Master’s Thesis

Memory Energy Optimizations for IoT Processors

Ricardo Gómez

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, 2016.

Memory Energy Optimizations for IoT Processors

Ricardo Gómez

Department of Electrical and Information Technology
Lund University

Advisor: Liang Liu
Flavius Gruian

June 27, 2016

Printed in Sweden
E-huset, Lund, 2016

Abstract

The main memory is often the principal culprit when it comes to energy consumption in embedded processors. This Master Thesis explores techniques to reduce the memory's contribution to the overall energy consumption. In this work, two different approaches are proposed.

The first approach explores different code compression techniques with the aim of reducing the memory size.

The second approach explores partitioning the heap into smaller memory banks to power gate blocks which are not being used by the processor.

These techniques have been successfully implemented in a Java processor and its functionality has been verified in a Xilinx FPGA. A 65 nm ASIC including the aforementioned optimizations has been designed and evaluated.

Acknowledgments

First of all, I would like to thank my advisors Liang Liu and Flavius Gruian for their continuous advice and guidance through this Master Thesis. They are responsible not only for making this thesis possible, but also for the valuable knowledge and experience that I have acquired during this year.

I want to express my gratitude towards all the people at the EIT department and my professors from Spain for helping me to become the engineer I am today. I would like to give special thanks to Oskar Andersson from EIT department for sharing his knowledge and helping me with the countless problems that I have faced during these years.

Thank you Inés for the warmth and love you have given to me. Although separated by thousands of kilometers, you have made me feel like home, and you have always given me the strength and courage to face new challenges.

I want to thank my parents and my brother for being always there supporting me and bringing the best out of myself.

Last but not least, I want to thank Luis, Claudio, Jorge, Daniel, and all my friends from Spain for the incredible moments we have enjoyed together.

Table of Contents

1 Background 1

1.1 The Internet of Things 1

1.2 JOP: A Java Optimized Processor 3

1.2.1 Introduction 3

1.2.2 Java and the JVM 4

1.2.3 Architecture 5

1.2.4 Memory units in JOP 7

1.2.5 JOP Garbage Collection 8

1.2.6 Power Profiling 8

1.3 CMOS Power and Energy 11

1.4 Thesis Contributions 12

1.4.1 Code Compression 12

1.4.2 Heap Partitioning and Power Gating 13

2 Code Compression 17

2.1 Theory 17

2.1.1 Paper Review 17

2.1.2 Compression Technique selection 18

2.1.3 Conclusion 25

2.2 Implementation 25

2.2.1 Code Compression System Overview 25

2.2.2 Offline Compression 28

2.2.3 Hardware Decompressor 30

2.2.4 Integration with JOP Toolchain 33

2.3 Verification 38

2.3.1 FPGA 38

2.3.2 ASIC 39

2.4 Results 42

2.4.1 Power 42

2.4.2 Delay 42

2.4.3 Area 42

2.5 Porting Code Compression to other architectures 42

3 Heap Partitioning and Power Gating 45
3.1 Theory . . . 45
3.2 Implementation . . . 47
3.2.1 Dynamic Memory Power Gating Protocol 47
3.2.2 Integration with JOP toolchain 48
3.2.3 Hardware integration 49
3.3 Verification . . . 51
3.4 Results . . . 51
3.4.1 Energy 51
3.4.2 Delay 53
3.4.3 Area 53
3.5 Porting Heap Partition to other architectures . . . 53

4 Conclusion 55

5 Further Work 57

References 59

List of Figures

1.1 Internet of Things Technologies 2
1.2 JOP architecture overview, as depicted in [13] 5
1.3 Java bytecode translation, as depicted in [13] 6
1.4 Cheney's Garbage Collection 8
1.5 JOP on-chip power consumption by power group 10
1.6 JOP memory power consumption by memory block 11
1.7 Code Compression proposal view 13
1.8 Heap Partitioning and Power Gating Explanation 14

2.1 Per-Byte Huffman Encoding 19
2.2 Per-N Bytes Huffman Encoding 19
2.3 Isomorphic Huffman Encoding 20
2.4 Isomorphic Huffman Decompression 21
2.5 Hybrid Huffman Encoding 21
2.6 Huffman Encoding Compression Code and Dictionary Sizes 22
2.7 Huffman Encoding Compression Ratio Results 23
2.8 Single Dictionary vs. Lumped Dictionary Results 24
2.9 Code Compression System Overview 26
2.10 Instruction Padding Process 30
2.11 Huffman Decompressor Architecture 32
2.12 Huffman Decompressor Behavior 32
2.13 Simplified FSM of the Huffman Decoder 33
2.14 Huffman decompressor integration 34
2.15 Asymmetrical FIFO's FSM 36
2.16 X-Propagation issue during system initialization 40
2.17 X-Propagation solution during system initialization 41
2.18 Code Compression Results 43
2.19 JVM and RISC compression blocks 44

3.1 Memory Fragmentation and Object Compaction 46
3.2 HW/SW Communication protocol in JOP 49
3.3 Second Proposal Hardware Integration 50
3.4 Heap Partitioning Energy profile 52

List of Tables

1.1 JOP on-chip power consumption by power group 9
1.2 JOP on-chip power consumption by source 10

Abbreviations

CISC	Complex Instruction Set Computer
CMOS	Complementary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
CR	Compression Ratio
FF	Flip Flop
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GC	Garbage Collector
IoT	Internet of Things
ISA	Instruction Set Architecture
JOP	Java Optimized Processor
PC	Program Counter
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RTL	Register-Transfer Language
ROM	Read-Only Memory
SoC	System-on-Chip
VHDL	VHSIC Hardware Description Language

Chapter 1

Background

In this section, the background of the project is presented. Firstly, a general discussion on IoT and the necessity of low power processors will be carried out. Secondly, the base processor (JOP) will be presented and its power will be profiled. Thirdly, a theoretical analysis of power consumption in CMOS circuits will be performed. Finally, two energy optimizations will be proposed.

1.1 The Internet of Things

The Internet of Things (IoT) is a broad concept that encompasses any dynamic infrastructure where different objects are interconnected, forming a global network of smart devices. This network allows objects to exchange and process information to interact between them and with the environment. This way, a response can be delivered without the need of an user as input. This concept can be applied to many different use cases, where IoT can improve the way different systems interact with the users and the environment. Smart Homes, Smart Lighting, Smart Energy, Smart Health or Smart Cars are examples of how IoT can make things "smarter". The key point is the ability of the nodes of the network to interact and process the data that is being gathered.

A common classification of node technologies is shown in Figure 1.1, as presented in [7]. The figure is divided into different areas, representing different layers of the IoT system. The leftmost area of the figure represents the IoT sensor and/or smart devices. This group can encompass any kind of device that senses and gathers information from the environment. As depicted in the figure, these devices usually contain local embedded processing. The complexity of this local processing may range from simple signal conditioning to more complex algorithms to process data. Sensing and smart devices can be connected to other devices locally, or to other layers of the system through *Connectivity nodes*. The group *Connectivity nodes* contains the wired or wireless technologies that communicate different layers of the system. *Layers of embedded processing nodes* encompasses a variable number of processing layers connected through connectivity nodes. Lastly, a cloud-based processing layer may be included, in case the specific application requires it.

Several challenges have arisen regarding the variety of technologies involved in the IoT development: security and privacy, consistent standards, technologi-

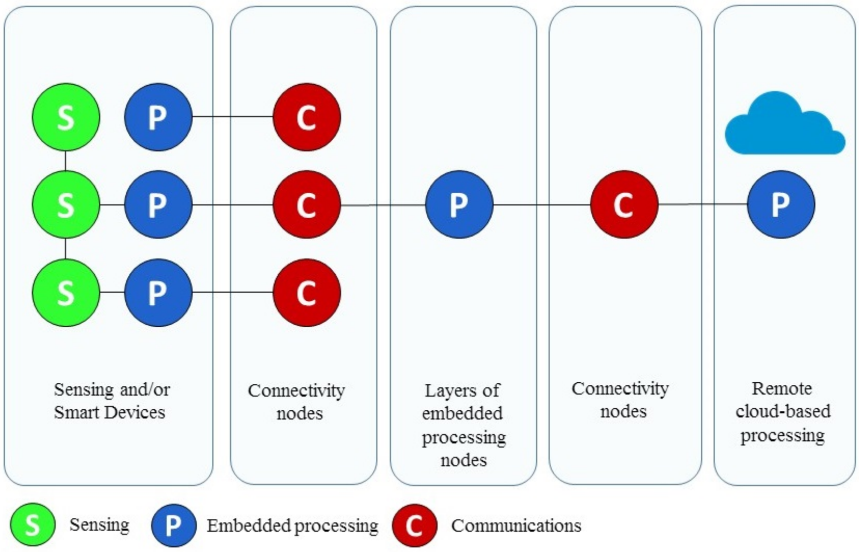


Figure 1.1: Internet of Things Technologies

cal issues, etc. Focusing on the technological challenges, it has been shown that the most crucial technological innovations for IoT is low power consumption and battery lifetime [8].

Energy optimizations can be performed at different levels of the network. This master thesis focuses the attention on the processing devices. [7] presents a list of requirements that make an MCU ideal for use in IoT. Again, low-power appears to be the crucial characteristic. However, a broad portfolio that enables different levels of performance is also mentioned. The reason for this is the large diversity of use cases and applications in the IoT. Some applications need ultra-low-power devices with simple local processing and long battery lifetime. An energy harvesting system [9] is also needed when the location of the sensor requires it. However, some other use cases require high performance embedded processing, while maintaining the energy consumption under reasonable levels. Examples of such devices are cell phones controlled by users to give input to the IoT network, or distributed computing systems in IoT [10].

[10] presents an IoT video sensor network for surveillance. In this application, the target is to store the fragments of video when important events occur. For this purpose, video processing is performed on the input stimuli, i.e. the video sequences recorded by the cameras. In a traditional non IoT setup, sensors (the cameras) stream the data to a centralized server, which performs the entire processing to detect critical events. This centralized approach presents some inconveniences: on the one hand, a high communication bandwidth is needed, as a result of all cameras streaming video back to the server. This fact contradicts the requirements on low power consumption, as wireless communication modules are usually power hungry and plays an important role in the overall power con-

sumption. On the other hand, a high processing load may overcome the server performance, limiting the amount of cameras that can be controlled. In order to overcome these limitations, an IoT distributed computing network is presented. In the distributed approach, video processing is embedded in the cameras, which perform the event detection without the need of the server. This effectively reduces transmission bandwidth up to 91.3% over the centralized system. In order to embed video processing to the devices, a SoC solution is developed and fabricated. The power consumption of this SoC and other related works [10] ranges between 50-400 mW at frequencies of 100-400 MHz. These numbers highlight the large variety in performance requirements along the different use cases in the IoT.

As it is presented in section 1.2, a Java optimized processor has been selected as a base processor to investigate the effectiveness of the proposals described in section 1.4.

It can be deduced from previous references that there are several trade-offs when discussing the features and characteristics of the ideal processor for IoT. When investigating the features of a processor for IoT, the entire ecosystem has to be considered. This includes not only the low power consumption requirement (which is probably the main issue nowadays), but also other features such as functionality and programmability.

As shown in Figure 1.1, there are many layers in an IoT network. Even inside a single layer, there might be the necessity of a broad range of processors with different profiles. In some applications the sensing and smart devices are designed to be as low power as possible. For example, some biomedical applications require simple sensors that are able to be powered by harvesting energy. However, in some other applications, such as the one shown in [10], the embedded processing is translated from higher layers to the smart devices. By embedding local processing on these devices, the energy efficiency can be increased, although these devices will consume more power.

There are several discussions on CISC vs RISC architectures for the IoT [14] and the feasibility of Java as the development language for IoT applications. [14] covers this topic, together with the advantage of reconfigurable hardware to cover the broad range of IoT applications. It chooses CISC architecture as a competitive solution for IoT application, and implements C and Java ISAs with microcode. Furthermore, it suggests instruction-specific accelerators for higher performance and efficiency. This comes in the necessity of other feature mentioned in [7]: a broad portfolio covering different levels of performance. For this reason, several IoT processors with reconfigurable architecture have been proposed.

For the reasons commented, the election of an IoT processor requires making a decision where many trade-offs are involved. Thus, a deep investigation of this trade-offs is needed and it is not covered in this Master Thesis.

1.2 JOP: A Java Optimized Processor

1.2.1 Introduction

JOP: A Java Optimized Processor [11] is an open core processor originally developed by Martin Schöberl at the Technischen Universität Wien. Since its creation, many users have contributed to the design of the system and the tool chain, including Flavius Gruian, supervisor of this Master Thesis.

JOP optimizes Java execution by implementing the Java Virtual Machine (JVM) in hardware. Among its contributions, JOP is intended to serve as embedded processor for hard real-time systems. For this reason, it exhibits a time-predictable execution of Java code integrating a real-time scheduler with the processor.

This processor has been used as a base processor for this Master Thesis, and the proposed optimizations have been implemented an evaluated in this platform.

1.2.2 Java and the JVM

In this section, a short introduction to Java and the Java Virtual Machine will be carried out. An extensive and updated description of Java and the JVM specifications can be found in [15].

Java is a general-purpose object-oriented concurrent programming language. It was originally developed by James Gosling at Sun Microsystems. Although proprietary licensed originally, since 2007 most of Java technologies are licensed under the GNU Public License. This technology consists of the Java language and standard library definition together with an intermediate ISA and execution environment.

The key point of the portability that Java offers, often referred as *write once, run anywhere*, is the JVM instruction set called Java bytecode set. The Java compiler generates bytecode, a neutral intermediate format that is independent to the device architecture. These bytecodes are interpreted by the Java Virtual Machine. The JVM can be implemented in the target device in several ways. Among them, the Just-In-Time (JIT) compilation approach is a common choice. This implementation compiles Java bytecodes to native instructions during runtime. In JOP, the JVM is implemented in hardware, where Java bytecodes form the instruction set of the processor.

The JVM defines three memory areas:

- *Method area*: The method area contains the bytecode implementation of the Java methods and the constant pool (containing literals and references to fields and methods of Java objects). In JOP, this memory section is located in the lowest addresses of the external memory.
- *Heap*: The heap is the memory area that contains the allocated objects and arrays. It also includes the handler area, where the pointers to objects in the heap are located. In JOP, the heap starts at the last address of the method area. Like the method area, it is located in the external memory
- *Stack*: The stack contains the operand stack, where the operations are performed; the local variable area, and the return frame. In JOP, the stack memory is implemented on-chip.

Unlike other programming languages, such as C or C++, where heap memory has to be deallocated explicitly, Java performs deallocation in an automatic manner with the Garbage Collector (GC). The GC is a process that automatically identifies which objects are used and which not, and deletes the unused objects. An object is said to be used if the program can reach that object. There are several GC implementation strategies that differ on how the garbage is collected: whether moving the live objects to a new area or not, whether the GC should be triggered concurrently to the main program or not, etc.

1.2.3 Architecture

Overview

Figure 1.2 shows the high-level architecture of JOP. The main units are the core pipeline, the memory controller, the extension module, the memory interface, the I/O interface and the scratchpad memory.

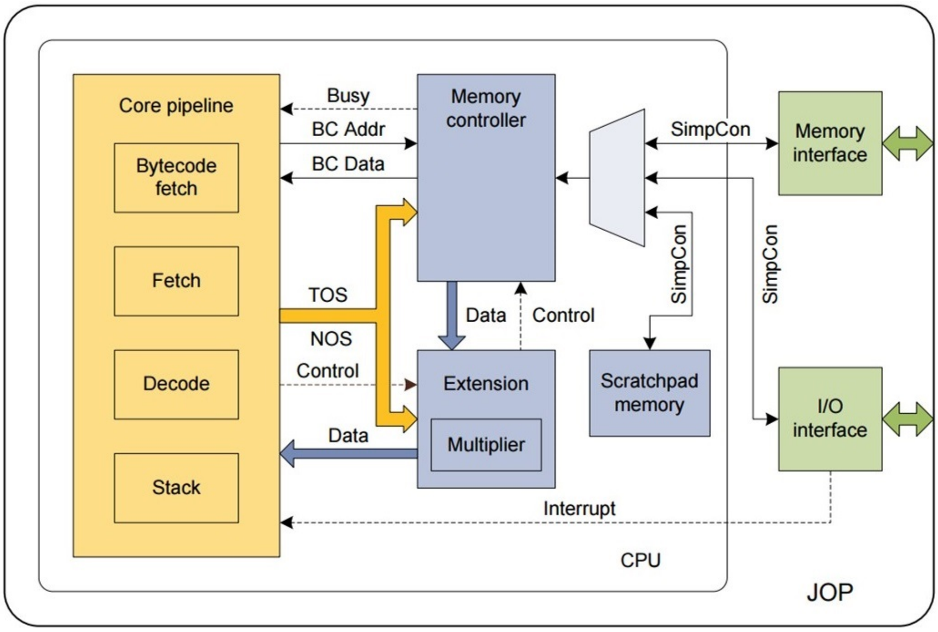

Core pipeline
Bytecode fetch
Fetch
Decode
Stack
Busy
BC Addr
BC Data
TOS
NOS
Control
Data
Memory controller
Data
Control
Extension
Multiplier
SimpCon
SimpCon
SimpCon
Scratchpad memory
Memory interface
I/O interface
Interrupt
CPU
JOP

Figure 1.2: JOP architecture overview, as depicted in [13]

JOP main modules can be described as follows:

- **Core pipeline**: The core pipeline is composed of three pipeline stages: bytecode fetch stage, microcode fetch stage, and decode and execution stage. In this block, bytecodes are fetched, translated and executed. The decode and execution stage (represented as stack in the figure) are merged into a single stage. To avoid write-back stage, data is read from the stack memory on the rising edge of the clock, and written on the falling edge. This block also contains the method cache. When a new method is invoked, the entire code is loaded into the cache, and the bytecode instructions are fetched from the method cache.

