Final Thesis

ARM9E Processor Specification for OPTIMIST

by

David Landén

LITH-IDA-EX--05/022--SE

2005-02-25

Linköpings universitet
Department of Computer and Information Science

Final Thesis

ARM9E Processor Specification for OPTIMIST

by

David Landén

LITH-IDA--EX--05/022--SE

2005-02-01

Supervisor: Andrzej Bednarski, Linköpings universitet
Examiner: Christoph Kessler, Linköpings universitet

Abstract

This thesis work provides hardware specifications of the ARM9E microprocessor for the retargetable code generator framework OPTIMIST. We evaluate and compare the code quality of OPTIMIST to a commercial C/C++ compiler for the ARM9E architecture.
Since OPTIMIST is a retargetable code generator it needs both a source code file of the program to generate code for (written in C) and a hardware description file of the processor. The main part of this thesis describes how the specification files for ARM9E are created, using a hardware description language called ADML (Architecture Description Mark-up Language). Finally, as the language could not cover all parts of the processors features, it was extended with new constructs, that are developed in the thesis.

OPTIMIST generated faster code to all test programs compared to the results from LCC with ARM as back-end. In the comparison between OPTIMIST and the compiler from IAR Workbench, OPTIMIST generated just as fast code for every other of the test programs, but the IAR compiler generated faster solutions for the rest of the test programs. This is true for both ARM and Thumb mode.

Sammanfattning

Detta examensarbete tillhandahåller hårdvaruspecifikationer för mikroprocessorn ARM9E till en retargetable code generator kallad OPTIMIST. Vi utvärderar och jämför kodkvaliteten för OPTIMIST och en kommersiell C/C++ kompilator för ARM9E arkitekturen.
Eftersom OPTIMIST är en retargetable code generator behöver den både en källkodsfil av programmet att generera kod för (skriven i C) och en fil som beskriver processorns hårdvara. Huvuddelen av examensarbetet visar hur dessa specifikationsfiler skrevs för ARM9E med hjälp av ett språk kallat ADML (Architecture Description Mark-up Language). Slutligen, eftersom språket inte kunde täcka processorns alla egenskaper utökades det med ytterligare konstruktioner, som är framtagna i detta examensarbete.

OPTIMIST genererade snabbare kod för alla testprogram i jämförelse med resultaten från LCC med ARM som back-end. I jämförelsen mellan OPTIMIST och kompilatorn från IAR Workbench genererade OPTIMIST lika snabb kod för vartannat av testprogrammen, men IAR-kompilatorn genererade snabbare lösningar för resten av testprogrammen. Det gäller både för ARM och Thumb läge.

Contents

Chapter 1 Introduction

The considerable development of embedded microprocessors and their wide-spread use in applications such as cellular phones, digital cameras and PDAs has increased the demand on efficient compilers for microprocessors. Such microprocessors have often irregular architectures[1] where much functionality is held on a very tiny silicon area. Additionally, the more complex the functionality of the processor is, the harder it is to write a compiler that uses the available features in a efficient way. And there is no point of developing new functionality if it is no benefit for the application. This problem is most apparent for a type of microprocessors, called DSPs (Digital Signal Processor), where the code quality (regarding performance, and/or code size) generated automatically (by a compiler) shows an overhead of hundreds of percent compared to hand-written assembler code [10]. The possibility to write programs directly in assembler remains, but it takes much time, it is harder to write applications without the benefits of using a high level language, and the assembler code, in most cases, cannot be moved to a different architecture without considerable changes.

A possibility to solve this problem is to build an optimizing retargetable code generator, that is a code generator that can generate optimized code for different architectures. The retargetability is achieved through the description of the underlying target processor that is provided to the framework simultaneously with the source application. The goal of the OPTIMIST project is to generate optimal code for DSP and VLIW (Very Large Instruction Word) processors.

In this thesis we provide two specifications for the ARM9E processor and evaluate OPTIMIST against LCC and a commercial compiler.

1.1 Goal and Intended Audience

The goal of this thesis work is to extend the specification language of OPTIMIST in order to provide ARM9E specifications and compare its performance to a commercial compiler for the same processor provided by IAR Systems. Two processor specifications files (describing the target architecture) are created, to be used by OPTIMIST, one for ARM mode (32-bit mode) and one for Thumb mode (16-bit mode).

Two different testbenches, Media bench [11] and MiBench [12] are used in the testing phase. The application programs are written in C and performs computations related to multimedia processing, such as image compression and decompression, speech recognition programs, etc. These catalogues represent typical programs for the ARM9E processor.

We assume that the reader has knowledge in the basics of compiler construction and computer hardware architecture.

1.2 Limitations and Sources

The limitations of this project have been quite natural since some parts of the instructions sets could not be implemented by OPTIMIST at the time of writing (see Section 9.2). However most part of the instruction set is implemented, using all addressing modes.

1. Irregular architectures means processors with features such as multiple memory banks, specialized register sets, intricate data paths.

The sources of information for this project have mainly been books (for the ARM processor and the retargetable C compiler LCC), data sheets and manuals from ARM's homepage[1]. Other sources of information are the books for the IAR Workbench [14] (used in the testing phase), and books [5] and online information[2] about OPTIMIST.

1.3 Contributions

This thesis shows how the processor specification files for ARM9E's ARM and Thumb mode were developed and evaluated using the ADML language (the specification language of OPTIMIST). Since some aspects of the processor were not covered by the original version of ADML, some extensions had to be introduced. The additional constructs are described in Chapter 6. The work also includes an overview of the ARM9E processor architecture, relevant for the creation of the specification. Chapter 7 shows how the processor specification files for ARM and Thumb mode were created. The results and conclusion are found in Chapter 8 and 9.

1.4 Related Work

AVIV is a retargetable integrated code generator framework, aimed for VLIW and DSP processors [13]. It takes a hardware specification file written in ISDL (Instructions Set Description Language), and the source file as input. The intermediate representation of the source file consists of so-called split-node DAGs, that are DAGs of basic blocks containing all possible ways to implement the operations on the target machine. Due to the combinatorial explosion, AVIV uses branch and bound heuristics to produce a solution. The instruction selection, scheduling and register allocation are carried out concurrently (but only on basic block level). Optimality of the resulting code cannot be guaranteed but the code is "close to optimal", when working on small basic block with up to 20 instructions.

MARION [15] is a code generator construction system, that is a retargetable code generator system designed specifically for uniprocessor RISCs that contain multiple functional units and multi-cycle operations. MARION creates a code generator from a natural machine description and performs instruction selection, instruction scheduling and global register allocation. Thoses phases are performed one by one, unlike AVIV and OPTIMIST and therefore cannot guarantee an optimal solution. The similarity between MARION, OPTIMIST and AVIV is the usage of a hardware specification file.

The retargetable code generator environment called CHESS [16] is specialized for DSPs and Application Specific Integrated Processors (ASIP), which have load-store architectures with homogeneous or heterogeneous register set where each instruction is micro-coded (computation takes one clock cycle for each instruction).

The hardware specification for CHESS is written in nML, which is a specification language for specifying target processor architectures and instruction sets at the register transfer level. CHESS takes the hardware specification and the source code of the application as input. Two internal data structures, called control-data flow graph and instruction-set graph are created. The intermediate representation of the source program is lowered until all operations can be covered by an instruction from the instruc-

1. www.arm.com

2. www.ida.liu.se/~chrke/optimist/

tion set of the processor. CHESS solves the code selection, register allocation, bit alignment and scheduling separately but each phase is bounded by the constraints from the remaining phases. The code selection is similar to OPTIMIST's approach, using pattern matching for DAGs. The patterns are derived from the target processor description in nML. The patterns are identified when processing DAGs, and the algorithm does not need to enumerate all possible coverage when unnecessary, which is similar to OPTIMIST's approach.

The CHESS framework not only produces the code for the application but also provides statistics that shows how well the model target processor fits the application.

1.5 The Organization of the Report

This report is organized into nine chapters. Chapter 1 briefly introduces the project and OPTIMIST. OPTIMIST is described in greater detail in Chapter 2. Chapter 3 describes the intermediate representation of LCC that represents the input form of programs for OPTIMIST. The ARM9E processor is outlined in detail in Chapter 4. The hardware description language used by OPTIMIST, called ADML, is explained in Chapter 5 and its extension in Chapter 6. Chapter 7 shows how the processor specifications for ARM and Thumb were created. Chapter 8 describes the testbenches and provides experimental result. Chapter 9 gives the conclusions of the work.

Chapter 2 OPTIMIST

OPTIMIST is a research project whose goal is to provide a retargetable code generator, that produces optimal or highly optimized code for irregular VLIW and DSP architectures. In order to make the retargetability feature possible, OPTIMIST takes the program as input together with the specification file of the target processor for which to generate code (see Figure 2.1). The hardware specification is written in a language that is based on XML, called ADML (Architecture Description Markup Language). The specification contains information about the target processor relevant to OPTIMIST. The full description of the language is provided in Chapter 5, “ADML”.

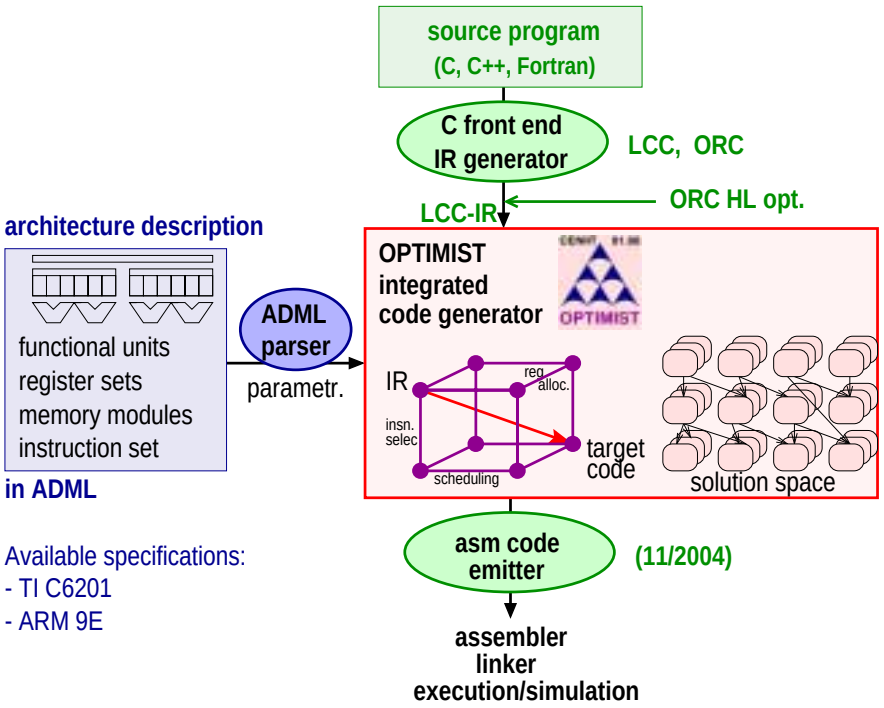


Figure 2.1 OPTIMIST takes LCC intermediate representation of the source program as input together with the hardware specification ADML file. (Christoph Kessler, OPTIMIST homepage: http://www.ida.liu.se/~chrke/optimist/)

OPTIMIST uses the information from the specification file to carry out the code generation for the source program. Code generation means to solve at least three problems; instruction selection (chooses the most appropriate instruction regarding the optimization goals), register allocation (which values should reside in registers, register assignment) and order the sequence of instructions that comply with the program precedence constraints and improve the optimization goal (orders of executing the instructions). In general, for complexity issue, the problems are solved independently in different phases. However, there exist strong couplings between the phases. An early instruction selection may leave a poor choice for the instruction scheduling phase. Thus for generating optimal code it is necessary to solve those three phases simultaneously. This is why OPTIMIST tries to solve the code generation in an integrated manner to produce optimal or highly optimized code, for a given optimization criterion. This is depicted in the compilation cube in Figure 2.1. In a decoupled code generation one follows along the edges of the cube from IR to target code. There are various discussions on order of phases. For instance, the compiler gcc adopted the following order: instruction selection, scheduling and register allocation. Instead of following the edges of the cube,

OPTIMIST solves all phases simultaneously, thus going straight from IR and produces target code.

LCC is used as front-end and it creates the intermediate representation of the program. More about LCC will be given in Chapter 3.

At the time of writing, OPTIMIST can optimize only on basic block level, but will be extended to cope with entire programs, in the long term. A basic block is a section of instructions in the control flow graph with one single entrance and exit, and no branches in the control flow in between.

OPTIMIST can be configured to optimize for different criteria such as: execution time, energy, or register usage. In the future it will also be able to optimize for program size.

Chapter 3 LCC

LCC is a retargetable C compiler, designed at AT&T Bell Laboratories and Princeton University [1, 2]. LCC is a retargetable compiler i.e., it can generate assembler code for different target processors, such as SPARC, MIPS R3000 or x86 Intel processors. The compiler is small and mostly used for education purposes. LCC comes with a book [1] that describes all the source code of the compiler and the 35 different IR-node types in the intermediate representation. The retargetability is obtained by IBurg (Bottom Up Rewriting Generator) [1]. The front-end performs lexical and syntactical analysis of the source program and generates ASTs (Abstract Syntax Trees) as an intermediate representation. The AST are fed (on the fly) to the back-end of LCC. The ASTs are transformed into code DAGs (Directed Acyclic Graphs). The code DAGs are input representation for the back-end and code generator. The back-end then maps the DAGs to assembler instructions. The instruction selection, scheduling and register allocation phases are decoupled, i.e. carried out one at the time.

In the OPTIMIST project, LCC is used as C front-end that produces LCC IR for OPTIMIST. LCC front-end will be used as it is, and OPTIMIST will take the role as back-end. Since OPTIMIST is tightly coupled to LCC-IR the specifications contains information related to that representation.

A code DAG is a graph G = (V, E), where V is the set of DAG nodes and E the set of directed edges, a subset of VxV, representing precedence constraints. The nodes are called IR-nodes throughout this report, meaning that they are parts of the intermediate representation of the program. Each node has an identifier, which is an integer number. A node can have zero, one or two children, referenced by kid[0], kid[1] if any exists, and are numbered from left to right in this report, as in Figure 3.1. Nodes with no children are called leaf nodes.

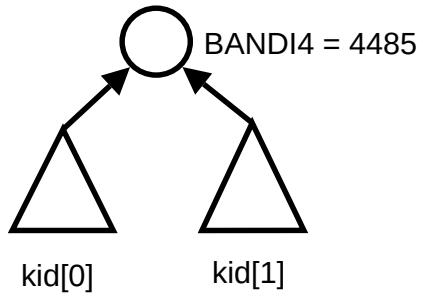


Figure 3.1 The IR-node BANDI4.

Each node has a symbolic name that informs about the operation and types of operands and operation [1,2]. For instance, the name BANDI4 is composed of the symbolic name BAND and type info I4. BAND means that the IR-node represents the operation of *binary AND* with two operands. The I4 gives the information about the operands, where I stands for integer. Both operands must have this type. And 4 means that operands are four bytes long.

Each IR-node is represented by a C structure with various fields. For instance, the operator CNST holds in its field syms[0]->u.c.v the integer value of the constant it represents.

The DAGs are built up by IR-nodes such as the ones described above. In order to understand the IR-node representation, here is an example of a short basic block of a function:

```
int x = 0;
void foo(int y)
{
   x = y + 3;
}
```

Only the DAG representation is of interest for OPTIMIST and for this report, so we skip all but the last representation. The DAG for the code segment is depicted in Figure 3.2.

