Final thesis

On SIMD code generation for the CELL SPE processor

by

Magnus Pettersson

LIU-IDA/LITH-EX-A--10/039--SE

2010-09-20
Final Thesis

On SIMD code generation for the CELL SPE processor

by

Magnus Pettersson

LIU-IDA/LITH-EX-A--10/039--SE

2010-09-20

Supervisor: Mattias Eriksson
Examiner: Christoph Kessler
**Språk**

- □ Svenska/Swedish
- ☒ Engelska/English
- □ 

**Rapporttyp**

- □ Licentiatavhandling
- □ Examensarbete
- □ C-uppsats
- □ D-uppsats
- □ Ovrig rapport
- □ 

**URL för elektronisk version**

- http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-ZZZZ

**Titel**

- **SIMD kodgenerering för CELL SPE processorn**
- **On SIMD code generation for the CELL SPE processor**

**Författare**

- Magnus Pettersson

**Sammanfattning**

- This thesis project will attempt to answer the question if it is possible to gain performance by using SIMD instructions when generating code for scalar computation. The current trend in processor architecture is to equip the processors with multi-way SIMD units to form so-called throughput cores. This project uses the CELL SPE processor for a concrete implementation. To get good code quality the thesis project continues work on the code generator by Mattias Eriksson and Andrzej Bednarski based on integer linear programming. The code generator is extended to handle generation of SIMD code for 32bit operands. The result show for some basic blocks, positive impact in execution time of the generated schedule. However, further work has to be done to get a feasible run time of the code generator.

**Nyckelord**

- **SIMD, code generation, integer linear programming**
Abstract

This thesis project will attempt to answer the question if it is possible to gain performance by using SIMD instructions when generating code for scalar computation. The current trend in processor architecture is to equip the processors with multi-way SIMD units to form so-called throughput cores. This project uses the CELL SPE processor for a concrete implementation. To get good code quality the thesis project continues work on the code generator by Mattias Eriksson and Andrzej Bednarski based on integer linear programming. The code generator is extended to handle generation of SIMD code for 32bit operands. The result show for some basic blocks, positive impact in execution time of the generated schedule. However, further work has to be done to get a feasible run time of the code generator.
Acknowledgments

I would like to extend my appreciation and thanks to my examiner Christoph Kessler, for presenting me with the opportunity to work on a challenging and interesting project. My deepest thanks to my supervisor Mattias Eriksson, for taking time out of his schedule to help assist and guide me. A big thanks to my family and friends for their support, understanding and patience with me during stressful periods. Finally, a big thank you to my fellow peers in room 3D:436 where I spent much of time. Thanks again for creating such a pleasant and interesting work environment.
Contents

1 Introduction ................................................. 1
  1.1 SIMD-Instructions ..................................... 1
  1.2 Overview of the CELL architecture and CELL SPE instructions . 1
  1.3 Compiler phases ...................................... 4
  1.4 Problem formulation ................................. 4
  1.5 Thesis outline ....................................... 5

2 Integrated SIMD code generation ......................... 7
  2.1 Eriksson’s integer linear programming formulation .......... 7
  2.2 Steps towards generation of SIMD instructions .......... 8

3 Preprocessing of the data flow graph .................... 11
  3.1 SIMD sets ........................................... 11
  3.2 Overlapping SIMD sets .............................. 12
  3.3 Operands of SIMD sets .............................. 13

4 Modelling the CELL SPE ................................ 15
  4.1 A first attempt ...................................... 16
  4.2 Current implementation ............................. 16

5 Extending the ILP formulation ............................ 19
  5.1 Current SIMD ILP formulation ....................... 19
    5.1.1 Solution Variables ............................. 19
    5.1.2 Constraints .................................. 20
  5.2 Schedule slots and SIMD instructions ................ 30

6 Evaluation .................................................. 31
  6.1 Hand made graphs ................................... 31
  6.2 Real world experiments ............................. 35

7 Limitations and Future Work .............................. 37
  7.1 Limitations ......................................... 37
  7.2 Future Work ....................................... 38
8 Related Work
  8.1 ILP for code generation ........................................ 39
  8.2 SIMD code generation ........................................... 39

9 Conclusions .......................................................... 41

Bibliography ............................................................ 43

A Solutions
  A.1 Solutions for graph in Figure 6.1 ............................... 47
  A.2 Solutions for graph in Figure 6.2 ............................... 55
  A.3 Solution for graph in Figure 6.3 ................................ 60

B Output from step 1 of the code generator, for the data flow graph in Figure 6.1 65
Chapter 1

Introduction

In this chapter the main problem of this thesis is formulated, some background information is presented and narrowing the scope of the problem is done throughout.

1.1 SIMD-Instructions

Today many modern processors include SIMD instructions to increase computation throughput. Two example application areas that can benefit from SIMD instructions are computer graphics and digital-signal-processing. A SIMD (single instruction multiple data) instruction is an assembler instruction that when issued, performs the same operation on several operands. At time of issuing a register-to-register SIMD instruction, its operands must reside in the same registers and be aligned within the registers properly. The illustration in Figure 1.1 show how three additions can be done simultaneously, when operands are aligned in the same register and a SIMD add instruction is issued. SIMD instructions to load and store operands to and from memory exist too. These instructions require the operands to reside in contiguous memory locations, with processor specific alignment constraints.

1.2 Overview of the CELL architecture and CELL SPE instructions

The CELL processor can have many different configurations. For example the configuration found in a Sony Playstation 3 contains [24]:

- One dual-threaded Power Processor Element (PPE).
- Eight Synergistic Processor Elements (SPEs). At least seven are operational but only six available to a programmer, when the system is running a Linux based OS.
Non-SIMD:
ADD R1 R2 RT1
ADD R3 R4 RT2
ADD R5 R6 RT3

SIMD:
ADD S1 S2 ST1
provided that:
Content of R1 R3 and R5 reside in register S1.
Content of R2 R4 and R6 in S2.
Operands are aligned within registers S1 and S2.

Polygons represent instructions, a square represent one register and circles operands in the registers. The example assume a 3-way SIMD architecture. R1 - R6, S1 and S2 are source registers. RT1 - RT3 and ST1 are destination registers.

Figure 1.1. Illustration of SIMD computation

- A Memory Interface Controller that connects the processor to 256MB off-chip XDR RAM.
- An Element Interconnect Bus, that connects the internal components of the processor.
- Two Input/Output interfaces that connect the processor to external peripherals such as a graphics card.

A program for the CELL processor that utilizes the SPEs has to go through a series of steps to get the SPEs executing. For readers familiar with GPGPU computing, the process is a bit similar to executing a NVIDIA CUDA-program. A CELL-program starts executing on a PPE that transfers data and program instructions to the requested SPEs. Once data and program instructions are uploaded to the SPEs, the PPE signals the SPEs to start executing. It is these programs that run on the SPEs this thesis is concerned with. A SPE is a throughput-oriented co-processor. The design of the SPE-architecture was focused on area and power efficiency[12]. Each SPE has the following specification:

- 256kb local memory, with non-translated local address space. A memory line is 128bit wide.
- 128 x 128bit-wide general-purpose registers.
- Superscalar execution with in-order issue of up to two instructions per clock-cycle, to two fully pipelined execution pipelines.

Most assembler instructions operate on full 128bit registers. Load and store instructions fetch/save entire 128bit memory-lines at once. The scope of this thesis is narrowed to only consider programs using 32bit operands. A register and memory-line is therefore thought of being built up by four slots. If data is placed strategically in memory, we can do up to four similar 32bit operations at once. Some instructions require that one of the operands reside at a certain slot within a register. An example is shift-left \texttt{shl rt,ra,rb}, shifts the content of register \texttt{ra}, according to the count in bits 26 to 31 of register \texttt{rb} and places the result in register \texttt{rt}. Instructions that require this alignment constraint on one of its operands are also handled by the current implementation. Only computation and data movement instructions are considered. Code for communication between different SPEs and PPEs is not considered.

An instruction we will pay extra attention to is the shuffle instruction \texttt{shufb rd,ra,rb,rc}. This is the instruction we will use to move data between registers and different placements within registers when operands need to be gathered. The instruction has four registers as input. The second and third registers will for us contain operands that we want to put into the output register \texttt{rd}. The fourth register \texttt{rc} contain a bit pattern to select which bytes from the registers \texttt{ra} and \texttt{rb} we want in, and where within, the register \texttt{rd}. See Figure 1.2 for an illustration of this rather powerful instruction. For further details see the documents 'Synergistic Processor Unit Instruction Set Architecture' and 'SPU Assembly Language Specification', both can be found on IBM's homepage developerWorks [15].

![Figure 1.2. Illustration of the shuffle instruction adapted from [19]. As seen; with the hexadecimal value \texttt{0a} in byte-slot 0 of register \texttt{rc}, will the value of the byte with index ten in register \texttt{ra} be copied into slot 0 of register \texttt{rd}. The convention used is that the first byte has index zero](image-url)
1.3 Compiler phases

This section introduces some basic compiler terminology. For a more in-depth coverage, we refer to a good text book such as the 'Dragon Book' [2].

A compiler is a program that translates programs from one computer language, the source language, into another language called the target language.

A compiler can at the very top level of abstraction be divided into two components. The first component is called the front end. The front end mostly deals with analyzing the input program for correctness and generates an intermediate representation IR, that is handed over to the back end. Front end related research to ease generation of SIMD instructions in the back end has been done by IBM [16], among others. However this is not the goal of this thesis.

The back end is the second component of a compiler. The back end synthesizes the target program from the data gathered and handed over by the front end. A part of the back end is called the code generator. It is the code generator that is to be extended with the ability to generate SIMD-instructions.

One usual way to deal with code generation is to divide the work into at least three phases: Instruction selection, Instruction scheduling and Register allocation. These three phases are executed in some predefined order, sometimes several times. The three problems are easier to solve one at a time. Each preceding step has implication on the next step though; this is the motivation behind integrating the three steps. The integrated problem is harder to solve but with better end result, if done right. The code generator we will build upon was first presented in Andrzej Bednarski's Ph.D thesis 'Integrated Optimal Code Generation for Digital Signal Processors' [4] and in the paper by Bednarski and Kessler [5]. The code generator has been extended and improved by Mattias Eriksson in his Licentiate Thesis 'Integrated Software Pipelining' [8]. The code generator integrates these three phases. The code generator is a part of a compiler research framework called OPTIMIST [1]. With this framework a retargetable compiler system is being built. Retargetable means that the final compiler system will be able to generate code for different architectures, given appropriate architecture specifications and program code written in the source language. This means that the compiler system should be able to adapt to new architectures by just adding a new architecture specification, instead of writing a new compiler from scratch. During implementation for this thesis project this was considered but not at high priority. Due to time constraints, in some places SPE specific features had to be added in the current implementation.

Throughout, more general approaches will also be discussed in this thesis.

1.4 Problem formulation

The problem to be solved in this thesis is to, at the basic block level, optimize code for scalar computations. Code can be optimized for different objectives like execution time, code size or even energy usage. This project is concerned with lowering total execution time of a basic block. The architecture of concern is the CELL SPE. Each slave processor’s performance relies heavily on utilizing SIMD
1.5 Thesis outline

instructions [7]. Therefore this project will, in order to gain performance, try to find opportunities where scalar computations can be organized as SIMD instructions. The project is built on top of Mattias Eriksson’s work described in his licentiate thesis "Integrated Software Pipelining" [8]. No front-end aspects of the compiler system are considered.

1.5 Thesis outline

Chapter 1: An introduction to some background information needed to understand the rest of this thesis. Topics covered are SIMD-instructions, the CELL processor and basic compiler information. This projects problem formulation is also stated.

Chapter 2: An introduction to the code generator this project has been built upon. An overview is also given of the work done during the implementation phase of this project.

Chapter 3: A description of the implemented process of finding opportunities for SIMD computation in data flow graphs.

Chapter 4: A description of the model of the CELL SPE that the code generator uses.

Chapter 5: A listing of the variables and constraints in the integer optimization program which is used to generate schedules is provided.

Chapter 6: Some examples of output from the code generator are shown. This chapter is divided into two main sections. One section with hand made graphs which shows the functionality of the code generator. A second section that shows some hints on performance gained, when the code generator is used with data flow graphs from the libavcodec and the DPSTONE benchmark suit.

Chapter 7: In this chapter we will discuss some of implementation limitations.

Chapter 8: In this chapter we present some related work.

Chapter 9: Conclusion of this thesis project.
Chapter 2

Integrated SIMD code generation

In this chapter the integer linear programming (ILP) formulation, from which the thesis started, is introduced and how the formulation is to be modified to enable generation of SIMD instructions. A full description of the formulation can be found in Mattias Eriksson’s Licentiate Thesis [8] and the paper by Kessler et al. [10]. The formulation is implemented as an optimization engine in the OPTIMIST framework. Information about OPTIMIST can be found on the project homepage [1] and in the paper by Bednarski and Kessler [20].

