Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Øivind Auseth Nielsen

Modelling of Cache/Interconnect Performance in an Embedded System

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Snorre Aunet
Co-supervisor: Torbjørn Ness (Nordic Semiconductor ASA)
July 2021

NTNU
Norwegian University of
Science and Technology

NORDIC
SEMICONDUCTOR

Øivind Auseth Nielsen

Modelling of Cache/Interconnect Performance in an Embedded System

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Snorre Aunet
Co-supervisor: Torbjørn Ness (Nordic Semiconductor ASA)
July 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Abstract

The cache and bus topology in an embedded system has a big influence on the system performance and energy efficiency. Producing a new silicon chip for testing is expensive, and this has made simulating architectural changes a common practice in the industry. Modern RTL simulations are able to provide highly accurate estimates of integrated circuit performance. However, modelling an architecture for such a simulation is a long and laborious process, and making modifications to the model is often time consuming. There is a need for a quick and easy way of experimenting with different bus topologies, which is still able to provide a good estimate of how various changes will affect performance.

This thesis presents an easy to use model, which allows for completely changing an architecture just by modifying a few values in a text file. The model uses a node tree representation of the bus hierarchy, abstracting away hard to model architecture features, such as timing jitter when crossing between clock domains, and bus contention. Each node represent a component in the interconnect topology and explicitly states the latency it contributes to a memory access. This allows a highly simplified architecture description to be written, only focusing on the aspects of the bus topology which significantly contribute to memory access performance. The bus topology simulated for testing purposes in this thesis is the main Cortex-M33 processor on Nordic Semiconductor's nRF5340 SoC.

The simulation results were compared to tests running on a nRF5340 development kit. When simulating Coremark with cache enabled, the model reported a 18.70% longer run time than hardware. When simulating Coremark with cache disabled, the model reported a 4.36% longer run time than hardware. When simulating sequential accesses to matrices, the model reported 68% more instruction cache hits and 18% fewer latency cycles than hardware for the smallest matrices, and 50% more instruction cache hits and 5% fewer latency cycles than hardware for the largest matrices. The model displayed high fidelity, but the simulated results were offset from the hardware results. It is useful for predicting whether a change would lead to an increase or decrease in instruction cache performance and total latency cycles. There were issues outside the scope of this thesis which were significant error sources in the results. In preliminary testing of how the model would perform if these issues were resolved, it reported cache hit and cache miss results which were within 0.2% of the results seen on hardware, and cache latency results which were 7.3% higher than the results seen on hardware. This shows that the model has the potential to achieve very accurate results, and be a useful tool for initial exploration of the performance impact of modifications to cache and bus topology.

Sammendrag

Cache og bus topologi i et innvevd system har en stor innflytelse på systemets ytelse og energi effektivitet. Å produsere en ny chip for testing er dyrt, og dette har gjort simulering av arkitektur forandrnger til en vanlig praksis i industrien. Moderne RTL simuleringer kan gi veldig nøyaktige estimater av ytelsen til integrerte kretser. Å modellere en arkitektur for en slik simulering er en lang og arbeidskrevende prosess, og å modifisere modellen er ofte tidkrevende. Det er et behov for en rask og enkel måte å eksperimentere med forskjellige bus topologier, som fortsatt klarer å gi et godt estimat på hvordan forandringer vil påvirke ytelse.

Denne avhandlingen presenterer en modell som er lett å bruke, og som gjør det mulig å fullstendig forandre en arkitektur ved å bare modifisere et par verdier i en tekst fil. Modellen bruker en node-tre representasjon av bus hierarkiet, og abstraherer vekk deler av arkitekturen som er vanskelig å modellere, slik som timing jitter når man krysser klokkedomener og bus contention. Hver node representerer en komponent i interconnect topologien og sier eksplisitt hvor mye latency den legger til en minneaksess. Dette gjøre det mulig å skrive en veldig forenklet arkitektur beskrivelse, som bare fokuserer på de aspektene av bus topologien som bidrar mye til minneaksess ytelsen. Bus topologien som er simulert for testing i denne avhandlingen er den sentrale Cortex-M33 prosessoren på Nordic Semiconductor sin nRF5340 SoC.

Simuleringsresultatene ble sammenlignet med tester som kjørte på et nRF5340 development kit. Når Coremark ble simulert med cache aktivert, rapporterte modellen en kjøretid som var 18.70% lengre enn kjøretiden på hardware. Når Coremark ble simulert med cache deaktivert, rapporterte modellen en kjøretid som var 4.36% lengre enn kjøretiden på hardware. Når sekvensielle aksesser til matriser ble simulert, rapporterte modellen 68% flere instruction cache hits og 18% færre latency sykluser enn hardware for de minste matrisene, og 50% flere instruction cache hits og 5% færre latency sykluser enn hardware for de største matrisene. Modellen viste høy fidelity, men de simulerte resultatene var forskyvet sammenlignet med resultatene fra hardware. Den er nyttig for å spå om en forandring vil øke eller senke instruction cache ytelsen og den totale mengden latency sykluser. Det var problemer utenfor omfanget av denne avhandlingen som var betydelige feilkilder i resultatene. I tidlig testing av hvordan modellen ville yte hvis problemene ble løst, så rapporterte den cache hit og cache miss resultater som var innen 0.2% av resultatene som ble sett på hardware, og cache latency resultater som var 7.3% høyere enn resultatene som ble sett på hardware. Dette viser at modellen har potensialet til å gi veldig nøyaktige resultater og være et nyttig verktøy for tidlig utforskning av hvordan modifikasjoner av cache og bus topologi vil påvirke ytelse.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many aspects of an embedded system architecture contribute to the system's performance and energy efficiency. An important factor is the design and optimization of the cache and bus topology. The gains from slightly reducing the average memory access latency add up to a big performance boost, and this leads engineers to constantly search for ways to improve the bus topology. This search involves making changes and observing how they affect performance. Producing a new silicon chip every time an engineer wants to test a small change would be overly expensive, and this makes FPGA implementations and software simulations a common practice in the industry. Modern RTL software simulations can provide highly accurate estimates of integrated circuit performance. However, the models used for such simulations are very complex, and building them is a long and laborious process. Making modifications to a model is also often time consuming, and many of the changes an engineer makes are for early exploratory purposes and do not require perfectly accurate results. For these types of changes the complexity of the RTL implementation, which is necessary to achieve its high accuracy, becomes a burden. A quicker and easier way of experimenting with different bus topologies is needed, while still being able to provide good performance estimates.

This thesis shows that a very simple model can be built, which allows for quick and easy modifications by only editing values in a text file. This is achieved by only focusing on the aspects of the bus topology which significantly contribute to memory access performance, explicitly stating how much latency each component contributes to a memory access, and abstracting away hard to model features such as timing jitter and bus contention. It is also shows that such a model has the potential to retain very high accuracy, as long as it is supported by the proper tools.

Chapter 2 provides useful theory about how cache memory works, describes relevant concepts of a bus architecture, explains what Coremark is and its significance, and gives a brief introduction to the JSON text format. Chapter 3 describes the materials used, how the model functions, and details how the tests were performed. Chapter 4 presents a compari-

son between the performance of Coremark and other tests on hardware and the developed model, followed by an examination of issues encountered during testing and the impact these issues had on the results. Chapter 5 discusses the results and what they mean for the stated goals of the presented work. Chapter 6 sums up the purpose of the work done, what was achieved and the potential of future work.

Chapter 2

Theory

2.1 Cache

The cache is a memory component designed to reduce the time needed to access instructions or data from memory. It is usually very fast, small in capacity, and placed close to the processor. When the processor needs information from memory, it first checks whether it has been stored in the cache. When the processor finds the information it is looking for in the cache, it is called a cache hit. When the processor does not find the information it is looking for in the cache, it is called a cache miss. When a cache miss happens, the processor has to find the information in the larger and slower memory, which takes a much longer time. The information fetched from the slower memory is then placed in the cache, so that it can be fetched from the cache the next time it is needed[1].

The cache can be organized in different ways, but the most common is what is called an n-way set-associative cache. In this structure the cache is divided into a number of equally sized ”sets”, which are divided into a number of equally sized ”ways”, which contain a number of equally sized ”words”. A way is the size of one cache line. Each memory address is only associated with a single set in the cache, and can only be placed in its associated set. A set can only hold data from 1 address for each way it has.

As an example, imagine a 8kB 2-way set associative cache which covers the addresses for 64kB, from 0x0 to 0xFFFF. Further, imagine that the word size is 4 bytes and a cache line is 4 words. Each set would be 32 bytes, and the cache would have 256 sets. 16 bits are needed for the addresses from 0x0 to 0xFFFF. When a address is looked up in the cache, the last 5 bits, bit 12 to 16, would be called the ”offset bits”, and point to a specific byte in a set. Bit 4 to 11 would be called the ”set bits”, and decide which set the address belongs to. Bit 1 to 3 would be called the ”tag bits”, and would indicate which part of the memory the address is from. When the information from a memory address is stored in a way, the tag from that address is also stored. During a cache lookup, the cache finds the set associated with the address using its set bits, and compares its tag with the tags stored in

the set's ways. If either way has a tag that matches, it's a hit, and if they don't, it's a miss. A different tag means that the address is for a very different place in the memory, so this type of cache assumes that most accesses within a short time period will be to the same part of the memory.

2.2 Coremark

Coremark is a simple benchmark from EEMBC. It measures the performance of microcontrollers (MCUs) and central processing units (CPUs) used in embedded systems. Coremark contains list processing (find and sort), matrix manipulation (common matrix operations), state machine (determine if an input stream contains valid numbers), and CRC (cyclic redundancy check)[2]. It is one of the most commonly used benchmarks for MCUs and CPUs in embedded systems, and some companies include Coremark scores in the offical documentation of their products.

