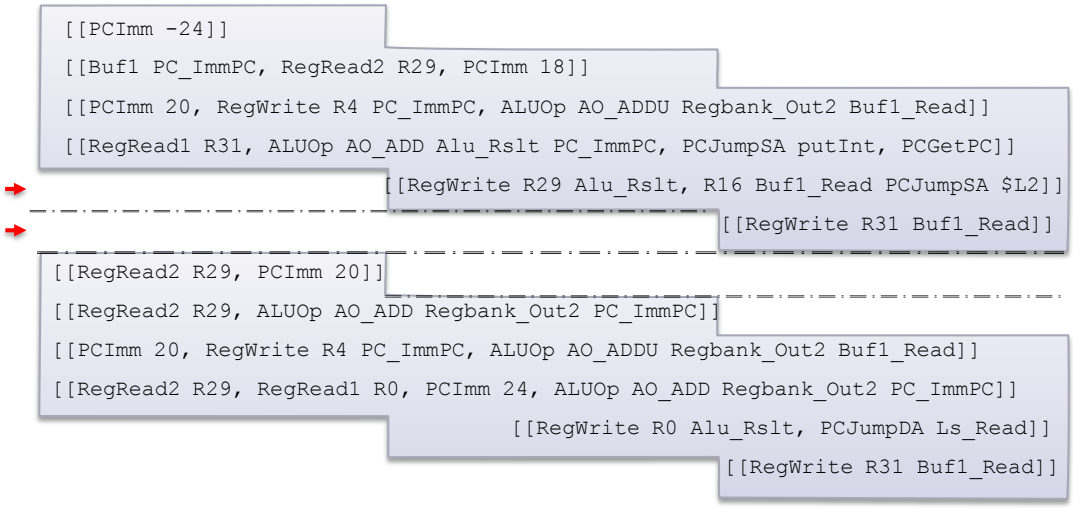
CHALMERS



Microcode Optimization in FlexCore Compiler

THESIS FOR THE MASTERS DEGREE OF COMPUTER SCIENCE ALGORITHMS, LANGUAGES & LOGIC

KASHAN KHURSHID ANSARI

Division of Computer Engineering
Department of Computer Science & Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2012

Microcode Optimization in FlexCore Compiler

Kashan Khurshid Ansari

Department of Computer Science & Engineering
VLSI Research Group

Division of Computer Engineering
Chalmers University of Technology
SE-412 96 GÖTEBORG, Sweden
Phone: +46 (0)31-772 10 00

Author e-mail: `kashan@student.chalmers.se`

Printed by Chalmers Library
Göteborg Sweden, 2012

Microcode Optimization in FlexCore Compiler

Kashan Khurshid Ansari
Division of Computer Engineering, Chalmers University of Technology

ABSTRACT

The aim of this study was to investigate the microcode optimization in the compiler of an embedded processor (FlexCore). The main motivation behind this study was that the compiler was only able to perform front-end compiler optimization, failing to fully harness the processor's potential. This was the problem that was in focus in this study..

This study has lead to a working implementation of filling delay slots optimization in the FlexCore compiler. A framework was used to read the FlexCore machine instruction, provide all the necessary information of each single instruction and then rewrite the optimized instructions. Filling delay slots optimization created redundant instructions, therefore another optimization was done to eliminate redundant instructions after the previous optimization. The optimizations lead to shorter processor execution time and, thus, a reduced energy expenditure.

The simulator executes the FlexCore instructions and generates binary data codes which facilitate in analyzing the processor's performance. Some EEMBC benchmarks are used to evaluate the result of optimization. All the benchmarks give positive results with respect to code size reduction, execution time reduction and energy dissipation. After the optimization the overall performance of FlexCore processor is increased by 11.5%.

Keywords: Microcode, Optimization, FlexCore, Compiler, Embedded, Processor, Delay slots

Acknowledgments

All praises to almighty Allah for the strengths and His blessing for providing me this opportunity and granting me the capability to proceed successfully. I am grateful to the following people for what they have done for me, for my career, and for this thesis.

- Per Larsson Edefors for his supervision and constant support. His inestimable help of constructive comments and suggestions throughout the thesis works have contributed to the success of this research.
- David Whalley for his supervision and allow to use his framework, which allow to more focus on study. His Support and knowledge make it possible to use his framework in this research.
- Magnus Själander for his supervision and describing the clear picture of problem and his support and knowledge regarding this topic.
- I would like to express my appreciation to all my colleagues and friends for their kindness, technical and moral support during my study.
- Last but not the least, I would like to thank my family members especially my parents for always encouraging and believing in me.

Kashan Khurshid Ansari
Göteborg, March 2012

Contents

List of Figures

List of Tables

1 Introduction

Compiler optimization is the process of altering the code of a software program, without affecting the result of the execution, in order to replace expensive instructions with less expensive instructions or rearrange the instructions so that the resources are better utilized. One objective of compiler optimization is to reduce the execution time of a program. This is because of intense use of embedded systems in today's world, mainly on mobile devices in which power consumption is the main issue. A very common example is smart phones that face the problem of battery drain. By performing optimizations, the compiler can generate code that yields the same result but at a lower power expenditure [1].

1.1 Background

Today's life is surrounded by various types of embedded systems. From consumer electronics like mobile phones, digital cameras, DVD players and printers to household appliances such as washing machinery and microwave ovens are based on embedded systems. Also transportation systems from flight to automobiles use embedded systems. In order to meet customer demands there is a trend nowadays towards higher performance and lower power dissipation in the embedded systems which happens mainly in mobile devises where power is the main issue. This also includes small area design with similar programming capabilities of General Purpose Processors, GPPs.

FlexCore is an embedded processor, designed by the VLSI research group of Chalmers, which has all the functionalities of conventional five-stage 32-bit GPP. Based on an evaluation using EEMBC benchmarks, the FlexCore processor's datapath was shown to be 40 percent more efficient as compare to MIPS when counting execution cycles [2].

The machine instructions of FlexCore are represented in a Register Transfer Notation (RTN) format. RTN instructions are very similar to MIPS assembly instructions or SPARC instructions. The only difference is that RTN instruction line can have more than one effect (RTN instruction) per line, which is because of the structure of FlexCore, where we can perform simultaneous operations. Therefore, more than one effect can be possible in an RTN instruction line. Figure 1.1 depicts an example of a basic block of RTN instructions.

```
main:

1     rtn   [[PCImm -24]]
2     rtn   [[Buf1 PC_ImmPC, RegRead2 R29, PCImm 18]]
3     rtn   [[PCImm 20, RegWrite R4 PC_ImmPC, ALUOp AO_ADDU Regbank_Out2 Buf1_Read]]
4     rtn   [[RegRead1 R31, ALUOp AO_ADD Alu_Rslt PC_ImmPC, PCJumpSA putInt, PCGetPC]]
5     rtn   [[RegWrite R29 Alu_Rslt, PCJumpDA Ls_Read]]
6     rtn   [[RegWrite R31 Buf1_Read]]
```

Figure 1.1: Basic block of a FlexCore instructions.

After analyzing the code of a basic block, we see that the first line of instructions only contains a read operation, while the second line contains two read operations. From the third line the system is fully exposed. This means that on the first and second line we are not utilizing all resources. The same case occurs on the fifth line, where we only have execute and write instructions. And on the sixth line we only have a write instruction. So in general the first two lines and the last two lines of a basic block are not utilizing all the datapath resources. In other words we can say that we have unused slots in the first two and last two lines of a basic block.

1.2 Objective

The objective of the thesis is to perform FlexCore compiler optimization, so that the compiler can produce the same output while having a fully exposed system. One way to achieve this goal is to perform optimization on RTN instructions in such a way that we are able to reduce the number of unused slots (as discussed above) as much as possible, to utilize all the resources. The current FlexCore compiler can only perform front-end compiler optimization, so a back-end compiler optimization is needed to fill the unused slots.

The unused slot arises because of the structure of the FlexCore processor, as it can perform read, ALU and write operations simultaneously. Think of the pipeline as if it just has been started directly before the

basic block and that it finishes its execution directly after the basic block.

As the pipeline has just started there is no state in the pipeline and to get any data to work on we first need to read that data into the pipeline. We start by reading it from the register file and/or inputting it through the immediate. There is no point of doing any ALU operations or writing to the memory or register file, as we don't have any data to do any computations on or address (and data) to read (or write).

When finishing the basic block the computation is to finish. So the last useful thing that can be done is to write data back to the register file or memory. There is no point in doing any ALU operations as the result cannot be taken care of by storing it. The same thing with register reads as the values will not be used for anything. The reason for all this is that the pipeline does not handle the transitions between basic blocks, so the easiest way to schedule for the compiler is as described above.

Performing the optimization of filling unused slots will lead to a fully exposed system and also speed up the processor, which as a consequence leads to more efficient processing.

1.3 Motivation

Before performing any optimizations, we not only need more information about blocks but each single instruction as well. We need a clear picture of the control flow graph, so that we know which block is going to execute next or which block is the predecessor of the current block. When we have information about each instruction, we can find out when a block is going to end on a jump instruction, branch instruction or the end of function. Also we need to find the loops within the functions and which block is the loop header.