2.1 Eriksson’s integer linear programming formulation

Given a data flow graph \( G \) of a basic block\(^1\) and a processor architecture specification, the ultimate goal of the ILP formulation is to cover the graph \( G' \)'s nodes and edges with instruction patterns such that execution time is minimized. See Chapter 3 for a full description of data flow graphs. The architecture specification includes information about the target processor’s characteristics: number of functional units, issue width, number of registers, instruction latencies, what functional units the instructions use, and instruction patterns. The instruction patterns can be thought of as pairs consisting of an instruction and a set of IR-nodes the instruction matches. The ILP formulation is written in the AMPL modeling language [13]. The data flow graph is a DAG (Directed Acyclic Graph), it is generated in the front end of the OPTIMIST framework. A requirement of the IR, of the data flow graph, is that the IR is low level enough that each pattern model exactly one instruction of the target machine [8].

\(^1\)A basic block is a block of code that contains no jump instructions, except possibly one as the last instruction. Also a basic block contains no jump targets, except possibly to the beginning of the block.
2.2 Steps towards generation of SIMD instructions

We first considered the idea of using register banks and an additional set of SIMD patterns. The slots of a vector register would then belong to different register banks. This attempt progressed rather far. However the number of patterns in the architecture specification became problematic.

Since almost every instruction of the SPE operates on full 128 bit registers, a third set \(^2\) of SIMD-instruction patterns did not seem necessary. The number of variables in the final ILP-problem is influenced by the number of patterns in the architecture specification file. If we can decrease the number of variables in the ILP formulation we will decrease the running time of the ILP-solver substantially, since solving ILP-problems is \(NP\)-hard\(^6\).

An architecture that has any of the following three properties still requires the third set of SIMD-patterns:

1. Separate SIMD and 'regular' SISD (Single Instruction Single Data) instructions. Different instructions will have different mnemonics, they might have different latencies, also they might work on different register sets.

2. Several register banks. We need to know where the operands come from and where the computed result should be stored.

3. Several functional units accepting the same operations, since we need to know on what resource the instruction can be scheduled.

Instead a new approach that does not break registers into register banks is considered. This time the idea is to gather IR nodes that correspond to SIMD

\(^6\)When this thesis project started the existing integer linear formulation partitioned patterns for instructions that are issued into two sets. One for instruction patterns covering single nodes of the graph and one for instruction patterns covering several nodes like multiply-and-add instructions. See Chapter 4 for further details.
instruction patterns. These instructions can operate on several operands in parallel. This is the current implementation and it consists of three steps illustrated in Figure 2.2. In the first step a parser, written in bison, parses the data flow graph and gathers information about the graph. The information is processed and the result of this step is written to a file on disk. This first step is described in Chapter 3. With this new information as input to the second step of the code generator, described in Chapter 5, we use CPLEX to solve an ILP-problem. The third and final step of the code generator is supposed to interpret the mathematical solution from step two.
Chapter 3

Preprocessing of the data flow graph

In this chapter the first step of the proposed code generator is described. This step is concerned with finding opportunities to gather operands and compute several IR-nodes with SIMD-instructions. This step has a data flow graph and a set of IR node operators as input. The set of IR node operators specifies what IR nodes the architecture can execute as SIMD instructions. The data flow graph is an acyclic directed graph $G = (V, E)$, where $V$ is the set of IR-nodes and $E = E_1 \cup E_2 \cup E_m$ are edges between nodes of $V$. The edges in $E_1$ and $E_2$, can be thought of as edges between operators denoting first and second operand respectively. $E_m$ contains edges that represent data dependencies in memory. The data flow graph also contains integer parameters $Op_i$ and $Outdg_i$ that represent operators and out-degree of every IR node $i \in G$.

As output this step computes three sets, each described in its own section of this chapter. The sets are written to a file on disk and used as input when solving the ILP-problem. The first set called $SIMDCandidates$ is the most important set, it contain sets of graph nodes that can be executed as a SIMD instruction. The other two sets are computed to ease formulation of constraints needed in the second step of the code generator. As a prerequisite a parser was written in bison [23]. The parser parses a data flow graph and stores edges and nodes in C++ vectors; these vectors are used to compute the three sets. An example of the output from this step for the graph found in Figure 6.1 on page 32 is found in Appendix B.

3.1 SIMD sets

As a concrete example, suppose a data flow graph contains several nodes for addition. Groups of these addition nodes should be considered to be covered by SIMD instructions. The main idea used is to store sets of the power set of nodes in the data flow graph. These sets are stored in the set $SC (SIMDCandidates)$. 
To create the full power set of the nodes is a waste. An algorithm that computes the sets of the power set with cardinality two, three and four is implemented since the execution unit of the SPE is of a four-way SIMD design. To further reduce the number of candidates, a set is only stored in the SC set if it passes the following tests:

- The cardinality of the set must be 2, 3 or 4.
- The nodes in the set must have the same operator. (Every node has an operator, which is either an addition, multiplication etc.).
- In the data flow graph a path must not exist between any two nodes of a stored set. A path from node a to node b indicates that b depends on a. The transitive closure of the data flow graph is computed and used when determining if a path exists between any two nodes of the set. See the book Introduction to Algorithms [6] for pseudo-code, the algorithm in use is sometimes referred to as Warshalls algorithm.
- Sets containing nodes for shift and rotate are only stored if all nodes share the operand indicating the number of steps to rotate/shift.
- Sets containing Load or Store nodes are only stored if the nodes have memory address nodes as direct predecessor\(^1\). Also these memory address nodes have to point to the same memory line. Verification that the address nodes point to the same memory line has to be done manually at the moment. The verification is done by replacing invalid sets with the empty set.

3.2 Overlapping SIMD sets

Several sets of the SC set, described above, will in most cases have one or more common nodes, as intended by using the power set. The reason is that it will not always be beneficial to cover as many nodes as possible with SIMD-instructions. For example it can happen that, without packing\(^2\) operands, running two add nodes as SIMD and a third directly after saves time. To pack the two operands of the third instruction into the registers of the SIMD operands will cost at least five clock cycles to complete: The packing phase only uses shuffle instructions and one shuffle requires four clock cycles. Since the architecture is fully pipelined, we can schedule the two shuffles in two cycles. Therefore overlapping sets of the set SC must be present and considered in the second step of the code generator. This second computed set, called OS (overlappingSIMD), contains for each set a of SC indexes to other sets in SC that overlap with a in the manner described.

\(^1\)If there is an edge from node a to node b, then a is a direct predecessor of b.
\(^2\)Before a SIMD instruction is issued its operands must reside in the same register. This gathering of operands will be called the pack phase of the SIMD instruction.
3.3 Operands of SIMD sets

This set called OOS (OperandsOfSIMD) also relates sets in SC.

Before a register-to-register SIMD instruction is issued, all operands must reside in the two registers the instruction uses. This is done in the pack phase of a SIMD-instruction. The set is computed to ease formulation of the constraints used to indicate how many shuffles have to be done before a SIMD instruction is scheduled. The set contains for each set $a$ in SC indexes to other sets $b$ in SC, such that there exists a node in $b$ that is an operand of a node in $a$. The set is partitioned into two sets $OPL$ (OperandsOfSIMDL) and $OPR$ (OperandsOfSIMDR). If the set $a$ in $OPL$ contains an index to a set $b$, then the set $b$ contains operands that are left hand side operands of an operator in $a$. $OPR$ is defined analogous, but for right hand side operands of an operator.
Chapter 4

Modelling the CELL SPE

In this chapter the model of the SPE will be presented. The first idea was to leave constraints and variables in the existing ILP formulation as untouched as possible. Instead a more elaborate architecture specification was supposed to be created. However this idea was abandoned because of the complexity the architecture file started to grow into. Instead we considered a new approach, which will be the topic of the second section of this chapter. An illustration of the CELL SPE can be found in figure 4.1.

![Figure 4.1. Illustration of the CELL SPE architecture](image-url)
4.1 A first attempt

The idea used first, as described in section 2.2, was to make as few modifications to the existing ILP-model as possible. The architecture specification was supposed to contain register files connected to artificial functional units. Each 32bit slot of the 128bit registers would be part of one of four register files. The existing formulation did already handle register files. Therefore it looked like a good starting point. However this approach has several flaws. To name three, the number of variables produced by modeling the registers in this way would increase by a factor of four. The number of instruction patterns would also be increased by a factor four. Finally the existing constraints and variables would still have to be modified to some extent.

4.2 Current implementation

After gathering ideas on ways to tackle the problem a new approach was considered. This time more effort was going to be spent on modifying the constraints and variables of the existing ILP formulation. When the project started an almost complete model of the TI-C62x [8] processor existed. To quickly get the project re-started a decision was made to start working from this architecture specification. Necessary modifications like number of registers and number of functional units was changed. Some patterns that represent instructions that do not exist in the SPE instruction set were removed. In the first attempt a set of patterns describing SIMD-instructions was used, but as mentioned in Section 2.2 this set is left out when modeling the SPE. The set is necessary when architectures have different assembler instruction mnemonics for SISD and SIMD instructions. These SIMD-patterns are supposed to be able to cover the sets found in the set "SIMDCandidates", described in Chapter 3. The following are the key points of the current architecture specification:\footnote{Only sets, parameters and variables needed to understand this thesis are presented. For a full description see Eriksson [8]}

- Issue width is set to two since the SPE can, in-order, issue up to two instructions into two execution pipelines. The two pipelines are called ODD and EVEN. The ODD pipeline handles data movement, memory fetch/load and branch instructions. The other pipeline called EVEN handles arithmetic/logical oriented instructions like add, multiply, or, rotate and so on. For the SPE to issue two instructions the first instruction must come from an even address and be routed to the EVEN pipeline and the second instruction must be routed to the ODD pipeline. In the architecture specification the issue width is modeled as a parameter $\omega$.

- An SPE has seven execution units distributed over the two execution pipelines. Therefore the architecture specification contains two 'resource units’ called EVEN and ODD. The resources are modeled by the set $F = \{EVEN, ODD\}$. 
- The register file of the SPE contains 128 entries, each entry is 128bit wide. It is unlikely that all entries will be used in one basic block if not deep loop unrolling is done. In order to reduce the solving time of the code generator we can lower the number of registers in the model, currently it is set to model all 128 entries. As mentioned before, the ILP model contains functionality for multiple register files. The SPE only has one register file called "RA" in the architecture file. Two artificial register files 'CONST' and 'SYMBOL', that model constants and memory locations respectively, are present. These three files are contained in the set \( RS = \{RA, CONST, SYMBOL\} \). The number of register file entries is modeled through the integer parameter \( M_r \) where \( r \in RS \).

- The latencies and resource usage of the patterns are modeled according to the book [24] by Scarpinopo. For instance 32bit integer addition has latency 2 and uses the EVEN execution pipe. A memory load has latency 6 and uses the ODD execution pipe.

- Not every instruction of the SPE is modeled by a pattern, due to project time constraints. Instead we selected data flow graphs of a few basic blocks from "real world" applications, such as a finite impulse response filter and a dot product function. Enough instructions were modeled to cover these graphs. The selected graphs did not test or show the functionality of the implementation satisfactorily. Several graphs, of synthetic basic block examples, have therefore been constructed by hand. The instruction patterns are modeled through a set \( P = P_0 \cup P_1 \cup P_{2+} \) where \( P_1 \) consists of patterns that cover single IR nodes, patterns from \( P_{2+} \) covers multiple IR nodes and patterns from \( P_0 \) covers non-issue instructions like memory address nodes. The patterns in \( P_{2+} \) are used to model assembler instructions like multiply-and-add. Currently no patterns in \( P_{2+} \) are implemented, this is why this set will be given less attention here. All patterns have a set \( B_p \) of generic pattern nodes. Each node \( k \in B_p \) of pattern \( p \in P_0 \cup P_{2+} \) has an associated operator number, stored in the parameter \( OP_{p,k} \). The operator number relates the pattern to operators of IR nodes. The latency information of the instructions corresponding to the patterns is stored in another parameter \( L_p \). The latency information is such that if an instruction corresponding to a pattern \( p \in P \) is scheduled at time slot \( t \), then the result is available at time slot \( t + L_p \).

A binary parameter \( U_{p,f,o} = 1 \) iff the instruction with pattern \( p \in P \) uses resource \( f \in F \) at time step \( o \) relative issue time. To model where operands of patterns are stored we have the set \( PD_r \subset P \) where \( r \in RS \). To model where first and second operand are fetched from, we have the following three sets \( PS1_r, PS2_r \subset P \) where \( r \in RS \). For example the instruction pattern for 32bit addition have entries in \( PS1_{RA}, PS2_{RA} \) and \( PSD_{RA} \), since it can only fetch its two operands from the register file \( RA \) and only output the computed result to the register file \( RA \).
Chapter 5

Extending the ILP formulation

In this chapter the second step of the code generator is described. This step has three files as input, a data flow graph, an architecture specification and the file containing the sets computed in the first step of the code generator. These three files and an additional file with constraints and variables described in Section 5.1 is the final integer linear programming formulation. Once they are parsed and run through AMPL, they make up the final integer linear optimization problem to be solved. The solver in use is CPLEX by ILOG [17].