2.3 JSON

JavaScript Object Notation, or ”JSON” as it is better known, is a text format used to store and transport data. json.org describes the format in the following way:

”It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others”[3].

The JSON text format is built on two structures. The first is a collection of name/value pairs, and the second is an ordered list of values. In C++ these would be similar to a struct and an array. Here is an example of how those structures are used to store data:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Chapter 3

Methodology

3.1 Materials

3.1.1 The nRF5340 Application Core

The architecture simulated in this thesis is the Application core of the nRF5340, developed by Nordic Semiconductor ASA. The nRF5340 is an ultra-low power wireless System on Chip (SoC) with two Arm Cortex-M33 processors[4], one of these processors is named the Application core and the other processor is named the Network core. The Application core is designed to be the main processor, handling the main functions of the application, while the connectivity back-end, for example the bluetooth stack, is running on the Network core. This architecture was chosen because it is the newest SoC from Nordic Semiconductor, making it their most relevant SoC to study in this thesis. The hardware tests were run on the Application core of a nRF5340 development kit PCA10095 v0.7.0, as seen in figure 3.1.

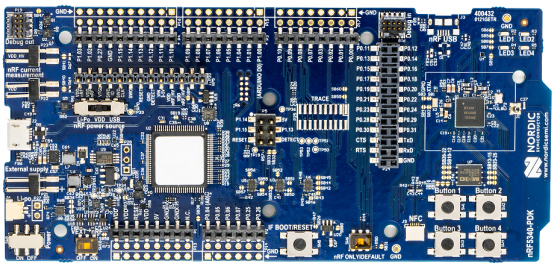

Figure 3.1: nRF5340 development kit

3.1.2 Cache Model

A pre-existing cache model, created by Nordic Semiconductor, was used to simulate cache functionality. The cache model was ported from python to C++, and the code for the cache model is attached in appendix listing 6.8. It is implemented as a class object, with ways, sets, linewidth, duwidth and lookahead as input arguments for its constructor function. These parameters are then used to initialize vector variables of the appropriate dimensions for the sets, ways, tag, data, valid bit and mru bit of a cache with matching parameters.

A lookup function is used to simulate instruction and data fetches. This function takes a memory address as an argument, and determines whether a fetch for that address should result in a cache hit or a cache miss by calculating its tag and associated set, and checking whether a way in that set is valid and populated with a matching tag. The cache content is also updated, using a least-recently-used replacement policy.

3.2 Modelling the Architecture

The model was implemented in C++, and uses an existing cache model which was ported from python to C++. The code for the model is attached in appendix listing 6.7. A recursive class called "memory_map" was created to build a tree hierarchy representing the architecture, with the CPU as the root node as shown in figure 3.2. A node represents a part of the bus hierarchy which can be traversed during a memory access, becoming more localized for each step, and with the leaf nodes representing the memory itself. This structure includes every detail relevant for the performance metrics being evaluated, and aims to make adding or removing components very easy. Every component is represented by a node with the same class skeleton, and all that is needed to represent a new component with a node are a few basic parameters.

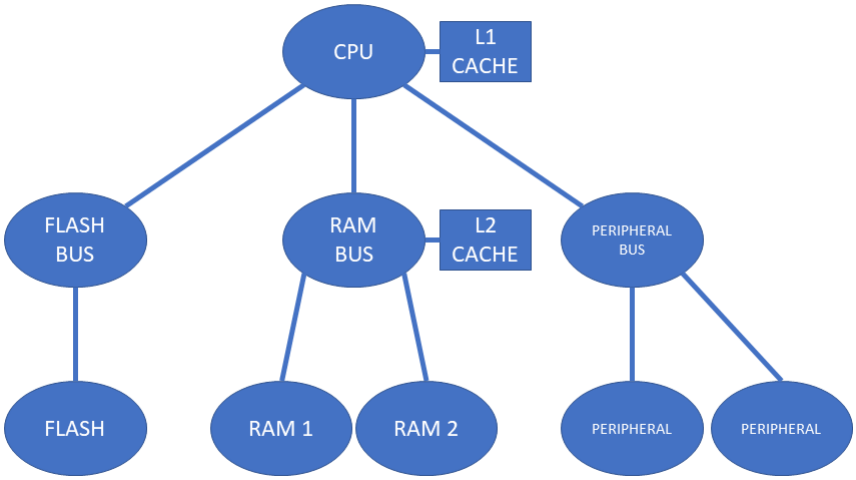

Figure 3.2: Memory Map

3.2.1 The Contents of a Node

Every node contains information about the section of the architecture it represents, such as its name, the frequency domain, and the additional latency cycles traversing it adds to a memory access, as shown in figure 3.3. Each parent node also contains one struct for each of its children, organized in a vector. These structs contain information about the child nodes, such as the memory region they represent, whether the memory region is cacheable, and a pointer to the class object representing the child node. Having this information available in the parent node reduces the amount of class objects which need to be accessed when traversing the node tree. If this information was stored in the child nodes, a function would potentially be required to access every child node when traversing through a parent node. A node can also hold a pointer to a cache object, based on

the pre-existing cache model. This pointer will be present in the CPU node for most architectures, and some architectures could also have lower level caches in lower nodes. This implementation models the delay caused by the distance from the CPU to lower level caches, and how architecture changes affect this delay. The node also has three functions: get_latency(), which traverses the node tree to calculate the latency of a memory access; freq_convert(), which handles the conversion from one frequency domain to another; and check_contention(), which handles the delay experienced when accessing a contended part of the architecture.

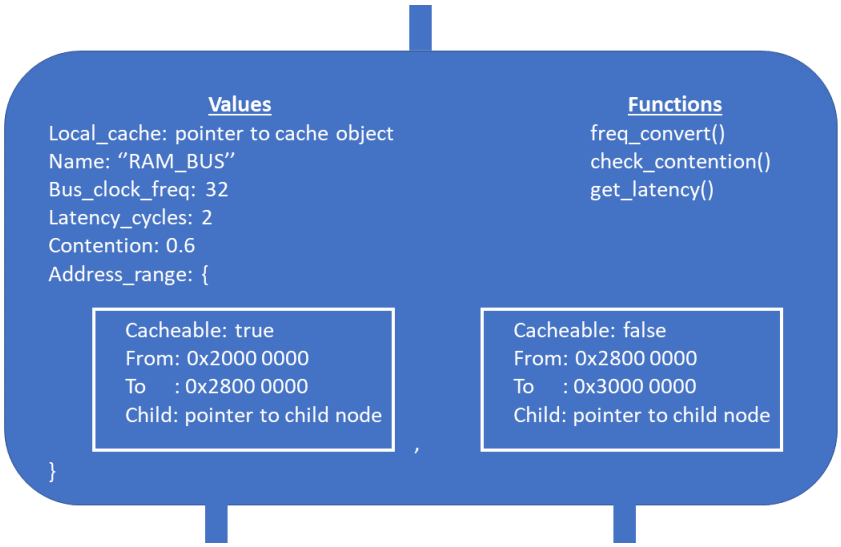

Figure 3.3: Node Contents

3.2.2 Building the Node Tree

The constructor of the memory map class is the function which builds the node tree, using the the JSON object it takes as an argument. The function traverses the JSON hierarchy and uses the contents to populate its own tree of recursive class objects. The JSON object is created from a description in a text file, and changes to the architecture can be made by editing this file with basic text editing software. The description in the text file is a node tree, showing how the node tree built by the constructor should be laid out.

3.3 Simulating a Memory Access

A class function is used to find the latency of a memory access in the architecture, measured in CPU clock cycles. This is a recursive function which takes a memory address as an argument and returns the latency of a memory access to that address. The function first looks through the start and end addresses of the child nodes and finds the child associated with the passed memory address. If there is a local cache and the child node is cacheable, a cache lookup is performed. On a hit, the cache latency is returned, and on a miss the function is recursively called in the child node. Through this process the function traverses the node tree until it gets a cache hit or enters a leaf node, as shown in the function float chart in figure 3.4. A leaf node has no child nodes, and in this situation the node's latency parameter is returned. As the function travels back up the node tree, the latency parameter of each node is added to the total latency value which is returned by the initial instance of the function.

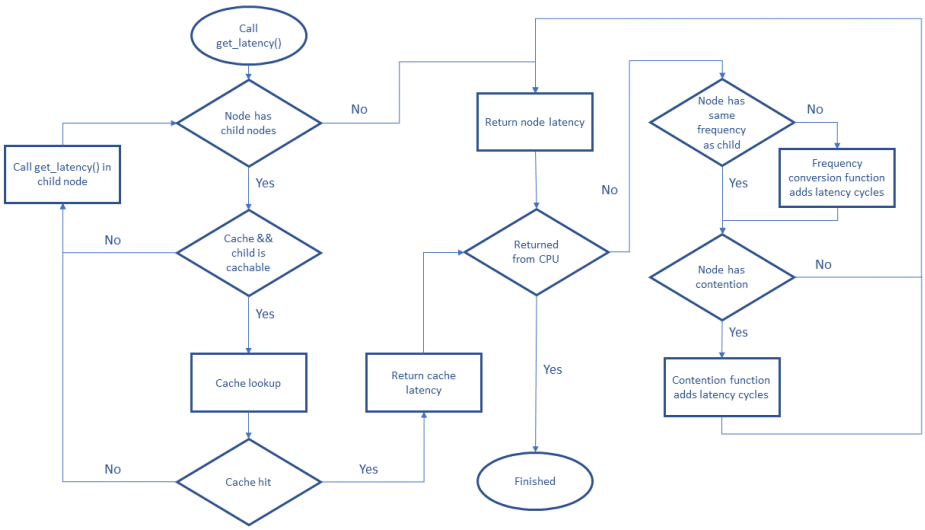

Figure 3.4: get_latency() functionality

3.3.1 Bus Contention

This is a very simplified implementation of a memory access, and some aspects of bus and interconnect architectures had to be abstracted away. The first of these is the situation where a component in the path of a memory access is being accessed by a master other than the CPU, and thus the CPU has to wait for the component to become available before the memory access can be carried out. This is called bus contention and has been abstracted away as a contention parameter for each node, which signifies the probability that the node will be busy during a given clock cycle. A master in this context is any component which controls or makes accesses to other components. Before the latency function in a

node returns, it passes the node's contention value to a function which generates a random amount of additional latency cycles based on the contention parameter.

