA5A6
B737B4BA
AAAA5555
74206E49
70206568
65736572

Figure 7.1-14: Beginning of heap decompression. Proof that 0xAAAA5555 was only the first few data.

7.2 Evaluating the Impact of CM0X_wrapper on SDRAM-based System

After we are convinced that our compressed main memory system works, we performed some experiments to evaluate the advantage and overheads of having compressed main memory system.

We have shown that our compressed main memory system works with On-chip RAM (AHBRAM). In that system, the most apparent benefit we can obtain is that we saves memory space up to half of its original size (due to the nature of X-MatchPROVW explained in literature review). Since we can only use AHBRAM to test CM0X_wrapper (due to SDRAM problem explained in previous chapter), it is hard to see the benefit of compressed main memory system in term of execution speed since AHBRAM is very fast. Therefore, the need of modifying the system (that works with AHBRAM) to perform as it was using SDRAM emerged.

Initially, to evaluate the impact of CM0X_wrapper on SDRAM-based system, we intended to compare execution time of program executed in CM0_busdelaycount_wrapper against modified CM0X_wrapper that would contain similar mechanism as bus_delay_controller. In this way, we would have an accurate comparison between normal system and compressed main memory system as both were using SDRAM although they are using AHBRAM. However, the mechanism similar to bus_delay_controller could not be applied to CM0X_wrapper due to complexity of implementing it and the time constraints of the project (considering tremendous efforts have been shown to make the system works properly although it works with AHBRAM). Hence, we could not model the CM0X_wrapper to perform as it was using SDRAM based on simulation.

Fortunately, we have found an alternative solution to model CM0X_wrapper SDRAM performance based on approximation. *The idea is that we try to approximate additional number of cycles introduced by main memory access at our AHBRAM-based CM0X-wrapper if SDRAM was used*. This is because main memory access is the access that would be affected by SDRAM performance (slower) if SDRAM was used. To do this, we must first identify what kind of operation that involves main memory access in our system.

Based on our design, compressed data fetched to the system will be decompressed and stored into near memory (buffer_ram) so that CPU fetches and writes data only with memory. This means that when CPU interacts with buffer_ram, CPU performs as fast as CM0_wrapper since in both case on-chip memory is used. However, we have mentioned that stack data is not compressed and therefore stack operation is bypassed to the main memory outside the wrapper. Thus, we have identified the first operation that involves main memory access, in which it is stack operation.

Secondly, we have mentioned that Cortex-M0 program at its beginning of execution, performs stack and heap initialization that we call it as “startup sequence” before executing the program (as can be seen in every simulation graphs of our design). This startup sequence performs write operation (writing 0) to stack and heap location in the memory, which implies that startup sequence is bypassed to main memory. Thus, these stack and heap write operations in startup sequence are considered as main memory access.

Thirdly, the activity of X-MatchPROVW to compress and decompress data obviously involves the main memory since compression is writing stream of data to memory and decompression is fetching stream of data to X-MatchPROVW. Therefore, the compression and decompression activity would also be affected by SDRAM performance as overhead.

Now that we know three operations or activities that would be affected by SDRAM performance if SDRAM was used since these activities deal with main memory. Thus, now we can approximate additional number of cycles introduced by SDRAM performance. In our research, we used Heap_write_read2KB_CR05.c . The details of approximating those three activities are explained in the next sub-chapter.

7.2.1 Approximation on Number of Main Memory Access in Startup Sequence.

We already know that startup sequence initializes stack and heap memory region to zero. Thus, the number of transaction involved in stack write and heap write operation can be approximated as long as we know the size of our main memory.

As we explained in sub-chapter 6.2.5, we have memory of 32KB size, in which 16 KB for instructions and another 16 KB for data. Based on this information, we know that startup sequence initializes 16 KB (accurately 16384 bytes) memory consisted of 4096 address location. Hence, there are **4096 transactions to main memory** during startup sequence.

7.2.2 Approximation on Number of Stack Operation Transactions after Startup Sequence.

The question is now, how are we going to approximate the number of stack operation in a program after startup sequence? Since stack operation can happen randomly depending on the program. Fortunately, we used Heap_write_read2KB_CR05.c program, in which its stack operation is predictable. Based on our observation from simulation of Heap_write_read2KB_CR05.c, we have identified unique characteristic of Heap_write_read2KB_CR05.c , in which *every time CPU is writing or reading to heap memory region, there are two stack read operations and one stack write operation* (proof for this statement can be found in appendix A). Therefore, we know that in Heap_write_read2KB_CR05.c we write 512 32-bit data (2KB) and read it back. Thus, the number of stack operation happens in this program excluding startup sequence is:

At writing to heap,

stack read = 2 x 512 data = 1024 transaction.

stack write = 1 x 512 data = 512 transcation.

Similarly for reading from heap,

stack read = 2 x 512 data = 1024 transaction.

stack write = 1 x 512 data = 512 transcation.

Not forgot to mention, there are also stack write transactions just after startup sequence is finished, which contributes for additional 756 transactions based on our observation.

So in total, there are **3828 transactions of stack operation** (excluding startup sequence) that would be affected by SDRAM performance in Heap_write_read2KB_CR05.c

7.2.3 Approximation on X-MatchPROVW Overhead Affected by SDRAM Performance

As explained before, compression involves writing to main memory and decompression involves reading from main memory. Thus, these write and read operations to main memory by X-MatchPROVW are the overhead that is affected by SDRAM performance.

In the case of Heap_read_write2KB_CR05.c, the original program size is 2844 bytes. The one stored in main memory is the compressed version which is 1760 bytes (0.61 compression ratio)1760 bytes are actually consists of 440 32-bit data. Thus, there are 440 transactions affected by SDRAM for the first decompression since these 440 data are fetched from main memory to X-MatchPROVW during the first decompression.

For compression part, 2KB (accurately 2048 bytes) of heap data are compressed to 1100 bytes (compression ratio of 0.537)Thus, there are 275 transactions affected by SDRAM for the compression since these 275 data are fetched from main memory to X-MatchPROVW during the first decompression.

The second decompression part has same main memory access as the compression part because it decompresses same amount of compressed heap data.

Hence, we can sum up all the compression decompression overhead affected by SDRAM performance. In total, there are 440 + 275 + 275 = **990 transactions involved**. We call this overhead as "engine-main memory overhead".

In addition, we also have to calculate overhead involved when engine writing decompressed data to buffer_ram or compressed data from buffer_ram (we call it "engine-buffer memory overhead").We identify these overhead in term time by measuring it from simulation waveform. For first decompression, we found that there were 6134 ns of overhead. For compression we found that there 2180 ns of overhead. For second decompression we found that there were 1236 ns of overhead. All these overheads can be observed in figure 7.1-12 to 7.1-14 (each marked by purple circle). Hence, In total, there are **9548 ns of engine-buffer overhead.**

7.2.4 Approximation Result of CM0X_wrapper SDRAM-based Execution Time.

The next step after we summed up all the main memory accesses is that we need to approximate the total execution time of Heap_write_read2KB_CR05.c in our compressed main memory system as it was using SDRAM.

To do that, we originally come up with the approximation formula below that can be applied to any easily predictable program as long as we have CM0_wrapper and CM0X_wrapper.

$$\mathbf{(TEcmo_{onram} - TDMcmo_{onram}) + (TDMcmo_{onram} * Cdd_{offram}) + TX = TEcmox_{offram}}$$

In which $TX = TXeb_{onram} + TXem_{offram}$

$TEcmox_{offram}$ = Total execution time of a particular program executed in CM0X_wrapper while off-chip RAM was being used.

$TEcmo_{onram}$ = Total execution time of a particular program executed in CM0_wrapper while on-chip RAM having a single clock cycle of data phase is being used.

$TDMcmo_{onram}$ = Total data phase executions time belong only to main memory access in a particular program, in which the program is executed in CM0_wrapper while on-chip RAM having a single clock cycle of data phase is being used.

Cdd_{offram} = Number of clock cycle of a data phase executed by off-chip RAM.

TX = Total X-MatchproVW overhead in time.

$TXeb_{onram}$ = Total "engine-buffer memory" overheads in time while on-chip RAM is being used

$TXem_{offram}$ = Total "engine-main memory" overheads in time while off-chip RAM is being used

We understand that it is little bit difficult to understand the formula at a glance. Therefore we are going to explain step by step the implementation of the approximation formula on Heap_write_read2KB_CR05.c

1. Firstly, we get the total execution time of Heap_read_write2KB_CR05.c done by CM0_wrapper, which is 868416 ns as shown by figure 7.1-3.
   $TEcmo_{onram =}$ 868416 ns

2. Then, we remove portion of time related to data phases of main memory access ($TDMcmo_{onram}$) from $TEcmo_{onram}$. This results to having a total execution time of Heap_read_write2KB_CR05.c without having execution time related to data phases of main memory access when system using fastest memory ($TEcmo_{onram}$ - $TDMcmo_{onram}$). We want this because total execution time of data phases of main memory access ($TDMcmo_{onram}$) is the only portion of time which get affected by SDRAM performance in our compressed main memory system (remember that SDRAM or slower memory extends data phase execution). This also implies that total program execution time without total data phase of main memory access execution time ($TEcmo_{onram}$ - $TDMcmo_{onram}$) is the same for both CM0_wrapper and CM0X_wrapper. *Thus, by having $TDMcmo_{onram}$ removed, we can extend it ($TDMcmo_{onram}$) depending on how slow the off-chip RAM (SDRAM) and add it back to $TEcmo_{onram}$ - $TDMcmo_{onram}$ so that we get a new total execution time of the program with new $TDMcmo_{onram}$ that has been extended several times by SDRAM performance.*

   To get $TDMcmo_{onram}$, we need firstly sum number of main memory accesses we calculated in previous section, in which **3828 + 4096 = 7924** (excluding engine's overhead).Inherently, data phase of main memory access for on-chip RAM is only one clock cycle for each main memory access. Thus, **$TDMcmo_{onram}$ = 7924 * 1 clock cycle * 20 ns (50MHz system clock frequency) = 158480 ns.**

   Hence, now **$TEcmo_{onram}$ – $TDMcmo_{onram}$ = 868416 ns - 158480 ns** = **709936 ns**.