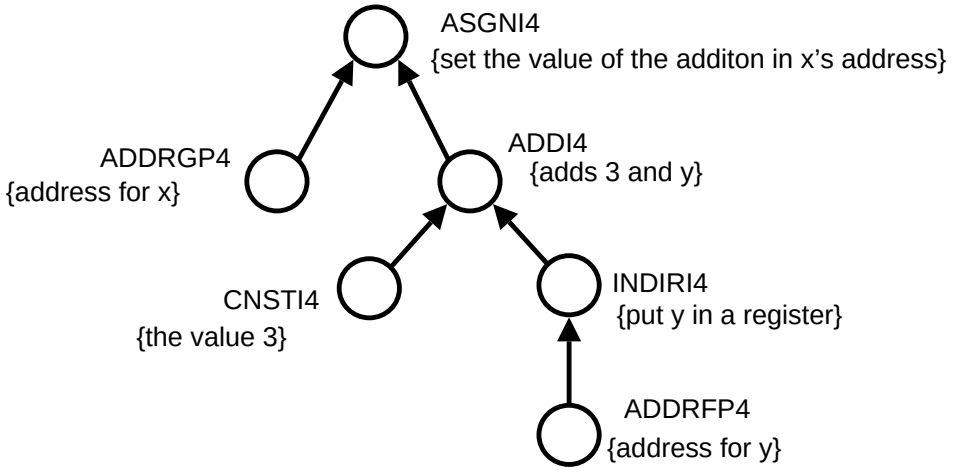


Figure 3.2 The DAG for the example basic block of function foo.

The bottom node ADDRFP4 represents the address of variable y. The name suffix GP4 stands for G: global variable, P: the node is of pointer type and 4: the operands are four bytes long. Before the value can be assigned to y, it has to be loaded into a register, and is fetched by the INDIRI4 node. The value of y is added with the constant CNSTI4, represented by the addition node ADDI4. The result of ADDI4 is the right child (kid[1]) of the assignment node ASGNI4. The result is stored at the address of variable x. We refer to [1] for more information about LCC.

In Chapter 7, "ARM9E Processor Specifications", we show how the specification file describes rules for mappings from IR DAGs to assembler instructions. In order to write the specification file, the processor has to be studied in detail first, which is done in the following chapter.

Chapter 4 ARM9E

This chapter describes the ARM9E architecture: its register file, functional units, pipeline, and pipeline hazards. It is shown how pipeline hazards occur and how they can be avoided. Finally, the instruction set and various addressing modes of the processor for both ARM and Thumb modes are described.

4.1 Overview

ARM9E is a 32-bit RISC (Reduced Instruction Set Computer) processor that is developed by ARM Ltd. [3, 4, 6, 7, 8, 9]. The processor is a single-issue architecture, which means it can only issue one instruction at a time. However, other instructions can be processed in parallel in the pipeline. The pipeline consists of five stages: fetch, decode, execute, memory and write back. All stages except for the execution stage take one clock cycle for the entire instructions set. The execution stage takes a different amount of clock cycles, depending on the instruction and its destination register. Most of the instructions spend one clock cycle in the execution stage.

The processor has 16 general purpose registers visible for the programmer, but there are 37 registers in total. The remaining 21 registers are used internally for speeding up the execution.

The outstanding feature of ARM9E is its two instruction sets associated with a distinct processor mode. The processor can both execute 32-bit instructions in ARM mode, as well as 16-bit instructions in Thumb mode. The processor can switch between the two modes at run time.

4.2 Register File

The ARM9E register file is depicted in Figure 4.1.

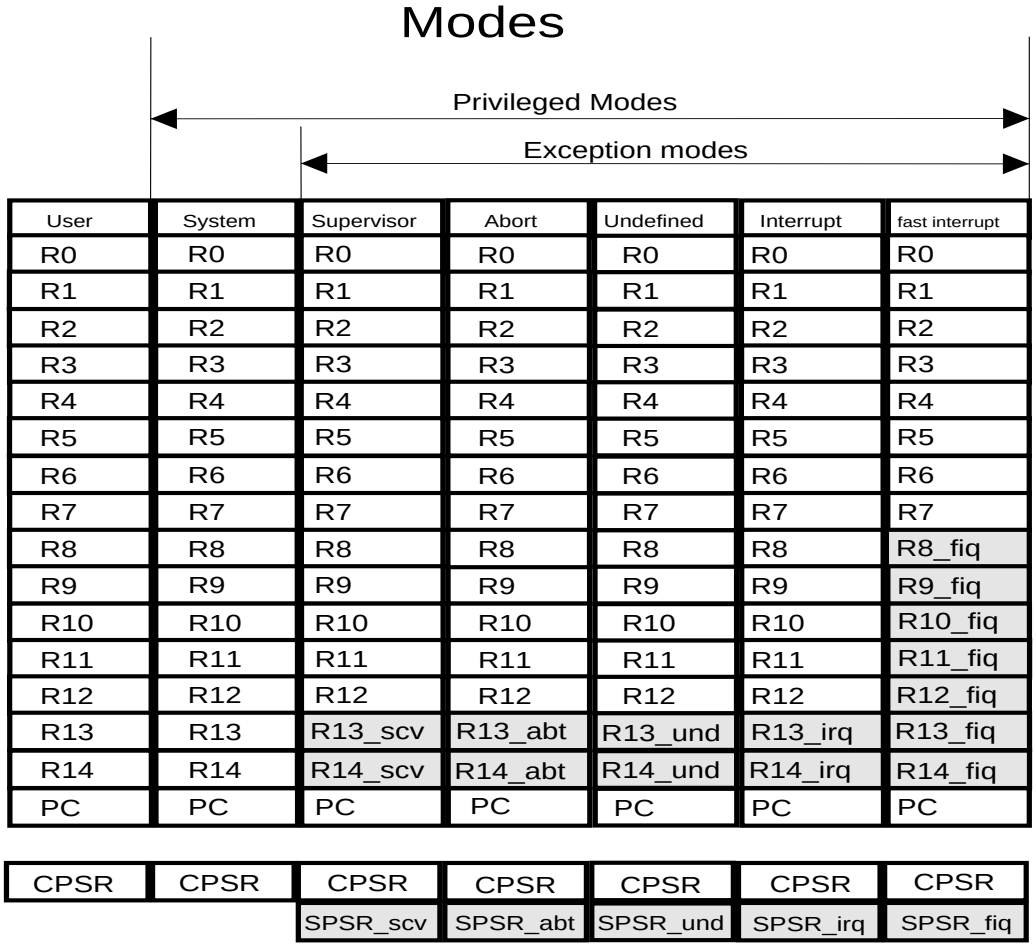


Figure 4.1 The register file with banked and unbanked registers.

The processor has in total 37 registers. Registers in shaded boxes are banked, which means that they are different from the general, user accessible registers (in white boxes). For instance, there is only one register for the acronym R0, but there are six different registers for the acronym R14. The R14 register used in both the User and System mode is the same, but five different R14 register for the modes: Supervisor, Abort, Undefined, Interrupt and Fast Interrupt. If an exception occurs in User mode, the processor enters another state. In order not to overwrite the value of R14 in the exception mode, the processor uses another register that is inaccessible for the user programs to save the value of R14. Thus, when the processor leaves this mode, the value of R14 is not altered because it is the Link Register (LR) and holds the subroutine return address. The same holds for R13 which is the stack pointer register. Each mode has its own stack pointer register, except User and System mode which share the same stack register. The Current Program Status Register (CPSR) contains condition code bits: N (Negative), Z (Zero), C (Carry), V (oVerflow). CPSR also contains the bits determining the current processor mode, if interrupts and fast interrupts are disabled or not, and if the processor is set to execute ARM or Thumb code. The CPSR register is saved in a Saved Program Status Register (SPSR) when entering an exception mode, and can thus be restored when leaving the exception mode.

4.3 Processor Architecture

The ARM9E processor core is depicted in Figure 4.2. The transitions between the stages of the pipeline are outlined with black thick lines.

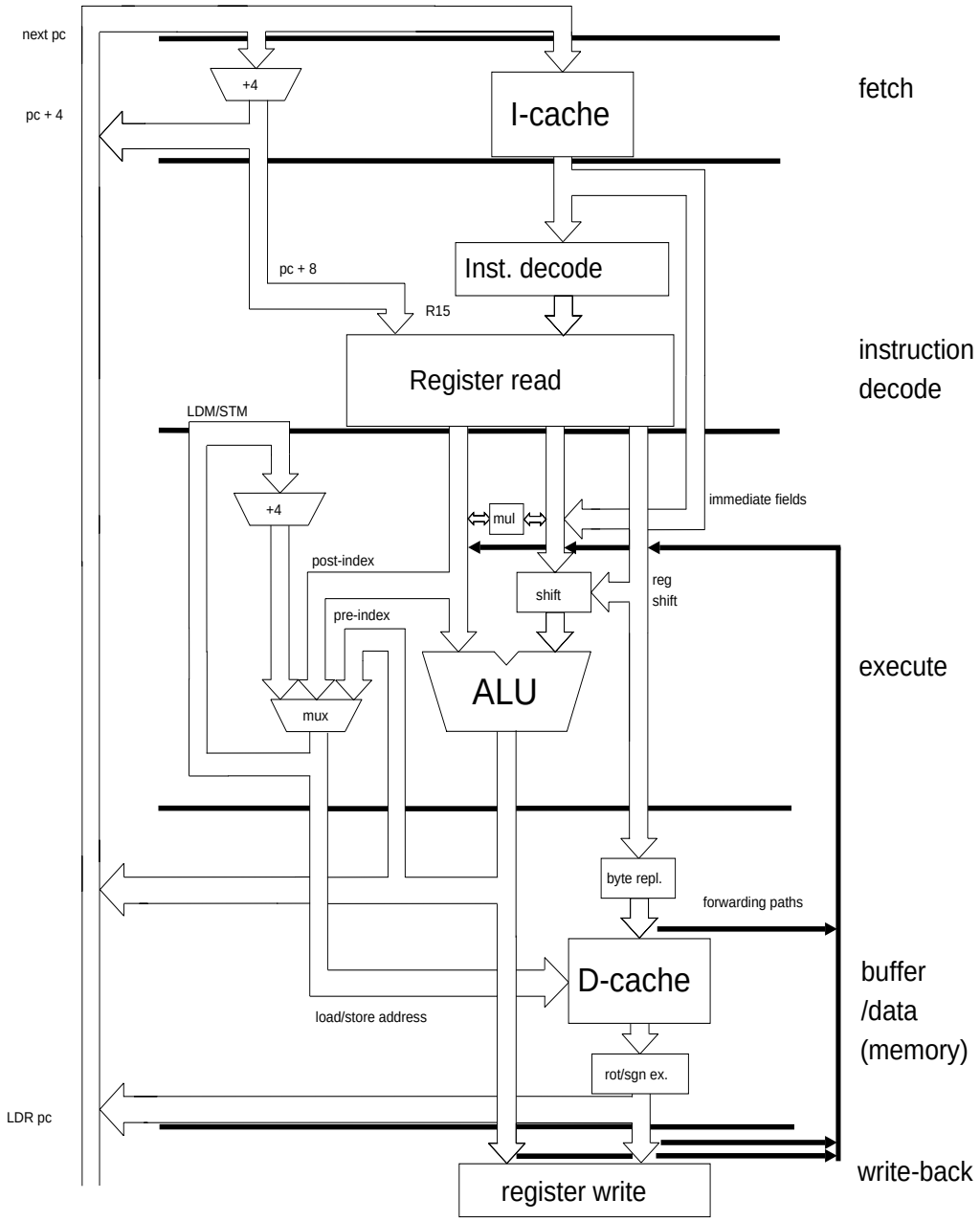


Figure 4.2 The ARM9E processor core.

The five pipeline stages are:

Fetch: The instruction is fetched from memory and placed in the instruction cache / pipeline.

Decode: The instruction is decoded and register operands read from the register file. There are three operand read ports in the register file and two write ports, making it possible to operate on two registers and put the result in a third one.

Execute: Any data operation is carried out in the ALU. One of ARM9E features is that the barrel shifter is connected to the second operand bus leading into the ALU. This makes it possible to perform a shift on the second operand before it enters the ALU in the same clock cycle as the operation is performed. The shifter unit can perform standard shifts such as logical left shift (LSL), logical right shift (LSR), arithmetic shift right (ASR), rotate right (ROR) and rotate right with extend (RRE). The rotate with extension performs a 33-bit rotate right, using the Carry Flag as the 33rd bit.

In case of a load or store instruction, the memory address is computed in the ALU in the execution stage.

Memory: Data memory is accessed if required. Otherwise the ALU result is simply buffered for one clock cycle.

Write-back: The result is written back to the register file. If data has been loaded from memory, for instance, by a LDR (Load register) instruction, it will be written to the register now.

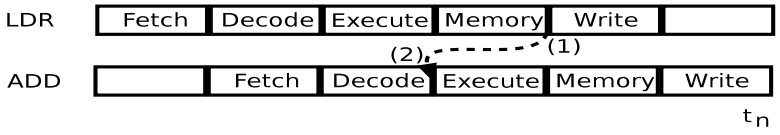
4.4 Pipeline Hazards

Between each pair of subsequent stage of the pipeline lies a so-called forwarding path, i.e. intermediate result registers that can pass data from one pipeline stage to another as soon as it is available (see Section 4.3). But in some occasions an additional delay is required when the result is needed before it is ready. This pipeline hazard is called read-after-write hazard. For instance, an additional delay needs to be added when a register is loaded and this register is set to be used in the following instruction, as depicted by the assembler code below:

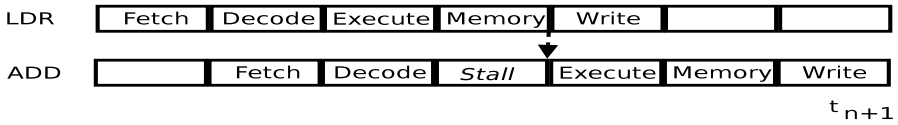
```
LDR R3, [<R2>, +<R4>] ; Loads R3, with the value from address R2 + R4.

ADD R5, R6, R3 ; This instruction uses the value of R3.
```

Both instructions spend one clock cycle each in the execution stage, since the program counter is not the destination register. The content of the pipeline is

:



The data loaded into R3 enters at the end of the memory stage (1) and is needed at the beginning of the execution stage for ADD (2). This leads to a pipeline hazard, which means that the processor has to wait for one clock cycle before it can execute ADD in the execute stage. The processor is stalled and the resulting pipeline looks as follows:

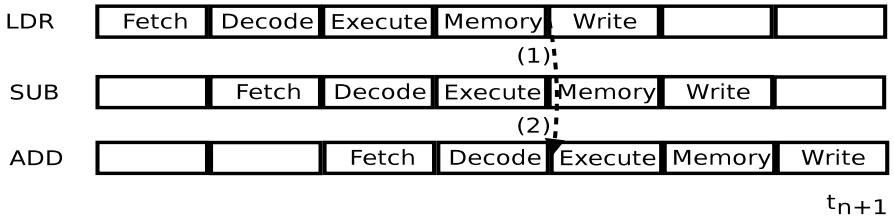


The ARM9E processor, automatically takes care of pipeline hazards, but it is of course not a good outline of the assembler code since the stall results in an additional idle

clock cycle. In case there is any other instruction, independent of the LDR and ADD instruction, that can be put in between LDR and ADD, the stall is removed. For instance, the instruction:

```
SUB R8, R9, R10 ; R8 = R9-R10.
```

can be inserted between the LDR and ADD resulting in the pipeline contents[1]:



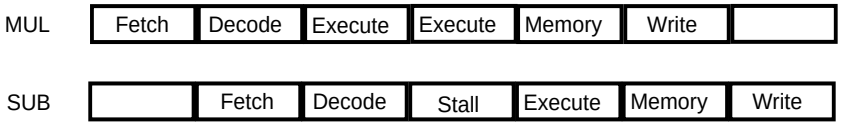
The data for R3 enters LDR's memory stage (1) before it is used by ADD's execution stage (2). No processor stall is needed and this group of instructions takes just as long time to be processed as the two instructions in the first example.

Another example of instructions that introduces stalls in the processor, but cannot be avoided are various branch instructions. This type of hazard is called control hazards. For instance, the instruction ADD using the program counter register (R15) as destination register, i.e. performing a jump:

```
ADD R15, R6, R3 ; R15 = R6 + R3

ORR  R4, R5, R7 ;  R4 = R5 | R7

ORR  R10, R9, R4 ;  R10 = R9 | R4

AND  R10, R10, R4 ;  R10 = R10 - R4

SUB  R7, R8, R2 ;  R7 = R8 - R2
```

When the program counter (register R15) is used as destination for ADD, the pipeline will be filled with fetches and decodes of the following instructions (ORR, ORR, AND), until the execution stage of the branch instruction is reached and then, the instructions have to be discarded and the pipeline flushed. This hazard cannot be avoided, and occurs with all branch instructions.