- **Memory controller**: This block controls the external memory reads/writes from/to the external memory. There is no direct connection between the core and the external memory. When a new method is requested, it performs the method read from the external memory and the method write into the method cache at the core pipeline. During this time, a busy signal is generated. This busy signal stalls the entire pipeline.
- **Extension module**: This block holds the multiplier hardware accelerator and interfaces the memory controller to the core pipeline. For example, a constant value read from the external memory is pushed into the stack by the extension module.
- **Memory interface**: The memory interface isolates the device-specific characteristics of the external memory from the rest of the system. It contains the logic needed to read and write from the external memory and offers a consistent interface to the internal bus (SimpCon).
- **I/O interface**: It controls the peripheral devices, such as external devices, and system timer and interrupts.
- **Scratchpad memory**: It holds temporary calculations and data. In this Master Thesis the Scratchpad memory is removed, as it leads to extra power consumption.

The data from the memory interface, the I/O interface and the Scratchpad memory are multiplexed to the memory controller. SimpCon, a system bus specifically developed by Martin Schöeberl [16]

Bytecode translation

The common processor architectures contain a single instruction fetch stage. However, in order to efficiently execute Java bytecodes, the JOP pipeline contains two different instruction fetch stages. Figure 1.3 represents how Java ISA is translated into microcode instructions.

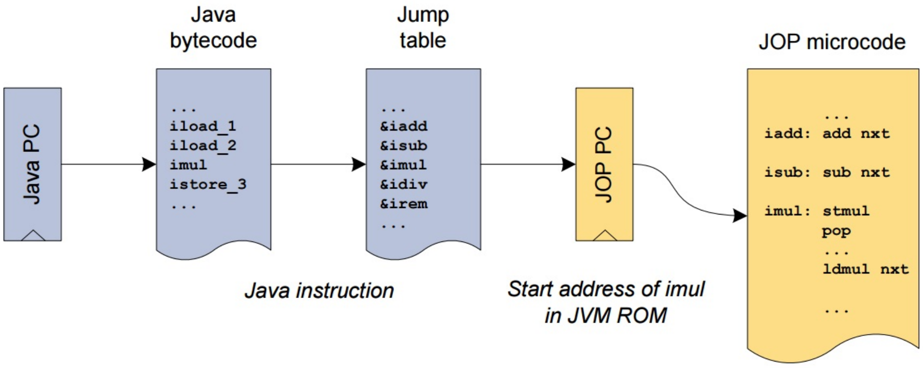

Java bytecode
Jump table
JOP microcode
Java PC
...
iload_1
iload_2
imul
istore_3
...
...
&iadd
&isub
&imul
&idiv
&irem
...
JOP PC
...
iadd: add nxt
isub: sub nxt
imul: stmul
pop
...
ldmul nxt
...
Java instruction
Start address of imul in JVM ROM

Figure 1.3: Java bytecode translation, as depicted in [13]

The Java PC holds the pointer to the Java bytecode instruction to be executed. The java bytecode is fetched from the bytecode memory and acts as an index for the jump table. The jump table, implemented as a LUT, contains pointers to the starting addresses of the microcode implementations of each bytecode. Some very simple bytecodes can be implemented with a single microinstruction, and some others need more microinstructions. When a bytecode is fetched, the starting address of the implementation of the bytecode is generated and held in the JOP PC. This PC, located in the microcode fetch stage, holds the address of the microcode instruction to be executed. The JVM ROM is the memory containing the microcode implementations of each bytecode. Every time a new bytecode is fetched from the bytecode memory, the system will execute all the microcodes that implement that specific bytecode. Thus, several microinstructions will be executed before a new bytecode is fetched. When a special microcode (*nxt*) is fetched, JOP increments the Java PC to fetch the next bytecode.

Most Java bytecodes are implemented as described before. However, some more complex bytecodes, such as *new*, are implemented as Java methods, although still executed as microcode.

1.2.4 Memory units in JOP

There are several memory units in JOP. These units are: method cache, microcode memory, object cache, stack memory, and main memory. Scratchpad memory is omitted as it has been removed for this Master Thesis.

- **Method cache**: the method cache stores the bytecode implementation of the current method. It is located at the bytecode fetch stage of the core block. It is implemented as a dual-port common-clock synchronous memory. Read and write data sizes are asymmetrical: input data is 32-bit width and output data is 8-bit width. Input data size is 32-bit due to the main memory data size, which is 32 bits. Output data size is 8-bit, as bytecodes range from 0 to 255. The total method cache size for this configuration is 2KB.

- **Microcode memory**: this unit stores the microcode implementation of every implemented Java bytecode. This unit is located at the microcode fetch stage of the core block. It is implemented as a single port ROM with a data width of 12 bits. The memory size is 24KB

- **Object cache**: this method caches the Java objects recently accessed. It is located at the extension unit. It is a dual-port common-clock synchronous memory with 32-bit data width. Its total size for this configuration is 512B.

- **Stack memory**: in this unit the core operations are performed. Thus, it is located at the stack stage of the core block. It is implemented as a true dual-port non-common clock memory. Its data width is 32 bit, and its total size is 1KB.

- **Main Memory**: this memory contains the Java code, the constant pool, and the heap. It is a single-port RAM, and it is external to the top entity of the processor. Data width can be either 16-bit and 32-bit, as the memory interface isolates the specific details of the implementation to the rest

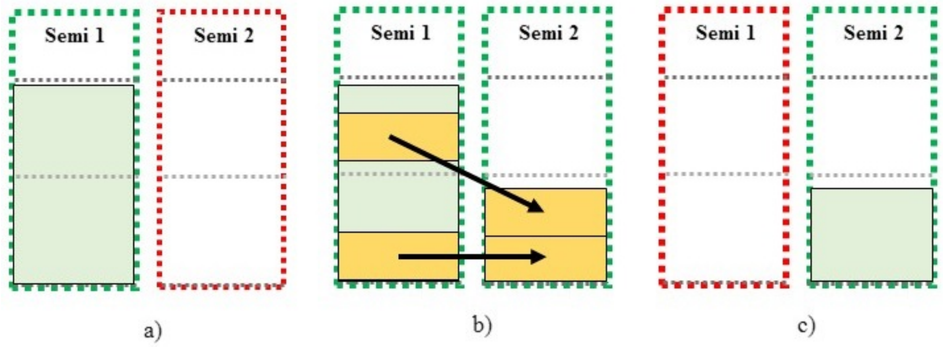


Figure 1.4: Cheney's Garbage Collection

of the system. The memory size is automatically detected by the startup software routine, which checks the physical size of the unit. However, required size can be hardcoded at the initialization routine. Depending on the application, the main memory size requirements can range from 100 KB to 2 MB.

1.2.5 JOP Garbage Collection

There are several implementations of the garbage collection task. JOP implements the Cheney's algorithm for Garbage Collection [27]. This algorithm can be explained as shown in Figure 1.4.

In Cheney's GC, the heap space is divided into two equally sized spaces: Semi 1 and Semi 2 in the figure. When an object is created, it is allocated in one of the semispaces (which receives the name of *tospace*). In Figure 1.4 a), the objects are allocated in the first semispace. When there is not enough space to allocate more objects, the garbage collection is triggered. Then, the GC starts scanning the objects to detect which are alive and which not. Finally, it copies the live objects to the second semispace, compacting them. This process is shown in b), where two live objects are being copied from the semispace 1 to the semispace 2. Then, the semispace where the objects are copied is now the *tospace*, and every time a new object is created it will be allocated there. This approach needs a larger heap space, as two identical heap regions are needed. This has an impact on both the area and the power consumption: while only one of the heap spaces is being used, the other one is leaking and consuming energy.

However, because of object compaction, this GC algorithm allows the integration of the second proposal, as it will be described in section 3

1.2.6 Power Profiling

The scope of this section is the power profiling of the base processor, JOP. Although this profiling is specific to this processor, some trends can be extended to similar designs. This profiling is intended to justify the optimization proposals in

Table 1.1: JOP on-chip power consumption by power group

Power Group	Internal Power (W)	Switching Power (W)	Leakage Power (W)	Total Power (W)	(%)
Clock Network	2.193e-03	1.050e-03	5.555e-09	3.243e-03	18.95%
Register	9.377e-05	7.473e-05	1.625e-07	1.687e-04	0.99%
Combinational	1.494e-04	3.260e-04	1.237e-07	4.755e-04	2.78%
Memory	0.0132	1.630e-05	2.280e-06	0.132	77.29%

section 1.4.

The original processor is expected to be implemented in FPGAs. Several versions targeting different FPGA vendors are available at JOP main project. These versions often utilize vendor specific primitives, which are not compatible with Synopsis tools. For this reason, migrating JOP to the ASIC flow has carried out several difficulties. This issues are addressed in section 2.3.2, where the full ASIC design flow is described in detail.

The processor has been synthesized using 65-nm high-Vt low-power libraries from ST Microelectronics. The system clock has been set to 100 MHz at 1.2V, and it has been constrained for minimum power. A post-layout simulation of a benchmark that includes basic software tasks, such as arithmetic operations and array and object creation has been performed. Lastly, a PrimeTime time-based power analysis has been carried out. The output activity file from the simulation and the parasitic extraction information has been used to increase accuracy of the results.

Table 1.2 shows the on-chip power consumption of JOP classified by power source. It can be observed that the internal power is dominant, followed by the net switching power. Cell leakage is negligible with a contribution of 0.02% to the total power consumption. This results are consistent with the frequency and operating conditions of the ASIC (100 MHz at 1.2V). At this voltage the main source of power consumption is the dynamic power consumption. This is due to the squared relation of the dynamic power consumption to the voltage, as stated in Equation (1.1).

Table 1.1 shows the on-chip power consumption of the processor classified by power group. It can be observed that the main source of power consumption in JOP are the memories. This trend is highlighted in Figure 1.5. As described in section 1.2.4, JOP contains many different memory units. For this reason, both area (as it is shown in section 2.3.2) and power consumption are mainly dominated by the memory. Figure 1.6 shows a bar chart containing the power consumption of the different memory units, including the estimated external memory power consumption. This estimation has been performed by assuming a linear relation between the memory size and the power consumption. Based on this assumption, the power numbers from the available ST memories have been extrapolated to the external memory size. In this estimation, the power consumption due to the system bus has not been included.

Regarding the presented results, several points can be remarked:

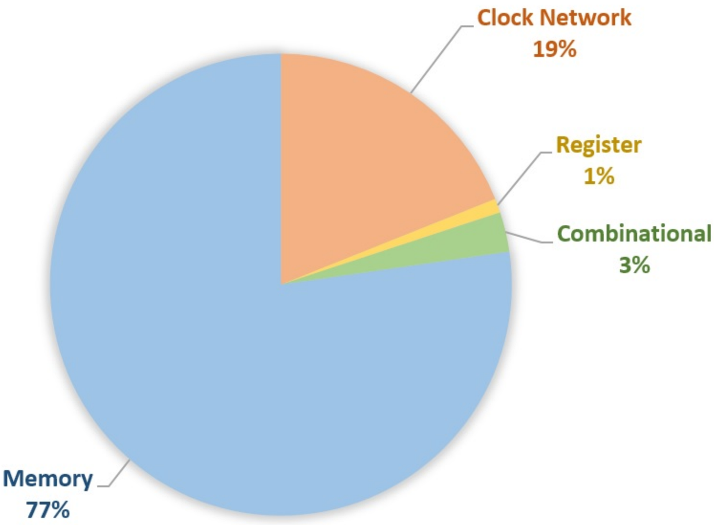


Figure 1.5: JOP on-chip power consumption by power group

- The main power consumption source in JOP are the memories. This issue is not characteristic of JOP, but a common issue in similar processors.
- The external memory dominates the system power consumption. This fact has been shown in Figure 1.6. Furthermore, it is known that roughly, on-chip memory accesses can represent even 1% of power consumption compared to off-chip memory [17].
- Dynamic power consumption has shown to be dominant on the previous simulations. However, this Master Thesis targets processors with different operating conditions. More concretely, IoT processors will operate at lower voltages and possibly lower frequencies. At these operating conditions, the static power consumption becomes non-negligible and can become dominant. This fact has been considered when proposing optimizations to reduce the energy consumption.

Table 1.2: JOP on-chip power consumption by source

Power Source	Power (W)	(%)
Net Switching	1.467e-03	8.57%
Cell Internal	0.0156	91.42%
Cell Leakage	2.572e-06	0.02%
Total	0.0171	100%

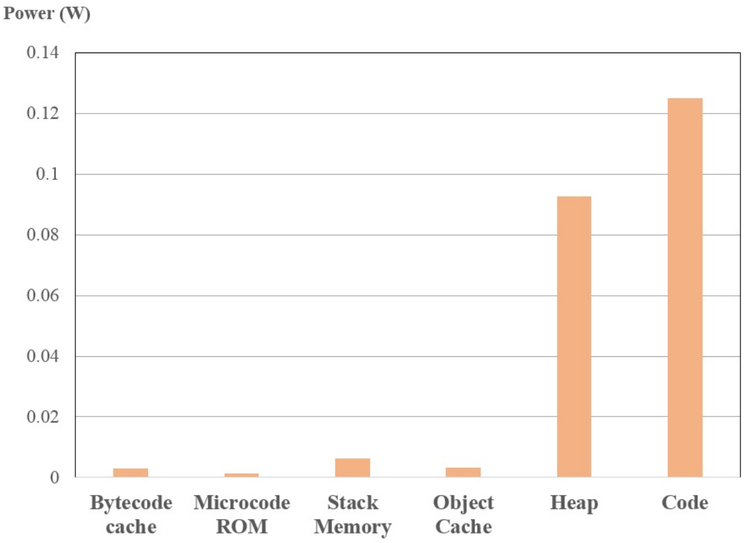


Figure 1.6: JOP memory power consumption by memory block

1.3 CMOS Power and Energy

CMOS power consumption is stated in the well-known formula (1.1).

$$P = P_{static} + P_{dynamic} = (I_{lkg}V_{DD})_{static} + (C_L V_{DD}^2 f_{clk}\alpha + I_{sc}V_{DD})_{dynamic} \tag{1.1}$$

where P_{static} stands for static power consumption, $P_{dynamic}$ for dynamic power consumption, I_{lkg} for leakage current, V_{DD} for the supply voltage of the circuit, C_L for the total capacitance being (dis)charged, f_{clk} for clock frequency, α for the number of transitions per clock cycle, and I_{sc} for direct path current.

Given the average power, P_{avg}, over a certain period of time, T, the energy consumption of the circuit can be calculated as Equation (1.2):

$$E = TP_{avg} \tag{1.2}$$

Until recently, dynamic power consumption has been the main contribution to the overall power consumption in circuits. For this reason, power optimization techniques have mainly targeted the dynamic power consumption. However, technology trends have provoked an increasing attention to static power dissipation in circuit s[1]. The reason for this is an increase of the magnitude of the static contribution over the overall power consumption. Die shrinking and smaller gate length suffer from higher leakage current [1]. Also, the decrease in the threshold voltage related to supply voltage scaling has also increased leakage currents [2].

This increasing attention has brought importance to the investigation of techniques that aim to reduce the leakage in CMOS circuits [3]. While traditionally the focus has been mostly on reducing dynamic power consumption, in newer technologies the static power consumption may even offset the savings on the overall consumption [1]. One particular example is the unfolding of filters. This technique can be applied to reduce the dynamic power consumption: unfolding a circuit allows the designer to reduce the supply voltage and still meet throughput requirements [4]. However, unfolding the circuit also implies doubling the amount of elements that are leaking. This increase on the static power consumption may overcome the savings on dynamic consumption.

Among different techniques targeting static power reduction, the following approaches can be mentioned:

- Minimizing the amount of logic present in a circuit might be the most straightforward approach to reduce the leakage. In Equation (1.1) the static power consumption is proportional to I_{lkg}. This factor represents the addition of all the leakage currents existing in a circuit. Each block of the circuit will contribute to the overall leakage current number. Thus, commonly a larger amount of logic will lead to a higher static power consumption, as there will be more leaking blocks. In this case, reducing the amount of logic present in a circuit by reutilization of digital blocks will reduce the static contribution.
- Gating the power supply of digital blocks which are not being used is another technique to reduce the leakage power consumption [5]. This technique, usually referred as power gating or MTCMOS, is based on the placement of High-V_T transistors between the power supplies and the circuits. By turning off the power gating transistors, the leakage drawn by the inactive circuit can be effectively reduced.
- Increasing the threshold voltage of the transistors by means of bulk voltage modulation also reduces the leakage power [6]. In this approach, often referred as Variable Threshold CMOS (VTCMOS), a reverse bias is applied to the substrate. As a consequence, the threshold voltage of the transistor is incremented and the leakage current diminished.

1.4 Thesis Contributions

In this section, two proposals to reduce the energy consumption in processors are presented. The thesis proposals are based on the premise that static energy consumption will dominate the overall energy consumption. This is a common fact in low-power processors that run under low voltage and low frequency operating conditions.

1.4.1 Code Compression

The first proposal is to reduce the energy consumption of processors by compressing the code which is stored in the main memory. Figure 1.7 shows the main view of the proposed architecture. It can be observed that a hardware decompressor has been placed between the external memory and the core unit.

The compiled code is compressed offline and stored in the external memory. When an instruction needs to be fetched, it is read in compressed mode from the memory. The instruction is on-chip decompressed on-the-fly. Finally, the decompressed instruction is issued to the core unit.

This reduces system energy consumption by two ways. On the one hand, by means of reducing the size of the code portion of the memory, smaller memories can be used. This reduces the energy consumption of the processor, as large memories' main source of energy consumption is commonly static. On the other hand, off-chip communications consume a significant portion of dynamic power consumption. The reason for this is the large off-chip wire capacitance of the system bus. Loading compressed methods imply lower amount of read cycles. This reduces the off-chip communications and thus the (dis)charging of large off-chip capacitances.