After gathering all this information we can perform optimization and try to move one or two lines of RTN effects of current blocks to a predecessor block, when certain conditions are satisfied. The EEMBC benchmark will be used to evaluate the efficiency of these optimizations.

1.4 Outline

In the next chapter we are going to discuss FlexCore, how the architecture of FlexCore differs from MIPS, how scheduling takes place, and why FlexCore is more efficient than a MIPS. In chapter three we will present an overview of FlexCore machine instructions that are represented by RTN instructions. We will also analyse the difference between MIPS instructions and FlexCore instructions. In chapter four we will describe the proposed solution and the methodology that we use to implement the optimization. Later, the fifth chapter will conclude with the simulation results. For simulation we will use a previously implemented simulator and the EEMBC benchmarks [10] that are used to evaluate the results of applying optimizations. The last chapter will contain conclusion and future work.

2

FlexCore

2.1 FlexCore

In this thesis we attempt to optimize code generated for the FlexCore processor that is based on Flexible System on Chip (FlexSoC) architecture, which is an on-going research project of the VLSI research group at Chalmers University of Technology. The first question which arises here is why do we need another processor when there are already very high speed and efficient GPP processors? Embedded systems designs such as digital consumer electronics, automotive, smart cards and mobile devices, require high performance and more functionality in combination with stringent energy requirements, which makes general purpose processors unsuitable for embedded systems [2].

The main objective of the Flexible System on Chip (FlexSoC) research is to merge the efficiency of special-purpose hardware and the flexibility of programming a GPP. In other words, a processor whose efficiency is like special-purpose hardware combined with the ability of programming similar to GPP, will be efficient in both processing and energy consumption. This approach allows us to handle diverse processor architecture requirements in a similar manner [3]

2.2 Architecture

GPPs have a fixed instruction set architecture (ISA), that gives a hardware/software interface for all applications. Using this ISA with a single interface on various embedded processors may lead to a large number of instructions, which may require high memory and instruction bandwidth. On the other hand a fixed ISA does not suit well with heterogeneous processors, as it is only able to control some blocks directly, while the remaining are controlled indirectly.

FlexSoC reduces these inefficiencies, by using a different hardware/software interface concept. This can be achieved by placing integrated hardware that can speed up the processing with a datapath similar to a GPP. The ISA's that are present in traditional GPPs are not used in FlexCore, in that sense FlexCore has a native ISA (N-ISA). This N-ISA can fully control all resources.

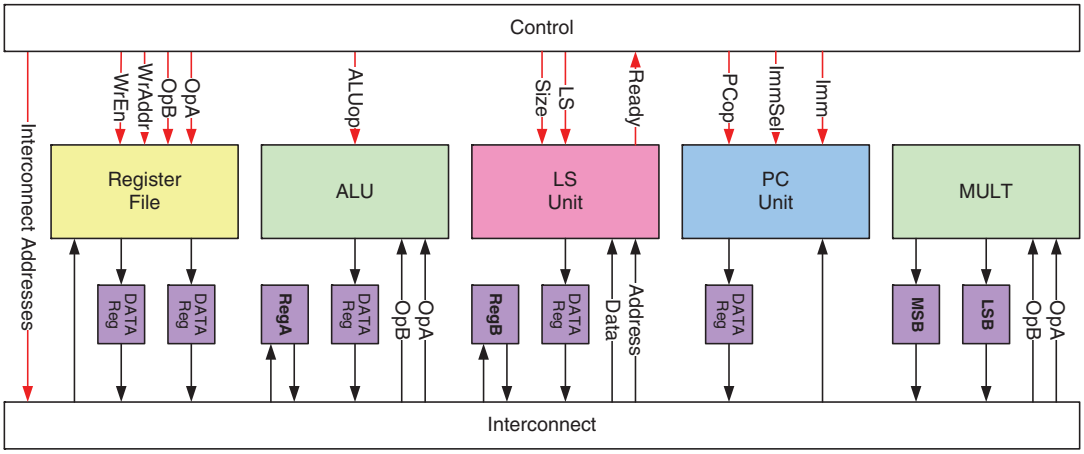


Figure 2.1: Illustration of a Flexcore Processor.

The datapath units used in FlexCore to provide full GPP programmability consist of a program counter (PC), a load/store (LS), a register file (RF), an arithmetic and logic unit (ALU) and a multiplier (MULT) (see Figure 2.1) to perform multiplication operation in one cycle. Two buffers are available to store or load data for datapath units rather than RF or Memory to provide more efficiency [2].

2.3 Scheduling

In traditional GPPs, a fixed ISA provides a hardware/software interface that is same for all applications, where an instruction proceeds through a set of pipeline stages over several clock cycles. Moreover each instruction is always executed in the same way, regardless of prior and subsequent instructions. Allowing data to flow through a datapath along all possible routes would require excessive amounts of logic circuits,

and lead to an extremely complex processor design. A modern conventional ISA therefore imposes strict limitations on how resources of a datapath implementing the ISA can be utilized.

The FlexCore processor removes the conventional fixed ISA to offer a more fine grained and greater control of the datapath resources. An advantage of FlexCore processor is that the FlexCore instructions control the entire FlexCore datapath and resources. All these control signals require very powerful scheduling, since the interaction between different instructions must be handled by the compiler. Another advantage is that the ISA is not fixed, therefore it is easy to add new datapath units. It is important in some embedded system to have the ability to add new datapath units [4]

3

FlexCore RTN Instructions

3.1 Register Transfer Notation

A FlexCore machine or assembly instruction can be divided down into small effects, which are normally called microinstructions. These microinstructions can be represented using a register transfer notation (RTN). From the name RTN instruction, one can understand that it is actually the representation of movement of data from one datapath register to another. To enable movements, all the registers of the processor must be connected. Figure 2.1 shows that all the datapath units and the registers are connected.

All FlexCore datapath units are connected through an interconnect and controlled by the N-ISA control word, see Sec 2.2. The length of the N-ISA control word is 91 bits for the baseline FlexCore configuration in Figure 3.1 and the distribution of bits is as follows: interconnect 24 bits, PC 37 bits (of which 32 bits are immediate), data buffers 2 bits, load/store 5 bits, ALU 5 bits, and register file 18 bits [2].

Figure 3.1: FlexCore N-ISA control world

The switchbox for the baseline FlexCore configuration in Figure 3.1 is connected to datapath units through data input ports, so the units receive input from the switchbox and also the data is coming from output ports to reuse the calculated results. Also each data output port is connected to the data register file so that FlexCore can act as a general purpose processor (GPP), where the data register will play the role of pipeline registers.

3.2 Datapath Unit Attributes

There are three types of ports namely, control in ports, data input ports and data output ports. Each type of port can be viewed according to each datapath unit.

3.2.1 Control in ports

PC unit handles immediate value (PCImm) and signal selects (PC_ImmSel). PC operations (pc_Ops) with the possible operations are JumpSA, JumpSR,JumpDA, BEQZR BNEZR, BEQZA, BNEZA.

LS consists of Load/Store width (ls_width) and load/Store operation (ls_Op). The four types of widths are LSW_1, LSW_2, LSW_3, LSW_4 and ls_Op consists of WRITE, READ, READU.

RF consists of two registers, where one contains the address to write and the other is a flag to enable write.

ALU operations (ALU_Op) consist of Add operator (AO_ADD, AO_ADDU), Subtract operator(AO_SUB, AO_SUBU), And operator(AO_AND), OR operator(AO_OR), Xor operator(AO_XOR), Shift left logical(AO_SLL), Shift right logical (AO_SRL), Shift right arithmetic (AO_SHR) and Set on less than (AO_SLT)

MULT has no control in port.

3.2.2 Data input port

PC has PC_FB

LS consists of the address (LS_Address) and the data to be written (LS_Write).

RF has no input port.

ALU unit has two inputs (ALU_OpA, ALU_OpB) to perform ALU operation.

MULT unit also has two inputs (ALU_OpA, ALU_OpB) to perform the multiplication operation.

Buffers two buffers receive input as well (Buf1, Buf2).

3.2.3 Data output port

PC gives the next current Immediate value on the output ports, or points the address to the next instruction.

LS gives the data on the output port (LS_Read), that is available for being read.

RF gives two outputs in the form of two data register (RegBank_Out1, RegBank_Out2).

ALU gives the result after performing its operation. (ALU_Rslt).

MULT gives two outputs that are least significant bits and most significant bits (MULT_LSW,MULT_MSW).

Buffers two buffers are available to read (Buf1_Read,Buf2_Read).

The control in port and data input port will be part of the RTN instruction and normally data output ports will be the arguments of RTN instructions. 32 registers (R0 - R31) may also be parameters of instructions. Table 3.1 illustrates all three types of ports.

Table 3.1: All the possible ports of a FlexCore processor

Units	Control in Ports	Data	
		Input Ports	Output Ports
PC	PCImm	pc_FB	pc_ImmPC
	pc_ImmSel		
	pc_Ops		
LS	ls_Op	ls_Address	ls_Read
	ls_Width	ls_Write	
RF	Reg_Add_1		regbank_Out1
	Reg_Add_2		regbank_Out2
	write_Add		
	Write_Enable		
ALU	ALUOp	alu_OpA	alu_Rslt
		alu_OpB	
MULT		mult_OpA	mult_LSW
		mult_OpB	mult_MSW
		Buf1	buf1_Read
		Buf2	buf2_Read

3.3 RTN Instructions

All the possible RTN instructions and their arguments are first described. The arguments of an instruction may contain registers (R), any ports that we discussed above (P), ALU operation (A) and Integers (I). Table 3.2 shows all RTN instructions. Jump and branch can be handled from the PC unit, ALUOp handles all the operations that are going to be performed in the ALU unit. For example ALUOp takes three arguments i.e. (A P P) where A is the identifier of which ALU operation needs to be performed and the two P's can be any values in the form of ports or registers.