5.1 Current SIMD ILP formulation

This section describes the constraints and variables used when covering the data flow graph and the SIMD sets with patterns from the architecture file. Ultimately the third step of the code generator should be able to parse the solution variables and derive the final schedule. The constraints restrict value assignment to variables in such a way that a valid and minimal (in execution time) schedule is created. In the existing implementation a script is calling the solver with increasing values of a parameter called $t_{\text{max}}$. This parameter gives the last time slot on which an instruction can be scheduled. A set $T = \{1, 2, 3, ..., t_{\text{max}}\}$ is created. This set defines the available time slots. Until now only parameters and sets of the final ILP formulation have been discussed. The following two paragraphs will introduce the variables and constraints present. These two elements (mostly the number of variables) greatly affect the execution time of the code generator.

5.1.1 Solution Variables

Existing variables The variables that were present in the existing implementation [8] were kept as is and are listed below. All variables are binary
variables. This means that their values are either zero or one in a solution. The variable for composite patterns of the set \( P_{2+} \) was kept because the SPE instruction set contains multiply-and-add instructions. Implementing patterns for such instructions is a topic for future work.

- \( c_{i,k,t} \), which is 1 iff IR node \( i \in V \) is covered by \( k \in B_p \), where \( p \in P \) and the instruction modelled by \( p \) is issued at time \( t \in T \).

- \( w_{i,j,p,k,l} \), which is 1 iff edge \((i,j) \in E_1 \cup E_2\), at time \( t \in T \), is covered by the pattern edge \((k,l) \in EP_{p} \) and \( p \in P_{2+} \). This is used to cover internal edges of composite patterns. The set \( EP_{p} = B_p \times B_p \) is for pattern \( p \in P_{2+} \) the set of edges between generic pattern nodes.

- \( s_{p,t} \), which is 1 iff the instruction modelled by pattern \( p \in P_{2+} \) is issued at time \( t \in T \).

- \( r_{rr,i,t} \), which is 1 iff the value of IR node \( i \in V \) is available in register bank \( rr \in RS \) at time slot \( t \in T \).

**New variables** The following binary variables have been added.

- \( sind_{scs,f,t} \), which is 1 iff the instruction covering the set \( scs \in SC \) of IR nodes, is issued at time \( t \in T \) using resource \( f \in F \).

- \( align_{32,i,t,s} \), which is 1 iff at \( t \in T \) the value computed by IR node \( i \in V \) is aligned within some register or memory location at slot \( s \in \{0,1,2,3\} \).

- \( x_{i,src,dst,t} \), which is 1 iff the computed value of IR node \( i \in V \) is re-aligned from slot \( src \in \{0,1,2,3\} \) to slot \( dst \in \{0,1,2,3\} \) in a new empty register at time slot \( t \in T \). The re-alignment is modelled to use a shuffle instruction. Future work should consider using rotate or shift instructions as an alternative.

- \( shuffDataForSindL_{scs,t,c} \), which is 1 iff a pack phase of the left operands of a SIMD instruction covering \( scs \in SC \), requiring \( c \in \{0,1,2\} \) shuffle instructions, is started at time slot \( t \in T \). At most two shuffle instructions are needed since operands from at most four registers might have to be packed into a new register. Each shuffle instruction can gather operands from two different registers into a third.

- \( shuffDataForSindR_{scs,t,c} \), which is 1 iff a pack phase of the right operands of a SIMD instruction covering \( scs \in SC \), requiring \( c \in \{0,1,2\} \) shuffle instructions, is started at time slot \( t \in T \).

**5.1.2 Constraints**

The constraints, while taking architectural limitations into consideration, put restrictions on the values the variables above can have in a valid solution. The following constraints are present in the current solution:
5.1 Current SIMD ILP formulation

**Instruction Selection** Instruction selection is done when each IR node is covered by a pattern node indicating a SISD operations or part of a SIMD instruction.

\[
\forall i \in V, \\
\sum_{p \in P} \sum_{k \in B_p} \sum_{t \in T} c_{i,p,k,t} + \sum_{f \in F} \sum_{scs \in SC} \sum_{t \in T} \simd_{scs,f,t} = 1
\] (5.1)

The following constraint assure that when a generic pattern node covers an IR node, then both nodes have the same operator number.

\[
\forall i \in V, \forall p \in P, \forall k \in B_p, \forall t \in T, \\
c_{i,p,k,t}(Op_i - OP_{p,k}) = 0
\] (5.2)

Two more constraints are present, they handle composite patterns and have currently no effect on the solution. They are left untouched and can be found in Mattias Eriksson's licentiate thesis [8].

**Instruction Scheduling** Instruction scheduling is done when each selected instruction is allocated to a time slot in the schedule. The selection is done so that no memory, precedence or resource violation occurs. As mentioned earlier we do not have a separate set for SIMD patterns. Instead sets of IR nodes that we know can be scheduled as SIMD instructions are precomputed and can be found in the set \( SC \). The following two constraints assure that these instructions are scheduled at the right execution pipe (functional unit).

\[
\forall scs \in SC: \forall i \in scs, \forall t \in T, \\
\simd_{scs,ODD,t} = 0
\] (5.3)

\[
\forall scs \in SC: \forall i \in scs, \forall t \in T, \\
\simd_{scs,EVEN,t} = 0
\] (5.4)

The set \( SOE \) (SIMDOPSEVEN) is part of the set mentioned as input to the first step of the code generator. It contains operators of instructions that can be scheduled on the EVEN execution pipe of the SPE. \( SOO \) (SIMDOPSODD) contain IR node operators for ODD pipe instructions.

**Resource Allocation** The constraints in this paragraph assure that the EVEN and ODD execution pipelines of the SPE never get overscheduled. In
other words, at each scheduling time slot there is at most one instruction for the EVEN pipe and one for the ODD pipe.

\[
\forall t \in T, \\
\sum_{p \in P_2, o \in T} U_{p,EVEN,o,s_{p,t-o}} + \sum_{p \in P_1, i \in V, k \in B_p, o \in T} U_{p,EVEN,o,c_{i,p,k,t-o}} + \sum_{scs \in SC} \text{simd}_{scs,EVEN,t} \leq 1
\]

(5.5)

\[
\forall t \in T, \\
\sum_{p \in P_2, o \in T} U_{p,ODD,o,s_{p,t-o}} + \sum_{p \in P_1, i \in V, k \in B_p, o \in T} U_{p,ODD,o,c_{i,p,k,t-o}} + \sum_{scs \in SC} \text{simd}_{scs,ODD,t} + \\
\sum_{i \in V, src \in \{0,1,2,3\}, dst \in \{0,1,2,3\}} x_{a_{i,src,dst,t}} + \\
\sum_{scs \in SC} \text{shuffDataForSimd}_{scs,t,1} + \sum_{scs \in SC} \text{shuffDataForSimd}_{scs,t,1} + \\
\sum_{scs \in SC, l \in \{0,1\}} \text{shuffDataForSimd}_{scs,t-l,2} + \sum_{scs \in SC, l \in \{0,1\}} \text{shuffDataForSimd}_{scs,t-l,2} \leq 1
\]

(5.6)

Shuffle instructions make up the packing phases of SIMD instructions. The shuffle instructions must be scheduled on the ODD pipe. When two shuffle instructions are flagged, the inequality 5.6 has to look back one time slot; this is the \(-l\) part in indexing of the variables \text{shuffDataForSimd}_{scs,t-l,2} and \text{shuffDataForSimd}_{scs,t-l,2}. Indices with number of shuffles equal to zero are omitted. Zero shuffles means that no shuffle instructions should be scheduled. No constraints are needed to check that we never exceed the issue width. The execution pipelines are fully pipelined and the SPE has as many pipelines as the issue width. The fact that the pipes are fully pipelined implies that no instruction will stall them.

**Register Values** The operands must be available in a register bank at the time of issuing an instruction.

The following two constraints are used to handle instructions covered by patterns \(p \in P_1\):
5.1 Current SIMD ILP formulation

\[ \forall (i, j) \in E_1, \forall t \in T, \forall r \in RS, \]
\[ r_{rr,t} \geq \sum_{p \in P S_1, \cap P_1} \sum_{k \in B_p} c_{j,p,k,t} \]  \hfill (5.7)

\[ \forall (i, j) \in E_2, \forall t \in T, \forall r \in RS, \]
\[ r_{rr,t} \geq \sum_{p \in P S_2, \cap P_1} \sum_{k \in B_p} c_{j,p,k,t} \]  \hfill (5.8)

The operands of an SIMD instruction \( scs \in SC \) must also be present in a register bank when the instruction is issued. The first Inequality 5.9 handles memory instructions such as load and store of multiple operands in a single instruction. The second Inequality 5.10 handles other operations such as addition, multiplication shift etc. that only work on registers.

\[ \forall (i, j) \in E, \forall t \in T, \]
\[ r_{SYMBOL,t} \geq \sum_{scs \in SC} \sum_{f \in F_{scs}} \sum_{i \in scs} \sum_{Op_i \in SL} \]  \hfill (5.9)

The set \( SL \) (SIMDLOAD) contains operators of IR nodes that correspond to a load instruction. The SYMBOL entry of the set \( RS \) is used to model operands that have to be fetched from memory.

\[ \forall (i, j) \in E, \forall t \in T, r = RS \cap SYMBOL, \]
\[ r_{rr,t} \geq \sum_{scs \in SC} \sum_{f \in F_{scs}} \sum_{i \in scs} \sum_{Op_i \in SL} \]  \hfill (5.10)

A value of an IR node is only available in a register bank if it was just put there or it was available in the previous time step.

\[ \forall i \in V, \forall t \in T, r = RS \cap SYMBOL, \]
\[ r_{rr,t} \leq \sum_{scs \in SC} \sum_{f \in F_{scs}} \sum_{i \in scs} \sum_{Op_i \in SL} \]  \hfill (5.11)

The parameter \( ML_i \) is the latency of the instruction with minimal latency that could cover the node \( i \). It is assumed that this will be the instruction to cover
the set \(scs\). For the SPE this is sufficient: An instruction with higher latency is only scheduled if the functional unit executing the lower latency instruction is fully scheduled while another functional unit executing the higher latency instruction is available. The SPE only has two execution pipelines (functional units) that accept distinct instructions, one for data movement and memory operations and the other for computation. Since a SIMD set, \(scs\) in the constraint, consists of nodes with the same IR operator the choice of the lower latency instruction is sufficient to consider.

\[
∀i ∈ V, ∀t ∈ T,
\]

\[
r_{\text{SYMBOL},i,t} ≤ \sum_{scs ∈ SC} \sum_{f \in F} \text{simd}_{scs,f,t-L_i} + \sum_{p ∈ PD_{\text{SYMBOL}} \cap P_{\text{SIMD}}} \sum_{k ∈ B_p} c_{i,p,k,t-L_p} + r_{\text{SYMBOL},i,t-1}
\]

(5.12)

The set \(SS\) (SIMDSTORE) contains IR node operators matching store instructions.

**Memory data dependences** The following constraint assures that no memory data dependence is violated:

\[
∀(i, j) ∈ E_m, ∀t ∈ T,
\]

\[
\sum_{p ∈ P} \left( \sum_{t_j = 0}^{t} c_{j,p,1,t_j} + \sum_{t_i = t-L_p}^{t_{\text{max}}} c_{i,p,1,t_i} + \sum_{s = 0}^{t} \sum_{scs ∈ SC} \sum_{f \in F} \text{simd}_{scs,f,ts} + \sum_{s = 0}^{t_{\text{max}}} \sum_{scs ∈ SC} \sum_{f \in F} \text{simd}_{scs,f,ts} \right) ≤ 1
\]

(5.13)

**Alignment of operands** Four 32-bit operands can fit into one 128-bit wide register entry of the SPE register file. The four places in a register are called slots. To keep track of where operands are at a certain time \(t \in T\), the align32 variable mentioned above is used.