3. Then, we extend $TDMcmo_{onram}$ by data phase clock cycle of off-chip RAM (Cdd_{offram}) , in which in this case Cdd_{offram} = 7 clock cycle since we model SDRAM.
   $TDMcmo_{onram}$ * Cdd_{offram} = 158480 ns * 7 = 1109360 ns.

4. Afterwards, we calculate total overhead by X-MatchPROVW (TX).
   $TXeb_{onram}$ = 990 * 20 ns * Cdd_{offram} = 990 * 20 ns * 7 = 138600 ns.
   $TXem_{offram}$ = 9548 ns (as stated in section 7.2.3).
   TX = $TXeb_{onram}$ + $TXem_{offram}$ = 138600 + 9548 = 148148 ns.

5. Lastly, we sum them resulting to
   ($TEcmo_{onram}$ – $TDMcmo_{onram}$) + ($TDMcmo_{onram}$ * Cdd_{offram}) + TX = $TEcmox_{offram}$
   709936 + 1109360 + 148148 = 1967444 ns

   Now that we have the total execution time of Heap_write_read2KB_CR05.c executed in CM0X_wrapper which is **$TEcmox_{offram}$ = 1967444 ns.**

To see how good this number represents, we compare this execution time with execution time of the same program executed by CM0_budelaycont_wrapper when it is set to perform as slow as SDRAM. We compare with it because it represents a working normal Cortex-M0 SoC that is able to perform like SDRAM. As we saw in figure 7.1-8, the execution time of CM0_busdelaycont_wrapper is 4242596 ns. **Hence, we can calculate that, in SDRAM-based system, there is 53% speed improvement** ((4242596-1967444)/4242596) **for executing Heap_write_read2KB_CR05.c in our compressed main memory system (CM0X_wrapper) as compared to normal Cortex-M0 SoC.**

Cdd_{offram} (clock cycles)	Execution time of CM0_busdelaycont_wrapper (ns)	Execution time of CM0X_wrapper (ns)	% Speed gain by CM0X_wrapper
3	1978176	1254324	36.6
5	3110376	1610884	48.2
7	4242596	1967444	53.6
9	5374776	2324004	56.8
11	6505976	2680564	58.8
13	7639176	3037124	60.2
20	11601876	4285084	63.1
30	17261636	6067884	64.8

Table 7.2-1: Heap_write_read2KB_CR05.c execution time in two different systems with varying main memory performance. Cdd_{offram} = 7 is equivalent to SDRAM performance

Since CM0_busdelaycont_wrapper we developed has the ability to vary Cdd_{offram}, we conducted experiment to see how much improvement our compressed main memory system can deliver at varying main memory performance. Table 7.2-1 shows that if we use slower main memory, we get more percentage speed gain.

It is important to understand that the high speed gain percentage shown in table 7.2-1 is not solely due to the X-MatchPROVW compression engine but also due to buffer_ram (decompression buffer) implementation, in which it stores all decompressed program near CPU thus it acts like cache system.

If we compare our system with cache system that bursting instructions or data from main memory to cache. Our system still has more advantage as compared to normal cache system because our system bursting less data from off-chip RAM to on-chip RAM.

We could not evaluate our system with related projects mentioned in literature review because our system is not as complete as those mentioned in literature review. However, we could state that our system has possibilities to be compared as good as SCMS [21] because SCMS uses X-Match based compression algorithm and using decompression buffer (in our case it is called buffer_ram).

7.3 Experiments on Compression Block Size.

Figure 7.3-1 shows histogram of how the duration of compression and decompression varies as block sizes increases. The duration shown for each compression block size is the duration for one time compression and decompression activity. In addition, data used in this experiment were data that has compression ratio 0.5.

From the histogram, we can clearly see compression and decompression duration are roughly similar. We also can notice that the duration is significantly increased as compression block size doubles. However, the difference decreases as we can see that duration from of 4 KB is 2.1 times of 2KB block size and duration of 8 KB is 1.8x of the duration of 4 KB. Thus, it is better to use higher compression block size because the duration of compressing and decompressing 8 KB of data using 8KB compression block size is lesser than compressing or decompressing 8KB of data using 2KB compression block size.

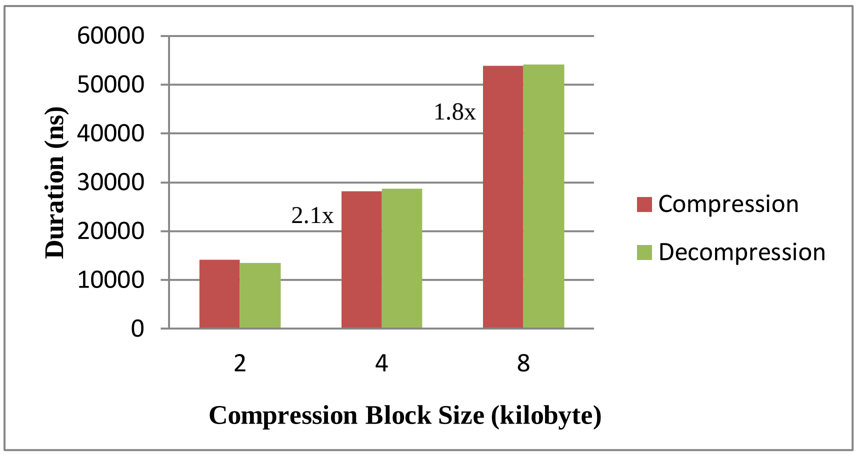


Figure 7.3-1: Histogram of compression and decompression duration against compression block size

For example, the duration is of 2KB compression block size is 14196 ns and duration of 8KB compression block size is 53936 ns as roughly shown in the figure above. if we compress 8KB size of data using 2KB compression block size, the total duration would be 14196 * 4 (four times consecutive compressions) = 56784 ns. This is 5% increments from using 8KB compression block size for an 8KB size of data. Obviously this percentage would be higher if we perform compression of larger size of data with small compression block size as 2KB and resulting to higher overhead. Thus, larger compression block size is preferred.

However, if compression block size is too large, it may cause slower CPU response. For instance, assume we have 100 KB of compressed data with compression block size of 100 KB. While 100 KB is being decompressed and CPU requests for compressing data to main memory (due to buffer is full), CPU has to stall waiting until the XMatchPROVW is vacant. In the other hand, in the case of smaller block size such as 50KB which is decompressing 100KB data, when CPU's request of compression interrupts before the first 50 KB (out of 100KB) finishes, X-MatchPROVW can switch to do compression as requested by CPU after first decompression of 50KB finishes. After the engine has finished the compression requested by CPU, the other half of 100KB decompression can be resumed.

Another point can be extracted from this experiment is that this experiment shows higher degree of confidence to our wrapper design. This is because during the development of the wrapper, once we could do compression and decompression of 1KB data, it did not mean that it would work with 2KB or higher until we could fix it properly. Figure 7.3-2 shows compressing and decompressing 8KB of data having compression ratio of 0.5 during execution of Recursive_Fibonacci.c program.

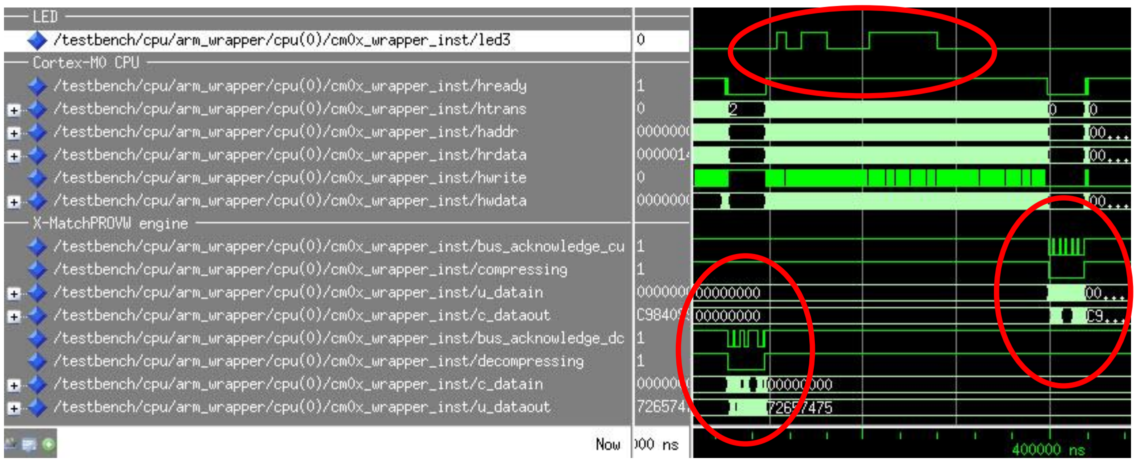

LED
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/led3 0
Cortex-M0 CPU
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hready 1
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/htrans 0
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/haddr 000000
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hrdata 000001
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hwrite 0
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/hwdata 000000
X-MatchPROVW engine
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/bus_acknowledge_cu 1
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/compressing 1
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/u_datain 000000
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/c_dataout C9840
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/bus_acknowledge_dc 1
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/decompressing 1
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/c_datain 00000
/testbench/cpu/arm_wrapper/cpu(0)/cm0x_wrapper_inst/u_dataout 726574
Now
)00 ns
2
0
00000000
00000000
00000000
72657475
400000 ns

Figure 7.3-2: Proof of 8 KB data being decompressed and compressed while performing Recursive_Fibonacci.c program.