3.3.2 Frequency Conversion

The concept of aligning different frequency domains has also been abstracted away. This has been replaced by a frequency parameter and a conversion function which is called when a recursively called latency function returns from the child node. The conversion function takes the latency value returned by the latency function and the frequency of the two nodes as parameters, and uses these to generate a random amount of additional latency cycles. This simulates the amount of cycles the CPU has to wait for clocks of the two frequency domains to line up, and that a number of latency cycles in a given frequency domain constitutes more cycles when they are converted to a higher frequency.

3.4 The Application

When the application is launched, the selected JSON text file is read into a JSON object. The JSON object is passed as an argument to the constructor of a memory_map class object. This results in a populated node tree representing the architecture described in the text file. The application reads memory addresses from another text file, and simulates memory accesses to those addresses. This can be used to simulate the memory accesses of entire programs. The application also simulates the results of cache lookups and calculates the delay incurred by each memory access. At the end of the simulation, the destructor function of the class objects prints the performance numbers of every implemented cache object.

3.5 Testing Methodology

3.5.1 Obtaining Input Data

The memory accesses used in the simulation were obtained by running tests on the nRF5340 development kit and logging the memory accesses using GDB[5]. The board was connected to a Windows 10 computer over J-Link, using a USB cable. The Windows 10 application J-Link GDB Server V6.88a[6] was used to connect to the board's debugger chip, and the GDB debugging application was connected to the GDB server's listening port. With this setup, GDB could issue commands to the debugger, and was able to step through the firmware instructions in a controlled manner. At every step the program counter was logged and the instruction word sent through a python decoder function. The python decoder code, attached in appendix listing 6.11, was provided by Nordic Semiconductor and is separate from the work carried out in this thesis. The decoder function determines whether the instruction was a data access. If a data access was identified, the address of the accessed data was also logged. The testing setup is shown in figure 3.5 and figure 3.6.

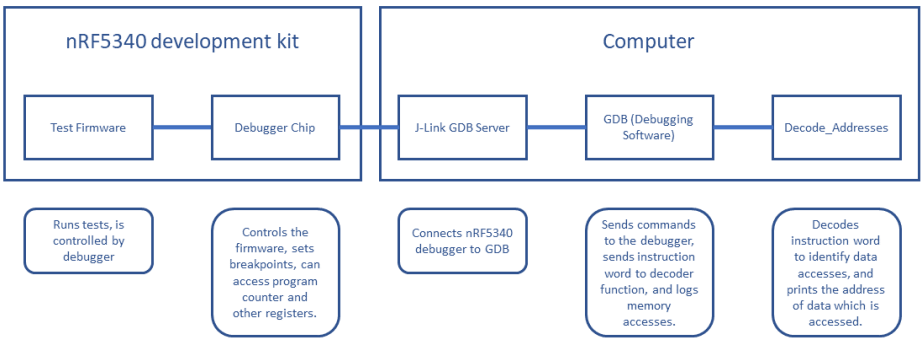

Figure 3.5: Testing Setup Graphic

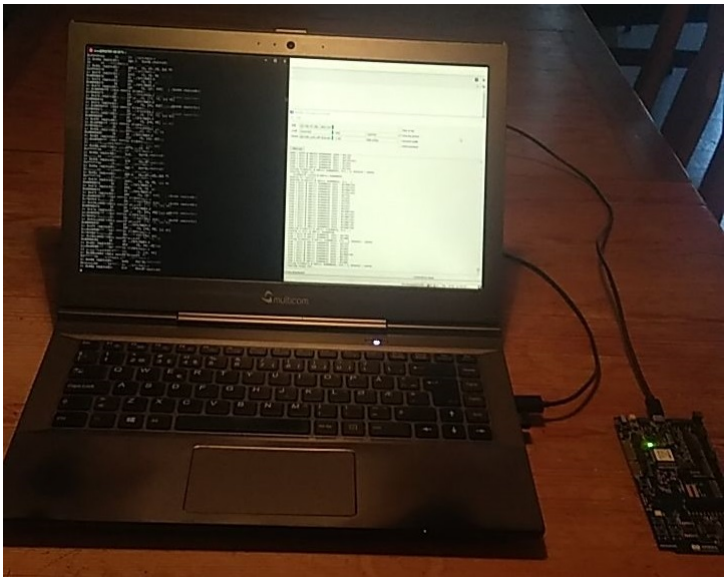

Figure 3.6: Testing Setup

3.5.2 nRF5340 Application Core JSON File

The JSON text file describing the nRF5340 Application core was written using the information available in the product specification at Nordic Infocenter[7], and is attached in appendix listing 6.9. Latency values for flash, RAM and buses were estimated based on this documentation. The documentation is clear about the layout of the cache, and this information was used for the cache in the ”CPU_BUS” node. Rough estimates were used for memory regions outside flash and RAM, as they were deemed to not be significant for the specific tests performed in this thesis.

3.5.3 Tests

The first test was Coremark, with and without cache enabled to verify that the cache model was behaving correctly. With a correctly modelled cache, the simulation and hardware should see the same relative performance changes when switching between the cache being enabled and the cache being disabled. Both scenarios ran two iterations of the Coremark benchmarks. This allowed the first iteration to fill the cache and the second iteration to run with a pre-filled cache when the cache was enabled. The core_portme.c and core_portme.h files of the Coremark code had to be configured for the nRF5340, and are attached in appendix listing 6.5 and 6.6, respectively. The Coremark code was compiled into .hex files using Make and flashed onto the nRF5340 development kit using Segger Embedded Studio for ARM (Nordic Edition) V5.10d (64-bit)[8]. The hardware results were reported by the benchmark code and read using a serial terminal on a computer connected to the board

using a USB cable.

The second set of tests were sequential accesses to the elements of matrices of various sizes. The various sizes were added to compare how the model performed when the entire data structure could fit in the cache, and how it performed when the data structure was larger than the cache. The sizes used were 10x10, 25x25, 50x50 and 100x100, and an example of the files used to store the matrices is attached in appendix listing 6.4. One matrix was stored in flash memory, while the other was stored in RAM, in order to test both flash memory accesses and RAM accesses of large data structures. The test was developed using the nRF Connect SDK v1.4.2[9], and the code is attached in appendix listing 6.1. Additional files used to compile this test are attached in appendix listing 6.2 and 6.3. The code was compiled into .hex files and flashed onto the nRF5340 development kit using Segger Embedded Studio.

Chapter 4

Results

4.1 Coremark

Tables 4.1 and 4.2 show the Coremark results on both the model and the nRF5340 application core, with and without cache. While running Coremark, the application core was running at a clock frequency of 128 MHz. For convenience, the run time reported by Coremark has been converted to latency cycles, and the latency cycles reported by the model have been converted to run time.

While Coremark outputs a lot of performance metrics, the only metric which can be properly compared with the output from the model is the run time. With the cache enabled, the Coremark simulation reports a run time which is 18.70% longer than the run time reported by Coremark running on hardware. This discrepancy is reduced when the cache is disabled, with the simulated run time being 4.36% longer than the run time on hardware.

Cache Status	Cache Hits	Cache Misses	Latency Cycles	Run Time
Enabled	462850	6935	585794	0.004577s
Disabled	–	–	1511493	0.011809s

Table 4.1: Simulation Results, Coremark

Cache Status	Cache Hits	Cache Misses	Latency Cycles	Run Time
Enabled	–	–	493568	0.003856s
Disabled	–	–	1448448	0.011316s

Table 4.2: Hardware Results, Coremark

4.2 Sequential Matrix Accesses

Tables 4.3, 4.4 and 4.5 show the results of sequential matrix accesses on both the model and the nRF5340 application core. The amount of simulated cache misses stayed consistent for all matrix sizes, similar to the instruction misses seen on hardware. For ease of viewing, the instruction hits and misses seen on hardware have been isolated in table 4.5. Table 4.6 shows that the ratio between the simulated cache hits and hardware instruction cache hits converges to 1.50 as the size of the matrices increases. Similarly the ratio between the simulated and hardware latency cycles converges to about 0.95, or more accurately $17/18$.