Table 3.2: All the possible Machine Instructions of RTN

Sections	Instructions	Arguments
Program Counter	PCImm	I
	PCImm2	I
	PCGetPC	
	PCJumpSA	I
	PCJumpSR	I
	PCJumpDA	P
	PCBEQZR	I,P
	PCBNEZR	I,P
	PCBEQZA	I,P
	PCBNEZA	I,P
Register bank	RegRead1	R
	RegRead2	R
	RegWrite	R,P
Arithmetic Logic Unit	ALUOp	A,P,P
	ALU2Op	A,P,P
	AGUOp	P,P
Load/ Store	LSWrite	L,P,P
	LSRead	L,P
	LSReadU	L,P
Buffers	Buf1	P
	Buf2	P
Multiplexer	Mult	P,P
	MultRegWrit	
Stalls	StallReg1	
	StallReg2	
	StallReg3	
	StallReg4	
	StallALU	
	StallLS	

3.4 Comparison between MIPS and Flexcore Instructions

In this section a comparison of MIPS instructions will be done with RTN instructions (see Table 3.3). Some ALU instructions have two FlexCore RTN instructions against one MIPS instruction, where one instruction is for Register and the other is for Integer. Therefore one add instruction has two RTN instructions, ALUr and ALUi, both performing the add operation but the arguments differ.

Table 3.3: The Comparison between the MIPS and RTN Instructions.

Instructions		
Mips	**rtn**	
add	ALUr AO_ADD	/ ALUi AO_ADD
addu	ALUr AO_ADDU	/ ALUi AO_ADDU
addiu	ALUr AO_ADDU	/ ALUi AO_ADDU
and	ALUr AO_AND	
andi	ALUi AO_AND	
b	J	
beq	BrALUr NZ AO_SEQ	
beqz	Br Z	
bgez	BrALUr NZ AO_SLE Reg	
bgtz	BrALUr NZ AO_SLT Reg 0	
blez	BrALUr NZ AO_SLE	
bltz	BrALUr NZ AO_SLT	
bnez	Br NZ	
bne	BrALUr NZ AO_SNE	
j	J	JR
jr	JR	
jal	JAL Reg 31	
jalr	JALR Reg 31	
la	ALUi AO_ADD	
li	ALUi AO_OR	
lb	Load LSW_1 Signed	
lbu	Load LSW_1 Unsigned	
lh	Load LSW_2 Signed	
lhu	Load LSW_2 Unsigned	
lui	ALUi AO_ADD d Reg 0	
lw	Load LSW_4 Signed	
mflo	MFLo	
move	ALUr AO_ADD d s Reg 0	
mul	Mult3	
mult	MultMIPS	

Instructions		
Mips	**rtn**	
sb	Store LSW_1	
sh	Store LSW_2	
seq	ALUr AO_SEQ	/ ALUi AO_SEQ
sll	ALUr AO_SLL	/ ALUi AO_SLL
sne	ALUi AO_SNE	/ ALUr AO_SNE
sra	ALUi AO_SHR	/ ALUr AO_SHR
srl	ALUi AO_SRL	/ ALUr AO_SRL
slt	ALUi AO_SLT	/ ALUr AO_SLT
sltu	ALUi AO_SLT	/ ALUr AO_SLT
sub	ALUi AO_SUB	/ ALUr AO_SUB
subu	ALUi AO_SUBU	/ ALUr AO_SUBU
sw	Store LSW_4	
xor	ALUr AO_XOR	
xori	ALUi AO_XOR	
neg	ALUr AO_SUB d Reg 0	
nop	UserNOP ""	
nor	ALUr AO_NOR	
or	ALUr AO_OR	
ori	ALUi AO_OR	

3.5 Example

This section demonstrates how scheduling is performed on a MIPS and on a FlexCore. For the example shown in Figure 3.2 MIPS requires six cycles, but FlexCore requires only four cycles. The three instructions (see Figure 3.2) can also be scheduled on FlexCore (see Figure 3.3b).

1. The latency of instructions is three cycles instead of four.
2. The add operations are performed in 2 cycles.
3. Register $1 is only going to be written once, and the write port is available for instructions
4. The effect of the load word instruction is zero, so there is no need to calculate a new address [2].

```
ADD   $1, $3, 16
ADD   $1, $5, $1
LW    $3, 0($9)
```

Figure 3.2: Example of three consecutive GPP assembly instructions

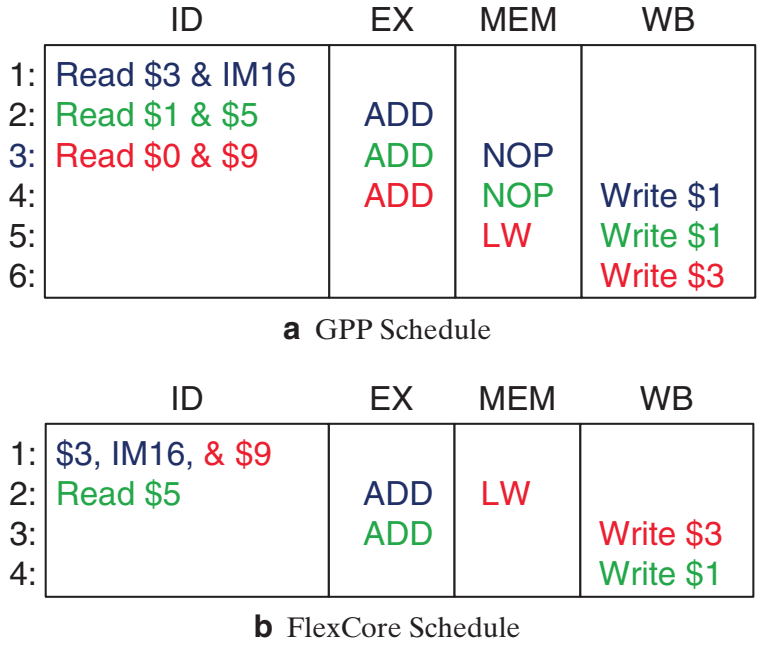


Figure 3.3: Instruction scheduling on a GPP and FlexCore datapath

4

Microcode Optimization

4.1 Problem Statement

Before going into the problem description, let's make a quick review of the FlexCore machine instructions which are also known as RTN instructions discussed in the previous chapter. Unlike MIPS and SPARC instructions, RTN have multiple effects per line, due to the architecture of FlexCore, with multiple datapath units. Therefore, FlexCore is capable of simultaneously executing these effects, as seen in Figure 4.1, which shows a basic block of RTN instructions.

```
main:

1       rtn     [[PCImm -24]]
2       rtn     [[Buf1 PC_ImmPC, RegRead2 R29, PCImm 18]]
3       rtn     [[PCImm 20, RegWrite R4 PC_ImmPC, ALUOp AO_ADDU Regbank_Out2 Buf1_Read]]
4       rtn     [[RegRead1 R31, ALUOp AO_ADD Alu_Rslt PC_ImmPC, PCJumpSA putInt, PCGetPC]]
5       rtn     [[RegWrite R29 Alu_Rslt, PCJumpDA Ls_Read]]
6       rtn     [[RegWrite R31 Buf1_Read]]
```

Figure 4.1: Basic block of FlexCore instructions.

In the first line of the basic block there is only a read instruction, while the second line has two read instructions. The third line contains read, execute and write instructions and at this stage the system is fully exposed. On the fourth line the system is still fully exposed, while the fifth line only contains execute and write instructions. The sixth line only has a write instruction. For a better explanation we can arrange the instructions according to their major types (see Figure 4.2).

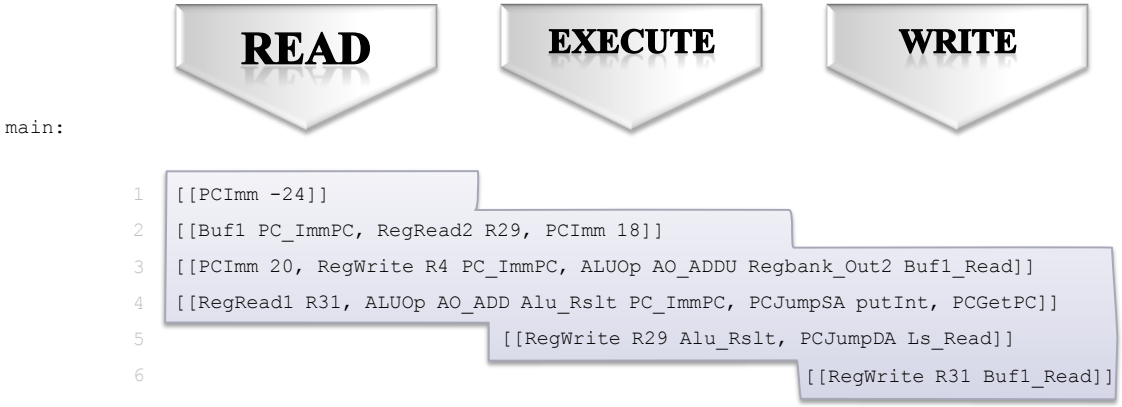


Figure 4.2: Basic block of FlexCore instructions.