7.4 Implementation Results

7.4.1 Zero Latches

After we have developed the designs and verify its functionality, we implemented CM0_wrapper and CM0X_wrapper into Xilinx Virtex 5 FPGA Board. In FPGA development, we normally have to design HDL code that avoids the use of latches in FPGA. Latches are not preferred because it may cause timing problems, synthesis problems, place and route problem and etcetera. Such problem occurs because during the HDL code development, designers unintentionally generate latches due to incomplete if-else statement, complicated loops, misplaced signals etc.

To avoid latches easily, we coded our design using "two processes" VHDL coding style by Jiri Gaisler [45]. What makes this coding style easily avoid latches is that the combinational part of the code is coded using variables that are connected to registered signals. This eliminates the latch caused by misplacing signals or incomplete if-else statement. Further information regarding two processes coding style can be found in [45].

In our designs, there are no latches for both CM0_wrapper and CM0X_wrapper. This resulting our designs are able to run LED_Blinking.c, Recursive_Fibonacci.c , and Heap_write_read2KB_CR05.c as expected in FPGA board although it only blinks LED as expected from simulation graph. Those programs we tested has been modified to suit board test in which we increase the period for each time LED blinks so that we can see how many times exactly LED blinks. The evidence of zero latches can be found in "Macro Statistics "section inside synthesis report, in which no latches is declared in the statistics.

7.4.2 Logic Resources Utilization and Power Consumption Estimation.

	CM0_wrapper whole system	CM0X_wrapper whole system	Differences	% Differences
No. of Slice Registers used	3058 units	5670 units	2612 units	85.4%
Utilization of slice registers	4%	8%	4%	100%
No. of Sice LUTs used	7849 units	13842 units	5993 units	76.3%
Utilization of slice LUTs	11%	20%	19%	81%
Total on-chip power	2293mW	2336 mW	43mW	1.8%

Table 7.4-1: Comparison of Logic Utilization and Estimated Power Consumption.

The table above compares important points of implementation reports between CM0_wrapper and CM0X_wrapper generated by Xilinx ISE. CM0_wrapper represents normal Cortex-M0 SoC and CM0X_wrapper represents our compressed main memory system.

From the table, we can observe that our compressed main memory system is significantly higher in term of FPGA logic utilizations which are slice registers and LUTs (Look up Table). As we observed from XPower report (Xilinx's Power Estimator tool), we found that X-MatchPROVW occupied huge portion of logic resources with 2087 units of registers and 5130 of LUTs. Although utilization is higher, estimated power consumption was increased by 1.8%. This is because X-MatchPROVW was estimated to consume only 1.42 mW, cpu_ahbbuffer_bridge consumes only 0.32 mW and comdec_ahbbuffer_bridge consumes only 0.05 mW.

We believe that high logic utilization can be further optimized once our system is designed using ASIC (Application Specific Integrated Circuit). The most important point from our system is that it saves space, improves execution speed and negligible power consumption overhead.

Chapter 8: Conclusion and Future Work

8.1 Conclusion

To sum up our dissertation, the compressed main memory system for an ARM Cortex system-on-chip was successfully designed to the extent that it satisfies all the aims and objectives listed. The first aim with its objectives which is to investigate the feasibility of implementing compressed main memory system with ARM Cortex CPU, was achieved by end of chapter 3. The second aim which is to design and implement compressed main memory system was achieved through the rest of the chapters.

The first milestone we need to mention is that we have fixed the problem brought by previous work regarding write operation despite considerable amount of time was spent on solving technical matters. As a result, we have improved the integration of Cortex-M0 CPU into GRLIB SoC by designing our own wrapper called CM0_wrapper, which performs proper Cortex-M0 program that contain stack operations even with lesser complexity than the previous work's design. The process of fixing the previous problem and designing working CM0_wrapper provide high degree of understanding in system-on-chip implementation. In spite of CM0_wrapper works with on-chip memory, we have successfully developed another called CM0_busdelaycont_wrapper, which is able to model system's performance as it was using any off-chip memory.

The second milestone is that we have realized the implementation of compressed main memory system by integrating X-MatchPROVW engine and Cortex-M0 CPU into GRLIB SoC. While the previous work's system was only able to perform read instruction and on-the-fly decompression. We have successfully implemented on-the-fly compression and decompression function into the wrapper we designed called CM0X_wrapper. The CM0X_wrapper we designed adopts decompression buffer (that we call as buffer_ram) introduced by Lee et al[21], which concludes that third objective of second aim was achieved.

The third milestone is that we have evaluated how our system performs if slower memory such as off-chip SDRAM was used in the system. 53% speed gain by CM0X_wrapper if SDRAM was used was shown in chapter 7. Although the result was based on approximation, this approximation could not be realized if we did not develop CM0_wrapper, CM0X_busdelaycont_wrapper and CM0X_wrapper. This evaluation concludes that we have achieved all the objectives of the second aim we have mentioned.

In addition, we have implemented the design using Xilinx Virtex 5 FPGA board and observed the result of the system by observing the behavior of LED on the board. By this implementation, it proves that our compressed main memory system works and there is no latches introduced into the design.

8.2 Future Work

By completing this dissertation, we have just opened the first gate of compressed main memory system implementation with ARM Cortex-M0 CPU, X-MatchPROVW compression-decompression engine, and GRLIB SoC. We have implemented the system without the use of sophisticated translation table that helps to executes much larger program size. However, before we look at other sophisticated improvements. It is better if we could get the most accurate performance evaluation of the current system by using real SDRAM to run CM0X_wrapper instead of approximating its performance.

Our CM0_wrapper by itself works with real SDRAM provided that the program we run does not contain stack operation as shown in sub-chapter 6.2.2.

We have also narrowed down the problem with SDRAM which can be found in sub-chapter 6.2.4. Therefore, it is suggested for the future student to tackle this SDRAM problem in GRLIB immediately so that CPU can perform stack operation with SDRAM .

After SDRAM issue in GRLIB can be solved, we suggest that some optimizations can be implemented in CM0X_wrapper. One optimization which is relatively fast to approach is to implement a mechanism for CPU to fetch decompressed data immediately after X-MatchPROVW has placed the first decompressed data into buffer_ram. This mechanism is believed to further reduce decompression overheads.

Afterwards, we may look at implementation of address translation table so that the system may behave like cache-based system and able to run much larger program.

Bibliography

[1] Ekman, M.; Stenstrom, P.;, "A robust main-memory compression scheme," *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on* , vol., no., pp. 74-85, 4-8 June 2005 doi: 10.1109/ISCA.2005.6

[2] Hennessy J.L.; Patterson D.A.;, *Computer architecture: A quantitative approach,* 2nd Edition, San Fransisco: Morgan Kaufmann Publishers, 1996, pp. 6-7 & 16-17.

[3] Alted, F.; , "Why Modern CPUs Are Starving and What Can Be Done about It," *Computing in Science & Engineering* , vol.12, no.2, pp.68-71, March-April 2010
doi: 10.1109/MCSE.2010.51

[4] ARM, *An Introduction to Thumb*, Version 2.0, March 1995
URL: http://www.cse.unsw.edu.au/~pcb/LPC210x/Thumb_intro.pdf

[5] Aeroflex Gaisler, GRLIB IP Library User's Manual, Version 1.10 B4108, June 2001

[6] Xilinx, *ML505/ML506/ML507 Evaluation Platform User Guide*,Version 3.1.2, 16 May 2011
URL: http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf

[7] Nunez-Yanez, J.L.; Chouliaras, V.A.; , "Gigabyte per Second Streaming Lossless Data Compression Hardware Based on a Configurable Variable-Geometry CAM dictionary," *Computers and Digital Techniques, IEE Proceedings -* , vol.153, no.1, pp. 47- 58, 10 Jan. 2006
doi: 10.1049/ip-cdt:20045130

[8] Benini, L.; Bruni, D.; Macii, A.; Macii, E.; , "Hardware-assisted Data Compression for Energy Minimization in Systems with Embedded Processors," *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings* , vol., no., pp.449-453, 2002
doi: 10.1109/DATE.2002.998312

[9] Mahapatra N. et al , "A Limit Study on the Potential of Compression for Improving Memory System Performance, Power Consumption, and Cost," *Journal of Instruction-Level Parallelism* , Vol. 7, no. , pp. 1-37, July 2005.

[10] Snir M.; Yu J.;," On the Theory of Spatial and Temporal Locality ",Dept. Computer Science, University of Illinois ar Urbana-Champaign, 21 July 2005.Report No. UIUCDCS-R-2005-2611

[11] Wilson, P.R.; , "Operating System Support for Small Objects," *Object Orientation in Operating Systems, 1991. Proceedings., 1991 International Workshop on* , vol., no., pp.80-86, 17-18 Oct 1991
doi: 10.1109/IWOOOS.1991.183026

[12] Tuduce,I, "Adaptive Main Memory Compression," Ph.D. dissertation, Swiss Federal Institute of Technology Zurich .Unpublished Doctoral Thesis ETH No. 16327,2005.