This first Inequality 5.14 just force alignment information onto the values of each IR node\(^1\)

\[
∀j ∈ V, ∀t ∈ T,
\]

\[
\sum_{s ∈ \{0, 1, 2, 3\}} \text{align32}_{j,t,s} = 1
\]

(5.14)

\(^1\)As futurework modifications, alignment information should only be forced on live variables.
5.1 Current SIMD ILP formulation

The computed value of an IR node $i \in V$ in a SIMD set $scs \in SC$ and the operand node in $E_0$ must have the same alignment.

$$\forall f \in F, \forall (i, j) \in E_0, \forall s \in \{0,1,2,3\}, \forall cs \in SC, \forall t \in T, \quad \text{align}_32_{i,t,s} + \text{align}_32_{j,t,s} + \text{simd}_{scs,f,t} \leq 2 \quad (5.15)$$

The same applies to IR nodes covered by pattern $p \in P$:

$$\forall (i, j) \in E_0, \forall s \in \{0,1,2,3\}, \forall t \in T, \quad \text{align}_32_{i,t,s} + \text{align}_2_{j,t,s} + \sum_{p \in P} c_{i,p,k,t} \leq 2 \quad (5.16)$$

For operands from $E_1$ it is a bit different. Some instructions like shift and rotate transform their operands in $E_0$ (right hand operands) according to the operand in $E_1$ (left hand operands), as long as the operator in $E_1$ is in the preferred slot.

The preferred slot for 32-bit operands is slot 0 in a register of the SPE register file. These instructions are collected in a set called $\text{preferredSlotOperators (PSO)}$.

Other instructions not part of $PSO$ have the same alignment restrictions as the ones in $E_0$. This is modelled by the following two constraints:

$$\forall f \in F, \forall (i, j) \in E_1, \forall s \in \{0,1,2,3\}, \forall cs \in SC, \forall t \in T, \quad \text{align}_32_{i,t,s} + \text{align}_32_{j,t,s} + \sum_{p \in P} c_{i,p,k,t} \leq 2 \quad (5.17)$$

$$\forall (i, j) \in E_1, \forall s \in \{0,1,2,3\}, \forall t \in T, \quad \text{align}_32_{i,t,s} + \text{align}_2_{j,t,s} + \sum_{p \in P} c_{i,p,k,t} \leq 2 \quad (5.18)$$

The next constraint ensures that the computed value of each IR node of a SIMD set has a unique alignment.

$$\forall f \in F, \forall s \in \{0,1,2,3\}, \forall cs \in SC, \forall t \in T, \quad \text{align}_32_{i,t,s} + \text{align}_32_{j,t,s} + \text{simd}_{scs,f,t} \leq 2 \quad (5.19)$$

Some instructions of the SPE require that the operands reside in the preferred slot of the registers worked upon (slot 0). Four such instructions are currently implemented: load, store, shift and rotate. For load and store it is the instructions that take the memory address from a register (indirect addressing) that have this requirement, this is modelled by the inequality (5.20). Indirect
addressing load and store nodes are not part of any set in the set $SC$. Only load and store instructions that have address nodes as predecessors (direct addressing mode) can be found in $SC$. For these indirect address load and stores to be "simdized" further work with pointer analysis must be done. This is to ensure that only load/store nodes pointing to the same memory line are put into a SIMD set. Shift and rotate instructions can be "simdized" as long as all the operators have the same operand in $E_1$. At time of issuing the SIMD instruction, this operand must be aligned at the preferred slot of its allocated register. This is modelled in Inequality (5.21):

$$
\forall (i,j) \in E_1: \forall t \in T, \sum_{s \in \{1,2,3\}} \text{align}_{32,t,s}^i + \sum_{p \in P} \sum_{k \in B_p} c_{j,p,k,t} \leq 1
$$

The sets $LO$ (LoadOperators) and $SO$ (StoreOperators) contain IR-node operator-numbers corresponding to load and store patterns respectively.

$$
\forall (i,j) \in E_1: \forall t \in T, \sum_{s \in \{1,2,3\}} \text{align}_{32,t,s}^i + \sum_{p \in P} \sum_{k \in B_p} c_{j,p,k,t} + \sum_{scs \in SC} \sum_{f \in F} \text{simd}_{scs,f,t} \leq 1
$$

**Packing of operands before scheduling a SIMD** Before a SIMD instruction can be scheduled it has to be verified that all the operands of the instruction are in the same registers. This is done in the pack phase of scheduling a SIMD instruction. The alignment constraints above take care of where, within the register, the operands should go. Two main Inequalities 5.22 and 5.23 are used to flag how many shuffle instructions are needed to pack the operands. The constraint 5.22 handles operands coming from the left side of operators in a scheduled SIMD set and the other 5.23 handles operands coming from the right side. For operands from each side, up to two shuffle instructions might be necessary to gather the operands\(^2\). When no shuffle instructions are needed zero shuffles are flagged. This happen when operands are in the same register and at correct alignment.

\(^2\)A single shuffle instruction can gather operands from two registers into a third.
∀scs ∈ SC,
\[
\sum_{t∈T} \sum_{s∈\{0,1,2\}} \text{shuffDataForSimd}_{scs,t,s} \cdot \text{numOps}_s + \\
\sum_{f∈F} \sum_{t∈T} \sum_{m∈OS_{scs}} \text{simd}_{scs,f,t,m} \cdot \text{BIG} ≥ \\
\sum_{scs_p∈OPL_{scs}} \sum_{t∈T} \sum_{f∈F} \text{simd}_{scs_p,f,t} + \sum_{(i,j)∈E_0} \sum_{p∈P} \sum_{k∈B_p} \sum_{t∈T} c_{i,p,k,t} - \\
\sum_{c∈scs} \sum_{p∈P} \sum_{k∈B_p} \sum_{t∈T} c_{ci,p,k,t} \cdot \text{BIG} \tag{5.22}
\]

The sets $OS$ and $OPL$ are the sets mentioned in Chapter 3. $BIG$ is a large integer number. The right hand side of the inequality counts the number of instructions that have resulted in the operands required by the SIMD set $scs$. Also this side of the inequality subtracts a lot if the nodes of the set $scs$ are not scheduled to execute as a SIMD instruction at all (the minus sign). Since each new instruction puts the computed value in a new register, this side of the inequality will at most be four. The left hand side of the inequality adds two variables: The part coming from $\text{simd} \cdot \text{BIG}$ is greater than or equal to one if an overlapping SIMD set of $scs$ is scheduled, in this case we do not want to have to shuffle for $scs$. If $scs$ is scheduled, then no $scs_m ∈ OS_{scs}$ will be scheduled\(^3\). Therefore the variable $\text{shuffDataForSimd}$ is forced to be one for a scheduled set $scs$. The table $\text{numOps}_s$ contains for each value of $s ∈ \{0,1,2\}$ an integer value $b$. The value $b$ is the maximum number of registers from which $s$ shuffle instructions can pack operands into a single register.

The next constraint works analogously, but it handles packing of operands coming into a SIMD set from the right ($E_1$).

\(^3\)A value can therefore only be computed once.
Extending the ILP formulation

∀scs ∈ SC,

\[
\sum_{t ∈ T} \sum_{s ∈ \{0, 1, 2\}} \text{shuffDataForSimdR}_{scs, t, s} * \text{numOps}_s + \\
\sum_{f ∈ F} \sum_{t ∈ T} \text{simd}_{scs, f, t} * \text{BIG} ≥ \\
\sum_{f ∈ F} \sum_{t ∈ T} \text{simd}_{scs, f, t} + \sum_{(i, j) ∈ E_1} \sum_{s ∈ \{0, 1, 2\}} \sum_{t ∈ T} \sum_{p ∈ P} \sum_{k ∈ B} \sum_{t ∈ T} \text{c}_{ci, p, k, t} * \text{BIG} (5.23)
\]

The set \( OPR_{scs} \) is the set \( \text{OperandsOfSIMD}_R \), that contain operands coming into the SIMD set \( scs \) from the right, discussed in chapter 3. The two constraints above handle the number of shuffle instructions required by the two pack phases of a SIMD instruction. The following four (two for each left and right side operands) assure that a pack phase must be completed before issuing a SIMD instruction. For left and right side operands, we only want one pack phase for each scheduled SIMD instruction:

∀scs ∈ SC,

\[
\sum_{t ∈ T} \sum_{s ∈ \{0, 1, 2\}} \sum_{f ∈ F} \text{shuffDataForSimdL}_{scs, t, s} ≤ \sum_{t ∈ T} \sum_{f ∈ F} \text{simd}_{scs, f, t} (5.24)
\]

∀scs ∈ SC,

\[
\sum_{t ∈ T} \sum_{s ∈ \{0, 1, 2\}} \sum_{f ∈ F} \text{shuffDataForSimdR}_{scs, t, s} ≤ \sum_{t ∈ T} \sum_{f ∈ F} \text{simd}_{scs, f, t} (5.25)
\]

For left and right side operands, the pack phase must have completed before issuing a SIMD instruction:

∀scs ∈ SC: \( s ∈ scs \), \( Op_p \notin SL \), ∀t ∈ T,

\[
\sum_{f ∈ F} \text{simd}_{scs, f, t} - \sum_{s ∈ \{0, 1, 2\}} \sum_{t ∈ T: t ∈ T - \text{lat}_s} \sum_{t ∈ T} \text{shuffDataForSimdL}_{scs, t, s} ≤ 0 (5.26)
\]
\[
\sum_{f \in F} \text{simd}_{scs,f,t} - \sum_{s \in \{0, 1, 2\}} \sum_{t \leq t \leq t - \text{lat}_{s}} \text{shuffDataForSIMDR}_{scs,t,s} \leq 0 \quad (5.27)
\]

The table \( \text{lat}_s \) contain the time taken to complete \( s \in \{0, 1, 2\} \) shuffle instructions.

**Re-Alignment of operands** Two "gadgets" are implemented and used to move data between different slots. Both use the shuffle instruction, one operates on single operands and the other is the one used in the pack phase of a SIMD instruction. The gadget that moves single operands is the one modelled by the \( xa \) variable. Whenever realignment is done through this variable a new register must be allocated. For realignments done in the pack phase of SIMD instructions, the two variables \( \text{shuffDataForSIMDL} \) and \( \text{shuffDataForSIMDR} \) are used. Each pack phase of a SIMD instruction requires that the operands are packed into a new register. New registers are used since it is hard, in the current implementation, to verify that if an existing register were to be re-used, other values should not be overwritten.

The following inequality controls realignment of operands. It should be interpreted as a value have the same alignment as in the previous time slot, if it was not realigned by an instruction of the variable \( xa \) four time slots earlier or was realigned in the pack phase of a SIMD instruction:

\[
\forall dst \in \{0, 1, 2, 3\}, \forall i \in V, \forall t \in T,
\]

\[
\text{align}_{32,t,dst} \leq \text{align}_{32,t-1,dst} + \sum_{src \in \{0, 1, 2\}} \text{xa}_{i,src,t-4} + \sum_{shuff \in \{1, 2\}} \sum_{\text{scs} \in \mathcal{SC}} \sum_{\text{E}_{0}} \text{shuffDataForSIMDL}_{\text{scs},t-\text{lat}_{\text{shuff}},\text{shuff}} + \sum_{shuff \in \{1, 2\}} \sum_{\text{scs} \in \mathcal{SC}} \sum_{\text{E}_{1}} \text{shuffDataForSIMDR}_{\text{scs},t-\text{lat}_{\text{shuff}},\text{shuff}} \quad (5.28)
\]

Three extra inequalities ensure that the value to be moved is available in some register when issuing these data movement instructions. These three constraints are simply forcing the variables \( \text{shuffDataForSIMDL} \), \( \text{shuffDataForSIMDR} \) and \( xa \) to be less than the variable \( r \) for every node and every time slot.

---

\(^4\)A shuffle/shift/rotate instruction has latency 4 and either of these instructions can be used to realign operands flagged by the \( xa \) variable.
5.2 Schedule slots and SIMD instructions

Eriksson’s formulation contain soonest-latest analysis of the nodes in $V$ [8]. This is done since many redundant variables can then be removed. This idea has also been implemented on the sets of $SC$. The schedule slots of a set in $SC$ is currently defined as:

$$\forall scs \in SC,$$

$$SIMDslots_{scs} = (\min_{i \in scs} (\text{soonest}(i))... \max_{i \in scs} (\text{latest}(i))) \tag{5.29}$$

With this set we do no longer have to consider the variable $\text{simd}_{scs,f,t}$ for $t \notin SIMDslots_{scs}$. Also, the pack phases of SIMD instructions are only considered within this schedule slot. This makes the variables $\text{shuffDataForSimdR}$ and $\text{shuffDataForSimdL}$ smaller in a similar way.

The parameters $\text{soonest}(i)$ and $\text{latest}(i)$ can be found described in the paper by Eriksson [9]. They are defined as:

$$\text{soonest}(i) = \begin{cases} 0, & \text{if } |\text{pre}(i)| = 0 \\ \max_{j \in \text{pre}(i)} \{\text{soonest}(j) + L_{\text{min}}(j)\}, & \text{otherwise} \end{cases} \tag{5.30}$$

$$\text{latest}(i) = \begin{cases} t_{\max}, & \text{if } |\text{succ}(i)| = 0 \\ \max_{j \in \text{succ}(i)} \{\text{latest}(j) - L_{\text{min}}(i)\}, & \text{otherwise} \end{cases} \tag{5.31}$$

$$T_i = \{\text{soonest}(i), \ldots, \text{latest}(i)\} \tag{5.32}$$

Where $L_{\text{min}}(i)$ is 0 if the node $i \in V$ may be covered by a composite pattern, or the lowest latency of any instruction $p \in P_i$ that may cover the node $i \in V$ otherwise. Further on, $\text{pre}(i) = \{j : (j,i) \in E\}$ and $\text{succ}(i) = \{j : (i,j) \in E\}$.