After analysing this RTN basic block, one can notice that the first two lines and the last two lines of a basic block of RTN are not fully exposed. If two consecutive basic blocks are executed one after another, the situation illustrated in Figure 4.3 appears.

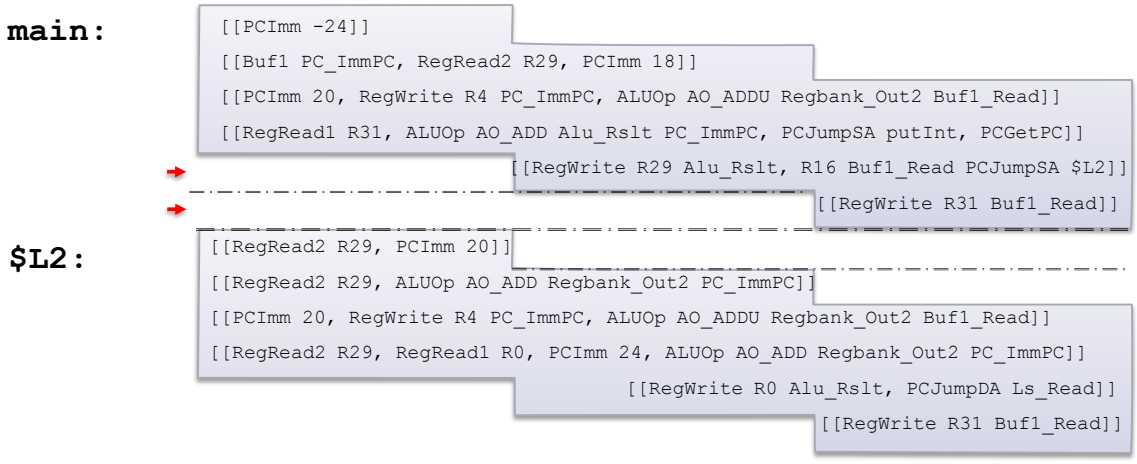


Figure 4.3: Two consecutive basic blocks

By arranging the instructions according to their major types, one can notice that there is a delay of two slots. This occurs every time when there is a transition from one basic block to another basic block. Therefore an optimization must be performed in such a way that there is a minimum number of delay slots, so that the system can be fully exposed as much as possible [5].

4.2 Proposed Solution

Whenever the control is transferred from one basic block to another, there are always two delay slots. A possible solution could following: Assume there are two RTN basic block A and B, where one is executing after another, or in other words the control is going to transfer from basic block A to basic block B (see Figure 4.4). Then we can try to move the first two lines of block B to the end of block A. By performing this

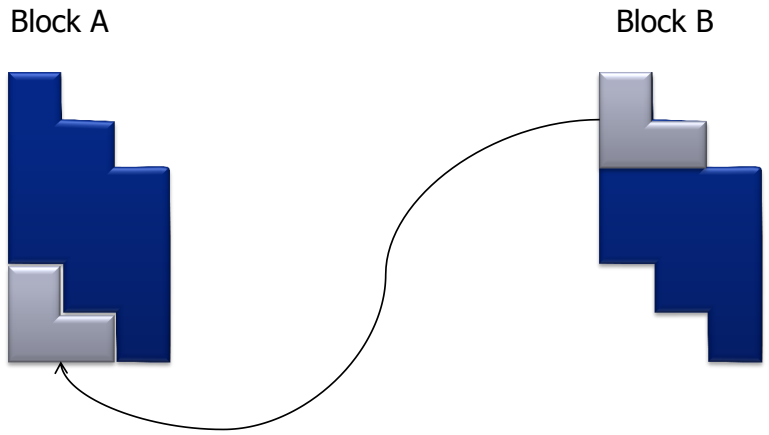


Figure 4.4: Graphical representation of basic blocks

optimization, the system will be fully exposed at the end of block A as well as at the beginning of block B. We also know that the sequence of occurrence of two basic blocks is not always the sequence of execution between two basic blocks.

Consider Figure 4.5, where block 2 is also executing after block 3 and from block 2 we can go to block 4 as well. To perform the optimization we need extensive information about the control flow, such as which block can execute after the current block, as well as the type of control transfer, which can be either a branch, jump, or a simple fall through. We also need to discover loops, including which block is the header of the loop and which one is a backedge within a function and also to notice if the function contains nested loops.

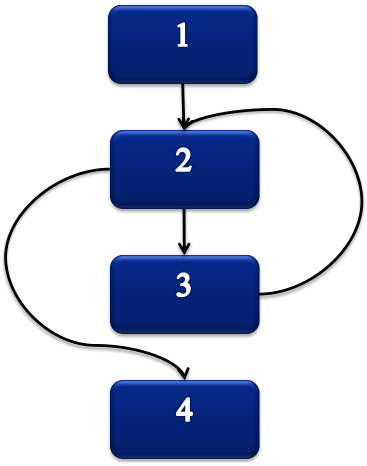


Figure 4.5: Control flow of basic blocks

4.3 Framework

To be able to perform the optimization, we need to collect all the information from the control flow of basic blocks to the information of individual instructions. This is done in order to be well informed about what are the types of instructions, including which registers are used and which are set. An extensive analysis is required for writing code to gain all this information. An alternative is that we can use available frameworks that could help us to collect all the information, then we can spend our energy on formulating the optimization.

In this study David Whalley's framework is chosen to collect all the necessary information we need [6]. It is developed for SPARC assembly language, written in C and it has ability to read SPARC instruction files. This framework gathers all the information, not only the control flow of basic blocks, meaning the sequence of blocks, but also detailed information of each instruction, like what is the type of instruction and which variables or registers are used or set in this instruction.

This framework provides flexibility within the instructions to parse any instruction. The framework can change variable, register or constant values, and can even delete instruction(s) as well. It can also create new instructions and insert an instruction at the desired position. In order for the framework to process RTN instructions, we accomplished a number of transformation within the framework. Some major changes are:

1. The first challenge is to modify the framework to read RTN instructions. We modified the framework so it can read RTN instructions line by line and can write them out after optimization.

2. We can identify the name of a function and also when a function/block starts and when it ends. Also, we can identify the sequence of occurrence as well as the sequence of execution of blocks.

3. We read each instruction line and gather the control flow of basic blocks. Either a block ends with fall

through or the block has a jump or a branch at the end (see Figure 4.6). This information is more difficult to track, because RTN has multiple instructions per line.

4. We track instruction lines within the basic block. Doing this can give information on the next and previous line and can also give the first and last line of a basic block in order to traverse within a function.

5. We parse each effect within an instruction line. A single line can have multiple effects, and all the effects differ from SPARC instructions.

6. We gather detail information on an effect, which includes the type of effect, which registers are set and which registers are used.

7. We parse the effects and are able to create multiple effects per instruction line.

4.4 Control Flow Information

The framework provides the control flow graph of basic blocks of RTN instructions. We not only have the information of the next and previous blocks from the current block, but we also have information on predecessor and successor blocks. We are maintaining two flows of basic blocks at a same time.

One is the order of their occurrence and the other is the order of their execution by predecessor and successor blocks. The first order is used to dump the blocks in a file according to their appearance. The second order is actually used to set up the control flow graph. A control flow graph example is given in Figure 4.6 . In this example, block 1 has a branch to block 3 or has a fall through to block 2, which has a fall through to block 3. Block 3 has a jump to block 8, while block 8 has a branch to block 4 or fall through to block 9. Block 4 has a branch to block 6 or fall through to block 5. Block 5 has a jump to block 7 and block 6 has a fall through to block 7. Block 7 has a fall through to block 8. Here we detect a backedge from block 7 to block 8, which means that we have a loop and that block 8 is a loop header. Now we have a clear picture of the control flow of basic blocks.

