Code Compaction and Parallelization for VLIW/DSP Chip Architectures

by

Tsvetomir P. Petrov

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

Author ....
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by ....
Saman P. Amarasinghe
Assistant Professor, MIT Laboratory for Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Theses

Code Compaction and Parallelization for VLIW/DSP Chip Architectures

by

Tsvetomir P. Petrov

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999, in partial fulfillment of the
requirements for the degree of
Master of Engineering

Abstract

The Master of Engineering Thesis presented in this paper implements an assembly code optimizer for Discrete Signal Processing (DSP) or Very Long Instruction Word (VLIW) processors. The work is re-targetable and takes as input minimal generalized chip and assembly language syntax description and unoptimized assembly code and produces optimized assembly code, based on the chip description. The code is not modified, but only re-arranged to take advantage of DSP/VLIW architecture parallelism. Restrictions are placed to make the problem more tractable and optimality is sought using linear programming and other techniques to decrease the size of the search space and thus performance close to that of native compilers is achieved, while maintaining retargetability. This document discusses motivation, design choices, implementation details and algorithms, performance, and possible extensions and applications.

Thesis Supervisor: Saman P. Amarasinghe
Title: Assistant Professor, MIT Laboratory for Computer Science

Contents

List of Figures

Chapter 1

Introduction

In recent years the advances in process technology and chip design have led to many high performance chips on the market, at more and more affordable prices per computational power. At the same time, the “smart devices” model of computation has also been gaining ground, giving rise to a large variety of embedded systems utilizing high performance, often highly specialized chips.

In particular, many of those chips have been designed to address the needs of signal processing application such as digital communications and digital television, computer graphics, simulations, etc. Being intended for a large production volumes, cost has been an even more pressing issue and a lot of the designs do not have the complicated optimization circuitry doing branch prediction, pre-fetching and caching, associated with modern general purpose microprocessors.

Because of the simplicity of the hardware design, which makes parallelism very explicit to the user, most of the optimizations have been left to the programmer. Still, providing optimal or nearly optimal solutions to the optimization problems in the transition to high performance chips has been both interesting, challenging and difficult.

This work presents an attempt at optimal or nearly optimal utilization of computing capabilities for high-performance chips. Given an unoptimized or partially optimized assembly code, this software tries to produce optimal or nearly optimal version of the same code for a specific DSP/VLIW chip. In order to make the work more practical, fairly general chip descriptions are allowed.

1.1 Motivation

1.1.1 Hand Optimization Techniques are not Scalable

While code parallelization of assembly code for VLIW/DSP chips has been an active research issue for decades, in most cases even now, assembly code for powerful DSP and other VLIW chips is hand-crafted and optimized. Unfortunately, a lot of the techniques employed by humans in this process (if any systematic techniques are employed at all) do not scale very well. For example, the introduction of the Texas Instruments TMS320C6000 series, capable of issuing up to 8 instructions with variable pipeline lengths per cycle, marks a new era, where it becomes increasingly hard to hand-optimize code with that much parallelism, and in fact, to even write remotely optimal code. In fact, one can argue that software such as the one presented here, extended with more high-level optimizations, could often produce better results than humans on long and algorithmically-complicated code. Thus, given the tendency of creation of even more powerful chips and the use of more convoluted algorithms, scalability becomes more and more important issue.

1.1.2 Hand-Optimized Code is not Portable

Not only techniques for hand optimization are not scalable, but they are also not portable. Historically, each generation of DSP chips has been taking advantage of process technology to optimize the instruction set architecture, because it is too costly in terms of power and gates to emulate a single instruction set architecture. Thus, with the introduction of new more powerful and different chips, all optimizations to existing code need to be re-done, which is a long and difficult process, involving both coding, optimization and validation. A generalized tool that provides good code optimization, such as the one presented here, would certainly make the transition much faster.

1.1.3 Code Optimizer as an Evaluation Tool for New Chip Designs

In addition to easing the transition between generations of chips, a code optimizer, extended with simulation or profiling capabilities can be very helpful in determining what features in a new chip are most useful. Given the long turnaround times in hardware design and the often fixed application code, a tool enabling specification of hardware capabilities and providing ballpark figures for running time of optimized code, could be very valuable just for that reason.

1.2 Benefits of Focusing on VLIW/DSP Chip Architectures

1.2.1 Widespread Use and Variety of Algorithms

Well, obviously, in order to talk about parallelization, we have to have the capability of executing many instructions in parallel and DSP/VLIW chips have this capability. In addition, they are widely used and very likely to grow in both usage and capabilities. Given the variety of applications those chips are used for (and the tendency of this variety to grow, for example, with the introduction of Merced, (backwards compatible with Intel x86 family)), seeking a general-purpose solution versus trying to optimize every algorithm for every chip is a better approach.

1.2.2 Generally More Limited Instruction Set and Simpler Pipelines

Since code parallelization is a hard problem, trying to solve it on a simpler architectures is helpful. Since most DSP/VLIW chips have a more limited instruction set and their pipelines are relatively simpler, one can achieve better optimization results.

1.2.3 Design Peculiarities Make Optimizations Difficult

While some of the high-performance DSP/VLIW chips (such as TI TMS320C6000 series) have had a very general, streamlined design, many of the older and more specialized DSP chips have had a very programmer-unfriendly design. In particular, because of the limited width of the instruction words, in high performance instruction word encodings only certain combinations of operands might be allowed, while in easier-to-encode, but less parallel instruction words all combinations are present. For example, on Qualcomm's proprietary QDSP II, in a cycle with 5 parallel instructions, the operands of multiply-accumulate instruction can only be certain sets of registers, although data paths exist for more registers to be operands and indeed, if the MAC instruction is combined with less instructions, more operand register sets can be used.

While the above problems are especially acute with more specialized chips, they are present in most VLIW/DSP chips to some extent. Thus, just learning what the instruction set is, can be an involved process with programmer's manuals easily exceeding hundreds and often a thousand of pages. While all those issues make optimization more difficult for both humans and compilers, as discussed in Section 1.1, only compilers are capable of providing scalable and portable solutions.

1.2.4 Performance Requirements Justify Longer Optimization Times

Finally, last, but not least, most of the applications currently running on DSP/VLIW chips are applications in a framework of very limited resources and very rigid requirements - both in terms of data memory, instruction space, and running time.

In addition, because many of them are used in embedded systems, often the problems solved are smaller and more self-contained and easier to achieve optimal or near-optimal results.

Finally, in embedded systems, even if resources are available, optimizations are still highly desirable, because of power consumption considerations, which often require running at the lowest clock speeds or limiting code size.

Thus, contrary to general purpose processors where performance might be sacrificed for speed of compilation, with most applications on DSP/VLIW chips, it is imperative that the the code is highly-optimized and therefore, slow, but good tools, as the one proposed here, are viable.

Chapter 2

Approach

Having described the motivation and the focus of the project, it is now time to define its place amidst other work in the field and define more clearly its objectives and approach to achieving them.

2.1 Related Work

There has been a fair amount of work in the field with many semiconductor companies releasing powerful chips and compilers that do some optimizations as commercial solutions. In fact some of the inspiration behind this work come from looking of older version of assembly optimizer tools running in real time and attempting to achieve better and more general solutions. The latest version (February, 1999) of the Optimizing C Compiler and Assembly Optimizer [24, 25, 26, 27] for TMS320C6201 by Texas Instruments is used for comparison and examples, throughout this paper. In fact they produce very good (but not always optimal) results but are limited by development time and real-time compilation constraints. In addition, such commercial tools are understandably non-retargetable.

Besides commercial code optimization solutions there has been a lot of research into retargetable compilation. Most of the proposed solutions are complicated and encompassing all from structural chip specification, register allocation and instruction scheduling. MIMOLA ([22]), for example, starts from net list structural description;

AVIV ([11]), CodeSyn ([23]) and Chess ([14]) start from somewhat higher level chip description, but go through register allocation, mapping high level instructions into assembly ones using tree covering and code compaction (as presented in this work). Solutions like those have more potential for overall improvement, but employ heuristic solutions for code compaction and most likely would benefit from replacing their code compactor with the tools in this paper.

Other publications relevant to this include work on languages for chip description (several varieties, depending on granularity [17, 19] - some not much different than the one chosen in this work) and work on dynamic programming, linear programming (Wilson et al, [16]) and other techniques ([1-10]) for code compaction. Many of those techniques are computationally infeasible (such as solving the entire optimization problem as a single integer linear program) or partially used in this work (linear programming for obtaining bounds), or inapplicable, because of different computation framework.

2.2 Attempt at Retargetable Optimality

Amidst many complicated solutions, the intent of this work was to be fairly practical and thus the following emerged as main priorities:

- simple, but retargetable, chip and assembly description - the first step to optimizations is describing the chip architecture - if this is a daunting task, people might as well not even attempt it
- attempt at optimal solutions - unlike many other tools, this work makes an attempt at provably optimal solutions, given some constraints
- attempt at capturing many varieties of architectures, but not at the price of less optimizations - if some instructions cannot fit in the optimization framework, they are left unchanged

Given those priorities, among some of the contributions of this work are:

- devising simple, yet powerful, assembly and chip description language
- using techniques to make optimizations more tractable
- describing extensible infrastructure for doing such optimizations

2.3 Restricting the Problem to Re-arranging Code

Probably the most restrictive choice in achieving simplicity, was that code was not to be modified, but just re-arranged. All valid re-arrangements allowable by the data dependency constraints are considered, however. This requirement might seem too restrictive from optimization point of view, but in practice it is not, if all other standard optimizations such as dead code elimination, constant propagation, common subexpression elimination and so on, are already performed. Peephole optimizations can be used to find better instruction combinations.

In fact, the standard compiler optimizations are only applicable if compiling from a high-level language - if the input is hand-crafted, those usually do not even apply. In addition, this code can run standalone after any other opimizing compiler has done its job, which is a good from systems design perspective.

2.4 Generalized Chip and Assembly Language Description

In order to make the work more practical, a fairly general chip description is allowed at a conveniently high level. In particular, the chip is described in terms of combinations of assembly instructions that can be executed in a single cycle. The description of assembly instructions itself contains both parsing information to allow describing a fairly general custom assembly language syntax as well as information on data dependencies and branch delay slots associated with every instruction.

Describing the system at a such a high level is simpler and easier and has the added advantage of being less prone to errors than a lower level hardware description in terms of functional units and datapaths.

2.5 Assuming Memory Aliasing and Basic Block Self-Sufficiency

In order to determine whether memory accesses are aliased or not, a whole new framework and a lot of infrastructure need to be created, and, it is still impossible to always determine that at compile time, especially when dealing with identifiers and/or pointer arithmetic inherited from a higher-level language (such as C). Thus, in this work all memory accesses to the same memory unit are assumed to be dependent.

Similarly, by default, all labels are considered possible entry points (to allow for optimizing of single components of a large hand-optimized application, for example) and they limit the size of basic blocks for optimization purposes. For those purposes, all calls or jumps are also delimiters, because of the difficulty (or impossibility) of obtaining information for required and modified resources for arbitrary (or non-present) procedures. Some of those restrictions can be relaxed, through user hints, however. Furthermore optimizations such as loop unrolling and trace scheduling can be used to provide larger blocks, if necessary.

2.6 Modularity

Modularity as a design choice is present in two levels:

- The optimization tools employ modular design in order to allow for extensibility and maintainability. An attempt at defining simple and easy to debug interfaces between modules has been taken and indeed many of the modules can be easily replaced or modified without major modifications to the rest of the code. An overview of all modules can be found in Chapter 3.
- The optimizations are performed in modular fashion in order to guarantee scalability. Optimizations are mostly done on basic blocks and although that might

sound too restrictive, in practice it is not, because often, in general, code cannot be moved outside certain bounds (see Section 2.5). An added benefit of this modular approach to processing is that this work might further be parallelized or restricted to optimizing just parts of application code with very little effort.

Chapter 3

Module Overview

Figure 3-1 contains a diagram of the basic modules of the optimizer and the flow of data through them. The process of optimizing assembly files for a particular architecture can roughly be partitioned into 3 phases:

- The first pre-requisite is to describe the chip capabilities and assembly syntax and run it through the Chip Description Analyzer and Parser Generator tool. This only needs to be done once per chip description. Those tools generate a parse tree and other data structures representing the chip for use later in the process and can easily be modified or replaced to suit other assemblies or different chip description styles. More detail on the input language and examples can be found in Chapter 4.

- Having created the necessary infrastructure to support given assembly syntax and chip capabilities once, one can run many assembly files through the optimizer. However, since it is desirable to take most of the work out of the optimization loop, the code goes through several transformations before arriving there. In particular:

  - The code is being parsed into internal representation using predictive parser (allowing for a wider range of grammars to be supported).
  - The code is then split into basic blocks. Doing so is essential for scalability. It is also useful as an abstraction barrier. For the motivation of making this

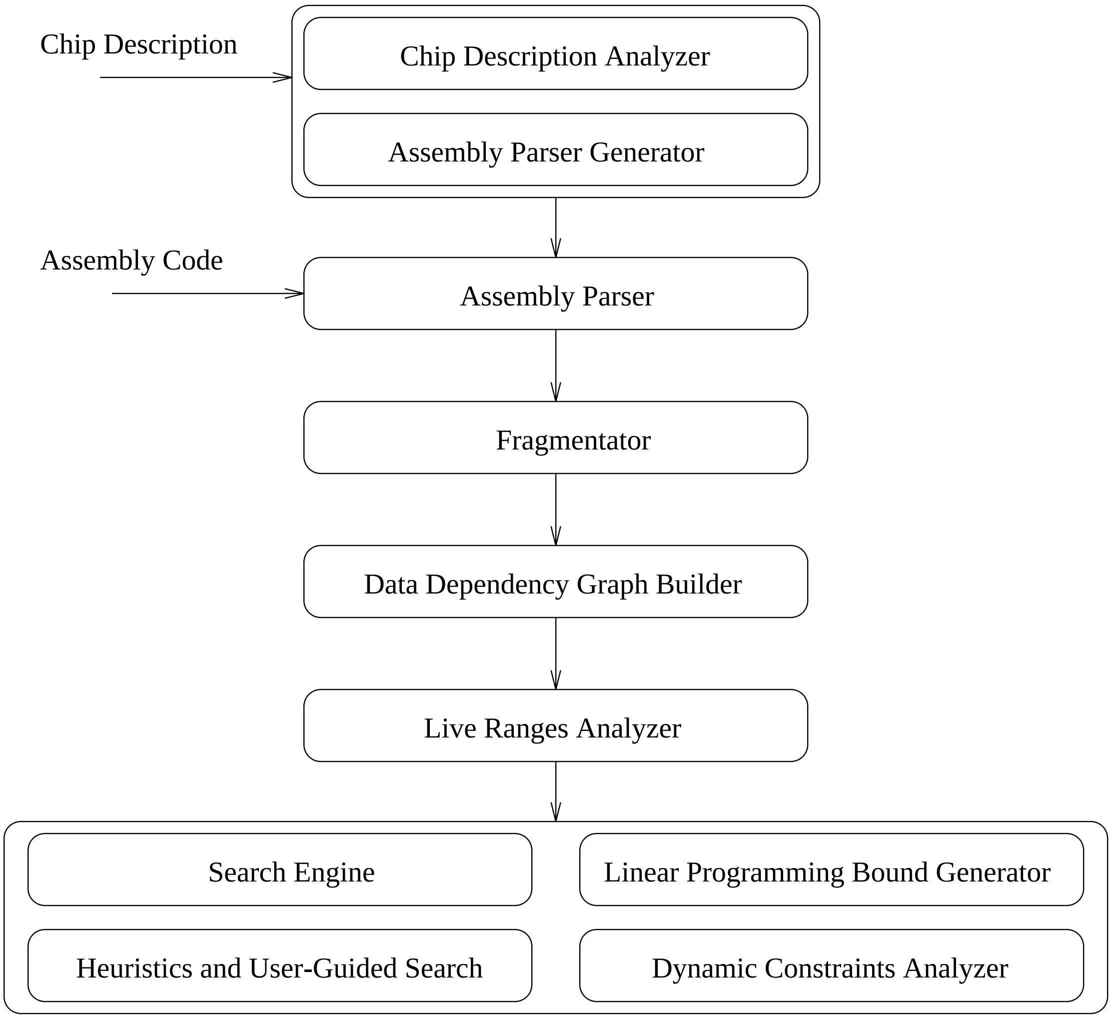

Figure 3-1: Module Overview

a separate stage and discussion of some of the issues involved in splitting the graph see Section 5.2.