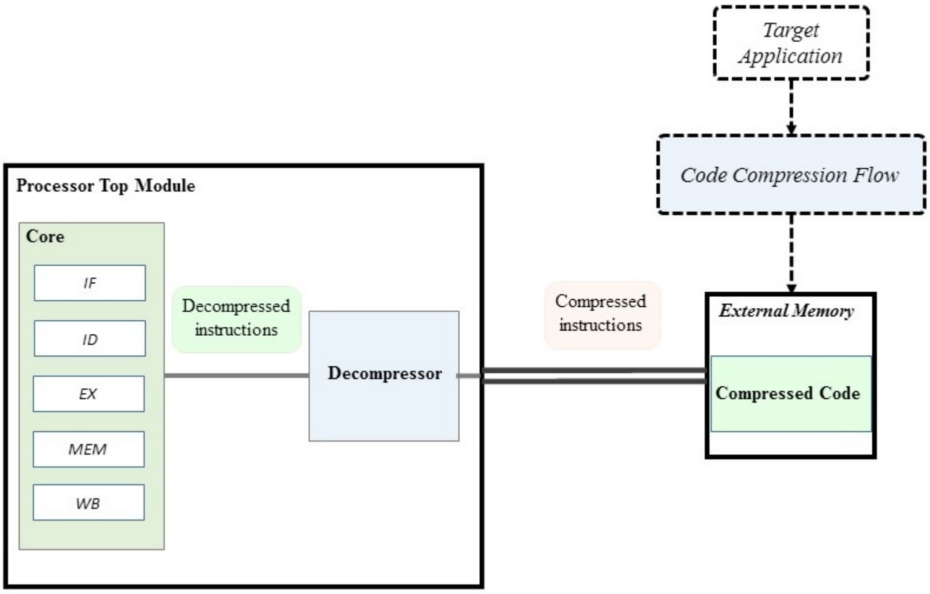

Target Application
Code Compression Flow
Processor Top Module
Core
IF
ID
EX
MEM
WB
Decompressed instructions
Decompressor
Compressed instructions
External Memory
Compressed Code

Figure 1.7: Code Compression proposal view

1.4.2 Heap Partitioning and Power Gating

The second proposal is to partition the commonly unified heap memory into smaller physical memory blocks and power gate those which are not being used. Figure 1.8 describes the proposed optimization.

In Figure 1.8 a), the initial setting of the system is shown. Three main blocks appear: the processor top module, the heap memory and the instruction memory. Inside the processor, the core block and the virtual memory are depicted. The virtual memory represents the main memory as seen from the processor: as a unified memory area. The lowest addresses belong to the code section. The rest of the virtual memory is the heap. Under *B1, B2, B3, and B4*, the regions corresponding to the physical blocks of the heap are marked only for explanation purposes. Although the memory unit is seen as a unified block, out of the processor

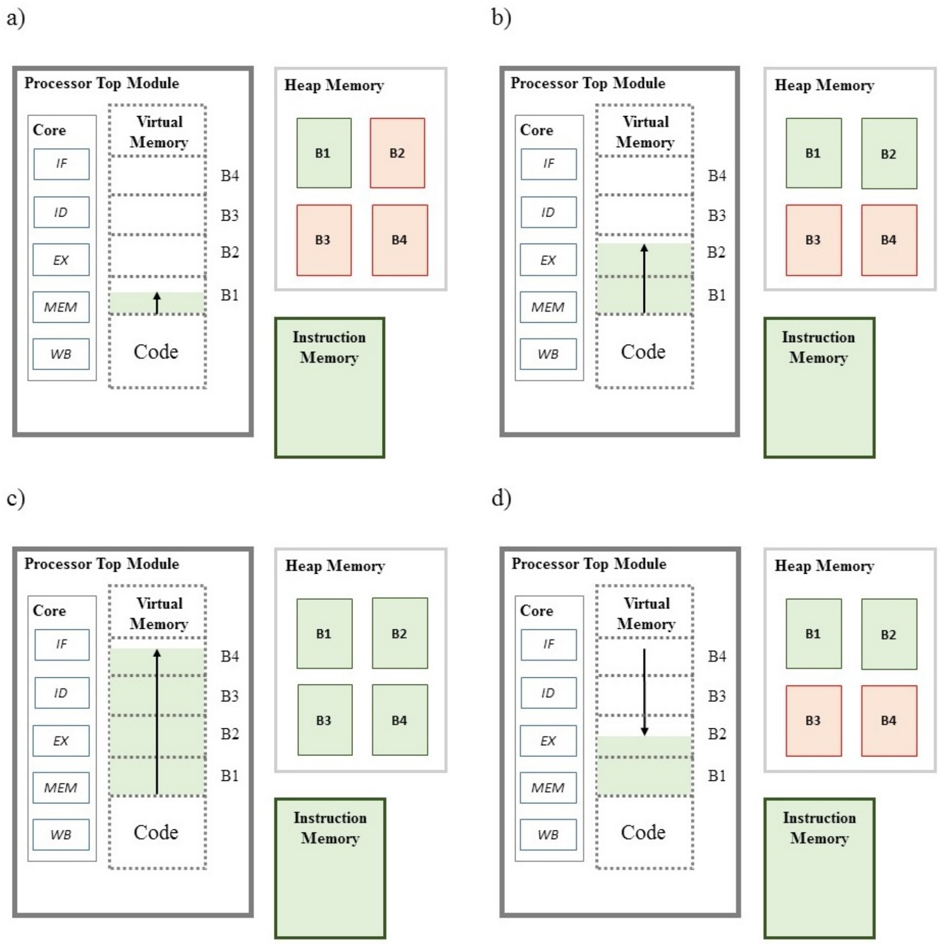

a)
b)
c)
d)
Processor Top Module
Core
IF
ID
EX
MEM
WB
Virtual Memory
Code
B4
B3
B2
B1
Heap Memory
B1
B2
B3
B4
Instruction Memory

Figure 1.8: Heap Partitioning and Power Gating Explanation

it is implemented in different hardware blocks: the Instruction Memory (containing the code) and the Heap Memory. The Heap Memory is also divided into different physical blocks, namely *B1, B2, B3, and B4*. At the virtual memory block, the memory regions belonging to the different physical blocks are marked only for illustration purposes. In this figure, powered on blocks are colored in green (*B1* in the first subfigure) and powered off blocks in red (*B2, B3, and B4*).

The behavior of the proposed system can be explained as shown in Figure 1.8:

- **a)**: In this case, a small portion of the heap memory is being used, as the entire memory that has been allocated can fit in a single block, *B1*. For this reason, blocks *B2, B3 and B4* are powered off. This way, the leakage energy consumption of these blocks is eliminated.
- **b)**: At a certain moment, more memory needs to be allocated. This heap memory cannot fit in a single block anymore, so en extra heap block is

needed. Thus, the system waits until the power on of B2 is performed. *B3 and B4* are still powered off.

- **c)**: It shows a case where most of the memory is being used. At this point, all blocks are powered on, and thus there are no savings from a case where a single heap memory is being used.
- **d)**: Last figure shows the case where a portion of the virtual memory is freed. Again, some memory blocks can be powered off, as they are no longer needed.

Many applications have irregular memory usage over time. This system adapts the memory energy consumption to the memory requirements of the processor. Partitioning the memory into smaller blocks and powering off those not being used will dramatically reduce the leakage energy consumption.

Chapter 2

Code Compression

In this chapter, the first proposal is investigated and implemented. Firstly, a theoretical study is carried out to select the compression algorithm implemented in the system. Secondly, the system implementation is detailed. Thirdly, the verification steps of the system are described. Finally, the results are presented and the feasibility of porting code compression to other architectures is studied.

2.1 Theory

The process of reducing the size (measured in bits) of a certain amount of data is called data compression. This technique can be applied to any source of data: audio, video, etc. In this Master Thesis, data compression is applied to code. Thus, by means of compression, a reduction of the code size is intended. The effectiveness of the compression technique is measured by the Compression Ratio (CR) as stated in Equation (2.1),

$$CR = \frac{size(compressed)}{size(original)} \tag{2.1}$$

2.1.1 Paper Review

Code compression for compressors have been widely studied. In this section some of them are presented. They are classified into two groups: dictionary-based approaches and statistical-based approaches

Dictionary-based Compression Approaches

This approach is based on the fact that, in programs, many sequences of instructions are repeated. For example, it may be the case that a certain instruction A is usually followed by instructions B and C. If that is the case, the code size can be reduced by substituting this sequence of instructions by a single codeword. Then, during instruction fetching, the special codeword is decompressed and the original instructions are inflated. Thus, dictionary-based compression techniques are based on discovering patterns of data and substituting them with smaller codewords that are kept in a dictionary. During decompression, this dictionary is used to decompress the original sequence. Lempel and Ziv compression algorithms are a common family of dictionary-based approaches. Examples of algorithms of this family are the LZ77 [18] and LZW [19] In [20], a thorough study of the state of the art of several dictionary based compression approaches is performed. Some of them are mentioned next. It is important to mention that in most of the presented papers, the dictionary size is not taken into account when reporting compression ratios.

In [21], the authors propose a method based on the fact that only a certain amount of distinct instructions are generated by compilers. Thus, the compressor searches all the different instructions of a certain program and re-encodes the instructions with a new binary word. This new binary word will be log_2N bits long, being N the number of distinct instructions of the code. Compression ratios between 22.7% and 54% were achieved. [22] studies the variation on the CR by considering identical whole instructions and isomorphic instructions. Isomorphic instructions are instructions that have either the same opcode with different operands, or different opcode over the same operands. Results showed that using isomorphic compression increased the CR by at least 17%.

In [24], agglomerative clustering heuristic approaches are proposed for pattern discovery in Java code. Bytecode size reduction ranged from 15% to 25%.

Statistical-based Compression Approaches

Another family of data compression techniques is the statistical-based. These approaches are based on the entropy encoding algorithm originally proposed by David A. Huffman [23]. In Huffman's compression algorithm, the original symbols are substituted with codes of varying length. The more frequent symbols are mapped to the shorter codes, and the less frequent symbols to the longer ones. This way, the total size of the symbol string is reduced.

The way codes are built is through the construction of the Huffman tree. This step is performed offline. During online decompression, a variable length decoder detects the codewords and substitutes them with the original symbols. Thus, a dictionary has to be kept to perform the decompression, as the case of the dictionary-based approaches.

Several examples of statistical-based code compression implementations can be found in [20].

2.1.2 Compression Technique selection

Huffman compression has been chosen as the algorithm for code compression of this master thesis. The first reason is the fact that among statistical compression algorithms, Huffman compression usually achieves the highest CR [25]. The second reason is the prefix free characteristic of Huffman codes. This means that no codeword is the prefix of other codeword, which simplifies the hardware decompression process.

Huffman code compression can be performed in several ways. Each one differing in the size of the symbol to be re-encoded: it can be applied to groups of 4 bits, 8 bits, or even to symbols of variable size. In order to decide the details and parameters of the Huffman algorithm, several experiments have been performed. In the following sections, the different candidates considered are presented.

Per-Byte Huffman Encoding

The most straightforward application of Huffman encoding is in a per-byte fashion. In this approach, the code memory is seen as a single byte stream. This byte stream is then Huffman-encoded, so that each byte is substituted by a variable length codeword. Figure 2.1 shows the process. As it can be seen, the memory is reorganized as a byte stream. The rectangle with intermittent line at the top represents the real decimal content of each byte. The Huffman encoding process is applied byte-per-byte. As a result, the two compressed instructions, add and sub will occupy less space in the compressed memory.

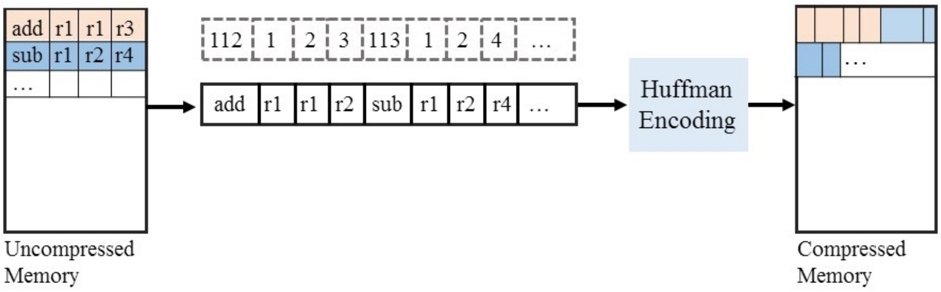

add r1 r1 r3
sub r1 r2 r4
...
112 1 2 3 113 1 2 4 ...
add r1 r1 r2 sub r1 r2 r4 ...
Huffman Encoding
...
Uncompressed Memory
Compressed Memory

Figure 2.1: Per-Byte Huffman Encoding

Per-N Bytes Huffman Encoding

This approach tries to enhance the single byte Huffman encoding CR by considering groups of 2 or more bytes. This way, the memory contents are packed into groups of N bytes, and then the Huffman tree is built. As a result, the symbols to be encoded consist of the binary concatenation of two bytes. Figure 2.2 depicts the process.

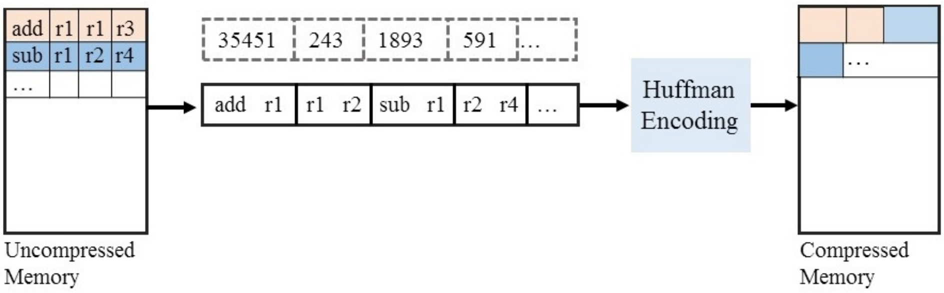

add r1 r1 r3
sub r1 r2 r4
...
35451 243 1893 591 ...
add r1 r1 r2 sub r1 r2 r4 ...
Huffman Encoding
...
Uncompressed Memory
Compressed Memory

Figure 2.2: Per-N Bytes Huffman Encoding

Per-Instruction Huffman Encoding

The previous approaches are common in data compression, where no data structure exists and the memory content is seen as a single byte stream. However, to exploit redundancy on the instruction generation by the compiler, the whole instruction (including opcond and operands) can be taken as symbol to be encoded. Depending on the ISA of the processor, symbols will contain 4 or more bytes concatenated.

The main difference of this method over the per-N Bytes Huffman encoding is the case of variable length instructions. Java ISA is not fixed length, which means that depending of the opcond, a variable length of operands will exist. Furthermore, certain Java bytecodes' lengths are not defined, as the number of operands is determined on a byte after the opcond. For this reason, symbol size is not fixed on per-instruction Huffman encoding for variable length instructions. The hardware implications of this issue are discussed at the end of this section.

Isomorphic Huffman Encoding

As presented in [22], considering the opcond and the operands as different symbol groups has the advantage of a higher CR. Figure 2.3 shows how encoding is performed. The code memory is processed and instructions are separated into opconds and operands. Then, Huffman encoding is applied to each group separately. Finally, the compressed memory is generated by substituting opcond symbols with opcond codewords, and the same with the operands.

Figure 2.4 shows the isomorphic decompression process. Instead of having one single dictionary, two dictionaries are used: opcond dictionary and operand dictionary. These dictionaries are activated alternatively to decompress the full instruction.

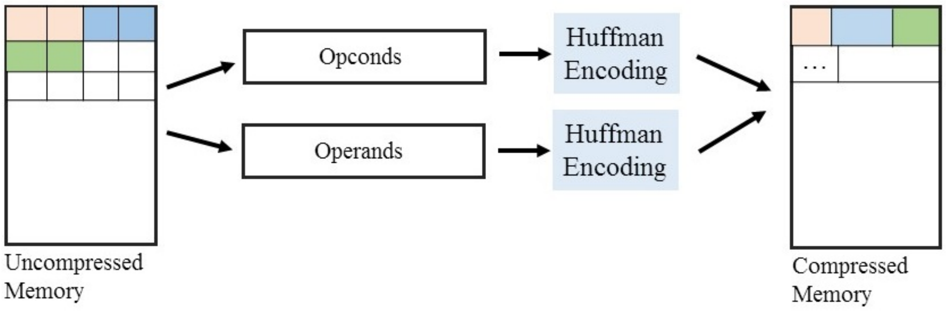


Figure 2.3: Isomorphic Huffman Encoding

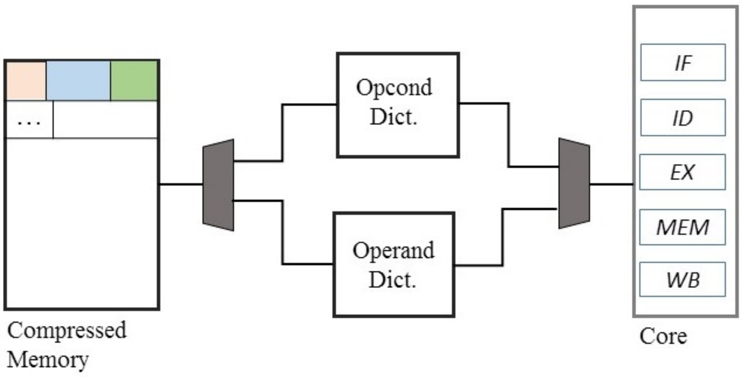

Opcond Dict.
Operand Dict.
...
Compressed Memory
IF
ID
EX
MEM
WB
Core

Figure 2.4: Isomorphic Huffman Decompression

Hybrid Huffman Encoding

A hybrid Huffman compression technique has been proposed in this Master Thesis for Java Compression. As stated before, Java bytecodes are variable length. Variable length codes' redundancy is not properly exploited by the isomorphic approach. It has been found that some large-size operands are highly repeating over the code. However, encoded codewords for these operands are not consequently short when being coded together with other operands' sizes. In order to overcome this issue, the Huffman encoder of Figure 2.5 is proposed.

In this approach, a hybrid between isomorphic and per-size Huffman encoding is presented. First, instructions are first divided into opconds and operands. Then, the operands are classified by size. After this, Huffman Encoding is applied to each group separately, and the compressed code is generated.