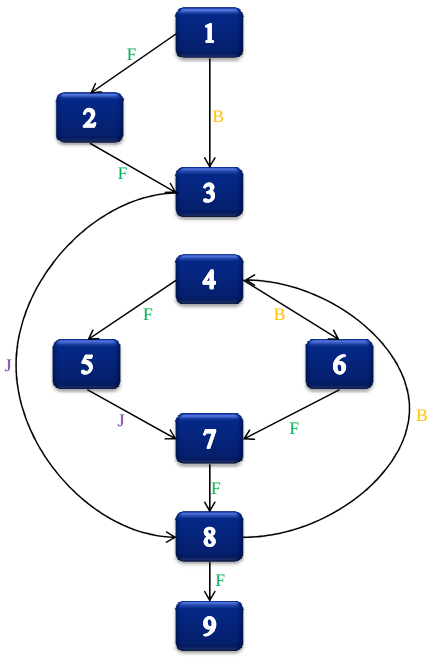


Figure 4.6: Detailed Control flow of basic blocks

4.5 Instructions Information

Unlike the MIPS or SPARC machine instructions, RTN instructions have effects per line, which is because of the FlexCore architecture, that can execute multiple effects per cycle. The framework parses the machine instruction and sets up the flow of instruction lines, gathers information about which line is first and which one is last, and saves the initialization point of traversing within a block. The framework also provides the information on which line is predecessor and which one is successor from the current line, which helps to traverse from one instruction line to another. Figure 4.7 depicts an example. Within a basic block, the

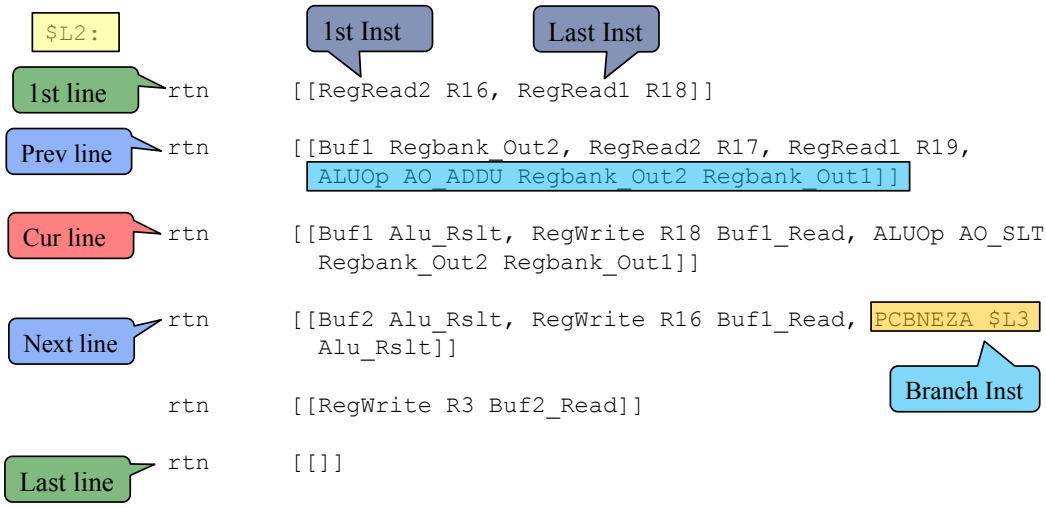


Figure 4.7: Detailed information of Instruction lines and Instructions

framework can identify the label of a basic block, each instruction line, and within instruction can identify each effect and can traverse within a line. The framework can also provide detailed information of each effect like registers uses and sets and also the type of effect. In the above example we can see in the second line, the last effect type is ALU and in the fourth line the type of the last effect is a branch.

4.6 Information That the Framework Provides

The framework provides extensive information to analyse basic blocks and helps performing the optimization. This information doesn't impact the output, because it is just an extra information in the form of comments by using # sign. An example is given in Figure 4.8. The framework shows the loops within a function, if they exist. It also shows the loop header and the blocks that comprise the loops. In the example we have a loop, its header is block 3 and blocks comprising the loop are block 3 and 4. The framework assigns a unique number to identify each block (for the block in the example, the framework assigns the

Fibonacci

```
# loops in function
#   loop: head = 3
#         blocks = 3 4
.
.
.
# block 4
# preds: 3
# succs: 5 3
#  doms: 1 2 3 4
#   ins=R2:R3:R4:R31:
#  outs=R2:R3:R4:R16:R17:R18:R19:R31:#

$L2:
          rtn       [[RegRead2 R16, RegRead1 R18]]
          rtn       [[Buf1 Regbank_Out2, RegRead2 R17, RegRead1 R19, ALUOp AO_ADDU Regbank_Out2
                      Regbank_Out1]]
          rtn       [[Buf1 Alu_Rslt, RegWrite R18 Buf1_Read, ALUOp AO_SLT Regbank_Out2 Regbank_Out1]]
          rtn       [[Buf2 Alu_Rslt, RegWrite R16 Buf1_Read, PCBNEZA $L3 Alu_Rslt]]
          rtn       [[RegWrite R3 Buf2_Read]]
          rtn       [[]]
          .
          .
          .
```

Figure 4.8: The information of RTN Instructions that the Framework is providing.

number 4) also which block(s) are predecessor (preds: 3) and which block(s) are successors (succs: 5 3). The framework provides information about dominating blocks, that is which blocks(s) are dominating the current block (doms: 1 2 3 4), this information is used to detect loops and live registers and variables. A number of live registers enter this block from its predecessor (ins = R2:R3:R4:R31) and a number of live registers leave this block to its successors (outs = R2:R3:R4:R16:R17:R18:R19:R31).

4.7 Cases

A basic block other than the last block of a function will end with one of the following cases.

i. Fall through

ii. Jump Instruction

iii. Branch Instruction

To be able to implement optimization, a detailed analysis is required on all the cases. Let's assess each case with respect to the possibilities of optimization for each individual case.

i. **Fall through**

The simplest case is fall through, which can be described as if there are two consecutive basic blocks and one executing after another. Figure 4.9 shows an example of fall through.
Here we have two consecutive basic blocks (1 & 2). Block 2 is executing after the execution of block 1, so there is a fall through from block 1 to block 2. In this case we can try to move the first two lines of block 2 to the end of block 1, so that the system can be fully exposed at the end of block 1 as well as at the beginning of block 2.

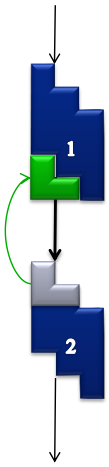


Figure 4.9: Fall through example.

ii. **Jump Instruction**

When a basic block ends with a jump instruction, then this block has an unconditional jump to another block. For example see Figure 4.10. Block 5 contains an unconditional jump and the jump instruction is from block 5 to block 2. We can move the first line of block 2 to the end of block 5. But one can notice that block 2 has actually two entry points which are from block 1 and from block 5. To be able to delete the first line of block 2 and move it to the end of block 5, we have to check if we can also move the first line of block 2 to the end of block 1. If it's feasible then we can try to move the first line of block 2 to the end of block 5 as well as block 1 and then we can be able to delete the first line of block 2.

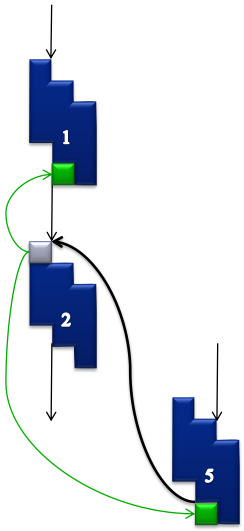


Figure 4.10: Jump case example.

iii. **Branch Instruction**

If a basic block contains a branch instruction then we have a branch at the end of the current block and we must have two successors of the current block. One will be a fall through if the condition is false and the other will be a target. An example is given in Figure 4.11. Here we have a branch instruction at the end of block 3 and block 4 is a fall through and block 5 is the target. Branches are the most difficult case to optimize [7]. For this reason one tries to determine which successor block has more probability to be executed, so that one can try to optimize that one. In the current example, let's say block 4 is within a loop. This means that block 4 will probably execute more often as compared to block 5. So we can try to move the first line of block 4 to the end of block 3 and then delete the first line of block 4 and leave the block 5 as it is.

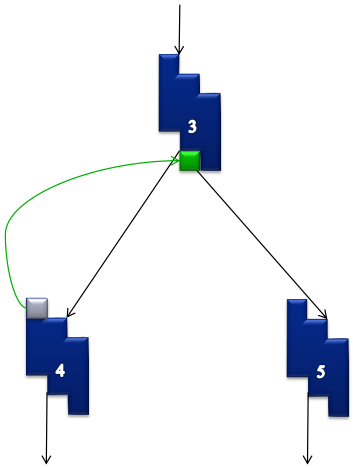


Figure 4.11: Loop case example.

iv. **Loops**

Like other machine instructions, RTN also has loops within functions. So if there is a backedge then there is a loop. An example of loop is given in Figure 4.12 In this example block 4 has a branch to block 3 and block 3 has a fall through to block 4. So block 3 is a loop header and also loop entry point. So we can try to move the first line of block 3 to the end of block 4, but again block 3 has two entry points, so we have to assure that we can also be able to move the first line of block 3 to the end of block 1. Using this assumption, we can try to move the first line of block 3 to the end of block 4 as well as block 1 and afterwards we are able to delete the first line of block 3.

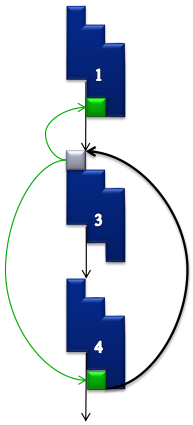


Figure 4.12: Branch case example.

4.8 Cases Priorities

We reviewed all four cases, now we see which case has priority over another. Loops have the most priority over all the other cases, as a loop will be executed the most as compared to others. So first we will perform optimization on loops and then, after that, the remaining cases will be optimized. As there is no contradiction between the remaining cases all these have the same priority. Figure 4.13 illustrates an example.
In this situation, we have a loop comprising block 4 and block 5, so first we try to optimize this loop and after that we can optimize the remaining cases. We can see in the example that the remaining cases have no conflicts with each other and can be optimized normally.