- Once the code is fragmented into basic blocks, a data dependency graph (DDG) preserving the most parallelism is derived. Note that the representation and terminology used in that is somewhat innovative. See Chapter 5 for more details.
- Even though the data dependency graph is a representation of the optimization problem preserving correctness, there are additional properties of the data dependency graph (DDG) that would enhance the search for solution. They are described in Section 5.4.

- Finally, given a data dependency graph and attempt at an optimal or nearly optimal is made. The basic search framework is branch and bound with bounds provided using linear programming, minimum distance to end and other methods, heuristic hints and directions provided by the user and an early detector of futile branches of the search space. All those components are closely interleaved in the code, but they are described in separate chapters, because each one of them contains interesting algorithms and other issues.

Chapter 4

Describable Chip Architectures

Having described the scope of the project and given an overview of the tools, it is now time to focus in detail on the range of chips describable within Assembly Description Language (ADL) of this project. The general idea of the language is to be able to describe the syntax of all assembly instructions together with with the minimum required information about how they can be reordered. In order to highlight the process of arriving at that language the supported features will be described first.

4.1 Description Semantics

All chip architectures that can be described correctly within the description semantics listed below can benefit from this work. Furthermore, even if the chip as a whole cannot be fully or correctly described within this framework, often by eliminating offending instructions or situations (eg. waiting and servicing interrupts, etc) from consideration, one could make this work applicable to many more architectures.

4.1.1 Multiple Instruction Issued on the Same Cycle

The description of the chip capabilities is in terms of assembly instructions capable of being issued at the same cycle. In the simple case of non-pipelined processors this translates to sets of instructions that can execute together in a single cycle. In fact, all pipeline stages common to all instructions can be viewed as occurring in a single cycle for the purpose of extraction of data dependencies from the source code in the case of pipelined processors with same length non-blocking pipelines for all instructions and no delay slots. Eliminating stages in this fashion is helpful when reasoning about the chip.

4.1.2 Pipeline Delay Slots Support

Recognizing, however, that many chips feature variable length instruction pipelines and delay slots (most commonly with branches, but also with other instructions) such capabilities are also supported. Different delay slots for different resource updates are also supported. An example of such situation would be a memory load instruction with address register modification. The address register modifications typically are in effect for the instructions issued on the next cycle, while the register getting data read from memory might not be modified until several cycles later.

4.1.3 Resource Modification Oriented Description

The description of the chip functionality for the purposes of detecting dependencies is based on specification of used and modified resources for each instruction. Registers are typical such resources. Each instruction lists the registers it requires and modifies. For the purposes of defining the chip semantics correctly unlimited number of other resources can be defined with the same semantics as registers. Memory is a typical example of such resource. Such additional resources can be used to capture “hidden” registers such as the ones used in stack modifications, register flags, special modes of operations and more. See Section 4.3 for examples.

Here are some of the key properties of that description (for simplicity, single-cycle execution terminology is used; non-blocking pipeline semantics are the same):

- The values of all used resources are assumed to be read at the very beginning of the cycle. Thus, if the same resource is used and modified at the same cycle, the old value is used in computation.

- The modification of all modified resources is assumed to happen at the end of the current cycle (ie. the modified value is to be used for the instructions issued on the next cycle), unless delay slots for the modification of that resource are specified, in which case the change is assumed to happen the specified amount of delay slots later.

- The same resource cannot be modified by two different instructions in the same cycle, because of unresolved ambiguity about the order of modification.

- Each resource can be used by any number of instructions and any number of operands in a given cycle. To describe more stringent constraints based on use of hardware data paths and so on, one could introduce resources for those data paths and specify their modification as required (see Section 4.3 for examples).

4.1.4 Conditional Instruction Execution Support

Many modern chips allow conditional execution of instructions based on the value of a certain register or flag. Conditional jumps or loops are the most essential examples of such instructions, but support for any conditionally executed instruction is included. Unless there is control flow transfer the semantics of those instructions are not much different than any other instructions - the only difference is that the register or flag being tested is an "used resource" in the above sense.

4.1.5 Support for Common Control Flow Instructions

Control flow operations are problematic in chip descriptions because they are less standardized in encoding and functionality from chip to chip. Specialized loop instructions and their placement right before, after or within the loop body can be a problem in describing the chip assembly and capabilities. Several common cases of such placements are supported. The work can easily be extended to support others. Calls, jumps and other similar instructions are supported, together with their conditional and delay slot versions. For complete description see Section 5.2.

4.2 Chip Description Syntax

For an added convenience and ease of description, the description of the chip will consist of description of all possible valid instructions in assembly. Note that the description might aceept invalid entries as well, as long as it is not ambiguous - on invalid input, the output is guaranteed to be invalid! All instructions, then, would be partitioned into instuctions sets or classes, such that all instructions in the same class combine in the same way with all other instructions (for example the class of ALU instructions). Finally, all possible combinations of instructions that can occur on a single cycle will be described in terms of instruction classes. Note that a class can consist of a single instruction, if it combines uniquely with instructions from other classes.

The description of a single instruction or a several instructions together when using shortcuts (see below) consist of a name, list of tokens and 3 other lists (optionally empty). The 3 additional lists are used to determine the data dependencies and are *requires*, *modifies* and *special* list respectively. Each element of the *requires* and *modifies* lists is either a number indicating which token in the token list (representing register) is meant, or a string identifying other resource or a particular register. An entry in the modifies list starting with the symbol '#' signifies signifies delay slots in the modifications of the resource immediately preceding it. The *special* list contains the type of a control-flow operation (eg. conditional, JUMP, JUMPD, CALL) and other information, if necessary.

4.2.1 Convenience Features

In order to make the process of describing the chip and its assembly language easier the following are supported:

- sets of registers
- sets of operators and other symbols

- use of _ to signify optional nothing or several tokens always used together (eg. << _16) in operator descriptions
- the special token STRING matches any alphanumeric sequence - useful as a placeholder for constants or identifiers in the source code
- definition of related resources (eg. if R0 is 16-bit and its 8-bit parts are R0h and R0l, then whenever R0 is modified, R0h and R0l are modified and vice versa) - useful in register sets

4.3 Examples

Having described the describable chip architectures in general and the description syntax features in detail, it is now time to give some examples.

Some of the chip architectures described within this framework include QDSP II by Qualcomm Incorporated and TMS320C601 by Texas Instruments. However, QDSP II is Qualcomm proprietary and therefore will not be used for examples, while information about TMS320C6201 is publicly available and it is widely used chip, so the examples here will be based on it.

4.3.1 Overview of TMS320C6201

TMS320C6201 is an Very Long Instruction Word chip, capable of issuing up to 8 instructions per cycle. Most instructions execute in a 11-cycle pipeline and the chip can be clocked at 200 MHz making it one of the most powerful fixed point integer DSP chips on the market. It features two sets of nearly identical computational units and two general purpose register files associated with each set. The shematic can be found on Figure 4-1. The L units perform long arithmetic operations such as addition and subtractions. The S units perform similar operations on different arguments. The M units perform 16 bit multiply operations and the D units perform memory loads and stores. The L, S and M units can only write to their register block, while the D units can store or load data from or to both register files. There are only 2 cross

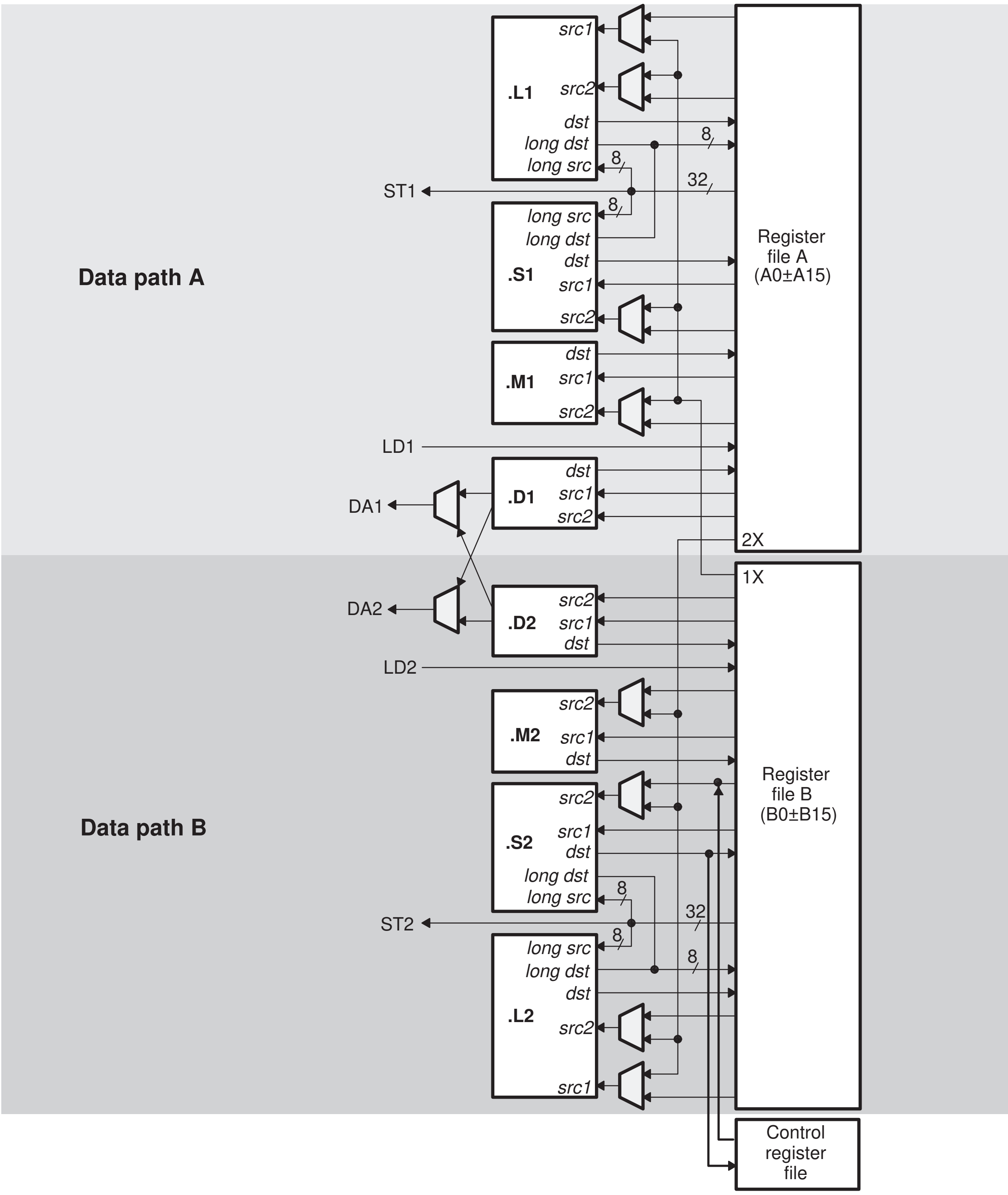

Figure 4-1: TMS320C6201 Schematics, [24]

register path so only one of the arguments of the L, S, M units on each side of the chip can come from the other side. The arguments to the D units must come from their respective part of the chip.

```
RSet AX        A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 ;
RSet BX        B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15 ;

OSet ADDLX     ADDL ADDLSU ADDLU ADDLUS ;
OSet Number    STRING -_STRING ;

inst ADDL1     ADDLX AX , AX , AX ; 2 4 ; 6 ; ;
inst ADDL2     ADDLX Number , AX , AX ; 4 ; 6 ; ;
inst ADDL3     ADDLX BX , AX , AX ; 2 4 ; 6 x1 ; ;
inst ADDL4     [ AX ] ADDLX AX , AX , AX ; 2 5 7 ; 9 ; 13 ;
inst ADDL5     [ AX ] ADDLX Number , AX , AX ; 2 7 ; 9 ; 13 ;
inst ADDL6     [ AX ] ADDLX BX , AX , AX ; 2 5 7 ; 9 x1 ; 13 ;

ISet UnitL1    ADDL1 ADDL2 ADDL3 ADDL4 ADDL5 ADDL6 ;
```

Figure 4-2: Sample Definition of Addition Instructions

4.3.2 Sample Instruction Definitions

Given the above functionality of the chip and the format of its assembly, a sample definition of a set of instructions executable on unit L1 is given on Figure 4-2. (Note that the *ISet* definition above, in practice, should include all other instructions executable on that unit).

First AX and BX are defined as enumerations (sets) of registers. Then ADDLX is defined as the set of several different instruction mnemonics for addition - note that for the purposes of this work it is not important what the operation really is, as long as the required and modified resources are identified correctly. Since the addition instructions might make use of a constant operand, Number is defined as an optionally signed alphanumeric sequence. Here, again the value of the operand is immaterial. It does not even matter whether it is a number or a string constant in the assembly - what is important is that there is no data dependency on it.

The 3 lists separated by commas in each *inst* statement indicate the token indices of required and modified resources. A special resoure 'x1' corresponding to the cross data path is introduced to make sure that only one operand of L, S or M unit comes from it. Finally, the square bracket notation indicates conditional execution, which is signified by the special code in the third semi-colon separated list.

Once the instruction dependencies and syntax are described in *inst* statements, the *ISet* statement is used to define a set of instructions that combine uniquely with all others. While on other chips those combinations can be very involved on the TMS320C6201, describing all possible combinations of instructions is done by something like:

```
Combo UnitL1 UnitL2 UnitS1 UnitS2 UnitM1 UnitM2 UnitD1 UnitD2 ;
```

Describing the rest of the assembly is just a simple addition of more syntactic structures. Some of the other notable features of it are:

- one delay slot for the result of all multiply (MPY) operations on the M units
- four delays slots for the result of memory read (LD) operations on the D units
- five branch delays slots for branch instructions
- conditional execution of single instructions based on register value

The description of TMS320C6201 chip used for optimizations can be found in Appendix A.

Chapter 5

Data Dependency Graph (DDG)

Once the unoptimized assembly file is parsed in, but before optimizations can be done, a data dependency graph has to be generated in order to insure the correctness of the output, and make it easier to pinpoint allowable instruction permutations.

5.1 Parser Output

The assembly parser takes in the unoptimized assembly file and produces a list of assembly instructions each of which contains - its original parse string (in order to allow printing it out) and the instruction word in the input in which it would have been encoded. Labels are preserved in a similar fashion. The used and modified resources are resolved to internal representation as well. This list is then passed on for further processing.

5.2 Fragmentator

Since assembly files for real applications can be quite large and searching for optimal or near-optimal solution in the optimization phase grows very fast with the size of the problem, being able to split the input file into small blocks for further processing is a necessity in order to provide a scalable solution.

5.2.1 Basic Case

The fragmentator module attempts to break up the input into more manageable blocks such that performance is not sacrificed if all blocks are optimized individually. In other words, those are blocks of code, where code cannot be moved into or outside the limits of the block without correctness hazards. It is easy to establish that there are basically two types of block delimiters:

- *label* - in order to allow for partial compilation, all labels are considered possible entry points and therefore code cannot be moved beyond them and code can be safely partitioned at them

- *control flow instructions* - analogously, because the points where control is transfered might be unavailable or called from many places, no code can be moved beyond a control flow instructions (such as LOOP, JUMP, CALL, RETURN)

5.2.2 Special Cases

Note that there can be complications if the those special instructions are conditional and they have delay slots. Specifying branch delay slots is supported and in order to insure correctness under the above partial compilation assumptions, two possible modes of operation are supported depending on the chip architecture:

- *branch delay instructions always executed* - in this mode all instructions placed in the branch delay slots of a conditional flow control instruction are included in its basic block together with the code before it

- *branch delay slots executed only if branch not taken* - since whether the branch will be taken or not cannot be determined at compile time the instructions in the branch delay slots cannot be moved outside those slots and therefore constitute a basic block by themselves - there is no point in optimizing it, however, since the possible gains have to be filled with NOPs anyway

Note that it is unwise to have instructions with delay slots past a control flow change (e.g. a RETURN instruction), because this can give rise to resource conflicts unless it is known where flow is transferred. Such situations are flagged and disallowed by making sure that all instructions in a basic block have their delay slots filled (possibly with NOPs) within the block.