Matrix Dimensions	Cache Hits	Cache Misses	Latency Cycles	Cycles per Operation
10 x 10	1649	14	1891	18.91
25 x 25	9538	15	10833	17.33
50 x 50	37789	14	42831	17.13
100 x 100	150589	14	170631	17.06

Table 4.3: Simulation Results, Sequential Matrix Accesses

Matrix Dimensions	Cache Hits (Instruction+Data)	Cache Misses (Instruction+Data)	Latency Cycles	Cycles per Operation
10 x 10	1247 (979+268)	19 (5+14)	2302	23.02
25 x 25	8100 (6321+1779)	85 (5+80)	12408	19.85
50 x 50	32362 (25192+7170)	319 (5+314)	45482	18.19
100 x 100	129175 (100442+28733)	1256 (5+1251)	180015	18.00

Table 4.4: Hardware Results, Sequential Matrix Accesses

Matrix Dimensions	Cache Hits	Cache Misses	Latency Cycles	Cycles per Operation
10 x 10	979	5	2302	23.02
25 x 25	6321	5	12408	19.85
50 x 50	25192	5	45482	18.19
100 x 100	100442	5	180015	18.00

Table 4.5: Hardware Results, Instruction Hits and Misses, Sequential Matrix Accesses

Matrix Dimensions	Simulated/Hardware Cache Hits Ratio	Simulated/Hardware Latency Cycle Ratio
10 x 10	1.68	0.82
25 x 25	1.51	0.87
50 x 50	1.50	0.94
100 x 100	1.50	0.95

Table 4.6: Simulated/Hardware Ratios, Sequential Matrix Accesses

4.3 Observed Issues

4.3.1 Missing Data Accesses

Through inspecting the logged memory accesses, it was discovered that the data accesses for the specific elements in the matrices were not being logged. An attempt was made to estimate which addresses would have been accessed by the missing data accesses, and add them to the log for the 100 by 100 sequential matrix accesses. One memory access was added after each ldr.w instruction, which is how the missing data accesses should have appeared. As the elements accessed were sequentially stored integer values, the memory addresses were incremented by 4 for each iteration. The starting address was set at 0x4e38, the location in memory where the accessed matrix was stored.

4.3.2 Wrong Tag Length

It was also discovered that the cache on the hardware uses a 17-bit tag, while inputting the parameters of the hardware cache into the cache model produces a 20-bit tag. This occurs because only the memory region from 0x0 to 0x1FFFFFFF is cacheable on the hardware, while the cache model assumes the cache address space to be from 0x0 to 0xFFFFFFFF. To observe how a smaller tag would affect results, a 17-bit tag was forced by setting the duwidth to 32 bytes, 8 times its actual size.

4.3.3 Too Many Instruction Cache Lookups

The third uncovered issue was related to the situation where two 16-bit instructions are stored consecutively in the memory, within the same 32-bit aligned word, and these instructions are consecutively executed. When the first instruction is fetched, either from the cache or memory, the nRF5340's CPU also fetches the second instruction, as it always fetches 32-bit sections. Then it skips the cache lookup for the second instruction, as it has already been fetched. The model does not take this into account, and as a result the simulation performs significantly more cache lookups compared to the board. In order to explore the impact of this oversight, a quick fix was implemented by keeping track of the most recently fetched cacheable address. The fix also took advantage of a quirk of the decoder function, which causes data accesses to always be padded to 32-bit length, to differentiate instruction and data addresses. This fix made it possible to simulate the ”double-fetching” behavior of the nRF5340.

4.4 Improvements

A simple script was written to add missing data accesses to the list of memory accesses for the 100 by 100 sequential matrix accesses, the code for this script is attached in appendix listing 6.13. The duwidth for the CPU cache was also increased to 32 bytes. Lastly, two different consecutive instructions in the same 32-bit word were made to only cause a single cache lookup. The results from running the simulation with these modifications are shown in table 4.7, and how these results compare with the hardware results is shown in table 4.8.

Modifications	Cache Hits	Cache Misses	Latency Cycles	Cycles per Operation
Added data accesses	160589	10014	210631	21.06
Increased duwidth	150600	3	170609	17.06
Added data accesses & increased duwidth	169349	1254	193111	19.31
Data & duwidth & double-fetching instructions	129050	1254	193111	19.31

Table 4.7: Simulation Results, 100x100 Sequential Matrix Accesses, modified

Modifications	Cache Hits Ratio	Cache Misses Ratio	Latency Cycles Ratio
Added data accesses	1.243	7.973	1.170
Increased duwidth	1.166	0.002	0.948
Added data accesses & increased duwidth	1.311	0.998	1.073
Data & duwidth & double-fetching instructions	0.999	0.998	1.073

Table 4.8: Potential Simulated/Hardware Ratios, 100x100 Sequential Matrix Accesses, modified

Chapter 5

Discussion

5.1 Coremark

The higher accuracy when the cache is disabled points to the cache being a significant error source in the model. The issues outlined in chapter 4.3 are likely the cause of the poor performance of the model when the cache is enabled. When the cache is disabled the model outputs results which are very close to the results seen on the nRF5340 development kit. A discrepancy of 4.36% should be acceptable for exploring how changes to the bus topology would affect performance, if the discrepancy is consistent and can be accounted for.

5.2 Sequential Matrix Accesses

The point of this test was to force cache misses, specifically data fetch misses, by accessing matrices of various sizes. This was successfully executed on the nRF5340 development kit, as seen in table 4.4, but the data in table 4.3 reveals that the simulation could not reproduce the cache misses. If we ignore the data cache hits and misses in the hardware results, and only look at the instruction cache, shown in table 4.5, it appears that the instruction cache numbers are a lot closer to the simulated numbers. This could mean that the model is unsuited for simulating programs which handle data in cacheable memory regions. Tables 4.7 and 4.8 show that this is likely caused by the issues described in chapter 4.3, and that if these issues are solved, the model has the potential to become very accurate.

5.3 Improvements

As explained in chapter 4.3.1, the method used to acquire input data for the model was not able to log every memory access. Specifically, the method is not able to correctly identify data accesses originating from 32-bit instructions like ldr.w. Future work should find a new method of acquiring input data, or improve the existing method such that it can properly log 32-bit instructions.

Chapter 4.3.2 describes how the cache model was not able to accurately simulate the cache parameters seen on the nRF5340 Application core. A high priority improvement would be to replace the cache model, or rewrite it to allow for deeper customization of the cache parameters.

The model is not able to reproduce how the cache on the simulated hardware fetches instructions, and as a result it reports inaccurate cache performance numbers. A robust solution to this problem will require sections of the model code to be rewritten, but will likely produce much more accurate results for all affected architectures.

Temporary implementations of fixes to these problems can be seen in table 4.7, and together the fixes appear to produce performance numbers which are very close to the hardware performance seen in table 4.4. Solving these issues would produce a much more robust and accurate model.

Chapter 6

Conclusion

This thesis shows that a simple bus topology model can be a quick and easy way of experimenting with different bus topologies, while still being able to provide a good estimate of how various changes will affect performance. In this thesis the model was held back by an insufficient cache model and inaccurate input data, so future work should pair the model with suitable supporting tools to fully realize its potential. When supported by proper tools, the model appears to be able to simulate cache hits and cache misses to within 0.2%, and cache latency to an error margin as low as 7.3%.

Bibliography

[1] John L. Hennessy and David A. Patterson. Computer Architecture A Quantitative Approach (5th edition). Morgan Kaufmann Publishers, Appendix B.1

[2] Shay Gal-On and Markus Levy (2009). Exploring CoreMark™ – A Benchmark Maximizing Simplicity and Efficacy. EEMBC

[3] Douglas Crockford (2021). Introducing JSON. Retrieved July 16, 2021, from https://www.json.org/json-en.html

[4] Nordic Semiconductor ASA (2021). Nordic Semiconductor Infocenter. Retrieved July 16, 2021, from https://infocenter.nordicsemi.com/topic/struct_nrf53/struct/nrf5340.html

[5] GNU (2021). GDB: The GNU Project Debugger. Retrieved July 16, 2021, from https://www.gnu.org/software/gdb/

[6] SEGGER (2021). J-Link GDB Server. Retrieved July 16, 2021, from https://www.segger.com/products/debug-probes/j-link/tools/j-link-gdb-server/about-j-link-gdb-server/

[7] Nordic Semiconductor ASA (2021). Nordic Semiconductor Infocenter. Retrieved July 16, 2021, from https://infocenter.nordicsemi.com/index.jsp

[8] Nordic Semiconductor (2021). SEGGER Embedded Studio - nordicsemi.com. Retrieved July 16, 2021, from https://www.nordicsemi.com/Products/Development-tools/Segger-Embedded-Studio

[9] Nordic Semiconductor (2021). Welcome to the nRF Connect SDK! – nRF Connect SDK 1.4.2 documentation. Retrieved July 16, 2021, from https://developer.nordicsemi.com/nRF_Connect_SDK/doc/1.4.2/nrf/index.html

Appendix

Listing 6.1: Sequential Matrix Accesses Code

```
#include <zephyr.h>
#include <device.h>
#include <devicetree.h>
#include <drivers/gpio.h>
#include <nrf.h>
#include <timing/timing.h>

#include "matrices100.h"

void main(void)
{
        timing_init();

        NRF_CACHE->ENABLE=0;
        NRF_CACHE->PROFILINGENABLE=0;

        volatile int tmp = 0;
        int x = 0;
        int y;

        timing_start();
        while (x < mdim) {
                y = 0;
                while (y < mdim) {
                        tmp += (mram[x][y] * mflash[x][y]);
                        y++;
                }
                x++;
        }
        timing_stop();
}
```

Listing 6.2: Sequential Matrix Accesses config file

```
CONFIG_GPIO=y
CONFIG_DEBUG=y
CONFIG_CONSOLE=y
CONFIG_UART_CONSOLE=y
CONFIG_TIMING_FUNCTIONS=y
```

Listing 6.3: Sequential Matrix Accesses CMakeLists.txt

```
cmake_minimum_required(VERSION 3.13.1)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(blinky)

target_sources(app PRIVATE src/main.c)
zephyr_library_include_directories(
  include
)
```