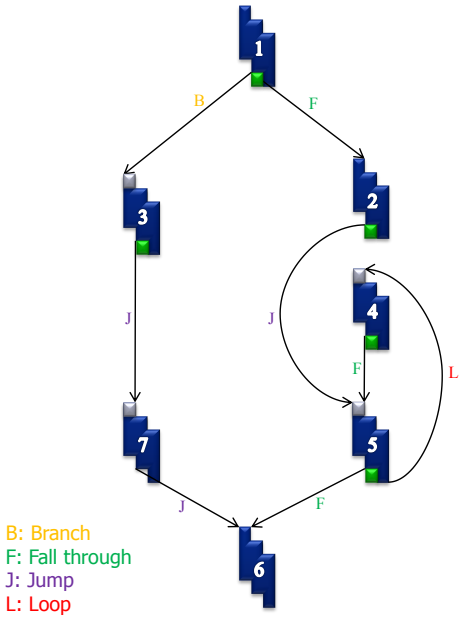


Figure 4.13: Cases Priorities example.

4.9 Before and After Optimization

We examine a portion of a function to see how the code appears before and after the optimization. The autocorrelation (autcor00) of EEMBC is used as example [10]. Figure 4.14 shows the code before performing the optimization. Figure 4.15 shows the code after the optimization. Here block $_6 has a branch to block $L7 and a fall through to block $_7. Block $L7 has a transfer of control instruction in the first line that is, a jump instruction. Since no more than one transfer of control is allowed in a single block, so we cannot move block $L7's first line. But we can move the first line of block $_7 to the end of block $_6. Now, since block $_7 has only one entry point, we can delete the first line of block $_7. since block $_7 has a fall through to block $_21, we can move the first two lines of block $_21 to the end of block $_7, as the instructions of the source lines are not conflicting with the instructions of the destination lines. We cannot move any lines to the end of block $_21 because the successor (block $L7) is not satisfied by their entire predecessor (i.e block $_6 and $_21).

```
$_6:
1           rtn       [[RegRead1 R18, PCImm 144]]
2           rtn       [[ALUOp AO_ADD Regbank_Out1 PC_ImmPC]]
3           rtn       [[RegRead2 R0, LSRead LSW_4 Alu_Rslt]]
4           rtn       [[PCImm %toHi(%hi(input_buf)), ALUOp AO_SEQ Ls_Read Regbank_Out2]]
5           rtn       [[Buf1 PC_ImmPC, PCBNEZA $L7 Alu_Rslt]]
6           rtn       [[RegWrite R2 Buf1_Read]]
$_7:
1           rtn       [[PCImm 9]]
2           rtn       [[PCImm input_buf, RegWrite R17 PC_ImmPC]]
3           rtn       [[RegRead1 R0, RegWrite R20 PC_ImmPC]]
4           rtn       [[PCImm input_buf, RegWrite R16 Regbank_Out1]]
5           rtn       [[RegWrite R4 PC_ImmPC]]
$_21:
1           rtn       [[RegRead2 R19]]
2           rtn       [[PCImm 500, RegWrite R7 Regbank_Out2]]
3           rtn       [[RegRead2 R29, RegRead1 R21, PCImm 16, RegWrite R6 PC_ImmPC]]
4           rtn       [[RegRead1 R17, RegWrite R5 Regbank_Out1, ALUOp AO_ADD Regbank_Out2 PC_ImmPC]]
5           rtn       [[RegWrite R31 PC_ImmPC, LSWrite LSW_4 Alu_Rslt Regbank_Out1]]
6           rtn       [[]]
$L7:
1           rtn       [[PCJumpSA th_signal_finished, PCGetPC]]
2           rtn       [[RegWrite R31 PC_ImmPC]]
3           rtn       [[]]
```

Figure 4.14: A portion of autocor00 benchmark, Before optimization

```
$_6:
1           rtn       [[RegRead1 R18, PCImm 144]]
2           rtn       [[ALUOp AO_ADD Regbank_Out1 PC_ImmPC]]
3           rtn       [[RegRead2 R0, LSRead LSW_4 Alu_Rslt]]
4           rtn       [[PCImm %toHi(%hi(input_buf)), ALUOp AO_SEQ Ls_Read Regbank_Out2]]
5           rtn       [[Buf1 PC_ImmPC, PCBNEZA $L7 Alu_Rslt]]
6           rtn       [[RegWrite R2 Buf1_Read, PCImm 9]]
$_7:
            rtn       [[PCImm 9]]
1           rtn       [[PCImm input_buf, RegWrite R17 PC_ImmPC]]
2           rtn       [[RegRead1 R0, RegWrite R20 PC_ImmPC]]
3           rtn       [[PCImm input_buf, RegWrite R16 Regbank_Out1, RegRead2 R19]]
4           rtn       [[RegWrite R4 PC_ImmPC, PCImm 500, RegWrite R7 Regbank_Out2]]
$_21:
            rtn       [[RegRead2 R19]]
            rtn       [[PCImm 500, RegWrite R7 Regbank_Out2]]
1           rtn       [[RegRead2 R29, RegRead1 R21, PCImm 16, RegWrite R6 PC_ImmPC]]
2           rtn       [[RegRead1 R17, RegWrite R5 Regbank_Out1, ALUOp AO_ADD Regbank_Out2 PC_ImmPC]]
3           rtn       [[RegWrite R31 PC_ImmPC, LSWrite LSW_4 Alu_Rslt Regbank_Out1]]
4           rtn       [[]]
$L7:
1           rtn       [[PCJumpSA th_signal_finished, PCGetPC]]
2           rtn       [[RegWrite R31 PC_ImmPC]]
3           rtn       [[]]
```

Figure 4.15: A portion of autocor00 benchmark, After optimization

4.10 Redundant Instructions after Optimization

After performing the filling delay slots optimization, we noticed that some blocks now have redundant register transfer instructions. This is because at the end of block we write the calculated value in a register to be fetched later and in successor block we are reading that particular register. Moving such lines to the end of predecessor block can actually create redundant register transfer instructions. So in one instruction line we are writing a register and then reading the same register, see the example code in Figure 4.16.

```
$_1:
1           rtn       [[RegRead1 R4]]
2           rtn       [[RegRead1 R0, RegWrite R15 Regbank_Out1]]
3           rtn       [[RegWrite R14 Regbank_Out1]]
$L6:
1           rtn       [[RegRead2 R14, RegRead1 R6]]
2           rtn       [[RegRead1 R0, ALUOp AO_SUBU Regbank_Out1 Regbank_Out2]]
3           rtn       [[RegWrite R12 Alu_Rslt, ALUOp AO_SLE Alu_Rslt Regbank_Out1]]
4           rtn       [[RegRead1 R0, PCBNEZA $L11 Alu_Rslt]]
5           rtn       [[RegWrite R11 Regbank_Out1]]
6           rtn       [[]]
```

(a) Portion of Benchmark autcor00, before optimization

```
$_1:
1           rtn       [[RegRead1 R4]]
2           rtn       [[RegRead1 R0, RegWrite R15 Regbank_Out1]]
3           rtn       [[RegWrite R14 Regbank_Out1, RegRead2 R14, RegRead1 R6]]
$L6:
1           rtn       [[RegRead1 R0, ALUOp AO_SUBU Regbank_Out1 Regbank_Out2]]
2           rtn       [[RegWrite R12 Alu_Rslt, ALUOp AO_SLE Alu_Rslt Regbank_Out1]]
3           rtn       [[RegRead1 R0, PCBNEZA $L11 Alu_Rslt]]
4           rtn       [[RegWrite R11 Regbank_Out1]]
5           rtn       [[]]
```

(b) Portion of Benchmark autcor00, after optimization

Figure 4.16: Before and after optimization of portion of benchmark autcor00.

Figure 4.16a shows a portion of code before optimization, while Figure 4.16b shows the code after optimization. In the last line of block $_1, we are first writing to register R14 and then we are reading the same register R14 which makes it redundant. A proposed solution is that we can delete the write and read instruction of a register, if the register is completely dead in the block [8]. A heuristic search is required to find out if the register is really dead or not. If so, then it's allowed to delete both instructions. Otherwise

we leave them as they are, because it will give wrong output if deleted, as the register will be used later and reading that register will get wrong value.

Here is a demonstration of elimination of the register transfers instructions that are becoming redundant after optimization. Figure 4.16b shows a portion of code which has redundant instructions that came into being after the delay slot optimization. In the last line of block $_1,register R14 is reading and writing in one instruction line. So we have to check if it is completely dead in the successor block. After analyzing the successor block $L6, it is found that register is not used again in block $L6, which indicates that this register is really dead. Deleting read and writes instruction of this register will not affect the output. Figure 4.18 shows the code after deleting these instructions.

```
$_1:
1           rtn       [[RegRead1 R4]]
2           rtn       [[RegRead1 R0, RegWrite R15 Regbank_Out1]]
3           rtn       [[RegRead1 R6]]
$L6:
1           rtn       [[RegRead1 R0, ALUOp AO_SUBU Regbank_Out1 Regbank_Out2]]
2           rtn       [[RegWrite R12 Alu_Rslt, ALUOp AO_SLE Alu_Rslt Regbank_Out1]]
3           rtn       [[RegRead1 R0, PCBNEZA $L11 Alu_Rslt]]
4           rtn       [[RegWrite R11 Regbank_Out1]]
5           rtn       [[]]
```

Figure 4.17: Portion of code after elimination of redundant instructions.