5.2.3 Optimization Controls at Fragmentator Level

In addition to being essential to scalability, the fragmentator is an useful abstraction barrier since many optimization directives can be applied at that level. For example, the user can specify different optimization strategies for the current basic block, might relax some correctness assumptions (eg. that all resources are needed upon exit, which influences data dependency graph generation) or might provide a "weight" of the block (eg. in an attempt to gain profiling information for the amount of speedup by optimizations).

5.3 DDG Builder

Given a (presumably small) basic block, it is now time to analyze the data dependencies and abstract the problem away from chip descriptions and program flow and assembly language into more of a problem of collapsing a colored graph into boxes, each of which can contain certain combinations of colors. (The colors here are the instruction sets and the boxes are what is being executed on each cycle).

5.3.1 Correctness Invariant

The key idea in being able to abstract away the assembly optimization problem is knowing what the key property guaranteeing correctness is. In fact, it turns out that it is very simple: if in the original code, instruction A uses resource B, which was last modified by instruction C, then the same should hold in the optimized code. In other words, for every pair of instructions where one modifies a resource (say instruction

M modifies resource R) that and the other (say instruction U) is using, before any other modifications to it, one could create a link from M to U, symbolizing that M has to execute before U and no other instructions modifying resource R can occur in between.

In order to accomodate delay slots the notion of the links can be extended by adding a delay slot field on them and adding to the semantics that for the next delay slot (*DS*) cycles the resource R is not modified, but it is modified at the end of the *DS* cycle following instruction M. Therefore, U should occur after that and no other modifications to resource R can occur after *DS* cycles after M, but before U.

5.3.2 Handling of Beginning and End of Block

In order to run the simple algorithm to create a DDG graph, however, one needs to pay special attention to the borderline cases - and specifically to the beginning and end of the block. One general enough and safe approach, commonly assumed, say for procedures, is to assume that the beginning modifies everything and the end requires everything. All that is saying is that if someone used a register value that was not initialized in the block then that should be true in the output as well, and that the original values of all resources upon exiting the block should be preserved. Using user defined pragmas at the fragmentator level, those can be overrided, resulting in more combinatorial possibilities and possibly more optimal code.

5.3.3 Properties of This Representation

It is important to note that all links like those capture both the data dependency constraints and the correctness invariants and in fact, they are both sufficient to guarantee correctness and at the same time preserve the most allowable parallelism, given our assumptions. Note that many other representations are more restrictive with respect to parallelism, but have less assumptions. By being restrictive at the basic block level we can achieve those properties at the DDG level.

Here are some other properties of the graph:

- If there is no link (or chain of links) from instruction to the end of the block the instruction is dead code and can be eliminated.

- If there is no link (or chain of links) from the start to an instruction then its result is independent of the state of the resources upon the entry of the basic block.

- There can be no two or more links with the same resource going into a node. This is equivalent to saying that there is only one instruction which last modified a certain resource.

- There can be many links with the same resource going out of a node, but they must have the same delay slots. What this is saying is that a modified resource might be used by many other instructions but is only modified at a certain time, before being used.

It can also be noted, that in this structure, if there is no link (or chain of links) from an instruction to the end of the block, the instruction is dead code and can be eliminated, and if there is no chain of links to the start, then it can be executed before the block (eg. an assignment statement).

5.4 Live Range Analysis and Optimizations

While the DDG structure outlined above is sufficient to guarantee correctness there is more analysis that can be done statically before even starting optimizations, particularly related to live ranges. Consider the case of two long sequences of code both using R0 as accumulator, except that the first one writes R0 to memory, while the second one produces the value of R0 available at the end of the block. Add to this all sorts of computation that can be executed in parallel with them. Now, if there are no large delay slots, one can conclude that the two accumulation sequences cannot be interleaved, and that, in fact, the second one should be completed after the first one, but this is not explicitly stated in the current form of the DDG.

What can go wrong in particular, is getting started on executing the second sequence and trying to combine it in all possible ways with other instructions, just to realize in the end that the first sequence cannot be executed. This situation can be amended by looking for continuous sequences of use and modification of a certain resource ending in a certain instruction, running a search backwards from there and adding constraints that the use or modify sequence should be executed no earlier than any other use of the same resource eventually leading to that instruction. Note that this optimization helps establish better bounds on shortest time to completion and thus it is performed before those bounds are calculated.

As can be seen, this is a very interesting issue with a great computation-saving potential, and attempting to solve it in the general case, involving many resources and delay slots would be interesting, but unfortunately this is beyond the scope of this work.

Chapter 6

Optimization Framework

6.1 General Framework

The process of optimizing a block of code given a data dependency graph consists of attempting to fill in instructions for each cycle and attempt to achieve a solution that is both correct and takes the least number of cycles. Some of the advantages of this framework are that it is fairly simple, natural and easier to trace by humans, that advancing and retracting by a cycle is relatively easy and that there is closure - meaning that the problem after several instruction words of instructions are chosen is similar to the original problem. Backtracking is supported and branch and bound or other varieties can be specified.

6.2 Basic Steps at Every Cycle

At every cycle, there are several basic steps:

1. Checking whether all instructions have been scheduled and a best-so-far solution has been achieved.

2. Checking whether bounds or heuristics indicate that no solution better than the current best can be obtained and, if so, going back a cycle.

3. Generating all combinations of instructions that can be issued on that cycle.

4. Saving the current state.
5. Selecting a non-attemted combination of instructions to try at the current cycle. If none going back a cycle.
6. Changing the state to reflect the current selection.
7. Advancing to the next cycle (a recursive call).
8. Restoring the state saved in step 4 and continuing from Step 5.

6.3 Definition of Terms

In order to describe the algorithms, it is necessary to introduce some terms and explain some of the state at each cycle.

6.3.1 Dynamic Link Properties

Links between DDG nodes are widely used in the optimization process, because they describe the constraints between nodes. In addition to the resource being modified and some other static data, each link has 2 major properties, that can vary with the cycle:

- *resource path length (RPL)* - the length of the longest path of links in the DDG, modifying the same resource without delay slots, starting from the current link - used as a lower bound on the number of cycles before any instruction using that resource, but not on that path can be scheduled
- *delay slots (DS)* - the delay slots signify the remaining number of cycles before the change in a modified resource takes place; for links whose start node has not been scheduled, the the delay slots are the original delay slots in the instruction definition; for links whose start node has been scheduled the delay slots decrease every cycle

Note that by definition, for every link at any time, if RPL>0, DS must be 0. Additionally, note that links that originally had delay slots, once their startnode is scheduled and all the delay slots have passed, behave in the same way as links without delay slots from then on. Indeed, links with delay slots turn into links without delays slots after DS cycles.

To summarize, if a link in the DDG has no delay slots, then it has DS=0 and RPL>=1, and that does not change. On the other hand, if the link has delay slots, then the DS field is initialized to the delay slots and RPL to 0 and once the starting node is scheduled, every subsequent cycle, DS is decreased until it reaches 0, when RPL is updated to a positive value as if the link had no delay slots. Those properties are restored to their correct value when backtracking.

6.3.2 The Ready List (LR) and the Links-to-unready List (LUL)

The Ready List (RL) is intended to be a list of all instructions that can be executed at the current cycle. At the beginning of the process in Section 6.2, it contains all instructions whose pre-requisites in the DDG have been fulfilled (the starting nodes of all links to them have been scheduled and all the links have their delay slots at 0). Some of them are later removed as unschedulable on the current cycle, through a complicated process.

The Links-to-unready List (LUL) is a list of all links whose start nodes have already been scheduled, but whose end nodes cannot be executed at the current cycle (ie. are not in the ready list).

Note that LUL entries impose restrictions on whether instructions can be in RL and might lead to the removal of RL entries. Note also, that the removal of instructions from RL leads to the placement of all their incoming links in the LUL.

6.3.3 Node Properties

Each node contains the following information:

- whether the instruction was scheduled and if so on what cycle
- if the node is not scheduled the highest DS and RPL values of all outgoing links for all resources
- its instruction set number, parse string and other similar information inherited from previous stages

6.4 Generation of All Executable Combinations

Since it is best to detect and eliminate combinations at the current cycle that will not lead to a feasible solution as early as possible, generating all executable combinations is a complicated process.

- RL is initialized as all non-scheduled instructions with fulfilled pre-requisites, all links pointing to RL elements are removed from LUL
- The Dynamic Constraint Analyzer (DCA) is called (see Chapter 7), and it could remove some elements of RL and put the links to them into LUL. Since the problem is more complex and just a list of executable instructions is inadequate, a structure representing more complex dependencies among the executable instructions is returned. In particular, certain instructions can only execute, if others execute on the same cycle, and some instructions can only execute all together or not at all, and this is captured in the data structure.
- Given that information a list of all allowable combinations is generated, making sure that all the restrictions are observed.
- The list of all combinations can then be heuristically sorted.

Chapter 7

Dynamic Constraint Analyzer

As discussed in Section 5.4, during the DDG generation phase, live ranges optimizations can result in great improvement in search times. However, many of the live range issues are only exhibited within a search framework and give raise to a whole new set of issues.

7.1 A Simple Example

```
b=abs(a)+1+a;
c=((a && 11) + 3) || 12;
```

Figure 7-1: Sampe C Code Fragment

Is is probably better to consider an example. Consider the C code on Figure 7-1. Some C compiler might translate it into the Assembly code given on Figure 7-2 - that translation assumes, a is stored in *A1*, b in *A2* and c in *B2*, and additionally, *A0* and *B0* are temporary variables.

If the assembly code is fed into the optimization tool and (assuming it is a complete basic block), the resulting DDG will be the one on Figure 7-3. The registers on each edge indicate the resource being used that should not be modified between the instructions (true dependency). Note, also, that on this graph, the dotted lines do

```
ABS  A1,A0;
ADDL 1,A0,B0;
ADDL A1,B0,A2;

ANDL 11,A1,B0;
ADDL 3,B0,A0;
ORL  12,A0,B2;
```

Figure 7-2: Sample Assembly Translation

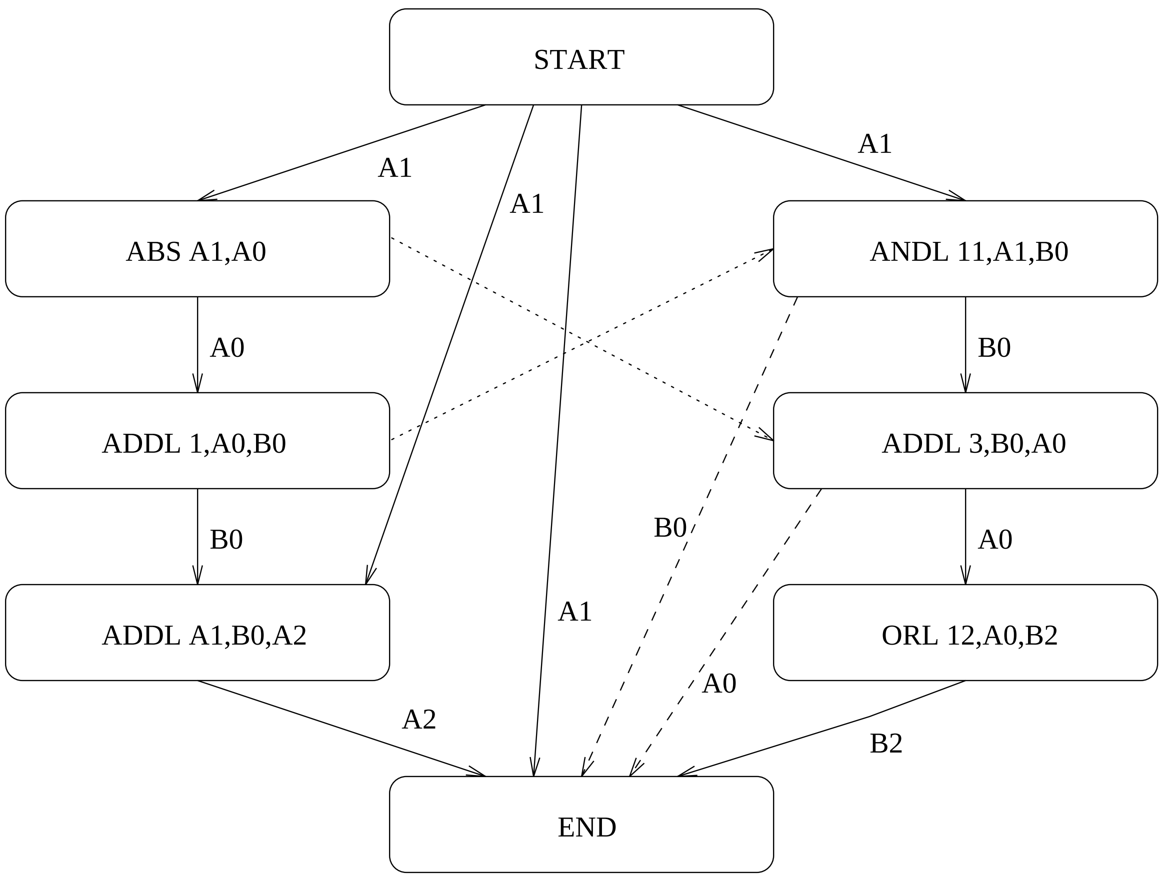

Figure 7-3: DDG Graph of Sample Assembly

```
ABS A1,A0;
ADDL 1,A0,B0;
ADDL A1,B0,A2 || ANDL 11,A1,B0;
ADDL 3,B0,A0;
ORL 12,A0,B2;
```

Figure 7-4: Solution of Original DDG

```
ABS A1,A0 || ANDL 11,A1,B0;
ADDL 1,A0,B0 || ADDL 3,B0,A0;
ADDL A1,B0,A2 || ORL 12,A0,B2;
```

Figure 7-5: Solution of Modified DDG

not have a resource associated with them - they are lines generated by the Live Range Analyzer (see Section 5.4), to capture implicit dependences. In particular, because of the dashed lines, indicating that the final values of A0 and B0 must come from the instructions on the right side of the graph (output dependencies), they have to execute after their left side counterparts modifying the same resources. Given this DDG the optimal solution found by the tools takes 5 cycles and is given on Figure 7-4. If the C compiler had specified that the values of A0 and B0 are unimportant at the end of the block, then we can have 2 operations per cycle for 3 cycles, as shown on Figure 7-5.

Let us step through some of the steps of obtaining the shorter solution. After *Start* (which is treated quite like an ordinary instruction) is asserted, the instructions with all their pre-requisites scheduled are *ABS A1,A0* and *ANDL 11,A1,B0*. Since the first executes on unit *L1* and the second on *L2*, they can execute together and therefore there are 3 possible combinations - each one separately or both of them together. The combination with both of them is heuristically prioritized and is selected.

On the next level *ADDL 1,A0,B0* and *ADDL 3,B0,A0* are available. They belong to different units and can be combined, so one might think that again here we have

3 possible combinations - both or each one of them separately. Strangely enough, however, the only option at this level is executing them together. Understanding why this is the only option in order to preserve correctness, leads us to the Dynamic Constraint Analyzer.

7.2 Dynamic Constraints

It turns out that in addition to the constraints on instruction combinations imposed by the chip architecture and the constraints associated with non-modification of a given resource more than once in a given cycle, there are constraints imposed by previously scheduled instructions. They will be referred to as dynamic constraints and further divided into two types:

1. *Hard Constraints* - constraints that unconditionally prevent instructions from being scheduled or unconditionally require that they must be scheduled on the current cycle

2. *Soft Constraints* - constraints indicating that certain instructions can execute only if other instructions execute on the same cycle

In the example above, scheduling *ABS A1,A0* means that no instructions modifying *A0* can be scheduled before scheduling *ADDL 1,A0,B0* and thus *ADDL 3,B0,A0* must occur no earlier than *ADDL 1,A0,B0*. Similarly, scheduling *ANDL 11,A1,B0* means that no instructions modifying *B0* can be scheduled before scheduling *ADDL 3,B0,A0* and thus *ADDL 1,A0,B0* must occur no earlier than *ADDL 3,B0,A0*. Those are examples of *Soft Constraints* leading to the conclusion that *ADDL 1,A0,B0* and *ADDL 3,B0,A0* must execute together or not at all to preserve correctness.

Note that the last statement is only true in the case where *ABS A1,A0* and *ANDL 11,A1,B0* are scheduled and *ADDL 1,A0,B0* and *ADDL 3,B0,A0* are not, and not in general - the original assembly code on Figure 7-2 is a trivial example where this situation does not occur.