[13] Kjelsø M.; Gooch.M.; Jones S.;, "Performance Evaluation of Computer Architectures with Main Memory Data Compression," *Journal of Systems Architecture*., vol. 45, no. 8, pp. 571-590, February 1999,ISSN 1383-7621,doi:10.1016/S1383-7621(98)00006-X

[14] Xianhong Xu; Jones, S.; , "Code Compression for the Embedded ARM/THUMB Processor " *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications,*

2003. Proceedings of the Second IEEE International Workshop on , vol., no., pp.31-35, 8-10 Sept. 2003,doi: 10.1109/IDAACS.2003.1249510

[15] Wolfe, A.; Chanin, A.; , "Executing Compressed Programs On An Embedded RISC Architecture," *Microarchitecture, 1992. MICRO 25., Proceedings of the 25th Annual International Symposium on* , vol., no., pp.81-91, 1-4 Dec 1992
doi: 10.1109/MICRO.1992.697002

[16] Lekatsas, H.; Wolf, W.; , "SAMC: a Code Compression Algorithm for Embedded Processors" *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* , vol.18, no.12, pp.1689-1701, Dec 1999
doi: 10.1109/43.811316

[17] Lefurgy, C.; Bird, P.; Chen, I.-C.; Mudge, T.; , "Improving code density using compression techniques," *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on* , vol., no., pp.194-203, 1-3 Dec 1997
doi: 10.1109/MICRO.1997.645810

[18] Huffman, D.A.; , "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE* , vol.40, no.9, pp.1098-1101, Sept. 1952
doi: 10.1109/JRPROC.1952.273898

[19] Howard P.G.; Vitter J.S.;, “Practical Implementations of Arithmetic Coding”, Dept. Computer Science, Brown University, Technical Report No. 92-18, April 1992.

[20] Bell T.; Witten I.; Cleary J.;, " Modelling for Text Compression," *ACM Computing Surveys (CSUR)*, Vol. 21, no. 4, pp. 557-591, December 1989,doi: 10.1145/76894.76896

[21] Jang-Soo Lee; Won-Kee Hong; Shin-Dug Kim; , "Design and evaluation of a selective compressed memory system," *Computer Design, 1999. (ICCD '99) International Conference on* , vol., no., pp.184-191, 1999
doi: 10.1109/ICCD.1999.808424

[22] Tremaine R. B. et al. , "IBM Memory Expansion Technology (MXT),"*IBM J. RES. & DEV.*, Vol. 45, no. 2, pp. 271-285, 2 March 2001.

[23] Abali, B.; Franke, H.; Xiaowei Shen; Poff, D.E.; Smith, T.B.; , "Performance of hardware compressed main memory, " *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on* , vol., no., pp.73-81, 2001
doi: 10.1109/HPCA.2001.903253

[24] Nunez-Yanez, J.L.; Jones, S.; Bateman, S.; , "X-MatchPRO: a high performance full-duplex lossless data compressor on a ProASIC FPGA," *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, International Workshop on, 2001.* , vol., no., pp.56-60, 2001
doi: 10.1109/IDAACS.2001.941979

[25] Kjelso, M.; Gooch, M.; Jones, S.; , "Design and performance of a main memory hardware data compressor ," *EUROMICRO 96. 'Beyond 2000: Hardware and Software Design Strategies'., Proceedings of the 22nd EUROMICRO Conference* , vol., no., pp.423-430, 2-5 Sep 1996
doi: 10.1109/EURMIC.1996.546466

[26] Pagiamtzis, K.; Sheikholeslami, A.; , "Content-addressable memory (CAM) circuits and architectures: a tutorial and survey," *Solid-State Circuits, IEEE Journal of* , vol.41, no.3, pp. 712- 727, March 2006
doi: 10.1109/JSSC.2005.864128

[27] Ziv, J.; Lempel, A.; , "A universal algorithm for sequential data compression," *Information Theory, IEEE Transactions on* , vol.23, no.3, pp. 337- 343, May 1977
doi: 10.1109/TIT.1977.1055714

[28] Lefurgy, C.; Piccininni, E.; Mudge, T.; , "Reducing code size with run-time decompression," *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on* , vol., no., pp.218-228, 2000
doi: 10.1109/HPCA.2000.824352

[29] ARM, *ARM Cortex-M0 DesignStart Release Note.* 4 August 2010

[30] Lin C.H.; Xie Y.; Wolf W.;, "LZW-Based Code Compression for VLIW Embedded Systems", *In Proc. of the Design, Automation and Test in Europe Conference*,vol.,no.,pp. 76-81, 2004

[31] Yiu J., *Definitive Guide to the ARM Cortex-M0*, Oxford: Newness, 2011, pp.1-35

[32] ARM, *AMBA 3 AHB-Lite Protocol Specification*, Version 1.0, 2006
URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0033a/index.html

[33] ARM, *AMBA Specification*, Revision 2.0, 2006
URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0011a/index.html

[34] Martos P.I.; Baglivo F.;, "Cortex-M0 Implementation on a Xilinx FPGA", Embedded Systems Laboratory, Faculty of Engineering – Univ. of Buenos Aires, Buenos Aires, Argentina.

[35] Xilinx, *Virtex-5 User Guide*, Version 5.4, March 16, 2012
URL: http://www.xilinx.com/support/documentation/user_guides/ug190.pdf

[36] Smith G.R., *FPGA 101: Everything you need to know to get started*, Oxford: Newness, 2010, p. 43-54.

[37] Xilinx, "Simulation Libraries" internet: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_simulation_libraries.htm, 2009 [10th April 2012].

[38] FPGA Central, "FPGA Synthesis and Implementation (Xilinx Design Flow)", Internet: http://www.fpgacentral.com/docs/fpga-tutorial/fpga-synthesis-and-implementation-xilinx, 2009 [10th April 2012].

[39] Xilinx, Synthesis and Simulation Guide, Version 12.3, 21st September 2010.
URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_4/sim.pdf

[40] Piekarek M. "Compressed Main Memory System for an ARM Cortex System-On-Chip". MEng. thesis, University of Bristol, 2012.

[41] Nunez Yanez J.L., "X-MatchPROVWCompression/Decompression FPGA Processor", *Preliminary Information*, Dept. of Electrical and Electronic Engineering, Loughborough University, May 2002.

[42] Xilinx, *Xilinx Power Tools Tutorial*, Version 1.0, 15th March 2010
URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ug733.pdf

[43] Bidmead C., “ARM Creators Sophie Wilson and Steve Furber”. Internet:
http://www.reghardware.com/2012/05/03/unsung_heroes_of_tech_arm_creators_sophie_wilson_and_steve_furber/ , 3rd May 2012 [12th May 2012]

[44] Wikipedia, *Endianness, r*etrieved 19:49, September 30, 2012
URL: http://en.wikipedia.org/w/index.php?title=Special:Cite&page=Endianness&id=515258095

[45] Gaisler,J., “A structured VHDL design method” *Fault-tolerant Microprocessors for Space Applications.* Gaisler Research.URL*:*http://www.gaisler.com/doc/vhdl2proc.pdf

Appendix

Appendix A: Proof of stack behavior in Heap_write_read2KB_CR05.c

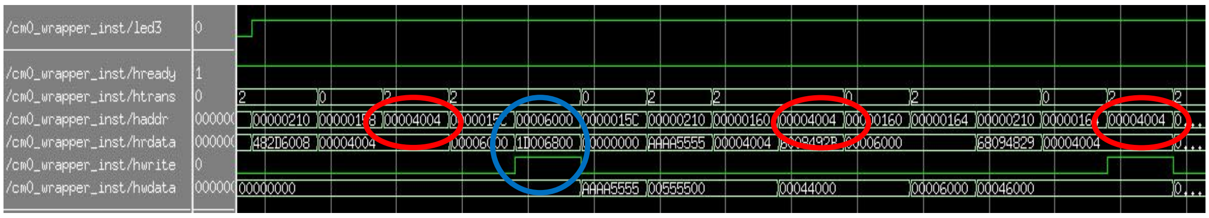


Figure 1: Proof that there is 2 stack read operation and one 1 stack write operation everytime CPU writes to heap in Heap_write_read2KB_CR05.c

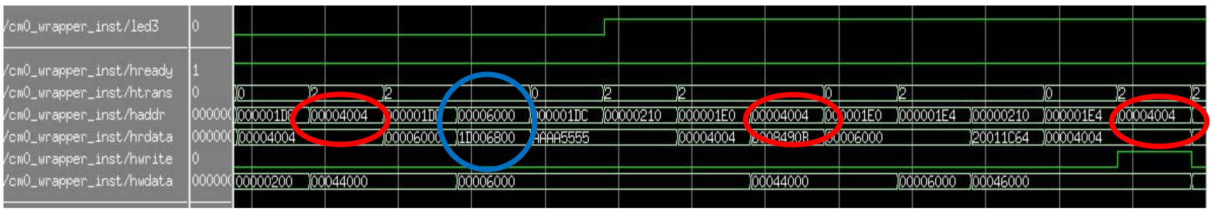


Figure 2: Proof that there is 2 stack read operation and one 1 stack write operation everytime CPU reads from heap in Heap_write_read2KB_CR05.c

We can see that blue circle marks heap write operation as 0x00006000 belong to heap address. Then we see that red circles marks stack operations since stack address starts from 0x00004000. Red circle with hrwrite = 1 is stack write operation , otherwise read operation.