4.11 Optimizers Output

The optimizer provide a detail information of a input file in the form of optimizers output that shows these information:

- List of Functions in the file.
- Number of instructions in each function.
- Number of optimization performed by fill delay slots phase.
- Number of optimization is performed by redundant instructions elimination phase.
- Total number of optimization performed.

Figure 4.18 shows the output of the optimizer.

```
-bash-3.2$ ./optkk <heap> heap.rtn
function     level instructions memory refs
------------ ----- ------------ -----------
heap_initize total           19           0
heap_alloc   total          108           0
                 0           97           0
                 1           11           0
heap_free    total          134           0
mem_heap_ine total           14           0
i_free       total            5           0
th_free_x    total            5           0
i_malloc     total           43           0
th_malloc_x  total            4           0
------------ ----- ------------ -----------
program      total          332           0
                 0          321           0
                 1           11           0

 42 transformations applied by fill delay slots phase.
  1 transformations applied by redundant instructions elimination
phase.
---
 43 transformations applied by all optimization phases.
```

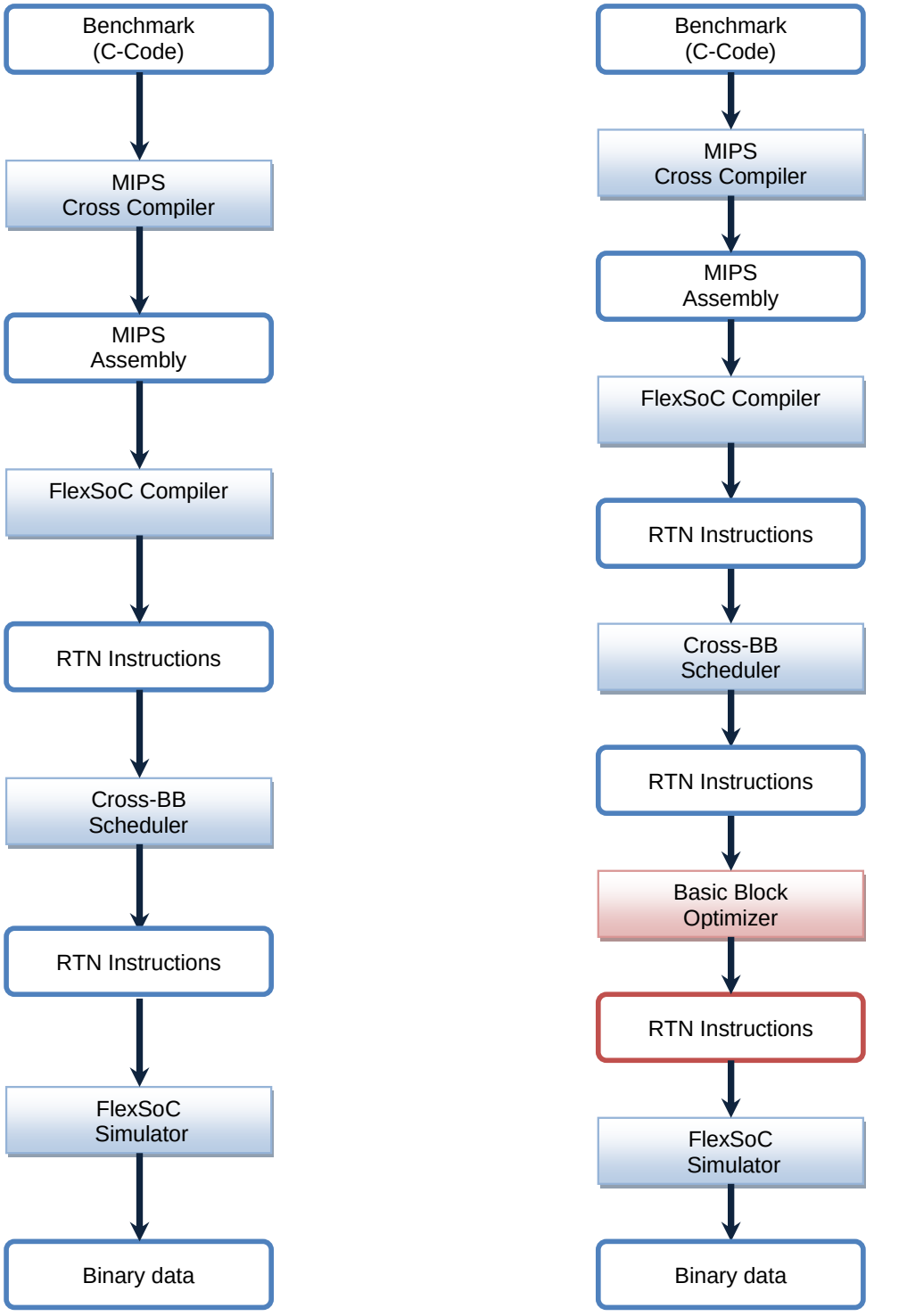
Figure 4.18: Optimizers output after optimization of a file heap.rtn

5 Simulation and Performance Results

5.1 FlexSoC Framework

In order to compile C code on the FlexCore processor, the FlexSoC framework is used. The FlexSoC framework is very extensive and, for example, includes a compiler and a simulator for the FlexCore processor shown in Figure 5.1.

- **Compiler**: To compile C code on FlexCore processor, C code is first converted into the MIPS instruction which is produced by a MIPS cross-compiler. Next, the compiler compiles the MIPS instructions into FlexCore RTN instructions. The output of the compiler thus will be RTN instruction code. These RTN format instructions are used to develop the necessary parallelism of the FlexCore processor.
- **Simulation**: The FlexCore simulator is implemented in Python. The simulator executes FlexCore instructions and helps to trace bugs in the compiler and measure its performance. The simulator is capable of giving simulation cycle count, profiling and simulation trace statistics, run time and average time of execution of a cycle [9]. After executing FlexCore instructions the simulator generates binary data codes, which are used to analyze processor's performance. The simulator can also be configured to a MIPS processor to imitate a GPP. An example of simulator output is shown in Figure 5.2, when benchmark *autcor00* is simulated.



(a) FlexSoc Framework

(b) FlexSoc Framework with optimizer

Figure 5.1: FlexSoC framework before and after the optimizer

```
RUN
Benchmark: eembc-autcor00
Started @ 2012-06-16 12:08:32.852978
Finished @ 2012-06-16 12:11:31.260984
Total execution time: 0:02:58.408006
Average ms/cyc: 1.341091

PROFILING STATS
+ Event: main end @ cycle #133032

BENCHMARK OUTPUT
>>-------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.000
>> EEMBC Member Company     : EEMBC
>> Target Processor         : PC-32bit-X86
>> Target Platform          : PC-Win32
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 0x000003e8
>> Target Timer Granularity : 0x0000000a
>> Recommended Iterations   : 0x00000001
>> Bench Mark               : Autocorrelation Bench Mark V1.0E0
--  Non-Intrusive CRC = 0x0000981bx
--  Iterations        = 0x00000001u
--  Target Duration   = 0x00000000u
--  v1                = 0x00000000
--  v2                = 0x00000000
--  v3                = 0x00000000
--  v4                = 0x00000000
>> DONE!
>> BM: Autocorrelation Bench Mark V1.0E0
>> ID: TEL autcor00
```

Figure 5.2: Simulator output after simulated benchmark autcor00

5.2 Benchmark

EEMBC [10] is used to observe the increased performance after performing the optimization. The benchmark is divided into 3 main suites. The automotive suite consists of finite and infinite impulse response (FIR, IIR) and finite impulse response (FFT) filters. The consumer suite consists of JPEG compression and decompression RGB to CMYK, and RGB to YIQ converter (RGBCMY, RGBHPG, RGBYIQ) while the telecom suite consists of autocorrelation (AUTCOR, CONVEN) and Viterbi decoder (VITERB).

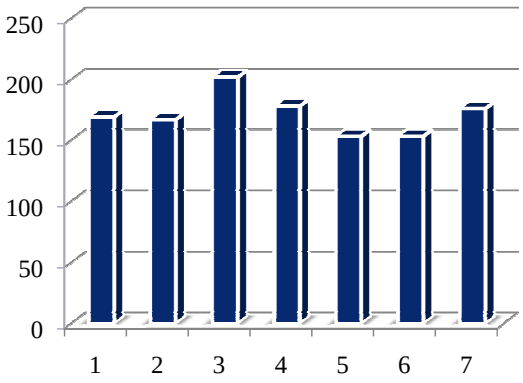
Because of the restrictions of the FlexCore processor we cannot use all the benchmarks, as the processor only supports integers. Furthermore, no floating point support is available yet and also division is not yet included in the FlexCore processor.

5.3 Number of Optimizations Performed

After selecting the benchmarks that are appropriate for the FlexCore processor, we are able to perform more than one hundred fifty number of optimizations on each benchmark see Table 5.1. The least number of optimizations are 153 on *rgbhpg & rgbyiq* and most number of optimizations are 202 on *fft*.