The mere existence of such constraints, however, has major implications.

7.2.1 Dynamic Constraint Analysis is Necessary

As seen in the case above and in many other cases, analysis of those dynamic constraints is necessary to guarantee correctness, even though it is somewhat involved, especially in the presence of delay slots.

Furthermore, not only is analysis necessary for correctness, but with little augmentation it can be very useful computationally in eliminating areas of the search space that cannot produce a solution.

Consider the example above as part of some much larger basic block and a chip that cannot execute those two particular instructions together. By realizing that no solution can be found given the initial scheduling of instructions right away, the potentially huge computation of all possible ways to schedule the remaining instructions can be spared.

On a final note, such analysis is practical because such cases arise fairly often in practice. Any sequences of straight-line code without dependencies using temporary registers give rise to such situations.

7.2.2 Greedy Might not Guarantee Any Solution

Not only the dynamic constraints analysis is necessary, but the fact that there are cases where past choices might block progress means that if DDG is used as a starting phase, greedy picking of instructions might not find solution. In fact, even with backtracking any feasible solution might be problematic. In particular, if the input code is written for a chip with different capabilities and there and the guidelines from it cannot be used, there is no proof that any solution will be found or impossibility of solving the problem will be proven within polynomial time. While in theory this is disheartening, in practice, most problems in reasonable time using heuristics and user hints.

7.3 Types of Constraints

Having described the need for analysis, in this section an overview of the constraints handled will be presented and implementation details will be discussed later.

As disscussed in Chapter 6, in the beginning of the process of generating allowable combinations for the current cycle list of links to "unready" (LUL) and list of "ready" (RL) instructions are established and based on constrains in LUL some of the elements of RL are removed. Since most constraints apply to situations of use of the same register, throughout all the figures in this section, assumption of the same register on the links will be used. On those figures boxes will represent instructions and links without a starting box would represent links whose start has been scheduled, while boxes without links pointing to them will generally be assumed to be in RL. The links will be characterized by their RPL and DS properties as discussed in Section 6.3.1. Subscripts of one and two will be used to refer to the properties of the links on the left and on the right side of the Figure.

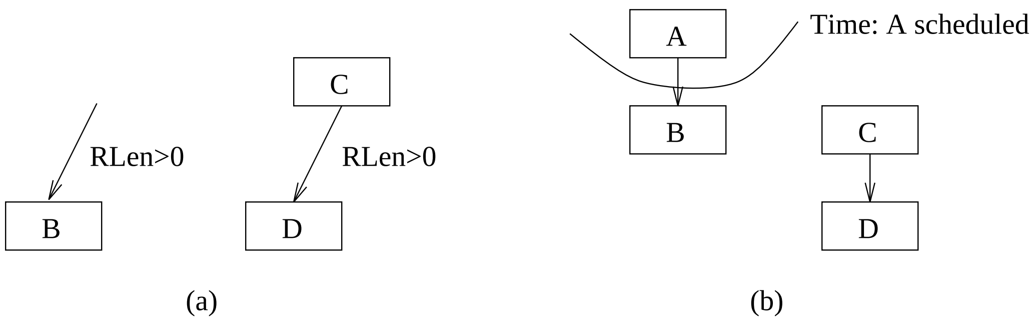

Figure 7-6: RPL-RPL Constraints

Consider the situation on Figure 7-6(a). This situation might occur as we are trying to schedule something like Figure 7-6(b). Basically, in this case we cannot schedule C, before B, but we might be able to schedule them together in the special case where $RPL_1 = 1$ and B is executable (or in the RL list). If $RPL_1 > 1$, this means that B modifies the resource and since the resource cannot be modified more than once C cannot execute together with it. Furthermore, if B is not executable (and the link is in LUL) then C cannot be executed (and should be removed from RL).

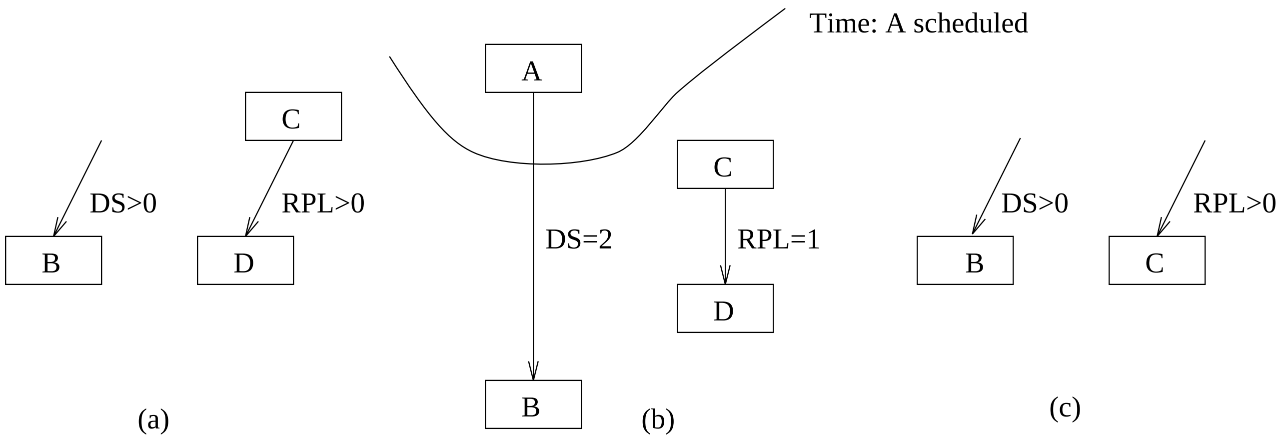

Figure 7-7: DS-RPL Constraints

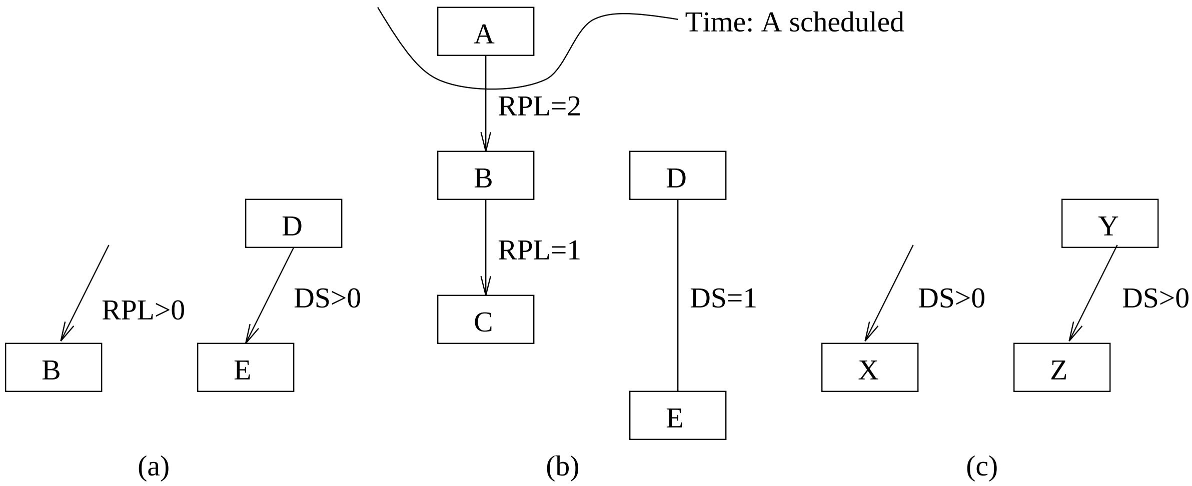

Figure 7-8: RPL-DS and DS-DS Constraints

The situation gets more complicated with the introduction of delay slots. Consider Figure 7-7(a). For C to be executable the delay slots have to be enough for all instructions using it to complete. Since C itself must be added what this means is that DS_1 must be greater than RPL_2 for this to happen. In addition, if $DS_1 = RPL_2 + 1$ and there was a link with the same resource to C then C must execute. This situation is captured on Figure 7-7(c), where RPL_2 is one larger than RPL_2 of of Figure 7-7(a), and if $DS_1 = RPL_2$ then C must execute on the current cycle.

The symmetric case is presented on Figure 7-8(a). There, again, the delay slots have to be long enough in order to allow the completion of the uses of B. Those requirements are strengthened if B is not executable (the link is in LUL). There are 3 cases:

- B is executable and $RPL_1 = DS_2 + 1$. Then, D can be executed only if B is executed (*Soft Constraint*).

- B is executable and $RPL_1 > DS_2$. Then D cannot be executed.

- B is not executable and $RPL_1 >= DS_2$. Then D cannot be executed.

The last case handled, is presented on Figure 7-8(c). There, if $DS_1 = DS_2 + 1$ then Y is not executable because two instructions cannot modify the same resource in the same cycle. More complicated analysis is possible at this level (ie. by looking at the RPL chains after that), but it is not that useful to perform. One could also make a point that more analysis could have been performed for all other cases, but given the existence of dependences on other resources and other search-limiting and optimization-guaranteeing mechanisms in the optimization framework, such efforts are most likely not practical.

7.4 Algorithm

Here is a brief description of the algorithm:

1. RL is initialized to all instructions that have no incoming links with unscheduled starts or positive delay slots.

2. LUL is initialized to all links from scheduled to non-scheduled instructions not in RL.

3. RL and LUL are examined for *Hard Constraints* and members of RL are removed and all their links are put in the LUL, potentially triggering additional removals.

4. If must-execute instructions are not in RL, backtracking is triggered.

5. If RL is empty, but there are links with positive delay slots in LUL, then the list of allowable combinations contains just *NOP*, the empty operation.

6. A graph of all *Soft Constraints* is generated among the elements of RL. The edges in the graph represent must-execute-no-later-than relation.

7. For instructions that must execute on the current cycle, links to all others are added, indicating that any other instructions can execute only if they are executed.

8. Topological sorting ([13]), based on that relation is performed by removing elements having no one earlier-or-together to them and their links in order. Maximal cycles (everyone-to-everyone directed connectivity) are identified, marked as instructions that must either execute together or not at all and considered a single instruction for the purposes of the topological sorting.

9. If the chip cannot do a set of instructions that must execute together, backtracking is triggered.

10. All combinations are generated making sure that instructions that must execute together execute together and that all pre-requisites are fulfilled. This list is guaranteed not to be empty.

Chapter 8

Linear Programming Bound Generator

8.1 Expressing The Problem As Linear Program

In a constrained search framework, it is critical to be able to find a good bound on the time to completion and go back as early as possible. One general approach that can be used for obtaining such lower bounds is integer linear programming. In particular, one can define $x_{i,c} \geq 0$ for all instruction sets i present in the current problem and all combinations c that they can be a part of. Then one can require that $\forall i$ $((\Sigma_c x_{i,c}) \geq X_i)$, where X_i is the number of instructions in set i in the current problem, and that $\forall c$, $\forall i$ $(x_{i,c} \leq Y_c)$, where Y_c is the number of combinations of type c used in the solution. Then by minimizing $\Sigma_c Y_c$ one will obtain a lower bound on the total cycles required for a solution. One can also minimize or maximize on any of $x_{i,c}$ or Y_c individually to obtain even more bounds.

8.2 Limitations Of Linear Programs

The linear program as defined above does not capture data dependencies or delay slots and thus is often too low and inacurate to be useful. In addition, the problems can be large and complicated for some instruction sets and integer solution might be hard to obtain within reasonable time. TMS320C6201 has very streamlined architecture and is not one of those cases, but even there solving the linear program in the integer domain (more accurate, higher bounds) takes more time than in the real domain and is not always justified.

While it might be possible to define the entire problem as a linear program as past work suggests, the number of variables and constraints will be overwhelming and there will be no gain from it, because that will be solving the same problem with a more general approach and less specific knowledge, which is counterproductive. Still in many cases, running real domain linear programming just for checking how far off from optimality the solution is, is useful.

8.3 Static Bounds and User-Defined Heuristic Bounds

In cases where linear bounds do not fare too well, such as sequences with large delay slots, an useful metric is minimum distance to end. This is computed statically in the beginning along all paths from a given instructions to the end and can be modified using user directives to help achieve faster unproductive search cut-offs. The more restrictive of those static bounds and the bound obtained from linear programming for all unscheduled instructions is returned.

Chapter 9

Optimizations, Heuristics and Results

Heuristics and good handling of special cases can make a significant impact on the running time of the tools.

9.1 Better or Faster Bounds

Better bounds can be achieved either deterministically by using more analysis on any particular chip architecture or can be heuristically set by the user, in which case the proof power of the extensive search is lost. One of the main advantages of such bounds is that linear programming is often fairly slow.

One example of better and faster bounds for TMS320C6201 is that since there is only one instruction set combination, where are 8 instructions sets (corresponding to instructions executable on each computational unit) can be combined, one could conclude that the linear programming code will just return the maximum of remaining instructions over all units. By replacing the call to the linear programming code, the speed of the code, especially in hard cases is more than doubled. Similar analysis can be performed in more complicated situations. Furthermore, if the bounds are computed so directly they could be augmented by adding the minimum (over all remaining instructions in a instruction set) delay slots to the total. For example - if there are 10 different memory loads that cannot be combined with each other and each one has 4 delay slots, then they cannot be completed in less that 14 cycles, which can be significant improvement over the LP generated bound. While TMS320C6201 is fairly simple and those bounds are sufficient, other bounds trying to capture more of the data dependencies can be devised, if necessary.

9.2 Prioritization of Combinations to Try

Another important place for heuristic choice is choosing how to prioritize instruction combinations to be attempted. With a simple chip architecture and easily computable good bounds as in the case of TMS320C6201, and with small basic blocks (which is often the case), those are not so important. However, with more complicated chips where means of combination are limited by instruction encoding space a good (or optimal) solution might be hard to obtain when using the wrong heuristic. The problem is complicated by the Dynamic Constraints phenomena, where doing more instructions earlier (which is good) might lead to an impossibility to construct a solution later. Heuristics that address such problems, but potentially need longer to converge to an optimal solution are choosing to do instructions with less outgoing links, instructions in the approximately same order as in the input file or instruction combinations with less instructions. A more flexible solution changing between some of those strategies depending on the current state would generally perform better, but again, those are not are not very commonly necessary. For the TMS320C6201, the best performing heuristic seems to be picking the combination with the instruction that has the longest path to completion, if equal, picking the combination with more instructions, and if equal, finally picking the one with instructions earlier in the input file.

9.3 Results

Given that heuristic, tests were run on all assembly files or C files compiled to assembly coming with the TMS320C6201 optimizing tools (37 files). The segments that cannot be optimized within this framework as defined in Section 5.2 were excluded. Note that while the excluded sections probably take a significant portion of the time and any performance evaluations are skewed, this is still code taken from an user manual ([27]), and thus is an characteristic of typical signal processing routines, and most likely tailored to exhibit the best performance of the TMS320C6201 software tools.

9.3.1 Better than Native Compiler

The code achieved solutions of the same length as the ones provided by the Texas Instruments tools for all 471 basic blocks, performing exhaustive search and confirming their optimality, in several minutes. It turned out that even the timing was comparable. Even though that there are examples where the Texas Instrument tools do not produce the optimal code, generated by this software, it turned out that in practice their output is almost always optimal, probably because of some hidden invariants in C-to-assembly conversion.

Here it is important to point out that the TMS320C6201 is a simple chip, not taking advantage of the full capabilities of this work, and that a specialized solution is always more likely to perform better on average (in this case slightly faster), or have more features than a general one.

9.3.2 Two Examples of Simple Retargeting

The main feature of this product is its retargetability as illustrated by the following examples, achievable by simple changes of several lines in the chip description file (see Appendix A):

- Suppose that we had a chip with the same assembly as TMS320C6201, where multiply instructions were executed by the L units instead, and there were no

M units. By making the changes and re-running in 5 minutes, all the 471 basic blocks were converted to the new chip architecture, and for 460 of them the solutions were provably optimal, within reordering constraints. Furthermore, interestingly enough, only 13 blocks exhibited increase in running time, by an average of about 30 per cent, while all others kept their original time. Those results suggest, for example, that it might have been better to eliminate the M units and integrate them more closely with the L units to save chip space. This kind of experiments and reasoning are very characteristic of the way this tool can be used.