Listing 6.4: 10x10 Matrices header file

```
#define mdim 10

volatile int mram[mdim][mdim] = {
{74, 576, 271, 107, 488, 313, 708, 505, 583, 812},
{878, 714, 232, 301, 578, 992, 294, 280, 491, 219},
{689, 973, 532, 373, 18, 315, 93, 295, 691, 303},
{175, 349, 34, 569, 38, 889, 369, 418, 373, 457},
{423, 521, 248, 985, 347, 91, 150, 163, 756, 439},
{580, 187, 592, 430, 7, 289, 567, 47, 473, 809},
{995, 350, 639, 24, 81, 16, 983, 199, 451, 472},
{628, 140, 815, 665, 111, 411, 651, 912, 563, 830},
{746, 425, 852, 785, 410, 499, 602, 665, 960, 547},
{888, 594, 763, 223, 797, 999, 115, 120, 833, 989}
};

const int mflash[mdim][mdim] = {
{838, 456, 845, 467, 394, 647, 765, 61, 918, 473},
{681, 407, 969, 985, 482, 236, 223, 77, 661, 595},
{72, 50, 392, 344, 239, 502, 175, 496, 564, 117},
{808, 796, 989, 427, 629, 261, 963, 174, 270, 459},
{605, 615, 652, 504, 30, 641, 872, 857, 521, 103},
{27, 597, 572, 298, 434, 667, 532, 842, 682, 31},
{627, 263, 950, 725, 939, 940, 4, 751, 103, 681},
{52, 618, 778, 63, 501, 747, 449, 914, 916, 905},
{513, 606, 30, 225, 937, 515, 625, 717, 510, 826},
{261, 275, 659, 864, 21, 957, 935, 99, 687, 485}
};
```

Listing 6.5: Coremark C file

```
/*
        File : core_portme.c
*/
/*
        Author : Shay Gal-On, EEMBC
        Legal : TODO!
*/
#include "coremark.h"
#include "core_portme.h"

#if VALIDATION_RUN
        volatile ee_s32 seed1_volatile=0x3415;
        volatile ee_s32 seed2_volatile=0x3415;
        volatile ee_s32 seed3_volatile=0x66;
#endif
#if PERFORMANCE_RUN
        volatile ee_s32 seed1_volatile=0x0;
        volatile ee_s32 seed2_volatile=0x0;
        volatile ee_s32 seed3_volatile=0x66;
#endif
#if PROFILE_RUN
        volatile ee_s32 seed1_volatile=0x8;
        volatile ee_s32 seed2_volatile=0x8;
        volatile ee_s32 seed3_volatile=0x8;
#endif
        volatile ee_s32 seed4_volatile=ITERATIONS;
        volatile ee_s32 seed5_volatile=0;
/* Porting : Timing functions
        How to capture time and convert to seconds must be ported
            to whatever is supported by the platform.
        e.g. Read value from on board RTC, read value from cpu
            clock cycles performance counter etc.
        Sample implementation for standard time.h and windows.h
            definitions included.
*/
CORETIMETYPE barebones_clock() {
        TIMER->TASKS_CAPTURE[0] = 1;
        return TIMER->CC[0];
}
/* Define : TIMER_RES_DIVIDER
        Divider to trade off timer resolution and total time that
            can be measured.

        Use lower values to increase resolution, but make sure
            that overflow does not occur.
        If there are issues with the return value overflowing,
            increase this value.
        */
```

```
#define GETMYTIME(_t) (*_t=barebones_clock())
#define MYTIMEDIFF(fin,ini) ((fin)-(ini))
#define TIMER_RES_DIVIDER 1
#define SAMPLE_TIME_IMPLEMENTATION 1
// 250kHz
#define CLOCKS_PER_SEC    (250000)
#define EE_TICKS_PER_SEC (CLOCKS_PER_SEC / TIMER_RES_DIVIDER)

/** Define Host specific (POSIX), or target specific global time
    variables. */
static CORETIMETYPE start_time_val, stop_time_val;

/* Function : start_time
        This function will be called right before starting the
            timed portion of the benchmark.

        Implementation may be capturing a system timer (as
            implemented in the example code)
        or zeroing some system parameters - e.g. setting the cpu
            clocks cycles to 0.
*/
void start_time(void) {
        GETMYTIME(&start_time_val );
}
/* Function : stop_time
        This function will be called right after ending the timed
            portion of the benchmark.

        Implementation may be capturing a system timer (as
            implemented in the example code)
        or other system parameters - e.g. reading the current
            value of cpu cycles counter.
*/
void stop_time(void) {
        GETMYTIME(&stop_time_val );
}
/* Function : get_time
        Return an abstract "ticks" number that signifies time on
            the system.

        Actual value returned may be cpu cycles, milliseconds or
            any other value,
        as long as it can be converted to seconds by <time_in_secs
            >.
        This methodology is taken to accomodate any hardware or
            simulated platform.
        The sample implementation returns millisecs by default,
        and the resolution is controlled by <TIMER_RES_DIVIDER>
*/
CORE_TICKS get_time(void) {
```

```
        CORE_TICKS elapsed=(CORE_TICKS)(MYTIMEDIFF(stop_time_val,
            start_time_val));
        return elapsed;
}
/* Function : time_in_secs
        Convert the value returned by get_time to seconds.

        The <secs_ret> type is used to accomodate systems with no
            support for floating point.
        Default implementation implemented by the EE_TICKS_PER_SEC
             macro above.
*/
secs_ret time_in_secs(CORE_TICKS ticks) {
        secs_ret retval=((secs_ret)ticks) / (secs_ret)
            EE_TICKS_PER_SEC;
        return retval;
}

ee_u32 default_num_contexts=1;

void uart_init(void) {
        UART_GPIO_PORT->DIRSET = (1 << UART_TX_PIN);
        UART_GPIO_PORT->OUTSET = (1 << UART_TX_PIN);
        UART_GPIO_PORT->PIN_CNF[UART_RX_PIN] = (
                        (GPIO_PIN_CNF_PULL_Pullup <<
                            GPIO_PIN_CNF_PULL_Pos)
                        | (GPIO_PIN_CNF_DIR_Input <<
                            GPIO_PIN_CNF_DIR_Pos)
                        | (GPIO_PIN_CNF_INPUT_Connect <<
                            GPIO_PIN_CNF_INPUT_Pos)
                );

        UART->BAUDRATE = (UARTE_BAUDRATE_BAUDRATE_Baud115200 <<
            UARTE_BAUDRATE_BAUDRATE_Pos);
        UART->PSEL.RTS = UART_RTS_PIN;
        UART->PSEL.TXD = UART_TX_PIN;
        UART->PSEL.RXD = UART_RX_PIN;
        UART->PSEL.CTS = UART_CTS_PIN;
        UART->CONFIG   = (UARTE_CONFIG_HWFC_Enabled <<
            UARTE_CONFIG_HWFC_Pos);
}

/* Function : portable_init
        Target specific initialization code
        Test for some common mistakes.
*/
void portable_init(core_portable *p, int *argc, char *argv[])
{
#ifndef NO_CACHE
        NRF_CACHE_S->ENABLE = 1;
```

```
#endif
        // UART init
        uart_init();
        ee_printf("portable_init()\n");
        if (sizeof(ee_ptr_int) != sizeof(ee_u8 *)) {
                ee_printf("ERROR! Please define ee_ptr_int to a
                    type that holds a pointer!\n");
        }
        if (sizeof(ee_u32) != 4) {
                ee_printf("ERROR! Please define ee_u32 to a 32b
                    unsigned type!\n");
        }
        p->portable_id=1;

        TIMER->BITMODE = TIMER_BITMODE_BITMODE_32Bit;
        TIMER->PRESCALER = 6; // 0.25MHz
        TIMER->TASKS_START = 1;
}
/* Function : portable_fini
        Target specific final code
*/
void portable_fini(core_portable *p)
{
        p->portable_id=0;
        TIMER->TASKS_STOP = 1;
        while(1) {} // Spin-wait upon completion, this is the last
             executed line in the test
}

// Keil printf support
int stdout_putchar(int ch) {
        UART->ENABLE = UARTE_ENABLE_ENABLE_Enabled <<
            UARTE_ENABLE_ENABLE_Pos;
        UART->TXD.PTR = (uint32_t)&ch;
        UART->TXD.MAXCNT = 1;
        UART->TASKS_STARTTX = 1;
        while (!UART->EVENTS_ENDTX); // Wait for completion
        // Ack ENDTX event
        UART->EVENTS_ENDTX = 0;
        UART->TASKS_STOPTX = 1;
        while (!UART->EVENTS_TXSTOPPED);
        UART->EVENTS_TXSTOPPED = 0;
        UART->ENABLE = 0;
        return ch;
}
```