The parameters $\text{soonest}$ and $\text{latest}$ are calculated beforehand, and provided as a parameters to the integer linear program when it is started. The constraints above therefore use $i \in V, t \in T_i$ when indexing and quantifying over time, instead of $t \in T$. This reduces the size of the variables $c, w, r, xa, align32$.

Another optimization done is to, for IR nodes, only consider patterns that have matching operator numbers. This reduces the size of the $c$ variable.
Chapter 6

Evaluation

This chapter will cover some aspects of the implemented code generator. The graphs in the first section are hand made. The second section will provide some numbers from real world examples.

6.1 Hand made graphs

Sensitivity to latency of shuffle instructions  This first example is supposed to illustrate the impact of a data movement instruction and placement of operands in memory. The graph in Figure 6.1 is supposed to represent computation of \((a^2 + b^2 + c^2 + d^2)\). It is assumed that the memory addresses of \(a, b, c\) and \(d\) point to the same memory line.

In the following solution, SIMD instructions are forced to cover the nodes 50 and 51. It is assumed that a bitmask for moving a 32bit operand from slot 0 to slot 1 reside in register r10. A full solution from AMPL is found in appendix A.1.

```
0: 1qa r1, _VecOpL
1: 1qa r2, _VecOpR
6: mpy r3, r1, r1
7: mpy r4, r2, r2
14: a r5, r3, r4
16: shufb r6, r5, r5, r10
20: a r7, r5, r6
```

**Listing 6.1.** Manual interpretation of the solution for the graph in Figure 6.1

The schedule takes 22 clock cycles to complete since the latency of the \(a^2\) is 2. If no SIMD instructions are forced a schedule that is 2 clock cycles faster is found. The re-alignment of either the value of node 50 or 51, takes longer time than what is gained by running the other instructions as SIMD instructions. The solution without SIMD instructions can be found in appendix A.1. If node 52 is removed completely a schedule with SIMD instructions costs 16 cycles, while one without costs 18 cycles to complete.
Figure 6.1. Hand made graph, representing the computation \((a^a+b^b+c^c+d^d)\)

**Instructions requiring operands in preferred slot**  This next example illustrates a graph with shift instructions. The graph, Figure 6.2, contains five load instructions and four shift instructions. The graph correspond to shifting values \(a, b, c\) and \(d\), according to the value \(e\). In the solution it is assumed that the values of \(a, b, c\) and \(d\) are at the same memory line. It is also assumed that the value \(e\) is at another memory line in its preferred slot. The full solution can be found in appendix A.2.
6.1 Hand made graphs

Figure 6.2. Hand made graph, representing shift of values a, b, c, and d (node numbers 10,11,12,13 respectively), e (node number 14) steps to the left

Listing 6.2. Manual interpretation of the solution for graph in Figure 6.2

0: lqa r1, _VecOp
1: lqa r2, _OpR
7: shl r3, r1, r2

The solution with SIMD instructions require 11 cycles (the latency of the instruction shl r3,r1,r2 is 4), while the best solution without any SIMD instructions requires 14 clock cycles. See appendix A.2 for full solutions from AMPL.
Pack phases The examples until now have not required any shuffle instructions in the pack phase of a scheduled SIMD instruction. This very contrived example will show when two shuffle instructions have to be done. In this example, Figure 6.3, the nodes in the range 40 to 49 are forced to not be covered by a SIMD instruction. This will require that the pack phase schedules two shuffle instructions for each side of incoming operands, for the SIMD instruction covering the nodes in the 50 range. Currently the model will schedule two pack phases even if they operate on the exact same operands and these operands are to be aligned equally in both pack phases.
6.2 Real world experiments

The experiments in this section are of basic blocks from the codec library libavcodec [11] and the DSPSTONE benchmark suite [26]. The results can be found in Table 6.1.

The experiments were run on a computer with 4 GB RAM and an AMD Athlon X2 6000+ processor. The version of CPLEX used was 10.2. A timer was used to stop the solver if a solution was not found within two minutes. Out of curiosity we kept the timer off in the experiment \textit{N complex updates - 2}. The values in column \textit{SISD} were generated by only having a single ’dummy’ element in the set \textit{SC}. The column \textit{t} shows the execution times of the code generator. The columns \textit{τ} contains execution times for the generated schedules. The column \textit{|G|} contains the number of nodes in the input basic blocks. The column \textit{|SC|} contains the number of candidates in the set \textit{SC}. Finally, the column \textit{#vars} contains the number of variables in the corresponding SIMD ILP-problem. The difference between \textit{N complex updates - 1} and \textit{N complex updates - 2} is that in the latter, the set \textit{SC} only contains sets of cardinality equal to two.

It seems feasible to solve basic blocks containing 15 - 20 nodes. The running time of the solver is increased greatly for basic blocks with more nodes. However, this vary depending on how many candidates the set \textit{SC} has. For example we can see that it is feasible to solve basic blocks with between 35 - 40 nodes as long as no SIMD candidates are present. Methods to further reduce the number of candidate sets in the set \textit{SC} should be addressed in future work.

Listing 6.3. Manual interpretation of the solution for graph in Figure 6.2

0: \texttt{lqa r1, _VecOp}
6: \texttt{a r3, r1, r1}
7: \texttt{a r4, r1, r1}
8: \texttt{a r5, r1, r1}
9: \texttt{a r6, r1, r1}
11: \texttt{shufb r7, r3, r4, r10}
12: \texttt{shufb r7, r5, r6, r10}
13: \texttt{shufb r7, r5, r6, r10}
14: \texttt{shufb r7, r5, r6, r10}
18: \texttt{a r8, r7, r7}
Table 6.1. Results from the code generator. An entry marked with a '-' corresponds to an instance where no solution was found.

| Basic Block | Name       | $|G|$ | $\tau$ | $\tau$ | $|SC|$ | #vars |
|-------------|------------|-----|--------|--------|-------|-------|
|             | Basic Blocks from libavcodec library |       |       |       |       |       |
|             | h264-1     | 8   | 13     | 10     | 0.032 | 12    | 1622  |
|             | h264-2     | 14  | 32     | 28     | 2.164 | 15    | 13018 |
|             | img-conv   | 17  | 36     | 28     | 0.376 | 20    | 14058 |
|             | mpeg-vid   | 44  | 37     | -      | -     | 3050  | -     |
|             | Basic Blocks from DPSTONE benchmark suite |       |       |       |       |       |
|             | FIR-filter | 20  | 30     | 30     | 0.296 | 4     | 14878 |
|             | N complex updates - 1 | 27  | 26     | -      | -     | 96    | 24378 |
|             | N complex updates - 2 | 27  | 26     | 26     | 307.271 | 38     | 21460 |
|             | Biquad n   | 30  | 27     | -      | -     | 60    | 30917 |
Chapter 7

Limitations and Future Work

The first section of this chapter will introduce some limitations that should be solved specifically for code generation for the SPE. The second section will present some thoughts on future work to make the code generator more retargetable.

7.1 Limitations

Limitations of the model of the CELL SPE The main drawback of the current implementation is that data movement instructions are part of the ILP formulation described in Chapter 5. The idea of a retargetable code generator is somewhat lost. Further work should be done to move this part into the architecture description file. Another, slightly smaller, drawback is the lack of a separate set $P_{simd}$ that contains SIMD patterns and their corresponding entries in the $B_p, OP_{p,k}, L_p, PD_r, PS1_r, PS2_r$ sets and parameters. The patterns of this set should then be used to cover the sets found in SIMDCandidates. In the case of code generation for SPEs, we can do without this set if more work is put in interpretation of the solution from the ILP-solver. For the SPEs the main difference between SIMD and SISD operations is the packing of operands before a SIMD is scheduled. As previously mentioned, by omitting the set of SIMD patterns, we reduce the number of variables in the ILP-problem.

Limitations from the constraints Five drawbacks will be mentioned regarding the part of the ILP formulation found in Chapter 5. First, shuffle instructions that pack left or right hand side operands of a SIMD computation will be scheduled in serial. Therefore, if two shuffle instructions are needed, it is not strictly integrated anymore. These two shuffle instructions should be allowed to be scheduled independently. Second, a constraint to verify that the schedule does not use too many registers is not currently present. Third, the solution
lacks a general solution to the pack phase. The current one is only applicable to architectures having a shuffle instruction. Fourth, future work should look into using patterns to cover the sets in SIMDCandidates. The additional variables needed could probably lower the complexity of some of the constraints above. Fifth, currently an operand can only be at one place at a given time. In some cases it could be beneficial if a scalar is available from several alignments. An example of this is when a scalar is supposed to scale a matrix. In theory, four elements can be fetched from the matrix in one load instruction. The operand that scales the matrix elements must reside in every slot of another register when the SIMD multiplication is issued. The biggest drawback is probably the lack of this scattering of an operand.

7.2 Future Work

Extending the model for other architectures The variable \( \text{simd}_{\text{exas},f,t} \) should be modified by replacing \( f \in F \) with \( p \in P_{\text{simd}} \) for the formulation to work with other architectures that differentiate SISD and SIMD instructions, have several register banks or several functional units accepting the same operations.

The third step of the code generator: interpretation The main reason for this third step is the pack phase of a scheduled SIMD instruction. The inspiration of this phase is described in the papers by Tanaka et al. [25] and Leupers [22]. To realign and shuffle operands we use the shuffle instruction of the SPE. The mathematical solution from the second step in the code generator only flags how many and when these shuffle instructions have to be scheduled. Therefore the final assembler code and bit patterns for the shuffle instructions have to be created during this third step. It should also be pointed out that the mathematical solution from step two is rather cumbersome and not very straightforward to extract assembler instructions from. There was not enough project time to start implementation of the third step though. A manual interpretation of the output from the second step of the code generator is therefore needed to get working assembler code.
Chapter 8

Related Work

In this chapter we list some related work done in ILP formulation of code generation as well as in code generation with SIMD instructions.

8.1 ILP for code generation

We have seen two ideas, both described by Kästner [18], for formulation of the integer linear program used to generate an optimal schedule for basic blocks. One is time based, which means it assigns events to points in time in a similar manner as our code generator does. The second one is order based. In this formulation the order of events is optimized. Thereby the points in time, when events actually happen, is implicit. No method is superior to the other one, but the order based formulation can give a more efficient integer linear program in some cases.

Work have also been done to make the integer linear problem easier to solve by transformation of the input DAG. However, these techniques are usually target machine dependent. See for example Wilken et al. [27].

The work in this thesis project is an extension to the model by Eriksson [8]. Our model takes into account SIMD instructions of the CELL SPE processor, to SIMDdizise scalar computation in basic blocks.

8.2 SIMD code generation

A lot of work has been carried out in the topic of SIMD code generation. Work specifically for SIMD code generation by integer linear programming was rather hard to find though. Two very interesting approaches was found, the first one is by Leupers [22]. The method extend a tree-based code selection technique to generate alternative covers. In a later step, once the the trees are covered by instructions, they are able to do a more detailed code selection to generate SIMD instructions. Another method is an extension to Leupers method [21] by Tanaka et al. [25]. In their method, nodes for data transfers are inserted into the basic
blocks DAGs. These nodes are later covered by so-called covering rules for the PACK instructions that move and align operands.

In the stream programming paradigm, Eichenberger et al. [28] proposed a technique to SIMDize arbitrary streams to an arbitrary offset without knowledge of alignments or offsets of the operands at compile time. This enables efficient handling of runtime alignment of operands. Work has also been done to create new programming languages and extensions to existing languages that specifically target processor architectures with SIMD instructions. For example, Kumar and Kahl [3] have embedded a domain-specific language into the programming language Haskell. This domain-specific language allows programmers to easily create SIMD parallel algorithms. Another programming language with the same spirit is the data parallel language CGiS and its corresponding compiler framework, presented in Fritz dissertation [14]. An optimizing compiler has been created specifically for the CELL processor. This is described in the paper by Eichenberger et al. [7]. This compiler handles SIMD code generation in several phases of the whole compilation process.
Chapter 9

Conclusions

This thesis work has extended Eriksson’s code generator to generate code for the
CELL SPE. The code generator is also extended with the ability to gather
operands into registers and schedule SIMD instructions. The ability to generate
SIMD instruction has been the main focus. The conclusion is that in some cases
it is highly likely that the execution time for a basic block can be reduced by a
few clock cycles with the implemented technique. This almost requires that the
operands are placed at good locations in memory so that a single load and no
packing of operands have to be done. The cost of packing the operands will
otherwise become too big, at least on the CELL SPE. Much of this is because of
the fully pipelined property and the well balanced distribution of which
instructions the two execution pipelines (EVEN and ODD) accepts. An idea
would be to first solve each basic block, see which basic block would gain most
from having the operands placed at arbitrary locations in memory. After this,
solve for the other basic blocks again but force placement on operands that got a
location fixed by previously solved basic blocks.
Bibliography


the generation of optimized simd-parallel assembly code, 2007.