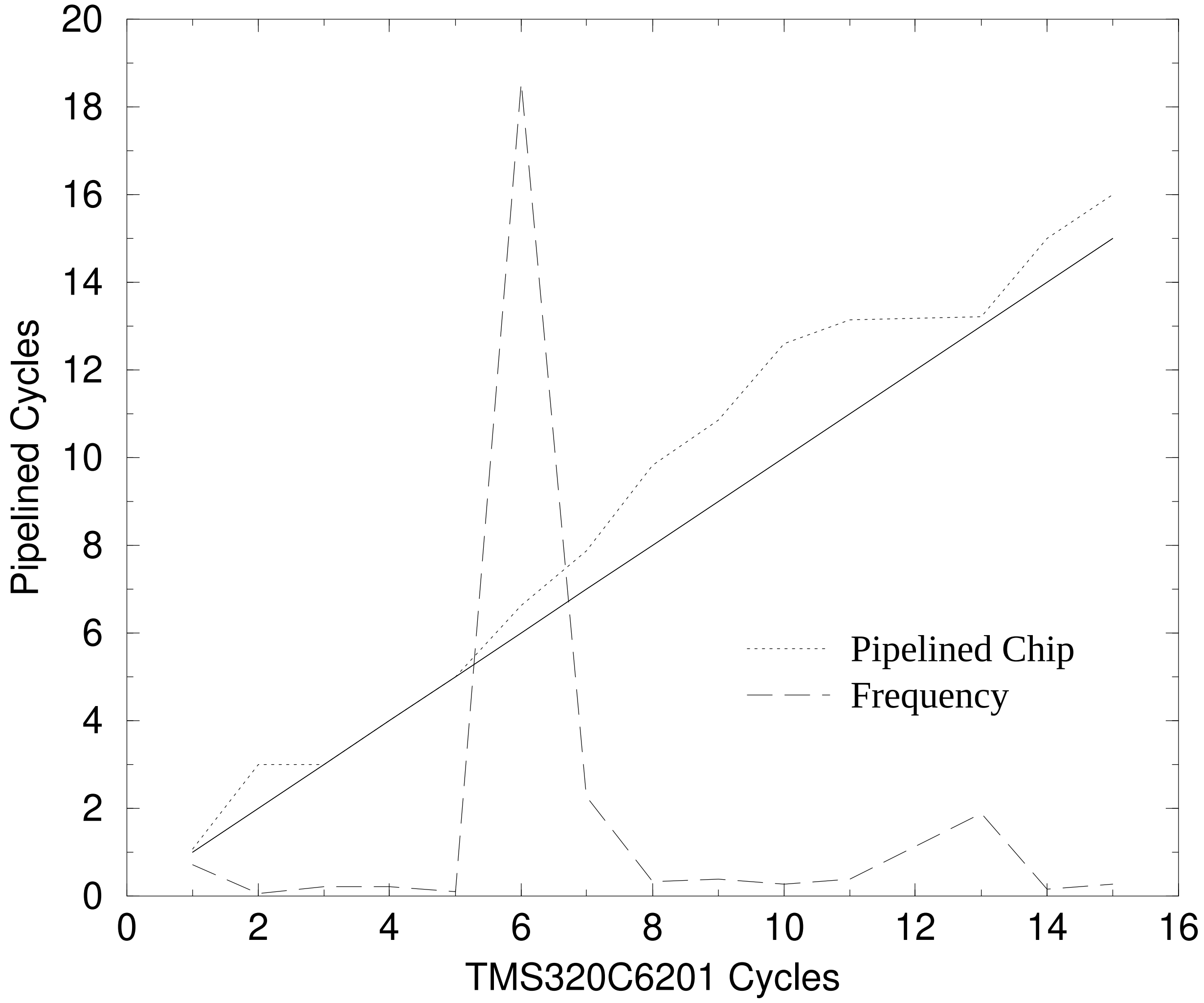

Figure 9-1: Performance Comparison Example

- In another example, suppose that we only had 4 computational units and the chip was pipelined in such a way that on a single cycle it could only issue instructions utilizing a single half of the original chip - eg. either ones using M1, S1, L1 and D1 or ones using M2, S2, L2 and D2 (refer to Figure 4-1). Again, by simple re-running, the code was converted to the new architecture. In this case 323 blocks kept their original running time, while 142 exhibited increases of about 30 per cent on average. For 6 blocks no solutions were found, because they could not be executed without modification of code (see Chapter 7 for discussion of such cases). The vast majority of solutions came with optimality proof as well. Experiments like this can be useful, since they are suggestive of the load of the system - in particular, on full load one could expect a factor of 2 slowdown overall. Figure 9-1 presents the actual average increase in running time, by basic block cycles, together with their frequency. The X-axis is the number of optimized cycles on TMS320C6201, the deshed line represents the relative frequency of such blocks and the dotted line represents the average increase in size over all blocks. Clearly the increase is not as significant as one might have expected and this would have been hard to achieve without this assembly optimizer.

9.3.3 Sample Code Fragment

Figure 9-2 presents a fairly representative code fragment, resulting from the Texas Instruments optimizing C compiler. The first fragment is the output of the Texas Instruments compiler. As can be seen they do not attempt too much packing and the techniques described herein and that is why their results are not always optimal. The middle fragment is the code produced by the assembly optimizer described in this work for the original chip and for the chip without M units. It takes the same amount of time (largely determined by the delay slots of the memory load and the branch), but the effects of the heuristic attempting denser packing in the beginning can be seen. The last fragment presents the code on the chip where instructions can execute on a single side of the chip at a time. Note that those solutions are guaranteed optimal

```
ADDAH .D A0,A3,A0 || MVK .S 512,B7;
MV .L A0,B5 || EXT .S B4,16,16,B6;
LDH .D *B5,B5 || CMPLT .L B6,B7,B0;
[!B0] B .S L21;
NOP;
NOP;
NOP;
SUB .L B5,1,B5 || MV .L B4,A0 || MV .S A0,B8;
STH .D B5,*B8 || EXT .S A0,16,16,A0;
;; --- TI Original Solution ------------------------------------


ADDAH .D A0,A3,A0 || MVK .S 512,B7;
MV .L A0,B5 || EXT .S B4,16,16,B6;
LDH .D *B5,B5 || CMPLT .L B6,B7,B0 || MV .L B4,A0 || MV .S A0,B8;
[!B0] B .S L21 || EXT .S A0,16,16,A0;
NOP;
NOP;
NOP;
SUB .L B5,1,B5;
STH .D B5,*B8;
;; --- Optimizer Solution for TMS320C6201 and No M Unit ---------


ADDAH .D A0,A3,A0;
MVK .S 512,B7 || MV .L A0,B5;
EXT .S B4,16,16,B6 || LDH .D *B5,B5;
CMPLT .L B6,B7,B0 || MV .S A0,B8;
[!B0] B .S L21;
MV .L B4,A0;
EXT .S A0,16,16,A0;
SUB .L B5,1,B5;
STH .D B5,*B8;
NOP;
;; --- Optimizer Solution for M1 L1 S1 D1 or M2 S2 L2 D2 --------
```

Figure 9-2: Sample Assembly Block for Different Chips by the exhaustive search, which by itself is not a trivial task, say to a human. It is easy to demonstrate how compiler technology will clearly outperform hand-crafted optimizations at this level for more complicated examples - all of those solutions are found in several seconds, more than it would take a programmer to read the problem.

Chapter 10

Extensions and Applications

Having given an overview of what the tools can do, it is time to address possible extensions of this work.

10.1 Extensions

- Although it is fairly self-sufficient, making the code more robust and extending it to handle a wider variety of chips or better handle special cases might be done (as discussed in Section 9.1).
- The work can also be extended to unroll loops to achieve better results in pipelined chips, similarly to optimizations performed by the TMS320C6201 tools ([26, 27]).
- Relaxing some of the memory restrictions would also be necessary and providing more flexible support and more memory access modes is also necessary.
- Another expansion path is to relax the non-modification of code. A step in that direction would be allowing description using virtual registers and optimizing their placement in appropriate register files together with optimizing the output. This should be done carefully, in order not to lose the manageability of most problems present when using the current version.

- An interesting idea for possible extension is providing feedback to upper levels of compilation for bottlenecks and proposed solutions to them, while remaining an independent tool focused on optimizations.
- One could also use neural nets, memoization and similar techniques for obtaining solutions and converging to better solution faster on hard problems.
- Blocks can be allowed to "float" within other blocks provided that there are no conflicting dependencies. Loops can be partially unrolled, as well. Both optimizations hope to combine instructions better than done at the individual block level.

10.2 Applications

The assembly optimizer at its current stage or together with some of the above extensions can be used in variety of different ways. They can also be supplemented by additional tools such as profilers and statistics collection and visualization tools.

10.2.1 Evaluation Tool for New Chip Designs

As shown in Section 9.3.2, the optimizer can be used to evaluate new chips designs even as presented here. However, some simple extensions, such as allowing reading file in one chip assembly and optimizing it and outputting it for another chip assembly, could make the process much smoother. With more support for different memory, interrupts or chip architectures, the differences between input and output chip assembly can go wider. With profiling capabilities and user-defined hints on performance the optimizer can be a valuable tool in evaluation of new chip designs, by providing good estimate on how long optimized code will run on the new chip.

10.2.2 Porting Existing Code

Not only can the optimizer provide evaluation data, but also, as soon as it goes through the input files, we have a new version of the code. What this means is that if there is correspondence between instruction sets, one could take a chip running certain code, disassembly the code and recompile it for another chip. Note that no assembly or higher level language source code is necessary. Note also that the new chip can be more powerful, less powerful or just different and this approach might still work for a large portion of the code. This portability feature is a very useful one to have, indeed.

10.2.3 Last Phase of Optimizing Compiler

Since no high level source is necessary, this tool can be used as the last stage, or even after all other stages of an optimizing compiler. Indeed, one can let the compiler perform all its optimizations and generate assembly code (or disassembly the code it produced), and run this assembly optimizer.

As a nice corollary, it might turn out that one could get by, without even developing any compilers for a chip architecture in some cases (of course performance will be worse than with native compiler, but maybe not that much). For example since Texas Instruments has good optimizing C compiler and linear assembly optimizer, then one might use those tools to generate parallel assembly instructions (that might be fairly generic). Capabilities like these are also very well worth the development effort.

Chapter 11

Conclusion

Compilation for powerful chips is a difficult problem and there are many stages and levels of optimization. Clearly, the task of optimal compilation from a high level language like C, is impossible to accomplish for all but very simple C programs and chip architectures. Each step of the process is a hard problem and is commonly solved using heuristic solutions. Rather than focusing on solving the problem at a global level, in this work a level was chosen and other design choices were made that would make the optimization problem tractable, while still being useful and maintaining retargetability. Practical comparisons showed that the Assembly Description Language devised was capable of capturing the architecture of fairly complex chips and that the optimization framework was capable of providing same or better solutions than commercial applications fairly close to their real-time frame. All that, together with the simplicity of use and the modularity of the implementation, make this work a good base for extensions and commercial applications, especially in embedded systems where processor architectures change very frequently.

Appendix A

TMS320C6201 Chip Description

RSet AX A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 SP DP CSR ;
RSet BX B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15 SP DP CSR ;
RSet AL A1:A0 A3:A2 A5:A4 A7:A6 A9:A8 A11:A10 A13:A12 A15:A14 ;
RSet BL B1:B0 B3:B2 B5:B4 B7:B6 B9:B8 B11:B10 B13:B12 B15:B14 ;
RSet ALX A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 SP DP CSR A1:A0 A3:A2 A5:A4 A7:A6 A9:A8 A11:A10 A13:A12 A15:A14 ;
RSet BLX B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15 SP DP CSR B1:B0 B3:B2 B5:B4 B7:B6 B9:B8 B11:B10 B13:B12 B15:B14 ;
RSet ABLX A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 A1:A0 A3:A2 A5:A4 A7:A6 A9:A8 A11:A10 A13:A12 A15:A14 B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15 B1:B0 B3:B2 B5:B4 B7:B6 B9:B8 B11:B10 B13:B12 B15:B14 SP DP CSR ;
RSet ABX A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15 SP DP CSR ;

OSet Nmb STRING -_STRING STRING_+_STRING ;
OSet MOD _ [_STRING_] ;
OSet LInst1 ABS NEG NORM MV NOT SAT ;
OSet LInst2 ADD ADDU AND CMPEQ CMPGT CMPGT CMPGTU CMPLTU CMPLT OR SADD SSUB SUB SUBU XOR ;
OSet SInst ADD ADDU SUB SUBU ADD2 SUB2 AND CLR EXT NEG OR SHR SHRU SHL XOR ;
OSet DInst ADD ADDAB ADDAH ADDAW SUB SUBA SUBAH SUBAW ;
OSet BBX B IRP NRP ;
OSet ECLR EXT EXTU CLR SET ;
OSet LDX LDW LDB LDH LDBU LDHU ;
OSet LDXL LDB LDH LDBU LDHU ;
OSet STX STW STH STBU STHU ;
OSet STXL STB STH STBU STHU ;
OSet +- + - ;
OSet ++− ++ − ;
OSet MPYX MPY MPYU MPYSU MPYUS MPYH MPYHU MPYHUS MPYHSU MPYLH MPYLHU MPYLUHS MPYLSHU MPYHL ;
OSet MVKX ADDK MVK MVKH MVKLH ;

```
OSet UniS MV NOT NEG MVC ;

inst ABS1A LInst1 .L AX , AX ; 3 ; 5 ; ;
inst ABS1B [ ABX ] LInst1 .L AX , AX ; 2 6 ; 8 ; 13 ;
inst ABS1C [ ! ABX ] LInst1 .L AX , AX ; 3 7 ; 9 ; 14 ;
inst ABS2A LInst1 .L BX , BX ; 3 ; 5 ; ;
inst ABS2B [ ABX ] LInst1 .L BX , BX ; 2 6 ; 8 ; 13 ;
inst ABS2C [ ! ABX ] LInst1 .L BX , BX ; 3 7 ; 9 ; 14 ;
inst ABS1XA LInst1 .L BX , AX ; 3 ; 5 x1 ; ;
inst ABS1XB [ ABX ] LInst1 .L BX , AX ; 2 6 ; 8 x1 ; 13 ;
inst ABS1XC [ ! ABX ] LInst1 .L BX , AX ; 3 7 ; 9 x1 ; 14 ;
inst ABS2XA LInst1 .L AX , BX ; 3 ; 5 x2 ; ;
inst ABS2XB [ ABX ] LInst1 .L AX , BX ; 2 6 ; 8 x2 ; 13 ;
inst ABS2XC [ ! ABX ] LInst1 .L AX , BX ; 3 7 ; 9 x2 ; 14 ;
inst INSTL1AAA LInst2 .L ALX , ALX , ALX ; 3 5 ; 7 ; ;
inst INSTL1AAB [ ABX ] LInst2 .L ALX , ALX , ALX ; 2 6 8 ; 10 ; 13 ;
inst INSTL1AAC [ ! ABX ] LInst2 .L ALX , ALX , ALX ; 3 7 9 ; 11 ; 14 ;
inst INSTL2AAA LInst2 .L Nmb , ALX , ALX ; 5 ; 7 ; ;
inst INSTL2AAB [ ABX ] LInst2 .L Nmb , ALX , ALX ; 2 8 ; 10 ; 13 ;
inst INSTL2AAC [ ! ABX ] LInst2 .L Nmb , ALX , ALX ; 3 9 ; 11 ; 14 ;
inst INSTL3AAA LInst2 .L ALX , Nmb , ALX ; 3 ; 7 ; ;
inst INSTL3AAB [ ABX ] LInst2 .L ALX , Nmb , ALX ; 2 6 ; 10 ; 13 ;
inst INSTL3AAC [ ! ABX ] LInst2 .L ALX , Nmb , ALX ; 3 7 ; 11 ; 14 ;
inst INSTL1BAA LInst2 .L ALX , BLX , ALX ; 3 5 ; 7 x1 ; ;
inst INSTL1BAB [ ABX ] LInst2 .L ALX , BLX , ALX ; 2 6 8 ; 10 x1 ; 13 ;
inst INSTL1BAC [ ! ABX ] LInst2 .L ALX , BLX , ALX ; 3 7 9 ; 11 x1 ; 14 ;
inst INSTL2BAA LInst2 .L Nmb , BLX , ALX ; 5 ; 7 x1 ; ;
inst INSTL2BAB [ ABX ] LInst2 .L Nmb , BLX , ALX ; 2 8 ; 10 x1 ; 13 ;
inst INSTL2BAC [ ! ABX ] LInst2 .L Nmb , BLX , ALX ; 3 9 ; 11 x1 ; 14 ;
inst INSTL3BAA LInst2 .L BLX , Nmb , ALX ; 3 ; 7 x1 ; ;
inst INSTL3BAB [ ABX ] LInst2 .L BLX , Nmb , ALX ; 2 6 ; 10 x1 ; 13 ;
inst INSTL3BAC [ ! ABX ] LInst2 .L BLX , Nmb , ALX ; 3 7 ; 11 x1 ; 14 ;
inst INSTL4BAA LInst2 .L BLX , ALX , ALX ; 3 5 ; 7 x1 ; ;
inst INSTL4BAB [ ABX ] LInst2 .L BLX , ALX , ALX ; 2 6 8 ; 10 x1 ; 13 ;
inst INSTL4BAC [ ! ABX ] LInst2 .L BLX , ALX , ALX ; 3 7 9 ; 11 x1 ; 14 ;
inst INSTL1BBA LInst2 .L BLX , BLX , BLX ; 3 5 ; 7 ; ;
inst INSTL1BBB [ ABX ] LInst2 .L BLX , BLX , BLX ; 2 6 8 ; 10 ; 13 ;
inst INSTL1BBC [ ! ABX ] LInst2 .L BLX , BLX , BLX ; 3 7 9 ; 11 ; 14 ;
inst INSTL2BBA LInst2 .L Nmb , BLX , BLX ; 5 ; 7 ; ;
inst INSTL2BBB [ ABX ] LInst2 .L Nmb , BLX , BLX ; 2 8 ; 10 ; 13 ;
inst INSTL2BBC [ ! ABX ] LInst2 .L Nmb , BLX , BLX ; 3 9 ; 11 ; 14 ;
inst INSTL3BBA LInst2 .L BLX , Nmb , BLX ; 3 ; 7 ; ;
inst INSTL3BBB [ ABX ] LInst2 .L BLX , Nmb , BLX ; 2 6 ; 10 ; 13 ;
inst INSTL3BBC [ ! ABX ] LInst2 .L BLX , Nmb , BLX ; 3 7 ; 11 ; 14 ;
inst INSTL1ABA LInst2 .L BLX , ALX , BLX ; 3 5 ; 7 x2 ; ;
inst INSTL1ABB [ ABX ] LInst2 .L BLX , ALX , BLX ; 2 6 8 ; 10 x2 ; 13 ;
inst INSTL1ABC [ ! ABX ] LInst2 .L BLX , ALX , BLX ; 3 7 9 ; 11 x2 ; 14 ;
```