Appendix B: Source codes designed by Agha

B.1 Cpu_ahb_bridge.vhd in CM0_wrapper

This is trimmed version, full version can be observed in submitted zip file.: \Sourcecodes_forassessment_ms1718\CompletelyDesigned by AGHA\CM0_wrapper files

```
begin

NMI <= '0';
IRQ <= (others=>'0');
RXEV <= '0';

ahbmo.hlock <= HMASTLOCK;
ahbmo.hburst<= HBURST;
ahbmo.hirq<= (others=>'0');

HRESP <= '0';

cpuahb_bridge:
process(PresSTATE,HWDATA,HSIZE,regin,regout,H
WRITE,HPROT,HTRANS,ahbmi.hrdata,ahbmi.hread
y,HADDR)
variable vreg : reg_type;
begin
vreg := regout;

vreg.ahbmo_hwdata  := HWDATA;
vreg.ahbmo_hsize   := HSIZE;
vreg.ahbmo_hwrite  := HWRITE;
vreg.ahbmo_hprot   := HPROT;
vreg.ahbmo_htrans  := HTRANS;
vreg.ahbmo_haddr   := HADDR;
vreg.m0_hrdata := ahbmi.hrdata;
vreg.m0_hready := ahbmi.hready;

if ahbmi.hready = '1' then-- when bus is ready
if HTRANS = "10" then-- and if cpu wants to do
transaction
vreg.ahbmo_hbusreq := '1';-- wrapper request to
the bus

else-- else if cpu is idle
if HWRITE_Prev = '1' then-- if previous cycle was
writing, (this if-else to prevent glitches and errors
after write operation.)
vreg.m0_hrdata := (others =>'0');--cpu should
not receive any data other than 0.
else
vreg.m0_hrdata := ahbmi.hrdata;
end if;
vreg.ahbmo_hbusreq := '0';--if cpu is idle ,
wrapper should stop request to bus.
end if;
else-- when bus is not ready
vreg.ahbmo_hbusreq := '0';-- wrapper should
not/stop request to bus
vreg.m0_hrdata := (others =>'0');-- cpu should
not receive any data other than 0.
end if;


regin <=  vreg;
ahbmo.hbusreq <= vreg.ahbmo_hbusreq;
ahbmo.haddr   <= vreg.ahbmo_haddr;
ahbmo.htrans  <= vreg.ahbmo_htrans;
ahbmo.hwdata  <= vreg.ahbmo_hwdata;
ahbmo.hwrite  <= vreg.ahbmo_hwrite;
ahbmo.hsize   <= vreg.ahbmo_hsize;
ahbmo.hprot   <= vreg.ahbmo_hprot;
HREADY <= vreg.m0_hready;
HRDATA <= vreg.m0_hrdata;
end process;



seq: process(RST,CLK)
begin
if rising_edge (CLK) then
HWRITE_Prev <= HWRITE;-- way to get previous
state of hwrite
regout <= regin;

end if;
end process;

end behavioral;
```

B.2 CM0_busdelaycont_wrapper.vhd in CM0_busdelaycont_wrapper

This is trimmed version, full version can be observed in submitted zip file.: Sourcecodes_forassessment_ms1718\CompletelyDesigned by AGHA\CM0_busdelaycont_wrapper filesbegin

```
begin
vreg := regout;

if RST = '0' then
vreg.delaycount := 0;
vreg.ahbmo_hbusreq := '0'; -- busreq lasts for 1
cycle only
vreg.ahbmo_htrans := "00"; -- htrans lasts for 1
cycle only
vreg.ahbmo_haddr := (others=>'0');
vreg.ahbmo_hsize := (others=>'0');
vreg.ahbmo_hwrite:= '0';
vreg.ahbmo_hwdata:= (others=>'0');
vreg.ahbmo_hprot := (others=>'0');
vreg.ahbmo_hburst:= (others=>'0');

vreg.MI_hrdata := (others=>'0');
vreg.MI_hready := '0';
NextSTATE <= PresSTATE;
else


NextSTATE <= PresSTATE ; -- avoiding latch.

case PresSTATE is

when stReady =>
vreg.MI_hrdata := AHBMI.HRDATA;             --
transfer data from bus to cpu.
vreg.MI_hready := AHBMI.HREADY;             --
transparents ready signal from bus to cpu

vreg.ahbmo_hbusreq := '0';          --request to
the bus should not happen in this moment
vreg.ahbmo_htrans := "00";
if MO.htrans = "10" then        -- when cpu wants
to do a transaction, store its requests parameter to
register, state goes to halt state.
vreg.MO_haddr := MO.HADDR;
vreg.MO_hprot := MO.HPROT;
vreg.MO_hsize := MO.HSIZE;
vreg.MO_hwrite:= MO.HWRITE;
vreg.MO_hwdata:= MO.HWDATA;
vreg.MO_hburst:= MO.HBURST;
NextSTATE <= stHalt;
end if;



when stHalt  =>
                                    -- stHalt . hold
issuance to bus for "delaycount" + 1 cycles.
vreg.MI_hready := '0';
vreg.MO_hwdata:= MO.HWDATA;
                       -- hwdata should be transparent
from cpu to bus when halting.
vreg.ahbmo_hwdata:= regout.MO_hwdata;
if vreg.delaycount = 5  then   -- defines duration of
data phase to delay the bus and mimic slower
memory. for mimicking SDRAM, put value as 5
vreg.delaycount := 0;
vreg.ahbmo_hbusreq := '1';-- bus is requested for
1 cycle
vreg.ahbmo_htrans := "10"; -- htrans lasts for 1
cycle only
vreg.ahbmo_haddr := regout.MO_haddr;
vreg.ahbmo_hsize := regout.MO_hsize;
vreg.ahbmo_hwrite:= regout.MO_hwrite;
vreg.ahbmo_hprot := regout.MO_hprot;
vreg.ahbmo_hburst:= regout.MO_hburst;
NextSTATE <= stReady;
else
vreg.delaycount := regout.delaycount + 1;
end if;
end case;
end if;


regin <= vreg;

AHBMO.HBUSREQ <= vreg.ahbmo_hbusreq;
AHBMO.HTRANS <= vreg.ahbmo_htrans;
AHBMO.HADDR  <= vreg.ahbmo_haddr;
AHBMO.HSIZE  <= vreg.ahbmo_hsize;
AHBMO.HWRITE <= vreg.ahbmo_hwrite;
AHBMO.HPROT  <= vreg.ahbmo_hprot;
AHBMO.HWDATA <= vreg.ahbmo_hwdata;
AHBMO.HBURST <= vreg.ahbmo_hburst;

MI.HRDATA <= vreg.MI_hrdata;
MI.HREADY <= vreg.MI_hready;

end process;

process(CLK,RST)
begin
if RST = '0' then
PresSTATE <= stReady;
regout <= regin;
elsif rising_edge (CLK) then
regout <= regin;
PresSTATE <= NextSTATE;
end if;
end process;

end behavioral;
```