Table 5.1: The number of optimization performed on each benchmark

	Benchmarks	# of Opt. Performed
1	autcor00	169
2	conven00	167
3	Fft00	202
4	rgbcmy01	178
5	rgbhpg01	153
6	rgbyiq01	153
7	viterb00	176

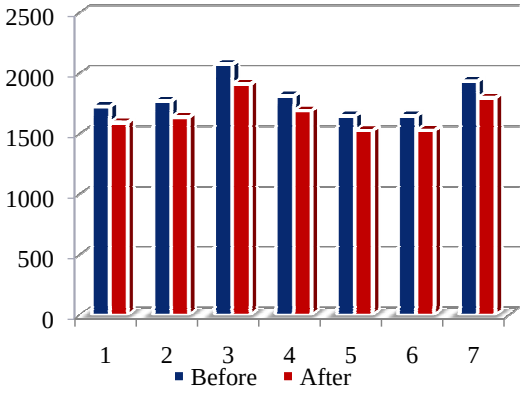


5.4 Code Size Reduction

Table 5.2 describes the number of code lines after performing optimization on each benchmark. More than one hundred code lines are reduced on each benchmark. The *FFT* is the benchmark that has the most number of reduced code lines, that are 164 code lines and in general we reduced 135 code lines after performing the optimization.

Table 5.2: The code size before and after optimization

	BenchMarks	Code	
		Before	After
1	autcor00	1727	1593
2	conven00	1770	1636
3	Fft00	2074	1910
4	rgbcmy01	1813	1691
5	rgbhpg01	1648	1528
6	rgbyiq01	1648	1529
7	viterb00	1934	1797
	Average	1802	1669

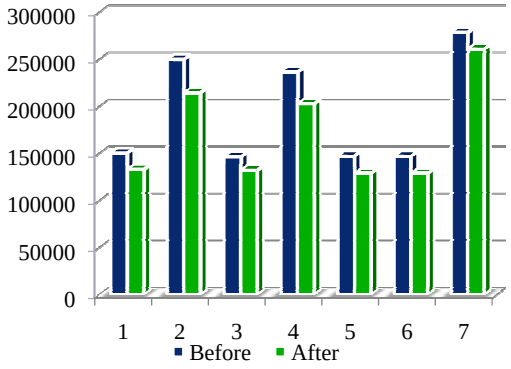


5.5 Execution Time Reduction

As Table 5.3 illustrates, the optimization significantly reduces the execution time of each benchmark. *FFT* has the least number of reduced cycles. *FFT* has 9.82 % of reduced cycles i.e. around 14000 cycles and *conven* has the most number of reduced cycles. *conven* has 14.22 % of reduced cycles i.e. more than 35000 cycles. And in average we are able to reduce around 11 % of cycles that are more than 22000 cycles.

Table 5.3: Number of Execution cycles before and after the optimization

	BenchMarks	Cycles	
		Before	After
1	autcor00	150 284	133 032
2	conven00	249 961	214 427
3	Fft00	146 470	132 090
4	rgbcmy01	236 848	202 864
5	rgbhpg01	146 951	129 019
6	rgbyiq01	146 951	129 019
7	viterb00	278 527	260 907
	Average	**193 713**	**171 623**

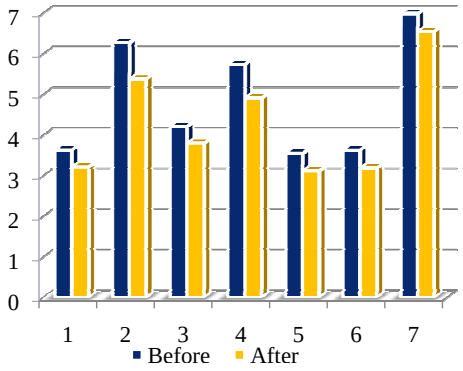


5.6 Energy Dissipation before and after Optimization

Table 5.4 presents the energy dissipation of each benchmark before and after the optimization. Energy dissipation can be calculated with the help of clock period per cycle and the power consumed. We used 2.7 nanosecond (ns) clock period per cycle and the power consumed in each benchmark is given in a previous evaluation [11].

Table 5.4: Energy dissipation before and after optimization

	Benchmarks	Energy Dissipation (μJ)		
		Before	After	Diff
1	autcor00	3.628	3.211	0.416
2	conven00	6.270	5.378	0.891
3	Fft00	4.208	3.795	0.413
4	rgbcmy01	5.736	4.913	0.823
5	rgbhpg01	3.543	3.111	0.432
6	rgbyiq01	3.626	3.184	0.443
7	viterb00	6.986	6.544	0.442
	Average	4.857	4.305	0.552



5.7 Performance Increase

After analysing the reduced cycle count, code size and energy dissipation, now we need to investigate increase in the performance.

Table 5.5: Performance increased by FlexCore Processor after the optimization

	Benchmarks	After Optimization		Performance Increased
		cycles left (%)	Code left (%)	
1	**autcor00**	88.52	92.24	**11.48**
2	**conven00**	85.78	92.43	**14.22**
3	**Fft00**	90.18	92.09	**9.82**
4	**rgbcmy01**	85.65	93.27	**14.35**
5	**rgbhpg01**	87.80	92.72	**12.20**
6	**rgbyiq01**	87.80	92.78	**12.20**
7	**viterb00**	93.67	92.92	**6.33**
	Average	88.50	92.24	**11.51**

Table 5.5 presents the reduction in cycles and code lines after optimization. The benchmark *rgbcmy* shows the largest number of reduction in cycles and code lines, where the optimization has reduced around 14% of cycles and 7% of code lines. In general the overall reduction in cycles is around 12% and code lines is 8%. The overall increase in performance after optimization is 11.5%.

6

Conclusion and Future Work

6.1 Concluding Remarks

This study has shown it is possible to implement microcode optimization in the FlexCore compiler. As discussed in the problem statement in Sec 4.1, the optimization of filling delay slots on the FlexCore compiler has been successfully implemented. During the study we noticed that the branches are more difficult to optimize correctly as compared to other cases, because at this stage it is hard to guess which block of branch would be executed next. The EEMBC benchmark suite is used to evaluate the effectiveness of the optimization on the FlexCore compiler. After implementing the optimization in FlexCore compiler the overall performance of FlexCore processor is increased by 11.5 %.

6.2 Future Work

In future it is highly recommended to perform the optimization when the control is going to be transferred from one function to another. Right now we are only performing the optimization when the control is going to be transferred from one block to another other. This is because the framework only provides the information of one function at a time. Including this might improve the optimization results in the case when the function is in the loop.

Profiling should also be included in the future work. By profiling one can get which block is executing most of the time. This will help to get better optimization of branch. Currently we are selecting block of branch by analyzing which block has the highest probability of execution.

Bibliography

[1] S. Daud, R.B. Ahmad, and N.S. Murhty, "The effects of compiler optimizations on embedded system power consumption," in *International Conference on Electronic Design, 2008. ICED 2008.*, December. 2008, pp. 1 –6.

[2] Martin Thuresson, Magnus Själander, Magnus Björk, Lars Svensson, Per Larsson-Edefors, and Per Stenstrom, "Flexcore: Utilizing exposed datapath control for efficient computing," *Journal of Signal Processing Systems*, vol. 57, pp. 5–19, 2009, 10.1007/s11265-008-0172-z.

[3] John Hughes, Per Larsson-edefors, Mary Sheeran, Per Stenström, and Lars "j". Svensson, "Flexsoc: Combining flexibility and efficiency in soc designs," *Norchip, Riga*, November 2003.

[4] T. Schilling, M. Själander, and P. Larsson-Edefors, "Scheduling for an embedded architecture with a flexible datapath," in *proceedings of the 27th Annual International Symposium on VLSI*, May 2009, pp. 151 –156.

[5] N. Bermudo, A. Krall, and N. Horspool, "Control flow graph reconstruction for assembly language programs with delayed instructions," in *fifth IEEE International Workshop on Source Code Analysis and Manipulation,*, September.-1 October. 2005, pp. 107 – 116.

[6] G. Uh, Y. Wang, D. Whalley, S. Jinturkar, C. Burns, and V. Cao, "Techniques for effectively exploiting a zero overhead loop buffer," in *International Conference on Compiler Construction*, March 2000, pp. 157 – 172.

[7] Ching-Long Su and A.M. Despain, "Minimizing branch misprediction penalties for superpipelined processors," in *ÂÍproceedings of the 27th Annual International Symposium on Microarchitecture, MICRO-27.*, November.-2 December. 1994, pp. 138 – 142.

[8] M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon, "Low-power data forwarding for *VLIW* embedded architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,*, vol. 10, no. 5, pp. 614 –622, October. 2002.

[9] K.P. Subramaniyan, E. Ryman, M. Själander, Tung Thanh Hoang, M.M. Islam, and P. Larsson-Edefors, "Flexdef: Development framework for processor architecture implementation and evaluation," in *Ph.D. Research in Microelectronics and Electronics (PRIME) 7th Conference on*, July 2011, pp. 37 –40.

[10] J Poovey, M Levy, S Gal-On, and T Conte, "A benchmark characterization of the eembc benchmark suite," *IEEE Micro*, vol. 29, no. 5, pp. 18 –29, September.- October. 2009.

[11] Tung Thanh Hoang, Ulf Jalmbrant, Erik der Hagopian, Kasyab P. Subramaniyan, Magnus Själander, and Per Larsson-Edefors, "Design space exploration for an embedded processor with flexible datapath interconnect," in *21st IEEE International Conference on application-specific Systems Architectures and Processors (ASAP)*, July 2010, pp. 55 –62.