inst INSTL2ABA LInst2 .L Nmb , ALX , BLX ; 5 ; 7 x2 ; ;
inst INSTL2ABB [ABX] LInst2 .L Nmb , ALX , BLX ; 2 8 ; 10 x2 ; 13 ;
inst INSTL2ABC [! ABX] LInst2 .L Nmb , ALX , BLX ; 3 9 ; 11 x2 ; 14 ;
inst INSTL3ABA LInst2 .L ALX , Nmb , BLX ; 3 ; 7 x2 ; ;
inst INSTL3ABB [ABX] LInst2 .L ALX , Nmb , BLX ; 2 6 ; 10 x2 ; 13 ;
inst INSTL3ABC [! ABX] LInst2 .L ALX , Nmb , BLX ; 3 7 ; 11 x2 ; 14 ;
inst INSTL4ABA LInst2 .L ALX , BLX , BLX ; 3 5 ; 7 x2 ; ;
inst INSTL4ABB [ABX] LInst2 .L ALX , BLX , BLX ; 2 6 8 ; 10 x2 ; 13 ;
inst INSTL4ABC [! ABX] LInst2 .L ALX , BLX , BLX ; 3 7 9 ; 11 x2 ; 14 ;
inst LZERO1A ZERO .L ALX ; ; 3 ; ;
inst LZERO1B [ABX] ZERO .L ALX ; 2 ; 6 ; 13 ;
inst LZERO1C [! ABX] ZERO .L ALX ; 3 ; 7 ; 14 ;
inst LZERO2A ZERO .L BLX ; ; 3 ; ;
inst LZERO2B [ABX] ZERO .L BLX ; 2 ; 6 ; 13 ;
inst LZERO2C [! ABX] ZERO .L BLX ; 3 ; 7 ; 14 ;

inst INSTS1AAA SInst .S ALX , ALX , ALX ; 3 5 ; 7 ; ;
inst INSTS1AAB [ABX] SInst .S ALX , ALX , ALX ; 2 6 8 ; 10 ; 13 ;
inst INSTS1AAC [! ABX] SInst .S ALX , ALX , ALX ; 3 7 9 ; 11 ; 14 ;
inst INSTS2AAA SInst .S Nmb , ALX , ALX ; 5 ; 7 ; ;
inst INSTS2AAB [ABX] SInst .S Nmb , ALX , ALX ; 2 8 ; 10 ; 13 ;
inst INSTS2AAC [! ABX] SInst .S Nmb , ALX , ALX ; 3 9 ; 11 ; 14 ;
inst INSTS3AAA SInst .S ALX , Nmb , ALX ; 3 ; 7 ; ;
inst INSTS3AAB [ABX] SInst .S ALX , Nmb , ALX ; 2 6 ; 10 ; 13 ;
inst INSTS3AAC [! ABX] SInst .S ALX , Nmb , ALX ; 3 7 ; 11 ; 14 ;
inst INSTS1BAA SInst .S ALX , BLX , ALX ; 3 5 ; 7 x1 ; ;
inst INSTS1BAB [ABX] SInst .S ALX , BLX , ALX ; 2 6 8 ; 10 x1 ; 13 ;
inst INSTS1BAC [! ABX] SInst .S ALX , BLX , ALX ; 3 7 9 ; 11 x1 ; 14 ;
inst INSTS2BAA SInst .S Nmb , BLX , ALX ; 5 ; 7 x1 ; ;
inst INSTS2BAB [ABX] SInst .S Nmb , BLX , ALX ; 2 8 ; 10 x1 ; 13 ;
inst INSTS2BAC [! ABX] SInst .S Nmb , BLX , ALX ; 3 9 ; 11 x1 ; 14 ;
inst INSTS3BAA SInst .S BLX , Nmb , ALX ; 3 ; 7 x1 ; ;
inst INSTS3BAB [ABX] SInst .S BLX , Nmb , ALX ; 2 6 ; 10 x1 ; 13 ;
inst INSTS3BAC [! ABX] SInst .S BLX , Nmb , ALX ; 3 7 ; 11 x1 ; 14 ;
inst INSTS4BAA SInst .S BLX , ALX , ALX ; 3 5 ; 7 x1 ; ;
inst INSTS4BAB [ABX] SInst .S BLX , ALX , ALX ; 2 6 8 ; 10 x1 ; 13 ;
inst INSTS4BAC [! ABX] SInst .S BLX , ALX , ALX ; 3 7 9 ; 11 x1 ; 14 ;
inst INSTS1BBA SInst .S BLX , BLX , BLX ; 3 5 ; 7 ; ;
inst INSTS1BBB [ABX] SInst .S BLX , BLX , BLX ; 2 6 8 ; 10 ; 13 ;
inst INSTS1BBC [! ABX] SInst .S BLX , BLX , BLX ; 3 7 9 ; 11 ; 14 ;
inst INSTS2BBA SInst .S Nmb , BLX , BLX ; 5 ; 7 ; ;
inst INSTS2BBB [ABX] SInst .S Nmb , BLX , BLX ; 2 8 ; 10 ; 13 ;
inst INSTS2BBC [! ABX] SInst .S Nmb , BLX , BLX ; 3 9 ; 11 ; 14 ;
inst INSTS3BBA SInst .S BLX , Nmb , BLX ; 3 ; 7 ; ;
inst INSTS3BBB [ABX] SInst .S BLX , Nmb , BLX ; 2 6 ; 10 ; 13 ;
inst INSTS3BBC [! ABX] SInst .S BLX , Nmb , BLX ; 3 7 ; 11 ; 14 ;
inst INSTS1ABA SInst .S BLX , ALX , BLX ; 3 5 ; 7 x2 ; ;

inst INSTS1ABB [ABX] SInst .S BLX , ALX , BLX ; 2 6 8 ; 10 x2 ; 13 ;
inst INSTS1ABC [! ABX] SInst .S BLX , ALX , BLX ; 3 7 9 ; 11 x2 ; 14 ;
inst INSTS2ABA SInst .S Nmb , ALX , BLX ; 5 ; 7 x2 ; ;
inst INSTS2ABB [ABX] SInst .S Nmb , ALX , BLX ; 2 8 ; 10 x2 ; 13 ;
inst INSTS2ABC [! ABX] SInst .S Nmb , ALX , BLX ; 3 9 ; 11 x2 ; 14 ;
inst INSTS3ABA SInst .S ALX , Nmb , BLX ; 3 ; 7 x2 ; ;
inst INSTS3ABB [ABX] SInst .S ALX , Nmb , BLX ; 2 6 ; 10 x2 ; 13 ;
inst INSTS3ABC [! ABX] SInst .S ALX , Nmb , BLX ; 3 7 ; 11 x2 ; 14 ;
inst INSTS4ABA SInst .S ALX , BLX , BLX ; 3 5 ; 7 x2 ; ;
inst INSTS4ABB [ABX] SInst .S ALX , BLX , BLX ; 2 6 8 ; 10 x2 ; 13 ;
inst INSTS4ABC [! ABX] SInst .S ALX , BLX , BLX ; 3 7 9 ; 11 x2 ; 14 ;
inst BRANCH1A B .S AX ; 3 ; ip #5 ; 10 ;
inst BRANCH1B [ABX] B .S AX ; 2 6 ; ip #5 ; 11 ;
inst BRANCH1C [! ABX] B .S AX ; 3 7 ; ip #5 ; 12 ;
inst BRANCH2A B .S BX ; 3 ; ip #5 ; 10 ;
inst BRANCH2B [ABX] B .S BX ; 2 6 ; ip #5 ; 11 ;
inst BRANCH2C [! ABX] B .S BX ; 3 7 ; ip #5 ; 12 ;
inst BRANCH3A BBX .S STRING ; ; ip #5 ; 10 ;
inst BRANCH3B [ABX] BBX .S STRING ; 2 ; ip #5 ; 11 ;
inst BRANCH3C [! ABX] BBX .S STRING ; 3 ; ip #5 ; 12 ;
inst BRANCHDA BD .S STRING ; ; ip #5 ; 10 ;
inst BRANCHDB [ABX] BD .S STRING ; 2 ; ip #5 ; 11 ;
inst BRANCHDC [! ABX] BD .S STRING ; 3 ; ip #5 ; 12 ;
inst ECLRAA ECLR .S AX , Nmb , Nmb , AX ; 3 ; 9 ; ;
inst ECLRAB [ABX] ECLR .S AX , Nmb , Nmb , AX ; 2 6 ; 12 ; 13 ;
inst ECLRAC [! ABX] ECLR .S AX , Nmb , Nmb , AX ; 3 7 ; 13 ; 14 ;
inst ECLRBA ECLR .S BX , Nmb , Nmb , BX ; 3 ; 9 ; ;
inst ECLRBB [ABX] ECLR .S BX , Nmb , Nmb , BX ; 2 6 ; 12 ; 13 ;
inst ECLRBC [! ABX] ECLR .S BX , Nmb , Nmb , BX ; 3 7 ; 13 ; 14 ;
inst ECLRAXA ECLR .S BX , Nmb , Nmb , AX ; 3 ; 9 x1 ; ;
inst ECLRAXB [ABX] ECLR .S BX , Nmb , Nmb , AX ; 2 6 ; 12 x1 ; 13 ;
inst ECLRAXC [! ABX] ECLR .S BX , Nmb , Nmb , AX ; 3 7 ; 13 x1 ; 14 ;
inst ECLRBXA ECLR .S AX , Nmb , Nmb , BX ; 3 ; 9 x2 ; ;
inst ECLRBXB [ABX] ECLR .S AX , Nmb , Nmb , BX ; 2 6 ; 12 x2 ; 13 ;
inst ECLRBXC [! ABX] ECLR .S AX , Nmb , Nmb , BX ; 3 7 ; 13 x2 ; 14 ;
inst MVKAA MVKX .S Nmb , AX ; 5 ; 5 ; ;
inst MVKAB [ABX] MVKX .S Nmb , AX ; 2 8 ; 8 ; 13 ;
inst MVKAC [! ABX] MVKX .S Nmb , AX ; 3 9 ; 9 ; 14 ;
inst MVKBA MVKX .S Nmb , BX ; 5 ; 5 ; ;
inst MVKBB [ABX] MVKX .S Nmb , BX ; 2 8 ; 8 ; 13 ;
inst MVKBC [! ABX] MVKX .S Nmb , BX ; 3 9 ; 9 ; 14 ;
inst UNI1A UniS .S AX , AX ; 3 ; 5 ; ;
inst UNI1B [ABX] UniS .S AX , AX ; 2 6 ; 8 ; 13 ;
inst UNI1C [! ABX] UniS .S AX , AX ; 3 7 ; 9 ; 14 ;
inst UNI2A UniS .S BX , BX ; 3 ; 5 ; ;
inst UNI2B [ABX] UniS .S BX , BX ; 2 6 ; 8 ; 13 ;
inst UNI2C [! ABX] UniS .S BX , BX ; 3 7 ; 9 ; 14 ;

```
inst UNI1XA UniS .S BX , AX ; 3 ; 5 x1 ; ;
inst UNI1XB [ ABX ] UniS .S BX , AX ; 2 6 ; 8 x1 ; 13 ;
inst UNI1XC [ ! ABX ] UniS .S BX , AX ; 3 7 ; 9 x1 ; 14 ;
inst UNI2XA UniS .S AX , BX ; 3 ; 5 x2 ; ;
inst UNI2XB [ ABX ] UniS .S AX , BX ; 2 6 ; 8 x2 ; 13 ;
inst UNI2XC [ ! ABX ] UniS .S AX , BX ; 3 7 ; 9 x2 ; 14 ;
inst SZERO1A ZERO .S AX ; ; 3 ; ;
inst SZERO1B [ ABX ] ZERO .S AX ; 2 ; 6 ; 13 ;
inst SZERO1C [ ! ABX ] ZERO .S AX ; 3 ; 7 ; 14 ;
inst SZERO2A ZERO .S BX ; ; 3 ; ;
inst SZERO2B [ ABX ] ZERO .S BX ; 2 ; 6 ; 13 ;
inst SZERO2C [ ! ABX ] ZERO .S BX ; 3 ; 7 ; 14 ;

inst DZERO1A ZERO .D AX ; ; 3 ; ;
inst DZERO1B [ ABX ] ZERO .D AX ; 2 ; 6 ; 13 ;
inst DZERO1C [ ! ABX ] ZERO .D AX ; 3 ; 7 ; 14 ;
inst DZERO2A ZERO .D BX ; ; 3 ; ;
inst DZERO2B [ ABX ] ZERO .D BX ; 2 ; 6 ; 13 ;
inst DZERO2C [ ! ABX ] ZERO .D BX ; 3 ; 7 ; 14 ;
inst INSTD1AAA DInst .D ALX , ALX , ALX ; 3 5 ; 7 ; ;
inst INSTD1AAB [ ABX ] DInst .D ALX , ALX , ALX ; 2 6 8 ; 10 ; 13 ;
inst INSTD1AAC [ ! ABX ] DInst .D ALX , ALX , ALX ; 3 7 9 ; 11 ; 14 ;
inst INSTD2AAA DInst .D Nmb , ALX , ALX ; 5 ; 7 ; ;
inst INSTD2AAB [ ABX ] DInst .D Nmb , ALX , ALX ; 2 8 ; 10 ; 13 ;
inst INSTD2AAC [ ! ABX ] DInst .D Nmb , ALX , ALX ; 3 9 ; 11 ; 14 ;
inst INSTD3AAA DInst .D ALX , Nmb , ALX ; 3 ; 7 ; ;
inst INSTD3AAB [ ABX ] DInst .D ALX , Nmb , ALX ; 2 6 ; 10 ; 13 ;
inst INSTD3AAC [ ! ABX ] DInst .D ALX , Nmb , ALX ; 3 7 ; 11 ; 14 ;
inst INSTD1BAA DInst .D ALX , BLX , ALX ; 3 5 ; 7 x1 ; ;
inst INSTD1BAB [ ABX ] DInst .D ALX , BLX , ALX ; 2 6 8 ; 10 x1 ; 13 ;
inst INSTD1BAC [ ! ABX ] DInst .D ALX , BLX , ALX ; 3 7 9 ; 11 x1 ; 14 ;
inst INSTD2BAA DInst .D Nmb , BLX , ALX ; 5 ; 7 x1 ; ;
inst INSTD2BAB [ ABX ] DInst .D Nmb , BLX , ALX ; 2 8 ; 10 x1 ; 13 ;
inst INSTD2BAC [ ! ABX ] DInst .D Nmb , BLX , ALX ; 3 9 ; 11 x1 ; 14 ;
inst INSTD3BAA DInst .D BLX , Nmb , ALX ; 3 ; 7 x1 ; ;
inst INSTD3BAB [ ABX ] DInst .D BLX , Nmb , ALX ; 2 6 ; 10 x1 ; 13 ;
inst INSTD3BAC [ ! ABX ] DInst .D BLX , Nmb , ALX ; 3 7 ; 11 x1 ; 14 ;
inst INSTD4BAA DInst .D BLX , ALX , ALX ; 3 5 ; 7 x1 ; ;
inst INSTD4BAB [ ABX ] DInst .D BLX , ALX , ALX ; 2 6 8 ; 10 x1 ; 13 ;
inst INSTD4BAC [ ! ABX ] DInst .D BLX , ALX , ALX ; 3 7 9 ; 11 x1 ; 14 ;
inst INSTD1BBA DInst .D BLX , BLX , BLX ; 3 5 ; 7 ; ;
inst INSTD1BBB [ ABX ] DInst .D BLX , BLX , BLX ; 2 6 8 ; 10 ; 13 ;
inst INSTD1BBC [ ! ABX ] DInst .D BLX , BLX , BLX ; 3 7 9 ; 11 ; 14 ;
inst INSTD2BBA DInst .D Nmb , BLX , BLX ; 5 ; 7 ; ;
inst INSTD2BBB [ ABX ] DInst .D Nmb , BLX , BLX ; 2 8 ; 10 ; 13 ;
inst INSTD2BBC [ ! ABX ] DInst .D Nmb , BLX , BLX ; 3 9 ; 11 ; 14 ;
inst INSTD3BBA DInst .D BLX , Nmb , BLX ; 3 ; 7 ; ;
```