B.3 Cpu_ahbbuffer_bridge.vhd in CM0X_wrapper

This is trimmed version, full version can be observed in submitted zip file.: Sourcecodes_forassessment_ms1718\CompletelyDesigned by AGHA\CM0X_wrapper files

```
begin
vreg:= regout;

voffsetaddr := HADDR - X"4000";
vHADDR_rw_buffer := "00" & voffsetaddr(31
downto 2);


voffsetaddr_prev := HADDR_Prev - X"4000";
vHADDR_rw_buffer_Prev := "00"&
voffsetaddr_prev (31 downto 2);

if RST = '0' then
vreg.comp_start := '0';
vreg.dec_start := '0';
vreg.buffer_read_startaddr := (others => '0');
vreg.buffer_write_startaddr := (others => '0');
vreg.ahbram_read_startaddr := (others => '0');
vreg.ahbram_write_startaddr := (others => '0');
vreg.delayready := '0';
vreg.heapwritecount := 0;
vreg.count0x6000 := 0;
vreg.startup_done := '0';
vreg.comptest_done := '0';
NextSTATE <= PresSTATE;

else

NextSTATE <= PresSTATE; -- to avoid latches
vreg.startup_done := regout.startup_done;
vreg.comptest_done := regout.comptest_done;
case PresSTATE is

-- start first decompression and initializes signals
when st_start =>
vreg.dec_start := '1'; -- start decompression
vreg.hready := '0';
vreg.hrdata := (others=>'0');
vreg.enb := '0';
vreg.addrb := (others=>'0');

vreg.web := '0';
vreg.dinb := (others =>'0');

vreg.comp_start := '0';
vreg.buffer_read_startaddr := (others => '0');
vreg.buffer_write_startaddr := (others => '0');
vreg.ahbram_read_startaddr := (others => '0');
vreg.ahbram_write_startaddr := (others => '0');
vreg.comptest_done := '0';

vreg.cpu2ahb_htrans := (others => '0');
vreg.cpu2ahb_hwdata := (others => '0');
vreg.cpu2ahb_hprot  := (others => '0');
vreg.cpu2ahb_hsize  := (others => '0');
vreg.cpu2ahb_haddr  := (others => '0');
vreg.cpu2ahb_hwrite := '0';
vreg.cpu2ahb_hbusreq := '0';
NextSTATE<= st_waitfirstdec;

-- wait until first decompression is finished.
when st_waitfirstdec =>
if dec_finished = '1' then
vreg.dec_start := '0';
NextSTATE <= st_idle;
else
NextSTATE <= st_waitfirstdec;
end if;

-- idle state to wait for cpu transaction request
-- and acts as address phase of a transaction
when st_idle =>
vreg.hready := READY;
vreg.heapbufferaccess := '0';
vreg.enb := '0';
vreg.web := '0';

if vreg.hready = '1' then

-- **case of read instructions address phase.
Communicate with buffer
if HTRANS = "10" and HADDR < rw_stack_base
then
vreg.addrb := HADDR_ro (11 downto 0);
vreg.enb:= HTRANS(1);
NextSTATE <= st_read_inst;

-- **case of stack operation address phase,
communicate directly with bus
elsif HTRANS = "10" and HADDR >=
rw_stack_base and HADDR < rw_heap_base then
vreg.enb := '0';
vreg.cpu2ahb_htrans := HTRANS;
vreg.cpu2ahb_hwdata := HWDATA;
vreg.cpu2ahb_hprot  := HPROT;
vreg.cpu2ahb_hsize  := HSIZE;
vreg.cpu2ahb_haddr  := HADDR;
vreg.cpu2ahb_hwrite := HWRITE;
vreg.cpu2ahb_hbusreq := '1';
NextSTATE <= st_readwrite_data;

-- **case of heap operation address phase.
Communicate with buffer
elsif HTRANS = "10" and HADDR >=
rw_heap_base and HADDR < rw_heap_top then --
vreg.enb := HTRANS(1);
vreg.addrb := vHADDR_rw_buffer(11 downto 0);
vreg.heapbufferaccess := '1';
NextSTATE <= st_readwrite_data;

-- **case of other operation address phase.
Communicate directly with bus
elsif HTRANS = "10" and HADDR >= rw_heap_top
then
vreg.enb := '0';
vreg.cpu2ahb_htrans := HTRANS;
vreg.cpu2ahb_hwdata := HWDATA;
vreg.cpu2ahb_hprot  := HPROT;
vreg.cpu2ahb_hsize  := HSIZE;
vreg.cpu2ahb_haddr  := HADDR;
vreg.cpu2ahb_hwrite := HWRITE;
vreg.cpu2ahb_hbusreq := '1';
NextSTATE <= st_readwrite_data;
else
NextSTATE <= st_idle;
end if;
else
-- if hready is 0 then hrdata should be 0
vreg.hrdata := (others=>'0');
NextSTATE <= st_idle;
```

```
end if;


-- state of data phase of reading instruction.
when st_read_inst =>
vreg.hready := READY;
vreg.heapbufferaccess := '0';
vreg.web := '0';
if vreg.hready = '1' then
-- **if cpu removes transaction request at this
point
if HTRANS = "00" then
-- Data Phase of read_inst only
vreg.hrdata := doutb;
vreg.addrb := HADDR_ro (11 downto 0);
vreg.enb:= HTRANS (1);
NextSTATE <= st_idle;

-- **if another address phase of read_inst happens
at this point
elsif HTRANS = "10" and HADDR < rw_stack_base
then
-- Address phase  & Data phase of read_inst
vreg.hrdata := doutb;
vreg.addrb := HADDR_ro (11 downto 0);
vreg.enb:= HTRANS (1);
NextSTATE <= st_read_inst;

-- **if address phase of stack operation happens
at this point
elsif HTRANS = "10" and HADDR >=
rw_stack_base and HADDR < rw_heap_base then
-- Data phase of read_inst
vreg.hrdata := doutb;
vreg.enb := HTRANS (1);
-- Address phase of stack operation
vreg.cpu2ahb_htrans := HTRANS;
vreg.cpu2ahb_hwdata := HWDATA;
vreg.cpu2ahb_hprot  := HPROT;
vreg.cpu2ahb_hsize  := HSIZE;
vreg.cpu2ahb_haddr  := HADDR;
vreg.cpu2ahb_hwrite := HWRITE;
vreg.cpu2ahb_hbusreq := '1';
NextSTATE <= st_readwrite_data;

-- **if address phase of heap operation happens at
this point
elsif HTRANS = "10" and HADDR >=
rw_heap_base and HADDR < rw_heap_top then
-- Data phase of read_inst
vreg.hrdata := doutb;
vreg.enb := HTRANS (1);
-- Address phase of heap operation
vreg.addrb := vHADDR_rw_buffer(11 downto 0);
vreg.heapbufferaccess := '1';
NextSTATE <= st_readwrite_data;

-- **if address phase of other operation happens
at this point
elsif HTRANS = "10" and HADDR >= rw_heap_top
then
vreg.enb := HTRANS (1);
vreg.hrdata := doutb;
vreg.cpu2ahb_htrans := HTRANS;
vreg.cpu2ahb_hwdata := HWDATA;
vreg.cpu2ahb_hprot  := HPROT;
vreg.cpu2ahb_hsize  := HSIZE;
vreg.cpu2ahb_haddr  := HADDR;
vreg.cpu2ahb_hwrite := HWRITE;
vreg.cpu2ahb_hbusreq := '1';
NextSTATE <= st_readwrite_data;
```

```
else null;

end if;

else
vreg.hrdata := (others=>'0');
NextSTATE <= st_read_inst;
end if;

-- state of data phase of reading/writing data.
when st_readwrite_data=>
vreg.hready := READY;

if vreg.hready = '1' then
-- **if cpu removes transaction request at this
point
if HTRANS = "00" then
-- Data Phase of reading/writing data.
-- if heap operation is happening, data phase of
heap operation
if vreg.heapbufferaccess = '1' then
vreg.heapbufferaccess := '0';
if HWRITE_Prev = '1' then --heap write transfer
data phase
vreg.enb := '1';
vreg.web := '1';
vreg.addrb := vHADDR_rw_buffer_Prev (11
downto 0);
vreg.dinb := HWDATA; -- although htrans is 0,
processor still in its dataphase for writing.
important.
vreg.hrdata := (others => '0');
NextSTATE <= st_idle;
else --heap read transfer data phase
vreg.enb := '0';
vreg.web := '0';
vreg.hrdata := doutb;
NextSTATE <= st_idle;
end if;

-- if stack operation is happening, data phase of
stack operation
else
vreg.cpu2ahb_htrans := HTRANS;
vreg.cpu2ahb_hwdata := HWDATA;
vreg.cpu2ahb_hprot  := HPROT;
vreg.cpu2ahb_hsize  := HSIZE;
vreg.cpu2ahb_haddr  := HADDR;
vreg.cpu2ahb_hwrite := HWRITE;
vreg.cpu2ahb_hbusreq := '0';
vreg.HRDATA := ahb2cpu_hrdata;
vreg.enb := '0';
NextSTATE <= st_idle;
end if;

--**if address phase of read_inst happens at this
point
elsif HTRANS = "10" and HADDR < rw_stack_base
then
--Address phase of read_inst
vreg.addrb := HADDR_ro (11 downto 0);
vreg.enb:= HTRANS (1);
NextSTATE <= st_read_inst;

-- Data Phase of reading/writing data.
-- if heap operation is happening, data phase of
heap operation
if vreg.heapbufferaccess = '1' then
vreg.heapbufferaccess := '0';
if HWRITE_Prev = '1' then --heap write transfer
data phase
```

```
vreg.enb := '1';
vreg.web := '1';
vreg.addrb := vHADDR_rw_buffer_Prev (11
downto 0);
vreg.dinb := HWDATA; -- although htrans is 0,
processor still in its dataphase for writing.
important.
vreg.hrdata := (others => '0');

vreg.delayready := '1';