Listing 6.6: Coremark header file

```
/* File : core_portme.h */

#include <nrf.h>
#include <stdint.h>

/*
        Author : Shay Gal-On, EEMBC
        Legal : TODO!
*/
/* Topic : Description
        This file contains configuration constants required to
            execute on different platforms
*/
#ifndef CORE_PORTME_H
#define CORE_PORTME_H
/*************************/
/* Data types and settings */
/*************************/
/* Configuration : HAS_FLOAT
        Define to 1 if the platform supports floating point.
*/
#ifndef HAS_FLOAT
#define HAS_FLOAT 1
#endif
/* Configuration : HAS_TIME_H
        Define to 1 if platform has the time.h header file,
        and implementation of functions thereof.
*/
#ifndef HAS_TIME_H
#define HAS_TIME_H 0
#endif
/* Configuration : USE_CLOCK
        Define to 1 if platform has the time.h header file,
        and implementation of functions thereof.
*/
#ifndef USE_CLOCK
#define USE_CLOCK 0
#endif
/* Configuration : HAS_STDIO
        Define to 1 if the platform has stdio.h.
*/
#ifndef HAS_STDIO
#define HAS_STDIO 1
#include <stdio.h>
#endif
/* Configuration : HAS_PRINTF
        Define to 1 if the platform has stdio.h and implements the
            printf function.
```

```
*/
#define HAS_PRINTF 0
#ifndef HAS_PRINTF
#define HAS_PRINTF 1
#endif

/* Definitions : COMPILER_VERSION, COMPILER_FLAGS, MEM_LOCATION
        Initialize these strings per platform
*/
#ifndef COMPILER_VERSION
 #ifdef __GNUC__
 #define COMPILER_VERSION "GCC"__VERSION__
 #else
 #define COMPILER_VERSION "Please put compiler version here (e.g. gcc 4.1)"
 #endif
#endif
#ifndef COMPILER_FLAGS
 #define COMPILER_FLAGS "-o3" /* "Please put compiler flags here (e.g. -o3)" */
#endif
#ifndef MEM_LOCATION
 #define MEM_LOCATION "STACK"
#endif

/* Data Types :
        To avoid compiler issues, define the data types that need
            ot be used for 8b, 16b and 32b in <core_portme.h>.

        *Imprtant* :
        ee_ptr_int needs to be the data type used to hold pointers
            , otherwise coremark may fail!!!
*/
typedef int16_t ee_s16;
typedef uint16_t ee_u16;
typedef int32_t ee_s32;
typedef double ee_f32;
typedef uint8_t ee_u8;
typedef uint32_t ee_u32;
typedef ee_u32 ee_ptr_int;
typedef size_t ee_size_t;

#ifndef NULL
#define NULL 0
#endif
/* align_mem :
        This macro is used to align an offset to point to a 32b
            value. It is used in the Matrix algorithm to
            initialize the input memory blocks.
*/
```

```
#define align_mem(x) (void *)(4 + (((ee_ptr_int)(x) - 1) & ~3))

/* Configuration : CORE_TICKS
        Define type of return from the timing functions.
 */
#define CORETIMETYPE ee_u32
typedef ee_u32 CORE_TICKS;

/* Configuration : SEED_METHOD
        Defines method to get seed values that cannot be computed
            at compile time.

        Valid values :
        SEED_ARG - from command line.
        SEED_FUNC - from a system function.
        SEED_VOLATILE - from volatile variables.
*/
#ifndef SEED_METHOD
#define SEED_METHOD SEED_VOLATILE
#endif

/* Configuration : MEM_METHOD
        Defines method to get a block of memry.

        Valid values :
        MEM_MALLOC - for platforms that implement malloc and have
            malloc.h.
        MEM_STATIC - to use a static memory array.
        MEM_STACK - to allocate the data block on the stack (NYI).
*/
#ifndef MEM_METHOD
#define MEM_METHOD MEM_STACK
#endif

/* Configuration : MULTITHREAD
        Define for parallel execution

        Valid values :
        1 - only one context (default).
        N>1 - will execute N copies in parallel.

        Note :
        If this flag is defined to more then 1, an implementation
            for launching parallel contexts must be defined.

        Two sample implementations are provided. Use <USE_PTHREAD>
             or <USE_FORK> to enable them.

        It is valid to have a different implementation of <
            core_start_parallel> and <core_end_parallel> in <
```

```
            core_portme.c>,
        to fit a particular architecture.
*/
#ifndef MULTITHREAD
#define MULTITHREAD 1
#define USE_PTHREAD 0
#define USE_FORK 0
#define USE_SOCKET 0
#endif

/* Configuration : MAIN_HAS_NOARGC
        Needed if platform does not support getting arguments to
            main.

        Valid values :
        0 - argc/argv to main is supported
        1 - argc/argv to main is not supported

        Note :
        This flag only matters if MULTITHREAD has been defined to
            a value greater then 1.
*/
#ifndef MAIN_HAS_NOARGC
#define MAIN_HAS_NOARGC 1
#endif

/* Configuration : MAIN_HAS_NORETURN
        Needed if platform does not support returning a value from
             main.

        Valid values :
        0 - main returns an int, and return value will be 0.
        1 - platform does not support returning a value from main
*/
#ifndef MAIN_HAS_NORETURN
#define MAIN_HAS_NORETURN 0
#endif

/* Variable : default_num_contexts
        Not used for this simple port, must cintain the value 1.
*/
extern ee_u32 default_num_contexts;

typedef struct CORE_PORTABLE_S {
        ee_u8    portable_id;
} core_portable;

/* target specific init/fini */
void portable_init(core_portable *p, int *argc, char *argv[]);
void portable_fini(core_portable *p);
```

```
#if !defined(PROFILE_RUN) && !defined(PERFORMANCE_RUN) && !defined
    (VALIDATION_RUN)
#if (TOTAL_DATA_SIZE==1200)
#define PROFILE_RUN 1
#elif (TOTAL_DATA_SIZE==2000)
#define PERFORMANCE_RUN 1
#else
#define VALIDATION_RUN 1
#endif
#endif

// Keil printf support
int stdout_putchar(int ch);
void uart_init(void);

#define TIMER NRF_TIMER0_S
#define GPIO_P0 NRF_P0_S

// nRF5340-DK pin config
#define UART              NRF_UARTE0_S
#define UART_GPIO_PORT    NRF_P0_S
#define UART_TX_PIN       20
#define UART_RX_PIN       22
#define UART_CTS_PIN      21
#define UART_RTS_PIN      19

#define ee_printf printf

#endif /* CORE_PORTME_H */
```

Listing 6.7: Model C++ code

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include <math.h>
#include <vector>
using namespace std;

#include "nlohmann/json.hpp"

using json = nlohmann::json;

#include "cache.h"

const unsigned int CACHE_LATENCY = 1;

class mem_map
{
public:
    int get_latency(long unsigned address);
    mem_map(json config);
    ~mem_map();

    cache* local_cache;

    int freq_convert(int parent_freq, int child_freq, int
        child_cycles);
    int check_contention();
    string name;
    int bus_clock_freq;
    int latency_cycles;
    double contention;

    struct mem_map_child
    {
        bool cacheable;
        uint32_t from;
        uint32_t to;
        mem_map* child;
    };

    vector<mem_map_child> address_range;

};
```

```
/**
 *  Note: Requires the entire address region of parents to be
     represented in their children.
 */
mem_map::mem_map(json config)
{

    this->name = config.at("name");
    this->bus_clock_freq = config.at("bus_clock_freq");
    this->latency_cycles = config.at("latency_cycles");
    this->contention = config.at("contention");

    for(json& json_cache : config.at("cache"))
    {
        this->local_cache = new cache(
            json_cache.at("ways"),
            json_cache.at("sets"),
            json_cache.at("linewidth"),
            json_cache.at("duwidth"),
            json_cache.at("lookahead")
        );
    }

    for (json& child_config : config.at("address_range"))
    {
        mem_map * child = new mem_map(child_config.at("child"));

        mem_map_child member{
            child_config.at("cacheable"),
            stoul((string)child_config.at("from"), nullptr, 16),
            stoul((string)child_config.at("to"), nullptr, 16),
            child
        };

        this->address_range.push_back(member);

    }

}


/**
 * @brief Recursive destructor for the mem_map class. Prints
     performance
 *        numbers for caches.
 */
mem_map::~mem_map()
{
    if (this->local_cache)
```

```
    {
        this->local_cache->print_performance();
        delete this->local_cache;
    }

    for ( mem_map_child i : this->address_range)
    {
        delete i.child;
    }
}


/**
 * @brief Calculates and returns the additional latency cycles
     which occur
 * when a higher frequency parent accesses a lower frequency child
     .
 *
 * @param parent_freq Clock frequency of the parent.
 *
 * @param child_freq Clock frequency of the child.
 *
 * @return Additional latency cycles due to frequency conversion.
 *
 * TODO: Check if (ratio*child_cycles) + ratio + 1 + (rand() %
     ratio) is more correct.
 * TODO: Add compatibility for child_freq higher than parent_freq.
 */
int mem_map::freq_convert(int parent_freq, int child_freq, int
    child_cycles)
{
    int ratio = parent_freq / child_freq;
    if (ratio > 1)
    {
        return (ratio * child_cycles) + 1 + (rand() % ratio);
    }
    else
    {
        return child_cycles;
    }
}


/**
 * @brief Traverses the memory map and calculates total delay of a
     data lookup
 *        request.
 *
 * @param address Address of data lookup.
 *
```

```
 * @return Total delay of data lookup.
 *
 * TODO: Keep track of hits and misses.
 */
int mem_map::get_latency(long unsigned address)
{
    if (this->address_range.size())
    {
        for (mem_map_child& child : this->address_range)
        {
            if ((address >= child.from) && (address < child.to))
            {
                if (this->local_cache && child.cacheable)
                {
                    if (this->local_cache->lookup(address, 0))
                    {
                        //cout << this->name << " hit" << '\n';
                        return CACHE_LATENCY;
                    }
                }
                int freq_cycles = freq_convert(this->
                    bus_clock_freq, child.child->bus_clock_freq,
                    child.child->get_latency(address));
                //cout << freq_cycles << '\n';
                return this->latency_cycles + this->
                    check_contention() + freq_cycles;
            }
        }
        std::cout << '\n' << "Warning: Address not found" << '\n'
            << '\n';
        return 0;
    }
    else
    {
        // add contention and frequency conversion
        return this->latency_cycles;
    }
}


int mem_map::check_contention()
{
    unsigned int cycles = 0;

    if (this->contention >= 1)
    {
        std::cout << '\n' << "Error: invalid contention value at "
            << this->name << '\n' << '\n';
    }
    while ((rand() % 100) < (int)(this->contention * 100))
```