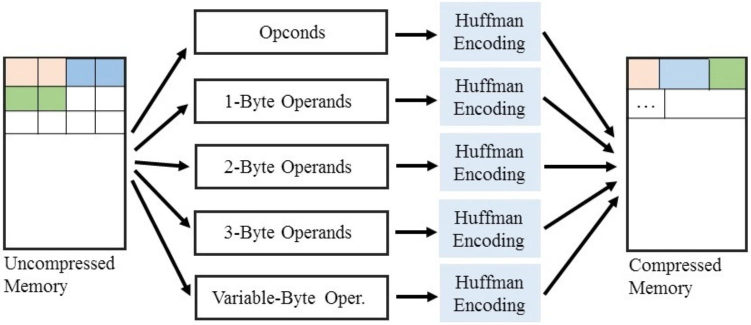

Opconds
1-Byte Operands
2-Byte Operands
3-Byte Operands
Variable-Byte Oper.
Huffman Encoding
Huffman Encoding
Huffman Encoding
Huffman Encoding
Huffman Encoding
Uncompressed Memory
...
Compressed Memory

Figure 2.5: Hybrid Huffman Encoding

During decompression, 5 different dictionaries are held, one for each group. The first byte to be decoded is the opcond. A LUT contains which bytecode opconds require 1-Byte operands, which require 2-Byte operands, etc. Thus, the

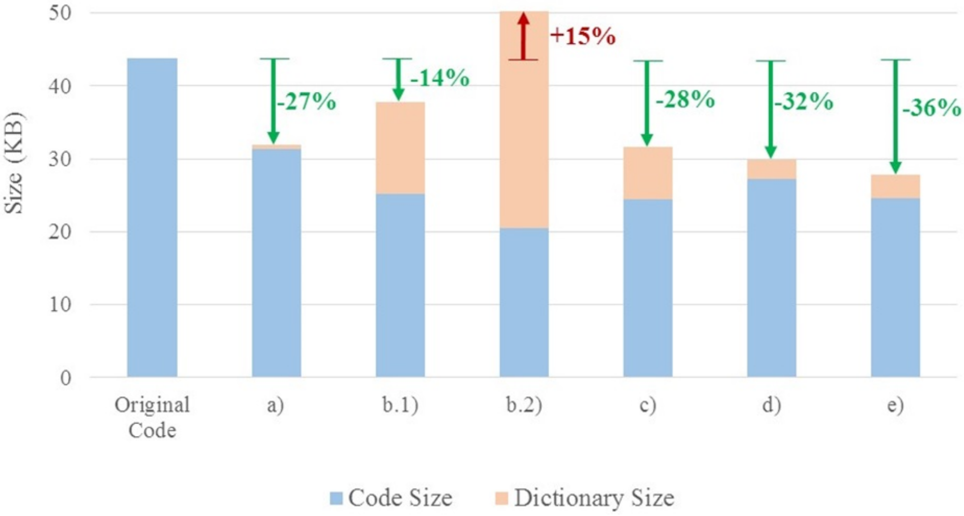


Figure 2.6: Huffman Encoding Compression Code and Dictionary Sizes

needed dictionary can be activated at each instant to decode the operand section. The rest of the dictionaries can be powered off to reduce leakage energy consumption.

Discussion

In order to select the best algorithm for the purpose of this Master Thesis, the mentioned candidates have been implemented in Python. To measure their performance, different Java programs included in the JOP project have been compiled for testing: *benchmark, dsp, lego, libcsp, sms* and *tal*. Then, this source code has been compressed and decompressed to verify the correct behavior.

Many related works do not include the dictionary size when reporting the CR of the compression algorithm. This measure is unrealistic, as dictionary size cannot be negligible [20]. For this work, the size of the dictionary has been included in the compressed size, as it gives a more realistic perspective of the CR. Figure 2.6 shows the compressed code size and dictionary size for the algorithms listed. In this figure, the *libcsp* benchmark has been used. *a)* represents per-byte compression, *b.1)* per-two bytes compression,*b.2)* per-two bytes compression,*c)* per-instruction compression, *d)* isomorphic compression, and *e)* hybrid compression. As it can be seen from the figure, original code size is 44 KB. 1-Byte Huffman compresses the size to near 31 KB, with a negligible dictionary size of only 255 entries. Above 1-Byte Huffman, the effect of dictionary size can be observed. Although achieving higher code compression than the 1-Byte Huffman, the dictionary size offsets the final result. For 3-Bytes Huffman, the dictionary size even causes a CR higher than 1. Instruction encoding achieves similar code compression as 2-Bytes Huffman, but with a significant smaller dictionary size. Isomorphic encoding performs slightly

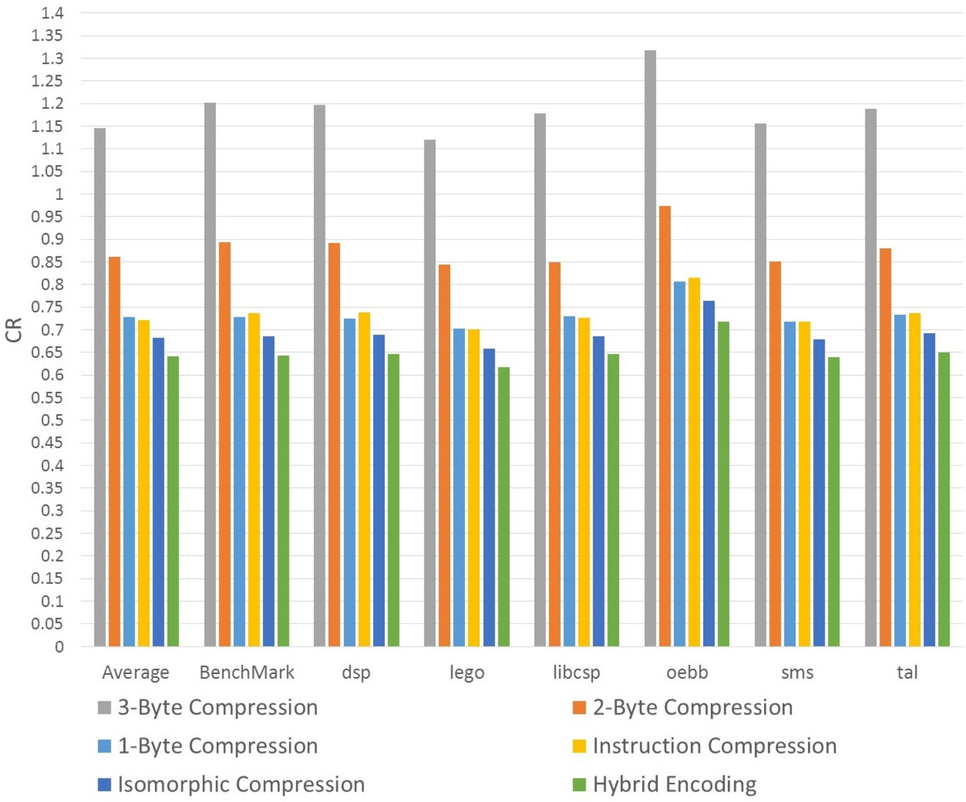


Figure 2.7: Huffman Encoding Compression Ratio Results

better, together with the proposed hybrid approach, which obtains the best results. Figure 2.7 shows the compression ratio of the listed approaches when compressing the test applications. As it can be seen, 1-Byte compression achieves similar CR as the instruction compression. As predicted by [22], isomorphic compression obtains better CR than the full instruction compression. Lastly, the proposed hybrid compression algorithms achieves the highest CR results, with an average CR of 0.64.

An important characteristic of the compression algorithm to be implemented is the simplicity of the hardware implementation. Having a complete compression-decompression flow integrated on the processor is one of the goals of this Master Thesis. For this reason, although more complex approaches achieve higher CR, simple hardware implementation is a crucial feature of the selected algorithm.

Among the presented algorithms, the simplest hardware implementation corresponds to the 1-Byte Compression. The proposed hybrid approach achieved the highest CR, however it needs an extra decoding stage to select the dictionary. The same issue is present the isomorphic compression implementation.

As an addition, an important goal of this Master Thesis is to investigate and

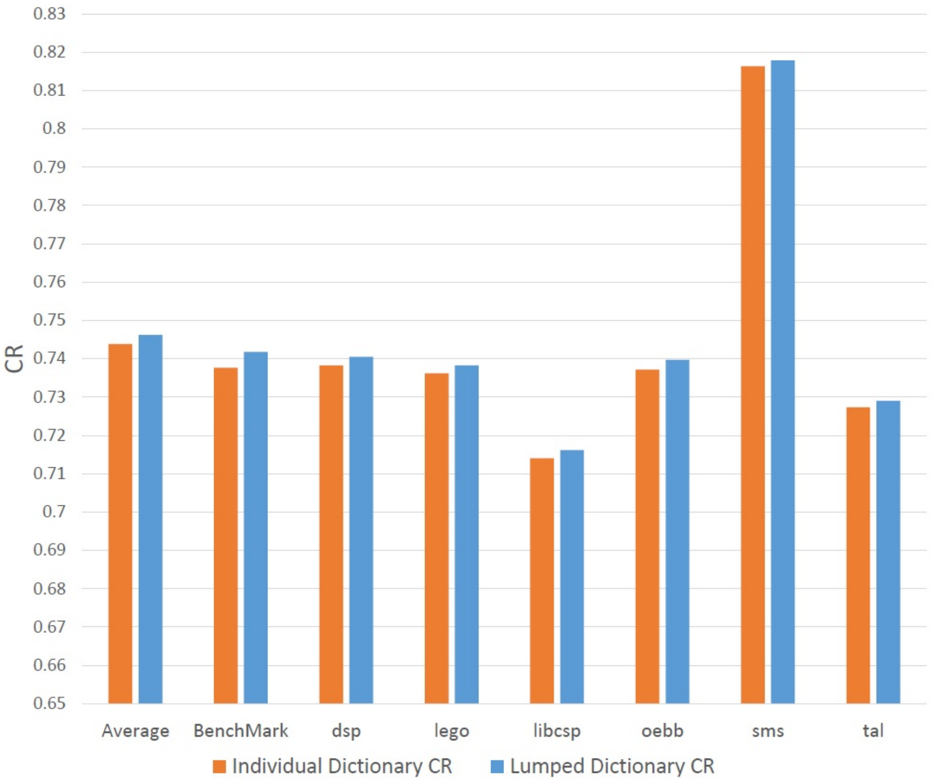


Figure 2.8: Single Dictionary vs. Lumped Dictionary Results

implement energy reduction techniques independently of the base processor. As shown before, the hybrid compression technique exploits redundancy in variable length ISAs, such as the JVM bytecode set. However, the fact that 1-Byte compression is independent to the ISA (it treats the memory as a simple byte stream) is a big advantage.

For these reasons, the technique chosen to be implemented in this Master Thesis is the per-Byte Huffman compression.

Single Dictionary vs. Lumped Dictionary

When decompressing instructions, a dictionary containing the codeword-symbol pairs is needed. This dictionary is, in principle, specific for each application: during compression phase, the number of occurrences of each symbol is annotated. After this, codewords are generated in such way that the most frequent symbol receives the shortest codeword and so on.

The aim of this Master Thesis is to develop a solution that can be integrated in any processor and to reduce the energy consumption caused by the memory. However, this conflicts with the previous point, as the hardware decompressor cannot be modified once it has been manufactured. This issue would lead to a situation where a hardware decompressor with its own dictionary would have to be manufactured for each application.

For this reason, the feasibility of generating a lumped dictionary, in a sense of a dictionary with the average occurrences of bytes across several applications, has been studied. In order to investigate this solution, different applications were compiled, and a dictionary was generated gathering byte occurrences in all of them. Then, the test applications were compressed with the lumped dictionary, and the results were compared to the CR achieved with an individual dictionary for each application. Figure 2.8 shows the results. As it can be seen, the worsening of the lumped dictionary CR is negligible, being lower than 0.3% of the compression ratio.

Regarding these results, it has been concluded that the generation of a single dictionary that covers all applications is feasible, and has a negligible impact over the CR. This allows the manufacture of a single processor with a hardware decompressor integrated, without any need of reconfiguration.

2.1.3 Conclusion

In this section, the theory behind code compression has been presented. After presenting a brief study of the state-of-art code compression techniques, statistical code compression has been chosen.

Huffman code compression can be implemented in several ways. Some of them have been presented, including a hybrid technique, which has been proposed to compress Java code. These approaches have been compared in terms of performance, complexity and ISA dependency. The big dependency of the final CR on the dictionary size has been shown. While the proposed hybrid solution has achieved the best CR results, per-Byte Huffman compression has been chosen as the technique to be implemented in this Master Thesis. The reason for this has been the good CR results and low complexity of the hardware implementation.

2.2 Implementation

In this section, the code compression technique is implemented. Firstly, a system overview containing the code compression flow is presented. Second, the offline compression process is described. Thirdly, the hardware decompressor implementation is detailed. Lastly, the integration with JOP tool chain and the issues related to it are described.

2.2.1 Code Compression System Overview

Figure 2.9 depicts the high level view of the code compression process. Green colored blocks represent software processes, blocks surrounded by a red line represent output files to the system and the remaining blocks are intermediate products.

As described in section 2.1.2, a single Huffman tree is generated from a code profiling process of several applications. For this purpose, source applications, depicted as Source App, are compiled and the respective native codes are generated.

These native codes are composed of the native opconds, operands, and ISA specific features, such as symbol tables, static references, etc.

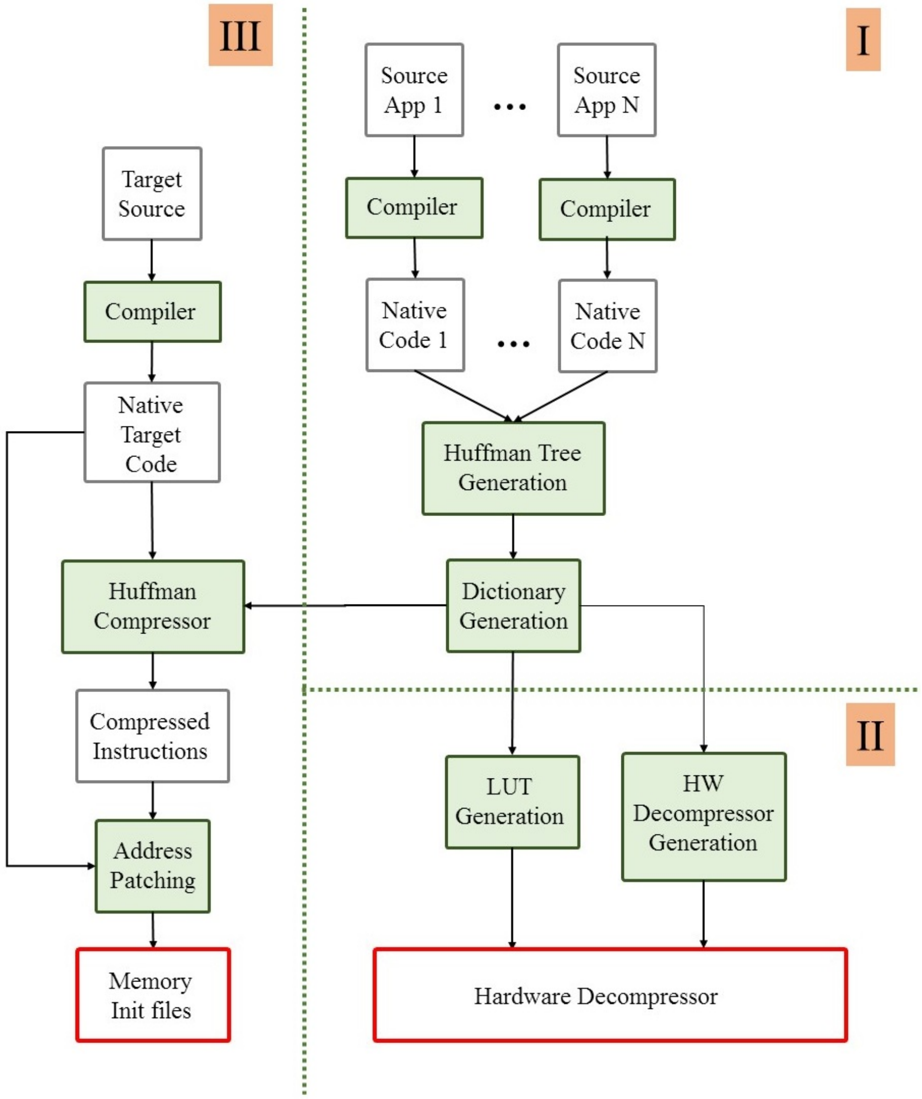

III
I
II
Source App 1
Source App N
Compiler
Compiler
Native Code 1
Native Code N
Huffman Tree Generation
Dictionary Generation
Target Source
Compiler
Native Target Code
Huffman Compressor
Compressed Instructions
Address Patching
Memory Init files
LUT Generation
HW Decompressor Generation
Hardware Decompressor

Figure 2.9: Code Compression System Overview

A Huffman Tree Generation Block generates the Huffman tree based on the occurrences of the different symbols across N applications' compiled code. This tree is then used by the Dictionary Generation block. The resulting Huffman dictionary is use in three different processes:

- **Huffman Compressor**. This block performs the compressed code generation of the target application. It receives the native target code, which has been compiled from the target source. Then, this native target code is compressed by using the generated lumped dictionary

- **LUT Generation**. During decompression, the dictionary must be held in order to detect codewords and substitute them with the original symbols. This dictionary is implemented as a LUT, which is automatically generated from the dictionary. As a result, a LUT VHDL File can be incorporated to the targeted processor.
- **HW Decompressor Generation**. Some parameters dependent on the resulting dictionary are needed for the hardware decompressor, as shown in section 2.2.3.

As detailed in section 2.2.2, a process called Address Patching is needed in order to update the static reference information of the jump addresses, method invocations, etc. This block combines the new compressed instructions with the static references and symbols from the Native Target Code. As a result, the main memory contents are generated, and an initialization file for the memory is generated.

The code compression flow has been specifically designed to be able to compress any application without modifying the underlying hardware. For this reason, the complete flow is not intended to be run every time a new application needs to be compressed. Instead, the dictionary is generated only once based on the profiling of different test applications. With this information, the hardware can be generated, integrated and manufactured. Then, the code compression generation flow (which corresponds to the leftmost branch of Figure 2.9) can be run whenever the target application needs to be recompiled. This allows the manufacture of a general purpose low-power processor and the integration of the compression flow software with the compilation toolchain.