The last type of hazard is the structural hazard, occurring when an instructions spends more than one cycle in the execution stage of the pipeline. For instance, the instruction MUL takes two clock cycles in the execution stage regardless of the following instructions operands. There is no way of solving this problem since the ARM9E has only one functional unit and the following instructions have to wait for accessing the execution stage.This gives us the following pipeline content:



1. Efficient instruction scheduling should fill, if possible, the delay slot of LDR instructions.

4.5 Multiplication Unit

The multiplication unit is connected to two of the read buses from the ALU (see Figure 4.2). The unit can perform integer multiplication, and unsigned or signed multiplication-accumulate instruction (multiply two operands, and add another, a common operation in DSP applications). The operands can be of 16 or 32 bits, producing 32- or 64-bit products. The multiplication unit uses a modified version of Booth's algorithm [4]. In the multiplication unit, there are four internal registers; two holding partial sums, two holding partial carries. In case a multiply-accumulate operation will be carried out, the partial sums registers are initially loaded by the value to add to the product.

4.6 ARM Instruction Set

The instruction set for the ARM processor is divided into four groups, according to the four groups of instruction types in the ARM Architecture Reference Manual [3]. The four groups are:

1. Data-processing Operations.
2. Load and Store Word or Unsigned Byte.
3. Miscellaneous Loads and Stores.
4. Load and Store Multiple.

The remaining instructions, which lack different addressing modes, are various multiplication instructions, branch instructions and swap instruction. They form additionally two groups.

Each group is described in greater details in the following sections. Whenever talking about different number of clock cycles, we refer to those of execution stage, since it is the only stage that may differ between instructions.

4.6.1 Data-processing Operands

The set of instructions of this group is given in Table 4.1. Only the instructions that are implemented in this project are listed here.

Table 4.1 ASM Description of the Data-Processing Operations Group.

ADD	Addition.
AND	Bitwise logical AND.
BIC	Performs bitwise AND of the value of the first operand and the complement of the second operand.
EOR	Performs bitwise Exclusive-OR.
MOV	Moves a value to a register.
MVN	Move negative, moves the logical one's complement to a register.
ORR	Logical OR, performs a bitwise (inclusive) OR of its two operands.
SUB	Subtraction.

Depending on how the second operand is stored and decoded, there are three different types of addressing modes for data-processing operations. The first operand always

resides in a register. The result is written to another register. The first type corresponds to the situation where the second operand is a constant, derived from two immediate fields in the instruction, called rotate_imm (four bits) and immed_8 (eight bits). The 8-bit field is the last eight bits in the 32-bit instruction word, and the 4-bit field lies directly before the 8-bit field. The immediate value is obtained in the following way:

1. The value of rotate_imm is extracted.
2. All bits in the instruction word are set to zero, except the last eight bits, immed_8.
3. The value in immed_8 is rotated right, by two times the value obtained in 1.

In other words, the constant is equal to:

immed_8 ROR (rotate_imm * 2)

Here is rotate_imm a number between zero and fifteen.

Observe that not all possible 32-bit constants can be constructed this way.

For this case, each instruction executes in one clock cycle; this includes data and constant retrieval.

The second type corresponds to the situation when the second operand is stored in a register. There is a possibility to shift this value (either a logical left or right shift, or a arithmetic shift right or a rotation right by a given number, or a rotation right with extension) before it is used. The shifting is possible since the second operand passes the barrel shifter unit before it enters the ALU. The number of shifts is an immediate value shift_imm (five bits) and lies in the instruction word. Using any kind of shift variant will not result in a longer execution time. Instructions of the second type execute in one clock cycle too.

In the third case, the second operand is stored in a register, as well as the value determining the shift. In this addressing mode it takes two clock cycles to execute an instruction.

Observe that if the program counter is set to be the destination register, all three addressing modes will take two additional clock cycles to complete (see Section 4.4).
The instructions in Table 4.2 have been left out of the project. TST, TEQ, CMP, CMN update the conditions flags and there are no such nodes in LCC, i.e. the instructions do not occur as a stand alone operation but rather as parts of jump and condition structures. The instructions are therefore used in the specification of the branch instructions (see Chapter 7, “ARM9E Processor Specifications”). The instructions ADC, CLZ, RSB, RSC and SBC, are specific assembler instructions that have no equivalent in the LCC IR-representation (but could be included in a future extended version, see Chapter 9).

Table 4.2 Instructions left out from the Data-processing Group.

ADC	ADD with carry. Addition with one register value and the carry flag and another arithmetic value.
CLZ	Count leading zeros. Returns the number of binary zero bits before the first binary one bit register value.
CMN	Compare negative. Compare a register value with the negative of another arithmetic value. The second value is added to the register value.
CMP	Compare. Compare a register value with another arithmetic value. The second value is subtracted from the register value.

Table 4.2 Instructions left out from the Data-processing Group.

ADC	ADD with carry. Addition with one register value and the carry flag and another arithmetic value.
CLZ	Count leading zeros. Returns the number of binary zero bits before the first binary one bit register value.
SBC	Subtract with carry. Subtract one arithmetic value and the carry flag from a register value.
RSB	Reverse SUB, i.e., RSB A B means B - A.
RSC	Reverse SUB with carry.
TEQ	Test if equal. Compare two values by logical exclusive OR-ing them together.
TST	Test. Compare two values by logical AND-ing them together.

4.6.2 Load and Store Word or Unsigned Byte

The set of instructions of this group is given in Table 4.3. Only the instructions that are implemented in the project are listed here.

Table 4.3 ASM Description of the Load and Store Word or Unsigned Byte Group.

LDR	Load register. Loads a word from memory to a register.
LDRB	Load a register byte.
STR	Store register. Store the content of a register to memory.
STRB	Store register byte.

There are nine addressing modes for the Load and Store Word or Unsigned Byte group:

1. Immediate offset.
2. Register offset.
3. Scaled register offset.
4. Immediate pre-indexed.
5. Register pre-indexed.
6. Scaled register pre-indexed.
7. Immediate post-indexed.
8. Register post-indexed.
9. Scaled register post-indexed

1. Immediate offset.

The first type obtains its address by adding or subtracting the value of an immediate offset *offset_12* (of 12 bits) from a base register. The addresses reached from the base register are base register +/- 4096. Using this address mode takes five clock cycles if the program counter is set to destination register, and one clock cycle otherwise.

2. Register offset.

The second type obtains its address by adding or subtracting the value of a register from a base register. The whole address space is accessible in this mode. Using this

address mode, it takes five clock cycles if the program counter is set to be destination register and one clock cycle otherwise.

3. Scaled register offset.

The third type calculates its address by adding or subtracting the shifted (logical left shift, logical right shift, arithmetic shift right) or rotated (rotation right, rotation with extend) value of the content of a register to the base register. The whole address space is accessible in this mode. Using this address mode, it takes six clock cycles if the program counter is set as destination register, and two clock cycles otherwise.

4-6. Pre-indexed modes.

The pre-indexed modes (4-6) works exactly as the corresponding offset modes (1-3), with only one difference. The register holding the base address is updated with the calculation of the address and its offset, *before* the value is stored or loaded from the new address.

7-9. Post-indexed modes.

The post-indexed modes (7-9), also update the base register, but *after* the value has been stored/loaded.

The LDRT, LDRBT, STRT and STRBT instructions have the same effect as their counter parts LDR, LDRB, STR and STRB, but if they are performed when the processor is in privileged mode, they are carried out as if the processor was in user mode. Information about the instructions is given in Table 4.4.

Table 4.4 Instructions left out from the Load and Store Word or Unsigned Byte Group.

LDRBT	Load register byte with translation.
LDRT	Load register with translation.
STRBT	Store byte with translation.
STRT	Store register with translation.

4.6.3 Miscellaneous Loads and Stores

The set of instructions of this group is given in Table 4.5. Two instructions have been left out from the project and are described in Table 4.6

Table 4.5 ASM Description of the Miscellaneous Loads and Stores Group.

LDRH	Load register with halfword. The halfword is zero-extended to a 32-bit word when loaded.
LDRSB	Load register signed byte. The instruction loads a byte from memory, sign-extends it to a 32-bit word.
LDRSH	Load register signed halfword. The instruction loads a halfword from the memory and sign-extend it to a 32-bit word.
STRH	Store halfword. The instruction stores a halfword from the least significant halfword of the register to memory

There are no STRSH or STRSB instructions, because the instructions STRH and STRB cover their functionality too.

There are six addressing modes in the group of Miscellaneous Loads and Stores:

1. Immediate offset.
2. Register offset.
3. Immediate pre-indexed.
4. Register pre-indexed.
5. Immediate post-indexed.
6. Register post-indexed.

The first type obtains its address by adding or subtracting the value of an immediate offset (8 bits) to the base register. The addresses reached by this mode are addresses within the range of the base register +/- 255.

The second type obtains its address by adding or subtracting the value of a register to the base register. The whole addressing space is accessible in this mode.

The addressing modes 3-6 in the group are exactly the same as the modes 1-2, but the base address register is also pre-indexed for modes 3 and 4 (updated before the store/load) and post-indexed (updated after the store/load) for modes 5 and 6.

The functionality of STRD and LDRD are covered by the instructions in the next section and are therefore not specified. Information for the instructions is given in Table 4.6.

Table 4.6 Instructions left out from the Miscellaneous Loads and Stores Group.

LDRD	Load double. Loads a pair of registers from two consecutive words of memory. The pair of registers must start with en even numbered register and end with the subsequent register, for instance, R0, R1.
STRD	Store double. Stores a pair of registers to two consecutive words of memory. The pair of registers must start with en even numbered register and end with the subsequent register, for instance, R0, R1.

4.6.4 Load and Store Multiple.

The instructions of this group, LDM and STM could not be implemented because of their constraints. The Load (LDM) and Store multiple (STM) use a sequential range of addresses. The first address is held in the base register. The remaining addresses are created by adding or subtracting four from the base register. The lowest-number register is stored at the lowest memory address, and the highest register at the highest memory address. The instruction has four addressing modes. The IB addressing mode adds four to the base register for the first address. IA adds four to the next address, and uses the address in the base register for the first store/load. The DB and DA works in a similar way, but subtracts four from the base register instead.

For both instructions, regardless of the addressing mode, it is also possible to alter the base register with the last calculated address value, performing a so-called base register write back.

But the fact that the registers loaded or stored have to be in an increasing order, OPTIMIST cannot guarantee this to happen, since the operands of the instructions are mapped to sets of registers, not a specific register.

The LDM and STM instructions are given in Table 4.7

Table 4.7 ASM Description of the Load and Store Multiple Group.

LDM	Load multiple. The instruction loads a subset of the 16 registers, possible all of them. If the PC is loaded, a branch will occur to that address.
STM	Store multiple. The instructions stores a subset of the 16 registers, possible all to sequential memory locations.

4.6.5 Branch Instructions and Swaps

The set of instructions of this group are given in Table 4.8. Five instructions have been left out from the project (see Table 4.9).

Table 4.8 ASM Description of the Branch Instructions and Swaps Group.

B	Makes an unconditional jump.
BEQ	Branch if equal. Branches if Z = 1.
BNE	Branch not equal. Branches if Z = 0.
BGE	Branch if greater than or equal. Branches if NV = 11 or NV = 00.
BLT	Branch if less than. Branches if NV = 10 or NV = 01.
BGT	Branch if greater than. Branches if Z = 1 and (NV = 11 or NV = 00).
BLE	Branch if less than. Branches if Z = 1 or NV = 01 or NV = 10.

All branch instructions have only one operand, a label. The branch can reach addresses in the range of $+/-2^{26}$. The immediate value is of 24 bits but is shifted 2 times when retrieving the address label and added to the program counter. All branch instructions takes three clock cycles in the execution stage.

The BL, BLX and BX instructions have been removed from the project, since the BX and BLX change the processor mode from ARM to Thumb. BL is branch with link and is more useful when programming a call to subroutine in assembler. There is no IR-node that represents this branch instruction. The SWP and SWPB instructions were too hard to specify (their patterns contains cycles, i.e. cannot be mapped to DAGs).

Table 4.9 Instructions left out from Branch Instructions and Swaps Group.

BL	Branch and link. Branches and saves the return address in the link register (R14). Other branch instructions can be combined with the link option too.
BLX	Branch and exchange. Branches and switches to Thumb mode. The return address is saved in the link register (R14).
BX	Branch and exchange. Branches and switches to Thumb mode incase the Thumb-bit is set in the CSPR or continues in ARM mode.
SWP	Swaps a word from memory to a register, and a register value (a word) to the same memory address. The receiving and sending register can be the same or two different registers.
SWPB	Swaps a byte from memory to a register, and a register value (a byte) to the same memory address. The receiving and sending register can be the same or two different registers.

4.6.6 Multiplication Instructions

The set of implemented instructions of this group is given in Table 4.10. The instructions that have been left out from the project are given in Table 4.11.

Table 4.10 ASM Description of the Multiplication Instructions Group.

MLA	multiply-accumulate. Multiplies two unsigned operands to produce a 32-bit product that is added to another 32-bit value.
MUL	Multiply two unsigned values to produce a 32-bit value.
SMLATT SMLATB SMLABT SMLABB	Signed multiply-accumulate on signed 16-bit values, taken from either top (T) or bottom (B) halves of two registers. The result is sign-extended and added to a 32-bit product.
SMLAWT SMLAWB	Signed multiply-accumulate. A signed 32-bit value is multiplied with a signed 16-bit value. The later one is taken from either the top (T) or bottom (B) half of a register. The 48-bit producet is added to a 32-bit value and the bottom 16 bits are ignored.
SMULTT SMULTB SMULBT SMULBB	Signed multiply on two signed 16-bit values, taken from either top (T) or bottom (B) halves of two registers.
SMULWT SMULWB	Signed multiply. A signed 32-bit value is multiplied with a signed 16-bit value. The later one is taken from either the top (T) or bottom (B) half of a register.

The multiplication instructions always operates directly on values in registers, never on immediate values.

The MUL and MLA instructions take two clock cycles in the execution stage. SMULTT, SMULTB, SMULBT, SMULBB, SMULWT and SMULWB takes three clock cycles in the execution stage. The SMULTT, SMULTB, SMULBT, SMULBB, SMULWT, SMULWB takes one clock cycle in the execution stage. If the following instruction tries to use the value stored in the destination register in its first cycle in the execution stage or memory stage the processor will be stalled for one clock cycle.

At the time of writing, OPTIMIST cannot handle instructions that generate a 64-bit product. See Section 9.2 for more information about the difficulty in implementing those instructions.

Table 4.11 Instructions left out from the Multiplication Instructions Group.

SMLAL	Signed multiply-accumulate long. Multiplies two signed 32-bit values and produces a 64-bit value. Another 64-bit value is added to the product.
SMULL	Signed multiply long. Multiplies two signed 32-bit values and produces a 64-bit product.
UMLAL	Unsigned multiply-accumulate long. Multiplies two unsigned 32-bit values and produces a 64-bit product. The product is added to another 64-bit value.
UMULL	Unsigned multiply long. Multiplies the two unsigned 32-bit values and produces a 64-bit product.
SMLALTT SMLALTB SMLALBT SMLALBB	Signed multiply-accumulate long on signed 16-bit values, taken from either top (T) or bottom (B) halves of two registers. The result is sign-extended and added to a 64-bit product.

4.7 Thumb Addressing Modes

The Thumb mode is much simpler than the ARM mode. The number of different addressing mode types is fewer, compared to ARM mode. In a similar way as the ARM instruction set, the Thumb instruction set can be divided into three groups:

1. Data-processing instructions.
2. Load and Store Register.
3. Load and Store Multiple.

Each group is described in greater details in the following three sections.