Processors. Ph.d thesis, Department of Computer and Information Science, 
Linkoping University, Sweden, 2006.

code generation with integer linear programming. In Proc. Euro-Par 2006, 


Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao 
Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for the Cell 

Department of Computer and Information Science, Linkoping University, 
Sweden, 2009.

[9] Mattias Eriksson and Christoph W. Kessler. Integrated code generation for 
loops. Accepted for publication in ACM Transactions on Embedded 

heuristic integrated code generation for clustered VLIW architectures. In 
SCOPES ’08: Proceedings of the 11th international workshop on Software 
and compilers for embedded systems, pages 11–20, New York, NY, USA, 
2008. ACM.


Appendix A

Solutions

A.1 Solutions for graph in Figure 6.1

These are schedules found, belonging to the graph 6.1 in chapter 6.
The solution found while forcing node 50 and 51 to be covered by SIMD instructions:

ILOG AMPL 10.100, licensed to "university–linkoping".
AMPL Version 20070505 (Linux 2.6.5–7.97–smp)

Presolve eliminates 5935 constraints and 6155 variables.
Adjusted problem:
3204 variables, all binary
4296 constraints, all linear; 17390 nonzeros
0 objectives.

ILOG CPLEX 10.200, licensed to "university–linkoping", options: em b
q use=1
CPLEX 10.2.0: mipdisplay=3
mipinterval=500
timing=1
prestats=1
mipemphasis=0
timelimit=120
MIP LP Presolve eliminated 21 rows and 0 columns.
Aggregator did 29 substitutions.

GUB cover cuts applied: 4
Implied bound cuts applied: 5
Gomory fractional cuts applied: 6

Times (seconds):
Input = 0.008
Solve = 0.380024
Output = 0.008
CPLEX 10.2.0: optimal integer solution; objective 0
1419 MIP simplex iterations
0 branch–and–bound nodes
Objective = find a feasible point.

```
c :=
10 ADDRGP4 0 0 1
11 ADDRGP4 0 0 1
12 ADDRGP4 0 0 1
13 ADDRGP4 0 0 1
52 ADD4.EVENRARARA 0 20 1

s; #empty

r [CONST, * , *] (tr)
[RA, * , *] (tr)
       : 20 21 22 23 41 42 43 44 50 51 :=
   6 1 0 1 0 0 0 0 0 . .
   7 0 1 0 1 0 0 0 0 . .
   8 0 0 0 1 0 0 0 0 . .
   9 0 0 0 1 0 0 0 0 . .
  10 0 0 0 1 0 0 0 0 . .
  11 0 0 0 1 0 0 0 0 0 0
  12 0 0 0 1 0 0 0 0 0 0
  13 0 0 0 1 1 0 1 0 0 0
  14 0 0 0 1 1 1 1 1 0 0
  15 0 0 0 1 0 0 0 0 0 0
  16 0 0 0 1 0 0 0 0 1 1
  17 0 0 0 1 0 0 0 0 1 1
  18 0 0 0 1 0 0 0 0 1 1
  19 0 0 0 1 0 0 0 0 1 1
  20 0 0 0 1 0 0 0 0 1 1

  [SYMBOL, * , *] (tr)
       : 10 11 12 13 :=
  0 1 1 1 1
  1 1 1 1 1
  2 1 1 1 1
  3 1 1 1 1
  4 1 1 1 1
  5 1 1 1 1
  6 1 1 1 1
  7 1 1 1 1
  8 1 1 1 1
  9 1 1 1 1
 10 1 1 1 1
 11 1 1 1 1
 12 1 1 1 1
 13 1 1 1 1
 14 1 1 1 1
 15 1 1 1 1
 16 1 1 1 1
 17 1 1 1 1
 18 1 1 1 1
 19 1 1 1 1

s :=

w: #empty
```
II = 9

\[
\text{set match } [10] := \text{ADDRGP4}; \\
\text{set match } [11] := \text{ADDRGP4}; \\
\text{set match } [12] := \text{ADDRGP4}; \\
\text{set match } [13] := \text{ADDRGP4}; \\
\text{set match } [20] := \text{LDP4\_ODDRARA LDP4\_ODDRASYMBO}; \\
\text{set match } [21] := \text{LDP4\_ODDRARA LDP4\_ODDRASYMBO}; \\
\text{set match } [22] := \text{LDP4\_ODDRARA LDP4\_ODDRASYMBO}; \\
\text{set match } [23] := \text{LDP4\_ODDRARA LDP4\_ODDRASYMBO}; \\
\text{set match } [41] := \text{MPY\_EVENRARARA MPY\_EVENRARACST}; \\
\text{set match } [42] := \text{MPY\_EVENRARARA MPY\_EVENRARACST}; \\
\text{set match } [43] := \text{MPY\_EVENRARARA MPY\_EVENRARACST}; \\
\text{set match } [44] := \text{MPY\_EVENRARARA MPY\_EVENRARACST}; \\
\text{set match } [50] := \text{ADDH4\_EVENRARARA ADDH4\_EVENRARACST}; \\
\text{set match } [51] := \text{ADDH4\_EVENRARARA ADDH4\_EVENRARACST}; \\
\text{set match } [52] := \text{ADDH4\_EVENRARARA ADDH4\_EVENRARACST}; \\
\text{sum\{v in G\} max\_lat[v] = 58}
\]

\[
\text{set SIMD\_sets[1] := 51 50; } \\
\text{set SIMD\_sets[2] := 20 22; } \\
\text{set SIMD\_sets[3] := 21 23; } \\
\text{set SIMD\_sets[4] := 41 43; } \\
\text{set SIMD\_sets[5] := 42 44; } \\
\]

SIMD :=
1 EVEN 14 1
2 ODD 0 1
3 ODD 1 1
4 EVEN 6 1
5 EVEN 7 1

Align32 [s, s, 0] (tr)
: 10 11 20 21 22 23 41 42 50 51 52 :=
0 1 1 1 1 0 0 . . . .
1 1 1 1 1 0 0 . . . .
2 1 1 1 1 0 0 . . . .
3 1 1 1 1 0 0 . . . .
4 1 1 1 1 0 0 1 1 . .
5 1 1 1 1 0 0 1 1 . .
6 1 1 1 1 0 0 1 1 . .
7 1 1 1 1 0 0 1 1 . .
8 1 1 1 1 0 0 1 1 . .
9 1 1 1 1 0 0 1 1 . .
10 1 1 1 1 0 0 1 1 . .
11 1 1 1 1 0 0 1 1 0 .
12 1 1 1 1 1 1 1 1 0 .
13 1 1 1 1 1 1 1 1 0 1
14 1 1 1 1 1 1 1 1 0 1
15 1 1 1 1 1 1 1 1 0 1
16 1 1 1 1 1 0 1 1 1 0 1
17 1 1 1 1 1 0 1 1 1 0 1
18 1 1 1 1 1 0 1 0 1 0 1
19 1 1 1 1 1 0 1 0 1 0 1
20 1 1 1 1 1 0 1 0 1 1 1
\[* , * , 1 \] \( (\text{tr}) \)

\[
\begin{array}{cccc}
17 & 1 \\
18 & 1 \\
19 & 1 \\
20 & 1 \\
\end{array}
\]

\[* , * , 2 \] \( (\text{tr}) \)

\[
\begin{array}{cccc}
16 & 1 & 0 \\
17 & 1 & 0 \\
18 & 1 & 1 \\
19 & 1 & 1 \\
20 & 1 & 1 \\
\end{array}
\]

\[* , * , 3 \] \( (\text{tr}) \)

\[
\begin{array}{cccccccc}
12 & 13 & 22 & 23 & 43 & 44 & 51 & := \\
0 & 1 & 1 & 1 & 1 & . & . & . \\
1 & 1 & 1 & 1 & 1 & . & . & . \\
2 & 1 & 1 & 1 & 1 & . & . & . \\
3 & 1 & 1 & 1 & 1 & . & . & . \\
4 & 1 & 1 & 1 & 1 & 1 & 1 & . \\
5 & 1 & 1 & 1 & 1 & 1 & 1 & . \\
6 & 1 & 1 & 1 & 1 & 1 & 1 & . \\
7 & 1 & 1 & 1 & 1 & 1 & 1 & . \\
8 & 1 & 1 & 1 & 1 & 1 & 1 & . \\
9 & 1 & 1 & 1 & 1 & 1 & 1 & . \\
10 & 1 & 1 & 0 & 1 & 1 & 1 & . \\
11 & 1 & 1 & 0 & 1 & 1 & 1 & . \\
12 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
13 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
14 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
15 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
16 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
17 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
18 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
19 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
20 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]

\( \text{xa} := \)

\[
\begin{array}{cccc}
22 & 2 & 0 & 6 \\
23 & 0 & 2 & 12 \\
23 & 2 & 0 & 8 \\
42 & 3 & 2 & 14 \\
43 & 3 & 1 & 13 \\
51 & 1 & 0 & 16 \\
\end{array}
\]

\( \text{shuffDataForSimd} := \)

\[
\begin{array}{cccc}
1 & 14 & 0 & 1 \\
4 & 6 & 0 & 1 \\
5 & 7 & 0 & 1 \\
\end{array}
\]

\( \text{shuffDataForSimdR} := \)

\[
\begin{array}{cccc}
1 & 14 & 0 & 1 \\
\end{array}
\]

\( \text{sp} := \)

\[
\begin{array}{cccc}
1 & 14 & 0 & 1 \\
4 & 6 & 0 & 1 \\
5 & 7 & 0 & 1 \\
\end{array}
\]
A.1 Solutions for graph in Figure 6.1

4 6 0 1
5 7 0 1
:

set TRIG := 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20;

set slots_SIMD[2] := 0 1 2 3 4 5;
set slots_SIMD[3] := 0 1 2 3 4 5;
set slots_SIMD[4] := 4 5 6 7 8 9 10 11;
set slots_SIMD[5] := 4 5 6 7 8 9 10 11;

set slots[10] := 0;
set slots[12] := 0;
set slots[13] := 0;
set slots[20] := 0 1 2 3 4 5;
set slots[21] := 0 1 2 3 4 5;
set slots[22] := 0 1 2 3 4 5;
set slots[23] := 0 1 2 3 4 5;
set slots[41] := 4 5 6 7 8 9 10 11;
set slots[42] := 4 5 6 7 8 9 10 11;
set slots[43] := 4 5 6 7 8 9 10 11;
set slots[44] := 4 5 6 7 8 9 10 11;
set slots[50] := 11 12 13 14 15 16 17 18;
set slots[51] := 11 12 13 14 15 16 17 18;
set slots[52] := 13 14 15 16 17 18 19 20;

The best solution found:

ILOG AMPL 10.100, licensed to "university–linkoping".
AMPL Version 20070505 (Linux 2.6.5−7.97−smp)

Presolve eliminates 12749 constraints and 5965 variables.
Adjusted problem:
3030 variables, all binary
8704 constraints, all linear; 34628 nonzeros
0 objectives.

ILOG CPLEX 10.200, licensed to "university–linkoping", options: embo q use=1
CPLEX 10.2.0: mipdisplay=3
mipinterval=500
timing=1
prestats=1
mipemphasis=0
timelim=120
MIP LP Presolve eliminated 7 rows and 0 columns.
Aggregator did 21 substitutions.