Hardware Implementation Selection

Huffman coding is a well-known compression technique. For this reason, there are several hardware implementations. Depending on the system constraints, they may aim for higher throughput, or higher logic utilization.

As a first approach, Huffman decoders or, more generally VLC decoders can be classified into two groups: serial decoders and parallel decoders.

Serial decoders are based on the traversing of the Huffman tree, taking the right hand child node when decoding a '1' in the bitstream, and the left hand child node when decoding a '0'. When a tree leaf is reached, the decoding process of the codeword has finished. This approach has the following disadvantages:

- It requires storing the entire Huffman tree. This can suppose a significant memory overhead as the number of encoded symbols increases.
- It needs several clock cycles to decode a single symbol: as decoding requires traversing the whole tree, in worst case decoding a symbol will take the same amount of clock cycles as number of levels the tree has.

As an advantage, storing the tree in a memory allows the Huffman tree to be reconfigured if it is needed. However, as shown in section 2.1.2, using a single dictionary avoids this problem with insignificant worsening on the compression ratio.

Parallel decoders are based on shifting the bitstream serially and a table lookup operation. Thus, a LUT containing the codeword-symbol pairs will generate an output when a match between the input codeword and a coded symbol occurs. This approach requires more logic, as a LUT has to be implemented to perform the decompression process. However, as it is shown in 2.4, the overhead of this table for the 1-Byte compression is negligible. As an advantage over the serial approach, parallel decoders can offer a much faster decompression rate, up to one symbol per clock cycle.

Instructions are decompressed when required by the core block, which means that the whole system will wait for the code to be decompressed. This fact makes the decompression process' throughput an important parameter. Significantly increasing the clock cycles required for a processor to perform a certain task will increase the energy consumption. On the other hand, the parallel decoder has a larger power consumption.

However, as it is shown in 2.4, the power consumption overhead of the processor is negligible. This makes parallel decoder a more energy efficient approach, as stalling the processor every time a new instruction needs to be decoded has a larger impact on the energy consumption of the system.

2.2.2 Offline Compression

In this section, the offline compression stage of the system is covered. Python language has been used to implement the software blocks.

Huffman Tree Generation

This block generates the Huffman Tree as a previous step to generate the codewords. In order to do so, the following steps are followed:

1. The source files are read and a list is created with symbols from 0 to 255.
2. The occurrences of each symbol in the source files are annotated.
3. The list of symbols is ordered, starting with the symbols with lower frequency.
4. The two nodes at the top (the ones with lower frequency) are combined to form a parent node, with a frequency that equals the addition of the two nodes' frequencies. The two original nodes are removed from the list, and the new parent node is added to it.
5. Steps 3 and 4 are repeated until only one node remains, which becomes the top node of the tree

Dictionary Generation

The Dictionary Generation block receives the Huffman tree and generates a dictionary based on it. In order to do this, the tree is traversed until a leaf (an ending node) is reached. Everytime a right hand branch is taken, a logical '1' is added to the codeword, and a logical '0' everytime a left hand branch is taken.

As a result, a dictionary containing symbol-codeword pairs is generated.

Huffman Compressor

This block substitutes the symbols of the target file with the codewords from the dictionary. This generates a continuous compressed bitstream. In order to fit the stream into the memory width, it is partitioned into 32-bit words.

Due to the fact that Huffman encoding is a variable length encoding, the starting positions of the compressed instructions is unknown. Furthermore, they do not necessarily start at word boundaries. However, certain instructions' locations must be at word boundaries and its addresses have to be known by the program, allowing branches and function invocation. For this reason, the resulting stream has to be formatted. This introduces an offset, as certain locations contain bit padding in order to locate desired instructions at word boundaries. Figure 2.10 illustrates this process.

In Figure 2.10, Branch-targeted instructions are instructions which need to be fetched from any other section of the code (for example, because they are the branch destinations or the starting instruction of a function). In order to locate these instructions at addressable boundaries (word boundaries in this implementation), these instructions have to be detected first. Then, a padding consisting of '0's is introduced in the bitstream.

This padding introduces an overhead that worsens the compression ratio. However, this overhead is negligible, adding less than 0.3% to the overall compressed size.

Address Patching

In almost every programming language, instructions that modify the program flow exist. These instructions are branches, function calls, method invocations, etc. In order to compute the branch destination, symbol tables containing destination addresses are implemented in different manners.

As introduced before, code compression modifies the addresses of the instructions, making an update of these references across the program necessary. In 2.2.4, the address patching process for JOP and Java programs is described.

LUT Generation

Hardware decompression requires a translation from codewords to the original symbols. This hardware dictionary can be implemented in several forms. In this Master Thesis, it has been implemented as a LUT. For this reason, a Python script has been developed to automatically generate the VHDL file for the LUT depending on the generated dictionary. This LUT contains the codeword-symbol

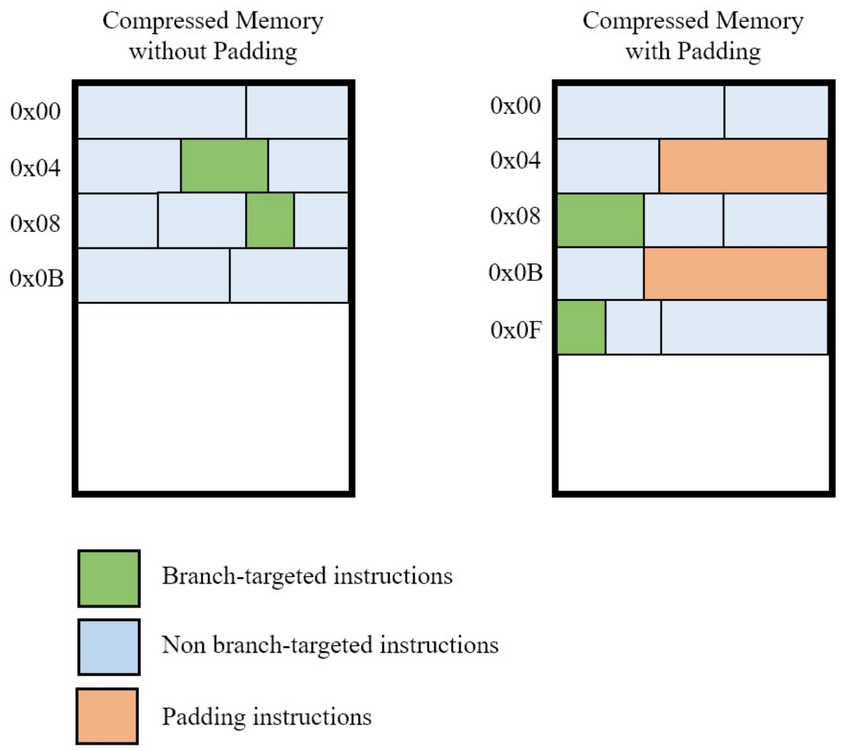

Compressed Memory without Padding
Compressed Memory with Padding
0x00
0x04
0x08
0x0B
0x0F
Branch-targeted instructions
Non branch-targeted instructions
Padding instructions

Figure 2.10: Instruction Padding Process

pairs, plus the codeword length for each pair, as this information is needed for the hardware decompressor.

2.2.3 Hardware Decompressor

In this section, the hardware decompressor implementation is detailed. This processor is automatically generated in the flow described in section 2.2.1. The integration of this decompressor in the JOP processor is described in section 2.2.4.

First, the high level architecture is shown. Then, blocks of the system that require further explanation are detailed.

Architecture Overview

Figure 2.11 shows the high level architecture of the Huffman decompressor. It is composed of the input registers D_0 and D_1, a barrel shifter, a LUT, an accumulator, a modulo operator, a simple aggregator and the FSM. FFs are represented as D in the picture. The input to the system, *data_in*, is stored at the input register D_0. The width of this signal equals the maximum codeword length, namely *maxCodeword*. D_1, a register of the same size, is connected to the output of D_0.

The concatenation of both D_1 and D_0 outputs are the input to a barrel shifter, which selects *maxCodeword* bits from the input. The output of the barrel shifter is connected to the LUT, which contains the codeword-symbol pairs. The LUT has two outputs, one outputs the decoded symbol (*lut_out*, which in our system is 8 bits wide) and the other outputs the length of the detected codeword. This length, namely *len_out*, is accumulated and used as selector at the barrel shifter. The *data_out* width is 4 bytes, as the system works over 32 bits data. However, the symbols being decompressed are 1 byte long. For this reason, 4 FFs at the output register the LUT output in order to form a system word.

Figure 2.12 shows 6 decompression cycles as an example to describe the system behavior. In this figure, D_1 and D_0 contents are represented. In this particular case, the maximum codeword length is 17 bits. Thus, both D_1 and D_0 are 17 bits-wide registers. The blue square selecting bits at the register represent the barrel shifter output.

During the first cycle, the entire D_1 register is selected as an input to the LUT. As indicated in the right hand side of the figure, symbol 00 is detected. Thus, the LUT will output the corresponding symbol, and the accumulator will increase by a value of 2. This moves the barrel shifter 2 positions to the right. During clock cycle number 2, the same procedure is repeated: this time the codeword '01' is detected and the barrel shifter changes the selected bits.

The same behavior occurs until the 5th clock cycle. This time, the amount of accumulated shifts (12 positions) plus the shifts of the new detected codeword (7 positions) exceeds the register length. At this point, new data is loaded into D_0, and the data in D_0 is transferred to D_1. In order to properly account for the last accumulated shift, a modulo operation is performed over the required translation.

FSM

In this section, the FSM of the Huffman decompressor is explained. Figure 2.13 depicts a simplified state machine of the system. The different states are explained as follows:

- **idle**: During this state, the system waits for the *enDecoding* signal to start decompressing.
- **preRead**: The system reads the first fragment of data from the bitstream. For this reason, it generates a memory read request, and waits until the data is ready. This status is controlled by the signal *mem_valid*.
- **read**: During this state, the second fragment of data is requested. The previous stored data in D_0 is transferred to D_1. When the data for the register array D_0 is ready, the system starts decoding.
- **decode**: During this state, the decoding blocks are enabled, and the data available in D_1 and D_0 is decompressed. When the barrel shifter exceeds the maximum codeword length, it returns to the read state, as new data needs to be fetched. In case no more decoding is needed, it returns to idle state.

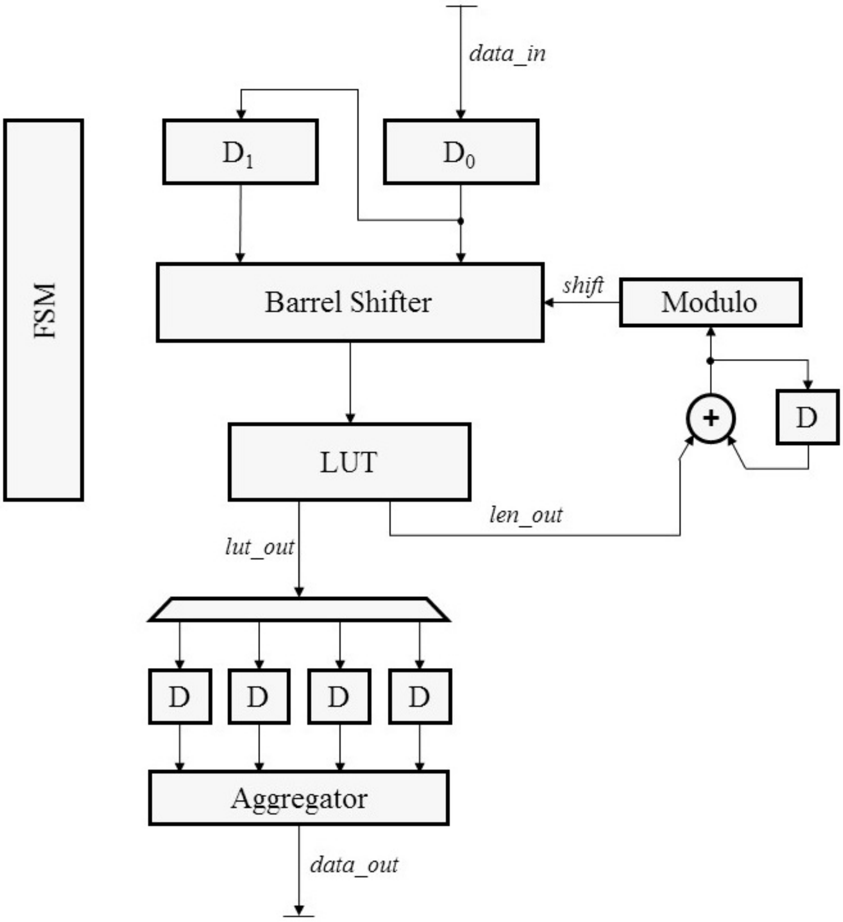

data_in
D1
D0
FSM
Barrel Shifter
shift
Modulo
+
D
LUT
len_out
lut_out
D
D
D
D
Aggregator
data_out

Figure 2.11: Huffman Decompressor Architecture

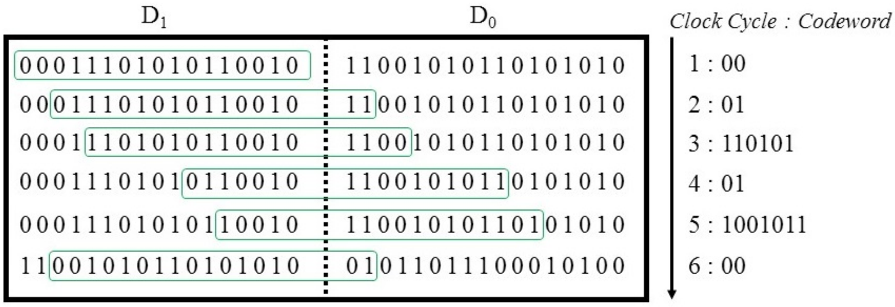

D1
D0
Clock Cycle : Codeword
00011101010110010 11001010110101010
00011101010110010 11001010110101010
00011101010110010 11001010110101010
00011101010110010 11001010110101010
00011101010110010 11001010110101010
11001010110101010 01011011100010100
1 : 00
2 : 01
3 : 110101
4 : 01
5 : 1001011
6 : 00

Figure 2.12: Huffman Decompressor Behavior

The FSM also generates the signals to control the word forming stage which appears at the bottom part of figure 2.12. When the full word is composed (4 bytes are decompressed) a signal is asserted, which informs that a new 32-bit word is ready to be read.

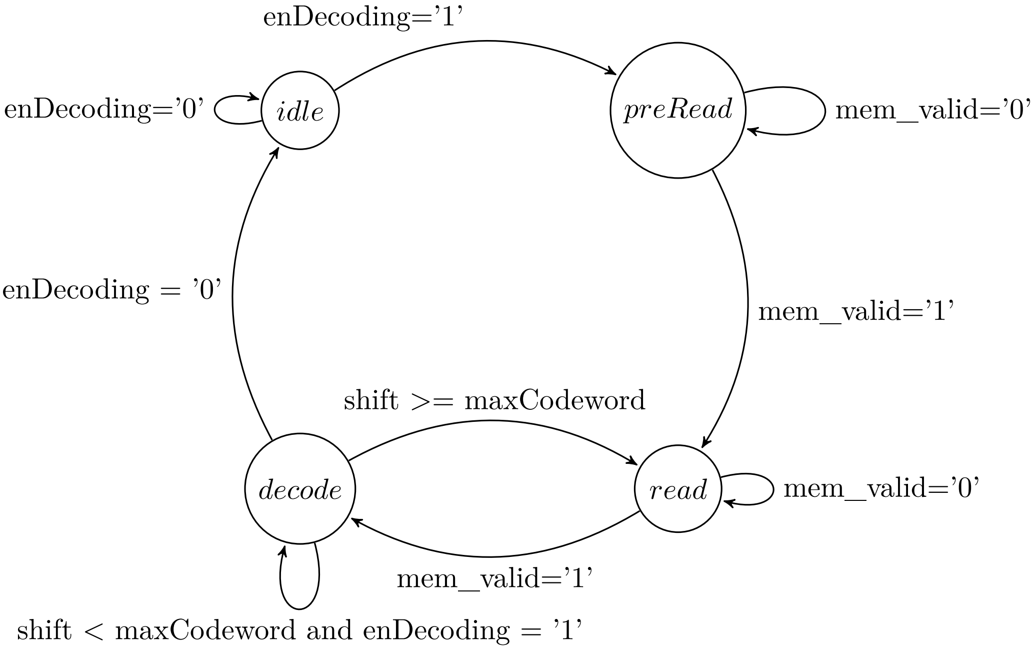

enDecoding='1'
enDecoding='0'
idle
preRead
mem_valid='0'
enDecoding = '0'
mem_valid='1'
shift >= maxCodeword
decode
read
mem_valid='0'
mem_valid='1'
shift < maxCodeword and enDecoding = '1'

Figure 2.13: Simplified FSM of the Huffman Decoder

2.2.4 Integration with JOP Toolchain

JOP processor has been used as the base processor to verify the proposals of this Master Thesis. However, integrating a complete new system into an existing processor entails some issues that require modifications on the ideal flow.

Both the integration process and the issues related are described in this section. First, the hardware integration of the hardware Huffman decompressor is detailed. Then, the issues and modifications in order to integrate the system are presented.

Hardware Integration

Figure 2.14 shows the high level view of the integrated decompressor. The decompressor top (including an FSM, the huffman decoder, and an asymmetrical FIFO) has been integrated between the memory controller and the memory interface.