4.7.1 Data-processing Instructions for Thumb

The set of implemented instructions of this group is given in Table 4.12. Instructions ADC, CMN, CMP(1), CMP(2), SBC and TST are not implemented in the Thumb specification basically, for the same reasons as their corresponding instructions in the ARM specification (see Section 4.6).

Table 4.12 ASM Description of the Data-processing instructions for Thumb Group.

ADD(1)	Addition of a register value and a 3-bit immediate. The result in another register than the operand register.
ADD(2)	Addition of a 8-bit immediate and a register value.
ADD(3)	Addition of a register value to another.
ADD(4)	Addition of any two registers (but both may not be of the registers R0..R7).
AND	Logical AND with the value of destination register and the value of another register.
ASR	Arithmetic shift right. Both the number of step to shift and the value to be shifted reside in registers.
BIC	Bit clear. Logical AND with the compliment of the second operand and the first operand.
EOR	Logical exclusive OR with the destination register and another register.
LSL(1)	Logical shift left. The value to be shifted resides in a register and the number of steps is given by a 5-bit immediate.
LSL(2)	Logical shift left. The value to be shifted resides in a register and the number of steps is given by another register value.
LSR(1)	Logical shift right. The value to be shifted resides in a register and the number of steps is given by a 5-bit immediate.
LSR(2)	Logical shift right. The value to be shifted resides in a register and the number of steps is given by another register value.
MOV(1)	Moves a 8-bit immediate value to register.
MOV(2)	Moves a register value to another register value.
MOV(3)	Moves a register value between any two registers (but both may not be of the low registers, i.e. R0-R7)
MUL	Multiply two register values.
NEG	Negates a value in a register.
ORR	Logical (inclusive) OR between the destination register and another register value.
ROR	Rotate right. The value to be rotated and the number of steps to rotate reside in registers.

Table 4.12 ASM Description of the Data-processing instructions for Thumb Group.

ADD(1)	Addition of a register value and a 3-bit immediate. The result in another register than the operand register.
SUB(1)	Subtract a 3-bit immediate from a register value. The result is assigned to another register.
SUB(2)	Subtract a 8-bit immediate from a register.
SUB(3)	Subtract a register value from another register value.

ADD(4) and MOV(3) are the only instructions that takes 3 respectively four clock cycles in the execution stage if the program counter is set as destination register. In all other cases, ADD(4) and MOV(3) as well as the rest of the data processing instructions take one clock cycle. The largest possible immediate value to be added respectively subtracted by ADD(1) and SUB(1) is 7. The number of steps of shift in LSR and LSL ranges from 0 to 31 steps. The largest value to be represented by the 8-bit immediate in ADD(2), MOV(1) and SUB(2) is 255.

4.7.2 Load and Store Register for Thumb

The set of instructions of this group are given in Table 4.13. Only the instructions that are implemented in this project are listed here. The instructions: LDR(1), LDR(3), STR(1) and STR(3) are left out since LDR(2) and STR(2) are regarded as sufficient (see Table 4.14 for information about the instructions).

Table 4.13 ASM Description of the Load and Store Register for Thumb Group.

LDR(2)	Load word (register offset).
LDRB(1)	Load unsigned byte (immediate offset).
LDRB(2)	Load unsigned byte (register offset).
LDRH(1)	Load unsigned halfword (immediate offset).
LDRH(2)	Load unsigned halfword (register offset).
LDRSB	Load signed byte (register offset).
LDRSH	Load signed halfword (register offset).
STR(2)	Store register (register offset).
STRB(1)	Store byte (immediate offset).
STRB(2)	Store byte (register offset).
STRH(1)	Store halfword (immediate offset).
STRH(2)	Store halfword (register offset).

Table 4.14 Instructions left out from the Load and Store Multiple for Thumb Group.

LDR(1)	Loads the value on the address equal to the base register + (an immediate 5-bit value times four) to a register.
LDR(3)	Loads the value of the program counter + (an 8-bit immediate value times four) to a register.
STR(1)	Stores a register value to the address equal to the base register + (an immediate 5-bit value times four) to a register.
STR(2)	Stores the value of the stack pointer + (an 8-bit immediate value times four) to a register.

4.7.3 Load and Store Multiple for Thumb

None of the instructions in this group were implemented in Thumb mode, due to the same reason as for ARM mode. The set of instructions of this group is given in Table 4.15.

Table 4.15 Instructions left out from the Load and Store Multiple for Thumb Group.

LDMIA	Load multiple increment after. Loads a non-empty subset or possibly all of the low registers.
POP	Loads a non-empty subset or possibly all of the low registers and the program counter (R15) from the stack.
PUSH	Stores a non-empty subset, or possibly all, of the general-purpose registers and the link register (R14) to the stack.
STMIA	Store multiple increment after. Stores a non-empty subset, or possibly all, of the general-purpose registers to sequential memory locations.

Chapter 5 ADML

ADML (Architecture Description Mark-up Language) is a hardware description language, based on XML (Extensible Mark-up Language). The language is developed for the OPTIMIST project solely, and is used to describe the relevant structures of the target processor [5].

This chapter describes different parts of the processor specification document. Examples from the ARM-specification for the processor ARM9E elucidate the different parts of the specification. The instructions that are mentioned here are all from the ARM9E's instruction set in ARM mode.

5.1 General ADML Document Structure

An ADML specification is an XML document. The specification is a tree with a root node labeled "architecture" with children labeled, "omega", "register", "constants", "residence classes", "functional units", "patterns", "instruction set", and "transfer". There is no predefined order in which those parts should appear, but the sections below follow the order in which they are specified in the processor specification for ARM. The different parts are described in greater detail below.

5.2 Notations

A simple notation is presented in this section for describing the specification. The expression [string] means a string of alphanumeric tokens. The string could start with a numeric token and contain spaces. In the same way, [integer] means a number either positive or zero. Adding a star after the type name, i.e., [integer*] means a strict positive number. The expression [type] corresponds to the string "unsigned" or "signed". Abbreviation (...) tells that parts of the specification have been left out.

5.3 Omega: Issue Width

The general description of the omega part is:

```
<omega>[integer]</omega>
```

The issue-width represents the maximum possible number of instructions that can be issued simultaneously. For VLIW processors, omega is greater than one. For ARM9E, there is only one instruction emitted every clock cycle, thus we specify:

```
<omega>1</omega>
```

5.4 Registers

The general description of the register part is:

```
<registers>
  <reg id = "[string]" size= "[integer*]"/>
  <reg id = "[string]" size= "[integer*]"/>
  ...
  <aliases>
      <alias id= "[string]" src= "[string]">
        <bits>
          <start>[integer]</start>
          <end>  [integer]</end>
        </bits>
       </alias>
       ...
 </aliases>
</registers>
```

The register part enumerates all of the registers that are accessible for the user programs. For each register, an optional attribute specifies its width in term of bits.

Example: The register part of the ARM-specification (there are 16 registers in ARM mode):

```
<registers>
  <reg id = "R0" size = "32"/>
  <reg id = "R1" size = "32"/>
  ...
</registers>
```

The attribute id is the name of the register, R0 and R1 in this example. The attribute size shows the length of the register in bits, 32 bits in this example.

Since there are processors that have instructions which operate on parts of registers, it has to be possible to name those parts of the registers. In ARM9E, a few instructions can operate on register halves. In order to specify those halves, the following alias part is added to the register specification:

```
<registers>
  <reg id = "R0" size= "32"/>
  <reg id = "R1" size= "32"/>
  ...
    <aliases>
      <alias id= "R0B" src= "R0">
        <bits>
          <start>0</start>
          <end>15</end>
        </bits>
       </alias>
       ...
    </aliases>
</registers>
```

The attribute id of the aliases node is the name of the register part, and src is the register, specified in the register declaration. The parts of the source register (section of bits) are specified with a list of start/end nodes. The different parts are described in their number of bits, given the start and end bit. In this manner, the 16 registers in ARM9E are split up by alias into 32 register-halves.

5.5 Constants

Just as operands could be stored in registers, they could also reside in an instruction word. Such a part of the instruction is a constant, and also has to be specified in the processor specification.

The general description of a constant specification is:

```
<constants>
  <constant id = "[string]" width = "[integer*]" type = "[type]"/>
  <constant id = "[string]" alias = "[string]"/>
  ...
</constants>
```

For instance, in the branch instruction B, there is a 24-bit immediate value, holding the destination address. Another example is the ADD instruction that could take one of its operands from a register and the other from the instruction itself.

Example:

```
<constants>
  <constant id = "const_data_immediate"  width = "12" type = "unsigned"/>
  <constant id = "const_signed_immed_24" width = "24" type = "signed"/>
 ...
</constants>
```

5.6 Residence Classes

The registers (see Section 5.4) are grouped together in different residence classes that are derived from the instruction set (how this is done will be shown in Section 7.1.2). A residence class contains a set of registers such that, regardless of which register is chosen from the set for an operand of an instruction, the instruction will take the same time to be executed. Here we assume that a register may not be present in two or more residence classes, i.e. the residence classes may not overlap. The id for the node reg is the name of the register in the residence class.

Another entity called residence is also specified in this part, meaning a memory module. The id for the node residence is the name of the memory module. The size of the memory module is considered to be infinite.

The general description of a residence class is as follows.

```
<residenceclasses>
  <residenceclass id = "[string]">
    <reg id ="[string]"/>
    <reg id ="[string]"/>
    ...
  </residenceclass>
  <residenceclass id = "[string]">
    <reg id ="[string]"/>
    <reg id ="[string]"/>
    ...
  </residenceclass>
</residenceclasses>
```

5.7 Functional Units

A functional unit is a hardware part of a processor that performs a certain number of computations, such as addition, multiply etc.

The general description of a functional unit in ADML is:

```
<funits>
  <fu id = “[string]“ occupation = “[integer*]“ latency=”[integer*]”/>
  <fu id = “[string]“ occupation = “[integer*]“ latency=”[integer*]”/>
  ...
<funits>
```

The attributes of a functional unit are id, the name of the unit. The occupation is the number of clock cycles an instruction occupies the functional unit before another instruction can enter the functional unit. The latency is the number of clock cycles taken by an instruction on the functional unit before the result is available.

At present time, OPTIMIST's algorithm assumes that occupation time is shorter or just as long as the latency. This is true for most architectures.

ARM9E has only one functional unit containing both the multiplication unit and the ALU, since multiplication instructions use the ALU in the computations too (see Figure 5.1).

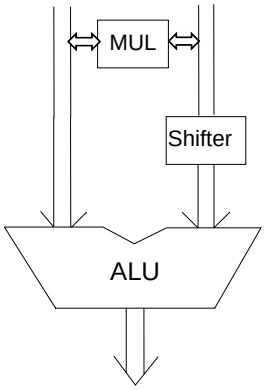


Figure 5.1 The MUL, ALU and shifter unit in the ARM9E processor.

The functional unit in Figure 5.1 is specified as:

```
<funits>
  <fu id = "ALU" occupation = "1" latency = "1"/>
</funits>
```

5.8 Patterns

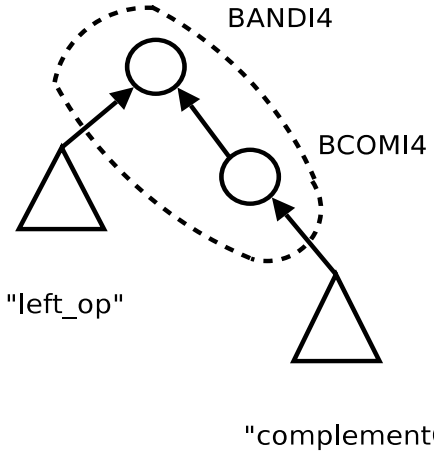
Patterns describe sub-DAGs (or forests of sub-DAGs) of IR-nodes that are computed by a single instruction of the processor's instruction set (described in Section 5.9).

To illustrate how a pattern is specified, we use a graph representation. Nodes with circular shape have at least one child. Triangular-shaped nodes are leaf nodes, i.e., they have no children. They are the parameters (operands) of the target instruction.

In the example, we have two nodes composing the BIC pattern, namely BANDI4 and BCOMI4 (see Figure 5.2).

Operand nodes can be named within the pattern. In the BIC pattern, the left operand is named "left_op" and the right operand is called "complementOf". Nodes are connected by data dependence edges and preceding edges forming an DAG called a DAG pattern. We use nesting of XML to implicitly construct edges. Additional edges such as prece-

dence edges added implicitly by LCC (from LCC front-end) can be specified with the clause <ddep src = "[string]" dest = "[string]"/>.

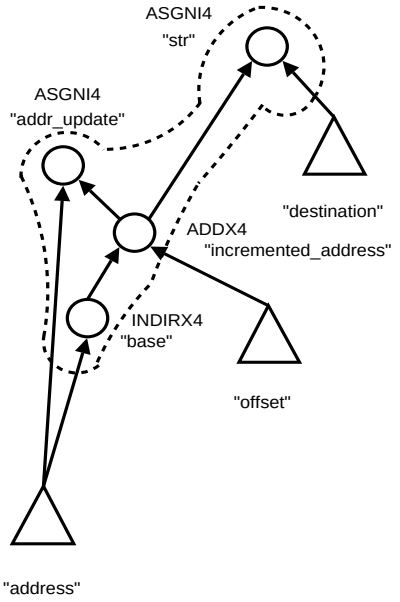



```
<pattern id = "BIC">
  <node>
     <pinst id = "BANDI4" arity = "2" op = "4485"/>
          <kid nr = "0" id = "leftOp"/>
          <kid nr = "1">
            <node>
              <pinst id = "BCOMI4" arity = "1" op = "4501"/>
                 <kid nr = "0" id = "complementOf"/>
            </node>
          </kid>
   </node>
</pattern>
```

Figure 5.2 The pattern BIC is created by grouping the IR-nodes BANDI4 and BCOMI4. The identifiers “left_op” and “complementOf” are references to those nodes, operands of BIC LCC-IR representation.

BIC is an example of a tree pattern, i.e., all IR-nodes in the pattern have exactly one parent node, except the root-node, that has no parent. The attribute arity specifies the number of child nodes for every node (the kids are numbered from left to right, as described in Chapter 3, “LCC”). In LCC, IR-nodes have arity zero for leaf nodes, one for unary operations and two for binary operations.

Another example of a pattern is the DAG pattern for the instruction STR (store one register) using the pre-indexed addressing mode. First it calculates the destination address by adding the immediate value as offset to the base register, and secondly stores the data. This is performed by a single instruction on the ARM9E processor. The pattern is specified as a DAG pattern, because there are two top nodes in this pattern. Figure 5.3 shows the pattern and its description. The nodes `str` and `addr_update` are the two top nodes.




```
<pattern id = "STR_ADD_OFFSET_PRE">
  <node id = “str”>
    <pinst id = "ASGNI4" arity = "2" op = "2102"/>
      <kid nr = "0" id = "incremented_address"/>
      <kid nr = "1" id = "destination"/>
  </node>
  <node id = “addr_update”>
      <pinst id = "ASGNI4" arity = "2" op = "4149"/>
        <kid nr = "0" id = "address"/>
        <kid nr = "1" id = "incremented_address">
          <node>
            <pinst id = "ADDI4" arity = "2" op = "4405"/>
              <kid nr = "0" id = "base">
                <node>
                  <pinst id = "INDIRI4" arity = "1" op = "4165"/>
                  <kid nr = "0" id = "address"/>
                </node>
              </kid>
              <kid nr = "1" id = "offset"/>
            </node>
          </kid>
    </node>
    <ddep src = "addr_update" dest = "str"/>
</pattern>
```

Figure 5.3 The pattern and pattern specification for the instruction STR, when adding a register or immediate value to the base register.

The general description of a pattern is:

```
<pattern id = "[string]">
  <node id = "[string]">
    <pinst id = "[string]" arity = "[integer]" op = "[integer]">
      <kid nr = “0”>
        <node>
        ....
        </node>
      <kidnr = “1”>
        <node>
        ....
        </node>
  </node>
  <node id = "[string]">
    <pinst id = "[string]" arity = "[integer]" op = "[integer]">
      <kid nr = “0”>
        <node>
        ....
        </node>
      <kidnr = “1”>
        <node>
        ....
        </node>
  </node>
  ...
</pattern>
```

The attribute id for the node pattern is the name of the pattern, referenced by an instruction specification in the instruction part (see Section 5.9). The attribute id of the node named node is an optional reference to the node. The pinst nodes are the IR-nodes of the pattern. Its attributes are the name id, the arity (number of children) and the operation number op. The children of an IR-node are numbered and referenced by the kid

nodes using the kid nr attribute of kid node. It is possible to set any node as commutative <node commutative="yes">, which means that the children of the nodes can be switched and OPTIMIST cover both cases. This construct is optional and nodes are noncommutative by default.