Times (seconds):
Input = 0.012001
Solve = 0.296018
Output = 0.012
CPLEX 10.2.0: optimal integer solution; objective 0
1354 MIP simplex iterations
0 branch–and–bound nodes
Objective = find a feasible point.
10 ADDRGP4 0 0 1
11 ADDRGP4 0 0 1
12 ADDRGP4 0 0 1
13 ADDRGP4 0 0 1
20 LDP4.ODDRASYMBOLO 0 3 1
21 LDP4.ODDRASYMBOLO 0 0 1
22 LDP4.ODDRASYMBOLO 0 2 1
23 LDP4.ODDRASYMBOLO 0 1 1
41 MPY.EVENRARARA 0 9 1
42 MPY.EVENRARARA 0 6 1
43 MPY.EVENRARARA 0 8 1
44 MPY.EVENRARARA 0 7 1
50 ADDI4.EVENRARARA 0 16 1
51 ADDI4.EVENRARARA 0 15 1
52 ADDI4.EVENRARARA 0 18 1

s; #empty
r [CONST,*,*] (tr)
[RA,*,*,*] (tr)
:= 20 21 22 23 41 42 43 44 50 51
6 0 1 0 0 0 0 0 0 . .
7 0 1 0 1 0 0 0 0 . .
8 0 1 1 0 0 0 0 0 . .
9 1 1 1 0 0 0 0 0 . .
10 1 1 1 0 0 0 0 0 . .
11 1 1 1 0 0 0 0 0 0 0
12 1 1 1 0 0 0 0 0 0 0
13 1 1 1 0 0 1 0 0 0 0
14 1 1 1 0 0 1 0 1 0 0
15 1 1 1 0 0 1 1 1 0 0
16 1 1 1 0 1 1 0 0 0 0
17 1 1 1 0 1 1 0 0 0 1
18 1 1 1 0 1 1 0 0 1 1

[S,*,*,*] (tr)
:= 10 11 12 13
0 1 1 1 1
1 1 1 1 1
2 1 1 1 1
3 1 1 1 1
4 1 1 1 1
5 1 1 1 1
6 1 1 1 1
7 1 1 1 1
8 1 1 1 1
9 1 1 1 1
10 1 1 1 1
11 1 1 1 1
12 1 1 1 1
13 1 1 1 1
14 1 1 1 1
15 1 1 1 1
16 1 1 1 1
17 1 1 1 1

;
A.1 Solutions for graph in Figure 6.1

\( x := 
\)

\( w: \#empty \)

\( H = 18 \)

set match [10] := ADDRGP4;
set match [12] := ADDRGP4;
set match [22] := LDP4.ODDRARA LDP4.ODDRASYMBO;
set match [23] := LDP4.ODDRARA LDP4.ODDRASYMBO;
set match [41] := MPY. EVENRARARA MPY. EVENRARACONST;
set match [42] := MPY. EVENRARARA MPY. EVENRARACONST;
set match [43] := MPY. EVENRARARA MPY. EVENRARACONST;
set match [44] := MPY. EVENRARARA MPY. EVENRARACONST;
set match [50] := ADDI4.EVENRARARA ADDI4.EVENRARACONST;
set match [51] := ADDI4.EVENRARARA ADDI4.EVENRARACONST;
set match [52] := ADDI4.EVENRARARA ADDI4.EVENRARACONST;

\[ \text{sum}\{v \in G\} \text{ max}_\text{lat}[v] = 58 \]

set SIMD_sets [1] := 51 50;
set SIMD_sets [2] := 44 43;
set SIMD_sets [3] := 44 42;
set SIMD_sets [7] := 42 41;
set SIMD_sets [8] := 23 22;
set SIMD_sets [9] := 23 21;
set SIMD_sets [10] := 23 20;
set SIMD_sets [12] := 22 20;
set SIMD_sets [14] := 44 43 42;
set SIMD_sets [15] := 44 43 41;
set SIMD_sets [16] := 44 42 41;
set SIMD_sets [17] := 43 42 41;
set SIMD_sets [18] := 23 22 21;
set SIMD_sets [19] := 23 22 20;
set SIMD_sets [20] := 23 21 20;
set SIMD_sets [21] := 22 21 20;
set SIMD_sets [22] := 44 43 42 41;
set SIMD_sets [23] := 23 22 21 20;

SIMD := 

Align32 \[ *\times *\times 0 \] (tr) 

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>41</th>
<th>42</th>
<th>43</th>
<th>44</th>
<th>50</th>
<th>51</th>
<th>52</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .
5 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .
6 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .
7 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .
8 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .
9 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .
10 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .
11 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .
12 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .
13 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .
14 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .
15 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .
16 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .
17 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .
18 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . .

\[*(.,1)\] (tr)
\[*(.,2)\] (tr)
\[*(.,3)\] (tr);

xa :=
21 1 2 10 1
;

shuffDataForSimd :=
;

shuffDataForSimdR :=
;

set TREG := 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18;

set slots_SIMD[2] := 4 5 6 7 8 9;
set slots_SIMD[3] := 4 5 6 7 8 9;
set slots_SIMD[4] := 4 5 6 7 8 9;
set slots_SIMD[5] := 4 5 6 7 8 9;
set slots_SIMD[6] := 4 5 6 7 8 9;
set slots_SIMD[7] := 4 5 6 7 8 9;
set slots_SIMD[8] := 0 1 2 3;
set slots_SIMD[9] := 0 1 2 3;
set slots_SIMD[10] := 0 1 2 3;
set slots_SIMD[11] := 0 1 2 3;
set slots_SIMD[12] := 0 1 2 3;
set slots_SIMD[13] := 0 1 2 3;
set slots_SIMD[14] := 4 5 6 7 8 9;
set slots_SIMD[15] := 4 5 6 7 8 9;
set slots_SIMD[16] := 4 5 6 7 8 9;
set slots_SIMD[17] := 4 5 6 7 8 9;
set slots_SIMD[18] := 0 1 2 3;
set slots_SIMD[19] := 0 1 2 3;
set slots_SIMD[20] := 0 1 2 3;
set slots_SIMD[21] := 0 1 2 3;
set slots_SIMD[22] := 4 5 6 7 8 9;
set slots_SIMD[23] := 0 1 2 3;

set slots[10] := 0;
A.2 Solutions for graph in Figure 6.2

These are the schedules found, belonging to the graph 6.2 in chapter 6
With SIMD instructions the following solution is found:

ILOG AMPL 10.100, licensed to "university–linkoping".
AMPL Version 20070505 (Linux 2.6.5–7.97–smp)

Presolve eliminates 1186 constraints and 2354 variables.
Adjusted problem:
751 variables, all binary
1388 constraints, all linear; 4369 nonzeros
0 objectives.

ILOG CPLEX 10.200, licensed to "university–linkoping", options: emb use=1
CPLEX 10.2.0: mipdisplay=3
mipinterval=500
timing=1
prestats=1
mipemphasis=0
timelimit=60
MIP LP Presolve eliminated 4 rows and 0 columns.
Aggregator did 12 substitutions.

Times (seconds):
Input = 0.004
Solve = 0.012
Output = 0.004
CPLEX 10.2.0: optimal integer solution; objective 0
172 MIP simplex iterations
0 branch–and–bound nodes
Objective = find a feasible point.
c :=
10 ADDRGP4 0 0 1
11 ADDRGP4 0 0 1
12 ADDRGP4 0 0 1
13 ADDRGP4 0 0 1
14 ADDRGP4 0 0 1
24 LDP4.ODDRASYMBOLO 0 1 1
;

set slots[12] := 0;
set slots[13] := 0;
set slots[20] := 0 1 2 3;
set slots[21] := 0 1 2 3;
set slots[22] := 0 1 2 3;
set slots[23] := 0 1 2 3;
set slots[41] := 4 5 6 7 8 9;
set slots[42] := 4 5 6 7 8 9;
set slots[43] := 4 5 6 7 8 9;
set slots[44] := 4 5 6 7 8 9;
set slots[50] := 11 12 13 14 15 16;
set slots[51] := 11 12 13 14 15 16;
set slots[52] := 13 14 15 16 17 18;
s; #empty
r[CONST,*,*]
[RA,*,*]: 6 7 := 20 1 1
21 1 1
22 1 1
23 1 1
24 0 1

[SOMBOLO,*,*]: 0 1 2 3 4 5 6 := 10 1 1 1 1 1 1
11 1 1 1 1 1 1
12 1 1 1 1 1 1
13 1 1 1 1 1 1
14 1 1 1 1 1 1

x := ;
w; #empty
II = 2

set match[10] := ADDRGP4;
set match[12] := ADDRGP4;
set match[14] := ADDRGP4;
set match[22] := LDP4.ODDARA LDP4.ODDASOMBOLO;
set match[23] := LDP4.ODDARA LDP4.ODDASOMBOLO;
set match[41] := SHL.EVENRARARA SHL.EVENRARACONST;
set match[42] := SHL.EVENRARARA SHL.EVENRARACONST;
set match[43] := SHL.EVENRARARA SHL.EVENRARACONST;
set match[44] := SHL.EVENRARARA SHL.EVENRARACONST;

sum{v in G} max_lat[v] = 46

set SIMD_sets[2] := 41 42 43 44;
SIMD :=
1 ODD 0 1
2 EVEN 7 1
;

Align32 [*,*,0]: 0 1 2 3 4 5 6 7 := 12 1 1 1 1 1 1 1
14 1 1 1 1 1 1 1
22 1 1 1 1 1 1 1
A.2 Solutions for graph in Figure 6.2

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>43</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$[*,*]$

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>44</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$[*,*]$

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>41</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$[*,*]$

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>42</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Forcing that no SIMD instructions are used, the following solution is found:

ILOG AMPL 10.100, licensed to "university−linkoping".
AMPL Version 20070505 (Linux 2.6.5−7.97−smp)

Presolve eliminates 2206 constraints and 3546 variables.
Adjusted problem:
1165 variables, all binary
1896 constraints, all linear; 6247 nonzeros
0 objectives.

ILOG CPLEX 10.200, licensed to "university–linkoping", options: emb
q use=1
CPLEX 10.2.0: mipdisplay=3
mipinterval=500
timing=1
prestats=1
mipemphasist=0
timeLimit=120
MIP LP Presolve eliminated 4 rows and 0 columns.
Aggregator did 12 substitutions.

Times (seconds):
Input = 0.004
Solve = 0.020001
Output = 0.004
CPLEX 10.2.0: optimal integer solution; objective 0
66 MIP simplex iterations
0 branch-and-bound nodes
Objective = find a feasible point.
c :=
  10 ADDRGP4 0 0 1
  11 ADDRGP4 0 0 1
  12 ADDRGP4 0 0 1
  13 ADDRGP4 0 0 1
  14 ADDRGP4 0 0 1
  20 LDP4.ODDRASYMBO10 4 1
  21 LDP4.ODDRASYMBO10 2 1
  22 LDP4.ODDRASYMBO10 0 1
  23 LDP4.ODDRASYMBO10 0 1
  24 LDP4.ODDRASYMBO10 0 1
  41 SXL.EVENRARARA 0 10 1
  42 SXL.EVENRARARA 0 8 1
  43 SXL.EVENRARARA 0 7 1
  44 SXL.EVENRARARA 0 9 1
;

s; #empty
r [CONST,*,*]
  [RA,*,*]
  . 6 7 8 9 10 :=
  20 0 0 0 0 1
  21 0 0 1 0 0
  22 1 1 0 0 0
  23 0 0 0 1 0
  24 0 1 1 1 1

  [SYMBOL,*,*]
  . 0 1 2 3 4 5 6 7 8 9 :=
  10 1 1 1 1 1 1 1 1 1 1
  11 1 1 1 1 1 1 1 1 1 1
  12 1 1 1 1 1 1 1 1 1 1
  13 1 1 1 1 1 1 1 1 1 1
A.2 Solutions for graph in Figure 6.2

14 1 1 1 1 1 1 1 1 1 1 1

; x :=

; w: #empty

I = 5

set match [10] := ADDRGP4;
set match [12] := ADDRGP4;
set match [14] := ADDRGP4;
set match [22] := LDP4.ODDRARA LDP4.ODDRASYMBOI;
set match [23] := LDP4.ODDRARA LDP4.ODDRASYMBOI;
set match [41] := SHL.EVENRARARA SHL.EVENRARACONST;
set match [42] := SHL.EVENRARARA SHL.EVENRARACONST;
set match [43] := SHL.EVENRARARA SHL.EVENRARACONST;
set match [44] := SHL.EVENRARARA SHL.EVENRARACONST;

\sum_{v \in X} \max_{\text{lat}} [v] = 46


SIMD :=

Align32 [*,*,0]

: 0 1 2 3 4 5 6 7 8 9 10 :=
10 1 1 1 1 1 1 1 1 1 1 1
11 1 1 1 1 1 1 1 1 1 1 1
12 1 1 1 1 1 1 1 1 1 1 1
13 1 1 1 1 1 1 1 1 1 1 1
14 1 1 1 1 1 1 1 1 1 1 1
20 1 1 1 1 1 1 1 1 1 1 1
21 1 1 1 1 1 1 1 1 1 1 1
22 1 1 1 1 1 1 1 1 1 1 1
23 1 1 1 1 1 1 1 1 1 1 1
24 1 1 1 1 1 1 1 1 1 1 1
41 . . . . 1 1 1 1 1 1 1
42 . . . . 1 1 1 1 1 1 1
43 . . . . 1 1 1 1 1 1 1
44 . . . . 1 1 1 1 1 1 1

\mbox{[*,*,1]}
\mbox{[*,*,2]}
\mbox{[*,*,3]};

xa :=

; shuffDataForSimd := ;
A.3 Solution for graph in Figure 6.3

ILOG AMPL 10.100, licensed to "university–linkoping".
AMPL Version 20070505 (Linux 2.6.5–7.97–smp)

Presolve eliminates 10371 constraints and 5759 variables.
Adjusted problem:
3541 variables, all binary
5673 constraints, all linear; 22496 nonzeros
0 objectives.