JOP main memory is composed of the static references and method pool section, the code section, and the heap. However, only the code section is compressed. For this reason, it is necessary to integrate the decompression engine without modifying the normal behavior of the processor: reads and writes to other sections of

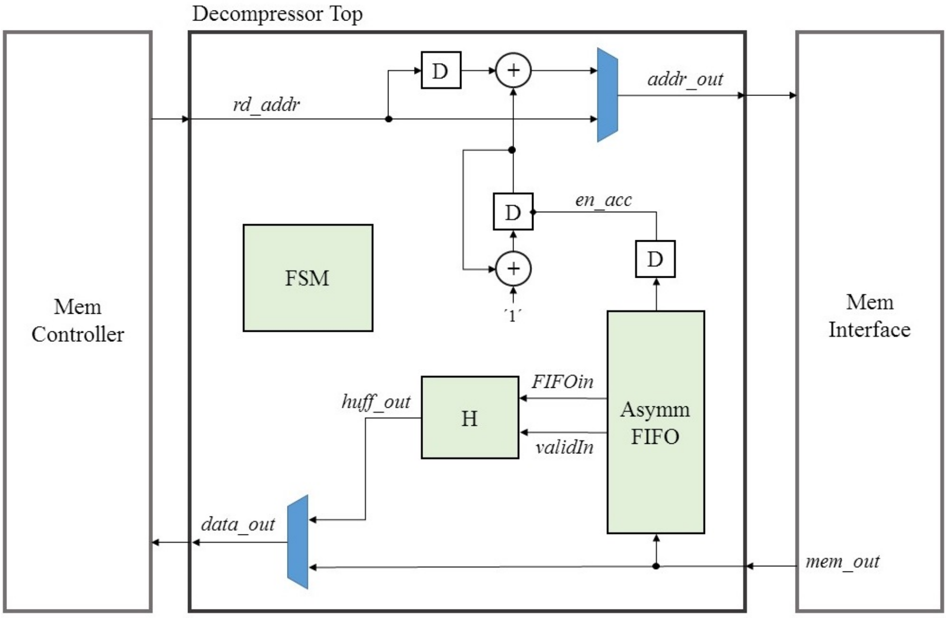


Figure 2.14: Huffman decompressor integration

the main memory need to be bypassed. For this purpose, two multiplexers bypass the read address signal *rd_addr* from the memory controller (plus some other control signals) and the output data *mem_out* from the memory interface.

Before the integration of the decompressor, the read protocol between the memory controller and the memory interface can be described using the following example:

1. When a single data read or write is requested from the core block, the memory controller outputs the address in *rd_addr*, plus a flag to indicate a word transaction request.
2. The memory interface receives the request and the address and then proceeds to read the requested word from the memory.
3. An internal counter in the memory controller is started, as the clock cycles required to read from the external memory (which is device dependent) is known. When the time expires, the data is read from the *mem_out* signal.

When more than a single word is required, for instance at a method invocation, the proceeding is similar. In this case, the memory controller holds the starting address and the size of the method invoked. Then, the memory controller generates single word requests to the memory interface (with incrementing address) until the full method is loaded.

The memory controller has been left unmodified when integrating the decompressor. Thus, during method invocation, the uncompressed size and the starting address of the methods are held in the memory controller. The following steps describe the system behavior when a method is requested:

1. In idle state (no code is requested), the read address and control signals that are generated by the memory controller are bypassed to the memory interface. In the same way, the output signals from the memory interface (including the data signal coming from the memory) are bypassed to the memory controller.
2. When a Java method invocation is performed, the decompressor is activated. The method invocation is detected by reading the internal state of the memory controller when a cache miss occurs. At this moment, the memory outputs the starting address of the method in *rd_addr*. This signal is registered and will be used as a base address to read the entire method.
3. The compressed size of the requested word in the main memory is not known beforehand. For this reason, the read requests to the memory interface are controlled by the FSM of the decompressor. The decompressor will generate as amount of read requests as are needed in order to fulfill the word request by the memory controller. These read request are generated at the asymmetrical FIFO, as explained in the assymetrical FIFO subsection. Consecutive word reads from the memory are performed with the accumulator *en_acc*, that increments the base address registered during the first read.
4. When the first word is issued to the memory controller, the system waits for another request. If the request occurs, the *rd_addr* is not registered anymore, as the correct read address is held internally. In case there is no request (as the full method has been loaded), the decompressor engine is turned off.

Asymmetrical FIFO

The data input to the D_0 register has a specific width, *maxCodeword*, which is calculated during hardware generation, as shown in Figure 2.9. However, memory data widths are commonly 16-bit or 32-bit wide. For this reason, an asymmetrical FIFO with different input and output widths has been implemented.

Figure 2.15 shows the FSM that describes the behavior of the FIFO. The asymmetrical FIFO stays on idle state until a read request from the Huffman decoder occurs. As shown before, this request is performed when new data has to be issued to the D_0 register. This FIFO is internally implemented as an array of registers and a barrel shifter. *rSize* represents the amount of bits that have been requested, and *wSize* the amount of bits that are stored in the FIFO. If a request occurs and there are not enough bits to fulfill the requirement (as signaled by rSize>=wSize) a word read request is generated at the FIFO.

This word request increments the accumulator at the top module, and a read request is sent to the memory interface. When the data is read, as represented by a logic '1' at *mem_valid*, the FIFO outputs the requested fragment of the stream and it is written into D_0, as described before.

In the implemented system, *maxCodeword* is 16 bits. For this reason, for every word that is read from the main memory, the D_0 register can be loaded twice.

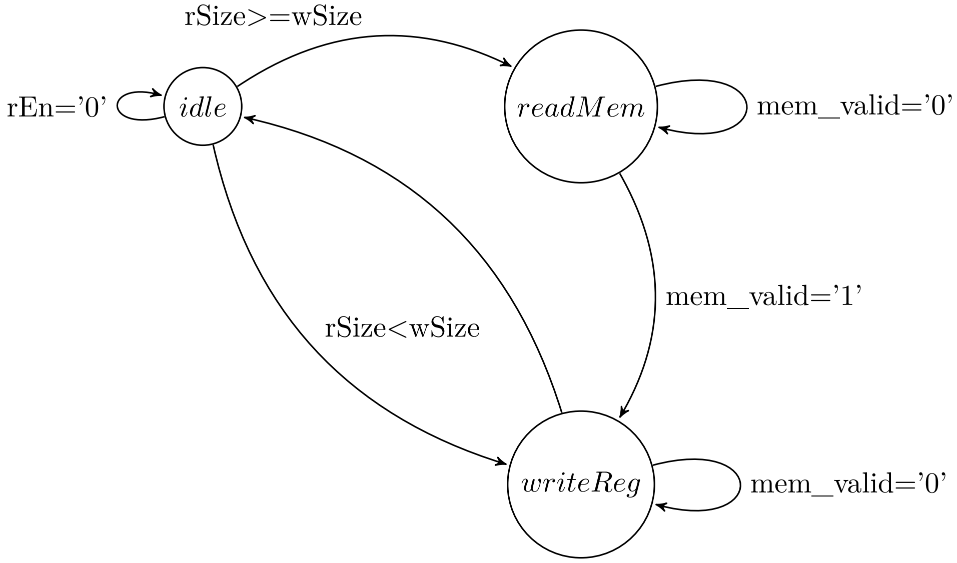


Figure 2.15: Asymmetrical FIFO's FSM

Clocked vs Handshaking memory protocol

The communication protocol between the memory controller and the memory interface is originally based on timing: as the memory read delay is known, the memory controller internally activates a timer when performing a read request. After timeout, the data is read at the input data port of the controller.

This fact supposes an issue when integrating the hardware decompressor. As the length of the words in compressed memory is unknown, a variable delay is introduced in the communication path between the memory controller and the memory interface. For this reason, it has been necessary to rewrite the communication protocol between the mentioned blocks. In order to allow a variable delay during read request, a handshaking protocol has been implemented. This handshaking protocol can be held in wait state during decompression until the required word has been decompressed.

File processing in JOP

In order to perform the whole code compression flow, the native code has to be obtained in a byte-per-byte format. JOP code compilation flow results in a .jop file which is loaded in the processor during initialization. The .jop file consists of:

- Application size and code size
- Static and reference fields
- Code section
- Special pointers

- Variable-length string table
- Class information and method table including: code start, code length, constant pool pointer and arguments' size for every method.

A Python script has been developed to process this file format and perform the following tasks:

1. Extract code section.
2. Transform 32-bit data into 8-bit data.
3. Generate the Huffman tree and dictionary.
4. Compress the code.
5. Transform 8-bit compressed stream into 32-bit data.
6. Detect branch addresses and perform bit padding.
7. Update application size, pointers and static references.
8. Regenerate the .jop file format to integrate it into the JOP software toolchain.

JOP startup modification

In order to obtain the size of the external memory, JOP has a startup method that performs native reads and writes across the external memory. However, this startup method conflicts one of the assumptions taken during system integration: the processor only performs read operations inside the code area, never write operations. For this reason, while performing reads to the compressed area, the data was being corrupted. In order to solve this problem, the Java startup procedure has been rewritten.

Java-implemented bytecodes

Although JVM bytecodes are mostly implemented in microcode (as described in section "Bytecode translation"), the most complex ones are implemented in Java. However, this feature causes an issue when integrating the decompressor. As Huffman compression consists of variable length encoding, the location of non branch-targeted instructions on the memory is unknown. This fact should not suppose a problem, as only branch-targeted instructions are accessed in branches and method invocations. However, the commented specific feature of JOP contradicts this principle. The issue can be described as follows:

1. When the system reaches one of these bytecodes, for instance *multinewarray*, a special flag is generated at the microcode fetch stage that informs the system that the bytecode is implemented in software.
2. The system checks if the method implementing the bytecode is located in the method cache. If there is a cache miss, a cache refill is requested.
3. The method is decompressed, and placed into the method cache.

4. Inside the method implementation, there is a native read back to where the method invocation was performed. The goal of this native read is to load the operands of the *multinewarray* instruction. However, as the bytecode is located inside a method (and thus, it is not a branch addreasable instruction) the original instruction cannot be found and the system crashes.

For this reason, the feasibility of having uncompressed methods has been studied. For this reason, the methods containing these problematic bytecodes were left uncompressed on the first section of the code region. Results has shown that the usage of these bytecodes is rare and thus, the amount of methods left uncompressed is negligible.

As a result, two different code sections exists. During method invocation, a register containing the uncompressed region boundary is checked. Depending on whether the required method falls into the compressed area or not, the decompressor will be activated or not.

2.3 Verification

In this section, the verification process of the proposed system is described. First, a formal verification has been realized in an FPGA. Then, in order to obtain power consumption and area results, the ASIC flow has been performed.

2.3.1 FPGA

The full processor has been implemented in a Xilinx Nexys-2 FPGA. First, the base processor has been tested. Then, the code compression engine has been integrated into the base processor. Implementing the full processor in an FPGA has carried out some modifications on the system that are described in this section.

16-bit external memory

The external memory's data width is 16 bit in Nexys-2 FPGAs, which makes the addressing mode of the memory different from the 32-bit system. Furthermore, the internal state machine of the memory interface is different from the one used for the behavioral system integration.

Several modifications have taken place in order to adapt the memory width from 32-bit to 16-bit. In order to debug these modifications on the final processor, Chipscope has been used.

External memory initialization

The external memory is initialized through a serial port. This process is integrated into the JOP toolchain: once the main memory has been loaded, the processor is booted and the system starts. The standard output from the processor is the serial port, which is connected to a software client in the computer.

However, due to modifications to the .jop file during code compression, the software client crashes after the memory initialization. In order to overcome this issue, the serial downloader program has been modified and recompiled.

Testbench

A testbench has been written to verify the correct functionality of the code compressed processor. For this reason, an application containing object initialization, floating point operations and standard out print instructions have been developed. In order to check the full integration of the decompressor with other features of JOP, the Java garbage collector has been triggered intentionally.

2.3.2 ASIC

In order to estimate the power and area overhead of the code compression engine, the full ASIC flow has been performed. ST Microelectronics Low Power 65nm HVT libraries have been used.

The original system is intended to be implemented in an FPGA, as several vendor specific primitives are used in the hardware description. For this reason, porting JOP to the ASIC flow has carried out some issues described in this section.

On-Chip Memories

On-chip memories are implemented in JOP by using vendor specific primitives. In order to design the ASIC, these memories have been substituted by the ST memories available. For this purpose, a profiling of the required memories has been performed first. Then, available memories at the department have been used to obtain the required size and behavior of the Xilinx memories. This has involved the design of wrappers that combine several memories in different manners to match the width, depth and timing of the required memories.

In some cases, larger memories than the required ones have been used. This has lead to unused memory cells, which however contribute to the power consumption. For this reason, the power profiling performed in 1.2.6 can be considered as pessimistic.

System Internal Reset

Initializing the system to a known state after reset is an important task to ensure proper functionality. In JOP, this is achieved by generating an internal reset after FPGA's programming process. By using Xilinx's register initialization primitives, an internal reset is asserted during a certain amount of clock cycles.

However, this functionality is not supported by the ASIC design flow, as there is no support for register initialization as common FPGA vendors do. For this reason, the processor has been modified to be initialized with an external reset, which is properly generated at the testbench.

On-Chip Memory Initialization

Another important issue when porting the design from the FPGA to the ASIC flow appears when the original system contains on-chip memory initialization primitives. This sort of instructions allows the memories to hold an specific value at the output during the FPGA programming process, regardless the input signals.

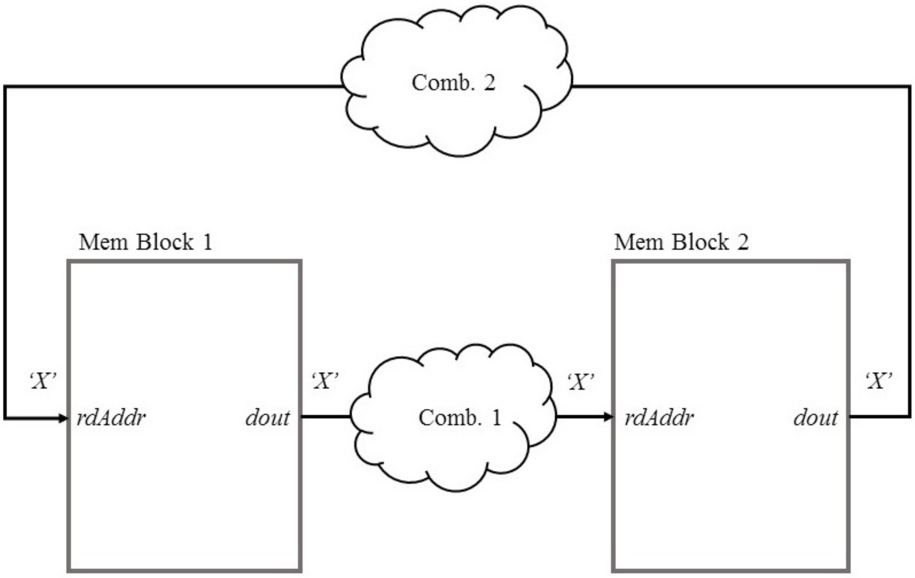


Figure 2.16: X-Propagation issue during system initialization

This allows initializing the system at a known state, even when the input address signals to the memory are not stable.

However, these primitives are not supported by the ASIC design flow. ST memories have conservative behavioral models which generate 'X' values when, for example, a non-initialized address of the memory is accessed. This 'X' value is then propagated across the circuit. When the circuit contains feedback paths, the 'X' propagation issue impedes circuit initialization.

Figure 2.16 describes the problem. In this figure, two memory blocks, namely *Mem Block 1* and *Mem Block 2* are connected via some combinational processes. After system initialization, the first memory block holds an 'X' at the data output, as the read address is not stable. *Comb. 1* is a combinational process which has as an input *dout* plus other signals. However, after synthesis, the generated netlist's simulation will propagate the 'X' from the input to its outputs, for instance the read address of the memory block 2. For this reason, *Mem Block 2* will output an invalid data signal, which is is fed back to *Mem Block 1*.

In order to overcome this issue, a memory initialization primitive's behavior is implemented as shown in Figure 2.17. In order to avoid X-propagation during initialization, multiplexers have been implemented in every memory's output. These multiplexers hold a stable and known value during one clock cycle after system's general reset.

Memory corruption during cache miss

Another issue regarding ST memories' behavioral modeling occurs when a memory address which has not been initialized is accessed. In JOP, this occurs when an a

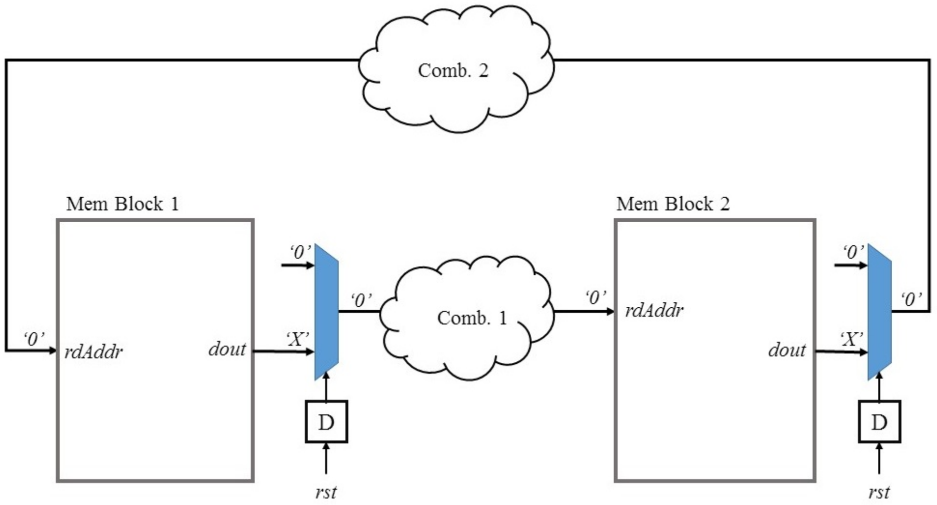


Figure 2.17: X-Propagation solution during system initialization

cache miss happens. During the method load, there are several clock cycles where the read address points to the starting cache address of the new method. Due to the fact that this method is not loaded yet, the memory will display 'X'. This 'X' does not represent an unstable signal, but an unknown output. However, as shown in Figure 2.16, the data output is usually connected to the read address of any other memory block. ST memories' content are deleted under the circumstances of an 'X' value at the address during the rising edge of the system clock. This behavior is pessimistic, as 'X' in this case does not mean unstable signal but unknown value. This fact would not have to cause any issue in the system as the output value of the second memory is not being used. Yet, the memory contents under these circumstances are deleted, crashing the system.