inst INSTD3BBB [ABX] DInst .D BLX , Nmb , BLX ; 2 6 ; 10 ; 13 ;
inst INSTD3BBC [! ABX] DInst .D BLX , Nmb , BLX ; 3 7 ; 11 ; 14 ;
inst INSTD1ABA DInst .D BLX , ALX , BLX ; 3 5 ; 7 x2 ; ;
inst INSTD1ABB [ABX] DInst .D BLX , ALX , BLX ; 2 6 8 ; 10 x2 ; 13 ;
inst INSTD1ABC [! ABX] DInst .D BLX , ALX , BLX ; 3 7 9 ; 11 x2 ; 14 ;
inst INSTD2ABA DInst .D Nmb , ALX , BLX ; 5 ; 7 x2 ; ;
inst INSTD2ABB [ABX] DInst .D Nmb , ALX , BLX ; 2 8 ; 10 x2 ; 13 ;
inst INSTD2ABC [! ABX] DInst .D Nmb , ALX , BLX ; 3 9 ; 11 x2 ; 14 ;
inst INSTD3ABA DInst .D ALX , Nmb , BLX ; 3 ; 7 x2 ; ;
inst INSTD3ABB [ABX] DInst .D ALX , Nmb , BLX ; 2 6 ; 10 x2 ; 13 ;
inst INSTD3ABC [! ABX] DInst .D ALX , Nmb , BLX ; 3 7 ; 11 x2 ; 14 ;
inst INSTD4ABA DInst .D ALX , BLX , BLX ; 3 5 ; 7 x2 ; ;
inst INSTD4ABB [ABX] DInst .D ALX , BLX , BLX ; 2 6 8 ; 10 x2 ; 13 ;
inst INSTD4ABC [! ABX] DInst .D ALX , BLX , BLX ; 3 7 9 ; 11 x2 ; 14 ;
inst LOAD1AA LDX .D * +- AX [STRING] , ABLX ; 5 mem1 ; 10 #4 ; ;
inst LOAD1AB [ABX] LDX .D * +- AX [STRING] , ABLX ; 2 8 mem1 ; 13 #4 ; 13 ;
inst LOAD1AC [! ABX] LDX .D * +- AX [STRING] , ABLX ; 3 9 mem1 ; 14 #4 ; 14 ;
inst LOAD2AA LDX .D * ++– AX MOD , ABLX ; 5 mem1 ; 5 8 #4 ; ;
inst LOAD2AB [ABX] LDX .D * ++– AX MOD , ABLX ; 2 8 mem1 ; 8 11 #4 ; 13 ;
inst LOAD2AC [! ABX] LDX .D * ++– AX MOD , ABLX ; 3 9 mem1 ; 9 12 #4 ; 14 ;
inst LOAD3AA LDX .D * AX ++– MOD , ABLX ; 4 mem1 ; 4 8 #4 ; ;
inst LOAD3AB [ABX] LDX .D * AX ++– MOD , ABLX ; 2 7 mem1 ; 7 11 #4 ; 13 ;
inst LOAD3AC [! ABX] LDX .D * AX ++– MOD , ABLX ; 3 8 mem1 ; 8 12 #4 ; 14 ;
inst LOAD4AA LDX .D * +- AX [AX] , ABLX ; 5 7 mem1 ; 10 #4 ; ;
inst LOAD4AB [ABX] LDX .D * +- AX [AX] , ABLX ; 2 8 10 mem1 ; 13 #4 ; 13 ;
inst LOAD4AC [! ABX] LDX .D * +- AX [AX] , ABLX ; 3 9 11 mem1 ; 14 #4 ; 14 ;
inst LOAD5AA LDX .D * ++– AX [AX] , ABLX ; 5 7 mem1 ; 5 10 #4 ; ;
inst LOAD5AB [ABX] LDX .D * ++– AX [AX] , ABLX ; 2 8 10 mem1 ; 8 13 #4 ; 13 ;
inst LOAD5AC [! ABX] LDX .D * ++– AX [AX] , ABLX ; 3 9 11 mem1 ; 9 14 #4 ; 14 ;
inst LOAD6AA LDX .D * AX ++– [AX] , ABLX ; 4 7 mem1 ; 4 10 #4 ; ;
inst LOAD6AB [ABX] LDX .D * AX ++– [AX] , ABLX ; 2 7 10 mem1 ; 7 13 #4 ; 13 ;
inst LOAD6AC [! ABX] LDX .D * AX ++– [AX] , ABLX ; 3 8 11 mem1 ; 8 14 #4 ; 14 ;
inst LOAD7AA LDX .D * AX , ABLX ; 4 mem1 ; 6 #4 ; ;
inst LOAD7AB [ABX] LDX .D * AX , ABLX ; 2 7 mem1 ; 9 #4 ; 13 ;
inst LOAD7AC [! ABX] LDX .D * AX , ABLX ; 3 8 mem1 ; 10 #4 ; 14 ;
inst LOAD1BA LDX .D * +- BX [STRING] , ABLX ; 5 mem2 ; 10 #4 ; ;
inst LOAD1BB [ABX] LDX .D * +- BX [STRING] , ABLX ; 2 8 mem2 ; 13 #4 ; 13 ;
inst LOAD1BC [! ABX] LDX .D * +- BX [STRING] , ABLX ; 3 9 mem2 ; 14 #4 ; 14 ;
inst LOAD2BA LDX .D * ++– BX MOD , ABLX ; 5 mem2 ; 5 8 #4 ; ;
inst LOAD2BB [ABX] LDX .D * ++– BX MOD , ABLX ; 2 8 mem2 ; 8 11 #4 ; 13 ;
inst LOAD2BC [! ABX] LDX .D * ++– BX MOD , ABLX ; 3 9 mem2 ; 9 12 #4 ; 14 ;
inst LOAD3BA LDX .D * BX ++– MOD , ABLX ; 4 mem2 ; 4 8 #4 ; ;
inst LOAD3BB [ABX] LDX .D * BX ++– MOD , ABLX ; 2 7 mem2 ; 7 11 #4 ; 13 ;
inst LOAD3BC [! ABX] LDX .D * BX ++– MOD , ABLX ; 3 8 mem2 ; 8 12 #4 ; 14 ;
inst LOAD4BA LDX .D * +- BX [BX] , ABLX ; 5 7 mem2 ; 10 #4 ; ;
inst LOAD4BB [ABX] LDX .D * +- BX [BX] , ABLX ; 2 8 10 mem2 ; 13 #4 ; 13 ;
inst LOAD4BC [! ABX] LDX .D * +- BX [BX] , ABLX ; 3 9 11 mem2 ; 14 #4 ; 14 ;

inst LOAD5BA LDX .D * ++– BX [BX] , ABLX ; 5 7 mem2 ; 5 10 #4 ; ;
inst LOAD5BB [ABX] LDX .D * ++– BX [BX] , ABLX ; 2 8 10 mem2 ; 8 13 #4 ; 13 ;
inst LOAD5BC [! ABX] LDX .D * ++– BX [BX] , ABLX ; 3 9 11 mem2 ; 9 14 #4 ; 14 ;
inst LOAD6BA LDX .D * BX ++– [BX] , ABLX ; 4 7 mem2 ; 4 10 #4 ; ;
inst LOAD6BB [ABX] LDX .D * BX ++– [BX] , ABLX ; 2 7 10 mem2 ; 7 13 #4 ; 13 ;
inst LOAD6BC [! ABX] LDX .D * BX ++– [BX] , ABLX ; 3 8 11 mem2 ; 8 14 #4 ; 14 ;
inst LOAD7BA LDX .D * BX , ABLX ; 4 mem ; 6 #4 ; ;
inst LOAD7BB [ABX] LDX .D * BX , ABLX ; 2 7 mem ; 9 #4 ; 13 ;
inst LOAD7BC [! ABX] LDX .D * BX , ABLX ; 3 8 mem ; 10 #4 ; 14 ;
inst LOAD8BA LDXL .D * B145 [STRING] , ABLX ; 4 mem2 ; 9 #4 ; ;
inst LOAD8BB [ABX] LDXL .D * B145 [STRING] , ABLX ; 2 7 mem2 ; 12 #4 ; 13 ;
inst LOAD8BC [! ABX] LDXL .D * B145 [STRING] , ABLX ; 3 8 mem2 ; 13 #4 ; 14 ;
inst STR1AA STX .D ABLX , * +- AX [STRING] ; 3 7 ; mem1 ; ;
inst STR1AB [ABX] STX .D ABLX , * +- AX [STRING] ; 2 6 10 ; mem1 ; 13 ;
inst STR1AC [! ABX] STX .D ABLX , * +- AX [STRING] ; 3 7 11 ; mem1 ; 14 ;
inst STR2AA STX .D ABLX , * ++– AX MOD ; 3 7 ; 7 mem1 ; ;
inst STR2AB [ABX] STX .D ABLX , * ++– AX MOD ; 2 6 10 ; 10 mem1 ; 13 ;
inst STR2AC [! ABX] STX .D ABLX , * ++– AX MOD ; 3 7 11 ; 11 mem1 ; 14 ;
inst STR3AA STX .D ABLX , * AX ++– MOD ; 3 6 ; 6 mem1 ; ;
inst STR3AB [ABX] STX .D ABLX , * AX ++– MOD ; 2 6 9 ; 9 mem1 ; 13 ;
inst STR3AC [! ABX] STX .D ABLX , * AX ++– MOD ; 3 7 10 ; 10 mem1 ; 14 ;
inst STR4AA STX .D ABLX , * +- AX [AX] ; 3 7 9 ; mem1 ; ;
inst STR4AB [ABX] STX .D ABLX , * +- AX [AX] ; 2 6 10 12 ; mem1 ; 13 ;
inst STR4AC [! ABX] STX .D ABLX , * +- AX [AX] ; 3 7 11 13 ; mem1 ; 14 ;
inst STR5AA STX .D ABLX , * ++– AX [AX] ; 3 7 9 ; 7 mem1 ; ;
inst STR5AB [ABX] STX .D ABLX , * ++– AX [AX] ; 2 6 10 12 ; 10 mem1 ; 13 ;
inst STR5AC [! ABX] STX .D ABLX , * ++– AX [AX] ; 3 7 11 13 ; 11 mem1 ; 14 ;
inst STR6AA STX .D ABLX , * AX ++– [AX] ; 3 6 9 ; 6 mem1 ; ;
inst STR6AB [ABX] STX .D ABLX , * AX ++– [AX] ; 2 6 9 12 ; 9 mem1 ; 13 ;
inst STR6AC [! ABX] STX .D ABLX , * AX ++– [AX] ; 3 7 10 13 ; 10 mem1 ; 14 ;
inst STR7AA STX .D ABLX , * AX ; 3 6 ; mem ; ;
inst STR7AB [ABX] STX .D ABLX , * AX ; 2 6 9 ; mem ; 13 ;
inst STR7AC [! ABX] STX .D ABLX , * AX ; 3 7 10 ; mem ; 14 ;
inst STR1BA STX .D ABLX , * +- BX [STRING] ; 3 7 ; mem2 ; ;
inst STR1BB [ABX] STX .D ABLX , * +- BX [STRING] ; 2 6 10 ; mem2 ; 13 ;
inst STR1BC [! ABX] STX .D ABLX , * +- BX [STRING] ; 3 7 11 ; mem2 ; 14 ;
inst STR2BA STX .D ABLX , * ++– BX MOD ; 3 7 ; 7 mem2 ; ;
inst STR2BB [ABX] STX .D ABLX , * ++– BX MOD ; 2 6 10 ; 10 mem2 ; 13 ;
inst STR2BC [! ABX] STX .D ABLX , * ++– BX MOD ; 3 7 11 ; 11 mem2 ; 14 ;
inst STR3BA STX .D ABLX , * BX ++– MOD ; 3 6 ; 6 mem2 ; ;
inst STR3BB [ABX] STX .D ABLX , * BX ++– MOD ; 2 6 9 ; 9 mem2 ; 13 ;
inst STR3BC [! ABX] STX .D ABLX , * BX ++– MOD ; 3 7 10 ; 10 mem2 ; 14 ;
inst STR4BA STX .D ABLX , * +- BX [BX] ; 3 7 9 ; mem2 ; ;
inst STR4BB [ABX] STX .D ABLX , * +- BX [BX] ; 2 6 10 12 ; mem2 ; 13 ;
inst STR4BC [! ABX] STX .D ABLX , * +- BX [BX] ; 3 7 11 13 ; mem2 ; 14 ;
inst STR5BA STX .D ABLX , * ++– BX [BX] ; 3 7 9 ; 7 mem2 ; ;
inst STR5BB [ABX] STX .D ABLX , * ++– BX [BX] ; 2 6 10 12 ; 10 mem2 ; 13 ;