5.9 Instruction Set

In the instruction set part, all of the processors instructions are specified. An instruction that corresponds to one IR-node solely, is specified as an instruction. An instruction that covers a DAG-pattern (described in the pattern part, Section 5.8) is specified as a pattern node of the instruction part.

5.9.1 Instructions

In this section we describe how instructions that cover single IR-nodes (i.e., one-to-one mapping) are specified in ADML.

The general description of an instruction specification is:

```
<instruction id =”[string]” op =”[integer]”>
  <target id=”[string]” op1 = “[string]“ ... opN = “[string]” use_fu = “[string]“/>
    <cycle_matrix ... />
    <format>...</format>
  </target>
  <target id=”[string]” op0 = “[string]“ ... opN = “[string]” use_fu = “[string]“/>
    <cycle_matrix ... />
    <format>...</format>
  </target>
  ...
</instruction>
```

Below is an example of BANDU4, the LCC IR-node for the instruction for binary AND with two operands each four unsigned bytes long. The instruction corresponds to a single IR-node, BANDU4.

```
<instruction id = "BANDU4" op = "4486">
  <target id = "AND" op0 = "PC" op1 = "all"
                op2 = "const_data_immediate" use_fu = "ALU">
    <cycle_matrix ... />
    ...
    <format> AND {op1},{op0},#{op2}</format>
  </target>
</instruction>
```

The attribute id of the instruction node is simply the name of the IR-node in LCC.

The attribute id of the node target is the name of the specified target instruction. Attributes op0, op1 and op2 specify the residence class of operands (op1 and op2) and the residence class of the result, if any (in op0). Attributes are optional, i.e., for leaf nodes, there are no op1 and op2, and for root nodes op0 is not specified. The functional unit to execute the instruction is set in the attribute use_fu.

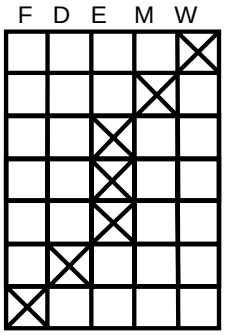
The format clause tells the emitter how to produce the assembler command for the instruction. Attributes inside curly brackets will be replaced by a register from the respective residence class. Constants are replaced by their values. Everything else is simply copied as strings. A possible output of the format specification may be:

```
AND R0, R1, #1
```

The cycle_matrix describes the reservation table of the target instruction. A reservation table is simply a matrix with time as y coordinate and x as the number of resources that can be occupied at a given time stamp. The resources in this case are the stages of the pipeline. Time is counted in number of clock cycles:

```
<cycle_matrix ... >
  <cycle_matrix ... >
    ...
  </cycle_matrix>
</cycle_matrix>
```

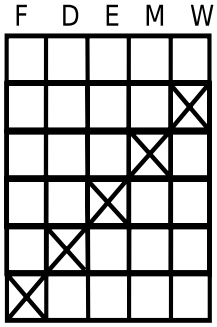
The outermost row describes which resources are occupied by the instruction in its first clock cycles. For instance, the instruction AND uses four clock cycles in the execution stage and one in remaining stages. The reservation table and its specification are as follows:




```
<cycle_matrix   fetch = “1” decode = “1” execute = “3”>
      <cycle_matrix   execute = “1” memory = “1” write_back = “1”>
</cycle_matrix>
```

The cycle_matrix specifies occupation status for each resource. Stages not occupied in a clock cycle are omitted in the specification part.

For instructions with delay slots (that can introduce processor stalls), it is possible to insert a latency row in the cycle_matrix clause specifying the amount of delay cycles. LDR is an example of a delayed instruction, i.e. one stall cycle can occur (see Section 4.6.2). The reservation matrix of LDR is modeled by the following cycle_matrix clause:




```
<cycle_matrix fetch = “1” >
  <cycle_matrix decode = “1”>
    <cycle_matrix execute = “1”>
      <cycle_matrix memory = “1”>
        <cycle_matrix write_back = “1”>
          <latency l = “1”/>
        </cycle_matrix>
      </cycle_matrix>
    </cycle_matrix>
  </cycle_matrix>
</cycle_matrix>
```

OPTIMIST uses the reservation matrix to place instructions in the best possible way, i.e. avoiding processor stalls and unnecessary bubbles in the pipeline. The concept of cyclic_matrix is applicable in a single-issue or a VLIW processor, since the reservation table is a general concept.

5.9.2 Patterns

In Section 5.8 we described how DAG-patterns are specified within pattern nodes. A pattern is a map from a DAG of IR-nodes to a target instruction, where the leaves of the pattern become the operands of the target instruction.

The general description of a pattern specification is:

```
<pattern id = “[string]“>
  <ptarget id = “[string]“/>
    <op op0 = “[string]“/>
    <op op2 = “[string]“/>
    <op id = “[string]“>
      <id>”[string]”</id>
    </op>
    ...
    <fu use_fu=”[string]”/>
    <cycle_matrix.../>
    <format>...</format>
  </ptarget>
  ...
</pattern>
```

The instruction description that connects to the BIC pattern described in Section 5.8, is as follows:

```
<pattern id = "pattern_BIC">
  <ptarget id = "BIC" op0 = "allExceptPC">
  <op id="leftOp">
    <id>all</id>
  </op>
  <op id= "complementOf">
    <id>const_data_immediate</id>
  </op>
  <fu use_fu = "ALU"/>
  <cycle_matrix.../>
  <format>...</format>
  </ptarget>
</pattern>
```

There are no significant differences between a pattern description and an instruction description. One difference, though, is the specification of the operands. Operands are referenced by their symbolic names declared in the pattern definition of the section <patterns>...</patterns>, described in Section 5.8. The pattern node of the specification is used to map a pattern definition to a specific target instruction. The pattern clause tells that it is an instruction for a pattern of IR-nodes. The word ptarget is used instead of target.

The general description of the instruction set part is:

```
<instructionset>
  <instruction id =”[string]” op =”[integer]”>
    <target id=”[string]” op0 = “[string]“ ...  opN = “[string]” use_fu = “[string]“/>
      <cycle_matrix ... />
    <format>...</format>
    </target>
    <target id=”[string]” op0 = “[string]“ ... opN = “[string]” use_fu = “[string]“/>
      <cycle_matrix ... />
    <format>...</format>
    </target>
     ...
  </instruction>
  ...
  <pattern id = “[string]“>
```

```
    <ptarget id = “[string]“ op0 = “[string]“/>
      <op id = “[string]“>
        <id>”[string]”</id>
      </op>
      ...
      <fu use_fu=”[string]”/>
      <cycle_matrix.../>
      <format>...</format>
    </ptarget>
    ...
  </pattern>
  ...
</instructionset>
```

Observe that the second and following target node in the instruction is optional.

5.10 Transfer

Instructions that move data between different residences (memory modules and registers), i.e., various move, load and store instructions, are specified in this part. The instructions in this part only have to be specified with their operands and residence classes, cycle_matrix and format clause, since transfer instructions are used implicitly. They are not to be mapped to IR-nodes, and thus not a part of the instruction selection.

The general description of the transfer part is:

```
<transfer>
  <target id = “[string]“ op0=”[string]” op1=”[string]”
    <fu use_fu = “[string]”/>
    <cycle_matrix execute = "1"/>
    <format>...</format>
  </target>
  ...
</transfer>
```

The attribute id of the node target is the name of the transfer instruction. The attribute use_fu of the fu node describes which functional unit will carry out the transfer. The cycle_matrix describes the behavior of the instruction in the pipeline. The format clause shows how the instruction will be emitted.

Here is an example of STR that stores one register to memory.

```
<target id = "STR" op0 = "mem" op1 = "allExceptPC">
  <fu use_fu = "ALU"/>
  <cycle_matrix execute = "1"/>
  <format> STR {op1}, [{op0}, +{op2}]</format>
</target>
```

Chapter 6 XADML

In this chapter we show the new constructions in ADML that this work has contributed. It extended original ADML to be able to specify a real world processor, i.e. ARM9E processor. ADML with its extension is called XADML.

6.1 cycle_matrix

The clause cycle_matrix describes the reservation table and usage of resources of an instruction in the pipeline. The different stages of the pipeline are specified with the number of clock cycles it uses in each stage. In case of delay slots, those are specified with a latency clause. For more information about this clause see Section 5.9.1 and 7.1.4.

6.2 Clause: test

The test clause contains a standard XML construct called CDATA, specified as ![CDATA[...]], that could contain any block of C code. The construct is used to evaluate simple mathematical expressions (logical and arithmetical), as the one in the example below:

```
<test><![CDATA[ const_offset_12 <= 4095]]></test>
```

If the test clause is true it returns 1 to the condition clause.

6.3 Clause: Logical and

The logical and clause is used to bind test clauses together in a condition clause.

```
<and>
  <test>...</test>
  <test>...</test>
  ...
</and>
```

6.4 Clause: Logical or

The logical or clause is used to bind test clauses together in a condition clause.

```
<or>
  <test>...</test>
  <test>...</test>
  ...
</or>
```

6.5 Clause: condition

Apart from the mapping of operands to residences (a register in a register class, or memory), a target instruction could have other constraints, such as a maximum value constraint for constants. This is checked via a condition clause, containing logical C expressions. If the condition clause is evaluated to true, the covering is possible.

As an example, the following specification examines if a constant value (offset in this case) fits into a 12-bit immediate field in an instruction:

```
<condition>
  <test><![CDATA[
    ({const_offset_12}->syms[0]->u.value >= 0)]]>) &&
    ({const_offset_12}->syms[0]->u.value <= 4095)]]>
  </test>
</condition>
```

6.6 Clause: format

The format clause tells the emitter how to print out the assembler instructions. The string in the clause is written as it is, but variables within curly brackets are replaced by their register or constant value. Escape sequences are written just as in C, for instance a newline is written as \n and the token “ is written as \”.

Here is an example:

```
<format>LDM IA {op0}, \{{1},{2},{3}\}</format>
```

The format line above corresponds to the output:

```
LDM IA R0, {R3, R4, R7}
```

6.7 Clause: Macro-or

The or clause is an extension of ADML and it is there only for compressing the size of the specification file (see the example below). The clause means that OPTIMIST should generate one pattern for each pinst (an IR-node) that appears in an or clause. The compression rate is almost equal to the number of pinst in an or clause. If a pattern has more than one or clause the compression rate is equal to the product of the number of pinst in each or clause. Example:

In the ADDLSL pattern specification, OPTIMIST chooses one pinst from each or clause, creating eight different ADDLSL patterns. Alternatively, we could specify in the case of ADDLSL, eight stand-alone patterns. The compressed pattern looks as follows:

```
<pattern id = "ADDLSL">
  <node>
    <or>
      <pinst id = "ADDI4"  arity = "2" op = "..."/>
      <pinst id = "ADDU4" arity = "2" op = "..."/>
      <pinst id = "ADDF4" arity = "2" op = "..."/>
      <pinst id = "ADDP4" arity = "2" op = "..."/>
    </or>
        <kid nr = "0" id = "left_op"/>
        <kid nr = "1">
          <node>
            <or>
              <pinst id = "LSHI4"  arity = "2" op = "..."/>
              <pinst id = "LSHU4"  arity = "2" op = "..."/>
            </or>
                <kid nr = "0" id = "shift_op"/>
                <kid nr = "1" id = "step"/>
          </node>
        </kid>
  </node>
</pattern>
```

Chapter 7 ARM9E Processor Specifications

This chapter is divided into two parts. First we describe how we created specifications for the ARM mode of the processor, and in the second part for the Thumb mode.

Both specification files can be found on the OPTIMIST homepage[1].

7.1 ARM Mode Specification

By applying the ADML sections and rules in Chapter 5 for the target architectures instruction set described in the ARM Reference Manual [3] or in Chapter 4, “ARM9E”, the main parts of the specification can be written. We used the extensions of Chapter 6 and specified some parts of the specification file that standard ADML could not cover.

The specification is presented in order of increasing complexity. We start with specifying functional unit and issues part, followed by the register and residence class part. The last part is the instruction and pattern part, which is the larger part of the specification file.

7.1.1 Functional Units and Issue Width

Section 4.1 described the ARM9E processor as a single-issue processor, i.e., only one instruction can be issued at a given time. Though, the processor exploits parallelism due to its pipeline where at most five different instructions can be fetched, decoded, executed, read from memory and written to the register file at the same time. The issue width specification looks as follows:

```
<architecture>
  <omega>1</omega>

  ...
```

Since all instructions use the ALU in ARM9E (even the multiplication unit to some extent, see Section 4) and there is only one such unit in the processor, the specification of this functional unit is as follows:

```
<funits>
  <fu id = "ALU" occupation = "1" latency = "1"/>
</funits>
...
```

Both the occupation and latency delays are set to one clock cycle. In general, the functional unit determines the time for each target instruction that is executed on it. In the ARM9E architecture, this is no longer true. The occupation and latency may vary depending on the registers used in an instruction and the instruction itself. Since there is only one functional unit, this information must be specified elsewhere. Therefore the pipeline and the number of execution cycles will be specified in each instruction and pattern instead. For the ARM9E, the functional unit part was thus not relevant and removed from the specification.

1. www.ida.liu.se/~chrke/optimist/

7.1.2 Registers, Residence Classes and Constants

The register section is simple. The 16 user registers are specified with names from the ARM Architecture Reference Manual [3] and a length of 32 bits. Since some multiplication and multiplication-accumulation instructions operate on halves of registers, 32 register halves have to be specified from the 16 registers. This is done by using aliases.

Residence classes and constants are created when studying the instruction set and writing the instruction part of the specification in the following manner:

If an instruction uses a different amount of clock cycles in the execution stage for any registers depending on which registers it uses, then the register file has to be divided into sets. Each register that is used as an operand for an instruction and gives the same number of clock cycles in the execution stage for this instruction belongs to the same register class.

Example: The instruction ADD uses one or three clock cycles in the execution stage depending on the register that serves as destination register. If the destination register is the PC it takes three clock cycles, otherwise one. For ARM9E, the location of all other operands for any instruction, i.e., in any of the 16 registers does not contribute to additional clock cycles. This gives us two register classes called PC (containing the PC only) and allExceptPC (contains the remaining 15 registers). The specification part of the residence classes of ARM9E is depicted below:

```
<residences>
  <residence id = "PC">
    <reg id = "PC"/>
  </residence>
  ...
  <residence id = "allExceptPC">
    <reg id = "R0"/>
    <reg id = "R1"/>
    <reg id = "R2"/>
    <reg id = "R3"/>
    <reg id = "R4"/>
    <reg id = "R5"/>
    <reg id = "R6"/>
    <reg id = "R7"/>
    <reg id = "R8"/>
    <reg id = "R9"/>
    <reg id = "R10"/>
    <reg id = "R11"/>
    <reg id = "R12"/>
    <reg id = "SP"/>
    <reg id = "R14"/>
  </residence>
  ...
</residences>
```

We identify additional residence classes: all is a register class with all sixteen registers, mostly used for base registers for many store and load instructions. The register classes halfB and halfT, representing the register halves of the registers R0,...,R14, are used for various multiplication and multiplication-accumulate instructions. The halfB register class contains the bottom (bit 0 to 15) of the registers and halfT contains the top (bit 16 to 31) of the registers.

The constants are treated as a residence much like the memory modules, but with a limited size specified by the number of bits. A constant refers to an immediate field in an instruction. They are specified as residence classes, one constant in each class.

7.1.3 Instructions and Patterns

The last part of the specification contains the instructions and the patterns[1].