ILOG CPLEX 10.200, licensed to "university–linkoping", options: emb
use=1
CPLEX 10.2.0: mipdisplay=3
mipinterval=500
timing=1
prestats=1
mipemphasis=0
timelimit=120
MIP LP Presolve eliminated 8 rows and 0 columns.
Aggregator did 28 substitutions.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Node</th>
<th>Objective</th>
<th>Inf</th>
<th>Best Integer</th>
<th>Cuts/</th>
<th>Best Node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ItCnt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Left</td>
<td>0.0000</td>
<td>275</td>
<td></td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>424</td>
<td>0.0000</td>
<td>321</td>
<td></td>
<td>0.0000</td>
<td>999</td>
</tr>
<tr>
<td>*</td>
<td>0+</td>
<td>999</td>
<td>0.00%</td>
<td>0.0000</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>999</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A.3 Solution for graph in Figure 6.3

Clique cuts applied: 17
Implied bound cuts applied: 7
Gomory fractional cuts applied: 12

Times (seconds):
Input = 0.012
Solve = 0.204013
Output = 0.008
CPLEX 10.2.0: optimal integer solution; objective 0
999 MIP simplex iterations
0 branch-and-bound nodes
Objective = find a feasible point.
c :=
10 ADDRGP4 0 0 1
11 ADDRGP4 0 0 1
12 ADDRGP4 0 0 1
13 ADDRGP4 0 0 1
41 ADDH4.EVENRARARA 0 9 1
42 ADDH4.EVENRARARA 0 7 1
43 ADDH4.EVENRARARA 0 6 1
44 ADDH4.EVENRARARA 0 8 1
r

s: #empty

r [CONST,*,*] (tr)
|RA,*,*| (tr)
: 20 21 22 23 41 42 43 44 :=
6 1 1 1 1 0 0 0 0
7 1 1 1 1 0 0 0 0
8 1 1 1 1 0 0 1 0
9 1 1 1 1 0 1 1 0
10 1 1 1 1 0 1 1 1
11 1 0 1 0 1 1 1 1
12 1 0 1 0 1 1 1 1
13 1 0 0 0 1 1 1 1
14 0 0 0 0 1 1 1 1
15 0 0 0 0 1 1 1 1
16 0 0 0 0 1 1 1 1
17 0 0 0 0 1 1 1 1
18 0 0 0 0 1 1 1 1

|SYMBOL,*,*| (tr)
: 10 11 12 13 :=
0 1 1 1 1
1 1 1 1 1
2 1 1 1 1
3 1 1 1 1
4 1 1 1 1
5 1 1 1 1
6 1 1 1 1
7 1 1 1 1
8 1 1 1 1
9 1 1 1 1
10 1 1 1 1
11 1 1 1 1
12 1 1 1 1
```
13  1  1  1  1
14  1  1  1  1
15  1  1  1  1
16  1  1  1  1
17  1  1  1  1
;
x  :=  
;
w;  #empty

II  =  8

set  match[10]  :=  ADDRGP4;
set  match[12]  :=  ADDRGP4;
set  match[22]  :=  LDP4.ODDRARA LDP4.ODDRASYMBOL;
set  match[23]  :=  LDP4.ODDRARA LDP4.ODDRASYMBOL;
set  match[41]  :=  ADD4.EVENRARARA ADD4.EVENRARACONST;
set  match[42]  :=  ADD4.EVENRARARA ADD4.EVENRARACONST;
set  match[43]  :=  ADD4.EVENRARARA ADD4.EVENRARACONST;
set  match[51]  :=  ADD4.EVENRARARA ADD4.EVENRARACONST;
set  match[52]  :=  ADD4.EVENRARARA ADD4.EVENRARACONST;
set  match[53]  :=  ADD4.EVENRARARA ADD4.EVENRARACONST;
set  match[54]  :=  ADD4.EVENRARARA ADD4.EVENRARACONST;

\sum\{v \text{ in } G\} \text{ max}_{lat}[v] = 40

set  SIMD_sets[1]  :=  51 52 53 54;

SIMD  :=
1  EVEN  18  1
2  ODD  0  1

; Align32 \[*,*,0\] (tr)
:  12  20  21  22  23  41  42  43  44  51  :=
  0  1  0  0  1  0  .  .  .  .
  1  1  0  0  1  0  .  .  .  .
  2  1  0  0  1  0  .  .  .  .
  3  1  0  0  1  0  .  .  .  .
  4  1  0  0  1  0  0  0  1  0
  5  1  0  0  1  0  0  0  1  0
  6  1  0  0  1  0  0  0  1  0
  7  1  0  0  1  0  0  0  1  0
  8  1  0  0  1  0  0  0  1  0
  9  1  0  0  1  0  0  0  1  0
 10  1  0  0  1  0  0  0  1  0
 11  1  1  0  1  0  0  0  1  0
 12  1  1  0  1  0  0  0  1  0
 13  1  1  1  1  0  0  0  1  0
 14  1  1  1  1  0  0  0  1  0
```
A.3 Solution for graph in Figure 6.3

\[
\begin{bmatrix}
15 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
16 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
17 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
18 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[\]
7 1 1 1 0 1
8 1 1 1 0 1
9 1 1 1 0 1
10 1 1 1 0 1
11 1 1 1 0 1
12 1 1 1 0 1
13 1 0 1 0 1
14 1 0 1 0 1
15 1 0 1 0 1
16 1 0 0 0 1
17 1 0 0 0 1
18 1 0 0 1 1

xa :=
20 2 0 7 1
21 1 0 9 1
23 2 0 10 1

shuffDataForSimd :=
1 11 2 1

shuffDataForSimdR :=
1 13 2 1

set TREG := 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18;
set slots_SIMD[1] := 6 7 8 9 10 11 12 13 14 15 16 17 18;
set slots_SIMD[2] := 0 1 2 3 4 5 6 7 8 9 10;

set slots[10] := 0;
set slots[12] := 0;
set slots[13] := 0;
set slots[20] := 0 1 2 3 4 5 6 7 8 9 10;
set slots[21] := 0 1 2 3 4 5 6 7 8 9 10;
set slots[22] := 0 1 2 3 4 5 6 7 8 9 10;
set slots[23] := 0 1 2 3 4 5 6 7 8 9 10;
set slots[41] := 4 5 6 7 8 9 10 11 12 13 14 15 16;
set slots[42] := 4 5 6 7 8 9 10 11 12 13 14 15 16;
set slots[43] := 4 5 6 7 8 9 10 11 12 13 14 15 16;
set slots[44] := 4 5 6 7 8 9 10 11 12 13 14 15 16;
set slots[51] := 6 7 8 9 10 11 12 13 14 15 16 17 18;
set slots[52] := 6 7 8 9 10 11 12 13 14 15 16 17 18;
set slots[53] := 6 7 8 9 10 11 12 13 14 15 16 17 18;
set slots[54] := 6 7 8 9 10 11 12 13 14 15 16 17 18;
Appendix B

Output from step 1 of the code generator, for the data flow graph in Figure 6.1

The set $SIMD\_sets$ is the set $SC$ in Chapter 3.
The set $simdOperands[x,0]$ and $simdOperands[y,1]$ are the sets $OPL$ and $OPR$ respectively, both described in Chapter 3.
The set $overlappingSIMD$ is the set $OS$ described in Chapter 3.

```plaintext
param SIMDSsetSize := 23;
set SIMD\_sets {k in 1 \ldots SIMDSsetSize};
let SIMD\_sets [1] := \{51,50\};
let SIMD\_sets [2] := \{44,43\};
let SIMD\_sets [3] := \{44,42\};
let SIMD\_sets [4] := \{44,41\};
let SIMD\_sets [5] := \{43,42\};
let SIMD\_sets [6] := \{43,41\};
let SIMD\_sets [7] := \{42,41\};
let SIMD\_sets [8] := \{23,22\};
let SIMD\_sets [9] := \{23,21\};
let SIMD\_sets [10] := \{23,20\};
let SIMD\_sets [12] := \{22,20\};
let SIMD\_sets [13] := \{21,20\};
let SIMD\_sets [14] := \{44,43,42\};
let SIMD\_sets [15] := \{44,43,41\};
let SIMD\_sets [16] := \{44,42,41\};
let SIMD\_sets [17] := \{43,42,41\};
let SIMD\_sets [18] := \{23,22,21\};
let SIMD\_sets [19] := \{23,22,20\};
let SIMD\_sets [20] := \{23,21,20\};
let SIMD\_sets [21] := \{22,21,20\};
let SIMD\_sets [22] := \{44,43,42,41\};
let SIMD\_sets [23] := \{23,22,21,20\};
set simdOperands {k in 1 \ldots SIMDSsetSize, lr in 0 \ldots 1};
let simdOperands [1,0] := \{2,4,5,6,7,14,15,16,17,22\};
let simdOperands [1,1] := \{2,3,4,5,7,14,15,16,17,22\};
```
let simdOperands [2, 0] := [8, 9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [2, 1] := [8, 9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [3, 0] := [8, 9, 10, 11, 13, 18, 19, 20, 21, 23];
let simdOperands [3, 1] := [8, 9, 10, 11, 13, 18, 19, 20, 21, 23];
let simdOperands [4, 0] := [8, 9, 10, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [4, 1] := [8, 9, 10, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [5, 0] := [8, 9, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [5, 1] := [8, 9, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [6, 0] := [8, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [6, 1] := [8, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [7, 0] := [9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [7, 1] := [9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [8, 0] := {};
let simdOperands [8, 1] := {};
let simdOperands [9, 0] := {};
let simdOperands [9, 1] := {};
let simdOperands [10, 0] := {};
let simdOperands [10, 1] := {};
let simdOperands [11, 0] := {};
let simdOperands [11, 1] := {};
let simdOperands [12, 0] := {};
let simdOperands [12, 1] := {};
let simdOperands [13, 0] := {};
let simdOperands [13, 1] := {};
let simdOperands [14, 0] := [8, 9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [14, 1] := [8, 9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [15, 0] := [8, 9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [15, 1] := [8, 9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [16, 0] := [8, 9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [16, 1] := [8, 9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [17, 0] := [8, 9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [17, 1] := [8, 9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [18, 0] := {};
let simdOperands [18, 1] := {};
let simdOperands [19, 0] := {};
let simdOperands [19, 1] := {};
let simdOperands [20, 0] := {};
let simdOperands [20, 1] := {};
let simdOperands [21, 0] := {};
let simdOperands [21, 1] := {};
let simdOperands [22, 0] := [8, 9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [22, 1] := [8, 9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let simdOperands [23, 0] := {};
let simdOperands [23, 1] := {};
set overlappingSIMD {k in 1..SIMDSetSize};
let overlappingSIMD [1] := {};
let overlappingSIMD [2] := [3, 4, 5, 6, 14, 15, 16, 17, 22];
let overlappingSIMD [3] := [2, 4, 5, 7, 14, 15, 16, 17, 22];
let overlappingSIMD [4] := [2, 3, 6, 7, 14, 15, 16, 17, 22];
let overlappingSIMD [5] := [2, 3, 6, 7, 14, 15, 16, 17, 22];
let overlappingSIMD [6] := [2, 4, 5, 7, 14, 15, 16, 17, 22];
let overlappingSIMD [7] := [3, 4, 5, 6, 14, 15, 16, 17, 22];
let overlappingSIMD [8] := [9, 10, 11, 12, 13, 18, 19, 20, 21, 23];
let overlappingSIMD [9] := [8, 10, 11, 13, 18, 19, 20, 21, 23];
let overlappingSIMD [10] := [8, 9, 12, 13, 18, 19, 20, 21, 23];
let overlappingSIMD [11] := [8, 9, 12, 13, 18, 19, 20, 21, 23];
let overlappingSIMD [12] := [8, 10, 11, 13, 18, 19, 20, 21, 23];
let overlappingSIMD [13] := [9, 10, 11, 12, 18, 19, 20, 21, 23];
let overlappingSIMD[14] := {2,3,4,5,6,7,15,16,17,22};
let overlappingSIMD[15] := {2,3,4,5,6,7,14,16,17,22};
let overlappingSIMD[16] := {2,3,4,5,6,7,14,15,17,22};
let overlappingSIMD[17] := {2,3,4,5,6,7,14,15,16,22};
let overlappingSIMD[18] := {8,9,10,11,12,13,19,20,21,23};
let overlappingSIMD[19] := {8,9,10,11,12,13,18,20,21,23};
let overlappingSIMD[20] := {8,9,10,11,12,13,18,19,21,23};
let overlappingSIMD[21] := {8,9,10,11,12,13,18,19,20,23};
let overlappingSIMD[22] := {2,3,4,5,6,7,14,15,16,17};
let overlappingSIMD[23] := {8,9,10,11,12,13,18,19,20,21};
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Magnus Pettersson]