inst STR5BC [! ABX] STX .D ABLX , * ++– BX [BX] ; 3 7 11 13 ; 11 mem2 ; 14 ;
inst STR6BA STX .D ABLX , * BX ++– [BX] ; 3 6 9 ; 6 mem2 ; ;
inst STR6BB [ABX] STX .D ABLX , * BX ++– [BX] ; 2 6 9 12 ; 9 mem2 ; 13 ;
inst STR6BC [! ABX] STX .D ABLX , * BX ++– [BX] ; 3 7 10 13 ; 10 mem2 ; 14 ;
inst STR7BA STX .D ABLX , * BX ; 3 6 ; mem2 ; ;
inst STR7BB [ABX] STX .D ABLX , * BX ; 2 6 9 ; mem2 ; 13 ;
inst STR7BC [! ABX] STX .D ABLX , * BX ; 3 7 10 ; mem2 ; 14 ;
inst STR8BA STXL .D ABXL , * B145 [STRING] ; 3 6 ; mem2 ; ;
inst STR8BB [ABX] STXL .D ABXL , * B145 [STRING] ; 2 6 9 ; mem2 ; 13 ;
inst STR8BC [! ABX] STXL .D ABXL , * B145 [STRING] ; 3 7 10 ; mem2 ; 14 ;
inst DMV1A MV .D AX , AX ; 3 ; 5 ; ;
inst DMV1B [ABX] MV .D AX , AX ; 2 6 ; 8 ; 13 ;
inst DMV1C [! ABX] MV .D AX , AX ; 3 7 ; 9 ; 14 ;
inst DMV2A MV .D BX , BX ; 3 ; 5 ; ;
inst DMV2B [ABX] MV .D BX , BX ; 2 6 ; 8 ; 13 ;
inst DMV2C [! ABX] MV .D BX , BX ; 3 7 ; 9 ; 14 ;
inst DMV1XA MV .D BX , AX ; 3 ; 5 x1 ; ;
inst DMV1XB [ABX] MV .D BX , AX ; 2 6 ; 8 x1 ; 13 ;
inst DMV1XC [! ABX] MV .D BX , AX ; 3 7 ; 9 x1 ; 14 ;
inst DMV2XA MV .D AX , BX ; 3 ; 5 x2 ; ;
inst DMV2XB [ABX] MV .D AX , BX ; 2 6 ; 8 x2 ; 13 ;
inst DMV2XC [! ABX] MV .D AX , BX ; 3 7 ; 9 x2 ; 14 ;
inst MPY1AAA MPYX .M ALX , ALX , ALX ; 3 5 ; 7 #1 ; ;
inst MPY1AAB [ABX] MPYX .M ALX , ALX , ALX ; 2 6 8 ; 10 #1 ; 13 ;
inst MPY1AAC [! ABX] MPYX .M ALX , ALX , ALX ; 3 7 9 ; 11 #1 ; 14 ;
inst MPY2AAA MPYX .M Nmb , ALX , ALX ; 5 ; 7 #1 ; ;
inst MPY2AAB [ABX] MPYX .M Nmb , ALX , ALX ; 2 8 ; 10 #1 ; 13 ;
inst MPY2AAC [! ABX] MPYX .M Nmb , ALX , ALX ; 3 9 ; 11 #1 ; 14 ;
inst MPY1BAA MPYX .M ALX , BLX , ALX ; 3 5 ; 7 #1 x1 ; ;
inst MPY1BAB [ABX] MPYX .M ALX , BLX , ALX ; 2 6 8 ; 10 #1 x1 ; 13 ;
inst MPY1BAC [! ABX] MPYX .M ALX , BLX , ALX ; 3 7 9 ; 11 #1 x1 ; 14 ;
inst MPY2BAA MPYX .M Nmb , BLX , ALX ; 5 ; 7 #1 x1 ; ;
inst MPY2BAB [ABX] MPYX .M Nmb , BLX , ALX ; 2 8 ; 10 #1 x1 ; 13 ;
inst MPY2BAC [! ABX] MPYX .M Nmb , BLX , ALX ; 3 9 ; 11 #1 x1 ; 14 ;
inst MPY3BAA MPYX .M BLX , ALX , ALX ; 3 5 ; 7 #1 x1 ; ;
inst MPY3BAB [ABX] MPYX .M BLX , ALX , ALX ; 2 6 8 ; 10 #1 x1 ; 13 ;
inst MPY3BAC [! ABX] MPYX .M BLX , ALX , ALX ; 3 7 9 ; 11 #1 x1 ; 14 ;
inst MPY1BBA MPYX .M BLX , BLX , BLX ; 3 5 ; 7 #1 ; ;
inst MPY1BBB [ABX] MPYX .M BLX , BLX , BLX ; 2 6 8 ; 10 #1 ; 13 ;
inst MPY1BBC [! ABX] MPYX .M BLX , BLX , BLX ; 3 7 9 ; 11 #1 ; 14 ;
inst MPY2BBA MPYX .M Nmb , BLX , BLX ; 5 ; 7 #1 ; ;
inst MPY2BBB [ABX] MPYX .M Nmb , BLX , BLX ; 2 8 ; 10 #1 ; 13 ;
inst MPY2BBC [! ABX] MPYX .M Nmb , BLX , BLX ; 3 9 ; 11 #1 ; 14 ;
inst MPY1ABA MPYX .M BLX , ALX , BLX ; 3 5 ; 7 #1 x2 ; ;
inst MPY1ABB [ABX] MPYX .M BLX , ALX , BLX ; 2 6 8 ; 10 #1 x2 ; 13 ;
inst MPY1ABC [! ABX] MPYX .M BLX , ALX , BLX ; 3 7 9 ; 11 #1 x2 ; 14 ;
inst MPY2ABA MPYX .M Nmb , ALX , BLX ; 5 ; 7 #1 x2 ; ;

inst MPY2ABB [ABX] MPYX .M Nmb , ALX , BLX ; 2 8 ; 10 #1 x2 ; 13 ;
inst MPY2ABC [! ABX] MPYX .M Nmb , ALX , BLX ; 3 9 ; 11 #1 x2 ; 14 ;
inst MPY3ABA MPYX .M ALX , BLX , BLX ; 3 5 ; 7 #1 x2 ; ;
inst MPY3ABB [ABX] MPYX .M ALX , BLX , BLX ; 2 6 8 ; 10 #1 x2 ; 13 ;
inst MPY3ABC [! ABX] MPYX .M ALX , BLX , BLX ; 3 7 9 ; 11 #1 x2 ; 14 ;

ISet L1 LZERO1A LZERO1B LZERO1C ABS1A ABS1B ABS1C ABS1XA ABS1XB ABS1XC INSTL1AAA INSTL1AAB INSTL1AAC INSTL2AAA INSTL2AAB INSTL2AAC INSTL3AAA INSTL3AAB INSTL3AAC INSTL1BAA INSTL1BAB INSTL1BAC INSTL2BAA INSTL2BAB INSTL2BAC INSTL3BAA INSTL3BAB INSTL3BAC INSTL4BAA INSTL4BAB INSTL4BAC ;

ISet L2 LZERO2A LZERO2B LZERO2C ABS2A ABS2B ABS2C ABS2XA ABS2XB ABS2XC INSTL1BBA INSTL1BBB INSTL1BBC INSTL2BBA INSTL2BBB INSTL2BBC INSTL3BBA INSTL3BBB INSTL3BBC INSTL1ABA INSTL1ABB INSTL1ABC INSTL2ABA INSTL2ABB INSTL2ABC INSTL3ABA INSTL3ABB INSTL3ABC INSTL4ABA INSTL4ABB INSTL4ABC ;

ISet S1 SZERO1A SZERO1B SZERO1C INSTS1AAA INSTS1AAB INSTS1AAC INSTS2AAA INSTS2AAB INSTS2AAC INSTS3AAA INSTS3AAB INSTS3AAC INSTS1BAA INSTS1BAB INSTS1BAC INSTS2BAA INSTS2BAB INSTS2BAC INSTS3BAA INSTS3BAB INSTS3BAC INSTS4BAA INSTS4BAB INSTS4BAC BRANCH1A BRANCH1B BRANCH1C ECLRAA ECLRAB ECLRAC ECLRAXA ECLRAXB ECLRAXC MVKAA MVKAB MVKAC UNI1A UNI1B UNI1C UNI1XA UNI1XB UNI1XC ;

ISet S2 SZERO2A SZERO2B SZERO2C INSTS1BBA INSTS1BBB INSTS1BBC INSTS2BBA INSTS2BBB INSTS2BBC INSTS3BBA INSTS3BBB INSTS3BBC INSTS1ABA INSTS1ABB INSTS1ABC INSTS2ABA INSTS2ABB INSTS2ABC INSTS3ABA INSTS3ABB INSTS3ABC INSTS4ABA INSTS4ABB INSTS4ABC BRANCH2A BRANCH2B BRANCH2C BRANCH3A BRANCH3B BRANCH3C ECLRBA ECLRBB ECLRBC ECLRBXA ECLRBXB ECLRBXC MVKBA MVKBB MVKBC UNI2A UNI2B UNI2C UNI2XA UNI2XB UNI2XC BRANCHDA BRANCHDB BRANCHDC ;

ISet D1 DMV1A DMV1B DMV1C DMV1XA DMV1XB DMV1XC DZERO1A DZERO1B DZERO1C INSTD1AAA INSTD1AAB INSTD1AAC INSTD2AAA INSTD2AAB INSTD2AAC INSTD3AAA INSTD3AAB INSTD3AAC INSTD1BAA INSTD1BAB INSTD1BAC INSTD2BAA INSTD2BAB INSTD2BAC INSTD3BAA INSTD3BAB INSTD3BAC INSTD4BAA INSTD4BAB INSTD4BAC LOAD1AA LOAD1AB LOAD1AC LOAD2AA LOAD2AB LOAD2AC LOAD3AA LOAD3AB LOAD3AC LOAD4AA LOAD4AB LOAD4AC LOAD5AA LOAD5AB LOAD5AC LOAD6AA LOAD6AB LOAD6AC LOAD7AA LOAD7AB LOAD7AC STR1AA STR1AB STR1AC STR2AA STR2AB STR2AC STR3AA STR3AB STR3AC STR4AA STR4AB STR4AC STR5AA STR5AB STR5AC STR6AA STR6AB STR6AC STR7AA STR7AB STR7AC ;

ISet D2 DMV2A DMV2B DMV2C DMV2XA DMV2XB DMV2XC DZERO2A DZERO2B DZERO2C INSTD1BBA INSTD1BBB INSTD1BBC INSTD2BBA INSTD2BBB INSTD2BBC INSTD3BBA INSTD3BBB INSTD3BBC INSTD1ABA INSTD1ABB INSTD1ABC IN-

STD2ABA INSTD2ABB INSTD2ABC INSTD3ABA INSTD3ABB INSTD3ABC INSTD4ABA INSTD4ABB INSTD4ABC LOAD1BA LOAD1BB LOAD1BC LOAD2BA LOAD2BB LOAD2BC LOAD3BA LOAD3BB LOAD3BC LOAD4BA LOAD4BB LOAD4BC LOAD5BA LOAD5BB LOAD5BC LOAD6BA LOAD6BB LOAD6BC LOAD7BA LOAD7BB LOAD7BC LOAD8BA LOAD8BB LOAD8BC STR1BA STR1BB STR1BC STR2BA STR2BB STR2BC STR3BA STR3BB STR3BC STR4BA STR4BB STR4BC STR5BA STR5BB STR5BC STR6BA STR6BB STR6BC STR7BA STR7BB STR7BC STR8BA STR8BB STR8BC ;

ISet M1 MPY1AAA MPY1AAB MPY1AAC MPY2AAA MPY2AAB MPY2AAC MPY1BAA MPY1BAB MPY1BAC MPY2BAA MPY2BAB MPY2BAC MPY3BAA MPY3BAB MPY3BAC ;

ISet M2 MPY1BBA MPY1BBB MPY1BBC MPY2BBA MPY2BBB MPY2BBC MPY1ABA MPY1ABB MPY1ABC MPY2ABA MPY2ABB MPY2ABC MPY3ABA MPY3ABB MPY3ABC ;

Combo L1 S1 D1 M1 L2 S2 D2 M2 ;

Parts A15:A14 A15 A14 ;
Parts A13:A12 A13 A12 ;
Parts A11:A10 A11 A10 ;
Parts A9:A8 A9 A8 ;
Parts A7:A6 A7 A6 ;
Parts A5:A4 A5 A4 ;
Parts A3:A2 A3 A2 ;
Parts A1:A0 A1 A0 ;

Bibliography

[1] Scalable instruction-level parallelism through tree-instructions
J.H. Moreno, M. Moudgill; 1997 International Conference on Supercomputing, Vienna, Austria, July 7-11, 1997, pp. 1-11.

[2] Performance analysis of a tree VLIW architecture for exploiting branch ILP in non-numerical code
S.M. Moon, K. Ebcioglu; 1997 International Conference on Supercomputing, Vienna, Austria, July 7-11, 1997, pp. 301-308.

[3] VLIW Compilation Techniques in a Superscalar Environment
K. Ebcioglu, R. Groves, K.C. Kim, G. Silbermam, I. Ziv; ACM SIGPLAN Notices, vol. 29, no. 6, pp. 36-48, June 1994 (PLDI'94).

[4] Making Compaction Based Parallelization Affordable
T. Nakatani, K. Ebcioglu; IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 9, pp. 1014-1029, September 1993.

[5] On Performance and Efficiency of VLIW and Superscalar
S.M. Moon, K. Ebcioglu; in International Conference on Parallel Processing, vol. 2, pp. 283-287, 1993, CRC Press, Ann Arbor.

[6] An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors
S.M. Moon, K. Ebcioglu; in Proceeding of MICRO-25, pp. 55-71, IEEE Press, December 1992.

[7] Using a Lookahead Window in a Compaction based Parallelizing Compiler
T. Nakatani, K. Ebcioglu; in Proceedings of MICRO-23, pp. 57-68, IEEE Press, 1990.

[8] A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture
K. Ebcioglu, T. Nakatani; in Languages and Compilers for Parallel Computing, D. Gelernter, A. Nicolau, D. Padua (eds.), Research Monographs in Parallel and Distributed Computing, pp. 213-229, MIT Press, 1990.

[9] Combining as a Compilation Technique for VLIW Architectures
T. Nakatani, K. Ebcioglu; in Proceedings of MICRO-22, pp. 43-57, ACM Press, 1989.

[10] A Global Resource-constrained Parallelization Technique
K. Ebcioglu, A. Nicolau; in Proceedings Third International Conference on Supercomputing, pp. 154-163, Crete, June 1989.

[11] AVIV: A Retargetable Code Generator for Embedded Processors
Silvina Zimi Hanono; MIT PhD Thesis, to be published June'99.

[12] Code Generation Using Tree Matching and Dynamic Programming
A. Aho, M. Ganapathi, S. Tjang; ACM Transactions on Programming Languages and Systems 11(4):491-516, 1989

[13] Introduction to Algorithms
T. Cormen, C. Leiserson, R. Rivest; The MIT Press, 1990

[14] Chess: Retargetable Code Generation for Embedded DSP Processors
D. Laneer et al; In Code Generation for Embedded Processors, Kluwer Academic Publishers

[15] Flexware: A Flexible Firmware Development Environment for Embedded Systems
P. Paulin et al; In Code Generation for Embedded Processors, Kluwer Academic Publishers, 1995

[16] An ILP-Based Approach to Code Generation
Wilson et al; In Code Generation for Embedded Processors, pp. 103-118, Kluwer Academic Publishers, 1995

[17] The nML Machine Description Formalism
M. Freericks; Technical Report, Technical University, Berlin, 1993

[18] Hardware-Software Cosynthesis for Digital Systems
R. K. Gupta and G. De Micheli; IEEE Design and Test of Computers, pages 29-41, September 1993

[19] ISDL: An Instruction Set Description Language for Retargetability
G. Hadjiyiannis, S. Hanono, S. Devadas; In Proceedingsa of the 34th Design Automation Conference, pages 299-302, June 1997

[20] Instruction Selection, Resource Allocation, and Scheduling in the AVIA Retargetable Code Generator
S. Hanono, S. Devadas; In Proceedings of 35th Design Automation Conference, pages 510-515, June, 1998

[21] Instruction Set Matching and Selection for DSP and ASIP Code Generation
C. Liem, T. May, P. Paulin; In Proceedings of the European Design and Test Conference, pages 31-37, February, 1994

[22] The MiMOLA Design System: Tools for the Design of Digital Processors
P. Marwedel; In Proceedings of the 21st Design Automation Conference, pages 587-593, 1984

[23] CodeSyn: A Retargetable Code Synthesis System
P. G. Paulin, C. Liem, T. C. May, S. Sutarwala; In Proceedings of the 7th International High-Level Synthesis Workshop, Spring 1994

[24] TMS320C6000 CPU and Instruction Set Reference Guide
Texas Instruments; literature number: SPRU189D,
http://www-s.ti.com/sc/psheets/spru189d/spru189d.pdf, February, 1999

[25] TMS320C62X / C67X Programmer's Guide
Texas Instruments; literature number: SPRU198B,
http://www-s.ti.com/sc/psheets/spru198b/spru198b.pdf, 1998

[26] TMS320C6000 Assembly Language Tools User's Guide
Texas Instruments; literature number: SPRU186E,
http://www-s.ti.com/sc/psheets/spru186e/spru186e.pdf, February, 1999

[27] TMS320C6000 Optimizing C Compiler User's Guide
Texas Instruments; literature number: SPRU187E,
http://www-s.ti.com/sc/psheets/spru187e/spru187e.pdf, February, 1999