All the instructions in the ARM instruction set can be divided into different groups depending on the instruction type, as mentioned in Section 4.6. The different groups are:

- Data-processing operations.
- Load and Store Word or Unsigned Byte.
- Miscellaneous Loads and Stores.

Some instructions do not have any special addressing modes, i.e. an instruction that takes operands from registers and produce the result in a register, such as the multiplication instruction MUL. Another type of instruction consists of instructions that take only one parameter, an immediate value or a register, such as an unconditional jump or a branch. So there are two more groups:

- Branch instructions.
- Multiplication instructions and Swap instructions.

From the description of how instructions and patterns from each group have been specified, the reader should be able to understand how the rest of the instructions from a given group have been specified.

7.1.4 General ARM Instruction Properties

Before going into the detailed description of instructions, we present the general ARM instruction specification features that apply to the whole of the specifications of the instruction part. The specification of the IR-node ADDU4 is used as an example:

```
<instruction id = "ADDU4" op = "4406">
  <target id = "ADD" op0 = "all" op1 = "allExceptPC" op2 = "all" use_fu = "ALU">
    <cycle_matrix execute = "1"/>
  <format> ADD {op1},{op0},{op2}</format>
  </target>
</instruction>
```

Since all instructions in the instruction set spend one clock cycle in following pipeline stages: fetch, decode, memory and write_back, those stages are not present in the cycle_matrix rows, only the execution stage is specified.

For various instructions with delay slots, such as load and multiplication instructions, we model delays with the latency clause. For instance, the LDR instruction, which has one delay slot, has the following cycle_matrix.

```
<cycle_matrix execute = "1">
  <latency L = “1”/>
</cycle_matrix>
```

1. In the instruction set part, a pattern covers a sub-DAG set of nodes of LCC-IR with a single target instruction.

In the following sections we show a general specification of a binary data processing operation and explain how the first group of instructions was implemented.

7.1.5 Data-Processing Operations

By showing a general specification of a binary data processing operation that corresponds to the ADD, AND, EOR, ORR and SUB instructions, the implementation of this group is explained. The unary instructions MOV and MVN are specified in a similar manner, apart from the fact that they have only one operand. For instructions that do not have their equivalence in IR-nodes, such as BIC, and all the data processing operations when applying a shift to their second operand, a pattern has to be specified. We will take a look at the ADDLSL pattern in this section.

We show how binary and unary instructions mentioned above are specified using a generic instruction BINARY_OP. The binary data-processing instruction, BINARY_OP, is specified with one specification for each of the first two addressing modes. Therefore there will be one specification where the second operand is an immediate value, and one where the second operand resides in a register. The letter `X` stands for any type: `I` (signed integer), `U` (unsigned integer), `F` (float) and `P` (pointer) in the specification.

```
<instruction id = "BINARY_OPX4" op = "...">
  <target id = "BINARY_OP" op0 = "allExceptPC" op1 = "all" op2 =
 "const_data_immediate">
    <fu  use_fu = "ALU"/>
    <cycle_matrix execute = "1"/>
    ...
    <format> BINARY_OP {op0},{op1},#{op2}</format>
  </target>
  <target id = "BINARY_OP" op0 = "allExceptPC" op1 = "all" op2 = "all">
    <fu  use_fu = "ALU"/>
    <cycle_matrix execute = "1"/>
    <format> BINARY_OP {op0},{op1},{op2}</format>
  </target>
</instruction>
```

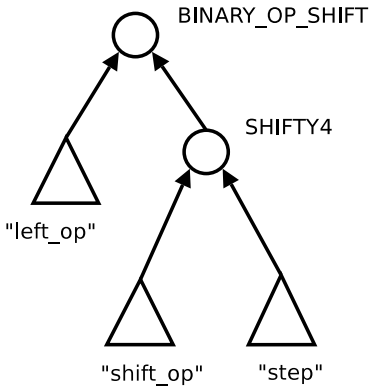
In case the program counter is the destination register, the BINARY_OP instruction takes two additional clock cycles in the execution stage. Therefore we add the following versions (common parts with previous specifications are left out for readability purposes and the differences are highlighted). It is only the number of clock cycles in the execution stage that differs between the two cases.

```
<instruction id = "BINARY_OPX4" op = "...">
  <target id = "BINARY_OP" op0 = "allExceptPC" op1 = "all" op2 =
 "const_data_immediate">
   ...
  </target>
  <target id = "BINARY_OP" op0 = "PC" op1 = "all" op2 = "const_data_immediate">
     ...
     <cycle_matrix execute = "3"/>
     ...
  </target>
  <target id = "BINARY_OP" op0 = "allExceptPC" op1 = "all" op2 = "all">
  ...
  </target>
  <target id = "BINARY_OP" op0 = "PC" op1 = "all" op2 = "all">
      ...
      <cycle_matrix execute = "3"/>
      ...
```

```
  </target>
</instruction>
```

Since X corresponds to four different types, there must be one copy of each BINARY_OP variant for each of these types.

But a BINARY_OP instruction can also be performed with its second operand shifted, as mentioned in Section 4.6.1. The shift and binary operation are executed in the same clock cycle. There is no IR-node that encodes a binary operation and a shift together. For that we use the pattern construction to first specify the operation in terms of sub-DAGs, and then we connect it to an ARM9E target instruction. The pattern for BINARY_OP_SHIFT, a BINARY_OPX4 node with a SHIFT node (LSHY4, LSRY4, ASRY4, RORY4 or RRXY4) is depicted in Figure 7.1. The letter Y is an abbreviation (for either I or U types). Leaf nodes of a pattern are represented by triangular shaped nodes that correspond to operands of the pattern. The names written in lower-case within quotation marks are references to nodes, used later in the instruction part to bind pattern operands to instruction operands.




```
<pattern id = "BINARY_OP_SHIFT">
  <node>
    <or>
      <pinst id = "BINARY_OPX4" arity = "2" op = "..."/>
    <or>
        <kid nr = "0" id = "left_op"/>
        <kid nr = "1">
          <node>
            <or>
              <pinst id = "SHIFTY4"  arity = "2" op = "..."/>
            </or>
                <kid nr = "0" id = "shift_op"/>
                <kid nr = "1" id = "step"/>
          </node>
        </kid>
  </node>
</pattern>
```

Figure 7.1 Textual and graphical representation of the pattern for a binary data operation with one of its operands shifted.

The first child <kid nr = "0" id = "shift_op"/> of the SHIFTY4 IR-node, referenced as “shift_op” is the value that needs to be shifted by the number of steps, specified by child <kid nr = "1" id = "step"/>, referenced as “step”. By using references, we can identify the operands of the pattern BINARY_OP_SHIFT. Below we show the binding between BINARY_OP_SHIFT pattern of Figure 7.1 and target instruction BINARY_OP of the instruction set section.

```
<pattern id = "BINARY_OP_SHIFT">
  <ptarget id = "BINARY_OP" op0 = "allExceptPC">
    <op id="add_left_op">
      <id>allExceptPC</id>
    </op>
    <op id="step">
      <id>const_shift_imm</id>
    </op>
    <op id="shift_op">
      <id>allExceptPC</id>
    </op>
```

```
    <fu use_fu = "ALU"/>
    <cycle_matrix execute = "1"/>
    <condition>...</condition>
    <format> BINARY_OP {op0}, {left_op}, {shift_op}, LSL #{step}</format>
  </ptarget>
</pattern>
```

The left operand[1] of the root node is referenced by the name "left_op". It is expected to be in any register of register class allExceptPC, i.e., the destination is set to any register, except PC. The instruction takes only one cycle in the execution stage of the pipeline as specified in the cycle_matrix.

This combined BINARY_OP and SHIFT instruction can also take the value determining the shift step from a register instead of using a constant. The tree pattern for this variant is the same as for the previous case. The ptarget for the BINARY_OP with register specified shift differs slightly from the one with an immediate value specifying shift step, and the BINARY_OP_SHIFT instruction is extended as follows:

```
<pattern id = "BINARY_OP_SHIFT">
  <ptarget id = "BINARY_OP" op0 = "allExceptPC">
  ...
  </ptarget>
  <ptarget id = "BINARY_OP" op0 = "allExceptPC">
  ...
  <op id="step">
    <id>allExceptPC</id>
  </op>
  ...
  <cycle_matrix execute = "2"/>
  <format> BINARY_OP {op0}, {0}, {shift_op}, SHIFT {step}</format>
  </ptarget>
</pattern>
```

The step reference for this variant is expected in the residence class allExceptPC, instead of being an immediate value of the instruction. This addressing mode takes two clock cycles in the execute stage, as described in the cycle_matrix. The format string is different since we take the value of register {step} to determine the number of shifts.

In the final specification, there are nine different variants of BINARY_OP instructions in the instruction set part and five patterns in the patterns part.

One part left out intentionally from the first BINARY_OP variant remains to be explained. Variants using an immediate value must assure that the immediate value actually fits into the limited space of the target instruction word. The omitted part of BINARY_OP corresponds to the C function testConstant({step}) within test and condition clauses:

```
<condition>
    <test>testConstant({step})</test>
</condition>
```

1. i.e. kids[0] in LCC notation.

The function is as follows:

```
int testConstant (int constantToCheck)
{
  int ones = 255;       //First eight bits are set to 1.
  int leftShifts = 0;   //Number of steps to leftshift
  int shiftedValue = 0; //The shifted value
  int number11 = 3;
  int number1111 = 15;
  int number111111 = 63;

  while (leftShifts <= 30)
   {
     if (leftShifts <= 24)
       {
         shiftedValue = ones << leftShifts;
       }
     if (leftShifts == 26)
       {
         shiftedValue = (ones << leftShifts) + number11;
       }
     if (leftShifts == 28)
       {
        shiftedValue = (ones << leftShifts) + number1111;
       }
     if (leftShifts == 30)
       {
         shiftedValue = (ones << leftShifts) + number111111;
       }
     if (leftShifts > 30)
       {
         printf("error");
       }
     if (constantToCheck == (constantToCheck & shiftedValue))
      {
        return 1;
      }
        leftShifts = leftShifts + 2;
   }
  return 0;
}
```

The function checks if the constant step consists of a bit pattern that has its one-valued bits in a row of maximum length eight bits, somewhere in the word. In case this is true, the function returns 1, and the constant can be written as an immediate value (a number of eight bits, rotated by an even number of steps through the instructions word, see Section 4.6.1). The following numbers can be used as immediate values for data processing operations. Observe that the first three series correspond to the case when the eight-bit number has been rotated and a part of it is at the end of the instruction word, and the other half in the beginning:

$$[0\ldots63]\cdot 2^{26}+[0\ldots3]$$

$$[0\ldots15]\cdot 2^{28}+[0\ldots15]$$

$$[0\ldots3]\cdot 2^{30}+[0\ldots63]$$

And for the general case:

$$[0\ldots255] \cdot 2^{[0\ldots24]}$$

During code generation, OPTIMIST evaluates the expressions in the test construction for each matched pattern. If the expression is evaluated to true, then the matching is accepted, and the instruction appended to the current slot (see details in [5]). Otherwise the instruction is not accepted, discarded, and OPTIMIST tries other matching possibilities.

The constant used in the BINARY_OP instruction is a 5-bit immediate holding the number of steps to shift the second operand. The condition part requires the constant to be a number between 0 and 31.

The unary instructions can also be combined with any shift. The only difference between them is the fact that the root node in their patterns has only one child. It looks as the tree in Figure 7.1, without the left_op node.

7.1.6 Load and Store Word or Unsigned Byte

The second group of instructions to be specified are the instructions of the Load and Store Word or Unsigned Byte Group (see Table 4.3). The general specification called LOAD will be used in this section (to specify both LDR and LDRB). Graphs of the corresponding patterns show the structure of LOAD for each of its addressing modes, see Figure 7.2 and Figure 7.3.

The LOAD instruction has nine addressing modes (see Section 4.6.2), which need many more pattern specifications, since the offset can be both added and subtracted and the shift, in three out of nine modes, can be of six different types. The letter N in the graphs in Figure 7.2 and Figure 7.3 means either one or four (bytes).

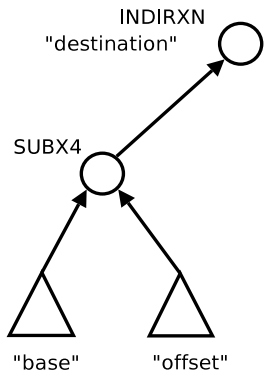
The LDR instruction in this group can incur one processor stall if the immediately following instruction tries to access the loaded value. This is specified with a latency clause.

```
<pattern id = "LDR_ADD_OFFSET">
        ....

        <cycle_matrix execute = "1">
          <latency l = "1"/>
        </cycle_matrix>
        ...
</pattern>
```

The LDRB instruction has two delay slots.

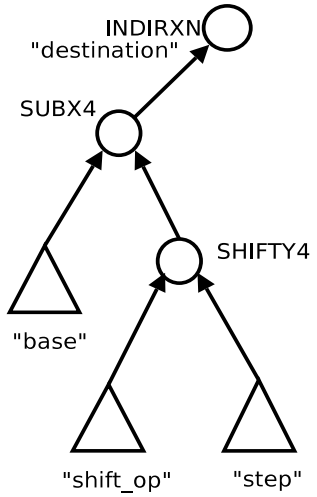
The first two modes can be matched using the same pattern specification. The pattern is depicted in Figure 7.2 with its ADML textual representation.

Figure 7.3 shows the third mode, where a register value is shifted and then subtracted from the base register.




```
<pattern id = "LOAD_SUB_OFFSET">
  <node id = "destination">
    <pinst id = "INDIRXN" arity = "1" op = "..."/>
      <kid nr = "0">
        <node>
          <pinst id = "SUBX4" arity = "2" op = "..."/>
            <kid nr = "0" id = "base"/>
            <kid nr = "1" id = "offset"/>
          </node>
        </kid>
  </node>
</pattern>
```

Figure 7.2 Textual and graphical representation of LOAD pattern, when subtracting the content of a register or an immediate value from the base register.




```
<pattern id = "LOAD_SUB_SHIFTED_LSL_OFFSET">
  <node id = "destination">
    <pinst id = "INDIRXN" arity = "1" op = "..."/>
      <kid nr = "0">
        <node>
          <pinst id = "SUBX4" arity = "2" op = "..."/>
            <kid nr = "0" id = "base"/>
              <kid nr = "1">
                <node>
                  <pinst id = "LSHY4" arity = "2" op = "..."/>
                    <kid nr = "0" id = "shift_op"/>
                    <kid nr = "1" id = "step"/>
                </node>
              </kid>
        </node>
      </kid>
  </node>
</pattern>
```

Figure 7.3 Textual and graphical representation of LOAD pattern, when subtracting a left shifted value from the base register.

The offset can also be added to the base register, so there are two more patterns just as those in Figure 7.2 and Figure 7.3 but with ADDX4 nodes instead of SUBX4 nodes. It is not possible to put both the ADDX4 and the SUBX4 in the same `or` clause and bind them to one instruction specification only since OPTIMIST does not keep track on which pinst has been chosen in an or clause.

The shift can be a right shift, an arithmetic right shift, a rotation or a rotation with extension. This means that there are 24 patterns for LOAD only for the three offset modes.

The three pre-indexed patterns are depicted in Figure 7.4 and Figure 7.5. The ASGNX4 node shows how the base register is updated before the value is loaded from the new address into “destination”. The nodes referenced by “address” and “inc_address” are not used in the instruction part of the instruction, but are there only as internal references in the pattern specification. This allows us to cover DAGs.