In order to overcome this issue, extra logic has been added to the system to detect these situations (for example, during cache miss) and multiplexers are used to hold valid signals to prevent memory corruption.

Multi-clock design constraints

The specific implementation of the pipeline stages is not presented in this Master Thesis, as it is not part of the scope. However, it is important to mention that the execution stage, namely *Stack* in Figure 1.2 is based on a multi-clock circuit. To avoid the need of a write back stage (with the overhead and data hazards that it implies) a true dual port RAM is implemented as stack memory. In this block, data is read on the rising edge of the clock, and it is written back on the falling edge of the clock. This fact implies generating a negated clock, issue that has to be constrained during the ASIC flow to ensure proper timing analysis. For this reason, specific instructions to generate the clock and analyze the timing have been added to the design flow.

2.4 Results

Figure 2.18 shows an overview of the results achieved in this work. As it is described in the following sections, the memory size and codeword transactions have been reduced, while a small and negligible power, area, and delay overhead have been introduced.

2.4.1 Power

An average size reduction of 28% of the instruction memory has been achieved, which represents an average size reduction of 54 KB for the tested applications. This average saving doubles the amount of memory integrated on-chip, which is around 27KB.

Furthermore, the amount of instruction read cycles to the external memory has been discovered to be reduced in almost the same amount as the CR (24%). For this reason, power savings will also come from dynamic power reduction due to a lower external bus activity.

The power overhead of the decompression engine has been discovered to be negligible. It has supposed a power overhead of 1.6% over the entire processor, excluding the external memory power consumption. Thus, considering the contribution of the external memory, the overhead of the decompression engine will become even smaller.

2.4.2 Delay

A negligible overhead on the system delay has been observed when running the test applications. The reason for this is the usage of a method cache, which enables reading the same method several times without activating the Huffman decompressor the same amount of times. Even without cache, the extra cycles added for decompression are negligible, and it is offset by the lower amount of words that need to be read from the external memory.

2.4.3 Area

The integration of the decompression engine has added an on-chip area overhead of 3.2%.

2.5 Porting Code Compression to other architectures

In order to port the code compression technique presented to other architecture, some modifications have to be performed:

- Depending on the targeted architecture and ISA, the code block to be compressed will be different. In Java, all the branches are produced locally inside a method. Only during method invocations the program jumps to other location out of the method. For this reason, the blocks to be compressed are the Java methods. However, in RISC microcode branches are performed in

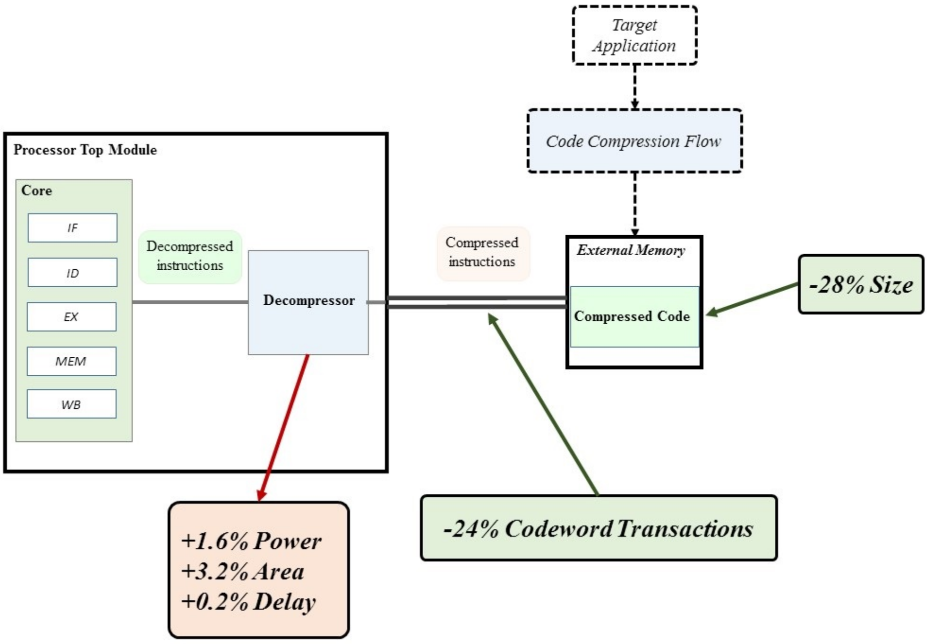
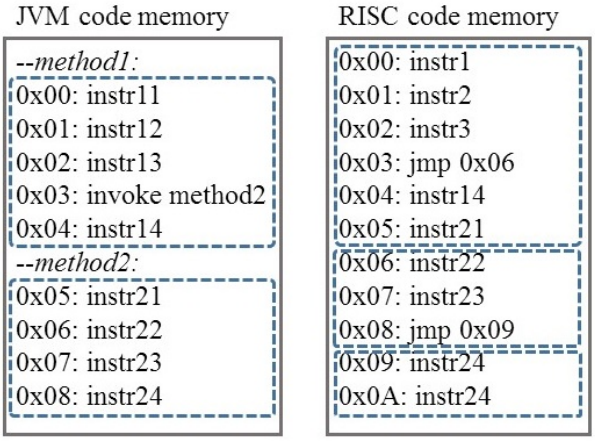

Target Application
Code Compression Flow
Processor Top Module
Core
IF
ID
EX
MEM
WB
Decompressed instructions
Decompressor
Compressed instructions
External Memory
Compressed Code
-28% Size
-24% Codeword Transactions
+1.6% Power
+3.2% Area
+0.2% Delay

Figure 2.18: Code Compression Results

a different fashion. Thus, in order to compress the code, the memory has to be processed and all the branching destinations have to be considered as delimiters of the compression blocks. Figure 2.19 describes this difference. In this figure, a JVM code memory and a RISC code memory are shown. As it can be seen, in the JVM memory the compressing blocks are the methods on themselves, as the branch addresses are the first instructions of the methods. In RISC, code is processed to obtain the branch addresses. Then, this addresses act as delimiters of the compression blocks.

- Address patching is performed in a different fashion depending on the code structure. In JOP, addresses that have to be patched are static references, parameters in the constant pool and method tables. This process differs from RISC' compiled code, where branch targets' addresses have to be patched after padding.



```
--method1:
0x00: instr11
0x01: instr12
0x02: instr13
0x03: invoke method2
0x04: instr14
--method2:
0x05: instr21
0x06: instr22
0x07: instr23
0x08: instr24
```

```
0x00: instr1
0x01: instr2
0x02: instr3
0x03: jmp 0x06
0x04: instr14
0x05: instr21
0x06: instr22
0x07: instr23
0x08: jmp 0x09
0x09: instr24
0x0A: instr24
```

Figure 2.19: JVM and RISC compression blocks

Chapter 3

Heap Partitioning and Power Gating

In this chapter, the second proposal is investigated and implemented in JOP. Firstly, a short theoretical study of power gating is performed. Secondly, the implementation process is detailed. Thirdly, the verification process is described. Lastly, the results are shown and a short discussion on porting heap partitioning to other architectures is performed.

3.1 Theory

One of the most efficient approaches to reduce the static energy consumption in circuits is power gating [3]. As shown in (1.1), static energy consumption is produced by the leakage currents. By turning off the power supply to an unused circuit, a significant portion of the energy consumption can be removed.

Power on and off circuits is usually performed by using high-V_T transistors. As these transistors present a higher threshold voltage, the leakage currents across them are smaller in comparison to standard-V_T and low-V_T transistors. For this reason, these power gating transistors are placed in between the circuit and the power supply (or in between the circuit and the ground net), dramatically reducing the leakage consumption.

The placement of these transistors carries some disadvantages. The main issue regarding power gating is the additional energy overhead spent when *waking up* the circuit. For this reason, power gating should only be used when a certain block is expected to be on *sleep* state during a significantly large period of time. Otherwise, continuously turning on and off a block can increase the overall energy consumption. Furthermore, the extra clock cycles needed for the power supply to be restored can also increase the energy consumption. Thus, a proper profiling of the block utilization prior to implementing power gating techniques is needed.

One of the most leaking blocks in common processors are the memories. The reason for this is the fact that, in a memory, only one word is accessed at most, while the rest of the memory locations are leaking. For this reason, many techniques aim to reduce leakage current in memories. This leakage energy consumption is dependent on, among other other factors, the memory size. The larger the memory size is, the higher the leakage consumption is.

In this master thesis, partitioning the heap portion of the memory into smaller blocks and power gating unused memory blocks is proposed. Figure 1.8 shows

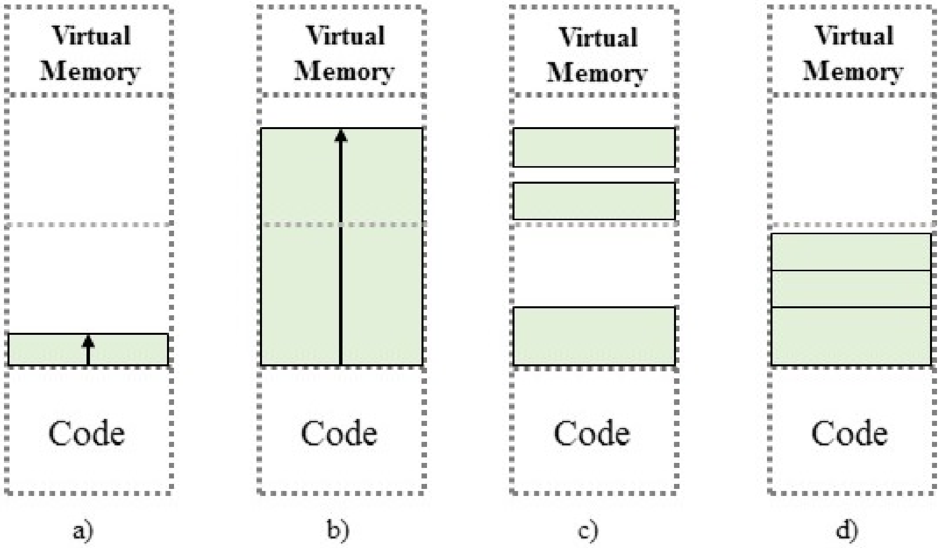


Figure 3.1: Memory Fragmentation and Object Compaction

the proposed optimization. The idea is to turn on and off memory blocks as there are live (reachable) objects stored in them. This figure, however, does not show an important feature of the processor to make power gating feasible: object compaction.

Figure 3.1 shows the object compaction issue. In this figure, the virtual memory of the processor in four different time instants is shown. This virtual memory is divided into two sections: the code section and the heap section. A dotted line dividing the heap section marks the boundary between the two physical blocks that implement the memory.

In Figure 3.1 a), a small object is allocated in the heap (for example, an array). Then, in b), more objects are allocated in the virtual memory. The allocation is performed so that the ending address of this object is the starting address of the following object (as marked by an allocation pointer). This way, empty spaces between allocated objects are avoided. Figure c) shows the memory fragmentation issue. After some portions of the memory are freed, unused regions of memory appear between live objects. This issue is often referred as memory fragmentation. As shown in the figure, memory fragmentation impedes power gating half of the heap memory, although the total size of live objects can fit into a single block.

In order to overcome this issue, an object compaction process is needed. This process eliminates fragmentation by compacting the live objects (or used regions of the memory). Figure 3.1 d) shows the virtual memory layout after object compaction. As it can be seen in the figure, this allows power gating half of the heap memory, thus saving leakage energy of the unused region.

As mentioned in the beginning of this section, in order to achieve an energy consumption improvement, a careful usage profiling has to be performed. In

[26], an investigation and profile of the heap footprints in several applications for embedded processors is performed. [26] shows several results that support the assumptions made in this proposal.

First, it is shown that heap energy constitutes 39.5% of the overall energy on the average for the selected applications. From that number, 75.6% of the heap energy is due to leakage. Thus, powering down unused regions of the heap will have a large impact on the overall energy consumption.

Second, a heap memory footprint study of several applications shows an irregular memory usage profile. This is, there are time intervals where most of the heap memory is used and intervals where only a small region of the heap memory contains live objects. For this reason, partitioning the memory into blocks and dynamically power gating them may be a effective energy reduction approach.

Third, it shows that the energy overhead due to the power gating logic and the power restoration delay of the heap memory blocks are negligible, being lower than 0.1% of the overall energy consumption.

For these reasons, memory partitioning and power gating seems to be an efficient approach to reduce the energy consumption in processors.

3.2 Implementation

In this section, the implementation of the memory partitioning and power gating and its integration to the JOP flow is shown.

3.2.1 Dynamic Memory Power Gating Protocol

The information of live/dead objects, its sizes and locations is held at a software level. However, power gating of blocks is performed at a RTL level, through the assertion/de-assertion of control signals. In order to link the high-level knowledge of the current memory layout and the low-level power control of the blocks, a system protocol has been developed. This protocol can be described as follows:

1. A variable is held at software level containing the amount of memory allocated, the heap size, and the block size.
2. During object allocation, the amount of memory available is calculated. If the object fits in the available memory, it is allocated. If it does not fit, a request to power on a memory block is requested.
3. The request is detected at RTL level, which proceeds to power on an extra block. During this time, the pipeline of the processor is stalled. After the memory has been powered on, the pipeline stall is removed.
4. The object is allocated normally.

The previous protocol describes the process of memory growth, when an object needs to be allocated and powering on a memory block is needed. However, it may happen that all the blocks are already powered on, and still extra space is needed. If that is the case, the garbage collector, which empties the memory by deleting unused objects is triggered as follows:

1. A new object needs to be allocated. If there is not enough memory to allocate it, the garbage collector is called.
2. During garbage collection, all the blocks are powered on. The GC collects the dead objects as normally.
3. When the garbage collection finishes, and the live objects are compacted, the unused memory blocks are powered off.
4. Lastly, the object is allocated normally. In case there is not enough available memory, a memory grow is requested.

3.2.2 Integration with JOP toolchain

The integration of the previous protocol in the JOP toolchain has carried out both software and hardware modifications.

Power Gating Protocol

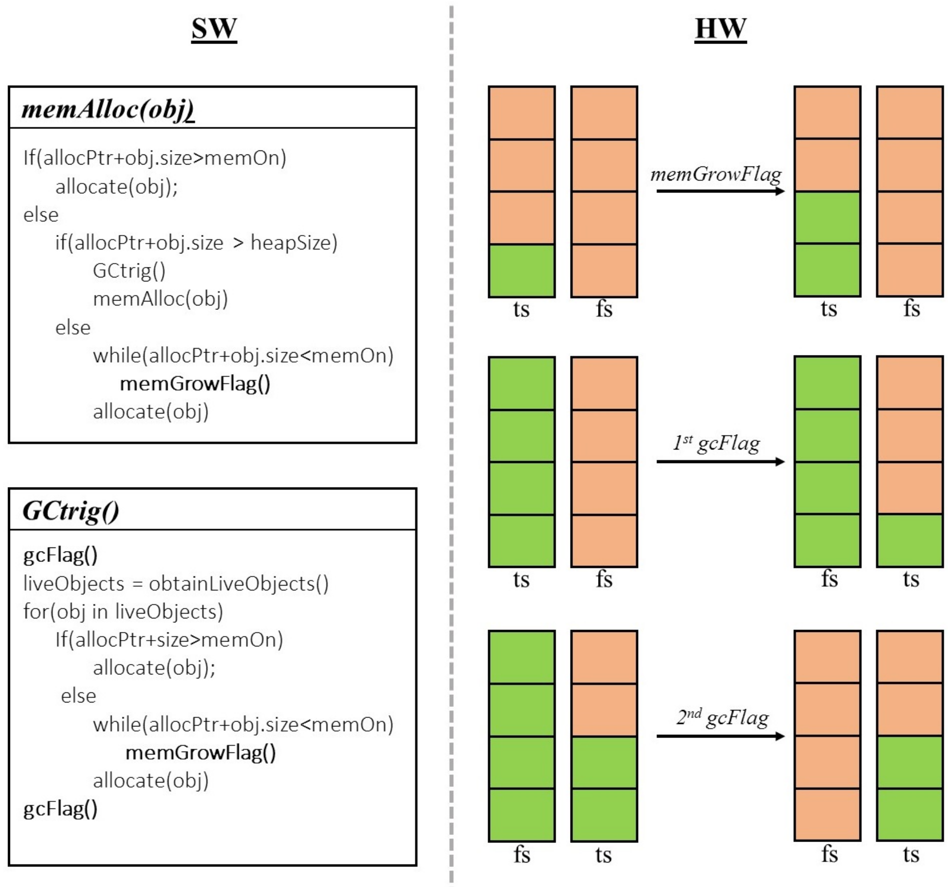
Figure 3.2 depicts the communication protocol implementation in JOP. On the left hand side of the figure, two pseudocode descriptions of the allocation method (*memAlloc*) and the garbage collection (*GCtrig*) are shown. Methods that generate hardware flags (which can be detected at RTL level) are presented in bold letters. *allocPtr* stands for the pointer to the memory when the object is going to be allocated. *obj* is the object to be allocated. *obj.size* is the size of the object to be allocated. *memOn* is the amount of memory which is powered on. *heapSize* is the total heap size of a single semispace.

The memory allocation method can be explained as follows:

1. In case there is enough memory, the object is allocated.
2. If there is not enough heap size, the GC is called. Then, the object is allocated.
3. If there is enough heap space to allocate the object, but there is not enough powered on memory, a mem grow flag is generated until there is enough powered on memory. Then, the object is normally allocated.