else --heap read transfer data phase
vreg.hrdata := doutb;
end if;

-- if stack operation is happening, data phase of
stack operation
else
vreg.cpu2ahb_htrans := HTRANS;
vreg.cpu2ahb_hwdata := HWDATA;
vreg.cpu2ahb_hprot  := HPROT;
vreg.cpu2ahb_hsize  := HSIZE;
vreg.cpu2ahb_haddr  := HADDR;
vreg.cpu2ahb_hwrite := HWRITE;
vreg.cpu2ahb_hbusreq := '0';
vreg.HRDATA := ahb2cpu_hrdata;--at this time,
still reading from bus due to and data phase of
rw_region
vreg.enb:= HTRANS(1);
end if;

--**if address phase of stack operation happens
again at this point
elsif HTRANS = "10" and HADDR >=
rw_stack_base and HADDR < rw_heap_base then
--Address phase of stack_operation, directly to bus
vreg.cpu2ahb_htrans := HTRANS;
vreg.cpu2ahb_hwdata := HWDATA;
vreg.cpu2ahb_hprot  := HPROT;
vreg.cpu2ahb_hsize  := HSIZE;
vreg.cpu2ahb_haddr  := HADDR;
vreg.cpu2ahb_hwrite := HWRITE;
vreg.cpu2ahb_hbusreq := '1';
NextSTATE <= st_readwrite_data;

-- Data Phase of reading/writing data.
-- if heap operation is happening, data phase of
heap operation
if vreg.heapbufferaccess = '1' then
vreg.heapbufferaccess := '0';
if HWRITE_Prev = '1' then --heap write transfer
data phase
vreg.enb := '1';
vreg.web := '1';
vreg.addrb := vHADDR_rw_buffer_Prev (11
downto 0);
vreg.dinb := HWDATA; -- although htrans is 0,
processor still in its dataphase for writing.
important.
vreg.hrdata := (others => '0');
else --heap read transfer data phase
vreg.hrdata := doutb;
end if;

-- if stack operation is happening, data phase of
stack operation
else
vreg.HRDATA := ahb2cpu_hrdata; --at this time,
still reading from bus due to and data phase of
rw_region
vreg.enb:= HTRANS(1);
end if;

--**if address phase of heap operation happens
again at this point
elsif HTRANS = "10" and HADDR >=
rw_heap_base and HADDR < rw_heap_top then
-- Address phase of heap operation
vreg.enb := HTRANS (1);
vreg.addrb := vHADDR_rw_buffer(11 downto 0);
NextSTATE <= st_readwrite_data;

-- Data Phase of reading/writing data.
-- if heap operation is happening, data phase of
heap operation
if vreg.heapbufferaccess = '1' then
--vreg.heapbufferaccess := '0';
if HWRITE_Prev = '1' then --heap write transfer
data phase
vreg.enb := '1';
vreg.web := '1';
vreg.addrb := vHADDR_rw_buffer_Prev (11
downto 0);
vreg.dinb := HWDATA; -- although htrans is 0,
processor still in its dataphase for writing.
important.
vreg.hrdata := (others => '0');
else --heap read transfer data phase
vreg.hrdata := doutb;
end if;

-- if stack operation is happening, data phase of
stack operation
else
vreg.heapbufferaccess := '1';
vreg.cpu2ahb_htrans := HTRANS;
vreg.cpu2ahb_hwdata := HWDATA;
vreg.cpu2ahb_hprot  := HPROT;
vreg.cpu2ahb_hsize  := HSIZE;
vreg.cpu2ahb_haddr  := HADDR;
vreg.cpu2ahb_hwrite := HWRITE;
vreg.cpu2ahb_hbusreq := '0';
end if;

--**if address phase of other operation happens
again at this point
elsif HTRANS = "10" and HADDR >= rw_heap_top
then
-- Address phase of other operation, directly to bus
vreg.cpu2ahb_htrans := HTRANS;
vreg.cpu2ahb_hwdata := HWDATA;
vreg.cpu2ahb_hprot  := HPROT;
vreg.cpu2ahb_hsize  := HSIZE;
vreg.cpu2ahb_haddr  := HADDR;
vreg.cpu2ahb_hwrite := HWRITE;
vreg.cpu2ahb_hbusreq := '1';
NextSTATE <= st_readwrite_data;

-- Data Phase of reading/writing data.
-- if heap operation is happening, data phase of
heap operation
if vreg.heapbufferaccess = '1' then
vreg.heapbufferaccess := '0';
if HWRITE_Prev = '1' then --heap write transfer
data phase
vreg.enb := '1';
vreg.web := '1';
vreg.addrb := vHADDR_rw_buffer_Prev (11
downto 0);
vreg.dinb := HWDATA; -- although htrans is 0,
processor still in its dataphase for writing.
important.
```

```
vreg.hrdata := (others => '0');
else --heap read transfer data phase
vreg.hrdata := doutb;
end if;

-- if stack operation is happening, data phase of
stack operation
else
vreg.HRDATA := ahb2cpu_hrdata; --at this time,
still reading from bus due to and data phase of
rw_region
vreg.enb:= HTRANS(1);
end if;
end if;
else
vreg.hrdata := (others=>'0');
NextSTATE <= st_readwrite_data;
end if;


when others => null;

end case;

end if;

--
================================
========================
-- Comdec Ignitor
-- section below is to start
compression/decompression
-- for future works, this area of code may be
replaced with address decoding
-- or use tranlation table
--
================================
========================
if HADDR = X"00006000" and HWRITE = '1' then
vreg.count0x6000 := vreg.count0x6000 + 1;
vreg.startup_done := '0';

elsif regout.count0x6000 = 2 then
vreg.startup_done := '1';
vreg.buffer_read_curaddr := X"00000800";
vreg.ahbram_write_curaddr := X"00006000";
vreg.heapwritecount := regout.heapwritecount +
1; --integer
vreg.count0x6000 := 0;
else
vreg.startup_done := regout.startup_done;
end if;


--end if;

if vreg.startup_done = '1' then
if vreg.heapbufferaccess = '1' and HWRITE = '1'
then
vreg.heapwritecount := regout.heapwritecount +
1; --integer
elsif regout.heapwritecount = 512 then  --2048
bytes
vreg.buffer_read_startaddr :=
regout.buffer_read_curaddr;
vreg.buffer_read_curaddr :=
regout.buffer_read_curaddr + "100"; -- increment
the current buffer read address pointer by 256
location for the next heap compression.
vreg.ahbram_write_startaddr :=
regout.ahbram_write_curaddr;
vreg.heapwritecount := 0;--reset counter to 0
vreg.comp_start := '1'; -- start comp
end if;


if comp_finished = '1' then
vreg.comp_start := '0';
vreg.ahbram_write_curaddr :=
ahbram_write_lastaddr;
end if;

if vreg.heapbufferaccess = '1' and HWRITE = '0'
and HADDR = X"00006000" and HADDR_Prev /=
X"00004004" then
vreg.dec_start := '1';
vreg.ahbram_read_startaddr := X"00006000";
vreg.buffer_write_startaddr := X"00000800";
end if;



end if;

if doutb = X"abcd1234" then
vreg.ahbram_read_startaddr := X"00000270";
vreg.buffer_write_startaddr := X"00000800";
vreg.dec_start := '1';
elsif doutb = X"1234abcd" and
vreg.comptest_done = '0' then
vreg.ahbram_write_startaddr := X"00000270";
vreg.buffer_read_startaddr := X"00000800";
vreg.comp_start := '1';
vreg.comptest_done := '1';

else null;
end if;


if comp_finished = '1' then
vreg.comp_start := '0';
vreg.ahbram_write_curaddr :=
ahbram_write_lastaddr;
end if;

if dec_finished = '1' then
vreg.dec_start := '0';
end if;

--
================================
============================
-- end of Comdec Ignitor
--
================================
============================


regin <= vreg;
dec_start <= regout.dec_start;
HRDATA <= vreg.hrdata;

-- important to solve clash of heap write data
phase and read instruction address phase
if vreg.delayready = '1' then
HREADY <= '0';
regin.delayready <= '0';
else
HREADY <= vreg.hready;
```

```
end if;


addrb <= vreg.addrb;
enb <= vreg.enb;
web(0) <= vreg.web;
dinb <= vreg.dinb;
heapbufferaccess<= vreg.heapbufferaccess;


comp_start <= vreg.comp_start;
buffer_read_startaddr <=
vreg.buffer_read_startaddr;
buffer_write_startaddr <=
vreg.buffer_write_startaddr;
ahbram_read_startaddr  <=
vreg.ahbram_read_startaddr;
ahbram_write_startaddr <=
vreg.ahbram_write_startaddr;

cpu2ahb_htrans <= vreg.cpu2ahb_htrans;
cpu2ahb_hwdata<=vreg.cpu2ahb_hwdata;
cpu2ahb_hprot<=vreg.cpu2ahb_hprot;
cpu2ahb_hsize<=vreg.cpu2ahb_hsize;
cpu2ahb_haddr<=vreg.cpu2ahb_haddr;
cpu2ahb_hwrite<=vreg.cpu2ahb_hwrite;
cpu2ahb_hbusreq<=vreg.cpu2ahb_hbusreq;

end process;



seq: process (CLK,RST)
begin

if RST = '0' then
PresSTATE <= st_start;
elsif rising_edge (CLK) then
PresSTATE <= NextSTATE;
HADDR_Prev <= HADDR;
HWRITE_Prev <= HWRITE;
regout <= regin;
else null;
end if;

end process;
```