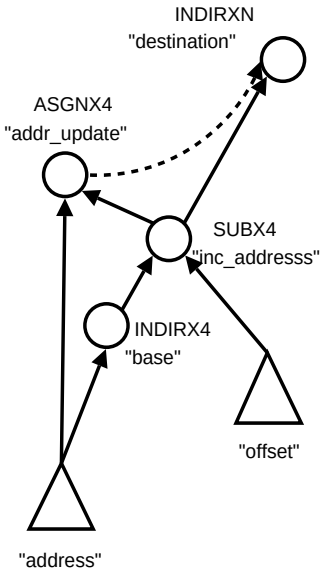
The instruction below is connected to the pattern description presented in Figure 7.4.

```
<pattern id = "LOAD_SUB_OFFSET_PRE">
  <ptarget id = "LOAD">
    <op id ="destination">
      <id>allExceptPC</id>
    </op>
    <op id = "base">
      <id>all</id>
    </op>
    <op id = "offset">
      <id>const_offset_12</id>Figure 6.6
    </op>
    <fu use_fu = "ALU"/>
    <cycle_matrix execute = "1">
      <latency l = "1"/>
    </cycle_matrix>
    <condition>
          <test><![CDATA[
               ({const_offset_12}->syms[0]->u.value >= 0)]]>) &&
               ({const_offset_12}->syms[0]->u.value <= 4095)]]>
          </test>
    </condition>
    <format> LDR {destination}, [{base}, #-{offset}]!</format>
  </ptarget>
</pattern>
```

The test clauses check if the value of the node referenced by “offset“ fits into a 12-bit immediate value.

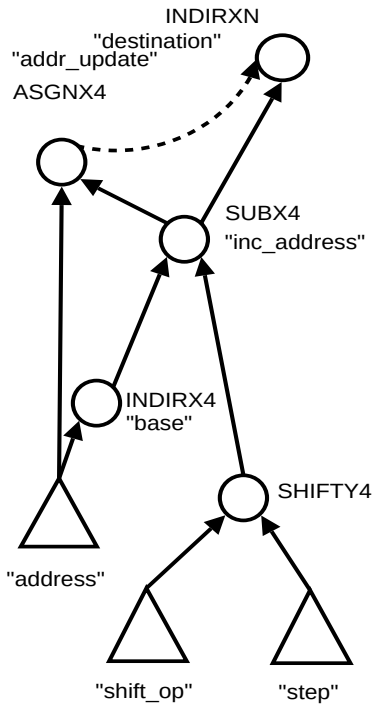
The pre-indexed modes can also add or subtract the offset from the base register, and the shift in Figure 7.4 can be of any type, which leads to another 24 patterns for LOAD with the pre-indexed modes.

The last three post-indexed modes are depicted in Figure 7.6 and Figure 7.7. They are also 24 variants, and sums up to 72 patterns in total for the LOAD instruction. The data dependencies arrow (doted) in Figure 7.6 and Figure 7.7, shows that the base register is updated (addr_update) before the value is loaded (destination).



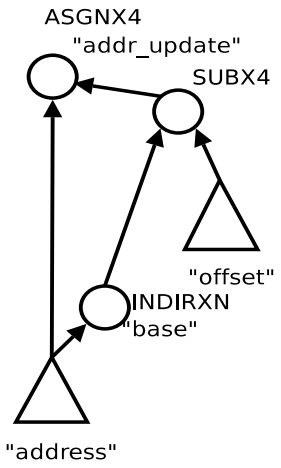

```
<pattern id = "LOAD_SUB_OFFSET_PRE">
  <node id = “destination”>
    <pinst id = "INDIRXN" arity = "1" op = "4165"/>
      <kid nr = "0" id = "inc_address"/>
  </node>
  <node id = “addr_update“>
    <pinst id = "ASGNX4" arity = "2" op = "4149"/>
      <kid nr = "0" id = "address"/>
      <kid nr = "1" id = "inc_address">
        <node>
          <pinst id = "SUBX4" arity = "2" op = "4421"/>
            <kid nr = "0" id = "base">
              <node>
                <pinst id = "INDIRX4" arity = "1" op = "4165"/>
                  <kid nr = "0" id = "address"/>
              </node>
            </kid>
            <kid nr = "1" id = "offset"/>
        </node>
      </kid>
  </node>
</pattern>
```

Figure 7.4 Textual and graphical representation of LOAD pattern, when subtracting a register or immediate value to the base register.



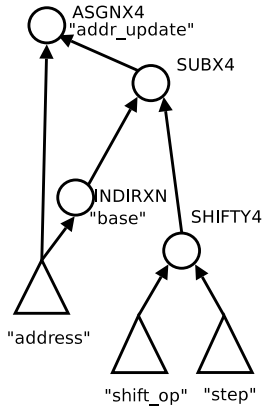

```
<pattern id = "LOAD_SUB_SHIFTED_LSL_OFFSET_PRE
  <node id = “destination>...</node>
  <node id = “addr_update“>
    <pinst id = "ASGNX4" arity = "2" op = "..."/>
      <kid nr = "0" id = "address"/>
      <kid nr = "1" id = "inc_address">
        <node>
          <pinst id = "SUBX4" arity = "2" op = "..."/>
            <kid nr = "0" id = "base">
              <node>
                <pinst id = "INDIRX4" arity = "1" op = "..."/>
                  <kid nr = "0" id = "address"/>
              </node>
            </kid>
            <kid nr = "1">
              <node>
                <pinst id = "LSHY4" arity = "2" op = "..."/>
                  <kid nr = "0" id = "shift_op"/>
                  <kid nr = "1" id = "step"/>
              </node>
            </kid>
        </node>
      </kid>
  </node>
</pattern>
```

Figure 7.5 Textual and graphical representation of LOAD pattern, when subtracting a register specified shift from the base register.




```
<pattern id = "LOAD_SUB_OFFSET_POST">
  <node id = “addr_update”>
    <pinst id = "ASGNX4" arity = "2" op = "..."/>
      <kid nr = "0" id = "address"/>
      <kid nr = "1">
        <node>
          <pinst id = "SUBX4" arity = "2" op = "..."/>
            <kid nr = "0" id = "base">
              <node>
                <pinst id = "INDIRX4" arity = "1" op = "..."/>
                  <kid nr = "0" id = "address"/>
              </node>
            </kid>
            <kid nr = "1" id = "offset"/>
        </node>
      </kid>
  </node>
</pattern>
```

Figure 7.6 Textual and graphical representation of LOAD pattern, when subtracting a register or immediate value from the base register.




```
<pattern id = "LOAD_SUB_SHIFTED_LSL_OFFSET_POST">
  <node id = “addr_update”>
    <pinst id = "ASGNX4" arity = "2" op = "..."/>
      <kid nr = "0" id = "address"/>
      <kid nr = "1">
        <node>
          <pinst id = "SUBX4" arity = "2" op = "..."/>
            <kid nr = "0" id = "base">
              <node>
                <pinst id = "INDIRIX4" arity = "1" op = "..."/>
                  <kid nr = "0" id = "address"/>
              </node>
            </kid>
            <kid nr = "1">
              <node>
                <pinst id = "LSHY4" arity = "2" op = "..."/>
                  <kid nr = "0" id = "shift_op"/>
                  <kid nr = "1" id = "step"/>
              </node>
            </kid>
        </node>
      </kid>
  </node>
</pattern>
```

Figure 7.7 Textual and graphical representation of LOAD pattern when using the post-indexed mode and subtracts a left shifted value from the base register.

7.1.7 Miscellaneous Loads and Stores

The third group (see Table 4.5) containing store and load instructions for signed or unsigned half-words and bytes are very much alike the group described in the previous section. Miscellaneous Loads and Stores lack the possibility to add or subtract a shifted offset value to the base register, which leads to a lower number of patterns that have to be specified for this group. The instruction STORE_MISC will be used as generic

name for instructions STRB and STRH. The LDRSB and LDRH and LDRSH are specified much the same as the LOAD instruction in the previous section.

The patterns for the addressing modes offset, pre-indexed and post-indexed for this group and their corresponding specifications are shown in Figure 7.8, Figure 7.9 and Figure 7.10. Since the patterns do not require the "offset" sub-DAG to be a register or an constant, one pattern is sufficient for both cases. Each pattern corresponds to two instructions in the instruction part, one for immediate offset, one for register offset. This leads to a number of 6 patterns and 12 instruction specifications in total. A check is performed in the instruction specification to see if the constant is 8 bit long.

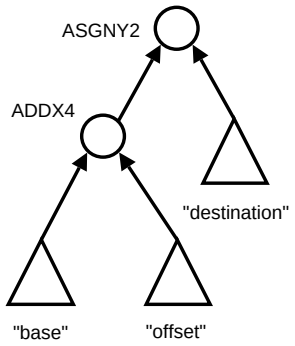
```
<!-- The instruction: STORE_MISC <Rd>, [<Rn>,#+<offset>] -->
<pattern id = "STORE_MISC_ADD_OFFSET">
  ....
  <condition>
    <and>
      <test><![CDATA[ {const_offset_8} >= 0]]></test>
      <test><![CDATA[ {const_offset_8} <= 255]]></test>
    </and>
  </condition>
  </ptarget>
  <format> STORE_MISC {destination}, [{base}, #+{const_offset}]</format>
</pattern>
```

OPTIMIST can choose the other STORE_MISC variant if the constant value is greater than 255 or negative. In case a register holds the offset, the format clause is replaced by: `<format> STORE_MISC {destination}, [{base}, +{const_offset}]</format>`

As mentioned in the ARM9E chapter, there is no separate STRSH instruction, and because of this, the STRH pattern matches both ASGNU2 and ASGNI2 nodes.

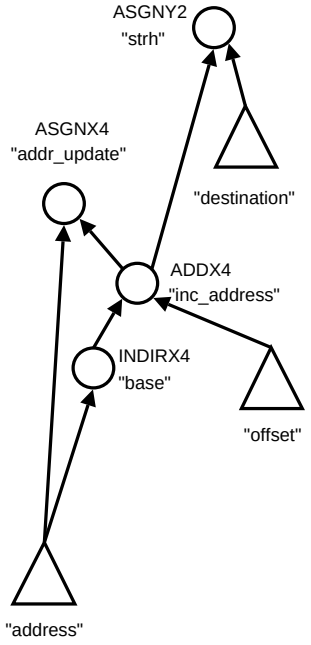
The latency for the load instruction in this group is two clock cycles if the register loaded is accessed in the following instruction, thus the delay slot is specified as follows:

```
<!-- The instruction: LDRH <Rd>, [<Rn>,#+<offset>]! -->
<pattern id = "LDRH_ADD_OFFSET_PRE">
  ...
  <cycle_matrix execute = "1">
    <latency l = "2"/>
  </cycle_matrix>
  ...
</pattern>
```



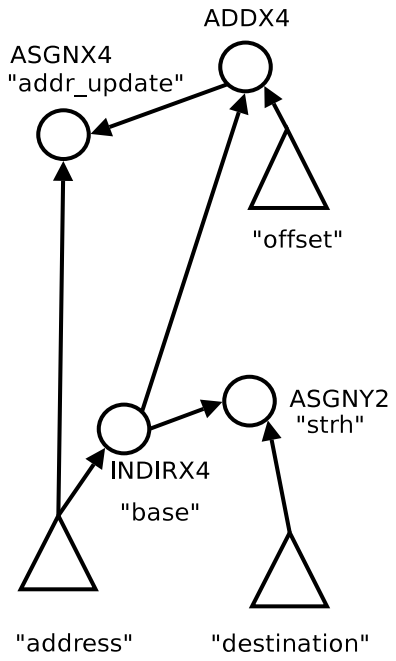

```
<pattern id = "STORE_MISC_ADD_OFFSET">
  <node>
    <pinst id = "ASGNY2" arity = "2" op = "..."/>
      <kid nr = "0">
        <node>
          <pinst id = "ADDX4" arity = "2" op = "..."/>
            <kid nr = "0" id = "base"/>
            <kid nr = "1" id = "offset"/>
        </node>
      </kid>
      <kid nr = "1" id = "destination"/>
  </node>
</pattern>
```

Figure 7.8 Textual and graphical representation of STORE_MISC pattern. It calculates the address by adding an immediate value or a register value to the base register.




```
<pattern id = "STORE_MISC_ADD_OFFSET_PRE">
  <node id = “strh”>
    <pinst id = "ASGNX2" arity = "2" op = "..."/>
      <kid nr = "0" id = "inc_address"/>
      <kid nr = "1" id = "destination"/>
  </node>
  <node id = “addr_update”>
    <pinst id = "ASGNY4" arity = "2" op = "..."/>
      <kid nr = "0" id = "address"/>
      <kid nr = "1" id = "inc_address">
        <node>
          <pinst id = "ADDX4" arity = "2" op = "..."/>
            <kid nr = "0" id = "base">
              <node>
                <pinst id = "INDIRX4" arity = "1" op = "..."/>
                  <kid nr = "0" id = "address"/>
              </node>
            </kid>
            <kid nr = "1" id = "offset"/>
          </node>
        </kid>
  </node>
</pattern>
```

Figure 7.9 Textual and graphical representation of STORE_MISC pattern using pre-indexed offset. It updates the base register with the new address value first, then performs the store.




```
<pattern id = “STORE_MISC_ADD_OFFSET_POST”>
  <node id = “strh”>
    <pinst id = "ASGNY2" arity = "2" op = "..."/>
      <kid nr = "0" id = "base">
        <node>
          <pinst id = "INDIRXF4" arity = "1" op = "..."/>
            <kid nr = "0" id = "address"/>
        </node>
      </kid>
      <kid nr = "1" id = "destination"/>
  </node>
  <node id = “addr_update”>
    <pinst id = "ASGNX4" arity = "2" op = "..."/>
      <kid nr = "0" id = "address"/>
      <kid nr = "1">
        <node>
          <pinst id = "ADDX4" arity = "2" op = "..."/>
            <kid nr = "0" id = "base">
              <node>
                <pinst id = "INDIRX4" arity = "1" op = "..."/>
              </node>
            </kid>
            <kid nr = "1" id = "offset"/>
        </node>
      </kid>
  </node><ddep src = "strh" dest = "addr_update"/>
</pattern>
```

Figure 7.10 Textual and graphical representation of STORE_MISC pattern that calculates the address by adding an immediate value or a register value to the base register. The base register is updated with this value afterwards.

7.1.8 Branch Instructions

LCC has seven IR-nodes corresponding to different branches. The simplest one is the unconditional JUMP. The node has only one child, which holds the label where to jump. The other six branch types are:

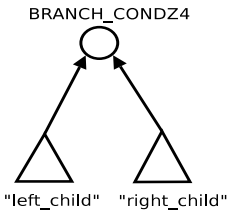
- EQ: branch if equal.
- GE: branch if greater than or equal.
- GT: branch if greater than.
- LE: branch if less than or equal.
- LT: branch if less than.
- NE: branch if not equal.

In this section, the general specification of pseudo BRANCH_COND corresponds to all branches listed above.

For BRANCH_COND, each node has two children, holding the left- and right-part of a logical expression. The node has an entry into the symbol table that is a label to which it should jump if the condition is true. Only the two children are specified in the pattern:

```
<pattern id = "BRANCH_COND">
  <pinst id = "BRANCH_CONDX4" arity = "2" op = "..."/>
    <kid nr = "0" id = "right_child"/>
    <kid nr = "1" id = "left_child"/>
</pattern>
```

ARM9E has one instruction called B, that performs the different jumps depending on how the flags in the program status register (the NZCV-flags) are set. The instruction CMP compares two values and updates the flags. Figure 7.11 describes the instruction the BRANCH_COND pattern is bound to, and its graphical representation.




```
<pattern id = "BRANCH_COND">
  <ptarget id = "CMP">
    <op id="right_child">
      <id>all</id>
    </op>
    <op id="left_child">
      <id>all</id>
    </op>
    <fu use_fu = "ALU"/>
    <cycle_matrix execute = "4"/>
    <condition>
      <and>
        <test><![CDATA[{top}->syms[0].x.name <= 33554428]]></test>
         <test><![CDATA[{top}->syms[0].x.name >= -33554432]]></test>
      </and>
    </condition>
    <format> CMP {right_child},{left_child}\n
                BRANCH_COND {{top}->syms[0].x.name}</format>
  </ptarget>
</pattern>
```

Figure 7.11 Textual representation of the instruction BRANCH_COND and graphical representation of its pattern.

The two test clauses check if the label fits into the immediate value of the branch instruction. Symbol entries in the IR-nodes are accessed by the top.SYM0.X.NAME expression.

Here we touch the issue of matching a single IR-node with two instructions. In OPTIMIST, optimality issue is based on the fact that the IR-level is low enough. In case of the branch instruction in ARM, this is no longer the case. Here we may loose optimality of the generated code. This does not matter as long as we are concerned with basic blocks. Beyond the basic block level OPTIMIST still does not guarantee optimality.