```
    {
        cycles++;
    }
    return cycles;
}

int main()
{
    std::cout << "Start\n\n";

    ifstream infofile;
    infofile.open("nrf53app.txt");
    json info1;
    infofile >> info1;

    mem_map the_map(info1);

    json tmp2;
    string tmp;



    unsigned int l;

    string line;

    int lookupcycles = 0;


    //ifstream myfile("gmat10_10f.txt");
    //ifstream myfile("gdb_coremark_nocachef.txt");
    ifstream myfile("gmat100_100f_fake2.txt");

    while (getline(myfile, line))
    {
        l = stoul(line, nullptr, 16);
        int cycles = the_map.get_latency(l);
        lookupcycles += cycles;
    }
    myfile.close();

    std::cout << "Cycles=" << lookupcycles << '\n' << '\n';

    return 0;

}
```

Listing 6.8: Cache model code

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <vector>

class cache
{
public:
        cache(int ways, int sets, unsigned int linewidth, int
            duwidth, int lookahead = 0);
        void print_state();
        int lookup(unsigned int addr, int verbose = 0);
        void print_performance();

private:
        vector<vector<vector<int>>> data;
        int ways;
        int sets;
        unsigned int linewidth;
        int lookahead;
        vector<vector<vector<int>>> duvalid;
        vector<vector<int>> tag;
        vector<vector<int>> lvalid;
        vector<int> mru;
        int dubits;
        int lbits;
        int sbits;

        unsigned int hits;
        unsigned int misses;
};

cache::cache(int ways, int sets, unsigned int linewidth, int
    duwidth, int lookahead)
{
        this->ways = ways;
        this->sets = sets;
        this->linewidth = linewidth;
        this->lookahead = lookahead;
        this->dubits = (int) ceil(log2(duwidth));
        this->lbits = (int) ceil(log2(linewidth)) + this->dubits;
        this->sbits = (int) ceil(log2(sets)) + this->lbits;
        this->duvalid = vector<vector<vector<int>>> (ways, vector<
            vector<int>>(sets, vector<int>(linewidth, 0)));
        this->tag = vector<vector<int>>(ways, vector<int>(sets, 0)
            );
        this->lvalid = vector<vector<int>>(ways, vector<int>(sets,
```

```
            0));
        this->mru = vector<int>(sets, 0);
        this->data = vector<vector<vector<int>>>(ways, vector<
            vector<int>>(sets, vector<int>(linewidth, 0)));

        this->hits   = 0;
        this->misses = 0;
}

void cache::print_state()
{
        for (int i = this->sets - 1; i > -1; i--)
        {

                printf("hello");
        }
}

void cache::print_performance()
{
        printf("Hits=%d, Misses=%d\n", hits, misses);
}

int cache::lookup(unsigned int addr, int verbose)
{
        unsigned int tagaddr = addr >> this->sbits;
        unsigned int setidx = (addr & (0xFFFFFFFF >> (32 - this->
            sbits))) >> this->lbits;
        unsigned int duidx = (addr & (0xFFFFFFFF >> (32 - this->
            lbits))) >> this->dubits;
        int lhit = -1;
        int duhit = 0;
        if (verbose)
        {
                printf("Addr=%d, Tag=%d Set=%d, DU=%d\n", addr,
                    tagaddr, setidx, duidx);
        }

        for (int i = 0; i < this->ways; i++)
        {
                if (this->lvalid[i][setidx] && (tagaddr == this->
                    tag[i][setidx]))
                {
                        lhit = i;
                        if (this->duvalid[i][setidx][duidx])
                        {
                                duhit = 1;
                                this->mru[setidx] = i;
                        }
                        else
```

```
                {
                        this->duvalid[i][setidx][duidx] =
                            1;
                }
                break;
        }
}

if (lhit < 0)
{
        int repl = rand() % (this->ways - 1);

        while ((this->ways > 1) && (repl == this->mru[
            setidx]))
        {
                repl = rand() % (this->ways);
        }

        this->tag[repl][setidx] = tagaddr;
        this->lvalid[repl][setidx] = 1;
        for (unsigned int i = 0; i < this->linewidth; i++)
        {
                this->duvalid[repl][setidx][i] = 0;
        }
        for (unsigned int i = duidx; i < (duidx + this->
            lookahead +1); i++)
        {
                if (i < this->linewidth)
                {
                        this->duvalid[repl][setidx][i] =
                            1;
                }
        }

        this->mru[setidx] = repl;

        if (verbose)
        {
                printf("Replacing way %d", repl);
        }
}

if (duhit)
{
        hits++;
}
else
{
        misses++;
}
```

```
        if (verbose)
        {
                printf("lhit=%d, duhit=%d", lhit, duhit);

                if (verbose > 1)
                {
                        this->print_state();
                }
        }

        return(duhit);

}
```

Listing 6.9: nRF5340 JSON file

```
{
    "name"            : "CPU_BUS",
    "bus_clock_freq" : 64,
    "latency_cycles" : 1,
    "contention" : 0,
    "cache" : [{
                "ways" : 2,
                "sets" : 256,
                "linewidth" : 4,
                "duwidth" : 32,
                "lookahead" : 0
        }],
    "address_range" : [{
        "cacheable"      : true,
        "from"   : "0x00000000",
        "to"     : "0x1fffffff",
        "child" : {
            "name"            : "FLASH_BUS",
            "bus_clock_freq" : 64,
            "latency_cycles" : 1,
            "contention" : 0,
            "cache" : [],
            "address_range" : [{
                "cacheable"      : true,
                "from"   : "0x00000000",
                "to"     : "0x1fffffff",
                                "child" : {
                                        "name"            : "
                                            FLASH_MEM",
                                        "bus_clock_freq" : 64,
                                        "latency_cycles" : 1,
                                        "contention" : 0,
                                        "cache" : [],
                                        "address_range" : []
                                }
            }]
        }
    },{
        "cacheable"      : false,
        "from"   : "0x20000000",
        "to"     : "0x3fffffff",
        "child" : {
            "name"            : "RAM_BUS",
            "bus_clock_freq" : 64,
            "latency_cycles" : 0,
            "contention" : 0,
            "cache" : [],
            "address_range" : [{
```

```
                "cacheable"      : false ,
                "from"   : "0x20000000",
                "to"     : "0x2003ffff",
                                "child" : {
                                        "name"           : "
                                            RAM_MEM1",
                                        "bus_clock_freq" : 64,
                                        "latency_cycles" : 0,
                                        "contention" : 0,
                                        "cache" : [],
                                        "address_range" : []
                                }
            },{
                "cacheable"      : false ,
                "from"   : "0x20040000",
                "to"     : "0x3fffffff",
                                "child" : {
                                        "name"           : "
                                            RAM_MEM2",
                                        "bus_clock_freq" : 64,
                                        "latency_cycles" : 3,
                                        "contention" : 0,
                                        "cache" : [],
                                        "address_range" : []
                                }
            }]
        }
    },{
        "cacheable"      : false ,
        "from"   : "0x40000000",
        "to"     : "0x4fffffff",
        "child" : {
            "name"           : "PERIPH_BUS_NS",
            "bus_clock_freq" : 16,
            "latency_cycles" : 1,
            "contention" : 0.6,
                        "cache" : [],
            "address_range" : [{
                "cacheable"      : false ,
                "from"   : "0x40000000",
                "to"     : "0x4fffffff",
                                "child" : {
                                        "name"           : "
                                            PERIPHERAL_NS",
                                        "bus_clock_freq" : 16,
                                        "latency_cycles" : 10,
                                        "contention" : 0,
                                        "cache" : [],
                                        "address_range" : []
                                }
```

```
            }]
        }
    },{
        "cacheable"      : false ,
        "from"   : "0x50000000",
        "to"     : "0x5fffffff",
        "child" : {
            "name"            : "PERIPH_BUS_S",
            "bus_clock_freq" : 16,
            "latency_cycles" : 1,
            "contention" : 0.6,
                        "cache" : [],
            "address_range" : [{
                "cacheable"      : false ,
                "from"   : "0x50000000",
                "to"     : "0x5fffffff",
                                "child" : {
                                        "name"            : "
                                            PERIPHERAL_S",
                                        "bus_clock_freq" : 16,
                                        "latency_cycles" : 10,
                                        "contention" : 0,
                                        "cache" : [],
                                        "address_range" : []
                                }
            }]
        }
    },{
        "cacheable"      : false ,
        "from"   : "0x60000000",
        "to"     : "0xdfffffff",
        "child" : {
            "name"            : "RAM_DEV_BUS",
            "bus_clock_freq" : 64,
            "latency_cycles" : 1,
            "contention" : 0.6,
                        "cache" : [],
            "address_range" : [{
                "cacheable"      : false ,
                "from"   : "0x60000000",
                "to"     : "0xdfffffff",
                                "child" : {
                                        "name"            : "RAM_DEV
                                            ",
                                        "bus_clock_freq" : 64,
                                        "latency_cycles" : 10,
                                        "contention" : 0,
                                        "cache" : [],
                                        "address_range" : []
                                }
```

```
            }]
        }
    },{
        "cacheable"      : false,
        "from"   : "0xE0000000",
        "to"     : "0xE00fffff",
        "child" : {
            "name"            : "PRIVATE_PERIPHERAL_BUS",
            "bus_clock_freq" : 64,
            "latency_cycles" : 1,
            "contention" : 0.6,
                        "cache" : [],
            "address_range" : [{
                "cacheable"      : false,
                "from"   : "0xE0000000",
                "to"     : "0xE00fffff",
                                "child" : {
                                        "name"            : "
                                            PRIVATE_PERIPHERAL",
                                        "bus_clock_freq" : 64,
                                        "latency_cycles" : 10,
                                        "contention" : 0,
                                        "cache" : [],
                                        "address_range" : []
                                }
            }]
        }
    }]
}
```