B.4 Comdec_ahbbuffer_bridge.vhd in CM0X_wrapper

This is trimmed version, full version can be observed in submitted zip file.: Sourcecodes_forassessment_ms1718\CompletelyDesigned by AGHA\CM0X_wrapper files

```
begin
ahbmo.hbusreq   <= '0';
ahbmo.hwrite    <= '0';
ahbmo.hprot     <= "0011";
ahbmo.hsize     <="010";
ahbmo.hwdata    <= (others=>'0');
ahbmo.htrans    <= "00";
ahbmo.haddr     <= (others=>'0');
ahbmo.hburst    <= (others=>'0');
ahbmo_haddr_prev <= (others => '0');



ADDRESS         <= (others => '0');
CONTROLW        <= (others=>'0');
CS       <= '1';
RW       <= '1';

C_DATAIN <= (others=>'0');
BUS_ACKNOWLEDGE_DC <= '1';
BUS_ACKNOWLEDGE_DU <= '1';
dec_finished <= '0';
--waitfirst_cdatain <= '1';
waitfirst_udataout <= '1';

dina <= (others=>'0');
wea(0) <= '0';
ena <= '0';
addra <= (others=>'0');


regin.cdatain <= (others=>'0');
regin.ahbmo_haddr <= (others=>'0');
regin.addra <= (others=>'0');
regin.busack_dc <= '1';
regin.ahbmo_htrans <= (others=>'0');
regin.wea <= '0';
regin.dina <= (others=>'0');
regin.ena <= '0';

regin.ahbmo_hwdata <= (others=>'0');

regin.ahbmo_hbusreq <= '0';

regin.ahbmo_hwrite <= '0';
regin.busack_cu <= '1';

regin.udatain <= (others=>'0');

regin.addra_prev <= buffer_read_startaddr(11
downto 0); -- important signals
regin.ahbmo_haddr_prev <= (others=>'0');

comp_finished <= '0';
BUS_ACKNOWLEDGE_CC <= '1';
BUS_ACKNOWLEDGE_CU <= '1';
waitfirst_cdataout <= '1';
waitfirst_udatain <= '1';
wait_cu <= '1';
U_DATAIN <= (others=>'0');

NextSTATE <= PresSTATE;

s_READY <= ahbmi.hready;
s_ahbram_write_lastaddr <= (others=> '0');
ahb2cpu_hrdata <= (others=>'0');

case PresSTATE is

-- when state in idle, it directs the transaction from
cpu to bus.
-- This kind of transaction is only for STACK
operation.
when st_idle =>
ahbmo.htrans <= cpu2ahb_htrans;
ahbmo.haddr <= cpu2ahb_haddr;
ahbmo.hwrite <= cpu2ahb_hwrite;
ahbmo.hprot <= cpu2ahb_hprot;
ahbmo.hsize <= cpu2ahb_hsize;
ahbmo.hwdata <= cpu2ahb_hwdata;
ahbmo.hbusreq <= cpu2ahb_hbusreq;
ahb2cpu_hrdata <= ahbmi.hrdata;
s_READY <= ahbmi.hready;


if comp_start = '1' then
NextSTATE <= stcomp_init_ubsr;
s_READY <= '0';

elsif dec_start  = '1' then
s_READY <= '0';
NextSTATE <= stdec_init_ubsr;
else
s_READY <= ahbmi.hready;
NextSTATE <= st_idle;
end if;
--
****************************************
*************
--Decompression STATES
--
****************************************
*************
-- initialize ubsr for decompression
when stdec_init_ubsr=>

ahbmo.hburst<= "001";
C_DATAIN <= (others=>'0');
BUS_ACKNOWLEDGE_DC <= '1';
CS <= '0';
RW <= '0';
ADDRESS <= "1001";
CONTROLW <= X"00000BFF"; -- 3071Bblock
--CONTORL <= X"00000400"; -- 1024B block
--CONTROL <= X"00001000"; -- 4096B block
--CONTROL <= X"00002000"; -- 8192B block
--CONTROL <= X"00003000"; -- 12288B block
--CONTROL <= X"00004000"; -- 16384B block
s_READY <= '0';

NextSTATE <= stdec_init_cr;

-- initialize cr for decompression
when stdec_init_cr =>

CS <= '0';
RW <= '0';
ADDRESS <= "1000";
CONTROLW <= X"00004200";
ahbmo.hburst<= "001";
```

```
regin.cdatain <= (others=>'0');
regin.ahbmo_haddr <= ahbram_read_startaddr; --
initialize ahbmo.haddr for reading instructions.
regin.addra <= buffer_write_startaddr(11 downto
0) - '1'; -- initialize buffer address for writing.
regin.busack_dc <= '1';
regin.ahbmo_htrans <= "10";
regin.wea <= '0';
regin.dina <= (others=>'0');
regin.ena <= '0';
s_READY <= '0';

-- start requesting to bus
if BUS_REQUEST_DC = '0' then
CS <= '1';
RW <= '1';
CONTROLW <= (others=>'0');
ADDRESS <= (others=>'0');
ahbmo.hbusreq <= '1';
ahbmo.htrans <= "10"; --nonseq

NextSTATE <= stdec_decompressing;
else
NextSTATE <= stdec_init_cr;
end if;

-- state for decompression process
when stdec_decompressing =>
vreg:= regout;

ahbmo.hburst<= "001";
s_READY <= '0';
if FINISHED_D ='0' then
NextSTATE <= stdec_finish;

elsif ahbmi.hready = '1' then

-- to prevent problem when bus request dc is high.
if BUS_REQUEST_DC = '1' then
vreg.busack_dc := '1';
vreg.ahbmo_haddr := vreg.ahbmo_haddr_prev;
else
-- fetching data from main_ram to comdec engine
vreg.busack_dc := '0';
vreg.cdatain := to_bitvector(ahbmi.hrdata);
vreg.ahbmo_htrans := "11";
vreg.ahbmo_haddr := vreg.ahbmo_haddr + "100";
vreg.ahbmo_haddr_prev :=  regout.ahbmo_haddr
- "100";
NextSTATE <= stdec_decompressing;
end if;
else
NextSTATE <= stdec_decompressing;
end if;



if Decompressing = '0' then
vreg.wea := '1';
vreg.ena := '1';
else null;
end if;

--filling the buffer with uncompressed data
if U_DATA_VALID = '0' then
vreg.dina := U_DATAOUT;
vreg.addra := vreg.addra + '1'; -- incrementing
buffer address.

else
waitfirst_udataout <= '1';
end if;

BUS_ACKNOWLEDGE_DU <= BUS_REQUEST_DU;

regin <= vreg;
BUS_ACKNOWLEDGE_DC <= regout.busack_dc;
C_DATAIN <= regout.cdatain;
ahbmo.htrans <= regout.ahbmo_htrans;
ahbmo.haddr <= regout.ahbmo_haddr;
ahbmo_haddr_prev <=
regout.ahbmo_haddr_prev;

wea(0)<=regout.wea;
ena <= regout.ena;
dina <= regout.dina;
addra <= regout.addra;

-- when decompression finishes, re-initialize
signals.
when stdec_finish =>
s_READY <= '0';
BUS_ACKNOWLEDGE_DC <= '1';
BUS_ACKNOWLEDGE_DU <= '1';
ahbmo.hbusreq <= '0';
ahbmo.htrans <= "00";
waitfirst_cdatain <= '1';
waitfirst_udataout <= '1';
wea(0) <= '0';
ena <= '0';
dec_finished<= '1';

ahbmo.hburst<= "000";

NextSTATE <= st_idle;
--
****************************************
*************
--End of Decompression STATES
--
****************************************
*************


--
****************************************
*************
--Compression STATES
--
****************************************
*************
-- initialize ubsr for compression
when stcomp_init_ubsr =>
s_READY <= '0';
CS <= '0';
RW <= '0';

ADDRESS <= "1101";
CONTROLW <= X"00000BFF"; -- 3071B
--CONTORL <= X"00000400"; -- 1024B block
--CONTROL <= X"00001000"; -- 4096B block
--CONTROL <= X"00002000"; -- 8192B block
--CONTROL <= X"00003000"; -- 12288B block
--CONTROL <= X"00004000"; -- 16384B block



NextSTATE <= stcomp_init_cr;

-- initialize cr for compression
when stcomp_init_cr =>
s_READY <= '0';
```

```
ADDRESS <= "1100";
CONTROLW <= X"00004208";
-- 0x4080 for test mode of compression
-- 0x4088 for normal mode of compression
CS <= '0';
RW <= '0';

if BUS_REQUEST_CU = '0' then
CS <= '1';
RW <= '1';
--ena <= '1';

regin.ahbmo_hwdata <= (others=>'0');
regin.ahbmo_htrans <= "00";
regin.ahbmo_hbusreq <= '0';
regin.ahbmo_haddr  <= ahbram_write_startaddr;
-- initialize main_ram address for writing.
regin.ahbmo_hwrite <= '0';
regin.busack_cu <= '1';

regin.udatain <= (others=>'0');
regin.addra <= buffer_read_startaddr(11 downto
0) - '1'; -- initialize buffer address for reading
regin.addra_prev <= buffer_read_startaddr(11
downto 0);
regin.ena <= '1';

NextSTATE <= stcomp_compressing;
else
NextSTATE <= stcomp_init_cr;
end if;

-- state for compression process
when stcomp_compressing =>
vreg:=regout;
s_READY <= '0';
if FINISHED_C ='0' then

vreg.ahbmo_hwdata := (others=>'0'); --
important, due to the nature of bus write, at this
cycle ,
--it still writes the prev hwdata value, so it must
be neutralized to avoid chaos

NextSTATE <= stcomp_finish;
elsif ahbmi.hready = '1' then

if BUS_REQUEST_CU = '1' then
vreg.busack_cu := '1';
vreg.addra := vreg.addra_prev;
else

-- fetch data/instructions from buffer to comdec
engine
vreg.ena := '1';
vreg.busack_cu := '0';
vreg.udatain := to_bitvector(douta);
vreg.addra := vreg.addra + '1';
vreg.addra_prev := vreg.addra - '1';

NextSTATE <= stcomp_Compressing;
end if;
else
NextSTATE <= stcomp_Compressing;
end if;



if Compressing = '0' then
vreg.ahbmo_htrans := "11";
vreg.ahbmo_hwrite := '1';
vreg.ahbmo_hbusreq := '1';

else
null;
end if;

--filling the main_ram with compressed data
if C_DATA_VALID = '0' then   -- when writing ,
address and data are asserted in the same clock

vreg.ahbmo_hwdata := C_DATAOUT;
vreg.ahbmo_haddr := regout.ahbmo_haddr +
"100";   -- here change the address to rw region

else
null;
end if;

BUS_ACKNOWLEDGE_CC <= BUS_REQUEST_CC;

regin <= vreg;
ahbmo.hwdata <= vreg.ahbmo_hwdata;
ahbmo.htrans <= vreg.ahbmo_htrans;
ahbmo.hbusreq <= vreg.ahbmo_hbusreq;
ahbmo.haddr <= vreg.ahbmo_haddr;
ahbmo.hwrite <= vreg.ahbmo_hwrite;
BUS_ACKNOWLEDGE_CU <= vreg.busack_cu;

U_DATAIN <= vreg.udatain;
addra <= vreg.addra;
addra_prev <= vreg.addra_prev;
ena <= vreg.ena;
s_ahbram_write_lastaddr <= vreg.ahbmo_haddr;

when stcomp_finish =>
s_READY <= '0';
BUS_ACKNOWLEDGE_CC <= '1';
BUS_ACKNOWLEDGE_CU <= '1';
ahbmo.htrans <= "00";
ahbmo.hbusreq <= '0';
ahbmo.hwrite <= '0';
waitfirst_cdataout <= '1';
waitfirst_udatain <= '1';
wea(0) <= '0';
ena <= '0';
wait_cu <= '1';
comp_finished <= '1';
NextSTATE <= st_idle;
--
****************************************
*************
--End of Compression STATES
--
****************************************
*************
when others => s_READY <= '0';
s_ahbram_write_lastaddr <= (others=>'0');

end case;
end process;
STMseq: process (CLK,RST)
begin

if RST = '0' then
PresSTATE <= st_idle;
elsif rising_edge (CLK) then
PresSTATE<= NextSTATE;
HRDATA_Prev <= HRDATA;
regout<= regin;end if;
```