The Garbage Collection method can be explained as follows:

1. A GC flag is generated to inform the hardware system that the garbage collection has been triggered.
2. The live objects are obtained.
3. The objects are allocated on the new *tospace* one by one. In case there is not enough powered blocks in the new *tospace*, a mem grow flag is generated until the object can be allocated. Then, the object is allocated.
4. A GC flag is generated again to inform the hardware system that the garbage collection process has finished.



memAlloc(obj)

```
If(allocPtr+obj.size>memOn)
    allocate(obj);
else
    if(allocPtr+obj.size > heapSize)
        GCtrig()
        memAlloc(obj)
    else
        while(allocPtr+obj.size<memOn)
            memGrowFlag()
        allocate(obj)
```

GCtrig()

```
gcFlag()
liveObjects = obtainLiveObjects()
for(obj in liveObjects)
    If(allocPtr+size>memOn)
        allocate(obj);
    else
        while(allocPtr+obj.size<memOn)
            memGrowFlag()
        allocate(obj)
gcFlag()
```

Figure 3.2: HW/SW Communication protocol in JOP

On the right hand side of the figure, the memory system behavior when the software flags are detected is shown. *ts* stands for *tospace*, and *fs* stands for *fromspace*. This figure can be explained as follows:

- When a *memGrowFlag* is detected, a memory block on the *tospace* is powered on.
- When the first *gcFlag* is detected, the *fromspace* and *tospace* are switched. A single memory block is powered on the new *tospace* in order to start the object allocation.
- When the second *gcFlag* is detected, the *fromspace* is powered off.

3.2.3 Hardware integration

Figure 3.3 depicts the hardware integration of the proposed system. As it can be observed, it has been integrated out of the core pipeline, but on-chip. The

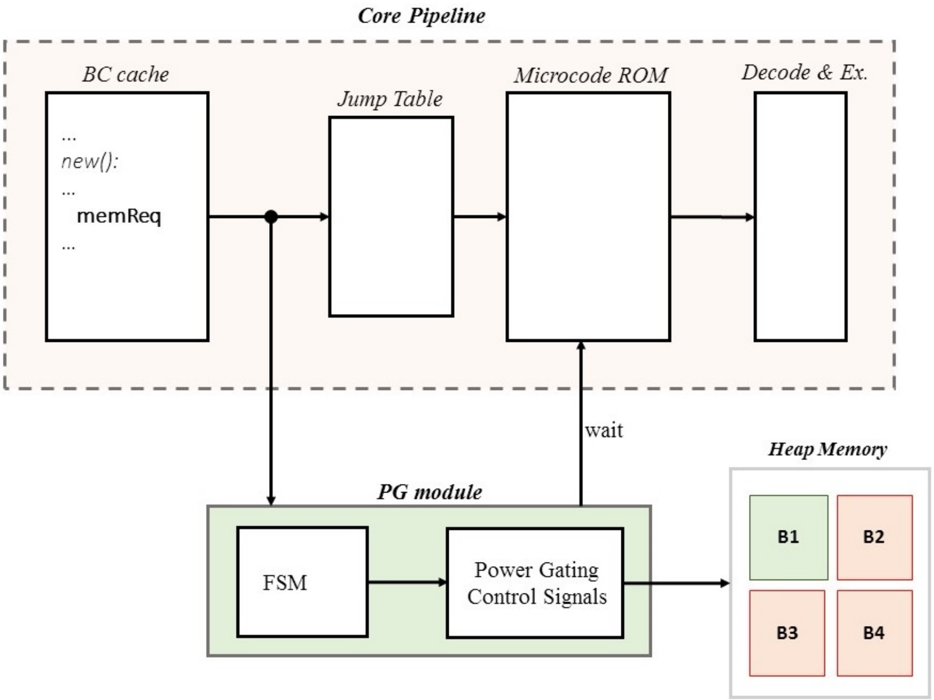

Core Pipeline
BC cache
...
new():
...
memReq
...
Jump Table
Microcode ROM
Decode & Ex.
wait
PG module
FSM
Power Gating Control Signals
Heap Memory
B1
B2
B3
B4

Figure 3.3: Second Proposal Hardware Integration

behavior of the system can be explained as follows:

1. The *new* method is invoked and loaded from the external memory to the *BC cache*
2. If there is not enough powered-on memory, the *memReq* bytecode is fetched from the *BC cache* memory and detected by the Power Gating (PG) module. The *Jump Table* generates the microcode ROM starting address of the bytecode implementation. Due to the fact that the *Jump Table* is a combinational LUT, the correct address is outputted in the same clock cycle.
3. The PG module generates a wait signal that impedes the fetching of the *nxt* microinstruction of the Microcode ROM, which would make the system fetch the next bytecode.
4. The block power on/off is performed by the PG module. This is achieved by a signal that contains the number of blocks to be powered on and off. This signal goes off-chip and controls the power gating transistors of the memories.
5. Once the memory has been powered on/off (the amount of cycles depend on the system and the technology) the wait signal is released and the *nxt* signal is fetched from the Microcode ROM.

6. the next bytecode from the BC cache is fetched, and the processor continues normally.

As described before, the power gating process has been designed to be transparent to the program. The bytecode flag is detected and the system is stalled while the memories are being powered on. After the process has taken place, the stall is released and the system continues with the next bytecode.

3.3 Verification

The system has been verified in a post-layout simulation in order to check the proper functionality of the proposed optimization. For this reason, the following steps have been performed:

1. An external partitioned memory behavioral model has been developed. In this model, the unified original memory has been divided into two different blocks: the code and static references section, and the heap. Then, the heap has been divided into 8 blocks and the logic simulating the powering on/off delays has been implemented.
2. A testbench that verifies the functionality has been developed. In this testbench, arrays are created in an iterative fashion. In every iteration, three new arrays are allocated and the three previous arrays references are deleted. This way, the previously allocated arrays become dead objects and will be deleted by the GC.

The expected behavior has been verified: the processor requests the powering on of a new block every time it is needed, and the whole system is stalled until the block is waken up. Then, it allocates the object correctly. When the processor runs out of heap memory, the garbage collector is triggered. The GC only reallocates the live objects (the three last arrays) and the unused memory blocks are powered off. Various testbenches with different sizes and behaviors have been developed to verify the functionality in different corner cases.

3.4 Results

3.4.1 Energy

Powering off the *fromspace* between GC tasks reduces the static energy consumption by 50%. In addition to this number, a variable energy saving is achieved depending on the memory footprint of the application. Figure 3.4 depicts an example.

In this figure, three different plots are shown. First, the virtual memory represents the actual amount of memory being currently used by the processor (as seen from the software side). Unified-Heap memory represents the amount of powered-on memory in the case no heap partitioning and power gating technique is applied. Finally, powered-on memory shows the current amount of memory which is powered considering a memory partitioned in 5 banks.

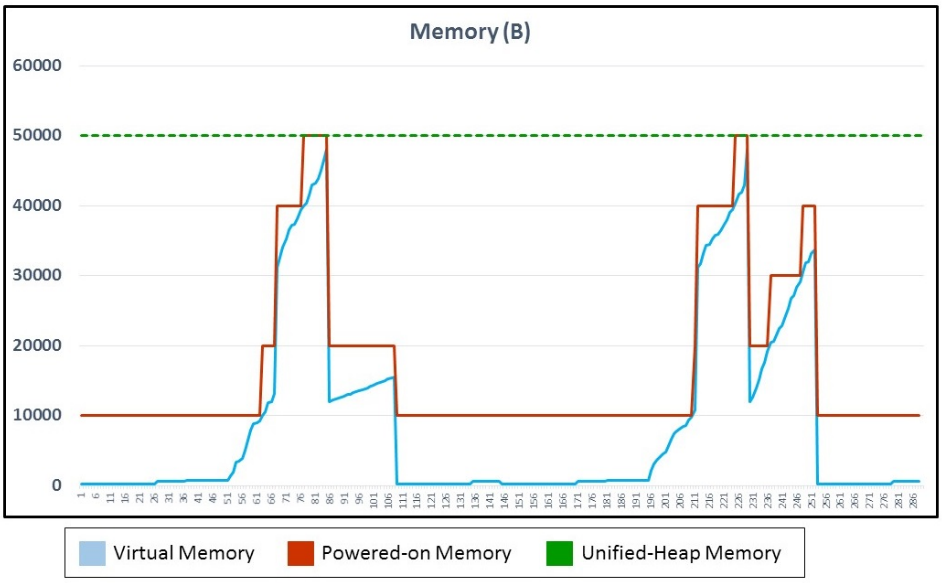


Figure 3.4: Heap Partitioning Energy profile

In this particular application, only a small amount of virtual memory is being used most of the time, while the rest of the memory is being unused, but leaking. By power gating the unused regions of the memory, around 80% of the static energy consumption is saved. When, at certain points, more memory is required by the processor, more blocks are powered-on by the system. Once the GC has been triggered, live objects are compacted and the unused regions of memory are turned off.

For this reason, the proposed optimization reduces the static energy consumption due to the memory by at least 50% because of the *fromspace* power gating, plus variable savings dependent on the specific application targeted.

[26] gives optimistic results when applying the same concept in a similar environment. However, the overhead and the dependency on the specific system cannot be neglected. For this reason, further work needs to be performed in order to evaluate the energy savings that can be achieved when considering system-specific overheads.

First, a proper application profiling has to be performed. Although the embedded processing applications in [26] suggest that only a certain portion of the heap memory is used at a time, further study has to be done to verify this assumption.

Second, the dependency of the energy consumption on the whole system has to be incorporated. This includes a profiling of the system bus energy consumption, the external memory parameters, etc.

Third, the power gating logic overhead cannot be neglected. As mentioned in the introduction, powering on a block consumes a significant portion of energy. This portion can even overhead the savings of the whole proposal if the frequency of the powering on/off exceeds a limit.

3.4.2 Delay

Delays are introduced when integrating the power gating system on the processor. These delays are produced by the fact that some clock cycles have to be waited for the supply power to be restored in a digital block. However, as suggested in [26], this overhead is negligible.

The integration flow that has been proposed has achieved satisfactory results, as the delay overhead has been mainly due to the powering on of the memory. The combined process of generating the flag and stalling the pipeline has not added almost any overhead to the system in comparison to the amount of clock cycles to wait for the power restoration.

3.4.3 Area

As the control system is simple, the area overhead introduced is negligible, supposing less than 0.1% of the area of the processor.

3.5 Porting Heap Partition to other architectures

In order to port the heap partition and power gating flow to other architecture, some modifications have to be performed.

On the one hand implementing a compacting garbage collector in the targeted processor is needed. As it has been shown, in order to efficiently power-off regions of the memory, fragmentation has to be avoided. Although the Garbage Collector is commonly included in Java environments, not all the Garbage Collectors compact the memory. For this reason, a compacting Garbage Collector has to be implemented first in order to utilize the proposed technique.

On the other hand, a new flow to generate the flags has to be developed by taking into account the specific characteristics of the targeted architecture. In this case, native instructions in Java are compiled into microinstructions detected by a hardware Power Gating block. However, specific architecture features may require to modify the flow, for instance in case there is no free user-defined microinstructions available.

Chapter 4

Conclusion

In this work, two optimizations that aim to reduce the energy consumption due to the memory in processors have been proposed and evaluated.

Firstly, a code compression approach has been implemented. A fully working flow which receives Java compiled code and compresses it has been designed. Furthermore, it automatically generates the required hardware to decompress it on the fly. As a result, an automatic tool that allows the user to reduce the memory size and energy consumption of the desired Java application has been obtained. The results achieved can be replicated by simply rerunning the tool having as an input the desired application.

This flow has been verified in an FPGA. In an automatic fashion, the tool has successfully compiled the Java code, the code has been compressed, the hardware has been generated and synthesized on the FPGA, and the memory has been loaded with the compressed code. After this, the processor has been booted up, and has shown correct behavior, being able to perform the same tasks as the processor without code compression. Finally, a 65nm ASIC has been designed in order to measure the area and power overheads of the decompression hardware, which have been discovered to be negligible.

As a result, around 28% code size reduction and 24% bus codeword transaction reduction at almost no cost have been achieved. For this reason, the code compression approach has been discovered to be promising in reducing the power consumption in processors.

Secondly, a heap partitioning and dynamic power gating protocol has been implemented. As a result, a method that successfully follows the memory utilization profile in powering on and off banks has been obtained. This technique has been capable of a 50% leakage power reduction, plus variable savings depending on the memory footprint profile of the application. The area and power overheads have been measured on a 65nm ASIC and discovered to be negligible.

Chapter 5

Further Work

In this chapter, different suggestions for further work are described.

- A IoT program profiling needs to be performed. As it has been seen in this work, the program specific features can have a large impact on the energy savings, as some parameters such as the optimal memory bank size are dependent on the program profiling.
- In order to increase the compression ratio achieved by the code compression approach, a different hardware decompressor can be implemented. The hybrid compression approach proposed in this work achieved higher CR than the per-byte Huffman, although hasn't been selected to implement due to the limited time available. For this reason, implementing more complex decompressor engines will increase the energy savings and it is suggested as future work.
- In order to obtain exact JOP power consumption numbers, the external memory has to be implemented in hardware. In this work, external memory power consumption has been estimated assuming a linear relation between memory size and power consumption. However, exact power numbers have to be obtained.
- The power gating circuits have to be implemented in order to measure the real overhead introduced by this scheme. Dynamic power consumption due to high frequency powering on and off can overhead the savings proposed. For this reason, the power overhead introduced by the gating circuits has to be studied.

References

[1] Nam Sung Kim, Todd Austin, David Blaauw, Trevor Mudge, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir, Vijaykrishan Narayanan, "Leakage Current: Moore´s Law Meets Static Power," *IEEE Computer*, vol 36, pp. 68-75, 2003

[2] Siva G. Narendra, Anatha Chandrakasan,
"Leakage in Nanometer CMOS Technologies," *Springer US*, pp 7-8, 2006

[3] Nikhil Jayakumar, Suganth Paul, Rajesh Garg, Kanupriya Gulati, Sunil P. Khatri ,
"Minimizing and exploiting leakage in VLSI design," *Springer US*, 2010

[4] Anatha P. Chandrakasan, Robert W. Brodersen,
"Minimizing Power Consumption in Difital CMOS Circuits," *Proceedings of the IEEE*, vol 83, no 4, April 1995

[5] Kao, J.T., Chandrakasan, A.P.,
"Dual-Threshold Voltage Techniques for Low-Power Digital Circuits," *IEEE Journal of Solid-State Circuits*, vol 35, pp 1009-1018, 2000

[6] A. Keshavarzi et al.,
"Effectiveness of reverse body bias for leakage control in scaled dual Vt CMOS ICs," *Low Power Electronics and Design, International Symposium on*, pp 207-212, 2001

[7] A. Kaivan Karimi, Gary Atkinson,
"What the Internet of Things (IoT)) Needs To Become a Reality," *Freescale and ARM White Paper*, June 2013

[8] Harad Bauer, Mark Patel, Jan Veira,
"Internet of Things: Opportunities and challenges for semiconductor companies," *McKinsey&Company article*, October 2015

[9] Xiaosen Liu, Edgar Sánchez-Sinencio,
"An 86% Efficiency 12 muW Self-Sustaining PV Energy Harvesting System With Hysteresis Resulation and Time-Domani MPPT for IOT Smart Nodes," *IEEE Journal of Solid-State Circuits*, vol 50, no 6, June 2015

[10] Shao-Yi Chien, Wei Kai Chan, Yu-Hsiang Tseng, Chia-Han Lee, V.Srinivasa Somayazulu, Yen-Kuang Chen,
"Distributed Computing in IoT: Systen-on-a-Chip for Smart Cameras as an Example," *The 20th Asia and South Pacific Design Automation Conference*, pp 130-135, January 2015

[11] JOP - Java Optimized Processor Webpage,
`http://www.jopdesign.com`
May 2016

[12] Martin Schöberl,
"JOP: A Java Optimized Processor for Embedded Real-Time Systems," *Technischen Universität Wien, PhD Thesis* January 2005

[13] Martin Schöberl,
"JOP Reference Handbook," 2009

[14] Ning Ma, Zhuo Zou, Zhonghai Lu, Lirong Zheng, Stefan Blixt,
"A Hierarchical Reconfigurable Micro-coded Multi-core Processor for IoT Applications," *9th International Symposium on Reconfigurable and Communication-Centric Systems-On-Chip*, 2014

[15] "The Java Language and Virtual Machine Specifications,"
`http://docs.oracle.com/javase/specs/#237601`

[16] Martin Schöberl,
"SimpCon - a simple and efficient SoC interconnect," *Proceedings of the 15th Austrian Workhop on Microelectronics (Austrochip)* 2007

[17] Ralph Wittig,
"Power-Efficient Machine Learning on Power systems using FPGA Acceleration," *OpenPOWER Summit* 2016

[18] Jacob Ziv, Abraham Lempel
"A universal algorithm for sequential data compression," *IEEE Transactions of Information Theory*, vol 23, no 3, pp 337-343, 1977

[19] T. Welch
"A technique for high-performance data compression," *IEEE Computer*, pp 8-19, 1984

[20] Talal Bonny
"Huffman-based Code Compression Techniques for Embedded Systems," *PhD Thesis*, 2009

[21] Y. Yoshida, B. Song, H. Okuhata, T. Onoye, I. Shirakawa
"An object code compression approach to embedded processors," *International Symposium on Low Power Electronics and Design*, pp 265-268, 1997

[22] S.J. Nam, I. C. Park, C. M. Kyung,
"Improving dictionary-based code compression in VLIW architectures," *IEICE TRansaction on Fundamentals of Electronics, Communications and Computer Sciences*, pp 2318-2324, 199

[23] David A. Huffman,
"A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol 40, issue 9, pp 1098-1101, 1952

[24] Dimitris Saougkos, George Manis, Konstantinos Blekas, Apostolos V. Zarras, "Revisiting Java Bytecode Compression for Embedded and Mobile Computing Environments," *IEEE Transactions On Software Engineering*, 2007

[25] X. Kavousianos, E. Kalligeros, D. Nikolos,
"Multilevel Huffman Coding: An Efficient Test-Data Compression Method for IP Cores," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems,* vol 26, no 6, pp 1070-1083, June 2007

[26] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, M.Wolczko, "Tuning Garbage Collection for Reducing Memory System Energy in an Embedded Java Environment," *ACM Transactions on Embedded Computing Systems,* vol 1, no 1, pp 27-55, November 2002

[27] C.J Cheney,
"A Nonrecursive List Compacting Algorithm," *Communications of the ACM,* vol 3, pp 677-678, November 1970

LUND UNIVERSITY

Series of Master’s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2016-526
http://www.eit.lth.se