Listing 6.10: Example List of Memory Accesses

```
0x466
0x46a
0x46c
0x46e
0x470
0x472
0x474
0x478
0x47a
0x47c
0x000004c0
0x47e
0x482
0x000004c6
0x484
0x488
0x20000e44
0x48a
0x48e
0x20000e44
```

Listing 6.11: Instruction Address Decoder python code

```
#!env python3
import sys
import time

print(__name__)
if __name__ != "__main__":
    # Assume we're running from GDB if this file is not the
        executable
    import gdb
    class GDBCommandDecode(gdb.Command):
        """Inject this script as a callable command in the GDB
            prompt"""

        def __init__ (self):
            """Creates new command 'get-addresses' in the GDB
                prompt that calls a custom function from this file
                """
            super (GDBCommandDecode, self).__init__ ("get-
                addresses", gdb.COMMAND_DATA)

        def invoke (self, arg, from_tty):
            decode_and_print(arg)

class ArmInstructionSimple():
    """
    Simplified ARM/Thumb instruction class
    ---------
    16-bit T32 relevant encodings for memory access instructions,
        taken from the ARMv8 ISA manual:
    [15:10]
    - 0101xx: load/store reg offset
        - base address in reg Rn [5:3], offset in reg Rm [8:6]
    - 011xxx: load/store word/byte imm. offset
        -0x: word - base address in reg Rn [5:3], immediate offset
            : {[10:6],2'b00}
        -1x: byte - base address in reg Rn [5:3], immediate offset
            : [10:6]
    - 1000xx: load/store halfword imm. offset
        - base address in reg Rn [5:3], immediate offset:
            {[10:6],1'b0}
    - 1001xx: load/store SP-relative
        - base address in reg SP, immediate offset: {[7:0],2'b00}
    - 1100xx: load/store multiple
        - 1x: load - base address in reg Rn [10:8], one 32-bit
            access per bit in the reg_list [7:0], incrementing by
            +4 for each
        - 0x: store - base address in reg Rn [10:8], one 32-bit
            access per bit in the reg_list [7:0], incrementing by
```

```
        +4 for each
    - 01001x: LDR (literal)
        - loads PC + immediate offset: {[7:0],2'b00}
    """
    c_INSTR_LEN = 16

    def get_bits(self, high, low=None):
        """
        Small helper function to get a slice of bits (or single
            bit) in a more familiar bit ordering
        """
        if low is None:
            # Case of single bit
            low = high
        if high < 0 or high > (self.c_INSTR_LEN - 1) or low < 0 or
             low > (self.c_INSTR_LEN - 1):
            raise ValueError("Only <= 32 bit instruction encodings
                are supported!\n\high: {:0d}, low: {:0d}".format
                (high, low))
        elif low > high:
            raise ValueError("low ({:0d}) must be <= high ({:0d})!
                ".format(low, high))

        return self.bitstring[(self.c_INSTR_LEN-1)-high:self.
            c_INSTR_LEN-low]

    def __init__(self, bitstring):
#         if len(bitstring) > ArmInstructionSimple.c_INSTR_LEN:
#             print("Warning: Bitstring ({:0d} bits) is longer than
     c_INSTR_LEN ({:0d}), expect errors!".format(len(bitstring),
    ArmInstructionSimple.c_INSTR_LEN))
        self.bitstring = bitstring

    c_REG_NAMES = [
        'r0',
        'r1',
        'r2',
        'r3',
        'r4',
        'r5',
        'r6',
        'r7',
        'r8',
        'r9',
        'r10',
        'r11',
        'r12',
        'sp',
        'lr',
        'pc'
```

```
    ]


def gdb_read_reg(reg_num):
    """ Placeholder function for hooking into GDB and returning
        the register contents """
    if reg_num > 15 or reg_num < 0:
        raise IndexError("Register number {:0d} out of range (must
            be between 0 and 15)!".format(reg_num))
    return int(gdb.newest_frame().read_register(
        ArmInstructionSimple.c_REG_NAMES[reg_num]))

def decode_and_print(arg):
    """time.sleep(0.05)"""
    try:
        if isinstance(arg, int):
            instr_intval = arg
        else:
            instr_intval = int(arg,16)
        bitstring = "{:016b}".format(instr_intval)
        for address in get_load_store_addresses(
            ArmInstructionSimple(bitstring)):
            print("=> 0x{:08x} (data memory access)".format(
                address))
    except ValueError as ve:
        print("Failed to parse value as hexadecimal number: " +
            arg)
        print(ve)


def get_load_store_addresses(word):
    """ word is a 16-bit string, returns a list of addresses this
        instruction causes acceses to, empty if none """
    # TODO: also add decoding to determine if it's a load or a
        store, because only loads affect the cache?
    access_list = []
    if word.get_bits(15,12) == '0101': # Load/store, reg offset
        base_address_reg     = int(word.get_bits(5,3), 2)
        offset_reg           = int(word.get_bits(8,6), 2)
#        print("Load/store, reg offset: base_address_reg = {:0d},
    offset_reg = {:0d}".format(base_address_reg, offset_reg))
        access_list.append(gdb_read_reg(base_address_reg) +
            gdb_read_reg(offset_reg))
    elif word.get_bits(15,13) == '011': # Load/store, word/byte,
        imm offset
        base_address_reg     = int(word.get_bits(5,3), 2)
        offset_bits          = word.get_bits(10,6)
        if word.get_bits(12) == '0': # Word access, scale
            immediate with 4
            offset_bits += '00'
```

```
#         print("Load/store word/byte, imm offset: base_address_reg
    = {:0d}, offset_bits = {:s}".format(base_address_reg,
    offset_bits))
        access_list.append(gdb_read_reg(base_address_reg) + int(
            offset_bits, 2))
    elif word.get_bits(15,12) == '1000': # Load/store, halfword,
        imm offset
        base_address_reg     = int(word.get_bits(5,3), 2)
        offset_bits          = word.get_bits(10,6) + '0'
#         print("Load/store halfword, imm offset: base_address_reg
    = {:0d}, offset_bits = {:s}".format(base_address_reg,
    offset_bits))
        access_list.append(gdb_read_reg(base_address_reg) + int(
            offset_bits, 2))
    elif word.get_bits(15,12) == '1001': # Load/store, SP-relative
        base_address_reg     = 13 # R13 is stack pointer
        offset_bits          = word.get_bits(7,0) + '00'
#         print("Load/store, SP relative: offset_bits = {:s}".
    format(offset_bits))
        access_list.append(gdb_read_reg(base_address_reg) + int(
            offset_bits, 2))
    elif word.get_bits(15,12) == '1100': # Load/store multiple
        base_address_reg     = int(word.get_bits(10,8), 2)
        base_address         = gdb_read_reg(base_address_reg)
        reg_list             = word.get_bits(7,0)
#         print("Load/store multiple: base_address_reg = {:0d},
    reg_list = {:s}".format(base_address_reg, reg_list))
        for bit in reg_list:
#             if bit == '0':
#                 print("Bit was 0, no memory access...")
            if bit == '1': # was elif
#                 print("Bit was 1, memory access and increment
    address by 4!")
                access_list.append(base_address)
                base_address += 4
    elif word.get_bits(15,11) == '01001': # LDR (load literal) -
        PC + immediate
        base_address_reg     = 15 # R15 is PC
        offset_bits          = word.get_bits(7,0) + '00'
#         print("Load/store, PC relative: offset_bits = {:s}".
    format(offset_bits))
        access_list.append(gdb_read_reg(base_address_reg) + int(
            offset_bits, 2))
#     else: # Unsupported instruction or not a load/store
#         print("Not a supported load/store instruction: {:s}".
    format(word.bitstring))
    return access_list

if __name__ == "__main__":
    try:
```

```
    # if True:
        # Try to parse the string as a hexadecimal number,
            representing an Arm instruction
        arg = sys.argv[1]
        instr_intval = int(arg,16)
        bitstring = "{:016b}".format(instr_intval)
#        print("Got instruction word {:s}, converted to bitstring
    {:s}".format(arg, bitstring))
#        print("Access_list:", get_load_store_addresses(
    ArmInstructionSimple(bitstring)))
    except ValueError as ve:
#        print("Failed to parse value as hexadecimal number: " +
    arg)
        print(ve)

else:
    # Register the command in GDB
    GDBCommandDecode()
```

Listing 6.12: GDB commands

```
target remote localhost:2331
py import sys
py sys.path.append('/home/user')
py import decode_addresses
monitor reset
break main.c:21
continue
set logging on
set height 0
while ($pc != 0x494)
x/i $pc
set $val = *(uint16_t *)$pc
py decode_addresses.decode_and_print(int(gdb.execute("output $val"
    , to_string=True).strip('"').split()[0]))
si
end
```

Listing 6.13: Python Script for Adding Missing Data Accesses

```
inF = open("gmat100_100f_fakedata.txt", "r")
outF = open("gmat100_100f_fake3.txt", "w")

counterInt = int("0x00004e38", 16)
for l in inF:
    outF.write(l)
    if l == "0x47c\n":
        outF.write('0x' + hex(counterInt)[2:].zfill(8))
        outF.write("\n")
    if l == "0x482\n":
        outF.write('0x' + hex(counterInt)[2:].zfill(8))
        outF.write("\n")
        counterInt += 4
outF.close()
inF.close()
```

NTNU
Norwegian University of
Science and Technology

NORDIC
SEMICONDUCTOR