7.1.9 Multiplication Instructions

The various multiply-accumulate instructions are all written as patterns, and the multiply variants are written as instructions only. The MUL and MLA instruction will be used as example in this part, starting with MUL.

```
<instruction id= "MULI4" op = "4565">
  <!-- Common multiplication -->
  <target id = "MUL" op0 = "allExceptPC" op1 = "allExceptPC"
                        op2 = "allExceptPC" use_fu = "ALU">
    <cycle_matrix execute = "2"/>
  <format> MUL {op0},{op1},{op2}</format>
  </target>
  <target id = "MOV" op0 = "allExceptPC" op1= "allExceptPC"
                        op2= "allExceptPC" use_fu = "ALU">
    <cycle_matrix execute = "1"/>
    <condition>
      <and>
        <test>0 == (log2(op1) - trunc(log2(op1)))</test>
        <test>log2(op1) < 32</test>
      </and>
    </condition>
  <format> MOV {op0},{op2}, LSL #{log2(op1)}</format>
  </target>
  <target id = "MOV" op0 = "allExceptPC" op1= "allExceptPC"
                        op2= "allExceptPC" use_fu = "ALU">
    <cycle_matrix execute = "1"/>
    <condition>
      <and>
        <test>0 == (log2(op2) - trunc(log2(op2))</test>
        <test>log2(op2) < 32</test>
      </and>
    </condition>
  <format> MOV {op0},{op1}, LSL #{log2(op2)}</format>
  <target id = "MOV" op0 = "allExceptPC" op1 = "allExceptPC"
                        op2= "allExceptPC" use_fu = "ALU">
    <cycle_matrix execute = "2"/>
      <condition>
        <test>0 == ((log2(op1)) - trunc(log2(op1)))</test>
      </condition>
  <format> MOV {op0},{op2}, LSL {log2(op1)}</format>
  </target>
  <target id = "MOV" op0 = "allExceptPC" op1 = "allExceptPC"
                        op2= "allExceptPC" use_fu = "ALU">
    <cycle_matrix execute = "2"/>
      <condition>
        <test>0 == (log2(op2) - trunc(log2(op2)))</test>
      </condition>
  <format> MOV {op0},{op1}, LSL {log2(op2)}</format>
</instruction>
```

The instruction MULI4 contains five different target specifications. The first one is a simple multiplication. The remaining four implement the multiplication as a MOV with left shift (that is faster and may not incur processor stalls, i.e., has no latency clause). This is only possible when at least one of the operands is a power of two. The test clauses <test>0 == (log2(op1) - trunc(log2(op1)))</test> check if this is the case. If the number of shifts is smaller than 32, the shift can be specified using an immediate value, and thus save one register. Otherwise, a register holds the number of shifts. There are four target specifications since any of the two operands may be a power of two. The {log2(op1)} expression in the format clauses has to be evaluated to a number, and it is done when the assembler code is emitted. OPTIMIST replaces all such expressions with their evaluated number.

In all, this group is specified with 23 instructions, and 13 of them are bound to 13 patterns.

7.1.10 Transfer

The transfer part is written by simplifying the STR, STRB, LDR, LDRB, STRH, LDRH, LDRSH, LDRSB, STM and LDM instructions, and specifying them as instructions without patterns, since they are not supposed to be matches to IR-nodes, but to move data from different residence classes, i.e., registers and memory units.

A version of LDR from the transfer is as follows.

```
<target id = "LDR" op0 = "all" op1 = "allExceptPC">
  <fu use_fu = "ALU"/>
  <cycle_matrix execute = "1">
    <latency l = "1"/>
  </cycle_matrix>
  <format> LDR {op1}, [{op0}, #+{op2}]</format>
</target>
```

For the transfer instructions, only two operands are used, showing sources and destination of the moved dated. The cycle_matrix information is needed to show the implication on pipeline stage and latency information.

7.2 The Thumb Specification

The three instruction groups for the Thumb specification are specified much in the same way as in the ARM specification, but with a lower number of instructions and patterns since the addressing modes of Thumb are a subset of the ARM addressing modes. An instruction can only operate on two registers, or one register and one immediate value, and produces the result in a register. This cuts down the length of the specification file enormously compared to the size of the ARM mode specification. Since no new features had to be specified for Thumb, the derivation of this specification is not a part of this report.

Chapter 8 Test

In this chapter we evaluate OPTIMIST and the processor specification files against a commercial C/C++ compiler for ARM9E (included in IAR Workbench [14]) and a publicly accessible IBurg back-end written for ARM7 for LCC (called LCC-ARM in this chapter).

8.1 Testbenches

Test cases were chosen from various applications in the MediaBench [11] and Mibench [12] testbenches. Every test case is a small segment of code corresponding to an IR-DAG with about 20 IR-nodes. The code segment was cut manually from its original application and put into a new main function and tested by the IAR compiler, LCC-ARM and OPTIMIST. The IAR compiler was set to produce medium optimized code and optimized for time. Test case 1 was taken from the dijkstlarge.c file (network, MiBench), test case 2 from patricia.c, (network, MiBench) test cases 3, 4, 5, 6 from jctrans.c (jpeg-6a, MediaBench), test case 7 from edges.c (epic, MediaBench) and test cases 8, 9, 10 from collapse_ortho_pyr.c (epic, MediaBench).

8.2 Results

The number of clock cycles for the output assembler code was calculated using IAR's simulator and the result was also checked by counting clock cycles by hand. The results are given in Table 8.1 for ARM mode and Table 8.2 for Thumb mode. The numbers in the 3rd **-** 6th column mean the level of optimization chosen in the IAR work bench: 0 = none, 1 = low, 2 = common subexpression elimination and 3 = common subexpression elimination, code motion and static clustering.

Since LCC only performs common subexpression elimination, a comparison between the IAR 2 and OPTIMIST is the fairest comparison. In ARM mode, OPTIMIST produces equally fast code for test cases 2, 3, 6, 7, 9 and 10. By inspecting the assembler code, one can see that OPTIMIST manages to group multiplication and addition nodes together into MLA operations. It performs similarly with data operations and shifts or load and store with shifted offset (in ARM mode).

One reason for the longer execution times of code produced by OPTIMIST could be the fact that major parts of the ARM specification file could not be used, including the pre- and post-addressing modes for the store and load instructions. Currently the ADML parser cannot handle their pattern descriptions, nor could the LDM/STM instructions be implemented (see Section 9.2).

Another thing, that is most likely to be the crucial point for the difference between OPTIMIST's and IAR Workbench's results is the shape of the intermediate representation. There is no reason to believe that the DAGs LCC produces for OPTIMIST looks exactly the same as the intermediate representation IAR Workbench uses. Another difference is the fact that OPTIMIST cannot check if data has to be stored to memory (using STR instructions) or could be stored in a register (using MOV). This is quite obvious in the 4th test case (in ARM mode). OPTIMIST that is conservative stores and loads partial computations while IAR Workbench uses MOV instructions instead (moving values between registers). If the information needed for solving this problem is provided in the C code it is propagated into LCC IR of ADDR-nodes, in the field

sclass (storage class). Currently OPTIMIST does not consider the information and it is left for future work.

The last column in Table 8.1 shows the number of clock cycles used by LCC-ARM's solutions to the test problems. LCC-ARM is simple LCC using a back-end written for ARM7. The most interesting aspect of this test is the fact that both OPTIMIST and LCC-ARM uses the same intermediate representation. The back-end for LCC-ARM is an earlier ARM version, that uses a three stages pipeline, and also has a slight smaller instruction set. But those missing instructions that ARM9E has but ARM7 does not, are not used in any of the solutions from either IAR or OPTIMIST, so the comparison is still fair regarding the instructions set. Why LCC-ARM uses many more clock cycles in all cases depends first and foremost on the fact that LCC-ARM introduces many more read-after-write hazards than OPTIMIST, but OPTIMIST also cover up some groups of instructions better by using patterns. The results were as expected, since LCC-ARM does not solve the three code-generation problems in one single phase as OPTIMIST does, but in three separate phases.

In Thumb mode, OPTIMIST produced faster code for test case 2, equal for test cases 1, 6, 9 and 10. Unlike the ARM specification, the Thumb specification contains a much greater percentage of the instruction set than of the ARM mode. From the previous section we know, that this comparison might not be completely fair.

Since there is no Thumb back-end for LCC available, only OPTIMIST and IAR Workbench were evaluated for Thumb mode.

Table 8.1 Number of clock cycles in ARM mode.

Tests	Clock cycles used					
Arm mode	OPTIMIST	IAR 0	IAR 1	IAR 2	IAR 3	LCC-ARM
Test 1	15	16	16	13	11	18
Test 2	14	16	15	14	14	18
Test 3	14	20	20	14	10	20
Test 4	20	18	15	12	12	32
Test 5	15	14	9	9	9	24
Test 6	15	31	31	15	11	16
Test 7	13	31	31	13	10	16
Test 8	15	16	14	11	11	25
Test 9	13	19	19	13	10	17
Test 10	11	16	14	11	9	19

Table 8.2 Number of clock cycles in Thumb mode.

Tests	Clock cycles used				
Thumb mode	OPTIMIST	IAR 0	IAR 1	IAR 2	IAR 3
Test 1	16	18	18	16	14
Test 2	14	24	21	20	20
Test 3	18	20	20	16	12
Test 4	20	31	19	19	19
Test 5	15	16	10	9	9
Test 6	17	31	31	17	13
Test 7	16	31	31	14	11
Test 8	15	14	14	11	11
Test 9	14	19	19	14	11
Test 10	11	15	13	11	9

Chapter 9 Conclusion

9.1 Evaluation of the Project

The results from Section 8.2 show that OPTIMIST performed quite well in ARM mode (under the circumstances with only parts of the specification file in use in ARM mode and different IR-representations to start with) and better in Thumb mode.

OPTIMIST produced faster code than LCC-ARM for all test programs, as expected.

Since it took some time before I could use OPTIMIST to test my specification files it was very hard to know if the constructs and patterns I had written were useful or not. The best thing would have been if I could have tested them under the time of writing the specifications. I also believe that some patterns have to be modified or added to cover some parts I have not thought about before, but this is left as future work.

It would be interesting to see how well it could have worked with the entire ARM specification and with the same IR.

The specification files need more constructs that can reduce the redundancies in the files and makes them smaller and easier to read. The or-construct is the only one used now, but one might think of various macros used for describing the shift options in those instructions that can shift its second operand. Over and above this, the new ADML constructs covered their corresponding problems. See Section 9.2 for more notes about further extensions.

9.2 Future Work

The most obvious extension of this project would be to add the instructions left out from the specifications. Instructions such as RSB (reverse subtraction) and ADC (add with carry) could be specified with so-called pragma clauses in the source program. This means that low-level instructions can be coded in the source file. There are no rules for how the statements following the pragma clause should look like, and any low-level instruction can be specified as convenient for LCC as possible. For instance, the following pragma would be recognized by an extended version of LCC, creating a RSB IR-node, later mapped by new instruction and pattern specifications in the specification files:

```
...
int a, b;
...
# pragma RSB b a LSL #2; meaning b - a LSL #2;
...
```

It would be possible to extend the LCC front-end instead, and creating new operations for ADC, for instance, but then the programmer has to learn those new constructs, therefore the usage of pragmas is better and follows the standard.

The problem regarding the MOV and STR instructions (mentioned in section 8.2) has to be solved.

Another way to improve the specifications file would be to introduce switch clauses in XADML. Now, every pattern that uses any sort of shift has to be specified one by one. A switch clause could make it possible to use one pattern, only for all shifts. Still, OPTIMIST has to extract all variants using the clause, but it would reduce the size of the specification file, limit possible typing errors and make the file easier to read.

Another future improvement would be to automate the generation or part of the generation of the ADML specification file, which also will limit possible typing errors.

The Load and Store Multiple instructions (STM) and (LDM) could not be specified due to the fact that the registers loaded have to be in increasing order. Since OPTIMIST does not regard this order when assigning a register class to an operand, STM and LDM could not be implemented. How this problem could be solved is not known at the time of writing.

Instructions that use more than one register as destination register introduce a problem for OPTIMIST when matching IR-nodes in bottom-up fashion. The destination nodes in such an instruction have two outgoing edges and it is supposed to be only one. The solution to this problem is not known.

Glossary

ADML: Architecture Description Mark-up Language. An XML language, used to specify the characteristics of the processor. See *XADML*.

ARM Ltd.: A Hardware Company, manufacturing various processors in the ARM-family, ARM9E is one of them.

ARM9E: Embedded processor with two instructions sets, a five stage pipeline, 16 general purpose registers, manufactured by ARM Ltd. (The specification files are written for this processor.)

AST: Abstract syntax tree.

Basic Block: A section of consecutive instructions in the control flow of the program. The section has only one entrance and one exit, and no branches in between.

cycle matrix: Description of reservation of resources when an instruction is carried out in the pipeline of the processor. The entity is known as reservation table in the literature.

DSP: Digital Signal Processor. An integrated circuit whose main task is to process a digital input signal and produce an output signal (usually connected to an Analog-Digital Converter, and Digital-Analog Converter).

Functional Unit: Part of the processor core that can carry out different operations.

IR-node: Intermediate Representation-node.

LCC: Lean C Compiler. A retargetable C compiler, designed at AT&T Bell Laboratories and Princeton University for the ANSI C programming language.

VLIW: Abbreviation for Very Large Instruction Word. It is a processor containing more than one functional unit. Rather than one instruction at a time, this processor can execute n instructions at a time, where n is the issue width of the processor.

XADML: Extended version of ADML provided by this project. The original specification language was enriched with new constructs to specify the ARM9E embedded processor.

References

[1] Christopher W. Fraser, David R. Hanson. *A Retargetable C Compiler*: Design and Implementation. Addison-Wesley, February 2003.

[2] Christopher W. Fraser, David R. Hanson, *The lcc 4.x Code-Generation Interface*, Microsoft Research, July 2001.

[3] David Seal, *ARM Architecture Reference Manual*, second edition, Addison-Wesley, 2001.

[4] Steve Furber, *ARM System-on-chip architecture*, second edition, Addison-Wesley, 2000.

[5] Andrzej Bednarski, *A Dynamic Programming Approach to Optimal Code Generation for Irregular Architectures*, Department of Computer and Information Science, Linköping, Sweden, 2002.

[6] *ARM9E-S™ Technical Reference Manual*, (Rev 2), ARM Limited, 2002.

[7] *ARM946-S System-on-chip enhanced processor*, (Rev 1), ARM Limited, 2000.

[8] *Product Comparison, ARM CPU Products, Performance of the ARM9TDMI™ and ARM9E-S™, cores compared to the ARM7TDM™ core*. ARM Limited 2000.

[9] Simon Segars, *The ARM9 Family - High Performance Microprocessors for Embedded Applications*, ARM Ltd.

[10] Rainer Leupers. *Code Optimization Techniques for Embedded Processors*. Kluwer Academic Publishers, 2000.

[11] Chunho Lee, Miodrag Potkonjak, William H. Mangione-Smith, MediaBench. http://cares.icsl.ucla.edu/MediaBench/ (2005-01-13).

[12] Todd Austin, Richard Brown, Matthew Guthaus , Trevor Mudge, Jeff Ringenberg, MiBench, University of Michigan, http://www.eecs.umich.edu/mibench/ (2005-02-28).

[13] Silvina Hanono, Srinivas Devadas, *Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator*, Proceedings of the 35th annual conference on Design Automation Conference, pages 510-515, pub-ACM, San Francisco, California, United States, 1998.

[14] *ARM IAR Embedded Workbench IDE User Guide*, IAR Systems, Ninth Edition, 2004

[15] David G. Bradlee, Tobert R. Henry, Susan J. Eggers, *The Marion System for Retargetable Instruction Scheduling*. Department of Computer Science and Engineering, FR-35. University of Washington, Seattle, Washington.

[16] Peter Marwedel, Gerd Goossens, *Code Generation for Embedded Processors*, Kluwer, 1995.